A Global Address Space Approach to Automated Data Management for Parallel Quantum Monte Carlo Applications

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Sravya Tirukkovalur, B.Tech.

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Thesis Committee:

Dr. P. Sadayappan, Advisor
Dr. Srinivasan Parthasarathy
ABSTRACT

Quantum Monte Carlo is a large class of computer algorithms that simulate quantum systems with the idea of solving the quantum many-body problem. Typical parallel quantum Monte Carlo (QMC) applications use very large spline interpolation tables that are unmodified after initialization. Although only a small fraction of the table may be accessed by each parallel thread/process in a window of execution, the accesses are quite random. Hence current implementations of these methods typically use replicated copies of the entire interpolation table at each node of a parallel computer. This limits scalability since increasing the number of processors does not enable larger systems to be run.

In this thesis, we take an automated data management approach which enables existing QMC codes to be adapted with minimal changes to significantly enhance the range of problem sizes that can be run. We primarily use Global Arrays partitioned global address space model to provide efficient distributed, shared storage and further the implementation is optimized by intelligent replication, locality, and data reuse management mechanisms. A transparent software caching mechanism is designed and built on the Global Arrays PGAS (Partitioned Global Address Space) programming model, to enable QMC codes to overcome their current memory limitations in running large-scale simulations. The new GA-read-cache (GRC) has been used to enhance the scalability of QWalk, one of the popular QMC applications.
I dedicate this work to my parents Dr. T. Vinoda and T.N. Venkata Swamy
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. P. Sadayappan, for his invaluable guidance and support. I would also like to thank Dr. Srinivasan Parthasarathy for his support.

I would like to thank all my friends for the technical inputs, especially Dr. James Dinan, for his persistent guidance and encouragement.

I am grateful to the Department of Computer Science and Engineering for supporting me financially during my Masters program.

Finally I would like to thank my parents for providing me with this excellent opportunity.
VITA

2009  ................. B.Tech, Computer Science and Engineering, 
      SVIT, JNTU 
      Hyderabad, AP, India.

2009-2011  ................. Masters Student, 
             Department of Computer Science and Engineering, 
             The Ohio State University.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

   High Performance Computing    Dr. P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapters</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Related work</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Road map</td>
<td>3</td>
</tr>
<tr>
<td>2. Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Global Arrays Toolkit</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 GA</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 ARMCI</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Quantum Monte Carlo (QMC)</td>
<td>5</td>
</tr>
<tr>
<td>3. QMC and Einspline</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Einspline library and Spline data</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Setup functions</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Evaluation functions</td>
<td>10</td>
</tr>
</tbody>
</table>

vi
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>QWALK: Silicon 250, 10 nodes</td>
</tr>
<tr>
<td>5.1</td>
<td>QWALK: Si 3x3x3- GRC</td>
</tr>
<tr>
<td>5.2</td>
<td>QWALK: Si 3x3x3- Selective Caching</td>
</tr>
<tr>
<td>5.3</td>
<td>QWALK: Si 250- Process Groups</td>
</tr>
<tr>
<td>5.4</td>
<td>QWALK: Si 250- Process Groups +GRC</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Typical Diffusion QMC Procedure. 6</td>
</tr>
<tr>
<td>4.1</td>
<td>Global Arrays - Can still access the highlighted memory unit as A(3,5) 13</td>
</tr>
<tr>
<td>4.2</td>
<td>Minicube access 14</td>
</tr>
<tr>
<td>4.3</td>
<td>Code snippet illustrating how the 4-dimensional mini-cube is used for interpolation inside an einspline evaluation call. 14</td>
</tr>
<tr>
<td>5.1</td>
<td>Reuse Distance 19</td>
</tr>
<tr>
<td>5.2</td>
<td>Reuse Distance Algorithm 20</td>
</tr>
<tr>
<td>5.3</td>
<td>Reuse Distance of SI 250 input 20</td>
</tr>
<tr>
<td>5.4</td>
<td>Aligned Multi Dimensional Bricks 21</td>
</tr>
<tr>
<td>5.5</td>
<td>Minicube from Bricks 24</td>
</tr>
<tr>
<td>5.6</td>
<td>Minicube straddling between bricks for brick sizes 2 and 4 (in 2D plane) 26</td>
</tr>
<tr>
<td>5.7</td>
<td>Reuse Distance Analysis for Brick=2 on Input SI.3x3x3 27</td>
</tr>
<tr>
<td>5.8</td>
<td>Brick size Analysis for QWalk Input: si3x3x3 29</td>
</tr>
<tr>
<td>5.9</td>
<td>Small group size / Small Cache size 32</td>
</tr>
<tr>
<td>5.10</td>
<td>Big group size / Big Cache size 33</td>
</tr>
</tbody>
</table>
Scientific computing is one of the important application domains for high performance computing. Monte-carlo methods are an important and widely-used class of sampling-based techniques for solving problems for which analytical solutions are unavailable. Such methods have also gained currency within the domain of Quantum chemistry - in particular, Quantum Monte Carlo (QMC) simulations are a method for solving Schrödinger’s equation, which is a necessary prerequisite to calculating different properties of many-electron systems. These applications are highly compute-intensive, as evidenced by their popularity on many large super-computing systems[4]. Some of the computational burden of QMC simulations is eased by table lookups on interpolation tables (a.k.a. spline tables). Large-scale QMC simulations can require more memory than is present on a single node, but because of lack of support for distributed memory access in existing implementations, such large-scale QMC simulations could not be carried out so far.

