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,
201-2366 Main Mall,
Vancouver, British Columbia, V6T 1Z4,
Canada
Phone: 604 822 2895
Fax: 604 822 5485

Abstract

Message passing programs commonly use buffers to avoid unnecessary synchronizations and to improve performance by overlapping communication with computation. Unfortunately, using buffers makes the program no longer portable, potentially unable to complete on systems without a sufficient number of buffers. Effective buffer use entails that the minimum number needed for a safe execution be allocated.

We explore a variety of problems related to buffer allocation for safe and efficient execution of message passing programs. We show that determining the minimum number of buffers or verifying a buffer assignment are intractable problems. However, we give a polynomial time algorithm to determine the minimum number of buffers needed to allow for asynchronous execution. We extend these results to several different buffering schemes, which in some cases make the problems tractable.

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

1 Introduction

In the last decade MPI [8] and PVM [14] have become the de facto standards for message passing programs. They have replaced the myriad of libraries that pro-

Email addresses: abrodsky@cs.ubc.ca (Alex Brodsky), matt@cs.ubc.ca (Jan Bækgaard Pedersen), wagner@cs.ubc.ca (Alan Wagner).

vided a degree of portability for message passing programs. One aspect of portability introduced in the MPI standard was that of a *safe* program. As defined in the standard, a program is safe if it requires no buffering, that is, if it is synchronous. Safe programs can be ported to machines with differing amounts of buffer space. However, to demand that the program execute correctly with no buffering is restrictive. Buffering reduces the amount of synchronization delay and also makes it possible to off-load communication to the underlying system or network components, thus overlapping communication and computation. Although one cannot assume an infinite number of buffers, by characterizing the buffer requirements of a given program it becomes possible to determine, with respect to buffer availability, whether the program can be ported to a given machine. The notion of k-safety is introduced to address the problem of identifying the buffer requirements of a program to avoid buffer overflows and deadlock. Determining the minimum k, under a variety of buffer placements, is important for constructing programs that are both safe and can effectively exploit the underlying hardware.

Unfortunately, the value of k is usually not known a priori. We investigate the complexity of determining a minimum value of k for programs using asynchronous buffered communication with a static communication pattern and a bounded message size. We consider the following three problems: the Buffer Allocation Problem (BAP), which is the problem of determining the minimum number of buffers required to ensure deadlock free execution (i.e., determine k for k-safety); the Buffer Sufficiency Problem (BSP), which is to determine whether a given buffer assignment is sufficient to avoid deadlock; and finally, the Nonblocking Buffer Allocation Problem (NBAP),which is to determine the minimum number of buffers needed to allow for asynchronous execution, that is, when send calls do not block.

The complexity of these questions also depends on the type of buffers provided by the system. We consider four types of system buffering schemes. In the first three schemes the buffers are (1) pre-allocated on the send side only, (2) the receive side only, or (3) mixed and pre-allocated on both sides. Finally, we also consider a scheme that pre-allocates buffers on a per channel basis, where each communication channel can buffer a fixed number of messages.

We show that the Buffer Allocation Problem is intractable under all four buffer allocation schemes. The Buffer Sufficiency Problem is intractable for the receive side buffer and for the mixed buffer allocation schemes, tractable for the channel scheme and conjectured tractable for sender side buffers. Finally, the Nonblocking Buffer Allocation Problem is tractable for all buffer placement schemes, except the mixed send and receive scheme.

2 Related Work

The multiprocess system that we consider is a collection of simultaneously executing independent asynchronous processes that compute by interspersing local computation and point-to-point message passing between processes; these are referred to as *A-computations* in [4]. Such a system is equivalent to one with three different events, such as the one defined by Lamport [18]: send events, receive events and internal events. As well, we only consider programs that are repeatable [6,7] when executed in an unrestricted environment, that is, programs with static communication patterns. While this narrows the class of programs we consider, the class of applications with static communication patterns is still considerable.

The message passing primitives considered in this paper are the traditional asynchronous, buffered communications: the *nonblocking send* and the *blocking receive*, which are the standard primitives used in MPI and PVM. 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* [6,7]. Informally, the send blocks until the message is copied out of the process into a send buffer; the receive blocks until the message has been copied into the receive buffer.

The notion of safety, as introduced in the MPI standard, underscore the concern that, when buffer resources are unknown, asynchronous communication can potentially deadlock the system. This notion was extended to k-safety, in order to better characterize the buffer requirements of the program, thus making it safe to take advantage of asynchronous communication. The definition of k-buffer correctness was introduced by Bruck *et al.* [2] to describe programs that complete without deadlock in a message passing environment with k buffers per process. Similarly, Burns and Daoud [3] introduced *guaranteed envelope resources* into LAM [12], a public domain version of MPI. Guaranteed envelope resources—a weaker condition than k-safety—was used in LAM to reserve a guaranteed number of message header slots on the receiver side.

Determining whether a system is buffer independent—the system is 0-safe—was investigated in [6,7]. In our model, the interesting systems are buffer-dependent, and require an unknown number of buffers to avoid deadlock.

More recently in modern clusters, greater overlap of computation and communication is possible by off-loading communication onto the network interface cards. Unfortunately, most NICs have orders of magnitude less memory than the average host, which makes message buffers a limited resource. Thus, programs that use asynchronous message passing, and that execute correctly otherwise, might deadlock when executing on a system where parts of the message passing system have been off-loaded to the NIC. These issues have been investigated in several papers [8,9,11,17].

To determine the minimum number of buffers, the execution of a system can be modeled using a (coloured) Petri net [16]. In order to determine whether the system can reach a state of deadlock, the Petri net occurrence graph [15] 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.

Variations of these problems have been investigated by the operations research community [1,20,21]. 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; that is, 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 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**. Figure 1 is an example of an unsafe system. The numbers above the graph in Figure 1 represent the buffer assignment.

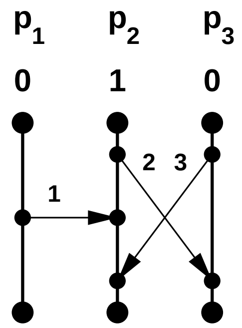

Fig. 1. Order of execution can cause deadlock.

A per-process **buffer assignment** is an n-tuple $B = (b_1, b_2, \ldots, b_n)$ of non-negative integers representing the number of buffers that can be allocated by each process. Similarly, a per-channel buffer assignment is a q-tuple $B = (b_1, b_2, \ldots, b_q)$, $q = \binom{n}{2}$, representing the number of buffers each channel in the system can allocate. Since buffers take up memory, which may be needed by the application, ideally, as few buffers as possible should be allocated. However, allocating too few buffers results in an unsafe system.

Buffer utilization is the nondeterministic phenomena of interest in the system. Making the choice of when to use a buffer affects future choices. For example, in Figure 1, using a buffer for communication 1 before communication 3 completes results in deadlock.

Two natural decision problems arise from this optimization problem. Given a system S and a non-negative integer k, the **Buffer Allocation Problem** (BAP) is to decide if there exists a buffer assignment B such that S is safe and $\sum b_i \leq k$. In order to solve this problem we need to solve a simpler one. Suppose we are given a buffer assignment B and a system S; the **Buffer Sufficiency Problem** (BSP) is then to decide whether 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 system S and a non-negative integer k, the **Nonblocking Buffer Allocation Problem** (NBAP) is to decide whether there exists a buffer assignment B, such that no send in S ever blocks, and $\sum b_i \leq k$.

We model systems by using communication graphs, and executions of systems by colouring games on these graphs. Communication graphs can be derived from execution traces of a program. The following subsection defines the graph based framework used throughout this paper.

3.1 The Graph Based Framework

A **communication graph** of S is a directed acyclic graph $G = G(S) = (V,A)$ where the set of vertices $V = \{v_{i,c} \mid 1 \leq i \leq n, 0 \leq c \leq (E_i+1)\}$ 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. Communication graphs are comparable to the time-space diagrams—without internal events—noted in [18].

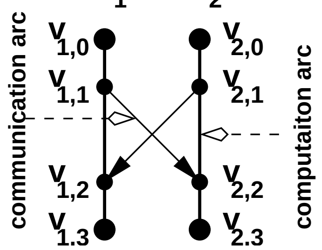

Fig. 2. An example of a communication graph with a 2-ring.

The ith **process component** G_i of G is the subgraph $G_i = (V_i, A_i)$ where $V_i = \{v_{i,c} \in V \mid 0 \leq c \leq (E_i + 1)\}$ 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 channel is represented by a **channel pair** (G_i, G_j) of process components.

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 [4].

A t-ring represents a circular dependence of alternating send and receive events; see the example in Figure 3. The shaded arcs in Figure 3 show how 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.

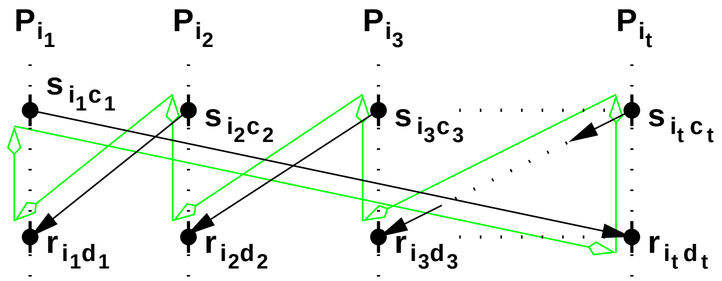

Fig. 3. Dependency cycle in $G(S)$.

To model the execution of a system S, 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 yet 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 light changing from red, to yellow, to green. [1]

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

We use tokens to represent buffer allocations. A buffer assignment of a process (or channel) is represented by a pool of tokens associated with the corresponding process component (respectively, the channel component). A instance of buffer utilization is represented by removing a token from a token pool and placing it on the corresponding communication arc.

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

$\mathit{send} \to \mathit{yel}$	A red send vertex may be coloured yellow if the preceding vertex is green—the send is ready.
$\mathit{recv} \to \mathit{yel}$	A red receive vertex may be coloured yellow if the corresponding send vertex is yellow, and the preceding vertex (in the same process component) is green—both the send and the receive are ready.
$\mathit{recv} \xrightarrow{\bullet} \mathit{yel}$	A red receive vertex may be coloured yellow if the corresponding send vertex is yellow, and a token from the corresponding token pool is placed on the incident communication arc—the send is ready and a buffer is used.
$\mathit{send} \to \mathit{grn}$	A yellow send vertex may be coloured green if the corresponding receive vertex is coloured yellow—the communication has completed from the sender's perspective.
$\mathit{recv} \to \mathit{grn}$	A yellow receive vertex may be coloured green if both of its preceding vertices are green. If the incident communication arc has a token, the token is returned to its token pool—a receive completes after the send completes.
$\mathit{end} \to \mathit{yel}$	A red end vertex may be coloured yellow if the preceding vertex is green.
$\mathit{end} \to \mathit{grn}$	A yellow end vertex may be coloured green.

