A Parallel Language for a Distributed-Memory Multiprocessor

Russell M. Clapp and Trevor Mudge *
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Abstract

Development of software for MIMD multiprocessor architectures is impeded by the lack of adequate programming environments for them. A vital component of these environments are high-level parallel programming languages which are needed to support the development of applications that are capable of exploiting parallelism. A parallel language provides a machine independent view of parallelism, but can also be sensitive to some major characteristics of the architecture (e.g., the hypercube interconnect). The language may also run efficiently if carefully constructed language specific run-time system code is used to support parallelism instead of general operating system primitives. In this paper we describe an approach to parallel programming using a process-model parallel language for a hypercube. This language is similar to other concurrent languages, but is different in that it is intended for *general-purpose* parallel programming. Process-model parallel programming primitives have been added by us to the C language with emphasis being placed on simplicity, flexibility, and efficiency. In addition to describing the process model language, we discuss the design and implementation of the run-time system. This set of run-time routines is responsible for supporting the process model efficiently, thus making the language a practical parallel programming tool.

1 Introduction and Motivation

The advent of parallel processors has led to the need for effective ways to program them. Although these machines have been available for some time, the techniques used to program them are still immature, and in most cases are ad hoc [11, 2, 12]. As technology has evolved, many different parallel computer architectures have been proposed and implemented. With architectures ranging from systolic arrays to shared-bus multiprocessors, there have been a wide variety of programming paradigms employed. Although we would like to see program parallelism expressed in an entirely machine independent fashion, it is clear, at this point, that some paradigms are not suitable for certain architectures, particularly if performance is important.

Our goal is to develop a parallel language suitable for distributed-memory multiprocessors. The language is intended to be easy to learn and understand, while providing an environment for programming distributed-memory multiprocessors that is superior to the ones available today. Current techniques for programming these machines usually involve the Single Program Multiple Data (SPMD) style [11, 4]. In this paradigm, a single program is written, and a copy of it is run on each processor. Communication occurs through explicit operating system calls. This style of programming is inherently machine dependent, and is difficult to write and debug. In the SPMD paradigm, the software becomes a processor manager and merely provides an interface between the individual processor's instruction set and the programmer's algorithm. We believe that a better medium is needed for expressing the parallelism of the algorithm.

In this paper, we discuss the design and implementation issues of a simple parallel language. A major goal of our design is to provide for an efficient implementation. The approach we take is to provide programming techniques that are somewhat similar to the current approach, while correcting the glaring deficiences of the SPMD style. The constructs of the language are provided with the target architecture and run-time system implementation in mind. However, the language does provide for some degree of machine independence, without presenting a view so abstract that algorithms cannot take advantage of the underlying architecture. In the next section, we discuss the programming paradigm of distributed-memory multiprocessors from a procedural point of view. We then relate our experiences with implementing Ada on a hypercube multiprocessor. Based on this discussion, we propose some simple parallel constructs that can be added to a sequential programming language (e.g. C) without the need for elaborate run-time system support. In the following section, we discuss the the requirements of the run-time system and their implementation strategies. Finally, we give some concluding remarks along with the status of our implementation.

2 The Programming Paradigm

As stated above, we intend to pursue a procedural style of programming. The specifics of this style are determined by the model of the target architecture. Accordingly, we describe a simplified view of distributed-memory architectures and then use this description to motivate a programming

*This work was supported in part by the National Science Foundation, grant number MIP-8802771.

paradigm. This basic paradigm is then compared with an approach to programming a distributed-memory machine that involves using the Ada language. We examine this approach and compare it with our basic requirements for a parallel programming language. This study suggests a basic set of parallel language constructs that can be implemented efficiently.