The main contribution of this thesis is to provide efficient, distributed data management for large-scale QMC applications using the Global Arrays (GA) toolkit, and thereby enabling efficient execution of large-scale QMC simulations. Popular QMC applications like QWalk and QMCPack use Einspline library, a C library which for interpolating splines, for all its interpolation purposes. Hence we enhance einspline library, there by enabling
all QMC applications which use einspline library to use our enhancements transparently. Further, along with enabling bigger simulations, we also introduce several optimizations on top of the basic GA extension, such as:

1. A Global Array Read-only Cache (GRC), a transparent software caching mechanism, is built on Global Arrays PGAS (Partitioned Global Address Space) programming model to exploit data reuse.

2. We utilize the aggregate memory of all nodes in the cluster by using Process Groups. We divide the processes into groups and create a replica of the data for each process group so as to minimize contention and exploit the aggregate memory of the cluster.

Our system delivers promising speedups compared to the existing implementation, both when using each optimization individually as well as when using all the optimizations together. In order to enable transparent and easy usage, our contributions have been implemented inside the Einspline library, which is used by many different QMC applications.

1.1 Related work

Kim et al.[11] have developed hybrid MPI/OpenMp implementations of QMC, which utilizes the aggregate memory of the node rather than per core memory. In this implementation, all the processes on a node share a copy of the Spline table. Effectively, by using this implementation, problem size is limited by the node memory rather than per core memory. However, this approach does not utilize the aggregate memory of the cluster.

Smith et al.[12] have discussed in more detail about such Hybrid MPI/OpenMp implementations in general (not specific to QMC). They demonstrated that Hybrid implementations can be especially helpful when the MPI code suffers from scalability problem due to
memory constraints. However, their results also state that this approach might not be the most effective for all the codes.

Though work has been done to utilize the node memory rather than per core memory, the idea of utilizing the aggregate memory of the cluster has not been studied before. This approach can significantly increase the upper limit on the problem sizes which can be simulated using QMC.

1.2 Road map

The rest of this thesis is organized as follows. A brief overview of Global Arrays Toolkit is provided in Section 2.1 and also Quantum Monte Carlo Simulations are explained from Computer science point of view in Section 2.2. Further Qwalk, one of the popular QMC, and Einspline library, the library which is used by popular QMC software packages for interpolation purposes, are explained in Section 3. We also talk about the scalability problem of existing implementations and the motivation for this thesis.

Our contributions are detailed in Chapters 4 and 5, where our enhancements to existing QMC implementations, which allow much bigger simulations to be run, are discussed. We also provide experimental results together along with the explanation of our approaches.

We provide an Appendix, which provides all the information for the users to use the enhanced einspline.
CHAPTER 2

BACKGROUND

2.1 Global Arrays Toolkit

2.1.1 GA

PGAS (Partitioned Global Address Space) is a parallel programming model which utilizes distributed global address space of a cluster. Each process in this model has a local address space and a global address space. PGAS models such as Global Arrays (GA) provides get and put functions to copy the data between global and local address spaces. It uses low latency rDMA operations supported by many high performance interconnects.

The Global Arrays toolkit[5] provides an efficient and portable shared-memory programming interface that extends existing languages like C and Fortran with library based interface to a global address space. Each process can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays. It automatically generates any needed communication thus providing a shared memory like abstraction to the programmer.
2.1.2 ARMCI

The Aggregate Remote Memory Copy (ARMCI)[6] library provides an efficient and widely portable remote memory access (RMA) operations (one-sided communication) optimized for both contiguous and noncontiguous data transfers. It also has support for atomic and mutual exclusion operations. GA uses ARMCI as the main communication layer. It relies on the global memory management provided by ARMCI(malloc and free), its communication libraries, and compilers. ARMCI is very efficient as it exploits the native network communication interfaces and system resources to achieve the best possible performance of the remote memory access/one-sided communication. It also exploits high-performance network protocols on clustered systems.

2.2 Quantum Monte Carlo (QMC)

Quantum Monte Carlo (QMC) simulations are a method of solving Schrodinger’s equation, in order to calculate properties of many-electron systems. They are popular applications on the largest super-computing systems. For example, DCA++ got the Gordon Bell prize for peak performance at SC08[4]. The most widely used QMC software packages are QMCPACK, QWalk, CASINO, and CHAMP. In this section, we discuss below a summary of the typical computational structure of QMC and the difficulties caused by large memory requirements. Further details can be found in[1][2][3]

Consider a molecular system with $n$ electrons. The positions of this set of electrons can be considered a point in $3n$ dimensional space. A quantum state of the system is a point in $\mathbb{R}^{3n}$. The wave function $\psi$ is a mapping from $\mathbb{R}^{3n}$ to complexes, where $|\psi(x)|^2$ can be interpreted as the probably density of finding the system in state $x$. The goal of QMC
1. Initialize \( W \) random walkers distributed according to the trial wave function.

