SEND-RECEIVE SEMANTICS AND RUNTIME SYSTEM

(NOTE: a later version of this paper appeared in “Proceedings of Eighteenth International Conference on Systems Engineering”, August 1005, Las Vegas NV, under the title “Preventing Deadlock with Dynamic Message Scheduling”. This draft more clearly focuses on our concerns in CS 624).

Ernesto Gomez, Yasha Karant, Keith Schubert
Department of Computer Science, California State University San Bernardino

Abstract. We show that implementation concerns such as buffering and blocking produce semantic differences in send and receive instructions, affecting correctness of high level programs. We illustrate this through dependence graphs applied to shift algorithm on a ring. We show that scheduling constraints imposed by the implementation can conflict with data dependences in the program.

We show a runtime system that uses information on read or write access to particular variables to schedule messages. We demonstrate that this simplifies send and receive semantics and allows the programmer to focus only on data constraints. Experimental results are displayed.

Contact: egomez@cs5.csusb.edu. The implementation of message passing systems is generally held to be a matter of efficiency, that does not affect the meaning of programs. We argue that this is not the case, display a classification in terms of blocking properties and proofs of semantic implications for high level parallel programs. We propose that non-blocking send and receive lead to simpler semantics dependent only on data restrictions.

1. Previous work

Message passing is a major communication mode employed in parallel programming, both in theory and in practice. The following references, though not exhaustive, we believe to be representative.

Most practical approaches focus on blocking and buffering as questions of program efficiency ([1, 2, 3, 10]). More theoretical approaches such as Hoare’s [8] focus on a particular form of messaging, in this case synchronous messages. Milner [7] considers buffered communications by explicitly introducing buffers as communicating agents, but does not treat other types of blocking and non-blocking communications. Other work ([4, 6]) treats message sends and receives as understood things, and concerns itself with higher level algorithms that employ them. However they do not necessarily agree. Lynch, for example, treats send-receive systems as ordered queues, whereas Hagit and Attiya develop systems using messages which derive their properties from synchronous or asynchronous systems but study the properties of messages only in the context of simulating one type of system on another. One of this paper’s authors has done previous work [9] on non-blocking messages, but without fully considering their implications for program semantics.
The authors described a non-blocking communications system and its use for dynamic deadlock avoidance in [11]; we here consider semantic implications of such a system.

2. Definitions

We will restrict ourselves to considerations of message passing by point to point messages in the SPMD (Single Program Multiple Data) model. We specifically restrict ourselves to the mode of SPMD execution in which a fixed number of copies of the same program text run concurrently, and are allowed to execute different code within the same program as a consequence of data dependent logic. This mode of execution is sometimes also called procedural parallelism and is the execution model of, for example, MPI.

We here give some definitions and notation that we will use in this paper. Since we are interested in relating properties of communication to the program code, we define state an execution in terms of an abstract definition of the code. We choose the Control Flow Graph (CFG) as our representation.

Given a program with a CFG \((V, A, s)\) where \(V\) is the set of basic blocks which are vertices, \(A\) is the set of arcs, \(s \in V\) is the start block of the program.

We now define the state of the program to the basic block. A number of alternate definitions tied to the code (for example, individual statements or lines) would serve equally well; we choose the basic block as the largest set of statements that must be executed together.

**Definition 2.1.** Serial state: \(s_i = (x_i, M_i, D_i)\) where \(i\) denotes the step in program execution, \(x\) is the basic block executed at step \(i\), \(M\) is the data in memory upon entry to block \(x\) and \(D\) is the external date read by the process in block \(x\).

We can now define a parallel state as a collection of individual process states:

**Definition 2.2.** Parallel SPMD state:

\[ S_{G,J} = \{s_j^p \mid p \in G, j \in J, G \subseteq \Gamma \} \]

\(G\) is an ordered set of processes and \(J\) is a multi-index that lists the step number of the state of each process in \(G\) in the same order as \(G\). For an execution \(E_{\Gamma}\), if \(G = \Gamma\) then we call \(S_{\Gamma,J}\) a total state. Note that \(S_{G,J}\) is a collection of serial states such that each state is selected from the execution of a different serial process.

