PREVENTING DEADLOCK WITH DYNAMIC MESSAGE SCHEDULING

ERNESTO GOMEZ, YASHA KARANT, KEITH SCHUBERT

Department of Computer Science, California State University San Bernardino

ABSTRACT. Although deadlock is not completely avoidable in distributed and parallel programming, we here describe theory and practice of a system that allows us to limit deadlock to situations in which there are true circular data dependences or failure or processes that compute data needed at other processes. The system works by dynamically scheduling messages, using semantic information on variable use to ensure correctness. This allows us to guarantee absence of deadlock in SPMD computations (absent process failure).

1. BACKGROUND

Deadlock is a major problem in parallel programming. Substantial amounts of effort have been dedicated to deadlock detection, which would allow breaking the deadlock so processing can continue. We believe that the wrong problem is being addressed, due to application of deadlock criteria that depend on assumptions not appropriate to parallel execution.

The standard definition of deadlock was given by Coffman, Elphick and Shoshani in 1971[6], citing four conditions that are required to hold simultaneously for deadlock to exist in a system: 1) mutual exclusion, 2) hold and wait (of some resource), 3) no preemption and 4) circular wait. The standard approach to deadlock, following this definition is to detect deadlock by using condition 4 and to break it by preempting one of the processes involved in the circular wait, breaking condition 3. These same conditions are frequently cited as applying to message passing parallel computation, in which the resource in question is a message.

The application of standard deadlock analysis to message passing systems is, however, incorrect, because the standard analysis assumes that the resources involved in a circular wait are independent of the processes holding them. This is not the case for message passing systems because a message that contains data computed by the process holding it is not independent of the process; preempting the process destroys the message and does not resolve the deadlock. Further, the absence of a process that is expected to send a message can produce a condition locally indistinguishable from deadlock without a circular wait condition.

We propose a modification of the standard definition of deadlock for message passing systems that corrects the above cited problems: We observe that condition 1 always exists in a message passing system, and condition 3 is not relevant because preemption destroys resources. We are left only with condition 2, which may occur where a process must wait to receive a message before sending, and a modified

Date: 27 October 2003.

condition 4 in which deadlock requires either a circular wait or a communication pattern, circular or otherwise, in which some sender process is missing.

Our modification of condition 4 unfortunately voids most deadlock detection methods. We are left with alternatives of trying to make deadlock structurally impossible, detecting deadlocking communication patterns through static analysis of code, or attacking condition 2, hold and wait.

We consider the use of fused send and receive operations as in the MPI sendrecv instruction,[10] Planguage fusion communications[26], and Hoare's atomic messages in CSP[14]. We conclude that, while the exclusive use of such constructs makes writing deadlock difficult, it also substantially hinders the expression of parallelism in communications.

We limit ourselves to the analysis of SPMD execution (Single Program, Multiple Data, model used by MPI and many other parallel programming systems), in which we have multiple processes each of which is executing the same code. This allows us to relate particular sends and receives at different process, and permits an analysis of data dependences. We show that such analysis is insufficient to detect deadlock, which can also arise due to message scheduling problems, even in the absence of circular data dependence. We show that even an analysis that takes schedule and data dependences into account may be insufficient, since deadlock can also arise in the allocation of buffers or message channels by the communication system. We show that such deadlock may exist even when there are no data dependences preventing execution and even when there are sufficient system resources. We have not been able to devise an analytic method that can statically detect a deadlock condition in all cases. Although we are not convinced that such analysis is impossible, we believe that it is certainly complex and possibly impractical.

We instead attack condition 2, hold and wait, through dynamic, out-of-order communication scheduling. Our methods are based on the overlapping protocol first described by Gomez and Scott in 1998[22]. Overlapping was originally designed to dynamically schedule messages to compensate for load imbalances in irregular problems by scheduling messages opportunistically. Correctness is preserved through information on variable use in the program code, available at compile time through standard data flow analysis. An individual process is made to wait only if some pending communication must have taken place before the process performs a read or write on some variable involved in the communication.

We show that the dynamic message scheduling system that implements the overlapping protocol is capable of resolving schedule and resource deadlocks, as long as no process is missing and no circular data dependence exists. The combination of static data dependence analysis and an overlapping protocol runtime system then allows us to guarantee absence of deadlock at execution for a broad class of SPMD programs. We show experimental results as proof of concept.

2. Deadlock

There is no universal agreement on what is meant by deadlock. Tanenbaum [3] characterizes deadlock as a subset Σ of states such that there is no transition out of Σ, and there are no transitions in Σ which cause forward progress. Apt and Olderog, [1] similarly characterize deadlock, as a configuration that has no successor in the state transition relation. A more common characterization is found in Lynch [8], who defines deadlock in terms of a circular dependence between processes. Elmasri

and Navathe [9] give a similar definition, as do most database references of which we are aware.

Silberschatz [2] and Tanenbaum [4] both quote the results derived by Coffman, Elphick and Shoshani [6], citing four conditions that are **required to hold simultaneously** for deadlock to exist in a system: 1) mutual exclusion, 2) hold and wait, 3) no preemption and 4) circular wait.

Conditions 2 and 4 mean that deadlock is a network property; it cannot be detected by examining the state of a single process. In an asynchronous network in particular, a process cannot infer that some other process will never release a resource from the fact that it has not done so even over an arbitrarily long time.

It is commonly assumed that the resource being held is independent of the process holding it. This is not true for messaging systems where the resource is data generated by a process. In particular, we can get deadlock by the definitions of [3] and [1] even if condition 4 does not hold, if some process p does not execute a send for which some other process q is waiting. In this case p waits forever, but there is no cycle because q can continue. This situation is sometimes called **live-lock,** for example by Elmasri and Navathe [9]. A different characterization of live-lock is given by Ullman [23], who describes a situation in which, even though a resource does become periodically available, a particular blocked process never in fact acquires it. We consider this to be deadlock of a subset of processes; it clearly falls under the definition of deadlock given by Apt and Olderog [1].

