Approximating the Buffer Allocation Problem Using Epochs

Jan Bækgaard Pedersen\textsuperscript{a}, Alex Brodsky\textsuperscript{b}, Jeffrey Sampson\textsuperscript{a}

\textsuperscript{a}School of Computer Science, University of Nevada, Las Vegas, 4505 Maryland Parkway, Las Vegas, Nevada, 89154-4019, U.S.A.
\textsuperscript{b}Department of Applied Computer Science, University of Winnipeg, 515 Portage Ave., Winnipeg, MB, R3B 2E9, Canada

Abstract

The correctness of applications that perform asynchronous message passing typically relies on the underlying hardware having a sufficient amount of memory (message buffers) to hold all undelivered messages—such applications may deadlock when executed on a system with an insufficient number of message buffers. Thus, determining the minimum number of buffers that an application needs to prevent deadlock is an important task when writing portable parallel applications. Unfortunately, both this problem (called the Buffer Allocation Problem) and the simpler problem of determining whether an application may deadlock for a given number of available message buffers are intractable [5].

We present a new epoch-based polynomial-time approach for approximating a solution to the Buffer Allocation Problem. Our approach partitions application executions into epochs and intersperses barrier synchronizations between them, thus limiting the number of message buffers necessary to ensure deadlock-freedom. This approach produces near optimal solutions for many common cases and can be adapted to guide application modifications that ensure deadlock-freedom when the application is executed on different systems. We describe a prototype implementation and present an empirical evaluation of this approach. Lastly, we describe a space-time trade-off between the number of available message buffers and the number of barrier synchronizations, and describe how this trade-off can be used to fine-tune application performance.

Key words: Message passing systems, Buffer allocation, Complexity, Parallel and distributed programming.

1. Introduction

As the price of computing power continues to decrease and connectivity continues to grow, cluster-based parallel applications continue to proliferate, and are now being used on many different platforms. A cluster-based parallel application comprises a number of asynchronous processes, each running on separate processors, that communicate with each other over the cluster’s interconnect to collaboratively solve a problem.

Typically, such applications use a point-to-point message passing communication model in which data are sent from one process to another via a standard message passing library. These libraries, such as MPI [11], manage the logistics of message passing and mask the differences between various cluster interconnects. This makes the applications easier to develop, and more importantly, ensures that an application written on the “development” system will execute on the “production” system.\textsuperscript{1} Unfortunately, these libraries can not completely mask the differences between the system on which an application was developed and the system on which it will be executed. Thus, these libraries cannot, by themselves, ensure that an application will run correctly on the target system.

Typical applications use a blocking communication [11, Section 3.4] that blocks until the message being sent is injected into the system. Namely, the message is either delivered to the receiver or stored in the system until it can be delivered. In the latter case, the message is stored in specially allocated memory, called a message buffer. Thus, the behaviour of the communication depends on the availability of message buffers: if no buffers are available, the send operation blocks until the message can be delivered to the receiver, otherwise, if buffers are available, the send blocks only until the message can be copied into a buffer. In the former case the communication is synchronous, and in the latter case, it is asynchronous. The latter is typically desirable since it facilitates overlapping computation and communication, yielding more efficient parallel applications.

Unfortunately, if the application is developed on a system with many buffers, its design may assume that blocking communication always behaves asynchronously. If the application is executed on a system with fewer message buffers, and the communications cease to be asynchronous, deadlock can ensue. That is, the application will deadlock due to lack of buffers. Although adding more memory to the machines themselves is

\textsuperscript{1} For example, the authors were granted access to a large production cluster to perform their evaluations, when it was not in use for other projects.

Email addresses: matt@cs.unlv.edu (Jan Bækgaard Pedersen), abrodsky@acs.uwinnipeg.ca (Alex Brodsky), jsampson@cs.unlv.edu (Jeffrey Sampson).
one way to rectify this problem. In many cases it may not be financially or physically feasible to do so. Alternatively, rewriting the application could prove to be equally prohibitive, depending on the application’s size, complexity and code quality.

Furthermore, in many cases it is not the speed but the size of the computation that is of issue, e.g., computations involving very large data sets. Consequently, if the majority of the memory is used by the application and the system cannot allocate additional buffers, deadlock can ensue. Since it is relatively easy for an application to determine the amount of memory available to it, it can ensure that it leaves sufficient memory free for buffers, provided it knows how many will be required.

Hence, it is necessary to determine the number of buffers needed to prevent such deadlock. Furthermore, if the production system does not allocate a sufficient number of buffers, the application must be modified to deal with the reduced number of available buffers. However, determining the number of buffers that an application needs (solving the Buffer Allocation Problem) is an intractable optimization problem [5], and modifying a complex parallel application is a challenging software engineering task. This paper describes how to mitigate these challenges.

First, we describe an integrated approach that determines a good upper-bound on the number of buffers that an application needs and then automatically adjusts the execution of the application to ensure that the specified bounds are not exceeded, thus avoiding deadlock. The approach is based on the observation that if an application performs a barrier synchronization, no message buffers will be in use immediately after the barrier, assuming no communication may begin before and end after a barrier. Thus, if the execution of the application is partitioned into epochs, and after each epoch a barrier operation is performed, we need only ensure that each epoch in the execution has a sufficient number of buffers—an easier task than ensuring that the application as a whole has enough buffers.

Our approach partitions an application’s execution into epochs and computes the minimum number of buffers needed for each epoch. To minimize the use of barriers, small epochs are composed into large epochs, called super-epochs. Barriers are automatically interspersed between the super-epochs by instrumenting the underlying communication library that tracks when an epoch ends and obviates the need for modifications to the application itself. If the target system can allocate more than the minimum number of buffers, bigger super-epochs are composed, resulting in fewer barrier synchronizations.

Second, we describe how a developer can use our approach to adjust how many buffers an application uses without having to modify the application source code. By increasing or decreasing the size of the epochs, a developer can increase or decrease the number of buffers that are necessary. As the size of the epochs increases, the number of barriers decreases. Thus, the developer can trade-off the number of buffers being used against execution speed. Furthermore, our approach facilitates the safe execution of multiple parallel applications on the same cluster by ensuring that all applications have a sufficient number of buffers available.

Third, we describe the implementation of a prototype system that we implemented and provide an empirical evaluation of our approach. Our implementation uses either the LAM [7,23] or the MPICH [3] MPI library as a base and instruments its communication primitives to automate the generation of communication graphs as well as to track epochs. Thus, any developer that wants to use our system needs only to recompile their application. The modifications themselves are relatively general and do not require extensive modification of the library code. In fact, the entire mechanism is implemented with the use of function wrappers. Furthermore, the mechanisms used to track epochs are completely local and do not require any out-of-band communication.

Our empirical analysis is based on several parallel applications that use a variety of computation paradigms. For each application we compute the buffer assignments using existing methods and compare them to the buffer assignment determined by our prototype. We show that in most cases our approach yields tighter approximations and never does worse than existing methods. Our results indicate that while our approach does affect performance, in many cases the effect is tolerable, and in some cases, negligible. Additionally, since barrier synchronizations do play an important role in our approach, we briefly investigate their cost and conclude that the cost of performing barrier synchronizations depends greatly on the application.

The paper proceeds as follows: The next section presents the related work and provides a context for our results. Section 3 describes the computation and communications models, defines the problem, and states definitions used throughout the paper. Section 4 presents our approach, first providing an overview, and then detailing each of its parts. Next, Section 5 describes the implementation of our prototype, and Section 6 describes our empirical evaluation of our new approach. Section 7 describes the trade-off between the number of available message buffers in a system and the number of barrier synchronizations in the course of an execution, and how this trade-off can be used by developers to fine-tune their applications. Lastly, Section 8 summarizes and discusses future directions.

2. Related Work

Our approach builds on the work by Brodsky et al. [5]. Consequently, this section focuses on work that is directly related to our new approach. Following a brief discussion of work related to communication models and program safety, we describe and discuss the work of Brodsky et al. [5]: the graph framework for modeling the execution of a parallel program, and the intractability of determining whether a program is safe for a fixed number of message buffers. Next, we summarize a related problem, a polynomial-time algorithm that solves it,

\[2^{A \text{ barrier can be implemented using a series of simple point-to-point communications.}}\]

\[3^{\text{Since barrier synchronizations are expensive, they tend to slow down the execution of an application.}}\]
and how this algorithm can yield an approximate solution to the former problem. Lastly, we discuss scheduling and its relationship to our new approach.

2.1. Communication and Safety

Point-to-point communication primitives are typically classified based on their behaviour. For example, MPI [22, Section 3.4] divides the primitives into two categories: blocking and nonblocking, and divides each category into four modes: synchronous, buffered, ready, and standard. Blocking primitives return once they have completed their function or an error has occurred. Nonblocking primitives return immediately, and require the program to poll the system, using MPI_Test, to determine if the operation has completed.

<table>
<thead>
<tr>
<th>Modes</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>Blocking</td>
</tr>
<tr>
<td></td>
<td>MPI_Send</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
</tr>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
</tr>
</tbody>
</table>

Synchronous mode primitives are buffer-independent; the buffer in which the message is passed should not be modified until the message is delivered and the operation completes. Buffered mode primitives may complete before the actual message is delivered by buffering the message. However, the user is responsible for allocating buffers, a potentially error-prone task. Namely, the memory that contains the data being sent cannot be reused by the user until the communication completes—the primitive MPI_Test is used to determine if a communication has completed. Implicitly, this memory becomes a buffer and it is up to the application to keep track of it, not use it until communication completes, and to determine when said communication has completed. For example, the application can become responsible for managing buffers either by using the nonblocking primitives MPI_I*, or by explicitly providing its own buffers using the buffered mode primitives, which require the user to explicitly allocate a buffer before they are used.

Ready mode primitives return an error if the corresponding receive is not already posted. Standard mode primitives are either synchronous or have system-managed buffers: If no buffers are available, the primitive is synchronous, that is, it does not complete until the corresponding receive occurs. However, if buffers are available, the primitive is asynchronous, that is, it can complete before the message is delivered to the receiver.

The blocking standard mode primitive is the one most commonly used in MPI programs and corresponds to the primitive assumed by our model. This primitive facilitates the overlap of computation and communication in parallel programs, improving their efficiency when buffers are available, and creating potential for deadlock when buffers are exhausted. MPI’s blocking receive (MPI_Recv) blocks until the message is delivered, just like the receive primitive assumed by our model.

In many instances, when dealing with such bimodal primitives, that is, ones that behave synchronously or asynchronously depending on the availability of message buffers, it is necessary to distinguish between four events: (i) start of a send, (ii) message injection (send completes), (iii) start of a receive, and (iv) message delivery (receive completes). This is similar to the decomposition of Cypher and Leu [9,10] who divided the send operation into a POST-SEND and WAIT-FOR-BUFFER-RELEASE, corresponding to the send’s start and completion, and divided the receive operation into a POST-RECEIVE and WAIT-FOR-RECEIVE-TO-BE-MATCHED, corresponding to the receive’s start and completion.

