Parallel Programming in SR

Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit, Amsterdam

Abstract
SR is a language for writing distributed programs. It supports many forms of interprocess communication to obtain high expressiveness. It has also been designed to be simple to use. We will evaluate the design of SR from the perspective of parallel programming. We study several programs for parallel applications, which we have implemented in SR, and we make some observations about the expressiveness and ease of use of SR.

Our results indicate that nearly all facilities provided by SR are useful for parallel programming. The language lacks message passing through mailboxes, message forwarding, and globally shared variables.

SR also is fairly simple to understand. A strong point in the design is the orthogonality of the message sending and receiving constructs. Some points of criticism concern the multicast mechanism and the type-insecurity of the language.

1. Introduction
SR (Synchronizing Resources) is a language for writing distributed programs, developed by Greg Andrews and Ron Olsson and their colleagues at the University of Arizona and the University of California at Davis. 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.

The SR language has been around for ten years now [1,2] and has been subject to several changes [4,17,6]. It probably is one of the most important and elaborate languages based on message passing. We therefor think it is important to evaluate the language design and see whether it meets the demands of applications programmers.

In this paper, we will study the language design from the perspective of parallel programming. SR is intended for a broad scope of applications, such as distributed operating systems, distributed servers (e.g., file servers), and parallel applications that run on distributed hardware. We have used SR only for implementing parallel programs, not for writing systems programs. Despite this limitation, however, we have been able to make important observations about the language design, which we will describe here. In particular, we will examine to what extent the goals of expressiveness and ease of use have been met. The third goal, efficiency, is far less subjective and has already been analyzed elsewhere [9].

The lessons learned from this evaluation may be of general interest to designers and users of parallel and distributed programming languages based on message passing. Our study gives insight in the usefulness and ease of use of several language features related to message passing.

The language used for this paper is referred to as SR Version 1.1 [7]. The SR designers are currently working on Version 2, in which many of the problems described here will be solved. SR has been used for several distributed applications, the largest of which is the file system of the Saguaro distributed operating system [5]. This experiment has also stimulated some changes to an earlier version of SR [18].

The outline of the rest of the paper is as follows. We first describe the applications we implemented in SR. Next, we present an overview of SR's parallel and distributed programming constructs. Subsequently, we discuss which of the features supported by SR are useful for parallel applications, and which of the missing features would have been useful. Next, we discuss issues related to ease of use, such as the number of concepts in the language, their integration, and their semantics. Finally, we present some conclusions.
2. The applications

The discussion in this paper is based on several programs we have implemented in SR. These programs are described briefly below. Nearly all of them implement parallel applications and are to be executed on a distributed (non shared-memory) system.

Matrix multiplication (MM).

The result matrix is partitioned evenly among the available processors. Each processor is assigned a fixed number of rows of this matrix. All processors execute independently from each other, so this program explores “trivial” parallelism.

All-Pairs Shortest Paths Problem (ASP).

Compute the shortest path between all pairs of nodes in a given graph, using a parallel iterative algorithm similar to the one described by Jenq and Sahni [16]. Each processor is assigned a fixed portion of the distance matrix to be computed. Unlike in MM, processors communicate heavily.

Alpha-beta search (AB).

Evaluate a game tree using the parallel alpha-beta search algorithm of [11]. Unlike in MM and ASP, the partitioning of work is done dynamically, using a replicated worker style program [8]. In such a program, a manager process dynamically generates work that is to be executed by one or more worker processes, each executing on a separate processor. For AB, the manager builds the top part of the search tree, up to a certain depth. The subtrees are further evaluated by worker processes. The results are sent back to the manager, which updates the alpha and beta values in the top part of the tree.

The Traveling Salesman Problem (TSP).

Compute the shortest route for a salesman that visits each of a set of cities in his territory exactly once. The program is based on the parallel branch-and-bound algorithm described in [11]. Again, the replicated worker style is used. The manager generates partial routes for the salesman and sends them out to the workers, which further expand the route. 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.

