Theory Comput. Systems **32**, 213–239 (1999)

**Theory of Computing Systems**
© 1999 Springer-Verlag New York Inc.

# Pipelining with Futures*

G. E. Blelloch[1] and M. Reid-Miller[2]

[1] School of Computer Science, Carnegie Mellon University,
Pittsburgh, PA 15213-3890, USA
blelloch@cs.cmu.edu

[2] Lycos, Inc., 5001 Centre Ave.,
Pittsburgh, PA 15213, USA
mrmiller@lycos.com

**Abstract.** Pipelining has been used in the design of many PRAM algorithms to reduce their asymptotic running time. Paul, Vishkin, and Wagener (PVW) used the approach in a parallel implementation of 2-3 trees. The approach was later used by Cole in the first $O(\lg n)$ time sorting algorithm on the PRAM not based on the AKS sorting network, and has since been used to improve the time of several other algorithms. Although the approach has improved the asymptotic time of many algorithms, there are two practical problems: maintaining the pipeline is quite complicated for the programmer, and the pipelining forces highly synchronous code execution. Synchronous execution is less practical on asynchronous machines and makes it difficult to modify a schedule to use less memory or to take better advantage of locality.

In this paper we show how futures (a parallel language construct) can be used to implement pipelining without requiring the user to code it explicitly, allowing for much simpler code and more asynchronous execution. A runtime system manages the pipelining implicitly. As with user-managed pipelining, we show how the technique reduces the depth of many algorithms by a logarithmic factor over the nonpipelined version. We describe and analyze four algorithms for which this is the case: a parallel merging algorithm on trees, parallel algorithms for finding the union and difference of two randomized balanced trees (treaps), and insertion into a variant of the PVW 2-3 trees. For three of these, the pipeline delays are data dependent

* This work was partially supported by DARPA Contract No. DABT63-96-C-0071 and by an NSF NYI award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government. This work was done while the second author was at Carnegie Mellon University.

making them particularly difficult to pipeline by hand. To determine the runtime of algorithms we first analyze the algorithms in a language-based cost model in terms of the work $w$ and depth $d$ of the computations, and then show universal bounds for implementing the language on various machine models.

## 1. Introduction

Pipelining in parallel algorithms takes a sequence of tasks each with a sequence of steps and overlaps in time the execution of steps from different tasks. Due to dependences between the tasks or the required resources, pipelined algorithms are designed such that each task is some number of steps ahead of the task following it. Pipelining has been used to improve the time of many parallel algorithms for shared-memory models. Paul et al. described a pipelined algorithm for inserting $m$ new keys into a balanced 2-3 tree with $n$ keys [28]. They first considered a nonpipelined algorithm that has $O(\lg m)$ tasks, each of which takes $O(\lg n)$ parallel time (steps), for a total time of $O(\lg n \lg m)$ on an EREW PRAM. Each task works its way up from the bottom of the insertion tree to the top, one level at a time. They then showed how to reduce the time to $O(\lg m + \lg n)$ by pipelining the tasks through the tree. The idea is that when task $i$ is working on level $j$ of the tree, task $i + 1$ can work on level $j - 1$, and so on.

Cole used a similar idea to develop the first $O(\lg n)$ time PRAM sorting algorithm that was not based on the AKS sorting network [19]; the AKS sorting network [2] has very large constants and is therefore considered impractical. The algorithm is based on parallel mergesort, and it uses a parallel merge that takes $O(\lg n)$ time. The natural implementation would therefore take $O(\lg^2 n)$ time—the depth of the mergesort recursion tree is $O(\lg n)$ and the merge task at level $i$ from the top takes $O(\lg n - i)$ time. Cole showed, however, that the merge tasks can be pipelined up the recursion tree so that each merge can pass partial results to the node above it before it completes, and that this leads to a work-efficient algorithm that takes $O(\lg n)$ time. The basic idea of Cole's mergesort was later used in a technique called cascading divide-and-conquer, which improved the time of many computational geometry algorithms [3].

Although pipelining has led to theoretical improvements in algorithms, from a practical point of view pipelining can be very cumbersome for the programmer—managing the pipeline involves careful timing among the pipeline tasks and assumes a highly synchronous model. The central idea of this paper is to show that many algorithms can be automatically pipelined using futures, a construct designed for parallel languages [21], [5]. Using futures, coding the pipelined algorithms is remarkably simple; we push the complexity of managing the pipeline and scheduling the threads to a single provably efficient runtime system. In addition, our approach is the first that addresses asynchronous pipelined algorithms where the pipeline depth is dynamic and depends on the input data. We present and analyze several algorithms that require such an asynchronous pipeline. The approach also gives a natural way to restrict algorithms so they have no concurrent memory accesses.

The futures construct was developed in the late '70s for expressing parallelism in programming languages and has been included in several programming languages [24], [25], [15], [17], [16]. Conceptually, the *future* construct forks a new thread $t_1$ to calculate

a value (evaluate an expression) and immediately returns a pointer to where the result of $t_1$ will be written. This pointer can then be passed to other threads. When a thread $t_2$ needs the result of $t_1$, it uses the pointer to request the value. If the value is ready (has been written) it is returned immediately, otherwise $t_2$ waits until the value is ready. To avoid deadlocks and for efficiency $t_2$ is typically suspended while waiting so that other threads can run.

To analyze the running times of algorithms programmed with futures we use a two-step process. We first consider a language-based cost model based on futures and analyze the algorithms in this model. We then show universal bounds for efficiently implementing the model on various machine models.

*Algorithm Analysis.* For the cost model we use a slight variation of the PSL model [23]. In this model computations are viewed as dynamically unfolding directed acyclic graphs (DAGs), where each node is a unit of computation (action) and each edge between nodes represents a dependence implied by the language. There are three types of dependence edges in the DAG, *thread edges* between two successive actions in a thread, *fork edges* from the node that creates a future to the first node of the future's thread, and *data edges* from the result of a future to all the nodes that request the result. The cost of a computation is then calculated in terms of total *work* (number of nodes in the DAG) and the *depth* (longest path length in the DAG). Analyzing an algorithm in the model involves determining the work and depth of the algorithm as a function of the input size.

As an example of the use of futures and of the DAG cost model consider Figure 1. This example has a producer that produces a list of decreasing integers from $n$ down to 0, where each element of the list is created by its own thread. In parallel, a consumer consumes these values by summing them. This code pipelines producing and consuming the values.

```
fun produce(n) =
  if (n < 0) then nil
  else n::?produce(n-1);

fun consume(sum,nil) = sum
  | consume(sum,h::t) = consume(h+sum,t);

consume(0,?produce(n));
```

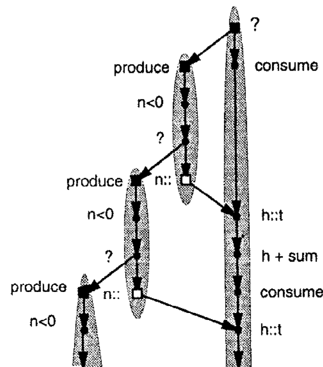



**Fig. 1.** Example code and the top of the corresponding computation DAG. The code syntax is based on ML and described in the Appendix. Futures are marked with a question mark (`?`). The `n::l` syntax adds the element `n` to the head of the list `l`. When used as a pattern, as in `h::t` in `consume`, it binds the head of the corresponding argument, which must be a list, to `h` and the tail to `t`. In the DAG each node represents an action and each vertical sequence of actions represents a thread. The vertical edges are thread edges, the edges going to the left are fork edges, and the edges going to the right are data edges.

We describe and analyze four algorithms with the cost model. The first is a merging algorithm. It takes two binary trees with the keys sorted in-order within each tree and merges them into a single tree sorted in-order. The code is very simple and, assuming both input trees are of size $n$, the nonpipelined parallel version requires $O(\lg^2 n)$ depth and $O(n)$ work. We show that, by using the same code but implementing it with futures, the depth is reduced to $O(\lg n)$, which meets previous depth bounds. The next two algorithms use a parallel implementation of the treap data structure [29]. We show randomized algorithms for finding the union and difference of two treaps of size $m$ and $n$, $m \leq n$ in $O(\lg n + \lg m)$ expected depth and $O(m \lg(n/m))$ expected work. Like the merge algorithm, the code is simple. There are no previous parallel or pipelined results for treaps of which we are aware. These three algorithms require a dynamic pipeline, which varies in depth depending on the input data. As such asynchronous algorithms have not been considered before, we developed a new technique for analyzing their computation depth. The fourth algorithm is a variant of Paul, Vishkin and Wagener's (PVW) 2-3 trees [28]. Because the bottom-up insertion used in the PVW algorithm does not map naturally into the use of futures, we describe a top-down variant that does. As with the PVW algorithm, the pipelining improves the algorithm complexity for inserting $m$ keys into a tree of size $n$ from $O(\lg n \lg m)$ to $O(\lg n + \lg m)$ depth. In both cases the work is $O(m \lg n)$. The algorithm can be implemented synchronously and with a fixed pipeline depth.

