ON PARALLEL DIVIDE-AND-CONQUER*

Liane Acker and Daniel Miranker

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-27 August 1991

* This research is supported by an IBM Graduate Fellowship.

On Parallel Divide-and-Conquer[1]

Liane Acker **Daniel Miranker**

acker@cs.utexas.edu **miranker@cs.utexas.edu**

Department of Computer Sciences
University of Texas
Austin, TX 78712

Abstract

Divide-and-conquer is a powerful problem-solving technique that is the basis for many effective sequential algorithms. We analyze the extent to which divide-and-conquer yields effective and efficient parallel algorithms. We identify fifteen equivalence classes of divide-and-conquer algorithms and determine which classes are good candidates for parallelization and the architectures for which they are best suited. None of the classes provide optimal speedup when the maximum possible number of processors are used, and four of them yield no more than constant speedup. However, eleven classes do provide optimal speedup under limited parallelism, and three of these classes have polylogarithmic runtime with a polynomial number of processors (that is, they are in NC) and they have speedup that is within a polylogarithmic factor of optimal under maximum parallelism. The communication cost incurred during parallelization is found to have a significant impact on the performance of a parallel divide-and-conquer algorithm. This factor alone can mean the difference between speedup that is within a polylogarithmic factor of optimal and no speedup at all.

1 Introduction

Divide-and-conquer is an established technique for designing effective algorithms for uniprocessors. The basic divide-and-conquer scheme is as follows:

> *Given a problem, divide it into independent subproblems of the same type. Once these subproblems have been solved (using the divide-and-conquer scheme recursively), combine their solutions into a solution to the original problem.*

[1]This research is supported by an IBM Graduate Fellowship.

For example, a divide-and-conquer approach to summing a list of numbers is to halve the list, sum the halves, then add the two results. Another example is the mergesort algorithm, in which a list to be sorted is halved, the halves sorted, and the results merged. In [AHU83], divide-and-conquer is called "perhaps the most important, and most widely applicable, technique for designing efficient algorithms."

Sequential divide-and-conquer algorithms can be partitioned into equivalence classes based on three factors:

1. The number of subproblems, a, generated by a problem division (the *branching factor*)
2. The size of a subproblem (assumed the same for all subproblems created at a single divide step) as a function of N, the size of the original problem
3. The time required for a single problem division and the combination of the results of solved subproblems (the *divide-and-conquer cost*, or *dac-cost*)

From a theoretical point of view, divide-and-conquer is an attractive algorithm abstraction because it allows complexity analysis to be performed on entire groups of algorithms. Algorithms falling within the same equivalence class based on the above partitioning have the same computational complexity. It is computed by solving the following recurrence equation:

$$T(N) = \text{branchingFactor}(N) * T(\text{subproblemSize}(N)) + \text{dacCost}(N)$$

Examples may be found in [AHU83], [AHU74], and [Sedgewick83].

2 Parallel Divide-and-Conquer

Because the divide-and-conquer technique is based on solving independent subproblems, a divide-and-conquer algorithm has a high degree of inherent parallelism. In particular, the independence of subproblems means that if each subproblem constitutes a separate process, interprocess communication is limited to communication between a process and the processes it spawns. In addition, this communication occurs only at the beginning and at the end of the child processes' lifetimes. Thus, it seems that divide-and-conquer algorithms would make efficient use of multiprocessor systems. [Stout87] gives several additional reasons why the divide-and-conquer approach should be particularly applicable to designing parallel algorithms.

