Interprocess Communication and Synchronization based on Message Passing

Henri E. Bal
Vrije Universiteit
Faculty of Sciences
Dept. of Computer Science

1. Introduction

A major issue which must be addressed in the design of a language for parallel programming is how the pieces of a program which are running in parallel on different processors are going to cooperate. This cooperation involves two types of interaction: communication and synchronization. For example, Process A may require some data X which is the result of some computation performed by Process B. There must be some way of getting X from B to A. In addition, if Process A comes to the point in its execution which requires the information X from Process B, but Process B has not yet communicated the information to A for whatever reason, A must be able to wait for it.

The basic mechanism for communication and synchronization is message passing. This paper gives an overview of interprocess communication and synchronization based on message passing. It first describes various language constructs that have been proposed for message passing. Next, it discusses the important issue of nondeterminism. Subsequently, it describes one example message-passing language (SR) and it discusses a parallel application in SR.

2. Message passing

Many factors come into play in the sending of a message: who sends it, what is sent, to whom is it sent, is it guaranteed to have arrived at the remote host, is it guaranteed to have been accepted by the remote process, is there a reply (or several replies), and what happens if something goes wrong. There are also many considerations involved in the receipt of a message: for which process or processes on the host, if any, is the message intended; is a process to be created to handle this message; if the message is intended for an existing process, what happens if the process is busy—is the message queued or discarded; and if a receiving process has more than one outstanding message waiting to be serviced, can it choose the order in which it services messages—be it FIFO, by sender, by some message type or identifier, by the contents of the message, or according to the receiving process's internal state.

We will begin with a general discussion of issues common to all message passing mechanisms. We then outline four specific message passing models: point-to-point messages, rendezvous, remote procedure call, and one-to-many messages.

General issues

The most elementary primitive for message-based interaction is the point-to-point message from one process (the sender) to another process (the receiver). Languages usually provide only *reliable* message passing. The language run time system (or the underlying operating system) automatically generates acknowledgement messages, transparent at the language level.

Most (but not all) message-based interactions involve two parties, one *sender* and one *receiver*. The sender initiates the interaction *explicitly*, for example by sending a message or invoking a remote procedure. On the other hand, the receipt of a message may either be *explicit* or *implicit*. With explicit receipt, the receiver is executing some sort of *accept* statement specifying which messages to accept and what actions to undertake when a message arrives. With implicit receipt, code is automatically invoked within the receiver. Implicit receipt is easier to use when it is not known in advance when a message will arrive or at which point in the program it should be handled. A good example is a message to request data from a remote process, which may arrive at any moment during the execution of the process. With explicit receipt, the programmer has to decide at which points during the execution such a message is accepted; with implicit receipt, the message will be handled whenever it arrives. Implicit receipt usually creates a new thread of control (a sub-process) within the receiving process; such threads run (pseudo-)concurrently and can share variables with each other. Whether the message is received implicitly or explicitly is transparent to the sender.

Explicit message receipt gives the receiver more control over the acceptance of messages. The receiver can be in many different *states*, and accept different types of messages in each state. More accurate control is possible if the accept statement allows messages to be accepted conditionally, depending on the arguments of the message. A file server, for example, may want to accept a request to open a file only if the file is not locked. In Concurrent C [1] this can be coded as

```
accept open(f) suchthat not_locked(f) {
    ...
    process open request
    ...
}
```

Some languages give the programmer control over the *order* of message acceptance. Usually, messages are accepted in FIFO order, but occasionally it is useful to change this order according to the type, sender, or contents of a message. For example, the file server may want to handle read requests for small amounts of data first:

```
accept read(f,offset,nr_bytes) by nr_bytes {
    ...
    process read request
    ...
}
```

The value given in the **by** expression determines the order of acceptance. If conditional or ordered acceptance is not supported by the language, an application needing these features will have to keep track of requests that have been accepted but not handled yet.

