June 1987.

[SS85] Edith Schonberg and Edmond Schonberg. Highly Parallel Ada – Ada on an Ultracomputer. In *Ada in Use: Proceedings of the Ada International Conference*, 1985. Special edition of Ada Letters, 5(2), September 1985.

[SS88] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[Wol91] Michael Wolfe. Personal communication, April 1991.

[Yem82] Shaula Yemini. On the Suitability of Ada Multitasking for Expressing Parallel Algorithms. In *Proceedings of the AdaTEC Conference on Ada*, 1982.

[Ost89] Anita Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Technical Publications, 1989.

[PCF88] Parallel Computing Forum. PCF FORTRAN: Language Definition, August 1988.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype RP3: Introduction and Architecture. In *International Conference on Parallel Processing*, pages 764–771, 1985.

[Pol87] Constantine Polychronopoulos. Loop Coalescing: A Compiler Transformation for Parallel Machines. In *International Conference on Parallel Processing*, pages 235–242, 1987.

[Qui87] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* Series in Supercomputing and Artificial Intelligence. McGraw-Hill, 1987.

[Ric89] V.F. Rich. Parallel Ada for Symmetrical Multiprocessors. In *Distributed Ada*, pages 61–69, December 1989.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Sconberg. *Programming with Sets An Introduction to SETL.* Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[Shu87] Normal Victor Shulman. *The Semantics of Shared Variables in Parallel Programming Languages.* PhD thesis, New York University,

[Jha90] Rakesh Jha. Parallel Ada – Issues in Programming and Implementation. In *Fourth International Workshop on Real-Time Ada Issues*, pages 126–132. ACM Press, July 1990. Appeared in Ada Letters 10(9).

[Ken88] Ken Kennedy. Public Communication, 1988.

[KKLW84] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The Structure of an Advanced Vectorizer for Pipelined Processors. In Kai Hwang, editor, *Supercomputers: Design and Applications*, pages 163–178. IEEE Computer Society Press, 1984.

[Kri89] E.V. Krishnamurthy. *Parallel Processing Principles and Practice.* International Computer Science Series. Addison-Wesley, 1989.

[KS84] Philippe Kruchten and Edmond Schonberg. The Ada/Ed System: a Large-Scale Experiment in Software Prototyping Using SETL. *Technology and Science of Information*, 3:175–181, 1984.

[KW85] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors (Extended Abstract). *IEEE Transactions on Computing*, 11(10):236–240, October 1985.

[Law75] Duncan Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computing*, "C-24":1145–1155, 1975.

[Mit88] Thanasis Mitsolides. Ada Tasks in a Parallel Environment. New York University, April 1988.

[Hil91] Paul Hilfinger. Personal communication, June 1991.

[Hin88] Michael Hind. Parallelizing Compilers. New York University, June 1988.

[HS91] Michael Hind and Edmond Schonberg. Efficient Loop-Level Parallelism in Ada. In *TRI-Ada*, October 1991.

[Hum88] Susan Flynn Hummel. *SMARTS - Shared-Memory Multiprocessor Ada Run Time Supervisor.* PhD thesis, New York University, December 1988.

[Hun90] Geoffrey Hunter. The Fate of FORTRAN-8X. *Communications of the ACM*, April 1990.

[IBM88] IBM. Parallel Fortran Language and Library Reference. Technical report, International Business Machines, March 1988. Pub. No. SC23-0431-0.

[IBM89] IBM. VAST-2 for VS FORTRAN User's Guide. Technical report, International Business Machines, December 1989. Pub. No. SC26-4668-0.

[Ich84] Ada: Past, Present, Future: An Interview with Jean Ichbiah, the Principal Designer of Ada, 1984. *Communications of the ACM, 27(10), September 1984.*

[Inc81] Cray Research Inc. Fortran (cft) reference manual. Technical report, Cray Research, Inc., August 1981. Pub. No. SR-009, rev. H.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[Fra87] Gary Frankel. Improving Ada Tasking Performance. In *International Workshop on Real-Time Ada Issues*, Fall 1987. Ada Letters 7(6).

[GGK+83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computing*, February 1983.

[GLR83] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[Hel78] Don Heller. A Survey of Parallel Algorithms in Numerical Linear Algebra. *SIAM Review*, 20(4):740–777, October 1978.

[Hil82a] Paul N. Hilfinger. *Abstraction Mechanisms and Language Design.* MIT Press, 1982.

[Hil82b] Paul N. Hilfinger. Implementation Strategies for Ada Tasking Idioms. In *Proceedings of the AdaTEC Conference on Ada*, 1982.

[Hil90] Paul N. Hilfinger. Tasking Idioms and Enhancements in Ada 9X. Draft Version 1, November 1990.

Burke. The NYU Ada Translater and Interpreter. In *ACM-SIGPLAN Symposium on the Ada Programming Languages*, pages 194–201, November 1980. SIGPLAN Notices 15(11).

[DFSS89] Robert Dewar, Susan Flynn, Edmond Schonberg, and Norman Shulman. Distributed Ada on Shared Memory Multiprocessors. In *Distributed Ada*, pages 229–241, December 1989.

[Dri90a] Kenneth W. Dritz. Personal communication, August 1990.

[Dri90b] Kenneth W. Dritz. Tutorial on Parallel Programming in Ada. Presented at the Eighth Annual National Conference on Ada Technology, Atlanta, Georgia, March 1990.

[Ell86] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* 1985 ACM Doctoral Disseration Awards. MIT Press, 1986.

[EN90] Kemal Ebcioglu and Toshio Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 213–229. MIT Press, 1990. Selected papers from the Second Workshop on Languages and Compilers for Parallel Computing at Urbana, Illinois in August 1989.

[FH90] Lawrence E. Flynn and Susan Flynn Hummel. Scheduling Variable-Length Parallel Subtasks. Technical Report RC # 15492, IBM T.J. Watson Research Center, February 1990.

Ada for Tightly Coupled Systems. In *Distributed Ada*, pages 183–205, December 1989.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method for Computing Static Single Assignment Form. In *16th Annual ACM Symposium on the Principles of Programming Languages*, January 1989.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic Generation of DAG Parallelism. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[Cla87] Donald R. Clarson. Proposal for Adding Discriminants for Ada Task Types. *Ada Letters*, 8(5), September 1987.

[Col87] Howard B. Coleman. The Vectorizing Compiler for the UNISYS ISP. In *International Conference on Parallel Processing*, pages 567–576, 1987.

[Cyt85] Ron Cytron. Useful Parallelism in a Multiprocessing Environment. In *International Conference on Parallel Processing*, pages 450–456, 1985.

[Cyt91] Ron Cytron. Personal communication, March 1991.

[Dew90] Robert B. K. Dewar. Shared Variables and Ada 9X Issues. Special Report SEI-90-SR-1, SEI, January 1990.

[DFS+80] Robert B. K. Dewar, Gerald A. Fisher Jr., Edmond Schonberg, Robert Froehlich, Stephen Bryant, Clinton F. Goss, and Michael

[BDH+87] Mark Byler, James R. B. Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple Version Loops. *International Conference on Parallel Processing*, pages 312–317, August 1987.

[Beh90] Kurt Behnke. Implementation of Task Arrays with Single RTS Nodes. Ada/Ed Documentation, New York University, 1990.

[Ber88] Wayne Berke. ParFOR - A Structured Environment for Parallel FORTRAN. Technical report, New York University, April 1988. Ultracomputer Note #137.

[Blu81] E. K. Blum. Programming Parallel Numerical Algorithms in Ada. In J.K. Reid, editor, *The Relationship Between Numerical Computation and Programming Languages*, pages 297–304. IFIP TC2, North-Holland Publishing Company, August 1981.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. *Proc. 1985 International Conference on Parallel Processing*, pages 782–789, 1985.

[BN87] Thomas M. Burger and Kjell W. Nielsen. An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada. *Ada Letters*, 7(1):49–58, January 1987.

[Bur85] A. Burns. Efficient Initialisation Routines for Multiprocessor Systems Programmed in Ada. *Ada Letters*, 5(1):55–60, July 1985.

[CCB89] Lawrence Collingbourne, Andrew Cholerton, and Tim Bolderston.

[AK84] John R. Allen and Ken Kennedy. PFC: A Program to Convert Fortran to Parallel Form. In Kai Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, 1984. Also appeared in the Proceedings of the IBM Conference on Parallel Computers and Scientific Computations, 1982.

[AL89] Anders Ardo and Lars Lundberg. The MUMS Multiprocessor Ada Project. In *Distributed Ada*, pages 243–266, December 1989.

[Ard87] Anders Ardo. Real-Time Efficiency of Ada in a Multiprocessor Environment. In *International Workshop on Real-Time Ada Issues*, Fall 1987. Ada Letters 7(6).

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bab88] Robert G. Babb II, editor. *Programming Parallel Processors*. Addison-Wesley, 1988.

[BC86] Michael Burke and Ron Cytron. Interprocedural Dependence Analysis and Parallelization (Extended Version). Technical report, IBM Research, 1986. Report RC11794.

[BCF+88] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, Vivek Sarkar, and David Shields. Automatic discovery of parallelism: A tool and an experiment. *ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 77–84., July 1988.

Bibliography

[Ada83] American National Standards Institute. Ada Programming Language Military Standard. Technical report, American National Standards Institute, January 1983. ANSI/MIL-STD-1815A.

[ABC+87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, October, 1988, 5(5):617–640.

[ACK87] Randy Allen, David Callahan, and Ken Kennedy. Automatic Decomposition of Scientific Programs for Parallel Execution. In *14th Annual ACM Symposium on the Principles of Programming Languages*, pages 63–76, January 1987.

[Ada90] Ada 9x Project Report. Ada 9x Requirements. Technical report, Office of the Under Secretary of Defense Acquisition, December 1990.

[AG89] George S. Almasi and Alan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company Inc., 1989.

```
                    -- query which exception was raised.
                end if;
            end loop;
            -- We inform the caller about the exception(s)
            -- by raising a user-defined exception.
            raise minitask_exception;
        end if;
        return C;
    end matmult;
end matmult_pack;
```

```
with gen_minitask;
package matmult_pack is
   type matrix is array(integer range <>, integer range <>) of float;
   function matmult(A, B : matrix) return matrix;
end matmult_pack;


package body matmult_pack is
   function matmult(A, B : matrix) return matrix is
      C : matrix(A'RANGE(1), B'RANGE(2));
      subtype ij_range is integer
                  range 1 .. A'LENGTH(1) * B'LENGTH(2);
      procedure ij_body(ij : ij_range) is
         i, j : integer;
      begin
         i := (ij-1) div B'LENGTH(2) + A'FIRST(1);
         j := (ij-1) mod B'LENGTH(2) + B'FIRST(2);
         C(i, j) := 0.0;
         -- For each of B's rows
         for k in B'RANGE(1) loop
            C(i, j) := C(i, j) + A(i, k) * B(k, j);
         end loop;
      end ij_body;
      package ij_minitask is new gen_minitask(ij_range, ij_body);
   begin
      if not ij_minitask.success then
         for i in ij_range loop
            if not task_completion(i) then
               -- We can use task_exception(i) to
```

In a standard parallelization of matrix multiplication the outer two loops are executed in parallel. Although this can be achieved by nesting gen_minitask instantiations, we coalesce these two parallel loops, we choose not to nest uses of our idiom for the usual performance reasons; by combining the two outer loops into one loop we reduce the number of forks (i.e. gen_minitask instantiations) from A'LENGTH +1 to 1. This user-specified optimization is typical when dealing with nested parallel loops.

The body of the parallel loop is represented by the procedure ij_body. The parameter of the procedure plays the role of the iteration variable. After values for the i and j variables have been computed, we perform the standard computation for the innermost loop of matrix multiplication.

```
            end if;
        end;
    end loop;
    declare -- Second nest of loops.
        subtype ij_range is integer range 1 .. X'LENGTH(1) * X'LENGTH(2);
        procedure ij_body(ij : ij_range) is -- i & j loops combined
            i, j : integer;
        begin
            i := (ij-1) div X'LENGTH(2) + X'FIRST(1); -- Compute appropriate value of i.
            j := (ij-1) mod X'LENGTH(2) + X'FIRST(2); -- Compute appropriate value of j.
            X(i, j) := AB(i, j + AB'LAST(1) / AB(i, i);
        end ij_body;
        package ij_minitask is new gen_minitask(ij_range, ij_body);
    begin
        if not ij_minitask.success then
            raise minitask_error;
        end if;
    end;
end parallel_gauss_jordan;
```

A.3 Matrix Multiplication

Our final example is a package version of matrix multiplication. This package contains two visible entities: an unconstrained type declaration (matrix) and a function (matmult) that multiplies two matrices and returns the product.

The function matmult takes the two matrices, A and B as parameters and once again uses the predefined array attributes to determine the size of these matrices.

```
procedure parallel_gauss_jordan(AB: in out matrix; X: out matrix) is
   mult : float;
begin
   for j in AB'RANGE(1) loop -- Sequential loop
      declare
         subtype i_range is integer range AB'RANGE(1);
         procedure i_body(i : i_range) is -- Body of i loop (par)
         begin
            if i = j then
               return;
            end if;
            mult := AB(i, j)/AB(j, j);
            declare
               subtype k_range is integer range AB'RANGE(2);
               procedure k_body(k : k_range) is -- Body of k loop (par)
               begin
                  AB(i, k) := AB(i, k) - mult * AB(j, k);
               end k_body;
               package k_minitask is new gen_minitask(k_range, k_body);
            begin
               if not ij_minitask.success then
                  raise minitask_error;
               end if;
            end;
         end i_body;
         package i_minitask is new gen_minitask(i_range, i_body);
      begin
         if not ij_minitask.success then
            raise minitask_error;
```

of nested parallel loops. As no operations are performed inside of the outer loop and outside the inner loop, these loops are *coalesced* [Pol87]. This transformations eliminates the spawning of the N minitask families without decreasing parallelism.

```
procedure serial_gauss_jordan(AB: in out matrix; X: out matrix) is
   mult : float;
begin
   for j in AB'RANGE(1) loop
      for i in AB'RANGE(1) loop -- Parallel Loop
         if i /= j then
            mult := AB(i, j)/AB(j, j);
            for k in j + 1 .. AB'LAST(2) loop -- Parallel Loop
               AB(i, k) := AB(i, k) - mult * AB(j, k);
            end loop;
         een if;
      end loop;
   end loop;
   for i in X'RANGE(1) loop -- Parallel Loop
      for j in X'RANGE(2) loop -- Parallel Loop
         X(i, j) := AB(i, j + AB'LAST(1))/AB(i, i);
      end loop;
   end loop;
end serial_gauss_jordan;
```

```
            if procnum_minitask.success then
               for i in num_tasks_type loop
                  pi := pi + sum_array(i);
               end loop;
            else
               raise minitask_exception; -- Minitask error condition.
            end if;
         end;
      end calculate_pi;
```

A.2 Gauss_Jordan

This program solves a linear system $AX = B$, where A is an N by N matrix, X is an N by M matrix, and B is an N by M matrix. The method used is Gauss-Jordan elimination as described by Heller [Hel78].

Two versions of the algorithm are supplied. The first performs Gauss-Jordan elimination by executing each of the parallel loops sequentially. The second procedure utilizes the gen_minitask idiom to express the inherent loop-level parallelism of the algorithm.

Both procedures are passed two matrices, A and X, as parameters. For convenience, matrices A and B are passed as one parameter, AB, (B extended to the right of A). Upon conclusion of the algorithm, out parameter matrix X contains the solution. The bounds of these parameters are obtained by using the predefined attribute operations, 'RANGE and 'LAST. In computing X, the algorithm overwrites matrix AB.

One optimization is performed in the gen_minitask version in the second set

```
with text_io; use text_io;
with gen_minitask;
procedure calculate_pi is
   pi : float; -- The final result
   num_tasks, num_intervals : integer; -- Input
   sum_array : array (num_tasks_type) of float;
   package int_io is new integer_io(integer); use int_io;
   package my_float is new float_io(float); use my_float;
begin -- Acquire number of threads and intervals.
   put("Enter number of tasks: "); get(num_tasks);
   put("Enter number of intervals: "); get(num_intervals); new_line;
   declare -- Call subroutine concurrently to do work.
      subtype iteration_range is integer range 1..num_tasks;
      procedure iteration_body(iterate : iteration_range) is
         local_sum : float := 0.0;
         current_interval : integer; -- Current interval
         x : float; -- Value of x used for integration
      begin
         current_interval := iterate;
         while current_interval <= num_intervals loop
            x := (float(current_interval) - 0.5) / float(num_intervals)
            local_sum := local_sum + interval_width * 4.0 / (1.0 + x * x);
            current_interval := current_interval + num_tasks;
         end loop;
         sum_array(iterate) := local_sum; -- Deposit local sum.
      end iteration_body;
      package procnum_minitask is new
                  gen_minitask(iteration_range,iteration_body);
   begin
```

the package instantiation, a check is performed on the family of minitasks. If the value true is returned, the local sums of each minitask are totaled and reported.

precision of the computation; it defines the number of intervals between 0 and 1. The second variable, num_tasks, specifies the amount of parallelism to be used in the computation. As the number of instructions required to compute an interval is small, it is desirable to have num_intervals larger than num_tasks.

The intervals are assigned in the following manner: each minitask receives one of the first num_tasks intervals. After computing the area for its first interval, a minitask acquires its next interval by adding num_tasks to its initial interval. This process is repeated until num_intervals is surpassed. For example, the first minitask computes intervals 1, num_tasks + 1, 2· num_tasks + 1, ..., $\left\lfloor \frac{num_intervals-1}{num_tasks} \right\rfloor \cdot$ num_tasks + 1.

Current_interval takes on values spanning from 1 to num_intervals. As it should be representative of the range of values from current_interval - 1 to current_interval, 0.5 is subtracted to obtain an average. To determine the value of x, current_interval - 0.5 is scaled down according to the number of intervals, i.e. it is divided by num_intervals. This value is substituted in $\frac{1}{1+x^2}$ to determine the y value for this interval. This value is used to obtain the area represented by this interval; it is the product of the y value and the interval_width. Lastly this product is added to the local_sum for this minitask.

As the computation for this interval is complete, an attempt is made to compute the minitask's next interval; current_interval is incremented by num_tasks. A check is performed to see if this new value represents a valid interval. If it does, this process continues. However, if there are no more intervals for this task to compute, the local_sum value is stored into the shared array sum_array in the appropriate index.

When the main program regains control after the **begin** statement following

Appendix A

Examples

In this chapter we illustrate the expressive power of the gen_minitask idiom by presenting three examples. Each of these examples utilizes loop-level parallelism, making them suitable candidates for our idiom. The examples we present are:

calculate_pi: A program that computes π by integrating $\frac{1}{1+x^2}$ between 0 and 1.

gauss_jordan: A program that performs Gauss-Jordan elimination on a linear system of equations.

matrix_mult: A package that contains a matrix multiplication function.

A.1 Calculate_Pi

This program computes π by integrating $\frac{1}{1+x^2}$ over the interval from 0 to 1 [Bab88]. The computation is based on the value of two variables given by the user and is given on page 185. The first variable, num_intervals, specifies the

beneficial. Once this representation is obtained, care must be taken to ensure that the algorithms used to obtain the control dependence graph and static single assignment form are still applicable. Work on this problem as it applies to a parallel version of FORTRAN is currently being done by Wolfe and his colleagues [Wol91]. Since Ada tasking semantics provide a more powerful mechanism than those of the FORTRAN being studied, devising a representation for Ada should prove to be more challenging.

architectures, such as distributed memory machines, vector machines, or even sequential machines.

- Another possible area of interest would be to attempt to find additional parallelism by transforming sequential portions of an Ada program to the idioms we have suggested. These idioms could then be executed efficiently under the proper implementation. As mentioned in Section 7.2.2, the exception semantics of Ada make it more challenging to accomplish this goal. Although there is discussion that these semantics will be relaxed in Ada 9X to allow vectorization, a further relaxation would be required for parallelization to occur.

  In addition, the presence of both multiple threads of execution and shared variables can lead to dependences that are not present when sequential programs are parallelized [SS88]. Parallelization of Ada programs that contain these two features must ensure that these dependences are upheld. Despite these challenges, we feel that this area merits attention.

- Traditionally, the *control flow graph* has provided a useful representation for performing data flow analysis on sequential programs [ASU86]. More recently this graph has been used to construct the *control dependence graph* [FOW87] and to convert a program into *static single assignment* form [CFR+89]. Both of these representations can be used to provide more efficient algorithms for various program optimizations.

  As the control flow graph is central to all of this work, obtaining an equivalent representation for parallel programs (in particular, Ada) would be

have cited. Therefore, it unfortunately appears that the definition and implementation of these requirements will not provide efficient loop-level parallelism.

7.3 Future Work

We now focus on possible extensions of our work.

- This work has provided efficient solutions to several significant shortcomings in the Ada tasking model. We have obtained these solutions by specifying Ada idioms that take the form of a generic package. This technique has two major advantages:

  - Since the specification is written solely in Ada, portability is maintained across all machines.
  - Due to the restricted accessibility of the contents of the package, special optimizations can be performed that normally would not be allowed if full programmer accessibility were present.

  Since this technique is not specific to the problems we have addressed, it may be used to develop other idioms (tasking and non-tasking) that an implementor may feel are desirable. This includes, but is not limited to, any future shortcomings that may arise out of the Ada 9X standard.

- Our work has focused on providing efficient implementations of our idioms on a shared memory multiprocessor. A logical extension of this work would be derive efficient implementations of these idioms for alternative

Although this requirement mentions the desirability of efficient loop-level parallelism in Ada, it appears to rule out the addition of a language construct that would allow the programmer to specify this form of parallelism. Instead it favors a compiler detection approach. While this continues to be an area of research, it seems odd that the requirements team would rely on the presence of an optimizing compiler to provide this desirable feature, instead of allowing the programmer to specify this type of parallelism directly.

In addition, it appears that an accommodation for the parallelization of loops is not included in this requirement. Although many of the compiler techniques used for vectorization can be used in parallelization, the semantics of a vector loop differ from those of a parallel loop. In a vector loop an ordering of iterations is defined; a loop may be vectorized even though certain dependences (anti-) exist among iterations. However, a parallel loop contains no implicit ordering; no dependences may exist among iterations.

Furthermore, the size of vector loops are ultimately limited by the functionality of the underlying machine; each loop is constrained to contain only one vector assignment. On the contrary, the size of a parallel loop may be arbitrarily large. Therefore, if the compiler detection approach is used in Ada 9X it appears that a specific requirement to accommodate Ada 9X to parallelization would be needed.

Both of the requirements we have cited address issues that are important in obtaining loop-level parallelism in Ada. As these are requirements, it is difficult to determine what language revisions will eventually result. However, these requirements fail to mandate the resolution of two of the three deficiencies that we

7.2.2 Vector Architectures

This requirement discusses statement level parallelism in vector architectures. Although the topic appears to be out of the context of our work, some of the issues addressed our relevant.

> "**Study Topic S7.3-A(1) – Statement Level Parallelism:** Ada 9X should accommodate compiler techniques for efficiently mapping sequences of Ada statements, including particularly appropriate loops, onto vector architectures.
>
> **Discussion:** Although Ada permits the use of tasks for explicitly describing parallelism, this is much too heavy a mechanism for loop level parallelism. In other languages, approaches to this problem have involved automatic recognition of loop parallelism by compilers and the introduction of specialized loop constructs explicitly specifying parallel execution.
>
> Compiler techniques for mapping sequential code onto vector architectures are well known. However, there is some difficulty in applying these techniques to Ada because of exception semantics. The relaxations permitted by section 11.6 of the Ada standard are insufficient to permit vectorization to the desired degree. For example, a simple loop that adds the elements of two vectors together cannot be vectorized since Ada requires that an exception occurring in the first [iteration of the] loop prevent further iteration[s] of the loop."

> **Discussion:** A critical requirement for efficiency of large scale parallel applications is that there be no "serial bottlenecks," i.e. points at which the execution time depends on executing serial code whose execution time is dependent on the number of processors.
>
> Ada 83 does provide many of the needed facilities to meet this requirement. Tasks can be initiated in parallel, and terminated in parallel (using the terminate alternative). However, there is no easy way to give tasks an identity so that the tasks that are initiated in parallel can work on separate parts of a problem without communicating with some master controlling task.
>
> There are a number of possible approaches to solving this problem. One approach is to provide a task with the ability to directly determine its own identity (for example, its index in an array of tasks). Another approach is to provide some kind of mechanism for parameterizing tasks."

It is encouraging to see that this requirement addresses the first shortcoming that we (and others) have identified, the inability of a task to determine its own identity. It appears that this deficiency was an oversight in the original design which will be fixed in the new revision. However, the same cannot be said for the two remaining obstacles to efficient loop-level parallelism. Although neither loop-level parallelism or these two deficiences are addressed by the requirements document, the next section does discuss related issues.

the Ada 9X requirements were defined and made public [Ada90]. These requirements were given to the Mapping/Revision Team whose goal is to design specific changes to the Ada-83 standard that satisfy these requirements. As these changes are made, the Implementation/Analysis Team investigates the ramifications of their implementation.[1]

Although many of the Ada 9X requirements are worthy of study, we refer the interested reader to [Ada90] for a full description. Instead, we point out the requirements that are relevant to this work.

Recall from Chapter 2 the three shortcomings in the Ada tasking model when trying to achieve loop-level parallelism:

1. the lack of the ability to distribute identities to tasks in parallel,
2. the synchronization point that is required during task initiation, and
3. the storage overhead required to manage a task's status.

The next two sections specify the two Ada 9X requirements that most closely address these deficiencies.

7.2.1 Managing Large Number of Tasks

Section 7.2 of the requirements document [Ada90] specifies:

> "**Study Topic S7.2-A(1):** Ada 9X must provide for the efficient creation, initialization, execution, and termination of large number of tasks.

[1]The NYUAda project has been chosen as the Implementation/Analysis team for Ada 9X.

(see Chapter 2) by modifying our idiom so that an instantiation would provide subprograms that can later be called to obtain a parallel loop. While this would place the parallel loop idiom in the sequence of statements section of the program, it would not remove the necessity of specifying the loop body and iteration range as parameters.

However, whatever we lose in "ease of programming", we gain in reliability. Due to the fact that our idiom is a package instantiation, we can associate more information with it then can be done with a parallel loop control structure. In particular, we can collect status information about each thread in the loop and provide functions that can be used to obtain this information, something that we have not seen in any other parallel loop construct. Furthermore, both of these features would not be present if the Dritz scheme were used.

7.2 Ada 9X

To maintain portability, we have placed an important restriction on our work: all idioms we supply must be written in standard Ada. During the development of this work, a revision of the 1983 standard has begun. The eventual outcome of this process will be a new standard called Ada 9X, (where X is a digit in the range 0-9). This section focuses on this revision process with particular emphasis on the problems we have discussed.

In October 1988, the Ada 9X project was initiated with an invitation to the public to submit revision requests. This period concluded in October 1989 and produced over 750 requests for language changes. Out of these requests

the overhead associated with initialization, issues of granularity may no longer need to be considered; all semantically parallel loops will be executed in parallel.

A Limitation of the Gen_Beacon Idiom

The gen_beacon idiom provides the programmer with a monitored fetch_and_add shared variable. Since this idiom takes the form of a package instantiation its declaration appears in a declarative part, as is the case with other variable declarations. We provide access to this variable through several subprograms. As Ada does not permit subprogram parameters, we are unable to pass access fetch_and_add variables to subprograms.

To fullfil this need, the designers of the language suggest the using generic subprograms. In Chapter 3 we illustrate this limitation with an example and show how it can be overcome with the use of generics.

Furthermore, recall that our goal in constructing the gen_beacon idiom was to provide fetch_and_add variables for our parallel loop idiom. As the gen_minitask idiom does not need to pass fetch_and_add variables as parameters, the gen_beacon idiom realizes this goal; it provides an efficient mechanism for obtaining fetch_and_add variables *within* Ada.

Readability of the Gen_Minitask Idiom

We have constructed the gen_minitask idiom to play the role of a parallel loop in Ada. In doing so we have substituted a declarative item (a package instantiation) for a control statement (a loop). Despite this fact, we have not sacrificed any expressive power. However, the resulting program can be difficult to read. A possible solution to this problem is to utilize the scheme suggested by Dritz

might lead one to expect our work to be only applicable to machines that implement the fetch_and_add primitive in this manner. While it is true that a combining network removes the serial bottleneck normally associated with task initialization, its absence does *not* preclude an efficient implementation.

In Chapter 6 we saw that the time to initialize one minitask compared favorably to both the time to initialize an iteration of a Parallel Fortran loop and a regular Ada task. This fact is independent of whether multiple initializations can be performed in parallel (as is the case with our work with the presence of a combining network) or serially, (as done in the other two methods). For this reason we conclude that our work can provide efficient loop-level parallelism even when the fetch_and_add primitive is not implemented using a combining network.