Although there has been some work on designing algorithms using futures, the emphasis of previous work has been on designing and implementing future-based languages. Because of this emphasis, to our knowledge none of the work has analyzed the asymptotic cost of algorithms. In fact, most algorithms previously designed using futures display no asymptotic performance advantage over simpler fork-join parallel algorithms. As an example consider the quicksort algorithm given in Figure 2. This algorithm was described by Halstead [24] as a prototypical future-based algorithm. The algorithm is pipelined since the partial results of a `partition` can be pipelined in recursive invocations of `qs`. From an asymptotic point of view, however, the expected depth of this algorithm is no better than a nonpipelined version, i.e., one that simply makes the two recursive calls to quicksort in parallel after the sequential partition is complete. In both

```
fun partition(elt,nil) = (nil,nil)
  | partition(elt,h::t) =
  let val (l,g) = ?partition(elt,t)
  in if (elt > h) then (h::l,g) else (l,h::g)
  end;

fun qs(nil,rest) = rest
  | qs(h::t,rest) =
  let val (l,g) = partition(h,t)
  in qs(l,h::?qs(g,rest)))
  end;

fun qsort(l) = qs(l,nil);
```

**Fig. 2.** The quicksort algorithm of Halstead transcribed from Multilisp into the ML syntax.

cases the algorithms have $O(n)$ expected depth. Even in terms of constant factors, the pipelined version has only a small factor more parallelism than the nonpipelined version.

*Implementation Analysis.* To complete the analysis we consider implementations of the language-based cost model on various machines. The work and depth costs along with Brent's scheduling principle [14] imply that, given a computation with depth $d$ and work $w$, there is a schedule of actions onto processors such that the computation will run in $w/p + d$ time on a $p$-processor PRAM. This principle, however, does not tell us how to find the schedule online—in particular it does not address the costs of dynamically assigning threads to processors nor the cost of handling the suspension and restarting required by futures at runtime. Since many of the algorithms are dynamic, the schedule cannot be computed off line. In addition, Brent's scheduling principle in general assumes concurrent memory access, requiring an implementation on a CRCW PRAM. Two key points of this paper are that all the scheduling and managing of futures can be handled by a runtime system in an algorithm-independent fashion with provable time bounds, and that by placing a restriction on the program type, we can guarantee the computation will require no concurrent memory accesses. We are interested in universal results that place bounds on the time taken by an implementation on various machine models, including all online costs for scheduling and management of futures.

Previous results on implementing a model similar to the one we use in this paper [23] have shown that any computation with $w$ work and $d$ depth can be implemented online on a CRCW PRAM in $O(w/p + d \cdot T_f(p))$ time, where $T_f(p)$ is the time for a fetch-and-add (or multiprefix) on $p$ processors. The fetch-and-add is used to manage queues for threads that are suspended waiting for a future to complete. In this paper we show that for programs that are converted to a form called *linear code*, any computation can be implemented on the EREW PRAM model in $O(w/p + d \cdot T_s(p))$ time, where $T_s(p)$ is the time for a scan operation (all-prefix-sums) used for load balancing the tasks. Our implementation also implies time bounds of $O(gw/p + d(T_s(p) + L))$ on the BSP [30], where $g$ is the BSP gap parameter and is inversely related to bandwidth and $L$ is the BSP periodicity parameter and is related to latency, $O(w/p + d \lg p)$ on an asynchronous EREW PRAM [20], and $O(w/p + d)$ on the EREW scan model [6]. The conversion to linear code is a simple manipulation that can be done by a compiler. Although this conversion can potentially increase the work and/or depth of a computation, it does not for any of the algorithms described in this paper. In fact, linear code seems to be a natural way to define EREW algorithms in the context of a language model.

When mapping algorithms onto a PRAM, our approach loses some time over previous pipelined algorithms. For example, when we map our $O(\lg n)$ depth, $O(m \lg n)$ work 2-3 tree algorithm onto the PRAM we get a time of $O(m \lg n/p + \lg n \cdot T_s(p))$ as opposed to $O(m \lg n/p + \lg n)$ for the PVW algorithm. We note, however, that when mapped directly onto more realistic models, such as the network models or the asynchronous PRAM, the algorithms perform equally well as the PRAM algorithms and with much simpler code: In the more realistic models, compaction using prefix sums has the same latency as either the memory read or write (network models) or the synchronization between steps (asynchronous PRAM). Furthermore, our approach can easily handle dynamic pipelines in which the structure and delays of the pipeline depends on the input data, such as the treap algorithms we describe. This would be considerably more difficult to do by hand and we know of no previous PRAM algorithms with dynamic pipelines.

## 2. The Model

As with the work of Blumofe and Leiserson [12], [13], we model a computation as a set of threads and the cost as the size of the computation DAG. Threads can fork new threads using a future, and can synchronize by requesting a value written by another thread. A computation begins with a single thread and completes when all threads have terminated.

A *future* call in a thread $t_1$ starts a new thread $t_2$ to calculate one or more values and allocates a *future cell* for each of these values.[1] The thread $t_1$ is passed *read pointers* to each future cell and continues immediately. These read pointers can be copied and passed around to other threads, and at any point any thread that has a pointer can read its value. The thread $t_2$ is passed *write pointers* to each future cell, which is where the results values are to be written as they are computed. The write pointers can also be passed around to other threads, but each can only be written to once. When a thread reads the value from a read pointer, sometimes called a *touch operation*, it must wait until the write to the corresponding cell has completed. As discussed in Section 4, the read is implemented by suspending the reading thread and reactivating it when the write occurs. Note that, although a future cell can be written to at most once, in general it can be read from multiple times. In Section 4 we show that when the code meets a certain condition called linearity the future cell is read at most once.

To specify when it is necessary to read from a read pointer we distinguish between strict and nonstrict operations. An operation is *strict* on an argument if it needs to know the value of that argument immediately. For example, all the arithmetic operations are strict on their arguments, and an operation that extracts an element from a cell is strict on that cell. An operation is *nonstrict* on an argument if it does not need to know the value of that argument immediately. For example, passing an object to a user-defined function or placing an object in a cell are nonstrict because the actual value is not needed immediately and a pointer to the value can be used instead. Whenever an operation is strict on an argument and that argument is a read pointer to a future cell, executing the operation will invoke a read on that future cell. We also assume that writing to a future cell is strict on the value that is being written. This means that a read pointer cannot be written into a future cell, which prevents chains of future cells. This restriction is important for proving bounds on the implementation.

Note that when building a data structure out of multiple cells, such as in a linked list or tree, operations are strict on the individual cells, not on the whole data structure. For example, if an operation examines the head of a linked list to get a pointer to the second element, the operation is strict on the head but not the second or any other element. We make significant use of this property in the algorithms in this paper.

To describe the algorithms in this paper, we use a subset of ML [27] extended with futures. The syntax is defined in the Appendix (see Figure 13). The subset we use is purely functional (no side effects), and we use arrays only for the 2-6 tree algorithm described in Section 3.4 and otherwise we just use trees. Futures are created by placing a `?` (question mark) before an expression, which will create a thread to evaluate the

[1] The ability to return multiple values and have separate future cells created for a single fork is actually quite important for some of the algorithms we present.

expression. The number of variables in an ML pattern determines the number of future cells that an expression creates. We make significant use of the ML pattern matching capabilities, and have, therefore, included a quick description in the Appendix.

We now consider the DAGs that correspond to computations in the model. The DAGs are generated dynamically as the computation proceeds and can be thought of as a trace of the computation. Each node in a DAG represents a unit-time action (the execution of a single instruction) and the edges represent dependencies among the actions. As mentioned in the Introduction, there are three kinds of dependence edges in the DAGs: thread edges, fork edges, and data edges. A thread is modeled as a sequence of actions connected by *thread edges*. When an action $a_1$ within a thread uses a future to start a thread $t_2$, a *fork edge* is placed from $a_1$ to the first action in $t_2$. When an action $a_1$ reads from a future-cell, a *data edge* is placed from the action $a_2$ that writes to that cell to $a_1$. The cost of a computation is then measured in terms of the number of nodes in the DAG, called the *work*, and the longest path length in the DAG, called the *depth*. In analyzing algorithms the goal is to determine the work and depth in terms of the input size. Determining the work is often simple since it is the time a computation would take sequentially if futures were not used. Determining the depth can be more difficult. As an aid we refer to the *time stamp* of a value as the depth in the DAG at which it is computed, and then find upper bounds on the time stamps of the results to determine the depth of the computation.

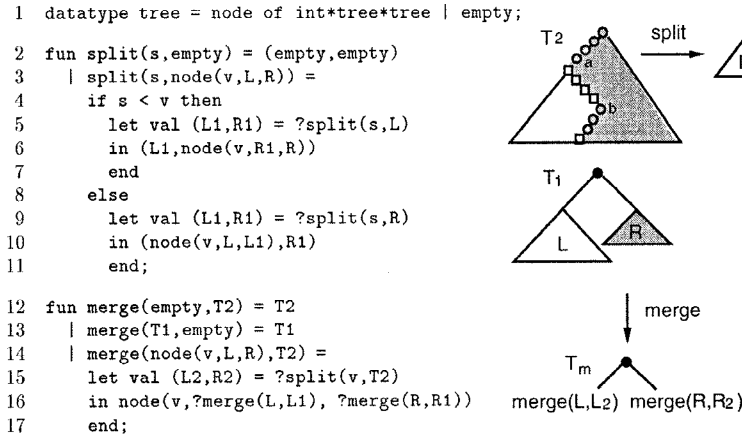
The model, as defined here, is basically the PSL (Parallel Speculative λ-Calculus) [23], augmented with arrays as in NESL [10]. Although the PSL only considered the pure λ-Calculus with arithmetic operations, the syntactic sugar we include affects work and depth by a constant factor only. In this paper we are actually assuming a slightly simplified model by considering only a first-order language (it cannot pass functions) since we do not need the more general case. We also explicitly mark where futures are to be created, while in the PSL model all expressions are implicitly made into futures.

