Spring 5-2010

Guppie: A Coordination Framework for Parallel Processing Using Shared Memory Featuring A Master-Worker Relationship

Sean Christopher McCarthy
University of Southern Mississippi

The University of Southern Mississippi

GUPPIE: A COORDINATION FRAMEWORK FOR PARALLEL PROCESSING USING SHARED MEMORY FEATURING A MASTER-WORKER RELATIONSHIP

by

Sean Christopher McCarthy

Abstract of a Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

May 2010

ABSTRACT

GUPPIE: A COORDINATION FRAMEWORK FOR PARALLEL PROCESSING USING SHARED MEMORY FEATURING A MASTER-WORKER RELATIONSHIP

by Sean Christopher McCarthy

May 2010

Most programs can be parallelized to some extent. The processing power available in computers today makes parallel computing more desirable and attainable than ever before. Many machines today have multiple processors or multiple processing cores making parallel computing more available locally, as well as over a network. In order for parallel applications to be written, they require a computing language, such as C++, and a coordination language (or library), such as Linda. This research involves the creation and implementation of a coordination framework, Guppie, which is easy to use, similar to Linda, but provides more efficiency when dealing with large amounts of messages and data. Greater efficiency can be achieved in coarse-grained parallel computing through the use of shared memory managed through a master-worker relationship.

DEDICATION

To My Mother, Father, Grandfather, and Wife Kimberly

ACKNOWLEDGEMENTS

I would like to thank Dr. Ray Seyfarth and the other committee members, Dr. Joe Zhang, Dr. Louise Perkins, Dr. Andrew Strelzoff, and Dr. Jonathan Sun for their advice and support throughout this long process. I would especially like to thank Dr. Seyfarth. He has helped me tremendously and kept me motivated throughout my doctoral study. I would also like to thank Dr. Perkins for taking me under her wing after my senior level software engineering class. She has always believed in me, which in turn helped me to believe in myself.

I would also like to thank my family. My wife Kimberly has supported my decision to remain in school the entire time we have known each other. I would also like to thank my mother and father. I never would have been able to enroll in graduate school if it were not for them. They have helped to support me emotionally and financially all the way through.

I also want to take the time to acknowledge my grandfather. He died when I was very young, but it was his dream to one day have a doctor in the family. I know that he would be very proud of me if he were alive today.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

CHAPTER 1

INTRODUCTION

Parallel computing consists of dividing a problem composed of independent sub-problems into multiple tasks that can be solved simultaneously. These independent tasks can be spread among multiple processors with each processor responsible for operating on its set of tasks. Once all processors have completed their tasks, the results are combined to form a solution to the original problem. Dividing an independent problem into multiple tasks, sending the tasks to multiple processors, performing a specific set of instructions by each processor on its tasks, and then combining the results of those instructions are the main steps in parallel computing.

Parallel processing can be achieved on either a local machine with multiple processors or over a network consisting of multiple machines. The machines connected by a network can either be multi-core or single-core. As long as there are multiple processors that can be used to solve a problem, and that problem can be divided into multiple tasks, parallel computing can be achieved.

The current trend in parallel computing is to use the processors in multiple machines connected by a network. This is referred to as distributed memory parallel processing. The computers used in distributed memory parallel processing might be in a dedicated cluster or even a collection of computers connected over the Internet. This is beneficial in that any machine with a processor (or multiple processors) can be added to the network to help parallelize a problem. The disadvantage to using multiple machines connected by a network is that there is added communication cost involved when sending

data from one process to another. This communication cost can be substantial when working with large data sets, such as images. For instance, sending an image containing one thousand rows and columns consists of sending one million pixels. Each pixel consists of red, green, and blue values, so the user is sending three million values from one process on one machine to another process on another machine. That is three million values that have to be sent and received by processes before computation is even started! There is another parallel computing paradigm to consider, and that paradigm is shared memory.

When multiple CPUs or multiple processor cores exist within a single machine, the user can take advantage of shared memory to help perform parallel computations. This is very advantageous when working with large data sets, such as images. By placing an image into shared memory, multiple processes can connect to the shared memory to retrieve the image, rather than having to send the entire image across a network to a process on another machine. The downside to this approach is that dedicated clusters can be built using thousands of processors while shared memory computers typically have fewer processors.

Both the shared memory and distributed memory parallel processing paradigms are analyzed in this dissertation. The two most popular parallel processing libraries, MPI and OpenMP, are discussed and compared to the Guppie parallel processing coordination framework developed as part of this research. Guppie utilizes shared memory features to reduce the need for communication and also provides a lot of the same features seen in the Linda coordination language. Like Linda, Guppie features a shared tuple space which is basically a shared associative memory system. In addition to the tuple space, Guppie

includes facilities for managing actual shared memory regions. By using Guppie's shared memory features, along with its tuple space, it can be shown that Guppie is easier to use, and when working with large data sets, Guppie achieves better performance than its counterparts, MPI and OpenMP.

In the following two chapters, coordination languages are explained and discussed. Then, the two most popular parallel processing libraries, OpenMP and MPI, are discussed. Guppie follows the OpenMP and MPI chapters. Then, a parallel image processing application written in both MPI and Guppie is compared, along with its CPU time, speedup, and efficiency results.

CHAPTER II

COORDINATION LANGUAGES

There are two major categories of parallel computing architectures which have been used in commercial systems: Single-Instruction Multiple-Data (SIMD) and Multiple-Instruction Multiple-Data (MIMD) [34]. An example of a SIMD system is the Connection Machine 1 [50] which was popular for a brief period. The current trend in parallel computing is MIMD systems built using collections of microprocessor-based computers connected by networks.

A typical parallel application is built using both a computation and a coordination model[1]. A computation model can be a programming language, such as C++, Java, or Fortran. The computation model is used by programmers to build a serialized application in a computation language. A serialized application is a program that executes code one step at a time. Serialized code can be parallelized, at least to a certain degree, by combining a coordination model with a computational model.

A coordination model coordinates how multiple processes function within the same application. The coordination model might be implemented as a library or as extensions to a language, forming a coordination language. MPI is an example of a coordination model implemented as a library [45]. OpenMP is a coordination model built with language extensions and a library [28]. Linda is an example of a coordination language [5,13,15,19].

[1] A SIMD application might be built with a language with parallel constructs and implicit coordination.

It is possible to combine a computation model and a coordination model into a single language, but they are usually treated separately and combined together to form a parallel application. For example, the computation language C++ is its own stand-alone language, but it can be combined with a coordination language such as Linda or MPI to create a parallel application.

Linda

In order to understand how Guppie works, Linda should first be explained. Linda is a coordination model, and at its core is its tuple space [4,9,11,18,6]. A tuple is an ordered sequence of data, such as a string to represent a name, an integer to represent an identifier (id), and some values associated with that name and id. The tuple space is simply a collection of tuples. As well as creating the tuple space, Linda coordinates the way that processes interact with the tuple space.

Linda provides two different kinds of tuples, process tuples and data tuples. Process tuples are active, meaning that they can exchange data by generating, reading, and consuming data tuples. Data tuples are passive, and are the result of an executed process tuple. Another important feature of tuples is that they are atomic, meaning that a single attribute within an individual tuple cannot be modified unless the entire tuple is removed from the tuple space, modified, and placed back into the tuple space.

It is important to note that Linda is a distributed associative memory model, not a tool. There are many existing tools that implement the Linda memory model, such as implementations in Java [25,26] and C, such as CLinda [8,48]. Most implementations have the same set of Linda operations, so only CLinda, is described here. CLinda

provides functions to interact with the tuple space. CLinda has four main functions: out, in, rd, and eval.

out(t) causes tuple t to be added to the tuple space. The executing process continues immediately. A tuple, as mentioned in the opening paragraph of this section, is a series of typed values, for example out("myData", 3.87, x, 5).

in(s) causes some tuple t that matches anti-tuple s to be withdrawn from the tuple space. An anti-tuple is some series of typed fields. These fields can either be actuals or formals. An actual is an actual field value, such as the string "myData" or the integer "3". Formals are place-holders for values that you are hoping to withdraw from the tuple that matches your actuals within the tuple space. A formal is prefixed with a question mark, for example ("myData", ?a, y, ?b) [2]. Once in(s) has found a tuple t matching all the actuals in s, the values of the actuals in t are assigned to the corresponding formals in s, and the executing process continues. In the example t would match “myData” and y, while variables a and b would receive values from t. If no matching tuple t is available when in(s) executes, the executing process suspends until a matching tuple is available, and then the process proceeds as normal. If many matching tuples are available, then one is chosen arbitrarily.

rd(s) is the same as in(s) except that the matched tuple remains in the tuple space after rd(s) is executed.

eval(t) is the same as out(t) except that t is evaluated after, rather than before, it enters the tuple space. For example, suppose a user creates a function named

[2] The use of the question mark, along with a few other syntactic issues, typically means that a special language processor is required in a Linda implementation.

"factorial(int)", where an integer is passed as the parameter to the function, and the function calculates the factorial of that parameter. The following eval function call eval("fact", 5, factorial(5)) places an active tuple into the tuple space. When the tuple arrives in the tuple space, Linda creates a process to evaluate the function "factorial(5)". Once evaluated, the computed value of "factorial(5)" replaces the function call in the tuple, and the tuple becomes a passive data tuple.

There are also two predicate forms of the in and rd functions, inp and rdp. The only difference between these two forms is that they do not suspend if a matching tuple is not immediately available. inp and rdp return immediately with a true or false value.

Linda's Preprocessor

Linda allows an arbitrary number of parameters in a tuple, and for input operations any of these parameters not preceded by a question mark are used for tuple selection. Versions of Linda have been developed for C for which a preprocessor is required to convert Linda operations into proper C syntax. There are several tasks that the Linda preprocessor must perform. The preprocessor must build a symbol table for determining variable types used in Linda operations, generate tuple usage information that may be used for load balancing, and modify the text of Linda calls for variable argument processing [1,2,10,12].

The Linda preprocessor has two phases. In the first phase, LPP-I, the preprocessor expands all the C preprocessor directives, such as the **#include** header files. This is done in a single pass and then passed to a C compiler for error checking in the C part of the program. Once the errors are corrected, the input is ready for the second

phase, LPP-II. LPP-II checks for errors in Linda's defined operators, such as in, out, rd, and eval. LPP-II then translates the code to the format needed by the kernel.

The C programming language has a preprocessor of its own, and Linda's preprocessor functions in a similar manner. The C preprocessor handles the directives for source file inclusion, such as the header files in the **#include** header statements found at the top of any C program. The preprocessor also handles **#define** macro definitions and **#if** conditional inclusions. The preprocessor translates source code into an acceptable form for the compiler. Linda's preprocessor performs similar operations in order for the code to be compilation ready.

The preprocessor is also used to facilitate the run-time search. This is because Linda's tuple space requires a great deal of searching for every Linda operation. One responsibility of the Linda preprocessor is to match similar operators together based on the operator's type of tuple space access. For example, the Linda operators "in" and "rd" are matched with "out" and "eval" and placed into groups based on matching tuple patterns. Then, they are mapped onto the appropriate family of routines in the run-time library [17].

For C-Linda, the preprocessor is a C program that converts C-Linda programs into the syntax required by the C-Linda library. The preprocessor adds format strings containing type information to all Linda calls. This is achieved by scanning through the C-Linda program and tracking the types of variables and functions in their current contexts. There are some syntax modifications that need to be made as well. The Linda model uses the "?" operator for matching formals in the tuple space. The preprocessor

replaces each “?” with “&”. By replacing with an ampersand, a C compiler can pass the formal parameter by reference and assign the correct value associated with the matching tuple into the memory space addressed by the formal parameter. Replacing the “?” operator with “&” is done for all parameters in Linda calls, except for parameters of pointer types, such as strings and arrays. For these, the “?” operator is simply removed.

In order to compile C-Linda programs, a compiler interface must be used. A compiler interface is a C program that represents the C-Linda system in the form of a compiler. When a user wants to compile a C-Linda program, the compiler interface calls the preprocessor for every Linda source file, then produces the executable with the C compiler while linking with the C-Linda library.

CHAPTER III

LINDA EXTENSIONS

There are extensions of Linda's associate distributed memory model that solve problems encountered in Linda. FT-Linda [3] addresses issues with tolerating failures in the underlying computing platform. Bauhaus Linda [16,22] extends the Linda tuple space by having multiple tuple spaces in the form of multi-sets. Hybrid Law governed Linda [36] enforces a set of laws that all processes must follow in order to exchange tuples in the tuple space. Lime [29,44,46] utilizes mobile agents in their extension.

FT-Linda

One of Linda's shortcomings involves the inability to tolerate failures in the underlying computing platform. FT-Linda resolves this issue by providing stable tuple spaces and atomic execution of tuple space operations. Stable tuple spaces ensure that tuple values are not lost when failures occur. Atomic execution of tuple space operations allows tuple operations to be executed in an "all-or-nothing" manner, even despite failures and concurrency.

Linda's lack of support for failures has an impact in two major ways. An application written using Linda cannot be recovered after an unexpected failure. If the execution time is short, then this may not be of much concern, but if the execution time is rather long, such as is needed for a decryption algorithm, the application may need to be restarted from the beginning. The other major impact of Linda's inability to tolerate

failures regards the near impossibility to write critical applications such as process control.

Fault tolerance in FT-Linda is achieved through the use of replication and multicasting. Tuple spaces are replicated on multiple processors to ensure that data exists if a failure occurs, such as a processor failing. These tuple space copies are updated by use of an atomic multicast. This is an efficient implementation because only a single multicast message is needed for each collection of tuple space operations.

Bauhaus Linda

Bauhaus Linda is more powerful and simpler than the original Linda model. Bauhaus Linda simplifies Linda by combining similar features found in Linda into a single, collective feature. Bauhaus Linda generalizes Linda by not differentiating between tuples and tuple spaces, tuples and anti-tuples, or active (process) and passive (data) objects.

Tuples and tuple spaces are replaced by a single structure called a multiset. Normal Linda operations add (out), read (rd), and remove (in) tuples to and from a tuple space. Bauhaus uses the add, read, and remove functions to add or remove multisets to or from another multiset. This results in ordered tuples being replaced with unordered multisets.

In Bauhaus Linda a single tuple may function as a tuple space. Multiple elements may be added to or removed from a tuple. The opposite can be achieved as well. A tuple space may be treated like a single tuple. Bauhaus extends the out operation to allow tuple

space creation, just like it can create a tuple. The tuple space can then be manipulated as a single object. Since tuples and tuple spaces are treated as multisets, they acquire multiset operations such as union, intersection, and subtraction.

Bauhaus Linda does not differentiate between tuples and anti-tuples. Regarding Linda operations, out specifies a tuple, and in and rd specify an anti-tuple. Bauhaus is different in that all three operations take a multiset argument. The Bauhaus out operation adds a specified multiset to a target multiset, similar to how Linda would add a data tuple to a tuple space. However, since Bauhaus revolves around the use of multisets, Linda's type and position-sensitive associative matching rule is replaced by a simple set inclusion operation.

There is no distinction in Bauhaus Linda between active (process) and passive (data) objects. In Bauhaus, there is no need for an eval operation, as seen in the Linda model. Active and passive objects are still distinct in Bauhaus, but the way the coordination language handles them is not. Bauhaus uses the out operation to handle both active and passive objects. Reading a process object in Bauhaus results in the reader's acquisition of a suspended copy of the process. Removing a process object in Bauhaus results in the remover's acquisition of the process itself, not a copy.

