1-1-2007

Buffer allocation in message passing systems: An implementation for Mpi

Jeffrey Sampson
University of Nevada, Las Vegas

BUFFER ALLOCATION IN MESSAGE PASSING SYSTEMS:

AN IMPLEMENTATION FOR MPI

by

Jeffrey Sampson

Bachelor of Science
University of Texas at Austin
2003

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2007

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

NOVEMBER 16TH , 2007

The Thesis prepared by

JEFFREY SAMPSON

Entitled

BUFFER ALLOCATION IN MESSAGE PASSING SYSTEMS: AN IMPLEMENTATION FOR MPI

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

1017-53

ABSTRACT

**Buffer Allocation in Message Passing Systems:
An Implementation for MPI**

by

Jeffrey Sampson

Dr. Jan B. Pedersen, Examination Committee Chair
Assistant Professor of Computer Science
University of Nevada, Las Vegas

Message passing applications that perform asynchronous communication need sufficient buffer space to hold all undelivered messages, or else the applications may deadlock. Determining the minimum amount of buffer space an application needs is called the Buffer Allocation Problem, and has been shown to be intractable [BPW]. However, an epoch based polynomial-time algorithm that approximates the Buffer Allocation Problem has been proposed by Pedersen et al. [PBS]. The algorithm partitions application executions into epochs and intersperses barrier synchronizations between them, thus limiting the number of message buffers necessary to ensure deadlock-freedom.

In this thesis, we describe an implementation of the epoch based algorithm. Our implementation analyzes and performs barrier synchronizations for MPI (Message Passing Interface) applications. We use a modified version of MPI to gather information about the messages sent during the execution, and then use a standalone Java program to analyze the protocol (communication structure) and build a graph which serves as the foundation for the computation of barrier synchronizations. We then pass this information to MPI, making it available for automatic barrier synchronization. Finally, we present the

results of an empirical study of various applications implemented to test our approximation algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

For several decades, advances in computer hardware have usually come from improvements in the design of single processor architectures. In recent years, however, the methods traditionally used to achieve better performance in CPUs have been yielding diminishing returns. As a result, there has been a trend towards parallel computing. In parallel computing, multiple processors simultaneously coordinate to solve a problem.

Distributed computing is one model of parallel computation. This model assumes that the processors in a system do not share any memory space. Therefore, the programs executed by the processors, called processes, cannot read each other's data. Instead, data is exchanged through messages sent between processes.

Unlike the shared memory model of parallel computing, which requires custom hardware, more processors can be easily added to a distributed system. Additional machines can be added to a cluster of computers by simply connecting them to a network. This makes distributed computing more scalable than the shared memory model. Distributed computing is becoming more available, thanks to the low cost of processor and network hardware.

In a distributed system consisting of multiple computers, processes must communicate over the network connecting the system. Dealing with network protocols is

cumbersome and time-consuming for programmers, and protocol implementations vary across networks and operating systems. Several popular libraries have been written that handle process communication, as well as process creation and initialization, called message passing libraries. Applications that use these libraries are referred to as message passing applications. By using a standard message passing library, it is easier for programmers to develop applications that run on multiple systems. The Message Passing Interface (MPI) library is the most popular library for message passing applications [BDHRS].

Although MPI makes it easier to write applications for multiple systems, it cannot guarantee the portability of applications that use asynchronous message passing. In synchronous message passing, the sending process must wait until the intended recipient is ready to accept a message. Asynchronous message passing allows the sender to proceed as soon as the message has been injected into the system by storing it in specially allocated memory, called a buffer. If no buffers are free, the sender must wait until one become available, causing the send operation to behave synchronously. Many MPI applications assume communication is asynchronous in order to run faster. If there are not enough buffers available, communication may cease to be asynchronous, and deadlock can ensue. However, the number of buffers available is dependent on the system hardware. Thus, an application that relies on asynchronous communication may deadlock when ported to systems with fewer buffers than the one used for development.

In order to port an MPI application, it is necessary to determine the minimum number of buffers needed to prevent such deadlock. Furthermore, if there are not enough buffers on the target system, the application must be modified to compensate for the

lower number of buffers. Unfortunately, determining the minimum number of buffers needed by a message passing application (solving the *Buffer Allocation Problem*) has been shown to be an intractable optimization problem [BPW]. There is a heuristic–based approach that finds an approximation equal to or greater than the optimal solution [BPW], but the number of buffers required by the approximation may be large.

One novel approach described by Pedersen et al. reduces the buffer requirements by dividing an application's execution into sequential intervals called epochs [PBS]. At the end of an epoch, every process must wait until all other processes have completed the epoch. This technique, called barrier synchronization, guarantees that any message buffers will be free at the end of the epoch, and can be reused in subsequent epochs. The new buffer requirements for each epoch and the entire application can be computed using the previously mentioned heuristic algorithm.

Objectives and Goals of this Thesis

In this thesis, we present an implementation of the epoch based algorithm for MPI applications. Information about an application's communication is collected at runtime using an addition to the MPI library, which we have written. This information is used by a standalone Java application to create epochs and determine the buffer requirements of the application. The output of the Java application can then be used by the MPI application at runtime to perform barrier synchronizations at the end of epochs.

We also describe an empirical investigation of this implementation, using five asynchronous MPI applications. Our investigation indicates that using epochs reduces the buffer requirements of MPI applications, while increasing the runtime by a constant

factor. Additionally, we show that a user can improve the runtime of an application if extra buffers are available. By providing more buffers, the user can trade memory for execution time, by allowing the application to use fewer epochs.

Organization of this Thesis

An overview of the Buffer Allocation Problem and the epoch based approach is given in Chapter 2. In Chapter 3, we provide details about our implementation of the epoch algorithm, including how to use if for MPI applications. Chapter 4 describes the results of our experiments with the epoch algorithm on five MPI applications. Finally, in Chapter 5 we present conclusions and recommendations for future work.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this chapter, we discuss previous research on preventing buffer related deadlock. This research only involves messages sent between individual processes, not messages that are broadcast to groups of processes. Any message passing algorithm can be implemented using process to process communication. Although this research is applicable to message passing programs in general and similar problems in the operations research community [ANA] [REI] [SHE], our focus is limited to applications that use the Message Passing Interface (MPI) standard.

Asynchronous Message Passing and Deadlock in MPI

At the start of an MPI application, n processes are created and execute simultaneously. Each process is assigned a process id. A process i can send a message to a process j by calling a send function with the id of j and the contents of the message. To receive the message from i, process j must call a receive function using the process id of i.

When a receive function is called, the receiving process will block until the message arrives. The send function can perform either a synchronous or asynchronous send. In a synchronous send, the send function waits until the receive function is called

by the destination process before returning. In an asynchronous send, the message is copied into a message buffer on the receiver's side, after which the send function returns. A message buffer holds the contents of a message until the receive function is called by the destination process, at which point the message is delivered. Asynchronous sends allow the sending process to continue execution without having to wait until the receiving process is ready for the message.

Message buffers require memory in the system running the receiving process. If many messages are being sent, then all available buffers may be used. When there are no available buffers, the send function will wait until a buffer is free or the receiving process is ready for the message. This causes the send function to behave synchronously. In MPI applications that rely on asynchronous message passing, this can lead to deadlock.

For example, suppose two processes exchange messages. If both processes call their send function first, followed by their receive function, then the messages must be stored in buffers. If no buffers are available for either process, they will both block, waiting for each other to call the receive function. Since this will never happen, both processes cannot proceed, and the application is deadlocked.

The amount of memory available for buffers differs on every system. An MPI application that terminates successfully on one system may deadlock on another due to lack of memory for buffers. A user will not know if an application is portable without manually testing it for any given system. An application would be more portable if it was known beforehand how many buffers needed to be allocated to prevent deadlock.

Communication Graphs

In order to determine the buffer requirements of an MPI application, it is necessary to record all communication that happens during the execution of the program. A program trace S is a log of all events between the start of the program and its termination. A send event is the completion of a send operation, and a receive event is the completion of a receive event. A send completes when the sending process is no longer blocked, not when the message is received. For a program trace to be useful, the MPI application must have a static communication pattern. That is, the application must produce the same trace S every time that it is run for a given problem size.

A communication graph G of a program trace S is a directed graph $G = G(S) = (V,A)$ where the set of vertices $V = \{v_{i,c} \mid 1 \leq i \leq n, 0 \leq c \leq e_i\}$ corresponds to events in the trace, where e_i is the number of events performed by process i. Vertex $v_{i,0}$ represents the start event of process i and vertex $v_{i,c}$, represents either a send or a receive event. The former is called a *start vertex* and the latter are called *send* and *receive* events respectively. For each vertex $v_{i,c}$, i is called the process number and c is called the event number.

The arc set A consists of two disjoint arc sets: the computation arc set P and the communication arc set C. A computation arc $(v_{i,c}, v_{i,c+1}) \in P$, $0 \leq c \leq e_i$, represents a computation within process i, which is an "internal event" in the terminology of Lamport [LAM]. A communication arc $(v_{i,c}, v_{j,d}) \in C$ represents a communication between different processes, i and j, where $v_{i,c}$ is a send vertex and $v_{j,d}$ is a receive vertex (see figure 2-1). The vertex $v_{i,c-1}$ is called the *parent* vertex of the vertex $v_{i,c}$, and the vertex $v_{i,c+1}$ is called the *child* vertex of $v_{i,c}$. The vertex $v_{j,d}$ connected to $v_{i,c}$ by a

communication arc is called the *sibling* vertex of $v_{i,c}$. Note, the process arcs are drawn without orientation for clarity; they are always oriented downwards. The i^{th} *process component* G_i of G is the subgraph $G_i = (V_i, A_i)$ where $V_i = \{v_{i,c} \in V \mid 0 \leq c \leq e_i\}$.

The communication graph contains an ordering of all events in the trace. That is, a path from vertex v_a to vertex v_b in the graph indicates that event a must occur before event b. By transitivity, an event a must occur before an event b if a path exists from vertex v_a to vertex v_b. Event a is said to precede event b, denoted by $a \rightarrow b$. Since no buffers are initially allocated, arcs between send and receive vertices are considered bidirectional.

