Ada Tasking for Parallel Computation: Supporting Fine-Grained **forall** Parallelism

James E. Hassett

Unisys Corporation
Electronic and Information Systems Group

Abstract

The kind of parallel computation that can be readily expressed with forall statements is not ordinarily well supported by implementations of Ada tasking because tasks demand too much memory to permit their use in large numbers. For a restricted class of tasks with properties well-suited to **forall** parallelism, we have developed implementation techniques that require only limited space for an array of tasks, regardless of the number of tasks in the array. Together with techniques for efficient task creation, initialization, and execution, this makes Ada tasking practical for certain kinds of fine-grained parallelism and highly-parallel computation.

1. Introduction

With the emergence of massively-parallel computers, it is becoming increasingly important for language implementations to provide efficient support for highly-parallel algorithms. While Ada is unusual among major languages in providing direct support for parallel programming through its tasking facilities [ALRM83], typical Ada task implementations are not well suited to fine-grained tasking or the use large numbers of tasks, due to the costs in both time and space associated with individual tasks. These costs arise from the power and generality of Ada tasking, so it is possible to provide more efficient support for certain restricted classes of tasks.

A common construct for explicit parallel computation is the **forall** (or DOALL) statement. A statement such as

```
forall I in 1 .. N do
   C(I) := A(I) + B(I);
end forall;
```

indicates that the body of the statement is to be executed `N` times, each time with the variable `I` assuming a different value in the range from `1` to `N`. The separate executions of the body may be carried out in any order or in parallel, with synchronization only after all executions have completed. While its use is limited to cases in which there are no dependencies among the executions of the body, this construct is conceptually simple, and can specify parallel execution from the subprogram level (through enclosed procedure calls) down to the statement level, as in the above example. It also allows expression of high degrees of parallelism, potentially calling for millions of executions of the statement body. This makes it useful for programming massively parallel systems. Since the body can contain conditional statements, it is suitable for programming MIMD (multiple instruction, multiple data) architectures.

Ada tasking can be used to specify parallel execution comparable to that of **forall** statements by defining a task type enclosing the body and creating an array of these tasks. But several problems arise in attempting to use tasks for fine-grained or highly-parallel programming:

1. Typical implementations of Ada tasks are prohibitively inefficient for the creation and execution of the tasks needed: execution overhead precludes efficient use of tasks for fine-grained parallelism; storage overhead precludes the simultaneous use of large numbers of tasks. Implementations may be unable to support even a few hundred tasks, let alone thousands or millions.

2. The individual tasks must be provided with the appropriate values of the **forall** index variable through explicit initialization, which can be a serial bottleneck [YEMI82].

3. It is syntactically more complex than the simple **forall** statement, resulting in awkward expression of certain algorithms.

Author's address: Unisys Corporation, P.O. Box 64525, MS U1J14, St. Paul, Minnesota 55164-0525;
Electronic mail: hassett@alice.sp.unisys.com

```
generic
   N: INTEGER;   -- Square matrix size
package MATRIX_PACKAGE is

   subtype MATRIX_INDEX is INTEGER range 1..N;
   type MATRIX is
         array (MATRIX_INDEX, MATRIX_INDEX) of FLOAT;

   procedure MULTIPLY(A, B: in MATRIX;
                      C: out MATRIX);

end MATRIX_PACKAGE;
```

Figure 1. Ada generic package specification for matrix multiplication

```
procedure MULTIPLY(A, B: in MATRIX; C: out MATRIX) is
begin
   for_all I in MATRIX_INDEX do

      -- Execute body for each row of result matrix:
      declare
         SUM: FLOAT;
      begin
         for J in MATRIX_INDEX loop
            SUM := 0.0;
            for K in MATRIX_INDEX loop
               SUM := SUM + A(I, K) * B(K, J);
            end loop;
            C(I, J) := SUM;
         end loop;
      end;

   end for_all;
end MULTIPLY;
```

Figure 2. Matrix multiplication with a hypothetical Ada for_all statement

This paper describes an Ada tasking optimization that can solve the first problem: providing space- and time-efficient tasks suitable for the kinds of parallel computation for which **forall** constructs are useful. The techniques are explained, a prototype implementation is described, and some performance data for example programs are discussed. Our program examples also illustrate approaches to the other two problems described above, but full discussion of them is beyond the scope of this paper.

