COMPARING THE PERFORMANCE OF DISTRIBUTED SHARED MEMORY AND MESSAGE PASSING PROGRAMS USING THE HYPERION JAVA VIRTUAL MACHINE ON CLUSTERS

BY

Mathew S. Reno

B.S., University Of New Hampshire, 1994

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science
in
Computer Science

May, 2003
This thesis has been examined and approved.

______________________________
Thesis Director, Philip J. Hatcher, Professor of Computer Science

______________________________
R. Daniel Bergeron, Professor of Computer Science

______________________________
Elizabeth Varki, Assistant Professor of Computer Science

______________________________
Date
TABLE OF CONTENTS

LIST OF TABLES............................................................................................v
LIST OF FIGURES..........................................................................................vi
ABSTRACT.......................................................................................................viii

CHAPTER........................................................................................................PAGE
I. INTRODUCTION.............................................................................................1
II. BACKGROUND..............................................................................................4

Hyperion........................................................................................................4
mpiJava..........................................................................................................6

Communication Protocols and Cluster Performance.......................................7
Java Grande Forum Benchmarks.....................................................................9
Related Work...............................................................................................10

III. IMPLEMENTATION....................................................................................12

MPI on Hyperion..........................................................................................12
Performance Tuning the Multithreaded Benchmarks and DSM Hyperion.........13

IV. BENCHMARKS............................................................................................16

Statistical Analysis Formulas........................................................................16
Benchmark Code Example...........................................................................17
Fourier Coefficient Analysis.........................................................................20
Lower Upper Factorization.........................................................................21
Successive Over-Relaxation.........................................................................24
IDEA Crypt....................................................................................................27
Sparse Matrix Multiplication.......................................................................30
Molecular Dynamics

Ray Tracer

Monte Carlo

V. CONCLUSIONS

APPENDIX

LIST OF REFERENCES
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: Fourier Coefficient Analysis Confidence Interval on the Star Cluster</td>
<td>47</td>
</tr>
<tr>
<td>Table 2: Fourier Coefficient Analysis Confidence Interval on the Paraski Cluster</td>
<td>47</td>
</tr>
<tr>
<td>Table 3: LU Factorization Confidence Interval on the Star Cluster</td>
<td>47</td>
</tr>
<tr>
<td>Table 4: LU Factorization Confidence Interval on the Paraski Cluster</td>
<td>48</td>
</tr>
<tr>
<td>Table 5: SOR Confidence Interval on the Star Cluster</td>
<td>48</td>
</tr>
<tr>
<td>Table 6: SOR Confidence Interval on the Paraski Cluster</td>
<td>48</td>
</tr>
<tr>
<td>Table 7: Crypt Confidence Interval on the Star Cluster</td>
<td>48</td>
</tr>
<tr>
<td>Table 8: Crypt Confidence Interval on the Paraski Cluster</td>
<td>49</td>
</tr>
<tr>
<td>Table 9: Sparse Matrix Multiplication Confidence Interval on the Star Cluster</td>
<td>49</td>
</tr>
<tr>
<td>Table 10: Sparse Matrix Multiplication Confidence Interval on the Paraski Cluster</td>
<td>49</td>
</tr>
<tr>
<td>Table 11: Molecular Dynamics Confidence Interval on the Star Cluster</td>
<td>49</td>
</tr>
<tr>
<td>Table 12: Molecular Dynamics Confidence Interval on the Paraski Cluster</td>
<td>50</td>
</tr>
<tr>
<td>Table 13: Ray Tracer Confidence Interval on the Star Cluster</td>
<td>50</td>
</tr>
<tr>
<td>Table 14: Ray Tracer Confidence Interval on the Paraski Cluster</td>
<td>50</td>
</tr>
<tr>
<td>Table 15: Monte Carlo Confidence Interval on the Star Cluster</td>
<td>50</td>
</tr>
<tr>
<td>Table 16: Monte Carlo Confidence Interval on the Paraski Cluster</td>
<td>50</td>
</tr>
</tbody>
</table>
Figure 23: Performance Ratio of DSM to MPI on the Paraski Cluster

Figure 24: Full Exchange With Four Nodes

Figure 25: Efficient Broadcast With Four Nodes
ABSTRACT

COMPARING THE PERFORMANCE OF DISTRIBUTED SHARED MEMORY AND MESSAGE PASSING PROGRAMS USING THE HYPERION JAVA VIRTUAL MACHINE ON CLUSTERS

By

Mathew S. Reno

University of New Hampshire, May, 2003

The Hyperion system, developed at the University of New Hampshire, is a high-performance implementation of a distributed Java virtual machine. Programs designed for Hyperion are run on a cluster, a pool of computers connected together with a network. Hyperion distributes a program’s workload amongst the computers in the cluster, which allows the user to view the cluster as if it were a single computer. This thesis compared two competing models for cluster computing: Hyperion’s distributed shared memory (DSM) and message passing (MP). The comparison was performed on two high-performance clusters using Java benchmarks developed for both shared memory and message passing. The analysis suggested that, in most cases, DSM could compete with MP, but only after significant modifications to the benchmarks.
As achievements in low-cost processor performance increased in the past two decades, the *cluster* arose as an environment for high-performance computing. A cluster is a collection of computers on a network that can function as a single resource. Network hardware has also benefited from recent advances. Low-cost high-performance networks can be created to increase bandwidth and throughput, as network performance is just as important as processor performance. With a high-performance cluster, users develop applications that can spread their workload between the individual computers of the cluster, thereby reducing computation time; this is cluster computing.

To harness the power of a cluster, an environment must be developed for distributing data to the nodes. Many environments are available to cluster users. Some place the burden of data distribution on the user, requiring the user to understand how the cluster is designed, while others try to hide the communication details. Each method has its advantages and disadvantages. One such environment is Hyperion.

Hyperion is an environment for cluster computing developed at the University of New Hampshire\(^1\). The system is an implementation of the Java 1.1 virtual machine (VM) specification. A compelling reason to use Java is that it supports a relaxed memory model. This type of model allows modifications to objects to be done with cached values updating main memory in bursts, possibly reordering for efficiency, instead of one at a time. When Java threads are employed, each thread has its own local memory cache and
synchronization is used to update main memory. Hyperion makes use of the cluster by distributing Java threads among the nodes of the cluster. Hyperion does this by employing a distributed shared memory (DSM) model implemented in the PM2[2] subsystem. The DSM model is an extension of the shared memory model, which allows all threads to access the same memory space. The DSM model extends this idea across the cluster, allowing a thread executing on one node to access memory on another node. PM2 is a highly portable parallel computing environment that creates and distributes lightweight threads throughout a cluster.

By providing parallelization through Java threads, the user does not have to learn a new extension to Java. Simply creating threads provides the mechanism for distributing the workload. However, this approach shifts a large burden to the environment. Hyperion must keep track of where Java objects are located and coordinate the remote threads for object sharing.

In this thesis we evaluate the performance of Hyperion by comparing it to another cluster environment, the Message Passing Interface (MPI). MPI is currently the most widely used environment as it is easy to learn and highly portable. MPI utilizes explicit messages to pass data between nodes in a cluster. An existing implementation of MPI, javaMPI[3], is used to do the comparison to Hyperion. The javaMPI implementation, however, is not a pure Java implementation of MPI as it relies on a native C language implementation of MPI, accessed by javaMPI through the Java Native Interface (JNI). To provide as close a comparison as possible, we ported the javaMPI implementation into the Hyperion environment. This allowed us to compile any program written for javaMPI with the Hyperion compiler and execute it in the Hyperion environment, but use MPI for data
distribution.

