A Distributed Programming Environment for Ada*

Peter Brennan, Tom McDonnell,
Gregory McFarland, Lawrence J. Timmins and John D. Litke
Grumman Data Systems
1000 Woodbury Road
Woodbury, New York 11797

Abstract

Despite considerable commercial exploitation of 'fault tolerant' systems, significant and difficult research problems remain in such areas as fault detection and correction. This paper describes a research project to construct a distributed computing test bed for loosely coupled computers. The project is constructing a tool kit to support research into distributed control algorithms, including a distributed Ada compiler, distributed debugger, test harnesses, and environment monitors. The Ada compiler is being written in Ada and will implement distributed computing at the subsystem level. The design goal is to provide a variety of control mechanisms for distributed programming while retaining total transparency at the code level.

Introduction

Many new system designs specify a distributed architecture to attain incremental growth or increased computational power. These systems typically have homogeneous processors linked either by shared memory or by a message passing system. Concomitant with easy expandability and large computational power, one gains some resiliance against hardware faults. That is, if one processor fails, only the work executing on that processor is lost, not the entire work load. If one adds the capability to detect processor failure and to move the work from that failed processor to other working processors, then some tolerance for both hardware and software faults is attained that cannot be achieved with single processor systems.

* Ada is a registered trademark of the Department of Defense, Ada Joint Program Office.

Further, if one can move work from a failed processor to an active processor, rather easy extensions allow work to move from one active processor to another, achieving a load balancing effect for maximum output from the processor resource.

This ability to function despite hardware failure has made distributed, loosely coupled architectures a favored architecture for ultra reliable systems. To make this architecture effective, we must partition a problem into several parts so that each part can execute relatively independently on separate computers. The partitioning process introduces requirements to coordinate the execution of the several parts and to verify that each part is operating properly so that, if a failure occurs, corrective action can be taken. Such methods for problem coordination and control in a distributed environment are the principal focus of this research. We wish to assess whether, given the proper tools, one can construct loosely coupled, distributed applications that are cost effective, reliable, and efficient.

When a problem solution is developed for execution on a single processor computer, the usual method is to design several modules that jointly solve the problem. Coordination of the solution process requires a communication channel between modules, usually via shared variables or messages. Further, any shared data must be specified and storage allocated. This design results in intimate coupling between the several modules, with a significant chance for error. Ada provides extensive checking of the interfaces between modules and the operations allowed on each data element, greatly reducing the severity of module interface errors.

When a problem is partitioned for execution on a distributed processing host, one designs several programs (instead of modules) that jointly solve the problem. Interface errors may still occur, but since a compiler can process only one program at a time, there is no compiler support for checking and controlling the inter-program interfaces. Hence one would like to extend the power of Ada to distributed programs. In such an extended language, a problem is still decomposed into separately executing programs (Ada tasks), but data sharing and module synchronization are implemented and checked by the compiler. While such an extension itself presents implementation difficulties, two additional problems are present in a loosely coupled environment: assuring liveness and serializability. Thus, we require a test environment to evaluate candidate implementation methods and to develop efficient new algorithms. The Ada language was designed to provide support for a distributed programming paradigm. Its' visibility and synchronization rules provide a model for data sharing, while the task and rendezvous constructs provide a control model. Ada provides primitive mechanisms for assuring liveness and serializability, but the attainment of these goals is left to the programmer. To assess the viability of Ada's

model, we require two things: a distributed host with a validated Ada compiler, and a tool kit for developing, debugging and measuring the performance of distributed programs. However, because Ada provides a model but not an implementation of distributed programming constructs, we must expect to try a variety of implementations before settling on one with acceptable performance. Hence we require a compiler that we can modify to try various implementations of Ada.

These considerations have led to the establishment of a Fault Tolerant Computing project at Grumman Data Systems with the following goals:

1. To construct an Ada compiler for a distributed architecture host so that the implementation of Ada's model can be varied significantly.

2. To implement several distributed programming models and assess their viability for serious problem solving in realistic environments.

3. To develop models and methods for solving the liveness and serializability problems, and to test these ideas on the distributed programming environment provided by the first two goals.