## 3. Pipelining Applications

In this section we show four applications that use pipelining to reduce the depth of the algorithms. The first three applications require a dynamic pipeline because the time at which data becomes available for the next task in the pipeline varies from task to task. The last application is sychronous and the pipeline depth can be fixed. For each application we give the parallel algorithm, explain how to modify the algorithm to pipeline the computation, and give an analysis of the depth.

### 3.1. *Merging Binary Trees*

The first algorithm we discuss is a simple divide-and-conquer algorithm that takes two binary trees $T_1$ and $T_2$, where the keys in each tree are unique and sorted when traversed in-order, and merges them into a new sorted binary tree, $T_m$. The code is shown in Figure 3. The function `split`$(s, T)$ splits a tree $T$ into two trees, one with keys less than the splitter $s$ and one with keys greater than or equal to $s$. The function traverses a path down to a leaf, separating subtrees based on the splitter to form the two result trees



```
1  datatype tree = node of int*tree*tree | empty;
2  fun split(s,empty) = (empty,empty)
3    | split(s,node(v,L,R)) =
4      if s < v then
5        let val (L1,R1) = ?split(s,L)
6        in (L1,node(v,R1,R))
7        end
8      else
9        let val (L1,R1) = ?split(s,R)
10       in (node(v,L,L1),R1)
11       end;
12 fun merge(empty,T2) = T2
13   | merge(T1,empty) = T1
14   | merge(node(v,L,R),T2) =
15     let val (L2,R2) = ?split(v,T2)
16     in node(v,?merge(L,L1), ?merge(R,R1))
17     end;
```



**Fig. 3.** Code for merging two binary search trees and a corresponding figure. The shaded regions are keys that are greater than the key at the root of $T_1$.

(see Figure 3). It requires work that is at most proportional to the depth of the tree. The function `merge` makes the root of $T_1$ the root of the result tree $T_m$ and splits $T_2$ by the key at the root of $T_1$. It then calls `merge` recursively twice to make the left and right subtrees.

The code is a natural sequential implementation for merging two binary trees, if we exclude the futures. Futures provide two forms of parallelism. First, they provide parallelism by allowing the two recursive `merge` functions to execute in parallel. If $T_1$ is balanced and of size $n$, then the `merge` will be called recursively to a depth of $O(\lg n)$. If $T_2$ is also balanced and of size $m$, then the split operation has $O(\lg m)$ depth. Therefore, the overall depth of the algorithm is easily bounded by $O(\lg n \lg m)$. Second, and more importantly for this paper, futures provide pipelining by allowing the partial results of `split` (i.e., nodes higher in the tree) to be fed into the two `merge` calls, thereby allowing for the overlap in time of multiple split calls at different levels of the recursion tree. With such pipelining `merge` has depth $O(\lg n + \lg m)$.

To illustrate how the algorithm pipelines, we consider the time (depth in the DAG) at which all nodes of the result trees, $(L_2, R_2) = \texttt{split}(v, T_2)$, are computed. If the roots of both $L_2$ and $R_2$ are created in constant time, and each child at a constant time after its parent, it is not hard to see that the algorithm would pipeline within $O(\lg n + \lg m)$ depth. The problem, however, is that one root may only be ready after a considerable delay. For example, in Figure 3 the root of $L_2$ is ready only after traversing five nodes in $T_2$. In addition, there may be further delays at lower levels of the tree. For example, there is a delay going from node `a` to node `b` in $R_2$; `b` is created only after four nodes of $L_2$ have been created. In general, the rightmost path of $L_2$ and the leftmost path of $R_2$ are made from the nodes of $T_2$ that `split` traversed, and the time stamp for a node in these paths is proportional to its depth in $T_2$. These delays can accumulate when one split is pipelined into the next. To prove the bounds, however, we show that when there is a delay there is a corresponding decrease in the depth of the result tree.

**Theorem 3.1.** *Merging two balanced binary trees of size $n$ and $m$, $m < n$, with keys sorted in-order takes $O(\lg n + \lg m)$ depth and $O(m \lg(n/m))$ work.*

*Proof.* Given in the next section. It is a simplification of the proof for taking the union of two treaps. □

A problem with the merge algorithm described is that even though the input trees may be balanced the resulting merge tree may have depth up to $\lg n + \lg m$. We now briefly describe how using pipelining, again, the unbalanced result can be balanced with $O(\lg n + \lg m)$ depth and $O(n + m)$ work. First, the algorithm makes a pass through the tree computing the size of every subtree, which it stores at the root of the subtree. From the size data it next finds the rank of each node (its in-order index). Both steps take $O(\lg n + \lg m)$ depth and $O(n + m)$ work and do not require pipelining. Next, it rebalances the tree using a parallel pipelined algorithm similar to `merge`. However, this time it uses a split operation (similar to `splitm` in the next section) that takes a rank argument and splits the tree into nodes with rank less than the argument and nodes with rank greater than the argument. It returns these two trees along with the node with equal rank. The rebalancing algorithm takes four arguments: a tree, a rank, and the number of lesser and the number of greater rank nodes in the tree. It calls this split operation on the tree and the rank. It uses the node returned by the split operation as the root and then recursively balances the two subtrees. The recursive call for the left (right) subtree supplies a rank that is the old rank minus (plus) half the lesser (greater) subtree size. The analysis of the depth of the algorithm is similar to the analysis of `union` in the next section.

### 3.2. *Treap Union*

Treaps [29] are balanced search trees that provide for search, insertion, and deletion of keys and can be used for maintaining a dynamic dictionary. Associated with each key in a treap is a random priority value. The keys are maintained in-order and the priority values are maintained in heap order, thus the name treap. The key with the highest priority is the root of the treap. Because the priorities are random, this key is a randomly chosen key. Similar to quicksort recursion depth, treaps, therefore, have an expected depth of $O(\lg n)$ for a tree with $n$ keys. Treaps have the advantage over other balanced tree techniques in that they allow for simple and efficient union. As we will see, they have the added advantage that it is easy to parallelize them.

We present two pipelined parallel operations on treaps—a *union* operation that takes the union of two treaps and can be used to insert a set of keys into a treap; and a *difference* operation that removes the values in one treap from another and can be used to delete a set of keys. Figure 4 shows the code for finding the union of two treaps. It is similar to merge in the previous section except that it removes any duplicate values and maintains the treap conditions so that the result treap is balanced. It uses a modified `split` operation, `splitm`, where the splitter can be a key in the treap. When the splitter is in the treap, `splitm` excludes it from the resulting treaps and returns it along with the two split treaps. Otherwise, it simply returns the two resulting treaps. Notice that `splitm` completes as soon as it finds the splitter in the treap.

```
1  datatype treap = node of real*int*treap*treap
2                   | empty;
3  datatype midval = mid of int | none;

4  fun splitm(v1,empty) = (empty,none,empty)
5    | splitm(s,node(p,v,l,r)) =
6      if s=v then (l,mid(v),r)
7      else if s<v then
8        let val (l1,m,r1) = ?splitm(s,l)
9        in (l1,m,node(p,v,r1,r))
10       end
11     else
12       let val (l1,m,r1) = ?splitm(s,r)
13       in (node(p,v,l,l1),m,r1)
14       end;

15 fun union(empty,b) = b
16   | union(a,empty) = a
17   | union(node(p1,k1,l1,r1), node(p2,k2,l2,r2)) =
18     if p1 > p2 then
19       let val
20         (l,m,r) = ?splitm(k1,node(p2,k2,l2,r2))
21       in node(p1, k1, ?union(l1,l), ?union(r1,r))
22       end
23     else
24       let val
25         (l,m,r) = ?splitm(k2,node(p1,k1,l1,r1))
26       in node(p2, k2, ?union(l,l2), ?union(r,r2))
27       end;
```

**Fig. 4.** Code for treap union.

To maintain the heap order `union` makes the root with the largest priority the root of the result treap (compare with `merge`, which always uses the root of the first tree). To maintain the keys in-order `union` splits the treaps by the key value of the new root. For one treap these are trivially the left and right children of the root. For the other treap the algorithm uses `splitm` with futures. It then recursively finds the union of the two treaps that have keys less than the root, and finds the union of the two treaps that have keys greater than the root. We show that the expected depth to find the union of two treaps of size $n$ and $m$ is $O(\lg n + \lg m)$. Without pipelining the expected depth would be $O(\lg n \lg m)$.

To analyze the depth of the algorithm we consider time stamps $t(v)$ for each node $v$ of a tree. The *time stamp* of a node is the depth in the DAG at which the node is created. For a tree $T$ we use the notation $v \in T$ to be a node in $T$, $h(v)$ to indicate the height of the subtree rooted at the node $v$ (longest path length to any of its leaves), and $l(v)$ and $r(v)$ to indicate the left and right children of the node $v$, respectively. We use $t(T), h(T), l(T), r(T)$ to mean $t(v), h(v), l(v), r(v)$, respectively, where $v$ is the root of $T$.

**Definition 1.** A $\tau$-value is *valid* if, for all $v \in T$, $t(v) \leq \tau + k_s(h(T) - h(v))$, where $k_s$ is a constant.

A *$\tau$-value* of a tree is some value that places an upper bound on each of the time stamps in the tree depending on the height of the subtree at the node. This definition means that $\tau \geq \max_{v \in T}\{t(v) - k_s(h(T) - h(v))\}$. These $\tau$-values capture a relationship between the height of subtrees and their time stamps which is important for the proofs of our time bounds. Notice, for example, that a $\tau$-value places the same upper bound on the time stamps for all leaves in the tree regardless of how far down they are in the tree. In the following theorem we show that, for each result treap of `splitm`, we can find a valid $\tau$-value that depends only on the result treap height, the input treap height, and the input treap's $\tau$-value. In the analysis of `union` we keep track of the $\tau$-values of the input treaps to recursive calls to bound the time stamps in these treaps.