When to use the Gen_Minitask Idiom

In Chapter 6 we define the lower bound granularity for a parallel loop construct and compute this value for the gen_minitask idiom. We show that a particular threshold exists where use of our idiom is expected to out perform a sequential execution. This threshold is measured in terms of the average size of each iteration and is a function of the number of iterations. We saw that although our idiom has a lower threshold than other parallel loop idioms, it should not be used when this threshold is not surpassed. As is the case with most parallel loop idioms used today, a potential parallel loop that has a small body, or a small number iterations should be executed serially.

It remains to be seen whether the need to consider this threshold will exist in the future. If developments in hardware and software can be found that reduce

7.1 Our Work

The goal of this work is to provide efficient loop-level parallelism in Ada without modifying the language. In Chapter 2 we discussed three significant shortcomings that must be overcome before this goal can be achieved.

In Chapter 3 we presented an enhanced version of a previous solution to the first of these deficiencies, the bottleneck that results when one wishes to distribute distinct identities to a group of tasks. The solution presented, the gen_beacon generic package, is more efficient in both time and space then its predecessor.

We addressed the latter two deficiencies, the synchronization point required between task creation and task activation and the storage overhead required to manage an Ada task, in the remaining chapters. In Chapter 4 we explored possible solutions to these deficiencies and presented, the gen_minitask generic package. In Chapter 5 we described the implementation of this idiom in detail and explored other implementation issues. In Chapter 6 we showed that the overhead and efficiency of our idiom outperforms the standard means of obtaining a parallel loop in Ada. Moreover, we showed that it compares favorably with the parallel loop construct of IBM Parallel Fortran.

In this section we address other issues concerning our work.

Combining Network Dependence

The analysis of our implementation assumes that multiple fetch_and_add operations to the same memory location are combined in the network. This fact

Chapter 7

Conclusions

In an interview in the 1984, Jean Ichbiah, head of the Ada design team stated:

> "The Ada language was designed with three overriding concerns: a recognition of the importance of program reliability and maintenance, a concern for programming as a human activity, and efficiency." [Ich84]

In this work we have addressed the last of these concerns. We have shown that although Ada's tasking model provides a robust method for specifying interprocess communication, its *semantics* hinder the realization of efficient loop-level parallelism.

This concluding chapter is divided into three sections. First, we summarize our work, pointing out it advantages and disadvantages. Next, we discuss the current effort to revise the 1983 standard of the language. Lastly, we outline some future areas of study.

```
for i in 1..N loop
    A(i) := B(i) * C(i); -- A(i) is stored here.
    D(i) := A(i-1) / 2; --  and used here in next iteration.
end loop;
        ⋮
declare
    -- Add an extra iteration.
    subtype i_range is integer range 0..N;
    procedure i_body (i : i_range) is
    begin
        if i > 0 then -- Must check the end points
            A(i) := B(i) * C(i);
        end if;
        if i < N then
            D(i+1) := A(i) / 2; -- Shift indices of D and A
        end if;
    end loop_body;
    package i_loop is new gen_minitask (i_range, i_body);
begin
    if not i_loop.success then
        -- Handle error condition appropriately.
    end if;
end;
```

Figure 6.10: An Example of Loop Alignment

variable where information is flowing, we can *align* the two statements in this loop so that the store and use are performed *in the same* iteration. This is done in the second part of Figure 6.10 where the resulting loop is executed in parallel using the *GMI*.

Although this transformation appears to be very promising it can only be applied to the special case illustrated by Figure 6.10, when no other flow dependences are adversely affected by the alignment. For more details about this transformation see [ACK87].

```
for i in 1..M loop
    for j in 1..N loop
        A(i, j) := A(i-1, j) * A(i+1, j);
    end loop;
end loop;
        ⋮
declare
    subtype j_range is integer range 1..N;
    procedure j_body(j : j_range) is
    begin
        for i in 1..M loop
            A(i, j) := A(i-1, j) * A(i+1, j);
        end loop;
    end loop_body;
    package j_loop is new gen_minitask(j_range, j_body);
begin
    if not j_loop.success then
        -- Handle error condition appropriately.
    end if;
end;
```

Figure 6.9: An Example of Loop Interchange

among iterations of the j loop, it can be executed in parallel. Although we could execute the inner j loop in parallel using the GMI, it will be more beneficial by having the i loop nested *inside* of the j loop as shown in the second loop part of Figure 6.9. Therefore, we interchange the i and j loop.

6.6.3 Loop Alignment

The last transformation we consider deals with loops that appear to exhibit a flow of values among iterations as shown in Figure 6.10. In the first loop we see that the value stored in A(i) during the i^{th} iteration is used in the next iteration. This appears to render this loop unparallelizable. However, since this is the only

```
for i in 1..N loop
    temp := A(i); -- Reuse of temp impedes parallelization
    A(i) := B(i);
    B(i) := temp;
end loop;
        ⋮
declare
    subtype swap_range is integer range 1..N;
    procedure loop_body (i : swap_range) is
        temp : item; -- Give each iteration its own copy
    begin
        temp := A(i);
        A(i) := B(i);
        B(i) := temp;
    end loop_body;
    package swap is new gen_minitask (swap_range, loop_body);
begin
    if not swap.success then
        -- Handle error condition appropriately.
    end if;
end;
```

Figure 6.8: An Example of Scalar Expansion

compilation system. Since dependence analysis is beyond the scoop of this work we refer the interested reader to [Hin88,BC86] for a more detailed discussion.

6.6.1 Scalar Expansion

The most useful transformation that can be performed to parallelize a loop is *scalar expansion*. Consider the two loops shown in Figure 6.8. In the first loop arrays A and B are being interchanged, element by element. Since the scalar, temp, is reused during each iteration, it appears that this loop is not parallelizable. However, notice that no values stored in temp during one iteration *flow* to any other iteration. Therefore, if each iteration is allocated its own temp variable, this loop can be executed in parallel. In the second part of Figure 6.8 we show the resulting transformation and incorporate it with the GMI.

6.6.2 Loop Interchange

Another useful transformation is *loop interchange*. Recall from Chapter 5 that when two nested loops, one parallel, one sequential, surround a section of code it is better to execute the outer loop in parallel, and the inner loop sequentially on a multiprocessor.[5] The primary reason for this is to reduce the number of fork-join points, while also increasing the size of the outer loop as done in chunking.

Consider the nested loops in the first part of Figure 6.9. Due to the flow of values in the i loop, it must be executed sequentially. However, as no values flow

[5]On a vector machine the opposite is true. Since a parallel loop can correspond to a vector operation, we desire these loops to enclose basic instructions only; a sequential loop enclosed by a parallel loop is not beneficial on a vector machine.

```
for i in 1..N loop
   A(i) := 2 * A(i-1);
end loop;
      ⋮
task body iteration is
   i : integer;
begin
   i := get_iteration; -- A function that gets the iterate
   if i /= 1 then -- First iterate need not wait
      accept signal do -- Wait for previous iterate
         null ;
      end signal;
   end if;
   A(i) := 2 * A(i-1);
   if i /= N then -- Last iterate has no one to signal
      par_loop(i+1).signal; -- Signal next iterate
   end if;
end iteration;
```

Figure 6.7: A Non-Parallel Loop with Synchronization Inserted

perform this synchronization. Therefore, we suggest a sequential execution for loops of this type.

However, some seemingly sequential loops can be executed in parallel with the help of special transformations. In the remainder of this section we discuss three types of transformations. The transformations we discuss have been developed by the fields of vectorizing and parallelizing compilers, compilers that attempt to automatically convert serial loops into parallel ones.[4] Although these transformation are predominately performed on Fortran programs, most apply to Ada as well. As is the case with chunking, these transformations can either be performed manually by the programmer or automatically by the compiler. The latter approach requires incorporating sophisticated dependence analysis into a

[4]Although vectorizing compilers deal only with loops, some parallelizing compilers [BCF+88,CHH89] also focus on other areas of the program to find parallelism.

c is the *chunking factor* (the number of iterations per processor). This enhancement increases the size of the body of the loop, without increasing the total computation, and thereby increases the efficiency of our idiom. Furthermore, the number of minitasks required is reduced by c, freeing processors for other uses.

This enhancement can be performed either by the programmer, sometimes called *loop coalescing*, or by the implementation, often referred to as *chunking*. The matrix multiplication example in Appendix A specifies an example of loop coalescing. For more details on chunking, see the work by Flynn and Flynn [FH90] and Kruskal and Weiss [KW85].

6.6 Other Types of Loops

Up to this point we have dealt solely with parallel loops with no synchronization requirements. In this section we discuss how loops that require synchronization among iterations can be handled by our idiom and other methods.

The first loop in Figure 6.7 gives an example of a loop that cannot be executed in parallel without the presence of synchronization; values computed in iteration i are used in iteration $i + 1$. In order to execute this loop in parallel, explicit synchronization is required; iteration i must *signal* iteration $i + 1$ after it stores the value in $A(i)$. Likewise, iteration $i + 1$ must *wait* for this signal before it may use the value of $A(i - 1)$. Although synchronization for this loop can be specified in Ada, (see Figure 6.7), the resulting program is inefficient. Since most of iteration i must be executed before iteration $i + 1$ is executed, any increase of performance due to parallelism is offset by the additional overhead required to

the same as the number of iterations. This implies a form of chunking is present. Chunking was discussed in Chapter 5 and is addressed in a different context in the next section.

Note that the unit of measure for VS FORTRAN is a machine instruction rather than a C instruction that was used for the gen_minitask idiom. With this in mind, it is clear that the complete constant overhead for VS FORTRAN is lower than the gen_minitask idiom (121 machine instructions vs. 413 C instructions). However, it is also clear that the cost for initializing each iteration is much less with the gen_minitask idiom than it is with VS FORTRAN (38 C instructions vs. 290 $*$ N machine instructions). As the latter comparison is more critical, we conclude that our idiom compares favorably with the parallel loop construct of VS FORTRAN.

Further Enhancements

In addition to providing a useful metric for the performance of a parallel construction, the efficiency formula can also be used as a guide to increase system performance. For example, let's consider the equation for E_{gm}.

$$E_{gm} = \frac{S-2}{451+S}$$

As we can see, E_{gm} is a function of S, the average size of loop body. By increasing this size, we can improve the efficiency of the GMI. Furthermore, we wish to increase S, without increasing the total amount of computation to be performed.

One way of achieving this is to assign *more than one* iteration to a minitask. Using this scheme each minitask will execute an average loop body of $c \cdot S$, where

Efficiency	Average Thread Size		
	Ada *PLE*	Tree-Based *PLE*	Gen_Minitask
5%	$22.4 \cdot N + 43.7$	$34.4 \cdot \lfloor \log N \rfloor + 29$	21.6
10%	$47.3 \cdot N + 94.4$	$72.6 \cdot \lfloor \log N \rfloor + 63.4$	47.9
20%	$106.5 \cdot N + 213$	$163.3 \cdot \lfloor \log N \rfloor + 143.3$	110.3
30%	$182.6 \cdot N + 370$	$279.9 \cdot \lfloor \log N \rfloor + 250.4$	190.4
40%	$284 \cdot N + 576.7$	$435.3 \cdot \lfloor \log N \rfloor + 390.7$	297.3
50%	$426 \cdot N + 866$	$653 \cdot \lfloor \log N \rfloor + 587$	447.0

Table 6.7: Average Thread Size Required for Various Efficiency Levels

A Comparison with IBM VS Fortran

Another way of evaluating the efficiency of our idiom is to compare its overhead with the overhead of a parallel loop from a different language. We have see that our idiom requires 451 C instructions before *all* minitasks begin executing. As a comparison we give the corresponding figures for VS FORTRAN [IBM89] as estimated by Cytron [Cyt91].

The Parallel Loop construct in VS FORTRAN executed on a 3090 requires:

- 58 machine instructions to initiate a parallel loop,
- 290 machine instructions to initialize each iteration thread,
- 5 machine instructions (performed in parallel) for each thread commence,
- 58 machine instructions to terminate the loop construct.

Thus, $121 + 290 * N$ instructions are required to execute a parallel loop in VS FORTRAN on a 3090, where N is the number of processors assigned to execute the loop. Note that the number of processors assigned to a loop isn't necessarily

for the tree-base PLE out performs the serial version.

Efficiency of the GMI

In this section we apply (13) to the gen_minitask construct and get:

$$E_{gm} = \frac{S+2}{451+S} \tag{17}$$

Comparing E_{gm} with E_{ple} and E_{tree}, we see two major differences:

1. E_{gm} does not depend on N.

2. E_{gm} has a lower constant factor associated with its overhead.

These two differences substantially reduce the size of each thread that is needed to obtain a particular level of efficiency. Since E_{gm} does not depend on N, we use (14) to solve for S.

$$S = \frac{E_{gm} \cdot 451 - 2}{1 - E_{gm}} \tag{18}$$

We use (18) with $E_{gm} = 25\%$ to compute the required value of S. We contrast this value which is constant for all values of N with the values of the other two idioms (Figure 6.6).

In Table 6.7 we use (18) to specify the size of each thread that is needed to achieve various levels of efficiency. From this table we see that obtaining an efficiency of 50% with the gen_minitask requires 451 instructions. Although this value may seem rather large, the GMI is able to achieve reasonable levels of efficiency with small thread sizes, something both versions of the Ada PLE are unable to do.

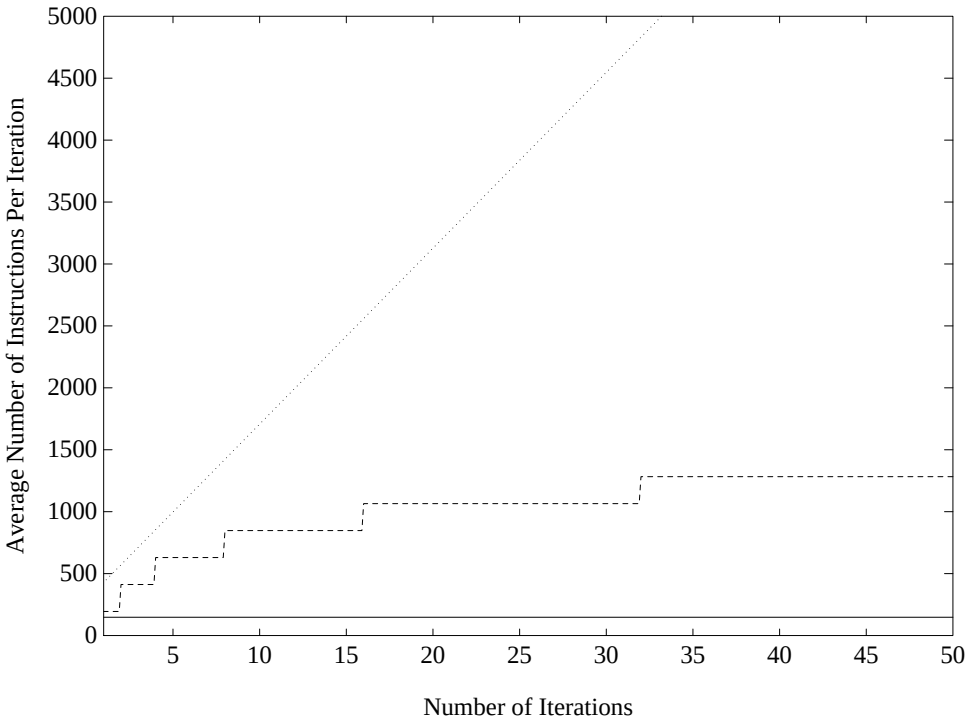

Figure 6.6: 25% Efficiency for $E_{ple}(\cdots)$, $E_{tree}(- - -)$, and $E_{gm}(—)$

Efficiency of the Ada PLE With Tree-Based Initialization

Applying (13) to the tree-base version of the Ada PLE, we get:

$$E_{tree} = \frac{S+2}{653 \cdot \lfloor \log N \rfloor + 591 + S} \tag{16}$$

As is the case with the serial version of the Ada PLE, the efficiency of the tree-based version is a function of the number of threads and the size of each thread. Note that a linear term has been replace by a logarithmic one.

Once again we use (14) to compute the values of S that is needed to obtain an efficiency rate of 25% (Figure 6.6). We compare these values with those of the serial version of the Ada PLE. As expected the average thread size required

$$\begin{aligned} &= \frac{\frac{N\cdot(S+2)}{O_c+S}}{N} \\ &= \frac{S+2}{O_c+S} \end{aligned} \tag{13}$$

As O_c is usually greater than 2, efficiency ranges from 0 to 1. Solving (13) for S, we get:

$$\begin{aligned} \frac{S+2}{O_c+S} &= E_c \\ S &= \frac{E_c \cdot O_c - 2}{1-E_c} \end{aligned} \tag{14}$$

Efficiency of the Ada PLE

Equation 13 specifies that the efficiency of a parallel loop construct is a function of its overhead and the size of each iteration. Applying this equation to the Ada PLE, we get:

$$E_{ple} = \frac{S+2}{426 \cdot N + 870 + S} \tag{15}$$

The efficiency of the Ada PLE is a function of the number of threads and the size of each thread. Since the denominator of (15) contains a rather large constant factor (870) and also a large factor of N (426), it takes a significant thread size to get the efficiency close to one.

To get a feel for the significance of this denominator, we use (14) to compute the values of S for particular values of N that are needed to achieve an efficiency of 25% (see Figure 6.6). In order to obtain this level of efficiency, the size of each thread must increases rapidly as N gets larger. Once again this is a result of the serial bottleneck that is required in distributing iterates to each task.

reducing the running time of their program, but reducing it in proportion to the amount of parallelism that is present. In this section we examine each parallel loop construct in this light; we determine how efficient they are in achieving their optimal performance.

Before we discuss efficiency we define the *speedup of parallel loop construct* c, SP_c, in the usual manner. That is, it is the ratio of the number of executed instructions of the sequential version to the number of executed instructions of the parallel version, i.e.

$$SP_c = \frac{SEQ}{PAR_c}. \tag{12}$$

Speedup is useful in measuring how much faster a parallel construct is compared to the sequential version. A speedup greater than one signifies that the parallel version is faster than the sequential version, a desirable result. However, this metric does not convey how much parallelism is required to achieve this value. An application with a speedup of five using five tasks should be distinguished from an application with a speedup of five using ten tasks. As the second application needs five more tasks to achieve the same speedup, it is not as efficient.[3]

To capture how efficient an application is in utilizing its resources, the metric, *efficiency* is defined. E_c is the ratio of the speedup for parallel loop construct c over the number of tasks that are used, i.e.

$$\begin{aligned} E_c &= \frac{SP_c}{N} \\ &= \frac{\frac{SEQ}{PAR_c}}{N} \end{aligned}$$

[3]In this discussion we assume all tasks are being used productively.

illustrates for a given number of iterations the number of instructions that are required for parallel execution to be beneficial. All coordinates that lie below the curve correspond to values of the iteration size and number of iterations where the parallel construct is *not* beneficial. Likewise, all coordinates above the curve show values where the parallel construct out performs sequential execution.

This figure illustrates that the gen_minitask idiom is the most general in that it requires both a smaller number of iterations and iteration size than the other two idioms. For example, when $N = 100$, the Ada PLE and its tree-based version require an average iteration size of greater than 437 and 43 instructions, respectively, to justify their use, while the GMI only requires an average iteration size of 2.5 instructions. While it is certainly possible for a parallel loop to contain an average iteration size of 43 instructions, this threshold is rather high considering this is the first point at which an increased performance is obtained.

While this metric shows that the GMI is much more efficient than the two versions of the Ada PLE, it more importantly illustrates that the granularity of our idiom is small; for all but the smallest loops, our idiom is superior to a sequential execution.

6.5.2 Efficiency

The previous section describes the lower bound granularity required to offset the overhead associated with a parallel loop construct. This metric is useful in determining when it is beneficial to use a parallel loop construct. However, most applications that employ parallelism do so with the expectation of not only

Computing LBG_{gm}

Lastly, we compute LBG_{gm}, the lower bound granularity for the GMI. Since $O_{gm} = 451$, we have

$$LBG_{gm} = \begin{cases} \frac{451-2 \cdot N}{N-1} & \text{if } N \leq 225 \\ 0 & \text{if } N \geq 226 \end{cases} \quad (11)$$

The value of LBG_{gm} is also equal to zero as N gets larger. This is as expected; as the amount of parallelism increases, the required grain size that is needed to offset the overhead decreases. Although the limits of LBG_{gm} and LBG_{tree} are equal, the GMI out performs the tree-based version of the Ada PLE.

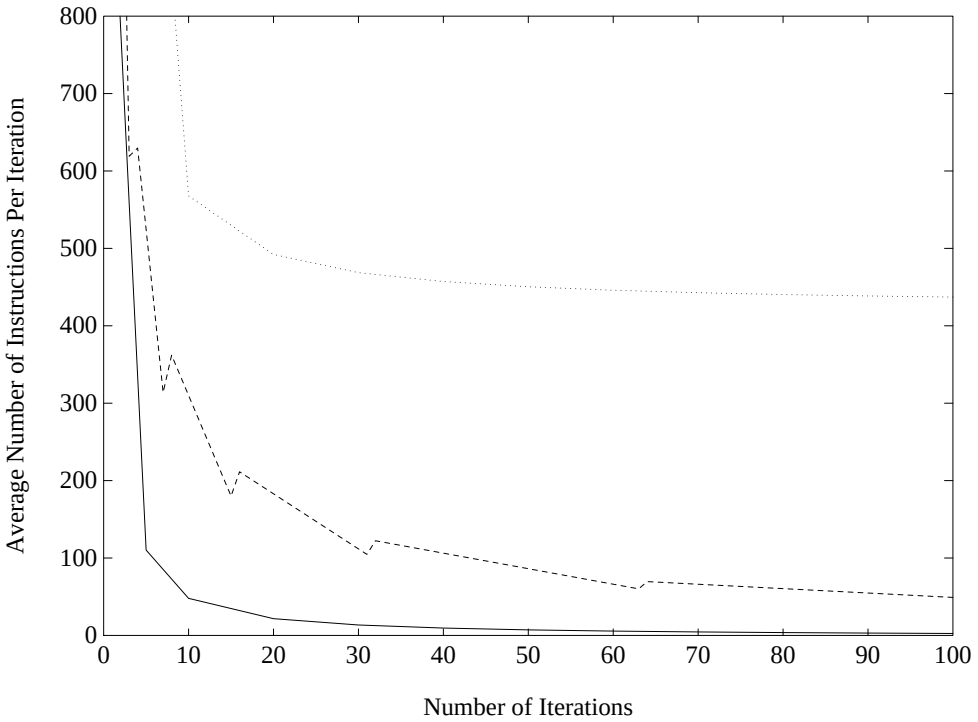

Figure 6.5: $LBG_{ple}(\cdots)$, $LBG_{tree}(--)$, and $LBG_{gm}(—)$

In Figure 6.5, we plot the values of LBG_{ple}, LBG_{tree} and LBG_{gm}. Each curve

Regardless of the level of parallelism of our machine, the average size of each iteration must be *greater than* 424 instructions in order for the execution time to be reduced. As this number is rather large, we can see why the Ada task is considered to be a coarse grain construct not amenable to loop-level parallelism. This is a direct consequence of the serial bottleneck that is present in distributing iterates to each task.

Computing LBG_{tree}

We next compute LBG_{tree}, the lower bound granularity for the Ada PLE using the tree-based initialization. As in the case of the serial Ada PLE we use (8) to compute this value. Since $O_{tree} = 653 \cdot \lfloor \log N \rfloor + 591$, for values of N where $653 \cdot \lfloor \log N \rfloor + 591 > 2 \cdot N$ we have

$$LBG_{tree} = \begin{cases} \frac{653 \cdot \lfloor \log N \rfloor + 591 - 2 \cdot N}{N-1} & \text{if } 653 \cdot \lfloor \log N \rfloor + 591 > 2 \cdot N \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

For sufficiently large N, $653 \cdot \lfloor \log N \rfloor + 591 < 2 \cdot N$, thereby making $LBG_{tree} = 0$. This fact confirms our intuition that as N becomes large, the tree-based version of initializing the Ada PLE is more efficient than its serial counterpart; the required granularity per processor is zero. However, we shall see that it still does not provide a means of realizing efficient loop-level parallelism.

the running time of the program. To compute LBG_c we need to find the value of S such that

$$SEQ = PAR_c.$$

Substituting (5) and (6), and solving for S we get the following:

$$LBG_c = \frac{O_c - 2 \cdot N}{N - 1} \tag{7}$$

As the unit of measure for LBG_c is the average number of instructions in an iteration, it is bounded below by 0. Thus, the formal definition of LBG_c is:

$$LBG_c = \begin{cases} \frac{O_c - 2 \cdot N}{N-1} & \text{if } O_c > 2 \cdot N \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Equation 8 provides a lower bound grain size for any parallel loop construct. In particular, it can be applied to our three parallel loop constructs. In this way we can determine the minimum number of instructions each must execute before *any* savings due to parallelism is realized.

Computing LBG_{ple}

First we consider the Ada PLE construct with serial initialization. Recall from Table 6.6 that $O_{ple} = 426 \cdot N + 870$. Therefore,

$$LBG_{ple} = \frac{424 \cdot N + 870}{N - 1} \tag{9}$$

For large values of N we find:

$$\lim_{N \to \infty} \frac{424 \cdot N + 870}{N - 1} = 424.$$

Definition 1 Let N be the number of iterations of a loop, S be the average number of instructions executed by each iteration, and O_{seq} be the iteration overhead associated with a sequential loop. The number of instructions required to execute the loop sequentially is given by

$$SEQ = N \cdot (S + O_{seq}). \tag{5}$$

Since two C instructions are needed to increment and test the index variable of loop, $O_{seq} = 2$.

Definition 2 Let O_c represent the *overhead* of a parallel loop construct c and S be the average number of instructions executed by each iteration. The number of instructions required to execute the loop in parallel using c is

$$PAR_c = O_c + S. \tag{6}$$

In order to determine the granularity of a particular parallel construct we first determine the point at which the parallel construct *reduces* the execution time as compared with the sequential execution, i.e. at what point is it beneficial to utilize this construct. We define this point formally.

Definition 3 Let c be a parallel loop construct. We define LBG_c, *Lower Bound Granularity of* c, to be the minimum average number of instructions executed by each iteration such that the number of instructions executed using c (a parallel construct) is equal to the number of instructions executed by the sequential version of the loop.

By computing the LBG_c for a loop construct c, we obtain a lower bound for this construct; any use of c with greater than this number of instructions *reduces*

ACTION	ADA PLE	TREE-BASED ADA PLE	GMI
Initiation	975	975	413
Initialization	$426 \cdot N - 105$	$653 \cdot \lfloor \log N \rfloor - 384$	38
Total	$426 \cdot N + 870$	$653 \cdot \lfloor \log N \rfloor + 591$	451

Table 6.6: Overhead Requirement for Each Parallel Loop Idiom

It shows that task initiation is reduced by over 50% using the GMI. Furthermore, a more significant savings is found in the initialization process. The GMI requires a constant number of instructions *regardless of the number of iterations to be executed*, where as both versions of the Ada PLE require a number of instructions dependent on the number of iterations of the parallel loop. Although the tree-based version of the Ada PLE replaces a linear term with a logarithmic one, it is nevertheless still dependent on N. This fact combined with the high initiation requirements required for both versions of the Ada PLE renders them inappropriate in obtaining efficient loop-level parallelism.

6.5.1 Granularity

Once the overhead for a parallel loop construct has been determined, its granularity can be computed. The previous section identified the amount of overhead required for the GMI and the two versions of the Ada PLE. In this section we compute the granularity for all three versions of parallel loops.

Before we discuss granularity, we define two terms, SEQ and PAR_c. The former corresponds to the number of instructions required to execute a loop sequentially, while the latter represents the number of instructions needed to execute the loop in parallel using parallel loop construct c.

Action Performed	Number of Instructions
faa Function Call	0
loop_body Function Call	38

Table 6.5: Initialization Overhead for each Minitask

increased functionality not normally associated with parallel loops. Namely, the ability to create local objects, be it variables, subprograms, or tasks. For this reason, we cannot "inline" the loop_body procedure. In Table 6.5, we specify the number of instructions required to implement the faa function call and the call to the loop_body procedure.

We now consider the combined overhead required before a minitask begins its execution. From Table 6.4, we see that 413 instructions are required for minitask initiation. Once a minitask has been initiated, it executes an additional 38 instructions during its initialization. Therefore a total of 451 instructions are executed before each minitask begins its execution.

6.5 Comparing Parallel Loop Idioms

In this section we compare the overhead requirements of the gen_minitask idiom with those of both versions of the Ada *PLE*. As expected our idiom compares favorably. We discuss issues of granularity and efficiency for each of the parallel loops idioms.

