# MINIMUM SPANNING TREE VERIFICATION, FAST PRIORITY QUEUES, AND MASSIVELY PARALLEL FACTORING

Brandon Dixon

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

October 1993

# Abstract

We present algorithms for problems from three different areas of research on parallel computation. The first problem that we consider is verifying minimum spanning trees. Given a weighted, undirected graph $G$ and a spanning tree $T$ of $G$, our algorithms determine whether $T$ is a minimum spanning tree. We present the first linear time algorithm for this problem, that is, if $G$ has $n$ vertices and $m$ edges, our algorithm runs in $O(m+n)$ time given a unit cost random access machine (RAM) with $O(\log n)$ word size. Along with the spanning tree verification problem, we consider the spanning tree sensitivity analysis problem. This problem is to determine the maximum possible change for each individual edge cost that does not destroy the minimality of the given spanning tree. We give two algorithms for this problem: the first runs in $O(m+n)$ expected time and the second runs in optimal deterministic time, but the exact time bound is not known.

We additionally present optimal parallel algorithms for the minimum spanning tree verification and sensitivity analysis problems. The parallel algorithms share top level ideas with the above sequential algorithms, but their implementations require different techniques. The parallel verification algorithm runs in $O(\log n)$ time using $\Theta((m+n)/\log n)$ processors in the concurrent read, exclusive write (CREW) PRAM model. The parallel sensitivity analysis algorithms are the parallel equivalents of the sequential sensitivity analysis algorithms; we give an algorithm that runs in $O(\log n)$

expected time using $\Theta((m+n)/\log n)$ CREW processors and a deterministic algorithm that run is optimal but unknown time using $\Theta((m+n)/\log n)$ processors as well.

The second problem that we consider is that of allowing concurrent operations on a priority queue over some bounded universe. Given a bounded universe, say $[0 \ldots N]$, we give algorithms for performing insertions, deletions and find minimum operations on a set of elements from the universe. Each operation completes in $O(\log \log N)$ time. Our algorithms are unique because the operations can be pipelined in such a way that a new operation can always begin on the set in a constant amount of time. This allows concurrent operations on the set and gives a way to increase the throughput of the data structure without increasing the latency of any single operation. We give an application of this data structure to the longest increasing subsequence problem, giving an optimal linear time algorithm using $\Theta(\log \log N)$ processors.

The last problem we consider is that of factoring integers. Our contribution to this widely studied problem is to give efficient factoring algorithms for massively parallel single instruction, multiple data (SIMD) machines. The algorithms presented here have been implemented on a 16K processor MasPar SIMD machine. We present both special purpose factoring algorithms (where the running time depends strongly on the size of the smallest factor) and general purpose methods (where the running time depends solely on the size of the integer to be factored). Additionally we present fast algorithms for the multiplication of large integers modulo a fixed integer, a problem useful for many applications in the area of computational number theory and cryptography.

# Acknowledegments

Many people have helped to make writing this thesis possible. Firstly I would like to thank my advisor Bob Tarjan for his support and guidance over the last years. His knowledge and intuition have been invaluable, I am grateful to have been his student. I want to thank Arjen Lenstra for sharing both his work on challenging problems and his sense of humor with me. I have enjoyed our time working together greatly. I also thank both Bob, Arjen, and Joel Friedman for their time reading my thesis.

I have enjoyed my time at Princeton due to the pleasant working environment at the department and my friendly officemates: Heather Booth, Leonidas Palios, Monika Rauch, Andrew Tolmach, and Arthur Watson. I thank Rebeca Davies for her help me with all of my TeX questions over the years.

Financial support for my work at Princeton has come from a National Science Foundation Graduate Fellowship and from DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center, Grant No. NSF-STC88-09648.

I especially want to thank my parents for instilling in me the values which I hold so dear and for giving me the confidence and the drive to achieve my goals.

Last, but certainly not least, I thank Monika Rauch for her professional and personal companionship, which has made the last four years so enjoyable.

# Contents

# List of Figures

# Chapter 1

# Introduction

This thesis contains results from two distinct areas of research in algorithms. Chapters 2 and 3 present sequential and parallel random access machine (RAM) algorithms. We analyze the running times of these algorithms, and for each of the problems considered, we give the first algorithms whose running time is provably optimal. Chapter 4 presents applied work in the area of parallel computing. We describe algorithms developed for factoring integers using a real world single instruction multiple data (SIMD) parallel computer. These algorithms and their implementations are designed to be as fast as possible in practice. Because of the difference in the design goals of the algorithms given in the thesis, the presentation varies somewhat between chapters, and each chapter is self contained.

In Chapter 2 we consider the minimum spanning tree verification and sensitivity analysis problems. To verify a minimum spanning tree we must determine whether a given spanning tree of a weighted graph has minimum cost. The sensitivity analysis problem is to determine for every individual edge of the graph, the maximum change in cost for that edge that preserves the minimality of a given spanning tree. In this chapter, we present both sequential and parallel algorithms for these problems. The verification result is intriguing due to the fact that minimum spanning trees can currently be verified faster than they can be found. This lends some hope to the idea that spanning trees can be found faster.

The sequential verification algorithm runs in linear time, and the sequential sensitivity analysis algorithm runs in linear expected time and optimal deterministic time. These results are joint work with Monika Rauch and Bob Tarjan and appear in [18]. The parallel verification algorithm runs in $O(\log n)$ time using $\Theta((n+m)/\log n)$ processors, where $n$ is the number of vertices and $m$ is the number of edges in the given graph. The parallel sensitivity analysis algorithm has time bounds similar to those of the sequential version, $O(\log n)$ expected time and optimal deterministic time, using $\Theta((n+m)/\log n)$ processors. The parallel results share top–level algorithmic ideas with the sequential algorithms, but the actual data structures and implementations are quite different. The parallel algorithms are joint work with Bob Tarjan.

In Chapter 3 we present a pipelined data structure for maintaining a priority queue. A priority queue is a data structure that supports insertion, deletion, and find minimum operations on elements from some ordered universe. If the range of the elements can be bounded by some range, say $[0 \ldots N]$, then the running time of the priority queue operations is $O(\log \log N)$ using the data structure due to van Emde Boas [20]. The algorithms in this chapter support priority queue operations on elements from a bounded universe in $O(\log \log N)$ time as well, but they additionally allow the priority queue operations to be pipelined in such a way that a new operation can begin every $O(1)$ time steps. In this way multiple processors can operate concurrently on the queue. A version using simple, easy to implement data structures gives an amortized running time of $O(\log \log N)$ per operation, and a version with worst case $O(\log \log N)$ time per operation is presented as well.

The focus shifts in Chapter 4 to the efficient implementation of massively parallel algorithms for integer factoring. The algorithms presented are for single instruction multiple data (SIMD) parallel computers, and all of the algorithms have been implemented on the MasPar SIMD machine. We present both special purpose factoring algorithms (where the running time depends strongly on the size of the smallest factor) and general purpose methods (where the running time depends solely on the size of the integer to be factored). Additionally we present fast algorithms for the multiplication of large integers modulo a fixed integer. This multiplication algorithm is useful for many applications in the area of computational number theory and cryptography. The results of this chapter are joint work with Arjen Lenstra and appear in [15, 16, 17].

# Chapter 2

# Verification of minimum spanning trees

## 2.1 Introduction

In this chapter, we present algorithms for solving the minimum spanning tree verification and sensitivity analysis problems. In Section 2.2 we present a sequential algorithm that tests the minimality of a given spanning tree of an $n$-vertex, $m$-edge connected graph in $O(m)$ time, and in Section 2.3 we give sequential algorithms performing sensitivity analysis of minimum spanning trees in optimal worst-case time and in linear expected time. In Sections 2.4 and 2.5, we present optimal parallel algorithms for the verification and sensitivity analysis problems. The parallel verification algorithm runs in $O(\log n)$ time using $\Theta((n+m)/\log n)$ processors in the Concurrent Read Exclusive Write (CREW) PRAM model.

The algorithmic technique used in the algorithms in this chapter is to apply divide–and–conquer strategies followed by table lookup to solve the remaining small problems. In a general setting, the idea is to split the problem to be solved into subproblems, which are categorized into classes. If the subproblems are small enough, they can be solved efficiently as follows: An optimal algorithm for each class is precomputed and stored in a look-up table, and each instance of a subproblem is solved by looking up and running the algorithm for its class. For this technique to pay off, solving all the subproblems must reduce the original problem sufficiently so that it can be solved quickly with respect to the size of the original problem by using a non-optimal algorithm.

We use this general technique to efficiently solve the minimum spanning tree verification and sensitivity analysis problems. This approach was first proposed explicitly by Larmore [36], who used it to solve a convex matrix searching problem. Related techniques were used by Gabow and Tarjan [25] to solve a disjoint set union problem, by Harel and Tarjan [31] to find nearest common ancestors in a tree, and by Fredman [22] to solve the all pairs shortest path problem.

The problem of verifying a minimum spanning tree is intriguing because it is a problem for which faster algorithms exist to verify a solution than exist to find a solution. In particular, the linear time algorithm presented here is in contrast to the fastest known algorithm to find minimum spanning trees [24], which runs in $O(m \log \beta(m,n))$ time, where $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq m/n\}$, and $\log^{(i)} n$ is defined recursively by $\log^{(0)} n = n$ , $\log^{(i+1)} n = \log \log^{(i)} n$. It is interesting to note that the gap in efficiency between verifying and finding minimum spanning trees is greater in the parallel models of computation than in the sequential case. The $O(\log n)$ time and linear work algorithm of Section 2.4 is in contrast to the fastest known parallel algorithm for finding minimum spanning trees without using concurrent writes, which requires $O((\log n)^{3/2})$ time using $\Theta(m+n)$ processors [32]. Thus verification takes a factor of $\Theta((\log n)^{3/2})$ less work. If we allow concurrent writes, the fastest known algorithm requires $O(\log n)$ time and $\Theta(n+m)$ processors [5], but this is still a factor of $O(\log n)$ more work than verification.

Previously, the verification problem was considered by Tarjan [60] and subsequently by Komlós [35]. Tarjan proposed an algorithm that runs in $O(m\alpha(m,n))$ time, where $\alpha$ is a functional inverse of Ackermann's function. Komlós showed that a minimum spanning tree can be verified in $O(m)$ binary comparisons between edge costs. Unfortunately, his method requires nonlinear time to determine which comparisons to make. An unpublished paper of Alon and Schieber [4] gives an algorithm for verifying minimum spanning trees in $O(\log n)$ time using $\Theta((n+m)\alpha(m,n)/\log n)$ processors.

Our model of computation allows edge costs to be compared, added, or subtracted in unit time, and side computations to be performed on a uniform-cost random-access machine (RAM) with word size $\Omega(\log n)$ bits. The sequential algorithm uses the comparison bound of Komlós [35] for the subproblems and Tarjan's $O(m\alpha(m,n))$ algorithm [60] for the reduced problem. For the parallel algorithms, our model is a concurrent read, exclusive write (CREW) parallel random-access machine (PRAM).

It is important to note that our computational model allows only very simple operations on the edge costs, namely binary comparison, addition, and subtraction. In a more powerful model in which bit manipulation of the edge costs is possible, better results can be obtained using table lookup and other ideas. In particular, Harel [30] showed that if the edge costs are integers polynomially bounded in $n$, then on a unit-cost random-access machine the minimum spanning tree verification and sensitivity analysis problems can be solved deterministically in linear time. More recently Fredman and Willard [23] showed that if the edge costs are single-precision integers, then a minimum spanning tree can be computed deterministically in linear time on a unit-cost random access machine. These results are incomparable with ours because of the stronger computation model used.

## 2.2 Sequential verification

Let $G = (V, E)$ be a connected, undirected graph with vertex set $V$ of size $n$ and edge set $E$ of size $m$. Suppose every edge $\{v, w\} \in E$ has a real-valued *cost* $c(v, w)$. A *minimum spanning tree* of $G$ is a spanning tree whose total edge cost is minimum. The *minimum spanning tree verification problem* is that of determining whether a specific spanning tree $T$ is a minimum spanning tree. Since $G$ is connected, $m \geq n - 1$. To simplify time bounds, we assume that $m \geq n$; otherwise, $G$ itself is a tree.

Let $T$ be a spanning tree whose minimality we wish to test. For any pair of vertices $v, w$, we denote by $T(v, w)$ the path in $T$ from $v$ to $w$. The spanning tree $T$ is minimum if and only if, for every nontree edge $\{v, w\}$, $c(v, w) \geq \max\{c(x, y) \mid \{x, y\} \in T(v, w)\}$. In order to efficiently verify this condition, we replace each nontree edge $\{v, w\}$ by a set of up to six replacement edges, each of cost $c(v, w)$. This replacement leaves invariant the minimality of $T$. Edge replacement is a two-stage process. To begin the first stage, we choose an arbitrary vertex $r$ and root $T$ at $r$. We denote by $p(v)$ the parent of vertex $v$ in the rooted version of $T$. For each nontree edge $\{v, w\}$, we compute the nearest common ancestor of $v$ and $w$ in $T$, say $u$. If $v$ and $w$ are unrelated in $T$ (i.e., $u$ is not either $v$ or $w$), we replace $\{v, w\}$ by the pair of edges $\{u, v\}$, $\{u, w\}$, each with cost $c(v, w)$. Such replacement leaves invariant the minimality of $T$, at most doubles the number of nontree edges, and results in a graph such that every nontree edge joins two related vertices in $T$. The time to perform this replacement is $O(m)$ using either of the known linear-time algorithms for computing nearest common ancestors [31],[56].

We can now assume that each nontree edge $\{u, v\}$ is such that vertex $u$ is an ancestor of vertex $v$. In the second stage, we replace each such edge by a set of up to three edges. In order to determine the edge replacements, we partition $T$ into a collection of edge-disjoint subtrees. Let $g \geq 1$ be an integer parameter, whose value we shall specify later. The subtrees have two properties:

(i) there are at most $(n-1)/g+1$ subtrees; and

(ii) deletion from any subtree of its root and all edges incident to the root leaves a collection of smaller subtrees, called *microtrees*, each containing at most $g$ vertices.

We compute the collection of subtrees in $O(n)$ time, as follows. We process all the vertices except $r$ in postorder [62]. (This order guarantees that a parent is processed after all of its children.) When processing a vertex $v$, we compute an integer value $s(v)$ for it; and, in addition, we may mark it as a subtree root. The computed value of $s(v)$ is the number of descendants of $v$ in $T$ (including $v$ itself) that are in the same microtree as $v$. Initially all vertices are unmarked. The vertex processing step is as follows:

*process*$(v)$: Compute $h = 1 + \sum\{s(w) \mid w \text{ is a child of } v\}$.
If $h \leq g$ then let $s(v) = h$;
otherwise, mark $v$ as a subtree root and let $s(v) = 1$.

Once the vertex processing is completed we mark $r$, the root of $T$, as a subtree root. See Figure 1 for an example of the microtree structure. Condition (ii) is immediate from the definition of the vertex processing. Condition (i) is also immediate: each subtree, except possibly the one rooted at $r$, contains more than $g$ vertices and hence contains at least $g$ edges, which means that there are at most $(n-1)/g+1$ subtrees.

Let $T'$ be the tree whose vertices are the marked vertices of $T$, with $v$ the parent of $w$ in $T'$ if $v$ is the deepest marked proper ancestor of $w$ in $T$ (i.e., the first marked vertex encountered on the path from $w$ to $r$ in $T$). We call $T'$ the *macrotree*. By (i), $T'$ has $O(n/g)$ vertices. Tree $T'$ can be computed in $O(n)$ time by doing a depth-first

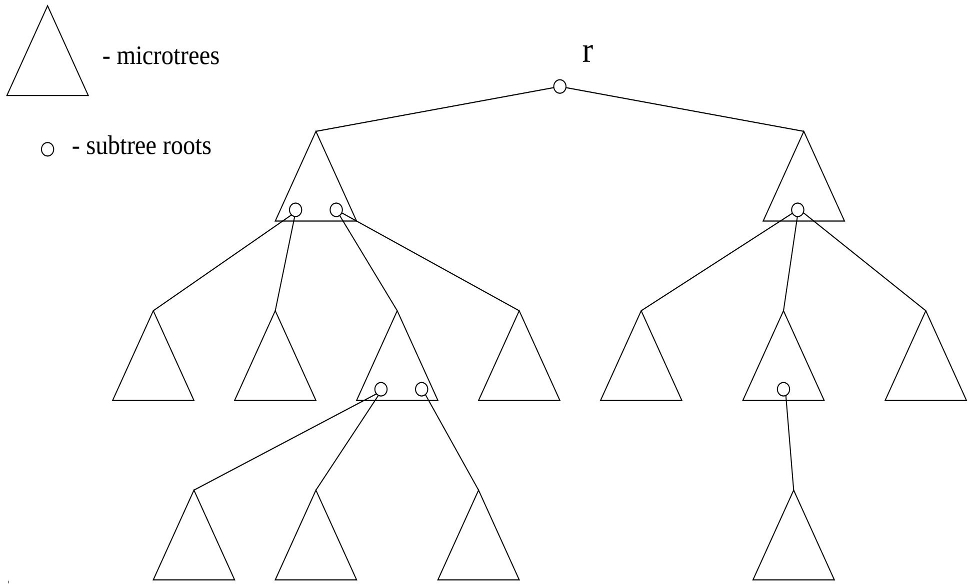


Figure 1: The microtree structure.

traversal of $T$ and maintaining the set of marked proper ancestors of the currently visited vertex on a stack; when the search visits a vertex $v$, the deepest marked proper ancestor of $v$, which we denote by $p'(v)$, is on top of the stack. (We adopt the convention that $p'(r)$ is undefined.)

We use the macrotree to define the replacement edges for each nontree edge. Let $\{u, v\}$ be such a nontree edge, with $u$ an ancestor of $v$. Let $r_1 = p'(u)$ if $u$ is unmarked or $u$ if $u$ is marked. Similarly, let $r_3 = p'(v)$ if $v$ is unmarked or $v$ if $v$ is marked. If $r_1 = r_3$, we do not replace $\{u, v\}$. If $r_1 \neq r_3$, let $r_2$ be the child of $r_1$ in $T'$ that is an ancestor of $r_3$; replace $\{u, v\}$ by $\{u, r_2\}$, $\{r_2, r_3\}$, $\{r_3, v\}$, deleting any of these edges that is a loop (an edge of the form $\{x, x\}$ for some $x$). See Figure 2. Each new edge has a cost of $c(u, v)$. This replacement leaves invariant the minimality of $T$ and at most triples the number of nontree edges.

We can compute the replacement edges for every nontree edge in a total of $O(m)$ time, as follows. A depth-first traversal of $T$ as described above allows us to compute $p'(v)$ for each vertex $v \neq r$. This gives $r_1$ and $r_3$ in the construction. It remains to

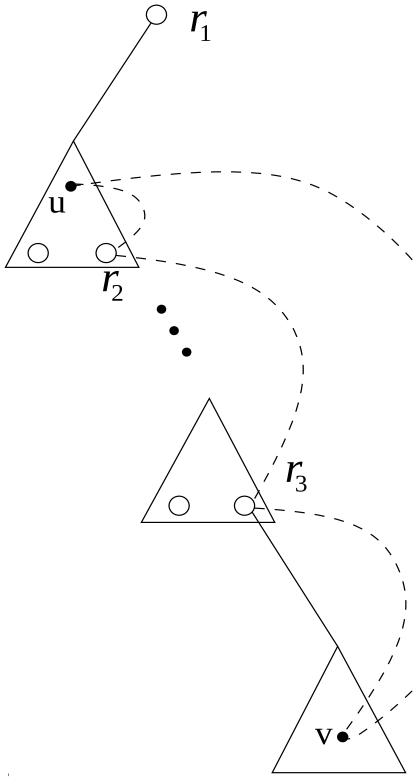


Figure 2: Edge $(u, v)$ is replaced by $(u, r_2)$, $(r_2, r_3)$, and $(r_3, v)$.

compute the $r_2$-vertices in the edge replacement construction. The computation of these vertices requires answering $O(m)$ queries of the following form on $T'$: given a vertex $z$ and another vertex $y$ that is a proper ancestor of $z$, determine the child of $y$ that is an ancestor of $z$. These queries can be answered in $O(m)$ time by performing a depth-first traversal of $T'$, maintaining a stack of the ancestors of the currently visited vertex, and answering the query for a pair $y, z$ when visiting $z$ during the search, by reporting as the answer to the query the vertex just after $y$ on the stack.

Having computed all the replacement edges, we must test, for each replacement edge $\{w, x\}$, whether $c(w, x) \geq \max\{c(y, z) \mid \{y, z\} \in T(w, x)\}$. In the rest of this section we describe how to perform this test for all replacement edges.

For each vertex $v \neq r$, we compute a value $high(v)$ equal to the maximum cost of an edge on the path $T(p'(v), v)$. These values can be computed for all vertices by doing a separate depth-first traversal of each of the subtrees of $T$ that were determined by the partitioning process described previously. During the traversal of the subtree rooted at a vertex $u$, we maintain the path of edges from $u$ to the currently visited vertex as a stack with heap order [26]; the values that are heap-ordered are the edge costs, and the *high*-values are computed using *find-max* operations. This data structure requires $O(1)$ amortized time per *push*, *pop*, or *find-max* operation [26]. Hence the total time to compute all *high*-values is $O(n)$. The *high*-values suffice to perform the required test for each of the $\{r_3, v\}$-replacement edges, in $O(1)$ time per edge: for such an edge, $high(v) = \max\{c(y,z) \mid \{y,z\} \in T(r_3, v)\}$.

We deal with the $\{r_2, r_3\}$-replacement edges by adding all of these edges to $T'$ to form a graph $G'$, giving each edge $\{p'(v), v\}$ in $T'$ a cost $c(p'(v), v) = high(v)$, and verifying that $T'$ is a minimum spanning tree in $G'$. To verify the minimality of $T'$ we use the algorithm of Tarjan [60], which runs in $O(m\alpha(m, n'))$ time, where $n'$ is the number of vertices in $T'$. If $g = \Omega(\log^{(i)} n)$ for any fixed positive integer $i$, then $n' = O(n/\log^{(i)} n)$, and $\alpha(m, n') = O(1)$ [62]. Thus verifying the minimality of $T'$ takes $O(m)$ time.

The remaining edges that must be tested are the $\{u, r_2\}$-replacement edges. Each such edge has $u$ and $r_2$ in the same microtree. Let $T_1, T_2, \ldots, T_k$ be the microtrees. For $1 \le i \le k$, we form a graph $G_i$ by adding each $\{u, r_2\}$-replacement edge to the tree $T_i$ such that $u$ and $r_2$ are in $T_i$. Together the graphs $G_1, G_2, \ldots, G_k$ contain $n$ vertices and $O(m)$ edges. By (ii), each $G_i$ contains at most $g$ vertices. We complete the task of verifying the minimality of $T$ by verifying that $T_i$ is a minimum spanning tree of $G_i$, for each $i$ in the range $1 \le i \le k$.

To verify the minimality of the microtrees, we use a preprocessing and table look-up technique. For each possible connected graph with no more than $g$ vertices and specified spanning tree, we construct a short integer encoding by numbering the vertices consecutively from 1, encoding each edge by the pair of numbers of its end vertices, and concatenating the encodings of the edges, listing the spanning tree edges first. (It does not matter that this encoding is not unique.) The encoding for a graph-tree pair contains at most $\lceil \log g \rceil g^2/2$ bits, since there are fewer than $g^2/2$ edges. The total number of possible code strings (not all of which are legal encodings of graphs) is not more than $2^{\lceil \log g \rceil g^2/2}$. We will choose $g$ such that each graph encoding fits into one computer word and such that there are at most $\sqrt{n}$ possible code strings. Choosing $g \leq c_2(\log n)^{1/3}$ for a suitably small value of $c_2$ more than suffices for this purpose.

Consider a connected graph with at most $g$ vertices and $e < g^2/2$ edges and having a specified spanning tree $T^*$. The result of Komlós [35] implies that there is a decision tree $D$ whose nodes represent binary comparisons of edge costs that will verify the minimality of $T^*$ and has a depth of at most $c_1 e$, for some sufficiently large $c_1$. The number of nodes in $D$ is at most $2^{c_1 e+1}$. Furthermore, an inspection of the construction of Komlós shows that $D$ can easily be constructed in $O(g^2)$ time per node, for a total of $O(g^2 2^{c_1 g^2/2})$ time.