2.1 The System Model

The architectures of multiprocessing systems fall into four broad categories. They are, 1) shared-memory multiprocessors, 2) non-uniform access shared-memory multiprocessors, 3) distributed-memory multiprocessors (DMMs), and 4) distributed computer systems. We classify hypercubes as DMMs. Such systems are typically made up of homogeneous processing elements with local memories connected together via some dedicated network. Because the network is used to connect the processing elements, remote memories are not directly accessible. Therefore, we do not classify these machines in either of the shared-memory categories. Hypercubes and other DMMs are also not considered distributed systems, because these configurations are typically made up of autonomous computer systems that are interconnected on either a broadcast or token-ring style local area network. In addition, these computers typically provide more resources than the processing elements in a hypercube, such as disks, keyboard, monitor, etc.

Our model of DMMs is one where multiple processing elements and memories are interconnected via a fixed, usually regular, network and are able to share resources such as disk files and monitors. We propose a programming model where a single program provides multiple execution threads, where a thread is a single program counter associated with a program fragment. Each thread has its own data area and is allowed to communicate with other threads by sending messages. We call a thread of execution a *process* and its language level representation the *process model*. This programming model is quite similar to the current SPMD style of programming, but there are some important advantages to using the process-model approach. By providing mechanisms for programming multiple processors with a single program the benefits of type checking across processor boundaries and machine independent interprocess communication are possible [5]. Furthermore, the number of program processes and the number of available processors need not be the same, allowing for program development on uniprocessor systems as well as the possibility of dynamic reconfiguration either for load balancing or to avoid faulty processors.

2.2 The Ada Experience

Our initial approach to providing a process-model parallel language for a hypercube was to implement an existing language, Ada. In addition to providing a block structured procedural language with variable scoping, Ada also provides a process model in the form of *tasks*. A brief overview of Ada is given in [5], together with a discussion of the run-time system requirements and our approach toward implementing them. Additional details of our implementation are given in [6, 7].

There are several problems that arise when implementing Ada on a a DMM. These problems occur mainly because of the shared-memory model that the language provides to the user. Although this problem can be isolated to a large extent, it hampers the freedom of the programmer to express parallelism effectively. We outline the main problems with supporting the Ada tasking model on a DMM below.

Units of Distribution The first issue to address in targeting Ada to a distributed-memory computer is to decide which program units are suitable to be distributed across the processing elements. While tasks may seem to be the obvious answer to this question, there are many undesirable side effects that occur when any task is allowed to be distributed [19, 16]. In fact, the recommendation is made in [19] to allow only library packages and library subprograms to be distributed. Packages are the unit of encapsulation in Ada, and may contain a set of variable and type definitions, subprograms (functions and procedures), and tasks. The specification of a package exports the visibility of a subset of the objects and types declared within it. By using library packages and library subprograms (those units that may be compiled separately and are not contained in enclosing packages) as the units of distribution, the concept of binding types, variables, tasks, and subprograms to a processing element is introduced. While this view may be appropriate for distributed embedded systems, we feel it is not consistent with our view of a programming paradigm for a DMM.

The Cactus Stack The main problem with supporting Ada on a DMM is the language's shared-memory model. Because Ada supports Algol like scoping and nesting of subprograms and tasks, a careless distribution of objects across processors could result in a large number of remote memory references. This is particularly true in the case of tasks. A task that is declared in the scope of another task is a child of the enclosing task. These tasks share the stack of the parent task, and the child task's local stack becomes a branch of the *cactus stack*. If tasks are freely distributed or allowed to migrate throughout the system, references to portions of the parent's stack space may be to remote memory. In order to avoid this situation, we have proposed only the above stated units of distribution, along with tasks that are dependent only on library packages, not other tasks [5]. While this policy for distribution simplifies the run-time system and prevents the programmer from many subtle pitfalls it nevertheless restricts the expression of parallelism in the program and makes load balancing very difficult.

Task Termination Task creation and termination in Ada is synchronized with the parent or creating task. In the case of termination, tasks may wait in a loop to receive communication or a signal to terminate. Tasks waiting in these loops terminate if their parent is completed and all other sib-

ling tasks are waiting in this loop. Support for this style of termination is not too difficult to implement given the above restrictions for units of distribution. However, when all tasks are allowed to be distributed, the problem becomes much more complex, and overhead is greatly increased [5].