2. A Matrix Multiplication Example

The Ada generic package specification shown in Figure 1 defines a data type representing an *N*-by-*N* matrix, and specifies a procedure for matrix multiplication. Figure 2 shows one way the matrix multiplication procedure could be implemented in an Ada-like language with a **forall** statement. The **forall** statement controls parallel computation of each row of the result, and a serial loop controls computation of each element of each row, carried out by the innermost serial loop. This program could be modified for finer granularity of parallel execution by replacing the outer serial loop statement with a **forall** statement. Note that data dependencies preclude the use of a **forall** statement for the innermost loop.

```
procedure MULTIPLY(A, B: in MATRIX; C: out MATRIX) is

   task type ROW_CALC is
      entry INITIALIZE(ROW: in MATRIX_INDEX);
         -- To receive index of row to compute
   end ROW_CALC;

   MULT_TASK: array (MATRIX_INDEX) of ROW_CALC;

   task body ROW_CALC is
      I: MATRIX_INDEX;
      SUM: FLOAT;
   begin
      accept INITIALIZE(ROW: in MATRIX_INDEX) do
         I := ROW;
      end INITIALIZE;
      for J in MATRIX_INDEX loop
         SUM := 0.0;
         for K in MATRIX_INDEX loop
            SUM := SUM + A(I, K) * B(K, J);
         end loop;
         C(I, J) := SUM;
      end loop;
   end ROW_CALC;

begin
   for I in MATRIX_INDEX loop
      MULT_TASK(I).INITIALIZE(I);
   end loop;
end MULTIPLY;
```

Figure 3. Matrix multiplication using Ada tasks

A comparable technique for using Ada tasks to express parallel matrix multiplication is shown in Figure 3. This implementation uses an array of *N* tasks, each of which computes the results for one row of the result corresponding to its own index. The entry INITIALIZE is used to tell each task what its row index is; each task then reads values from the input matrices A and B to compute the values needed, and places results in the matrix C.

This example illustrates the problems enumerated above. It is obviously more complex than the **forall** version of Figure 2, primarily due to the need to declare the task type, task array, and task body. Adapting it for finer granularity would introduce still more complexity. Typical implementations of Ada tasks would require the serial allocation of task control blocks and stack space for each task at the time of task array elaboration, resulting in a serial bottleneck and possibly exhausting available resources for large values of *N*. The initialization of the tasks through the INITIALIZE entry is another serial bottleneck for this example. While it may not be a serious problem in the version shown, a version using finer granularity (e.g., one task per result element) could be seriously hampered. It would require N^2 rendezvous to initialize all of the tasks, while the processing time for each task would be $O(N)$, so for sufficiently large *N*, the initialization could take much

```
package FA_BEACON is
   type FA_BEACON_TYPE is limited private;
   function FETCH_ADD (B: in FA_BEACON_TYPE)
            return INTEGER;
private
   task type FA_BEACON_TASK is
      entry FETCH_ADD(L: out INTEGER);
   end FA_BEACON_TASK;
   type FA_BEACON_TYPE is new FA_BEACON_TASK;
end FA_BEACON;

package body FA_BEACON is
   task body FA_BEACON_TASK is
      V: INTEGER := 0;
   begin
      loop
         select
            accept FETCH_ADD(L: out INTEGER) do
               L := V;   V := V + 1;
            end FETCH_ADD;
         or
            terminate;
         end select;
      end loop;
   end FA_BEACON_TASK;

   function FETCH_ADD (B: in FA_BEACON_TYPE)
            return INTEGER is
      RESULT: INTEGER;
   begin
      B.FETCH_ADD(RESULT);
      return RESULT;
   end FETCH_ADD;
end FA_BEACON;
```

Figure 4. The FA_BEACON package can be used for efficient task initialization.

longer than an individual task execution, limiting effective parallelism.