We now define a general parallel transition from one state to another:

**Definition 2.3.** A parallel transition \(S_{G,J} \rightarrow S_{H,K}\) exists between two states such that:

i. \(G \cap H \neq \emptyset\) (there are processes in common between both states).

ii. There is at least one serial state \(s_j^p \in S_{G,J}\) and at least one serial state \(s_k^p \in S_{H,K}\) such that there is a serial transition \(s_j^p \rightarrow s_k^p\) (at least one process takes a step).

iii. There are total states \(S_{G,J} \subseteq S_{\Gamma,L}\) and \(S_{H,K} \subseteq S_{\Gamma,M}\) such that \(\forall p \in \Gamma\), if \(s_j^p \in S_{\Gamma,L}\) and \(s_k^p \in S_{\Gamma,M}\) then \(0 \leq k - j \leq 1\) (states of individual processes that appear in both total states are separated by not more than one step).

We now define an SPMD parallel execution:
Definition 2.4. Parallel SPMD execution: Given a set $\Gamma$ of serial processes numbered 0 to $M$, and an external data set $\Delta = \cup^{p=0,M} D^p$ we have a (non-divergent) parallel execution:

$$E_{\Gamma, \Delta} = S_{\Gamma, 0} \to S_{\Gamma, 1} \to \ldots \to S_{\Gamma, \text{End}}$$

0 denotes the multi-index in which every process in $\Gamma$ is in its initial state, and End denotes the multi-index in which every process is in its final state.

Note that such a parallel execution does not always exist; our claim is that if it does exist, it can be represented in this form.

3. Clocks

We wish to consider scheduling constraints imposed by communications. To do this we need to define what clocks we are going to use.

3.0.1. System clocks: A first consideration is to use the system clock (or clocks, on a distributed system). Although this approach is of interest for real-time programming, it may not accurately reflect the view of time inside each process, and introduces extraneous considerations such as the accuracy and drift of clocks in distributed systems. Further, it would obscure the meaning of a synchronization in which two processes are said to execute code at the same time by making it appear impossible to have this unless the processes run on physically distinct processors.

The use of system or other clocks that measure actual time independent of process leads to the conclusion that the meaning of a program is changed depending on the hardware on which it is executed. While this may be true for programs that need to interact in real time to the outside world, it is not the problem we address here.

3.0.2. Process clocks:

Definition 3.1. A clock tied to the process can be any scalar quantity that increases monotonically as the process executes. Such a quantity could be, for example, the CPU time used, or the number of statements or basic blocks executed. We call such a quantity a process clock.

We will use the count of basic blocks executed (in the Control Flow Graph), since it gives us an immediate link between our clock and the code, is independent of processor speed, and corresponds to our definition of transition between states (definition 2.3).

Using process clocks it makes sense to speak, for example, of a barrier synchronization executed by all processes at the same execution step, and it does not matter whether the synchronization is in fact executed by all participants at the same wall clock time.

However, the implicit use of process clocks leads to the common consideration that, for example, a send at process $p$ matched by a receive at process $q$ means the same thing whether it is or is not buffered and independent of blocking considerations, since process $q$ sees the receive occurring at the same step in either case.

Another problem with process clocks is that some statements, such as barriers, can be rendered unintelligible - a barrier means that all participants wait for others to arrive, and are therefore present at the same time; but unless all processes have executed exactly the same number of statements, the process clocks at a barrier
will not match. We resolve this conflict by normalizing process clocks to the state (definition 2.2) in which the synchronization occurs.

**Definition 3.2.** A process clock is normalized to a state $S_{G,J}$ if it measures the difference between the clock value $\text{clock}_p$ at a process $p \in G$ and the clock value denoted by the index of $p$ in $J$. We refer to a state $S_{G,K-J}$, normalized to $S_{G,J}$, if it has the same process group $G$ and the index of each individual process state in $K$ represents the difference $K - J$.

We now say that a barrier takes place at a particular state (definition 2.2), and take the difference between clocks at each process and the corresponding entry in the multi-index of the barrier state. When this difference is zero at all processes, then all are present at the barrier and it can be released. Rather than talking about absolute time at each process as measured from the start state, we can talk about the difference between the clock at a particular process and its value at some particular state.

3.0.3. Message constraints: A message from a sender process $p$ to a receiver process $q$ is a synchronization between the two processes if it establishes a temporal order between events at each process. It may be argued that this order is always determined by the direction of data transfer - that is, that the clock at the sender is earlier than the clock at the receiver. While this is obviously true for a physical message, the actual relation between processes, and therefore the type of synchronization, is determined by the blocking properties of send and receive statements.

