Performance of a Class of Highly-Parallel Divide-and-Conquer Algorithms

John Thornley
Computer Science Department
California Institute of Technology
Pasadena, California 91125, U.S.A.
john-t@cs.caltech.edu
http://www.cs.caltech.edu/~john-t/

October 1, 1995

Abstract

A wide range of important problems have efficient methods of solution based on the divide-and-conquer strategy. However, the traditional approach to parallel divide-and-conquer does not scale well due to the sequential component of the algorithms. We investigate the performance of a non-traditional, highly-parallel divide-and-conquer strategy that we refer to as "one-deep parallel divide-and-conquer". As representative examples, we measure and analyze the performance of highly-parallel variants of the quicksort and mergesort algorithms on a shared-memory multiprocessor. Both algorithms deliver impressive speedups. We present a systematic approach to modeling the performance of one-deep parallel divide-and-conquer algorithms and we demonstrate this approach with the two parallel sorting algorithms.

1 Introduction

A large number of important problems from a wide range of application areas have efficient methods of solution based on the divide-and-conquer strategy. The traditional divide-and-conquer approach consists of: (i) dividing the problem into a fixed number of smaller subproblems of the same type, (ii) solving the subproblems recursively, and (iii) combining the solutions to give the solution to the original problem. Divide-and-conquer algorithms traditionally are parallelized by solving the subproblems in parallel at the upper levels of recursion. However, the parallelism obtained by this approach does not scale well, due to the sequential composition of the divide and combine operations before and after the parallel solution of the subproblems.

In this report, we investigate the performance of a class of algorithms based on a non-traditional, highly-parallel divide-and-conquer strategy that we refer to as "one-deep parallel divide-and-conquer". The one-deep parallel divide-and-conquer approach consists of: (i) dividing the problem into a variable number of subproblems using a scalable parallel algorithm, (ii) solving the subproblems in parallel using an efficient sequential algorithm, and (iii) combining the solutions using a scalable parallel algorithm to give the solution to the original problem. The key points are that the divide and combine operations are scalable parallel algorithms, there is only one level of problem division, and the number of subproblems can be varied depending on the size of the problem and the number of processors.

As representative examples, we present and analyze the performance of one-deep parallel quicksort and mergesort algorithms. Between them, these two algorithms demonstrate the full scope of the general one-deep parallel divide-and-conquer strategy. The quicksort algorithm performs a parallel K-way partitioning, then sorts the K subarrays in parallel. The mergesort algorithm sorts K equal-sized subarrays in parallel, then performs a parallel K-way merge of the sorted subarrays. These algorithms have very small sequential components, allowing much greater scalability than traditional parallel quicksort and mergesort algorithms. The ingenuity in the algorithms is in the parallel partitioning and merging.

We have implemented both parallel sorting algorithms and measured their performance on a shared-memory multiprocessor. For an array of 5 million elements, the one-deep parallel sorting algorithms give speedups of up to 13-fold on 16 processors and 22-fold on 32 processors, over an optimized sequential quicksort algorithm. In comparison, traditional parallel quicksort gives speedups of only 6-fold on 16 processors and 7-fold on 32 processors.

Neither of the two sorting algorithms is new. The parallel quicksort is also known as the PS (Probabilistic Splitting) algorithm [1], and the parallel mergesort is also known as the PSRS (Parallel Sorting by Regular Sampling) algorithm [2]. However, these algorithms have not previously been recognized as instances of the much wider class of one-deep parallel divide-and-conquer algorithms, and their performance has not previously been analyzed in this more general context.

We present a generic framework for modeling the performance of one-deep parallel divide-and-conquer algorithms. This performance modeling framework provides a systematic method of understanding the factors that influence and limit the performance of this class of algorithms. In this report, we confine our performance models and analysis to shared-memory implementations. However, one-deep parallel divide-and-conquer algorithms also are well-suited to distributed-memory implementations, where similar modeling techniques are applicable.

The structure of one-deep parallel divide-and-conquer algorithms lends itself to a simple framework for performance modeling. All one-deep parallel divide-and-conquer algorithms consist of a sequence of stages, where each stage is either a small sequential operation or a parallel loop in which the iterations are executed independently. The basic performance model ignores the small sequential stages and presumes perfect efficiency of the parallel stages. The coefficients and validity of the model are determined by fitting the model to measured performance results using least-squares linear regression. If the goodness of fit is insufficient, the model can selectively be extended to take memory contention into account and to include the larger sequential stages of the algorithm.

To demonstrate the general performance modeling framework, we formulate and test performance models for our implementations of the one-deep parallel sorting algorithms. For these two algorithms, the basic performance model fits remarkably well for both small and larger numbers of processors. Modeling memory contention marginally improves the parallel quicksort model and does not significantly improve the parallel mergesort model. From these models, we discover that diminishing speedups observed for increasing processors are primarily caused by the inherent limitations of the parallel sorting algorithms, not by increasing contention for memory.

The remainder of this report is organized as follows: in section 2, we describe our parallel programming model and notation; in section 3, we describe our experimental methods; in section 4, we describe the traditional sequential and parallel divide-and-conquer strategies; in section 5, we describe the one-deep parallel divide-and-conquer strategy; in section 6, we describe the one-deep parallel quicksort algorithm; in section 7, we describe the one-deep parallel mergesort algorithm; in section 8, we present and discuss performance measurements for both parallel sorting algorithms; in section 9, we model the performance of one-deep parallel divide-and-conquer algorithms in general and the parallel sorting algorithms in particular; in section 10, we summarize and conclude.

2 Parallel Programming Model and Notation

In this section, we describe the parallel programming model that we assume and the notation that we use to express our algorithms.

2.1 Parallel Programming Model

We present our algorithms in the context of a very general parallel programming model. Programs are assumed to execute as a collection of asynchronous concurrent threads of control, with all threads sharing access to a single memory-space and a common pool of processors. Concepts of local versus non-local memory access and of the mapping of threads onto processors are outside the scope of the programming model. These issues may be handled differently by different implementations.

2.2 Parallel Programming Notation

We express our algorithms in a simple Ada/Pascal-like pseudocode enhanced with parallel block and parallel for-loop statements for creating parallel threads of control. Our algorithms do not require any synchronization constructs, except for the implicit barrier associated with parallel block and parallel for-loop statements. The parallel block statement has the following form:

```
parbegin
   statement_1
   ...
   statement_n
parend;
```

The statements of a parallel block are executed as parallel threads of control. Execution of the parallel block terminates after all the individual statements have terminated execution. The parallel for-loop statement has the following form:

```
parfor I in First .. Last loop
   sequence-of-statements
end loop;
```

The iterations of a parallel for-loop are executed as parallel threads of control. Execution of a parallel for-loop terminates after all the individual iterations have terminated execution.

We restrict our use of these constructs to cases where there are no data dependencies between parallel threads of control, i.e., we do not use parallel blocks or parallel for-loops in which one thread writes to a given variable and another thread reads or writes to the same variable in parallel.

3 Experimental Methods

In this section, we describe the methods that we use in our performance experiments.

3.1 Programming Language

The algorithms that we discuss and analyze have been implemented in Ada 95 [3]. Ada 95 was chosen primarily because it is a conventional imperative language that provides language-level support for parallelism. The algorithms could also be implemented using almost any other structured imperative language with calls to a thread library.

Although Ada 95 does not directly support parallel blocks and parallel for-loops, there exist extremely straightforward transformations of these constructs into equivalent standard Ada 95 tasking constructs. A parallel block is equivalent to a block containing a sequence of task declarations, and a parallel for-loop is equivalent to a block declaring an array of tasks. These simple transformations are given in appendix A and discussed in [4]. In our programs, parallel blocks and parallel for-loops were manually transformed into Ada tasking constructs.

3.2 Compiler

The programs were compiled with the GNAT (GNU-NYU Ada Translator) compiler [5], release 2.08 for the SGI–IRIX operating system. GNAT is an Ada 95 front-end and runtime system for the GCC (GNU C Compiler) family of compilers [6]. GNAT is part of the GNU software distributed by the Free Software Foundation.

The efficiency of code produced by GNAT is comparable to that produced by GCC for C/C++ programs. This to be expected, since Ada 95 is a conventional imperative language with similar constructs and capabilities to C/C++ and the GNAT compilation system uses the standard GCC code generator and optimizer as its back-end. Ada 95 tasking is implemented on top of Pthreads (POSIX threads) as described in [7]. For our performance experiments, the programs were compiled with the `-O2` optimization option and with index and range checking suppressed.

3.3 Computer System

Our experiments were performed on from 1 to 32 processors of a 36-processor SGI CHALLENGE system running the IRIX 6.1 operating system. Details of the system are given in Figure 1.

Name	cydrome
Kind	SGI CHALLENGE
Operating system	IRIX 6.1
Processors	36 × 100 MHz MIPS R4400
Data cache	16 Kbytes
Instruction cache	16 Kbytes
Secondary cache	1 Mbyte
Main memory	768 Mbytes, 2-way interleaved

Figure 1: Computer system details.

The experiments were performed when there were no other active processes executing on the system. For each experiment, five to ten trials were performed and averaged with outliers discarded to ensure consistent and accurate measurements.

3.4 Sorting Comparison

The parallel sorting speedups that we report are all relative to a good sequential quicksort adapted from [8, section 8.2]. This quicksort also is used as the sequential base-case in our parallel sorting algorithms. Therefore, any improvements to the sequential quicksort would also improve the performance of the parallel algorithms. All our experiments involve sorting arrays of 32-bit integers randomly generated with a uniform distribution.

Quicksort is widely held to be the fastest general-purpose sequential sorting algorithm and is therefore the sensible basis of comparison for claims regarding parallel sorting speedups. We do not compare a parallel program running on many processors to the same program running on a single processor. Such comparisons (though commonplace) are misleading and inflationary.

4 Traditional Divide-and-Conquer

In this section, we describe the traditional sequential and parallel divide-and-conquer problem-solving strategies. We explain why the performance of traditional parallel divide-and-conquer does not scale to more than a small number of processors for most problems.

4.1 Traditional Sequential Divide-and-Conquer

In the traditional formulation of divide-and-conquer, a problem is divided into a fixed number of smaller problems of the same type, the subproblems are solved, then the solutions to the subproblems are combined to give the solution to the original problem. Base-case problems, i.e., problems that are sufficiently small or simple, are solved directly. An algorithm template for traditional sequential divide-and-conquer is given in Figure 2. Although the algorithm template is expressed recursively, the implementation of a divide-and-conquer algorithm may be either recursive or iterative.

```
procedure Sequential_Solve (P :  in     Problem;
                            S :     out Solution) is
   P1, ..., PK :  Problem;
   S1, ..., SK :  Solution;
begin
   if Is_Base_Case(P) then
      Base_Case_Solve(in P, out S);
   else
      Divide(in P, out P1, ..., out PK);
      Sequential_Solve(in P1, out S1);
      ...
      Sequential_Solve(in PK, out SK);
      Combine(in S1, ..., in SK, out S);
   end if;
end Solve;
```

Figure 2: Traditional sequential divide-and-conquer template.

A significant characteristic of traditional divide-and-conquer is that the number of subproblems is a fixed attribute of the algorithm. The number of subproblems does not vary with the problem size or the available computational resources.