A process in a parallel program that starts a send operation but cannot inject the message into the system becomes deadlocked, implying that the corresponding program is not safe. The MPI standard [22, Section 9.2] defines a program as safe and portable, if the program requires no message buffers, that is, all point-to-point communication primitives can be replaced with synchronous mode primitives. Bruck et al. [6] introduced a more general notion of safety, called k-buffer correctness: a program is k-buffer correct if k message buffers per process is sufficient to prevent deadlock. A very general notion of safety, defined by Brodsky et al. [5], is k-safety. A program is k-safe if the total number of message buffers that it needs to ensure deadlock-freedom is at most k.

Our focus is on the blocking standard mode primitive because k-safety is not applicable to the other primitives. That is, the nonblocking primitives (MPI_I*send) will never block, while the remaining synchronous mode primitives either work with user supplied buffers, or explicitly block until the message is received. In all but the last case, the user, becomes responsible for buffer management. Although this reduces the potential for deadlocks, it exposes the user to the complexity and pitfalls of buffer management.

2.2. Determining k-Safety

Determining the k-safety of a program, necessitates a representation of its execution. Parallel program executions are typically represented using directed graphs where vertices represent states or events in the program and arcs represents transitions, computations, or communications between them. Communication graphs, which can be traced to Lamport’s time-space diagrams [19], are one such representation where vertices represent receive, send, and internal events, and the arcs represent computations between two consecutive events in the same process, or communications between two corresponding events (send and receive) in different processes. Brodsky et al. [5] do not use vertices to represent internal events, implicitly representing them by computation arcs, because their focus is on preventing communication oriented deadlock. They extend the communication graphs by assigning a colour state to the vertices, representing the states within send and receive operations, and by associating tokens with communication arcs to represent allocated message buffers.

Using the communication graph framework, Brodsky et
al. [5] investigated the complexity of determining the $k$-safety of a parallel program—they defined this as the Buffer Allocation Problem (BAP). Via a colouring game, played on the communication graphs, Brodsky et al. showed that solving BAP is NP-hard. In fact, even determining if a given buffer allocation is sufficient to ensure deadlock-freedom, which they called the Buffer Sufficiency Problem (BSP), was shown to be coNP-complete. In both cases, the proof was performed by reducing 3-SAT and Tautology to BAP and BSP, respectively. Consequently, unless P = NP, approximation techniques are the only tractable option for solving general instances of BAP and BSP, and is a primary motivation of the work presented in this paper.

In addition to BAP and BSP, Brodsky et al. investigated the complexity of a slightly different task: that of determining the size of a buffer allocation necessary to ensure a delay-free execution—they defined this as the Delay-Free Buffer Allocation Problem (DBAP). As mentioned previously, standard send operations behave asynchronously if message buffers are available: they do not block until the message is delivered, and hence do not delay the program. If each send operation is guaranteed to have a buffer available, the program's execution will be delay-free. Brodsky et al. showed that this problem has a polynomial-time solution by exhibiting a Delay-Free Buffer Allocation (DFBA) algorithm, which is summarized in Section 4.3. Since any delay-free execution will also be deadlock-free, the DFBA algorithm yields an approximation to the BAP. The DFBA algorithm is a key part of our new approximation.

2.3. Buffer Allocation and Deadlock Prevention

There has been significant investigation of buffer allocation in the area of networks [21,20,2,4,12,26]. In fact, the Operations Research community addresses very similar problems in settings dealing with assembly lines and production, and queuing theory considers the issues of systems with multiple fixed size queues. However, in all of these cases the setting assumes a dynamic rather than a static model. Specifically, message arrival is typically modeled probabilistically, and the number of messages sent to a particular process is not known or fixed a priori. Furthermore, in some cases, such as store and forward networks, packet loss, while undesirable, is acceptable, and allocations are typically computed on-line.

In contrast the setting for the problem we are addressing is static. The communication pattern is fixed and known a priori, and the buffer allocation is computed off-line. However, in our case, deadlock or lost messages are not an option. The user assumes a reliable deadlock-free system with no mechanism for backing out of a deadlock situation.

Our techniques and analysis are more closely related to Petri nets and colouring games [16,17] where systems are modeled by graphs whose nodes can contain tokens, which are transferred to other nodes along the arcs of the graph according to certain logic rules. In order to determine deadlock-freedom in coloured Petri nets, an NP-complete task, an occurrence graph is constructed—an occurrence graph represents all possible markings, that is, all possible configurations of the Petrinet, and strongly connected components are computed. If a strongly connected component exist, which does not have any arcs leaving it, then there is a configuration that represents either a livelock or a deadlock. Techniques for reducing the size of the occurrence graph do exist, but never the less, the size of the graphs and the time to evaluate them can both be exponential.

2.4. Barriers and their Implementations

Barrier synchronizations are an important part of our approach. Consequently, we briefly discuss their implementation here. In MPI the MPI_Barrier is implemented using the Butterfly or Pair Exchange (PE) [18,24] algorithm. This can easily be visualized by letting each of the $n$ processes be represented by a vertex in a hypercube of dimension $\log n$. The barrier is performed in $\log n$ stages: in the first stage each process synchronizes with their neighbour along the first dimension of the hypercube, in the second stage each process synchronizes with their neighbour along the second dimension, and so on. By the end of the last synchronization, each process has synchronized with every other process, albeit indirectly, in the hypercube. This algorithm evenly distributes the communication among all the processes, takes $O(\log n)$ steps and performs $O(n \log n)$ communications.

An alternative, simpler, algorithm would be for all processes to send to a single “leader” process and wait for a response. Once the “leader” process receives from all other processes, it sends back a reply to the waiting processes, allowing them to proceed. Although this algorithm is simpler, and uses less communication, it is slower because the “leader” process can only receive or send one message at a time. Consequently, the time-complexity of this algorithm is $O(n)$, but in practice is a lot worse because the communication flood to the “leader” process slows down the entire interconnect. As part of our investigation we looked at the efficiency of MPI’s barrier implementation and report on it in Section 6.

A third possibility is to use the multicast or broadcast capabilities of the underlying network. However, the most commonly used MPI implementations MPICH [3] and LAM[7,23] build their MPI_Barrier primitive using the MPI send and receive primitives.

2.5. Scheduling

One component of our approach is related to block scheduling in the code generation phase of a compiler [1, Section 9.8]. To minimize the buffer allocation of a parallel program execution we partition the execution into epochs and schedule the epochs such that only a minimal number of buffers is needed. This is similar to scheduling blocks of code, which have causal dependencies and must use a limited number of resources such as registers. Similar to block scheduling, we use a directed acyclic graph (DAG) to represent the causal order of the epochs and then schedule them using a hierarchical greedy algorithm. Similar algorithms are also used for scheduling instructions on
Very Long Instruction Word (VLIW) architectures [13] where many small subinstructions that are causally independent can be scheduled to execute in parallel. We note that this is not an instance of one of many scheduling problems, which are known to be NP-complete [15].

3. Preliminaries

In this section we describe the computation model, define the problem of interest, and define our terminology.

3.1. The Model

Our process model assumes the properties present in cluster-based parallel applications, such as those written with MPI [11]. We assume that there are \( n \) processes that run on separate or shared processor elements. These processes run asynchronously, meaning that they run on variable speed processor elements and do not have access to a global clock, implying that no inference can be made about the state of a process based solely on how long it has been running, that is, there is no global time.

Similarly, our communication model assumes the properties of typical applications that use libraries such as MPI. Namely, processes communicate via point-to-point message passing, which may or may not be synchronous, depending on the state of the application. Specifically, a process sends a message to another process by performing a send that blocks until the message is injected into the underlying communication system. Injection occurs only if the destination process is ready to receive the message or if the message can be buffered at the destination processor—we call this receiver-side buffering [5]. That is, if the destination process is not ready to receive the message and a buffer is available, an asynchronous communication can ensue. A buffer is an allocated piece of memory that is used to store one message at the destination, until it is delivered to the designated process. For simplicity, we assume that all messages have a fixed size and that one buffer can hold exactly one message—messages of greater size can be divided into fixed-size segments. We explore this possibility in Appendix A.

A process receives a message from another process by performing a receive, which removes (and delivers) the earliest available message sent by the specified process to the receiving one. If no such message is present, the receive blocks until such a message is injected into the system.

For reasons first described in [5], we require that the receive operations explicitly specify the source of the message: we disallow any kind of receive-any mechanism. This is because our analysis relies on the order of the communications within the application. If a process uses a receive-any mechanism, then, the communication order can change from execution to execution, invalidating the analysis. The Future Work section describes a straightforward extension to our approach that would relax the restriction against use of a receive-any mechanism.

We call send and receive operations communication events, where each event is defined by the type of operation, the source, and the destination of the message. A point-to-point communication from one process to another involves the execution of a send event by the source process and a corresponding receive event by the destination process.

Processes synchronize with each other via point-to-point communication or a barrier \(^4\), which is an all-process synchronization that can be implemented using point-to-point communication. Several different implementations are described in Section 2.4.

For conciseness, we call a parallel application a program. A program execution is a causal ordering of all events in the \( n \) process executions. We assume that the causal ordering of the communication events remains static. Namely, that the order of events is oblivious to the problem instance being computed and hence is repeatable [9]. For example, if during a computation the third event of process \( i \) was to send a message to process \( j \), then the third event of process \( i \) always sends a message to process \( j \), regardless of the problem instance.

3.2. Problem Definition

In our model, a communication can be performed asynchronously if the receiver has an available buffer to hold the message. However, if the receiver has no available buffers, the sender is blocked until the receiver performs the receive operation or a buffer becomes available. In the former case, the communication becomes synchronous. If what was assumed to be an asynchronous communication becomes synchronous, deadlock can ensue.

For example, suppose two processes simultaneously send and then receive from each other, as diagrammed in Figure 1. Unless one of the communications remains asynchronous, both processes will deadlock, waiting to complete their send operations before receiving each other's message. Thus, each process needs to be allocated a sufficient number of buffers to prevent this form of deadlock.

The problem is to determine, for each process, the minimum number of buffers necessary to ensure a deadlock-free execution. This is the optimization version of the Buffer Allocation Problem (BAP), which was defined by Brodsky et al. [5]. Namely, determining a minimal deadlock-free buffer allocation for a given program.

3.3. Definitions

Given a program comprising \( n \) processes, a buffer allocation (also called a buffer assignment) is an \( n\)-tuple \( \beta = (\beta_1, \ldots, \beta_n) \) of integers representing the number of buffers that are allocated to each respective process. A deadlock-free allocation ensures that the program will not deadlock due to lack of buffers. A delay-free allocation ensures that no send operation blocks during an execution. That is, in all communications, the message can be immediately injected into the system because either the

\(^4\) This primitive is implemented by MPI.
corresponding receive operation has begun, or there is an available buffer for storing the message.