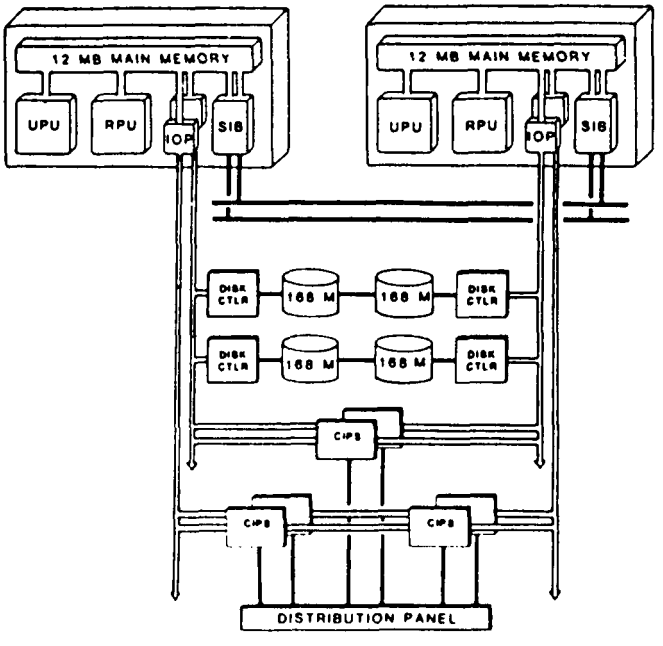
The project began in July of 1985 with a goal of constructing the foundation Ada compiler by summer 1986 and providing the first implementation of a distributed programming model by early 1987. The remainder of this paper describes the design and development of the foundation compiler and its supporting environment, and concludes with an outline of a distributed programming implementation for Ada.

Hardware Technology

The hardware base is the Eternity E-5000 computer system from Tolerant Systems, San Jose, CA. This system contains loosely coupled processors built with the National Semiconductor 32000 series VLSI processors. The operating system is TX, a superset of Unix BSD 4.2 and System V with extensions for transaction processing, distributed file systems, and built- in fault detection and recovery. The hardware is targeted for the commercial on-line transaction processing market, and so features a particularly robust and flexible communications capability. The fault tolerant capability is achieved with fail fast processors, dual redundant communication paths, and fault detection and reconfiguration software. Further, operating system services themselves are distributed in such a way as to support process migration, either to avoid faults or to provide load balancing. This support for distributed programming algorithms is an important advantage; it minimizes the infrastructure we must build.

Each processing element is actually a tightly coupled set of 32000 processors. (See Fig. 1). One processor (UPU) is dedicated to user applications, one (RPU) to the operating system, and one (CIP) to I/O and communications protocols. The UPU provides a UNIX compatible executing environment, while the CIP provides a real-time environment. Both processors have a common system language (C) and machine language. Although operating system services differ somewhat on each processor, one compiler can produce code that will execute on either processor. This permits us to develop an Ada compiler that will produce code for both a time sharing and a real time environment.

The file system is UNIX compatible at its interface, but highly modified in its implementation to provide a global name space and a robust foundation for system operation. In addition to traditional services, the file system provides an efficient, guaranteed message delivery system and plexed files with automatic restoration after failure. This is an essential system service for effective implementation of Ada's distributed programming model.



E-5000 Configuration Example
Figure 1

Compiler Technology

The Ada compiler must be constructed in such a way that the run time library can be modified. Since Ada's model for distributed computing is centered on the task construct, the

inter-task rendezvous and task scheduling algorithms must also be modifiable. We have chosen the retargetable compiler technology from TeleSoft, San Diego CA as the base on which to build. This system provides the syntactic and semantic analysis for Ada, manages an Ada library, and provides output in a tree structured form at approximately assembly language level. Our task is to build a suitable code generator for the E-5000 hardware. A key feature is that sufficient information on the Ada task implementation is available so that we can modify the Telesoft implementation model if required.