2. Repeat the following calculations until variance is sufficiently small

(a) Repeat the following calculations \( N\text{StepsPerBlock} \) times

i. Repeat the following for each random walker \( i \)

A. Propose a new state \( x_i \) for the random walker and accept it or reject it based on Metropolis probabilities, and compute a weight for its new state, based on the value of the trial wave function at \( x_i \)

B. Compute the energy and any observables, weighted by the computed weight

C. Terminate walkers or split walkers into multiple ones

(b) Update the trial energy based on the block averages for most recent block for all the walkers

(c) Renormalize the number of walkers so that there are roughly \( W \) of them

---

Figure 2.1: Typical Diffusion QMC Procedure.

Simulations is to determine the wave function, from which other quantities of interest, such as energy, can be computed.

QMC accomplishes this through a Metropolis sampling process. It tracks the states of \( W \) random walkers. Each random walker represents a quantum state consisting of the positions of \( n \) electrons. A random walker starts at an initial state \( y_0 \). Then, in each step of the sampling process, a random change in this state is proposed, which is accepted or rejected. The new state is considered another sample. However, the samples are not independent. So, statistics are obtained for a number of blocks, where each block consists of several steps. The statistics of each block can be considered independent. The basic approach described here leads to a high statistical error. The technique of importance sampling, which uses an approximation to the true wave function, called a trial wave function, is used to reduce the statistical error.
These calculations are typically parallelized by having a distinct set of walkers on each processor. Communication is involved only after each block of calculations, and is small except on extremely large parallel machines. On large machines, load balancing too can become an issue.

Much of the computational effort in QMC arises in evaluating the energy using the trial wave function. Different bases differ in the amount of computational effort and memory requirement. The computational effort required for computationally intensive bases, such as the plane-wave basis, can be reduced using table look-ups and interpolation. The simulation of large systems can require more memory than available in a given computational node. Prior to this work, such QMC simulations could not be carried out. In this work, the Einspline interpolation library is enabled to distribute the table amongst the multiple processes. This library is used by many QMC packages, including QMCPACK and QWalk. Thus, this work enables these applications to scale to larger physical systems.

While both the software packages use essentially the same QMC methods, implementation differences bring out different computational issues that need to be addressed. For example, QMCPACK uses hybrid MPI-OpenMP parallelization. Thus, a single process on a node can use all the cores of a node and only one copy of the table needs to be maintained per node. If the table size exceeds the memory of a node, then the table will need to be distributed across multiple nodes. QWalk, on the other hand, uses only MPI for parallelization, and each core will require its own copy of the table (prior to this work), reducing the largest table size that can be used without wasting the computational resources available on a node. The implementation provides it the ability to both, distribute tables across a single node, to be shared by all processes on a node, and across nodes, if the memory on a single node is not sufficient.
The memory access pattern for the table entries too varies between the two codes. Note that the memory access pattern of different random walkers is quite random, but there is some temporal correlation between entries accessed by a single random walker. Certain table access patterns, which are correlated, are batched in QMCPACK, but not on QWalk. When a table is distributed across nodes, this can have an effect on performance.
A common aspect of many quantum Monte Carlo (QMC) methods is spline interpolation using large multidimensional interpolation tables. These interpolation tables are created at the start of the Monte Carlo simulation and are used in a read-only mode throughout the execution. The time required to create the tables is negligible compared to the total Monte Carlo simulation time, but the memory space required to hold the table entries is a significant constraint that limits the size of systems that can be modeled using QMC codes. The size of the interpolation tables grows with the size of the simulated system and cannot be effectively partitioned among the nodes of a parallel QMC execution because the accessed points in the spline tables are random and impossible to characterize prior to the actual simulation.

3.1 Einspline library and Spline data

Einspline is a “C library for the creation and evaluation of interpolating cubic basis splines (B-splines) in 1, 2, and 3 dimensions.” [7]. QMC softwares such as QWalk and QMCPack use this library for all their interpolation purposes. Since, the memory required for the interpolation table is the main constraint that is currently limiting the scalability of QMC applications, modifying the relevant parts of Einspline library is a clean way to
address this issue. We now give a brief overview of the relevant API calls of the Einspline library and how they process the spline data.

### 3.2 Setup functions

In the setup stage, initial configuration files are read in and trial wave function data is built using create and set functions of the Einspline library.

```c
multi_UBspline create_multi_UBspline() {
    // Read orbital configuration files, return a handle to the spline data.
}
```

```c
void set_multi_UBspline(multi_UBspline* spline, int num, complex_double* data) {
    // Compute the spline data from the initial orbital configurations.
}
```

The above two functions are called only at the start of the simulation, and set up the stage for the rest of the simulation.

### 3.3 Evaluation functions

Much of the computational effort of QMC lies in the evaluation calls provided by the Einspline library, which evaluate energy using trial wave function data. These are described below.

```c
void eval_multi_UBspline(multi_UBspline *spline, double x, double y, double z, complex_double* restrict vals) {
    // Evaluate energy using trial wave function data.
}
```

```c
void eval_multi_UBspline_vg (multi_UBspline *spline, double x, double y, double z, complex_double* restrict vals) {
    // Evaluate energy using trial wave function data.
}
```