**Property 3.2.** If $\tau$ is a valid $\tau$-value for a tree $T$, then a valid $\tau$-value for a subtree $T'$ is

$$\tau + k_s(h(T) - h(T')).$$

**Property 3.3.** If $\tau_l$ and $\tau_r$ are valid $\tau$-values for $l(T)$ and $r(T)$, respectively, then a valid $\tau$-value for $T$ is

$$\max\{t(T), \tau_l - k_s, \tau_r - k_s\}.$$

**Lemma 3.4** (`Splitm` $\tau$-Values). *Consider any split value $s$ and any treap $T$ with associated $\tau$-value $\tau$ and let $k_s$ be the time between two successive recursive calls to* `splitm`. *If we call the* `splitm`$(s, T)$ *function at a time $t$, then, for each of the two results $T' \in \{L', R'\}$, a valid $\tau$-value for $T'$ is $\tau' = \max\{t, \tau\} + k_s(1 + h(T) - h(T'))$.*

*Proof.* We assume that the splitter does not appear in the treap since this is the worst case (if the splitter is found, then the split will return earlier). We use induction on the height of the input treap. The lemma is clearly true when $h(T) = 1$. Assume it is true for treaps of height less than or equal to $h - 1$. We show it is true when $h(T) = h$. Let $L = l(T)$ and $R = r(T)$. Without loss of generality, assume that $s$ is less than the key at the root of $T$, and let $(L_1, R_1) =$ `splitm`$(s, L)$ (see Figure 5). First, we find a

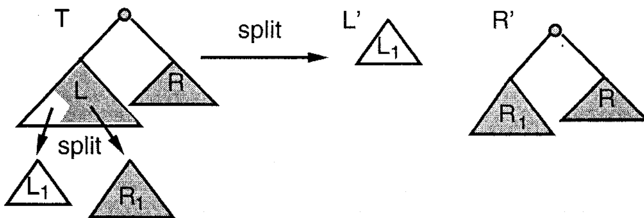



**Fig. 5.** Split of treap $T$ into $L'$ and $R'$. The shaded areas are keys that are greater than the splitter.

valid $\tau$-value for the greater than result treap, $R'$, by finding the time stamps for all its nodes. Consider the root of $R'$. Once the root of $T$ is available `union` can obtain $R$ and $L$, which may be futures, compare the key at the root with $s$, and call `splitm`, which returns immediately since it returns three futures. Thus, `union` has all the information needed to create the root of $R'$ in constant time, $k_s$, and $t(R') = \max\{t, t(T)\} + k_s$. Because $r(R') = R$, a valid $\tau$-value for $r(R')$ is $\tau + k_s(h(T) - h(r(R')))$ by Property 3.2. Next we find upper bounds of the times in $L_1$ and $R_1$.

The recursive call to `splitm` on $L$ can be called at time $\max\{t, \tau\} + k_s$ and, by Property 3.2, a valid $\tau$-value for $L$ is $\tau + k_s(h(T) - h(L))$. Therefore, by the induction hypothesis a valid $\tau$-value $\tau''$ for the resulting treap $T'' \in (L_1, R_1)$ is

$$\begin{aligned}\tau'' &= \max\{\max\{t, \tau\} + k_s, \tau + k_s(h(T) - h(L))\} + k_s(1 + h(L) - h(T''))\\ &\leq \max\{t, \tau\} + k_s(1 + h(T) - h(T'')). \end{aligned} \tag{1}$$

Since $l(R') = R_1$, by Property 3.3, a valid $\tau$-value for $R'$ is

$$\begin{aligned}\tau' = \max\{&\max\{t, t(T)\} + k_s, \tau + k_s(h(T) - h(r(R')) - 1),\\ &\max\{t, \tau\} + k_s(h(T) - h(l(R')))\}\\ \leq\ &\max\{t, \tau\} + k_s(1 + h(T) - h(R')).\end{aligned}$$

Finally, since $L' = L_1$, a $\tau$-value of $L'$ is a $\tau$-value for $L_1$ as given in (1). □

Note that `union` creates new treaps by only dividing a treap into its left and right children or by running the `splitm` operation on it. Given the above lemma, we can find $\tau$-values for the treaps in all the recursive calls, and use these $\tau$-values to find upper bounds $\hat{t}(v)$ for $t(v)$, the time stamps on the nodes $v$ of the union result treap.

**Theorem 3.5** (Depth Bound on Union). *Consider two treaps $T_1$ and $T_2$ with $\tau$-values $\tau_1$ and $\tau_2$. If we call* `union`$(T_1, T_2)$ *at time $t$, then the maximum time stamp on any of the nodes of the result $T_m$ will be* $\max\{t, \tau_1, \tau_2\} + O(h(T_1) + h(T_2))$.

*Proof.* Once the two roots of $T_1$ and $T_2$ are ready, `union` can compare their priorities, start up `splitm` and the two recursive `union`s, and create the root of the result treap $T_m$ with pointers to the futures for its two children. This all takes constant time, $k_m$, because `splitm` and `union` are called with futures. Thus, $t(T_m) \leq k_m + \max\{t, \tau_1, \tau_2\}$. This upper bound $k_m + \max\{t, \tau_1, \tau_2\}$ on the time stamp of the root of the result treap will be referred to as $\hat{t}(T_m)$.

We now calculate $\hat{t}(l(T_m))$, an upper bound on the time stamp of the left child of the root of the result treap, in terms of $\hat{t}(T_m)$. Consider the two treaps $T_1^l$ and $T_2^l$, which are the inputs to the left call to `union`, and $T_m^l = l(T_m)$, which is the result of the call. Without loss of generality consider the case when the priority of $T_1$ is greater than the priority of $T_2$. Then $T_1^l = l(T_1)$ and $T_2^l$ is the left result of `splitm`$(k_1, T_2)$, where $k_1$ is the key at the root of $T_1$, see Figure 6. Due to the previous bound on the `splitm` operation, a $\tau$-value for $T_2^l$ is

$$\begin{aligned}\tau_2^l &= \max\{t(T_m), \tau_2\} + k_s(1 + h(T_2) - h(T_2^l))\\ &\leq \hat{t}(T_m) + k_s(1 + h(T_2) - h(T_2^l))\end{aligned}$$

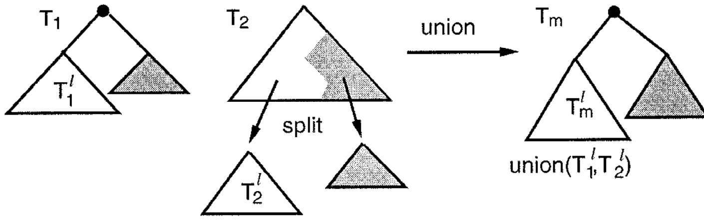



**Fig. 6.** Union of treaps $T_1$ and $T_2$ into $T_m$, when the priority at the root of $T_1$ is greater than the priority at the root of $T_2$. $T_2$ is split by $k_1$, the key at the root of $T_1$. The subtreap $T_m^l$ is the union of the subtreaps with keys less than $k_1$ (not shaded) and the subtreap $T_m^r$ is the union of the subtreaps with keys greater than $k_1$ (shaded).

By Property 3.2, a $\tau$-value for $T_1^l$ is

$$\begin{aligned}\tau_1^l &= \tau_1 + k_s(h(T_1) - h(T_1^l)) \\ &< \hat{t}(T_m) + k_s(h(T_1) - h(T_1^l)).\end{aligned}$$

These, along with the condition at the beginning of the proof, give an upper bound on the time stamp of $T_m^l$:

$$\begin{aligned}t(T_m^l) &\leq k_m + \max\{t(T_m), \tau_1^l, \tau_2^l\} \\ &\leq \hat{t}(T_m) + k_m + k_s \max(h(T_1) - h(T_1^l), 1 + h(T_2) - h(T_2^l)).\end{aligned}$$

That is, the only way the bound on the time stamp of a child can be $k_m + \delta \cdot k_s$ more than its parent's bound is by a corresponding height decrease of either $\delta$ in the depth of $T_1$ or $\delta - 1$ in $T_2$. Because `union` removes the root of $T_1$, $\delta \geq 1$. We can show the same bound for $r(T_m)$.

Now consider a path in $T_m$ from the root to a leaf. Let $\Delta_i = \hat{t}(c) - \hat{t}(v)$, where $c$ is a child of $v$ and $v$ is a node at depth $i - 1$. Let $h_j^i$, $j = 1, 2$, be the height of the input treaps of the union that created $c$. From the above discussion and $j = 1(2)$ and $k = 2(1)$,

$$\begin{aligned}\Delta_i &\leq k_m + k_s \max(h_j^{i-1} - h_j^i, 1 + h_k^{i-1} - h_k^i) \\ &\leq k_m + k_s(h_1^{i-1} - h_1^i + h_2^{i-1} - h_2^i + 1).\end{aligned} \tag{2}$$

Since the algorithm terminates whenever one of the input treaps has height 0, and the height of at least one of the treaps decreases by one for each recursive call, the depth of the recursion treap is at most $O(h(T_1)+h(T_2))$. Therefore, the total increase in the bound on the time stamps along the path to any new node is $\sum \Delta_i \leq (k_m + 2k_s)(h(T_1) + h(T_2))$. Since the time stamp on the root is bound by $k_m + \max\{t, \tau_1, \tau_2\}$ and the path bound is true for all paths, this bounds the time stamp on any new node in $T_m$ by $\max\{t, \tau_1, \tau_2\} + O(h(T_1) + h(T_2))$. The untouched nodes are also clearly similarly bounded. □

**Corollary 3.6** (Expected Union Depth). *The expected depth to find the union two treaps of size n and m is* $O(\lg n + \lg m)$.