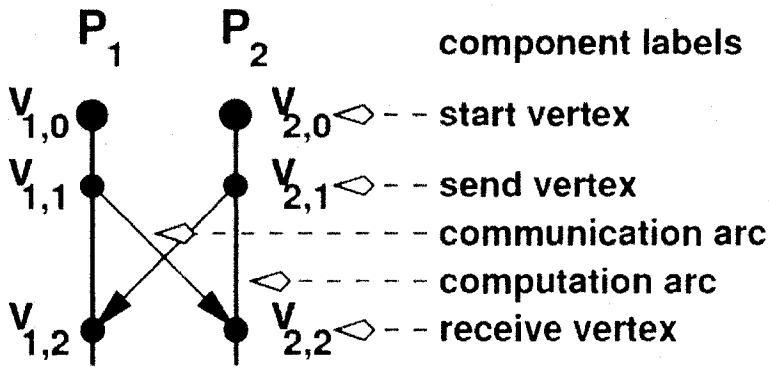

Figure 2-1: A communication graph for two processes.

The Buffer Allocation Problem

Determining the minimum amount of buffers needed to avoid deadlock in a message passing program was formally defined by Brodsky et al. as the Buffer Allocation Problem [BPW]. To solve the Buffer Allocation Problem, an algorithm must compute for an application consisting of n processes the n-tuple of nonnegative integers $\beta = \{b_1, b_2, \ldots, b_n\}$ representing the number of buffers needed by each process to avoid deadlock.

The Buffer Allocation Problem was proven by Brodsky et al. to be NP-hard [BPW]. This was done by using the communication graph framework to reduce a special case of the Boolean Satisfiability Problem to the Buffer Allocation Problem.

Problems that are NP-hard or NP-complete cannot be optimally solved by any currently known algorithm in polynomial time or less. Consequently, no program can be used to find an optimal solution for these problems because the large run time required is impractical. Instead, a program must use efficient algorithms or heuristics that find an approximate solution. An approximate solution may be suboptimal, but it can still be useful if it is a certain range from the optimal solution.

To approximate a solution to the Buffer Allocation Problem, a program can use the solution to the Delay Free Buffer Allocation Problem, which was also defined by Brodsky et al. [BPW]. A delay is defined as the wait time that occurs when there are no message buffers available and the sending process must block until one is available. The Delay Free Buffer Allocation Problem is to determine the minimum amount of buffers $\beta = \{b_1, b_2, \ldots, b_{n-1}\}$ such that there are no delays when sending messages. Unlike the Buffer Allocation Problem, the Delay Free Buffer Allocation Problem is tractable because there is an algorithm that solves it in polynomial time. Since a message passing application that is delay free will also be deadlock free, the number of buffers β will also be sufficient to avoid deadlock during execution. Therefore, this algorithm provides a suboptimal solution to the Buffer Allocation Problem.

The Delay Free Buffer Allocation Algorithm

In the Delay Free Buffer Allocation algorithm, or DFBA, the number of buffers $\beta = \{b_1, b_2, \ldots, b_n\}$ needed to avoid delay is determined by examining the communication graph G of the message passing program. For each receive vertex in G, the algorithm must find the interval, $I_{i,t}$, which corresponds to the time between a message arriving at process i and its receipt. An interval requires one buffer to ensure delay free sends. If two intervals overlap in a process, two buffers will be required, three overlapping intervals will require three buffers, etc. Thus, the minimum number of buffers b_i needed is the maximum overlap density over all of the intervals in process i.

Intervals are found by computing the terminal communication dependency of each receive vertex. For two vertices $v_{i,c}$ and $v_{i,t}$ in process i, $t > c$, vertex $v_{i,t}$ is communication dependent on vertex $v_{i,c}$ if $v_{i,c}$ is the start vertex or if there is a vertex $v_{j,d}$ in process j, such that there is a path from $v_{i,c}$ to $v_{j,d}$ and there is an arc from $v_{j,d}$ to $v_{i,t}$. Vertex $v_{i,t}$ is terminally communication dependent on $v_{i,c}$ if $v_{i,t}$ is communication dependent on $v_{i,t}$, and not communication dependent on any vertices $v_{i,l}$, where $c < l < t$.

The terminal communication dependencies of every vertex in G can be computed using a dynamic programming algorithm. Each vertex $v_{j,d}$ is associated with an integer vector $a_{j,d}$ containing n entries, where $a_{j,d}[i] = c$ means that there is a path from vertex $v_{i,c}$ to vertex $v_{j,d}$. Initially, $a_{j,d}[k] = -1$ for $k \neq j$ and $a_{j,d}[k] = d$, otherwise. The entries in vector $a_{j,d}$ are computed by taking the element wise maximum of the vectors in the parent and sibling vertices of vertex $v_{j,d}$. To do this, a depth first traversal of G is done, starting at the last vertex of each process component and following the arcs in the opposite direction.

The Delay Free Buffer Allocation algorithm consists of three steps. In the first step, the terminal communication dependency of each receive vertex is computed. This step takes O ($|V| n$) time, where V is the set of vertices in G and n is the number of processes, because the number of arcs in G is bounded by $3|V| / 2$ and the pairwise comparison takes n steps. In the second step, the interval for each receive vertex is found. This is done by looking up the terminal communication dependency in the vector of the sibling vertex. Because this step requires one table lookup per receive vertex, the run time is O ($|V|$). For the final step, the intervals within each process component are sorted and a sweep is performed to find the maximum overlap density, which takes O ($|V| \log |V|$) time. So, the total complexity of DFBA is O ($|V| n + |V| \log |V|$) time. Since the number of processes n is usually much smaller than the size of the set of vertices $|V|$, the run time of DFBA in practice is O ($|V| \log |V|$).

The Epoch Based Approach to the Buffer Allocation Problem

Since the Delay Free Buffer Allocation algorithm is not an optimal solution to the Buffer Allocation Problem, the buffer allocation given by the DFBA algorithm for a message passing application may greatly exceed what is necessary for the application to stay deadlock free. In some cases, it may require more buffers than are available in memory. This limits the utility of the algorithm to users of message passing applications. There is another approach described by Pedersen et al. that can lower the buffer requirements for a message passing application [PBS]. In this approach, the communication graph G is decomposed into discrete sections called epochs.

An epoch E is a subgraph of G, containing vertices from G and the arcs between them. The subgraph is a maximal strongly connected component of G, meaning that for

every vertex in E there is a path to every other vertex in E. Since there can be no vertex outside the subgraph that has a path to and from a vertex in the subgraph, all epochs in G must be disjoint. That is, a vertex can belong to only one epoch. Since the arcs between sibling vertices are considered bidirectional, every epoch E has at least one send and receive vertex. An epoch is called simple if contains exactly one send and receive vertex. An epoch is called complex if it contains more than two vertices.

The communication graph G can be represented as a series of epochs $E_1, E_2, \ldots, E_m$, such that for any two vertices $a \in E_i$ and $b \in E_j$, if $a \rightarrow b$ then $i \leq j$. Two epochs, E_i and E_j, are causally ordered if there are two vertices $a \in E_i$ and $b \in E_j$ such that $a \rightarrow b$ or $b \rightarrow a$. In the first case E_i precedes E_j, while in the second case E_j precedes E_i. Otherwise, the two epochs are causally unordered, meaning they can be ordered either way. Figure 2-2 shows a partitioning of a graph G into epochs.

The buffer requirements for a message passing program can be reduced by requiring every process to synchronize at the end of an epoch. When a process reaches the end of an epoch, it must wait for every other process to reach the end of the epoch before it can proceed. This is called barrier synchronization. If a process does not have an event in an epoch, it can simply perform a barrier synchronization event immediately after finishing the preceding epoch. Since any receive events will have been completed by the time each process reaches the end of an epoch, all message buffers will be free and can be reused in subsequent epochs.

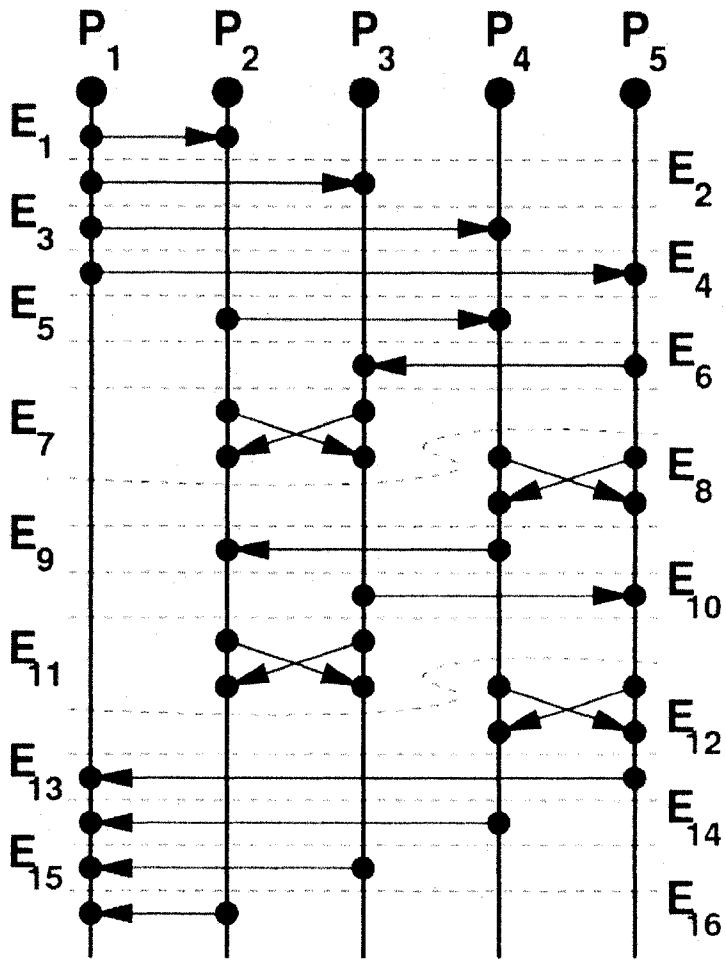

Figure 2-2: The graph *G* partitioned into epochs.

Because every epoch is a subgraph of *G*, the DFBA algorithm can be used to determine the number of buffers needed to avoid deadlock during the epoch. A simple epoch does not require any buffers, because it contains only one receive event. A complex epoch will require at least minimal buffer allocation to avoid deadlock. The number of buffers required for the entire application is determined by taking the element wise maximum over the delay free buffer allocations of each epoch.