Table 6.6 specifies the overhead requirements for each form of parallel loop.

```
task body minitask is
    iterate : iteration_range;
    procedure record_exception (exception_raised : exception_type) is
        ⋮
    end record_exception;
begin
    iterate := iteration_range (iteration_counter.faa (1));
    loop_body (iterate);
    exception
        ⋮
end minitask;
```

Figure 6.4: The Body of a Minitask

phase of both versions of the Ada PLE.

Consider the body of the minitask as shown in Figure 6.4. Once a minitask is activated, it performs two operations before the sequence of statements of the parallel loop are executed; it calls the faa function to obtain its iterate, and then it calls the procedure, loop_body, to execute the corresponding iteration.

The faa function call is implemented in a highly optimized manner; multiple faa requests are performed in parallel. Moreover, as described in Chapter 4, we acquire the value of the iterate for free; we do not need to execute any additional instructions to obtain this value. As we have seen in Sections 6.2.2 and 6.3.1, this is a substantial savings over the corresponding method used to distribute iterates to each task of the Ada PLE.

We implement the procedure call to loop_body in the usual manner. At first glance, one may wish to remove the procedure call by inserting the code of the procedure into the minitask body itself. However, recall that by allowing the user of our idiom to specify the body of the loop using a procedure, we provide

Action Performed	Number of Instructions	Percentage of Total
Pre-Minitask Creation	149	36%
Minitask Creation/Activation	198	48%
Post Execution	66	16%
Total	413	100%

Table 6.4: Initiation Overhead for a Family of Minitasks

Recall that in order to initiate tasks in the Ada PLE a synchronization point exists between task creation and task activation. To implement this synchronization point, a "join" and subsequent "fork" must take place. If a processor executing a thread is delayed before the join occurs, the fork, and subsequently all threads will be delayed. Although our analysis accounts for the instructions that are executed to perform the join and fork, it does not account for the possible delays in performing this join.

In the GMI task creation and activation are combined; no synchronization point exists. Therefore, delays of this type only affect the task executing on the processor that exhibits the delay, not all tasks.

6.4.2 Minitask Initialization

In this section we analyze the number of instructions each minitask executes to receive its unique identity or iterate. This initialization must occur before a minitask can begin to execute the sequence of statements of the procedure that corresponds to the body of the parallel loop. The overhead required for our idiom is substantially less than the overhead associated with the initialization

- Passing these objects as parameters to the generic instantiation.

Minitask **Creation/Activation:** This category represents all actions involved with the creation and activation of minitasks. Since minitask creation and activation are combined, this category includes actions that are the semantic equivalent to task creation, post task creation, and task activation in the Ada PLE. It includes the following actions:

- The parent task places the appropriate number of minitask items on to the parallel queue.
- Each of these items is removed by a processor and a minitask is created.
- The declarative part of each minitask is elaborated.

Post Execution: This category involves:

- reclaiming storage for each of the minitasks,
- exiting the package's stack frames,
- unblocking the parent task, and
- the parent task exiting the stack frame of the block statement.

Table 6.4 specifies the number of the instructions that are performed to complete each of these categories. Although these initiation figures offer a better than 50% improvement over the corresponding figures for the Ada PLE, the actual performance may be even better:

- Our implementation combines task creation with task activation and performs these combined operations in parallel with other creation/activation operations.

- Our implementation reduces the number of instructions needed to create each iteration task by reducing the status required for each of these tasks.

- Our implementation distributes unique iterates to each task in parallel.

Although all four of these factors contribute to the reduction in overhead, the last factor plays the most significant role.

In the next two sections we elaborate on these factors by providing the number of instructions that are required to perform minitask initiation and initialization. This is followed by Section 6.5 where we compare these numbers against the corresponding values for both versions of the Ada PLE.

6.4.1 Minitask Initiation

In this section we describe the amount of overhead that is required to initiate a family of minitasks. The figures we present include the elaboration of the subtype and procedure body, in addition to performing instantiation of the generic package. We divide this initiation into the following three categories:

Pre-Minitask **Creation:** Actions that are performed before the creation of any minitasks. These correspond to:

- Elaborating the subtype,

- Elaborating the loop body procedure, and

326 and 327 instructions, respectively, we have the following upper bound for the number of instructions needed to initialize N tasks in an incomplete tree:

$$\begin{aligned} INTL_{tree}(N) &\leq 206 + 326 \cdot \lfloor \log N \rfloor + 327 \cdot (\lfloor \log N \rfloor - 1) + 57 \quad (3) \\ &= 653 \cdot \lfloor \log N \rfloor - 64 \quad (4) \end{aligned}$$

The first term in (3) corresponds to the parent task initializing task number 1. The second and third terms represent the most left and right child initializations that can occur, respectively. The last term corresponds to the number of instructions that a leaf node executes to determine that it has no children to initialize.

Combining (2) and (4), we have the following inequality:

$$653 \cdot \lfloor \log N \rfloor - 384 \leq INTL_{tree}(N) \leq 653 \cdot \lfloor \log N \rfloor - 64$$

As expected, the initialization overhead for the tree-base PLE is logarithmic in N. As the lower and upper bounds for $INTL_{tree}$ are relatively close, we choose to use the lower bound in our comparisons of Section 6.5.

6.4 The Gen_Minitask Idiom (GMI)

In this section we describe the overhead associated with both versions of the GMI. As expected, this overhead is substantially less than the overhead associated with the Ada PLE as described in the previous sections. The reduction in overhead is due to a number of factors:

- Our implementation does not need to elaborate the types and objects required for the Ada PLE.

The true value of $INTL_{tree}(N)$ is a function of how much of the leaf level is complete. Instead of determining this value we derive an upper bound for it, and show that this bound is quite close to the lower bound for $INTL_{tree}(N)$.

As all tasks numbered less than CT_N are initialized when we initialize CT_N, any task that would make $INTL_{tree}(N)$ greater than $INTL_{tree}(CT_N)$ must be numbered greater than CT_N, i.e. it must be on the incomplete leaf level. Consider one of these leaf tasks, T. As we have seen, the time at which T becomes initialized is a function of when its ancestors become initialized. Left children become initialized after their parents are initialized. However, as right children must wait for both their parent to become initialized and for the parent to initialize its left child, right children ancestors of T must wait the longest. Since task CT_N has to wait for $\lfloor \log N \rfloor$ left and right children initializations, any leaf task that would make the value of $INTL_{tree}(N)$ greater than $INTL_{tree}(CT_N)$ must have to wait for more than $\lfloor \log CT_N \rfloor$ left and right children initializations.

As task T is at level $\lfloor \log T \rfloor$, it must wait for $\lfloor \log T \rfloor$ left initializations or $326 \cdot \lfloor \log T \rfloor$ instructions. Since $\lfloor \log T \rfloor = \lfloor \log CT_N \rfloor + 1$, at most 326 additional instructions can be attributed to left initializations. As right initializations require one more instruction than left initializations, in order for $INTL_{tree}(T)$ to be greater than $INTL_{tree}(CT_N)$, at least $\lfloor \log CT_N \rfloor$ right children initializations must occur via the ancestors of T.

Only one task at level $\lfloor \log T \rfloor$ requires $\lfloor \log T \rfloor$ right initializations: the rightmost task at level $\lfloor \log T \rfloor$. As the initialization of N tasks does not form a complete tree, $\lfloor \log T \rfloor$ right initializations cannot take place. Therefore, the most delay task T can exhibit is by waiting for $\lfloor \log N \rfloor$ $(= \lfloor \log T \rfloor)$ left initializations and $\lfloor \log N \rfloor - 1$ right initializations. Since a left and right initialization require

initialize the root, or first task of the tree. Next, each of the ancestors of the N must initialize their children. Since N forms a tree, there are $\lfloor \log N \rfloor$ ancestors of N. Each ancestor must initialize two children which comprises 653 instructions.[2] Therefore, an additional $653 \cdot \lfloor \log N \rfloor$ instructions must be executed before task N is initialized.

Once task N has been initialized, it attempts to initialize potential children tasks. Although it has no children tasks to initialize, it nevertheless has to execute 57 instructions to determine this fact. Thus, the total number of instructions required to initialize N tasks, where N forms a complete tree is:

$$\begin{aligned} INTL_{tree}(N) &= 206 + 653 \cdot \lfloor \log N \rfloor + 57 \\ &= 653 \cdot \lfloor \log N \rfloor + 263 \end{aligned} \quad (1)$$

Incomplete Trees

We now compute $INTL_{tree}(N)$, where the initialization of these N tasks *does not* form a complete tree. Therefore, we assume

$$N \neq 2^k - 1, \text{for all integers } k \geq 1,$$

and let CT_N be the task number that corresponds to the rightmost node on the last complete level of the initialization tree for N tasks, i.e. $CT_N = 2^{\lfloor \log N \rfloor} - 1$.

Since the initialization of N tasks must include the initialization of the task numbered CT_N, a lower bound for $INTL_{tree}(N)$ exists:

$$INTL_{tree}(CT_N) \leq INTL_{tree}(N) \quad (2)$$

[2] We do not include the last 6 instructions associated with the initialization of a child task because the children do not need to wait for these instructions to be executed before they are initialized.

Child	Pre-Entry Call	Entry Call	Rendezvous	Total
Left	170	59	97	326
Right	171	59	97	327
Post-Children	–	–	–	6
Total Per Iterate	–	–	–	659

Table 6.3: Per Task Initialization Overhead for Tree-Based Version

task $2i$ becomes initialized 326 instructions after $INTL_{tree}(i)$, i.e.

$$INTL_{tree}(2i) = INTL_{tree}(i) + 326.$$

Likewise, task $2i + 1$ is initialized 327 instructions after $INTL_{tree}(2i)$ or 653 instructions after $INTL_{tree}(i)$. Note that while task i is performing the initialization of task $2i + 1$, task $2i$ can begin the initialization of its children tasks.

Complete Trees

To analyze the total overhead in performing these initializations we first consider the case when the initialization of N forms a complete tree. Thus, let

$$N = 2^k - 1, \text{for some integer } k \geq 1.$$

Consider any level of the tree. As the right child of a task always get initialized after the left child, it follows that the last task to be initialized in any complete level of the tree is the rightmost task. Since a task cannot be initialized until its parent is initialized, the last task to be initialized in a complete tree of size N is the rightmost leaf task of the tree, i.e. task N.

To determine when task N is initialized, we consider what must precede it. We have already seen that 206 instructions are required for the parent task to

6.3.1 Initialization Overhead

We now focus our attention on the overhead required to initialize N tasks using this method. After all tasks have been initiated, the parent task calls the entry of the first task. Since a for loop is no longer needed, the category label pre-entry call in Table 6.2 is reduced. However, the instructions required to perform the entry call and rendezvous remain the same. A total of 206 instructions are required for the first task to receive its iterate, and to reach the point immediately following the accept body. At this point the parent task no longer plays a role in distributing iterates.

Since tasks in the array are activated in parallel, we assume they will reach their accept statement before the parent task reaches its entry call. The instructions associated with these pre-initialization actions are included in the instructions required for the parent task to distribute an iterate to the first task. After a child task receives its iterate, it attempts to call the entries of its two children. In Table 6.3, we categorize the instructions executed to carry out the associated rendezvous into three phases: pre-entry call, entry call, and rendezvous. As we can see, 326 and 327 instructions are required to initialize the left and right child, respectively. As right child initialization occurs after left child initialization, a total of 653 instructions are required to initialize a right child.

We now determine the *total* amount of instructions required to initialize N tasks using the tree-based approach. Consider a child task, i, with two children. We define $INTL_{tree}(i)$ to be the number of instructions required for task i to become initialized using tree-based initialization. From Table 6.3 we see that

```
declare
   task type iteration_task is
      entry get_iterate(iterate : in integer);
   end iteration_task;

   par_loop : array (1..N) of iteration_task;

   task body iteration_task is
      my_iterate, left_child : integer;
   begin
      accept get_iterate(iterate : in integer) do
         my_iterate := iterate;
      end get_iterate;

      -- Distribute iterates to two potential children.

      left_child := 2 * my_iterate;
      if left_child ≤ N then
         par_loop(left_child).get_iterate(left_child);
         if left_child + 1 ≤ N then
            par_loop(left_child + 1).get_iterate(left_child + 1);
         end if;
      end if;
      S_1
      ⋮
      S_n
   end iteration_task;

begin -- Main only initializes the first task.
   par_loop(1).get_iterate(1);
end;
```

Figure 6.3: Tree-Based Initialization Variant of the Ada PLE

executing its iteration 426 instructions after the i^{th} task has begun its iteration, i.e.

$$INTL_{ple}(i+1) = INTL_{ple}(i) + 426.$$

Therefore, the total overhead of the Ada PLE is linearly dependent on N and is given by

$$O_{ple} = 426 \cdot N + 870.$$

We shall see in Section 6.5 that this fact renders the Ada PLE inappropriate for expressing loop-level parallelism.

6.3 Tree-Based Initialization

As the last section illustrates, distributing identities sequentially significantly increases overhead. An alternative method of task initialization is to have each child task distribute identities to other tasks in a tree-like manner. We refer to this method as *tree-based initialization.*

As shown in Figure 6.3, this scheme replaces the N entry calls formerly in the parent task with one entry call to the first task of the par_loop array. However, each task, T, of the array is now responsible for distributing iterates to two children tasks.[1] We refer to these children as the left and right child of T. To determine these tasks a "heapsort"-like method is used; a task with iterate i has descendants with iterates $2i$ and $2i+1$. As this method has greater built-in parallelism, we can expect a logarithmic overhead, instead of a linear one.

[1] From here on "children" is meant in the sense of the initialization tree.

the 198 instructions that are attributed to the child task in the accept statement category of Table 6.2.

Once the parent and first child task have executed their pre-rendezvous instructions, a rendezvous occurs. While the body of the accept statement is performed by the child's processor, the parent task is blocked. The body of the accept statement requires 97 instructions to execute. Upon its completion the child task begins to execute the iteration of the parallel loop that it has been assigned. At this point this iteration has begun.

The parent task, however, has only begun with its job. After the rendezvous has completed, the processor executing the parent task executes instructions labeled post entry call to unblock the parent task. Once the parent task has been unblocked, it attempts to execute the next iteration of the for loop, which includes some loop overhead. Once this has been performed, the next entry call is made, and this process continues.

From Table 6.2 we can conclude that 321 instructions are required to initialized the *first* task using the Ada PLE. Combined with the 975 instructions that are executed by the parent task to initiate this task, we see that 1,296 instructions are executed before the first iteration commences.

We define $INTL_{ple}(N)$ to be the number of instructions required to initialize N iterations using the Ada PLE. Note that $INTL_{ple}$ does *not* include those instructions required to *initiate* these N tasks.

Table 6.2 allows us to compute when the second and subsequent children tasks begin executing their iterations. As task initialization is performed sequentially, the i^{th} child task must wait for $i-1$ children tasks to acquire their identity before it can receive its own. In particular, the $i+1^{th}$ child task begins

Action(s) Performed	Parent Task	Child Task
Pre-Entry Call/Pre-Accept Stmt.	165	16
Entry Call/Accept Stmt.	59	198
Rendezvous	97	
Post Entry Call	130	†
Loop Overhead	140	
Entry Call/Accept Stmt.	59	198
Rendezvous	97	
Post Entry Call	130	†
⋮	⋮	⋮

†At this point the child task begins its execution.

Table 6.2: Task Initialization Overhead for the Ada PLE

the execution rate of each task must be made. As mentioned in Section 6.1.1, our analysis assumes that enough processors are available to execute each task, and that each task executes at the same rate. Therefore, while the parent task executes the instructions labeled pre-entry call and entry call, each of the N children tasks execute the instructions labeled pre-accept statement and accept statement. Since the number of instructions required to execute the pre-entry call category is greater than the number of instructions required to execute the pre-accept statement category (165 vs. 16), we assume that each child task reaches its accept statement before the parent calls any entry.

While a child task waits for the parent task to perform its entry call, it is blocked. In the Ada/Ed system a rendezvous is executed by the processor of the task that executes the accept statement. Thus, when the parent task executes this entry call, it awakens the child task. This blocking of the child task, and its subsequent awakening by the parent account for the majority of

Action Performed	Number of Instructions	Percentage of Total
Pre-Task Creation	465	47.7%
Task Creation	216	22.1%
Post Task Creation	33	3.4%
Task Activation	227	23.3%
Post Execution	34	3.5%
Total	975	100.0%

Table 6.1: Task Initiation Overhead for the Ada *PLE*

amount of the overhead associated with the Ada *PLE*.

In order to distribute a unique identity, or iterate, to each task, a task must call each child task's entry, specifying the iterate as a parameter. In the case of the Ada *PLE*, these calls are performed by the parent task. As it is not possible for these entry calls to be performed parallel, iterates are distributed sequentially (see Figure 6.1 on page 130). Since each child task cannot begin its execution until it receives this iterate, the first action it performs is to execute an accept statement on that entry.

At the conclusion of the rendezvous the child task begins executing the iteration of the parallel loop it has been assigned. At the same time, the parent task continues executing its for loop and calls the entry of the next task. This process continues until all iterates have been distributed. As this is a serial operation for the parent, the amount of delay a child task must endure is dependent on when its get_iterate entry is called by the parent.

Table 6.2 specifies the overhead involved in distributing iterates to each child task. Since this involves the coordination of two tasks, some assumptions about

Post Task Creation: After all tasks are created the rest of the declarative part is elaborated. This includes elaborating the task body of iteration_task.

Task Activation: At the conclusion of the elaboration of the declarative part all tasks are activated. This activation occurs in two steps:

- The parent task notifies all tasks that they may begin their execution; it places them on the ready queue.
- Each new task elaborates its declarative part.

Post Execution: Once all tasks have terminated, their stacks are reclaimed and the stack frame associated with the block statement in the parent task is popped.

We specify the number of instructions needed to perform each of these actions in Table 6.1. As this table illustrates, pre-task creation accounts for almost 50% of the instructions that comprise task initiation. This is due to the elaboration of the entities (task type and array) that are needed to specify a parallel loop in Ada. As expected, most of the remaining overhead is due to task creation (22.1%) and task activation (23.3%). Although these actions are also present in the GMI, we shall see that most actions performed in the pre-task creation category are not required, therefore allowing for the elimination of a significant portion of task initiation overhead.

6.2.2 Task Initialization

In this section we examine the overhead that results when task initialization is performed sequentially. We show that this process accounts for a substantial

Task Initiation: This category includes the instructions required to initiate the tasks of the array.

Task Initialization: Once the tasks have been initiated, their iterates must be distributed. This category includes the instructions required to distribute unique identities to each task, i.e. to solve the *task initialization* problem.

In the next two sections we discuss the overhead associated with each of these categories in executing the Ada *PLE*.

6.2.1 Task Initiation

Although task creation and task activation comprise a major portion of task initiation, other actions must be performed. These actions include:

Pre-Task Creation: This category includes actions that are completed before the tasks are created. These actions create the Ada objects that are needed to construct a parallel loop. They include:

- the elaboration of the task type, iteration_task, and
- the elaboration of the array, par_loop, up to, but not including, the creation of the tasks.

Task Creation: Once the task type and array have been created, the creation of each task takes place. This involves allocating and initializing the task's stack and tcb. As the reference manual specifies that the order in which these creations occur is "not defined by the language" (ARM 9.2(2), 3.2.1), the Ada/Ed system performs them in parallel.

6.1.1 Assumptions

In order to provide a reasonable measurement of overhead we make some simplifying assumptions:

1. All tasks execute at the same rate. If N tasks execute the same ten instructions starting at the same time, we assume all complete in the time it takes one task to execute ten instructions. To compare with a sequential execution we say that *ten instructions are executed.*

2. There are sufficient processors available to execute all parallel actions. If N tasks are eligible to execute, we assume at least N processors exist to execute them. It is a simple matter to scale our results when the number of tasks is greater than the number of processors.

3. No error conditions arise. Although it is certainly possible for a run time error (e.g. storage_error) to occur, we are most interested in the cases where no errors occur. Thus, when code is encountered that is conditionally dependent on whether an error condition has occurred, we assume that the non-error branch is taken.

6.2 The Ada Parallel Loop Equivalent (PLE)

In this section we describe the amount of overhead that is required to execute the Ada PLE in the Ada/Ed system. We divide the instructions that are executed by the Ada PLE into two categories:

methods of obtaining parallel loops in Ada and suggest further enhancements.

6.1 Preliminaries

As noted in Chapter 2, our implementation is based on the Ada/Ed system, a descendant of the first validated Ada compiler [DFS+80]. In addition to providing the basis for the implementation of the gen_minitask idiom, we use the Ada/Ed implementation to determine the overhead associated with executing the Ada *PLE*. Although this overhead figure is likely to differ from the corresponding figures of other implementations, we feel that as our implementation utilizes the same compiler model as the Ada/Ed system, it is a fair comparison. Moreover, sources of overhead in the actions performed by the Ada/Ed system are mandated by the semantics of Ada, and therefore, must be performed by any implementation.

Since we do not have access to an Ada implementation on a parallel machine, our overhead analysis is static. As both the Ada/Ed system and our implementation are interpreters written in C, the unit of measure we use in our analysis is the *number of C instructions* executed by each implementation. Although this approach may not be as accurate as counting machine instructions, we feel that the results presented in this chapter are a good approximation of the magnitude of the overhead of both, our implementation, and a standard implementation of the Ada *PLE*.

```
declare
   subtype my_loop_range is integer range 1..N;
   procedure my_loop_body (my_iterate : my_loop_range) is
   begin
      S_1;
      ⋮
      S_n;
   end my_loop;
   -- This instantiation corresponds to the execution
   --     of all N iterations.
   package my_minitask is new
            gen_minitask (my_loop_range,my_loop_body);
begin
   if not my_minitask.success then
      -- Handle error condition.
   end if ;
end;
```

Figure 6.2: A Parallel Loop Using the Gen_Minitask Idiom (GMI)

for the parent thread to inspect the status of the tasks that play the role of iterates; the conditional in Figure 6.2 cannot be written in Figure 6.1. For this reason, we do not include the instructions associated with executing this conditional in computing its overhead. However, we do account for the allocation and recording of status information that is used when executing this conditional. The GMI also provides two functions that allow the parent thread to inspect the completion status of each iteration. It thereby provides a more robust version of a parallel loop, in addition to a reduction in overhead.

The rest of this chapter is organized in the following manner. First we present the assumptions that underly our analysis. Next we analyze the overhead associated with the Ada PLE and a tree-based variant. This is followed by a similar analysis of the gen_minitask idiom. We conclude the chapter by comparing both

```
declare
    task type iteration_task is
        entry get_iterate(iterate : in integer);
    end iteration_task;

    par_loop : array (1..N) of iteration_task;

    task body iteration_task is
        my_iterate : integer;
    begin
        accept get_iterate(iterate : in integer) do
            my_iterate := iterate; -- Acquire iterate from parent.
        end get_iterate;
        S1
        ⋮
        Sn
    end iteration_task;

begin           -- All "iterations" begin to execute.
    for i in 1..N loop -- Iterations are distributed sequentially.
        par_loop(i).get_iterate(i);
    end loop;
end;            -- Wait for all iteration_tasks to terminate.
```

Figure 6.1: An Ada Parallel Loop Equivalent (PLE)

Chapter 6

Performance Analysis

In this chapter we analyze the performance of three implementations of a parallel loop written in Ada. The first two methods represent standard ways of expressing such a parallel loop. We describe the amount of overhead that is required to execute both of these methods. These figures are compared to the execution overhead of our implementation using the gen_minitask idiom. We show how the latter overhead compares favorably with the other two. We then examine issues of granularity and efficiency for all three forms of parallel loops, supporting the conclusion that our idiom provides a more efficient version of loop-level parallelism.

We use the standard means of expressing a parallel loop in Ada as shown in Figure 6.1. Throughout the rest of this chapter, we refer to this code as an Ada "parallel loop equivalent" or PLE. Figure 6.2 contains the corresponding example of the gen_minitask idiom or GMI.

Although both versions of the parallel loop offer similar semantics, the GMI provides an increase in functionality. In the Ada PLE there is no mechanism

Function Call	Value Returned
my_mini.task_completion(i);	my_mini.task_comp(i)
my_mini.task_exception(i);	my_mini.task_exc(i)

Note: iteration_range refers to the type parameter that was specified for the instantiation of my_mini.

Table 5.7: Implementation of the task_completion and task_exception Subprograms

The two functions, task_completion and task_exception are implemented in a straightforward manner. Each call is translated to the appropriate dereference operation into the two run time storage arrays: task_comp and task_exc. returning the appropriate value to the caller. Table 5.7 specifies the implementation details for these two functions. It assumes that the gen_minitask instantiation is called my_mini.

```
procedure create_and_execute_minitask
    INPUTS: node
begin
    Initialize local stack and tcb.
    Mark that this processor is executing a minitask.
    loop
        iterate := node.id + node.iteration_range'first;
        task_comp(iterate) := true;
        task_exc(iterate) := None;
        loop_body(iterate);
        -- Check if we are the last minitask to terminate.
        if fetch_and_add(node.parent.num_items,-1) = 1 then
            unblock parent;
            Mark that processor is NOT executing a minitask.
            exit;
        else -- Try to get another minitask.
            node := dequeue next minitask in this family;
            if node = NIL then
                Mark that processor is NOT executing a minitask.
                exit;
            end if;
        end if;
    end loop;
end create_and_execute_minitask;
```

Figure 5.9: Algorithm to Create, and Execute a Minitask

```
procedure create_minitask_family
    INPUTS: iteration_range, loop_body
    OUTPUTS: success
begin
    success := alloc(1);
    if alloc is unsuccessful then
        raise STORAGE_ERROR in instantiating declarative part.
    end if;
    success := true;
    Let N = iteration_range'last - iteration_range'first + 1.
    task_comp := alloc(N);
    task_exc := alloc(N);
    if either alloc is unsuccessful then
        raise STORAGE_ERROR in instantiating declarative part.
    end if;
    Mark exception_type, task_completion and task_exception
        visible to the instantiator.
    if processor is already executing a minitask then
        Execute minitasks sequentially.
    else -- Execute minitasks in parallel.
        my_tcb.num_items := N;
        node := alloc(MINI_RTS_NODE_SIZE);
        if alloc is unsuccessful then
            raise STORAGE_ERROR.
        end if;
        node.mult := N;
        node.proc := loop_body;
        node.range := iteration_range;
        node.parent := my_id;
        node.type := CREATE_MINI;
        enqueue(node);
        block until a minitask child unblocks us.
    end if;
end create_minitask_family;
```

Figure 5.8: Instantiating Task's Algorithm

```
procedure handle_exception
    INPUTS: exception, iterate
    MODIFIED: success, task_comp, task_exc
                    of current instantiation.
begin
    case exception is
        when CONSTRAINT_ERROR =>
            task_exc(iterate) := Constraint;
            task_comp(iterate) := false;
            success := false;
        when NUMERIC_ERROR =>
            task_exc(iterate) := Numeric;
            task_comp(iterate) := false;
            success := false;
        when PROGRAM_ERROR =>
            task_exc(iterate) := Program;
            task_comp(iterate) := false;
            success := false;
        when STORAGE_ERROR =>
            task_exc(iterate) := Storage;
            task_comp(iterate) := false;
            success := false;
        when TASKING_ERROR =>
            task_exc(iterate) := Tasking;
            task_comp(iterate) := false;
            success := false;
        when others =>
            task_exc(iterate) := Others;
            task_comp(iterate) := false;
            success := false;
end handle_exception;
```

Figure 5.7: Minitask Exception Handling Algorithm

Consider the case where an exception is raised and handled inside of loop_body. Under this scenario, the standard implementation ensures that control passes to the appropriate exception handler; no special implementation is required.

Now consider the case when an exception is raised inside of the minitask, but is *not* handled by the loop_body procedure. A "normal" implementation of a minitask would install a handler upon entering the body of this task. This handler would contain pointers to instructions to be executed depending on what exception is raised. Since the minitask handler is known, we can create a predefined handler that satisfies the semantics of the handler specified in the body of the minitask. Using this approach we are able to avoid the overhead involved with calling the procedure, record_exception; we execute its instructions inline.

Figures 5.8 and 5.9 summarize the implementation described in this chapter. Figure 5.8 specifies the actions that the instantiating task executes, while Figure 5.9 describes the manner in which each minitask is created, activated, and commences execution. In the latter algorithm we assume that a process which is not currently blocked on a minitask has dequeued a minitask item from the queue. This item with multiplicity N is enqueued by the former algorithm.

Upon returning from the loop_body procedure the minitask algorithm attempts to begin the execution of another minitask. If no more minitasks exist, this routine terminates.

If an exception is raised or propagated to the body of the routine in Figure 5.9, it is handled as previously described; a default minitask exception handler is present to handle these exceptions appropriately.

```
package body gen_minitask is
   ⋮
   -- Create a fetch_and_add variable for minitask initialization.
   package iteration_counter is new
      gen_beacon(integer(iteration_range'first));
   ⋮
begin
   declare
      loop_tasks : array(iteration_range) of minitask;
   begin
      null ;
   end;
end gen_minitask;
```

Figure 5.6: The Instantiation of Gen_Beacon Revisited

can be raised during the instantiation itself:

- an exception raised during the elaboration of the package declaration, and
- an exception raised during the execution of the package body.