As with the sequential divide-and-conquer scheme, parallel divide-and-conquer is a useful algorithm abstraction because it permits analysis of entire classes of algorithms simultaneously. In a parallel environment, we must consider *communication cost*, the time required to transfer subproblems or results from one processor to another, in addition to branching factor, number of subproblems, and divide-and-conquer cost, when performing algorithm analysis. Based on this partitioning, the computational complexity of a parallel divide-and-conquer algorithm is specified by the following recurrence, assuming that all subproblems of the same size are solved in parallel:

$$T(N) = T(\text{subproblemSize}(N)) + \text{dacCost}(N) + \text{communicationCost}(N).$$

In this paper we analyze the complexity, speedup, and efficiency of fifteen classes of divide-and-conquer algorithms to determine which ones are good candidates for parallelization and the architectures for which they are best suited. (A similar analysis is given in [Zorat79] and [HZ83]; however, they assume a constant number of subproblems and constant communication cost.)

The models of computation covered by this analysis include the PRAM shared-memory model and the distributed-memory (or message-passing) model. For the PRAM model, communication introduces only a constant delay. For the distributed-memory model, communication cost may introduce an $\Theta(N)$ delay. The next section examines each of these models and their impact on communication cost in more detail. Section 4 discusses the other three factors affecting the performance of a parallel divide-and-conquer algorithm: branching factor, number of subproblems, and divide-and-conquer cost. Sections 5 and 6 present our analysis of parallel divide-and-conquer algorithms based on these factors, and in Section 7 we discuss our conclusions.

3 Communication Cost with Respect to Machine Model

As [Cvetanovich87] shows, communication cost can drastically affect the performance of a parallel algorithm, even one with a large degree of inherent parallelism. In some cases, communication cost can cause a parallel algorithm to perform worse than the same algorithm running on a uniprocessor. Unlike the other characteristics of parallel divide-and-conquer algorithms, communication cost depends on both the nature of the algorithm and the architecture of the machine on which it is executed.

Let us first consider shared-memory architectures, and in particular those described by the PRAM model. A PRAM (Parallel Random Access Machine) is composed of several independent

processors, each with private memory, and a shared memory which any processor can access in unit time [KR90]. The PRAM model has three variants: exclusive-read/exclusive-write (EREW), concurrent-read/exclusive-write (CREW), and concurrent-read/concurrent-write (CRCW). For divide-and-conquer algorithms, we assume that the independence of subproblems avoids write conflicts; thus, the most powerful PRAM model (CRCW) is not required. (If this assumption does not hold, resolving write conflicts may take $\Theta(\log P)$ time with P processors [KR90].)

For the PRAM model, communication cost for a parallel divide-and-conquer algorithms is $\Theta(1)$. That is, the time to transfer a subproblem from one processor to another does not depend on the size of the subproblem, because only a pointer to a block of data in memory needs to be passed, rather than an actual data block. Under the PRAM model, communication cost does not affect the asymptotic execution time or speedup.

Let us now consider distributed-memory, or message-passing, architectures, in which each processor has a local memory that is not accessible to the other processors.[2] For these machines, communication cost is either linear ($\Theta(N)$) or constant ($\Theta(1)$), depending on the I/O characteristics of the problem. Typically, communication cost is linear, because the time to transfer a subproblem of size N from one processor to another is $\Theta(N)$ for distributed-memory machines. However, certain application problems avoid this cost by allowing the data to be preallocated to processors. In other words, the "divide" phase of the divide-and-conquer algorithm is omitted, eliminating the need to transfer subproblems. This strategy is only possible for algorithms in which the "divide" step consists of simple partitioning (those with $\Theta(1)$ dac-cost). In addition, preallocation of data to processors is only advantageous for algorithms in which the size of a result is dominated asymptotically by the size of the subproblem from which it is derived. (Otherwise, the cost of returning a result to a parent process is the dominating factor of communication cost, and preallocation yields only a constant savings.) Fortunately, this is typically the case for algorithms having $\Theta(1)$ dac-cost.

