GUIDELINES TO THE SELECTION OF ADA-BASED MULTIPROCESSOR COMPUTER SYSTEM CONFIGURATIONS FOR USE IN TRAINING AND SIMULATION

Don Law and Gary Croucher
Gould Inc., Computer Systems Division
Ada Development
6901 West Sunrise Boulevard
Fort Lauderdale, FL 33313

ABSTRACT

A multiprocessor real-time Ada-based environment is becoming increasingly necessary to support the growing class of complex simulation and training applications. The Bare Machine Ada runtime, or BMA, project at Gould C.S.D. is an effort to produce such an environment. The results of this development effort are providing valuable insights into the mechanisms required to adequately and efficiently support Ada on multiple processors.

Variations of a distributed real-time environment are evaluated with respect to very tightly coupled processors running with a single (common) local memory, tightly coupled processors equipped with a range of common memory, and loosely coupled processors containing no common memory. Finally, we consider methods for obtaining a processing system able to distribute Ada tasks freely over multiple processors. Each processor in the system maintains its own local memory, but entire ranges of local memories shadow each other's contents, forming a single common address range with a minimum of contention.

INTRODUCTION

As the machinery or vehicles for which large scale simulations, monitors, training facilities, or control programs are written become more complex, so too does the application program. Programs which once required a single task have become increasingly complex and, in general, can no longer be considered as a single design problem. To produce a truly efficient and open ended design, the problem must now be broken down into a set of logically related subproblems, each of which is relatively independent of the others and has the potential to execute with a degree of parallelism, so that real-time response becomes a reality. This has led vendors to a multiprocessor bare machine solution for an Ada target computer system. Typically, a simulator will divide the simulated functions up into tasks executed on separate processors. The bare machine approach isolates the application from proprietary operating systems, which simplifies programmer training, maintenance, and future porting efforts. For a more detailed description of the advantages of a bare machine Ada implementation, refer to (1).

The implementor of a simulation system needs to consider many factors that will affect the application. A few of the items that need to be considered are:

1) Synchronization: The Ada rendezvous is the mechanism to synchronize the tasks within the program. The speed and limitations of interprocessor rendezvous should be considered.
2) Task Distribution: The limit to the number of processors that are available for Ada task distribution and the ability and complexity of task distribution and the impact of task assignment.
3) Data Sharing: The method of establishing common data areas between tasks executing on different processors.

These considerations are evaluated on very tightly coupled, tightly coupled, and loosely coupled multiprocessor computer systems. We consider the possibilities with respect to the possible hardware capabilities. The user must also consider whether or not the vendor has taken advantage of the possibilities.

Definitions

Configuration - the particular set of hardware that the program is to be executed on.
LRM - Language Reference Manual for Ada; ANSI/MIL-STD-1815A; the specification that defines the Ada language.
Processor - A piece of hardware that can execute the instructions of a program, also called a "CPU."
Task - an Ada entity of execution which may proceed in parallel with other tasks in the same program. A *task* is not to be confused with a "job" or "process" in an operating system sense.

VERY TIGHTLY COUPLED SYSTEMS

System Architecture

A *very tightly coupled multiprocessor system* is a system where the processors share all of memory, all of the memory is cacheable, both processors execute with cache coherency, and either processor may prompt the other. The processors on the bus may not all have the same capabilities. A typical block diagram of the system appears below in Figure 1:

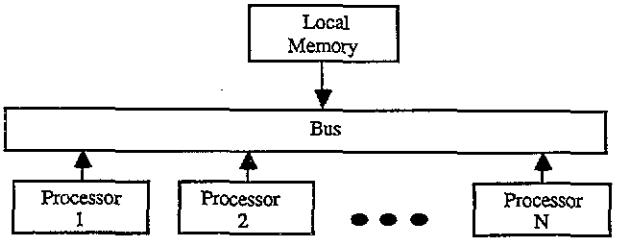

FIGURE 1. MULTIPLE PROCESSORS CONTENDING FOR LOCAL MEMORY ACCESS.

