approach for approximating 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 ported. 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.

KEYWORDS: Resource Allocation, Parallel Programming, Message Passing Systems

1 Introduction

As the price of computing power continues to drop and connectivity continues to grow, cluster-based parallel applications continue to proliferate, and are being used on and ported to 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 and are built on top of message passing libraries, such as MPI [2], that manage all the logistics of message passing. Such libraries make the applications easier to write, and more importantly, easier to port to new systems.

Unfortunately, use of these libraries is not sufficient to ensure portability. One common problem is that various platforms allocate different amounts of memory for buffering messages. The problem arises because most parallel applications, for efficiency reasons, overlap computation and communication, by sending messages asynchronously. Unlike synchronous communications, which blocks the sender until the receiver is ready to participate in the communication, asynchronous communication does not block the sender if the receiver is not ready. Instead the message is buffered by the message passing library, and delivered when the receiver is ready. Unfortunately, if the buffers become exhausted, the communication ceases to be asynchronous, blocking the sender until the receiver is ready or more buffers become available. If the application’s design does not account for this, the application can deadlock.

Consequently, a presumably correct application can deadlock if the platform to which it is ported allocates fewer buffers than the one for which it was designed. Determining the minimum number of buffers necessary (solving the Buffer Allocation Problem) and modifying the application if fewer buffers are available mitigates the potential for deadlock. However, determining the buffer requirements of an application is intractable [1], and it is not obvious how the application should be modified given the memory limitations of the target platform. In this paper we propose an integrated approach for solving both of these problems.

An execution of an application consists of the local computation of each process and the communication that takes place between the processes. The latter is represented by a directed graph, called a communication graph, in which the vertices represent send and receive events and the arcs represent the causal dependencies between them. Since the graph characterizes the interprocess communication of the entire application (not just a single execution), we assume that the send and receive events, and their causal order is independent of program input. Even with this restriction, the class of applications to which our technique is applicable is substantial. Furthermore, since most one-to-many, many-to-one, and many-to-many communication operations can be implemented using point-to-point operations, we only consider applications that use point-to-point communication. Determining the minimum number of message buffers necessary to prevent deadlock, an NP-hard problem, is accomplished by analyzing an application’s entire communication graph.

Our new integrated approach partitions the application’s execution into epochs, interspersing a barrier syn-
As a result, i) each portion of the execution can be analyzed separately using a small subgraph of the communication graph, ii) the minimum number of message buffers for the entire application is equal to that of the epoch with the largest minimum requirement, and iii) by varying the size of the epochs, the application developer can adjust the minimum number of message buffers that the application requires.

The next section presents the related work and provides a context for our results. Section 3 describes the model and definitions. Section 4 presents our approach, first providing an overview, and then detailing each of its parts. Next, Section 5 describes various ways in which our approach can be applied, and Section 6 evaluates and compares our approach to previously known ones. 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 the developer to fine-tune their application. Lastly, Section 8 summarizes our results and describes some ongoing and future work.

2 Related Work

The notion of a communication graph can be traced back to Lamport’s time-space diagrams [3]. In Lamport’s framework, graph vertices were used to represent receive, send, and internal events. For our purposes, we dispense with internal events. We also make use of an idea of Cypher and Leu [4, 5] who decomposed the send event into a POST-SEND and WAIT-FOR-BUFFER-RELEASE, corresponding to the start and return of the send operation, and decomposed the receive event into a POST-RECEIVE and WAIT-FOR-RECEIVE-TO-BE-MATCHED corresponding to the start and return of the receive operation. The former two correspond to the start of a send event and the injection of the message into the communication system, and the latter two correspond to the start of the receive and the delivery of the message. A process that starts a send, but cannot inject the message into the system, becomes deadlocked, implying that the corresponding program is not safe.

