REAL-TIME PERFORMANCE OF DISTRIBUTED ADA PROGRAMS

Mark C. Paulk
Unisys Defense Systems
Huntsville, AG 3580, USA

Logical concurrency is the natural approach to solving many problems; physical concurrency is frequently the most efficient method of implementing the solution. Recognizing this, the Department of Defense's Ada programming language provides multitasking as an integral part of the language specification. Concurrent Ada 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.

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 Ada language definition do not necessarily hold true in the distributed environment. The implementation of physical concurrency may place restraints on the design of logical concurrency.

The specifiers of the Ada run-time environment have a high degree of freedom. There are at least 40 issues which are not specified in the Ada language definition, including the broad categories of job scheduling, memory management, fault tolerance. and distributed systems, which will have a significant effect on run-time performance. Considerations in the area of distributed processor management 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; and general network topics: encryption, protocols, fault handling, and so forth.

What influence does the distributed environment have on the design and implementation of logical concurrency? How do Ada programs interface with the network environment when the communications primitives are intrinsic to the language? Does the distributed environment necessarily imply modifications to the compiler? Should a distributed communcations package be used? In view of Ada's strong typing requirements, how should the interface to a communications package be specified? What is the effect of a unique distributed environment on compiler validation by the Department of Defense?

There are two extremes to using Ada in the distributed environment. One extreme extends the Ada language to the distributed environment. There are, however, inherent problems in the Ada model of concurrency when applied to the distributed environment. Although there are solutions to these problems, the performance penalites extracted can be excessive.

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

Global variables and access objects as rendezvous parameters imply a common memory. Tasks using global variables communicate by means other than rendezvous. Even if the application does not explicitly share variables across tasks, there may be side-effects due to the encapsulation of global data inside shared packages.

Time can be a perplexing problem in any distributed system. Package CALENDAR implies a single sense of time within a program. In the distributed environment, however, the local clocks may differ significantly. Clocks tick at different rates, providing a skew as well as a synchronization problem. Although a global clock will be available in some distributed environments, in most there will be only a rough agreement between time in the distributed portions of the program.

All possible constraints on synchronization cannot be expressed using the rendezvous primitives. 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 delay. When the called task is on a remote node, timely response becomes a critical — and unquantified — issue. Timed entry calls are the only Ada contruct which place an upper bound on the time within which an action must take place.

Packages STANDARD and SYSTEM need multiple definitions in a heterogeneous distributed environment. If data interfaces occur only when rendezvous parameters are passed, then the appropriate conversions can be applied as appropriate by the underlying network protocols.

Fault tolerance is not addressed. 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? The Ada language makes no explicit provision for continuation. There is no provision for detection of processor failures unless the run-time system supplies it. 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.

Should these problems with concurrent Ada programs in the distributed environment be addressed transparently? Although that may be an admirable goal from a purist point of view, the effect on system performance can be drastic. The "transparent solution" may enter areas where the language standard must be interpreted. In the near future, distributed Ada systems will probably rely on communication packages to address the problems of real-time distributed processing.

Using a communications package implies 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. It may be feasible to build a translator that takes as input a single multitasking Ada program and outputs multiple Ada programs (one per node) that use site-specific machanisms for interprocessor communication. Not all of the problems described could be fully solved using this approach. In particular, the issues involved with system time would require detailed analysis.