A set of benchmarks developed by the Java Grande Forum (JGF) was chosen to test Hyperion. Two versions of these benchmarks are available from the JGF, an MPI (using javaMPI) version and a multithreaded version. These benchmarks tested both the computation performance of the processors in the cluster and the network performance. Our experiment executes both sets of DSM and MPI benchmarks on two different clusters: the Star cluster, 16 Pentium III 667MHz computers networked with 100Mb/s Ethernet using TCP located at the University of New Hampshire and the Paraski cluster, 16 Pentium III 1GHz computers networked with 2Gb/s Myrinet using BIP or GM located in Rennes, France. BIP (Basic Interface for Parallelism) is a high-performance message-passing system implemented on top of Myrinet. BIP can sustain one Gb/s bandwidth and can provide less than 5 µs latency for small messages. GM (Glenn’s Messages) is also a low-level message-passing communication layer for Myrinet. Myrinet provides a low-latency, high-performance environment while Ethernet provides a low-cost, real-world environment.

In the next chapter we discuss the details of Hyperion, mpiJava and the JGF Benchmarks as well as review past experiments and results. In Chapter 3 we explain what had to be implemented to perform this experiment. Chapter 4 details the execution of the benchmarks and their results. Finally, in Chapter 5 we draw our conclusions.
Hyperion

The goal of the design of Hyperion is to allow a developer to create standard multithreaded Java programs that, when executed on a cluster, will parallelize. To provide this support, the Hyperion environment conceals all cluster communication from the developer. The cluster is, in effect, viewed by the user as one virtual machine. Java threads can share objects and it is up to the developer to control access through standard Java language thread management. This mechanism maps well in the distributed shared memory (DSM) model as the same management techniques can be used to control access across nodes.

The Java memory model specifies that threads can cache local copies of the objects they are using. This cache is flushed when a critical section is entered. Critical sections are signified by the Java keyword “synchronized”. When the critical section is exited, local modifications to objects are propagated back to the main memory.

In Hyperion, this shared-memory model carries over to the cluster in that each node has objects that it creates and, when needed, local cached copies of remote objects. When a remote object is accessed, the object’s home node is sent a request for the object. The home node replies by sending the object to the requesting node. The node can now access the object as if it originally created it. When the object exits the critical section only the modifications are sent back to the original home node.
To provide an efficient execution, Hyperion translates Java bytecode into C code, and then a C compiler is used to generate a natively optimized executable. This process is hidden from the user by the Hyperion compiler. The C compiler is a natural choice for native compilation given that a C compiler is available for virtually every platform available. This allows Hyperion to remain portable, while producing native executables.

In order for Hyperion to communicate on a variety of high-performance networks and still maintain its portability, Hyperion was built on top of the DSM-PM2 (Distributed Shared Memory on a Parallel Multithreaded Machine) library. DSM-PM2 provides Hyperion with lightweight thread management, page-based DSM, and high-performance communication. When Hyperion attempts to create a Java thread, the PM2 thread creation routines are called. These routines create and distribute the threads to the appropriate nodes. When a thread accesses a Java object that is not in its memory, a page-fault occurs. Hyperion intercepts the signal and, using PM2, retrieves the missing page from the node executing the thread that created the object. The advantage to page-based DSM is that the user does not need to be aware of the cluster or the underlying communication protocols. It also eliminates the need for the user to coordinate object sharing, except for the cases where regular Java based object synchronization is necessary.

This simplicity comes at a potential disadvantage. While accessing local or cached objects is fast, accessing a remote object is not. When reading a non-cached remote object, Hyperion must first request the object from the remote thread where the object exists and the remote thread has to send the page containing the object back. This request-response style of page migration causes the requestor to block while waiting for the remote thread to respond. If many pages need to be migrated, through many remote
object accesses, then much communication is generated. The savvy developer will limit 
remote object accesses when developing DSM applications.

**mpiJava**

The message-passing interface (MPI) model is commonly used for cluster computing. This well-defined model has been implemented on many platforms and in 
various languages. For Java, mpiJava was developed. While not a true native Java MPI implementation, mpiJava allows developers to use MPI style objects and methods as an 
extension to Java. Internally, mpiJava translates the Java objects from the VM memory 
space into native memory space through the Java Native Interface (JNI). Once in the 
native memory space, mpiJava employs a native implementation of MPI. MPICH$^{[4][5]}$ is a 
popular implementation of MPI and the one we use in this thesis.

The most compelling reason MPI has become popular is its performance. The MPI 
model relies on the application developer to explicitly manage all data communication. 
This requirement forces the developer who wishes to produce optimized high-performance 
parallelized applications to understand the strengths and weaknesses of the nodes, 
network, language, and algorithms used. This requirement does add complexity to the 
overall design and implementation of a parallel application, but this has not proved to be a 
major deterrent when performance is important.

MPI implementations use explicit commands for all data communication. 
Commands, such as MPI_Send and MPI_Recv, coordinate data exchanges by 
programmatically indicating which nodes are to send or receive the data. By programming 
at this level of detail, developers can avoid unnecessary communication if they are careful.
Unfortunately, JNI performs poorly, as most of the data that is transferred between the VM and the native space has to be copied. There is a movement to produce a “100% pure” Java implementation of MPI, but none are suitable for use. Essentially, the mpiJava implementation is a Java wrapper for a native implementation of MPI. However, as we will discuss in Chapter 3, we avoid the JNI overhead in our javaMPI implementation by replacing JNI access with the user level Hyperion API.

![Figure 1: TCP Latency on the Star Cluster](image)

**Communication Protocols and Cluster Performance**

When examining the results of the experiments of this thesis, it is helpful to understand the differences between the two clusters used, both in hardware and in software. On the Star cluster, both the PM2 and the Hyperion ported javaMPI versions of the benchmarks communicate via TCP. TCP has a high overhead when transmitting data
across a network and this overhead impacts the performance of the benchmarks. Figure 1 shows the latency of TCP for various message sizes on the Star cluster. We see that the Hyperion port of javaMPI performs better as the message size increases due to PM2 having more overhead than javaMPI, especially with larger messages. However, most of the JGF benchmarks use small message sizes.

![Graph showing latency of TCP for various message sizes on the Star cluster.](image)

*Figure 2: GM and BIP Latency on the Paraski Cluster*

On the Paraski cluster we had to use two different communication protocols, BIP and GM. BIP is used by PM2 and GM is used by javaMPI. The same protocols could not be used for both, as MPICH implemented on BIP is not stable and PM2 has not been ported to GM. This poses a problem, as the performance of BIP and GM are not equivalent. Figure 2 displays the latency on the Paraski cluster, with PM2 now performing
better than MPI for larger messages. The additional overhead of PM2 is negligible when compared to the overall performance of GM. We can see that GM is competing with BIP for smaller message sizes, but performance seriously degrades from a 4KB message size and up.

**Java Grande Forum Benchmarks**

To test the MPI and DSM environments we chose to use Java benchmarks developed by the Java Grande Forum (JGF). The JGF provided both multithreaded and MPI versions of the benchmarks, which were then ported to Hyperion. These benchmarks execute code that would be most commonly used in large-scale high-performance applications, such as scientific applications. We chose a subset of the benchmarks that provided a wide range of performance testing. The benchmarks chosen were:

- Fourier coefficient analysis
- Lower/upper matrix factorization
- Successive over-relaxation
- IDEA encryption
- Sparse matrix multiplication
- Molecular dynamics simulation
- Monte Carlo simulation
- 3D Ray Tracer

The first five benchmarks listed above are considered kernel benchmarks, which represented testing of specific operations that were frequently used in large-scale applications. The last three benchmarks represent large-scale applications. All the benchmarks were developed in pure Java. The MPI benchmarks were developed using the
Related Work

Researchers at Rice University performed a similar experiment\textsuperscript{[6]}. For their experiment they compared message passing as implemented in the Parallel Virtual Machine (PVM) to DSM as implemented in TreadMarks. TreadMarks, built at Rice University, like Hyperion uses page-fault based DSM, but since they use C as the programming language, additional routines have to be added to a program developed for TreadMarks. These routines provide synchronization and memory allocation. PVM implements message passing similar to MPI, where the developer must explicitly coordinate all data distribution when considering program parallelization.

