Distributed Ada

on a

Loosely Coupled Multiprocessor

Russell M. Clapp
Trevor Mudge

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan

January 1988

Center for Research on Integrated Manufacturing

Robot Systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109-2110

ensp

UMR1196

Abstract

This paper examines the design and implementation of run-time support for distributed Ada on a loosely coupled multiprocessor. First, loosely coupled architectures are defined and some examples given. The programming model associated with such multiprocessors, particularly large scale ones, is then examined. Discussion and rationale for suitable program units of distribution is addressed for this particular style of architecture. The requirements of the run-time support are then presented together with discussions of their relationship to a loosely coupled architecture. An overview is then given for a subset of the run-time system that has been implemented on a large scale loosely coupled multiprocessor available to us – an NCUBE hypercube machine. Prospects for expanding the support to a full distributed Ada run-time system are also included as well as concluding remarks on the general problem of putting parallel programming languages on large scale loosely coupled multiprocessors.

Contents

1 Introduction

It is apparent from statements in the Ada Language Reference Manual (LRM) [LRM83] that Ada is not just intended for execution on shared memory multiprocessors but also on loosely coupled processors that do not share memory. Accordingly, the LRM provides some explanations of language constructs that consider the distributed program case. Unfortunately, the semantics are not clear for the distributed case. As a result, there have been several investigations into the issues and feasibility of distributing Ada. The results of these investigations can be found in [VMB88], [VMN85], [Ard84], and [Cor84a]. Related discussions of run-time support for distributed Ada can be found in [Wea84a], [Wea84b], [FiW86], [Ros87], and [Ard86].

The advent of inexpensive commercial multiprocessors has made loosely coupled systems an attractive alternative to monolithic supercomputers. Their design and operation raises several issues, especially with respect to the Ada programming model and specification of language units for distribution over multiple processors. In order to provide some insight into the approach and feasibility of targeting Ada to a distributed memory multiprocessor, we have implemented a partial run-time system for distributed Ada targeted to an NCUBE/ten. This hypercube machine is configured here with 64 processors and serves as a model for a generic loosely coupled multiprocessor. The kernel implemented runs together with the existing operating system and includes support for executing Ada tasks in parallel. In order to expand this core run-time system, the issues concerning the Ada programming model and the language units of distribution need to be addressed.

This paper is organized as follows. First, a definition of what we mean by a loosely coupled multiprocessor architecture is given as well as a discussion of programming paradigms that are currently preferred by their users. Issues concerning this particular type of target architecture will also be discused, especially their interaction with the shared memory programming model of the Ada language. Following that, examination of potential language units of distribution is presented, and a proposal of suitable units of distribution is made based on their impact on the run-time system. The components of the distributed run-time system will then be presented with an implementation approach given for each. An overview of a subset of this system that has been implemented is also given. Finally, consideration is given to the expansion of the run-time system and to the feasibility of retargeting an Ada compiler to our hypercube machine.

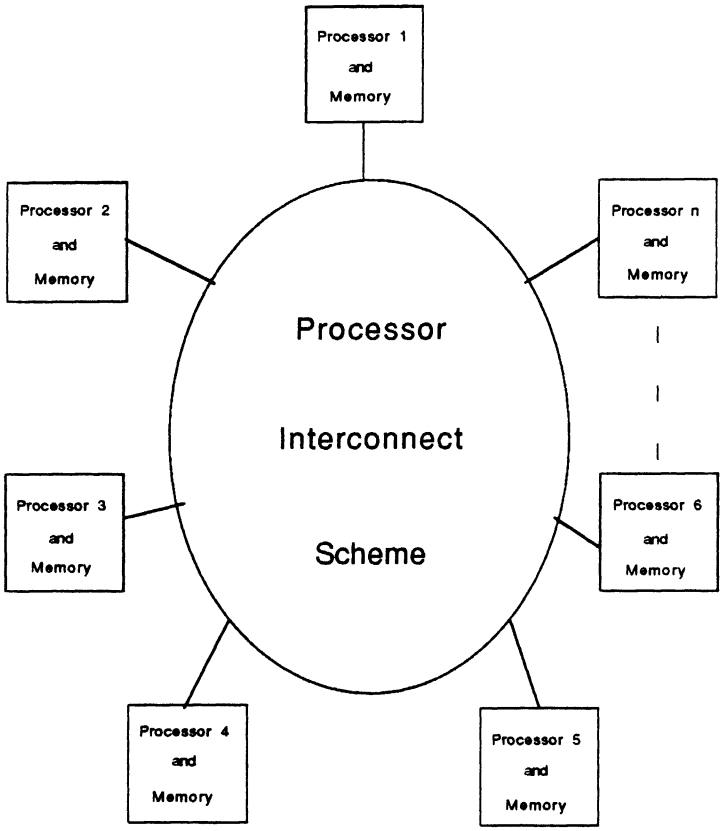

Figure 1: Generic loosely coupled multiprocessor.

2 Loosely Coupled Multiprocessors

The term *loosely coupled* is one commonly used to refer to multiprocessors where there is no shared memory, i.e., each processor of the computer has its own private store. This type of multiprocessor has also been called a *distributed memory multiprocessor.* Multiprocessors which have memory units that can be accessed by all processors are referred to as *tightly coupled.* Figure 1 shows a diagram of a generic loosely coupled multiprocessor.

Since loosely coupled processors share no memory, there is more freedom in designing their interconnect scheme. Tightly coupled multiprocessors must provide a path from each processor to the shared memory, which is usually done with a shared bus or a crossbar switch. There are physical limits to the number of processors that can be connected this way, as well as diminishing returns on throughput as more processors are added due to congestion at the shared memory. For loosely coupled multiprocessors, any interconnect scheme that provides a path from one processor to any other is allowable. Since these paths may pass through

intermediate nodes, it is possible to configure a loosely coupled multiprocessor with a very large number of processors. Thus, it is often the case that distributed memory machines are large scale multiprocessors.