The MPI standard [6, Section 9.2] calls a program that requires no message buffers safe and portable. A more general notion of safety, called $k$-buffer correctness, was introduced by Bruck et al. [7], where an application is considered safe if $k$ message buffers per process is sufficient to prevent deadlock. Brodsky et al. [1] defined an even more general notion, called $k$-safety, where $k$ message buffers are distributed among the processes. The vector $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ encodes the distribution, where $\beta_i$ message buffers are allocated to the $i$th process and $k = \sum \beta_i$.

Brodsky et al. [1] defined the Buffer Allocation Problem, the problem of determining the minimum number of message buffers required to ensure a deadlock-free execution. They showed that both this problem and the seemingly simpler problem of determining if a given buffer allocation is sufficient to avoid deadlock were intractable. However, a slightly different problem known as the Delay-Free Buffer Allocation Problem was shown to be tractable and an algorithm was given. This problem imposes a stricter requirement: That no send operation ever blocks due to lack of message buffers. Their algorithm is an important component of the new approximation.

3 Preliminaries

The Model. Our process model assumes the properties present in cluster-based parallel applications, such as those written with MPI [2]. Specifically, we assume that $n$ processes run asynchronously, do not fail, and do not have access to a global clock. Processes can synchronize with each other by sending and receiving messages, or by using a barrier—a barrier can be implemented using a series of point-to-point communications. For conciseness, we call a parallel application a program. A program execution is a causal ordering of all events in the $n$ process executions.

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. Thus, if the destination process is not ready to receive the message and a buffer is available, an asynchronous communication can ensue. 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. We call send and receive operations events, where each event is defined by the type of operation, the source, and the destination of the message. We assume that all messages are of fixed size, and that messages greater than the fixed size are divided into fixed-size chunks.

Lastly, 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. 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 $i$ will always be to send a message to process $j$, regardless of the problem instance.

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, and a delay-free allocation ensures that no send operation blocks during an execution.
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. 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 send 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 $C$. An arc, $(v_{i,c}, v_{i,c+1}) \in P$, $0 \leq c < e_i$, represents a computation within process $i$ and an arc $(v_{i,c}, v_{j,d}) \in C$ represents a communication between different processes, $i$ and $j$, where $v_{i,c}$ is a send vertex, and $v_{j,d}$ is a receive vertex (see Figure 1). Note, the process arcs are drawn without orientation for clarity; they are always oriented downwards.

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

### Ordering Events

A communication graph is an encoding of the causal ordering of all events in a trace. 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 until the corresponding receive event $r$ occurs, these events comprise a synchronization between the participating processes. Thus, the 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 $v_s \leftrightarrow v_r$.

### 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

1. $E$ contains at least one send or receive event,
2. if $a \in E$ and there exists $b \in S$ such that $a \rightarrow b$ (more concisely $a \rightarrow b$), then $b \in E$,
3. if $a, b \in E$ then $a \rightarrow b$.

This implies that all epochs are disjoint and that each epoch contains at least one send and one receive event. 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 $i \neq j$.

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. 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 minimum buffer allocation.

### 4 Main Result

Brodsky et al. [1] describe an approach for bounding the number of buffers that each process must be allocated in order to ensure a deadlock-free execution. This bound, is equal to the number of buffers necessary to ensure a delay-free execution, which can be determined efficiently from the communication graph. Delay-freedom is ensured if a buffer is available for each send operation in the execution and since delay-freedom is a stronger condition than deadlock-freedom, the latter is ensured if the former is achieved.

Unfortunately, in many cases the number of buffers required to ensure delay-freedom greatly exceeds the number of buffers necessary to prevent deadlock. A good example of this (see Figure 2) 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 number of buffers required to ensure that each epoch is deadlock-free, and using the maximum buffer allocation for each process over all epochs as the bound. In the above example, each of the $k$ epochs contains a single send (and receive) event. Thus, the original approximation on each epoch yields an allocation of one buffer per epoch. Since all epochs are the same, the new approximation yields a buffer allocation of one buffer for the entire program, which is sufficient to ensure deadlock freedom.\footnote{In fact, no buffers are necessary in this case.}

