Pipelining with Futures

Guy E. Blelloch Margaret Reid-Miller

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
{guyb,mrmiller}@cs.cmu.edu

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 it forces the code to be executed in a highly synchronous manner, making it harder to schedule for locality or space efficiency.

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 then 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, a parallel version of insertion into and deletion from randomized balanced trees (treaps), and insertion into a variant of the PVW 2-3 trees. To determine the runtime of algorithms we first analyze algorithms in a language-based cost model in terms of the work w and depth d of 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, Vishkin and Wagener described a pipelined algorithm for inserting m new keys into a balanced 2-3 tree with n keys in it [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.

A similar idea was then used by Cole [19] to develop the first $O(\lg n)$ time sorting algorithm for the PRAM that was not based on the AKS sorting network [2], which has very large constants. 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 [4].

Although pipelining has lead 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 the futures construct, alleviating these problems. The futures construct was developed in the late 70s for expressing parallelism in programming languages [21, 6] 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 with the pointer needs its value (the result of t_1) it requests it. If the value is ready (has been written) it is returned immediately, otherwise t_2 waits until the value is ready.

To analyze the running time of algorithms programmed with futures we use a two phase 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 implementing the model on various machine models. For the language-based model we use a slight variation of the PSL model [23]. In this model computations are viewed as dynamically unfolding 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

To appear in the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97), 22–25 June 1997, Newport, Rhode Island.

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 `cons` cell is created by its own thread. It also has a consumer running in parallel that consumes these values by summing them. This code pipelines producing and consuming the values.

```
datatype list = cons of int*list | null;

fun produce(n) =
  if (n < 0) then null
  else cons(n,?produce(n-1));

fun consume(sum,null) = sum
  | consume(sum,cons(a,l)) = consume(a+sum,l);

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

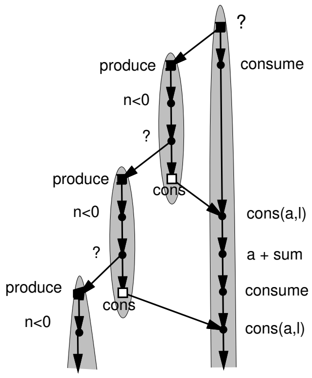

Figure 1: Example code and the top of the corresponding DAG. The code syntax is based on ML and described in the appendix. Futures are marked with a question mark (?). Each node represents an action and each vertical sequence of actions represents a thread. The vertical edges are thread edges, the edges going to the left are fork edges and the edges going to the right are data edges. The `cons(a,l)` in the `consume` code refers to pulling apart the `cons` cell into its two components, `a` and `l`.

We describe and analyze four algorithms with the language-based 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 [3]. We show randomized algorithms for inserting m keys into and deleting m keys from a treap of size n in $O(\lg n + \lg m)$ expected depth and $O(m \lg(n/m))$ expected work. Like the merge algorithm, the code is very simple. There are no previous parallel or pipelined results for treaps of which we are aware. The fourth algorithm is a variant of Paul, Vishkin and Wagener's (PVW) 2-3 trees [28]. The bottom-up insertion used in the PVW algorithm does not map naturally into the use of futures, so 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 $O(\lg n \lg m)$ depth to $O(\lg n + \lg m)$. In both cases the work is $O(m \lg n)$.

To complete the analysis we next consider implementations of the language-based 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, however, does not tell us how to find the schedule online—in particular it does not address the costs of load-balancing the threads and handling the suspension and restarting required by futures at runtime. Since many of the algorithms are dynamic, the schedule cannot be computed offline. A major point of this paper is that all the scheduling and managing of futures can be handled by a runtime system in an algorithm-independent fashion. We are therefore 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 the PSL [23] have shown that any computation with w work and d depth can be implemented online on an 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], $O(w/p + d \lg p)$ on an Asynchronous EREW PRAM [20], and $O(w/p + d)$ on the EREW Scan model [9]. 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 using our approach we lose 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, we perform equally well as the PRAM algorithms: In these 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, we note that the $T_s(p)$ factor in the mapping of our model to the PRAM comes from the allocation of tasks to processors and not by the pipelining itself; in the PRAM the processor allocation needs to be done by the user and often requires significant effort.

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 a directed acyclic graph (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 them.[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 7, 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 7 we show that when the code meets a certain condition, 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. We say that 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. We say that 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 will 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 8). The subset we use is purely functional (no side effects), and we use arrays only for the 2-6 tree described in Section 6 and otherwise just use trees. Futures are created by placing a ? (question mark) before an expression, which will create a thread to evaluate the expression. The number of variables in an ML pattern determines the number of futures 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 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 will refer to the *time stamp* of a value as the depth in the DAG at which it is computed, and will then find upper bounds on the time stamps of the results in order 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 [11]. Although the PSL only considered the pure λ-Calculus with arithmetic operations, the syntactic sugar we have included only affects work and depth by a constant factor. In this paper we are actually assuming a slightly simplified model by only considering 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 Merging binary trees

The first algorithm we discuss takes two binary trees T_1 and T_2, where keys in each tree are unique and sorted when traversed in-order, and merges them into a new binary tree, T_m. It uses the function $\texttt{split}(s,T)$ to split a tree T into two trees, one with keys less than the splitter s and one with keys greater than or equal to s. This function traverses a path down to a leaf separating subtrees based on the splitter to form the two result trees (see Figure 2). 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 generate the left and right subtrees. The left subtree is generated by merging the left subtree of T_1 and the first tree returned by the split, and the right subtree is generated from the right subtree of T_1 and the second tree returned by the split.

Notice that 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 bound 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, therefore allowing

[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.

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

fun split(s,empty) = (empty,empty)
  | split(s,node(v,L,R)) =
    if s < v then
      let val (L1,R1) = ?split(s,L)
      in (L1,node(v,R1,R))
      end
    else
      let val (L1,R1) = ?split(s,R)
      in (node(v,L,L1),R1)
      end;

fun merge(empty,T2) = T2
  | merge(T1,empty) = T1
  | merge(node(v,L,R),T2) =
    let val (L2,R2) = ?split(v,T2)
    in node(v,?merge(L,L1), ?merge(R,R1))
    end;
```