During the development of the nine benchmarks that were used for the comparison, they found that developing for TreadMarks was far more natural than PVM. In the case of two benchmarks, PVM versions were significantly harder to develop. The benchmarks represented a wide range of parallel tasks.

The tests were performed on a 8-node cluster of HP9000-735/125 workstations each with a single 125Mhz PA-RISC7100 processor and 96M of memory connected by a 100Mbps FDDI ring. The results ranged from PVM being 10-30\% better to PVM being two times faster. They concluded that TreadMarks required more messages and more data to be sent which impacts performance. Four main reasons why TreadMarks performed worse than PVM were given: the separation of synchronization and data transfer, extra messages to request updates for data, false sharing, and diff accumulation, a result of several processors modifying the same data, for migratory data. Ultimately, they felt that a
well-optimized DSM system is comparable to a message passing system, especially with problems of practical size. They also felt that the development time saved in programs with complicated communication patterns, by not having to determine what data to send and whom to send it to, was an important benefit for developing with DSM.
MPI on Hyperion

To eliminate as many differences in the comparison as possible, we decided that we would move mpiJava into Hyperion. Hyperion can be configured to operate in a single-node environment; threads created in this configuration are not distributed and all memory is local to the node. Hyperion in this configuration allows mpiJava to handle all communication while using the Hyperion compiler and runtime to build and execute the MPI JGF benchmarks. However, since Hyperion does not implement the Java Native Interface (JNI), moving mpiJava into Hyperion was not trivial.

The Java part of mpiJava, fortunately, could almost entirely be moved into Hyperion. A few classes were not implemented by Hyperion, namely the I/O based classes, and had to be replaced in the MPI benchmarks. Most classes simply compiled without modification. The ones that did not compile were mpiJava routines that we were not interested in since they were not used by the JGF benchmarks. These superfluous classes were removed.

The JNI portion of mpiJava was implemented in C, but since it used JNI constructs, the code could not simply be linked into Hyperion. Translating JNI code into the Hyperion runtime API proved difficult. Hyperion and JNI access native memory differently and we wanted to avoid overhead introduced by JNI when the code was ported into the Hyperion runtime. The Hyperion runtime API allows developers to write native
code that can access Java level objects. This C based API replaced the JNI API. JNI provides a high level of data security, as a result JNI has high overhead when translating Java memory to native or native memory to Java. Often, data is copied to safeguard against corruption; this can be avoided in most cases in Hyperion. By providing direct access to the Java object’s memory, Hyperion allows us to read and modify the object directly, without copying.

Hyperion’s runtime API does not provide methods for object serialization. This is something Java provides and that the JGF benchmarks require. The mpiJava implementation varies from the MPI specification by providing an object data-type that can be used when transmitting a Java object with MPI. This is a useful extension to the regular data-types (char, double, int, etc) and is used in a number of the JGF benchmarks. Object serialization is performed in mpiJava with the Java standard classes java.io.ObjectOutputStream and java.io.ObjectInputStream. These classes serialize a Java object and any object contained within it and places them in the stream, ultimately as a byte array. This stream can be sent via native MPI to another node where it can be deserialized back into the object. Hyperion does not implement the java.io package, however it provides a native way to determine all references within an object. This led to the development of a recursive routine to perform object serialization. With this routine complete, all mpiJava classes and methods used in the JGF benchmarks were now supported in Hyperion.

Performance Tuning the Multithreaded Benchmarks and DSM Hyperion

The multithreaded JGF benchmarks were designed to execute on one computer
with shared memory multiprocessors. While the multithreaded benchmarks parallelize well in this environment, their design is not optimum for a distributed shared memory environment. Synchronization among threads on a single processor or SMP machine is relatively fast when compared to synchronization amongst remote threads, so developers of multithreaded applications tend not to worry about object access between threads. However, in the Hyperion environment we need to limit object access as much as possible to reduce network traffic and blocked processors waiting for data. The multithreaded JGF benchmarks do little to minimize object access among threads; in fact, most of the benchmarks rely on this. After further examination of the JGF benchmarks, we determined that they could not be run “out-of-the-box” in Hyperion and be expected to perform competitively with MPI. To achieve speeds comparable to MPI, these benchmarks were modified.

Many of the performance problems encountered in the JGF benchmarks when run “out-of-the-box” stemmed from the fact that they are designed to create a large number of objects in one thread and require all other threads use synchronization to access these objects. In Hyperion, this causes one node in the cluster to become a bottleneck, as all other nodes need to communicate with it to retrieve the data they are working with. If more nodes are added, more traffic is generated and performance decreases.

The Java programming language does not define a high level way to coordinate groups of threads. Threads must simply share data by regular Java mechanisms. This approach does not scale well in the Hyperion environment: as more object accesses are created, which would be the case when using regular Java object sharing, more network traffic is produced. To coordinate data access efficiently, Hyperion was modified to
include new thread management classes. The Reduction class was created to handle data reduction. Many of the MPI JGF benchmarks perform an MPI_Reduce, which efficiently reduces values on all processes to a single value given a specific operation, such as addition. The Hyperion reduce is like the MPI_Reduce, however it operates on threads, not processes, enrolled with the Reduction class. The MPI JGF benchmarks also benefit from MPI_Bcast, which broadcasts a message from the root process to all other processes executing. A broadcasting facility was added to Hyperion to help efficiently distribute data among participating threads.

The final modification to Java was to create a distributed mechanism for accessing arrays. Remote array access is problematic in Hyperion and all of the JGF benchmarks operate on arrays. Writing to a remote array causes voluminous network traffic. To minimize remote array access, the developer must first perform a System.arraycopy, a preexisting Java operation, to copy the remote array into a locally allocated array. The arraycopy method was modified in Hyperion to send the entire block of data to the remote thread, instead of relying on page faults, thereby reducing the amount of messages required to retrieve the array. In the next chapter we show how these techniques are applied to these benchmarks to enhance performance.
CHAPTER 4
IV. BENCHMARKS
BENCHMARKS

Statistical Analysis Formulas

To understand the results of the benchmarks, we used techniques for determining population characteristics described by Jain\textsuperscript{[7]}. We concluded that we needed to run each benchmark 50 times to obtain a reasonable sample mean \( \bar{x} \). The sample size of 50 also satisfies the sample size requirement for the formulas listed below, as they only apply for samples of size greater than 30. Since it is not possible to get a perfect estimate for the population mean \( \mu \), we can determine the probabilistic bounds in which we are confident that the population mean exists. The formula to describe this is:

\[
\text{Probability} \left\{ c_1 \leq \mu \leq c_2 \right\} = 1 - \alpha
\]

The interval \((c_1, c_2)\) is the confidence interval for the population mean, \( \alpha \) is the significance level, \( 100(1 - \alpha) \) is the confidence level and \( 1 - \alpha \) is the confidence coefficient. We chose a confidence level of 95%, which defines \( \alpha \), the significance level, to be 0.05. The formula for determining the confidence interval from one sample is:

\[
(\bar{x} - z_{1-\alpha/2}/\sqrt{n}, \bar{x} + z_{1-\alpha/2}/\sqrt{n})
\]

Here, \( s \) is the sample standard deviation, \( n \) is the sample size and \( z_{1-\alpha/2} \) is the \((1 - \alpha/2)\) quantile of a unit normal variate. These quantiles are frequently used and have predefined values based on the significance level. With the \( \alpha \) we chose of 0.05, the \( z_{1-\alpha/2} \) value is 1.960. Given that \( z_{1-\alpha/2} \) and \( \sqrt{n} \) are constants, all we need to determine
is the sample mean $\bar{x}$ and sample standard deviation $s$ of each benchmark to determine
the lower and upper bounds for the confidence interval based on the confidence interval
formula.