An example algorithm for which preallocation is beneficial is the divide-and-conquer algorithm to sum N numbers. If every processor initially holds two of the numbers, then the sum can be computed in $\Theta(\log N)$ time; otherwise, the linear communication cost increases the computation time to $\Theta(N)$, asymptotically no better than the sequential algorithm (and possibly even worse, in terms of actual execution times). For the mergesort algorithm, communication cost on a distributed-memory machine is linear regardless of the initial distribution of data, because the result of a

[2]Thus, this analysis does not include distributed-memory machines emulating shared-memory machines, and we assume that at the end of a parallel computation the result is stored in its entirety in some processor's local memory.

subproblem of size N is also of size N, and this result must be transferred from one processor to another, taking $\Theta(N)$ time. However, because mergesort also has linear divide-and-conquer cost (when the merge is done sequentially), the linear communication cost has only a constant effect on runtime and speedup. In general, divide-and-conquer algorithms having constant divide-and-conquer cost can use preallocation of data to avoid linear communication cost, and algorithms having linear dac-cost are not severely affected by it.

To summarize, we can characterize communication cost for both the PRAM and distributed-memory models by a communication cost function that is either constant ($\Theta(1)$) or linear ($\Theta(N)$).

4 Branching Factor, Subproblem Size, and Divide-and-Conquer Cost

We denote the *branching factor* of a divide-and-conquer algorithm by a. Several well-known divide-and-conquer algorithms (towers of Hanoi[AHU83], integer multiplication[AHU83], polynomial multiplication[Sedgewick83], quicksort, and mergesort) have branching factor of only two. Strassen's matrix multiplication algorithm[Strassen69] has $a = 7$, and the matrix multiplication algorithm given in [HZ83] has $a = 8$. Our analysis includes algorithms having constant branching factors as well as $a = \sqrt{N}$.

We assume that *subproblem size* is the same for all subproblems created at a particular "divide" step. (Thus, our analysis does not include algorithms like those given in [Bentley80] for solving problems in multidimensional space.) In this analysis we consider three classes of subproblem size: $\frac{N}{k}$ for constant k (where k is not required to be the same as a, the branching factor), $N - 1$ (as in the towers of Hanoi algorithm), and $\sqrt{N}$ (as suggested in [AHU74] and as in the merging algorithm in [KR90]). For subproblem size $\sqrt{N}$ we assume that $a = \sqrt{N}$. The most common subproblem size for divide-and-conquer algorithms is $\frac{N}{2}$.

The *divide-and-conquer cost*, or *dac-cost*, of an algorithm is the sum of the costs of dividing a single problem of size N into subproblems and combining the results of solved subproblems. We consider costs that are constant ($\Theta(1)$), linear ($\Theta(N)$), $\Theta(N \log N)$, and $\Theta(N^2)$. For algorithms with constant divide-and-conquer cost, problem division consists of simple partitioning. This is the case for the problem of adding two N-bit numbers and for compacting a sparse array. More common are algorithms with linear dac-cost. For example, the quicksort algorithm divides its input into two portions, one containing all data elements less than a specific element and the

other containing all data elements greater than that element. The division takes linear time. The mergesort algorithm also has linear dac-cost, but in this case it is incurred during the "conquer" phase rather than during the "divide" phase: each half of the input is sorted separately, then the sorted lists are merged. Algorithms having $\Theta(N \log N)$ dac-cost would include any algorithm requiring that the data elements be sorted prior to each division or after each merging of results. (For some algorithms this cost can be avoided using a technique called "presorting", given in [Bentley80]. In addition, the dac-cost of many algorithms can be reduced by performing problem decomposition or result combination in parallel. For example, the overhead of mergesort is reduced from $\Theta(N)$ to $\log \log N$ by using the efficient parallel merging algorithm given in [KR90] rather than a sequential merge.) Algorithms having $\Theta(N^2)$ dac-cost include the matrix multiplication algorithms presented in [Strassen69] and [HZ83].

In the remainder of this paper we evaluate the complexity, speedup, and efficiency of the equivalence classes of divide-and-conquer algorithms as determined by branching factor, subproblem size, dac-cost, and communication cost. In this way we can determine the extent to which the divide-and-conquer approach can realistically provide effective and efficient parallel algorithms.