Efficient divide-and-conquer algorithms have been devised to solve problems from a diverse range of application areas, e.g., sorting (quicksort and mergesort), searching (binary search), matrix multiplication (Strassen's algorithm), geometric algorithms (convex hull and closest pair), and signal processing (fast Fourier transform). Any comprehensive algorithms text, e.g., [9], presents and discusses a large number of divide-and-conquer algorithms.

One of the canonical examples of divide-and-conquer is quicksort [10][11]. In the quicksort algorithm, an array is sorted in place by partitioning the elements into two smaller subarrays and a pivot element. Elements in the left subarray are no greater than the pivot element, and elements in the right subarray are no less than the pivot element. The two subarrays are then recursively sorted, leaving the entire array sorted. For efficiency, small arrays may be sorted using insertion sort. The sequential quicksort algorithm is given in Figure 3.

```
procedure Sequential_Quicksort (Data : in out array [First .. Last] of Element) is
   Pivot : First .. Last;
begin
   if Last - First + 1 <= Small_Length then
      Insertion_Sort(in out Data);
   else
      Partition(in out Data, out Pivot);
      Sequential_Quicksort(in out Data[First .. Pivot - 1]);
      Sequential_Quicksort(in out Data[Pivot + 1 .. Last]);
   end if;
end Sequential_Quicksort;
```

Figure 3: Quicksort as an example of traditional sequential divide-and-conquer.

In the quicksort algorithm, the divide operation of the divide-and-conquer template is complex and the combine is trivial. It is often the case that either the divide or the combine operation is degenerate. For example, in the mergesort algorithm, the divide operation is trivial (division into evenly-sized subarrays) and the combine operation is complex (ordered merge).

4.2 Traditional Parallel Divide-and-Conquer

The most straightforward method of obtaining parallelism in the traditional divide-and-conquer strategy is to solve the subproblems in parallel at each level of recursion [12][13]. Since the subproblems can be solved independently, the only synchronization required is the implicit barrier associated with the parallel block. For efficiency, to constrain the amount and granularity of parallelism, problems smaller than a given threshold size can be solved using sequential divide-and-conquer. The appropriate threshold is a function of the size of the initial problem and the characteristics of the target computer system, e.g., number of processors and thread management costs. An algorithm template for traditional parallel divide-and-conquer is given in Figure 4.

Based on the traditional parallel divide-and-conquer strategy, parallelism can be obtained from quicksort by sorting the two subarrays in parallel [14]. This parallel quicksort algorithm is given in Figure 5. In our experiments, we have found an appropriate threshold value to be `T = N/(4P)`, where `N` is the length of the entire array and `P` is the number of processors. This provides enough parallelism to take advantage of the available processors, without incurring the cost of creating too many fine-grain parallel threads.

Traditional parallel divide-and-conquer is simple to implement and can easily be applied to pre-existing sequential divide-and-conquer algorithms. However, the parallel speedup that can be obtained with this approach is inherently limited by the sequential composition of the divide and combine operations with the parallel block. The essential problem is that parallel speedup is constrained by the sequential component of a program, as described by Amdahl's law [15].

For parallel quicksort of `N` elements on `P` processors, the linear partition operation is executed before any parallelism is obtained at each level of recursion. For `P` a power of two and assuming partitioning of the array into two evenly-sized parts, parallel speedup is limited as follows:

```
procedure Parallel_Solve (P :  in     Problem;
                          T :  in     Problem_Size;
                          S :     out Solution    ) is
   P1, ..., PK :  Problem;
   S1, ..., SK :  Solution;
begin
   if Size(P) <= T then
      Sequential_Solve(in P, out S);
   else
      Divide(in P, out P1, ..., out PK);
      parbegin
         Parallel_Solve(in P1, in T, out S1);
         ...
         Parallel_Solve(in PK, in T, out SK);
      parend;
      Combine(in S1, ..., in SK, out S);
   end if;
end Solve;
```

Figure 4: Traditional parallel divide-and-conquer template.

```
procedure Parallel_Quicksort (T    :  in     Positive;
                              Data :  in out array [First .. Last] of Element) is
   Pivot :  First .. Last;
begin
   if Last - First + 1 <= T then
      Sequential_Quicksort(in out Data);
   else
      Partition(in out Data, out Pivot);
      parbegin
         Parallel_Quicksort(in T, in out Data[First .. Pivot - 1]);
         Parallel_Quicksort(in T, in out Data[Pivot + 1 .. Last]);
      parend;
   end if;
end Parallel_Quicksort;
```

Figure 5: Quicksort as an example of traditional parallel divide-and-conquer.

$$\begin{aligned}
parallel\ speedup\ (N,P) &= \frac{sequential\ execution\ time\ (N)}{parallel\ execution\ time\ (N,P)} \\
&\approx \frac{CNlog_2N}{C(N+\frac{N}{2}+\frac{N}{4}+\ldots+\frac{N}{P/2}+\frac{N}{P}log_2(\frac{N}{P}))} \\
&= \frac{log_2N}{2-\frac{1}{P/2}+\frac{1}{P}log_2(\frac{N}{P})} \\
&= \frac{Plog_2N}{2P-2+log_2(\frac{N}{P})} \\
parallel\ speedup\ (N,\infty) &= \frac{log_2N}{2}
\end{aligned}$$

In practice, due to thread management overheads, memory contention, imperfect load balancing, etc., achievable speedups are somewhat less than this upper limit. Figure 6 gives computed speedup limits and measured actual speedups for traditional parallel quicksort over sequential quicksort of 5,000,000 elements. These speedups demonstrate that performance of the traditional parallel quicksort algorithm does not scale to more than a small number of processors.

In general, the performance of traditional parallel divide-and-conquer does not scale well if either the divide or the combine operation is non-trivial. Since this is almost always the case, a more advanced strategy is required to obtain scalable parallelism from divide-and-conquer algorithms.

5 One-Deep Parallel Divide-and-Conquer

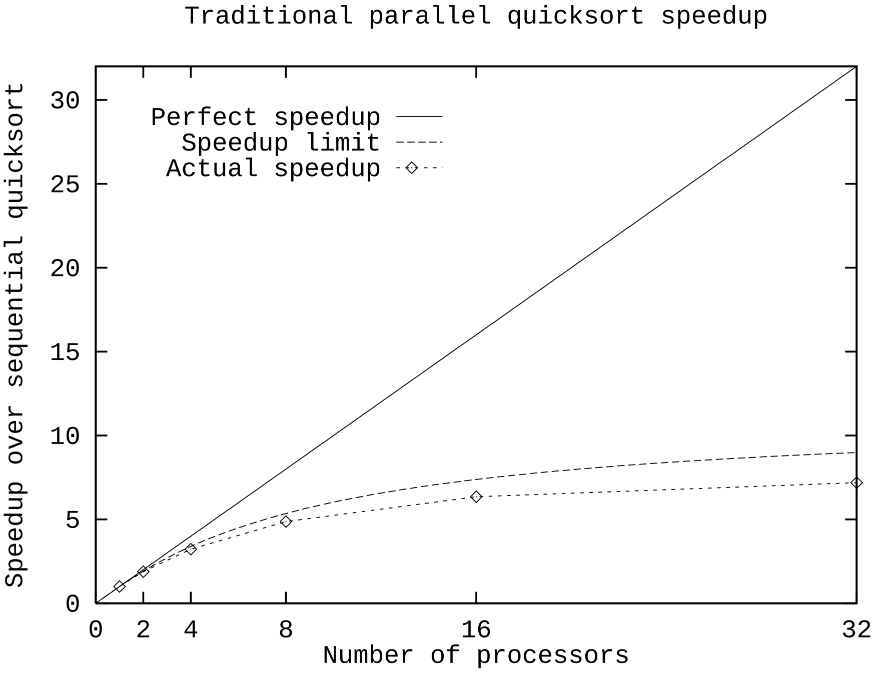
In this section, we describe the one-deep parallel divide-and-conquer problem-solving strategy. We explain why one-deep parallel divide-and-conquer algorithms are generally more efficient than equivalent traditional parallel divide-and-conquer algorithms.

5.1 General Framework

In one-deep parallel divide-and-conquer, a problem is divided into a variable number of subproblems of the same type, the subproblems are solved in parallel using an efficient sequential algorithm, then the solutions to the subproblems are combined to give the solution to the original problem. The divide and combine operations are parallel algorithms with parallelism scalable to the number of subproblems. The number of subproblems, `K`, is chosen according to factors such as the size of the problem and the number of processors. An algorithm template for one-deep parallel divide-and-conquer is given in Figure 7.

In general, one-deep parallel divide-and-conquer yields better parallel speedups than traditional parallel divide-and-conquer because all three phases of the algorithm are parallelized. If the divide and combine operations are completely parallel, there is no inherent sequential component to limit parallel speedup.

The emphasis of this report is on performance analysis. We present one-deep parallel variations on quicksort and mergesort as example algorithms because the problem of sorting requires no explanation and the two algorithms together cover the scope of the one-deep parallel divide-and-conquer strategy. Quicksort demonstrates a non-trivial parallel divide operation and mergesort demonstrates a non-trivial parallel combine operation. Other work [16] presents one-deep parallel divide-and-conquer algorithms to solve several more complicated problems, including the convex



Number of processors	Time (seconds)	Actual speedup	Speedup limit
sequential	31.13	—	—
1	31.17	1.00	1.00
2	16.56	1.88	1.91
4	9.67	3.22	3.39
8	6.39	4.87	5.35
16	4.90	6.35	7.38
32	4.33	7.19	8.99
∞	—	—	11.13

Figure 6: Speedup of traditional parallel quicksort over sequential quicksort of 5,000,000 elements.

```
procedure Parallel_Solve (K :  in     Positive;
                          P :  in     Problem;
                          S :     out Solution ) is
   Subproblems  :  array [1 ..  K] of Problem;
   Subsolutions :  array [1 ..  K] of Solution;
begin
   Parallel_Divide(in P, out Subproblems);
   parfor I in 1 ..  K loop
      Sequential_Solve(in Subproblems[I], out Subsolutions[I]);
   end loop;
   Parallel_Combine(in Subsolutions, out S);
end Parallel_Solve;
```

Figure 7: One-deep parallel divide-and-conquer template.

hull, Manhattan skyline, and closest pair problems.

5.2 Parallel Divide and Combine

The one-deep parallel divide-and-conquer strategy provides a structured approach to developing scalable highly-parallel algorithms from traditional divide-and-conquer algorithms. However, designing these algorithms is still likely to require some ingenuity. Devising efficient parallel `K`-way divide and combine algorithms is not always trivial. For example, the 2-way partitioning algorithm from traditional quicksort does not trivially generalize to an efficient `K`-way partitioning algorithm and it is not trivial to convert an efficient sequential `K`-way partitioning algorithm to an efficient parallel `K`-way partitioning algorithm.

In all one-deep parallel divide-and-conquer algorithms that we know, efficient parallel divide and parallel combine algorithms follow similar patterns. The parallel divide algorithm pattern is:

1. Use a small sample from the problem to compute the parameters for dividing the problem into the subproblems.

2. In parallel, for equal-sized segments of the problem, copy components of the problem to the appropriate subproblems.

The parallel combine algorithm pattern is:

1. Use a small sample from the subsolutions to compute the parameters for combining the subsolutions into the solution.

2. In parallel, for all the subsolutions, copy components of the subsolutions to the appropriate parts of the solution.

In most cases, the first stages of these patterns can be parallelized to a large extent as well. However, this is not usually efficient because the execution time of these operations is very small compared to that of the rest of the algorithm.

