A No-Busy-Wait Balanced Tree Parallel Algorithmic Paradigm

Uzi Vishkin *

Abstract

Suppose that a parallel algorithm can include any number of parallel threads. Each thread can proceed without ever having to busy wait to another thread. A thread can proceed till its termination, but no new threads can be formed. What kind of problems can such *restrictive* algorithms solve and still be competitive in the total number of operations they perform with the fastest serial algorithm for the same problem?
Intrigued by this informal question, we considered one of the most elementary parallel algorithmic paradigms, that of balanced binary trees. The main contribution of this paper is a new balanced (not necessarily binary) tree no-busy-wait paradigm for parallel algorithms; applications of the basic paradigm to two problems are presented: building heaps, and executing parallel tree contraction (assuming a preparatory stage); the latter is known to be applicable to evaluating a family of general arithmetic expressions.
For putting things in context, we also discuss our "PRAM-on-chip" vision (actually a small update to it), presented at SPAA98.

1 Introduction

For warm-up, we start with the following example.

1.1 Example: Parallel Summation

The summation problem is considered. Two similar parallel algorithms, both guided by a balanced binary tree, are presented. The second algorithm, which is less synchronous, demonstrates a paradigm for "no-busy-wait" parallel algorithms. This paradigm is used in later sections for two no-busy-wait parallel algorithms which are a bit more involved.

Given an array $A[1 \dots n]$ of numbers, we wish to find their sum $A(1) + A(2) + \dots + A(n)$. Let us assume without loss of generality that $n = 2^k$ for some integer k. Consider a complete binary tree structure. The *height* of a node in the tree is the length of its longest directed path to a leaf of the tree. Using a common binary tree representation, locations $n \dots 2n-1$ in an array B of size $2n-1$ are the leaves of the tree. (That is, $A(i)$ occupies $B(n - 1 + i)$ for $i = 1 \dots n$.) The parent of node i is node $\lfloor i/2 \rfloor$. The left child of node i is node $2i$. The right child of node i is node $2i+1$.

We discuss two parallel algorithms: a synchronous algorithm and a less synchronous one. Both algorithms compute the same things: for every node of the tree they find the sum of the leaves in the subtree rooted at the node.

The synchronous algorithm The standard "PRAM textbook" solution for the summation problem works in $\log n$ steps. The first step operates (in parallel) on nodes at height 1 in the tree. For each such node, the sum of its two leaf children is computed. Step i operates on nodes at height i, adding for each its two children. This can be summarizes as a layer-by-layer lock-step approach, which requires $\log n$ steps. *Time complexity:* The number of steps is $O(\log n)$ each taking $O(1)$ time. *Work complexity:* The total number of operations is $O(n)$.

The less-synchronous algorithm There will be a thread for each node of height 1 in the binary tree. We will also have a *gatekeeper* for every node k whose height is 2, or higher. The gatekeeper will ensure that the sum of the node is computed after the sum of both its children is ready. The gatekeeper is implemented using an "enabling" counter, denoted $E(k)$. $E(k)$ is initialized to 0. A thread i begins at its node of height 1. After adding its two children leaves to produce $SUM(i)$, it will perform a "prefix-sum operation" relative to $E[PARENT(i)]$. For now, if suffices to know that the prefix-sum operation reveals to the thread if it was the first child of $PARENT(i)$ to do that. If it

*Supported by NSF grant 9820955. University of Maryland. E-mail: vishkin@umiacs.umd.edu was, the thread terminates. However, if it was the second, the thread will proceed to add the two children of $PARENT(i)$. Then it will advance to its parent (i.e., $PARENT(PARENT(i))$) and so on. The thread which computes $SUM(1)$, the sum for the root, ends the computation.

Time complexity: The number of operations in the longest thread is $O(\log n)$. *Work complexity:* The total number of operations over all threads is $O(n)$. In other words, the time and work complexities of the two algorithms are the same.