Because there is no shared memory, loosely coupled multiprocessors communicate between nodes by sending messages. Each processor runs a message passing communications kernel along with the code it is executing in addition to any other run-time support that is needed. It is possible for the communications kernel to be a *store and forward* one. This allows intermediate processors on a communications path to read messages from a directly connected processor and then forward them to another connected processor in the direction of the ultimate destination.

In some cases, loosely coupled machines are made up of heterogeneous processors. Heterogeneous systems usually have peripherals private to each processor as well as memory. In these cases, the system is referred to as a *multicomputer* instead of a multiprocessor. The scope of this paper, though, is limited to consideration of loosely coupled homogeneous multiprocessors. These machines are normally used in a manner where the multiple processors cooperate in solving a single problem. A closer look at one such architecture, the one used for an initial implementation here, is given below.

2.1 The Hypercube Architecture

A hypercube computer consists of several microprocessors interconnected by communication links. A hypercube of dimension n is a multiprocessor with $N = 2^n$ processors. Each processor is directly connected via communication links to n neighbors. An operating system kernel supports store and forwarding of messages so that processors that are not neighbors can still send messages to each other through intermediate processors. The maximum number of distinct communication links that must be used to transfer a message between any two processors is n, the dimension of the hypercube.

In the case of the NCUBE machine, each processor has its own local memory (128 K-bytes) and the hypercube array is managed by a host computer (an Intel 80286 based system). The configuration used in our work at the University of Michigan is an NCUBE/ten which is partially configured with 64 processors but has the capacity to hold 1024 processors. The processors are Vax-like 32-bit microprocessors with IEEE standard floating point capability. Each processor runs an operating system to support communications called Vertex, as noted above. The host runs a multi-user Unix-like operating system called Axis. Axis allows the hypercube array to be partitioned into subcubes that may be allocated to different users. More details on the NCUBE architecture can be found in [HMS86] and more information on Vertex can be found in [MBA86].

Considering the issues associated with an implementation of Ada on a hypercube multiprocessor subsumes those relevant to an Ada implementation on some other architectures. For instance, the hypercube can be used as a model for a generic system of loosely coupled homogeneous processors. The number of processors used in the hypercube can be varied, and the kernel operating system shields the run-time system from details of the processor interconnect scheme. Considering the complete hypercube simply adds the issue of how to effectively use a large number of processors with a distributed Ada language. The various programming issues are considered below.

3 Programming Model

3.1 Single Code Paradigm

The programming of large scale multiprocessors usually involves writing a single program and running an exact copy of it on each node in the system. The program may have data dependent branches that result in different nodes executing different parts of the program at different times. The data affecting the execution path of a node may come from messages passed between nodes and the location of the processor in the network. Such a programming style has been referred to as *single code multiple data* (SCMD) [Buz88]. Sometimes several different programs are written to run on the various nodes. This style is referred to as *multiple code multiple data* (MCMD). However, in the MCMD case, it is unlikely that a large number of distinct programs will be written. What is more apt to occur, is that the number of distinct programs written will be small, and the number of copies of each program running on different nodes will be high in order to exploit the large number of processors available. In this case, MCMD begins to appear to be more like SCMD. In any case, the communication between nodes in these programming styles involves use of low-level operating system primitives and the overall level of cohesion in the system is low since each processor is executing its own complete program. This style of programming can be difficult to debug and requires considerable coding effort to interface the several programs.

3.2 Distributed Language

There are several advantages to a distributed language for parallel programming of a loosely coupled multiprocessor. They are summarized in the following points.

- **Strong Type Checking:** Implementing software for a multiprocessor as a single program in a distributed language allows for extensive compile time type checking. This

checking includes user defined types and type checking across the boundaries of distributed program units, greatly reducing run-time errors.

- **Inter-task Communication:** Inter-task communication across nodes in a distributed system is simplified with a distributed language. Programmers are relieved of packing and unpacking message buffers and performing type coercions on raw data. The predefined communication mechanism of the language is used instead, effectively transferring the low-level details of communication to the run-time system. This also has the advantage of making inter- and intra-nodal task communication appear the same to the programmer. In the case of Ada, the rendezvous task communication scheme is synchronous, and can be implemented on a system with asynchronous communication primitives in a straightforward manner (as we will show in Sec. 5).

- **Shared Memory Model:** In the case of Ada, the scope and visibility rules provide for a shared memory model. Executable units can access variables and subprograms imported by a **with** clause on a package. Also, tasks can access these items in the address space of their ancestor tasks. To implement the memory model of Ada, support for shared memory must be included in the run-time system, even when the hardware has no shared memory.

  The run-time system, then, must provide a mechanism for remote variable access and remote subprogram calls. In the case of referencing items in packages, the fact that they are remote can be determined at compile time if the package contents are bound to a node statically. Task references are more complex, as they include variables in a task's address space as well as its entries. By carefully specifying the language units of distribution, it is possible to eliminate remote references to a task's variable space (discussed in Sec. 4). As for a task's entries, it is necessary to incorporate a processor id into all task references, so that the run-time system can immediately determine where messages concerning entries are to be sent. Approaches for implementing run-time support for the Ada shared memory model are outlined in Sec. 5.

  Supplying run-time support for the shared memory model allows existing Ada programs to be compiled and run without modification on multiprocessors without shared memory. Although careful consideration of the distributed target may provide for more efficient programs, the shared memory support does allow a programmer to write programs as he or she would in the uniprocessor case.

- **SCMD Support:** In order not to limit the programming options with distributed Ada on a large scale multiprocessor, it is necessary that the distributed language provide support for the SCMD programming style. Including support for SCMD allows many existing algorithms to be programmed in distributed Ada and to benefit from the above mentioned advantages of a distributed language.

The hypercube architecture is appropriate for programs with a large number of tasks, since a large number of processors are available. An example of such a program appears in [CMV86], where a parallel solution to the n Queens problem is presented. This solution requires n^2 identical tasks, one for each square on the chess board. In the case where $n = 8$, the full configuration of our NCUBE/ten could be used with one task running on each processor, with the main program unit sharing one processor with one of the tasks. This example demonstrates how Ada's ability to replicate tasks of a specific task type can support the SCMD programming style.

