Parallel Sorting and Star-P Data Movement and Tree Flattening

by

David R. Cheng

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

Author .....
Department of Electrical Engineering and Computer Science
January 28, 2005

Certified by .....
Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Parallel Sorting and Star-P Data Movement and Tree Flattening

by

David R. Cheng

Submitted to the Department of Electrical Engineering and Computer Science
on January 28, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

This thesis studies three problems in the field of parallel computing. The first result provides a deterministic parallel sorting algorithm that empirically shows an improvement over two sample sort algorithms. When using a comparison sort, this algorithm is 1-optimal in both computation and communication. The second study develops some extensions to the Star-P system [7, 6] that allows it to solve more real problems. The timings provided indicate the scalability of the implementations on some systems. The third problem concerns automatic parallelization. By representing a computation as a binary tree, which we assume is given, it can be shown that the height corresponds to the parallel execution time, given enough processors. The main result of the chapter is an algorithm that uses tree rotations to reduce the height of an arbitrary binary tree to become logarithmic in the number of its inputs. This method can solve more general problems as the definition of tree rotation is slightly altered; examples are given that derive the parallel prefix algorithm, and give a speedup in the dynamic programming approach to the computation of Fibonacci numbers.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Acknowledgments

When I first met Alan Edelman, I immediately noticed his infectious enthusiasm, and his joyful impulses to just try things out, never minding if things just blew up in his face. It has been a delight to work with him, as he encouraged the group to build something great. He has been immensely supportive of my efforts, and I thank him for all his time and guidance.

My collaborators have been Viral Shah on sorting, Ron Choy on Star-P, and Percy Liang on tree flattening. Without them, the work presented here would not be made possible, and I am grateful that we have been able to enjoy these discoveries together.

I would like to thank all the friends who have supported me through the years, and taught me that life is more than just work and sleep.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Problems in parallel computing have long been studied, with the intention of performing some task more quickly when using more than one processor. In the past, the field has been primarily a scientific interest: with so-called supercomputers being expensive to acquire and maintain, their availability was limited to scientists who needed to perform computations on large data sets. While this may still the case about today's fastest systems (such as those listed in [26]), smaller parallel systems are becoming increasingly prevalent. With the arguable slowdown of Moore's Law, system designers have been looking towards parallelization for increased performance. Multi-processor motherboards are gaining popularity, and technologies such as simultaneous multi-threading and multiple processors on a single die are being commercialized. Furthermore, distributed-memory parallel computers can be made cheaply from commodity hardware: many of these "Beowulf" [24] clusters have been built, both on the low-end as well as the high-end. Therefore, the the field of parallel computing is particularly relevant in this era.

This thesis focuses on methods that lead towards the more efficient use of parallel computers. The results have been ordered based on their generality: the first tackles a specific problem; the second involves a set of tools intended to allow a programmer develop parallel code more naturally; the third makes some efforts at automatically converting a sequential computation into one that completes quickly on a parallel computer.

Chapter 2 concerns an algorithm for practical sorting in parallel. This problem, in addition to being a classic in its study in its single-processor form, is also one of the most useful subroutines. We develop a new algorithm that pays particular care to distributing the workload almost perfectly, and also uses close to the optimal amount of communication. Included is an empirical study that supports the theoretical claims of efficiency.

Chapter 3 discusses the engineering challenges of building a system for parallel computing. It describes the goals and design of the Star-P project [7, 6] as a means of reducing the development time of parallel applications. The main contributions in this chapter are two extensions to the system: distributed input and output and inter-process communication through shared memory.

Chapter 4 makes some theoretical advances in the automatic parallelization of algorithms. The premise is that users should make the most use of multiple processors while having to learn a minimal amount of material specific to the domain of parallel computing. This chapter isolates one of the core problems on this topic, by assuming a tree representation of a computation. We present some novel ideas on how tree rotations can be used to derive efficient parallel algorithms. The chapter finishes with some examples that demonstrate the generalizations attainable by this method.

Chapter 2

Parallel Sorting

Parallel sorting has been widely studied in the last couple of decades for a variety of computer architectures. Many high performance computers today have distributed memory, and commodity clusters are becoming increasingly common. The cost of communication is significantly larger than the cost of computation on a distributed memory computer. We propose a sorting algorithm that is close to optimal in both the computation and communication required.

Blelloch et al. [1] compare several parallel sorting algorithms on the CM–2, and report that a sampling based sort and radix sort are good algorithms to use in practice. We first tried a sampling based sort, but this had a couple of problems. A sampling sort requires a redistribution phase at the end, so that the output has the desired distribution. The sampling process itself requires "well chosen" parameters to yield "good" samples. We noticed that we can do away with both these steps if we can determine exact splitters quickly. Saukas and Song [21] describe a quick parallel selection algorithm. Our algorithm extends this work to efficiently find $p-1$ exact splitters in $O(p \log n)$ rounds of communication, providing a 1-optimal parallel sorting algorithm.

2.1 Algorithm Description

Before giving the concise operation of the sorting algorithm, we begin with some assumptions and notation.

We are given p processors to sort n total elements in a vector v. Assume that the input elements are already load balanced, or *evenly distributed* over the p processors.[1] Note that the values may be arbitrary. In particular, we rank the processors $1 \ldots p$, and define v_i to be the elements held locally by processor i. The *distribution* of v is a vector d where $d_i = |v_i|$. Then we say v is evenly distributed if it is formed by the concatenation $v = v_1 \ldots v_p$, and $d_i \leq \lceil \frac{n}{p} \rceil$ for any i.

In the algorithm description below, we assume the task is to sort the input in increasing order. Naturally, the choice is arbitrary and any other comparison function may be used.

Algorithm.
Input: A vector v of n total elements, evenly distributed among p processors.
Output: An evenly distributed vector w with the same distribution as v, containing the sorted elements of v.

1. Sort the local elements v_i into a vector v'_i.
2. Determine the exact splitting of the local data:
   (a) Compute the partial sums $r_0 = 0$ and $r_j = \sum_{k=1}^{j} d_k$ for $j = 1 \ldots p$.
   (b) Use a parallel select algorithm to find the elements $e_1, \ldots, e_{p-1}$ of global rank $r_1, \ldots, r_{p-1}$, respectively.
   (c) For each r_j, have processor i compute the local index s_{ij} so that $r_j = \sum_{i=1}^{p} s_{ij}$ and the first s_{ij} elements of v'_i are no larger than e_j.
3. Reroute the sorted elements in v'_i according to the indices s_{ij}: processor i sends elements in the range $s_{ij-1} \ldots s_{ij}$ to processor j.
4. Locally merge the p sorted sub-vectors into the output w_i.

The details of each step now follow.

[1] If this assumption does not hold, an initial redistribution step can be added.

2.1.1 Local Sort

The first step may invoke any local sort applicable to the problem at hand. It is beyond the scope of this study to devise an efficient sequential sorting algorithm, as the problem is very well studied. We simply impose the restriction that the algorithm used here should be identical to the one used for a baseline comparison on a non-parallel computer. Define the computation cost for this algorithm on an input of size n to be $T_s(n)$. Therefore, the amount of computation done by processor i is just $T_s(d_i)$. Because the local sorting must be completed on each processor before the next step can proceed, the global cost is $\max_i T_s(d_i) = T_s(\lceil \frac{n}{p} \rceil)$. With a radix sort, this becomes $O(n/p)$; with a comparison-based sort, $O(\frac{n}{p} \lg \frac{n}{p})$.

2.1.2 Exact Splitting

This step is nontrivial, and the main result of this paper follows from the observation that exact splitting over locally sorted data can be done efficiently.

The method used for simultaneous selection was given by Saukas and Song in [21], with two main differences: local ranking is done by binary search rather than partition, and we perform $O(\lg n)$ rounds of communication rather than $O(\lg cp)$ for some constant c. For completeness, a description of the selection algorithm is given below.

Single Selection

First, we consider the simpler problem of selecting just one target, an element of global rank r.[2] The algorithm for this task is motivated by the sequential methods for the same problem, most notably the one given in [2].

Although it may be clearer to define the selection algorithm recursively, the practical implementation and extension into simultaneous selection proceed more naturally from an iterative description. Define an active range to be the contiguous sequence of elements in v_i' that may still have rank r, and let a_i represent its size. Note that

[2]To handle the case of non-unique input elements, any element may actually have a range of global ranks. To be more precise, we want to find the element whose set of ranks contains r.

the total number of active elements is $\sum_{i=1}^{p} a_i$. Initially, the active range on each processor is the entire vector v_i' and a_i is just the input distribution d_i. In each iteration of the algorithm, a "pivot" is found that partitions the active range in two. Either the pivot is determined to be the target element, or the next iteration continues on one of the partitions.

Each processor i performs the following steps:

1. Index the median m_i of the active range of v_i', and broadcast the value.

2. Weight median m_i by $\frac{a_i}{\sum_{k=1}^{p} a_k}$. Find the weighted median of medians m_m. By definition, the weights of the $\{m_i | m_i < m_m\}$ sum to at most $\frac{1}{2}$, as do the weights of the $\{m_i | m_i > m_m\}$.

3. Binary search m_m over the active range of v_i' to determine the first and last positions f_i and l_i it can be inserted into the sorted vector v_i'. Broadcast these two values.

4. Compute $f = \sum_{i=1}^{p} f_i$ and $l = \sum_{i=1}^{p} l_i$. The element m_m has ranks $[f, l]$ in v.

5. If $r \in [f, l]$, then m_m is the target element and we exit. Otherwise the active range is truncated as follows:

   Increase the bottom index to $l_i + 1$ if $l < r$; or decrease the top index to $f_i - 1$ if $r < f$.

   Loop on the truncated active range.

We can think of the weighted median of medians as a pivot, because it is used to split the input for the next iteration. It is a well-known result that the weighted median of medians can be computed in linear time [20, 9]. One possible way is to partition the values with the (unweighted) median, accumulate the weights on each side of the median, and recurse on the side that has too much weight. Therefore, the amount of computation in each round is $O(p) + O(\lg a_i) + O(1) = O(p + \lg \frac{n}{p})$ per processor.

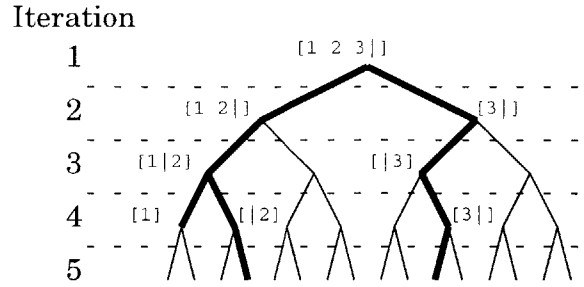


Figure 2-1: Example execution of selecting three elements.
Each node corresponds to a contiguous range of v'_i, and gets split into its two children by the pivot. The root is the entire v'_i, and the bold traces which ranges are active at each iteration. The array at a node represents the target ranks that may be found by the search path, and the vertical bar in the array indicates the relative position of the pivot's rank.

Furthermore, as shown in [21], splitting the data by the weighted median of medians will decrease the total number of active elements by at least a factor of $\frac{1}{4}$. Because the step begins with n elements under consideration, there are $O(\lg n)$ iterations. The total single-processor computation for selection is then $O(p \lg n + \lg \frac{n}{p} \lg n) = O(p \lg n + \lg^2 n)$.

The amount of communication is straightforward to compute: two broadcasts per iteration, for $O(p \lg n)$ total bytes being transferred over $O(\lg n)$ rounds.

Simultaneous Selection

The problem is now to select multiple targets, each with a different global rank. In the context of the sorting problem, we want the $p-1$ elements of global rank $d_1, d_1 + d_2, \ldots, \sum_{i=1}^{p-1} d_i$. One simple way to do this would call the single selection problem for each desired rank. Unfortunately, doing so would increase the number communication rounds by a factor of $O(p)$. We can avoid this inflation by solving multiple selection problems independently, but combining their communication. Stated another way, instead of finding $p-1$ paths one after another from root to leaf of the binary search tree, we take a breadth-first search with breadth at most $p-1$ (see Figure 2-1).

To implement simultaneous selection, we augment the single selection algorithm

with a set A of active ranges. Each of these active ranges will produce at least one target. An iteration of the algorithm proceeds as in single selection, but finds multiple pivots: a weighted median of medians for each active range. If an active range produces a pivot that is one of the target elements, we eliminate that active range from A (as in the leftmost branch of Figure 2-1). Otherwise, we examine each of the two partitions induced by the pivot, and add it to A if it may yield a target. Note that as in iterations 1 and 3 in Figure 2-1, it is possible for both partitions to be added.

In slightly more detail, we handle the augmentation by looping over A in each step. The local medians are bundled together for a single broadcast at the end of Step 1, as are the local ranks in Step 3. For Step 5, we use the fact that each active range in A has a corresponding set of the target ranks: those targets that lie between the bottom and top indices of the active range. If we keep the subset of target ranks sorted, a binary search over it with the pivot rank[3] will split the target set as well. The left target subset is associated with the left partition of the active range, and the right sides follow similarly. The left or right partition of the active range gets added to A for the next iteration only if the corresponding target subset is non-empty.

The computation time necessary for simultaneous selection follows by inflating each step of the single selection by a factor of p (because $|A| \leq p$). The exception is the last step, where we also need to binary search over $O(p)$ targets. This amount to $O(p + p^2 + p \lg \frac{n}{p} + p + p \lg p) = O(p^2 + p \lg \frac{n}{p})$ per iteration. Again, there are $O(\lg n)$ iterations for total computation time of $O(p^2 \lg n + p \lg^2 n)$.

This step runs in $O(p)$ space, the scratch area needed to hold received data and pass state between iterations.

The communication time is similarly inflated: two broadcasts per round, each having one processor send $O(p)$ data to all the others. The aggregate amount of data being sent is $O(p^2 \lg n)$ over $O(\lg n)$ rounds.

[3]Actually, we binary search for the first position f may be inserted, and for the last position l may be inserted. If the two positions are not the same, we have found at least one target.

Producing Indices

Each processor computes a local matrix S of size $p \times (p+1)$. Recall that S splits the local data v_i' into p segments, with $s_{k0} = 0$ and $s_{kp} = d_k$ for $k = 1 \ldots p$. The remaining $p-1$ columns come as output of the selection. For simplicity of notation, we briefly describe the output procedure in the context of single selection; it extends naturally for simultaneous selection. When we find that a particular m_m has global ranks $[f, l) \ni r_k$, we also have the local ranks f_i and l_i. There are $r_k - f$ excess elements with value m_m that should be routed to processor k. For stability, we assign s_{ki} from $i = 1$ to p, taking as many elements as possible without overstepping the excess. More precisely,

$$s_{ki} = \min \left\{ f_i + (r_k - f) - \sum_{j=1}^{i-1} (s_{kj} - f_j),\ l_i \right\}$$

The computation requirements for this step are $O(p^2)$ to populate the matrix S; the space used is also $O(p^2)$.

2.1.3 Element Rerouting

This step is purely one round of communication. There exist inputs such that the input and output locations of each element are different. Therefore, the aggregate amount of data being communicated in this step is $O(n)$. However, note that this cost can not be avoided. An optimal parallel sorting algorithm must communicate at least the same amount of data as done in this step, simply because an element must at least move from its input location to its output location.

The space requirement is an additional $O(d_i)$, because we do not have the tools for in-place sending and receiving of data.