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

A **colouring** of G, denoted by χ, is a colour assignment to all vertices, which 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 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 sequence on $G(S)$. 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. Using the correspondence between colouring sequences on $G(S)$ and executions of system S, we reason about system S by reasoning about colouring sequences 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. A system S is safe if and only if every colouring sequence on the graph $G(S)$ completes.

We say that a colouring sequence **blocks** if there exists a sequence on $G(S)$, ending with a colouring containing a yellow send vertex, that cannot be extended by applying rule *recv*$\overset{\bullet}{\rightarrow}$*yel* to the corresponding receive vertex. A colouring sequence is **block free** if every prefix of the sequence does not block; a communication graph G, is block free if all colouring sequences on it are also block free. If $G(S))$ is block free, then no send in S will ever block during an execution.

A **token assignment**, also denoted by B, is a list of nonnegative integers, denoting the number of tokens assigned to each token pool; the token assignment is the abstract representation of a buffer assignment. The number of tokens required depends on the number of times that rule *recv*$\overset{\bullet}{\rightarrow}$*yel* can be invoked. If a token pool is empty, this means all buffers are in use.

4 Useful Lemmas

The following lemmas are used throughout our proofs. Lemma 4.1 characterizes the conditions under which a colouring sequence will deadlock. Lemma 4.2 characterizes conditions under which a single colouring sequence may represent all possible colouring sequences. Finally, Lemma 4.3 characterizes a class of communication graphs on which no colouring sequence will deadlock.

Lemma 4.1 (The t-Ring Lemma) *Let G be a communication graph comprising a single t-ring. Any colouring sequence on G completes if and only if rule recv*$\overset{\bullet}{\rightarrow}$*yel is invoked at least once.*

Proof: Assume by contradiction that there exists a complete colouring sequence Σ that does not make use of rule *recv*$\overset{\bullet}{\rightarrow}$*yel*. Consider the first colouring in Σ where one of the send vertices is green; call the vertex s_i. Let r_j be the corresponding receive vertex. According to rule *send*$\rightarrow$*grn*, the vertex r_j must be yellow. Since rule *recv*$\overset{\bullet}{\rightarrow}$*yel* has not been applied, rule *recv*$\rightarrow$*yel* 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 *recv*$\rightarrow$*yel* 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.

In the other direction, if rule *recv*$\overset{\bullet}{\rightarrow}$*yel* is invoked on receive vertex r_j, then rule *send*$\rightarrow$*grn* can be invoked on the corresponding send vertex s_j, breaking the circular dependency. ■

Define the dependency graph of communication graph $G = (V,A)$ to be $H = (V,E)$ where all process arcs in A are reversed in E and all communication arcs in A are bidirectional in E. Define the depth $d(v)$ of a vertex $v \in V$ to be the maximum length path in H from v to a start vertex.

Lemma 4.2 *Let G be communication graph with a token assignment of* 0*. For any vertex v in G, if there exists a colouring sequence that colours vertex v green, there does not exist a colouring sequence that deadlocks before colouring v green.*

Proof: Proof by contradiction. Assume that there exist two colouring sequences, such that one colouring sequence colours a vertex green and the other deadlocks and does not colour the vertex green. Let $v \in V$ be such a vertex of minimum depth; that is, all vertices of lesser depth will be coloured green eventually by any colouring sequence on G. In order for a vertex to be coloured green, its component predecessor must be green. Since the depth of the predecessor is less than the depth of v, it can always be coloured green. Furthermore, since a send and its corresponding receive vertex are adjacent to each other, their depths differ by at most 1.

Since v must be a communication vertex, by rules *send*$\rightarrow$*grn* and *recv*$\rightarrow$*grn*, the adjacent communication vertex t must be coloured yellow before v can be coloured green. If vertex t is of a lesser depth than v, then t must be green colourable in all colouring sequences; hence, v must also be green colourable. If t is at the same depth as v, then its component predecessor is at a lesser depth and must be green colourable, hence t is yellow colourable, and v is green colourable. If t is at a greater depth than v, the component predecessor of t, say u, is at the same or a lesser depth than t. If the latter, then u is green colourable and t is yellow colourable, otherwise, we apply the same argument to u first. Since there is no path from u to v in H—because $d(u) \leq d(v)$—we need only recurse a finite number of times. ■

Lemma 4.3 *If G is a communication graph whose dependency graph is acyclic, then no colouring sequence on G will deadlock.*

Proof: Proof by contradiction. Assume that a colouring sequence deadlocks on G. Let v be the vertex of minimum depth that cannot be coloured green. If v is a send (receive) vertex, let u be the corresponding receive (send) vertex. Let vertex t be the component predecessor of vertex u and let vertex w be the component predecessor of vertex v. Since the dependency graph is acyclic, the depths of both t and w are less than the depths of u and v. Hence, both t and w may be coloured green based on our minimality assumption. However, then both u and v may be coloured green;

this is a contradiction! If v is an end vertex, then it has only one predecessor, which is of a lesser depth, which leads to the same contradiction. ■

5 Buffer Allocation in Systems with Receive Side Buffers

In systems with receive side buffers, messages are buffered only by the receiver. Buffers are allocated by the receiving process when a message arrives, but cannot be received, and are freed when the message is received by the application. Analogously, when colouring a receive vertex of the corresponding communication graph, only a token belonging to the same process component may be used. We call this the **receive side allocation scheme**.

5.1 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, determining the required number of buffers, such that the system is safe, is intractable.

The corresponding graph-based decision problem is this: given a communication graph G and a positive integer k, determine if there is a token assignment of size k such that no colouring sequence deadlocks on G. We show that BAP_r is **NP**-hard by a reduction of the well known 3SAT problem [5] to BAP_r. Recall the definition of 3SAT: 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 $\{x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n\}$.

Theorem 5.1 *The Buffer Allocation Problem (*BAP_r*) is* **NP***-hard.*

Proof: Proof by reduction of 3SAT to BAP_r. For any 3SAT instance F we construct a corresponding communication graph G such that for a token assignment of size n, any colouring sequence completes on G if and only if the corresponding variable assignment satisfies F.

Let F be an instance of 3SAT with n variables and c clauses; the variables are denoted $x_1, x_2, \ldots, x_n$, and the jth clause is denoted $(a_j \vee b_j \vee c_j)$, where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. The corresponding communication graph G comprises $2n+1$ process components: $2n$ of the components—called **literal** components—are labeled P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \ldots n$, and correspond to the literals of F. The last component—called the **barrier** component—is labeled P_{barrier}.

Each process component is divided into $c+1$ epochs, where each epoch is a consecutive sequence of zero or more vertices within the component. All epochs are synchronized, that is, the vertices of one epoch must be coloured green before any of the vertices in the next epoch may be coloured. To ensure this we use a barrier component; the jth epoch of the barrier component, $j = 0, \ldots, c$, comprises $2n$ receive vertices, labeled $q_{l,j}$, and $2n$ send vertices, labeled $t_{l,j}$, $l \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. At the end of each epoch there is an arc from each of the literal components P_l, $l \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$, to the barrier component. Each arc emanates from vertex $s_{l,j}$, called a barrier send vertex, and is incident on vertex $q_{l,j}$, where $l \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$ and $j = 0 \ldots c$. These arcs are followed by arcs emanating from the barrier component to the literal components; the arcs emanate from vertices $t_{l,j}$ and are incident on vertices $r_{l,j}$, called barrier receive vertices. The barrier widget has no cyclic dependencies. Hence, by Lemma 4.3, no colouring sequence will deadlock on a barrier widget.

Epoch 0 fixes a token assignment corresponding to a variable assignment in 3SAT. Each pair of process components, P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \ldots n$, forms a variable widget, which corresponds to a variable. The two process components of a pair share a 2-ring; see Figure 4. By Lemma 4.1, at least one token must be assigned to either process component P_{x_i} or $P_{\bar{x}_i}$ to prevent all colouring sequences from deadlocking on G. Since only n tokens are available, each component pair can be assigned exactly one token. Finally, assigning the token to process component, P_{x_i} or $P_{\bar{x}_i}$, corresponds to fixing variable x_i to true or false. The epoch terminates with a barrier send vertex $s_{l_i,0}$, followed by a barrier receive vertex $r_{l_i,0}$, $l_i \in \{x_i, \bar{x}_i\}$.

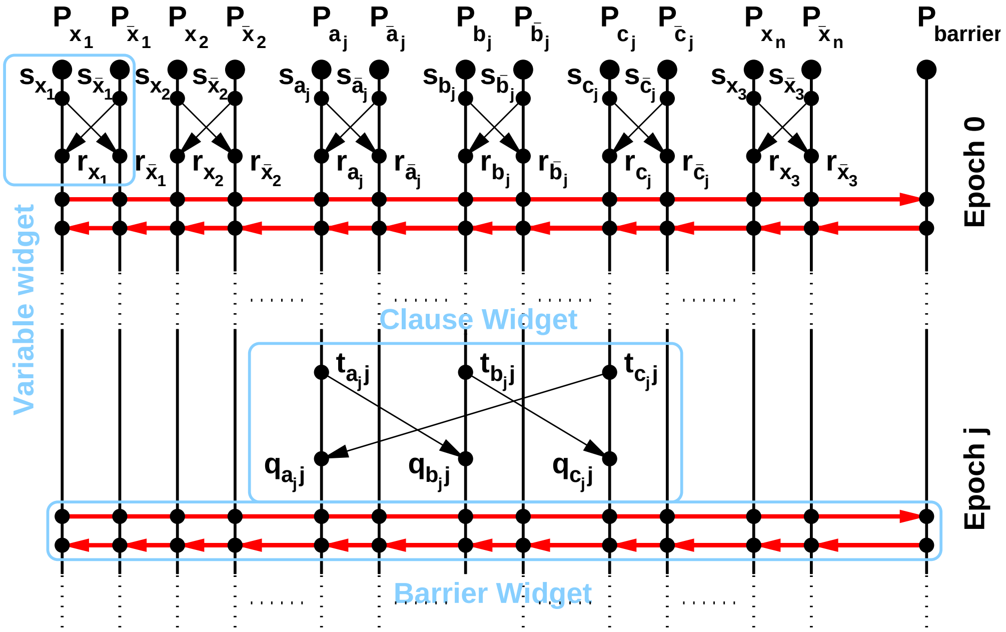

Fig. 4. Construction of G.