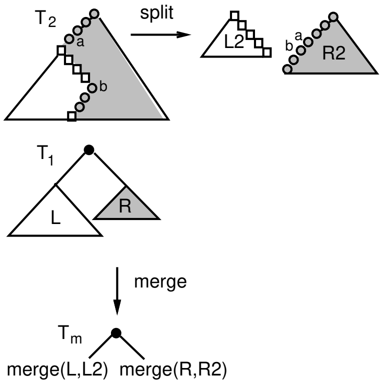

Figure 2: Code for merging two binary search trees and a corresponding figure. The shaded regions are keys that are greater than the key at the root of T_1.

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 appreciate this claim, let us consider the time (depth in the DAG) at which all nodes of the result trees, $(L2, R2) =$ $\mathtt{split}(v, T_2)$, are computed. If the roots of both $L2$ and $R2$ 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 2 the root of $L2$ 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 $R2$; `b` is created only after four nodes of $L2$ have been created. In general, the rightmost path of $L2$ and the leftmost path of $R2$ are made from the nodes of T_2 that `split` traversed, and the time stamp (depth in the DAG) 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. The proof of this property and the following theorem is given in the next section since it is a simplification of the proof we give in that section for melding two treaps.

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.*

Even though the input trees may be balanced the resulting merge tree may have depth up to $\lg n + \lg m$. But again, using pipelining, it can be rebalanced using $O(\lg n + \lg m)$ depth and $O(n+m)$ work, as we briefly describe here. First, the algorithm makes a pass through the tree computing the size of the every subtree, which it stores at the root of the subtree. This takes $O(\lg n + \lg m)$ depth and $O(n+m)$ work and does not use pipelining. Next, it rebalances the tree using a parallel pipelined algorithm similar to `merge`. But this time it uses a split operation (similar to `splitm` in the next section) that takes a size argument and splits the tree into nodes with size less than the argument and nodes with size greater than the argument. It returns these two trees along with the node with size equal to the size argument. The rebalancing algorithm takes three arguments: a tree, half the size of the tree, and an offset. It calls this split operation on the tree and the offset as the size. 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 an offset which is the old offset minus (plus) half the subtree size. The analysis of the depth of the algorithm is similar to the analysis of `meld` in the next section.