Sorting.

Sort a sequence of items, using the algorithm of Horiguchi and Shigei [15]. Each processor receives an equal portion of the items and pre-sorts them. Next, processors repeatedly exchange items with their left and right neighbors and merge them with their own items, until the entire sequence has been sorted.

Discrete event simulation.

A package for simulating queuing networks. This program does not try to achieve speedups and is usually executed on only a single processor. Using a language with parallel constructs (like SR), however, simplifies the implementation of the package.

The first four programs are based on earlier versions written in Orca [11], a language for implementing parallel applications on distributed systems. The discrete event simulation package is based on a Concurrent C program by Gehani and Roome [14].

Each program has been tested on a Sequent Symmetry with 6 processors. Although the Sequent is a shared-memory machine, the programs are designed for distributed systems and do not rely on shared memory. Moreover, the SR implementation on the Sequent does not use the shared memory as such. Instead, it uses the communication primitives of the operating system (Unix®), which simulate message passing through shared memory. The programs therefore could be run on a distributed system without any modifications.

3. An overview of SR

One of the main ideas behind SR is that, of the many existing IPC mechanisms [3,10], 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 parallel and distributed programming constructs supported by SR. Also, we will examine which other constructs are not supported by the language. 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 resources. A resource is a module run on one physical node (either a single processor or a shared-memory multiprocessor). The definition of a resource consists of two parts: a specification part and a body. The specification part describes the interface to users of the resource; the body gives the implementation.

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:

\[
\text{op name(parameters) [returns result]}
\]

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 is similar to the select statement in Ada®. It allows a process to wait for one of a number of operation invocations. Unlike Ada, however, SR allows operations to be accepted conditionally based on the operation's parameters. For example, in

\[
\text{in PivotRow(iter, row) st iter = CurrentIteration} \rightarrow ...
\]

the operation will only be accepted if its first parameter equals a certain value. As another difference with Ada, the alternatives of an in statement can be given scheduling expressions, possibly also based on the parameters of the operations. If multiple invocations can be serviced, the one that minimizes the scheduling expression is selected 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 at 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.

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, calling 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

\[
\text{co (for i := 1 to nslaves)}
\quad \text{send slaves[i].PivotRow(iter, row)}
\quad \text{oc}
\]

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.

As should be clear now, SR has many different IPC mechanisms. Still, there is always room for more, so we will now consider what SR does not support. This is not to say these are omissions in the language design; we merely wish to make clear which other options exist. Their usefulness will be discussed in a later section.

A restriction common to all SR message passing constructs is the way receivers are determined. An operation is sent to one specific resource, which is fixed at the time the operation is invoked. The sender does not always have to know the identity of the receiver, since capabilities (pointers) for operations can be used. Even with capabilities, however, the receiving resource is fully determined when the operation is invoked. The operation will always be serviced by a process local to that resource. In contrast, other languages support sending messages to a mailbox, where they can be picked up by any process that has access to the mailbox.

SR does not support forwarding of messages. The reply of a rendezvous must therefore always be sent by the process that accepts (ins) the operation. If the reply is to be delegated to another process, separate point-to-point messages must be used instead of the 2-way rendezvous.

Finally, SR restricts the sharing of variables to processes running in the same resource. Processes executing on the same machine (or the same shared-memory multiprocessor) can be put together in a single resource and can communicate through shared variables. Since these processes share a common memory, the implementation is easy. Processes on different resources do not have a common memory and can only communicate through message passing. Other languages (e.g., Linda and Orca) do not impose such restrictions on shared
data [lo]. The run time systems of these languages take care of the physical distribution of shared data and typically replicate or migrate such data.

4. The usefulness of the SR distributed programming constructs

Potentially, there are two pitfalls in designing a programming language: including useless features and excluding useful features. This section discusses the usefulness of SR's constructs. The next section describes which of the facilities that are not supported would have been useful.