By using the confidence interval we can state with, in our case, 95% confidence
that the DSM or MPI versions perform equal or better. One performs better than the other
if the confidence intervals for the sample do not overlap. This means that one population
mean will always be better. The population mean is always bounded by the confidence
interval and can exist anywhere within it. If the confidence intervals do overlap, then we
cannot say one is better than the other since we cannot determine which population mean
would be better. We list all the confidence intervals for each benchmark in the Appendix
and plot the sample mean (always the median of the confidence interval) in the figures
used throughout this chapter.

**Benchmark Code Example**

The modifications made to all the JGF benchmarks were similar in nature. We
illustrate the nature of these changes in this section by describing one example. Figure 3
shows a code listing from the original multithreaded sparse matrix multiplication
benchmark. In this version, as the SparseRunner objects are constructed they are passed
references to arrays created by the root node (lines 11 and 12). The SparseRunner object
is distributed to the remote nodes (line 13). This causes a bottleneck when accessing these
arrays from the remote threads (such as line 48). However, the greater problem is that
each thread updates an array, “yt”, that is a public member of the SparseMatmult class
every iteration (line 48). Clearly, this will produce excessive network traffic to the root
node as each thread vies for access to this array.

```
1 // Excerpt from SparseMatmult.java
2 // Start of the Sparse Matrix Multiplication timer
3 SparseRunner thobjects[] =
4   new SparseRunner[JGFSparseMatmultBench.nthreads];
5 Thread th[] = new Thread[JGFSparseMatmultBench.nthreads];
6
7 JGFInstrumentor.startTimer("Section2:SparseMatmult:Kernel");
8
9 for(int i = 1;i < JGFSparseMatmultBench.nthreads; i++) {
10   thobjects[i] =
11       new SparseRunner(i, val, row, col, x, NUM_ITERATIONS, nz,
12            lowsum, highsum);
13   th[i] = new Thread(thobjects[i]);
14   th[i].start();
15 }
16
17 thobjects[0] =
18   new SparseRunner(0, val, row, col, x, NUM_ITERATIONS, nz,
19    lowsum, highsum);
20 thobjects[0].run();
21
22 for(int i = 1;i < JGFSparseMatmultBench.nthreads; i++) {
23   try {
24      th[i].join();
25   } catch (InterruptedException e) {}
26 }
27
28 JGFInstrumentor.stopTimer("Section2:SparseMatmult:Kernel");
29
30 for (int i = 0; i < nz; i++) {
31    ytotal += yt[row[i]];
32 } // End of the Sparse Matrix Multiplication Timer
33
34 // SparseRunner class
35 class SparseRunner implements Runnable {
36 // Member variables removed
37     public SparseRunner(int id, double val [], int row [],
38                  int col [], double x [], int NUM_ITERATIONS,
39                  int nz, int lowsum [], int highsum []) {
40        // Assignment to member variables removed
41     }
42
43     public void run() {
44        for (int reps = 0; reps < NUM_ITERATIONS; reps++) {
45           for (int i = lowsum[id]; i < highsum[id]; i++) {
46             SparseMatmult.yt[row[i]] += x[col[i]] * val[i];
47           }
48        }
49     }
50 }
51 //
52 }
```

Figure 3: Original Threaded Version of the Sparse Matrix Multiplication Benchmark
Figure 4 and Figure 5 show code listings of the differences between the DSM and MPI, respectively. The figures are a listing for the main loop of the sparse matrix multiplication benchmark. Figure 4 is the DSM version with optimizations discussed in chapter 3. Figure 5 is the javaMPI version. The modified DSM version in Figure 4 only

```java
Reduction.sum(0);
if (id == 0)
    JGFIstrumentor.startTimer("Section2: SparseMatmult: Kernel");
for (int reps = 0; reps < numIter; reps++) {
    for (int i = 0; i < myLen; i++) {
        locY[locRow[i] - ilow] += locX[locCol[i]] * locVal[i];
    }
}
System.arraycopy(locY, 0, globY, ilow, ilen);
Reduction.sum(0);
if (id == 0)
    JGFIstrumentor.stopTimer("Section2: SparseMatmult: Kernel");

Figure 4: DSM Version of the Sparse Matrix Multiplication Benchmark

```
accesses thread local arrays during the loop and when complete uses the arraycopy (line 12) command to return the results to the root node by storing the remote thread’s data into the “globy” array. The javaMPI version uses MPI_Allreduce (line12) to return the results to the root process.

**Fourier Coefficient Analysis**

The Fourier coefficient analysis benchmark computes the first \( N \) pairs of Fourier coefficients of the function:

\[
f(x) = (x + 1)^x\text{ on the interval } 0, 2 \text{ with } N = 10,000
\]

The benchmark was designed to heavily exercise transcendental and trigonometric functions. Most of the run time is spent in the loop over the Fourier coefficients. Each iteration of the loop is independent of every other, thus work is simply distributed between the nodes with each node being responsible for updating the elements of its own block.

The multithreaded version of this benchmark was modified so that each thread could update local arrays that contain its portion of the Fourier coefficients. Once the algorithm completed its portion of the calculation, arraycopy was used to copy the pair of local arrays into their final destinations. After a barrier, the timer is stopped. The multithreaded version simply wrote directly to the final arrays, which in Hyperion would cause a communication bottleneck on the root node as more threads are added.

The MPI version is similar to the DSM version. After each process computes its portion, it sends the resulting arrays back to the root node. The MPI version uses the methods MPI_Ssend (synchronous send) on the remote nodes and MPI_Recv on the root node to perform this. Once data is collected and a barrier reached, the timer is stopped.
However, one extra barrier that the MPI version requires for synchronization, located prior to the data communication.

Figure 6 shows the sample mean comparison of the DSM and the MPI versions on the Star cluster. The results show that both versions perform roughly the same. Figure 7 indicates similar results on the Paraski cluster, except for the 16 node case where DSM outperforms MPI. This could be attributed to better performance in BIP than in GM.

**Lower Upper Factorization**

The LU Factorization benchmark solves an $N \times N$ (with $N = 500$) linear system using LU factorization followed by a triangular solve. The benchmark is memory and floating point intensive. The factorization is the only part of the computation that is
parallelized while the triangular solve is computed serially.

In the DSM version, after the matrix is distributed to the participating threads, the timer is started, the factorization begins and the Hyperion broadcast facility is used to broadcast the pivot column and column number to all threads on each iteration for Gaussian elimination. After each thread loops over its portion of the matrix, `arraycopy` is used to send the matrix to the root node. Again, this benchmark was modified because the root thread created the matrix and all threads accessed the matrix object through the root thread. Finally, a barrier is used to coordinate the stopping of the timer.

The MPI version is similar in that it uses the MPI_Bcast command to broadcast the pivot column and column number to the other processes. To acquire the final matrix on the root process, all remote processes call MPI_Send while the root process calls

![Figure 7: Fourier Coefficient Analysis Sample Mean on the Paraski Cluster](image-url)
MPI_Recv. A barrier is then reached and the timer is stopped.