Law-Governed Linda

Linda's process communication with a tuple space is very efficient and easy. However, it is too simple, thus making communication unsafe. Law-governed Linda provides more secure process-tuple space interaction by providing additional attributes within tuples.

Law-governed Linda uses a concept already introduced in centralized and message passing systems. There are laws that processes must abide by in order to communicate with the tuple space. These laws are needed because of Linda's inability to provide any security from eavesdropping or interference when a process must communicate through a network to access a distributed shared memory tuple space that is common in the Linda model and the majority of its extensions. Linda gives processes the ability to read and remove tuples at will, even though some tuples may be intended to be received by particular processes. The message passing needed for processes to communicate with a distributed shared memory tuple space assume that messages come from their apparent senders and are delivered to their intended recipients. However, this should not always be assumed due to problems in unrelated modules that could result in interference with a message.

Another problem regarding Linda's need for message passing is that it can be rather inefficient because it relies on an implicit convention, unknown to the Linda implementation. Linda's tuple space is usually distributed. The actual implementation, not the Linda model, is responsible for the placement and retrieval of tuples into the tuple space memory. The inefficiency is a result of the inability to determine the best placement for a tuple. Obviously the best place for a tuple would be within or close to a processor that uses it most frequently, but this could be rather difficult to achieve. Global optimization of a distributed shared memory tuple space is feasible, but also impractical. The time needed to organize the tuples with processors that use them most frequently would create overhead that is comparable to the time needed to access them without being globally optimized.

LIME

LIME (Linda in a Mobile Environment) is a model that supports the development of applications that exhibit physical mobility of hosts, logical mobility of agents, or both. Linda uses a global, distributed shared memory, tuple space. Lime has mobile agents carry their own tuple spaces, which are shared between agents.

Tuple spaces are also extended with a notion of location, and programs are given the ability to react to specified states. The resulting model provides a minimalist set of abstractions that facilitate rapid and dependable development of mobile applications

Lime assumes that mobile agents exist in a set of hosts. These hosts are then connected through some network. Agents can then move from one host to another while carrying their tuple spaces with them. Access to the tuple space is performed using an extended set of tuple space operations. Each agent may own multiple tuple spaces, and these tuple spaces may be shared with other agents. Tuple spaces can be shared by extending the contents of each agent's local tuple space to include any tuples that exist in other agents' tuple spaces.

Extensions Review

One common feature found in the Linda model and its extensions is the focus on the tuple space being placed in distributed shared memory, meaning that the memory allocated for the tuple space can span over several computers. While the Lime model does not technically put the tuple space in distributed shared memory, it is still distributed amongst the mobile agents. Some processes or agents may even have local tuple spaces.

Tools invoking Linda's memory model might be implemented on top of a message passing system like MPI. Data access control to the tuple space is also needed and can be achieved asynchronously through the use of a mutex or semaphore. When working with multiple computers using network communications, mutiple tuple server processes may be used as well. This involves defining a method for mapping tuples to servers so that there is a definite place to start searching for a tuple. There are more extensions than the ones listed above; however, none consider using shared memory to efficiently handle large data [38,39,41,42].

CHAPTER IV

OPENMP AND MPI

There are two main paradigms to consider for parallel computing. The first is shared memory multiprocessing (SMP). Historically SMP computers have been expensive computers with multiple processors sharing a single address space. Most of today's computers are multi-core computers, allowing shared memory parallel computing to be widely available. The other major paradigm is distributed memory parallel computing. For this method, processors communicate with each over through a network. This method is utilized in systems ranging from small clusters using commodity network switches to supercomputers consisting of thousands of computers connected with specially designed networks.

The main difference between the two paradigms is where the memory is stored. This effects how a parallel application is written. For shared memory parallel computing, different processes can access the same data. This is similar to referencing data in serialized programs but adds the complexity of shared data integrity. Shared data integrity deals with read/write issues. For example, if multiple processes need to just read from the shared data, then there is no issue as long as no other process is currently trying to write to the data. If one process needs to update a value contained in a shared memory variable, the process needs to lock the shared memory variable so no other process has access to it. Once the process that is reading or writing to the shared memory has completed its operation, the process unlocks the memory so the other processes can access it. In distributed memory parallel computing, one must decide how to distribute

the independent tasks available to be parallelized to multiple processes. Once the tasks have been distributed among the processes, each process performs its necessary operations, and then each process sends results back to a single or multiple processes (depending on the task at hand), and the results are combined.

There are two major standards for parallel computing, OpenMP [28] and Message Passing Interface (MPI) [45]. OpenMP is a set of library functions, compiler directives, and environment variables for developing parallel applications utilizing threads and shared memory. MPI is a library containing functions that allow multiple processes to communicate with each other in a distributed memory environment.

OpenMP

OpenMP is a thread-based paradigm that provides for parallel computing utilizing shared memory. OpenMP implementations exist for C, C++, and Fortran. One major benefit of using OpenMP is that a program using OpenMP is portable to other platforms. The compiler directives for OpenMP inform a compiler which regions of code should be parallelized. The compiler directives also allow the programmer to define specific options for the parallel regions.

Threads provide parallelization in OpenMP. A thread is an active execution sequence of instructions within a process. A process is a running program that is allocated its own memory space by the operating system once the program is loaded into memory. Multiple threads may exist within a process. A thread can be thought of as a light-weight process. A thread has access to all the variables of the process from which it originated, but it is not assigned its own copy of those variables. Therefore, any thread

can modify data that can be accessed by another thread, or the parent process. An example of an application that people use quite often is a web browser. In a browser there may exist separate threads to display web pages, request and receive web pages, and accept user input. Without using threads, the web browser could become completely unresponsive at times. For example, if the browser were just a single process, and a user typed an invalid web address, the user might not be able to enter another web address until the browser had proceeded with all the steps of sending the invalid address across the network and receiving a response. Another example of multi-threading in a web browser is having multiple tabs. A user can view a web page in one tab while another web page loads in another.

In a program that includes OpenMP directives, a process runs serially until an OpenMP directive is encountered. Once a directive is encountered, new threads are created to compute in parallel. The threads are distributed amongst the processors on the machine running the parallel application, and each thread performs a segment of the overall computation simultaneously. The results of each thread can be easily combined because threads share the same memory for all variables created by the original process.

Load balancing is a problem to consider when dealing with parallel programming. Load balancing is assigning each processor as nearly as possible an equal amount of work to be done as the other processor(s) working on the parallelized problem. Since OpenMP uses threads rather than creating new processes that are copies of the original parent process, there can be issues with thread scheduling. OpenMP addresses thread scheduling by allowing the programmer to specify the type of scheduling to be used in

the program. Some of the main options for thread scheduling in a for loop that can be parallelized with OpenMP directives include:

static scheduling - all threads execute *n* iterations and then wait for all other threads to finish their *n* iterations

dynamic scheduling - *n* iterations of the remaining iterations are dynamically assigned to threads that are idle

guided scheduling - when a thread finishes executing its assigned iterations, it is assigned approximately the number of remaining iterations divided by the number of threads

OpenMP makes the programmer really consider which scheduling method is best for the code segment to be parallelized. On first glance, it might appear that dynamic scheduling is the most beneficial of the three mentioned; however, it is not always the best scheduling method. There is communicational overhead involved with assigning additional iterations to a thread. This overhead slows down the overall execution time and forces the programmer to run several test cases in order to choose an optimal scheduling strategy and number of threads.

There is another major issue when using threads rather than separate processes. Threads use the same variables and memory locations as the other threads and the process from which they originated. The issue regarding reading and writing to the same memory locations as other threads is called a race condition. A race condition occurs when two threads try to modify a value at the same time. A simple example of a race

condition is when two threads have to both read from and write to the same variable. Suppose both processes perform the following statement:

```
balance = balance – withdrawal[i];
```

Imagine the above code statement is to be used in banking and the value retrieved from the shared memory location is a bank balance. A race condition is evident when two people try to withdraw money at the same time. The race condition occurs during the time that one thread gets the bank balance, updates the bank balance, and then restores the updated bank balance into the memory location. If another thread accesses the bank balance while the other thread is currently calculating the updated balance but has yet to finish, then the thread will retrieve the old bank balance. Imagine two people at different ATMs that are accessing the same bank account at the same time. Suppose that the bank account contains five hundred dollars, and both users want to retrieve four hundred out of the five hundred dollars from the ATM. If both people access the bank account at the same time and there is no way to lock the bank account, then both people would be able to withdraw four hundred dollars from the account, totaling eight hundred dollars even though there is only five hundred dollars in the account. This is an example of a race condition that would greatly benefit the people withdrawing from the account and leave the people at the bank scratching their heads wondering what happened.

There is a way to prevent race conditions. All one thread has to do when accessing a shared variable is lock it so that no other thread has access to it until the thread that is locking is finished with it. Once finished with the shared variable, the thread unlocks it for other threads to access. The other threads that need to access the

shared variable are forced to wait until it has been unlocked by another thread that is currently using it. When working with locking shared variables between multiple threads, the programmer must be aware of the possibility of a deadlock. A deadlock is a situation when two threads are both waiting for a lock to be released that the other thread currently holds.

OpenMP has a couple of methods for dealing with shared data accessed by multiple threads. One way to avoid a race condition in OpenMP is by using the the "#pragma critical" directive to specify a critical section. A critical section is a code segment that uses and/or modifies the shared variable. By using the "#pragma critical" directive, one specifies that the critical section may be executed by only one thread at a time. By using this directive, one can prevent a race condition from ever occurring. The "#pragma critical" directive acts like a lock, and the shared variable is unlocked once the thread exits the critical section.

There is another way to avoid race conditions in OpenMP. The programmer can use the "#pragma local(*variables*)" or "#pragma threadprivate(*variables*)", where variables are a set of shared variables. The variables in the parameter list are then copied to the thread. This means that the variables are no longer shared by all of the threads. The thread that uses the above directive receives its own copy of each variable needed.

OpenMP makes it easy to combine partial solutions once each thread has performed its computation. The directive "#pragma reduction(*operator: variables*)" states to do the following:

1: Every thread is given its own copy of each variable in the variables list.

2: When the threads finish executing their code portion, a shared copy of each variable in the variables list receives the finished result that is stored in each thread's variable list, and then combines the copies based on the operator. For example, consider the code below:

```
int main(int argc, char *argv[]) {
      int total = 0; size = 10000;
      int counter, myArray[size];
      for(counter = 0; counter < size; counter++)
            myArray[counter] = counter;
      #pragma parallel for
      #pragma threadprivate(counter)
      #pragma reduction(+: total)
      for (counter = 0; counter < size; counter++)
            total = total + myArray[counter];
      cout << "The total is: " << total;
}
```

Figure 4.1 OpenMP array example

This code declares an array of type int containing ten thousand values. The array is then initialized. There are three pragma directives in the above code that are necessary for parallelizing the second for loop. The first pragma directive, "#pragma parallel for", notifies the compiler that the following for loop needs to be parallelized. The second pragma directive, "#pragma threadprivate(*counter*)", states that each thread will get its own copy of the variable "*counter*". The third pragma directive, "pragma reduction(+: *total*)", states for the variable "*total*" to contain the contents of all of the other threads' variable "*total*", and each thread's total is to be added to the shared memory variable total. Suppose there are 4 processors available to perform the parallelization of the above code and 4 threads are used. In theory, each processor receives a partition that is equal in size as the other processors. Since the for loop counts from 0 to 10000, each processor should receive a partition of size 2500. Behind the scenes, the compiler assigns each thread its own variable "*counter*" and initializes that variable to the value that the thread

is supposed to begin its work. For example, each thread would logically receive the following partitions:

Thread 1: for(counter = 0; counter < 2500; counter++)

Thread 2: for(counter = 2500; counter < 5000; counter++)

Thread 3: for(counter = 5000; counter < 7500; counter++)

Thread 4: for(counter = 7500; counter < 10000; counter++)

Once each thread iterates through its partition, its total is added with the other totals. By using 4 processors and 4 threads there should be 4x computational speedup, but this is rarely achieved due to communication overhead and thread scheduling. However, for this example, it should be very close to 4 times as fast as the serial code.

One downfall to using OpenMP is that some libraries are not thread-safe and should not be used in conjunction with OpenMP. In general a library which maintains internal state variables, such as file pointers, will not be thread-safe. An example of such a library is the GDAL (Geospatial Data Abstraction Library) [32]. GDAL is a translator library for raster geospatial data formats that is released under an X/MIT style Open Source license by the Open Source Geospatial Foundation.

MPI

MPI (Message Passing Interface) is a set of library routines for C/C++ and Fortran designed to add communication functions. MPI has the same benefit as OpenMP in that parallel applications written for one system can be easily used on another system supporting MPI. MPI, as its name suggests, is based on message passing. The main difference between MPI and OpenMP is that MPI uses multiple processes instead of

multiple threads. The essential difference is that each process gets its own copy of each variable used in the application, while threads do not unless it is specified by the programmer in OpenMP to give a thread a copy of a variable in order to make that variable usable by only that thread [7,24,35]. Because MPI is based on using multiple processes to parallelize some region of code, the processes must communicate with each other during several phases of execution. One process is usually considered the master process in an MPI application. This process is responsible for dividing the work to be performed and sending equal amounts of work to the other processes. This is done by sending messages to the other processes which define the work to be done. Once the other processes receive from the master process the data and the information on how to use the data, the other processes perform their computations in parallel, and then they send the results to the master process. The master process waits to receive the result from each of the other processes, gathers the results when the processes are finished with their operations, and performs some useful computation with the gathered results.

The execution model of an application written with MPI is very different than one written with OpenMP. An application written in OpenMP runs as a single process until an OpenMP parallel directive is encountered. At that time, it creates threads to handle the parallel region. When the parallel code is completed, the application runs as a single process once again. In MPI, all processes are created upon startup. The user has the option to specify on the command line how many processes need to be used for the application. Listed below is a basic "hello world" MPI program:

```
#include <iostream>
#include "mpi.h"
```

```
using namespace std;
int main(int argc, char *argv[]) {
      int rank, numtasks;
      // initialize MPI environment
      MPI_Init(&argc, &argv);
      // get current process id
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      // get total number of processes
      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
      cout << "Hello from process: " << rank
           << " of: " << size << endl;
      MPI_Finalize();
      return 0;
}
```

Figure 4.2 MPI hello world example

Parallel applications written in MPI are much more complex than OpenMP. First of all, the MPI environment must be initialized. This is done by using the "MPI_Init", "MPI_Comm_rank", and "MPI_Comm_size" functions. In the "MPI_Init" function call, *argc* and *argv* are the process's command line arguments. In the "MPI_Comm_rank" function call, the first parameter, *MPI_COMM_WORLD*, is a communicator, which is a collection of processes that can send messages to each other. In this instance *MPI_COMM_WORLD* is the communicator consisting of all the MPI processes. The second parameter in the function call is an integer, named *rank*, which receives a unique id for each process. Ranks in a communicator range from 0 to $p - 1$, where p is the size of the communicator. In the "MPI_Comm_size" function call, the first parameter is the same communicator used in "MPI_Comm_rank", and the second parameter is an integer, named *numtasks* in this case, that stores the number of processes that will execute the application. "MPI_Init" must be called before any other MPI functions are used. "MPI_Comm_size" and "MPI_Comm_rank" are not required, but they are generally used.