2.1.4 Merging

Now each processor has p sorted sub-vectors. and we want to merge them into a single sorted sequence. The simple approach we take for this problem is to conceptually

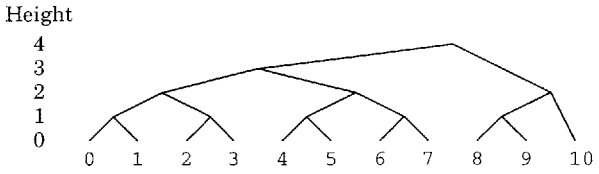


Figure 2-2: Sequence of merges for p not a power of 2.

build a binary tree on top of the vectors. To handle the case of p that are not powers of 2, we say a node of height i has at most 2^i leaf descendants, whose ranks are in $[k \cdot 2^i, (k+1) \cdot 2^i)$ for some k (Figure 2-2). It is clear that the tree has height $\leq \lceil \lg p \rceil$.

From this tree, we merge pairs of sub-vectors out-of-place, for cache efficiency. This can be done by alternating between the temporary space necessary for the rerouting step, and the eventual output space.

Notice that a merge will move a particular element exactly once (from one buffer to its sorted position in the other buffer). Furthermore, there is at most one comparison for each element move. Finally, every time an element gets moved, it goes into a sorted sub-vector at a higher level in the tree. Therefore each element moves at most $\lceil \lg p \rceil$ times, for a total computation time of $d_i \lceil \lg p \rceil$. Again, we take the time of the slowest processor, for computation time of $\lceil \frac{n}{p} \rceil \lceil \lg p \rceil$.

2.1.5 Theoretical Performance

We want to compare this algorithm against an arbitrary parallel sorting algorithm with the following properties:

1. Total computation time $T_s^*(n,p) = \frac{1}{p} T_s(n)$ for $1 \leq p \leq P$, linear speedup in p over any sequential sorting algorithm with running time $T_s(n)$.

2. Minimal amount of cross-processor communication $T_c^*(v)$, the number of elements that begin and end on different processors.

We will not go on to claim that such an algorithm is truly an optimal parallel algorithm, because we do not require $T_s(n)$ to be optimal. However, optimality of $T_s(n)$ does imply optimality of $T_s^*(n,p)$ for $p \leq P$. Briefly, if there were a faster

$T'_s(n,p)$ for some p, then we could simulate it on a single processor for total time $pT'_s(n,p) < pT^*_s(n,p) = T_s(n)$, which is a contradiction.

Computation

We can examine the total computation time by adding together the time for each step, and comparing against the theoretical $T^*_s(n,p)$:

$$\begin{aligned} & T_s(\lceil \tfrac{n}{p} \rceil) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ \leq\ & \frac{1}{p} T_s(n+p) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ =\ & T^*_s(n+p,p) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \end{aligned}$$

The inequality follows from the fact that $T^*_s(n) = \Omega(n)$.

It is interesting to note the case where a comparison sort is necessary. Then we use a sequential sort with $T_s(n) \leq c\lceil \frac{n}{p} \rceil \lg \lceil \frac{n}{p} \rceil$ for some $c \geq 1$. We can then combine this cost with the time required for merging (Step 4):

$$\begin{aligned} & c\lceil \tfrac{n}{p} \rceil \lg \lceil \tfrac{n}{p} \rceil + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ \leq\ & c\lceil \tfrac{n}{p} \rceil \lg(n+p) + \lceil \tfrac{n}{p} \rceil (\lceil \lg p \rceil - c \lg p) \\ \leq\ & c\lceil \tfrac{n}{p} \rceil \lg n + c\lceil \tfrac{n}{p} \rceil \lg(1 + \tfrac{p}{n}) + \lceil \tfrac{n}{p} \rceil (\lceil \lg p \rceil - c \lg p) \\ \leq\ & \frac{cn \lg n}{p} + \lg n + 2c + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \end{aligned}$$

With comparison sorting, the total computation time becomes:

$$T^*_s(n,p) + O(p^2 \lg n + p \lg^2 n) + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \qquad (2.1)$$

Furthermore, $T^*_s(n,p)$ is optimal to within the constant factor c.

Communication

We have already established that the exact splitting algorithm will provide the final locations of the elements. The amount of communication done in the rerouting phase is then the optimal amount. Therefore, total cost is:

$$T_c^*(v) \text{ in 1 round} + O(p^2 \lg n) \text{ in } \lg n \text{ rounds}$$

Space

The total space usage aside from the input is:

$$O\left(p^2 + \frac{n}{p}\right)$$

Requirements

Given these bounds, it is clear that this algorithm is only practical for $p^2 \leq \frac{n}{p} \Rightarrow p^3 \leq n$. Returning to the formulation given in Section 2.1.5, we have $p = \lfloor n^{1/3} \rfloor$. This requirement is a common property of other parallel sorting algorithms, particularly sample sort (i.e. [1, 23, 14], as noted in [13]).

Analysis in the BSP Model

A bulk-synchronous parallel computer, described in [37], models a system with three parameters: p, the number of processors; L, the minimum amount of time between subsequent rounds of communication; and g, a measure of bandwidth in time per message size. Following the naming conventions of [12], define π to be the ratio of computation cost of the BSP algorithm to the computation cost of a sequential algorithm. Similarly, define μ to be the ratio of communication cost of the BSP algorithm to the number of memory movements in a sequential algorithm. We say an algorithm is c-optimal in computation if $\pi = c + o(1)$ as $n \to \infty$, and similarly for μ and communication.

We may naturally compute the ratio π to be Equation 2.1 over $T_s^*(n,p) = \frac{cn \lg n}{p}$.

Thus,

$$\pi = 1 + \frac{p^3}{cn} + \frac{p^2 \lg n}{cn} + \frac{1}{c \lg n} = 1 + o(1) \text{ as } n \to \infty$$

Furthermore, there exist movement-optimal sorting algorithms (i.e. [11]), so we compute μ against $\frac{gn}{p}$. It is straightforward to verify that the BSP cost of exact splitting is $O(\lg n \max\{L, gp^2 \lg n\})$, giving us

$$\mu = 1 + \frac{pL \lg n}{gn} + \frac{p^3 \lg^2 n}{n} = 1 + o(1) \text{ as } n \to \infty$$

Therefore the algorithm is 1-optimal in both computation and communication.

Exact splitting dominates the cost beyond the local sort and the rerouting steps. The total running time is therefore $O(\frac{n \lg n}{p} + \frac{gn}{p} + \lg n \max\{L, gp^2 \lg n\})$. This bound is an improvement on that given by [13], for small L and $p^2 \lg^2 n$. The tradeoff is that we have decreased one round of communicating much data, to use many rounds of communicating little data. Our experimental results indicate that this choice is reasonable.

2.2 Results

The communication costs are also near optimal if we assume that p is small, and there is little overhead for a round of communication. Furthermore, the sequential computation speedup is near linear if $p \ll n$, and we need comparison-based sorting. Notice that the speedup is given with respect to a sequential algorithm, rather than to itself with small p. The intention is that efficient sequential sorting algorithms and implementations can be developed without any consideration for parallelization, and then be simply dropped in for good parallel performance.

We now turn to empirical results, which suggest that the exact splitting uses little computation and communication time.

2.2.1 Experimental Setup

We implemented the algorithm using MPI with C++. The motivation is for the code to be used as a library with a simple interface; it is therefore templated, and comparison based. As a sequential sort, it calls `stable_sort` provided by STL. For the element rerouting, it calls `MPI_Alltoallv` provided by LAM 6.5.9. We shall treat these two operations as primitives, as it is beyond the scope of this study to propose efficient algorithms for either of them.

We test the algorithm on a distributed-memory system built from commodity hardware, with 16 nodes. Each node contains a Xeon 2.40 GHz processor and 2 Gb of memory. Each node is connected through a Gigabit Ethernet switch, so the network distance between any two nodes is exactly two hops.

In this section, all the timings are based on the average of eight trials, and are reported in seconds.

2.2.2 Sorting Uniform Integers

For starters, we can examine in detail how this sorting algorithm performs when the input values are uniformly distributed random (32-bit) integers. After we get a sense of what steps tend to be more time-consuming, we look at other inputs and see how they affect the times.

Figure 2-3 displays the speedup of the algorithm as a function of the number of processors p, over a wide range of input sizes. Not surprisingly, the smallest problems already run very efficiently on a single processor, and therefore do not benefit much from parallelization. However, even the biggest problems display a large gap between their empirical speedup and the optimal linear speedup.

Table 2.1 provides the breakdown of the total sorting time into the component steps. As explained in Section 2.1.3, the cost of the communication-intensive element rerouting can not be avoided. Therefore, we may examine how this algorithm performs when excluding the rerouting time; the speedups in this artificial setting are given by Figure 2-4. The results suggest that the algorithm is near-optimal for large input

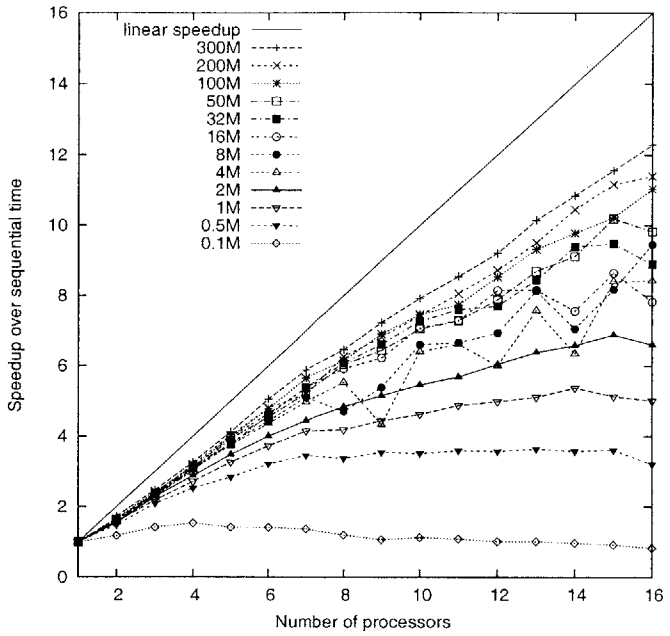


Figure 2-3: Total speedup on various-sized inputs

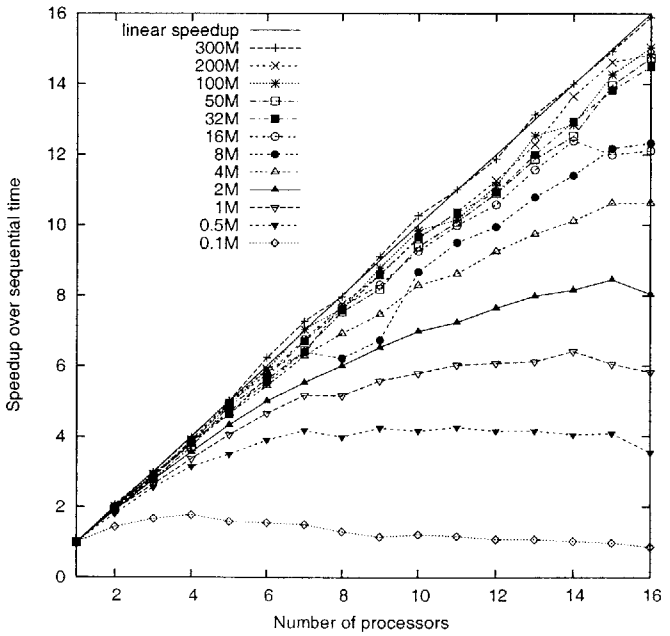


Figure 2-4: Speedup when leaving out time for element rerouting

Table 2.1: Absolute times for sorting uniformly distributed integers

0.5 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
1	0.097823	0.097823	0	0	0
2	0.066122	0.047728	0.004427	0.011933	0.002034
4	0.038737	0.022619	0.005716	0.007581	0.002821
8	0.029049	0.010973	0.012117	0.004428	0.001531
16	0.030562	0.005037	0.021535	0.002908	0.001082

32 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
1	7.755669	7.755669	0	0	0
2	4.834520	3.858595	0.005930	0.842257	0.127738
4	2.470569	1.825163	0.008167	0.467573	0.169665
8	1.275011	0.907056	0.016055	0.253702	0.098198
16	0.871702	0.429924	0.028826	0.336901	0.076051

300 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
1	84.331021	84.331021	0	0	0
2	48.908290	39.453687	0.006847	8.072060	1.375696
4	25.875986	19.532786	0.008859	4.658342	1.675998
8	13.040635	9.648278	0.017789	2.447276	0.927293
16	6.863963	4.580638	0.032176	1.557003	0.694146

sizes (of at least 32 million), and "practically optimal" for very large inputs.

We can further infer from Table 2.1 that the time for exact splitting is small. The dominating factor in the splitting time is a linear dependence on the number of processors; because little data moves in each round of communication, the constant overhead of sending a message dominates the growth in message size. Therefore, despite the theoretical performance bound of $O(p^2 \lg^2 n)$, the empirical performance suggests the more favorable $O(p \lg n)$. This latter bound comes from $O(\lg n)$ rounds, each taking $O(p)$ time. The computation time in this step is entirely dominated by the communication. Figure 2-5 provides further support of the empirical bound: modulo some outliers, the scaled splitting times do not show a clear tendency to increase as we move to the right along the x-axis (more processors and bigger problems).

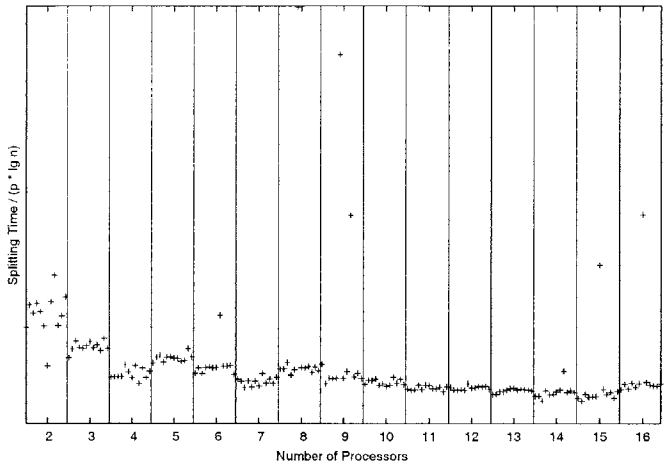


Figure 2-5: Evidence of the empirical running time of $O(p \lg n)$ for exact splitting. Inside each slab, p is fixed while n increases (exponentially) from left to right. The unlabeled y-axis uses a linear scale.

2.2.3 Sorting Contrived Inputs

As a sanity check, we experiment with sorting an input consisting of all zeros. Table 2.2 gives the empirical results. The exact splitting step will use one round of communication and exit, when it realizes 0 contains all the target ranks it needs. Therefore, the splitter selection should have no noticeable dependence on n (the time for a single binary search is easily swamped by one communication). However, the merge step is not so clever, and executes without any theoretical change in its running time.

We can elicit worst-case behavior from the exact splitting step, by constructing an input where the values are shifted: each processor initially holds the values that will end up on its neighbor to the right (the processor with highest rank will start off with the lowest $\lceil \frac{n}{p} \rceil$ values). This input forces the splitting step to search to the bottom, eliciting $\Theta(\lg n)$ rounds of communication. Table 2.3 gives the empirical results, which do illustrate a higher cost of exact splitting. However, the time spent in the step remains insignificant compared to the time for the local sort (assuming sufficiently large $\frac{n}{p}$). Furthermore, the overall times actually decrease with this input because much of the rerouting is done in parallel, except for the case where $p = 2$.