A general approach to deadlock detection relies on constructing a graph of dependences between all processes and detecting a cycle. This approach, however, can not detect live-lock or non-cyclic deadlock that may occur in some definitions. Even if deadlock is detected, recovery generally requires killing one of the processes that holds a resource. In parallel computation, our resources are messages between processes. If we kill a process to break a circular dependence, we lose data needed for a correct result. In the absence of some mechanism for check-pointing and restarting individual processes to allow the regeneration of messages, deadlock detection is useful only to determine that a parallel computation should be aborted. We will not here discuss methods for rolling back processes to previous states; we will merely note that such methods (which are in fact used in database systems, see [9]) are likely to be complicated, expensive and slow.

Database and operating systems often attempt to prevent circular dependence by ensuring that all resources required by a transaction or process are acquired first, before the process is allowed to proceed (see [2],[3],[9]). If the resources are not available, the process is aborted. Again, this only protects against some definitions of deadlock.

Avoidance is the strategy we normally use with message passing systems. It is possible to inadvertently write sends and receives in a way which will deadlock (see MPI examples [10]). We try to carefully write sends and receives so that they match, and so that we avoid a circular dependence. This is difficult in practice, particularly in the presence of many processes with the possibility that a cycle will involve some large combination of them. The difficulty of avoiding such dependences can lead to a very conservative programming style; for example BSP [11], in which an algorithm is divided between computation stages, in which no communication is permitted, and synchronization stages where all communication takes place.

In view of the disagreement over the nature of deadlock, we prefer to define absence of deadlock. First we need some definitions relating to SPMD.

3. SPMD

We limit our discussion to SPMD, characterized by execution of multiple copies of the same program text, each acting on possibly different data. We call the execution of each separate copy of the code a *process*, and refer to the execution of an entire parallel program consisting of multiple processes, as a *parallel execution*. SPMD may include *task parallelism*, in which different operations may be performed concurrently at different processes. Foster [12] points out that task parallelism appears necessary to deal with programs that require adaptive or irregular data structures.

SPMD processes are independent programs which can each have private local memory, and may also have shared memory. Each process progresses independently except as specified by the programmer with synchronization or communication statements. The execution model maps well to a distributed computing environment.

SPMD has become a common style of parallel program, implemented for example by the Planguages [17] and by a host of message passing systems, including MPI [10], PVM [18] and many vendor-specific implementations. It has the advantage, for analysis, of providing a known starting point for the parallel execution. Within SPMD, we consider the *static* process model, in which the number of processes is held constant; this is the execution model of Planguages and MPI.

We represent the program as a control flow graph (CFG) $F = \{V, A, s, E\}$, where V is a set of nodes representing basic blocks, A is the set of directional arcs connecting them, s is the start node, and E is a set of end nodes [24]. We model serial execution of a program as a path in F the from s to an end node $e \in E$. Program execution progresses through a series of steps which may be numbered starting from 0, and continues until the program reaches an end state or diverges. We define serial state in terms of CFG node and data in memory when that node is executed:

Definition 3.1. ***Serial state and transition****: The jth state of a serial execution is a tuple:*

$\sigma_j = (x_j, D_j)$

Here x_j denotes execution of basic block x at step j, and D_j is the set of data values stored in variables by the process when execution reaches the entry point of the basic block after performing j transitions from the start of execution. We denotes serial transition from a serial state to the next state by:

$\sigma_j \rightarrow \sigma_{j+1}$

D_j depends on the entire execution history. Every state implies execution of one basic block, and every transition follows an arc of the CFG. The basic blocks appear in an order determined by the CFG, however the state number j is a simple count of steps or transitions and always increases. If the instructions in the code are deterministic, then the results of applying them to the same input data will be the same output data and selection of the same path through the CFG. . A deterministic execution may not terminate if it includes an infinite cycle (called a diverging execution), but it is still deterministic if this always happens on the same data.

We will impose an additional condition:

Condition 3.2. In the absence of interaction with other processes, each serial process p in a parallel execution E progresses from whatever state it is in to a successor state, until it reaches an end state.

We now define a parallel SPMD state as the state of a group of serial processes:

Definition 3.3. *Let Γ be the set of all processes in an SPMD execution. A **parallel state** of a process group $G \subseteq \Gamma$ executing on a CFG is:*

$S_{G,J} = \{x_J, D_J^G\}$

*J is an index formed from the indices of nodes $x_J = \{x_{j_p} \mid p \in G\}$ and D_J^G is a list of mappings from the set of variables indexed by J to their values at the corresponding step j_p for each process. We call $S_{G,J}$ a **total state** if $G = \Gamma$.*

We consider that each process may be progressing at a different rate, as a reflection of reality. This implies that a parallel state has multiple possible successor states.

Definition 3.4. *A **parallel transition** is a relation $S_{G,J} \to S_{H,K}$ between a pair of parallel states such that $G \cap H \neq \emptyset$ where both G and H are subsets of Γ; for each $p \in G \cap H$, we have $k_p - j_p \leq 1$, for at least one p we have $k_p - j_p > 0$, and there exists a transition $S_{\Gamma,J_\Gamma} \to S_{\Gamma,K_\Gamma}$ such that $J \subseteq J_\Gamma$, $K \subseteq K_\Gamma$ and which is compatible with $S_{G,J} \to S_{H,K}$.*

This transition requires at least one of the serial processes in $G \cap H$ to advance by one step, and leaves open the possibility that any or all processes will advance by one step. We use the notation $S_{G,J} \to_p S_{H,K}$ to denote a parallel transition such that $p \in G \cap H$ and $k_p = j_p + 1$, in which process p increments its state. The p subscript $\to_p$ does not exclude the possibility that for some $p \neq q \in G \cap H$ we may also have $S_{G,J} \to_q S_{H,K}$.