We discuss next some significant differences between the two algorithms which are not captured by the standard PRAM measurements of time and work complexity. (i) *Total number of threads.* Using a "language-based" description, the synchronous algorithm would use one "thread" per node of height 1, or higher. That is, for each level of the tree, starting at a height of 1, the synchronous algorithm spawns as many threads as the nodes of that level and then joins them before proceeding to the next level. This is nearly double the number of threads in the less-synchronous algorithm (one per node of height 1). Note that: (a) these time and work measurements do not count threads; and (b) use of asymptotic complexity measurements would have hidden the difference in the number of threads even if we did. In the same vein asymptotic measurements also hides differences in time and work between the two algorithms. (ii) *Synchronization.* We will need to further articulate what we mean by synchronization. Observe that: (1) The number of Spawn-Join pairs in the synchronous algorithm is $\log_2 n$, while in the less-synchronous one there is only *one* Spawn-Join pair, and (2) *any possible order of summations in the synchronous algorithm is also possible in the less-synchronous one, but not vice versa; that is, there are summation orders which can happen in the less-synchronous algorithm but not in the synchronous algorithm.* Item (2) provides a **criterion** for determining when one algorithm is *less synchronous* than another. This criterion, however, is meaningful *only if* two algorithms perform a sufficiently similar set of operations.

Summary For the purpose of this paper, we highlight the following things in the parallel summation example. (1) In the less-synchronous algorithm, each thread could continue from start to finish without having to ever busy wait for another thread. (2) The balanced binary tree still imposed some order on the summation instructions. (3) The two observations on synchronization in the previous paragraph. (4) The part of an algorithm which is encompassed within a single Spawn-Join pair is said to be *no-busy-wait.* When a whole algorithm is encompassed within a single Spawn-Join pair, it is a no-busy-wait algorithm.

1.2 Background

A motivating context One motivating context for this paper is the *Explicit Multi-Threaded (XMT)* parallel computing framework which could be summarized as a *"PRAM-on-chip" vision.* Looking to the forthcoming "Billion transistor chip era", the following question has been attracting growing attention. The returns on adding more on-chip memory will sooner or later start to diminish; what to do then with all this hardware? XMT is a parallel computing approach which seeks to minimize parallel computing overheads by implementing a few primitives directly in hardware and relying on the practically unlimited on-chip communication bandwidth (as explained in [DL99]). From the programming/algorithmic point of view, which is the main interest of the current paper, XMT seeks to harness PRAM algorithmics with its most valued attribute: programmability. To reconcile efficient architecture implementation and PRAM programmability, a parallel algorithms designer will write code using certain parallel programming constructs.

We use a simple-minded single-program multiple-data (SPMD) language-based version of the PRAM model similar to [VDBN98]. A program in the model proceeds by alternating between running a plurality of parallel threads (parallel state) and running a single thread (serial state). A (preferred) programming style permits every parallel thread to run from its inception till its termination without having to ever *busy wait* for any other thread. Threads can interact, but an *independence of order semantics (IOS)* among parallel threads is used; that is, any order in which the interaction among the threads occurs is valid. The execution of the parallel program (i.e., scheduling instructions for execution) will of course have to abide by the order constraints of such a program. But, the programming style seeks to impose few constraints on instruction scheduling. Although not explicated here, this will allow the executing computing system a relatively high degree of freedom for optimizing performance.

A single thread typically executes a standard serial program. It can switch to parallel state using a Spawn command which specifies a number of threads to be spawned. Each thread is a serial program which runs till it terminates (at a Join command). Once all threads terminate the program switches back into serial state. There are a few differences between a thread and a standard serial program: (i) The thread can have local memory (similar to registers in a standard assembly language). (ii) All threads have a shared memory. (iii) Threads may interact: (1) A thread may include prefix-sum commands of the form $PS(a, s)$, where a is a local variable of the thread and s is a shared variable (called *base*). The prefix sum command means the following atomic outcome: (a) $s := s + a$, and (b) $a := s$, where the right hand side means the values of a and s prior to the command. In case a is a single bit (the base s can be a general integer), XMT assumes unit time execution with respect to PS commands with respect to the same base. This assumption is based on a certain hardware gadget. The prefix-sum command often facilitates no-busy-waits among concurrent threads. It remains to be seen how efficiently our various uses of the prefix-sum instruction will end up being implemented. (2) Writes into a shared memory variable either follow the Arbitrary CRCW PRAM convention (where if several threads try to write into the same shared memory location an arbitrary one succeeds and we don't know in advance which one) or use a prefix-sum command. No nesting of threads is allowed.