One of our themes when implementing this compiler is program execution efficiency. Execution efficiency not only requires an efficient algorithm, one of the primary foci of this research, but also an efficient implementation of those algorithms by the compiler. Thus code optimization becomes a theme of the first part of the project. Because of our implementation strategy, the potentially arduous construction of optimization algorithms splits naturally into three parts. We will depend on the TeleSoft front end for optimization flow of control, common sub-expression elimination, etc. The output of the compiler is National Semiconductor assembly code for the 32000 processor. The assembler on the E-5000 implements extensive optimizations that are effective for a C language compiler, such as code hoisting and instruction selection. Thus our code generator will concentrate on optimizations such as register allocation, minimization of bounds checks, efficient exception propagation, and the like.

Since the compiler will produce code for a real time environment, we must ensure that efficient programs are possible. Further, a highly modular language like Ada could invoke a large number of subroutine calls, making efficient call/return mechanisms a requirement. We focus on our implementation decisions surrounding the call/return mechanism as an example of tradeoffs involved during the compiler construction process.

The call/return mechanism has several basic requirements. It must:

1. Allow passing of data into and out of a subroutine.

2. Allow saving and restoring of temporary registers.

3. Allow access to out of scope variables.

4. Allow exception propagation out of the local scope.

5. Allow task switching and hence logical reentrancy.

The E-5000 uses a stack mechanism, growing down from high memory locations, for temporary variables including frame pointers. Thus the basic call/return paradigm is a classic one:

Call: **Put variables on the stack**
Put return address on the stack
Branch to the subroutine

(in called routine)
Save old stack base and old frame pointer on stack
Set new stack base and new frame pointer

Return: **Restore old stack base and old frame pointer**
Branch back to return address on stack

(in calling routine)
Remove return variables from stack

To extend this model for Ada, we must decide how to allocate stack space considering the multi-tasking Ada model and how to propagate information to/from the called routine with a minimum of overhead. Ada requires extra information be passed across this interface to allow out of scope variable references and to propagate exceptions. It was a particular goal to minimize the overhead of these latter requirements.

For the stacks, we have adapted the results from [GUPT85], namely to use a static storage area for module instances and a dynamic heap for temporary variables, satisfying requirements 1,2,5. (This scheme is often called a Berry-heap after [BERR78]). When allocating stacks for independent tasks, one must account for the possibility of collision of these stacks with each other [YEH86]. There are only two solutions, impose a static limit on the size of the stacks, or dynamically create room when required. In either case, the stack-full detection mechanism provided by the hardware is no longer useful for multiple stacks. We must implement the checks efficiently in software.

We allocate an initial stack with the intent to dynamically allocate more stack space if and when required. This approach makes effective use of available memory even for very large numbers of tasks, and imposes very little overhead [YEH86]. However, we now must check for stack overflow before every stack usage, an unacceptable overhead. Our first solution was to check, before every call, that parameters would fit on the stack, and then check at every entry that local variables (the frame) would fit on the stack. This is a two call overhead for every original call, an unacceptable result. The final design depends on the observation that stack requirements for local data and parameter passing are known at compile time, so that we can substitute one call on entry to each routine to check for sufficient stack space. Further, since routines that invoke no other routines typically have very small stack requirements, by requiring a small buffer space be present on all stacks we can remove all stack checking overhead for such calls. We accepted such minimal overhead for the benefits of a highly dynamic stack allocation mechanism.

The remaining two requirements, to implement out of scope references and to permit exception propagation to cross the call/return interface, each require separate treatments. Out of scope references in a multi-task environment are often implemented by copying a 'display' onto the currently active stack before every call. This display contains the storage offset pointer for each visible module, including the calling module. Each out of scope reference is implemented as an indirect reference relative to the proper pointer plus an offset. The difficulty with this solution is the requirement to set up the stack before each call. Although optimization algorithms could avoid setting up unnecessary displays, we would prefer to avoid the overhead altogether.

Our solution requires a static display area, one per task. Each module has a statically determinable lexical level that serves as an index into this table. When calling any module, we save the current value in the table at our lexical level, and overwrite the proper frame pointer in the table. On return from the routine, we merely copy back the original contents of the display. This requires an overhead of one load and two stores per call, optimizable to no action at all if we can determine that the routine being called does not reference any variables at our lexical level or higher and calls no other routine.