*Proof.* We assume that the treaps are "ready" when `union` is called at time $t$. That is, the treaps have valid $\tau$-values, $\tau_1$ and $\tau_2$, with $\tau_1 < t$ and $\tau_2 < t$. Since the expected heights of the two treaps is $O(\lg n)$ and $O(\lg m)$ [29], the expected depth to find the union is $O(\lg n + \lg m)$. □

**Theorem 3.7.** *The expected work to meld two treaps of size $n$ and $m, m < n$, is $O(m \lg(n/m))$.*

*Proof.* See [11]. □

We now return to the proof of depth on the merge computation described in the previous section.

*Proof of Theorem* 3.1. The proof for the depth bound on merge is the same as for the depth bound on union, except that we do not need to consider the case when $T_1$ is split. Thus, in (2), $j = 1$ and $k = 2$. Since $h(T_1) = \lg n$ and $h(T_2) = \lg m$, to merge the two trees takes $O(\lg n + \lg m)$ depth. The proof for the work bound for merge is easier than for union because the input trees are balanced. Union requires an expected case analysis. □

### 3.3. *Treap Difference*

The inverse operation to taking the union of two treaps is taking their difference; remove any keys from the first treap that appear in the second treap. The `diff` algorithm is, again, quite simple and uses two operations `splitm` (shown previously in Figure 4) and `join` (shown in Figure 7). The `join` operation is the inverse of `split`—it takes two treaps, $T_1$ and $T_2$, where the largest key in $T_1$ is less than the smallest key in $T_2$, and joins them into a single treap, $T'$. A join only requires $O(h(T_1) + h(T_2))$ work since it need only descend the rightmost path of $T_1$ and the leftmost path of $T_2$, interleaving the nodes depending on their associated priorities.

The function `diff` takes two treaps, $T_1$ and $T_2$, and returns a treap $T_d$ which is $T_1$ with any keys in $T_2$ removed. First, it calls `splitm` on $T_2$ and the key at the root of $T_1$ as the splitter to obtain two treaps, $l_2$ and $r_2$, and possibly the splitter. Next, `diff` recursively finds the difference of $l(T_1)$ and $l_2$ and the difference of $r(T_1)$ and $r_2$. If the root key of $T_1$ was not in $T_2$ the results of the recursive calls become the left and right branches of the root. Otherwise, the root and its subtreap is replaced by the join of the two treaps resulting from the recursive calls. As in `union`, without pipelining it takes $O(h(T_1)h(T_2))$ depth to descend to the bottom of the recursion call tree. On the way back up, a path may contain as many as $\min(h(T_1), m)$ nodes to delete, where $m$ is the size of $T_2$. Each such node can add $O(h(T_d))$ depth due to the required `join`. Thus, the overall depth for `diff` not considering pipelining is $O((h(T_1)h(T_2) + h(T_d)\min(h(T_1), m))$.

The pipelining for `diff` is notably different from the pipelining for `union` because the algorithm requires work after the recursive calls (the join) as well as before them (the split). The pipelining while descending $T_1$ is much like the tree `merge`, except no actual merging takes place and, therefore, that part of the computation DAG has $O(h(T_1) + h(T_2))$ depth. We next show that the ascending phase of the algorithm takes

```
 1  fun join(empty,b) = b
 2    | join(a,empty) = a
 3    | join(node(p1,k1,l1,r1),
 4           node(p2,k2,l2,r2)) =
 5      if p1 > p2 then
 6        node(p1,k1,l1,?join(r1,node(p2,k2,l2,r2)))
 7      else
 8        node(p2,k2,?join(node(p1,k1,l1,r1),l2),r2);

 9  fun diff(empty,b) = empty
10    | diff(a,empty) = a
11    | diff(node(p1,k1,l1,r1),t2) =
12      let val (l2,m,r2) = ?splitm(k1,t2);
13          val l = ?diff(l1,l2);
14          val r = ?diff(r1,r2);
15      in if m = none then node(p1,k1,l,r)
16         else ?join(l,r)
17      end;
```

**Fig. 7.** Code for taking the difference of two treaps.

$O(h(T_1) + h(T_d))$ depth. First we show the worse-case time stamps on the results of a `join`. Then we show the worse case time stamps on the final result treap. We use the same definitions as in Section 3.2, except we replace $\tau$-values with a similar concept of $\rho$-values.

**Definition 2.** Let $d_T(v)$ of a node $v \in T$ be the depth of the node in the tree, such that the $d_T(T) = 0$, $d_T(l(T)) = d_T(r(T)) = 1, \ldots$. A $\rho$-value is *valid* for a tree $T$ if, for all $v \in T$, $t(v) \leq \rho + kd_T(v)$, where $k$ is a constant.

That is, a *valid $\rho$-value* for a tree $T$ defines upper bounds for the time stamps of the tree, namely for all $v \in T$, $t(v) \leq \rho + kd_T(v)$, where $k$ is a constant. In contrast to $\tau$-values, $\rho$-values are independent of the heights of the subtrees.

**Property 3.8.** If $\rho$ is a valid $\rho$-value for $T$, then $\rho$ is a valid $\tau$-value for $T$.

**Property 3.9.** If $\tau$ is a valid $\tau$-value for $T$, then $\tau + kh(T) - 2$ is a valid $\rho$-value for $T$.

**Lemma 3.10** (`join` $\rho$-Values). *If* `join` *is called at time $t$ on two treaps $T_1$ and $T_2$ with valid $\rho$-values $\rho_1$ and $\rho_2$, then a valid $\rho$-value for the resulting joined treap $T'$ is $\rho' = \max\{t, \rho_1, \rho_2\} + k$, where $k$ is a constant at least as large as the maximum computation DAG depth between successive recursive calls to* `join`.

*Proof.* We find upper bounds of the time stamps of each node of the $T'$ by induction on the size of $T'$. Let $n$ be the size of $T'$. The lemma is clearly true when the size of the result treap is 1. Assume it is true for result treaps of size $n - 1$. We show it is true for result treaps of size $n$. Since `join` can test the root priorities, receive a pointer to the future

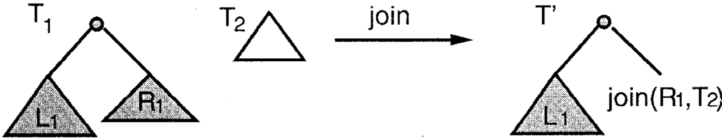



**Fig. 8.** Join of treaps $T_1$ and $T_2$ into $T'$, when the priority at the root of $T_1$ is greater than the priority at the root of $T_2$.

which is the result of the recursive call to `join`, and create the root node of $T'$ in constant depth $k$, once the roots of $T_1$ and $T_2$ are ready, $t(T') = \max\{t, \rho_1, \rho_2\} + k$. Call this value $\rho'$. Without loss of generality, assume that the priority of the root of $T_1$ is greater than the priority of the root of $T_2$ (see Figure 8). Because $l(T_1) = l(T')$, then, for all $v \in l(T')$, $t(v) \leq \rho_1 + kd_{T_1}(v) \leq \rho' + kd_{T'}(v)$, since the depth of $v$ is the same in $T_1$ as in $T'$. By the induction hypothesis we can find the time stamps on $r(T') =$ `join`$(r(T_1), T_2)$, since the size of $r(T_1)$ is less than $n$. A valid $\rho$-value for $r(T_1)$ is $\rho_1 + k$. Therefore a valid $\rho$-value for $r(T')$ is $\max\{\rho', \rho_1 + k, \rho_2\} + k = \rho' + k$. Since $v$'s depth in $r(T')$ is one less than its depth in $T'$, $t(v) \leq \rho' + kd_{T'}(v)$ for all $v \in r(T')$. Thus, $\rho'$ is a valid $\rho$-value for $T'$. □

**Theorem 3.11** (Bound on Difference Depth). *If* `diff`$(T_1, T_2)$ *is called at time t and valid $\rho$-values for $T_1$ and $T_2$ are $\rho_1$ and $\rho_2$, then the maximum time stamp on the result treap $T_d$ is* $\max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2) + h(T_d))$.

*Proof.* Let $k$ be a constant greater than the maximum computational DAG depth between successive recursive calls to `splitm`, `join`, and `diff`. Since $\rho_1$ and $\rho_2$ are valid $\tau$-values for $T_1$ and $T_2$, by Property 3.8, and using the same arguments as in Theorem 3.5, after $\max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2))$ depth in the computation DAG, `diff` has reached the bottom of every recursive path (either lines 9 or 10 in Figure 7 applies) and every future result of `splitm` has been computed. Thus, by Property 3.9 there exists a constant $\rho' = \max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2))$ which is a valid $\rho$-value for all trees (treaps `l` and `r` on lines 13 and 14) that are the result of these calls at the leaves of the call tree. At this point we can find $\rho$-values for the results of each recursive call to `diff`. Let $\rho_l$ and $\rho_r$ be valid $\rho$-values for the results treaps `l` and `r`. Because the recursive calls to `diff` are called with futures, the call to `join` is always made by $\max\{\rho_l, \rho_r\}$. By Lemma 3.10 a valid $\rho$-value for result of the `diff` recursive call is $\max\{\rho_l, \rho_r\} + k$ (compare with the definition of the height of a tree). However, since all the result treaps at the leaves of the recursive call tree have $\rho'$ as a valid $\rho$-value and the height of the recursive call tree is no more than $h(T_1)$, a valid $\rho$-value for the treap at the root of the call tree must be $\rho' + kh(T_1)$. By definition of $\rho$-values, the time stamp of the deepest node in that treap is $\rho' + O(h(T_1) + h(T_d)) = \max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2) + h(T_d))$. □

**Corollary 3.12** (Expected Difference Depth). *The expected depth to find the difference of two treaps of size n and m is* $O(\lg n + \lg m)$.

