On the Complexity of Buffer Allocation in Message Passing Systems

Alex BRODSKY, Jan Bækgaard PEDERSEN & Alan WAGNER
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada, V6T 1Z4
{abrodsky,matt,wagner}@cs.ubc.ca

Abstract

In modern cluster systems message passing functionality is often off-loaded to the network interface card for efficiency reasons. However, this limits the amount of memory available for message buffers. Unfortunately, buffer insufficiency can cause an otherwise correct program to deadlock, or at least slow down. Hence, given a program trace from an execution in an unrestricted environment, determining the minimum number of buffers needed for a safe execution is an important problem. We present three related problems, all concerned with buffer allocation for safe and efficient execution. We prove intractability results for the first two problems and present a polynomial time algorithm for the third.

1 Introduction

For efficiency reasons, most modern clusters off-load message passing functionality to the network interface card (NIC) [1] to facilitate a greater overlap of computation and communication. Unfortunately, most NICs have two orders of magnitude less memory than the average host, which makes message buffers a limited resource. Thus, programs that use asynchronous message passing and execute correctly otherwise, might dead-lock when executing on a system where parts of the message passing system have been off-loaded to the NIC; such issues have been investigated in [2, 3, 4].

Ideally, we want to detect the possibility of dead-lock and prevent it from happening. The MPI message passing standard [4] defines a “safe” program as one that requires no buffering and uses synchronous communication. Since better performance can be obtained by taking advantage of system or network buffers to reduce unnecessary synchronizations—asynchronous communication—the notion of a “k-safe” (asynchronous) program arises. A “k-safe” program requires k buffers to guarantee completion [1, 5, 6].

Unfortunately the value of k is usually not known a priori. In this paper we investigate the complexity of determining a minimum value of k for programs that use asynchronous buffer communication and have static communication patterns. Under these assumptions we show that the problem of determining k, the minimum number of buffers for deadlock free execution, is **NP**-hard. Furthermore, we show that the simpler task of verifying whether a buffer assignment is sufficient to avoid deadlock is **coNP**-complete. Finally, we give a polynomial time algorithm for determining the exact number of buffers needed to ensure nonblocking sends, achieving efficient program execution. Interestingly, the nonblocking requirement makes the problem tractable! Motivations and precise definitions of the system and the problems considered are given in the following sections.

2 Background

Determining the k-safety of a program is important for several reasons. First, the semantics of asynchronous communication depend on the availability of buffers, and change—in an implementation dependent fashion—when buffer space becomes exhausted. Since buffer space is typically limited, or pre-allocated during application initialization, a priori knowledge of the application's buffer requirements ensures that communication semantics do not change part way through the execution. Second, to improve performance, many systems use zero-copy techniques: messages are transferred directly from the NIC to application buffers. Since these buffers must be pre-allocated by the application, the application must know the number required. Third, one of the most common purposes of parallelization is to enable the solutions of bigger problems on larger data sets. Typically, host memory is the limiting resource and, if the number of buffers needed is known, memory utilization can be improved. Finally, communication libraries like MPI allow the application to manage the buffer space; this can be optimized if the number required is known in advance.

The most natural primitives for asynchronous buffered communication are "nonblocking sends" and "blocking receives"; these are also the standard communication primitives in MPI [4] and PVM [7]. Cypher and Leu formally define the former as a *POST-SEND* immediately followed by a *WAIT-FOR-BUFFER-RELEASE* and the latter as a *POST-RECEIVE* immediately followed by a *WAIT-FOR-RECEIVE-TO-BE-MATCHED* [8, 9]. Informally, the send blocks until the message is copied out of the send buffer and the receive blocks until the message has been copied into the receive buffer. We only consider point-to-point communication similar to the model used in [10], since multicast and broadcast communication can be simulated with point-to-point communication.

A multiprocess system S is a set of simultaneously executing independent asynchronous processes that perform a computation by interspersing local computation and point-to-point message passing between processes; these are referred to as *A-computations* in [10]. Such a system is equivalent to the system with three different events like the one defined by Lamport [11]: send events, receive events and internal events. Send events cause messages to be sent, receive events causes messages to be received and internal events represent computation on internal state. The system uses nonblocking sends and blocking receives; processes synchronize by communicating and no assumptions can be made about the computation time of any process.

When a process performs a send, the message may either be sent to the receiving process where it is either received or buffered, or the sending process blocks. If the receiving process is ready to receive (i.e., has issued a receive request for the incoming message), or the message passing system has free buffers available, the sending process does not block. If no buffers are available and the receiving process is not ready to receive, the sending process blocks until one of the conditions changes. These communication primitives are comparable to the default send/receive primitives found in PVM and MPI: blocking receives and nonblocking sends when buffers are available and blocking when not.

In such a model each process has a pool of buffers for incoming messages used on a first come first served basis; we call these **receive buffers**. Buffers are used when a message is not ready to be received by the receiving process. Buffers are returned to the receiving process's buffer pool when the message has been received (i.e., the corresponding receive request has completed).

Other models that use only **send buffers**, a combination of send and receive buffers, or buffers that are allocated on a per channel basis, are possible. While the focus of this paper is on the receive buffer model, our results are applicable to other models as well.

In this paper we consider programs that are repeatable [8, 9] when executed in an unrestricted environment, i.e., programs with static communication patterns. While this narrows the class of programs we consider, the class of applications with static communication patters is still considerable, including: grid computations, linear system computations, and pipe line computations. Since the communication pattern must be static we do not consider the case where a receive can specify a wild card, i.e., a receive specifies exactly one sender; regardless of this restriction the problems we consider remain intractable.

A message history M for the execution of a system S is a set of messages of the form $m = (i, s_i, j, s_j)$ where i and j are process identifiers; i represents the sender and j represents the receiver. The sequence numbers s_i and s_j are local to the sending and receiving processes, respectively. For process i the first send or receive has sequence number 1, the second has number 2, and so forth. We use a communication graph $G(S)$, described in the next section, to encode the message history; sends and receives are encoded as vertices, and the sequence numbers are represented by the topological order of the vertices in $G(S)$.

Even though the communication pattern is static, the arrival time of messages varies from execution to execution. The following example shows this effect and how the allocation of buffers can result in deadlock. Consider the three process system depicted in Figure 1 where p_1 and p_3 each have zero buffers available and process p_2 has one buffer available. Two different scenarios exist. In the first scenario, message 1 arrives before message 3 and takes up the buffer. Now processes p_2 and p_3 can never progress as one additional buffer is needed to break the deadlock between them. In the second scenario, message 3 arrives first and takes up the buffer. Process p_3 can proceed to its receive call and block until message 2 arrives. Consequently, process p_2 will send message 2, receive message 1 followed by message 3. Thus, all processes finish and no deadlock occurs. For example, if process p_3 had one buffer rather than process p_2, the system depicted in Figure 1 is guaranteed to complete for all executions.