Epoch j of each process component corresponds to the jth clause of F. The epoch of a process component P_l, $l \neq a_j, b_j, c_j$—not labeled by a literal of the jth clause—contains only two vertices: the barrier send vertex $s_{l,j}$ and the barrier receive vertex $r_{l,j}$. The three process components, P_{a_j}, P_{b_j}, P_{c_j}, whose labels correspond to the literals in the jth clause share a 3-ring in the jth epoch; see Figure 4. By Lemma 4.1, to avoid deadlock, at least one of the three process components must have a token. If none of the components are assigned a token, all literals in the jth clause are false. The epoch is terminated by the barrier send and the barrier receive vertices.

A satisfying assignment on F satisfies at least one literal in every clause. A corresponding token assignment assigns a token to the corresponding process component in each 3-ring—corresponding to the true literal. Hence, by Lemma 4.1 none of the colouring sequences will deadlock on any of the clause widgets and any colouring sequence on G will complete.

For a falsifying assignment of F, there exists at least one clause comprising false literals. The corresponding token assignment fails to assign any tokens to the process components that share the corresponding 3-ring. Thus, by Lemma 4.1 all colouring sequences will deadlock in that clause widget.

Hence, for a token assignment of size n, any colouring sequence on G will complete if and only if the corresponding assignment satisfies F. Since finding a token assignment of size n such that no colouring sequence on G deadlocks is as hard as finding a satisfying assignment for F, BAP_r is **NP**-hard. ■

5.2 The Buffer Sufficiency Problem

A potentially simpler problem involved verifying whether a given buffer assignment is sufficient to prevent deadlock. Formally, given a graph G and a token assignment on G, determine if none of the colouring sequences on G deadlock. This problem turns out to be intractable as well.

We show that BSP_r is **coNP**-complete by a reduction from the TAUTOLOGY problem [13, Page 261] to BSP_r. Given an instance of a formula in disjunctive normal form (DNF), $\bigvee_{i=1}^{t} \bigwedge_{j=1}^{l_i} a_{i,j}$ where $a_{i,j} \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$, the formula is a tautology if it is satisfied by all assignments. An assignment that falsifies F is a concise proof that the formula is not a tautology. We shall restrict our attention to 3DNF formulas, where each term has three literals: $\bigvee_{i=1}^{t}(a_i \wedge b_i \wedge c_i)$.

Theorem 5.2 *The Buffer Sufficiency Problem (*BSP_r*) is* **coNP***-complete.*

Proof: Let G be a communication graph along with a token assignment. If there exists a deadlocking colouring sequence on G, then the sequence itself is a certificate. The sequence is at most twice the number of vertices in G. Hence, BSP_r is in **coNP**.

Let F be a 3DNF formula with t terms where each term has three literals. For any 3DNF formula F, we construct a communication graph G and fix a token assignment such that there is a colouring sequence on G that deadlocks if and only if the corresponding assignment falsifies F. The construction consists of four types of widgets that correspond to fixing an assignment, a term in the disjunction, the disjunction, and the interconnects between widgets.

Each variable in F is represented by a variable widget comprising three process components that are labeled P_{x_i}, $P_{\bar{x}_i}$, and Q_i. The latter, called the **arbitrator** component, comprises three receive vertices, labeled q_i, r_{x_i}, and $r_{\bar{x}_i}$. The former two process components, called **variable** components, comprise two send vertices each. The first, labeled s_{x_i} ($s_{\bar{x}_i}$), is adjacent to the corresponding receive vertex r_{x_i} ($r_{\bar{x}_i}$) in the arbitrator component. The second, labeled t_{x_i} ($t_{\bar{x}_i}$), is adjacent to receive vertices in widgets called dispersers, described later. The vertex q_i in the arbitrator component is similarly adjacent to a vertex in a disperser widget. The corresponding token assignment for each variable widget assigns one token to Q_i and no tokens to the other two components; see Figure 5. The widget has the following property:

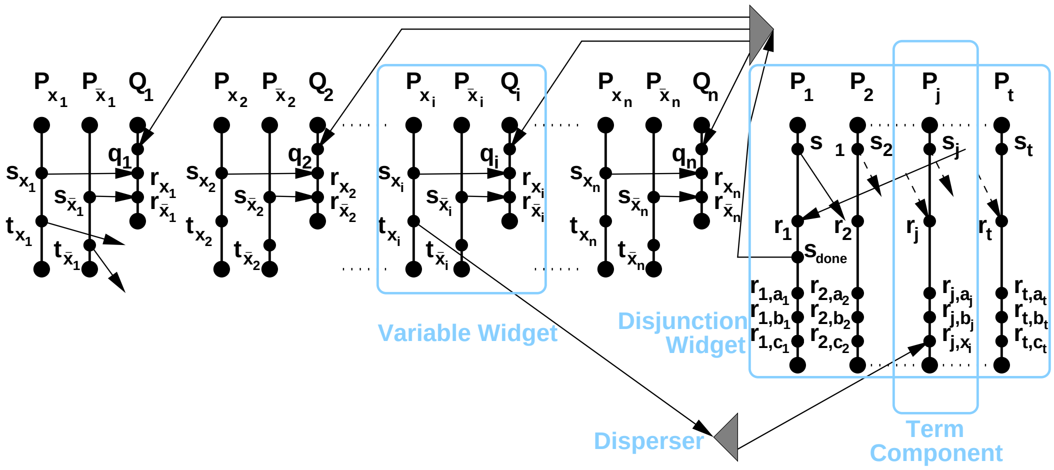

Fig. 5. The construction.

Property 5.3 *Let G be a communication graph that contains a variable widget. Any colouring sequence on G may colour exactly one of the two vertices t_{x_i} or $t_{\bar{x}_i}$ yellow before vertex q_i is coloured green.*

Proof: By rule *send→yel*, in order for t_{x_i} ($t_{\bar{x}_i}$) to be coloured yellow, vertex s_{x_i} ($s_{\bar{x}_i}$) must be coloured green. Hence, by rule *send→grn*, vertex r_{x_i} ($r_{\bar{x}_i}$) must first be coloured yellow. Since vertex q_i is red, vertex r_{x_i} ($r_{\bar{x}_i}$) can only be coloured yellow via rule $recv\overset{\bullet}{\rightarrow}yel$. However, there is only one token assigned to process component Q_i, hence rule $recv\overset{\bullet}{\rightarrow}yel$ may only be invoked once. ■

The jth term in the disjunction is represented by a term widget comprising a process component, which is called the **term** component and labeled P_j. The first part of each term component consists of a send vertex s_j and a receive vertex r_j; these vertices are part of a t-ring. In the first term component, P_1, there is an additional send vertex labeled s_{done}; these are described in the next paragraph. The second part of each term component consists of three receive vertices labeled r_{j,a_j}, r_{j,b_j}, and r_{j,c_j}, where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$ correspond to the literals in the jth term;

see Figure 5. These receive vertices are adjacent to send vertices in widgets called dispersers, which are described later. The term components are used to construct a disjunction widget.

The disjunction widget comprises t term components, where the first two vertices, s_j and r_j, are part of a t-ring spanning all t components. Specifically, each send vertex s_j, $j < t$, is adjacent to receive vertex r_{j+1} and vertex s_t is adjacent to receive vertex r_1; see Figure 5. Each term component is assigned one token. The disjunction widget has the following property.

Property 5.4 *Let G be a communication graph that contains a disjunction widget. Any colouring sequence on G can colour r_j, $j \in [1,t]$, green if and only if at least one of r_k, $k \in [1,t]$, is coloured yellow before any r_{k,a_k}, r_{k,b_k}, or r_{k,c_k} are coloured yellow.*

Proof: By Lemma 4.1, vertex r_j can be coloured green, if and only if rule $recv \stackrel{\bullet}{\rightarrow} yel$ is invoked, colouring one of the receive vertices r_k, $k \in [1,t]$, yellow. The rule may only be invoked if and only if a token is available. Since each term component only has one token assigned and since vertex r_k precedes vertices r_{k,a_k}, r_{k,b_k}, and r_{k,c_k}, a token is available if and only if none of the vertices r_{k,a_k}, r_{k,b_k}, and r_{k,c_k}, are coloured yellow via rule $recv \stackrel{\bullet}{\rightarrow} yel$, before vertex r_k is coloured yellow. ■

Once vertex r_k, $k \in [1,t]$, is coloured yellow, all r_j, $j = 1 \ldots t$ may be coloured green, and vertex s_{done} may be coloured yellow. We now describe how the widgets are connected together using disperse widgets. Let s be a send vertex and R be a set of receive vertices. An (s,R)-disperser comprises $|R|+1$ process components: one **master** component, labeled M_s, and $|R|$ **slave** components labeled S_r, $r \in R$. The master component comprises one receive vertex labeled r_s, followed by $|R|$ send vertices labeled s_r, $r \in R$. Each send vertex is adjacent to the receive vertex on the corresponding slave component S_r. Each slave component has two vertices: a receive vertex q_r, followed by a send vertex t_r; see Figure 6. The latter vertex is adjacent to the receive vertex r in some other widget. None of the components are assigned any tokens. The following property of a disperser follows from Lemma 4.3.

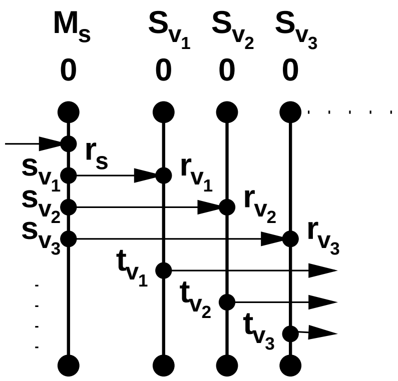

Fig. 6. The disperser widget.