This approach is a trade-off between run time and memory requirements. Because buffers are reused in each epoch, fewer buffers are needed during the lifetime of the

application. However, requiring every process to wait at the end of an epoch causes a delay. The more epochs that are in the graph *G*, the greater the overall cost to the application's run time.

G can be partitioned into epochs using the standard algorithm for computing the strongly connected components of a graph [CLR], which by definition are epochs. The strongly connected components algorithm uses two depth first searches on *G*, which takes linear time. Since the algorithm decomposes *G* into a smaller graph, the epochs and their order can be represented by a directed acyclic graph (DAG) *D*. The arcs between two epochs in *D* correspond to the arcs between the last vertices in first epoch and the first vertices in the second epoch. An example of a graph *G* being decomposed into its epochs in *D* is shown in figure 2-3.

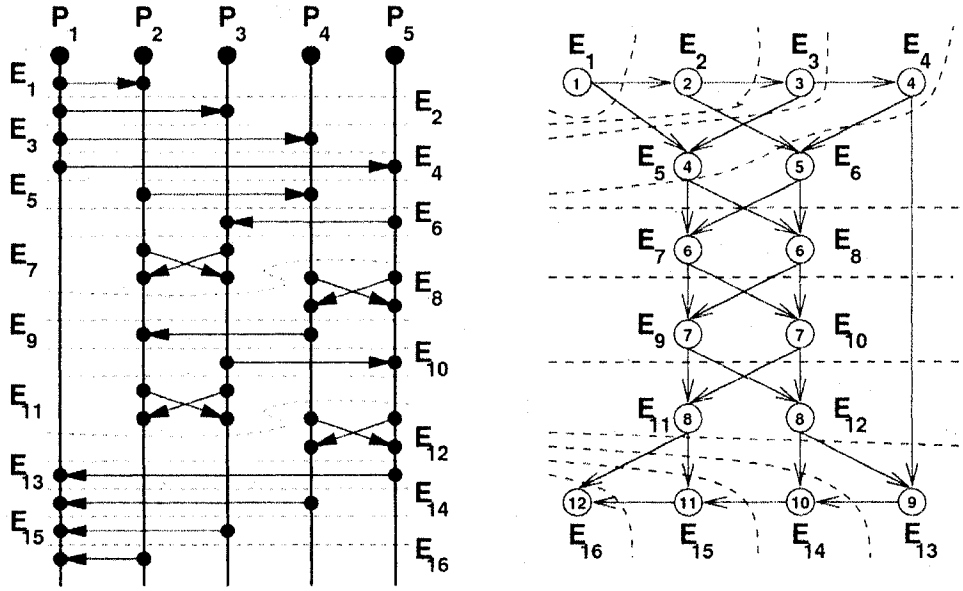

Figure 2-3: A communication graph and its corresponding DAG *D*.

Combining Epochs into Super Epochs

The drawback of using epochs is the runtime overhead associated with the barrier synchronization at the end of each epoch. To improve on this scheme, Pedersen et al. also introduce an algorithm that reduces the number of epochs in the graph, and hence reduces the number of barrier synchronizations required during the execution of the application, without increasing the buffer requirements [PBS]. The algorithm is used after the graph G has been decomposed into its strongly connected components and the delay free buffer allocation has been computed over all epochs. This algorithm combines epochs into larger ones called super epochs.

A super epoch is a composition of consecutive epochs $E_i \circ E_{i+1} \circ \ldots \circ E_j$, where E_i precedes E_{i+1} and $E_i \circ E_{i+1}$ is the composition of two epochs. Like epochs, super epochs are disjoint, meaning an epoch can belong to only one super epoch. Also like epochs, super epochs are either simple or complex. A simple super epoch contains only simple epochs, whereas a complex super epoch contains one or more complex epochs. The graph G can be represented as an ordered series of super epochs, where every epoch in G belongs to a super epoch and G equals the composition of every super epoch.

In a super epoch, processes are required to perform a barrier synchronization event at the end of the super epoch, not at the end of each epoch within. This lowers the number of barrier synchronizations each process must perform, but it may also raise the number of buffers required during the super epoch. The DFBA algorithm can be used to find the necessary buffer allocation for a super epoch, since all epochs in the super epoch are consecutive subgraphs of G. A simple super epoch requires no buffers, because it comprises only simple epochs, making it an acyclic graph. It was proven by Brodsky et

al. that buffers are not needed to avoid deadlock in acyclic communication graphs [BPW]. Complex super epochs, however, do require buffer allocation.

The algorithm described by Pedersen et al. builds super epochs by examining the epochs in the DAG *D* [PBS]. Since the arcs between epochs are held in *D*, it can be used to locate consecutive epochs. None of the super epochs created by the algorithm require a greater buffer allocation than any epoch in *D*. This leads to fewer barrier synchronizations without raising the buffer allocation.

Ideally, the number of super epochs should be as small as possible. To minimize the amount of super epochs, the algorithm exploits the fact that simple epochs require no buffers. Any number of simple super epochs can be composed together without requiring any buffers, because the composition will remain simple. Furthermore, a simple super epoch can be added to the beginning of a complex super epoch without increasing its buffer requirement [PBS]. Unfortunately, a simple super epoch cannot be added to the end of a complex super epoch, because it might require more buffers. Therefore, it is advantageous to add as many simple super epochs as possible to the start of a complex super epoch. To do this, super epochs are built in two parts, the head and the tail. The head is built by composing consecutive simple epochs, until a complex epoch is reached are there are no remaining epochs. The tail is then built by composing epochs until the buffer limit is reached and there are no remaining epochs. Finally, the head and tail are composed into one super epoch.

The algorithm for creating super epochs, Algorithm 1, is shown in figure 2-4. The input to Algorithm 1 is the DAG *D*, which is found by running the strongly connected components algorithm on the graph *G*. For output, Algorithm 1 returns a list

of super epochs L and a Delay Free Buffer Allocation β. L represents the partitioning of G into a consecutive sequence of super epochs, and β is the buffer allocation necessary for avoiding deadlock during each super epoch in L. A result of the algorithm is shown in figure 2-5.

```
Algorithm 1: Constructing Super Epochs
Input: D
Output: L, β
Local: Z, X, Π_head, Π_tail
Z ← {v | v ∈ D ∧ indegree(v) = 0}
β ← max_{v∈D} {DFBA(E_v)}
While Z ≠ ∅ do
      Π_head ← ∅
      Foreach v ∈ Z do
            If E_v is simple then
                  Π_head ° Π_head ° E_v
                  X ← {u | (v,u) ∈ D ∧ indegree(v) = 1}
                  Remove v from Z and D
                  Append X to Z
            end
      end
      Π_tail ← ∅
      Foreach v ∈ Z do
            If DFBA(Π_tail ° E_v) ≤ β then
                  Π_tail ← Π_tail ° E_v
                  X ← {u | (v,u) ∈ D ∧ indegree(v) = 1}
                  Remove v from Z and D
                  Append X to Z
            end
      end
      Append Π_head ° Π_tail to L
      end
```

Figure 2-4: The algorithm for constructing super epochs.

The main loop of Algorithm 1 constructs one super epoch per iteration. It runs until the list Z is empty, that is when there are no epochs in D left to process. The first inner loop

constructs the head of the super epoch, and the second inner loop constructs the tail. Afterwards the head and tail are composed to form a super epoch, which is added to the list L.

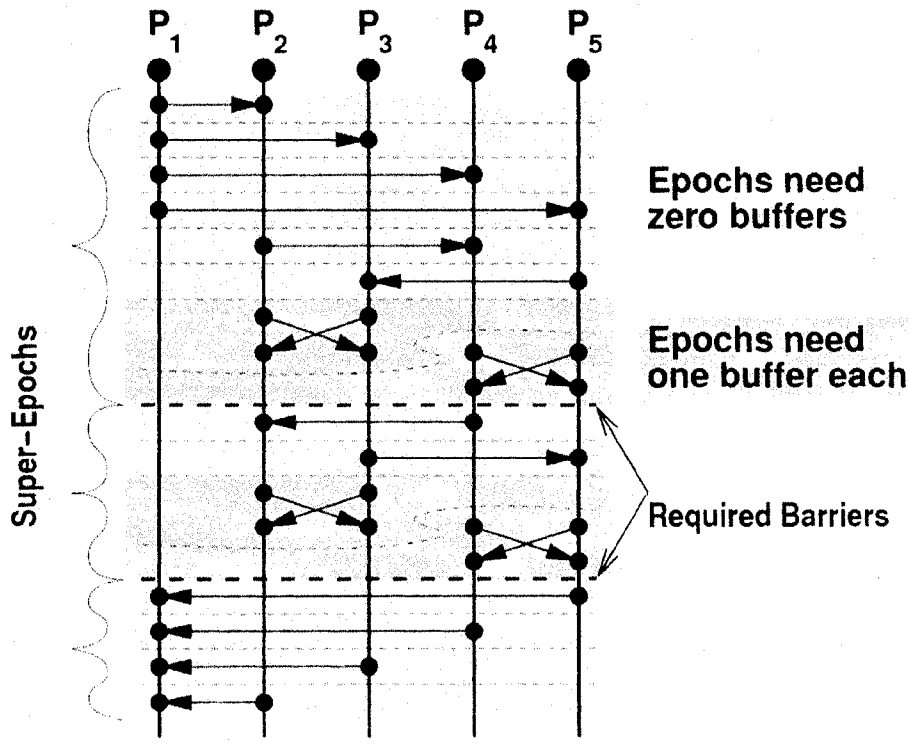

Figure 2-5: A graph partitioned into Super Epochs.

To build the head, the first loop iterates through each epoch in Z. If an epoch is simple, it is removed from Z and D, and composed with the head. When an epoch is removed from the Z, all adjacent epochs are added to Z. This ensures that every epoch in D will be processed eventually. The first loops halts when Z is empty or there are no simple epochs left in Z.