If an exception is raised during the elaboration of the generic package declaration, this exception is propagated to the declarative part that instantiated the generic (ARM 11.4.2(7)). If an exception is raised while elaborating the declarative part of gen_minitask, our implementation simply propagates this exception, and abandons any further action concerning this instantiation.

When an exception is raised during the execution of the package body of the minitask generic, different semantics are in effect. As mentioned in section 4.5.2, the manner in which this type of exception is handled depends on where in the minitask the exception is raised and whether an exception handler exists for the generic parameter, loop_body.

Synchronization Point	Action†
Start of activation	Flush activator's cache
End of activation	None
Start of faa rendezvous	None
Start of any other rendezvous	Flush both tasks' caches
End of faa rendezvous	None
End of any other rendezvous	Flush both tasks' caches
Completion of task's execution	Flush terminating task's cache

†"Flush Cache" refers to flushing all Ada shared variables that have been written to since the last cache flush.

Table 5.6: Cache Actions Required for Minitasks

Theorem 5.4 *The instantiation of* iteration_counter *requires no run time action to be performed.*

Proof Consider iteration_counter, an instantiation of the gen_beacon generic package as shown in Figure 5.6. This instantiation provides three subprograms: read, write, and faa. In the body of each minitask object, only the faa function is called. From the discussion in the previous paragraph and Section 5.5.2, we have concluded that this call need not be performed. As this is the only call that accesses the underlying beacon variable, no allocation of the beacon variable, and therefore, the gen_beacon_struct need occur. Hence, instantiating iteration_counter does not require any run time action. ¶

5.5.4 Handling Exceptions

This section describes the method employed to handle exceptions that may arise during the execution of an instantiation of gen_minitask. Two types of exceptions

occur: the current value of the beacon shared variable is saved to be returned, and the beacon shared variable is incremented by the appropriate value. As the beacon shared variable is actually a local variable to the beacon task, no access to an Ada shared variable is performed during the rendezvous. For this reason, no cache flush is required at this time.

If other minitask rendezvous exist, a cache flush is required if any Ada shared variables have been updated during the rendezvous.

At the completion of a task's execution: When a minitask completes its execution it can potentially have reference Ada shared variables. These variable could have been cached locally and used by the instantiating task. Therefore, when a minitask completes its execution, any Ada shared variables that have been store locally are flushed into global memory.

These results are summarized by Table 5.6.

In Section 5.5.2, we mentioned that the function call to faa can be optimized away. However, we must note that this function call led to a rendezvous, and thus, a synchronization point for Ada shared variables. Although this operation is no longer needed, we must still ensure that the proper semantics are upheld with respect to any Ada shared variables. Specifically, if any local copies of an Ada shared variable exist, they must be flushed at the point where the rendezvous would have occurred. As described above, no local copies of Ada shared variables can exist, allowing for the cache flush, too, to be optimized away.

The following theorem allows for another optimization to be performed.

The start of an activation is also a synchronization point with the task that causes this activation, the task that instantiates the gen_minitask generic. Since this task can possess local copies of Ada shared variables, they must be flushed at the start of minitask activation.

At the end of its activation: Since no Ada shared variable is accessed during the activation of a minitask, no cache flush is required. Similarly, as the instantiating task does not access any variables since the start of this activation, no cache flush is required for this task.

At the start of a rendezvous: Since minitasks do not contain entry points, they can only execute rendezvous when they call another task. As we saw in Chapter 3, the call to the faa function results in a rendezvous. In the general case, this rendezvous would require a cache flush of any of the caller's Ada shared variables. However, this statement is the first statement executed by a minitask; no local copies can exist. Thus, no cache flush is required for this synchronization point.

If the procedure parameter to gen_minitask contains tasks, then other rendezvous are possible. Under this scenario, the procedure would declare a task and then call one of its entries. This call constitutes a synchronization point for any Ada shared variables this procedure may have accessed. Therefore, at this time any local copies of these variables must be flushed.

At the end of a rendezvous: Between the beginning and end of the rendezvous that is executed by the call to the faa function, only two operations

> The execution of the program is erroneous if any of these assumptions is violated."

A synchronization point for a shared variable is defined in the following:

> "Two tasks are synchronized at the start and end of their rendezvous. At the start and at the end of its activation, a task is synchronized with the task that causes this activation. A task that has completed its execution is synchronized with any other task." ARM 9.11(2)

As mentioned in ARM 9.11(7) an implementation is allowed to keep local (cache) copies of an Ada shared variable associated with a task, flushing them to global memory at the task's synchronization points. Although the gen_minitask and gen_beacon generic packages do not contain any Ada shared variables, we cannot guarantee that a program that uses these generics does not contain these variables. Therefore, in our implementation of an instantiation of the gen_minitask generic, we must ensure that proper semantics are upheld in respect to these Ada shared variables.

Note that although minitasks are not visible to the users of the gen_minitask generic, the reference manual nevertheless requires that the above mentioned synchronization points be honored with respect to Ada shared variables.

In considering how these synchronization points affect the minitask implementation, we consider each potential point in turn.

At the start of its activation: The beginning of a minitask activation is the first synchronization point a minitask reaches. Since no memory accesses have been made by the minitask, no local copies exist; no cache flushes need to be performed for the activating minitask.

Theorem 5.3 allows us to remove the constraint check that is associated with the faa call. Since the identity values returned by the dequeue operation of the Ada/Ed RTS is a value starting at one [Hum88], our implementation simply adds iteration_range'first to this identity to obtain the proper iterate. This addition replaces the faa function call.

5.5.3 Ada Shared Variables

In Section 3.3.1 of Chapter 3 we discussed how Ada shared variables and the gen_beacon package interact. In this section we explore how the potential use of these variables affect the implementation of the gen_minitask package.

Recall from Chapter 2 that an Ada shared variable is a scalar or access variable that is not declared as pragma shared, but is nonetheless used in a shared fashion, i.e. by several tasks without explicit synchronization. The manner in which these types of shared variables can be used is described in ARM 9.11(3-6):

> "For the actions performed by a program that uses shared variables, the following assumptions can always be made:
>
> - If between two synchronization points of a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.
> - If between two synchronization points of a task, this task updates a shared variable whose type is a scalar of access type, then the variable is neither read nor updated by any other task at any time between these two points.

that uniquely distinguishes the task to be created.

At the heart of the parallel queue algorithm is the fetch_and_add primitive.[6] Although the value that is returned by this fetch_and_add operation is not available to the Ada programmer, it nevertheless does exist. The first operation of a minitask is to acquire a unique identity via the fetch_and_add construct. Since this identity has already been obtained by the dequeue operation, *there is no need to execute a* fetch_and_add *operation at the time a* minitask *is created.*

As suggested by the code in Figure 5.5, after executing the fetch_and_add operation, a constraint check is made to ensure that the value returned by faa is in the bounds of iteration_range. However, the following theorem allows this check to be eliminated.

Theorem 5.3 *The value returned by the* faa *function in the body of* minitask *is always within the range* iteration_range'first ... iteration_range'last.

Proof Upon instantiation of the gen_beacon package, the initial value supplied for the beacon variable is iteration_range'first. The first call to faa returns this initial value, while incrementing the beacon variable by one. By inspecting the body of the minitask we see that exactly one call to faa is made for each minitask. Since iteration_range'last - iteration_range'first + 1 minitasks exist, precisely this number of calls to faa are made. Since each call provides a unit increment, the values iteration_range'first, iteration_range'first + 1, ... iteration_range'last are returned to each of these calls. Therefore, the values returned by the faa satisfy the constraint specified by the iteration_range type. ¶

[6] For more information about this algorithm see [Hum88].

```
task body minitask is
    iterate : iteration_range;
    procedure record_exception(exception_raised : exception_type) is
    begin
        task_status(iterate).exception_kind := exception_raised;
        task_status(iterate).completed := false;
        success := false;
    end record_exception;
begin
    iterate := iteration_range(iteration_counter.faa(1));
    loop_body(iterate);
    exception
        when CONSTRAINT_ERROR => record_exception(Constraint);
        when NUMERIC_ERROR => record_exception(Numeric);
        when PROGRAM_ERROR => record_exception(Program);
        when STORAGE_ERROR => record_exception(Storage);
        when TASKING_ERROR => record_exception(Tasking);
        when others => record_exception(Other);
end minitask;
```

Figure 5.5: The Body of Minitask Revisited

minitask to obtain a unique identity in the appropriate iteration range. The value returned is converted into an integer type and assigned to the local variable, iterate.

As specified in Chapter 3, the function call to faa can be translated into a fetch_and_add operation. However, in the context of minitasks a more efficient implementation can be realized.

In our implementation, tasks are created and activated in parallel; the task performing the elaboration of loop_tasks enqueues an item on to a parallel queue. This item has a multiplicity corresponding to the number of tasks to be created. As available processes perform a dequeue operation, the multiplicity of this item is decremented. The processor that performs the decrement receives an identity

task type. Since the minitask task type is specified without a priority, a predetermined priority is assigned to it by the implementation, eliminating the need for this field in the tcb.

rdv This boolean flag is set when a task executes an open accept statement. It signifies that a calling task may participate in a rendezvous immediately. Since minitasks contain no entry points, no accept statements can be executed, rendering this flag unnecessary.

num_entries This field contains the total number of entries associated with a task. Since a minitask does not contain any entries, this field can be eliminated.

save_prio This field is used to save the priority of a calling task when it performs a rendezvous with this task. Since minitasks do not have any entries, it need not allocate storage for any potential callers' priorities.

tcb_serviced This field corresponds to the current calling task of an entry. Once again, since minitasks do not contain any entries, this field is not required.

5.5.2 Further Reducing Beacon Overhead

We have just seen how to reduce the amount of storage associated with a minitask. Now we turn out attention to the operations a minitask must perform before it executes the "sequence of statements" given by the loop_body procedure.

Consider the body of the minitask as shown in Figure 5.5. The first action an activated minitask performs is the call to the faa function. This allows a

FIELD NAME	TYPE	USAGE
action	integer	action executing when blocked
entry_item	pointer	pointer to item in entry queue
current_entry	pointer	pointer to current entry queue
event	integer	event to unblock (set by others)
first	integer	used for task termination
io_item	pointer	current io_item (made when delay)
num_items	integer	number of items
num_noterm	integer	number of nonterminatable direct dependents
num_deps	integer	number of nonterminated direct dependents
num_events	integer	number of pending events
next	integer	next task to activate or previous task being serviced by this task

Table 5.5: Tcb Fields Needed for Minitasks that Contain Tasks

directly. ¶

As mentioned in Chapter 3, a minitask can be aborted *indirectly* if its master task is aborted. However, in this case there is no task that retains visibility to the minitask, eliminating for any abort-related context.

We now turn to those fields not included in Tables 5.4 or 5.5. With each field we give a brief description of why it is not needed in the tcb of a minitask.

abnormal This boolean is set to true when a task reaches an abnormal state; it is either aborted or an exception is raised during its execution that is not handled. By theorem 5.2, a minitask cannot be aborted directly. When a minitask is aborted Since all exceptions are caught by the handler located in each minitask, (see Figure 5.5), this field is not needed.

priority This is the priority of a minitask. A priority is associated with a

NAME	TYPE	USAGE
id	integer	index of task in rts item
master_task	integer	task number of the master task
master_block	integer	number of the master block
block_ptr	pointer	currently executing block frame
exception	integer	current exception raised
parent	integer	tcb index of parent task
template_base	integer	task template base address
template_offset	integer	task template offset
brother	integer	next direct dependent of master construct
rts_item	pointer	current rts item
tcb_status	integer	task's current status, set before blocking

Table 5.4: Tcb Fields Needed for Minitask Implementation

can call an entry of a minitask. Therefore, we need only keep status information in the tcb that pertains to entry calls, rather than entry management. Table 5.5 specifies the additional fields that need to be included.

Before we discuss those fields that can be eliminated from the tcb of a minitask, we prove the following theorem:

Theorem 5.2 *A* minitask *cannot be aborted directly.*

Proof A task is allowed to abort any task (ARM 9.10(10)). However, in order to abort a task directly, the name of the task must be supplied (ARM 9.10(2)). Therefore, only tasks that are visible may be aborted directly. To abort a minitask, another task would need to name the corresponding task object, loop_tasks(i), for some i. This array of tasks is only visible inside of the block statement of the gen_minitask generic package. Since no **abort** statement exists anywhere in this statement, we conclude that minitasks cannot be aborted

NAME	TYPE	USAGE
template_base	integer	task template base address
template_offset	integer	task template offset
brother	integer	next direct dependent of master construct
block_ptr	pointer	currently executing block frame
exception	integer	current exception raised
id	integer	index of task in RTS item
master_task	integer	task number of the master task
master_block	integer	number of the master block
num_entries	integer	number of entries
parent	integer	tcb index of parent task
rts_item	pointer	current rts item
tcb_status	integer	task's current status, set before blocking
action	integer	action executing when blocked
entry_item	pointer	pointer to item in entry queue
current_entry	pointer	pointer to current entry queue
event	integer	event to unblock (set by others)
first	integer	used for task termination
io_item	pointer	current io item (made when delay)
num_items	integer	number of items
num_noterm	integer	number of nonterminatable direct dependent
num_deps	integer	number of nonterminated direct dependent
num_events	integer	number of pending events
abnormal	boolean	flag set when task is aborted
priority	integer	task priority
rdv	boolean	flag set when task executes an open accept
save_prio	integer	save area for priority
tcb_serviced	integer	current calling partners chain
next	integer	next task to activate or previous task being serviced by this task

Table 5.3: The Ada/Ed Task Control Block

for the management of a regular Ada task are not necessarily needed for the management of a minitask; these fields can be omitted from the tcb of a minitask. Table 5.4 gives the fields that are required to manage the context of a minitask.

5.5.1 Parallelism Within a Minitask

As mentioned in the previous section, a programmer may find it useful to nest instantiations of the gen_minitask generic package. Under this scenario, the loop_body procedure would contain its own instantiation, thereby creating its own minitask family. Matrix multiplication as described in Appendix A is one example of this type of application.

Another common example of nested parallelism is when a parallel library package is used inside of a program that already contains loop-level parallelism. While the additional parallelism may not necessarily improve performance, it should nevertheless be executed correctly. For this reason our implementation of the gen_minitask idiom supports full nesting of gen_minitask instantiations.

Although the intended usage of the gen_minitask package is to simulate a parallel loop, there is nothing to prevent the threads of the minitask, represented by the loop_body procedure, from creating and activating other tasks. Since these tasks can be of any type, they can contain entry points. In particular, they can allow a minitask to call an entry of one of its child tasks. While this may have a detrimental effect on the execution performance of a minitask, it is nonetheless valid. Therefore, the tcb of a minitask must be expanded.

Note that this form of communication is only one-way. Since we have assured that minitasks do not contain any entry points, no task, specifically a child task,

contexts [FH90,Cyt85,KW85].

An alternative scheme would be to postpone the decision regarding the manner in which the inner loop is executed until run time; the inner loop would be executed in parallel if the number of available processors is sufficient to improve program performance. Otherwise, the inner loop would be executed sequentially. This scheme is similar to the transformations described by Byler, et. al. in [BDH+87].

The second potential disadvantage of this scheme is the extra storage that it requires. In a standard implementation of Ada tasks, a stack is allocated for each *task*. In our approach we allocate a stack for each *processor*, regardless of the number of tasks. While this approach does indeed use excess storage when the number of tasks in a program are less than the number of processors, our intended domain of applications will almost always have *at least* as many tasks as processors. Under this more likely scenario, our approach *reduces* the amount of stack storage required.

5.5 Reducing Minitask Overhead

The previous section described a mechanism to remove the synchronization point that exists between task creation and task activation. In this section we consider the overhead that is associated with each minitask, and describe how our implementation reduces it.

Each executing or blocked task in an Ada implementation has a task control block (tcb) associated with it. For a regular Ada task in the Ada/Ed run time system (rts), the tcb is given in Table 5.3. Many of the fields that are required

```
declare
   subtype outer_range is integer range 1..M;
   procedure outer_loop_body(outer_id : outer_range) is
      subtype inner_range is integer range 1..N;
      procedure inner_loop_body(inner_id : inner_range) is
      begin
         -- Instructions nested inside of both "loops" go here.
      end inner_loop_body;
      package inner_minitask is new
               gen_minitask(inner_range,inner_loop_body);
   begin
      if not inner_minitask.success then
         -- Handle error.
      endif ;
   end my_loop;
   package outer_minitask is new
            gen_minitask(outer_range,outer_loop_body);
begin
   if not outer_minitask.success then
      -- Handle error.
   endif ;
end;
```

Figure 5.4: Nested Parallel Loops using the Gen_Minitask idiom

The first potential disadvantage of this approach is that if a minitask blocks its execution, the processor it is executing on is unable to execute other minitasks; the minitask stack for that processor is already in use. To better understand the severity of this limitation, we consider the circumstances that would cause a minitask to block.

Consider the case where an instantiation of the gen_minitask package is used strictly as a parallel loop construct; the loop body contains a simple sequence of statements. Used in this fashion, the only reason for a minitask to block is if it performs I/O,[4] an event that we expect to happen infrequently.

However, since the body of a minitask is specified by a procedure (loop_body), it may contain other instances of the gen_minitask generic package (see Figure 5.4). As a consequence of this, a minitask may need to block its execution while it awaits the completion of the activation of its subtasks.

However, in most cases this nesting of tasks will create more tasks than there are processors, giving the *appearance* of increased parallelism. If we attempt to execute these nested tasks in parallel, we will most likely *reduce* program performance. Therefore, our implementation attempts to find "useful parallelism" by executing nested occurrences of minitasks sequentially.[5]

This approach directly corresponds to executing the outer of two parallel loops in parallel, while the inner loop is executed sequentially. This scheme reduces the number of fork and join points (from N to 1) in addition to increasing the size of each parallel thread. This concept of reducing parallelism to increase program performance has been called *chunking*, and is described in several other

[4] We assume scheduler interference was a minitask begins its execution.

[5] The concept of "useful parallelism" is discussed in [Cyt85].

Since all stack allocations are made before a program begins its execution, there is no need to request storage for this purpose during the instantiation of the gen_minitask package. All other storage requests are performed by the instantiating task *before* any minitasks are enqueued.[3] Once these requests have been satisfied, we are assured that STORAGE_ERROR will not be raised during the elaboration of loop_tasks. Applying Theorem 5.1, we combine task creation and task activation.

In addition to allowing the combination of task creation and activation, this scheme also offers the following advantages:

- All minitask stacks reside in the local memory of the processor that is executing it. Hence, all stack and tcb operations are local accesses.
- The total number of stacks used for the execution of minitasks in an application is bounded by the number of processors, not by the number of minitasks.

5.4.3 A Critique of the Stack Recycling Scheme

In the previous section we discussed some of the advantages of using the stack recycling scheme. In this section we critique this scheme, discussing the potential disadvantages. We shall show that these disadvantages are either insignificant or can be overcome by enhancing our scheme. The two disadvantages we discuss both rise from the fact that minitask stacks are allocated on a per processor basis rather than one per minitask.

[3] In our implementation, the initialization of the task_status array is performed by each minitask avoiding a potential bottleneck.

If this storage request is successful, we know that STORAGE_ERROR will not be raised for this elaboration. By applying Theorem 5.1, we can combine task creation with task activation. Each minitask could then use its identity to index into the pool of storage to obtain its own section.

Although this approach does remove the synchronization point associated with task initiation, there is one major drawback: since the processor that allocates the memory is not necessarily the same one that uses it, this chunk of storage must be allocated in shared memory. Consequently, all stack operations that a task undertakes would need to suffer the performance penalty of a shared memory access. Since accessing shared memory on a machine such as the RP3 can take up to ten times as long as an access to local memory, this approach is rejected.

5.4.2 Stack Recycling

In this section, we suggest an alternative stack allocation scheme, called *stack recycling*, which satisfies the condition needed to apply Theorem 5.1.

Under the stack recycling scheme, each processor has associated with it a stack which is used exclusively for the execution of minitasks.[2] This "minitask stack" is allocated by a processor as part of that processor's initial execution, before it attempts to execute any tasks. Once this allocation has been performed, a processor can execute a minitask using its local minitask stack. When the minitask terminates, this stack is recycled by using it for the next minitask that is executed by this processor.

[2]Recall that a "stack" also includes a tcb.

we attempt to ascertain if an elaboration of loop_tasks is going to raise STORAGE_ERROR before we begin the elaboration.

Consider how STORAGE_ERROR could be raised during the elaboration of loop_tasks. As was noted in the proof of lemma 5.1, storage is requested at several places throughout this elaboration; storage is also allocated for the array and to manage the new task's status (the task control block) and to store values (its stack). In our implementation, these requests can combined. If this combined request fails, STORAGE_ERROR is raised. However, we would like to determine this *before* we begin the elaboration.

If we implement this elaboration in a straightforward manner, we will not be able to determine if STORAGE_ERROR is raised until after all the storage requests are made, i.e. after task creation is performed. Thus, under this straightforward implementation a synchronization point exists; task creation cannot be combined with task activation. The next two sections describe methods in which the synchronization point can be removed.

5.4.1 One-Shot Allocation of Stacks

One implementation approach is have the parent task attempt to allocate a large chunk of storage for all minitasks. This approach takes advantage of the fact that the amount of storage needed for a minitask instantiation is a function of the number of minitasks that are to be created. Since this number is known *before* the elaboration of loop_tasks, the *total* amount of storage to be requested by this elaboration is known. Therefore, this storage can be requested by the instantiating task *before* the elaboration of loop_tasks commences.

Proof Assume STORAGE_ERROR is not raised during the elaboration of loop_tasks. By Lemma 5.1, we conclude that no exception is raised during this elaboration. By lemma 5.2, all minitasks are activated immediately following this elaboration. Thus, there is no need for a synchronization point between the creation and activation of the minitasks; this activation can be combined with the elaboration of loop_tasks. ¶

Before we discuss how our implementation utilizes this theorem, we note the existence of a second synchronization point associated with task initiation. This point occurs when the last task that has completed its activation unblocks its parent task. At this point the parent can begin executing the statements that follow the **begin** statement that prompted this activation.

In our implementation of the gen_minitask idiom, we are able to ignore this synchronization point. The only statement after the inner begin statement of Figure 5.3 is a null statement. Executing a null statement has no effect (ARM 5.1(5)), so the parent task is blocked until all of its dependent tasks have terminated. Thus, our implementation does *not* awaken the parent task until all of its dependent tasks have terminated. Although a synchronization point is not implemented, we nevertheless uphold proper Ada semantics.

We now consider how to remove the synchronization point referred to in Theorem 5.1. This theorem states that if we can determine that the elaboration of loop_tasks does not raise STORAGE_ERROR, then task creation and activation can be combined. While one cannot guarantee that STORAGE_ERROR will never be raised, we can tailor our implementation to take advantage of this theorem;

1. "The subtype indication or the constrained array definition is first elaborated."
2. "If the object declaration includes an explicit initialization, the initial value is obtained by evaluating the corresponding expression. Otherwise any implicit initial values for the object or for its subcomponents are evaluated."
3. "The object is created."
4. "Any initial value is assigned to the object."

Table 5.2: The Four Steps Involved in Elaborating an Object Declaration

Lemma 5.2 *If no exception is raised during the elaboration of* loop_tasks *in Figure 5.3, then all* minitask*s can be activated immediately following their creation.*

Proof Assume no exception is raised during the elaboration of loop_tasks. During this elaboration all minitasks are created. As there are no other items between this declaration and the begin statement, no exception can be raised after this item is elaborated. By (ARM 9.3(2)), once the declarative items of the block statement have been elaborated, the activation of all tasks declared in the corresponding declarative part commences. Hence, once all minitasks are created, they can be activated. ¶

These two lemmas allow us to prove the following theorem. This theorem forms the basis for our implementation of the block statement and allows us to eliminate the synchronization point.

Theorem 5.1 *If* STORAGE_ERROR *is not raised during the elaboration of* loop_tasks*, then the creation and activation of* minitask*s can be combined, and performed in parallel.*

The elaboration of an index constraint consists of evaluating the discrete range which can potentially raise NUMERIC_ERROR (ARM 3.6.1(11)). However, since the range itself is a type definition that has already been successfully elaborated as a generic parameter, we are assured that when control reaches this point of the text, this evaluation does not raise NUMERIC_ERROR. Elaborating the component subtype, minitask, creates that subtype.

Summarizing this step, we see that the only exception that can be raised is STORAGE_ERROR. Due to insufficient storage for the creation of any of the objects mentioned. (ARM 11.1(8)).

Step 2: Since no explicit initialization is specified (or allowed) for the loop_tasks declaration, the first part of this step does not apply. The implicit initial value for a task designates a task. Since storage is allocated when a task is created, STORAGE_ERROR may be raised in this step.

Step 3: The creation of the array object, loop_tasks, can also raise STORAGE_ERROR. No other exception can be raised by this creation.

Step 4: Assigning the initial value of each task to the appropriate components of the array cannot raise any exception.

In each of the four steps, no other exception besides STORAGE_ERROR can be raised. Therefore, STORAGE_ERROR is the only exception that can be raised during the elaboration of the loop_tasks. ¶

5.4 Removing the Synchronization Point

This section describes how the two phase process of task creation and task activation can be combined into one phase. The optimization we describe removes the synchronization point between the time when a task is created and when it is activated. This optimization uses the first property, that the behavior of the declarative part that contains the task object is known.

Recall the origin of this synchronization point (see Section 4.2): a statically declared task object is not activated until all declarative items that follow it in the same declarative part have been elaborated successfully. If the elaboration of any of these items raises an exception, then the task is not activated. We provide an implementation of the block statement that removes the synchronization point, while still satisfying proper Ada semantics. We begin with two lemmas.

Lemma 5.1 STORAGE_ERROR *is the only exception that can be raised during the elaboration of* loop_tasks.

Proof The elaboration of an object declaration proceeds in the four steps described in Table 5.2. We consider each step, in turn, examing what exceptions might be raised.

Step 1: Since the object declaration we are considering is a constrained array, we consider the manner in which this entity is elaborated. Section 3.6(10) of the reference manual specifies that in addition to creating the array type and array subtype, the index constraint and the component subtype indication are elaborated.

the instantiating declarative part.

Now that we have specified the semantics of the block statement, we consider how to obtain an efficient implementation that satisfies these semantics.

In Section 4.2, two deficiencies in the Ada tasking model were specified that prevent the realization of efficient loop-level parallelism. In summary, they are:

- The synchronization point required between task creation and task activation.
- The storage overhead used to maintain a task's status.

The body of the gen_minitask package is constructed in such a way that optimizations may be performed to overcome these two deficiencies. Note the following properties about the block statement:

1. Once all minitasks have been created, they are immediately activated; there are no declarative items between the creation of these tasks and their activation.
2. Minitasks do not possess any entry points, nor can they be aborted directly. In addition, no priority is specified for the minitask.

In the next two sections, we considered the aforementioned deficiencies. We show how the properties mentioned in the previous paragraph are instrumental in removing these obstacles (as described in Section 4.2) to efficient loop-level parallelism.

```
begin
   declare
      loop_tasks : array(iteration_range) of minitask;
   begin
      null;
   end;
end gen_minitask;
```

Figure 5.3: The Sequence of Statements of the Gen_Minitask Body

to the instantiating block. Figure 5.3 specifies the sequence of statements for the body of the gen_minitask package.

As mentioned in Section 4.5.2, a block statement is the only statement contained in the sequence of statements section of the gen_minitask body. A block statement is executed in a straightforward manner:

> "The execution of a block statement consists of the elaboration of its declarative part (if any) followed by the execution of the sequence of statements." ARM 5.6(4)

The declarative part of the block statement contains only one declarative item, an array of minitasks. When this item is elaborated, the array and the appropriate number of tasks are created. Each element of the array is assigned an implicit initial value designating the corresponding task (ARM 3.2.1(11)).

After the array has been successfully elaborated, the elaboration of declarative part is complete. Since the declarative part contains task objects that were created, they are activated upon reaching the begin statement. From then on all minitasks execute in parallel. Before exiting the sequence of statements section of the block statement, the task that executed this block, the instantiating task, must wait for all minitasks to terminate. At this point, control is returned to

DECLARATIVE ITEM	VISIBLE?	IMPLEMENTATION ACTION†
type exception_type **is** ...	Yes	No Action Required.
success : boolean := true;	Yes	Allocate a boolean variable. Initialize it to true.
function task_completion ...	Yes	No action required.
function task_exception ...	Yes	No action required.
task type minitask;	No	No action required.
type task_status_record **is** ...	No	No action required.
task_status : **array**(iteration_range) **of** task_status_record;	No	Allocate $2N$ storage units.
package iteration_counter **is** ...	No	Allocate one storage unit.‡ Set it to iteration_range'first. Make read, write, and faa visible.
task body minitask **is** ...	No	No action required.
function task_completion **is** ...	No	No action required.
function task_exception **is** ...	No	No action required.