An efficient solution to exception propagation requirements is more complex. For locally raised exceptions, we can clearly use a direct transfer to the exception handling code. However, if an exception must be propagated to an outer scope, we must 'unravel' the stack frames as we search for the handler. In addition, we require that the cause and location of the exception raising be determinable in case a handler is not found. For real time programming, we would like such a mechanism to be swift. Further, if the exception could not be handled at any level, for debugging purposes we should not unravel the entire stack before we determine that the exception is unhandlable. Otherwise, essential debugging information is lost.

Our solution requires no overhead at call time and uses a binary search to identify the relevant exception handler before unraveling the stack. At compile time, each exception is uniquely identified as to raise location and reason, and every exception handler is uniquely identified as to the exceptions it handles, permitting identification of exceptions in a user friendly way should a handler not be found. The identification information, together with the addresses of the scope of each exception handler, is stored in a table in static memory. An initialization routine sorts this table before the program runs. If an exception must be propagated, the propagation code follows the stack pointer chain backwards, searching the common exception table for exception handlers that apply to the address given by each instance of the stack pointer chain until a handler is found. The table can be searched quickly for

rapid exception propagation. When a handler is found, the stack is quickly unraveled in one operation to the proper point and the handler invoked.

While not an exhaustive list, these items illustrate some directions we are taking in the development of an efficient Ada compiler. Many of our efficiency oriented algorithms are heavily parameterized so that we can vary their effect and study the resulting program behavior. This approach will allow us to tune the compiler for best effect under realistic conditions. Results of these efforts will be reported at a future conference.

Distributed Programming Model

When implementing an Ada compiler for a distributed programming host computer, there are three levels of capability to be considered, namely:

1. Minimum capability that satisfies the Ada Language Reference Manual [ANSI83].

2. Permit advice from the programmer to influence the implementation or execution of the model.

3. Enhanced functionality within the requirements of the Ada Language Reference Manual.

The remainder of this section addresses some issues pertinent only to the minimal capability implementation.

The execution of parallel, distributed processes under one computational model introduces such complexities that few practical systems today are entirely transparent to the user. The mark of a successful implementation is correctness, general applicability, and the capability to simplify the task of programming parallel execution paths. In contrast, Ada seeks to achieve two different goals: a simple inter-process communication paradigm and the efficacy of a complete semantic check of the entire collection of processes, viewed as a whole. Whether these goals are necessary or sufficient for a successful implementation is to be determined.

Ada defines a task model that provides a set of primitive communication mechanisms (accepts/entry calls) to implement parallel tasks. Although use of these requires explicit programmer cognizance, the programmer's task is simplified somewhat. The price for this simplification is that the compiler writer must implement correct interpretations for three shared elements: data, control via exceptions, and program state. Each of these elements is considered separately below.

To provide a background for this discussion, some fundamental design decisions must be noted. The first version of the distributed compiler will produce an executable image that executes on each distributed host unaltered. In other words, the instantiation of any module will execute on only one host, though its code image is present on all hosts. This decision means that the code on each host is larger than the minimal required, but that addresses not on the stack and not dynamically allocated are universally correct from host to host. Further, our hardware is a segmented, virtual memory machine, so that physical memory is not significantly wasted by this decision.

A second design decision is to use the operating system message passing facility for all intertask communication. Since we have compiled the program as a whole, targeted for one uniform processor, this communication need not incur the overhead of formatting/unformatting data, and so it can be used for co-located tasks as well as distributed tasks.

A third decision is that only tasks will be considered for distribution during the first implementation. (While this is not strictly true as we shall see, this provides the primary focus when designing the implementation model.) Further, to ease initial implementation efforts, no access variables can be referenced across a distributed interface. Now let us return to a discussion of how we intend to share data, control via exceptions, and program state information.