Another major issue in message passing is *naming* (or addressing) of the parties involved in an interaction: to whom does the sender wish to send its message, and, conversely, from whom does the receiver wish to accept a message? These parties can be named *directly* or *indirectly*. Direct naming is used to denote one specific process. The name can be the static name of the process or an expression evaluated at run time. A communication scheme based on direct naming is *symmetric* if both the sender and the receiver name each other. In an *asymmetric* scheme only the sender names the receiver. In this case, the receiver is willing to interact with any sender. Note that interactions using implicit receipt of messages are always asymmetric with respect to naming. Direct naming schemes, especially the symmetric ones, leave little room for expressing nondeterministic behavior. Languages using these schemes therefore have a separate mechanism for dealing with nondeterminism. Some message passing systems (e.g., MPI) support both symmetric and asymmetric naming.

Indirect naming involves an intermediate object, usually called a *port*, to which the sender directs its message and to which the receiver listens. In its simplest form a port is just a global name. More advanced schemes treat ports as values that can be passed around, for example, as part of a message. This option allows highly flexible communication patterns to be expressed.

In most systems, only a single process is allowed to receive messages from a given port. The ability to receive messages may sometimes be transferred from one process to another, but at any given time only one process can listen to the port. In contrast, a *mailbox* can have any number of receivers. If a message is sent to a mailbox, the system determines which processes currently are willing to receive the message, and then gives it to one of them. Clearly, implementing a mailbox is more complicated than implementing a port, since a mailbox can have multiple receivers located at different processors.

Mailing a letter to a post office box rather than a street address illustrates the difference between direct naming and the two forms of indirect naming (ports and mailboxes). A letter sent to a post office box can be collected by anyone who has a key to the box. People can be given access to the box by duplicating keys or by transferring existing keys (possibly through another P.O. box). A P.O. box with a single key is similar to a port; a box with multiple keys is similar to a mailbox. A street address, on the other hand, does not have this flexibility at all, so it is similar to direct naming.

Synchronous and asynchronous point-to-point messages

The major design issue for a point-to-point message passing system is the choice between *synchronous* and *asynchronous* message passing. With synchronous message passing, the sender is blocked until the receiver has accepted the message (explicitly or implicitly). Thus, the sender and receiver not only exchange data, but they also synchronize. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message. Conceptually, the sender continues immediately after sending the message. The implementation of the language may suspend the sender until the message has at least been copied for transmission.

In the asynchronous model, there are some semantic difficulties to be dealt with. As the sender S does not wait for the receiver R to be ready, there may be several *pending* messages sent by S, but not yet accepted by R. If the message passing primitive is *order preserving*, R will receive the messages in the order they were sent by S. The pending messages are *buffered* by the language run time system or the operating system. The problem of a possible buffer overflow can be dealt with in one of two ways. Message transfers can simply fail whenever there is no more buffer space. Unfortunately, this makes message passing less reliable. The second option is to use *flow control*, which means the sender is blocked until the receiver accepts some messages. This introduces a synchronization between the sender and receiver and may result in unexpected deadlocks.

In the synchronous model, however, there can be only one pending message from any process S to a process R. Usually, no ordering relation is assumed between messages sent by different processes. Buffering problems are less severe in the synchronous model, as a receiver need buffer at most one message from each sender, and additional flow control will not change the semantics of the primitive. On the other hand, the synchronous model also has its disadvantages. Most notably, synchronous message passing is less flexible than asynchronous message passing, because the sender always has to wait for the receiver to accept the message, even if the receiver does not have to return an answer [2].

Rendezvous

A point-to-point message establishes one-way communication between two processes. Many interactions between processes, however, are essentially two-way in nature. For example, in the client/server model the client requests a service from a server and then waits for the result returned by the server. This behavior can be simulated using two point-to-point messages, but a single higher level construct is easier to use and more efficient to implement. We will describe two such constructs, *rendezvous* and *remote procedure call*.