A program trace (or just trace) is the physical manifestation (a log) of events that occurred during an execution. An event is either the start of a process, the completion of a send operation, or the completion of a receive operation. A send completes when the sender ceases to be blocked, rather than when the message is delivered. A communication graph, which is derived from a program trace, is an abstract representation of these events used to model the operation of the application.

A communication graph of a program trace \( S \) is a directed acyclic 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 \( v_{i,c} \), \( 1 \leq c \leq e_i \), represents either a send or a receive event. The former is called a start vertex and the latter are called send and receive vertices, respectively.

The arc set \( A \) consists of two disjoint arc sets: the computation arc set \( P \) and the communication arc set \( C \). A computation arc, \( (v_{i,c}, v_{i,c+1}, c) \in P \), \( 0 \leq c < e_i \), represents a computation within process \( i \), which is an “internal event” in the terminology of Lamport [19]. A communication arc \( (v_{i,c}, v_{j,d}, c) \in C \) 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 1). Note, the process arcs are drawn without orientation for clarity; they are always oriented downwards.

The \( i \text{th} \) process component \( G_i \) of \( G \) is the subgraph \( G_i = (V_i, A_i) \) where \( V_i = \{ v_{i,c} \mid 0 \leq c \leq e_i \} \) and \( A_i = \{ (v_{i,c}, v_{i,c+1}) \mid 0 \leq c < e_i \} \). Process component \( G_i \) encodes the events and their causal order of the \( i \text{th} \) process.

3.4. Ordering Events

A communication graph is an encoding of the causal ordering of all events in a trace—it is equivalent to Lamport’s space-time diagram [19] without internal events. Namely, an arc from vertex \( v_a \) to vertex \( v_b \) in the graph indicates that event \( a \) must complete before event \( b \). Transitivity, event \( a \) precedes event \( b \) if there is a path in the communication graph from vertex \( v_a \) to vertex \( v_b \). We denote this by \( a \rightarrow b \) or \( v_a \rightarrow v_b \), as context demands.

The graph does not represent the global time of completed events, and hence it is not possible to determine how far apart in time two events occur. A send event \( s \) may block, if no buffers are available, until the corresponding receive event \( r \) occurs: these events comprise a synchronization between the participating processes. That is, both the send and receive events occur simultaneously. Since no buffers are initially allocated to any of the processes, an arc from vertex \( v_s \) to vertex \( v_r \) is considered bidirectional, and we say that \( v_s \rightarrow v_r \) and \( v_r \rightarrow v_s \), which can be shortened to \( v_s \leftrightarrow v_r \).

3.5. Epochs

Informally, an epoch is a self-contained sequence of consecutive events in the trace. Formally, an epoch \( E \) in a trace \( S \) is a subsequence of \( S \) such that

(i) \( E \) contains at least one send or receive event,

(ii) if \( a \in E \) and there exists \( b \in S \) such that \( a \rightarrow b \rightarrow a \) (more concisely \( a \leftrightarrow b \)), then \( b \in E \).

(iii) if \( a, b \in E \) then \( a \leftrightarrow b \).

This implies that all epochs are disjoint (from 2) and that each epoch contains at least one send and one receive event (from 1). Consequently, a trace comprises a series of epochs \( E_1, E_2, \ldots, E_m \) such that for any two events \( a \in E_i \) 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 events \( a \in E_i \) and \( b \in E_j \) such that \( a \rightarrow b \) or \( b \rightarrow a \). In the former case \( E_i \) must precede \( E_j \) and in the latter case \( E_j \) must precede \( E_i \). Otherwise, we say that the epochs are causally unordered, that is, the epochs can be ordered either way.

Figure 2 depicts a partitioning of a trace, which is represented by a communication graph, into epochs. Observe that the ordering is not unique. For example, epochs \( E_5 \) and \( E_6 \) can be ordered either way because they are causally unordered. Similarly, the epoch pairs \( (E_7, E_8) \), \( (E_9, E_{10}) \), and \( (E_{11}, E_{12}) \) are also causally unordered and hence can be order either way.

An epoch \( E \) is represented by a same named subgraph of \( G \) that contains the vertices corresponding to the events in \( E \) and the arcs between those vertices. The subgraph is a maximal strongly connected component of \( G \), since there is a path from any vertex in the subgraph to any other vertex in the subgraph, and no vertex outside the subgraph has both a path to and from a vertex in the subgraph. This characterization is a simple proof that the epochs must be disjoint.

We use the term epoch to refer to both the event subsequence of a trace and the corresponding subgraph. Thus, we say that \( G = E_1 \circ E_2 \circ \cdots \circ E_m \) where \( \circ \) composes epochs \( E_i \) and \( E_{i+1} \) by adding a process arc between the last event vertex of the former and the first event vertex of the latter for each process component.

We say that an epoch is simple if it contains exactly one send and one receive vertex, and hence one communication arc between them. That is, the epoch is guaranteed to be deadlock-free regardless of the number of allocated buffers. An epoch is complex if it contains more than two vertices, and hence is not guaranteed to be deadlock-free without some minimal buffer allocation. In Figure 2, epochs \( E_7, E_8, E_{11}, \) and \( E_{12} \) are complex and all remaining epochs are simple.

\(^5\) An alternate justification is presented in Appendix B.
4. Main Result

Brodsky et al. [5] describe an approach that uses a program’s delay-free buffer allocation as a bound for its deadlock-free buffer allocation. The delay-free buffer allocation can be determined efficiently from the communication graph, and since delay-freedom is a stronger condition than deadlock-freedom, the latter is ensured if the former is achieved. Unfortunately, in many cases a program’s delay-free buffer allocation greatly exceeds its deadlock-free buffer allocation. A good example of this (see Figure 3) is a simple two-process program in which process \( P_1 \) sends \( k \) messages to process \( P_2 \).

In such a program, process \( P_2 \) requires \( k \) buffers to ensure delay-free execution (process \( P_2 \) may be extremely tardy). However, such a program will never deadlock, regardless of the number of available buffers. Consequently, the approximation in this case is extremely poor. In fact, for sufficiently large \( k \), that is, if many messages are sent during the execution, the approximation becomes meaningless because the number of buffers exceeds the amount of memory available to a process.

The new approach surmounts the short-comings of the delay-free approximation by decomposing the communication graph into epochs, approximating the deadlock-free buffer allocation for each epoch, and taking the element-wise maximum over these buffer allocations. In the above example, each of the \( k \) epochs contains a single send (and receive) event. Thus, the original approximation for each epoch yields an allocation of one buffer per epoch. Since all epochs are the same, the new approximation yields a deadlock-free buffer allocation of one buffer for the entire program.

This approach hinges on the assumption that all processes complete an epoch before beginning the next one, ensuring that all buffers are released and thus can be reused in the next epoch. However, this is not enforceable without requiring processes to synchronize at the end of each epoch. Consequently, to use the better approximation, the program must be modified to perform barrier synchronizations between epochs. This does not change the correctness of the program, but adds delays to the program because of the additional synchronization points. Thus, the approach trades off time for memory. Since an execution may have many epochs, interspersing barrier synchronizations between them may be prohibitively expensive. For example, if the execution represented by the communication graph in Figure 3 synchronized after every epoch, \( k - 1 \) barrier synchronizations would be performed. This is expensive, and in this case unnecessary, in order to ensure deadlock-freedom.

The key is to minimize the number of barrier synchronizations during an execution to only those that are necessary to ensure deadlock-freedom. Our approach is illustrated in Figure 4. First, we identify which epochs require a deadlock-free buffer allocation of at least one buffer. For these epochs we approximate the deadlock-free buffer allocation using the delay-free buffer allocation (DFBA) algorithm [5]. Second, we merge epochs into super-epochs and require that barrier synchronizations occur only between super-epochs. This greatly reduces, and in some cases eliminates, the number of barrier synchronizations that need to be performed. For the example in Figure 3, our approach correctly identifies that no buffers and barrier synchronizations are required to ensure deadlock-freedom.

Lastly, this approach grants the user control over the trade-off between the number of barrier synchronizations that need to be performed and the size of the deadlock-free buffer allocation. This is accomplished by adjusting the per process buffer allocations. This in turn dictates the size and hence number of super-epochs, which determines the number of barrier synchronizations that need to occur.

In the remainder of this section we describe 1) how to partition the communication graph into epochs and why interspersing synchronization points between them does not affect program correctness; 2) how to compute (or bound) the buffer allocation for each epoch; and 3) which epochs must be followed by barrier synchronizations, how epochs are merged into super-epochs, and how to determine their buffer allocations.

4.1. Partitioning the Communication Graph

First, the communication graph \( G \) is partitioned into epochs, \( E_1, \ldots, E_m \). Epochs, by definition, are maximal strongly connected components of \( G \). Consequently, computing the
maximal strongly connected components, which are by definition disjoint, yields the partition of $G$ into epochs. Computing the maximal strongly connected components yields a directed acyclic graph (DAG) in which vertices correspond to the strongly connected components, $E_i$, and the arcs to the causalities between the components. This DAG, $D$, can be computed using the standard algorithm for computing strongly connected components [8, Page 489]. Since the algorithm is linear in the number of vertices and arcs, and $G$ is a constant-degree graph, the partitioning can be performed in linear time.

Second, the epochs must be in a causal order. That is, if there is a path from a vertex in epoch $E_i$ to a vertex in epoch $E_j$, then epoch $E_i$ must precede epoch $E_j$. Conveniently, this causal order is already encoded by the arcs of the DAG $D$. A total order is derived by performing a topological sort on the DAG $D$, which is also linear in the size of DAG $D$ [8]. Once the epochs are linearly ordered, barriers can be interspersed between them.

4.2. Inserting Barriers

Ensuring that an epoch completes before the next one begins can be enforced by modifying the corresponding program to perform a barrier synchronization at the end of each epoch. This can be stated formally in the following manner: for each epoch, each process performs a barrier synchronization as the last event in the epoch. Namely, if a process performs no events in an epoch, then the process must perform a barrier event. Otherwise, if a process performs events during an epoch, then the process must perform a barrier synchronization immediately after all its other events in the epoch have occurred.

Since no event in an earlier epoch depends on an event in a later epoch, requiring the barrier synchronization does not negate the correctness of the program. Furthermore, since the application of barriers ensures that all events in an epoch complete before the next epoch is executed, all buffers can safely be reused in the next epoch. Thus, we need only compute the maximum buffer allocation over all epochs.

4.3. Bounding an Epoch’s Buffer Allocation.

To bound the buffer allocation for each epoch we to compute the delay-free buffer allocation for each epoch using the DFBA algorithm of Brodsky et al. [5]. Since delay-freedom is a stronger property than deadlock-freedom, a buffer allocation that ensures the delay-freedom also ensures deadlock-freedom.