Of course, even if a given feature is useful, it does not necessarily mean it should be included in a language. Adding more features to a language clearly is not for free. It takes more time to learn the language; programs may become harder to understand; and the implementation of the language will be more difficult. However, we will not discuss the issue of language complexity until a later section. First, we will study SR's language features objectively, without worrying too much about their impact on the size of the language.

Although our programming experience with SR is limited and strongly oriented towards one application area (parallel applications), we found many uses for most of SR's constructs. Even facilities intended primarily for other types of applications turn out to be useful for parallel programming. We give many examples below.

4.1. Explicit and implicit message receipt

Implicit receipt is most convenient if messages may arrive at arbitrary points of time during a computation. As an example, consider the TSP program. This program consists of a manager and one or more workers. Each worker—implemented as a resource and executing on a separate processor—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. Clearly, the receivers do not know in advance when such update messages arrive. With implicit receipt, 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)
   P(mutex)  # lock copy
   if value <= minimum ->
      minimum := value
   fi
   V(mutex)  # unlock copy
end UpdateMinimum
```

The local copy of the bound (called minimum) is protected by a semaphore, to prevent the main computation from reading it while it is being written. (As we will discuss later, it is not clear whether SR actually requires explicit locking of shared integers.) 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. A disadvantage of implicit receipt is the overhead of creating a new local process for each message. On a SUN-3, this overhead is about 0.5 msec [9].

Explicit receipt also has many applications. Sometimes, messages should be received in a certain order. In the sorting program, for example, each resource first needs to know the identities of its left and right neighbors, before it can start exchanging items. These identities are sent by the main program, after it has created all sorter processes. Each sorter process first waits for this message to arrive, by doing an explicit receipt:

```
receive neighbors(LeftNeighbor, RightNeighbor)
```

(A receive is a simplified form of ia.) With implicit receipt, semaphores would have to be used to get the synchronization right.

With explicit receipt, it is also simple to make all operations indivisible. Many server-type applications just want to handle all incoming messages one at a time. Such a single-threaded server loop is easy to program:

```
do true ->
in op1( .. ) -> ..
[ ] op2( .. ) -> ..
...
[ ] opn( .. ) -> ..
ni
```

This pattern occurs frequently in parallel programs. Again, with implicit receipt the operations would have to use explicit condition synchronization to serialize the operations.

Other advantages of explicit receipt are conditional and ordered message acceptance based on the parameters of the message. These features ease the implementation of some servers (e.g., disk schedulers) [13], and we found them useful for parallel applications as well.

Conditional acceptance is used to great advantage in the ASP program. This program performs a fixed number of iterations. Before each iteration, one processor multicasts the pivot row for this iteration to all the others. A processor cannot start working on an iteration until it has received the pivot row for that iteration. Apart from this restriction, all processors execute asynchronously and
may be working on different iterations.

The process that multicasts the pivot row is different for each iteration. Moreover, multicast in SR is not indivisible; if two processors simultaneously multicast two messages, these messages need not arrive in the same order everywhere. As a result, the pivot rows may arrive out of order. Still, each processor should accept the pivot rows in the right order. With conditional message acceptance, this is easy to implement. Each processor executes the following code:

```bash
fa k := 1 to N -> # perform N iterations
  if I have the pivot row for iteration k ->
    pivot := my pivot row
    # multicast the pivot row
    co (for all other resources s)
    send s.PivotRow(k, pivot)
  fi
  [ ] else ->
    # wait for the pivot row
    in PivotRow(iter, row)
    st iter = k ->
      pivot := row
    [ ]
  fi
af
```

The `in` statement accepts the pivot row only if the iteration number (passed as first argument) is right. Without this possibility, the receivers would have to buffer all incoming messages themselves, examine them, and block until they have received the right one. With the implementation given above, the SR run time system does the buffering automatically.

The second advantage of explicit message receipt in SR is the possibility to accept incoming messages in a user-specified order. Ordered message acceptance is used by the BoundManager process of the TSP program. 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 $M_2$ arrives, it also has to be multicast, because it is better than the current minimum ($M_1$). On the other hand, if the BoundManager 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.