Informally, a parallel SPMD execution is a set of serial processes, each of which executes the same program text, and all of which begin execution (logically) together. No process requires data that is computed at another process to begin execution.

Definition 3.5. *An **SPMD parallel execution** E is a sequence of transitions between total states:*

$E_\Gamma = S_{\Gamma,0} \to^* S_{\Gamma,End}$,

Given a CFG $\{V, A, s, E\}$, every process in $S_{\Gamma,0}$ executes node s, and every process in $S_{\Gamma,End}$ executes some node in E.

The existence of a parallel transition $S_{G,J} \to S_{H,K}$ does not require that every $p \in G$ have a serial transition. However, if $E_\Gamma = S_{\Gamma,0} \to^* S_{\Gamma,End}$ exists and reaches an end state, it must be the case that every process in it progresses from the start state to the end state. We require $E_\Gamma = S_{\Gamma,0} \to_p^* S_{\Gamma,End}$ for every $p \in \Gamma$.

We are not here in a position to say that the execution does in fact exist; we make the more limited point that if it does exist, then every process in it must have a set of transitions from beginning to end.

Each process in E_Γ could take a different path through the CFG, and so execute a different number of steps. If some process q takes fewer steps to reach an end state than some different process p, we may be able to identify different sequences of $\to$ transitions which represent the same execution.

Grouping processes by the paths they take through the CFG leads to an alternative definition of parallel execution as a directed graph of partial states instead of a sequence of total states. Such an execution can be uniquely described. We have developed a construct called an *Ipstream*, and a runtime system called *SOS* which support implicit process groups and give us such a description; this work is described in [7] and in a forthcoming paper.

4. Absence of deadlock

In the absence of failure of some individual serial process (condition 3.2) we recognize two reasons why an execution may not reach the end state: divergence, or deadlock. In divergence, all processes continue to advance indefinitely; for example because the exit condition from a loop is never met. In this paper, we will assume that processes do not diverge, and consider only deadlock. In a deadlocked execution some processes is unable to progress out of a state because a resource controlled by another process does not become available.

One way of characterizing a deadlocked parallel execution is by noting that the transition sequence $S_{\Gamma,0} \rightarrow^* S_{\Gamma,End}$ does not exist. This is not particularly useful since we would have to wait for infinite time to know that an execution is in fact deadlocked. (It does reflect a common situation in practice, however).

We find it more useful to define absence of deadlock as follows:

Definition 4.1. *We will say there is* ***absence of deadlock*** *in a parallel state $S_{G,J} \neq S_{\Gamma,End}$, if every $p \in G$ such that the serial state $\sigma_{j_p} \in S_{G,J}$ and $\sigma_{j_p} \notin S_{\Gamma,End}$ has a transition $S_{G,J} \rightarrow_p S_{H,K}$ to some other state $S_{H,K} \neq S_{G,J}$.*

That is, if a state is not deadlocked, every serial process that has not already reached the end has to be able to transition out of the state. Absence of deadlock captures the consequences of deadlock from any of the previous definitions. We believe it is more useful because it does not specify what the cause of deadlock might be.

If we assume that processes do not diverge or otherwise fail due to some local computational error, then the condition of serial progress 3.2 holds and each process should be able to reach an end state, unless it is prevented by the failure of some interaction with another process. This has the consequence (as before) that deadlock or its absence is a property of more than one process, and in general we would expect deadlock to result from some failure in communications.

5. Communications

In order to establish conditions for non-deadlocking and deterministic execution, we need to establish conditions on communications that preserve these goals. In this work we will restrict our considerations to message passing communications, and we will further assume these communications are reliable. In SPMD execution, each process normally has its own local copy of memory, which makes message passing a good match to the execution paradigm.

We consider that the semantics of message passing is given by **blocking communication** [18], characterized by a requirement that all participating processes wait at the communication statement, and must receive confirmation of success before being allowed to continue. A form of blocking communications is the default

mode of MPI messaging [2]. Blocking communications makes a message transmission into an atomic action and is generally accepted to define the semantics of message passing. However, many systems ([2][5][6][7]) implement less restrictive modes through buffering or other protocols, and these modes may allow execution of more parallelism than strict blocking communications would permit.

We restrict our consideration to processes that do not diverge or fail, so deadlock becomes a consequence of communication failures. Given also reliable communications, it is evident that the only way a point to point message can fail is if either the sender or receiver process does not execute its part of the message transfer. In SPMD this may happen if either send or receive statement is in conditionally executed code that is not executed at some process. This possibility is the subject of a paper in preparation. If all involved processes execute their respective sends and receives it is not possible under our assumptions that a single send-receive pair will fail.

5.1. **Communications patterns.** Therefore we need to analyze communication involving multiple processes and multiple messages. We will attempt to represent such communications as a directed graph which we will call a **communication pattern**. We will postpone for the moment the problem of how to determine which particular messages need to be considered as part of the same pattern,

We now consider how to express a communication pattern. It is not enough to simply graph data dependences between processes because communication frequently is written in such a way that some processes pass on messages to others. That is, the actual pattern of messages does not necessarily conform to the sources and destinations of data. Further, the order in which message code is written into a program can introduce schedule dependences unrelated to actual data dependence.

Consider an example:

Example 5.1. Take a ring of processes numbered $0 \leq p \leq N-1$. Assume each process needs to transfer a value contained in a local variable x_p to the next process in the ring, given by $(p+1) mod(N)$. Assume each process stores the received value in local variable y_p. There is no data dependence since no process needs to receive any value before performing its send, and because data sent and received is stored in different variables. If every process executes a receive from $(p-1) mod(N)$ before sending to $(p+1) mod(N)$, no process ever completes its receive because no process in the ring can reach its send statement. Schedule dependences at each process prevent execution. If send and receive instructions are exchanged at one process, the deadlock is broken.