The notion of running different tasks of the same program on different processors raises the issue of distributable program units. This is discussed in the following section.

4 Language Units of Distribution

4.1 Background

The choice of which language units to distribute is an important one and a decision that greatly impacts the form and function of a resulting distributed system. In spite of this fact, it is often ignored or under emphasized in much of the literature concerning distributed Ada systems. However, there are some papers that address this issue, most notably [Cor84b], [VMB88], [Jes82].

Various approaches to distributing Ada program units are surveyed in [Cor84b]. These approaches to distribution range from writing separate programs for each processor to allowing any construct of a single source program to be distributed. Also, extending the Ada language to make it more appropriate for distribution is considered.

Intuitively, it seems desirable to limit the number of distributable units in an Ada program. This has the benefit of reducing the demands on the run-time system. On the other hand, allowing only separate programs to be written for each processor has all the disadvantages associated with the current state of SCMD programming. In [VMB88], a proposal is made to limit the units of distribution to library packages and library subprograms. The reference demonstrates how such limitations can greatly reduce the complexity of the run-time system.

4.2 Target Dependencies

The most effective uses of a loosely coupled multiprocessor involve running programs that require the use of a large number of processors. In order to support this situation with

distributed Ada, the units of distribution must allow many packages, subprograms, and tasks (of the same and different types) to reside on the many processors available. Since the number of processors may be quite large and the SCMD programming style is to be supported, it is also necessary to have provisions for loading identical code on several processors.

These goals are somewhat difficult to meet if the units of distribution are limited to library subprograms and library packages. With these distributable units, the only way to replicate code for many different processors is to use a generic package, or to define a unique package containing the same code for each node used. This creates an inconvenience, in that each instantiation of a generic package and each uniquely defined package has its own distinct name. If a memory object or task in each package instantiation is to be referenced, a different name may have to be used in each reference. This precludes the use of a loop, and is a definite annoyance if the number of unique packages, both generic instantiations and otherwise, is large. It is possible, however, to have an array of system wide pointers so that dynamically created tasks and objects can reside on remote processors and be referenced in a loop. This scheme, though, does not prevent the need for a distinct package for each processor, and requires the task or object to be dynamically allocated.

One way to avoid this problem is to extend Ada to include a package type as suggested in [Jes82]. This would allow packages to be referenced by pointers or array references instead of names. This would require, however, that arrays of packages be distributable.

4.3 Proposal

The units of distribution proposed here are based on the above discussion and a desire to comply with the Ada standard. Therefore, the language units of distribution proposed are library subprograms, library packages and tasks, where a task's specification is declared in a package specification. It is also proposed that arrays be distributable whenever the array index references a task object or an access variable that points to a task object, as long as the task or task type specification and the array are declared in a package specification. This approach follows the advice of [VMB88], but also allows tasks to be distributed. This is done so that the multiple copies of identical code requirement can be met with simple naming, while also allowing statically allocated tasks to be distributed without requiring a unique package for each processor.

4.4 Ramifications

In choosing these language units for distribution, the question arises as to what extra requirements are placed onto the run-time system above and beyond those imposed by the library

subprograms and library packages approach. In [VMB88], it is shown that their approach to distributable units requires no cross processor dynamic scope management, but the problem of distributed sibling task termination must still be considered. However, using the units of distribution proposed in [VMB88] or the ones proposed here, the distributed task termination problem is greatly simplified. This is discussed in Sec. 5.7 below.

The strategy adopted here for dynamically created tasks is the same as that in [VMB88], that is, the created task is located on the processor performing the allocation. In addition, with tasks allowed as distributable units, it is possible for statically declared tasks to be located on separate processors if their specification appears in a package specification. However, outside of this scenario, it is not possible for parent and child tasks to reside on separate processors. This is due to the restriction that only tasks whose specification is declared in a package specification can be distributed. If any tasks are nested within such a task, their specification must be declared in the body of the parent task, which by language definition must occur in the body of the containing package. Therefore, even though it is possible for tasks to be nested indefinitely within an outer scope, the distribution strategy ensures that all of these tasks will reside on the same node. This simplifies the termination algorithm for these tasks and makes references to variables declared in parent tasks local memory references.

There is one area, though, where the distribution of tasks allows for more remote variable references. This occurs when a task refers to a variable that is declared in a package body. If that task is distributed to a remote processor, it then has the ability to refer to a package variable that is not visible through the package specification. All types of remote references are carried out by the run-time system. The situation described here can potentially increase the number of these references. The strategy for supporting all types of remote references is discussed in Sec. 5.9 below.

In short, it can be said that the language units for distribution proposed here require little more of the run-time system than those proposed in [VMB88]. However, the units proposed here do provide for more flexibility in specifying code to be replicated on a large number of processors as described above.

5 Run-Time System Components

The various components of a distributed Ada run-time system are outlined below along with approaches for their implementation. For more details on the language features supported by this run-time system, the reader is referred to [Bar84] and the LRM.

5.1 Task Elaboration and Activation

The operations of task elaboration and activation can be implemented in two run-time system routines, one for elaboration and one for activation. The elaboration routine changes the state of a task to allow it to execute its declarative part. The activation routine will allow the calling task to become active if all of its children are active. If the task becomes active, a message indicating this is sent to its parent. Otherwise, the task is suspended until it receives messages indicating active status from all of its children. This enforces the parent-child rules regarding elaboration and activation.

5.2 Normal and Abnormal Task Termination

For these run-time operations, two routines are needed. One is for normal termination and the other is for abnormal termination. When a task is finished with its execution, the routine for normal termination is called. The requesting task becomes completed and, if all of its children are in the terminated state, it also becomes terminated. If any of its children are still active, it stays in the completed state and waits for a message from each child indicating that the child is terminated. Whenever a task enters the terminated state, it sends a message to its parent to allow the parent to terminate.