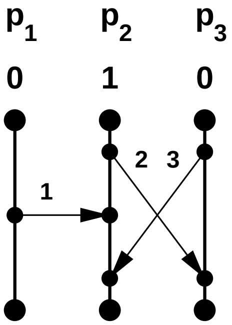

Figure 1: Order of execution can cause deadlock.

As demonstrated, the order in which messages arrive at a process determines whether a system deadlocks. We call a system that exhibits such behaviours **unsafe**. This situation can be rectified by an appropriate allocation of buffers to processes.

To conserve memory, we want to allocate only as many buffers as needed. For programs with static communication patterns, we would like to determine a minimum buffer assignment or to determine whether a given buffer assignment prevents deadlock. As we will show, both of these problems are intractable.

If system S is to execute efficiently, ideally, no send operation blocks. In addition to safety, we like to determine the minimum buffer assignment necessary to prevent blocking sends. Interestingly, the stricter constraint makes the corresponding buffer allocation problem tractable.

Determining whether a system is buffer independent—the system is 0-safe—was investigated in [8, 9]. In our model the interesting systems are buffer-dependent, and require an unknown number of buffers to avoid deadlock. To determine the minimum number of buffers, the execution of a system can be modeled using a (coloured) Petri net [12]. In order to determine if the system can reach a state of deadlock, the Petri net occurrence graph [13] is constructed, and a search for dead markings is performed. However, the size of the occurrence graph is exponential in the size of the original Petri net.

A variation of this problem has been investigated by the operations research community [14, 15, 16]. In these models, events or products are buffered between various stations in the production process, however, the arrival of these events is governed by probability distributions, which are specified a priori. In our model, since processes are asynchronous, the time for a message to arrive is non-deterministic, i.e., a message may take an arbitrarily long time to arrive and a process may take an arbitrarily long time to perform a send or a receive.

3 Definitions

Let S be a multiprocess system with n processes and E_i communication events occurring in process i; a communication event is either a send or a receive. A **communication graph** of S is a directed acyclic graph $G(S) = (V,A)$ where the set of vertices $V = \{v_{i,c} \mid 1 \leq i \leq n, 0 \leq c \leq (E_i + 1)\}$ corresponds to the communication events and the arc set A consists of two disjoint arc sets, the computation arc set P and the communication arc set C. Each vertex represents an event in the system: vertex $v_{i,0}$ represents the **start** of process i, vertex $v_{i,c}$, $1 \leq c \leq E_i$, represents either a **send** or a **receive** event, and vertex $v_{i,(E_i+1)}$ represents the **end** of a process. An arc $(v_{i,c}, v_{i,c+1}) \in P, 0 \leq c \leq E_i$, represents a computation within process i and an arc $(v_{i,s}, v_{j,t}) \in C$ represents a communication between different processes, i and j, where $v_{i,s}$ is a send vertex, and $v_{j,t}$ is a receive vertex (e.g. Figure 2). Note, the process arcs are drawn without orientation for clarity; they are always oriented downwards. In general vertices are labelled with double indices, representing the process label and the sequence number of the corresponding event. The former is dropped when the process label is given by the context. Communication graphs are comparable to the time-space diagrams—without internal events—noted in [11].

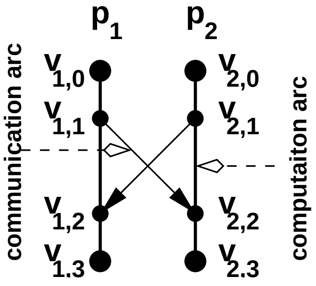

Figure 2: An example of a communication graph with a 2-ring.

A multiprocess system S is **unsafe** if a deadlock can occur due to an insufficient number of available buffers; if S is not unsafe, then S is said to be **safe**. Figures 1 and 3 are examples of unsafe systems. Note: the numbers above the graph in Figure 3 represents the buffer assignment.

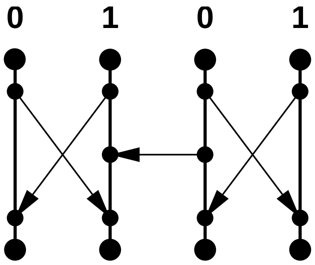

Figure 3: Order of buffer selection can cause deadlock.

A **buffer assignment** is an n-tuple $B = (b_1, b_2, ..., b_n)$ of non-negative integers that represent the number of buffers that can be allocated by each process. Since buffers use up memory, which may be needed by the application, ideally, as few buffers as possible should be allocated. However, allocating too few buffers can result in an unsafe system.

Two natural decision problems arise from this optimization problem. Given a communication graph $G(S)$ and a non-negative integer k, the **Buffer Allocation Problem** (BAP) is to decide if there exists a buffer assignment $B = (b_1, b_2, ..., b_n)$ such that S is safe and $\sum_{i=1}^{n} b_i \leq k$. In order to solve this problem we need to solve a simpler one. Suppose we are given a buffer assignment $B = (b_1, b_2, ..., b_n)$ and a communication graph $G(S)$, the **Buffer Sufficiency Problem** (BSP) is to decide if the assignment is sufficient to make S safe.

Additionally, we can require that no process in system S should ever block on a send. Given a communication graph $G(S)$ and a non-negative integer k, the **Nonblocking Buffer Allocation Problem** (NBAP) is to decide if there exists a buffer assignment $B = (b_1, b_2, ..., b_n)$, such that no send in S ever blocks, and $\sum_{i=1}^{n} b_i \leq k$. Next, we introduce the terminology used throughout the paper.

3.1 Terminology

The ith **process component** $G_i(S)$ of $G(S)$ is the subgraph $G_i(S) = (V_i, A_i)$ where $V_i = \{v_{i,c} \in V \mid 0 \leq c \leq (E_i + 1)\}$ and $A_i = \{(v_{i,c}, v_{i,c+1}) \in A \mid 0 \leq c \leq E_i\}$. The process component corresponds to a process in S. We construct communication graphs by connecting process components with arcs. Hence, it is more intuitive to treat a process component as a chain of send and receive vertices bound by a start and an end vertex.

A **t-ring** is a subgraph of a communication graph $G(S)$, consisting of $t > 1$ process components such that in each of the t process components there is a send vertex s_{i_j,c_j} and a receive vertex r_{i_j,d_j}, $c_j < d_j$, $1 \leq j \leq t$ such that the arcs $(s_{i_1,c_1}, r_{i_t,d_t})$ and $(s_{i_{j+1},c_{j+1}}, r_{i_j,d_j})$, $1 \leq j < t$, are in A. This definition is equivalent to the definition of a "crown" in [10].

A t-ring is a circular dependence of alternating send and receive events; an example of a t-ring is illustrated in Figure 4. As the shaded arcs in Figure 4 show, each receive event depends on the preceding send event and each send event depends on the corresponding receive event. Thus, without an available buffer, there is a circular dependency that results in the system deadlocking.