5 Complexity and Speedup Analysis

In a *dynamically parallel* divide-and-conquer algorithm (also termed a *partitioning* algorithm in [MS91]), each new subproblem that is created becomes a separate process. The computation can be viewed as a tree of processes, where only one level of the tree is active at any time and processes at the same level can be solved in parallel. Activity begins at the root process, with activity moving down the tree to the leaves as the original problem is successively divided into subproblems. Then, after the indivisible subproblems have been solved, activity moves back up the tree as results are returned. The execution time can be viewed as the time required to take any path from the root to a leaf then back to the root. (Our analysis does not cover algorithms with multiple divide/conquer phases, such as those presented in [Stout87].) With an unlimited number of processors, the execution time for a problem of size N is

$$w + \sum_{i=1}^{\text{height}(N)} (\text{dac and overhead cost for a subproblem at level } (i-1))$$

where height(N) is the height of the tree of processes, and w is the constant time required to solve a single indivisible subproblem.

Height(N) depends on the subproblem size created at each problem division. If the subproblem size is $\frac{N}{k}$, then height(N) = $\Theta(\log_k N)$. If the subproblem size is $N-1$, then height(N) = $\Theta(N)$. If the subproblem size is $\sqrt{N}$, then height(N) = $\Theta(\log \log N)$.

As an example, consider algorithms having constant dac-cost, constant communication cost, and subproblem size $\frac{N}{k}$. Execution time is $w + \sum_{i=1}^{\log_k N} c$ for some constant c, which is $\Theta(\log N)$. For algorithms having linear divide-and-conquer or communication cost and subproblem size $\frac{N}{k}$, execution time is $w + \sum_{i=1}^{\log_k N} \frac{cN}{k^{i-1}}$, or $w + cN \sum_{i=1}^{\log_k N} \frac{1}{k^{i-1}}$, which is $\Theta(N)$.

Because only one level of the tree of processes is active at any time, the maximum number of processors that can be used concurrently, P_{max}, is equal to the number of leaves on the tree. [LRGKMT90] show that this is also the maximum number of processors required over the entire computation for certain efficient mappings of divide-and-conquer algorithms to parallel architectures. For algorithms that create a subproblems of size $\frac{N}{k}$, the number of leaves on the tree (and hence the maximum number of processors needed) is $a^{\log_k N}$ (or $N^{\log_k a}$). For algorithms that create a subproblems of size $N-1$, the maximum number of processors is a^N. ([GR88] notes that this many processors is probably not feasible.) For algorithms that create $\frac{N}{f(N)}$ subproblems of size $f(N)$, the maximum number of processors is N.

We now turn to the question of how much faster a parallel divide-and-conquer algorithm is than the same algorithm executed serially. In other words, how much speedup can be achieved through parallelism? We use the conventional definition of speedup: $\frac{serial-execution-time}{parallel-execution-time}$.

Table 1 gives asymptotic execution times and speedups for the classes of divide-and-conquer algorithms previously described, assuming that the maximum possible number of processors (P_{max}) are used (one per indivisible subproblem).

From Table 1 we see that none of these classes of divide-and-conquer algorithms provide optimal ($\Theta(P)$) speedup when the maximum possible number of processors are used.[3] However, the algorithms having $\Theta(1)$ dac-cost and $\Theta(1)$ communication cost, as well as the algorithms having $\Theta(1)$ dac-cost and problem size $N-1$, do provide speedup that is within a polylogarithmic factor of optimal when the maximum possible number of processors are used. In addition, algorithms having $\Theta(1)$ dac-cost, $\Theta(1)$ communication cost, and subproblem size $\frac{N}{k}$ or $\sqrt{N}$ have polylogarithmic runtime with a polynomial number of processors, which makes them *efficient* algorithms by

[3]The notion of optimal *speedup* should be distinguished from the notion of an optimal *algorithm*. Definitions of the latter in [GR88] and [KR90] refer to the fastest sequential algorithm, which may differ from a sequential version of the parallel algorithm. It is not possible in general to decide the optimality of an entire class of algorithms because the complexity of the fastest sequential algorithm may vary within the group.