The above approach for supporting completion and termination covers the simple case where a task reaches the end of its executable statements. The case of tasks waiting to terminate at a **terminate** alternative of a **select** statement is more complex to implement, especially for a distributed system with no shared memory. An approach to solving this problem is discussed in Sec. 5.7 below.

When a task wants another task to abort execution, a routine is called to perform the abnormal termination. The aborted task stops any rendezvous or delay in progress, enters the abnormal state and sends an abort message to all of its children. If the task has no children or its children are already terminated, the task enters the terminated state and sends a message indicating this to its parent.

5.3 Task Delay

There are several ways to implement task delay, but the one usually used is dictated by the target architecture. Ideally, the processor would have an interval timer dedicated solely to implementing task delays. The timer could be given a value that it would count down and then interrupt the processor to indicate that the specified delay had passed. Such a timer is used in a similar way to implement time slicing of tasks. If time slicing is to be supported

and only one such timer exists, another strategy must be adopted.

There is only one interval timer on each processor in the hypercube computer used as our initial target [CMV87]. Since time-slicing is to be supported, an alternative implementation of task delay is used. When a task is to delay, an entry for it is inserted into a linked list of task names and expiration times that is sorted in increasing order of time. When a time slice expires or a system call is made, a check is performed to determine if any delays have expired. If so, the appropriate entries are removed from the list and the specified tasks are processed. This method is used to implement time limits on inter-task rendezvous as well. It is possible in this scheme for delays to last longer than specified by the program. This is, however, consistent with the LRM statements regarding task delays.

5.4 Task Synchronization

The Ada language provides for task synchronization through the rendezvous construct. The interface between the tasks involved in a rendezvous looks much like a procedure call. There are variations as to how these calls can be requested and accepted. Run-time system routines to implement this construct are discussed below.

5.4.1 Entry Call

A task wishing to rendezvous with another task executes an entry call. This call specifies a particular entry in the task receiving the call. There are three types of entry calls, the simple entry call, the conditional entry call, and the timed entry call. For a simple entry call, the calling task suspends execution until the entry call is accepted. A timed entry call requests the entry but withdraws its request if the rendezvous cannot be started within the specified duration. If the time bound specified is zero, a conditional entry call is implemented where the rendezvous occurs only if it can be accepted immediately. (Actually, there are problems with considering a zero delay to mean immediate. The check to determine if the entry can be accepted immediately takes time. Also, it may have been the intent of the task to not attempt the entry call based on a zero delay value. For more details on this issue and the interpretation of the Ada Language Maintenance Committee, see [VoM87a]).

The three types of entry calls can be implemented with a straightforward message passing scheme. This is appropriate for loosely coupled processors, since there is no shared memory. In the case of a simple call, a message is sent and the calling task blocks and waits for a reply indicating that the rendezvous is complete (Fig. 2). For a conditional call, a check is made of the request queue for the called entry to see if the rendezvous is immediately possible. If it is, a call message is sent and a reply is awaited as described above. However, if the conditional call is to a remotely located task, a special conditional call message is sent. When

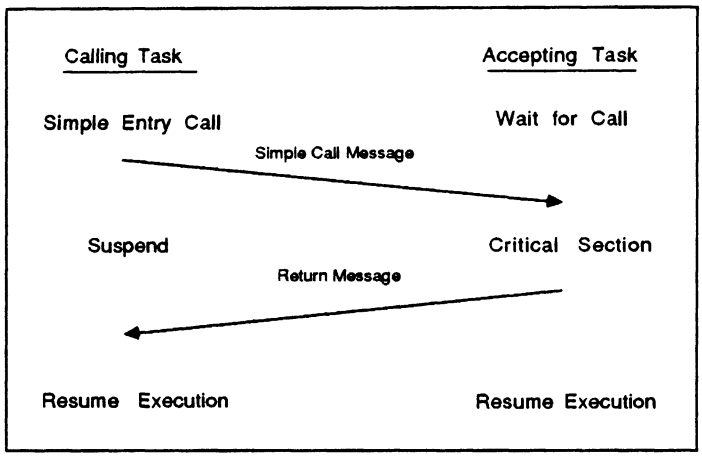

Figure 2: Message passing for simple entry call.

this message is received, a check is made on the request queue as before, and a message is returned to the caller is an immediate rendezvous is not possible. If the rendezvous is possible, a return message is not sent until the rendezvous is completed. In the case of a timed entry call, a call message is sent and a timer node inserted in the list (described in Sec. 5.3 above). When the call message is received, a reply is sent to allow the rendezvous to proceed. If this reply is received before the timeout condition occurs, a message is sent to the serving task to begin the rendezvous (Fig. 3). If a timeout does occur, an abort rendezvous message is sent by the caller (Fig. 4). After a successful rendezvous, a return message is sent to the caller to allow it to resume execution. This scheme of message passing for the timed entry call was originally proposed in [Wea84a] for all types of entry calls. The schemes presented here for simple and conditional calls are an optimization of that general case.

The entry call routine itself performs the actions necessary to initiate the rendezvous. After the first message has been sent, the run-time system dispatches another task that is ready. When the reply messages are received, the run-time system communications kernel processes the rest of the rendezvous based on system data structures and data contained in the messages. Rendezvous parameters are passed in the appropriate messages exchanged between the two tasks involved.