```bash
process BoundManager
  do true ->
    in NewMinimum(v) by v ->
      if v < minimum ->
        minimum := v
        co (i := 1 to ncpus)
        call workers[i].UpdateMinimum(v)
      oc
      ni
      fi
  end
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.

### 4.2. Synchronous and asynchronous message invocations

Synchronous and asynchronous message passing each have their defenders. Synchronous message passing is sometimes claimed to be easier to understand and use, because there can be only one message pending between any two processes. Also, it requires less buffering than asynchronous messages. On the other hand, synchronous message passing frequently is overly restrictive and reduces parallelism. Not surprisingly, our SR programs use both forms of message passing.

The synchronous form is mostly useful if a reply for the message is expected. For example, in TSP and AB, if a worker needs to fetch a job from the manager, it uses synchronous communication (rendezvous or remote procedure call). There is little point in using asynchronous messages here, since the worker cannot continue anyway until it has received a job.

Asynchronous communication is preferred if a process wants to send information to other processes, but it does not care when these processes use the information. The SR programs show many examples of this kind. In TSP, if a worker finds a shorter full route, it sends this value to the BoundManager and immediately resumes its own computation. In ASP, the process having the pivot row for the current iteration multicasts it to all the others and continues. In both examples, the sender can continue immediately, doing useful computations.
4.3. Concurrent message invocations

The concurrent invocation (co) construct is used only a few times in our programs. It is used in TSP and ASP to multicast information. A concurrent send is not very useful in the current SR implementation, since it is not implemented as a real multicast. In other words, sending a message concurrently to $P$ processes will take $O(P)$ time, rather than constant time. For ASP, having a real multicast is much more useful, as shown in [11].

Another purpose of the co statement is to try multiple invocations in parallel, until one of them succeeds, and then abandon all the other calls. This is useful, for example, for reading one copy of a replicated file, where it does not matter which copy is read [17]. We have found no such uses for the statement in our applications.

5. Features not supported in SR

In our overview of SR we described several features supported in other languages than SR. We will now discuss them in more detail and see which of them would have been useful for our applications. Several of the problems described below are addressed in Version 2 of SR, which the designers are currently working on.

5.1. Mailboxes

The most prominent missing feature is the ability to send messages indirectly through mailboxes. In the replicated worker style programs (i.e., AB and TSP), mailboxes could have been used for communication between the manager and workers. If the manager has generated a job, it cannot send it out yet, since it does not know which worker will be available first. With mailboxes, the manager can put the job in a mailbox, where the first available worker can pick it up.

Of course, the same behavior can be obtained using an intermediate buffer process between the manager and workers. Our programs take a simpler approach and let the manager block until a worker asks for a job. 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. In AB and TSP, job generation is computationally inexpensive, so this is not a problem.

5.2. Message forwarding

The lack of a message forwarding facility in SR has already been noticed by Purdin, during the design of Saguaro. Purdin calls it "an oversight in the otherwise complete set of SR facilities" [18].

We have encountered only one case where message forwarding would have been useful, which is the queue process of the discrete event simulation package. Suppose a consumer in a queuing network requests an item from a queue. If the queue is not empty, the queue process responds immediately by sending the item. If the queue is empty, the consumer must be blocked until an item becomes available. One might be tempted to program this as

\begin{verbatim}
in TakeRequest(item) st queue not empty ->
    item := fetch next item from queue
ni
end
\end{verbatim}

With discrete event simulation, however, this does not work, because the scheduler must be informed that a process has blocked. (If all processes are blocked, the scheduler will set the simulated time to the time of the next event.) So, if a request comes in while the queue is empty, the request must be accepted, even though a reply cannot yet be sent to the consumer. If message forwarding had been allowed, this could have been programmed as follows:

\begin{verbatim}
in TakeRequest(item) =>
    if queue not empty ->
        # reply immediately
        item := fetch next item from queue
    [] else
        scheduler.passive()
        # inform scheduler
        forward TakeWait(item)
        # not allowed in SR
    fi
[] TakeWait(item) st queue not empty ->
    # return result to invoker of TakeRequest
    item := fetch next item from queue
ni
\end{verbatim}

If the queue is empty, the original TakeRequest is forwarded as a TakeWait operation to the queue process itself; only after this operation has been serviced, the consumer that invoked TakeRequest is reactivated.

