

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9510644

Distributed Green's function Monte Carlo calculations

Chen, Jian, Ph.D.

City University of New York, 1994

Copyright ©1994 by Chen, Jian. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

DISTRIBUTED GREEN'S FUNCTION MONTE CARLO CALCULATIONS

by

JIAN CHEN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1994

© 1994

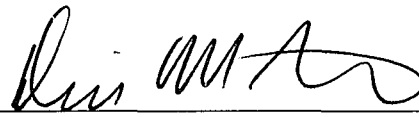
JIAN CHEN

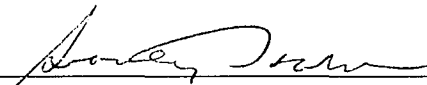
ALL Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

7/19/94
Date

Aug 30, 1994
Date


Chair of Examining Committee


Executive Officer

Ora Percus

Gerald Weiss

Paula Whitlock

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

DISTRIBUTED GREEN'S FUNCTION
MONTE CARLO CALCULATIONS

by

Jian Chen

Adviser: Professor David Arnow

A useful and interesting problem is to solve a linear partial differential equation using the Monte Carlo method. For example, the only general method for obtaining exact solutions to the Schrödinger equation is the Green's function Monte Carlo method. The fermion Green's function Monte Carlo requires computations on multi processors since it demands on memory and CPU power. Parallel simulations, like distributed Monte Carlo, have their unique characteristics which do not exist in sequential Monte Carlo computations. For instance, an distributed estimator is needed. Load balancing has to be taken into account to achieve better computational performance. There is no guarantee that correlations will not occur in such a distributed computation if subsequences of a single pseudorandom sequence is used by individual worker processes. Therefore, implementing a distributed random number generator system to support these simulations is essential. A procedure that integrates all above was developed to carry out distributed Green's function Monte Carlo calculations. A distributed message passing system, DP, that is suitable for large scale scientific computations is employed to carry out data transmission and process synchronization of the distributed computation.

Acknowledgements

I thank Professor David Arnow. Without his support and help, this dissertation would not be possible. The topic was well-specified by Professor Arnow soon after our adviser/student relationship was established so that I had adequate time to develop the subjects. Many times when I encountered technique problems, he was always willing to provide ideas and possible solutions through our discussion. Because of the convenience that he provided to me, his office, desks and facilities in the room, I survived in this program in which teaching and researching were my responsibilities for all the years. Furthermore, the advantages I took were that he outlined the dissertation and major topics that had to be covered. He was always taking care of my progress even when he was in conference trips and in his summer and winter vacations. He proofread all the written material and made corrections. The design goals and his advising were consistent so that I could follow it even though sometimes it was difficult.

I particularly thank Professor Paula Whitlock for advising me to develop the distributed random number generators. She taught me physics and simulation that are relate to my research topics. In the last year of this long program, she patiently read the entire manuscript, twice for some of the chapters. She provided enormous suggestions and corrections, many of them spelling mistakes and grammatical errors.

I also thank Professor Ora Percus for providing the details of her work from which the generator system was developed.

I thank Professor Gerald Weiss and Professor Dayton Clark for their help of distributed computing. Many thanks go to my colleagues, Haibin Jiu and Ajit Jaiswal, for all useful conversations about physics and writing skills. Our system administrator Bill Goodridge provided prompt help whenever I needed, thank you Bill.

I especially thank Professor Aaron Tenenbaum and the Department of CIS for their financial support.

Finally, I thank all my relatives, especially my aunt Helena and Rose for their understanding and a great encouragement.

Table of Contents

Title page	i
Copyright page	ii
Approval page	iii
Abstract	iv
Acknowledgments	v
Table of contents	vi
List of figures	xi
List of code fragments	xiii
List of tables	xvii
1 The problem	1
1.1 Distributed computation	2
1.2 The problem of physics	3
1.3 Monte Carlo solution	6
1.4 Portable distributed programming	7
1.5 Problem of parallel random number generations	8
1.6 The results	9
2 Background	10
2.1 Quantum physics	12
2.1.1 The Schrödinger equation	13
2.1.2 Antisymmetry	15
2.1.3 The fermion problem	17

2.1.3.1	Nine-dimensional Euclidean space	17
2.1.3.2	The model problem	18
2.1.3.3	The analytical solution	20
2.2	The Monte Carlo method	32
2.2.1	Estimating definite integrals	33
2.2.2	Variance and the estimation of error	35
2.2.3	Sampling techniques	37
2.2.3.1	The rejection technique	39
2.2.3.2	Sampling by random walk - the Metropolis method	40
2.2.3.3	Importance sampling	42
2.3	Green's function	43
2.4	Distributed message systems	48
2.4.1	The notion of parallelism in this work	48
2.4.2	Distributed Process system, DP	50
2.4.3	The interface of DP	53
2.4.3.1	DP primitives	53
2.4.3.2	Standard DP functions	56
3	Implementing the Distributed Process system	58
3.1	Small DP systems	58
3.2	Concurrent distributed process system	60
3.2.1	System limitations	63
3.2.2	Interprocess Communication resource allocation	65

3.2.3 Queue system	74
3.2.4 Mechanism of mutual exclusion and concurrency	81
3.2.5 System V-like queue primitives	95
3.2.6 DP primitives	103
4 Distributed Pseudorandom Number Generators	113
4.1 Distributed Monte Carlo calculation	113
4.2 Linear congruential random number generators	115
4.3 The Spectral tests	118
4.3.1 The Knuth's algorithm	119
4.3.2 The Percus and Kalos' algorithm	123
4.4 Implementing Spectral Tests	127
4.4.1 New abstract data types	128
4.4.2 A short <i>C++</i> program	130
4.4.3 Implementing the Knuth' Spectral test	134
4.4.4 Implementing the Percus & Kalos' Spectral test	146
4.5 Searching good additive constants	153
4.6 Managing of parallel generators	155
4.6.1 The distributed random number generator package	159
4.6.2 Drand primitive functions	161
4.6.3 Paged binary tree	163
4.6.4 Remote procedure calls	168
4.6.5 System server	172

4.6.6 Examples of using DRAND	177
4.6.6.1 Example: user interface functions	178
4.6.6.2 Example: use DRAND programs	181
4.6.7 Critical issues of implementing DRAND	186
4.6.7.1 Parameters of RPC procedures	186
4.6.7.2 Parameter filters	188
4.6.7.3 Communication set up	191
4.6.7.4 RPC procedure calls	194
4.6.7.5 Return to user's programs	198
4.6.7.6 Faults tolerance	199
4.6.7.7 Handling died clients	201
4.6.7.8 Other issues	202
4.7 The abstract data types	203
4.7.1 Representing operands	204
4.7.2 The division method	206
5 The Distributed Monte Carlo – One-dimensional case	210
5.1 One dimensional quantum problem	213
5.2 GFMC in one dimension: The iteration	214
5.3 GFMC in one dimension: Estimate the lowest eigenvalue	216
5.4 GFMC in one dimension: Sampling the Green's function	218
5.5 Distributed 1-D GFMC: Overall and initialization	220
5.6 Distributed 1-D GFMC: Messages	224

5.7 Distributed 1-D GFMC: Estimating energy and balancing work load	233
6 Distributing Fermion Green's Function Monte Carlo	242
6.1 The Green's function of many particles	242
6.2 Estimate the lowest antisymmetric state energy	246
6.3 The Green's function for fermion systems	247
6.4. Distribute FGFMC on DP system	250
6.4.1 The representation of two populations	250
6.4.2 Distributing sampled populations	256
6.4.3 Searching pairs in distributed memory	263
6.4.4 Distributing FGFMC	264
6.4.5* The FGFMC program	283
7 Conclusion	300
Appendices	303
References	337

List of Figures

Figure 2.1 The potential of the model problem	19
Figure 2.2 Approximating energy	31
Figure 2.3 Energy levels	32
Figure 3.1 The concurrent distributed process	62
Figure 3.2 A memory chunk	67
Figure 3.3 Chained memory chunks	68
Figure 3.4 Memory shared by processes	71
Figure 3.5 Message structures	77
Figure 3.6 Message queues	77
Figure 4.1 Two pseudorandom number generators	118
Figure 4.2 Two abstract data types	131
Figure 4.3 Frederickson random tree	156
Figure 4.4 Percus and Kalos random tree	158
Figure 4.5 Two abstract operations	163
Figure 4.6 Paged complete binary trees	166
Figure 4.7 Three application operations	178
Figure 4.8 Long integer division	209
Figure 5.1 Potential of the model problem	214
Figure 5.2 Sampling the Green's function at x	218
Figure 5.3 Sampling by a random walk at x	219
Figure 5.4 Sampling the Green's function in parallel	220

Figure 6.1 Hash system	251
Figure 6.2 The pairing scheme	255
Figure 6.3 The structure of the FGPMC program	284

List of Code fragments

Code fragment 2.1	34
Code fragment 3.1	66
Code fragment 3.2	69
Code fragment 3.3	70
Code fragment 3.4	72
Code fragment 3.5	76
Code fragment 3.6	78
Code fragment 3.7	79
Code fragment 3.8	80
Code fragment 3.9	85
Code fragment 3.10	86
Code fragment 3.11	88
Code fragment 3.12	89
Code fragment 3.13	91
Code fragment 3.14	93
Code fragment 3.15	93
Code fragment 3.16	94
Code fragment 3.17	95
Code fragment 3.18	97
Code fragment 3.19	100
Code fragment 3.20	101

Code fragment 3.21	104
Code fragment 3.22	106
Code fragment 3.23	107
Code fragment 3.24	109
Code fragment 3.25	109
Code fragment 3.26	110
Code fragment 3.27	111
Code fragment 3.28	111
Code fragment 4.1	132
Code fragment 4.2	137
Code fragment 4.3	138
Code fragment 4.4	139
Code fragment 4.5	140
Code fragment 4.6	141
Code fragment 4.7	142
Code fragment 4.8	144
Code fragment 4.9	147
Code fragment 4.10	148
Code fragment 4.11	150
Code fragment 4.12	161
Code fragment 4.13	179
Code fragment 4.14	179

Code fragment 4.15	187
Code fragment 4.16	189
Code fragment 4.17	189
Code fragment 4.18	191
Code fragment 4.19	193
Code fragment 4.20	194
Code fragment 4.21	196
Code fragment 4.22	198
Code fragment 4.23	201
Code fragment 4.24	202
Code fragment 5.1	221
Code fragment 5.2	222
Code fragment 5.3	223
Code fragment 5.4	224
Code fragment 5.5	226
Code fragment 5.6	227
Code fragment 5.7	228
Code fragment 5.8	230
Code fragment 5.9	231
Code fragment 5.10	232
Code fragment 6.1	285
Code fragment 6.2	286

Code fragment 6.3	287
Code fragment 6.4	288
Code fragment 6.5	289
Code fragment 6.6	290
Code fragment 6.7	291
Code fragment 6.8	291
Code fragment 6.9	292
Code fragment 6.10	293
Code fragment 6.11	294
Code fragment 6.12	295
Code fragment 6.13	296
Code fragment 6.14	297
Code fragment 6.15	298
Code fragment 6.16	298

List of tables

Table 5.1 Work load unbalanced	238
Table 5.2 Work load balanced	239
Table 5.3 The variance of the computation	241
Table 6.1 (P0)Points loaded in P0	259
Table 6.1 (P1)Points loaded in P1	260
Table 6.1 (P2)Points loaded in P2	261
Table 6.1 (P3)Points loaded in P3	262
Table 6.2 Distributed hash tables	267
Table 6.3 (P0)Neighbors that hit other process' areas	272
Table 6.3 (P1)Neighbors that hit other process' areas	273
Table 6.3 (P2)Neighbors that hit other process' areas	274
Table 6.3 (P3)Neighbors that hit other process' areas	275
Table 6.4 The number of unblocked messages	276
Table 6.5 The number of local points	280
Table 6.6 The fermion energy (1)	281
Table 6.6 The fermion energy (2)	282

1

The problem

This work is an integration of several disciplines, distributed computing, Monte Carlo methods and parallel random number generation. All of them are applied here to a problem in computational physics.

One of the scientific applications that requires massively large-scale computation is Monte Carlo simulation to solve many-dimensional linear partial differential equations. Such applications need not only high-speed processors to process data, but large quantities of primary memory as well to hold intermediate results. In the past, most Monte Carlo calculations were carried out sequentially. Therefore, the computations were limited by available processor speed and memory. Today, powerful computer systems, like supercomputers and computer networks, are widely available and allow us to conduct Monte Carlo simulations in parallel.

In this work, a nine-dimensional linear partial differential equation is studied and the lowest fermion state energy is obtained by employing the Monte Carlo method. The simulation is carried out on a system of heterogeneous computers and other parallel computer systems. Computations of this kind on a computer network are usually called distributed computing. The problem that limited memory was installed in the “old days” (early 1980s) is no longer a problem because the memory installed on the hosts on a network can be used together in a single computation. The full utilization of the processors’ power to carry out these computations depends on the demand posed by the data communication tasks. The utilization of

processors for such computation may be high if communication between processors is minimized.

1.1 Distributed computation

Our solution to overcome the limitations of processor speed and low memory is to distribute Monte Carlo computations on a local area network (LAN). An integral that is evaluated using the Monte Carlo method can be naturally divided into smaller regions of integration that may be carried out by distributed processes, each of which may be running on a different network host. The difference between distributed and sequential computation is that in distributed computing data exchange and synchronization are needed. The new problem becomes how to minimize the amount of data transferred between processes during computations in order to reduce traffic on the network and effectively utilize processors' power.

The problem we compute requires primarily a large quantity of memory to store intermediate data. In this computation the more available memory employed, the better the results that will be obtained. High-speed processors are obviously welcome since a huge amount of data needs to be processed. There are some major concerns of distributed computing. In the problem in our hands, minimizing message traffic is primary, which is not only to reduce costs, but the way to keep the computation going as well. Managing average computational work load on each involved host is another. Computational, the Monte Carlo calculation can be characterized as iterating a recurrent relation to a desired number of times. Both network traffic reduction and work load balance have to be handled properly.

1.2 The problem of physics

The partial differential equation addressed in this work has the form:

$$\left[-\sum_{i=1}^N \frac{\partial^2}{\partial x_i^2} + V(x_1, \dots, x_N)\right]\psi(x_1, \dots, x_N) = E\psi(x_1, \dots, x_N) \quad (1.1)$$

Eq(1.1) describes a quantum mechanical system of N particles; each particle is denoted by x_i for $i = 1, \dots, N$. The particles move in a three-dimensional space with the potential of interaction between the particles given by V . The solution we seek, ψ , or wave function, is the one associated with the lowest possible value E .

The set of all $3N$ dimensional vectors, $\{R\}$, representing the system configuration, form an Euclidean space called a *configuration space*, C . Physical states of the system are completely characterized by those real-valued functions on C that satisfy both the symmetry constraints and boundary conditions of the problem. (In other words, not all real functions on C describe the physical states of the given problem.) Boundary conditions are constraints on certain subsets of C .

Eq(1.1) can be solved by, first, transforming it into an integral equation,

$$\psi(x_1, \dots, x_N) = \int E G(x_1, \dots, x_N, x'_1, \dots, x'_N) \psi(x'_1, \dots, x'_N) d(x'_1, \dots, x'_N) \quad (1.2)$$

and then, starting with a trial solution $\psi^0 = \psi_T$ and a trial energy value $E_0 = E_T$, iterating the above recurrent equation,

$$\psi^{(n)}(x_1, \dots, x_N) = \int E_T G(x_1, \dots, x_N, x'_1, \dots, x'_N) \psi^{(n-1)}(x'_1, \dots, x'_N) d(x'_1, \dots, x'_N) \quad ,$$

where $n = 1, 2, \dots, m$, and G in Eq(1.2) is the Green's function for the operator of Eq(1.1). The Green's function will be discussed in detail in chapter 2 (*Background*). After m times of iterating Eq(1.2), ψ , the ground state solution, is obtained.

E in Eq(1.1) can be computed by evaluating a quotient having the following form with the solution ψ :

$$\frac{\int f(x_1, \dots, x_N) \psi(x_1, \dots, x_N) d(x_1, \dots, x_N)}{\int g(x_1, \dots, x_N) \psi(x_1, \dots, x_N) d(x_1, \dots, x_N)} \quad (1.3)$$

Functions f and g in Eq(1.3) are real-valued functions.

There are two kinds of particles that may be involved in Eq(1.1): bosons and fermions. Function f is called a wave function. It is symmetric if

$$f(\dots x_i, \dots x_j, \dots) = f(\dots x_j, \dots x_i, \dots) \quad (1.4)$$

and is antisymmetric if

$$f(\dots x_i, \dots x_j, \dots) = -f(\dots x_j, \dots x_i, \dots) \quad (1.5)$$

where x_i and x_j are the vector positions of two particles. If x_i and x_j are identical fermions, then the wave function f must be antisymmetric with respect to the i th particle exchanged with the j th particle. Conversely, if x_i and x_j are identical bosons, then f is symmetric with respect to particle interchange.

The system studied here is one containing three fermions. So Eq(1.1) was solved with $N = 3$, and it was assumed that particles 1 and 2 had like spin, which differed from the spin of particle 3. A central potential was used in our problem:

$$V(x_1, x_2, x_3) = V_{12}(x_{12}) + V_{12-3}(x_{12-3}) \quad (1.6)$$

where V_{12} and V_{12-3} both are in the form of square potentials with repulsive and attractive parts. x_1 , x_2 and x_3 are 3-dimensional vectors; $x_{12} = x_1 - x_2$ and $x_{12-3} = (x_1 + x_2)/2 - x_3$.

Specifically, V_{12} , V_{12-3} have the form:

$$V_{12,12-3}(x) = \begin{cases} V_{rep} & \text{if } x < P \\ V_{wel} & \text{if } P \leq x \leq Q \\ 0 & \text{if } x > Q \end{cases} \quad (1.7)$$

where V_{rep} , V_{wel} , P and Q are constants.

The differential equation of three particles with potential Eq(1.6) can be separated, and the separated equations are not coupled meaning that there are no interacting potential functions V_{12} or V_{12-3} involved in more than one of such equations. V_{12} is the interacting potential between particles 1 and 2. V_{12-3} is the interacting potential between the center of mass of particles 1 and 2 and particle 3. There is no external force involved in Eq(1.6).

By decoupling the equations of motion, Eq(1.1) is separated by transforming it to another set of canonical coordinates. Each of the separated individual equation can be solved analytically. The numerical results obtained from a simulation can therefore be compared with the results obtained analytically.

Eq(1.1) can also be written as the Schrödinger wave equation:

$$H\psi_n(R) = E_n\psi_n(R), \quad (1.8)$$

where H is the Hamiltonian operator and is given by:

$$H = -\nabla_R^2 + V(R)$$

R is understood as nine-dimensional vector in the configuration space. $V(R)$ is the potential of the interaction, the real function in Eq(1.6). The functions that are solutions of the Schrödinger equation characterizing well- defined energies, E_n (eigenvalues of the system), are eigenfunctions ψ_n , where $n = 0, 1, 2, \dots$

The functions we are interested in have the correct symmetry and boundary conditions, and are solutions of the Schrödinger equation Eq(1.8) as well. In particular, we seek the ground state, ψ_0 with eigenvalue E_0 .

1.3 Monte Carlo solution

Unlike a single-host Monte Carlo computation in which the sampled populations are stored in local memory, in our problem two populations are distributed over all the processes involved in the computation. Unfortunately, each point in a population is not totally independent of the other in the sense that it may need to be correlated with some points in the other population. To get better solutions to Eq(1.1), it must be taken into account that there are correlated pairs in the two different populations.

There are other problems in distributed simulations. One of them is how to decompose a sequential computation so that the given problem can be evaluated in parallel. A similar problem is how to gather results from distributed processes at the end of a computation to obtain the final results. In our work, the results refer

to the expected value E for solution ψ . Another problem is to increase efficiency of the entire computation on a network. This is the usual load balance issue in distributed computing.

Monte Carlo methods and related techniques are the main topics been discussed throughout this thesis. Here, it is worthwhile to discuss them briefly to help with the following discussion.

The Monte Carlo (MC) methods explicitly use pseudo-random numbers to estimate the solutions of a given problem. The methods are usually used to simulate stochastic processes. In our problem, MC is used to estimate the deterministic integral given in Eq(1.2). The ψ is then considered a random variable. To obtain an accurate and produce expectation, $\langle\psi\rangle$, close to the actual solution ψ_0 , requires a large number of pseudo-random values sampled. Monte Carlo methods are discussed in section 2.2.

A set of pseudo-random particle positions is called a population. For solving a one-dimensional integration of the form of Eq(1.2), a population may be stored in a linearly organized data structure. In some many-dimensional problems, a population of points may be stored in some simple data structures. However, in our problem, two populations are needed to be sampled and processed together. Complicated data structures must be used to maintain and process two different, simultaneous populations.

1.4 Portable distributed programming

The computational work of this thesis is accomplished using the Distributed Pro-

cess library (DP) [Arnow, Weiss]. DP is a message-passing system that is suitable for various scientific distributed applications, including Monte Carlo calculations. The DP libraries are currently supporting several different platforms. All of these running versions of DP have an identical interface and implement the same semantics. Application programs written in the DP standard are portable to all of these platforms.

In this work, two implementations of DP were constructed to facilitate the development of the distributed Monte Carlo calculation. They are the concurrent DP running on Unix SYSTEM V, and Mach DP running on a KSR parallel-computer. The pthread program running under Mach operating system is also ported to Microsoft Windows NT systems, which supports a similar function interface as that of the KSR machines.

1.5 Problem of parallel random number generations

Monte Carlo methods assume that all random numbers will be drawn independently. This may not be a problem if a good pseudo-random number generator is available and one sequence is drawn from it. However, for our distributed computing, there is no guarantee that the sequences are uncorrelated if all individual processes draw random numbers from one random number generator.

In this work, a distributed random number system (DRAND) is implemented based on the paper titled "Random Number Generators for MIMD Parallel Processors" [1]. DRAND is a distributed system that supports all kinds of distributed simulations. It guarantees that in a computation all distributed processes draw

their pseudo-random numbers independently. It also supports simultaneous multi-distributed applications.

1.6 The results

As the result of this work, two Distributed Process library variations were written, on which distributed one and nine dimensional Monte Carlo algorithms were developed. Primarily, the concurrent DP under Unix was employed to help understand distributed programming and therefore to develop new computational schemes for MC. And the pthread DP under Mach was written to test general behavior of distributed Monte Carlo and DP on a different platform. A distributed random number generator system was developed to work with the distributed Monte Carlo calculations. In this part of the work, two Spectral test program were implemented for the purpose of finding MIMD linear pseudorandom number generators. In addition, two abstract data types, Fixed and Real, were created to help with all long integer arithmetic calculations. An immediate demand of these data types was to develop a system of generators with a period length of 2^{48} on 32-bit architectures. A sequential nine-dimensional fermion Green's function Monte Carlo program was reconstructed based on Arnow's work [2]. This program was successfully distributed in the Distributed Process environment, employing the distributed random number generators.

2

Background

The goal of this work is to integrate distributed computing and the Green's function Monte Carlo method, which is used to stochastically determine the lowest eigenfunction and eigenvalue of a system of quantum particles. This chapter will provide the background on these methods.

First, the problem of estimating the ground state energy of a system of a few quantum particles is a well-studied problem in physics. There is a large collection of literature in computational physics that discusses and provides approaches to this problem. Because the concern here is in the computer science aspects, only the physics that is essential for a basic understanding of the application will be covered. For additional details, the reader should consult the relevant references, found in the bibliography [2,3,4,8].

Second, the computations in this work involve a special kind of simulation technique that is used to estimate definite integrals, namely the Monte Carlo method. It is therefore necessary to discuss its basic principles at sufficient length so that the reader, who may not be familiar with the method, can follow the main thread of this work.

Next, the DP (Distributed Processing) environment [Arnow, et al, 1993] will be discussed. DP is a distributed message passing system suitable for carrying out scientific computations, especially the Green's function Monte Carlo method for fermion cases.

One of the primary goals of distributed computing is to achieve greater efficiency in the parallel computation. Efficiency is determined by the way that programs are coded, the algorithm being used and the platform on which the programs are executed. Balancing the work load is a typical treatment to increase computation efficiency. In our work, all DP processes are considered symmetric so balancing the work load means that every process carries out the average computational work. This shall certainly help to achieve maximum utilization of processors, and therefore, increase the efficiency of computations.

Another concern that is particularly important to our problem is the correctness of the distributed computation. A result that is not correct, of course, should be absolutely rejected. However, the results of a distributed simulation may not agree exactly with the answers obtained from sequential simulations. For a Monte Carlo computation with 10,000 sampled points, for instance, the results of a sequential computation and a distributed run are, in general, not the same even the same algorithm is used for the both. That may be understood as some expectations of random variables are no longer obtained by a single expression in a process, rather they were obtained by partial solutions contributed by distributed processes. The random number generations used in the computations also have to be taken into account. In our cases, sequential MC programs use one pseudorandom number generator, while parallel MC computations employ the distributed random number generators discussed in chapter 4. We should expect results obtained from the distributed computations to appear as good on the average as if the problems is

carried out sequentially. Nevertheless, standard deviation can be obtained by running a program, in parallel, many times, that will be used to judge the performance of the parallel computation.

There has been much work on parallel simulation in the field of computational physics. Most of these works focus on vector machines. It is difficult to discuss distributed simulation without having examples, so the distributing simulation part of this work is not discussed until chapter 5, in which it is discussed with an example of one free particle.

Finally, a system of distributed random number generators is discussed. In our simulation, a system of good distributed pseudorandom number generators with a long period is employed. Chapter 4 will discuss the earlier work on parallel random number generators as well as my work in this area.

All of the work discussed above was carried out using DP. In this work, my algorithms were developed in the concurrent model of DP, and the actual results were obtained from a computational network model. Earlier work on DP and related system, as well as my work in that area will be given in chapter 3.

2.1 Quantum Mechanics

The major concern of this work from the standpoint of physics is the solution of the Schrödinger equation, which describes motion, energy and momentum of a system of quantum particles. Specifically, the goal of this work is to obtain the lowest eigenvalue and eigensolution of Eq(1.1) for three fermions. The lowest eigenvalue is the ground state energy of the system.

2.1.1 The schrödinger equation

Consider a free quantum particle†with mass m having energy E , momentum $p = \sqrt{2mE}$ and the particle travels in a particular direction. The motion of the particle may be described by the wave function Ψ

$$\Psi(x, t) = Ae^{i(kx - \omega t)} \quad (2.1)$$

where A is the amplitude, $\omega = E/\hbar$ (angular frequency), $k = p/\hbar$ (wave number) and $\hbar$ is Planck's constant. Differentiate Eq(2.1) twice with respect to x we have:

$$\frac{\partial^2 \Psi}{\partial x^2} = -k^2 Ae^{i(kx - \omega t)} = -\frac{p^2}{\hbar^2} \Psi = \frac{2mE}{-\hbar^2} \Psi \quad (2.2)$$

For a more general situation, where potential, $V \neq 0$, the kinetic energy of the particle is $E - V(x)$. Here potential energy V is a function of a position of the dimensional variable x . Then the momentum is expressed as $p^2 = 2m(E - V) = \hbar^2 k^2$ that changes from one place to another according to the variation of function $V(x)$. Thus,

$$\frac{\partial^2 \Psi}{\partial x^2} = -\frac{2m}{\hbar^2} (E - V) \Psi \quad (2.3)$$

Written in the normal form for the time-independent Schrödinger Equation:

$$-\frac{\hbar^2}{2m} \frac{\partial^2 \Psi}{\partial x^2} + V \Psi = E \Psi \quad (2.4)$$

† A free quantum particle is one that no external forces affect on it, or $V = 0$.

or:

$$(-\nabla^2 + V)\Psi = E\Psi \quad (2.5)$$

In quantum mechanics, the significance of Ψ is that Ψ^2 is the probability distribution function (pdf) of finding the particle at any point in configuration space.

From this, all information about the motion of the particle may be obtained.

Let $H = -\nabla^2 + V$ where $H^\dagger$ is the Hamiltonian operator. Eq (2.5) becomes:

$\hat{O}f = \lambda f$. $\hat{O}$ is a generic notation representing the

$$H\Psi = E\Psi \quad (2.6)$$

$\Psi(x)$ is an eigenfunction when it is applied on operator H , and result is merely $\Psi(x)$ itself, multiplied by a constant E , the corresponding eigenvalue. These eigenfunctions meet boundary conditions and constraints representing physical states with a well defined energy given by the associated eigenvalue.

If there are more than one orthogonal eigenfunctions Ψ_i corresponding to the eigenvalue E , the eigenfunctions of Ψ_i are degenerate. Degenerate eigenfunctions occur in a quantum system when the system is in an excited state. A fermion state is considered an excited state, therefore, in our problem we will solve a Schrödinger equation that has degenerate wave function solutions.

[†] Action of an operator $\hat{O}$ is mapping a function f into another function g , $\hat{O}f = g$. When g linearly relates to f , $\hat{O}f = \lambda f$. $\hat{O}$ is a generic notation representing the mathematical operator, here $\hat{O} \equiv H$.

2.1.2 Antisymmetry

Suppose we have a system in which there are two identical fermions. The quantum mechanical rules require the particles changes sign when the coordinates of the particles are exchanged. Our model systems include identical fermions and so we will have to take this constraint into account.

In the following we provide an example showing the situation of antisymmetry of a system of two fermions.

Suppose r_1, r_2 are two 3D coordinates representing two particles's positions. $\Psi(r_1, r_2)$ is the wave function of the system. When r_1, r_2 are given, Ψ has a fixed value, and Ψ^2 , the probability that particle one is found at r_1 and particle two is found at r_2 , is positive. Because the particles are indistinguishable, the following holds:

$$\Psi^2(r_1, r_2) = \Psi^2(r_2, r_1) \quad (2.7)$$

In fact, there is no way to identify which particle is at which position. So, the probability that particle one is at r_1 and particle two is at r_2 is the same as the probability that particle one is at r_2 and particle two is at r_1 .

Therefore, there are two equations implied by Eq(2.7):

$$\Psi_B(r_1, r_2) = \Psi_B(r_2, r_1) \quad (2.8)$$

$$\Psi_F(r_1, r_2) = -\Psi_F(r_2, r_1) \quad (2.9)$$

In Eq(2.8) the wave function Ψ_B is symmetric. In Eq(2.9) the wave function Ψ_F is antisymmetric. To make the example more concrete, we can plot two real functions to the wave function representing the system of two identical particles. Let $\Psi(x_1, x_2) = \Psi_\alpha \Psi_\beta$ that Ψ_α and Ψ_β are two states consisting of Ψ and the two particles are restricted in an infinitely deep well with width L . Consider the particles at positions $x_1 = \frac{L}{2}$ and $x_2 = \frac{L}{4}$. $\Psi_\alpha \Psi_\beta$ is proportionally defined as:

$$\Psi_\alpha(x_1)\Psi_\beta(x_2) \sim \sin\left(\frac{\pi x_1}{L}\right)\sin\left(\frac{2\pi x_2}{L}\right) = 1$$

$$\Psi_\alpha(x_2)\Psi_\beta(x_1) \sim \sin\left(\frac{\pi x_2}{L}\right)\sin\left(\frac{2\pi x_1}{L}\right) = 0$$

For the symmetric case, we construct a wave function Ψ that has the same function value when the two particles are interchanged.

$$\Psi(x_1, x_2) \sim \Psi_\alpha(x_1)\Psi_\beta(x_2) + \Psi_\alpha(x_2)\Psi_\beta(x_1) = 1 + 0 = 1$$

$$\Psi(x_2, x_1) \sim \Psi_\alpha(x_2)\Psi_\beta(x_1) + \Psi_\alpha(x_1)\Psi_\beta(x_2) = 0 + 1 = 1$$

We can also construct a wave function Ψ for the antisymmetric case that function values are in opposite sign when the two particles are interchanged.

$$\Psi(x_1, x_2) \sim \Psi_\alpha(x_1)\Psi_\beta(x_2) - \Psi_\alpha(x_2)\Psi_\beta(x_1) = 1 - 0 = 1$$

$$\Psi(x_2, x_1) \sim \Psi_\alpha(x_2)\Psi_\beta(x_1) - \Psi_\alpha(x_1)\Psi_\beta(x_2) = 0 - 1 = -1$$

The former corresponds to the lowest energy state (Boson state) and the latter corresponds to the lowest excited antisymmetric state of the system.

2.1.3 The fermion problem

In this section the model problem will be specified. In addition, the method used to obtain the analytical solution is described. The details of solving the model problem analytically is given here to help understanding the model problem mathematically. The solution using Fermion Green's function Monte Carlo will be discussed in chapter 6.

2.1.3.1 Nine-dimensional Euclidean space

The wave function of N particles is in general a function of N 3D coordinate positions, $\Psi(r_1, \dots, r_N)$. Here r_i is a three dimensional vector, $r_i = (x_i, y_i, z_i)$. The set of coordinates of the N particles belong a configuration space C ;

$$\begin{aligned} \{R\} &= \{(r_1, \dots, r_N)\} \\ &= \{(x_1, y_1, z_1, \dots, x_N, y_N, z_N)\} \\ &= \{(q_1, q_2, q_3, \dots, q_{3N})\} \end{aligned} \tag{2.10}$$

Ψ is a real-valued function on the $3N$ -dimensional Euclidean space, C . To be physically meaningful, the function Ψ must satisfy the property of correct symmetry and boundary conditions. For any two identical fermions in positions, r_i, r_j , by exchanging the two particles, antisymmetry requires that,

$$\Psi(\dots, r_i, \dots, r_j, \dots) = -\Psi(\dots, r_j, \dots, r_i, \dots).$$

The boundary conditions are usually restrictions on the values of $\Psi(R)$ on a certain subset of C . For our purpose, $N = 3$ and R is a set of nine-dimension

vectors. The Ψ is a solution of the Schrödinger equation,

$$H\Psi_n(R) = E_n\Psi_n(R) \quad (2.11)$$

and,

$$H = -\nabla_R^2 + V(R)$$

where H , is the Hamiltonian operator, and $V(R)$, is a real-valued function on the space C . E_n is an eigenvalue, and Ψ_n is its corresponding eigenfunction, where $n = 0, 1, \dots$

Our work is to find the lowest antisymmetric energy state ($E_f = E_m$) such that ($m > 0$). Specifically, to obtain E_f , the process can be outlined as: solve Eq(2.11) by finding Ψ_f ; estimate the eigenvalue E_f by evaluating the quotient $E_f = \frac{\int \Psi_T(R)H\Psi_f(R)dR}{\int \Psi_T(R)\Psi_f(R)dR}$, where R is a set of nine-dimensional vectors, and Ψ_T is a trial function. *Here I will replace Ψ_T by a notation for growth estimator.*

2.1.3.2 The model problem

The system which we will compute is a system of three fermions, each moving in three dimensions, the details can be found in [8]. We assume that particles 1 and 2 are identical and differ from particle 3. The potential of the system is a superposition of two square potentials $V_{12}(r_{12})$ and $V_{12-3}(r_{12-3})$. The V_{12} is the interacting potential between particle 1 and 2 and is independent of particle 3. The V_{12-3} is the interacting potential between particle 3 and the center of mass of particle 1 and 2. The total system potential is

$$V(r_1, r_2, r_3) = V_{12}(r_{12}) + V_{12-3}(r_{12-3}) \quad (2.12)$$

where $r_{12} = r_1 - r_2$ and $r_{12-3} = \frac{(r_1+r_2)}{2} - r_3$, and V_{12} and V_{12-3} both are in the form of square potentials defined by four constants, V_{rep} , V_{wel} , P and Q as following:

$$V_{12,12-3}(r) = \begin{cases} V_{rep} & \text{if } r < P \\ V_{wel} & \text{if } P \leq r \leq Q \\ 0 & \text{if } r > Q \end{cases} \quad (2.13)$$

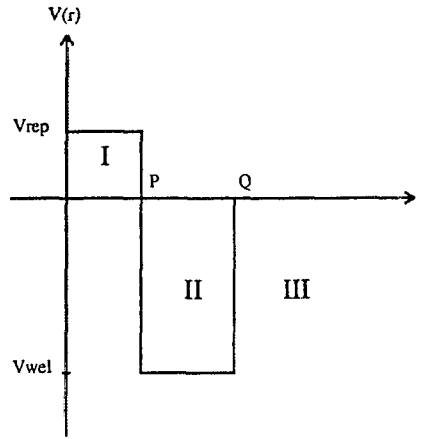


Figure 2.1 the Potential in the model problem

There is an analytical solution for this particular system because of the form of the potentials. The Schrödinger equation is separable. After certain coordinate transformations, the Schrödinger equation can be reduced to nine different ordinary differential equations, each of which can be solved numerically.

2.1.3.3 The analytical solution

The reason for choosing this model problem is that it has an analytical solution. So the simulation result will be compared with the analytical solution. In this section, we discuss the analytical solution.

The model problem originally, by the nature of the system, is described in the Cartesian coordinate system. For convenience, it first is transformed into polar coordinates in which the position of each particle is denoted as a vector $\mathbf{r}$. Then it is transformed and separated twice. Specifically, the first transformation transforms the original equation of vector $\mathbf{r}$ into an equation of internal coordinates. Then it is separated into three individual equations of the internal coordinates. The second transformation is needed to solve the three resulting equations. Each of the equation is transformed into an equation of polar spherical coordinates, and then it is further separated into three equations.

In classical mechanics, the system of three particles can be understood as moving in nine dimensional space and its motion is relative to the origin. The system has kinetic energy that characterizes the system. However, in our problem the major concern is to estimate internal energy so the motion of the entire system relative to external coordinates is ignored. A new internal coordinate system associated with the problem has to be defined.

In specifying the original problem, nine dimensions are used. Transforming the problem into a new coordinate system that also completely describes the same problem requires nine dimensions. By decoupling Eq(2.12), three-dimension coor-

dinate systems can be formed. Two of them are called internal coordinate systems which are formed as follows. One, $\mathbf{r}_{12}$, specifies the motion of particle one relative to particle two or vice versa. Notice that particle one and particle two are indistinguishable. And the other coordinate system, $\mathbf{r}_{12-3}$, specifies the motion of particle three to the center of mass of particles one and two. The last of the three-coordinate system is not an internal coordinate system. It specifies the motion of the center of mass, $\mathbf{R}_{CM}$, of the entire system to the coordinate origin.

With the new coordinate systems, Eq.(2.11) is separated into three independent equations, one for each problem.

To put the above discussion together, let's consider the following equation

$$\left[-\frac{1}{2m_1} \nabla_{\mathbf{r}_1}^2 - \frac{1}{2m_2} \nabla_{\mathbf{r}_2}^2 - \frac{1}{2m_3} \nabla_{\mathbf{r}_3}^2 + V(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3) \right] \Psi(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3) = E \Psi(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3) \quad (2.14)$$

The desired coordinates can be achieved through the transformation,

$$\{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3\} \rightarrow \{\mathbf{r}_{12}, \mathbf{r}_{12-3}, \mathbf{R}_{CM}\}$$

The general idea behind constructing a new coordinate system is to specify a point in two different mutually perpendicular families. That is to superimpose a new coordinates system on the nine dimensional coordinates. The relationship of these two coordinate systems can be formally described as the following set of functions:

$$\begin{aligned}
\xi_1 &= g_1(r_1, r_2, r_3), \\
\xi_2 &= g_2(r_1, r_2, r_3), \\
\xi_3 &= g_3(r_1, r_2, r_3).
\end{aligned} \tag{2.15}$$

ξ_1, ξ_2, ξ_3 are the new coordinates, and reverse functions for finding the original coordinates

$$\begin{aligned}
r_1 &= f_1(\xi_1, \xi_2, \xi_3), \\
r_2 &= f_2(\xi_1, \xi_2, \xi_3), \\
r_3 &= f_3(\xi_1, \xi_2, \xi_3).
\end{aligned} \tag{2.16}$$

Our problem involves transforming $\{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3\}$ to $\{\mathbf{r}_{12}, \mathbf{r}_{12-3}, \mathbf{r}_{CM}\}$. As stated previously, the $\mathbf{r}_{12}$ coordinate is the vector difference between particles one and two. The $\mathbf{r}_{12-3}$ coordinate is the vector difference between particle three and the center of mass of particles one and two. The third coordinate, $\mathbf{R}_{CM}$ is the coordinates for the center of mass of the system.

To find analytical expression for the transform functions, we first look at how the center of mass of two particles, with m_1 and m_2 at the locations of $\mathbf{r}_1$ and $\mathbf{r}_2$, is defined,

$$\begin{cases} \mathbf{R} = \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2}{m_1 + m_2} \\ \mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1 \end{cases} \tag{2.17}$$

It suggests that the transformation must consist of internal and the center of mass coordinate systems, $\mathbf{r}$ and $\mathbf{R}$, respectively and the total number of coordinates are six as the number of coordinates specifying the two particle problem.

Applying the same idea, transformation functions for our problems are:

$$\begin{aligned}
\mathbf{r}_{12} &= \mathbf{r}_1 - \mathbf{r}_2 \\
\mathbf{r}_{12-3} &= \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2}{m_1 + m_2} - \mathbf{r}_3 \\
\mathbf{R}_{CM} &= \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2 + m_3 \mathbf{r}_3}{m_1 + m_2 + m_3}
\end{aligned} \tag{2.18}$$

Next, the Laplacian for the new coordinates must be derived. This requires the definition of the operation of differentiating the new dimensional coordinates corresponding to the unit of length. (Later, the newly obtained coordinate systems will be transformed into spherical system. Some of those coordinates can not be expressed as length, for example ϕ, θ .) Let H_i be the measurement coefficients in the newly obtained metric system ξ_i . The differentiation operation is defined as:

$$ds_i = H_i d\xi_i, \quad i = 1, 2, 3 \tag{2.19}$$

where H_i is a function of mass m_1, m_2, m_3 . Using the definition of the reduced mass of two particles

$$\frac{1}{\mu} = \frac{1}{m_1} + \frac{1}{m_2},$$

the coefficients can be obtained

$$\begin{aligned}
H_{12}^2 &= \frac{m_1 m_2}{m_1 + m_2} \\
H_{12-3}^2 &= \frac{(m_1 + m_2) m_3}{m_1 + m_2 + m_3} \\
H_{CM}^2 &= m_1 + m_2 + m_3.
\end{aligned} \tag{2.20}$$

By letting

$$\begin{cases} M = m_1 + m_2 + m_3 & \text{and} & m = m_1 + m_2 \\ \mu_{12} = \frac{m_1 m_2}{m}, & \mu_{12-3} = \frac{m m_3}{M}, & \text{and} & \mu_{CM} = M, \end{cases}$$

Eq.(2.20) can be written as

$$\begin{aligned}
 H_{12} &= \sqrt{\mu_{12}} \\
 H_{12-3} &= \sqrt{\mu_{12-3}} \\
 H_{CM} &= \sqrt{\mu_{CM}}.
 \end{aligned} \tag{2.21}$$

The gradient ∇ is defined as:

$$\nabla \equiv \left(\frac{\partial}{H_1 \partial r_{12}}, \frac{\partial}{H_2 \partial r_{12-3}}, \frac{\partial}{H_3 \partial R_{CM}} \right),$$

and the Laplacian $\nabla \cdot \nabla = \nabla^2$ is:

$$\nabla^2 = \frac{1}{H_{12} H_{12-3} H_{CM}} \sum_{i=12,12-3,CM} \frac{\partial}{\partial r_i} \left(\frac{H_{12} H_{12-3} H_{CM}}{H_i^2} \frac{\partial}{\partial r_i} \right)$$

or:

$$\nabla_{r_{12}}^2 = \frac{1}{\sqrt{\mu_{12} \mu_{12-3} \mu_{CM}}} \frac{\partial}{\partial r_{12}} \left(\frac{\sqrt{\mu_{12} \mu_{12-3} \mu_{CM}}}{(\sqrt{\mu_{12}})^2} \frac{\partial}{\partial r_{12}} \right) = \frac{\partial^2}{\mu_{12} \partial^2 r_{12}}$$

$$\nabla_{r_{12-3}}^2 = \frac{1}{\sqrt{\mu_{12} \mu_{12-3} \mu_{CM}}} \frac{\partial}{\partial r_{12-3}} \left(\frac{\sqrt{\mu_{12} \mu_{12-3} \mu_{CM}}}{(\sqrt{\mu_{12-3}})^2} \frac{\partial}{\partial r_{12-3}} \right) = \frac{\partial^2}{\mu_{12-3} \partial^2 r_{12-3}}$$

$$\nabla_{r_{CM}}^2 = \frac{1}{\sqrt{\mu_{12} \mu_{12-3} \mu_{CM}}} \frac{\partial}{\partial R_{CM}} \left(\frac{\sqrt{\mu_{12} \mu_{12-3} \mu_{CM}}}{(\sqrt{M})^2} \frac{\partial}{\partial R_{CM}} \right) = \frac{\partial^2}{M \partial^2 R_{CM}}$$

By putting them together and Eq(2.14) becomes:

$$\left[-\frac{1}{2\mu_{12}} \nabla_{r_{12}}^2 - \frac{1}{2\mu_{12-3}} \nabla_{r_{12-3}}^2 - \frac{1}{2M} \nabla_{R_{CM}}^2 + V_{12}(r_{12}) + V_{12-3}(r_{12-3}) \right] \Psi(R) = E \Psi(R) \quad (2.22)$$

Now, Eq(2.22) is in our desired coordinate system. To solve Eq(2.22), there are two more steps to be completed: to separate Eq(2.22) and to transform the resulting equations to spherical polar coordinates systems. First let

$$\Psi(R) = \Psi_{12}(r_{12}) \Psi_{12-3}(r_{12-3}) \Psi_{CM}(R_{CM}) \quad (2.23)$$

and substitute Ψ in Eq(2.23) into Eq(2.22). We have

$$\begin{aligned} & \left[-\frac{1}{2\mu_{12}} \frac{\partial^2 \Psi_{12}(r_{12})}{\partial^2 r_{12}} + V_{12} \Psi_{12}(r_{12}) \right] \frac{1}{\Psi_{12}(r_{12})} \\ & + \left[-\frac{1}{2\mu_{12-3}} \frac{\partial^2 \Psi_{12-3}(r_{12-3})}{\partial^2 r_{12-3}} + V_{12-3} \Psi_{12-3}(r_{12-3}) \right] \frac{1}{\Psi_{12-3}(r_{12-3})} \\ & + \left[-\frac{1}{2M} \frac{\partial^2 \Psi_{CM}(r_{CM})}{\partial^2 r_{CM}} \right] \frac{1}{\Psi_{CM}(r_{CM})} = E \end{aligned} \quad (2.24)$$

The right hand side of Eq(2.24) is a constant, E . Let $E = E_{12} + E_{12-3} + E_{CM}$, then three separated equations are defined as following,

$$\left[-\frac{1}{2\mu_{12}} \frac{\partial^2}{\partial^2 r_{12}} + V_{12} \right] \Psi_{12}(r_{12}) = E_{12} \Psi_{12}(r_{12}) \quad (2.25)$$

$$\left[-\frac{1}{2\mu_{12-3}} \frac{\partial^2}{\partial^2 r_{12-3}} + V_{12-3} \right] \Psi_{12-3}(r_{12-3}) = E_{12-3} \Psi_{12-3}(r_{12-3}) \quad (2.26)$$

$$-\frac{1}{2M} \frac{\partial^2}{\partial^2 r_{CM}} \Psi_{CM}(r_{CM}) = E_{CM} \Psi_{CM}(r_{CM}) \quad (2.27)$$

Each of the above is a Schrödinger equation in three dimensions. Because of the similarity of these equations, our discussion will be concentrated on Eq(2.25), which can be written in Cartesian coordinates as:

$$-\frac{1}{2\mu_{12}} \left[\frac{\partial^2}{\partial^2 x_{12}} + \frac{\partial^2}{\partial^2 y_{12}} + \frac{\partial^2}{\partial^2 z_{12}} + V_{12} \right] \Psi_{12} = E_{12} \Psi_{12} \quad (2.28)$$

Next step is to transform it to spherical polar coordinates. This transformation is a routine in physics to solve a Helmholtz type equation $\nabla^2 u + \lambda u = 0$ that has a central potential. The transformation is similar to the one discussed earlier in Eq(2.15) and Eq(2.21) to obtain mapping functions, ξ_i and new coefficient measurements, H_i . The coordinates of the spherical system are:

$$\begin{cases} \xi_1 = r, \xi_2 = \theta, \xi_3 = \phi, \\ H_1 = 1, H_2 = r, H_3 = r \sin \theta; \end{cases} \quad (2.29)$$

Applying the technique of transformation, Eq(2.28) becomes,

$$\begin{aligned} -\frac{1}{2\mu_{12}} \left[\frac{1}{r^2} \frac{\partial}{\partial r} \left(r^2 \frac{\partial \Psi_{12}}{\partial r} \right) + \frac{1}{r^2 \sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial \Psi_{12}}{\partial \theta} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial^2 \Psi_{12}}{\partial \phi^2} \right] \\ + V_{12} \Psi_{12} = E \Psi_{12} \end{aligned} \quad (2.30)$$

By letting $\Psi_{12} = R(r)\Theta(\theta)\Phi(\phi)$ in Eq(2.30), the equation can be once more be separated into three spherical equations:

$$\Phi'' = -m^2 \Phi \quad (2.31)$$

$$\frac{1}{\sin \theta} \frac{d}{d\theta} \left(\sin \theta \frac{d\Theta}{d\theta} \right) + \left(l(l+1) - \frac{m^2}{\sin^2 \theta} \right) \Theta = 0 \quad (2.32)$$

$$\frac{1}{\mu_{12} r^2} \frac{d}{dr} \left(r^2 \frac{dR}{dr} \right) + \left((V - E) - \frac{l(l+1)}{\mu_{12} r^2} \right) R = 0 \quad (2.33)$$

Here m and l are introduced during the separation, these values have to be determined according to the boundary conditions. Eq(2.31)-Eq(2.33) all are ordinary differential equations. Particularly, Eq(2.33) is called the radial equation that determines the energy value E .

Now, by rewriting Eq(2.33) in terms of a parameter κ^2 , we have

$$\frac{1}{r^2} \frac{d}{dr} \left(r^2 \frac{dR}{dr} \right) + \left(\kappa^2 - \frac{l(l+1)}{r^2} \right) R = 0 \quad (2.34)$$

where κ^2 is expressed:

$$\kappa^2 = \begin{cases} \kappa_I^2 = \mu_{12}(E - V_{rep}) & \text{if } r < P \\ \kappa_{II}^2 = \mu_{12}(E - V_{wel}) & \text{if } P \leq r \leq Q \\ \kappa_{III}^2 = \mu_{12}E & \text{if } r > Q \end{cases} \quad (2.35)$$

Each of the $\kappa_{I,II,III}^2$ determines the energy in a different region in which the potential varies. And therefore, solving Eq(2.34) is equivalent to solving three equations for the energy for different potentials. Before going further, let's first discuss the solution of Eq(2.34) in general. Let $\kappa r = x$, and $R(r) = x^{\frac{1}{2}} Z(x)$, Eq(2.34) can be written:

$$\frac{d^2 Z}{dx^2} + \frac{1}{x} \frac{dZ}{dx} + \left[1 - \frac{(l + \frac{1}{2})^2}{x^2} \right] Z = 0 \quad (2.36)$$

Eq(2.36) is a spherical Bessel's equation of order $l + \frac{1}{2}$. Its solution may be one of the following functions $j_l(x)$, $n_l(x)$ or $h_l(x)$, which are defined as:

$$j_l(x) = \sqrt{\frac{\pi}{2x}} J_{l+\frac{1}{2}}(x) \quad (2.37)$$

$$n_l(x) = \sqrt{\frac{\pi}{2x}} Y_{l+\frac{1}{2}}(x) \quad (2.38)$$

$$h_l^{(1,2)}(x) = \sqrt{\frac{\pi}{2x}} H_{l+\frac{1}{2}}^{(1,2)}(x) \quad (2.39)$$

$J_{l+\frac{1}{2}}(x)$, $Y_{l+\frac{1}{2}}(x)$ and $H_{l+\frac{1}{2}}^{(1,2)}(x)$ are the Bessel function, Neumann function and Hankel function of order $l + \frac{1}{2}$, respectively. Specifically, the Bessel function has the form:

$$J_{l+\frac{1}{2}}(x) = (-1)^l \sqrt{\frac{2}{\pi}} x^{l+\frac{1}{2}} \left(\frac{d}{x dx} \right)^l \left\{ \frac{\sin x}{x} \right\}$$

And $j_l(x)$ is defined accordingly,

$$\begin{aligned} j_l(x) &= \left(\sqrt{\frac{\pi}{2x}} \right) \left((-1)^l \sqrt{\frac{2}{\pi}} x^{l+\frac{1}{2}} \left(\frac{d}{x dx} \right)^l \left\{ \frac{\sin x}{x} \right\} \right) \\ &= (-1)^l x^l \frac{d^l}{(x dx)^l} \left\{ \frac{\sin x}{x} \right\} \end{aligned}$$

The Neumann function and Hankel function are defined similarly. Details can be found in any mathematical physics book.

Eq(2.33) is actually equivalent to three equations with respect to three regions of our problem, so are Eq(2.31) and Eq(2.32). To simplify the discussion, here Eq(2.33) (and Eq(2.34)) is discussed as an example with no subscripts. κ^2 in Eq(2.34) is defined on interval $[0, \infty)$ and has piecewise constants:

$$\begin{array}{lll} \mu(E - V_{rep}) & \text{on } [0, P) & (\text{ region I }) \\ \mu(E - V_{wel}) & \text{on } [P, Q) & (\text{ region II }) \\ \mu E & \text{on } [Q, \infty) & (\text{ region III }) \end{array}$$

Because the discontinuity of the potential $V(x)$, $Z(x)$, the solution to the potential $V(x)$, cannot be defined as one function in the entire domain of x . Rather it will be defined as a linear combination of several functions. Each of the individual functions is defined on a region where $V(x)$ is a constant. The solution requires certain demands of continuity on the $Z(x)$ on the boundary surfaces; $\frac{dZ(x)}{dx}$ must be continuous.

The general form of the linear combination of functions for each region can be as following:

$$Z(x) = C_1 \psi_{C_1} + C_2 \psi_{C_2}$$

where C_1, C_2 are constants and ψ_{C_1} and ψ_{C_2} are two appropriate Bessel functions. In region I, III k is imaginary so that the k functions with imaginary argument must be employed. Those functions are usually denoted by $i_l(x)$ and $k_l(x)$. Further, $k_l(0)$ is undefined in region I and $i_l(\infty)$ is undefined in region III.

Thus, the functions are:

$$\begin{cases} c_I i_l(k_I r) & \text{region I} \\ c_{II} j_l(k_{II} r) + c'_{II} n_l(k_{II} r) & \text{region II} \\ c_{III} k_l(k_{III} r) & \text{region III} \end{cases}$$

And the continuity condition on the boundary P requires:

$$\begin{cases} c_I i_l(k_I P) & = c_{II} j_l(k_{II} P) + c'_{II} n_l(k_{II} P) \\ c_I k_l i'_l(k_I P) & = c_{II} k_{II} j'_l(k_{II} P) + c'_{II} k_{II} n'_l(k_{II} P) \end{cases}$$

The continuity condition on the boundary Q requires:

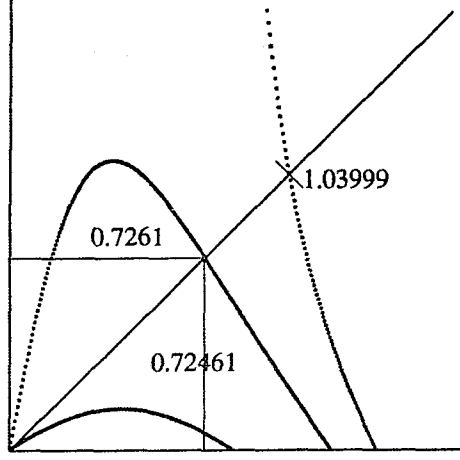
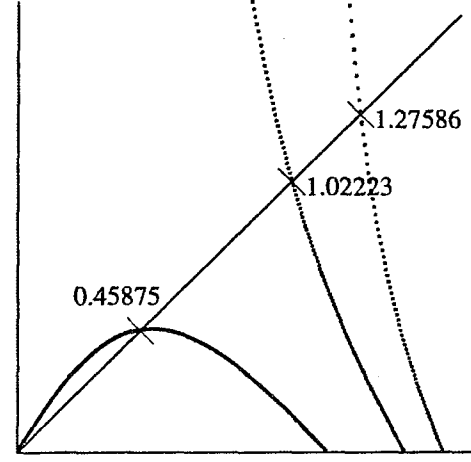
$$\begin{cases} c_{III} k_l(k_{III} Q) & = c_{II} j_l(k_{II} Q) + c'_{II} n_l(k_{II} Q) \\ c_{III} k_{III} k'_l(k_{III} Q) & = c_{II} k_{II} j'_l(k_{II} Q) + c'_{II} k_{II} n'_l(k_{II} Q) \end{cases}$$

The exact solution is obtained by applying the fixed point method on

$$x_{n+1} = Z(x_n) \quad n \geq 0$$

The root $a = x_{n+1}$ of $Z(x)$ has property that $|x_n - x_{n+1}|$ is less than some small value, δ , if the concurrent iteration converges. x_0 can be made arbitrarily small.

The details of the numerical method and program that computes the problem are omitted. The numerical solutions are plotted in two graphical pictures.

Figure 2.2 (a) approximating V_{12} .Figure 2.2 (b) approximating V_{12-3} .

By plotting the above diagrams, two roots are obtained, 1.03999 and 0.72461 for V_{12} . And three roots are found, 1.27586, 1.02223 and 0.45875 for V_{12-3} . To compute the energy, apply

$$\{ E_f = V_{12}^2 * \frac{1}{2.0 * \hbar} E_b = V_{12-3}^2 * \frac{2.0}{3.0 * \hbar} \quad (2.40)$$

For example, take a value, $V_{12-3} = 1.27586$, and plug it into Eq(2.40). We have:

$$\begin{aligned} E_b &= -1.27586^2 \frac{2.0}{3.0 * \hbar} \\ &= -1.27586^2 \frac{2.0}{3.0 * 20.76} \\ &= -50.69 \end{aligned}$$

Finally, the energy levels for V_{12} and V_{12-3} are plotted into the following graphs.

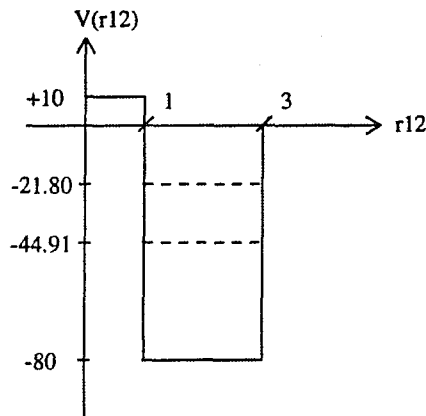


Figure 2.3 (a) Energy levels for r_{12} .

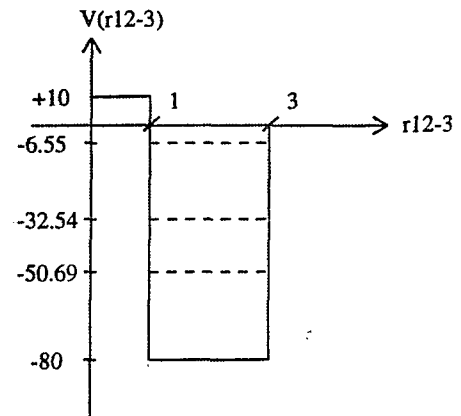


Figure 2.3 (b) Energy levels for r_{12-3} .

2.2 The Monte Carlo Method

First, we start with the classic definition of the Monte Carlo method. Then we introduce some techniques that are used in this work, [17]. The techniques we are going to address are: the rejection technique, random walk, and importance sampling. Unlike the general Monte Carlo simulation literature, the way we discuss the techniques is in a fairly intuitive way, some with a short example to show how they work. And if necessary, we cite some mathematical results from other work to show general uses of the technique.

Rejection techniques may be used to sample any probability density function

(pdf). A random walk is a fundamental technique that is used very often in many different disciplines. Importance sampling is used to reduce the variance of the simulation and possibly to increase the efficiency of computation. The technique of sampling the Green's function introduced in this section is considered as general knowledge that links simulation to actual problem solving. There was a great effort made to use the sampling of the Green's function in solving problems of this kind. We will discuss it in detail throughout this work.

2.2.1 Estimating definite integrals

Unlike other numerical methods that calculate areas or volumes by summing small areas or volumes obtained from evaluating a known integral, the Monte Carlo method is a method that estimates a definite integral by explicitly using n pseudorandom numbers. They are sampled from an integrated area or volume to form an estimator with statistical error of order, $O(1/\sqrt{n})$. This method is particularly attractive for solving complicated integrations, especially in many-dimensions.

In the first place, the Monte Carlo method can be used to estimate the following definite integral:

$$\int_0^1 f(x)dx \simeq \frac{1}{n} \sum_{i=1}^n f(\xi_i) \quad (2.41)$$

That is, the integral is approximated by the average of n numbers $f(\xi_i)$, where ξ_i is a random variable drawn from a probability density function $u(\xi)$, and $u(\xi)$ is uniform in $[0, 1)$.

If the integration has to be calculated in an interval other than $[0,1]$, the estimate would be:

$$\int_a^b f(x)dx \simeq \frac{(b-a)}{n} \sum_{i=1}^n f(x_i), \quad (2.42)$$

where x_i is drawn from pdf $p(x)$, and $p(x)$ can be considered as a linear transformation $p(\xi) = (b-a)u(\xi) + a$.

In many-dimensional case, for instance three, the Monte Carlo integration may be written as:

$$\int \int \int_{\Omega} f(x, y, z) dx dy dz \simeq \frac{\Omega}{n} \sum_{i=1}^n f(x_i, y_i, z_i) \quad (2.43)$$

where (x_i, y_i, z_i) is a random sequence, that can be drawn at random from $p(x)$ consecutively. In Eq.(2.43) Ω denotes the volume of the integration, and $p(x)$ is uniformly distributed in the volume.

To illustrate the idea of Monte Carlo integration, we present a simple example that applies all issues addressed in this section. The result of the integration $\int_1^5 (x^2 + 1)dx$ is $\simeq 45.33$. In the Monte Carlo method we can construct a Fortran program as follows:

```

1
2      DOUBLE PRECISION PDF,SUM/0./
3      INTEGER I
4      PARAMETER(ICOUNT=100000, INC=20000, ILB=1, IUB=5)
5      DO 10 I=1,ICOUNT
6          SUM=SUM+(PDF(ILB,IUB)**2+1)
7          IF (MOD(I,INC).EQ.0) THEN
8              PRINT '(A2,I8,A7,F10.6)', 'I=', I, 'INTEG=',
9              1      (IUB-ILB)*SUM/I
10             ENDIF
11 10    CONTINUE
12      STOP
```

```

13      END
14
15      DOUBLE PRECISION FUNCTION PDF(IA,IB)
16      DOUBLE PRECISION UPDF
17      PDF=(IB-IA)*UPDF()+IA
18      RETURN
19      END
20

```

Code fragment 2.1 Monte Carlo integration

UPDF in the function PDF is an uniform probability density function in $[0, 1)$, and PDF is a linear transformation function obtaining the desired pdf. The main program prints five lines of output that are provided as:

```

1      I=    20000 INTEG= 45.166407
2      I=    40000 INTEG= 45.239558
3      I=    60000 INTEG= 45.325381
4      I=    80000 INTEG= 45.377422
5      I=   100000 INTEG= 45.284973

```

2.2.2 Variance and the estimation of error

Because of the statistical error that is inherent in the Monte Carlo method, a major part of the work is to estimate variance and error. There are several probability theorems that can be employed in the Monte Carlo methods to estimate errors. We can use the Law of Large Numbers to show that the errors in a Monte Carlo calculation can be made quite small and use the Central Limit Theorem of probability to estimate the probability of a deviation of our estimate (of any size) from the correct answer.

In general, the errors in Monte Carlo calculations are statistical ones that are introduced in the process of sampling. For Eq(2.41), suppose all random variables ξ_i are drawn independently from one distribution and that the expectation, a , of $f(\xi_i)$

exists and is finite. Then, for an arbitrarily small positive number ϵ , the following holds†:

$$\lim_{n \rightarrow \infty} P\left(\left|\frac{1}{n} \sum_{i=1}^n f(\xi_i) - a\right| > \epsilon\right) = 0. \quad (2.44)$$

From this equation, called the Law of Large Numbers, we know that, as n becomes sufficiently large, the error of the arithmetic average $\frac{f(\xi_1) + \dots + f(\xi_n)}{n}$ as the estimate for the value a becomes very small.

The Law of Large Numbers does not show a way to estimate the variance and the distribution of the error. The Central Limit Theorem (CLT) states that the distribution of the error of a random variable that is the sum of independent random variables, each of which has a finite variance, converges to the normal distribution as the number of terms in the sum increases. Again, suppose all random variables ξ_i are drawn mutually independently from a common distribution and that the expectation a of $f(\xi_i)$ exists and is finite. In addition, suppose that the variance of $f(\xi_i)$, σ^2 , exists and is finite. Then let

$$S_n = \sum_{i=1}^n f(\xi_i).$$

Then the CLT states that†:

$$\lim_{n \rightarrow \infty} P\left\{\left(\frac{1}{n} S_n - a\right) < \left(\frac{\sigma}{\sqrt{n}}\right) \beta\right\} = \mathcal{N}(\beta, 0, 1)$$

or

† William Feller, “An Introduction to Probability Theory and Its Applications”, John Wiley & Sons, Inc. New York, Vol. I, pp 243

† Feller, “An Introduction to Probability Theory and Its Applications”, Vol. I, pp. 244

$$\lim_{n \rightarrow \infty} P\left\{\left(\frac{1}{n}S_n - a\right) < \beta\right\} = \mathcal{N}(\beta, 0, \sigma/\sqrt{n})$$

In other words, the deviation of the estimator, $\frac{1}{n}S_n$ from the mean a becomes normally distributed about 0 as n gets sufficiently large. The variance of this limit distribution is σ^2/n .