The rendezvous model is used in Ada, Concurrent C, SR and several other languages. It is based on three concepts: the entry declaration, the entry call, and the accept statement. The entry declaration and accept statement are part of the server (receiver) code, while the entry call is on the client (sender) side. An *entry declaration* syntactically looks like a procedure declaration. An entry has a name and zero or more formal parameters. An *entry call* is similar to a procedure call statement. It names the entry and the process containing the entry and it supplies actual parameters. An *accept statement* for the entry is used to explicitly receive an entry call. It may contain a list of statements, to be executed when the entry is called, as in the following accept statement for the entry `incr`:

```
accept incr(X: integer; Y: out integer) do
    Y := X + 1;
end;
```

An interaction (called a *rendezvous*) between two processes S and R takes place when S calls an entry of R, and R executes an **accept** statement for that entry. The interaction is fully synchronous, so the first process that is ready to interact waits for the other. When the two processes are synchronized, R executes the **do** part of the accept statement.

While executing these statements, R has access to the input parameters of the entry, supplied by S. R can assign values to the output parameters, which are passed back to S. After R has executed the **do** statements, S and R continue their execution in parallel. *R* may still continue working on the request of *S*, although *S* is no longer blocked.

Remote Procedure Call

Remote Procedure Call (RPC) is another primitive for two-way communication. It resembles a normal procedure call, except that the caller and receiver are different processes. When a process *S* calls a remote procedure *P* of a process *R*, the input parameters of *P*, supplied by *S*, are sent to *R*. When *R* receives the invocation request, it executes the code of *P* and then passes any output parameters back to *S*. During the execution of *P*, *S* is blocked. *S* is reactivated by the arrival of the output parameters. This is in contrast with the rendezvous mechanism, where the caller is unblocked as soon as the accept statement has been executed. Like rendezvous, RPC is a fully synchronous interaction. Acceptance of a remote call is usually *implicit* and creates a new thread of control within the receiver.

One-to-many message passing

Many networks used for distributed computing systems support a fast *broadcast* or *multicast* facility. A broadcast message is sent to *all* processors connected to the network. A multicast message is sent to a specific subset of these processors. On some networks (e.g., Ethernet), it takes about the same time to broadcast or multicast a message as to send it to one specific processor. Unfortunately, it is not guaranteed that messages are actually delivered at all destinations. The hardware attempts to send the messages to all processors involved, but messages may get lost because of communication errors or because some receiving processors are not ready to accept a message. Fortunately, software protocols can be used to make broadcasts reliable [3].

Even if broadcast and multicast are unreliable, they are useful for operating system kernels and language run time systems. For example, to locate a processor providing a specific service, an enquiry message may be broadcast. In this case, it is not necessary to receive an answer from every host: just finding one instance of the service is sufficient. Reliable broadcast and multicast are also useful for implementing distributed algorithms, so some languages provide a *one-to-many* message passing primitive.

One-to-many communication has several advantages over point-to-point message passing. If a process needs to send data to many other processes, a single multicast will be faster than many point-to-point messages. More importantly, a broadcast primitive may guarantee a certain *ordering* of messages that cannot be obtained easily with point-to-point messages [4]. A broadcast primitive that delivers messages at all destinations in the same order, for example, is highly useful for consistent updating of replicated data [5, 6].

3. Expressing and controlling nondeterminism

The interaction patterns between processes are not always deterministic, but sometimes depend on run time conditions. For this reason, models for expressing and controlling nondeterminism have been introduced. Some communication primitives that we have already seen are nondeterministic. A message received indirectly through a port or mailbox, for example, may have been sent by any process. Such primitives provide a way to *express* nondeterminism, but not to *control* it. Most programming languages use a separate construct for controlling nondeterminism. We will look at one such construct, the select statement, below.