Property 5.5 *Let G be a communication graph containing an (s,R)-disperser. If a colouring sequence colours vertex r_s yellow, then the colouring sequence can be extended to colour all vertices t_r, $r \in R$ yellow.*

Let R_{x_i}, $i = 1 \ldots n$, be the set of receive vertices labeled $r_{j,x_i} \in P_j$, $j \in [1,t]$, and let $R_{\bar{x}_i}$ be similarly defined; recall that a_j, b_j, c_j are simply literal place holders in the vertex labels r_{j,a_j}, r_{j,b_j}, r_{j,c_j}. Hence, a (t_{x_i}, R_{x_i})-disperser connects send vertex $t_{x_i} \in P_{x_i}$ to vertices in R_{x_i}—belonging to the term components. Furthermore, let Q be the set of receive vertices q_i (in the variable widgets), $i = 1 \ldots n$; a (s_{done}, Q)-disperser connects vertex s_{done} to all variable widgets via receive vertices q_i. The construction of G comprises n variable widgets and one disjunction widget, composed of t term widgets; these are connected together by a (s_{done}, Q)-disperser, and $2n$ (t_a, R_a)-dispersers, where $a \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. We claim that there exists a colouring sequence that deadlocks on G if and only if there is a falsifying assignment for formula F, that is, F is not a tautology.

Suppose that F has a falsifying assignment x, that is every term in the disjunction is false because each term has a literal x_i or $\bar{x}_i$, which is false. To construct a colouring sequence on G that deadlocks, we construct a set of vertices U. The first half of the colouring sequence is a maximal colouring sequence involving only the vertices of U. The second half of the sequence may involve all vertices in G. The resulting colouring sequence will always deadlock.

Let $X = \{a \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\} \mid a|_x = 0\}$, which is the set of literals that are false, and let $Z = \{s_a \in P_a \mid a \notin X\} \cup \{s_j \mid j = 1, \ldots, t\}$, which contains the set of send vertices from the variable components that are labeled by a true literal and the numbered send vertices in the disjunction widget; the set Z contains the vertices which may not initially be coloured. Let $U = V \backslash Z$ be the rest of the vertex set.

Consider a colouring sequence involving only vertices in U. By property 5.3 any maximal colouring sequence will colour the vertices t_a yellow (in the variable widget), where $a \in X$. Hence, by property 5.5 the vertices t_r (in the dispersers) will be coloured yellow, where $r \in \bigcup_{a \in X} R_a$—the send vertices t_r in the dispersers are adjacent to the receive vertices in R_a. Since x is a falsifying assignment, every term contains a literal, which is falsified by x. Without loss of generality, let a_j denote a literal that is false in the jth term; therefore, process component P_j contains a receive vertex r_{j,a_j}, which is adjacent to the yellow send vertex $t_{r_{j,a_j}}$ (in the disperser). Since none of the vertices of the t-ring (in the disjunction widget) are not in U—they are still coloured red—the token belonging to component P_j is used to apply rule *recv*$\overset{\bullet}{\to}$*yel* to colour vertex r_{j,a_j} yellow. Since every term has a false literal, the colouring sequence colours a receive vertex r_{j,a_j}, $j = 1 \ldots t$ in every term component P_j. After the sequence cannot be extended, allow all vertices to be coloured; since vertices r_{j,a_j} (in the term components), $j = 1 \ldots t$, have been coloured yellow before vertex r_j (in term component P_j), according to property 5.4, the sequence will deadlock.

If a colouring sequence on G deadlocks, according to property 5.4, deadlock occurs only if there is a yellow vertex labeled r_{k,a_k}, r_{k,b_k}, or r_{k,c_k} in each of the term components. Their predecessors—vertices t_l, $l \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_m\}$, in the dispersers—must be green. Since the colouring sequence is maximal, by property 5.3 exactly one of t_{x_i} or $t_{\bar{x}_i}$ is red, thus this corresponds to a valid assignment: setting $x_i = 0$ if t_{x_i} is green, or $x_i = 1$ if $t_{\bar{x}_i}$ is green yields an assignment that falsifies F.

Thus, a colouring sequence on G deadlocks if and only if the corresponding assignment falsifies F. Hence, BSP$_r$ is **coNP**-complete. ■

Therefore, just determining whether a buffer assignment is sufficient is intractable, even one as simple as in the preceding example. Intuitively, the buffers of a process are assigned based on the behaviour of other processes; thus, buffer utilization is not locally decidable. Further, the order in which buffers are assigned is nondeterministic, exploding the search space of possible buffer utilizations. This phenomena, which our proofs rely on, is what we call *buffer stealing*. For example, in a system corresponding to the variable widget (see Figure 5), the first process to send its message gets the buffer, and the other process remains blocked until the arbitrator performs the receives. This stealing corresponds to fixing a value of a variable. Similarly, the system corresponding to the disjunction widget allocates buffers for each of the term processes. However, if the buffer is stolen in all terms, corresponding to a falsifying assignment, then the system will deadlock within the ring.

For completeness, we note the following corollary:

Corollary 5.6 *The Buffer Allocation Problem (*BAP$_r$*) is in* Σ_2**P**.

Proof: By Theorem 5.2, verifying that a token assignment is sufficient to prevent deadlock (BSP$_r$) is **coNP**-complete. Since we can nondeterministically guess a sufficient token assignment, the result follows. ■

5.3 The Nonblocking Buffer Allocation Problem

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$_r$) is to determine the minimum number of buffers needed to achieve nonblocking sends.

Formally, the corresponding decision problem is this: given a communication graph G and an integer k, determine if there exists a token assignment of size k such that no colouring sequence on G blocks. Recall that a colouring sequence does not block if, whenever a send vertex is coloured yellow, rule $recv \xrightarrow{\bullet} yel$ may be applied to the corresponding receive vertex.

Let P_i and P_j, $j \neq i$, be two process components. Given two vertices, $v_{i,c}$ and $v_{i,t}$, in P_i, $t > c$, vertex $v_{i,t}$ is **communication dependent** on vertex $v_{i,c}$ if $v_{i,c}$ is the start vertex or if there exists a vertex $v_{j,d} \in P_j$, such that there is a path from $v_{i,c}$ to $v_{j,d}$ and the arc $(v_{j,d}, v_{i,t})$ is in A (see Figure 7). Vertex $v_{i,t}$ is **terminally communication dependent** on vertex $v_{i,c}$ if $v_{i,t}$ is communication dependent on $v_{i,c}$ and is not communication dependent on the vertices $v_{i,l}$, $c < l < t$. The algorithm depicted in Figure 8 computes an optimal token assignment such that no colouring sequence on G can block.

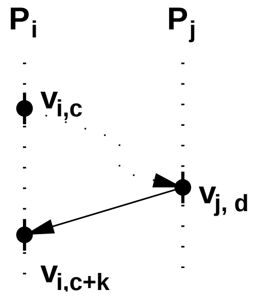

Fig. 7. $v_{i,t}$ is communication dependent on $v_{i,c}$.

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

Fig. 8. Algorithm for computing an optimal nonblocking buffer assignment.

Remark 5.7 *In a system corresponding to communication graph G, the time between a message arriving at process i and its receipt corresponds to the interval $I_{i,t}$. 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 within process p_i.*

Computing the terminal communication dependencies for G can be done via dynamic programming in $O(|V|n)$ time, where V is the vertex set of G and n is the number of process components. 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. Since the number of arcs is bounded by $3|V|/2$ and the pairwise comparison takes n steps, the traversal takes $O(|V|n)$ time.

Next, computing the $O(|V|)$ intervals, $I_{i,t}$, 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.

5.3.1 Proof of Correctness of the Nonblocking Buffer Allocation Algorithm

Lemma 5.8 *Let G be a communication graph. For all vertices $v_{i,c}, v_{j,d} \in G$; if $v_{j,d}$ is a send vertex and there exists a path from the vertex $v_{i,c}$ to vertex $v_{j,d}$, then vertex $v_{j,d}$ cannot be coloured yellow until vertex $v_{i,c}$ is coloured green.*

Proof: By rule *send→yel*, the predecessor of $v_{j,d}$ must first be coloured green before $v_{j,d}$ can be coloured yellow. Since rules *send→grn*, and *recv→grn*imply that the predecessors of a green vertex must be green, the result follows. ■

Corollary 5.9 *Let G, $v_{i,c}$, and $v_{j,d}$ be as in Lemma 5.8 and let $v_{i,t}$ be the receive vertex corresponding to the send vertex $v_{j,d}$. Rule recv$\overset{\bullet}{\to}$yel will never be applied to vertex $v_{i,t}$ before vertex $v_{i,c}$ is coloured green.*

The preceding corollary implies that a token, which is needed to colour the receive vertex $v_{i,t}$ yellow, need not be available until the vertex on which $v_{i,t}$ is terminally communication dependent is coloured green. Hence, it is sufficient to ensure token availability just before colouring the respective send vertex green; this is also necessary.

Theorem 5.10 *Given G, let $v_{i,c}$ be a send vertex and $v_{i,t}$ be a receive vertex that is terminally communication dependent on vertex $v_{i,c}$. A token for the application of rule recv$\overset{\bullet}{\to}$yel on arc $(v_{j,d}, v_{i,t})$ must be available as soon as vertex $v_{i,c}$ is coloured green.*

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

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

Thus, if a receive vertex r is terminally communication dependent on a send vertex s, then it is necessary and sufficient that a token, which is used to apply rule *recv*$\overset{\bullet}{\to}$*yel* to receive vertex r, must be available as soon as the send vertex s is coloured green; the start vertex may be thought of as a special send vertex. Since the interval corresponding to r begins when s is coloured green, and ends when

r is coloured green, a token must be available for the $recv \xrightarrow{\bullet} yel$ rule, which can occur during this interval. Computing the maximum overlap of intervals yields the required number of tokens.

5.3.2 Example Use of the NBAP$_r$ *Algorithm*

To demonstrate the NBAP$_r$ algorithm we have implemented it, and analyzed the pipe-and-roll parallel matrix multiplication algorithm [10]. The program has one control process and a number of worker processes arranged in a 2 dimensional mesh. We ran the NBAP$_r$ 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 9. The corresponding optimal buffer assignment is listed in the second column of Table 1.