A solution to the initialization problem is available for parallel hardware that supports combinable fetch-add operations. In work associated with the NYU Ultracomputer, Schonberg and Schonberg [SCHO85] describe the use of *beacons* for task initialization. Beacons are tasks with an entry having the semantics of the fetch-add operation: returning the current value of a variable, and incrementing the variable by a specified value. An Ada implementation of a simple beacon package is shown in Figure 4 (For simplicity, the package shown self-initializes the beacon value to zero, and always increments by one, while our usual implementation provides an initialization entry and an increment parameter for FETCH_ADD). If an implementation using hardware fetch-add operations is provided, calls to the beacon FETCH_ADD entry can be handled without causing the caller to block. Any number of processors could then perform their task initializations simultaneously. The adaptation of matrix multiplication to use beacons is shown in Figure 5. It is actually somewhat simpler than the use of initialization entries in Figure 3.

```
with FA_BEACON;
procedure MULTIPLY(A, B: in MATRIX; C: out MATRIX) is
   BEACON: FA_BEACON.FA_BEACON_TYPE;

   task type ROW_CALC;

   MULT_TASK: array (MATRIX_INDEX) of ROW_CALC;

   task body ROW_CALC is
      I: MATRIX_INDEX;
      SUM: FLOAT;
   begin
      I := FA_BEACON.FETCH_ADD(BEACON);
      for J in MATRIX_INDEX loop
         SUM := 0.0;
         for K in MATRIX_INDEX loop
            SUM := SUM + A(I, K) * B(K, J);
         end loop;
         C(I, J) := SUM;
      end loop;
   end ROW_CALC;

begin
   null; -- Allow tasks to execute, await completion.
end MULTIPLY;
```

Figure 5. Matrix multiplication with initialization using beacons

The simple use of beacons can provide only a single integer value to each task, and the distribution of the values is arbitrary: a given task may receive any unique value from the effective range, not necessarily its own index. For some algorithms this is adequate, but for others it may be necessary to implement more complex initialization schemes to provide several values (possibly other than integers) by using auxiliary computations or arrays of initialization values.

3. Implementing Efficient forall Tasks

A major cost of using large numbers of tasks in typical Ada implementations is memory. The demand on memory arises from the need to maintain the complete state of each task if it is suspended. The state information required includes not only the contents of machine registers and data associated with control of the task (such as entry queues and priority information), but also the task's workspace: data typically contained in stack activation records, such as the contents of local variables. To allow tasks to make subprogram calls, a generous stack space must be allocated to each task when it begins activation, to be deallocated when the task completes. This is expensive when large numbers of tasks are live simultaneously. (We say that a task is *live* if it has started its activation, but has not yet completed its execution. With this definition, a suspended task is live, though not executing.)

Our optimization limits the cost of large groups of tasks (such as task arrays) by placing bounds on the number of tasks that are live at any given time. Tasks meeting certain

conditions described below can be scheduled in a manner that limits the number of such tasks that are simultaneously live. The conditions are based on the properties of typical **forall** bodies, which do not require synchronizations, delays, or other actions that would result in suspension. Each execution of a body can proceed without suspending itself, so that once a processor is assigned to it, it can execute to completion independently and release its resources.

To take advantage of this approach, we define the class of *autonomous* tasks, and describe a scheduling policy that yields bounds on live autonomous tasks. A task type (and each task object of this type) is called autonomous if

1. it has no entries,
2. its body makes no entry calls, and does not call (directly or indirectly) any subprograms that make entry calls (except for specially-implemented non-blocking entry calls such as the beacons described earlier),
3. its body contains no delay statements, and does not call (directly or indirectly) any subprograms that contain delay statements or could otherwise block execution (e.g., blocking I/O operations),
4. it creates no subtasks, and does not call (directly or indirectly) any subprograms that create subtasks.