Like the first loop, the second loop builds the tail by iterating through Z. If an epoch and the tail can be composed without exceeding the delay free buffer allocation β,

then the composition is performed and the epoch is removed from Z and D. As in the first loop, after the epoch is removed any epochs it has arcs to in D are added to Z. The second loop halts when Z is empty or when there are no epochs in Z that can be added to the tail. Since β is sufficient for every epoch in D, and the tail is initially empty, at least one epoch must be added to the tail during the second loop. Therefore, at least one epoch is removed from Z and D and added to L during an iteration of the main loop, and the algorithm must eventually terminate. The total complexity of Algorithm 1 is $O(|V|^2)$, where V is the set of vertices in G.

CHAPTER 3

THE PROTOTYPE BARRIER SYNCHRONIZATION TOOL

Our prototype synchronization tool consists of two parts: a C-library that is used with MPI applications to perform the data collection and the synchronization, and a standalone Java program which computes the synchronization points based on the data collected during the initial run. To use the synchronization tool on the data collected from executing an MPI application, the user must recompile the application to include the C-library. The C-library allows the user to run the application in collection or synchronization mode. To create a log file for a given application (i.e., the input data for the Java analysis program), the user must execute the MPI application in collection mode. During collection mode, the C-library will record every send and receive event in separate files for every process. These files are then concatenated into one log file with a shell script. The log file is used as input to the Java program, which performs the off-line analysis portion of the process. The Java program creates the communication graph by reading the log file, then partitions the graph into epochs using the strongly connected components algorithm [CLR]. Next, the Java application computes the Delay Free Buffer Allocation β over all epochs. It then uses β to run Algorithm 1 and create a list of super epochs. Finally, it outputs a synchronization file, which tells each process where to perform synchronization. This entire procedure is illustrated in Figure 3-1.

The user can now use the synchronization file to run the MPI application in synchronization mode for problem instances of the same size and communication pattern. Using the synchronization file for any other application or communication pattern is not legal. There are two types of synchronization modes that the user can run the program in: *full barrier synchronization* and *partial barrier synchronization*. In full barrier synchronization, every process will synchronize with each other at the end of a super epoch. In partial barrier synchronization, only processes that have events in the following super epoch will synchronize. This allows processes that do not need to synchronize to continue computation without delay.

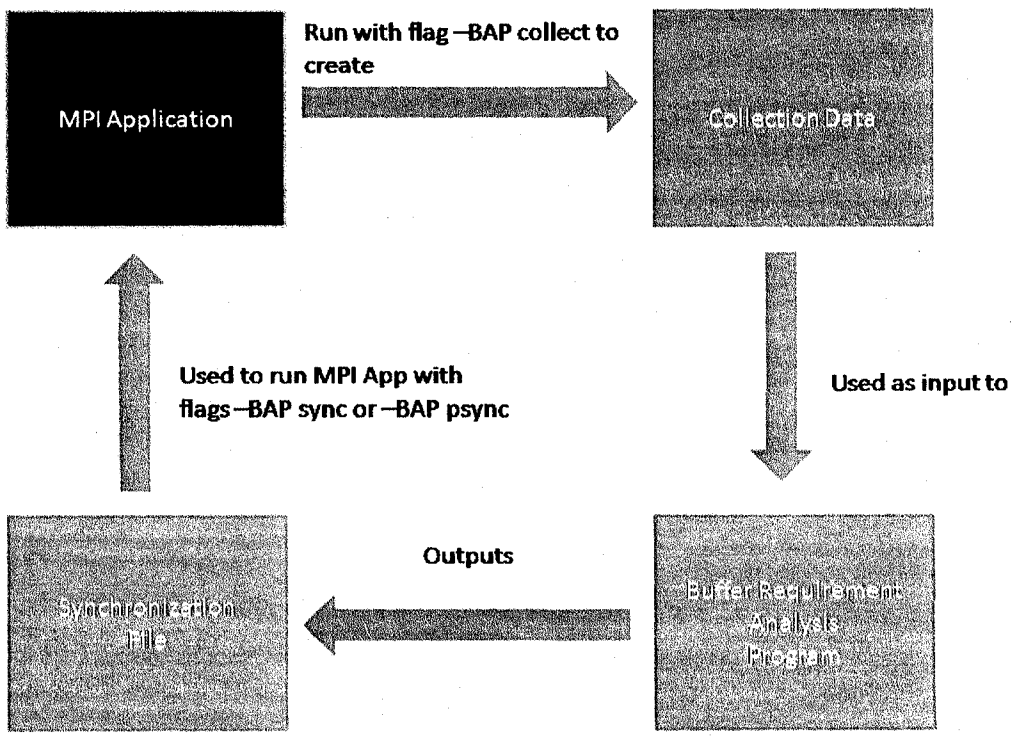

Figure 3-1: Using the synchronization tool with an MPI application.

Implementing Collection, Synchronization, and Partial Synchronization in MPI

The C-code needed for collection, full barrier synchronization, and partial barrier synchronization is contained in our file *bap.c*. This file is used as an interface between the MPI program and the standard MPI library. All of the MPI applications that we tested use four functions defined in the MPI library: *MPI_Init()*, *MPI_Send()*, *MPI_Recv()*, and *MPI_Finalize()*. In *bap.c*, we define our own versions of each of these functions that act as wrappers around the original versions. C-preprocessor *#define* macros are used to replace MPI calls with calls to the wrapper functions. Our implementation of these functions perform the additional work for collection or synchronization, before or after calling the original MPI function.

To use collection or synchronization in an MPI application, the application must be recompiled with the MPI compiler to include the code in *bap.c*. The application can then be executed with special flags that enable collection or synchronization. The *collect* flag is used to execute in collection mode, the *sync* flag is used to execute in full barrier synchronization mode, and the *psync* flag is used to execute in partial barrier synchronization mode. We refer to these as *collect*, *sync*, and *psync* modes respectively. When using the *collect* flag, the user must specify the name of the log file in which to record sends and receives. Each process will create a file using this name and its process id as a suffix. All of these files are combined to create the log file. For *sync* and *psync*, the user must specify the name of the synchronization file generated by the buffer requirements analysis program on the command line.

The *MPI_Init()* Function

The *MPI_Init()* function is used at the beginning of each MPI process to register with the MPI system. Since it is used at the beginning of an MPI process, our version of the function also performs the initialization needed for collection and synchronization after calling the original MPI function. In *collect* mode, each process will open a file to record send and receive events. In *sync* and *psync* mode, each process will open and read the synchronization file. The synchronization file lists every super epoch and the event numbers where each super epoch ends for a process. Each process records these event numbers in an array, and then closes the file. In *psync* mode, an additional array is used to record which super epochs the process must synchronize after.

The *MPI_Send()* and *MPI_Recv()* Functions

The *MPI_Send()* and *MPI_Recv()* functions are used to send and receive messages between processes respectively. Our implementations of these functions call the original versions at the end, after doing any necessary work, as in Figure 3-2. In *collect* mode, a process will record all the information about a send or receive event in a file before calling the original *MPI_Send()* or *MPI_Recv()*. In *sync* and *psync* mode, a process synchronizes with other processes if needed before calling the original function. A process checks if synchronization is necessary using the information recorded from the synchronization file.

To synchronize in *sync* mode, a process calls the *MPI_Barrier()* function. When a process calls this function, it will block until every other process in the group has also called it. This forces every process to synchronize. This function cannot be used in *psync* mode, because not every process may need to synchronize. Instead, a special

process called *pbarrier* is created through the MPI function *MPI_Comm_spawn()*, which every process calls. This function creates a special process that the other processes can communicate with through the MPI system. The *pbarrier* process is used to perform synchronizations in *psync* mode. After it is created, *pbarrier* will also read the synchronization file so it will know where each process needs to synchronize. A process synchronizes in *psync* mode by sending a message to the *pbarrier* process and waiting for a reply. For each super epoch, the *pbarrier* process will wait for a message from each process that needs to synchronize at the end of the super epoch. After receiving a message from every process, *pbarrier* will send a reply to all of them in turn. Since the processes will not receive replies until each one has sent a message to *pbarrier*, this will cause them to synchronize.

```
int _MPI_Send(char pname[100], int line, void *buf,
              int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm) {
  if (__bap__sync)        //if running in sync or psync mode:
    _bap_sync();          //synchronize if necessary
  if (__bap_collect)      //if running in collect mode, write
                          //send event to file
    fprintf(fp,"%d:E=%d:S:%d:%d;\n",__bap_my_rank,
                                    __bap_event,dest,line);
  __bap_event++;          //increment event counter
  //call original MPI_Send function in MPI library and
  //return its return value
  return MPI_Send(buf, count, datatype, dest, tag, comm);
}
```

Figure 3-2: The wrapper function for *MPI_Send()*.

The *MPI_Finalize()* Function

The *MPI_Finalize()* function is called at the end of every MPI process. In *collect* mode, a process will close the file that it has been recording in. In *sync* and *psync* mode, any final synchronizations will be performed by the process. The original *MPI_Finalize()* function is then called.

Java Classes used in the Epoch, DBFA, and Super Epochs Algorithms

We use four data structures in our Java implementation of the strongly connected components algorithm, the Delay Free Buffer Allocation algorithm, and Algorithm 1. These are the *BAP* class, the *Epoch* class, the *Interval* class, and the *Vertex* class. The *Epoch* and *Interval* classes are inner classes of the *BAP* class, because they are not needed outside of *BAP*.

The largest class in our Java implementation is the *BAP* class. Every communication graph requires a different buffer allocation. Therefore we have a class called *BAP* (for Buffer Allocation Problem) where each object or instance of the class corresponds to a communication graph. A BAP object is created by giving the constructor an ordered list of vertices from a log file. This object will contain a representation of the communication graph as a private data member. The user can then call public methods in *BAP* that partition the graph into epochs, find the Delay Free Buffer Allocation over those epochs, compose the epochs into super epochs, and create a synchronization file.

To represent the communication graph *G*, the *BAP* class has a two dimensional array called *graph*. It is not necessary to use a canonical graph data structure because

communication graphs have a much simpler structure. See Figure 3-3 for an example of a graph with 5 process components. The first index for the array selects a process component, and the second index selects an event in that component. Each event in a component is represented by a vertex object, which contains information about the event or vertex. With a two-dimensional array the graph *G* can be traversed easily through the use of two nested loops. More importantly, an array allows for an efficient method of representing epochs.