This approach hinges on the assumption that all processes complete an epoch before beginning the next one, ensuring that all buffers are freed 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 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. This can be achieved in two ways. First, we identify which epochs require at least one buffer in order to ensure deadlock-freedom. For these epochs we allocate buffers using the original approximation method. 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. In the example above our approach correctly identifies that no buffers and no 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 number of buffers that need to be allocated to ensure a deadlock-free execution. This is accomplished by setting maximum thresholds for 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 i) how to partition the communication graph into epochs and why interspersing synchronization points between them does not affect program correctness; ii) how to compute (or bound) the buffer allocation for each epoch; and iii) which epochs must be followed by barrier synchronizations, how epochs are merged into super-epochs, and how to determine their corresponding buffer allocations.

**Partitioning the Communication Graph.** We first describe how to partition a communication graph $G$ into epochs, $E_1, \ldots, E_m$. We use a greedy algorithm to perform the decomposition, starting at the top of the graph and working down. The approach “peels off” one epoch at a time, removing the corresponding vertices from the graph.

The algorithm (see Algorithm 1) uses a queue $Q$, a vertex list $L$, and an epoch list $R$: $Q$ holds simple epochs, $L$ holds vertices that have so far been examined by the algorithm, and $R$ holds epochs that have been “peeled off.” Initially, $R$ is empty, the vertices $v_{i,1}, 1 \leq i \leq n$, are inserted into $L$ and all simple epochs whose vertices are in $L$ are inserted into $Q$.\footnote{For our purposes, the start vertices are ignored.} The list $L$ contains the “front” of vertices from which the next epoch is “peeled off” and must satisfy the following invariant at the beginning of each iteration:

$$L = \{v_{i,c} \mid v_{i,c} \in G \land \forall_{0 < d < c} v_{i,d} \notin G\}. \quad (1)$$

In each iteration the algorithm first checks if $Q$ is empty, if not, a simple epoch $E = (v_{i,c}, v_{j,d})$ is dequeued (line 3). Otherwise, if $Q$ is empty, no simple epochs can be peeled off. Hence, the next epoch to be peeled off must be complex. In this case, a vertex, $v$, is selected from $L$ such that if any other vertex in $L$ precedes $v$, then $v$ also precedes that vertex (line 5). That is, all other vertices in $L$ that precede $v$ must be in $v$’s epoch. A vertex in $L$ that precedes $v$ but is not in $v$’s epoch must belong to an epoch that precedes $v$’s, and hence must be chosen first. Line 6 computes $v$’s epoch, $E$, via reachability and breadth-first search.

Next, epoch $E$ is appended to $R$ (line 8), and its vertices are removed from $L$ (line 9). Line 10 assigns to $F$ the “follow” set comprising all vertices in $G$ that are adjacent to, but not contained in $E$. These vertices replace (line 11) those that were removed from $L$, ensuring that invariant 1 holds once $E$ is removed from $G$ (line 12).

Lastly, the vertices in $F$ are evaluated (line 14): if a vertex in $F$ is adjacent to a vertex in $L$ then it and the vertex comprise a simple epoch. Thus, all such epochs are added to $Q$.
Algorithm 1: Partition of $G$ into Epochs

Input: $G$
Output: $R$
Local: $L = \{v_{i,1} \mid v_{i,1} \in G, 1 \leq i \leq n\}$
$Q = \{(r,s) \mid (r,s) \in G \land r,s \in L\}$

1. while $L \neq \emptyset$
   2. if $Q \neq \emptyset$
      3. Dequeue $E$ from $Q$
      4. else
         5. Select $v \in L : \forall u \in L (u \rightarrow v) \Rightarrow (v \rightarrow u)$
         6. $E \leftarrow \{u \mid u \in G \land u \leftrightarrow v\}$
      7. end
   8. Append $E$ to $R$
   9. $L \leftarrow L - E$
10. $F \leftarrow \{u \mid u \in G \land \exists v \in E : (v,u) \in G\}$
11. $L \leftarrow L + F$
12. $G \leftarrow G \setminus E$
13. foreach $u \in L, v \in F : (u,v) \in G$ do
      14. if $(u,v) \notin Q$ then Enqueue $(u,v)$ on $Q$
15. end
end