†When the compiler is created, it is initialized with the first four items.
‡Section 5.5 describes a method that eliminates these actions.

Table 5.1: The Elaboration of the Declarative Part of the Gen_Minitask

time action is required.

function task_completion is ...

function task_exception is ...

Subprogram bodies are elaborated in the same manner as task body as described in the following:

> "The elaboration of a subprogram body has no other effect than to establish that the body can from then on be used for the execution of calls of the subprogram." ARM 6.3(5)

As in the case with the minitask body, a "callable" boolean variable can be used to mark the elaboration of a function body. However, our implementation need not maintain this variable; these functions are not used within the gen_minitask generic and any use by the instantiating task must follow the instantiation itself. Since these function bodies are elaborated during this instantiation, no boolean variable or run time action is needed.

Table 5.1 summarizes the actions that we perform for the elaboration of the each item in the declarative part of the gen_minitask package.

5.3 Executing the Sequence of Statements

After the elaboration of the package specification and the declarative part of the package body, the sequence of statements associated with the package body is executed. Once these statements have been executed, control is returned back

However, although the time to perform each initialization is small, N initializations are required. As it is our goal to reduce run time overhead, an action that is dependent on the size of N is undesirable. We avoid this serialization point by postponing the initialization until the corresponding minitask is created. Upon being created each minitask initializes its element in the task_status array. These initializations are performed in parallel.

package iteration_counter is new gen_beacon(...)

This item is an instantiation of the gen_beacon package. Elaborating this instantiation involves extracting the 'first value from the iteration_range type, converting it to an integer, and using it for the gen_beacon instantiation. As described in Chapter 3, instantiating this package corresponds to allocating an integer variable and assigning it the initial value that is passed as its parameter. It also makes visible the three monitoring subprograms: read, write, and faa. Section 5.5 describes how all these actions are performed implicitly and therefore, have zero run time cost.

task body minitask is ...

The elaboration of a task body "has no other effect than to establish that the body can from then on be used for the execution of tasks designated by objects of the corresponding task type" (ARM 9.1(5)).

This can be accomplished by setting a "callable" boolean variable associated with this task to true. However, by inspecting the the gen_minitask package, we can conclude that the minitask body is always elaborated before any of its uses are encountered. Therefore, no boolean variable or run

From this section we see that this elaboration corresponds to creating the array type, and elaborating the index constraint and the array subtype. As is the case with other types, actually creating the type can be omitted; this type is not to be used anywhere else.

Since the index constraint corresponds to the type parameter to the generic, it already has been evaluated in order to satisfy the parameter matching rules. Similarly, the component subtype, task_status_record, has also been elaborated. Thus, the first part of the elaboration of the array object does not require any action.

Turning to the second step, we note that no explicit initialization of the array is specified. However, the record type does specify an initial value for both of its subcomponents. Since these values are known constants their run time evaluation is not required.

The last two steps specified for the elaboration of the array object require run time action. The object must be created and its subcomponents assigned their initial values. To comply with this requirement, our implementation allocates a block of storage of $2N$ units in shared memory, where N = task_range'last - task_range'first + 1. It should be noted that we could pack the boolean and the enumerated type into four bits, allowing for a total allocation of $4N$ bits.

Once this storage has been allocated, it would normally be initialized; each of the completed fields would be set to true, while each of the exception_kind fields would be set to None. This initialization would complete the elaboration of this object declaration.

As is the case with other types, a *type template* is associated with the type task_status_record. Since this type is static, its type template is generated at compile time. This template is used when an object of this type is created or accessed.

task_status : array(iteration_range) of task_status_record;

This item is an object declaration that utilizes the previous type definition.

> "The elaboration of an object declaration proceeds as follows:
>
> 1. The subtype indication or constrained array is first elaborated. This establishes the subtype of the object.
> 2. If the object declaration includes an explicit initialization, the initial value is obtained by evaluating the corresponding expression. Otherwise any implicit initial values for the object or for its subcomponents are evaluated.
> 3. The object is created.
> 4. Any initial value is assigned to the object or to the corresponding subcomponent." ARM 3.2.1(4-8)

The first step specifies that the array is first elaborated. To determine the manner in which this is done, we consider the following:

> "The elaboration of a constrained array definition creates the corresponding array type and array subtype. For this elaboration, the index constraint and the component subtype indication are elaborated." ARM 3.6(10)

```
package body gen_minitask is
   task type minitask;
   type task_status_record is record
      completed : boolean := true;
      exception_kind : exception_type := None;
   end record;
   task_status : array(iteration_range) of task_status_record;
   package iteration_counter is new
      gen_beacon(integer(iteration_range'first));
   task body minitask is ...
   function task_completion is ...
   function task_exception is ...
begin
   ⋮
end gen_minitask;
```

Figure 5.2: The Declarative Part of Gen_Minitask Body

type task_status_record is record ...

task_status_record is a record type definition. The semantics of its elaboration are given by the following:

> "The elaboration of a record type definition creates a record type; it consists of the elaboration of any corresponding (single) component declarations, in the order in which they appear, including any component declaration in a variant part." ARM 3.7(7)

This quote specifies that elaborating this record type creates that type. Since this type can only be used within this package, we do not need to actually create it. However, when we subsequently use this type we do need to recognize it as a created type with the components and default values that are specified.

In summary, elaborating the package specification requires the designating of three entities (exception_type, task_completion, and task_exception) as elaborated in addition to allocating a boolean variable with an initial value of true. All four of these entities are visible to the instantiating block.

5.2 Elaborating the Declarative Part

Once the package specification is elaborated, we can begin elaborating the declarative part of the package body. In contrast to the package specification the declarative entities in the package body are not visible to the programmer; they cannot be used in any program specified by a user. Hence, our implementation of these entities does not necessarily have to process these items in the same manner as it would if they were visible. However, the elaboration and subsequent use of these entities must still uphold proper Ada semantics.

Figure 5.2 specifies the declarative part of the gen_minitask generic package body. In the remainder of this section, we consider each item, specifying the semantics of its elaboration.

task type minitask**;**

A task type is an example of a *task specification*.

> "For the elaboration of a task specification, entry declarations and representation clauses, if any, are elaborated in the order given." ARM 9.1(5)

Since the minitask specification does not contain any entry or representation clauses, this elaboration requires no action.

> enumeration type; this elaboration includes that of every enumeration literal specification." ARM 3.5.1(3)

success : boolean := true;

The elaboration of this object declaration involves the following four steps:

1. Elaborate the type, boolean.
2. Evaluate the initial value, true.
3. Create the object, success.
4. Assign the initial value, true, to success.

Since the type and initial value are predefined, the first two steps do not require any action. However, the last two steps, creating the object and assigning it an initial value do need to be performed at run time.

function task_completion ...

function task_exception ...

These two function declarations are elaborated in the same manner:

> "The elaboration of a subprogram declaration elaborates the corresponding formal part. The elaboration of a formal part has no other effect." ARM 6.1(6)

These statements specify that no action is required in elaborating these two subprogram declarations except recording the fact that they have been elaborated.

```
package gen_minitask is
   type exception_type is (None, Constraint, Numeric, Program,
                           Storage, Tasking, Other);
   success : boolean := true;
   function task_completion(iterate : in iteration_range) return boolean;
   function task_exception(iterate : in iteration_range) return exception_type;
end gen_minitask;
```

Figure 5.1: The Package Specification of Gen_Minitask

5.1 Elaborating the Package Specification

Consider the gen_minitask generic package specification as shown in Figure 5.1. Since the items declared in this specification are visible to an instantiating block, they are be treated as any other declarative item; the optimizations we can perform are limited. Therefore, we make these items available in the standard way other library packages (Calendar and System, for example) are processed; the compiler is initialized with the information contained in these packages when it is created. Any program that instantiates the gen_minitask generic gains visibility to the appropriate entities declared in the specification of this package.

For completeness, we review the semantics of the elaboration of the package specification. We shall see that although we are limited in the optimizations that we can perform, the run time overhead involved with elaborating these items is minimal.

type exception_type is ...

The elaboration of this enumerated type is specified by the following:

"The elaboration of an enumeration type definition creates an

2. The entities in the package body are *not* accessible outside of its body. Therefore, the manner in which these entities are used is also known at *compiler writing* time. Of particular interest, minitasks are used in a light-weight manner; they do not contain any entry points and cannot be aborted directly.

To determine the manner in which an instantiation of the gen_minitask generic package is elaborated, we consult the appropriate section in the reference manual:

> "For the elaboration of a generic instantiation, each expression supplied as an explicit generic actual parameter is first evaluated. ... Finally, the implicitly generated instance is elaborated."
>
> ARM 12.3(17)

Since the parameters of our generic are a type and a procedure, no expressions need be evaluated.[1] An elaboration of an instantiation corresponds to elaborating the gen_minitask generic package with the actual parameters substituted for the formals. This elaboration is broken into three phases: the elaboration of the package specification, the elaboration of the package body, and the execution of the sequence of statements associated with the package body. We describe each of these phases in the following three sections.

[1] At compile time, checks are performed to ensure that the actual parameters satisfy the constraints given by the formals.

Chapter 5

Gen_Minitask Implementation

This chapter describes the manner in which an instantiation of the gen_minitask generic package is implemented. Our goal in this implementation is to minimize common initialization, and to eliminate, if possible, all serial operations in realizing a "parallel loop" idiom. Ada semantics force the serialization of a number of operations, some of which are not obvious. Through the use of several optimizations, we attempt to either remove these operations all together, or perform them in parallel, while still upholding Ada semantics.

The implementation we describe is directly related to the usefulness of our idiom in obtaining efficient loop-level parallelism. To obtain this efficiency we make use of several optimizations. Underlying these optimizations are two principles:

1. As is the case with the gen_beacon generic, the contents of the gen_minitask generic are known at *compiler writing* time. This information is synthesized into our implementation to allow for an efficient implementation.

```
function task_completion(iterate : in iteration_range) return boolean is
begin
    return(task_status(iterate).completed);
end task_completion;

function task_exception(iterate : in iteration_range) return exception_type is
begin
    return(task_status(iterate).exception_kind);
end task_exception;
```

Figure 4.11: The Task_Completion and Task_Exception Functions

In turn, the sequence of statements of the gen_minitask body cannot be completed until this block statement is completed. Therefore, control is *not* returned to the instantiating task *until* all minitasks have terminated. In addition, the sequence of statements that corresponds to the declarative region that instantiated the gen_minitask package retains visibility to the completion status information.

4.5.3 The Completion Status Routines

In addition to creating the appropriate number of parallel threads, an instance of the gen_minitask package provides a boolean variable and two functions to help determine the status of a minitask family. We have already seen how the success variable is manipulated. In Figure 4.11, we specify the two functions task_completion and task_exception. We shall show in Chapter 5 how a call to these functions corresponds to a straightforward access to the task_status data structure to determine the behavior of a particular iterate.

```
package body gen_minitask is
   task type minitask;

   type task_status_record is record
      completed : boolean := true;
      exception_kind : exception_type := None;
   end record;

      -- The data structure to hold the each minitask's status:
   task_status : array(iteration_range) of task_status_record;

      -- Create a fetch_and_add variable for minitask initialization.
   package iteration_counter is new
            gen_beacon(integer(iteration_range'first));

   task body minitask is ... (See Figure 4.9)

   function task_completion is ... (See Figure 4.11)
   function task_exception is ... (See Figure 4.11)

begin
   declare -- This block is the minitasks' master.
      loop_tasks : array(iteration_range) of minitask;
   begin
      null;
   end;-- All minitasks terminate before this block is exited.
end gen_minitask;
```

Figure 4.10: The Improved Version of Package Body of gen_minitask

termination of these tasks can be made.

If we place the gen_minitask instantiation in its own block statement, we can guarantee that all minitasks terminate before executing any other statements. However, consider the case where the instantiating task wishes to discover the completion status of the minitask family, a common occurrence. Normally this would be achieved by either inspecting the success variable or by calling one of the two status reporting functions. Since these activities can not be performed until we are assured that the minitask family has completed its execution, we must wait until after the **end** statement of the proposed block statement. However, since the instance of the gen_minitask occurs in the declarative region of this block, the boolean variable and two functions are no longer visible once this block is exited.

Thus, this solution is not effective. By placing the instantiation inside the block statement we ensure that the minitask family terminates, but at the point where this termination takes place the information we desire is no longer available.

A solution that solves this problem is given in Figure 4.10. It involves the placement of the array declaration of minitasks. By placing this array object in a block statement *inside* of the gen_minitask package we achieve the best of both worlds; we can guarantee to an instantiating task that all minitasks will terminate at a point where visibility to the boolean and two functions is maintained.

By placing the array object inside a block statement we have effectively changed the master of these tasks from the instantiating task to the block statement itself. Thus, this block cannot be exited until all minitasks have terminated.

The Placement of the Minitask Array Object

Consider the placement of the declaration of the array of minitasks. At first glance, it seems natural to declare this array in the declarative part of the gen_minitask package body as was done in Figure 4.8. Under this scenario, the tasks begin their execution once the begin statement of the package body is reached.

However, we must consider when these minitasks would terminate. The reference manual specifies that

> "the instance of a generic package is a package." ARM 12.3(5)

Thus, an instance of the gen_minitask package is the same as declaring this package in the declarative region where the instantiation occurs. Unlike subprograms, packages cannot be a master of any task. The master of a task that is declared inside a package body is the nearest enclosing task, block statement, subprogram, or library package (ARM 9.4(1)). Since an instance of a generic package is not a library package, if we declared the minitasks in the declarative region of the gen_minitask package body, their master would be in the *instantiating* task, subprogram, or block statement.

While this in itself is not undesirable, the termination semantics it implies are. Consider the instantiation of the gen_minitask package. At instantiation time all minitask will begin to execute. However, when can the instantiating task be assured that all minitasks have terminated? The answer is clear, although undesirable. All minitasks must terminate upon the completion of the sequence of statements corresponding to the declarative region where the instantiation occurred. Thus, in the sequence of statement section no guarantee about the

Handling Exceptions in a Minitask

This section describes how exceptions are handled during the execution of a minitask. Of primary importance is where the exception is raised. If the exception is raised in the body of the minitask, it is handled by the appropriate case of the exception handler located in the minitask itself. This handler records the fact that an exception was raised by setting the success variable to false and marking the appropriate record element of the task_status array. Since all cases in the exception handler are similar, they combined into a procedure, record_exception.

If an exception is raised while executing the loop_body procedure, this procedure has the capability to catch this exception and continue processing. Since this procedure is provided by the programmer, it enables the programmer to specify the semantics of an iteration should an exception be raised. Possible scenarios include:

1. abandon executing this iteration by catching the exception and returning from the procedure,

2. continue execution of this iteration by catching it and execute alternative instructions,

3. abandon execution of this iteration and signal (via a shared variable) to all other minitasks to abandon their execution,

4. do not catch the exception and let it propagate to the minitask body.

If the last option is used (the default when no exception handler is specified) the exception is caught by the minitask's exception handler. The handler records this exception in the same manner as specified above.

gen_beacon package, iteration_counter. Recall that this instantiation provides a monitored variable onto which fetch_and_add operations can be performed efficiently in parallel. As each minitask calls the faa procedure with an increment of one, a unique identity is returned to it. This identity is used as the iterate for that minitask; it is passed as the parameter to the procedure parameter of the generic package.[4]

In this way an iteration of the parallel loop is executed. As our implementation of an instantiation of gen_beacon allows multiple faa calls to proceed in parallel, all minitasks execute fully in parallel; no bottleneck is present. Furthermore, the semantics of a parallel loop are realized. This satisfies constraint 3 of Table 4.1.

Note that the value returned from the faa function is converted to the subtype iteration_range before it is assigned to iterate. This is done to ensure type correctness with the local variable iterate and the formal parameter of the procedure parameter. We show how the range checking associated with this type conversion can be eliminated when we discuss gen_minitask implementation.

Although minitasks are not visible outside of their package, their execution can still be controlled. The procedure parameter corresponding to loop_body may access any visible object, allowing minitasks to collaborate on a computation. Furthermore, each minitask may decide to abandon its execution by simply exiting from its procedure. In this manner, one minitask may signal a whole family of minitasks to abandon their execution by setting a boolean variable that is visible to all minitasks.

[4]Note that the value returned to loop_tasks(i) is *not* necessarily i.

```
task body minitask is
   iterate : iteration_range;
   procedure record_exception(exception_raised : exception_type) is
   begin
      task_status(iterate).exception_kind := exception_raised;
      task_status(iterate).completed := false;
      success := false;
   end record_exception;
begin
      -- Acquire an iterate & increment counter atomically.
   iterate := iteration_range(iteration_counter.faa(1));
   loop_body(iterate);
   exception    -- Any unhandled exception raised
                -- in loop_body is caught here.
      when CONSTRAINT_ERROR => record_exception(Constraint);
      when NUMERIC_ERROR => record_exception(Numeric);
      when PROGRAM_ERROR => record_exception(Program);
      when STORAGE_ERROR => record_exception(Storage);
      when TASKING_ERROR => record_exception(Tasking);
      when others => record_exception(Other);
end minitask;
```

Figure 4.9: The Body of a Minitask

element in the iteration_range parameter as the initial value to the underlying fetch_and_add variable. Next, the bodies of the minitask, and the two status reporting functions are specified.

The body of the minitask is given in Figure 4.9. A local variable of type iteration_range is declared to hold the iteration value this task should execute. The only other item in the declarative part of this task is a local procedure. This procedure is used solely for convenience and is discussed when we deal with exceptions.

Two statements comprise the sequence of statements part of the minitask body. The first statement acquires a unique iterate by using an instance of

```
package body gen_minitask is
   task type minitask;

   type task_status_record is record
      completed : boolean := true;
      exception_kind : exception_type := None;
   end record;

      -- The data structure to hold the each minitask's status:
   task_status : array(iteration_range) of task_status_record;
   loop_tasks : array(iteration_range) of minitask;

      -- Create a fetch_and_add variable for minitask initialization.
   package iteration_counter is new
            gen_beacon(integer(iteration_range'first));

   task body minitask is ... (See Figure 4.9)

   function task_completion is ... (See Figure 4.11)
   function task_exception is ... (See Figure 4.11)
begin
   null;
end gen_minitask;
```

Figure 4.8: The First Version of Package Body of Gen_Minitask

status of the minitask family in the manner described previously.

Since the building blocks of a minitask instance, the subtype and the procedure, are declarative items, a declarative section must be entered in order to utilize our idiom. At first glance this seems to violate constraint 2 for our parallel loop idiom: that it should be able to be used in the same spot as a parallel loop. However, since Ada allows a block statement to be placed arbitrarily in a sequence of statements, this constraint is indeed satisfied.

4.5.2 The Gen_Minitask Body

In this section we present the body of gen_minitask package. The manner in which this package is constructed is important in our efficient realization of loop-level parallelism; it allows for an optimized implementation of an instance of this generic package.

The gen_minitask package as specified in Figure 4.8 declares two types and an array object. As mentioned previously, the minitask is the underlying task that executes an iteration. The record type, task_status_record, is used by the array object, task_status, as a data structure to store termination information about each minitask.

After this object declaration we declare the array of minitasks. These tasks begin their execution once the begin statement of the package body is reached. We shall see later that this declaration may be more effective if it is placed elsewhere.

The next item is the instantiation of the gen_beacon package which is used to to distribute task identities to each minitask in parallel. We supply the first

on the range of 1..100 is declared.[3] In the body of the gen_minitask package this range is used to distribute iterates to each task executing an iteration. As is the case with all integer ranges, the lower bound need not necessarily start at one. An application can have its iterations span over any integer range by simply constructing the appropriate integer range subtype.

The body of the parallel loop is represented by the procedure my_loop_body. The statements would have appeared in the parallel loop are placed in this procedure. The parameter of the procedure plays the role of the iteration variable. This procedure can reference any visible global variables, thus allowing minitasks to collaborate on a computation on composite data structures or to communicate via shared variables.

After the subtype and procedure are declared, the instantiation of the gen_minitask can occur. This package is constructed so that the following occur upon instantiation (barring an exception being raised):

1. All minitasks are created.
2. All minitasks are activated.
3. All minitasks receive a unique identity in the subtype range.
4. All minitasks execute the body of the procedure parameter.
5. All minitasks terminate.

At the point where control is returned back to the instantiating task, all minitasks have completed their execution allowing the instantiating task to inspect the

[3]Since the upper and lower bounds of this range need not be constant, a subtype is used even though in this particular case a type is sufficient.

```
declare
   subtype my_range is integer range 1..100;
   procedure my_loop_body(id : my_range) is
   begin
      S_1;
      ⋮
      S_n;
   end my_loop_body;
   package my_minitask is new gen_minitask(my_range, my_loop_body);
begin
   if not my_minitask.success then
      for i in my_range loop
         if not task_completion(i) then
            -- An exception was raised in the i^th iterate.
            -- We can use task_exception to determine
            --          which exception.
         end if;
      end loop;
   endif;
end;
```

Figure 4.7: A Typical Usage of gen_minitask

However, if success is false, the instantiating task may want to determine where the exception(s) was raised. To determine if an exception was raised for a particular iteration, the function, task_completion, is called. This function takes an iterate as a parameter, and returns a boolean value corresponding to whether execution of this iteration raised an exception. If an exception was raised in this iteration, task_exception can be called to determine the type of exception that was raised. The value returned by this function is of exception_type, an enumerated type which is made visible when gen_minitask is instantiated.

By providing this information our idiom satisfies the fifth constraint of a parallel loop idiom (Table 4.1). Namely, that a programmer should be able to determine exception information about the parallel loop and each of its iterations.

4.5.1 Gen_Minitask Usage

Before considering the body of the gen_minitask package we illustrate the manner in which this package can be used. Figure 4.7 gives a typical usage of the gen_minitask package. It simulates a parallel loop with one hundred iterations. The declarative region can be broken down into three parts:

1. The constrained subtype declaration,
2. The loop body declaration,
3. The instantiation of the generic package.

The first two declarations provide the parameters for the gen_minitask instantiation. Since one hundred iterations are desired a integer subtype constrained

```
with gen_beacon;
generic
   type iteration_range is range <>;
   with procedure loop_body(iterate : in iteration_range);
package gen_minitask is
   type exception_type is (None, Constraint, Numeric, Program,
                           Storage, Tasking, Other);
   success : boolean := true; -- Minitask family completion status
      -- These functions provide access to status information.
   function task_completion(iterate : in iteration_range)
            return boolean;
   function task_exception(iterate : in iteration_range)
            return exception_type;
end gen_minitask;
```

Figure 4.6: The Visible Part of Gen_Minitask

The visible part of the package contains information that aid an application in determining the completion status of both the minitask family and each minitask. Upon completing the execution of a minitask family, the instantiating subprogram can ascertain the following information:

- If an exception was raised in the minitask family
- If an exception was raised in a particular minitask
- What exception was raised for a particular minitask

This information is made available through a boolean variable, success and two functions, task_completion and task_exception. A typical use of this information would be for the instantiating task to first inspect the boolean variable, success, to determine if any exception was raised in the minitask family. If this boolean is true, then all minitasks completed successfully and normal processing can continue.

Since our idiom is to be used as a general means for expressing loop-level parallelism, it must be able to take parameters, specifically the loop body and the range of iterations to be executed. Since the loop body is a sequence of statements, it can be represented as a procedure as shown in Figure 4.5. This procedure contains one parameter, the iteration value of the loop it is executing. The body of the procedure contains the sequence of statements that would have appeared as the statements of the parallel loop. As is the case with a parallel loop, these statements can reference the value of the loop iteration.

To ease in our presentation, we note that the underlying tasks that execute a loop iteration are called *minitasks*. A minitask is simply an Ada task object. A collection or array of minitasks that simulate a parallel loop is referred to as a minitask *family*.

We construct our idiom in a similar fashion to the gen_beacon generic package discussed in Chapter 3. By using a generic package as the basis of our construct, we can group together the underlying tasks that execute the loop iterations with two functions and a boolean variable that summarize the completion behavior of the these tasks.

Figure 4.6 gives the gen_minitask package specification. It uses the gen_beacon package to distribute task identities to each task. We shall see the manner in which this is done when we discuss the body of this package. Two parameters are passed to the gen_minitask package: a type and a procedure. The type corresponds to an integer range over which the iterations of the parallel loop span. The procedure parameter corresponds to the body of the parallel loop. It contains one parameter of type iteration_range. This parameter is passed the iterate this procedure should execute.

1. It should be written in Ada.
2. It should be usable in the same spot as a parallel loop.
3. The semantics of the idiom should be the same as those of a parallel loop.
4. It should be constructed so that optimizations can be performed to obtain comparable efficiency of a parallel loop.
5. Since it is written in Ada, a programmer should be able to determine exception information about each "iteration" task, and about all "iterations".

Table 4.1: Five Constraints of a Parallel Loop Idiom

```
procedure loop_body(iteration_value : integer) is
begin
    S_1;
    ⋮
    S_n;
end
```

Figure 4.5: An Example of a Loop Body Procedure

correct transformations,[2] it ignores a vital asset in the design of programs: the programmer's knowledge of the program.

For these reasons, compiler detection of light-weight tasking is rejected.

4.4.3 The Idiom Approach

The method of choice in this work is to construct an Ada idiom such that its implementation can be optimized into an efficient realization of loop-level parallelism. Since this idiom is written solely in Ada, portability is maintained. This approach is similar to the pragma approach in that the *programmer* designates where loop-level parallelism is being employed. However, it differs in that we are able to provide an increase in functionality, as well as an efficient implementation. The remainder of this chapter describes this idiom and the increase in functionality that it provides. Its implementation is described in the Chapter 5.

4.5 Our Idiom – The Gen_Minitask Package

In this section the gen_minitask generic package is introduced. This package provides its users with an efficient means of obtaining loop-level parallelism *without* modifying the language. Before specifying this package, we specify several constraints a parallel loop idiom should satisfy (See Table 4.1).

Consider the first constraint. As the task is the only parallel construct in Ada, it will form the basis for any idiom we suggest. In the course of presenting our idiom, we show that all five of these constraints are satisfied.

[2] provided the compiler is correct, of course!

4.4.2 The Compiler Detection Approach

For a compiler to detect if a task is light-weight, it would have to determine if any heavy-weight tasking operations (rendezvous, abort, etc.) are performed. In order to detect *all* occurrences of light-weight tasking, tasks that could *potentially* perform heavy-weight operations would be considered in an attempt to determine whether these operations are actually performed at execution time. If they were not performed, this task could be deemed light-weight.

In order to determine if a task is going to perform a rendezvous or be aborted, knowledge of the actual execution is required. Therefore, detecting light-weight tasking, in general, is not possible. Despite this grim news, one may wish to limit the tasks that are detected to a subset of all light-weight tasks.

By taking this less ambitious approach, a compiler may be able to detect some uses of light-weight tasking: those tasks that do not *contain* any "heavy-weight" tasking operations. Although it is possible to statically check if a task type *contains* an accept statement or entry call, determining if the task can ever be aborted would require inspection of all *uses* of the task type. This type of analysis is not only undesirable, but, in general, impossible when the compilation unit is a library package.

Another problem with this approach, as is the case with parallelizing compilers, is that the programmer is dependent solely on the compiler for choosing the correct form of parallelism, be it a regular Ada task or a light-weight task. The programmer may know that a particular task is being used in a light-weight fashion, but since there is no means of communicating this fact to the compiler, it may go undetected. While this "hands off" approach to parallelism only makes

2. The burden of tasking efficiency can be kept on the compiler, by requiring it to *detect* when a task is used in a light-weight manner.

3. We can construct an Ada *idiom* that allows the efficiency of loop-level parallelism to be realized without sacrificing portability.

4.4.1 The Pragma Approach

By designating a task as being light-weight, the compiler can reduce the context that must be maintained for that task. However, consider what happens if that task subsequently attempts an operation that the light-weight task does not support. Assuming that the compiler does not attempt to detect these operations, an *error* will occur. Moreover, when this program is run on a machine that does not support this pragma the compiler will simply ignores the pragma and treats the task as a regular Ada task. If a heavy-weight operation is attempted no error will occur. Thus, this program exhibits different behavior depending on the system it is executed on and appears to be non-portable.

Hilfinger remarks that this hypothetical program is incorrect and "that the portability of incorrect programs has never been a serious concern" [Hil91]. He puts a further damper on the "lack of portability" argument by pointing out that the language maintenance body (ARG) has deemed that "pragmas *are* allowed to cause a program to become erroneous" [Hil91]. Thus, it appears that this approach is a reasonable one to pursue.

have been created. Therefore, this proposed solution has *not* eliminated the synchronization point, merely relocated it.