A simple graph of communications in which nodes are processes and arcs are messages will not distinguish the two cases given above, one of which deadlocks and the other which does not. The problem is that arcs entering and departing a particular node imply a sequence which may or may not exist depending on the specific code in a node. In our example, there is a schedule dependence between a receive followed by a send at the same node, since the send can not occur until the receive finishes. At the single node where we exchange send and receive instructions and execute the send first, the send can succeed independent of the receive and there is no dependence.

A first attempt at a communication pattern definition is:

Definition 5.2. ***Communication pattern definition 1:*** *$P = \{V, A\}$ is a directed graph of communication dependences, constructed as follows:*

Let the set of nodes $v_{p,i} \in V$ be the set of communication statements involved in a set of messages, where i denotes the statement number and p denotes the process at which a particular statement is executed (note that in SPMD the same statement may be executed at multiple processes).

Construct A as follows: if $v_{p,i}$ is a send statement, and $v_{q,j}$ is the corresponding receive statement, place the arc $v_{p,i} \rightarrow v_{q,j}$ in A. If $v_{p,k}$ is a receive statement, and $v_{p,l}$ is a send statement at the same process, place $v_{p,k} \rightarrow v_{p,l}$ in A. Continue until no more arcs can be added.

Upon closer examination, however, this definition of communication pattern does not satisfy blocking semantics, although it may work for buffered communications.

Example 5.3. Consider again example 5.1, this time writing the send statement before the receive at every process, as follows:

$send(x) from(p) to(p+1) mod(N)$

$recv(y) from(p-1) mod(N)$

By definition 5.2 this would work. There are no dependence arcs linking statements in the same process, so the communication pattern is a series of disconnected arcs between send-receive statements at different processes. Since this pattern contains no cycles, if all designated processes participate, then the messages can be sent concurrently and the communication pattern succeeds.

A pattern of disconnected arcs suggests the possibility of execution in any order, but this is not the case here. In order for a send from, say, $p = 0$ to $p = 1$ to complete, process 1 must first have performed its send. The argument applies to any message between any pair of processes. Since processes are arranged in a ring, it follows that before any message pair can be completed, all sends must have been performed. If we are using blocking communications, however, no send can finish before its corresponding receive does. We see that example 5.3 cannot be executed by blocking communications. It requires non-blocking sends, achievable for example by buffering.

Let us try a different definition of communication pattern:

Definition 5.4. ***Communication pattern definition 2:*** *$P = \{V, A\}$ is a directed graph of communication dependences, constructed as follows:*

Let the set of nodes $v_{p,i} \in V$ be the set of communication statements involved in a set of messages, where i denotes the statement number and p denotes the process at which a particular statement is executed (note that in SPMD the same statement may be executed at multiple processes).

Construct A as follows: if $v_{p,i}$ is a send statement, and $v_{q,j}$ is the corresponding receive statement, place the arc $v_{p,i} \rightarrow v_{q,j}$ in A. If $v_{p,k}$ and $v_{p,l}$ are statements at the same process, place $v_{p,k} \rightarrow v_{p,l}$ in A. Continue until no more arcs can be added.

Definition 5.4 gives us blocking semantics by recognizing that execution cannot advance past either a send or a receive statement until its matching statement (at some other process) has also completed. Therefore there is a schedule dependence even when receive statements precede sends.

Using this definition, we see that the first alternative in example 5.1 does not work, but the second one would give us a directed acyclic graph in which the start

node is the send statement at the single process that performs its send first, and in which receives and sends at each process around the ring follow in order, until the receive at the first process is reached.

It is not clear, however, that example 5.3 can not be executed, since the graph of the communication pattern would still be acyclic: with all send statements first, each statement would have two departing arcs and each receive would have two entering arcs, one due to schedule from the preceding send at the same process and another representing data transmitted from a different process. Our communication pattern does not meet the classic conditions for deadlock, and yet we appear to have a situation in which no process can progress.

A failing of our definitions may be that send-receive pairs, which we regard as atomic, are still represented as arcs. We may try to remedy this:

Definition 5.5. ***Communication pattern definition 3:*** *$P = \{V, A\}$ is a directed graph of communication dependences, constructed as follows:*

Let $v_{p,i} \in V$ be communication statements involved in a set of messages, where i denotes the statement number and p denotes the process at which a particular statement is executed (note that in SPMD the same statement may be executed at multiple processes).

Construct V as follows: if $v_{p,i}$ is a send statement, and $v_{q,j}$ is the corresponding receive statement, let $n_{p,i,q,j} = v_{p,i} \rightarrow v_{q,j}$ be a node and place it in V. If $v_{p,k}$ is a send statement, and $v_{p,l}$ is a receive statement, both at process p, and the receive statement appears after the send statement in the program text, find a node $n_{p,k,q,l}$ in which $v_{p,k}$ appears as sender, and a node $n_{r,l,p,j}$ in which $v_{p,j}$ appears as receiver, and place an arc $n_{r,l,p,j} \rightarrow n_{p,k,q,l}$ in A. Continue until no more arcs can be added.

Definition 5.5 captures the schedule dependences in 5.1; since it collapses the send-receive pairs into single nodes, we see that the graph for example 5.3 is a cycle and would deadlock.

This definition gives us a sequence of nodes which are point to point messages, with dependences that indicate an order in which these messages can be sequentially executed. A cycle here would clearly imply deadlock, since no message in the cycle could execute before the previous message was executed, and there would be no way to select one particular message to execute first.

The problem with definition 5.5 is applying it in actual programming practice. The definition forces us to think of send-receive pairs as single entities, which may not be what they look like in the code. Furthermore, there is nothing in the definition that tells us which messages should be selected as the first or last messages in a communication pattern. Ultimately, we may need to apply definition 5.5 to every message in a program. (We may be able to restrict the set of messages in a communication pattern when we are writing the underlying code for a collective communication operation such as a broadcast, but the collective operation may then itself have to be a node in the communication pattern for the whole program).