Conditions 1 through 3 ensure that tasks of this type do not attempt rendezvous, delays, or other actions that would cause them to be suspended. Condition 4 ensures that such tasks will not wait for the activation or completion of any subtasks. These conditions could be weakened (e.g., to allow the task body to contain entry calls if they are never executed), but as stated they could be verified (on the basis of special pragmas) or possibly even detected by an implementation. With these conditions, any autonomous task can proceed from the beginning of its activation through completion without "voluntarily" suspending to await any actions of other tasks.

Large numbers of autonomous tasks can be efficiently executed by using a restricted scheduling policy:

> **Scheduling Policy for Autonomous Tasks:** Once a processor is allocated to an autonomous task t, it cannot be reallocated to any other task of the same type until t completes.

The conditions defining an autonomous task ensure that the completion of t does not depend on the actions of any task whose execution could be deferred by this scheduling policy, so the completion of an autonomous task will not be impeded. Since this policy only affects scheduling choices between tasks of the same type, and therefore the same priority, it does not conflict with the required semantics of task priorities.

An immediate result of this scheduling policy is the following bound on live tasks:

> **Live Task Bound for Autonomous Tasks:** The number of live tasks of any autonomous task type cannot exceed the number of processors.

Even if arrays of thousands of autonomous tasks have been declared, at most one task of each autonomous type can be live for each processor. Since only live tasks require stack space, large arrays of autonomous tasks will require only limited stack space. Autonomous tasks may be suspended to handle interrupts or to schedule higher-priority tasks, but only one task of each autonomous type could possibly be suspended for each processor, so the bound on memory allocation still holds. Unless the preempting tasks result in starvation, the autonomous tasks will eventually resume and complete. While such preemption will force context switching, the autonomous tasks can otherwise proceed very efficiently, without context switches. In multiprocessor systems, it may be possible for most processors to avoid context switching during the execution of autonomous tasks. Another technique for gaining efficiency is to immediately reallocate the processor and stack space of a completed autonomous task to any waiting task of the same type, to avoid the costs of deallocating and reallocating such resources.

Similar treatment can be given to a broader class of tasks, allowing the creation of subtasks. We define *dynastic* task types by imposing conditions 1 through 3 of autonomous task types, but changing condition 4 to the following:

> 4'. Any task created by a task of dynastic type T (or by a subprogram called directly or indirectly by the task) must be a dynastic task with priority not less than that of T.

Note that this allows a dynastic task to create tasks of any dynastic type, requiring only that the subtasks' priorities be at least as high as the parent's priority. Dynastic tasks need a modified scheduling policy, allowing such tasks to relinquish control to their "children":

> **Scheduling Policy for Dynastic Tasks:** Once a processor is allocated to a dynastic task t, it cannot be reallocated to any other task of the same type, except for (direct or indirect) subtasks of t, until t completes.

This scheduling policy is compatible with Ada semantics because of the subtask priority restriction for dynastic tasks. If task t is suspended while a subtask activates, the subtask's equal or higher priority will permit any other task of the parent type to be deferred.

The bound on live tasks must now be formulated differently, since tasks that recursively create new tasks of their own type could cause arbitrary numbers of dynastic tasks to be live simultaneously.

> **Live Task Bound for Dynastic Tasks:** The number of live tasks of any dynastic type, created directly by any single task, cannot exceed the number of processors.

This still results in the same bound on the number of live tasks in an array of tasks: one per processor.

While the properties of autonomous and dynastic tasks allow them to be handled with limited stack space, they also reduce the amount of task-related data needed by the Ada runtime system. Since these tasks have no entries and cannot engage in rendezvous, no entry queue data is needed for them, and their scheduling priorities cannot change, so no per-task priority information is needed. With some additional restrictions, it is possible to completely eliminate the need for task-specific information other than the internal state of the task itself during its execution. We define *simple* task types as dynastic task types with the following additional conditions:

1. The program contains no abort statements that identify tasks of this type, and
2. the program contains no references to the CALLABLE or TERMINATED attributes of tasks of this type.