Figure 8 and Figure 9 show the results on the Star and Paraski clusters, respectively. On the Star cluster, the DSM version does not parallelize well after two nodes while the MPI version continues to perform through 16 nodes. Hyperion’s broadcast has more overhead than the MPI version's. This is due to the fact that the broadcast facility was built on top of the reduction class. This class requires a full exchange when distributing data. The full exchange mechanism causes extra empty broadcast messages on nodes that do not currently contain the relevant data. This overhead affects the Star cluster, where message cost is an important factor, while on the Paraski cluster the overhead does not impact parallelization through 16 nodes, since message cost is low.
**Successive Over-Relaxation**

The successive over-relaxation (SOR) benchmark performs 100 iterations of successive over-relaxation on an $N \times N$ grid, where $N$ is 1000. The algorithm uses a “red-black” ordering mechanism that allows array rows to be distributed among nodes in a block manner. Once data is distributed, the SOR algorithm only requires neighboring rows to be communicated. Initial data distribution time is not reflected in the timing results.

![Figure 9: LU Factorization Sample Mean on the Paraski Cluster](image)

Once the data is distributed and the over-relaxation begins, the DSM version uses `arraycopy` to update each neighbor’s boundary rows. This is done after a barrier is reached during the main loop over the grid. After the neighbor rows are updated, another barrier is reached and the over-relaxation can continue. Once the over-relaxation is complete,
arraycopy is used to copy the local grid portion to the root thread to assemble the final grid. Once all threads complete the copy, another barrier is reached and the timer is stopped. Once again, the multithreaded version of this benchmark was modified because of excessive remote object access that would occur running the benchmark unmodified.

The MPI version is similar to the DSM version up to the point where the neighbor rows are exchanged. The MPI version uses the MPI_Sendrecv command to exchange neighbor rows. This command sends data and immediately waits to receive. Once the grid is traversed, a barrier is reached and then the local portion of the grid is copied back to the root process via MPI_Ssend and MPI_Recv commands. After another barrier, the timer is stopped.
Figure 10 and Figure 11 list the results on the Star and Paraski clusters, respectively. On the Star cluster, the DSM version suffers from an extra barrier used for synchronization after the row neighbors are swapped. This barrier blocks every thread from continuing with the algorithm. This is due to the “network reactivity” problem. In order for the root thread to efficiently handle all the requests for objects from the remote threads, it cannot be processing its part of the algorithm. If the barrier is removed, the root thread would continue with the algorithm and begin working on its portion of the data while, at the same time, the remote threads would be trying to obtain their data. A lightweight PM2 level thread is required to be scheduled to process requests from other threads, however, the root Java thread is busy calculating and scheduling is delayed. By using the barrier, the root thread is blocked waiting for the other threads to reach the barrier allowing PM2 thread scheduling to happen immediately. This strategy greatly
improves performance as the number of nodes increases.

The MPI version uses the MPI_Sendrecv method, which only blocks the processes involved in the neighbor row swap. The extra barrier communication coupled with all threads being blocked at each iteration of the SOR algorithm causes serious performance degradation by 16 nodes on the star cluster. The performance of the DSM version on the Paraski cluster is not as significantly affected by the barrier and the benchmark continues to parallelize well through 16 nodes. The larger capacity, speed and efficiency of the Myrinet network allows the benchmark to respond well, even with the extra barrier.

**IDEA Crypt**

The crypt benchmark performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes, where N is 3,000,000. To parallelize the algorithm, the array is divided among nodes in a block fashion. The IDEA algorithm then processes the blocks. Once the process has encrypted and decrypted its block, the process returns the decrypted block back to the root process, which it then uses to validate the encryption.

After the timer starts, the DSM version distributes the encryption and decryption keys to all participating threads from the root thread. Once the keys are distributed, the array to be encrypted is divided among the threads and distributed to them by the root node, using `arraycopy`. Since all remote threads request their portion of the array, a barrier is used to block progress until all threads have requested their data. Once again, this is to eliminate the “network reactivity” problem on the root thread. Next, each thread encrypts and decrypts its portion of the array using the distributed keys. Once a thread finishes, it
sends the decrypted portion back to the root node via `arraycopy`. After a final barrier is reached, the timer is stopped. The original multithreaded version of the benchmark created the whole array on one thread and the remote threads accessed their own portion of the array from the root thread independently from the other threads. However, this type of access still causes significant network traffic and the root thread becomes a bottleneck while servicing the requests.

The MPI version is similar to the DSM version, but it does not distribute the encryption and decryption keys to each process. Each process generates its own keys, which are the same. The MPI version uses `MPI_Ssend` and `MPI_Recv` to distribute the array data from the root process to the participating processes. Another difference is that the MPI version uses a barrier after the encryption and decryption of the array whereas the

![Figure 12: Crypt Sample Mean on the Star Cluster](image-url)
DSM places the barrier before the encrypt and decrypt to address the “network reactivity” problem. After this barrier, the MPI version uses the MPI_Ssend and MPI_Recv commands to return the decrypted portion of the array to the root process. The timer is stopped after a final barrier.

Figure 12 and Figure 13 show the results from the Star and Paraski clusters, respectively. The results of DSM and MPI are similar, but an anomalous DSM result exists with 4 and 8 nodes on Paraski. This may be due to the positioning of the barrier used in both the DSM and MPI version. The important difference here is that a process in the MPI version can continue once the process receives the data independent of the other waiting processes while the DSM version uses a barrier that requires all threads to wait until every thread has received the data before encrypting (“network reactivity”).
Sparse Matrix Multiplication

The sparse matrix multiplication benchmark uses an unstructured $N \times N$ sparse matrix that is stored in compressed-row format. For this benchmark, $N$ was 50,000 and the matrix is used for 200 iterations, exercising indirection addressing and non-regular memory references. Data is distributed to all nodes before timing is started. Only the matrix multiplication loop and the communication for building the result array are timed.

The original implementation of the multithreaded version required all threads to access the same member variable, created on the root thread, from each thread. Of course in Hyperion, this causes excessive network access. The benchmark was modified such that only a single `arraycopy` and a barrier are required for communication during the timing of

![Figure 14: Sparse Matrix Multiplication Sample Mean on the Star Cluster](image)

The plot shows the sample mean time in seconds for different numbers of nodes using MPI and DSM. The time decreases as the number of nodes increases, indicating improved performance with more nodes.

this benchmark. Each portion of the matrix is multiplied serially in a loop over the matrix. The barrier is used to coordinate the threads so the timer can be stopped.

The MPI version, oddly enough, uses MPI_Allreduce to collect the result of the matrix multiplication. The MPI_Allreduce command reduces a series of array values into a single array, in this case summing them. The end result is that all processes contain the final result. After a barrier, the timer is stopped.

Figure 14 and Figure 15 display the results obtained from the Star and Paraski clusters, respectively. Since the only communication timed is the building of the result array, which requires very little network usage, the network and communication protocols are not significant and do not impact performance. Both the DSM and the MPI versions of the benchmark parallelize and they both perform similarly. However, with this benchmark the MPI version suffers as more nodes are added. This is most likely due to the way the
benchmark was designed. Instead of using send and receive methods to build the final array, the benchmark uses the MPI_Allreduce method. Since only the root node need know the answer for validation reasons, using this method is causing unneeded communication. The DSM version simply uses arraycopy to simulate a send and receive environment, which bests MPI as more nodes are added on both the Star and Paraski cluster.

**Molecular Dynamics**

The molecular dynamic benchmark is an N-body code, with N set to 2048, which models particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The Lennard-Jones potential is an effective potential that describes the interaction between two uncharged molecules or atoms. The particle force calculation is the most computational intense part of the benchmark. This calculation involves an outer loop operating over all particles in the system and an inner loop with a range of the current particle number to the total number of particles. Parallelization has been added to the outer loop by dividing the range of iterations between nodes. This is done in a cyclic manner to avoid load imbalance. All nodes maintain their own copy of the particle data.