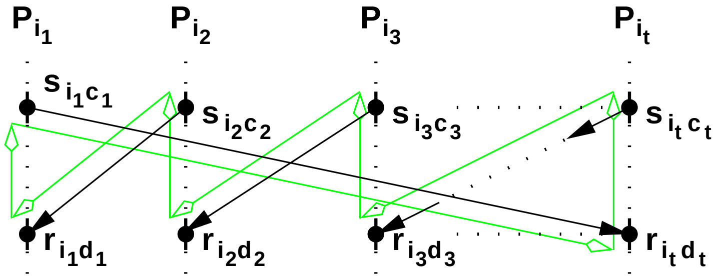

Figure 4: Dependency cycle in $G(S)$.

Therefore, a system S whose communication graph $G(S)$ contains a t-ring will deadlock unless one of the processes in the t-ring has an available buffer. Since $G(S)$ must be a DAG, no cycle will ever occur in $G(S)$. Figure 2 shows an example of a 2-ring. If $G(S)$ contains a t-ring, then we say that system S also contains a t-ring.

In this paper we use $G(S)$ rather than the dependency graph because the order in which received messages are allocated to buffers depends on the execution history. In order to model the execution of a system we define a colouring game that simulates the execution of the system with respect to $G(S)$.

3.2 Colouring the Communication Graph

Given a communication graph $G(S)$, an execution of a corresponding system S is represented by a colouring game where the goal is to colour all vertices green; a green vertex corresponds to the completion of an event. We use three colours to denote the state of each event in the system: a red vertex indicates that the corresponding event has not started, a yellow vertex indicates that the corresponding event has started but not completed, and a green vertex indicates that the corresponding event has completed. Hence, a red vertex must first be coloured yellow before it can be coloured green; this corresponds to a traffic lights changing from red, to yellow, to green [1].

To represent buffer allocations we use tokens. For each process with a number of allocated buffers, we associate an equal size pool of tokens with the corresponding process component. To represent a buffer allocation, tokens are removed from the process component's token pool and placed on the receive vertices.

The colouring game represents an execution via the following rules. Initially, the start vertices of $G(S)$ are coloured green and all remaining vertices are coloured red; this is called the **initial colouring**.

1. A red send vertex may be coloured yellow if the preceding vertex is green (i.e., the send is ready).
2. A red receive vertex may be coloured yellow if the corresponding send vertex is yellow, and
   (a) the preceding vertex (in the same process component) is green (i.e., both the send and the receive are ready), or
   (b) a token from the corresponding buffer pool is moved on to the vertex (i.e., the send is ready and a buffer is available).
3. A yellow send vertex may be coloured green if the corresponding receive vertex is coloured yellow (i.e., the communication has completed from the senders perspective).
4. A yellow receive vertex may be coloured green if both of its preceding vertices are green. If the vertex has a token, the token is returned to the process component's pool (i.e., a receive completes once the send completes and the preceding computation completes).
5. A red end vertex may be coloured yellow if the preceding vertex is green.
6. A yellow end vertex may be coloured green if the preceding vertex is green.

Buffer utilization is represented by placing a token from the token pool of the process component on the selected vertex, and colouring it yellow. If no tokens are available, the rule cannot be invoked.

A **valid colouring** of G, denoted χ_G, is a colour assignment to all vertices that can be obtained by repeatedly applying the colouring rules, starting from the initial colouring. A **colouring sequence** $\Sigma = (\chi^1, \chi^2, \ldots)$ is a sequence of valid colourings such that each colouring is derived from the preceding one by a single application of one of the colouring rules. An execution of a multiprocess system S with buffer assignment B is represented by a colouring

[1] Naturally, we refer to a European traffic light.

sequence on $G(S)$. We say that a colouring sequence **completes** if and only if the last colouring in the sequence comprises only green vertices. A colouring sequence **deadlocks** if and only if the last colouring in the sequence has one or more non-green vertices and the sequence cannot be extended via the application of the colouring rules. Furthermore, each transition, from one colouring to the next, within a colouring sequence, corresponds to a change of state of an event in the corresponding execution.

Assuming that all events in the system are ordered, there is a correspondence between the colouring sequences on $G(S)$ and the executions of system S. A system S is safe if and only if every colouring sequence on the graph $G(S)$ completes. Furthermore, sends in S never block if and only if every partial colouring sequence on $G(S)$ that ends with a colouring containing a yellow send vertex and a corresponding red receive vertex can be extended by applying rule 2 to the red receive vertex. The choice of when to apply rule 2b affects future choices. For example, in Figure 3, applying the rule to the receive vertex corresponding to the send from process p_1, before the send vertex in process p_3 is coloured green, results in a deadlocked colouring sequence.

4 The Buffer Allocation Problem

In order to prevent deadlock in distributed applications the underlying system needs to allocate a sufficient number of buffers. Ideally, it should be the minimum number required. Unfortunately, the corresponding decision problem, BAP is intractable: given a communication graph $G(S)$ and a positive integer k, determine whether there exists a buffer assignment of at most k buffers such that S is safe. We show that BAP is **NP**-hard by a reduction of the well known 3-SAT problem[17] to BAP. Recall the definition of 3-SAT: determine if there exists a satisfying assignment to $\bigwedge_{i=1}^{n}(a_i \vee b_i \vee c_i)$, where a_i, b_i, and c_i are boolean literals in $\{x_1,\overline{x_1},x_2,\overline{x_2},\ldots,x_n,\overline{x_n}\}$. We first need the following lemma.

Lemma 4.1 (The t-Ring Lemma) *Let G be a communication graph comprising a single t-ring. No colouring sequence on G can complete without invoking rule 2b at least once.*

Proof: Assume by contradiction that there exists a complete colouring sequence Σ that does not make use of rule 2b. Consider the first colouring in Σ where one of the sending vertices is green; call the vertex s_i. Let r_j be the corresponding receive vertex. By rule 3, the vertex r_j must be yellow. Since rule 2b has not been applied, rule 2a must have been invoked earlier in the sequence. By the definition of a t-ring, the send vertex s_j must be the predecessor of r_j. Since the rule 2a was applied to r_j, s_j must be green. Hence, there is an earlier colouring in Σ with a green send vertex. This is a contradiction. ■

Theorem 4.2 *The Buffer Allocation Problem (BAP) is* **NP**-*hard.*

Proof: We prove this by reduction of 3-SAT to BAP. For any 3-SAT instance F we construct a corresponding system S and the corresponding communication graph $G(S)$, both which are polynomial in the size of F.

The system has $2n+1$ processes, where n is the number of variables. Each process contains $c+1$ epochs where c is the number of clauses in F. An epoch is a consecutive sequence of one or more events in a process. An epoch terminates on a send to a barrier process, or when the process terminates. An epoch begins on a receive from a barrier process or when the process starts. A barrier process is used to synchronize all processes at the end of its epoch. Each process performs a send to the barrier process at the end of their epoch, and waits for a response from the barrier process. The barrier process sends the response to every process only after it has received a message from each process. An epoch of a process component is correspondingly defined. Epochs are used to prevent unwanted interaction between processes.