We would like, however, to make a different point: we believe that the code of example 5.3 should be correct. Note that there is no conflict in the order in which data is computed or made available. In particular, sends make messages available before matching receives, and no send statement needs any data from another process. Although the second alternative in example 5.3 will certainly succeed without buffering, it sequentializes the messages in a way that may not be necessary. Much as virtual memory hides the absence of sufficient real memory

for a program to execute, we believe that a proper runtime system should allow maximal expression of concurrency and hide the absence of system resources such as buffers that might prevent code like example 5.3 to execute.

6. Dynamic message scheduling

The issue of schedule dependences and buffering is rendered moot by the use of a dynamic runtime scheduling system such as the one we described in [22].

Such a system queues multiple message actions as long as there are no data dependences that force any particular message to finish before another message. Execution of a particular send or receive action is independent of the order in which the actions were entered on the queue resolving all schedule conflicts. Our system additionally matches sends and receives at different processes before executing a transfer of data, and uses semantic information on variable use to block processes when needed. As a consequence, each message can be sent in synchronous mode without buffering. With such a system, only data dependences need to be considered for a message pattern to succeed - for example we have performed experiments in which the system automatically resolves purely schedule conflicts such as the ring we have described in which all receives appear before send statements. Since a fully sequential set of message sends is always possible for a non-cyclic pattern of data dependences, a dynamic scheduling system will select such an order if required, but it can take advantage of availability of hardware allowing concurrent messages.

Our original system did not fully realize the implications of only restricting communications on data dependences, but in fact enforced some schedule dependences as well in an ad-hoc manner. We have now implemented a system that in fact enforces only data dependences. Our function library is originally conceived to support irregular scientific computation, and includes features not directly relevant to the present paper.

We term our system SOS, which stands for IpStreams, Overlapping and Shortcutting. Shortcutting and overlapping were described in [22]. Ipstreams are a construct that supports groups of processes formed implicitly by program logic, and are described in a paper currently in preparation. Here we are concerned only with the overlapping functions of the system.

In the SOS system, a communication instruction does the following:

Computes the communication pattern required to carry out the specified instruction, as a series of point to point sends.

Assigns a Finite State Machine (FSM) to control the communication.

Queues the particular sends and receives required at the local process to a queue maintained by the SOS runtime system.

If the particular communication is a send, the SOS runtime sends a status message to the intended receiver indicating that the particular send is ready to execute. If the queued communication is a receive, the runtime system waits for a status message from the sender before proceeding.

SOS periodically examines all queued communications and carries out whichever sends or receives can proceed to completion, in any order - for example, sends for which matching receives are posted will execute before sends that have no matching receives, even if they were queued later. Similarly, receives will execute as their

FIGURE 6.1. **shift**

```
send(x, from myIdent, to (myIdent+1) modulo N )
receive(y, at myIdent, from (myIdent-1) modulo N)
```

FIGURE 6.2. **schedule deadlock**

```
receive(y, at myIdent, from (myIdent-1) modulo N)
send(x, from myIdent, to (myIdent+1) modulo N )
```

matching sends become available, not necessarily in the order in which they are queued.

Two additional calls to the runtime system are used to indicate variable use to the system. These are SOSdoLH(id) and SOSdoRH(id), where ID refers to a variable identifier. The LH call is entered immediately before the next appearance of variable ID on the left-hand side of an expression, and the RH call before the next use of ID on the right-hand side of an expression.

6.1. **Implementation of example:** We will now further examine variations on our examples 5.1 and 5.3, and show their implementation in actual code. We will compare the implementation using our SOS system with MPI code.

Consider, specifically, a set of 3 processes {0,1,2}, logically arranged as a ring. Each process has a local value X which must be copied to a variable Y at the next process. We implement the logic of example 5.3 in pseudo-code given in figure 6.1. In the figure, myIdent is the process number of the local process, and N=3 is the number of processes. The code in figure 6.1 requires buffering to execute, as we have seen in considering example 5.3. It is possible that a very short message in a slow system will simply be in the connecting network between execution of send and receive statements, but it is likelier that a message will need to sit in a system or network buffer. The MPI standard talks about this situation [10], stating that code such as that of figure 6.1 works only in the presence of sufficient system buffering, and otherwise deadlocks. As a result, it calls such code 'unsafe', but not incorrect.

Figure 6.2 illustrates an extreme case of schedule deadlock, in which receives appear before sends. Such a schedule deadlock is easy to detect in this code, but may not be easily visible in more complicated code, particularly if the communication statements are surrounded by logic that leads to this kind of deadlock only some of the time. Note that there is no data dependence problem, since the send and receive statements read and write, respectively, independent variables.

We would like to reschedule the messages to resolve buffer and schedule deadlock. For example, the deadlock of 6.1 could be resolved if each send and its matching receive were executed synchronously, possibly in some serial order. The deadlock of6.2 would be resolved if the receives were not blocking and permitted computation to continue to reach the sends, or if the order of sends and receives could be swapped.

However, any attempt to reschedule presents the difficulty that the same communication pattern can also arise from incorrect code. Suppose, for example, that we rewrite the code of figures 6.1 and 6.2 to transfer the value of X into the same variable X at another process.

FIGURE 6.3. **buffering and barrier required between sends and receives**

```
send(x, from myIdent, to (myIdent+1) modulo N )
receive(x, at myIdent, from (myIdent-1) modulo N)
```

FIGURE 6.4. **circular data dependence**

```
receive(x, at myIdent, from (myIdent-1) modulo N)
send(x, from myIdent, to (myIdent+1) modulo N )
```

The code of figure 6.3 depends critically on all the sends happening before any of the receives, otherwise the wrong value of x could be sent. The code of figure 6.4is a true circular data dependence that cannot be executed with the expectation of any meaningful result.