1Note that there for each core task, there exist multiple functions that implement essentially the same task except that each function is specialized to handle a certain datatype. For example, there exist multiple versions of the `create_multi_UBspline()` function for complex double, double and other datatypes.
void eval_multi_UBspline_vgl (multi_UBspline *spline, double x, double y, double z, complex_double* restrict vals, complex_double* restrict grads)
void eval_multi_UBspline_vgh (multi_UBspline *spline, double x, double y, double z, complex_double* restrict vals, complex_double* restrict grads, complex_double* restrict lapl)
void eval_multi_UBspline_vgh (multi_UBspline *spline, double x, double y, double z, complex_double* restrict vals, complex_double* restrict grads, complex_double* restrict hess)

These functions evaluate different properties such as energy of an electron at a given three-dimensional location or point in space. These are the functions in which the major part of the computational expense of the simulation is incurred. Furthermore, these are the only functions which actually access the spline data, meaning that it is sufficient to modify these functions in order to incorporate GA into the simulation.
CHAPTER 4

AUTOMATIC DATA MANAGEMENT

In this chapter, we explain the core contributions of this thesis i.e. how we enable QMC applications to scale to large datasets by incorporating a variety of data management techniques. First, we describe how we incorporate GA into the einspline library in Section 4.1. Next, we describe various optimizations to this basic setting in the subsequent sections.

4.1 Incorporating GA in Einspline

We modify the Einspline library to use GA for storing the spline data which would be shared by all the processes in the QMC run. Thus each parallel process need not hold the spline data copy of its own, but can access the shared and distributed array. Figure 4.1 explains the use of Global Arrays. Though the Global Array ”A” is spread over 8 nodes, we need not know the location of a memory unit in order to be able to access it. The memory unit highlighted in the figure can be directly accessed as A(3, 5). We use this model to distribute the spline data, in such a way that any QMC application which uses einspline library for interpolation can transparently use the modified einspline library with GA support. ²

²The one exception is in the setup phase of QWalk, where we have added additional parallelization; this is not essential but makes the setup process much faster.
Figure 4.1: Global Arrays - Can still access the highlighted memory unit as A(3,5)

4.2 Chunk GA get

Observation: In each evaluation call of a point in 3-dimensional space, a 4-dimensional mini-cube of spline data of size $4 \times 4 \times 4 \times \text{numsplines}$ is used for interpolation as shown in figure 4.2. Basically, these are the control points used for interpolation. The code snippet in figure 4.3 shows how this mini-cube is accessed from within the evaluate functions. In the 4-dimensional mini-cube, the first three dimensions correspond to spatial dimensions and the fourth dimension corresponds to spline data. For each point in 3-dimensional space, data corresponding to all the splines ($4^{th}$ dimension) at that point are utilized for interpolation.

Such a pattern of data access is well suited to the get-compute-put model of the GA toolkit. Hence, rather than performing a point-by-point access of the Global Array, we do a get of the required mini-cube, and then use it for interpolation. This reduces communication overhead in comparison to a naive point-by-point access strategy.
Figure 4.2: Minicube access

```
for (int i=0; i<4; i++)
    for (int j=0; j<4; j++)
        for (int k=0; k<4; k++) {
            ...
            complex_double* coefs
                = spline->coefs + ((ix+i)*xs + (iy+j)*ys + (iz+k)*zs);
            for (int n=0; n<spline->num_splines; n++) {
                ... = coefs[n];
                ...
            }
            ...
        }
```

Figure 4.3: Code snippet illustrating how the 4-dimensional mini-cube is used for interpolation inside an einspline evaluation call.

### 4.3 API

Instead of directly implementing the functionalities inside existing Einspline code, we used a more general strategy of providing a general-purpose API and then using that API to make minimal changes to Einspline. This allows us to easily enable or disable GA support within Einspline. We describe this API next. Note: spline data is also often referred to
as the coefficients matrix inside einspline, hence the use of “coefs” often in the function signatures and datatypes.

coefs_t is a datatype for storing meta-data regarding specific Global Arrays that store spline data.

Basic calls:

1. coefs_t* coefs ga create(Mx,My,Mz,nsplines);
   This function creates the GA to hold the spline data. This function is called from within the create_multi UB spline function of einspline.

2. coefs ga put spline (coefs_t*,num);
   This function puts the spline data into GA, after it is computed from the initial orbital configurations. This is called from within the set_multi UB spline function of einspline.

3. coefs ga destroy (coefs_t*);
   This function is used at the end for GA clean-up purposes.

4. void coefs ga get (coefs_t *coefs, int x, int y, int z, void *minicube);
   This returns the minicube corresponding to the 3-dimensional location (x, y, z).

4.4 Experimental Results

Input: si250 is a system with dimensions of its spline table being 79 × 79 × 79 × 250. As the data is all complex double type, spline table takes up 1.6 GB memory. For all the experiments, Configuration is as follows: nblocks=1; nsteps=1; ndecorr=1; nwalkers=1.

System: The per node memory on our system is 4GB, and each machine has 8 cores.
As spline table of si 250 input is 1.6GB, with the traditional parallel QWalk each process needs 1.6GB for spline table. Hence for the systems where per node memory is less than 1.6GB, this simulation cannot be run. Though in our system each machine has eight cores, not more than 2 cores can be utilized with the traditional QWalk. (If we had use 3 cores, we will need $3 \times 1.6GB = 4.8GB$ memory per node).

Accuracy of the simulation depends on number of total walkers, which is equivalent to number of parallel processes $\times$ walkers per parallel process. When number of parallel processes is increased, walkers per parallel task can be proportionally reduced, there by speeding up the simulation maintaining the same accuracy. Hence utilizing more number of parallel units can speed up the simulations drastically.

