

Scalable Parallel Algorithms for High Dimensional Numerical Integration

August 2010

Prepared by

Yahya M. Masalmah , Ph.D.

Yu (Cathy) Jiao, Ph.D.

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via the U.S. Department of Energy (DOE) Information Bridge.

Web site <http://www.osti.gov/bridge>

Reports produced before January 1, 1996, may be purchased by members of the public from the following source.

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone 703-605-6000 (1-800-553-6847)
TDD 703-487-4639
Fax 703-605-6900
E-mail info@ntis.gov
Web site <http://www.ntis.gov/support/ordernowabout.htm>

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange (ETDE) representatives, and International Nuclear Information System (INIS) representatives from the following source.

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Telephone 865-576-8401
Fax 865-576-5728
E-mail reports@osti.gov
Web site <http://www.osti.gov/contact.html>

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Computational Sciences & Engineering Division

**SCALABLE PARALLEL ALGORITHMS FOR HIGH
DIMENSIONAL NUMERICAL INTEGRATION**

Yahya M. Masalmah, Ph.D.*

Yu (Cathy) Jiao, Ph.D.†

*Universidad del Turabo Electrical and Computer Engineering Department

†ORNL Computational Sciences & Engineering Division

Date Published: August 2010

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

	Page
LIST OF FIGURES	v
LIST OF TABLES	vii
ABSTRACT	9
1. INTRODUCTION	9
2. Background	11
2.1 NEUTRON SCATTERING	11
2.2 NUMERICAL INTEGRATION METHODS	13
2.3 PARALLEL PROGRAMMING CHALLENGES	14
2.3.1 Message Passing Interface (MPI)	15
2.3.2 Multithread Programming	17
2.3.3 Mixed MPI and OpenMP programming	18
3. literature review	19
4. PROBLEM Statement	20
4.1 OBJECTIVES	21
5. Implemenation Details	22
5.1 SERIAL VERSION	22
5.2 PARALLEL VERSION	23
6. performance evaluation	26
6.1 ACCURACY EVALUATION	26
6.2 SCALABILITY EVALUATION	30
7. Conclusions and future work	32
8. References	33
APPENDIX A . Integration Variables Transformation	34
APPENDIX B. ORNL Institutional cluster (OIC)	36

LIST OF FIGURES

Figure	Page
Fig. 1. Jaguar Supercomputer at Oak Ridge National Laboratory.	10
Fig.2. Elastic Neutron Scattering ¹²	12
Fig. 3. Inelastic Neutron Scattering ¹²	12
Fig.4. points Sequences examples	14
Fig. 5. Master Workers Parallel Programming Scheme ¹⁴	16
Fig.6. Client-server scheme ¹⁴	17
Fig. 7. Illustration of Multithreading ¹⁰	18
Fig. 8. Mixed MPI and OpenMP Scheme ⁹	18
Fig. 9. Serial version flow diagram.	23
Fig. 10. Parallel version flow diagram.....	25
Fig. 11. Intensity Simulated/ Measured at (Q_x, E)	28
Fig. 12. Intensity predicted by Refined Model at (Q_x, E)	29
Fig.13. Execution time Vs number of data points.	30
Fig.14. Execution time Vs number of processes for 300, 000 data points.	31
Fig. B.1 . ORNL Institutional Cluster (OIC) ¹³	36

LIST OF TABLES

Table	Page
Table 1. Genz's testing functions ⁹	26
Table 2. True value of testing functions integral	27
Table 3. Accuracy and execution time (in Seconds) results of testing functions integral	27
Table 4. Accuracy and execution time (in seconds) obtained from integrating R*S at (-1, -1, - 1, 0)..	29

ABSTRACT

We implemented a scalable parallel quasi-Monte Carlo numerical high-dimensional integration for tera-scale data points. The implemented algorithm uses the Sobol's quasi-sequences to generate random samples. Sobol's sequence was used to avoid clustering effects in the generated random samples and to produce low-discrepancy random samples which cover the entire integration domain. The performance of the algorithm was tested. Obtained results prove the scalability and accuracy of the implemented algorithms. The implemented algorithm could be used in different applications where a huge data volume is generated and numerical integration is required. We suggest using the hybrid MPI and OpenMP programming model to improve the performance of the algorithms. If the mixed model is used, attention should be paid to the scalability and accuracy.

1. INTRODUCTION

Computational Science contributes significantly to most disciplines. Initially, science was primarily *empirical*. More recently, each discipline has developed a new *theoretical* component. Theoretical models play an important role in motivating experiments and generalizing our understanding. In the last 50 years, a *computational* branch has grown in different disciplines. It has grown out of our inability to find closed form solutions for complex mathematical models. Computers can simulate these complex models

Information management makes scientists and engineers face mountains of data that stem from different areas such as: the flood of data from new scientific instruments driven by Moore's Law – doubling their data output every year or so, the flood of data from simulations, the ability to economically store petabytes of data online, and the internet and computational grid that makes all these archives accessible to anyone anywhere, allowing the replication, creation, and recreation of more data².

The volume of data produced by different science and engineering applications is enormous. Acquisition, organization, query, and visualization tasks scale almost linearly with data volumes. By using parallelism, these problems can be solved within fixed times (minutes or hours). Some tasks do not scale linearly with the data volumes which makes challenging to analyze these data. If the data increases a thousand-fold, the work and time to process these data can grow by a significant factor. Many algorithms scale even worse with data volumes. Algorithms with poor scalability are infeasible for terabyte-scale or higher scale datasets.

Most current applications in science and engineering produce huge amounts of data. Data size ranges from regular scale up to exascale. Small data sets can be analyzed using serial computation, simple computing resources, and regular hardware architectures. As the data size gets larger and larger, the demand for supercomputing resources increases

proportionally. There are serious exascale problems that just cannot be solved in any reasonable amount of time with the available computers. The next generation of supercomputers could be used to solve big programming problems and allow for the development of a new generation of scientific and engineering applications.

Different supercomputers are now available around the world. The world's fastest supercomputer today, a Cray XT5 system at Oak Ridge National Laboratory that's known as Jaguar and shown in Figure 1, has a peak performance of 2.3 petaflops. A petaflop is a quadrillion, or 1,000 trillion, sustained floating-point operations per second.



Fig. 1. Jaguar Supercomputer at Oak Ridge National Laboratory.¹

In this research, different multidimensional integration algorithms were explored. The explored algorithms were tested for parallelization suitability. The selected algorithms will be used to compute a four dimensional integral for up to 10^{12} data points of neutron scattering application

The range of Monte Carlo applications is enormous, from the simulation of galactic formation to quantum chromodynamics to the solution of systems of linear equations. Our implemented algorithm could be used for all applications similar to our neutron scattering problem.

2. BACKGROUND

2.1 NEUTRON SCATTERING

X-ray and neutron scattering are very useful techniques in the study of the properties of solids. x-rays are limited in applications due to the high energy of the sources on which typical x-rays are generated. The energy of sources is on the order of several thousand electron volts. This energy is much greater than the average excitation of materials found at room temperature. This makes x-rays particularly well-suited for the study of static properties of systems. Neutrons, on the other hand, have thermal energies on the order of milli-electron volts; the energy range of lattice and spin excitations in solids. Therefore, neutron scattering is a very powerful probe of both statics and dynamics in solids. The neutron also has a known spin, which interacts with other magnetic moments within a material. Thus, neutron scattering can be used as a technique to probe magnetic structures and excitations as well as lattice structure and excitations.

Neutron scattering can be elastic as shown in Figure 2 or inelastic as shown in Figure 3. Inelastic scattering is an experimental technique commonly used in condensed matter research to study atomic and molecular motion as well as magnetic and crystal field excitations. It distinguishes itself from elastic neutron scattering techniques by resolving the change in kinetic energy that occurs when the collision between neutrons and the sample is an inelastic one. Results are generally communicated as the dynamic structure factor (also called inelastic scattering law) $S(q,\omega)$, or as the dynamic susceptibility $\chi(q,\omega)$ where the scattering vector q is the difference between incoming and outgoing wave vector, and $\hbar\omega$ is the energy change experienced by the sample (negative that of the scattered neutron). When results are plotted as a function of ω , they can often be interpreted in the same way as spectra obtained by conventional spectroscopic techniques; that is, inelastic neutron scattering can be seen as a special spectroscopy¹².