These results are promising, and suggest that the algorithm performance is quite

Table 2.2: Absolute times for sorting zeros

32 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
2	2.341816	2.113241	0.000511	0.1525641	0.075500
4	1.098606	0.984291	0.000625	0.0758215	0.037869
8	0.587438	0.491992	0.001227	0.0377661	0.056453
16	0.277607	0.228586	0.001738	0.0191964	0.028086
300 Million integers					
16	2.958025	2.499508	0.001732	0.1855574	0.271227

Table 2.3: Absolute times for shifted input

32 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
2	5.317869	3.814628	0.014434	1.412940	0.075867
4	2.646722	1.809445	0.018025	0.781250	0.038002
8	1.385863	0.906335	0.038160	0.384264	0.057105
16	0.736111	0.434600	0.061537	0.211697	0.028277
300 Million integers					
16	7.001009	4.601053	0.074080	2.044193	0.281683

robust against various types of inputs.

2.2.4 Comparison against Sample Sorting

Several prior works [1, 14, 23] conclude that an algorithm known as sample sort is the most efficient for large n and p. Such algorithms are characterized by having each processor distribute its $\lceil \frac{n}{p} \rceil$ elements into p buckets, where the bucket boundaries are determined by some form of sampling. Once the buckets are formed, a single round of all-to-all communication follows, with each processor i receiving the contents of the ith bucket from everybody else. Finally, each processor performs some local computation to place all its received elements in sorted order.

The major drawback of sample sort is that the final distribution of elements is uneven. Much of the work in sample sorting is directed towards reducing the amount of imbalance, providing schemes that have theoretical bounds on the largest amount

of data a processor can collect in the rerouting. The problem with one processor receiving too much data is that the computation time in the subsequent steps are dominated by this one overloaded processor. As a result, 1-optimality is more difficult to obtain. Furthermore, some applications require an exact output distribution; this is often the case when sorting is just one part of a multi-step process. Then an additional redistribution step would be necessary, where the elements across the boundaries are communicated.

We compare the exact splitting algorithm of this paper with two existing sample sorting algorithms.

A Sort-First Sample Sort

The approach of [23] is to first sort the local segments of the input, then use evenly spaced elements to determine the bucket boundaries (splitters). Because the local segment is sorted, the elements that belong to each bucket already lie in a contiguous range. Therefore, a binary search of each splitter over the sorted input provides the necessary information for the element rerouting. After the rerouting, a p-way merge puts the distributed output in sorted order. Note that the high-level sequence of sort, split, reroute, merge is identical to the algorithm presented in this paper. If we assume the time to split the data is similar for both algorithms, then the only cause for deviation in execution time would be the unbalanced data going through the merge. Define s to be the smallest value where each processor ends up with no more than $\lceil \frac{n}{p} \rceil + s$ elements. The additional cost of the merge step is simply $O(s \lg p)$. Furthermore, the cost of redistribution is $O(s)$. The loose bound given in [23] is $s = O(\frac{n}{p})$.

One of the authors of [14] has made available[28] the source code to an implementation of [23], which we use for comparison. This code uses a radix sort for the sequential task, which we drop into the algorithm given by this paper (replacing STL's `stable_sort`). The code also leaves the output in an unbalanced form; we have taken the liberty of using our own redistribution code to balance the output, and report the time for this operation separately. From the results given in Table

Table 2.4: Comparison against a sort-first sample sort on uniform integer input

8 Million integers			
p	Exact	Sample	Redist
2	0.79118	0.842410	0
4	0.44093	0.442453	0.081292
8	0.24555	0.257069	0.040073
64 Million integers			
p	Exact	Sample	Redist
2	6.70299	7.176278	0
4	3.56735	3.706688	0.702736
8	2.01376	2.083136	0.324059

2.4, we can roughly see the linear dependence on the redistribution on n. Also, as $\frac{n}{p}$ decreases (by increasing p for fixed n), we see the running time of sample sort get closer to that of the exact splitting algorithm.

The result of [14] improves the algorithm in the choice of splitters, so that s is bounded by $\sqrt{np}$. However, such a guarantee would not significantly change the results presented here: the input used is uniform, allowing regular sampling to work well enough. The largest excess s in these experiments remains under the bound of [14].

A Sort-Last Sample Sort

The sample sort algorithm given in [1] avoids the final merge by performing the local sort at the end. The splitter selection, then, is a randomized algorithm with high probability of producing a good (but not perfect) output distribution. Given the splitters, the buckets are formed by binary searching each of the $\lceil \frac{n}{p} \rceil$ input elements over the sorted set of splitters. Because there are at least p buckets, creating the buckets has cost $\Omega(\frac{n}{p} \lg p)$. The theoretical cost of forming buckets is at least that of merging.

Additionally, the cost of an imbalance s depends on the sequential sorting algorithm used. With a radix sort, the extra (additive) cost simply becomes $O(s)$, which is less than the imbalance cost in the sort-first approach. However, a comparison-based

Table 2.5: Performance of a sort-last sample sort on a uniform integer input

32 Million integers				
p	Sample	Bucket	Sort	Redist
2	0.000223	1.131244	3.993676	0.584702
4	0.000260	0.671177	1.884212	0.308441
8	0.000449	0.373997	0.926648	0.152829
16	0.000717	0.222345	0.472611	0.081180
300 Million integers				
16	0.000717	1.972227	4.785449	0.695507

setting forces an increase in computation time by a super-linear $\Omega(s \lg \frac{n}{p})$.

We were unable to obtain an MPI implementation of such a sample sort algorithm, so implemented one ourselves. Table 2.5 contains the results, with the rerouting time omitted as irrelevant. By comparing against Table 2.1, we see that the local sort step contains the expected inflation from imbalance, and the cost of redistribution is similar to that in Table 2.4. Somewhat surprising is the large cost of the bucket step; while theoretically equivalent to merge, it is inflated by cache inefficiencies and an oversampling ratio used by the algorithm.

2.3 Discussion

The algorithm presented here has much in common with the one given by [21]. The two main differences are that it performs the sequential sort after the rerouting step, and contains one round of $O(\frac{n}{p})$ communication on top of the rerouting cost. This additional round produces an algorithm that is 2-optimal in communication; direct attempts to reduce this one round of communication will result in adding another $\frac{n}{p} \lg \frac{n}{p}$ term in the computation, thus making it 2-optimal in computation.

Against sample sorts, the algorithm also compares favorably. In addition to being parameter-less, it naturally exhibits a few nice properties that present problems for some sample sort algorithms: duplicate values do not cause any imbalance, and the sort is stable if the underlying sequential sort is stable. Furthermore, the only memory

copies of size $O(\frac{n}{p})$ are in the rerouting step, which is forced, and the merging, which is cache-efficient.

There lies room for further improvement in practical settings. The cost of the merging can be reduced by interleaving the p-way merge step with the element rerouting, merging sub-arrays as they are received. Alternatively, using a data structure such as a funnel (i.e. [5, 10]) may exploit even more good cache behavior to reduce the time. Another potential area of improvement is in the exact splitting. Instead of traversing search tree to completion, a threshold can be set; when the active range becomes small enough, a single processor gathers all the remaining active elements and completes the computation sequentially. This method, used by Saukas and Song in [21], helps reduce the number of communication rounds in the tail end of the step. Finally, this parallel sorting algorithm will directly benefit from future improvements to sequential sorting and all-to-all communication.

Chapter 3

Star-P Data Movement

The previous chapter presented an efficient algorithm for a single task, sorting. While the execution time of the algorithm may be fast, its inception time is quite long. The algorithm designer has to keep track of which processor has which data at each step, minimize the communication taken, and keep the work evenly balanced. Furthermore, the implementer has to carefully use the low-level communication primitives such as barrier and nonblocking send and receive. It is surprisingly easy to write inefficient parallel code: it takes significantly more effort to produce a functioning parallel program than its sequential equivalent.

This difficulty is well known, and there are both ongoing research projects (such as [25, 27]) as well as commercial products ([35, 29, 30]) that provide a parallel programming language and compiler. By shifting some of the task of parallelization to the machine, the programmer has hopefully traded off low-level control for faster development time.

This chapter details the method and goals of several extensions to the Star-P project.

3.1 Star-P

Star-P, presented in detail in [15, 8, 6, 7], is another such project. It is a collection of basic operations implemented efficiently on a parallel machine, that can be ex-

posed through various front-end interfaces. In this chapter, we focus on the Star-P project with a MATLAB [31] interface, which is currently the only interface that has received significant development; collectively, the two together have been referred to as MATLAB*p.

The user may distribute his matrix data among the processors, and use them in computations the same way he would use ordinary MATLAB matrices. Operations that use single-processor data are executed by MATLAB as usual, while simple operations on distributed objects are typically overloaded to use the parallel counterparts. One design goal is for a large number of higher-level MATLAB functions and scripts to operate unchanged on both single-processor and distributed inputs.

3.1.1 Data Distribution

A matrix is created as either input from a file, or as the result of a computation. Section 3.2.2 describes the method for loading a matrix as a distributed object. Otherwise, a computation will generally return a distributed output if given distributed input, unless the output is very small. The motivation for these semantics is that distributed matrices should be quite large, in order to overcome the overhead of parallelization.

The user can directly construct a distributed object in one of two ways: by converting a single-processor object, or by using any number of MATLAB functions such as `rand`, `eye`, `ones`, and so on. The first method specifies the distribution by number, while the second uses the more intuitive `*p` notation. The use of either method to produce one of the three supported data distributions is summarized in Table 3.1. In the figures that illustrate each distribution, the matrix is split over four processors. Each processor is represented by a different pattern, which indicates the range of elements the processor holds.

For a row-distributed matrix, each processor holds a contiguous block of $\frac{n}{p}$ rows[1] The row and column distributions are similar; one is precisely the transpose of the

[1]In the case of n not divisible by p, all but the last processor each have $\lceil \frac{n}{p} \rceil$ rows, while the processor with highest rank holds the remainder.

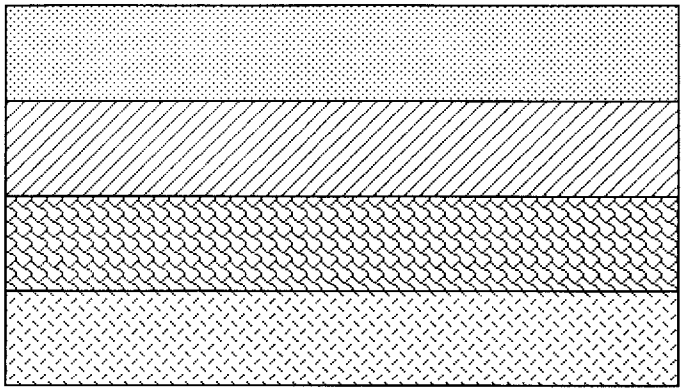


Figure 3-1: A row distributed matrix.

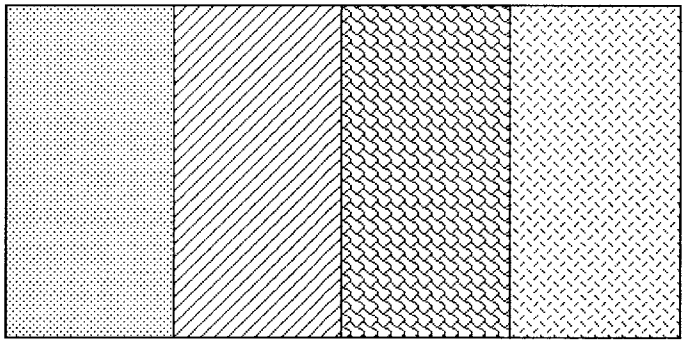


Figure 3-2: A column distributed matrix.

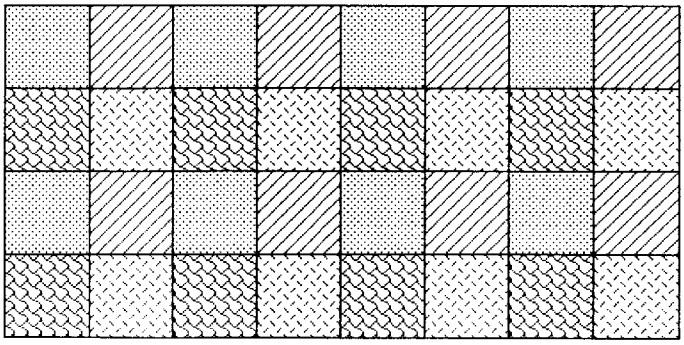


Figure 3-3: A matrix with block cyclic distribution.

Table 3.1: Summary of supported data distributions.

Distribution	Illustration	Integer Descriptor	*p Notation
Row	Figure 3-1	`matlab2pp(A, 1)`	`rand(100*p, 100)`
Column	Figure 3-2	`matlab2pp(A, 2)`	`rand(100, 100*p)`
Block cyclic	Figure 3-3	`matlab2pp(A, 3)`	`rand(100*p, 100*p)`

other. The block cyclic distribution is a little more complicated: the matrix is "tiled" by a rectangle of $a \times b$ cells, where each cell rectangle has a fixed size and is assigned to a unique processor. Therefore, $a \cdot b = p$ and each processor holds the elements that lie in its cells. Figure 3-3 gives the example of $p = 4$, $a = 2$, $b = 2$.

3.1.2 Examples

Figure 3-4 contains a simple session of MATLAB*p, where the `tic/toc` timings have been included in the right margin. This example illustrates several points:

- Line 1 runs the Star-P server on eight processors of the parallel machine.
- Line 4 shows MATLAB starting after the Star-P server has initialized.
- Lines 6–13 is standard MATLAB startup.
- Line 15 indicates that the MATLAB*p startup script has started running, making a socket connection to the Star-P backend indicated by Line 3 (see Section 3.1.3).
- Lines 20–21 demonstrate matrix multiplication on the MATLAB frontend.
- Line 22 generates random data directly on the Star-P backend.
- Line 23 performs parallel matrix multiplication.
- Lines 25–29 repeat the example on larger input. Note that operations are faster in parallel than on the MATLAB frontend. It is typical to see speedup only when the input becomes large enough to overcome the overhead of parallelization. This is especially so with MATLAB*p.

```
 1: % ./startmp 8
 2: Starting StarP
 3: StarP started, IP=10.12.0.1, port=59472
 4: Starting MATLAB with no X support
 5:
 6:                          < M A T L A B >
 7:              Copyright 1984-2004 The MathWorks, Inc.
 8:             Version 7.0.1.24704 (R14) Service Pack 1
 9:                        September 13, 2004
10:
11:
12:   To get started, type one of these: helpwin, helpdesk, or demo.
13:   For product information, visit www.mathworks.com.
14:
15: Connecting to Star-P Server with 8 processes
16:
17: Star-P Client.
18:
19: >> n = 1000;
20: >> A = randn(n, n);          Elapsed time is 0.056825 seconds.
21: >> A * A;                    Elapsed time is 0.508834 seconds.
22: >> B = randn(n*p, n);        Elapsed time is 0.165991 seconds.
23: >> B * B;                    Elapsed time is 0.720544 seconds.
24: >>
25: >> n = 3000;
26: >> A = randn(n, n);          Elapsed time is 0.527446 seconds.
27: >> A * A;                    Elapsed time is 13.094276 seconds.
28: >> B = randn(n*p, n);        Elapsed time is 0.423919 seconds.
29: >> B * B;                    Elapsed time is 6.526370 seconds.
```

Figure 3-4: MATLAB*p session performing matrix multiplication.

Figure 3-5 is another session that contains a few more items of note:

- Line 5 indicates that explicitly distributing a large MATLAB object onto the Star-P backend is slow.
- Lines 6–9 illustrate the standard display of distributed objects. By just sending the dimensions, the data remains on the backend; also, it is clear that the column vector is row-distributed.
- Lines 5 and 10 show the parallel sort of Chapter 2 outperforming sequential MATLAB sort.
- Line 12 shows the computation using distributed and frontend structures producing a distributed object. The computation first distributes `v_s` before performing the subtraction.
- Lines 17–22 verify the correctness of the parallel sort.
- Lines 23–35 demonstrate a MATLAB`*p` function that shows both frontend and backend variables and their sizes.