6 One-Deep Parallel Quicksort Algorithm

In this section, we describe the one-deep parallel quicksort algorithm. This algorithm is also known as the PS (Probabilistic Splitting) algorithm [1][17]. We present the algorithm as an instance of the one-deep parallel divide-and-conquer template.

6.1 Outer Structure

The one-deep parallel quicksort algorithm consists of two stages:

- Stage 1: Using a parallel algorithm, the `Data` array is partitioned into `K` partitions in the `Result` array. The elements of a given partition are no smaller than the elements of the partition to the left and no larger than the elements of the partition to the right.

- Stage 2: In parallel, the `K` partitions of the `Result` array are individually sorted. This leaves the `Result` array completely sorted.

The combine stage of the one-deep parallel divide-and-conquer template is degenerate for this algorithm. The one-deep parallel quicksort algorithm is given in Figure 8 and an example is given in Figure 9.

```
procedure Parallel_Quicksort (K      :  in     Positive;
                              Data   :  in     array [First ..  Last] of Element;
                              Result :     out array [First ..  Last] of Element ) is
   RP : array [0 ..  K] of First ..  Last + 1; -- Result Partitions.
begin
   -- Stage 1: In parallel, partition Data into Result.
   Parallel_Partition(in Data, out Result, out RP);
   -- Stage 2: In parallel, sequentially sort Result partitions.
   parfor I in 0 ..  K - 1 loop
      Sequential_Quicksort(in out Result[RP[I] ..  RP[I + 1] - 1]);
   end loop;
end Parallel_Quicksort;
```

Figure 8: One-deep parallel quicksort algorithm.

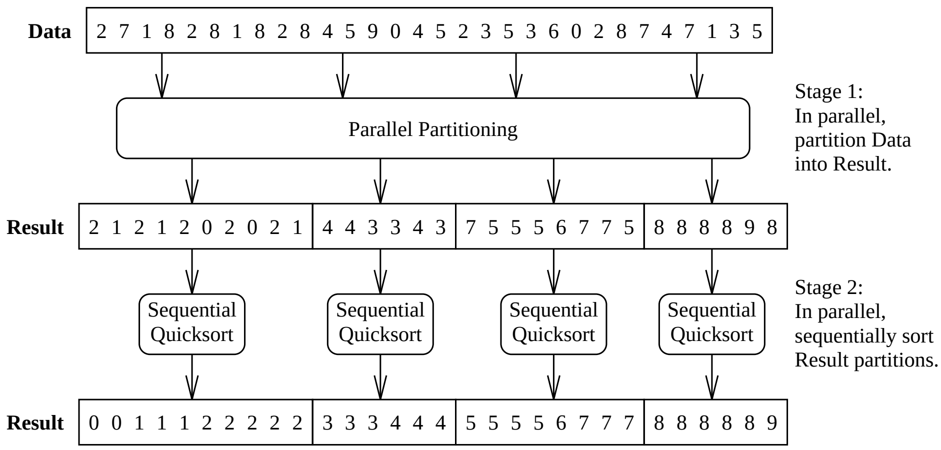


Figure 9: Example of one-deep parallel quicksort.

```
procedure Parallel_Partition (Data   :  in     array [First .. Last] of Element;
                              Result :     out array [First .. Last] of Element;
                              RP           out array [0 .. K] of First .. Last + 1) is
   M  :  constant Positive := Max(K, (Last - First + 1)/((K + 9)*(K + 9))); -- Sample length.
   S  :  array [0 .. M - 1] of Element;               -- Sample.
   P  :  array [1 .. K - 1] of Element;               -- Pivots.
   DS :  array [0 .. K] of First .. Last + 1;         -- Data Segments.
   EP :  array [First .. Last] of 0 .. K - 1;         -- Element Partitions.
   PC :  array [0 .. K - 1][0 .. K - 1] of Integer; -- Partition Counts.
   CP :  array [0 .. K - 1][0 .. K - 1] of First .. Last + 1; -- Copy Positions.
begin
   -- Stage 1.1: Take sample of elements from Data.
   for I in 0 .. M - 1 loop
      S[I] := Data[First + Integer(Float(Last - First + 1)*(Float(I)/Float(M)))];
   end loop;
   -- Stage 1.2: Sequentially sort sample.
   Sequential_Quicksort(in out S);
   -- Stage 1.3: Choose evenly-spaced pivot elements from sorted sample.
   for I in 1 .. K - 1 loop
      P[I] := S[Integer(Float(M)*(Float(I)/Float(K)))];
   end loop;
   -- Stage 1.4: Trivially, divide Data into equal-sized segments.
   for I in 0 .. K loop
      DS[I] := First + Integer(Float(Last - First + 1)*(Float(I)/Float(K)));
   end loop;
   -- Stage 1.5: In parallel, find Result partitions of Data elements and count elements per partition.
   parfor I in 0 .. K - 1 loop
      for J in 0 .. K - 1 loop PC[I][J] := 0; end loop;
      for J in DS[I] .. DS[I + 1] - 1 loop
         EP[J] := Find(Data[J], P);
         PC[I][EP[J]] := PC[I][EP[J]] + 1;
      end loop;
   end loop;
   -- Stage 1.6: Compute boundaries of Result partitions.
   RP[0] := First;
   for I in 1 .. K - 1 loop
      RP[I] := RP[I - 1];
      for J in 0 .. K - 1 loop
         RP[I] := RP[I] + PC[J][I - 1];
      end loop;
   end loop;
   RP[K] := Last + 1;
   -- Stage 1.7: Compute positions of Data elements in Result.
   for I in 0 .. K - 1 loop
      CP[0][I] := RP[I];
      for J in 1 .. K - 1 loop
         CP[J][I] := CP[J - 1][I] + PC[J - 1][I];
      end loop;
   end loop;
   -- Stage 1.8: In parallel, copy Data elements to Result.
   parfor I in 0 .. K - 1 loop
      for J in DS[I] .. DS[I + 1] - 1 loop
         Result[CP[I][EP[J]]] := Data[J];
         CP[I][EP[J]] := CP[I][EP[J]] + 1;
      end loop;
   end loop;
end Parallel_Partition;
```

Figure 10: Parallel K-way partitioning algorithm.

6.2 Parallel Partitioning

The parallel `K`-way partitioning algorithm is as follows:

- Stages 1.1–1.3: An ordered list of `K-1` pivot elements is chosen from the `Data` array. Each pivot will be the boundary element between two partitions in the `Result` array.

- Stages 1.4–1.5: In parallel, for equal-sized `Data` segments, the partition of each element is computed by searching for the position of the element in the list of pivots.

- Stages 1.6–1.7: The lengths and boundaries of the partitions and the positions of elements within the partitions are computed.

- Stage 1.8: In parallel, for the equal-sized `Data` segments, elements are copied to their positions in the partitions of the `Result` array.

The complete parallel partitioning algorithm is given in Figure 10. Note that only stages 1.5 and 1.8 are parallel loops. The loops in stages 1.1, 1.3, 1.4, and 1.7 are also parallelizable, but involve too little work for this to be efficient in any realistic scenario.

The efficiency of one-deep parallel quicksort is dependent on the `Result` partitions being of approximately equal lengths, to ensure good load balancing. This is achieved by sampling and sorting a larger number of elements, then choosing evenly-spaced pivots from the sorted sample. Issues involved in choosing an appropriate sample size are discussed in [18]. In our algorithm, we choose a sample size such that the time sorting the sample is inversely proportional to the time per processor sorting the `Result` partitions. In our experiments, we have found that this method produces almost equal partition lengths.

7 One-Deep Parallel Mergesort Algorithm

In this section, we describe the one-deep parallel mergesort algorithm. This algorithm is also known as the PSRS (Parallel Sorting by Regular Sampling) algorithm [2][17]. We present the algorithm as an instance of the one-deep parallel divide-and-conquer template.

7.1 Outer Structure

The one-deep parallel mergesort algorithm consists of three stages:

- Stage 1: Trivially, the `Data` array is divided into `K` equal-sized segments.

- Stage 2: In parallel, the `K` segments of the `Data` array are individually sorted.

- Stage 3: Using a parallel ordered-merge algorithm, the `K` sorted segments of the `Data` array are merged into the `Result` array.

The one-deep parallel mergesort algorithm is given in Figure 11 and an example is given in Figure 12.

7.2 Parallel Ordered-Merge

The parallel `K`-way ordered-merge algorithm is as follows:

- Stages 3.1–3.3: An ordered list of `K` pivot elements is chosen from the sorted `Data` segments. Each pivot will be the lower boundary element of a partition of the `Result` array.

```
procedure Parallel_Mergesort (K      :  in     Positive;
                              Data   :  in out array [First ..  Last] of Element;
                              Result :     out array [First ..  Last] of Element ) is
   DS : array [0 ..  K] of First ..  Last + 1; -- Data Segments.
begin
   -- Stage 1: Trivially, divide Data into equal-sized segments.
   for I in 0 ..  K loop
      DS[I] := First + Integer(Float(Last - First + 1)*(Float(I)/Float(K)));
   end loop;
   -- Stage 2: In parallel, sequentially sort Data segments.
   parfor I in 0 ..  K - 1 loop
      Sequential_Quicksort(in out Data[DS[I] ..  DS[I + 1] - 1]);
   end loop;
   -- Stage 3: In parallel, merge Data segments into Result.
   Parallel_Merge(in Data, in DS, out Result);
end Parallel_Mergesort;
```

Figure 11: One-deep parallel mergesort algorithm.