A *select statement* consists of a list of guarded commands of the following form:

when *guard* **do** *statements*

The guard consists of a boolean expression and some sort of ‘‘communication request.’’ The boolean expression must be free of side effects, as it may be evaluated more than once during the course of the select statement’s execution. In CSP [7], for example, a guard may contain an explicit receipt of a message from a specific process P. Such a request may either *succeed* (if P has sent such a message), *fail* (if P has already terminated), or *suspend* (if P is still alive but has not sent the message yet). The guard itself can either succeed, fail, or suspend: the guard succeeds if the expression is ‘‘true’’ and the request succeeds; the guard fails if the boolean expression evaluates to ‘‘false’’ or if the communication request fails; or the guard suspends if the expression is ‘‘true’’ and the request suspends. The select statement as a whole blocks until either all of its guards fail or some guards succeed. In the first case, the entire select statement fails and has no effect. In the latter case, one succeeding guard is chosen nondeterministically and the corresponding statement part is executed.

The select statement can be used to wait nondeterministically for specific messages from specific processes. The select statement contains a list of input requests and allows individual requests to be enabled or disabled dynamically. As an example, consider a *bounded buffer* process, which interacts with consumers and producers. A buffer process may accept a request from a producer process to store an item in the buffer whenever the buffer is not full; it may accept a request from a consumer process to add an item whenever the buffer is not empty.

```
SELECT
  WHEN NOT full(buffer) AND ACCEPT StoreItem(x: integer) DO
        store x in buffer;
OR
  WHEN NOT empty(buffer) AND ACCEPT FetchItem(x: OUT integer) DO
      x := first item in buffer
      remove x from buffer
END SELECT
```

Communication takes place as soon as either (1) the buffer is not full and the producer sends a message `StoreItem`, or (2) the buffer is not empty and the consumer sends a message `FetchItem`. In the latter case the buffer process responds by sending the item to the consumer.

In most languages (e.g., CSP and Ada), the select statement is asymmetric in that the guard can only contain a receive (accept) statement but not a send statement. Thus, a process P can only wait to receive messages nondeterministically; it cannot wait nondeterministically until some other process is ready to accept a message from P [8]. Output guards are excluded from most languages, because they usually complicate the implementation. Languages that do allow output guards include Joyce [9] and Pascal-m [10].

Select statements can also be used for controlling nondeterminism other than communication. Some languages allow a guard to contain a *timeout* instead of a communication request. A guard containing a timeout of T seconds succeeds if no other guard succeeds within T seconds. This mechanism sets a limit on the time a process wants to wait for a message. Another use of select statements is to control *termination* of processes. In Concurrent C, a guard may consist of the keyword **terminate**. A process that executes a select statement containing a **terminate** guard is willing to terminate if all other guards fail or suspend. If all processes are willing to terminate, the entire Concurrent C program terminates. Ada uses a similar mechanism to terminate parts of a program. Roughly, if all processes created by the same process are willing to terminate and the process that created them has finished the execution of its statements, all these processes are terminated. This mechanism presumes hierarchical processes.

A final note: select statements in most languages are *unfair*. In the CSP model, for example, if several guards are successful, one of them is selected nondeterministically. No assumptions can be made about which guard is selected. Repeated execution of the select statement may select the same guard over and over again, even if there are other successful guards. An *implementation* may introduce a degree of fairness, by assuring that a successful guard will be selected within a finite number of iterations, or by giving guards equal chances. On the other hand, an implementation may evaluate the guards sequentially and always choose the first one yielding ‘‘true.’’ The semantics of select statements do not guarantee any degree of fairness, so programmers cannot rely on it.

4. An example message passing language: SR

SR (Synchronizing Resources) is a language for writing parallel and distributed programs developed by Greg Andrews and Ron Olsson and their colleagues at the University of Arizona and the University of California at Davis [11]. The design of SR has been guided by three goals: expressiveness, ease of use, and efficiency. The first goal has led to the inclusion of a wide variety of message passing constructs for interprocess communication (IPC). Yet, the designers have tried to keep the language simple and easy to use, by minimizing the number of underlying concepts and by integrating them smoothly with the sequential parts. Finally, they have tried to make each communication primitive as efficient as possible. Below, we give a brief description of SR.