5.4.2 Accept Statement

There are variations of the accept statement just as there are for the entry call. An accept statement may be simple and occur by itself, or it may be embedded in a **select** statement. A simple accept waits indefinitely for a call on that entry, but a **select** statement is used to wait for a number of entries that may be guarded. A **select** may also have a **delay** or

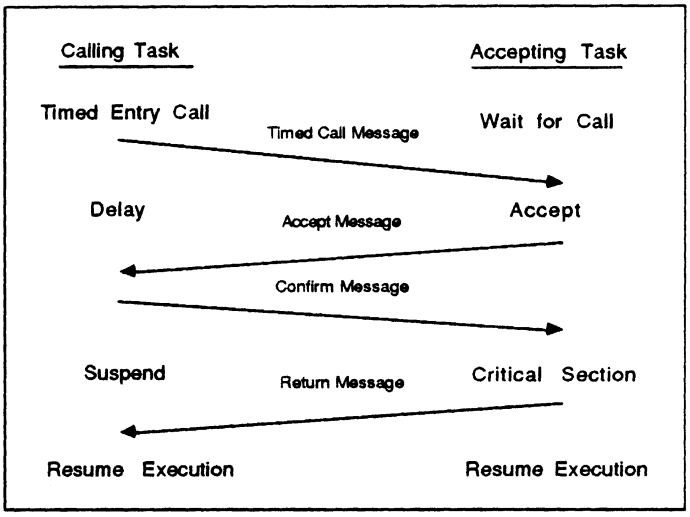

Figure 3: Message passing for successful timed entry call.

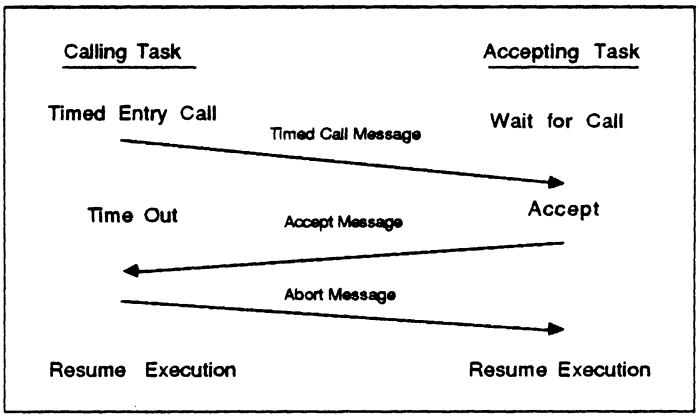

Figure 4: Message passing for unsuccessful timed entry call.

else alternative, which places a time limit on the accept or specifies that the accept must occur immediately. There is also the possibility of a **terminate** alternative in the **select**. The strategy for handling this case is discussed in Sec. 5.7 below.

The accept statement can be handled by the run-time system with two routines, one to begin the rendezvous and one to complete it. To start the rendezvous on the receiving end of the call, the run-time system is given a list of open entries to which calls may be currently accepted. If this list indicates that the timer should be set, the value passed to the run-time system is used in inserting a node into the timer list (as described in Sec. 5.3 above). If a call is pending on an open entry, the next step in the message passing protocol described above is taken. Otherwise, the task is suspended until a call is received or a timeout occurs, unless an **else** alternative occurs in place of a **delay** alternative. In this case, if the rendezvous cannot be started immediately, the code specified in the **else** branch is executed and the task is not blocked.

When the code for the rendezvous is completed, the run-time system is called so that a return message can be sent with parameters to the calling task. The state of the accepting task is also changed so that it no longer is recognized as being in a critical section.

5.5 Dynamic Task Creation

Ada provides for dynamic task creation through the use of access variables that point to objects of a particular task type. There are several possibilities for specifying the processor location of a dynamically created task, and they are discussed in [VMB88]. The recommendation made there is to load the task on the node that executes the allocating statement. This provides the most flexibility, since it allows any number of nodes to allocate tasks of the same type.

In order to support this operation, a copy of the code must be made available to the allocating node. A system wide unique task name must also be determined. Part of this name should indicate which processor the task is executing on. This name can then be the value of the access variable. After the code is loaded, the new task can proceed with its elaboration and activation by calling the routines described above. Once active, the access variable can be formally assigned a value and the thread of control that executed the allocating statement can proceed.

5.6 Task Migration

While task migration may be a desirable feature to support in order to achieve effective load balancing, there are many issues that should be considered, especially in the distributed

memory multiprocessor case. For instance, pointers to a task would have to be updated when a task moves, or a scheme to hunt down the location of a moved task would have to be implemented. (One such scheme is presented in [Ros87]). In either case, additional overhead would be incurred. Another problem with allowing tasks to migrate arbitrarily is that it could violate the scope and visibility restrictions set forth by the allowed language units of distribution. This would require support for more types of remote references and complicate the implementation of collective task termination. For these reasons, it appears that a policy of disallowing task migration for distributed memory multiprocessor implementations is reasonable. After an initial run-time system is completed, future work aimed at resolving these issues may be undertaken.

5.7 Collective Task Termination

The Ada language has specific rules regarding the order of termination of tasks in a program. These rules state that no task may be terminated and no block or subprogram exited until all of the tasks dependent on that task, block, or subprogram have terminated. These rules are complicated somewhat by the provision for collective task termination. This involves the **terminate** alternative in a **select** statement. In this case, the parent task, block, or subprogram may terminate or exit if all dependent tasks are terminated or waiting at a **terminate** alternative. If this condition is met, the waiting tasks are terminated and the parent task terminates or the parent block or subprogram is exited.

Tasks may also be dependent on library units or packages, if they are declared in one of those units. Since these units are passive and not active, the termination of the dependent tasks is not defined by the language. The main program may terminate while these tasks are waiting to terminate or even waiting for an entry call.

A message passing scheme for implementing the task termination protocol of the simple case is discussed in Sec. 5.2 above. These scheme is easily implemented on a distributed system. In general, supporting the collective termination protocol on a distributed system with no shared memory is not so straightforward.

However, the problem here is greatly simplified by the allowable language units of distribution proposed in Sec. 4. As for the simple case of task termination, it is discussed in Sec. 4.4 that tasks dependent on other tasks, blocks, or subprograms will always reside on the same node as their parent. The only way for a task to be dependent on a parent located on a remote processor is for the parent to be a library package. In this case, the termination of the task is not defined by the language, and no coordination of termination with the remote parent is necessary. In short, the entire task termination problem, both the simple case and the collective termination case, is reduced to that of the problem in the uniprocessor situation.