To resolve deadlock in the above (and other similar) cases while preserving the execution, the communication system uses semantic information on variable access from the program to determine how or if to resolve communications deadlock.

SOS (Fortran) code for figure 6.1 is given in figure 6.5.

MPI code with this logic does not execute at all with synchronous sends, and only works with MPI standard sends if messages are not overly large. On our system, using MPICH and Red Hat Linux Version 7.1, messages larger than about 100KBytes deadlock). Much more complicated MPI code is required, inverting the send/receive order at one process, to ensure that messages of any size can be used (figure 6.6)

The SOS code (figure 6.5) executes by scheduling the sends and receives as matching pairs. As long as there is some sequence in which the paired sends and receives can be executed, the system will succeed. If the code runs, for example, on a single processor the paired sends and receives are executed serially; on a cluster of workstations parallelism is achieved which may depend on progress of individual processes or particular system resources.

7. PERFORMANCE

Figure 7.1 shows some sample runs of the MPI code of figure 6. Figure 8 shows runs of the SOS code of figure 6.5, but inverting the order of sends and receives as in figure 6.2 (schedule deadlock).

All runs were on the Raven cluster at Cal State San Bernardino. This is a cluster of 14 Compaq dual processor machines, with 1.4 GHz Pentium III processors and connected through Compaq gigabit Ethernet NICs. Message size is given in 32 bit words. We see that, although deadlock is avoided, these runs in which computation load is insignificant fully reveal significant added costs for SOS.

Figure 7.3 shows a comparison of timings for a test application using MPI and SOS collective operations (SOS is implemented with MPI send and receive instructions). The computation is a modified downhill simplex algorithm described in [22]. There are two broadcasts and one reduction per iteration in the main program loop. In this application which has substantial computational imbalance for algorithmic

FIGURE 6.5. **Fortran for 6.1 with SOS communications**

```
      dest1 = me+1
      if( me+1.GT.top ) dest1=0
      src1 = me
      dest2 = me
      src2 = me-1
      if( me-1.LT.0 ) src2=top

      print *,'at ',me,src1,'->',dest1

      call SOSdoRH(ix)
      call SOSpoint2point(ix,iy,MSIZE,0,0,MPI_REAL8,src1,dest1)

      print *,'at ',me,src2,'->',dest2

      call SOSdoLH(iy)
      call SOSpoint2point(ix,iy,MSIZE,0,0,MPI_REAL8,src2,dest2)

c     the following two calls ensure the program blocks until communication
c     completes

      call SOSdoRH(ix)
      call SOSdoRH(iy)

      print *,'at ',me,x,' done =', y
```

reasons, SOS, even though implemented on top of MPI point to point communications, shows a significant advantage over MPI collective communications. (This work was carried out on the network of workstations at the High Performance Computing Lab at the University of Chicago - see http://hpcl.cs.uchicago.edu/). In this example we see that SOS communication costs are effectively hidden as irregular computation load increases.

8. SOS OPERATION

Calls to SOSpoint2point (as in figure 6.5) have the semantics of a send at the sender process, and a receive at the receiver process, but have the same syntactic form at both. Collective communications calls (broadcast and reduction) are resolved into particular sends and receives at each participating process.

The system maintains a queue at each process that holds pending send and receive actions. SOS periodically interrupts the process and checks the queue. Any queued communication that can be performed are performed and removed from

FIGURE 6.6. **Fortran code for figure 6.2 with MPI and inverting order of sends and receives at one process**

```
c       invert order on top
        if( me.EQ.top ) then

         if( me.EQ.dest2 )then
          print *,'at ',me,src2,'->',dest
call
MPI_Recv(y,MSIZE,MPI_REAL8,src2,1,MPI_COMM_WORLD,status,ierror)
         endif

         if( me.EQ.src1 )then
          print *,'at ',me,src1,'->',dest1
call
MPI_Send(x,MSIZE,MPI_REAL8,dest1,1,MPI_COMM_WORLD,status,ierror)
         endif

        else

         if( me.EQ.src1 )then
          print *,'at ',me,src1,'->',dest1
call
MPI_Send(x,MSIZE,MPI_REAL8,dest1,1,MPI_COMM_WORLD,status,ierror)
         endif

         if( me.EQ.dest2 )then
          print *,'at ',me,src2,'->',dest2
call
MPI_Recv(y,MSIZE,MPI_REAL8,src2,1,MPI_COMM_WORLD,status,ierror)
         endif

        endif

        print *,'at ',me,x,' done =', y
```

the queue. Since every pending communication is checked each time, they may be executed in an order different from the order in which they were posted.

The current version of SOS operates on top of the MPICH distribution of MPI. The system uses asynchronous, non-blocking MPI receives and synchronous, blocking and unbuffered MPI sends. An SOSpoint2point call at the receiver causes an

FIGURE 7.1. **figure 7: Test runs, MPI program of figure 6.6**

```
[ernesto@raven deadlock]$ run 3 case1 2
 at  1 x=   101.
 at  1 1-> 2
 at  1 0-> 1
 at  1  101. done =  100. time=  0.00041875
 at  2 x=   102.
 at  2 1-> 2
 at  2 2-> 0
 at  2  102. done =  101. time=  0.0005945
 MPI - message size = 2000 reps= 4
 at  0 x=   100.
 at  0 0-> 1
 at  0 2-> 0
 at  0  100. done =  102. time=  0.00067225

[ernesto@raven deadlock]$ run 3 case1 2
 at  1 x=   101.
 at  1 1-> 2
 at  1 0-> 1
 at  1  101. done =  100. time=  0.248482
 at  2 x=   102.
 at  2 1-> 2
 at  2 2-> 0
 at  2  102. done =  101. time=  0.26606225
 MPI - message size = 2000000 reps= 4
 at  0 x=   100.
 at  0 0-> 1
 at  0 2-> 0
 at  0  100. done =  102. time=  0.3830305
```

MPI asynchronous receive to be posted, and queues an instruction to check for completion of that receive. At the sender, SOSpoint2point queues an MPI synchronous send instruction. The runtime system at each process executes a send and removes it from the queue when a status message is arrives indicating that the matching receive has been posted. A receive is removed from the queue when a call to MPI indicates the data has arrived.