Figure 3-6 contains a function for computing the histogram of an unsorted input vector, given the edge values. Note that the code is quite simple, and makes use of several high-level operations provided by MATLAB and Star-P. Furthermore, nowhere does the code explicitly change its operation if the input is distributed: the parallel operation comes from careful use of overloading basic routines such as `length`, `sort`, concatenation, `find`, and `<=`. Because the number of edges is assumed to be small, the final operation (Line 6) is a computation on the frontend. The simplicity of this implementation has one drawback: by using sorting, its asymptotic running time is $\Omega(\frac{n \lg n}{p})$ in parallel, while a bucketing operation requires $\Theta(n)$ on a single processor. The toy parallel program will not be very fast in comparison, unless p and n are both large.

Figure 3-7 implements a more complex operation, computing the PageRank [4] of a link graph. The code is adapted from that given by Cleve Moler in [18]. The matrix

```
 1: >> n = 10e6;
 2: >> v = rand(n, 1);           Elapsed time is 0.782857 seconds.
 3: >> v_s = sort(v);            Elapsed time is 3.418544 seconds.
 4: >>
 5: >> u = matlab2pp(v, 1)       Elapsed time is 2.173629 seconds.
 6:
 7: u =
 8:
 9:         ddense object: 10000000p-by-1
10: >> u_s = sort(u);            Elapsed time is 1.759546 seconds.
11: >>
12: >> u_s - v_s                 Elapsed time is 2.209233 seconds.
13:
14: ans =
15:
16:         ddense object: 10000000p-by-1
17: >> norm(ans)                 Elapsed time is 0.081045 seconds.
18:
19: ans =
20:
21:      0
22:
23: >> whose
24: Your variables are:
25:   Name       Size             Bytes         Class
26:   ans        1x1              8             double array
27:   n          1x1              8             double array
28:   u          10000000px1      80000000      ddense array
29:   u_s        10000000px1      80000000      ddense array
30:   v          10000000x1       80000000      double array
31:   v_s        10000000x1       80000000      double array
32:
33: Grand total is 40000002 elements using 320000016 bytes
34: MATLAB has a total of 20000002 elements using 160000016 bytes
35: MATLAB*P server has a total of 20000000 elements using 160000000 bytes
```

Figure 3-5: MATLAB*p session demonstrating parallel sorting.

```
1: function binsizes = histc(data, edges);
2:
3: ne=length(edges); nd=length(data);
4: [ignore, perm] = sort([ edges data ]);
5: f = find(perm <= double(ne))';
6: binsizes = diff([ f double(1+nd+ne) ]) - 1;
```

Figure 3-6: A toy histogram function.

```
 1: function x = pagerankpow(G, U)
 2: % PAGERANKPOW  PageRank by power method.
 3: %   G is a 0-1 valued link matrix,
 4: %     where G(i,j) = 1 implies j links to i
 5: %   U is an optional cell array of url strings
 6:
 7: % Link structure
 8:
 9: [n,dn] = size(G);
10: out = (ones(1, n) * G)';     % compute out-degrees
11: czv = (out == 0);
12: out = out + (czv*eps);       % so we don't divide by 0
13:
14: % Power method
15:
16: damp = .85;
17: delta = (1-damp)/dn;
18: x = ones(n,1)/dn;            % start with uniform dist.
19: z = zeros(n,1);
20: while max(abs(x-z)) > .000001
21:    z = x;
22:
23:    % z(i) is the previous round's page rank for page i
24:    % distribute z(i) evenly among all the pages that i links
25:    % x(i) = sum_{j | G(i,j) = 1} z(j) / out(j)
26:    x = G * (z ./ out);
27:
28:    % if a page has zero out degree, its page rank is
29:    % dispersed among everybody
30:    x = x + (dot(z, czv) / dn);
31:
32:    % apply the damping factor
33:    x = damp*x + delta;
34: end
```

Figure 3-7: A function that computes the PageRank vector of a link graph.

G that encodes the graph is intended to be a sparse structure, although the function will also work with dense input. Again, the function has been carefully crafted to take either MATLAB or distributed input. Although it is a design goal for existing sequential code to take parallel inputs with minimal modifications, Star-P does not yet fully implement all the functions of MATLAB.

3.1.3 Design

The Star-P backend server and the MATLAB frontend are two separate components, and in some cases run on separate systems. The two communicate through a single socket connection between the frontend and the root node (rank 0 processor) of the backend. Figure 3-8 sketches a simplified view of the system ([8, 6] contain a more detailed description). One can expect a reasonably fast interconnect between the nodes of a parallel computer (indicated by the thick lines). However, the frontend-backend connection has no guarantees; one may want to use a parallel machine with a laptop connected over wireless Ethernet, for example. Therefore, any parallel code in MATLAB*p must be careful to keep large objects on the same end (either front or back) throughout the computation; using the client-to-server connection easily becomes a bottleneck in this system.

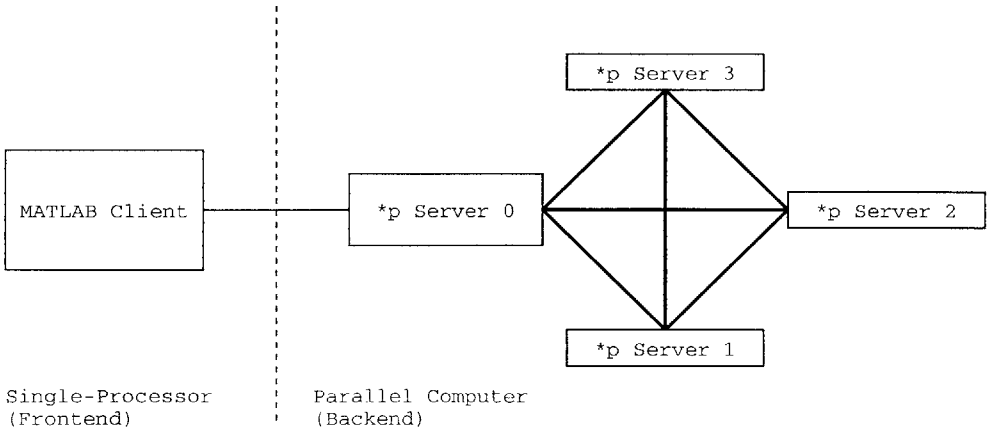


Figure 3-8: High-level sketch of MATLAB*p design.

To understand more concretely the design of MATLAB*p, it is instructive to follow

the execution of a MATLAB side command through the major control components: PPServer, Client Connection Manager, Server Connection Manager, and Package Manager. Figure 3-9 illustrates the control flow through a two-node parallel computer. The arrows are numbered by their relative order of traversal:

1. The user calls a function overloaded by MATLAB*p, which typically performs input checking and then invokes PPClient.

2. PPClient, which uses the MEX [33] interface, blindly shuffles commands (which are strings) and their arguments (data arrays) to the Star-P server and propagates results to the MATLAB side.

3. The root node contains a Client Connection Manager that receives commands over the socket connection with the PPClient. Note that any future work to allow multiple clients to use the same Star-P server would require modifications to only this component.

4. The PPServer serves as a single point to access any of the various components, and also contains the main control loop. The PPServer of the root node receives commands from the Client Connection Manager and passes them to the Server Connection Manager.

5. The command string and its arguments then gets broadcast from the root node to the remaining nodes; MATLAB arguments are copied to each node, distributed arguments are passed by reference.

6. The other PPServers receive the broadcasted command and its arguments.

7. All the nodes in parallel use the Package Manager to find the appropriate handler for the command string, and invoke it.

8. The package functions perform the computation, using the Matrix Manager to access distributed data, and MPI [34] to directly communicate with other nodes. The results or errors return to the MATLAB frontend exactly in the reverse direction.

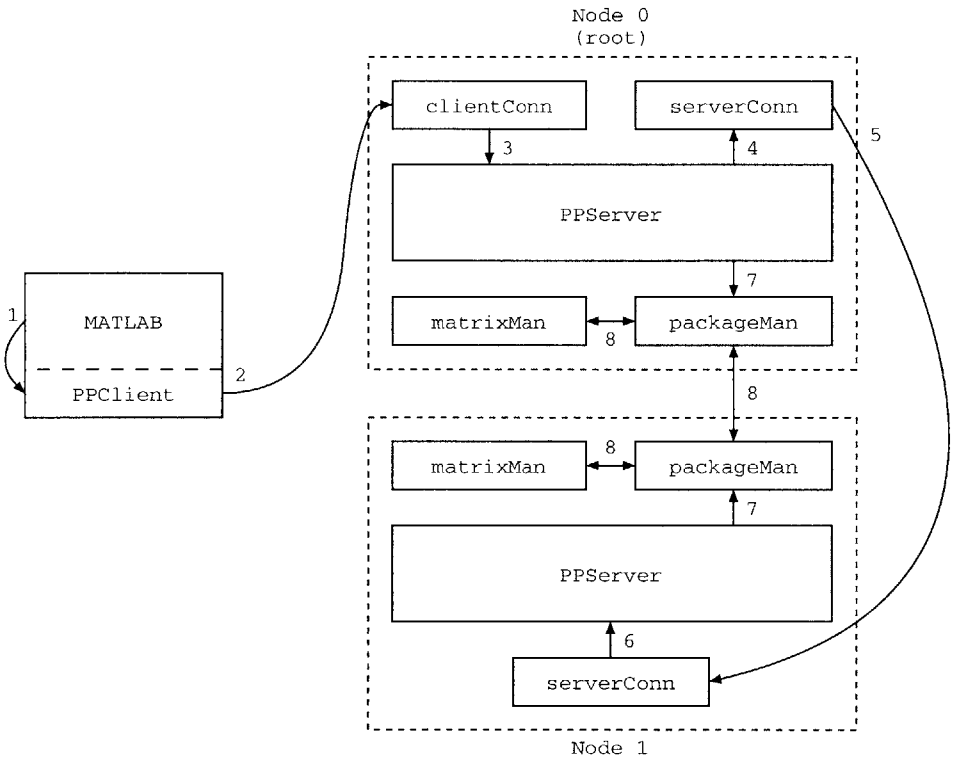


Figure 3-9: The execution of a Star-P command from the frontend.

3.2 Improvements

3.2.1 Extending the System

The simplest way to add functionality to the MATLAB*p system is to make the appropriate modifications in the control path illustrated in Figure 3-9. The section walks through a simple example, of implementing left multiplication of a dense vector and a sparse matrix.

The MATLAB side additions are minimal, as shown in Figure 3-10. Lines 6 and 7 force the input and output vectors to be column-distributed row vectors prior to the `ppclient` call.

```
 1: function c = mtimes(a,b)
 2:
 3: % (omitted)
 4:
 5: if (size(a, 1) == 1) & isa(b,'dsparse')
 6:   changedist (a,2);
 7:   c = ddense(size(a,1), size(b,2), 2);
 8:   [errorcode, errorstr] = ppclient('ppsparse_vecmat',a,b,c);
 9:   return;
10:
11: % (omitted)
12:
```

Figure 3-10: MATLAB operator overloading.

When the command has been broadcast to all nodes, the PPServer on each will invoke the handler for the `'ppsparse_vecmat'` command. This handler is registered at initialization:

```
PPError PPSparse_initialize(PPPackage *pack, PPLogger *Log) {
  ...
  pack->addFunc("ppsparse_matvec", &ppsparse_matvec);
  pack->addFunc("ppsparse_vecmat", &ppsparse_vecmat);
  ...
}
```

The actual algorithm for left-multiplication is rather simple, although inefficient for some inputs. The data layout of the sparse matrix, described in [22], forces the

matrix to be row-distributed. If the dense vector is column-distributed, then the data distributions of the two objects align perfectly. Therefore, a natural algorithm is to for each processor to first perform the computation on its local components, producing partial vectors c_i. The result of the multiplication is just the sum of the c_i, and one round of communication can get this sum distributed correctly (see Figure 3-11). The drawback of this method is that it is not communication efficient: if the sparse matrix is very sparse, the c_i will contain many zeros that are unnecessarily communicated in the reductions.

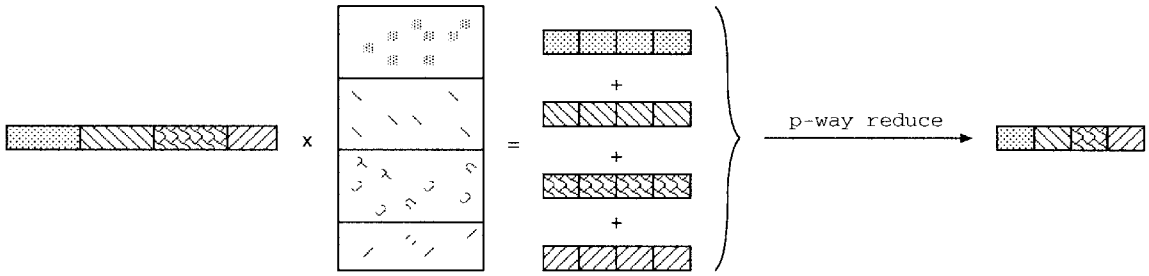


Figure 3-11: Schematic of vector-matrix multiplication.

Figure 3-12 provides the entire code of the algorithm. Note that the actual computation is less than ten lines; the remainder is arguably boilerplate: Lines 4–12 use macros to check the input, Lines 14–26 use the Package Manager to access the distributed objects, and Lines 39–45 just compute the output distribution for the reduction.

Table 3.2: Performance of vector and sparse matrix multiplication.

Operation	$p = 2$	$p = 4$	$p = 8$	$p = 16$
`A*v` (first)	0.497	0.473	0.368	0.552
`A*v` (rest)	0.461	0.300	0.272	0.260
`u*A`	0.452	0.264	0.194	0.181
MATLAB `A*v`	0.700			
MATLAB `u*A`	0.762			

Table 3.2 provides the performance of various multiplication operations (in seconds) averaged over five runs on a distributed-memory "Beowulf" machine [24] made

```
 1: // c = a * B, dense vector times sparse matrix
 2: void ppsparse_vecmat (PPServer &inServer,
 3:                       PPArgList &inArgs, PPArgList &outArgs) {
 4:   // input checking
 5:   CHECK_ARGC(inArgs.length() != 3);
 6:
 7:   PPMatrixID idA=0;
 8:   CHECK_REAL(0); ASSIGN_ID(idA, 0, DENSE);
 9:   PPMatrixID idB=0;
10:   CHECK_REAL(1); ASSIGN_ID(idB, 1, SPARSE);
11:   PPMatrixID idC=0;
12:   CHECK_REAL(2); ASSIGN_ID(idC, 2, DENSE);
13:
14:   // Get the inputs and outputs.
15:   int nproc = (inServer.sCon)->numProcessors();
16:   ELTYPE *dataA = (ELTYPE *) (inServer.mMan)->getData (idA);
17:
18:   PPcsr *Store  = (PPcsr *) (inServer.mMan)->getData (idB);
19:   int_t nnz_loc = Store->nnz_loc;
20:   int_t m_loc   = Store->m_loc;
21:   int_t *rowptr = Store->rowptr;
22:   int_t *colind = Store->colind;
23:   ELTYPE *nzval = (ELTYPE *) Store->nzval;
24:
25:   int_t sizeC   = (inServer.mMan)->getGSize (idC);
26:   ELTYPE *dataC = (ELTYPE *) (inServer.mMan)->getData (idC);
27:
28:   // do the work
29:   ELTYPE *partC = new ELTYPE[sizeC];
30:   fill (partC, partC + sizeC, 0);
31:
32:   int_t cur_row = 0;
33:   for (int_t i = 0; i < nnz_loc; ++i) {
34:     while (i >= rowptr[cur_row + 1]) ++cur_row;
35:     partC[colind[i]] += dataA[cur_row] * nzval[i];
36:   }
37:
38:   // have partial solution, need to add them together
39:   // determine the distribution of output vector C
40:   int_t distC[nproc + 1];
41:   fill (distC, distC + nproc + 1, sizeC); distC[0] = 0;
42:   int_t step = (sizeC + nproc - 1) / nproc;
43:   for (int i = 1; i < nproc; ++i) {
44:     distC[i] = distC[i - 1] + step;
45:   }
46:
47:   // reduce each part
48:   for (int i = 0; i < nproc; ++i) {
49:     MPI_Reduce (&partC[distC[i]], dataC, distC[i + 1] - distC[i],
50:                 MPI_ELTYPE, MPI_SUM, i, MPI_COMM_WORLD);
51:   }
52:
53:   delete [] partC;
54:   outArgs.addNoError ();
55: }
```

Figure 3-12: Implementation of vector-matrix multiplication.

from commodity hardware. The matrix A is an $n \times n$ symmetric sparse matrix with density about 0.001 and $n = 10^6$; v is a column vector and u is a row vector. The parallel `A*v` algorithm, given in [22], caches the sparsity pattern of the matrix to make subsequent calls communication-efficient: each processor receives only the necessary values of the distributed vector. That the left-multiplication algorithm outperforms even the cached calls indicates its relatively good performance.