Although it seems that we have the same scenario as in the case of statically declared tasks, there is one key difference: the semantics of the allocator specify that task activation takes place *immediately* after the tasks have been created. When task objects are statically declared, this is not necessarily the case; if any declarative items follow the task declaration, they must be elaborated, without an exception being raised, before any task is activated.

In our quest to reduce the overhead of task initiation, the knowledge that activation is performed once creation has been successfully completed, may provide a source of optimization. As we shall see, this type of knowledge is used in the construction of the gen_minitask package.

4.4 Reducing Tasking Context

As previously discussed, a reason why the Ada task is not suitable for loop-level parallelism deals with its granularity. The functionality, and thus granularity, of the Ada task is not something that we have the power to modify. Therefore, we search for other means that a programmer can inform the compiler that a light-weight version of the Ada task can be used.

There are several ways this can be achieved:

1. By using the pragma facility, a programmer can designate a task type as being light-weight. In doing so the implementation can perform optimizations that reduce the context that is required for objects of this type.

```
declare
   task type worker is ...
   type loop_tasks is array (1..10) of worker;
   type ref_loop_tasks is access loop_tasks;
   ten_tasks : access_loop_tasks;
   task body worker is ...
begin          -- All tasks are activated
   ⋮
   ten_tasks := new loop_tasks;
   ⋮
end            -- Wait for all tasks to terminate
```

Figure 4.4: A Dynamic "Parallel Loop"

Consider the assignment statement. The right hand side of this statement contains a dynamic allocation of an array of ten tasks. This is processed in the following manner (ARM 9.3(6)):

1. The array is created.
2. The ten tasks of the array are created.
3. The ten tasks are activated.
4. The access value is assigned to the variable ten_tasks.

Although the first two steps may be interchanged, the reference manual clearly states that the activation of these tasks can not begin until initialization of the object has completed (ARM 9.3(6)). In particular, all the tasks must be created. These semantics are consistent with static tasking semantics; a distinct synchronization point exists during the initiation of a task object, so that if an exception has been raised before that point, then no tasks are activated. For statically declared tasks, this point is the **begin** statement of the frame that declares the task. For dynamic tasks, this point is right after the tasks

in a more efficient manner.

In order to be able to express loop-level parallelism in Ada, these two problems must be overcome.

4.3 Removing the Synchronization Point

As we have just seen there are two reasons why Ada tasks are not suitable for expressing loop-level parallelism:

- the synchronization point that is necessary between task creation and task activation,
- the significant context required to manage an Ada task.

In this section we attempt to overcome the first problem.

4.3.1 An Observation

A naive, but instructional approach to removing the synchronization point required between task creation and activation is to use tasks in a dynamic fashion. Consider the block statement shown in Figure 4.4. The declarative part contains three type declarations: a task type, an array of ten tasks, and an access to this array. After these declarations, an access object, ten_tasks, is declared, followed by the body of worker. Since the declarative part contains no task objects, upon reaching the begin statement, the sequence of statement is executed immediately; the task executing this block statement does *not* have to wait for any tasks to be activated.

the tasks are activated and begin their execution. The point at which this activation occurs depends on whether the task is declared statically or allocated dynamically. The motivation behind this is that if a problem arose *creating* one or more tasks, then the programmer probably does not want "normal" processing to continue, be it trying to create more tasks, or activating the ones that already have been created. While this two stage process is beneficial for handling exceptions, the synchronization point it mandates reduces the usefulness of Ada in specifying loop-level parallelism.

Unfortunately, this is not the only reason why Ada is seemingly unsuitable for loop-level parallelism.

A second deficiency in the Ada tasking model is its relative coarseness, as it is intended primarily for programming embedded applications with relatively small rate of intertask communication. As has been often noted, the tasking model of Ada is nevertheless very general [Fra87,Hil82b,Blu81]. While this generality provides the programmer with a powerful mechanism to express a program's communication and synchronization semantics, the overhead it requires is a hindrance in expressing light-weight parallelism.

By providing a sophisticated method of task communication and synchronization, an Ada implementation must maintain additional context (information associated with rendezvous, entry queues, task priorities, etc.). Since the full generality of the Ada task is not needed to specify loop-level parallelism – parallel loop iterations do not communicate with each other or specify different priorities – a heavy price is being paid for something that is not required. However, if a compiler can ascertain that a task will execute in a light-weight fashion (i.e. with no synchronization with other tasks), then it can implement this task

> "Should one of these tasks thus become completed during its activation, the exception TASKING_ERROR is raise upon conclusion of the activation of all of these tasks (whether successfully or not); the exception is raised at a place that is immediately before the first statement following the declarative part (immediately after the reserved word **begin**)." ARM 9.3(3)

This quote illustrates that the constraint that the master must wait for all dependent tasks to complete their activation before continuing its execution is indeed a practical one; if any of these tasks raise an exception the master must raise TASKING_ERROR instead of continuing its normal execution.

What happens if an exception is raised by more than one activating task?

> "Should several of these tasks thus become completed during their activation, the exception TASKING_ERROR is raised only once." ARM 9.3(3)

An implementation can satisfy this rule by maintaining a boolean for the master signifying if TASKING_ERROR should be raised upon conclusion of the all of the dependent's activation.

Notice that the reference manual takes a different view when a task raises an exception during its activation as opposed to when it is being created. This is because activation marks the beginning of a task's execution, and thus, once this has begun, abnormal actions in it should not affect a sibling task.

These quotes illustrate that the initiation of a task is a two stage process. First, the task objects are created, and then, provided no exceptions are raised,

> "If an exception is raised during the elaboration of the declarative part of a given frame, this elaboration is abandoned." ARM 11.4.2(1)

Therefore, task objects are not elaborated when the elaboration of a previous item raises an exception.

Next, we consider the scenario when an exception is raised *after* a task object has been elaborated, but *before* the corresponding task(s) has been activated:

> "Should an exception be raised by the elaboration of a declarative part or package specification, then any task that is created (directly or indirectly) by this elaboration and that is not yet activated becomes terminated and is therefore never activated." ARM 9.3(4)

As in the case when the exception is raised before the task object is elaborated, no tasks are activated.

We next consider the case when no exception has occurred, and a task object has been elaborated successfully. What happens if an exception is raised during task activation?

> "Should an exception be raised by the activation of one of these tasks, that task becomes a completed task; other tasks are not directly affected." ARM 9.3(3)

Thus, if an exception is raised during the activation of one of the tasks associated with the array ten_tasks, that task becomes completed, while the other nine tasks, in addition to one_task, are not affected.

How does an exception being raised during the activation of a task affect the master task?

Once again, consider Figure 4.3. The master of the single task, one_task, and the array of tasks, ten_tasks, is the block that declares it. Likewise, each of these eleven tasks depends on this block. Although the task that is created dynamically is also dependent on this block, it is for a different reason; namely, the access type definition is declared in this block. This is consistent with normal scope rules for objects created by an allocator.

The concept of task dependence is useful in determining the tasks a block must wait for before exiting. This is precisely all tasks that are dependent on that block, i.e.

- all task objects that were statically declared in that block statement, and
- all tasks that were created by an allocator (possibly in some inner frame) using an access type definition that was declared in this block.

Returning to the block statement in Figure 4.3, we see that the task executing this block statement must wait for the tasks associated with one_task and ten_tasks. In addition, any tasks associated with the access type, ref_worker, must also terminate prior to the completion of this block.

4.2.2 Tasks and Exceptions

Since exceptions are an integral part of Ada, a reasonable question to ask is:

How are tasking semantics affected by the presence of exceptions?

The reference manual defines how these two important features interact. First, it defines what happens if an exception is raised before or while a task object is elaborated:

a successful activation has occurred, an access to the value of the task is stored in ref_task. Execution of both the rest of the sequence of statements and the newly activated task proceed in parallel. Upon reaching the end statement, the task that is executing the block statement must wait for the twelve other tasks to terminate, if they haven't already done so, before proceeding.

Note that an initial value for the access variable ref_task could have been provided in the declarative part by using a call to the allocator. If this were the case, a task would be created and then immediately activated.

4.2.1 Task Dependence

The reference manual defines the related concepts of *task dependence* and a *master* of a task (ARM 9.4(1-3)):

> "Each task *depends* on at least one master. A *master* is a construct that is either a task, a currently executing block statement or subprogram, or a library package. The dependence on a master is a direct dependence in the following two cases:
>
> - The task designated by a task object that is the object, or a subcomponent of the object, created by the evaluation of an allocator depends on the master that elaborates the corresponding *access type definition*.
> - The task designated by any other task object depends on the master whose execution *creates* the task object."

4. "Any initial value is assigned to the object."

Thus, ten tasks are created and are associated with the array, ten_tasks. The reference manual does not specify how these tasks are created, so they may be created sequentially, in parallel, or some combination of the two. As is the case with one_task, none of these tasks may start their execution at this time.

Next, the access object, ref_task, is created. As with any access type, unless an initial value is provided by a call to the allocator, no object is created. Thus, no task is created when ref_task is elaborated.

The elaboration of the declarative region is complete once the body of worker is elaborated. This elaboration "has no other effect than to establish that the body can from then on be used for the execution of tasks designated by objects of the corresponding task type" (ARM 9.1(6)).

Upon conclusion of this elaboration, the sequence of statements is ready to be executed. However, prior to executing these statements, *all* tasks that were created in the declarative part are *activated* (ARM 9.3(2)). Activation of a task "consists of the elaboration of the declarative part, if any, of the task body" (ARM 9.3(10)). After a task has been activated, it can start executing its sequence of statements; it need not wait for any other task to complete its activation. Although *different* task objects are elaborated *sequentially*, activation of their corresponding tasks proceeds in *parallel* (ARM 9.3(1)).

Once all tasks have been successfully activated and begun their execution, the assignment of ref_task is encountered. Since ref_task is an access variable, the object it accesses is not created until the allocator, *new*, is called. When this transpires, a task object is dynamically created, and then activated. Once

```
declare
   task type worker is ...
   type ref_worker is access worker;
   one_task : worker;          -- Create a task.
   ten_task : array (1..10) of worker;     -- Create ten tasks.
   ref_task : ref_worker;
   task body worker is ...
      ⋮
begin     -- Activate one_task and ten_task(i), i ∈ 1..10
   ⋮
   ref_task := new ref_worker;     -- Create & activate a worker task.
end       -- Wait for all tasks to terminate.
```

Figure 4.3: Task Creation and Activation Example

part and begin its execution.

To illustrate these two notions, consider the block statement of Figure 4.3. During execution, the declarative part of this block is elaborated sequentially (ARM 3.9(3)); followed by the execution of the sequence of statements (ARM 5.6(4)).

After elaborating the task type, worker, and the access type, ref_worker, the task object, one_task is encountered. Elaborating a task object corresponds to creating the task, i.e. allocating its storage, making it known to the system, etc., but *not* starting its execution. The array of tasks, ten_tasks is treated similarly. Elaborating this array is just like elaborating any other array object declaration, and proceeds as follows (ARM 3.2.1(4-8)):

1. " ...the constrained array is first elaborated."

2. "...any implicit initial values for the object ...are evaluated."

3. "The object is created."

```
declare
   task type iteration_task is
      entry get_id(id : in integer);
   end iteration_task;

   par_loop : array (1..N) of iteration_task;

   task body iteration_task is
      my_id : integer;
   begin
      accept get_id(id : in integer) do
         my_id := id; -- Acquire an id from parent.
      end get_id;
      S1
      :
      Sn
   end iteration_task;

begin          -- All "iterations" begin to execute.
   for i in 1..N loop
      par_loop(i).get_id(i); -- Identities are distributed sequentially.
   end loop;
end;           -- Wait for all "iteration" tasks to terminate.
```

Figure 4.2: An Ada "Parallel Loop"

a coarse grained construct, the Ada task. While the introduction of a loop-level parallel construct, i.e. a parallel loop, would solve this problem, our goal is *not* to modify Ada, but to use the current language to express this form of parallelism efficiently. Before suggesting how this can be achieved, let us review the process of task initiation in Ada.

4.2 Task Initiation

This section describes the manner in which tasks are initiated. Tasks can be initiated statically or dynamically. Task initiation can be broken into two phases: *task creation* and *task activation*. The former corresponds to when the task object is created, while the latter is when the task elaborates its declarative

requires that each task acquires a unique identity. In the case of a parallel loop, this identity is used as the iteration value of the loop. As discussed in Chapter 2, a major shortcoming of Ada is the inability to distribute task identities in parallel, (the *task initialization* problem).

The serial bottleneck it implies increases the amount of overhead associated with the task initialization. Since this overhead is of more importance with loop-level parallelism than with a more coarse grain type, this shortcoming is most critical in the construction of a parallel loop.

In Chapter 3 we presented a solution, the gen_beacon idiom, to this problem using the fetch_and_add primitive. Although this idiom is used in subsequent sections of this chapter, we first describe the manner in which a parallel loop can be constructed *without* using the gen_beacon construct.

Figure 4.2 shows how one might achieve the effects of a parallel loop in Ada. A task type, iteration_task is declared; it corresponds to an iteration of the loop. This task contains one entry point, get_id, which is used to receive its identity or iteration value. This is followed by the declaration of an array of N tasks, corresponding to the N iterations of the loop.

After the task body is specified, iteration values of the "parallel loop" are distributed to each of the iteration_tasks by the **for** loop. Iteration i of this loop calls the get_id entry of the i^{th} task, supplying it with the iteration value i. This distribution of task identities is done sequentially, forming an undesirable bottleneck in our "parallel loop".

Even though we have suggested an idiom to remove this serial bottleneck, a source of inefficiency still exists; loop-level parallelism is being expressed using

```
Parallel Do i = 1, N
    S_1
    ⋮
    S_n
End
```

Figure 4.1: A Typical Parallel Loop

fine or instruction-level: At this level, parallelism is obtained by expanding the width of a machine instruction. Several operations are executed concurrently within one instruction on distinct processing units. Machines that support this form of parallelism are know as VLIW (Very Long Instruction Word) machines. As this form of parallelism is directly accepted by the machine, no overhead is required to initiate a parallel thread. The difficulty lies in finding enough operations that can be performed in parallel by one instruction [Ell86,EN90].

Parallelism in Ada is of the coarse grain variety. The Ada *task* is a "procedure-like" object that executes in parallel with other active tasks in the program. As described in Chapter 2, the *rendezvous* mechanism provides a manner in which synchronization and communication can be performed between two otherwise asynchronous tasks. Although the Ada tasking model is well-suited for those applications that utilize coarse grain parallelism, many scientific applications require the lower overhead associated with loop-level parallelism.

Figure 4.1 illustrates a typical parallel loop. The programmer specifies that all N iterations can be executed in parallel. Although no "parallel loop" construct exists in Ada, one can use tasks to specify a semantically equivalent entity. Constructing a parallel loop, like many other problems in parallel processing,

of the parallel construct that is used. Although granularity has been defined as "the basic size of a process chosen for parallelism" [Kri89], synchronization and communication costs must also be considered.

We define the *overhead* associated with a parallel construct as the amount of additional computation that is required to initiate and terminate the parallel threads that are associated with the construct. Overhead plays a significant role in determining the granularity of a construct. From it we can determine the minimum amount of computation that each thread must execute to ensure that the expected execution time is reduced, compared to sequential execution.

Granularity can be divided into three levels:[1]

coarse or procedure-level: A thread at this level of granularity can have up to the functional equivalent of an operating system process. It can contain calls to subprograms, access global variables, and perform synchronization and communication with other executing threads. However, due to this robust functionality a significant amount of overhead is required to manage these *heavy-weight* threads. Thus, this level of parallelism is only utilized when each thread requires this full functionality, and is going to execute a significant amount of computation to offset the corresponding overhead.

medium or loop-level: This form of parallelism severely limits the amount of communication and synchronization that a thread is allowed. Due to this limitation, this level of parallelism typically requires less overhead. Thus, a sequence of statements, or a loop iteration, often supplies enough computation to offset this overhead.

[1] Although most people agree that three levels of parallelism exist, there is no consensus as to what these three levels are. See [Kri89] for an alterative definition.

Chapter 4

The Gen_Minitask Package

A significant shortcoming of the Ada tasking model is the inability to specify *loop-level* parallelism in an *efficient* manner. This shortcoming, first discussed in Chapter 2, impedes the use of Ada as a language for expressing a large class of parallel algorithms, namely, those algorithms that utilize loop-level parallelism.

In this chapter, we introduce the gen_minitask package to overcome this shortcoming. This package is similar to the gen_beacon package, described in Chapter 3, in that both of these packages supply efficient solutions to well known Ada shortcomings, *without* modifying the language. In order to understand the need for this package, we discuss various types of parallelism before illustrating why Ada is lacking in the loop-level variety.

4.1 Types of Parallelism

An important characteristic of a parallel algorithm is the manner in which the parallelism is expressed. Of particular significance is the *granularity* or *grain size*

```
procedure insert(data : item) is
    my_insert, dummy : integer;
    function num_upper_tir is new
            test_increment_reset(num_upper.read, num_upper.faa);
begin
    if num_upper_test_increment_reset(1, queue_size) then
        my_insert := next_insert.faa(1) mod queue_size;
        -- Wait turn at my_insert
        Q(my_insert) := data;
        dummy := num_lower.faa(1);
    else
        raise OVERFLOW;
    end if;
end insert;

function delete return item is
    my_delete, dummy : integer;
    function num_lower_tdr is new
            test_decrement_reset(num_lower.read, num_lower.faa);
begin
    if num_lower_tdr(1) then
        my_delete := next_delete.faa(1) mod Queue_Size;
        -- Wait turn at my_delete
        data := Q(my_delete);
        dummy := num_upper.faa(-1);
    else
        raise UNDERFLOW;
    end if;
end delete;
```

Figure 3.14: The revised insert and delete Routines

```
generic
   with function read_beacon return integer;
   with function faa_beacon return integer;
procedure test_increment_reset(delta, bound : in integer) is
   dummy : integer;
begin
   if read_beacon + delta <= bound then
      if faa_beacon(delta) <= bound then
         return true;
      else
         dummy := faa_beacon(-delta);
         return false;
      end if;
   else
      return false;
   end if;
end test_increment_reset;

generic
   with function read_beacon return integer;
   with function faa_beacon return integer;
procedure test_decrement_reset(delta : in integer) is
   dummy : integer;
begin
   if read_beacon - delta >= 0 then
      if faa_beacon(-delta) >= 0 then
         return true;
      else
         dummy := faa_beacon(delta);
         return false;
      end if;
   else
      return false;
   end if;
end test_decrement_reset;
```

Figure 3.13: Generic Version of test_increment_reset and test_decrement_reset

variable of interest. This provides the test_increment_reset procedure a means to access the shared variable without specifying the variable itself. Thus, a subprogram can instantiate this generic function, and later call it, using the read and fetch_and_add routines that are made available by the gen_beacon package. The test_decrement_reset routine is treated in a similar manner. Figure 3.14 gives the revised insert and delete routines using the generic functions of Figure 3.13.

an attempt to claim this item. Assuming this is successful, true is returned to delete. This allows delete to remove this item and decrement the num_upper counter.

Both inserts and deletes can be performed in parallel. Since shared variables are accessing using the fetch_and_add construct, potential bottlenecks are avoided. Thus, the insert and delete routines provide a mechanism to support highly parallel queues.

3.4.1 An Apparent Shortcoming of Gen_Beacon

Consider the test_increment_reset routine shown in Figure 3.11. As previously mentioned, the original test_increment_reset routine has three parameters as input: a shared variable, a delta value, and an upper bound. However, in Figure 3.11 we *assume* the first parameter is always num_upper. While this is the case with this particular algorithm, one would hope to write the test_increment_reset procedure in a more general manner.

Intuitively, one would expect to pass the shared variable by reference to the test_increment_reset procedure. However, by using an instantiation of a package to represent a shared variable, we lose the ability to pass references of this variable; packages are not first class objects. Thus, it appears that we are unable to specify test_increment_reset in a general way.

This, however, is not the case. As Figure 3.13 illustrates, test_increment_reset can be specified as a generic procedure. Instead of passing the shared variable to this procedure, we provide the means to access this variable by passing two functions. The functions correspond to a read and a fetch_and_add on the shared

```
procedure test_decrement_reset(delta : in integer) is
    dummy : integer;
begin
    if num_lower.read - delta >= 0 then -- anything to delete?
        if num_lower.faa(-delta) >= 0 then -- try to claim it
            return true;
        else -- num_lower decremented between ifs
            dummy := num_lower.faa(delta); -- reset num_lower
            return false;
        end if;
    else
        return false;
    end if;
end test_decrement_reset;

function delete return item is
    my_delete, dummy : integer;
begin
    if test_decrement_rest(1) then -- see if an item exists
        my_delete := next_delete.faa(1) mod queue_size;
        -- Wait turn at my_delete
        data := Q(my_delete);
        dummy := num_upper.faa(-1); -- delete is complete
    else        -- no items to delete
        raise UNDERFLOW;
    end if;
end delete;
```

Figure 3.12: The delete and test_decrement_reset Routines

takes three parameters as input: a shared variable, a delta value, and an upper bound. For the call located in insert, these parameters correspond to the num_upper counter, 1, and queue_size, respectively. If the upper bound reports that no space exists, then test_increment_reset returns the value false to insert, which in turn raises the OVERFLOW exception.

If test_increment_reset returns true, we are not only guaranteed that space exists somewhere in q to insert the new item, but also that this space has been reserved. To acquire this space, a fetch_and_add is performed on the insertion index, next_insert, yielding a unique value which is stored in the local variable, my_insert. This value modulo queue_size is the index of q where the item is stored.

It is possible that a delete operation has not finished removing the previous element stored at this index. Therefore, we must wait until we are assured that this element is free to use.[6] Once q(my_insert) is no longer occupied, data is inserted and num_lower is incremented atomically via the fetch_and_add construct; this signifies that the element has been inserted.

The delete procedure, shown in Figure 3.12, is constructed in a similar manner. It also utilizes a function to determine, *and reserve*, an item to be removed from the queue. This routine, test_decrement_reset, checks the lower bound on the number of items in the queue to determine if one exists to be deleted. If no item exists, false is returned to the delete function and the UNDERFLOW exception is raised. If, however, an item does exist, num_lower is decremented in

[6] This can be done by using two semaphores, insert and delete, for each element of the array. Although these semaphores can be represented in Ada, we omit their specifications in this example. Before an insert operation is performed, a P operation is performed on the insert semaphore. The corresponding V operation is performed after the item is deleted. Similar actions are taken for the delete semaphore. For more details, see [GGK+83].

```
procedure test_increment_reset(delta, bound : in integer) is
   dummy : integer; -- used to receive unwanted faa values
begin
   if num_upper.read + delta <= bound then -- Any room?
      if num_upper.faa(delta) <= bound then -- Try to claim it.
         return true;
      else -- num_upper incremented between two if s
         dummy := num_upper.faa(-delta); -- reset num_upper
         return false;
      end if;
   else
      return false;
   end if;
end test_increment_reset;

procedure insert(data : item) is
   my_insert, dummy : integer;
begin
   if test_increment_reset(1, queue_size) then -- try to find room
      my_insert := next_insert.faa(1) mod queue_size;
      -- Wait turn at my_insert
      Q(my_insert) := data;
      dummy := num_lower.faa(1); -- insert is complete
   else       -- no room available
      raise OVERFLOW;
   end if;
end insert;
```

Figure 3.11: The insert and test_increment_reset Routines

respectively. These two variables never differ by more than the number of active insertions and deletions [GGK+83]. Both counters are initialized to zero, indicating an empty queue.

Figure 3.11 specifies the insert and test_increment_reset routines. Before considering the insert procedure, we turn our attention to the test_increment_reset function. The goal of test_increment_reset is to guarantee and reserve space for an insertion into the queue. As originally specified in [GGK+83], this routine

Upon instantiating this package, two subprograms are visible: insert and delete. The former is passed a parameter of type item to be inserted at the back of the queue. The latter returns an item after removing it from the front of the queue. If an attempt is made to insert into a full queue, the exception, OVERFLOW is raised. Similarly, an attempt to delete an item from an empty queue will raise the exception, UNDERFLOW. The insert and delete routines are specified so that these operations are performed in a "highly parallel" manner.

In the package body, we represent a queue of length, queue_size, by a circular array. Since concurrent inserts and deletes are desired, q is used in a shared manner. Although Ada does not allow composite objects to be declared **pragma** shared (ARM 9.11(10)), others feel this restriction is too severe [Dew90,Shu87,Hum88]. Even though atomic updates to the entire composite object are not possible on most machines, atomic updates to a component of the object are possible, and therefore, in the spirit of ARM 9.11(11). In the work of Hummel [Hum88], a new **pragma**, volatile, is introduced as a means of marking those composite objects that are used in a shared fashion. In this work, we subscribe to this usage.

In addition to the shared array, q, the algorithm utilizes four integer shared variables. Since fetch_and_add operations are performed on these variables, the gen_beacon construct is used to realize them. The shared variables represented by the packages, next_insert and next_delete, correspond to the front and rear of the queue, respectively. The initial value passed to these packages is zero, signifying an empty queue.

The algorithm also makes use of two counters, num_lower and num_upper, corresponding to lower and upper bounds for the number of items in the queue,

```
generic
   queue_size : integer;
   type item is private;
package parallel_queue is
   OVERFLOW, UNDERFLOW : exception;
   procedure insert(data : item);
   function delete return item;
end parallel_queue;

package body parallel_queue is
   q : array(0..queue_size - 1) of item;
   pragma VOLATILE(q);
   package next_insert is new gen_beacon(0);
   package next_delete is new gen_beacon(0);
   package num_upper is new gen_beacon(0);
   package num_lower is new gen_beacon(0);

   procedure test_increment_reset(delta, bound : in integer) is ...
   procedure insert(data : item) is ...

   procedure test_decrement_reset(delta : in integer) is ...
   function delete return item is ...
end parallel_queue;
```

Figure 3.10: The parallel_queue Generic Package

is used to solve the task initialization problem. In this section we illustrate by example how a "fetch_and_add algorithm" would appear in Ada, when the gen_beacon construct is employed.

The algorithm we choose, given by Gottlieb *et al.* in [GGK+83], provides a means of management of "highly parallel queues". Since the algorithm was originally described as a collection of Pascal-like routines, it seems intuitive to collect these routines and their data structures into an Ada package. This package, shown in Figure 3.10, is constructed in a generic fashion with two parameters: queue_size and item, corresponding to the maximum size of the parallel queue and the type of item to be stored in the queue.

Translated subprogram calls to read, write, and faa are given in Table 3.1. Note that although we have eliminated the querying of status variables, we do need to flush any *Ada shared variables*[5] associated with the calling task that have been stored locally. This observation is also true with all previous variants of the beacon task and was first attributed to Robert Dewar in [Hum88]. The need for this cache flush is because a fetch_and_add-like rendezvous, as with any other rendezvous, is a synchronization point for Ada shared variables. This implies that any local copies of Ada shared variables of the calling task must be stored into global memory before and after a rendezvous occurs (ARM 9.11(7)).

The only variable used in a shared fashion that is associated with the gen_beacon package is v. While it is possible to keep local copies of this variable for each task that has access to it, the usage of this variable seems to suggest that storing it in the shared global memory would be a better choice. Therefore, there are no Ada shared variables associated with the gen_beacon package. Although the gen_beacon package provides a means to obtain an integer shared variable, there is no guarantee that the user will not employ Ada shared variables elsewhere. Thus, each read, write, and fetch_and_add is preceded by a flush of any of the user's Ada shared variables.

3.4 An Example

By defining the gen_beacon construct, we have given the Ada programmer an efficient means to utilize the fetch_and_add primitive. In Chapter 4 this construct

[5] An Ada shared variable is a variable that is not declared by a **pragma** as shared, but nevertheless is used in a shared manner.

Original Code	Transformed Code
x := my_beacon.read;	flush current task's Ada shared variables x := my_beacon.value;
my_beacon.write(x);	flush current task's Ada shared variables my_beacon.value := x;
x := my_beacon.faa(inc);	flush current task's Ada shared variables x := *fetch_and_add*(my_beacon.value,inc);

Table 3.1: Code Fragments for read, write, and faa Subprograms

1. call the read_var **entry**,

2. access the monitor variable,

3. return its value to the read function,

4. return this value to its caller

is replaced by a single hardware read instruction. The write procedure is transformed in a similar fashion.

Consider the faa procedure. As is the case with the read and write subprograms, the procedure and **entry** calls can be eliminated. On an architecture that supports the fetch_and_add hardware primitive, the faa procedure is transformed exactly as is the case with read and write. However, even on machines that do *not* possess this primitive, this type of transformation can still be performed. In particular, the procedure and **entry** calls can both be removed. However, since the read and increment operations that form the fetch_and_add, cannot be performed atomically, semaphores must be used to ensure that concurrent fetch_and_adds do not interfere with each other.

the generic have been evaluated, the elaboration of the instantiated package can begin.