Choosing $g \leq c_2(\log n)^{1/3}$ for a suitably small value of $c_2$ guarantees that the construction time for one decision tree is $O(\sqrt{n})$, and the total time required to construct decision trees for all possible graphs with at most $g$ vertices is $O(n)$. Furthermore, the space needed to store all the decision trees is $O(n)$.

We construct one decision tree for each possible graph with at most $g$ vertices and then build a table that maps code strings for graphs to the corresponding decision trees. Then we use the table to verify the minimality of the microtrees $T_i$ in the respective graphs $G_i$, by computing a code string for each $G_i, T_i$ pair, accessing the decision tree corresponding to the code string, and following the path through the decision tree determined by the edge costs of $G_i$. The total time to perform all the verifications is $O(m)$. This completes the verification of $T$.

The only constraints imposed on the choice of $g$ in this construction are $g = \Omega(\log^{(i)} n)$ for some fixed positive integer $i$ and $g \leq c_3(\log n)^{1/3}$ for $c_3 = \min\{c_0, c_2\}$. Thus it suffices to choose $g = c_3(\log n)^{1/3}$.

## 2.3 Sequential sensitivity analysis

An extension of the minimum spanning tree verification problem is the sensitivity analysis problem. Let $G$ be an undirected graph with edge costs and let $T$ be a minimum spanning tree of $G$. The *sensitivity analysis problem* is to compute, for each edge $\{v, w\}$ of $G$, by how much $c(v, w)$ can change without affecting the minimality of $G$. Tarjan [61] has extended his verification algorithm to an algorithm that solves the sensitivity analysis problem in $O(m\alpha(m, n))$ time. For the special case of planar graphs, Booth and Westbrook [8] have given an algorithm running in $O(m)$ time. We shall describe a randomized $O(m)$-time algorithm and a deterministic algorithm that runs in time minimum to within a constant factor, although all that we can say for sure about the running time of the latter algorithm is that it is $O(m\alpha(m, n))$ and $\Omega(m)$. Our technique is the same as that of Section 2.2; namely, we reduce the original problem in $O(m)$ time to a collection of subproblems, each of which is small enough to solve by using a decision tree selected from a precomputed set of such trees. For the randomized algorithm, we solve the subproblems using a result of Goddard, King, and Schulman [28], and we enumerate of all possible algorithms in the deterministic case. In both cases Tarjan's $O(m\alpha(m, n))$-time sensitivity analysis algorithm [61] processes the reduced problem.

Let $\{v, w\}$ be a nontree edge. Let $a(v, w) = \max\{c(x, y) \mid \{x, y\} \in T(v, w)\}$. Then $T$ remains minimum until the edge cost of $\{v, w\}$ decreases by more than $c(v, w) - a(v, w)$ (See Figure 3). Similarly, let $\{v, w\}$ be a tree edge. Let $b(v, w) = \min\left\{c(x, y) \mid \{x, y\} \text{ is a nontree edge such that } \{v, w\} \in T(x, y)\right\}$. Then $T$ remains minimum until the edge cost of $\{v, w\}$ increases by more than $b(v, w) - c(v, w)$. (See Figure 4).

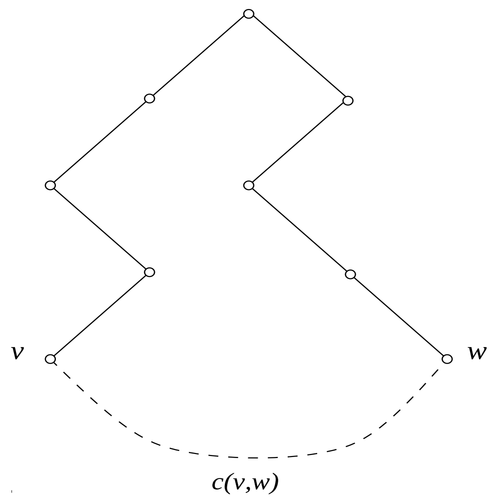


Figure 3: Sensitivity analysis of the nontree edges: $c(v, w)$ can decrease to the minimum cost on the tree path from $v$ to $w$.

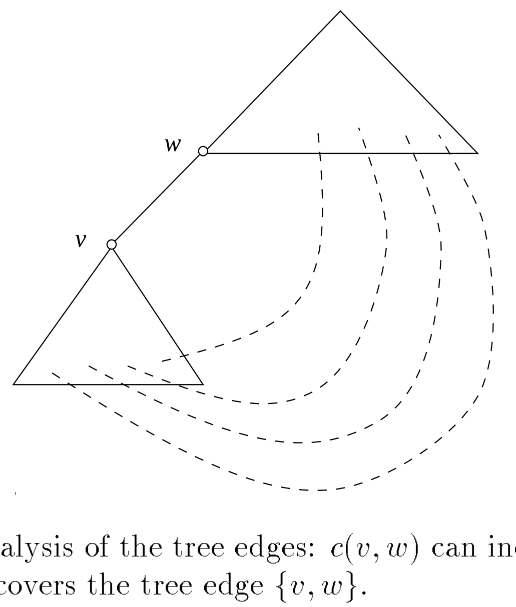


Figure 4: Sensitivity analysis of the tree edges: $c(v, w)$ can increase to the maximum cost nontree edge that covers the tree edge $\{v, w\}$.

The value of $a(v, w)$ for every nontree edge $\{v, w\}$ can be computed in $O(m)$ time by a simple extension of the verification algorithm in Section 2.2: instead of verifying that $c(v,w) \geq a(v,w)$, we compute $a(v,w)$ explicitly. Explicit computation of the $a(v,w)$ values can be done by computing, for each of the six replacement edges for an original edge $\{v,w\}$, the maximum-cost tree edge on the path in $T$ joining the endpoints of the edge. These maxima can be computed by slightly modifying the algorithm of Section 2.2; the algorithm of Komlós actually computes the maxima along tree paths needed to handle the replacement edges within the microtrees.

Computing $b(v,w)$ for every tree edge $\{v,w\}$ is harder. We first replace the nontree edges exactly as in Section 2.2: each nontree edge $\{x,y\}$ is replaced by a set of up to six nontree edges, each of cost equal to that of $\{x,y\}$, in a way that preserves $b(v,w)$ for every tree edge $\{v,w\}$. In the process of performing this replacement, we choose a root $r$ of $T$ and compute subtree roots, subtrees, and microtrees exactly as in Section 2.2. After the replacement, each nontree edge $\{x,y\}$ is such that $x$ and $y$ are related in $T$, say $x$ is an ancestor of $y$. In addition, such an edge is of exactly one of three types:

Type 1: $x$ is a subtree root, $y$ is not a subtree root, and $x = p'(y)$, where $p'$ is defined as in Section 2.2: $p'(y)$ is the deepest ancestor of $y$ that is a subtree root.

Type 2: $x$ and $y$ are subtree roots.

Type 3: $x$ and $y$ are in the same microtree.

We compute each value $b(v,w)$ using the equation

$$b(v,w) = \min\{b_1(v,w),\ b_2(v,w),\ b_3(v,w)\}\,,$$

where $b_i$ for $i = 1,2,3$ is defined exactly like $b(v,w)$ but with the minimum taken only over nontree edges of type $i$.

To compute the $b_1$-values, we begin by computing, for each vertex $v \neq r$, the value $min_1(v) = \min\big\{c(x,y) \,|\, \{x,y\}$ is a type-1 edge such that $x = p'(y)$ is a proper ancestor of $v$ and $v$ is an ancestor of $y\big\}$. The $min_1$-values can be computed in $O(m)$ time by visiting the vertices of $T$ in postorder and applying the recurrence

$$min_1(v) = \begin{cases} \infty & \text{if } v \text{ is a subtree root;} \\ \min\big(\{c(x,v)\,|\,\{x,v\} \text{ is a type-1 edge}\} & \text{if } v \text{ is not a subtree root.} \\ \qquad \cup\ \{min_1(w)\,|\,p(w)=v\}\big) & \end{cases}$$

Then, for every vertex $v \neq r$ , $b_1(p(v),v) = min_1(v)$.

We compute the $b_2$-values in three steps. First, we form the graph $G'$ as in Section 2.2 by adding to the macrotree $T'$ each type-2 edge. Second, we compute, for each tree edge $\{v,w\}$ of the macrotree $T'$, the value $b'(v,w) = \min\big\{c(x,y)\,|\,\{x,y\}$ is a type-2 edge such that $(v,w) \in T'(x,y)\big\}$. All the $b'$-values can be computed in $O(m\alpha(m,n'))$ time by applying the sensitivity analysis algorithm of Tarjan [61] to the graph $G'$ and the tree $T'$. Choosing $g$ (the size parameter for the macrotrees) to be $\Omega(\log^{(i)} n)$ for any fixed positive integer $i$ results in an $O(m)$ time bound for this computation. Third, we compute, for each vertex $v \neq r$ in $T$, the value $min_2(v) = \min\big\{b'(p'(y),y)\,|\,\{p'(y),y\}$ is an edge of $T'$ such that $y$ is a descendant of $v$ and $p'(y)$ is a proper ancestor of $v\big\}$. The $min_2$-values can be computed in $O(n)$ time by visiting the vertices of $T$ in postorder and applying the recurrence

$$min_2(v) = \begin{cases} b'(p'(v),v) & \text{if } v \text{ is a subtree root and} \\ \min\{min_2(w)\,|\,p(w)=v\} & \text{if } v \text{ is not a subtree root.} \end{cases}$$

Then, for every vertex $v \neq r$ , $b_2(p(v),v) = min_2(v)$.

All that remains is to compute the $b_3$-values. Since the definition of a type-3 edge $\{x,y\}$ implies that $x$ and $y$ are in the same microtree, we can compute the $b_3$-values by adding the type-3 edges to the appropriate microtrees to form graphs $G_1, G_2, \ldots, G_k$ as in Section 2.2 and then process each $G_i, T_i$ pair separately. We again use preprocessing to construct a fast decision-tree algorithm for each possible graph-tree pair and then use table lookup to select the correct algorithm for each actual pair $G_i, T_i$.

It is only in the construction of the decision trees that the randomized and deterministic algorithms differ. We first consider the deterministic case. A decision tree for the sensitivity analysis problem consists of a binary tree, each internal node of which specifies a comparison between the costs of two edges, and each leaf $x$ of which provides a mapping $f_x$ from the tree edges of the problem graph to the nontree edges, such that the $b_3$-value of any edge $e$ is $c(f_x(e))$, assuming that the edge costs are consistent with the outcome of the comparisons leading to leaf $x$. The algorithm of Tarjan [61] implies the existence of an $O(m\alpha(m,n))$-depth decision tree for the sensitivity analysis of an $n$-vertex, $m$-edge graph and given spanning tree. Since $\alpha(g^2, g) = O(1)$ [62], these decision trees have depth $O(g^2)$.

For each possible connected graph with no more than $g$ vertices and specified spanning tree, we construct a minimum-depth decision tree for sensitivity analysis by brute-force enumeration. We restrict our attention to complete binary trees, enumerating all possible decision trees of each possible depth in increasing order by depth until finding a correct one. A complete binary decision tree of depth $d$ has $2^d - 1$ internal nodes and $2^d$ leaves. Each internal node corresponds to one of the less than $g^4$ possible binary comparisons of edge costs; each leaf can correspond to one of the less than $g^{2g}$ possible mappings of the tree edges to the nontree edges. Thus there are less than $(g^4)^{2^d-1}(g^{2g})^{2^d} < g^{g2^{d+2}}$ possible decision trees of depth $d$, assuming $g \geq 2$. The total number of trees that must be considered before encountering a correct one is $O(g^{g2^{c_3 g^2}})$ for some suitably large constant $c_3$. This is $O(2^{2^{g^3}})$. The space needed to store a decision tree of depth $d$ is $O(2^d g \log g) = O(2^{g^3})$, if $d = O(g^2)$. To determine whether a particular decision tree correctly solves the sensitivity analysis problem, it suffices to test that the correct answer is obtained for each of the at most $(g^2)! = O(2^{g^3})$ possible permutations of edge costs. Testing one permutation requires $O(g^2)$ time. The time to test a particular decision tree is thus $O(g^2 2^{g^3}) = O(2^{g^4})$, and the time to find a minimum-depth decision tree is $O(2^{g^4} 2^{2^{g^3}}) = O(2^{2^{g^4}})$. If we choose $g = c_4(\log\log n)^{1/4}$ for some sufficiently small constant $c_4$, then the time to find minimum-depth decision trees for all possible graph-tree pairs is $O(n)$, as is the space needed to store them in a table.

We compute the $b_3$-values for all tree edges in a microtree $T_i$ by indexing the lookup table with the code string of the pair $G_i, T_i$ to get a decision tree and evaluating the decision tree with the given edge costs.

The total time needed for sensitivity analysis is $O(m)$ plus time proportional to the sum of the minimum numbers of comparisons needed to perform sensitivity analysis for all of the $G_i$, $T_i$ pairs. Performing sensitivity analysis for all of the $G_i$, $T_i$ pairs is at most a constant factor more time-consuming than performing sensitivity analysis for a worst-case $n$-vertex, $m$-edge graph. Thus the sensitivity analysis algorithm runs in time minimum to within a constant factor, assuming that only binary comparisons between edge costs are used as tests.

In the randomized case, we replace the deterministic decision trees used for sensitivity analysis of the microtrees by randomized decision trees. In a randomized decision tree, each internal node corresponds either to a comparison of two edge costs or to a test of a distinct random bit. As in the deterministic case, we require every path of the decision tree to give the correct answer, but as a measure of the complexity of the tree we use the weighted average depth of a leaf, rather than the worst-case depth, taking the weight of a leaf to be $1/2^i$, where $i$ is the number of tests of random bits along the path from the root to the leaf.

Goddard, King and Shulman [28] have found a randomized algorithm to compute the maxima of $n$ subsets of an ordered universe of size $n$ in $O(n)$ comparisons on the average. Their result, together with the observation of King [34] that their algorithm needs only $O(n)$ random bits on the average, implies the existence of a randomized decision tree of average depth $O(m)$ for the sensitivity analysis problem. Such a decision tree can be converted into a decision tree of $O(m)$ average depth and $O(m \log m)$

worst-case depth by trimming the decision tree at depth $m \log m$ and replacing each subtree that was cut out by a decision tree that merely sorts by cost the edges of the problem graph. A brute-force enumeration can be used to find minimum-average-depth randomized decision trees for all possible microtree problems. The details mimic the deterministic case, so we omit them. The Goddard-King-Shulman result then implies that the resulting randomized sensitivity analysis algorithm runs in $O(m)$ expected time, for a suitable choice of the microtree size bound $g$.

## 2.4 Parallel verification

In this section we present a parallel algorithm for the minimum spanning tree verification problem. The algorithm presented here shares the general algorithmic ideas with those in the previous sections; however, the details of the algorithm are somewhat different. Our model of computing is the concurrent read, exclusive write (CREW) parallel random access machine (PRAM). Again, we only allow comparisons of edge costs, which we assume can be accomplished in unit time.

We describe a different decomposition of the problem that can be done in parallel, and we also show that a reduced tree $T'$ with $O(n/\log n)$ vertices, computed by applying the decomposition twice, can be verified in $O(\log n)$ time (as we now have a processor for each vertex in $T'$). This yields an algorithm that runs in $O(\log n)$ time using $\Theta((n+m)/\log n)$ processors. For the parallel version of the verification algorithm, we assume that the graph is stored in the shared memory of the machine, and that edges are stored in adjacency lists at the nodes of the graph. Since we use the sequential linear–time result to verify the microtrees, no precomputation is needed for the parallel portion of the algorithm.

### 2.4.1 Decomposition

We decompose $T$ into microtrees differently from the sequential case. We first build a compressed tree $C$ (similar to [31]) in the following way. We compute $size(v)$ for each vertex in $T$, where $size(v)$ is the number of nodes in the subtree of $v$. Define an edge $(v, p(v))$ of $T$ to be *heavy* if $size(v) > \frac{1}{2}size(p(v))$. An edge is considered *light* otherwise. This partitions $T$ into a collection of heavy paths and light edges. We store each heavy path in $C$ by creating a single node $a$ that represents the highest node on the path. We make all other nodes on the path children of $a$ and we call $a$ the *apex* of the heavy path. An example of this decomposition appears in Figure 5. The cost of the edge to each child $v$ of $a$ is the maximum edge cost on the heavy path between $a$ and $v$ and can be computed using prefix–sum [33]. We additionally store each heavy path as a linked list for further processing. Note that $size(p(p(v))) > 2size(v)$ after this construction. Therefore the depth of $C$ is $O(\log n)$. This transformation can be done using $O(\log n)$ time and $\Theta(n/\log n)$ processors using parallel tree contraction[33].

Next we break the compressed tree into the reduced tree $T'$ and the microtrees. In the following decomposition, we cut the microtrees only from the bottom of $C$. We cut the edge $(v, p(v))$ in $C$ if $size(v) \leq (\log n)^{1/2} < size(p(v))$. Each subtree removed from the bottom of $C$ is a microtree, and the reduced tree $T'$ is the remainder. In $C$, since the size doubles every other level up the tree, there are at most $2n/(\log n)^{1/2}$ nodes with size between $(\log n)^{1/2}$ and $2(\log n)^{1/2}$, at most $n/(\log n)^{1/2}$ nodes with size between $2(\log n)^{1/2}$ and $4(\log n)^{1/2}$, and so on. The number of nodes in each size category is bounded by a geometric series and therefore $T'$ has size $O(n/(\log n)^{1/2})$. We use the root of a microtree, say vertex $v$, to identify the tree: we label each vertex in the microtree with $v$. We call the node $p(v)$ a subtree root as in the previous section and we store the edge $(v, p(v))$ with the microtree of $v$.

Cutting the microtrees only from the bottom of $C$ (as opposed to the sequential case) helps to simplify the processing of each nontree edge. We replace each nontree

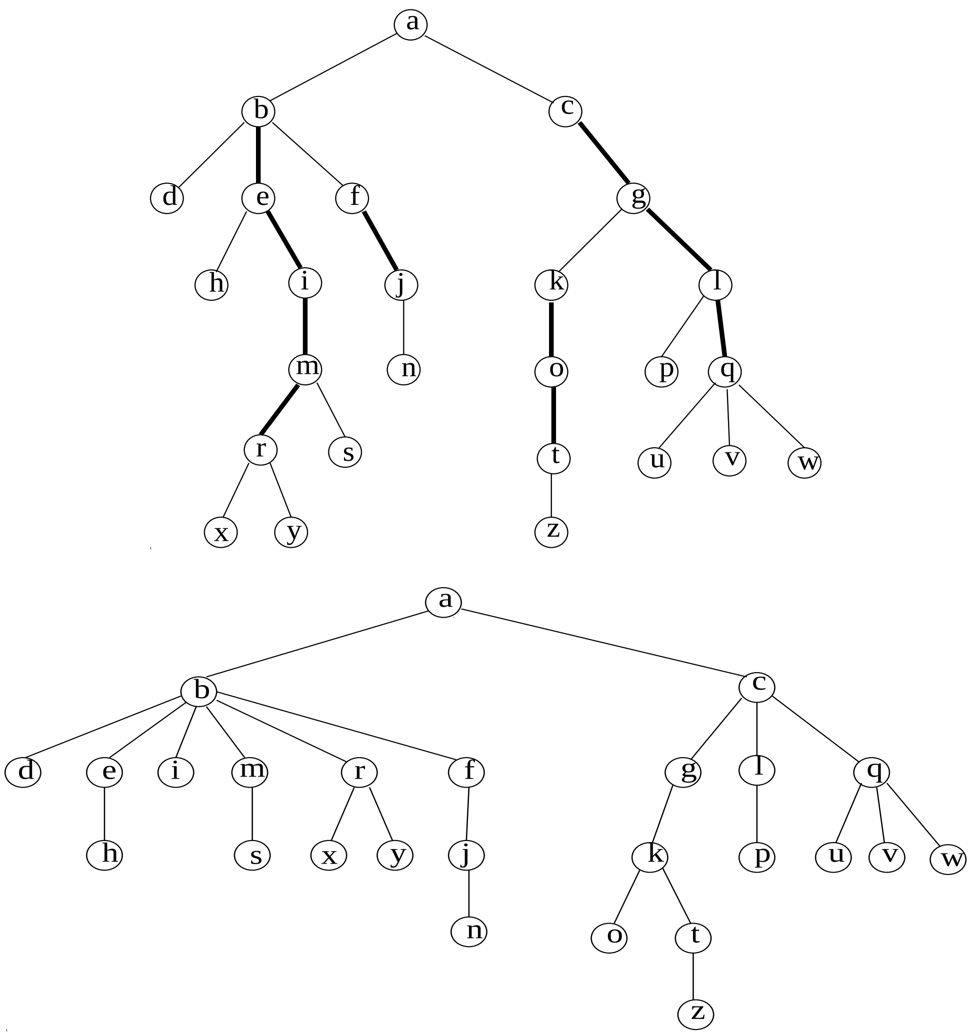


Figure 5: A tree with indicated heavy edges and its corresponding compressed tree.

edge by up to four edges. To do this, we first compute the nearest common ancestor $nca(x,y)$ for each nontree edge $(x,y)$ using the $O(\log n)$ time parallel algorithm of [56]. We replace the edge $(x,y)$ by two edges $(x, nca(x,y))$ and $(y, nca(x,y))$ which are given cost equal to that of $(x,y)$. In order to avoid write conflicts, we store the new edges only at the endpoints $x$ and $y$ and not at $nca(x,y)$. We replace each of the new edges in turn; if the node $x$ is in a microtree, then we replace an edge $(x, nca(x,y))$ by $(x,r)$ and $(r, nca(x,y))$ where $r$ is the subtree root of $x$. Note that we can store the new edge $(r, nca(x, y))$ at $r$ by computing a post-fix ordering of the nontree edges in the microtrees of $r$ and writing each edge in an array position indexed by the ordering. After this decomposition there can be many edges from a node $v$ to its subtree root; we compute and keep only the minimum–cost edge. We repeat the cutting of the microtrees and replacing of the nontree edges one more time, cutting at a node $v$ if $size(v) \leq \log n < size(p(v))$. This reduces the size of the remaining problem, yielding a reduced graph $G'$ and associated spanning tree $T'$ with $O(n/\log n)$ nodes. The reduced graph $G'$ can still have $O(m)$ edges. The time to compute this decomposition is $O(\log n)$ given $O((n + m)/logn)$ processors.

We have broken the verification of the spanning tree into three pieces: the verification of the microtrees, the verification of the reduced tree $T'$, and the verification of the heavy paths. In the remainder of this section, we show how to solve each of these subproblems.

### 2.4.2 Verification of the microtrees

Given this decomposition of the graph, we can verify the minimality of the microtrees in parallel. Each microtree can be verified in time proportional to its size using the linear time sequential algorithm of the previous chapter. In the following, we show how to assign each processor a set of microtrees of size $O(\log n)$ to verify. When we cut the microtrees from $C$, each processor cuts up to $O(\log n)$ microtrees from $C$. Thus each processor writes the size of the microtrees that it cuts in an array of length $O(\log n)$. We collect all of the non-zero entries from all of the processors into one large array. Using prefix-sum [33], we can compute a starting and ending array index for each processor so that each range contains microtrees having total size at most $O(\log n)$. Thus each processor can then verify the microtrees in the array positions between its starting and ending index in time $O(\log n)$.

We must still verify the heavy paths and the reduced tree $T'$. We first describe the verification of the reduced tree $T'$ and we show later how to verify the heavy paths using similar techniques.

### 2.4.3 Verification of $T'$

The next step is to verify the reduced tree $T'$ that has $O(n/\log n)$ nodes and $O(m)$ nontree edges using $O((n+m)/\log n)$ processors and $O(\log n)$ time. We binarize $T'$ in a balanced way as in [31]. This is a recursive procedure that replaces every node $v$ with more than two children by a new binary node. The children of $v$ are split into two sets that become the children of the new binary node. The new tree edges are given cost $-\infty$ so that they do not affect the minimality of $T$. This procedure is repeated until there are no nodes with more than two children. If we split the children of $v$ into two sets such that the sum of the sizes of each set is roughly equal, then the depth of the binarized tree is still $O(\log n)$ [31], and the number of nodes in the tree is only increased by a constant factor. Since we have a processor for each node in the tree, we can assign $d$ processors to every node with $d$ children. Given an initial ordering of the children at every node, each split can be performed in constant time. Since the depth of the tree is increased by at most $O(\log n)$, this binarization can be done in $O(\log n)$ time.