3.2.2 Distributed Load and Save

One of the design goals of Star-P is to keep large objects distributed over the parallel computer. For testing purposes, it is convenient to have the server simply generate large random matrices without using the frontend. However, practical applications rarely use large amounts of generated data; even Monte Carlo methods would not use large random matrices, as doing so would typically waste space and inflate runtime. Applications need to input large data sets from disk.

Functionality

In an effort to again minimize the frontend to backend connection, the data is loaded and saved directly by the Star-P server[2]. Because MATLAB is currently the only frontend, this new PPIO package only handles the MATLAB MAT file format [32]. However, the MAT file is quite reasonable: for each matrix, the file first lists some metadata (data type, real or complex, dimensions) and then lays out the values in a linear fashion. Future extensions to PPIO can create new metadata parsing functions while reusing much of the data reading and writing methods.

A MAT file holds the matrix data in column-major form, and column-distributed matrices store the local data also in column-major form. Therefore, it is relatively straightforward to implement the load and save of column-distributed matrices. For simplicity, other distributions are not handled natively: they are either redistributed to or from the column distribution as necessary.

[2]We assume the presence of a shared filesystem between the nodes of the parallel machine

Table 3.3: Distributed load and save times on a 16-node Beowulf

Operation	$p = 2$	$p = 4$	$p = 8$	$p = 16$
`matlab2pp`	6.088	6.260	6.149	6.189
`dload`	3.118	2.318	2.578	2.600
MATLAB `load`	0.810			
C++ `fread`	0.620			
`pp2matlab`	6.079	6.002	5.979	6.097
`dsave`	12.725	13.645	13.344	10.612
`dsave` (p files)	8.168	7.769	8.530	8.727
MATLAB `save`	6.104			
C++ `fwrite`	1.880			

Table 3.4: Distributed load and save times on an 8-processor Altix

Operation	$p = 2$	$p = 4$	$p = 8$
`matlab2pp`	6.483	6.159	5.975
`dload`	3.547	1.806	0.975
C++ `fread`	0.264		
`pp2matlab`	5.780	5.462	5.287
`dsave`	6.060	6.245	6.140
`dsave` (p files)	3.113	1.774	1.161
C++ `fwrite`	0.432		

Timings

The distributed load and save operations were tested on two types of machines: a distributed-memory "Beowulf" machine made from commodity hardware [24], and a shared-memory SGI Altix [36]. Tables 3.3 and 3.4 give the timings (in seconds elapsed) for both machines working with a column-distributed matrix of dimensions 3×10^7.

The performance on the Beowulf is rather poor. The `matlab2pp` and `pp2matlab` times are mostly unaffected by p, and reflect the bandwidth utilization of the frontend to backend connection. However, the `dload` times do not scale as expected with p. The main cause of the slow read time, and gradual worsening for larger p is that the the filesystem is shared over the network, using NFS. The NFS server is one of the machines in the computation, therefore making one machine's work local; as this

amount decreases with larger p, we see a growing bottleneck from the network. For `dsave`, the performance is even worse, as each processor blocks on the previous one so the data is written out sequentially. For debugging purposes, the distributed save also has the option of producing p separate files, each processor independently saving its local data. However, even this "embarrassingly parallel" operation is slow, primarily because of network contention from the p simultaneous write requests.

The distributed performance on the Altix is much better, due to the more carefully engineered disk usage among the processors. Note the significant speedup with larger p, as each processor performs fewer low-level operations (byte swapping and padding) and the hard drive connection remains relatively free of congestion. Because MATLAB does not exist for the architecture, the only points of comparison are the C++ operations as a best-case, or the sum of a Beowulf MATLAB `load` (or save) and `matlab2pp` (or `pp2matlab`).

3.2.3 Shared Memory

Focusing mostly on scientific computation, Star-P is just one of many tools for utilizing a parallel computer. Other tools have different specialties, such as SCIRun for visualization [19]; it is often valuable to use them together as complements. The ideal interaction of various large projects would be to avoid the filesystem altogether, for one process to hand off its data seamlessly over to the other via shared memory. There are a few engineering challenges associated with this task, and this section describes approaches taken for one side: data export from Star-P through shared memory.

Design Considerations

The principal concern with shared memory is that it must be known to be shared at allocation time; that is, one can not simply declare an existing section of memory to become shared. There exist several alternatives to performing a memory copy.

One solution, the simplest, is to just use shared memory for everything. When the user wants to export a matrix, the server simply needs to flip some permission bits

Table 3.5: Shared memory performance with 10^9 bytes on an Altix.

Operation	Shared	Non-Shared	Percent Difference
Allocate	0.000034	0.000040	17.86
Write	5.953	5.638	5.598
Read	6.339	6.337	0.030
De-allocate	0.089	0.035	157.6

on the shared memory segment. Unfortunately, this scheme is inefficient: there is a cost associated with using shared memory. Table 3.5 provides the execution times (in seconds), averaged over five runs, for various memory operations. With the allocation and de-allocation times negligible, there remains a nontrivial inflation in the write time. Given that exporting through shared memory is a relatively rare event, this cost is quite high.

Another option is to declare a variables to be shared, and maintain its shared property throughout any computation or reassignment (similar to the function of the `global` operator in MATLAB). For example, if v is in shared memory, then it will still be in shared memory after executing the out-of-place operation $v = A \cdot v$. On the other hand, $u = A \cdot v$ will not place u in shared memory. The main drawback for this method is that it is technically challenging, and may violate the Star-P principle that the functionality should be independent of the frontend. When MATLAB evaluates `u = A * b`, it first evaluates the multiplication and then the assignment, in two steps. Therefore, any method that distinguishes between `v = A * b` and `u = A * b` would be heavily MATLAB dependent.

The approach we take is a compromise between the two options above: we introduce a global flag, `AllocateShared`, that enables (or disables) the allocation of all new objects in shared memory segments. The intended use is to form a block of code where `AllocateShared` is enabled, and export values computed inside this block. The actual export function produces a $p \times 5$ matrix, where each processor produces one row; the columns specify the shared memory id to be attached by the importing application, the offset of the data, the local dimensions, and the distribution.

Table 3.6: Exporting an n-element vector with `AllocateShared = false`.

Operation	$n = 10^5$	$n = 10^7$	$n = 5 \cdot 10^7$
`v = v*A; v = v/norm(v);`	0.158	2.130	11.313
`ppexport(v)`	0.036	0.127	0.704

Table 3.7: Exporting an n-element vector with `AllocateShared = true`.

Operation	$n = 10^5$	$n = 10^7$	$n = 5 \cdot 10^7$
`v = v*A; v = v/norm(v);`	0.231	2.237	11.788
`ppexport(v)`	0.021	0.020	0.022

Results

The bulk of the implementation dealt with subclassing the distributed matrix objects to alter the method of memory allocation. The `AllocateShared` flag is a constant on the Star-P side and is exposed to the MATLAB frontend; the Matrix Manager looks up its value (in a hash table) before creating an instance of any matrix.

Tables 3.6 and 3.7 provide empirical data for executing two lines inside and outside an `AllocateShared` block, on 4 processors of an Altix. We see that the export time without the copy is steady at around 20 ms, while the export time grows with n when copying. When the data is small, the overhead of performing the computation entirely in shared memory does not offset the benefit of having a cheap export. However, when exporting large objects (such as a dense matrix), it may be more beneficial to set the flag to avoid the copy.

3.3 Discussion

This chapter details the method of several improvements made to the Star-P system. Some future efforts could be made in the direction of implementing the operations described here for some of the other distributions. Further interesting extensions include subscripted indexing and assignment, and reducing the amount of unnecessary communication or memory copies, particularly in stencil operations.

Chapter 4

Tree Flattening

One long standing goal of parallel computing is that of automatic parallelization: given a sequential algorithm (or implementation), invoke a method that allows it to execute more efficiently on a parallel computer than on a single processor.

This problem is long researched, with results ranging from systems to theory. In this chapter, we take an approach that is more theoretical in nature: we assume the existence of sophisticated systems for the parsing and execution of parallel programs. The goal is to examine a tree that represents the computation being performed, and determine how well a parallel computer can evaluate it.

We draw our motivation from studying the well-known Parallel Prefix algorithm. In particular, we examined how the Parallel Prefix algorithm could be derived, and then generalized to different problems.

We develop the notion of tree rotation as an operation that alters the method of computation without affecting the result. We give several preliminary algorithms that use tree rotations to determine a good parallelization of a problem. Note that this task is more general than that of directly performing a computation in parallel. Finally, we state interesting observations that point to further uses of tree rotations in automatic parallelization.

4.1 Basics

Computation Tree

We can naturally represent any computation involving a single binary operator as a binary tree. Each node holds a value, a leaf is an input value, and an internal node is computed from its two children. In many cases, the computation produces a single output, the value computed at the root. An example, given by Figure 4-1, is computing the sum of four values. The extra edge at the root indicates that the value is an output. In other situations, the goal is to produce an output from each internal node. Figure 4-2 gives the example of the prefix operation, that of computing the sum of the first k values, for $k = 1 \ldots n$.

We define the height of a node u to be the maximum number of nodes in the path from node u to a leaf. The height of a leaf is therefore 1, and for convenience we define a nonexistent node to have height 0.

Tree Rotations

The tree rotation is a well-known operation on binary trees, illustrated in Figure 4-3. In the figure, nodes a, b, and c are the roots of arbitrary subtrees. Converting tree T to T' is sometimes referred to as a right rotation on node 2 (or a zig), and converting tree T' to T is a left rotation on node 1 (or a zag). Alternatively, we can say that rotating up node 1 converts tree T to T', and rotating up node 2 does the reverse.

The important properties of this operation are that it can be executed in constant-time, and that it preserves the inorder traversal.

Deriving an Efficient Parallel Algorithm

For any computation tree T with n leaves and height h, there exists a straightforward PRAM algorithm that can compute the value of each output node in $\Theta(h)$ time with $O(n)$ processors. This basic algorithm begins by assigning each internal node to a different processor. In each round, each processor checks if values are available for the children of its node; if so, it applies the operator and is done. After $h - 1$ such

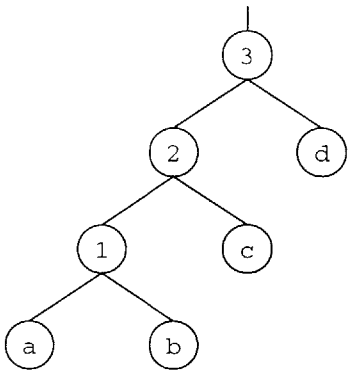


Figure 4-1: Tree representation of $(((a + b) + c) + d)$.

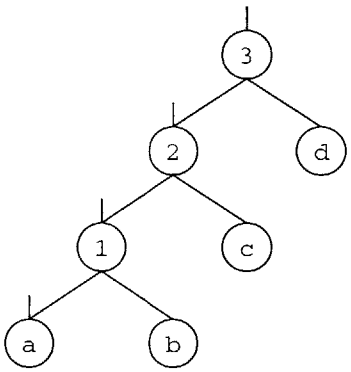


Figure 4-2: Tree that represents the prefix computation.

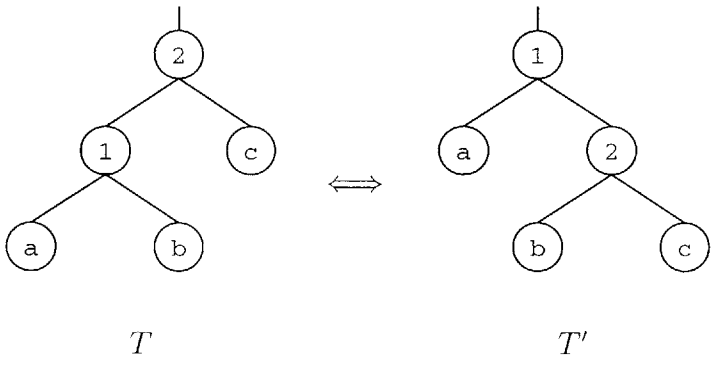


Figure 4-3: Left and Right tree rotations.

rounds, each internal node will have been computed.

Therefore, the height of the computation tree determines the parallel computation speed. In focusing solely on computation trees, we make three simplifying assumptions in this study: that the operator is associative, that this computation tree can be efficiently procured in practice; and that communicating one value between two processors takes constant time (independent of what the other processors may be doing).

With an associative operator, the parenthesization is irrelevant: $a + (b + c) = (a + b) + c$. Note that these two expressions are represented exactly by the two trees given in Figure 4-3. A computation tree can be subjected to any sequence of rotations without the root value being affected.

We now present algorithms to reduce the height of various computation trees.

4.2 Tree Flattening Algorithms

4.2.1 Greedy Algorithm

Our first algorithm does reduce the height of any computation tree to $O(\lg n)$, but is rather inefficient.

Algorithm Greedy Flatten.
Input: A tree T with n leaves and height h.
Output: A tree T' with the same inorder traversal as T and height $O(\lg n)$.

1. For each node u with parent p and grandparent g,
   (a) Check if rotating up node u will reduce the height of the subtree rooted at node p; if so, perform the rotation.
   (b) Otherwise, check if rotating up node u twice will reduce the height of the subtree rooted at node g; if so, perform the double rotation.
2. Repeat Step 1 until no more rotations are performed.

Note that Step 1 could have alternatively been phrased as: perform a zig, a zag, a zig-zag, or a zag-zig such that the resulting subtree is reduced in height.

Running Time

The algorithm may have to test each node before it determines the existence (or non-existence) of a rotation that strictly reduces the height of a subtree. Therefore, each round of the algorithm takes $O(n)$ time. To get a loose upper bound on the number of rounds, note that each round will reduce the height of at least one node by 1, and that each node has height at most h. Therefore, the running time is $O(n^2 h) = O(n^3)$ using the fact that $h = O(n)$.