Data sharing between two asynchronous tasks takes several forms. The first is via data that is visible to two different tasks by operation of the scope rules of Ada. The Ada Language Reference Manual specifies that two tasks can 'see' the effects of updating shared variables only at synchronization points such as those associated with a rendezvous or by pragma 'SHARED', allowing every access of a variable to be a synchronization point. However, the Ada Language Reference Manual does not require that the compiler detect erroneous programs that violate these rules. A second, indirect way to share data is by the common invocation of library routines that reference static data. For example, a terminal I/O routine in a library package might reference static data to define the current line number on the screen; every call to this routine may alter the data.

Motivated by these two concerns, we have decided that the pragma 'SHARED' will not be allowed for two tasks that are not co-located within one host process. To address the indirect sharing of variables via library packages, we define three classes of objects (functions and procedures): idempotent, serially reusable, and re-entrant. The first class will execute correctly without any historical information. Any routine that does not access static data or any 'state of the world' is in this class. The second class indicates routines that access some static data, but that can accept successive

calls once the first call is complete. Most library routines are in this class. The third class, while they may depend on static data, may also be called by another routine before the first request is complete. These routines, such as I/O drivers, usually depend on a separate temporary data store (stack) per task to achieve their re- entrancy.

If a routine is declared idempotent, then we may execute any available local copy of the relevant code, taking no care to share static data among distributed tasks. This is the default nature for procedures and functions. If the routine is declared serially reusable, then we will execute the call on the one host that contains the instantiated version of the routine, and all calls will be queued in a FIFO manner. If the routine is declared re-entrant, then we will execute the call on the local host and broadcast any updated static data at the completion of the call. It is the programmer's responsibility to ensure that the specification of the proper behavior model matches his or her intent.

Another information sharing between two concurrent tasks is via the exception propagation structure. Since the colocation of a task and any exception handlers that it may invoke are not guaranteed, we must provide both a fast means to determine the location of the exception handler and a means to propagate the exception to that handler. Our decision to use a common program image allows the exception propagation logic to execute as a idempotent routine, determining the location of the handler before invoking any communication overhead. The communication required to pass control to a remote site is reduced to the identification of the raised exception.

Global state information is shared among distributed processes by the Ada requirement for task termination. When a task has an open terminate alternative, it must consider the state of all dependent tasks, sibling tasks, and the state of the parent task before entering the terminated state. In turn, this means that one must achieve a globally consistent picture of the state of all such tasks so that a correct decision can be made. There are only two solutions to this requirement. One solution elects or appoints a master controller to determine the state of the world, while the other solution requires periodic broadcasting of all states to achieve a consensus on a consistent state. The latter approach is often called a consistent checkpoint method, and often entails significant overhead waiting for all tasks to achieve a stable state. For this reason, we have elected to use the first method, by electing a 'controller' task as that task that dominates the immediate termination decision. By polling means, outlined in [JAH85], this one task (actually the local run time system attached to that task) will calculate the termination condition for all subordinate tasks.

Our general direction for implementation of the Ada distributed programming model has been decided. Our next step is to consider means to debug distributed processes and to measure the effectiveness of our initial implementation. This effort will result in a test suite of distributed programs, designed especially to test distributed control algorithms rather than just the computational advantage of parallel computation. The suite will be then used to evaluate the effectiveness of various distributed programming models.

References

[ANSI83] ANSI/MIL-STD 1815A, Reference Manual for the Ada Programming Language; January 1983

[BERR78] D. Berry, L. Chirica, J. Johnston, D. Martin, and A Sorkin, "Time required for reference count management in retention block- structured languages, part 1,", Int. J. Comput. Inform. Sci., 7(1), pp.91-119 (1978)

[GUPT85] Rajiv Gupta and Mary Lou Soffa, "The efficiency of storage management schemes for Ada programs", Ada Letters, Vol 5, 2, pp.164- 172, (1985)

[JAH85] Rakesh Jha and Dennis Kafura, "Implementation of Ada Synchronization in Embedded, Distributed Systems", Virginia Tech report TR-85-23, 1985.

[YEH86] D. Yun Yeh and Toshinori Munakata, "Dynamic Initial Allocation and Local Reallocation Procedures for Multiple Stacks", Comm. ACM, Vol 29, 2, pp.134-141, February 1986