As a result of these restrictions, the program cannot refer to any individual simple task, except to access certain attributes (SIZE, STORAGE_SIZE, and ADDRESS) that have the same value for any task of the type. Certain information required by the runtime system, such as the master of each task, is the same for all tasks which are components of a single array or other structure. Consequently, the runtime system does not need to retain any information about individual tasks of simple types, unless they are live. The task control blocks (TCBs) and task identifiers typically used in task implementations are unnecessary. A small data structure containing a few items (such as the unactivated task count, the amount of stack space to be allocated for each task, and the address of the machine code for the task body) is all that is needed for an array of simple tasks, regardless of the number of tasks in the array. An array of several million simple tasks could require only a few words of memory. For large enough N, an array of N simple tasks could require less memory than an array of N Booleans!

In addition to being space-efficient, simple tasks can be very efficiently executed because they avoid much of the context-switching needed to handle the suspension and resumption of ordinary tasks; they also contain few synchronization points at which the runtime system must check for abnormal status or for preemption by higher-priority tasks.

In spite of the restrictions imposed, simple tasks remain useful. Their properties are based on the normal properties of **forall** statement bodies, whose executions must be independent. While they are incapable of engaging in ordinary rendezvous to exchange information, they can access shared variables and data structures. By allowing simple tasks to call specially-optimized entries like the beacon FETCH_ADD, which never cause the caller to block, the beacon task-initialization technique can be applied to arrays of simple tasks, so that procedures such as that shown in Figure 5 can be efficiently supported.

```
generic
   with procedure PARALLEL_BODY(INDEX: in INTEGER);
procedure FOR_ALL(FROM, TO: in INTEGER);

with FA_BEACON;
procedure FOR_ALL(FROM, TO: in INTEGER) is
   J : FA_BEACON.FA_BEACON_TYPE;

   task type PARALLEL_TASK;

   TASK_ARRAY: array (FROM .. TO)
                        of PARALLEL_TASK;

   task body PARALLEL_TASK is
   begin
      PARALLEL_BODY(FA_BEACON.FETCH_ADD(J));
   end PARALLEL_TASK;

begin
   null;
end FOR_ALL;
```

Figure 6. Generic FOR_ALL procedure for specifying parallel execution

4. Hiding the Details: a Generic FOR_ALL Procedure

Ada's generic procedure facility can be used to encapsulate much of the complexity of using tasks for parallel programming. Figure 6 shows a generic FOR_ALL procedure that provides a straightforward means of specifying parallel execution. The use of this procedure to program the matrix multiplication example is shown in Figure 7. This approach completely suppresses the linguistic complexity of the tasks, but is still somewhat awkward due to the need to express each **forall** body as a procedure. This can be cumbersome for algorithms such as matrix multiplication, where the bodies are very simple, but it would be reasonable for more complex algorithms, where bodies may be large enough to warrant separate procedures anyway.

Variations on this basic FOR_ALL procedure could even be used to go beyond the semantics of the simple **forall** statement by varying more than one parameter to the body procedure. This could be done to distribute array indices to multidimensional arrays of tasks, for example. While such techniques can be very application-dependent, the use of generic procedures with procedure parameters still serves to encapsulate the details of tasking and task initialization. In fact, such generic procedures can be implemented to use completely different approaches to the use of tasks. One implementation of FOR_ALL could use the simple-task approach described above; another could use one task per physical processor, with each task making sequential calls to the body procedure to handle all parameter values; and a third, suitable for a single processor, could be completely sequential. This provides a method for encapsulating approaches suitable for different hardware architectures or runtime system implementations, to promote portability.

```
with FOR_ALL;
procedure MULTIPLY(A, B: in MATRIX; C: out MATRIX) is

   procedure ROW_MULTIPLY(ROW: in MATRIX_INDEX) is
      SUM: FLOAT;
   begin
      for J in MATRIX_INDEX loop
         SUM := 0.0;
         for K in MATRIX_INDEX loop
            SUM := SUM + A(ROW, K) * B(K, J);
         end loop;
         C(ROW, J) := SUM;
      end loop;
   end ROW_MULTIPLY;

   procedure FOR_ALL_ROWS is FOR_ALL(ROW_MULTIPLY);

begin
   FOR_ALL_ROWS(MATRIX_INDEX'FIRST,
                MATRIX_INDEX'LAST);
   -- Invoke ROW_MULTIPLY for each row index
   -- (in parallel).
end MULTIPLY;
```