The results are shown in the table 4.1. With the original version only 2 cores out of 8 cores could be used. Where as with the enhanced einspline library all the 8 cores per node have been used which speeded up the simulation very well. The experiment is run using 10 nodes. As shown in the table the Qwalk with traditional einspline took 207 secs utilizing 20 parallel units, where as the enhanced einspline took 66 secs by utilizing all the available cores(80).

**Input:** si1010 is a system with dimensions of its spline table being $79 \times 79 \times 79 \times 2000$. As the data is all complex double type, spline table takes up 13 GB memory. For all the experiments, Configuration is as follows: nblokk=1; nsteps=1; ndecorr=1; nwalkers=1.

**System:** The per node memory on our system is 4GB, and each machine has 8 cores.

<table>
<thead>
<tr>
<th>Original</th>
<th>207 secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA</td>
<td>66 secs</td>
</tr>
</tbody>
</table>

Table 4.1: QWALK:Silicon 250 ,10 nodes
As spline table of si 1010 input is 13GB, with the traditional parallel QWalk each process needs 13GB for spline table. Hence for the systems where per node memory is less than 13GB, this simulation cannot be run. As in our system per node memory is just 4GB, Qwalk with traditional einspline cannot run this input. With the enhanced einspline, we could run this simulation and it finished in 11226 secs.
5.1 GRC (Global Array Read only Cache)- A Software Cache

Though access pattern of mini-cubes is quite random, there is good temporal reuse of spline table data by each process. We have implemented a cache at the software layer - called Global Array Read Only Cache (GRC) - to exploit this interesting pattern. In this section, we explain the motivation for having a software cache by performing reuse distance analysis on simulation traces, followed by the design details and respective experimental results.

5.1.1 Reuse distance Analysis

Reuse distance of a memory access can be defined as the number of accesses to unique addresses made since the last reference to the requested data [10].

If the notion of time is not defined by the number of clock cycles but by the number of memory references, reuse distance is equivalent to the temporal reuse of data. Unlike the traditional notion of temporal reuse, reuse distance is precisely defined and machine independent.
Figure 5.1 shows an example sequence and the distance for each pair of data reuses. The distance between the reuses of $a$ is two because two distinct data elements, $b$ and $d$, are accessed in between. Reuse distance of $b$ is $\infty$ as $b$ is never accessed again. Similarly, $c$ has a Reuse distance of one and $d$ has a Reuse distance of two as there are two distinct elements accessed between its reuse.

![Trace: a b d a c c d](image)

Figure 5.1: Reuse Distance

Reuse Distance Analysis models the cache behavior by directly measuring the program locality. Data reuse is an inherent program property and does not depend on cache parameters. The figure 5.2 provides the algorithm for Reuse Distance Analysis, which computes the reuse distances for the entire trace and models accurately fully associative caches with LRU replacement policy[8]. In the algorithm, $N$ is the length of the memory trace, and $M$ is the number of distinct memory accesses. The trace is indexed by the letter $i$, where $1 \leq i \leq N$. Further by using the stack distances of the entire trace, the reuse histograms can be built, which can be used to understand optimal parameters to maximize the utilization of data locality.

As reuse distance analysis models a fully-associative LRU cache, if the cache has $n$ lines, for access $a$ if the reuse distance is $d$:

1. $a$ is a hit, if $d < n$.

2. $a$ is a miss, if $d \geq n$. 

19
Repeat the following steps for each memory reference \( x(i) \) \( 0 \leq i \leq N \):

1. Find the location in the Reuse of the most recent reference to the current location.

2. Compute \( \text{dist}(i) \), the Reuse distance for the current location, by finding the last reference to the current location and counting the number of elements on the Reuse above it. If the most recent reference is not found, \( \text{dist}(i) \) is defined as \( \infty \).

3. Bring the most recent reference to the top of the Reuse.

**Figure 5.2: Reuse Distance Algorithm**

3. a is a cold miss, if \( d = \infty \) (i.e. a has not been accessed before).

We have performed reuse distance analysis on Simulation traces for different inputs and plotted the reuse distance histograms, where stack distance in MB is plotted against cumulative percentage of references. The figure 5.3 show the histograms for the inputs SI 250. As you see in the figure, the reuse looks extremely good.

**Figure 5.3: Reuse Distance of SI 250 input**
5.1.2 Design

The GRC is a software cache, which can cache multi dimensional bricks of data.

As mentioned in the previous section, the main hope of using GRC is to be able to hold the data which will be re-used temporally, in order to reduce communication costs. In traditional hardware caches, whenever a memory location is accessed, a (aligned boundary) block containing the particular location is brought into the cache for three reasons: (i) this better utilizes the memory bandwidth, (ii) to exploit spatial locality, and (iii) to enable efficient cache lookup - if the boundaries of the retrieved block are arbitrary this creates additional inefficiencies in address translation. While point (ii), spatial locality, is not very high in applications for which GRC is intended, points (i) and (iii) nonetheless are analogously true. Hence, for GRC, we follow the strategy of dividing up the 3d space into aligned 3-dimensional bricks as shown in the figure 5.4. However, one point which is true

![Figure 5.4: Aligned Multi Dimensional Bricks](image_url)

for traditional hardware caches, but not for us, is that the smallest unit of memory access

21
in our case is a mini-cube, which need not be fully contained inside a single brick. In other words, in order to serve a request for a particular mini-cube, it may be necessary to bring in multiple bricks that together cover the requested mini-cube. One point that makes the design of GRC easier is that the data is read-only, allowing us to ignore issues of cache consistency etc. which come with distributed read-write caches.