Time The Ada language provides a predefined function called CLOCK that returns the current point in time. A way to provide this value quickly and accurately to all processing elements in a DMM is a difficult problem. One approach is to replicate the CLOCK function on each processor, and have it read a local timer when called. But this requires that each processor have a timer that is synchronized with all others in the system. Alternatively, a central timer could be provided so that a call to CLOCK results in a remote procedure call to read this timer. However, the accuracy of the returned value would be in question, due to the variable overhead of making a remote procedure call over an unknown number of communication links. In general, providing an accurate CLOCK function in a DMM is complicated, particularly if the hardware has not been designed to support it. This is usually the case, unless the system was designed explicitly for a real-time application.

2.3 Basic Language Features

Because of the problems we encountered in targeting Ada to a hypercube architecture, we have studied alternative approaches to specifying a parallel language that is suitable for execution on a DMM. These alternatives are based on previous work done in the development of other concurrent languages and languages for distributed systems. Examples of concurrent languages include Concurrent C [8], Occam [15], and Mesa [14]. Examples of distributed languages include Lynx [18], Network Implementation Language (NIL) [17], and Synchronizing Resources (SR) [1]. Overall, we found that concurrent languages tend to be designed for implementing operating systems or other low-level applications. Their model of parallelism tends to favor a shared-memory environment, or a particular architecture in the case of Occam. Distributed languages also tend to be designed for use in programming operating systems. While these languages reflect a distributed-memory model, they often support techniques that allow for dynamic loading and linking of modules to executing programs. While the distributed-memory model is appropriate for programming DMMs, the additional features are not needed for general-purpose parallel programming on DMMs.

The process model we propose adding to a sequential language is based on our experience with Ada and a consideration of these alternative approaches. Because there are several features of both Ada and Concurrent C that do seem appropriate, our proposed process model is very similar to that of Concurrent C. We discuss the process model of our proposal below.

3 The Parallel Language

As stated above, our model of a parallel programming language for a DMM is one where multiple threads of execution are available, each with its own data area. We provide this capability by adding a process model to a sequential language. The language used initially to provide our process model is C [13]. Because of its simplicity, C has many desirable features that fit in well with our proposals. The parallel features added to C are intended to be straightforward and require only basic run-time support. The following is a summary of language features:

- Process model concurrency.
- Dynamic and static process creation and scheduling.
- Global data sharing.
- Support for synchronization and communication.

In addition to the features provided, there are other primitives which are intentionally omitted from the language. They are:

- Shared data is **not** protected from concurrent access.
- **No** lock variables or semaphores are provided.
- **No** remote procedure call facility is provided.
- Timing is **not** supported.

The language features are described in the following paragraphs together with the reasons for the above omissions.

Process Scope and Visibility Function declarations in C may not be nested. We apply this restriction to processes as well. Process dependencies are determined by create and merge points, not by declaration. The scope and visibility rules we use are the same as they are for C. Global variables are declared before the beginning of the `main` function or are visible in other files using the `extern` declaration. Variables declared within a process are not visible outside the process. Variables may be visible to a subset of processes if all are declared in a single file, and the variables use the `static` declaration. Processes themselves may also be declared as `static`, which restricts their visibility to the file where they are declared and prevents processes declared in other files from creating instances of them.

Processes must be reentrant, so that multiple instantiations of them may all execute simultaneously. All variables local to a process are allocated on a local stack. These variables are then deallocated when the process completes, just as in the case of a return from a function call. `Static` variables are not allowed in processes, they must instead be declared as described above. Besides its local variables, a process can only access parameters, global variables, and visible static variables. Data that is to be shared by subsets of processes is controlled by the programmer by using parameters or static variables. If access to shared static or global data must be synchronized, the **create** and **merge** primitives must be used.