**Definition 3.3.** A blocking communication statement is one that must complete the specified action before the program can continue to the execution of the next statement. Specifically, a blocking send completes when a matching receive statement receives the transmitted message, and a receive statement completes when the memory into which it places the message contains all the transmitted data.

A non-blocking communication statement is a declaration that the process that executes the statement is ready to send or receive a message, but it returns immediately. Non-blocking communication may take place some arbitrary time after the execution of the communication statement, and may require additional code to verify that the message has actually been sent or received.

Buffered sends are usually considered non-blocking, in that execution can continue immediately after the message has been copied into a buffer. It is safe to immediately re-use the memory location named in the send statement, because the message has been copied to other memory. This is by no means the only way to achieve a non-blocking send, however. For example, the mpi_isend non-blocking send instruction does not use an additional buffer and returns immediately. It is then necessary to use an mpi_wait or mpi_test instruction to verify completion of the send.

A receive instruction usually is accomplished by a pair of instructions, such as mpi_irecv to declare that the receive can take place and specify the memory location that holds the received data, and again mpi_test or mpi_wait to verify that the data has been received and the memory location may be accessed.

Using Jordan's [1] classification, we have four possible combinations of blocking and non-blocking send and receive actions, given in table 1.
Let process $p$ be the sender and process $q$ be the receiver.

Case i in table 1 implies a state $S_{G,J}$ (definition 2.2) in which the process group is $G = \{p, q\}$ and the step multi-index is $\{i, j\}$, at which the communications occurs. Since, for case i both send and receive are blocking, neither process is allowed to advance beyond $S_{G,J}$ until communication has occurred. We can take the process clock values at \((p, q)\) to be the multi-index values \((\text{clock}_p, \text{clock}_q) = \{i, j\}\). Normalized to $S_{G,J}$ the clock values at which communication takes place are $\{\Delta_p, \Delta_q\} = \{0, 0\}$; this is a barrier synchronization, $\Delta_p = \Delta_q$.

In case ii, the sender is not blocked, but the receiver is. Therefore $p$ is allowed to advance while $q$ is blocked, and we have communication occurs when $\Delta_p \geq \Delta_q$. Similarly, in case iii of blocking send but non-blocking receive we have $\Delta_p \leq \Delta_q$.

In case iv, the communication itself does not place added constraints on the clocks, although considerations of correctness would force the receiver process $q$ to block and wait for communications to complete before reading the transmitted data (this also applies to case iii). On the sender side, if buffer space is limited, then the sender must be blocked if a buffer that has not been communicated needs to be re-used. Therefore all cases except i may require additional code to execute after the send and receive statements to block processes as needed.

Note that in we must always have $\{\Delta_p, \Delta_q\} \geq \{0, 0\}$ since communication cannot occur before the code that specifies it.

3.1. **Semantics of send-receive pairs.** In every case, the semantics of send and receive requires that data appear to have been transferred when those statements execute. That is, if there is a send($x$) statement in the code which transmits the value of a memory location $x$, then writes to $x$ that appear in the text after the send can not affect the value that is sent, whereas writes to $x$ textually before send must affect the value transferred. Similarly, if there is a receive($y$) statement, any reads of location $y$ textually after the receive must get the transmitted value, but reads before the receive statement must get the original value.

The above considerations apply to every case in table 1, which leads to the belief that semantics of send and receive are independent of implementation. We show that this is not the case through schedule dependence graphs.

**Definition 3.4.** A schedule dependence between two statements $s_1$ and $s_2$ is a synchronizing relation between clocks that measure the execution of each statement, which relation is enforced by code or program logic.

We distinguish the following possibilities:

**Theorem 3.5.** A schedule relation between normalized process clocks at two statements $s_1$ and $s_2$ must be one of the following:

- i. Equals: $\text{clock}_1 = \text{clock}_2$. Holds if both statements are constrained to execute synchronously. Applies if both statements are blocking.
ii. Unequal: clock\(_1 \neq \text{clock}_2\). Holds if either statement executes in a critical section.

iii. Greater: clock\(_1 > \text{clock}_2\). Holds if s2 is blocking but s1 is not; s2 may not execute until s1 does.

iv. Independent: Holds if neither s1 nor s2 has to wait for the other; applies if neither s1 nor s2 is blocking.

Proof. A normalized process clock (definitions 3.1, 3.2) is a scalar number, the only possible relations between a pair of scalars are =, >, $\neq$, combinations of these or no relation.