In Monte Carlo practice, what this means is that if n is sufficiently large one could, if σ were known, make precise statements about the likelihood of a large deviation of the estimator $\frac{1}{n}S_n$ from its mean a . For example, one could write

$$P\left\{\left(\frac{1}{n}S_n - a\right) < 2\sigma/\sqrt{n}\right\} = .954$$

In practice, σ is not known, though it can be estimated. But even apart from estimations of σ , for purpose of making statements like the above, the CLT says that for sufficiently large n , quadrupling n decreases the error of the estimate by half. It also tells us that the error of the estimate is proportional to the error of $f(\xi_i)$. So besides increasing n , an alternative and more effective way of decreasing the error of the estimate is to employ variance reduction techniques.

2.2.3. Sampling techniques

The Monte Carlo method can be generally described as a stochastic process that can be used to solve nonprobabilistic problems. When carrying out a stochastic process in a digital computer, it is assumed there exists a mechanism that produces a long random or pseudorandom sequence of real numbers between the interval $[0, 1)$.

For example, evaluating the definite integral in Eq(2.41) is a nonprobabilistic problem. The sampling of $u(x)$ using the probability density function, involves the simulation of the stochastic process.

Sampling $u(x)$ means generating a sequence of n independent real numbers that is uniformly distributed in the interval $[0, 1)$. Formally the *pdf* $u(x)$ can be characterized as

$$x \in [0, 1)$$

$$\int_0^x u(y)dy = P(X < x)$$

To actually sample $u(x)$, one usually employs a pseudorandom number generator that is provided by most computer systems†

However, in most Monte Carlo calculations the probability density functions $p(x)$, from which random numbers must be drawn, are not uniform. Let Ω be a set of points, $\Omega \subset [0, 1)$, and define a new *pdf* $p(x)$ such that

$$P\{x \in \Omega\} = \int_{\Omega} p(x)dx \leq 1. \quad (2.51)$$

Therefore, mapping a uniformly distributed sequence of real numbers in $[0, 1)$ into a set of points satisfying Eq.(2.51) is necessary. We call this mapping a sampling procedure. There are many different techniques to sample probability density functions; one of which is the rejection technique.

† In our simulations we use our own pseudorandom number generators which are well-studied and are good ones. For computations using computer system provided prn's, one should read *Random Number Generators: Good Ones Are Hard to Find*, Stephen K. Park & Keith W. Miller. Communications of ACM, Oct. 1988 p. 1192-1201

2.2.3.1 The rejection technique

The rejection sampling technique is a general technique and can be used to sample any probability distribution. An important property of this technique is that the sampled points are normalized while the normalization of the sampled density function is not necessarily explicitly known.

The technique can be described as follows: a trial value, a random number, is selected and that the value is subjected to one or more tests. It can be accepted or rejected based on tests of another random value. If it is rejected, the sampling process starts over again until an acceptance takes place.

The rejection and acceptance have the nature that more values are accepted when the sampled density function is large and fewer values are accepted when the sampled density function is small.

To illustrate the idea, we provide a simple example. Suppose we wish to sample function $f(x)$ in an interval $[0, 1]$, and also suppose that the maximum value of $f(x)$ is denoted as $\text{Max}f(x) = M < \infty$. A trial value ξ_1 and a test value ξ_2 between $[0, 1]$ are drawn from a uniform density function $u(x)$. The test for accepting ξ_1 as a sample from $f(x)$ is

$$\text{accept } \xi_1 \quad \text{if } \xi_2 \leq \frac{f(\xi_1)}{\text{Max}f(x)}$$

that is, ξ_1 has to satisfy the expression

$$\xi_2 \leq \frac{f(\xi_1)}{\text{Max}f(x)} \leq 1$$

Notice that ξ_2 is uniformly distributed on $[0, 1]$, while $f(\xi_1)/\text{Max}f(x)$ is not. The requirement, that the larger the value of $f(\xi_1)/\text{Max}f(x)$, the higher chance that ξ_1 is accepted, is achieved. The drawback of this technique is that it may reject many ξ_1 before it accepts one. So it may have low efficiency.

The sampling algorithm can be summarized as follows:

1. Draw two random numbers ξ_1, ξ_2 from a uniform distribution
2. If $\xi_2 \text{Max}f(x) \leq \xi_1$ accept ξ_1
3. Otherwise, repeat the steps (1) and (2)

The rejection technique will be used in this work to sample Green's functions.

2.2.3.2 Sampling by random walk – the Metropolis method

Here we introduce the Metropolis algorithm [21] that can be used to sample probability density distribution functions in any dimension. The Metropolis algorithm is usually used in variational Monte Carlo calculations [15]. Two essential techniques that are used in the Metropolis algorithm are also employed in Green's Function Monte Carlo, namely the use of a transition function and conditional sampling.

Suppose we want to sample a nondecreasing density function $f(x)$ in 1-dimension. A transition function, $T(X|Y)$, governs whether a given point Y randomly moves to either its left or its right to X . When Y walks to its left, $X < Y$ and $f(X) < f(Y)$. And when Y walks to its right, $X > Y$ and $f(X) > f(Y)$. If Y moves to the right meaning that Y tends to a point where $f(x)$ is larger, Y is always accepted. So X replaces Y . The movement of Y to a smaller value of $f(x)$, to its left in this case, is decided by the probability $\frac{f(X)}{f(Y)} > \xi$, where ξ is a random

variable drawn from a uniform density distribution. If the movement is accepted, X replaces Y ; otherwise, Y stays as is.

The Metropolis algorithm is more general than that of the rejection technique. It correctly samples from $f(x)$ asymptotically and does not reject values, so it has no loss in efficiency. $f(x)$ is larger and fewer values will be accepted in the region where

for a given Y in the random walk

1. Draw a random value X from a uniform distribution
2. Let q be $f(X)/f(Y)$
3. Let acc be $\min(1, q)$
4. If acc is equal to 1, then accept X
5. Else if acc is greater than ξ drawn from an uniform distribution, accept X as the next step in the random walk
6. Else accept Y as the next step in the random walk

The transition function, which generates a new position X , can be as simple as a uniform density function. acc that has the value of 1 implies that the ratio of $f(X)/f(Y)$ is greater than 1. It means that Y will tend to move toward its right, therefore, the movement is always accepted.

The disadvantage of this technique is that values sampled by using this technique are serially correlated. Therefore, when they are used in Monte Carlo calculation to estimate integrals, the variance may be decreased or may be increased because the sampled values are not independent.

The rejection technique will be used in this work to sample Green's functions.

Next, we discuss variance reduction techniques.

2.2.3.3 Importance sampling

Importance sampling is used as a technique to reduce variance. From Eq.(2.45) and

Eq.(2.46) the variance σ^2 is:

$$\begin{aligned}
 \sigma^2 &= \int_a^b (f(x) - \langle Q \rangle)^2 p(x) dx \\
 &= \int_a^b f^2(x) p(x) dx - 2 \int_a^b f(x) \langle Q \rangle p(x) dx + \int_a^b \langle Q \rangle^2 p(x) dx \\
 &= \int_a^b f^2(x) p(x) dx - 2 \langle Q \rangle^2 + \langle Q \rangle^2 \\
 &= \int_a^b f^2(x) p(x) dx - \langle Q \rangle^2
 \end{aligned} \tag{2.52}$$

Notice that here $\langle Q \rangle = \int f(x) p(x) dx$ and $\int p(x) = 1$.

As an example, we evaluate the integral $\int_0^1 x^3 dx = 0.25$. Using the density function $p(x)$ such that $P(X = x) = 1$, the variance becomes

$$\sigma^2 = \int_0^1 (x^3)^2 - \langle x^3 \rangle^2 = \int_0^1 x^6 - 0.25^2 = 0.0804$$

In order to reduce the variance, the density function $p(x)$ should be chosen so that it is integrable and is as close to function $q(x)$ as possible.

Now let $\tilde{p}(x)$ be an importance function. Then Eq.(2.52) becomes:

$$\sigma^2 = \int_0^1 \frac{f^2(x)}{\tilde{p}(x)} \tilde{p}(x) dx - \langle Q \rangle^2 \tag{2.53}$$

where $\int_0^1 p(x) = 1$ and choose $\tilde{p}(x) = \frac{p'(x)}{\int p'(x)dx}$, $p'(x)$ is integrable. Let us choose $\tilde{p} = 3x^2 / \int_0^1 3x^2 dx = 3x^2$ as the important function. The variance of the integration can be calculated as:

$$\sigma^2 = \int_0^1 \frac{(x^3)^2}{3x^2} dx - 0.25^2 = \frac{1}{15} x^5 \Big|_0^1 - 0.25^2 \simeq 0.066666 - 0.06250 \simeq 0.0042$$

In this particular work an importance function can not be explicitly expressed. It will be obtained by a probability density function which is known to be close to the function to be sampled.

2.3. The Green's functions

One possible way to solve differential equation Eq(1.1) is to translate Eq(1.1) into an integral equation in which the function to be determined appears under the integral sign. The method to solve the integral equation by using simulation is called the Green's function Monte Carlo, [Kalos, 1962]. In this section, the transformation to an integral equation from a linear differential equation is discussed. The Green's function Monte Carlo, GFMC, will be discussed in chapters 5 and 6.

Consider the linear ordinary equation,

$$\frac{d^2 y}{dx^2} + \lambda y = 0 \tag{2.54}$$

with the boundary conditions: $y(0) = y(l) = 0$. Integrating it over on interval $(0, x)$, we have,

$$\frac{dy}{dx} = -\lambda \int_0^x y(x_1) dx_1 + C_1. \tag{2.55}$$

where C_1 is a unknown constant, $\frac{dy}{dx}\Big|_{x=0}$. λ is a constant that shall be interpreted differently depending on the applications. Integrating Eq(2.55) over $(0, x)$ one more time, we have:

$$y(x) = -\lambda \int_0^x (x - \xi)y(\xi)d\xi + C_1x + C_2$$

Incorporating the boundary condition $y(0) = 0$ in C_2 , the above equation becomes:

$$y(x) = -\lambda \int_0^x (x - \xi)y(\xi)d\xi + C_1x. \quad (2.56)$$

Now C_1 has to be determined so it will satisfy the condition $y(l) = 0$ as well.

Substituting x by l , we have,

$$\lambda \int_0^l (l - \xi)y(\xi)d\xi = C_1l$$

or,

$$C_1 = \frac{\lambda}{l} \int_0^l (l - \xi)y(\xi)d\xi \quad (2.57).$$

Combining with Eq(2.57), then Eq(2.56) can be written as,

$$y(x) = \lambda \int_0^x \frac{\xi}{l}(l - x)y(\xi)d\xi + \lambda \int_x^l \frac{x}{l}(l - \xi)y(\xi)d\xi. \quad (2.58)$$

Let $K(x, \xi)$ be defined as:

$$K(x, \xi) = \begin{cases} \frac{\xi}{l}(l - x) & \text{if } \xi < x, \\ \frac{x}{l}(l - \xi) & \text{if } \xi > x. \end{cases} \quad (2.59)$$

$K(x, \xi)$ is called the kernel of Eq(2.58) whose value depends on x , the current variable, and ξ , the auxiliary variable. Rewrite Eq(2.58), then:

$$y(x) = \lambda \int_0^l K(x, \xi) y(\xi) d\xi. \quad (2.60)$$

A problem to solve the original differential equation becomes a problem to solve an integral equation, Eq(2.60). The kernel is presented as two analytical expressions on two different intervals $(0, x]$ and $[x, l)$ while when $\xi = x$ the two expressions are equivalent. For any given ξ , $K(x, \xi)$ is a function of x and $K(x, \xi)$ is continuous at $x = \xi$. The derivative $\partial K / \partial x$ is $(1 - \xi/l)$ when $(x < \xi)$, and $(-\xi/l)$ when $(x > \xi)$. Thus the derivative is discontinuous at $x = \xi$, and it has a finite jump of magnitude -1 as x increases through ξ . $K(x, \xi)$ satisfies the boundary conditions and $\frac{\partial^2 K(x, \xi)}{\partial x^2} = 0$, and $K(x, \xi)$ vanishes at the end points $x = 0$ and $x = l$. Furthermore, the kernel $K(x, \xi)$ is symmetric, that is: $K(x, \xi) = K(\xi, x)$.

More general, consider a homogeneous differential equation

$$\nabla^2 y + \phi(y) = 0 \quad (2.61)$$

together with boundary conditions, each has the form: $\alpha y + \beta y' = 0$, where ∇^2 is the second order differential linear operator. What we need is to determine the Green's function G so that the Eq(2.61) can be transformed into an integral equation.

Linear integral equations are used as the result of possibility of superimposing the effects due to several causes. In physics, it is quite often that one wants to deal with problems described by the following function:

$$\delta(x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{1}{h} & \text{if } 0 < x < h \\ 0 & \text{if } x > h \end{cases} . \quad (2.62)$$

And integrating $\delta(x)$ over $(-\infty, \infty)$ when $h \rightarrow 0$ is given as:

$$\lim_{h \rightarrow 0} \int_0^h \frac{1}{h} dx = 1.$$

Eq(2.62) is not a usual function for as $h \rightarrow 0$, the function tends to be singular. And it is defined under the integration when $h \rightarrow 0$. The function is known as the unit impulse function, or as the Dirac delta function.

For explaining the causes and effects, two variables, $\mathbf{r}$ and $\mathbf{r}'$, are employed. Both can take on all values in a certain common interval or region in which the problem is studied. Suppose there is a unit cause, for instance, a unit positive charge, located at $\mathbf{r}'$ in a suitable environment with the given boundary conditions, then the potential (or electrical potential) at $\mathbf{r}$ is a superimposing of the effects from $\mathbf{r}'$ and the boundaries that first were effected by $\mathbf{r}'$. Now, we must to define a function that is used to express density distribution of the unit cause. Again suppose the unit cause is concentrated at $\mathbf{r}'$ and $\mathbf{r}$ is the position at which we want to determine the effects. The characteristics of the distribution are: when $\mathbf{r} \neq \mathbf{r}'$, the density at $\mathbf{r}$ is zero; whereas, when $\mathbf{r} = \mathbf{r}'$, the density becomes infinity. The reason that it tends to infinity is that the intensity of the unit cause is finite, but volume where the unit cause is concentrated is zero. It can be seen from Eq(2.62) that $\lim_{h \rightarrow 0} \frac{1}{h} = \infty$.

δ function, is used to represent the unit cause:

$$\delta(\mathbf{r} - \mathbf{r}') = \begin{cases} 0 & \text{if } \mathbf{r} \neq \mathbf{r}' \\ \infty & \text{if } \mathbf{r} = \mathbf{r}' \end{cases}$$

$$\int \delta(\mathbf{r} - \mathbf{r}') d\mathbf{r} = 1$$

The Green's function, $G(\mathbf{r}, \mathbf{r}')$, is used to express the effect, which is to be determined at $\mathbf{r}$ due to a unit cause concentrated at $\mathbf{r}'$. Then $G(\mathbf{r}, \mathbf{r}')$, as a function of $\mathbf{r}'$, should satisfy:

$$\nabla^2 G(\mathbf{r}, \mathbf{r}') = -\delta(\mathbf{r} - \mathbf{r}') \quad (2.63)$$

Then the Green's function, associated with the linear differential operator ∇^2 and the given boundary conditions, is the solution of Eq(4.61), where Eq(2.63) is equivalent to Eq(2.61) with δ replacing $\phi(y)$.

If superposition is valid, the effects of sum of more than one causes is the sum of the effects due to each of the causes. Let $\delta(\mathbf{r}')$ be the intensity of a uniform distribution of causes over the region $(\mathbf{r}', \mathbf{r}' + d\mathbf{r}')$, then the effects $h(\mathbf{r})$ at $\mathbf{r}$, due to a distribution of causes $\delta(\mathbf{r}')$ over the entire region is:

$$h(\mathbf{r}) = \int G(\mathbf{r}, \mathbf{r}') \delta(\mathbf{r}') d\mathbf{r}'$$

Finally, if the Green's function exists, then the following integral equation is equivalent to the Eq(2.61):

$$y(\mathbf{r}) = \int G(\mathbf{r}, \mathbf{r}') y(\mathbf{r}') d\mathbf{r}' \quad (2.64)$$

2.4 Distributed message systems

At the beginning of this section, an overview of message systems is discussed. Then the general picture of the Distributed Process library (DP), and its interface are introduced.

2.4.1 The notion of parallelism in this work

The type of parallelism that this work is concentrating on is distributed multi-processing. The distributed processing system consists of a set of individual processors that are networked together. Each of these processor has its own (local) memory which cannot be accessed by other processors directly, and there is no commonly, physically shared memory on the network. As the objective “distributed” indicates, there is no centralized control on a network environment. Communications among the processors are handled by messages. The flexibility of this type of parallel processing is that different sequences of instructions can be carried out by the distributed processors on different sets of data. The physical systems usually refer to local area networks that are widely available. To put processors working together to solve a problem needs a software environment that supports synchronization and communication between processes running on distributed processors. There are different approaches to building up such an environment. A message-passing mechanism is usually considered as a low level service that logically connects distributed processors into one.

A distributed environment can be presented to users as program languages or message-passing interface. Commonly used programming languages designed to

handle distributed computing are Ada, occam, and some extended languages having the ability to deal with distributed processing, like concurrent Pascal.

There are many message-passing system packages available. Each operating system or network system provides, at least, one message system. The differences between these message systems can be categorized by levels, abilities, orientation/purposes or some other sense of grouping.

A lowest level message system that can be seen on a network layered-protocol scheme is the Internet Protocol†(IP). And a highest level message system can be used as a programming language like CSP (Communication Sequential Processes). But in general, message systems that we are dealing with for computation reside in neither of these extremes. In this work, distributed Monte Carlo is carried out on a message systems running on the transport layer. And distributed random number generator system is built on a message system available on the session layer.

Some message system have more functions than others. One of the major functions that differentiates message systems is the ability of selecting incoming packets.

† The OSI (Open Systems Interconnection) Reference Model defines seven layers peer-top-peer communication functionalities. Five of out of the seven layers are following. The lowest layer of OSI model is the physical layer. It is responsible for transmitting unstructured bits of information across network wires. An example protocol that resides in this layer is X.21. The next layer is the data link layer. In this layer structured packets are recognized and physical hosts addresses are specified. It has abilities to detect data corruptions which is achieved by computing checksum. The higher layer that lies on the data link layer is the network layer. In this layer processor to processor communication is established or connected. Services that handle routing, packet fragmentation and reassembly, congestion control are provided. In the TCP/IP suite, functionality of IP is equivalent to the network layer of the OSI model. The layer that establishes a reliable communication is called the transport layer. It can handle problems of lost packets, or duplicated packets. Packet ordering is guaranteed in the service of this layer. In the TCP/IP network the functionality of TCP (Transport Control Protocol) matches the specified functionalities of transport layer. However, TCP/IP suite also provide UDP (User Datagram Protocol) which is a unreliable communication protocol. DP (Distributed Process) assumes services of UDP. The last layer discussed here is the session layer. With the services of session layer, simple full-duplex reliable communication is established. A typical package built on this layer is RPC (Remote Procedure Call). By applying RPC, server-client distributed environment can be formed. In this work, the distributed random number system core is built in a RPC server.

A message-passing system that possesses selection abilities simplifies application-level programming. Another function that is available in some message-passing systems is the interrupting message. Its functionality is similar to “out-of-band” data, however it does not flush messages that were buffered earlier. Other functionalities provided by some message systems that make the system more flexible are the mechanism that enables unblocked message operations and unreliable message transmission. Fully reliable message-passing operations on network data transmission are expensive. However, for some applications unreliable message-passing operations may be tolerated and satisfy their needs.

All these distributed means, languages, message interface libraries require explicit parallel programming schemes. Unlike the single-instruction, multiple-datastream model (SIMD) in which parallelism is implicitly given by program constructs that are recognized and interpreted by compilers, distributed programming explicitly construct simultaneously executed instruction. Besides the communication interface, there are other mechanism needed, such as the way to specify partitioning (separated) instructions being evaluated in parallel and the way to synchronize execution among processes.

2.4.2 Distributed Process system

The Distributed Process (DP) system is a message-passing system designed by Arnow[22]. In this section it is discussed as an abstract model. Different implementations are available on different platforms.

DP is a set of functions, organized as a run-time library, that provide interfaces

to non-distributed programming languages, such as C and Fortran to accomplish parallel processing. DP function interface is simple and easy to use. A DP program can be edited into one file that contains all partitions of instructions. It means that initial process and worker processes all run the same program. In fact, all DP processes, primary and secondary are symmetric in the sense that all processes carry out the work load. *dpinit()* is the only function that all distributed processes have to call. It builds the distributed environment. An initial/primary process calls it, and the desired number of processes are spawned. Worker/secondary processes also call it, DP communication channels are established through the primary.

DP message transmission functions are *dpsend()* and *dprecv()*. They have the advantages that their parameters specify source, destination processes, involved message, as well as operation mode and options together. Source and destination processes are named by integers, *dppid*. The process owns an *dppid* and possesses receiving rights. All other processes, including the owner of the *dppid*, have the right to send messages to the integer-named process. All names, *dppid*, are globally known by all the distributed DP processes. *dpsend()*, *dprecv()* are used in all synchronous messages. The constants *DPBLOCK*, *DPNOBLOCK* are typical flags specifying the behavior of the *dprecv()* operation. Mode values, *DPREL*, *DPUNREL* are used to request reliability of data transmission across networks.

DP provides “out_of_band” messages. They are called, by DP jargon, asynchronous messages because the receiving DP routine notifies the user program to process, asynchronously, the “out_of_band” message that has just arrived.

Many non-scientific computation oriented message-passing systems do not support “out_of_band” messages, or some systems support “out_of_band” data of a single ASCII character[23]. DP asynchronous messages can be arbitrarily long.

DP is a low level internet wide distributed computing environment. Its level can be characterized to match functionalities defined in transport layer of OSI Reference Model. Programming in this level, users are responsible for presenting their data correctly from one host to destination hosts. A typical problem is that different byte orders are used in different machine architecture. So data representation has to be taken care of by DP users.

To partition a problem, native language provided conditional and iteration constructs can be employed. Using integers to name DP processes helps to directly apply them as conditional variable in the *if* structure or variable counter in the *for* loop in C language. The same situation also applies to loops in Fortran.

Message operations that have the ability to block calling processes can be programmed in such a way as to synchronize processes. DP does not provide an explicitly mutual exclusion mechanism. *dpsend()* and *dprecv()* are used to achieve this mechanism. abilities.

To meet the demand for high portability, DP is designed as a low level message-passing system. Semantically, message transfer interface merely achieves the primitive message operations needed. High level, useful routines, like *in()*, *out()* in Linda, may be built by programming DP functions. High portability is also achieved by using no environment support. For instance, no daemons are needed.

DP can be implemented under totally on different operating systems or on different machines. Because it is very low level and no environment is required, DP can be built on all possible systems. DP does not necessarily have to run on Unix on TCP/IP protocols. It can be written for other IPC's, PC Desqview, System V IPC or Mach pthread, pthread in Microsoft windows NT. All these platforms provide the same DP interface so application programs developed on one system can ran on other DP systems.

2.4.3 The interface of DP

There are two sets of routines supported by dp library. The primitives are the lower level dp functions which provide only reliable, unreliable, block, unblock message operations. The standard functions, the upper level dp routines, provide more services, such as selection of messages. In the following subsections, the semantics of the two sets of functions are introduced. In chapter 3, one dp implementation on a Unix shared memory environment is discussed, and usage of the dp routines is discussed in chapter 5.

2.4.3.1 DP primitives

`dpinit()` is called when a process wishes to establish distributed a message-passing environment. The process which makes the first call to `dpinit()` is called the primary process. `dpinit()` then, reads a designated host file that consists of the number of distributed processes the computation expects to use; the name of the program that the remote processes execute; the name of the hosts on which the

program runs; the directories to which output should be written and possibly passwords that are required by remote hosts. During the reading of the host file, the remote processes are spawned. Each remote process, a secondary process, will call `dpinit()` to communicate with the primary and establish message channels. `dpinit()` differentiates primary and secondary by looking at the parameters, `argc` and `argv`, it receives. `dpinit()` will return an integer indicating the number of processes spawned.

```

1
2      int      argc;          /* arguments of main() */
3      char     **argv;
4
5      int
6      dpinit(&argc, argv);
7
```

`dpexit()` is the function that each dp process calls to terminate the connection to the distributed environment it has established. The parameter of the function call is a message which will be printed on the log file that `dpinit()` created.

```

1
2      void
3      dpexit(msg)
4      char  *msg;
5
```

`dpgetpid()` should be called by a process that has already called `dpinit()`. Once `dpinit()` is returned, dp process identification numbers are set. Each dp process makes a call to `dpgetpid()` to fetch its own id, which is an integer.

```

1
2      int
3      dpgetpid();
4
```

`dpsend()` sends a message, data, to a dp process denoted by `dest`. The last parameter of the function carries the operation mode and option. An operation

mode could be block or unblock, reliable or unreliable. An option may be used to indicate how the message is delivered, synchronously or asynchronously. The latter needs special care on the receiving side.

```

1
2     int
3     dpsend(dest, data, nbytes, flags)
4     int  dest;
5     char *data;
6     int  nbytes;
7     int  flags;
8
```

`dprecv()` is the operation to receive incoming synchronous messages. The caller also receives the source process id.

```

1
2     int
3     dprecv(src, data, limit, mode)
4     int  *src;
5     char *char;
6     int  limit;
7     int  mode;
8
```

`dpgetmsg()` is used to receive asynchronous messages. It is called by an interrupt handler when the handler is triggered by the operation system informing that a message is arrived.

```

1
2     int
3     dpgetmsg(src, data, limit)
4     int  *src;
5     data *data;
6     int  limit;
7
```

`dpcatchmsg()` is a signal catcher routine which is installed by using, under Unix, the `signal()` function call. A dp process is programmed in such a way that it can receive a signal generated by the operating system whenever a message arrives to the host. The catcher function then calls `dpgetmsg()` to physically to fetch

the incoming message. The message may be stored in any data structure that the application program desires.

```

1
2     typedef void      (*FUNCPTR)();
3     #define NULLFUNC  ((FUNCPTR)0)
4
5     FUNCPRT
6     dpcatchmsg(f)
7     FUNCPTR f;
8

```

2.4.3.2 Standard DP functions

These functions are generally used by users' programs to interface with the dp environment. Many functions take the same number of parameters with the same meaning as the primitives. The one which differs from the primitive is DPrecv(). It has an ability to select the possible sources a message is sent and messages types it is expecting. The syntax of all the functions are listed below with no further explanation.

```

1
2     int
3     DPinit(argc, argv)
4     int  *argc;
5     char **argv;

```

```

1
2     int
3     DPexit(msg)
4     char *msg;
5

```

```

1
2     int
3     DPgetpid()
4

```

```

1
2     int
3     DPsend(data, nbytes, mode, dest, type)
4     char *data;
5     int  nbytes;
6     int  mode;

```

```

7      int    dest;
8      int    type;
9

```

```

1
2      typedef struct {
3          int    r_size;
4          int    r_type;
5          int    r_src;
6      } SDPRESULT;
7
8      SDPRESULT *
9      DPrecv(data, lim, mode src, type, ..., lastsrc)
10     char    *data;
11     int      lim;
12     int      mode;
13     int      src;
14     int      type;
15     int      lastsrc;
16

```

```

1
2      FUNCPTR
3      DPcatch(f)
4      FUNCPTR f;
5

```

```

1
2      typedef void      (*FUNCPTR)();
3      #define NULLFUNC  ((FUNCPTR)0)
4
5      FUNCPR
6      DPcatchmsg(f)
7      FUNCPTR f;
8

```

3

Implementing The Distributed Process System

The message passing system needed for this work is one that is suitable for Green's Function Monte Carlo, (GFMC). The requirements of such a system are met by the DP library developed at Brooklyn College. In this work, two such DP systems were written and used to develop distributed MC and random number generator systems.

3.1 Small DP systems

Part of my interest in distributed computing is in building distributed message systems. Two small DP systems, the Concurrent Distributed Process system and the pthread parallel Distributed Process system, were built on completely different environments. They are both "small" in the sense that the concurrent DP was designed along with the development of distributed one dimensional Green's Function Monte Carlo algorithm. It helped to understand a distributed computing approach to GFMC. The concurrent DP runs on a single host under the Unix operating system. The pthread parallel DP was developed on the KSR-1, a multi-processor supercomputer. Distributed Monte Carlo programs were run on both small DP's, and performance was compared comprehensively.

There are several good reasons to develop and use such systems, especially at the beginning of conducting distributed computing research.

First, such a system is reliable in the sense that the liveness of the system is

transparent to users during processing application-level programs. In contrast to a message system running on a network, in this system, the developer can avoid spending time struggling with sporadic, unrelated, or probably unfamiliar problems in a network, and hence can concentrate on addressing application issues. Further, because interprocess communication is inherently reliable, complications that result from low level communication problems are less likely to occur. Given this isolation from data communication problems, the developer can assume that most of the program's problems occur at the application level.

Secondly, it is easy to control the distributed/concurrent processes on a one-host system. Debugging a distributed application is always difficult. Debugging application programs running on a small model of a distributed system that is implemented under Unix can be fairly easy, because the operating system provides useful and convenient utility programs. The status of running processes can be found by using system utility programs, and all running processes are under the users' control. For example, by using the signal utility command, the concurrent processes can be debugged in an interactive way. Another example is that, in some network message systems, cleaning networkwide resources, such as suspended processes, network ports, is needed when a distributed application program is terminated unexpectedly. This is not a problem in a small system because of the simple system maintenance.

Next, the system is scalable. The problem in my hands is characterized as stochastic simulation. So testing sensitivity of the developed algorithm in term

of processor scaling is one of the major task. Appropriate issues of scaling and behavior also have to be studied. The Unix operating system provides the ability to support many processes running at the same time. It is true that a concurrent DP program may overload the system when many concurrent processes are running, but, for test purposes it is worth to explore the general picture of a newly developed algorithm.

Finally, for me, a DP implementor and application programmer, knowing both the nature of the application and the way parallelism is achieved is always helpful. It will help to develop a distributed Monte Carlo calculation scheme and improve the existing message system. This chapter completely documents the implementation of the concurrent DP. It is hoped that the discussion will help new DP users better understand the nature of DP.

3.2 Concurrent distributed process system

In this section, I discuss the implementation of one variation of the DP library on a single host. Because its parallelism is virtual and is achieved by sharing one processor, it is called a concurrent distributed process system. As I mentioned earlier, this model is used for development purposes. When the distributed random number generator system (chapter 4) was under development, the system was often being used to do the tests. It was found to be very convenient working on the small system. Concurrent DP does not increase the computational ability of processing data, but it is equipped with all the features defined in the DP documentation.

Concurrent DP consists of up to five levels of program components, name-

ly Interprocess Communication resources allocations, queue primitives, concurrent mechanism, System V-like msg queue function/user interface and DP level message operations. Fig. 3.1 shows the overall design of the system. The complete system is reliable. It has been used about one year and was found flawless. The system has also been ported onto the Unix operating system running on a KSR supercomputer at Cornell, DEC Ultrix at Graduate Center/CUNY, IBM AIX at Cornell and SCO System V-based PC computers.

Message System

Shared Memory DP Architecture

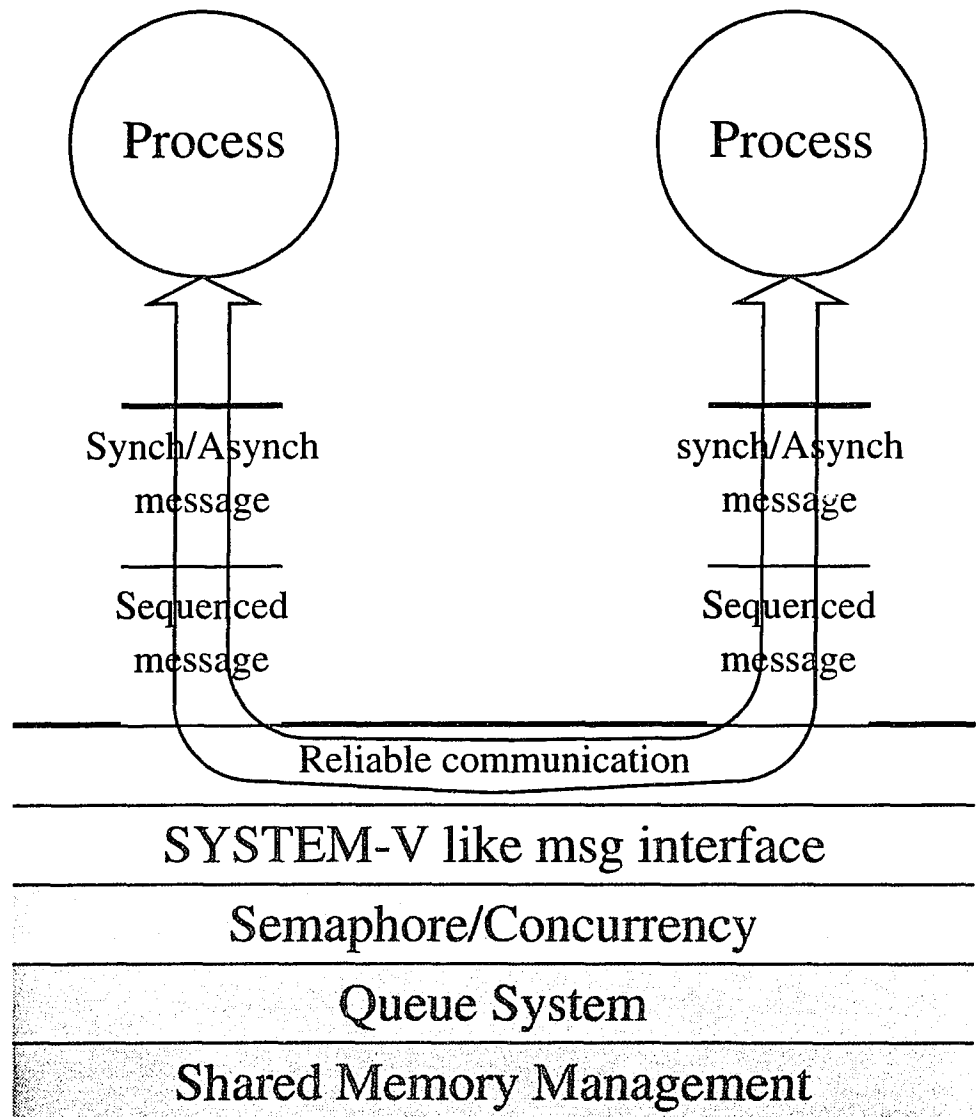


Figure 3.1 Architecture of concurrent DP

3.2.1 System limitations

The early work of the single host DP program was written on System V, and it relied upon system provided *msg* resources. Later I found that it could not meet real applications' needs. The major problem was the limit of *msg* queues.

The limitations of System V *msg* are: (1) Message queues are defined in the system kernel and are limited to 2KB each in a normal configuration. With that limit, a message in the queue can take only tens or hundreds of bytes. (2) The number of queues is limited and the maximum number of messages systemwide is also limited to about 40-60 (messages). Both limits made my early single host DP much less satisfactory, and is no use for non-trivial Monte Carlo computations. Thus, I started to implement my own message system.

The message system of the concurrent DP is implemented using System V shared memory. The maximum size of a shared memory segment is defined in Unix system design. SunOS 4.1.3 allows 8MB of shared memory which is large enough for my application's need. With this large amount of shared space, I defined message size as 1KB. So there are thousands of messages available to the concurrent DP system, and therefore, virtually no limit on the number of messages. This provided flexibility for scaling distributed processes during application level computation. In general, the more processes involved in a computation and the larger quantity of data need to be transported, the more messages will be used. With the 8MB message pool and 1KB maximum message size, normally there is no need to worry about lacking messages to use.

However, before starting implementing this system, some design ideas have to be made clear. Unlike other scientific computations, distributed Monte Carlo requires large volume of data to be transmitted. Here we are talking about hundreds of messages periodically being sent by a process, and possibly tens of such processes are required in a computation. One difficulty is that because the message media is part of the concurrent DP system, if the 8MB message pool is exhausted, a dead lock may occur in the system. It will happen if many processes employ message send operations at the same time. In other DP implementation the physical message media may not be a part of the DP systems. It assumed that good capability is available in the target network to buffer messages.

The mechanism of data central flow may explore different issues in different platforms. But to conduct Monte Carlo calculations at the user level, they are related to the same problems. In network Monte Carlo computing, too many messages employed at the same time causes heavy data collisions. Therefore, they slow down the speed of computation or decrease utilization of network resources. The worst case is that it may paralyze the ongoing computation. In a concurrent DP Monte Carlo calculation, the same problem in a computation will immediately lock the system if the message pool is exhausted. To appropriately use any DP system to conduct Monte Carlo calculations, one has to deal with some mechanism that slows down message insertions to the DP system at the user level.

In concurrent DP implementation, to prevent the system from involving a dead lock, some mechanism must be considered. (The dead lock is a unique characteristic

in concurrent DP.) The implemented protection can only prevent the system from getting stuck, and system throughput may be sacrificed. We can imagine that once the computation has entered in to the stage in which the pool is exhausted, the entire computation is going to be very slow for a long period of time due to the message system awaiting available messages, especially when the user level program conducts blocked message operations. Again, I will assume that the system productivity (or throughput) is not a concern of developing my DP library.

3.2.2 Interprocess Communication resources allocations

Interprocess Communication (IPC) on the Unix operating system is the basic requirement for constructing the concurrent DP system. The mechanism by which IPC is created and used is discussed in this section. Some C program fragments are provided to help the discussion. The code is not contiguous; only the critical parts are presented.

A process in Unix contains text or instructions, stack space and data section(s) as well as other entities, such as environment variables. A child process can be spawned by a system *fork()* call. The newly created process shares the text part with its parent process, inherits the data and environment from its parent and obtains its own stack area. Other system resources like files, pipes, semaphores and the shared memory, obtained by a parent process are all shared with its child process(es), or are all automatically available to spawned process(es).

To make concurrent DP work, some system resources have to be allocated by a running process and recognized by some or all other involved processes as commonly

shared communication/synchronization media. The ability of child processes to inherit resources from its parent is utilized in this design. In the single host DP implementation, the shared memory system resource is used as the major IPC resource. One characteristic of shared memory that is very useful is that it is allocated as virtual space, and so the size of it can be very large. It immediately solves the two limitations discussed above. Another type of system IPC used in concurrent DP to achieve mutual exclusion on accessing shared memory media is a set of semaphores.

As discussed in Chapter 2, DP secondaries are all created by a primary process. Concurrent DP does the same. However, before spawning processes, the primary sets up commonly shared media, a number of shared memory units which I call “chunks” and a set of semaphores. All processes spawned by the primary inherit the shared memory and semaphores.

The shared memory is organized as a pool of messages. Data that are written into the shared memory by any process can be read by any other processes. Before worrying about which process has the right to access the shared memory, let’s see how the message pool is allocated and organized.

The data structures of desired shared memory are defined as follows:

```

1      #define  chnksize          1000
2      #define  datasize         1024
3
4      typedef  struct node node, *list;
5      struct  node {
6              node      *next;
7              char       data[datasize];
8      };
9
10     typedef  struct chunk {
11             node        pool[chnksize];

```

```

12      } *p_chunk, chunk;
13
14      #define  LNULL          ((list)0)

```

Code fragment 3.1

The above fragment defines a chunk of memory of about 1MB in which an array of 1000 nodes, each of it 1024 characters long, is set up. Here I assume that the system allows allocating 1MB of shared memory at one time†.

The potential usage of the defined structure is shown by the following figure.

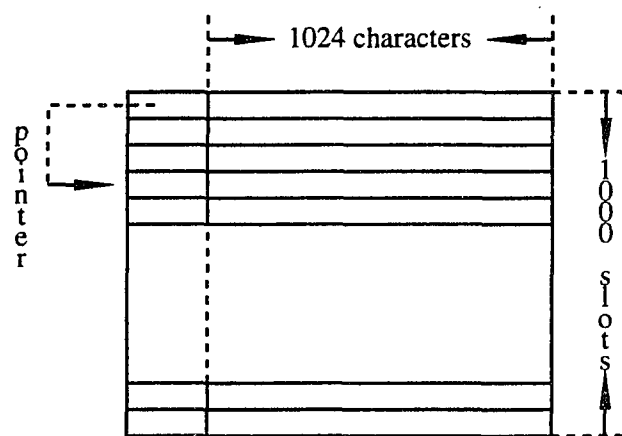


Figure 3.2 A memory chunk.

Each node of the array can hold a piece of message. Here, the message refers to the data field in the above definition. An array of consecutive 1024 characters is

† Earlier versions of SunOS did not allow allocating such a large chunk at a time even though the manual indicated otherwise, so an appropriate size of array pool may be 128, rather than 1000. (The total allocation would be 131072 byte. `chnksize = 128` is experimental. The next reasonable scale `chnksize = 192` was tried, but it was rejected by the system.)

the smallest entity that can be identified by all process in exactly the same way. In terms of Fig. 3.1, these nodes reside in the lowest level of the design. They are called messages for they will carry user data across processes. When the messages pass through the layers, some protocol information or header(s) will be added. Actual user data will not be exactly 1024 characters in each message.

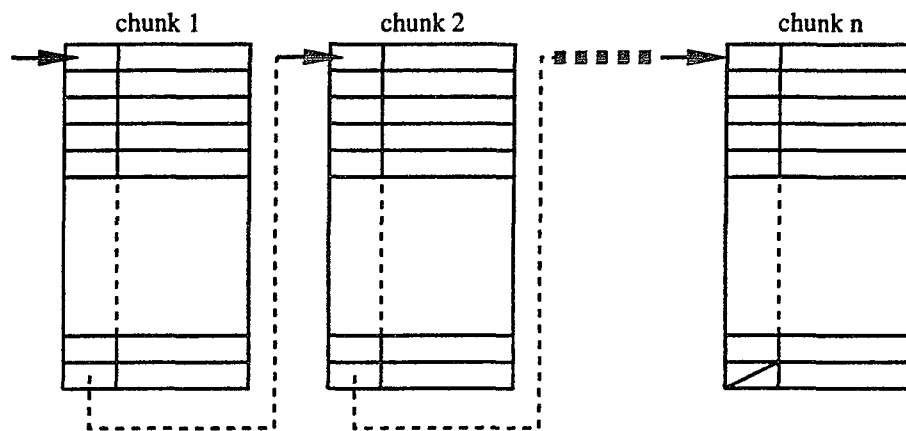


Figure 3.3 Chained memory chunks.

The system call to allocate shared memory is `shmget(key, size, perm)`, where `key` is required to be unique to all `shmget` calls, `size` refers to the number of bytes the call requires, `perm` indicates read/write permissions. `shmget()` returns an identification number, `shmid`, for the granted memory if no error arises. Following the call to `shmget()`, a system call `shmat(shmid, shmaddr, flags)` has to be invoked. Here `shmid` is obtained by the earlier `shmget()` call. `Shmaddr` parameter in `shmat()` allows the caller to specify a memory address in which the shared memory will be attached. `flags` is the indicated option(s). The returned

value of `shmat()` call is a pointer to `char` type, which points to the allocated shared memory.

The use of the above shared memory is to link all nodes into a list. Figure 3.3 shows the initial state when all of the nodes are linked.

The `list` data type is given in Line 4 in code fragment 3.1. A chunk of 1MB shared memory is allocated by a sequence of `shmget()` and `shmat()` calls and a pointer returned by the call to a `shmat()` is cast into `p_chunk` type defined in Line 12 in code fragment 3.1. A function `initchnk()` that takes such a pointer `chnkptr` and physically links the nodes of the referenced chunk of shared memory pointed to by `chnkptr` is listed as following:

```

1      list
2      initchnk(chnkptr)
3      p_chunk chnkptr;
4      {
5
6          int      i;
7          p_chunk p = chnkptr;
8
9          for (i=0; i<chnksize-1; i++)
10             p->pool[i].next = (list) &(p->pool[i+1].next);
11
12         p->pool[chnksize-1].next = LNULL;
13         return ( (list) p );
14     }
15
```

Code fragment 3.2

To allocate n MB of shared memory, n sets of `shmget()`, `shmat()` and `initchnk()` calls have to be made in that sequence. The final goal of allocating these chunks of memory is to link them all together as a single list that is managed by the concurrent DP system. (see Fig. 3.3) The function `ipcinit()` which puts all the above discussion together will be presented shortly. Because `ipcinit()` allo-

cates several other system resources besides the large chunks of memory, the other required IPC's shall be discussed first before the listing of the `ipcinit()` function is given.

When the allocated chunks of shared memory have been constructed into a linked list, the head of the list should be managed in such a way that every process can access it, and that these accesses should be mutually exclusive. Making the head of the list accessible to all processes is not a problem because a single variable allocated in a piece of shared memory can be used. The piece of shared memory is called `p_share` and the head of the list residing in the memory is called `mallocp`.

The following definition is used to declared `p_share`:

```

1      typedef      struct      share share;
2      struct      share {
3          list mallocp;
4      };
5
6      static      share      *p_share;
7
```

Code fragment 3.3

Up to this point of the development, the shared memory, list of free nodes , relation between `p_share` and the head of the node list can be plotted into a diagram as in Figure 3.4. Details of constructing this structure will be discussed in the code fragment 3.4.

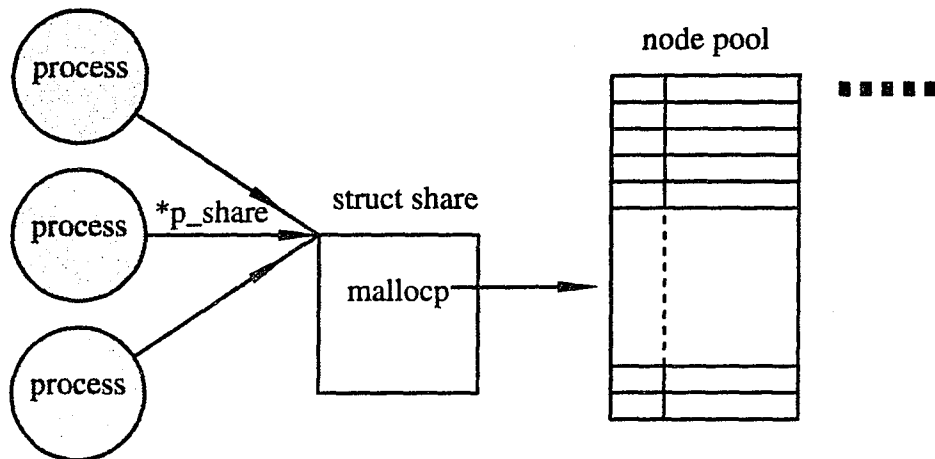


Figure 3.4 Nodes shared by processes.

The variable `p_share` is used to hold the address of `share`, a piece of shared memory. The structure residing in the shared memory that has a member `mallocp` to a list, or more precisely, a pointer to a node of 1024 characters defined as data in Line 7 in code fragment 3.1. Clearly, `mallocp` is subject to be accessed synchronously. For the purpose of synchronization, a semaphore is used to achieve mutual exclusion.

Concurrent DP manages a system table in which message queues and their states are kept. The table is another object needed to be commonly shared by all processes. Discussion of the table involves topics of higher layer design. Here a brief description of what is needed to handle the table is presented. Like `p_share`

for the head of the list, another piece of shared memory is used to hold an address to a structure. This structure is the center of the queue system in the second layer of the system; see Fig.(3.1). The variable that is used to hold the address of the structure is p_qtab and its type is pointer to qentry which will be discussed in the next section.

Both p_share and p_qtab are allocated in the same way as that of the memory for a linked list. The code that allocates all required shared memory is written as the function ipcinit() below:

```

1      #define MAXCHNKS      8 /* or some integer < 8 */
2      static p_chunk      chunktab[MAXCHNKS];
3
4      ipcinit()
5      {
6          int      i;
7          list      q = LNULL;
8
9          p_qtab = (qentry *) ipcgetptr("SHM qtab", sizeof(qentry),
10                                     IPC_CREAT);
11          p_share = (share *) ipcgetptr("SHM share", sizeof(share),
12                                     IPC_CREAT);
13
14          chunktab[0] = (p_chunk) ipcgetptr("SHM chunk",
15                                     sizeof(chunk), IPC_CREAT);
16          p_share->mallocp = initchnk(chunktab[0]);
17          for (i=1; i<MAXCHNKS; i++) {
18              chunktab[i] = (p_chunk) ipcgetptr("SHM chunk",
19              sizeof(chunk), IPC_CREAT);
20              q = initchnk(chunktab[i]);
21              chunktab[i-1]->pool[chnksize-1].next = q;
22          }
23
24          i = 0;
25          q = p_share->mallocp;
26          while (q) {
27              i += 1;
28              q = q->next;
29          }
30          fprintf(stdout, "TOTAL nodes:  %d\n", i);
31
32          semqtab = sem_create(SEMQ, 1, 1);
33          semalloc = sem_TST(SEMA, i);
34          return(0);
35      }

```

Code fragment 3.4

The function `ipcinit()` allocates not only shared memory, but semaphores as well (lines 32-33). The purpose of using these semaphores will be discussed in section 3.2.4.

The listing of `initchnk()` was given as code fragment 3.2. `ipcgetptr()` calls two utility functions `shmget()`, and `shmat()` subsequently and prints some messages to indicate success of the function calls. Several parameters of the calls to `shmget()` and `shmat()` were determined internally as constants by `ipcgetptr()`. For instance, `perm` parameter in `shmget` call is set to 0600 which indicates that the owner of the memory has the right to read and write the memory.

`ipcgetptr()` takes three arguments: a string for output indicating which piece of shared memory has been allocated for diagnostic purposes; the size of shared memory required for `shmget()` system call and creating the status for `perm` in `shmget()` call. Lines 9 and 11 are used to allocate `p_share` and `p_qtab`. Details of usage of these structures will be given in section 3.2.3, 3.2.4 and 3.2.5. Lines 14 to 18 repeatedly allocate shared memory chunks. Line 16 initializes the head of the list. Lines 17 to 22 link all shared memory chunks into a list. Lines 24 to 29 travel the entire list and count available nodes. `Chunktab` is an array used to save shared memory identification numbers, so later they can be used to release the allocated memory.

At this point, a summary can be made to explain how a piece of data can be delivered across processes. Initially, the primary process calls `ipcinit()`. Then it spawns secondary processes. All IPC resources are shared by the primary and

secondaries. `p_share` is a pointer that is inherited by all the secondary processes. Through the pointer, `mallocp`, the head of the system maintained linked-list can be accessed to remove messages from or return messages to the list by all processes. The `mallocp` initially points to the first node of the list. It will dynamically point to any possible node in the list. A value in `mallocp` is a virtual address that every process reads as the same. Through a pointer to the virtual address, a node of the list, a sequence of 1024 characters can be written into or retrieved from as a message. (see Fig. 3.4)

When the concurrent DP is compared with a network version of DP, the lowest level, the shared memory, can be considered as providing the *physical* and the *data link layer* functionalities of the network. Now all the nodes in the list can be accessed by the processes which share the virtual memory. There are two remaining problems. One of which is how to synchronize accesses. The other is how to handle the situation in which the list is exhausted. Dealing with these situations relates to dead lock avoidance. Solutions for both problems are embodied in the use of semaphores. Semaphores will be used to build the concurrent environment in which multiple processes can run. In the section 3.2.4 semaphore and concurrence will be discussed.

3.2.3 Queue systems

In this section the second layer of concurrent DP, the Queue System, is discussed. Without the mechanism of synchronization, the free list that was allocated in the shared memory can not be used by more than one process. The queue system being

presented in this section is a set of queue primitives. There is no direct uses of these queues in this layer. But it can be imagined that these queues can be used in the fourth layer of the design since the third layer provides concurrence ability. (see Fig.(3.1))

Queues of the concurrent DP are built right into the system table, `qtab`. The table captures all information of the queues and it is managed in such a way that queues are dynamically deleted and inserted. `qtab` resides in a piece of shared memory so every process can directly access it. (see Line 9 in code fragment 3.4) The design idea is that every distributed process should have the ability to create queues in the DP system, possibly in response to dynamic process spawning. However, creating or deleting queues must be handled in a very low level routine. There is no interface for users to access the queues and the system table. (see chapter 2, the DP user interface routines.) Queue primitives in this level are the simulation of IP layer services in the network. Through these queues, *physical* connections between concurrent processes are built, and structured *frames*, each of which has 1024 consecutive characters, are recognized.

In the following, the data structures of system queues and table are discussed. Two functions `allocfix()` and `freefix()` are used in this section, while their listings will be given in 3.2.4 since their uses involved semaphore operations.

In each data field of a node of the free list, a `msgnode` is built. The 1024-character field (`datasize=1024`) is further divided into two fields containing a message and a pointer pointing to next `msgnode`. Code fragment 3.5 shows the `msgnode`

structure.

```

1
2  typedef      struct  msgnode msgnode, *p_msgnode;
3  #define      msgsize  datasize-sizeof(p_msgnode)-3*sizeof(int)
4
5  struct      msgbuf {
6              int      mtype;
7              char      mtext[msgsize];
8              int      msnd, mrcv;
9  };
10
11 struct      msgnode {
12             struct msgbuf  info;
13             p_msgnode      follow;
14 };
15
16 #define      MNUL      ((p_msgnode)0)
17

```

Code fragment 3.5

Line 2 in code fragment 3.5 defines `msgnode` data type. Line 3 defines the size of each msg, `msgsize` which is a subtraction of an architecture address size and three integer size from 1024 (`datasize`). The `msgbuf` structure is defined in Lines from 5 to 9. And the `msgnode` structure is define from lines 11 to 14.

To make the structures used in this level clear to readers, a diagram is provided to show the overlapping between the structure node, a slot of memory in the free list, the structure `msgnode`, a node of a queue, which will be defined shortly and the structure `msgbuf`, a message that will be used to carry higher level data.

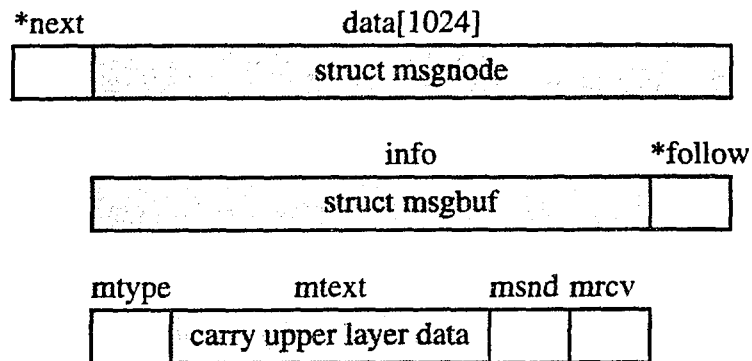


Figure 3.5 message structure residing in a node

The usage and reason why `struct msgbuf` is defined that way will be discussed in the next section. In Fig. 3.5 record fields are provided to indicate the memory boxes. Each system queue is defined as two pointers of `p_msgnode` data type. It looks like the following diagram:

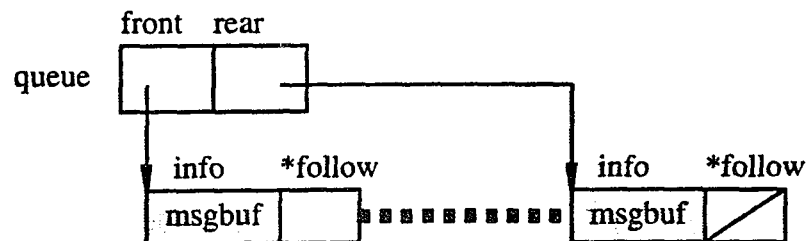


Figure 3.6 A message queue

A node of a queue is a structure of `msgnode` which is residing in the data field, that with the field `next` is a slot in the memory pool. The structure `msgnode` has two fields: `info` and `follow`. `info` is used to carry a structured message, `msgbuf`, and `follow` is a pointer to another identical structure. A slot of 1024 characters in the memory pool is either linked in the free list that belongs to the DP system or lined in a queue as a node that carries upper level data across distributed processes.

The following defines system queues and the system queue table structure.

```

1
2     #define      MAXQS      256  /* or whatever */
3
4     struct      queue {
5         p_msgnode front, rear;
6     };
7
8     typedef      struct sysqs qentry;
9
10    struct      sysqs {
11        struct {
12            queue q;
13            int present;
14            int semid;
15            int semnum;
16        } qtab[MAXQS];
17    };
18
```

Code fragment 3.6

Line 2 in code fragment 3.6 defines the length of the queue table. A queue, in Line 4 to Line 6, is defined as two pointers pointing to the front and rear of a queue. Each node of the queue is called `msgnode`, given in the code fragment 3.5. A data type `qentry` is defined for the system queue table in the rest of code fragment 3.6. The table can be used to contain total `MAXQS=256` queues. In each entry of the table, besides the fields `q` and `present` that record individual queue's information,

two fields `semid` and `semnum` are used to store semaphore information. In the other words, every queue in the system has its own semaphore.

This layer provides eight primitives. `createq()` and `deleteq()` functions used to insert queues into and delete queues from the system queue table. `enrear()` inserts nodes into a queue at the tail of the queue. `defront()` removes nodes from a queue at front. Normally the system queues are operated in FIFO fashion. However, one function `debyfunc()` is provided for occasionally removing nodes by some criteria. One boolean function `qisempty()` is provided for convenience to test emptiness of a queue. `qidentity()` is responsible for the validating of a queue. Finally, a primitive `walkqueue()` is used to check the linkage of a queue. In this section, `createq` and `enrear()` functions are listed for help in understanding the mechanism at work in this layer.

```

1
2     #define    qfront    q.front
3     #define    qrear     q.rear
4
5     static     qentry    *p_qtab;
6     static     int       qtabptr = 0;
7
8
9     createq()
10    {
11        register int    t, i;
12
13        t = qtabptr % MAXQS;
14        for (i = t+1; p_qtab->qtab[i%MAXQS].present; i++)
15            if (i % MAXQS == t) {
16                sprintf(errmsg,"%s\n",
17                    "createq: qtab full");
18                return(-1);
19            }
20        qtabptr = i % MAXQS;
21
22        p_qtab->qtab[qtabptr].qfront = p_qtab->qtab[qtabptr].qrear
23                                     = MNULL;
24        p_qtab->qtab[qtabptr].present = 1;
25        qtabptr += 1;
26
27        return( qtabptr - 1 );
28
29
30
```



```

31     }
32

```

Code fragment 3.7

Line 5 in this code fragment defines the system queue table as a pointer which is obtained from a piece of shared memory. The function that allocates the shared memory is given in line 9 of code fragment 3.4. The table is initialized before `createq()` is called: the `present` field for all entries of the table are set to 0; `qtabptr` is defined as the queue table entry index and initially is set to 0 in line 6 in code fragment 3.7. Line 16 to Line 20 searches for an empty slot. If one is found, the loop terminates; otherwise, it returns to the caller indicating the the table is full. The rest of the code sets appropriate values to the table entry allocated to the caller and update the table index pointer. The fields reserved for semaphores will not be filled until these queues are actually assigned to concurrent processes in higher level functions.

Function `enrear()` is provided in code fragment 3.8.

```

1
2   enrear(qid, xmsg)
3   int    qid;
4   struct msgbuf *xmsg;
5   {
6       p_msgnode  rtn;
7       if ( qidentity(qid) < 0 ) {
8           sprintf(errmsg,"enrear:  invalid queue id");
9           return( -1 );
10      }
11      if ( (rtn = (p_msgnode) allocfix()) == MNULL ) {
12          sprintf(errmsg,"%s %s \n",
13              "enrear:  since allocfix() returns:", errmsg);
14          return(-2);
15      }
16
17      if (bcopy( (char *) xmsg, (char *) &(rtn->info),
18          sizeof(struct msgbuf)) < 0) {
19          sprintf(errmsg,"enfront:  memcpy failed");
20          return (-1);

```

```

21     }
22
23     if (p_qtab->qtab[qid].qfront == MNULL) {
24         rtn->follow = MNULL;
25         p_qtab->qtab[qid].qfront = p_qtab->qtab[qid].qrear
26                                 = rtn;
27         return(0);
28     }
29
30     rtn->follow = p_qtab->qtab[qid].qrear->follow;
31     p_qtab->qtab[qid].qrear->follow = rtn;
32     p_qtab->qtab[qid].qrear = rtn;
33     return(0);
34 }
35

```

Code fragment 3.8

`qidentity(qid)` in line 7 in code fragment 3.8 checks whether `qid` is a valid queue table index in the system queue table. In line 11 `enrear()` calls `allocfix()`, requesting a node from the system maintained free list. The function `allocfix()` is given in the next section. A `(-2)` returned by `allocfix()` indicates the free list is empty; and `(-1)` indicates some other errors. Lines 23 to 28 insert `xmsg` into the required queue and make the queue head point to the queue rear if the queue is empty; otherwise, lines 30 to 32 are used to insert the `xmsg` at the rear of the queue `qid`.

3.2.4 Mechanism of mutual exclusion and concurrence

The semaphore is one of the Unix system provided mutual exclusion mechanisms. It is another system resource required for implementing the concurrent DP. Usually, a semaphore `s` is defined by an integer associated with two primitive operations, `wait(s)` and `signal(s)`. A process calls `wait(s)` to obtain the semaphore `s`. The call is returned if the process acquired the semaphore `s`; otherwise, it enters a wait state managed by the operating system, or it is suspended. In general, only one

process can acquire a semaphore. It is possible that multi-processes are waiting for an occupied semaphore. The process which has the semaphore s calls *signal(s)* to release it. A *signal(s)* call will wake up a process waiting for the semaphore s . No operation of consequence will take place if no processes have been waiting for the semaphore s when a *signal(s)* was issued. A combination of employing *wait(s)* and *signal(s)* can be used to either synchronize events in different processes or protect critical regions. Unix System V provides semaphores that are organized a little differently from the usual ones. It provides a set of integer semaphores associated with (or structured as) a semaphore identifier. An individual integer semaphore can be referred to by a subscript of the semaphore identifier. Semaphore operations consist of a sequence of semaphore primitives, like *wait()* or *signal()* with possibly other operations on specific integer semaphores in the semaphore set. These operations are guaranteed to be executed atomically by the system kernel.

Like shared memory, Unix permits many processes, even unrelated ones to create or access one semaphore with the same semaphore request key. Only one process will succeed in creation. Unfortunately, there are two separate system calls to create and initialize a System V semaphore. Because of the race condition that more than one process issues the same sequence of semaphore creation and initialization calls, some care is needed to ensure correct semaphore creation and initialization. However, in concurrent DP, to avoid this complication, the semaphore resource is allocated in the primary before the secondaries are spawned. The situation is like that of allocating shared memory. Semaphores obtained by the primary are inher-

ited by all its created processes. Thus semaphore creation and initialization is done by the primary and knowledge of the semaphore is passed to secondaries, so the race condition is avoided.

There are two major reasons to employ semaphores in concurrent DP. One of them was mentioned earlier; the head of the free list and the pointer that points to the queue table need to be protected so that accesses to each variable can be serialized. The semaphore used for protecting the free list has to be treated a little differently because the situation where an exhausted free list is present has to be taken care of.

The other use of semaphores is to synchronize accesses to each individual queue. The goal of designing a queue system is that for each queue only one process which possesses the queue has the right to remove items from the queue; all other processes have the right to insert items to the queue. Therefore, for each of these queues, a semaphore should be used to protect a critical region which contains the code manipulating the queue.

System V semaphores are programmable and provide flexibility to construct semaphores in appropriate ways to meet an application's needs. In the concurrent DP, the two different uses have different natures.

To protect the head of the free list and the pointer to the system table from being destroyed by concurrent processes, similar, but not identical mechanisms are used. To protect the system table, only the mechanism synchronizing table accesses needs to be provided. However, to protect the head of the free list, the emptiness

of the list has to be considered. If a process calls `wait(free_list_sem)` and suspends on the semaphore `free_list_sem`, a dead lock may occur since the process is perhaps holding some other system resources for which other processes are waiting. To prevent this from happening, the `free_list_sem` has to be programmed to have the so-called “test_and_set ability”. If there is no item available in the free list, the calling process should not be suspended.

Semaphores associated with message queues can be programmed in such a way that a set of integer semaphores are grouped together into a semaphore identifier. The number of running processes in a single computation in this design is expected to a range from 8 to 128. And in SunOS, the maximum number of semaphore identifiers is 60 (`SEMMNS`). So rather than allocating tens of individual semaphores from the system, a few semaphore identifiers will be used, each with a set of integer semaphores.

There are two operations needed to create a System V semaphore: `semget(key, nsem, options)` and `semctl(shmid, semnum, cmd, arg)`. The first system call, `semget()` is used to create a semaphore in the Unix system kernel. Key is required to be unique to create different semaphores. `nsem` indicates the number of integer semaphores in this semaphore set. The last parameter, `options`, sets access permissions and creating options. An integer `shmid` is returned from `semget()` based upon successful completion. System call `shmctl()` alters a semaphore operation based on the parameter `cmd` on a specific integer semaphore `semnum`. `initval` is used to initiate the semaphore `shmid`.

