N89-16347

COMPARING HOST AND TARGET ENVIRONMENTS FOR DISTRIBUTED ADA PROGRAMS

MARK C. PAULK
SYSTEM DEVELOPMENT CORPORATION
4810 BRADFORD BLVD NW
HUNTSVILLE, AL 35805

Abstract

The Ada* programming language provides a means of specifying logical concurrency by using multitasking. Extending the Ada multitasking concurrency mechanism into a physically concurrent distributed environment which imposes its own requirements can lead to incompatibilities. These problems are discussed. Using distributed Ada for a target system may be appropriate, but when using the Ada language in a host environment, a multiprocessing model may be more suitable than retargeting an Ada compiler for the distributed environment. The tradeoffs between multitasking on distributed targets and multiprocessing on distributed hosts are discussed. Comparisons of the multitasking and multiprocessing models indicate different areas of application.

Keywords: Ada, distributed processing, multitasking, multiprocessing, Ada Programming Support Environment (APSE), software engineering, computer networks, interprocess communication.

1. INTRODUCTION

In designing a solution to a real-world problem, the systems analyst is frequently faced with the fact that the real world functions in terms of concurrent activities. Many applications are modelled most naturally by logically concurrent tasks, but most computer languages do not support concurrency. Even when concurrent activities can be distributed on a computer network to achieve physical as well as logical concurrency, the designer must build the interfaces between the physically distributed components of the system as well as partition on its logically concurrent boundaries.

---

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

Recognizing that concurrency is the natural approach to solving many problems, the Department of Defense (DOD) developed multitasking as an integral part of the Ada programming language specification. Concurrent tasks may communicate through task activation and termination; they may share global variables; or the communicating tasks may rendezvous using entry calls and accept statements. Synchronization between communicating tasks may use selective waits, conditional entry calls, or timed entry calls [1].

The host/target model was used in designing the framework for Ada environments [2,3]. Ada programs are developed on a host computer system. The program is compiled by a cross-compiler, and the executable module downloaded to the target system on which it is to execute. This model describes the typical softwared development environment for embedded systems.

A range of options can be considered for the distributed target. Alternatives to multitasking may be chosen, such as a multiprocessing approach relying on an I/O-oriented interface for interprocess communication. Fully transparent distribution of the program can be implemented, or, as is more common for most efforts, only tasks can be distributable. Although the Ada multitasking model intuitively seems to be the natural model for concurrency in the distributed environment, Cornhill has suggested that the Ada programs could and should be arbitrarily distributable [4,5]. Packages and individual blocks of code as well as tasks should be distributable. Ada programs should be developed using the Ada multitasking model for logical concurrency regardless of the underlying physical concurrency. The physical distribution of the Ada program can be specified using a distribution language which is input to the compiler with the Ada source code. The tradeoffs between the various alternatives must be carefully considered before an approach to implementing distributed Ada programs is selected.

The terms "task" and "process" are frequently used interchangeably. In this paper tasks are independent but interacting program components which execute in parallel. A process is an independent program execution and its context. It is the basic unit scheduled for execution by the operating system and represents

the execution of a program [6]. A single Ada program may contain many tasks, yet execute as a single process under an operating system which runs many concurrent independent processes.

2. DISTRIBUTED PROCESSING REQUIREMENTS

Distributed processing may be implemented on radically different types of architecture. Shared memory architectures have multiple processors sharing one or more global memories, or processors with local memory may be interconnected by message-oriented communications links. These message-oriented links may be strictly point-to-point, or they may have a broadcast or multi-drop capability.

Distributed systems may interface by messages, remote procedure calls, rendezvous, monitors, or shared variables to name a few of the approaches. At the most fundamental level there are only two classes of communication technology: those which copy data, e.g., an I/O-oriented approach, and those which reference shared data, e.g., using global (shared) memory. Interrupts may provide an asynchronous change of control flow to signal an event or message exchange, or the message exchange may be referenced synchronously within the process. Various communications methods may be layered on these basic technologies to provide different access techniques and control flow structures.

