SR

A Language for Parallel and Distributed Programming

Ronald A. Olsson
Gregory R. Andrews
Michael H. Coffin
Gregg M. Townsend

TR 92-09

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

SR: A Language for Parallel and Distributed Programming

March 9, 1992

Ronald A. Olsson

Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.
olsson@cs.ucdavis.edu

Gregory R. Andrews

Department of Computer Science
The University of Arizona
Tucson, AZ 85721 U.S.A.
greg@cs.arizona.edu

Michael H. Coffin

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
mhcoffin@watmsg.waterloo.edu

Gregg M. Townsend

Department of Computer Science
The University of Arizona
Tucson, AZ 85721 U.S.A.
gmt@cs.arizona.edu

Abstract

This paper introduces the newest version of the SR concurrent programming language and illustrates how it provides support for different execution environments, ranging from shared-memory multiprocessors to distributed systems. SR uses a few well-integrated mechanisms for concurrency to provide flexible, yet efficient, support for parallel and distributed programming. This paper gives several realistic examples to illustrate these language mechanisms.

1. Introduction

The SR concurrent programming language has been around, in one form or another, for over ten years. The earliest version, now called SR_0, contained mechanisms for asynchronous message passing and rendezvous [Andr81,Andr82]. Its form of rendezvous, unique at the time, provided a means by which the process servicing a rendezvous could choose which invocation to service based on the values of invocation parameters. Experience using SR_0 substantiated the general appropriateness of the language, but also pointed out several deficiencies. That experience led us to redesign the language [Andr86]. The result (SR version 1) [Olss86,Andr88] provided additional mechanisms for remote procedure call, dynamic process creation, and semaphores, as well as a means for specifying distribution of program modules.

Experience using version 1 of SR has led to further evolution of the language. Version 2 retains much of version 1's structure. However, it also enhances the mechanisms that support sharing of objects. This sharing is especially important in shared-memory environments, for which earlier versions of SR were not not really intended. (It is also important for supporting libraries, e.g., mathematical and windowing libraries.)

SR supports many 'features' useful for concurrent programming. However, our goals have always been to keep the language simple and easy to use, while at the same time to provide an efficient implementation. We achieve these goals by integrating common notions, both sequential and concurrent, into a few powerful mechanisms. We implement these mechanisms as part of a complete language to determine their feasibility and cost, to gain hands-on experience, and to provide a tool that can be used for research and teaching.

This paper introduces version 2 of SR, henceforth referred to as simply SR. It illustrates how a single language can provide support for different execution environments, ranging from shared-memory multiprocessors to distributed systems. This paper focuses on the highlights of the language; details can be found in [AnOl92].

The rest of this paper is organized as follows. Section 2 gives an overview of the SR model of computation. Section 3 describes how synchronization, sharing, and distribution are supported in SR. Section 4 illustrates, by means of examples, SR's language mechanisms that support parallel and distributed programming. Finally, Section 5 contains some concluding remarks, including a brief discussion of some current research related to SR.

2. SR Model of Computation

An SR program can execute within multiple address spaces, which can be located on multiple physical machines. Processes within a single address space can also share objects. Thus, SR supports programming in distributed environments as well as in shared-memory environments.

The SR model of computation allows a program to be split into one or more address spaces called *virtual machines.* Each virtual machine defines an address space on one physical machine. Virtual machines are created dynamically; they are referenced indirectly through *capability variables.* Virtual machines contain instances of two related kinds of modular components: *globals* and *resources.*

Each of these components contains two parts: a specification (aka a spec) and an implementation (aka a body). An import mechanism is used to make available in one component objects declared in the spec of another. In these two ways, globals and resources are similar to modules in Modula-2 [Wirt82] but they are created differently. Instances of resources are created dynamically, by an explicit create statement. These instances, and the services they provide, are referenced indirectly through *resource capability variables.* Instances of globals are also created dynamically. However, they are created implicitly as needed—specifically, when an instance of an importing resource or global is itself created and an instance of that global does not already exist on the same virtual machine. Furthermore, each virtual machine can contain only a single instance of a global. Globals, and the services they provide, can be referenced directly through their names.

The spec of a global or resource can contain declarations of types, constants, and operations; a global's spec can additionally contain declarations of variables. An operation defines a service that must be provided somewhere in the program. It can be considered a generalization of a procedure: it has a name, and can take parameters and return a result. An operation declared in a resource's spec must be serviced in that resource's body. Similarly, an operation declared in a global's spec *can* be serviced in the global's body; it can also be serviced within an importing resource or global.