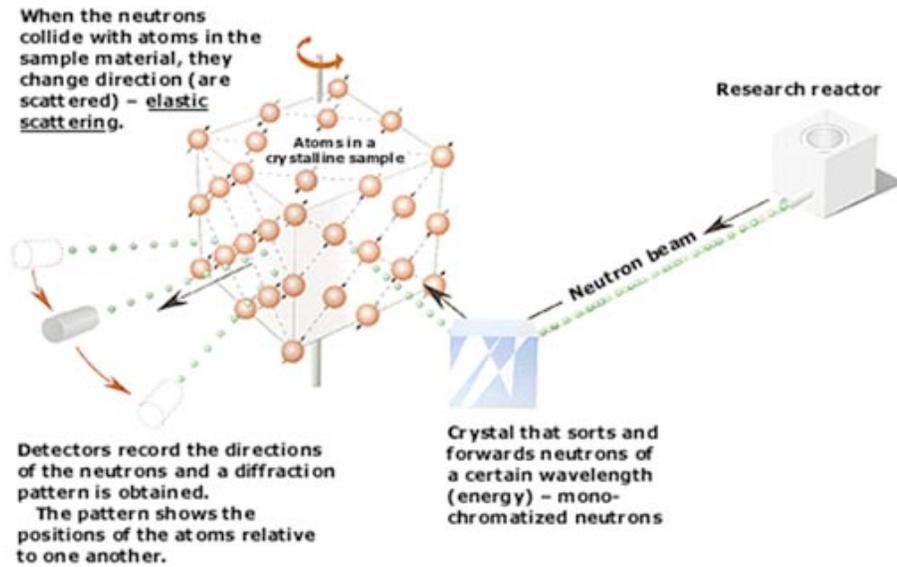


Fig.2. Elastic Neutron Scattering¹²

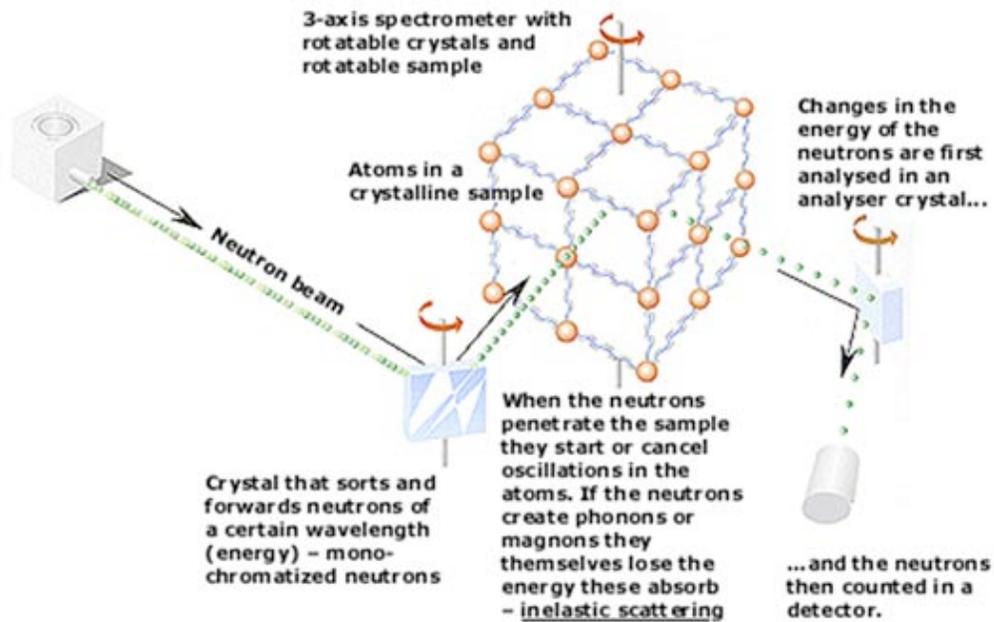


Fig. 3. Inelastic Neutron Scattering¹²

2.2 NUMERICAL INTEGRATION METHODS

Numerical computation of a definite integral of a function of several variables is one of the basic problems in numerical analysis. The problem is considered hard due to the curse of dimensionality, i.e. the computing cost is growing exponentially with the dimension of the problem. The numerical solution of integration problems sometimes requires extensive computation. Therefore, substantial effort has been invested in finding ways to exploit the power of advanced computer architectures like vector or parallel computers to increase the efficiency of algorithms.

A review of existing algorithms for numerical integration of multivariate functions was presented by Thomas Gerstner and Michael Griebel⁶. Some of the presented integration algorithms use sparse grids to estimate the integrations. Some of the sparse grid methods examined were the Trapezoidal rule, Clenshaw-Curtis formulas, Gauss, and Gauss-Patterson formulas are examples of sparse grid methods examined. These formulas are examples of nested univariate quadrature formulas. Nested formulas perform multidimensional integration by recursively calling one dimensional integration formulas.

Monte Carlo integration is one of the most widely used methods in multidimensional integration. It is also considered among the most accurate methods in different applications¹. The Monte Carlo method interprets the integrand function as a random variable, and estimates the multidimensional integration by the statistical average over independent, identically distributed samples $\bar{x}_i \in [0,1]^d$. The samples are generated using pseudo-random sequences. The integral can be written as

$$I = \int_{[0,1]^d} f(\bar{x}) d^d x \cong \frac{1}{N} \sum_{i=1}^N f(\bar{x}_i)$$

The Monte Carlo method converges⁷ at the rate of $O(N^{-1/2})$. It is superior to the Newton-Cotes formula for high dimensional problems. One key factor that slows the convergence of Monte Carlo method is using the pseudo-random sequences to generate the random samples. The implementation of pseudo-random sequences produced a clustering effect which slows the convergence of the Monte Carlo method. Many attempts were made to improve the convergence of the Monte Carlo method. The quasi-Monte Carlo method is an improved version of the Monte Carlo method. It uses deterministic sequences, called quasi-random or low discrepancy, sequences instead of pseudo-random sequences. Quasi-random sequences have the advantage of uniformity of generated samples which can be quantified by its discrepancy, where sequences with low discrepancy are closer to uniformity.⁶ Examples of generated sequences are shown in Figure 4.

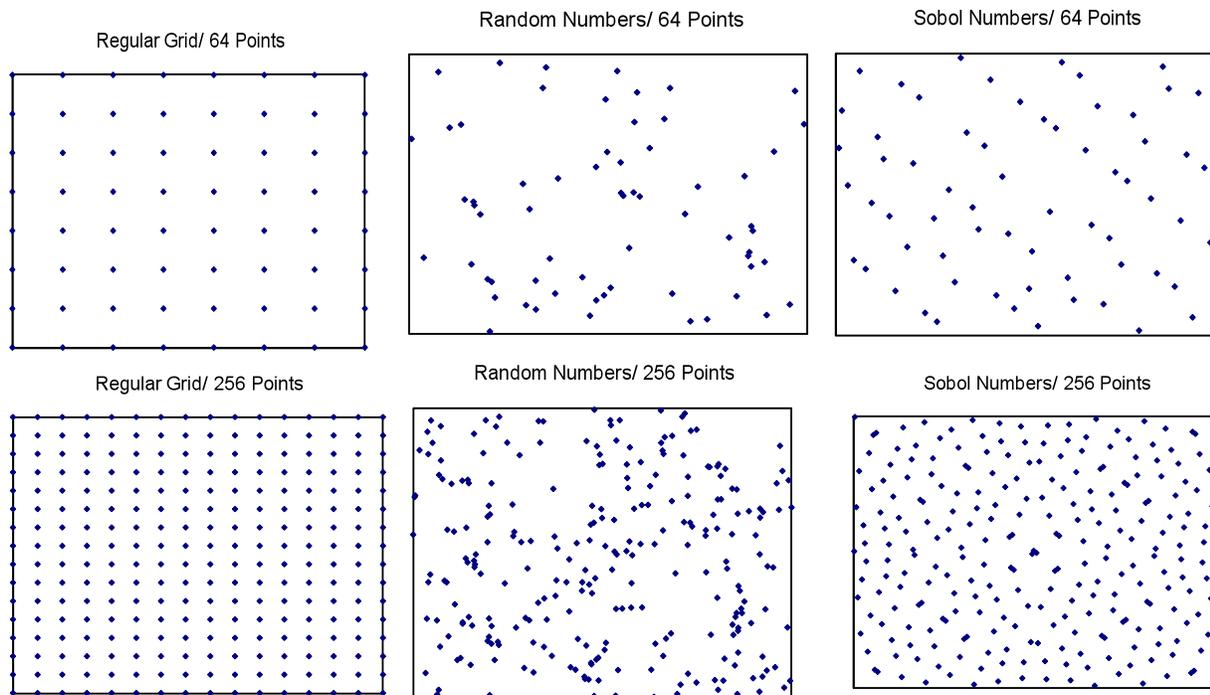


Fig.4. points Sequences examples

2.3 PARALLEL PROGRAMMING CHALLENGES

Parallel programming is intended to achieve high performance computing. In parallel programming, there is no general framework to implement parallel programs for different software applications. Programmers face four immediate challenges when writing parallel programs: scalability, correctness, maintainability, and problem decomposition. There are two types of problem decomposition: functional decomposition, and data or domain decomposition.