The *Epoch* class is used to represent epochs. An epoch is a sub-graph of the entire communication graph *G*. Each epoch corresponds to a strongly connected component in *G* and a vertex in the DAG *D* (*D* is the output of the strongly connected components algorithm). An epoch object is empty when first created. Vertices are added to the epoch object through a public method. A new epoch object can also be created by calling the *compose()* method in the *BAP* class, which takes two epoch objects as parameters and returns a new epoch object containing both epochs.

The *Epoch* Class

A naïve approach to representing an epoch would be to either use another two-dimensional array or a list of vertex references. But this is an inefficient use of memory, because the number of vertices in an epoch can become quite large and results in duplicate vertex references that are already in the *graph* array. Moreover, our algorithm for creating super epochs requires an operation that composes two epochs. The complexity of the compose operation would be O(N), where N is the number of vertices

in both epochs, if an array or list is used to represent an epoch. Again, the number of compose operations that are performed in the algorithm may be large.

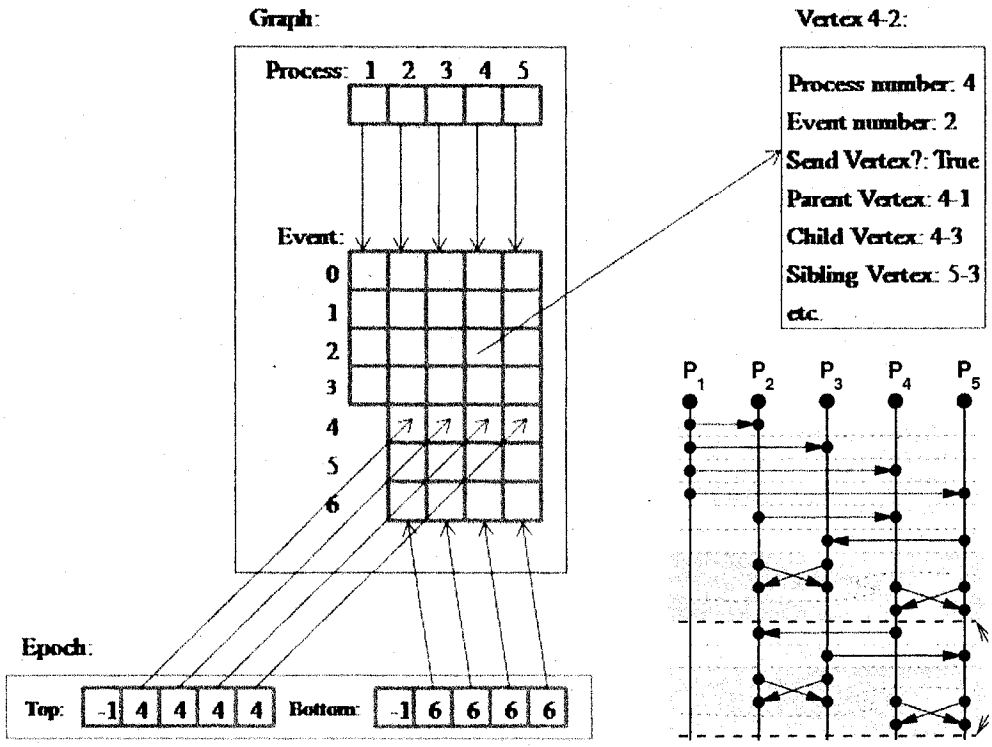

Figure 3-3: The *graph*, *epoch*, and *vertex* data structures.

In our approach, an epoch object simply stores indices into the *graph* array using two arrays called *top* and *bottom*. We make use of the following observation: If there are one or more vertices within the process component of an epoch, those vertices will be part of a consecutive sequence. This follows from the definition of an epoch. Hence, there are two non-negative integers x and y such that the event number of each vertex in the component will be between x and y, or $x <= \textit{Vertex Event Number} <= y$. So, every process component in an epoch has an upper and lower bound. Therefore, for each component in the epoch, we simply record the smallest and largest event numbers in *top*

and *bottom* respectively. The arrays *top* and *bottom* will have p entries, one for each process component in the graph. If an epoch contains no vertices within a process component, then the two entries will be -1. Since we can represent any epoch with these two arrays, the amount of memory needed for an epoch object will always be $O(p)$, regardless of the actual number of vertices that belong to an epoch. Furthermore, the compose operation for two epochs can be done in $O(p)$ time. To compose two epochs, the *top* and *bottom* arrays from both epochs are compared. Each entry in the *top* array of the new epoch will contain the minimum of the corresponding entries in the *top* arrays of the original epochs. Likewise, each entry in the *bottom* array of the new epoch will contain the maximum of the corresponding entries in the *bottom* arrays of the original epochs. Two epochs should only be composed if they are adjacent to each other in the communication graph. Since epochs are composed after being removed from the DAG D, it is guaranteed that only adjacent epochs will be composed. Adding a single vertex to an epoch requires simply comparing its event number to the entries in *top* and *bottom* for the vertex's process component, and recording the new minimum and maximum.

In addition to the two arrays *top* and *bottom*, the *Epoch* class also contains a list of adjacent epochs in D, and an integer representing the epoch's in-degree. The adjacency list and in-degree are determined during the strongly connected components algorithm, after each epoch object is created. They are later used during Algorithm 1, when they are needed to choose the next epoch to remove from the queue Z.

The *Interval* Class

The *Interval* class is used in the code for the Delay Free Buffer Allocation algorithm. Recall that in the DFBA algorithm we compute the interval between each receive vertex and the vertex that it is terminally communication dependent on. The purpose of the *Interval* class is to record the beginning and end of an interval within a process component. An interval object is created by calling the constructor with: the event-number of the first vertex in the process component (the one with the lowest event number), the event-number of the first vertex in the interval, and the event number of the last event in the interval. For the second argument, -1 can be given if the first vertex of the interval is equal to the first vertex of the process component. The interval object stores the offset between the start of the component and the start of the interval, and the offset between the start of the component and the end of the interval. These offsets are accessed through the public methods *start()* and *end()*.

The *Vertex* Class

The *Vertex* class contains all the information about a vertex within the graph G that is needed for our algorithms. This includes the event and process numbers, whether it is a send or receive vertex, etc. There are also references to the parent, child, and sibling vertices, which are useful for the parts of our code that perform depth-first searches on the graph or epochs. A depth-first search also requires a way of marking vertices that have been visited. Therefore, the *Vertex* class includes a data member called *color*, which is a special enumeration type. In order to rank and sort all vertices in the epoch building algorithm, there is an integer member called *finishTime*, which is

explained in the next section. There is also a reference to the epoch that a vertex belongs to called *epoch*, which is useful for building an epochs' adjacency list. Finally, each vertex object contains an array of integers called *vector*, which is used in the Delay Free Buffer Allocation algorithm.

Implementing the Epoch, DBFA, and Super Epochs Algorithms in Java

To create a synchronization file for an MPI program, a Java application must include our four classes. The Java application must read the log file generated when the MPI program was executed in collection mode, and create an ordered list containing vertex objects for each event listed in the log file. A BAP object is created by giving the list of vertex objects to the constructor. After the BAP object has been created, either the *creatSyncFile()* or *createCustomSyncFile()* methods can be called. The first method will create a synchronization file using the minimum buffer allocation for the program, while the second method will create a synchronization file using a custom buffer allocation if it is not less than the minimum buffer allocation. Both methods take the name of the synchronization file to be created as an argument. The second method takes an array representing the custom buffer allocation as a second argument. There is also a method called *wholeProgramDFBA()*, which computes the Delay Free Buffer Allocation for the entire program. This method was used during our tests to measure the difference in buffer requirements when using Algorithm 1.

For our prototype tool, we have written several driver files that handle user input and the creation of a BAP object. The driver files use a parser created by the JFlex [JFLX] and CUP [CUP] parser generating tools to read the log file and create a list of

vertex objects. They then create a BAP object and call either *createSyncFile()*, *createCustomSyncFile()*, or *wholeProgramDFBA()*.

In the *BAP* constructor, we first scan the input list of vertices to determine the number of process components in the graph, and the number of events in each component. We then use this information to allocate a two-dimensional array structure to contain each process component in the graph *G*. Vertices are added to the structure by scanning the input list again and using their process and event numbers to place them in the correct position.

After the two-dimensional array has been created and all vertices have been added, the parent, child, and sibling references in each vertex object must be initialized. The parent and child of a vertex are found trivially, since they precede and succeed the vertex respectively in the process component. If a vertex does not have a parent or child vertex (because it is the first or last vertex in the component), the corresponding reference is set to *null*. To find a vertex's sibling vertex we make use of the *destination* data member in the *Vertex* class. The *destination* member is the process that a send or receive event communicates with. Starting at the top of the first process component, we visit each send vertex. When we visit a send vertex, we go to the process component listed in *destination* and, starting at the top, find the first receive vertex where *sibling* is *null* and *destination* equals the process of the send vertex. This receive vertex must be the sibling of the send vertex, otherwise the MPI application would have deadlocked and never finished executing. So, the sibling references of both vertices are set to point to each other. We repeat this process for each process component, which matches each send

vertex to its corresponding receive vertex. After this is done, the *graph* data structure will be fully initialized, and the constructor will return.

The *createSyncFile()* and *createCustomSyncFile()* methods both call the private method *findEpochs()* in the *BAP* class. This method runs the strongly connected components algorithm on the communication graph and returns a list of all nodes in *D* (which represents the epochs of *G*). Both *createSyncFile()* and *createCustomSyncFile()* then call the *findSuperEpochs()* method, which takes the list of Epochs in *D* as an argument and runs Algorithm 1. The *createCustomSyncFile()* method also passes the array holding the custom buffer allocation as a second argument. The *findSuperEpochs()* method returns a list of super epochs called *L*. This list is used by *createSyncFile()* and *createCustomSyncFile()* to make the synchronization file.