There are a number of desirable capabilities for a distributed processing system. These include:

- support for multiple readers;
- support for multiple writers;
- support for multiple independent message streams;
- asynchronous input, i.e., a non-blocking receive;
- asynchronous output, i.e., a non-blocking send;
- support for locking shared memory data structures for mutual exclusion;
- control over the scheduling discipline;
- access to a system clock;
- an interval timer which can asynchronously signal events;
- control over the distribution of processes on the network;

- fault detection and damage assessment.
- transparent fault tolerance;
- support for multicast;
- support for broadcast;
- security features such as encryption.

Although a feature may be desirable, it may be impractical to implement for performance reasons. There is a trade-off between performance and desirable features such as fault tolerance that is application dependent.

Considerations in the area of distributed processor management [7] include

- the allocation of processors: static, dynamic, user-defined, or automatic
- the atomicity of distribution: packages, tasks, or procedures
- possible remote operations: rendezvous, activation/termination, remote procedure calls, and global variables
- remote dependencies and exception handling
- general network topics such as encryption, protocols, and fault handling.

There are two extremes to using Ada in the distributed environment. One extreme transparently distributes Ada programs across the distributed environment. There are, however, inherent problems in the Ada model of concurrency when applied to the distributed environment. Although solutions may exist to many, if not all, of these problem, the performance penalties extracted may render the multitasking model impractical.

The other extreme follows the multiprocessing model in which separate sequential programs are developed which can be concurrently executed. Ada programs can use the techniques developed during years of research into distributed processing issues. The drawback is the loss of the advanced software engineering concepts intrinsic to the Ada concurrency model. The advantage is that the system designer is explicitly aware of the underlying distributed architecture.

3. DISTRIBUTED ADA PROGRAMS

Implementing the Ada concurrency mechanisms on a distributed system is not a straightforward matter. A number of issues which are of concern in distributed processing are not adequately addressed by the Ada multitasking capabilities, and a number of assumptions implicit in the definition of Ada tasks do not necessarily hold true in the distributed environment [8]. The implementation of physical concurrency may place restraints on the design of logical concurrency, for example, the use of global variables in the absence of shared memory. These constraints may be driven by both performance and feasibility restrictions.

The Ada Language Reference Manual indicates that multitasking can be transparently implemented on a distributed system [9]. Several features of the language, however, imply a single-memory system [10]. Although entry calls and accept statements are the primary means of synchronization of tasks, and of communicating values between tasks, the use of shared variables is also described in the language specification. Global variables imply a common memory. Access objects as rendezvous parameters imply a common memory. Many distributed systems, however, do not support shared memory.

Connection management is not supported. There is no suitable language construct to represent a node in the network; therefore distribution of the program cannot be handled from within the language.

All possible constraints on synchronization cannot be expressed using the rendezvous primitives. The rendezvous provides synchronization points for communicating tasks. Ada provides only synchronous communication (other than through shared variables). Asynchronous communication implies nonblocking sends and receives. This problem can be addressed by inserting a buffering task (also called agent tasks [11]) between the sender and receiver, but this may impose a significant degree of overhead.

Conditional entry calls imply that it can be quickly established whether the called task has executed the accept and that the queue is empty. Since the delay statement would be used if a "timed" response was adequate, conditional

entry calls will be used by tasks that cannot tolerate excessive delay. When the called task is on a remote node, timely response becomes a critical - and unquantified - issue.

Timed entry calls may imply a potential race condition between the rendezvous and the timeout. Should timeout be measured from the calling or accepting tasks involved in the rendezvous? If from the calling task, as seems logical, race conditions may occur where the calling task has aborted a rendezvous that the accepting task has initiated. If from the accepting task, are the semantics of the language preserved?

An interval timer capability is not supported. The Ada delay statement guarantees a minimum delay; the actual time interval can be arbitrarily longer than that specified by the delay statement and still satisfy the semantics of the delay.