GRC is implemented as a fully associative cache with LRU eviction policy. Cache is implemented using simple data structures like linked list and hash tables. We have two different implementations one using ghashtable, the other one using UThashtable. User can choose the specific implementation at compile time.

5.1.3 Selective caching

In QMC applications it is observed that few evaluation calls (interpolation) have good temporal locality and a few of them are very random in nature. For example, eval vg, vgh and vgl calls are very random in nature though these are very few in number compared to evaluation calls with good reuse.

Good reuse

1. void eval_multi UBspline_3d_z (multi UBspline_3d_z *spline,
   double x, double y, double z, complex double* restrict vals)

Very random in nature

1. void eval_multi UBspline_3d_z vg (multi UBspline_3d_z *spline,
   double x, double y, double z, complex double* restrict vals,
   complex double* restrict grads)
2. `void eval_multi_UBspline_3d_z_vgl (multi_UBspline_3d_z *spline, double x, double y, double z, complex_double* restrict vals, complex_double* restrict grads, complex_double* restrict lapl)`

3. `void eval_multi_UBspline_3d_z_vgh (multi_UBspline_3d_z *spline, double x, double y, double z, complex_double* restrict vals, complex_double* restrict grads, complex_double* restrict hess)`

This is especially true for QMCPack, where all the correlated calls are batched. This feature is expected to be incorporated in Qwalk as well in the future. Keeping this in mind, support to mark the evaluation calls is provided. User can explicitly mark which GA.gets need to be cached. So user will have the flexibility to choose the accesses with only good reuse to be cached. This feature can result in drastic improvement over our previous cache all scheme in QMCPack.

### 5.1.4 API

A very simple API is provided for the GRC, this GRC is customized for einspline library and can be integrated into future einspline releases. With more generalization, it can be used as a Global Arrays added functionality.

`cache_t` is a datatype for storing meta-data regarding cache and Global Arrays that store spline data.

1. `cache_t* GRC_create(coefs_t *coefs, int bricksize, int max_bricks);`

   This function creates a cache with the given brick size and cache size and returns a handle to the cache. This handle is used to attach the cache to the required global array.
2. `void GRC_get(cache_t *cache, int x, int y, int z, void *minicube);`

This function returns the specified minicube using the cache. The cache itself holds aligned boundary multi-dimensional bricks, hence a minicube is built from the corresponding bricks. This is depicted in the figure 5.5.

![Figure 5.5: Minicube from Bricks](image)

This function first checks if the cache has the required bricks to build the requested minicube, if not, gets the required bricks from Global Array and returns the built minicube.

3. `void GRC_destroy(cache);`

This function is used for cleaning up the GRC and detaching it to the Global Array.
5.1.5 Analysis for optimal cache utilization

To better capture the data reuse pattern, it is important to optimally utilize the cache, which in turn requires good characterization of the access pattern. In this section we discuss the design of GRC, its API and how to set the parameters of GRC in order to enable optimal cache utilization.

It is important to characterize the access pattern to optimize the cache utilization. We performed a variety of analytical tests such as reuse Distance Analysis, to understand the access pattern. Based on this analysis we try to set the cache parameters such as brick size and cache size, so that we optimize the cache utilization for a given application. In this section we discuss the analysis performed and the results of it.

5.1.5.1 Brick Size-Communication Overhead

Brick can be thought of as a multi dimensional cache block. Analogous to how cache block strongly affects the hardware cache performance, choice of brick size can tremendously affect the cache utilization.

Brick size will affect cache reuse and communication overhead. By bricked access of spline data, we would get some unwanted data for each access. This might be beneficial when there is good spatial reuse. But it also might fill up the cache with unwanted data. On one extreme if brick size is one, the cache would not have any unwanted data. But it would result in more access of Global Array for each evaluation call, which would result in higher communication overhead. On the other hand if the brick size is too big, cache might get polluted with unwanted data, but each evaluation call would need fewer accesses of Global Array which might reduce the communication overhead. As shown in the figure 5.6 for the specified minicube access, when brick size 4 is used 4 bricks in the 2 dimensional plane are
required. But when brick size 2 is used 9 bricks are required. Though the number of bricks required might increase with decrease in brick size, the total volume overhead might also be reduced.

Figure 5.6: Minicube straddling between bricks for brick sizes 2 and 4 (in 2D plane)

The advantages and disadvantages associated with choice in brick size can be listed as follows:

**Bigger brick sizes**

1. Good spatial locality can be captured by bigger block sizes, which essentially amortizes the start up latency (ts) across a larger message.