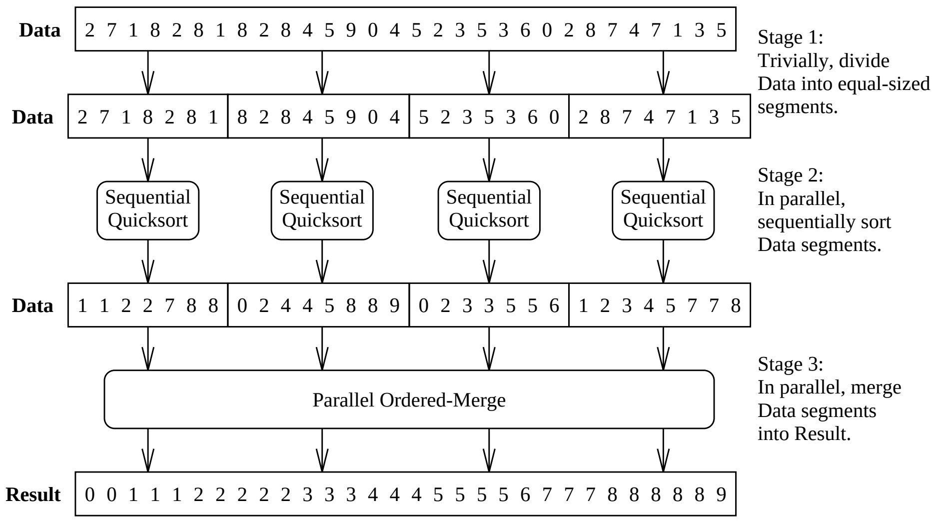


Figure 12: Example of one-deep parallel mergesort.

```
procedure Parallel_Merge (Data   : in     array [First .. Last] of Element;
                          DS     :     out array [0 .. K] of First .. Last + 1;
                          Result :     out array [First .. Last] of Element     ) is
   LP : array [0 .. 2*K*K - 1] of Element;          -- Local Pivots.
   PS : array [0 .. K - 1] of Segment;              -- Pivot Segments.
   MP : array [0 .. 2*K*K - 1] of Element;          -- Merged Pivots.
   GP : array [0 .. K - 1] of Element;              -- Global Pivots.
   SS : array [0 .. K - 1][0 .. K - 1] of Segment; -- Sub-Segments.
   RP : array [0 .. K] of First .. Last + 1;        -- Result Partitions.
   P  : First .. Last;
begin
   -- Stage 3.1: Choose evenly-spaced local pivot elements from Data segments.
   for I in 0 .. K - 1 loop
      PS[I].First := 2*I*K;
      PS[I].Last := PS[I].First + 2*K - 1;
      LP[PS[I].First] := Data[DS[I]];
      for J in 1 .. K - 1 loop
         P := DS[I] + Integer(Float(DS[I + 1] - DS[I])*(Float(J)/Float(K)));
         LP[PS[I].First + 2*J] := Data[P];
         LP[PS[I].First + 2*J - 1] := Data[P - 1];
      end loop;
      LP[PS[I].Last] := Data[DS[I + 1] - 1];
   end loop;
   -- Stage 3.2: Merge local pivot elements into one ordered list.
   Merge(in PS, in LP, out MP);
   -- Stage 3.3: Choose evenly-spaced global pivot elements from local pivot elements.
   for I in 0 .. K - 1 loop
      GP[I] := MP[Integer(Float(2*K*K)*(Float(I)/Float(K)))];
   end loop;
   -- Stage 3.4: Find boundaries of Data sub-segments between global pivot elements.
   for I in 0 .. K - 1 loop
      SS[0][I].First := DS[I];
      for J in 1 .. K - 1 loop
         SS[J][I].First := Find(GP[J], Data[DS[I] .. DS[I + 1] - 1]);
         SS[J - 1][I].Last := SS[J][I].First - 1;
      end loop;
      SS[K - 1][I].Last := DS[I + 1] - 1;
   end loop;
   -- Stage 3.5: Compute boundaries of Result partitions.
   RP[0] := First;
   for I in 1 .. K - 1 loop
      RP[I] := RP[I - 1];
      for J in 0 .. K - 1 loop
         RP[I] := RP[I] + SS[I - 1][J].Last - SS[I - 1][J].First + 1;
      end loop;
   end loop;
   RP[K] := Last + 1;
   -- Stage 3.6: In parallel, merge Data sub-segments into Result partitions.
   parfor I in 0 .. K - 1 loop
      Merge(in SS[I], in Data, out Result[RP[I] .. RP[I + 1] - 1]);
   end loop;
end Parallel_Merge;
```

Figure 13: Parallel K-way ordered-merge algorithm.

- Stage 3.4: The sorted `Data` segments are partitioned into sub-segments with respect to the pivots by searching for the positions of the pivots within each segment.

- Stage 3.5: The boundaries of the `Result` partitions are computed from the lengths of the `Data` sub-segments.

- Stage 3.6: In parallel, for all `Result` partitions, the corresponding `Data` sub-segments are merged into the `Result` partitions.

The complete parallel ordered-merge algorithm is given in Figure 13. Note that only stage 3.6 is a parallel loop. The loops in stages 3.1, 3.3, and 3.4 are also parallelizable, but involve too little work for this to be efficient in any realistic scenario.

As with one-deep parallel quicksort, the efficiency of one-deep parallel mergesort is dependent on the `Result` partitions being of approximately equal lengths, to ensure good load balancing. Again, this is achieved by sampling a larger number of elements to obtain the pivots. In our algorithm, we choose a sample of `2K` elements from each sorted `Data` segment, merge the samples, then choose evenly-spaced pivots from the merged sample. In our experiments, we have found that this method produces almost equal partition lengths.

8 Performance Measurements

In this section, we present and discuss performance measurements from experiments with both one-deep parallel sorting algorithms. Each experiment has three independent input parameters: (i) the number of elements to be sorted, (ii) the number of processors, and (iii) the number of threads in each parallel loop. Note that the number of threads need not be the same as the number of processors.

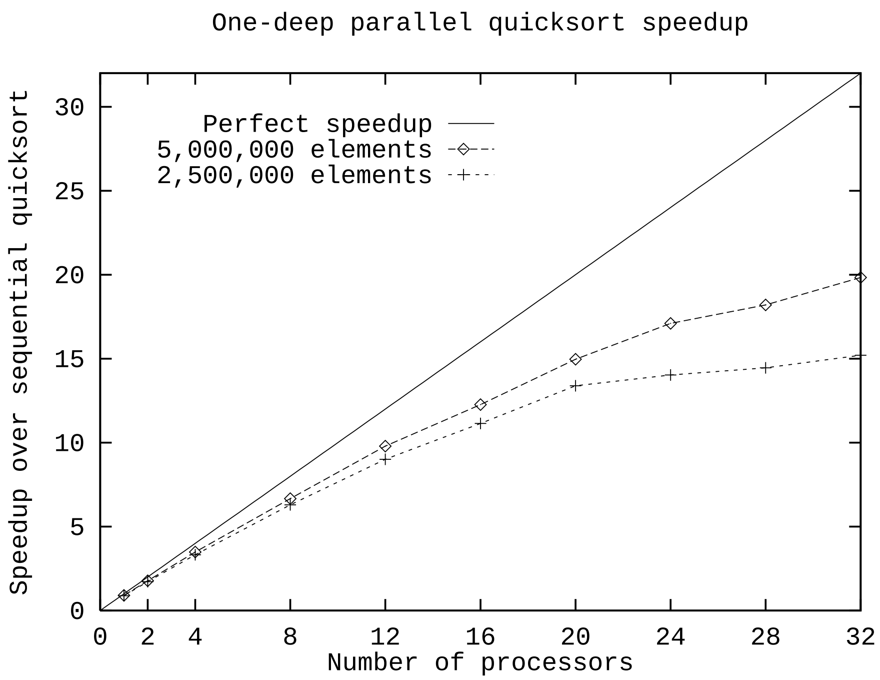
8.1 One-Deep Parallel Quicksort Performance Measurements

Figure 14 gives the speedup of one-deep parallel quicksort over traditional sequential quicksort for 2.5 million and 5 million elements on 1 to 32 processors. In each case, the time is given for the number of threads that produced the best speedup. For more than four processors, the best speedup was always obtained with the number of threads equal to the number of processors. Speedups increase with the number of processors and the number of elements. However, the rate of speedup decreases with the number of processors.

Figure 15 gives the breakdown of execution time per stage for one-deep parallel quicksort of 5 million elements on 1 to 32 processors. Times are given for the number of threads that produced the best speedup. The only stages that have significant execution times are the parallel stages: stage 1.4 (compute `Result` partitions of `Data` elements), stage 1.5 (copy `Data` elements to `Result` partitions), and stage 2 (sort `Result` partitions). The execution time of the sequential stages is insignificant.

8.2 One-Deep Parallel Mergesort Performance Measurements

Figure 16 gives the speedup of one-deep parallel mergesort over traditional sequential quicksort for 2.5 million and 5 million elements on 1 to 32 processors. In each case, the time is given for the number of threads that produced the best speedup. For more than one processor, the best speedup was always obtained with the number of threads equal to the number of processors. Speedups

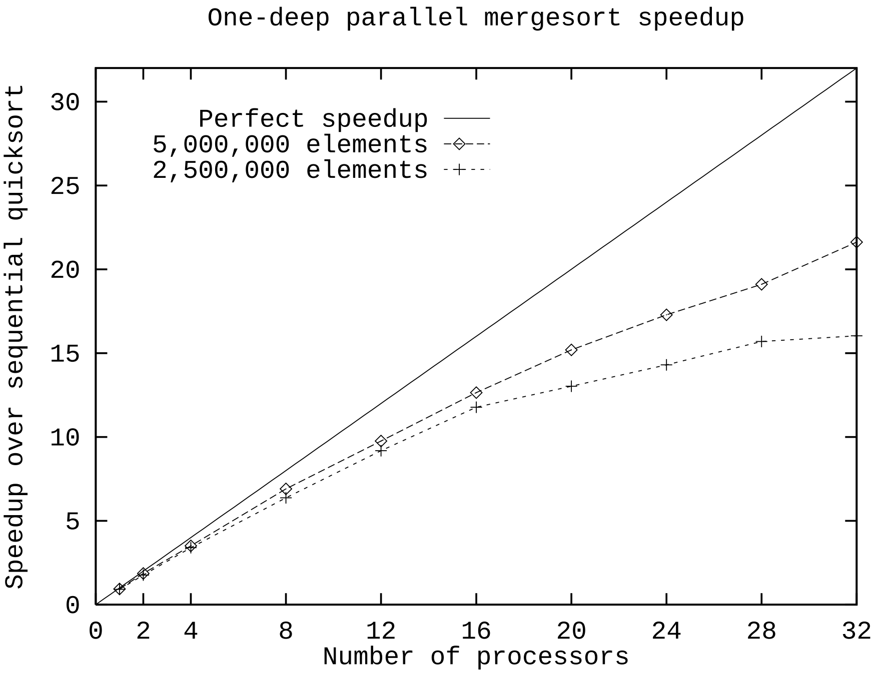


Number of processors	2,500,000 elements		5,000,000 elements	
	Time (sec)	Speedup	Time (sec)	Speedup
sequential	14.59	—	31.13	—
1	16.62	0.88	34.49	0.90
2	8.48	1.72	17.55	1.77
4	4.39	3.32	8.95	3.48
8	2.32	6.29	4.67	6.67
12	1.62	9.01	3.18	9.79
16	1.31	11.14	2.54	12.26
20	1.09	13.39	2.08	14.97
24	1.04	14.03	1.82	17.10
28	1.01	14.45	1.71	18.20
32	0.96	15.20	1.57	19.83

Figure 14: Speedup of one-deep parallel quicksort over traditional sequential quicksort.

Number of processors	Time per stage (seconds)								
	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	2
1	0.12	0.21	0.00	0.00	3.10	3.05	0.00	0.00	30.43
2	0.10	0.18	0.00	0.00	2.07	1.54	0.00	0.00	14.65
4	0.07	0.12	0.00	0.00	1.33	0.81	0.00	0.00	6.93
8	0.04	0.06	0.00	0.00	0.84	0.44	0.00	0.00	3.38
12	0.03	0.04	0.00	0.00	0.64	0.31	0.00	0.00	2.16
16	0.02	0.03	0.00	0.00	0.51	0.26	0.00	0.00	1.71
20	0.01	0.02	0.00	0.00	0.45	0.23	0.00	0.00	1.33
24	0.01	0.01	0.00	0.00	0.41	0.22	0.00	0.00	1.16
28	0.01	0.01	0.00	0.00	0.39	0.23	0.00	0.00	1.07
32	0.01	0.01	0.00	0.00	0.36	0.22	0.00	0.00	0.97

Figure 15: Breakdown of one-deep parallel quicksort execution-time for 5,000,000 elements.



Number of processors	2,500,000 elements		5,000,000 elements	
	Time (sec)	Speedup	Time (sec)	Speedup
sequential	14.59	—	31.13	—
1	16.05	0.91	33.23	0.94
2	8.30	1.76	16.73	1.86
4	4.29	3.40	8.87	3.51
8	2.29	6.37	4.51	6.90
12	1.59	9.18	3.19	9.76
16	1.24	11.77	2.46	12.65
20	1.12	13.03	2.05	15.19
24	1.02	14.30	1.80	17.29
28	0.93	15.69	1.63	19.10
32	0.91	16.03	1.44	21.62

Figure 16: Speedup of one-deep parallel mergesort over traditional sequential quicksort.