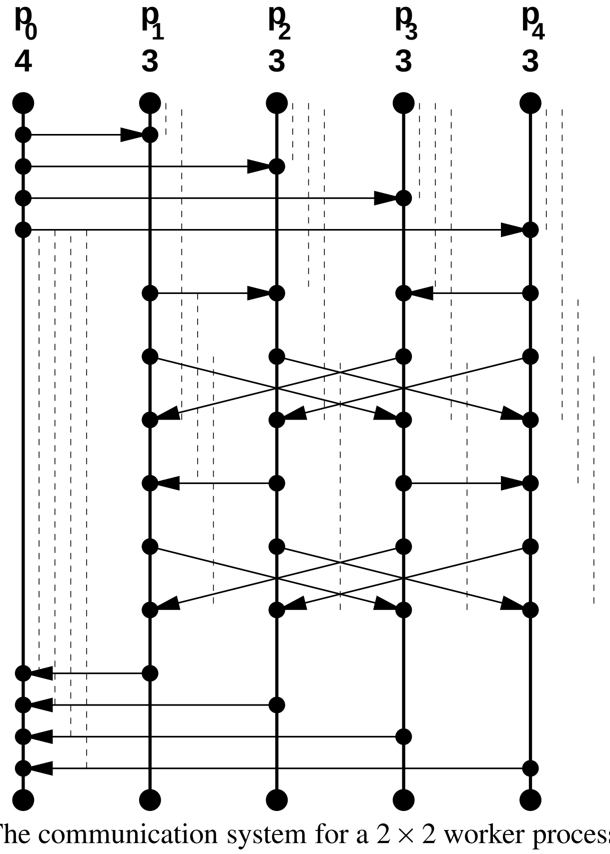

Fig. 9. The communication system for a 2×2 worker process mesh.

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$_r$ 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.

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 running the NBAP_r algorithm on the 2×2 worker example.

5.3.3 *Approximating* BAP_r *with* NBAP_r

The NBAP_r algorithm is useful for determining a token assignment that prevents deadlock, that is, approximating BAP_r. Since a nonblocking colouring sequence does not deadlock, a token assignment determined by the NBAP_r algorithm ensures that the graph is deadlock free. However, the token assignment may be far from optimal. A simple example of this phenomena is a two process component graph comprised of n arcs emanating from the first component and incident on the second. Such a graph requires zero tokens to avoid deadlock, but requires n tokens to be block free. Thus, the aforementioned token assignment may entail many more tokens than required.

6 Buffer Allocation in Systems with Send Side Buffers

In this section we consider the second of the four buffer placement strategies: send side buffers. Buffers are now allocated on the sending process side if the receive is not ready to accept the message. Correspondingly, the token pool used when applying rule $recv \stackrel{\bullet}{\rightarrow} yel$ to the receive vertex of arc (s,r) belongs to the process component containing the send vertex s. We call this the **send side allocation scheme**.

The Buffer Allocation Problem (BAP_s) remains intractable. The problem is conjectured to be **NP**-complete (see the following paragraph). The **NP**-hardness follows from the observation that each t-ring in the construction in Theorem 5.1 has to have a token assigned to a process component pair in order to prevent deadlock. It does not matter if the token is allocated from the token pool of the sending or the receiving process component. Hence, the reduction used in Theorem 5.1 can be applied with no modification.

We conjecture that the corresponding Buffer Sufficiency Problem (BSP_s) is in **P**. This is because the relative order in which tokens from a particular token pool are utilized is invariant with respect to the colouring sequences. Hence, we believe that the determining sufficiency is similar to the nonblocking buffer allocation problem and hence is in **P**. If this is the case, BAP_s is **NP**-complete.

The Nonblocking Buffer Allocation Problem (NBAP_s) 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 in Figure 8.

7 Buffer Allocation in Systems with Send and Receive Side Buffers

So far we have considered systems exclusively with send side or receive side buffers. In this section we investigate systems with buffers on both the send and the receive sides; many communication systems use per-host buffer pools for both receiving and sending messages. The choice of where to buffer the message—on the sender or on the receiver—increases the difficulty of determining the system's properties.

We assume a lazy mechanism for utilizing buffers: first use a buffer from the sender's pool. If none is available, use a buffer from the receiver's pool. If neither is available, attempt to free a send side buffer by transferring its contents to a buffer belonging to the corresponding receiver. Intuitively, the system attempts to maximize buffer use, without attempting to predict the future.

The corresponding colouring game allows tokens to be allocated from the pools belonging to both the sending component and the receiving component. Correspondingly, a lazy token utilization scheme is used: let (s_i, r_j) be a communication arc from process component P_i to process component P_j. The following rules apply during the application of rule $recv \overset{\bullet}{\rightarrow} yel$ to vertex r_j:

(1) If a token belonging to component P_i is available, use it.
(2) Otherwise, if a token belonging to component P_j is available, use it.
(3) Otherwise, if a token belonging to component P_i is currently placed on arc (t_i, r_k), $t_i \in P_i$, $r_k \in P_k$, and a token belonging to component P_k is available. Then the token on arc (t_i, r_k) may be replaced with the one belonging to P_k, freeing a token to be used in the current application of rule $recv \overset{\bullet}{\rightarrow} yel$.

We call this the **mixed allocation scheme**.

Not unexpectedly, the Buffer Allocation Problem (BAP_{sr}) remains intractable within the mixed allocation scheme. This is because the receive side allocation scheme, which provides no choice of token pools, can be simulated within the mixed allocation scheme. Concretely consider the receive side allocation scheme analyzed in Section 5: to simulate the receive side allocation scheme on communication graph G, within the mixed allocation scheme, each arc in G is replaced by the widget illustrated in Figure 10. Since vertex q cannot be coloured green until vertex r is coloured yellow, and component P' has no tokens, applying rule $recv \overset{\bullet}{\rightarrow} yel$ to r requires that P_j has an available token, regardless of whether P_i has an available token.

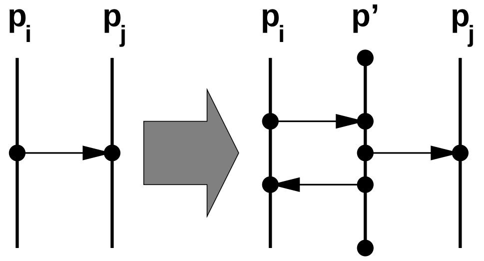

Fig. 10. Nullifying send side token pools.

Similarly, the Buffer Sufficiency Problem (BSP$_{sr}$) within the mixed allocation scheme is also **coNP**-complete. The hardness follows from Theorem 5.2 and the preceding argument. Since a colouring sequence also serves as a deadlock certificate in this case, the **coNP**-completeness result follows.

The interesting property of the mixed allocation scheme is that the Nonblocking Buffer Allocation Problem (NBAP$_{sr}$) is intractable; the choice of token pools increases the search space of solutions exponentially! The reduction is from 3SAT.

Theorem 7.1 *The Nonblocking Buffer Allocation Problem (*NBAP$_{sr}$*) is* **NP***-hard.*

Proof: Let F be an instance of 3SAT, comprising n variables, labeled x_i, $i = 1 \ldots n$, and c clauses. We construct a communication graph G such that there exists a token assignment of $n+2$ tokens that prevents any colouring sequence from blocking on G if and only if the corresponding assignment satisfies F.

The graph G comprises $2n+3$ process components: the first $2n$ are labeled P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \ldots n$, and the remaining three process components are labeled P, Q_0 and Q_1, respectively. The graph is divided into $c+1$ epochs: epoch 0 corresponds to the variable assignment, and epochs 1 through c correspond to clause evaluation.

In epoch 0 each process component P_{x_i} contains a single send vertex s_i that is adjacent to the receive vertex r_i located in epoch 0 of process component $P_{\bar{x}_i}$. Process component Q_0 (and Q_1) contains four vertices: two receive vertices $q_{0,1}$ and $q_{0,2}$ (respectively $q_{1,1}$ and $q_{1,2}$), followed by two send vertices $t_{0,1}$ and $t_{0,2}$ (respectively $t_{1,1}$ and $t_{1,2}$). Finally, process component P contains eight vertices: two send vertices, $s_{0,1}$ and $s_{0,2}$, that are adjacent to vertices $q_{0,1}$ and $q_{0,2}$; two receive vertices, $r_{0,1}$ and $r_{0,2}$, that are adjacent to $t_{0,1}$ and $t_{0,2}$; two more send vertices, $s_{1,1}$ and $s_{1,2}$, that are adjacent to $q_{1,1}$ and $q_{1,2}$; and two more receive vertices, $r_{1,1}$ and $r_{1,2}$, that are adjacent to $t_{1,1}$ and $t_{1,2}$. See Figure 11. Epoch 0 has two important properties.

Property 7.2 *Any token assignment must assign at least one token to either component P_{x_i} or $P_{\bar{x}_i}$ to prevent the colouring sequence from blocking after colouring vertex s_i yellow.*

Property 7.3 *A token assignment on G having only $n+2$ tokens must assign two tokens to process component P to prevent a colouring sequence from blocking after yellow colouring one of the send vertices $s_{0,1}$, $s_{0,2}$, $s_{1,1}$ or $s_{1,2}$.*

Proof: Since n tokens must be allocated to the process components P_{x_i} or $P_{\bar{x}_i}$, $i = 1, \ldots, n$, this leaves only two tokens to be allocated. Since the colouring rule sequence *send*$\rightarrow$*yel*, *recv*$\xrightarrow{\bullet}$*yel*, *send*$\rightarrow$*grn*, *send*$\rightarrow$*yel*, *recv*$\xrightarrow{\bullet}$*yel*can colour send vertices $s_{0,1}$ and $s_{0,2}$, or send vertices $s_{1,1}$ and $s_{1,2}$, component pairs (P, Q_0) and (P, Q_1) must each have two tokens between them. This can only happen by assigning the tokens to P. ∎

A corollary of these properties is that once a legal token assignment is made, no colouring sequence will block in epoch 0. The choice of allocating the token on P_{x_i} versus $P_{\bar{x}_i}$ corresponds to fixing the variable assignment.