4 Treap Meld

Treaps [3] 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 tree. 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 melding. As we will see, they have the added advantage that they can be easily parallelized.

We present two pipelined parallel operations on treaps—a *meld* 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 3 shows the code for melding any 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 tree is balanced. It uses a modified `split` operation, `splitm`, where the splitter can be a key in the tree. When the splitter is in the tree, `splitm` excludes it from the resulting trees and returns it along with the two split trees. Otherwise, it simply returns the two resulting trees. Notice that `splitm` completes as soon as it finds the splitter in the tree.

To maintain the heap order `meld` makes the root with the largest priority the root of the result tree (compare with merge, which always uses the root of the first tree). To maintain the keys in-order `meld` splits the trees by the key value of the new root. For one tree these are trivially the left and right children of the root. For the other tree the algorithm uses `splitm` with futures. It then recursively melds the two trees that have keys less than the root, and melds the two trees that have keys greater than the root. We will show that the expected depth to meld 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 let us consider time stamps $t(v)$ for each node v of the tree (i.e., the depth in the DAG at which each node is created). For a tree T we

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

fun splitm(v1,leaf) = (leaf,none,leaf)
  | splitm(s,node(p,v,l,r)) =
    if s=v then (l,mid(v),r)
    else if s<v then
      let val (l1,m,r1) = ?splitm(s,l)
      in (l1,m,node(p,v,r1,r))
      end
    else
      let val (l1,m,r1) = ?splitm(s,r)
      in (node(p,v,l,l1),m,r1)
      end;