One of the main ideas behind SR is that, of the many existing IPC mechanisms, no single primitive will be ideally suited for all applications. SR, therefore, supports a variety of mechanisms, including shared variables (for processes on the same node), asynchronous message passing, rendezvous, remote procedure call, and multicast. In addition, it provides a powerful select statement for expressing nondeterministic behavior.

In this section, we will describe the distributed programming constructs supported by SR. Of course, some constructs can be simulated through others. For example, asynchronous message passing can be used to simulate RPC. Still, a language designer may prefer to support both primitives and let the programmer use the one most natural to the application.

A distributed SR program consists of multiple modules called *resources*. A resource definition consists of two parts: a specification part (containing the interface to the resource) and a body (containing the implementation). Note that this is similar to a package in Ada. Each resource is run on one physical node (either a single processor or a shared-memory multiprocessor). A resource contains one or more *processes*, which may communicate through shared variables. Such processes can synchronize through semaphores. Both resources and processes can be created and destroyed dynamically.

Any two processes, whether in the same resource or not, can communicate through *operations*. An operation unifies the concepts of a local procedure, remote procedure, and process. An operation must be declared before it may be used. Such a declaration has the following form:

op name(parameters) [**returns** result]

If other resources are to use the operation, the declaration should appear in the specification part of the resource; else, the declaration may be local to the body. In any case, the operation is to be implemented in the body, in either one of two ways:

1. As a procedure; whenever the operation is invoked, a new process will be created within the resource to service it.
2. In one or more **in** statements; the operation will be serviced by an already existing process, when such a process executes the **in** statement.

These two cases correspond to *implicit* and *explicit* receipt of messages. The **in** statement is used to receive (accept) messages explicitly. It is a very general statement with the same functionality as the **select** statement described in Section 3.

SR allows operations to be accepted *conditionally* based on the operation's parameters. For example, in

in PivotRow(iter, row) **st** iter = CurrentIteration -> ... **ni**

the operation will only be accepted if its first parameter equals a certain value. (The **st** stands for *such that*.) SR also supports ordered message receipt based on the parameters of the operations. Each alternative in an **in** statement can be given a scheduling expression. If multiple invocations can be serviced, the one that minimizes the scheduling expression is selected first. For example, in

in NewMinimum(value) **by** value -> ....

the calls to *NewMinimum* that contain the lowest value for the parameter are serviced first.

Besides these two different ways of *receiving* an operation, there also are two ways for *invoking* operations:

1. Asynchronously, using a **send**; the sender will proceed as soon as the operation has been delivered to the receiver's node.
2. Synchronously, using a **call**; the sender will block until the operation has been serviced and any result values have been returned.

Either way, the sender directly specifies the receiver, for example:

```
send P.NewMinimum(12);   # send message to process P
```

So, SR does not support ports or mailboxes.

By combining the ways of servicing and invoking operations, four primitives can be used:

1. Process creation (**send** + implicit receipt)
2. Asynchronous message passing (**send** + explicit receipt)
3. Remote procedure call (**call** + implicit receipt)
4. Rendezvous (**call** + explicit receipt)

In addition, **call**ing an operation implemented as a procedure in the same resource is equivalent to a local procedure call. So, we have local procedure calls, process creation, and three IPC mechanisms.