The simple case of task termination is supported by the simple message passing scheme discussed in Sec. 5.2 above. As for collective task termination, a parent task, block, or subprogram must call the run-time system in order to terminate or exit. (A block or subprogram need only call if it is possible for a task to depend on it. This can be determined at compile time). The run-time system then checks the status of dependent tasks before dispatching any new tasks to execute. If the dependent tasks are terminated or waiting to terminate, the waiting tasks are terminated and the parent is terminated or allowed to exit. Otherwise, the task, block, or subprogram is completed and must wait for terminate messages from its still active children (messages from the terminated and waiting to terminate children will have already been received). When these messages arrive, another check is made to see if all the children are in one of the two allowable states, since it is possible for a waiting to terminate task to become active again. When the conditions are met, the parent task is terminated, the parent block is exited, or the parent subprogram is allowed to return.

5.8 Exception Handling

With the specified units of distribution, any thread of execution that has an exception handler will have it on the same processor that the thread is executing on. This allows exceptions to be handled in the same way that they are in the uniprocessor case. If a block that experiences an exception does not have a handler, that exception must be propagated up the dynamic chain. It is possible in this case that the exception must be propagated across processor boundaries. This can be implemented in a straightforward manner by indicating the exception in the return message of a remote subprogram call.

It is also possible for exceptions to occur during the synchronous activation and termination of tasks as well as during inter-task rendezvous. In such cases, when an exception is required to be propagated to one of the involved tasks, a message is sent to it indicating this situation instead of the usual message the task expects. The semantics of task activation, synchronization, and termination are quite clearly defined, so that such a scheme of propagating exceptions can be implemented in a straightforward manner.

5.9 Support for the Shared Memory Model

The issue here is determining where an object, subprogram, or task is located at run-time. With the units of distribution proposed here, objects, subprograms, and tasks in packages can be bound to a processor at compile time, and reference to them is achieved through a run-time system call. The only objects that may not be bound to a processor statically are dynamically created ones, including tasks. These objects are referenced by pointers that

indicate processor location. Dereferencing these pointers must be handled by the run-time system.

5.9.1 Remote Object Reference

This area is straightforward and involves the details of specifying the various types of local and remote references. In the case of accessing variables in remotely located packages, a run-time system call requesting the reference replaces the normal reference. A message is then sent by the run-time system to the node containing the object. Unlike the suggestion made in [LeB82], the remote reference results in a scheduling point for the referencing processor, so that other useful work may take place. This appears to be a better approach than suspending the processor in the case of the NCUBE, since the current amount of overhead involved in Vertex message passing can be high, especially in the case where the remote reference is to a processor which is several communication links away. The strategy then, is to have the run-time system send a message (including the new value if the reference is on the left hand side of an assignment) and suspend the referencing task to await a reply. If the remote variable is being read, the reply contains its value. Write operations on remote variables also block the referencing task to force the same conditions for the following statements as would occur in the uniprocessor case. Also, this operation can be implemented simply, since non-local references to statically allocated objects can be determined at compile time.

5.9.2 Pointers

The strategy for implementing task pointers is discussed above in Sec. 5.5, but the issue of system wide pointers to memory objects must be addressed. Although some distributed language proposals recommend that passing pointers between processors be disallowed [Lis82], it appears that employing this strategy in the case of Ada is in violation of the language standard. A system wide pointer scheme for tasks and other memory objects can be implemented as processor–address pairs. For each pointer reference, a check must be made to see if the processor is remote. This can be handled as part of the code generator. In the case of a remote reference, the run-time system is called as in the strategy outlined above for non-local memory references.

5.9.3 Remote Subprogram Call

The scheme for implementing remote subprogram call is a straightforward one. The calling task simply blocks, and the run-time system on the caller's node passes a message containing the activation frame. The run-time system kernel receiving the message places the frame on

some stack space and passes control to the subprogram as if it were a task. Upon completion, a call to the run-time system is made so that the results in the frame can be passed back to the calling node. This scheme is similar to the implementation described in [BiN84].

5.10 System Dependent Features

The following four sections describe the requirements of the run-time system to support the mentioned system dependent features. Since their implementation is so closely related to the target machine, the discussions that follow consider the NCUBE multiprocessor in particular.

5.10.1 I/O

The environment for I/O provided by Ada consists of the predefined package TEXT_IO as well as several predefined generic packages. These packages provide variables and subprogram interfaces for performing file I/O. In order to provide the Ada I/O environment, these packages must be implemented. Because they are packages, they are not part of the run-time system, and therefore are not replicated on each processor in the network. Instead, a site can be specified for TEXT_IO as well as each individual instantiation of any of the generic I/O packages. This will provide for some remote variable access and subprogram calls, but as is discussed below, file I/O in the case of the NCUBE is handled through the host processor, which is remote to all node processors.

The operations provided by the I/O packages are common ones that can be implemented easily on top of existing operating system primitives. In the case of the NCUBE hypercube, the host processor is needed to provide the interface to the file subsystem through Axis. The nodes running the I/O routines simply forward their file I/O requests through Vertex to the host processor.

5.10.2 Dynamic Memory Allocation

Dynamic non-stack allocation of blocks of memory is a common feature in languages and is usually supported at the operating system level. The node operating system Vertex is no exception, as it provides memory allocation and deallocation services. The code generator can insert calls to the run-time system along with code to initialize the newly allocated object. The size of the object is a parameter to the run-time system, which then passes it on to the allocation routine.

In the implementation discussed above, the run-time system plays an intermediary role between the program and the operating system. In some implementations, calls to the oper-

ating system are made less frequently, with a large block of memory being allocated each time. These large blocks then form a heap that is managed by the run-time system. This scheme is used to avoid the large amount of overhead associated with an operating system call. However, this scheme is not appropriate for use with Vertex, because it is not a large scale time shared operating system. Rather, it is more an extension of the run-time system, providing low level operations and communications only. The run-time system is invoked to provide run-time error checking on the amount of memory requested, and to raise STORAGE_ERROR if necessary. For error free requests, the call to the Vertex allocation code is made, but at this assembly language level, the call is relatively inexpensive.