fun meld(empty,b) = b
  | meld(a,empty) = a
  | meld(node(p1,k1,l1,r1), node(p2,k2,l2,r2)) =
    if p1 > p2 then
      let val
        (l,m,r) = ?splitm(k1,node(p2,k2,l2,r2))
      in node(p1, k1, ?meld(l1,l), ?meld(r1,r))
      end
    else
      let val
        (l,m,r) = ?splitm(k2,node(p1,k1,l1,r1))
      in node(p2, k2, ?meld(l,l2), ?meld(r,r2))
      end;
```

Figure 3: Code for Melding Treaps

will 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 will 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.

We define a τ-value of a tree to be 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. In particular, for all $v \in T$, $t(v) \leq \tau + k_s(h(T) - h(v))$, where k_s is a constant. This means that $\tau \geq \max_{v \in T}\{t(v) - k_s(h(T) - h(v))\}$. These τ-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 τ-value places an upper bound on the time stamps that is the same 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 tree of `splitm`, we can find a valid τ-value that depends only on the result tree height, the input tree height, and the input tree's associated τ-value. In the analysis of `meld` we will keep track of the τ-values of the input trees to recursive calls to bound the time stamps in these trees.

Lemma 4.1 (Splitm τ-values) *Consider any split value s and any tree T with associated τ-value τ 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 $T' \in \{L', R'\}$ $\tau' = \max\{t, \tau\} + k_s(1 + h(T) - h(T'))$ is a valid τ-value for T'.*

Proof. We assume that the splitter does not appear in the tree 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 tree. The lemma is clearly true when $h(T) = 1$. Assume it is true for trees of height $\leq h - 1$. We will show

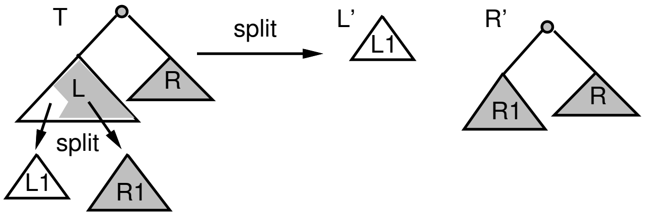

Figure 4: Split of tree T into L' and R'. The shaded areas are keys that are greater than the splitter.

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 $(L1, R1) =$ `splitm`(s, L) (see Figure 4). First, we find a valid τ-value for the right result R' by finding the time stamps for all its nodes. Consider the root of R'. Once the root of T is available we can compare the key at the root with s, obtain R and L, which may be futures, and call `splitm` which returns immediately, since it returns a future. All these take constant time, k_s. Thus, at the root $t(R') = \max\{t, t(T)\} + k_s$. Because $r(R') = R$ we have, for $v \in r(R')$, $t(v) \leq \tau + k_s(h(T) - h(v))$ by definition of τ-values. We next find upper bounds of the times in $L1$ and $R1$.

The recursive call to `splitm` on L can be called at time $\max\{t, \tau\} + k_s$ and, by definition, a valid τ-value for L is $\tau + k_s(h(T) - h(L))$. Therefore, by the induction hypothesis a valid τ-value τ'' for the resulting tree $T'' \in (L1, R1)$ is

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

Since $l(R') = R1$, for $v \in l(R'), t(v) \leq \max\{t, \tau\} + k_s(1 + h(T) - h(v))$. Thus, a valid τ-value for R' is

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

Finally, since $L' = L1$, a τ-value of L' is a τ-value for $L1$ as given in Equation 1. ∎

Note that in the `meld` function we only create new trees by dividing a tree into its left and right children or by running the `splitm` operation on it. Given the above lemma, this means we can find τ-values for the trees in all the recursive calls. We use these τ-values to find upper bounds $\hat{t}(v)$ for $t(v)$, the time stamps on the nodes v of the meld result tree.

Theorem 4.2 (Depth bound on Meld) *Consider two treaps T_1 and T_2 with τ-values τ_1 and τ_2. If we call* `meld`(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, we can compare their priorities, start up `splitm` and the two recursive `meld`s, and create the root of the result tree T_m with pointers to the futures for its two children. This all takes constant time, k_m, because `splitm` and `meld` 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 tree 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 tree, in terms

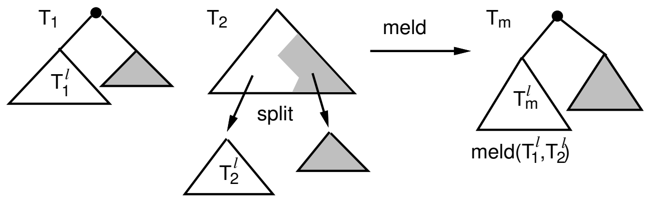

Figure 5: Meld of trees T_1 and T_2 into T_m, when the priority at the root of T_1 is greater than the priority at the root of T_2. T_2 is split by k_1, the key at the root of T_1. The subtree $l(T_m)$ is the meld of the subtrees (not shaded) with keys less than k_1 and the subtree $r(T_m)$ is the meld of the subtrees (shaded) with keys greater than k_1.

of $\hat{t}(T_m)$. Consider the two trees T_1^l and T_2^l, which are the inputs to the left call to `meld`, and T_m^l which is the result of the call $(l(T_m))$. Without loss of generality consider the case where the priority of T_1 is greater than the priority of T_2. Then T_1^l is the left branch of T_1 and T_2^l is the left result of $\texttt{splitm}(k_1, T_2)$, where k_1 is the key at the root of T_1, see Figure 5. Due to the previous bound on the `splitm` operation, a τ-value for T_2^l is

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

By definition, a τ-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

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

which is saying that 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 δ in the depth of T_1 or $\delta - 1$ in T_2. Because we removed 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 trees of the meld that created c. From the above discussion and $j = 1(2)$ and $k = 2(1)$

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

Since the algorithm terminates whenever one of the input trees has height 0, 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 4.3 (Expected meld depth) *The expected depth to meld two treaps of size n and m is $O(\lg n + \lg m)$.*

Proof. We assume that the treaps are "ready" when meld is called at time t. That is, the treaps have valid τ-values, τ_1 and τ_2, with $\tau_1 < t$ and $\tau_2 < t$. Since the expected heights of the two treaps is $O(\lg n)$ and $O(\lg m)$ [3], the expected depth to meld them is $O(\lg n + \lg m)$. ∎

Theorem 4.4 *The expected work to meld two treaps of size n and $m, m < n$ is $O(m \lg(n/m))$.*

Proof. See [29]. ∎

The proof of the depth to merge two trees follows directly from Corollary 4.3. The proof the the work bound for merge is easier than for meld because the input trees are balanced. Meld requires an expected case analysis.

Proof. (of Theorem 3.1) The proof for the depth bound on merge is the same as for the depth bound on meld, except that we do not need to consider the case when T_1 is split. Thus, in Equation 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. ∎

5 Treap Difference

The inverse operation to melding two treaps is taking their difference by removing 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 3) and `join` (shown in Figure 6). 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.

