USING PARALLEL ADA IN THE IMPLEMENTATION OF SIMULATION AND TRAINING SYSTEMS

By Gary Croucher and Don Law
Encore Computer Corporation
Fort Lauderdale, Florida 33313

ABSTRACT

As simulation and training systems become more complex, vendors must rely on the ability of the target system to meet the processing needs of the application. The ever increasing complexity of today's training systems has exceeded the processing capabilities of many single CPU systems. As an alternative, more and more vendors are now considering multi-processor systems.

The Ada language is the logical choice as a software environment for developing these large scale applications. The Ada tasking mechanism can be extended to schedule and distribute tasks over multiple processors. This resulting parallel Ada runtime is capable of executing Ada tasks in parallel, while upholding the rules of the Ada language.

The decision to migrate to a parallel Ada environment is an important one involving many important factors. The intention of this study is to provide the applications developer with an insight into the specific features available in parallel Ada environments, and which features will be most useful throughout the life cycle of his application. With this information, the decision maker should be able to determine if a parallel Ada target environment is worth considering, and which types of parallel environments provide the individual features most essential to the success of his application.

INTRODUCTION

There are a number of parallel Ada environments currently available in the marketplace, each with its own set of functionality and restrictions. Unfortunately, the term "parallel" has been left open for interpretation, resulting in a conglomerate of vastly different Ada development systems being labeled as parallel. For example, it is uncertain whether an Ada development environment is to be labeled parallel because it compiles an Ada program in parallel, executes an Ada program in parallel, or compiles and executes the Ada program in parallel.

The relevant issue is not whether a particular Ada environment is or isn't parallel, or even which Ada environments have the most sophisticated parallel features. The major consideration for the developer of a large scale Ada application should be whether or not a particular parallel Ada implementation has the necessary features and performance to support and enhance the execution of his or her individual application throughout its life cycle. The ideal parallel environment for one developer may be totally inadequate for a second developer. Developers need to evaluate parallel Ada environments with respect to their own applications to determine which parallel implementation maintains the most useful set of features.

Parallel Ada environments may be capable of compiling and/or executing the target application in parallel. While parallel compilation can significantly increase the speed at which the application can be built, it has no effect on the execution speed of the application. Parallel execution (the ability of the Ada runtime system to distribute the Ada application across multiple processors to be executed in parallel), on the other hand, can have a significant impact on application performance. The remainder of this study focuses specifically on parallel runtime features that impact the execution of the application.

Clearly, the focal point for the developer is to first understand the features and attributes of parallel Ada runtimes in general, then to determine the advantages

and disadvantages of specific parallel features with respect to his application. With the information in this paper, the developer is more likely to make a more intelligent, cost-effective decision about the future direction of his or her application.

The remainder of this study is devoted to presenting a general set of features needed in parallel Ada environments used for the real-time execution of simulation and training systems. Specific advantages and disadvantages associated with some of these features, and how these features impact large-scale time critical applications, is also investigated.

PARALLELISM AND APPLICATION SUITABILITY

Before the developer decides to make the transition to a parallel Ada environment, he or she must examine the application to determine which types of parallel features would be most advantageous. The answers to the following questions typically provide good insight into how an application will perform in a parallel environment.

• Does the application utilize a large number of Ada tasks, or are runtime calls (from the underlying operating system) used to schedule and invoke concurrent code segments?

If a particular Ada runtime is parallel because it distributes Ada tasks in parallel to run over multiple processors, but the target application utilizes proprietary runtime calls, or for whatever reason, does not use Ada tasking, then this parallel implementation will not benefit the application at all. Further, applications that do utilize Ada tasks must be designed so that Ada task execution can take place in parallel. For example, an application that uses Ada tasks, but serializes execution with rendezvous, synchronization, or other methods, reaps no benefit from task parallelism.

• Does the application consist of multiple Ada tasks within a single Ada main program, or is it made up of multiple main programs communicating with each other?