Number of processors	Time per stage (seconds)							
	1	2	3.1	3.2	3.3	3.4	3.5	3.6
1	0.00	30.41	0.00	0.00	0.00	0.00	0.00	3.16
2	0.00	14.27	0.00	0.00	0.00	0.00	0.00	2.45
4	0.00	7.01	0.00	0.00	0.00	0.00	0.00	1.90
8	0.00	3.26	0.00	0.00	0.00	0.00	0.00	1.24
12	0.00	2.19	0.00	0.00	0.00	0.00	0.00	0.99
16	0.00	1.59	0.00	0.00	0.00	0.01	0.00	0.85
20	0.00	1.27	0.00	0.00	0.00	0.01	0.00	0.76
24	0.00	1.09	0.00	0.00	0.00	0.01	0.00	0.68
28	0.00	0.97	0.00	0.00	0.00	0.02	0.00	0.63
32	0.00	0.83	0.00	0.00	0.00	0.02	0.00	0.58

Figure 17: Breakdown of one-deep parallel mergesort execution-time for 5,000,000 elements.

increase with the number of processors and the number of elements. However, the rate of speedup decreases with the number of processors.

Figure 17 gives the breakdown of execution time per stage for one-deep parallel mergesort of 5 million elements on 1 to 32 processors. Again, times are given for the number of threads that produced the best speedup. The only stages that have significant execution times are the parallel stages: stage 2 (sort `Data` segments) and stage 3.6 (merge `Data` segments into `Result` partitions). The execution time of the sequential stages is insignificant.

9 Performance Model and Analysis

In this section, we present a generic framework for modeling the performance of one-deep parallel divide-and-conquer algorithms. To demonstrate the framework, we develop performance models for our implementations of the one-deep parallel quicksort and mergesort algorithms. From these models, we draw conclusions regarding the factors that are most important to the performance of the algorithms.

9.1 The Value of a Performance Model

There is considerable value to having a quantitative analytic model of the performance of an implementation of a parallel algorithm:

- With a model of where time is spent in the execution an algorithm, effort can be appropriately directed to improving the efficiency of an algorithm and its implementation.
- With a model of the relationship between input parameters and execution time, parameters can be chosen to decrease execution time.
- With a model of the relationship between input parameters and execution time, execution times can be predicted to assist in the allocation of computing resources.

These capabilities can mostly be achieved by experimentation as well. However, computer time on large multiprocessors remains a scarce and valuable resource, making exhaustive experimentation prohibitively expensive.

9.2 Approaches to Performance Modeling

At one extreme, all questions regarding performance could be answered by experimentation. This approach is most likely too expensive. At the other extreme, an attempt could be made to automatically generate a quantitative or comparative performance model from the details of the algorithm, its implementation, and the computer system. This approach is most likely too ambitious.

Our approach to modeling the performance of one-deep parallel divide-and-conquer algorithms lies between these two extremes. The structure of the performance model is determined by traditional complexity analysis of each stage of the algorithm, then the coefficients of the model are determined using statistical analysis of a representative set of experimental performance measurements. Subsequently, the completed model can be used to accurately predict and analyze performance without further experimentation.

This approach is successful for one-deep parallel divide-and-conquer algorithms because of the relatively simple structure of these algorithms. In particular, there is no communication or synchronization between parallel threads of control except for the implicit barrier synchronization at the end of each parallel for-loop. Although this approach could also be applied to many other classes of algorithms, it does not generalize to all parallel algorithms.

9.3 One-Deep Parallel Divide-and-Conquer Performance Model Framework

One-deep parallel divide-and-conquer algorithms have a simple structure, consisting of a sequence of stages, where each stage is either a sequential operation or a parallel loop with independent iterations. This structure makes the algorithms amenable to performance modeling based on stage-by-stage complexity analysis.

The model consists of a formula equating total execution time to the sum of the execution times of the individual stages. In the basic model, we ignore the sequential stages and presume perfect efficiency of the parallel stages. This is reasonable because the sequential stages of efficient parallel algorithms are designed to have insignificant execution time, and the efficiency of independent parallel operations can be close to perfect on a moderate number of processors. The execution time of each stage is given by traditional complexity analysis with each term preceded by an unknown constant.

A series of experiments is performed to gather execution time measurements for a representative set of input and system parameters on the computer system of interest. Least-squares linear regression is used to obtain values for the constants in the model that best fit the performance measurements. The success of the model can be judged by the correlation coefficient and by the standard deviation of the difference between the performance measurements and the model. If necessary, the basic model can be extended by including terms representing the more time-consuming sequential stages and by taking memory contention into account in the terms representing the parallel stages.

In this report, our model is for a computer system in which each processor has a local cache memory and access to a shared main memory. Memory contention occurs when the rate of main memory accesses by parallel threads is too high because of insufficient locality, e.g., in a simple copy of one large array to another. We model memory contention by dividing the term representing a parallel operation into a term representing the degree to which the operation can be parallelized and a term representing the degree to which the operation cannot be parallelized:

$$C\frac{W}{P} \quad \longmapsto \quad C_a\frac{W}{P} + C_b W$$

Although simple, we have found that this model provides a good fit to measured memory contention behavior.

9.4 One-Deep Parallel Quicksort Performance Model

The basic performance model for the one-deep parallel quicksort algorithm is as follows:

$$
\begin{aligned}
execution\ time\ (seconds) &= C_{1.1}M + C_{1.2}M\,log_2(M) + C_{1.3}K + C_{1.4}K + \\
&\quad C_{1.5.1}\frac{N}{P}log_2(K) + C_{1.5.2}\frac{N}{P} + C_{1.6}K^2 + C_{1.8}\frac{N}{P} + C_2\frac{N}{P}log_2(\frac{N}{K}) \\
&\approx C_{1.5.1}\frac{N}{P}log_2(K) + (C_{1.5.2} + C_{1.8})\frac{N}{P} + C_2\frac{N}{P}log_2(\frac{N}{K})
\end{aligned}
$$

$$N = number\ of\ elements,\ P = number\ of\ processors,\ K = number\ of\ threads$$

We fitted this model to performance measurements for 82 different sets of input parameters. The fit is remarkably good: the correlation coefficient is 0.9996 (1.0 is a perfect fit) and the standard deviation is 0.48 seconds. The most likely source of memory contention in the algorithm is the linear-time copying of elements from the `Data` array to the `Result` array in stage 1.8. Modeling memory contention in stage 1.8 improves the model a little:

$$execution\ time\ (seconds) = C_{1.5.1}\frac{N}{P}log_2(K) + (C_{1.5.2} + C_{1.8a})\frac{N}{P} + C_{1.8b}N + C_2\frac{N}{P}log_2(\frac{N}{K})$$

The correlation coefficient remains at 0.9996, but the standard deviation improves to 0.28 seconds. Figure 18 compares this performance model with the experimental performance measurements. Adding the sequential stages of the algorithm and modeling memory contention in the other parallel stages does not futher improve the model.

9.5 One-Deep Parallel Mergesort Performance Model

The basic performance model for the one-deep parallel mergesort algorithm is as follows:

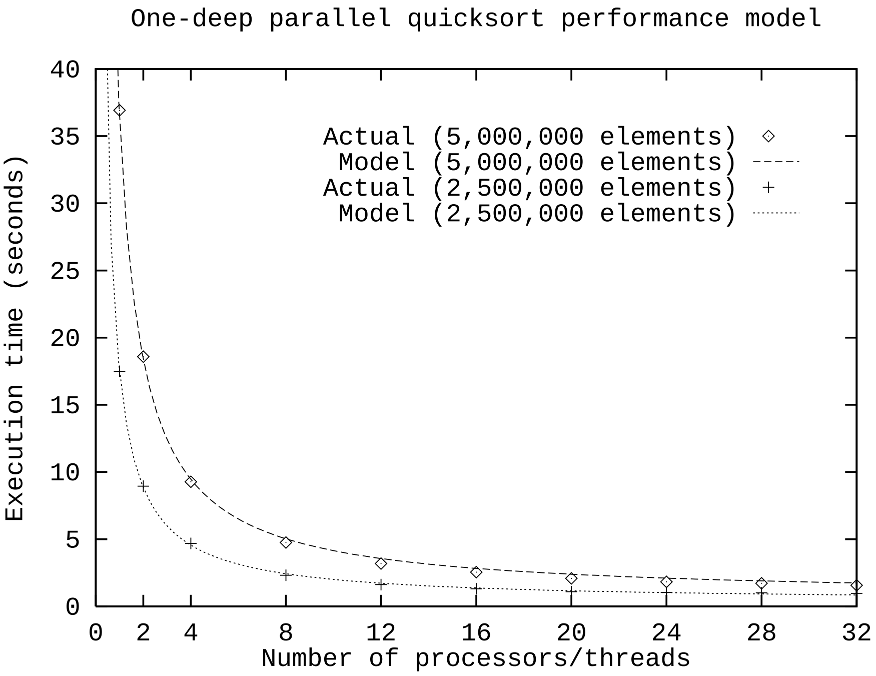
$$
\begin{aligned}
execution\ time(sec) &= C_1K + C_2\frac{N}{P}log_2(\frac{N}{K}) + C_{3.1}K^2 + C_{3.2}K^2log_2(K) + C_{3.3}K + \\
&\quad C_{3.4}K^2log_2(\frac{N}{K}) + C_{3.5}K^2 + C_{3.6.1}\frac{N}{P}log_2(K) + C_{3.6.2}\frac{N}{P} \\
&\approx C_2\frac{N}{P}log_2(\frac{N}{K}) + C_{3.6.1}\frac{N}{P}log_2(K) + C_{3.6.2}\frac{N}{P}
\end{aligned}
$$

$$N = number\ of\ elements,\ P = number\ of\ processors,\ K = number\ of\ threads$$

We fitted this model to performance measurements for 82 different sets of input parameters. The fit is good, with a correlation coefficient of 0.9964 and a standard deviation of 0.93 seconds. Modeling memory contention in stage 3.6 improves the model only very slightly:

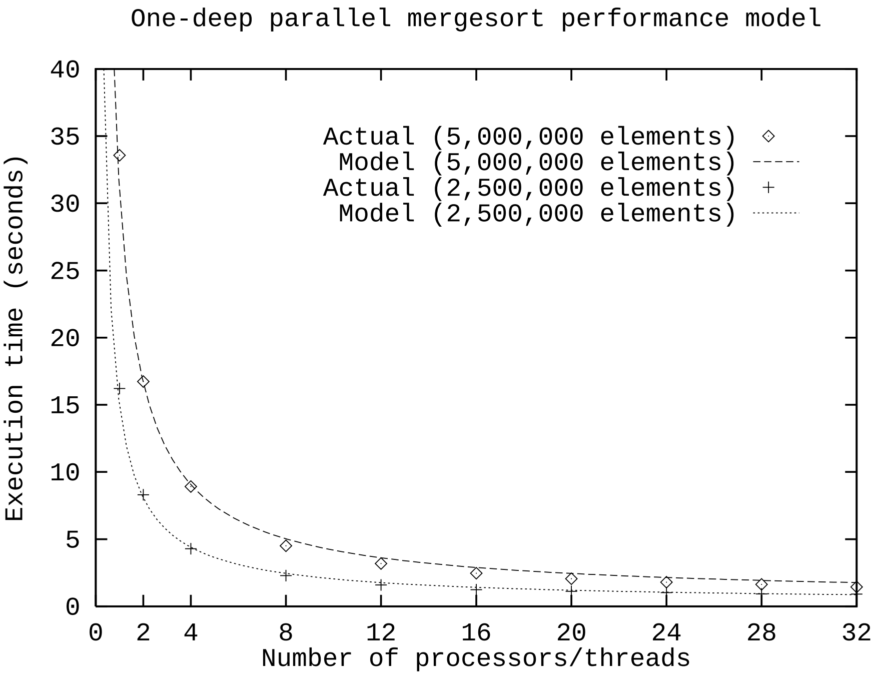
$$execution\ time\ (seconds) = C_2\frac{N}{P}log_2(\frac{N}{K}) + C_{3.6.1}\frac{N}{P}log_2(K) + C_{3.6.2a}\frac{N}{P} + C_{3.6.2n}N$$

The correlation coefficient remains at 0.9964 and the standard deviation improves insignificantly to 0.88 seconds. Figure 19 compares this performance model with the experimental performance measurements. Adding the sequential stages of the algorithm and modeling memory contention in the other parallel stages does not improve the model.