Functional decomposition is used to introduce concurrency in the problems that can be solved by different independent tasks. All these tasks can run concurrently. On the other hand, data decomposition works best on an application with large data structure. A task is decomposed by partitioning the data on which computations are performed. The tasks performed on the data partitions are usually similar.

Parallel overhead is another parallel programming challenge. Parallel overhead refers to the amount of time required to coordinate parallel tasks as opposed to doing useful work. Parallel overhead typically includes the time to start and terminate a task, the time to pass messages between tasks, synchronization time, and other extra computation time.

Synchronization is necessary in multithreading programs to prevent race conditions. It limits parallel efficiency even more than overhead in that it serializes parts of the program⁷. Load balancing refers to the practice of distributing work among tasks so that all processes are kept

busy all of the time. It can be considered a minimization of process idle time. Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance. For this reason, load balancing is considered one of the reasons behind poor scalability.

As discussed before, the goal of writing parallel programs is to achieve high performance. Parallel programs that perform a task quickly and with high accuracy are the most desirable. However, parallel programs can sometimes perform worse than serial programs for the same problem. This is due to poor scalability of parallel programs, resulting from the challenges discussed previously. Another important issue is the accuracy. Fast programs with inaccurate results make no sense. Sometimes low accuracy comes from incorrect communications between processes.

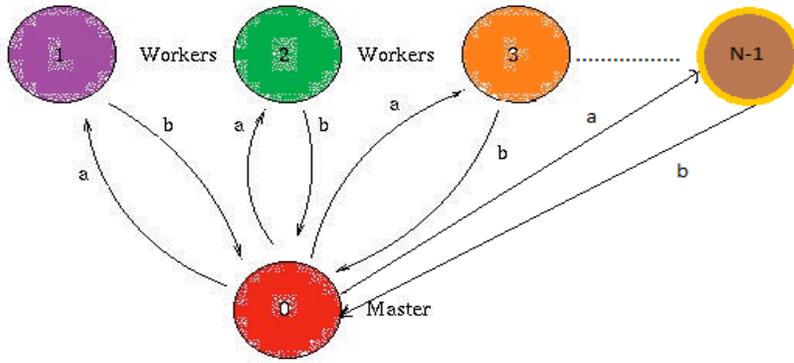
Parallel programming can be done in one of the following models: Cluster parallelization (Message Passing Interface), Open Multi-Processing (OpenMP), or a mixture of both MPI and OpenMP.

2.3.1 Message Passing Interface (MPI)

Message Passing Interface (MPI)⁹ is a cluster-based parallel programming model. It is a library of functions (in C language) or subroutines (in Fortran) that a programmer inserts into source code to perform data communication between processes. MPI provides a portable code and allows efficient implementation across a range of computer architectures. Usually, MPI programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and terminate communications between processes. These calls are responsible for starting communications, identifying the number of processes being used, creating subgroups of processes, and identifying which process is running a particular instance of a program.
2. Calls used to communicate between pairs of processes. This class of calls is called point-to-point communications operations. It consists of different types of send and receive operations.
3. Calls used to perform communications operations among groups of processes. This class of calls is known as the collective operations that provide synchronization or certain types of well-defined communications operations among groups of processes.
4. Calls used to create arbitrary data types. This class of calls provides flexibility in dealing with complicated data structures.

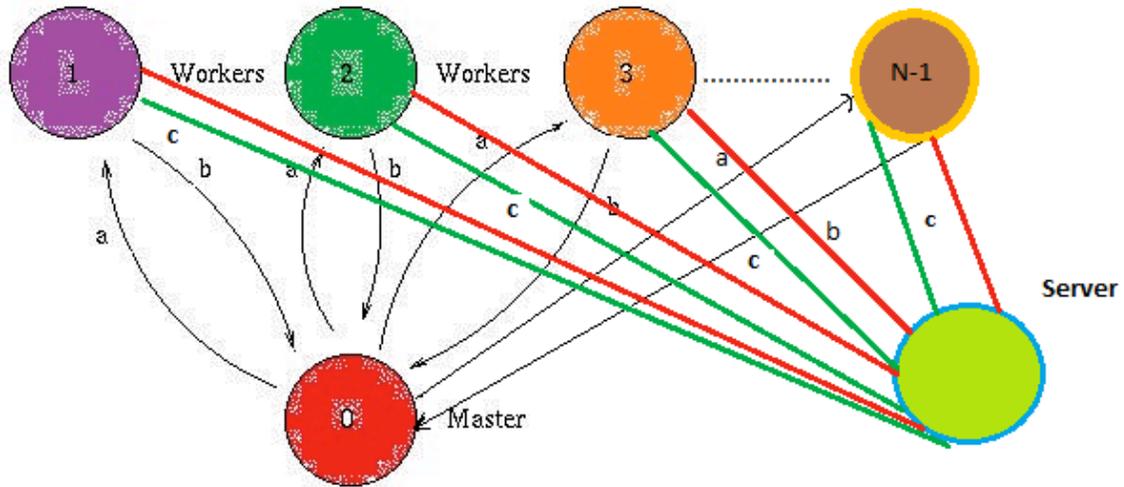
There are different MPI code implementation schemes: master-worker, client-server, and full workers schemes. In the master-worker scheme, the master process manages all tasks between workers. Since there is no inter-worker communication in this scheme, it gives reasonable execution time. A main disadvantage is that a lot of the workload is carried by the master process. The master-worker scheme is shown in Figure 5.



Step a: Node 0 (Master) send the data points assigned for each worker.
 Step b: Send the computed integral at the assigned data points for the Master

Fig. 5. Master Workers Parallel Programming Scheme¹⁴.

The client-server scheme is different from the master-worker scheme. In addition to the master process, this scheme assigns another process known as the server to handle tasks or generate data that will be used by all the workers in the group. The server process will communicate with all processes except the master. This means additional communication times between server and workers which can cause poor scalability in some applications. The client-server scheme is illustrated in Figure 6.



Step a: Node 0 (Master) send the data points assigned for each worker.
Step b: Send the computed integral at the assigned data points for the Master
Step c: Server receives request for data and respond with the requested data.

Fig.6. Client-server scheme¹⁴.

An example of using the full workers scheme is computing a numerical integral using the Monte Carlo method. In this scheme, the master administrates the ranges of random number to be generated for each worker, but leaves the actual generation of random numbers to the workers. Upon receiving a range, the workers generate these numbers and calculate the Monte Carlo sub-sums. This implies that they must generate and discard those random numbers prior to their range selection. This is a waste of computing resources.

2.3.2 Multithread Programming

Open Multi-Processing (OpenMP)⁹ is a multithread programming technique. OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables. Figure 7 illustrates the multithread programming concept.

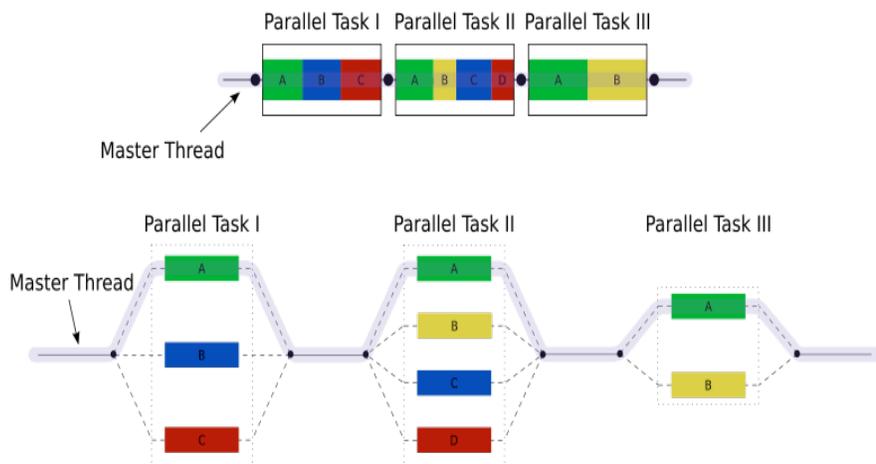


Fig. 7. Illustration of Multithreading¹⁰

2.3.3 Mixed MPI and OpenMP programming

To accomplish better performance, a mixture of MPI and OpenMP code can be implemented. In this mixture, each process in MPI forks the assigned independent tasks into multithreads to get better performance. The mixture programming model is illustrated in Figure 8.