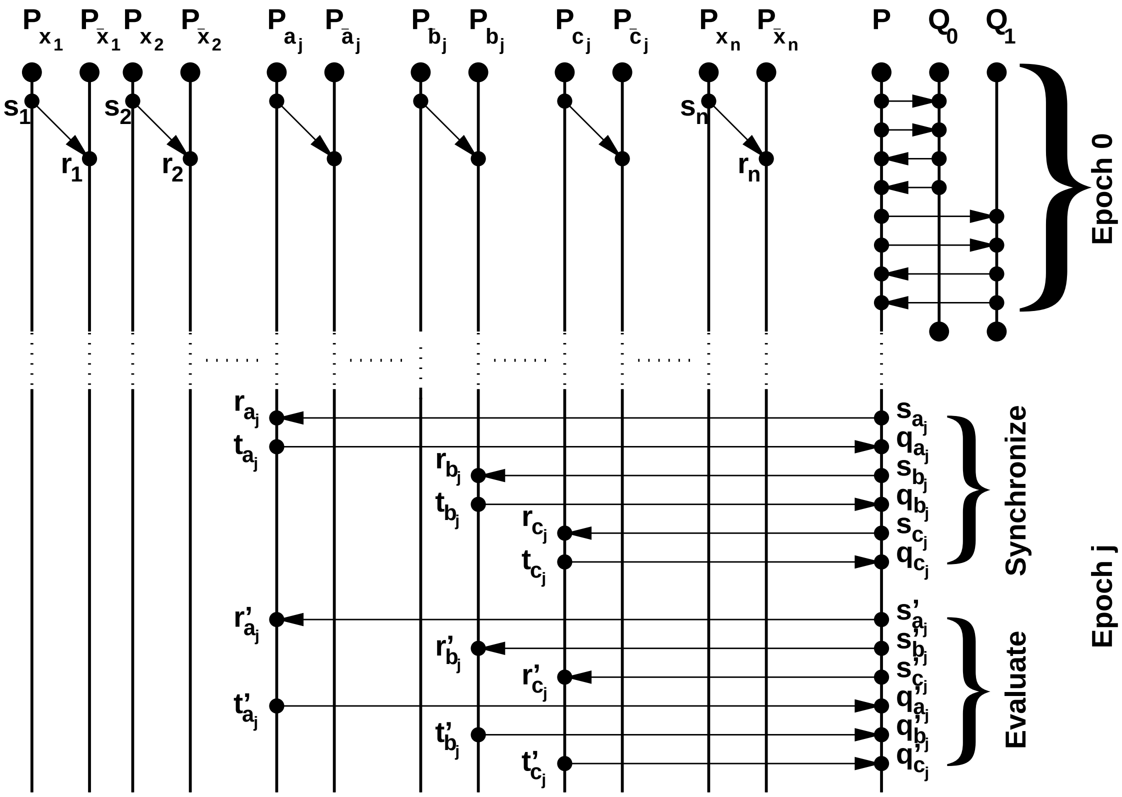

Fig. 11. Reduction from 3SAT to NBAP$_{sr}$.

For $j = 1 \ldots c$, epoch j corresponds to the jth clause. Each epoch comprises two parts of six arcs each: the synchronization part and the evaluation part. Four process components are involved in an epoch: the three components, P_{a_j}, P_{b_j}, and P_{c_j}, whose labels are the literals in the jth clause, where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$, and component P, which is involved in every epoch. Epoch j of component P_{a_j} comprises four vertices: receive vertex $r_{a_j,j}$, send vertex $t_{a_j,j}$, receive vertex $r'_{a_j,j}$, and send vertex $t'_{a_j,j}$. Process components P_{b_j} and P_{c_j} are analogously formed.

In epoch j component P has 12 vertices, the first six are these: send vertex $s_{a_j,j}$, receive vertex $q_{a_j,j}$, send vertex $s_{b_j,j}$, receive vertex $q_{b_j,j}$, send vertex $s_{c_j,j}$, and receive vertex $q_{c_j,j}$. These are followed by three send vertices: $s'_{a_j,j}$, $s'_{b_j,j}$, and $s'_{c_j,j}$, and three receive vertices: $q'_{a_j,j}$, $q'_{b_j,j}$, and $q'_{c_j,j}$.

Each vertex $s_{l,j}$ is adjacent to vertex $r_{l,j}$, each vertex $t_{l,j}$ is adjacent to vertex $q_{l,j}$, each vertex $s'_{l,j}$ is adjacent to vertex $r'_{l,j}$, and each vertex $t'_{l,j}$ is adjacent to vertex $q'_{l,j}$; see Figure 11. For conciseness we drop the last index, j, if it is obvious from the context. Epoch j has three important properties:

Property 7.4 *If vertex q'_{c_j} (in epoch j) is coloured green and vertex $s_{a_{j+1}}$ (in epoch $j+1$) is still red, then no tokens that belong to component P are assigned to arcs. The same applies to vertex pairs (q_{a_j}, s_{b_j}), (q_{b_j}, s_{c_j}), and (q_{c_j}, s'_{a_j}), also in epoch j.*

Proof: All ancestors of q'_{c_j} must be coloured green and all descendants of $s_{a_{j+1}}$ must be coloured red. This includes all vertices in G, except some vertices s_i and r_i in epoch 0, which are not adjacent to vertices in component P. Hence, the tokens belonging to P are not assigned to any arc. The same argument applies to the other vertex pairs. ■

Property 7.5 *A colouring sequence on G can block only when yellow colouring receive vertices r'_{a_j}, r'_{b_j}, r'_{c_j}, q'_{a_j}, q'_{b_j}, or q'_{c_j}.*

Proof: As a corollary of properties 7.2 and 7.3, no colouring sequence can block in epoch 0. Thus, we need only check that no colouring sequence can block in the first part of epoch j, $j = 1 \ldots c$.

By property 7.4, if s_{a_j} is red and its predecessor is green, then no tokens of P are in use. Hence, to colour s_{a_j} green, a token is available to colour r_{a_j} yellow. Since vertex r_{a_j} is a predecessor of t_{a_j}, vertex r_{a_j} must be coloured green before t_{a_j} may be coloured yellow. Thus the token is freed before t_{a_j} is coloured green, and may be used to colour vertex q_{a_j} yellow after t_{a_j} is coloured yellow. A similar argument applies to the vertices r_{b_j}, q_{b_j}, r_{c_j}, and q_{c_j}. ■

Property 7.6 *A colouring sequence can block in epoch j if and only if none of the three process components, P_{a_j}, P_{b_j}, and P_{c_j}, have a token assigned.*

Proof: For the 'if' direction consider a colouring sequence that colours vertex q_{c_j} green, but has not yet coloured vertex s'_{a_j} yellow. By definition, blocking does not occur, if rule $recv \overset{\bullet}{\rightarrow} yel$ may always be applied to colour a receive vertex yellow. To colour the send vertices s'_{a_j}, s'_{b_j}, and s'_{c_j} yellow and then green, the receive vertices r'_{a_j}, r'_{b_j}, and r'_{c_j}, must be coloured yellow via rule $recv \overset{\bullet}{\rightarrow} yel$. Since the receive vertices r'_{a_j}, r'_{b_j}, and r'_{c_j} are not ancestors of the send vertices s'_{a_j}, s'_{b_j}, and s'_{c_j}, none of the receive vertices need be coloured green before the send vertices are coloured yellow. However, component P has only two tokens, and none of components P_{a_j}, P_{b_j}, P_{c_j} have any. Hence, rule $\mathit{recv}\xrightarrow{\bullet}\mathit{yel}$ can only be invoked twice, instead of the requisite three times. Thus, a colouring sequence can block in epoch j.

For the 'only if' direction we claim that if a literal component P_{a_j}, P_{b_j}, or P_{c_j} has a token, rule $\mathit{recv}\xrightarrow{\bullet}\mathit{yel}$ can be invoked on any of the six receive vertices r'_{a_j}, r'_{b_j}, r'_{c_j}, q'_{a_j}, q'_{b_j}, and q'_{c_j}. Since r'_{a_j} is a predecessor of t'_{a_j}, r'_{a_j} must be coloured green before t'_{a_j}, and hence before q'_{a_j} is coloured yellow. Thus, the same token that was allocated upon the application of rule $\mathit{recv}\xrightarrow{\bullet}\mathit{yel}$ to vertex r'_{a_j}, may also be allocated upon the application of rule $\mathit{recv}\xrightarrow{\bullet}\mathit{yel}$ to vertex q'_{a_j}; the same argument is applicable to vertices q'_{b_j} and q'_{c_j}. Applying rule $\mathit{recv}\xrightarrow{\bullet}\mathit{yel}$ to vertices r'_{a_j} and r'_{b_j}, uses the two tokens from component P. To colour vertex r'_{c_j} yellow there are three possible scenarios:

(1) the colouring sequence has already freed one of the tokens, allowing it to be reused,
(2) component P_{c_j} has a token, in which case it is used, or
(3) component P_{a_j} (or P_{b_j}) has a token, in which case it replaces the token used to yellow colour vertex r'_{a_j} (or r'_{b_j}) and the freed token is used to colour vertex r'_{c_j}.

Since at least one component P_{a_j}, P_{b_j}, or P_{c_j} have a token, the claim is proven. ■

By property 7.6 a colour sequence will block in epoch j if and only if none of the process components P_{a_j}, P_{b_j}, or P_{c_j} has a token, which corresponds to the jth clause having no literals that are true. Thus, a token assignment of size $2n+2$ prevents any colouring sequence on G from blocking if and only if the corresponding assignment satisfies F. ■

8 Buffer Allocation in Channel Based Systems

In channel based systems processes communicate via pairwise connections that are created at start-up. Each connection, called a channel, is specified by its end-points and is used by one process to send messages to the other. Each channel functions independently of other channels in the system, and resources such as buffers are allocated on a per channel basis, rather than per process. Finally, channels behave like queues, that is, messages are removed from the channel in the same order that they are inserted.

Channels may either be unidirectional, comprising source and destination end-points, or bidirectional, comprising two symmetric end-points. In the former case, only the source process may insert messages into the channel and only the destination process may remove messages from the channels. A bidirectional channel is equivalent to two unidirectional channels, allowing both processes to insert and remove messages from the channel. Here we only consider unidirectional channels.

Except for buffer allocation, channel based communication does not differ from the previously described send/receive mechanism. In fact, an unbuffered channel communication is just a synchronous send/receive communication. Thus, we can derive similar results for channel based systems.

In the corresponding colouring game tokens, are allocated to channels (component pairs) instead of to components. This change does not change the properties used in our proofs. In fact, Lemma 4.1 may be used unchanged. We call this the **per channel allocation scheme**.

8.1 The Buffer Allocation Problem

The corresponding Buffer Allocation Problem (BAP_{sr}) is this: given a communication graph G and an integer k, determine whether there exist a token assignment of size k, such that no colouring sequence deadlocks on G. Even though token utilization, during the colouring of a communication graph, is only dictated by the communication arcs within a process component pair, determining the number of tokens needed remains **NP**-hard. The proof is similar in spirit to Theorem 5.1.

Theorem 8.1 *The Buffer Allocation Problem (*BAP_{sr}*) is* **NP**-*hard.*

Proof: We prove this by reducing 3SAT to BAP_{sr}. For any 3SAT instance F we construct a corresponding communication graph G—polynomial in size of F—such that for a token assignment of size n, any colouring sequence will complete on G if and only if the corresponding variable assignment satisfies F.