We now have a binary tree of depth $O(\log n)$ with $O(n/\log n)$ nodes to verify. Additionally, each nontree edge is from a node to one of its ancestors. We have a processor for each node in the tree as well as one for every $O(\log n)$ nontree edges. We verify $T'$ using pipelining as follows. We assign each node a level based on its depth. The nodes at the lowest depth are assigned level 0, the nodes one step higher in the tree are assigned level 1, and so on to the root of the tree. At each node in the tree, we bucket sort the nontree edges according to the level of their ancestor endpoint. Note that this only requires $O(\log n)$ time with $\Theta((n+m)/\log n)$ processors[53] and $O(n+m)$ total space. All edges in the same bucket start and end at the same nodes and thus cover the same tree path. Therefore only the minimum-cost edge in the bucket needs to be considered. We can compute the minimum edge in each bucket and discard the rest of the edges in $O(\log n)$ time since we can assign $\Theta(l/\log n)$ processors to each bucket of size $l$.

We can now pipeline the comparisons of the nontree edges to the tree edges, noting that $T$ is a minimum spanning tree only if every tree edge has cost at least as large as every nontree edge with which it is compared. For ease of description, we assign the tree edge $(v, p(v))$ to each node $v$ in the tree. The pipelining process starts at the bottom of the tree and works its way toward the root. The nodes that are already active in the process make a comparison and pass an edge up to their parent, activating their parent. Thus at step $i$ nodes at level $i-1$ become active. Each node receives two edges that end at the same level, say $k$, and computes the minimum value of those two edges and its nontree edge to level $k$ (if it exists). This minimum cost edge is compared against the tree edge and passed to the parent node. Each nontree edge is discarded as soon as it is compared to a smaller nontree edge or it reaches its ancestor endpoint. The leaves can have both a tree edge and a nontree edge to their parent, so the process begins with each node at level 0 comparing the costs of those two edges (if the nontree edge exists). In the next step, the nodes at level 0 compare the cost of their tree edge with the nontree edge to their grandparent (if it exists). They then pass this nontree edge up to their parent. At step $i$, the level 0 nodes compare the cost of their tree edge to their nontree edge to level $i$ (if the edge exists), the nodes at level 1 compare their tree edge with the nontree edge to level $i-1$, and so on. Since the depth of the tree is $O(\log n)$, this continues for $O(\log n)$ steps and finishes when all edges have been discarded. The spanning tree $T'$ is a minimum spanning tree of $G'$ if and only if no nontree edge is compared against a larger tree edge.

### 2.4.4 Verification of the heavy paths

The total length of all heavy paths that must now be verified is $O(n)$. There might be $O(m)$ nontree edges incident to the path, however. We describe how to verify the heavy paths in $O(\log n)$ time using $O((n+m)/\log n)$ processors. The verification of the heavy paths is quite similar to the verification of the compressed tree $C$. We begin with the assumption that all incident nontree edges are stored at both endpoints of the heavy path, and we show below how to actually store the edges in such a way. We transform a heavy path of length $k$ into a balanced binary tree with $k$ leaves; we call the new binary tree $H$. For ease of description, we place the edge costs of the heavy path at the leaves of $H$. The cost of the deepest edge on the path is placed at the leftmost leaf of $H$, the next edge up the path is assigned to the next leaf of $H$ to the right, and so on until the highest edge of the path is assigned to the rightmost leaf of $H$. At each internal node of $H$, we compute and store the maximum edge cost in its subtree. This is an easy form of expression tree evaluation and can be handled in $O(\log n)$ time [33].

The verification of a nontree edge in $H$ involves comparing the nontree edge against some of the max values stored in the internal nodes as follows. We again replace each nontree edge $(u,v)$ in $H$ by two edges: $(u, nca(u,v))$ and $(v, nca(u,v))$. Let $u$ be to the left of $v$; we define the edge $(u, nca(u,v))$ to be a *right* edge since we compare this edge cost against leaves to the right of $u$, and similarly, we define $(v, nca(u,v))$ to be a *left* edge. To verify a right edge $(u, nca(u,v))$ we perform the following test at every node $x$ on the path in $H$ from $u$ to $nca(u,v)$. If the left child of $x$ also lies on the path from $u$ to $nca(u,v)$ then we compare the max cost stored at the right child of $x$ with the cost of $(u, nca(u,v))$. Otherwise, when the left child of $x$ is not on the path no comparisons are needed. We perform the symmetric comparisons to verify a left edge. When we are finished with the above comparisons, the maximum tree edge on the path from $u$ to $v$ has been compared to the nontree edge $(u,v)$.

$H$ can be verified using similar methods to those used to verify the compressed tree $C$. The process is nearly identical: we cut microtrees from $H$ as we did for $C$, except this time each microtree represents a sub–path, which is easily verified as a microtree. The remainder after the microtrees are removed is verified using the pipelining technique which we used to verify $T'$, with the only difference being that a local test is used (checking to see if the child is also on the path) to determine if a comparison is needed. What remains to be shown is how to store the nontree edges at the appropriate nodes of the heavy paths. The edges which must be considered are those that enter and leave the heavy path through some node other than the apex, since those that pass through the apex have been properly verified. Because of the techniques which we wish to use to perform comparisons, we must store each nontree edge at both of its endpoints on the heavy path. The task will be broken into two pieces; we first move the nontree edges to their lowest ancestor in $T$ on the heavy path. Since the endpoints of the edges that were moved to the heavy path are not necessarily the proper nodes, we correct this in the second step.

If a heavy path node is cut from its apex node to form a microtree, then the heavy path node is a microtree root and storing the appropriate nontree edges at the microtree root can be done in exactly the same way that the edges were moved to the subtree roots in the original decomposition (compute an ordering and store the edges in an array indexed by the ordering). The remaining edges to consider are those whose $nca$ in $T'$ is a heavy path apex. Thus we can restrict our attention to the reduced tree $T'$, which has $O(n/\log n)$ nodes. If we use the previously described pipelining technique but replacing the $min$ operation with list concatenation, then this moves a list of nontree edges up the tree $T'$. If we stop moving the list up the tree when we reach the child of the appropriate heavy path apex, then the heavy path node reached is the lowest ancestor in $T$ for each of the edges stored there. This gives one of the endpoints for each of the edges. To find the other endpoint for each edge we must store some additional information with each edge. Remember that we store both $(v, nca(u, v))$ and $(u, nca(u, v))$ in the original decomposition. We store three additional pieces of information with each nontree edge at the heavy path: the endpoints of the edge in $T'$, the $nca$ in $T$, and the $nca$ in $T'$. There are two kinds of edges in the list at the heavy path node $v$: those whose $nca$ in $T$ is $v$ and those whose $nca$ in $T$ is higher than $v$.

If the $nca$ in $T$ is higher in the path, no other processing needs to be done; the other endpoint is simply the $nca$. If the $nca$ is $v$ itself, we do not know the other endpoint on the path for this edge. We compute it in the following way. We write at each apex of a heavy path its depth in $T'$. At each leaf $l$ of $T'$, we store an array of size $O(\log n)$. If the path from $l$ to the root of $T'$ contains an apex $a$ at depth $i$, then entry $i$ in the array contains the child of $a$ which is on the path from $l$ to the root. In other words we store the node where the path from $l$ to the root enters a heavy path. If a nontree edge where we need to find an endpoint in the heavy path has an endpoint $x$ in $T'$ and the apex of the heavy path is at depth $k$, then we find the missing endpoint by looking at position $k$ in the array stored at $x$. The entries in these arrays are easily filled by traversing the path to the root from each leaf, taking $O(\log n)$ time and making use of the fact that we have a processor for each node in the tree. Once all edges are stored at both endpoints, the path is verified by using the bucket sorting and pipelining exactly the same way as we verified $T'$.

## 2.5 Parallel sensitivity analysis

We present a parallel algorithm for the sensitivity analysis problem described in Section 2.3. We modify the sequential algorithm of Section 2.3 to produce a parallel algorithm. The techniques are much like those used for the parallel verification algorithm. Our results here are the parallel equivalents of the algorithms presented in

Section 2.3; a randomized $O(\log n)$ time and $\Theta(n+m)$ work algorithm and an optimal deterministic algorithm. The best known upper bound on the running time of the deterministic algorithm is $O(\alpha(m,n)\log n)$, but the exact bound is not known. As in the verification algorithm, we use $\Theta((n+m)/\log n)$ processors.

To perform the sensitivity analysis, we must find the maximum cost tree edge on the tree path determined by each nontree edge and also the minimum cost nontree edge that spans each tree edge. We consider each of these cases separately. We first show how to find the minimum cost nontree edge for each tree edge.

### 2.5.1 Sensitivity analysis of the tree edges

This portion of the sensitivity analysis algorithm closely resembles the parallel verification algorithm of Section 2.4. The decomposition of the tree is exactly as for the verification algorithm. We perform sensitivity analysis of the microtrees using the sequential sensitivity analysis algorithm from Section 2.3. To perform sensitivity analysis of the reduced tree $T'$, we modify the parallel verification algorithm so that during the verification of the reduced tree $T'$ we store at every tree edge the minimum nontree edge cost that is compared to the tree edge. This is exactly the information needed for the sensitivity analysis of the tree edges of $T'$. Note that the minimum cost nontree edge covering a tree edge will be compared to the tree edge cost during the verification algorithm. The only remaining tree edges that need sensitivity analysis information are those stored in the heavy paths.

We again modify the verification algorithm to perform this task. During the verification of the heavy paths, nontree edge costs are compared against maximum costs from subtrees of the balanced binary tree $H$. We additionally store the minimum nontree edge that is used in a comparison at each internal node of $H$. When all comparisons have been performed, we propagate the sensitivity analysis information down the tree. Any nontree edge whose cost is stored at a node $v$ in the binary tree covers all leaves of the subtree rooted at $v$. Therefore each node sends its minimum nontree edge cost to its children. If the children are internal nodes then they compare the received value with the value already stored at their node and send the minimum of these values to their children. We can propagate the sensitivity analysis information down the tree in $O(\log n)$ time, and this completes the sensitivity analysis for the tree edges. The next section describes the sensitivity analysis for the nontree edges.

### 2.5.2 Sensitivity analysis of the nontree edges

In this section, we describe how to determine the maximum tree edge on the tree path determined by each outside edge $(u, v)$. As mentioned above, we use the sequential sensitivity analysis algorithm from Section 2.3 to handle the microtrees, so we only need to worry about the heavy paths and the reduced tree $T'$. In both cases we rely on the unpublished result of Alon and Schieber [4]. Using their algorithm, given $O(\log n)$ preprocessing time and $O((n+m)\alpha(m,n)/\log n)$ processors, queries of the form "What is the maximum value on the tree path between $u$ and $v$?" can be answered in $O(\alpha(m,n))$ time using a single processor.

In this reduced tree, we simply use the Alon and Schieber result to perform sensitivity analysis on $T'$ since $\alpha(m, n/\log n)$ is $O(1)$. The preprocessing therefore takes $O(\log n)$ time and each outside edge can be processed in $O(1)$ time. Since we have $\Theta(m/\log n)$ processors, we can process all of the outside edges in $O(\log n)$ time.

To perform sensitivity analysis on the heavy paths, we use the same binary tree transformation as in the verification case. We make the additional observation that we can apply the sensitivity analysis algorithm on this binary tree recursively. Note that no new heavy paths are created by the recursive solution. Therefore the solution only needs to perform sensitivity analysis on some new microtrees and a new reduced graph $T'$.

## 2.6 Remarks and open problems

We have illustrated by means of examples a general technique of speeding up divide-and-conquer algorithms by a preprocessing and table lookup technique. A curious phenomenon is that the technique can give algorithms running in time minimum to within a constant factor, but for which we cannot presently offer a tight asymptotic time analysis. This is the case for both our sequential and parallel deterministic minimum spanning tree sensitivity analysis algorithms and for Larmore's convex matrix searching algorithm [36]; all have running times somewhere between linear and an inverse Ackerman function times linear. Providing tight analysis of these algorithms amounts to bounding the number of comparisons needed to solve the corresponding problems. Obtaining tight bounds remains open. A related question is whether the randomized maxima-finding algorithm of Goddard, King, and Shulman [28] can be made deterministic.

Another question is whether our model for computations not involving edge costs can be weakened from a random-access machine to a pointer machine. The only feature of a random-access machine we have actually used is the ability to make multiway branches to look up the special-purpose algorithms for processing the microtrees. It may well be the case that binary branching suffices for this purpose, and hence that our results hold for pointer machines with binary comparison of edge costs (and edge cost subtraction for sensitivity analysis). In the parallel case the algorithms given here are for the CREW PRAM model. Can these algorithms be transformed into EREW algorithms? Another interesting question is to reduce the gap in efficiency between the verification problem and finding minimum spanning trees.

The technique we have illustrated is not limited to comparison-based problems. We can allow arithmetic operations in the decision trees used to solve the subproblems. Testing the correctness of such a decision tree amounts to testing the validity of a first-order sentence about the real numbers. Such sentences can be tested in double-exponential time [6], which suffices for the use of the method: we merely reduce the size of the subproblems to double-logarithmic, triple-logarithmic, or further, as needed. As an example, the technique can be applied to the $O(n \log^* n)$-time algorithm of Chazelle [12] for triangulating a simple $n$-sided polygon, to produce an algorithm running in time minimum to within a constant factor. The bound for this algorithm is in fact $O(n)$, because of the even more recent result of Chazelle [13] giving an explicitly linear-time algorithm. Further applications remain to be discovered.

# Chapter 3

# Concurrency in an $O(\log \log N)$ priority queue

## 3.1 Introduction

In this chapter we consider a pipelined model of parallel computing and present two optimal pipelined algorithms for the bounded universe priority queue problem. Our goal is to increase the throughput of a priority queue data structure by pipelining the operations. In our model each operation on the data structure is executed by a single processor and the time for the operation to complete is the same as in the sequential version, but our new pipelined data structure allows a new operation to begin every time step. Using multiple processors, this gives a way to increase the throughput of the data structure without increasing the latency time of a single operation. An application of our data structures to the longest increasing subsequence problem is presented as well.

A priority queue is a data structure supporting insertions, deletions, and find min operations [1]. Using balanced trees, these operations can be supported in $O(\log n)$

time, where $n$ is the number of elements in the queue. If the universe of elements is restricted to the range $[0 \ldots N]$, then these operations can be performed in $O(\log\log N)$ time using the van Emde Boas priority queue [20, 21]. This result requires the addressing power of the RAM model of computing, and the running time no longer depends on the number of elements in the queue. The van Emde Boas algorithm is therefore an improvement whenever $n > \log N$. The range of operations that the van Emde Boas algorithm supports is expanded to include find predecessor (successor) operations as well. These operations require $O(\log\log N)$ time as well, and the data structures that we present support these operations.

As mentioned above, we are concerned with versions of the priority queue that allow parallel processing of the priority queue operations. This form of the priority queue is important in various parallel algorithms, e.g. multiprocessor scheduling, graph search, and branch-and-bound. If the size of the universe of the priority queue is unrestricted, it is known how to perform queue operations concurrently [52, 10]. In these algorithms, multiple operations are allowed to access the queue, and each operation requires $O(\log n)$ time, and the throughput is increased by the use of multiple processors. In this chapter, we show how to perform concurrent priority queue operations in a restricted universe. Like the previous algorithms, our goal is to increase the throughput of the data structure given some number of processors.

In our algorithms, we assume a synchronized, shared memory parallel computing environment. We define a time step to be a constant amount of time, whose length we define later. Each of the concurrent operations executes one unit of work every time step, and one new operation is allowed to begin at each time step. Two concurrent algorithms are presented in this model. In the first algorithm, each individual insertion, deletion, or find min operation completes in $O(\log\log N)$ amortized time and $O(\log N)$ worst case time. This yields a parallel algorithm for maintaining a priority queue in which a sequence of $m$ operations can be performed in $O(\log N + m)$

time using $\Omega(\log\log N)$ processors. When $m$ is $\Omega(\log N)$ this represents an optimal amount of work due to the lower bound in [2].

In the second algorithm, the operations complete in $O(\log\log N)$ worst–case time, but the algorithm requires $O(N^\epsilon)$ sequential preprocessing time for any fixed $\epsilon$. This algorithm also yields a parallel algorithm for maintaining a priority queue. Given $\Theta(\log\log N)$ processors, a sequence of $m$ operations can be performed in $O(\log\log N + m)$ time plus $O(N^\epsilon)$ sequential preprocessing time.

The algorithm given by van Emde Boas [21] requires space linear in $N$. This bound can be reduced to $O(N/(\log N)^c)$ for any fixed $c$ using some techniques pointed out later in this chapter. Willard [64] has given an algorithm that requires space linear in $n$; the drawback is that the insertion and deletion algorithms are randomized, and thus the bounds for running times of these operations is expected. Our algorithms require space $O(N/(\log N)^c)$. The techniques used in the Willard paper are not directly applicable to our algorithms; this will be discussed more in the remarks.

We describe the sequential version of each algorithm before giving the concurrent version. The amortized algorithms appear in Sections 3.2 and 3.3, and the worst–case versions appear in Sections 3.4 and 3.5. Section 3.6 presents top–down algorithms for performing splits and joins of 2–4 trees, which are required only for the amortized concurrent version of the priority queue. An application of the concurrent priority queues, a parallel algorithm solving the longest increasing subsequence problem, is presented in Section 3.7, and Section 3.8 presents remarks and open problems.

## 3.2 An amortized $O(\log\log N)$ priority queue

This section presents a sequential priority queue for which the amortized time of an operation is $O(\log\log N)$ and the data structure uses space $O(n + N/\log N)$ where $n$ is the number of elements present in the queue.

We use a two level scheme to form the data structure. Let $t$ be the smallest power of 2 larger than $N/\lceil \log N \rceil$. The top level is a fixed complete binary tree, $T_{top}$, with $t$ leaves. The depth of $T_{top}$ is $\lceil \log N - \log\lceil \log N \rceil \rceil$. We subdivide the universe $[0, \ldots, N]$ into contiguous blocks of size $\lceil \log N \rceil$ (with the last block possibly shorter) that are assigned to the leaves of the tree. We assign the $i^{th}$ block to the $i^{th}$ leaf of $T_{top}$ starting from the left.

The lower level is comprised of up to $N/\lceil \log N \rceil$ balanced trees of size $O(\log N)$. Each leaf of the fixed tree contains a pointer that is either null or points to a lower level tree. We call a leaf of $T_{top}$ active if it has a non–null pointer, and we call the lower level tree to which it points its *lower tree*. The tree $T_{top}$ and lower tree structure is shown in Figure 6. The internal nodes of $T_{top}$ have a mark bit that is set for all ancestors of active nodes, as well as pointers to the rightmost and leftmost active leaves in their subtree. Initially the leftmost leaf points to an empty tree, the pointers at the other leaves are null, and only the leftmost path in $T_{top}$ is marked. To find neighboring active leaves quickly, we keep a doubly linked list of the active leaves. The space used by $T_{top}$ is $O(N/\log N)$ and the total space used by the lower trees is $O(n)$ where $n$ is the number of elements in the queue.

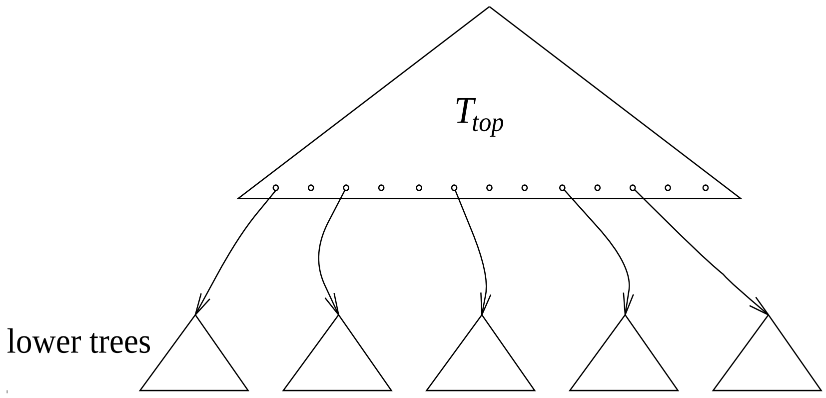


Figure 6: The tree $T_{top}$ and lower tree structure.

If a leaf $l$ is active, then its lower tree contains the elements of its block that are in the queue, and as shown below, we allow additional elements from the queue to be stored in this lower tree. Consider the insertion of an element $x$, and let $l$ be the leaf of $T_{top}$ assigned to the block containing $x$. If $l$ is active, we insert $x$ into its lower tree. Otherwise, we search in $T_{top}$ for the first active leaf, $l'$, to the left of $l$. To find $l'$, we perform binary search on the tree path from $l$ to the root of $T_{top}$ to find the lowest marked ancestor of $l$. At this node, either the pointer to the leftmost active leaf points to the first active leaf to the right of $l$ or the pointer to the rightmost active leaf points to $l'$. We can determine which case holds in constant time. If we reach the active leaf to the right of $l$, then we use the doubly linked list to reach $l'$. Then we insert $x$ into the lower tree of $l'$. Thus each lower tree can contain elements from the queue that are from blocks to its right. We allow this because we cannot afford to mark the path from a leaf of $T_{top}$ to its root for every insertion, so we instead use the first non–empty tree to the left.

We also cannot afford for the lower trees to become too large, so we split a lower tree $t$ when it contains $3\log N$ elements into two trees, $t_1$ and $t_2$, where all elements of $t_1$ are smaller than those of $t_2$. We want that the resulting trees have roughly equal size and do not contain elements from the same block. To achieve this, we split $t$ only along block boundaries. Thus the size of $t_1$ and also $t_2$ is at least $logN$ and at most $2\log N$. The leaf that pointed to $t$ now points to $t_1$, and the leaf whose block contains the smallest element of $t_2$ points to $t_2$. The later leaf becomes active and the path from it to the root is marked and the pointers updated.

To search for an element in the queue, we perform binary search on the path in $T_{top}$ as in the insertion, followed by a standard tree search in the lower tree. To perform a deletion, we simply search for the element and then perform a delete in the lower tree. When we delete the element, we check if the lower tree is now empty. If it is, we remove the empty tree and update the path from its leaf to the root appropriately. Note that this means that we do not maintain any lower bound on the size of the lower trees, which becomes important only in the concurrent version.

The operations consist of a binary search on a path of length $O(\log N)$ and a search, insertion, or deletion in a balanced tree of size $O(\log N)$, each of which can be performed in $O(\log \log N)$ time. Some insert operations additionally mark a path from a leaf to the root and split a lower tree. This takes time $O(\log N)$, which can be amortized over the last $\Theta(\log N)$ insertions into the tree since it was last split. The cost of unmarking the path due to an empty tree is amortized against the $\Theta(\log N)$ deletions since its last split. Thus each operation takes $O(\log \log N)$ amortized time.

## 3.3 Concurrent operations in the amortized priority queue

The scheme presented in the previous section gives a way to maintain a priority queue in amortized $O(\log \log N)$ time. The goal of this section is to achieve the same time bounds for each individual operation, but modify the operations so that several operations can access the priority queue concurrently. The data structure remains basically the same for the concurrent version, but the algorithms for accessing the data structure change to allow concurrent access.

### 3.3.1 Concurrent searching

We begin by describing the search algorithm and later describe the insert and delete operations. The $T_{top}$ structure remains exactly as in the sequential version, so to start a search operation, we perform binary search on the path in $T_{top}$ to find the proper lower tree. We force the following additional constraint on the binary search procedure. Since the distance from a leaf to the root of $T_{top}$ is $\lceil \log N - \log \lceil \log N \rceil \rceil$, the binary search takes at most $\lceil \log \log N \rceil$ time steps to find the lowest marked node on this path if we execute one iteration of the binary search per time step. We force all such searches to take exactly $\lceil \log \log N \rceil$ time steps by requiring operations that finish early to wait. This synchronization implies that at each time step, at most one new operation begins at the lower trees.

The lower tree reached by the search algorithm may not be the correct tree due to relaxations in the marking of internal nodes and the splitting of lower trees. The details of these relaxations appear below, but for now we simply assert that the correct tree can be reached in constant additional time.

We specifically choose the lower trees to be 2–4 trees [57] since it is known how to perform concurrent insertions and deletions in 2–4 trees [10, 19, 43], and we present algorithms to perform concurrent split and join operations in Section 3.6.

### 3.3.2 Concurrent insertion

As in the search algorithm, we find the correct lower tree for an item $x$ to be inserted. Then we perform a top–down insertion of the item $x$, which is accomplished easily since we have chosen the lower trees to be 2–4 trees. The synchronization of the binary search in $T_{top}$ guarantees that at most one new operation begins at a lower tree at each time step. The top–down insertions are, therefore, never blocked.