Number of processors	2,500,000 elements		5,000,000 elements	
	Actual (sec)	Model (sec)	Actual (sec)	Model (sec)
1	17.50	17.51	36.92	36.40
2	8.94	8.85	18.58	18.39
4	4.69	4.56	9.27	9.46
8	2.32	2.43	4.75	5.03
12	1.62	1.72	3.18	3.56
16	1.31	1.37	2.54	2.83
20	1.09	1.16	2.08	2.39
24	1.04	1.02	1.82	2.10
28	1.01	0.92	1.71	1.89
32	0.96	0.85	1.57	1.74

Figure 18: Comparison of one-deep parallel quicksort performance model with actual performance measurements.



Number of processors	2,500,000 elements		5,000,000 elements	
	Actual (sec)	Model (sec)	Actual (sec)	Model (sec)
1	16.22	14.97	33.57	30.95
2	8.30	8.08	16.73	16.67
4	4.29	4.41	8.92	9.06
8	2.29	2.45	4.51	5.03
12	1.59	1.77	3.19	3.62
16	1.24	1.41	2.46	2.89
20	1.12	1.20	2.05	2.45
24	1.02	1.05	1.80	2.15
28	0.93	0.95	1.63	1.93
32	0.91	0.87	1.44	1.77

Figure 19: Comparison of one-deep parallel mergesort performance model with actual performance measurements.

9.6 Analysis

The parallel sorting performance experiments show diminishing rate of speedup with increasing numbers of processors. However, it is unclear initially whether this is an effect of the inherent limitations of the algorithm or increasing contention for memory. For example, our experiments were performed on a machine with 2-way interleaved memory; would be expect to see significantly different behavior on a similar machine with 8-way interleaved memory?

Our performance models without memory contention are remarkably accurate. Therefore, it appears that memory is not a major performance bottleneck in our implementations of the parallel sorting algorithms. The primary limiting factor to speedup seems to be that the total amount of computation performed by the algorithms increases with the number of parallel threads. The algorithms are efficiently parallelizing an increasing workload.

9.7 Limitations

We have shown the performance modeling framework to be successful for our two parallel sorting algorithms and we expect it would be valuable for most other one-deep parallel divide-and-conquer algorithms. However, we are also aware of some of its potential limitations. Notably, the model does not include any concept of uneven load balancing and its does not account for the discontinuous performance degradation that can sometimes occur due to caching and paging effects. For poorly fitting models, these should be investigated as possible explanations.

10 Conclusions

One-deep parallel divide-and-conquer is a strategy for developing highly-parallel algorithms to solve the wide range of important problems that have sequential solutions based on traditional divide-and-conquer. We have demonstrated this problem-solving strategy by describing one-deep parallel quicksort and mergesort algorithms. Performance results have been presented for these algorithms executing on a shared-memory multiprocessor. We report speedups of up to 22-fold on 32 processors.

The regular structure of one-deep parallel divide-and-conquer algorithms lends itself to a systematic approach to performance modeling. We have presented a framework for developing an accurate performance model for an implementation of a one-deep parallel divide-and-conquer algorithm. The approach is based on fitting experimental measurements to stage-by-stage complexity analysis. We have used this framework to develop performance models for the parallel quicksort and mergesort algorithms. Our models have helped us gain a better understanding of the factors that influence and limit the performance of these algorithms. In the future, we plan to develop and model the performance of additional one-deep parallel divide-and-conquer algorithms for both shared-memory and distributed-memory architectures.

Acknowledgments

Thanks to the following people in the Computer Science Department at Caltech for their help with this research: Mani Chandy and Svetlana Kryukova for identifying the basic design strategy and developing a collection of algorithms; Paul Sivilotti for long discussions and collaboration in the early stages of this work; Rajit Manohar and Marcel van der Goot for their help with many details; and Berna Massingill and Adam Rifkin for their proofreading. Special thanks to the Ada 95 group of the Visual Magic Division at Silicon Graphics, Inc., in particular Tom Quiggle and Wes

Embry, for providing the software, computer time, and technical support necessary for this project. Without their assistance, this report could not have been written.

References

[1] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *International Conference of Parallel and Distributed Information Systems*, pages 280–291, December 1991.

[2] Hanmao Shi and Jonathon Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, April 1992.

[3] *Ada 95 Reference Manual.* International Organization for Standardization, January 1995. International Standard ANSI/ISO/IEC-8652:1995.

[4] John Thornley. Integrating parallel dataflow programming with the Ada tasking model. In *Proceedings of ACM TRI-Ada '94*, pages 417–428, Baltimore, Maryland, November 6–11 1994.

[5] Edmond Schonberg and Bernard Banner. The GNAT project: A GNU-Ada 9X compiler. In *Proceedings of ACM TRI-Ada '94*, pages 48–57, Baltimore, Maryland, November 6–11 1994.

[6] Richard Stallman. *Using and Porting GNU GCC.* Free Software Foundation, Cambridge, Massachusetts, 1994.

[7] E. W. Giering, Frank Mueller, and T. P. Baker. Features of the GNU Ada runtime library. In *Proceedings of ACM TRI-Ada '94*, pages 93–103, Baltimore, Maryland, November 6–11 1994.

[8] William H. Press, Saul A Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes in C.* Cambridge University Press, Cambridge, Great Britain, second edition, 1992.

[9] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[10] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.

[11] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[12] Ellis Horowitz and Alessandro Zorat. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers*, C-32(6):582–585, June 1983.

[13] Tom Axford. The divide-and-conquer paradigm as a basis for parallel language design. In Lydia Kronsjö and Dean Shumsheruddin, editors, *Advances in Parallel Algorithms*, pages 26–65. Halsted Press, New York, 1992.

[14] D. J. Evans and N. Y. Yousif. Analysis of the performance of the parallel quicksort method. *BIT*, 25:106–112, 1985.

[15] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of AFIPS*, volume 30, pages 483–485, Atlantic City, New Jersey, April 18–20 1967.

[16] K. Mani Chandy and Svetlana Kryukova. Patterns of specifications. Paper in preparation, Computer Science Department, California Institute of Technology.

[17] Mark J. Clement and Michael J. Quinn. Overlapping computations, communications and I/O in parallel sorting. *Journal of Parallel and Distributed Computing*, 28(2):162–172, August 1995.

[18] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

A Transformation of Parallel Constructs to Ada 95 Tasking

In this appendix, we give transformations from parallel block and parallel for-loop statements to equivalent Ada 95 tasking constructs. These transformations are discussed further in [4].

A.1 Parbegin-Parend

A parallel block statement:

```
parbegin
    statement_1
    ...
    statement_n
parend;
```

is equivalent to the following Ada 95 block containing a sequence of task declarations:

```
declare
    task T_1; task body T_1 is
    begin
        statement_1
    end T_1;
    ...
    task T_n; task body T_n is
    begin
        statement_n
    end T_n;
begin
    null;
end;
```

A.2 Parfor-Loop

A parallel for-loop statement:

```
parfor I in First .. Last loop
    sequence-of-statements
end loop;
```

is equivalent to the following Ada 95 block declaring an array of tasks:

```
declare
    task type Iteration (I : Integer); task body Iteration is
    begin
        sequence-of-statements
    end Iteration;
    type Iteration_Access is access Iteration;
    Iterations : array (Integer range First .. Last) of Iteration_Access;
begin
    for I in First .. Last loop
        Iterations(I) := new Iteration(I);
    end loop;
end;
```

B Program Source Code

B.1 Traditional Sequential and Parallel Quicksort

B.1.1 Traditional Quicksort Package Declaration

```
------------------------------------------------------------------------
--
-- Generic Traditional Quicksort Package Declaration
--
------------------------------------------------------------------------

generic
   type Element is private;
   type Elements is array (Integer range <>) of Element;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function "=" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package Traditional_Quicksort is

   procedure Sequential_Quicksort (
         First, Last : in     Integer;
         Data        : in out Elements;
         Base_Length : in     Positive );
   --| Requires:
   --|    First <= Last + 1 and
   --|    Data'First <= First and Last <= Data'Last and
   --|    2 <= Base_Length.
   --| Ensures:
   --|    Ascending(Data(First .. Last)) and
   --|    Permutation(Data(First .. Last), Data`(First .. Last)).

   procedure Parallel_Quicksort (
         First, Last       : in     Integer;
         Data              : in out Elements;
         Base_Length       : in     Positive;
         Sequential_Length : in     Positive );
   --| Requires:
   --|    First <= Last + 1 and
   --|    Data'First <= First and Last <= Data'Last and
   --|    2 <= Base_Length and Base_Length <= Sequential_Length.
   --| Ensures:
   --|    Ascending(Data(First .. Last)) and
   --|    Permutation(Data(First .. Last), Data`(First .. Last)).

end Traditional_Quicksort;

------------------------------------------------------------------------
```