In addition to the IPC mechanisms listed above, SR supports a **co** statement for concurrent invocations. For example, the statement

```
co (for i := 1 to nslaves) send slaves[i].PivotRow(iter, row) oc
```

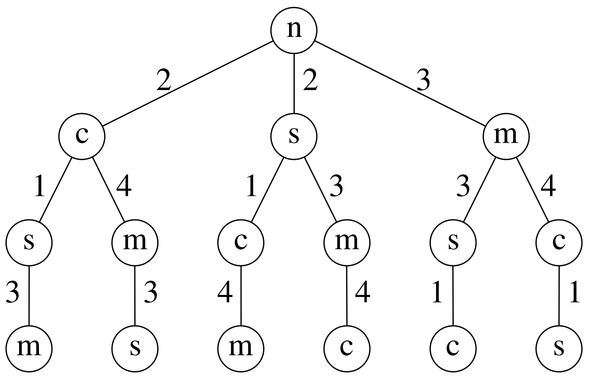
sends the operation *PivotRow* to slave resources, whose identities are stored in an array. The **co** statement basically supports multicast communication. It can only be used in combination with **call** and **send** statements.

5. An example application in SR

We will now discuss an example parallel application written in SR. The application is the Traveling Salesman Problem (TSP), which is to compute the shortest route for a salesman that visits each of a set of cities in his territory exactly once.

5.1. The Traveling Salesman Problem

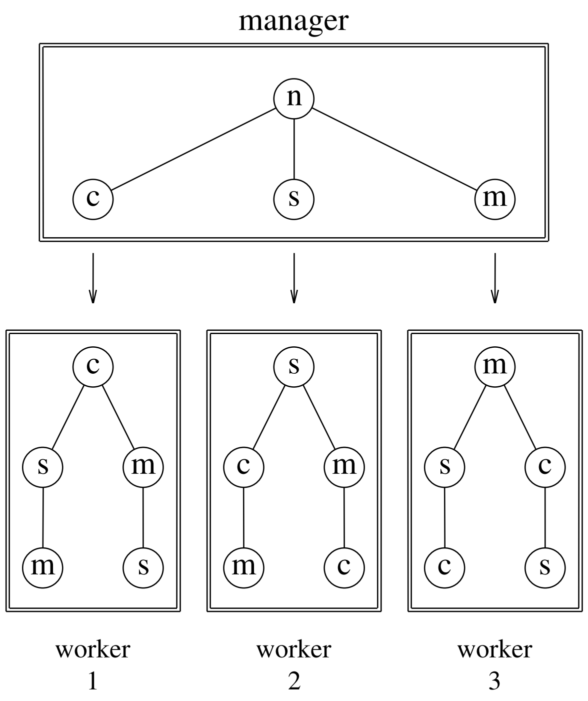
The SR program is based on a parallel *branch-and-bound* algorithm. Abstractly, branch-and-bound uses a *tree* to structure the space of possible solutions. A *branching rule* tells how the tree is built. For the TSP, a node of the tree represents a partial tour. Each node has a branch for every city that is not on this partial tour. The figure below shows a tree for a 4-city problem. Note that a leaf represents a full tour (a solution). For example, the leftmost branch represents the tour New York - Chicago - St. Louis - Miami.



Search tree for 4-city TSP (New York, Chicago, St. Louis, Miami).

A *bounding rule* avoids searching the whole tree. For TSP, the bounding rule is simple. If the length of a partial tour exceeds the length of any already known solution, the partial tour will never lead to a solution better than what is already known. In the figure above, for example, the leftmost full route has length 6, so the partial route starting with New York - Miami - Chicago (of length 7) cannot lead to an optimal solution and thus can be pruned. A simple heuristic is used to make pruning as effective as possible. The tree is traversed in *nearest-city-first* order. Given an initial route T_1, T_2, .. , T_i, the algorithm first considers the city closest to T_i that is not on the initial route.

Parallelism in a branch-and-bound algorithm is obtained by searching parts of the tree in parallel. If enough processors were available, a new processor could be allocated to every node of the tree. Every processor would select the best partial path from its children and report the result back to its parent. If there are `N` cities, this approach would require O(`N`!) processors. More realistically, the work has to be divided among the available processors. For this purpose, we use *replicated worker style* parallelism [12, 13]. In this approach, a *manager* process dynamically generates work that is to be executed by one or more *worker* processes, each executing on a separate processor. For TSP, the manager generates partial routes (up to a certain depth) for the salesman and sends them out to the workers for further evaluation. The figure below shows an example of this.