The body of a global or a resource can contain declarations of additional objects; these objects are visible only within the body, not to any importer. Bodies also contain code that, among other things, services operations. The code is split into units called *processes* and *procs.* Processes are created implicitly when the enclosing global or resource is created. Instances of procs are created when they are invoked; they too execute as independent processes. All processes created within a global or a resource execute on the same virtual machine on which the enclosing global or resource was created. Processes and procs can declare additional variables and operations; they must contain the code that services invocations of any locally declared operations.

Figure 1 summarizes SR's model of computation. In its simplest form, a program consists of a single virtual machine executing on one physical machine, possibly a shared-memory multiprocessor. A program can also consist of multiple virtual machines executing on multiple physical machines. Hybrid forms are possible and in fact useful.

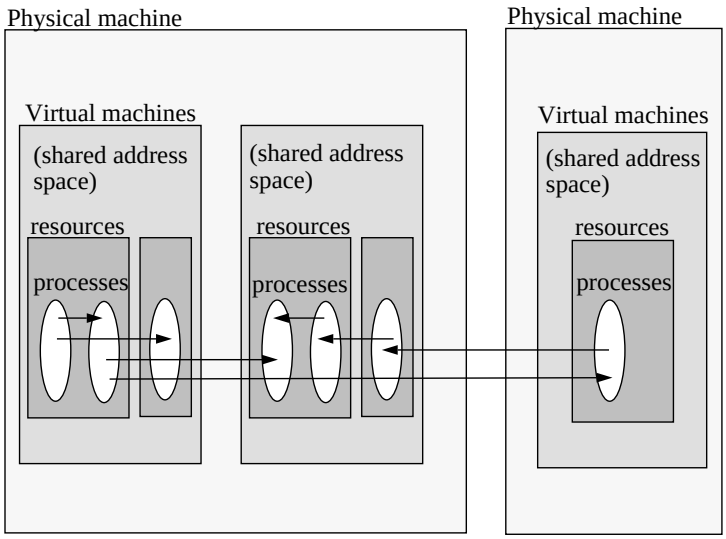

Figure 1. SR model of computation

Data and processor(s) are shared within a virtual machine; different virtual machines can be placed on (distributed across) different physical machines.

Processes on the same or different virtual machines can communicate through operation invocation. Operations may be invoked directly through the operation's declared name or through a resource capability variable; or they may be invoked indirectly through an *operation capability variable.* These capability variables are strongly typed and may point to operations with structurally equivalent signatures. They may also be passed as parameters to operations during invocation or to resources during resource creation, allowing processes in different resource instances, on possibly different virtual machines, to communicate.

Communication between processes is independent of their virtual machine locations. For example, message passing between processes in the same resource instance has the same syntax and semantics as message passing between processes on different virtual machines.

3. Language Support

This section describes how synchronization, sharing, and distribution are supported in SR. The examples in Section 4 will illustrate these points in the context of specific programming situations.

3.1. Support for Synchronization

SR is rich in the functionality it provides for concurrent programming: dynamic process creation, semaphores, message passing, remote procedure call, and rendezvous. However, these are all provided through a single mechanism: the operation.

The key idea is that operations can be invoked in two ways, synchronously (`call`) or asynchronously (`send`), and can be serviced in one of two ways, by procs or by input statements (`in`). This yields the following four combinations:

Invoke	*Service*	*Effect*
`call`	`proc`	(possibly remote) procedure call
`call`	`in`	rendezvous
`send`	`proc`	dynamic process creation
`send`	`in`	asynchronous message passing

One virtue of this approach is that it allows the declaration of an operation to be separated from the code that services it (i.e., proc or input statements). This allows resource and global specifications to be written and used without concern as to how an operation is serviced.

SR provides abbreviations of the above basic mechanisms to simplify the most common usages such as background process creation, semaphores, and simple asynchronous message passing. Briefly: a `process` is an abbreviation for a proc and an implicit send to it when the enclosing resource or global is created; a `sem` declaration is an abbreviation for an operation declaration, a `P` is an abbreviation for an input statement, and a `V` is an abbreviation for a send; and a `receive` statement is an abbreviation for a simple form of input statement.