Process Declaration Processes are declared at the beginning of the program before the **main** function. The process declaration specifies a type, a mode, input and output parameters, and code. The type identifier is used in **create** statements to create an instance of a declared process. The mode specifies whether process termination is *synchronous* or *asynchronous*. Synchronous processes may return data upon completion, and must *merge* with the process that creates it. Asynchronous processes do not return data, and simply run to completion. Parameters may also be passed into processes upon creation and parameter semantics are by copy. The code of a process is basically the same as a function, but may also contain interprocess communication statements (described below). Synchronous processes must also contain **complete** a statement that specify values or variables to be used for output parameters. Asynchronous processes also contain **complete** statements, but without data. The **complete** statement is necessary in the case of asynchronous processes, because the main program may not terminate until all of its processes have completed. An example of process declaration appears in Fig. 1.

```
process synch proc(data_item)
      returns (result)
            /* Parm type declaration. */
int data_item, result;
{           /* Begin code.            */
   int i;   /* Declare locals here.   */
. . .
   complete(result);
}
```

Figure 1: Process declaration.

Process Creation Processes may be created statically or dynamically via the **create** statement. Static creation is possible when the **create** statement occurs before the **main** function in the global declaration area, and the parameters passed into the process are known at compile time. An implementation may create these processes at run-time before the execution of the main program begins, and, in this case, parameters passed to the process may be variables. Dynamic creation occurs when a **create** statement is executed at some point in the main program or in some other process. The process creating another process is suspended until the creation is complete. The **create** statement may indicate an integer processor identifier that specifies the initial location of the created process. This is referred to as a *location clause*, and it uses the keywords **on node.** The **create** statement returns a *process identifier*, which can be used for process communication and termination. A variable must be declared of the predefined type **pid**, and specified in the **create** statement to hold the process identifier.

The **create** statement also may be used to create groups of processes at once. The created processes may receive exact copies of the parameters passed in, or they may receive separate values from an array. If an array is specified as an input parameter and has a $*$ in any of its subscript fields, a unique value of that subscript's range is provided for each created process. If more than one $*$ appears, the subscripts are sequenced in row-major form, i.e. with the right-most index varying first. The input parameters of the process are declared to be of the same type as the component type of the array designated in the **create**. The **create** statement returns an array of process identifiers in the case where a group of processes are created. Figure 2 shows an example of process creation.

```
int data[N], datum;    /* N is a constant */
pid procs[N], oneproc;
. . .
procs[*] = create [N] proc(data[*]);
 /* Each index of procs gets a unique pid.
  * Each process gets a unique value from
  * data. N may be replaced by a variable.
  * A single process is created for node 2
  * by the following:
  */
oneproc = create proc(datum) on node 2;
```

Figure 2: Process creation.

Process Termination There are several types of termination possible for processes. Synchronous processes terminate when they execute the **complete** statement. The creating process may then complete an execution of a **merge** statment that optionally reads values of output parameters from the completed process. Groups of processes may also be specified in single **merge** statement, using the $*$ notation in a manner similar to that described above. An example of the syntax for **merge** is given in Fig. 3.

Asynchronous process also execute a **complete** statement as their last statement, but no output parameters are specified and the creating process does not merge with the completed asynchronous process. The completion of asynchronous processes is necessary to allow the entire program to terminate. Thus, in a sense, the last statement of the main program can be thought of as a merge with all asynchronous processes.

```
int results[N], result;
. . .
      /* A group merge. */
merge procs[*](results[*]);
. . .
      /* A single merge. */
merge oneproc(result);
```

Figure 3: Process merging.

Processes may also be forcefully terminated using the **kill** statement. Any process including the main program may kill any other process for which it has a valid process identifier. Process groups may also be killed in a single statement using the $*$ notation.

Process Communication In addition parameter passing between processes, primitives for explicit interprocess communication are provided. This allows processes to exchange data while running, and avoids the need for a large number of process creations and terminations. Asynchronous communication is supported using basic send and receive primitives. A rendezvous mechanism is also proposed to support synchronous communication. It is very similar to the ones provided in Concurrent C and Ada [9], except that the complex timed and conditional forms are not included.