Let F be an instance of 3SAT on n variables and comprising c clauses. The construction is nearly identical to that in Theorem 5.1, except for the widgets representing the clauses of F. The graph G has $2n$ process components that are labeled by the literals of F, P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \ldots n$. Each component comprises $c+1$ epochs, where each epoch contains zero or two vertices.

As in Theorem 5.1, epoch 0 fixes a variable assignment. In epoch 0 each component has two vertices: a send vertex, labeled s_{x_i} (or $s_{\bar{x}_i}$), and a receive vertex r_{x_i}, (respectively $r_{\bar{x}_i}$), $i = 1 \ldots n$. Vertex s_{x_i} is adjacent to vertex $r_{\bar{x}_i}$, and vertex $s_{\bar{x}_i}$ is adjacent to vertex r_{x_i}; this is a 2-ring, identical to epoch 0 in Theorem 5.1. Epoch 0 has the the following property:

Property 8.2 *Any colouring sequence on G will deadlock in epoch* 0 *unless each* *process component pair has a token assigned to the token pool of either $(P_{x_i}, P_{\bar{x}_i})$, or $(P_{\bar{x}_i}, P_{x_i})$, $i = 1 \ldots n$. Thus, the token assignment must be of at least size n. (Follows from Lemma 4.1.)*

Property 8.2 yields the following correspondence between assignments on F and token assignments of size n.

Property 8.3 *The corresponding token assignment of a variable assignment on F assigns a token to the channel $(P_{x_i}, P_{\bar{x}_i})$ if x_i is true, or to $(P_{\bar{x}_i}, P_{x_i})$ if x_i is false.*

The jth epoch represents the jth clause of F, denoted (a_j, b_j, c_j), where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. The process components P_{a_j}, $P_{\bar{a}_j}$, P_{b_j}, $P_{\bar{b}_j}$, P_{c_j}, and $P_{\bar{c}_j}$ form a 6-ring, while the remaining components have no vertices in the jth epoch. Process component P_{a_j} has two vertices in the jth component: a send vertex, $s_{a_j,j}$, and a receive vertex $r_{a_j,j}$; similarly, the other five components have a send and receive vertex that are correspondingly named. The arcs linking the 6 components are these: $(s_{a_j,j}, r_{\bar{a}_j,j})$, $(s_{\bar{a}_j,j}, r_{b_j,j})$, $(s_{b_j,j}, r_{\bar{b}_j,j})$, $(s_{\bar{b}_j,j}, r_{c_j,j})$, $(s_{c_j,j}, r_{\bar{c}_j,j})$, and $(s_{\bar{c}_j,j}, r_{a_j,j})$. These form a 6-ring, as illustrated in Figure 12. The key property of the jth epoch is this:

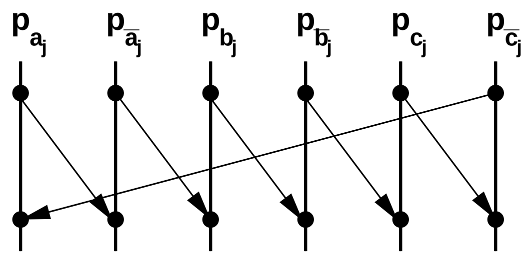

Fig. 12. The clause representation in epoch j.

Property 8.4 *No colouring sequence on G will deadlock in the jth epoch if and only if at least one of the channels has a token: $(P_{a_j}, P_{\bar{a}_j})$, $(P_{\bar{a}_j}, P_{b_j})$, $(P_{b_j}, P_{\bar{b}_j})$, $(P_{\bar{b}_j}, P_{c_j})$, $(P_{c_j}, P_{\bar{c}_j})$, $(P_{\bar{c}_j}, P_{a_j})$. (Follows from Lemma 4.1.)*

A refined version of property 8.4 is more useful:

Property 8.5 *For any token assignment of size n such that no colouring sequence deadlocks on G in epoch 0, no colouring sequence on G will deadlock in the jth epoch if and only if at least one of the channels $(P_{a_j}, P_{\bar{a}_j})$, $(P_{b_j}, P_{\bar{b}_j})$, and $(P_{c_j}, P_{\bar{c}_j})$, has a token.*

Proof: By property 8.2, all token assignments that do not cause deadlock in epoch 0 only assign tokens to channels of the form $(P_{x_i}, P_{\bar{x}_i})$ or $(P_{\bar{x}_i}, P_{x_i})$. Hence, only channels $(P_{a_j}, P_{\bar{a}_j})$, $(P_{b_j}, P_{\bar{b}_j})$, and $(P_{c_j}, P_{\bar{c}_j})$ can have a token. By property 8.4, no colouring sequence on G will deadlock in epoch j if one of these channels has a token. ■

We claim that given a token assignment of size n, any colouring sequence will complete on G if and only if the corresponding variable assignment satisfies F.

If an assignment x satisfies F, then every clause has at least one literal that evaluates to true. By Property 8.3, in each of the j epochs at least one of the channels listed in Property 8.5 will be allocated a token. Hence, by Property 8.5 no colouring sequence will deadlock on G.

If an assignment x does not satisfy F then there is at least one clause in which all literals are false. Let (a_j, b_j, c_j) be the unsatisfied clause. By property 8.3, the corresponding token assignment will not assign a token to $(P_{a_j}, P_{\bar{a}_j})$, $(P_{b_j}, P_{\bar{b}_j})$, or $(P_{c_j}, P_{\bar{c}_j})$, hence, by Property 8.5, all colouring sequences will deadlock.

Thus, NBAP$_{sr}$ is **NP**-hard. ■

Since tokens are assigned on a per channel basis, token usage depends only on the two process components that comprise the channel. Consequently, the sufficiency of a token assignment can be verified in linear time. Thus, the easier problem BSP$_{sr}$ is in **P**, implying that BAP$_{sr}$ is **NP**-complete. We describe the verification algorithm and prove its correctness.

To verify the sufficiency of a token assignment, perform a colouring on G: at each step of the colouring a vertex of G is coloured according to the rules in section 3. Using a queue to keep track of colourable vertices, means that determining which vertex to colour next takes $O(1)$ time. Since each vertex changes colour at most twice—the maximum length of any colouring sequence is $2|V|$ colourings—colouring a graph takes $O(|V|)$ time. The token assignment is sufficient if and only if the colouring sequence completes. The algorithm's correctness follows immediately from the following theorem: any colouring sequence on G completes if and only if some colouring sequence on G completes. Thus, a token assignment is sufficient if and only if some colouring sequence on G completes.

Theorem 8.6 *Let G be a communication graph and B a token assignment on G. Any colouring sequence on G completes if and only if a colouring sequence on G completes.*

Proof: For any communication graph G, we construct a new graph G' where every token is simulated by a process component, the size of the corresponding token assignment is zero, and every colour sequence on G corresponds to a colouring sequence on G', such that a colouring sequence on G completes if and only if the corresponding colouring sequence on G' completes. Since the token assignment on G' is zero, by Lemma 4.2 a colouring sequence on G' completes if and only if every colouring sequence on G' completes. Hence, every colouring sequence on G completes if and only if a colouring sequence on G completes.

To simulate an m token channel (a channel that has been assigned m tokens) m process components are chained together. For each channel (P, Q) with m tokens,

m process components $P_1, P_2, \ldots, P_m$ are interspersed between P and Q. The channel (P,Q) is replaced with these channels: $(P,P_1),(P_1,P_2),\ldots,(P_{m-1},P_m),(P_m,Q)$. Each arc from P to Q is replaced by a chain of arcs from $P \to P_1 \to P_2 \to \ldots \to P_{m-1} \to P_m \to Q$. The replacement is illustrated in Figure 13.

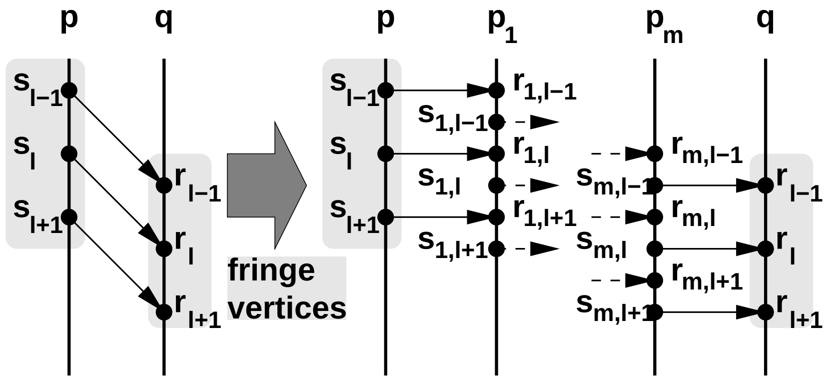

Fig. 13. Simulating m tokens by m components.

We claim that a colouring sequence, Σ, on G will deadlock if and only if the corresponding colouring sequence, Σ', on G' deadlocks. First, we construct the correspondence and argue its correctness. Second, we argue that sequence Σ deadlocks on G if and only if the corresponding sequence Σ' deadlocks on G'. Finally, we apply Lemma 4.2 to prove our result.

Since the transformation is iterative—each m token channel is independent of the other channels—it is sufficient to derive the correspondence between the colouring sequence on G and the graph G' where a single m token channel has been replaced. Let (P,Q) denote the channel in G that is replaced in G'.

Let $(s_l, r_l) \in G$, $l = 1,2,\ldots$, denote the arcs from process component P to Q. The corresponding paths in G' are

$$(s_l, \underbrace{r_{1,l}, s_{1,l}}_{P_1}, \underbrace{r_{2,l}, s_{2,l}}_{P_2}, \ldots, \underbrace{r_{m,l}, s_{m,l}}_{P_m}, r_l),$$

where each arc $(r_{k,l}, s_{k,l})$ is within process component P_k and each arc $(s_{k,l}, r_{k+1,l})$ is between process components P_k and P_{k+1}; the vertices s_l and r_l, $l = 1,2,\ldots$ are called the **fringe** vertices.

A colouring sequence Σ can be represented as a sequence of differences (or moves), δ_i, between every two consecutive colourings χ_i and χ_{i+1}. The sequence $\Delta_\Sigma = \delta_1\delta_2\ldots$ is a sequence of colouring game moves $\delta_i = \langle v, \text{colour}\rangle$ such that applying δ_i to colouring χ_i yields χ_{i+1}, the next colouring in Σ; Δ_Σ can be derived from Σ and, Σ can be derived from Δ_Σ and G. The sequence Δ_Σ comprises two types of moves: those that colour fringe vertices, called **fringe moves**, and those that do not, called **normal moves**.