Packages STANDARD and SYSTEM need multiple definitions in a heterogeneous distributed environment. This implies an interface to the network presentation layer and possibly a canonical representation of entities. Assumptions in target-dependent representation clauses may imply a specific system in a heterogeneous environment.

Fault tolerance is not addressed [12,13]. What happens when a distributed system has a processor crash? Can a "shadowing" task take over the functionality of a "dead" task? Can the system degrade gracefully? Ada makes no explicit provision for continuation. When a processor failure occurs, services and data may be lost; tasks may be permanently suspended on the surviving processors; and the context of some tasks may be lost. A replacement task cannot assume the name of the task it is intended to replace, and there is no provision for redirecting the communication path used before the failure.

Using Ada in the distributed environment may require extensions to the language [12], which, by definition, means the language is no longer Ada. If there are restrictions on what Ada constructs are distributable, i.e., shared variables are not permitted, can the compiler be validated? If the compiler generates

full Ada for a uniprocessor and a subset for a distributed target, can it pass validation as a derived compiler based on its uniprocessor mode? The issue of validating Ada compilers for distributed environments is not resolved at this time. By one philosophy each host/target pair must be validated. Although validation policy has evolved beyond that point, the question of a distributed architecture on validation is debatable.

One way of avoiding the entire validation issue and the problems of distribution is to not support physical concurrency in the compiler. Traditionally, distributed computer systems have applied some variation of multiprocessing.

4. ADA AND MULTIPROCESSING

Multitasking enters an area traditionally considered the province of the operating system. In attempting to define the Ada host/target environment, the Stoneman document specifies an Ada Programming Support Environment (APSE) to provide a framework for writing Ada programs [2,3]. Examining the boundaries between an APSE and the target system reveals several related areas: the Ada language, the run-time system, the operating system, and the programming support environment. The Kernel Ada Programming Support Environment (KAPSE) provides access to the operating system routines. An APSE provides a multiprocessing host environment for software development. The target's run-time system provides the virtual machine on which an Ada program runs. Issues which are not specified in the Ada language definition and must be addressed by the run-time system include the broad categories of job scheduling, memory management, security, fault tolerance, and distributed systems.

In an APSE tool composition implies a need for one Ada program to invoke another completely separate Ada program [14]. Since the Ada language has no such facility, support for tool composition must be supported by the KAPSE. An INVOKE_PROGRAM primitive can suspend the calling program, execute the called program to completion, and then resume the calling program. The primitive can also be non-blocking.

The Common APSE Interface Set (CAIS) attempts to provide a standard host environment for developing host tools [6]. The CAIS includes both process

initiation and interprocess communication mechanisms. The distributed environment, however, is a deferred topic under the proposed MIL-STD-CAIS. If the CAIS is extended to address the distributed host environment, applying the same mechanisms to the distributed target is straightforward. The distinction between host and target systems is largely artificial for this instance.

Research in distributed systems has explored many avenues for implementing concurrency including multiprocessing and integrated approaches similar to multitasking. The most significant problem with the Ada and multiprocessing approach is that it discards the software engineering concepts central to the language. The strong type checking and information hiding capabilities integral to Ada are seriously compromised by using message-oriented mechanisms.

Part of the Ada design philosophy is that modularity and abstraction are well-proven means to overcome natural human limitations in dealing with complexity. Should a system designer be aware of an underlying distributed system? To provide the time-critical performance required by the application it may be essential that the designer have explicit understanding and control of the distributed system. In other systems which do not have real-time requirements it may be irrelevant to the system designer how the underlying hardware implements the design.

A compromise between these approaches is to develop a pre-processor which takes as input a single multitasking Ada program and outputs multiple Ada programs (one per node) that use site-specific mechanisms for interprocessor communication [15,5]. Such a hybrid approach would provide a portable tool for building distributed Ada programs. The pre-processor could be written in Ada, accept an Ada program as its input, and output a set of Ada programs which could then be compiled for the appropriate target. The pre-processor could use a standard software communications package which provides a basic message-oriented networking capability. This package could be reimplemented for a given distributed architecture without changing the pre-processor. Proxy tasks could then be used to handle rendezvous between nodes.