```
fun join(leaf,b) = b
  | join(a,leaf) = a
  | join(node(p1,k1,l1,r1),
         node(p2,k2,l2,r2)) =
    if p1 > p2 then
      node(p1,k1,l1,?join(r1,node(p2,k2,l2,r2)))
    else
      node(p2,k2,?join(node(p1,k1,l1,r1),l2),r2);

fun diff(leaf,b) = leaf
  | diff(a,leaf) = a
  | diff(node(p1,k1,l1,r1),t2) =
    let val (l2,m,r2) = ?splitm(k1,t2);
        val l = ?diff(l1,l2);
        val r = ?diff(r1,r2);
    in
      if m = none then node(p1,k1,l,r)
      else ?join(l,r)
    end;

```

Figure 6: Code for Taking the Difference of Two Treaps

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 subtree is replaced by the join of the two trees resulting from the recursive calls. As in `meld`, 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 `meld` 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 $O(h(T_1) + h(T_d))$ 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 tree.

We use the same definitions as in Section 4, except we replace τ-values with a similar concept of ι-values. Let $d_T(v)$ of a node $v \in T$ be the depth of the node in the tree, such that the $d_T(T) = 0, d_T(l(T)) = d_T(r(T)) = 1, \ldots$. We say a valid ι-value for a tree T defines upper bounds for the time-stamps of the tree, namely for all $v \in T$, $t(v) \leq \iota + k\, d_T(v)$, where k is a constant. In contrast to τ-values, ι-values are independent of the heights of the subtrees.

Lemma 5.1 (Join ι-values) *If* `join` *is called at time t on two treaps T_1 and T_2 with valid ι-values ι_1 and ι_2, then a valid ι-value for the resulting joined treap T' is $\iota' = \max\{t, \iota_1, \iota_2\}+k$, where k is a constant at least as large as the maximum computation DAG depth between successive recursive calls to* `join`.

Proof. We will 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 tree is 1. Assume it is true for result trees of size $n-1$. We will show it is true for result trees of size n. Since `join` can test the root priorities, receive a pointer to the future 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, $t(T') = \max\{t, \iota_1, \iota_2\} + k$. Call this value ι'. 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. Because $l(T_1) = l(T')$, then for all $v \in T', t(v) \leq \iota_1 + k\, d_{T_1}(v) \leq \iota' + k\, d_{T'}(v)$, 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 $r(T') = \texttt{join}(r(T_1), T_2)$, since the size of $r(T_1)$ is less than n. A valid ι-value for $r(T_1)$ is $\iota_1 + k$. Therefore, a valid ι-value for $r(T')$ is $\max\{\iota', \iota_1 + k, \iota_2\} + k = \iota' + k$. Since v's depth in $r(T')$ is 1 less than its depth in T', $t(v) \leq \iota' + k\, d_{T'}(v)$ for all $v \in r(T')$. Thus, ι' is a valid ι-value for T'. ■

Theorem 5.2 (Bound on difference depth)
If $\texttt{diff}(T_1, T_2)$ *is called at time t and valid ι-values for T_1 and T_2 are ι_1 and ι_2, then the maximum time stamp on the result tree T_d is $\max\{t, \iota_1, \iota_2\} + O(h(T_1) + h(T_2) + h(T_d))$.*

Proof. Let k be a constant greater than the maximum computational DAG depth between successive recursive calls to `splitm`, `join`, and `diff`. If ι is a valid ι-value for a tree then $\iota + k\, h(T)$ is a valid τ-value for the tree, since $h(T) - h(v) \geq d_T(v)$, for all $v \in T$ and $k_s < k$. Then, using the same arguments as in Theorem 4.2, after $\max\{t, \iota_1, \iota_2\} + O(h(T_1) + h(T_2))$ depth in the computation DAG, `diff` has reached the bottom of every recursive path (either lines 9 or 10 in Figure 6 applies) and every future result of `splitm` has been computed. Thus, there exists a constant $\iota' = \max\{t, \iota_1, \iota_2\} + O(h(T_1)+h(T_2))$ which is a valid ι-value for all trees (trees `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 ι-values for the results of each recursive call to `diff`. Let ι_l and ι_r be valid ι-values for the results trees `l` and `r`. Because the recursive calls to `diff` are called with futures, the call to `join` is always made by $\max\{\iota_l, \iota_r\}$. By Lemma 5.1 a valid ι-value for result of the `diff` recursive call is $\max\{\iota_l, \iota_r\} + k$. But since all leaves of the recursive call tree have ι' as a valid ι-value and the height of the recursive call tree is no more than $h(T_1)$, a valid ι-value at the root of the call tree must be $\iota' + k\, h(T_1)$ (compare with the definition of the height of a tree). By definition of ι-values, the time stamp of the deepest node in that tree is $\iota' + O(h(T_1) + h(T_d))$ $= \max\{t, \iota_1, \iota_2\} + O(h(T_1) + h(T_2) + h(T_d))$. ■

Corollary 5.3 (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 the two treaps are $\lg n$ and $\lg m$ and the expected height of the result treap is $\lg(n - m)$, the expected depth to find the difference is $O(\lg n + \lg m)$. ■