Correctness

We now show that the algorithm does reduce the height to $O(\lg n)$. The proof for correctness uses two lemmas, the first one stated without proof:

Lemma 4.2.1. *For any three values α, β, γ, the inequality*

$$\max(\alpha, \beta + 1, \gamma + 1) \leq \max(\alpha + 1, \beta + 1, \gamma)$$

is satisfied if and only if the following relation holds:

$$(\beta \geq \max(\alpha, \gamma)) \quad \vee \quad (\alpha = \gamma) \quad \vee \quad (\alpha \geq \max(\beta, \gamma) + 1)$$

This relation is also equivalent to:

$$(\beta \geq \max(\alpha, \gamma)) \quad \vee \quad (\alpha \geq \gamma)$$

Proof. This lemma can be verified by case analysis (not shown). □

To aid this discussion, we consider the trees T, T_L, and T_R given in Figure 4-4. For simplicity of notation, we use a variable u to denote both the node u and its height. Therefore, tree T has the properties $x = \max(a, b) + 1$ and $y = \max(c, d) + 1$. Note that nodes a, b, c, and d are not necessarily leaves, and may have any non-negative height. We can impose this structure T at the root of any node u in an arbitrary binary tree: doing so may require some nodes to have zero "height."

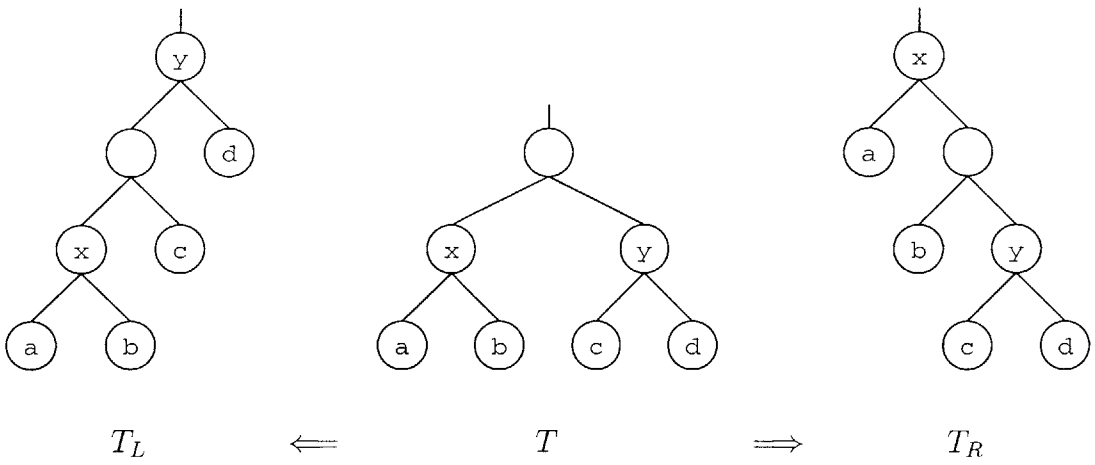


Figure 4-4: Tree T and its left and right rotations T_L and T_R.

Lemma 4.2.2. *When there exist no single or double rotations that strictly decrease the height of a subtree, one of these conditions must hold:*

$$\begin{array}{c} (x = y) \\ \vee \quad ((x \leq (c = d)) \quad \vee \quad ((d \leq (x = c)) \quad \vee \quad ((c \leq (x = d)) \\ \vee \quad ((y \leq (a = b)) \quad \vee \quad ((b \leq (y = a)) \quad \vee \quad ((a \leq (y = b)) \end{array}$$

Where the $(x \leq (c = d))$ *is shorthand to mean* $((x \leq c) \wedge (c = d))$.

Proof. First, we examine what conditions must hold if the algorithm has stopped making single rotations.

By inspection, we can glean expressions for the heights of the three trees:

$$\begin{aligned} \text{height}(T) &= \max(x, y) + 1 \\ \text{height}(T_L) &= \max(x + 1, c + 1, d) + 1 \\ \text{height}(T_R) &= \max(a, b + 1, y + 1) + 1 \end{aligned}$$

Therefore, a left rotation will not occur only when $\max(x, y) \leq \max(x + 1, c + 1, d)$, or:

$$\max(x, c + 1, d + 1) \leq \max(x + 1, c + 1, d) \tag{4.1}$$

Similarly, a right rotation will not occur only when

$$\max(a+1, b+1, y) \leq \max(a, b+1, y+1) \tag{4.2}$$

We can invoke Lemma 4.2.1 to derive the following two sets of conditions from Equations 4.1 and 4.2, respectively:

$$(c \geq \max(x, d)) \quad \vee \quad (x = d) \quad \vee \quad (x \geq \max(c, d) + 1) \tag{4.3}$$

$$(b \geq \max(a, y)) \quad \vee \quad (a = y) \quad \vee \quad (y \geq \max(a, b) + 1) \tag{4.4}$$

We want to determine the conditions that hold if and only if there exist neither left nor right rotations that will reduce the height. For this, we can compute the logical AND of Equations 4.3 and 4.4; by distributing the AND, we have nine pairwise ANDs that simplify to:

$$\begin{array}{cccc} & & (x = y) & \\ \vee & (c \geq \max(x, d)) & \vee & (x = d) \\ \vee & (b \geq \max(a, y)) & \vee & (a = y) \end{array} \tag{4.5}$$

Now we focus on the stopping conditions for the double rotations. Again, we base the discussion on the layout of nodes given by tree T in Figure 4-4. Note that if a double rotation of a or d reduces the subtree height, then a single rotation will also reduce the height. Therefore, we only have to consider the double rotations of b and c. Figure 4-5 illustrates rotating node b up two times (the oval has been added to indicate the rotated nodes, and the children of node y have been omitted). Similarly, Figure 4-6 demonstrates a double rotation of node c. The height of the trees T_b and T_c are:

$$\begin{aligned} \text{height}(T) &= \max(x, y) + 1 = \max(a+1, b+1, y) + 1 = \max(x, c+1, d+1) + 1 \\ \text{height}(T_b) &= \max(a+1, b_1+1, b_2+1, y+1) + 1 = \max(a+1, b, y+1) + 1 \\ \text{height}(T_c) &= \max(x+1, c_1+1, c_2+1, d+1) + 1 = \max(x+1, c, d+1) + 1 \end{aligned}$$

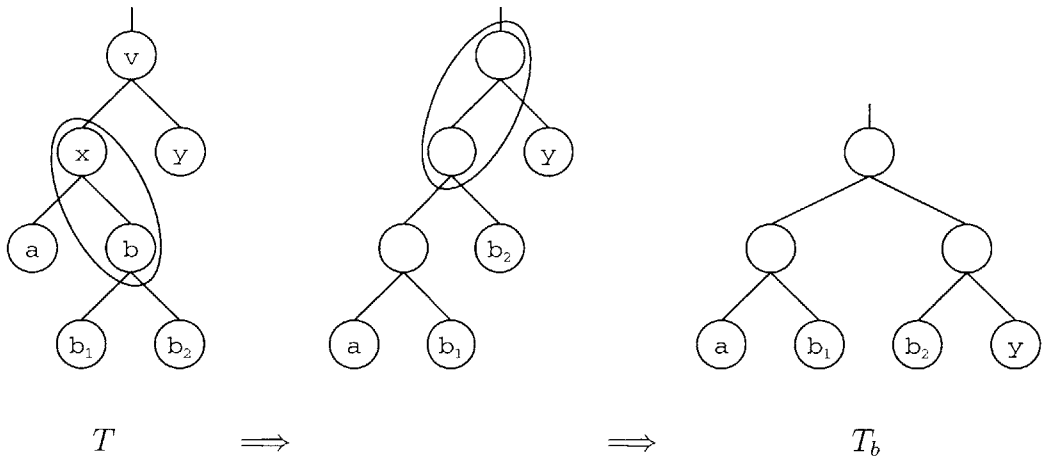


Figure 4-5: Rotating up node b twice through Tree T (a zig-zag rotation on node v).

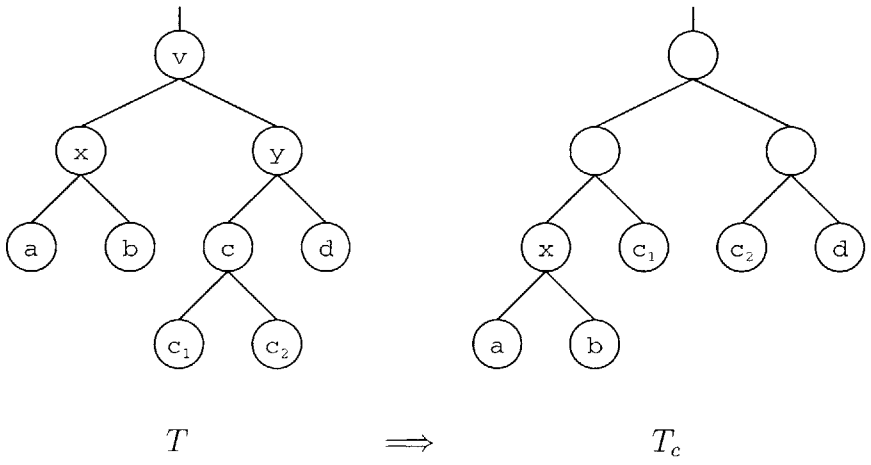


Figure 4-6: A double rotation of node c (a zag-zig rotation on node v).

Here, we use $b = \max(b_1, b_2) + 1$ and $c = \max(c_1, c_2) + 1$. The rotation illustrated in Figure 4-5 will not happen iff:

$$\begin{aligned} & \max(y, a+1, b+1) \leq \max(y+1, a+1, b) \\ \Rightarrow \quad & (a \geq \max(y, b)) \vee (y \geq b) \end{aligned} \tag{4.6}$$

The implication uses Lemma 4.2.1. Similarly, the rotation of Figure 4-6 will not happen iff:

$$\begin{aligned} & \max(c+1, d+1, x) \leq \max(c, d+1, x+1) \\ \Rightarrow \quad & (d \geq \max(x, c)) \vee (x \geq c) \end{aligned} \tag{4.7}$$

The logical AND of Equations 4.5, 4.6, and 4.7 will produce the desired set of constraints. □

Theorem 4.2.3. *If no rotations or double rotations can be performed on a tree T so that the local height is strictly reduced, then T has height at most $2 \lg n + 1$ where n is the total number of leaves.*

Proof. We use strong induction to prove the claim. Again, we consider the tree T in Figure 4-4. We also use U to denote the number of leaves in the subtree rooted at node u.

For the inductive step, Lemma 4.2.2 provides us with the following cases:

1. $x = y$

   By the induction hypothesis, we have:

$$\begin{aligned} x &\leq 2 \lg X + 1 \\ y &\leq 2 \lg Y + 1 \end{aligned}$$

   By definition, we have

$$X + Y = n$$

Without loss of generality, we assume $X \geq Y$.

We want to show that:

$$\begin{aligned} \text{height}(T) = y + 1 &\leq 2 \lg n + 1 \\ \Rightarrow 2 \lg Y + 2 = 2 \lg(Y \cdot 2) &\leq 2 \lg n \\ \Rightarrow 2Y &\leq X + Y \\ \Rightarrow Y &\leq X \end{aligned}$$

which is true. The first step uses $x = y \leq \min(2 \lg X + 1,\ 2 \lg Y + 1)$.

2. $y \leq (a = b)$

   We are given the following relations, by induction and definition:

$$\begin{aligned} a &\leq 2 \lg A + 1 \\ b &\leq 2 \lg B + 1 \\ y &\leq 2 \lg Y + 1 \\ A + B + Y &= n \end{aligned}$$

   Assume without loss of generality that $A \leq B$.

   We want to show that:

$$\begin{aligned} \text{height}(T) = a + 2 &\leq 2 \lg n + 1 \\ \Rightarrow 2 \lg A + 2 = 2 \lg(2A) &\leq 2 \lg n \\ \Rightarrow 2A &\leq n \end{aligned}$$

   To show this, note that $2A = A + A \leq A + B \leq A + B + Y = n$.

   The case of $x \leq (c = d)$ is analogous.

3. $b \leq (y = a)$

   Again, we are given the same set of relations as in the case above. Now assume (without loss of generality) that $A \leq Y$.

   We want to show that:

$$\begin{aligned} \text{height}(T) = a + 2 &\leq 2\lg n + 1 \\ \Rightarrow 2\lg A + 2 = 2\lg(2A) &\leq 2\lg n \\ \Rightarrow 2A &\leq n \end{aligned}$$

   Which is true because $2A \leq A + B \leq n$.

   The cases of $a \leq (y = b)$, $d \leq (x = c)$, and $c \leq (x = d)$ all lead to analogous arguments.

For the base case, consider a leaf of height 1 at the root of tree T. Then the claim holds with $1 \leq 2\lg(1) + 1$. □

4.2.2 Top-Down Algorithm

We now present a more efficient form of the greedy algorithm given above.

Algorithm Top-Down Flatten.
Input: A tree T with n leaves and root v.
Output: A tree T' with the same inorder traversal as T and height $O(\lg n)$.

1. Perform zig, zag, zig-zag, and zag-zig rotations on v until no such operation exists that will strictly decrease the height of T.
2. Recurse on the left and right children of the root node.
3. Invoke the Greedy Flatten algorithm on T.

Correctness

That the algorithm does flatten the tree follows from the third step, and the correctness of the greedy algorithm.

Running Time

The height of a tree is bounded by the number of its leaves. By Lemma 4.2.2, Step 1 will produce a node whose two children have heights that differ by at most 1. The guarantee bounds the height of T after Step 1 to be at most $\lceil \frac{n}{2} \rceil + 2$ – that is, a function of the number of leaves. However, we have no good guarantee about how evenly the leaves will be split among the two children. Therefore, an upper bound on the running time is given by the recurrence:

$$\begin{aligned} T(n) &= n + T(n - \lg n) + O(\lg^3 n) \\ &\leq n + T(n-1) + O(\lg^3 n) \\ &= O(n^2) \end{aligned}$$

The first linear term comes from the fact that each rotation in Step 1 reduces the global height of the tree, and the tree begins with at most height n. The second recursive term provides the worst case distribution of leaves among the children. However, note that if this "worst case" actually happens, then the algorithm is finished: the tree has height $O(\lg n)$. The third term is the cost of the greedy algorithm.

This analysis is not tight, and it is conceivable that the algorithm can run in $O(n \lg n)$ time.

4.2.3 Handling Multiple Outputs

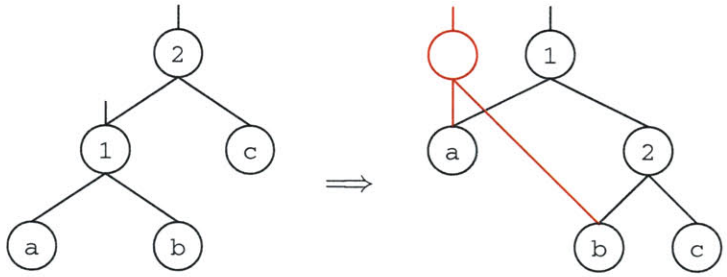
Stepping back, the algorithm presented so far will parallelize a single-output computation (where the single output is at the root). This section describes how to adapt the rotation operation so that computation trees with output nodes other than the root can also be parallelized.

Value-Preserving Rotation

The key is to use a modified rotation operation, so the value at the node being rotated will still be computed. This involves the addition of a new node and two new edges,

which we shall call augmenting. The tree with these augmentations now a computation DAG, where each node has in-degree exactly two, and arbitrary out-degree (one can imagine directed edges always upward oriented, from inputs to outputs). However, the same parallel evaluation algorithm may be used on this DAG as on a tree; the difference is that the number of processors necessary is now O(number of nodes).

Figure 4-7 illustrates a right rotation, with the red color indicating the augmenting graph structure originating from the rotation; the left rotation, not shown, is analogous.



T T'

Figure 4-7: Right rotation on a tree with two output values.