We now consider the body of the gen_beacon package. All entities declared here are *not* visible to the instantiator. Elaborating a package body corresponds to elaborating its declarative part (ARM 7.3(2)). This corresponds to elaborating the task type definition, creating the task object, and then elaborating the subprograms and beacon task body. Since the task type and bodies of the all of these entities are known, no action is required to elaborate them (ARM 6.3(5)). Since the beacon task is transformed into a passive object, creating it corresponds to allocating space for the beacon_struct record.

After elaborating the declarative part of the package body, the associated sequence of statements is executed (ARM 7.3(2)). The gen_beacon package does not contain a sequence of statements. Thus, instantiating gen_beacon corresponds to allocating space for the integer variable which holds the beacon variable.

The read, write, and faa Subprograms

Since the beacon task is no longer visible to its user, access to the integer variable it monitors must be provided by some other means. This is the role of the three visible subprograms: read, write, and faa. The semantics of these routines are straightforward; each routine calls the appropriate **entry** call of the hidden beacon task.

As was the case when transforming the original beacon task's **entry** calls, these subprogram calls, and their corresponding **entry** calls, are also transformed. For example, a call to the function read which formerly did the following:

```
task body beacon is
   v: integer := initial_value;
begin
   loop
      select
         accept read_var(val out integer) do
            val := v;
         end read_var;
      or
         accept write_var(val: in integer) do
            v := val;
         end write_var;
      or
         accept F_and_A(val: out integer; inc: in integer) do
            val := v;
            v := v + inc;
         end F_and_A;
      or
         terminate;
      end select;
   end loop;
end beacon;
```

Figure 3.9: Beacon Task Body for the Generic Beacon Package

this initial value is supplied before the beacon task is activated, we not only ensure that an initial value is provided, but also eliminate the need to coordinate initialization attempts by competing tasks. This fact, combined with the "packaging" of the beacon task, reduces the representative beacon data structure to a single integer variable: an integer that holds the value of the fetch_and_add variable. Thus, the data structure to manage a beacon task becomes:

```
type gen_beacon_struct is record
    value : integer;
end record;
```

An instantiation of gen_beacon provides three visible subprograms: the functions read and faa, and the procedure write. Both read and faa return integer values, while write accepts an in parameter of the type integer. Each of the visible subprograms of gen_beacon calls the appropriate entry of the hidden beacon_task (see Figure 3.8).

3.3.1 Implementation

In this section we describe the Ada semantics of gen_beacon and show how under these semantics an efficient implementation can be realized. Below is an example of instantiation of the gen_beacon package.

```
my_beacon is new gen_beacon(1);
```

The instantiation of a generic package has the same effect as specifying the package in the place where the generic is instantiated, substituting all references to generic parameters with the actuals (ARM 12.3). Once the parameters of

```
generic
   initial_value : integer;
package gen_beacon is
   function read return integer;
   procedure write(val: in integer);
   function faa(inc: in integer) return integer;
end gen_beacon;
package body gen_beacon is
   task type beacon is
      entry read_var(val out integer);
      entry write_var(val: in integer);
      entry F_and_A(val: out integer; inc: in integer);
   end beacon;
   beacon_task : beacon;
   function read return integer is
      tmp : integer;
   begin
      beacon_task.read_var(tmp);
      return tmp;
   end read;
   procedure write(val: in integer) is
   begin
      beacon_task.write_var(val);
   end write;
   function faa(inc: in integer) return integer is
      tmp : integer;
   begin
      beacon_task.F_and_A(tmp, inc);
      return tmp;
   end faa;
   task body beacon is ...
      -- See Figure 3.9
   end beacon;
end gen_beacon;
```

Figure 3.8: The Generic Beacon Package

is used to represent a transformed beacon task is reduced to the following:

```
type beacon_struct is record
    value : integer;
    initialized : boolean := false;
    init_sync : integer := 0;
end record;
```

The code fragments are likewise reduced by eliminating all checks of the eliminated completed field.

3.3 Our Construct: The Generic Beacon Package

Although we have reduced the run time overhead to calls of read, write and F_and_A by encorporating the beacon task in a library package, Ada semantics still requires us to inspect the initialized field before access to the beacon variable can occur. Furthermore, two variables are still required for the management of initialization.

In an attempt to eliminate these variables and the undesirable overhead that is required to inspect them, while still providing means to ensure proper initialization, we introduce the generic beacon package, gen_beacon (see Figures 3.8 and 3.9). This generic package captures all of the advantages of the beacon task and package, while also solving the problem of beacon initialization.

As seen in Figure 3.8, the generic beacon package overcomes the problem of initialization by having an initial value provided as the generic parameter. Since

- The beacon task contained in the beacon package no longer contains a local variable. Thus, all entry calls to this task must supply an access variable that is to be modified or read. This gives the appearance that operations on two *different* fetch_and_add variables will occur in a sequential manner. Although this is true of beacon package at the Ada level, a compiler can removal this potential bottleneck.

  Consider two tasks, both of which are to perform a fetch_and_add operation on distinct variables. Since both tasks subsequently call the F_and_A rendezvous, Ada semantics specify that these rendezvous must occur sequentially. However, since the two tasks operate asynchronously, the order in which these calls are made is not known.

  Now consider when these entry calls are translated directly to the fetch_and_add hardware primitive. Instead of engaging in a rendezvous, each task now performs a fetch_and_add operation. Since the programmer is unable to detect the order in which these operations occur, they can be executed concurrently.

  Thus, although an undesirable bottleneck is present at the Ada level, it is removed when these entry calls are transformed. The underlying hardware decides in what order the operations occur.

The second difference is transparent to the programmer and therefore not an issue. However, the first difference is one the programmer must be aware of. A solution to this problem, given by Mitsolides [Mit88], would be to provide the user of the beacon package with another procedure to allow initialization (and creation) of new fetch_and_add variables. Using this approach, the structure that

```
task type beacon is
   entry read_var(var: in shared_integer; val out integer);
   entry write_var(var: out shared_integer; val: in integer);
      entry F_and_A(var: in out shared_integer;
                    inc: in integer; val: out integer);
end beacon;

task body beacon is
   v: integer;
begin
   loop
      select
         accept read_var(var: in shared_integer; val out integer) do
            val := var.all;
         end read_var;
      or
         accept write_var(var: out shared_integer; val: in integer) do
            var.all := val;
         end write_var;
      or
         accept F_and_A(var: in out shared_integer;
                        inc: in integer; val: out integer) do
            val := var.all;
            var.all := var.all + inc;
         end F_and_A;
      or
         terminate;
      end select;
   end loop;
end beacon;
```

Figure 3.7: Beacon Task Type as in [Hum88]

of a task dependence relation with its calling tasks, and also can not be aborted by its callers.

In an approach suggested by Hummel [Hum88], the beacon task is hidden inside the private part of a library package body (see Figure 3.6). Access to its entries is only possible by procedure calls visible to the subprogram using this package.

This approach offers the following advantages:

- Since the task is hidden in the private part of the library package, no subprogram that uses it can name it, and therefore abort it directly.[4]
- Since the task is declared in a library package, the subprogram that uses this package does not need to wait for this task to complete before it can terminate (ARM 9.4(13)).

A direct result from these two observations is that we no longer need to store information about the beacon task's status (completed or not); it will not complete until after all tasks that can potentially call it have terminated. Furthermore, since it is contained in a library package, tasks that can call it need not wait for its completion in order to terminate (ARM 9.4(13)).

However, the proposed solution, (Figure 3.7) differs from the original beacon task (Figure 3.1) in a few ways:

- The beacon package version no longers ensures that the local variable associated with the beacon task has been properly initialized. Thus, calls to read and F_and_A may return unknown results.

[4]The task can still be aborted if its master task is aborted. However, in this case all potential callers will also be aborted.

```
package beacon_pack is
   type shared_integer is access integer;
   function read(var: in shared_integer) return integer);
   procedure write(var: in out shared_integer; val: in integer;
   function faa(var: in out shared_integer; inc: in integer) return integer;
end beacon_pack;

package body beacon_pack is
   type shared_integer is access integer;
   task type beacon is
      entry read_var(var: in shared_integer; val out integer);
      entry write_var(var: out shared_integer; val: in integer);
      entry F_and_A(var: in out shared_integer;
                inc: in integer; val: out integer);
   end beacon;

   beacon_task : beacon;

   function read(var: in shared_integer) return integer is
      tmp : integer;
   begin
      beacon_task.read_var(var, tmp);
      return tmp;
   end read;

   procedure write(var: in out shared_integer; val: in integer) is
   begin
      beacon_task.write_var(var, val);
   end write;

   function faa(var: in out shared_integer; inc: in integer) return integer is
      tmp : integer;
   begin
      beacon_task.F_and_A(var, inc, tmp);
      return tmp;
   end faa;

end beacon_pack;
```

Figure 3.6: Body of Beacon Package as in [Hum88]

When a beacon task is aborted, the completed field is set to true. As we have seen by the code fragments, any task that calls an entry of this aborted task, will find the completed field set to true. The calling task will then take the appropriate action depending on the type of entry call.

When a calling task is aborted, the transformed beacon task is not affected; the beacon data structure still exists for other tasks to access. These semantics are consistent with those of the untransformed beacon task.

Since a task that is aborted need not terminate until it reaches a synchronization point (ARM 9.10(6)), the body of the code fragments may be executed in its entirety. However, since completing the code fragment represents the end of the entry call, and hence a synchronization point, an aborted task must cease execution at this time.

3.2 Packaging a Beacon Task

Recall that one motivation behind creating the beacon task is to make the fetch_and_add primitive available at the Ada level. To this extent, the beacon task achieves this goal. However, by introducing an active object (a task) to represent a passive one (a shared variable), the semantics of Ada have necessitated the inclusion of additional status variables. These status variables are undesirable; we desire a direct translation of uses of the fetch_and_add idiom to actual fetch_and_add operations.

Since some of these status checks are required by the semantics of task completion and abort instructions, one may try to specify a beacon task in a manner such that these issues are not of concern; we want a beacon task that is not part

```
DT := CLOCK + D;
while not T.initialized or T.completed loop
   if T.completed then
      raise TASKING_ERROR;
   end if;
   exit when CLOCK > DT;
end loop;
if CLOCK < DT then
   case type of operation is
      when read => var := T.value;
      when write => T.value := val;
      when F_and_A => val := fetch_and_add(T.value, inc);
   end case;
   -- Execute any other statements associated
   -- with the entry call part of the select statement.
else
   -- Execute the statements associated with
   -- the delay part of the select statement.
end if;
```

Figure 3.5: Code Fragment for a Timed Entry Call with delay D

part of the select statement are executed.

3.1.2 Handling Aborts

Since the beacon task is originally an Ada task, it may be aborted by any task that can name it. Although the abort statement "should be used only in extremely severe situations" (ARM 9.10(10)), care must be taken to ensure that proper Ada semantics are upheld when beacon tasks are abstracted away. Therefore, we must ensure that any transformation of the beacon task properly handles any attempts to abort it. Additionally, we must show that if any task that can call one of the beacon task's entries is aborted, the integrity of the transformed beacon task is maintained.

```
if T.completed then
    raise TASKING_ERROR;
end if;
if T.initialized then
case type of operation is
    when read => var := T.value;
    when write => T.value := val;
    when F_and_A => val := fetch_and_add(T.value, inc);
end case;
    -- Execute any other statements associated
    -- with the conditional part of the select statement.
else
    -- Execute the statements associated with
    -- the else part of the select statement.
end if;
```

Figure 3.4: Code Fragment for a Conditional Entry Call

the appropriate operation is performed followed by any other statements present in the select option of this entry call. If T is not initialized, then the rendezvous does not occur; the statement(s) contained in the else part of the select statement are executed.

The code fragment for *timed* entry calls utilizes the methods used in both the simple entry call and the conditional entry call. It is given in Figure 3.5. A timed entry call specifies a delay D which is the maximum amount of time the calling task is willing to wait for the receiving task to reach its accept statment (ARM 9.7.3(1)). Thus, a timed entry call does not necessarily require that the rendezvous occurs immediately. Once again, a loop is employed to check the initialized and completed status variables.

Once initialization is completed, a check must be performed to ensure that D time has not passed. If D time has passed since execution of the code fragment begun, the rendezvous does not occur; the statements associated with the delay

```
while not T.initialized or T.completed loop
   if T.completed then
      raise TASKING_ERROR;
   end if;
end loop;
case type of operation is
   when read => var := T.value;
   when write => T.value := val;
   when F_and_A => val := fetch_and_add(T.value, inc);
end case;
```

Figure 3.3: Code Fragment for an Unconditional Entry Call

calling task access to the beacon variable, check must be performed to ensure that the beacon task has been initialized and has not been aborted. We describe below the manner in which this is done.

The code fragment for an *unconditional* entry call is given in Figure 3.3. If T is yet to be initialized and is not marked completed, the calling task loops until one of these two conditions change. If T becomes completed, i.e. another task aborts it, TASKING_ERROR is raised in the calling task. This corresponds to the beacon task being aborted before it reaches one of the accept statements contained in the select statement. As is the case with the init code fragment, raising TASKING_ERROR precludes execution of the rest of the code fragment. However, if T becomes initialized, before it is completed, the value field is accessed in the appropriate fashion (read, write, or fetch_and_add).

If the entry call is of the *conditional* variety, then the code fragment described in Figure 3.4 is executed. Since a conditional entry call attempts to perform a rendezvous immediately, a loop is no longer needed; we first check if the task is completed, raising TASKING_ERROR if it has. If T is not completed, a check is made to see if the variable has been initialized. If T has been properly initialized,

receives this value is designated as the "winner" and is allowed to set the value field to the value passed. The initialized flag is then set to true. By executing a fetch_and_add on the init_sync field, we ensure that only one task is allowed to initialize the value field, avoiding a race condition.

If a non-zero value is returned from the fetch_and_add operation, then another task has, or is about to, initialize the value field. When this occurs the action performed is dependent on the type of init entry call that was executed. If the call was a *timed* entry call, the calling task would normally wait the specified amount of time before proceeding with the statement immediately following the call. To uphold these semantics, the calling task delays the appropriate amount of time and then exits the code fragment for init.

When a *conditional* entry call to init is unsuccessful – another task has already been selected as the initializer – the calling task executes the else part associated with the entry call. Thus, when a non-zero value is returned from the fetch_and_add of init_sync, the else part of the entry call is executed.

If the entry call is *unconditional*, then Ada semantics specify that the calling task should wait until this call can be satisfied. However, the beacon task is constructed so that it only accepts one init entry call; a subsequent rendezvous can never occur. Thus, if a task that executes an unconditional entry call receives a non-zero value, it is blocked until it is aborted. This program most likely contains a logic error.

The Read, Write, and F_and_A Entry Calls

The code fragments for the read, write, and F_and_A entry calls are similar. We present each fragment according to the type of entry call. Before allowing a

```
if T.completed then
    raise TASKING_ERROR;
end if;
num := fetch_and_add(T.init_sync, 1);
if num = 0 then
    T.value := e;
    T.initialized := true;
else        -- task T has already been initialized.
    case type of entry call is
        when timed => wait for the stated time
        when unconditional => block until calling task is aborted
        when conditional =>
                    execute statements associated with else part
    end case;
end if;
```

Figure 3.2: Code Fragment for T.init(e)

The Init Entry Call

Consider the code fragment executed when the init entry is called (Figure 3.2). First, a check is performed to ensure that T is not marked completed; it has not completed its execution or been aborted. Ada Semantics specify that both "the call of an entry of an abnormal task" (ARM 9.10(7)) and "an attempt to call an entry of a task that has completed its execution" (ARM 9.5(16)) result in TASKING_ERROR being raised at the place of the call. Thus if T.completed is true, TASKING_ERROR is raised in the calling task. This precludes the execution of the rest of the code fragment.

If T.completed is still false, an attempt is made to mimic the init rendezvous. To do this we must ensure that only one of several potential callers is allowed to initialize v. This is done by having each of the potential initializers execute a fetch_and_add operation with increment of one on the init_sync field. Only one of these tasks will have the original value of zero returned. The calling task that

to Ada tasking semantics, each of these operations must be preceded by a check of some status variables associated with the transformed beacon task. These variables are used to coordinate beacon initialization, as well as ensuring that an abnormal beacon is not accessed. They are contained in a data structure along with the integer value associated with the beacon task. The data structure that represents the transformed beacon task is:

```
type beacon_struct is record
    value : integer;
    initialized : boolean := false;
    completed : boolean := false;
    init_sync : integer := 0;
end record;
```

The value field corresponds to the local variable, v, in the original beacon task. Two boolean fields, initialized and completed hold information on the callability of the beacon task. A beacon task is initialized once some other task provides an initial value via the init **entry** call. A beacon task becomes completed when it is aborted or when the task that declares it, its *master* task, and all of its descendents terminate.[3]. If any of these tasks try to access the beacon task when it is completed, TASKING_ERROR is raised in the calling task. The integer, init_sync, is used to coordinate multiple initialization attempts.

We now discuss the code fragments that a calling task executes in place of the entry calls to a beacon task T.

[3]We discuss task dependence in more detail in Chapter 4

Of particular importance is the semantics of the F_and_A entry. As mentioned in Chapter 2.1, the semantics of the fetch_and_add primitive specify that the variable is incremented by the value supplied and the original value returned. These two operations must appear to occur atomically. Since access to the local variable is monitored by the beacon task, the F_and_A entry ensures that these semantics are upheld.

3.1.1 Transforming the Beacon Task

Due to the manner in which the beacon task is constructed, it can be transformed into a passive data structure. On a machine that supports the fetch_and_add primitive, reads, writes, and F_and_A's are performed atomically. Hence, we would like to replace a call to any of these entries with the actual read, write or fetch_and_add operation.[2] As described in [SS85], the "beacon thus disappears as a separate task entity, and leaves behind a simple data structure and code fragments to access it." This type of transformation is an example of the kind proposed by Hilfinger [Hil82a,Hil82b,Hil90].

Note that even on a machine that does not support the fetch_and_add primitive, a transformation of this type can still be performed. However, synchronization must be provided to ensure that the two instructions that comprise the F_and_A rendezvous are both executed before any other task is permitted access to v. A simple semaphore is sufficient to provide this synchronization.

Although it would appear that each call to the beacon task can be directly translated into the corresponding hardware operation, this is not the case. Due

[2] Initialization is viewed as a form of a write operation.

```
task type beacon is
   entry init(e: in integer);
   entry read(l: out integer);
   entry write(e: in integer);
   entry F_and_A(l: out integer; e: in integer);
end beacon;

task body beacon is
   v: integer;
begin
   accept init(e: in integer) do
      v := e;
   end init;
   loop
      select
         accept read(l: out integer) do
            l := v;
         end read;
      or
         accept write(e: in integer) do
            v := e;
         end write;
      or
         accept F_and_A(l: out integer; e: in integer) do
            l := v;
            v := v + e;
         end F_and_A;
      or
         terminate;
      end select;
   end loop;
end beacon;
```

Figure 3.1: Definition of the beacon Task Type as in [SS85]

3.1 The Beacon Task Type

In this section we present the beacon task type as originally formulated [SS85]. An instance of a beacon task provides the user with an integer variable which may be used in a shared fashion. In addition to reading and writing this variable, an operation that satisfies the fetch_and_add semantics is permitted

As shown in Figure 3.1, the beacon task contains an integer variable, v. This variable corresponds to the shared variable to be used. Access to it is provided through four entry calls. In addition to the three operations mentioned, an initialization entry call, init, is also provided. In order to ensure that v is properly initialized, another task must call the init **entry**, supplying an initial value. Once this rendezvous has occurred, the beacon task accepts any of the other three entry calls: read, write, and F_and_A.

These entry calls are enclosed in a select statement. The semantics of this statement specify that if more than one of the accept statements are *open*, i.e. a task has called their corresponding entry, then "one of them is selected arbitrarily" (ARM 9.7.1(6)). Since the select statement is enclosed by a loop, this process continues until no more active callers exist. These semantics are precisely the way our target machine model treats simultaneous reads, writes, and fetch_and_adds – as if they occurred in some unspecified serial order. Thus, the semantics of the beacon task match nicely with its desirable implementation.

By declaring v local to the beacon task, access to it is restricted; it is only possible through one of the four entry calls. This characteristic, when combined with the fact that the beacon task can only be executing one of these entries at a time, ensures that atomic access to this variable is maintained.

has been rewritten in Ada to utilize the fetch_and_add construct we suggest.

In specifying a fetch_and_add construct at the Ada level, we impose the following three constraints:

1. The fetch_and_add construct must be easy to use syntactically and should not add unnecessary complexity to any program that utilizes it.

2. The semantics of the fetch_and_add construct must be consistent with those of the fetch_and_add primitive. This enables architectures that do not support the fetch_and_add primitive to execute this construct with the same results,[1] albeit not as efficiently.

3. To make best use of those architectures that do support the fetch_and_add primitive, a translation of the fetch_and_add construct to the hardware primitive must be available. This requires that a compiler targeted for a machine that supports the fetch_and_add primitive be able to recognize and exploit uses of the fetch_and_add construct. A direct translation from the fetch_and_add construct to the fetch_and_add primitive is desirable.

The original beacon task satisfies all three of these constraints. However, by allowing for a more direct translation from the fetch_and_add construct to the fetch_and_add primitive, the new gen_beacon generic package satisfies the 3^{rd} constraint in a more robust manner. In order to understand why this is the case, we turn our attention to the beacon task.

[1]Since the fetch_and_add primitive incorporates non-determinism into its semantics, same *results* may not be possible even on the same architecture. However, we expect the *semantics* of fetch_and_add construct to be consistent across architectures, regardless of whether the fetch_and_add primitive is available.

Chapter 3

Beacon Tasks and Fetch_And_Add

This chapter presents a solution to the *task initialization* problem as discussed in Chapter 2. This solution uses a generic package, gen_beacon, to make the fetch_and_add primitive available at the Ada level. By making this primitive available to the Ada programmer, we not only solve the task initialization problem, but also extend the possibilities of process synchronization in Ada.

Schonberg and Schonberg [SS85] first introduced this idea through the use of the beacon task type. In this work, we specify an enhanced version of the beacon task. Since the work of Schonberg and Schonberg forms the foundation of this work, we first describe the beacon task as given in [SS85]. We then show how encapsulating this task into a generic package helps to reduce run-time overhead and storage. This generic package is used in Chapter 4 where the problem of tasking overhead is addressed. We also present a parallel queue algorithm that utilizes the fetch_and_add primitive [GGK+83]. This algorithm

```
generic
   type index_type is (<>); -- any discrete subtype
   with procedure body_proc (i : in index_type);
procedure self_scheduling_loops (num_tasks : in positive;
                                 lb, ub : index_type);

procedure self_scheduling_loops (num_tasks : in positive;
                                 lb, ub : index_type) is
   task type worker;
   workers : array (1 .. num_tasks) of worker;
   task loop_control is
      entry get_loop_status (done : out boolean);
      entry get_index_index_value (i : out index_type);
   end;

   task body worker is
      i : index_type;
      done : boolean;
   begin
      loop
         loop_control.get_loop_status (done);
         exit when done;
         loop_control.get_index_value (i);
         body_proc (i);
      end loop;
   end;

   task body loop_control is
   begin
      for j in lb .. ub loop
         accept get_loop_status (done : out boolean) do
            done := false;
         end;
         accept get_index_value (i : out index_type) do
            i := j;
         end;
      for k in 1..num_tasks loop
         accept get_loop_status (done : out boolean) do
            done := true;
         end;
      end loop;
   end;
begin
   null;
end;
```

Figure 2.5: Dritz's Self_Scheduling_Loops

and is shown in Figure 2.5.

A typical use of this generic would require an instantiation as follows:

```
procedure self_scheduling_loop is new self_scheduling_loops (st, loop_body);
```

where st corresponds to the "common base type of the expressions for the bounds [of the loop], or any subtype thereof containing both of the bounds". Declaring st as integer will certainly suffice. The second parameter, loop_body, corresponds to a procedure that contains the statements in the body of the loop. After this generic has been instantiated, a parallel loop is obtained by calling the instantiated procedure in the following manner:

```
self_scheduling_loop(10, 1, 1000);
```

Although this solution is quite elegant, it differs from our work in the following ways:

1. No optimized implementation algorithm has been devised [Dri90a]. Therefore, worker tasks are implemented as a regular Ada task.

2. The *task initialization* problem has not been overcome; iterates are assigned sequentially. This implies that overhead associated with this paradigm is similar to that of the Ada parallel loop equivalent shown in Figure 2.4.

3. There is no mechanism for the parent thread to inspect the status of any of the child threads that execute the iterates.

In [Dri90b], Dritz specifies an enhanced version of this generic that reuses the iteration tasks in subsequent instantiations. This facility is desirable if the overhead associated with the initiation of worker tasks proves to be a bottleneck.

experiment was to construct an Ada compiler. Due to the flexibility that high-level programming in SETL provides, the compiler was rapidly produced. In 1983, the Ada/Ed system became the first validated Ada compiler, in addition to becoming the de facto standard operational definition of Ada [Hum88].

The next goal of the experiment was to increase system performance by translating it to low-level SETL and eventually C [KS84]. The C version was developed on a Vax and has been ported to a variety of machines including Sun, Alliant, and the PC.

In her PhD thesis, Hummel [Hum88] described a highly parallel version of the Ada/Ed run time system free of serialization points. It relies on the fetch_and_add primitive and supports Ada's tasking features in a highly parallel manner. Her work has been incorporated into the Ada/Ed-C system and has been implemented on the NYU Ultracomputer [Beh90]. Although Hummel's work avoids implementing Ada tasks as heavy weight operating system processes, these tasks are still not suitable to specify loop-level parallelism. Our work extends the work of Hummel by specifying two idioms and their implementations that allow efficient loop-level parallelism to be realized.

2.3.2 Dritz's Work

We conclude this chapter by describing a suggested solution developed independently by Dritz [Dri90b] to obtain loop-level parallelism in Ada. In this work, Dritz shows how several important tasking paradigms can be realized in Ada. He describes monitors, barriers, gates, and a self-scheduling for-loop. The latter paradigm is similar in spirit to the gen_minitask generic discussed in Chapter 4

called Ada 9X. Dewar [Dew90] discusses some of the issues involved with shared variables in relation to possible Ada 9X requirements. In Chapter 7 we discuss the current state of the revision process, focusing on the proposed revisions that directly relate to our work.

In addition to problems discussed by Dewar, Shulman [Shu87] and Hummel [Hum88] discuss other issues pertaining to shared variables. In particular, they specify an algorithm to detect unmarked synchronous shared variables. Although in most cases these variables can be found at compilation time, the use of Ada's separate compilation facility can hamper this detection, forcing its postponement until bind time.

Synchronous shared variables are relevant to our work because the cache semantics they imply affect the optimizations that we can perform. Chapters 3 and 5 address this issue further.

2.3 Related Work

This section describes related work to this thesis and is broken into two sections. The material presented in the first section forms the basis for our work. The last section describes a proposed solution to the problem of obtaining loop-level parallelism developed independently by Dritz [Dri90b].

2.3.1 The NYU Ada/Ed System

The basis for our work is the NYU Ada/Ed system, a translator/interpreter of the full Ada language. The NYUAda project began as an experiment in large-scale prototyping using the SETL language [SDDS86]. The goal of the

SHARED, while the synchronous variety cannot.[4] Thus, all variables not marked as pragma SHARED, but nevertheless used by more than one task, are synchronous shared variables.

Dewar [Dew90] notes:

> It is a source of continuing confusion that pragma SHARED does not correspond to the concept of shared variable, as defined in Steelman[5] or the RM, but rather to a particular subset of shared variables, namely the asynchronous ones. This means for example that the Steelman requirement that shared variables be marked is not satisfied by the provision of pragma SHARED, since synchronous shared variables do not have to be marked in Ada.

In addition to this deficiency, not allowing the programmer to designate composite data structures as pragma SHARED variables presents a problem. Most applications that employ loop-level parallelism typically have several threads working independently on parts of a shared data structure. Since different tasks access different parts of this composite object simultaneously, it is used in an asynchronous manner. However, Ada does not permit composite objects to be declared as pragma SHARED variables.

As suggested by Hummel [Hum88], an additional pragma can be used to overcome this deficiency. For example the DEC Ada compiler supports a pragma VOLATILE to designate composite objects that are used in a shared manner. This problem and many others are being addressed in the revision of the language,

[4] Asynchronous shared variables are sometimes referred to as pragma SHARED variables, while the synchronous variety are called *Ada shared* variables.

[5] Steelman is the requirements document for Ada-83.

These deficiencies and their solutions are further described in Chapters 3 and 4.

2.2.3 Shared Variables in Ada

In Ada a shared variable corresponds to a variable that is referenced by more than one task. There are two types of shared variables:

Synchronous: A synchronous shared variable is a variable referenced by more than one task, but for which local copies can be maintained between language-defined *synchronization points*; if one task writes to this variable, no other task can read or write to it until a synchronization point is reached.