If both statements are blocking, then each has to wait for the other and they execute at the same normalized clock time.

If one statement is executing in a critical section, then the other cannot execute and the normalized clock times must be unequal.

Properties of blocking and non-blocking follow definition 3.3.

Take a state \(S_{(i, j)}\) such that s1 is at process 1 in the block executed at step i, s2 is at process 2 in the block executed at step j. Normalize clocks by \([-i, -j]\) giving a (normalized) state \(S_{(i, j)}\). If s2 is blocking, it must wait for s1 to execute. Since s1 is non-blocking then execution program execution and process clock can advance at process 1. Therefore normalized clocks are \(\Delta_1 > \Delta_2\).

If neither statement is blocking, the actual execution of both statements is determined by the runtime system and is not constrained by the actual communication statement code.

If s1 is blocking and s2, we invert indices in case iii so this does not add anything new; the given relations exhaust all possibilities.

A schedule dependence graph represents scheduling (but not data) constraints between communication statements. There is a schedule dependence between a pair of statements at the same process if the first statement must complete execution before the second statement can start. This holds if the first statement is blocking, but not if the first statement is non-blocking (def 3.3). Schedule dependence between two communications at different processes is described by theorem 3.5 holds.

We denote a dependence by an arrow directed from a non-blocking statement that must execute in order to allow a blocking statement to complete its execution and advance (as in case iii of theorem 3.5). Note that, since the non-blocking statement allows program execution to continue, the more advanced clock is at the base of the dependence arrow.

If two clocks are related by a greater than or equal operator: \(\Delta_1 \geq \Delta_2\) for s1, s2 we note from theorem 3.5 that, of the relations > and $\geq$, the > relation imposes fewer restrictions on execution since it applies when only one statement is blocking; therefore we take the dependence arrow from s1 to s2 as above.

We construct a schedule dependence graph as follows:

**Algorithm 3.6. Construction of a communication schedule dependence graph**

Input: a list of processes; ordered lists of communication statements at each process, annotated with destination, source, and blocking or non-blocking properties.

Output: A directed graph \(S = \{V, A\}\) in which the arcs A denote schedule dependences and indicate restrictions on the execution order of communication statements. \(S\) denotes a partial order in which statements may be executed.
Procedure:
1. Initially, nodes in \( V \) are individual communication statements, labeled with process numbers at which each statement is executed (If the same text is executed at multiple processes, it appears multiple times in \( V \)).
2. If \( l_1, l_2 \in V \) are statements that are executed by the same process, \( l_1 \) appears before \( l_2 \), and \( l_1 \) is blocking, place an arc \( l_1 \rightarrow l_2 \) in \( A \), indicating that \( l_1 \) must execute before \( l_2 \). Repeat until every consecutive pair of communication statements appearing in the list of statements for each process has been examined.
3. If a statement \( c_1 \) contains a send or receive which is matched by a send or receive in a statement \( c_2 \), and communications are synchronous (case i table 1) then merge nodes \( c_1 \) and \( c_2 \), indicating statements that must execute together. Repeat until no more nodes can be merged.
4. If a statement \( c_1 \) contains a send which is matched by receive in a statement \( c_2 \), sends are non-blocking and receives are blocking (case ii table 1) then place an arc \( c_1 \rightarrow c_2 \) in \( A \), since \( \text{clock}_1 > \text{clock}_2 \).
5. If a statement \( c_1 \) contains a send which is matched by receive in a statement \( c_2 \), sends are blocking and receives are non-blocking (case iii table 1) then place an arc \( c_2 \rightarrow c_1 \) in \( A \), since \( \text{clock}_2 > \text{clock}_1 \).

Repeat steps 4 and 5 until no new arcs are be added.

Given a schedule dependence graph as above, we can now state:

**Theorem 3.7.** A necessary condition for the execution of a set of communication statements is that their schedule graph be non-cyclic.

**Proof.** Assume false. A cycle in a dependence graph is one of the conditions of deadlock since every statement in the cycle requires that some other statement in the cycle be executed first and provides no starting point. \( \square \)

Note that, since the graph constructed by algorithm 3.6 takes only schedule into account, it is still possible that execution is prevented by a data dependence.

### 3.2. Examples
We now show through an example that the same syntax of send and receive statements leads to different schedule graphs.