Given a colouring sequence Σ on G, we transform it into the corresponding colouring sequence Σ' on G'. The transformation replaces some fringe moves in sequence Δ_Σ with sequences of moves, resulting in the corresponding move sequence $\Delta_{\Sigma'}$. This sequence comprises normal moves and **added moves**; added moves are a mixture of fringe moves and moves on the vertices within the added components P_i. There are four types of fringe moves in Δ_Σ: colour a send vertex s_l yellow ($\langle s_l, \text{yel}\rangle$), colour a send vertex s_l green ($\langle s_l, \text{grn}\rangle$), colour a receive vertex r_l yellow ($\langle r_l, \text{yel}\rangle$), and colour a receive vertex r_l green ($\langle r_l, \text{grn}\rangle$). The transformation is performed in the order that the moves occur in sequence Δ_Σ.

- If $\delta_i = \langle s_l, \text{yel}\rangle$, then no action is taken.
- If $\delta_i = \langle s_l, \text{grn}\rangle$, we replace it with the sequence

  $$\langle r_{1,l}, \text{yel}\rangle, \langle s_l, \text{grn}\rangle, \langle r_{1,l}, \text{grn}\rangle, \langle s_{1,l}, \text{yel}\rangle,$$

  suffixed by the sequences

  $$\langle r_{j,l}, \text{yel}\rangle, \langle s_{j-1,l}, \text{grn}\rangle, \langle r_{j,l}, \text{grn}\rangle, \langle s_{j,l}, \text{yel}\rangle, \quad j = 2 \dots k-1$$

  where k is the smallest integer such that the move $\langle s_{k,l-1}, \text{grn}\rangle$ has not yet been inserted into the move sequence Δ_Σ, that is, vertex $s_{k,l-1}$ has not yet been coloured green.
- If $\delta_i = \langle r_l, \text{yel}\rangle$, we remove it from the sequence; it is restored when we replace the move $\langle r_l, \text{grn}\rangle$.
- If $\delta_i = \langle r_l, \text{grn}\rangle$ we replace this move with the sequence

  $$\langle r_l, \text{yel}\rangle, \langle s_{m,l}, \text{grn}\rangle, \langle r_l, \text{grn}\rangle,$$

  suffixed with the sequences

  $$\langle r_{g_j,h_j}, \text{yel}\rangle, \langle s_{g_j-1,h_j}, \text{grn}\rangle, \langle r_{g_j,h_j}, \text{grn}\rangle, \langle s_{g_j,h_j+1}, \text{yel}\rangle, \quad j = 0 \dots k-1,$$

  where $g_j = m - j$, $h_j = l + 1 + j$, and k is the smallest integer such that the move $\langle s_{m-k,l+1+k}, \text{yel}\rangle$ has not yet been inserted into the sequence, that is, vertex $s_{m-k,l+1+k}$ has not yet been coloured yellow. Since the head of this sequence colours $s_{m,l}$ green, $r_{m,l+1}$ could be coloured yellow, if $s_{m-1,l+1}$ is yellow, then $s_{m-1,l+1}$ could be coloured green followed by $r_{m,l+1}$ and finally $s_{m,l+1}$ could be coloured yellow; this colouring cascades down the added process components.

It is important to note that each of the replacement sequences is maximal, that is, no additional valid colouring moves on the chain process components P_i, $i = 1 \dots m$, may be suffixed to them. The new sequence looks like this:

$$\Delta_{\Sigma'} = \overbrace{\delta_1 \dots \delta_{h_1}}^{\text{normal moves}} \underbrace{\delta'_1 \dots \delta'_{g_1}}_{\text{added moves}} \overbrace{\delta_{h_1+1} \dots \delta_{h_2}}^{\text{normal moves}} \underbrace{\delta'_{g_1+1} \dots \delta'_{g_2}}_{\text{added moves}} \dots .$$

Since G is a contraction of G', all normal vertices are coloured by $\Delta_{\Sigma'}$ in the same order as in Δ_Σ. Recall that normal vertices are not adjacent to the process component chain, and hence, are not affected by the transformation. While normal vertices within process components P and Q may depend on the order that the fringe vertices are coloured, the dependence is via process arcs, not communication arcs. Consequently, the normal vertices only depend on the order that the fringe vertices are coloured green. Fortunately, this order is preserved. By inspection, the replacement sequences of moves are valid. Thus, the transformed sequence $\Delta_{\Sigma'}$ is valid. Additionally, all green colouring moves on fringe vertices are preserved by the transformation; a vertex is coloured green by Δ_Σ if and only if the corresponding vertex is coloured green by $\Delta_{\Sigma'}$. The following property is key:

Property 8.7 *Δ_Σ deadlocks on G if and only if $\Delta_{\Sigma'}$ deadlocks on G'.*

Proof: By contradiction, suppose that Δ_Σ deadlocks on G while $\Delta_{\Sigma'}$ can be extended, that is, another vertex colouring move may be suffixed to $\Delta_{\Sigma'}$. Let v be the vertex that can be coloured by the extension. Vertex v may either be a normal vertex, a fringe vertex, or a vertex belonging to a process chain. The latter is impossible because every replacement sequence of moves is maximal.

If v is a normal vertex, then its predecessors are either a normal vertex or a fringe vertex that has been coloured green. Since the transformation preserves the colourings of normal vertices and the order in which vertices are coloured green, if $\Delta_{\Sigma'}$ can be extended by colouring v, then so can Δ_Σ, which is a contradiction.

If v is a fringe vertex, there are four cases: either v is a send vertex s_l being coloured yellow or green, or v is a receive vertex r_l being coloured yellow or green. The transformation does not affect moves that colour send vertices yellow and such a colouring only depends on its process component predecessor being green. Hence, if the colouring can be suffixed to $\Delta_{\Sigma'}$, it can also be suffixed to Δ_Σ; resulting in a contradiction. If the extension colours the send vertex green, this means that the original sequence Δ_Σ can be extended by either adding the colourings $\langle s_l, \text{grn}\rangle$ or $\langle r_l, \text{yel}\rangle\langle s_l, \text{grn}\rangle$, depending on whether r_l has been coloured yellow or not in the original sequence Δ_Σ; thus, it is a contradiction.

Similarly, if v is a fringe receive vertex being coloured green, this is not possible, because the transformation colours fringe receive vertices yellow, then green, by a single replacement sequence. Finally, if v is a fringe receive vertex r_l that can be coloured yellow, the original sequence Δ_Σ can be extended by the move $\langle r_l, \text{grn}\rangle$, because in the original sequence the corresponding send vertex s_l has already been coloured green. Thus, we have another contradiction.

In the other direction, if the original sequence can be extended, then transforming the extension of the sequence Δ_Σ yields an extension to the presumably deadlocked sequence $\Delta_{\Sigma'}$. Thus, Δ_Σ deadlocks on G if and only if $\Delta_{\Sigma'}$ deadlocks on G'. ■

A corollary of Property 8.7 is that the colouring sequence Σ deadlocks if and only if the colouring sequence Σ' deadlocks.

By Lemma 4.2 a colouring sequence on G' completes if and only if all colouring sequences on G' complete. Hence, a colouring sequence on G completes if and only if all colouring sequences on G complete. ■

Corollary 8.8 *A colouring sequence on G completes if and only if the token assignment is sufficient.*

8.2 The Buffer Allocation Problem

For the Nonblocking Buffer Allocation Problem, the algorithm derived in section 5.3 suffices with a small modification. Since the token pools are per channel, rather than per process component, the computation must be performed on a per pool basis. Hence, there is an additional factor of n in the runtime. Since each process may be using up to n channels, the runtime of the algorithm becomes $O(|V|n^2 + |V|n\log(|V|n))$; the cost increases because the number of allocations to be made becomes quadratic in n.

9 Conclusion

As message passing becomes increasingly popular, the problem of determining k-safety plays an increasingly important role. The relevance of this problem is grows as more and more functionality of message passing systems is off-loaded to the network interface card, where limited buffer space is a serious issue. Even if message passing is kept in main memory, buffer space can still be limited due to the sometimes very large data sets used in many parallel and distributed programs. Unfortunately, determining k-safety 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. On the positive side, if we require that no send blocks, we provide a polynomial time algorithm for computing the minimum number of buffers. By allocating this number of buffers, safe execution is guaranteed. In addition, we have implemented the NBAP$_r$ algorithm, and it is now part of the Millipede debugging system [19].

For systems with only send buffers, the Buffer Allocation Problem remains **NP**-complete. In addition, we conjecture that the Buffer Sufficiency Problem can be solved in polynomial time because the order of the sends in each process is fixed.

The Nonblocking Buffer Allocation problem for systems with only send buffers can be solved in polynomial time.

For systems with both send and receive buffers, the Buffer Allocation Problem as well as the Buffer Sufficiency Problem remain intractable. More interestingly, the Nonblocking Buffer Allocation problem has become intractable.

For systems with unidirectional channel buffers, both the Buffer Sufficiency Problem as well as the Nonblocking Buffer Allocation Problem have polynomial time algorithms. However, the Buffer Allocation Problem still remains an **NP**-complete problem. The results (conjectures) are summarized below.

	Buffer Placement			
Problem	Receive	Send	Send & Receive	Channel
BAP	**NP**-hard	**NP**-hard	**NP**-hard	**NP**-complete
BSP	**coNP**-complete	(**P**)	**coNP**-complete	**P**
NBAP	**P**	**P**	**NP**-hard	**P**

9.1 Strategies for Reducing Buffer Requirements

There are several strategies that a programmer can use to reduce the likelihood of deadlock when only a few buffers are available.

The obvious solution is to use synchronous communication, which does not require any buffers at all. However, this is not always desirable.

For efficiency reasons asynchronously buffered communication is often preferred. To decrease the risk of deadlock the programmer can introduce epochs that are separated by barrier synchronizations. This might reduce the number of buffers needed for each epoch, as no buffer requirement spans an epoch boundary. If each epoch only needs a small number of buffers, the risk of deadlock due to buffer insufficiency is reduced.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] G. Burns, R. Daoud and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994.

[13] 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.

[14] 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.

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

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

[17] C. Keppitiyagama and A. Wagner. Asynchronous MPI messaging on myrinet. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*. IEEE, 2001.

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

[19] J. B. Pedersen. *Multi-level Debugging of Parallel Message Passing Programs*. PhD thesis, University of British Columbia, Canada, 2003. In preparation.

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

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