If the functionality of the target application is divided up into multiple Ada main programs instead of Ada tasks, then a parallel Ada software environment is not necessary. In this case, the developer need only build and execute each main program encompassing the application using a non-parallel execution environment and assign each of the resulting executable programs to a different processor. The disadvantages to this approach are that the rich flow of control between constructs within a single Ada program is lost. Additionally, the developer is now forced to consciously partition the application himself. Finally, this approach is very inflexible. The degree of parallelism is limited to the number of main programs, and the addition or reduction of processing power requires major application rework.

• Does the application require strict priority scheduling and preemption?

Most simulation and training systems rely on preemption of executing tasks by more urgent operations, such as the expiration of a delay or an interrupt. For these systems to run smoothly using an Ada tasking model, it is usually necessary to use a runtime environment that supports strict priority scheduling. Additionally, some runtimes may even allow very high priority tasks to be locked exclusively to individual processors.

• Are fast real-time features (typically found on sequential, uniprocessor based runtimes) necessary for application execution?

Some Ada implementations support real-time features that exploit the underlying computer system. There are Ada runtime environments available with good support of real-time programs. A parallel environment to be used for simulation and training systems should not only support real-time features, but should integrate them with the parallel tasking model. These high speed features are essential to time critical applications and must not be overlooked. Some of these features may include task connections to external interrupts, optimized context switch times, and user configurable device drivers. If the application under consideration utilizes any of these relevant features, then it behooves the developer to consider parallel runtimes that incorporate some, or possibly even all, of these unique real-time features.

If the answers to the above questions indicate that a parallel implementation may be capable of providing increased application performance and additional functionality, then the next step is to evaluate typical features of parallel Ada runtimes to determine which features will be most beneficial.

In addition to systems that are in the coding or maintenance phase of their life cycle, new systems should also be evaluated for whatever type of parallel Ada execution environment is needed. An ideal situation would be for the developer to study the list of parallel features presented in the next section before beginning the high-level design of his application. In this way, the application could be designed to fully exploit the parallel features of whichever parallel Ada implementation the developer chooses.

PARALLEL RUNTIMES, FEATURES, AND FUNCTIONALITY

Assumptions

- In this section, the scope of the discussion is limited to the execution environment in which the unit of parallelism is the Ada task.
- The computer systems discussed are multiprocessor systems.
- The runtime execution environment conforms to the Ada Language Reference Manual, MIL-STD-1815.

System Architecture

The choice as to what features will be offered on a given parallel Ada implementation is ultimately driven by the underlying system architecture. System hardware can range from dual to massively parallel processors, with very tightly coupled, tightly coupled, or loosely coupled CPU/memory configurations.

A very tightly coupled multiprocessor system, where all of the processors have access to all of memory (see figure 1), is the most widely accepted architecture for parallel Ada implementations. With this architecture, task synchronization and task distribution can be controlled by the runtime kernel, which is accessible by all processors. Additionally, the rules for governing the Ada language, including the rules that dictate the scope of visibility of variables, can be adhered to without restriction, because the entire range of memory is visible to each processor in the configuration. If anything less than the entire range of memory is visible to any of the processors, then the Ada implementation either must run the risk of placing restrictions on the rules governing Ada or introduce additional overhead to support inter-processor communication. This additional overhead is typically not acceptable when introduced into large-scale time critical applications.

The applications developer should seriously consider the number of processors available on the underlying hardware before deciding whether or not to utilize a parallel Ada implementation on top of this hardware. If the processing needs of the application could increase over the life cycle of the application, then the developer should ensure that additional processors can be easily added to the existing hardware, or that compatible systems with augmented numbers of processors are available. Of course, the parallel Ada implementation should be capable of utilizing all of the available processors in the system.

Lightweight Thread Implementation

If a parallel Ada runtime environment is implemented on a multiprocessor bare machine, then the Ada environment is free to schedule the processors according to Ada semantics. One of the most desirable features of a bare machine implementation is its freedom from any constraint of an operating system scheduler. The disadvantages of a bare machine approach are that the machine must be dedicated to a single application and resources typically found on a general purpose operating system are not available.