The rendezvous mechanism is supported with **call** and **accept** statements. A **call** statement names a process identifier, entry point for the rendezvous, and parameters to be passed in and out. The output parameters returned from the accepting process must be specified in a **receive** clause in the **call** statement. A sequence of statements may be specified in a **call** statement after the reserved word **else**. These statements are executed if the rendezvous is not completed, either because of an invalid process identifier or because the called process has terminated. The calling process is suspended at the point of the call, and does not resume until the output parameters are provided by the accepting process.

The **accept** statement specifies the entry point, the parameters, and a sequence of statements to be executed while the two processes are synchronized. This sequence of statements must end with a **send** clause that returns the output parameters to the caller. Execution is suspended if no calls are pending at the **accept**. The **accept** statement may be embedded along with several others in a **select** statement, which nondeterministically selects one of the **accept** statements for execution. A **select** statement may specify a sequence of statements following the reserved word **else** that are executed if no calls are pending on any of the **accept** statements. Boolean conditions, known as *guards*, may be specified before each **accept** statement. No **accept** is attempted unless its guard is true.

Asynchronous communication is supported with **send** and **receive** primitives. The **send** statement is similar to a **call**, but there are no output parameters. Execution of the sending process continues immediately after the **send**. A broadcast function can also be used in conjunction with **send**. In this case, the target of the **send** is specified using an array for a process group with the * notation.

The **receive** statement is identical to the **accept**. Guards may be used, and several **receive** statements may be embedded in a **select** statement with an **else** part. The sequence of statements is specified in a **receive** just as it is in an **accept**, since these statements are not executed if the guard is false or an alternative **receive** is executed. Both calls and sends are enqueued in FIFO order at their respective entry points.

Example syntax for all interprocess communication statements is shown in Fig. 4.

```
  /* Basic call statement. */
call oneproc.deposit(data_in)
    receive(data_out);
    else
        {
         . . .
        }
. . .
  /* The following is a broadcast. */
send procs[*].mailbox(message);
. . .
  /* Accepts embedded in a select. */
select
    accept deposit(data1)
                  {
                   . . .
                   send(data2);
                  }
    when flag then accept withdraw()
                  {
                   . . .
                   send(data3);
                  }
    else
        {
         . . .
        }
end select;
. . .
  /* The basic receive statement. */
receive mailbox(message);
```

Figure 4: Process communication.

Shared and Private Data As stated above, processes may share global variables or visible static variables. It must be recognized, though, that these references may be remote. However, these variables may be placed on specific processors using a location specification that is similar to the one used in a **create** statement, but the location integer specified must be a constant. This feature is provided since it may be of use in some algorithms. For example, using the location specification in a **create** statement, a group of processes may be created to run on a single node. These processes may then share a static variable that is located on that processor.

While sharing of variables is supported to a certain extent, the language encourages sharing through communication calls and parameter passing. This reflects an underlying assumption that memory is plentiful in a DMM. If this is indeed the case, multiple copies of both code and data in multiple processor memories should not cause any problems, and may increase efficiency if it enables remote references to be avoided.

Omitted Features Several features that are common in languages for distributed and parallel computing have been omitted from our proposal, as stated above. These features are semaphores, shared data protection, remote procedure calls, and explicit timing support. The first three features are not needed, since the ones provided are sufficient to implement these operations. Timing is not supported, since the language is not intended for real-time systems.

Synchronization is provided in the language through pro-

cess creation, termination, and communication. Mutual exclusion can be coded using processes that protect shared resources, and provide access through rendezvous. This style of protection can also be used to protect shared variables from concurrent access. The omission of semaphores follows the advice of the Ada language designers [10], who state that semaphores lead to a condition of "distributed critical sections," where accesses to shared resources are not centralized within a process. The other feature omitted from our proposal is the remote procedure call [3]. This operation is not necessary, since communication with remote processes is possible, and these processes may be bound to a processor if necessary. It is our philosophy that procedures (or functions) should be reentrant and shared amongst all processes to which they are visible. Multiple calls to one procedure may then execute simultaneously. Instead of using procedures to manipulate shared data or create other side effects, processes should be used.