Let processes be numbered \( 0 \ldots MAX \) for \( N = MAX + 1 \) processes. Let \( ME \) represent the identity of the local process. Processes are part of an SPMD execution in which each process has its own local memory and variable names are replicated at each process. Consider now the following pseudo-code.

**Example 3.8.** At every process do:

\[
\begin{align*}
&\text{send}(x, \text{to}((ME + 1) \mod N)); \\
&\text{receive}(y, \text{from}((ME - 1) \mod N));
\end{align*}
\]

These statements take processes to be arranged in a ring; the \( \mod \) function replaces \( MAX + 1 = N \) with 0 and \( 0 - 1 \) with \( MAX \). The program logic takes a local value stored in \( x \) at each process, and copies it to a memory location \( y \) at the next process in the ring. To do this with send and receive, each process sends to the next process in the ring and receives from the previous process (taking next and previous to refer to numeric order).

We note that there are no conflicts due to data dependences, since the source value and the received value are held in different memory locations at each process, and neither location is read or written between the communication statements.
Case 1: blocking send, blocking receive

Schedule diagrams 1, 2, 3 and 4 correspond to each of the four cases in table 1 as applied to three processes numbered 0 to 2 running the code of example 3.8. Although the data transfer appears does not change, we see that graphs 1 and 3 deadlock. We note also that graph 4 only shows communications schedule dependences; since all statements are non-blocking, there are none. However it may still be necessary to block processes in this case to satisfy data dependences; for example a process may need to be blocked after a non-blocking receive statement when the received data needs to be referenced.

Schedule diagrams 1, 2, 3 and 4 correspond to each of the four cases in table 1 as applied to three processes numbered 0 to 2 running the code of example 3.8. Although the data transfer appears does not change, we see that graphs 1 and 3 deadlock. We note also that graph 4 only shows communications schedule dependences; since all statements are non-blocking, there are none. However it may still be necessary to block processes in this case to satisfy data dependences; for example a process may need to be blocked after a non-blocking receive statement when the received data needs to be referenced.

We see in the example that the code fragment 3.8 has different execution dependent on the blocking constraints. We conclude that the semantics of the code in 3.8, and by extension the semantics of send and receive statements is different depending on the blocking behavior. It is our contention therefore that buffering, blocking and in general the underlying runtime system that supports messages is not simply a question of efficiency, but rather is one of the determining factors of message semantics.

3.3. Conflict between schedule and data dependences. A cycle in the schedule graph can often be resolved by a re-ordering of the communication statements. For example, the conflict in graph 1 can be resolved by switching the order of the send and receive statements in one process, yielding graph 5.
Case 2: Non–blocking send, blocking receive

Case 3: Blocking send, non–blocking receive

The conflict in graph 3 can be resolved by a somewhat counter-intuitive reversal of send and receive statements at all processes (graph 6). Note that in both cases, deadlock-free correct code under one set of blocking assumptions must be modified so that it is correct under a different set of assumptions.

So far we have limited ourselves to considering just schedule dependences, and figures 5 and 6 resolve these for code fragment 3.8. Suppose we modify this code by adding a reference to one of the memory locations involved in the communications:
Case 4: Non–blocking send, non–blocking receive

Example 3.9. At every process do:
\[\text{send}(x, \text{to}((ME + 1) \mod N));\]
\[x = \text{some.function}(y);\]
\[\text{receive}(y, \text{from}((ME - 1) \mod N));\]

Now the solution given in 5 which involves swapping the send and receive actions at one process will not work; since we are updating \(x\) between the communication statements, we would send the wrong value of \(x\) at one process. The solution in
Case 3: Blocking send, non-blocking receive
Send-receive order switched at all processes

6 would replicate the same error at all processes. What we have in code fragment 3.9 is a straightforward pair of data anti-dependences (read before write) which, however, only work as stated in case ii from table 1; and in case iv only if the runtime system is somehow aware of the anti-dependence and blocks processes as needed to preserve them.

4. A STANDARD SEND-RECEIVE SEMANTICS

A problem with the variety of semantic interpretations permitted by different blocking modes is that whether code is correct or not becomes uncertain. For example, is the code 3.8 correct? The MPI standard [2] considers this code to be correct but unsafe. Others would consider the code always incorrect because it may require unlimited buffer space. We feel that a standard semantics would be desirable; we would like to be able to talk about code as being correct or incorrect, not have to consider things like correct but unsafe, or incorrect but efficient.