If the vendor's runtime kernel provides the capability to distribute the Ada tasks over the multiple processors in the system, then the application can use the facilities designed into the language to solve the problems of synchronization, distribution, and data sharing. Typically, the program will initially be loaded onto one of the processors on the target

system, which will begin elaborating the main program. As tasks in the program are elaborated, they are scheduled on the other processors in the system by the runtime kernel. Figure 2 shows how the memory will be used in the system.

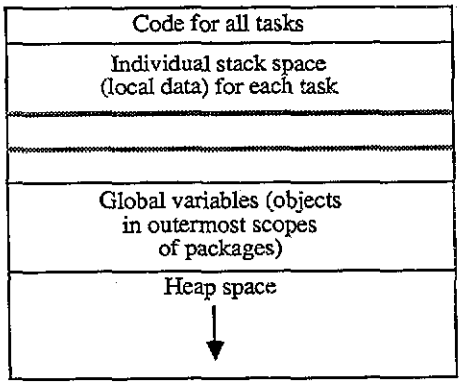

Figure 2: Memory map for a very tightly coupled system.

Task Synchronization

The well-defined standard Ada synchronization mechanism, rendezvous, can be used for inter-task communication. There is no need to use mechanisms outside the language. The logic for doing so is provided in the vendor runtime kernel. Not only does this make the program easier to maintain, but it will also be easier to port to a different bare machine environment. The vendor may or may not provide for interprocessor preemptive scheduling and still conform to the LRM. Preemptive scheduling is normally implemented with traps or interrupts between the processors in the system. There is no limitation on the rendezvous construct. Any task may rendezvous with any other task in the system.

Task Distribution

Task assignment to processors can be dynamic; that is, tasks can migrate among the processors at runtime without losing visibility to data, local or global. Since all of memory is equally accessible by all processors, the task is not impacted by a migration, except for scheduling overhead of prompting the other processor. The code for the task is equally visible to all processors. This would allow dynamic load balancing. The user need not be concerned with memory segmentation since all memory is cached by the processors. When a task is migrated to a different processor, its local data (stack) will still be in the same memory with the same address, so no memory copy or address translation is required. The tasks may migrate to any of the processors that are available on the bus.

Data Sharing

There are three categories of data that may be shared among tasks: local data, global data, and heap data:

Local data. Local data are considered to be any objects that are normally only visible to the task at hand. This is normally data that are allocated on the stack for that task. If multiple tasks are executing a procedure concurrently, each will have its own set of local data items for that procedure. However, the language allows a task to pass a local data item as a parameter through rendezvous to another task which may be executing on a different processor. It is common for Ada implementations to pass these parameters by reference instead of by value. This is not a problem under this particular configuration because the reference pointer will point to the same data on the other processors.

Global data. Global data are considered to be any objects that are neither local data or from the heap. Typically these are objects residing in package bodies and specifications but outside the declarative part of any subprograms (in the outermost scope). In the configuration under consideration, all tasks may share global data at any level of scoping. Since all memory is essentially global, all package data is visible to all tasks in the system. There is no performance penalty for the sharing of data because the shared memory operates at full speed.

Heap data. Heap data are data that are dynamically allocated from free memory, known as access types. The very tightly coupled system allows all tasks to share heap data. Parameters may be passed through rendezvous (or global data areas) which contain pointers (access types) to heap data. All of the heap area is common to all processors, so an access value on one processor will point to the same item on a different processor.

Possible limitations

The Ada runtime kernel provided by the vendor may not have the capability to distribute the tasks of a single program across the multiple processors of the system. In this case, the user has more of a burden, especially in the areas of task synchronization, task distribution, and data sharing. In this alternative, there are typically multiple Ada programs executing in the system, each on a different processor. The user is required to use facilities outside of the language to share data. The complexity added by having multiple separate programs can be minimized by making the number of programs equal to the number of processors. All tasks within each program execute on the associated processor.

Synchronization between processors cannot be done with the standard Ada mechanisms, but must be implemented by the user with the shared memory area and some semaphore facility.

Task distribution among the processors must be static. To move a task from one processor to another, the program must be edited and recompiled.