5.10.3 The Packages STANDARD, SYSTEM, and CALENDAR

There are three required predefined packages in Ada in addition to those used to implement I/O. They are the packages STANDARD, SYSTEM, and CALENDAR. These packages provide type definitions and subprograms (no objects) that provide basic operations. For this reason, we believe that these packages should be viewed as user interfaces to the run-time system routines. This implies that each processor that can view these packages should have a copy of them resident in its memory. In other words, every processor should have a copy of packages STANDARD and SYSTEM, and processors holding a section of code containing a **with** CALENDAR clause should have a copy of that package as well. This strategy is implied in [VMB88].

This approach to supporting these predefined packages in a distributed system causes no problems in the case of packages STANDARD and SYSTEM. However, this approach to implementing the CALENDAR package has a major pitfall. That pitfall is the need for each processor to provide a CLOCK function that returns a system wide valid value of the current time. This requires synchronization among the various CLOCK functions. This is not a problem if the underlying hardware clocks are synchronized, but this is not usually the case in any distributed architecture. In the case of the NCUBE, there is only one interval timer per processor and no time-of-day clock. This interval timer is used in our run-time system for time-slicing, but a count of the total number of clock ticks that have elapsed since the start of execution on that particular node is maintained. This count of ticks varies from node to node, as it is updated only when a new value is about to be loaded into the timer. Also, each node does not necessarily begin execution at the same time.

There are several possible approaches to solving this problem. One possibility is to provide the CLOCK function on only one processor. This maintains a consistent clock, but requires variable network delay in reading the clock that may be significant. Another possibility is to synchronize the start of each processor, and then work from the assumption that a clock value read on any processor is potentially a time slice behind the real current

time. This scheme, however, requires significant start up overhead to synchronize the starting times. Also, such a synchronization may not even be possible.

The best solution to this problem is support in hardware for two timers per processor. In addition to the interval timer, a time-of-day clock is also provided. This allows time-slicing and the event queue to be managed as described above, but the time-of-day is provided by a separate clock. A similar solution that is proposed in [VoM87b] utilizes a time-of-day clock along with a readable/writable compare register. The compare register contains an absolute time value that indicates the next timer interrupt. In the case of either solution, though, the time-of-day clocks for all processors must be synchronized. This can be achieved by driving all ticks from the same line, as is currently done with the interval timers on the NCUBE.

5.10.4 Attributes

The Ada language provides attributes whose evaluation yields a predefined characteristic of a named entity. An example of an attribute is P'COUNT where P denotes a task entry and P'COUNT evaluates to a universal integer equal to the number of tasks currently queued and waiting for that entry. Clearly this attribute, as well as several others, require a call to the run-time system in order to be evaluated. Simple functions can be inserted into the run-time system, one to evaluate each necessary attribute. The call to the run-time system is inserted in the code where the attribute occurs.

5.11 Additional Features

5.11.1 Priorities

Although support for priorities is not required by the Ada language definition, their inclusion in the run-time system is desirable, particularly for real-time systems. Fortunately, implementing priorities is a straightforward task.

The Language Maintenance Committee has interpreted the language definition regarding priorities to mean that no task can execute on a given processor if another local task of higher priority is runnable. This requires a preemptive scheduling strategy where, if a higher priority task becomes runnable while a lower priority task is executing, the lower priority task must be preempted. This is not difficult to implement, however, since a change in a task's state making it runnable can only occur by action of the run-time system. All that is necessary then, is to schedule the highest priority available task at every exit from the run-time system.

Task priorities in Ada are static values. They are simply stored in a location reserved in a

task's task control block. A task's priority changes temporarily, however, when it is engaged in a rendezvous with a higher priority task. In this case, the value in the task control block is altered and the old value saved in another location in the block. Because of the possibility of cross processor rendezvous, priority values should be embedded in rendezvous messages so that they are readily available to the run-time system.

5.11.2 Generics

Although the instantiation of generics can be supported at run-time, it is preferred that they be supported at compile time. A possible run-time issue occurs if one copy of a generic piece of code is to be shared among several instantiations at once. This would only be considered for groups of instantiations that reside on the same processor. Such sharing would in general require some run-time type checking and execution of different code branches depending on types. This run-time support though can be achieved through code insertions and modifications and need not involve the run-time system.

6 Status of Current Implementation

This section provides an overview of a subset of the above described run-time system that has been implemented on our NCUBE machine. The run-time systems routines are implemented in C, with the exception of the extensions to the operating system which are coded in assembly language. C was used since it is a high-level language with a compiler for it that is targeted to the NCUBE node processor. Using a high-level language allows for rapid implementation and easy modification in this development stage of the run-time system. When complete, the code can be converted to assembly language.

The features currently supported by the run-time system implementation are those described in Sec. 5.1 through Sec. 5.4 above, namely task elaboration, activation, normal and abnormal termination, delay, and rendezvous. These features were implemented with the simple assumption that tasks in an Ada program would be allowed to reside on separate processors. The issues associated with specifying language units of distribution did not affect the implementation of these simple run-time system primitives.

In addition to the routines directly called to support the above stated features, several other routines were needed to perform the behind the scenes but necessary basic operations. These routines include support for timing management, scheduling of tasks, and reception and processing of messages. Modifications to the operating system were also necessary to support context switching and interrupt processing. These low-level functions form the core

environment necessary upon which all the run-time system functions described in Sec. 5 can be implemented.

One advantage to implementing the Ada run-time system on this target is that a duplication of run-time functions is avoided. Vertex is basically a communications and interrupt processing kernel that assumes one process per processor. The existing features of Vertex were used as building blocks for the run-time system. Where necessary, Vertex was modified to help implement the run-time system and support multitasking on a node. This situation is in sharp contrast to several current Ada compilers that run on top of complex time-sharing operating systems and do not or cannot use all of the operating system's primitives. Although performance figures for the run-time system implemented are not yet available for the NCUBE implementation, it is expected that they will be favorable in comparison to those obtained from commercially available compilers. Performance figures for the features already implemented can now be measured using the algorithms proposed in [CDV86], and a comparison to existing compilers made.