Figure 7. Using the generic FOR_ALL procedure for matrix multiplication

5. Prototype Implementation

We have developed a prototype implementation of optimized simple tasks for the MIPS-by-four computer, a four-processor system built as a testbed for development of the Multitude architecture. The Multitude architecture is a shared-memory parallel processor architecture under development by Unisys. In this architecture, each processor resides in a node that also contains its local memory, I/O hardware, and an interface to a multistage interconnection network that provides access to the local memories of all other nodes. As in the NYU Ultracomputer [GOTT83] and IBM RP3 [PFIS85], the interconnection network provides support for combinable operations such as fetch-add. This allows any number of processors to simultaneously perform a combinable operation on the same memory location without blocking. The result is as if the operations had been performed in some particular serial order. Special features of the Multitude network minimize contention problems to provide efficient global memory access. The architecture is designed to be scalable to thousands of processors, to support fault-tolerance, and to allow efficient creation of many parallel threads of control (lightweight processes).

The MIPS-by-four computer contains four processor nodes, each with a MIPS Inc. R2300 CPU card, 8 megabytes of local memory, various I/O cards, and an interface to interconnection hardware that simulates the Multitude interconnection network through a set of four separate VMEbuses. An operating system kernel for the MIPS-by-four computer provides facilities that can efficiently create large numbers of lightweight processes. The essential mechanism for creating lightweight processes is the procedure *create_light*, which creates *N* lightweight processes, each of which will execute the same specified procedure with a different parameter value in the range from zero to *N-1*. The *create_light* procedure returns when all of the processes have completed execution.

Our prototype implementation of optimized simple tasks is based on the MIPS-Ada system (a version of the Verdix VADS system for the MIPS M/120 computer), and targets our MIPS-by-four computer. The code generated by the MIPS-Ada compiler is directly executable by the MIPS-by-four processors, so all that was necessary was to replace the runtime system with one designed for the MIPS-by-four computer, and based on the techniques described above.

Because we restricted programs to use only simple tasks, the runtime system implementation was straightforward. While a typical Ada implementation uses a record known as a task control block (or TCB) for each task, we use a single TCB for a group of tasks of the same type (such as a task array) created by a single declaration or allocator. Task creation involves setting up a TCB containing the information needed to execute the task: the amount of stack space required, the address of the task body, and a pointer to the activation record of the master (i.e., a static link). If a declaration or allocator creates two or more tasks, all of the same type (as for a task array), the compiler calls the runtime task-creation procedure once for each task. After the first, each of these identical tasks is created by simply incrementing a task count in the TCB. Ideally, task arrays should be handled differently by the compiler: instead of sequentially calling the task creation procedure, a special runtime system call should create a TCB with the proper task count. Since we did not modify the compiler itself for this prototype, we are not taking full advantage of the optimizations possible.

Activation of the tasks is handled by using the kernel's *create_light* facility to create a lightweight process for each task. The parent suspends while the tasks for a single TCB are executed, with as many available processors as needed participating in the execution of the tasks. After all of the tasks for a TCB have terminated, the next TCB (if any) for the same declarative part or allocator is handled. The parent resumes only after all tasks represented by these TCBs have terminated. This simple scheduling policy is unusual (and inappropriate for tasks in general, which may need to interact with parents or siblings), but it is consistent with proper Ada semantics for simple tasks, and can be handled very efficiently by the MIPS-by-four system.