4 The Run-Time System

The run-time system for our parallel language must support the execution of processes. The process is the unit of distribution in this language. Because processes may share data only through the use of globals or parameters, there is no cactus stack structure. This greatly simplifies the support for distributed processes, and avoids the problems encountered with distributing Ada tasks and packages. The major components of the run-time system for our parallel language are described briefly in the paragraphs below. Where similarities to the run-time support for Ada occur, additional detail can be found in [5, 6, 7].

Run-Time Kernel The run-time system is organized as a kernel which executes on each processor in the DMM. The kernel provides support for scheduling processes, creating and terminating processes, interprocess communication, and remote object reference. In order to provide these services, the kernel must interface with the operating system support for message passing, and use assembly code for some low-level routines.

Processes are managed by the kernel using process control blocks (PCBs). A PCB is a contiguous block of memory that contains a process' state, stack space, identifier, and pointers to entry queues for communication (see Fig. 5). The state of the process indicates its execution status in addition to its register values and other process state information. The stack space is used to allocate local variables and frames for function calls. The process identifier is simply an integer, but when other processes use process identifiers defined at the language level, they are records made up of a unique integer value and the node number where the process is executing. The entry queue pointers are to blocks of memory assembled into a linked list. Each entry in the list is referred to as a *call record*, and contains data for an execution of either a **send** or **call** operation, depending on the type of entry point (either a **receive** or **accept**). When the process with the entry queue executes a **receive** or **accept**, the first entry on the appropriate queue is consumed and the queue is updated.

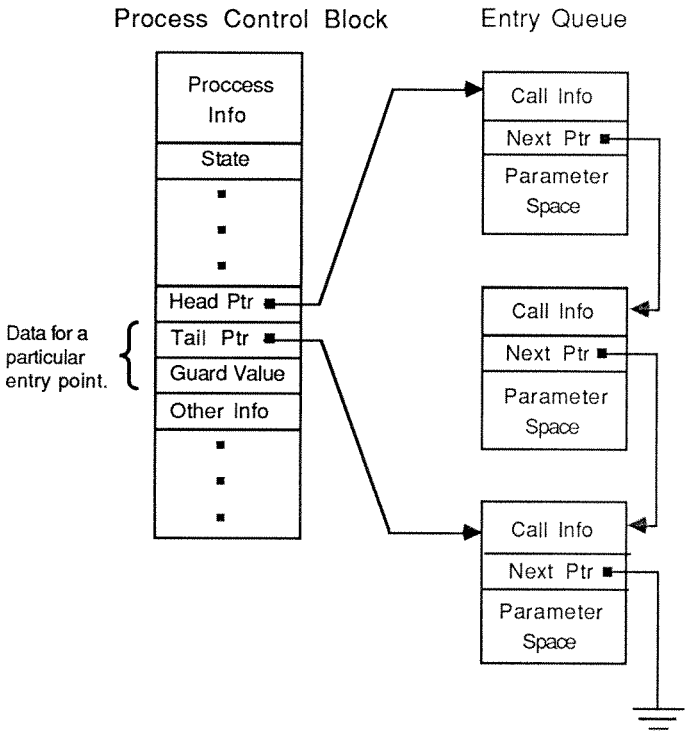

Figure 5: Process control block and entry queue.

Scheduling is supported by maintaining a queue of runnable processes. Time slicing of processes is easily supported using an interval timer [5]. Scheduling points occur at time-slice interrupts and process state changes. These state changes result from traps into the run-time system to request interprocess communication, process creation, or process termination.

Process Creation and Termination As stated above, process creation can be static or dynamic. Static creation can be viewed as an optimization, and can occur if the **create** statement is in the global declaration area, and the parameters for it are all statically known. Otherwise, process creation occurs dynamically when it is encountered during the execution of the program. We require that each processor retain a copy of the code for any process which may be executed on it.[1] Creation is performed by the run-time system by allocating and initializing a process control block. The generated code for each process is similar to that of a procedure, and the process allocates and initializes its own local variables on its stack space. If a process is to be created on a remote node, the run-time system sends a message containing the input parameters to the node where the creation is to take place. When the creation is complete, a message containing the process identifier is returned to the creating node.