Example: Parallel Prefix

Recall that Figure 4-2 gives the computation tree for the straightforward for-loop implementation of the prefix operation. Figure 4-8 extends the computation tree to eight inputs, and demonstrates that there exists a sequence of modified rotations that derives exactly the Parallel Prefix algorithm. The final structure is visually satisfying, as it illustrates the recursive nature of parallel prefix: we see the solution for the $n = 4$ case repeated twice, with the root of the first instance added to every output node of the second.

In the construction, note how the last rotation destroys a node that is necessary to compute the seventh output value. Therefore, the subtree rooted at this output node acquires an augmenting node. Note that in general, this extra node may increase the height of the output computation; for example, consider a right rotation of the root

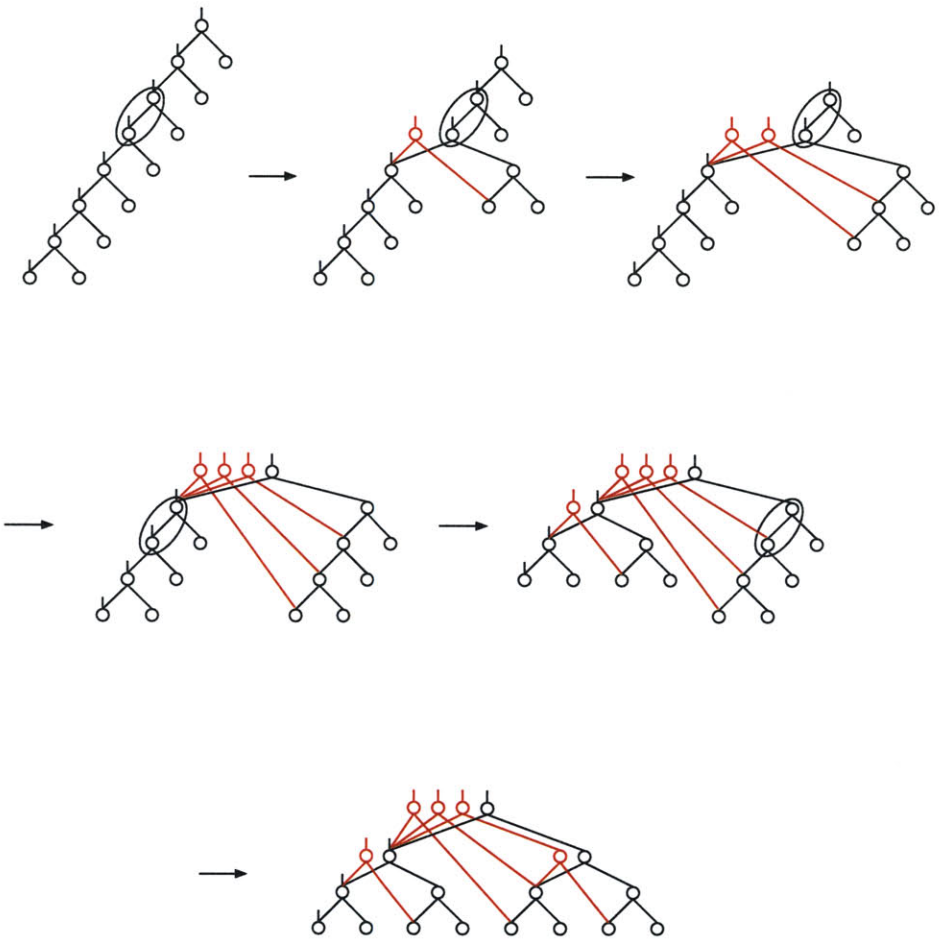


Figure 4-8: Derivation of Parallel Prefix for $n = 8$.

(eighth output node). It is possible for a sequence of k modified rotations to produce an augmented DAG with a path length $O(k)$.

Algorithm

The goal is to flatten a computation tree with multiple output nodes to a computation DAG whose longest path is $O(\lg n)$.

The algorithm is rather simple, based on that given in the previous sections. We invoke the top-down tree flattening algorithm on the tree as usual, but use the modified tree rotations. Throughout the execution of the algorithm, we ignore any augmenting structure (other than create it as dictated by the rotations).

Observe that we have at most n output nodes, and each of these is initially the root of a subtree. When the height of one of these subtrees is increased by a modified rotation, note that the new augmenting structure replaces a leaf. Furthermore, the new augmenting node always takes its inputs from non-augmenting nodes; the modified tree rotations are not performed on augmenting structure. Therefore, every node in the augmenting structure rooted by an output node has exactly one outgoing edge and two incoming edges; the augmenting structure is a tree, with leaf nodes in the non-augmented tree. It may be possible for the leaves to be not unique; in this case, we can split the leaves as shown in Figure 4-9. The dotted line indicates that the two nodes are equivalent (have the same value).

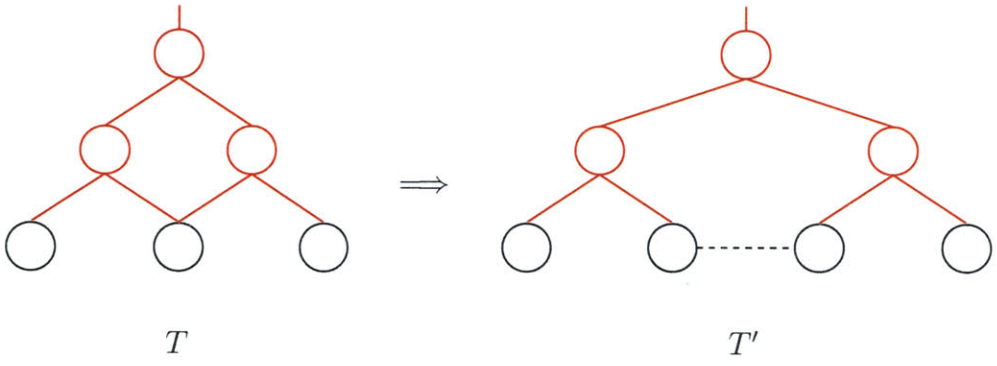


Figure 4-9: Splitting a leaf node of the augmenting tree.

Thus, each output node is the root of a proper tree of augmenting nodes, with

input values being nodes of the flattened (non-augmented) tree. We can therefore decompose the augmenting structure into at most n trees, and invoke the top-down flattening algorithm on each. The maximum height of an output node is then:

$$(2 \lg n + 1) + (2 \lg O(n^2) + 1) = O(\lg n)$$

The first term is the height of the flattened input tree (disregarding the augmentations); after this height, all the inputs to the augmented structure must be computed. The second term is the flattened height of an augmented tree; the $O(n^2)$ is a worst-case height coming from the $O(n^2)$ time of the top-down flattening algorithm.

An upper bound on the computation time for this algorithm is now $O(n^4)$, as we may have to flatten $O(n^2)$ nodes in the augmented tree (note that this bound is quite loose). It would take $O(n^2)$ processors to evaluate the final computation DAG in $O(\lg n)$ time.

4.3 Interesting Extensions

These tree rotation algorithms as presented above are not entirely practical: if takes $O(n)$ time to just construct the tree, and longer to flatten it; it may be faster to simply perform the computation directly. Furthermore, the task of computing one or multiple output values has been studied in the past with stronger results. This section will note a few extensions that strengthen the tree flattening results given above.

4.3.1 Towards a Practical Solution

In practice, one often has a fixed number of processors p, and an input that n that varies. The standard method of accommodation is to use the p processors to simulate the work of however many are necessary by the algorithm. However, here we propose an alternative solution.

We define a tree T to exhibit a period τ if all subtrees of height τ are isomorphic

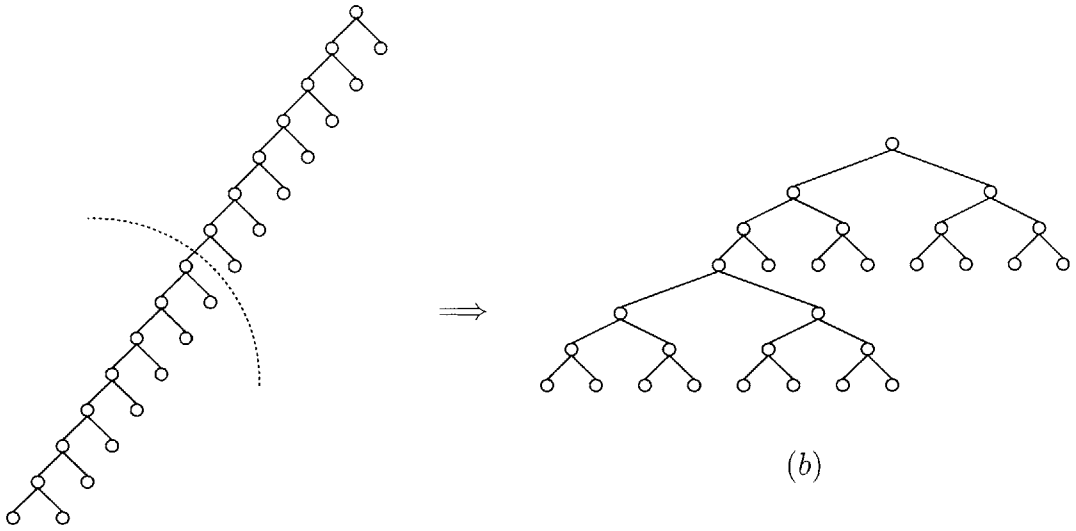


Figure 4-10: Forming a cut, flattening it, and reusing the solution.

---

in structure (modulo edges incident to nodes outside the subtree). For example, the sum and prefix trees of Figures 4-1 and 4-2 exhibit a period $\tau = 2$.

Let us assume that a sophisticated compiler has constructed a function f that takes in n and other necessary parameters, and can be called at runtime to produce the computation tree. Furthermore, let us assume the compiler has also constructed a function g that takes in the same parameters as f and determines the minimum period of the computation tree that f will produce.

With these two black boxes, our goal is to perform the computation faster on a parallel computer than on a single processor. We do this by first invoking g to determine some $k = \max(\tau, p)$, then invoking f to produce a tree T_k with k leaves. Thus, T_k is large enough to keep the p processors busy, while exhibiting the periodic structure of the full tree T. The tree T_k is a cut of the complete tree T, containing the first k leaves. We designate the nodes of T_k incident to edges in the cut to be output nodes, and then proceed to flatten T_k into T'_k. Now we memoize this flattened tree, and use it as necessary to complete the intended computation. Figure 4-10 gives an example for the sum computation, where only one edge is crossing the cut (shown in dotted lines).

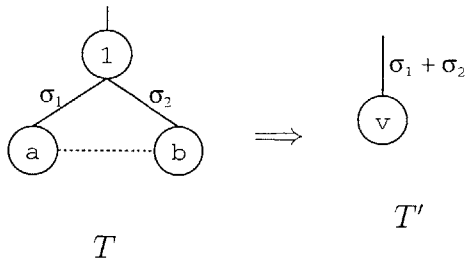


Figure 4-11: Simplification when two identical nodes are siblings.

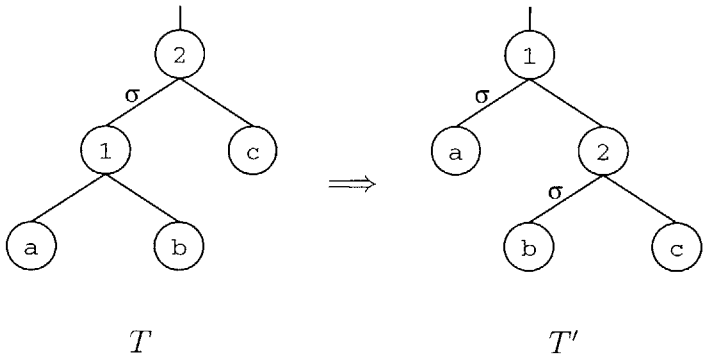


Figure 4-12: Rotating an edge that carries a scaling constant.

In the case where the complete tree T does not split exactly over T_k, we just find the parallelization again for the last part, whose size is strictly less than k. The total computation time is $O(k^4) + O(\frac{n}{k} \lg k)$ with p processors. The first term comes from using the multi-output flattening algorithm, and the second comes from the fact that T has become split into $O(\frac{n}{k})$ subtrees connected sequentially, each of height $O(\lg k)$. Note that for problems such as sum or prefix, $k = p$.

4.3.2 Solving More Difficult Problems

We can make further use of node splitting, to parallelize computation DAGs. The idea is to first convert the DAG into a tree, and then apply the appropriate flattening algorithm. However, one difficulty is in dealing with the split nodes; for example, two equivalent nodes can not be directly connected by one edge.

One solution for this difficulty is to take an operator-specific approach. If the binary operator used in the computation tree is addition, then we can use another specialized rotation rule that allows for constant-time scaling by a constant. Each

edge holds a constant, initially set to 1; as the edge connects one output node to another as input, it will now also scale the output by its constant. Figure 4-11 illustrates how a scalar of value greater than 1 comes about, and Figure 4-12 gives the rotation operation for these edges. Note that tree T in Figure 4-12 represents the computation $(a + b) \cdot \sigma + c$, and its rotation $a \cdot \sigma + (b \cdot \sigma + c)$ is equivalent to distributing the scalar.

However, now with these two additions, there are no guarantees about the performance of the flattening algorithms given in Section 4.2. We shall present sequences of rotations and simplifications as an indication of existence, but it remains a future study to develop flattening algorithms that also handle split nodes.

Example: Fibonacci Numbers (Dynamic Programming)

With this extension, we can flatten a computation DAG that represents the dynamic programming method of computing the nth Fibonacci number, F_n. Figure 4-13 gives an example computation DAG, and the tree formed by node splitting; the inputs are given at the leaves, and the values of each node are given.

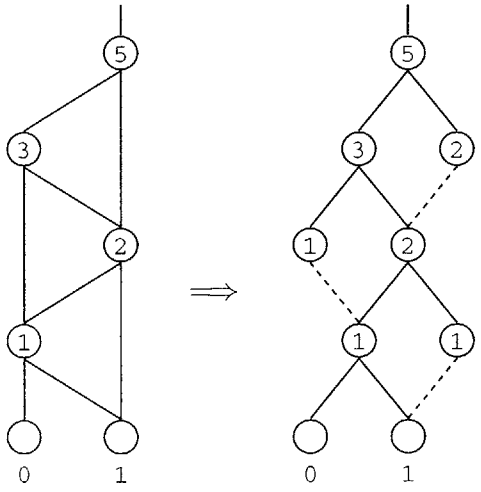


Figure 4-13: Computation DAG and expanded tree for Fibonacci computation.

Using the operations described in Figures 4-11 and 4-12, we can perform a sequence

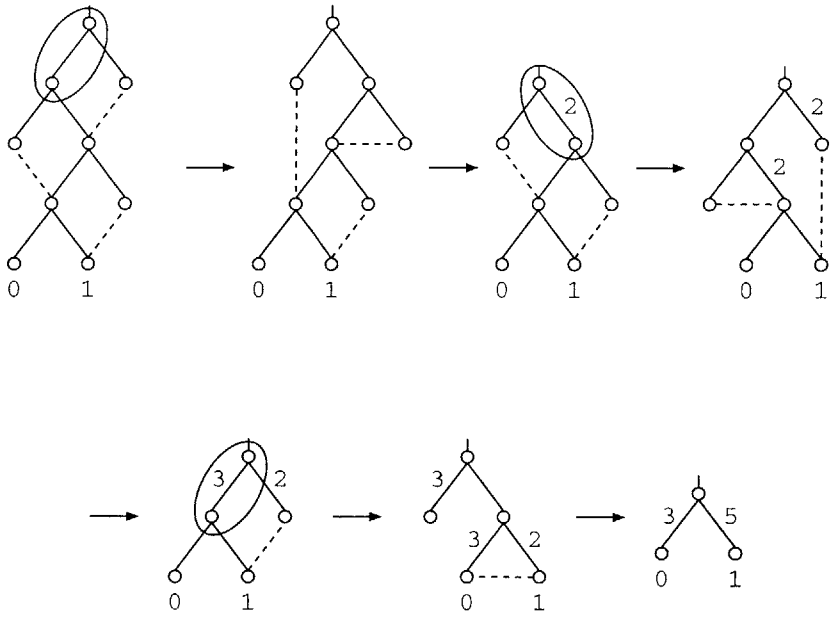


Figure 4-14: Flattening Fibonacci computation.