When an insert completes, we must split the lower tree if it becomes too large. In the sequential case, we were allowed to spend time proportional to the size of the tree to perform the split. In the concurrent version, however, the split must be performed while other insert and delete operations continue to act on the tree. Thus we create a top–down splitting procedure, that fulfills the same requirements as in the sequential case: the resulting trees must be of roughly equal size and they may not both contain elements from a common block. We choose an element $s$, called the split value, and partition a lower tree $t$ into $t_1$ and $t_2$ such that all elements in $t_1$ are smaller than $s$ and all elements in $t_2$ are at least as large as $s$. If we always choose $s$ to be a multiple of $\lceil \log N \rceil$, then $s$ is a block boundary, and we are guaranteed that $t_1$ and $t_2$ will not contain elements from a common block.

In the concurrent version, we must know the split value when we begin a split operation, because after a constant number of time steps, a subsequent search operation must be able to determine whether to continue the search in $t_1$ or in $t_2$. If we do not know the split value until the end of the split, then search operations might be blocked. To help find a split value, we keep a pointer in each lower tree to a leaf that will be roughly in the middle of the tree when the tree is split, see figure 7. We now describe how to maintain this pointer for a lower tree $t$ that has never been split and

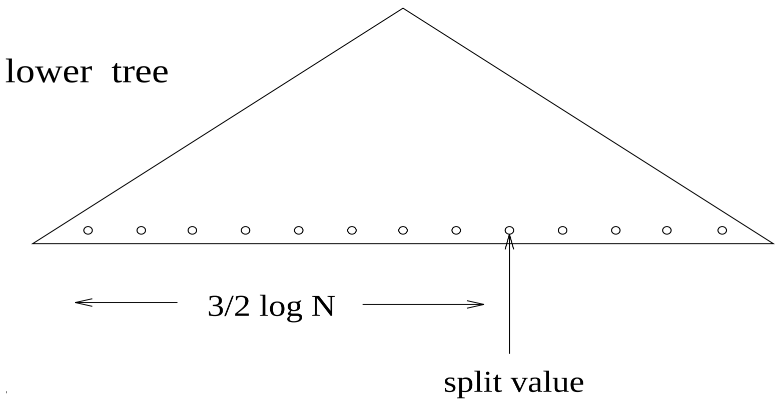


Figure 7: A lower tree and its split value pointer.

later we describe how to handle a tree that has just been split. If $t$ has never been split, then the pointer stays at the rightmost leaf of $t$ until $t$ has size $\lceil 3/2 \log N \rceil$. Once $t$ has this size, we maintain as an invariant that there are $\lceil 3/2 \log N \rceil$ nodes to the left of the pointer. If the inserted item is smaller than the item at our pointer, then we move our pointer one leaf to the left, otherwise it remains in place. Similarly, if a node to the left of the pointer is deleted then the pointer is moved one step to the right. To be able to move the pointer as necessary, we keep a doubly linked list of the leaves of the lower trees. Thus there are $\lceil 3/2 \log N \rceil$ nodes to the left of our pointer during these operations, and therefore when it is time to split $t$, the pointer is at a node near the middle of $t$ as described below.

When a split operation begins, there can be other operations acting on the tree. Because the pointer is moved only when an insert or delete operation completes, the pointer location does not reflect these uncompleted operations. Since the tree is of size $3 \log N$ when the split begins and only one operation can be at each level in the tree, there can be at most $\log 3 + \log\log N$ operations acting on the tree. Thus the size of the trees resulting from the split can be increased or decreased by $\log\log N + O(1)$. Since we have chosen to split only along block boundaries, we get the actual split value from the pointer by rounding the value stored at the pointer to the nearest multiple of $\lceil \log N \rceil$. This split value guarantees that the resulting trees have size between $\log N - \log\log N - O(1)$ and $2\log N + \log\log N + O(1)$, and have depth $\log\log N + O(1)$. Thus our split value is roughly in the middle of the tree if we are given a tree that has never been split.

Now we must consider the case of a lower tree $t$ that has just been split. Such a lower tree may contain more than $\lceil 3/2 \log N \rceil$ nodes, and thus the pointer is not at the correct location if we place it at the rightmost leaf of $t$ and run the above algorithm. To correct the pointer location, we keep a counter that is initially $size(t) - \lceil 3/2 \log N \rceil$, or in other words, the counter tells us how many steps to the left the pointer should move in order to maintain our invariant. At each time step we move the pointer one extra step to the left in addition to the regular pointer movement, and we decrement the counter. We continue the extra pointer movement until the counter is zeroed. Since a tree that is the result of a split has at most $2\log N + \log\log N + O(1)$ nodes, the largest that the counter can be is $\frac{\log N}{2} + \log\log N + O(1)$. Therefore we know that the counter can be reduced to zero before the next split operation takes place.

We have shown how to find the split value, so now the remaining task is to actually split the 2–4 tree from the top down. The split procedure, described in Section 3.6, maintains two trees while performing the split operation: the left tree containing the values smaller than the split value and the right tree containing the rest. At each time step, the split procedure moves one step down the initial tree and therefore finishes after $\log\log N + O(1)$ time steps.

Since it is possible that either the left or the right tree is empty for some number of time steps, some operations may have to wait while such a tree is empty. During every time step that an operation waits, a split operation steps one level lower in the tree, which means that the resulting tree has depth one step less. Thus each time step that an operation waits means that the height of the tree that it must traverse is reduced by one, and therefore the total waiting time together with the traversal time is still $\log\log N + O(1)$. Note also that once the tree is non–empty, the operation never waits; the splitting or joining process stays ahead of the traversal.

Once the split is performed, the leaf of $T_{top}$ that pointed to the original tree now points to the left tree, and the leaf of $T_{top}$ whose block contains the smallest element of the right tree points to the right tree. The later leaf becomes active and the path from it to the root is marked from the bottom–up and the pointers updated. Since other operations are still searching in $T_{top}$ while this path is being marked, it is possible that searches do not find the correct tree. Therefore, when a search completes, we test to see if the proper tree was located, and if not, we use the doubly linked list to move to the correct tree. Note that the path will be marked after $\log N$ time steps and that a new split operation cannot begin at either the left or the right tree in this time. This guarantees that a search is never off by more than one tree to the left or to the right.

### 3.3.3 Concurrent deletion

As in the sequential algorithm, we find the correct lower tree and delete the element from the lower tree. In the concurrent version we use a top–down 2–4 tree deletion [57] to remove the element. As we mentioned in the sequential version, we do not maintain a lower bound on the size of a lower tree. To maintain such a lower bound would require joining lower trees which would give rise to the following difficult situation. Suppose that some consecutive number of lower trees are about to need to be joined, and starting from the leftmost, they each perform a join, one tree per time step. The size of the resulting tree can grow rapidly. This will require performing split operations while the joins are being performed, but the problem is that it is not possible to maintain the split value while the tree is growing so rapidly. Because we do not maintain a lower bound on the lower tree size, we face a different problem. When a lower tree becomes empty as a result of a deletion, the path from the leaf of the $T_{top}$ tree to the root needs to be updated. Because other operations are accessing $T_{top}$ during the unmarking, it is possible that a binary search can reach a leaf in $T_{top}$ that is no longer active. Therefore our solution is to maintain a pointer from the emptied lower tree to the first active leaf to its left. Since this leaf can become inactive as well, at every time step that the deletion operation unmarks a node on the path to the root it updates the predecessor pointer of the leaf as well (which can be accomplished in constant time). Thus when an operation reaches an inactive leaf, it finds the correct lower tree using this pointer. Note that checking the predecessors requires concurrent reads, since many inactive leaves may have the same predecessor. After $\log N$ time steps, the unmarking of the path is complete and the predecessor pointer at the inactive leaf is no longer needed.

This completes the description of the amortized version of this data structure. The next two sections present the worst–case versions of both the sequential and concurrent variants of the data structure.

## 3.4 Worst case $\log\log N$ priority queue

This section presents a sequential algorithm for performing inserts, deletes, and find min operations in $O(\log\log N)$ worst–case time where the universe is the range $[0\ldots N]$. The algorithm and the data structures are quite different from those presented in the previous two sections. For the description of the algorithm, we use the notation presented in Mehlhorn [47], although the data structure was first presented in [20, 21]. The space used by the data structure presented here is $O(N\log\log N)$. We will show in the next section a simple technique to reduce the space to $O(n + N/(\log N)^c)$ where $n$ is the number of elements in the queue and $c$ is a fixed constant. For a further reduction in space see [64]. The insertion and deletion times using his algorithm become expected bounds. The lack of worst case bounds on his algorithm prevents us from using his space reduction techniques.

To perform insertions, deletions, and membership queries, a bit vector is an efficient data structure since we can perform each of the operations in $O(1)$ time. A priority queue must support find min operations as well; one possible solution is to keep a sorted linked list of the elements in the queue. The problem with only keeping a linked list is that insertions are not efficient. However, if we combine the data structures using pointers from the bit vector to the linked list, we can reduce our problem to the problem of finding predecessors as follows. If we wish to insert element $x$ into the list, and we can find the largest element $y < x$ that is in the queue, then we use the pointer from the bit vector entry for $y$ to find $y$ in the list. We insert $x$ into the list as the successor to $y$. Thus we have reduced the problem to that of finding predecessors in the bounded universe $[0\ldots N]$.

An outline of the recursive searching technique used to find predecessors follows. The idea is roughly to break the current search range, call it $[0\ldots N]$ into about $\sqrt{N}$ blocks of size about $\sqrt{N}$. We have two cases when searching for the predecessor of

$x$. If the block for $x$ already contains an element smaller than $x$, then we recurse on the block of $x$ to find its predecessor. If the block containing $x$ is empty or does not contain an element smaller than $x$, then we recurse on the set of blocks themselves. In this case, the largest element in the first non–empty block to the left of $x$ will be its predecessor. In each case, we recurse on only one $\sqrt{N}$ size sub–problem, so the recurrence relation describing the running time of the algorithm is $S(N) = S(\sqrt{N}) + O(1)$, which is $O(\log\log N)$[47]. The insertions and deletions use a similar strategy and achieve the same running time. The algorithms are presented in detail below.

We formally define the data structure as a recursive structure $T$. Since we may assume that $N$ is a power of 2, let $2^k = N$. We define $T_k$ to be a recursive data structure for the range $[0 \ldots 2^k]$. It contains an ordered doubly linked list $L$ of the items stored in $T_k$ and a bit vector $V$ from $[0 \ldots 2^k]$ where entry $i$ in the bit vector is set to 1 if and only if $i$ is in the queue. Each non–zero entry in the bit vector also contains a pointer to its corresponding item in $L$, and we have pointers to the largest and the smallest elements of $L$ as well. $T_k$ additionally contains an integer *size* that represents the number of elements present in the structure, an array $T_{bot}[0 \ldots \lceil k/2 \rceil]$ of pointers to $T_{\lfloor k/2 \rfloor}$ structures, and a $T_{\lceil k/2 \rceil}$ structure called $T_{top}$. Each sub–structure $T_i$ for some $i$ is a data structure supporting predecessor queries, insertions, and deletions in $O(\log i)$ time and supports find min and find max operations in constant time. See figure 8 for a description of this recursive tree structure.

The first check when searching for the predecessor of $x$ is to see that $x$ has a predecessor in $T$ (i.e. it is not smaller than all elements of $T$), and if not then we return an appropriate message. If $x$ has a predecessor, then let $x_{high}$ be $x$ div $2^{\lceil k/2 \rceil}$ (the high–order $\lceil k/2 \rceil$ bits of $x$) and $x_{low}$ be $x$ mod $2^{\lceil k/2 \rceil}$ (the low–order bits of $x$). We examine the top level structure $T_k$, and if the structure pointed to by $T_{bot}[x_{high}]$ is non–empty and the smallest element in that structure is smaller than $x$, then we

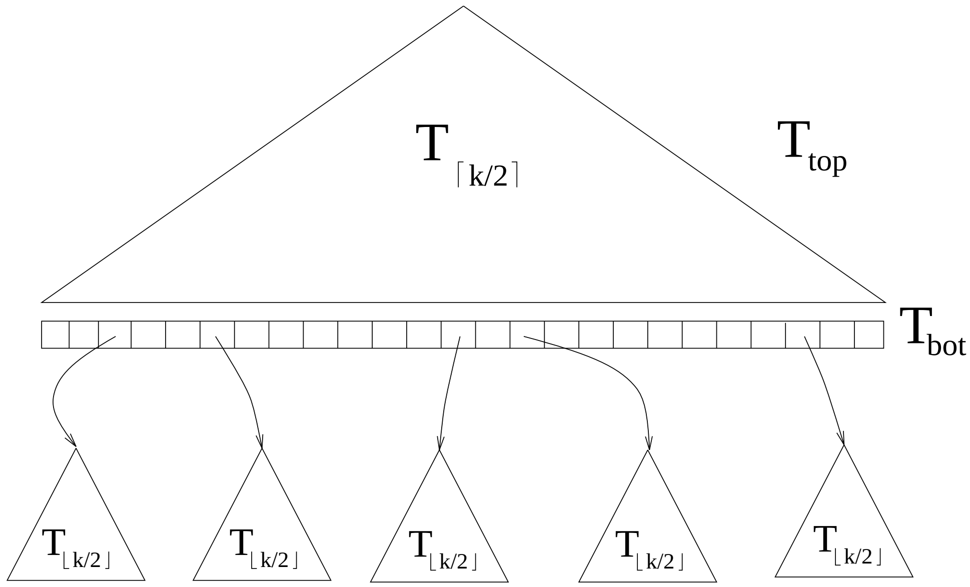


Figure 8: The recursive tree structure.

recursively search in that $T_{\lfloor k/2 \rfloor}$ structure for the predecessor of $x_{low}$. If the structure pointed to by $T_{bot}[x_{high}]$ is empty or its smallest element is larger than $x$, then we recursively search $T_{top}$ to find the predecessor of $x_{high}$. Let $p$ be the predecessor of $x_{high}$ in $T_{top}$. Then $p$ is the largest index such that $p < x_{high}$ and $T_{bot}[p]$ is non–empty. Therefore the largest element in the structure pointed to by $T_{bot}[p]$ is the predecessor of $x$, which can be retrieved in constant time using the largest element pointer. When the size of a structure is one, no recursion is needed, we simply examine the element and determine if that element is the predecessor.

The insert and delete operations are similar to finding predecessors. If $x$ is the smallest element, it is added to $L$ and $V$ in constant time, otherwise, we first find the predecessor $y$ of $x$ as described above. We insert $x$ into $L$ following the entry for $y$, and we set the bit $V[x]$ to 1. Next, we examine the $T_k$ structure, incrementing the size by one and updating the smallest and largest element pointers as needed. If the structure at $T_{bot}[x_{high}]$ is non–empty, then we recursively insert $x_{low}$ into the $T_{\lfloor k/2 \rfloor}$ structure stored there. If this structure is empty, we insert $x_{low}$ as a single element there (without recursion) and recursively insert $x_{high}$ into the $T_{top}$ structure since the entry at $T_{bot}[x_{high}]$ is no longer empty. In both cases, only one recursive call is made.

Note that during the recursive insertion calls, no further predecessor searches are required. Whenever a predecessor of $x$ is needed (i.e. $x$ is not the smallest element of the sub–structure), then the predecessor can be computed from $y$. This follows from the fact that the recursive insertion and predecessor queries follow exactly the same path in $T$ until $x$ is the smallest element of its structure. Thus all required predecessors can be found in the same time as one predecessor search operation.

The insertion of $x$ into a sub–structure $t$ which has size 1 is a special case. Let $y$ be the element already stored at $t$, and remember that $y$ has not been stored recursively inside $t$. First we store $y_{high}$ in $t_{top}$ and $y_{low}$ in a new structure at $t_{bot}[y_{high}]$. Note that both of these operations can be done in constant time since no recursion is required. Now, we can insert $x_{high}$ and $x_{low}$ appropriately. The critical point is that only one of these two insertions requires recursion. If $x_{high} = y_{high}$, then $t_{top}$ is a one element structure and $t_{bot}[x_{high}]$ is a two element structure where recursion is required. If $x_{high} \neq y_{high}$, then $t_{top}$ is the two element structure, but both $t_{bot}[x_{high}]$ and $t_{bot}[y_{high}]$ are one element structures. The time bound follows from the fact that only one recursion is needed as in the case of finding predecessors.

The deletion algorithm is the inverse of the insertion algorithm, and at each recursive step returns the structure to exactly as it would be if the insertion had never taken place.

## 3.5 Concurrent operations in the worst case priority queue

Now we present the version of the worst–case data structure that allows concurrent operations. The data structures and algorithms are similar to the previous sequential version, and our model of concurrency is identical to the one used for the amortized version. We use synchronization to help achieve concurrency, and we again define operations in terms of a time step. The duration of the time step is determined by the longest set of operations that we wish to perform in one time step, and will, as in the amortized version, be some constant amount of time.

The predecessor search operations proceed as in the sequential version. During each time step, the search determines the proper sub–structure to recursively search in, and makes exactly one recursive call. When the search procedure begins to return from the recursive calls, it returns from exactly one call per time step. Note that it is only when the search returns that the predecessor at that sub–structure is known. A predecessor search can take $\log\log N$ time steps to recurse to the bottom of the structure and another $\log\log N$ time steps to return to the top level. We further synchronize the operations by requiring all operations to take $2\log\log N$ time steps as follows. We say that a sub–queue is at depth $d$ if it can be reached after $d$ recursive steps, i.e. the top level queue is at depth 0 and its $T_{top}$ and $T_{bot}$ structures are at depth 1, and so on. Operations that complete their recursion at a depth $d < \log\log N$ wait $2(\log\log N - d)$ time steps to begin returning from their recursive calls so that operations that began before them can be sure to complete before them. Thus all operations take exactly $2\log\log N$ time steps.

The insertion and deletion procedures are synchronized in much the same manner. The insertion cannot first find its predecessor and then insert the item, because other insertion operations might cause the predecessor to change before the insertion takes place. The predecessor is, therefore, found while inserting the element from the top–down. Except for the list $L$, all other data structures at the current level can be updated in constant time without knowing the predecessor. This means that at each time step, an insert $x$ operation can update the size of the sub-structure, change the value $V[x]$ in the bit vector, and determine the proper sub-structure for the recursive insertion. If a one element tree is created, then this is also accomplished during the single time step. However, it remains to insert $x$ into the list $L$. This can only be accomplished once the predecessor of $x$ in the sub–structure is known.

It is possible that the predecessor search must recurse into a different sub–structure than the insertion (when $x$ is a new smallest element of its sub–structure). In this case, $x$ is the smallest element of its sub–structure and thus no further predecessors are required. The predecessor search and the insertion take place simultaneously. Because of the imposed synchronization, the insertion and the predecessor query both return at the same time step, and thus the insertion can continue with adding $x$ to $L$. In all other cases, the predecessor is computed in the same sub–structure as the insertion and thus is known when the insertion returns from its recursion.

To be able to decide whether the predecessor search is in the same sub–structure as the insertion, we must be able to determine the smallest element in a sub–structure. However it is possible that there may be elements which are waiting to be inserted into $L$, and thus we cannot simply use $L$ to determine the smallest element. Therefore we introduce an additional data structure stored at each sub–structure to help answer find min queries. We store the items waiting to be inserted into $L$ using the result of Ajtai, Fredman, and Komlós [3]. They give a solution to the bounded universe priority queue problem where insert, delete, and find min operations take constant time if the queue size is limited to $O((\log N)^c)$ for any constant $c$. Their technique involves $O(N^\epsilon)$ storage and preprocessing time to build a look–up table. They store an encoding of the set using $O(\log N)$ bits, which we call the AFK–encoding, used to index the look–up table. We keep an AFK–encoding at each sub–structure to store the items that have not yet been inserted into $L$. We insert the element $x$ into the AFK–encoding at a sub–structure when the insert $x$ operation reaches the sub–structure. To determine the smallest element in a sub–structure we check both its list $L$ and its AFK–encoding. When the insert $x$ operation returns from its recursion, we delete $x$ from the AFK–encoding and insert $x$ into $L$. This use of the AFK–encoding guarantees that a find min operation returns the proper value.

The delete operation is not an exact inverse of the insert operation. A delete $x$ operation makes one recursive call per time step, decrementing the size of the sub–structure, and unmarking the bit $V[x]$ before making the recursive call. Because the sub–structure must support find min and find max operations in constant time, the element $x$ must be removed from $L$ or the AFK–encoding, depending on where it is stored. Removing $x$ from the AFK–encoding is a simple constant time operation. The insert $x$ operation which has not yet completed can detect that $x$ has been deleted by examining the bit $V[x]$. Removing $x$ from $L$ presents the following problem. Suppose that an insert $y$ operation just returns from its recursive call and thinks that $x$ is its predecessor. This is possible because $x$ has not been removed from lower sub–structures. The insert $y$ operation will try to insert $y$ following $x$ in $L$ ($y$ becomes the successor of $x$), but $x$ has been removed.

To fix this problem, we remove $x$ from $L$, but we retain the node $x$ and its pointer to its successor. Using the successor pointer from $x$, an insert $y$ operation can find its proper location in $L$ in constant time. When the insert $y$ operation changes the successor of $x$, it updates the pointer from $x$. Note that the successor of $x$ cannot change unless $x$ is found as the predecessor. When the deletion of $x$ returns from its recursive call, the node $x$ can be removed entirely. If a delete operation deletes the largest or the smallest elements of the list $L$, then the largest or smallest element pointers must be updated appropriately.

Using the above data structures and algorithms we maintain the invariant that at a sub–queue $T_i$ for some $i$, an insertion or deletion completes in $2 \log i$ time steps. Additionally, find min or find max operations take constant time, reflecting the value of all insertions or deletions which have reached the sub–queue but possibly have not yet completed. This invariant guarantees the correctness of our algorithm and shows that a find min or find max operation for the entire queue is answered in $O(1)$ time.

An easy reduction in the space used by the data structure comes from applying the AFK-encoding technique. If we group the elements from the range $[0 \ldots N]$ into consecutive blocks containing $\Theta((\log N)^c)$ elements for some fixed constant $c$, then each block can be represented by an AFK–encoding. This allows us to reduce the range for the priority queue to $[0 \ldots N/(\log N)^c]$ and thus the total space used is $O(n + N \log\log N/(\log N)^c)$ where $n$ is the number of elements in the queue.

## 3.6 The top–down split procedure

This section presents an algorithm for performing top–down splits of a 2–4 tree. Techniques required to perform top–down joins are presented as well. This material is of a technical nature and presented because these top–down operations are needed by the concurrent amortized version of the priority queue. The reader may therefore elect to skip this section.

Given a split value $s$, we create two trees, the left tree containing values $< s$ and the right tree containing values $\geq s$. We assume that some child of the root of the 2–4 tree belongs in the left tree so that we may view the operation as splitting subtrees from the right tree and joining subtrees to the left tree otherwise we reverse the roles of the right and the left trees. This paragraph shows how to maintain the 2–4 tree structure of the right tree as subtrees are split away, and the following paragraph shows how to join the split away subtrees to the left tree. Call $x$ the node of the original 2–4 tree that we are currently splitting. During the split procedure we maintain the following invariant: the parent of $x$ is not a 2 node. We show how this invariant can be maintained and how it allows us to restructure the right tree after some children of $x$ are split away, and then we show how to initialize the invariant. Since the parent of $x$, $p(x)$, has at least 3 children, $p(x)$ must have at least that a find min or find max operation for the entire queue is answered in $O(1)$ time.

An easy reduction in the space used by the data structure comes from applying the AFK-encoding technique. If we group the elements from the range $[0 \ldots N]$ into consecutive blocks containing $\Theta((\log N)^c)$ elements for some fixed constant $c$, then each block can be represented by an AFK–encoding. This allows us to reduce the range for the priority queue to $[0 \ldots N/(\log N)^c]$ and thus the total space used is $O(n + N \log\log N/(\log N)^c)$ where $n$ is the number of elements in the queue.

## 3.6 The top–down split procedure

This section presents an algorithm for performing top–down splits of a 2–4 tree. Techniques required to perform top–down joins are presented as well. This material is of a technical nature and presented because these top–down operations are needed by the concurrent amortized version of the priority queue. The reader may therefore elect to skip this section.