5. CONCLUSIONS

In a real-time embedded target environment the expense and complexity of implementing an efficient Ada compiler for a given distributed architecture may be a comparatively minor issue. A distributed system could be built incorporating solutions to the problems with distributed multitasking which have been discussed. Whether such a system could provide adequate response in a hard real-time environment is questionable unless the compiler is customized for a specific distributed target.

Using the multiprocessing approach requires knowledge of the distributed architecture at system design. This is not necessarily bad, but current work in designing distributed computing systems emphasizes deferring a binding of the system to the architecture. The host environment, as opposed to the target environment, requires an interprogram communications mechanism to aid in tool composition. The extension of such a mechanism for the distributed environment can provide a portable distributed processing capability.

Combining multitasking and multiprocessing may be the most promising approach, but the basic problems in distributing Ada programs must still be addressed. For real-time environments the designer must remain aware of the performance implications of design decisions.

6. REFERENCES

1. Paulk, M.C., "Interprocess Communication in Ada," Proceedings of IEEE Southeastcon '84, April, 1984, pp. 33-35.

2. "Requirements for Ada Programming Support Environments: Stoneman," Department of Defense, February 1980.

3. Buxton, J.N., and Druffel, L.E., "Requirements for an Ada Programming Support Environment: Rationale for Stoneman," COMPSAC 80, October, 1980, pp. 66-72, reprinted in The Ada Programming Language: A Tutorial, ed. S.H. Saib and R.E. Fritz, IEEE Computer Society Press, 1982, IEEE Catalog No. EHO 202-2.

4. Cornhill, D., "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," Proceedings of the 1984 IEEE Conference on Ada Applications and Environments, pp. 153-162.

5. Cornhill, D., "A Survivable Distributed Computing System for Embedded Application Programs Written in Ada," ACM Ada Letters, Vol. 3, No. 3, Nov/Dec 1983, pp. 79-86.

6. Proposed Military Standard Common APSE Interface Set (CAIS), 31 January 1985.

7. Lomuto, N., Rajeev, S., and Grover, V. "The Ada Runtime Kit (ARK)," IEEE Real-time Systems Newsletter, Vol. 2, No. 2, Summer 1984, pp. 27-33.

8. Paulk, M.C., "Problems with Distributed Ada Programs," Proceedings of the 5th Phoenix Conference on Computer and Communications, 1986, pp. 396-400.

9. ANSI/MIL-STD-1815A, The Ada Programming Language Reference Manual, American National Standards Institute, 1983.

10. Dapra, A., et al, "Using Ada and APSE to Support Distributed Multimicroprocessor Targets," ACM Ada Letters, Vol. 3, No. 6, May/June 1984, pp 57-65.

11. Hilfinger, P.N., "Implementation Strategies for Ada Tasking Idioms," Proceedings of the AdaTEC Conference on Ada, October, 1982, pp. 26-30.

12. Knight, J.C., and Urquhart, J.I.A., "On the Implementation and Use of Ada on Fault-tolerant Distributed Systems," ACM Ada Letters, Vol. 4, No. 3, Nov/Dec 1984, pp. 53-64.

13. Knight, J.C., and Gregory, S.T., "A Testbed for Evaluating Fault-Tolerant Distributed Systems," submitted to Proceedings of the 14th Conference on Fault-Tolerant Computing Systems, June, 1984.

14. Stenning, V., Froggatt, R.G., et. al., "The Ada Environment: A Perspective," IEEE Computer, Vol. 14, No. 6, June, 1981, pp. 26-36.

15. R.A. Volz, A.W. Naylor, et al., "Some Problems in Distributing Real-time Ada Programs Across Machines," Ada in Use, Proceedings of the Ada International Conference, May 1985, issued as ACM Ada Letters, Vol. 5, No. 2, Sept/Oct 1985, pp. 72-84.