Function `sem_create()` is provided below with parameter validating steps omitted from lines 11 to 20 in the code fragment:

```

1
2  int
3  sem_create(key, nsem, initval)
4  key_t    key;
5  int      nsem;
6  int      initval;
7  {
8      int    semid;
9      union  semun  semctl_arg;
10     int    i;
11     ...
21     if ( (semid = semget(key, nsem, 0600 | IPC_CREAT)) < 0) {
22         fprintf(stdout,
23             "\nsem_create: (FATAL) %s err=%d",
24             "can't semget failed", errno);
25         exit(1);
26     }
27
28     fprintf(stdout, "\nsem_create: key=%d id=%d ", key, id);
29
30     semctl_arg.val = initval;
31     for (i = 0; i < nsem; i++)
32         if (semctl(semid, i, SETVAL, semctl_arg) < 0) {
33             (void) fprintf(stdout,
34                 "\nsem_create: %s err=%d",
35                 "(FATAL) can't initiate semid", errno);
36             exit(1);
37         }
38     ...
40     return(semid);
41 }
```

Code fragment 3.9

`sem_create` is used to create all semaphores except the one associated with the free list of memory slots. The semaphore used with the free linked-list will be discussed separately. For `p_qtab`, there is only one integer semaphore in the semaphore set, so `nsem` is set to 1; (see Line 32 in code fragment 3.4). For the system queues, `nsem` is set to 20 meaning that there are 20 integer semaphores associated with a semaphore identifier[†]. Lines 21 to 28 handle semaphore creating.

[†] The reason that 20 was picked will be stated later.

When a semaphore is created by this function, it is initialized to integer `initval` for all integer semaphores in that set. `initval` is set to 1 for all `sem_create` calls. Setting a semaphore to 1 indicates that the semaphore is not occupied. Line 30 is used to prepare for initialization. Lines 31 to 37 initialize each independent integer semaphore separately. The `cmd` in the system call is `SETVAL`, meaning “set the value of the *i*th semaphore in `semid` semaphore to `initval`.” Upon success, `semid` will be returned back to the caller. A fatal condition causes the program to terminate.

There are two primitive functions `P()` (`wait()` operation) and `V()` (`signal()` operation) associated with `sem_create()` that manipulate the semaphores which have been created. `P()` and `V()` are listed below. They both use the system call `semop(shmid, sops, nsops)`. `shmid` is the target semaphore of the operations. `sops` is a sequence of semaphore operations. Different sequences of semaphore operations change the behavior of the semaphore. So the interpretation of a sequence of `sops` has to be made when the sequence is applied. `nsops` indicates the number of operations in the given sequence.

```

1      P(semid, sem)
2      int      semid;
3      int      sem
4      {
5
6          int      rtrn;
7          static struct sembuf  op_wait[1];
8
9          op_wait[0].sem_num = sem;
10         op_wait[0].sem_op = -1;
11         op_wait[0].sem_flg = 0;
12
13         again:  errno = 0;
14
15         rtrn = semop(semid, &op_wait[0], 1);
16
17         if (rtrn<0) {
18             if (errno == EINTR)
19                 goto again;
```

```

...
25         }
26         return(0);
27     }

```

Code fragment 3.10

`P()` takes two parameters: an identifier of a semaphore set and a specific integer semaphore within the set. The purpose of this function is to program the system call `semop()`, which is invoked in line 15. As was mentioned, a sequence of `sops` in a `semop()` call determines the behavior of a semaphore. The required `sops` are contained in an array of a record of three fields. The length of the array is the number of operations, in a sequence of operations. Each of the records is one actual semaphore operation. An actual semaphore operation has to specify which particular integer semaphore should be affected by the operation; how the semaphore's value should be modified and should the calling process be suspended if the required operation can not be carried out at once.

In `P()`, `sops` contains only one semaphore instruction. The array of operations used as `sops` is declared in line 7, `op_wait[1]`. Its length is 1, meaning that this `sops` contains only one record of operations. Lines 9 to 11 describe how the operation should be carried out. Line 9 initializes `op_wait[0].sem_num` to `sem`, the specific semaphore in the set `semid`. Next, the value of the `sem_op` field of a operation record may be either a positive or negative integer. The value has two uses in the semaphore operation; to increase or decrease the semaphore value and flags (options) on the operation. A positive value causes `sem_op` to be added to the target semaphore's integer value; otherwise, subtract the absolute value. In line 10,

`op_wait[0].sem_op` is assigned to -1, a negative value. So it will subtract 1 from the target semaphore. Finally, the last field of `op_wait` is assigned to 0 in line 11. A value in the field `sem_flg` may be 0 or some other value. The particular value 0 requires a suspension if the required `sem_op` can not take place immediately during the operation.

Let's go back to interpret the `P()` operation. When `P()` is called, it checks the current semaphore value. If the current value is less than the absolute value of the field `sem_op` assigned in Line 10, the calling process will be suspended which is required by Line 11, (`sem_flg = 0`), until the semaphore value becomes greater than or equal to the absolute value of `sem_op`. It can be understood that a `wait()` operation is achieved here. If the subtraction leaves the semaphore value negative, the case when the current semaphore value is less than the absolute value of `sem_op`, the calling process should enter the wait state.

If the current value is greater than or equal to `sem_op` in Line 10, then the absolute value of `sem_op` is subtracted from the current semaphore value. This operation is exactly the `wait()` operation described earlier. During suspension, if the process is awakened up by some unexpected signal, the process will be suspended again. see Lines 18 to 19.

The following is the listing of function `V()`. It is nearly the same as `P()` operation except that `sops` is named `op_signal` in Line 7 and `sem_op = 1` in Line 10.

```

1      V(semid, sem)
2      int      semid;
3      int      sem
```

```

4      {
5
6          int      rtn;
7          static struct sembuf  op_signal[1];
8
9          op_signal[0].sem_num = sem;
10         op_signal[0].sem_op = 1;
11         op_signal[0].sem_flg = 0;
12
13         again:  errno = 0;
14
15         rtn = semop(semid, &op_signal[0], 1);
16
17         if (errno == EINTR)
18             goto again;
19
20     ...
21     }
22     return(0);
23 }

```

Code fragment 3.11

This operation simply adds 1 to the current semaphore value, where the semaphore is specified by `sem_num` in Line 9.

The semaphore set used to protect the head of the free list is more complicated. In the previous case, each individual semaphore in a semaphore set is an independent integer semaphore. Here, the semaphore used is a set of two integer semaphores. However, they work as a whole. One is the semaphore value, like a regular integer semaphore; the other is a counter counting the number of items remaining in the free list. The code that creates such a semaphore set is given below:

```

1
2     sem_TST(key, initval)
3     key_t   key;
4     int     initval;
5     {
6         int      shmid;
7         union     semun  semctl_arg;
8
9         if ( (semid = semget(key, 2, 0600 | IPC_CREAT)) < 0) {
10             (void) fprintf(stdout,
11                 "\nsem_TST: %s key %d errno %d",
12                 "(FATAL) can't semget for", key, errno);
13             exit(1);
14         }

```

```

15
16     (void) fprintf(stdout,
17         "\nsem_TST: key=%d id=%d \n", key, id);
18
19     semctl_arg.val = initval;
20     if (semctl(semid, 0, SETVAL, semctl_arg) < 0) {
21         (void) fprintf(stdout, "sem_TST: %s err=%d\n",
22             "(FATAL) can't init. counter", errno);
23         exit(1);
24     }
25
26     semctl_arg.val = 1;
27     if (semctl(semid, 1, SETVAL, semctl_arg) < 0) {
28         (void) fprintf(stdout, "sem_TST: %s err=%d\n",
29             "(FATAL) can't init. semaphore", errno);
30         exit(1);
31     }
32
33     return(semid);
34 }
35

```

Code fragment 3.12

Function `sem_TST(key, initval)` is constructed similarly to the function `sem_create()`, which is made up of a sequence of `semget()` and `semctl()` calls. But `sem_TST()` is used to create a set of two-integer semaphores. The `key` parameter of `sem_TST()` specifies the key for creating the semaphore. `initval` indicates the total number of nodes in the free list (see Line 33 in code fragment 3.4). `initval` was determined after the shared memory units were created, and the free list was built. The value of `initval` is variant for some operating systems which allow fewer shared memory units allocated, or if one wants to compile the concurrent DP with any number of shared memory units. The fact of the matter, however, is that the number of nodes in the initial free list has to be passed to `sem_TST()` in order for let the semaphore to work properly.

`semget()` in Line 9 creates a set of two semaphores. The semaphore 0 in the set is used as an item counter in the free list and initiated in Line 19 and line 20. The

semaphore 1 is used as a regular integer semaphore and is initialized to 1 in Lines 26 to 27. In this function, any error arising from system calls causes the program to terminate. Removing a node from the free list with the lock option set with the *wait()* operation may cause a dead-lock. To avoid this, the *wait()* operation used for removing nodes has to be programmed with a so-called *Test_And_Set* ability. That is, if a removing operation finds no node available, *wait()* should not be called. Notice that the free list counter has already been built into the semaphore. Here is the way that the test is performed before executing the *wait()* instruction:

```

1
2     static struct sembuf    op_TST[2] = {
3         0,      -1,      IPC_NOWAIT,
4         1,      -1,      0
5     };
6
7     TST(semid)
8     int    id;
9     {
10         int    rtrn;
11
12     again:  errno = 0;
13         rtrn = semop(semid, &op_TST[0], 2);
14
15         if (rtrn<0) {
16             if (errno == EINTR)
17                 goto again;
18             if (errno == EAGAIN)
19                 return(-2);
20             ...
24             return(-1); /* all other errors */
25         }
26         return(0);
27     }
28

```

Code fragment 3.13

Lines 2 to 5 declare a sops, called *op_TST[2]*, and initialize it. The sops used here is a sequence of two semaphore operations rather than only one discussed in *P()* and *V()*. The two records are initialized in lines 3 and 4. The first column has

0 and 1 indicating that this operation is targeting, first, the 0th integer semaphore of the set, then the 1st integer semaphore of the set. In the second column, there are two -1's. Two semaphore operations require subtractions. In the last column, the top value is a Unix system defined constant, `IPC_NOWAIT`. It modifies the the first semaphore operation so that it will not suspend when the target semaphore value would be negative if the subtraction had taken place. The bottom value in the column indicates that it is a pure `wait()` operation. The semaphore with which this function deals is a set of two integers: the 0th one, a counter of the free list, its initial value could be 8000 if 8 units of shared memory were allocated; the 1st one, an integer semaphore doing the `wait()` operation. (see `sem_TST()`)

The way this sequence of operations works is as follow. The first instruction in `op_TST` tries to subtract 1 from the targeted semaphore value. If the semaphore value is 0 (less than the absolute value of `op_TST[0].op`), the calling process is subject to suspension. However, the option associated with this instruction is set to unblocked, `IPC_NOWAIT`, so, it will terminate the semaphore operation if the first instruction would block the calling process, and the second semaphore instruction in `op_TST` will be ignored. If there is, at least, one free node in the free list, `TST` will decrease the free list counter (the 0th semaphore in the set) by 1, and execute the regular semaphore (the 1th semaphore in the set). After `semop` is call in line 13, some tests are evaluated. If the free list is empty, `semop()` returns `EAGAIN`, and `TST()` returns -2 to indicate the caller encountered an empty list.

On the other hand, the `wait()` operation for inserting a node into the free list

will not cause any problem. So it is coded as:

```

1
2     static struct sembuf    op_NST[2] = {
3         1,        -1,      0,
4         0,         1,      0
5     };
6
7
8     NST(semid)
9     int      semid;
10    {
11        int    rtrn;
12
13        again:  errno = 0;
14
15                rtrn = semop(semid, &op_NST[0], 2) ;
16
17                if (rtrn<0) {
18                    if (errno == EINTR)
19                        goto again;
20                }
21                return(-1);
22            }
23            return(0);
24        }
25    }
26
27

```

Code fragment 3.14

In `NST()` a `sops` is defined and initialized in lines 2 to 5. This sequence of semaphore operations, first, subtracts 1 from the 1st semaphore in the set, and then adds 1 to the 0th semaphore.

`NST()` executes a sequence of two semaphore instructions. The first one calls a `wait()` instruction on the regular integer semaphore. Once the semaphore is acquired by the calling process, the second instruction increases the 1st semaphore, the counter of the free list, by one. Finally, the removing operation and the inserting operation both need the `signal()` operation to release the semaphore. The following is what is needed:

```

1
2     TSTV(semid)

```

```

3      int      semid;
4      {
5          return (V(semid, 1));
6      }
7

```

Code fragment 3.15

The test-and-set semaphore function discussed above is used in `allocfix()` and `freefix()` functions, which are listed below:

```

1
2      char      *allocfix() /* call for only fixed (=1024char) slot */
3      {
4          list    p;
5          int      ret;
6          if ( (ret = TST(semalloc)) == -1 ) {
7              sprintf(errmsg, "(FATAL) Test-And-Set failed");
8              fprintf(stdout, "\n %s", errmsg);
9              exit(1);
10         }
11
12         if ( ret == -2 ) {
13             sprintf(errmsg, "allocfix:  NO space available");
14             return(CNULL);
15         }
16
17         p = p_share->mallocp;
18         p_share->mallocp = (list) p->next;
19         TSTV(semalloc);
20         return((char *) p->data);
21     }
22 }
23

```

Code fragment 3.16

`allocfix()` is called with no argument for it only handles a fixed size allocation of 1024 characters. Line 6 in code fragment 3.16 tests the semaphore `semalloc` by using `TST()` operation. If (-1) is returned, there an unsolvable problem occurred. However, if (-2) is the result of calling `TST()`, `allocfix()` returns a `CNULL` pointer. Notice that in both of the above cases, the `TSTV()` operation is not needed because the semaphore `semalloc` has not been set! If none of the above cases occur, Lines

17 to 20 will be executed. Under the semaphore `semalloc` protection, `mallocp` is advanced and a newly allocated data, 1024 characters, is returned.

The last function in this section is `freefix()`. It takes a parameter which is a pointer pointing to a string of 1024 characters:

```

1
2     void
3     freefix(p)
4     char    *p;
5     {
6         list    q;
7         NST(semalloc);
8         q = (list) (p - sizeof(p));
9         q->next = p_share->mallocp;
10        p_share->mallocp = q ;
11        TSTV(semalloc);
12        return;
13    }
14
```

Code fragment 3.17

In this routine, the `NST()` operation is invoked in Line 7, details of which were discussed earlier. It will not test any conditions, but it increases the free list item counter. The following three lines in the code fragment, lines 8 to 10, do actually return an node back to the free list. At the end of the function, `TSTV()` is called to release semaphore `semalloc`.

3.2.5 System V-like queue primitives

In the second layer of this system, a "physical" connection is established by system managed queues. What is needed in the fourth layer is to build a primitive message system. Its functionality is like that of the TCP/UCP layer in the TCP/IP suite. In this layer, the message channels are bound to concurrent processes and

from this layer the structured frame is used to carry sender and receiver process identifications. The structured frame is built in `msgnode`.

The desired interface is a set of System V message utility functions. The reason it was chosen is that the early concurrent DP system was built on System V and the designer was familiar with those message operations. There are some changes to the primitive message function calls to allow this message system to handle higher level DP messages. For instance, the message structure used by this layer's routines has two more fields `msnd` and `mrcv`, which do not exist in System V messages.

Unfortunately, for this layer and above, the program code becomes too complicated to list all the lines of each function. Only those code segments which, I believe, essential to the design of concurrent DP are discussed.

There are four major primitives in the System V message operation set, namely `msgget()`, `msgsnd()`, `msgrcv()` and `msgctl()`. Two of these functions implemented in this system are listed. `msgget()` will aid in the discussion of how message channels are established between processes and how the concurrence is achieved. `msgsnd()` is listed for its operational connection to the low-level function `enrear()`.

The semaphores used to guarantee that each system queue is accessed synchronously are allocated through `msgget()`. When a message queue is required, `msgget()` is called and a semaphore is allocated to the message queue which will be returned by `msgget()`. Because this system is used to develop Monte Carlo algorithms, a large number of processes are expected to test their scalability so there must be a large number of message queues. System V provides a couple of tunable

constants that regulate usage of system semaphores. `SEMMNS` defines the total number of semaphores available systemwide. It is usually a figure in the range of 40-60. Another constant is `SEMMLS`, which defines the maximum number of allowed integer semaphores associated with a system semaphore identifier. It also is a number between 40-60. To accommodate the DP system demand with the system resources available, two sets of semaphores are used in the DP message system, and in each set there are 20 integer semaphores.

In each DP system, two different forms of message services are provided: synchronous and asynchronous messages. Handling these message types is different from one DP system to another. In some DP implementations, these two forms of messages are transmitted in the same message channel. In this design, however, these two types of messages are transmitted separately. The characteristic of asynchronous messages is to request *immediate* attention. In the lower level, queues are handled purely in the FIFO manner. So an asynchronous message can not be removed at once, except if the `debyfunc()` is used. This is not the only concern about using two different message channels. In the handling of each message, a sequence of semaphore operations must take place. A synchronous message operation within a semaphore activity can slow down an asynchronous message. Two sets of semaphores are used to handle these two forms of message transmission. Each set of these semaphores has 20 integer semaphores. The following is the listing of `msgget`:

```

1    static int    forasync;
2    static int    forsync;
3
```

```

4  msgget(key, msgflg)
5  key_t    key;
6  int      msgflg;
7  {
8
9      int    qid, existing;
10     static int    allocated = 0;
11     static int    keyptr = 0;
12
13     if (!allocated) {
14         allocated = 1;
15         if (allocit(&forsync, &forasync) < 0)
16             return( -1 );
17     }
18
19     existing = (searchkey(key, keyptr) == 0) ? 1 : 0;
20
21     if (msgflg&IPC_CREAT && !existing) {
22         if (msgflg&FOR_SYNC)
23             qid = setsid( forsync, key % SEMSET );
24         else if (msgflg&FOR_ASYNC)
25             qid = setsid( forasync, key % SEMSET );
26         else { sprintf(errmsg,"%s %d","msgget: no flg",
27                     msgflg);
28                 return(-1);
29             }
30         createdkey[ keyptr++ ] = key;
31     }
32
33     if (msgflg&IPC_CREAT && msgflg&IPC_EXCL && existing) {
34         sprintf(errmsg,"%s %d","msgget: existed key", key);
35         return(-1);
36     }
37
38     if (msgflg&IPC_CREAT && !(msgflg&IPC_EXCL) && existing) {
39         if (msgflg&FOR_SYNC)
40             qid = getqid( forsync, key % SEMSET );
41         else if (msgflg&FOR_ASYNC)
42             qid = getqid( forasync, key % SEMSET );
43         else { sprintf(errmsg,"%s %d","msgget: no flg",
44                     msgflg);
45                 return(-1);
46             }
47     }
48
49     if (qid < 0)
50         return(-1);
51     else return(qid);
52 }
53
54

```

Code fragment 3.18

Lines 1 to 2 in code fragment 3.18 define two global semaphore identifiers, which are also used when removals are needed. In line 4, `msgget()` takes two parameters: `Key` is used to identify a message queue. It is not required to be unique, depending upon the value of `msgflg`. Several processes can call `msgget()` with the same `key` value. The `msgflg` determines the actual operation. `Msgget()` will return the value (-1) if an attempt is made to create an existing message queue with `msgflg=IPC_EXCL`. If `msgflg` is not set to `IPC_EXCL` when a `msgget()` is applied to an existing message queue, the existing message queue's identifier will be returned. In this function `key_t` is equivalent to `int` type in C.

Because no explicitly initialization takes place before `msgget()` is called, `msgget()` itself initialize the required semaphores in Lines 13 to 17. Through its arguments, `allocit()` creates and initiates semaphores `forasync` and `forsync`. `allocit()` is not listed. It simply calls `sem_create()` which is also used in the code fragment 3.3 with the second parameter set to 20.

In line 19, `searchkey()` is called to determine whether a message queue associated with the given `key` exists. The listing of `searchkey()` is omitted. There are three cases which depend on the value of `msgfal` and appropriate action is taken.

The first case in lines 21 to 31, a new message queue is created, as is a semaphore. The condition tested in line 21 says that the function call requires the creation of a message queue and the queue does not exist. In line 23, `setsid()` is called to set a semaphore for the newly created queue for synchronous message passing. And in line 25 a semaphore is set for asynchronous message passing. The

function `setsid()` is provided in which a queue is actually created and a semaphore is set. `Msgget` may be called simultaneously by several processes to create message queues (recall that dynamic spawning is supported by the system queue table.) The critical regions that contains code which modifies the system queue table has to be protected. While up to 256 message queues can be created, there will not be enough semaphores to associate with each message queue. (there are a total of 40 individual semaphores allocated in the system.) Key is modulated to be mapped for available semaphores. The second argument of `setsid` is key modulo `MAXSEM` (`=20`) into the semaphore set identified by the first argument of the call. The following code fragment lists `setsid()`:

```

1
2     setsid(sid, snum)
3     int     sid;
4     {
5         int     qid;
6
7         P(semqtab, 0);
8         qid = createq();
9         V(semqtab, 0);
10
11        if (qid<0)
12            return(-1);
13        fprintf(stdout,
14            "\nsetsid:  sid %d mapped into qid %d",
15                sid, qid);
16
17        P(semqtab, 0);
18        p_qtab->qtab[qid].semid = sid;
19        p_qtab->qtab[qid].semnum = snum;
20        V(semqtab, 0);
21
22        return(qid);
23    }
24
```

Code fragment 3.19

Lines 7 to 9 in code fragment 3.19 protect the system queue table index pointer and call `createq()`. Lines 17 to 20 make assignments to the `qid` entry of the system

queue table. Up to this point, a queue is functionally concurrent. Recall that `createq()` obtains an entry of the system queue table, initializes the entry with the queue front and, queue rear pointers and stamps the entry to `present`, indicating that a queue is available. `setsid()` initializes the second half in the table entry to the semaphore identifier. `P()` and `V()` operations in code fragment 3.19, target the integer semaphore in the semaphore identifier `semqtab`.

Returning to code fragment 3.18, lines 22 to 25 set different arguments when calling `setsid()` to reflect two forms of message transmission.

Lines 38 to 47 in code fragment 3.18 handle the case in which the message queues associated with the required key exist. `getsid()` searches the system queue table to try to find the message queue identifier.

The next function to be discussed is `msgsnd()`. This function inserts a message into a requested message queue depending on various operation modes. The listing is shown below:

```

1
2  msgsnd(msqid, msgp, msgsz, msgflg)
3  int      msqid;
4  struct    msgbuf  *msgp;
5  int      msgsz,   msgflg;
6  {
7      int      rtn;
8      int      retrycount = 0;
9      /*      msgsz is not supported      */
10
11      if ( msqid < 0 ) {
12          sprintf(errmsg,"%s %d", "msgsnd:  ILLEGAL qid ",
13                  msqid);
14          return(-1);
15      }
16  lock:      /*      blocked operation      */
17      P(p_qtab->qtab[msqid].semid, p_qtab->qtab[msqid].semnum);
18      if ( (rtn = enrear(msqid, msgp)) < 0 ) {
19          if ( rtn == -2 ) {
20              V(p_qtab->qtab[msqid].semid,
21                p_qtab->qtab[msqid].semnum);

```

```

22         return(rtrn);
23     }
24     if (msgflg==msgblock) {
25         V(p_qtab->qtab[msqid].semid,
26         pp_qtab->qtab[msqid].semnum);
27         if ( ++retrycount % 100 == 0 )
28             fprintf(stderr, "msgsnd:  retry for:  %s",
29                     errmsg);
30         goto lock;
31     } else {
32         sprintf(errmsg, "%s %d ",
33                 "msgsnd:  can't snd to ", msqid);
34         sprintf(errmsg, "%s msgflg=%d ",
35                 errmsg, msgflg);
36         V(p_qtab->qtab[msqid].semid,
37         p_qtab->qtab[msqid].semnum);
38         return(-1);
39     }
40 }
41 V(p_qtab->qtab[msqid].semid, p_qtab->qtab[msqid].semnum);
42 return(0);
43 }
44 }
45

```

Code fragment 3.20

Currently, `msgsz` is not interpreted; it is present for compatibility with the System-V like `msg` interface. Line 11 to Line 15 in code fragment 3.20, check `msqid`; it will return (-1) if an invalid `msqid` is received. When an attempt by `enrear()` is made, two possible failures may occur. (-2) indicates emptiness of the system free list. This case will be handled by a higher routine, so it causes a return with the same return value (-2) in Line 22. The other failure is (-1) with various possibilities. Retries are made if `msgflg` is set to `msgblock`. Because it may trigger an infinite loop, some message will be printed in `stderr` in Line 28. Other cases causes of a (-1) returned from `enrear()` will not be corrected; these include semaphore operation failures and nonexistent message queues. Upon completion, 0 is returned to indicate a success.

Up to this point, the complete picture of how this level of operations are working should be clear.

3.2.6 The DP primitives

With the services provided by the set of *msg* operations in the fourth layer of the system, the Distributed Process primitives can be constructed. In every Distributed Process system, there are two levels of interface: primitives and standard DP functions, (*stdDP*) which are usually used in application level programming. *StdDP* functions are designed to rely on the message transport mechanism supported by the lower level primitives. In addition, *stdDP* has more operational options, such as the ability to select incoming messages. *StdDP* functions are not part of my DP implementation because of its standard nature that it is written once and for all (by Arnow).

The entire code of this level is very long and involves using other utilities provided by the native operating system, such as the signal facility. Only parts of *dpinit()*, *primary()*, *secondary* and *dpsend()* are discussed here. *dpinit()* may be the most important function in the set of the primitives. It not only builds a multi-processing environment by spawning secondary processes, but merges all efforts made in lower levels into a distributed computational system as well. The entire code of this routine is listed because it gives the readers a picture showing how the lower level IPCs are created, initiated and used. However, some lines in the code fragment will not be further discussed, like the statements involving signal handling.

The complete `dpinit()` is listed below:

```

1      static int MSGkey[MAXPROC], AMMSGkey[MAXPROC];
2
3      #define    dpinit(ac, av)  tinipd()
4
5      tinipd()
6      {
7          int    i, spawn = 0, cid;
8          struct sigvec  vec, *p_vec;
9          struct sigvec  ovec, *p_ovec;
10
11          setbuf(stdout, CNULL);
12          setbuf(stderr, CNULL);
13
14          for ( i = 0; i < MAXPROC; i++) {
15              MSGkey[i] = 2047 + i;
16              AMMSGkey[i] = 255 + i;
17          }
18
19          ipcinit();
20          msginit();
21
22          for ( spawn = 1; spawn < MAXPROC; spawn++ )
23              if ( (cid = fork()) < 0 ) {
24                  fprintf(stdout, "\n %s %dth proc, err=%d\n",
25                      "dpinit: (FATAL) can't spawn the",
26                      spawn, errno);
27                  exit(1);
28              } else
29                  if ( cid == 0 ) {
30                      sleep(1);
31                      secondary(spawn);
32                      break;
33                  } else
34                      if ( spawn == MAXPROC-1 ) {
35                          dppid = 0;
36                          primary();
37                      } else
38                          continue;
39
40          p_vec = &vec;
41          p_ovec = &ovec;
42          p_vec->sv_handler = usr1catch;
43          p_vec->sv_mask = 0;
44          p_vec->sv_flags = 0;
45          (void) sigvec(SIGUSR1, p_vec, p_ovec);
46
47          return(MAXPROC);
48      }
49
50

```

Code fragment 3.21

To make the user interface consistent for all DP programs, `dpinit()` is written as a macro with two dummy parameters. In other DP implementations, such as that for TCP/IP DP, the parameters passed into `dpinit()` are interpreted for some purposes. In this implementation, these parameters are not used at all. To avoid a compiler warning, the actual `dpinit()` is mapped into `tinipd()`, a real function with no arguments.

Line 2 in code fragment 3.21 defines two static arrays that are used to hold keys for creating message queues. `MAXPROC` is a constant that may be defined in a header file or defined through the command line when this routine is compiled. Literally, `MAXPROC` stands for the maximum number of processes that can be included in the distributed computation. A reasonable number would be 4, 8, 16, 32, 64 or may be 128, as the maximum possible number of processes to be created.

In line 6 the function is started. It prepares two output files by removing output buffers so data written into the output files will be immediately printed.

Line 15 to Line 18 assign some keys to the arrays `MSGkey` and `AMSGkey`. They will be used when a primary calls `msgget()`. The key values are arbitrary. The values once assigned are suggested to be used all the time for debugging purposes. Remember that the `msgget()` is not a the real System V function. In our system `msgget()` can be employed to allocate up to couple of thousands message queues. On contrast, System V has a limit about 60.

`ipcinit()` is called in line 20, and that routine is listed in code fragment 3.4. All required shared memory is allocated at this point. Line 21 calls another routine

`msginit()` to create message queues. Part of `msginit()` is provided below:

```

1
2     static int msgqid[MAXPROC], amsgqid[MAXPROC];
3
4     msginit()
5     {
6         int    i;
7
8         for ( i = 0; i < MAXPROC; i++ )
9             if ((msgqid[i] =
10                 msgget(MSGkey[i],IPC_CREAT|FOR_SYNC))<0)
11                 fprintf(stdout, errmsg);
12             else
13                 fprintf(stdout, "OK msgid is %d\n",
14                         msgqid[i]);
15
16         ...
17
18     }
19
20
21
22
23 }
```

Code fragment 3.22

Line 10 in code fragment 3.22 creates message queues by calling `msgget()`. A similar loop exists in the omitted part in which a set of message queues dedicated to asynchronous messages are created. Now let's go back to code fragment 3.21.

From Line 23 to Line 38 in code fragment 3.21, a number of secondary processes are created. Lines 24 to 28 call `fork()` to spawn a child process, and exit if a creation fails. If the creation succeeds two independent processes will continue to execute Line 30 of code fragment 3.21. One copy of the program has `cid == 0` which executes Lines 31 to 33, and line 32 calls `secondary()`; While the other copy of the program having `cid != 0` runs lines 35 to 39. In this part, the loop will be continued by line 39 until the last iteration in which routine `primary()` is called. In any of these cases, once the loop terminated, the statements in line 41 to Line 48 are carried out. To make the idea clear, let's go back to look at the code one more time.

Line 21 and Line 22 are executed only once. When the program branches, the newly created program will start to run at the place where it is created. In this case, it is the `if` statement in line 30. The one which executes the `ipcinit()` and `msginit()`, (lines 21 to 22) is the process having a non-zero value in `cid` in line 34. Therefore, it is called the primary process. A program with `cid==0` calls `sleep()`, `secondary()` and exits the loop at line 41. There will be a total of `(MAXPROC -1)` of these programs or secondary processes. The primary process having `cid !=0` always generates the iteration until, at last, it calls `primary()`. When the loop terminates, it starts to run the statement in line 41.

When `tinipd()` returns, it returns the constant `MAXPROC`. The constant is returned because the required number of processes should be created. Any failure to do so, causes the primary process to abort in Line 28. In network DP, the actual number of processes created may not agree with the number of processes requested when `dpinit()` is called. Lines 9 to 10 and Lines 41 to 46 are used to define and manipulate signal facilities. The discussion is omitted.

When the primary process calls the routine `primary()` and secondary processes call the routine `secondary()`, some connection and exchange of information takes place. After spawning secondary processes, the `primary()` enters a stage waiting for the secondary processes response. The following is the function that `primary()` calls to receive the system process identification numbers of the secondary processes.

```

1
2     preadsyspid()
3     {
4         int     ite;
5         struct  msgbuf msg, *p_msg;
```

```

6
7      p_msg = &msg;
8      for (ite = 1; ite < MAXPROC; ite++) {
9          if ( msgrcv(dppid, p_msg,
10                 sizeof(msg), msgfifo, msgblock) < 0 ) {
11              fprintf(dplog, "%s \n",
12                     "(FATAL) preadsyspid:  can't msgrcv");
13              exit(RUNERROR);
14          }
15          if (dppidtab[msg.msnd] == 0)
16              dppidtab[msg.msnd] = msg.mrcv;
17      }
18  }
19
20      dppidtab[dppid] = getpid();
21      return(0);
22  }
23

```

Code fragment 3.23

During the initialization stage, the lower level message operations, `msgrcv()` and `msgsnd()`, are used. In Line 8 in code fragment 3.23, `preadsyspid()` loops `MAXPROC` number of times to receive messages sent by the secondaries. `msgrcv()` is invoked in line 9 with five arguments. The first argument of the `msgrcv()` call is always `dppid` which makes sure that only the process owning the message channel reads the messages. The second argument of the call is a pointer pointing to a message buffer defined in lines 5 to 7, which is used to receive an incoming message. The next argument specifies the number of characters in the message buffer. The fourth argument is the message type, its value is `msgfifo`. The last argument denotes the operation mode. In this stage of message transmission `msgblock` is used for all operations. In lines 16 to 20, `dppidtab` is initiated in the primary process. `dplog` will be discussed when the formal listing of `primary()` is given.

Now, the following short code fragment is the function called by `secondary()` to send the system process id.

```

1
2     swritesyspid(dppid)
3     int      dppid;
4     {
5         struct msgbuf msg, *p_msg;
6
7         p_msg = &msg;
8         p_msg->mrcv = getpid();
9         p_msg->msnd = dppid;
10
11         if ( msgsnd( 0, p_msg, sizeof(struct msgbuf),
12                 msgblock) < 0 ) {
13             fprintf(dplog, "\n %s",
14                 "(FATAL) swritesyspid:  can't dpsend pid";
15             exit(RUNERROR);
16         }
17         return(0);
18     }
19

```

Code fragment 3.24

When `swritesyspid()` is called by `secondary()`, it takes an argument `dppid`. The `secondary` receives its `dppid` by the argument `spawn` in Line 32 of code fragment 3.21. `swritesyspid()` initiates `msg` by its `dppid` and its system process id in Lines 8 to 9. Then a `msgsnd()` is attempted to message queue 0, which the process with `dppid==0` will read. The operation is blocked. Blocked operations in message send operations seems unsound. However, in this implementation, a blocked send operation requires retries when the addressed queue is not fully functional.

After the first handshake between primary and secondaries, the primary broadcasts its `dppidtab` to all secondary processes. Here is the function doing that:

```

1
2     pbroadpidtab()
3     {
4         int      i, j;
5         struct msgbuf msg, *p_msg;
6
7         p_msg = &msg;
8         for (i=1; i<MAXPROC; i++)
9             for (j=0; j<MAXPROC; j++) {
10                 p_msg->msnd = j;
11                 p_msg->mrcv = dppidtab[j];

```

```

12             if (msgsnd(i, p_msg, sizeof(msg),
13                     msgblock) < 0) {
14                 fprintf(dplog,
15                     "pbroadpit: can't msgsnd");
16                 exit(RUNERROR);
17             }
18         }
19     }

```

Code fragment 3.25

pbroadpidtab is simple, and it can explain itself. Each message carries only two integers, the index of the table and the content of the entry. It may be improved by a large portion of the table being sent by one message. But that requires some code to fill the messages.

On the secondary side, sreadpidtab() runs a loop to receive MAXPROC number of messages from the dp process 0.

```

1
2  sreadpidtab(dppid)
3  {
4      int      i;
5      struct  msgbuf  msg, *p_msg;
6
7      p_msg = &msg;
8      for ( i = 0; i < MAXPROC; i++ ) {
9          if (msgrcv(dppid, p_msg, sizeof(struct msgbuf),
10                  msgfifo, msgblock) < 0) {
11              fprintf(dplog, errmsg1);
12              exit(RUNERROR);
13          }
14
15          if (dppidtab[p_msg->msnd] == 0)
16              dppidtab[p_msg->msnd] = p_msg->mrcv;
17      }
18
19      dppidtab[dppid] = getpid();
20      return(0);
21  }
22

```

Code fragment 3.26

The following are the complete listings of primary() and secondary. They are self-explanatory.

```

1
2 FILE      *dplog, *openfile(); /* no listing */
3 void      dumpqid();           /* no listing */
4 void      dumpidtab();         /* no listing */
5 int       pgood();             /* no listing */
6 int       scheck();            /* no listing */
7 #define maxtryct 10 /* num. of tries to dialog with */
8                               /* secondaries */
9 primary()
10 {
11     int     tryct = 0;
12     int     i;
13     char     FILENAME[20];
14     /* create log file */
15     (void) sprintf(FILENAME,"dplog0");
16     dplog = openfile(FILENAME);
17
18     (void) fprintf(dplog,"Total queues=%d\n",MAXPROC*2);
19     (void) dumpqid(msgqid, amsgqid);
20
21     /* SETUP communications with other processes */
22     for (i=0; i<MAXPROC; i++)
23         dppidtab[i] = 0; /* dppidtab for ASYNC signals */
24 startover:
25
26     if (++tryct == maxtryct) {
27         fprintf(dplog,
28             "primary: MaxTryCount reached\n");
29         exit(RUNERROR);
30     }
31
32     (void) preadsyspid();        /* see the code fragment 3.23 */
33     (void) dumppidtab();        /* for debug purpose */
34     (void) pbroadpidtab();      /* see the code fragment 3.25 */
35
36     /* pgood() checks feedback from all secondaries */
37     /* It returns 0 if some secondary pidtab not setup */
38     if (!pgood())
39         goto startover;
40
41     fprintf(dplog,"SETUP DONE BY PRIMARY\n");
42     return(0);
43 }
44

```

Code fragment 3.27

```

1
2 secondary(pid)
3 int     pid;
4 {
5     int     tryct = 0;
6     char     FILENAME[20];
7
8     dppid = pid;
9     sprintf(FILENAME, "dplog%d",dppid);
10    dplog = openfile(FILENAME);

```



```

11
12      /* SET UP communications with other processes */
13 startover:
14     if (++tryct == maxtryct) {
15         fprintf(dplog,"secondary: MaxTryCount reached\n");
16         exit(RUNERROR);
17     }
18
19     /* sends sys pid to primary */
20     (void) swritesyspid(dppid);
21
22     /* Receive 1st message from primary for dppidtab */
23     (void) sreadpidtab(dppid);
24
25     /* dppidtab is checked, if a problem is detected, */
26     /* scheck sends a repeat-request to primary.          */
27     if ( scheck() < 0 )
28         goto startover;
29
30     fprintf(dplog,"\tFinal Table Of Pids \n");
31     (void) dumppidtab();
32     fprintf(dplog,"SETUP DONE BY  SECONDARY\n");

```

Code fragment 3.28

4

Distributed Pseudorandom Number Generators

A system of pseudorandom number generators (prn) to be used in distributed scientific computations is highly desirable. Such a system is necessary to conduct distributed Monte Carlo calculations. This chapter discusses an implementation of a distributed pseudorandom number generator system on a computer network. The work is based on “Random Number Generators for MIMD Parallel Processors”. [1]

4.1 Distributed Monte Carlo calculation

A distributed Monte Carlo calculation is carried out by many worker processes, usually on different hosts on a network. The work that each worker process carries out must be statistically independent of that of any other worker process. Such Monte Carlo simulation requires different random sequences for all the worker processes. In order to increase the efficiency of computation, no random sequences shall be used in different worker processes in the same way. Moreover, in some applications such as nuclear physics, MC methods are used to study particles which are subject to random collisions. In these applications, worker processes that simulate particles may be required to terminate dynamically, and may or may not be required to create new worker processes as needed to continue the simulation. A system of generators should be available to meet the needs of branching, random processes. Many methods can be used to implement random number generators in parallel computations. Each of these methods chooses a suitable strategy to satisfy

the requirement of independent random sequences for worker processes. Nevertheless, there are other requirements for pseudorandom number generations, such as long period of pseudorandom sequences, good statistic properties, etc. They will be discussed specifically with the method used in this work.

The approach in this work to generate independent random sequences for distributed Monte Carlo calculations is to use a family of linear congruential pseudorandom number generators that are organized into a binary tree which is used for all distributed processes. Basic concerns of this structure are supporting process creation during a computation and guaranteeing that no generator is used by more than one worker process. One desirable feature of a distributed pseudorandom number generator is reproducibility. This feature will help to debug logic errors by allowing a run to be reproduced. A binary tree structure that maintains a system of prns was suggested by Fredrickson[24]; later it was modified and proposed by Percus & Kalos[1].

The tree structure has good characteristics which can be used to keep track of distributed processes and generators used. Consider a distributed computation that the first process spawns n child processes for some n . Its first child spawns x children, its second child spawns y child processes and so forth. The process creation is going on to many levels. And the number of levels of the process creation may not be able to be predicated or defined before the computation starts.

In this sense, it is difficult to define structures with fixed sizes, like n -dimensional arrays, to capture the process creation information. However, unlike

n -dimensional arrays, the tree structures can be viewed as if they have dimensions and the dimensions can become virtually infinite. Take look a binary tree, for instance, the first process takes a dimension along its right subtree. The length of the dimension can be arbitrarily long. Each its child is located at one of its left subnode along the dimension. With the same idea, each of the children takes its right subtree as its dimension. So the structure of a series of distributed process creation is homomorphic to a binary tree. And the tree can be implemented.

This scheme can also be applied on unrelated distributed processes. If the processes are not rooted at a common process, as long as an unrelated process shares work load with some process that is derived from the root process, the relationship of these processes can be mapped into the binary tree.

In the following sections, linear congruential random number generators and random number generators for distributed processes are discussed.

4.2 Linear congruential random number generators

The multiplicative congruential random number generator method was first introduced by Lehmer [25]. The algorithm with parameters a, m is:

$$X_{n+1} = aX_n \pmod{m} \quad (4.1)$$

where m is modulus, $a(< m)$ is multiplier and both are positive integers. Given an initial value, $X_0 \neq 0$, Eq(4.1) defines a sequence of integers in $[1, m - 1]$. If m is a prime and a is a primitive element modulo m , then Eq(4.1) has maximal period $(m - 1)$. A real sequence $u_{i,i=0,1,2,\dots}$ uniformly distributed on $(0, 1)$ may be obtained

via dividing $X_{i,i=0,1,2,\dots}$ by modulus m .

A sequence of numbers generated arithmetically by Eq(4.1) is called a pseudo-random number sequence to distinguish it from a sequence of numbers chosen truly randomly. If the parameters a and m in Eq(4.1) are chosen properly, the finite set of $(m - 1)$ numbers in $[1, m - 1]$ can be apparently random, meaning that it satisfies some test for randomness[26].

Another family of generators is called the linear congruential random number generators and has the form:

$$X_{n+1} = aX_n + c \pmod{2^\beta} \quad (4.2)$$

where $c \neq 0$, $\pmod{2^\beta}$ is an additive constant. This generator can achieve the maximal period of m iff $a \equiv 1 \pmod{4}$ and c is odd. The modulus in Eq(4.2) can be some other number, such as 10^β . The reason that Eq(4.2) is presented here is that it is the generator we are interested in most, and it is used in most of the following discussion.

In both families of generators, selecting a multiplier is a critical issue. Besides the concern of the period of the sequence, a will affect the randomness of the sequence it produces, and part of the effort in statistical tests, the Spectral test, for instance, is to judge the performance of an a .

One major weakness of linear congruential random number generators was found when the joint distribution was investigated [26]. Let a sequence of k numbers $(X_i, X_{i+1}, \dots, X_{i+k-1})$ be drawn consecutively from Eq(4.1) or Eq(4.2) and R be a set of such sequences. There exist families of $(k - 1)$ -dimensional, parallel, equally-

spaced hyperplanes, and each of these individual families covers R .

Specifically, sequences of three numbers can be studied geometrically in a three-dimensional unit cube. For $(t > 3)$ -dimensional cases, sequences of t numbers can be studied mathematically. A method of studying this distribution will be discussed in the next section.

To see how two-dimensional parallel hyperplanes cover a set of 3-tuples (X_i, X_{i+1}, X_{i+2}) , the following generator was studied [26]:

$$X_{n+1} = 137X_n + 187 \pmod{2^8}$$

A set of 256 3-tuples, (X_i, X_{i+1}, X_{i+2}) , were plotted in a three-dimensional cube in Fig(4.1 (a)). The above generator has period length 256. It was considered a good generator in the sense that it exhibited good performance on several tests†. But from direct observations, the set of 3-tuples are not randomly distributed in the unit cube. They are all correlated in a number of parallel planes. Even though congruential random number generators with short period length, like the one showed above, fail to distribute the sequences randomly in three-dimensional space, other random number generators with much longer periods, will still be very useful for studying the joint distribution.

Another congruential random number generator*studied was:

$$X_{n+1} = 11^{13}X_n \pmod{2^{48}}$$

† e.g. the Spectral test, see [26]

* The well known rannyu() from Kalos, the code is obtained from D. Arnow

Again, 256 3-tuples points were drawn from the above generator and plotted in Fig. (4.1 (b)). Because the above generator has a very long period, (2^{48}), it can be seen that the set of points appear to have less correlation in the unit cube. The fact of the matter is that the set of points are, in the same way as above, covered by number of families of parallel planes. The distances between parallel planes are so small that a small number of points, relative to the period, do not form easily recognizable parallel planes.

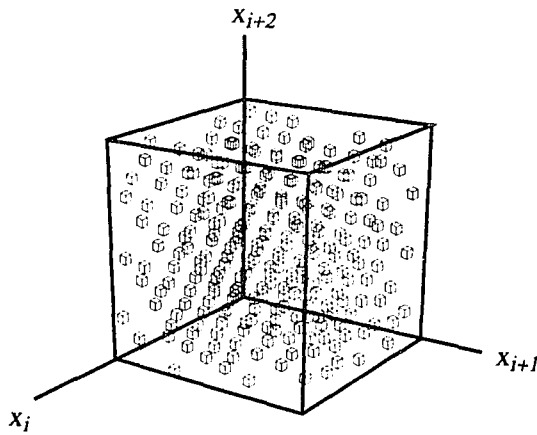


Figure 4.1 (a)

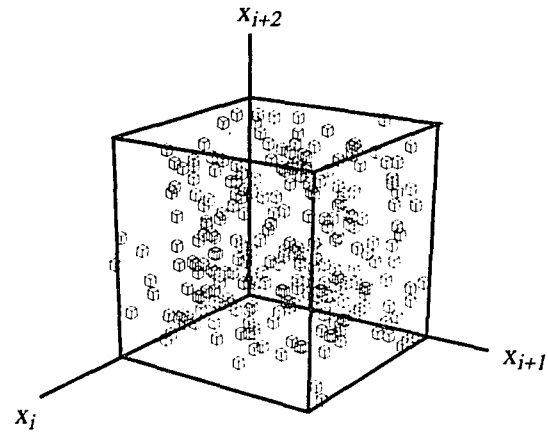


Figure 4.1 (b)

4.3 The Spectral tests

A pseudorandom number generator is usually subjected to two types of tests, the period test and the randomness test. A test that determines the periods of linear random number generators in the form of Eq(4.1) or Eq(4.2) is trivial. But to determine good or poor randomness of these generators is a non-trivial matter. To test the randomness of a generator, usually longer and longer sequences are

observed and non-random or flat sequences may be identified. Generators having a short period may hardly be random, so longer periods are desired. Generally, randomness tests can be categorized as statistical tests, empirical tests or theoretical tests.

The Spectral test for multiplicative or linear congruential generators has characteristics of both theoretical and empirical tests. Like theoretical tests, it examines sequences of full period. Like empirical tests, it requires computational work. The Spectral test is the *most* powerful test so far that has been used to test congruential random number generators.

Descriptions of the spectral test are given in [1] and [26]. Knuth's algorithm [26] is used to test the randomness of the joint distribution in dimensions less than 10. The Percus and Kalos' algorithm [1] is used to select good constants for parallel linear congruential random number generators, each of which has the form as in Eq(4.2).

4.3.1 Knuth's algorithm

A set of t -tuples drawn from a linear congruential random number generator is covered by families of $(t - 1)$ -dimensional equally-spaced hyperplanes. Within a given family, the distance between planes is constant. This has been illustrated for a 3-tuple case in the previous section. The Spectral test searches for the family with the maximum interplane distance $\frac{1}{\nu_t}$. ν_t is called the accuracy of the t -dimensional random number generator, which is the exact value that the Spectral test algorithm determines.

In each equally-spaced $(t - 1)$ -dimensional, parallel, hyperplane family, there is one hyperplane passing through the origin $(0, 0, \dots, 0)$. There are only two adjacent planes that have to be found to determine the distance $\frac{1}{\nu_t}$. The plane passing through the origin can be chosen as one of two adjacent parallel planes. The other plane can be determined by the one from the family having the shortest distance to the origin.

To find the $(t - 1)$ -dimensional parallel hyperplane family having maximum distance between two adjacent planes, we can study the vector product $V \cdot U = q$, where V is a vector representing a t -tuple, a point in a hyperplane, and U is a t -tuple non-zero vector which is perpendicular to the parallel planes of the family. q is a constant having different values for different planes.

Vector V can be formed as follows. Consider the set of sequence of t -tuples generated by Eq(4.2):

$$\begin{aligned} & \left\{ \frac{1}{m} (x_0, x_1, \dots, x_{t-1}) \mid 0 \leq x_i < m \right\} \\ &= \left\{ \frac{1}{m} (x_0, ax_0 + c, a^2x_0 + ac + c, \dots, a^{t-1}x_0 + a^{t-2}c + \dots + c) \right\} \end{aligned} \quad (4.3)$$

Eq(4.3), then, can be written as a sum of two vectors V_0 and L^0 :

$$V_0 = \{0, c, (1 + a)c, \dots, (1 + a + a^2 + \dots + a^{t-2})c\}$$

and

$$L^0 = \left\{ \frac{1}{m} (x_0, ax_0, \dots, a^{t-1}x_0) \right\}$$

V_0 is a constant vector; the effect of it is to merely shift hyperplanes in t -dimensional space without changing the relative distance between parallel planes. Shifting does not affect the Spectral test, so V_0 is ignored. Notice that the constant c will not be a part of the following discussion. We rewrite L^0 so that it represents infinite many copies of the original set Eq(4.3):

$$L = \left\{ \frac{1}{m} (x_0 + k_0, ax_0 + k_1, \dots, a^{t-1}x_0 + k_{t-1}) \mid x_0, k_0, k_1, \dots, k_{t-1} \text{ are integers} \right\} \quad (4.4)$$

The set L extends L^0 in $(t-1)$ -D space. It employs $(t+1)$ arbitrary integers $x_0, k_0, \dots, k_{t-1}$. But by appropriate transformations, only t arbitrary integers $y_1, \dots, y_t$ are needed. Let $y_1 - k_0m$ substitute for x_0 . Then the first term in L would be:

$$\frac{y_1 - k_0m}{m} + k_0 = \frac{y_1}{m} - k_0 + k_0 = \frac{y_1}{m}$$

And the second term in L would be:

$$\frac{a(y_1 - k_0m)}{m} + k_1 = \frac{ay_1}{m} - ak_0 + k_1 = \frac{ay_1}{m} + y_2$$

All subsequent terms in L can be obtained in the same way. Finally, L is written as:

$$\begin{aligned} L &= \left\{ \frac{y_1}{m}, \frac{ay_1}{m} + y_2, \frac{a^2y_1}{m} + y_3, \dots, \frac{a^{t-1}y_1}{m} + y_t \mid y_1, y_2, \dots, y_t \text{ are integers} \right\} \\ &= \left\{ y_1 V_1 + y_2 V_2 + \dots + y_t V_t \right\} \end{aligned} \quad (4.5)$$

where the V_i are:

$$\begin{aligned}
 V_1 &= \frac{1}{m}(1, a, a^2, \dots, a^{t-1}) \\
 V_2 &= (0, 1, \dots, 0) \\
 V_3 &= (0, 0, 1, \dots, 0) \\
 &\dots \\
 V_t &= (0, 0, \dots, 1)
 \end{aligned} \tag{4.6}$$

All the above effort is made to find vectors $V_i, 1 \leq i \leq t$ such that each has t coordinates. The vector $U = (u_1, \dots, u_t)$ is used to define a family of parallel $(t - 1)$ -dimensional hyperplanes that covers all the points of L . A set of points, $(x_1, \dots, x_t) \in L$, that is on a particular hyperplane is given as:

$$\left\{ (x_1, \dots, x_t) \mid x_1 u_1 + \dots + x_t u_t = q \right\} \tag{4.7}$$

where q is a constant which determines a $(t - 1)$ -dimensional hyperplane. The distance between the plane passing through the origin and any other parallel plane within the family is then:

$$\frac{1}{\sqrt{u_1^2 + \dots + u_t^2}} = \frac{1}{\nu_t}$$

The Spectral test tries to find the closest plane to the origin, so it minimizes:

$$\begin{aligned}
 &\min \sqrt{u_1^2 + \dots + u_t^2}, \\
 &\text{subject to: } (y_1 V_1 + \dots + y_t V_t) \cdot (u_1^2 + \dots + u_t^2) \text{ is an integer}
 \end{aligned} \tag{4.8}$$

In the other words, ν_t , what the test seeks, is the length of the shortest vector $U = (u_1, \dots, u_t)$. The above minimization can be further simplified. Since the points $V_1, \dots, V_t$ are all in L , u_i must be integers. Furthermore, from $V_1 = \frac{1}{m}(1, a, a^2, \dots, a^{t-1})$, u_i must satisfy

$$u_1 + au_2 + \dots + a^{t-1}u_t \equiv 0 \pmod{m}$$

Eq(4.8) can finally be written as::

$$\nu_t^2 = \min_{(u_1, \dots, u_t) \neq (0, \dots, 0)} \left\{ u_1^2 + \dots + u_t^2 \mid u_1 + au_2 + \dots + a^{t-1}u_t \equiv 0 \pmod{m} \right\} \quad (4.9)$$

The details of the procedure for searching for vector U is given in [26] as the S algorithm. In section 4.4 an implementation of the S algorithm will be discussed.

4.3.2 Percus and Kalos' algorithm

Percus and Kalos' MIMD (multiple instruction stream/multiple data stream) parallel random number generator (prn) system consists of two parts: the MIMD Spectral test algorithm which is used to determine the simultaneous generators and a tree structure in which a system of generators is maintained for parallel computing. The random number generator system is suitable for computer systems which are similar to the MIMD model; for example, networked computers having no commonly shared memory.

The simultaneous or parallel random number generators considered are a class of linear congruential generators

$$x_{n+1,i} \equiv ax_{n,i} + b_i \pmod{m}$$

where the subscript i denotes different individual random number generators. Each generator is thus a generator having the form of Eq(4.2). The MIMD Spectral test proposed by Percus & Kalos guarantees that good constants b_i can be found, and that the generators produce different pseudorandom sequences with no subsequences in common.

A tree structure was first proposed by Fredrickson[†], et al. It specifies a mechanism to maintain pseudorandom generations. Then in “Random number generators for MIMD parallel processors” [1], a tree structure using linear congruential pseudorandom number generators is proposed to support parallel computations.

The MIMD tree is constructed as a binary tree with each node of the tree representing a random number generator. Left and right branches of a node are given as:

$$x_{L,n+1} \equiv ax_{L,n} + b_L \pmod{m}$$

$$x_{R,n+1} \equiv ax_{R,n} + b_R \pmod{m}$$

In a computation, random number generators that are defined in the binary tree can be associated with the distributed processes. The right branch of a node is always used by a parent process. The left branch of a node is allocated to the left child process of the node. All the congruential generators use the same multiplier a .

[†] Fredrickson, P., Hiromoto, R., Jordon, T. L., Smith, B., and Warnock, T. Pseudorandom trees in Monte Carlo, *Parallel Computing*, Vol. 1. 1984, p. 175.

Generators are differentiated by additive constant b 's. Different b 's guarantee that pseudorandom numbers produced by the generators are in a different order on the set of integers in $[0, m - 1]$, and that no possible overlapping occurs. The MIMD Spectral test searches for good b 's so that a high degree of randomness for parallel generators is ensured.

The MIMD Spectral test assumes the existence of a good multiplier, a . a is not affected in the algorithm, but it should, together with m , pass the sequential spectral test given in the previous section. The MIMD congruential generator consisting of t -parallel generators is:

$$x_{n+1,i} \equiv ax_{n,i} + b_i \pmod{m} \quad (4.10)$$

where $i = 1, 2, \dots, t$. The b_i are odd, and $m = 2^\beta$. It is assumed that Eq(4.10) has the maximum period length m . Then Eq(4.10) can be written as:

$$\begin{aligned} x_{n+1,i} &\equiv ax_{n,i} + b_i \pmod{m} \\ &\equiv a^n x_{0,i} + a^{n-1}b_i + a^{n-2}b_i + \dots + ab_i + b_i \pmod{m} \\ &\equiv x_{0,i} + (a^{n-1} + a^{n-2} + \dots + a + 1)b_i + a^n x_{0,i} - x_{0,i} \pmod{m} \quad (4.11) \\ &\equiv x_{0,i} + b_i \frac{a^n - 1}{a - 1} + (a^n - 1)x_{0,i} \pmod{m} \\ &\equiv x_{0,i} + \gamma_i \frac{a^n - 1}{a - 1} \pmod{m} \end{aligned}$$

where $\gamma_i \equiv b_i + (a - 1)x_{0,i}, i = 1, \dots, t$. Here γ_i are relatively primes to m (or prime numbers less than m). $\gamma_i^{-1} \pmod{m}$ can be uniquely determined. So, the t dimensional pseudorandom sequences can be expressed as:

$$x_{n,j} \equiv \gamma_1^{-1} \gamma_j x_{n,1} + (x_{0,j} - \gamma_1^{-1} \gamma_j x_{0,1}) \pmod{m} \quad (4.12)$$

For any integers $s_1, \dots, s_t$, then

$$\begin{aligned} \sum_{j=1}^t s_j x_{n,j} &\equiv \gamma_1^{-1} x_{n,1} \sum_{j=1}^t s_j \gamma_j + \sum_{j=1}^t s_j (x_{0,j} - \gamma_1^{-1} \gamma_j x_{0,1}) \pmod{m} \\ &\equiv \gamma_1^{-1} x_{n,1} \sum_{j=1}^t s_j \gamma_j + C \pmod{m} \end{aligned} \quad (4.13)$$

where $C \equiv \sum_{j=1}^t s_j (x_{0,j} - \gamma_1^{-1} \gamma_j x_{0,1}) \pmod{m}$ is a constant with respect to n .

Comparing a truly random sequence, $\mathbf{x} = (x_1, \dots, x_t)$, in a t -dimensional vector space over the field of integers ($\text{mod } m$) with an ideal pseudorandom sequence $\mathbf{x}_n = (x_{n,1}, \dots, x_{n,t})$ in Eq(4.11), the former sequences are uniformly distributed in a probability measure:

$$g(\mathbf{x}) = \frac{1}{m_t}$$

while the latter are distributed as:

$$\bar{g}(\mathbf{x}_n) = \begin{cases} \frac{1}{m} & \text{if } x \in \{(x_{n,1}, \dots, x_{n,t}), n = 1, \dots, m\}, \\ 0 & \text{otherwise.} \end{cases}$$

By expressing the above as an exponential, the t -dimensional vector distribution for the truly random sequences can be written as:

$$f(s_1, \dots, s_t) = \frac{1}{m_t} \sum \exp\left(-\frac{2\pi i}{m} \sum_{j=1}^t s_j x_j\right) \quad (4.14)$$

And for pseudorandom sequences,

$$\bar{f}(s_1, \dots, s_t) = \frac{1}{m} \sum_{n=1}^m \exp\left(-\frac{2\pi i}{m} \sum_{j=1}^t s_j x_{n,j}\right) \quad (4.15)$$

where $\bar{f}(s_1, \dots, s_t)$ can be considered as the frequency of the vector $\mathbf{x} = (x_{n,1}, \dots, x_{n,t})$ and the generator in Eq(4.10) is the source of the RHS of Eq(4.15).

By applying Eq(4.13) and the fact that the set $\{x_{n,1}\}_{n=1}^m$ is equal to the set of integers $\{0, \dots, m-1\}$, Eq(4.15) becomes:

$$\bar{f}(s_1, \dots, s_t) = \exp\left(-\frac{2\pi i C}{m}\right) \sum_{x=0}^{m-1} \exp\left(-\frac{2\pi i}{m} \gamma_1^{-1} x \sum_{j=1}^t s_j x_{n,j}\right) \quad (4.16)$$

The relation between Eq(4.16) and a derived test that minimizes $(s_1^2 + \dots + s_t^2)$ is

$$|\bar{f}(s_1, \dots, s_t)| = \begin{cases} 1 & \text{if } \gamma_1^{-1} \sum_{j=1}^t s_j \gamma_j \equiv 0 \pmod{m}, \\ 0 & \text{otherwise.} \end{cases}$$

or when $\gamma_1^{-1} \sum_{j=1}^t s_j \gamma_j \equiv 0 \pmod{m}$, $|\bar{f}(s_1, \dots, s_t)|$ has the maximum value.

Then the Percus and Kalos' Spectral test is:

$$\nu_t = \min_{\substack{(s_1, \dots, s_t) \\ \exists (s_1, \dots, s_t) \neq (0, \dots, 0)}} \left\{ \sqrt{s_1^2 + \dots + s_t^2} \left| s_1 + \gamma_1^{-1} \gamma_2 s_2 + \dots + \gamma_1^{-1} \gamma_t s_t \equiv 0 \pmod{m} \right| \right\} \quad (4.17)$$

The implementation of Percus and Kalos' algorithm is discussed in the next section.

4.4 Implementing Spectral Tests

The proposed pseudorandom number generator to be used in distributed MC calculations is one with a long period. Specifically, in this work, a system of pseudorandom number generators with period of 2^{48} is chosen and implemented. To

write the Spectral test program that is used to deal with integer operations in a modular system, $Z_{2^{48}}^\dagger$, is difficult because 48-bit arithmetic operations can not be carried out by the 32-bit machine that is available to us. The alternative solution of using floating point (or double floating point) operations to simulate the desired modular system causes truncations which are not appropriate in the Spectral tests because nearly all arithmetic operations in the tests are integral operations and the computations are very precision sensitive.

In this and section 4.7, a solution to the above problem for simulating true integer operations is discussed with a newly invented division algorithm, called Cascade Division. Discussion of implementing the Spectral tests is the main topic of this section, some actual program codes are presented. Results generated by the Percus & Kalos' test (MIMD test) for the proposed random number generator system is presented at the end of this section.

4.4.1 New abstract data types

Precisely, in the Spectral tests, some multiplications of large integers can not be carried out by the normal integer operator $*$ which is provided by high level programming languages. For a large multiplier, a , with a value that is very close to $2^{32} - 1$, the result of $a * a$ is modulated into the range $[0, 2^{32} - 1]$, which makes the result incorrect. Nevertheless, some tricks can be made if the Knuth' Spectral test program with smaller multiplier, a , is the only program required to be implemented; because in that test *only* integral additions and multiplications are actually needed.

$\dagger$ Z_m is an integer system consists of integers from 0 to $(m-1)$

These techniques may not be able to be applied to test larger multipliers. Here, the integers being discussed are 48-bits long, and may not even be able to be loaded into a 32-bit register. So it makes all available tricks difficult to implement. Besides the above difficulty, integer divisions are also needed to prepare the MIMD Spectral test, for example, to compute multiplicative inverse elements modulo 2^{48} .

It was recognized, at the first stage of studying of this project, that a table provided in [2] should be reproduced for testing the correctness of the implementation of Knuth's algorithm. The table contains twenty nine pseudorandom number generators in the form of Eq(4.2) with the results of the Spectral tests. Most of the generators have large modulus, like 2^{35} , 2^{48} and even 2^{64} .

To overcome the obstacles, a general solution to carry out integer arithmetic is needed. For instance, the inverse elements of large prime numbers have to be computed in order to run the MIMD Spectral test. Some other pseudorandom number generator projects also need a general solution to handle very long integer operations. An implementation of the Spectral test that can be used to test a generator of any period length may be useful.

Therefore, implementing two abstract data types was the front end of this project. One of the expected abstract data type should be able to carry out very long integral operations; and the other was used to carry out real arithmetic operations with any expected precision.

Two implemented abstract data types are called `Real` and `Fixed` for the reasons to avoid using C built-in data type names `float` and `int`. The associated primitive

operations with each of these data types are addition, subtraction, multiplication, division, modulo and comparison operators. They are required for the essential numerical operations. In addition, more operations, such as *floor*, *abs*, *sqrt*, *log*, and *pow* functions are implemented in each of the data types. (The data types were written in C++.) Figure 4.2 shows the hierarchical structure of the implemented data types. The outer shell provides all the operators and function interfacing to users' programs.

4.4.2 A short C++ program

The Fixed and Real data types can be used in exactly the same way as C built-in data types `int` and `double` are used. The following is an example written using Real. The program computes the integral $\int_1^5 x^2 + 1$ by using the Monte Carlo method. The example was discussed in chapter 2 and the expected result is 45.33.

The example has three functions: `main()`, `updf()` and `pdf()`. `updf()` is the random number generator; its algorithm is given as:

$$x_{n+1} = 11^{13} * x_n + 11863279 \pmod{2^{48}} \quad (4.18)$$

where 11863279 is the largest prime that is smaller than $\sqrt{2^{48}} = 11863283$. `pdf()` is used to shift the pseudorandom sequence generated by `updf()` to a given interval (a, b) from $(0, 1)$. `main()` implements the Monte Carlo integration. Code fragment 4.1 lists the program.