The scheduling advantage of the bare machine approach may be captured in an Ada environment implemented on top of a real-time operating system if that operating system supports a lightweight thread model. The term "lightweight thread" refers to the ability to create additional threads of execution in a program without the overhead of creating a new process. A lightweight thread is an execution entity that is added to an existing program. It does not have its own set of files, its own address space, or the other items associated with a process. A lightweight thread consists only of a stack, a set of registers, and a program counter. Figure 2 illustrates Ada tasks implemented on traditional processes versus Ada tasks implemented with lightweight threads.

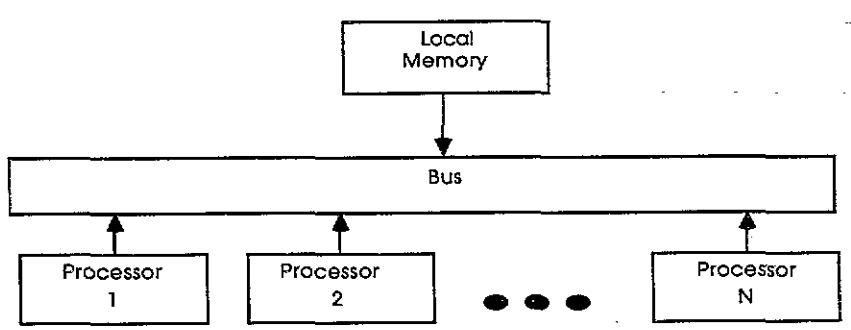

Figure 1: Multiple processors sharing one common memory

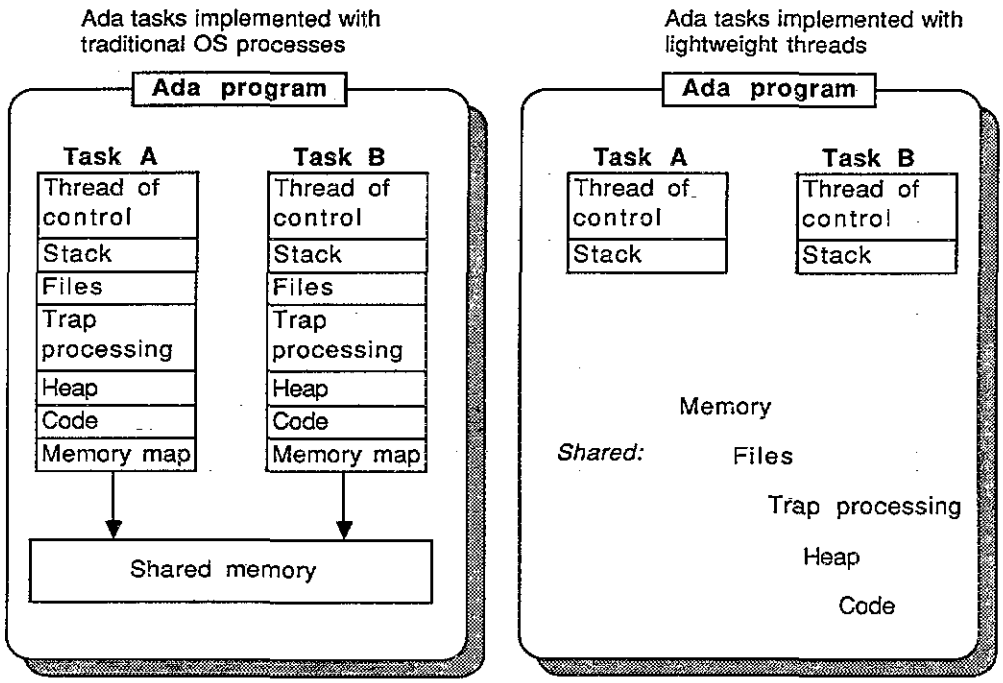

Figure 2: Traditional processes versus lightweight threads

The lightweight thread model is exactly the paradigm that is needed for Ada execution. Ada programs consist of multiple parallel threads of execution within a single program, all of which share open files, address space, trap processing, heap, and so forth.