DaC Cost	*Communication Cost*	*Subproblem Size*	*Number of Subproblems*	P_{max} *Processors* (P)	*Runtime With* P_{max}	*Speedup With* P_{max}
$\Theta(1)$	$\Theta(1)$	$\frac{N}{k}$	$a = k$	$\Theta(N)$	$\Theta(\log N)$	$\Theta(\frac{P}{\log P})$
$\Theta(1)$	$\Theta(1)$	$\frac{N}{k}$	$a \neq k$	$\Theta(N^{\log_k a})$	$\Theta(\log N)$	$\Theta(\frac{P}{\log P})$
$\Theta(1)$	$\Theta(1)$	$\sqrt{N}$	$\sqrt{N}$	$\Theta(N)$	$\Theta(\log\log N)$	$\Theta(\frac{P}{\log\log P})$
$\Theta(1)$	$\Theta(1)$	$N-1$	a	$\Theta(a^N)$	$\Theta(N)$	$\Theta(\frac{P}{\log P})$
$\Theta(1)$	$\Theta(N)$	$\frac{N}{k}$	$a = k$	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
$\Theta(1)$	$\Theta(N)$	$\frac{N}{k}$	$a \neq k$	$\Theta(N^{\log_k a})$	$\Theta(N)$	$\Theta(\frac{P}{N})$
$\Theta(1)$	$\Theta(N)$	$\sqrt{N}$	$\sqrt{N}$	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
$\Theta(1)$	$\Theta(N)$	$N-1$	a	$\Theta(a^N)$	$\Theta(N^2)$	$\Theta(\frac{P}{(\log P)^2})$
$\Theta(N)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a = k$	$\Theta(N)$	$\Theta(N)$	$\Theta(\log P)$
$\Theta(N)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a < k$	$\Theta(N^{\log_k a})$	$\Theta(N)$	$\Theta(1)$
$\Theta(N)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a > k$	$\Theta(N^{\log_k a})$	$\Theta(N)$	$\Theta(\frac{P}{N})$
$\Theta(N)$	$\Theta(1)$ or $\Theta(N)$	$\sqrt{N}$	$\sqrt{N}$	$\Theta(N)$	$\Theta(N)$	$\Theta(\log\log P)$
$\Theta(N\log N)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a = k$	$\Theta(N)$	$\Theta(N\log N)$	$\Theta(\log P)$
$\Theta(N^2)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a \leq k^2$	$\Theta(N^{\log_k a})$	$\Theta(N^2)$	$\Theta(1)$
$\Theta(N^2)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a > k^2$	$\Theta(N^{\log_k a})$	$\Theta(N^2)$	$\Theta(\frac{P}{N^2})$

Table 1: Runtime and Speedup for Divide-and-Conquer Algorithms.

the criteria given in [GR88] and members of the class NC [KR90]. In the next section we see that all of the algorithms from Table 1 that have non-constant speedup can provide optimal speedup if fewer processors are used. Thus, we can increase processor efficiency by increasing runtime.

Surprisingly, four of the 15 classes of algorithms in Table 1 provide only constant speedup. For a particular algorithm running on a particular machine, the speedup may be less than unity, meaning that the parallel algorithm actually runs slower than its sequential counterpart.

From Table 1 we see that, in general, algorithms having constant divide-and-conquer cost provide greater speedup than those with non-constant dac-cost. Thus, these algorithms are better candidates for parallelization.