Performance measurements We measure three things in an algorithm, of which the first two have been standard for PRAM algorithms. (i) W for *work*. The total number of operations performed over all threads. (ii) T for *time*. The longest serial chain of operations. Among concurrent threads the longest thread is picked for the chain. The underlying assumption being that all parallel threads can progress simultaneously. (iii) S for *synchronizations*. One approach is to count Spawn-Join pairs (plus the number of serial steps). The second approach is relative. It provides a partial order among algorithms, by comparing orders of execution that they allow (as demonstrated for parallel summation).

Discussion Assume that *synchronization* is loosely defined as the arrangement of events to indicate coincidence or coexistence. Synchronization in the above virtual (or synthetic) language-based model is not explicit and actually appears to be rather relaxed. Since there are no busy-waits among concurrent threads, their coexistence during execution is actually not restricted. (However, within each thread separately the execution of its instructions must respect serial semantics, which means that out-of-order execution of instructions within a thread must respect dependences among those instructions.)

At the *executing hardware* level some hardware parts will go unused for some time. The XMT scheduling of threads to thread control units (TCUs), or processors, in [VDBN98] follows a standard greedy approach which starts by filling up all TCU slots. Then, as soon as some program threads terminate, new program threads are assigned to take their place. Towards the end of a Spawn-Join execution, there are no more unassigned program threads and the fewer active program threads remain the more TCU slots become unused. The number of active threads goes down to one and stays at one until a new Spawn command is reached. This explains why, under some assumptions, counting the number of Spawn-Join pairs could also provide a good first approximation for evaluating hardware underuse. For more, see the Scheduling Lemma in [VDBN98].

Preview In the next two sections we consider two problems. (i) Building heaps, which is a standard problem (see [CLR90]). And, (ii) tree contraction, which is a basic technique in parallel algorithmics (see [ADKP89], [CV88a], [KD88], [MR89], [MR91], and [RMM]); these references also describe a considerable number of tree contraction applications.

Section 4 attempts to put the main contribution in context. We recognize the limits of the quantitative reduced-synchrony model as it applies only to "Spawn-Join" parallel algorithms but does not extend to ones that use Fork commands. The partial order criterion is more general. Before doing that, a general parallel programming methodology, which is in line with the *Explicit Multi-Threaded (XMT)* "PRAM-on-chip" vision, is reviewed. This is followed by an example which demonstrates the gap between the two reduced-synchrony models.

2 Build Heap

2.1 Basics

The basis for the algorithm described in the current section is the serial build-heap algorithm in Chapter 7 of [CLR90].

The *(binary) heap* data structure is an array $A[1 \ldots n]$ of numbers (or elements from a totally ordered domain) that can be viewed as a complete binary tree of n nodes. The parent of node i is node $\lfloor i/2 \rfloor$. The left child of node i is node $2i$. The right child of node i is node $2i+1$.

A heap should also satisfy the *heap property*: $A[PARENT(i)] \geq A[i]$.

Suppose that an array B of size n is given as an input. The *Build Heap* problem is to permute the elements of B so that B satisfies the heap property.

The serial algorithm works by iterating a primitive called *Heapify*. The input for Heapify is a subtree rooted at some node i. Node i may have up to 2 children. The subtrees rooted at each of them are assumed to each satisfy the heap property.

Heapify works as follows. It starts at node i. Its *basic step* finds the largest key among node i and its children. If the largest key was at node i Heapify terminates. Otherwise, it exchanges the key at node i with the largest key and advances to the child j that had the largest key. This ends the basic step, which requires $O(1)$ operations. Heapify then applies recursively the basic step to node j. Clearly, the number of basic steps that Heapify(i) applies is upper bounded by the length of the longest path from i to a leaf in its subtree (the *height* of its subtree).

The serial build-heap algorithm works by serially iterating Heapify for all non-leaf nodes of the binary tree. So, for example if n is even then iterate Heapify $n/2$ times for $i = n/2$ down to 1.