Given a split value $s$, we create two trees, the left tree containing values $< s$ and the right tree containing values $\geq s$. We assume that some child of the root of the 2–4 tree belongs in the left tree so that we may view the operation as splitting subtrees from the right tree and joining subtrees to the left tree otherwise we reverse the roles of the right and the left trees. This paragraph shows how to maintain the 2–4 tree structure of the right tree as subtrees are split away, and the following paragraph shows how to join the split away subtrees to the left tree. Call $x$ the node of the original 2–4 tree that we are currently splitting. During the split procedure we maintain the following invariant: the parent of $x$ is not a 2 node. We show how this invariant can be maintained and how it allows us to restructure the right tree after some children of $x$ are split away, and then we show how to initialize the invariant. Since the parent of $x$, $p(x)$, has at least 3 children, $p(x)$ must have at least

5 grandchildren after $x$ is split. We can therefore restructure $p(x)$ and its children so that $x$ contains at least 3 nodes after $x$ is split. Note that $p(x)$ may become a 2 node in this restructuring , but this is no longer important since we are now splitting the left child of $x$. This maintains the invariant, but we have not shown how to initialize it. We can assume that once the root of the tree is split, 2 children remain since if more than 2 remain, then the invariant is initialized, and if only one remains then the root may be deleted and the child considered as the root. Let $x$ be the child of the root that is being split. Once $x$ is split, we have two cases to consider. Suppose that the root has only 3 or 4 grandchildren once $x$ is split. We can delete the root and merge the remaining part of $x$ with its sibling, which becomes a 3 or a 4 node. If the root has 5 or more grandchildren after the split, then they can be restructured so that $x$ becomes at least a 3 node. This initializes the invariant.

In the left tree, we join the split subtrees from the top down as described below. This takes place concurrently with the splitting, and because the height of the subtrees decreases as we proceed down the right tree, the splitting and joining can proceed down their respective trees in lockstep. Let $y$ be the rightmost node in the left tree that is at the same height as the node $x$ in the right tree ($x$ is the node we are splitting in the right tree). The pieces split from $x$ are added to the node $y$. This increases the degree of $y$ by up to 3, and $y$ may now have to be split. We therefore maintain the invariant that $p(y)$ is not a 4 node. Since the degree of $y$ is at most 7 after the new children are added, we can split $y$ such that the rightmost child of $p(y)$ has degree at most 3, thus maintaining the invariant. This invariant is easily initialized by splitting an initial 4 node if required. Thus we have shown how to perform top–down splits and joins in 2–4 trees. This allows top–down concurrency while performing these balanced tree operations.

## 3.7 An application

The data structures presented in this chapter can be used to solve the longest increasing (or decreasing) subsequence problem. Given a sequence of $n$ integers, $x_1, x_2 \ldots x_n$, each from the range $[0 \ldots N]$, the problem is to find the longest increasing sequence of indices $j_1, j_2 \ldots j_k$ such that $x_{j_1} < x_{j_2} < x_{j_3} \ldots x_{j_k}$. One solution to this problem involves maintaining an ordered list of elements such that the $i^{th}$ element of the list is the smallest element that can end a list of length $i$. The list is initially empty and the elements of the sequence are inserted one at a time. To insert an element $e$, we find the predecessor of $e$ in the list and insert $e$ after its predecessor. If $e$ has a successor in the list then the successor is deleted, otherwise $e$ is the last element of a new longest sequence. If we store the predecessor of $e$ at $e$ when $e$ is inserted into the list, then the sequence can be reconstructed once the insertions have completed by walking backward starting at the last element in the final list.

Clearly if the elements come from the range $[0 \ldots N]$, then the van Emde Boas data structure can be used to yield an algorithm with running time $O(n \log\log N)$. Using either the worst case or the amortized concurrent versions of the van Emde Boas structure we can get a parallel algorithm that runs in $O(n)$ time using $\Theta(\log\log N)$ processors as follows. The insertions into the list are pipelined using the concurrent data structure. At each of $n$ time steps, a new insertion begins, and after $O(n)$ time steps the procedure is finished. The only difficulty lies in the deletion of successor elements. The sequential algorithm assumes that these deletions take $O(1)$ time. In our case, these elements must be removed from the data structure as well. The solution to this problem is similar to the deletion of a lower tree in the amortized version. If a list element $y$ is to be deleted, then we remove $y$ from the list; we retain the node $y$ and its predecessor pointer, however. The insertion operation that causes the deletion of $y$ removes $y$ from the predecessor data structure as a normal deletion, but since we keep the node $y$ and its predecessor pointer, the operation also updates $y$'s predecessor pointer at each time step. In this way, some other insertion operation which returns $y$ as its predecessor can reach its correct predecessor using $y$'s pointer. This completes the description of the parallel longest increasing subsequence algorithm.

## 3.8 Remarks and open problems

This chapter has presented two schemes for performing concurrent operations in a bounded universe priority queue. Each priority queue operation requires $O(\log\log N)$ time and a new operation can begin every $O(1)$ time steps. The amortized version is conceptually simpler and is easier to implement, since the worst–case version uses the AFK–encoding [3] to maintain the largest and smallest elements for the sub–queues. The full power of the AFK result is not needed, however, since only $O(\log\log N)$ elements ever need to be stored, so an interesting question is whether the dependence on their data structure can be eliminated.

The synchronization of the presented algorithms is of a global nature, because operations must synchronize based on a clock, but less strict synchronization might be possible. In the worst–case algorithm, an insertion must sometimes wait on the result of a predecessor query in a different sub-structure. Can the imposed synchronization be relaxed while maintaining robustness and correctness?

The space required for our algorithms is $O(N/(\log N)^c)$. This bound could prove prohibitively expensive in an implementation of these algorithms. Willard [64] has shown how to reduce the space for the van Emde Boas data structure, but his approach is much like the amortized version presented earlier. He saves space by only storing the marked nodes, using a hashing scheme for each level of the fixed tree, for instance. This causes his running times for insertions and deletions to be expected, because the marking or unmarking of a node may require some randomized steps to update the hashing scheme. This will not work in the algorithms presented here. Because operation proceed down the tree in lockstep, the lack of a worst case bound means that many operations could be forced to wait. This could destroy the order of completion of the operations and therefore their correctness.

It is not known how to implement the van Emde Boas data structure using both linear space and having worst case $O(\log \log N)$ running times. We leave this as an interesting research question. This thesis contains results from two distinct areas of research in algorithms. Chapters 2 and 3 present sequential and parallel random access machine (RAM) algorithms. We analyze the running times of these algorithms, and for each of the problems considered, we give the first algorithms whose running time is provably optimal. Chapter 4 presents applied work in the area of parallel computing. We describe algorithms developed for factoring integers using a real world single instruction multiple data (SIMD) parallel computer. These algorithms and their implementations are designed to be as fast as possible in practice. Because of the difference in the design goals of the algorithms given in the thesis, the presentation varies somewhat between chapters, and each chapter is self contained.

In Chapter 2 we consider the minimum spanning tree verification and sensitivity analysis problems. To verify a minimum spanning tree we must determine whether a given spanning tree of a weighted graph has minimum cost. The sensitivity analysis problem is to determine for every individual edge of the graph, the maximum change in cost for that edge that preserves the minimality of a given spanning tree. In this chapter, we present both sequential and parallel algorithms for these problems. The verification result is intriguing due to the fact that minimum spanning trees can currently be verified faster than they can be found. This lends some hope to the idea that spanning trees can be found faster.

The sequential verification algorithm runs in linear time, and the sequential sensitivity analysis algorithm runs in linear expected time and optimal deterministic time. These results are joint work with Monika Rauch and Bob Tarjan and appear in [18]. The parallel verification algorithm runs in $O(\log n)$ time using $\Theta((n+m)/\log n)$ processors, where $n$ is the number of vertices and $m$ is the number of edges in the given graph. The parallel sensitivity analysis algorithm has time bounds similar to those of the sequential version, $O(\log n)$ expected time and optimal deterministic time, using $\Theta((n+m)/\log n)$ processors. The parallel results share top–level algorithmic ideas with the sequential algorithms, but the actual data structures and implementations are quite different. The parallel algorithms are joint work with Bob Tarjan.

In Chapter 3 we present a pipelined data structure for maintaining a priority queue. A priority queue is a data structure that supports insertion, deletion, and find minimum operations on elements from some ordered universe. If the range of the elements can be bounded by some range, say $[0 \ldots N]$, then the running time of the priority queue operations is $O(\log\log N)$ using the data structure due to van Emde Boas [20]. The algorithms in this chapter support priority queue operations on elements from a bounded universe in $O(\log\log N)$ time as well, but they additionally allow the priority queue operations to be pipelined in such a way that a new operation can begin every $O(1)$ time steps. In this way multiple processors can operate concurrently on the queue. A version using simple, easy to implement data structures gives an amortized running time of $O(\log\log N)$ per operation, and a version with worst case $O(\log\log N)$ time per operation is presented as well.

The focus shifts in Chapter 4 to the efficient implementation of massively parallel algorithms for integer factoring. The algorithms presented are for single instruction multiple data (SIMD) parallel computers, and all of the algorithms have been implemented on the MasPar SIMD machine. We present both special purpose factoring algorithms (where the running time depends strongly on the size of the smallest factor) and general purpose methods (where the running time depends solely on the size of the integer to be factored). Additionally we present fast algorithms for the multiplication of large integers modulo a fixed integer. This multiplication algorithm is useful for many applications in the area of computational number theory and cryptography. The results of this chapter are joint work with Arjen Lenstra and appear in [15, 16, 17].

# Chapter 4

# Massively parallel factoring

## 4.1 Introduction

In this chapter we study integer factoring algorithms and their implementations on massively parallel single instruction multiple data (SIMD) machines. We present SIMD implementations of two factoring algorithms, namely the elliptic curve method (ECM) [42] and the multiple polynomial quadratic sieve (QS) [50]. One piece of our ECM implementation, a fast SIMD modular multiplication algorithm, is presented in a general framework in its own section since it has applications outside of factoring.

Factoring some given number $n$ is not a difficult task given $O(\sqrt{n})$ time. The difficulty is, of course, that an algorithm for factoring $n$ is given only the $O(\log n)$ bits representing $n$ as input, thus $O(\sqrt{n})$ time is exponential in the input size. It is not currently known whether a polynomial time (in $\log n$) algorithm exists for factoring integers, and the problem is also not known to be NP–complete [1]. The algorithms that we present are some of the most efficient algorithms currently being used, and fall into two basic categories as follows.

Usually one distinguishes between two types of factoring algorithms, the *general purpose algorithms* whose expected run time depends solely on the size of the number

$n$ being factored, and the *special purpose algorithms* whose expected run time additionally depends on properties of the (unknown) factors of $n$. Examples of special purpose algorithms are *trial division, Pollard's rho method, Pollard's $p-1$ method,* and the *elliptic curve method.* To find a factor $p$, trial division needs time approximately linear in $p$, Pollard's rho method is approximately linear in $\sqrt{p}$, and Pollard's $p-1$ method is approximately linear in the largest prime factor of $p-1$. Thus, in the worst case all these methods take exponential time in the number of bits in the input. The running time of the elliptic curve method, while no formal proof is known, is thought to be $O((\log n)^2 L_p[\sqrt{2}])$ expected time to find $p$, where

$$L_x[a] = \exp((a + o(1))\sqrt{\log x \log\log x}),$$

for $x \to \infty$. It follows that in the worst case $p \approx \sqrt{n}$ the method takes expected time $L_n[1]$, which is subexponential in $O(\log n)$. Because their run time depends so strongly on the size of smallest factor of $n$, and only polynomially on the size of $n$ itself, the performance of special purpose methods is usually measured by the size of the prime discovered.

The above exponential time methods do not present a threat to the security of a factoring based cryptosystem, except that one sometimes takes the precaution to construct $n$ such that Pollard's $p-1$ method will not be unexpectedly lucky. Such precautions cannot be taken against the elliptic curve method. Whereas in Pollard's $p-1$ method $p$ will be discovered if $p-1$ has only small factors, the elliptic curve method will find $p$ if the method is so lucky to hit upon a number *close to $p$* that has only small factors. Because the method consists of many independent trials, there is always the possibility that $p$ will be discovered quickly.

Because a cryptosystem can choose $n$ to be the product of two roughly equal size primes, the success of the general purpose methods is of more interest to the designer of a crypto system. The general purpose method which we have implemented is the multiple polynomial version of the quadratic sieve algorithm. Another example of a general purpose method would be the number field sieve [7]. The running time of the quadratic sieve algorithm is, again without rigorous proof, thought to be $L_n[1]$ expected time, which is the same as the worst case of the elliptic curve algorithm. The quadratic sieve, however, is much better in practice for an $n$ that is the product of two large primes. For instance, the largest prime factor found by the elliptic curve method has 42 digits, which corresponds to an $n$ with 84 digits in the worst case. This would be a relatively easy chore for our quadratic sieve implementation, which can factor a 100 digit $n$ in a matter of hours, and has been able to factor a 110 digit integer.

In the next section we discuss the architecture of the machine used for our implementations. Section 4.3 discusses the elliptic curve implementation, and Section 4.4 presents a fast algorithm for the modular multiplication. This algorithm was developed with the elliptic curve implementation, but is presented separately since the multiplication algorithm is of independent interest. Finally, Section 4.5 presents the massively parallel quadratic sieve algorithm.

## 4.2 The hardware

This section contains a short overview of the 16K MasPar, the massively parallel computer that we have used for the implementations to be described in this chapter. Our description is incomplete, and only covers those aspects of the machine that are referred to in the following sections. For a complete description of the MasPar we refer to the manuals, like [45].

The 16K MasPar is a SIMD machine, consisting of, roughly, a *front end*, an *array control unit* (ACU), and a $128 \times 128$ array of *processing elements* (PE array). Masks, or conditional statements, can be used to select and change a subset of active processors in the PE array, the so-called *active set*. The fact that it is a SIMD machine means that instructions are carried out sequentially, and that instructions involving parallel data are executed simultaneously by all processors in the active set, while the other processors in the PE array are idle. The instructions involving singular (i.e., non-parallel) data are executed either on the front end or on the ACU; for the purposes of our descriptions the front end and the ACU play the same role.

According to our rough measurements, each PE can carry out approximately $2 \cdot 10^5$ additions on 32 bit integers per second, and can be regarded as a 0.2 MIPS processor. Furthermore, each PE has 64KBytes of memory, which implies that the entire PE array has 1GByte of memory. Each processor can communicate with its north, northeast, east, southeast, south, southwest, west, and northwest neighbor, with toroidal wraparound. Actually, a processor can send data to a processor at any distance in one of these eight directions, with the possibility that all processors that lie in between also get a copy of the transmitted data. There is also a global router that allows any processor to communicate with any other processor, but we never needed it.

Each job has a size between 4K and 64K, reflecting the amount of PE-memory it uses. Only those jobs which together occupy at most 64K are scheduled in a round robin fashion, giving each job 10 seconds before it is preempted, while the others must wait. This means that jobs are never swapped out of PE-memory.

For our implementations we used the MasPar Parallel Application Language MPL, which is from our perspective, a simple extension of C.

## 4.3 SIMD elliptic curve factoring

In this section we consider two implementations of the elliptic curve method on the SIMD machine described above. As far as we know SIMD elliptic curve implementations have not been considered before, although massive parallelism is not new to this area. In [40], for instance, a distributed network implementation of the elliptic curve method is described, which may be viewed as a large scale *multiple instruction, multiple data* (MIMD) approach. We will see that SIMD machines, even though they are relatively inexpensive compared to MIMD machines, nevertheless can achieve an impressive elliptic curve performance.

Using one of our implementations we have been able, for the first time for the elliptic curve method, to find a 40 digit factor[1]. The previous elliptic curve record was 38 digits, which occurred three times as far as we know. On the negative side, the elliptic curve method has also missed many smaller factors, in the 35 digit range, even after serious efforts. Using the run time estimates from Section 1, one finds that finding a 60 digit factor can be expected to be more than 3000 times more difficult than finding a 40 digit factor. Given how much computing time has been invested to this date in the elliptic curve method, it seems safe to say that it is unlikely that with present day technology we will ever be able to discover factors of 60 or more digits using the elliptic curve method. Estimates of this type are, for instance, useful for the design of cryptosystems based on discrete logarithms modulo a composite modulus, like [46].

The remainder of this section is organized as follows. Section 4.3.1 contains a description of the elliptic curve method and our first SIMD implementation. Section 4.3.2 gives a rough outline of the block-wise SIMD multi-precision modular integer arithmetic, which is presented in detail in Section 4.4. The second elliptic curve

[1] In the mean time Dave Rusin from the University of Northern Illinois has found a 42 digit factor, using Peter Montgomery's elliptic curve program.

implementation, which is based on this block-wise arithmetic, is presented in Section 4.3.3, along with some results concerning random generators with very long period, that were constructed using this second elliptic curve implementation.

### 4.3.1 The elliptic curve method

In this section we briefly describe the elliptic curve method and our first SIMD implementation of the method. For a detailed description of the elliptic curve method, hints for its implementation and parameter choices, we refer to [38, 42, 49, 59]. For our purposes it suffices to know that the elliptic curve method consists of a number of independent trials. For each trial an elliptic curve $E$ modulo $n$ and a point $x$ in a group $G$ related to $E$ are randomly selected. The group operation in $G$, which we will write multiplicatively, consists of several additions, subtractions, multiplications, and inversions of integers modulo $n$, and can be carried out in time $O((\log n)^2)$ per operation. Clearly, the group operation breaks down if some integer $y$ for which $\gcd(n, y) \neq 1$ has to be inverted modulo $n$ in the course of a group operation; in this case a factor of $n$ has been found.

This is exploited as follows. Using the group operation, the point $x$ is raised to a huge power $k$ consisting of the product of all prime powers below a certain bound $B_1$. The trial is lucky if this computation cannot be completed because the group operation breaks down, since in this case a factor of $n$ has been found. The trial fails if $x^k \in G$ has been computed successfully. If $p$ is $n$'s smallest prime divisor and $B_1 = L_p[\sqrt{1/2}]$, then each trial has probability $L_p[-\sqrt{1/2}]$ to factor $n$, as explained in [38, 42]. This implies that the number of independent trials needed to factor $n$ can be expected to be $L_p[\sqrt{1/2}]$. One trial takes time $O((\log n)^2 B_1)$, from which the total expected run time $O((\log n)^2 L_p[\sqrt{2}])$ follows.

The computation of $x^k \in G$ is usually referred to as the *first phase* of the algorithm. In the *second phase* $x^{kq} \in G$ is computed for all primes $q$ in $[B_1, B_2]$ for some bound

$B_2$. This requires approximately an additional $\pi(B_2)-\pi(B_1)$ group operations, where $\pi(b)$ denotes the number of primes $\leq b$. It appears to be close to optimal to select $B_1$ and $B_2$ in such a way that the two phases take approximately the same amount of time, which makes $B_2$ an order of magnitude larger than $B_1$.

The optimal parameter choice for the elliptic curve method depends on the unknown factor $p$ of $n$, and therefore cannot be known beforehand. A common approach is to do a few trials with fairly low $B_1$ (and corresponding $B_2$), upon failure a few trials with slightly larger $B_1$, and so on, until either the number is factored or the factoring attempt is aborted. The expected amount of work to find a certain factor usually varies only slightly: the success probability for $t$ trials with bound $B_1$ is not dramatically different from the success probability for $c \cdot t$ trials with bound $B_1/c$, but only for $c$ in a limited range, say $1/2 < c < 2$. The optimal parameter choices depend on the implementation as well. For instance, for the implementation described in [40], the optimal choices are $t = 300$ and $B_1 = 65000$ to find 25 digit factors with a 60% success probability, $t = 950$, $B_1 = 275000$ for 30 digits, and $t = 2300$, $B_1 = 1100000$ for 35 digits.

The trials of the elliptic curve method are independent, but any number of them can be carried out simultaneously. Since the sequence of operations involved in the computation of $x^k \in G$ depends only on the value of $k$, and *not* on the actual data to which the operations are applied, $t$ elliptic curve trials can be carried out in parallel on a $t$ processor SIMD machine. An exception occurs if one of the trials factors $n$, but in that case the process can be terminated. Using this elementary approach we ran 16K trials in parallel on the 16K MasPar, with each of the 16K PE's working on its own elliptic curve, uniquely generated using a single random seed. For this purpose the multi precision integer arithmetic used in [40] was adapted to the MasPar, in such a way that each PE operates independently on its own extended precision integers, simultaneous with the other PE's (in the active set). Thus we can carry out

16K elementary operations ($+$, $-$, $*$, quotient and remainder) on extended precision integers in parallel, without interprocessor communication.

Because inversions modulo $n$ are slow, particularly so on a SIMD machine where the cost is determined by the processor that needs the most iterations, we kept track of the numerators and denominators (modulo $n$) of the group elements, without performing the inversions. But since these inversions are supposed to lead to the factorization of $n$, they cannot be avoided entirely. At regular intervals we therefore computed the product modulo $n$ of all denominators (using 14 (i.e., $\log_2(16K)$) multiplications modulo $n$ on the PE array), and computed the greatest common divisor $g$ of the resulting product and $n$ on the (much faster) front end. If $g$ turns out to be $> 1$, the PE's can be inspected to see how many of them found a factor, or to refine the factorization if $g$ is not prime. The latter usually happened only in the presence of several very small factors (up to 10 digits); larger factors are usually found on only one PE.

Addition and subtraction modulo $n$ are easily made efficient in SIMD mode, but multiplication modulo $n$ is more problematic. This is caused by the remainder computation modulo $n$, where the instruction stream is more dependent on the values involved. We used the so-called *Montgomery representation* throughout our program, because it allows a modular multiplication that is oblivious of the data involved, without affecting the addition or the subtraction; see Section 4.4 or [48].

Although these and various other improvements considerably enhanced the performance of our first implementation, there was not much we could do about the fact that a 0.2 MIPS PE is a fairly slow processor: it took about a day to complete 16K curves with $B_1 = 50000$ and $B_2 = 10^6$ for a 100 digit $n$. With these parameters one can almost guarantee that all factors up to 28 digits will be found, but this is not an optimal parameter choice. To find 28 digit factors far fewer curves with much larger $B_1$ would be better, whereas the optimal $B_1$ that corresponds to 16K trials is more than $10^7$. The latter would be a good choice if one wants to look for 45 digit factors, but it would require an inordinate amount of time on this machine.

Consequently, this parallel elliptic curve program is not ideally suited for the present MasPar. For future generations of SIMD machines, however, our program might turn out to be useful: if future PE's run at speeds comparable to that of current workstations, then 16K parallel trials with matching bounds could be processed in at most a few days.

Given the current PE speed, the only way to get a better parallel elliptic curve program on the 16K MasPar seems to be to divide the work per curve over $r$ PE's, for some $r$. This would decrease the number of trials by $r$ and hopefully increase the speed per curve by the same factor, thus allowing parameter choices which are closer to optimal for factors in the 30 digit range. We achieved this by designing a multi precision integer arithmetic that is entirely different from the one mentioned above. This arithmetic will be described in the next section.

### 4.3.2 Block-wise modular arithmetic

In this section we describe an alternative integer arithmetic for SIMD machines, that essentially reconfigures the machine into a machine with fewer but faster processors, at least for arithmetic modulo a fixed integer $n$.

Let $b$ be some small integer such that arithmetic operations on $b$ bit integers can be carried out efficiently on a PE. Because multiplication of 32 bit integers with a 64 bit result is the most efficient multiplication that a PE can perform, we used $b = 30$, so that we could also add without overflow problems. Let $r$ be the smallest integer with $2^{b \cdot r} > n$, where $n$ is the odd number being factored. Each of the 128 rows of 128 PE's is divided into $u = [128/(r+1)]$ disjoint *blocks* of $r+1$ consecutive PE's, and $128 - (r+1) \cdot u$ idle PE's. Thus, there are $128 * u$ blocks. The active set consists of the PE's that are contained in a block.

Suppose that the consecutive PE's in a block are numbered from 0 to $r$. If the $i$th PE contains a $b$ bit integer $v_i \geq 0$, then $v_0, v_1, \ldots, v_r$ together represent the number $v = \sum_{i=0}^{r} v_i 2^{b \cdot i}$. Since we use the integers in $\{0, 1, \ldots, n-1\}$ to represent residue classes modulo $n$, we usually have that $v < n$ and therefore $v_r = 0$. This extra PE is used in the multiplication modulo $n$. Let $v = \sum_{i=0}^{r} v_i 2^{b \cdot i}$ and $w = \sum_{i=0}^{r} w_i 2^{b \cdot i}$ be two integers modulo $n$. To compute the sum $s = v + w$ the $r+1$ PE's compute $\bar{s}_i = v_i + w_i$ and $c_i = \bar{s}_i / 2^b$, next $c_i$ is sent by the $i$th PE to the neighboring $(i+1)$th PE, and finally $s_i$ is computed as $\bar{s}_i - c_i \cdot 2^b + c_{i-1}$. In the unlikely event that one of the $s_i$ is still $\geq 2^b$ the carry propagation is repeated until all $s_i$ are $< 2^b$. Here we note that it requires only one fast instruction on the MasPar to check if there are still any carries to be propagated. To complete the addition modulo $n$, we simply subtract $n$ from $s$, using a similar technique; if $s - n$ is negative then $s$ is the final outcome, otherwise it is $s - n$.