Once again, multiple object access on the root thread makes using the original multithreaded version not possible. After the timer is started, each thread in the modified DSM version moves the particles it is responsible for and updates velocities and forces. The updated forces are copied to the root thread using arraycopy and a barrier is employed to block all threads until all forces are copied. The root thread then updates the
entire force array with the local values copied from each thread and distributes the portions of the updated array to each thread with `arraycopy`. Another barrier is used to hold the threads from continuing until the root thread is finished with the distribution. The simulation continues for 50 time steps. The timer is stopped after a final barrier.

![Graph showing simulation results on Star and Paraski clusters](Figure 16: Molecular Dynamics Sample Mean on the Star Cluster)

The MPI version places a barrier after the particle move as well as after the local force calculation. The `MPI_Allreduce` command is used six times to update the x, y, and z forces and the potential energy, virial (Virial theorem), and interactions on all processes. This eliminates the need for one process to coordinate all force updates. A barrier is reached after all the `MPI_Allreduce` commands complete. The simulation completes and after the final barrier, the timer is stopped.

Figure 16 and Figure 17 show the results obtained on the Star and Paraski clusters,
respectively. The DSM version on the Star cluster suffers from requiring the root thread to update the particle forces on all remote threads, which requires all threads to first send local force changes to the root thread and then the root thread combines all the remote forces and sends the new force data back to all the remote threads. The MPI version uses the MPI_Allreduce method to sum all the local force changes on all nodes. On the Paraski cluster this extra work is not as significant given the efficient communication of the cluster, which leads to better results for the DSM version.

![Figure 17: Molecular Dynamics Sample Mean on the Paraski Cluster](image)

**Ray Tracer**

The ray tracer benchmark measures the performance of a 3D ray tracer. The scene contains 64 spheres and is rendered at a resolution of $N \times N$ pixels, where $N$ is 150. The benchmark was parallelized by using a cyclic distribution for load balance when looping
over the rows of pixels.

![Figure 18: Ray Tracer Sample Mean on the Star Cluster](image)

The original benchmark, both multithreaded and MPI, was poorly designed. The benchmark created far too many temporary objects, which severely impacted performance. Even comments in the JGF code suggested that they should reduce the number of temporary variables created. Both MPI and DSM versions were modified to eliminate temporary object creation during the timed part of the benchmark.

The DSM version uses Hyperion’s broadcast to transmit the row array reference to each thread. The rendering algorithm renders the scene and sends the row data back to the root thread with `arraycopy`. Using Hyperion’s Reduction class, a pixel-based checksum is generated on all nodes. Next, the root node places all the row data received from the remote threads into their final position in the row array. The timer is stopped after a
barrier is reached.

After the scene is rendered in the MPI version, an MPI_Reduce command is issued to create the pixel checksum used for validation. After this, MPI_Send and MPI_Recv commands are called to send the row data back to the root node after which the timer is stopped.

The results on the Star and Paraski clusters are listed in Figure 18 and Figure 19, respectively. The overall cost of data distribution is low in this benchmark as data is created locally on all the nodes and only a small amount of result data is actually distributed. The net result is that both versions of the benchmark perform almost identically.

Figure 19: Ray Tracer Sample Mean on the Paraski Cluster
Monte Carlo

The Monte Carlo benchmark is a financial simulation using Monte Carlo techniques to price products derived from the worth of an underlying asset. The benchmark generates N samples, where N is 2,000, with the same mean and fluctuation as a series of historical data. Dividing the work in block fashion in the principal loop over the Monte Carlo runs parallelizes the benchmark.

The results for this benchmark do not contain a DSM version due to the nature of the design of the original multithreaded benchmark. The Monte Carlo benchmark creates a large amount of object data on each participating node, which it distributes to the root node when complete. Hyperion is allowed to allocate a fixed amount of memory that is shared amongst all processors. By default this amount is 80 megabytes, thus with 4 nodes participating each node would only have 20 megabytes of memory to work with. The amount of memory required for each thread created by the Monte Carlo benchmark exceeds this limit, causing Hyperion to fail. We considered increasing the base memory, but we felt that this benchmark exceeded a reasonable limit for the amount of memory it consumed. When using MPI, Hyperion is configured in a single-process shared memory mode. In this mode all threads are created on the single machine and each node in the cluster executing Hyperion is independent from the other. This allows the full 80 megabytes of memory to be available for use on each node.

Since this is yet another example of an embarrassingly parallel algorithm (as is Raytracer, Crypt and Sparse Matrix Multiplication), we felt that it was unnecessary to convert the large amount of code that made up this benchmark into a working DSM benchmark. We did feel that the MPI version of the benchmark was worth covering.
Figure 20 and Figure 21 show the results on the Star and Paraski clusters, respectively.

The javaMPI implementation allows Java objects to be distributed with standard MPI calls by supplying a javaMPI specific data-type, MPI.OBJECT, to the MPI commands. The Monte Carlo benchmark uses the Java object-based send and receive methods to distribute the price data. When a Java object is distributed, javaMPI serializes the object when sent and deserializes it when received. Hyperion does not implement object serialization, so an object serialization mechanism was developed to facilitate the Java object-based javaMPI methods. While this benchmark was not the only one to use Java object-based javaMPI methods, it is the benchmark that stressed the serialization code the most, exposing some initial implementation flaws.

Initially, we planned on using the native implementation of MPI to launch the MPI
benchmarks that were compiled by Hyperion. While testing this benchmark, we found a performance problem began when executing the benchmark on two or more nodes. As more nodes were added, the performance improved, but was still much worse than expected.

Figure 21: Monte Carlo Sample Mean on the Paraski Cluster

Locking primitives are used when Java performs synchronization with the `synchronized` keyword. In this benchmark, the Java class “Random” is used to obtain a random number. The method “nextGausean” from the “Random” class uses synchronization to protect the random object from being modified from multiple threads. This method is repeatedly called from a loop in the benchmark. Removing the synchronization keyword from the “Random” class returned the performance to what was expected. However, modifying Java was not an ideal solution.
The default build for the single node version of Hyperion uses POSIX threads. We found that the POSIX thread implementation of lock and unlock are inefficient and about three times slower than the PM2 thread implementation. This deficiency forced us to recompile Hyperion with the PM2 library, allowing PM2 to be used for locking in the MPI benchmarks thereby replacing POSIX. This required the benchmarks to be loaded by the PM2 loading mechanisms even though PM2 was not used to distribute any data as the javaMPI benchmarks do not create Java threads. We still use the MPI launching mechanisms to distribute the PM2 processes to each node. This made for a slightly complex benchmark launching procedure.
For this thesis we wanted to test the performance of Hyperion using a variety of benchmarks on multiple cluster environments. The results obtained showed us that DSM could perform close to MPI, a well-established model for distributed communication, in many situations. However, to achieve this performance, a user of Hyperion simply cannot develop a multithreaded Java application, compile it with Hyperion and expect parallelized performance on a cluster. We feel that the user needs to be aware of how the threads interact in a cluster environment. Techniques, like those described in Chapter 3, should be employed by the user to facilitate parallelized performance when in the cluster environment. We also feel that these techniques, while essential in the cluster environment, can also benefit a user on a symmetric multiprocessor (SMP) machine.