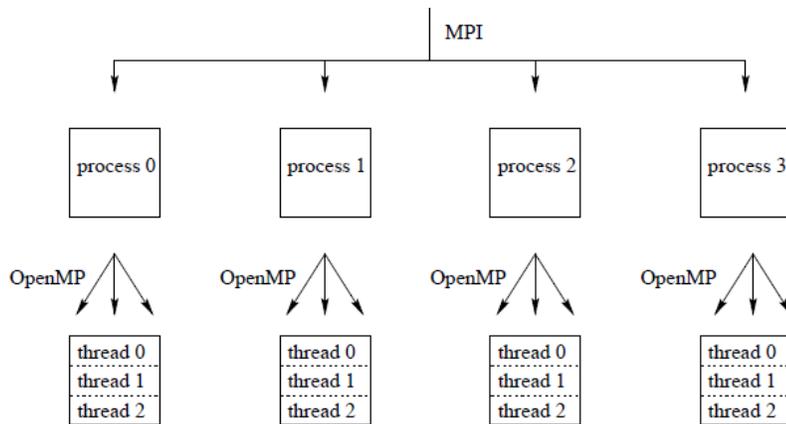


Fig. 8. Mixed MPI and OpenMP Scheme⁹

3. LITERATURE REVIEW

Parallel high-dimensional numerical integration is used in different science and engineering applications. The most used parallel algorithms for high-dimensional numerical integration are quasi-Monte Carlo and adaptive cubature rules^{2,11}. The comparison between these two methods is based on their performance for high-dimensional applications. Comparison between performance of Quasi-Monte Carlo and adaptive Cubature rules depends on the dimension of integration and one the smoothness of the integrand.

The advantage of quasi-Monte Carlo over the adaptive cubature rules is that it is easy to implement and parallelize. It splits the random samples into block with each process taking care of one of them. The adaptive cubature algorithms apply cubature rules successively to smaller subregions of the original integration domain. It adapts to difficult areas in the integration domain by refining subregions with larger estimated errors. The algorithms are iterative. The loop can be terminated using a convergence criteria on the relative error or when the number of integrand evaluations exceeds an upper bound. Adaptive algorithms scale badly even for a moderate number of processes. To improve the scalability, all global communication has to be removed. Therefore master-worker does not work well for adaptive algorithms. Each process executes the sequential algorithm on a subset of subregions of the initial integration domain.

A list of adaptive algorithms for numerical integration was presented by Bull⁴. The adaptive algorithms were divided into two classes. They are single list algorithms and multiple list algorithms³. In single list algorithms, sequential globally adaptive algorithms are modified by introducing a subregion selection strategy that enables multiple subregions to be subdivided concurrently. The objective of the subregion selection strategy is to identify a sufficient number of subregions with larger error estimates to keep all processes usefully busy. These methods work best in one dimension. In multidimensional applications, these algorithms divide the region into different equal subregions in the most badly behaved direction, which has unfortunate consequences, such as longer execution time, and low accuracy. This due to that: parts of the region where the integrand is locally well behaved in the selected direction may be subdivided.

Multiple list algorithms incorporate a load balancing strategy to obviate the difficulty of using an initial static subdivision of the region alone. These algorithms require synchronization between processes. This could cause a delay in the execution time.

Our algorithm is quasi-Monte Carlo based. It uses Sobol's quasi-sequence to generate random numbers. The reason behind choosing this algorithm is that it works better than other algorithms in high-dimensional applications. Another reason is that it uses Sobol's sequence, a low-discrepancy sequence. It generates the random samples to uniformly cover the entire integration domain. It avoids the clustering effect caused by pseudo-random number sequences.

4. PROBLEM STATEMENT

The observed intensity for a given scattering at a particular point (\mathbf{Q}_0, E_0) in the (\mathbf{Q}, E) space is given by

$$I^{calculated}(\vec{Q}_0, E_0) = \int_{\Delta Q = -\infty}^{\infty} \int_{\Delta E = -\infty}^{\infty} S(\vec{Q}_0 + \Delta\vec{Q}, E_0 + \Delta E) \times R(\Delta\vec{Q}, \Delta E) d\Delta\vec{Q} d\Delta E$$

Where

$S(\vec{Q}, E)$ is the sample scattering function, and $R(\Delta\vec{Q}, \Delta E)$ is the resolution function of the spectrometer.

The sample scattering function consists of three different components: background, incoherent, and the ladder components. The background component is a constant. The incoherent component is given by

$$S_{incoh} = \frac{\max incoh}{\sqrt{2\pi} \sin coh} \times e^{-\frac{E^2}{2 \sin coh^2}}$$

Where $\max incoh$ a constant parameter, and $\sin coh$ is the standard deviation.

The ladder component is given by

$$S_{ladder} = \frac{\max int}{\sqrt{2\pi} \sigma E} \times e^{-\frac{(E-w)^2}{2 \sigma E^2}} \times \frac{f dperp \times f f}{w}$$

where $\max int$ is a constant parameter, σE is the standard deviation, w is the dispersion, $f dperp$ is a dimer form factor, and $f f$ is a function of magnetic form factor:

$$w = gap + \frac{width}{2} (1 + \cos(2\pi Q_x))$$

$$f f = j_0[0] \times e^{(-j_0[1] \times s_2)} + j_0[2] \times e^{(-j_0[3] \times s_2)} + j_0[4] \times e^{(-j_0[5] \times s_2)} + j_0[6] + (1 - \frac{2}{g}) \times s_2 \times$$

$$(j_2[0] \times e^{(-j_2[1] \times s_2)} + j_2[2] \times e^{(-j_2[3] \times s_2)} + j_2[4] \times e^{(-j_2[5] \times s_2)} + j_2[6])$$

$$s_2 = 0.25 \times \left(\frac{Q_x^2}{a^2} + \frac{Q_y^2}{b^2} + \frac{Q_z^2}{c^2} \right)$$

j_0, j_2 are eight elements arrays.

$$f dperp = \sin^2(\pi \times (0.3904 \times Q_x + 0.4842 \times Q_y))$$

Combining these terms, $S(\vec{Q}, E)$ can be written as

$$S(\vec{Q}, E) = s_background + s_incoh + s_ladder$$

The resolution function is given by

$$R(\Delta\vec{Q}, \Delta E) = R_0 \times e^{-([\Delta Q_x \ \Delta Q_y \ \Delta Q_z \ E] M [\Delta Q_x \ \Delta Q_y \ \Delta Q_z \ E])}$$

Where

R_0 is a constant and M is a symmetric matrix.

The problem we are solving here is a challenging problem. We have a huge amount of data points at which to compute the above four dimensional integral. We have tera-scale data produced by the neutron scattering experiment. The points are acquired at approximately 100 angles. For each angle, 10^{10} data points are acquired. Although the problem is tera-scale in terms of data, it needs extreme-scale computing to be solved. This is due to the number of floating point calculations per quadruple integral at each data points, 1000 iteration to optimize the algorithm, and 10 scattering functions. In terms of computation, the problem performs 10^{18} FLOPS.

The large computing scale that our problem has makes it impossible to solve serially in a reasonable amount of time. For this reason, we decide to parallelize the algorithm.

4.1 OBJECTIVES

The main objective of this research is to implement parallel algorithms to solve the neutron scattering problem discussed in the previous section. The implemented algorithms should satisfy:

1. Correctness: should give the correct answer.
2. Efficient: solve the problem in a reasonable amount of time.
3. Scalable: given more computing resources, it will solve the problem faster.
4. Easy: the parameters of integration are easy to supply and track.
5. General: It works for any integrand and variable number of nodes.

5. IMPLEMENTATION DETAILS

The algorithm was implemented in two different versions; serial and parallel. The serial version was implemented and tested at a variety of data points.