*Proof.* Since the expected height of the two input treaps are $O(\lg n)$ and $O(\lg m)$ and

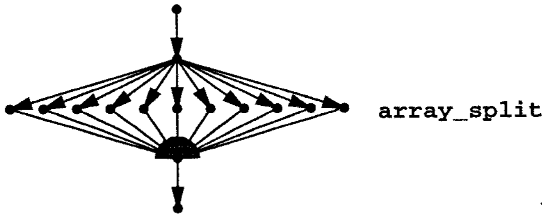



**Fig. 9.** The DAG for an `array_split` on an array of length 11.

the expected height of the result treap is $O(\lg(n-m))$, the expected depth to find the difference is $O(\lg n + \lg m)$. □

### 3.4. 2-6 *Trees*

We can obtain a pipelined variant of top-down 2-3-4 trees using 2-6 trees. It is analogous to the bottom-up pipelined 2-3 trees of PVW [28]. Each node of a 2-6 tree has one to five keys in increasing value and one child for each range defined by the keys. The children are 2-6 trees with key values within their range. Every key appears only once, either in internal nodes or at the leaves, and all leaves are at the same level. We refer to the keys in the tree as *splitters*.

We consider the problem of inserting a set of sorted keys into a 2-6 tree. For this problem we use an array primitive `array_split`, which splits a sorted array of size $m$ into two arrays, one with values less than the splitter and one with values greater than the splitter. In our cost model we define this operation to have $O(1)$ depth and $O(m)$ work—in the DAG we view the operation as a DAG of depth 2 and breadth $m$ (see Figure 9).[2] First we consider inserting an ordered set of keys in which there is at least one key in the 2-6 tree between each pair of keys to be inserted. We call such an array a *well-separated* key array. Later, we show how to insert any ordered set of keys.

If the root of the 2-6 tree has more than three children, the algorithm `insert` splits the root into two 2-3 nodes (nodes with two or three children) and creates a new root using the "middle" splitter and these new 2-3 nodes as children. From now on `insert` maintains the invariant that the root of the tree into which it is inserting is a 2-3 node. It does so by always splitting any child, as necessary, before applying a recursive call on that child. Every time it splits a child it needs to include one of the child's splitters into the root. However, since the root has at most two splitters and three children (by the invariant), the resulting root will have at most five splitters and six children.

To insert an ordered well-separated key array, `insert` first splits the keys by the smallest splitter at the root into two arrays using the `array_split` primitive. It will insert the first of the two arrays into the left child. If there is no second splitter, it will insert the second key array into the right child. Otherwise, it splits the second array by the second splitter and will insert the resulting key arrays into the middle and right children.

---

[2] The reader might argue that the split operation should have depth greater than $O(1)$ because of the need to collect the two sets of values. We show in Section 4, however, that the cost of the `array_split` is fully accounted for in the implementation.

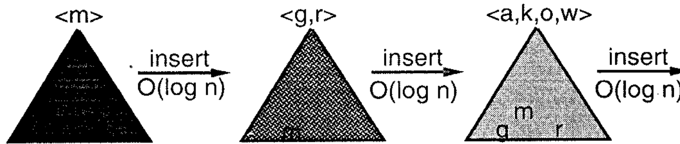



**Fig. 10.** Inserting an ordered set of keys into a 2-6 tree of size $n$. The array (items enclosed in angle brackets $\langle\rangle$) at the root of a tree is the well-separated key array to be inserted in the tree. First insert the median `<m>` into the tree (dark shading). Next insert the first and third quartile `<g,r>` into the resulting tree (medium shading). Then insert the next well-separated array into the next resulting tree (light shading) and so on. Inserting each well-separated key array takes $O(\lg n)$ depth.

Before recursively inserting a key array into a child, `insert` first checks whether the child needs to be split to maintain the 2-3 root node invariant. When a child is split, it obtains a new splitter and two new children. It uses the new splitter to split the key arrays into two arrays that it will insert into the two new children. Next it recursively inserts the key arrays into the appropriate children to obtain new children for the root. Eventually, `insert` will reach a leaf node, which must be a 2-3 node by the invariant. Because of the requirement that there is always at least one key in the 2-6 tree between each key to be inserted, there can be at most three keys that need to be inserted in any one leaf; these keys can be included in the leaf without having to split the node. Note that the height of the tree increases by at most one, when the root of the tree was split.

If `insert` uses futures when making its recursive calls, then it traverses the different paths down the tree in parallel by forking off new tasks for each recursive call. Since the paths are at most $\lg n$ long, inserting an ordered well-separated key array of size $m$ into a 2-6 tree of size $n$ takes $O(\lg n)$ depth and $O(m \lg n)$ work. No pipelining is needed.

To insert an arbitrary ordered set of keys of size $m$, `insert` first forms a balanced binary tree of the keys (conceptually), and then creates a list of arrays of keys, where each array is made up of the keys from one level of the tree. Thus, the first array contains the median key, the next array contains the first and third quartiles, and so on. It then successively inserts each array into the 2-6 tree using the tree returned by the previous insertion, see Figure 10. By inserting the keys in this manner, `insert` guarantees that for any array of keys, there is at least one key in the 2-6 tree between each pair of keys in the array, because it has inserted such keys previously. Without pipelining, inserting the $\lg m$ arrays into a tree of size $n$ would require $O(\lg n \lg m)$ depth and $O(m \lg n)$ work.

By simply making the recursive call that inserts a well-separated key array return a future (in addition to the futures used in its recursive calls), `insert` can pipeline inserting each array of keys into the 2-6 tree—no other changes to the code need to be made. The crucial fact that makes the pipelining work is that, in constant depth, `insert` can return the root node with its keys values filled in, although its children may be futures, see Figure 11. It can then insert the next well-separated key array in the list into this new root, which is the root of the 2-6 tree that will eventually contain the original and previous well-separated key arrays. With this structural information in the root the next insertion can also return the root in constant depth. Although it may need to wait a constant depth before the children nodes are ready, from then on the children of all descendants will be ready when it reaches them. In this way there can be an array of keys being inserted at every second level and possibly every level of the 2-6 tree.

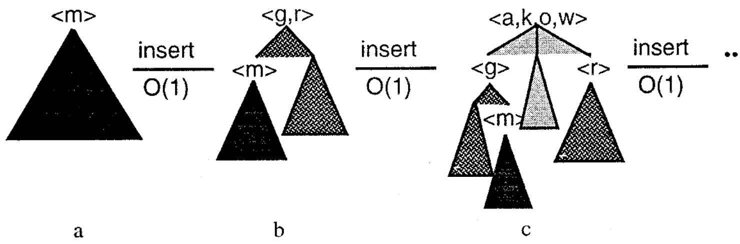



**Fig. 11.** Inserting an ordered set of keys into a 2-6 tree of size $n$ using pipelining. An array (items enclosed in angle brackets $\langle\rangle$) at the root of a tree is the well-separated key array to be inserted in the tree and refers to a future in the computation. (a) First, the median `<m>` is inserted into the original tree (dark shading). (b) As soon as the root node of the resulting tree is ready (medium shading), the first and third quartile `<g,r>` are inserted into it. The root is ready in $O(1)$ depth. The median `<m>` still needs to be inserted into a child of the original tree root (dark shading), the result of which is a future. When the future value is available it becomes a child in the second result tree (medium shading). (c) The next well-separated array is inserted into the next resulting tree (light shading) and so on.

**Definition 3.** $\gamma$ is a valid $\gamma$-value for a 2-6 tree $T$ if, for all $v \in T$, $t(v) \leq \gamma + k_b d_T(v)$, where $t(v)$ is the time stamp for $v$, $d_T(v)$ is the depth of $v$ in $T$, and $k_b$ is constant.

**Theorem 3.13** (Insertion into a 2-6 Tree). *A set of $m$ ordered keys can be inserted in a* 2-6 *tree of size $n > m$ in $O(\lg n + \lg m)$ depth and $O(m \lg n)$ work.*

*Proof.* First note that we can create a pipeline of well-separated key arrays from an arbitrary array of sorted keys. Each successive well-separated key array can be found in constant time, $k_w$, given the indices of the keys that made up the previous key array. That is, the time stamp for the $i$th key array is $k_w \cdot i$. Let $T_0$ be the original 2-6 tree we are inserting into, and let $\gamma_0$ be its associated valid $\gamma$-value. Let $T_i$ be the resulting 2-6 tree after inserting the $i$th well-separated key array into $T_0$. We will show that

$$\gamma_{i+1} = \gamma_i + 3k_b \tag{3}$$

are valid $\gamma$-values for $T_{i+1}$, $i = 0, \ldots, \lg m$.[3]

Assume $\gamma_i$ is a valid $\gamma$-value for $T_i$ and $k_b$ is large enough such that $\gamma_i > (i+1)k_w$. The `insert` function can start to insert the $(i+1)$st well-separated array once both it and the root of $T_i$ are available; that is, at time $\min((i+1)k_w, \gamma_i) = \gamma_i$. In the worse case the root of $T_i$ needs to be split. It can do so in constant depth $k_r$, since it has all the structural information it needs to create the new root and its two children. Again we assume $k_b$ is large enough such that $k_b > k_r$. This splitting results in a new intermediate tree $T_i'$, with a valid $\gamma$-value $\gamma_i + k_b$. By induction on $d$ we will find upper bounds on the time stamps of nodes at depth $d$ of $T_{i+1}$.