The FSM controlling each communication executes the overlap protocol described in [22]. Briefly, a process is blocked when a variable that has a pending receive appears on the right-hand side of an expression, or when a variable that has a pending send needs to be updated (appears on the left-hand side).

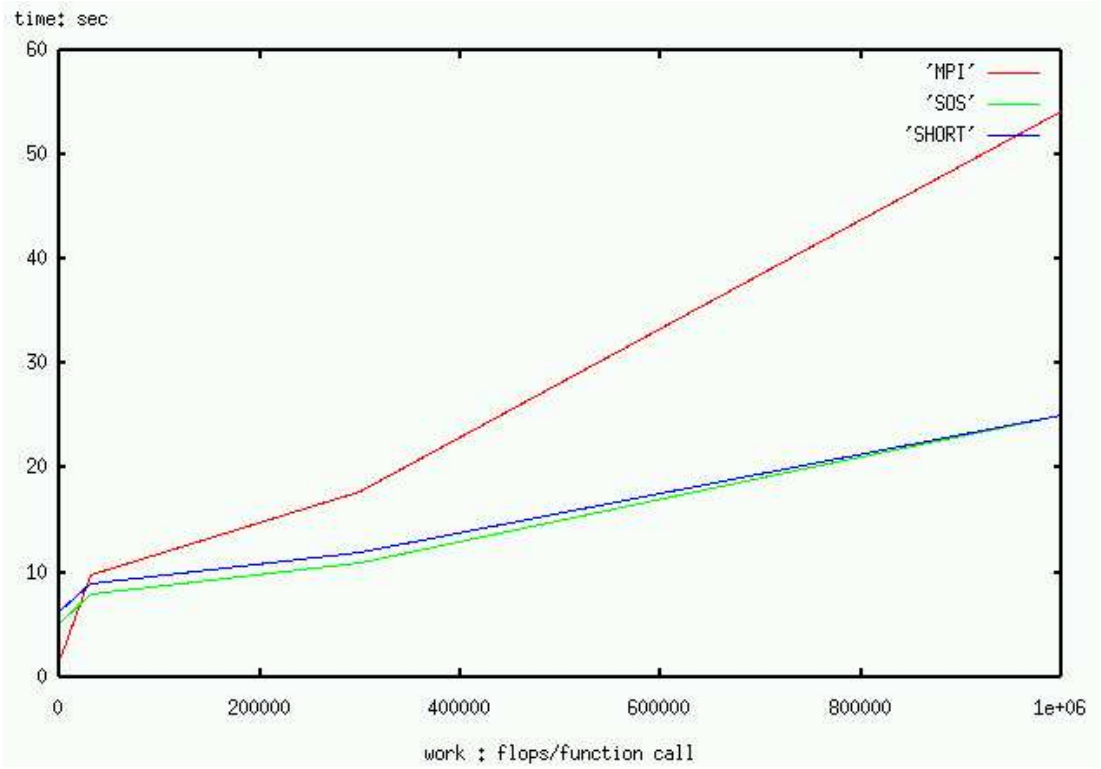
FIGURE 7.2. **Test runs, SOS program of figure 6.5 modified to implement the schedule deadlock of figure 6.2**

```
[ernesto@raven deadlock]$ run 3 case4 2
 at  2 x=   102.
 at  2 1-> 2
 at  2 2-> 0
 at  2  102. done y=  101. time=  0.0113015
 at  1 x=   101.
 at  1 0-> 1
 at  1 1-> 2
 at  1  101. done y=  100. time=  0.01121925
 SOS - message size = 2000 reps= 4
 at  0 x=   100.
 at  0 2-> 0
 at  0 0-> 1
 at  0  100. done y=  102. time=  0.011025

[ernesto@raven deadlock]$ run 3 case4 2
 at  1 x=   101.
 at  1 0-> 1
 at  1 1-> 2
 at  1  101. done y=  100. time=  1.10046675
 at  2 x=   102.
 at  2 1-> 2
 at  2 2-> 0
 at  2  102. done y=  101. time=  1.10594225
 SOS - message size = 2000000 reps= 4
 at  0 x=   100.
 at  0 2-> 0
 at  0 0-> 1
 at  0  100. done y=  102. time=  1.0583775
```

SOS requires more messages than standard message passing systems because it requires status messages in addition to the messages that actually carry data. However, in the asynchronous environment of a cluster of workstations, SOS transmits first between processes that are running faster than others, which reduces the need for such processes to wait for slower processes. In the pure communication test runs reported here, the extra cost of SOS status messages is fully visible, since the test program performs no computation and therefore processes do not have any

FIGURE 7.3. **Scientific application performance**



significant synchronization delays. However if there is either a significant load imbalance or significant asymmetry in processor performance between different nodes in a cluster, SOS effectively hides the communication load.

9. How far should we take deadlock avoidance?

The first issue we confronted when we first realized the deadlock avoidance possibilities in the SOS technology was correctness. The argument for correctness previously given in [22] does not explicitly address dynamically breaking a cycle introduced by scheduling, but close examination shows it in fact applies; the protocol preserves dependences between processes. The fact that a receive is written before a send in figure 6.2 does not create a data dependence, because the statements refer to different variables. (In fact, the protocol can be proved correct, but the proof is too lengthy for this paper).

We here limit ourselves to considering our example as motivation. The overlap protocol requires an LH call before a receive, since it is a write to the received variable, and an RH call before a send, since it is a read of the sent variable. Inserting an LH between send and receive statements in figure 6.3 would force processes to complete each send before receiving the same variable, requiring buffering. Inserting an RH call between receive and send in figure 6.4 would force the receive, which

is a write to the variable, to complete before the send of the same variable could start, which would give us deadlock.