SR also provides three statements—forward, return, and reply—that provide additional flexibility in servicing invocations. (However, none of these statements are used in the examples in this paper.) A process executing a reply statement causes the invocation being serviced to complete; result parameters and return values are immediately passed back to the caller. The process that executes a reply statement then continues execution with the statement following the reply.

3.2. Support for Sharing

SR provides support for sharing on several levels. First, processes within a resource instance can share variables. They can coordinate access to shared variables through shared semaphores or other operations declared within the resource. Second, processes that execute in possibly different resource instances but on the same virtual machine can share variables and operations declared in the spec of globals.

Consider, for example, a program that is to be written for execution on a shared-memory multiprocessor. It might be written as a single resource program, with processes sharing variables and operations declared at the resource level. For a program of any complexity, though, splitting the program into multiple resources is desirable. This kind of structure is possible, too. Resources can be created on a single virtual machine, with shared variables and operations declared in one or more globals.

3.3. Support for Distribution

As suggested in Section 2, virtual machines are the unit for program distribution. They can be created (or destroyed) dynamically as needed in response to program execution. Instances of resources and globals can then be created on virtual machines. Processes in different virtual machines communicate with other processes by invoking operations.

Operation invocations exhibit two kinds of transparency. First, an operation is invoked in the same way regardless of how a program is distributed. The invocations by the client of the operations in the server remain the same regardless of whether the client and server are located on the same virtual or physical machine. Second, an operation is invoked in the same way regardless of how the operation is serviced, i.e., by an input statement or by a proc.

4. Example: Parallel Matrix Multiplication

Matrix computations lie at the heart of most scientific computing problems. Matrix multiplication is one of the most basic of these computations. Here we develop four realistic algorithms. Two employ shared variables, and hence are suitable for execution on shared memory multiprocessors. The other two algorithms employ message passing, and hence are suitable for execution on distributed memory systems. Each algorithm also illustrates a different programming technique and a different combination of SR mechanisms.

The problem is to compute the product of two $n \times n$ real matrices a and b. This requires computing n^2 inner products, one for each combination of a row of a and a column of b. On a massively parallel, synchronous multiprocessor, all inner products could be computed in parallel with reasonable efficiency since, by default, every processor executes the same sequence of instructions at the same time. However, on an asynchronous multiprocessor each process has to be created and destroyed explicitly, and each inner product requires relatively little computation. In fact, the parallel program would be much slower than a sequential program since the cost of creating and destroying processes would far outweigh any benefits derived from parallel execution.

To execute efficiently on an asynchronous multiprocessor, each process in a parallel program must perform quite a bit of work relative to the amount of time it takes to create the process and the amount of time the process spends communicating and synchronizing with other processes. In short, the sequential execution time of the process must be much greater than the concurrency and communication overhead. The exact balance depends, of course, on the underlying hardware and on the concurrent programming mechanisms that are employed. This section develops four matrix-multiplication algorithms that employ different combinations of communication and synchronization mechanisms. Each can readily be modified to alter the balance between sequential execution time and concurrency overhead.

4.1. Pre-Scheduled Strips

Given are real matrices `a[N,N]`, `b[N,N]`, and `c[N,N]`. Assume that these are shared variables, and that we wish to use `PR` processes to compute the product of `a` and `b` and store it in `c`. For simplicity, we will also assume that `N` is a multiple of `PR`; for example, `N` might be 100 and `PR` might be 10.

To balance the amount of computation performed by each process, each should compute $\texttt{N}^2/\texttt{PR}$ inner products. The simplest way to do this is to assign each process responsibility for computing the values for all elements in a strip of matrix `c`. In particular, let `S` be `N/PR`. Then the first process could compute the values of the first `S` rows of `c`, the second could compute the values of the next `S` rows of `c`, and so on. This kind of approach is sometimes called prescheduling since each process is assigned in advance a certain number of ''chores,'' i.e., inner products in this case.

To implement this algorithm in SR, we will use one global and one resource, which are compiled in that order. The global, shown in Figure 2, declares the shared constants `N`, `PR`, and `S` and reads values for `N` and `PR` from the command line. It then computes `S`; if `N` is not a multiple of `PR`, the global prints an error message and stops the program. Variables `N` and `PR` are given default initial values; these are used if there are no command-line arguments. (Calling `getarg` has no effect if there is no corresponding argument.)