For each literal x_i and $\overline{x_i}$, let the system S contain processes p_{x_i} and $p_{\overline{x_i}}$. In addition, let p_{barrier} denote the barrier process in S.

Epoch 0 is used to fix a buffer assignment corresponding to a variable assignment in 3-SAT. In epoch 0 add a 2-ring between processes p_{x_i} and $p_{\overline{x_i}}$. This corresponds to fixing an assignment, because by Lemma 4.1, we have to assign a buffer to either p_{x_i} or $p_{\overline{x_i}}$ to prevent deadlock (see Figure 5). Next, every process p_{x_i} and $p_{\overline{x_i}}$, $1 \leq i \leq n$, performs a send to process p_{barrier}. After p_{barrier} receives from all processes, it performs a send to all processes, allowing them to proceed into the next epoch.

The jth epoch of each process corresponds to the jth clause of F and is a 3-ring on the processes p_{a_j}, p_{b_j}, and p_{c_j} that correspond to the literals a_j, b_j, and c_j (see Figure 6). By Lemma 4.1, in order to avoid deadlock, at least one of

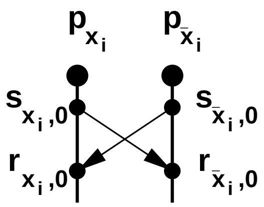

Figure 5: The choice widget.

the three processes, p_{a_j}, p_{b_j}, or p_{c_j}, must have a buffer. Finally, at the end of the epoch, all processes perform a send to the process p_{barrier} and wait for a reply. This is formalized in the following Lemma.

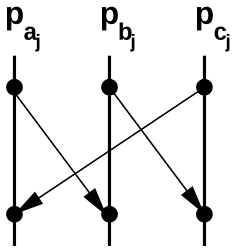

Figure 6: The clause widget.

Lemma 4.3 *There exists a token assignment of size n such that all colouring sequences on $G(S)$ complete if and only if formula F is satisfiable.*

Proof: Let $s_{x_i,0}$, $s_{\bar{x}_i,0}$, $r_{x_i,0}$, and $r_{\bar{x}_i,0}$ be the send and receive vertices in epoch 0. By Lemma 4.1, to colour vertices $s_{x_i,0}$ and $s_{\bar{x}_i,0}$ green, a free token must be available at either process component p_{x_i} or $p_{\bar{x}_i}$. Hence, we need at least n tokens. Note, at the end of each epoch, each token is released back into its token pool.

As long as all vertices in each epoch of each process component can be coloured green, the barrier component can also be coloured green. A colouring sequence will only deadlock in the barrier if the corresponding send vertex in one of the process components can not be coloured yellow.

If F has a satisfying assignment, then at least one literal in every clause will be true. A corresponding token assignment will ensure that each 3-ring has at least one process component with a token (one corresponding to a true literal). Hence, by Lemma 4.1 none of colouring sequences will deadlock on any of the the 3-rings. Hence, any colouring sequence on $G(S)$ will complete.

If F does not have a satisfying assignment, then for any assignment there exists at least one clause comprising false literals. The corresponding token assignment will not assign any tokens to the process components in the corresponding 3-ring. Thus, by Lemma 4.1 all colouring sequences will deadlock in that 3-ring. Further, none of the colouring sequences on $G(S)$ will complete. Hence, any colouring sequence on $G(S)$ will complete if and only if the corresponding assignment satisfies F. ■

Hence, there exists a buffer assignment of size n such that S is safe if and only if F is satisfiable. Thus, BAP is **NP**-hard. ■

Theorem 4.4 *The Buffer Allocation Problem (BAP) is in Σ_2**P**.*

Proof: By Theorem 5.1, verifying that a token assignment is sufficient to prevent deadlock (BSP) is **coNP**-complete. Since we can non-deterministically guess a sufficient token assignment, the result follows. ■

The Buffer Allocation Problem remains intractable for systems with send buffers only, and for systems with a combination of both send and receive buffers. In the latter case, the problem remains in Σ_2**P** because the class of systems with receive buffers only is a subclass of systems with both receive and send buffers. In the former case, we conjecture that the problem is **NP**-complete. The **NP**-hardness follows from the observation that each t-ring in the system has to have a buffer assigned to one of its processes in order for the system to progress. It does not matter if it is a send or a receive buffer. Hence, the reduction used in Theorem 4.2 can be applied with no modification.

5 The Buffer Sufficiency Problem

We now turn our attention to the possibly simpler problem of verifying whether a given buffer assignment is sufficient to prevent deadlock. However, this turns out to be an intractable problem as well; we show that BSP is **coNP**-complete by a reduction from the TAUTOLOGY problem [18, Page 261] to BSP. Given an instance of a formula in disjunctive normal form (DNF), $\bigvee_{i=0}^{n} \bigwedge_{j=0}^{l_i} L_{i,j}$ where $L_{i,j} \in \{x_1, \bar{x_1}, \ldots, x_n, \bar{x_n}\}$, the formula is a tautology if it is true on all assignments. An assignment for which the formula is not true is a concise proof that the formula is not a tautology.

Theorem 5.1 *The Buffer Sufficiency Problem (BSP) is* **coNP***-complete.*

Proof: We first observe that if S is unsafe, then there exists a concise certificate of this fact comprising a colouring sequence on $G(S)$ that does not complete. Since the size of a colouring sequence is at most twice the number of vertices in $G(S)$, the certificate is polynomial in size and hence, BSP is in **coNP**.

Let F be a DNF formula with t terms where the ith term has l_i literals. For any formula F, we construct a system S and show that there exists a deadlocking colouring sequence on $G(S)$ if and only if the formula is not a tautology. We represent the disjunction of t terms by a subsystem containing a t-ring on t processes where each term is a subsystem consisting of a single process, called a 'term' process. The communication graphs corresponding to the term and the disjunction are illustrated in Figures 7 and 8 respectively. Following the t-ring, one 'term' process performs a send – called a 'done' send – to signal that the t-ring did not deadlock. Finally, the ith 'term' process performs l_i receives, corresponding to the literals of the ith term; the latter we call 'literal' receives. The first receive in the 'term' process is called a 't-ring' receive.