Briefly, DFBA is polynomial-time optimization algorithm that computes the minimum number of buffers that each process needs to ensure delay-freedom: no send can ever block. A naïve approach would simply allocate one buffer for each receive that a process performs, however, in many cases buffers can be reused. For each process the DFBA algorithm determines the execution intervals during which a buffer is needed. The maximum per process overlap of these intervals is exactly
the number of buffers that the process must be allocated. The interval, say $I_r$, in which a buffer is needed ends at the respective receive event, say $r$. If the corresponding send event, say $s$, depends on another send event, say $t$, in the same process component as $r$, then the interval $I_r$ commences immediately after send event $t$ (see Figure 5). Otherwise, if no such event exists, the interval commences from the start of the process.

Each epoch $E$ can be turned into a valid communication graph $G_E$ by adding start vertices to each process component. Namely, if a process component of $E$ is nonempty, that is, it has at least one event vertex, the corresponding process component of $G_E$ has the additional start vertex with a single process arc from it to the earliest event vertex in the component. Otherwise, if a process component is empty, the corresponding process component of $G_E$ consists of a single start vertex (see Figure 6). Intuitively, the added start vertices of $G_E$ correspond to the barrier synchronization performed in the preceding epoch.

For each epoch $E_i$, $i = 1 \ldots m$, the corresponding communication graph $G_{E_i}$, is determined, and for each graph the corresponding delay-free (and hence deadlock-free) buffer allocation $\beta$ is independently computed. Each buffer allocation is valid because at the start of the epoch, after a barrier synchronization, all message buffers are free, just like at the start of an execution.

Assuming that barriers are inserted between every two consecutive epochs, computing a better buffer allocation bound for $G$ is trivial. If we ensure that $\beta_i \geq \beta_j$ for all $i = 1 \ldots m$ and $j = 1 \ldots n$, then each process is guaranteed to have at least as many buffers as it needs in each of its epochs. Thus, the element-wise maximum of the $m$ epochs’ buffer allocations, $\beta = \max\{\beta_i\}$ yields the buffer allocation for $G$.

If we were to apply this new approach to the program represented by the communication graph in Figure 3, this would yield a buffer allocation $\beta = (0, 1)$, much improved over the previous bound of $(0, k)$. Unfortunately, the price for this reduction in buffer allocation, is the requirement to perform barriers between every two epochs. Although, in some cases this is unavoidable, it clearly should not be necessary for a program such as the one represented in Figure 3. The need for barriers may be obviated, or reduced, by merging many epochs into larger super-epochs.

4.4. Super-Epochs.

A super-epoch is a consecutive subsequence of one or more epochs such that no barriers are interspersed between them. For example, the initial application, represented by $G$, can be considered one large super-epoch. Formally, a super-epoch $\Xi \subseteq G$, is a composition of consecutive epochs $E_i \circ E_{i+1} \circ \cdots \circ E_j$, where each epoch may only belong to a single super-epoch.

That is, super-epochs are disjoint and the composition of all the super-epochs yields $G$. Just like epochs, super-epochs are either simple or complex: A simple super-epoch comprises only simple epochs, and a complex super-epoch contains at least one complex epoch (see Figure 7). Super-epochs are used in exactly the same manner as epochs. Each super-epoch has a corresponding buffer allocation and taking the element-wise maximum over these buffer allocations yields the buffer allocation for $G$. While we can apply the DFBA algorithm directly to super-epochs, a much better approximation can typically be derived by considering the inherent structure of both simple and complex epochs.

A key observation is that a communication graph comprising only simple epochs is acyclic. Consequently, by Lemma 6.3 in [5], the corresponding application is deadlock-free regardless of its buffer allocation. Thus, any simple epoch or simple super-epoch corresponds to a part of the application that is deadlock-free regardless of the application’s buffer allocation. Thus, each simple super-epoch has a deadlock-free buffer allocation of $(0, \ldots, 0)$.

Determining the buffer allocation bound for a complex super-epoch is more involved. The simplest approach is to compute a delay-free buffer allocation for the super-epoch by using the DFBA algorithm of [5]. However, in many cases, this bound can be improved. Instead of considering the super-epoch as a whole we treat it as a composition of two parts: a head and a tail. The head comprises only simple epochs $E_i \circ \cdots \circ E_{k-1}$ where $E_k$ is the first complex epoch in the super-epoch. The tail comprises the epochs $E_{k} \circ \cdots \circ E_j$—a simple super-epoch has no tail and a complex super-epoch whose first epoch is complex has no head. For example, in Figure 4, the first two super-epochs are complex and comprise a head and a tail, while the last super-epoch is simple, comprising only a head.

Since the head is acyclic, its corresponding buffer allocation is $(0, \ldots, 0)$. The buffer allocation for the tail is bounded by the corresponding delay-free buffer allocation. The deadlock-free buffer allocation bound for the entire super-epoch is the pair-wise sum of the buffer allocations for the head and tail, which reduces to the bound for the tail. This is implied by the following lemma:

**Lemma 4.1** Let $\Xi$ be complex super-epoch and let $X$ be the corresponding communication graph that is formed by prefixing start vertices to the process components of $\Xi$. Let $E$ be a simple epoch, let $\Xi' = E \circ \Xi$, and let $X'$ be the communication graph corresponding to $\Xi'$. A deadlock-free buffer allocation $\beta$ for
the program corresponding to $X$, is also deadlock-free for the program corresponding to $X'$.  

**Proof:** Proof by contradiction. Suppose that $\beta$ is not a deadlock-free buffer allocation for the program corresponding to communication graph $X'$. Thus, there exists an execution $\alpha'$ of the program that deadlocks. Consider an execution that comprises four types of events, a la Cypher and Leu [9], denoting when a send commences, when the message is injected into the system, when a receive commences, and when the message is delivered. Let $\sigma$ be the subsequence of events in $\alpha'$ corresponding to the send and receive operation in epoch $E$, and let $\alpha = \alpha' - \sigma$, that is, the same as $\alpha'$, but without the events corresponding to epoch $E$.

Observe that since $E$ is a simple epoch, the program corresponding to communication graph $X'$ cannot deadlock during the send and receive operation in $E$. Hence, $\sigma$ comprises exactly one of each of the four possible events. Now consider the suffix of the event sequence $\alpha'$, beginning after the last event in $\sigma$. Both $\alpha'$ and $\alpha$ have this suffix and the number of buffers available in both executions is the same.

Lastly, observe that $\alpha$ is a valid execution for the program corresponding to $X$. Thus, $\beta$ is not a deadlock-free buffer allocation for the program corresponding to $X$, which is a contradiction. $\square$

Unfortunately, a similar argument does not work if a simple epoch follows a complex one. Without synchronization between the two super-epochs, the latter can use up the message buffers allocated for the former, resulting in a deadlock. An example of this is depicted in Figure 8: super-epoch $\Xi_0$ has a corresponding delay-free (and deadlock-free) buffer allocation $(1, 1, 0)$, and super-epoch $\Xi_1$ has a deadlock-free buffer allocation $(0, 0, 0)$. However, the element-wise sum, $(1, 1, 0)$, of these buffer allocations is not deadlock-free for the corresponding composition of the two super-epochs into one. Namely, in an execution of a program corresponding to the composed super-epoch, process $P_3$ can inject its messages before processes $P_1$ and $P_2$ commence their send operations, using up the message buffers and preventing processes $P_1$ and $P_2$ from proceeding beyond their first send operation. Consequently, super-epoch $\Xi_1$ must also have a buffer allocation $(1, 1, 0)$ if it is to be composed with a preceding complex super-epoch. That is, (in this case) its corresponding buffer allocation must be delay-free as well as deadlock-free to ensure correctness.

**Fig. 8. An unsafe composition of two super-epochs.**


Instead of starting with the linearly ordered epochs generated during the partitioning of graph $G$, the input to our super-epoch composition algorithm is the DAG $D$, which was generated during the first step of partitioning. To compute super-epochs requires three traversals of DAG $D$ (see Algorithm 1). The first traversal computes $Z$, the set of vertices in $D$ that have indegree $0$, that is, no predecessors. The second traversal computes the maximum delay-free buffer allocation, $\beta$, over all epochs $E_v$, corresponding to the vertices $v \in D$. The third traversal constructs the super-epochs, using $\beta$ to bound their size. The construction involves a topological sort of DAG $D$, using a greedy approach to order the epochs.

The third traversal iteratively builds super-epochs by selecting epochs from $G$ and adding them to a super-epoch. Once a super-epoch is filled with epochs it is added to the list of completed super-epochs. Each super-epoch comprises a maximal sequence of simple epochs followed by additional epochs that can be simple or complex. The algorithm chooses which epochs to add to the super-epoch by choosing the corresponding vertices from $Z$, that is, vertices of indegree $0$. When a vertex $v$ is removed from $Z$ and $D$, the corresponding epoch, $E_v$, is added to the current super-epoch, and the list $Z$ is updated with vertices from $D$ whose indegree was reduced to $0$. This continues until $D$ is empty.

The first traversal (line 1) determines the initial set $Z$ comprising all vertices of indegree $0$, which correspond to all epochs that are not preceded by any other epochs in the causal order. The traversal takes linear time but can be avoided by constructing the list $Z$ while constructing DAG $D$. The vertices $v \in Z$ correspond to causally unordered epochs $E_v$ that have not yet been added to a super-epoch—these epochs are candidates for inclusion into the current super-epoch. As epochs are added to a super-epoch, their corresponding vertices are removed from DAG $D$. This decreases the indegree of other vertices in $D$, which are appended to the list $Z$ when their indegree reaches 0.

Using the DFBA algorithm of [5], the second traversal (line 2) computes the delay-free buffer allocation (DFBA) of the complex epochs in $G$, which are represented by vertices in DAG $D$. The traversal computes an element-wise maximum buffer allocation, $\beta$, over these buffer allocations, which is sufficient to ensure deadlock-freedom for each epoch in $G$ and is used as a threshold in the construction of super-epochs by the third traversal (line 3). The second traversal could be incorporated into the first traversal, but is treated separately for clarity.

The third traversal uses the list $Z$ and buffer allocation $\beta$ to assemble epochs into super-epochs. An iteration of the main loop (line 3) assembles a super-epoch from one or more epochs. The first half of an iteration, starting at line 4, constructs the head of the super-epoch. The second half of the iteration, starting at line 13, constructs the tail of the super-epoch. Lastly, the head and tail are composed and appended to the list of completed super-epochs (line 22).