At first we also were concerned that resolving schedule deadlock from code such as figure 6.4 would change the meaning of the program as written, and therefore should not be done. We felt that the question "If the programmer writes deadlock, should she not get deadlock?" should be answered in the affirmative.

We became convinced of the contrary by several considerations. Firstly, there is the issue of figure 6.3, where deadlock may occur sometimes due to lack of system buffers. We find the comment in the MPI standard [10] that such code is correct but unsafe to be an unfortunate characterization justified only by insufficient technology. Of course, careful programming, with full consciousness of buffer requirements, possibly using non-blocking sends or receives (as SOS in fact uses) can avoid problems with figure 6.3. However, such programming is difficult and prone to error; the whole trend of high level programming languages is to hide such details where possible.

Having convinced ourselves that figure 6.3 should not require specific knowledge or consideration of underlying system buffers, consistency forces us to conclude that the same technology should not be arbitrarily blocked from also fixing the problem with figure 6.4.

The present system is an experimental system designed specifically for irregular problems and execution on heterogeneous networks of workstations. In its design context the system is capable of hiding communications costs; it however has not been optimized for more general use and is therefore more costly than standard communications if execution happens to be load balanced or if computational load is negligible and communications are fully exposed. We therefore regard this work, in its present state, as merely a 'proof of concept' for its deadlock avoidance properties.

We conclude that technology which uses variable read-write information from the program text to control dynamic communication scheduling can be used to hide system buffer details, and correct schedule deadlock where there are no true data dependences. In the example here presented, the use of such technology allows the programmer to concentrate on the logic of the data communications, and not have to worry about scheduling or buffer details. We believe other examples should present similar advantages. In particular, such technology should make programs with more complex communication and logic than the simple example presented here easier to write and more reliable, by hiding detail that is not necessarily relevant to the algorithm being implemented. We believe our present system shows promise which justifies continuing efforts.

References

[1] K. R. Apt, E. Olderog; *Verification of Sequential and Concurrent Programs, Second Edition.* Chapter 6, Springer, 1997

[2] A. Silberschatz, P. B. Galvin, *Operating System Concepts*, Chapter 7, Addison-Wesley 1995

[3] A. S. Tanenbaum, *Distributed Operating Systems*, Chapter 3, Prentice-Hall 1995

[4] A. S. Tanenbaum, *Computer Networks*, Chapter 4, Prentice-Hall 1981

[5] A. S. Tanenbaum, *Operating Systems - Second Edition*, Section 3.3, Prentice-Hall 1997

[6] E. G. Coffman, M. J. Elphick and A. Shoshani, *System Deadlocks*, Computing Surveys, vol. 3, pp 67-78, June 1971 (This paper derives the four conditions for deadlock)

[7] E. Gomez, *Single Program Task Parallelism*, http://people.cs.uchicago.edu/~ernesto/spmd.ps

[8] N. A. Lynch,*Distributed Algorithms*, Chapter 19,Morgan Kaufman 1995

[9] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, Second Edition*, Benjamin/Cummings 1994
[10] Message Passing Interface Forum; *MPI: A message Passing Interface Standard*. June 12, 1995, Version 1.1, http://www.mcs.anl.gov/mpi/
[11] D. B. Skillicorn, J. M. D. Hill, W. F. McColl, *Questions and answers about BSP,* technical report TR-15-96, Oxford University Computing Laboratory, August 1996
[12] I. Foster, *Task Parallelism and High Performance Languages*, IEEE Parallel and Distributed Technology, 1994.
[13] H. F. Jordan, *The Force*, in The Characteristics of Parallel Algorithms, L. H. Jamieson, D. B. Gannon and R. J. Douglass, Eds., Chap. 16, MIT Press (1987)
[14] C. A. R. Hoare, *Communicating Sequential Processes*, Communications of the ACM, Vol. 21, pp 666-677, 1978
[15] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985
[16] G. Jones, M. Goldsmith, *Programming in Occam 2*, Prentice-Hall 1988
[17] B. Bagheri, T. W. Clark, L. R. Scott, *Pfortran (a parallel extension of Fortran) reference manual*. Technical report 1991, revised March 1999. http://planguages.cs.uchicago.edu/
[18] J. Dongarra, A. Geist, R. Manchek, V. Sunderam, *Integrated PVM framework supports heterogeneous network computing*, Computers in Physics, 7(2), pp. 166-175, April 1993
[19] A. Gottlieb, *An Overview of the NYU Ultracomputer Project*, technical report TR1987-086-U100, NYU Computer Science, 1987
[20] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985
[21] R. Milner; *Communication and Concurrency,* Prentice-Hall International Series in Computer Science, 1989
[22] Ernesto Gomez and L. Ridgway Scott, *Overlapping and Short-cutting Techniques in Loosely Synchronous Irregular Problems*, in *Solving Irregularly Structured Problems in Parallel*, volume LNCS 1457, pp. 116-127, Springer, August 1998.
[23] Jeffrey D. Ullman, *Principles of Database Systems, Second Edition*, Computer Science Press 1982
[24] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley 1985
[25] P. Bala ,T. W. Clark; *Experiences with Co-Array Fortran and Pfortran as Tools for Parallelization of a Large-Scale Scientific Application*. Proceedings of the Third International Conference on Parallel Processing and Applied Mathematics, pp. 367-377. R. Wyrzykowski, B. Mochnacki, H. Piech, J. Szopa, Editors., Technical University of Czestochowa, 1999
[26] B. Bagheri, T. W. Clark and L. R. Scott; *Pfortran: A Parallel Dialect of Fortran*. Fortran Forum, ACM Press, 11, pp. 3-20, 1992.