The resource, shown in Figure 3, declares the matrices and an array of `PR` processes to compute the inner products. It also contains a process that implements a barrier synchronization point and final code to print results. Each instance of process `strip` first initializes its bands of matrices `a`, `b`, and `c`. For simplicity, we have initialized all elements of `a` and `b` to 1.0; in general, initial values would come from a prior computation or from external files.

Because all elements of `a` and `b` must be initialized before they are used by other processes, we need to implement a barrier synchronization point. Here we have simply used two semaphores and a coordinator process. The `coordinator` first waits for all `PR` instances of `strip` to signal semaphore `done`, then it signals semaphore `continue` `PR` times. Since the barrier is executed only once, this approach is reasonable for this program. In general, however, one will want to use one of the more efficient barriers described in [Andr91] or [MCS91].

```
global sizes
  var N := 10    # matrix dimension, default 10
  var PR := 2    # number of processes, default 2
  var S: int     # strip size
body sizes
  getarg(1, N); getarg(2, PR); S := N/PR
  if N mod PR != 0 ->
    write("N must be a multiple of PR"); stop (1)
  fi
end
```

Figure 2. Global `sizes` for strips algorithm

```
resource mult()
  import sizes
  var a[N,N], b[N,N], c[N,N]: real
  sem done := 0, continue := 0

  process strip(p := 1 to PR)
    const R := (p-1)*S + 1  # starting row of strip
    # initialize parts of a and b
    fa i := R to R+S-1, j := 1 to N ->
      a[i,j] := 1.0; b[i,j] := 1.0
    af
    # barrier to wait for all initialization
    V(done); P(continue)
    # compute S*N inner products
    fa i := R to R+S-1, j := 1 to N ->
      var inner_prod := 0.0  # local accumulator
      fa k := 1 to N ->
        inner_prod +:= a[i,k]*b[k,j]
      af
      c[i,j] := inner_prod
    af
  end

  process coordinator
    fa i := 1 to PR -> P(done) af
    fa i := 1 to PR -> V(continue) af
  end

  final  # print results
    fa i := 1 to N ->
      fa j := 1 to N -> writes(c[i,j], " ") af
      write()
    af
  end
end
```

Figure 3. Resource `mult` for strips algorithm

The finalization code in `mult` is executed when all instances of `strip` have terminated. That code prints the results. This use of finalization code frees the programmer from having to program termination detection.

Many shared-memory multiprocessors employ caches, with one cache per processor. Each cache contains the memory blocks most recently referenced by the processor. (A block is typically a few contiguous words.) The purpose of caches is to increase performance, but they have to be used with care by the programmer or they can actually

decrease performance (due to cache conflicts). Hill and Larus [HiLa90] give three rules-of-thumb programmers need to keep in mind:

- Perform all operations on a variable, especially updates, in one process (processor).
- Align data so that variables updated by different processors are in different cache blocks.
- Re-use data quickly when possible so that it remains in the cache and does not get ''spilled'' back to main memory.

Since SR stores matrices in row-major order (i.e., by rows), the above program uses caches well. In particular, each `strip` process reads one distinct strip of `a` and writes one distinct strip of `c`, and it references elements of `a` and `c` by sweeping across rows. Every process references all elements of `b`, but that is unavoidable. (If `b` were transposed, so that columns were actually stored in rows, it too could be referenced efficiently.)

4.2. Dynamic Scheduling: A Bag of Tasks

The algorithm in the previous section statically assigned an equal amount of work to each `strip` process. If the processes execute on homogeneous processors without interruption, they would be likely to finish at about the same time. However, if the processes execute on different speed processors, or if they can be interrupted—e.g., in a timesharing system—then different processes might complete at different times. To dynamically assign work to processes, we can employ a *shared bag of tasks.* This approach uses a shared work queue (represented by an operation). Initially, an administrator process places in the bag the initial tasks to be solved. Multiple worker processes take tasks from the bag and service them. For this problem, a task corresponds to the finding the `N` inner products for a given row of the result matrix `c`. More generally, the worker processes often generate new tasks—corresponding to subproblems—that are put into the bag. This is the case in one solution to adaptive quadrature [AnOl92]. There, worker processes are given tasks of approximating the area for a given interval; they add new tasks—corresponding to finding areas for two sub-intervals—to the bag if their approximation was not acceptable. In this section, we present a matrix multiplication program that implements a shared bag of tasks solution.