Since message forwarding is not supported in SR, the above solution is not legal. Instead, the behavior must be simulated using multiple messages.
5.3. Shared variables
In SR, only processes running in the same resource can share variables. In fact, the primary purpose of a resource is to group together processes executing on the same node. For several parallel applications, however, the ability to share data between processes on different nodes is very useful [11]. One good example is the TSP program, in which all workers basically need to share the same node. For several parallel applications, however, the global bound variable. Since this variable is changed infrequently, it can be implemented efficiently in a distributed system by replicating it. In the SR program, this replication is done by the programmer: each worker maintains a local copy of the variable; after a write, all these copies are updated.

In Orca, processes can share data encapsulated in objects. The run time system of Orca takes care of the replication of objects [11]. TSP is simpler to program in Orca than in SR, because the programmer does not have to worry about replication. The global bound is put in an object shared among all the workers; this object is replicated by the run time system.

6. Ease of use
The second goal in the design of SR, besides expressiveness, is the ease of use. In other words, the language must be simple to understand and use. Simplicity and expressiveness are somewhat conflicting goals. Expressiveness might be obtained by including lots of features in the language, but this would clearly hurt simplicity. The SR designers have tried to solve this conflict by using a small number of concepts that can be combined in many ways. So, they have used the principle of orthogonality. Also, they have tried to integrate these concepts in a clean way with the sequential components of the language. Below, we will look at the number of concepts used in SR, their integration, and their semantics.

6.1. Number of concepts
In general, the designers of SR did a good job in reducing the number of concepts in the language. The best example of orthogonality in SR is the design of the message send and receive mechanisms. A message can be sent in two ways (synchronously and asynchronously) and received in two ways (explicitly and implicitly). Message sending and receiving are fully orthogonal in SR, so there are four different combinations, each yielding a different point-to-point IPC mechanism.

The SR select statement is not fully orthogonal, since it can only be used for receiving messages, not for sending them. In other words, like most other languages, SR supports input guards but not output guards [12]. Still, the SR select statement is much simpler than its counterpart in Ada, which has separate cases for timed entry calls and conditional entry calls. In addition, the SR select statement is more expressive. The guard expressions can access the parameters of the operation, which is not allowed in Ada.

In contrast with the message passing mechanism, the concurrent invocation (co) construct strikes one as rather ad hoc. This statement may use (1) call statements, (2) send statements, and (3) assignment statements whose right hand sides consist of single invocations. It cannot be used, for example, to create a number of resources in parallel, as in

```co
co (for i := 1 to ncpus) create worker( ... )
```
even though some people consider such a construct useful for highly parallel computing [19].

In addition to being ad hoc, SR's multicast facility has rather weak semantics. The co statement, for example, does not make any guarantees about the order in which messages will be delivered. If two processes simultaneously multicast two different operations, the receivers may obtain them in inconsistent orders. This weak semantic model makes programming difficult.