B.1.2 Traditional Quicksort Package Definition

```
------------------------------------------------------------------------
--
-- Generic Traditional Quicksort Package Definition
--
-- Algorithm details:
--
-- * Recursive sequential and parallel quicksort algorithms.
-- * Base-case data arrays sorted using insertion sort.
-- * Median of first, middle, and last elements used as
--   pivot in partitioning.
-- * Sentinels used to reduce number of comparisons the partitioning.
-- * Source: "Numerical Recipes in C, Second Edition".
--
------------------------------------------------------------------------

with Assertions; use Assertions;

package body Traditional_Quicksort is

   ---------------------------------------------------------------------

   pragma Suppress(Index_Check);
   pragma Suppress(Range_Check);

   ---------------------------------------------------------------------

   procedure Insertion_Sort (
         First, Last : in     Integer;
         Data        : in out Elements) is
   --| Requires:
   --|    First <= Last + 1 and
   --|    Data'First <= First and Last <= Data'Last.
   --| Ensures:
   --|    Ascending(Data(First .. Last)) and
   --|    Permutation(Data(First .. Last), Data`(First .. Last)).
   begin
      Assert(First <= Last + 1);
      Assert(Data'First <= First and Last <= Data'Last);
      for I in First .. Last - 1 loop
         declare
            Temp : Element;
            Pos  : Integer range Data'Range;
         begin
            Temp := Data(I + 1);
            Pos  := First;
            for J in reverse First .. I loop
               if Temp < Data(J) then
                  Data(J + 1) := Data(J);
               else
                  Pos := J + 1;
                  exit;
               end if;
```

```
      end loop;
      Data(Pos) := Temp;
   end;
end loop;
end Insertion_Sort;

pragma Inline(Insertion_Sort);

----------------------------------------------------------------------

procedure Swap (X, Y : in out Element) is
--| Requires:
--|    True.
--| Ensures:
--|    X = Y‘ and Y = X‘.

   Temp : Element;

begin
   Temp := X;
   X := Y;
   Y := Temp;
end Swap;

pragma Inline(Swap);

----------------------------------------------------------------------

procedure Partition (
      First, Last : in     Integer;
      Data        : in out Elements;
      Pivot_Index :    out Integer  ) is
--| Requires:
--|    First <= Last + 1 and
--|    Data'First <= First and Last <= Data'Last and
--|    Last - First + 1 >= 3.
--| Ensures:
--|    First <= Pivot_Index and Pivot_Index <= Last and
--|    for all I in First .. Pivot_Index - 1 :
--|       Data(I) <= Data(Pivot_Index) and
--|    for all I in Pivot_Index + 1 .. Last :
--|       Pivot_Index <= Data(I).

   Length : constant Integer := Last - First + 1;
   Pivot_Value : Element;
   Left, Right : Integer range Data'Range;

begin
   Assert(First < Last + 1);
   Assert(Data'First <= First and Last <= Data'Last);
   Assert(Length >= 3);
   Swap(Data(First + 1), Data((First + Last)/2));
   if Data(First + 1) > Data(Last) then
      Swap(Data(First + 1), Data(Last));
```

```
   end if;
   if Data(First) > Data(Last) then
      Swap(Data(First), Data(Last));
   end if;
   if Data(First + 1) > Data(First) then
      Swap(Data(First + 1), Data(First));
   end if;
   Pivot_Value := Data(First);
   Left := First + 1; Right := Last;
   loop
      Left := Left + 1;
      while Data(Left) < Pivot_Value loop
         Left := Left + 1;
      end loop;
      Right := Right - 1;
      while Data(Right) > Pivot_Value loop
         Right := Right - 1;
      end loop;
      exit when Right < Left;
      Swap(Data(Left), Data(Right));
   end loop;
   Data(First) := Data(Right);
   Data(Right) := Pivot_Value;
   Pivot_Index := Right;
   Assert(First <= Pivot_Index and Pivot_Index <= Last);
end Partition;

pragma Inline(Partition);

----------------------------------------------------------------------

procedure Sequential_Quicksort (
      First, Last : in     Integer;
      Data        : in out Elements;
      Base_Length : in     Positive ) is

   Length : constant Integer := Last - First + 1;

begin
   Assert(First <= Last + 1);
   Assert(Data'First <= First and Last <= Data'Last);
   Assert(2 <= Base_Length);
   if Length <= Base_Length then
      Insertion_Sort(First, Last, Data);
   else
      declare
         Pivot_Index : Integer range Data'Range;
      begin
         Partition(First, Last, Data, Pivot_Index);
         Sequential_Quicksort(
            First, Pivot_Index - 1, Data, Base_Length);
         Sequential_Quicksort(
            Pivot_Index + 1, Last, Data, Base_Length);
      end;
```

```
      end if;
   end Sequential_Quicksort;

   ----------------------------------------------------------------------

   procedure Parallel_Quicksort (
         First, Last       : in     Integer;
         Data              : in out Elements;
         Base_Length       : in     Positive;
         Sequential_Length : in     Positive ) is

      Length : constant Integer := Last - First + 1;

   begin
      Assert(First <= Last + 1);
      Assert(Data'First <= First and Last <= Data'Last);
      Assert(2 <= Base_Length and Base_Length <= Sequential_Length);
      if Length <= Sequential_Length then
         Sequential_Quicksort(First, Last, Data, Base_Length);
      else
         declare
            Pivot_Index : Integer range Data'Range;
         begin
            Partition(First, Last, Data, Pivot_Index);
            begin
               pragma Parallelizable_Sequence;
               Parallel_Quicksort(First, Pivot_Index - 1,
                  Data, Base_Length, Sequential_Length);
               Parallel_Quicksort(Pivot_Index + 1, Last,
                  Data, Base_Length, Sequential_Length);
            end;
         end;
      end if;
   end Parallel_Quicksort;

   ----------------------------------------------------------------------

end Traditional_Quicksort;

-------------------------------------------------------------------------
```

B.2 One-Deep Parallel Quicksort

B.2.1 One-Deep Parallel Quicksort Package Declaration

```
-------------------------------------------------------------------------
--
-- Generic One-Deep Parallel Quicksort Package Declaration
--
-------------------------------------------------------------------------

with Ada.Calendar; use Ada.Calendar;

generic
   type Element is private;
   type Elements is array (Integer range <>) of Element;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function "=" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package One_Deep_Parallel_Quicksort is

   procedure Parallel_Quicksort (
         First, Last : in     Integer;
         Data        : in     Elements;
         Result      :    out Elements;
         Num_Threads : in     Positive );
   --| Requires:
   --|    Data'First <= First and Last <= Data'Last and
   --|    Result'First <= First and Last <= Result'Last and
   --|    Num_Threads <= Last - First + 1.
   --| Ensures:
   --|    Ascending(Result(First .. Last)) and
   --|    Permutation(Result(First .. Last), Data`(First .. Last)).

end One_Deep_Parallel_Quicksort;

-------------------------------------------------------------------------
```

B.2.2 One-Deep Parallel Quicksort Package Definition

```
--------------------------------------------------------------------------
--
-- Generic One-Deep Parallel Quicksort Package Definition
--
-- Algorithm:
--
-- 1.1 Take sample of elements from Data.
-- 1.2 Sequentially sort sample.
-- 1.3 Choose evenly-spaced pivot elements from sorted sample.
-- 1.4 Trivially, divide Data into equal-sized segments.
-- 1.5 In parallel, find Result partitions of Data elements
--     and count elements per partition.
-- 1.6 Compute boundaries of Result partitions.
-- 1.7 Compute positions of Data elements in Result.
-- 1.8 In parallel, copy Data elements to Result.
-- 2   In parallel, sequentially sort Result partitions.
--
--------------------------------------------------------------------------

with Ada.Calendar; use Ada.Calendar;
with Assertions; use Assertions;
with Traditional_Quicksort;

package body One_Deep_Parallel_Quicksort is

   -----------------------------------------------------------------------

   pragma Suppress(Index_Check);
   pragma Suppress(Range_Check);

   -----------------------------------------------------------------------

   package Element_Traditional_Quicksort is
      new Traditional_Quicksort (Element, Elements);
   use Element_Traditional_Quicksort;

   -----------------------------------------------------------------------

   Base_Length : constant Positive := 16;

   -----------------------------------------------------------------------

   function Divide (
         First, Last : Integer;
         I : Natural; N : Positive) return Integer is
   --| Requires
   --|    First <= Last and I <= N.
   --| Ensures:
   --|    Divide = First +
   --|       Integer(Float(Last - First + 1)*(Float(I)/Float(N)));
   begin
      Assert(First <= Last and I <= N);
```

```
   return First +
      Integer(Float(Last - First + 1)*(Float(I)/Float(N)));
end Divide;

pragma Inline(Divide);

----------------------------------------------------------------------

function Find (
      First, Last : Integer;
      Data        : Elements;
      Target      : Element  ) return Integer is
--| Requires:
--|    First <= Last + 1 and
--|    Data'First <= First and Last <= Data'Last and
--|    Ascending(Data(First .. Last)).
--| Ensures:
--|    (Find = First - 1 or else Data(Find) <= Target) and
--|    (Find = Last or else Target < Data(Find + 1)).

   F, L, Mid : Integer range First - 1 .. Last + 1;

begin
   Assert(First <= Last + 1);
   Assert(Data'First <= First and Last <= Data'Last);
   F := First; L := Last;
   while F <= L loop
      Mid := (L - F)/2 + F;
      if Data(Mid) <= Target then
         F := Mid + 1;
      else
         L := Mid - 1;
      end if;
   end loop;
   Assert(F = L + 1);
   Assert(L = First - 1 or else Data(L) <= Target);
   Assert(L = Last or else Target < Data(L + 1));
   return L;
end Find;

pragma Inline(Find);

----------------------------------------------------------------------

procedure Parallel_Quicksort (
      First, Last : in     Integer;
      Data        : in     Elements;
      Result      :    out Elements;
      Num_Threads : in     Positive ) is

   Sample_Length : constant Positive := Integer'Max(Num_Threads,
      (Last - First + 1)/((Num_Threads + 9)*(Num_Threads + 9)));

   S  : Elements (0 .. Sample_Length - 1);
```

```
   P  : Elements (1 .. Num_Threads - 1);
   DS : array (0 .. Num_Threads) of
         Integer range First .. Last + 1;
   EP : array (Integer range Data'Range) of
          Integer range 0 .. Num_Threads - 1;
   PC : array (0 .. Num_Threads*Num_Threads - 1) of Natural;
   RP : array (0 .. Num_Threads) of
          Integer range First .. Last + 1;
   CP : array (0 .. Num_Threads*Num_Threads - 1) of
          Integer range First .. Last + 1;

-- S  = Sample.
-- P  = Pivots.
-- DS = Data Segments.
-- EP = Element Partitions.
-- PC = Partition Counts.
-- RP = Result Partitions.
-- CP = Copy Positions.

   Start, Finish : Time;

begin
   Assert(Data'First <= First and Last <= Data'Last);
   Assert(Result'First <= First and Last <= Result'Last);
   Assert(Num_Threads <= Last - First + 1);

   for I in S'Range loop
      S(I) := Data(Divide(First, Last, I, Sample_Length));
   end loop;

   Sequential_Quicksort(S'First, S'Last, S, Base_Length);

   for I in 1 .. Num_Threads - 1 loop
      P(I) := S(Divide(S'First, S'Last, I, Num_Threads));
   end loop;

   for I in 0 .. Num_Threads loop
      DS(I) := Divide(First, Last, I, Num_Threads);
   end loop;

   pragma Parallelizable_Loop;
   for I in 0 .. Num_Threads - 1 loop
      declare
         Count : array (0 .. Num_Threads - 1) of Natural;
      begin
         for J in Count'Range loop
            Count(J) := 0;
         end loop;
         for J in DS(I) .. DS(I + 1) - 1 loop
            EP(J) := Find(1, Num_Threads - 1, P, Data(J));
            Count(EP(J)) := Count(EP(J)) + 1;
         end loop;
         for J in Count'Range loop
            PC(I*Num_Threads + J) := Count(J);
```

```
         end loop;
      end;
   end loop;

   RP(0) := First;
   for I in 1 .. Num_Threads - 1 loop
      RP(I) := RP(I - 1);
      for J in 0 .. Num_Threads - 1 loop
         RP(I) := RP(I) + PC(J*Num_Threads + I - 1);
      end loop;
   end loop;
   RP(Num_Threads) := Last + 1;

   for I in 0 .. Num_Threads - 1 loop
      CP(0*Num_Threads + I) := RP(I);
      for J in 1 .. Num_Threads - 1 loop
         CP(J*Num_Threads + I) := CP((J - 1)*Num_Threads + I) +
            PC((J - 1)*Num_Threads + I);
      end loop;
   end loop;

   pragma Parallelizable_Loop;
   for I in 0 .. Num_Threads - 1 loop
      declare
         Position : array (0 .. Num_Threads - 1) of
                          Integer range First .. Last + 1;
      begin
         for J in Position'Range loop
            Position(J) := CP(I*Num_Threads + J);
         end loop;
         for J in DS(I) .. DS(I + 1) - 1 loop
            Result(Position(EP(J))) := Data(J);
            Position(EP(J)) := Position(EP(J)) + 1;
         end loop;
      end;
   end loop;

   pragma Parallelizable_Loop;
   for I in 0 .. Num_Threads - 1 loop
      Sequential_Quicksort(
         RP(I), RP(I + 1) - 1, Result, Base_Length);
   end loop;
end Parallel_Quicksort;

end One_Deep_Parallel_Quicksort;

------------------------------------------------------------------------
```