As in the previous program, we again employ one global and one resource. The global, shown in Figure 4, declares the matrix dimension `N` and the number of worker processes `W`, and reads values for these variables from the command line. As before, the shared variables are given default initial values.

The resource `mult`, shown in Figure 5, imports `sizes` and declares shared matrices `a`, `b`, and `c`; the sizes of these matrices again depend on `N`. The resource then declares an operation, `bag`, which is shared by the `worker` processes in the resource. The initialization code in `mult` sets all elements of `a` and `b` to 1.0 and sends each row index to `bag`. After initialization has completed, the worker processes are created. Each worker process repeatedly receives a row index `i` from `bag` and computes `N` inner products, one for each element of row `i` of result matrix `c`. The computation terminates when `bag` is empty and all worker processes are blocked waiting to receive from it. At this point, the finalization code is executed; it prints out the values in `c`.

This program has been executed on a Sequent multiprocessor using 1, 2, 4, and 8 workers and processors. It shows nearly perfect speedup for reasonable-size matrices, e.g., when `N` is 100 or more. In this case, the amount of computation per iteration of a worker process far outweighs the overhead of receiving a message from the bag. Like the previous program, this one uses caches well since SR stores matrices in row-major order, and each worker fills in an entire row of `c`. If the bag of tasks contained column indices instead of row indices, performance would be much worse since workers would encounter cache update conflicts.

4.3. A Distributed Broadcast Algorithm

The program in the previous section can be modified so that the workers do not share the matrices or bag of tasks. In particular, each worker (or address space) could be given a copy of `a` and `b`, and an administrator process could

```
global sizes
  var N := 10    # matrix dimension, default 10
  var W := 2     # number of workers, default 2
body sizes
  getarg(1, N); getarg(2, W)
end
```

Figure 4. Global `sizes` for bag of tasks algorithm

```
resource mult()
  import sizes
  var a[N,N], b[N,N], c[N,N]: real
  op bag(row: int)

  # initialize the arrays and bag of tasks
  fa i := 1 to N ->
    fa j := 1 to N ->
      a[i,j] := 1.0; b[i,j] := 1.0
    af
    send bag(i)
  af

  process worker(id := 1 to W)
    var i: int    # index of row of c to compute
    do true ->
      receive bag(i)
      fa j := 1 to N ->
        var inner_prod := 0.0
        fa k := 1 to N ->
            inner_prod +:= a[i,k]*b[k,j]
        af
        c[i,j] := inner_prod
      af
    od
  end

  final
    fa i := 1 to N ->
      fa j := 1 to N -> writes(c[i,j], " ") af
      write()
    af
  end
end
```

Figure 5. Resource `mult` for bag of tasks algorithm

dispense tasks and collect results. With these changes, the program could execute on a distributed memory machine.

This section and the next present two additional distributed algorithms for matrix multiplication. To simplify the presentation, we use N^2 processes, one to compute each element of `c`. Initially each such process also has the corresponding values of `a` and `b`. In this section, we have each process broadcast its value of `a` to other processes on the same row and broadcast its value of `b` to other processes on the same column. In the next section, we have each process interact only with its four neighbors. Both algorithms can readily be generalized to use fewer processes, each of which is responsible for a block of matrix `c`.

Our broadcast implementation of matrix multiplication uses three components: a global, a resource to compute elements of `c`, and a main resource. They are compiled in that order. The global, shown in Figure 6, declares and reads a command-line argument for the matrix dimension `N`.

Instances of resource `point`, shown in Figure 7, carry out the computation. The main resource creates one instance for each value of `c[i,j]`. Each instance exports three operations: one to start the computation, one to exchange row values, and one to exchange column values. Operation `compute` is implemented by a `proc`; it is invoked by a send statement in the main resource and hence executes as a process. The arguments of the `compute` operation are capabilities for other instances of `point`. Operations `rowval` and `colval` are serviced by `receive` statements; they are invoked by other instances of `point` in the same row `i` and column `j`, respectively.