Planguages [10] adopt synchronous semantics, allowing buffering or blocking to become an implementation efficiency issue issue. This however forces us to write code that implements the dependence graph 5, even where resources exist that would allow us to express greater concurrency as in graph 1.

Synchronous semantics is safe, can run with minimal resources and helps to assure freedom from deadlock [10]. Hoare [8] shows that synchronous communication is sufficiently expressive for any computation. However, we still would prefer to be able to take advantage of resources that allow greater concurrency if they exist, so we hesitate to standardize on synchronous messages semantics.

Although the authors have found non-blocking receives useful in writing real systems, we also find that thinking in terms of writing receives first and then sends is not particularly natural or intuitive. Again, we would not choose this as our semantic standard.
Non-blocking sends with blocking receives have the advantage of allowing the expression of more concurrency than synchronous send-receive pairs, and also coincide with our intuition about causality of messages. That is, they allow the sender process to perform a send action and continue, but require the receiver to wait for the message. However, the implementation of this mode requires either extra code (as in MPI asynchronous sends) or buffers. The buffered implementation appears simpler to use, but again leaves open the issue of safety. That is, without unlimited buffer space, running out of buffer or buffer overflow can occur in a program, and this would not allow us to guarantee correctness.

All of the blocking alternatives share the problem that the ability to execute correctly depends on schedule considerations imposed on top of data dependence considerations. We see in section 3.3 that code that appears to satisfy all data dependences can nevertheless be incorrect because of schedule problems.

5. NON-BLOCKING MESSAGING.

This brings us to the fourth alternative of non-blocking send and non-blocking receive. This has the desirable property that the only theoretical restrictions on the communications are that data dependences be preserved. The semantics of data transfer using this option are clear - all reads from, and writes to, variables involved in communication must be consistent with the send or receive having happened where the send or receive code appears in the program, similar to the semantics of put and get except that synchronization rather than being explicitly inserted would need to be deduced from the data dependences.

The difficulty is the implementation of this idealized form of non-blocking messaging. In practice most message systems require complex code to determine when communications have completed that in effect forces processes to block at the first point where a data conflict might arise (see examples of MPI asynchronous code in [2]).

One of the authors of this paper proposed an overlapping communications protocol in [9], and we further develop and describe its implementation in [11]. The protocol requires the addition two new instructions: LH(memory_location) and RH(memory_location), used to indicate variable use to the system. The LH call is entered immediately before the next appearance of a variable on the left-hand side of an expression (for write access). The RH call is inserted before the next use of a variable on the right-hand side of an expression for read access. These are calls to the runtime system designed for automatic insertion by a compiler, but it is straightforward to enter them by hand as needed.

A communication instruction does the following: First it computes the communication pattern required to carry out the specified instruction, as a series of point to point sends. Then it assigns a Finite State Machine (FSM) to control the communication (described in [9]). Particular sends and receives required at the each process are inserted in a list maintained by the runtime system, local to each process. If the particular communication is a send, the runtime sends a status message to the intended receiver indicating that the particular send is ready to execute. If the communication is a receive, the runtime system waits for a status message from the sender before proceeding.

The runtime system periodically examines all pending 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 posted later. Similarly, receives will execute as their matching sends become available, not necessarily in the order in which they are posted.

The LH and RH instructions provide the system with information about variable use. If there is no pending communication with respect to a particular variable, the instruction is ignored. Otherwise an LH or RH code is given to the FSM that regulates the pending communication. Depending on the current state of the machine, this may cause the process to block.

The runtime system, however, remains active and continuously checking for status update from other processes. In this way, as matching sends or receives become available, all pending communications that do not violate data constraints can complete. Therefore the system solves deadlocks that may arise from schedule conflicts that do not involve circular data dependence [11].

We term our system SOS, which stands for Streams, Overlapping and Shortcutting. Shortcutting and overlapping were described in [9]. Streams are a construct that supports groups of processes formed implicitly by program logic, and are described in a paper currently in preparation. In this paper we are concerned only with the semantic implications of overlapping, non-blocking communications.

The overlapping protocol, first described in [9], refers to overlapping of sections of code between communication statement and variable use at different processes. Conceptually, it treats a communication statement as a declaration that communication can take place. If the pending communications is a send, read access to the variable being transmitted is allowed, but the process is blocked before any write access. If the pending communication is a receive, then either read or write access to the variable to be received causes the process to be blocked. Overlapping in this context refers to the possibility that the interval between the communication statement and variable use at a sender process may coincide at least in part with the equivalent interval at a receiver process. If this is the case, communication can take place without needing to block either process. The complexity of the FSM described in [9] is mostly required to support collective communications in which a single variable may be involved in multiple sends and receives as part of a single instruction.