The algorithm iterates until $L$ is empty, namely, there are no more vertices, and hence no more epochs, to peel off. To initialize $L$ and $Q$ takes at most $O(|V|)$ operations since the former is initialized by performing a linear scan of $G$ and the latter is initialized by performing a linear scan of $L$. Observe that each vertex is added to $L$ once and at least two vertices are removed from $L$ during each iteration. Thus, at most $O(|V|)$ iterations are performed. If $Q$ is empty, the search for an epoch takes $O(|V|)$ time, because $G$ is of degree 3. Given $E$, updating $L$, $R$, $F$, $G$, and $Q$ all take $O(|V|)$. In fact, since $G = E_1 \circ E_2 \circ \cdots \circ E_m$, all update operations combined over all operations take $O(|V|)$ time. Thus, the runtime of the algorithm is $O(|V|^2)$.

More importantly, observe that the epochs are peeled off in a casual order, that is, an epoch whose events casually depend on events in another epoch will not be peeled off prior to that epoch. Thus, $R$ contains the sequence of epochs in a valid casual order. This implies that for any two epochs $E_i$ and $E_j$ in $R$, where $i < j$, there does not exist events $a \in E_i$ and $b \in E_j$ such that $b \rightarrow a$. Consequently, all events in $E_i$ can complete, prior to any events in $E_j$ beginning. One additional feature of the algorithm is that simple epochs take precedence over complex epochs. That is, if a simple epoch $E$ and a complex epoch $E'$ can be causally ordered either way, the algorithm will peel off $E$ before $E'$.

Bounding an Epoch’s Buffer Allocation. To bound the buffer allocation for each epoch we use the algorithm of Brodsky et al. [1] to compute the delay-free buffer allocation for each epoch. Since delay-freedom (no send operation can block) is a stronger property then deadlock-freedom, a buffer allocation that ensures the delay-freedom also ensures deadlock-freedom.

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 empty, that is, it does not have any event vertices, the corresponding process component of $G_E$ consists of a single start vertex. Otherwise, if a process component of $E$ does have send or receive vertices then 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. 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 $G_E$ is determined, and for each graph the corresponding delay-free (and hence deadline-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. Namely, we need to ensure that $\beta_j \geq \beta'_j$ for all $i = 1 \ldots m$ and $j = 1 \ldots n$. That is, each process is guaranteed to have at least as many buffers as it needs in each of its epochs. Thus, taking the element-wise maximum of the $m$ epoch buffer allocations, $\beta = \max \{\beta'\}$ yields the buffer allocation for $G$

If we were to apply this new approach to the program represented by the communication graph in Figure 2, 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 2. In fact, the need for barriers may be obviated, or reduced, by merging many epochs into larger super-epochs. 

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
**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 \( E_0 \circ E_{1} \circ \cdots \circ E_j \), where each epoch may only belong to a single super-epoch. That is, super-epocs 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. 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 \( G \).

A key observation is that a communication graph comprising only simple epochs is acyclic. Consequently, by Lemma 6.3 in [1], 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 the corresponding buffer allocation \((0, \ldots, 0)\).

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 algorithm of [1]. 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 \( E_0 \circ \cdots \circ E_{k-1} \) 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.

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 \(^3\). 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 follows directly from 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 each process component of \( \Xi \). Let \( E \) be a simple epoch, let \( \Xi' \equiv E \circ \Xi \), 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:** (sketch) 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 denoting when a send commences, when the message is injected into the system, when a receive commences, and when the message is delivered. Let \( \varepsilon \) be the subsequence of events in \( \alpha' \) corresponding to the send and receive operation in epoch \( E \), and let \( \alpha = \alpha' - \varepsilon \), 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, \( \varepsilon \) comprises exactly one of the four possible events. Now consider the suffix of the event sequence \( \alpha' \), beginning after the last event in \( \varepsilon \). 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. \( \blacksquare \)

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 this is depicted in Figure 3: super-epoch \( \Xi_0 \) has a corresponding delay-free (and deadlock-free) buffer allocation \((1,1)\), and super-epoch \( \Xi_1 \) has a deadlock-free buffer allocation \((0,0)\). However, the element-wise sum, \((1,1)\), 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 be allocated two message buffers if it is to be composed with a preceding complex super-epoch. That is, its corresponding buffer allocation must be delay-free as well as deadlock-free to ensure correctness.