Examples that require $r$ carry propagation steps are easy to construct, and a depth $O(\log r)$ carry propagation tree would give a better worst case performance. Our simplistic approach, however, works on average much faster because the second carry propagation step hardly ever occurs.

Multiplication modulo $n$ within blocks is more complicated. As in the first elliptic curve implementation we used the Montgomery representation to avoid divisions. Let $R = 2^{b \cdot r}$, the smallest power of $2^b$ larger than $n$. The Montgomery representation $\tilde{x}$ of an integer $x$ modulo $n$ is the integer $x \cdot R \bmod n \in \{0, 1, \ldots, n-1\}$. Addition and subtraction of numbers in Montgomery representation is not different from ordinary addition and subtraction modulo $n$, and is carried out as described above. Multiplication, however, becomes much simpler than ordinary multiplication modulo $n$. Let $z$ be such that $z \equiv x \cdot y \bmod n$. Then $\tilde{z}$ equals $\tilde{x} \cdot \tilde{y} / R \bmod n$. This $\tilde{z}$ can be computed efficiently as follows. First compute $v = \tilde{x} \cdot \tilde{y}$. Let $v = \sum_{i=0}^{2r} v_i 2^{b \cdot i}$ and $d$ be such that $d \cdot n \equiv -1 \bmod 2^b$, which is well-defined because $n$ is odd. Next, for $i = 0, 1, \ldots, r-1$

in succession replace $v$ by $v + 2^{b \cdot i} \cdot n \cdot (v_i \cdot d \bmod 2^b)$, where the $v_i$ for $i > 0$ are the radix $2^b$ digits of the $v$ that was computed in the previous iteration. Notice that after iteration $j$ the new $v_j$ is zero, and that the new $v$ is congruent to the old $v$ modulo $n$. Consequently, the resulting $v_0$ through $v_{r-1}$ are all zero, and the division by $R$ can be carried out by simply shifting the resulting $v$ to the right. The result might be $\geq n$, in which case it suffices to subtract $n$ once to make it $< n$. Montgomery multiplication can be carried out using $2r^2 + r$ multiplications on $b$-bit integers.

By merging the iterations for the (ordinary) multiplication of $\tilde{x}$ and $\tilde{y}$, and the division modulo $n$ by $R$, the multiplication of numbers in Montgomery representation can be done quite easily in a block of PE's. Straightforward application of the above algorithm leads to a block-wise modular multiplication that can be made to fit in blocks of only $r$ consecutive PE's, with 3 multiplications per iteration: two on all PE's in the block (on different data), but one (the computation of $v_i \cdot d \bmod 2^b$) that operates on identical data for all PE's in the block or that could be carried out by only one PE and sent to the others. This is inefficient, because it requires time for $3r$ multiplications on $r$ PE's in parallel, instead of the $2r+1 = (2r^2+1)/r$ multiplications we hoped for. As shown in the next section the $3r$ can be improved to $2r + 2$, giving a ratio which is close to optimal. Using this method we got an acceptable speed for the modular multiplication: for a 95 digit $n$ one modular multiplication takes about 0.003 seconds, and because $r = 11$ implies $[128/(11 + 1)] = 10$ blocks per row, 1280 of these multiplications can be carried out simultaneously.

An additional advantage of the block-wise arithmetic is that all relevant values can be kept in registers, so that costly memory fetches and stores can be avoided. A detailed description can be found in the next section.

### 4.3.3 A second implementation and results

Incorporation of the modular arithmetic from the previous section into our first SIMD elliptic curve implementation led to a second version of our program that runs one curve per block instead of one curve per processing element. This implies that the number of trials depends on the size of $n$. For a 95 digit $n$ we get 1280 trials, for 80 digits $r$ becomes 9, and the number of trials goes up to 1536.

We used the following elementary method to lower the number of group operations needed to compute the $k$th power of the point $x$ on the curve. According to the definition of $k$ given in Section 4.3.1, we have $k = \prod_{i=1}^{l} q_i$, where $\{q_1, q_2, \ldots, q_l\}$ is the set of prime powers $< B_1$. The usual way of computing $x^k$ is to first raise $x$ to the power $q_1$, next raise the result to the power $q_2$, etc., until all $q_i$ have been processed. Let for some integer $m$ the *weight* $w(m)$ be defined as the number of ones in the binary representation of $m$. If ordinary repeated squaring and multiplication is used for the exponentiation, then the cost of the computation of $x^k \in G$ is $\sum_{i=1}^{l} [\log_2 q_i]$ squarings in $G$ and $\sum_{i=1}^{l} (w(q_i) - 1)$ multiplications in $G$.

Let $S = S_1 \cup S_2 \cup \ldots S_s$ be a partition of $\{1, 2, \ldots, l\}$, and let $\bar{q}_j = \prod_{i \in S_j} q_i$. Clearly, $x^k$ can also be computed by first raising $x$ to the power $\bar{q}_1$, next the result to the power $\bar{q}_2$, etc., up to $\bar{q}_s$. The number of squarings in $G$ needed for this computation is approximately the same as the number of squarings given above. The number of multiplications in $G$, however, can be made substantially smaller by choosing a partition $S$ for which $\sum_{j=1}^{r} w(\bar{q}_j)$ is small. Finding the best partition with respect to this metric is in general a hard problem. In practice we will have to do with what can be found in a reasonable amount of time. Using a simple greedy algorithm (and $B_1 = 100000$) we found a partition in subsets of cardinality at most two that had approximately half the original weight; this solution was used to derive an optimal partition under the same restrictions, but the resulting weight was not significantly lower. Next, we considered subsets of cardinality at most three. This resulted in approximately a third of the original weight. Given how much time it took us to find these triples, this is probably the best we may hope to achieve. The triples were found by means of a greedy-type algorithm on the 16K MasPar. To make the run times acceptable we processed the primes in intervals. More precisely, for each $i \in \{1, 2, \ldots, 20\}$ we determined partitions into triples of the prime powers $< 2 \cdot 10^6$ for the primes in the interval $[(i-1) \cdot 10^5, i \cdot 10^5]$. This separation into intervals is also useful because it allows a choice of bounds. The total run time of the program was reduced by 18% using this technique. To give an example, the primes 1028107, 1030639, and 1097101 have weights 10, 16, and 11, but their product has weight 8; in binary

$$11111011000000001011 \cdot 11111011100111101111 \cdot 100001011110110001101 =$$

$$10000001000100000010000001000000000000001000000100000000000000001.$$

This example is remarkable but by no means exceptional.

For $n$ with $r = 11$ it takes about 34 hours to complete 1280 elliptic curve trials with $B_1 = 10^6$ and $B_2 = 2 \cdot 10^7$. For $n$ with $r = 12$ it takes approximately $34 * 12/11$ hours to complete 1152 trials with the same bounds, and other timings can be derived similarly. Consequently, this second elliptic curve program allows a much more balanced choice of the parameters for a search of factors in the 30 digit range. We have used the program to factor various numbers from the list of composite numbers from [9] and many numbers from the 'Partition List' of the 'RSA Data Security Factoring Challenge.' To date the largest factor we have found has 40 digits, which was a new elliptic curve factoring record:

$$\mathrm{p}(11279) = 2^6 \cdot 5 \cdot 8418\,735626\,949973\,617503 \cdot$$

$$1232\,079689\,567662\,686148\,201863\,995544\,247703 \cdot$$

$$78\,507734\,924917\,342278\,622201\,969372\,653526\,213641\,483293.$$

The number factored was the 89 digit product of the last two factors, and p(11279) denotes the 11279th partition number. The factorization was found by one of 1408 trials with $B_1 = 10^6$ after $q = 1208269$ in the second phase, which means that we have been quite lucky since finding the factor at this point happens with probability 0.7% [2].

We found another, unexpected, application of our SIMD elliptic curve program in [44]. In this paper a new class of pseudo random generators is introduced with the remarkable property that they have provably long periods. Such generators are based on integers $b$, $r$, and $s$ such that $m = b^r \pm b^s \pm 1$ is prime, and have a period equal to the order of $b$ modulo $m$. The size of $b$ depends on the type of pseudo random number generator that is needed; typically it varies between 16 and 64 bits. Consequently, primes $m$ as above can fairly easily be found, if $r$ and $s$ are not excessively large (typically they vary between 10 and 50). Establishing the order of $b$, however, requires factoring $m-1$, which might be (and usually is) hard. We found, among others, that $b = 2^{53} - 1052$ has order $(m-1)/2$ modulo the 511 digit prime $m = b^{32} + b^{16} - 1$, where the factorization of the factor $b^8+1$ of $m-1$ was found using our SIMD elliptic curve implementation:

$$b^8 + 1 = 257 \cdot 16673 \cdot 275422\,758002\,613571\,762817\cdot$$

$$29\,464604\,796724\,055573\,394001 \cdot 65726\,748717\,597552\,856331\,429857\cdot$$

$$18\,955214\,139747\,587789\,390113\,133069\,907813\,629489.$$

This was one of the few examples where the global inversion on the front end produced a composite factor: one block had found the 24 digit factor and simultaneously another block had found the 26 digit factor. This $b$ and $m$ provide an attractive

[2]This corresponds to a 0.0006% probability of success per curve. For comparison, the 42 digit factor referred to above, was found with $B_1 = 2 \cdot 10^6$, and $B_2 \approx 10^8$, which leads to a 0.003% probability of success per curve. The number of curves used in that factorization is unknown to us.

way to generate double-precision floating point pseudo random numbers using just addition.

## 4.4 Fast modular multiplication

In the previous section, we have seen one of many examples where fast modular multiplication is crucial for an efficient implementation of a number theoretic algorithm or cryptographic scheme. The elliptic curve integer factoring algorithm of the previous section spends most of its time multiplying numbers mod $n$, where $n$ is the number we wish to factor. In this section we describe how modular multiplication can be carried out efficiently and in parallel on large SIMD arrays of processors. Using the algorithm presented here, a modular multiplication using $k$ processors requires time about $1/k$ times the amount of time to perform a modular multiplication using one processor, which is the optimal speed-up one can achieve using $k$ processors. This should be compared to the straightforward SIMD implementation mentioned in the previous section, which requires $1.5/k$ times the time for the one processor version, due to an inherent unparallelizable inefficiency in the algorithm. We have been able to overcome this efficiency, thus gaining a speed-up of a factor 1.5.

In addition to applications to cryptanalysis, our method can be used for a 'central facility' that has to sign or verify public key certificates, or decrypt RSA messages, for many users simultaneously. There too, as in the elliptic curve implementation, a low degree of fast parallelism is more desirable than a high degree of slow parallelism. Such applications are not uncommon in the telecommunication industry. In the following sections, we describe the multiplication algorithms. Section 4.4.1 describes the sequential version, and section 4.4.2 begins with the description of SIMD addition and subtraction algorithms and finishes with the SIMD multiplication algorithm.

### 4.4.1 Modular arithmetic

It is well known that if the modulus remains unchanged throughout the computation, as is always the case in our applications, a considerable speed-up can be gained by using the so-called *Montgomery representation* [48]. This speed-up comes from the fact that divisions are replaced by shifts to return a result in the required range. The remainder of this section discusses numbers in Montgomery form.

We fix $n$ as an odd positive integer that will serve as the modulus. For an integer $x$ we define the residue of $x$ modulo $n$, denoted $x \bmod n$, as the smallest non-negative integer that is congruent to $x$ modulo $n$. We will measure the run-time of the algorithms to be presented in number of *elementary multiplications*, where one elementary multiplication computes the $2b$-bit product of two $b$-bit integers, for some positive integer $b$ that depends on the word size of the machine/processor we are using. We will assume that $b \geq 2$. The case $b = 1$ is trivial, and can be found in [48]. We are concerned with arithmetic on numbers which are much too large to be stored in one computer word. Thus we represent each number $x \bmod n$ as a sequence $x_{r-1}x_{r-2}\ldots x_2x_1x_0$ of $r$ non–negative $b$ bit words, which means that $x \equiv \sum_{i=0}^{r-1} x_i 2^{bi} \bmod n$ and $0 \leq x_i < 2^b$ for $0 \leq i < r$, where $r$ is the smallest integer with $2^{br} > n$. The Montgomery representation $\tilde{x}$ of an integer $x$ is the integer $(x \cdot R) \bmod n$, where $R = 2^{br}$. Notice that because $n$ is fixed $r$ and $R$ are also fixed.

Clearly, if $s = (x+y) \bmod n$, then $\tilde{s}$ is either $\tilde{x}+\tilde{y}$ or $\tilde{x}+\tilde{y}-n$, so that addition (or subtraction) of numbers in Montgomery representation is not different from ordinary modular addition (or subtraction). Multiplication, however, becomes simpler than ordinary multiplication modulo $n$. If $z = (x \cdot y) \bmod n$, then $\tilde{x} \cdot \tilde{y} \equiv x \cdot y \cdot R^2 \bmod n$, and therefore $\tilde{z} = (x \cdot y \cdot R) \bmod n = (\tilde{x} \cdot \tilde{y} \cdot R^{-1}) \bmod n$. It follows that the Montgomery product of two numbers in Montgomery representation can be computed using one ordinary multiplication of $\tilde{x}$ and $\tilde{y}$ followed by a division by $R$ modulo $n$. To be able to easily divide by $R$, the Montgomery multiplication algorithm adds a multiple of $n$

to $\tilde{x} \cdot \tilde{y}$ which is computed so that the low order $br$ bits of the result are zero. This allows division by $R$ as a simple shift. The result of the shift has only $br$ bits and thus can easily be reduced to the required residue mod $n$. The details of the computation follow.

**4.4.1.1 Montgomery multiplication**. Let $d$ be such that $d \cdot n \equiv -1 \bmod 2^b$ and $0 < d < 2^b$; this $d$ is well-defined because $n$ is odd. First, compute the ordinary product $u = \tilde{x} \cdot \tilde{y}$. Let $u = u_{2r-2}u_{2r-3} \ldots u_2u_1u_0$ be the radix $2^b$ representation of $u$. Next, we iterate the following from $i = 0$ to $r - 1$: compute $t = \Big((u_i \cdot d) \bmod 2^b\Big) \cdot n$ and note that $t_0 \equiv -u_i \bmod 2^b$. Thus, we add $2^{bi} \cdot t$ to $u$ to "zero" $u_i$. We increment $i$ and continue the loop. Notice that after iteration $i$, the $b$ bit words $u_i$, $u_{i-1}$, …, $u_0$ are all zero and the new $u$ is congruent to the old $u$ modulo n. Thus when the loop terminates, we replace $u$ by $u/R$ by shifting $u$ over $br$ bits to the right. We now have that either $\tilde{z} = u$ or $\tilde{z} = u - n$; this follows from the fact that the original $u$ is $\leq (n-1)^2$, that at most $\sum_{i=0}^{r-1} 2^{bi}(2^b - 1)n = (R-1)n$ was added to $u$ in the course of the computation, and that $((n-1)^2 + (R-1)n)/R < 2n$. This implies that the final result follows after a few subtractions and comparisons.

This multiplication algorithm can be carried out in $2r^2 + r$ elementary multiplications, plus some additions: $r^2$ multiplications for the computation of $\tilde{x} \cdot \tilde{y}$, plus 1 multiplication for $(u_i \cdot d) \bmod 2^b$ and $r$ multiplications for $((u_i \cdot d) \bmod 2^b) \cdot n$ for each of the $r$ iterations. Here we assume that $d$ is computed once and for all at the initialization of the modular arithmetic for $n$ using, for instance, the extended Euclidean algorithm. Notice that no divisions are needed because division with remainder by $2^b$ can be carried out by shifting and masking operations.

**4.4.1.2 Reducing the number of multiplications**. If we precompute $d \cdot n$, then the product $u_i \cdot (d \cdot n)$ can be computed using only $r$ multiplications since the low order block of $(d \cdot n)$ is known to be $-1$. This means that the total number of multiplications could be reduced to $2r^2$, if we would replace $u$ by $u + 2^{bi} \cdot u_i \cdot (d \cdot n)$

instead of $u + 2^{bi} \cdot \left((u_i \cdot d) \bmod 2^b\right) \cdot n$, for $i = 0, 1, \ldots, r-1$ in succession. The disadvantage of this approach is, however, that $\tilde{z}$ cannot be derived easily from the final $u$, because the final $u$ can be as large as $2^b \cdot n$.

**4.4.1.3 Remark**. There are various other ways to avoid divisions in modular multiplication. For instance, if a table containing $2^{bi} \bmod n$ for $i = r, r+1, \ldots, 2r$ has been precomputed, the product of two residues modulo $n$ can be reduced modulo $n$ using $r^2$ elementary multiplications. This is less than the $r^2 + r$ elementary multiplications needed in the reduction part of Montgomery multiplication, and has the (small) advantage that one can work with the 'ordinary' representation. In practice, however, it often turns out to be slower, due to the frequent memory fetches to retrieve the elements of the table; another disadvantage that might be serious for some implementations is the fact that the method requires storage for the table.

In the next section we discuss a version of Montgomery multiplication that uses $r+1$ processors, and that takes time $2r+1$, plus some additions.

### 4.4.2 SIMD Montgomery arithmetic

Let $P_0$, $P_1$, …, $P_r$ be a *processor array*, with $r$ as above, that operates in 'SIMD mode,' i.e., the $P_i$ carry out the same instruction at the same time, but each $P_i$ has its own data, and not every $P_i$ has to participate. The instruction stream for the processor array is provided by a *master processor* $M$, which might simultaneously provide several different processor arrays with (identical) instructions. We assume that $P_i$ can send data to $P_{i+1}$ (for all $i < r$ simultaneously) or to $P_{i-1}$ (for all $i > 0$ simultaneously), and that both $P_0$ and $P_r$ can broadcast data to all other $P_i$. Furthermore, we assume that each $P_i$ has a special flag register that can be set either to *true* or *false*, such that it takes $M$ a small constant amount of time (independent of $r$ and independent of the number of processor arrays governed by $M$) to decide if at least one of the flags is set to true. Finally, we assume that each

$P_i$ can carry out elementary multiplications, for some appropriately chosen $b > 1$, as well as additions, subtractions, and shifting and masking operations on local $2b$-bit integer variables. These last assumptions are meant to allow division with remainder by $2^b$ without actually dividing.

This set-up is not unrealistic. For instance, a $128 \times 128$ processor MasPar computer [45], can be thought to consist of $128 \cdot [128/(r+1)]$ disjoint processor arrays that satisfy the above conditions, for any $r < 128$ and $b \leq 32$.

In this section we describe how Montgomery arithmetic can be performed on numbers represented by the processor array $P_0$, $P_1$, ..., $P_r$, where the processor array represents a non-negative integer $a < 2^{b(r+1)}$ if each $P_i$ contains a non-negative $b$-bit integer $a_i$ such that $a = \sum_{i=0}^{r} a_i 2^{bi}$. The $a_i$'s are said to represent $a$. Let $x$ and $y$ be two integers, and let $\tilde{x}$ and $\tilde{y}$ be their Montgomery representations.

### SIMD Montgomery addition

Let $\tilde{x}_i$ and $\tilde{y}_i$ be non-negative $b$-bit integers stored at $P_i$, for $i = 0, 1, \ldots, r$, such that the $\tilde{x}_i$'s represent $\tilde{x}$ and the $\tilde{y}_i$'s represent $\tilde{y}$. Notice that both $\tilde{x}_r = 0$ and $\tilde{y}_r = 0$. We describe how to compute non-negative $b$-bit integers $\tilde{s}_i$ on $P_i$, for $i = 0, 1, \ldots, r$, that represent $\tilde{s}$, where $s = x + y$. We assume that $P_i$ has local variables $c_i$, $e_i$, $n_i$, $t_i$, and $u_i$, where the $n_i$'s represent the modulus $n$, and that these variables can contain $(b + 2)$-bit integers.

*Rough description:* The algorithm first computes $t = \tilde{x} + \tilde{y}$ (steps 1 through 7), next $u = t - n$ (steps 8 through 14), and finally sets $\tilde{s} = t$ if $u < 0$ or $\tilde{s} = u$ if $u \geq 0$ (Steps 15 and 16).

*The algorithm.* Perform the steps below sequentially on all $P_i$ simultaneously, unless indicated otherwise.

| | | |
|---|---|---|
| 1 | $t_i = \tilde{x}_i + \tilde{y}_i$; | {The addition} |
| 2 | $e_i = t_i/2^b$; $c_i = 0$; | {Compute carry bit $e_i$} |

3 while some $e_i = 1$ do {Carry propagation loop using special flag}
4 for all $P_i$ with $i > 0$ do
5 $c_i = e_{i-1}$; {Send the carry to the next processor}
6 $t_i = t_i \bmod 2^b + c_i$; {Add the carry}
7 $e_i = t_i / 2^b$; {Compute the new carry}
8 $u_i = t_i - n_i$; {Compute $t - n$ in case $t$ might be $> n$}
9 $e_i = 0$; {Zero the carry bit and compute carry}
for all $P_i$ with $i < r$ do {on all processors except the leftmost}
if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$
10 while some $e_i = 1$ do {Carry propagation loop using special flag}
11 for all $P_i$ with $i > 0$ do
12 $c_i = e_{i-1}$; {Send the carry to the next processor}
13 $u_i = u_i - c_i$; {Subtract the carry}
14 $e_i = 0$; {Zero the carry bit and compute carry}
for all $P_i$ with $i < r$ do {on all processors except the leftmost}
if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$
15 $c_i = u_r$; {Broadcast $u_r$ to all $P_i$}
16 if $c_i < 0$ then $\tilde{s}_i = t_i$ else $\tilde{s}_i = u_i$; {Select $t$ or $t - n$}

In steps 5 and 12 we use our assumption that processors can send data to neighboring processors; in Step 15 the broadcast feature is used. It is easy to find examples where the carry propagation loops (steps 4 through 7, and steps 11 through 14), have to be repeated $r$ times. A $\log r$ depth carry propagation tree would give a better worst case performance (if the processor array would allow implementation of such a structure). In our experience, however, our variant gives a much better average performance, because the carry propagation loops are hardly ever executed more than once. This is the case even if $M$ governs several thousand processor arrays simultaneously. In that case the 'speed' is determined by the processor array that needs most iterations through steps 3 or 10; notice that the other processor arrays carry out the same instructions, but that the instructions have no effect on the values stored.

We subtract $n$ from $t = \tilde{x} + \tilde{y}$ without first checking if the subtraction is actually needed; this is because it is as easy to compute $u = t - n$ as it is to test whether $t \geq n$, and given $t$ and $u$ it is easy to decide which of the two is the final answer. This is faster than doing a comparison to see if the subtraction has to be carried out, in particular if $M$ governs more processor arrays. Clearly, this SIMD addition can also be used for ordinary (i.e., non-Montgomery) addition of residues modulo $n$.

### SIMD Montgomery subtraction

Under the same assumptions as in the addition algorithm, we describe how to compute non-negative $b$-bit integers $\tilde{h}_i$ on $P_i$, for $i = 0, 1, \ldots, r$, that represent $\tilde{h}$, where $h = x - y$.

*Rough description.* The algorithm first computes $u = \tilde{x} - \tilde{y}$ (steps 1 through 7) and terminates if $u \geq 0$, setting $\tilde{h} = u$ (Step 8), otherwise it computes $t = u + n$ (steps 9 through 15) and sets $\tilde{h} = t$ (Step 16).