Simple MPI applications have processes send messages to each other using the MPI_Send and MPI_Recv functions. Each one of these functions has a long list of parameters that must be used for each MPI_Send and MPI_Recv function call:

```
Int MPI_Send(void*           message      /* in  */,
             Int             count        /* in  */
             MPI_Datatype    datatype     /* in  */,
             Int             dest         /* in  */,
             Int             tag          /* in  */,
             MPI_Comm        comm         /* in  */)

Int MPI_Recv(void*           message      /* out */,
             Int             count        /* in  */,
             MPI_Datatype    datatype     /* in  */,
             Int              source      /* in  */,
             Int              tag         /* in  */,
             MPI_Comm         comm        /* in  */,
             MPI_Status*      status      /* out */)
```

Figure 4.3 MPI send and receive functions

The content of the message is stored in a block of memory referenced by the parameter "message". The "count" and "datatype" parameters determine how much storage is needed for the message. For example, if you are sending an array of type int that contains 1000 locations, count is set to 1000, and datatype is set to the MPI datatype equivalent of type int, MPI_INT. The "dest" and "source" parameters specify which process is sending the data and which process is receiving the data. "tag" is an integer used to categorize the message, and "comm" is a communicator, which is typically MPI_COMM_WORLD. The "status" parameter in the receive function is used to return information on the data that was received. This information is in the form of a struct with three important attributes: source, tag, and error. The "status" parameter is used to determine which process sent the data, and if there was an error in communication, an error code is stored in the error attribute.

Programming a simple parallel application in MPI is rather tedious and complicated. In OpenMP, it was very easy to parallelize the for loop in the example. Only three OpenMP pragma directives were needed. However, in MPI the programmer has to initialize the MPI environment and have the processes send and receive multiple messages to parallelize the code. Listed below is the source code for an MPI application that performs the same task as the previous OpenMP code:

```
int main(int argc, char *argv[]) {
      int rank, numtasks, counter, source, tempTotal,
          myArray[size];
      int total = 0, size = 10000, tag = 0;
      MPI_Status status;

      for(counter = 0; counter < size; counter++)
            myArray[counter] = counter;

      // initialize MPI environment
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

      // have every process calculate their share
      for(counter = (size/numtasks)*rank;
          counter < ((size/numtasks)*rank + (size/numtasks));
          counter++)
          total = total + myArray[counter];

      if(rank == 0) {
            for(source = 1; source < numtasks; source++) {
                  MPI_Recv((int*)&tempTotal, 1, MPI_INT, source,
                        tag, MPI_COMM_WORLD, &status);
                  total = total + tempTotal;
            }
            cout << endl << "total: " << total << endl;
      } else { // other process
            MPI_Send((int*)&total, 1, MPI_INT, 0, tag,
                        MPI_COMM_WORLD);
      }
      MPI_Finalize();
      Return 0;
}
```

Figure 4.4 MPI array example

The above MPI code is much more complex than the OpenMP code. There are many more operations that have to be done in order for the code to be parallelized with

MPI. MPI needs six parameters in order to send a data message to a process using MPI_Send, while seven parameters are required to receive a message using MPI_Recv. Furthermore the differing loop limits are explicit with MPI while implicit with OpenMP. On the other hand, MPI simplifies some tasks by providing a large collection of functions to assist with common parallel computing paradigms including MPI_Bcast, MPI_Scatter, MPI_Reduce, MPI_Allreduce, MPI_Gather, and MPI_Allgather. MPI_Bcast and MPI_Scatter functions can be used to broadcast or scatter the data, such as an array, amongst the other processes. Once each process has performed its necessary operations on the data, it can combine the results in a similar fashion as the OpenMP example by using MPI_Reduce, MPI_Allreduce, MPI_Gather, or MPI_Allgather functions. Each one of these functions has parameters like MPI_Send and MPI_Recv functions and is just as complex to use.

MPI, similar to OpenMP, can suffer from load balancing issues. In the above example, it is easy to determine the partition to assign to each process for computation. However, in more complex applications, such as manipulating sparse arrays, determining an equal partition to assign to each process can be more difficult. In addition processors can execute at different rates if one is using a heterogeneous cluster. Finally, without having a dedicated parallel system, some of the processes of a parallel application might be competing with other, unrelated, processes for CPU time, thereby slowing down the parallel application.

OpenMP is generally more appropriate than MPI if a programmer wants to perform some simple parallel programming on a single, multi-core machine. In the two code examples listed above, the OpenMP version is more desirable in that it is simpler

and probably more efficient. It is more efficient in that it does not use as much memory and does not require copying data between processes. If a programmer has a complex, parallelizable algorithm that needs as many processors as possible to solve it as quickly as possible, then MPI is preferred over a network with distributed memory.

OpenMP and MPI are two effective ways to parallelize an application. The main difference between the two is that OpenMP uses threads and shared memory, and MPI uses multiple processes and message passing. OpenMP benefits from threads in that not as much memory is needed to run an application; however, extra steps must be taken to protect the data integrity of shared variables, and not all libraries are thread safe. OpenMP is very easy to use when parallelizing simple independent tasks, such as for loops, but it can be cumbersome when working with more complex algorithms. MPI is efficient in its message passing, but MPI is much more complicated to use than OpenMP. The rest of this dissertation discusses the parallel coordination framework Guppie. Guppie combines the convenience and shared memory functionality of OpenMP, while avoiding thread safety issues, and it is easier to use than MPI.

CHAPTER V

GUPPIE

The two main paradigms in parallel computing, shared memory and distributed memory multiprocessing, and two major standards for parallel computing, OpenMP and MPI, have been discussed. Both OpenMP and MPI are efficient but have some disadvantages. OpenMP makes it easy to parallelize loop structures, but thread safety can become an issue when OpenMP is combined with other libraries. MPI provides an efficient method for transferring data between multiple processes, but MPI routines are rather complex. Linda, a coordination model utilizing a shared tuple space, and Linda extensions have been discussed. The Guppie framework designed in this research addresses some of the shortcomings of these models. Guppie combines the message passing functionality seen in MPI, shared memory functionality seen in OpenMP, and a shared tuple space seen in Linda models. By comparing image processing applications written in MPI and Guppie, it is shown that Guppie is an effective way to do some forms of parallel computing while providing a convenient programming environment.

Guppie is a parallel coordination framework which features a master-worker relationship providing a tuple space like Linda and utilizing shared memory to avoid message passing for large data objects. The master delegates workers' access to the tuple space. The master also is responsible for starting worker processes. Workers communicate with the master synchronously through pipes.

Guppie creates processes, rather than threads, for parallelization. This allows each process to have its own copy of each variable, similar to how the above OpenMP

“#pragma” directives assign each thread its own copy of the variables specified within the directives’ parameter list. The difference is that when Guppie starts each process, each process has each variable in its initial state. When threads are created in OpenMP and the OpenMP “#pragma” directives are used, each thread receives a copy of the variables specified within the OpenMP directives’ parameter list, and the variables’ values are equal to what they currently are when the threads are created.

Suppose a Guppie application uses large arrays such as images. These images are placed into shared memory and then attached to worker processes using system calls. There are different system calls for Windows and UNIX, but both operating systems are accommodated in Guppie.

By using shared memory to attach large arrays of data to processes, Guppie avoids sending large messages back and forth. This is a major advantage over Linda's distributed shared memory system and MPI’s message passing. However, even though large data objects are placed in shared memory and attached to processes, coordination is still required to avoid race conditions. The Guppie functions set and get can be used to coordinate access to large data arrays. When a process completes its preparation of some data in an image, it uses a set or add function to create and place a tuple into the tuple space, indicating completion of its work. Any processes needing that image use a get function call to retrieve the completion tuple. Using set and get function calls in this fashion is equivalent to using semaphores. Guppie can effectively and efficiently send a small get message to coordinate the use of the data stored in shared memory. This is faster than Linda in two regards:

1: Linda models generally have a distributed shared memory tuple space. Processes in Linda have to communicate over a network in order to access the tuple space. This results in more communication cost than would be required to communicate within a single computer. However, Linda could be implemented on an SMP system and use internal message passing such as pipes or use shared memory and semaphores.

2: On top of the communication needed as described in part 1, Linda then has to send the large data array across the network. Guppie's shared memory resolves this issue by only having to send a small message to coordinate the use of shared memory. The large data object will not be sent in a tuple as it would be in the Linda environment.

Guppie's Main Functions

Guppie has four main functions, and three of the four functions closely resemble CLinda's function set: set(t), add(t), get(s), and take(s). Guppie uses a name and identifier as a key, rather than matching on any subset of values in an anti-tuple. This is simpler and avoids the need for a preprocessor while allowing a rich programming environment. A name and id are required for any tuple to be stored or retrieved from the tuple space. Guppie allows a user to store or retrieve up to eight values at a time by using templated function calls, which are described in a subsequent section. Listed below are Guppie's four main functions, along with an example for each:

set(t) closely resembles CLinda's out(t). set(t) causes tuple t to be stored in the tuple space. The operation "set("data", 5, 6.32)" sets a tuple into the tuple space consisting of name: "data", id: 5, and value: 6.32. Each tuple placed into the tuple space using the set(t) function should have a unique name and id.

add(t) causes tuple t to be stored in the tuple space in the form of a first in, first out queue. add(t) allows the user to add a tuple that may match the name, id, and value type of a tuple already stored in tuple space. The operation "add("data", 5, 6.32)" adds a tuple into the tuple space consisting of name: "data", id: 5, and value: 6.32. This creates a tuple and stores it in the tuple space much like the set(t) operation. However, by using the add(t) function, the user can place multiple tuples with the same name and id into the tuple space, and the tuples are stored in a FIFO queue. By contrast in the Linda model, if there are multiple tuples having the same name and id, the user had no control over which tuple is returned from the tuple space. By using add(t), the user can structure a queue of related tuples within the tuple space.

get(s) closely resembles CLinda's rd(t). get(s) causes some tuple t that matches anti-tuple s to be withdrawn from the tuple space. The first two parameters of a get function call are the tuple name and id which are matched against stored tuples' names and ids. This is a simpler matching process than in CLinda. The remaining parameters of a get function call are variables which will receive data from the tuple when a matching tuple is found. For example, suppose the tuple ("data", 5, 6.32) exists within the tuple space. The operation get("data", 5, dataValue) retrieves the matched tuple in the tuple space and stores the value, 6.32, into the variable *dataValue* sent as a parameter in the get(s) function call. The matched tuple t remains in the tuple space for other processes to retrieve.

take(s) closely resembles CLinda's in(s). take(s) causes some tuple t that matches anti-tuple s to be withdrawn from the tuple space. This function behaves the same way as the get(s) function, but the matched tuple t does not remain in the tuple space. Once a

process retrieves a tuple using the take(s) function call, no other process can retrieve that tuple.

Guppie Master Process Example

Guppie uses a master-worker relationship. The master has direct access to the tuple space, and workers may access the tuple space by sending requests to the master. The master handles each request, searches the tuple space for the requested value, and then if found, retrieves the value and sends it to the process that requested it. If the value is not found, the master has the worker wait until the tuple is in the tuple space, and the master proceeds to service other requests.

The programmer is responsible for programming the master process code and the worker process code. Below is an example of a master process code that starts ten processes and sets a tuple with three values into the tuple space in order for the workers to use:

```
#include <stdlib.h>
#include "master.h"

master *m;
int main(int argc, char **argv)
{
      m = new master();
      for(int i = 0; i < 10; i++)
      m->start_process("worker");
      m->set("data", 0, 1, 2.5, "output");
      m->run();
      return 0;
}
```

Figure 5.1 master starting worker processes in Guppie

In this main program a master object is created. For simplicity[3] this object is used to start the 10 worker processes by calling the member function "start_process". Then the master uses the "set" member function to set a tuple named "data" with id 0 having 3 data items: 1, 2.5 and the string "output". Finally, the master calls the run member function to allow the master process to handle tuple requests from the workers.

Guppie Complete Example

The following example demonstrates how to program the same parallel application that was previously demonstrated using OpenMP and MPI. For this example, the master process starts the worker processes and issues tasks for each worker. The master process file is as follows:

```
#include <stdlib.h>
#include "master.h"

class newMaster:public master {
     public:
     void setTasks();
     void startWorkers();
};
newMaster *m;
void newMaster::setTasks() {
      for(int i = 0; i < 4; i++)
            add("task", 0, i);
}
void newMaster::startWorkers() {
      for(int i =  0; i < 4; i++)
            start_process("worker");
}
int main(int argc, char **argv) {
      m = newMaster();
      m->startWorkers();
      m->setTasks();
```

[3] In general it might be advantageous to define a class derived from the master class and place the operations performed by the master into member functions of the derived class.

```
      m->run();
      return 0;
}
```

Figure 5.2 Guppie master process

To remain consistent with the OpenMP and MPI applications, the master process will start four worker processes. By starting four worker processes, the Guppie version of the parallel code is consistent in that there are four worker processes; however, there are five total processes running. The five processes consist of the four worker processes and the one master process. This is a slight drawback to Guppie when working with very simple parallel applications. One extra process is needed for the master process.

The worker process code is in a similar format to the master process code. It contains a class that has a function named "run", where the user writes the worker process code. Listed below is the worker process code:

```
#include <stdlib.h>
#include "worker.h"
#include <unistd.h>

class workerProc:public worker {
    public:
    void run();
};
void workerProc::run() {
    int size = 10000, numtasks = 4, total = 0, finalTotal = 0;
    int pid, counter, tempTotal, myArray[size];

    take("task", 0, pid);

    // initialize array
    for(counter = 0; counter < size; counter++)
        myArray[counter] = counter;

    // have every process calculate their share
    for(counter = (size/numtasks)*pid;
        counter < ((size/numtasks)*pid + (size/numtasks)); counter++)
        total = total + myArray[counter];

    // have every process store its total in the tuple space
    cout << endl << "Process: " << pid << " individual total: "
        << total << endl;
```

```
    set("total", pid, total);

    // have only one process calculate the result
    if(pid == 0) {
        for(counter = 0; counter < numtasks; counter++) {
              take("total", counter, tempTotal);
              finalTotal = finalTotal + tempTotal;
        }
        cout << endl << "Process: " << pid << " final computed total: "
             << finalTotal << endl;
    }
}
workerProc *a;
int main(int argc, char **argv) {
    a = new workerProc();
    a->connect ( argc, argv );
    a->run();
    return 0;
}
```

Figure 5.3 Guppie worker process

The above code creates a class called "workerProc" and allows this class to have access to all of the main worker functions. The class "workerProc" has one class function, "run", which is where the user writes the parallel application. In the main function, the class is created, and the "connect" function is called. The "connect" function is a part of the main worker file, worker.cpp, and is responsible for connecting the worker process with the master process so that communication between this process and the master is possible. After the connection has been established, the "run" function is called, which is where the actual parallel application code is written.

This parallel application solves the exact same problem as seen in the OpenMP and MPI examples. The master process creates four worker processes, and each worker process executes the code within the "run" function. Each worker process initializes its own copy of the array, "myArray". For demonstration purposes, the array has been initialized to contain all 10000 elements, even though each process is only working with 2500 elements. The array was initialized in this format for demonstration purposes, to

stay consistent with the OpenMP and MPI code, and to better show afterwards how each process only works with a segment of the array for the parallel computation of the problem.