Another lesson to be learned from the above analysis is the extreme effect that communication cost can have on the performance of a parallel algorithm. For the algorithms in Table 1 having constant dac-cost, communication cost can mean the difference between speedup that is within a polylogarithmic factor of optimal and no speedup at all. (For algorithms having linear or superlinear dac-cost, communication cost has no effect on the asymptotic runtime or speedup.) This implies that algorithms having constant dac-cost are implemented most effectively on shared-memory machines. They may also be implemented effectively on distributed-memory machines if preallocation of data is feasible. Algorithms having linear or superlinear dac-cost can be implemented on either kind of architecture with only constant loss of performance, although in general these algorithms are not good candidates for parallelization.

In the preceding analysis we have assumed that the maximum possible number of processors are used to execute a parallel divide-and-conquer algorithm. While this many processors yields the fastest execution time *asymptotically*, reducing the number of processors by some constant factor may reduce the execution time by a constant factor. The actual number of processors to use to achieve the fastest runtime can be determined using the following inequality:

$$T(N,1) < \text{communicationCost}(N) + \text{dacCost}(N) + T(\text{subproblemSize}(N),1)$$

where $T(N,1)$ is the time to solve a problem of size N using a single processor. Values for N that satisfy this inequality indicate problem sizes that should be considered indivisible. Intuitively, the inequality states that the problem should be solved serially rather than in parallel when the time to solve a problem using a single processor is less than the time required to divide the problem into subproblems, send all but one of the subproblems to other processors, and solve the remaining subproblem using a single processor. [Stout87] suggests that, in some cases, a faster serial algorithm

can be used to solve the leaf subproblems, rather than using a purely recursive divide-and-conquer approach.

6 Most Efficient Number of Processors

For none of the classes of divide-and-conquer algorithms presented is the most *effective* value of P (the value that yields the fastest execution time) and the most *efficient* value of P the same. That is, the most effective value for P yields speedup less than $\Theta(P)$. Therefore we might wish to use a sub-optimal value of P (in terms of effectiveness) in order to achieve higher processor efficiency.

For P processors and a problem size of N, the speedup of a particular divide-and-conquer algorithm is given by the expression $\frac{T(N,1)}{T(N,P)}$, where $T(N,P)$ is the execution time for a problem of size N using P processors. To find the most efficient number of processors, P_{eff}, we determine a value for P such that $(P * T(N,P)) = \Theta(T(N,1)$. For example, algorithms with constant ($\Theta(1)$) dac-cost, constant communication cost, and subproblem size $\frac{N}{k}$ where $a = k$, have $T(N,1) = \Theta(N)$ and $T(N,P) = \Theta(\frac{N}{P} + \log_a P)$. Thus, the (asymptotic) number of processors that yields maximum processor efficiency is any P such that $N + P\log_a P = O(N)$. That is, any P such that $P\log_a P = O(N)$. The largest such value for P is $\Theta(\frac{N}{\log_a N})$.

Table 2 gives the largest number of processors that yields maximum efficiency, P_{eff}, for most of the algorithm classes given in Table 1. We omit the calculations involving $T(N,1)$ and $T(N,P)$ for brevity. In Table 2, N^c denotes any polynomial in N.

Algorithms for which no value of P yields optimal speedup (other than $P = 1$) are those having

- constant dac-cost, linear communication cost, and subproblem size $\frac{N}{a}$
- constant dac-cost, linear communication cost, and subproblem size $\sqrt{N}$
- linear dac-cost and subproblem size $\frac{N}{k}$, where $a < k$
- $\Theta(N^2)$ dac-cost and subproblem size $\frac{N}{k}$, where $a \leq k^2$

In fact, no value of P yields more than a constant speedup for any of these algorithms.