5.1 SERIAL VERSION

The serial version of the developed algorithm is used as the base for the parallel version. This version consists of sequential steps to accomplish the required tasks. The following pseudo code summarizes the steps used in this version

Step 1: Generate the data point at which the four dimensional integral should be computed.

Step 2: Generate the quasi-random samples needed by Monte Carlo method to compute the integral.

Step 3: For each data point,

- i. Compute the integral using the quasi-Monte Carlo method.
- ii. Compute the relative error between two successive iterations of the integral value.
- iii. If the relative error is satisfactory or the maximum iterations were reached, then stop and return the value of the integral at the given data point.
- iv. Else, increase the number of random samples.
- v. Go to i.

The flow diagram in Figure 9, illustrates these steps.

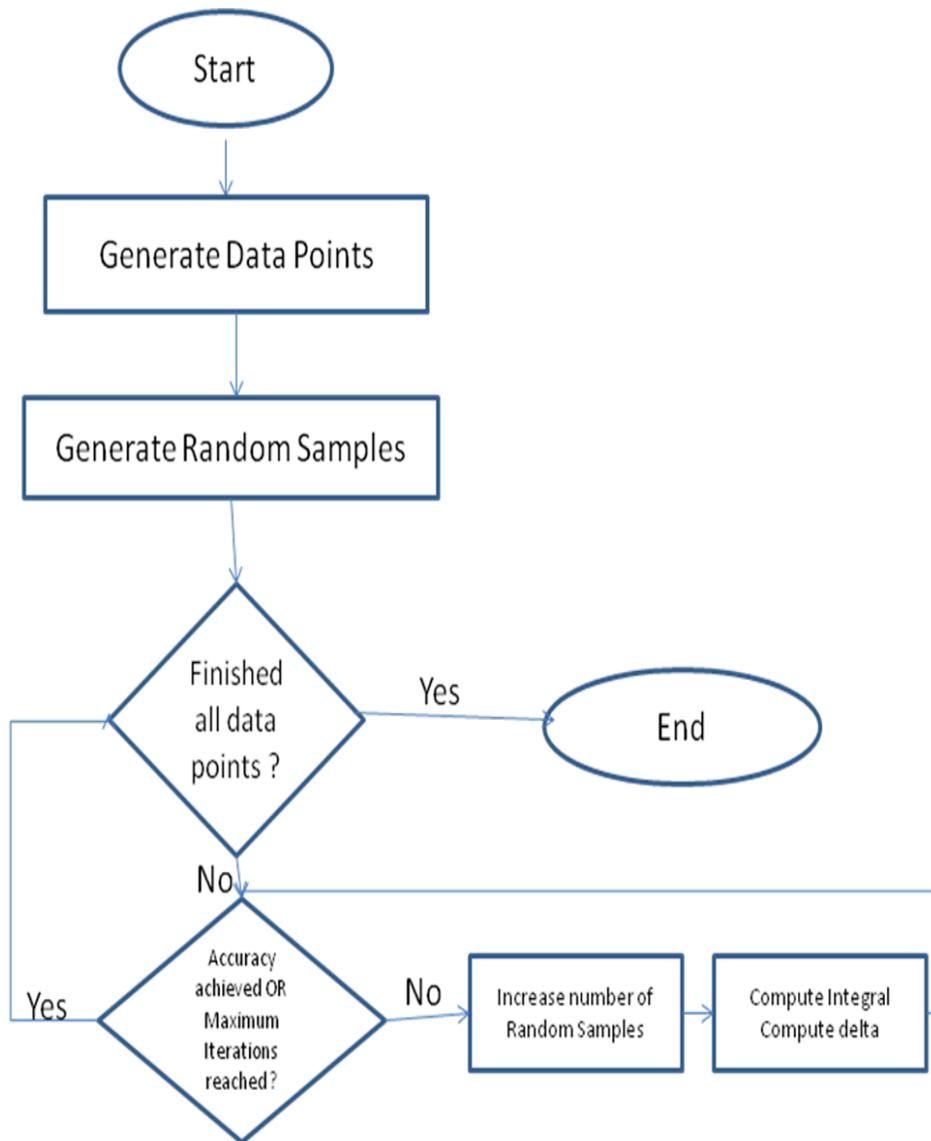


Fig. 9. Serial version flow diagram.

5.2 PARALLEL VERSION

The serial code works efficiently for a small number of points. However, the when number of points gets sufficiently large, parallelization of the serial code is necessary. As discussed before, problem decomposition is a challenge in solving problems in parallel. In our case, it was determined that the most challenging issue was the large number of data points at which the integral needs to be computed. Nevertheless, the evaluation by the quasi- Monte Carlo method using a large number of random samples could be parallelized.

Our algorithm uses the master-slave parallelization scheme. This scheme was preferred over other existing parallelization schemes due to ease of implementation and minimized inter-worker communication. This, in turn, minimizes the execution time. The illustration of the master-slave scheme is shown in Figure 5. Although this scheme is easy to implement, it suffers from the disadvantages of sequential slave creation and heavy communication overhead at the master process.

Our parallel code implements the following pseudo code.

- Step 1:** Master generates the data points.
- Step 2:** Master generates the random samples.
- Step 3:** Master decides the number of data points each slave should receive.
- Step 4:** Master sends random samples to slaves.
- Step 5:** Master sends data points decided in step 3 to slaves.
- Step 6:** Slaves receives random samples.
- Step 7:** Slaves receives assigned data points.
- Step 8:** Each slave computes the integral for all the assigned data points.
- Step 9:** Each slave sends the integral values of the assigned data points to Master.
- Step 10:** Master receives the computed integral values for all data points.
- Step 11:** Master displays execution time and computed integral values.

Figure 10, illustrates the parallel version of the developed algorithm.

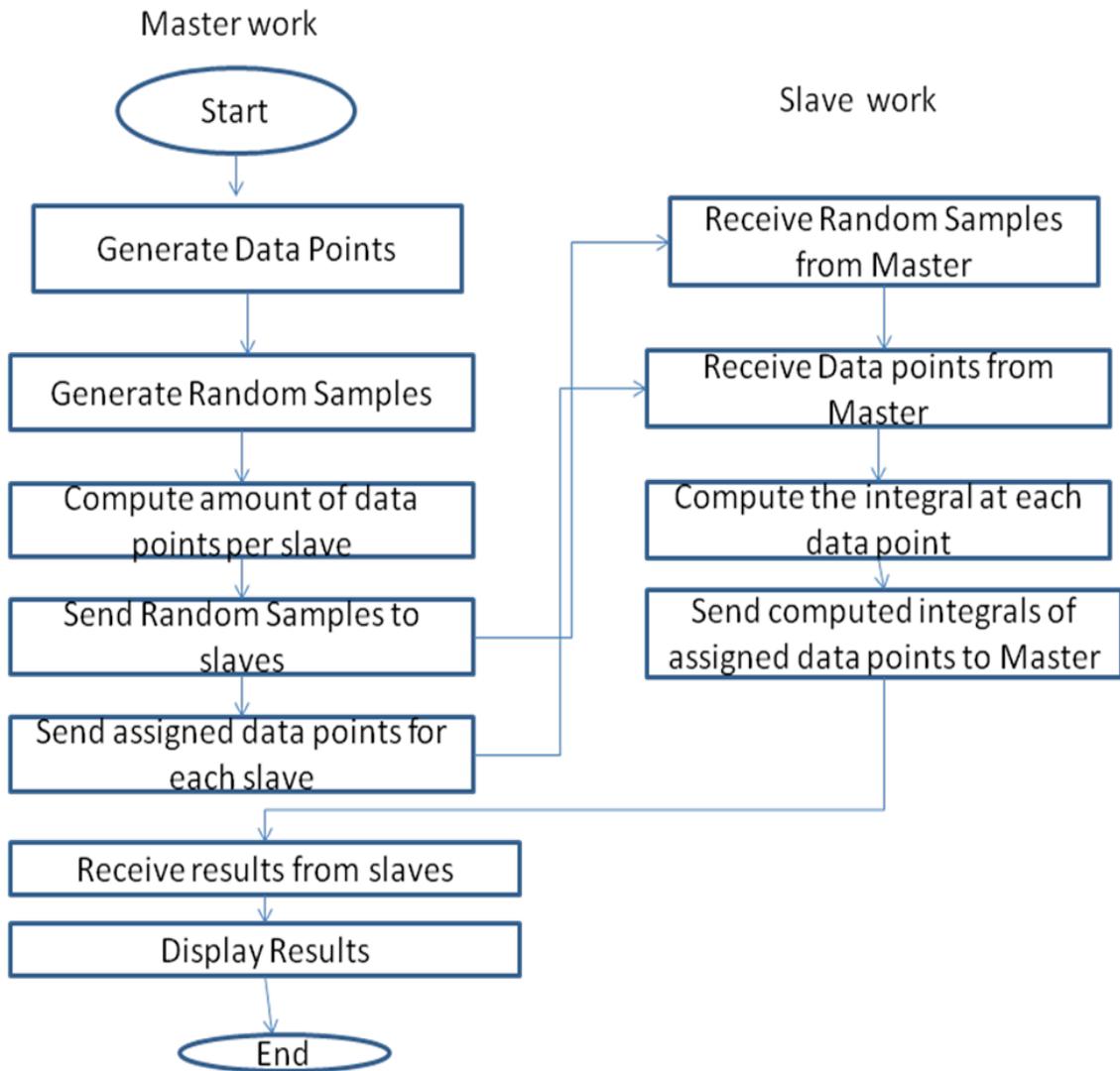


Fig. 10. Parallel version flow diagram.