Once each process initializes the array, it performs its segment of code based on the process id. Because four worker processes are used in this application, the first worker process computes through locations 0 to 2499 of the array, the second worker process computes through locations 2500 to 4999 of the array, the third worker process computes through locations 5000 to 7499 of the array, and the fourth worker process computes through locations 7500 to 9999 of the array. Once each process has calculated its total, the total is stored in the tuple space using Guppie's set(t) operation. The following tuples are stored in the tuple space using Guppie's set(t) function:

Worker process 0: set("total", pid, total), where pid = 0, total = 3123750
Worker process 1: set("total", pid, total), where pid = 1, total = 9373750
Worker process 2: set("total", pid, total), where pid = 2, total = 15623750
Worker process 3: set("total", pid, total), where pid = 3, total = 21873750

The programmer could choose to use the add function rather than the set function. The add function is intended to store multiple tuples in the tuple space in a queuing format, which should be based on having the same name and id parameters and a different value parameter. The following tuples could be stored in the tuple space using the add(t) function rather than the set(t) function:

Worker process 0: add("total", 0, total), where total = 3123750
Worker process 1: add("total", 0, total), where total = 9373750
Worker process 2: add("total", 0, total), where total = 15623750

Worker process 3: add(“total”, 0, total), where total = 21873750

After each process sets a tuple containing the finished total into the tuple space, it is necessary for one worker process to collect the tuples and calculate the overall total of the four values stored in the tuple space. This is performed in a similar manner to MPI. All that is needed is an if statement to be true for only one process. That process, process 0 in this case, retrieves the tuples from the tuple space and calculates the overall total. This is performed by using Guppie’s get(s) or take(s) function. For this application, there is no reason to leave the tuples in the tuple space, so take(s) is a logical choice. Worker process 0 calls the take function four times because there are four worker processes that each stored an individual tuple in the tuple space. The following for loop is constructed to call the necessary take(s) functions:

```
for(counter = 0; counter < numtasks; counter++) {
        take("total", counter, tempTotal);
        finalTotal = finalTotal + tempTotal;
}
```

Each call to take fetches a tuple matching “total” and *counter* and stores a value into *tempTotal*. After each take function executes, *tempTotal* is added to *finalTotal*. Process 0 withdraws the tuples and their values in the following order:

take(“total”, counter, tempTotal), where counter = 0, tempTotal contains 3123750 after the take operation

take(“total”, counter, tempTotal), where counter = 1, tempTotal contains 9373750 after the take operation

take(“total”, counter, tempTotal), where counter = 2, tempTotal contains 15623750 after the take operation

take(“total”, counter, tempTotal), where counter = 3, tempTotal contains 21873750 after the take operation

After all four take functions are performed and the final total has been calculated, the variable “finalTotal” contains the value 49995000.

Guppie’s four main operations, set, add, get, and take are similar to MPI’s functions in that they can be used for communication. However, the Guppie functions are much easier to use. The programmer does not have to specify explicitly the variable’s type, how many bytes need to be used, what process is to receive the data, what process is to send the data, a tag to keep messages from being sent and received by processes not intending on sending or receiving the data, a communicator, or a variable to maintain the message’s status. The master, worker, and buffer classes handle all of those steps for the programmer.

Guppie makes it easy to implement an application that uses the “bag of tasks” programming paradigm. The bag of tasks paradigm is a parallel computing method in which tasks are placed in a bag shared by worker processes. In Guppie, the tuple represents the bag, and the tuples represent the tasks. Each worker process repeatedly takes a task tuple from the tuple space, executes it, and possibly generates new task tuples to be placed into the tuple space to be retrieved by other worker processes. Matrix multiplication is an example using the bag of tasks paradigm. For example, suppose three matrices of equal size are to be multiplied. Matrix X is multiplied by matrix Y, and matrix Z is multiplied by the result of X * Y. In order to compute the result for row R and column C, a worker process only needs to know the R row values from the first matrix and the C column values from the second matrix. In Guppie, once a value has been calculated, that value can be stored in the tuple space with a logical name and id. For example, the calculated value for the first row of X times the first row of Y could be

added to the tuple space with “add(“XY”, 0, value)”, where the XY is the name representing the matrix X * Y, 0 is the id representing the location in the matrix, and *value* is the calculated value of the matrix multiplication. Suppose the matrices are 5 x 5 matrices. Then, the calculated value for the second row of X times the third row of Y would be added to the tuple space with “add(“XY”, 7, value)”, where the id of 7 is used to represent the location in the matrix. The first location in the matrix (first row, first column) is represented with a 0, the second location (first row, second column) is represented with a 1, continuing to the final location in the matrix (fifth row, fifth column) is represented with a 24. By storing these task tuples, the values needed for the matrix multiplication of the results of X * Y multipled by Z can be retrieved by using a get operation that matches the name and id of the required location of the matrix. In MPI, this type of application is more difficult to create. The master would most likely send the necessary values in matrices X and Y to the worker processes in order for the worker processes to perform their computations. Once the worker processes have computed the results, they would send the results to the master process. The master process would most likely have to store the entire result into a result matrix before sending any further values to worker processes in order to calculate X * Y multiplied by Z. In Guppie, there is no need to keep track of any results matrices while performing matrix calculations.

Guppie’s capability to create applications using the bag of tasks paradigm allows for dynamic load balancing. Unlike in MPI, there is no need for the user to have to assign particular data partitions to particular workers. In the event that the data partitions are not even for each worker process, meaning that some workers have to perform more

work than others, there is no extra effort required from the user to handle this. In Guppie, each worker process retrieves tuples from the tuple space until all work is complete. However, in MPI, the user has to handle such a situation explicitly. Chapter VI compares an image processing application written in both MPI and Guppie, and it demonstrates Guppie's ability to provide dynamic load balancing.

It is easy to see how Guppie's function set is much easier to use than MPI's function set. One does not need to specify which process sends or receives the data. Guppie's use of a tuple space that is accessed by worker process by sending requests to a master process keeps the data organized and makes parallel computing easy for the programmer. Guppie is currently able to work with integer, double, string, and vector data, but the library can easily be extended to contain other data types. The following two sections discuss Guppie's templated code and buffer functionality. The section following those two sections discusses Guppie's main attraction shared memory.

Templated Functionality

Guppie features templated functions to handle both basic and derived data types in its main operations: get, take, add, and set. The basic C++ data types (integers, longs, floats, doubles, and strings) and vectors of those types are allowed as components of tuples. In addition, structs without internal pointers may also be included in tuples. Guppie also allows for up to ten parameters to be passed at one time. An example of the get templated operation can be seen in the code below:

Worker Header File Function Declarations

```
template <class T>
        void get(string name, int id, T &t);
```

```
template <class T1, class T2>
        void get(string name, int id, T1 &t1, T2 &t2);

template <class T1, class T2, class T3>
        void get(string name, int id, T1 &t1, T2 &t2, T3 &t3);

/* Guppie allows up to ten templated parameters following
   the above format */

// Function to get and leave an integer based on name and id
int handle_get(string name, int id, int &value);

// Function to get and leave a double based on name and id
int handle_get(string name, int id, double &value);

template <class T>
    void worker::get(string name, int id, T &t) {
      handle_get(name, id, t);
    }

template <class T1, class T2>
    void worker::get(string name, int id, T1 &t1, T2 &t2) {
      handle_get(name, id, t1);
      handle_get(name, id, t2);
    }

template <class T1, class T2, class T3>
    void worker::get(string name, int id, T1 &t1, T2 &t2, T3 &t3) {
      handle_get(name, id, t1);
      handle_get(name, id, t2);
      handle_get(name, id, t3);
    }
```

Worker CPP File Function Code

```
int worker::handle_get(string name, int id, int &value) {
    // process the get request for an integer value
}

int worker::handle_get(string name, int id, double &value) {
    // process the get request for an integer value
}
```

Figure 5.4 Guppie templates

The above code demonstrates Guppie's use of templates. The templated get functions of different length allow essentially arbitrary tuple contents. For example, a user can use the operation "get("data", 0, *int1*, *int2*, *double1*)" where name = "data", id = 0, *int1* and *int2* are integer variables, and *double1* is a double variable. Guppie's templates allow for each parameter in each get function to be treated individually. The

single get template function with 3 template types can handle any combination of 3 basic types or vectors.

Guppie's templated functions handle almost any data type used in its main operations: get, take, add, and set. Linda's preprocessor has to maintain type information for each one of Linda's function calls. Unlike in Linda implementations, with Guppie there is no need to write a separate preprocessing application to track the types of variables during certain contexts of a program, to help facilitate the run-time search, or swap operators (such as the "?" in Linda) with ones that are logically equivalent.

Specialized Buffer

Guppie features a buffer to facilitate the sending and receiving of data. One advantage to using a buffer is having the ability to transfer multiple data items at once during a master-worker communication. This can be achieved by packing multiple data items into the buffer object and then sending the buffer as one unit, rather than sending the data items individually. Another advantage to using a buffer is that it can contain any combination of basic data types, as well as user-defined data types.

The buffer is implemented as a class in C++. This provides for easy use and functionality. The data for the buffer is an unsigned character array. Each process has its own buffer. The buffer class provides member functions to place and remove data to and from the buffer. A code example is shown below demonstrating how a buffer is declared and how data is packed and unpacked. Above each segment of code is a comment describing it.

Buffer Code Example

Inside of a worker process:

```
/* initialize value and use as a place holder within the buffer for
   integer data to be retrieved from the master */

int value = 0;

/* This initializes buffer class variables and reserves four bytes at
   the beginning of the buffer to store an integer value intended to be
   the size of the buffer.  This is the first value removed from the
   buffer, so the unpacking process knows exactly how many bytes to
   unpack */

myBuffer.initializeMessage();

/* for this code example, t will function as the id parameter and is
   given an arbitrary value */

t = 1;

/* this is calling the get function, stating that we want whatever
   value is stored in the tuple space that matches the name parameter
   "data" and its id parameter, 1.  The get function is found in the
   worker class */

get("data", t, value);
```

Inside the handle_get function found in the worker class:

```
int worker::handle_get(string name, int id, int &value) {
      // name = "data", id = 1, value = 0

      /* variables needed for the request to be sent to the master
         are declared */

      /* signifies the type of request, such as "get_integer",
      "get_double",
         "exit", "shutdown", "set_string", etc. */
      unsigned char req;

      // stores internal process id of a worker process
      unsigned char pid;

      // value associated with "get_integer"
      req = '5';

      /* myid is a worker class variable which is an internal id
         starting with 0 for each worker.  the master has a myid value
         of -1 */

      pid = myid;

      /* Four bytes are stored in the buffer as an integer.  In some
         systems, an integer may be represented as eight bytes.  There
```

```
            is a check to determine the correct number of bytes to be
            stored for proper data representation.  In this case, four
            bytes are stored.  The variable value is used as a placeholder
            to store the integer that is being retrieved from the master
            process */

        myBuffer.addValue(value);

        // stores entire request into the buffer
        myBuffer.packRequest(id, req, pid, name);

        // send the buffer from the worker to the master for processing
}
```

Figure 5.5 Guppie buffer

Buffer Contents

The following is a view of the buffer after the data and request has been stored for the get operation seen in the above code example: get("data", 1, value), where value = 0. The buffer contents are in hexadecimal values.

Index	0	1	2	3	4	5	6	7	8
Value	17	00	00	00	31	00	00	00	00

Index	9	10	11	12	13	14	15	16	17
Value	01	00	00	00	05	00	44	61	74

Index	18	19	20	21	22	23	24	25	26
Value	61	00	00	00	00	00	00	00	00

Figure 5.6 Guppie Buffer Contents

Bytes 0 – 3: Integer to represent the overall length of the buffer, minus the actual four bytes to represent the overall length of the buffer. The buffer length has a value of 17 in hexadecimal, which is 23 in decimal. This indicates that 23 bytes need to be read by the unpacking process.

Byte 4: Unsigned character descriptor to indicate the data type, which is an integer in this case.

Bytes 5 – 8: Stores reserved bytes for the integer variable "value".

Bytes 9 – 12: Stores the integer id value, which is a parameter in the get function. In this example, the id has a value of 1.

Byte 13: Unsigned character for the variable "req". This signifies what type of request is being sent by the worker to the master.

Byte 14: Unsigned character for the variable "pid". This variable is the internal process id of the worker process that sent the buffer.

Bytes 15 – 26: Represent the string name parameter, which has the value of "data" in this example. The name field has been arbitrarily given 12 characters to store the name. It might make more sense to only store the number of bytes needed to construct the name value. However, by storing the name in that format, an additional integer, or short, value must also be stored that indicates the number of bytes composing the name value that needs to be extracted by the unpacking process.

The buffer object may contain data for basic and user-defined types. For basic types such as integers or doubles, a character descriptor is used to notify the unpacking

object, whether it be the master or a worker, what kind of object it is about to unpack. Since the Guppie operations are templated, the descriptor is determined without the user explicitly having to state the value of the descriptor. Based on the descriptor, the unpacking object will know how many bytes to unpack, so it is able to unpack the entire object without overstepping its bounds. In the case of an object such as a string or a vector, the descriptor will notify the unpacking object that a length must first be unpacked in order to determine how many bytes to extract from the buffer.

User-defined types, such as structs and classes, can also be placed into the buffer provided that there are no internal pointers in the classes or structs. In such cases a class object is copied as a stream of bytes into the buffer without any consideration of the data types of the class data members.

As seen in the buffer code example, a request must also be placed into the buffer whenever data is sent from a worker to the master process. In the example, a get operation is called and a request is placed into the buffer. A request is a data structure that contains important information that the master needs to know. This information includes an integer id, which is a numeric identifier for a data item, and an unsigned character variable which signifies the type of request that needs to be processed, such as a "get_integer" or "take_vector_int". Also included in the request is an unsigned character variable that stores the internal process id of a worker process and a character array that stores the name of a data item.

Guppie's Shared Memory

Guppie's main intent is to be run on a computer with multiple processors. Guppie uses true shared memory, not memory distributed across multiple machines. By using shared memory, Guppie is very efficient when working with parallel programs containing large arrays, such as images. The large data array can be placed into shared memory by one process in order for the other processes to access it. This drastically cuts down on the communication costs usually needed to transfer data between processes, as would be necessary with MPI. Since Guppie uses multiple processes and not threads, Guppie is able to work with thread unsafe libraries, unlike OpenMP.

Guppie is implemented in the Qt environment. Qt is a cross-platform, graphical, application development toolkit that enables a programmer to compile and run applications on Windows, Mac OS X, Linux, and different brands of Unix. A large part of Qt is devoted to providing a platform-neutral interface. Guppie utilizes Qt's shared memory library to send and receive data to and from a shared memory segment. The shared memory component in Qt provides a portable, platform-independent class API to the shared memory mechanism of the underlying operating system.

Below is a list of Guppie's shared memory functions that are a part of the main worker.cpp file:

void create_shm(string key, int id, int size) – creates a shared memory region of *size* bytes with identifiers, *key* and *id*

void attach_shm(string key, int id) – attaches a process to a shared memory region referenced by *key* and *id*

void detach_shm(string key, int id) – detaches a process from a shared memory region referenced by *key* and *id*

void* location_shm(string key, int id) – returns the starting location of a shared memory region referenced by *key* and *id*

int size_shm(string key, int id) – returns the size in bytes of a shared memory region referenced by *key* and *id*

void lock_shm(string key, int id) – locks the shared memory region referenced by *key* and *id* so that no other processes may access it while the process that called "lockSM" has locked it

void unlock_shm(string key, int id) – unlocks the shared memory region referenced by *key* and *id* so that other processes may now access it.