The head, by definition, comprises simple epochs. Thus, the head is constructed by scanning the list $Z$ for vertices corre-
Algorithm 1: Constructing Super Epochs

Input: D
Output: L, β
Local: Z, X, Π/head, Π/tail
1. Z ← {v | v ∈ D ∧ indegree(v) = 0}
2. β ← maxv∈D{DFBA(E_v)}
3. while Z ≠ ∅ do
   4. Π/head ← ∅
   5. foreach v ∈ Z do
      6. if E_v is simple then
         7. Π/head ← Π/head ∪ E_v
         8. X ← {u | (v, u) ∈ D ∧ indegree(v) = 1}
         9. Remove v from Z and D
         10. Append X to Z
      end
   end
   11. Π/tail ← ∅
   12. foreach v ∈ Z do
      13. if DFBA(Π/head ∪ E_v) ≤ β then
         14. Π/tail ← Π/tail ∪ E_v
         15. X ← {u | (v, u) ∈ D ∧ indegree(v) = 1}
         16. Remove v from Z and D
         17. Append X to Z
      end
   end
   18. Append Π/head ∪ Π/tail to L
end

sponding to simple epochs (line 6), removing the vertices from Z and D (line 9), and appending the corresponding simple epochs to the head, Π/head (line 7). After a vertex is removed from D, any vertices whose indegree is reduced to 0 are appended to Z (lines 8 and 10), regardless of whether the corresponding epochs are simple or complex. Consequently, all vertices that are appended to Z will also be scanned and processed in the same way.

It is important to note that all vertices in Z correspond to epochs that are not causally ordered. Hence, the order in which vertices are removed from Z is irrelevant. Furthermore, since Z is scanned from start to end, all vertices in Z that correspond to simple epochs will be removed. Thus, after the head has been constructed, the list Z will not contain any vertices that correspond to simple epochs.

The construction of the tail, is similar to the construction of the head, except that the condition for addition to a super-epoch is not whether the epoch is simple, but rather, whether buffer allocation β is delay-free for the current super-epoch composed with the epoch being processed. The second loop (line 14) scans Z, starting at the front of the list. For each vertex v ∈ Z, it computes the DFBA on Π/tail ∪ E_v (line 15); the current super-epoch composed with the corresponding epoch. If the resulting allocation is bounded (element-wise) by β, the buffer allocation computed by the second traversal, then epoch E_v is permanently appended to super-epoch Π/head (line 16), the vertex v is removed from Z and D (line 18), and all vertices in D whose indegree is reduced to 0 are added to the list Z (lines 17 and 19). This continues until all vertices in Z have been scanned. The tail construction completes when the second loop terminates. Lastly, the head and tail are composed and appended to L, the list of super-epochs (line 22).

The result is an ordered list of super-epochs for which β is a deadlock-free buffer allocation. Thus, β is also a deadlock-free buffer allocation for the program corresponding to the original communication graph G, provided that barrier synchronizations are inserted between consecutive super-epochs instead of epochs.

For example, consider the communication graph and corresponding DAG D in Figure 9. The vertices are labeled by the corresponding epochs and are numbered indicating the order in which they are added and removed from the list Z. Initially, only vertex E1 is added. When it is removed from the list Z, vertex E2 is added. In many cases multiple vertices are added and removed from the list Z.

5. Implementation

The partition of the communication graph into super-epochs provides a precise guide pin-pointing where in the application barrier synchronizations must be added to ensure the validity of the computed buffer allocations. This can be derived directly from the communication graph since each vertex in the communication graph corresponds to an invocation of a send or receive. Based on our initial assumptions, the communication operations performed by each process are well ordered and the ordering does not change from execution to execution. Consequently, if event i of a process occurs in super-epoch j, and event i + 1 event occurs super-epoch j + 1, then, a barrier synchronization must occur immediately prior to event i + 1.

To insert the barrier synchronizations several options are available. If the application is relatively simple, the synchronizations could be inserted by hand directly into the code. A more elegant solution is to use wrappers for the communication primitives. A wrapper implementation of a communication function is a new function that calls the original communication function, but which also performs some pre- and/or post processing. The barrier synchronization can be implemented in the pre- or post processing code, depending on where it is needed. Prior to each invocation, the code in the wrapper would increment a counter and use a lookup table to determine if a barrier synchronization should be performed. In particular, if a barrier synchronization is needed in a loop construct, no code rewriting is needed, the wrapper code will automatically detect the need for a barrier synchronization regardless of where in the code the communication occurs.

In Section 5.1 we describe an actual implementation of our algorithms in MPI. The implementation’s functionality includes gathering the application messaging information from an initial run, computing the strongly connected components of the resulting communication graph, composing super-epochs from
these epochs, and utilizing the super-epoch partition in subsequent executions of an application.

5.1. An MPI implementation

The implementation of the system for MPI consists of three separate parts:

Data Collection Facility
A runtime system that gathers data about the communication pattern of an application in order to compute a buffer allocation. Since all executions will exhibit the same communication pattern, collecting this data once is sufficient to compute an allocation for future executions on the same problem size.

The Buffer Allocation Approximation
An implementation of the algorithms described in the previous sections of this paper that partitions the communication graph into super-epochs and generates a barrier synchronization schedule.

Runtime Support for Automatic Barrier Synchronization
A runtime system that loads the barrier synchronization schedule and uses it to automatically perform barrier synchronizations at the determined times.

Both the data collection as well as the barrier synchronization support require special runtime code to be linked to the application (thus, must be written in C like the MPI message passing library), whereas the barrier synchronization schedule can be computed offline, based on the data collected, using any programming language.

Implementing MPI Runtime Support and Data Collection.
The information that needs to be collected is simply the process’ ID number (equivalent to the process component id), and the event’s number. This information is sufficient to construct the communication graph, but must be collected transparently.

To make of our system as transparent as possible, users should not have to make any changes to their program. Thus, the collection of the initial data to construct the communication graph, as well as the automatic barrier synchronization must not require application modification.

To achieve this we use function wrappers to intercept calls to the underlying communication library. For each MPI function, say MPI_XXX(...), we implement a wrapper function _MPI_XXX(...) that performs data collection and scheduled barrier synchronizations. We use the C-preprocessor to replace all calls to MPI_XXX(...) in the application with calls to _MPI_XXX(...) before compiling the code. When the application is executed, the variables _BAP_COLLECT and _BAP_SYNC are set appropriately depending on the mode in which the program runs. This mode is controlled through the command line parameters passed to mpirun, the execution script with which all MPI programs must be started. Figure 10 shows a generalized implementation of such a function.

```c
int _MPI XXX(...) {
    if (_BAP_COLLECT) {
        fprintf(fp, "%d:%d. . .
", _bap_my_rank, _bap_eventNo);
    } else if (_BAP_SYNC) {
        while (_bap_eventNo < SyncEvents.top()) {
            MPI_BARRIER(MPI_COMM_WORLD);
            SyncEvents.pop();
        }
        _bap_eventNo++;
        return MPI XXX(...);
    }
}
```

Fig. 10. A function wrapper with data collection and barrier synchronization mechanisms.
Initialization code that opens the output file for data collection or loads the barrier synchronization schedule for normal execution is invoked from the wrapper function for MPI_Init(...) that all MPI processes must call before engaging in any interprocess communication.

The Buffer Allocation Approximation. A Java prototype of the buffer allocation approximation algorithm uses the collected data to generate a buffer allocation and a barrier synchronization schedule. The schedule comprises a list of entries, one per process. Each entry consists of the process’s ID and a list of tuples comprising barrier identifiers and event ranges. Thus, counting communication events suffices to determine when to perform barrier synchronizations.

Performing Barrier Synchronization. Every process must participate in a barrier synchronization in order for any process to pass a barrier. In general, to perform the barrier synchronization, the first event of each super-epoch will participate in the barrier. However, if one or more of the processes do not have any events in the super-epoch, performing a barrier synchronization is impossible as no MPI functions are called. This is illustrated in Figure 11, where process $P_3$ does not have any events in super-epoch (1). In order to ensure that $P_3$ participates in the barrier synchronization, the synchronization must then happen at the first event of the following super-epoch, in which the second barrier synchronization must also occur. This event is marked $\Diamond$ in the figure. If a process never again participates in any communication, all remaining barrier synchronizations must happen in the final call to MPI_Finalize(...).

Optimizing the Barrier Synchronization using Partial Barriers. A potential slowdown of the execution can occur if a process does not have events in a number of (consecutive) super-epochs, that is, it does not communicate with any other processes, but performs a large amount of computation, thus taking a long time to reach the next synchronization point. It is possible to exclude the process from the synchronization according to the following principle:

**Principle 1** Process $P_i$ must synchronize at the beginning of super-epoch $e$ if and only if $P_i$ has at least one event in super-epoch $e$.

If $P_i$ has zero events in super-epoch $e$, it does not take part in any communication that results in a buffer requirement for process $P_i$ in super-epoch $e$. This implies that all available buffers are free, which in turn implies that the next super-epoch can commence.

In the example in Figure 11, process $P_3$ has no events in the super-epoch (1). Instead, two synchronizations can take place the following super-epoch (marked by a $\Diamond$ in Figure 11). In reality, the first synchronization done at $\Diamond$, belonging to the previous super-epoch, is not needed, but performing barrier synchronizations that do not involve all processes (known as a partial barrier) is not easily accomplished in MPI (i.e., there is no built-in primitive). However, by using a large number of MPI communicators, one for each combination of processes that need to synchronize, it is possible to utilize the regular MPI_Barrier() function. A different approach is to implement an independent partial barrier mechanism by using an additional “barrier process”.

Implementing this in MPI is straightforward. A barrier synchronization corresponds to a pair of send and receive calls:

```
MPI_Send(&e, 1, MPI_INT, P_BAR_PROC_ID, COMM);
MPI_Recv(&e, 1, MPI_INT, P_BAR_PROC_ID, COMM, &status);
```

The send sends the current super-epoch number to the “barrier process”, which has a list, $SE$, of all the super-epochs. A list entry, $SE[i]$, comprises a list of the processes that must perform a barrier synchronization at the start of the super-epoch ($SE[i].members$) and a count of those processes yet to perform this synchronization ($SE[i].count$). Upon receipt of a super-epoch number, the corresponding count is decremented and if the count reaches 0, a response is sent to all the corresponding members. Figure 12 shows the code for such a “barrier process”:

```
noBarriers = noEpochs−1;
MPI_Init(&argc, &argv);
while (noBarriers > 0) {
    MPI_Recv(&i, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, inter_comm, &status);
    E[i].count--; 
    if (E[i].count == 0) {
        for (j=0;j<E[i].memberCount;j++)
            MPI_Send(&i, 1, MPI_INT, E[i].members[j], 1, inter_comm);
    }
    noBarriers--; 
}
MPI_Finalize();
```

Fig. 12. A “barrier process” that implements partial barriers.

6. Evaluation

To verify that our new approach (NA) for buffer allocation is better than the simpler DFBA approach, we implemented an application suite of five different parallel algorithms. We first briefly describe the problems and their communication...
structure, and then compare the performance of the suite with respect to the two approaches (NA and DFBA).

6.1. The Evaluation Platform

The evaluation was performed on a 192-node Suse Linux-based cluster with AMD Athlon 64 bit 3200+ CPUs, each with 2 GB of memory and a 512 KB cache, connected via a 1 Gb Ethernet interconnect for communication, and a 100 Mb Ethernet for I/O and file access. Communication between processes was realized using the MPIICH implementation of MPI [3].