6 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 Paul, Vishkin and Wagener [28]. Each node of a 2-6 tree has 1 to 5 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 only appears once, either in internal nodes or at the leaves, and all leaves are at the same level. We will 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 $O(1)$ depth and $O(m)$ work. The `array_split` operation can be implemented with all-prefix-sum to broadcast the splitter in constant time in our model as discussed in Section 7. First we consider inserting an ordered set of keys, where 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 2 or 3 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. But 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. It will insert the resulting key arrays into the middle and right children. Before recursively inserting a key array into a child, it 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 3 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 it makes the 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 array. By inserting the keys in this manner we guarantee 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 `insert` has inserted such keys previously. Without pipelining, inserting the $\lg m$ arrays into a tree of size n would have a computation depth of $O(\lg n \lg m)$ and takes $O(m \lg n)$ work.

By simply making the call to insert a well-separated key array return a future (in addition to the futures used in its recursive calls), we 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. 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 contain the current and previous well-separated key arrays. With this structural information in the root this 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 descendents 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.

Theorem 6.1 (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 we 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 i^{th} key array is $k_w \cdot i$.

We say that τ is a valid τ-value for a 2-6 tree T, if for all $v \in T$, $t(v) \leq \tau + k_b(h(T) - h(v))$, where $t(v)$ is the time stamp for v and k_b is constant. Let T_0 be the original 2-6 tree we are inserting into, and τ_0 be its associated valid τ-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

$$\tau_{i+1} = \tau_i + 3k_b \qquad (3)$$

are valid τ-values for $T_{i+1}, i = 0, \ldots, \lg m$[2].

Assume τ_i is a valid τ-value for T_i and k_b is large enough that $\tau_i > k_w(i+1)$. `insert` 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 $min(k_w(i+1), \tau_i) = \tau_i$. 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. Again we assume k_b is large enough such that $k_b > k_r$. This splitting result in a new intermediate tree T_i', with a valid τ-value $\tau_i + k_b$. By definition, $t(T_i') \leq \tau_i + k_b$ and $t(v) \leq \tau_i + (d+1)k_b$ for all descendants v at depth d of T_i', since $h(v) = h(T_i') - d$. 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' 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 $\tau_i + 2k_b$. Then, in constant depth, `insert` can split the keys, determine which 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 $t(T_{i+1}) = \tau_i + 3k_b$. The recursive calls on nodes at level 1 of T_i' are made by $\tau_i + 3k_b$ and these nodes and their children have time stamps no more than that. Therefore, by $\tau + 4k_b$ 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 $\tau_i + (d+2)k_b$ 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 $\tau_i + (d+3)k_b$, proving equation 3 holds. Since there are $\lg m$ well-separated key arrays, the final 2-6 tree has a valid τ-value $\tau + O(\lg m)$ and the tree has depth $O(\lg(m+n))$. Therefore, the largest time stamp is no more than $\tau + O(\lg m + \lg n)$.

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. ■

7 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 tasks on processors. The implementation is based on that described in [23], but we show that for certain types of code we can improve the bounds.

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 associate 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. Previ-

[2] It is also possible to show that $\tau_{i+1} = \tau_i + 2k_b$

ous 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 an all-prefix-sums 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.

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).[3] 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, 5, 1, 18].

Linearizing code does not affect the performance of any of the algorithms we have considered in this paper. For example, consider the body of the *split* code in Figure 2, lines 4–11. 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 $L1$ and $R1$ appear twice in both `then` and `else` parts, but one case is simply defining them (lines 5 and 9) while the other actually references them (lines 6 and 10).

We now consider the main result of this section. Here we state the bounds in terms of the EREW scan model [9], which is the EREW extended with a unit-time scan (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 7.1 (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 bounded-sized structures containing a code pointer and pointers to 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. We store the set of active threads S as an array.

[3] Note that the copy must be strict on the full structure in order to copy it—all futures must be written before they can be copied.

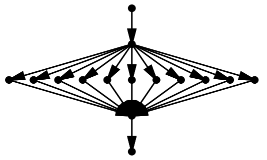

Figure 7: The DAG for an `array_scan` on an array of length 11.