Termination occurs either when processes execute **complete** statements or when they are killed. A **complete** state-

[1] As an optimization, analysis of the program may be performed at compile time to reduce the number of code copies needed.

ment informs the creating process of the completion and provides any output parameters. This may involve sending a message to the processor where the creating process is executing. The creating process consumes the output parameters when it executes a **merge** statement. If the merge point is reached before the named process has completed, the creating process blocks and waits for the completion data to be provided. In the case of asynchronous processes, no data is returned upon completion, but a message is still sent. This enables the run-time system on a given processor to record when all of the processes that it has created have terminated. This is necessary for determining when the entire program has completed. Kill statements change the state of a process so that it is no longer runnable. An acknowledgement of a kill is made available to the creating process, so it can tally the termination. The state of any returned parameters is undefined, so kills should be used with caution. Killing a terminated process has no effect.

Interprocess Communication The support for interprocess communication is straightforward, based on our experience with implementing the Ada rendezvous mechanism on a hypercube [6]. The rendezvous mechanism proposed in our language is simpler, since there is no support for timed or conditional calls. The asynchronous communication primitives are also easily implemented. If the send or call is to a locally located process, the entry queues of the receiving or accepting processes are manipulated by the run-time system to include call records. If the send or call is to a remote process, the call record is formed and copied into a message that is sent by the operating system. When the message is received by the run-time system on the remote processor, the call record is copied from the message buffer and inserted into the entry queue. In the case of a rendezvous, the calling process blocks and awaits a reply. The accepting process executes any statements specified as part of the rendezvous, and then returns the output parameters, either through the original call record or via another message. The run-time system then copies the parameters into the stack of the caller or provides a pointer to the results. The run-time system requires storage for the allocation of call records. In the case of calls local to a processor, the call record may be allocated off the caller's stack.

In the simple case of a **receive** or **accept**, a process enters the run-time system when the statement is executed. The run-time system attempts to read data from the appropriate entry queue and make the call record available to the process. If there is no data available, the process waits until a send or call arrives. In the general case, the **receive** or **accept** is embedded in a **select** statement and/or may be guarded. In this case, the process must indicate to the run-time system which subset of entries it is willing to accept data from, and whether or not the process should block if no data is available. This can be accomplished by manipulating a data structure contained in the process control block. This structure contains the addresses of the code associated with each entry point and the guard values.

Remote Object Reference Any reference to a global or static variable is potentially a remote reference. The process must enter the run-time system at the point of a global variable reference. Because these variables must be bound to processors statically, the process knows which processor the global variable resides on. The run-time system uses this information to send a message to the appropriate processor, and includes data if the access is a write. The process performing the global variable access is blocked until an acknowledgement is received from the processor performing the access. The acknowledgement contains the value of the accessed object and/or an indication of an error condition.

The use of pointers as globals and the sharing of pointers between processes is allowed but not recommended. It is up to the programmer to guarantee that a pointer refers to a local object. If it does not, the result of a dereference is undefined.

Because of the expense involved in accessing globals without the guarantee of mutual exclusion, it is better for the programmer to control the sharing of data through processes and the rendezvous mechanism. The cost is roughly the same in both cases, but the rendezvous ensures mutual exclusion. The only exception to this rule is when it can be determined statically that the variable and the process accessing it will reside on the same processor. In this case, the variable may be accessed directly without entering the run-time system. However, this optimization causes problems when we consider the possibility of process migration.