CHAPTER VI

GUPPIE'S SHARED MEMORY VS MPI'S MESSAGE PASSING

A test program was written to compare Guppie's shared memory efficiency with MPI's message passing scheme. The application reads 64 images, averages a 21 x 21 block of neighboring pixels in each image 10 times, and outputs the images to a larger mosaic image. In other words, the program severely blurs each image, and each image is placed into an 8 x 8 mosaic image that contains each blurred image. Each single image is 600 rows x 800 columns. The mosaic image, containing 64 of the individual blurred images, is 4800 rows x 6400 columns. In order to keep the code comparisons as simple as possible, the same image is used 64 separate times. Extra programming would be necessary in both MPI and Guppie to account for 64 different images, but to compare the efficiency between MPI and Guppie, using the same image 64 different times is sufficient.

Both the MPI program and the Guppie program were written to have each process perform a distinct task. The MPI program is composed of a master process and multiple worker processes. The master process has many responsibilities. The master process reads and stores into an array the image that is to be sent to the worker processes. Once the image has been read, the master process enters a loop to process worker requests until all images have been used. The master waits for requests from the worker processes. There are two types of worker requests in the MPI program. The first type of worker request is a request for the master to send a worker an image. The second type of worker request is to notify the master that the worker has finished averaging its image, and the

worker needs to send the averaged image to the master. Once an image request has been received, the master sends the image to the worker that made the request. This process is repeated for all worker processes. Once all worker processes have been sent an image, the master then waits to receive other requests from the worker processes. Once a worker process has finished averaging its image, it sends a request to the master to notify the master to get ready to receive the averaged image. Once the request has been processed by the master, the master receives the averaged image. The master is responsible for storing each worker's averaged image into the mosaic. Even though only one image is used 64 different times in this program, the program treats it as though it is a different image that needs to be placed in the mosaic in its proper location. For every averaged image received, the master must store that image into the mosaic before another averaged image is received by another worker process. This results in idle time for all worker processes in two regards. The first is that each worker process must wait to send its averaged image while the master is servicing a different worker's averaged image. The second is that each process must wait to receive another image while the master is servicing a different worker's averaged image. Once all images have been completed by the worker processes and stored in the mosaic by the master process, the master process stores the mosaic into an output file. The complete MPI code can be seen in Appendix B.

In order to parallelize the program in MPI, there are several communications needed to send multiple images to multiple processes. This communication cost is avoided using Guppie's shared memory functionality. The Guppie program, like the MPI program, is composed of a master process and multiple worker processes. The master process is responsible for creating the shared memory region for the individual image to

be used 64 times and creating the shared memory region for the mosaic image. The master is also responsible for reading the image data and storing it into the shared memory region. Below is the master process code, inside the "setTasks" function discussed previously:

```
void newMaster::setTasks() {
        // LOCAL VARIABLES ARE DECLARED HERE

        // add image counters
        for(i = 0; i < 64; i++)
           add("Images", 0, i);

        // add tuples so workers know image processing is complete
        for(i = 0; i < numWorkers; i++)
           add("Images", 0, -1);

        /* create shared memory segment for a single image for workers
           to read */
        size = rowSize * colSize * 3; // size is type int
                                      // rowSize = 600, type int
                                      // colSize = 800, type int
        create_shm("image", 0, size);

        // create shared memory segment for the mosaic
        size = mosaicRow * mosaicCol * 3; // mosaicRow = 1200, type int
                                          // mosaicCol = 3200, type int
        create_shm("mosaic", 1, size);

        fd = open("shapes.img", O_RDONLY);

        // imageData is an unsigned char array to store the image
        read(fd, imageData, rowSize*colSize*3);

        cd = close(fd);

        // store pointer to shared memory segment
        ucharPtr = (unsigned char*)location_shm("image", 0);

        // STORE IMAGEDATA ARRAY INTO SHARED MEMORY
}

int main(int argc, char **argv) {
        m = newMaster();
        m->startWorkers();
        m->setTasks();
        m->run();
        m->detach_shm("image", 0);
```

```
        m->detach_shm("mosaic", 1);
        return 0;
}
```

Figure 6.1 Guppie master creating and using shared memory

Some of the code was left out of the above figure in order to demonstrate only the Guppie code that is needed to solve the problem. The complete code can be seen in Appendix C. In the "setTasks" function within the master process code, after the variables have been declared, an image counter is initialized to 0 and is set into the tuple space by the master process. This tuple is used by the worker processes to coordinate what image each process is using. This is demonstrated in the worker process code. After the image counter tuple has been set, the master gets ready to create a shared memory segment for the image to be used by the worker processes. The master must first calculate the image size. Then, the master creates the shared memory segment using the "create_shm" function, assigning the shared memory segment the key "image", id = 0, and the previously calculated image size. The same steps are repeated to create a shared memory segment for the mosaic. After the shared memory segments have been created, the master reads the image named "shapes.img" and stores it into an unsigned character array named "imageData". Then, the "location_shm" function is used to return a pointer to the starting location of the shared memory segment created for the image. Now that a pointer exists to the shared memory region, the image array "imageData" is stored. Afterwards, the four worker processes are created with the "create" function, and the master process detaches from the shared memory segments using the "detach_shm" function. The worker process code is as follows:

```
void workerProc::run()
{
    // LOCAL VARIABLES ARE DECLARED HERE

    // attach to single image segment
    attach_shm("image", 0);
    // attach to mosaic segment
    attach_shm("mosaic", 1);

    // imageSize and mosaicSize are type int
    imageSize = size_shm("image", 0); // store size of image in bytes
    mosaicSize = size_shm("mosaic", 1); // store size of mosaic

    // store pointer to shared memory segment
    ucharPtr = (unsigned char*)location_shm("image", 0);

    /* STORE IMAGE IN SHARED MEMORY SEGMENT INTO UNSIGNED CHAR
       IMAGEDATA ARRAY */

    take("Images", 0, currentImage);

    ucharPtr = (unsigned char*)location_shm("mosaic", 1);
    while(currentImage != -1) {
         // AVERAGE THE PIXELS IN THE ARRAY 40 TIMES

         /* STORE THE AVERAGED IMAGE INTO THE MOSAIC SHARED MEMORY
         SEGMENT POINTED TO BY "ucharPtr". */

         take("Images", 0, currentImage);
    } // end of while(currentImage != -1)

    /* NOW HAVE ONE WORKER PROCESS STORE THE CONTENTS IN THE MOSAIC
    SHARED MEMORY SEGMENT INTO THE OUTPUT FILE */

    // detach from shared memory segments
    detach_shm("image", 0);
    detach_shm("mosaic", 1);
}
```

Figure 6.2 Guppie worker attaching and using shared memory

After the local variables are declared within the worker process, the worker process uses two "attach_shm" function calls. The first function call, attach_shm("image", 0), attaches to the shared memory segment containing the single image. The second function call, attach_shm("mosaic", 1), attaches to the shared memory segment that will contain the mosaic after each worker process averages its image and places it in its proper location in the "mosaic" shared memory segment. Next, the worker

process needs to determine the overall size of each one of the shared memory segments. This is achieved by using the "size_shm" function call. The first function call, size_shm("image", 0), returns an integer value of the size in bytes of the "image" shared memory segment and stores the value in a local integer variable named "*imageSize*". The second function call, size_shm("mosaic", 1), stores the size of the "mosaic" shared memory segment into a local integer variable named "*mosaicSize*". Then, a pointer to the "image" shared memory segment is retrieved by calling location_shm("image", 0), and is stored in an unsigned character pointer named "*ucharPtr*". Once the pointer to the "image" shared memory segment has been stored in "*ucharPtr*", the image can be retrieved from the shared memory segment and stored in the unsigned character "*imageData*" array. Next, it is necessary for the user to use Guppie's coordination function, take("Images", 0, *currentImage*). This retrieves a tuple placed into the tuple space by the master process, which is either an image number 0 – 64 or a -1 indicating all images have been processed. This "take" operation is necessary to keep an accurate counter that determines how many images have been completed thus far and where to place the finished image within the mosaic. For example, the first worker process to call take("Images", 0, *currentImage*) retrieves the tuple containing the value of 0, which was placed into the tuple space by the master process. The value of 0 is stored in the integer variable "*currentImage*". Since the take operation is used rather than the get operation, the tuple is removed from the tuple space. The next worker process that calls the take function will receive a tuple containing the value of 1, which is the next image number. The master adds 64 tuples into the tuple space, one for each image. The master also adds a tuple containing the value -1 for each worker process. These tuples indicate to the worker processes that all

images have been completed. This is how Guppie is used as a coordination language along with utilizing shared memory for efficient and easy parallel processing. After the "*currentImage*" variable has been retrieved from the tuple space, a pointer to the "mosaic" shared memory region is stored in "*ucharPtr*". Now, the worker process enters a while loop to process images as long as there still exists an image to be processed. Inside the while loop, the image is averaged and stored into the proper location in the shared memory segment (reference Appendix C to see how this is achieved). At the end of the while loop, another tuple is taken from the tuple space, either indicating the next image to be processed or all images have been processed. After all images have been averaged and stored into the "mosaic" shared memory segment, all workers except for one detach from both the "image" and "mosaic" shared memory segments. The one remaining worker is responsible for storing the contents of the mosaic into an output file. After the mosaic has been stored into an output file, the last remaining process detaches from the shared memory regions.

Execution Times and Analysis

Both the MPI and the Guppie applications were tested on a sixteen core Dell blade server. A serialized application was created to be compared to the MPI and Guppie applications. The serialized program ran in 11 minutes 44 seconds. Both the MPI and Guppie applications were tested with the following number of worker processes and cores:

2 – 4 cores 4 workers

3 – 8 cores 8 workers

9 – 16 cores 16 workers

Listed below are the execution time, speedup, and efficiency for each test run of the MPI and Guppie applications:

MPI

cores	2	3	4	5	6	7	8
time	8m50.57s	8m49.73s	5m53.29s	4m24.46s	4m25.62s	4m1.67s	2m52.46
speedup	1.33	1.33	1.99	2.67	2.65	2.91	4.09
efficiency	0.66	0.44	0.50	0.53	0.44	0.42	0.51

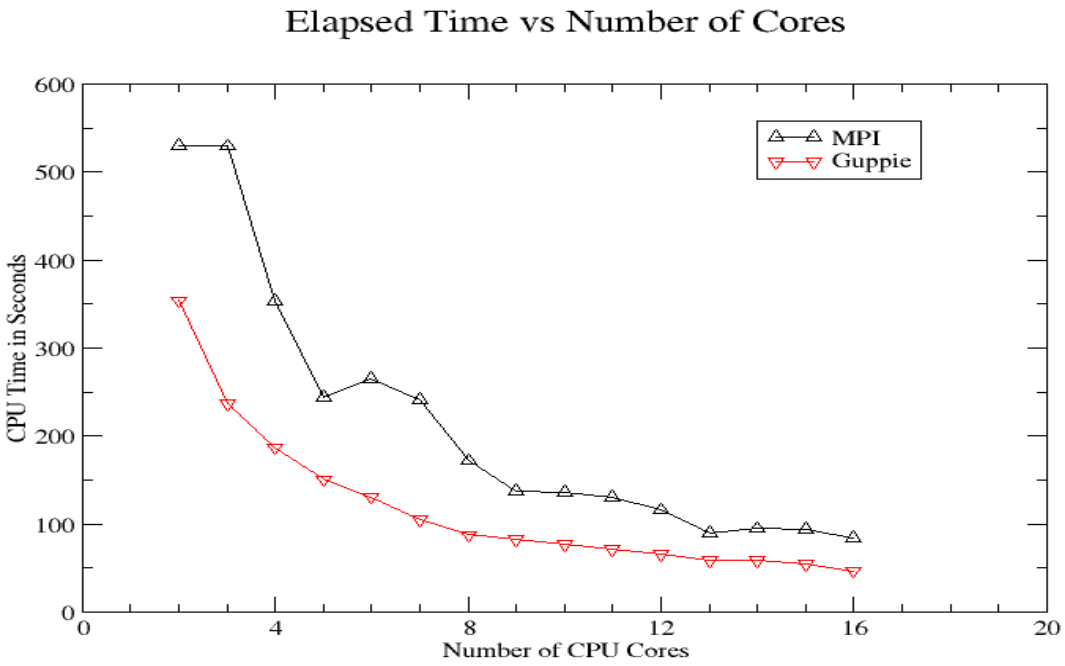
9	10	11	12	13	14	15	16
2m18.18s	2m16.46s	2m11.26s	1m56.56s	1m30.73s	1m35.16s	1m34.04s	1m24.91s
5.18	5.18	5.37	6.02	7.74	7.49	7.49	8.28
0.57	0.52	0.49	0.50	0.60	0.53	0.50	0.52

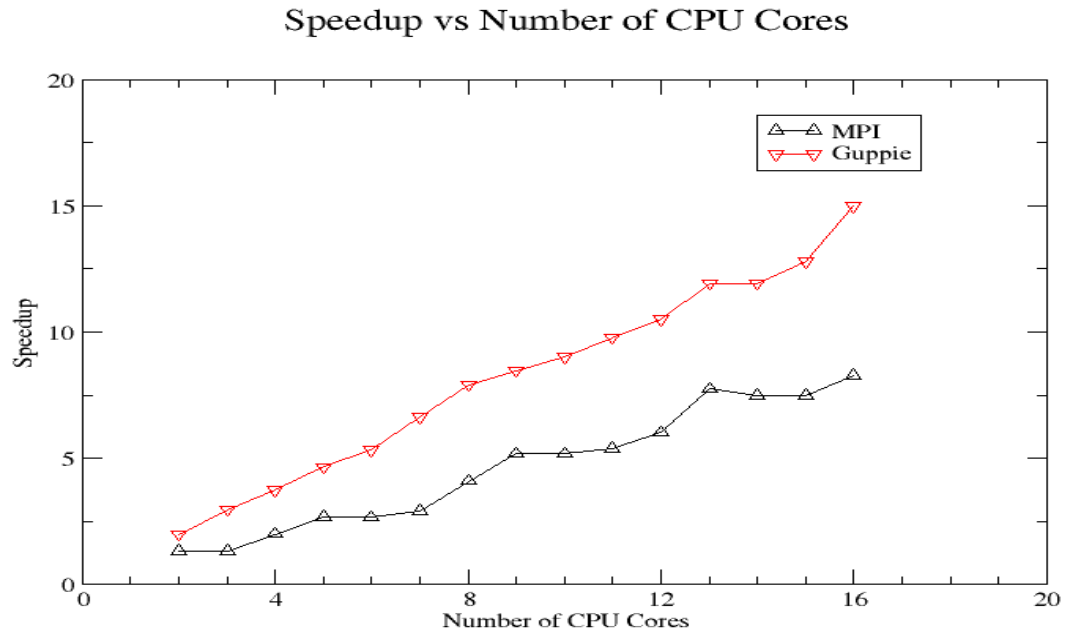
Guppie

cores	2	3	4	5	6	7	8
time	5m54.03s	3m57.15s	3m7.86s	2m31.40s	2m11.57s	1m45.59s	1m28.88s
speedup	1.99	2.97	3.74	4.66	5.33	6.64	7.91
efficiency	0.99	0.99	0.94	0.93	0.89	0.95	0.99