When parallel Ada is used for a simulation system, it should be implemented on a system that matches the Ada tasking model to the computer being used. The lightweight thread model of a real-time operating system matches the needs of a multitasking Ada program.

Fine Grain versus Coarse Grain Parallelism

There are varying degrees of parallelism, even among parallel Ada runtime environments. Within the environment itself are critical regions, where tasks modify data structures shared by other tasks. The modifications must occur atomically, requiring an internal locking mechanism that is invisible to the user.

The most coarse grain environment contains a single lock for all runtime environment data structures. Whenever any task is performing any runtime action, the lock is held. In other words, the entire runtime library is considered a critical region.

Fine grain parallelism is achieved with more locking mechanisms, with locks being associated with individual data structures. This allows multiple tasks to be performing runtime system operations at the same time. Instead of contention occuring at the level of the entire runtime system, it occurs only when more than one task attempts to modify the same data structure within the runtime system at the same time.

Figure 3 illustrates the different behavior of fine grain and coarse grain environments. Time progresses from left to right in the diagram. In the coarse grain system, all four tasks compete for the same critical region, the runtime kernel. When any task is executing in the kernel, all other tasks are blocked and have to wait for the lock to be released (shown as thick arrows). This represents wasted computer resources and should be minimized. In the lower portion of the diagram, a fine grain system allows different tasks to hold locks to different resources concurrently. The only time that a task is blocked is when it needs a resource that is held by another task, illustrated by task C requesting resource X while it is in use by task B.

For an application which is not parallel or that executes on a small number of processors, a coarse grain locking system may execute slightly faster than a fine grain system because the locking mechanism has some overhead associated with it. The coarse grain system only requires a single lock and unlock operation for any runtime system call. The coarse grain system may be a satisfactory solution when a small number of processors are being used, but as the number of processors increases, the contention for the critical region increases.

Fine grain locking environments reduce the contention and therefore the blocking time of Ada