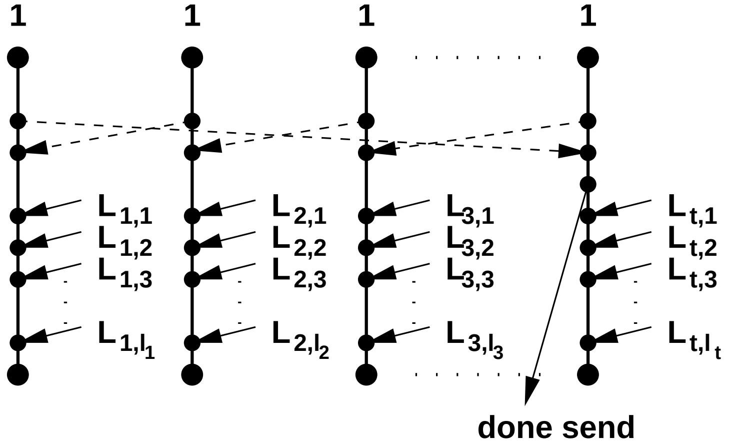

Figure 7: Disjunction subsystem.

When a message to a 'term' process arrives before any send within the t-ring begins, the message is buffered, using up the one available buffer and preventing the process from buffering any additional messages until it leaves the t-ring. This corresponds to the falsification of the corresponding literal and term.

To represent a variable assignment, we use a select subsystem consisting of three processes p_{x_i}, $p_{\bar{x_i}}$, and q_i; the first two processes correspond to the truth values of x_i and are called 'select' processes. The third process, called the 'arbiter', fixes the truth value of x_i. Processes p_{x_i} and $p_{\bar{x_i}}$ each perform a send to the third 'arbiter' process, but the 'arbiter' process can receive neither message until it receives a 'done' message. Since the 'arbiter' has only one buffer, only one of the sends from the processes p_{x_i} and $p_{\bar{x_i}}$ will be buffered; the other send will cause the sending process to block until the 'done' message arrives.

The process that did not block performs a nonblocking send, via a disperser (described below), to each 'term' process whose corresponding term contains the complement of the fixed variable (i.e., 'select' process p_{x_i} sends a message to all 'term' processes whose corresponding terms contain the literal $\bar{x}_i$). Once the 'arbiter' process receives the 'done' signal, all three processes, p_{x_i}, $p_{\bar{x_i}}$, and q_i, can complete without deadlock.

A disperser is a nonblocking subsystem that upon receipt of a 'disperse' message performs a nonblocking broadcast by using additional processes as buffers (see Figure 10).

To construct a multiprocess system S that corresponds to formula F, instantiate a select subsystem for each variable and a disjunction subsystem corresponding to F. The disjunction subsystem performs the 'done' send to a disperser,

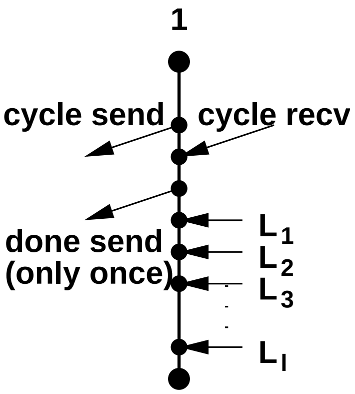

Figure 8: Each term is a process.

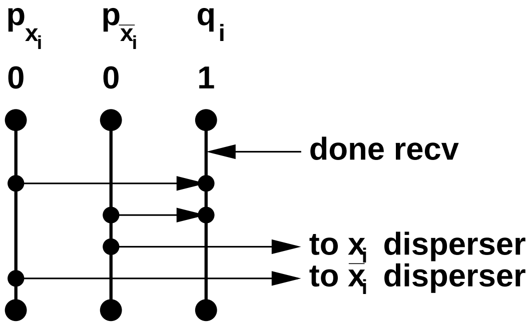

Figure 9: Select subsystem.

which broadcasts it to the select subsystems. Finally, each select subsystem broadcasts its selection, via a disperser, to the corresponding 'literal' receives of the 'term' processes; Figure 11 depicts the corresponding communication graph $G(S)$ of a composition.

Intuitively, if each of the 'term' processes receive a message from a select subsystem – sent via a disperser – before any of the 'term' processes initiate a send event, the buffers on each of the 'term' processes will be used up. By Lemma 4.1 this will prevent the disjunction subsystem from advancing past the t-ring. If the formula cannot be falsified, then at least one 'term' process will not receive any messages before the disjunction subsystem advances past the t-ring. This is formalized in the following lemma.

Lemma 5.2 *There exists a deadlocking colouring sequence on the communication graph $G(S)$ if and only if the formula F is not a tautology.*

Proof: If F is not a tautology, then there exists an assignment for which every term in the disjunction is false. In this case we show that a deadlocking colouring sequence exists. Let $v_{x_i,0}$ and $v_{\bar{x}_i,0}$ be the start vertices of process components p_{x_i} and $p_{\bar{x}_i}$ respectively. Similarly, let the send vertices $s_{x_i,1}$ and $s_{\bar{x}_i,1}$ be adjacent to the receive vertices r_{q_i,x_i} and $r_{q_i,\bar{x}_i}$ in the process component q_i, and let $s_{x_i,2}$ and $s_{\bar{x}_i,2}$ be the send vertices of the arcs that are incident on the disperser components. Let $Z \subset V$ be the set of start vertices corresponding to the falsifying assignment of F; Z will contain one of $v_{x_i,0}$ or $v_{\bar{x}_i,0}$ for each i.

First, let Σ_Z be the longest colouring sequence constructed by applying the colouring rules only to vertices that have ancestors in Z. Since each q_i has one free token, for every $v_{x_i,0} \in Z$ or $v_{\bar{x}_i,0} \in Z$, vertex r_{q_i,x_i} (respectively $r_{q_i,\bar{x}_i}$), will be assigned a token and coloured yellow via rule 2b; later in the sequence the vertices $s_{x_i,2}$ (respectively $s_{\bar{x}_i,2}$) can be coloured yellow and then green. The corresponding receive vertex in the disperser component can thus be coloured green. Since the dispersers are adjacent to the 'literal' receive vertices in the term process components, the corresponding send vertex of each of the receive vertices can be coloured yellow, implying that rule 2b can be applied to the 'literal' receive vertices. Thus, if Z contains $v_{x_i,0}$ or $v_{\bar{x}_i,0}$, and a term process component contains a 'literal' receive vertex corresponding to $\bar{x}_i$ (respectively x_i), then the token in the corresponding process component will be used up.

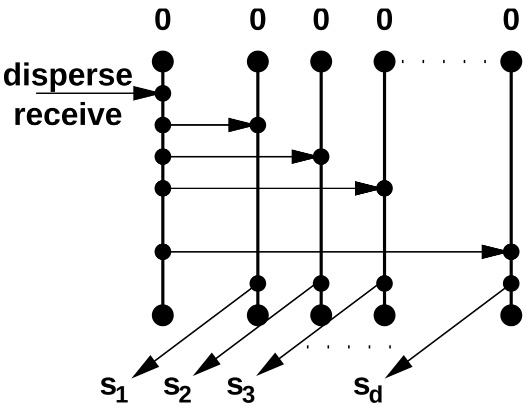

Figure 10: Disperser subsystem.