9	10	11	12	13	14	15	16
1m23.339s	1m17.56s	1m12.01s	1m6.63s	0m59.38s	0m59.72s	0m55.72s	0m47.08s
8.47	9.03	9.78	10.51	11.93	11.93	12.8	14.98

0.94	0.90	0.89	0.88	0.92	0.85	0.85	0.94





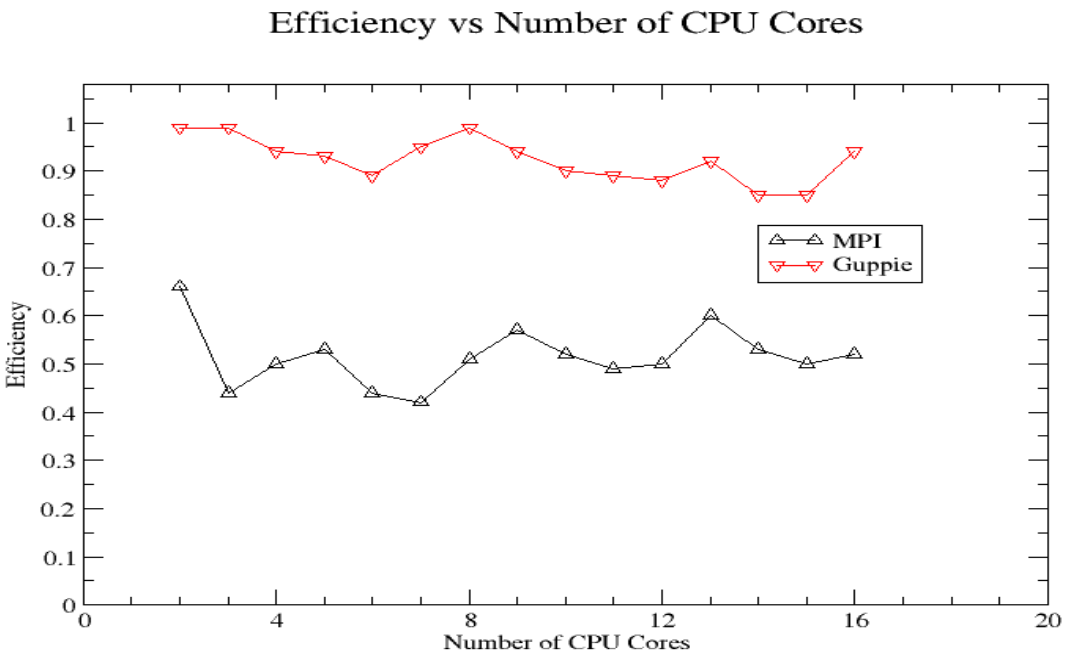


Figure 6.3 Results (10 Averaging Iterations)

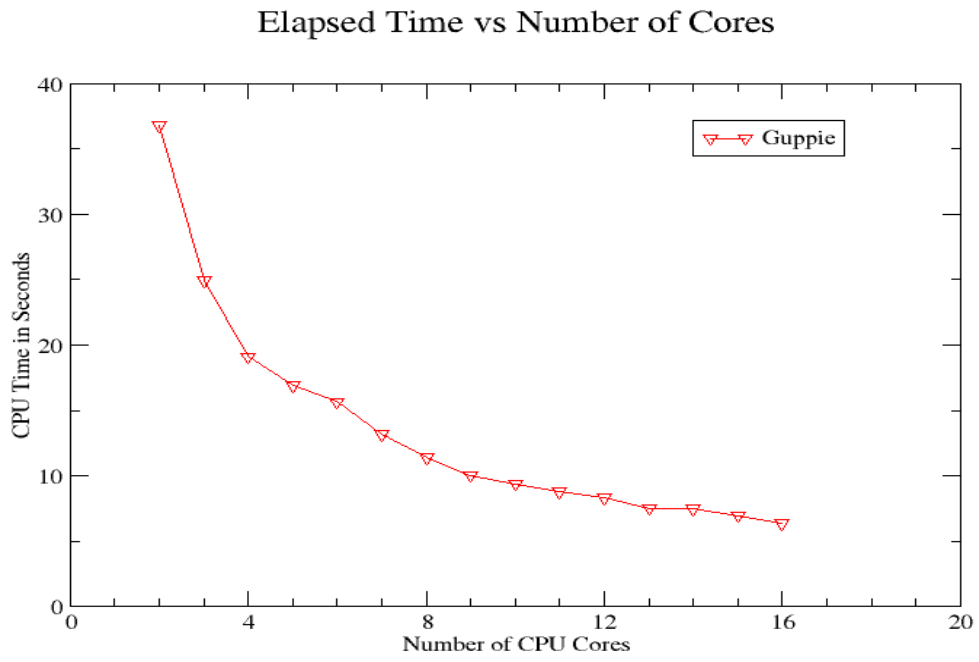
For the MPI and Guppie run-time analysis, the Guppie code outperforms the MPI code by about 38 seconds when 16 workers and 16 processors are used. The Guppie code is very efficient throughout the testing of the application. When the total number of cores matches the total number of worker processes (4, 8, and 16), Guppie's efficiency runs very high with 0.94 for 4 cores, 0.99 for 8 cores, and 0.94 for 16 cores. This is much better than MPI's efficiencies of 0.50 for 4 cores, 0.51 for 8 cores, and 0.52 for 16 cores.

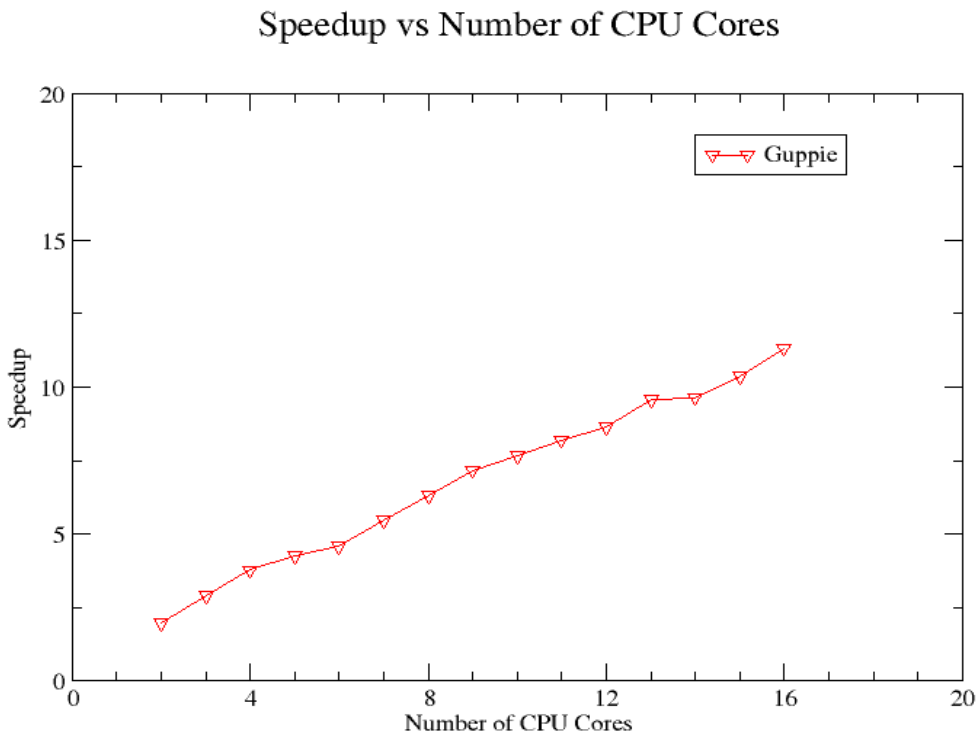
In the previous application, the image averaging algorithm was executed ten times. This produced very high speedup and efficiency results. An additional test was conducted to demonstrate that Guppie maintains high speedup and efficiency even when the computation amount is not as significant. The same application was tested using only one averaging iteration, rather than ten averaging iterations as seen previously. Listed

below are the execution time, speedup, and efficiency for each test run of the Guppie application using only one averaging iteration:

cores	2	3	4	5	6	7	8
time	0m36.79s	0m24.92s	0m19.12s	0m16.89s	0m15.67s	0m13.18s	0m11.40s
speedup	1.96	2.89	3.77	4.26	4.59	5.46	6.32
efficiency	0.98	0.96	0.94	0.85	0.77	0.78	0.79

9	10	11	12	13	14	15	16
0m10.04s	0m9.39s	0m8.78s	0m8.33s	0m7.51s	0m7.47s	0m6.95s	0m6.36s
7.17	7.67	8.20	8.64	9.59	9.64	10.36	11.32
0.79	0.77	0.75	0.72	0.73	0.69	0.69	0.71




Speedup vs Number of CPU Cores
Guppie
Speedup
Number of CPU Cores
0
5
10
15
20
0
4
8
12
16
20

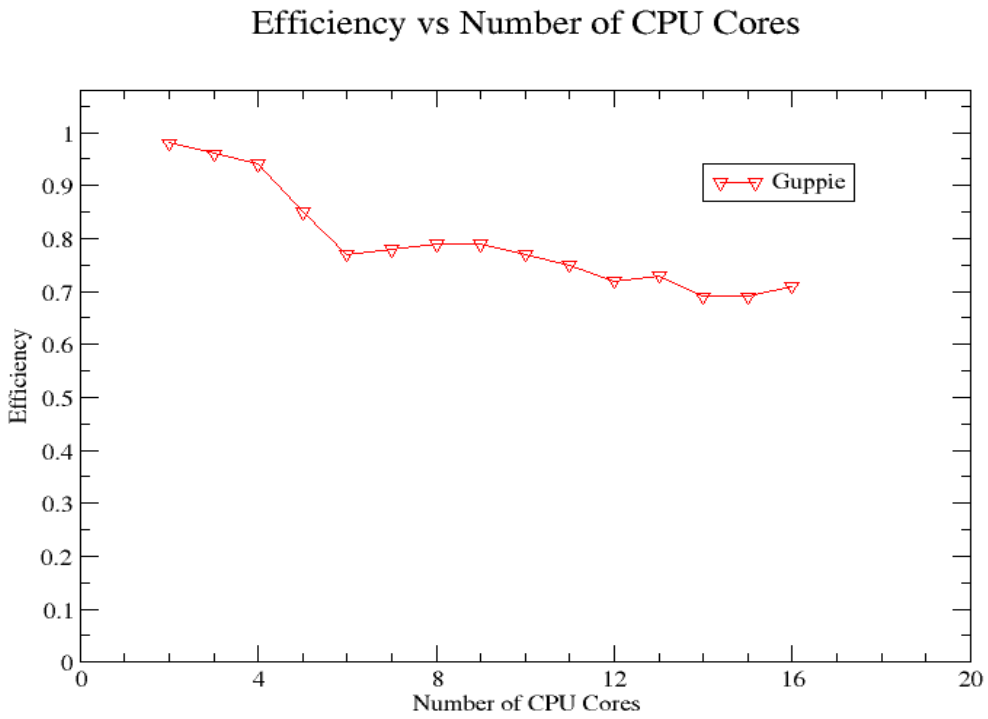


Figure 6.4 Results (1 Averaging Iteration)

The above results demonstrate Guppie maintains efficiency even when the computational workload is not very large. In this case, the images were only averaged one time each, as opposed to ten times in the previous application. When ten averaging iterations are used, Guppie achieves very high efficiencies with 0.94 for 4 cores, 0.99 for 8 cores, and 0.94 for 16 cores. When using only one averaging iteration, Guppie produces more traditional efficiencies with 0.94 for 4 cores, 0.79 for 8 cores, and 0.71 for 16 cores.

One important feature of Guppie is its implicit dynamic load balancing. If the total number of images is not evenly divisible by the number of worker processes, then some worker processes have to process more images than other worker processes. For example, if 64 images are used along with 5 worker processes, then 4 worker processes

have to process 13 images each, and 1 worker process has to process only 12 images. In MPI, extra code has to be written to account for uneven load balancing (The MPI program in Appendix B accounts for uneven load balancing, even though uneven load balancing was not encountered in the analysis tests). Guppie features dynamic load balancing by using the tuple space to coordinate the number of images that need to be processed. In the Guppie code discussed in the previous section, the add("Images", 0, imageCt) and take("Images", 0, currentImage) function calls are used to determine if an image needs to be processed. As soon as all the images have been processed, the value contained in the tuple that matches the parameters name = "Images" and id = 0 is equal to 64, stating that all images have been processed. Any worker process to withdraw the tuple will see that the value indicates that all images have been processed and there is no more processing to do. This is how Guppie allows for dynamic load balancing without any extra effort from the user.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Guppie achieves high efficiency on parallel processes using large arrays on shared memory multiprocessors. This is shown by writing, testing, and comparing parallel applications with both MPI and Guppie. When the number of cores matches the number of workers in the image processing application, Guppie achieves a speedup of 3.74 for 4 cores, 7.91 for 8 cores, and 14.98 for 16 cores. This translates to an efficiency of 0.94 for 4 cores, 0.99 for 8 cores, and 0.94 for 16 cores. This is much more efficient than MPI's speedup of 1.99 for 4 cores, 4.09 for 8 cores, and 8.28 for 16 cores, which translates to an efficiency of 0.50 for 4 cores, 0.51 for 8 cores, and 0.52 for 16 cores.

Guppie is not only efficient, but it is an easy programming environment. Guppie provides shared memory capabilities that are easy to use, similar to OpenMP. MPI can be rather confusing to use at times due to its lengthy number of parameters in the majority of its functions. For instance, the "MPI_Recv" function has seven parameters that are needed in order to receive a message from another process. Those parameters include the message, number of bytes, data type, source, tag, MPI communicator, and message status. Guppie allows for the user to avoid this hassle by performing the necessary computations in the master and worker classes. The user should only be concerned with the data itself.

Guppie has an advantage over MPI in two other critical areas. By using a tuple space to store data tuples, it is easy to perform a bag of tasks application. Bag of tasks

type of applications provide for dynamic load balancing. In MPI, more effort is needed in creating an application that efficiently manages load balancing.

There has been research demonstrating that MPI is more efficient than Linda models [40]. While MPI may be faster than a Linda-like model when working with small data sets, Guppie's utilization of shared memory on a local multi-core machine to handle large data objects, such as images, has better efficiency than MPI. Guppie's ease of use and efficiency should be desirable in certain applications, such as image processing or any bag of tasks applications.

One of Guppie's best attributes is that it is modular and scalable. The Linda model or any of its extensions can take advantage of Guppie's shared memory mechanism. This would give a viable option to parallel programmers that need to work with large data or do not have access to multiple machines connected over a network for added processing power. Perhaps some models could implement a dual feature of using multiple machines over a network when that is the most efficient and using Guppie's local processing power when that is most efficient. It may be hard to determine in some automated way which would be more efficient, but it could be worth researching.

Guppie can also be extended to be able to work with other data types besides integers, doubles, strings, and vectors. It would be useful to extend Guppie to automatically handle access to multidimensional arrays in shared memory, relieving the programmer of many of the bookkeeping details of managing matrices and images.