The algorithm will take a sequences of steps, each of which takes $\min\{|S|, p\}$ threads from S, executes one action on each thread, and returns the resulting active threads to S. Each thread can return zero, one or two threads to S (zero if it dies or suspends, one if it continues, and two if it forks or reactivates another thread). We will show that each step takes constant time. When a thread with a read pointer to a future cell wants to read the future cell it checks whether the flag has been set. If it has, it reads the pointer and continues. Otherwise, it sets the flag, writes a pointer to itself into the future cell, and suspends. If a task with a write pointer to a future cell wants to write a result, it also checks the flag of the future cell. If the flag is set it reactivates the task pointed to in the future cell, otherwise it sets the flag. In both cases it writes the value into the future cell. 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.[4] All these operations take constant time. The implementation can also place the threads back in S in constant time using a scan. Therefore, since actions take constant time (by definition) the whole step takes constant time. Since, on each step, the implementation processes $\min\{|S|, p\}$ actions, and S holds all the ready actions (by definition), the implementation will execute a greedy schedule of the computation DAG. The number of steps is therefore bound by $w/p + d$ [12] and the total time by $O(w/p + d)$.

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). 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 [7]—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 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. To account for the cost of the the spawned threads in the `array_scan` operation, we view the operation as a DAG of depth 2 and breadth n (see Figure 7). Each node of this DAG can be executed in constant work. Since the schedule remains greedy (on each step the implementation always schedules $\min\{|S|, p\}$ threads), the number of steps is bound 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

[4] A test-and-set operation will suffice, but we don't have such an operation in an EREW PRAM.

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 [9] for example). Each can be implemented with $O(n)$ work and $O(1)$ depth in a similar way as described above. ∎

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.

8 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 model which 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, 7, 8] 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 yet 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 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 [10]).

Acknowledgements

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. 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.

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 *Proc. ACM Symposium on Theory of Computing*, pages 1–9, Apr. 1983.

[3] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[4] 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.

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

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

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

[8] G. Blelloch, P. Gibbons, Y. Matias, and G. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

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

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

[11] G. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, May 1996.

[12] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. 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 *Proc. 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, 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 Computing*, Research Monographs in Parallel and Distributed Computing, pages 95–113. The MIT Press, 1990.

[16] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with OLDEN (parallel programming). In *Proc. 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Springer-Verlag, 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. The MIT Press, 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 *Proc. 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. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proc. 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 *Proc. SIGPLAN 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, Mass., 1990.

[28] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In *Lecture Notes in Computer Science 143: Proc. Colloquium on Automata, Languages and Programming, Barcelona, Spain*, pages 597–609, Berlin/New York, July 1983. Springer-Verlag.

[29] M. Reid-Miller. Expected work to meld two treaps. Unpublished manuscript, Nov. 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 *Proc. Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, New Haven, Connecticut, June 1991.

A 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 8. 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.

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 will first check if the first argument is a `leaf` type, and return `B` if it is, it will then check if the second argument is a `leaf` type, and return `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.

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* `IN` *exp* `END`	local bindings
		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*`]` `[ | ` *sumtype*`]`	sum type
prodtype	`::=`	**name** `[*` *prodtype*`]`	product type

Figure 8: The ML syntax used in this paper.