6. PERFORMANCE EVALUATION

Different experiments have been conducted to evaluate the performance of the developed algorithms, specifically the speed and accuracy of the algorithm. In this section, we present some results obtained from these experiments.

6.1 ACCURACY EVALUATION

For this purpose, we conducted two experiments. The first experiment was conducted using Genz's testing functions listed in Table 1.

Table 1. Genz's testing functions⁹

Function		Name
$f_1(x, y, z, t)$	$\cos(2\pi + x + y + z + t)$	Oscillatory
$f_2(x, y, z, t)$	$\frac{1}{(1+x^2)(1+y^2)(1+z^2)(1+t^2)}$	Product Peak
$f_3(x, y, z, t)$	$\frac{1}{(1+x+y+z+t)^5}$	Corner Peak
$f_4(x, y, z, t)$	$e^{-\frac{1}{2}(x^2+y^2+z^2+t^2)}$	Gaussian
$f_5(x, y, z, t)$	$e^{-(x + y + z + t)}$	Continuous
$f_6(x, y, z, t)$	$\begin{cases} 0 & x, y > 0 \\ e^{(x+y+z+t)} & \text{Otherwise} \end{cases}$	Discontinuous

The integral of each testing function was done analytically and the following true values were obtained. All the testing functions except the third one were calculated over an integration interval of $[-\pi/2, \pi/2]$. The third function was integrated over an interval of $[0, 1]$.

Table 2. True value of testing functions integral.

Function	Integral True Value
f_1	16
f_2	16.26189
f_3	0.00833333
f_4	8.867925
f_5	6.26748
f_6	13.2919

Table 3. Accuracy and execution time (in Seconds) results of testing functions integral.

Function	Mathematica Methods									
	NIntegrate		QM		Trapezoidal		GaussKronrod		QM-Serial	
	Acc.	time (Sec)	Acc.	Time (Sec)	Acc.	time (Sec)	Acc.	time (Sec)	Acc.	Time (Sec)
f_1	0	0.25	0.00006	0.28	0.05055	0.05	0	0.11	0.00019	0.11
f_2	0.00072	0.28	0.00049	0.09	0.00219	0.22	0.00072	2.62	0.00001	0.10
f_3	0	0.70	0.00131	0.218	0.02204	0.11	0	0.50	0.00034	0.11
f_4	.00032	0.33	0.0012	0.15	0.00363	0.03	0.00032	1.06	0.0006	0.13
f_5	0.00506	0.09	0.0048	0.14	0.01315	0.12	0.00506	0.12	0.0058	0.10
f_6	0	0.20	0.0002	0.20	0.03240	0.11	0	0.11	0.00157	0.09

Table 3 shows the accuracy and execution time obtained from our serial version code, different Mathematica methods such as, NIntegrate, QuasiMonteCarlo, Trapezoidal, and GaussKronrodRule. The accuracy in Table 3 was calculated based on the following formula:

$$Accuracy = \frac{|exact - estimated|}{exact}$$

The exact value was taken from Table 2.

For most of the testing functions, there is a noticeable tradeoff between accuracy and execution time. Our code has an acceptable accuracy and reasonable execution time. For functions, f_1 , f_3 , and f_6 , the NIntegrate, and GaussKronrodRule Mathematica methods have better accuracy than our code. On the other hand, their execution times are much longer than our serial version code execution times.

To further test the accuracy of our algorithm, an experiment was conducted to compute the integral which represents the intensity at the given point. In this experiment, Q_y and Q_z were set to -1. The intensity was computed for this setting and the obtained results were plotted as a function of Q_x and E . The results were compared with simulated data point intensity values obtained from a supplied code, as shown in Figure 11. Results obtained from integration re shown in the Figure 12 below. The obtained results are consistent with the results obtained from the supplied intensity simulation code.

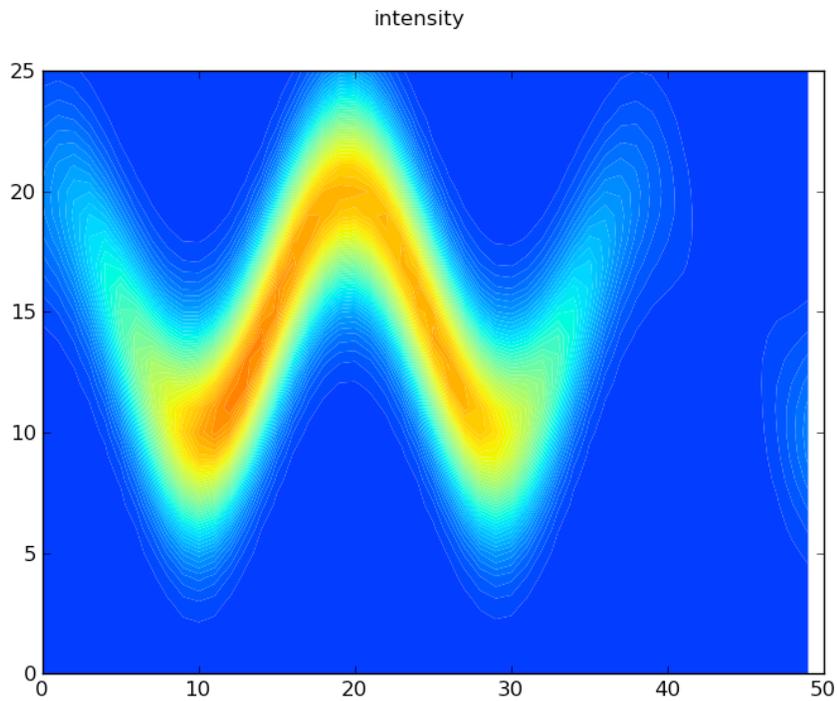


Fig. 11. Intensity Simulated/ Measured at (Q_x , E)

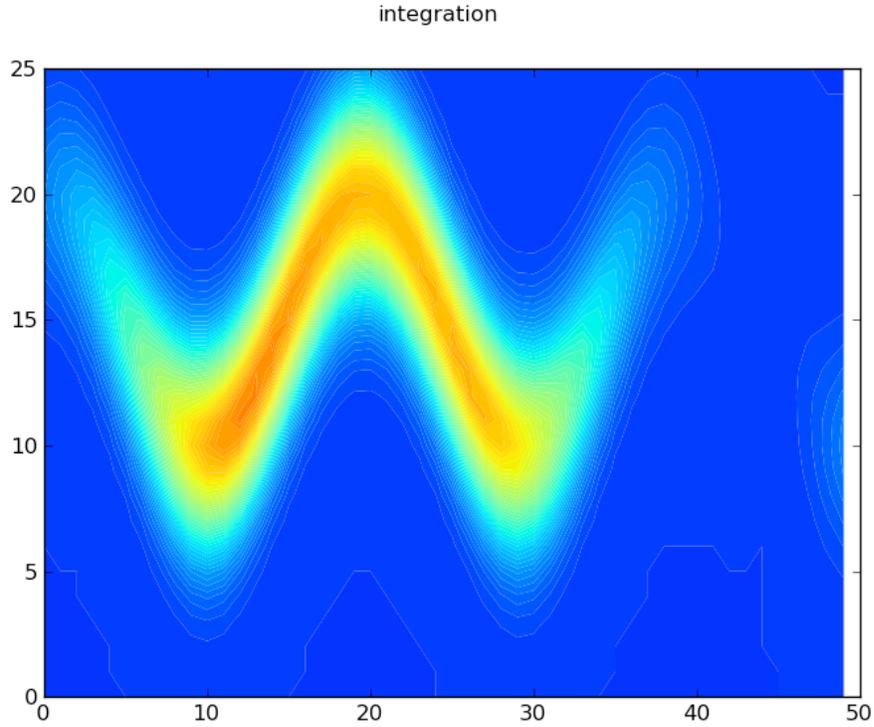


Fig. 12. Intensity predicted by Refined Model at (Q_x, E)

Another experiment was conducted to evaluate the accuracy of our algorithms compared to Mathematica methods. For this experiment, we evaluated the integral of our problem of interest at the point $(Q_0, E_0) = (-1, -1, -1, 0)$. The value of the integral at this point is known. The accuracy and the execution time for different Mathematica methods and our developed algorithm are shown in the Table 4.