The analysis summarized in Table 2 is akin to the proof of a parallel divide-and-conquer algorithm's optimality in [Zorat79], in which an algorithm is optimal if it is possible to achieve $\Theta(P)$ speedup with P processors, even if P is not the number of processors yielding fastest execution time. [GR88] defines an optimal algorithm as one for which $P*T(N,P)$ is as good as the *fastest* sequential

DaC Cost	*Communication Cost*	*Subproblem Size*	*Number of Subproblems*	P_{max}	P_{eff}
$\Theta(1)$	$\Theta(1)$	$\frac{N}{k}$	$a = k$	$\Theta(N)$	$\Theta(\frac{N}{\log N})$
$\Theta(1)$	$\Theta(1)$	$\frac{N}{k}$	$a \neq k$	$\Theta(N^{\log_k a})$	$\Theta(\frac{N^{\log_k a}}{\log N})$
$\Theta(1)$	$\Theta(1)$	$\sqrt{N}$	$\sqrt{N}$	$\Theta(N)$	$\Theta(\frac{N}{\log N})$
$\Theta(1)$	$\Theta(1)$ or $\Theta(N)$	$N-1$	a	$\Theta(a^N)$	$\Theta(N^c)$
$\Theta(1)$	$\Theta(N)$	$\frac{N}{k}$	$a \neq k$	$\Theta(N^{\log_k a})$	$\Theta(\frac{N^{\log_k a}}{N})$
$\Theta(N)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a = k$	$\Theta(N)$	$\Theta(\log N)$
$\Theta(N)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a > k$	$\Theta(N^{\log_k a})$	$\Theta(\frac{N^{\log_k a}}{N})$
$\Theta(N^2)$	$\Theta(1)$ or $\Theta(N)$	$\frac{N}{k}$	$a > k^2$	$\Theta(N^{\log_k a})$	$\Theta(N^{\log_k a - 2})$

Table 2: Most Efficient Number of Processors for Divide-and-Conquer Algorithms.

algorithm, which may perform better than the sequential version of the parallel algorithm. The algorithms shown here to be optimal in the weaker sense may also be optimal in this stronger sense. (Because the complexity of the fastest sequential algorithm may vary within the group, however, it is not possible in general to decide the optimality of the group as a whole.) [KR90] gives an even stronger definition of algorithm optimality, in which $T(N, P)$ must be polylogarithmic. The only classes of divide-and-conquer algorithms given in Table 2 that might be optimal in this sense are those having $\Theta(1)$ dac-cost, $\Theta(1)$ communication cost, and subproblem size $\frac{N}{k}$ or $\sqrt{N}$.

7 Conclusions

Divide-and-conquer is a powerful problem-solving technique that is the basis for many effective sequential algorithms. This paper has examined the extent to which the divide-and-conquer approach yields effective and efficient parallel algorithms. The results show that possible speedups range from constant to linear (worst case to best case) depending on the characteristics of the algorithm, the communication costs incurred during parallelization, and the number of processors used.

Divide-and-conquer algorithms have a high degree of inherent parallelism because they involve solving independent subproblems. Yet contrary to initial expectations, none of the divide-and-conquer algorithms analyzed here have best-case speedup when the maximum possible number of

processors are used. (In fact, some of them have only constant speedup, regardless of the number of processors used.) One reason for this stems from the nature of parallel divide-and-conquer algorithms. This is, the amount of parallelism utilized by the algorithm is not static, but varies during the execution of the algorithm. At the start of execution, only a single processor is used. As execution proceeds, more processors are used, and it is only during the constant time required to solve the indivisible subproblems that all the processors are used. As results are returned and combined, fewer processors are used until once again only a single processor is running. Thus, the more *effective* we try to make a divide-and-conquer algorithm (by using more processors to solve separate subproblems), the less *efficient* we make it, because more processors must sit idle during the start-up and wind-down periods. These effects are even more significant when one considers the cost of executing the algorithms on real machines rather than on idealized architectural models.