Each worker keeps track of the best solution found so far and informs all the others whenever it finds a shorter route. This value is used as a ‘‘bound’’ for pruning parts of the search tree.

5.2. Implementing TSP in SR

Let us now consider how the TSP algorithm described above can be implemented in SR. Two types of resources are used, one for the manager and one for the workers. Multiple instances of the worker resource are created, one per available machine. Since the manager and workers are on different machines, they communicate through message passing (i.e., SR’s operations). The TSP program uses communication for two reasons: to distribute work and to maintain the global bound. We discuss these two cases below.

First, the manager needs to distribute work to the workers. When the manager has computed a new job (to be executed in parallel), it should send it to one of the workers. Since it is not known in advance which worker process will accept the job, it is inconvenient to use direct message passing. What is actually required here is communication through a mailbox, which allows any process to receive a message. Despite its large number of features, SR does not directly support message passing through a mailbox. The receiver of a message is fully determined when the message is *sent*. With a mailbox, the destination process is not determined until the message is *accepted* (serviced).

A solution to this problem is as follows. When the manager wants to send a message, it blocks until a receiver asks for a message. In this case, the receiver can directly fetch a message from the sender. As a disadvantage, the manager will always have only one job available. If multiple workers ask for a job simultaneously, they will experience some delay. If this becomes a problem, a buffer process could be inserted between the manager and the workers.

The second form of communication used in the TSP program concerns the global bound. The branch-and-bound algorithm requires a global variable containing the length of the current best solution. This variable is used for pruning partial solutions whose initial paths are already longer than the current best full route. To obtain good performance, it is very important that all processors immediately get informed when a new (better) value for the bound has been found. The bound is used to prune parts of the search tree, so the lower this value the more effective pruning is. If a new value for the bound is not made known to all processors immediately, the parallel program will search too many nodes, resulting in *search overhead*.

The global bound cannot be put in a shared variable, since the manager and workers are on different machines. One solution is to store the variable on one processor (resource) and let other processors access it through remote operations. For TSP, however, a much more efficient solution is possible. The bound is usually changed (improved) only a few times, but may be used millions of times by each processor, so its read/write ratio is very high. Therefore, the variable can be implemented efficiently by *replicating* it in the local memories of the processors. Each processor can directly read the variable. Physical communication only occurs when the variable is written, which happens infrequently.

In SR, we have implemented this global bound as follows. Each worker keeps a copy of the global bound. Whenever a worker finds a shorter route for the salesman, it sends it to a *BoundManager* process within the manager resource. This process multicasts the new solution to all workers, as discussed below. The message to the BoundManager is sent asynchronously, because the worker does not have to wait until the new value is forwarded to the other processes. So, the worker continues immediately, without wasting any time. This is a good example of the usefulness of asynchronous message passing.

Clearly, the receivers (the other workers) do not know in advance when the update messages will arrive. Therefore, it would be inconvenient to use explicit message receipt. With implicit receipt, however, this is no problem at all, since a new process will automatically be created to handle the incoming message. The code for handling the update message from the BoundManager is shown below.

```
proc UpdateMinimum(Value)          # update local copy of the bound
        P(mutex)                   # lock the copy, using a semaphore
        if value <= minimum ->     # ignore higher values
                minimum := value   # "minimum" is the local copy
        fi
        V(mutex)                   # unlock copy
end UpdateMinimum
```

The local copy of the bound is protected by a semaphore, to prevent the main computation from reading it while it is being written. This example clearly illustrates the advantages of implicit message receipt. If update messages were to be implemented with explicit receipt, an extra process would be needed for handling these messages. So, the implementation described above is simpler.

Now let us also look at the implementation of the BoundManager itself. As described above, this process receives changes to the global bound and forwards them to the workers. Note that the BoundManager is only interested in *lower* values of the bound than what it currently has, since the objective is to find the shortest path. If it receives a higher value, it simply throws away the message.