Arithmetic Operation Heirarchical Structure

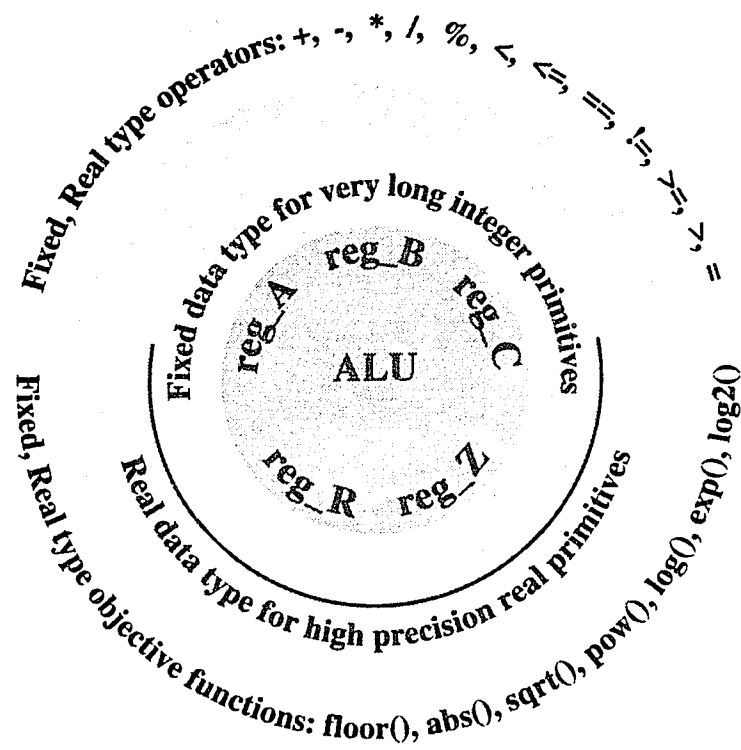


Figure 4.2 Fixed, Real type operators and functions.

```

1      #include "Kclass.h"
2
3      ostream fp = stdout;
4
5      Real
6      updf() {
7          static Real seed = 0.0;
8          static Real a = Rpow( 11, 13 );
9          static Real m = Rpow( 2, 48 );
10         seed = ( ( seed * a ) + (double) 11863279 ) % m;
11         return ( seed / m );
12     }
13
14     Real
15     pdf(double a, double b) {
16         return ( ( b - a ) * updf() + a );
17     }
18
19     // Compute the integral:  $f(x) = x^2 + 1$ , by the Monte Carlo
20     // method in the interval [1,5]
21
22     main() {
23         const int icount = 100000, inc = 20000;
24         const double lb = 1.0, ub = 5.0;
25         Real sum = 0.0;
26
27         for ( int i = 1; i <= icount; i++ ) {
28             sum = sum + Rpow( pdf( lb, ub ), 2 ) + 1.0;
29             if ( i % inc == 0 )
30                 fp << "\n\ttI= " << i << " RES= "
31                     << (ub-lb)*sum/(double)i;
32         }
33     }
34 }
35

```

Code fragment 4.1

Line 2 in code fragment 4.1 includes `Kclass.h`, which contains the definition of `Real` and `Fixed`. An object class `ostream` in Line 4 is also defined in `Kclass.h`. `ostream` is used to simulate the `C++` provided `ostream` which is used to route the output data to streamed i/o. The new feature that is added to `ostream` is that `Fixed` and `Real` objects are also realized in the streamed i/o. Line 31 and Line 32 show the `C++` notation to route streamed i/o.

Line 6 to Line 13 implement the pseudorandom number generator Eq(4.18). Notice that fractional numbers are manipulated on the Real domain. `Rpow()` used in Lines 9 and 10 take two `int` arguments a, b to evaluate a^b . Line 15 to Line 18 define the `pdf()` function mentioned above.

Line 24 in `main()` defines two constants `icount` and `inc`. `icount` will be used to control the loop constructed in Line 28 to Line 32 as the upper bound. `inc` is used to manipulate the loop variable counter so that only a few lines of outputs will be printed. `lb` and `ub` in Line 25 are the two ends of the interval in which the integration is evaluated. Line 29 computes the integration by using random values provided by `pdf()`. Here `Rpow()` is the same function used in `updf()`. However, it is called with the arguments of type `Real`, `pdf()` and `int 2`, meaning that it is to obtain the 2nd power of `pdf()`. Line 31 prints output items to `fp`. Notice that the last item is an expression in `Real` data type.

The output from the program is also provided below:

```

1
2      I= 20000 RES= +0045.30574225269204539807
3      I= 40000 RES= +0045.24385289925421446081
4      I= 60000 RES= +0045.24246132220781569111
5      I= 80000 RES= +0045.26051328998806620837
6      I= 100000 RES= +0045.31476886004499294368
7

```

The arithmetic operations are computed under a precision of 20 decimal fraction digits. The precision is a tunable constant in `Real`. (By default it is set to 20.) The output format is different to distinguish it from *floating point* figures of *IEEE* or *IBM* standard etc. Note that the `Reals` are not floating point numbers, they are real decimal figures with certain digits of accuracy in the fractional part.

Even the generators in Eq.(4.18) can be implemented by using `Real`; however, in this project Eq(4.18) is provided by double floating point computations. `Real` and `Fixed` are primarily used in this work to implement the two spectral tests.

4.4.3 Implementing the Knuth' Spectral test

The first half of this section discusses the mathematical part of the implementation and the second half of it presents the function called `KnuthSpectral()`.

In section 4.3.1, the Spectral test is described to minimize:

$$\begin{aligned}
 \nu_t^2 &= \min_{(u_1, \dots, u_t) \neq (0, \dots, 0)} \left\{ u_1^2 + \dots + u_t^2 \mid u_1 + au_2 + \dots + a^{t-1}u_t \equiv 0 \pmod{m} \right\} \\
 &= \min_{(x_1, \dots, x_t) \neq (0, \dots, 0)} \left((mx_1 - ax_2 - \dots - a^{t-1}x_t)^2 + x_2^2 + x_3^2 + \dots + x_t^2 \right).
 \end{aligned} \tag{4.19}$$

Before putting it into a computation, two steps of refinement to Eq(4.19) have to be done so that the actual computational work will be reduced. First, rather than searching each coordinate of x in Eq(4.19) over all integers, some upper bounds are computed to limit the search. Second, although setting some upper bound may lower the computational work substantially, there is still, much computational work to be done; therefore, transformations are introduced in the algorithm so that the computational work is minimized.

The problem of finding the minimum value ν_t^2 from Eq(4.19) is analogous to find the minimum value of a more general problem in the following *positive definite quadratic form*:

$$\begin{aligned} f(x_1, \dots, x_t) &= (u_{11}x_1 + \dots + u_{t1}x_t)^2 + \dots + (u_{1t}x_1 + \dots + u_{tt}x_t)^2 \\ &= (x_1U_1 + \dots + x_tU_t) \cdot (x_1U_1 + \dots + x_tU_t), \end{aligned} \quad (4.20)$$

where U is a $t \times t$ matrix and $U_1, \dots, U_t$ are vectors having form the $(u_1, \dots, u_t)$ which are the rows of U . $U_k = (u_1, u_t, \dots, u_t)$ must be an integer vector for $k = 1, \dots, t$ and satisfy

$$u_1 + au_2 + \dots + a^{t-1}u_t \equiv 0 \pmod{m} \quad (4.21)$$

To achieve these requirements, U_k , $1 \leq k \leq t$ are initialized as:

$$U_1 = (m, 0, 0, \dots, 0).$$

$$U_2 = (-a, 1, 0, \dots, 0)$$

$$U_3 = (-a^2, 0, 1, \dots, 0)$$

...

$$U_t = (-a^{t-1}, 0, 0, \dots, 1)$$

Clearly, each vector U_k , $1 \leq k \leq t$ satisfies Eq.(4.21). The vectors U_i , $i = 1, \dots, t$ can be changed to reduce the computational work. The transformation will be discussed shortly. During the transformation, U remains nonsingular, and therefore, there exist vectors V_j , $j = 1, \dots, t$ such that:

$$U_i \cdot V_j = \delta_{i,j} \quad i \leq i, j \leq t, \quad (4.22)$$

for some $\delta_{i,j}$. Initially, V_j $1 \leq j \leq t$ are the vectors given by Eq(4.6).

To search $(x_1, \dots, x_t) \in L$ such that

$$\min_{(x_1, \dots, x_t)} \left\{ x_1^2 + \dots + x_t^2 \mid x_1 u_1 + \dots + x_t u_t = 1 \right\} \quad (4.23)$$

is out of the question since L contains a huge number of members. The Spectral test computes some upper bound so that the search can go up through the bound.

The search of each coordinate $x_k = (x_1 U_1 + \dots + x_t U_t) \cdot V_k$ can be bounded as:

$$x_k^2 = (x_1 U_1 + \dots + x_t U_t)^2 \leq f(x_1, \dots, x_t)(V_k \cdot V_k)$$

or

$$x_k^2 \leq f(y_1, \dots, y_t)(V_k \cdot V_k) \quad (4.24)$$

Let $y_i = \delta_{i,j}$ for all i . Then Eq(4.24) can be written as:

$$x_k^2 \leq (V_k \cdot V_k)(U_j \cdot U_j) \quad \text{for } 1 \leq j, k \leq t \quad (4.25)$$

The right hand side of equation of Eq(4.25) is initially still large. So searching x_k up to the upper bound requires tremendous amount of computational work. However, $(V_k \cdot V_k)(U_j \cdot U_j)$ and $(x_1, \dots, x_t)$ can be transformed, by applying Euclidean's algorithm, into $(V'_k \cdot V'_k)(U'_j \cdot U'_j)$ and $(x'_1, \dots, x'_t)$ in which $U'_i \cdot V'_j = \delta_{i,j}$ are preserved and $f'(x'_1, \dots, x'_t) = f(x_1, \dots, x_t)$. Let j be a fixed subscript, $1 \leq j \leq t$. The transformation relations are:

$$\text{for a fixed } j \begin{cases} V'_j = V_j \\ U'_j = U_j + \sum_{i \neq j} q_i U_i \\ x'_j = x_j \end{cases}$$

and

$$\text{for } i \neq j \begin{cases} V'_i = V_i - q_i V_i, \\ U'_i = U_i \\ x'_i = x_i - q_i x_j \end{cases}$$

and q_i can be determined by:

$$q_i = V_i \cdot V_j / V_j \cdot V_j \quad (4.26)$$

where $i = 1, \dots, t$.

Based on the above discussion, the Spectral test can be implemented. The algorithm was described in eleven steps. It is implemented as a function `KnuthSpectral()` in this work. The function calls two other functions to perform two major routines in the algorithm: One is `transform()` and the other is `search()`. The program code is discussed below and the complete listing can be found in Appendix A.

```

1
2  #include      "Kclass.h"
3  #define      min(x,y)      ( (x > y) ?  (y) :  (x) )
4  #define      SIZ      10
5  static Fixed  U[SIZ][SIZ], V[SIZ][SIZ];
6  static int    t, j, k;
7  static Fixed  s;
8
9  void  transform(outstream , int);
10 void  search(Fixed);
11 int   KnuthSpectral(outstream, Fixed, Fixed, int, int, Fixed *);
12
```

Code fragment 4.2

The algorithm is implemented using data type `Fixed` which is defined in the included file `Kclass.h`. Currently, the program is designed to deal with tests up

to ten dimensions. Matrices U and V are defined in Line 5 in code fragment 4.2. t , j , k are used to index subscripts of the structured variables. s holds the result in a designated dimension. These variables are used in the main function and two auxiliary functions.

```

1
2      KnuthSpectral(outstream fp, // file used to report errors
3                    Fixed a,      // tested multiplier
4                    Fixed m,      // modulus
5                    int dim,      // requested dim test (<= 10)
6                    int debug,    // flag to trace execution
7                    Fixed nu[])   // output parameter, the results
8      {
9
10
11         Fixed  h, hprime, p, pprime, r; // used in initial
12         int    jmp;                      // loop flag
13         Fixed  u, v, q;                  // intermediate results
14         Fixed  Vij, Vjj;                 // hold sums of V
15         Fixed  Ujj, YY;                  // hold sums of U
16         int    i;                        // for loop counter
17         int    T;                        // dimension indicator
18         Real   r1;                       // only Real used
19
20
21         if ( dim > SIZ || dim < 0 ) {
22             fp << "\ KnuthSpectral:  error dim ?  " << dim;
23             return(-1);
24         }
25
26         t = 1;                            // t stands for the current tested dim
27         T = dim - 1;                       // T is the last dim to be tested
28         nu[0] = 0;                         // v^1 is set to 0
29

```

Code fragment 4.3

`KnuthSpectral()` takes six parameters. The first parameter, `fp`, has `ostream` type which is defined in `Kclass.h`. One purpose of using the output stream is to report errors in the input parameters, shown in lines 21 to 24 in code fragment 4.3. The next two parameters are the parameters of the random number generator being tested. The fourth parameter takes a valid value in range $[1, 10]$, which stands for the highest dimension which the test goes up to. The other purpose of

using the output stream is to allow users to observe the program execution. One value that may be interesting to observe is q discussed earlier, Eq(4.26). If the fifth parameter is set, then all generated q 's will be written onto the output stream. The last parameter returns the results from the test. The first result, ν^2 , is stored in the subscript 1 of the array. Subscript 0 of the array is set to 0. The last result, ν^{10} , will be stored in the subscript 9 of the array. The function will return 0 to indicate success, or -1, otherwise.

`KnuthSpectral()` performs four tasks: initializing the test by using the input parameters; calculating the test in two dimensions; preparing for the higher dimension tests and calling two separate functions in which transformation and searching are performed. These two functions are also used in the program that implements the Percus and Kalos's Spectral test, as well. The following is the actual code that implements the S algorithm. There is no line by line explanation. For the serious reader, this code has to be read with the original algorithm.

```

30
31     h = a;
32     hprime = m;
33     p = 1;
34     pprime = 0;
35     r = a;
36     s = 1 + a*a;
37
38     do {
39         q = Fixed( Rfloor( Real( hprime ) / Real( h ) ) );
40         u = hprime - q * h;
41         v = pprime - q * p;
42
43         jmp = 0;
44         if ( s > (u*u + v*v) ) {
45             s = u*u+v*v;
46             hprime = h;
47             h = u;
48             pprime = p;
49             p = v;
40             jmp = 1;

```

```

51         }
52     } while ( jmp );
53
54     u = u - h;
55     v = v - p;
56     if ( s > (u*u + v*v) ) {
57         s = u*u + v*v;
58         hprime = u;
59         pprime = v;
60     }
61

```

Code fragment 4.4

Lines 31 to 36 initialize some variables using the input parameters. A conditional loop constructed in Line 38 to Line 52 computes the Euclidean transformation for two dimensions, since it can be applied directly. Lines 54 to 60 compute the result and store it in *s* for the second dimension. The function uses the following instructions to prepare for higher dimension tests. Higher dimensional tests are computed with the help of matrices *U* and *V* which are initialized as:

```

62
63     U[0][0] = -h;
64     U[0][1] = p;
65     U[1][0] = -hprime;
66     U[1][1] = pprime;
67
68     if (pprime > 0) {
69         V[0][0] = -pprime;
70         V[0][1] = -hprime;
71         V[1][0] = p;
72         V[1][1] = h;
73     } else {
74         V[0][0] = pprime;
75         V[0][1] = hprime;
76         if ( !( p > Fixed(0) ) ) {
77             V[1][0] = p;
78             V[1][1] = h;
79         } else {
80             V[1][0] = -p;
81             V[1][1] = -h;
82         }
83     }
84

```

Code fragment 4.5

The long if statement from Line 68 to Line 83 in code fragment 4.5 constructs matrices U and V. The way to construct the matrices is not clear in the original algorithm. It was found that by this setup, $U_i \cdot V_j = \delta_{i,j}$, are guaranteed in the subsequent iterations. The rest of the function is a loop that terminates when the requested test completes.

```

85
86     while (1) {
87
88         nu[t] = s ;
89         if (t == T)
90             return(0);
91
92         t += 1;
93
94         r = a * r % m;
95
96         U[t][0] = -r;
97         for (i = 1; i < t; i++)
98             U[t][i] = 0;
99         U[t][t] = 1;
100        for (i = 0; i < t; i++)
101            U[i][t] = 0;
102
103        for (i = 0; i < t; i++)
104            V[t][i] = 0;
105        V[t][t] = m;
106
107        for (i = 0; i < t; i++) {
108            q = (Fixed) Rfloor( (Real) (V[i][0] * r)
109                / (Real) m + (Real) 0.5 );
110
111            V[i][t] = ( V[i][0] * r - q * m ) ;
112
113            for (j = 0; j <= t; j++)
114                U[t][j] = U[t][j] + q * U[i][j];
115        }
116
117        Ujj = 0;
118        for (i = 0; i <= t; i++)
119            Ujj = Ujj + U[t][i] * U[t][i];
120
121        s = min(s,Ujj);
122
123        k = t;
124        j = 0;
125
126        transform( fp, debug );
127        search( m );

```

```

128     }
129
130 }
131

```

Code fragment 4.6

Line 88 in code fragment 4.6 stores the result from the previous step in the t subscript of the output array. Then in line 90, the function returns 0 if the requested work has been done. Line 92 increases the dimensional subscript to continue the computation. Lines 96 to 101 prepare the matrix U for the newly added row and column. Lines 103 to 105 add a new row to the matrix V . Lines 107 to 115 add a column to V , and meanwhile, apply the Euclidean algorithm so that the values assigned to the new column will be reduced. During the computation of V , U is updated in Lines 113 to 115 by employing the transformation. Lines 117 to 121 choose a new value s . After preparing for the next step of the computation, `transform()` and `search()` are called in Line 126 and Line 128, respectively.

Function `transform()` uses the Euclidean algorithm to reduce the search upper bound $V_i \cdot V_j / V_j \cdot V_j$.

```

1
2  void
3  transform(outstream fp, int debug) {
4      #define          KSGUBED          if (debug)
5
6      Fixed    Vjj, Vij, Ujj;
7      Fixed    q;
8      Real     r1;
9      int      jmp;
10
11      do {
12
13          Vjj = 0;
14          for ( int ii = 0; ii <= t; ii++)
15              Vjj = Vjj + V[j][ii] * V[j][ii];
16
17          KSGUBED fp << "\ n ( j= " << j+1;
18          for ( int i = 0; i <= t; i++) {

```

```

19         if ( i != j ) {
20             KSGUBED fp << " q" << i << " = ";
21             Vij = 0;
22             for ( ii = 0; ii <= t; ii++)
23                 Vij = Vij + V[i][ii] * V[j][ii];
24
25             if ( Rabs(Vij) * 2 > Vjj ) {
26
27                 r1 = (Real) Vij / (Real) Vjj
28                     + (Real) 0.5;
29                 q = (Fixed) Rfloor( r1 );
30                 KSGUBED fp << q;
31                 for ( ii = 0; ii <= t; ii++) {
32                     V[i][ii] = V[i][ii] - q
33                         * V[j][ii];
34                     U[j][ii] = U[j][ii] + q
35                         * U[i][ii];
36
37                 }
38
39                 k = j;
40             } else KSGUBED fp << " 0 ";
41         } else KSGUBED fp << " q" << j << " = * ";
42     }
43     KSGUBED fp << ") \n";
44
45     Ujj = 0;
46     for ( ii = 0; ii <= t; ii++)
47         Ujj = Ujj + U[j][ii]*U[j][ii];
48
49     if ( k == j )
50         s = min(s,Ujj);
51
52     if ( j == t )
53         j = 0;
54     else
55         j += 1;
56     } while ( j != k );
57     #undef KSGUBED
58     return;
59 }
60

```

Code fragment 4.7

Lines 13 to 15 compute V_{jj} for $j = 0, 1, \dots, k$. The instructions in Lines 18 to 42 first, compute U_{ij} so that a new q can be obtained. Then the q is used to transform matrices U and V . In the loop, KSGUBED is inserted in several places. (KSGUBED is the reverse sequence of DEBUGSK, representing debugging Knuth' Spectral test.) If the debug parameter of `KnuthSpectral()` is set, newly obtained q will be written onto

the caller provided output stream, `fp`. q is an important indicator to show matrix transformation. If t consecutive $q_t = 0$ are reported, then no more reduction can be done. The transformation is complete. A new s is selected if the transformation has done some useful work.

Next is the last function in the code, `search()`. In `search()`, an exhaustive search takes place

```

1
2 void
3 search( Fixed m ) {
4
5     Fixed   Vjj, YY;
6     Real    r1;
7     int     jmp;
8
9     for ( int ii = 0; ii <= t; ii++)
10        Y[ii] = X[ii] = Z[ii] = 0;
11     k = t;
12
13     for ( int jj = 0; jj <= t; jj++) {
14         Vjj = 0;
15         for ( ii = 0; ii <= t; ii++)
16             Vjj = Vjj + V[jj][ii] * V[jj][ii];
17
18         r1 = Rsqrt( Rfloor( (Real)(Vjj*s)/(Real)(m*m)
19                        + (Real)0.5 ) );
20         Z[jj] = (Fixed) Rfloor( r1 + (Real)0.5);
21     }
22
23
24
25     while (1) {
26
27         while ( X[k] == Z[k]) {
28             k -= 1;
29             if ( k < 0) {
30                 jmp = 1;
31                 return;
32             }
33         }
34
35         X[k] = X[k] + 1;
36         for ( ii = 0; ii <= t; ii++)
37             Y[ii] = Y[ii] + U[k][ii];
38
39         do {
40             k += 1;
41             jmp = 0;
42             if ( k <= t ) {

```

```

43             X[k] = -Z[k];
44             for (ii = 0; ii <= t; ii++)
45                 Y[ii] = Y[ii] - 2 * Z[k]
46                     * U[k][ii];
47             jmp = 1;
48         }
49     } while ( jmp );
50
51     YY = 0;
52     for ( ii = 0; ii <= t; ii++)
53         YY = YY + Y[ii] * Y[ii];
54
55     s = min(s, YY);
56     k -= 1;
57     if ( k < 0 )
58         return;
59 }
60 }
61

```

Code fragment 4.8

Line 10 in code fragment 4.8 initializes three arrays. $X[]$ is used to store $x_i, i = 0, \dots, t$, the values currently being tested. $Z[]$ will be the upper bound of searched x_i . $Y[]$ will be used to store the values that equal to $x_1 U_1 + \dots + x_t U_t$. Lines 13 to 22 compute the upper bound $Z[]$. Then the routine enters a loop in Lines 25 to 59, and returns when the exhaustive search completes in line 27 to 33. Line 35 increases $X[k]$ indexed by the subscript k . Meanwhile, $YY = x_1 U_1 + \dots + x_t U_t$ is computed in line 36 to 53. In line 55, a smaller s is selected as the result of the test in the t th dimension.

Appendix B is a sample program that calls `KnuthSpectral()` and produces a table testing 30 random number generators. In Appendix C, the 30 generators are listed and associated with the outputs of the test, 29 generators out of 30 are given by [26]. The 28th generator is the one that is implemented in this work. The table shows that the generator passes the Spectral test with high scores in all three

categories.

Unfortunately, in this implementation, the include file "Kclass.h" and object file "Kclass.o" are needed to produce a run module. These files, as well as all other source files discussed later, can be obtained by contacting with Jian Chen or Professor Paula Whitlock via e-mail addresses: `jerry@sci.brooklyn.cuny.edu` or `whitlock@sci.brooklyn.cuny.edu`.

4.4.4 Implementing the Percus & Kalos' Spectral test

The γ 's in Eq(4.17) can be determined by:

$$\gamma_i \equiv b_i + (a - 1)x_{0,i} \pmod{m} (i = 1, \dots, t)$$

by letting $x_{0,i} = 0$, $\gamma_i = b_i$, where b_i are integers modulo m and $\gcd(b_i, m) = 1$. γ_o^{-1} are multiplicative inverse elements of γ_i in the modular system $Z_m^\dagger$. In the algorithm, primes, $p_i < m$, can be selected as b_i , because $\gcd(p_i, m) = 1$ and inverse elements of the primes can be uniquely determined. The method how to select the b_i to test will be given shortly. Below, how to compute the multiplicative inverse elements is discussed. The number theory provides the following theorem:

$$\gcd(b_i, m) = 1 \text{ if and only if } b_i x_i \equiv 1 \pmod{m} \text{ has a solution, } x_i \quad (4.27)$$

To find x_i , the Euclidean algorithm is employed, that is the $q = ab + r$ is repeatedly applied. For instance, to find the multiplicative inverse of 117 in a modular system Z_{256} , Eq(4.27) is used by first applying the Euclidean algorithm as follows:

$\dagger$ Z_m is used to denote a modular system, from 0 to $(m-1)$

$$256 = 2 * 117 + 22$$

$$117 = 5 * 22 + 7 \quad (4.28)$$

$$22 = 3 * 7 + 1$$

then to find the inverse:

$$\begin{aligned}
 1 &= 22 - 3 * 7 \\
 &= 22 - 3(117 - 5 * 22) \\
 &= 22 - 3 * 117 + 15 * 22 \\
 &= -3 * 117 + 16 * 22 \\
 &= -3 * 117 + 16(256 - 2 * 117) \\
 &= 16 * 256 - 35 * 117
 \end{aligned} \quad (4.29)$$

By applying mod 256 to both sides, the above result becomes:

$$-35 * 117 \equiv 1 \pmod{256}$$

$$221 * 117 \equiv 1 \pmod{256}$$

Then the multiplicative inverse of 117 modulo 256 is 221. The function minv()

that carries out the above algorithm is listed below:

```

1
2      #include      "Kclass.h"
3
4
5      static Fixed euclidean( Fixed, Fixed, Fixed, int *, Fixed * );
6
7      Fixed
8      minv( Fixed m, Fixed a ) {
9          Fixed  t, nem, den;
10         int    flg = 0;
11         Fixed  b[6];
12
13         for ( int i = 0; i < 6; i++ )
14             b[i] = 0;
15
16         nem = m;
17         den = a;
```

```

18
19         if (nem < den) {
20             t = nem;  nem = den; den = t;
21         }
22
23         return( euclidean( den, nem, nem &flg, b ) );
24     }
25

```

Code fragment 4.9

The function is written by using the Fixed data type so it can be used for any size modular system. The function takes two parameters. a is relative prime to the parameter m . It returns the multiplicative inverse of a in the modular system m . Line 11 defines an array $b[6]$ which is initialized in Lines 13 to 14. It should be realized that in each of the expressions in Eq(4.29) six integers will be used, at most. So the length of the array $b[]$ is defined as 6. An initial flag, flg , in Line 10 is used to initialize some state of the recursive function `euclidean()`. `euclidean()`, which takes five parameters, is first called in Line 23 in code fragment 4.9, and will be called by itself recursively. When it finally returns, the desired inverse is obtained. `euclidean()` is listed as:

```

1
2     Fixed
3     euclidean(Fixed den, Fixed nem, Fixed mod, int *flg, Fixed b[6]) {
4         Fixed  f[4], t;
5
6         if ( (nem % den) != 0 ) {
7
8             t = den;
9             den = nem % den ;
10            f[0] = nem;
11            f[1] = nem / t;
12            f[2] = t;
13            f[3] = den;
14
15            euclidean( den, t, mod, flg, b );
16
17            if ( !(*flg) ) {
18                b[0] = 1;
19                b[1] = f[0];
20                b[2] = (-f[1]);

```

```

21             b[3] = f[2];
22             b[4] = 0;
23             b[5] = 0;
24             *flg = 1;
25             return(Fixed(0));
26         }
27
28         b[4] = (b[2]) * (-f[1]);
29
30         t = b[0];
31         b[0] = b[2];
32         b[1] = f[0];
33         b[2] = t + b[4];
34         b[3] = f[2];
35         b[4] = b[5] = 0;
36
37         if ( b[2] > 0 )
38             return ( b[2] );
39         else
40             return ( mod - (-b[2]) ) ;
41     } else
42         return( Fixed(0) );
43 }
44

```

Code fragment 4.10

`euclidean()` has one long `if` conditional statement, which determines the stopping point of the recursive calls. Line 8 to Line 13 in code fragment 4.10 compute the first part of the work, which is described in Eq(4.28). The instructions after the recursive call in Line 15 will not be reached until Eq(4.28) finds the greatest common divisor, which is 1, or the condition in Line 8 fails. When the condition fails, the function recursively returns. It will return to the place where it was called most recently. In this function, it will return to Line 15 as many times as it was called there. The first time it returns, it initializes an array `f[4]` defined in Line 4. In all the subsequence returns, Lines 28 to 35 will be executed. The statements in Lines 28 to 35 carries out the second part of the work described in Eq(4.29). Finally, it will return a positive value judged in lines 37 to 40, as the multiplicative inverse.

The MIMD Spectral test program was implemented into a function called PercusKalosSpectral() which is listed below. It is the fundamental program used to test parallel random number generator in this work. Like KnuthSpectral(), it calls the function transform() and search().

```

1
2  #include      "Kclass.h"
3  #define      min(x,y)      ( (x > y) ? (y) : (x) )
4  #define      COMMENT(x)
5  #define      SIZ      10
6  static Fixed U[SIZ][SIZ], V[SIZ][SIZ];
7  static int   t, j, k;
8  static Fixed s;
9
10 void transform(outstream, int );
11 void search( Fixed );
12
13 PercusKalosSpectral(outstream fp,
14     Fixed bs[], // bs contains inverses of prm1, prm2, ...
15     Fixed m,    // multiplier
16     int dim,    // desired dimension to be tested
17     int debug,  // trace iterations
18     Fixed nu[]) { // output results
19
20     Fixed h, hprime, p, pprime, r, gam[SIZ];
21     int jmp;
22     Fixed u, v, q;
23     Fixed Vij, Vjj;
24     Fixed Ujj, YY;
25     int i;
26     int T;
27     Real r1;
28
29     if ( dim > SIZ || dim < 0 ) {
30         fp << "\nPercusKalosSpectral:"
31         << " error dim ? " << dim;
32         return(-1);
33     }
34
35     t = 1;
36     T = dim - 1;
37     nu[0] = 0;
38     gam[0] = 0;
39     for ( i = 1; i < dim; i++ )
40         gam[i] = ( bs[0] * bs[i] ) % m;
41
42     h = gam[1];
43     hprime = m;
44     p = 1;
45     pprime = 0;
46     r = gam[1];
47     s = 1 + gam[1]*gam[1];
48

```

```

49     do {
50         q = Fixed( Rfloor( Real( hprime ) / Real( h ) ) );
51         u = hprime - q * h;
52         v = pprime - q * p;
53
54         jmp = 0;
55         if ( s > (u*u + v*v) ) {
56             s = u*u+v*v;
57             hprime = h;
58             h = u;
59             pprime = p;
60             p = v;
61             jmp = 1;
62         }
63     } while (jmp);
64
65     u = u - h;
66     v = v - p;
67
68     if ( s > (u*u + v*v) ) {
69         s = u*u + v*v;
70         hprime = u;
71         pprime = v;
72     }
73
74     U[0][0] = -h;
75     U[0][1] = p;
76     U[1][0] = -hprime;
77     U[1][1] = pprime;
78
79     if (pprime > 0) {
80         V[0][0] = -pprime;
81         V[0][1] = -hprime;
82         V[1][0] = p;
83         V[1][1] = h;
84     } else {
85         V[0][0] = pprime;
86         V[0][1] = hprime;
87         if ( !( p > Fixed(0) ) ) {
88             V[1][0] = p;
89             V[1][1] = h;
90         } else {
91             V[1][0] = -p;
92             V[1][1] = -h;
93         }
94     }
95
96     while (1) {
97
98         nu[t] = s ;
99         if (t == T)
100             return(0);
101
102         t += 1;
103         r = gam[t] % m;
104
105         U[t][0] = -r;
106         for ( i = 1; i < t; i++ )

```



```

107         U[t][i] = 0;
108     U[t][t] = 1;
109
110     for ( i = 0; i < t; i++ )
111         U[i][t] = V[t][i] = 0;
112     V[t][t] = m;
113
114     for ( i = 0; i < t; i++ ) {
115         q = (Fixed) Rfloor( (Real) (V[i][0] * r)
116                             / (Real) m + (Real) 0.5 );
117
118         V[i][t] = ( V[i][0] * r - q * m );
119
120         for ( j = 0; j <= t; j++ )
121             U[t][j] = U[t][j] + q*U[i][j];
122     }
123
124     Ujj = 0;
125     for ( i = 0; i <= t; i++ )
126         Ujj = Ujj + U[t][i]*U[t][i];
127
128     s = min(s,Ujj);
129
130     k = t;
131     j = 0;
132
133     transform( fp, debug );
134
135     search( m );
136
137 }
138
139 }
140

```

Code fragment 4.11

There are only a couple of changes from `KnuthSpectral()`. One of them is the second parameter of the function. The second parameter in this function is an array of type `Fixed`. It contains the γ 's in Eq(4.17). The sequence of γ 's is $\gamma_1^{-1}, \gamma_2, \dots, \gamma_t$, and the last value of the array is γ_1 . All differences from `KnuthSpectral()` are as follows. Lines 38 to 42 compute coefficients for the test based on input γ 's. Line 46 and Line 47 initialize `r` and `s`, which contain results of the test for the given dimension. In Line 103, `r` is assigned to a new value for a higher dimensional test.

4.5 Searching good additive constants

Eq(4.17) specifies the Spectral test for a t disjoint distribution. Practically, t is a small number, usually less than 10, for one test. For finding massive number of generators, the test was applied many times, to each t generators that are tested.

[1] provides methods to choose additive constants, b 's. The method to choose primes as b 's is discussed. Suppose a finite set of primes $\{b_1, \dots, b_m\}$ are chosen to be tested. Let ν_t^* denote the accuracy of the t -dimensional uniform distribution:

$$\nu_t^* = \min_{\substack{\text{all } t\text{-tuples } \in \{(b_1, \dots, b_t)\} \\ b_1 < b_2 < \dots < b_t}} \nu_t(b_1, \dots, b_t). \quad (4.30)$$

In this work, ν_t^* is tested up to $t = 3$. That is: three primes are used in each run of a test. When $t = 3$, Eq(4.30) defined the lower bound of $\nu_3(b_1, b_2, b_3)$, where b_1, b_2, b_3 is a combination chosen from a large pool of primes. Specifically, when $x_{0,i} = 0$ for all i in Eq(4.10), three primes are chosen from the following two sets:

$$\text{Set I} = \left\{ p_l : p_l \text{ prime}, p_l = O(m^{\frac{2}{3}}), \text{ and } p_l < m^{\frac{2}{3}} \right\}$$

$$\text{Set II} = \left\{ p_s : p_s \text{ prime}, p_s = O(m^{\frac{1}{3}}), \text{ and } p_s < m^{\frac{1}{3}} \right\}.$$

It is shown in [1] that whenever b_1, b_2 are members of either the same set or different sets,

$$\nu_2 \geq O(m^{\frac{1}{3}})$$

and whenever $b_1 \in \text{set I}, b_2, b_3 \in \text{set II}$,

$$\nu_3 \geq O(m^{\frac{1}{3}}). \quad (4.31)$$

Therefore, the lower bounds of ν_2, ν_3 are:

$$\begin{aligned} \nu_2^* &= O(m^{\frac{1}{3}}) \\ \nu_3^* &= O(m^{\frac{1}{3}}) \end{aligned} \quad (4.32)$$

whenever one of the b 's $\in$ set I and the other b 's $\in$ set II.

The upper bounds of set I and II are 2^{32} and 2^{16} , respectively when m is set to 48. The lower bounds of these sets are specified by the expressions of $O(2^{32})$ and $O(2^{16})$. For a large m , such as 48, they are 2^{31} integers between $O(2^{31})$ and $O(2^{32})$. Among these integers, they are hundreds of thousands of primes. The primes close to $O(2^{32})$ in set I and $O(2^{16})$ in set II satisfy the requirements: $\nu_2 \geq O(m^{\frac{1}{3}})$ and $\nu_3 \geq O(m^{\frac{1}{3}})$. The MIMD Spectral test can be employed as a supplemental tool to search some lower bounds of primes in the set I and II such that these primes satisfy Eq.(4.32).

An approach to select the sets is that a certain member of primes are chosen in set II as b_2, b_3 , and the largest prime numbers less than 2^{32} are selected or tested to be in set I as b_1 . The upper bound of set II, 2^{16} , is relatively not a very large number. The primes that are very close to the boundary can be considered to satisfy to the requirement in Eq(4.31). To fix the number of members in this set II, the lower bound of set I is searched. This approach, first, simplifies the searching procedure, and second, only the lower bound of set I has to be found. There is a large band of primes to be candidates of set I. It depends on how many parallel generators one wants to find.

The proposed work is to select 10,000 good b 's. the 40 largest primes less than 2^{16} are selected as members of set II. These primes are b_2, b_3 in a test, and satisfies the requirement whenever b_1 in the other set satisfy the requirement. 9,000 largest primes less than 2^{32} were selected to be the members of set I, without going through

the test. They are considered satisfy the requirement of set I. Next 1,000 largest primes less than any of the first 9,000 are chosen with MIMD test. These selections are obviously very conservative. There may be many more primes satisfying the order of (2^{16}) or (2^{32}) in these sets. But, for us, only 10,000 are proposed. If many more parallel random number generators are needed, this scheme can be applied to search the lower bound of the set I.

MIMD Spectral test is used to find some boundaries for these sets. What is wanted is to make sure that no prime in set I causes the MIMD test to fail. It was tested that small primes in set I cause the test to fail.

For each b_1 , there are $\binom{40}{2}$ possible ways to select b_2, b_3 from set II, therefore, it is the number of times to run $\nu_3(b_1, b_2, b_3)$ to decide whether or not $b_1 \in$ set I. To find if there is a b_1 among the 1,000 primes failing, every b_1 has to run against all combinations of b_1, b_2 . To carry out such work on one machine seems impossible. Distributed programs were written, so the demanded work was carried out in a computer network by tens of workstations. During the computation, one server was used to allocate tests, and collect results. Tens of clients ran the MIMD test for any given prime configuration from the server.

4.6 Managing of parallel generators

Mapping pseudo-random sequences to parallel processes was first proposed by Frederickson et. al.[24]. A binary tree, called a Lehmer tree, was used to manipulate two linear congruential generators.

The following diagram shows psuedo-random sequences that can be differen-

tiated by the paths to the leaves from the root of the tree. Two random number generators, called the left and the right, are used as following: the left is used to generate an initial value of a new random sequence and the right is used for all sampling. An initial value is chosen to begin the root of the tree. When a branch is needed, usually responding to process creation, the left random number generator is employed once to supply a seed to the right generator to initialize a new sequence for the newly created process. The newly created process then applies the initial value, the seed, on the right generator to draw random numbers.

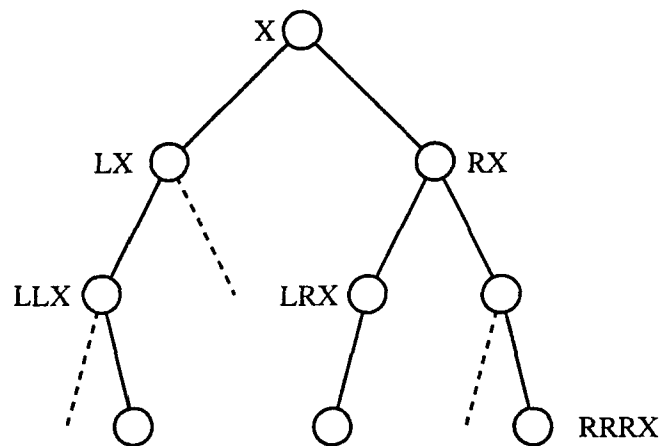


Figure 4.3 A pseudo-random tree

The label of a node is the suffix string of its branches. The prefix of its branches differ in the leading character. Each node is a distinct sequence in terms of its path from the root, or the order that the generators are applied.

The generators need five parameters, a_L, c_L, a_R, c_R and m defined as:

$$\begin{cases} L(X) = a_L X + c_L \pmod{m}, \\ R(X) = a_R X + c_R \pmod{m}. \end{cases}$$

Later, it was found that there were possible correlations by employing such a scheme [28]. Then, Bowman and Robinson proposed a scheme using two multiplicative pseudo-random number generators [28]:

$$\begin{cases} L(X) = aX \pmod{m}, \\ R(X) = a^n X \pmod{m}. \end{cases}$$

They use the same multiplier, a and modulus, m . However, the right generator employs n , the upper bound on the number of processes in a run, to determine pseudo-random sequences. Since n might not be available to most of applications, the proposal seems to have little potential.

In the Percus and Kalos proposal [1], to guarantee that all individual processes employ different generators, parallel generators are maintained in a tree structure. Each node of the tree is used, associated with a worker process of the computation and therefore a generator. The relationships of generators is that for each generator i :

$$x_{i,n+1} \equiv ax_{i,n} + b_i \pmod{m}$$

there are two generators on its left and right subnodes:

$$x_{L,n+1} \equiv ax_{L,n} + b_L \pmod{m}$$

$$x_{R=i,n+1} \equiv ax_{R=i,n} + b_{R=i} \pmod{m}.$$

The generator on the right subnode is the parent generator itself.

The following diagram shows that a complete binary tree can be constructed corresponding to the proposal. Each right node of the tree shares the generator that its parent has. All the left nodes of the tree have different generators. Notice that only eight distinct generators are used to form a complete binary tree of four levels.

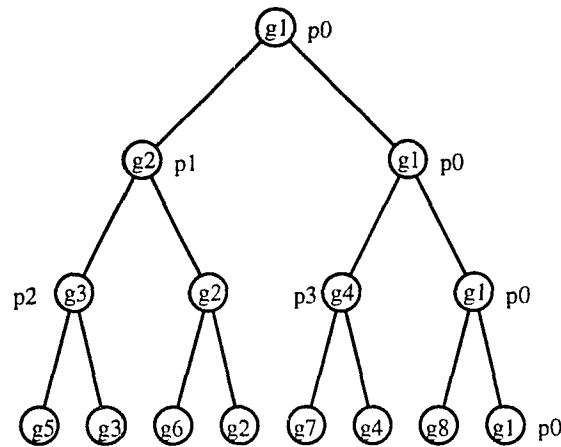


Figure 4.4 Generator tree

In a parallel computation, the initial process, $p0$, uses the generator, $g1$, in the root of the tree. When the initial process $p0$ spawns process $p1$, the generator residing in its left subnode becomes the generator which $p1$ uses. In order to allocate a different generator after spawning the first child process, $p0$ moves to its right subnode and uses the same generator. The tree structure guarantees the sequences of generators allocated to newly spawned processes are unique and reproducible.

The immediate problem is how these generators are associated with worker processes. The basic operations that we need to manipulate the complete binary tree are two operations: *move_to_left_subtree* and *move_to_right_subtree*. These

two operations are used during process creation and moving one process to the *right* position after spawning a process. Since the generators are built in the tree with fixed positions, these two operations can be used simultaneously by multiple processes of a computation to assign generators to newly created processes. For instance, in Fig. 4.4, processes $p0$ and $p1$ can independently allocate generators $g2$ and $g4$ to their spawned processes, $p2$ and $p3$, at the same time. And no a generator is allocated twice.

There are two feasible solutions to implement the operations to manipulate parallel generators. One is to provide these operations as *primitives* to computations with no means of implicitly parallelizing. In this case, users of the generators have to manually employ *move_to_left_subtree* and *move_to_right_subtree* to meet the computations' need. The other solution is to build these operations into some existing library that supports parallel processing, like the distributed process library, *dp*, discussed in chapter 2. The first solution is discussed in the following sections. The discussed primitives can be interfaced into many existing distributed processing libraries. In the next section, the implemented package is discussed, two functions in the package are operationally equivalent to *move_to_left_subtree* and *move_to_right_subtree* together with other utility functions.

4.6.1 The distributed random number generator package

The tree structure discussed above helps to understand the general situation when parallel generators are needed to associate with parallel processes. This is just a part of the problem doing parallel processing. When we discuss distributed processing,

we have to deal with another problem: the computational processes are not located in one host, rather, they are running on different hosts communicating through a network. However, the tree accommodating the generators remains the same: generators reside in fixed positions in the tree, and their relations are recognized by all running processes using them.

The way that the tree is stored also has to be taken into account. If only a small number of generators is used, a copy of the tree can be stored in all local hosts. However, when thousands of generators are proposed, it is desirable to store only one physical copy of the generators in one machine. So an implementation has to not only implement the operations mentioned above, but remotely access the generator tree as well.

A package called DRAND is implemented having all the features discussed above. Let's postpone presenting the part that implements the remote accesses, and discuss the user interface functions first.

There are three groups of DRAND functions to support distributed computations, they all can be called from a user program which is linked with a library, `libdrand.a`, provided by DRAND.

There are four primitives in the first group: `drand_create()` `drand_new()`, `drand_self()` and `drand_remove()`. Among them, `drand_new()` is operationally equivalent to the abstract operation, *move_to_left_subtree*, and `drand_self()` is equivalent to *move_to_right_subtree*.

And three application functions are in the second group: `drand_advance()`,

`drand_branch()` and `drand_fork()`. These functions employ the primitives to provide useful operations on the binary tree of generators. Their uses will be discussed shortly.

Finally, there are five functions in the utility group. They are `drand_random()`, `drand_copy()`, `drand_fprint()`, `drand_encode()` and `drand_decode()`. Some of these functions are provided for convenience, such as `drand_fprint()`, which is used to print a generator to an output file. `drand_random()` is the random number generator that provides pseudorandom numbers. `drand_encode()` and `drand_decode()` are used to encode and decode generators in string format that can be used by operating system function calls, so that the generators can be passed from one location to another via the communication network.

4.6.2 DRAND primitive functions

The package is written in the C programming language for large demands of distributed C or Fortran-applications. A data structure `dhandle` is a representative of a generator in the binary tree. Initially, a logical copy of the binary tree can be obtained by calling primitive `drand_create(ac, av, &handle)`. When the function returns, `handle` contains the first generator of the system, or the root node of the binary tree together with other information, discussed shortly.

The following C program illustrates how `drand_new()` and `drand_self()` are employed in process creation.

```

1      main(ac, av)
2      int      ac;
3      char      **av;
4      {
```

```

5          dhandle handle;
6          int      cid, status, ret;
7          ...
8          drand_create(ac, av, &handle);
9          if ( (cid = fork()) == 0 ) {
10             drand_new(&handle);
11             ...
12             exit(0)
13         }
14         drand_self(&handle);
15         ...
16         ret = wait(&status);
17         ...
18         drand_remove(ac, av, handle);
19     }

```

Code fragment 4.12 `drand_creat()`

Code fragment 4.12 shows that after the `drand_creat()` is called in line 8, a parallel computation can take place. Usually, a pair of related (concurrent) processes is started by calling the system function `fork()` with *if* statement. It is shown in line 9 of the code fragment. `fork()` causes the process to become two different processes: one carries out the instructions between lines 10 to 12 in the *then* part, and the other carries out the instructions between lines 14 to 18 in the rest of the program. However, before executing their instructions, both processes have the same program state: they have the same variables and the same values in them.

Now, let's go back to our generators. The process that receives 0 when `fork()` returns is the child process, which calls `drand_new(&handle)` to move itself to the left subtree. Before the child process calls `drand_new(&handle)`, the information stored in `handle` is exactly the information obtained from `drand_creat(&handle)`. After it calls `drand_new(&handle)`, there is a new generator in the `handle`. This operation is analogous to *move_to_left_subtree* discussed earlier.

The process that receives a non-zero value is the parent process. In or-

der to prepare itself to spawn more children, the parent process has to call `drand_self()`. When `drand_self(&handle)` returns, the parent still has the same generator, but position of the generator in the tree is changed. This is analogous to *move_to_right_subtree*. In general, the `wait()` statement in line 16 is needed to make sure that the child process has terminated, before calling `drand_remove()`, whose function is to make this logical copy of generators not available. The following diagram is provided to show the situation when these two operations take place.

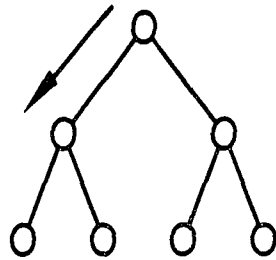


Figure 4.5 (a) new

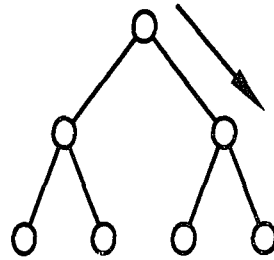


Figure 4.5 (b) self

4.6.3 Paged binary tree

Theoretically, the tree that is expected is a complete binary tree. Each of the interior nodes has left and right subtrees. When worker processes are associated with the nodes of the tree, all the processes have the same properties, such as each process can spawn new processes and grant new generators to them. To construct a complete binary tree of 16 levels, 32768 generators are needed. Although such a large number of generators are prepared, a computation can not spawn more than 16

generations of processes, which does not satisfy many computational simulations.[†] A complete binary tree structure is suitable only for a computation in which every worker process is guaranteed to spawn new worker processes.

In general, the creation of new processes is a random variable based on a computation's needs. In the simulation of a random walk, for example, a new process is needed based on some probability. So the tree actually needed is not necessarily complete. The actual shape of the tree is variant, or a binary tree computationally generated. The tree automatically expands itself. It is clear that if the creation of processes is based on probabilities that are determined by pseudorandom numbers, the sequence of process creation should be reproducible by another run of the same program. It is concluded that if the reproducibility of creating processes of a computation must be guaranteed, then the reproducibility of using the same generators in the same sequence must be guaranteed.

The solution to the problem of a variant binary tree in computations with birth and death of processes was to introduce a dynamically expanded binary tree. By using the same number of nodes as a complete tree of 16 levels, the dynamically expanded binary tree can be constructed in many hundreds of levels. The expanded binary tree is built up by "page"s or a number of nodes as a fundamental unit whose nodes are constructed as a binary tree. To increase the utilization of generators, the page size is expected to be small, but not too small to avoid unnecessary operational overhead. Consider a binary tree of 15 nodes shown in Fig. 4.6 (a). Because the

[†] A complete binary tree of 16 level has $2^{16} - 1$ number of nodes. However, in this particular tree only half of the node are distinct.

generator in the right subnode is used by a process itself, each node and its right subnode are not distinct. For example, in nodes 1, 3, 7 and 15 there is only one generator needed in these locations. So the generators in nodes 3, 7 and 15 can be considered as the generator in node 1. See Fig. 4.6 (b). A page of a complete subtree can be constructed with exactly the same levels of a binary tree using only 8 distinct generators. Each leaf of the tree will generate two subtrees. A left subtree is always distinct from a right subtree because the nature of the generator tree. The root of a left subtree is a new process and so a new generator. The root of a right subtree is the same process and therefore the generator is inherited from its parent. The configuration of left and right subtrees expanded from node 8 are given in Fig. 4.6 (c) and Fig. 4.6 (d), respectively.

The number of generators in a page of n levels is either 2^{n-1} or $2^{n-1} - 1$, depending on the left expanded or the right expanded. The utilization, u can be computed approximately by: $\frac{n}{2^{n-1}} \leq u \leq 1$. So we expect that the tree in a page has a small value n .

The numbers in the tree are not generator numbers. They are process identifications assigned and manipulated by the generator system. If the sequences of process creation are to be reproduced by an application program in another run, then the processes should be assigned to the same process identification numbers.

When a generator is allocated to a worker process, the position of the generator in the tree and the parameters of the generator are all stored in the data structure, `dhandle`, mentioned in the above section. Some other information about the gen-

erator, or precisely, about the application, is also stored in `dhandle`, which will be introduced shortly. More description of them will be given when the implementation of the distributed environment is discussed in sections 4.6.5 and 4.6.6.

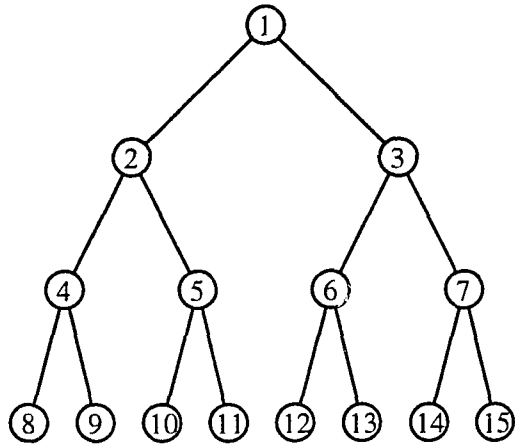


Figure 4.6 (a) subtree in a page.

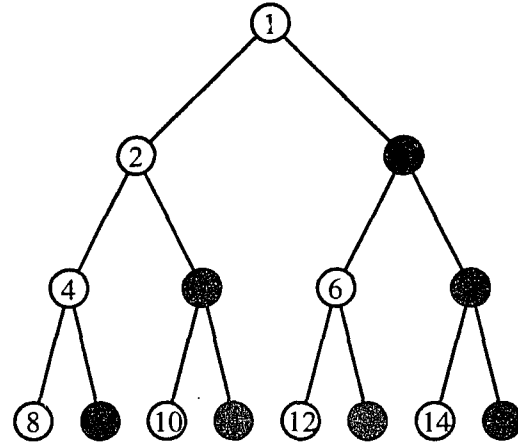


Figure 4.6 (b) reduced page

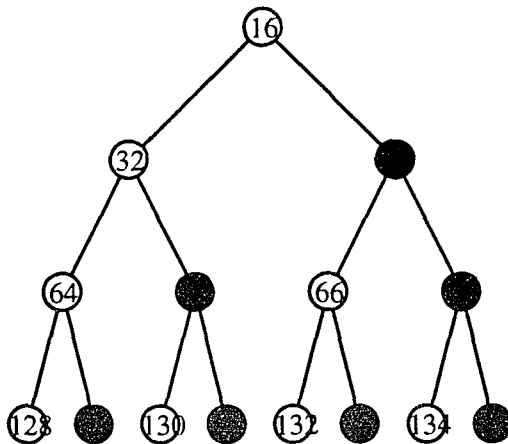


Figure 4.6 (c) left expand

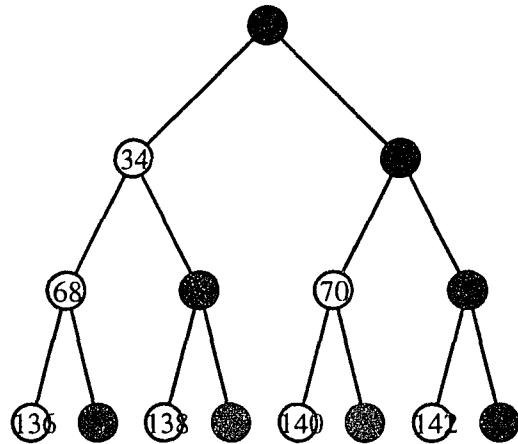


Figure 4.6 (d) right expand

10,000 generators were used and stored in a binary file (a data base), each node occupies 6 byte. (See the previous section, 10,000 good b 's were found and a

48bit constant takes 6 bytes of storage.) The data base is maintained in a network server, called `drand_svc`, The server can support multiple logical copies of the tree while only one physical database file is needed. In the other words, the system supports many applications at the same time.

The distributed random number generators we implemented can be viewed such that each of them not only captures the parameters needed to determine random numbers, but has additional information used by application programs to communicate with the database server as well. Each handle is a generator, represented as parameters, and a state of the generator. The algorithm to produce random numbers is given by the utility function `drand_random`. The server is also considered as a part of the distributed generators, even though it dose not have any parameters needed to produce a random number. Without it, new generators can not be allocated, and reproducibility can not be achieved.

A data structure that represents the main part of the expanding binary tree is used. The data structure is centrally controlled, so it resides in the server. The functionality of the structure is to support multiple applications and to guarantee reproducibility of a run. The problem of central control is that some storage has to be reserved and some maintenance may be needed. Possibly the storage eventually will be full, therefore it causes the server to go down or become not fully functional. In the next three sections, the distributed environment in which the system of generators were implemented and maintained is discussed.

4.6.4 Remote procedure calls

Like ordinary programs where a subprogram can be called from the main body, the remote procedure call (RPC) mechanism allows a the local program to call a procedure that is running on a remote host on a network. A running procedure on a remote host is generally called a server, and local program that calls a remote procedure is called a client. RPC forms a client/server computational environment.

On a Sun network, RPC is built on low level socket interface, TCP or UDP protocols. The advantages of the RPC mechanism is that a message passing program becomes a parameter passing program. Application programs no longer handle messages directly, instead, subprograms, procedures or functions are written. And they are used to handle message communications by passing parameters of the subprograms. The way to construct such subprograms is exactly like the way to construct local subprograms.

The protocols between local calls and remote procedures can be generated by a RPC provided protocol compiler called *rpcgen*, its syntax is specified by a language, RPCL. Data structures and procedure headings in RPCL are pretty much like that in C language. *rpcgen* translates RPCL files into C programs.

Each RPCL file has file extension *.x* in SunOS systems. The most important information that RPC needs to know to generate protocols between a remote procedure and a local call are parameter structures used in the procedure and the procedure definition. *rpcgen* translates a *.x* file into a header file with the extension *.h*, a standard C header file; two stub files and one filter file, all of which have *.c*

extension.

The way that RPC works is as follows: When a remote procedure is called from a local program, it is first mapped into a local procedure. The procedure is located in one of the stub files called the client stub file. Because the remote procedure is specified in the .x file, *rpcgen* produces a local procedure body with the requested procedure definition residing in the client stub file. In the stub file, the called procedure transforms the procedure parameters into eXternal Data Representation format (XDR) by using the filter file. In the filter file, how each simple data structure is transmitted is specified. Then the locally called procedure make a call to a remote procedure by means of the RPC mechanism with the filtered parameters.

On the remote side, *rpcgen* generates a main program in which the desired remote procedure may be called or dispatched. The stub file in which the main program resides is called the server stub file. This main body can be systematically constructed since the remote procedure heading is defined in the .x file. The main program first registers the desired procedure(s), the procedure(s) being called remotely, on the network provided map called *portmap* as a port. So later, a message sent to the port will be delivered to the main program. Then the main program enters an infinite loop, waiting for messages, or calls. The expected call will be launched/delivered from the client stub file through RPC means. Once a call arrives, the main program dispatches the a call to an appropriate procedure, the so-called remote procedure. This procedure is provided by users, its name and parameters match the calling instruction that was issued in the original program

on the local side.

When the procedure in the remote side returns, it first returns to the place where it was dispatched. The returned data structure is formatted into XDR format, and then is sent back through RPC means to the procedure provided in the client stub file running on the local host. Finally, the procedure in the client stub file returns the result to the initiating calling statement in the user's program.

By using *rpcgen*, only the main program on the client side has to be written, from which a remote procedure is called. On the server side, only the actual procedure has to be written. The main program and the subprogram are in different modules. The user client main program has to link with the client stub file and the filter file. The file containing the actual procedure has to link with the main server stub file and the filter file. And the server program has to run before a client issues a remote procedure call.

rpcgen provides a minimal service to make RPC calls. However, sometimes the stub files it generates do not satisfy users' needs. For instance, users want to select a desired protocol, or some other low level RPC routines, or even produce more useful error messages. In these cases, users can write their own stub files by following RPC programming conventions. In this project, the stub files were written by hand with *rpcgen*'s help.

The distributed random number generator system, DRAND, is implemented in RPC server/client format. Its server, *drand_svc* is running on one host in a network. Four functions *drand_creat_1()*, *drand_new_1()*, *drand_self_1()* and

`drand_tool_1()` are remote procedures in the server. One of them will be dispatched when a request is made from local client's program. `drand_tool_1()` has not been discussed. This function is provided for maintaining the server when it is running. The details of this function will be discussed in the next section.

Because two procedure names are used to refer to one particular operation, on the client side and on the server side; let's list the procedure names and where they are located.

`drand_creat()`, `drand_new()`, `drand_self()` are located in the local stub files, which are called directly from users' program. `drand_creat_1()`, `drand_new_1()`, `drand_self_1()`, `drand_tool_1()` are remote procedures located in the server stub file, called by `drand_creat()`, `drand_new()`, `drand_self()`, `drand_tool()`, respectively. The definitions of `drand_creat()`, `drand_new()`, `drand_self()` are in the library, `libdrand.a` provided to ordinary users. `drand_tool()` is provided by another library, called `libdrandadm.a` to *drand administrators*.

Notice that there are no direct correspondences for the function `drand_remove()` and the functions in the application and utility groups to the remote procedures. `drand_remove()` will be mapped into a `drand_creat()` call, therefore, it will use `drand_creat_1()` service in the remote side. `drand_fork()`, `drand_advance()`, `drand_branch()` in the application group employ combinations of `drand_new()` and `drand_self()` calls in the local side first. There are no demands for remote procedure services for the functions in the utility group.

4.6.5 System server

The DRAND system is a distributed environment. The server runs on one host on a network. The resources of the server can be accessed by all other hosts, or clients on the network.

The major resource that the server maintains is a database whose components are the additive constants selected by [1] and the MIMD Spectral test. Besides the header of the database file, all components in the database file are six byte long integers. The file is read-only, so the components in the database are static and every process sees the file exactly the same.

The server maintains an array of database file pointers, each of these pointers belongs to a logical generator tree. So the server can support as many applications at the same time as the number of elements of the array. An application program registers a logical copy by calling `drand_creat()` and unregisters it by calling `drand_remove()`.

A pointer of the database file is advanced by seven or eight six-byte units depending upon whether it is advancing for the left subtree or the right subtree. The distance of movement of a file pointer is said to be a page (a page of generators). The file pointers, or current positions for file reading are stored in the server table which can not be rewound, except when the a logical copy is destroyed.

`handle` is altered by each remote procedure call. The initial position of a page is stored in the handle. In addition, the level and location of a generator being used are also stored in the handle. For example, the first time the `drand_creat(ac, av,`

`&handle`) is called, a generator specified by the initial page position, 1st level in the subtree and node location number 1 will be returned. The file pointer of this logical copy of the generator tree will be advanced to the eighth generator in the database file. That means that the first eight generators have been allocated. When the program which called `drand_creat(ac, av, &handle)` calls `drand_new(&handle)` for the first time, the server detects that there are more generators allocated in this page. So a generator specified by the same page, 2nd level in the subtree and node location number 2 will be returned.

When there are many handles used by many processes in an application program, the only thing that server knows about the file is to advance the pointer when needed. The information needed to specify a generator is stored in both server and client's generator handle.

The server will generate a process identification number for each primitive call, and the number is stored in the application's handle. For instance, `drand_creat(ac, av, &handle)` generates an identification number 1; `drand_new(&handle)` generates a number which is twice the number it received; (the received number is its parent's identification number.) `drand_self(&handle)` generates a number which is twice the number it received plus 1. These numbers, that stand for the absolute positions in the complete binary tree, can be uniquely and simultaneously generated by more than one process which calls the primitives. These numbers are not used until a new page is needed to be allocated. The identification number could be very large. Even though there are only 10,000 generators

prepared in the database, a binary tree can be expanded into hundred of levels, so very large identification numbers are possible. Identification numbers are represented by the data type `Fixed`, and are capable of representing any number needed.

The server also maintains other important information about applications, such as the name of the application file, the client user's login id, the host from which the registration is made and a *job registration identification*.

The user's login id and the host name are used when a user queries the server to list his or her registered applications. *job registration identification* is used to identify an registered application for the purpose of reproducing a run. All these three pieces of information are not provided by users, rather they are obtain by the `drand_creat(ac, av, &handle)` call running on the client side. The *job registration identifications* are also informed to users in order that the users can reproduce specific runs. More details can be found in 4.6.6.2 and Appendices D, E, F.

When a new page is allocated from the database file, an ordered pair (process identification number, page offset in the file) is recorded in the server at the corresponding slot of the system table, which is an array of records. All of these records for one application are used to represent the sequence in which the remote procedures were called by the application, or the history of the application. (Notice that an application may consist of many processes.) The ordered pairs stored in the table are linked together and are pointed to by a pointer with which an application was registered. The pointer is initialized when the first `drand_creat()` is called. Subsequent calls of `drand_new()` and `drand_self()` will update the record

list whenever a new page of generators is allocated.

The system supports reproducing a run of an application based on the *job registration identification*, which is obtained by the first run of the application. The application program can be modified. The *job registration identification* can be submitted with the modified application to the server; the server guarantees to allocate the same generators, if process creation of the application remains the same. This implies that some applications can not be reproduced. For example, if the application was not completed at the first run, the server will reproduce the result up to the place where the first run stopped in the subsequent run. Another example, if some “major” changes were made to the application program, the server will reproduce up to the point that the runs differ. An “additional” primitive function call inserted somewhere in the application program during the modification is considered a “major” change. The server will not detect the change immediately until some new pages of generators are needed. If the modified application requires more pages of generators, the server will detect the change and refuses to provide generators to that modified application. An occurrence of a *page fault* indicates that the second run is not equivalent to the first run.

Some application issues should be made clear. Suppose the following function is written which is used to allocate the generator located at the *n*th level down from the right subtree at a given node, *myself*.

```

1  drand_init(n, myself)
2  int      n;
3  dhandle  *myself;
4  {
5      int      i;
```



```

6      dhandle  child;
7
8      for ( i = 0; i <= n; i++ )
9          (void) drand_fork(myself, &child);
10
11     return ( (drand_copy(myself, child) < 0 ) ?  -1 :  0 );
12 }

```

There is a pitfall when one uses the function in parallel computation to allocate generators for different processes. That problem can be shown by the following construct:

```

1  (void) drand_creat(ac, av, &drand);
2  for ( i = 1; i < maxprocess; i++ )
3      if ( fork() == 0 ) {
4          (void) drand_init(i, &drand);
5          break;
6      }

```

`drand` is used to allocate the root of the generator system in line 1. Then the loop in lines 2 to 6 uses the `drand`, `maxprocess` number of times, to allocate the generator in the `i`th level below. The problem is that each time `drand_init()` is called, by different processes, the root of the tree, `drand` is used in the same way, or say in the same path along the right branch of `drand`. In general, only one process can successfully go through the path and the additional ones will fail to allocate the generators they desired.

One exception to the application is that if `maxprocess` is less than the length of a page of generators, or the height of the complete binary tree resided in a page, all processes will have the generators they desired. The reason it works is that all these processes allocate the generators in the same page rooted in `drand`. From it, all processes compute subnodes in the same way and no any conflict that is generated. Moreover, no new generators are demanded from the server. From the

server's point of view, there is no generators that are released twice.

However, if `maxprocess` is equal to or larger than the length of the page size, some process' demand will be denied by the server. If the path goes beyond the node 15 in figure 4.6, for example, the server allocates two new pages of generators: one relates to node number 30; and the other 31. Once the pages are allocated associated with nodes 30 and 31, additional allocations at the nodes are denied by the server.

To reproduce a run, the `drand_remove()` should not be called in the program. To remove an unwanted logical copy in the server, an option `-d` can be used in the command line when a user program is re-submitted. See the section 4.6.6.2.