The Java language specification lacks facilities to control groups of threads running on separate processors. Simply using synchronization to control object access is not sufficient for a DSM model. Even in a single node shared-memory environment, studies suggest that a program developed using a nonshared-memory programming model performs better than one written in the traditional shared-memory programming model\[8]. In order for developers to create efficient Hyperion-based implementations, they need access to barrier, reduction and broadcast mechanisms. While Hyperion implements these mechanisms, the best place for these mechanisms would be as part of the Java language API. There is some movement towards this. JSR (Java Specification Requests) 166, part
of Sun Microsystem’s Java Community Process (JCP), the method for enhancing the Java platform, proposes new concurrency utilities and among these new utilities are barrier and reduction implementations.

It is possible, however, to achieve high-performance when a multithreaded program is optimized with DSM in mind. Figure 22 and Figure 23 show the performance ratio of the DSM benchmarks to the MPI benchmarks on both clusters. On the Star cluster, four of the optimized DSM benchmarks run comparable with the MPI version benchmarks. The other three benchmarks suffer from network saturation and root node bottlenecks caused by excessive network traffic to a single node, which manifests itself as more nodes are added. On the Paraski cluster we do not see this, as the network bandwidth is large enough to minimize the bottleneck effect and all benchmarks are very
close to a 1 to 1 ratio.

We firmly believe that mpiJava is not a “clean” solution for cluster computing. MPI emerged from Fortran-based and C-based environments and has evolved over the years based on specific requirements for these languages. These improvements make MPI very specific to a procedural-based language API, which does not translate well to a object-based API. Even a 100% pure Java implementation of MPI would still be a confusing API with an excess of antiquated methods. Hyperion, despite its current limitations, is a far cleaner solution when considering cluster computing for the Java language.

Ultimately, with the modifications to the DSM benchmarks, we can see that the DSM benchmarks resemble message-passing more than a shared-memory approach. Data
is sent and received explicitly using `arraycopy`, reductions are employed, data is broadcast
to all threads and barriers are used to control execution. When comparing the DSM code
with the MPI code we can see many similarities, which might be thought of as negating
the inherent advantages of DSM. However, these additional methods do not require
explicit knowledge of the nodes they interact with to coordinate data, allowing the cluster
detail to remain hidden from the Hyperion user. This is the most compelling reason to use
Hyperion and reinforces the “clean” approach we describe above.

The most promising course for future work is optimizing the Hyperion extensions.
The current broadcast facility is built upon the reduction class. This class forces
participating threads to engage in a full exchange of data. A full exchange requires a
thread to both send and receive data from a neighbor thread before exchanging with
another neighbor. Figure 24 shows an example of a full exchange with four nodes. The full
exchange allows Hyperion to perform reductions in $\log P$ steps. A full exchange makes
sense in the reduction case as all nodes have a portion of the answer. When performing a
broadcast, only one node has the data initially. By imposing a full exchange, the first
exchange only has one message that contains usable data, the others contain an empty
message. In this case a full exchange is not necessary and is actually degrading
performance. Figure 25 is an example of how broadcast could be implemented to reduce
the number of messages. In the example case the number of messages is reduced from
eight to three. By avoiding a full exchange, threads can immediately continue broadcasting
to their neighbors without waiting for a response. In the four node example, thread one
has sent its data to thread two it can begin sending its data to thread three. In a full
exchange, thread one would wait for a response from thread two before sending to thread
Hyperion also lacks array-based reductions. An AllReduce extension to Hyperion would greatly improve the performance of the Molecular Dynamics benchmark. Currently, the root thread has to collect all forces from the remote threads, update them serially and redistribute the update forces back to the remote threads. An efficient AllReduce could perform the same function avoiding this overhead. This is a common task that could easily fit within the DSM model, much like the current reduction mechanism.
By providing a robust, efficient and easy to use thread-group API in Hyperion, we feel that developers will be able to produce quality, high-performance applications. These applications would perform well on a variety of clusters as well as SMP machines and would not require the developer to coordinate data exchange, preserving the DSM model. Developers would produce simpler code when compared to MPI and this code would be highly portable. If this API was developed to its potential, we believe that Hyperion can be a realistic alternative for high-performance cluster computing.
APPENDIX

The following tables contain the confidence intervals for the benchmarks used in this thesis. The values \((c_1, c_2)\) represent the lower and upper bounds that the mean exists in. The first column indicates the number of nodes used to obtain the interval.

### Table 1: Fourier Coefficient Analysis Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(c_1)</td>
<td>(c_2)</td>
</tr>
<tr>
<td>1</td>
<td>16.75971199</td>
<td>16.76212541</td>
</tr>
<tr>
<td>2</td>
<td>8.381693829</td>
<td>8.384558931</td>
</tr>
<tr>
<td>4</td>
<td>4.197539041</td>
<td>4.198202719</td>
</tr>
<tr>
<td>8</td>
<td>2.1100579</td>
<td>2.11018546</td>
</tr>
<tr>
<td>16</td>
<td>1.07263566</td>
<td>1.0727683</td>
</tr>
</tbody>
</table>

### Table 2: Fourier Coefficient Analysis Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(c_1)</td>
<td>(c_2)</td>
</tr>
<tr>
<td>2</td>
<td>4.875200038</td>
<td>4.875279322</td>
</tr>
<tr>
<td>4</td>
<td>2.436090053</td>
<td>2.436109107</td>
</tr>
<tr>
<td>8</td>
<td>1.218789301</td>
<td>1.218808539</td>
</tr>
<tr>
<td>16</td>
<td>0.611438052</td>
<td>0.611450988</td>
</tr>
</tbody>
</table>

### Table 3: LU Factorization Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(c_1)</td>
<td>(c_2)</td>
</tr>
<tr>
<td>1</td>
<td>4.725273548</td>
<td>4.732586892</td>
</tr>
<tr>
<td>2</td>
<td>2.828902477</td>
<td>2.831270683</td>
</tr>
<tr>
<td>4</td>
<td>1.820170731</td>
<td>1.822021389</td>
</tr>
<tr>
<td>8</td>
<td>1.40019145</td>
<td>1.40251919</td>
</tr>
<tr>
<td>16</td>
<td>1.317813382</td>
<td>1.322460258</td>
</tr>
</tbody>
</table>
Table 4: LU Factorization Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.065821077</td>
<td>3.066425403</td>
<td>3.304060451</td>
<td>3.304103309</td>
</tr>
<tr>
<td>2</td>
<td>1.687429419</td>
<td>1.687742261</td>
<td>2.06590876</td>
<td>2.06651612</td>
</tr>
<tr>
<td>4</td>
<td>0.856623624</td>
<td>0.856829216</td>
<td>1.122575654</td>
<td>1.122707826</td>
</tr>
<tr>
<td>8</td>
<td>0.391311865</td>
<td>0.391322615</td>
<td>0.610776684</td>
<td>0.610941036</td>
</tr>
<tr>
<td>16</td>
<td>0.277675262</td>
<td>0.277679738</td>
<td>0.460495463</td>
<td>0.460534057</td>
</tr>
</tbody>
</table>

Table 5: SOR Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.70743683</td>
<td>15.71949129</td>
<td>15.56794253</td>
<td>15.57375599</td>
</tr>
<tr>
<td>2</td>
<td>8.804101363</td>
<td>8.823497437</td>
<td>9.278081083</td>
<td>9.302506677</td>
</tr>
<tr>
<td>4</td>
<td>5.11738994</td>
<td>5.12471726</td>
<td>6.522592754</td>
<td>6.68699926</td>
</tr>
<tr>
<td>8</td>
<td>3.234492717</td>
<td>3.236685643</td>
<td>4.953414233</td>
<td>5.037642647</td>
</tr>
<tr>
<td>16</td>
<td>2.300499355</td>
<td>2.30256325</td>
<td>5.140862882</td>
<td>5.357486718</td>
</tr>
</tbody>
</table>