Asynchronous: An asynchronous shared variable is a variable referenced by more than one task which does *not* satisfy the exclusive access constraint of synchronous shared variables between synchronization points. In addition, asynchronous shared variables are restricted to scalar and access types.

The motivation behind these definitions is that a synchronous shared variable may be cached locally to a processor, and flushed to shared memory at synchronization points of the task executing on the processor. On the other hand, asynchronous shared variables cannot be cached locally; they must reside in shared memory.

A major distinction between these two types of shared variables is that the asynchronous type can be tagged as such by the programmer using the pragma

requires that each task possess a unique identity. In the case of a parallel loop, this identity is the iteration value for the corresponding iterate. The only mechanism by which each task can initialize its state uniquely (with its iterate) is to provide each with an appropriate entry, and to have another task perform a rendezvous with each iterate task.[3] This solution is undesirable not only because of the rendezvous that must be performed, but more importantly because of the bottleneck that results. Additional descriptions of this shortcoming can be found in [Yem82,Cla87,Blu81].

2. Task initiation is a two stage process [Ada83]. Task objects are first created, and then, provided no exceptions are raised, they are activated. While this two stage process is beneficial for handling exceptions, the synchronization point it mandates increases overhead, and thereby limits the usefulness of Ada in specifying loop-level parallelism.

3. In order to provide a sophisticated method of task communication and synchronization, an Ada implementation must maintain additional context (information associated with rendezvous, entry queues, task priorities, etc.). As the full generality of the Ada task is not needed to specify loop-level parallelism, unnecessary context is present. However, if a compiler can ascertain that a task will execute in a light-weight fashion (i.e. with no synchronization with other tasks), then it can implement this task in a more efficient manner [Fra87,Hil82b,Blu81].

[3] Although this initialization is performed serially in Figure 2.4, a more efficient tree-based solution is possible. However, as we shall see in Chapter 6 a bottleneck is also present in this solution.

```
declare
    task type iterate_task is
        entry get_id(id : in integer); -- iterate_task contains one entry.
    end iterate_task;
    par_loop : array (1..N) of iterate_task; -- Create N iterate_tasks.
    task body iterate_task is
        my_id : integer;
    begin
        accept get_id(id : in integer) do
            my_id := id; -- Acquire an id from parent.
        end get_id;
        S1
        ⋮
        Sn
    end iterate_task;
begin -- All iterate_tasks begin to execute.
    for i in 1..N loop -- Identities are distributed sequentially.
        par_loop(i).get_id(i);
    end loop;
end; -- Wait for all iterate_tasks to terminate.
```

Figure 2.4: An Ada Parallel Loop Equivalent

description here, but instead focus on other issues concerning the Ada tasking model. See ARM 9.7.2 and ARM 9.7.3 for more details.

2.2.1 The Suitability of Ada for Shared Memory Multiprocessors

Although it may appear that Ada's tasking model is designed with a distributed memory architecture in mind, it is nevertheless well-suited for shared memory multiprocessors. The very fact that Ada's block structure implies that more than one task can access the same variable implies that a shared memory architecture is required.

Several implementations of Ada exist on shared memory architectures [Ric89] [DFSS89,AL89,CCB89] [Alliant,Sequent]. In addition, it has also been shown that parallel numerical algorithms normally written in FORTRAN for these machines can also be written in Ada [Blu81,SS85]. The intended goal of our work is to add to this effort by illustrating how loop-level parallelism can be both expressed, and efficiently implemented, in Ada.

2.2.2 Deficiencies of Ada Tasking Model

Three significant deficiencies in the Ada tasking model are manifested when one attempts to express loop-level parallelism:

1. Although no parallel loop construct exists in Ada, one can use tasks to specify a semantically equivalent entity as shown by Figure 2.4. Constructing a parallel loop, like many other problems in parallel processing,

```
task buffer is
   entry write(in_item: in item);
   entry read(out_item: out item);
end buffer;

task body buffer is
   local_item: item;
begin
   accept write(in_item: in item) do
      local_item := in_item;
   end write;
   loop
      select
         accept read(out_item: out item) do
            out_item := local_item;
         end read;
      or
         accept write(in_item: in item) do
            local_item := in_item;
         end write;
      or
         terminate;
      end select;
   end loop;
end buffer;
```

Figure 2.3: An Example of a Selective Wait Statement

terminate, endlessly looping, waiting for entry calls. However, the terminate alternative specifies that upon reaching the select statement, if all potential callers have either terminated or reached a select statement with a terminate alternative, then this task terminates. The apparently infinite loop of Figure 2.3 is indeed finite. If upon reaching the select statement no entry calls are pending, but potential callers still exist, this task does not terminate; it waits for either an entry call or for all potential callers to terminate or reach a select with a terminate alternative.

The select statement can also be used for entry calls. We avoid a detailed

```
task student is
begin
   select
      accept study do
         ⋮
      end study;
   or
      accept have_fun do
         ⋮
      end have_fun;
   end select;
end student;
```

Figure 2.2: An Example of a Selective Wait Statement

In addition to this simple form, three other forms of select statements are available. In Figure 2.3, we consider a select statement with a terminate alternative. This figure illustrates an example of a monitor of a protected variable, local_item. All accesses to this variable are made by calling the read or write entries of the task buffer.

Consider the body of the buffer task. After accepting a write entry call to initialize the local variable, an apparently infinite loop containing a select statement with a terminate alternative is entered. The semantics of this select are as follows: each of the entries corresponding to the accept statements is evaluated to determine if a task has called that entry. If one or more of these entries has a task waiting, a rendezvous occurs with one of these waiting tasks as demonstrated in the simple version of the select statement.

At the conclusion of the rendezvous, the select statement has completed its execution. The next iteration of the loop is executed and the process continues. Without the presence of the terminate alternative, this task would never

```
shared : integer;

task t1 is
   entry sync;
end t1;

task body t1 is
   x : integer;
begin
   accept sync do
      null;
   end sync;
   x := shared; -- shared is declared globally
end t1;

task body t2 is
begin
   shared := 5;
   t1.sync; -- signal to t1 it can now use shared.
end t2;
```

Figure 2.1: A Rendezvous used for Synchronization

The Select Statement

A *select* statement can be used to non-deterministically accept one of several potential entry calls (ARM 9.7.1). Consider Figure 2.2 which shows an example of a select statement. In this example the student task wishes to accept either one of two entry calls. This can be accomplished by using a select statement. Upon reaching this statement, a check is made to determine if any tasks have called either the study or have_fun entries. If only one of these entries has a caller waiting, that entry is accepted and a rendezvous occurs. If neither entry has a caller waiting, execution blocks until a call on one of these entries occurs. If at least one caller exists for each of these entries, one is chosen non-deterministically and the rendezvous is performed. At the conclusion of the rendezvous the select statement is complete.

communication between two otherwise asynchronous tasks, Ada allows a task to declare one or several *entry* points. An entry may be called by another task to participate in a *rendezvous* with the first task. The task that declares this entry can specify when in its execution it wishes the rendezvous to occur by executing an accept statement naming the appropriate entry. Likewise, the calling task can specify the point at which it wishes to call an entry by executing an entry call. An entry may have parameters associated with it in the same manner as subprograms, making the rendezvous mechanism a means of communication as well as synchronization.

When two tasks perform an entry call and accept statement for the same entry, a rendezvous occurs (ARM 9.5(14)); the calling task suspends its execution while the body of the accept statement is executed.[2] Upon completion of this rendezvous, both tasks continue their independent execution at the point immediately following the body of the accept statement and entry call, respectively.

Figure 2.1 shows an example of how two tasks can synchronize access to a shared variable by using the rendezvous mechanism. In this example, tasks t1 and t2 coordinate their accessing of the global variable, shared, by synchronizing via the sync entry located in task t1. The programmer has deemed that up until t2 calls this entry, it is allowed to write to shared. After this has occurred, t2 signals to t1 that it may now access this variable exclusively by calling the sync entry.

[2] We assume that no other task has executed an entry call, prior to this entry call, for the same entry.

While the fetch_and_add primitive can be implemented in software using locks and/or critical sections, machines such as RP3 and Ultracomputer implement this primitive directly in hardware. This is accomplished by adding extra logic to the switches in the omega network. This logic has the ability to combine multiple fetch_and_add operations on the same variable, substituting one operation with an increment specified by the sum of the increments of the requests. By performing this combining in the network, multiple fetch_and_adds are satisfied in the time it takes for one shared memory access.

Consider the scenario where all N processors simultaneously issues a fetch_and_add operation on the same variable. Without the presence of a combining network, hot spots will arise at certain nodes of the network. By combining these requests, all N requests can be satisfied in the time it takes for one fetch_and_add operation.

2.2 Ada Tasking

In this section we present a brief overview of the Ada tasking model. In addition to highlighting some of its features, we illustrate three deficiencies of this model that arise when one attempts to express loop-level parallelism. Other features of the language are illustrated in subsequent chapters.

Parallelism in Ada is expressed by using the *task* construct (ARM 9(1-2)).[1] A task is an entity that executes in parallel with other tasks. To facilitate

[1]The semantics of the language are defined by the Ada Reference Manual [Ada83]. As many references are made to this document throughout this work, we employ a shorthand in referring to a particular section. We use "ARM **c.s(i)**" to correspond to item **i** of section **s** located in chapter **c** of the Ada Reference Manual.

2.1.2 Memory Hierarchy

An important consideration in constructing a shared memory machine is deciding how processors are connected to memory. Although we assume a configuration in the spirit of the Ultracomputer and RP3, our work can be tailored to alternative configurations. In what follows we discuss the memory hierarchies found in the Ultracomputer and RP3.

To facilitate massive parallelism in a uniform way, processors are connected to shared memory via a connection network. Both the RP3 and Ultracomputer utilize an *Omega*-network for this purpose. An omega network provides processors with uniform access to shared memory. In addition, an omega network provides a very efficient processor/memory routing scheme [GLR83].

In a traditional omega network (as used in the Ultracomputer) processors reside on one side of the network with the memory units on the opposite side [Law75]. Since a path of $\log N$ switches connects each processor to shared memory, all shared memory requests require $\log N$ steps. In an attempt to reduce the amount of network accesses, the designers of the RP3 also connect each processor directly to a memory unit. If a processor references shared memory of the memory unit that it is connected to, the access is performed locally, avoiding a network traversal. If a processor references shared memory that is *not* in its companion memory unit, then the network is used to satisfy this request.

In addition to this local memory, a cache is often provided to further reduce network accesses. In the RP3 this cache can be managed by software, allowing for both greater flexibility and responsibility in coordinating data placement. For more details see [BMW85].

For example, suppose V is a shared variable and processor i executes

$$X_i := \mathsf{fetch_and_add}\,(V, e_i),$$

while processor j executes

$$X_j := \mathsf{fetch_and_add}\,(V, e_j).$$

If V is not updated by any other processor, then the result of these two operations is either

$$X_i := V$$
$$X_j := V + e_i$$

or

$$X_i := V + e_j$$
$$X_j := V$$

As a further illustration of the usefulness of the fetch_and_add primitive we consider the following problem: suppose we wish to distribute N entries of an array to N distinct threads in parallel. By using the fetch_and_add primitive, we can solve this problem in a straightforward manner. First, initialize a shared variable to the value of the first index, n_1, of the array. Next have each of the N processors execute a fetch_and_add operation on this variable with a unit increment. By the semantics of the fetch_and_add primitive, each processor receives a unique index value ranging from n_1 to $n_1 + N - 1$.

We describe the implementation of the fetch_and_add primitive in the next section.

architecture allows faster access to local memory, while rendering non-local accesses more costly. A shared memory machine provides uniform access to all of memory, but this access cannot be performed in constant time.

We follow the work of Hummel [Hum88] and assume an asynchronous, shared memory multiprocessor as our model. The IBM RP3 [PBG+85] and its predecessor, the NYU Ultracomputer [GGK+83], are examples of machines that fall under this model.

2.1.1 The Fetch_And_Add Primitive

Our work utilizes the fetch_and_add primitive. The fetch_and_add primitive is a universal coordination primitive that provides an efficient means of synchronization among many asynchronous processors while avoiding "hot spots". The fetch_and_add primitive can be used as a solution to the mutual exclusion problem, the readers-writers problems, and other problems dealing with parallel activities [GLR83]. In addition, it can simplify program analysis; it presents a form of synchronization that can be used to prove an algorithm is free of race conditions.

The semantics of the fetch_and_add primitive are as follows: if a process executes fetch_and_add(shared_var, inc), the value of shared_var is incremented by inc, and the original value is returned. These two operations are performed *atomically*. If two processors execute fetch_and_add operations on shared_var, the result is that shared_var is incremented by the sum of the two increments and one processor is returned the original value of shared_var, while the other receives the sum of the original value and the first processor's increment.

Chapter 2

Background

This chapter is divided into three sections, providing the background for this work. The first section describes the machine model. The second section discusses the Ada tasking model, highlighting its advantages as well as its shortcomings. We conclude this chapter by discussing the NYU Ada/Ed system and related work.

2.1 The Machine Model

As our work deals with practical parallelism, our underlying machine contains multiple processors. A fundamental decision in designing a multiprocessor is the placement of memory. In a *shared memory* machine a common memory can be accessed by each processor. In a *distributed memory* machine each processor has its only local memory; no shared memory exists. Since both architectures provide the same computational power, the main issue is efficiency. A distributed

in this regard. Geoffrey Hunter [Hun90] postulates that,

> "The only technically rational way of advancing the art of scientific and engineering programming is to abandon FORTRAN in favor of a modern, block-structured language such as Algol-68 or Ada."

We believe this work will help towards this transition.

> "Ada tasking can be made as efficient as any other method of concurrency programming, but the generality of the Ada model requires special cases to be extracted and used."

This work will focus on making loop-level Ada tasking efficient by using program transformations similar to those described in [SS85], [Hil82a] and [Hil82b]. Using these methods, we are able to achieve an efficient implementation without sacrificing portability.

Chapter 2 provides a background for this work; it discusses our multiprocessor model and the fetch_and_add primitive. It also gives an overview of the Ada tasking model and introduces three shortcomings in Ada: the lack of the ability to distribute identities to tasks in parallel, the synchronization point that is required during task initiation, and the high overhead required to manage a task's status.

Chapter 3 not only solves the first of these two problems, but also makes the fetch_and_add primitive available at the Ada level. By doing this we provide the user with a efficient synchronization mechanism that can be used to prove an algorithm is free of race conditions.

Chapter 4 provides a solution for obtaining efficient loop-level parallelism. This solutions is illustrated by the examples in Chapter A. Chapter 7 concludes this work and suggest some future research directions.

The expected result of this work is to widen substantially the use of Ada in large scale scientific programming on current and future multiprocessor architectures. One may even hope for a progressive replacement of FORTRAN in this domain of applications, even though past history gives us the slimmest of hopes

to a parallel thread before it starts to execute, no synchronization is required between starting and terminating each parallel thread. This characteristic allows this type of parallelism to be implemented efficiently on a variety of machines.

Ada is a modern programming language which supports parallelism as well as programming in the large, data abstraction, strong typing, and exception handling. An ANSI standard [Ada83] is intended to ensure portability across implementations. Many compilers exist for Ada on sequential machines and efforts have been undertaken to make Ada available on parallel architectures as well [Ric89,AL89,CCB89]. For these reasons, Ada appears to be an excellent choice for algorithms geared toward parallel architectures.

The parallel construct of Ada is the *task*. It is a powerful generalization of coroutines, that supports coarse-grain parallelism, as well as synchronization and communication (both conditional and unconditional) among otherwise asynchronous processes. Due to its generality, a significant amount of overhead is required to activate and terminate a task [BN87,Ard87,Hil82b]. This overhead makes the Ada task seemingly inappropriate for loop-level parallelism [Yem82,Bur85,Jha90].

However, the attractiveness of programming in Ada with its desirable features (exception handling, programming in the large, strong typing, etc.) suggests that if the overhead associated with task initiation and termination were eliminated, Ada would be a language of choice for the largest scientific computations – an area of application where FORTRAN continues to reign. Frankel [Fra87] postulates,

```
doall i = 1, N
    S_1
    ⋮
    S_n
endall
```

Figure 1.1: A ParFOR Code Fragment

level of granularity required by multiprocessors [Ken88].

An alternative approach is to allow the programmer the full power to express the parallelism of an algorithm directly, by writing it in a parallel programming language [Bab88,Qui87]. This approach not only lets the programmer decide the appropriate granularity of parallelism, but also how and where this parallelism should be employed in the program. The manner in which the programmer is allowed to specify the parallelism in an algorithm is a very important decision.

An increasing amount of programs are being written for parallel machines using languages that are essentially sequential languages onto which parallel constructs have been grafted [IBM88,Ber88,Inc81,Ost89]. Since each parallel machine often provides its own idiosyncratic parallel constructs, portability across machines is troublesome. Although an effort has been undertaken to agree on common parallel constructs for multiprocessors [PCF88], it remains to be seen whether this work will develop into an industry-wide standard.

Figure 1.1 presents a code fragment written in a version of FORTRAN that has been modified to support parallelism [Ber88]. The semantics of the doall construct specify that each iteration of the loop may be executed in parallel. This medium grain or loop-level parallelism is the most common form of parallelism used in scientific applications. Each iteration of the loop executes the same set of instructions with only the loop index varying. Since each iterate can be given

Chapter 1

Introduction

More and more computationally intensive applications are being executed on parallel machines [AG89]. These machines achieve their increased processing power by exploiting the parallelism present in an application and distributing it among multiple processing elements. Since most of these applications have been written for sequential machines, their implementation on parallel architectures requires new approaches to algorithmic development.

One approach, the field of *Parallelizing Compilers* [Hin88,ACK87,ABC+87] takes programs written for sequential architectures and *automatically* converts them to semantically equivalent parallel programs. Typically, these compilers take as input a sequential FORTRAN program (FORTRAN still being the prevalent language for scientific computation) and produce a FORTRAN program with appropriate parallel constructs added. Although great success has been achieved in the related field of *Vectorizing Compilers* [AK84,KKLW84,Col87], compilers that generate code for vector machines, it remains to be seen whether parallelizing compilers can detect and exploit a sufficient amount of parallelism at the

6.4 The Body of a Minitask 150
6.5 $LBG_{ple}(\cdots)$, $LBG_{tree}(--)$, and $LBG_{gm}(—)$ 156
6.6 25% Efficiency for $E_{ple}(\cdots)$, $E_{tree}(- - -)$, and $E_{gm}(—)$ 160
6.7 A Non-Parallel Loop with Synchronization Inserted 165
6.8 An Example of Scalar Expansion 167
6.9 An Example of Loop Interchange 168
6.10 An Example of Loop Alignment 169

3.13 Generic Version of test_increment_reset and test_decrement_reset . 54
3.14 The revised insert and delete Routines 55
4.1 A Typical Parallel Loop 58
4.2 An Ada "Parallel Loop" 60
4.3 Task Creation and Activation Example 61
4.4 A Dynamic "Parallel Loop" 69
4.5 An Example of a Loop Body Procedure 74
4.6 The Visible Part of Gen_Minitask 76
4.7 A Typical Usage of gen_minitask 78
4.8 The First Version of Package Body of Gen_Minitask 81
4.9 The Body of a Minitask 82
4.10 The Improved Version of Package Body of gen_minitask 87
4.11 The Task_Completion and Task_Exception Functions 88
5.1 The Package Specification of Gen_Minitask 91
5.2 The Declarative Part of Gen_Minitask Body 94
5.3 The Sequence of Statements of the Gen_Minitask Body 100
5.4 Nested Parallel Loops using the Gen_Minitask idiom 110
5.5 The Body of Minitask Revisited 117
5.6 The Instantiation of Gen_Beacon Revisited 124
5.7 Minitask Exception Handling Algorithm 126
5.8 Instantiating Task's Algorithm 127
5.9 Algorithm to Create, and Execute a Minitask 128
6.1 An Ada Parallel Loop Equivalent (PLE) 131
6.2 A Parallel Loop Using the Gen_Minitask Idiom (GMI) 132
6.3 Tree-Based Initialization Variant of the Ada PLE 141

List of Figures

1.1 A ParFOR Code Fragment 2

2.1 A Rendezvous used for Synchronization 12

2.2 An Example of a Selective Wait Statement 13

2.3 An Example of a Selective Wait Statement 14

2.4 An Ada Parallel Loop Equivalent 16

2.5 Dritz's Self_Scheduling_Loops 23

3.1 Definition of the beacon Task Type as in [SS85] 27

3.2 Code Fragment for T.init(e) 30

3.3 Code Fragment for an Unconditional Entry Call 32

3.4 Code Fragment for a Conditional Entry Call 33

3.5 Code Fragment for a Timed Entry Call with delay D 34

3.6 Body of Beacon Package as in [Hum88] 36

3.7 Beacon Task Type as in [Hum88] 38

3.8 The Generic Beacon Package 41

3.9 Beacon Task Body for the Generic Beacon Package 43

3.10 The parallel_queue Generic Package 47

3.11 The insert and test_increment_reset Routines 49

3.12 The delete and test_decrement_reset Routines 51

List of Tables

3.1 Code Fragments for read, write, and faa Subprograms 45

4.1 Five Constraints of a Parallel Loop Idiom 74

5.1 The Elaboration of the Declarative Part of the Gen_Minitask . . . 99

5.2 The Four Steps Involved in Elaborating an Object Declaration . 104

5.3 The Ada/Ed Task Control Block 113

5.4 Tcb Fields Needed for Minitask Implementation 114

5.5 Tcb Fields Needed for Minitasks that Contain Tasks 115

5.6 Cache Actions Required for Minitasks 123

5.7 Implementation of the task_completion and task_exception Subprograms 129

6.1 Task Initiation Overhead for the Ada PLE 137

6.2 Task Initialization Overhead for the Ada PLE 138

6.3 Per Task Initialization Overhead for Tree-Based Version 143

6.4 Initiation Overhead for a Family of Minitasks 149

6.5 Initialization Overhead for each Minitask 151

6.6 Overhead Requirement for Each Parallel Loop Idiom 152

6.7 Average Thread Size Required for Various Efficiency Levels . . . 162

6.5 Comparing Parallel Loop Idioms 151
6.5.1 Granularity 152
6.5.2 Efficiency 157
6.6 Other Types of Loops 164
6.6.1 Scalar Expansion 166
6.6.2 Loop Interchange 166
6.6.3 Loop Alignment 168

7 Conclusions **170**
7.1 Our Work 171
7.2 Ada 9X 174
7.2.1 Managing Large Number of Tasks 175
7.2.2 Vector Architectures 177
7.3 Future Work 179

A Examples **182**
A.1 Calculate_Pi 182
A.2 Gauss_Jordan 186
A.3 Matrix Multiplication 189

5 Gen_Minitask Implementation **89**

5.1 Elaborating the Package Specification 91

5.2 Elaborating the Declarative Part 93

5.3 Executing the Sequence of Statements 98

5.4 Removing the Synchronization Point 102

5.4.1 One-Shot Allocation of Stacks 106

5.4.2 Stack Recycling 107

5.4.3 A Critique of the Stack Recycling Scheme 108

5.5 Reducing Minitask Overhead 111

5.5.1 Parallelism Within a Minitask 112

5.5.2 Further Reducing Beacon Overhead 116

5.5.3 Ada Shared Variables 119

5.5.4 Handling Exceptions 123

6 Performance Analysis **130**

6.1 Preliminaries 133

6.1.1 Assumptions 134

6.2 The Ada Parallel Loop Equivalent (PLE) 134

6.2.1 Task Initiation 135

6.2.2 Task Initialization 136

6.3 Tree-Based Initialization 140

6.3.1 Initialization Overhead 142

6.4 The Gen_Minitask Idiom (GMI) 146

6.4.1 Minitask Initiation 147

6.4.2 Minitask Initialization 149

3.1 The Beacon Task Type 26
3.1.1 Transforming the Beacon Task 28
3.1.2 Handling Aborts 34
3.2 Packaging a Beacon Task 35
3.3 Our Construct: The Generic Beacon Package 40
3.3.1 Implementation 42
3.4 An Example 46
3.4.1 An Apparent Shortcoming of Gen_Beacon 52

4 The Gen_Minitask Package 56

4.1 Types of Parallelism 56
4.2 Task Initiation 60
4.2.1 Task Dependence 63
4.2.2 Tasks and Exceptions 64
4.3 Removing the Synchronization Point 68
4.3.1 An Observation 68
4.4 Reducing Tasking Context 70
4.4.1 The Pragma Approach 71
4.4.2 The Compiler Detection Approach 72
4.4.3 The Idiom Approach 73
4.5 Our Idiom – The Gen_Minitask Package 73
4.5.1 Gen_Minitask Usage 77
4.5.2 The Gen_Minitask Body 80
4.5.3 The Completion Status Routines 88

Contents

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

2 Background **6**

2.1 The Machine Model 6

2.1.1 The Fetch_And_Add Primitive 7

2.1.2 Memory Hierarchy 9

2.2 Ada Tasking 10

2.2.1 The Suitability of Ada for Shared Memory Multiprocessors 15

2.2.2 Deficiencies of Ada Tasking Model 15

2.2.3 Shared Variables in Ada 18

2.3 Related Work 20

2.3.1 The NYU Ada/Ed System 20

2.3.2 Dritz's Work 21

3 Beacon Tasks and Fetch_And_Add **24**

her inspiration, and her love.

particular, I would like to thank Ron Cytron for his guidance and insight.

I would also like to acknowledge David Clark, Anthony Dos Reis, Hanamantagouda Sankappanavar, and the rest of the faculty of the Mathematics and Computer Science department at S.U.N.Y. New Paltz for their fine instruction during my undergraduate years. In particular, I would like to thank Hanamantagouda Sankappanavar whose guidance and inspiration eventually led to my decision to pursue a doctorate.

I have had the pleasure of knowing many students during my years at Courant. I thank all of them for making this time memorable. In particular, I would like to thank my friend and officemate, Ernest Campbell, for the many theoretical discussions we have shared. I would also like to thank Pankaj Agarwal for his motivation and friendship, Babu Narayanan for letting me win a few racquetball games and making me laugh, Juan Cardenas, Prasad Tetali, Naomi Silver, Patricia Bedard and Karen Clark for many good times, and Venkataraman Sundareswaran for showing me how to draw. John Goodman, my officemate for my complete stay at NYU, deserves special thanks for his advice, for putting up with me, and especially for use of his Ada Reference Manual.

I would also like to thank the members and officers of the Imprecision Bridge Club at NYU for offering both a challenging and friendly outlet during the many hectic years of my study. In particular, I would like to thank Ofer Zajicek for letting me become involved with the club, as well as for his advice with my work.

Last, but *certainly* not least, I would to thank my fiancee, Laureen Treacy. From the start she has encouraged me to pursue my dreams and helped me in every way that she could. Her careful proofreading and advice have no doubt contributed to the quality of this work. I will forever appreciate her sacrifices,

Acknowledgements

First and foremost I would to thank my parents and the rest of my family who have supported me throughout the years. Their sacrifices and encouragement will always be remembered.

I would like to thank my advisor, Edmond Schonberg, not only for his technical guidance, but also for his encouragement and motivation in helping me produce this work. I would like to thank Robert Dewar for insight into shared variables, and Ben Goldberg and Paul Hilfinger for their careful reading and valuable comments. In addition, I would like to thank the rest of my committee, Malcolm Harrison and Alan Gottlieb.

I would also like to thank Susan Flynn Hummel and other members of the NYU Ada/Ed group, past and present. In particular, Bernard Banner deserves a special thank you for the help he has given me during my time with the group. I would also like to thank Bob Paige for his encouragement, Anina Karmen-Meade for always being helpful, and the rest of the faculty and administrative staff at Courant for their services.

I am thankful to Fran Allen and the members of the PTRAN group at IBM Research (Michael Burke, Ron Cytron, Jeanne Ferrante, Vivek Sarkar, and Dave Shields, etc.) for making my stays with them enjoyable and educational. In

Abstract

Parallelism in scientific applications can most often be found at the loop level. Although Ada supports parallelism via the *task* construct, its coarseness renders it unsuitable for this *light-weight* parallelism. In this work, we propose Ada constructs to achieve efficient loop-level parallelism in ANSI-Ada. This is accomplished in two steps. First, we present an idiom that allows the specification of light-weight tasks. Second, we give an efficient implementation of this idiom that is considerably more efficient than a standard Ada task.

In addition, we present an idiom that makes the fetch_and_add synchronization primitive available at the Ada level. Our implementation of this idiom is more efficient in both time and space than previous results. In addition to providing universal synchronization, using fetch_and_add simplifies program analysis (e.g. proving the absence of race conditions in the implementation of a parallel algorithm). Since all these idioms are written in standard Ada, they maintain the portability that is central to the mandated uses of the language.

© Copyright 1991

by Michael Hind

All Rights Reserved

Efficient Loop-Level Parallelism in Ada

Michael Hind

October 1991

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University

Approved: _____

Edmond Schonberg

Research Advisor