There are several different approaches to establishing global data sharing. It is desirable to preserve the Ada strong typing so that all units reference the same data in the same way. A library common to each program in the system may be established which contains packages that declare global data. These data may be placed in specific locations in the system memory using representation clauses or some other implementation dependent facility. Then all tasks in each program would see the same global data in the same area of memory.

The sharing of local data is not applicable since interprocessor rendezvous is not possible. The sharing of heap data is only possible if the runtime kernel can establish a common heap area from which all programs can allocate dynamic objects.

TIGHTLY COUPLED SYSTEMS

System Architecture

As the application becomes larger and more complex, the number of concurrent tasks needed to support the application in real-time will also increase. This will require additional processor support. The methods chosen to increase the number of processors need to be evaluated very carefully, however, as there are several key issues that must be resolved, some of which are:

1) Speed: Can processors be added so that they are tightly coupled with a minimum of bus contention to provide a real-time response?
2) Inter-processor Communication using shared memory: Will processors share a single common memory or will each processor be provided with a separate local memory? If individual local memories, will all or a region of these memories be shared among processors (globally accessible)?
3) Support for Ada tasks and global data: Will tasks be allowed to migrate freely between processors, and will the user be responsible for placing global data in a globally accessible memory region (if global regions are used), or will this be the responsibility of the system?

The simplest method of increasing the number of available processors is to add processors to the system bus, which is just an expansion of the system described in the previous section. This preserves all of the capabilities described above, however the number of processors that share the system bus must be kept low to avoid bus contention and memory contention. A major bottleneck occurs as processors are added and contend for the system bus. As the number of processors grows, contention delays increase accordingly, severely impacting processor speed and response time. Simply using faster processors for the application may be prohibitively expensive, so some other solution must be found.

To help combat high levels of processor contention, processors are equipped with individual local memories along with a shared memory segment accessible to all processors. Processors can communicate through the shared memory range, and processor contention is reduced by providing processors local storage for local data and code, thus introducing contention delays only where there are loads and stores to global data (figure 3). Support for Ada task migration and global data, which will be covered later, becomes a more complex issue, since not all processors have access to all data. More importantly, however, is that the failure of any of the shared memory modules can inhibit the use of these modules by any or all of the participating processors. Furthermore, the demand for shared memory by the application can still cause a bottleneck on the shared memory bus.

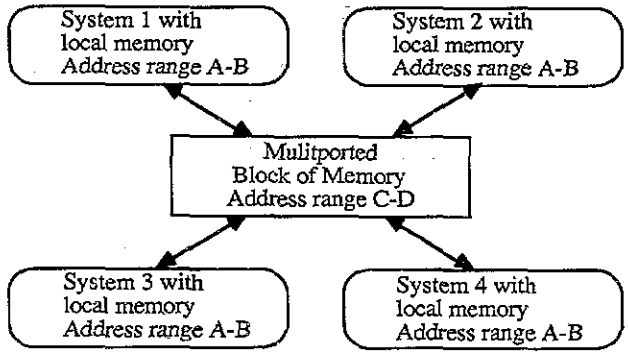

Figure 3. Traditional shared memory system

An alternative to the traditional common memory segments is a distributed, or reflective, memory system. The distributed memory system provides a real-time link between the local memories of nodes (nodes consisting of one or more processors in a very tightly coupled configuration) and enables them to shadow each others' contents in real-time, forming a single common address range. Any processor writing to any location in its reflected address range causes the real-time link to image that same transaction to all other processors maintaining the appropriate reflected address range. Likewise, any processor reading a shared location need only read the local copy of the global data from physical memory on its node's bus. Access contention is thus further reduced, since contention results only from processor writes ("stores") to global regions, but not from global reads of data or instruction fetches. Also, any one processor reads/writes in the logically shared portion and cycles at its full bandwidth without any additional access latency or contention timing latencies. Moreover, memory module failures, when they occur, are localized. Any one memory module failure causes only its associated node to fail, while the other nodes remain operational (figure 4). We will consider the implementation that does not provide cache coherency to the processors through the distributed memory bus. In order for all processors to register memory updates, they must not cache the reflected area of memory. This will cause some memory timing difference that must be considered.