The system was designed originally to support irregular computation by performing this overlapping. We have since realized its extension to deadlock prevention [11] and its semantic implications.

6. Experiments

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 3.8 in pseudo-code given in figure SOS (Fortran) code is given in figure 7.

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 7. *Fortran* for ?? 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(iy,ix,MSIZE,0,0,MPI_REAL8,src2,dest2)

the following two calls ensure the program blocks until communication completes

call SOSdoRH(ix)
call SOSdoRH(iy)

print *,’at ’,me,x,’ done =’, y

The SOS code (figure 7) 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.

Figure 8 shows some sample runs of MPI code transformed as in graph 5. Figure 8 shows runs of the SOS code.

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.

The present system is only efficient in cases where asymmetric computation load leads to significant synchronization wait using MPI calls directly, but allows the SOS to minimize these through overlapping. Timing results for an irregular scientific computation, given in figure 10, show speedup by a factor of as much as three times compared to MPI.

Results in this section are quoted from [11].
7. Conclusions and Future Work

We have seen that implementation concerns such as buffering and blocking can produce semantic differences in send and receive instructions, because these considerations determine what kind of schedule can execute correctly and what schedules are risky or incorrect. We have also seen that scheduling constraints imposed by the implementation can conflict with data dependences in the program.

We propose a solution based on non-blocking sends and receives, which eliminates all scheduling constraints and leaves only restrictions due to data dependences. We propose that data dependence restrictions be enforced by the runtime system, in order to minimize the need to block processes and tolerate computational imbalance. We further propose two new instructions, suitable for insertion by hand or automatically by a compiler, to inform the runtime system of read or write access to variables and allow it to block processes to ensure correctness.

We have implemented a runtime system that serves as proof of concept for our proposal. Our present system is efficient for computationally intensive applications.
Figure 9. SOS runs

\[
\text{[ernesto@raven deadlock]\$ run 3 case4 2}
\]
\[
\begin{align*}
\text{at} & \quad x = 102. \\
\text{at} & \quad 1 -> 2 \\
\text{at} & \quad 0 -> 0 \\
\text{at} & \quad 102. \text{ done } y = 101. \text{ time} = 0.0113015 \\
\text{at} & \quad x = 101. \\
\text{at} & \quad 0 -> 1 \\
\text{at} & \quad 11 -> 2 \\
\text{at} & \quad 101. \text{ done } y = 100. \text{ time} = 0.01121925 \\
\text{SOS - message size} & \quad 2000 \text{ reps} = 4 \\
\text{at} & \quad x = 100. \\
\text{at} & \quad 0 -> 0 \\
\text{at} & \quad 0 -> 1 \\
\text{at} & \quad 100. \text{ done } y = 102. \text{ time} = 0.011025
\end{align*}
\]

\[
\text{[ernesto@raven deadlock]\$ run 3 case4 2}
\]
\[
\begin{align*}
\text{at} & \quad x = 101. \\
\text{at} & \quad 1 -> 1 \\
\text{at} & \quad 11 -> 2 \\
\text{at} & \quad 101. \text{ done } y = 100. \text{ time} = 1.10046675 \\
\text{at} & \quad x = 102. \\
\text{at} & \quad 21 -> 2 \\
\text{at} & \quad 22 -> 0 \\
\text{at} & \quad 102. \text{ done } y = 101. \text{ time} = 1.10594225 \\
\text{SOS - message size} & \quad 2000000 \text{ reps} = 4 \\
\text{at} & \quad x = 100. \\
\text{at} & \quad 02 -> 0 \\
\text{at} & \quad 00 -> 1 \\
\text{at} & \quad 100. \text{ done } y = 102. \text{ time} = 1.0583775
\end{align*}
\]

with irregular load balance. Additional work is required to reduce the overhead, particularly in the transmission of protocol information.

We conclude that a system of non-blocking sends and receives, coupled with restrictions imposed by data dependences, allows a maximal expression of parallelism and a clearer semantics for message passing; and that such a system can be implemented in practice.

References

Figure 10. Irregular application - time to completion as function of work performed