4.6.6 Examples of using DRAND

In this section, some examples of using user interface functions are presented. For each user application program, some command options can be used for additional services of DRAND. One typical option is: `-q`, queering the usage of the user who issues this command.

When the server has some difficulties, some actions should be considered first, rather than just killing the program. It should also provide a way to shut down the server in a friendly manner, and not damage users' applications. There is a companion program, called `drand_tool`, provided for maintaining the server. Details can be found in 4.6.6.2 and Appendix D.

4.6.6.1 Example: user interface functions

For convenience, three application functions are provided for a variety of uses of the generators. Instead of call `drand_self()` and then `drand_new()` to use a new generator, `drand_advance(&handle)` can be called to use the first new generator in the right subtree. When `drand_advance()` returns, `handle` will be changed to represent the generator in the right subtree. Calling `drand_self()` may be necessary if the user designates that a child process is always to use the first generator on the left subtree. However, the parent process can use the first generator in its left subtree by calling `drand_new()`. As long as the suggested pattern given in code fragment 4.2 is followed, there should be no a conflict in a computation. Fig.4.7 (a) indicates the situation when advancing a generator.

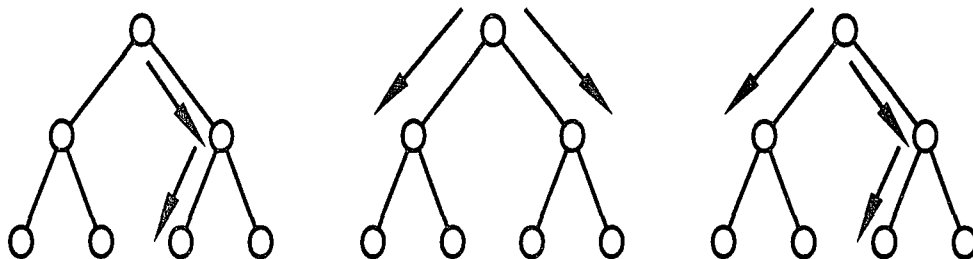


Figure 4.7 (a) advance

Figure 4.7 (b) fork

Figure 4.7 (c) branch

In some applications, two new individual random sequences may be required. In this case, `drand_fork(&parent, &child)` or `drand_branch(&parent, &child)` may be used. When `drand_fork()` returns, parent handle has the same generator in a different position in the tree; child handle has a new generator. On the return

of a `drand_branch()` call, both parent and child handles have different generators. The situations are shown in Fig. 4.7 (b) for `drand_fork()` and in Fig. 4.7 (c) for `drand_branch()`. In either of these calls, multiple handles can be used:

```

1      #define MAX      10 /* or whatever */
2      dhandle   handle, handle_array[MAX];
3      int       i;
4      ...
5      drand_creat(ac, av, &handle);
6      for ( i = 0; i < MAX; i++ )
7          drand_fork(&handle, &handle_array[i]);
8      ...
9      drand_remove(&handle);
10     ...

```

Code fragment 4.13 showing `drand_fork()`.

`drand_fork()` applies functions `drand_copy()` to make a duplicate copy and `drand_new()`, `drand_self()` to set up new positions in the generator tree. The situation is indicated in Fig 4.7 (b).

For programs that have many simultaneous birth events, `drand_branch()` may be used.

```

1      #define MAX      100 /* or whatever */
2      #define threshold 0.5 /* or whatever */
3
4      dhandle   oldhandles[MAX], newhandles[MAX], handle;
5      int       i, j;
6      ...
7      /* suppose oldhandles[0..9] are initiated */
8      /* suppose handle is initiated */
9      j = -1;
10     for ( i = 0; i < 10; i++ ) {
11         if ( drand_random(&handle) < threshold )
12             /* newhandles[++j] and oldhandles[i] all */
13             /* have new generators after the branch */
14             drand_branch(&oldhandles[i], &newhandles[++j]);
15             drand_copy(&newhandles[++j], oldhandles[i]);
16     }
17     ...

```

Code fragment 4.14 showing `drand_branch()`.

The above code either copies an old handle from `oldhandles[i]` to `newhandles[]` or allocates two new generators based on an old handle, `oldhandles[i]` into `newhandles[]` depending on a probability number generated in line 11.

There are five utility functions in the DRAND package. `drand_random( &handle )` returns a pseudorandom number. Function `drand_random()` is a template that codes the algorithm in Eq(4.2). It needs the parameters to compute the next number in the sequence. `handle` provides all parameters the function needs. The difference between `drand_random()` and other conventional `random()` functions is that `handle` determines the sequence, not the function `drand_random()`. `drand_fprint( FILE *fp, dhandle *handle )` prints the generator parameters into the `fp`. `drand_copy( dhandle *dest, dhandle src )` duplicates `src` into `dest`. The last two utility functions are `drand_code()` and `drand_encode()`. To support other distributed message passing systems, such as `dp` programs running on a network, these two functions are needed. In general, distributed processes are spawned through the system call `rexec()`. A handle in string format can be transmitted with the `cmd` parameter, a string, of a `rexec()` call. The parent process calls `drand_create(ac, av, &handle)` to initialize a binary tree and obtain the first generator. Then it calls `drand_encode( &code, handle )` to convert the handle into a string format. And the code and `cmd`, which may be already representing something else, can be concatenated together to form a new parameter `cmd` to a `rexec()` call. The host on the destination side will decode the code by using `drand_decode( code, &handle )`, Then the handle will be used in a call

`drand_new(&handle)`. The situation we illustrated in code fragment 4.2 is suitable for a single host, in which all processes are native to the host and related share one run module. Therefore, there is no need for encoding/decoding the drand handles. In a network, two processes, parent and child, are not located on the same host, and handles have to be passed through the `rexec` command in the literal string format†

The page expanding mechanism is transparent to the users. In the user's point of view, there are no page boundaries, but a complete binary tree, whose nodes are generators. If a new page is needed during a call to `drand_new()` or to `drand_self()`, the database management in the server takes care of it.

4.6.6.2 Example: use DRAND programs

There are two programs provided by the DRAND: `drand_svc`, the server and `drand_tool`, the companion program. All users' programs are client programs of the DRAND. Several options are available for use by these programs, so the options can be employed in a command line when a program is started.

The server can run in two different modes that all relate to application removal. Currently the server is compiled with an internal table of 50 entries. If users do not voluntarily remove their application from the server, the server's table will be full. To handle this problem the server can run either *voluntary* mode or *pre-emptive* mode.

When a command line is issued like: `drand_svc -v`, the server runs in the

† Several mechanisms can be used to pass drand handles (DRAND does not provide any such mechanism to deliver handles). `rexec()` discussed earlier can be used to pass handles to remote processes; `exec()` can be employed to pass handles to unrelated processes in the same host; `sdpspawn()` can be used to carry handles when new distributed processes are spawned, see code fragment 6.1.

voluntary mode. The server itself can not remove any application registered in its table even though it detects the registration has expired. Each client program, when it registers, is given a 24-hour period to stay in the server. It can be modified by an option in the application programs, discussed shortly. When the server runs in this mode and its table is full, all new applications will be rejected. This mode is the default mode when the server program starts with no a `-e` option.

The other option can be used when the server is started by a command line: `drand_svc -e`. This is the pre-emptive mode that the server can remove an application if it needs to. When an application registers itself, it is given a number of hours to stay registered. Then each subsequent access to the server by `drand_new()` and `drand_self()` is used to count down the registered hours. Eventually the number that represents the period it should stay registered becomes negative and it is said that the registration has expired. The expired registrations are subject to be removed by the server when a new application requires services. Even though the server runs in this mode, its table may still be full. Suppose all applications do not access `drand_new()` and `drand_self()` after their initial registrations. The server is unable to update the registration periods.

The person who starts the server program becomes the superuser of the program. The superuser can execute the companion program, `drand_tool`, to query the server for some information; to turn debug on or off; to block new application; to remove an application; and to shutdown the server.

The reason to provide such tool is to monitor the server during it is running

and shutdown the server in a friendly way. The options that can be given with `drand_tool` in a command line for debugging purposes are: `-d0` – turns debugging of all levels off; `-d1` – turns debugging on for the entrance of each monitoring function; `-d2` – turns debugging on for mapping page numbers and generators; `-d3` – turns `-d1,d2` on. The output from the debug option is written into a file called: `drandsvc.log` in the same directory where the server resides. Some of these debugging options produce a lot of output, so watch out for the disk space!

For troubleshooting purposes, several options can be employed. When the server works abnormally, some treatments *must* be tried first before killing the server, the last and worst treatment. `drand_tool -b` – block the server so that no more new application can register. After the server is blocked, `drand_tool -d1` can be issued to see which application causes a problem. If the problem can be identified, then use: `drand_tool -r<table entry>` to remove the application, which is recognized by its table entry. A trouble application may be the one that runs too long, its “history” is so complicated that it needs a lot of computation time. Or it may be one which constantly causes internal problems, that are trapped by the server.

Whenever a `drand_tool` command is issued, the server’s internal table is dumped to the console where the command is given. The output contains all useful information, and it also written into a file called: `drandtool.log` in the same directory where the server program is stored.

When a decision is made to shutdown the server, the command: `drand_tool -t` is used. The server immediately exits. It is a friendly shutdown after realizing

that all registered existing periods are negative.

When a new application program starts, a job registration identification is assigned to the user and it is created as a file name in the user's directory. The format of the identification is: `<program name>_on_<host name>_for_<user login id>_at_<current time>_<demedanded stay>`, where `<program name>` is the name of the application; `<host name>` is the host name on which the application is issued; `<user login id>` refers to the id a user used to login to the system; `<current time>` is the time the application program was created by host system; `<demedanded stay>` is the desired registration period. It is an integer, n , representing a n 24-hour period. Inside of the file, there is some information about this application that DRAND knows. The job registration identification is used to reproduce a run when it is needed.

The following discussion uses `client`, which is compiled and linked at time, Jun 1 19:17, as the user, jerry's application program. The first time when jerry typed: `client on eastham`, a job registration identification is created, like this: `client_on_eastham_for_jerry_at_Jun_1_19_17_1`.

The option for reproducing a run is `-u<job registration identification>`. Suppose jerry modified the `client` program and the new executable module is created at Jun 1 20:10. The command to run the modified program with the job registration identification, obtained at first time, would be: `client -uclient_on_eastham_for_jerry_at_Jun_1_19_17_1`. This run will reproduce the result obtained at the first time. There is no a new job reg-

istration identification created for this run because it uses the `-u` option. However, if a command is typed as: `client`, then a new id, `client_on_eastham_for_jerry_at_Jun__1_20_10_1`, is created for the modified program.

All these program will run, by default, one 24-hour period. The option `-t<n>` is used to alter the registration period. If jerry types: `client -t5 on_eastham` to run the program the first time, the registration would be: `client_on_eastham_for_jerry_at_Jun__1_19_17_5` for a five 24-hour period.

The `-t<n>` option may be very useful for Monte Carlo users. If jerry types: `client -t2`, then types: `client -t3`, then the program will register twice: one runs a two 24-hour period and the other runs a three 24-hour period.

If the client program is not modified and runs twice with *no* a job registration identification given, the second time will produce the result obtained by the first run.

The option `-r<job registration identification>` or `-d<job registration identification>` can be used to remove an application from the server's table. For example, if jerry types: `client -rclient_on_eastham_for_jerry_at_Jun__1_19_17_1`, the registration for the client program is removed. Later, an error would be received if jerry wants to reproduce the result that was calculated before.

The *remove* option `-r` and use option `-u` are mutually exclusive. The first one in a command line is effective. There is another option `-q` for ordinary users. When jerry, as a ordinary user, types: `client -q`, the entries in the server belonging to

jerry are dumped on the console where the command is issued. However when jerry, as the superuser, types: `client -p`, all entries in the server table are dumped. Once the entries are received and then printed, the `client` voluntarily exits. More details can be found in Appendix D+

4.6.7 Critical issues of implementing DRAND

`drand_svc` is a stateful network utility server. It records “history” of accesses when application programs call the procedures in the server. To achieve reliability, it is running upon the Transport Control Protocol, TCP. The server program is long. There is no possibility to list all details in the chapter. However some critical issues should be addressed. In the following sections, some code is discussed with a presence of the header file: `<rpc/rpc.h>`.

4.6.7.1 Parameters of RPC procedures

The parameter of a remote procedure call† and the returned variable are pointers of either simple variables or structures (records). Each remote procedure call takes only one parameter, a pointer. The pointer must be *static* in the local address space. If no parameter is needed, a null pointer should be passed to the remote procedure. The remote procedure may or may not return a value. If it does, the

+ There are many pages available. They are: `drand_svc(8D)`, `drand_tool(8D)`, `drand_creat(8D)`, `drand_demo(8D)`.

† A RPC procedure call consists of four stages: 1) users launch a procedure which resides in the local stub file; 2) the local stub file makes a RPC procedure call to the remote stub file and 3) the identified procedure is dispatched by the server. When the server engine receives the result from the dispatched procedure, it transmits the result back to the local stub procedure; 4) finally, the procedure in the local stub file returns the received result back to the user. In this section, we discuss the parameter used in the stage 2).

returned value is also a pointer of either a simple variable or a structure. The record passing to a remote procedure and the record received when the procedure returns are discriminant records: all variant fields are unioned, and differentiated by a *tag* field. *dhandle* given below is the structure used to pass the parameter to the remote procedures, such as *drand_creat_1()*, discussed earlier. The structure is also used to receive a returned value from the remote procedures.

```

1
2     struct randcore {
3         int status;                /* Validity of this record */
4         u_int mults[4];            /* Multiplier of generator */
5         u_int adds[4];            /* Additive of generator */
6         u_int seeds[4];           /* Seed of generator */
7         u_int loc;                /* Location of a page */
8         u_int level;             /* Level of a page */
9         u_int tabid;             /* Registration entry id */
10        u_int absoffset;          /* Database file offset */
11        char *p_pid;              /* Node location in a tree */
12        char owner[40];           /* User's login id */
13        char host[40];            /* User host name */
14    };
15
16    typedef struct randcore randcore;
17
18    struct dhandle {
19        int errno;                /* Tag of this record */
20        union {                   /* Two fields in the record */
21            randcore info;        /* Used as the parameter */
22            char *c;              /* Used for return, case one */
23            int dummy;            /* Used for return, case two */
24        } dhandle_u;
25    };
26
27    typedef struct dhandle dhandle;
28
```

Code fragment 4.15 Parameter structures

struct randcore is the core part used to store a distributed random number generator. *status* is used to indicate the validity of the generator. Once it is marked *INVALID*, all user interface functions will reject the *handle* when it is received. It is marked *INVALID* whenever an error arises during the computation in a remote procedure. The type of errors that might arise in remote procedures will be discussed

shortly. `mult`, `adds` and `seeds` are array of integers, only 12bit of each integer is used. Four such values make a 48bit long integer. `loc`, `level` and `absoffset` are used to maintain the generator and dialog with the server. `tabid` indicates in which entry in the server's table this application is registered. The `tag` field, `errno`, describes the type of information the record contains. (`errno` does not mean some error.)

`struct dhandle` is the parameter of a remote procedure call and contains the structure result upon return from the remote procedure. `info` field is a structure of `randcore` discussed earlier. Each remote procedure receives a `randcore` as part about `handle`, and all information of a generator is stored there. When the remote procedure successfully accomplish its computation, a new generator is returned via `randcore`, a different one from the received parameter. Notice that when a user makes a remote procedure call, such as `drand_new(&handle)`, `handle` is both the input parameter and the output parameter, with respect to the user. When the procedure in the local stub file communicates with the remote procedure, the parameter to the call and the result received when it returns are two different records. It is the local stub file that makes `&handle` in `drand_new(&handle)` both the input and output parameter.

4.6.7.2 Parameter filters

When data are transmitted across networks, especially on networks of different architectures, the data will be formatted into a desired format, the eXternal Data Representation format, (XDR), by RPC. What XDR does is that for each different

elementary structure, one function is used to encode/decode the structure into XDR format. The following is the XDR function which is used to format the structure `randcore`:

```

1  #include    <rpc/rpc.h>
2  bool_t
3  xdr_randcore(xdrs, objp)          /* RPC defined boolean type */
4  XDR *xdrs;                        /* filter for randcore */
5  randcore *objp;                   /* encoded, XDR, RPC define */
6  {                                /* to be encoded */
7      if (!xdr_int(xdrs, &objp->status))
8          return (FALSE);          /* encode the field status */
9
10     if (!xdr_vector(xdrs, (char *)objp->mults, 4,
11                     sizeof(uint), xdr_uint))
12         return (FALSE);          /* encode the mults[4] */
13     ...                            /* omit adds[4], seeds[4] */
14     if (!xdr_uint(xdrs, &objp->loc))
15         return (FALSE);          /* encode the field loc */
16     ...                            /* omit level, absoffset,... */
17     if (!xdr_string(xdrs, &objp->p_pid, 0))
18         return (FALSE);          /* encode char* pointer, p_pid */
19     if (!xdr_vector(xdrs, (char *)objp->owner, 40,
20                     sizeof(char), xdr_char))
21         return (FALSE);          /* encode owner[40] */
22     ...                            /* omit host[40] */
23 }
```

Code fragment 4.16 randocre filter

As we see, for each field of the struct `randcore`, there is a corresponding encoding/decoding function, which is responsible for formatting the field into XDR format or unformatting a XDR parameter into a designated field. The parameter passed to our remote procedures is filtered by the following function, targeting `dhandle`:

```

1
2  bool_t
3  xdr_dhandle(xdrs, objp)
4  XDR *xdrs;
5  dhandle *objp;
6  {
7      if (!xdr_int(xdrs, &objp->errno))
8          return (FALSE);          /* encode the tag errno */
9
10     switch (objp->errno) {
11     case 0:                        /* reuse demand, when call drand_creat */

```

```

12      case 1:      /* create, when call drand_creat      */
13                  /* new, when call drand_new           */
14                  /* self, when call drand_self         */
15      case 2:      /* delete, when call drand_creat      */
16      case 3:      /* print entries, when call drand_creat */
17      case 4:      /* print all entries, superuser only  */
18      case 7:      /* companion program uses            */
19                  if (!xdr_randcore(xdrs, &objp->dhandle.u.info))
20                      return (FALSE); /* encode for all the opt*/
21                  break;
22      case 5:      /* return server entries                */
23                  if (!xdr_string(xdrs, &objp->dhandle.u.c, 0))
24                      return (FALSE);
25                  break;
26      default:     /* neg, error arised                  */
27                  if (!xdr_int(xdrs, &objp->dhandle.u.dummy))
28                      return (FALSE);
29                  break;
30      }
31      return (TRUE);
32  }
33

```

Code fragment 4.17 the parameter filter

Even though there are only three remote procedures used to serve users's applications, there are more than three functions that are needed. To perform different function using the same call, `errno` will be set to different values. To perform `drand_remove()`, `errno` is set to 2, then `drand_creat()` is employed. Recognizing the value in `errno`, only the corresponding filter is used to format the input field, represented by `objp` in line 3 in code fragment 4.17. On the local side, the stub procedure prepares the `errno ← 2`. On the server side, it is recognized as a removal operation. For another example, the local side calls `drand_creat()` with `errno ← 3` to request dumping the server entries in line 16.

When a RPC procedure returns, the returned record may be one of three possible structures. The reason to have variant structures is that some returned structures accommodate more data than others. Dumping server entries requires a

very long string with no special format, case 5 in the code fragment 4.17. Returning an error, only an integer is needed, no generator should be transmitted, shown in the case default. A normal return requires sending a generator handle, in line 11, the case 0.

When a returned handle contains a `errno=0`, it is a successful remote operation; If `errno<0` is received, there is no handle returned. The local procedure in the stub file will find this situation and returns to the user a `INVALID` handle. If the handle is, again, submitted with another user function call, the call will be rejected by the local stub procedure. If `errno=5` is returned from a remote procedure, the local procedure will print the string it received, then executes `exit()` to terminate the application program.

4.6.7.3 Communication set up

The goal of designing the server is to provide the best possible service to users' long running programs, especially Monte Carlo programs. Therefore, TCP, the reliable protocol, is chosen to run RPC procedures. The following is the way to setup the TCP protocol in the server:

```

1  #include <rpc/rpc.h>
2  #define DRAND_PROGRAM ((u_long)0x071194)
3  #define DRAND_VERS ((u_long)1)
4  void    dispatch();
5  SVCXPRT *transp;
6
7          (void) pmap_unset(DRAND_PROGRAM, DRAND_VERS);
8          /* make a request to obtain a communication port */
9          transp = svctcp_create(RPC_ANYSOCK, 0, 0);
10         if (transp == NULL) {
11             fprintf(stderr, "\n svctcp_create(): failed\n");
12             exit(1);
13         }
14         /* register this server to portmap                */

```



```

15      if (svc_register(transp, DRAND_PROGRAM, DRAND_VERS,
16          dispatch, IPPROTO_TCP) == FALSE) {
17          fprintf(stderr, "\nsvc_register(TCP): failed\n");
18          svc_destroy(transp);
19          exit(1);
20      }
21

```

Code fragment 4.18 setup communication in the server side

In lines 2 and 3 of code fragment 4.18, two constants are defined. When a user program is registered into *portmap*, which is a network server providing services to RPC, the name of the program is represented by a number. The hexadecimal value, 71194, is used as the *name* of this server in line 2. *portmap* helps to route those procedure calls that are sent to this server, so that the desired procedure in the server will be dispatched. In line 3 a constant `DRAND_VERS` is defined, its value is 1. The number will be used in some RPC provided calls. In line 7, an attempt is made to reset a previous registration in *portmap*. In line 9, a try is made to obtain a TCP socket. If it is successful, line 15 registers this server together with the socket number and, a function called `dispatch()`, in *portmap*. `DRAND_VERS` is used to differentiate versions of the server when it is upgraded. If the server is upgraded to version 2 from 1, the client programs that compiled with version 1 will not be served by the new server. So the versions make client/server programs consistent. `svctcp_create()`, `svc_register()` and `svc_destroy` are RPC utility functions. `dispatch()` is a user provided function, which will be discussed shortly.

After registering the server program, the server enters an infinite loop waiting for request calls. There is a RPC provided function, called `svc_run()`. The functionality of `svc_run()` is to run infinitely. However, it does not satisfy our server's

needs because there is an *exit* in the function when it receives some requests that it can not be handled. To support multiple clients in our server, this *exit* is not allowed. Although the server can not serve a bad request, it has to continuously support other requests. The infinite loop is, therefore, constructed as:

```

1
2     fd_set  readfdset;
3     int     tsize;
4
5     /* the number of file descriptor allowed */
6     tsize = getdtablesize();
7
8     while (1) {
9         /* inspect RPC requests with respect to sockets */
10        readfdset = svcfdset;
11        /* take RPC requests */
12        switch (select(tsize, &readfdset, (fd_set *)NULL,
13                      (fd_set *) NULL, (struct timeval *) NULL)) {
14            case -1:          /* DON'T exit! */
15            case 0:           /* No action for it */
16                continue;
17            default:          /* A request comes */
18                svc_getreqset(&readfdset);
19        }
20    }
21
```

Code fragment 4.19 for the server runs an infinite loop

Line 2 of code fragment 4.19 defines `readfdset` with the `fd_set` type defined by RPC. `readfdset` is used to hold a class of file descriptors. In our case, it will hold a set of RPC sockets descriptors for reading. Line 6 `getdtablesize()` is called to obtain the maximum possible number of file descriptors that are allowed to open in a class of descriptors, and stores the number into `tsize`, declared in line 3. The code in lines 8 to 20 is the constructed infinite loop. Line 10 sets RPC file descriptors to `readfdset`. This line must be placed in the infinite loop because RPC requests come from different sockets from time to time. In line 12, `select()`, the socket selection function is called. It inspects `readfdset`, a class of socket addresses, and tries to

find a request to read. This call is programmed to be blocked, denoted by the fifth parameter, a NULL pointer to the call, until a read request arrives. The third and fourth parameters of the call are NULL pointers, meaning that neither a write nor an exception pending descriptor is expected.

4.6.7.4 RPC procedure calls

Once the communication protocol is settled, remote procedure calls can be accepted by the server. There are some functions that have to be written. One of them is the dispatch function; the others are the functions to serve remote demands. One is needed for each demand corresponding to users calls. For example, in our server, there are three primitives that can be called by users: `drand_creat_1()`, `drand_new_1()` and `drand_self_1()`, and one for the companion tool program, `drand_tool_1()`. See section 4.6.4 for the relationships between the function users call and the remote procedures that reside in the server. The following is a part of the `dispatch()` function:

```

1
2
3  #define DRAND_CREAT ((u_long)4) /* define a constant          */
4  dhandle *drand_creat_1();      /* prototype of drand_creat_1 */
5
6  static void
7  dispatch(rqstp, transp)
8  struct svc_req *rqstp;
9  SVCXPRT      *transp;
10 {      /* dispatch one of the unioned function */
11     union {
12         dhandle drand_tool_1_arg;
13         dhandle drand_creat_1_arg;
14         dhandle drand_self_1_arg;
15         dhandle drand_new_1_arg;
16     } argument;
17
18     bool_t (*xdr_argument)(), (*xdr_result)();
19     char *result;
20     char *(*local)();

```

```

21      ...
22
23      switch (rqstp->rq_proc) {
24
25      case DRAND_CREAT:          /* handle drand_creat() */
26          xdr_argument = xdr_dhandle; /* the filter function */
27          xdr_result = xdr_dhandle; /* the filter function */
28          local = (char *(*)) drand_creat_1; /* dispatch this */
29          break;
30      ...                        /* handle drand_new() */
31      ...                        /* handle drand_self() */
32      ...                        /* handle drand_tool() */
33      default:
34      ...
35      }
36
37      ...
38
39      svc_getargs(transp, xdr_argument, &argument);
40      result = (*local)(amp;argument, rqstp);
41
42      /* sending reply is discussed in 4.6.7.7 */
43
44      return;
45  }
46

```

Code fragment 4.20 the dispatch function in the server

Code fragment 4.20 should be read with code fragment 4.18 in which this dispatch function is registered in the RPC portmap together with the server program. In line 3 in code fragment 4.20, DRAND_CREAT, which is also in the local stub file, is defined as a constant, 4. On the local side, this number is used to select a remote procedure. Lines 11 to 16 define the parameter, a union, to a dispatched function. Line 18 declares two pointer functions, each represents a filter function, argument or result. Line 20 defines another pointer function, which will be used to point at a designated function. Line 23 evaluates the field, rq_proc of rqstp received by the dispatch function in its first parameter. Lines 25 to 29 makes a call to the remote procedure drand_creat_1 once DRAND_CREAT is recognized by the evaluation in line 23. Lines 26 and 27 assign the addresses of two filter functions to xdr_argument, which is used to filter the argument to drand_creat_1() and to xdr_result, which

is used to filter the result when the result is sent back to the local side illustrated in code fragment 4.22. In line 28, an address of a designated function, `drand_creat_1` is assigned to the variable `local`. Some lines are omitted following line 29. Line 40 is used to unformat the received parameter in XDR format to the variable `argument` in `dhandle` format. Finally, the designated function is called in line 41. The lines following line 41 will be discussed in code fragment 4.22 since sending a reply will involve some difficulty if the receiving side process does not exist.

On the client side, the definition of `drand_creat()` is defined in a library, `libdrand.a`, which is linked with the users' programs. Once a user's program makes a call to `drand_creat()`, for example, the function `drand_creat()` in the library is executed. After parameter checking and some computations for preparing for a remote procedure call, a remote `drand_creat_1()` call is made.

```

1
2  drand_creat(ac av handle)
3  int      ac;
4  char     **av;
5  dhandle handle;
6  {
7      static dhandle arg;
8      dhandle res;
9      struct timeval TIMEOUT = { 60, 0 };
10     CLIENT      *clnt_handlep;
11     ...
12     /* handle options, parameter checking */
13     ...
14     /* obtain shell environment */
15     strcpy(DRANDSERV, getenv("DRANDSERV"));
16     /* connect to the server */
17     clnt_handlep = clnt_create(DRANDSERV, DRAND_PROGRAM,
18                             DRAND_VERS, "tcp");
19     ...
20     bzero((char *)&res, sizeof(res));
21     /* launch a remote procedure call to drand_creat_1() */
22     clnt_call(clnt_handlep, DRAND_CREAT, xdr_dhandle, &arg,
23             xdr_dhandle, &res, TIMEOUT);
24
25     /*
26     now, the client side received the result from the remote
27     procedure. The way returning the result back to user's

```

```

28      drand_creat() will be discussed shortly because the
29      result has to be decoded before submit to user.
30      */
31 }

```

Code fragment 4.21 local side calls a remote procedure

In line 7 in code fragment 4.21, a variable `arg` of type `dhandle` is declared, which is used to pass the parameter to the remote procedure `drand_creat_1()`; it must be declared as `static`. In lines 11 to 13, which are omitted, the option is extracted from `av` and assigned to `arg`. Line 15 calls `getenv()` to obtain a shell environment variable, `DRANDSERV`, which tells where the server is running. Lines 17 and 18 contain a `clnt_create()` call to connect the server for `DRAND_PROGRAM` and `DRAND_VERS` by using TCP protocol. On a return, a client handle, `clnt_handlep` represented as a pointer, is obtained in line 17. Some code that checks the return value is omitted. For a failed `clnt_creat()` call, this function has to return an appropriate message to user's program. The error handling will be discussed in the next section.

`clnt_call()` is called in line 22. `clnt_call()` is a RPC provided vehicle to carry users remote procedure calls. Upon success, `clnt_call()` provides `res`, which is the result sent by the remote side. `clnt_call()` uses the handle `clnt_handlep` obtained in line 17, to access the remote procedure, `drand_creat_1()`, represented by `DRAND_CREAT`. The call is programmed with a time out clock of 60 seconds, seen in line 9. However, code that makes some efforts to retry the communications from line 17 to line 23 is omitted. The function call also uses the same filter, `xdr_dhandle`, to pass the input parameter `&arg`. Failure of all tries involves some error handling that is not shown here in lines 25 to 30. The error handling will be discussed in section 4.6.7.5.

Now, the remote procedure call is complete. The only code left in the function is to analysis the returned value in `res`, and make an appropriate return, shown in the next section.

4.6.7.5 Return to user's programs

Returning the result to a user's program is an important matter. There are thirty different error messages associating with errors arised in the remote side. Some errors report problems of the server program. For example, an error message: `System error: can not locate the desired location` indicates that the location of the generator in a page is wrong. This error may be introduced during parameter transmission. When parameters are passed through a network which involves two memory address spaces, error detections are a must to ensure the server operational. Another error may be: `Error: No such copy to delete`. When a user issues a deleting operation and there is no such entry in the server's registration table, this error arises in the server side. A less common error message may look like this: `Error: Out of capacity: too many levels of the random number tree`. Virtually all generators in the database can be used. The higher level a tree is built, the more dynamically allocated memory is consumed. There is an upper limit which when reached generates this error in the server. When a user's program tries to build a tree more than 2000 levels, this error arises.

```

1
2      /* continuation to line 25 in code fragment 4.21 */
3      if ((char *) &res == NULL) { /* failed in the local side */
4          sprintf(drandmsg,
5                  "drand_creat(): bad rpc call");
6          handle->dhandle_u.info.status = INVALID;
7          return(-1);

```

```

8      }
9
10     if ( res.errno == 5 ) {          /* dump server's table      */
11         fprintf(stdout, res.dhandle.u.c);
12         ...
13         exit(0);                    /* yes, no more work      */
14     }
15     else if ( res.errno != 0 ) {
16         sprintf(drاند_msg, "%s", errormessages[abs(res.errno)]);
17         handle->dhandle.u.info.status = INVALID;
18         return(-1);
19     }
20
21     (void) drاند_copy(handle, &res);
22
23     return(0);

```

Code fragment 4.22 return to users call

When the *clnt_call()* fails, there is not much what we can do. Lines 3 to 8 in code fragment 4.22 write a message into an external variable, *drاند_msg*, which is obtained in the header file, *drاند.h*, and then makes the *handle* invalid. The constant, *INVALID* is also defined in the header file. If the users' program checks the returned function value, it will find (-1), an error, and then should print *drاند_msg*. Lines 5 to 14 are used to print the server's table, which is required by the local side. After printing the result, the program is terminated since the printing command is issued from a command line by the option -q. In lines 15 to 19, if a well-known error arises, the returned error code is used to index an error message stored in *errormessages*, and the message is saved into *drاند_msg* for return. The *handle* is also marked *INVALID*. If everything goes fine, in line 21 the result is copied into *handle*, and passed back to users' program in line 23.

4.6.7.6 Faults tolerance

In any event, it is not allowed that server is killed by an interrupt or a voluntary

exit. In the server program, there are no `exit()`s, except at the beginning when the server fails to initialize itself and the place designed to exit, for shutting down. Some difficulties that the server may face were studied, and there are some handling for those problems in the program. Other difficulties are synchronous interrupts, such as a memory fault or a segmentation violation. In the server program, quite a lot of dynamic memory allocation operations are used. Some memory addressing problems may occur.

The ideal handling is that it should be only affect the application which causes the fault, and the server remains to operate. A `handle` may contain a very long data; for instance, it contains very long tree position id. Although every handle is encoded by XDR, there were found that some arguments were not passed correctly. When that happens, it most likely triggers a segmentation error or bus error in the server program.

The solution of this is that interrupt handlers are installed in some functions in which there is a potential place where an interrupt will occur. When an interrupt occurs, the control is passed to the end of the function. Then the function returns an negative integer back to the local call. The local side library function will manipulate the error code and mark the user's `handle` `INVALID`.

Meanwhile, the server will write information for the interrupt into a file called `drandcatch.log`. During the period of testing the program, all interrupts were trapped. The server program continued to run.

4.6.7.7 Handling died clients

Because the RPC procedures run under TCP protocol, there may be a problem when the remote procedure sends a reply back to a dead client. The operation the remote procedure uses to send the reply is blocked. If the client has died, then the server will be blocked on a write operation, and no more requests will be accepted. To overcome the difficulty, an asynchronous interrupt is used to unblock the write operations. Code fragment 4.23 defines an interrupt handler and code fragment 4.24 is used in the dispatch in line 43 of code fragment 4.20.

```

1
2     jmp_buf    jbuf;
3     int        intrno;
4     void
5     catcher(sig)
6     int        sig;
7     {
8         longjmp(jbuf, sig);
9     }
10

```

Code fragment 4.23 an interrupt handler

In line 2 of code fragment 4.23, `jbuf` is defined, which is used to contain a program state: registers, variables, an address of the instruction currently been executed, etc. An interrupt will be handled as follows: prior to an interrupt, a program state is stored in `jbuf`. Using `signal()`, a system function call installs an interrupt handler subject to a specific interrupt. Once the interrupt occurs, the interrupt handler restores the previously saved `jbuf`, and rolls back to the program state saved in `jbuf`.

Lines 4 to 9 of code fragment 4.23 defined the interrupt handler. Once the handler is called after an interrupt is trapped by the operating system, it jumps

into the program state that jbuf represents.

```

1      ...
2      /* continue from line 43 in code fragment 20 */
3      signal(SIGALRM, catcher);    /* install interrupt handler */
4      alarm(60);                  /* set clock to run 60 seconds */
5      if ( (intrno = setjmp(jbuf)) == 0 ) { /* normal return */
6          if (result != NULL)
7              (void) svc_sendreply(transp,
8                  xdr_result, result);
9          alarm(0);                /* ok. reset the alarm clock */
10         ...
11     } else                      /* signal of alarm */
12         fprintf(stderr, "\nLoss client, sig=%d", intrno); 13
14     return;
15

```

Code fragment 4.24 handling reply to local side

Line 3 in code fragment 4.24 installs the interrupt handler subject to the signal SIGALRM. Line 4 sets an alarm clock of 60 seconds. Line 5 calls *setjmp()* to store the program state as jbuf. Lines 6 to 8 employ *svc_sendreply()* to query the client. If the call returns within 60 seconds, the alarm is turned off in line 9. Otherwise, after approximately 60 seconds, a signal will be sent to the program to unblock the send operation. The signal also causes the interrupt handler to be called. When the handler is called, program control is send back to line 5 with a non-zero integer. That value causes the *if* statement to execute the *else* clause. In this case, it is assumed that no process is running on the client side to receive a reply. Finally, the *dispath* function returns with no value.

4.6.7.8 Other issues

In the above sections some important or interesting issues were discussed. However, there are many other issues in the program that were not mentioned. In this section, two such issues are discussed.

When a random number generator, `drand_random(&handle, 0)`, is called the first time, it returns the same random number. Of course, the succeeding numbers are all different. This is a result of using a default initial seed. To avoid this embarrassment, `drand_creat(&handle)` and `drand_new(&handle)` advance the generator in `handle` before they return. So all generators produce different first random values.

Each generator consumes some dynamically allocated storage to hold a position in a binary tree. These spaces can be obtained either from application programs or from the server. Naturally, these spaces might be allocated in the server since they are not user manipulated values. However, consider that there may be a tremendous amount of memory allocated to all applications, some of them use thousands of generators. These spaces are actually allocated from the application clients. So the only dynamically allocated storage that the server uses is used to maintain the mapping table, tree positions to page numbers.

4.7 The abstract data types

Nearly all programs in this project are written using the abstract data types (ADT), `Fixed` and `Real`. The reason to implement such abstract data types is to handle very long integer arithmetic operations.

It is not very difficult to simulate the behavior of very *long* integer addition, subtraction, multiplication by using built-in data types in high level programming languages, like the SUN/C programming language, which allows 32-bit arithmetic operations. When operands of such operations can not be loaded into the language

provided data objects, the operations can be carried out in cascade-fashion. Appropriate fragments of digits can always be selected from target *long* operands so that the desired operation can be carried out by a sequence of additions and multiplications within the language allowed integer size. Then the only problem that has to be taken care of is the carry after additions. This solution, however, is not very easy to be used on *long* integer divisions. The reason is that the quotient can not be determined when the dividend and divisor can not be completely loaded into the language provided data objects.

This work provides a method to divide *long* integers and the method was implemented as parts of the abstract data types. The usual way to carry out *long* integer operations using the architecture-size arithmetic operations is to evaluate the operation from the right end. For instance, several digits are taken from the right end of each operand each time. However, a quotient of a division is determined from the left end, the highest significant digit. In the method, the quotient is determined by a series of divisions that are allowed by the target machine.

4.7.1 Representing operands

Each of these ADT is a series of grouped decimal digits associated with number of operators and functions. The interested architectures are 32-bit and 64-bit machines. Not lost any generalities, only the implementation for a 32-bit architecture is introduced. The machine register size is usually represented by an integer type available in high level programming languages.

To accommodate very long numbers, series decimal digits are grouped and

stored. The size of the group used to store digits depends on the architecture being used. A C-programming language provided integer data type is relied upon in implementing these abstract data types. The C compiler in SonOS provides both 4-byte `int` and `long` data types. Avoiding involves some confusion with 8-byte *long* integer, the implementation chooses 4-byte `int` as the base data objects and implementing the abstract operation on the objects.

Fixed and Real data types are achieved by simulating *arithmetic logic unit*. From Fig. 4.2, it can be seen that the inner shell of the hierarchy consists of five registers: `reg_A`, `reg_B`, `reg_C`, `reg_R` and `reg_Z`. They are 4-byte unsigned `int`. Here `reg_A` and `reg_B` are used to hold two operands for binary arithmetic operations. `reg_C` is used to hold carry for addition operation. `reg_Z` is used to represent product, quotient for a multiplication, division or sum of an addition. `reg_R` is used to hold remainder for an integer division. The arithmetic operations truly carried out in the simulation are

$$\begin{cases} \text{reg\_Z, reg\_C} & \leq \text{reg\_A} + \text{reg\_B} \\ \text{reg\_Z} & \leq \text{reg\_A} * \text{reg\_B} \\ \text{reg\_Z, reg\_R} & \leq \text{reg\_A} / \text{reg\_B} \end{cases}$$

Subtraction is carried out by addition in the arithmetic unit.

Numbers stored in these registers are positive and all arithmetic operation have to be guaranteed no overflow. This implies that for a safe multiplication, both operands should be the numbers less than $2^{16} - 1$, or 65535. For debugging purpose, each operand in the simulation is moduled by 10000, which is smaller the maximum possible. A long integer in this implementation is a series of 4-decimal digit units. For example, the number 1234567890123456 is represented as:

1234 5678 9012 3456. The algorithm of addition and multiplication are straightforward. One 4-decimal digits from each operand is taken into `reg_A` and `reg_B` for the desired operation. The operation can be carried out in a 4-byte register. The result is first obtained in `reg_Z`, later it is shifted into 4-decimal units. For long number addition or multiplication, the operations can be cascaded.

4.7.2 The division method

For carrying out a multiplication, it is not needed that both *full* operands to be loaded into the operation register at one time when the desired operation takes place. However, to determine the quotient of a division of two numbers the knowledge of both dividend and divisor have to be employed. The method provided in this section is that when dividend and divisor can not be loaded into an operation register, partial quotient(s) can be *approximated*. And some adjustment may be need to make the precise quotient.

To show how the method works, one example is provided with the diagram 4.8. Suppose we want to evaluate: $5678991234567890/12340021$. We can do series of additions, subtractions and multiplications on what we described hypothesis arithmetic unit and carry out on 4-decimal digit operands.

In step 1 and 2, we draw dividend and divisor in the designated places with groups of 4-decimal digits, see Fig. 4.8. Arrows are used in Fig. 4.8 associating with the step number. In step 3, the first group of 4-decimal from each of these operands are taken into account. A division: $5678/1234$ is made and a quotient is obtained, which is 4. In steps 4 and 5, $4 * 1234 = 4936$ is calculated, and the

difference between the product and dividend is computed, which is 742. Up to this point, the first 4-decimal group from both operands are employed. In step 6, the difference obtained by the previous step is joined together with the second group of 4-decimal digits from dividend to form 742 9912. Then a multiplication is made by the quotient, 4, and the second group of 4-decimal digits from the divisor, 21. The result is 84 in step 7. In step 8, the difference is counted based on the result from steps 6 and 7. Whatever the intermediate result carried out by the earlier operations, it is ensured no an overflow. The operations described from step 3 to step 8 can be repeated. The idea is clear: *make one division on appropriate integer size, then make more multiplications*. Let's go over more steps of this division. In step 9 uses the first group of divisor to divide the result in the previous step, which is 742 9828 to obtain the second group of quotient. The quotient is group of digits less than 4 digits. In this case, it is 6020. In next steps, multiplications are needed. In step 10, the product of the newly obtained quotient and the first group of divisor is calculated, which is 746 8680. The difference obtained by steps 8 and 10 is made and joined with the third group of digits from the dividend to make: 1148 3456 in step 12. In step 13, the product of the newly obtained quotient and the second group of digits from the divisor is computed, the result is: 12 6420. In the step 14, a subtraction is made on the results obtained in steps 12 and 13. These steps can be repeated up to 20, the final result quotient is: 4 6020 9203.

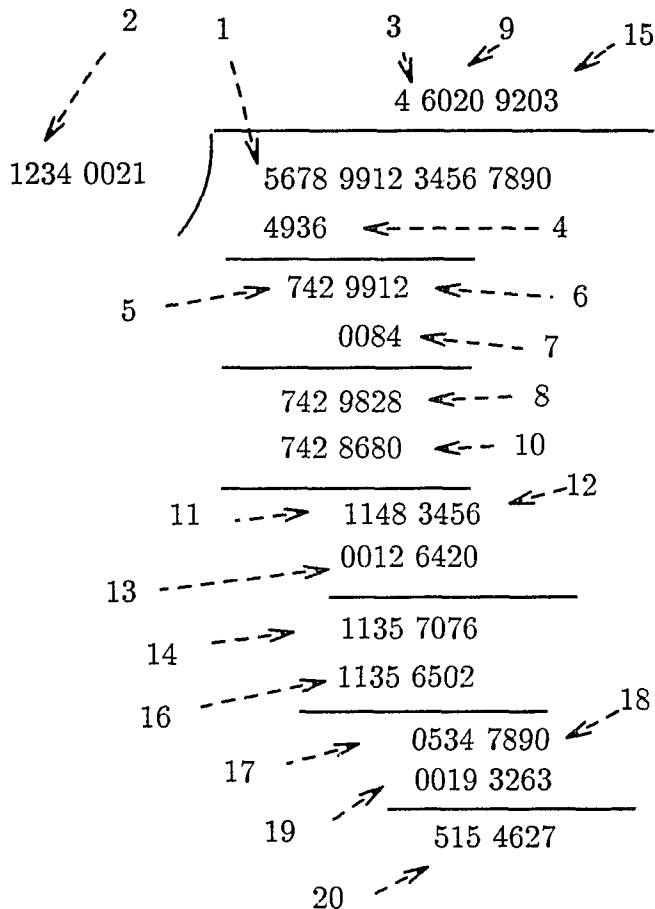
The method can be viewed as several multiplications follow a division which can be made in an available architecture. It is certainly very expensive. However,

if we do not or can not program register level instructions, this high level solution is a way to overcome the problem we meet.

The problem of this method is that if a grouped partial quotient is not determined correctly in the division step, some subtractions are needed. For example, if a subtraction is negative, in step 8 or 14 or 20, the quotient was determined too large. The quotient has to be lowered, and corresponding corrections have to be made.

The approximate division algorithm

Illustration by an example



5678991234567890 div 12340021

quotient: 460209203

remainder: 5154627

Suppose that we want to evaluate:

$$5678991234567890 / 12340021$$

we carry the following steps:

- 1 write down the dividend
- 2 write down the divisor (they are written in a group of 4-digit
- 3 evaluate $5678/1234 (=4)$
- 4 write 4936 ($4 * 1234$)
- 5 write 742 ($5678-4936$)
- 6 move the 2nd dividend group down to make 743 9912
- 7 make the multiplication of $4 * 0021 (= 0084)$
- 8 subtract 84 from 742 9912
- 9 (repeat the steps 3 to 8) evaluate $742\ 9828 / 1234 (=6020)$
- 10 write 742 8680 ($=6020*1234$)
- 11 write 1148 ($742\ 9828 - 742\ 8680$)
- 12 move the 3rd dividend group down to make 1148 3456
- 13 make the multiplication of $6020 * 0021 (=00126420)$
- 14 subtract 12 6420 from 1148 3456 ($=1135\ 7076$)
- 15 (repeat the steps of 5 to 8) evaluate $1135\ 7076 / 1234 (=9203)$
- 16 write 1135 6502 ($=9203 * 1234$)
- 17 write 0534 ($=1135\ 7076-1135\ 6502$)
- 18 move the 4th dividend group down to make 0534 7890
- 19 make the multiplication of $9203 * 0021 (=0019\ 3263)$
- 20 subtract 0019 3263 from 0534 7890 ($=515\ 4627$)

Figure 4.8 very long integer division

5

The Distributed Monte Carlo – One-dimensional case

In this chapter a solution to the problem of distributing a quantum system of one particle using the Green's function Monte Carlo method is presented. There are many ways to conduct Monte Carlo calculations in parallel; distributing the calculations by employing message-passing systems on computer networks is one of them. The simplest method of distributing a Monte Carlo computation is to start many computations on different computer hosts, using independent random number generators, and taking the average of the final results obtained from all the computer hosts at the end of the computation. This type of distributed computing needs very little interprocess communication, so the overhead of the computation is very low. However, this method does not solve the problem that a large population of sample points which are needed for variance reduction may not be able to be held in a single host. The solution presented here is a *truly* distributed parallel computation – a single sequential solution is broken down and evaluated on a group of computer hosts.

One characteristic of the Green's function Monte Carlo method is its iterative nature. Synchronization is needed among the processes at the end of each iteration. In the implementation of the one particle problem, all message operations used in the computation are synchronized.

In most message passing systems, during message transmission, sending and receiving processes may be either *blocked* or *unblocked*. If processes are blocked

when they employ message options, it is said that the processes are synchronized and the message operations are called synchronous messages. Unblocked messages refer to processes that do not involve transmission synchronization.

In the dp implementation, message types are classified a little differently. Synchronized messages can be *blocked* and *unblocked*; asynchronous messages are *interrupted* messages. Even though no processes are blocked during transmitting *unblocked* and *interrupted* messages, the *unblocked* message operation is still classified as a synchronized operation.

A message operation can also be classified as *reliable* or *unreliable*. In some sense, *blocked* messages imply *reliable* operations. However, *unreliable* operation does not mean that the operation can not be fully trusted. It merely means that some chance exists that the message will not arrive at its destination. The advantage of such an operation is that it has little overhead, at least, when an *unreliable* message arrives at its destination, there is no need to acknowledge to the sending side.

In the program developed for solving the one particle problem, synchronized messages are employed. On the destination side, processes receiving messages are *blocked*. On the sending side, operations are *reliable*, but not blocked. Even though sending process is not blocked, the message transmission is synchronized in the sense that the process on the receiving side will not return until a message arrives. At nearly the same time, a message must be sent to the receiving process.

When two processes need to exchange information, the receiving side will be

informed how many receive operations should be executed before real transmission takes place. The data needed to be transmitted are sampled points. The sending process can pre-compute the number of messages needed to carry these points to a particular process.

Similar synchronized messages are employed that immediately follows data transmission. These messages are used to synchronize iterations among the processes. It would appear that the sending side should apply *blocked* messages, while the receiving side also employs *blocked* message operations when synchronizing two processes. This has no problems if only two processes need to be synchronized. To program synchronization on more than two processes is difficult. Point to point synchronization has to occur correctly. It is not only slow, but a potential dead-lock occurs as well. As mentioned above, leave sending operations *unblocked* can achieve synchronization on a group of processes without deadlock.

Unlike some client/server environments in which message loss is tolerable, or some distributed systems in which retransmits are allowed, our solution requires *deterministic* behavior: the reliable operation, because of the synchronous nature. If any expected event does not occur, the entire computation is paralyzed. This behavior could be altered to tolerate message loss, particularly for Monte Carlo simulation. But it is not discussed in this solution.

The growth energy estimator is computed in such a way in the distributed computation that each process contributes a partial solution obtained from its local population. The solution can automatically balance work load among distributed

processes based on the dynamically adjusted geometric boundaries. The format developed for solving the one particle problem will be employed to solve a quantum system of three fermions, discussed in chapter six.

Because the one-dimensional problem is relatively small in terms of required populations, compared with our model problem of three fermions, some issues mentioned in chapter one will not be discussed here, such as the mechanism which does message flow control at the application level.

5.1 A one dimensional quantum problem

The partial differential equation introduced in Eq(1.1) for solving many-particle systems can be written as follows to solve a 1-dimensional one-particle system:

$$\left(-\frac{d^2}{dx^2} + V(x)\right)\psi(x) = E\psi(x) \quad (5.1)$$

The system studied has the particle travels only in one dimension, say along the x -axis, and with potential $V(x)$:

$$V(x) = \begin{cases} 0 & \text{if } |x| < 1; \\ \infty & \text{if } |x| \geq 1. \end{cases}$$

Then Eq(5.1) can be expressed as:

$$-\frac{d^2\psi(x)}{dx^2}\psi(x) = E\psi(x), \quad \psi(x) = 0 \quad \text{for } |x| \geq 1 \quad (5.2)$$

That is, the probability of finding the particle when $|x| > 1$ is 0 since it is *restricted* in a deep-well with an infinite potential at the boundaries: $(-1, 1)$. The diagram in Fig. 5.1 is provided to show the potential and ground state energy.

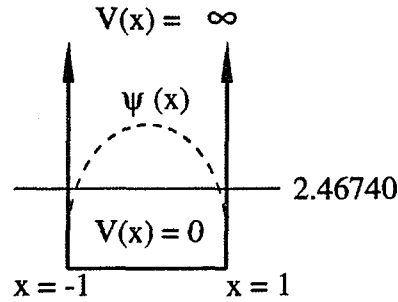


Figure 5.1 Potential of the problem

The analytical expression of the ground state wave function and energy solution are given below:

$$\psi = \cos\left(\frac{x\pi}{2}\right)$$

$$E_0 = \left(\frac{\pi}{2}\right)^2 \simeq 2.46740$$

5.2 GFMC in one dimension: The iteration

For the operator $H = -\frac{d^2}{dx^2}$ of Eq(5.1), the Green's function can be written as:

$$G(x, x') = \begin{cases} \frac{1}{2}(1 - x')(1 + x), & x \leq x' \\ \frac{1}{2}(1 + x')(1 - x), & x \geq x' \end{cases} \quad (5.3)$$

It has been discussed in chapter 2 that Eq(5.1) has a solution, ψ_0 , for the ground state given by the following integral equation:

$$\psi_0(x) = E_0 \int G(x, x') \psi_0(x') dx' \quad (5.4)$$

E_0 is the lowest energy of the system. The way to solve Eq(5.4) is to iterate the following recurrent relation with a trial energy E_T and an initial wave function ψ^0 :

$$\psi^{(n+1)}(x) = E_T \int G(x, x') \psi^{(n)}(x') dx' \rightarrow \psi_0, \quad n \rightarrow \infty \quad (5.5)$$

The above equation defines a Neumann series, that converges to ψ_0 , the ground state, when n becomes large.

For an initial $\psi^{(0)}(x)$, as long as it is not orthogonal to the eigensolution $\psi_0(x)$, $\langle \psi_0(x) | \psi^{(0)}(x) \rangle \neq 0$, the series defined in Eq.(5.5) shall geometrically converge to $\psi_0(x)$. That is $\psi^{(n)}(x) \rightarrow \psi_0(x)$ as $n \rightarrow \infty$. In the method of the Green's function Monte Carlo, $\psi(x)$ is represented as a set of points R . The probability distribution function of the set of points is $\psi(R)$. To approximate $\psi_0(R)$, a set of points R is selected to represent $\psi^{(0)}(R)$. Presumably, $\psi^{(0)}(R)$ is close to $\psi_0(R)$. The probability distribution function of the set of points R can be considered normalized to the distribution $\frac{\psi^{(0)}(R)}{\int \psi^{(0)}(R) dR}$. The procedure of three steps listed as follows uses a random walk to solve Eq(5.5):

1. select a set $\{R'_j\} = A^{(n)}$ of N points from $\frac{\psi^{(n)}(R')}{\int \psi^{(n)}(R') dR'}$, where $n = 0, 1, 2, \dots$
2. for each R'_j in $A^{(n)}$:
 - a) let $M(R'_j)$ be a random integer whose expectation is $\int E_T G(R, R'_j) dR$
 - b) let $B_j = R$ be a set of $M(R'_j)$ points, sampled from $\frac{G(R, R'_j) dR}{\int G(R, R'_j) dR}$
3. the union of all the B_j constitutes the desired set, $A^{(n+1)}$ which represents $\psi^{(n+1)}$.

Here we assumed that the Green's function $G(x, x'_j)$ is integrable. The three

steps can be explained literally as: 1) Select N points from $\psi^{(n)}(R)$ to form the initial set of points; 2) Compute the integral of $\int E_T G(R, R')$. Next, convert the result into an integer by adding a random value between 0 and 1 to form the expected value $M(R'_j)$. Then sample $M(R'_j)$ points from the normalized Green's function $G(R, R'_j)$ with the rejection technique to compose a set, $B_j = R$, where R is sampled from the Green's function; 3) From the union of the N number of sets, B_j , to constitute $A^{(n+1)}$. The $A^{(n+1)}$ is representing the normalized density distribution $\frac{\psi^{(n)}(R)}{\int \psi^{(n)}(R)}$ in the next iteration, which will be used in step 1).

E_T is a trial energy value which substitutes for E_0 in Eq(5.4). E_0 is unknown and to estimate it is the ultimate goal of this simulation. In the GFMC algorithm, if the trial value E_T is selected larger than the actual eigenvalue E_0 , the population of the set generated by each iteration becomes larger and larger. Otherwise, if $E_T < E_0$, the population becomes smaller and smaller. This does not effect the series convergence to the eigensolution, $\psi_0(R)$. From a practical point of view, only N points are selected from the previous population at the beginning of each iteration, so the population will not grow infinitely large even though a trial energy value E_T is selected larger than E_0 .

5.3 GFMC in one dimension: Estimate the lowest eigenvalue

Assume that Eq(5.5) is iterated and that, ψ_0 has converged. The following quotient is used to estimate the energy E_0 . The expected number of points sampled from the density $\frac{\psi_0(R)}{\int \psi_0(R') dR'}$ is $E_T \int G(R, R'_j) dR$. There are a total of N points of R'_j in each iteration. Then let $\langle M(n+1) \rangle$ be the expected number of points sampled in

the n th iteration, we have,

$$\begin{aligned}\langle M \rangle &= N \int \left[E_T \int G(R, R') dR' \right] \left[\frac{\psi_0(R')}{\int \psi_0(R'') dR''} \right] dR' \\ &= N \frac{E_T}{E_0}.\end{aligned}\tag{5.6}$$

The expected number of points sampled in the n th iteration is linearly dependent on the factor E_T/E_0 . So we have

$$E_0 = E_T \frac{N}{\langle M \rangle}.\tag{5.7}$$

The estimator,

$$\langle M \rangle \simeq \sum_{j=1}^N M(R'_j)\tag{5.8}$$

used to estimate $\langle M \rangle$ is biased because $\sum_{j=1}^N M(R'_j)$ is not equal to $\langle M \rangle$. However, the variance of the random variable $\sum_{j=1}^N M(R'_j)$ tends to be smaller as the sampled size N becomes larger.

Computationally, in order to lower the variance of E_0 , Eq(5.6) is repeatedly evaluated during the iteration, and the numerator and denominator of the formula are accumulated independently. Let m be the number of times that Eq(5.6) is iterated. Then the following, called the growth estimator, is used to estimate E_0 :

$$E_0 = E_T \frac{mN}{\sum_1^m \sum_{j=1}^N M(R'_j)}.\tag{5.9}$$

5.4 GFMC in one dimension: Sampling the Green's function

The Green's function given in Eq(5.3) can be viewed as follows. The particle is at the position x , and its *influence*, or the possibility to move to next location, will be deflected into two linear parts along the x-axis to the ends, -1 and 1 . A triangle can be draw to illustrate the physical situation in Fig. 5.2:

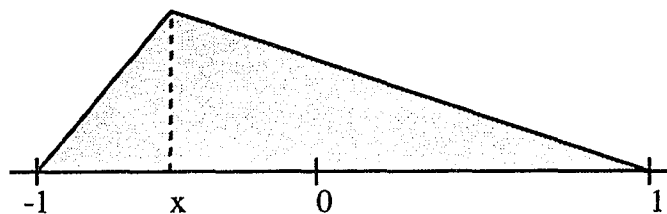


Figure 5.2 Sampling the Green's function at x

Sampling the Green's function is to iterate random walkers based on the triangles with the walkers constructed at x 's by using the Monte Carlo method, which is given in sections 5.2 and 5.3. The number of triangles that have to be sampled is the number of walkers (points) in the population of the current iteration. The walkers are distributed in the entire area from -1 to 1 .

A triangle can be sampled by a series of isosceles triangles starting at x , see Fig. 5.3. One end of the first isosceles triangle reaches the nearest endpoint of the interval, -1 . The second isosceles triangle starts at the other end of the first isosceles triangle. One end of the second isosceles triangle touches the nearest endpoint of the interval, and the third isosceles triangle will start at the other end

of the second isosceles triangle, and so forth. The sequence of constructing isosceles triangles can be determined by sampling a random variable. The first isosceles triangle is the initial step to sample the Green's function. The difference from the actual Green's function it introduced will be corrected by the subsequent isosceles triangles. Sampling by random walks is very useful when a Green's function is not known analytically. In this case, the following diagram shows how a random walk is used to sample a triangle area:

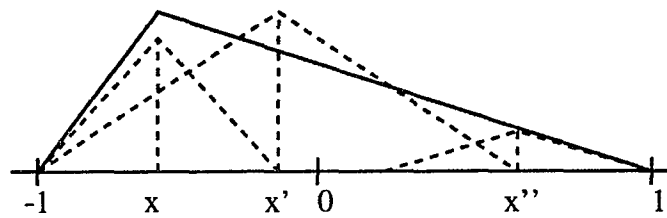


Figure 5.3 Sampling by a random walker at x

In a parallel computation, the Green's function can be sampled by multiple random walkers simultaneously. Because the random walks may be executed in different machines and the eigenvalue will be estimated partially, the reasonable partition for parallelism may be considered along the x -axis. The following diagram shows that the original interval is divided equally and the Green's function is sampled by four points started at different subintervals at the same time.

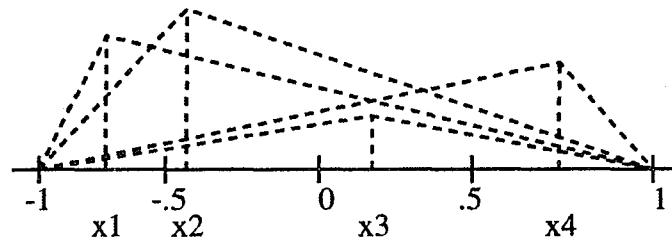


Figure 5.4 Sampling the Green's function in parallel

The interval $[-1, 1]$ can be divided into more subintervals depending upon the number of worker processes in a computation. In general, the more machines involved in a computation, the larger population that can be sampled. Therefore, the limits of memory and CPU power of a single processor are no longer limit factors existing in Monte Carlo computations. Comparing four hosts sampling 10,000 points with the same number of hosts sampling 100,000, the latter case may have better performance than the former due to that there are larger number of points in each local population. The errors introduced by the eigenvalue partially evaluated will be reduced. In the other words, sampling large population needs parallel computation, and parallel simulation requires large number of points located in local population.

5.5 Distributing 1-D GFMC: Overall and initialization

In the previous four sections of this chapter, the basic concepts that are needed to develop the one-dimensional Monte Carlo program were covered. In this section, a program that computes the problem in parallel is discussed. Those functions in the program, that are not involved in process communication and synchronization

are not presented here. They are basically sampling and running in parallel. Other omitted code involves code debugging, error detection and recovering. The way that the Green's function is sampled can be found in [Arnow]. The parts of the program that are discussed in this section are those involved in distributed computing. The job loads assigned to individual distributed processes are determined by subintervals along the x -axis for the integration in Eq(5.4). To achieve the request that nearly equal jobs are carried by all processes, the subintervals are dynamically adjusted. Calculating subintervals, synchronizing processes and how to construct a modified growth energy estimator to deal with the distributed population are discussed.

The procedure below, written in pseudo-code sketches the structure of the computation.

```

1      program()
2      begin
3          starts from the primary, then to many secondaries;
4          initialize dp environment;
5          gfmc:
6              initialize mc:
7                  initialize user program level variables;
8                  iterate(to obtain wave function):
9                      loop NINIT times to converge wave function, psi;
10                     iterate(to estimate energy):
11                         loop NGENS times with an energy estimator;
12             terminate:
13                 stop dp processes;
14         end
15
16         iterate(converges psi or estimates E0)
17         begin
18             loop begin
19                 sample:
20                     sample Green's function;
21                 divide:
22                     group sampled points by dp-process's
23                     interval;
24                 communications:
25                     synchronize iteration with all other
26                     processes (barrier);
27                     sendmsg/recvmsg messages
28                     between processes;
29             toprimary:

```

```

30         rebalance work load;
31         if estimate needed, energy();
32     loop end
33 end
34

```

Code fragment 5.1 sketched main program

The above fragment is self-contained, so what is needed is to expand the sketched pseudocode using the lines in the program, which is written in C. Like all dp programs, when the program starts, it calls several dp library functions, such as `DPinit()`, `DPgetpid()`. Once these calls return, the distributed computing environment is established, and dp processes are running. Those dp library function calls are the equivalent of lines 3 to 4 in the sketched program. What is needed at this point in the Monte Carlo calculation is to initialize subintervals in which the dp processes sample the Green's function.

```

1
2     double interval;                /* hold the init. interval*/
3     double r;                      /* hold the boundaries    */
4     extern double bounds[MAXPROC][2]; /* array of boundaries  */
5     int i;                          /* loop variable counter */
6     interval = 2.0 / DPnnodes;
7     r = -1.0;
8     bounds[0][0] = r;
9     for ( i = 0; i < DPnnodes-1; i++ ) {
10         r += interval;
11         bounds[i][1] = bounds[i+1][0] = r;
12     }
13     bounds[DPnnodes-1][1] = 1.0;
14

```

Code fragment 5.2 initializing intervals

`r` and `interval` are two auxiliary variables helping to obtain the boundaries for all processes. `MAXPROC` is a constant, denoting the number of dp processes for the computation, given during compile time. `bounds` is used to hold the two ends of the intervals of all `MAXPROC` number of processes. `DPnnodes` is obtained by calling dp library function `DPinit()`, its value usually equals `MAXPROC` unless the desired

number of dp processes could not be obtained. The constant 2 in line 6 is the entire interval for the computation counted from -1 to 1 on the x-axis. At first, r in line 7 is initialized to the left end of the problem, -1 , then in lines 10 to 11 the right end of a left subinterval is initialized as the same value as the left end of the right adjacent subinterval, so adjacent subintervals are contiguous. The right end of the right most subinterval is initialized to 1 in line 13, each interval is given the same length, `interval`, computed in line 6.

This initialization, along with some other initializations, such as initializing the distributed random number generators, is carried out by the primary process. After these initializations, a set of points, an initial population, is obtained to start the entire simulation.

```

1
2     double    r;                               /* used locally          */
3     extern    dhandle myrand;                   /* for using dist. rand    */
4     extern    double RNEW[NGNSIZ];              /* sampled population      */
5     extern    int  hostsz[MAXPROC];             /* number of points to dp's */
6     int       i, j;                             /* used as loop var. counts*/
7
8     for ( i = 0; i < NGNSIZ; i++ ) {
9         r = (i % 2 == 0)?Drand(&myrand,0):(Drand(&myrand,0)-1);
10        RNEW[i] = r;
11        for ( j = 1; j <= DPnnodes; j++ )
12            if ( r < (-1.0 + j * interval) ) {
13                hostsz[j-1] += 1;
14                break;
15            }
16    }
17
```

Code fragment 5.3 initializing the population

In this program the distributed random number system discussed in chapter 4 is employed, but not listed here. In code fragment 5.3, it is assumed that `myrand` is properly initialized. `RNEW`, declared globally, is used to hold `NGNSIZ` number of points. `NGNSIZ`, the number of points to be sampled, defined at compilation time,

is supposed to be a large integer.