The *findEpochs()* Method

The *findEpochs()* method is our implementation of the strongly connected components algorithm, which partitions the graph into epochs. To create the list of epochs in *D*, we use the algorithm for finding a graph's strongly connected components from [CLR]. First, we perform a depth first search on the graph to determine the finish time for each vertex. The finish time is the timestamp recorded in a vertex when it and all vertices connected to it have been visited. Before performing the search, we initialize the *color* member of each vertex to white. A vertex's *color* variable is used by the Depth First Search code to mark vertices that have already been visited. Next, we visit every vertex in the graph, and begin a DFS at each vertex whose color is still white. The DFS code will mark each vertex as it is visited, and will record a finish time for a vertex after

visiting it and every vertex connected to it. To visit the vertices connected to a vertex, the DFS code simply uses the child and sibling references.

When the depth first search is completed, every vertex will have a finish time associated with it. We then reinitialize the color of every vertex to white, and place them all in a list. The vertices are then sorted from highest finish time to lowest. Starting at the first vertex in the list, we begin a DFS at each vertex whose color is still white.

The code for this second depth first search behaves slightly differently. Before a new DFS is begun, an epoch object is created. As new vertices are visited, they will be added to the epoch object. When a vertex is added to the epoch, the *epoch* member in the vertex object is set to point to that epoch. After a DFS is finished, the epoch will be added to a list. The other difference is that the DFS code will follow the parent and sibling references instead of the child and sibling references. This is equivalent to performing a depth first search on the transpose of the communication graph, which is what the strongly connected components algorithm calls for.

We do not use a recursive implementation of depth first search, since the number of recursive calls can become large. Instead we use an iterative stack-based implementation. When a vertex is first visited, it will be placed on a stack. It is later removed from the stack when it, and all vertices connected to it, have been visited. This approach avoids the overhead associated with recursive calls.

After all epochs have been added to a list, the adjacency list with respect to D must be built for each one. To create an epochs' adjacency list, we visit the vertices that immediately follow the end of each process component in the epoch. When we visit each of these vertices, we add the epoch containing it (by checking the *epoch* reference in the

vertex object) to the adjacency list if it has not been added already. We use a hash table to keep track of which epochs have already been added. As each epoch is added to the list, its in-degree is incremented by one. Afterwards, the list of epochs (which holds all the nodes in the DAG D) is returned. The total runtime of the method is O $(V + E)$, where V is the set of vertices and E is the set of arcs between vertices.

The *findSuperEpochs()* Method

The *findSuperEpochs()* method implements Algorithm 1. First, β is computed over all epochs using the method *DFBA()* in *BAP*. If the calling method supplied a custom buffer allocation in the second argument, then it is compared to β. We set β equal to the custom buffer allocation if it is greater than or equal to β for every process. Otherwise an error message is printed and the method returns prematurely.

Next, we use β to build the list of super epochs L, as described in Algorithm 1. Before the main loop of the algorithm, we add every epoch in D to a hash table. By doing this, we can test if an epoch is in the set and remove it in constant time. For the list Z, we use a linked list. This allows for epochs to be added and removed from Z in constant time. The rest of the implementation of Algorithm 1 is a straight forward application of the *DFBA()* and *compose()* methods in the *BAP* class. The list L of super epochs is returned at the end of the method. The total runtime of the method is O $(|V|^2)$.

The *DFBA()* Method

The *DFBA()* method implements the Delay Free Buffer Allocation algorithm. It takes an epoch as input and returns the minimum buffer allocation for that epoch in an

array. To determine the minimum buffer allocation, we first find the terminal communication dependency of each receive vertex. This is done using the dynamic programming algorithm from DFBA, which requires another depth first search. For each vertex in the epoch, we initialize the *color* and *vector* members. We then perform the depth first search, starting at the end of each process component. This search visits the parent and sibling vertices of a vertex. After visiting all vertices connected to a vertex, *vector* is computed by taking the element wise maximum of the *vector* objects in the parent and sibling vertices.

To determine the maximum overlap density (the maximum number of buffers required at any point) for a process component, we first allocate an array that has an entry for each event in the component, and set each entry to zero. We then create an interval object for each receive vertex in the component and add it to a list. An interval object is created by giving the constructor the event number of the first vertex in the process component, the event-number of the first vertex in the interval, and the event number of the receive vertex. The event number of the start of the interval is found by checking the *vector* object in the receive vertex's sibling vertex.

Finally, we use the list of intervals with the array we allocated earlier. For each interval in the list, we increment the elements in the array between the indices returned by the interval's *start* and *end* methods by one. The maximum overlap density can then be found by finding the maximum element in the array. This procedure is repeated for each process component, and the minimum buffer allocation for the epoch is returned at the end. The total runtime of the method is O ($|V_E| \log |V_E|$), where V_E is the set of all vertices within the epoch.

The *wholeProgramDFBA()* method runs the Delay Free Buffer Allocation algorithm on the entire communication graph. To do this, we place every vertex within the graph in an epoch object. We then simply pass this epoch to the *DFBA()* method, and return the array given by the *DFBA()* method.

Summary of Commands used for the Barrier Synchronization Tool

We give a brief summary of the commands used to run an MPI application in collection mode, analyze the log file, and run in synchronization mode using the synchronization file produced as output. First, the file *bap.c* must be compiled with the MPI compiler to produce an object file called *bap.o* that can be linked with the application.

```
$mpicc -c bap.c
```

The object file *bap.o* should be placed in the same directory as the application, and the application should be recompiled and linked with *bap.o*.

```
$mpicc -DBAP -c mpiApp.c
$mpicc -o mpiApp mpiApp.o bap.o -DBAP
```

The application must be executed with the *collect* flag, and the name of the log file used to record every send and receive event must be specified.

```
$mpirun -np 4 mpiAPP [mpiApp args] -BAP collect logFile.txt
```

This will create a file for each process, containing that processes' message events. Each file will be called *logFile.txt-i*, where *i* is the process id. These files must be combined into one log file using a shell script. The argument to the script is the filename used with the *collect* flag.

```
$./combine logFile.txt
```

There will now be one file called *logFile.txt* containing the communication information for every process. This file is the first argument to the Java analysis program. The second argument is the name of the synchronization file to that will be created.

```
$./bap logFile.txt syncFile.txt
```

This command creates a synchronization file using the minimal buffer allocation. To use a custom buffer allocation, a third argument is given specifying the number of buffers to allocate to each process.

```
$./bap logFile.txt syncFile.txt 4
```

If the number of buffers supplied is less than the minimal buffer allocation necessary, an error will be returned.

To use the synchronization file, the MPI application must be executed using the *sync* or *psync* flags, and the name of the synchronization file must be given. The *sync* flag will run the application in full barrier synchronization mode, while the *psync* flag will run it in partial barrier synchronization mode. The application should only be run with the same number of processors and the same problem size used in collection mode.

```
$mpirun -np 4 mpiApp [mpiApp args] -BAP sync syncFile.txt
$mpirun -np 4 mpiApp [mpiApp args] -BAP psync syncFile.txt
```

CHAPTER 4

RESULTS

The tools that we have developed and described in the previous chapter allow a user to run an MPI application with the number of buffers needed to avoid deadlock capped at an upper bound, which is reported to the user by our tools. This upper bound is potentially lower than the one given by using the Delay Free Buffer Allocation algorithm alone. However, to use less buffer space, the MPI application must perform barrier synchronizations, which increases the application's run time. To show that this is an acceptable trade off, we tested the synchronization tool on a test suite of five different MPI applications. In this chapter, we demonstrate that our approach requires fewer buffers than the DFBA algorithm, and that the run time cost of data collection and barrier synchronization is not prohibitively expensive. We also show that the user can trade memory for execution time by increasing the buffer allocation used by the synchronization tool.

For testing, we used an 8-node Linux-based cluster with dual 3GHz hyper-threaded CPUs, each with 2 GB of memory, connected by a 1 GB Ethernet connection. Clusters such as this one are commonly used along with MPI applications to achieve parallel performance gains. All of the applications that we test utilize asynchronous message passing to increase efficiency, and thus require message buffers. Each application uses a different communication pattern, all of which are common to message

passing programs. Five applications were implemented for our test suite. These include a pipe-and-roll matrix multiplication algorithm (MM), a fast Fourier transform computation (FFT), a 2-D heat grid simulation (HG), an N-body problem solver (NBP), and a 1-D differential equation solver.

Pipe-and-roll Matrix Multiplication (MM)

This algorithm comprises one coordinator process and n worker processes that are arranged in a torus-like 2-dimensional $\sqrt{n}$ by $\sqrt{n}$ grid. The comparison proceeds in rounds. Each round consists of two parts: first, one process in each row initiates a pipe across the row, comprising $(\sqrt{n} - 1)$ messages. Second, each process sends a message to its north neighbor, resulting in an additional $\sqrt{n}$ messages per column. A total of $\sqrt{n}$ rounds are performed and in each round the initiator is the east neighbor (with wrap around) if the initiator in the preceding round. Our tests used 320 x 320 matrices with floating point entries.

Fast Fourier Transformation (FFT)

Given a vector $x = \{x_o, \ldots, x_{m-1}\}$ of size m (in our case $m = 2^{15}$), this algorithm computes the Fast Fourier Transform of x. Namely, $x' = \{x_o', \ldots., x_{m-1}'\}$, where $x_k' = \Sigma(j=0 \text{ to } m - 1)\ x_j * e^{2\pi i\,(jk\,/\,m)}$. The number of processes n should also be a power of 2 (process numbers begin at 0 in FFT). Each process is assigned m / n elements from an array. The algorithm uses a "butterfly communication pattern": Each process performs $\log n$ exchanges of its array with other processes, where the i^{th} exchange is done with the process whose id number differs only in the i^{th} most significant bit. So, for $p = 64$,

process 0 exchanges data with processes 32, 16, 8, 4, 2, 1, in that order [WA]. In total n log n exchanges take place. After log n exchanges, each process has computed the vector x'. Our tests perform the computation 2,250 times using an input vector of size 2^{15}.

2-D Heat Grid (HG)

A 2-dimensional grid is divided into n row-wise slices, each of which is assigned to a process. Each process calculates the heat distribution within its slice and communicates the boundary conditions to the processes associated with adjacent slices. The algorithm executes in rounds. In each round each process sends and receives messages from its neighbors. The first process also acts as a master and collects the results from all the processes at the end of the computation. Our tests use a grid of size 1,000 x 1,000 and ran the simulation for 1,000 rounds.