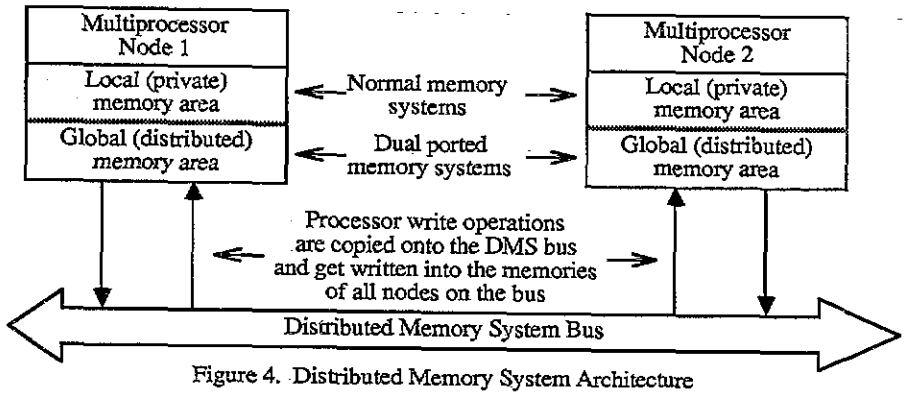

Figure 4. Distributed Memory System Architecture

The most desirable Ada runtime environment for this configuration is one in which the tasks of the program could be distributed over all of the processors connected by the distributed memory system. Then the application can be built using similar features discussed in the very tightly coupled systems but with some additional restrictions:

Task Synchronization

Like in very tightly coupled systems, rendezvous can be used for inter-task communication. Interprocessor preemptive scheduling may be available if cross-coupled interrupts or the equivalent are available between the nodes. Since all the tasks are in the same program, any task may rendezvous with any other task in the system.

Task Distribution

Task assignment to processors can still be dynamic amongst processors within the same node, but becomes impractical when taken across nodes in the system. It is not acceptable to keep all of the local data for tasks in the distributed memory region of a node because that would prevent the local data from being cached. This would cause too much of a performance impact on the tasks. Thus a task cannot move to a different node because the local stack for the task would disappear. There are implementation alternatives, such as copying the stack across to the new node, but that is not considered practical.

The code for each task on a node must be kept in private memory for reasonable performance. This allows the instruction stream to be cached by the processors. Multiple copies of the code will appear in each node, even though it is not part of the distributed area of memory.

Data Sharing

Again, three categories of data sharing among tasks may be examined: local data, global data, and heap data. (An explanation of each of these is in the previous section and will not be repeated here.)

Local data. As already mentioned, the local data for a task should be kept (by the runtime kernel implementation) in the area of memory that is not shared amongst the nodes. Furthermore, the user must take care not to pass a local data item as a parameter through rendezvous to another task which may be executing on a different node. If the implementation passes these parameters by reference instead of by value, then the internal reference pointer will not be valid on the processors of other nodes.

Global data. The logical place for global data to reside is in shared memory. Thus all tasks in the system always have access to the data no matter on which node they are executing. The disadvantage to this simple placement is a loss of speed because of the cache restriction. An alternative is the segmentation of the global data. There may be global data that is only shared by tasks executing on the same node. That data may be kept in the cached memory local to that node. The user must decide which packages should be distributed and which should be localized to particular nodes. A host development tool that can analyze the use of each global datum and segment out any global data that are isolated to one node would be useful. A second alternative to using the uncached memory is to identify read-only areas of global data and to keep those areas in the private memory of each node. Again, a host development tool to assist is in order.

Heap data. Like global data, heap space would logically be in shared memory. Yet for performance critical data, some restrictions may be applied to put some heap data in faster local memory. However, heap data is even more difficult to automatically localize than global data. As tasks in the system pass pointer parameters amongst themselves, it is not practical for the runtime to track what processor needs access to what heap area. Furthermore, the heap can become fragmented with localizable data. If the user cannot tolerate heap data in the slower memory, then an alternative is user-explicit allocation from a local memory heap. The runtime kernel is then responsible to maintain the multiple heaps. If this feature is available in a runtime kernel implementation, it might have the interface of a procedure call to select from which heap the executing task is to allocate dynamic data. Unless all of the heap is kept in the global distributed memory, parameters should not be passed through rendezvous (or global data areas) which contain pointers (access types) to heap data.