We noted that the cluster’s interconnect was susceptible to significant slowdown, and occasional failure when large amounts of data were sent to a single host. Typically this occurred at the end of the test runs, during the data gathering phase of the application, when the slaves sent their results to the master process.

Additionally, in a couple instances the variability in the run times increased dramatically for a particular configuration of runs, but remained low for all other configurations. This high variability persisted regardless of when we ran the tests or the choice of cluster nodes. This is further discussed in Section 6.6.

6.2. The Test Suite

Five applications were implemented for our test suite. These included 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 (DES). In the following subsections we describe the problems and the instance sizes.

6.2.1. Pipe-and-Roll Matrix Multiplication (MM)

This algorithm comprises one coordinator process and worker processes that are assigned in a torus-like 2-dimensional $\sqrt{n} \times \sqrt{n}$ grid. The computation proceeds in rounds. Each round comprises 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 neighbour, 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 neighbour (with wrap around) of the initiator in the preceding round. For example, suppose that nine processes are arranged in three rows of three processes each:

$$
\begin{align*}
& P_0 \quad P_1 \quad P_2 \\
& P_3 \quad P_4 \quad P_5 \\
& P_6 \quad P_7 \quad P_8
\end{align*}
$$

The first round will have initiators $P_0, P_3, P_6$, the second round $P_1, P_4, P_7$, and finally $P_2, P_5, P_8$. Implementation details can be found in [14, Chapter 10].

Our tests used 3,000 × 3,000 matrices with floating point entries.

6.2.2. Fast Fourier Transformation (FFT)

Given a vector $x = \{x_0, \ldots, x_{m-1}\}$ of size $m$ (in our case $m = 2^17$), this algorithm computes the fast Fourier transform of $x$. Namely, $x' = \{x'_0, \ldots, x'_{m-1}\}$, where

$$
x'_k = \sum_{j=0}^{m-1} x_j e^{2\pi i k j / m}.
$$

The number of processes ($n$) should also be a power of 2. Each process is assigned $2^{17}/n$ elements from the array. The algorithm uses a “butterfly communication pattern”: Each process performs $\log n$ exchanges with 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 [25, Page 398]—in total $n \log n$ exchanges take place. After $\log n$ exchanges, each process has computed $x'$.

Our tests perform the computation 4,000 times using an input vector of size $2^{17}$.

6.2.3. 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 neighbours. A master process collects the results from all the processes at the end of the computation.

Our tests used a grid of size $3,842 \times 3,842$ and ran the simulation for 1,000 rounds.

6.2.4. N-Body Problem (NBP)

The N-Body problem is an instance of the Long Range Interaction Problem [14, Chapter 9]. 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

$$
\sum_{i=0}^{m} \sum_{j=0}^{m} f(e_i, e_j)
$$

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 used a problem instance of 57,344 elements, and performed 4 rounds.

6.2.5. 1-D Differential Equation Solver (DES)

This algorithm arranges the $n$ processes in a ‘string’ each with west and east neighbours (except the end points). Each process receives $m/n$ elements of an $m$-element array. Each element represents a point of a 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 neighbours, and then refines the values of the elements that it has been allocated. Further details can be found in [14, Chapter 5].

Our tests use an instance size of 8,000,000 elements that were refined over 1,000 rounds.

6.3. A Comparison of Buffer Allocations

Table 1 shows the buffer allocations for the five applications, executing on an \( n \) process system, yielded by the new approach (NA) and by the delay-free buffer allocation (DFBA) approach. The new approach yields smaller buffer allocations than DFBA in nearly all of the test cases. Most significantly, for all five applications, the NA approach allocated at most two buffers per process, instead of \( O(\log n) \) to \( O(n) \) buffers (depending on the application), reducing the per process buffer consumption to a lower order constant and the total buffer consumption by up to a factor of \( n \).

6.4. Number of Epochs versus Super-Epochs in an Application

Another useful measure, which arose from the techniques applied in the algorithm, is the number of barrier synchronizations required during the execution of an application. As described earlier, the algorithm uses the concept of an epoch to ensure that buffers can safely be reused instead of allocating additional ones. Since each epoch must commence with a barrier synchronization, which is an expensive operation, we developed the concept of super-epochs to minimize the number of barrier synchronizations that need to be performed.

Table 2 summarizes the number of barrier synchronizations that our suite of test applications needed to perform in order to use the buffer allocations yielded by NA. The last column shows the relative improvement (the reduction in the number of synchronizations) obtained by using super-epochs over epochs.

<table>
<thead>
<tr>
<th>App.</th>
<th>Buffer Allocation</th>
<th>Buffs. / Proc</th>
<th>Total Buffers</th>
<th>Improvement Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>((0, \ldots, 0))</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((n, O(\sqrt{n}), \ldots, O(\sqrt{n})))</td>
<td>(n)</td>
<td>(O(n\sqrt{n}))</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>((1, \ldots, 1, 1))</td>
<td>1</td>
<td>(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((O(\log n), \ldots, O(\log n), n))</td>
<td>(O(\log n))</td>
<td>(O(n \log n))</td>
<td></td>
</tr>
<tr>
<td>HG</td>
<td>((1, \ldots, 1))</td>
<td>1</td>
<td>(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((n - 1, 2, 3, \ldots, 3, 2))</td>
<td>(n - 1)</td>
<td>(4n - 6)</td>
<td></td>
</tr>
<tr>
<td>NBP</td>
<td>((1, \ldots, 1))</td>
<td>1</td>
<td>(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((n, n, \ldots, n))</td>
<td>(n)</td>
<td>(n^2)</td>
<td></td>
</tr>
<tr>
<td>DES</td>
<td>((1, 1, 2, \ldots, 2, 1))</td>
<td>2</td>
<td>(2n - 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((n - 1, 2, 4, \ldots, 4, 2))</td>
<td>(n - 1)</td>
<td>(5n - 9)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1

Buffer allocations computed by the NA and DFBA on \( n \) process systems.

In the majority of cases the number of barrier synchronizations is reduced by a couple orders of magnitude.

There are two exceptional cases: the Matrix Multiplication and the Differential Equation Solver. The Matrix Multiplication application consists entirely of simple epochs, all of which form one super-epoch. Thus, for such applications no additional synchronization is needed. On the other hand, in the case of the Differential Equation Solver, almost no improvement was observed. This is because the communication pattern consists of, almost exclusively, causally dependent complex epochs, each of which becomes a super-epoch (assuming the minimal buffer allocation). The number of super-epochs in this case can be reduced by using additional per process buffers. For example, if each process was allocated \( b \) buffers, then each super-epoch would contain up to \( b \) consecutive complex epochs, implying that only \( \frac{1}{b} \) of the super-epochs would be required.

6.5. Overview of Evaluation

As part of the evaluation of our approach we wished to determine its scalability, overhead, and cost, in terms of application run-time. To investigate scalability we varied the number of processors on which the application ran, ranging the number of from 17 to 145 processors in a geometric progression—all of our applications used an additional process to gather the results of the computations.
The other variable was the mode of our system. Specifically, our system has four modes: off (no synchronization) mode, in which our system is turned off; data collection mode, in which our system collects data to construct the communication graph; full barrier synchronization mode, in which our system performs barrier synchronizations between the computed super-epochs using the MPI’s barrier mechanism; and partial barrier synchronization mode, in which our system uses our own barrier mechanism to synchronize between super-epochs.

The results are summarized in Figure 13 and analyzed in the following sections. The four bars in each group correspond to the four modes of the system and the groups represent the runtimes on various number of processors. The first bar in each group represents the base-line run-time against which other runtimes are to be compared.

6.6. The Data Collection Overhead

The overhead associated with collecting the data to construct the communication graph is primarily due to the need to log and aggregate communication events to persistent storage. Table 3 summarizes the runtimes of the application suite both with and without data collection. The reported runtimes are the minimums of ten different runs for each application in the same configuration. The runtime of a single execution is the runtime of the slowest process in the application. This is typically the process responsible for gathering the results from the remaining processes.

<table>
<thead>
<tr>
<th>App.</th>
<th>( n )</th>
<th>Runtime with Data Collection</th>
<th>Std. Dev.</th>
<th>Runtime without Data Collection</th>
<th>Std. Dev.</th>
<th>Slowdown Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>17</td>
<td>50.41</td>
<td>0.13</td>
<td>50.36</td>
<td>0.18</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>53.01</td>
<td>0.28</td>
<td>53.15</td>
<td>0.21</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>29.78</td>
<td>0.56</td>
<td>29.71</td>
<td>0.71</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>23.95</td>
<td>0.24</td>
<td>24.12</td>
<td>0.55</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>145</td>
<td>22.26</td>
<td>1.92</td>
<td>22.20</td>
<td>1.73</td>
<td>1.00</td>
</tr>
<tr>
<td>FFT</td>
<td>17</td>
<td>52.42</td>
<td>0.18</td>
<td>51.43</td>
<td>0.25</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>42.86</td>
<td>0.03</td>
<td>44.50</td>
<td>0.86</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>45.62</td>
<td>0.36</td>
<td>43.29</td>
<td>0.32</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>52.67</td>
<td>1.65</td>
<td>41.30</td>
<td>8.34</td>
<td>1.28</td>
</tr>
<tr>
<td>HG</td>
<td>17</td>
<td>61.82</td>
<td>0.33</td>
<td>61.20</td>
<td>0.40</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>60.70</td>
<td>0.42</td>
<td>60.78</td>
<td>0.45</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>41.93</td>
<td>0.62</td>
<td>43.11</td>
<td>0.58</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>30.89</td>
<td>0.30</td>
<td>31.15</td>
<td>0.36</td>
<td>0.99</td>
</tr>
<tr>
<td>NBP</td>
<td>17</td>
<td>57.21</td>
<td>0.76</td>
<td>55.85</td>
<td>1.29</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>50.21</td>
<td>0.35</td>
<td>51.03</td>
<td>0.21</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>25.59</td>
<td>0.57</td>
<td>25.74</td>
<td>1.18</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>13.98</td>
<td>0.17</td>
<td>13.97</td>
<td>2.45</td>
<td>1.00</td>
</tr>
<tr>
<td>DES</td>
<td>17</td>
<td>59.44</td>
<td>0.29</td>
<td>59.33</td>
<td>0.92</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>56.73</td>
<td>0.87</td>
<td>58.35</td>
<td>0.55</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>47.99</td>
<td>1.09</td>
<td>49.02</td>
<td>0.82</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>49.02</td>
<td>0.50</td>
<td>48.13</td>
<td>0.65</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 3

Runtimes of applications on \( n \) processes with and without data collection.
We conclude that the overhead is negligible. In general, the slowdown experienced by performing data collection is swamped by the variance in the ten runs of each application. This is not surprising since many well designed parallel applications are not communication-bound and the buffering and caching mechanisms of the operating system overlap disk accesses with the application’s computation.