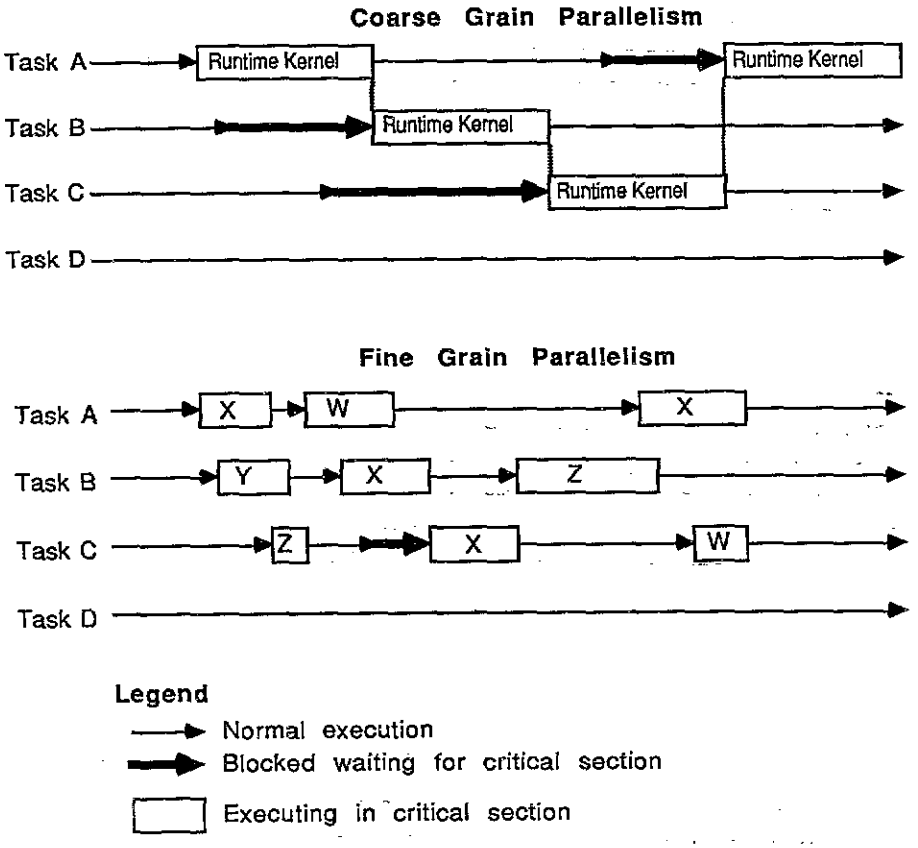

Figure 3: Coarse grain versus fine grain parallelism

tasks during runtime system operations. They are well-suited for applications that use larger numbers of processors because they use system resources more efficiently. A fine grain system may have slightly slower runtime operations than a coarse grain system because a larger number of locks must be processed for runtime operations, but the system will be more deterministic because of the reduced waiting times.

Task Priorities and Scheduling Methodology

The task priority mechanism of Ada has been a topic of much debate in the real-time systems arena. There have been several papers published on the topic that discuss some possible solutions.

A major issue is that of priority inversion on task entry queues. Ada requires that a server task execute at the priority of the client task during the rendezvous between these two tasks. This helps to reduce priority inversion, but doesn't prevent it. The priority of the server task is not affected by the client tasks which are queued waiting for a rendezvous with the server. A high priority client task may be queued waiting for a low priority operation to occur. This is priority inversion since the high priority task is held off by the execution of a lower priority task.

An example of priority inversion is shown in figure 4. The upper block shows the tasks involved, with the Fire_alarm task unable to obtain service from the Comm_server task because the Check_calendar task is using the rendezvous. The example shows how the urgent Fire_alarm task is prevented from continuing execution because of the unimportant Check_calendar task. Even worse, the unrelated Sort_database task may hold off the urgent completion of the Fire_alarm rendezvous indefinitely.

The lower portion of figure 4 shows the sequence of events leading to priority inversion from left to right. When the Fire_alarm task is scheduled to run, it gains control of the processor immediately because of its priority. However is is blocked awaiting the rendezvous with Comm_server, which is executing at priority 2 (since it is serving a task with priority 2). The rendezvous may be kept from execution indefinitely by the Sort_database task, which preempts the server because of its higher priority.