N-Body Problem (NBP)

The N-Body problem is an instance of the Long Range Interaction problem [FJLOSW]. The system consists of n processes and m elements divided equally between the processes. The goal of the computation is to compute a global sum $\Sigma(i = 0 \text{ to } m)\ \Sigma(j=0 \text{ to } m)\ f(e_i, e_j)$ by circulating chunks of size m / n around a virtual ring formed by the processes. The algorithm has $n - 1$ rounds, in which each process sends its "visiting" m / n elements onwards to the process to its right. Our tests use a problem instance of 30,000 particles.

1-D Differential Equation Solver (DES)

This algorithm arranges the n processes in a "string" each with west and east neighbors (except the end points). Each process receives m / n elements of an m-element array. Each element represents a point of the solution to a 1-dimensional differential equation. Over several rounds of computations, the solution is refined using the values of the elements from the preceding round as input to the current one. In each round a process exchanges boundary values with its neighbors, and then refines the values of the elements that it has been allocated. Further details can be found in [FJLOSW]. Our tests use an instance size of 1,000,000 elements that were refined over 1,000 rounds.

Comparison of Buffer Allocations

To confirm that our tool requires fewer buffers, we measure the buffer allocations for n processes given by the new epoch based approach (NA) and the by the Delay Free Buffer Allocation algorithm (DFBA), both shown in Table 1. For every application, the NA approach yields fewer buffers. The NA approach needs at most two buffers per process, as opposed to the O (log n) or O (n) buffers required by the DFBA approach. This is an improvement of up to factor n in the buffer requirements for every application.

The Number of Super Epochs vs. the Number of Epochs

It is also useful to measure the number of barrier synchronizations required during each application's execution. A barrier synchronization must be performed at the end of an epoch. Due to the overhead associated with barrier synchronization, we implement the super epochs approach in Algorithm 1, in order to minimize the number of epochs

and the attendant barrier synchronizations. Table 2 shows the number of super epochs used by each application, and the number of original epochs. The last column shows the improvement factor, which is the number of epochs divided by the number of super epochs.

Table 1: Buffer Allocations computed by NA and DFBA approaches for n processes.

App.	Method	Buffer Allocation (β)	Bufs. / Proc	Total Buffers
MM	NA	(0, 1, ..., 1)	1	$n-1$
	DFBA	(n, O ($\sqrt{n}$), ..., O ($\sqrt{n}$))	n	O (n ($\sqrt{n}$)
FFT	NA	(1, 1, ..., 1)	1	n
	DFBA	(O (log n), ..., O (log n))	O (log n)	O (n log n)
HG	NA	(0, 1, 2, ..., 2, 1)	2	2 (n - 2)
	DFBA	(3 ($n-1$), 6, 7, 7, ..., 7, 6)	3 ($n-1$)	$10n-9$
NBP	NA	(1, 1, ..., 1)	1	n
	DFBA	(n, ..., n)	n	n^2
DES	NA	(0, 1, 2, ..., 2, 1)	2	$2n-4$
	DFBA	($n-1$, 2, 4, ..., 4, 2)	$n-1$	$5n$ - 9

For most cases, the number of epochs is reduced considerably. In the case of the Differential Equation Solver, however, the improvement factor is negligible for every process configuration. This is due to the fact that the communication graph consists almost entirely of complex epochs, each of which becomes a super epoch when using the minimal buffer allocation. The number of super epochs could be reduced in all cases by allocating additional buffers. For example, if every process in the Differential Equation Solver had at least b buffers, the number of super epochs would be reduced by $1/b$.

The Data Collection Overhead

The data needed to construct a communication graph must be recorded at runtime in a log file. Since every event must be written to a file on disk, an application may run longer when executed in collect mode. Table 3 shows the runtimes of the applications in the test suite with and without data collection. The runtime of an application is considered to be the time elapsed between the start of the application and the time when the last process finishes executing. Runtimes in the table are taken from the minimum of ten separate runs for each application and process configuration. The last column lists the slowdown factor.

Table 2: Number of epochs and super epochs per execution for n processes.

App	n	# Epochs	# Super Epochs	Improvement Factor
MM	17	112	5	22.40
	26	200	6	33.33
	65	704	9	78.22
	101	1,300	11	118.18
	257	4,864	17	286.18
FFT	16	105,750	9,000	11.75
	32	249, 750	11,250	22.20
	64	573,750	13,500	42.50
	128	1,293,750	15,750	82.14
HG	17	2,128	2,001	1.06
	33	2,256	2,001	1.13
	65	2,512	2,001	1.26
	129	3,024	2,001	1.51
NBP	16	450	151	2.98
	32	930	311	2.99
	64	1,890	631	3.00
	128	3,810	1,271	2.98
DES	17	1,016	1,001	1.01
	33	1,032	1,001	1.03
	65	1,064	1,001	1.06
	129	1,128	1,001	1.13

Table 3: Runtime in seconds of applications for n processes with and without collection.

App.	n	Runtime with Data Collection	Std. Dev.	Runtime without Data Collection	Std. Dev.	Slowdown Factor
MM	17	16.62	5.96	17.17	5.66	0.97
	26	11.77	3.93	12.05	4.20	0.98
	65	11.62	1.37	11.41	0.94	1.02
	101	9.28	0.57	9.18	0.62	1.01
	257	11.19	0.53	11.29	0.50	0.99
FFT	16	7.24	0.04	7.17	0.03	1.01
	32	7.39	0.12	7.15	0.16	1.03
	64	8.24	0.40	8.27	0.20	1.00
	128	10.61	0.42	9.21	0.29	1.15
HG	17	18.21	0.21	11.31	3.26	1.61
	33	11.30	0.59	10.82	1.04	1.04
	65	9.04	0.82	8.82	1.03	1.03
	129	9.09	0.53	8.81	1.03	1.03
NBP	16	47.63	10.77	70.52	8.73	0.68
	32	45.38	6.18	45.24	4.65	1.00
	64	39.32	3.98	36.42	1.90	1.08
	128	37.12	2.13	34.66	2.29	1.07
DES	17	8.61	0.44	8.80	0.39	0.98
	33	6.35	0.24	6.54	0.07	0.97
	65	4.60	0.50	4.50	0.39	1.02
	129	4.18	0.11	3.96	0.22	1.06

In every case, the slowdown caused by data collection is less than 7 seconds in all of our examples. For some cases, runs with data collection took less time than runs without data collection. This implies that the variance in an application's run time is greater than the additional time needed for collection. It is likely that the low overhead is a result of the operating system's buffering and caching mechanisms, which overlap disk accesses with computation. Since data collection needs to be done only once for an application and process configuration, the runtime overhead of barrier synchronization is more important.

The Barrier Synchronization Overhead

When processes participate in a barrier synchronization, their computation and communication is delayed, adding to the application's total runtime. The more barriers used during execution, the greater the overall cost. Table 4 shows the runtimes of the applications in our test suite when using partial barrier synchronization (Pbs), full barrier synchronization (Fbs), and when using no barrier synchronization (Nbs). Again, the run times in the table are the minimum of ten runs for an application and process configuration. The table also lists the slowdown between the Nbs and Pbs modes, and the speedup factor between the Fbs and Pbs modes.

It is important to note that the runtimes of an application can vary even when using the same process configuration. This is expected due to underlying issues in the network and processor hardware. It is also important to note that adding more processes does not necessarily decrease an application's runtime, since the number of processors in the system is fixed. The purpose of these results is to measure the cost of barrier synchronization, not how well the applications scale or the performance of the hardware.

In some cases, there is no measured slowdown between the Nbs and Fbs modes. These are cases where the overhead from barrier synchronization is low enough to be within the runtime variance. Also, in the 17 process test of the Differential Equation Solver, the Fbs configuration outperforms the Nbs configuration. This may because the *MPI_barrier*() function prefetches the MPI runtime system into the cache, ensuring fewer cache misses during the communication phase of each round. This effect would only be noticeable when the number of processes is small, since it would be swamped by the cost of additional barrier synchronizations when more processes are added. The fact that the

runtime for 17 processes using the Pbs, which does not use *MPI_barrier*(), nearly matches the one for the Nbs configuration supports this hypothesis.

Table 4: Runtime in seconds of applications for *n* processes with and without barriers.

App.	*n*	Pbs (Sec.)	Std. Dev.	Fbs. (Sec.)	Std. Dev.	Nbs. (Sec.)	Std. Dev.	Nbs/Fbs Factor	Fbs/Pbs Factor
MM	17	26.29	1.95	16.67	1.57	16.65	5.81	0.63	1.00
	26	14.01	3.41	11.43	5.82	11.99	3.95	0.82	0.95
	65	11.93	0.69	13.53	1.05	11.49	1.11	1.13	1.18
	101	9.58	0.46	11.62	0.57	9.68	0.73	1.21	1.20
	257	10.62	0.29	13.35	0.31	11.04	0.67	1.26	1.21
FFT	16	12.31	0.19	11.70	0.24	7.07	0.07	0.95	1.65
	32	19.79	1.94	13.74	0.25	7.09	0.17	0.69	1.94
	64	42.92	2.42	18.75	0.26	7.80	0.32	0.44	2.40
	128	81.20	1.85	26.39	0.46	9.33	0.21	0.33	2.83
HG	17	17.73	0.64	11.99	0.83	11.28	0.02	0.68	1.06
	33	14.31	1.48	12.61	2.04	11.11	0.86	0.88	1.13
	65	15.33	3.65	13.81	0.46	8.98	0.71	0.90	1.54
	129	23.78	4.87	14.65	0.19	8.79	0.43	0.62	1.67
NBP	16	73.07	6.61	86.95	2.05	63.36	7.50	1.19	1.37
	32	48.45	3.71	51.44	1.43	42.49	5.49	1.06	1.21
	64	50.96	1.86	42.23	0.28	37.36	3.66	0.83	1.13
	128	47.76	4.95	40.59	0.84	36.35	2.11	0.85	1.12
DES	17	9.31	0.49	6.84	0.65	9.12	0.29	0.74	0.75
	33	7.24	0.60	7.18	0.44	6.29	0.20	0.99	1.14
	65	4.50	1.65	6.83	0.15	4.45	0.09	1.52	1.54
	129	9.70	1.59	7.25	0.15	3.96	0.26	0.75	1.83

In general, the slowdown factor increases with the number of processes. This is unsurprising, since the cost of barrier synchronization grows as more processes must participate. The magnitude of the slowdown varies between applications. As expected, applications with a greater number of super epochs experience larger slowdown. For example, the FFT, HG, and DES applications have more super epochs and greater slowdown than the MM and NBP applications.