It was mentioned in the beginning of this chapter that all receiving message operations are *blocking*. Prior to actually receiving data messages, each process is informed of the number of data messages that should be received. In the next computational stage, sending processes calculate the number of messages needed to be sent over to other processes.

`hostsz` is an array of `MAXPROC` integers. Its use is to count the number of points thrown into the subintervals. In line 8 of code fragment 5.3, the loop runs `NGNSIZ` number of times, each time a random value is picked up. In line 9 of the same code fragment, the picked random value is mapped into the interval, half of the region is in the negative x-axis. From lines 11 to 15, an appropriate subinterval is searched with respect to the random value. The subintervals will be assigned to distributed processes, each for per process. Therefore, the process owning the interval will sample the part of the Green's function based on the points in the subinterval. `hostsz` is used to count the number of points in each subinterval.

5.6 Distributed 1-D GFMC: Messages

After these variables are prepared, it is the time to discuss message structures used to delivery data between processes. Two type of messages are used in the program: one, `MCPCK`, for data transmission and the other, `SYNC`, for the purpose of synchronizing processes.

```

1
2      typedef      struct mcpck {
3          int      iteno;      /* carries iteration number      */
4          int      pckno;      /* carries packet number      */
5          int      totpts;      /* num. of pts. in the packet */

```

```

6          int      sampts;      /* carries number of sampled ptr*/
7          double   ddata[NUMPOINTS]; /* data array */
8      } MCPCK;
9
10     typedef      struct {
11         double matrix[MAXPROC][2] ; /* carries boundaries */
12     } table;
13
14     #define      msgleng  DATASIZE - sizeof(table) - 4*sizeof(int)
15
16     typedef      struct syncpck {
17         int      flg1, flg2;      /* synchronization packet flags*/
18         int      opt1, opt2;      /* operational options */
19         table     itv;            /* table of bounds */
20         char      msg[msgleng]; /* deliver a str. for debugging*/
21     } SYNC;
22

```

Code fragment 5.4 Definitions of MC packets

DATASIZE and NUMPOINTS are constants defined at compile time. NUMPOINTS is defined based on dp message size, DATASIZE. For a dp system's packet that can carry 1000 bytes (DATASIZE) of user data, NUMPOINTS is set to 120, that means 120 double floating point can be accommodated in the packet.

The first packet sent by the primary to all secondaries is the "greeting message". Its use is to inform the secondaries their intervals, so each secondary knows not only its boundaries, but the boundaries of all other processes as well. In the entire computation, the boundary allocation is carried out by the primary process. All the secondary processes are passive in that they do not need any knowledge of work load allocations. The primary can carry out any scheme to distribute the work load, either dynamic work allocation or pre-programmed fixed work allocation.

Dp processes' IDentification numbers are required to be integers so the intervals can be named by the integers, equivalent to the corresponding dp ID's. The left most subinterval is named 0, so its corresponding process is dp process 0. By this

agreement, each dp process knows the corresponding processes for all subintervals, so that dialog can be established and data messages can be delivered.

```

1
2  extern int outtab[MAXPROC];    /* the num. of pcks a dp to rcv. */
3  extern int samplepts[MAXPROC]; /* keep a record of bounds      */
4  #define TYPEcmsg    (-3)      /* communication message type  */
5  int    i, j;
6  SYNC   pck;
7
8  for ( i = 0; i < DPnnodes; i++ ) {
9      samplepts[i] = hostsz[i];
10     outtab[i] = ( hostsz[i] / NUMPOINTS ) + (
11         ( hostsz[i] % NUMPOINTS > 0 ) ? 1 : 0 );
12     pck.flg1 = -1; /* flg1 will carry the ite. num. */
13     pck.opt1 = outtab[i]; ;
14     for ( j = 0; j < DPnnodes; j++ ) {
15         pck.itv.matrix[j][0] = bounds[j][0];
16         pck.itv.matrix[j][1] = bounds[j][1];
17     }
18     sprintf(pck.msg, "Welcome to dp");
19     (void) DPsend((char *)&pck, sizeof(SYNC),
20         DPREL, i, TYPEcmsg);
21 }
22

```

Code fragment 5.5 The greeting message sent by the primary

The code fragment 5.5 is executed by the primary process. `outtab`, in line 2, is used to hold the number of messages that will be sent to the secondaries. These numbers have to be sent and received by the secondaries before actual data transmission takes place so that the receiving side knows the number of times that `DPrecv()` operations should be issued. `samplepts`, in line 3, is an array used to hold the number of points sent to secondaries. These numbers will be used when estimating energy and balancing work load are required. The *std dp standard dp* provides a “select” feature. The receiving side can select a particular type of message when it arrives. The “select” messages are targeted by the so-called `TYPExmsg`, where `x` may be `c`, *communication* message, `b`, *data* message, or some other letter. On the sending sides, `DPsends` are programmed in unblocking and reliable mode. So a sending

side may run faster than a receiving side. After sending data messages, the sending processes immediately start sending synchronous messages. On the destination side, receiving processes use the “select” feature to distinguish incoming messages and pick up the expected ones by using `DPrevc()`. In this communication/synchronous stage, both, sending and receiving sides use `TYPEcmsg` type messages.

`TYPEcmsg` in the `dp` call informs that the incoming message is a “synchronization” message. It is used to synchronize events in the user application level and has no significance to the lower `dp` level service with message transmission. The importance is that the message types on both sides have to match. Line 6 declares a variable of `SYNC` type. In line 9, the number of points sampled in each subinterval is saved in `samplepts`. In line 10, the precise number of messages is computed based upon the number of points sampled in the subinterval, `hostsz`. Lines 12 to 17 initialize the greeting message. Finally, line 18 calls `DPsend()` to deliver the message. All message are sent synchronously, and reliably, as denoted by `DPREL`. The fourth parameter of the call is used to name the destination side, and last parameter specifies the type of the message.

In the meantime, all the secondary processes are expecting the greeting message. The following code fragment is executed by all the secondaries to receive the message.

```

1  DPresult      *rtrn;           /* receive a message      */
2  SYNC          pck;             /* received message stored */
3  int           j;
4  extern int     intab[MAXPROC]; /* the num.of data msgs expected */
3  rtrn = DPrevc((char *) &pck, sizeof(SYNC),
4           DPBLOCK, 0, TYPEcmsg, NOSRC);
5  for ( j = 0; j < DPnnodes; j++ ) {
6           bounds[j][0] = pck.itv.matrix[j][0];
7           bounds[j][1] = pck.itv.matrix[j][1];

```

```

8     }
9     intab[0] = pck.opt1;
10
11    pck.flg1 = -1;
12    pck.opt1 = 0;
13    sprintf(pck.msg, "Ready receiving the initial");
14    (void) DPsend((char *) &pck, sizeof(SYNC), DPRECV, 0, TYPEcmsg);
15

```

Code fragment 5.6 The secondaries handshake with the primary

`DPreresult` is defined in the `stdp.h`, the standard dp header file. `DPrecv()` returns a pointer pointing to `DPreresult`, which is a record of three fields (see chapter 2). The incoming message is expected from the primary, so the fourth parameter of the `DPrecv()` in line 3 is specified as 0. Immediately after receiving the message, some code is needed to check the message `flg`'s and `opt`'s; they are all omitted here. The useful information is kept in `bound` and `intab` in lines 5 to 9. `intab` is an array of `DPnnodes` integers and stores the number of messages to be received from all other processes in the next step. In this case, the only source of messages is coming from host 0 (`dppid 0`). In lines 11 to 14, a reply message is send back to the primary to indicate the process is ready to continue.

After sending the greeting messages, the primary, then, executes a loop to receive all the reply message from the secondaries. Following this synchronizing stage, data transmission takes place.

A function called `sendover(pid, ite)` is written to send an array of an arbitrary number of sampled points stored in `RNEW` (see code fragment 5.3). The primary calls `sendover(i, -1)` for sending points to the *i*th process and the -1th iteration. The following is part of the function:

```

1
2    int      pno = 0;          /* count packet number      */

```

```

3   int      nnew;                /* the num.  of points to be sent */
4   int      srtptr = 0;          /* an index to RNEW          */
5   extern  int  hostsz[MAXPROC];
6   MCPCK    mcpck;
7   #define TYPEbmsg    (-2)      /* date message type          */
8
9   nnew = hostsz[pid];
10  while ( nnew > 0 ) {
11      if ( nnew >= NUMPOINTS ) {
12          mcpck.totpts = NUMPOINTS;
13          filddata(&srtptr, pid, NUMPOINTS, &mcpck);
14      } else {
15          mcpck.totpts = nnew;
16          filddata(&srtptr, pid, nnew, &mcpck);
17      }
18
19      nnew -= ps;
20      pno += 1;
21      mcpck.iteno = ite;
22      mcpck.pckno = pno;
23      mcpck.sampts = hostsz[pid];
24
25      (void) DPsend((char *) &mcpck, sizeof(MCPCK),
26                  DPRECV, pid, TYPEbmsg);
27  }
28

```

Code fragment 5.7 Send data points over

Code fragment 5.7 sends all points that should be sent to pid. pid is used here primarily to identify the subinterval which is assigned to the dp process, pid. In line 9, nnew is initialized to hold the total number of points belonging to the subinterval that is indexed by pid. Then the loop from lines 10 to 27 counts NUMPOINTS number of points from nnew, and sends the NUMPOINTS number of points to process pid. Lines 11 to 17 take care of two different cases: send exactly NUMPOINTS number of points or the rest of the points if nnew is less than NUMPOINTS. filddata() is a function that searches for points that should be send to pid, and packs them in mcpck. srtptr is a global index pointer to filddata(). Each time filddata is called, it continues to scan RNEW starting from srtptr. filddata() will pick the number of points, which is given through the third parameter, which is either

NUMPOINTS or `nnew`, and store them in the fourth parameter of the function, which is a data packet. Lines 19 to 20 keep track of the number of points and the number of packets sent. Line 21 to 23 fill up the rest of the fields of the packet. Line 25 sends the packet to process `pid` with the `TYPEbmsg` message type.

The lines that follow are used in receiving processes.

```

1
2     extern int    intab[MAXPROC]; /* the num.  of expected pck */
3     extern DPresult rtrn;
4     MCPCK  mcpck;
5     while ( intab[0] > 0 ) {
6         intab[0] -= 1;
7         rtrn = DPrevc((char *) &mcpck, sizeof(MCPCK),
8                     DPBLOCK, 0, TYPEbmsg, NOSRC);
9         appendtoRNEW(&mcpck, ite);
10    }
11

```

Code fragment 5.8 Receive data points from host 0

Code fragment 5.8 receives `intab[0]` number of messages from host 0. `appendtoRNEW` is a function that literally copies the points in `mcpck.ddata` into its local `RNEW` which is used to store the initial population. Notice that when the messages are received, the original message order is not preserved. So the mechanism used here is not a general solution to other situations in which the message sequencing is needs to be known.

Code fragments 5.2 to 5.8 are the actual lines to expand `initialize use program level variables` in line 6 of code fragment 5.1. The next stage of the computation is iterating the Nuemman recurrent relation given in Eq(5.5). The specified tasks are sketched in code fragment 5.1, `iterate()`. A function that samples the Green's function is called first. The newly sampled `NRNEW` number of points again are stored in `RNEW`. This stage of the computation runs completely in

parallel, each dp process executes its instructions sequentially and independently. The details can be found in [Arnow].

Following the independent computations are four tasks: for each process, divide the points in `RNEW` into `DPnnodes` groups based on the subintervals assigned by the primary at the initial time; synchronize the computation; send the groups of points to the appropriate dp processes; and receive points sent to the current process by other dp processes. The first task can be achieved by the following code, which is used to expand lines 21 to 23 in code fragment 5.1:

```

1
2     int      i, j;
3     double   r;
4     extern int   hostsz[MAXPROC], bounds[MAXPROC][2];
5     extern int   outtab[MAXPROC];
6     extern double RNEW[NGNSIZ];
7     extern int   NRNEW;
8     for ( i = 0; i < DPnnodes; i++ )
9         hostsz[i] = 0;
10
11     for ( i = 0; i < NRNEW; i++ ) {
12         r = RNEW[i];
13         for ( j = 0; j < DPnnodes; j++ )
14             if ( r >= bounds[j][0] && r < bounds[j][1] ) {
15                 hostsz[j] += 1;
16                 break;
17             }
18     }
19
20     for ( i = 0; i < DPnnodes; i++ )
21         outtab[i] = ( hostsz[i] / NUMPOINTS ) + (
22             ( hostsz[i] % NUMPOINTS > 0 ) ? 1 : 0 ) ;
23

```

Code fragment 5.9 Divide `RNEW` according to the subintervals

Many lines in code fragment 5.9 are familiar, so no line by line explanation is needed. Lines 8 to 9 set `hostsiz` back to 0's. Lines 11 to 18 preprocess all points in `RNEW`, and `hostsiz` is reinitialized in line 15. In line 21, `outtab` is initialized to the number of messages supposed to be sent to all dp processes.

Unlike the initial stage, in which the primary sends messages and the secondaries receive them, in this stage of the iteration, all processes send and receive messages. The following code is used to synchronize all dp processes before executing message transmission operations.

```

1
2     int     i;
3     SYNC    pck;
4     extern  int  DPpid, outtab[MAXPROC];
5     for ( i = 0; i < DPnnodes; i++ )
6         if ( DPpid != i ) {
7             pck.flg1 = ite;
8             pck.opt1 = outtab[i];
9             (void) DPsend((char *) &pck, sizeof(SYNC),
10                DPREL, i, TYPEcmsg);
11         }
12
13     for ( i = 0; i < DPnnodes; i++ )
14         if (DPpid != i) {
15             rtn = DPrevc((char *) &pck, sizeof(SYNC),
16                DPBLOCK, i, TYPEcmsg, NOSRC);
17             intab[i] = pck.opt1;
18         }
19

```

Code fragment 5.10 Synchronizes all dp processes

No matter how fast or slow a process executes the sampling part of the program, it will be synchronized in this stage. A process can not complete this code until it receives a synchronization packet from every other process, and the synchronization packet also carries the number of packets that should be received from the sending side of the packet in the next stage. Lines 5 to 11 loop for all other processes, and send a packet to each process telling that how many packets that will be sent by this process. Lines 13 to 18 run another loop and receive a packet from each other processes. The `DPrevc()` call in line 15 is made to select a packet from the particular source `i`. The number of packets that should be received from that source is stored in `intab` in the `i`th subscript. The next two tasks are sending and receiving data

messages, the code for these operations is similar to the code fragments 5.5 and 5.6. Each process makes a new population, in `RNEW`, by receiving points that all belong to the subinterval it is working on from other dp processes plus those points locally sampled that belong to its own subinterval.

These four tasks can be repeatedly executed and the wave function will converge. However, because of the shape of the wave function, (see fig. 5.1) the processes that sample the middle subintervals of the Green's function have more work to do than the processes doing their work on the end subintervals if all subintervals are kept the same. Some processes end up having nearly no work to do. In the next section, balancing the work load is discussed.

5.7 Distributed 1-D GFMC: Estimating energy and balancing work load

When the above four tasks are repeatedly executed, the population will grow larger and larger if the trial energy E_T is set larger than the eigenvalue E_0 . In a sequential execution, `NGNSIZ` points are selected from the current population before the next iteration starts. So no matter what value is given to E_T , a population of `NGNSIZ` points is always prepared, locally, for the next sampling period. It should be understood that the new population has to be chosen at random, wherever these points are stored. We can select a new set of points from the distributed population using the same idea. The decision will be how many points should be chosen from local memory.

Like the sequential computation, the new population should be selected at random from the current population so that the distribution of the generated set

is preserved in the newly selected population. At this stage of the computation, all sampled points from the current iteration are classified into subintervals. The only information that a process needs is the number of points, proportional to the entire population, that have to be chosen from the local memory to initialize the next iteration. In the distributed computation, a message can be sent to the primary reporting the number of points, $NRNEW$, stored locally. The primary then sends a message back telling the proportional ratio, from it a number of points can be computed and a selection can be made from the local population.

When the primary received the messages from all processes, except itself, the following formulas are used to determine the number of points, $ngns_i$, that should be selected by each local population:

$$\begin{cases} NNEW = \sum_{i=0}^{Dpnnodes-1} NRNEW_i \\ ngns_i = (NRNEW_i * NGNSIZ) / NNEW. \end{cases} \quad (5.10)$$

Balancing the work load is based on counting the number of random walks at the beginning of an iteration rather than the number of points sampled by dp processes at the end of an iteration. It is expected that for each dp process, the average number of random walks are loaded. A random walk can contribute one or more points during sampling, or contribute no point at all. Sampling a function by random walks is discussed in section 5.4. The number of random walks for each process are indicated by the number of points stored in $RNEW$. So we want to, dynamically, regulate the number of points in local $RNEW$ toward the average of the entire population.

The way to control the average number of walks for each process is to adjust subintervals so that the subintervals generating large number of points will be reduced in size in the next iteration. When a subinterval shrinks, the number of points located in the interval will be smaller, so the number of random walks will be less, and more toward the average number of random walks. The question is, how should subintervals be reduced.

The useful information that we can use are the subintervals and the number of points generated by the subintervals. We can use these pieces of information to *predict* the correct size subintervals for the next iteration. The new subintervals are obtained by three steps. Notice that the primary has already received the messages from all the secondaries and the number of points, `NRNEW_i`, stored locally have been reported to the primary. The significant of `NRNEW_i` is that in the entire population exactly `NRNEW_i` points are located in the *i*th subinterval. In the first step of the computation, the primary averages the subtotals based on *ngns_i* obtained in Eq(5.10):

$$\begin{cases} ng_{nct} = \sum_i ng_{ns_i} \\ avg \leftarrow ng_{nct} / DP_{nodes} \end{cases} \quad (step1)$$

Next we must obtain an adjustment for each subinterval:

$$\forall i, \quad \Delta leng_i \leftarrow (avg - ng_{ns_i}) / ng_{nct} * 2.0 \quad (step2)$$

The last step is to adjust the subintervals:

$$\forall i, \quad new\_interval_i \leftarrow old\_interval_i + \Delta leng_i \quad (step3)$$

$\Delta leng_i$ may be positive or negative, so the $new\_interval_i$ may be larger or smaller than $old\_interval_i$. But there are some cases in which $new\_interval_i$ for some i becomes negative after the adjustment, which is not allowed. This happens when many dp processes are involved in a computation and $ngns_i$, for some i , is much larger than avg . What we can do is to exponentially average $\Delta leng_i$, and repeatedly execute the following before step 3:

$$\forall i \quad \Delta leng_i \leftarrow \Delta leng_i / 2.0$$

until $new\_interval_i > 0$ for all i .

When the new subintervals are computed, the primary sends them out by messages to each secondary, in the same way that the primary initializes the secondaries at the beginning of the computation.

These formulas automatically adjust the subintervals. Two snapshots were taken from a run on four dp processes with a total of 10,000 sample points. The first one, on the next page, shows the situation in which no balancing of the work load takes place. The following page shows the situation with balancing of the work load. Each snapshot obtains the initial stage, and the next five iterations. `leng[i]` shows $\Delta leng_i$. In the first snapshot, all `leng[i]` are 0's meaning, that there is no interval adjustment required. In the second snapshot, the numbers are changing, some of them negative and others positive. The four rows of numbers in a group,

leng[0] to leng[3], stand for dp process 0 to process 3, respectively. The Lower BD, Upper BD and Interval columns in the first table remain the same during the iterations. While, the numbers in these columns in the second table are different reflecting the situation that the intervals are adjusted. The last column shows the number of random walks in each interval prior to the sampling. In the first table, work loads are not balanced. But in the second table, the work loads are changing toward the average, which is 2,500 random walks.

Snapshot of 4 processes without balancing work load

	1	DBG	Del length	Lower BD	Upper BD	Interval	WorkLoad
	2	DBGinit:leng[0]=	0.0000	-1.0000	-0.5000	0.5000	25039
	3	DBGinit:leng[1]=	0.0000	-0.5000	0.0000	0.5000	24961
	4	DBGinit:leng[2]=	0.0000	0.0000	0.5000	0.5000	24952
	5	DBGinit:leng[3]=	0.0000	0.5000	1.0000	0.5000	25048
	6	DBG	Del length	Lower BD	Upper BD	Interval	WorkLoad
	7	DBG 0: leng[0]=	0.0000	-1.0000	-0.5000	0.5000	14848
	8	DBG 0: leng[1]=	0.0000	-0.5000	0.0000	0.5000	35127
	9	DBG 0: leng[2]=	0.0000	0.0000	0.5000	0.5000	35247
	10	DBG 0: leng[3]=	0.0000	0.5000	1.0000	0.5000	14779
	11	DBG	Del length	Lower BD	Upper BD	Interval	WorkLoad
	12	DBG 1: leng[0]=	0.0000	-1.0000	-0.5000	0.5000	14987
	13	DBG 1: leng[1]=	0.0000	-0.5000	0.0000	0.5000	35035
	14	DBG 1: leng[2]=	0.0000	0.0000	0.5000	0.5000	35138
	15	DBG 1: leng[3]=	0.0000	0.5000	1.0000	0.5000	14840
	16	DBG	Del length	Lower BD	Upper BD	Interval	WorkLoad
	17	DBG 2: leng[0]=	0.0000	-1.0000	-0.5000	0.5000	14972
	18	DBG 2: leng[1]=	0.0000	-0.5000	0.0000	0.5000	35125
	19	DBG 2: leng[2]=	0.0000	0.0000	0.5000	0.5000	34991
	20	DBG 2: leng[3]=	0.0000	0.5000	1.0000	0.5000	14911
	21	DBG	Del length	Lower BD	Upper BD	Interval	WorkLoad
	22	DBG 3: leng[0]=	0.0000	-1.0000	-0.5000	0.5000	14884
	23	DBG 3: leng[1]=	0.0000	-0.5000	0.0000	0.5000	34963
	24	DBG 3: leng[2]=	0.0000	0.0000	0.5000	0.5000	35312
	25	DBG 3: leng[3]=	0.0000	0.5000	1.0000	0.5000	14841
	26	DBG	Del length	Lower BD	Upper BD	Interval	WorkLoad
	27	DBG 4: leng[0]=	0.0000	-1.0000	-0.5000	0.5000	14916
	28	DBG 4: leng[1]=	0.0000	-0.5000	0.0000	0.5000	35023
	29	DBG 4: leng[2]=	0.0000	0.0000	0.5000	0.5000	35165
	30	DBG 4: leng[3]=	0.0000	0.5000	1.0000	0.5000	14895

Table 5.1 Work load unbalanced

Snapshot of 4 processes with balancing work load

1	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
2	DBGinit:leng[0]=	0.0000	-1.0000	-0.5000	0.5000		25194
3	DBGinit:leng[1]=	0.0000	-0.5000	0.0000	0.5000		24806
4	DBGinit:leng[2]=	0.0000	0.0000	0.5000	0.5000		24903
5	DBGinit:leng[3]=	0.0000	0.5000	1.0000	0.5000		25097
6	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
7	DBG 0:	leng[0]=	0.2016	-1.0000	-0.2984	0.7016	14920
8	DBG 0:	leng[1]=	-0.1981	-0.2984	0.0034	0.3019	34906
9	DBG 0:	leng[2]=	-0.2069	0.0034	0.2965	0.2931	35344
10	DBG 0:	leng[3]=	0.2034	0.2965	1.0000	0.7035	14829
11	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
12	DBG 1:	leng[0]=	-0.0562	-1.0000	-0.3547	0.6453	27812
13	DBG 1:	leng[1]=	0.0465	-0.3547	-0.0063	0.3484	22674
14	DBG 1:	leng[2]=	0.0623	-0.0063	0.3491	0.3554	21887
15	DBG 1:	leng[3]=	-0.0526	0.3491	1.0000	0.6509	27628
16	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
17	DBG 2:	leng[0]=	0.0314	-1.0000	-0.3232	0.6768	23428
18	DBG 2:	leng[1]=	-0.0201	-0.3232	0.0051	0.3283	26003
19	DBG 2:	leng[2]=	-0.0341	0.0051	0.3264	0.3213	26704
20	DBG 2:	leng[3]=	0.0227	0.3264	1.0000	0.6736	23865
21	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
22	DBG 3:	leng[0]=	-0.0170	-1.0000	-0.3402	0.6598	25848
23	DBG 3:	leng[1]=	0.0028	-0.3402	-0.0091	0.3311	24860
24	DBG 3:	leng[2]=	0.0258	-0.0091	0.3380	0.3471	23710
25	DBG 3:	leng[3]=	-0.0116	0.3380	1.0000	0.6620	25581
26	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
27	DBG 4:	leng[0]=	0.0113	-1.0000	-0.3289	0.6711	24434
28	DBG 4:	leng[1]=	0.0088	-0.3289	0.0110	0.3399	24559
29	DBG 4:	leng[2]=	-0.0204	0.0110	0.3377	0.3266	26022
30	DBG 4:	leng[3]=	0.0003	0.3377	1.0000	0.6623	24985
31	DBG		Del length	Lower BD	Upper BD	Interval	WorkLoad
32	DBG 5:	leng[0]=	-0.0073	-1.0000	-0.3362	0.6638	25366
33	DBG 5:	leng[1]=	-0.0117	-0.3362	-0.0080	0.3282	25584
34	DBG 5:	leng[2]=	0.0115	-0.0080	0.3302	0.3381	24424
35	DBG 5:	leng[3]=	0.0075	0.3302	1.0000	0.6698	24626

Table 5.2 Work load balanced

The growth estimator, given in Eq(5.9), can be written as follows when a run executes sequentially:

$$E_0 = E_T \frac{\sum_j NGNSIZ}{\sum_j NNEW} \quad (5.11)$$

where *NGNSIZ* and *NNEW* are the accumulations up to the *j*th iteration. Although Eq(5.11) can be evaluated by the primary process at the time it computes the new subintervals for all dp processes, it introduces an enormous error into the estimator. The problem is that *NGNSIZ* is a constant expressing the number of points in a new population before sampling takes place. In the distributed computation, the *NGNSIZ* number of points in the new population is not guaranteed. From the way that *ngns_i* and *ngnct* are computed where *ngnct* holds the semantic of *NGNSIZ*, it should be recognized that there are some truncations because of the divisions. The difference between *NGNSIZ*, the expected number of points in a new population, and *ngnct*, the actual number of points in the new population, is about 0 – 5 points. It is a large error to introduce in to the estimator. Therefore, *NGNSIZ* in Eq(5.11) has to be replaced by *ngnct* which varies over the iterations.

The following table shows the variance of actual runs of the distributed program. The population used is a set of 100,000 points, and five process configurations are employed. One hundred runs are executed for each of the configurations.

one particle, one dimensional case (100 runs)					
100,000 points	4 proc	8 proc	16 proc	32 proc	48 proc
variance	.00000067	.00000098	.00000108	.00000085	.00000077

Table 5.3 The variance of the computation

6

Distributing Fermion Green's Function Monte Carlo

The Green's function Monte Carlo, (GFMC), has been used to obtain the exact solution for systems of many boson particles since 1961, [Kalos]. The problem of obtaining exact solutions for system of fermions remains. The essential problem is that the wave function sampled by GFMC for systems of fermions is partially negative. In 1980, a modified GFMC, called Fermion Green's function Monte Carlo (FGFMC), was developed and used to obtain the eigenvalue of the lowest fermion state of a system of three fermions [Arnow, 1981]. In this chapter, FGFMC is discussed, and the program to simulate the system of three fermions is reconstructed and evaluated in parallel.

6.1 The Green's function of many particles

In order to concentrate on the problem of interest, parallelizing FGFMC, the following discussion is restricted to a system of three particles. The identical particles are fermions, each moving in three dimensional space. Two of these have like-spin which differs from the third. The Schrödinger equation of a three-body system in $3D$ Cartesian coordinates is:

$$\left[-\sum_{i=1}^3 \frac{\hbar^2}{2m_i} \nabla^2 + V(x_1, x_2, x_3) \right] \psi(x_1, x_2, x_3) = E\psi(x_1, x_2, x_3). \quad (6.1)$$

where x_i denotes one of the three particles in real $3D$ space whose mass is denoted as m_i .

For convenience, let $\hbar^2/(2m_i) = 1$ and by an appropriate transformation, the particles in $3D$ coordinates in Eq.(6.1) are represented as vectors $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$ in 9-dimensional configuration space. With these changes, Eq.(6.1) can be written simply as:

$$[-\nabla_{\mathbf{R}}^2 + V(\mathbf{R})]\psi(\mathbf{R}) = E\psi(\mathbf{R}), \quad (6.2)$$

where $V(\mathbf{R})$ is a central potential comprised of two square potentials $V_{12}(\mathbf{r}_1, \mathbf{r}_2) + V_{12-3}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$. The potential of the system is discussed in section 2.1.3.2.

Conventionally, the energy is scaled so that the eigenvalue E is negative, and $E > (V_{12(wel)} + V_{12-3(wel)})$, where $V_{12(wel)}$ and $V_{12-3(wel)}$ are two constants less than zero. To make the potential positive everywhere for Monte Carlo sampling purposes, let $V_0 = |V_{12(wel)} + V_{12-3(wel)}|$ and assume that $V(\mathbf{R}) + V_0 \geq 0 \Rightarrow (E + V_0) \geq 0$.

Then Eq.(6.2) becomes

$$[-\nabla_{\mathbf{R}}^2 + V(\mathbf{R}) + V_0]\psi(\mathbf{R}) = (E + V_0)\psi(\mathbf{R}) \quad (6.3)$$

We seek a Green's function for the operator $[-\nabla_{\mathbf{R}}^2 + V(\mathbf{R}) + V_0]$ that satisfies

$$[-\nabla_{\mathbf{R}}^2 + V(\mathbf{R}) + V_0]G(\mathbf{R}, \mathbf{R}') = \delta(\mathbf{R} - \mathbf{R}') \quad (6.4)$$

with the same boundary condition as Eq.(6.1). Because it is in general impossible to analytically construct a Green's function for this operator, a Green's function that is known for a simpler operator is employed to construct random walks so the

sampled results are exactly the same as if we had employed the Green's function for the actual operator.

A Green's function for the operator $[-\nabla_{\mathbf{R}}^2 + U]$ can be explicitly expressed:

$$G_U(\mathbf{R}, \mathbf{R}') = \frac{1}{(2\pi)^{3N/2}} \frac{K_{3N/2-1}(|\mathbf{R} - \mathbf{R}'|)}{|\mathbf{R} - \mathbf{R}'|^{3N/2-1}} \quad (6.5)$$

where N denotes the number of particles in the system. $K_\nu(z)$ is the Bessel function of an imaginary argument. For the argument z of the Bessel function, vectors $\mathbf{R}$ are transformed into some appropriate spherical coordinates for sampling the Bessel function. Eq.(6.5) is a function of $|\mathbf{R} - \mathbf{R}'|$ and vanishes on the surface $\sum_{D_u(\mathbf{R})}$. Domain D_u covers the space where G_U is defined.

With the help of Green's theorem, an integral equation is constructed from Eqs. (6.4) and (6.5) for the operator $[-\nabla_{\mathbf{R}}^2 + V(\mathbf{R}) + V_0]$ on the domain D from which the solution is to be obtained:

$$\begin{aligned} G(\mathbf{R}, \mathbf{R}') = G_U(\mathbf{R}, \mathbf{R}') &+ \int_{\sum_{D_u(\mathbf{R}')} [-\nabla_n G_U(\mathbf{R}_1, \mathbf{R}')] G(\mathbf{R}, \mathbf{R}_1) dR_1 \\ &+ \int_{D_u(\mathbf{R}')} \left[\frac{U - V(\mathbf{R}) - V_0}{U} \right] U G_U(\mathbf{R}_1, \mathbf{R}') G(\mathbf{R}, \mathbf{R}_1) dR_1. \end{aligned} \quad (6.6)$$

For each $R' \in D$, construct a subdomain $D_u(R')$ such that $D_i \subset D$ and $\bigcup_i D_i = D$.

G_U is the analytically known Green's function given by Eq.(6.5) and is evaluated on the subdomains D_u . U is assigned a large positive value so that $[-\nabla^2 + U] > 0$

and

$$U \geq V(\mathbf{R}) + V_0 \quad \text{for} \quad \forall \mathbf{R} \in D_u$$

It, therefore, is the upper bound of the system potential.

The potential $V(\mathbf{R})$ of the system is understood as the strength that binds the particles in the system. By bounding the $V(\mathbf{R})$, by the constant U , an analytical Green's function can be written down and used to determine $G(\mathbf{R}, \mathbf{R}')$. Eq.(6.6) has three terms on the right hand side. The first term corresponds to sampling a next point from the Green's function, $G(\mathbf{R}, \mathbf{R}')$. The second term corresponds to a walker terminated at the boundary $\sum_{D_u}$ of the subdomain D_u and the third term corresponds to the walker absorbed in the interior of the D_u .

When a walker terminates at the surface of subdomain $D_u(\mathbf{R}')$, the position from which the walker will restart is determined from the probability function $\nabla_{\vec{n}} G_U(\mathbf{R}_1, \mathbf{R}') d(\sum_{D_u})$. Here $\mathbf{R}_1 \in \sum_{D_u(\mathbf{R}')}$, and the gradient of G_U is the outward directed derivative normal to the surface $\sum_{D_u}$. The probability that the walker terminated on the surface $\sum_{D_u(\mathbf{R}')}$ is given by $\int_{\sum_{D_u(\mathbf{R}')}} [\nabla_{\vec{n}} G_U(\mathbf{R}_1, \mathbf{R}')] d(\sum_{D_u})$.

The third term of Eq.(6.6) is the compensation by a fraction, $[U - V(\mathbf{R}) - V_0]/U$, for U set larger than the system potential $V(\mathbf{R})$. The excess absorption within the interior at the position $\mathbf{R}_1$ is determined by $pdf \frac{[U - V(\mathbf{R}) - V_0]}{U} G_U(\mathbf{R}_1, \mathbf{R}') d\mathbf{R}_1$. A walker may be legitimately absorbed by the first term of the right hand side of Eq.(6.6), and thus sample a next point from $G(\mathbf{R}, \mathbf{R}')$. The probability that the walker is absorbed in the interior of the subdomain is given by

$$\int_{D_u(\mathbf{R}')} \frac{[U - V(\mathbf{R}) - V_0]}{U} G_U(\mathbf{R}_1, \mathbf{R}') d\mathbf{R}_1.$$

6.2 Estimating the lowest antisymmetric state energy

By considering the Eq.(6.3) and Eq.(6.4), the corresponding integral equation can be obtained as:

$$\psi(\mathbf{R}) = (E + V_0) \int G(\mathbf{R}, \mathbf{R}') \psi(\mathbf{R}') d\mathbf{R}'. \quad (6.7)$$

Then the recurrent equation can be written with initial $\psi^{(0)}(\mathbf{R})$ by

$$\psi^{(n)}(\mathbf{R}) = (E_T + V_0) \int G(\mathbf{R}, \mathbf{R}') \psi^{(n-1)}(\mathbf{R}') d\mathbf{R}', \quad (6.8)$$

The eigenvalue E_0 of the equation is asymptotically obtained as a constant:

$$E_0 = \frac{(E_T + V_0) \int \psi^{(n-1)}(\mathbf{R}) d\mathbf{R}}{\int \psi^{(n)}(\mathbf{R}) d\mathbf{R}} - V_0. \quad (6.9)$$

A trial energy value E_T is used in the iterations. $\psi^{(n)}$ will grow in normalization reflected in a growth in the population of walkers if E_T is set larger than the actual eigenvalue E_0 ; otherwise, the population of $\psi^{(n)}$ will decline. The ratio of the population at beginning of the n th iteration to that of the ending of the iteration tends to be a constant when n becomes large.

From the discussion in chapter 5, the growth estimator can be constructed for iterating m times:

$$E_0 \simeq (E_T + V_0) \frac{mN}{\sum_{j=1}^m R_j} - V_0. \quad (6.10)$$

6.3 The Green's function for fermion systems

Often, a solution for a higher eigenvalue is desired, for instance, the energy of the lowest excited state. Ideally, it may be obtained if ψ_0 , the eigensolution corresponding to the lowest state, is known. ψ_1 can be iteratively approximated by the Neumann series in the same way as ψ_0 . To make it work, some pre-processing has to be done before each iteration starts. To guarantee that the iteration converges to the higher eigenstate, the initial population $\psi_1^{(0)}$ has to be chosen in such a way that it is orthogonal to the eigenstate ψ_0 , that is $\psi_1^{(0)}(r) = F(r) - c\psi_0(r)$, where $F(r)$ is an arbitrary set of points, perhaps selected from uniform distribution density function, and c is a constant determined by the equation,

$$c = \frac{\int F(r)\psi_0(r)dr}{\int [\psi_0(r)]^2 dx}.$$

Thus the ψ_0 components are subtracted from the initial population $F(x)$. This subtraction shall be carried out before each iteration takes place and as a result the eigenstate ψ_1 is approximated. By the same procedure, an even higher eigenstate can be obtained.

It is difficult, or precisely impossible, to carry out the above procedure when ψ_0 is represented as a set of points. First of all, unless ψ_0 is known to some high accuracy, the initial population supposedly being used to approximate the higher eigenstate may not be orthogonal to ψ_0 . Secondly, there is no known way of subtracting a probability density function represented by a set of points from another set of points. Furthermore, ψ_1 , an excited fermion solution, is not represented the

same way as ψ_0 , the lowest eigensolution. The difference is that ψ_0 is positive everywhere and easily represented by a stochastic process explicitly using random variables; whereas, ψ_1 is antisymmetric. In some regions of R , ψ_1 is positive and in other regions, ψ_1 is negative.

A scheme to carry out Green's function Monte Carlo for fermion systems was suggested by Arnow, [Arnow, 81, Arnow et al, 1982]. An antisymmetric eigensolution, ψ_F , is represented by two different sets of points, one for the positive region of ψ_f and the other for the negative region of ψ_f . The associated Green's function is defined as

$$G^\pm(R; R^+, R^-) = G(R, R^+) - G(R, R^-). \quad (6.11)$$

Consider two sets of points, $\psi^+(R^+)$ and $\psi^-(R^-)$, representing probability density functions in positive and negative regions, respectively. Using the Neumann series, the two independent populations may be iterated

$$\psi^{+(n)}(R^+) = E_F \int G(R^+, R^{+'}) \psi^{+(n-1)}(R^{+'}) dR^+, \quad (6.12)$$

$$\psi^{-(n)}(R^-) = E_F \int G(R^-, R^{-'}) \psi^{-(n-1)}(R^{-'}) dR^-. \quad (6.13)$$

Before each of the populations converges to the ground state, it consists of symmetric points as well as antisymmetric points, $\psi_n = \alpha_B \psi_B + \alpha_F \psi_F$. If separate iterations are carried out, each of these populations would converge to the

ground state, independently. To avoid this, the difference of two Green's functions is sampled from correlated points having opposite signs:

$$\begin{aligned}\psi^+(R^+) + \psi^-(R^-) &= \sum_j E_F G(R^+, R_j^{+'}) + \sum_j E_F G(R^-, R_j^{-'}) \\ &= E_F \sum_j G^\pm(R; R_j^+, R_j^-).\end{aligned}\tag{6.14}$$

It should be understood that the antisymmetric Green's function accepts a pair of points (R_j^+, R_j^-) selected from the two different populations in the current iteration, and R is a union of points produced by two series of random walks.

Suppose the integral $\int E_T G(R, R') \psi_F(R) dR$ is to be evaluated with the two populations R^+ and R^- and define $\psi_F(R) = \psi_F^+(R^+) - \psi_F^-(R^-)$. Given a pair (R^+, R^-) selected from $\psi_F^+(R)$ and $\psi_F^-(R)$ in one iteration, the expectation of their contribution to Eq.(6.14) is

$$\begin{aligned}& \int E_T G(R, R') [\psi_F^+(R') - \psi_F^-(R')] dR' \\ &= \int E_T G(R, R^+) \psi_F^+ dR^+ - \int E_T G(R, R^-) \psi_F^- dR^- \\ &= \int E_T G^\pm(R; R^+, R^-) \psi_F dR \\ &= \int_{>0} E_T |G^\pm(R; R^+, R^-)| \psi_F^+ dR - \int_{<0} E_T |G^\pm(R; R^+, R^-)| \psi_F^- dR,\end{aligned}\tag{6.15}$$

or written in the Monte Carlo way by two sets of points:

$$\{R^+\} \cup \{R^-\} = \sum_j E_T G^\pm(R; R_j^+, R_j^-).\tag{6.16}$$

6.4. Distribute FGPMC on DP system

In addition to the concerns of distributing the GPMC program discussed in chapter 5, such as process synchronization, data transmission, a new concern has to be dealt with. In distributing FGPMC, we must find pairs of antisymmetric points, $E-q.(6.14)$, from two different populations stored in the local memory of the distributed processes. In the following discussion, first, the unique characteristics of sampling FGPMC are presented. Then, the issues special to distributing Fermion Green's function Monte Carlo program are discussed. To help the discussion, outputs from a run of four processes are provided.

6.4.1 The representation of two populations

In representing the positive region and negative region of the wave function of the system of three fermions, two sampled populations are used. In the process of sampling, each of the population is treated as if there was no associated sign. In fact, computations using probability distribution functions can only sample the problem not involving signs because a negative probability is not well defined. Indeed, if each population is sampled independently, both of them will geometrically converge to the lowest eigensolution, while the goal of the algorithm is to obtain the eigenvalue of the lowest fermion state.

To prevent the solution from converging to the lowest eigenstate, a process that finds a pair of antisymmetric points in the two populations is employed. Suppose we have a method to find the points in the opposite populations that are the best match about the origin that make the points antisymmetric. (The method used to

find pairs will be discussed shortly.) To sample the two populations, a point is select alternatively, and that point is used to sample the Green's function by a random walk. For a given point, there may or may not be a match in the opposite population. If a match is not found, the point is used to sample the Green's function given in Eq(6.5), otherwise, the found pair of points are used to sample the antisymmetric Green's function discussed in section 6.3.

A pair found in the two opposite populations are almost identical in position. One of them is, at best, located in a small neighborhood of the other considering the points as if they were in the same population. For each of these points, a point with the opposite sign is literally added into the opposite population based upon some probability threshold. The pair of points then generate two series of random walks during sampling. This process mechanically strengthens the signal of antisymmetry.

In order to obtain the eigenvalue of the lowest fermion state, a large number of pairs has to be found during iterating Eq.(6.8), and, therefore, two large populations must be present. In other words, the density of the points in the 9D configuration space has to be high so that there will exist some pairs of points, R^+ and R^- that are close enough, or in adjacent "9D neighborhoods", to be found. A neighborhood in this problem refers to a nine-dimensional volume in configuration space.

In general, the number of n -dimensional " n D neighborhoods" is exponentially increasing when n increases. To achieve a high population density, a very large number of points has to present. In our problem, the points in the populations are not uniformly distributed in the entire 9D configuration space since only two of the

particles are antisymmetric. So it is not needed that the sizes of the populations increase exponentially. Experimentally, 10,000 points in each population can be used in a sequential run.

The reason to run the problem in parallel is to achieve sampling large populations. There are two factors to consider: 1) To sample the two populations, each has 10,000 points, about 1.5MB RAM is needed to store the populations, plus additional RAM to save intermediate results. The additional RAM may be very large. When 10,000 random walkers are sampled, there may be a need for nearly twice the RAM as is used to store the original populations, depending upon how the trial energy, E_T , is chosen. Even though, E_T is chosen smaller than E_0 , the intermediate results may still occupy large RAM for a short period of time, based on our observation. This type of computation usually reaches the limit of the memory, RAM, installed in a local host. Therefore, to run a population larger than 10,000 points, RAM's in multiple machines must be considered. 2) When large populations are employed, the speed of the CPU has to be taken into account. For each point, comparisons, arithmetic multiplications are applied on nine double precision floating points. They all take time. Multiple CPU's are needed to accelerate the computations.

A neighborhood in this problem refers to a nine-dimensional volume in the configuration space. Searching for a pair in such a space is expensive in terms of computational complexity, especially when a population is large. The best deal that can be accomplished in a reasonable period of time and not too costly is to use a

hash function so that a population is linearly partitioned into subsets of points.

Mechanically, a hash table is an array of keys that are generated by a hash function of integers. Items that are the subjects of a problem are mapped by the hash function as indices and inserted into the hash table, later they may be retrieved/searched by employing the same hash function for their indices. The hash tables, one for each population, are used here in the same way. Keys are generated by computing some criterion on a given point of nine coordinates. However, to achieve better pairing, two levels of keys are employed. One, the primary key that embodies information for the nine coordinates of a point, is used to index a hash table. Points having the same index are stored in linked-lists with no ordering. The linked-lists with the same primary index are differentiated by vectors of three integers, the refinement keys, or supplemental keys, which are obtained by computing coordinates of individual particles.

Neighbors of a point are considered as nine-dimensional units that are adjacent to the unit, in which the point is the geometric center. To avoid an exhaustive search for all adjacent neighbors, a 2^{th} -order neighboring cube is defined which helps to construct adjacent cubic units that are needed to be searched. If two N -dimensional cubes share an $(N-n)$ -dimensional surface, then they are called n^{th} -order neighboring cubes. What is needed to find pairs in the 2^{nd} -order cubic units is to compute the geometric center of the given point. Then a small value is added and subtracted to each of its nine coordinates, one at a time, to generate the geometric centers of 9D neighborhoods. And each of the geometric centers can be used to compute a hash

code to index some linked-list.

During sampling, whether or not a pair that can be found for a point determines which Green's function it is going to sample. As the initial position of each iteration during sampling, the geometric centers of all points are computed, and used to index into hash tables. The following diagram is what a hash table looks like:

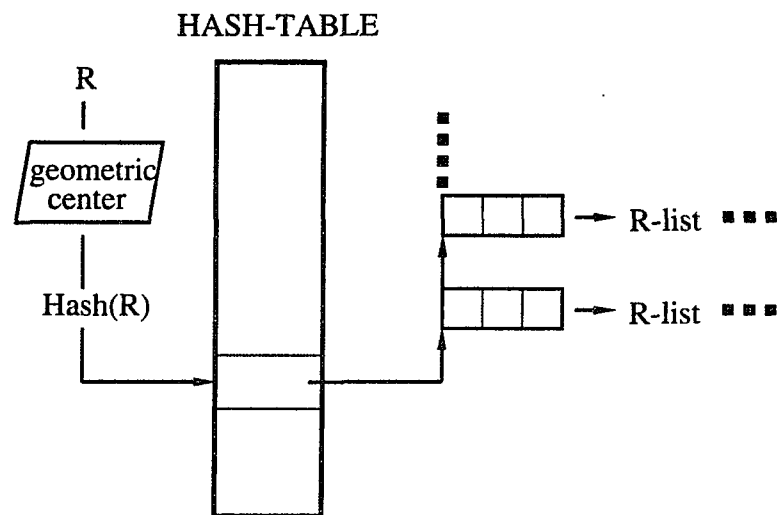


Figure 6.1 Hash system

Points in a R-list have the same refinement/supplemental key which is a vector of three integers, each is determined by the position of a particle in its three coordinates. There may be several R-lists having the same hash index, therefore, they are linked together forming a list. There are two such hash tables, one for each population. The process of pairing two points is as follows: randomly select a point

from a hash table; compute its 2^{nd} -order neighbors; hash each of the neighbors to the opposite hash table, search the linked-list for the same supplemental key; for each point in the linked-list with the supplemental key, obtain the geometric distance from the given point. The points having the smallest geometric distance less than some threshold is considered as a match, or a pair.

The following diagram shows the scheme of pairing. A point is chosen from the positive population, denoted as $r+$ from the leftmost vertical box, HASH-TABLE(+). Its neighbors in 9D space are computed and represented as an array of points, shown by the vertical box next to HASH-TABLE(+). The geometric center of each of those points is computed and used to index HASH-TABLE(-) shown by the next vertical box. Then the index list of R-list is searched. The only R-list that has the same supplemental keys is further searched for a match. Each point in the

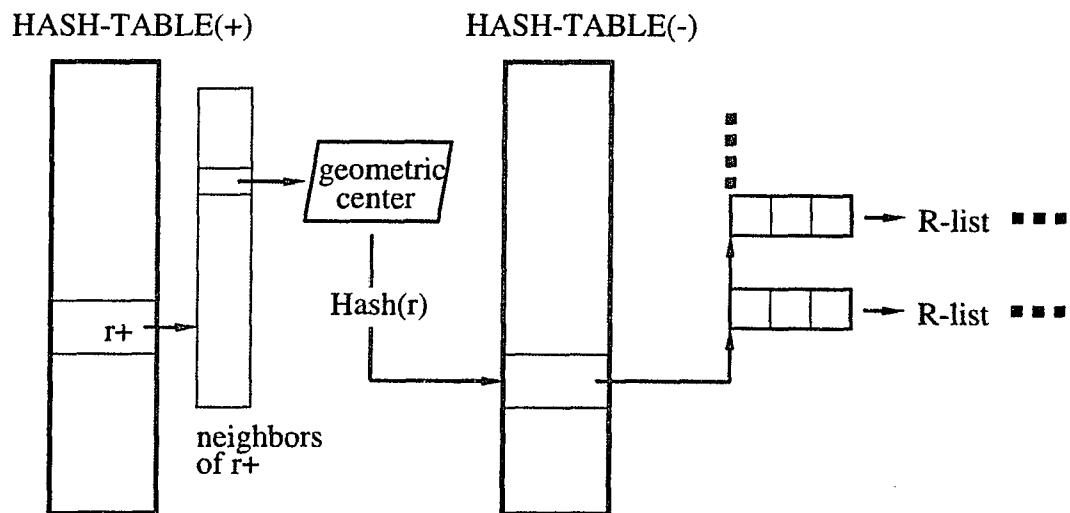


Figure 6.2 Pairing scheme

6.4.2 Distributing sampled populations

The linear distribution and discrete nature of the hash tables are used in distributing the populations in our parallel computation. The indices in the hash tables, are divided into n sets, and the sets of indices are assigned to n distributed worker processes. There are several advantages to such a partition. The primary one is that every process knows the boundaries in the hash table corresponding to other processes. This accelerates the process of pairing; Secondly, the hash tables are dimensionless so that this mechanism can be used to handle the points in any number of coordinates.

The purpose of distributing a Green's function Monte Carlo computation is to gain the ability to process large populations. However, due to the way that eigenvalue is estimated, sampled points have to be transmitted between processes through communication links. In the one-dimensional case discussed in chapter 5, points were moved across processes based on their locations at the end of each iteration. During sampling, some points tend to randomly walk to other intervals, such data movements can not be avoided. Moving a point of nine-coordinates is expensive. First, each coordinate is represented by 8 bytes (double precision). A point takes 72 bytes of memory and plus 16 bytes for its hash codes so that the re-computation of the hash codes can be saved. For a message system supporting 1000-byte packet size, a packet can carry only a few more than ten such points. Second, if a large volume of such data has to be transmitted in a short period of time in network, message collisions may be heavy. This situation happens when

an iteration ends, and all processes are sending messages to deliver points to all other processes. Even worse, such heavy message traffic can cause the distributed message systems to give up delivering packets when a certain number of retries are attempted.

An effort to reduce message traffic has to be made to accomplish the computation of large populations. The reduction in traffic is made possible by distributing points based on hashing. When neighbors are computed for a point, they tend to be in a very small set of indices in a hash table. Most of these indices have the same hash index as the initial point. Notice that the neighbors obtained for a point are used to index points in the opposite hash table. If the two hash tables are partitioned in the same way in the distributed processes, a plus point will be able to find most potential pairs in the minus population in the same local memory, or vice versa.

This decomposition of hash tables can substantially reduce message traffic because once a point can not be paired in the local memory, the question whether there is a pair that can be made globally has to be answered. To answer such questions, messages and data transmission have to be employed. For partitioning the same hash indices of both populations in the same local memory, it is highly probable that most neighbors can be found in the opposite hash table locally, and therefore, a high probability that pairs are made.

A hash function is usually a combination of simple arithmetic operations, that attempts to distribute hashed items evenly in the hash table. Here is another place

where an effort can be made to reduce message traffic if unbalanced hash tables are employed. It is observed that some hash functions can project a population into a hash table in such a way that the table is sparse, and many indices are empty. It helps to make the boundaries of the hash index assignments on the sparse areas. The boundaries can be cut very cleanly. Certainly, the hash function is experimental and can be varied. The assignments to hash indices then can be made based on the density of hash index collisions.

As an example to help the discussion, the output of a dp program of four processes is presented. Each process samples 5,000 points for each of its local population. 500 entries are defined for the hash tables and tables are evenly divided for four processes. A well-studied hash function is used to project the populations into the four hash intervals.

Table 6.1 is provided below to show that the number of points needed to be transmitted at the end of each iteration is very small. For the processes P0 and P1, no a point needs to be sent out, except at the end of the first iteration. It is observed that a small number of points are needed to be sent to other processes at the end of each iteration of P2 and P3.

At the end of each iteration, some points are subject to be moved to other processes based on their hash indices. The way to interpret the following table is: the first column: --,-- shows the iteration number and population sign; the second column: dplog-: shows that it is the output from the log file dplog-; the next four column show the number of points needed to be sent. For example, (0, 2067), in the first line, shows that 2067 points are needed to be sent to P0. However, since this is the output of P0, 2067 points will be moved locally. For another example, (1,0) says that no point will be sent to P1 to its positive population in the 2nd iteration by P0. (see line 3 in Table 6.1(P0))

```

01,P dplog0:(0, 2067) (1, 2105) (2, 428) (3, 400)
01,M dplog0:(0, 2134) (1, 2020) (2, 403) (3, 443)
02,P dplog0:(0, 3154) (1, 0) (2, 0) (3, 0)
02,M dplog0:(0, 3154) (1, 0) (2, 0) (3, 0)
03,P dplog0:(0, 4675) (1, 0) (2, 0) (3, 0)
03,M dplog0:(0, 4675) (1, 0) (2, 0) (3, 0)
04,P dplog0:(0, 4163) (1, 0) (2, 0) (3, 0)
04,M dplog0:(0, 4163) (1, 0) (2, 0) (3, 0)
05,P dplog0:(0, 3850) (1, 0) (2, 0) (3, 0)
05,M dplog0:(0, 3850) (1, 0) (2, 0) (3, 0)
06,P dplog0:(0, 3873) (1, 0) (2, 0) (3, 0)
06,M dplog0:(0, 3873) (1, 0) (2, 0) (3, 0)
07,P dplog0:(0, 3870) (1, 0) (2, 0) (3, 0)
07,M dplog0:(0, 3870) (1, 0) (2, 0) (3, 0)
08,P dplog0:(0, 3989) (1, 0) (2, 0) (3, 0)
08,M dplog0:(0, 3989) (1, 0) (2, 0) (3, 0)
09,P dplog0:(0, 4199) (1, 0) (2, 0) (3, 0)
09,M dplog0:(0, 4199) (1, 0) (2, 0) (3, 0)
10,P dplog0:(0, 4128) (1, 0) (2, 0) (3, 0)
10,M dplog0:(0, 4128) (1, 0) (2, 0) (3, 0)
11,P dplog0:(0, 4239) (1, 0) (2, 0) (3, 0)
11,M dplog0:(0, 4239) (1, 0) (2, 0) (3, 0)
12,P dplog0:(0, 4379) (1, 0) (2, 0) (3, 0)
12,M dplog0:(0, 4379) (1, 0) (2, 0) (3, 0)
13,P dplog0:(0, 4074) (1, 0) (2, 0) (3, 0)
13,M dplog0:(0, 4074) (1, 0) (2, 0) (3, 0)
14,P dplog0:(0, 4237) (1, 0) (2, 0) (3, 0)
14,M dplog0:(0, 4237) (1, 0) (2, 0) (3, 0)
15,P dplog0:(0, 4344) (1, 0) (2, 0) (3, 0)
15,M dplog0:(0, 4344) (1, 0) (2, 0) (3, 0)
16,P dplog0:(0, 4536) (1, 0) (2, 0) (3, 0)
16,M dplog0:(0, 4536) (1, 0) (2, 0) (3, 0)
17,P dplog0:(0, 4356) (1, 0) (2, 0) (3, 0)
17,M dplog0:(0, 4356) (1, 0) (2, 0) (3, 0)
18,P dplog0:(0, 4540) (1, 0) (2, 0) (3, 0)
18,M dplog0:(0, 4540) (1, 0) (2, 0) (3, 0)
19,P dplog0:(0, 4556) (1, 0) (2, 0) (3, 0)
19,M dplog0:(0, 4556) (1, 0) (2, 0) (3, 0)
20,P dplog0:(0, 4596) (1, 0) (2, 0) (3, 0)
20,M dplog0:(0, 4578) (1, 0) (2, 0) (3, 0)

```

Table 6.1(P0) Points moved at the end of iterations

At the end of each iteration, some points are subject to be moved to other processes based on their hash indices. The way to interpret the following table is: the first column: --,- shows the iteration number and population sign; the second column: dplog1: shows that it is the output from the log file dplog1; the next four column show the number of points needed to be sent. For example, (1, 2095), in the first line, shows that 2095 points are needed to sent to P1. However, since this is the output of P1, 2095 points will be moved locally. For another example, (0,0) says that no point will be sent to P0 to its positive population in the 2nd iteration by P1. (see line 3 in Table 6.1(P1))

```

01,P dplog1:(0, 2158) (1, 2095) (2, 383) (3, 364)
01,M dplog1:(0, 2091) (1, 2078) (2, 411) (3, 420)
02,P dplog1:(0, 0) (1, 2873) (2, 0) (3, 0)
02,M dplog1:(0, 0) (1, 2873) (2, 0) (3, 0)
03,P dplog1:(0, 0) (1, 5665) (2, 0) (3, 0)
03,M dplog1:(0, 0) (1, 5665) (2, 0) (3, 0)
04,P dplog1:(0, 0) (1, 5521) (2, 0) (3, 0)
04,M dplog1:(0, 0) (1, 5521) (2, 0) (3, 0)
05,P dplog1:(0, 0) (1, 5366) (2, 0) (3, 0)
05,M dplog1:(0, 0) (1, 5366) (2, 0) (3, 0)
06,P dplog1:(0, 0) (1, 4943) (2, 0) (3, 0)
06,M dplog1:(0, 0) (1, 4943) (2, 0) (3, 0)
07,P dplog1:(0, 0) (1, 5208) (2, 0) (3, 0)
07,M dplog1:(0, 0) (1, 5208) (2, 0) (3, 0)
08,P dplog1:(0, 0) (1, 5924) (2, 0) (3, 0)
08,M dplog1:(0, 0) (1, 5924) (2, 0) (3, 0)
09,P dplog1:(0, 0) (1, 5529) (2, 0) (3, 0)
09,M dplog1:(0, 0) (1, 5529) (2, 0) (3, 0)
10,P dplog1:(0, 0) (1, 5610) (2, 0) (3, 0)
10,M dplog1:(0, 0) (1, 5610) (2, 0) (3, 0)
11,P dplog1:(0, 0) (1, 5493) (2, 0) (3, 0)
11,M dplog1:(0, 0) (1, 5493) (2, 0) (3, 0)
12,P dplog1:(0, 0) (1, 5745) (2, 0) (3, 0)
12,M dplog1:(0, 0) (1, 5745) (2, 0) (3, 0)
13,P dplog1:(0, 0) (1, 5496) (2, 0) (3, 0)
13,M dplog1:(0, 0) (1, 5496) (2, 0) (3, 0)
14,P dplog1:(0, 0) (1, 5667) (2, 0) (3, 0)
14,M dplog1:(0, 0) (1, 5667) (2, 0) (3, 0)
15,P dplog1:(0, 0) (1, 5763) (2, 0) (3, 0)
15,M dplog1:(0, 0) (1, 5763) (2, 0) (3, 0)
16,P dplog1:(0, 0) (1, 5881) (2, 0) (3, 0)
16,M dplog1:(0, 0) (1, 5881) (2, 0) (3, 0)
17,P dplog1:(0, 0) (1, 5902) (2, 0) (3, 0)
17,M dplog1:(0, 0) (1, 5902) (2, 0) (3, 0)
18,P dplog1:(0, 0) (1, 5800) (2, 0) (3, 0)
18,M dplog1:(0, 0) (1, 5800) (2, 0) (3, 0)
19,P dplog1:(0, 0) (1, 5953) (2, 0) (3, 0)
19,M dplog1:(0, 0) (1, 5953) (2, 0) (3, 0)
20,P dplog1:(0, 0) (1, 5930) (2, 0) (3, 0)
20,M dplog1:(0, 0) (1, 5930) (2, 0) (3, 0)

```

Table 6.1(P1) Points moved at the end of iterations

At the end of each iteration, some points are subject to be moved to other processes based on their hash indices. The way to interpret the following table is: the first column: --,- shows the iteration number and population sign; the second column: dplog2: shows that it is the output from the log file dplog2; the next four column show the number of points needed to be sent. For example, (0, 2060), in the first line, shows that 2060 points are needed to be sent to P0 to its positive population at the end of the first iteration by P2.

```

01,P dplog2:(0, 2060) (1, 2132) (2, 409) (3, 399)
01,M dplog2:(0, 2117) (1, 2123) (2, 393) (3, 367)
02,P dplog2:(0, 43) (1, 39) (2, 1620) (3, 12)
02,M dplog2:(0, 40) (1, 59) (2, 1604) (3, 11)
03,P dplog2:(0, 1551) (1, 1406) (2, 1947) (3, 258)
03,M dplog2:(0, 1584) (1, 1425) (2, 1923) (3, 230)
04,P dplog2:(0, 1470) (1, 1175) (2, 2423) (3, 223)
04,M dplog2:(0, 1498) (1, 1227) (2, 2378) (3, 188)
05,P dplog2:(0, 1212) (1, 1000) (2, 2830) (3, 193)
05,M dplog2:(0, 1249) (1, 1056) (2, 2767) (3, 163)
06,P dplog2:(0, 927) (1, 834) (2, 3180) (3, 166)
06,M dplog2:(0, 969) (1, 904) (2, 3092) (3, 142)
07,P dplog2:(0, 658) (1, 668) (2, 3460) (3, 138)
07,M dplog2:(0, 706) (1, 735) (2, 3368) (3, 115)
08,P dplog2:(0, 16) (1, 19) (2, 3593) (3, 3)
08,M dplog2:(0, 58) (1, 57) (2, 3506) (3, 10)
09,P dplog2:(0, 358) (1, 365) (2, 3769) (3, 80)
09,M dplog2:(0, 416) (1, 409) (2, 3688) (3, 59)
10,P dplog2:(0, 288) (1, 296) (2, 3923) (3, 56)
10,M dplog2:(0, 335) (1, 331) (2, 3853) (3, 44)
11,P dplog2:(0, 193) (1, 199) (2, 4041) (3, 35)
11,M dplog2:(0, 231) (1, 232) (2, 3978) (3, 27)
12,P dplog2:(0, 129) (1, 117) (2, 4143) (3, 24)
12,M dplog2:(0, 154) (1, 161) (2, 4080) (3, 18)
13,P dplog2:(0, 208) (1, 217) (2, 4244) (3, 36)
13,M dplog2:(0, 242) (1, 231) (2, 4197) (3, 35)
14,P dplog2:(0, 85) (1, 86) (2, 4316) (3, 15)
14,M dplog2:(0, 117) (1, 97) (2, 4276) (3, 12)
15,P dplog2:(0, 31) (1, 49) (2, 4374) (3, 7)
15,M dplog2:(0, 63) (1, 52) (2, 4339) (3, 7)
16,P dplog2:(0, 4) (1, 11) (2, 4423) (3, 0)
16,M dplog2:(0, 23) (1, 22) (2, 4392) (3, 1)
17,P dplog2:(0, 75) (1, 77) (2, 4477) (3, 13)
17,M dplog2:(0, 99) (1, 76) (2, 4450) (3, 17)
18,P dplog2:(0, 21) (1, 24) (2, 4520) (3, 4)
18,M dplog2:(0, 32) (1, 35) (2, 4497) (3, 5)
19,P dplog2:(0, 98) (1, 110) (2, 4567) (3, 13)
19,M dplog2:(0, 96) (1, 121) (2, 4549) (3, 22)
20,P dplog2:(0, 76) (1, 88) (2, 4612) (3, 11)
20,M dplog2:(0, 79) (1, 97) (2, 4591) (3, 20)

```

Table 6.1(P2) Points moved at the end of iterations

At the end of each iteration, some points are subject to be moved to other processes based on their hash indices. The way to interpret the following table is: the first column: --,- shows the iteration number and population sign; the second column: dplog3:- shows that it is the output from the log file dplog3;- the next four column show the number of points needed to be sent. For example, (0, 2135), in the first line, shows that 2135 points are needed to be sent to P0 to its positive population at the end of the first iteration by P3.

```

01,P dplog3:(0, 2135) (1, 2106) (2, 388) (3, 371)
01,M dplog3:(0, 2113) (1, 2088) (2, 388) (3, 411)
02,P dplog3:(0, 445) (1, 439) (2, 61) (3, 1623)
02,M dplog3:(0, 393) (1, 381) (2, 68) (3, 1726)
03,P dplog3:(0, 2114) (1, 1419) (2, 251) (3, 1881)
03,M dplog3:(0, 2046) (1, 1363) (2, 244) (3, 2012)
04,P dplog3:(0, 2264) (1, 1200) (2, 217) (3, 2340)
04,M dplog3:(0, 2190) (1, 1149) (2, 209) (3, 2473)
05,P dplog3:(0, 1769) (1, 1022) (2, 194) (3, 2729)
05,M dplog3:(0, 1713) (1, 977) (2, 172) (3, 2852)
06,P dplog3:(0, 1854) (1, 876) (2, 161) (3, 3052)
06,M dplog3:(0, 1791) (1, 821) (2, 146) (3, 3185)
07,P dplog3:(0, 1747) (1, 758) (2, 131) (3, 3332)
07,M dplog3:(0, 1680) (1, 688) (2, 127) (3, 3473)
08,P dplog3:(0, 2318) (1, 642) (2, 110) (3, 3567)
08,M dplog3:(0, 2232) (1, 587) (2, 106) (3, 3712)
09,P dplog3:(0, 1716) (1, 596) (2, 99) (3, 3664)
09,M dplog3:(0, 1613) (1, 528) (2, 100) (3, 3834)
10,P dplog3:(0, 1467) (1, 526) (2, 83) (3, 3830)
10,M dplog3:(0, 1371) (1, 453) (2, 88) (3, 3994)
11,P dplog3:(0, 1369) (1, 472) (2, 75) (3, 3963)
11,M dplog3:(0, 1288) (1, 393) (2, 76) (3, 4122)
12,P dplog3:(0, 1535) (1, 427) (2, 64) (3, 4068)
12,M dplog3:(0, 1456) (1, 341) (2, 69) (3, 4228)
13,P dplog3:(0, 1268) (1, 395) (2, 56) (3, 4156)
13,M dplog3:(0, 1196) (1, 310) (2, 61) (3, 4308)
14,P dplog3:(0, 1130) (1, 356) (2, 51) (3, 4249)
14,M dplog3:(0, 1069) (1, 268) (2, 51) (3, 4398)
15,P dplog3:(0, 931) (1, 317) (2, 48) (3, 4316)
15,M dplog3:(0, 871) (1, 233) (2, 48) (3, 4460)
16,P dplog3:(0, 1025) (1, 291) (2, 43) (3, 4369)
16,M dplog3:(0, 960) (1, 217) (2, 41) (3, 4510)
17,P dplog3:(0, 928) (1, 270) (2, 39) (3, 4414)
17,M dplog3:(0, 859) (1, 201) (2, 39) (3, 4552)
18,P dplog3:(0, 897) (1, 246) (2, 34) (3, 4470)
18,M dplog3:(0, 826) (1, 183) (2, 33) (3, 4605)
19,P dplog3:(0, 660) (1, 230) (2, 34) (3, 4516)
19,M dplog3:(0, 606) (1, 171) (2, 26) (3, 4637)
20,P dplog3:(0, 678) (1, 201) (2, 29) (3, 4569)
20,M dplog3:(0, 623) (1, 147) (2, 24) (3, 4683)

```

Table 6.1(P3) Points moved at the end of iterations

6.4.3 Searching pairs in distributed memory

Once a pair can not be made for a sampled point in the local memory, the attempt has to continue globally. One possibility is to broadcast the question by sending the point over to all other distributed processes. However there is an immediate problem, only one random walk should be allowed to continue. If more than one pair can be made by different processes, which pair should be considered? Or if no pair can be made globally, which process is responsible for sampling that point when this point was duplicated to many copies in the distributed processes? Indeed, it needs some global knowledge to make such decisions and therefore not practicable for many such points.