Second, extend Σ_Z by allowing all valid colourings. To avoid deadlock, Lemma 4.1 requires at least one of the term process components in the disjunction to have a free token. Since F is not a tautology and we used a falsifying assignment, none of the process components has a free token, and the colouring sequence will deadlock.

If F is a tautology, then regardless of the assignment, one of the terms will be true (i.e., all the literals in the term will be true). Hence, at least one term process component will have no 'literal' receive vertices that are descendants of Z, and will have a token available to prevent the colouring sequence from deadlocking on the t-ring. Observe that the selection component allows at most one of r_{q_i,x_i} or $r_{q_i,\overline{x_i}}$ to be coloured yellow, via rule 2b. Provided that the 'done' receive vertex of process component q_i is not coloured green, at least one term component will not have any 'literal' receive vertices to which rule 2b can be applied. Thus, the token belonging to that process component will be available to prevent the colouring sequence from deadlocking on the t-ring. On the other hand, if the 'done' receive vertex is coloured green, then the 'done' send vertex in the term component is coloured green. This means that at least one 't-ring' receive vertex is coloured green, and hence the colouring sequence will not deadlock in the t-ring. Once the 'done' send vertex is coloured green, the colouring sequence can always complete. Hence, the system S is safe. ■

Hence, system S is unsafe if and only if the formula is not a tautology. Since BSP is in **coNP** and is **coNP**-hard via a reduction from TAUTOLOGY, BSP is **coNP**-complete. ■

This result also holds for systems that use a combination of both send and receive buffers. The **coNP**-hardness follows from the fact that systems with receive buffers only are also systems that use combinations of send and receive buffers. Since a colouring sequence also serves as a deadlock certificate for combination systems, the **coNP**-completeness result follows. In the case of systems with send buffers, we conjecture that the corresponding BSP is in **P**. Unlike communication in systems with receive buffers only, the order of the sends implies an order on the allocation of buffers. Hence, we believe that the computation of sufficiency is similar to the nonblocking buffer allocation problem and hence is in **P**.

6 The Nonblocking Buffer Allocation Problem

We now turn to the last of the three problems we consider in this paper. In addition to the system being safe, we can require that no sending process ever blocks due to insufficient buffers on the receiving process. The Nonblocking Buffer Allocation Problem (NBAP) is to determine the minimum number of buffers needed to achieve nonblocking sends. Given $G(S)$, the following algorithm computes the number of buffers needed to assure that none of the sends will ever block.

Given two vertices, $v_{i,c+k}$ and $v_{i,c}$, in $G(S)$, $k > 0$, vertex $v_{i,c+k}$ is **communication dependent** on vertex $v_{i,c}$ if $v_{i,c}$ is the start vertex or if there exist a vertex $v_{j,d}$, $j \neq i$, such that there exists a path from $v_{i,c}$ to $v_{j,d}$ and the arc $(v_{j,d}, v_{i,j+c})$ is in A (see Figure 12). Vertex $v_{i,c+k}$ is **terminally communication dependent** on vertex $v_{i,c}$ if $v_{i,c+k}$ is communication dependent on $v_{i,c}$ and is not communication dependent on the vertices $v_{i,c+l}$, $0 < l < k$. The algorithm is depicted in Figure 13.

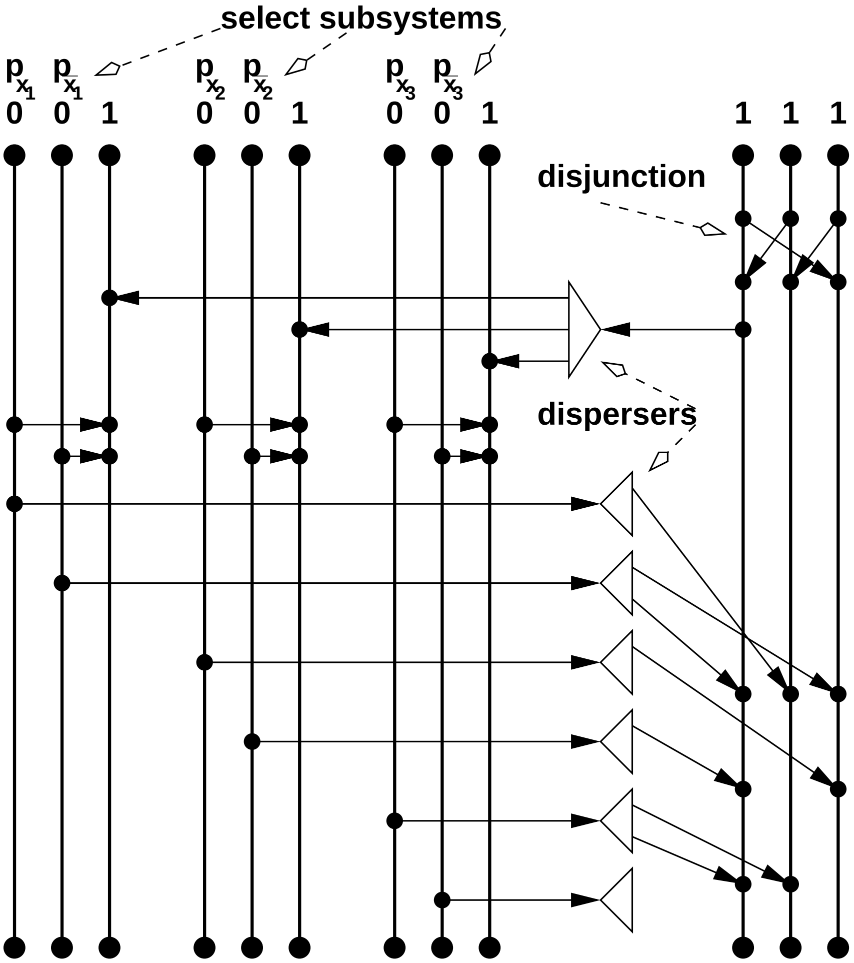

Figure 11: The composite widget for the formula $(x_1 \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_3}) \vee (x_1 \wedge \overline{x_2})$.

The time between when a message can arrive and when it is received at receive vertex $v_{i,c}$ is represented by the interval $I_{i,c}$. Each interval must have a buffer to ensure nonblocking sends. Hence, the minimum number of buffers, b_i, is the maximum overlap over all intervals of a process component $G_i(S)$.

Computing the terminal communication dependencies for $G(S)$ can be done via dynamic programming in $O(|V|n)$ time, where V is the vertex set of $G(S)$ and n is the number of processes. If there exists a path from vertex $v_{i,c}$ to $v_{j,d}$, then there exists a path from $v_{i,c}$ to all vertices $v_{j,d+k}$, $k > 0$. Associate with each vertex $v_{i,c}$ an integer vector $a_{i,c}$ of size n; $a_{i,c}[j] = d$ means that there exists a path from $v_{i,c}$ to $v_{j,d}$, and thus to $v_{j,d+k}$, $k > 0$. The vector $a_{i,c}$ is computed by taking the elementwise minimums over the vectors of the adjacent vertices $v_{i,c}$; this is simply a depth first traversal of $G(S)$. Since the number of arcs is at most $3|V|/2$ and the pairwise comparison takes n steps, the traversal takes $O(|V|n)$ time.