Thus, the deadlock-free buffer allocation bound for a super-epoch is simply the deadlock-free buffer allocation bound for the tail of the super-epoch.

---

\(^3\)It is actually possible to improve on this bound by decomposing the tail into simple and complex parts, and analyzing the dependencies between them. This analysis will be presented in the journal version of this paper.
Computing Super-Epochs. To compute super-epochs we use a two-pass approach (see Algorithm 2). The first pass computes the maximum delay-free buffer allocation, $\beta$, over all epochs in the sequence, $R$, generated by the partitioning algorithm. The second pass uses this maximum to determine the size of the super-epochs. This pass partitions the sequence into super-epochs each of which comprises a maximal sequence of simple epochs followed by at most one complex epoch. Lastly, the second pass greedily assembles the smaller super-epochs into larger ones and appends them to the output list.\(^4\)

The first pass (line 1) computes the delay-free buffer allocation (DBFA), using the algorithm of [1], for each complex epoch in $R$, and computes buffer allocation $\beta$, the maximum (element-wise) over these allocations. This ensures that any epoch in $R$ is allocated sufficient buffers. Since a program is likely to have many epochs of various sizes, a sequence of smaller epochs can be composed to yield larger super-epochs for which the buffer allocation $\beta$ suffices to ensure deadlock freedom. Furthermore, as we have shown in the preceding section, prefixing any number of simple epochs to a super-epoch does not affect the size of the required buffer allocation.

The pass iterates over the sequence $R$. The second pass keeps track of the current super-epoch, $\Pi$, dequeues epochs from $R$, constructs super-epochs, and suffixes them to $\Pi$ as long as the buffer allocation $\beta$ is sufficient to ensure delay-freedom. Once no more super-epochs can be suffixed, $\Pi$ is appended to the list of completed super-epochs and the construction of the next current super-epoch begins. During each iteration, a maximal prefix of simple epochs is removed from $R$, followed by a single complex epoch (lines 5, 6, and 7). The epochs are composed into a single super-epoch, $\Xi$. Note, that on its own, $\beta$ is sufficient to ensure deadlock freedom of the super-epoch because only the complex epoch requires a nonzero buffer allocation.

Next, in line 8, a check is made whether the current super-epoch $\Pi$ can be composed with super-epoch $\Xi$. If so, the composition is done (line 9). Otherwise, the current epoch, $\Pi$, is added to the output list $L$ (line 11) and then $\Xi$ becomes the new current super-epoch (line 12). After the main loop completes, the last current-epoch is added to $L$ (line 15).

Note, given the choice between suffixing a simple epoch to the last super-epoch, until no more can be added, or prefixing them to the next super-epoch, in which case they do not affect the size of its deadlock-free buffer allocation, the algorithm chooses the latter.

The result is an ordered list of super-epochs for which the buffer assignment $\beta$ is sufficient to ensure deadlock freedom. Thus, the buffer allocation $\beta$ is sufficient to ensure a deadlock-free execution of the program corresponding to the original communication graph $G$, provided that barrier synchronizations are inserted between the resulting super-epochs instead of between every two epochs in the application.

5 Application Modifications

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 the $i$'th operation of a process occurs in $j$th super-epoch and the $i+1$st operation occurs in $j+1$st super-epoch. Then, a barrier operation needs to occur immediately prior to the $i+1$st operation.

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. 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 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 is invoked. An even better approach would be for the wrapper code to load a partitioned and attributed communication graph and use the information from the graph to determine when to perform synchronizations.