2. At the same time unwanted data might be read which results in more communication volume overhead.

**Smaller brick size**

1. Lesser unwanted data will be read, less communication volume overhead.

2. More accesses to Global Array to get smaller bricks required for each evaluation call. More start up latency.
5.1.6 Reuse Distance Analysis on Brick trace

Traditional reuse distance analysis is performed on point access memory traces. Unlike traditional programs our unit of access for QMC is a minicube. As we use bricks to utilize the bandwidth and amortize the start latency costs, we would want to model the spatial locality by using a specific brick size. For this reason, we performed reuse distance analysis on the trace of brick accesses rather than the conventional trace of unit memory accesses. This analysis also has the advantage that it accounts for spatial reuse.

In this section, we perform Reuse distance analysis on real traces of bricks. We have taken simulation traces(mini cube accesses) of different inputs, translated them into brick accesses of different sizes and performed Reuse distance analysis on these brick traces.

Figure 5.7: Reuse Distance Analysis for Brick=2 on Input SI.3x3x3
5.1.7 Best brick size for a given cache size

Using the Reuse distance histograms, we can accurately calculate cache misses for varying cache sizes. In our Reuse distance analysis on brick traces, cache misses correspond to communication costs.

In our previous Reuse distance histograms, the unit of Reuse distance differs for each brick size and also the total references. Hence we have revised our analysis by incorporating our analysis on volume overhead and latencies into Reuse distance analysis on brick traces. We use Reuse distance data and plot the communication costs for varying cache sizes for each brick size.

Total Communication cost for brick size \( x \) and cache size \( y \) = No. of Cache misses * cost per brick access = (No. of cache misses) * (totalrefs) * (cost per brick access for size \( x \)) where No. of cache misses = fraction of References with Reuse distance less than cache size \( y \).

Total Communication cost if GRC was not used = (No. of eval calls) * (cost per minicube access)

For each combination of cache and brick sizes, total Communication Overhead is the ratio of the total communication cost at that brick and cache size combination to the total communication cost if GRC was not used. This ratio can be simplified by noting that totalrefs = (no. of bricks per eval call) * (no. of eval calls). Hence, total communication overhead = (cache misses) * (no. of bricks per eval call) * (cost overhead per brick access)

Using the above expression, we can calculate the communication overheads for different combination of cache and brick sizes. Clearly, GRC results in speedups for only those settings of the cache and brick sizes where the above expression evaluates to less than 1. In general, we try to obtain the particular setting which minimizes this overhead.
Figure 5.8: Brick size Analysis for QWalk Input: si3x3x3

Figure 5.8 shows the plot for communication overhead for various brick sizes. As shown in the figure, brick size 2 outperforms any other brick size for this input. This is due to the fact that brick size two is enough to amortize the latency costs.

5.1.8 Experimental Results

**Input:** Silicon 3x3x3 is a system with dimensions of its spline table being $39 \times 39 \times 39 \times 54$. As the data is all complex double type, spline table takes up 48MB memory.

**System:** These experiments are run on ri cluster of CSE Department OSU. Each core in this cluster has 8 cores. The experiment is run using 60 nodes and 480 processors.

The table 5.1 shows the experimental results of using GRC when all the calls are cached. As shown in the table, as cache size increased the performance increases. All the results are for a brick size 2.

The table 5.2 shows the experimental results of using selective caching as compared to basic GRC.
### Table 5.1: QWALK:Si.3x3x3- GRC

<table>
<thead>
<tr>
<th>Cache size</th>
<th>GA (without GRC)</th>
<th>657 secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>20MB</td>
<td>681 secs</td>
<td></td>
</tr>
<tr>
<td>25MB</td>
<td>629 secs</td>
<td></td>
</tr>
<tr>
<td>30MB</td>
<td>583 secs</td>
<td></td>
</tr>
<tr>
<td>40MB</td>
<td>512 secs</td>
<td></td>
</tr>
<tr>
<td>45MB</td>
<td>480 secs</td>
<td></td>
</tr>
<tr>
<td>60MB</td>
<td>430 secs</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5.2: QWALK:Si 3x3x3- Selective Caching

<table>
<thead>
<tr>
<th>Cache size</th>
<th>All</th>
<th>Marked</th>
</tr>
</thead>
<tbody>
<tr>
<td>7MB</td>
<td>954 secs</td>
<td>567 secs</td>
</tr>
<tr>
<td>10MB</td>
<td>890 secs</td>
<td>522 secs</td>
</tr>
</tbody>
</table>

#### 5.2 Process groups

Although with the use of GA the upper limit on the problem sizes is aggregate memory of the nodes, often spline data is not as big as the aggregate memory of the nodes. Instead of just one single copy of spline data on all nodes, it is possible to have multiple copies of spline data - that is one copy for each process group.

#### 5.2.1 Approach

As all the processes on a node share the same physical memory, it is appropriate to have all processes on a node belong to the same process group. Process group size is minimized based on the Spline table size, so that local portion of the Global Array is maximized, thereby decreasing the communication costs.
### 5.2.2 API

A simple call is provided which can activate the process groups. No other changes in the einspline library are required. This function calculates the optimum group size based on processes per node and Spline data size, creates lists of process groups and creates process groups using process groups support of Global Array toolkit.

```c
void coefs_create_groups();
```

### 5.2.3 Experimental Results

**Input:** Silicon 250 is a system with dimensions of its spline table being $79 \times 79 \times 79 \times 250$. As the data is all complex double type, spline table takes up 1.6 GB memory.

**System:** These experiments are run on ri cluster of CSE Despartment OSU. Each core in this cluster has 8 cores.

The experiment is run using 10 nodes. The table 5.3 shows the experimental results of using Process groups as compared to using plain GA. As process group size is minimized results improved.