More intricate details of the system and its interface to the architecture and operating system are given in [CMV87]. This reference also contains some code listings that demonstrate the use of the system.

7 Future Directions

At this point, a partial distributed run-time system for Ada on the NCUBE machine has been implemented. This portion of the run-time system supports core operations and was developed somewhat independently of the language issues addressed in this paper. Now that these issues and their impact on run-time system components has been considered, work to implement the additional run-time system features may proceed.

The features to be incorporated into the system next are named in Sec. 5. Initial implementation strategies have been discussed, and an attempt to realize these features will proceed. If unreasonable overhead results from the implementation of these constructs, it will most likely be associated with implementing the shared memory model. Performance measurement of these features must be used to help evaluate the efficiency of their implementation. Various implementation strategies must be considered, so that one with favorable performance can be uncovered.

The next step after completing the run-time system is the retargeting of an Ada compiler to the NCUBE multiprocessor. This is a large undertaking, and must be carefully considered. A key factor in the decision to proceed in this direction will be the performance of the run-time system. It may be the case that some areas will suggest that modifications to the language are necessary for this architecture. Although this breaks away from the Ada standard, it may

very well provide an efficient high level parallel language for loosely coupled multiprocessors that incorporates most of the advantages discussed in Sec. 3 above.

Certainly it is desirable to have a high level parallel language running on a multiprocessor. We would like to see this accomplished in the form of Ada running on our NCUBE machine. However, if circumstances suggest that this is unwise, our approach can still be used to implement a similar high level parallel language on a hypercube or some other loosely coupled homogeneous architecture.

References

[Ard84] Ardö, A., "Experimental Implementation of an Ada Tasking Run-Time System on the Multiprocessor Computer Cm*," *Proceedings of the 1st Annual Washington Ada Symposium,* pp. 145-153, March 1984.

[Ard86] Ardö, A., "Efficiency Aspects on Ada Run-Time Support for Multiprocessors with Shared Memory," Technical Report, Department of Computer Engineering, University of Lund, March 1986.

[Bar84] Barnes, J.G.P., *Programming in Ada,* Addison-Wesley, Reading, Mass., 1984.

[BiN84] Birrell, A.D. and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems,* vol. 2, no. 1, pp. 39-59, February 1984.

[Buz88] Buzzard, G.D. "High Performance Communications on Hypercube Multiprocessors," Ph.D. Thesis, The University of Michigan, (work in progress).

[CDV86] Clapp, R.M., L. Duchesneau, R.A. Volz, T.N. Mudge and T. Schultze, "Toward Real-Time Performance Benchmarks for Ada," *Communications of the ACM,* vol. 29, no. 8, pp. 760-778, August 1986.

[CMV86] Clapp, R.M., T.N. Mudge and R.A. Volz, "Solutions to the *n* Queens Problem Using Tasking in Ada," *ACM SIGPLAN Notices,* vol. 21, no. 12, pp. 99-110, December 1986.

[CMV87] Clapp, R.M., 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, The University of Michigan, 1987.

[Cor84a] Cornhill, D. "Partitioning Ada Programs for Execution on Distributed Systems," *1984 Computer Data Engineering Conference,* 1984.

[Cor84b] Cornhill, D. "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pp. 153-162, October 1984.

[FiW86] Fisher, D.A. and R.M. Weatherly, "Issues in the Design of a Distributed Operating System for Ada," *IEEE Computer*, pp. 38-47, May 1986.

[HMS86] Hayes, J.P., T.N. Mudge, Q. Stout, S. Colley and J. Palmer, "A Microprocessor-based Hypercube Supercomputer," *IEEE Micro*, pp. 6-17, October 1986.

[Jes82] Jessop, W.H., "Ada Packages and Distributed Systems," *ACM SIGPLAN Notices*, vol. 17, no. 2, February/March 1982.

[LeB82] LeBlanc, T.J., "The Design and Performance of High-Level Language Primitives for Distributed Programming," Ph.D. Thesis, TR-492, Computer Science Department, University of Wisconsin, December 1982.

[Lis82] Liskov, B., "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 203-210, May 1982.

[LRM83] *Ada Programming Language (ANSI-MIL-STD-1815A)*. Washington, D.C. 20301: Ada Joint Program Office, Department of Defense, OUSD (R&D), January 1983.

[MBA86] Mudge, T.N., G.D. Buzzard and T.S. Abdel-Rahman, "A High Performance Operating System for the NCUBE," *Proceedings of the 1986 SIAM Conference on Hypercube Multiprocessors*, pp. 90-99, September 1986.

[Ros87] Rosenblum, D.S., "An Efficient Communication Kernel for Distributed Ada Run-Time Tasking Supervisors," *ACM Ada Letters*, vol. 7, no. 2, pp. 102-117, March/April 1987.

[VMB88] Volz, R.A., 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*, (to appear).

[VMN85] Volz, R.A., T.N. Mudge, A.W. Naylor and J.H. Mayer, "Some Problems in Distributing Real-Time Ada Programs Across Machines," *Ada in use, Proceedings of the 1985 International Ada Conference*, pp. 72-84, May 1985.

[VoM87a] Volz, R.A. and T.N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 449-459, April 1987.

[VoM87b] Volz, R.A. and T.N. Mudge, "Instruction Level Mechanisms for Accurate Real-Time Task Scheduling," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 988-992, August 1987.

[Wea84a] Weatherly, R.M., "Design of a Distributed Operating System for Ada," Ph.D. Thesis, Clemson University, August 1984.

[Wea84b] Weatherly, R.M., "A Message-Based Kernel to Support Ada Tasking," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pp. 136-144, October 1984.