---

of rotations on the Fibonacci computation tree T to produce a flattened tree T' of height 2 (see Figure 4-14). Note that the flattening algorithm has unwittingly computed the nth Fibonacci number in its "parallelization."

While this effort is not impressive on its own, it can be used in conjunction with the ideas given in Section 4.3.1. In particular, assume that the compiler has determined that the computation tree exhibits a period $\tau = 2$. At run time, the system automatically flattens a truncated computation DAG of height $O(p)$, and then uses it to compute the pth Fibonacci number in $O(\frac{n}{p})$ time.

Figures 4-15 and 4-16 give an example with $p = 5$, $n = 9$. First, the algorithm makes a cut to determine the output nodes, and then it proceeds to flatten the augmented tree. The algorithm determines that for leaf values a and b, the Fibonacci computation DAG of height 5 will produce outputs $2a + 3b$ and $3a + 5b$. With this information, the Fibonacci numbers are calculated as given in Table 4.1.

In this method, some more general problems can be solved. For example, a computation DAG that exhibits repeated structure can be formed for any recurrences of

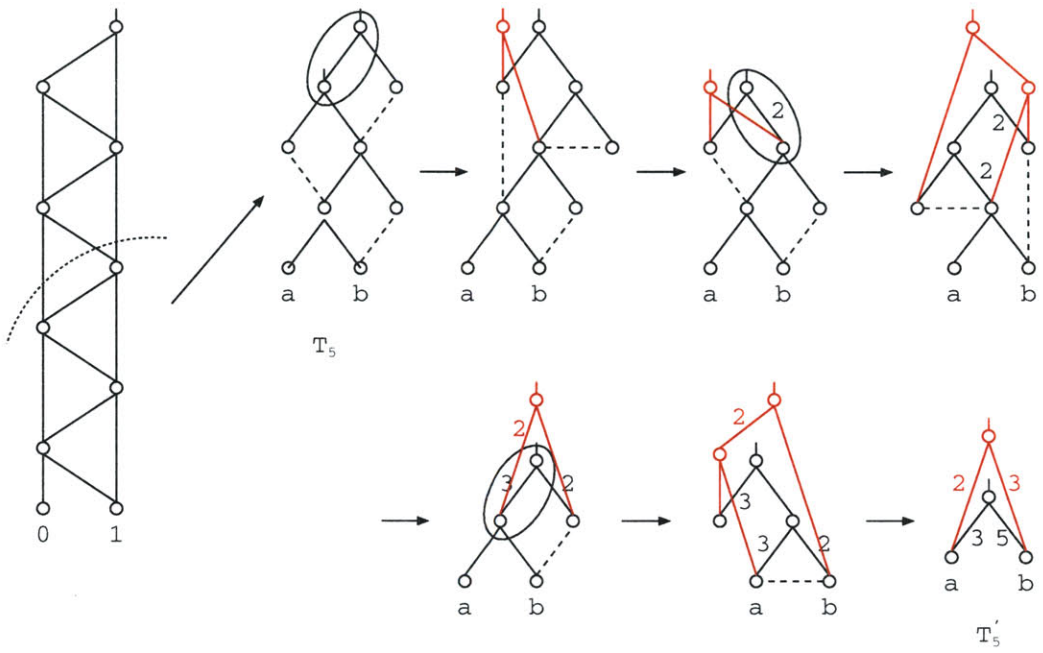


Figure 4-15: Computing the 9th Fibonacci number: cut and flatten.

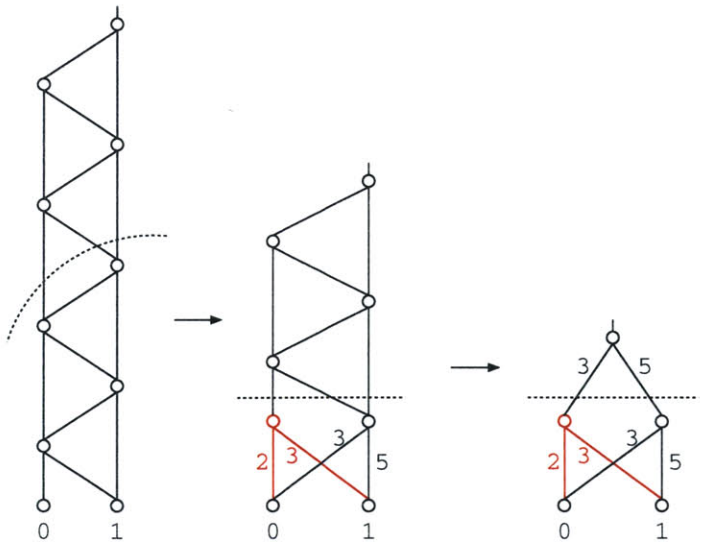


Figure 4-16: Computing the 9th Fibonacci number: applying T_5' twice.

Table 4.1: Computing Fibonacci numbers with the flattened tree.

Height	Output 1	Output 2
F_1	0	1
F_5	$2 \cdot 0 + 3 \cdot 1 = 3$	$3 \cdot 0 + 5 \cdot 1 = 5$
F_9	$2 \cdot 3 + 3 \cdot 5 = 21$	$3 \cdot 3 + 5 \cdot 5 = 34$
F_{13}	$2 \cdot 21 + 3 \cdot 34 = 144$	$3 \cdot 21 + 5 \cdot 34 = 233$

the form:

$$F_k = \sum_i c_i \cdot F_{k-b_i} + a_k$$

with the c_i, a_k being arbitrary constants, and b_i integers at least 1.

Example: Unlimited Item Knapsack Problem

Here, we briefly describe how to the same framework with a different operator can parallelize a dynamic programming solution to the knapsack problem. The problem states that there is a knapsack of capacity C, that can be filled with items of which there are n types. Each type of item has a value x_i and a size c_i. In this version, an unlimited supply of each item is available, and the objective is to fill the knapsack so that its value is maximized. The dynamic programming approach is to build a table where $t(i, j)$ is the maximum value knapsack attainable with items $1, \ldots, i$ and capacity j. This table is populated by setting

$$t(i, j) = \max_{r \in \{0,1\}} \max_{k \in \{1,\ldots,i\}} t(i - r, j - c_k) + x_k$$

as j ranges from 1 to C, for each $i = 1, \ldots, n$.[1]

This algorithm can be parallelized in its entirety as one giant computation with one output, or alternatively just its inner loop with multiple outputs. The main operation is the construction of the computation DAG; to this end, we use max as the binary operator in the tree and also allow edge constants as additive offsets. Figure 4-17 illustrates the rotation rule, where a missing edge constant is assumed to

[1]We ignore the detail of keeping the indices at least one.

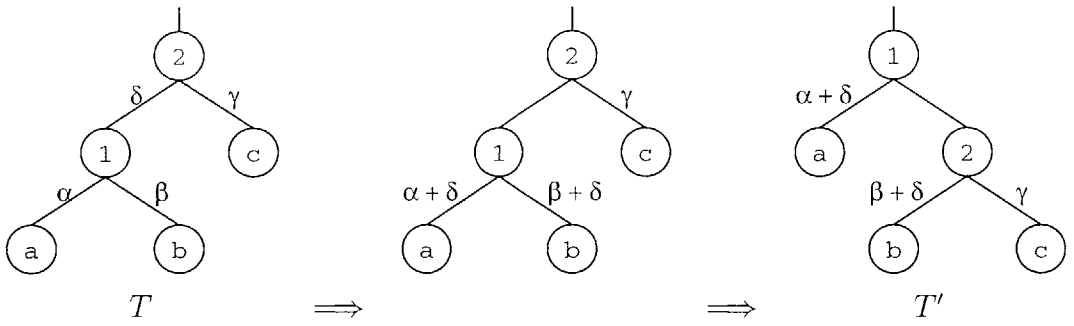


Figure 4-17: Rotating with the max operator and edge offsets.
$\max(\max(a+\alpha, b+\beta)+\delta, c+\gamma) \Rightarrow \max(a+\alpha+\delta, \max(b+\beta+\delta, c+\gamma))$

---

be zero. It can be shown that this operator is sufficient to construct a computation DAG, which can be flattened to logarithmic height.

4.4 Discussion

We have provided some promising results in using tree rotations as a means of parallelizing a computation. However, there remain one significant limitation of this method: constructing the computation tree is a non-trivial task. An algorithm is not implemented as a computation tree, nor does an implementation always lead directly to the construction of one. In some problems, such as prefix or Fibonacci, the tree can typically be inferred directly from the syntax of the implementation. However, a problem such as knapsack transfers itself more indirectly. Moreover, all the problems mentioned so far involve purely computation: there are no decision points. It remains unclear how branching can be incorporated into this model. One final concern is that the algorithms given run in polynomial time with relatively high degree; however, we do not believe the problems are inherently difficult, and that the high degrees may be made tighter in future studies.

Some prior works achieve results to problems similar to the ones explored here. In [3], Brent provides an algorithm for evaluating arithmetic expressions of n terms in $O(\lg n + \frac{n}{p})$ time. Later, Miller and Reif in [17] developed the idea of tree contractions: these dictate how a parallel computer can contract a computation tree to a single node,

performing the computation in the process, in $O(\lg n)$ time with $O(n/\lg n)$ processors (using randomization). Leiserson and Maggs further used this idea of contraction in [16] to provide an algorithm that computes the value at every node of the computation tree, in $O(\lg n)$ time using randomization.

The primary distinction between these prior results and the work given in this chapter is that they perform the computation directly. For these results to be useful, the computation tree needs to be available at runtime. If the entire tree is created or even loaded, then it is possible that already too much time has been spent to make its use worthwhile. The tree rotations state only how the problem is to be computed by a parallel computer, and depends only on the tree structure: the leaf values are open variables. In this way, we may memoize the parallelization and achieve results by solving just a subtree. Furthermore, the usefulness of tree rotations in both Fibonacci numbers and knapsack problems demonstrates that the domain of parallelizable problems is not limited to strict trees: there is hope that tree rotations may be used to solve a wide range of problems.

Chapter 5

Conclusions

The problems studied in this thesis range from an applied algorithm, to an engineered system, to a theoretical situation. This conclusion shall give a brief summary of the results from each chapter.

Regarding parallel sorting, to the best of our knowledge, we have presented a new deterministic algorithm. This algorithm makes a strong case for exact splitting, with empirical evidence indicating that the cost of finding the exact split points scales well. One primary concern is that the processors need to be homogenous: because many rounds of blocking communication are required, if any one processor is significantly slower (due to load or other factors), the efficiency of the algorithm rapidly deteriorates. From a theoretical standpoint, this algorithm exhibits the desirable traits of 1-optimality in both communication and computation in the comparison model. This sorting algorithm also compares favorably against various sample sort algorithms, which have been argued to be the best for large inputs. Finally, modulo some intricacies of determining the exact splitters, the algorithm is conceptually simple to understand, analyze, and implement.

With the Star-P project, we have provided two extensions that increase the project's ability to solve real problems. The distributed load and save maintain the design principle of keeping large data sets on the parallel computer, and empirical evidence indicates particularly good scalability of distributed load on an Altix machine. The shared memory export functionality was added with the motivation of

incorporating the operation of Star-P with other tools on a parallel computer. We explain the motivation behind a particular interface to the functionality, and give empirical results of different modes. Finally, we presented numerous examples that illustrate various aspects of both the Star-P system design, as well as the functionality it provides.

For the long-range goal of automatic parallelization, we have made some advances in a restricted domain. We define a computation tree as a general representation of an algorithm, and then relate the height of a computation tree to its parallel execution time. We then give a simple algorithm that uses tree rotations to make the height of an arbitrary binary tree logarithmic in the number of its leaves. Based on this tree flattening algorithm, we extend the idea of tree rotation to get further generalization power. It is not inconceivable for a sufficiently sophisticated system to use these methods to parallelize computations ranging from prefix sums, to Fibonacci numbers, to knapsack problems. The two main drawbacks are that the algorithms presented have weak bounds on their performance, and that there is much work still to be done to make this effort translate to practice.

Bibliography

[1] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *SPAA*, pages 3–16, 1991.

[2] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Linear time bounds for median computations. In *STOC*, pages 119–124, 1972.

[3] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[5] Gerth Stolting Brodal and Rolf Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 219–228. Springer-Verlag, 2002.

[6] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it right. *Proceedings of the IEEE*, 93(2), 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[7] Ron Choy, Alan Edelman, John Gilbert, Viral Shah, and David R. Cheng. Star-P: High productivity parallel computing. *HPEC*, 2004.

[8] Ron Long Yin Choy. MATLAB*P 2.0 : Interactive supercomputing made practical. Master's thesis, MIT, Cambridge, MA, 2002.

[9] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[10] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, Lecture Notes in Computer Science. BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[11] Gianni Franceschini and Viliam Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. In *FOCS*, page 242, 2003.

[12] Alexandros V. Gerbessiotis and Constantinos J. Siniolakis. Deterministic sorting and randomized median finding on the bsp model. In *SPAA*, pages 223–232, 1996.

[13] Michael T. Goodrich. Communication-efficient parallel sorting. In *STOC*, pages 247–256, 1996.

[14] David R. Helman, Joseph JáJá, and David A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *J. Exp. Algorithmics*, 3:4, 1998.

[15] Parry Husbands. *Interactive Supercomputing.* PhD thesis, MIT, Cambridge, MA, 1999.

[16] Charles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1986.

[17] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *FOCS*, pages 478–489, 1985.

[18] Cleve Moler. *Numerical Computing with MATLAB*, chapter 2, pages 22–30. SIAM, 2004.

[19] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95.* IEEE Press, 1995.

[20] Angelika Reiser. A linear selection algorithm for sets of elements with weights. *Information Processing Letters*, 7(3):159–162, 1978.

[21] E. L. G. Saukas and S. W. Song. A note on parallel selection on coarse grained multicomputers. *Algorithmica*, 24(3/4):371–380, 1999.

[22] Viral Shah and John R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *HiPC*, pages 144–155, 2004.

[23] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *J. Parallel Distrib. Comput.*, 14(4):361–372, 1992.

[24] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[25] `http://supertech.lcs.mit.edu/cilk/`. The Cilk Project.

[26] `http://top500.org/lists/current.php`. Top 500 Supercomputer Sites.

[27] `http://www.cs.washington.edu/research/zpl/home/`. ZPL.

[28] `http://www.eece.unm.edu/~dbader/code.html`, 1999.

[29] `http://www.engineeredintelligence.com/products/cxc_compilers.cfm`. CxC.

[30] `http://www.hpc-design.com/bert.html`. BERT 77: Automatic and Efficient Parallelizer for FORTRAN.

[31] `http://www.mathworks.com`. MATLAB.

[32] `http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matfile_format.pdf`. MATLAB MAT-file format.

[33] `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external`. MATLAB external interfaces.

[34] http://www.mpi-forum.org. The Message Passing Interface MPI Standard.

[35] http://www.semdesigns.com/Products/Parlanse/. PARLANSE: A PARallel LANguage for Symbolic Expression.

[36] http://www.sgi.com/products/servers/altix/. SGI altix.

[37] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.