First we find $t(T_{i+1})$. Once the root of $T_i'$ or $T_i$ and its children are available `insert` can do all the work necessary to create the root of $T_{i+1}$. These nodes have time stamps at most $\gamma_i + 2k_b$. Then, in constant depth, `insert` can split the keys, determine which

[3] It is also possible to show that $\gamma_{i+1} = \gamma_i + 2k_b$

children need to be split, determine any new keys and children that need to be added to the root of $T_i'$, split the key arrays by the new keys, and proceed with the recursive calls, which return futures to the children of the new root. Let this constant depth be $k_b$. Thus, it has the structural information needed to create and return the root so that $t(T_{i+1}) = \gamma_i + 3k_b$. The recursive calls on nodes at depth 1 of $T_i'$ are made by $\gamma_i + 3k_b$ and these nodes and their children have time stamps no more than that. Therefore, by $\gamma + 4k_b$ it can create the nodes at depth 1 of $T_{i+1}$ and proceed with the recursive calls on nodes at depth 2. In general, the recursive call on nodes at depth $d$ occur by $\gamma_i + (d+2)k_b$ and the nodes of $T_i'$ at level $d$ and level $d + 1$ are also available at that time. Thus, the time stamps for a node at level $d$ of $T_{i+1}$ is at most $\gamma_i + (d + 3)k_b$, proving (3) holds. Since there are $\lg m$ well-separated key arrays, the final 2-6 tree has a valid $\gamma$-value $\gamma + O(\lg m)$ and the tree has depth $O(\lg(m + n))$. Therefore, the largest time stamp is no more than $\gamma + O(\lg m + \lg n)$.

It is easy to see that inserting $m$ keys into a tree of size $n$ using the above algorithm does no more work, within constants, than inserting the $m$ keys one at a time. Since the latter takes $O(m \lg n)$ work so does the former. □

## 4. Implementation

In this section we describe an implementation of futures and give provable bounds on the runtime of computations based on this implementation. The bounds include all costs for handling the suspension and reactivation of threads required by the futures and the cost of scheduling threads on processors. The implementation is an extension of the implementation described in [23] which allows us to improve the time bounds and avoid concurrent memory access.

The main idea of the implementation is to maintain a set of active threads $S$, and to execute a sequence of steps repeatedly, each of which takes some threads from $S$, executes some work on each, and returns some threads to $S$. The interesting part of the implementation is handling the suspension and reactivation of threads due to reading and writing to future cells. As suggested for the implementation of Multilisp [24], a queue can be associated with each future cell so that when a thread suspends waiting for a write on that cell, it is added to the queue, and when the write occurs, all the threads on the associated queue are returned to the active set $S$. Since multiple threads could suspend on a single cell on any given time step, the implementation needs to be able to add the threads to a queue in parallel. Previous work [23] has shown that by using dynamically growing arrays to implement the queues in parallel, any computation with $w$ work and $d$ depth will run in $O(w/p + d \cdot T_f(p))$ time on a CRCW PRAM, where $T_f(p)$ is the latency of a work-efficient fetch-and-add operation on $p$ processors.

By placing a restriction on the code called linearity, we can guarantee that every future cell is read at most once so that only a single thread will ever need to be queued on a future cell. This greatly simplifies the implementation and allows us to replace the fetch-and-add with a scan operation. A further important advantage of linearity is that it guarantees that the implementation only uses exclusive reads and writes to shared memory. The linearity restriction is such that any code can easily be converted to be linear, although this can come at the cost of increasing the work or depth of an algorithm.

```
1  datatype tree = node of int*tree*tree | leaf;

2  fun split(s,leaf) = (leaf,leaf)
3    | split(s,node(v,L,R)) =
4    let val (sa,sb) = copy(s);
5        val (va,vb) = copy(v);
6    in if sa < va then
7         let val (L1,R1) = ?split(sb,L)
8         in (L1,node(vb,R1,R))
9         end
10      else
11        let val (L1,R1) = ?split(sb,R)
12        in (node(vb,L,L1),R1)
13        end;
14   end;
```

**Fig. 12.** Linearized code for splitting two binary trees. Two copies of `s` and `v` are made so that no variable is referenced more than once. A variable that is referenced once in the `then` clause and once in the `else` clause of an `if` statement is referenced once overall because only one of the two clauses is executed. Similarly a variable must be referenced at most once in each function body.

The linearity restriction on code is based on ideas from linear logic [22]. In the context of this paper linearizing code implies that whenever a variable is referenced more than once in the code a copy is made implicitly for each use [26]. The copy must be a so-called deep copy, which copies the full structure (e.g., if a variable refers to a list, the full list must be copied, not just the head).[4] Linearized code has the property that at any time every value can only have a single pointer to it [26]. This implies that there can only be a single pointer to a future cell and it can therefore only be read from once. Similarly it implies that there can only be exclusive read access to any value, even if it is not a future cell. Linear code has been studied extensively in the programming language community in the context of various memory optimizations, such as updating functional data in place or simplifying memory management [26], [31], [4], [1], [18].

Linearizing code does not affect the performance of any of the algorithms we considered in this paper. For example, consider the body of the *split* code in Figure 3, lines 4–11. Figure 12 shows the linearized version of the same code. The only variables that are read more than once refer to keys and splitters ($v$ and $s$). Since it is no more expensive to copy $v$ and $s$ than to compare them, such copying does not affect the costs. The trees themselves are never referenced more than once—although, $L$ and $R$ appear once each in the `then` or the `else` part of the `if` statement, only one of these branches can be executed. The trees $L_1$ and $R_1$ appear twice in both `then` and `else` parts, but one case is simply defining them (lines 7 and 11) while the other actually references them (lines 8 and 12).

We now consider the main result of this section. Here we state the bounds in terms of the EREW scan model [6], which is the EREW extended with a unit-time plus-scan

[4] Note that to copy the structure, the copy must be strict on the full structure—all futures must be written before they can be copied.

(all-prefix-sums) operation. The bounds we prove on the scan model imply bounds of $O(w/p + d \lg p)$ time on the plain EREW PRAM, $O(gw/p + d(T_s + L))$ on the BSP [30], and $O(w/p + d \lg p)$ on an asynchronous EREW PRAM [20] using standard simulations.

**Lemma 4.1** (Implementation of Futures). *Any linearized future-based computation with $w$ work and $d$ depth can be simulated on an EREW scan model in $O(w/p + d)$ time.*

*Proof.* In the following discussion we say that an action (node in the computation DAG) is *ready* if all its parents have been executed and that a thread is *active* if one of its actions is ready. We store threads as *closures*, which are fixed-sized structures containing a code pointer and pointers to a constant number of local variables. We store each future cell as a structure that holds a flag and a pointer. Initially the flag is unset; when the pointer is filled the flag is set. The pointer points to either a value or a suspended thread (i.e., its closure).

We store the set of active threads in an array $S$. The algorithm takes a sequence of steps, where each step takes $m = \min\{|S|, p\}$ threads from $S$, executes one action on each thread, and returns the resulting active threads to $S$. We treat the array $S$ as a stack so that threads are removed from and added to the top of the stack. Let $t$ be the stack top such that the active threads are stored in $S[0], S[1], \ldots, S[t]$.

To take threads from $S$:

1. Remove $m$ threads from the top of $S$. That is, processor $i$ takes thread $S[t - i]$, unless $t - i < 0$, in which case it does nothing on this step.
2. Decrement the stack top by $m$ ($t = t - m$).

The above operations take constant time.

Next we show that each action takes constant time. After executing one action, each thread can return zero, one, or two active threads to $S$ (zero if it terminates or suspends, one if it continues, and two if it forks or reactivates another thread).

1. If a thread with a read pointer to a future cell wants to read the future, then
   - if the future cell has been set, then dereference the pointer (return one thread),
   - otherwise set the flag, write a pointer to the thread's closure into the future cell, and suspend (return zero threads).
2. If a thread with a write pointer to a future cell wants to write a result, then
   - if the future cell has been set, then read the future cell, which has a pointer to the closure of the thread suspended on that cell, write a pointer to the result into the future cell, and reactivate the suspended thread (return two threads),
   - otherwise write a pointer to the result into the future cell and set the future cell's flag (return one thread).
3. If a thread wants to fork a new thread, then
   (a) create a closure for the forked thread,
   (b) create future cells for each result to be returned by the forked thread,
   (c) write pointers to the future cells in the forking thread's closure (for reading) and the forked thread's closure (for writing), and
   (d) activate the forked thread (return two threads)

4. Otherwise execute the action (return one thread if it continues and zero threads if the thread terminates).

To prevent both the writer and reader from accessing the flag concurrently we can assign even steps to the reader and odd steps to the writer.[5] Thus, reading from and writing to a future cell takes constant time; forking a new thread takes constant time because closures are fixed sized and the number of new future cells created is contant; and by definition of the DAG in our model, actions not involving a future cell or forking take constant time.