There are some encouraging results, however. Three of the fifteen classes analyzed execute in polylogarithmic time with a polynomial number of processors, making them efficient algorithms by the standards given in [GR88] and making them members of the class NC. These are the algorithms having constant dac-cost, constant communication cost, and subproblem size $\frac{N}{k}$ or $\sqrt{N}$. This includes the divide-and-conquer algorithm for summing N numbers, if implemented on a PRAM architecture or on a distributed-memory architecture with preallocation of data to processors. It also includes the merging algorithm given in [KR90], if implemented on a PRAM architecture. A second encouraging result is that these three classes of algorithms, as well as those having constant dac-cost and subproblem size $N-1$, are within a polylogarithmic factor of optimal speedup when the maximum possible number of processors are used. (The latter, however, require an exponential number of processors, which may not be feasible for real architectures.) Thirdly, although none of the classes of divide-and-conquer algorithms analyzed here yield optimal speedup in general, most of them do provide optimal speedup when the degree of parallelism is limited (that is, when fewer than the maximum possible number of processors is used). Thus, parallel divide-and-conquer algorithms exhibit a speed-efficiency tradeoff.

The analysis presented here is useful for determining whether a particular divide-and-conquer algorithm is a good candidate for parallelization and the architecture for which it is best suited. In general, algorithms having constant ($\Theta(1)$) divide-and-conquer cost yield more effective parallel algorithms than algorithms having non-constant dac-cost. However, this is true only when these algorithms are implemented on a shared-memory machine or on a distributed-memory machine with preallocation of data to processors. (Preallocation of data may not always be feasible, however.

In particular, it is not possible when the initial data results from a previous computation that is completed in a single processor, and it is not effective when the size of results are just as large as the data from which they are computed.) Algorithms having linear or superlinear dac-cost can be effectively implemented on either shared-memory or distributed-memory architectures, although these algorithms generally yield only minimal speedup.

8 Acknowledgements

This research is supported by an IBM Graduate Fellowship. We appreciate the helpful comments made by Greg Plaxton, Martin Wong, and James Lester on earlier drafts of this paper.

References

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[AHU83] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Bentley80] Bentley, J. L., Multidimensional Divide-and-Conquer, Communications of the ACM, Vol. 23 (1980), pp. 214-229.

[Cvetanovich87] Cvetanovich, Z., The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems, IEEE Transactions on Computers, Vol. C-36, No. 4, April 1987, pp. 421-432.

[GR88] Gibbons, A., and Rytter, W., *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

[HZ83] Horowitz, E., and Zorat, A., Divide-and-Conquer for Parallel Processing, IEEE Trans. on Computers, Vol C-32, No. 6, June 1983, pp 582-585.

[KR90] Karp, R., and Ramachandran, V., Parallel Algorithms for Shared-Memory Machines, *Handbook of Theoretical Computer Science*, Volume A, J. Van Leeuwen, Ed., Elsevier Science Publishers, 1990, pp. 870-941.

[LRGKMT90] Lo, V. M., Rajopadhye, S., Gupta, S., Kelksen, D., Mohamed, M. A., and Telle, J., Mapping Divide-and-Conquer Algorithms to Parallel Architectures, Proceedings of the 1990 ICPP, Vol III, Aug. 1990, pp 128-135.

[MS91] Madala, S., and Sinclair, J. B., Performance of Synchronous Parallel Algorithms with Regular Structures, IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 1, Jan. 1991, pp 105-116.

[Sedgewick83] Sedgewick, R., *Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Stout87] Stout, Q. F., Supporting Divide-and-Conquer Algorithms for Image Processing, Journal of Parallel and Distributed Computing, vol. 4 (1987), pp. 95-115.

[Strassen69] Strassen, V., Gaussian Elimination is not Optimal, Numerische Mathematik, Vol. 13, 1969, pp 354-356.

[Zorat79] Zorat, A., *A Divide-and-Conquer Computer*, Ph.D. Dissertation, Department of Computer Science, Univ. Southern California, Los Angeles, CA, July 1979.