The N^2 instances of `point` interact as follows. The `compute` process in `point` first sends its value of `aij` to the other instances of `point` in the same row and receives their elements of `a`. The `compute` process then sends

```
global sizes
  var N := 6    # matrix dimension, default 6
body sizes
  getarg(1, N)
end
```

Figure 6. Global `sizes` for distributed broadcast algorithm

```
resource point    # one instance per point
  op compute(rlinks[*], clinks[*]: cap point)
  op rowval(sender: int; value: real)
  op colval(sender: int; value: real)
body point(i, j: int)
  import sizes
  var aij := 1.0, bij := 1.0, cij := 0.0
  var row[N], col[N]: real
  row[j] := aij; col[i] := bij

  proc compute(rlinks, clinks)
    # broadcast aij to points on same row
    fa k := 1 to N st k != j ->
      send rlinks[k].rowval(j, aij)
    af
    # acquire other points from same row
    fa k := 1 to N st k != j ->
      receive rowval(sender, row[sender])
    af
    # broadcast bij to points on same column
    fa k := 1 to N st k != i ->
      send clinks[k].colval(i, bij)
    af
    # acquire other points from same column
    fa k := 1 to N st k != i ->
      in colval(sender, v) -> col[sender] := v ni
    af
    # compute inner product of row and col
    fa k := 1 to N -> cij +:= row[k]*col[k] af
  end

  final writes(cij, " ") end
end point
```

Figure 7. Resource `point` for distributed broadcast algorithm

its value of `bij` to other instances of `point` in the same column and receives their elements of `b`. After these two data exchanges, `point(i,j)` now has row `i` of `a` and column `j` of `b`. It then computes the inner product of these two vectors. The final code prints out the value of `cij`. It is executed when the resource instance is destroyed explicitly. (Only the initial instance of the main resource is destroyed implicitly.)

The main resource, shown in Figure 8, creates N^2 instances of `point` and gets back a capability for each, which it stores in matrix `pcap`. It then invokes the `compute` operations, passing each instance of `point` capabilities for other instances in the same row and column. We can use a row slice `pcap[i,1:N]` to pass row `i` of `pcap` and a column slice `pcap[1:N,j]` to pass column `j` of `pcap` to `compute`. When the program terminates, the final code in `main` is executed. It destroys instances of `point` in row-major order, which causes the elements of `c` to be printed in row-major order.

As noted, this program can readily be modified to have each instance of `point` start with a block of `a` and a block of `b` and compute all elements of a block of `c`. The basic algorithmic structure and communication pattern

```
resource main()
  import sizes, point
  var pcap[N,N]: cap point
  # create points
  fa i := 1 to N, j := 1 to N ->
    pcap[i,j] := create point(i, j)
  af
  # give each point capabilities for its neighbors
  fa i := 1 to N, j := 1 to N ->
    send pcap[i,j].compute(pcap[i,1:N], pcap[1:N,j])
  af

  final
    fa i := 1 to N ->
      fa j := 1 to N -> destroy pcap[i,j] af
      write()
    af
  end
end
```

Figure 8. Main resource for distributed broadcast algorithm

would be identical.

This program executes on only a single virtual machine, and therefore also on a single physical machine. However, it can be easily modified so that, for example, instances of the `point` resource for a given row are placed in their own virtual machine. Only `main`'s loop that creates resources needs to be changed; the new loop is:

```
fa i := 1 to N ->
  var vmcap: cap vm
  vmcap := create vm()
  fa j := 1 to N ->
    pcap[i,j] := create point(i, j) on vmcap
  af
af
```

Each iteration of the outer loop creates a new virtual machine (by creating a new instance of `vm`); the inner loop then creates instances of `point` on that virtual machine. The above loop can be further modified, for example, so that each virtual machine is on a different physical machine. For example, the assignment statement that creates virtual machines can be changed to the following:

```
vmcap := create vm() on i
```

The value of `i` is taken to be a physical machine number; its use is installation dependent but can be made to be relatively portable.

4.4. A Distributed Heartbeat Algorithm

In the broadcast algorithm, each instance of `point` acquires an entire row of `a` and an entire column of `b` and then computes their inner product. Also, each instance of `point` communicates with all other instances on the same row and same column. Here we present a matrix multiplication algorithm that employs the same number of instances of a `point` resource. However, each instance holds only one value of `a` and one of `b` at a time. Also, each instance of `point` communicates only with its four neighbors. Again the algorithm can readily be generalized to work on blocks of points and to execute on multiple virtual machines.