A surprising result is that the Pbs configuration performs worse than the Nbs configuration in the majority of cases. We believe this is a consequence of that approach's implementation. In partial barrier synchronization, all processes must communicate with a single barrier process. When the number of processes participating in a barrier is large, this can lead to a communication bottleneck. Furthermore, when the number of processes is small, the overhead of creating and communicating with the barrier process may be greater than the cost of a full barrier synchronization.

Based on our results, barrier synchronization may decrease performance by up to a factor of 3. However, the buffer requirements are small, making this approach safer than using no barrier synchronization. Application slowdown in acceptable if the alternative is deadlock. The cost of barrier synchronization can also be mitigated by allocating more buffers when creating super epochs. Improving the implementation of the partial barrier synchronization may also help. However, there are some cases where partial barrier synchronization cannot improve on full barrier synchronization.

The purpose of partial barrier synchronization is to decrease the overall number of synchronizations performed during an application. The *synchronization count* for a process is the number of barrier synchronizations that it participates in. The synchronization count for the entire application is the sum of the synchronization count for each process. An application using full barrier synchronization will have a synchronization count of $(b - 1)\ n$, where b is the number of super epochs and n is the number of processes. For an application that uses partial barrier synchronization, the synchronization count will be at most equal to the one for full barrier synchronization,

although it is typically less. Table 5 shows the synchronization count for each configuration in the test suite.

Table 5: Full versus Partial barrier synchronization counts.

App	*n*	Sync. Count (Pbs)	Sync. Count (Fbs)	Improvement Factor
MM	17	65	68	0.96
	26	126	130	0.97
	65	513	520	0.99
	101	1,001	1,010	0.99
	257	4,097	4,112	1.00
FFT	16	143,984	143,984	1.00
	32	359,968	359,968	1.00
	64	863,936	863,936	1.00
	128	2,015,872	2,015,872	1.00
HG	17	32,001	34,000	0.94
	33	64,001	66,000	0.97
	65	128,001	130,000	0.98
	129	256,001	258,000	0.99
NBP	16	2,400	2,400	1.00
	32	9,920	9,920	1.00
	64	40,320	40,320	1.00
	128	162,560	162,560	1.00
DES	17	16,001	17,000	0.94
	33	32,001	33,000	0.97
	65	64,001	65,000	0.99
	129	128,001	129,000	0.99

Increasing the Per Process Buffer Allocation

Super epochs are constructed in Algorithm 1 by finding a minimal buffer allocation that guarantees deadlock-free execution. However, an application may have more buffers available per process than the minimal buffer allocation. If these extra buffers were utilized, the number of super epochs could be reduced. This would reduce

the number of barrier synchronizations and improve the application's runtime performance.

To verify this, we experimented with three of our applications by increasing the buffer allocation used to create super epochs. For each application we used the configuration with the highest number of processes, and hence the largest number of barrier synchronizations. Table 6 shows the results of using more buffers for the MM, FFT, and HG applications. The buffer allocations used are listed, along with the resulting number of super epochs and the run time.

Table 6: Performance of various configurations using larger buffer allocations.

App.	Buffer Allocation	Super Epochs (number of)	Pbs time (seconds)	Fbs time (seconds)
MM (257)	(0, 1, 1, ..., 1)	17	10.62	13.35
	(5, 5, 5, ..., 5)	7	10.91	12.37
	(10, 10, 10, ..., 10)	5	10.84	12.21
FFT (128)	(1, 1, 1, ..., 1)	15,750	81.20	26.39
	(5, 5, 5, ..., 5)	3,375	20.88	13.22
HG (129)	(0, 1, 2, ..., 2, 1)	2,001	23.78	14.65
	(4, 4, 4, ..., 4)	2	8.68	8.64

Our results confirm that allocating more buffers reduces the number of super epochs in an application, and thus improves the runtime performance. For the Heat Grid (HG) simulation, a small increase in the number of buffers dramatically reduced the number of super epochs, leading to a lower run time also. This is because the communication pattern of HG resembles the one in Figure 4-1. If the minimal number of buffers is allocated, each complex epoch becomes a single super epoch. However, if the number of buffers is slightly increased, each complex epoch can be composed into a

single super epoch. This is an example of a small additional allocation resulting in a significant performance improvement.

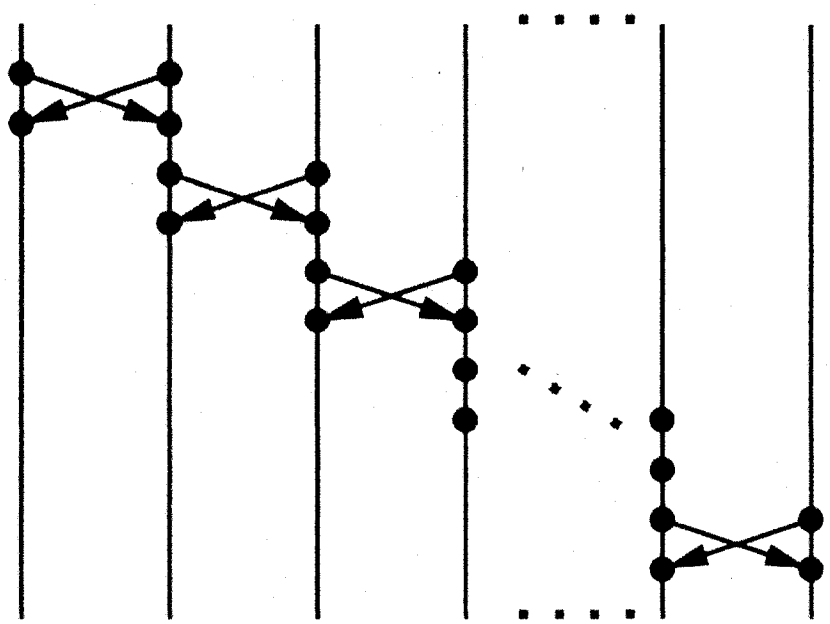

Figure 4-1: Applications with similar communication patterns benefit from additional buffers.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we presented a tool that limits the number of message buffers needed to avoid deadlock in MPI applications. This tool separates the execution of an MPI application into separate periods called epochs, by recording and analyzing the communication pattern of the application. Available buffers are reusable during each epoch. Our tests confirm that using this tool decreases the buffer requirements of MPI applications, at the cost of a constant increase at most in runtime. We also confirmed that additional message buffers can be traded for faster execution time. Limiting the buffer requirements of an MPI application makes it easier to port it between systems.

Future Work

The complexity of the analysis phase is dominated by the Delay Free Buffer Allocation (DFBA) algorithm, which is run many times, proportionate to the number of epochs in the communication graph. Every time the algorithm executes, data structures used in the previous execution have to be rebuilt. This work is redundant if the same epochs were present in the last execution. It may be possible to improve the run time of the DFBA algorithm by using auxiliary data structures to record previous computations.

We also believe that it is possible to improve the runtime of MPI applications that use epochs. This can be done by improving the implementation of the barrier synchronization used at the end of each epoch. The partial barrier synchronization scheme currently uses one process to coordinate the synchronization with other processes, leading to a communication bottleneck. A distributed implementation of the partial barrier can alleviate this problem.

Finally, the MPI tools from this thesis can be integrated into sophisticated debugging programs for message passing applications. The debugging program can automate the data collection and analysis, which presently must be done via several steps on the command line. The code for analysis is in our object-oriented Java classes, and is available for future programs. Our work is also applicable to other message passing libraries and languages that rely on asynchronous communication.

REFERENCES

[ANA] V. Anantharam. The optimal buffer allocation problem. *IEEE Transactions on Information Theory 35 (4)*, pages 721-725, July 1989.

[BDHRS] J. Bruck, D. Dolev, C. Ho, M. Rosu, and R. Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64-73, July 1995.

[BPW] A. Brodsky, J. Pedersen, and A. Wagner. On the complexity of buffer allocation in message passing systems. *Journal of Parallel and Distributed Computing 65*, pages 692-713, March 2005.

[CLR] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 1991.

[CUP] CUP LALR Parser Generator for Java. *http://www2.cs.tum.edu/projects/cup/*

[FJLOSW] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors-General Techniques and Regular Problems*, volume 1. Prentice-Hall, 1988.

[JFLX] JFlex – The Fast Scanner Generator for Java. *http://jflex.de*

[LAM] L. Lamport. Time, clocks and the orderings of events in a distributed system. *Communications of the ACM 21 (7)*, pages 558-565, July 1978.

[PBS] J. Pedersen, A. Brodsky, J. Sampson. Approximating the Buffer Allocation Problem Using Epochs. Under review for the *Journal of Parallel and Distributed Computing*, submitted October 2007.

[REI] M. Reiman. The optimal buffer allocation problem in light traffic. *Proceedings of the 26th IEEE Conference on Decision and Control*, pages 1499-1503, December 1987.

[SHE] T. Sheskin. Allocation of interstage storage along an automatic production line. *American Institute of Industrial Engineers Transactions 8 (1)*, pages 146-152, March 1976.

[WA] B. Wilkinson and M. Allen. *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, 2nd edition, 2005.

VITA

Graduate College
University of Nevada, Las Vegas

Jeffrey Sampson

Home Address
4910 Von Leidner Street
Las Vegas, Nevada 89149

Degrees:
Bachelor of Science, Computer Science, 2003
University of Texas, Austin, Texas

Publications:
Ju-Yeon Jo, Yoohwan Kim, and Jeffrey Sampson. A Spam Mail blocking scheme with puzzles and tokens. *Network/Computer Security Workshop*, Bethlehem, PA, August 2005.

Thesis Title: Buffer Allocation in Message Passing Systems: An Implementation for MPI

Thesis Examination Committee:
Chairperson, Dr. Jan B. Pedersen, Ph. D.
Committee Member, Dr. Evangelos Yfantis, Ph. D.
Committee Member, Dr. Renee Bryce, Ph. D.
Graduate Faculty Representative, Dr. Jacimaria Batista, Ph. D.