Table 6: SOR Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.92706294</td>
<td>12.9275019</td>
<td>13.84974912</td>
<td>13.85181124</td>
</tr>
<tr>
<td>4</td>
<td>3.62059561</td>
<td>3.620205599</td>
<td>4.986194293</td>
<td>4.986578187</td>
</tr>
<tr>
<td>8</td>
<td>1.986404093</td>
<td>1.986474187</td>
<td>2.657566992</td>
<td>2.657733088</td>
</tr>
<tr>
<td>16</td>
<td>1.185459957</td>
<td>1.185494163</td>
<td>1.500752582</td>
<td>1.500785738</td>
</tr>
</tbody>
</table>

Table 7: Crypt Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.441310531</td>
<td>4.442243029</td>
<td>4.35603871</td>
<td>4.36007193</td>
</tr>
<tr>
<td>2</td>
<td>2.49739359</td>
<td>2.49817105</td>
<td>2.375701837</td>
<td>2.375997643</td>
</tr>
<tr>
<td>4</td>
<td>1.518991183</td>
<td>1.519128577</td>
<td>1.314144258</td>
<td>1.318552062</td>
</tr>
<tr>
<td>8</td>
<td>1.02669681</td>
<td>1.029124519</td>
<td>0.908402777</td>
<td>0.923901423</td>
</tr>
<tr>
<td>16</td>
<td>0.78543886</td>
<td>0.78572478</td>
<td>0.779109043</td>
<td>0.814775637</td>
</tr>
</tbody>
</table>
Table 8: Crypt Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.09199389</td>
<td>3.09207795</td>
<td>3.04407678</td>
<td>3.044092133</td>
</tr>
<tr>
<td>2</td>
<td>1.606202114</td>
<td>1.606328246</td>
<td>1.631397228</td>
<td>1.631429172</td>
</tr>
<tr>
<td>4</td>
<td>0.859972061</td>
<td>0.859999939</td>
<td>1.002962636</td>
<td>1.002989564</td>
</tr>
<tr>
<td>8</td>
<td>0.484815695</td>
<td>0.484826585</td>
<td>0.768678263</td>
<td>0.768718217</td>
</tr>
<tr>
<td>16</td>
<td>0.307329003</td>
<td>0.307338917</td>
<td>0.396950713</td>
<td>0.396958807</td>
</tr>
</tbody>
</table>

Table 9: Sparse Matrix Multiplication Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18.98306678</td>
<td>19.0031359</td>
<td>19.3539695</td>
<td>19.38264334</td>
</tr>
<tr>
<td>4</td>
<td>5.026763149</td>
<td>5.030376411</td>
<td>4.00640779</td>
<td>4.02248425</td>
</tr>
<tr>
<td>8</td>
<td>2.625794599</td>
<td>2.62653201</td>
<td>1.84522546</td>
<td>1.857396014</td>
</tr>
<tr>
<td>16</td>
<td>1.503327365</td>
<td>1.505126355</td>
<td>0.903204595</td>
<td>1.003844445</td>
</tr>
</tbody>
</table>

Table 10: Sparse Matrix Multiplication Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.67811264</td>
<td>17.67917536</td>
<td>18.62890501</td>
<td>18.62925091</td>
</tr>
<tr>
<td>2</td>
<td>8.932575242</td>
<td>8.933061838</td>
<td>8.64134536</td>
<td>8.641987444</td>
</tr>
<tr>
<td>4</td>
<td>4.559391463</td>
<td>4.559724497</td>
<td>3.912273874</td>
<td>3.91272726</td>
</tr>
<tr>
<td>8</td>
<td>2.310062851</td>
<td>2.310137749</td>
<td>2.12486293</td>
<td>2.12516827</td>
</tr>
<tr>
<td>16</td>
<td>1.193174125</td>
<td>1.193207675</td>
<td>0.794293703</td>
<td>0.794621417</td>
</tr>
</tbody>
</table>

Table 11: Molecular Dynamics Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23.30897202</td>
<td>24.26478582</td>
<td>25.583873</td>
<td>26.41785824</td>
</tr>
<tr>
<td>4</td>
<td>7.373158898</td>
<td>7.583997622</td>
<td>7.928852168</td>
<td>8.071381472</td>
</tr>
<tr>
<td>8</td>
<td>4.956405746</td>
<td>5.04087454</td>
<td>7.41682748</td>
<td>7.47742292</td>
</tr>
<tr>
<td>16</td>
<td>4.011910379</td>
<td>4.046467061</td>
<td>14.52606339</td>
<td>15.40037297</td>
</tr>
</tbody>
</table>

Table 12: Molecular Dynamics Confidence Interval on the Paraski Cluster
Table 12: Molecular Dynamics Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DSM</th>
<th></th>
<th>MPI</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c_1$</td>
<td>$c_2$</td>
<td></td>
<td>$c_1$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>1</td>
<td>15.59464999</td>
<td>15.59514797</td>
<td>17.25145256</td>
<td>17.25276312</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7.96859946</td>
<td>7.96879474</td>
<td>12.61990002</td>
<td>12.62016226</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.184065543</td>
<td>4.184153217</td>
<td>6.491183843</td>
<td>6.491435117</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.314139621</td>
<td>2.314199779</td>
<td>3.665834678</td>
<td>3.668304002</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.498320486</td>
<td>1.500340594</td>
<td>2.431286286</td>
<td>2.43227594</td>
<td></td>
</tr>
</tbody>
</table>

Table 13: Ray Tracer Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DSM</th>
<th></th>
<th>MPI</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c_1$</td>
<td>$c_2$</td>
<td></td>
<td>$c_1$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>1</td>
<td>26.08764918</td>
<td>26.09146566</td>
<td>25.91626657</td>
<td>25.91884975</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6.512668778</td>
<td>6.520054862</td>
<td>6.702928034</td>
<td>6.709530526</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.29939969</td>
<td>3.30085775</td>
<td>3.64678996</td>
<td>3.662303844</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.656418635</td>
<td>1.656467885</td>
<td>2.08173854</td>
<td>2.10077406</td>
<td></td>
</tr>
</tbody>
</table>

Table 14: Ray Tracer Confidence Interval on the Paraski Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DSM</th>
<th></th>
<th>MPI</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c_1$</td>
<td>$c_2$</td>
<td></td>
<td>$c_1$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>1</td>
<td>15.92339618</td>
<td>15.92350998</td>
<td>15.80337214</td>
<td>15.8035209</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7.945288705</td>
<td>7.94531015</td>
<td>8.580265774</td>
<td>8.58046666</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.981141738</td>
<td>3.981237502</td>
<td>4.41024664</td>
<td>4.41036284</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.05485844</td>
<td>2.05493148</td>
<td>2.29422121</td>
<td>2.29425179</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.04031629</td>
<td>1.040882411</td>
<td>1.15046603</td>
<td>1.150475957</td>
<td></td>
</tr>
</tbody>
</table>

Table 15: Monte Carlo Confidence Interval on the Star Cluster

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th></th>
<th></th>
<th>MPI</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c_1$</td>
<td></td>
<td>$c_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>63.32977356</td>
<td></td>
<td>63.338471</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>37.1814641</td>
<td>37.19423054</td>
<td></td>
<td>37.19423054</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>23.24584538</td>
<td>23.25246698</td>
<td></td>
<td>23.25246698</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>16.30466939</td>
<td>16.31078381</td>
<td></td>
<td>16.31078381</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>12.83423961</td>
<td>12.83683331</td>
<td></td>
<td>12.83683331</td>
<td></td>
</tr>
</tbody>
</table>

Table 16: Monte Carlo Confidence Interval on the Paraski Cluster
LIST OF REFERENCES


5. W. Gropp, E. Lusk, User’s guide for MPICH, a portable implementation of MPI, ANL-96/6, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