The difficulties of implementation

There are some drawbacks to implementing an Ada runtime kernel capable of distributing the tasks across nodes on a distributed memory bus. A few of the drawbacks that may prevent vendors from implementing this system are listed here.

A slower kernel. Since the tasks in the system are visible from all nodes, the kernel data structures, such as the task ready queue, must be kept in the global reflected area. This prevents these items from being cached, causing the basic tasking operations such as rendezvous to be slower. The time-critical portions of a simulation system cannot tolerate slower tasking operations. Also, rendezvous speed is often considered a critical metric when selecting a runtime environment.

Semaphore operations. A runtime kernel which uses the more flexible multiple master scheme for the scheduler requires atomic test-and-set operations to protect the global data. Such operations are not readily available on distributed memory systems and the workarounds can be cumbersome.

Tool requirements. For the system to work effectively, development tools must be implemented in addition to the runtime kernel itself.

User caveats. The scheme requires the user to manually take precautions to stay out of trouble, which is not consistent with Ada environments. Some of the performance solutions are dangerous from a software maintenance point of view for obvious reasons and should be avoided.

Clearly, these problems need a solution to provide the processing power of the tightly coupled system discussed plus the functionality, performance, and usability of the very tightly coupled system. A proposed answer to these problems is discussed in the final section of this paper.

Alternatives for the user

While vendors are working out the multiprocessor problems, the user has many of the multiprocessor bare machine Ada facilities available. The same solutions described above for running separate programs on each processor can be extended to the tightly coupled system. Each multiprocessor node can run a single Ada program and the programs can communicate over the distributed memory system. The

complexity added by having multiple separate programs is minimized by only having as many programs as there are nodes. All of the capabilities described in the very tightly coupled processor section are available in each of the nodes.

To allow consistent sharing of global data, the same technique described above of using a common set of packages for each program may be used in this configuration. The representation clauses (or equivalent) are used to place global data in the distributed memory range.

SUPPLEMENTAL COMMUNICATIONS NEEDS.

The distributed memory system provides the necessary real-time communication needs of a tightly coupled system. However, there are other applications which are not as time critical, that have less stringent communication needs. Among these needs are:

- the ability to communicate with systems beyond the range of distributed memory.
- an alternate method of node-to-node communication that does not require a dedicated region of local memory.
- a communication medium that provides built-in semaphore control.

This alternative form of communication is loosely coupled and most likely implemented via some type of packet switched mechanism such as a Local Area Network. In particular, the current implementation of bare machine Ada supports ethernet as the solution to its additional communications needs.

The LAN is not capable of addressing the real-time needs in the way possible with a very tightly coupled system or tightly coupled system, discussed above. However, a LAN does provide a supplemental form of communication while supporting some additional, useful features. Some of the more useful features provided by the LAN are:

- a means of communicating with outside systems beyond the local configuration.
- elimination of the global/local memory control needed for node-to-node communication. In fact, most LAN's, need little or no local memory control to operate. For example, ethernet on bare machine Ada maintains two megabytes of its own internal memory to temporarily hold incoming and outgoing frame information.
- support for transfer controls, such as collision avoidance mechanisms and automatic retries on unsuccessful frame transfers.
- allows all tasks to pass packet data at any level of scoping (by value).

It is not feasible to assume that a LAN can provide real-time support equivalent to a tightly-coupled shared memory system. The LAN is, however, a useful communications tool capable of providing the target application with an alternative level of processor communication.

THE FINAL OBJECTIVE - COMPLETE MULTIPROCESSOR SUPPORT FOR ADA

System Architecture

Consider the implementation problems of the tightly coupled system (discussed above) based on the distributed memory system. The difficulties arise from two restrictions on the shared memory area: 1) Those areas cannot be cached by the processors and 2) Those areas do not support atomic test-and-set operations.