We have also implemented a package based on the Schonberg beacon [SCHO85] to permit access to fetch-add operations. This package is similar to that shown in Figure 4, although it is implemented using a kernel function call for the fetch-add operation, instead of using a specially-optimized task. The generic FOR_ALL procedure of Figure 6 is also provided.

With these facilities, we have run various parallel Ada programs using arrays of up to 25,000 tasks. These programs include fast Fourier transform procedures, prime-number sieves, and matrix-multiplication procedures such as those discussed above. Performance data (see Table I)

indicate that the execution-time overhead for tasking is quite low. The procedure in Figure 7, applied to 40-by-40 matrices (so that 40 tasks are used), exhibits less than 1% overhead compared to a similar version in which a single task performs the matrix multiplication serially. A version using much finer granularity (one task per result element, or 1600 tasks) exhibits about 30% overhead, putting it near the limit for practical use of fine-grained parallelism for our current implementation. A similar program executed on the MIPS M/120 system, using conventional Ada tasking, experienced execution-time overhead in excess of 96%, and ran slowly due to page faults resulting from the large amount of memory (using default task size) allocated to the 1600 tasks.

6. Conclusions

We have shown that Ada tasking can be used to express the kind of parallel computation associated with the **forall** statement, and that efficient support for this style of tasking can be provided to make its use practical for highly parallel processing.

Our simple tasks, based on the properties of normal **forall** statement bodies, allow an optimized implementation that provides for time- and space-efficient support. While the conditions imposed on simple tasks are restrictive in comparison to full Ada tasking, they still permit a conceptually simple and natural expression of parallel computation. Together with the use of beacons for task initialization (in an architecture that supports combinable fetch-add operations), this approach makes the use of large arrays of Ada tasks practical even for relatively fine-grained parallelism.

Tasks	M/120	MIPS-by-Four Processors		
		1	2	3
1	0.7	10.6	-	-
40	-	10.6	-	-
1600	20.2	13.7	8.6	6.1

Table I. Execution times (in seconds) for 40-by-40 matrix multiplication using one task, 40 tasks, and 1600 tasks on the MIPS M/120 system, and on the MIPS-by-four computer. For the 1600-task version, times for 1 through 3 MIPS-by-four processors are shown.

Note: While the M/120 and MIPS-by-four computers use similar R2000 processors, their speeds differ greatly due to different clock speeds, data busses, and cache sizes. Also, data in the MIPS-by-four computer was interleaved among processor nodes, so that most data accesses were to remote memory, resulting in slower access. For these programs, instructions resided in node 0 memory, resulting in slow access for other processors; this was the primary reason for the highly nonlinear performance improvement with 2 and 3 processors.

By using generic procedures, the complexities of Ada tasks can be encapsulated to make the expression of parallel algorithms clearer.

Further work is needed to integrate our techniques for the implementation of simple tasks with more general methods suitable for full Ada tasking to provide for complete implementation. It may be possible to develop other related optimizations for additional classes of Ada tasks. Such approaches will make highly-parallel computers efficient hosts for Ada applications.

References

[ALRM83] U.S. Department of Defense, Reference Manual for the Ada Programming Language MIL-STD 1815A, Feb. 1983.

[GOTT83] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., and Snir, M. "The NYU Ultracomputer—Designing an MIMD parallel computer." IEEE Trans. on Computers **C-32**, 2 (Feb. 1983), pp. 75-89.

[PFIS85] Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A., and Weiss, J. "The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture." Proceedings of the 1985 International Conference on Parallel Processing (St. Charles, Ill., Aug. 1985) pp. 764-771.

[SCHO85] Schonberg, E., and Schonberg, E. "Highly parallel Ada—Ada on an Ultracomputer." Ada in Use: Proceedings of the Ada International Conference (Paris, May 14-16, 1985) Cambridge University Press, 1985, pp. 58-71.

[YEMI82] Yemini, S. "On the suitability of Ada multitasking for expressing parallel algorithms." Proceedings of the AdaTec Conference on Ada (Arlington, VA, Oct. 1982) pp. 91-97.