It also may be useful to be able to store the tuple space on disk. This would be useful in the event of executing algorithms that take hours or days to execute, such as in

decrypting codes. During certain phases of the application, a “snapshot” of the tuple space can be stored on disk in the event of a failure. For example, if an algorithm takes four hours to run, and it fails after three and a half hours, it would be beneficial to have a snapshot of the tuple space around the time it failed so the application would not have to be run from the beginning. This could also be used to share the tuple space with unrelated processes and processes executing at different times.

A feature I would like to provide is a GUI which will act as a debugging tool. The GUI will allow a user to view the contents of the tuple space. If an error occurs, whether that is an error that disrupts or stops the running of the application, or just a logical error, the user will be able to perform a single step progression of what has been processed in the tuple space. This GUI will allow for much easier debugging than current methods. The GUI will track the master's interaction with the tuple space. The master is the only process that has direct access to the tuple space. The workers send requests to the master, the master processes those requests, and the master performs the required work and accesses the tuple space. The GUI could also provide request information that has been processed by the master. This would be beneficial because the user could see what process made the request and exactly what that request was. From that point, the user would have more of an advantage of debugging the code than he would if he had to debug it an alternate way.

APPENDIX A

SERIAL APPLICATION

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <fcntl.h>
#include <sys/times.h>

using namespace std;

#define rowSize 600
#define colSize 800
#define mosaicRow 4800
#define mosaicCol 6400
#define totalImages 64
#define xDimension 8
#define yDimension 8
#define numWorkers 1

unsigned char imageData[rowSize][colSize][3];
unsigned char convertedImage[rowSize][colSize][3];
unsigned char outputImage[rowSize][colSize][3];
int pixels[rowSize][colSize][3];
int averagedPixels[rowSize][colSize][3];
unsigned char mosaic[mosaicRow][mosaicCol][3];

void writeToFile(int fd);
void average();

int main()
{
        int fd, cd, i, j, k, a, b;
        int imagesCompleted = 0;
        int startRow, finishRow, startCol, finishCol;
        ofstream output;

        fd = open("shapes.img", O_RDONLY);
        read(fd, imageData, rowSize*colSize*3);
        cd = close(fd);
        fd = open ( "mosaic.img", O_WRONLY | O_CREAT, 0666 );


        while(imagesCompleted < totalImages)
        {
           // store unsigned char data into workable integers
           for(i = 0; i < rowSize; i++)
           {
               for(j = 0; j < colSize; j++)
               {
                  for(k = 0; k < 3; k++)
```

```
                {
                    pixels[i][j][k] = imageData[i][j][k];
                }
            }
        }

        average();

        // convert integer data back to unsigned char data
        for(i = 0; i < rowSize; i++)
        {
            for(j = 0; j < colSize; j++)
            {
               for(k = 0; k < 3; k++)
               {
                   convertedImage[i][j][k] = pixels[i][j][k];
               }
            }
        }

        startRow = (imagesCompleted / yDimension) * rowSize;
        finishRow = startRow + rowSize;
        startCol = (imagesCompleted % xDimension) * colSize;
        finishCol = startCol + colSize;

        a = 0;
        for(i = startRow; i < finishRow; i++)
        {
             b = 0;
             for(j = startCol; j < finishCol; j++)
             {
                 for(k = 0; k < 3; k++)
                 {
                    mosaic[i][j][k] = convertedImage[a][b][k];
                 }
                 b++;
             }
             a++;
        }

        imagesCompleted++;
    } // end of while(imagesCompleted < totalImages)

    // store in output file
    writeToFile(fd);
    output.close();

    return 0;

}

void writeToFile(int fd)
{
        write(fd, mosaic, mosaicRow*mosaicCol*3);
}
```

```
void average()
{
        int counter, i, j, k, x, y, a, b, c;
        int sum, totalCounted, average;
        int upperLeftRow, lowerRightRow;
        int upperLeftCol, lowerRightCol;

        for(counter = 0; counter < 10; counter++)
        {
            for(i = 0; i < rowSize; i++)
            {
                for(j = 0; j < colSize; j++)
                {
                   for(k = 0; k < 3; k++)
                   {
                      sum = 0;
                      totalCounted = 0;

                      // get an 21 x 21 grid
                      upperLeftRow = i - 10;
                      if(upperLeftRow < 0)
                         upperLeftRow = 0;

                      upperLeftCol = j - 10;
                      if(upperLeftCol < 0)
                         upperLeftCol = 0;

                      lowerRightRow = i + 10;
                      if(lowerRightRow >= rowSize)
                         lowerRightRow = rowSize;

                      lowerRightCol = j + 10;
                      if(lowerRightCol >= colSize)
                         lowerRightCol = colSize;

                      // sum 21 x 21 grid
                      for(x = upperLeftRow; x <= lowerRightRow; x++)
                      {
                        for(y = upperLeftCol; y <=  lowerRightCol; y++)
                        {
                           sum = sum + pixels[x][y][k];
                           totalCounted++;
                        }
                      }

                      average = sum / totalCounted;
                      averagedPixels[i][j][k] = average;
                   }
                }
             }
             // update pixels array in case of multiple iterations

             for(a = 0; a < rowSize; a++)
             {
                for(b = 0; b < colSize; b++)
```

```
                    {
                        for(c = 0; c < 3; c++)
                        {
                            pixels[a][b][c] = averagedPixels[a][b][c];
                        }
                    }
                }

            }
        }
```

APPENDIX B

MPI APPLICATION

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <fcntl.h>
#include "mpi.h"

using namespace std;

#define rowSize 600
#define colSize 800
#define mosaicRow 4800
#define mosaicCol 6400
#define totalImages 64
#define xDimension 8
#define yDimension 8
#define numWorkers 4

unsigned char imageData[rowSize][colSize][3];
unsigned char convertedImage[rowSize][colSize][3]; // used in worker
process
unsigned char outputImage[rowSize][colSize][3]; // used in writer
process, received from worker
int pixels[rowSize][colSize][3];
int averagedPixels[rowSize][colSize][3];
unsigned char mosaic[mosaicRow][mosaicCol][3];

void average();
void writeToFile(int fd);

int main(int argc, char* argv[])
{
   ofstream mosaicOutput;
   int source, rank, numtasks;
   int tag = 0;
   int fd, cd;
   int currentImage, averagedImages;
   int i, j, k, a, b, x;
   int startRow, finishRow, startCol, finishCol;
   MPI_Status status;
   int workerRequest[2];
   int imagesStillLeft;

   // initialize mpi environment
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   if(rank == 0) // master process
```

```
{
   // read image
   fd = open("shapes.img", O_RDONLY);
   read(fd, imageData, rowSize*colSize*3);
   cd = close(fd);

   // open output file
   fd = open("mosaic.img", O_WRONLY | O_CREAT, 0666);

   // store unsigned char data into workable integers
   for(i = 0; i < rowSize; i++)
   {
      for(j = 0; j < colSize; j++)
      {
         for(k = 0; k < 3; k++)
         {
            pixels[i][j][k] = imageData[i][j][k];
         }
      }
   }

   currentImage = 0;
   averagedImages = 0;
   while(currentImage < totalImages)
   {
      // process image requests from workers and send images
      for(x = 1; x < (numWorkers + 1); x++) // + 1 for master
      {
         if(currentImage < totalImages)
         {
            MPI_Recv(workerRequest, 2, MPI_INT, x, tag,
                     MPI_COMM_WORLD, &status);
            if(workerRequest[0] == 1) // need an image
            {
               source = workerRequest[1];
               MPI_Send(pixels, (rowSize * colSize * 3), MPI_INT,
                        source, tag, MPI_COMM_WORLD);
               currentImage++;
            }
         }
      }

      // receive averaged images from workers
      for(x = 1; x < (numWorkers + 1); x++)
      {
         if(averagedImages < totalImages)
         {
            MPI_Recv(workerRequest, 2, MPI_INT, x, tag,
                     MPI_COMM_WORLD, &status);
            if(workerRequest[0] == 2) // need to receive image
            {
               source = workerRequest[1];
               MPI_Recv(averagedPixels, (rowSize * colSize * 3),
                        MPI_INT, source, tag, MPI_COMM_WORLD,
                        &status);
```

```
                // convert averaged image to unsigned char array
                for(i = 0; i < rowSize; i++)
                {
                   for(j = 0; j < colSize; j++)
                   {
                      for(k = 0; k < 3; k++)
                      {
                         convertedImage[i][j][k] =
                             averagedPixels[i][j][k];
                      }
                   }
                }

                // store averaged image in mosaic
                startRow = (averagedImages / xDimension) * rowSize;
                finishRow = startRow + rowSize;
                startCol = (averagedImages % xDimension) * colSize;
                finishCol = startCol + colSize;

                a = 0;
                for(i = startRow; i < finishRow; i++)
                {
                   b = 0;
                   for(j = startCol; j < finishCol; j++)
                   {
                      for(k = 0; k < 3; k++)
                      {
                         mosaic[i][j][k] = convertedImage[a][b][k];
                      }
                      b++;
                   }
                   a++;
                }
                averagedImages++;
             } // end of if(workerRequest[0] == 2)
          } // end of if(averagedImages < totalImages)
       } // end of for(x = 1; x < (numWorkers + 1); x++)
    } // end of while(currentImage < totalImages)

    writeToFile(fd);
    cd = close(fd);
}// end of if(rank == 0)
else if(rank >= 1) // worker processes
{
    currentImage = 0;
    while(currentImage < (totalImages / numWorkers))
    {
       // send image request to master
       workerRequest[0] = 1;
       workerRequest[1] = rank;
       MPI_Send(workerRequest, 2, MPI_INT, 0, tag, MPI_COMM_WORLD);

       // receive image from master
       MPI_Recv(pixels, (rowSize * colSize * 3), MPI_INT, 0, tag,
                MPI_COMM_WORLD, &status);
```

```
            average();

            // send request to master saying to receive averaged image
            workerRequest[0] = 2;
            workerRequest[1] = rank;
            MPI_Send(workerRequest, 2, MPI_INT, 0, tag, MPI_COMM_WORLD);

            // send averaged image to master so master can store in mosaic
            MPI_Send(pixels, (rowSize * colSize * 3), MPI_INT, 0, tag,
                     MPI_COMM_WORLD);
            currentImage++;
         } // end of while(currentImage < (totalImages / numWorkers))

         /* need to account for images still left to be computed
            for example, 5 worker processes and 16 total images
            one worker process will have to do 4 images while the others
            do 3 */

         if((currentImage * numWorkers) < totalImages)
         {
            imagesStillLeft = totalImages % numWorkers;
            if(rank < (imagesStillLeft + 1)) // + 1 to account for master
            {
               // send image request to master
               workerRequest[0] = 1;
               workerRequest[1] = rank;
               MPI_Send(workerRequest, 2, MPI_INT, 0, tag,
                        MPI_COMM_WORLD);

               // receive image from master
               MPI_Recv(pixels, (rowSize * colSize * 3), MPI_INT, 0, tag,
                        MPI_COMM_WORLD, &status);

               average();

               // send request to master to receive averaged image
               workerRequest[0] = 2;
               workerRequest[1] = rank;
               MPI_Send(workerRequest, 2, MPI_INT, 0, tag,
                        MPI_COMM_WORLD);

               // send avg image to master so master can store in mosaic
               MPI_Send(pixels, (rowSize * colSize * 3), MPI_INT, 0, tag,
                        MPI_COMM_WORLD);
               currentImage++;
            } // end of if(rank < (imagesStillLeft + 1))
         } // end of if((currentImage * numWorkers) < totalImages)
      } // end of else if(rank >= 1)

      MPI_Finalize();
      return 0;
   }
```

```
void writeToFile(int fd)
{
        write(fd, mosaic, mosaicRow*mosaicCol*3);
}


void average()
{
        int counter, i, j, k, x, y, a, b, c;
        int sum, totalCounted, average;
        int upperLeftRow, lowerRightRow;
        int upperLeftCol, lowerRightCol;

        for(counter = 0; counter < 10; counter++)
        {
            for(i = 0; i < rowSize; i++)
            {
                for(j = 0; j < colSize; j++)
                {
                   for(k = 0; k < 3; k++)
                   {
                      sum = 0;
                      totalCounted = 0;

                      // get an 21 x 21 grid
                      upperLeftRow = i - 10;
                      if(upperLeftRow < 0)
                         upperLeftRow = 0;

                      upperLeftCol = j - 10;
                      if(upperLeftCol < 0)
                         upperLeftCol = 0;

                      lowerRightRow = i + 10;
                      if(lowerRightRow >= rowSize)
                         lowerRightRow = rowSize;

                      lowerRightCol = j + 10;
                      if(lowerRightCol >= colSize)
                         lowerRightCol = colSize;

                      // sum 21 x 21 grid
                      for(x = upperLeftRow; x <= lowerRightRow; x++)
                      {
                        for(y = upperLeftCol; y <=  lowerRightCol; y++)
                        {
                           sum = sum + pixels[x][y][k];
                           totalCounted++;
                        }
                      }

                      average = sum / totalCounted;
                      averagedPixels[i][j][k] = average;
                   }
                }
             }
```

```
                // update pixels array in case of multiple iterations

                for(a = 0; a < rowSize; a++)
                {
                   for(b = 0; b < colSize; b++)
                   {
                      for(c = 0; c < 3; c++)
                      {
                         pixels[a][b][c] = averagedPixels[a][b][c];
                      }
                   }
                }

          }
}
```

APPENDIX C

GUPPIE APPLICATION (MASTER)

```
#include <stdlib.h>
#include "master.h"
#include <QSharedMemory>
#include <fstream>

#define rowSize 600
#define colSize 800
#define mosaicRow 4800
#define mosaicCol 6400
#define totalImages 64
#define numProcs 16

class newMaster:public master
{
     public:

     void setTasks();
     void startWorkers();
};

char actor_name[32] = "newMaster";
newMaster *m;
unsigned char imageData[rowSize][colSize][3];

void newMaster::setTasks()
{
     int imagesCt;
     int size;
     int fd, cd;
     int byteCounter;
     int i, j, k;
     unsigned char *ucharPtr;
     unsigned char *to;

     // set shared memory variable counter
     sharedMemArrayCounter = 0;

     // add image counters
     for(i = 0; i < totalImages; i++)
        add("Images", 0, imagesCt);

     // add tuples for workers to know image processing is complete
     for(i = 0; i < numProcs - 1; i++)
        add("Images", 0, -1);

     // add a tuple for the last worker process to output mosaic
     add("Images", 0, -2);
```

```
     // create shared memory seg for a single image for workers to read
     size = rowSize * colSize * 3;
     create_shm("image", 0, size);

     // create shared memory segment for the mosaic
     size = mosaicRow * mosaicCol * 3;
     create_shm("mosaic", 1, size);

     fd = open("shapes.img", O_RDONLY);
     read(fd, imageData, rowSize*colSize*3);
     cd = close(fd);

     // store pointer to shared memory segment
     ucharPtr = (unsigned char*)location_shm("image", 0);

     for(byteCounter = 0; byteCounter < size_shm("image, 0");
         byteCounter++)
     {
        for(i = 0; i < rowSize; i++)
        {
           for(j = 0; j < colSize; j++)
           {
              for(k = 0; k < 3; k++)
              {
                 to = ((unsigned char*)(ucharPtr + byteCounter));
                 memcpy(to, &imageData[i][j][k],
                        sizeof(unsigned char));
                 byteCounter++;
              }
           }
        }
     }
}

void newMaster::startWorkers()
{
     int i;

     for(i = 0; i < 16; i++)
     {
         start_process("./philosopherRev");
     }
}

int main(int argc, char **argv)
{
     int i;

     m = new newMaster();
     m->setTasks();
     m->startWorkers();
     m->register_task((char *)"data", recordData);
     m->run();
     m->detachSM("image", 0);
     m->detachSM("mosaic", 1);
```

```
     return 0;
}
```