Table 4. Accuracy and execution time (in seconds) obtained from integrating $R \cdot S$ at $(-1, -1, -1, 0)$

Mathematica Integration Methods	Accuracy	Time (Seconds)
QuasiMonteCarlo	0.00812738	0.484
Monte Carlo	0.0071752	0.312
AdaptiveQuasiMonteCarlo	0.232069	0.53
MultiDimensionalRule	1.14901E-05	27.8
GlobalAdaptive	0.000014901	28.471
Trapezoidal	0.299859	0.078
Oscillatory	1.15877E-05	27.612
Serial Quasi Monte Carlo	0.003696	0.813

6.2 SCALABILITY EVALUATION

In addition to the experiments for accuracy and speed of the serial version of the code, we conducted experiments to test the scalability of the parallel code. For this purpose, we conducted two different experiments. The first experiment was conducted to study the scalability of the code at a number of data points. We used 160 processes, and changed the number of data points to cover the range from 5×10^4 to 10^8 data points, and the execution time was measured. The results obtained from this experiment are shown in Figure 13. It can be seen that increasing the number of data points causes a linear increase in execution time.

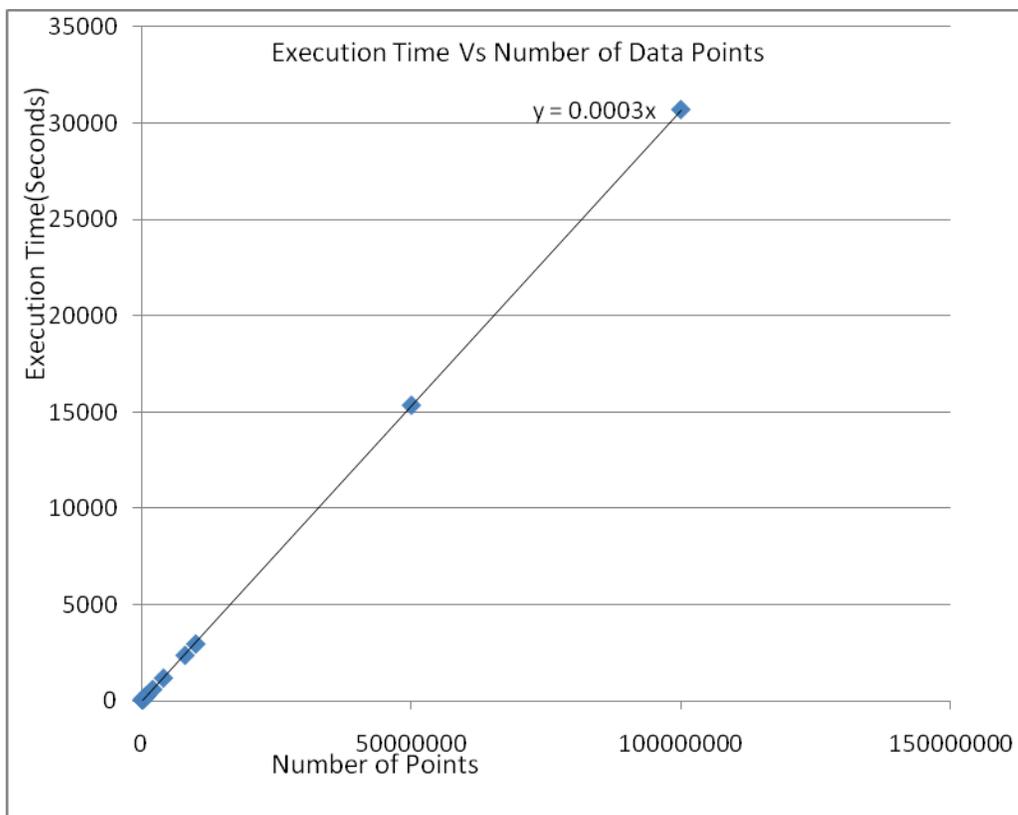


Fig.13. Execution time Vs number of data points.

The second experiment was conducted to test the scalability of the parallel code with the number of processes. Using 3×10^5 data points, we changed the number of processes to cover the available range (8 to 160 processes), and we measured the execution time. The results obtained from this experiment are shown in Figure 14. It can be seen from the figure that the execution time fits to a power function. A linear execution time was expected. The total execution time includes communication time, time to generate Sobol's sequence, and the

computation time. The computation time is expected to be linear, but the communication time and Sobol's sequence generation time are much larger than the computation time which may cause the total execution time to behave like power function. Figure 14 below shows that between 8 processes and 32 processes, the execution time is linear. For number of processes larger than 32, the execution time starts to be steady.

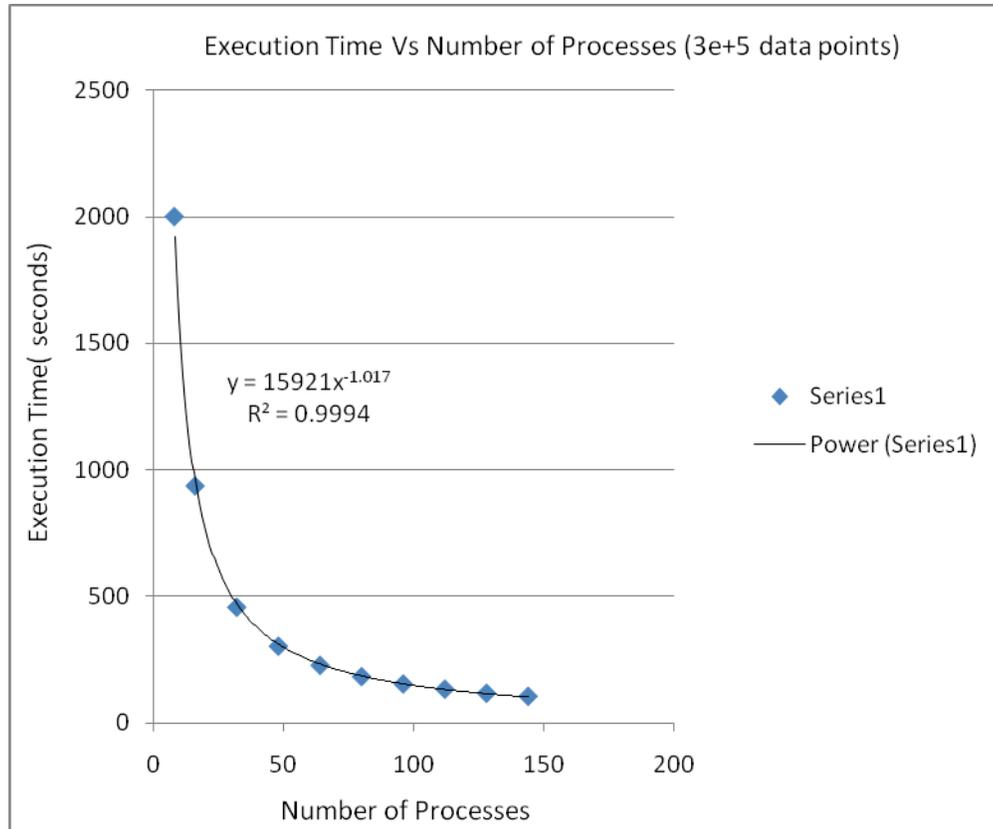


Fig.14. Execution time Vs number of processes for 300, 000 data points.

The parallel code was run on Oak Ridge Institutional Cluster. The specifications of OIC are listed in Appendix B.