<table>
<thead>
<tr>
<th></th>
<th>Basic GA</th>
<th>GA +PGsize = 24</th>
<th>GA +PGsize = 16</th>
<th>GA +PGsize = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>66 secs</td>
<td>59 secs</td>
<td>57 secs</td>
<td>53 secs</td>
</tr>
</tbody>
</table>

Table 5.3: QWALK:Si 250- Process Groups
5.3 Coupling Process groups and GRC

If no caching was used, minimizing the process group size is beneficial. This is because, smaller the group, lesser the contention and also because local portion of GA would be more.

But when process groups and GRC are both utilized, as for both of these maximizing the memory usage on node is beneficial, there is no straight advantage of maximizing the memory for one over the other.

5.3.1 Smaller group / Smaller Cache

Smaller group size essentially means more node memory utilized for storing the GA which results in smaller cache sizes. This means:

1. Lesser contention as there are fewer processes accessing the GA.

2. Less communication overhead, local portion of GA is more.

3. Bad hits/misses ratio

Figure 5.9: Small group size / Small Cache size
5.3.2 Bigger group / Bigger Cache

Bigger group size means the GA copy is spread across more number of nodes giving more space to the software cache. This means:


2. More contention involved with every ga access (local and non local).

3. More communication overhead, as non local portion of GA is more.

![Figure 5.10: Big group size / Big Cache size](image)

5.3.3 Experimental Results

**Input:** Silicon 250 is a system with dimensions of its spline table being $79 \times 79 \times 79 \times 250$. As the data is all complex double type, spline table takes up 1.6 GB memory.
Table 5.4: QWALK:Si 250- Process Groups +GRC

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic GA</td>
<td>66 secs</td>
</tr>
<tr>
<td>GRC only</td>
<td>63 secs</td>
</tr>
<tr>
<td>GRC (0.43GB) + PG(24)</td>
<td>55 secs</td>
</tr>
<tr>
<td>GRC (0.4GB) + PG(16)</td>
<td>54 secs</td>
</tr>
<tr>
<td>GRC (0.3GB) + PG(8)</td>
<td>52 secs</td>
</tr>
</tbody>
</table>

**System:** These experiments are run on ri cluster of CSE Department OSU. Each core in this cluster has 8 cores.

The experiment is run using 10 nodes. The table 5.4 shows the experimental results of coupling Process groups along with GRC. Experimental results showed that minimizing the process group size is beneficial.
CHAPTER 6

CONCLUSIONS

We have designed an automatic data management solution utilizing aggregate memory of the cluster, for enabling bigger Quantum Monte Carlo Simulations. Our contributions can be summarized as:

1. Our system can run larger data sets which were earlier not possible. We ran si1010 (13GB) input using the enhanced einspline library.

2. Our optimizations performed well individually as well as together.

3. GRC can be especially useful for data sets whose 4th dimension is greater than around 500, for which even brick size one will amortize the startup costs.


[9] George Almasi, Calin Cascaval and David A. Padua, Calculating Stack Distances Efficiently


APPENDIX A

USING ENHANCED EINSPLINE

A.1 API call to initiate Enhanced Einspline

To use the enhanced einspline library, `coefs_ga_init()` should be called from QMC application before the setup phase where Spline Data is created. Once this function is called, enhancements are enabled and no other changes to the QMC application are needed. In Qwalk, this function is called from `read function` of `MultiEinsplineOrbComplex` class before reading the orbital files.

A.2 Compiling

All the enhancements to the einspline library are implemented such that they can be just plugged in by providing the appropriate flags while compiling einspline.

FLAGS:

1. USE_GA

To enable Spline data to be distributed using Global Arrays.

2. USE_CACHE

To enable GRC, software caching optimization.
3. USE_GROUPS

To enable Process Groups, where each process group can share a copy of the Spline data.

A.3 Runtime Parameters

User also is provided the flexibility to tune the parameters like cache parameters and the process group size. Modified Einspline Library reads in the required parameters from a file "GA_info"

These parameters can be specified, if no values are specified the default values are used.

Example:

Brick_size=4
Cache_size= 1000 KB
PG_size = 16

Where brick size 4 means, brick has dimensions 4x4x4xnumsplines and process group size 16 means, each set of 16 processes form a group, which has its own copy of spline data.

A.4 Parellel creation of Spline Data in Qwalk

In the traditional QWalk, each process needs a copy of Spline Data and hence each process reads in the orbital files and creates its own copy of Spline data. This happens in two steps: memory required to hold spline data is created by each process using create function of einspline and then four-dimensional spline data is computed as slices across fourth dimension one spline at a time using set function of einspline, where there is no dependency between slices.
With the enhanced einspline, as not every process needs its own copy of the spline data, this process of building spline data is parallelized. All the processes call the create function which internally creates the Global Array, and then each slice of the spline Data is computed by one process parallelly. All the changes are made inside the function `read` of `MultiEinsplineOrbComplex` class of Qwalk. This feature can be enabled by using a flag:

`USE_PARALLEL_SET`

Note: In Qwalk, this feature must be enabled as changes have been made in Qwalk which restricts all the processes from reading the orbital files and hence a serial setting of Spline Data is not possible.