*The algorithm.* Perform the steps below sequentially on all $P_i$ simultaneously, unless indicated otherwise.

| | | |
|---|---|---|
| 1 | $u_i = \tilde{x}_i - \tilde{y}_i$; $c_i = 0$; | $\{$The subtraction$\}$ |
| 2 | $e_i = 0$; | $\{$Zero the carry bit and compute carry$\}$ |
| | for all $P_i$ with $i < r$ do | $\{$on all processors except the leftmost$\}$ |
| |     if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$ | |
| 3 | while some $e_i = 1$ do | $\{$Carry propagation loop using special flag$\}$ |
| 4 |     for all $P_i$ with $i > 0$ do | |
| 5 |         $c_i = e_{i-1}$; | $\{$Send the carry to the next processor$\}$ |

6 $\quad u_i = u_i - c_i;$ $\quad\{$Add the carry$\}$<br>
7 $\quad e_i = 0;$ $\quad\{$Zero the carry bit and compute carry$\}$<br>
for all $P_i$ with $i < r$ do $\quad\{$on all processors except the leftmost$\}$<br>
if $u_i < 0$ then $e_i = 1$, $u_i = u_i + 2^b$<br>
8 $\quad e_i = u_r;$ $\quad\{$Broadcast $u_r$ to all $P_i$ so that$\}$<br>
if $e_i = 0$ then $\tilde{h}_i = u_i$ else $\quad\{$the result can be decided$\}$<br>
9 $\quad t_i = u_i + n_i;$ $\quad\{$The addition$\}$<br>
10 $\quad e_i = t_i/2^b;$ $\quad\{$Compute carry bit $e_i\}$<br>
11 $\quad$while some $e_i = 1$ do $\quad\{$Carry propagation loop using special flag$\}$<br>
12 $\quad$for all $P_i$ with $i > 0$ do<br>
13 $\quad c_i = e_{i-1};$ $\quad\{$Send the carry to the next processor$\}$<br>
14 $\quad t_i = t_i \bmod 2^b + c_i;$ $\quad\{$Add the carry$\}$<br>
15 $\quad e_i = t_i/2^b;$ $\quad\{$Compute the new carry$\}$<br>
16 $\quad \tilde{h}_i = t_i;$ $\quad\{$Store the result$\}$

If $M$ governs more processor arrays, it might be better to perform steps 9 through 15 irrespective of the value of $u_r$, and to choose between $u_i$ and $t_i$ at the very end. With only one processor array, however, the version presented above is faster. We refer for the discussion after the addition algorithm for additional comments.

### SIMD Montgomery multiplication

The straightforward SIMD implementation of Montgomery multiplication as it appears in (4.4.1.1) would involve $3r$ multiplications per processor using $r$ processors. In each of $r$ iterations, the product $x_i \cdot y_j$ is first computed (for $i = 0, 1, \ldots, r-1$ in parallel, i.e., $r$ multiplications in the time of a single one), and then $u_0 \cdot d \bmod 2^b$ must be computed (a single multiplication). The final product in each iteration is computing $(u_0 \cdot d \bmod 2^b) \cdot n_i$ (again, $r$ multiplications in the time of a single one) using the result of the previous product. Since the second result is needed at all processors in the final step, the second product can either be computed on a single processor and the result broadcast to all other processors or it can be computed at all processors. Since we are dealing with SIMD algorithms, there is no advantage to computing the product on a single processor; all other processors would simply be idle during this time. In any case, this straightforward method clearly leads to a major inefficiency caused by the inherent sequential nature of the process, which allows us to parallelize only $2r$ of the $2r+1$ multiplications per iteration; *one* multiplication remains unparallelizable and its execution costs a full multiplication per iteration.

Our goal is to reduce this inefficiency in the SIMD version and to get rid of this single unparallelizable multiplication per iteration. The observation in (4.4.1.2) helps lead to an efficient algorithm. By precomputing the product $d \cdot n$ and storing one word of the result at each processor in the array, the Montgomery multiplication will only take 2 multiplications per iteration. The drawback to this approach, as mentioned in (4.4.1.2), is that the result can be rather large and cannot be easily reduced to a residue mod $n$. The fix to this problem is to use the precomputed value only for the first $r-1$ iterations of the algorithm and use the longer 3 multiplication form for the last iteration. Using this approach the result can be shown to be less than $3n$ (cf. our analysis below), so that at most two subtractions suffice to do the final reduction modulo $n$.

More precisely, in the notation of (4.4.1.1), the algorithm works by first replacing $u = \tilde{x} \cdot \tilde{y}$ by $u + 2^{bi} \cdot u_i \cdot (d \cdot n)$ for $i = 0, 1, \ldots, r-2$ in succession, after which $u$ is replaced by $u + 2^{b(r-1)} \cdot ((u_{r-1} \cdot d) \bmod 2^b) \cdot n$, where the computation of $\tilde{x} \cdot \tilde{y}$ is merged with the divisions by $2^b$ modulo $n$. So, during iteration $i$ for $i < r-1$, we add a multiple of $n$ to $u$ which is less than $2^{bi}(2^b-1)^2 n$, and during the last iteration we add a multiple of $n$ which is less than $2^{b(r-1)}(2^b-1)n$ because the last iteration is identical to ordinary Montgomery multiplication presented in (4.4.1.1). This implies that at most

$$\Big(\sum_{i=0}^{r-2} 2^{bi}(2^b-1)^2 n\Big) + 2^{b(r-1)}(2^b-1)n$$

is added to $u$ in the course of the computation. Since the original $u$ is bounded by $(n-1)^2$, the final $u$ is bounded by

$$\Big((n-1)^2 + (2^b-1)(2^{b(r-1)}-1)n + Rn - 2^{b(r-1)}n\Big)/R < 3n.$$

The final result is therefore $u$, $u-n$, or $u-2n$.

Given $\tilde{x}_i$ and $\tilde{y}_i$ representing $\tilde{x}$ and $\tilde{y}$, we describe how to compute non-negative $b$-bit integers $\tilde{z}_i$ on $P_i$, for $i = 0, 1, \ldots, r$, that represent $\tilde{z}$, where $z = (x \cdot y) \bmod n$. We assume that $P_i$ has local variables $c_i$, $e_i$, $m_i$, $n_i$, $t_i$, $u_i$, and $w_i$, where the $n_i$'s represent the modulus $n$ and the $m_i$'s represent $(d \cdot n)/2^b$, where $d$ as in (4.4.1.1) is assumed to exist on all $P_i$'s. The variable $w_i$ can contain $2b$-bit integers, for the others $b+2$ bits suffice.

*Rough description.* The algorithm consists of $r$ iterations. During the $j$th iteration, with $0 \le j < r$, it computes $w = \tilde{y}_j \cdot \tilde{x}$ (steps 3 and 4) and replaces the accumulator $u$ by $u + w$ (steps 5 through 9), where $u$ is initially set to 0 (step 1). Not all carries are propagated in this addition, but $u_0$ *is* normalized (steps 8 and 9). Next $u$'s last $b$ bits are made zero by adding an appropriate multiple of $n$ to $u$. For this purpose, $u_0$ is first broadcast to all processors and stored in $t_i$ there (Step 10), after which $u$ is shifted to the right (in Step 11, where $\tilde{y}$ is also shifted to the right, to facilitate the broadcast of $\tilde{y}_j$ in Step 3), and a multiple of $n$ is added to the shifted $u$. In the first $r-1$ iterations this is done in steps 14 through 20 by adding $w = t \cdot ((d \cdot n)/2^b)$ to the already shifted $u$ (steps 15 through 17, where the carry of the part of $u$ that was shifted out is taken care of by $c_0$ on $P_0$); after this addition, carries in $u$ are propagated once (steps 18 through 20). In the last iteration (steps 21 through 30) a different $w$, namely $((t \cdot d) \bmod 2^b) \cdot (n/2^b)$, which takes one more elementary multiplication to compute, is added (steps 22 through 26), and in this last iteration, all carries in $u$ are propagated (steps 27 through 30). In steps 31 through 37 the correct result ($u$, $u-n$, or $u-2n$) is chosen.

*The algorithm.* Perform the steps below sequentially on all $P_i$ simultaneously, unless indicated otherwise.

1 $u_i = 0$; $c_i = 0$; {Zero the accumulator and carry bit}
2 for $j = 0$ to $r-1$ do {One iteration for each block of $y_i$}
  {Multiplication step: Compute $\tilde{x} \cdot \tilde{y}_j$ and add the result to $u$}
3  $t_i = \tilde{y}_0$ {Broadcast the next block of $y_i$}
4  $w_i = t_i \cdot \tilde{x}_i$; $e_i = w_i/2^b$; {Do the multiplication and compute the carry}
5  for $P_i$ with $i > 0$ set $c_i = e_{i-1}$; {Propagate carries once for $w$}
6  $u_i = u_i + (w_i \bmod 2^b) + c_i$; {Add new result to accumulator}
7  $e_i = u_i/2^b$; {Compute the new carry}
8  for $P_i$ with $i > 0$ set $c_i = e_{i-1}$; {Propagate carries once for $u$}
9  $u_i = (u_i \bmod 2^b) + c_i$; {Add the carry and normalize $u$}
  {Mod step: Zero $u_0$ by adding the correct multiple of $n$}
10  $t_i = u_0$; {Broadcast the low–order bits of $u$}
11  for all $P_i$ with $i < r$ do
   $u_i = u_{i+1}$; {Shift out the "zeroed" bits of $u$}
   $\tilde{y}_i = \tilde{y}_{i+1}$; {Move the next block of $y$ to $P_0$}
12  $u_r = 0$; {Shift a zero into the high bits of $u$}
13  if $j < r-1$ then {If this is not the last iteration}
14   if $t_0 > 0$ then {No correction necessary if $u_0 = 0$}
15    $w_i = t_i \cdot m_i$; $e_i = w_i/2^b$; {$w = t_0 \cdot \left((d \cdot n)/2^b\right)$ and compute carry}
16    $c_i = e_{i-1}$ for $i > 0$, and $c_0 = 1$ for $i = 0$ only;
     {The above multiplication would have produced a carry into $u_0$}
17    $u_i = u_i + (w_i \bmod 2^b) + c_i$; {Add the multiple of $n$ to $u$}

18 $e_i = u_i/2^b$; {Propagate carries once}
19 $c_i = e_{i-1}$ for $i > 0$, and $c_0 = 0$ for $i = 0$ only;
20 $u_i = (u_i \bmod 2^b) + c_i$;
21 else if $t_0 > 0$ then {Last iteration, check for $u_0 = 0$}
22 $e_i = n_{i+1}$ for $i < r$, and $e_r = 0$ for $i = 0$ only;
23 $w_i = \Big((t_i \cdot d) \bmod 2^b\Big) \cdot e_i$; {$w = \Big((t_0 \cdot d) \bmod 2^b\Big) \cdot (n/2^b)$ }
24 $e_i = w_i/2^b$; {Propagate carries once}
25 $c_i = e_{i-1}$ for $i > 0$;
26 $u_i = u_i + (w_i \bmod 2^b) + c_i$; {Add the proper multiple of $n$ and the carry}
27 $e_i = u_i/2^b$; {Compute the carry and}
28 while some $e_i = 1$ do {propagate completely using special flag}
29 $c_i = e_{i-1}$ for $i > 0$;
30 $u_i = (u_i \bmod 2^b) + c_i$; $e_i = u_i/2^b$;
{Choose the proper value: $u$, $u - n$, or $u - 2n$}
31 $t_i = u_i - n_i$; {Subtract $n$ and propagate carries}
32 $e_i = 0$; if $t_i < 0$ and $i < r$ then $e_i = 1$, $t_i = t_i + 2^b$;
33 while some $e_i = 1$ do
34 $c_i = e_{i-1}$ for $i > 0$;
35 $t_i = t_i - c_i$;
36 $e_i = 0$; if $t_i < 0$ and $i < r$ then $e_i = 1$, $t_i = t_i + 2^b$;
37 if $t_r < 0$ then $\tilde{z}_i = u_i$ {If $t < 0$ then $u < n$ is the proper result}
else $u_i = t_i$ and repeat the subtraction in lines 31–37;

There are two elementary multiplications per $P_i$ in the first $r - 1$ iterations (in steps 4 and 15), and three in the last iteration (one in Step 4 and two in Step 23). This implies that the algorithm takes time $2r+1$, plus some additions. In the computation of $t_i \cdot d$ in Step 23, the value of $t_i$ is identical on all $P_i$; computing this product on only one processor, and broadcasting the result to the others, does not save any time in our SIMD architecture. Notice that the full $2b$-bit product $t_i \cdot d$ is not needed in Step 23, but that its least significant $b$ bits suffice. This might make this single extra multiplication slightly faster.

### 4.4.3 Remarks

This section has shown how to avoid a direct but inefficient parallel implementation of a well known algorithm through a reformulation of the algorithm and its invariants. In our particular case, an efficient solution can be attained by relaxing the invariant for the first part of the algorithm and returning to a tighter invariant for a final stage in order to produce a correct result. Although we have not discussed further applications, we suspect that this general technique can be used to enhance the SIMD parallelizability of other algorithms.

## 4.5 SIMD sieve factoring

In this section we present an efficient SIMD-implementation of the multiple polynomial quadratic sieve (QS) factoring algorithm [50], still the most practical general purpose method for integers in the range from 80 to 120 digits.

The largest number factored by QS is a 116-digit number. This factorization was carried out in a few months on a widely distributed network of workstations, and took a total computation time of approximately 400 mips-years (1 mips-year is about $3.15 \cdot 10^{13}$ instructions) [41]. The previous QS record for a single-machine implementation had 101 digits, and was carried out on one processor of a four processor Cray Y-MP4/464 in 475 CPU-hours [54]. This record was broken by our factorization of the 110-digit RSA-challenge number.

As will be explained in Section 4.5.2, QS consists of two main steps: the sieving step, and the matrix elimination step. All successful parallel implementations of QS that we know of have followed the approach described in [11]: distribute the sieving step over any number of available processors, which work independently of each other, and collect their results and perform the matrix elimination at a central location. In [11] the sieving step was done on a local network of workstations using the Ethernet for the communication, in [40] the workstations are scattered around the world, and communicate with the central location using electronic mail. In both of these implementations the network of participating machines can be viewed as a loosely coupled multi-processor machine, where the processors work asynchronously in *multiple-instruction multiple data* (MIMD) mode; i.e., each processor carries out its own set of instructions on its own set of data. Very powerful and fairly expensive massively parallel MIMD machines, with more computational power than was ever achieved using the approach from [40], are currently available. It should not be hard to break the 116-digit QS record on such a machine, but getting enough computing time might be prohibitively expensive, and is certainly more expensive than it was to use the donated cycles from the internet network.

SIMD machines have proven to be very useful for the matrix elimination step of QS [37, 39]. For that application, as well as for various other factoring applications (cf. [37]), it suffices to have a fairly restrictive type of SIMD machines, namely machines that only allow direct addressing (where the memory address is part of the instruction). For efficient sieving applications, however, it is essential that indirect addressing is available as well, i.e., the memory address depends on a value local to the processor. Although this is not the case for all SIMD machines, we will assume throughout this paper that this requirement is met (as it is for the 16K MasPar SIMD machine that we used for our work, cf. Section 4.2).

Nevertheless, at first sight the sieving step does not look like the type of operation that would run well on any SIMD machine. Indeed, a MasPar-implementation of the

QS-sieving step which was attempted in [27] seems to support this supposition. In this section we describe a different approach to the SIMD-implementation of the sieving step which works quite efficiently: on a 16K MasPar 100-digit integers can be factored within three CPU-days. A 110-digit number took one CPU-month, where we used only 5/8 of the total available memory; using all the memory this factorization would have taken about 20 days. With a later, faster version of the program we were able to do a 105-digit number in 5.3 CPU-days, using half of the available memory. This shows that relatively inexpensive SIMD machines are much better for general purpose factoring than was previously expected.

The success of this implementation prompted work on a SIMD-implementation of the general number field sieve factoring method [7]. With this number field sieve implementation we broke the record set by the factorization of the ninth Fermat number, by factoring the 151-digit number $(2^{503}+1)/3$ and the 158-digit number $2^{523}-1$.

The remainder of this section is organized as follows. Section 4.5.1 contains a general description of QS. A simple algorithm for the redistribution of data on a SIMD machine is given in Section 4.5.2. An overview of our SIMD QS implementation is presented in Section 4.5.3, and Section 4.5.4 contains an example.

### 4.5.1 The multiple polynomial quadratic sieve factoring method

Let $n > 1$ be an odd positive integer that is not a prime power. For each random integer $x$ satisfying

$$x^2 \equiv 1 \bmod n \tag{4.1}$$

there is a probability of at least 1/2 that $\gcd(n, x-1)$ is a non-trivial factor of $n$. To factor $n$ it therefore suffices to construct several such $x$'s in a more-or-less random manner.

In many factoring algorithms, solutions to (4.1) are sought by collecting integers $v$ such that

$$v^2 \equiv \prod_{p \in P} p^{e_p(v)} \bmod n, \tag{4.2}$$

where the *factor base* $P$ is some finite set of integers that are coprime to $n$, and $e_p(v) \in \mathbf{Z}$ for $p \in P$. A pair $(v, e(v))$ satisfying (4.2), with $e(v) = (e_p(v))_{p \in P} \in \mathbf{Z}^{\#P}$, is called a *relation*, and will be denoted by $v$ for short. If $V$ is a set of relations with $\#V > \#P$, then there exist at least $2^{\#V - \#P}$ distinct subsets $W$ of $V$ with $\sum_{v \in W} e(v) = (2w_p)_{p \in P}$ and $w_p \in \mathbf{Z}$; these subsets can be found using Gaussian elimination modulo 2 on the set of vectors $e(v) \bmod 2$. Each such $W$ leads to an $x \equiv (\prod_{v \in W} v) \cdot (\prod_{p \in P} p^{-w_p}) \bmod n$ satisfying (4.1).

In the original quadratic sieve factoring algorithm [50] relations are collected as follows. Let $P$ consist of $-1$ and the primes $\leq B$ with Legendre symbol $(\frac{n}{p}) = 1$, for some bound $B$. An integer is called $B$-smooth if it can be written as a product over $P$. Relations are collected by looking for small integers $i$ such that $f(i) = (i + [\sqrt{n}])^2 - n$ is $B$-smooth; for such $i$ we have that $v = i + [\sqrt{n}]$ satisfies (4.2). Because a prime $p$ divides $f(i)$ if and only if it divides $f(i + kp)$ for any integer $k$, smooth values can be found efficiently using a sieve, if the roots of $f$ modulo the primes in $P$ are known. For this reason, the relation collecting step is called the *sieving step*. Notice that only primes $p$ with $(\frac{n}{p}) = 1$ can divide $f(i)$; this explains the definition of $P$.

The second step, finding subsets $W$ as above, is called the *matrix elimination step*. In this paper we will not pay any further attention to this step, as it is well known how it can be dealt with for $\#P$ up to, say, 200000 (cf. [37, 39]).

Let

$$L[\gamma] = \exp((\gamma + o(1))\sqrt{\log n \log\log n}), \tag{4.3}$$

for a real number $\gamma$, and $n \to \infty$. With $B = L[1/2]$ it can be shown on loose heuristic grounds that both steps of the quadratic sieve algorithm can be completed in expected time $L[1]$.

Because in the original quadratic sieve only one polynomial $f$ is used to generate all $> \#P$ relations, the interval of $i$-values to be inspected is rather large. Since $f(i) \approx 2i[\sqrt{n}]$ grows linearly with $i$, the probability of $f(i)$ being $B$-smooth decreases with increasing $i$. Davis [14] suggested using more than one polynomial, thus allowing a smaller $i$-interval per polynomial which should increase the yield (per $i$) of the sieving step. In our implementation we used Montgomery's version of this same idea [58]. Let $P$ be as above, and let

$$f(i) = a^2 i^2 + bi + (b^2 - n)/(4a^2), \tag{4.4}$$

for integers $a$, $b$ with $b^2 \equiv n \bmod 4a^2$. This requires $n$ to be 1 mod 4, which can be achieved by replacing $n$ by $3n$ if necessary; in practice it might even be advantageous to use some other multiplier, cf. [58]. If $f(i)$ is $B$-smooth, then $v = (ai + b/(2a)) \bmod n$ satisfies (4.2), and $B$-smooth $f(i)$'s can again be found using a sieve once the roots of $f$ modulo the primes in $P$ are known. Thus, polynomials satisfying (4.4) can be used to generate relations efficiently. The expected run-time of the resulting factoring algorithm, however, is still $L[1]$.

#### 4.5.1.1 Constructing polynomials.

We show how polynomials as in (4.4) can be constructed. Let $M$ be such that $f(i)$'s with $i \in [-M, M)$ will be tested for smoothness in the sieve. Let $a \equiv 3 \bmod 4$ be a probable prime with $a^2 \approx \sqrt{n/2}/M$ and Jacobi symbol $(\frac{a}{n}) = 1$. Since $a$ is free of primes in $P$, the polynomial $f$ has two roots modulo all primes in the factor base $P$. To find $b$ such that $b^2 \equiv n \bmod 4a^2$, we first set $b = n^{(a+1)/4} \bmod a$ so that $b^2 \equiv n \bmod a$. Next we replace $b$ by

$$b + a((2b)^{-1}((n - b^2)/a) \bmod a),$$

and finally, if $b$ turns out to be even we replace $b$ by $b - a^2$. It follows that $b^2 \equiv n \bmod 4a^2$, and $|f(i)| = O(i\sqrt{n})$ for $i \in [-M, M)$. Notice that the roots of $f \bmod p$ are $(-b \pm \sqrt{n})/(2a^2) \bmod p$, so that computation of the roots requires one inversion modulo $p$ for all primes $p$ in $P$; the values of $r_p \equiv (\sqrt{n}) \bmod p$ should be precomputed and stored in a table.

Another method to generate polynomials is presented in [51], and has the advantage that the roots of a polynomial modulo the primes in the factor base can be derived easily from the roots of the previous polynomial. We have no practical experience with this method.

In implementations of QS one usually also collects values of $i$ for which $f(i)$ factors almost completely using the elements of $P$, i.e., except for one (or two) larger primes. If the large primes in these so-called *partial relations* match, they can be combined to form relations as in (4.2). In practice this enhancement leads to a speed-up factor of 4 to 6. For a detailed description of how the number of useful combinations among the partial relations can be counted, and how the combinations can be actually formed, we refer to [41].

#### 4.5.1.2 The sieving step.

Summarizing, for some fixed choice of factor base $P$, sieving bound $M$ and $n \equiv 1 \bmod 4$, the QS-sieving step can be carried out by performing steps (a) through (h).

**(a)** For all primes $p$ in $P$ compute $r_p \equiv (\sqrt{n}) \bmod p$;

**(b)** Set $a_{\text{low}} = [\sqrt[4]{n/2}/\sqrt{M}]$;

**(c)** Compute the smallest $a > a_{\text{low}}$ and corresponding $b$ that satisfy the requirements of Section 4.5.1.1, and let $f$ be as in equation (4.4);

**(d)** For all primes $p$ in $P$ compute the roots $r_{p1}$ and $r_{p2}$ of $f \bmod p$ as $(-b \pm r_p)/(2a^2) \bmod p$;

**(e)** For all integers $i$ with $i \in [-M, M)$ set $s(i)$ to zero;

**(f)** For all primes $p$ in $P$ and $v = 1, 2$ replace $s(i)$ by $s(i) + [\log p]$ for all $i \in [-M, M)$ which are equal to $r_{pv}$ modulo $p$ (this is the actual sieving step);

**(g)** For all $i \in [-M, M)$ for which $s(i)$ is sufficiently close to the report bound $\log|f(i)|$, try to factor $f(i)$ using the elements of $P$, and store the resulting relations and partial relations (an $i$ for which $s(i)$ is close to the report bound is called a *report*);

**(h)** If more relations and partial relations are needed, replace $a_{\text{low}}$ by $a$ and go back to step (c); otherwise terminate.

#### 4.5.1.3 Practical remarks.

In step (f) one often does not sieve with the small primes in $P$, or replaces them by sufficiently large powers, to increase the speed. This lowers the number of reports in step (g), so that the report bound has to be lowered accordingly. For a 100-digit $n$, the factor base $P$ will have approximately 50000 elements, and the largest element of $P$ will be about $1.3 \cdot 10^6$. Because a small multiple of this largest prime is a good choice for $M$, several million $s(i)$'s have to be stored. Although each $s(i)$ is usually represented by a single byte (8 bits), the $s(i)$'s together might not fit in memory. In that case, the interval $[-M, M)$ is broken into smaller subintervals, which are processed consecutively as above. In practice the report bound $\log|f(i)|$ is often replaced by some appropriately chosen fixed bound.