Most importantly, the collection needs to be done only once, whereas the execution of the algorithm, which uses the buffer allocation computed from the data, will likely occur many times. Thus, the overhead associated with barrier synchronization of the latter kind of executions is much more important.

We note that the variance for the FFT configuration on 33 processors is very high. The interesting point is that this happens only for this application, and only when it is running on 33 processors in a non-synchronized mode. We conjecture that the amount of data being sent is on the cusp of the threshold needed to swamp the interconnect. Correspondingly, on some runs the interconnect becomes swamped and on others it does not.

One interesting artifact is that in some cases the data collection run-time seems to be lower than the base-line run-time. This is because we report the minimum run-times, rather than the average. In fact, in all cases the run-time difference is within variance, and hence is effectively negligible.

6.7. Application Execution Overhead

An application slowdown is caused when the application must perform barrier synchronizations on a regular basis; the greater the number of barriers, the greater the slowdown. This slowdown is the inherent cost of our approach—in Section 6.9 we investigate the cost of a barrier operation.

Table 4 summarizes the slowdowns experienced by the suite of test applications. As before, each application was executed ten times for each configuration and the minimum is reported. The table reports runtimes of the applications executing in three different synchronization modes:

No Barrier Synchronization (Nbs) in which the application is run without the runtime system for data collection and barrier synchronization linked in,

Full Barrier Synchronization (Fbs) in which the application uses the primitive MPI_Barrier() to synchronize at the end of each super-epoch, and

Partial Barrier Synchronization (Pbs) in which the application uses the partial synchronization mechanism that we described earlier.

The table also reports the slowdown factor measured between Nbs and the Fbs modes, and the speedup factor between the Fbs and the Pbs modes.

Several interesting features can be noted from the table. First, since the test suite was run on a cluster comprising commodity hardware and software, there is some variance between the run-times of an application in the same configuration. Consequently, for some of the tests, the difference in runtimes is swamped by the variance. That is, there is no perceived slowdown. We now consider each application in detail. In particular we focus on the overhead of performing full barrier synchronizations—we consider partial barrier synchronizations later on.

The Matrix Multiplication (MM) application comprises 1 super-epoch and consequently uses no barriers. In such cases our system should not, and does not present any additional overhead. The minor increase in the run-time of the test run on 145 nodes, using partial barrier synchronization, falls within the variance of the measurements, and hence, as mentioned in the previous section, is an artifact of the fact that we list minimum run-times. One interesting anomaly is the lack of speed-up from 17 to 33 processors. We believe that this is caused by the swamping of the interconnect—we have observed this happening when many processors send sufficiently large amounts of data simultaneously. As the number of processors increases, the amount of the data sent decreases, and the interconnect ceases to be swamped, resulting in a good speed-up.

The FFT application is an excellent measure of the overhead of our system on applications that have many super-epochs. In the case of FFT, the number of epochs increases proportionately to the logarithm of the number of processes. Thus, as the number of processes increases, so does the number of epochs. Furthermore, as the number of processes increases, the amount of data exchanged between processes, during a single communication, decreases. Thus, there are three competing effects on the run-times: the computation time, the communication time,
and the synchronization time. For test runs on few (17) processes, the former two dominate. The reason for the slowdown in the synchronization run-times is that the synchronization is restricting concurrency, preventing overlap of some computation and communication, thus increasing the overall run-time. For 33 processes this penalty is mitigated because the computation and communication times are halved and since the cost of synchronization remain small, resulting in an overall negligible overhead. For 65 and 129 processes the time to synchronize among the processes dominates, resulting in a significant overhead. Thus, for applications with many super-epochs, running on many processors, the overhead is significant.

Potentially, slowdown due to a reduction in concurrency can occur as a direct effect of a barrier synchronization—this is independent of the time needed to perform the synchronization itself. The Heat Grid (HG) application is a prime example where the reduction in concurrency produces a significant effect. When running on few processes, the runtime of HG is dominated by computation and communication, which can be overlapped when there are no barrier synchronizations. However, due to the staggered nature of the HG communication pattern, barrier synchronization between super-epochs prevents this overlap. As the number of processors increases, the communication and computation times decrease, reducing the effect of the synchronizations. Since HG has one tenth the number of super-epochs of FFT, its barrier synchronization times, on many processors, are correspondingly lower.

The N-Body problem test illustrates the correlation between the overhead of our system and the number of super-epochs that comprise the application. The speed-up from running on 17 processors, to running on 33 processors, to running on 65 processors, is healthy, and the overhead of our system is negligible. This changes when the application is run on 129 processors. The key point to note is that in this application, the number of super-epochs is proportionate to the number of processors. Thus, doubling the number of processors, doubles the number of super-epochs, and increases the time it takes to perform a barrier synchronization. Thus, the overhead in the test runs on 129 processors is a result of these two factors. In Section 6.10 we describe how to mitigate this effect.

The Differential Equation Solver (DES) application test illustrates two interesting properties. First, the application is communication bound, resulting in almost no speed up as the number of processors is increased. Second, the number of super-epochs does not change with respect to the number of processors. Thus, the only overhead should be from the time to perform the synchronizations. The overhead for performing synchronization is negligible in all runs. In fact, it may seem that the synchronized tests run faster than the unsynchronized ones. However, as before, the differences are within the variance.

Partial barrier synchronizations typically had a much higher overhead than full barrier synchronizations. The reason for the poor performance is partially due to the implementation of the approach. The current implementation uses a single process to manage all the partial barrier synchronizations. This creates a communication bottle-neck when many processes synchronize at the same time. Consequently, for configurations with low number of processes, the partial barrier performed no worse than the native full barrier synchronization. On the other hand, for configurations with large number of processes, the partial barrier performed significantly worse, and in fact no benefit from a partial barrier implementation was realized. In the next section we explain why.

6.8. Full versus Partial Barrier Synchronization

We initially believed that partial barrier synchronization would perform better than full barrier synchronization. This turned out not to be the case. The reason for this is that most barrier synchronizations in each of our applications involved almost all processes. Consequently, the additional overhead of using a partial barrier synchronization was not recouped, because almost all processes participated in each synchronization.

The synchronization count for a process is the number of barrier synchronizations that it participates in. The synchronization count for an entire application is the sum of the synchronization counts of all the processes of that application. For applications using full barrier synchronization, the synchronization count is \( (e - 1)n \), where \( n \) is the number of processes, and \( e \) the number of super-epochs. For applications using partial barrier synchronization, the synchronization count is at most that of full barrier synchronization. As Table 5 illustrates the differences

<table>
<thead>
<tr>
<th>App.</th>
<th>( n )</th>
<th>Sync. Count (Pbs)</th>
<th>Sync. Count (Fbs)</th>
<th>Improvement Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>145</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>FFT</td>
<td>17</td>
<td>260,000</td>
<td>272,000</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>644,000</td>
<td>660,000</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>1,540,000</td>
<td>1,560,000</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>3,588,000</td>
<td>3,612,000</td>
<td>0.99</td>
</tr>
<tr>
<td>HG</td>
<td>17</td>
<td>30,043</td>
<td>34,221</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>62,091</td>
<td>66,957</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>126,187</td>
<td>133,965</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>254,379</td>
<td>274,125</td>
<td>0.93</td>
</tr>
<tr>
<td>NBP</td>
<td>17</td>
<td>1,028</td>
<td>1,088</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>4,100</td>
<td>4,224</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>16,388</td>
<td>16,640</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>65,540</td>
<td>66,048</td>
<td>0.99</td>
</tr>
<tr>
<td>DES</td>
<td>17</td>
<td>16,001</td>
<td>17,000</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>32,001</td>
<td>33,000</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>64,001</td>
<td>65,000</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>129</td>
<td>128,001</td>
<td>129,000</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 5

Full versus Partial barrier synchronization counts.
in synchronization counts for applications using full and partial barrier synchronizations are small. Consequently, there is no benefit to using partial barrier synchronization in this case.

The reason for this is subtle. Our algorithm attempts to reduce the number of super-epochs by making them as large as possible. In doing so, it maximizes the number of processes participating in each super-epoch. Consequently, it is not surprising that the number of processes performing a barrier synchronization at the end of each super-epoch is maximal.

Given that full barrier synchronization is the better performing approach, the next question that we need answered is: how long does it take to perform a barrier?

6.9. Cost of Barrier Synchronizations

To evaluate how long it takes to perform a barrier implementation we wrote a simple parallel application in which the processes communicated in a ring: in each of the 10,000 rounds each process sent 1 byte to its left neighbour and received 1 byte from its right neighbor. Naturally, each round corresponds to 1 super-epoch and between the super-epochs a full barrier synchronization (or not) was performed, via MPI_Barrier. The application was run on a number of different processor configurations, ranging from 2 processors to 128 processors, and the results are illustrated in Figure 14 and are tabulated in Table 6.

As the number of processors increases the overhead of performing a barrier synchronization also increases. However, the overhead is relatively minor: for 128 processors, the overhead of performing 10,000 barriers is 3 seconds. Thus, while barrier synchronizations are not free, they, in themselves, do not contribute significantly to the overhead.

What does contribute significantly to the overhead is the effect of the synchronization on application concurrency, particularly the ability to overlap computation and communication within a process and between the processes. One way to mitigate this problem is to increase the size of the super-epochs, allowing more activity within the super-epoch and hence increase concurrency within the application. The next section reports on our experiments with alternative buffer allocations.

6.10. Increasing the Per Process Buffer Allocations

The buffer allocations used in the previously reported tests were the minimal buffer allocations determined by our new approach. That is, the smallest DFBA allocation suitable for all super-epochs of the application. In all the computed allocations no per process allocation was greater than 2, that is, if 2 buffers were allocated for each process, then the applications in the application suite would be safe. However, if more buffers are available to each process, then by increasing the per process allocation, prior to the super-epoch construction, a reduction in the number of super-epochs is possible. Consequently, the number of barrier synchronizations would also be reduced.

We experimented with this approach on some of the applications in our test suite; in particular the configurations that had relatively his synchronization overhead. Table 7 shows the results of using an alternative buffer allocation.

The table illustrates an important property. Namely, the number of super-epochs affects the performance of the applications. Reducing the number of super-epochs by increasing the per process buffer allocations greatly improved the performance.

In the case of the Heat Grid (HG) simulation, a small increase in the number of buffers resulted in a dramatic decrease in the number of super-epochs and a comparable performance increase. This is because the communication pattern of HG is very similar to the one in Figure 15. If only the minimum number of per process buffers are allocated, each complex epoch corresponds to a super-epoch. However, by slightly increasing the number of buffers, the complex epochs can be folded up into a single super-epoch, greatly reducing the number of barrier synchronizations required. This is an example where a small additional allocation yields a large performance gain.