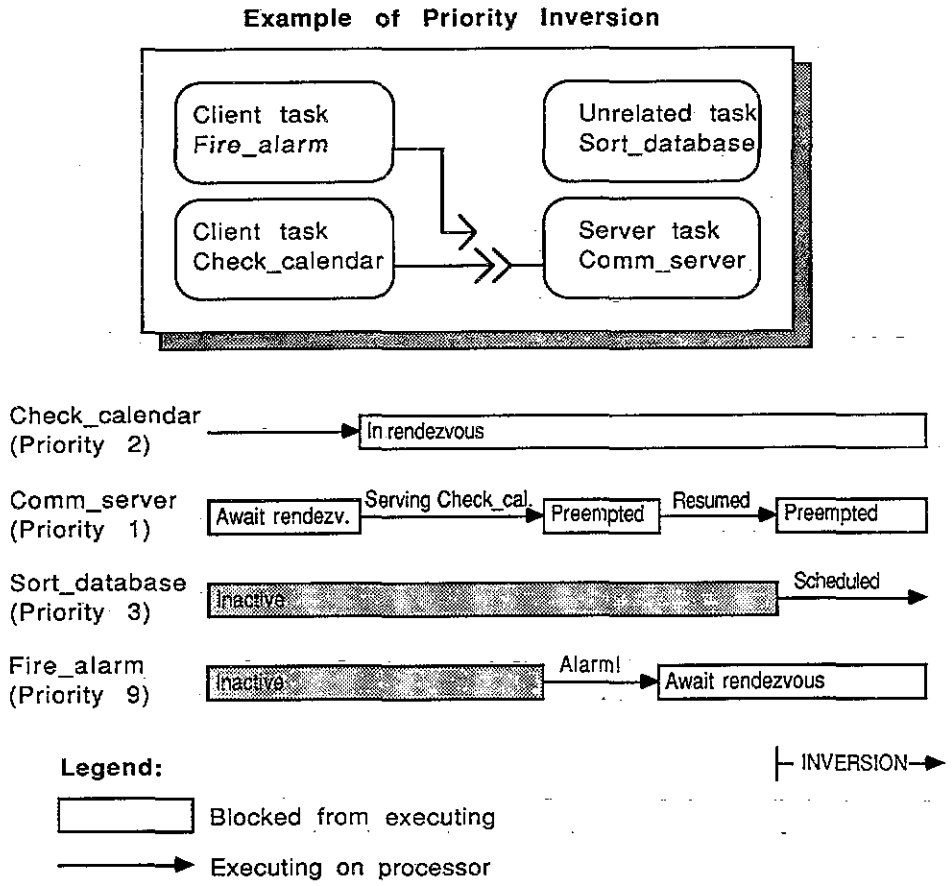

Figure 4: Priority Inversion

The problem of priority inversion is partially alleviated in a parallel Ada execution environment. With multiple parallel processors, more intermediate priority tasks must be present to starve out the low priority server. The inversion example pertains to a single processor system, but may be scaled to any number of processors by adding an unrelated medium priority task for each added processor.

The best approach when using of priorities in real-time simulation systems is to keep the model very simple. Ideally, it should be simple enough to facilitate formal proofs that priority inversion and other priority problems will not occur.

Dynamic Allocation of Processing Power

In a multiprocessor system, there is some point at which the simulation application developer specifies the number of processors that will be used to execute the target application. The longer the developer can defer this decision, the more flexible he can be with the parallelism of the application. If a non-Ada task allocation method is used, then the decision about the number of processors may have to be made as early as the design phase of the application. This is undesirable since the computer technology is likely to change before the deployment phase of the project. (see "Proprietary versus Generic Elements" above.)

The developer may be required to specify the number or processors at compile time or link time, with some directive to the compilation system or with some statements or pragmas in the source code. This requires a much lower turnaround time to change the number of processors used.

Some parallel Ada environments may allow the user to change the number of processors at the time when the program is invoked. The ultimate flexibility is the environment that allows processors to be added or removed while the program is executing. The latter feature is useful for systems that execute for long periods of time or have some fault-tolerant requirement.

Context Switch Times

Context switch times in a sequential Ada runtime are minimal, because all of the context switching and task rendezvous overhead is being carried out within a single executing process. By contrast, a parallel runtime will usually have longer context switch and task rendezvous times, because the parallel runtime