To return active threads to $S$:

1. Compute the plus-scan of the number of active threads each processor returns.
2. If a processor receives scan result $j$, then it places its zero, one, or two active threads on $S$ starting at $S[t + j + 1]$.
3. Increment the top of the stack by $k$, the total number of threads added to $S$ ($t = t + k$).

Since each processor has at most two threads to return to $S$, the implementation can place the threads back in $S$ in constant time using the unit-time plus-scan primitive assumed in the machine model. The above assumes that unbounded space is allocated for $S$. It is possible to allocate bounded space for $S$, in the same manner as in [23], and still place threads back on $S$ in constant (amortized) time.

In summary, since the algorithm can remove $\min\{|S|, p\}$ threads from the top of $S$ in constant time, can execute one action of each thread in constant time, and can place resulting active threads back on $S$ in constant time, the whole step takes constant time. Since, on each step, the implementation processes $\min\{|S|, p\}$ threads, and $S$ holds all the active threads (by definition), the implementation executes a greedy schedule of the computation DAG. The number of steps is therefore bounded by $w/p + d$ [12] and the total time by $O(w/p+d)$. Note that for the time bounds it does not matter which threads are taken from $S$ on each step, allowing the implementation some freedom in selecting a schedule that is space or communication efficient. The stack discipline we describe above, however, is probably much better for space than a queue discipline.

We now outline how to handle the `array_split` operation used in the 2-6 trees. We first consider implementing a simpler `array_scan` which, given an array of integers of length $n$, returns the plus_scan of the array in $O(n)$ work and $O(1)$ depth (remember that $n$ could be much larger than $p$). As with the `array_split` we account for the cost of the `array_scan` in our cost model as a DAG of depth 2 and breadth $n$. When coming to an `array_scan` in the code the implementation spawns $n$ threads and places them in the set of active threads. Since creating $n$ threads could take more than constant time on $p$ processors, they are created lazily using a stub as described in [8]—threads are expanded when taken from $S$ instead of when inserted. For each block of $p$ or less threads that are scheduled from the set in a particular step, we can use the unit-time scan primitive assumed in the machine model to execute the scan across that subset and place the new running sum back into the stub. When the last thread finishes, it reactivates the parent thread and the scan is complete. If we associate each created thread as a node

[5] A test-and-set operation will suffice, but we do not have such an operation in an EREW PRAM.

in the breadth $n$ DAG, then each node of this DAG can be executed in constant work, and the sink node (bottom node of the $2 \times n$ DAG) is ready as soon as the last thread is done. Since the schedule remains greedy (on each step the implementation always schedules $\min\{|S|, p\}$ threads), the number of steps is bounded by $O(w/p + d)$, where $w$ is now the total number of nodes in the DAG including the expanded DAGs for each `array_scan` (i.e., we are including $O(n)$ work for each `array_scan`). Each step of the scheduling algorithm still takes constant time so the total time on the EREW scan model is also bound by $O(w/p + d)$.

The `array_split` can be implemented by broadcasting the pivot, comparing the array elements to it, executing two scans to determine the final locations of the array elements, and writing the values to these locations (see [6] for example). Each step can be implemented with $O(n)$ work and $O(1)$ depth in a similar way as described above. □

## 5. Conclusions

This paper suggests an approach for designing and analyzing pipelined parallel algorithms using futures. The approach is based on working with an abstract language-based cost model that hides the implementation of futures from the user. Universal bounds for implementing the model are then shown separately.

The main advantages of our approach over pipelining by hand is that it leaves the management of pipelining to the runtime system, greatly simplifying the code. The code we gave for merging and for treaps is indeed very simple, and is just the obvious sequential code with future annotations added in a few places. We expect that it would be very messy to pipeline the treaps by hand because of the unbalanced and dynamic nature of the tree structures. In particular, the depth at which subtrees returned by the `split` function become available is data dependent, and to maintain the depth bounds an implementation must start the next computation as soon as a node becomes available. The immediate reawakening of suspended tasks is therefore a critical part of any implementation. Our code for the 2-6 trees is somewhat more complicated, but still significantly simpler than a version in which the pipelining is done by hand.

Another important advantage of the approach is that it gives more flexibility to the implementation to generate efficient schedules. The algorithms of Cole and PVW specify a very rigorous and synchronous schedule for pipelining while the specification of pipelining using futures is much more asynchronous—the only synchronization is through the future cells themselves and there is no specification in the algorithms of what happens on what step. This gives freedom to the implementation as to how to schedule the tasks. The implementation, for example, could optimize the schedule for either space efficiency [12], [8], [9] or locality [13]. On a uniprocessor the implementation could run the code in a purely sequential mode without any need for synchronization.

We are not yet sure how general the approach is. We have not been able to show, for example, whether the method can be used to generate a sort that has depth $O(\lg n)$. We conjecture that a simple mergesort based on the merge in Section 3.1 has expected depth (averaged over all possible input orderings) close to $O(\lg n)$, perhaps $O(\lg n \lg\lg n)$. This algorithm has three levels of pipelining (i.e., has depth $O(\lg^3 n)$ without pipelining).

This paper is part of our larger research theme of studying language-based cost

models, as opposed to machine-based models, and is an extension of our work on the NESL programming language and its corresponding cost model based on work and depth (summarized in [7]).

## Acknowledgments

We would like to thank Jonathan Hardwick and Girija Narlikar for looking over drafts of this paper and making several useful comments. We would also like to thank Bob Harper for pointing out the connection of linear logic to our attempts to impose a language restriction that would permit a simple EREW implementation.

## Appendix. ML Code

All code in this paper is a subset of ML [27] augmented with future notation, a question mark (`?`). The syntax we use is summarized in Figure 13. The `LET VAR` *pattern* `=` *exp* `IN` *exp* `END` notation is used to define local variables and is similar to Let in Lisp. The `DATATYPE` notation is used to define recursive structures. For example, the notation

```
datatype tree = node of int*tree*tree | leaf;
```

is used to define a datatype called `tree` which can either be a `node` with three fields (an integer, and two trees), or a leaf.

```
defn     ::= FUN body [ | body ]*;          function def'n
         ::= DATATYPE name = sumtype;       datatype def'n

body     ::= name pattern = exp             function body

exp      ::= const                          constant
             name                           variable
             IF exp THEN exp ELSE exp       conditional
             LET VAR pattern = exp          local bindings
               IN exp END
             name (exp,...)                 fn application
             exp binop exp                  binary op
             ( exp )                        paren expr'n
             ?  exp                         future

pattern  ::= name                           var or datatype
             pattern,pattern                tuple
             name(pattern)                  datatype
             ( pattern )                    paren pattern

sumtype  ::= name [OF prodtype]             sum type
               [ | sumtype]

prodtype ::= name [* prodtype]              product type
```

**Fig. 13.** The ML syntax used in this paper.

Pattern matching is used both for pulling datatypes apart into their components (e.g., separating a list into its head and tail) and for branching based on the subtype. For example, in the pattern:

```
fun merge(leaf,B) = B
  | merge(A,leaf) = A
  | merge(node(v,L,R),B) = .....
```

the code first checks if the first argument is a `leaf` type, and returns `B` if it is, it then checks if the second argument is a `leaf` type, and returns `A` if it is, otherwise it pulls the first argument, which must be a `node` into its three components (the integer `v` and the two subtrees `L` and `R`) and executes the remaining code.

## References


[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(n \lg n)$ sorting network. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 1–9, Apr. 1983.

[3] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM Journal of Computing*, 18(3):499–532, June 1989.

[4] H. Baker. Lively linear lisp—"Look Ma, no garbage!" *ACM SIGPLAN Notices*, 27(8):89–98, Aug. 1992.

[5] H. G. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, Aug. 1977.

[6] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, Nov. 1989.

[7] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, Mar. 1996.

[8] G. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[9] G. Blelloch, P. Gibbons, Y. Matias, and G. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23, June 1997.

[10] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[11] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 16–26, June 1998.

[12] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 362–371, May 1993.

[13] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Nov. 1994.

[14] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–206, Apr. 1974.

[15] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In D. Padua, D. Gelernter, and A. Nicolau, editors, *Languages and Compilers for Parallel Comput-*

*ing*, Research Monographs in Parallel and Distributed Computing, pages 95–113. MIT Press, Cambridge, MA, 1990.

[16] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with OLDEN (parallel programming). In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Springer-Verlag, New York, Aug. 1993.

[17] R. Chandra, A. Gupta, and J. Hennessy. COOL: a language for parallel programming. In D. Padua, D. Gelernter, and A. Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 126–148. MIT Press, Cambridge, MA, 1990.

[18] J. L. Chirimar, C. A. Gunter, and J. G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, Mar. 1996.

[19] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, Aug. 1988.

[20] R. Cole and O. Zajicek. The APRAM: incorporating asynchrony into the PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, June 1989.

[21] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, Apr. 1978.

[22] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[23] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings of the ACM Symposium on Principals of Programming Languages*, pages 309–321, Jan. 1996.

[24] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[25] D. A. Krantz, R. H. Halstead, Jr., and E. Mohr. Mul-T: a high-performance parallel lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[26] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[27] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[28] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 597–609. Lecture Notes in Computer Science, vol. 143. Springer-Verlag, Berlin, July 1983.

[29] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[30] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[31] P. Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, June 1991. Also in *ACM SIGPLAN Notices*, 26(9):255–273, Sept. 1991.

*Received December* 3, 1997, *and in final form September* 17, 1998.