Another interesting attempt at economizing on the number of concepts has been made for abstract data types. The idea is to use resources not only as the unit of distribution, but also as the unit of data abstraction. For example, an abstract data type stack can be specified as:

```resource stack
  op push(item: int)
  op pop returns item: int
end stack```

The body of this resource may either implement a stack as a sequential or a distributed data type. The difference is that for distributed types the operations should synchronize. They should use semaphores or explicit receipt for mutual exclusion. For a sequential type, the operations can be implemented as regular procedures, without any synchronization. Users of the resource must be aware of this difference, however.

We think this approach is cleaner than that provided in most other languages. Ada, for example, uses distinct encapsulation mechanisms for both sequential and distributed data types (packages and tasks). Orca, on the other hand, goes much further than SR, and fully eliminates the distinction between sequential and distributed abstract data types. In Orca, each abstract type can be used for both local (nonshared) variables and distributed (shared) objects, no matter how the type is implemented. For operations on shared objects, the Orca runtime system automatically does mutual exclusion.

316
A major problem in using SR resources as abstract data types is the execution time overhead of the operations. In the current implementation, the overhead is about 400 microseconds [9], which is two orders of a magnitude higher than what is acceptable for sequential abstract data types. Unfortunately, the problem is not easy to solve, because it is not known at compile time whether the resource implementing the abstract data type is on the same node as the resource using it.

6.2. Integration of the concepts

Many parallel languages have problems in integrating parallel and sequential constructs. Especially languages that are intended for distributed systems and that extend existing sequential languages frequently run into problems with global variables, pointers, and call-by-reference parameters [11]. SR is not based on an existing language, and has a smooth integration of constructs as a design goal.

To some extent, SR succeeds in achieving this goal. One nice example is the treatment of procedures as a special case of operations. A regular procedure is just a locally declared operation that is invoked synchronously and serviced implicitly. Also, the syntax of the sequential and parallel constructs is similar.

The problem with global variables has been solved in a simple way. A variable declared in a resource can be used by all processes in that resource, but not by other processes. Although this is less flexible than allowing globally shared variables, it is cleaner than what most other distributed languages come up with. In Concurrent C [14], for example, the programmer can use global variables just as in sequential C, but it depends on the implementation what the semantics are. On a multiprocessor, the variable is truly shared: on a distributed system, each processor gets a copy of the variable, which is independent from all other copies.

With regard to pointers, SR suffers from the same problem as many other distributed languages. A pointer is only valid within one address space. If it is dereferenced on another machine, chaos will result. SR makes no restrictions on the usage of pointer variables, and leaves it up to the programmer to assure they are used in a proper way.

A third well-known problem area (besides global variables and pointers) is the parameter mechanism. A well-designed parameter mechanism should

1. Provide similar, if not identical, semantics for local procedures and remote operations (transparency concept), and

2. Be efficient at least in the normal (local) case.

SR provides copy-in/copy-out semantics for both local and remote operations. The parameter mechanism has uniform semantics, but unfortunately may be highly inefficient. Passing a large array as a parameter to a local procedure, for example, is extremely expensive, since the entire array has to be copied at least once. In Pascal, a programmer can avoid the overhead by using the var parameter mode. Even though this is far from elegant, it at least gives a simple way to get around the problem.

In SR, using the var mode causes the parameter to be copied twice (in and out), thus making things even worse. SR programmers can only obtain the required efficiency by explicitly passing a pointer to a data structure as the actual parameter. This solution is unattractive, however, since pointers in SR are unsafe and should only be used within a single address space. Moreover, using pointers for local calls and copy-in/copy-out for remote calls clearly violates the requirement of a uniform semantics.

As with the data abstraction mechanism described earlier, the design of the parameter mechanism is clean and elegant, but the inefficient implementation will prevent it from being used. Again, it is not clear if a more efficient implementation is possible. In theory, a compiler could sometimes optimize copy-in/copy-out into call-by-reference. To do this, it must prove that both mechanisms have the same effect. In the presence of pointers and potential aliases, however, this analysis may be hard to do, if not impossible.

A final problem we encountered is the lack of a simple, light-weight mechanism for sharing procedures among different resources. Each procedure (operation) is local to some resource and can only be called from outside that resource through an expensive remote procedure call. As an example, consider the TSP program again. Both the manager and the workers need to call a procedure present that checks if a certain city is on a given initial route. If this routine is included in the manager resource, all calls from the workers will be remote procedure calls, which is far too expensive. Putting the code in the worker resource has similar problems.

At present, the only way out is to put the source code in both resources. Clearly, this replication of source code is undesirable, because it complicates maintenance of programs. It would be much better to have static modules containing source code that can be shared among resources. SR already provides something similar, the global components, but these modules may only contain constants and types.
6.3. Semantics

With a few exceptions, the semantics of SR are clear and easy to understand. The exceptions are discussed below.

A serious problem with the semantics of SR is the large number of constructs that are not type-secure. The usage of pointers across machine boundaries has already been mentioned. Other examples are the lack of array bound checking (although this is implementation dependent) and the use of explicit, unchecked storage deallocation.

SR allows declarations and executable code to be mixed, which also is potentially insecure. In some cases, it is possible to access variables before storage has been allocated for them. Consider the following program fragment.