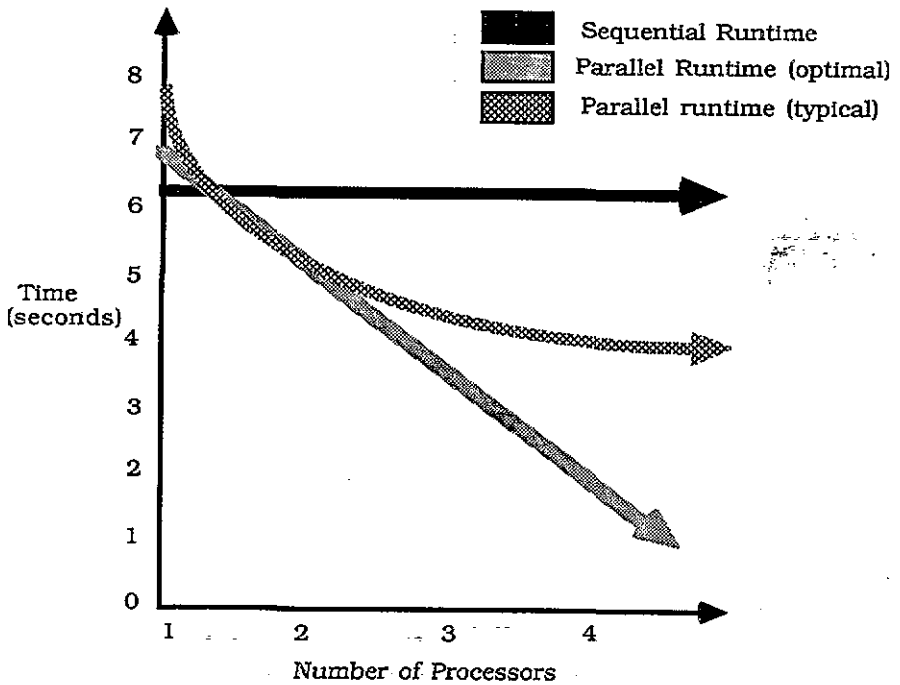

Figure 5: Execution speed of a sample application for 1 to 4 processors using a sequential and coarse grain parallel Ada runtime.

incurs the additional overhead of creating and communicating between independent threads of execution. Running on a single processor, the sequential Ada implementation will typically outperform its parallel counterpart, particularly if the application contains Ada tasks. However, as the number of available processors increase, the advantages of a parallel Ada implementation become clearer and clearer. Figure 5 is a graph of the execution of an Ada application containing a specified number of Ada tasks (in this case, more than four tasks were used). There is no interprocessor communication between tasks. The results are graphed for 1 to 4 processors when executing with a target load module linked to a sequential runtime and a parallel runtime.

The parallel runtime begins slightly slower for a single processor, due to the aforementioned overhead. However, as the number of processors increases, the parallel runtime's performance increases almost linearly. Speedup is almost linear because the application represents the optimal case (no interprocess communication and tasks immediately ready to run on available processors). A more realistic performance expectation is generated by the graph of the "typical" parallel runtime, which takes into account overhead for task initialization and communication, and assumes that there are not always tasks available to run immediately. As more and more processors are added to a parallel system, the benefits of the parallel runtime tend to diminish, until the number of processors exceed the number of available tasks and the line graph becomes horizontal.

Real-Time Features

It is difficult to implement a real-time simulation system using nothing but the generic features of Ada. However, to maximize the portability and maintainability of the system, it is best to keep the use of proprietary interfaces to a minimum.

It is good programming practice to isolate these interfaces into small areas of the system and to contain them within Ada package bodies where possible. This "information hiding" technique will make it easier to move the application to a different real-time system later in the life cycle.

The encapsulation of non-Ada real-time interfaces in packages may add some overhead to their use. The pragma "inline" may be used to reduce this overhead by causing the compiler to include the "wrapper" code directly in the calling procedure. Pragma "interface" may also be used in Ada package specifications to allow calling programs to call non-Ada interfaces.

Some simple real-time features of the system may be needed in some instances, especially where the Ada