<table>
<thead>
<tr>
<th>Prob. Inst.</th>
<th>Buffer Allocation</th>
<th>Super-Epochs (number of)</th>
<th>Pbs time (seconds)</th>
<th>Fbs time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT (65)</td>
<td>(1,1,1,...,1)</td>
<td>24,001</td>
<td>83.12</td>
<td>56.15</td>
</tr>
<tr>
<td></td>
<td>(8,8,8,...,8)</td>
<td>4,000</td>
<td>62.82</td>
<td>45.21</td>
</tr>
<tr>
<td></td>
<td>(64,64,64,...,64)</td>
<td>1</td>
<td>43.76</td>
<td>43.15</td>
</tr>
<tr>
<td>FFT (129)</td>
<td>(1,1,1,...,1)</td>
<td>28,001</td>
<td>199.95</td>
<td>82.01</td>
</tr>
<tr>
<td></td>
<td>(8,8,8,...,8)</td>
<td>4,000</td>
<td>82.56</td>
<td>46.13</td>
</tr>
<tr>
<td></td>
<td>(128,128,128,...,128)</td>
<td>1</td>
<td>43.34</td>
<td>43.41</td>
</tr>
<tr>
<td>HG (17)</td>
<td>(1,1,...,1)</td>
<td>2,014</td>
<td>172.24</td>
<td>172.33</td>
</tr>
<tr>
<td></td>
<td>(3,3,...,3)</td>
<td>2</td>
<td>63.89</td>
<td>64.21</td>
</tr>
<tr>
<td>HG (33)</td>
<td>(1,1,...,1)</td>
<td>2,030</td>
<td>102.36</td>
<td>100.91</td>
</tr>
<tr>
<td></td>
<td>(3,3,...,3)</td>
<td>2</td>
<td>63.38</td>
<td>63.36</td>
</tr>
</tbody>
</table>

Table 7 Performance of select configurations with larger per process buffer allocations.
ensure that the head portion of a super-epoch is deadlock-free. In the static case, the developer specifies buffer allocation \( \beta' > \beta \) and runs the algorithm, that uses \( \beta' \) instead of \( \beta \). Since \( \beta' > \beta \), the program can be partitioned into fewer, but larger, super-epochs. Since barrier synchronizations need only occur between super-epochs, barrier fewer synchronizations will be required. Analogous to bin-packing, if some epochs require large deadlock-free buffer-allocations, buffer allocation \( \beta' \) may need to be significantly larger than \( \beta \) before larger super-epochs can be composed. The developer is free to decide what the right trade-off is for their application, or even experiment with different buffer allocations, as long as \( \beta' \geq \beta \). However, it is not always possible to know a priori how many buffers may be available to the application.

In the future, with the advent of grid and cloud computing, dynamic environments may pervade, where multiple parallel applications execute on a single host. In such a dynamic environment, the number of available buffers may change quickly and unpredictably. Thus, a dynamic approach may be more applicable. Initially, each application reserves the minimal buffer allocation, \( \beta \). If additional buffers can be reserved, in order to extend the current super-epoch, the application reserves the buffers and dynamically extends the super-epoch. The super-epochs are essentially computed online by the epoch composition algorithm, using the currently reserved buffer allocation. If the additional reserves are recalled by the underlying system—if say another application starts up—the additional reserves are released at the end of the current super-epoch. The next super-epoch is then composed and executed with respect to the minimal buffer allocation, \( \beta \).

As described in Section 5, the mechanisms for accomplishing this are invoked within the calls to the communication library. Thus, the application itself needs no modifications. Furthermore, instead of the developers making local decisions about how many buffers their applications need, the entire cluster can allocate resources globally, which typically results in better utilization. This requires close interaction between the communication library and the operating system, which is typically not the norm.

8. Conclusion

Applications that use asynchronous message passing can suffer from deadlock if the underlying system has an insufficient number of message buffers. Consequently, determining or bounding the minimum number of message buffers needed to prevent such deadlocks is an important part of developing portable parallel applications. Since the Buffer Allocation Problem is intractable, approximation remains the only recourse.

We presented an epoch-based polynomial-time approximation that yields tight bounds in many common cases, and typically performs better—and never worse—than previously known approaches. The approach partitions the application execution into disjoint epochs that are interspersed by barrier synchronizations. These ensure that all message buffers are free at the start of each epoch, facilitating the determination of buffer allocations on a per-epoch basis. The application’s...
buffer allocation is the element-wise maximum over all the epochs' buffer allocations.

Our approach consists of three parts. First, the data collection phase that logs all communication events in the application, from which a communication graph is constructed. Second, the analysis phase that partitions the communication graph into disjoint epochs, determines the minimum buffer allocation over all the epochs, and combines the epochs into super-epochs. Third, future executions of the application use the derived super-epoch schedule to perform barrier synchronizations, ensuring that buffers are available when needed.

Overall the analysis has a $O(|G|/n)$ time complexity, where $|G|$ is the size of the communication graph and $n$ is the number of processes. While the partitioning of the communication graph has linear complexity, determining the buffer allocation of an epoch, via the DFBA algorithm of Brodsky et al. [5], takes $O(|E|/n)$ time, where $|E|$ is the size of the epoch. Consequently, the analysis on communication graphs that comprise a few large epochs would have $O(|G|/n)$ time complexity.

We have also described several ways in which our approach can be integrated into existing systems, such as MPI, and have demonstrated theoretically and empirically a trade-off between the number of available buffers and the needed number of barrier synchronizations.

We tested our approach on a suite of five applications, representing various parallel programming paradigms and approaches. We compared the size of the buffer allocations yielded by our approach versus just applying the DFBA algorithm to the entire communication graph. We also investigated the overhead involved in collecting the data to construct the communication graph, and the overhead due to the inter-super-epoch barrier synchronizations. We also compared two different approaches to performing the barrier synchronizations. Lastly, we looked at how application performance can be improved by increasing the per-process buffer allocations.

First, we showed that the new approach yielded very memory efficient buffer allocations that comprised a small fixed number of buffers per process. Second, we showed that by grouping epochs into super-epochs, the number of barrier synchronizations can be greatly reduced. In some cases the reductions were on the order of one or two magnitudes. Third, the data collection phase did not impose a noticeable overhead on the runtime of the applications. Fourth, we showed that the overhead imposed by barrier synchronizations was reasonable. In the worst case the overhead was at most twice the runtime with no synchronization—this was due to a reduction in the concurrency of the application. Finally, we showed that increasing the buffer allocation can improve performance. In particular, the latter two-fold overhead is eliminated by increasing the minimum buffer allocation by adding two buffers to each process.

Future Work.

Currently, our analysis decomposes a super-epoch into a head and a tail. However, a finer decomposition of the tail portion into simple and complex portions may result in even tighter bounds. It would be interesting to investigate the nature of these optimizations.

The worker farm is a commonly use parallel programming paradigm in which a master process divides work among the workers and then waits to receive the results, perhaps issuing more work to follow. The master process is most easily implemented using a wild-card in the MPI_Recv in order to receive from any worker. Unfortunately, our present system cannot handle any kind of wild-cards. However, since the work is typically evenly divided among the workers, the workers will tend to finish and contact the master in roughly the same order as they began. Thus, even though the communication pattern is not static. For all intents and purposes it could be.

One possible extension to our system is to transform such nonstatic communication patterns into static ones by replacing the wild card with explicit process identifiers. This can be accomplished by having the system automatically substitute the wild card MPI_RECV_ANY in a MPI_Recv call with the ID of the process that performed that performed the corresponding MPI_Send during the collection phase. Thus, system would ensure that the receives always occur in the same order, transforming a nonstatic communication pattern into a static one.

In any kind of grid-like computation, the same set of communications are performed within a loop, resulting in a regular communication graph. Presently, our system cannot distinguish this common feature and generates large communication graphs and super-epoch schedules. Furthermore, increasing the number of iterations necessitates the regeneration of a new communication graph. If explicit epoch boundaries could be annotated on the graph, this would reduce the size of the communication graphs and epoch schedules, and would also make it possible to increase the number of computations that an application performs without having to regenerate the communication graph. There are two complementary approaches that we could take.

The first, and easier, of the two approaches is to have the user annotate their program with calls to our system’s library, informing it when a loop iteration commences or terminates. In fact, through use of macros, it may be possible to have the C preprocessor annotate the program for the user. Alternatively, it may be possible to use pattern recognition approaches to identify loop iterations. However, it is not clear if there is a deterministic way to ascertain when a loop has terminated. Regardless, either approach would allow a much more concise representation of the epoch schedules and would vacate the need for the user to regenerate the communication graphs in many instances.

Acknowledgments

This work was supported by a Discovery grant from the Natural Sciences and Engineering Research Council of Canada. We wish to acknowledge Trevor Wilcox for granting us exclusive access to the Nevada Radiological Computational Center’s Beowulf cluster, located at the College of Engineering at the University of Nevada, Las Vegas, to perform our evaluations.
Appendix A. Limited Buffer Size

One of our assumptions is that all message are the same size. This is a simplifying assumptions, since in practice message sizes range from a few to millions of bytes. If a system on which the buffers are allocated has an upper limit on the size of a single buffer, say $s_{\text{max}}$, it may need to send multiple message segments if a message’s size $s$ is greater than $s_{\text{max}}$. This can be modeled by applying a simple transformation to the communication graph $G$ before the buffer allocation is computed. That is, each arc that corresponds to a communication of size $s$ is replaced by $\lceil s/s_{\text{max}} \rceil$ consecutive communication arcs. This preprocessing step can be implemented at either the data collection during the construction of the communication graph. Figure A.1 illustrates the transformation.

![Fig. A.1. Examples of transformations of both simple and complex epochs.](image-url)

Appendix B. Bidirectionality of Communication Arcs

Although communication arcs are drawn as unidirectional with the arrow indicating the receiver, our epoch-based definitions treat the arc as bidirectional. A simple justification is that without buffers, the communication is synchronous, i.e., the send and receive events are corequisite to each other. Thus, the send must precede the receive and vice versa. A more rigorous justification is illustrated in Figure B.1.

Consider a decomposition of the send event into a start-send and finish-send (inject), and a decomposition of the receive-event into a start-receive and an finish-receive (deliver). Fig-
ures B.1a and B.1b exhibit the send and receive events in the normal and decomposed sense.

Now consider the causal order of these events. The start-send must precede the finish-send event and the start-receive event must precede the finish-receive event. Neither the start-send or start-receive events must be preceded by any other of the four events. However, as shown in Figure B.1c, the finish-receive event must be preceded by the start-send event. Thus, if the arcs denote precedence, an arc must extend from the start-send to finish-receive event.

Secondly, assuming that there are no buffers, the finish-send cannot complete until the entire message has been received, i.e., until finish-receive completes. Hence, there is also an arc from finish-receive to finish-send. However, since the send events (and receive events) are treated as one event, the two unidirectional arcs is treated as a single bidirectional one.

Fig. B.1. Decomposition of send and receive events into two events.