As in the broadcast algorithm, we will use N^2 processes, one to compute each element of matrix `c`. Again, each initially also has the corresponding elements of `a` and `b`. The algorithm consists of three stages [Manb89]. In the first, processes shift values in `a` circularly to the left; values in row `i` are shifted left `i` columns. Second, processes shift values in `b` circularly up; values in column `j` are shift up `j` rows. The following display illustrates the result of the initial rearrangement of the values of `a` and `b` for a 3×3 matrix:

```
a[1,2], b[2,1]   a[1,3], b[3,2]   a[1,1], b[1,3]

a[2,3], b[3,1]   a[2,1], b[1,2]   a[2,2], b[2,3]

a[3,1], b[1,1]   a[3,2], b[2,2]   a[3,3], b[3,3]
```

In the third stage, each process multiplies one element of `a` and one of `b`, adds the product to its element of `c`, shifts the element of `a` circularly left one column, and shifts the element of `b` circularly up one row. This compute and shift sequence is repeated `N-1` times, at which point the matrix product will have been computed.

We call this kind of algorithm a *heartbeat algorithm* since the actions of each process are like the beating of a heart: first send data out to neighbors, then bring data in from neighbors and use it. To implement the algorithm in SR, we again use one global and two resources, as in the broadcast algorithm. The global, shown in Figure 9, is identical to the one in the previous section.

The computation is carried out by N^2 instances of a `point` resource, shown in Figure 10. It exports three operations as did its counterpart in the previous section. However, here the `compute` operation passes capabilities for only the left and upward neighbors, and the `rowval` and `colval` operations are invoked by only one neighbor. Also, the body of `point` implements a different algorithm.

Finally, the main resource, shown in Figure 11, creates instances of `point` and passes each capabilities for its left and upward neighbors. Function `prev` in `main` uses modular arithmetic so that instances of `point` on the left and top borders communicate with instances on the right and bottom borders, respectively.

5. Concluding Remarks

The examples in the previous section illustrated the flavor of SR programming for parallel and distributed environments, as well as some specific usages of SR mechanisms. SR's mechanisms can also be used in ways not illustrated by the examples. For example, SR allows operations to be declared within processes, or even within blocks of code, and allows these local operations to be assigned to operation capability variables. These are useful, for example, in programming *conversational continuity*. In such an interaction, a client process interacts with a server process and wishes to carry out a private conversation with it (see [Andr91]).

In many ways, the mechanisms that SR provides for sharing, distribution, and synchronization are a superset of those found in other languages, such as Ada [US83], Concurrent C [Geha89], Argus [Lisk83], and occam [Burn88]. SR achieves this flexibility by having just a few well-integrated mechanisms, which can be used alone or freely in combination with others. One interesting question is whether such generalization is inherently more costly. For example, since SR operations subsume rendezvous, local procedure call, remote procedure call, process creation, semaphores, etc., are they therefore expensive to use? Our implementation currently recognizes some commonly occurring patterns and generates lower-cost code than would be required in the worst, most general case. The version 1 SR compiler, for example, optimizes certain message passing scenarios to use low-cost semaphores, and certain remote procedure call scenarios to use conventional procedure call. The results are that the cost of synchronization in SR is competitive with those reported for other languages [Atki88].

One current effort involves identifying further optimization of synchronization mechanisms, including those that cross resource boundaries. Our overall approach applies source-level transformations to concurrent programs, replacing costly synchronization mechanisms with less costly ones [McOl90a,OlMc91]. The techniques involve the application of dataflow analysis and an extension of interprocedural analysis and inter-module analysis to concurrent programs. An interesting aspect of this work is the use of attribute grammars to perform such analysis [McOl90b]. These techniques are also applicable to other programming languages, e.g., Ada, Concurrent C, Argus, and occam.

Version 2 of SR works on a variety of UNIX-based systems, including a Sequent multiprocessor, and is in the public domain. For information on how to obtain SR, contact the authors or the SR project (by electronic mail to sr-project@cs.arizona.edu).

```
global sizes
  var N := 6    # matrix dimension, default 6
body sizes
  getarg(1, N)
end
```