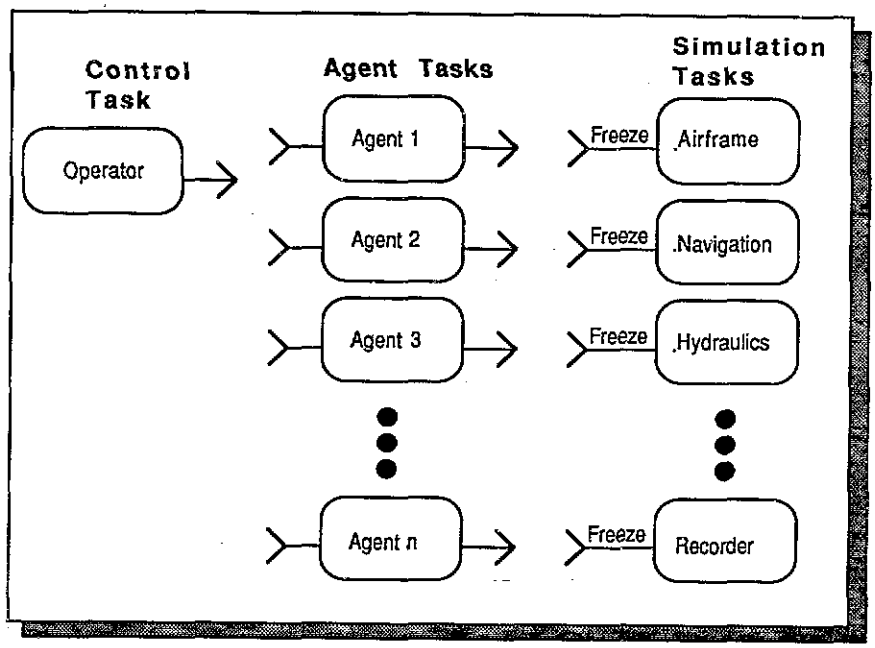

Figure 6: Solution to waitng for Freeze rendezvous

tasking model does not answer all the needs of the simulation system. One such example of this occured in a European nuclear power plant simulation and training system. The trainer required a "freeze" and "unfreeze" function, whereby the simulation could be frozen in time. Various simulation functions were needed while the freeze was in effect, so the implementation was to suspend the execution of a subset of tasks in the system. The tasks that were moving the simulation through time were to be frozen, while the maintenance tasks continued to execute.

The tasks to be frozen were synchronized by rendezvous from a master, but had "select" statements to allow the freeze operation to take place instead of the normal start of a frame. The task that performed the "freeze" operation was required to rendezvous with every task which was to be frozen. Each such rendezvous required the freeze operation task to wait until the task to be frozen reached the rendezvous before it could proceed and rendezvous with the next task to be frozen. The resulting lag caused the freeze operation to take too long to complete.

The solution to the problem was to introduce an "agent" task for every task that was to be frozen. The agent was always ready to accept the rendezvous from the freeze operation task and would then rendezvous with the task that was actually to be frozen. This solution accomplished the desired operation, but required significantly more tasks to be introduced into the system and required two rendezvouses to occur for every task to be frozen. The solution is shown in figure 6.

If a real-time supplement to suspend and resume an Ada task had been available, then the solution could have been much simpler. Using this real-time supplement, the freeze operation could have gone down the list of tasks to be frozen and issued a suspend on each one.

A basic set of real-time features should be included in a Parallel Ada system which is to be used for simulators and trainers. Besides the suspend and resume operations already mentioned, services to manage interrupts and timers, facilities for writing custom device drivers, and interfaces to real-time disk I/O are examples of other supplemental real-time services. These services could have a significant impact on the performance and success of the simulation system.

CONCLUSION

Parallel Ada development systems are an important step in the maturation of the Ada language. What was once seen only as research projects are now maturing into commercially available systems.

Some research into the high-level design of the simulation system should occur before the supporting Ada development system is selected. The implementor needs to know what questions to ask in the selection of a parallel Ada execution

implementation. There are many issues that are not apparent at the beginning of a project, but can be brought to light by looking at other similar projects that have used Ada.

The use of the Ada language does not guarantee parallelism, portability, or maintainability in itself. Such goals must be incorporated in the design of the simulation and training system. The parallelism of Ada is yet another tool, that if properly evaluated and utilized, can provide the implementor with an additional resource to help achieve his execution goals.

ABOUT THE AUTHORS

Both Don Law and Gary Croucher are senior members of the technical staff at Encore Computer Corporation in Fort Lauderdale, Florida. Both authors hold a Bachelor of Science in Computer Science and Mathematics from Furman University.

Mr. Croucher is the manager of Series 90 Ada development at Encore. He holds a Master of Science degree in Computer and Information Sciences from the University of Florida, College of Engineering. He as worked at Encore for seven years and with Apple Computers for two years prior to that.

Mr. Law has been with Encore since graduating from college seven years ago. He has worked on a prototype of the Common APSE Interface Set (CAIS) and the Ada Real-Time Executive project for Concept computers. He is currently the project leader of the Micro-ARTE project for Series 90 computers and working on a Master of Science degree from Florida Atlantic University and Carnegie-Mellon University.