Process Migration The description of the language and run-time system to this point has assumed that running processes do not migrate. Instead, load balancing is achieved by creating processes on processors with low loads. This should be sufficient for most applications. However, migration of processes after creation may be desirable in some cases. This, of course, requires more effort from the run-time system. As long as code for a process is available on another processor, migration involves the copying of the process control block and entry queues from one processor to another. We require that all code be read-only, so that this strategy will work correctly. Also, this strategy requires that all data accesses within a process are relative to the stack pointer. This may require that processes be allowed to migrate only at specific point in their execution. The run-time system must also reconstruct all call records into the proper entry queues. The migration of a process may be expensive, and should only occur when a net increase in efficiency can be guaranteed. To reduce the cost, only processes with empty entry queues should be moved.

The movement of a process will have some side effects, including the invalidation of its process identifier. The run-time system will have to forward messages for processes that have relocated, and normal message contents will have to be altered so that stale process identifiers can be updated when encountered. Migration of processes may also invalidate assumptions regarding the locality of objects referenced by pointers. When migration is allowed, all references to global variables must be assumed to be remote. Pointers

should only be used within processes to refer to local data, and should be relative to the stack pointer.

5 Status and Conclusions

At this point, the run-time support for our proposed parallel language is being developed using our Ada run-time kernel as a base. The run-time support is simpler than that for Ada, due to the fact that our language provides a more direct reflection of the underlying environment. The main differences in the run-time support for our proposed language are summarized as follows:

- The cactus stack is not present in our proposal, so the arbitrary distribution of processes causes no problems.
- Timing support is completely omitted.
- Remote object references may only occur in the case of globally shared or static variables. There is no remote procedure call.
- Migration is still somewhat difficult, but may be implemented without violating any rules regarding units of distribution.

In summary, the language described here is one suitable for expressing general-purpose parallel programs for execution on distributed-memory multiprocessors. A level of machine independence has been provided, which allows software to be developed independently of the number of processors available. However, location clauses may be used to take advantage of optimal distribution strategies.

References

[1] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilson, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[2] R. G. Babb II, ed. *Programming Parallel Processors.* Addison-Wesley, Reading, Mass., 1988.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] G. D. Buzzard. *High Performance Communications for Hypercube Multiprocessors.* PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, 1988.

[5] R. M. Clapp and T. Mudge. Ada on a hypercube. In *Proceedings of The Third Conference on Hypercube Concurrent Computers and Applications*, pages 399–408, Pasadena, CA, January 1988.

[6] R. M. Clapp and T. N. Mudge. Distributed Ada on a loosely coupled multiprocessor. Technical Report RSD-TR-3-88, Robotics Research Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, January 1988.

[7] R. M. Clapp, T. N. Mudge, and R. A. Volz. Distributed run-time support for ada on the ncube hypercube multiprocessor. Technical Report RSD-TR-10-87, Robotics Research Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, August 1987.

[8] N. H. Gehani and W. D. Roome. Concurrent C. *Software Practice and Experience*, 16(9):821–844, September 1986.

[9] N. H. Gehani and W. D. Roome. Rendezvous facilities: Concurrent C and the Ada language. *IEEE Transactions on Software Engineering*, SE-14(11):1546–1553, November 1988.

[10] J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Breckner, O. Roubine, and B. A. Wichmann. Rationale for the design of the Ada programming language. *ACM SIGPLAN Notices*, 14(6), June 1979.

[11] A. H. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[12] A. H. Karp and R. G. Babb. A comparison of 12 parallel FORTRAN dialects. *IEEE Software*, pages 52–67, September 1988.

[13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[14] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[15] D. May. Occam. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[16] T. N. Mudge. Units of distribution for distributed Ada. In *Proceedings of the International Workshop on Real-time Ada Issues*, pages 64–66, Moretonhampstead, UK, May 1987.

[17] F. N. Parr and R. E. Strom. NIL: A high level language for distributed systems programming. *IBM Systems Journal*, 22(1 and 2), April 1983.

[18] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[19] R. A. Volz, T. N. Mudge, G. D. Buzzard, and P. Krishnan. Translation and execution of distributed Ada programs: Is it still Ada? *IEEE Transactions on Software Engineering*, SE-15(3):281–292, March 1989.