A system which provides these two capabilities on the distributed memory system will support an Ada runtime environment that is:

- standardized, executing as a single logical Ada machine (bare).
- expandable, as processors can be added to the system without rapidly consuming bus bandwidth.
- parallel, with a two-level hierarchy of processors available to execute tasks within the program.
- consistent, with all tasks and data contained within a single program.
- high performance, using parallel memories to reduce central bottlenecks.

Such a system is depicted in Figure 5. Each node's memory represents a copy of the one logical memory available to the program.

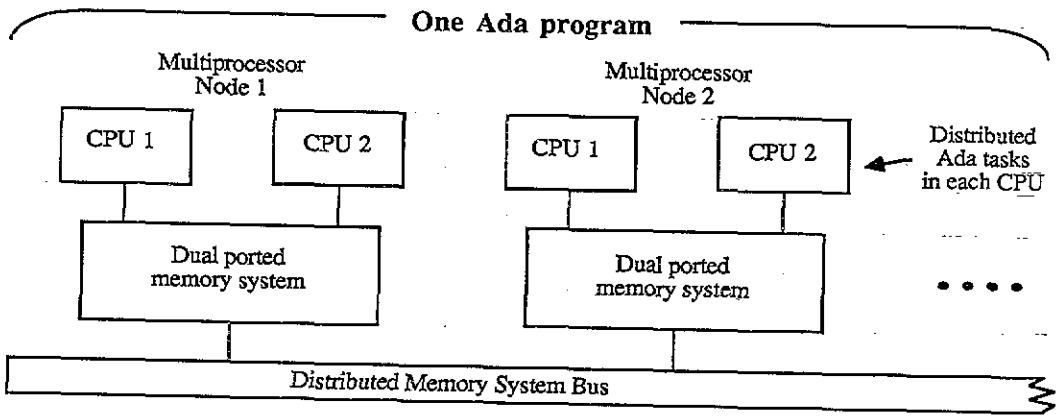

Figure 5. Complete mulitprocessor Ada environment

The system will normally operate with all of the memory for each node distributed, so that each node has the same memory image. Only operand stores from the processors cause traffic on the central bus. Such a system offers the features named in the full implementation of a very tightly coupled system discussed above:

Task Synchronization

The simple Ada rendezvous can be used for inter-task communication between any two processors in the system, whether or not they are in the same node. There is no need to use mechanisms outside the language.

Task Distribution

Task migration can occur freely among the processors at runtime since all processors have the same memory image.

Data Sharing

Local data may be freely passed through interprocessor rendezvous (by reference), because all tasks will have visibility to them. Global data are also readily available to all tasks, so that the user does not have to be concerned about where in memory it resides. There is only one heap that the runtime must maintain, and all access types will be globally visible.

CONCLUSION

Ada development environments increase the productivity of simulation and training system builders by making multiprocessor primitives available in the language. The most flexible and easily maintainable execution environments place large demands on the memory sharing capabilities of the underlying system hardware. This trend makes the development of the cache-coherent distributed memory system more desirable to Ada system vendors. The development of these target computer systems and the Ada runtime environment to fully utilize them will present the advantages of Ada to the high-performance real-time community. There are also alternative solutions that may be implemented in software, but more responsibility is put on the user as less language features are available.

REFERENCES

(1) Seymour, Burch, 1988, "Bare Machine Ada Solves Real Time Problems." Computer Design magazine, March 1, 1988, pp. 58-62.
(2) ANSI/MIL-STD-1815A, Ada Language Reference Manual.

About the authors

Mr. Don Law and Mr. Gary Croucher are senior design and development engineers with Gould Inc, Computer Systems Division in Fort Lauderdale, Florida. Both authors graduated from Furman University in Greenville, SC with a degree in Mathematics and Computer Science.

Don Law has been working at Gould since graduating from college five years ago. He has worked on several Ada projects including a prototype of the Common APSE Interface Set (CAIS). He has been a contributing member of the BMA project team since its onset two years ago and is now project leader for this ongoing effort.

Gary Croucher has been working as a software and systems engineer for the past seven years and is currently completing the final requirements for his Master of Engineering degree from the University of Florida. Gary has been with Gould the past three years, and has been working on the BMA project for the past year.