Next, computing the $O(|V|)$ intervals, $I_{i,c}$, requires one table lookup per interval. To compute the maximum overlap we sort the intervals and perform a sweep, keeping track of the current and maximum overlap; this takes $O(|V|\log|V|)$ time. Thus, the total complexity is $O(|V|n + |V|\log|V|)$ time. In the worst case where $p \approx |V|$, this algorithm is quadratic. However, in practice n is usually fixed, in which case the $|V|\log|V|$ term dominates.

6.1 Proof of Correctness of the Nonblocking Buffer Allocation Algorithm

In terms of the colouring game, a system S will not block on any send if for any valid colouring on $G(S)$ containing a yellow send vertex s and a corresponding red receive vertex r, vertex r can be coloured yellow by applying rule 2b. This corresponds to guaranteeing buffer availability for every send in the system.

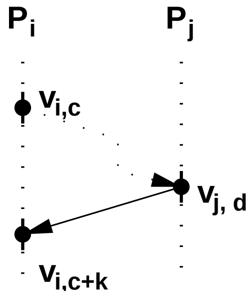

Figure 12: $v_{i,c+k}$ is communication dependent on $v_{i,c}$.

1. For each receive vertex $v_{i,c}$ determine its terminal communication dependency, vertex $v_{i,t}$, where $t < c$.
2. Set $I_{i,c} = [t,c]$ to be the interval between vertex $v_{i,t}$ and vertex $v_{i,c}$.
3. For each process component $G_i(S)$, compute b_i, the maximum overlap over all intervals $I_{i,c}$.
4. $B = \{b_1, b_2, ..., b_n\}$ is the optimal nonblocking buffer assignment.

Figure 13: Algorithm for computing an optimal nonblocking buffer assignment.

Lemma 6.1 *Given a multiprocess system S, let $G(S)$ be the corresponding communication graph. For all vertices $v_{i,s}, v_{j,t} \in G(S)$, if $v_{j,t}$ is a send vertex and there exists a path from the vertex $v_{i,s}$ to vertex $v_{j,t}$, then vertex $v_{j,t}$ cannot be coloured yellow until vertex $v_{i,s}$ is coloured green.*

Proof: By rule 1, the predecessor of $v_{j,t}$ must first be coloured green before $v_{j,t}$ can be coloured yellow. Since rules 3 and 4 imply that the predecessors of a green vertex must be green, the result follows. ■

Corollary 6.2 *Let S, $G(S)$, $v_{i,s}$, and $v_{j,t}$ be as in Lemma 6.1 and let $v_{i,r}$ be the receive vertex corresponding to the send vertex $v_{j,t}$. Rule 2b will never be applied to vertex $v_{i,r}$ before vertex $v_{i,s}$ is coloured green.*

The preceding corollary implies that a buffer for the receive event corresponding to vertex $v_{i,r}$ need not be available until the completion of the send event corresponding to the vertex, on which $v_{i,r}$ is terminally communication dependent. Hence, it is sufficient to allocate the buffer just before the completion of the respective send event. Finally, we argue that this is also necessary.

Theorem 6.3 *Given S and $G(S)$, let $v_{i,s}$ be a send vertex and $v_{i,r}$ be a receive vertex that is terminally communication dependent on vertex $v_{i,s}$. A token for the application of Rule 2b on vertex $v_{i,r}$ must be available before vertex $v_{i,s}$ is coloured green.*

Proof: Let $v_{j,t}$ be the send vertex corresponding to the receive vertex $v_{i,r}$ and let $Q = \{v_{i,q} \mid s < q < r\}$ be the set of vertices that are predecessors of $v_{i,r}$, but on which $v_{i,r}$ is not communication dependent.

Since $v_{i,r}$ is not communication dependent on the vertices in Q, we can construct a colouring sequence on $G(S)$ that fixes the vertices in Q to be red, and colours vertex $v_{j,t}$ yellow, making the application of rule 2b possible in the next step. Since no progress is made in the ith process component after colouring vertex $v_{i,s}$ green, the state of the associated token pool does not change until the application of rule 2b to vertex $v_{i,r}$. Hence, when vertex $v_{i,s}$ is coloured green, the token pool must have a token destined for vertex $v_{i,r}$. ■

Thus, if receive event r is terminally communication dependent on send event s, then it is necessary and sufficient that a buffer to be used for receive event r must be allocated before the send event s completes. The start event may be thought of as a special send event. Since a buffer is required for each receive over a corresponding interval, computing the maximum overlap of intervals yields the number of buffers required.

6.2 Other Models

For systems with only send buffers the problem remains in **P**. The problem can be solved by first reversing all arcs in the communication graph, swapping the start and end vertices, and then running the algorithm described above.

For systems with both send and receive buffers, we conjecture that the nonblocking buffer allocation problem is **NP**-hard. This follows from the observation that we have a choice of either allocating a buffer on the sending or the receiving side, each time a buffer is needed.

6.3 Approximating BAP with NBAP

The NBAP algorithm may be useful for determining a buffer assignment that prevents deadlock (BAP). Since a non-blocking execution is guaranteed not to deadlock, any buffer assignment determined by the NBAP algorithm ensures a safe execution. However, the buffer assignment may be far from optimal. A simple example of this phenomena is a two process producer-consumer system comprising of *n* messages sent by the producer and received by the consumer in the respective order. Such a system requires zero buffers to execute safely, but requires *n* buffers to execute without blocking. Thus, the aforementioned buffer assignment may entail infinitely more buffers than required.

6.4 Implementation of the NBAP algorithm in Millipede

We implemented the NBAP algorithm and added it to the Millipede parallel debugging system, which is a a multi level parallel debugger for message passing programs [19, 20, 21]. Millipede logs all messages between processes in a parallel system; these message histories are then used to analyze program execution and locate bugs. Determining the number of buffers required for block free execution is one such analysis.

To demonstrate the NBAP algorithm we ran Millipede on a program that implements the pipe-and-roll parallel matrix multiplication algorithm [22]. The program has one control process and a number of worker processes arranged in a 2 dimensional mesh. We ran the NBAP algorithm on meshes of size 2×2, 3×3 and 4×4. The communication graph for the smallest example, comprising four workers ordered in a 2×2 mesh is depicted in Figure 14. The corresponding optimal buffer assignment is, listed in the second column of table 1.