Figure 9. Global `sizes` for distributed heartbeat algorithm

```
resource point    # one instance per point
  op compute(left, up: cap point)
  op rowval(value: real), colval(value: real)
body point(i, j: int)
  import sizes
  var aij: real := i, bij: real := j, cij := 0.0

  proc compute(left, up)
    # shift values in aij circularly left i columns
    fa k := 1 to i ->
      send left.rowval(aij); receive rowval(aij)
    af
    # shift values in bij circularly up j rows
    fa k := 1 to j ->
      send up.colval(bij); receive colval(bij)
    af
    cij := aij*bij
    fa k := 1 to N-1 ->
      # shift aij left, bij up, then multiply
      send left.rowval(aij); send up.colval(bij)
      receive rowval(aij); receive colval(bij)
      cij +:= aij*bij
    af
  end

  final writes(cij, " ") end
end point
```

Figure 10. Resource `point` for distributed heartbeat algorithm

```
resource main()
  import sizes, point
  var pcap[N,N]: cap point

  procedure prev(index: int) returns lft: int
    lft := (index-2) mod N + 1
  end

  # create points
  fa i := 1 to N, j := 1 to N ->
    pcap[i,j] := create point(i, j)
  af
  # give each point capabilities for its left
  # and upward neighbors
  fa i := 1 to N, j := 1 to N ->
    send pcap[i,j].compute(pcap[i,prev(j)], pcap[prev(i),j])
  af

  final
    fa i := 1 to N ->
      fa j := 1 to N -> destroy pcap[i,j] af
      write()
    af
  end
end
```

Figure 11. Main resource for distributed heartbeat algorithm

Acknowledgements

This work was supported in part by National Science Foundation grant CCR-8810617 at the University of California, Davis, and by NSF grants CDA-8822652, CCR-8811423, and CCR-9108412 at the University of Arizona.

References

[Andr81] Andrews, G.R. Synchronizing resources. *ACM Trans. on Prog. Languages and Systems 3*, 4 (Oct. 1981), 405-430.

[Andr82] Andrews, G.R. The distributed programming language SR—mechanisms, design and implementation. *Software—Practice and Experience 12*, 8 (Aug. 1982), 719-754.

[Andr86] Andrews, G.R., and Olsson, R.A. The evolution of the SR language. *Distributed Computing 1*, 3 (July 1986), 133-149.

[Andr88] Andrews, G.R., Olsson, R.A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G. An overview of the SR language and implementation. *ACM Trans. on Prog. Lang. and Systems 10*, 1 (Jan. 1988), 51-86.

[Andr91] Andrews, G.R. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, 1991.

[AnOl92] Andrews, G.R., and Olsson, R.A. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, to appear 1992.

[Atki88] Atkins, M.S., and Olsson, R.A. Performance of multitasking and synchronization mechanisms in the programming language SR. *Software—Practice and Experience 18*, 9 (Sept. 1988), 879-895.

[Burn88] Burns, A. *Programming in occam 2*. Addison-Wesley, 1988.

[Geha89] Gehani, N. *The Concurrent C Programming Language*. Silicon Press, Summit, New Jersey, 1989.

[HiLa90] Hill, M.D., and Larus, J.R. Cache considerations for multiprocessor programmers. *Comm. ACM 33*, 8 (Aug. 1990), 97-102.

[Lisk83] Liskov, B., and Scheifler, R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Trans. on Prog. Lang. and Systems 5*, 3 (July 1983), 381-404.

[Manb89] Manber, U. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, Reading, MA, 1990.

[McOl90a] McNamee, C.M., and Olsson, R.A. Transformations for optimizing interprocess communication and synchronization mechanisms. *International Journal of Parallel Programming 19*, 5 (Oct. 1990) pages 357-387.

[McOl90b] McNamee, C.M., and Olsson, R.A. An attribute grammar approach to compiler optimization of intramodule interprocess communication. Submitted for publication. (An earlier version appears as CSE-89-11, Div. of Computer Science, The University of California, Davis, 1989.)

[MCS91] Mellor-Crummey, J.M., and Scott, M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems 9*, 1 (Feb. 1991), 21-65.

[OlMc91] Olsson, R.A., and McNamee, C.M. An overview of compiler optimization of interprocess communication and synchronization mechanisms. Proceedings of the 1991 *International Conference on Parallel Processing*, St. Charles, IL, II-31 to II-35, August 12-17, 1991.

[Olss86] Olsson, R.A. *Issues in Distributed Programming Languages: The Evolution of SR.* TR 86-21 (Ph.D. Dissertation), Dept. of Computer Science, The University of Arizona, August 1986.

[US83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, 1983.

[Wirt82] Wirth, N. *Programming in Modula-2*. Springer-Verlag, New York, 1982.