#### 4.5.1.4 Parallelization.

The sieving step can easily be parallelized on any number of independent processors, by restricting each of them to a unique interval of candidate $a$-values, disjoint from the intervals assigned to other processors. Notice that two different identical processors that run the same sieving program and that started at the same time, each on its own interval of candidate $a$-values, are most likely to be at entirely different points in the program, even after a very short run: one processor might find a 'good' $a$-value earlier than the other in step (c), and thus begin earlier with the next steps, or one processor might find more reports in step (g) and spend more time on the trial divisions of the corresponding $f(i)$'s. Also at other points the precise instruction stream that gets executed might differ (for instance, in the inversions modulo $p$ in step (d)), but these differences are minor compared to the entirely different things that might happen in steps (c) and (g). In a situation where several sieves as presented above are processed simultaneously in SIMD-mode, this might lead to major inefficiencies, because the process that happens to be the slowest for a particular step sets the pace for that step. In the next section it is shown how these inefficiencies can be avoided at the expense of some memory.

### 4.5.2 Redistributing data

In our SIMD-implementations of (4.5.1.2)(c) and (4.5.1.2)(g) we find ourselves in the following situation. We have a toroidal mesh $M$ of $m$ SIMD processing elements (PE's) that allows fast communication between each PE and its eight nearest neighbors. For the 16K MasPar described in Section 2, for instance, $M$ would be the array of PE's, and $m$ would be 16K. Furthermore, there is an inexpensive SIMD-process $G$, such that each PE running $G$ has a fairly small probability $p$, independent from the other PE's, to generate a useful packet of information. These packets have to be processed by a time-consuming SIMD-process $B$. The goal is to process as many packets as possible, by repeating $G$ and $B$ indefinitely.

Clearly, since $p$ is small it is quite inefficient to perform $B$ right after $G$ because then only the few PE's that have found a packet would be processing $B$. Fortunately, in our situation we can take advantage of the following.

**(i)** $G$ can be repeated an arbitrary number of times before $B$ is executed (i.e., packets do not have to be used immediately after they have been generated);

**(ii)** It is irrelevant for $B$ on which PE a particular packet was found (i.e., packets may be generated and processed on different PE's);

**(iii)** It is not crucial that *all* packets that have been generated are also actually processed by $B$, but of course generating packets that will not be used leads to inefficiencies.

Using (i) we could keep a stack of packets per PE, and apply $G$ until each stack contains at least one packet. At that point all $m$ top of stack elements could be popped and processed by $B$, after which $G$ is again applied, and so on. For small $p$ and large $m$ this approach would require rather large stacks on the PE's unless many packets are discarded, using (iii).

A better solution that uses much smaller stacks and that avoids discarding too many packets redistributes the packets after every application of $G$, thus making use of (ii) as well. There are many ways to do this; for us the following worked entirely satisfactorily.

#### 4.5.2.1 Random redistribution

On all $m$ PE's simultaneously, do the following in SIMD-mode. Fix some arbitrary ordering $N_1$, $N_2$, ..., $N_8$ of the eight nearest neighbors (the same ordering for all PE's). For $i = 1, 2, \ldots, 8$, set $N = N_i$ and perform steps (a) and (b).

**(a)** Get the number of packets $n$ on $N$'s stack;

**(b)** If $n + 1$ is smaller than the number of packets on the PE's own stack, then perform steps (b1) through (b3);

**(b1)** Set $e$ equal to the top of stack packet, and pop this packet from the stack;

**(b2)** Push $e$ on the top of the stack of $N$;

**(b3)** Go back to step (a).

This approach resulted in the following behavior. Starting from empty stacks on all PE's it took on average $2/p$ applications of $G$ (each followed by (4.5.2.1)) until none of the PE's had an empty stack. From that point on it takes, after each execution of $B$ on all $m$ PE's, on average $1/p$ applications of $G$ (plus (4.5.2.1)), with a very small variance, before $B$ can be applied again to process $m$ packets. Except for the start-up stage, this is the best one could hope for.

Although we tried several orderings of the neighbors, we never noticed a significant difference in the performance. With stacks of at most 5 packets we occasionally lost packets but this introduced only a minor inefficiency. A simpler variant of (4.5.2.1) would be to remove the jump back to step (a) in the case that at least one packet has been moved to a neighbor. Similarly, $G$ can be repeated a few times, before (4.5.2.1) is applied (with the jump). We have no experience with these simplifications, but we suspect that they work equally well. Notice that (4.5.2.1) uses only communication with nearest neighbors, with toroidal wraparound, which keeps communication costs to a minimum.

### 4.5.3 A SIMD implementation of the QS-sieving step

Given the redistribution algorithm from the previous section, there are various ways to implement the QS-sieving step efficiently on a SIMD-machine, as long as the machine provides reasonably fast communication between neighbors. The simplest approach would be to let each PE generate polynomials as in (4.5.1.2)(c) (making sure that they try different $a$-values), until each PE got at least one polynomial (using a stack of polynomials and (4.5.2.1)), after which each PE performs steps (4.5.1.2)(d)-(g) using the polynomial it ended up with (on the top of its stack of polynomials). This works efficiently in SIMD mode and without further inter-processor communication, except for the trial divisions in (4.5.1.2)(g); the $f(i)$'s, however, can again be redistributed using (4.5.2.1), so that trial division too can be performed on all PE's simultaneously.

Although these applications of (4.5.2.1) solve the synchronization problems caused by SIMD execution of (4.5.1.2)(c) and (4.5.1.2)(g), this approach is inefficient on the 16K MasPar, because there is not enough memory per PE to store a sufficiently large chunk of the interval $[-M, M)$. Furthermore, the roots in (4.5.1.2)(d) would have to be recomputed for each subinterval of $[-M, M)$ to be processed, because there is not enough memory on a PE to store them.

The opposite approach would be to process one polynomial at a time, and to spread the interval $[-M, M)$ over the PE's. This is a feasible approach if there if an ordering $\mathrm{PE}_0$, $\mathrm{PE}_1$, ..., $\mathrm{PE}_{m-1}$ such that $\mathrm{PE}_i$ and $\mathrm{PE}_{i+1}$ can communicate quickly (with indices modulo $m$). On the 16K MasPar this would not be impossible, but it would lead to a fairly small subinterval per PE with a very small hit-probability during the sieving step, unless the combined interval $[-M, M)$ is chosen exceedingly long.

One row of 128 PE's on a 16K MasPar has a total amount of memory of $128 \times 64\mathrm{K}=$ 8MBytes, which is just about the right amount of memory to store the sieving interval $[-M, M)$ for the factorization of a 100-digit $n$. This observation suggests that on the 16K MasPar it might be a good idea to process 128 polynomials at a time, with each of the 128 rows of 128 processors taking care of one polynomial. This is the approach that we opted for. Since there will be no communication between processors in different rows, except for the redistributions (cf. (4.5.2.1)) during the polynomial generation and trial division steps, we restrict our description to what happens in a single row of 128 processors.

Let $\mathrm{PE}_0$, $\mathrm{PE}_1$, ..., $\mathrm{PE}_{127}$ be a row of 128 processors, such that $\mathrm{PE}_j$ and $\mathrm{PE}_{j+1}$ can communicate quickly (with indices modulo 128, i.e., with toroidal wraparound).

We remove $-1$ and the small primes from the factor base $P$, choose the remaining $P$ such that $\#P = 128 \cdot k$, for some integer $k$, and we partition $P$ over the row of processors in such a way that each processor stores $k$ primes (but see Remark (5.2)(f)). Furthermore, each processor contains $k$ square roots of $n$ modulo its $k$ primes (the $r_p$ from (4.5.1.2)(a)). Finally, we choose $M$ such that $M = 64 \cdot L$ for some integer $L$, and we divide the sieving interval $[-M, M)$ over the processors in such a way that $\mathrm{PE}_j$ stores the length $L$ interval $I_j = [(j-64)L, (j-63)L)$.

Suppose that we have repeatedly applied (4.5.1.2)(c) combined with (4.5.2.1) on the entire $128 \times 128$ processor array simultaneously until each of the 16K processors has a non-empty stack of polynomials. In particular, we suppose that each $\mathrm{PE}_j$ contains a unique polynomial $f_j$, for $0 \leq j \leq 127$. These polynomials are processed one at a time in 128 iterations. To process $f_j$ processor $\mathrm{PE}_j$ first broadcasts $f_j$ to the other processors in the row, so that all 128 processors share the same polynomial, after which the pipelined sieving from the next section is performed.

#### 4.5.3.1 Pipelined sieving

Suppose that $\mathrm{PE}_0$ through $\mathrm{PE}_{127}$ all have the same polynomial $f$, represented by $a$ and $b$ as in equation (4.4). The values for $m$ and $l$ below are the same on all processors and can thus be taken care of by the ACU (cf. Section 2). Perform steps (a) through (c) on $\mathrm{PE}_0$ through $\mathrm{PE}_{127}$ simultaneously.

**(a)** For all $i \in I_j$ set $s(i)$ to zero (cf. (4.5.1.2)(e));

**(b)** For $m = 1, 2, \ldots, k$ in succession (where $\#P = 128 \cdot k$), perform steps (b1) through (b5);

**(b1)** Let $p$ be the $m$th prime on $\mathrm{PE}_j$ and $r_p$ the corresponding squareroot of $n$ (with different $p$'s for different $j$'s);

**(b2)** Compute $r_{p1}$ and $r_{p2}$ as the smallest integers $\geq (j-64)L$ which are equal to $(-b+r_p)/(2a^2)$ and $(-b-r_p)/(2a^2)$ modulo $p$, respectively (cf. (4.5.1.2)(d)), and compute $\tilde{r}_{p1}$ and $\tilde{r}_{p2}$ as the smallest integers $\geq -64L$ which are equal to $r_{p1}$ and $r_{p2}$ modulo $p$, respectively;

**(b3)** Set $l = 0$;

**(b4)** For $v = 1, 2$, as long as $r_{pv} < (j-63)L$, replace $s(r_{pv})$ by $s(r_{pv}) + [\log p]$ and next $r_{pv}$ by $r_{pv} + p$ (cf. (4.5.1.2)(f));

**(b5)** Replace $l$ by $l+1$. If $l < 128$, replace the 5-tuple $(p, r_{p1}, r_{p2}, \tilde{r}_{p1}, \tilde{r}_{p2})$ by the corresponding 5-tuple from the left neighbor (with wraparound), on $\mathrm{PE}_0$ only replace $r_{p1}$ and $r_{p2}$ by $\tilde{r}_{p1}$ and $\tilde{r}_{p2}$, respectively, and return to step (b4);

**(c)** For all $i \in I_j$ for which $s(i)$ is sufficiently close to the report bound $\log |f(i)|$, push $i$, $a$, and $b$ on the top of a stack of reports (cf (4.5.1.2)(g)).

This finishes the description of the pipelined sieving algorithm. Notice that 128 polynomials are processed simultaneously on 128 rows of processors. After every execution of the algorithm the elements of the stack built in step (c) will be redistributed using (4.5.2.1). As soon as all 16K processors have at least one report, the $f(i)$'s are computed and the actual trial divisions (using the original factor base including the small primes) are carried out on 16K processors simultaneously. And after 128 applications of (4.5.3.1) new polynomials are generated using (4.5.1.2)(c) and (4.5.2.1).

### 4.5.4 Practical remarks

(a) In our implementation we removed the first 50 primes from $P$, to speed up the sieving step; the threshold is best determined experimentally.

(b) We gained considerable additional efficiency by distributing $P$ over the $\mathrm{PE}_i$ in such a way that its smallest 128 elements are taken care of by $m = 1$ in step (b), the next smallest 128 by $m = 2$, etc.: as soon as for some $m$ all $p$'s are $> L$, at most one sieve location per PE has to be updated in step (b4), which means that for larger $m$ simpler code can be used. An additional advantage of this approach is that $[\log p]$ in (4.5.3.1)(b4) can be replaced by a sufficiently close approximation which is the same over the entire array of processing elements, and which can be changed depending on the value of $m$. As usual, instead of $\log|f(i)|$ we used some fixed value for the report bound in (4.5.3.1)(c).

(c) In our implementation we only kept track of $r_{p1}$, $\tilde{r}_{p1}$ and $r_{p1} - r_{p2}$. Although this leads to slightly more complicated computations, it also resulted in fewer communications (a 4-tuple instead of a 5-tuple in (4.5.3.1)(b5)) and thus greater speed.

(d) If the interval $[-M, M)$ is too large, (4.5.3.1) can be applied repeatedly to subintervals of $[-M, M)$ simply by changing the definition of $I_j$ appropriately (but see Remark (f) below). In this way we could decrease the memory requirements of our implementation considerably, at the expense of an acceptable slow-down. So, instead of the full 64K per PE, we normally use only about 28K, thus allowing other people to share the machine with us.

(e) In the trial division, the PE's simultaneously process the primes in the complete factor base to build a list of primes occurring in their $f(i)$. After that, each PE works on this list of primes to determine the multiplicities and the remaining factor. All PE's for which the remaining factor is 1, or a sufficiently small prime $> B$, or a sufficiently small composite, dump their trial division results to disk. The remaining composites (which lead to partial relations with two large primes) are factored by a separate program that runs independently on the front end.

(f) In the above description each processor stores $k$ elements of $P$ and the corresponding roots, where $\#P = 128 \cdot k$, and processors in the same column store the same subsets. We found it more efficient to distribute these subsets over the columns, so that processors store at most $[(k-1)/128] + 1$ elements of $P$ plus the corresponding roots. This leads to some extra communication (in Step (4.5.3.1)(b1)), but it makes more memory available for the sieve. If the sieve is broken into smaller subintervals, as suggested in Remark (d), then we either need $O(k)$ memory per processor to remember various useful values for the next subinterval, or these values have to be recomputed for each subinterval. This leads to loss of memory (which could have been used for the sieve) or loss of time. In particular for large $\#P$ we found it more efficient *not* to use Remark (d) at all, even though the sieve gets so small compared to $\max P$ that many primes do not even hit it once.

### 4.5.5 Example

The largest number we have factored using our SIMD QS implementation is the 110-digit number from the RSA-challenge list [55]:

$$\begin{aligned} \mathrm{RSA}(110) = {} & 35794234179\,72586877499\,18078325684\,55403003778\,02422822619 \\ & 35329081904\,84670252364\,67741151351\,61112045040\,60317568667 \\ = {} & 58464182144\,06154678836\,55318297916\,23841986105\,05601062333 \cdot \\ & 61224210904\,93547576937\,03731756141\,88412257585\,54253106999 \end{aligned}$$

At the time of writing, this is the largest number factored on a single machine using a general purpose factoring method. This factorization took approximately a month of computing time on the 16K MasPar, where we used only 40K of the 64K bytes per PE. The factor base consisted of approximately $80,000$ primes, and the sieving interval $[-M, M)$ was broken into two consecutive pieces (cf. (4.5.4)(d)). We did not use the suggestions from Remark (4.5.4)(f) for this factorization; we later found out that they would have saved us quite some time. They were used, however, for the factorization of a 105-digit number in 5.3 CPU-days using 30K per PE. Extrapolation to a 110-digit number would give at most 20 CPU-days, still with 30K per PE.

# Bibliography


[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA 1974.

[2] M. Ajtai, A lower bound for finding predecessors in Yao's cell probe model, *Combinatorica*, 8(3) (1988) pp. 235–247.

[3] M. Ajtai, M. Fredman, and J. Komlós , Hash functions for priority queues, *Information and Control*, **63** (1984) pp. 217–225.

[4] N. Alon and B. Schieber Optimal preprocessing for answering on-line product queries, *Unpublished manuscript*

[5] B. Awerbuch and Y. Shiloach, New connectivity and MSF algorithms for shuffle-exchange network and PRAM, *IEEE Trans. on Computers*, Vol. C-36, No. 10, Oct. 1987, pp. 1258–1263

[6] M. Ben-Or, D. Kozen, and J. Reif, The Complexity of Elementary Algebra and Geometry, *J. Comput. System Sci.* **32**(2) (1986) pp. 251–264.

[7] D. J. Bernstein, A. K. Lenstra, A general number field sieve implementation, *Lecture Notes in Math, to appear*, 1993.

[8] H. Booth and J. Westbrook, Linear Algorithms for Analysis of Minimum Spanning and Shortest Path Trees in Planar Graphs, Yale University, Department of Computer Science, TR-768, Feb. 1990; also *Algorithmica*, to appear.

[9] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, S. S. Wagstaff, Jr., Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers, second edition,*Contemporary Mathematics* **22**, Amer. Math. Soc., Providence, 1988.

[10] M. Carey and C. Thompson, An efficient implementation of search trees on $\lceil \log N + 1 \rceil$ processors, *IEEE Trans. on Computers*, Vol. c–33, No. 11, Nov. 1984, pp. 1038–1041.

[11] T. R. Caron, R. D. Silverman, Parallel implementation of the quadratic sieve, *J. Supercomputing*, **1** (1988), 273–290.

[12] B. Chazelle, Efficient Polygon Triangulation, Princeton University, CS-TR-249-90, Feb. 1990.

[13] B. Chazelle, Triangulating a Simple Polygon in Linear Time, in *Proc. 31$^{st}$ Annual IEEE Sympos. on Foundations in Comput. Sci.*, 1990, Vol. 1, pp. 220–230; also *Discrete and Computational Geometry* **6**(5) (1991), to appear.

[14] J. A. Davis, D. B. Holdridge, Factorization using the quadratic sieve algorithm, Tech. Report SAND 83-1346, Sandia National Laboratories, Albuquerque, NM, 1983.

[15] B. Dixon, A. K. Lenstra, Massively parallel elliptic curve factoring, Advances in Cryptology, Eurocrypt '92, *Lecture Notes in Comput. Sci.* **658** (1993), 183–193.

[16] B. Dixon, A. K. Lenstra, Factoring integers using SIMD sieves, Advances in Cryptology, Eurocrypt '93, *Lecture Notes in Comput. Sci.* To appear.

[17] B. Dixon, A. K. Lenstra, Fast Modular Multiplication, Contributed paper, *1993 DAGS symposium on parallel programming.*

[18] B. Dixon, M. Rauch, and R. E. Tarjan, Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time, *SIAM J. Comput.* **21**(6) (1992) pp. 1184–1192.

[19] C. Ellis, Concurrent search and insertion into 2–3 trees, *Acta Informatica*, 14 (1980) pp.63–86.

[20] P. van Emde Boas, R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Systems Theory*, **10** (1977) pp.99–127.

[21] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Information Processing Letters*, Vol. 6, No. 3 (1977) pp.80–82.

[22] M.L. Fredman, New Bounds on the Complexity of the Shortest Path Problem, *SIAM J. Comput.* **5**(1) (1976) pp. 83–89.

[23] M.L. Fredman and D.E. Willard, Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths, in *Proc. 31$^{st}$ Annual IEEE Sympos. on Foundations in Comput. Sci.*, 1990, Vol. 2, pp. 719–725.

[24] H.N. Gabow, Z. Galil, T. Spencer, and R.E. Tarjan, Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs, *Combinatorica* **6**(2) (1986) pp. 109-122.

[25] H.N. Gabow and R.E. Tarjan, A Linear-Time Algorithm for a Special Case of Disjoint Set Union, *J. Comput. System Sci.* **30**(2) (1985) pp. 209–221.

[26] H. Gajewska and R.E. Tarjan, Deques with Heap Order, *Information Processing Letters* **22**(4) (1986) pp. 197–200.

[27] A. Gjerken, Faktorisering og parallel prosessering (in norwegian), Bergen 1992.

[28] W. Goddard, V. King, and L. Schulman, Optimal Randomized Algorithms for Local Sorting and Set-Maxima, in *Proc. $22^{nd}$ Annual ACM Sympos. on Theory of Computing*, 1990, pp. 45-53.

[29] R.L. Graham and P. Hell, On the History of the Minimum Spanning Tree Problem, *Ann. Hist. Comput.* **7**(1) (1985) pp. 43–47.

[30] D. Harel, A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems, in *Proc. $17^{th}$ Annual ACM Sympos. on Theory of Computing*, 1985, pp. 185–194.

[31] D. Harel and R.E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors, *SIAM J. Comput.* **13**(2) (1984) pp. 338–355.

[32] D. B. Johnson and P. Metaxas, A parallel algorithm for computing minimum spanning trees, *4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp. 363–372.

[33] R. Karp and V. Ramachandran, Parallel Algorithms for Shared Memory Machines, *Handbook of Theoretical Computer Science* (1990) Chapter 17.

[34] V. King, personal communication.

[35] J. Komlós, Linear Verification for Spanning Trees, *Combinatorica* **5** (1985) pp. 57–65.

[36] L.L. Larmore, An Optimal Algorithm with Unknown Time Complexity for Convex Matrix Searching, *Information Processing Letters*, to appear.

[37] A. K. Lenstra, Massively parallel computing and factoring, Proceedings Latin'92, *Lecture Notes in Comput. Sci.* **583**.

[38] A. K. Lenstra, H. W. Lenstra, Jr., Algorithms in number theory, Chapter 12 in: J. van Leeuwen (ed.), *Handbook of theoretical computer science*, Volume A, *Algorithms and complexity*, Elsevier, Amsterdam, 1990.

[39] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard, The factorization of the ninth Fermat number, *Math. Comp*, **61** (1993).

[40] A. K. Lenstra, M. S. Manasse, Factoring by electronic mail, Advances in Cryptology, Eurocrypt '89, *Lecture Notes in Comput. Sci.* **434** (1990), 355–371.

[41] A. K. Lenstra, M. S. Manasse, Factoring with two large primes, *Math. Comp, to appear.*

[42] H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Ann. of Math.* **126** (1987), 649–673.

[43] U. Manber and R. Ladner, Concurrency control in a dynamic search structure, *ACM Trans. on Database Sys.*, Vol. 9, No.3 (1984) pp. 439–455.

[44] G. Marsaglia, A. Zaman, A new class of random number generators, *Ann. of Appl. Prob.* **1** (1991), 462–480.

[45] *MasPar MP-1 principles of operation*, MasPar Computer Corporation, Sunnyvale, CA, 1989.

[46] U.M. Maurer, Y. Yacobi, Non-interactive public key cryptography, Advances in Cryptology, Eurocrypt '91,*Lecture Notes in Comput. Sci.*, to appear.

[47] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer–Verlag, Berlin, 1984.

[48] P. L. Montgomery, Modular multiplication without trial division, *Math. Comp* **44** (1985), 519–521.

[49] P. L. Montgomery, Speeding the Pollard and elliptic curve methods of factorization, *Math. Comp.* **48** (1987), 243–264.

[50] C. Pomerance, Analysis and comparison of some integer factoring algorithms, 89–139 in: H. W. Lenstra, Jr., R. Tijdeman (eds), *Computational methods in number theory*, Math. Centre Tracts **154/155**, Mathematisch Centrum, Amsterdam, 1983.

[51] C. Pomerance, J. W. Smith, R. Tuler, A pipeline architecture for factoring large integers with the quadratic sieve algorithm, *SIAM J. Comput.* **17** (1988) 387–403.

[52] V. Rao and V. Kumar, Concurrent access of priority queues, *IEEE Trans. on Computers*, Vol. 37, No. 12, Nov. 1988, pp. 1657–1665.

[53] J. H. Reif An Optimal parallel algorithm for integer sorting, *Proc. STOC 26* (1985) 496-504.

[54] H. te Riele, W. Lioen, D. Winter, Factorization beyond the googol with mpqs on a single computer, *CWI Quarterly*, **4**, 69–72, Amsterdam, 1991.

[55] RSA Data Security Corporation Inc., sci.crypt, May 18, 1991; information available by sending electronic mail to challenge-rsa-list@rsa.com.

[56] B. Schieber and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM J. Comput.* **17**(6) (1988) pp. 1253–1262.

[57] R. Sedgewick, *Algorithms*, Addison–Wesley, 1988.

[58] R. D. Silverman, The multiple polynomial quadratic sieve, *Math. Comp.* **84** (1987), 327–339.

[59] R. D. Silverman, S. W. Wagstaff, Jr., A practical analysis of the elliptic curve factoring algorithm, manuscript.

[60] R.E. Tarjan, Applications of Path Compressions on Balanced Trees, *J. Assoc. Comput. Mach.* **26**(4) (1979) pp. 690–715.

[61] R.E. Tarjan, Sensitivity Analysis of Minimum Spanning Trees and Shortest Path Trees, *Information Processing Letters* **14**(1) (1982) pp. 30–33. Corrigendum, Ibid **23**(4) (1986), p.219.

[62] R.E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[63] S. S. Wagstaff, Jr., Update 2.4 to [9].

[64] D. E. Willard, Log–logarithmic worst case range queries are possible in space $\Theta(N)$, *Information Processing Letters* **17**(2) (1983) pp. 81–84.