Another possible solution is to send the point out in a round-robin fashion. This is worthwhile to do if the point can be matched by the early processes in the round-robin. However, if there is no pair that can be made, the data transmissions around processes are useless and wasted.

To avoid all this expensive and useless work, the appropriate solution is to guide the migration of a point that is looking for a match. In the beginning of each iteration, each process broadcasts its local hash tables to all other processes. The broadcasted information includes the boundaries of its indices and the number of points existing in the indices. By looking up the neighbors' hash tables before sending a point, unnecessary data transmission is avoided. Moreover, there is only one point that is looking for a match in the population. Whether or not that the point may be paired depends on the computation in the receiving process.

6.4.4 Distributing FGPMC

The structure of distributing the GPMC presented in chapter 5 (see code fragment 5.1) is used in the nearly the same way to distribute the FGPMC program. The only difference that has to be made is to deal with two populations and pairing points in the two sets of points. First, the GPMC sequential program was reconstructed, which was programmed first time by Arnow in 1981. The eigenvalue for the lowest excited state was obtained as discussed earlier. Then, the program was distributed by inserting codes that initialize the distributed environment; send and receive messages at the end of each iteration; synchronize all the distributed processes at certain points so that all processes can send appropriate information to the primary process to have energy estimated and finally, all processes start a new iteration at the same time (nearly the same physical time).

At the beginning of the computation, the primary process calls dp functions that initialize the desired number of secondary processes and establishes communication links. Then the primary sends messages to initialize local variables used for MC calculation in the secondaries. The messages carry sets of initial points for both populations and assignments telling the boundaries of the hash tables and indices for all the secondary processes.

Once this stage is settled, each process enters a loop that iterates the wave function by sampling the Green's function. At the end of each iteration, all processes

distribute their local information that is needed by other processes. To continue sampling in the next iteration, the points generated in the current iteration that are not located in the local indices have to be sent to the process which has the corresponding hash table.

Because the initial populations of the next iteration are selected evenly from the current populations, each process has to know how many points it has to choose from its local memory. The number of points stored in each local memory are generally not equal, so the number of initial points for the next iteration vary. To obtain the exact numbers that the processes need to initialize their populations for the next iteration, all the processes send a message informing the primary of the number of points actually stored in their memory. These numbers, one for each population, are obtained after all points are settled down in the processes they belong. The primary calculates the exact number of points that each of the process has to choose from their local memory. In this case, there are two such numbers for each process, the positive and negative populations.

To have better utilization of network communication during pairing points, the intervals of local hash tables are distributed so that the intervals are used to guide points to reach the best possible process where the points may be paired.

To distributed the hash tables, three steps of message transmission are used. In the first stage, all points were transferred into the processes they belong; (Table 6.1 shows the number of points need to be sent at the end of each iteration) in the second stage, the numbers of points that had to be chosen from the local memory

were received from the primary after reporting the number of points that were in the local memory; In the third stage of data communication, local hash indices were sent to all other processes after the two populations selected from the previous populations were hashed into tables. After the third stage of the communication, each local hash table embodied all information that the local process needs for sampling and pairing. The information includes the hash table intervals that the local process is responsible for and the points that are stored; the intervals that are corresponded to other processes and the number of points in each of the indices, which is some number, possibly zero. The actual points are stored remotely. These numbers can be looked up when a point is looking for a pairing.

Table 6.2 shows the distributed hash tables in the first five iterations. Each process maintains the same tables before sampling takes place.

The following table shows the hash distribution in the first five iterations. The hash index intervals are 0..125, 126..250, 251..375, 376..499 for four worker processes, respectively. The first column indicates the iteration number and population of two different signs; the second column shows the names of the worker processes: P0 to P3; the following columns show the number of points in the specific hash index. For example, [20]=18 indicates that 18 points are in the lists indexed by the 20th entry.

0							
01,plus	P0	[20]=18	[56]=4126	[57]=386	[110]=83	[111]=387	
01	P1	[156]=285	[157]=75	[210]=412	[211]=4208	[247]=20	
01	P2	[310]=943	[311]=2119	[320]=301	[347]=1637		
01	P3	[420]=1739	[447]=182	[456]=2199	[457]=880		
01,minus	P0	[20]=29	[56]=4163	[57]=363	[110]=92	[111]=353	
01	P1	[156]=257	[157]=95	[210]=541	[211]=4085	[247]=22	
01	P2	[310]=825	[311]=1966	[320]=230	[347]=1979		
01	P3	[420]=1848	[447]=115	[456]=2080	[457]=957		
0							
02,plus	P0	[20]=35	[56]=5839	[57]=616	[110]=112	[111]=464	
02	P1	[156]=467	[157]=111	[210]=545	[211]=5344	[247]=34	
02	P2	[310]=628	[311]=1362	[320]=216	[347]=1054		
02	P3	[420]=1122	[447]=116	[456]=1310	[457]=619		
02,minus	P0	[20]=49	[56]=5726	[57]=575	[110]=149	[111]=460	
02	P1	[156]=406	[157]=125	[210]=501	[211]=5344	[247]=52	
02	P2	[310]=534	[311]=1287	[320]=149	[347]=1273		
02	P3	[420]=1222	[447]=88	[456]=1384	[457]=676		
0							
03,plus	P0	[0]=945	[20]=35	[56]=5742	[57]=578	[110]=113	[111]=468
03	P1	[156]=502	[157]=125	[210]=693	[211]=6674	[247]=29	
03	P2	[310]=392	[311]=886	[320]=156	[347]=643		
03	P3	[420]=743	[447]=75	[456]=840	[457]=363		
03,minus	P0	[0]=1001	[20]=31	[56]=5598	[57]=579	[110]=131	[111]=508
03	P1	[156]=476	[157]=134	[210]=730	[211]=6608	[247]=40	
03	P2	[310]=356	[311]=856	[320]=86	[347]=749		
03	P3	[420]=770	[447]=52	[456]=904	[457]=392		
0							
04,plus	P0	[0]=1428	[20]=41	[56]=5022	[57]=511	[110]=83	[111]=437
04	P1	[156]=398	[157]=125	[210]=641	[211]=6319	[247]=39	
04	P2	[310]=463	[311]=1086	[320]=156	[347]=810		
04	P3	[420]=893	[447]=90	[456]=1017	[457]=441		
04,minus	P0	[0]=1416	[20]=52	[56]=4969	[57]=465	[110]=153	[111]=424
04	P1	[156]=464	[157]=133	[210]=676	[211]=6206	[247]=44	
04	P2	[310]=417	[311]=1026	[320]=97	[347]=924		
04	P3	[420]=931	[447]=50	[456]=1060	[457]=494		
0							
05,plus	P0	[0]=1083	[20]=20	[56]=4672	[57]=489	[110]=104	[111]=408
05	P1	[156]=480	[157]=134	[210]=614	[211]=6077	[247]=23	
05	P2	[310]=534	[311]=1263	[320]=200	[347]=1002		
05	P3	[420]=1068	[447]=117	[456]=1170	[457]=542		
05,minus	P0	[0]=1127	[20]=31	[56]=4630	[57]=417	[110]=143	[111]=409
05	P1	[156]=447	[157]=130	[210]=687	[211]=6024	[247]=51	
05	P2	[310]=455	[311]=1185	[320]=130	[347]=1144		
05	P3	[420]=1086	[447]=87	[456]=1263	[457]=554		

Table 6.2 Distributed hash tables

Now, it is time to sample the Green's function by dealing with the two populations. Points, in turn, were chosen from two different hash tables. Once a point is chosen, first, its geometric center is computed. From it, all the 2nd order neighbors are computed. Each of these neighbors is represented as the geometric center of a 9D unit, and it is used to calculate the hash index to the table of the opposite population. If the indexed hash table entry is not empty, then the points linked to the entry are compared with the original point for geometric distance. The distance that is less than some threshold is considered a match. The search goes through all the neighbor geometric centers. If a pair is found, it is treated as the *best* pair that can be found. Here we assume that the *best* pair locally implies the best possible pair for the entire population.

If a pair is found, each of the points is used to sample more or less points by a random walk. However, if no pair can be found, the problem becomes difficult. In the sequential program, a point that can not be paired is a symmetric point, therefore, it samples the symmetric Green's function. Here whether or not to sample the symmetric Green's function can not be decided until the global population is searched. Otherwise, the wave function of the excited ground state of the system does not converge.

So the effort has to be made to search the global population. The problem is what type of message operation can be used to carry the points looking for a pairing. Consider that the point migrates to another process for pairing. It may or may not be paired, this does not matter. What matters is that when the random

walk continues, some points that are generated by this point may migrate back or to another process. In that sense, large number of points have to be moved dynamically from process to process. Blocked message operation is out of the question, since synchronization is needed and too many such messages operations are transmitted.

An unblocked message has the characteristic that no synchronization is needed. It will preserve the parallel computation without inserting barriers in the sequence of instructions. Because no synchronization is needed, it triggers another problem when a computation passes from one iteration to another iteration. Messages may be sent in one iteration, and received when the destination process enters the next iteration.

Suppose that one process has no point left in its local hash tables. It executes a blocked operation to prepare to enter the next iteration by synchronizing with other processes. And also suppose that all processes do not know the status of other processes. Messages are still coming in to the process, however, no more receive operations are issued to take care of these message since the process is expecting synchronous messages. One possible solution to handle this situation is to discard the messages when the next iteration is entered. From our observation, large numbers of such points may be lost, and therefore, the results are affected, especially, since some processes run much *faster* than other processes. If a process gains 90 percent or more CPU time from its host, it may quickly finish its sampling work and dispatch the points, without receiving any request for pairing from other processes.

On the other hand, if the points are not discarded when entering the next iteration, the destination process has to step back to the current iteration once it receives a request for pairing. Or, if a process does not step into the next iteration once its sampling work is done, and waits for requests, it may wait forever since no such request been made by any other processes. If all processes wait for requests, a deadlock occurs.

It will be very inefficient to guarantee that no point is discarded. All processes defer entering the synchronization stage, by running a loop polling messages that carry requested for pairing. It may be that a few points that have long random walks making the computation longer, most of the time is contributed to message transmission.

The problem seems very sticky with no appropriate solution. We can think of a solution as a passive one rather than an active one. That is, one process can not send a point requesting pairing to another process unless the permission is granted by the receiving process. It can be easily implemented. When a process is ready to enter the synchronization stage after its points are all handled, it first send a message to all other processes not to send any request for pairing. After that it can poll messages that were sent before other processes received the “do not sending pairing request” message. The newly received points are inserted into hash tables according to which population it belongs. The sign is sent associated with the point itself. The point will go through the sampling process as if it was located in the local memory.

Even though no point will be discarded when completing an iteration, there is a problem relating to the above solution. If one process runs very fast, it may close its door early and refuse to receive paring requests from other *slower* processes. The result is that points can not be paired which when should be.

Because the paring is handled dynamically in processing distributed populations, there are some race-conditions in terms of finding a pair. There may be some points which should be paired, while for some reason they are not. For example, the opposite points were in transmission, or not yet generated by other processes on time. To judge whether or not the solution is reflecting the nature of pairing or efficiency, the results generated by the processes have to be checked.

Tables 6.3 given below shows the number of points having neighbors located at other processes. This table has to be used together with table 6.2. In this table, some neighbors hit some indices that no points associated with. For instance, in the 2nd iteration, P0 demands some matches in index 319, which is P2 sponsored index. By looking at the distributed tables, there is no a point indicated in the index in P2. The points indexing 319 can not be paired.

The following table shows the number of times that P0 hits the foreign positive hash tables. The output is produced by the process P0. There are no miss hits recorded in the indices 0..125. For those points that hit the intervals of other processes may need to be transmitted if the distributed hash tables show some points can be compared with. The first four rows indicate the situation in the 1st iteration.

```

01,P0::
01,P1::      [156]=340 [157]=264 [210]=282 [211]=225 [247]=284
01,P2::      [310]=240 [311]=259 [320]=342 [347]=311
01,P3::      [420]=283 [447]=214 [456]=242 [457]=234
02,P0::
02,P1::      [147]=32 [148]=2 [156]=628 [157]=620 [210]=530
*            [211]=405 [219]=13 [220]=34 [247]=552 [248]=73
02,P2::      [256]=43 [257]=36 [310]=423 [311]=523 [319]=69
*            [320]=708 [347]=589 [348]=83 [356]=32 [357]=50
02,P3::      [410]=38 [411]=33 [419]=76 [420]=506 [447]=410
*            [448]=50 [456]=587 [457]=537
03,P0::
03,P1::      [147]=35 [148]=13 [156]=785 [157]=798 [210]=620
*            [211]=418 [219]=12 [220]=30 [247]=623 [248]=62
03,P2::      [256]=54 [257]=40 [310]=470 [311]=507 [319]=75
*            [320]=825 [347]=617 [348]=67 [356]=55 [357]=35
03,P3::      [410]=35 [411]=24 [419]=69 [420]=541 [447]=481
*            [448]=83 [456]=550 [457]=854
04,P0::
04,P1::      [147]=36 [148]=7 [156]=788 [157]=699 [210]=594
*            [211]=355 [219]=9 [220]=20 [247]=662 [248]=63
04,P2::      [256]=60 [257]=35 [310]=425 [311]=407 [319]=63
*            [320]=959 [347]=659 [348]=74 [356]=35 [357]=27
04,P3::      [410]=26 [411]=38 [419]=69 [420]=575 [447]=460
*            [448]=36 [456]=616 [457]=821
05,P0::
05,P1::      [147]=10 [156]=749 [157]=704 [210]=511 [211]=399
*            [219]=6 [220]=15 [247]=667 [248]=55
05,P2::      [256]=28 [257]=37 [310]=455 [311]=355 [319]=53
*            [320]=904 [347]=670 [348]=51 [356]=28 [357]=21
05,P3::      [410]=16 [411]=7 [419]=58 [420]=452 [447]=449
*            [448]=43 [456]=511 [457]=836
...
48,P0::
48,P1::      [147]=2 [156]=441 [157]=405 [210]=227 [211]=194
*            [220]=2 [247]=465 [248]=11
48,P2::      [256]=3 [257]=1 [310]=165 [311]=62 [319]=6
*            [320]=3651 [347]=319 [348]=3 [356]=3
48,P3::      [410]=5 [419]=6 [420]=150 [447]=91 [448]=1
*            [456]=60 [457]=825
49,P0::
49,P1::      [147]=2 [156]=428 [157]=353 [210]=214 [211]=162
*            [220]=2 [247]=501 [248]=5
49,P2::      [256]=4 [257]=6 [310]=157 [311]=57 [319]=2
*            [320]=3583 [347]=313 [348]=9 [356]=3
49,P3::      [410]=1 [419]=3 [420]=150 [447]=125 [448]=1
*            [456]=48 [457]=734
50,P0::
50,P1::      [156]=465 [157]=370 [210]=187 [211]=149
*            [220]=7 [247]=462 [248]=7
50,P2::      [256]=5 [257]=5 [310]=157 [311]=71 [319]=3
*            [320]=3758 [347]=255 [348]=8 [357]=1
50,P3::      [410]=4 [419]=4 [420]=157 [447]=104
*            [448]=5 [456]=50 [457]=763

```

Table 6.3(P0) Neighbors hit other process' area

A similar table below shows the situation in process P1.

01,P0::	[20]=290	[56]=213	[57]=255	[110]=402	[111]=398
01,P1::					
01,P2::	[310]=266	[311]=325	[320]=170	[347]=237	
01,P3::	[420]=234	[447]=255	[456]=156	[457]=159	
02,P0::	[10]=27	[11]=36	[19]=65	[20]=921	[47]=12
*	[48]=7	[56]=754	[57]=899	[110]=1012	[111]=1132
*	[119]=11	[120]=37			
02,P1::					
02,P2::	[256]=36	[257]=31	[310]=865	[311]=827	[319]=58
*	[320]=665	[347]=626	[348]=92	[356]=29	[357]=11
02,P3::	[410]=28	[411]=42	[419]=79	[420]=813	[447]=866
*	[448]=64	[456]=605	[457]=594		
03,P0::	[10]=37	[11]=35	[19]=60	[20]=1112	[47]=40
*	[48]=9	[56]=828	[57]=945	[110]=1208	[111]=1143
*	[119]=11	[120]=40			
03,P1::					
03,P2::	[256]=34	[257]=42	[310]=929	[311]=800	[319]=83
*	[320]=760	[347]=741	[348]=93	[356]=38	[357]=17
03,P3::	[410]=49	[411]=56	[419]=59	[420]=1037	[447]=1009
*	[448]=88	[456]=689	[457]=644		
04,P0::	[10]=32	[11]=52	[19]=51	[20]=1050	[47]=41
*	[48]=8	[56]=633	[57]=929	[110]=1169	[111]=1187
*	[119]=17	[120]=36			
04,P1::					
04,P2::	[256]=26	[257]=31	[310]=1002	[311]=767	[319]=65
*	[320]=727	[347]=662	[348]=68	[356]=16	[357]=39
04,P3::	[410]=37	[411]=26	[419]=62	[420]=920	[447]=1061
*	[448]=78	[456]=632	[457]=573		
05,P0::	[10]=40	[11]=20	[19]=58	[20]=944	[47]=24
*	[48]=3	[56]=534	[57]=864	[110]=1095	[111]=1097
*	[119]=13	[120]=12			
05,P1::					
05,P2::	[256]=23	[257]=28	[310]=976	[311]=563	[319]=37
*	[320]=636	[347]=529	[348]=38	[356]=23	[357]=31
05,P3::	[410]=19	[411]=29	[419]=35	[420]=861	[447]=1024
*	[448]=79	[456]=532	[457]=606		
48,P0::	[10]=2	[11]=4	[19]=2	[20]=1400	[47]=3
*	[48]=1	[56]=211	[57]=388	[110]=709	[111]=902
*	[119]=1	[120]=10			
48,P1::					
48,P2::	[256]=4	[257]=4	[310]=501	[311]=84	[319]=3
*	[320]=463	[347]=136	[348]=13	[356]=1	[357]=8
48,P3::	[410]=3	[411]=6	[419]=7	[420]=606	[447]=1530
*	[448]=10	[456]=106	[457]=406		
49,P0::	[11]=3	[19]=8	[20]=1395	[47]=6	[56]=163
*	[57]=342	[110]=562	[111]=843	[120]=10	
49,P1::					
49,P2::	[256]=2	[257]=2	[310]=474	[311]=110	[319]=1
*	[320]=473	[347]=89	[348]=11	[356]=1	
49,P3::	[410]=2	[411]=3	[419]=5	[420]=639	[447]=1605
*	[448]=11	[456]=80	[457]=373		
50,P0::	[10]=2	[11]=2	[19]=5	[20]=1455	[47]=2
*	[48]=2	[56]=178	[57]=364	[110]=678	[111]=867
*	[120]=1				
50,P1::					
50,P2::	[257]=6	[310]=487	[311]=107	[319]=3	[320]=475
*	[347]=118	[348]=20	[356]=1	[357]=2	
50,P3::	[419]=8	[420]=580	[447]=1593	[448]=5	[456]=52
*	[457]=448				

Table 6.3(P1) Neighbors hit other process' area

The following table shows the situation in process P2.

01,P0::	[20]=175	[56]=173	[57]=68	[110]=239	[111]=300
01,P1::	[156]=118	[157]=79	[210]=279	[211]=225	[247]=241
01,P2::					
01,P3::	[420]=60	[447]=128	[456]=96	[457]=49	
02,P0::	[10]=33	[11]=32	[19]=53	[20]=587	[47]=17
*	[48]=20	[56]=882	[57]=369	[110]=864	[111]=1087
*	[119]=1	[120]=7			
02,P1::	[147]=11	[148]=12	[156]=655	[157]=352	[210]=1058
*	[211]=766	[219]=17	[220]=25	[247]=664	[248]=46
02,P2::					
02,P3::	[410]=25	[411]=46	[419]=48	[420]=471	[447]=696
*	[448]=46	[456]=404	[457]=333		
03,P0::	[10]=27	[11]=12	[19]=29	[20]=551	[47]=19
*	[48]=5	[56]=873	[57]=343	[110]=871	[111]=935
*	[119]=2	[120]=13			
03,P1::	[147]=14	[148]=8	[156]=639	[157]=255	[210]=946
*	[211]=661	[219]=3	[220]=7	[247]=644	[248]=20
03,P2::					
03,P3::	[410]=18	[411]=18	[419]=29	[420]=476	[447]=743
*	[448]=25	[456]=451	[457]=326		
04,P0::	[10]=18	[11]=17	[19]=25	[20]=540	[47]=13
*	[48]=3	[56]=943	[57]=328	[110]=807	[111]=1145
*	[119]=3	[120]=11			
04,P1::	[147]=12	[148]=4	[156]=589	[157]=247	[210]=1023
*	[211]=716	[219]=5	[220]=8	[247]=615	[248]=12
04,P2::					
04,P3::	[410]=17	[411]=17	[419]=16	[420]=468	[447]=834
*	[448]=42	[456]=471	[457]=317		
05,P0::	[10]=14	[11]=20	[19]=18	[20]=548	[47]=11
*	[48]=9	[56]=1040	[57]=311	[110]=747	[111]=1223
*	[119]=2	[120]=8			
05,P1::	[147]=18	[148]=4	[156]=477	[157]=266	[210]=1119
*	[211]=708	[219]=5	[220]=14	[247]=603	[248]=18
05,P2::					
05,P3::	[410]=12	[411]=20	[419]=29	[420]=473	[447]=870
*	[448]=34	[456]=378	[457]=302		
...					
48,P0::	[10]=2	[11]=1	[19]=4	[20]=485	[47]=5
*	[56]=1601	[57]=189	[110]=380	[111]=1496	[120]=1
48,P1::	[147]=2	[148]=2	[156]=386	[157]=201	[210]=578
*	[211]=535	[220]=2	[247]=128	[248]=4	
48,P2::					
48,P3::	[410]=1	[411]=1	[419]=1	[420]=394	[447]=378
*	[456]=70	[457]=388			
49,P0::	[10]=3	[11]=4	[19]=1	[20]=463	[47]=1
*	[56]=1590	[57]=136	[110]=385	[111]=1401	[120]=3
49,P1::	[147]=8	[156]=342	[157]=157	[210]=808	[211]=387
*	[219]=1	[220]=4	[247]=107	[248]=6	
49,P2::					
49,P3::	[410]=6	[411]=7	[419]=4	[420]=326	[447]=281
*	[448]=3	[456]=92	[457]=313		
50,P0::	[11]=2	[19]=7	[20]=425	[56]=1595	[57]=106
*	[110]=465	[111]=1380	[120]=2		
50,P1::	[147]=1	[156]=260	[157]=92	[210]=832	[211]=454
*	[219]=1	[220]=1	[247]=95	[248]=6	
50,P2::					
50,P3::	[410]=1	[411]=1	[419]=3	[420]=401	[447]=293
*	[448]=10	[456]=60	[457]=331		

Table 6.3(P2) Neighbors hit other process' area

The following table shows the situation in process P3.

01,P0::	[20]=369	[56]=419	[57]=473	[110]=131	[111]=214
01,P1::	[156]=468	[157]=339	[210]=130	[211]=256	[247]=196
01,P2::	[310]=125	[311]=245	[320]=460	[347]=239	
01,P3::					
02,P0::	[11]=15	[19]=38	[20]=914	[47]=5	[48]=10
*	[56]=872	[57]=1105	[110]=385	[111]=633	[119]=13
*	[120]=11				
02,P1::	[147]=9	[148]=5	[156]=1208	[157]=916	[210]=453
*	[211]=869	[220]=18	[247]=592	[248]=45	
02,P2::	[256]=25	[257]=30	[310]=360	[311]=587	[319]=27
*	[320]=906	[347]=479	[348]=59	[356]=35	[357]=42
02,P3::					
03,P0::	[10]=9	[11]=10	[19]=28	[20]=747	[47]=6
*	[48]=1	[56]=768	[57]=1070	[110]=376	[111]=706
*	[119]=6	[120]=12			
03,P1::	[147]=7	[148]=3	[156]=1135	[157]=884	[210]=393
*	[211]=812	[219]=7	[220]=10	[247]=667	[248]=23
03,P2::	[256]=11	[257]=16	[310]=420	[311]=557	[319]=39
*	[320]=1055	[347]=518	[348]=23	[356]=21	[357]=14
03,P3::					
04,P0::	[10]=2	[11]=4	[19]=15	[20]=674	[47]=10
*	[48]=8	[56]=757	[57]=1068	[110]=385	[111]=651
*	[119]=2	[120]=7			
04,P1::	[147]=14	[148]=1	[156]=1237	[157]=811	[210]=349
*	[211]=1066	[219]=9	[220]=16	[247]=557	[248]=30
04,P2::	[256]=10	[257]=22	[310]=374	[311]=441	[319]=30
*	[320]=857	[347]=485	[348]=20	[356]=15	[357]=19
04,P3::					
05,P0::	[10]=6	[11]=12	[19]=13	[20]=667	[47]=12
*	[48]=7	[56]=772	[57]=1257	[110]=309	[111]=710
*	[119]=5	[120]=12			
05,P1::	[147]=2	[148]=2	[156]=1370	[157]=796	[210]=302
*	[211]=1266	[220]=19	[247]=474	[248]=30	
05,P2::	[256]=16	[257]=20	[310]=367	[311]=502	[319]=25
*	[320]=1055	[347]=505	[348]=26	[356]=12	[357]=24
05,P3::					
...					
48,P0::	[11]=1	[19]=5	[20]=276	[47]=2	[56]=276
*	[57]=792	[110]=124	[111]=668	[120]=2	
48,P1::	[156]=761	[157]=523	[210]=329	[211]=4804	[219]=1
*	[220]=7	[247]=209	[248]=4		
48,P2::	[257]=3	[310]=233	[311]=123	[319]=9	[320]=914
*	[347]=199	[348]=9	[356]=4	[357]=11	
48,P3::					
49,P0::	[11]=3	[19]=5	[20]=228	[47]=2	[56]=329
*	[57]=717	[110]=141	[111]=614	[120]=5	
49,P1::	[147]=1	[156]=740	[157]=617	[210]=245	[211]=4632
*	[220]=6	[247]=251	[248]=9		
49,P2::	[256]=4	[257]=5	[310]=223	[311]=105	[319]=12
*	[320]=797	[347]=222	[348]=9	[356]=1	[357]=4
49,P3::					
50,P0::	[10]=1	[11]=4	[19]=5	[20]=220	[47]=3
*	[48]=1	[56]=384	[57]=761	[110]=143	[111]=580
*	[120]=4				
50,P1::	[147]=1	[156]=690	[157]=783	[210]=223	[211]=4627
*	[220]=8	[247]=205	[248]=7		
50,P2::	[256]=1	[257]=1	[310]=228	[311]=137	[319]=13
*	[320]=981	[347]=229	[348]=4	[357]=5	
50,P3::					

Table 6.3(P3) Neighbors hit other process' area

Table 6.4 shows the number of points sent to and received from by each of the processes. The total number of points sent and received are equal within an iteration.

```

0
01:P0 RECEIVE-SEND:501, 349
01:P1 RECEIVE-SEND:445, 380
01:P2 RECEIVE-SEND:266, 444
01:P3 RECEIVE-SEND:252, 291
0
02:P0 RECEIVE-SEND:1502, 1758
02:P1 RECEIVE-SEND:1457, 1608
02:P2 RECEIVE-SEND:1188, 1092
02:P3 RECEIVE-SEND:1135, 824
0
03:P0 RECEIVE-SEND:2402, 3371
03:P1 RECEIVE-SEND:2363, 3143
03:P2 RECEIVE-SEND:2190, 1464
03:P3 RECEIVE-SEND:2186, 1163
0
04:P0 RECEIVE-SEND:3320, 4970
04:P1 RECEIVE-SEND:3313, 4571
04:P2 RECEIVE-SEND:3168, 1922
04:P3 RECEIVE-SEND:3221, 1559
0
05:P0 RECEIVE-SEND:4315, 6496
05:P1 RECEIVE-SEND:4301, 6038
05:P2 RECEIVE-SEND:4138, 2455
05:P3 RECEIVE-SEND:4266, 2031
...
48:P0 RECEIVE-SEND:56028, 63609
48:P1 RECEIVE-SEND:54054, 58162
48:P2 RECEIVE-SEND:42155, 39641
48:P3 RECEIVE-SEND:44101, 34926
0
49:P0 RECEIVE-SEND:57194, 64769
49:P1 RECEIVE-SEND:55084, 59189
49:P2 RECEIVE-SEND:42986, 40481
49:P3 RECEIVE-SEND:44860, 35685
0
50:P0 RECEIVE-SEND:58354, 65930
50:P1 RECEIVE-SEND:56123, 60193
50:P2 RECEIVE-SEND:43808, 41318
50:P3 RECEIVE-SEND:45611, 36455

```

Table 6.4 The number of *unblocked* messages

Table 6.4 shows the number of *unblocked* messages used to deliver the points searching for matches. The first column of the table consists of the iteration number and the process number; two numbers in each line show the number of messages received and sent in the iteration. Four lines are grouped for each iteration. The total number of messages received by all processes in an iteration is the same as that of messages sent. The numbers shown are accumulative.

Once a point is sent out, the point may travel to several processes searching for a pairing. To avoid the point is being sent back to a process it already has visited, a travel history is kept with the point. The history is stored in a two-byte unsigned integer. It is sufficient to hold a the history of sixteen distributed processes.

Now, we discuss in detail how a decision is made to migrate a point to another process. When a point looks for a pair, it first obtains geometric centers for all its 2nd order neighbors. And the indices in the hash table for the geometric centers are also computed and stored in an array. The indices of the array may involve many different subintervals that correspond to separate processes. Then, a pair is searched for from those indices that reside in the corresponding subinterval of the hash table of the opposite population. If the indexed hash slot is not empty and the supplemental key is matched, then distances between the points in the list and the point are calculated. If a pair is found, the Green's function for fermions is sampled . Otherwise, a decision whether or not to deliver the point to other processes has to be made.

If no indices of the geometric centers of the neighbors refer to an interval other

than the local one, then no pair can be made. If an index of the geometric centers of the neighbors refer to an interval other than the local one and the hash index is empty, the point is not going to be sent. If the foreign hash index is not empty, then the travel history for the local process is set off. If the travel history bit for the destination process is not set off, the point is sent associated with its hash index, the sign of the point, and its travel history to the process which is responsible for the index.

When the point is received by the destination process, it will be stored in the indexed hash table for the correct population. It becomes a local point. When it is selected for sampling from its hash table, the geometric centers of its neighbors are recomputed by the new process. If a pair can not be made by the local process, and some requested foreign hash indices are not empty, it will be sent again. Because the travel history records the processes it has been gone, the point only flows to a process to which it has not yet reached. Eventually, all the places that should searched have been gone through, a decision which Green's function to sample can be made.

The distributed paring mechanism can be summarized as following. In the beginning of sampling for each iteration, two populations stored are in two hash tables. Then a point is chosen alternately from the populations. If a positive point is chosen, a pair is expected with a point in the negative population, or vice versa. The first effort to make a pair is to look at the points stored locally. If no pair can be found, then the effort to search for a pair from other process is made. The migrate

direction is decided by the hash tables which is initialized before the sampling takes place. The point may have to travel to several processes, but it will not travel back to a process it has already visited.

Finally, table 6.5 below shows the number of points in each population before an iteration starts. The initial sets are 20,000 points for each population.

	PROCESS P0		PROCESS P1		PROCESS P2		PROCESS P3	
ITE=1	P=8419	M=8454	P=8437	M=8308	P=1607	M=1594	P=1533	M=1640
ITE=2	P=3641	M=3586	P=3350	M=3312	P=1680	M=1671	P=1634	M=1736
ITE=3	P=8339	M=8304	P=8489	M=8452	P=2197	M=2166	P=2138	M=2241
ITE=4	P=7896	M=7850	P=7895	M=7896	P=2639	M=2586	P=2562	M=2660
ITE=5	P=6830	M=6811	P=7387	M=7398	P=3023	M=2938	P=2921	M=3014
ITE=6	P=6653	M=6632	P=6652	M=6667	P=3340	M=3237	P=3217	M=3326
ITE=7	P=6274	M=6255	P=6633	M=6630	P=3590	M=3494	P=3469	M=3587
ITE=8	P=6322	M=6278	P=6584	M=6567	P=3702	M=3611	P=3569	M=3721
ITE=9	P=6272	M=6227	P=6489	M=6465	P=3867	M=3787	P=3743	M=3892
ITE=10	P=5882	M=5833	P=6431	M=6393	P=4005	M=3940	P=3885	M=4037
ITE=11	P=5800	M=5757	P=6163	M=6117	P=4115	M=4053	P=3997	M=4148
ITE=12	P=6042	M=5988	P=6288	M=6246	P=4206	M=4148	P=4091	M=4245
ITE=13	P=5549	M=5511	P=6107	M=6036	P=4299	M=4257	P=4191	M=4342
ITE=14	P=5451	M=5422	P=6108	M=6031	P=4366	M=4326	P=4263	M=4409
ITE=15	P=5305	M=5277	P=6128	M=6047	P=4421	M=4386	P=4322	M=4466
ITE=16	P=5564	M=5518	P=6182	M=6119	P=4465	M=4432	P=4368	M=4510
ITE=17	P=5358	M=5313	P=6248	M=6178	P=4515	M=4488	P=4426	M=4568
ITE=18	P=5457	M=5397	P=6069	M=6017	P=4553	M=4529	P=4473	M=4609
ITE=19	P=5313	M=5257	P=6292	M=6244	P=4600	M=4574	P=4528	M=4658
ITE=20	P=5349	M=5297	P=6218	M=6173	P=4640	M=4614	P=4579	M=4702
ITE=21	P=5358	M=5304	P=6164	M=6130	P=4678	M=4647	P=4623	M=4742
ITE=22	P=5414	M=5360	P=6486	M=6465	P=4708	M=4672	P=4660	M=4771
ITE=23	P=4798	M=4746	P=6276	M=6231	P=4640	M=4636	P=4688	M=4789
ITE=24	P=5064	M=5018	P=6135	M=6107	P=4709	M=4687	P=4713	M=4809
ITE=25	P=5146	M=5098	P=6040	M=6002	P=4650	M=4649	P=4736	M=4823
ITE=26	P=5091	M=5043	P=6091	M=6061	P=4576	M=4575	P=4757	M=4836
ITE=27	P=5038	M=4991	P=6165	M=6156	P=4722	M=4703	P=4777	M=4852
ITE=28	P=5461	M=5424	P=6182	M=6171	P=4736	M=4717	P=4797	M=4864
ITE=29	P=5029	M=4999	P=6335	M=6319	P=4753	M=4733	P=4821	M=4887
ITE=30	P=4850	M=4811	P=5962	M=5941	P=4649	M=4647	P=4833	M=4895
ITE=31	P=5548	M=5525	P=5719	M=5710	P=4778	M=4753	P=4862	M=4919
ITE=32	P=5229	M=5207	P=6407	M=6402	P=4798	M=4772	P=4890	M=4943
ITE=33	P=5284	M=5272	P=6045	M=6028	P=4803	M=4779	P=4898	M=4951
ITE=34	P=5001	M=4978	P=5935	M=5913	P=4717	M=4717	P=4906	M=4951
ITE=35	P=4941	M=4923	P=6062	M=6049	P=4794	M=4785	P=4912	M=4952
ITE=36	P=4954	M=4947	P=5952	M=5936	P=4808	M=4791	P=4920	M=4960
ITE=37	P=5144	M=5133	P=6132	M=6130	P=4827	M=4807	P=4941	M=4974
ITE=38	P=4892	M=4884	P=5911	M=5917	P=4844	M=4820	P=4958	M=4984
ITE=39	P=4891	M=4890	P=6000	M=6011	P=4861	M=4831	P=4974	M=4994
ITE=40	P=4988	M=4994	P=5920	M=5923	P=4863	M=4832	P=4975	M=4997
ITE=41	P=5112	M=5121	P=5993	M=6002	P=4873	M=4836	P=4982	M=5001
ITE=42	P=5096	M=5114	P=6159	M=6171	P=4887	M=4843	P=4993	M=5007
ITE=43	P=5035	M=5024	P=6029	M=6027	P=4842	M=4841	P=4993	M=5007
ITE=44	P=5211	M=5220	P=6184	M=6211	P=4895	M=4850	P=5001	M=5010
ITE=45	P=4833	M=4830	P=6091	M=6095	P=4863	M=4850	P=5000	M=5012
ITE=46	P=4819	M=4808	P=6043	M=6042	P=4676	M=4676	P=5000	M=5012
ITE=47	P=5093	M=5082	P=5935	M=5934	P=4642	M=4642	P=5000	M=5012
ITE=48	P=5101	M=5090	P=6173	M=6172	P=4767	M=4767	P=5000	M=5012
ITE=49	P=4893	M=4882	P=6242	M=6241	P=4827	M=4827	P=5000	M=5012
ITE=50	P=4927	M=4942	P=6207	M=6241	P=4906	M=4860	P=4958	M=4955

Table 6.5 The number of points distributed

Table 6.6 shows the number of points sampled in each iteration and the number of points been paired in the iteration, together with the fermion energy.

O	PIRS/P0	P1	P2	P3	ENERGY	TIME (iteration starts)
I=1	7855	7058	5970	5681	0.000000	Sun Jun 26 19:21:00
I=2	7443	4608	2789	3252	-57.475169	Sun Jun 26 20:06:17
I=3	3941	2398	2181	2691	-120.044092	Sun Jun 26 20:40:35
I=4	2726	1493	1531	1840	-69.530752	Sun Jun 26 21:15:09
I=5	2108	1095	1246	1557	-89.498525	Sun Jun 26 21:46:28
I=6	1768	961	900	1295	-87.125134	Sun Jun 26 22:18:06
I=7	987	1373	431	1169	-85.670325	Sun Jun 26 22:46:58
I=8	1635	834	639	1023	-80.762345	Sun Jun 26 23:17:26
I=9	1340	771	565	974	-79.332700	Sun Jun 26 23:49:58
I=10	1328	690	470	941	-76.138886	Mon Jun 27 00:23:27
I=11	1276	614	396	861	-75.783794	Mon Jun 27 00:59:25
I=12	1128	586	424	773	-74.619634	Mon Jun 27 01:39:32
I=13	1040	591	390	703	-74.839148	Mon Jun 27 02:25:14
I=14	1075	555	347	641	-74.842662	Mon Jun 27 03:12:19
I=15	1123	564	337	665	-74.598842	Mon Jun 27 03:58:25
I=16	1086	603	342	631	-73.957906	Mon Jun 27 04:54:13
I=17	1066	523	358	578	-74.339663	Mon Jun 27 05:53:16
I=18	906	518	331	538	-74.228851	Mon Jun 27 06:57:56
I=19	952	479	267	534	-74.085554	Mon Jun 27 08:05:39
I=20	865	485	324	447	-74.080538	Mon Jun 27 09:16:40
I=21	961	453	376	424	-74.175482	Mon Jun 27 10:38:29
I=22	782	452	325	399	-74.366317	Mon Jun 27 11:52:03
I=23	751	463	318	443	-74.475669	Mon Jun 27 12:59:44
I=24	809	456	294	427	-74.710150	Mon Jun 27 14:15:21
I=25	814	458	295	462	-74.795492	Mon Jun 27 15:31:12
I=26	692	427	253	462	-74.969418	Mon Jun 27 16:41:03
I=27	719	461	258	458	-74.924713	Mon Jun 27 17:53:34
I=28	649	436	243	417	-74.619277	Mon Jun 27 19:15:02
I=29	666	395	246	395	-74.355571	Mon Jun 27 20:33:02
I=30	643	338	262	406	-74.052328	Mon Jun 27 21:55:14
I=31	653	342	249	388	-73.673364	Mon Jun 27 23:31:16
I=32	652	359	246	368	-73.386538	Tue Jun 28 01:05:19
I=33	572	351	228	390	-73.562087	Tue Jun 28 02:29:56
I=34	633	306	214	355	-73.584604	Tue Jun 28 04:08:06
I=35	532	300	233	323	-73.712750	Tue Jun 28 05:40:53
I=36	623	281	180	267	-73.704401	Tue Jun 28 07:17:56
I=37	515	317	246	268	-73.481020	Tue Jun 28 08:56:59
I=38	450	295	202	326	-73.571351	Tue Jun 28 10:25:34
I=39	463	245	176	271	-73.349837	Tue Jun 28 11:49:46
I=40	477	264	184	276	-73.311675	Tue Jun 28 13:16:42
I=41	470	279	236	253	-73.358955	Tue Jun 28 14:51:24
I=42	435	259	158	242	-73.268012	Tue Jun 28 16:30:21
I=43	443	273	188	261	-73.255105	Tue Jun 28 18:17:03
I=44	523	245	168	249	-73.091262	Tue Jun 28 20:07:34
I=45	516	244	175	219	-72.891615	Tue Jun 28 21:53:29
I=46	467	265	140	224	-72.868275	Tue Jun 28 23:36:10
I=47	407	221	178	218	-72.861905	Wed Jun 29 01:21:10
I=48	464	270	151	221	-72.926127	Wed Jun 29 03:08:30
I=49	426	282	215	190	-72.888660	Wed Jun 29 05:06:17
I=50	417	264	170	222	-72.825375	Wed Jun 29 07:06:10

Table 6.6 The fermion energy

Table 6.7 shows estimated energy from all 116 iterations of a run. It runs 100 iterations to converge the fermion state wave function, and then estimates the fermion energy.

I=1	E=0.000000	I=41	E=-73.358955	I=81	E=-72.146970
I=2	E=-57.475169	I=42	E=-73.268012	I=82	E=-72.144810
I=3	E=-120.044092	I=43	E=-73.255105	I=83	E=-72.142649
I=4	E=-69.530752	I=44	E=-73.091262	I=84	E=-72.135420
I=5	E=-89.498525	I=45	E=-72.891615	I=85	E=-72.158231
I=6	E=-87.125134	I=46	E=-72.868275	I=86	E=-72.136861
I=7	E=-85.670325	I=47	E=-72.861905	I=87	E=-72.180958
I=8	E=-80.762345	I=48	E=-72.926127	I=88	E=-72.195336
I=9	E=-79.332700	I=49	E=-72.888660	I=89	E=-72.228533
I=10	E=-76.138886	I=50	E=-72.825375	I=90	E=-72.193777
I=11	E=-75.783794	I=51	E=-72.792097	I=91	E=-72.174485
I=12	E=-74.619634	I=52	E=-72.713474	I=92	E=-72.150745
I=13	E=-74.839148	I=53	E=-72.629251	I=93	E=-72.148480
I=14	E=-74.842662	I=54	E=-72.660797	I=94	E=-72.145672
I=15	E=-74.598842	I=55	E=-72.687669	I=95	E=-72.124780
I=16	E=-73.957906	I=56	E=-72.611208	I=96	E=-72.160998
I=17	E=-74.339663	I=57	E=-72.643281	I=97	E=-72.160027
I=18	E=-74.228851	I=58	E=-72.669128	I=98	E=-72.131681
I=19	E=-74.085554	I=59	E=-72.561554	I=99	E=-72.149359
I=20	E=-74.080538	I=60	E=-72.498599	ENERGY EVALUATION	
I=21	E=-74.175482	I=61	E=-72.467645	I=100	E=-70.354406
I=22	E=-74.366317	I=62	E=-72.482134	I=101	E=-72.528891
I=23	E=-74.475669	I=63	E=-72.433337	I=102	E=-71.877274
I=24	E=-74.710150	I=64	E=-72.479868	I=103	E=-71.547128
I=25	E=-74.795492	I=65	E=-72.457173	I=104	E=-71.944225
I=26	E=-74.969418	I=66	E=-72.454961	I=105	E=-72.124445
I=27	E=-74.924713	I=67	E=-72.417712	I=106	E=-72.393522
I=28	E=-74.619277	I=68	E=-72.437345	I=107	E=-72.565537
I=29	E=-74.355571	I=69	E=-72.372798	I=108	E=-72.417300
I=30	E=-74.052328	I=70	E=-72.401105	I=109	E=-72.346171
I=31	E=-73.673364	I=71	E=-72.387149	I=110	E=-72.884066
I=32	E=-73.386538	I=72	E=-72.356771	I=111	E=-72.670652
I=33	E=-73.562087	I=73	E=-72.315877	I=112	E=-72.732734
I=34	E=-73.584604	I=74	E=-72.285600	I=113	E=-72.766169
I=35	E=-73.712750	I=75	E=-72.219078	I=114	E=-72.883464
I=36	E=-73.704401	I=76	E=-72.182393	I=115	E=-72.780912
I=37	E=-73.481020	I=77	E=-72.140524	I=116	E=-72.747271
I=38	E=-73.571351	I=78	E=-72.127037		
I=39	E=-73.349837	I=79	E=-72.101243		
I=40	E=-73.311675	I=80	E=-72.108193		

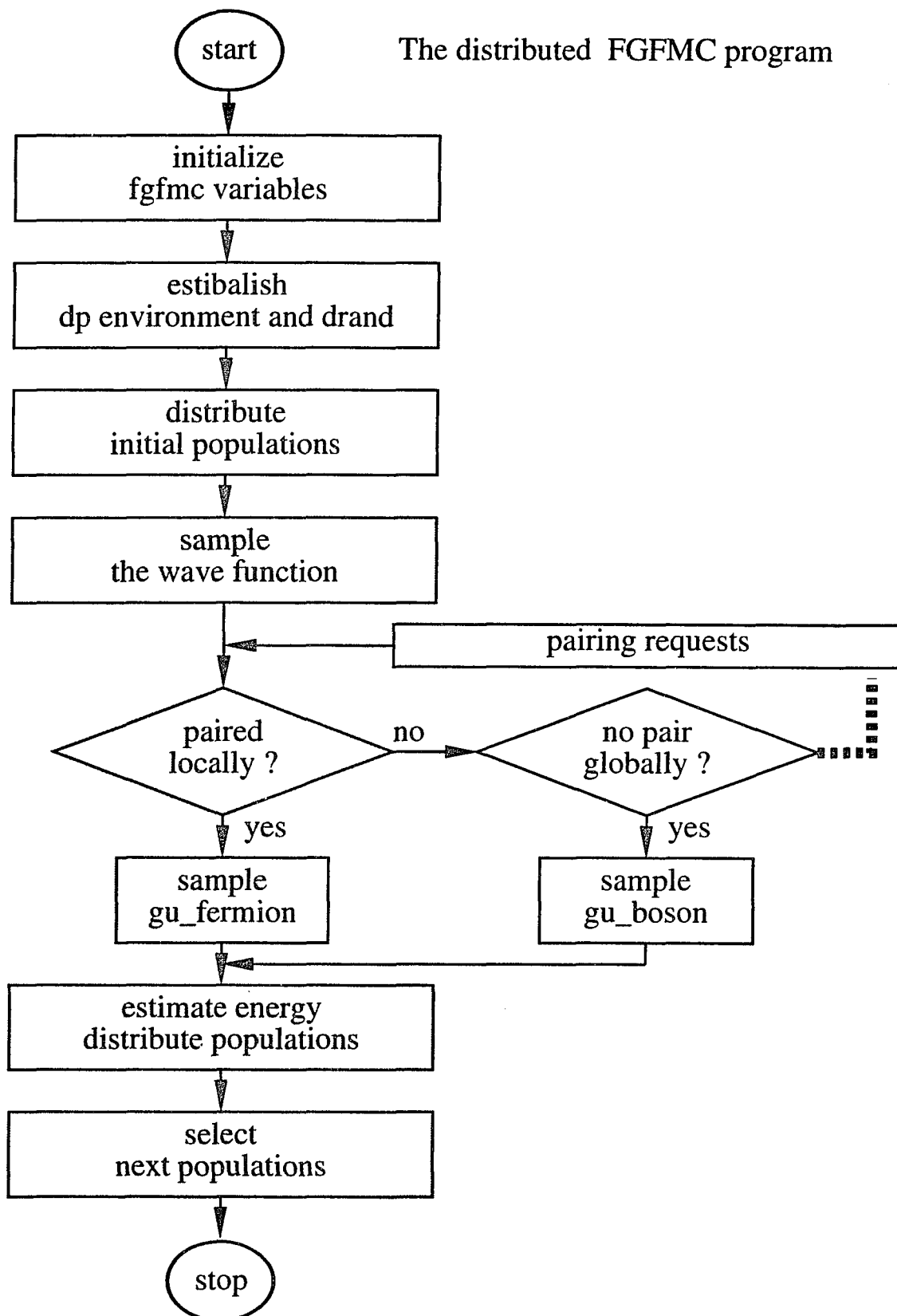
Table 6.7 The complete energy result

6.4.5* the FGFMC program

In this section, the distributed fermion Green's function Monte Carlo program is discussed. The program uses the latest version of the dp library. Again, only the code that is involving distributing simulation is presented. Figure 6.3 shows the overall structure of the program. The program starts by a process, the primary process. It first initializes FGFMC related variables, such as the constants for the system potential. Then the primary process spawns distributed processes, the secondary processes, and at the same time assigns distributed random number generators to the newly created processes. When all work processes enter the working stage, they first initialize their local populations, then sample the wave function. The shade area indicates a loop which runs a desired number of time.

Each process runs the loop to sample the wave function: either sample the Green's function for boson points or the Green's function for fermion points. It works mainly for the local points, but takes care of pairing requests from foreign processes as well. After the sampling stage, each secondary sends the information about their local populations to the primary to have energy estimated. The last stage of the loop is to select populations for the next iteration.

The distributed FGPMC program



The code corresponding to the first box, *initialize fgfmc variables*, in Fig. 6.3 is omitted. Code fragment 6.1 initializes dp environments and drand generators. `sdpinit()` is called in line 11. When it returns to the primary process, the first process in the distributed computation, the last parameter, `size`, has a negative value and the second parameter, `s`, has no meaning. `nhosts` indicates the number of processes in the computation, including the primary process itself. When `sdpinit()` returns to a secondary process, `size` is non-negative and `s` is a record sent by the primary process with pre-defined meaning. In this program, `s` passes a random number generator. Lines 12 to 16 initialize `dppid`, `dplog` for the computation uses. Lines 19 to 28 are used by the primary process to create a system of distributed random number generators and sequentially to spawn secondary processes. Notice that in line 22 `drand_encode` returns a pointer to the encoded handle, `childrand`. The pointer can not be passed directly to `sdpspawn()`, so the encoded handle is literal copied to a semantic structure described in line 7. Each process spawned by the primary process in line 24 will run the same code starting from line 1. These processes will jump into line 29 from line 18 to receive their random number generators.

```

1  int    i, nhosts, size, dppid, dpnnodes;
2  char   filename[];
3  FILE   *dplog;
4  drandle myrand, childrand;
5  codetype *h;
6  /*
7   *  s.s_drand -- semantic structure
8   *  MACPROC   -- the number of processes
9   *  ac, av    -- main's parameters
10  */
11  nhosts = dpnnodes = sdpinit( av[0], &s, &size );

```

```

12 dppid = sdpgetpid();
13 sprintf(filename, "dplog%d", (int) dppid);
14 dplog = fopen( filename, "w");
15
16 setbuf(dplog, (char *)0);
17
18 if ( size < 0 ) {
19     (void) drand_creat(ac, av, &myrand);
20     for ( i = 1; i < dpnnodes; i++ ) {
21         (void) drand_fork(&myrand, &childrand);
22         (void) drand_encode(&h, childrand);
23         strcpy( (s.s_drand), h );
24         if ( SDPNOERR != sdpspawn( s, sizeof(s) ) ) {
25             fprintf(dplog, "\ncan't sdpspawn i=%d \n", i
26 );
27             exit(1);
28         }
29     } else (void) drand_decode(&myrand, s.s_drand);
30 }

```

Code fragment 6.1 Initialize dp and drand

After initializing local populations, each process distributes the points that should be stored in foreign processes. Code fragment 6.2 shows the code counting the number of messages which will be used to carry the points to other processes.

```

1  int i, j, k, hostsz[], outtab[], intab[];
2  /* count points in population popu[] */
3  for ( i = 0; i < dpnnodes; i++ )
4      hostsz[i] = 0;
5  for ( i = 0; i < *psize; i++ ) {
6      for ( j = 0; j < dpnnodes; j++ )
7          if ( popu[i].index < bounds[j][1] ) {
8              hostsz[j] += 1;
9              break;
10         }
11     }
12     /* compute the number messages to send */
13     for ( i = 0; i < dpnnodes; i++ )
14         outtab[i] = ( hostsz[i] / NUMPOINTS ) + (
15             (hostsz[i] % NUMPOINTS > 0) ? 1 : 0);
16

```

Code fragment 6.2 Compute the number of messages before sending them

Code fragment 6.3 illustrates how the messages are exchanged among all worker processes. In lines 2 to 8 each process sends a message to all other processes telling the number of data messages that will be delivered to them. Then it receives a message from each and every other processes which indicates the number of data messages that shall be received in lines 10 to 14. It is followed by three stages of data transmission: lines 16 to 18 are used to send all data out; then each process rearrange its local memory by using the points not been sent; finally, in lines 22 to 27 each process receives the expected number of messages and stores the received points in its local memory.

```

1
2 for ( i = 0; i < dpnnodes; i++ ) {
3     if ( i == dppid )
4         continue;
5     pck.long1 = outtab[i];
6     (void) sdpsend((char *) &pck, MINIMUM, SDPREL,
7                   i, TYPEdmsg);
8 }
9
10 for ( i = 1; i < dpnnodes; i++ ) {
11     rtn = sdprecv((char *) &pck, MINIMUM, SDPBLOCK,
12                  SDPANYSRC, TYPEdmsg, SDPNOSRC);
13     intab += pck.long1;
14 }
15
16 for ( i = 0; i < dpnnodes; i++ )
17     if ( dppid != i )
18         sendover(popu, psize, i, ite);
19
20 initnew(popu, &psize);
21
22 while ( intab ) {
23     rtn = sdprecv((char *) &mcpck, MINIMUM, SDPBLOCK,
24                  SDPANYSRC, TYPEbmsg, SDPNOSRC);
25     intab -= 1;
26     appendtosampt(popu, &psize, mcpck, ite);
27 }
28

```


Code fragment 6.3 Prepare message distribution and receive messages

Function `initnew()` and `appendtosampt()` used in code fragment 6.3 are omitted and `sendover()` is listed below to show how `sdpsend()` is employed to deliver data messages.

```

1
2  sendover(popu, scanover, pid, ite)
3  {
4      int  ps = NUMPOINTS, pno = 0, nnew;
5      ...
6      nnew = hostsz[pid];
7      while ( nnew > 0 ) {
8          if ( nnew > ps ) {
9              mcpck.totpts = ps;
10             ...
11             fill ps number of points into mcpck
12             ...
13         } else {
14             mcpck.totpts = nnew;
15             ...
16             fill nnew number of points into mcpck
17             ...
18         }
19
20         nnew -= ps;
21         pno += 1;
22         mcpck.iteno = ite;
23         mcpck.pckno = pno;
24         (void) sdpsend((char *) &mcpck, MINIMUM, SDPREL,
25                        pid, TYPEbmsg);
26     }
27     ...
28 }
29

```

Code fragment 6.4 Send messages to other processes

The actual number of points sent by each process is listed in table 6.1. And the number of points stored in each local process after the distribution is shown in table 6.5. The code fragments discussed above correspond to the third box, *distribute initial populations*, in Fig. 6.3.

The sampling is started by the code fragment 6.5. Line 9 is used to prepare the hash tables before constructing linked-lists to the tables. Lines 10 to 16 is part of the code to choose points for the current iteration from the positive population of the previous iteration. The counterpart for the negative population is omitted. Line 14 describes that the selected points are inserted into the positive hash table, `htabplus`. Line 18 flushes the arrays, `rnewp` and `rnewm` which are used to store the sampled points in the previous iteration.

```

1
2  sam_wf( rnewp, nrnewp, rnewm, nrnewm, ite )
3  {
4      int      i, j, k, l, stpunblck = 0;
5      point    *eplus, *eminus;
6      double   rand;
7      extern int STOPUNBLOCK;
8
9      free_hashes();
10     for ( i = 0; i < PNGNSIZ; i++ ) {
11         rand = drand_random(&myrand, 0);
12         j = (int) ( rand * (*nrnewp) );
13         ...
14         copy the jth point in rnewp to htabplus
15         ...
16     }
17     ...
18     free_rr( rnewp, nrnewp, rnewm, nrnewm );

```

Code fragment 6.5 Sample the wave function

The major part of code for sampling is listed in code fragment 6.6. Lines 19 to 22 count the number of points linked in each local hash index, and then distribute the numbers. The numbers will be used to guide migrating points searching for matches. Lines 25 to 43 construct a loop sampling the wave function using the points in hash tables. When the loop starts at line 25, unblocked receiving operations are

employed to unqueue messages if any is arrived. When there is no point left in both local hash tables, each local process sends a message to all other processes to tell remote processes not sending pairing requests. Before moving to the next stage of the computation, each local process goes back to line 24 to take care pending pairing requests.

```

19     count_hash( htabplus, 'p' );
20     count_hash( htabminus, 'm' );
21     dist_hash( 'p' );
22     dist_hash( 'm' );
23     STOPUNBLOCK = 0;
24 again:
25     do {
26         retri_unblk( rnewp, nrnewp, rnewm, nrnewm, ite );
27         if ( (eplus = get_pointfromhash( htabplus ))
28             != LISTEND ) {
29             eplus->travel = 1 << dppid;
30             g_ferm( eplus, rnewp, nrnewp, rnewm, nrnewm,
31                   htabplus, htabminus, 'p', ite );
32             free_point( &eplus );
33             eplus = (point *) 1;
34         }
35         if ( (eminus = get_pointfromhash( htabminus ))
36             != LISTEND ) {
37             eminus->travel = 1 << dppid;
38             g_ferm( eminus, rnewm, nrnewm, rnewp, nrnewp,
39                   htabminus, htabplus, 'm', ite );
40             free_point( &eminus );
41             eminus = (point *) 1;
42         }
43     } while ( eplus || eminus );
44     if ( !stpunblk ) {
45         stpunblk = stop_unblk(ite);
46         STOPUNBLOCK = 1;
47         goto again;
48     }
49 }
50

```

Code fragment 6.6 cont'd Sample the wave function

After lines 19 and 20 in code fragment 6.6, the indices to be distributed are stored in `hcountp[]`, `hcountm[]` and code fragment 6.7 is used to distributed them.

```

1
2  dist_hash( c )
3  char    c;
4  {
5      int    i, j, k, begin, end;
6      extern hcountp[], hcountm[];
7      /*
8       * HPCKSIZ  -- the num.  of entries per pck
9       * TYPEhmsg -- a message type, user defined
10      */
11     begin = j = bounds[dppid][0];
12     do {
13         /* fill package */
14         for ( k = 0; k < HPCKSIZ; k++ ) {
15             if ( c == 'p' )
16                 miscpck.un.hc.hashct[k] = hcountp[j];
17             else
18                 miscpck.un.hc.hashct[k] = hcountm[j];
19             j++;
20             if ( j > bounds[dppid][1] )
21                 break;
22         }
23         miscpck.un.hc.from = begin;
24         miscpck.un.hc.to = j-1;
25         begin = j;
26         /* distributed hash tables */
27         for ( i = 0 ; i < dpnnodes; i++ ) {
28             if ( i == dppid )
29                 continue;
30             (void) sdpsend((char *) &miscpck,
31                 sizeof(miscpck), SDPREL, i, TYPEhmsg);
32         }
33     } while ( j < bounds[dppid][1] );

```

Code fragment 6.7 Distribute hash tables

The following code fragment is used to receive the distributed hash indices.

```

34  /* receive hash tables */
35  for (i=0;i<interval/HPCKSIZ+((interval%HPCKSIZ>0)
36      ?1:0)*(dpnnodes-1); i++ ) {
37      rtrn = sdprecv((char *) &miscpck,

```

```

38             sizeof(miscpck), SDPBLOCK, SDPANYSRC,
39             TYPEhmsg, SDPNOSRC);
40     k = 0;
41     for ( j = miscpck.un.hc.from;
42          j <= miscpck.un.hc.to; j++ ) {
43         if ( c == 'p' )
44             hcountp[j] = miscpck.un.hc.hashct[k];
45         else
46             hcountm[j] = miscpck.un.hc.hashct[k];
47         k++;
48     }
49 }
50 }

```

Code fragment 6.8 Receive distributed hash tables

Table 6.2 shows the distributed indices in the first five iterations. The following function is called in line 25 in code fragment 6.6 to retrieve unblocked messages. In line 9 in code fragment 6.9, `sdprecv()` is called with `SDPNOBLOCK` option. If a message is received, lines 11 to 29 take care of all points packed in the message. The unblocked receive operation is repeatedly executed by line 28 until all `PSEUDOASYN` type messages are unqueued.

```

1
2  retri_unblk( rnewp, nrnewp, rnewm, nrnewm, ite )
3  {
4      DPresult    *rtrn;
5      point       *p;
6      int         l, k;
7      extern int  RECEIVE;
8      tryonemore:
9      rtrn = sdprecv((char *) &mcpcck, sizeof(MCPCCK),
10                  SDPNOBLOCK, SDPANYSRC, PSEUDOASYN, SDPNOSRC);
11      if ( rtrn->r_src != SDPNOSRC && (mcpcck.iteno == ite) ) {
12          RECEIVE++;
13          for ( l = 0; l < mcpcck.totpts; l++ ) {
14              ...
15              copy mcpcck.ddata[l] to p, including
16              p->travel = mcpcck.ddata[l].travel;
17              ...
18              if (mcpcck.pckno > 0) {

```

```

19          ...
20          sample g_ferm using p against minus sign
21          ...
22      } else {
23          ...
24          sample g_ferm using p against plus sign
25          ...
26      }
27  }
28      goto tryonemore;
29  }
30  }
31

```

Code fragment 6.9 Receive pairing requests

Function `stop_unblk()`, which is called in line 46 in code fragment 6.6, is listed below. When the function returns, no pairing requests will be delivered to the calling process.

```

1
2  stop_unblk(ite)
3  {
4      int i;
5      SYNC sync;
6      sync.int1 = ite;
7      for ( i = 0; i < dpnnodes; i++ ) {
8          if ( i == dppid )
9              continue;
10         (void) sdpsend((char *)&sync, sizeof(SYNC),
11                        SDPREL, i, TYPEnmsg);
12     }
13     for ( i = 1; i < dpnnodes; i++ ) {
14         rtrn = sdprecv((char *) &sync, MINIMUM,
15                      SDPBLOCK, SDPANYSRC, TYPEnmsg, SDPNOSRC);
16         ...
17     }
18     return( 1 );
19 }
20

```

Code fragment 6.10 Stop receiving pairing requests

Function `g_ferm()` called in lines 29, 38 in code fragment 6.6 and in lines 20, 24 in code fragment 6.9 is given below. It determines which Green's function is to sample for a given point: if a match is made locally, the fermion Green's function is sampled; if no a pair can be made both locally and globally, the boson Green's function is sampled; otherwise, the point is sent using the knowledge of foreign hash tables.

```

1
2  g_ferm( r, rnew, nrnew, rnewf, nrnewf, hashus,
3          hashoppo, c, ite )
3  {
4      double    sd
5      point     *rtrn;
6
7      sd = get_pair(r->data, hashoppo,  &p_clist,
8                                     &p_plist);
9
10     if ( sd > 1.0 ) {
11         ...
12         if ( !hit_send( r, c, ite ) )
13             sam_gu(r->data, rnew, nrnew, rnewf, nrnewf,
14                   hashus, hashoppo);
15     }
16     else {
17         rtrn = remove_point(p_clist, p_plist );
18         sam_guferm(r->data, rtrn->data, rnew, nrnew,
19                  rnewf, nrnewf, hashus, hashoppo);
20         ...
21     }
22 }
23

```

Code fragment 6.11 Decide which Green's function to sample

The function receives a point `r` and it tries to make a match with a point in the opposite hash table, `hashoppo`. If `get_pair()` in line 11 returns a value less than 1.0, then a match is made, therefore, fermion Green's function is sampled; Otherwise, line 12 calls `hit_send()` to search a possible process by which a pair

may be made. If no such a possible can be found, then the boson Green's function is sampled. The code fragment 6.12 shows how the possible process can be found so that the point is sent to the process. Before sending a point, a history bit indicating the process been visited is set. Line 6 checks if pairing requests are permitted. Line 10 sets the bit indicating that the current process has been visited. If the condition in line 13 is true, all processes have reached and no a pair can be made. Line 18 describes that the distributed hash table is searched for a possible routing. The matching requests are not immediately sent until five of them are grouped in a message, which is sent by line 27. Table 6.4 lists the number of unblocked messages received and sent by all processes.

```

1
2  hit_send( R, c, ite )
3  {
4      ...
5      /* if the flag set, no more global pairing req. */
6      if ( STOPUNBLOCK )
7          return( 0 );
8
9      /* update R's travel history */
10     R->travel = R->travel | 1 << dppid;
11
12     /* if traveled all processes, sample gu_boson */
13     if (R->travel == (short unsigned int) pow(2.0,
14                                                (double) dpnnodes)-1)
15         return( 0 );
16     ...
17     /*
18      * if no global pairing attempts can be made,
19      *     return (0);
20      * else collect R into a packet.
21      */
22     ...
23     /*
24      * if five requests accumulated to process j,
25      *     send the packet
26      */

```



```

27         (void) sdpsend((char *) &(mcpckp[j]),
28             sizeof(MCPCK), SDPREL, j, PSEUDOASYN);
29         SEND++;
30         ...
31     return ( 1 );
32 }
33

```

Code fragment 6.12 Route a pairing request

The code fragments from 6.5 to 6.12 cover the first part of the computations in the shade area in Fig. 6.3. The second part of the computation is to estimate the system energy and to select points for the next iteration.