```plaintext
op foo() returns r: int
var n: int := foo()
var x[1:n]: int
proc foo() returns r: int
# implementation of operation foo
fa i := 1 to 10000 -> x[i] := 0 af
r := 10000
end
```

According to the SR language definition, this program is perfectly legal. Each variable or operation is declared before being used. Unfortunately, the latter condition only concerns the static ordering of declarations and usages. As shown above, the dynamic ordering may be subtly different. When the operation `foo` is invoked during the initialization of `n`, storage for the array `x` cannot have been allocated yet, since its size is not yet known. So, the operation `foo` will clear 10000 words of unallocated memory.

Another complaint concerns the semantics of concurrent access to shared variables. The SR language definition does not specify which operations on shared variables are atomic. For example, on most machines, simple read and write operations on integers will be done atomically. It is not clear at all, however, if an SR programmer is allowed to use this property, since the language definition does not even mention atomicity.

We have encountered one example where atomicity is important. In the TSP program, each worker maintains a copy of the global bound. This integer variable is updated asynchronously in the operation `UpdateMinimum`, which was shown before. If read and write operations on integers were atomic, no semaphores would be needed for mutual exclusion between `UpdateMinimum` and the main computation. As the variable is read very frequently, a lot of overhead would be saved. For a 12-city problem, the variable is read about a million times and the locking overhead is on the order of 10% of the total execution time.

The language definition should at least be explicit about atomicity. It should either state which operations are atomic (if any) or it should specify that atomicity is implementation dependent. At present, programmers are totally left in the dark.

7. Conclusions

We have implemented several parallel applications in SR and we have used the resulting programs to evaluate the design of this language with respect to parallel programming. In particular, we have looked at the design goals of expressiveness and simplicity.

The experiments have confirmed that SR is an expressive language, in which many problems can be solved in a straightforward way. Even within the small set of programs we have written, all of synchronous and asynchronous message invocation, explicit, implicit, conditional, and ordered message receipt, and multicast are used. We also found that message passing through mailboxes, message forwarding, and globally shared variables would have eased the implementation of some applications even further. These constructs are by no means essential, however, given the flexibility SR already provides.

In general, SR has also been simple to understand, despite its expressive power. Strong points in the design are: the message sending/receiving mechanism is orthogonal; substantial effort has been made to integrate the sequential and distributed constructs of the language in a clean way; the semantics of most parts of the language are simple to understand. Some less satisfying points in the design are: multicast communication is included in a rather ad hoc way; the language supports many constructs that violate type-security. Finally, the abstract data type mechanism and parameter mechanism of SR are implemented in a highly inefficient way and it is not clear whether efficient implementations are possible.

8. Acknowledgements

This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.). The author is grateful to the Department of Computer Science of the University of Arizona for receiving him as a visiting scholar. He would like to thank Greg Andrews, Dave Bakken, Gregg Townsend, Mike Coffin, and Ron Olsson for their discussions of SR, and Bruce Schatz for the use of his room and his Sun workstation. Finally, Greg Andrews, Andy Tanenbaum, Frans Kaashoek, and Arnold Geels provided useful comments on the paper.
References