B.3 One-Deep Parallel Mergesort

B.3.1 One-Deep Parallel Mergesort Package Declaration

```
------------------------------------------------------------------------
--
-- Generic One-Deep Parallel Mergesort Package Declaration
--
------------------------------------------------------------------------

with Ada.Calendar; use Ada.Calendar;

generic
   type Element is private;
   type Elements is array (Integer range <>) of Element;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function "=" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package One_Deep_Parallel_Mergesort is

   procedure Parallel_Mergesort (
         First, Last : in     Integer;
         Data        : in out Elements;
         Result      :    out Elements;
         Num_Threads : in     Positive );
   --| Requires:
   --|    Data'First <= First and Last <= Data'Last and
   --|    Result'First <= First and Last <= Result'Last and
   --|    2*Num_Threads*Num_Threads <= Last - First + 1 and
   --| Ensures:
   --|    Ascending(Result(First .. Last)) and
   --|    Permutation(Result(First .. Last), Data`(First .. Last)).

end One_Deep_Parallel_Mergesort;

------------------------------------------------------------------------
```

B.3.2 One-Deep Parallel Mergesort Package Definition

```
------------------------------------------------------------------------
--
-- Generic One-Deep Parallel Mergesort Package Definition
--
-- Algorithm:
--
-- 1   Trivially, divide Data into equal-sized segments.
-- 2   In parallel, sequentially sort Data segments.
-- 3.1 Choose evenly-spaced local pivot elements from Data segments.
-- 3.2 Merge local pivot elements into one ordered list.
-- 3.3 Choose evenly-spaced global pivot elements from
--     local pivot elements.
-- 3.4 Find boundaries of Data sub-segments between
--     global pivot elements.
-- 3.5 Compute boundaries of Result segments.
-- 3.6 In parallel, merge Data sub-segments into Result segments.
--
------------------------------------------------------------------------

with Ada.Calendar; use Ada.Calendar;
with Assertions; use Assertions;
with Traditional_Quicksort;

package body One_Deep_Mergesort is

   ---------------------------------------------------------------------

   pragma Suppress(Index_Check);
   pragma Suppress(Range_Check);

   ---------------------------------------------------------------------

   package Element_Traditional_Quicksort is
      new Traditional_Quicksort (Element, Elements);
   use Element_Traditional_Quicksort;

   ---------------------------------------------------------------------

   Base_Length : constant Positive := 16;

   ---------------------------------------------------------------------

   function Divide (
         First, Last : Integer;
         I : Natural; N : Positive) return Integer is
   --| Requires
   --|    First <= Last and I <= N.
   --| Ensures:
   --|    Divide = First +
   --|       Integer(Float(Last - First + 1)*(Float(I)/Float(N)));
   begin
      Assert(First <= Last and I <= N);
```

```
   return First +
      Integer(Float(Last - First + 1)*(Float(I)/Float(N)));
end Divide;

pragma Inline(Divide);

----------------------------------------------------------------------

function Find (
      First, Last : Integer;
      Data        : Elements;
      Target      : Element  ) return Integer is
--| Requires:
--|    First <= Last + 1 and
--|    Data'First <= First and Last <= Data'Last and
--|    Ascending(Data(First .. Last)).
--| Ensures:
--|    (Find = First or else Data(Find - 1) < Target) and
--|    (Find = Last + 1 or else Target <= Data(Find)).

   F, L, Mid : Integer range First - 1 .. Last + 1;

begin
   Assert(First <= Last + 1);
   Assert(Data'First <= First and Last <= Data'Last);
   F := First; L := Last;
   while F <= L loop
      Mid := (L - F)/2 + F;
      if Data(Mid) < Target then
         F := Mid + 1;
      else
         L := Mid - 1;
      end if;
   end loop;
   Assert(F = L + 1);
   Assert(F = First or else Data(F - 1) < Target);
   Assert(F = Last + 1 or else Target <= Data(F));
   return F;
end Find;

pragma Inline(Find);

----------------------------------------------------------------------

type Segment is
   record
      First, Last : Integer;
   end record;
type Segments is array (Integer range <>) of Segment;

procedure Merge (
      N           : in     Positive;
      S           : in     Segments;
      Data        : in     Elements;
```

```
      First, Last : in     Integer;
      Merged      :    out Elements ) is
--| Requires:
--|    S'First = 0 and S'Last = N - 1 and
--|    for all I in 0 .. N - 1 :
--|      (Data'First <= S(I).First and S(I).Last <= Data'Last and
--|       Ascending(Data(S(I).First .. S(I).Last))) and
--|    Last - First + 1 =
--|       Sum(I in 0 .. N - 1 : S(I).Last - S(I).First + 1) and
--|    Merged'First <= First and Last <= Merged'Last.
--| Ensures:
--|    Ascending(Merged(First .. Last)) and
--|    Permutation(Merged(First .. Last),
--|       Join(I in 0 .. N - 1 : Data(S(I).First .. S(I).Last))).

   Max_Index : Integer range Data'Range;

   Next : array (Integer range 0 .. N - 1) of
             Integer range Data'First .. Data'Last;
   Heap : array (Integer range 0 .. N - 1) of
             Integer range 0 .. N - 1;

begin
   Assert(S'First = 0 and S'Last = N - 1);
   Assert(First <= Last + 1);
   Assert(Merged'First <= First and Last <= Merged'Last);
   if First <= Last then
      declare
         Max_Element : Element;
         Found       : Boolean;
      begin
         Found := False;
         for I in 0 .. N - 1 loop
            if S(I).Last >= S(I).First and then (not Found
                  or else Data(S(I).Last) > Max_Element) then
               Max_Index := S(I).Last;
               Max_Element := Data(Max_Index);
               Found := True;
            end if;
         end loop;
      end;
      for I in 0 .. N - 1 loop
         declare
            Pos : Integer range 0 .. N - 1;
         begin
            if S(I).First <= S(I).Last then
               Next(I) := S(I).First;
            else
               Next(I) := Max_Index;
            end if;
            Pos := 0;
            for J in reverse 0 .. I - 1 loop
               if Data(Next(Heap(J))) > Data(Next(I)) then
                  Heap(J + 1) := Heap(J);
```

```
            else
               Pos := J + 1;
               exit;
            end if;
         end loop;
         Heap(Pos) := I;
      end;
   end loop;
   for I in First .. Last loop
      declare
         Top   : Integer range 0 .. N - 1;
         Child : Integer range 0 .. 2*N;
      begin
         Top := Heap(0);
         Merged(I) := Data(Next(Top));
         if Next(Top) = S(Top).Last then
            Next(Top) := Max_Index;
         elsif Next(Top) /= Max_Index then
            Next(Top) := Next(Top) + 1;
         end if;
         Child := 1;
         while Child < N loop
            if Child < N - 1 and then
                  Data(Next(Heap(Child + 1)))
                  < Data(Next(Heap(Child))) then
               Child := Child + 1;
            end if;
            exit when Data(Next(Top)) <= Data(Next(Heap(Child)));
            Heap((Child - 1)/2) := Heap(Child);
            Child := 2*Child + 1;
         end loop;
         Heap((Child - 1)/2) := Top;
      end;
   end loop;
end if;
end Merge;

pragma Inline(Merge);

---------------------------------------------------------------------

procedure Parallel_Mergesort (
      First, Last : in     Integer;
      Data        : in out Elements;
      Result      :    out Elements;
      Num_Threads : in     Positive ) is

   DS : array (0 .. Num_Threads) of
          Integer range First .. Last + 1;
   LP : Elements (0 .. 2*Num_Threads*Num_Threads - 1);
   PS : Segments (0 .. Num_Threads - 1);
   MP : Elements (0 .. 2*Num_Threads*Num_Threads - 1);
   GP : Elements (0 .. Num_Threads - 1);
   SS : array (0 .. Num_Threads - 1) of
```

```
         Segments (0 .. Num_Threads - 1);
   RS : array (0 .. Num_Threads) of
         Integer range First .. Last + 1;

-- DS = Data Segments.
-- LP = Local Pivots.
-- PS = Pivot Segments.
-- MP = Merged Pivots.
-- GP = Global Pivots.
-- SS = Sub-Segments.
-- RS = Result Segments.

   Start, Finish : Time;

begin
   Assert(Data'First <= First and Last <= Data'Last);
   Assert(Result'First <= First and Last <= Result'Last);
   Assert(2*Num_Threads*Num_Threads <= Last - First + 1);

   for I in 0 .. Num_Threads loop
      DS(I) := Divide(First, Last, I, Num_Threads);
   end loop;

   pragma Parallelizable_Loop;
   for I in 0 .. Num_Threads - 1 loop
      Sequential_Quicksort(DS(I), DS(I + 1) - 1, Data, Base_Length);
   end loop;

   for I in 0 .. Num_Threads - 1 loop
      PS(I).First := 2*I*Num_Threads;
      PS(I).Last := PS(I).First + 2*Num_Threads - 1;
      LP(PS(I).First) := Data(DS(I));
      for J in 1 .. Num_Threads - 1 loop
         declare
            P : Integer range First .. Last;
         begin
            P := Divide(DS(I), DS(I + 1) - 1, J, Num_Threads);
            LP(PS(I).First + 2*J) := Data(P);
            LP(PS(I).First + 2*J - 1) := Data(P - 1);
         end;
      end loop;
      LP(PS(I).Last) := Data(DS(I + 1) - 1);
   end loop;

   Merge(Num_Threads, PS, LP, MP'First, MP'Last, MP);

   for I in 0 .. Num_Threads - 1 loop
      GP(I) := MP(Divide(MP'First, MP'Last, I, Num_Threads));
   end loop;

   for I in 0 .. Num_Threads - 1 loop
      SS(0)(I).First := DS(I);
      for J in 1 .. Num_Threads - 1 loop
         SS(J)(I).First := Find(DS(I), DS(I + 1) - 1, Data, GP(J));
```

```
            SS(J - 1)(I).Last := SS(J)(I).First - 1;
         end loop;
         SS(Num_Threads - 1)(I).Last := DS(I + 1) - 1;
      end loop;

      RS(0) := First;
      for I in 1 .. Num_Threads - 1 loop
         RS(I) := RS(I - 1);
         for J in 0 .. Num_Threads - 1 loop
            RS(I) := RS(I) + SS(I - 1)(J).Last - SS(I - 1)(J).First + 1;
         end loop;
      end loop;
      RS(Num_Threads) := Last + 1;

      pragma Parallelizable_Loop;
      for I in 0 .. Num_Threads - 1 loop
         Merge(Num_Threads, SS(I), Data, RS(I), RS(I + 1) - 1, Result);
      end loop;
   end Parallel_Mergesort;

   ----------------------------------------------------------------------

end One_Deep_Parallel_Mergesort;

-------------------------------------------------------------------------
```