\(^4\)For clarity, the algorithm is presented without any optimizations. These optimizations, and the proofs of correctness and optimality will be presented in the journal version of this paper.
This approach has several benefits. First, it would significantly improve portability by obviating the need to modify the lookup tables every time the application was ported or modified. Second, it also provides the ability to run the same application with varying resource utilizations by simply loading the attributes that correspond to the desired resource usage. Third, this approach could also lend itself to a dynamic deadlock detection system where the communication graph is computed dynamically as the application executes and the super-epochs are computed and partitioned on an ongoing basis.\footnote{We are presently investigating this approach.}

6 Evaluation

In this section we evaluate our approach vis a vis the delay-free approach and the optimal solution. We select five well-known communication patterns, determine the optimal and approximate buffer allocations using the delay-free and our new approach, and compare the resulting allocations. For completeness, we also detail the number of epochs and super-epochs that an execution of a communication pattern would be partitioned into. Recall that the delay-free approximation simply computes the delay-free buffer allocation of the corresponding communication graph. Since the selected patterns are relatively simple, it was possible to analytically determine the general optimal buffer allocation for each of the five patterns. However, in the general case, this is not feasible. The selected patterns are:

Master/Slave: This pattern comprises a single master and \( n \) slave processes. The master process sends a message to each slave, delegating some amount of computation. Each slave performs the work and sends back one message to the master after the work is completed. Typically, this pattern may be repeated for an arbitrarily many rounds. For our purposes we only consider one round of the pattern.

Torus: This pattern comprises \( n \) processes that are arranged in a 2-dimensional \( \sqrt{n} \times \sqrt{n} \) grid in which each process has four neighbours, typically called: north, south, east, and west. The grid “wraps around” such that processes on the boundary of the grid have neighbours on the opposite boundary of the grid. Each process sends a message to its east neighbour and receives a message from its west neighbour, and then sends a message to its north neighbour and receives another message from its south neighbour. Typically this communication is repeated for \( \sqrt{n} - 1 \) rounds.

Ring: This pattern comprises \( n \) processes arranged in a ring. Each process has a single left and a single right neighbour. In such a pattern, each process sends a message to their left neighbour and then receives a message from their right neighbour. This is typically repeated for \( n - 1 \) rounds.

Pipe-and-Roll. Used in matrix multiplication algorithms, this pattern comprises one coordinator process and \( n \) worker processes that are arranged 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, sending \( \sqrt{n} - 1 \) messages, one to each process in its row. 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 of the initiator in the preceding round. For example, suppose that nine processes are arranged in three rows of three processes each.

\[
P_0 \quad P_1 \quad P_2 \\
P_3 \quad P_4 \quad P_5 \\
P_6 \quad P_7 \quad P_8
\]

The first round will have initiators \( P_0, P_4, P_8 \), second round \( P_1, P_3, P_5 \), and finally \( P_2, P_3, P_7 \).

All-to-All: This pattern comprises \( n \) processes, each of which sends one message to each of the \( n - 1 \) processes and then receives \( n - 1 \) messages from all the processes. Although this pattern can be performed an arbitrary number of times during an execution, for our purposes we consider only one round of the pattern.

Table 1 summarizes the information about each of the five selected patterns and lists the approximated and optimal buffer allocations for each of them. For clarity, in two cases, we give a lower bound on the buffer allocation approximation rather than an explicit buffer allocation. Furthermore, in several cases, such as the Ring pattern, there is more than one optimal buffer allocation. For example, for the Ring pattern, exactly one buffer is needed on one of the processes, it does not need to be the first, and for the Torus and the Pipe-and-Roll patterns, buffers must be arranged such that each column and each row has at least one process that has a buffer.

Table 1 illustrates two important points. First, that in many cases the number of super-epochs is much smaller than the number of epochs. This means that in these cases the cost of performing barrier synchronization is acceptable, particularly if the latency of the interconnect, such as in a cluster, is relatively small. In fact, in most cases, the computation time and the data transmission time will dwarf the barrier synchronization time.