Proc.	Max overlap	Overlap for intervals I_j								
		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
0	4	0	0	0	0	4	3	2	1	0
1	3	2	1	2	3	2	1	1	0	0
2	3	3	2	1	2	1	1	1	0	0
3	3	3	2	1	2	1	1	1	0	0
4	3	2	1	2	3	2	1	1	0	0

Table 1: The result of NBAP algorithm on the 2×2 example.

In this example process 0 is the control process and processes 1 through 4 are the workers. The control process needs 4 buffers and the workers each need 3 to execute without blocking. The results obtained when executing the NBAP algorithm on a 3×3 worker system is 9 buffers for the control process and between 4 and 5 buffers for the worker processes. For the 4×4 system the numbers are 16 for the control process and between 5 and 7 buffers for the workers.

7 Conclusion

As more and more functionality of message passing systems is off-loaded to the network interface card, limited buffer space becomes an increasingly important issue. Hence, the problem of determining *k*-safety plays an increasingly important role. Unfortunately this problem is intractable.

We have shown that in the receive buffer model, determining the number of buffers needed to assure safe execution of a program is **NP**-hard, and that even verifying whether a number of assigned buffers is sufficient is **coNP**-complete.

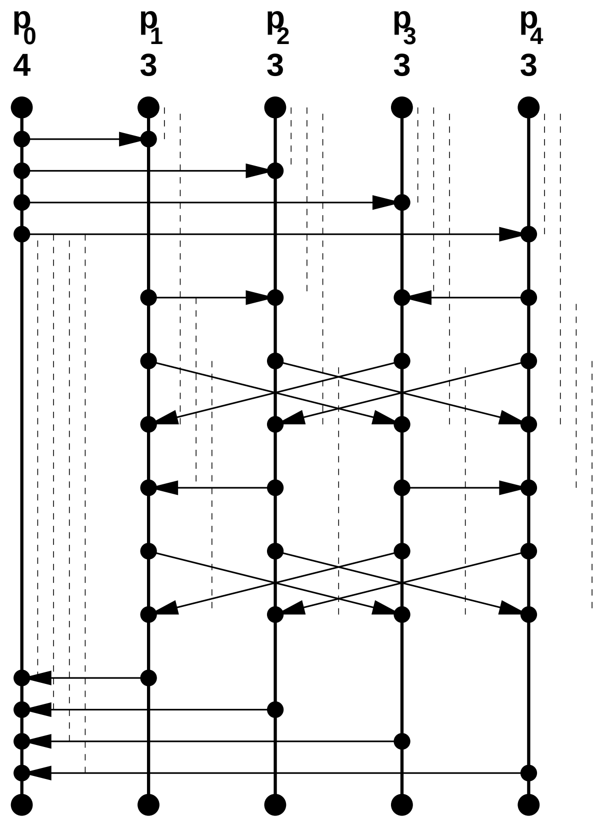

Figure 14: The communication system for a 2 × 2 worker process mesh.

On the positive side, if we require that no send blocks, we provide a polynomial time algorithm for computing the minimal number of buffers. By allocating this number of buffers, safe execution is guaranteed!

There are several strategies that a programmer can use to reduce the likelihood of deadlock when only a few buffers are available. To decrease the risk of deadlock the programmer can introduce epochs that are separated by barrier synchronizations. If each epoch only needs a small number of buffers, the risk of deadlock due to buffer insufficiency is reduced.

For systems with only send buffers we conjecture that the Buffer Sufficiency Problem can be solved in polynomial time because the order of the sends in each process is fixed. This would imply that the buffer allocation problem for systems with only send buffers is **NP**-complete. For systems with both send and receive buffers we conjecture that the nonblocking buffer allocation problem is **NP**-hard. Unlike in the other two models, a buffer can be assigned either on the send side or on the receive side, dramatically increasing the size of the search space. The results (conjectures) are summarized below.

Problem	Receive Buffers	Send Buffers	Send/Receive Buffers
BAP	**NP**-hard	**NP**-hard	**NP**-hard
BSP	**coNP**-complete	(**P**)	**coNP**-complete
NBAP	**P**	**P**	(**NP**-hard)

References

[1] C. Keppitiyagama and A. Wagner. Asynchronous MPI messaging on myrinet. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2001.

[2] D. Frye, R. Bryant, H. Ho, R. Lawrence, and M. Snir. An external user interface for scalable parallel systems. Technical report, IBM highly parallel supercomputing systems laboratory, November 1992.

[3] J. Dongarra, R. Hempel, A. Hey, and D. Walker. A proposal for a user-level, message-passing interface in a distributed memory environment. Technical Report TM-12231, ORNL, June 1993.

[4] J. Dongarra. MPI: A message passing interface standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

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

[6] G. Burns and R. Daoud. Robust MPI Message Delivery with Guaranteed Resources. MPI Developers Conference at the University of Notre Dame, June 1995.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, 1994.

[8] R. Cypher and E. Leu. Repeatable and portable message-passing programs. In *Proc. of The Symposium on the Principles of Distributed Computing (PODC)*, pages 22–31, 1994.

[9] R. Cypher and E. Leu. The semantics of blocking and nonblocking send and receive primitives. In *Proceedings of 8th IEEE International parallel processing symposium (IPPS)*, pages 729–735, 1994.

[10] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Journal of Distributed Computing*, 9(4):173–191, 1996.

[11] L. Lamport. Time, clocks and the orderings of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[12] K. Jensen. *Coloured Petri nets. Basic Concepts, Analysis Methods and Practical use*, volume 1. Springer, 1992.

[13] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for hihg-level Petri nets. *Theoretical Computer Science*, 45:261–292, 1985.

[14] V. Anantharm. The optimal buffer allocation problem. *IEEE Transactions on Information Theory*, 35(4):721–725, 1989.

[15] M. Reiman. The optimal buffer allocation problem in light traffic. In *IEEE Conference on Decision and Control*, 1987.

[16] T. Sheskin. Allocation of interstage storage along an automatic production line. *AIEE Transactions*, 8(1), 1975.

[17] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[19] J. Bækgaard Pedersen and A. Wagner. Sequential Debugging of Parallel Programs. In *Proceedings of the international conference on communications in computing, CIC'2000*. CSREA Press, June 2000.

[20] J. Bækgaard Pedersen and A. Wagner. Correcting errors in message passing systems. In Frank Mueller, editor, *High-Level Parallel Programming Models and Sypportive Environments, 6th international workshop, HIPS 2001 San Francisco, CA, USA*, volume 2026 of *Lecture Notes in Computer Science*, pages 122–137. Springer Verlag, April 2001.

[21] J. Pedersen and A. Wagner. Protocol Verification in Millipede. In *Communicating Process Architectures 2001*. IOS Press, September 2001.

[22] G. Fox, M. Johnson, G. Lyzenga, S. Otto J. Salmon, and D. Walker. *Solving problems on concurrent processors. General techniques and regular problems*, volume 1. Prentice-Hall, Inc., 1988.