7. CONCLUSIONS AND FUTURE WORK

Serial and parallel algorithms were implemented to compute the four dimensional integral for the neutron scattering application. The implemented algorithms were tested for correctness by using Genz's functions and Mathematica's integration methods. The implemented parallel algorithm was tested for scalability. The execution time versus the size of data points was measured. Results show that the algorithm behaves linearly as the size of the data is increased and as a power function as the number of processes is increased.

The implemented algorithm with up to 160 processes could process 10^8 data points efficiently. More work is needed to extend the capability of the implemented algorithm to process all 10^{12} data points generated by the neutron scattering experiment.

Our implemented algorithm uses the single level MPI technique. That is, the master is responsible for all task management. One idea to deal with huge data size is to develop a mixture of MPI and OpenMP algorithms. That way, each process forks out the parallelized tasks into multithreads which is expected to achieve faster execution. Another way to approach this problem is to implement multilevel MPI algorithms. In multilevel, the master will divide the load and send it to different processes where each process behaves as a master for another group of processes. This could make it easier to process more data points and could result in a faster implementation. In both suggested cases, correctness and scalability are important to test.

As discussed previously, problem decomposition is a big challenge in writing parallel programs. In our algorithm, we used data point decomposition. The problem has a potential for more nested decomposition. As Monte Carlo methods need random samples to compute the integral; the integral at each point could be computed by dividing the random numbers between different processes to compute the sub-sums. This method could make the per data point computation time smaller. Significant attention should be paid to scalability and correctness.

8. REFERENCES

1. Amos G. Anderson, et al, *Quantum Monte Carlo on graphical processing units*, Computer Physics communications, volume 177, pages 298-306, 2007.
2. R. Schürer. *Parallel High-dimensional Integration: Quasi-Monte Carlo versus Adaptive Cubature Rules*, Proceedings of the International Conference on Computational Science – ICCS 2001, volume 2073 of *Lecture Notes in Computer Science*, pages 1262–1271. Springer-Verlag, San Francisco, CA, USA, May 2001
3. R. Schürer. *Adaptive Quasi-Monte Carlo Integration Based on MISER and VEGAS*. In H. Niederreiter, editor, *Monte Carlo and Quasi-Monte Carlo Methods 2002*, pages 393–406. Springer-Verlag, February 2004.
4. J. Bull and T. Freeman, *Parallel algorithms for multi-dimensional integration*, Parallel and Distributed Computing Practices, 1(1):89–102, 1998.
5. P. Zinterhof, *High Dimensional Integration: New Weapons Fighting the Curse of Dimensionality*, Physics of Particles and Nuclei Letters, Vol. 5 No. 3, pp. 145-149, 2008.
6. Thomas Gerstner and Michael Griebel, *Numerical Integration using sparse grids*, Numerical Algorithms, Vol. 18 , pp. 209-232, 1998.
7. S. C. Kugele and L. T. Watson, *Multidimensional Numerical Integration for Robust Design Optimization*, Proc. 45th ACM Southeast Conf. pp. 472-477, 2007.
8. Art B. Owen, *The dimension distribution and Quadrature test functions*, Statistica Sinica, Vol. 13, pp. 1-17, 2003.
9. L. Smith and M. Bull, *Development of mixed model MPI/OpenMP applications*, Scientific Programming, Vol. 9, No. 2-3, pp. 83-98, 2001.
10. OpenMp illustration, retrieved from: http://en.wikipedia.org/wiki/File:Fork_join.svg
11. Bailey, David H., & Borwein, Jonathan M., *Highly Parallel, High-Precision Numerical Integration*. Lawrence Berkeley National Laboratory: Lawrence Berkeley National Laboratory. LBNL Paper LBNL-57491, 2005.
12. Neutron Scattering, retrieved from: http://neutron.magnet.fsu.edu/neutron_scattering.html.
13. ORNL Institutional Cluster. <https://oic.ornl.gov/>.
14. Monte Carlo Techniques, retrieved from: http://einstein.drexel.edu/courses/PHYS405/Monte_carlo/index.html.

APPENDIX A . INTEGRATION VARIABLES TRANSFORMATION

In order to solve the integral

$$I^{calculated}(\vec{Q}_0, E_0) = \int_{\Delta\vec{Q}=-\infty}^{\infty} \int_{\Delta E=-\infty}^{\infty} S(\vec{Q}_0 + \Delta\vec{Q}, E_0 + \Delta E) \times R(\Delta\vec{Q}, \Delta E) d\Delta\vec{Q} d\Delta E$$

A variables transformation is necessary.

Recall $\Delta\vec{Q} = (\Delta Q_x, \Delta Q_y, \Delta Q_z)$

Let

$$\Delta Q_x = \tan(\varphi), -\frac{\pi}{2} \leq \varphi \leq \frac{\pi}{2}$$

$$\Rightarrow d\Delta Q_x = \frac{1}{\cos^2 \varphi} d\varphi$$

$$\Delta Q_y = \tan(\gamma), -\frac{\pi}{2} \leq \gamma \leq \frac{\pi}{2}$$

$$\Rightarrow d\Delta Q_y = \frac{1}{\cos^2 \gamma} d\gamma$$

$$\Delta Q_z = \tan(\eta), -\frac{\pi}{2} \leq \eta \leq \frac{\pi}{2}$$

$$\Rightarrow d\Delta Q_z = \frac{1}{\cos^2 \eta} d\eta$$

$$\Delta E = \tan(\varepsilon), -\frac{\pi}{2} \leq \varepsilon \leq \frac{\pi}{2}$$

$$\Rightarrow d\Delta E = \frac{1}{\cos^2 \varepsilon} d\varepsilon$$

Substitute these values in the integral

$$I^{calculated}(\vec{Q}_0, E_0) = \int_{\varphi=-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{\gamma=-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{\eta=-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{\varepsilon=-\frac{\pi}{2}}^{\frac{\pi}{2}} S(Q_x + \tan(\varphi), Q_y + \tan(\gamma), Q_z + \tan(\eta), E_0 + \tan(\varepsilon)) \times R(\tan(\varphi), \tan(\gamma), \tan(\eta), \tan(\varepsilon)) \frac{d\varphi}{\cos^2 \varphi} \frac{d\gamma}{\cos^2 \gamma} \frac{d\eta}{\cos^2 \eta} \frac{d\varepsilon}{\cos^2 \varepsilon}$$

The implemented Quasi Monte Carlo uses the Sobol's sequences to generate random samples. The generated samples are numbers in the interval [0,1]. To compute the integral, we need to transform variables of integration.

Let

$$\varphi = -\frac{\pi}{2} + \pi\alpha$$

$$\Rightarrow d\varphi = \pi d\alpha$$

$$\gamma = -\frac{\pi}{2} + \pi\beta$$

$$\Rightarrow d\gamma = \pi d\beta$$

$$\eta = -\frac{\pi}{2} + \pi\lambda$$

$$\Rightarrow d\eta = \pi d\lambda$$

$$\varepsilon = -\frac{\pi}{2} + \pi\xi$$

$$\Rightarrow d\varepsilon = \pi d\xi$$

$$I^{\text{calculated}}(\vec{Q}_0, E_0) = \int_{\alpha=0}^1 \int_{\beta=0}^1 \int_{\lambda=0}^1 \int_{\xi=0}^1 \frac{S(Q_x + \tan(-\frac{\pi}{2} + \pi\alpha), Q_y + \tan(-\frac{\pi}{2} + \pi\beta), Q_z + \tan(-\frac{\pi}{2} + \pi\lambda), E_0 + \tan(-\frac{\pi}{2} + \pi\xi)) \times R(\tan(-\frac{\pi}{2} + \pi\alpha), \tan(-\frac{\pi}{2} + \pi\beta), \tan(-\frac{\pi}{2} + \pi\lambda), \tan(-\frac{\pi}{2} + \pi\xi))}{\cos^2(-\frac{\pi}{2} + \pi\alpha) \cos^2(-\frac{\pi}{2} + \pi\beta) \cos^2(-\frac{\pi}{2} + \pi\lambda) \cos^2(-\frac{\pi}{2} + \pi\xi)} da db d\lambda d\xi$$

APPENDIX B. ORNL INSTITUTIONAL CLUSTER (OIC)



Fig. B.1 . ORNL Institutional Cluster (OIC)¹³

Our code was tested on ORNL Institutional Cluster (OIC) shown in Figure B.1. OIC has different types of nodes. The NSSD “Block” that we used to run our code is a block of 22 nodes, with 3.11 peak TFLOP, and 16GB memory per node, but that is not the entire OIC. More information about OIC is available at <http://oic.ornl.gov>.