Guppie Application (worker)

```
#include <stdlib.h>
#include "actor.h"
#include <unistd.h>
#include <QSharedMemory>
#include <fstream>

#define rowSize 600
#define colSize 800
#define mosaicRow 4800
#define mosaicCol 6400
#define totalImages 64
#define xDimension 8
#define yDimension 8
#define numProcs 16

char **id;
QSharedMemory sharedMemory, smMosaic;
unsigned char imageData[rowSize][colSize][3];
unsigned char convertedImage[rowSize][colSize][3]; // used in worker
process
unsigned char outputImage[rowSize][colSize][3]; // used in writer
process, received from worker
int pixels[rowSize][colSize][3];
int averagedPixels[rowSize][colSize][3];
unsigned char mosaic[mosaicRow][mosaicCol][3];

class workerProc:public worker
{
    public:
    void run();
    void average();
};


void workerProc::average()
{
        int counter, i, j, k, x, y, a, b, c;
        int sum, totalCounted, average;
        int upperLeftRow, lowerRightRow;
        int upperLeftCol, lowerRightCol;

        for(counter = 0; counter < 10; counter++)
        {
            for(i = 0; i < rowSize; i++)
            {
                for(j = 0; j < colSize; j++)
                {
                   for(k = 0; k < 3; k++)
                   {
                      sum = 0;
                      totalCounted = 0;
```

```
                        // get an 21 x 21 grid
                        upperLeftRow = i - 10;
                        if(upperLeftRow < 0)
                           upperLeftRow = 0;

                        upperLeftCol = j - 10;
                        if(upperLeftCol < 0)
                           upperLeftCol = 0;

                        lowerRightRow = i + 10;
                        if(lowerRightRow >= rowSize)
                           lowerRightRow = rowSize;

                        lowerRightCol = j + 10;
                        if(lowerRightCol >= colSize)
                           lowerRightCol = colSize;

                        // sum 21 x 21 grid
                        for(x = upperLeftRow; x <= lowerRightRow; x++)
                        {
                          for(y = upperLeftCol; y <=  lowerRightCol; y++)
                          {
                             sum = sum + pixels[x][y][k];
                             totalCounted++;
                          }
                        }

                           average = sum / totalCounted;
                           averagedPixels[i][j][k] = average;
                     }
                  }
               }
               // update pixels array in case of multiple iterations

               for(a = 0; a < rowSize; a++)
               {
                  for(b = 0; b < colSize; b++)
                  {
                     for(c = 0; c < 3; c++)
                     {
                        pixels[a][b][c] = averagedPixels[a][b][c];
                     }
                  }
               }

         }
}



void workerProc::run()
{
    sharedMemArrayCounter = 0;
    int pid, imageSize, mosaicSize, byteCounter;
    int i, j, k, currentImage, imageCt;
```

```
unsigned char *ucharPtr, *to;
int startRow, finishRow, startCol, finishCol;
ofstream output, output2;
int fd;
string name;
int a, b, c;  // temporary counters

pid = atoi(id[1]);

// attach to single image segment
attach_shm("image", 0);
// attach to mosaic segment
attach_shm("mosaic", 1);

imageSize = size_shm("image", 0);
mosaicSize = size_shm("mosaic", 1);

ucharPtr = (unsigned char*)location_shm("image", 0);

byteCounter = 0;
for(i = 0; i < rowSize; i++)
{
   for(j = 0; j < colSize; j++)
   {
      for(k = 0; k < 3; k++)
      {
         imageData[i][j][k] = *(ucharPtr + byteCounter);
         byteCounter++;
      }
   }
}

name = "Images";
take("Images", 0, currentImage);

ucharPtr = (unsigned char*)location_shm("mosaic", 1);
while(currentImage != -1 && currentImage != -2)
{
   // store unsigned char data into workable integers
   for(i = 0; i < rowSize; i++)
   {
      for(j = 0; j < colSize; j++)
      {
         for(k = 0; k < 3; k++)
         {
            pixels[i][j][k] = imageData[i][j][k];
         }
      }
   }

   average();

   // convert integer data back to unsigned char data
   for(i = 0; i < rowSize; i++)
   {
      for(j = 0; j < colSize; j++)
```

```
      {
         for(k = 0; k < 3; k++)
         {
            convertedImage[i][j][k] = pixels[i][j][k];
         }
      }
   }

   byteCounter = 0;
   for(i = 0; i < rowSize; i++)
   {
      for(j = 0; j < colSize; j++)
      {
         for(k = 0; k < 3; k++)
         {
            to = ((unsigned char*)((ucharPtr + byteCounter)+
                  (currentImage * rowSize * colSize * 3)));
            memcpy(to, &convertedImage[i][j][k],
                   sizeof(unsigned char));
            byteCounter++;
         }
      }
   }

   take("Images", 0, currentImage);
} // end of while(currentImage != -1 && currentImage != -2)

// now pull image from shared memory mosaic seg and store in mosaic
if(currentImage == -2)
   /* all images have been used, have a process read from
      shared memory and output to file */
{
   ucharPtr = (unsigned char*)location_shm("mosaic", 1);
   byteCounter = 0;

   a = 0;
   fd = open ( "mosaic.img", O_WRONLY|O_CREAT, 0666 );

   while(a < totalImages)
   {
      startRow = (a / yDimension) * rowSize;
      finishRow = startRow + rowSize;
      startCol = (a % xDimension) * colSize;
      finishCol = startCol + colSize;

      for(i = startRow; i < finishRow; i++)
      {
         for(j = startCol; j < finishCol; j++)
         {
            for(k = 0; k < 3; k++)
            {
               mosaic[i][j][k] = *(ucharPtr + byteCounter);
               byteCounter++;
            }
         }
      }
```

```
            a++;
        } // end of while(a < totalImages)

        write(fd,mosaic,mosaicRow*mosaicCol*3);
        close(fd);
    } // end of if(currentImage == -2)
}


char worker_name[32] = "workerProc";
worker *a;

int main(int argc, char **argv)
{
    int i, value;

    id = argv;
    a = new workerProc();
    a->connect ( argc, argv );
    a->run();
    a->detach_shm("image", 0);
    a->detach_shm("mosaic", 1);
    a->send_exit();
    return 0;
}
```

REFERENCES

1: Ahuja, S., Carriero, N., Gelernter, D., and Krishnaswamy, V. *The architecture of a Linda coprocessor*. Proceedings of the 15th annual international symposium on computer architecture (1988), 240-249.

2: Ahuja, S., Carriero, N., Gelernter, D., Krishnaswamy, V.: *Matching Language and Hardware for Parallel Computation in the Linda Machine*. IEEE Trans. Computers 37(8): 921-929 (1988)

3: Bakken, D., Schlichting, R.: *Supporting Fault-Tolerant Parallel Programming in Linda*. IEEE Transactions on Parallel and Distributed Systems, 6 (3), March 1995, 287–302

4: Bjornson, R., Carriero, N., Gelernter, D.: *From Weaving Threads to Untangling the Web: A View of Coordination from Linda's Perspective*. COORDINATION 1997: 1-17

5: Bjornson, R., Carriero, N., Gelernter, D., Leichter, J.: *Linda in adolescence*. ACM SIGOPS European Workshop 1986

6: Bruni, R., Montanori, U.: *Concurrent Models for Linda with Transactions*. Mathematical Structures in Computer Science, vol. 14, no. 3, pp. 421-468, June 2004

7: Cappello, F., Krawezik, G.: *Performance comparison of MPI and OpenMP on shared memory multiprocessors*. Concurrency and Computation: Practice and Experience 18(1): 29-61 (2006)

8: Carriero, N., Gelernter, D.: *Applications Experience with Linda*. 1988: 173-187

9: Carriero, N., Gelernter, D., Leichter, J.: *Distributed Data Structures in Linda*. 1986: 236-242

10: Carriero, N., Gelernter, D.: *The S/Net's Linda Kernel*. ACM Trans. Comput. Syst. 4(2): 110-129 (1986)

11: Carriero, N., Gelernter, D., Chandran, S., Chang, S.: *Parallel Programming in Linda*. ICPP 1985: 255-263

12: Carriero, N., Gelernter, D.: *A Foundation for Advanced Compile-time Analysis of Linda Programs*. LCPC 1991: 389-404

13: Carriero, N., Gelernter, D., Hupfer, S., Kaminsky, D.: *Coordination Applications of Linda*. Research Directions in High-Level Parallel Programming Languages. 1991: 187-194

14: Carriero, N., Gelernter, D.: *How to Write Parallel Programs: A Guide to the Perplexed*. ACM Comput. Surv. 21(3): 323-357 (1989)

15: Carriero, N., Gelernter, D.: *Linda in Context*. Commun. ACM 32(4): 444-458 (1989)

16: Carriero, N., Gelernter, D., Zuck, L.: *Bauhaus Linda*. ECOOP Workshop 1994: 66-76

17: Carriero, N., Gelernter, D., Mattson, T., Sherman, A.: *The Linda Alternative to Message-Passing Systems*. Parallel Computing 20(4): 633-655 (1994)

18: Carriero, N., Gelernter, D.: *Data Parallelism and Linda*. LCPC 1992: 145-159

19: Carriero, N., Gelernter, D.: *Coordination Languages and Their Significance*. Commun. ACM 35(2): 96-107 (1992)

20: Carriero, N., Gelernter, D.: *A Computational Model of Everything*. Commun. ACM 44(11): 77-81 (2001)

21: Carriero, N.: *An Implementation of* Linda *for a NUMA Machine.* Parallel Computing 24(7): 1005-1021 (1998)

22: Carriero, N., Gelernter, D., Hupfer, S.: *Collaborative Applications Experience with the Bauhaus Coordination*. HICSS (1) 1997: 310-319

23: Carriero, N., Freeman, E., Gelernter, D., Kaminsky, D.: *Adaptive Parallelism and Piranha*. IEEE Computer 28(1): 40-49 (1995)

24: Ceriotti, M., Murphy, A., Picco, G.: *Data Sharing vs. Message Passing: Synergy or Incompatibility? An Implementation-Driven Case Study*. 23rd Annual ACM Symposium on Applied Computing (SAC), Fortaleza, Brazil, March 2008

25: Chalmers, A., Clayton, P., Wells, G.: *Linda implementations in Java for concurrent systems*: Research Articles, Concurrency and Computation: Practice & Experience, v.16 n.10, p.1005-1022, August 2004

26: Chalmers, A., Clayton, P., Wells, G.: *A comparison of Linda implementations in Java*. Communicating Process Architectures 2000, volume 58 of Concurrent Systems Engineering Series, pages 63--75. IOS Press, September 2000

27: Chalmers, A., Clayton, P., Wells, G.: *Extending Linda to simplify application development*. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), pages 108--114. CSREA Press, June 2001

28: Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, 2001.

29: Costa, P., Mottola, L., Murphy, A., Picco, G.: *TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks*. Proceedings of the International Workshop on Middleware for Sensor Networks. 7th International Middleware Conference (Middleware), Melbourne, Australia, November, 2006

30: Costa, P., Mottola, L., Murphy, A., Picco, G.: *Tuple Space Middleware for Wireless Networks*. Invited contribution to the book Middleware for Network Eccentric and Mobile Applications. Springer, 2009. (to appear)

31: Ganeshamoorthy, K., Ranasinghe, D.: *On the Performance of Parallel Neural Network Implementations on Distributed Memory Architectures*. CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID) - Volume 00, May 2008

32: GDAL – Geospatial Data Abstraction Library. (n.d.). Retrieved from http://www.gdal.org

33: GigaSpaces Technologies Ltd. GigaSpaces. (n.d.). Retrieved from http://www.gigaspaces.com/index.htm, 2001

34: Grama, A., Gupta, A., Karypis, G., and Kumar, V.: Introduction to Parallel Computing, Second Ed. Pearson Education Limited, 2003.

35: Krawezik, G.: *Performance comparison of MPI and three openMP programming styles on shared memory multiprocessors.* ACM Symposium on Parallel Algorithms and Architectures archive. Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, 2003

36: Leichter, J., Minsky, N.: *Law-Governed Linda as a Coordination Model*. Object-Based Models and Languages for Concurrent Systems, LNCS, No. 924, pages 125-146, Springer-Verlag 1995

37: Minsky, N., Ungureanu, V.: *Regulated Coordination in Open Distributed Systems*. In Proc. of Coordination'97: Second International Conference on Coordination Models and Languages, LNCS 1282, September 1997

38: Minsky, N., Minsky, Y., Ungureanu, V.: *Making Tuple Spaces Safe for Heterogeneous Distributed Systems*. ACM Symposium on Applied Computing (SAC 2000) March 2000 Como, Italy

39: Minsky, N., Minsky, Y., Ungureanu, V.: *Safe Tuplespace-Based Coordination in Multi Agent Systems*. Journal of Applied Artificial Intelligence (AAI), (Vol 15, No. 1, pages: 11-33) January 2001

40: Morariv, V., Cunningham, M., Letterman, M.: *A performance and portability study of parallel applications using a distributed computing testbed*. HCW'97: Proceedings of the 6th Heterogeneous Computing Workship, April 1997.

41: Mueller, B., Schule, L., Wells, G.: *A Tuple Space Web Service for Distributed Programming - Simplifying Distributed Web Services Applications*. WEBIST (1) 2008: 93-100

42: Murphy, A., Picco, G.: *Transiently Shared Tuple Spaces for Sensor Networks.* Proceedings of the Euro-American Workshop on Middleware for Sensor Networks. 2nd International Conference on Distributed Computing in Sensor Systems, San Francisco (CA, USA), June 2006. (Invited contribution).

43: Murphy, A., Roman, G., Varghese, G.: *Dependable Message Delivery to Mobile Units*. Mobile Computing Handbook (CRC Press), pp. 227-252, 2004

44: Murphy, A., Picco, G., Roman, G.: *Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, no. 3, pp. 279-328, July 2006

45: Pacheco, Peter S.: Parallel Programming with MPI. Morgan Kaufmann Publishers, 1997.

46: Picco, G., Murphy, A., Roman, G.: *Lime: Linda Meets Mobility*. 21st International Conference on Software Engineering (ICSE), Ed. D. Garlan, Los Angeles, CA, USA, May, ACM Press, pp. 368-377, 1999

47: Roman, G., Murphy, A., Picco, *G.: Coordination and Mobility, In Coordination of Internet Agents: Models, Technologies, and Applications*. Springer, pp. 254-273, 2000

48: Sluga, T.: *Modern C++ Implementation of the Linda Coordination Language for Distributed Applications. CPP Linda. Thesis*. University of Applied Sciences and Arts, Germany. 2007

49: Wells, G.: *A Programmable Matching Engine for Application Development in Linda*. PhD thesis, University of Bristol, U.K., 2001

50: Zenios, S., Lasken, R.: *The Connection Machines CM-1 and CM-2: solving nonlinear network problems*. ICS'88: Proceedings of the 2nd international conference on Supercomputing, pp. 648-658, 1988.