Now, assume that the current value of the bound is M_0 and that two workers simultaneously discover better values, say M_1 and M_2 (so $M_1 < M_0$ and $M_2 < M_0$). Without loss of generality, also assume that $M_2 < M_1$. If the BoundManager first accepts M_1 and then M_2, it will have to multicast both values. It first multicasts M_1 and makes it the new minimum. Next, when the M_2 message arrives, it also has to be multicast, because it is better than the current minimum (M_1). On the other hand, if it accepts M_2 first, it can ignore the M_1 message, so it needs to do only one multicast. In conclusion, it is most efficient to accept the messages in increasing order of their value. The code for the BoundManager is shown below.

```
process BoundManager
    do true ->
        in NewMinimum(value) by value -> # order messages
            if value < minimum ->           # is it really better?
                minimum := value            # update primary copy
                co (i := 1 to ncpus)        # and secondary copies
                    call workers[i].UpdateMinimum(value)
                oc
            fi      # else ignore it
        ni
    od
end BoundManager
```

The **in** statement orders the messages by increasing value. We have done some experiments with this program, which show that the optimization indeed saves several multicasts, so it is worth while.

In summary, this example demonstrates several different aspects of message passing, including asynchronous sends, multicast, implicit and explicit receipt, and ordered receipt.

REFERENCES

1. N. Gehani and W.D. Roome, *The Concurrent C Programming Language,* Silicon Press, Summit, N.J. (1989).
2. N.H. Gehani, *Message Passing: Synchronous versus Asynchronous,* AT&T Bell Laboratories, Murray Hill, N.J. (1987).
3. M.F. Kaashoek, ‘‘Group Communication in Distributed Computer Systems,’’ Ph.D. Thesis, Vrije Universiteit, Amsterdam (1992).
4. K.P. Birman and T.A. Joseph, ‘‘Reliable Communication in the Presence of Failures,’’ *ACM Trans. Comp. Syst.* **5**(1), pp. 47-76 (Feb. 1987).
5. T.A. Joseph and K.P. Birman, ‘‘Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems,’’ *ACM Trans. Comp. Syst.* **4**(1), pp. 54-70

(Feb. 1986).

6. H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M.F. Kaashoek, ‘‘Performance Evaluation of the Orca Shared Object System,’’ *ACM Trans. Comp. Syst.* **16**(1), pp. 1-40 (Feb. 1998).

7. C.A.R. Hoare, ‘‘Communicating Sequential Processes,’’ *Commun. ACM* **21**(8), pp. 666-677 (Aug. 1978).

8. A.J. Bernstein, ‘‘Output Guards and Nondeterminism in ‘‘Communicating Sequential Processes’’,’’ *ACM Trans. Program. Lang. Syst.* **2**(2), pp. 234-238 (April 1980).

9. P. Brinch Hansen, ‘‘Joyce - A Programming Language for Distributed Systems,’’ *Softw. Prac. Exper.* **17**(1), pp. 29-50 (Jan. 1987).

10. S. Abramsky and R. Bornat, ‘‘Pascal-m: a Language for Loosely Coupled Distributed Systems,’’ pp. 163-189 in *Distributed Computing Systems*, ed. Y. Paker and J.-P. Verjus, Academic Press, London (1983).

11. G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, ‘‘An Overview of the SR Language and Implementation,’’ *ACM Trans. Program. Lang. Syst.* **10**(1), pp. 51-86 (Jan. 1988).

12. G.R. Andrews, ‘‘Paradigms for Process Interaction in Distributed Programs,’’ *ACM Computing Surveys* **23**(1), pp. 49-90 (March 1991).

13. N. Carriero, D. Gelernter, and J. Leichter, ‘‘Distributed Data Structures in Linda,’’ *Proc. 13th ACM Symp. Princ. Progr. Lang.*, St. Petersburg, Fl., pp. 236-242 (Jan. 1986).