Second, and more important, in almost all cases the new approximation is significantly better than the delay-free one, resulting in buffer allocations that allocate a constant number of message buffers per process. In fact, in all but one case the approximation differs by a constant number of buffers (per process) from the optimal solution, and in a couple cases the approximation is the optimal solution.

Interestingly, the only pattern for which the new approximation does no better than the delay-free approximation is the All-to-All pattern. However, it is important to
note that in this case the delay-free approximation is also an optimal solution. An important lesson here is that the delay-free approximation is still useful and complements the new approximation.

For all but the All-to-All pattern, the reduction in the number of per process buffers that need to be allocated, with respect to the difference between the delay-free approximation and the new approximation, is of the order of $O(\sqrt{n})$ to $O(n)$. This can be a sizable memory reduction depending on the size of the message buffers and the number of processes, particularly if each processor runs multiple processes.

### 7 Time/Space Tradeoff

The approach we described in Section 4 attempts to determine the smallest buffer allocation, $\beta$, that suffices for every epoch in the communication graph. Then, super-epochs are created that take maximal advantage of the buffer allocation $\beta$. Thus, the number of barrier synchronizations is determined by the the largest per epoch buffer allocation.

Instead, suppose that the maximum number of message buffers that can be allocated per process is known a priori, say $\beta_{\text{max}}$. Then, this buffer allocation determines the size of the resulting super-epochs. Consequently, the partitioned graph may comprise fewer, but larger, super-epochs, thus requiring fewer barrier synchronizations overall.

This is a general time/space trade-off that is inherent to our approach. A large number of available memory buffers results in a few large super-epochs, few synchronizations between them, and hence a faster run-time. A small number of available memory buffers results in many small super-epochs, some possibly comprising a single complex epoch, many synchronizations between them, and hence a slower run-time.

Such trade-offs are particularly useful since clusters and their resources are typically shared by many users and a single application cannot be allocated all memory resources. In fact, it is foreseeable that an application, using the mechanism described at the end of Section 5 could automatically adjust its resource utilization based on the current load on the cluster.

### 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 deadlock is an important part of developing or porting an application to a new platform. Since the Buffer Allocation Problem is intractable, approximation remains the only recourse.

We presented an epoch-based polynomial-time approximation approach that yields tight bounds in many common cases, and typically performs better, but never worse, than previously known approaches. The approach partitions the application execution into disjoint epochs that are interspersed by barrier synchronization. These ensure that all message buffers can be reused at the start of each epoch, facilitating the determination of buffer allocations on a per-epoch basis. The bound for the application is the element-wise maximum over the buffer allocations over all epochs. Both partitioning the execution and determining a buffer allocation for each epoch can be accomplished in time quadratic in the number of communications performed by the application.

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

### Future Work

For clarity and conciseness we have described algorithms that take $O(n^2)$ run-time in the number of communications. However, we believe it is possible to reduce the complexity of these algorithms to $O(n)$ or $O(n \log n)$ run-time complexity. Presently, the algorithms locate epochs and compute delay-free buffer allocations from scratch during each iteration. By using auxiliary data structures, we believe that the need for these recomputations can be obviated, resulting in a significant speed-up.

Such optimizations would also make it feasible to transform our off-line approach into an on-line one, letting the underlying system deal dynamically with the potential exhaustion of message buffers by throttling the application.
execution with barrier synchronizations if the number of available message buffers drops below a predefined low-water mark. To accomplish this we plan to fully implement our approach within the context of a system, such as MPI.

Lastly, there are many additional analyses that can be performed to reduce the bounds on a per-epoch-basis even further. Furthermore, the current approach requires all processes to synchronize at the end of an epoch or super-epoch. However, this is not necessary in many cases. By reducing the set of processes performing a barrier synchronization to only the necessary ones would further improve the overall application efficiency.

Acknowledgments

This work was supported by a Discovery grant from the Natural Sciences and Engineering Research Council of Canada.

References