Each secondary reports its contribution to the primary process by the following code fragment:

```

1
2  /*
3   * each secondary process sends:
4   * alpha[] -- double, FGFMC determined
5   * Nin, Nout -- int, FGFMC determined
6   */
7
8  miscpck.un.ev.al[0] = alpha[0];
9  miscpck.un.ev.al[1] = alpha[1];
10 miscpck.un.ev.al[2] = alpha[2];
11 miscpck.un.ev.nin = Nin;
12 miscpck.un.ev.nout = Nout;
13
14 (void) sdpsend((char *) &miscpck, sizeof(miscpck),
15             SDPREL, 0, TYPEdmsg);
16 Nin = Nout = 0;
17

```

Code fragment 6.13 Partial energy contribution

$\alpha[]$ and N_{in} , N_{out} are fermion Green's function Monte Carlo determined, and N_{in} , N_{out} will be used to estimate fermion state energy by the primary process.

The following code fragment is run by the primary process to collect the partial results and to estimate the energy.

```

1
2  /* the primary receives
3   *
4   * alpha[], energy1, energy2  -- double
5   * NNin, Nout, Nin, Nout Nrtop, Nrbot -- int
6   * ETrial, V0 -- constant to FGPMC
7   *
8   * Nin, Nout -- determined locally
9   * NNin, NNout, Nrtop, Nrbot -- accumulative
10  */
11
12  NNin = Nin;
13  NNout = Nout;
14  for ( i = 1; i < dpnnodes; i++ ) {
15      rtn = sdprecv((char *) &miscpck, sizeof(miscpck),
16                  SDPBLOCK, SDPANYSRC, TYPEdmsg, SDPNOSRC);
17      alpha[0] += miscpck.un.ev.al[0];
18      alpha[1] += miscpck.un.ev.al[1];
19      alpha[2] += miscpck.un.ev.al[2];
20      NNin += miscpck.un.ev.nin;
21      NNout += miscpck.un.ev.nout;
22  }
23
24  Nrtop = Nrtop + NNin;
25  Nrbot = Nrbot + NNout;
26  /* accumulative FGPMC energy */
27  energy1 = ( ETrial + V0 ) * Nrtop / Nrbot - V0;
28  /* FGPMC energy obtained in the current iteration */
29  energy2 = ( ETrial + V0 ) * NNin / NNout - V0;
30
31  Nin = 0;
32  Nout = 0;
33

```

Code fragment 6.14 Energy evaluation

The way to select point for the next generation is similar to the solution discussed in chapter 5. The only difference is that in this case two populations have to deal with. The following code is what the secondaries execute. They send the

population sizes to the primary process, then receive the numbers of points needed to choose from their local memory to form the next generation.

```

1
2  rcvngnsiz( nrplus, nrminus, pngnsiz, mngnsiz, ite )
3  {
4      SYNC    sync;
5      ...
6      /* report sizes of populations to the primary */
7      sync.int1 = nrplus;
8      sync.int2 = nrminus;
9      sync.long1 = ite;
10     ...
11     (void) sdpsend((char *) &sync, MINIMUM, SDPREL, 0,
12                      TYPEsmg);
13     ...
14     /*
15      * receive the number of points
16      * to choose from current populations
17      */
18     rtn = sdprecv((char *) &sync, MINIMUM, SDPBLOCK, 0,
19                      TYPEsmg, SDPNOSRC);
20     ...
21     *pngnsiz = sync.int1;
22     *mngnsiz = sync.int2;
23 }
24

```

Code fragment 6.15 Choose points for next generation

The primary process runs code fragment 6.16 to assign the numbers of points to choose in local memory of all secondary processes.

```

1
2  sndngnsiz(nrplus, nrminus, pngnsiz, mngnsiz, ite)
3  {
4      SYNC    sync;
5      int     i, k;
6      double  locran, prodp, prodm;
7      int     NGNP[MAXPROC], NGNM[MAXPROC], NNEWP, NNEWM;
8      /* NGNSIZ fixed value */
9      for ( i = 1; i < dpnnodes; i++ ) {
10         rtn = sdprecv((char *) &sync, MINIMUM,
11                      SDPBLOCK, i, TYPEsmg, SDPNOSRC);
12         NGNP[i] = sync.int1;      /*      NNEWPs      */

```

```

13         NGNM[i] = sync.int2;      /*      NNEWMs      */
14         if (sync.long1 != ite)
15             fprintf(dplog,"NOTICE: ite not nomatch\n");
16     }
17     NGNP[0] = NNEWP = nrplus;
18     NGNM[0] = NNEWM = nrminus;
19     for ( i = 1; i < dpnnodes; i++ ) {
20         NNEWP += NGNP[i];
21         NNEWM += NGNM[i];
22     }
23     ngncpt = ngncmt = 0;
24     for (i = 0; i < dpnnodes; i++ ) {
25         prodp = (double) NGNP[i] * (dpnnodes * NGNSIZ);
26         prodm = (double) NGNM[i] * (dpnnodes * NGNSIZ);
27         locran = drand_random(&myrand, 0);
28         ngensp[i]=(int) ( prodp / NNEWP + locran);
29         ngnsml[i]=(int) ( prodm / NNEWM + locran);
30         ngncpt += ngensp[i];
31         ngncmt += ngnsml[i];
32     }
33     O_PNGNSIZ = ngncpt;
34     O_MNGNSIZ = ngncmt;
35
36     *pngnsiz = ngensp[0];
37     *mngnsiz = ngnsml[0];
38
39     sync.long1 = ite;
40     for ( i = 1; i < dpnnodes; i++ ) {
41         sync.int1 = ngensp[i];
42         sync.int2 = ngnsml[i];
43         (void) sdpsend((char *) &sync, MINIMUM,
44                     SDPREL, i, TYPEsmg);
45     }
46 }
47

```

Code fragment 6.16 Compute the sizes of next generation

The work load for all work processes is given in table 6.5. By the 50th iteration the work load is approaching 5000 points for each population, which is the average of the entire work load. Tables 6.6 and 6.7 shows the converged fermion state energy.

7

The conclusion

During developing this work, the goals described in chapter 1 are focused. The previous chapters completely documented the work: a solution of solving a system of fermions on the Distributed Process platform with applying a system of distributed pseudorandom number generators.

The concurrent Distributed Process system, or the desktop DP system, developed on one host is found very helpful. It provides an ensured messages system so that users can concentrate on developing application programs. The ability to reproduce parallel runs helps to analysis distributed programs. The similar system developed on the multiple processor KSR machines has good potential since it is constructed on shared memory. That model can be ported on many other machines that support shared memory and multiple threads, including Windows NT machines.

Two C++ abstract data types are developed that have ability to handle very large numbers. They are used to write two Spectra Test programs: the Knuth' Spectral test and the Percus & Kalos MIMD Spectral test. The ADT's are also used in the implementation of the distributed random number generator system to represent any binary tree.

A division method is constructed and it is used in the implementation of the ADT's mentioned above. Without the ability to divide large integers, implementing such ADT's is impossible. The characteristics of the method are: it is a high level

program language implementation, able to port any machine; The quotient of a division is approximated by group of decimal digits, one group at a time. Quotient may need to adjust determined by the method itself.

Since the Knuth' Spectral test program developed, test requests are accepted from outside. It will help to test linear congruential pseudorandom number generators of very long periods. The generators used in this work are also tested by the Spectral test program.

The Percus and Kalos Spectral test program is written in parallel for it needs massive scale computations. This program may be used for the purpose searching for the lower bound of a prime set. 10,000 generators in our database are tested by the parallel program.

A system of distributed random number generators is developed on a computer network. It supports multiple distributed or concurrent applications. Virtually complete binary trees are used to achieve better utilization of the available random number generators. The system satisfies a very large variety of simulations since it has ability to construct thousand levels of virtually complete binary trees. It also achieve the essential requirement of parallel simulation that it guarantees reproduce relationships between worker processes and random number generators so that random sequences used in simulations are reproducible. Manual pages are also written to help interesting users to explore the distributed generator system.

A one-dimensional Green's function Monte Carlo distributed program is developed. A general program scheme is constructed that is suitable for distributing

sequential GFMC. A successful distributed eigenvalue estimator is constructed. The performance is observed and satisfied by actual simulations. The problem of balancing work loads on distributed processes is solved by using a mathematical equation that is evaluated during the computation.

A nine-dimensional Fermion Green's Function Monte Carlo sequential program is re-constructed. It is successfully distributed on the Distributed Process platform. The problems solved in this program are: distributing two very large populations of random walkers and searching correlated walkers in distributed memory during the simulation. The eigenvalue estimator developed in the one-dimensional case is employed and the result it produced agrees with the sequential runs.

The Knuth Spectral Test Program**Description**

The program is used to test a multiplier for a linear congruential pseudorandom number generator. See K. Knuth *The Art of Computer Programming*.

```
#include "Kclass.h"

#define    min(x,y) ( (x > y) ? (y) : (x) )
#define    COMMENT(x)

#define    SIZ    10
static Fixed  U[SIZ][SIZ], V[SIZ][SIZ];
static  int t, j, k;
static  Fixed  s;

void transform(outstream , int );
void search( Fixed );

KnuthSpectral(outstream fp, Fixed a, Fixed m, int dim, int debug, Fixed nu[]) {
#define    KSGUBED    if (debug)

    Fixed  h, hprime, p, pprime, r;
    int jmp;
    Fixed  u, v, q;
    Fixed  Vij, Vjj;
    Fixed  Ujj, YY;
    int i;
    int T;
    Real r1;

    if ( dim > SIZ || dim < 0 ) {
        fp << "\nKnuthSpectral: error dim ? " << dim;
        return(-1);
    }

    t = 1;
    T = dim - 1;
    nu[0] = 0;

    // S1 [Initialize.]

    h = a;
    hprime = m;
    p = 1;
    pprime = 0;
```



```

r = a;
s = 1 + a*a;

// S2 [Euclidean step.]
do {
  q = Fixed( Rfloor( Real( hprime ) / Real( h ) ) );
  u = hprime - q * h;
  v = pprime - q * p;

  jmp = 0;
  if ( s > (u*u + v*v) ) {
    s = u*u+v*v;
    hprime = h;
    h = u;
    pprime = p;
    p = v;
    jmp = 1;
  }
} while (jmp);

// S3 [Compute v2]

u = u - h;
v = v - p;

if ( s > (u*u + v*v) ) {
  s = u*u + v*v;
  hprime = u;
  pprime = v;
}

U[0][0] = -h;
U[0][1] = p;
U[1][0] = -hprime;
U[1][1] = pprime;

if (pprime > 0) {
  V[0][0] = -pprime;
  V[0][1] = -hprime;
  V[1][0] = p;
  V[1][1] = h;
} else {
  V[0][0] = pprime;
  V[0][1] = hprime;
  if ( !( p > Fixed(0) ) ) {
    V[1][0] = p;
    V[1][1] = h;
  } else {
    V[1][0] = -p;
    V[1][1] = -h;
  }
}

```

```

// S4 [Advance t.]

while (1) {

    nu[t] = s ;
    if (t == T)
        return(0);

    t += 1;
    r = a * r % m;

    U[t][0] = -r;
    for (i = 1; i < t ; i++)
        U[t][i] = 0;
    U[t][t] = 1;

    for (i = 0; i < t ; i++)
        U[i][t] = 0;

    for (i = 0; i < t; i++)
        V[t][i] = 0;
    V[t][t] = m;

    for (i = 0; i < t; i++) {
        q = (Fixed) Rfloor( (Real) (V[i][0] * r) / (Real) m +
            (Real) 0.5 );

        V[i][t] = ( V[i][0] * r - q * m ) ;

        for (j = 0; j <= t; j++)
            U[t][j] = U[t][j] + q * U[i][j];
    }

    Ujj = 0;
    for (i = 0; i <= t; i++)
        Ujj = Ujj + U[t][i] * U[t][i];

    s = min(s, Ujj);

    k = t;
    j = 0;

    transform( fp, debug );

    search( m );

} // end while(1) S4

#undef KSGUBED
}
// end of KnuthSpectral()

```

```

void
transform(outstream fp, int debug) {
#define KSGUBED if (debug)
// S5 [Transform.]
Fixed Vjj, Vij, Ujj;
Fixed q;
Real r1;

do {

    Vjj = 0;
    for ( int ii = 0; ii<=t; ii++)
        Vjj = Vjj + V[j][ii] * V[j][ii];

    KSGUBED fp << "\n ( j= " << j+1;
    for ( int i = 0; i<=t; i++) {
        if (i != j) {
            KSGUBED fp << " q" << i << " = ";
            Vij = 0;
            for (ii = 0; ii<=t; ii++)
                Vij = Vij + V[i][ii] * V[j][ii];

            if ( Rabs(Vij) * 2 > Vjj ) {

                r1 = (Real) Vij / (Real) Vjj + (Real) 0.5;
                q = (Fixed) Rfloor( r1 );
                KSGUBED fp << q;
                for (ii = 0; ii <= t; ii++) {
                    V[i][ii] = V[i][ii] - q * V[j][ii];
                    U[j][ii] = U[j][ii] + q * U[i][ii];
                }

                k = j;
            } else KSGUBED fp << " 0 ";
            } else KSGUBED fp << " q" << j << " = * ";
        }
    KSGUBED fp << ")\n";

// S6 [Examine new bound.]
    Ujj = 0;
    for (ii = 0; ii <= t; ii++)
        Ujj = Ujj + U[j][ii]*U[j][ii];

    if (k == j)
        s = min(s,Ujj);
// S7 [Advance j.]

    if (j == t)
        j = 0;
    else
        j += 1;
}

```

```

    } while ( j != k );
#undef KSGUBED
    return;
}

void
search( Fixed m ) {
// S8 [Prepare for search.]
    Fixed Vjj, YY, X[SIZ], Y[SIZ], Z[SIZ];
    Real r1;
    int jmp;

    for ( int ii = 0; ii<=t; ii++)
        Y[ii] = X[ii] = Z[ii] = 0;
    k = t;

    for ( int jj = 0; jj<=t; jj++) {
        Vjj = 0;
        for (ii = 0; ii<=t; ii++)
            Vjj = Vjj + V[jj][ii] * V[jj][ii];

        r1 = Rsqrt( Rfloor( (Real)(Vjj*s)/(Real)(m*m) + (Real)0.5) );
        Z[jj] = (Fixed) Rfloor( r1 + (Real)0.5);
    }

// S9 [Advance x]
    while (1) {

        while ( X[k]==Z[k]) {
            k -= 1;
            if ( k < 0 )
                return;
        }

        X[k] = X[k] + 1;
        for ( ii = 0; ii <= t; ii++)
            Y[ii] = Y[ii] + U[k][ii];
// S10 [Advance k.]
        do {
            k += 1;
            jmp = 0;
            if ( k <= t ) {
                X[k] = -Z[k];
                for (ii = 0; ii <= t; ii++)
                    Y[ii] = Y[ii] - 2 * Z[k] * U[k][ii];
                jmp = 1;
            }
        } while (jmp);

        YY = 0;
        for ( ii = 0; ii <= t; ii++)

```

```
        YY = YY + Y[ii] * Y[ii];

    s = min(s, YY);
    k -= 1;
    if ( k < 0 )
        return;
    } // end of while(1) S9
}
#undef SIZ
```

Program Tests Thirty Generators**Description**

The program is used to reproduce the table given in, *The Art of Computer Programming*, p. 102-103.

```
#include "Kclass.h"

KnuthSpectral(outstream, Fixed, Fixed, int, int, Fixed *);

#define MSIZE 6
#define TAB 30
main()
{
    FILE *readin;
    outstream fp = stdout;
    int i, j;
    Fixed a, m[TAB];
    Fixed nu[TAB][MSIZE];
    Real tmp, pi = 3.1415926;

    COMMENT( "the content of table.dat is listed below " )
    if ( ( readin = fopen("table.dat", "r") ) == NULL ) {
        fprintf(stdout, "\ncan't open table.dat file !!! stop !!! ");
        exit(1);
    }
    instream fpi = readin;

    fp << "\nNo. multiplier modulo v2^2 v3^2 "; fp <<
    "v4^2 v5^2 v6^2\n";

    for ( i = 0; i < TAB; i++ ) {

        fpi >> &a ;
        fpi >> &m[i] ;

        if ( i == TAB - 3 ) {
            fp << "\nNo. multiplier ";
            fp << "modulo ";
            fp << "v2^2 v3^2 ";
            fp << "\nv4^2 ";
            fp << "v5^2 v6^2 ";
        }
        fprintf(stdout, "\n%3d ", i+1);

        if ( i < TAB - 3 ) {
            a.iwrite(stdout, 15);

```

```

        m[i].iwrite(stdout, 15);
    } else {
        a.iwrite(stdout, 22);
        m[i].iwrite(stdout, 22);
    }

    KnuthSpectral( fp, a, m[i], MSIZE, 0, nu[i] );

    if ( i < TAB - 3 )
        for ( j = 1; j < MSIZE; j++ )
            nu[i][j].iwrite(stdout, 14-2*j);
    else
        for ( j = 1; j < MSIZE; j++ ) {
            nu[i][j].iwrite(stdout, 20);
            if ( j == 2 )
                fp << "\n";
        }

    }

    fp << "\n\nNo. lgv2 lgv3 lgv4 lgv5 lgv6 ";
    fp << "u2 u3 u4 u5 u6\n";

    for ( i = 0; i < TAB; i++ ) {
        fprintf(stdout, "%3d ", i+1 );
        for ( j = 1; j < MSIZE; j++ ) {
            tmp = Rlog2( Rsqrt( Real(nu[i][j]) ) );
            tmp.rwrite(stdout, 7);
        }

        for ( j = 1; j < MSIZE; j++ ) {
            switch (j) {

                case 1: tmp = pi * nu[i][j] / m[i];
                    tmp.rwrite(stdout, 7);
                    break;
                case 2: tmp = 4.0/3.0*pi*Rpow(Real(nu[i][j]),Real(1.5))/m[i];
                    tmp.rwrite(stdout, 7);
                    break;
                case 3: tmp = 1.0/2.0*pi*pi*Rpow(Real(nu[i][j]),Real(2.0))/m[i];
                    tmp.rwrite(stdout, 7);
                    break;
                case 4: tmp = 8.0/15.0*pi*pi*Rpow(Real(nu[i][j]),Real(2.5))/m[i];
                    tmp.rwrite(stdout, 7);
                    break;
                case 5: tmp = 1.0/6.0*pi*pi*pi*Rpow(Real(nu[i][j]),Real(3.0))/m[i];
                    tmp.rwrite(stdout, 7);
                    break;
            }
        }

        fp << "\n";
    }

```

```
        fp << "\n";  
    }
```


The Results Generated by The Program In Appendix 4B**Description**

The results also can be found in *The Art of Computer Programming*, p. 102-103.

No. multiplier	modulo	v2^2	v3^2	v4^2	v5^2	v6^2
1 +23	+100000001	+530	+530	+530	+530	+447
2 +129	+34359738368	+16642	+16642	+16642	+15602	+252
3 +262145	+34359738368	+34359738368	+6	+4	+4	+4
4 +3141592653	+34359738368	+2997222016	+1026050	+27822	+1118	+1118
5 +137	+256	+274	+30	+14	+6	+4
6 +3141592621	+10000000000	+4577114792	+1034718	+62454	+1776	+542
7 +3141592221	+10000000000	+4293881050	+276266	+97450	+3366	+2382
8 +4219755981	+10000000000	+10721093248	+2595578	+49362	+5868	+820
9 +4160984121	+10000000000	+9183801602	+4615650	+16686	+6840	+1344
10 +3141592221	+34359738368	+13539813818	+5795090	+88134	+12716	+2938
11 +2718281829	+34359738368	+22939188896	+2723830	+148116	+10782	+2914
12 +1220703125	+34359738368	+33161885770	+2925242	+113374	+13070	+2256
13 +30517578125	+34359738368	+22078865098	+10274746	+167558	+5844	+2592
14 +8392709	+34359738368	+167510120	+8052254	+21476	+16802	+1630
15 +8396805	+34359738368	+168231328	+5335322	+21476	+2008	+1134
16 +8404997	+34359738368	+12256151168	+5733878	+21476	+13316	+2032
17 +4202501	+34359738368	+8201443840	+1830230	+21476	+7786	+3080
18 +16785413	+34359738368	+8364058	+8364058	+21476	+16712	+1496
19 +19935388837	+34359738368	+32300850938	+705518	+22270	+9558	+2660
20 +1175245817	+34359738368	+36436418002	+7362242	+95306	+3006	+2860
21 +17059465	+34359738368	+39341117000	+9476606	+202796	+18758	+2382
22 +65539	+536870912	+536936458	+118	+116	+116	+116
23 +1812433253	+4294967296	+4326934538	+1462856	+15082	+4866	+906
24 +1566083941	+4294967296	+4659748970	+2079590	+44902	+4652	+662
25 +69069	+4294967296	+4243209856	+2072544	+52804	+6990	+242
26 +1664525	+4294967296	+4938916874	+2322494	+63712	+4092	+1038
27 +314159269	+2147483647	+1432232969	+899290	+36985	+3427	+1144
No. multiplier	modulo	v2^2	v3^2			
28 +34522712143931	+281474976710656	+282776785804456	+2733479264			
v4^2	v5^2	v6^2				
+12130342	+308750	+54478				
No. multiplier	modulo	v2^2	v3^2			
29 +31167285	+281474976710656	+322492826755072	+4111841446			
v4^2	v5^2	v6^2				
+17341510	+306326	+59278				
No. multiplier	modulo	v2^2	v3^2			
30 +6364136223846793005	+18446744073709551616	+8810664174654508192	+6398304806574			
v4^2	v5^2	v6^2				
+4112636266	+45662836	+1846368				

No.	_lgv2_	_lgv3_	_lgv4_	_lgv5_	_lgv6_	_u2_	_u3_	_u4_	_u5_	_u6_
1	+4.524	+4.524	+4.524	+4.524	+4.402	+0.000	+0.000	+0.013	+0.340	+4.615
2	+7.011	+7.011	+7.011	+6.964	+3.992	+0.000	+0.000	+0.039	+4.658	+0.002
3	+17.50	+1.382	+1.035	+1.035	+1.035	+3.141	+0.000	+0.000	+0.000	+0.000
4	+15.74	+9.985	+7.381	+5.063	+5.063	+0.274	+0.126	+0.111	+0.006	+0.210
5	+4.050	+2.461	+1.939	+1.382	+1.035	+3.362	+2.688	+3.778	+1.813	+1.316
6	+16.04	+9.993	+7.965	+5.397	+4.541	+1.437	+0.441	+1.924	+0.069	+0.082
7	+16.00	+9.037	+8.286	+5.858	+5.609	+1.348	+0.060	+4.686	+0.346	+6.984
8	+16.65	+10.65	+7.795	+6.259	+4.840	+3.368	+1.751	+1.202	+1.388	+0.284
9	+16.54	+11.06	+7.013	+6.369	+5.196	+2.885	+4.153	+0.137	+2.036	+1.254
10	+16.82	+11.23	+8.213	+6.817	+5.760	+1.237	+1.700	+1.115	+2.793	+3.814
11	+17.20	+10.68	+8.588	+6.698	+5.754	+2.097	+0.548	+3.150	+1.849	+3.721
12	+17.47	+10.74	+8.395	+6.837	+5.569	+3.032	+0.609	+1.846	+2.991	+1.726
13	+17.18	+11.64	+8.677	+6.256	+5.670	+2.018	+4.015	+4.032	+0.399	+2.619
14	+13.65	+11.47	+7.195	+7.018	+5.335	+0.015	+2.785	+0.066	+5.605	+0.651
15	+13.66	+11.17	+7.195	+5.485	+5.073	+0.015	+1.502	+0.066	+0.027	+0.219
16	+16.75	+11.22	+7.195	+6.850	+5.494	+1.120	+1.673	+0.066	+3.134	+1.268
17	+16.46	+10.40	+7.195	+6.463	+5.794	+0.749	+0.301	+0.066	+0.819	+4.394
18	+11.49	+11.49	+7.195	+7.014	+5.273	+0.000	+2.966	+0.066	+5.531	+0.503
19	+17.45	+9.714	+7.221	+6.611	+5.688	+2.953	+0.072	+0.071	+1.368	+2.830
20	+17.54	+11.40	+8.270	+5.776	+5.741	+3.331	+2.435	+1.304	+0.075	+3.518
21	+17.59	+11.58	+8.814	+7.097	+5.609	+3.597	+3.556	+5.906	+7.382	+2.032
22	+14.50	+3.441	+3.429	+3.429	+3.429	+3.141	+0.000	+0.000	+0.001	+0.015
23	+16.00	+10.24	+6.940	+6.124	+4.912	+3.164	+1.725	+0.261	+2.024	+0.894
24	+16.05	+10.49	+7.727	+6.091	+4.685	+3.408	+2.932	+2.316	+1.808	+0.349
25	+15.99	+10.49	+7.844	+6.385	+3.960	+3.103	+2.914	+3.203	+5.006	+0.017
26	+16.10	+10.57	+7.980	+6.007	+5.009	+3.612	+3.451	+4.664	+1.330	+1.345
27	+15.20	+9.889	+7.587	+5.871	+5.079	+2.095	+1.663	+3.143	+1.685	+3.602
28	+24.00	+15.67	+11.76	+9.118	+7.866	+3.156	+2.126	+2.579	+0.990	+2.968
29	+24.09	+15.96	+12.02	+9.112	+7.927	+3.599	+3.923	+5.272	+0.971	+3.824
30	+31.46	+21.27	+15.96	+12.72	+10.40	+1.500	+3.675	+4.524	+4.020	+1.763

NAME

drand_svc, **drand_tool** — distributed random number generator system server and its companion program

SYNOPSIS

drand_svc [**-dpv**]
drand_tool [**-bdr**]

DESCRIPTION

drand_svc is a local area network utility program, which can be scheduled to run on a computer host. **drand_svc** manages a set of seven hundred thousands linear congruential pseudorandom number generators to support distributed computations. It can serve up to fifty application clients at the same time, each of the application obtains a logical copy of the entire set of the generators. A logical set of the generators will be well organized and is suitable to the application's need. The linear congruential random number generators were found by the method provided by [Percus & Kalos, 1987], reproducibility is guaranteed by the server. The person who starts **drand_svc** becomes the *superuser* to the program.

drand_creat(8D) describes user utility functions interfacing to the server.

COMMAND

drand_svc

OPTIONS

- d** Debug flag. When it is issued, **drand_svc** writes all useful intermediate messages into a file *drandsvc.log*.
- p** **drand_svc** supports two mutually exclusive removal options to help with maintaining registered applications. If **-p** option is employed, **drand_svc** runs pre-emptive mode. It has the super priority to manage registered application. When an application's registration is expired and a new application requires to support, the expired registration is subject to be removed. (see **drand_creat(8D)** for registering applications.)
- v** When **drand_svc** is started with **-v** option, users are responsible for removing registered client programs (see **drand_creat(8D)**). **drand_svc** will not be able to remove any registered programs even if those registered programs that are not running any more. No other users can remove registered client applications beside those of themselves. An exception of removal a registered application *should* be employed when **drand_svc** can no longer serve the application due to some difficulties. **-v** option is default. (see **drand_tool**)

COMMAND

drand_tool

A **drand_tool** request will be accepted by **drand_svc** if the tool command is issued by the *superuser*. The contents of server's table will be printed on the console where **drand_tool** is lunched.

OPTIONS

- dn** **drand_tool** is provided to help managing **drand_svc**, especially, when the server enters some peculiar states. **-d** is used to alter *debugging* method during the serve is running. When **drand_svc** behaves un-normally, some treatments can be taken. Turning debug flags on is the first choice. **-d1** will trace all undergoing functions; **-d2** will print mapping tables, allocated addresses when these values are changed; **-d3** serves combination of two. The output will be written into the log file: *drandsvc.log*. **-d0** will turn all debug flags off.
- b** deleting or removing a running **drand_svc** is the last and worst choice to solve server's problems. When an identified and in-troubled application is removed by a **-r** operation, the server should be shutdown and be examined. The first step to shut down **drand_svc** is to run **drand_tool** with **-b** option. This option *blocks* all additional registrations, and lets the registered application continuously run, until they exit.
- rn** Even if **drand_svc** is running in ill-condition, **drand_tool** may still be able to communicate with the server. **drand_tool** is designed to make best effort to dialog with the running server, and sends troubleshooting instructions. After **-d3** option with **drand_tool** is employed, an in-troubled application may be identified. **-rn** is used to delete the application by an identification number, denoted by n. When **drand_svc** dumps its map table and generator handles, the identification numbers are part of them.

ENVIRONMENT

DRANDSERV=fusion or some other host
 DRANDBASE=path to locate the drand database

DIAGNOSTICS

Potential ill-conditioned applications are those that apply trees with thousands levels, or *too* many generators in the trees. (Although large number of generators is prepared for *most* of application's needs, the program that uses all the generators by generating a complete tree of hundred levels can not be supported. It strongly supports a large *variety* of process creations.) All these demands requires heavy memory, computation and communication message passing. Therefore, they may cause some problem.

BUGS

drand_svc is a stateful server which keeps partial information of users' applications. And the other part of information of the parallel random number generators is distributedly stored in client' processes. The server has no way of knowing when a client application terminates. On the other hand, it may not be available that how long a client programs will run when the program is started, typical such programs are Monte Carlo calculations. In addition, if a re-run is expected, an application program will not call **drand_remove()** to remove the resisted entry from the distributed random number generator server. Therefore, some registered clients will stay in the server for very long period of time and it may never be reused and removed. **drand_svc** can run either *voluntary* or *pre-emptive* mode to meet general requests.

EXAMPLES

run nice and background mode routing initial messages into out:
 host\$ nohup **drand_svc** > out&

block additional registrations by **drand_tool**:

```
host$ drand_tool -b
```

SEE ALSO

drand_client(8D), drand_creat(8D), drand_demo(8D)

NAME

drand_client — user application programs

SYNOPSIS

drand_client [**-dpqrtu**]

DESCRIPTION

A **drand_client** is a user application program, which is linked with *libdrand.a*. Client programs employ **drand** interfacing functions to manipulate a virtually complete binary tree. A virtually complete binary tree is transparent to users and it is expanded itself along with needs. The tree is reproducible, meaning of it is that results of users' application program can be re-generated. (see **drand_demo** files)

COMMAND

drand_clnt

drand_clnt is a generic name of users' application executable program. After first run of the program, a drand **JOBID** is created as a disk file stored in the directory where **drand_clnt** is issued, or say **JOBID** is the name of a file. In side of the file, there is some record written. A **JOBID** can be submitted as an option with the program to reproduce the results. The program is allowed to be modified, as long as the process creation sequence is preserved, the results can be completely reproduced. If the process creation is not preserved, some error messages will be generated. A **JOBID** has the following format:

<drand_client>_on_<host>_for_<user>_at_<time>_<duration>. where **<drand_client>** is the program's name; **<host>** is the machine' name on which the program is submitted; **<user>** is the user's login id; **<time>** is the time the program is started; **<duration>** is the user specified registration period.

If no options specified, this client program enters the registration mode. If this client program runs the second time with no option(s) given, the reuse mode will be switched. If the first run did not or has not yet complete(d), an error will be generated to the second run; The first unfinished run will not be affected.

OPTIONS

-dJOBID

Delete a registered application recognized by **JOBID**.

-p This option is privileged, only *superuser*'s demand will be recognized by **drand_svc**. This option causes **drand_svc** registration table dumped to the console where the option is issued.

-q is query option supporting to all users. **drand_svc** prints all entries belong to the user who demand the query to the console where the command is issued. After printing the demanded entries of **drand_svc**' table, the program terminates.

-rJOBID

This option is exactly the same option as **-d**.

-tn When **drand_svc** runs on *pre-emptive* mode, a registered application may be removed upon some requests. To prevent that from happening, users can submit their application with some time estimation that the program should be kept *n* days. Default set is 1 day

long.

-uJOBID

This option supports re-use a run. **drand_clnt** will use JOBID to fetch mapping history.

ENVIRONMENT

DRANDSERV=fusion or some other host

DRANDBASE=path to locate the drand database

DIAGNOSTICS

drand system will try the best possible to communicate with **drand_svc** running remotely. When the network is too busy, users' demand may be refused after a number of tries.

BUGS

Error messages produced by **drand** have the format: **Error: <reason>**. If the message is non-trivial, the **drand handle** is set *bad*. If message produced by **drand** has subject, **System Error:**, please report the message to **drand_svc superuser**.

EXAMPLES

run nice and background mode routing initial messages into out:

```
host$ nohup drand_client > out&
```

to fetch all entries belonging to me:

```
host$ drand_clnt -q
```

to use demo1_on_fusion_for_jerry_at_May__1_20_24_1:

```
host$ demo1_modified \  
-udemo1_on_fusion_for_jerry_at_May__1_20_24_1&
```

SEE ALSO

drand_svc(8D), **drand_creat(8D)**, **drand_demo(8D)**

NAME

drand_creat, drand_new, drand_self, drand_remove, drand_fork, drand_branch, drand_advance, drand_random, drand_fprint, drand_copy, drand_encode, drand_decode – utility functions interfacing to the server program, **drand_svc**

SYNOPSIS

```
#include "drand.h"
```

```
int drand_creat( ac, av, h )
```

```
int  ac;
```

```
char **av;
```

```
dhandle *h;
```

```
int drand_new( h )
```

```
dhandle *h;
```

```
int drand_self( h )
```

```
dhandle *h;
```

```
int drand_remove( ac, av, h )
```

```
int  ac;
```

```
char **av;
```

```
dhandle *h;
```

```
int drand_copy( duph, h )
```

```
dhandle *duph;
```

```
dhandle h;
```

```
int drand_fprint( fp, h )
```

```
FILE *fp;
```

```
dhandle *h;
```

```
int drand_random( h, i )
```

```
dhandle *h;
```

```
int  i;
```

```
int drand_advance( h )
```

```
dhandle *h;
```

```
int drand_fork( h, child )
```

```
dhandle *h;
```

```
dhandle *child;
```

```
int drand_branch( h, child )
```

```
dhandle *h;
```

```
dhandle *child;
```

```
int drand_encode ( codeword, h )
```

```
codetype *codeword;
```

```
dhandle h;
```



```
int drand_decode ( h, codeword )
dhandle *h;
codetype codeword;
```

DESCRIPTION

drand_creat(), **drand_new()**, **drand_self()** are remote procedure calls to interface the distributed random number system server: **drand_server(8D)**. **drand_remove()**, **drand_fork()**, **drand_advance()**, **drand_branch()** are application functions which employ the three remote procedure calls. **drand_copy()**, **drand_random()**, **drand_fprint()**, **drand_encode()**, **drand_decode()** are utility functions. These functions are defined in the header file: **drand.h** and the definitions are stored in the run time library called **libdrand.a**.

drand_creat(ac, av, h)

drand_creat() is used to obtain a logic copy of the distributed random number generators. **h** will contain a fresh drand dhandle if no error arises. **ac** and **av** are used to fetch command line options.

drand_new(h), drand_self(h)

drand_new() and **drand_self()** manipulate positions in a complete binary tree. The tree will expanded when it is needed. Upon a successful operation, the **h** in the call **drand_new(h)** will contain a new generator that is the left subnode of the complete binary tree from the original position, **drand_self(h)** remains the same generator, but position of the complete binary tree is altered.

drand_advance(h), drand_branch(h, child), drand_fork(h, child)

employ **drand_new(h)** and **drand_self(h)** to allocate generators. **h** will hold the first generator in **h**'s right subtree. after **drand_advance(h)** is called. **h** will hold the first generator in **h**'s right subtree; and **child** will hold the first generator in **h**'s left subtree after **drand_advance(h, child)** is called. **h** will hold the same generator and **child** will hold the first generator in **h**'s left subtree after **drand_fork(h, child)** is called.

drand_remove(ac, av, h) removes a registered application entirely.

drand_random(h, i) produces a pseudorandom number, where **i** is an initial value.

drand_copy(dumph, h) copies handle **h** to **dump**.

drand_fprint(fp, h) prints handle **h** into file **fp**.

drand_encode(codeword, h), drand_decode(codeword, h) encodes or decodes handle **h** into **codeword**. The **codeword** in string format can be used by functions provided by other libraries.

SEE ALSO

drand_svc(8D), **drand_clnt(8D)**, **drand_demo(8D)**

NAME

drand_demo1.c – demonstration of basic uses

COMPILE with drand.h

DESCRIPTIONS

There are six demonstration c files provided to help using distributed random number package. **drand_demo1.c** illustrates the uses of all user interface functions. **drand_demo2.c** shows how to generate a complete binary tree by using a recursive function *traversal()*. **drand_demo3.c** is another way to generate a complete binary tree. The difference is that this demo program uses *fork()* function calls so that **drand_new()** are called by children processes, real branches. Because the essential feature of *fork()* is just run two copies of the same program, there is no need to pass **handles** from one process to another. **drand_demo4.c** is again designed to generate a complete binary tree, but it employs *execl()* function calls that **handles** are needed to transmit between two un-related processes. **drand_demo5.c** is the program that runs remote spawns. It is pretty same as **drand_demo4.c**, but *rexec()* functions are called. **drand_demo6.c** is a simple simulation program that shows individual realization. No a serious reproducibility test can be shown until **drand_demo4.c**. After re-run the program, two copies of output can be compared. The output files are not the same, but mapping generators to processes are reproducible.

THE PROGRAM

```
/*
 * Demo program number 1
 * All documented functions are employed
 * Notice: set your shell environment variable DRANDSERV
 */

#include "drand.h"
extern char drand_msg[];

testit(ac, av)
int ac;
char **av;
{
    dhandle myself, child;
    int i, j;
    codetype encoded;
    char buf[256];

    /*
     * drand_creat() is the first interface function an
     * application program to call.
     * If a negative integer returns, print err to see the
     * problem.
     */
    if (drand_creat(ac, av, &myself) < 0) {
        fprintf(stdout,
            "\ntestit(): call drand_creat err=<%s>0,
            drand_msg);
        exit(1);
    }
}
```

```

/*
 * print handle myself
 */
fprintf(stdout, "\nPrint the initial handle ");
(void) drand_fprint(stdout, myself);

/*
 * test function: drand_copy()
 * now, child and myself are the same handle, be careful!
 */
(void) drand_copy(&child, myself);

/*
 * print handle child, is it the same as myself ?
 */
(void) fprintf(stdout, "\nDuplicat myself into child, child is ");
(void) drand_fprint(stdout, child);

/*
 * use child to apply drand_new()
 */
if (drand_new(&child) < 0) {
    fprintf(stdout, "\ntestit(): call drand_new err=<%s>\n",
            drand_msg);
    exit(1);
}

/*
 * print new handle in child
 */
(void) fprintf(stdout,
    "\nchild is used to apply drand_new(), now child is ");
(void) drand_fprint(stdout, child);

/*
 * use myself to apply drand_self()
 */
if (drand_self(&myself) < 0) {
    fprintf(stdout, "\ntestit(): call drand_self err=<%s>\n",
            drand_msg);
    exit(1);
}

/*
 * print new handle in myself
 */
(void) fprintf(stdout,
    "\nmyself is used to apply drand_self(), now myself is ");
(void) drand_fprint(stdout, myself);

/*
 * now use myself to apply drand_random()

```

```

    */
    (void) fprintf(stdout, "\nRandom number from handle myself ");

    for ( i = 0; i < 10; i++ )
        fprintf(stdout, "\nDRAW: %10.8f", drand_random(&myself, 0));

    /*
    * now use child to apply drand_random()
    */
    (void) fprintf(stdout, "\nRandom number from handle child ");
    for ( i = 0; i < 10; i++ )
        fprintf(stdout, "\nDRAW: %10.8f", drand_random(&child, 0));

    /*
    * now print child's handle, generator, seed[], changed
    */
    (void) fprintf(stdout, "\nSeed[] in handle child are changed ");
    (void) drand_fprint(stdout, child);

    /*
    * now test drand_advance( handle )
    * drand_advance( handle ) advances a position from "handle"
    * to righ subtree and returns the first new generator
    */
    if (drand_advance(&myself) < 0) {
        fprintf(stdout,
            "\ntestit(): call drand_advance err=<%s>\n",
            drand_msg);
        exit(1);
    }
    (void) fprintf(stdout,
        "\nAdvance to a new generator from myself's RHS ");
    (void) drand_fprint(stdout, myself);

    /*
    * drand_fork() applies drand_new() and drand_self()
    */
    if (drand_fork(&myself, &child) < 0) {
        fprintf(stdout,
            "\ntestit(): call drand_fork err=<%s>\n",
            drand_msg);
        exit(1);
    }
    (void) fprintf(stdout, "\n%s%s%s",
        "drand_for(&myself, &child) does the operations: ",
        "\ndrand_copy(&child, myself); \ndrand_self(&myself); ",
        "\ndrand_new(&child)");
    drand_fprint(stdout, myself);

    /*
    * drand_branch() applies drand_new() and drand_advance()
    */

```

```

if (drand_branch(&child, &myself) < 0) {
    fprintf(stdout,
        "\ntestit(): call drand_branch err=<%s>\n",
        drand_msg);
    exit(1);
}
(void) fprintf(stdout, "\n%s%s%s",
    "drand_branch(&myself, &child) does the operations: ",
    "\ndrand_copy(&child, myself); \ndrand_advance(&myself); ",
    "\ndrand_new(&child)");
drand_fprint(stdout, myself);

/*
 * Now call combination of drand functions to
 * test the virtually complete binary tree,
 * You set an upper bound for the following loop,
 * and let it runs. Any positive number is OK.
 * Just see what happens. Set to 1000, 2000,...
 * I have enough generators for you!!
 * If you set an upper bound like 5000, it will
 * take lot of time to get job done.
 */
for ( i = 0; i < 5000; i++ ) {

    if (drand_advance(&myself) < 0) {
        fprintf(stdout,
            "\ntestit(): call drand_advance err=<%s>\n",
            drand_msg);
        exit(1);
    }

    if (drand_fork(&myself, &child) < 0) {
        fprintf(stdout,
            "\ntestit(): call drand_fork err=<%s>\n",
            drand_msg);
        exit(1);
    }

    if (drand_branch(&child, &myself) < 0) {
        fprintf(stdout,
            "\ntestit(): call drand_branch err=<%s>\n",
            drand_msg);
        exit(1);
    }
}
/*
 * See what the handles look like
 */
drand_fprint(stdout, myself);
drand_fprint(stdout, child);

/*

```

```

    * encode a handle into string format
    */
    (void) fprintf(stdout, "\nBefore encode:");
    (void) drand_fprint(stdout, myself);
    drand_encode( &encoded, myself );
    (void) fprintf(stdout, "\nAfter encode the string is ");
    (void) fprintf(stdout, "%s", (char *) &encoded );

    drand_decode( &myself, encoded );
    fprintf(stdout, "\nAfter decode:\n");
    drand_fprint(stdout, myself);

    /*
    * Finally, remove the logical copy from
    * drand server
    */
    (void) fprintf(stdout, "\n—— REMOVE IT ——");
    if ( drand_remove(ac, av, &myself) < 0 ) {
        fprintf(stdout, "\ntestit(): call drand_remove err=<%s>\n",
            drand_msg);
        exit(1);
    }
    /*
    * myself is removed, when we print it,
    * we will see some error message
    */
    (void) drand_fprint(stdout, myself);

    /*
    * When we continue using it,
    * we will see it never work again.
    * Yes, give up!
    */
    if (drand_self(&myself) < 0) {
        fprintf(stdout, "\ntestit(): call drand_self err=<%s>\n",
            drand_msg);
        exit(1);
    }
}

main( argc, argv )
int  argc;
char *argv[];
{
    fprintf(stdout, "\n\n\t START OVER AGAIN0);

    /*
    * set no buffering to stdout
    */
    setbuf( stdout, (char *)0 );

    /*

```

```
    * do variant tests
    */
    testit(argc, argv);

    (void) fprintf(stdout, "\nTHAT'S IT\n");
}
```

NAME

drand_demo2.c – demonstration of generating a complete binary tree

COMPILE

with drand.h

DESCRIPTIONS

There are six demonstration c files provided to help using distributed random number package. **drand_demo1.c** illustrates the uses of all user interface functions. **drand_demo2.c** shows how to generate a complete binary tree by using a recursive function *traversal()*. **drand_demo3.c** is another way to generate a complete binary tree. The difference is that this demo program uses *fork()* function calls so that **drand_new()** are called by children processes, real branches. Because the essential feature of *fork()* is just run two copies of the same program, there is no need to pass **handles** from one process to another. **drand_demo4.c** is again designed to generate a complete binary tree, but it employs *execl()* function calls that **handles** are needed to transmit between two un-related processes. **drand_demo5.c** is the program that runs remote spawns. It is pretty same as **drand_demo4.c**, but *rexec()* functions are called. **drand_demo6.c** is a simple simulation program that shows individual realization. No a serious reproducibility test can be shown until **drand_demo4.c**. After re-run the program, two copies of output can be compared. The output files are not the same, but mapping generators to processes are reproducible.

THE PROGRAM

```
/*
 * Demo program number 2
 * traversal expands the virtually complete binary tree to complete
 * Notice: set your shell ENVIRONMENT variable DRANDSERV
 */

#include "drand.h"

/*
 * MAXEXPENSION can be anything. If there is no enough
 * generators to use, the recursive function will return.
 */
#define MAXEXPENSION 100
static int count = 0;

/*
 * int traversal( dhandle handle, int level, int upperbd, int debug )
 * handle is an initialized dhandle;
 * level is the lower bound to expended from;
 * upperbd is the upper bound to be expended to;
 * debug, when set to 1, all intermediate results will be printed.
 */
traversal( handle, level, upperbd, debug )
dhandle *handle;
int level;
int debug;
```



```

{
    dhandle another;

    if ( level != upperbd ) {
        fprintf(stdout, " [%d] ", count++);
        if ( count % 10 == 0 )
            fprintf(stdout, "\n");
        if ( debug ) {
            (void) fprintf(stdout, "\nexpnd handle ");
            (void) drand_fprint(stdout, *handle );
        }
        if (drand_fork(handle, &another) < 0) {
            fprintf(stdout,
                "\ntraversal:drand_fork() err=<%s>, level=%d",
                drand_msg, level);
            return(1);
        }
        if ( debug ) {
            (void) fprintf(stdout,
                "\nPrint at height %d, self ", level);
            (void) drand_fprint(stdout, *handle);
            (void) fprintf(stdout,
                "\nPrint at height %d, new ", level);
            (void) drand_fprint(stdout, another);
        }
        return( traversal( handle, level + 1, upperbd, debug ) +
            traversal( &another, level + 1, upperbd, debug ));
    }
    return(0);
}

/*
 * A recursive function is used to travel a complete tree up
 * to MAXEXPENSION
 */

/*
 * A program using drand generators MUST have argc and argv from
 * shell
 */
main( argc, argv, env )
int argc;
char *argv[];
char *env[];
{
    dhandle handle;

    /*
     * not buffering for stdout
     */
    setbuf( stdout, (char *)0 );

```

```
/*
 * create a logic copy of drand generators
 */
if ( drand_creat(argc, argv, &handle) < 0 ) {
    fprintf(stdout, "\ncan't create a handle err=%s", drand_msg );
    exit(1);
}
/*
 * call traversal function, which returns 0 or some other integer
 */
/*
 * int traversal( dhandle handle, int itl, int ub, int debug )
 * handle is an initialized dhandle;
 * itl is the lower bound to expended from;
 * ub is the upper bound to be expended to;
 * debug, when set to 1, all intermediate results will be printed.
 */
fprintf(stdout, "\n %s \n",
( traversal(&handle, 1, MAXEXPENSION, 0) == 0 )
    ? "successful" : "some failure(s)" );
fprintf(stdout, "\nTHAT'S IT\n");
}
```

NAME**drand_demo3.c** – demonstration of process branches**COMPILE**

with drand.h

DESCRIPTIONS

There are six demonstration c files provided to help using distributed random number package. **drand_demo1.c** illustrates the uses of all user interface functions. **drand_demo2.c** shows how to generate a complete binary tree by using a recursive function *traversal()*. **drand_demo3.c** is another way to generate a complete binary tree. The difference is that this demo program uses *fork()* function calls so that **drand_new()** are called by children processes, real branches. Because the essential feature of *fork()* is just run two copies of the same program, there is no need to pass **handles** from one process to another. **drand_demo4.c** is again designed to generate a complete binary tree, but it employs *execl()* function calls that **handles** are needed to transmit between two un-related processes. **drand_demo5.c** is the program that runs remote spawns. It is pretty same as **drand_demo4.c**, but *rexec()* functions are called. **drand_demo6.c** is a simple simulation program that shows individual realization. No a serious reproducibility test can be shown until **drand_demo4.c**. After re-run the program, two copies of output can be compared. The output files are not the same, but mapping generators to processes are reproducible.

THE PROGRAM

```

/*
 * Demo program number 3
 * traversal expands the virtually complete binary tree to complete
 * Notice: set your shell ENVIRONMENT variable DRANDSERV
 * Rather than using drand_fork() in a recursive function,
 * this program recursively forks children processes to generate
 * the binary tree. Suggest not to expend too many levels of the tree
 * due to many system processes will be created at short period of time.
 */

#include "drand.h"

/*
 * MAXEXPENSION should be not more than 50 levels
 */
#define MAXEXPENSION 10
static int count = 0;

/*
 * int traversal( dhandle handle, int level, int upperbd, int debug )
 * handle is an initialized dhandle;
 * level is the lower bound to expended from;
 * upperbd is the upper bound to be expended to;
 * debug, when set to 1, all intermediate results will be printed.
 */

```

```

traversal( handle, level, upperbd, debug )
dhandle *handle;
int level;
int debug;
{
    int pid, status;

    if ( level != upperbd ) {
        if ( debug ) {
            fprintf(stdout, " [%d] ", count++);
            if ( count % 10 == 0 )
                fprintf(stdout, "\n");
        }
        if ( (pid = fork()) == 0 ) {
            if (drand_new( handle ) < 0) {
                fprintf(stdout,
                    "\ntraversal:drand_new() err=<%s>, level=%d",
                    drand_msg, level);
                exit(1);
            }
            if ( debug ) {
                (void) fprintf(stdout,
                    "\nPrint at height %d, new ", level);
                (void) drand_fprint(stdout, *handle);
            }
            exit( traversal( handle, level + 1, upperbd, debug ));
        }
        else
            if ( pid > 0 ) {
                if (drand_self( handle ) < 0) {
                    fprintf(stdout,
                        "\ntraversal:drand_self() err=<%s>, level=%d",
                        drand_msg, level);
                    exit(1);
                }
                if ( debug ) {
                    (void) fprintf(stdout,
                        "\nPrint at height %d, self ", level);
                    (void) drand_fprint(stdout, *handle);
                }
                wait(&status);
                exit( traversal( handle, level + 1, upperbd, debug ));
            }
        else exit(1);
    }
    return;
}

/*
 * A recursive function is used to travel a complete tree up
 * to MAXEXPENSION
 */

```

```

/*
 * A program using drand generators MUST have argc and argv from
 * shell
 */
main( argc, argv, env )
int  argc;
char *argv[];
char *env[];
{
    dhandle  handle;

    /*
     * not buffering for stdout
     */
    setbuf( stdout, (char *)0 );

    /*
     * create a logic copy of drand generators
     */
    if ( drand_creat(argc, argv, &handle ) < 0 ) {
        fprintf(stdout, "\ncan't create a handle err=%s", drand_msg );
        exit(1);
    }
    /*
     * call traversal function, which forks child process to
     * generate the tree.
     */
    /*
     * int traversal( dhandle handle, int itl, int ub, int debug )
     * handle is an initialized dhandle;
     * itl is the lower bound to expended from;
     * ub is the upper bound to be expended to;
     * debug, when set to 1, all intermediate results will be printed.
     */
    traversal(&handle, 1, MAXEXPENSION, 1);
    return 0;
}

```

NAME

drand_demo4.c – demonstration of branches on unrelated processes

COMPILE

with drand.h

DESCRIPTIONS

There are six demonstration c files provided to help using distributed random number package. **drand_demo1.c** illustrates the uses of all user interface functions. **drand_demo2.c** shows how to generate a complete binary tree by using a recursive function *traversal()*. **drand_demo3.c** is another way to generate a complete binary tree. The difference is that this demo program uses *fork()* function calls so that **drand_new()** are called by children processes, real branches. Because the essential feature of *fork()* is just run two copies of the same program, there is no need to pass **handles** from one process to another. **drand_demo4.c** is again designed to generate a complete binary tree, but it employs *execl()* function calls that **handles** are needed to transmit between two un-related processes. **drand_demo5.c** is the program that runs remote spawns. It is pretty same as **drand_demo4.c**, but *rexec()* functions are called. **drand_demo6.c** is a simple simulation program that shows individual realization. No a serious reproducibility test can be shown until **drand_demo4.c**. After re-run the program, two copies of output can be compared. The output files are not the same, but mapping generators to processes are reproducible.

THE PROGRAM

```
/*
 * Demo program number 4
 * traversal expends a complete binary tree.
 *
 * In addition to recursively fork children processes, this program calls execl() *
to carry new copy of itself as its child, which employs drand_new(). The parent *
process employs drand_self().
 * A random number generator handle is passed from one process to another. It is a
good * test of drand_encode() and drand_decode(). The purpose is to test unrelated
processes * and concurrent reproducibility. Distributed reproducibility will be
demoed in Demo 5. *
 * Suggest not to expend too many levels of the tree
 * due to many system processes will be created at short period of time.
 *
 * Because this program checks the number of command arguments, it is NOT
 * suggested using -d, -r, -u and -q options when you start running this program.
 */

#include "drand.h"

/*
 * MAXEXPENSION should be not more than 50 levels
 */
#define MAXEXPENSION 10
static int count = 0;
```

```

static char path[BUFSIZ];

static
getpath( av, path )
char **av;
char path[];
{
    FILE *fpread;
    extern char *sys_errlist[];
    char buf[BUFSIZ];

    if ( (fpread = popen ( "pwd", "r" )) == NULL ) {
        fprintf(stderr, "can't exec sh pwd: %s\n", sys_errlist[errno]);
        exit(1);
    }

    if ( fgets(buf, sizeof(buf), fpread) != NULL )
        buf[strlen(buf)-1] = ' ';
    else {
        fprintf(stderr, "can't get path: %s\n", sys_errlist[errno]);
        exit(1);
    }

    pclose( fpread );
    sprintf(path, "%s/%s", buf, av[0]);
}

/*
 * int traversal( dhandle handle, int level, int upperbd, int debug )
 * handle is an initialized dhandle;
 * level is the lower bound to expended from;
 * upperbd is the upper bound to be expended to;
 * debug, when set to 1, all intermediate results will be printed.
 */

traversal(av, handle, level, upperbd, debug )
char **av;
dhandle *handle;
int level;
int debug;
{
    int pid, status;
    codetype encoded;
    char dec[3];
    extern char *sys_errlist[];

    if ( level != upperbd ) {
        sprintf(dec, "%d", level + 1 );
        if ( (pid = fork()) == 0 ) {
            (void) drand_encode( &encoded, *handle );
            while ( execl(path, av[0], (char *) encoded,

```

```

        dec, (char *) 0 ) < 0 ) {
            fprintf(stdout, "\nexcl failed: s",
                sys_errlist[errno] );
            sleep(1);
        }
        wait(&status);

    } else
    if ( pid > 0 ) {
        if (drand_self( handle ) < 0) {
            fprintf(stdout,
                "\ntraversal:drand_self() err=<%s>, level=%d",
                drand_msg, level);
            exit(1);
        }
        if ( debug ) {
            (void) fprintf(stdout,
                "\nPrint at height %d, self ", level);
            (void) drand_fprint(stdout, *handle);
        }
        exit( traversal(av, handle, level + 1, upperbd, debug ));
    } else exit(1);

    }
    return;
}

/*
 * A recursive function is used to travel a complete tree up
 * to MAXEXPENSION
 */

/*
 * A program using drand generators MUST have argc and argv from
 * shell
 */
main( argc, argv, env )
int  argc;
char *argv[];
char *env[];
{
    dhandle handle;
    int i, level;
    char *buf;

    /*
     * not buffering for stdout
     */
    setbuf( stdout, (char *)0 );
    getpath(argv, path );

    if ( argc == 1 ) { /* the first process */

```



```

/*
 * create a logic copy of drand generators
 */
if ( drand_creat(argc, argv, &handle ) < 0 ) {
    fprintf(stdout, "\ncan't create a handle err=<%s>",
        drand_msg );
    exit(1);
}
level = 1;
fprintf(stdout, "\nThe initial handle ");
drand_fprint(stdout, handle );

} else {

    if ( drand_decode( &handle, argv[1] ) < 0 ) {
        fprintf(stdout, "\ncan't decode the handle err=<%s>",
            drand_msg );
        fprintf(stdout, "\n the handle is %s", argv[1] );
        exit(1);
    }

    if (drand_new( &handle ) < 0) {
        fprintf(stdout,
            "\nmain:drand_new() err=<%s> ", drand_msg);
        exit(1);
    }
    level = atoi( argv[2] );
    fprintf(stdout, "\nPrint at height %d, new ", level);
    (void) drand_fprint(stdout, handle);
}
/*
 * call traversal function, which forks child process to
 * generate the tree.
 */
/*
 * int traversal( dhandle handle, int itl, int ub, int debug )
 * handle is an initialized dhandle;
 * itl is the lower bound to expended from;
 * ub is the upper bound to be expended to;
 * debug, when set to 1, all intermediate results will be printed.
 */
traversal(argv, &handle, level, MAXEXPENSION, 1);
return 0;
}

```

Reference:

1. O. Percus & M. Kalos, *Random Number Generators for MIMD Parallel Processors*, Journal of Parallel and Distributed Computing, 6, 1989.
2. D. Arnow *Stochastic Solutions to the Schrödinger Equation of Fermions*, Courant Institute of Mathematical Sciences, 1982.
3. S. Zhang & M. Kalos *Exact Monte Carlo for Few-Electron Systems* Cornell Theory Center Technical Report, 1992.
4. K. Schmidt & M. Kalos *Few- And Many-Fermion Problems*, Monte Carlo Methods in Statistical Physics II, Springer-Verlag, New York, 1979.
5. M. Kalos *Stochastic Wave Function for Atomic Helium*, Journal of Computational Physics 2, 257-276, 1967.
6. G. Tawa, P. Whitlock & K. Schmidt *Accurate First Principles Calculation of Many-Body Interactions*, The international Journal of supercomputer applications, Vol. 5, No. 1 Spring 1991.
7. D. Ceperley & G. Chester *Exact Calculations of the Ground State of Model Neutron Matter*, Physical Review, Vol. 13, No. 12, June 1976.
8. D. Arnow, M. Kalos, M. Lee & K. Schmidt *Green's Function Monte Carlo for Few Fermion Problems*, J. Chem. Phys 77(11), 1 Dec. 1982.
9. S. Wieder *The Foundations of Quantum Theory*, Academic Press, New York and London, 1973.
10. A. French & E. Taylor *An Introduction to Quantum Physics*, W.W. Norton &

- Company Inc. New York, 1982.
11. J. Powell, *Quantum Mechanics*, Addison Wesley Publishing Company, Inc., New York, Reading, Massachusetts, 1975.
 12. Matthews, *Introduction to Quantum Mechanics*, 3rd Edition, McGraw-Hill Book Company, New York, 1963.
 13. M. Scharff *Elementary Quantum Mechanics*, Wiley Interscience, London, New York, 1969.
 14. J. Bnderson *Quantum Chemistry by Random Walk: Higher Accuracy* J. Chem. Phys 73(8), 15 Oct. 1980.
 15. P. Whitlock & M. Kalos *Monte Carlo Calculation of the Radial Distribution Function of Quantum Hard Spheres at Finite Temperatures*, Journal of Computational Physics, Vol. 30, No. 3, March 1979.
 16. M. Kalos *Energy of A Simple Triton Model*, Nuclear Physics A126 609-614, 1969.
 17. M. Kalos & P. Whitlock *Monte Carlo Methods, Volume I: Basics* A Wiley-Interscience Publication John Wiley & Sons, New York, 1986.
 18. W. Feller, *An Introduction to Probability Theory and Its Applications*, 2 vols., John Wiley & Sons, New York, 1957 and 1966.
 19. K. Trivedi, *Probability & Statistic with Queuing, and Computer Science Applications*, Prentice-Hall, Inc., New York, 1982.
 20. I. Olkin, L. Gleser & C. Derman, *Probability Models and Applications*, Macmillan Publishing Co., Inc., New York, 1978.

21. N. Metropolis, A. Rosenbluth, N. Rosenbluth, A. Teller and E. Teller, *Equations of state calculations by fast computing machines*, *J. Chem. Phys.*, 21, 1953.
22. D. Arnow, *DP technical report, presentation, manual pages*, Brooklyn College, John Jay College, 1993.
23. D. Comer, *Internetworking with TCP/IP, Vol. I, 2nd Edition*, Prentice-Hall, Inc., New York, 1991.
24. P. Fredrickson, R. Hiromoto, T. Jordon, B. Smith and T. Warnock, *Pseudorandom trees in Monte Carlo*, *Parallel Computing*, Vol 1. 1984.
25. H. Lehmer, *Mathematical Methods in Large-Scale Computing Units*, *Annu. Comput. Lab. Harvard Univ.* 26, 1951.
26. E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 1969.
27. R. Coveyou & D. Macphersson *Fourier Analysis of Uniform Random Number Generators* *Journal of the Association for Computing Machinery* , Vol. 14, No. 1. pp 100-119, January 1967.
28. K. Park & W. Miller *Random Number Generators: Good Ones are Hard to Find*, *Communication of the ACM*, Vol. 31, No. 10, Oct. 1988.
29. G. Marsaglia *Random numbers fall mainly in the planes* In *Proceedings of the National Academy of Sciences of the United States of America* 60, pp. 25-28, 1968.
30. G. Marsaglia *A Current View of Random Number Generation*. *Computer*

- science and statistics. In *Proceedings of the Sixteenth Symposium on the Interface* (Atlanta, March 1984). Elsevier Science Publ., North-Holland, 1985, pp. 3-10, 1984.
31. P. L'ecuyer *Random Numbers for Simulation*, *Communications of the ACM*, Vol. 33, No.10, Oct. 1990.
 32. P. L'ecuyer *Implementing a Random Number Package with Splitting Facilities*, *ACM Transactions on Mathematical Software*, Vol. 17, No. 1, March 1991.
 33. R. Hanxleden & L. Scott *Load Balancing on Message Passing Architectures*, *Journal of Parallel and Distributed Computing* 13, pp 312-324, Academic Press, Inc., New York, 1991.
 34. A. Hac & T. Johnson *Sensitivity Study of the Load Balancing Algorithm in a Distributed System* *Journal of Parallel and Distributed Computing* 10, 85-89, Academic Press, Inc., New York, 1990.