2.2 No-Busy-Wait Build Heap

We show a parallel build-heap algorithm with just one Spawn-Join pair. That is, $S = 1$.

The following simple observation is important for understanding the asynchronous algorithm.

Observation. The primitive Heapify (from the serial algorithm) can apply to a node after its children have "cleared" (that is heapify has been applied to them).

The asynchronous Build Heap algorithm - overview There will be a thread for each node of height 1 in the binary tree. That is, a node whose children are all leaves. We will also have an "enabling" counter for every node k whose height is 2, or higher. The counter is denoted $E(k)$ for enabling and is initialized to 0. So, for example, if n is even, and $n/2$ is odd, there will be parallel threads for locations $n/2$ down to $(n+2)/4$ and a counter $E(k)$ for $k \leq (n-2)/4$.

A thread i will begin by applying Heapify to its node exactly as in the serial algorithm. Then, it will perform a prefix-sum operation relative to $E[PARENT(i)]$. If it was the first child of $PARENT(i)$ to do that, the thread terminates. If it was the second, the thread will proceed to execute Heapify on $PARENT(i)$. Then is will advance to its parent (i.e., $PARENT(PARENT(i))$) and so on.

Time complexity We seek an upper bound on the length of the longest thread. That thread may require a call to the Heapify primitive at each level of the tree. Each call require up to as many basic steps as the height of the node at hand. So, the longest thread requires $O(log^2 n)$ operations. This is T, the parallel time.

Work complexity We show that the total number of operations over all threads is $O(n)$. Each time a basic step of the Heapify primitive is performed we charge the node to which Heapify applies (not the node at which the basic step is applied) \$1 (one Dollar). To establish that the total number of basic steps does not exceed n we put initially \$2 at each of the $\leq n/2$ non-leaves of the tree. An *amortized analysis* can now proceed by induction. The *inductive step* will assume that upon completing a Heapify at a node of height h, a total of $\$h$ remain at that node. Its parent will inherit therefore a total of $\$2h$ from its two children and add to it its own \$2 for a total of $\$2h + 2$. Applying Heapify to the parent requires at most $h + 1$ basic steps and therefore at least $\$h + 1$ of the $\$2h + 2$ could be later forwarded to the parent of that parent. This concludes the inductive step. So far, this is not new: the same analysis applies to the serial algorithm as well. The only operations performed by the parallel threads which are not matched by similar operations in the serial algorithm relates to the E variables. There are nearly $n/4$ such variables and each is accessed twice by a prefix-sum instruction. This adds $n/2$ prefix-sum operations to the complexity of the serial algorithm. The total work will therefore be $O(n)$.

Synchronization The number of Spawn-Join pairs above is one which justifies the attribute no-busy-wait. It should be clear that we went ahead and presented directly a less synchronous algorithm. The rudimentary synchronous PRAM algorithm that we started out with (and which has not been presented) should be obvious to the reader. It ran in $O(log^2 n)$ lock-steps and its work and time complexities were the same. It should also be clear why the no-busy-wait Build Heap algorithm is "less synchronous".

Speedup The *speedup* of a parallel algorithm is the ratio between actual parallel running time and the serial running time of the best serial algorithm. If W is very close to W_{ser}, the total number of operations of the serial algorithm, good speeds can be expected. The ratio W/W_{ser} of all algorithms presented in the current paper is upper-bounded by a constant. An experimental follow-up on this work should try to get a closer evaluation of this constant. However, as this work does not include an experimental component, we avoid elaborating on this point.

3 Tree Contraction

The balanced tree paradigm The Summation and the Build Heap algorithms follow a similar paradigm. The computation advances from the leaves of a balanced binary tree to the root. At each node v of the tree, the thread which reaches v first terminates and the one which reaches v second proceeds.

Parallel tree contraction algorithms received considerable attention. In the next section, we show an unsuccessful attempt for how to extend our new no-busy-wait balanced tree paradigm to parallel tree contraction; this will be followed by a successful attempt.

Consider a rooted binary tree T where each node has either two children - a left child and a right child - or is a leaf, and let x be a leaf of T. Assume that the root has two children, and at least one of them has two children. We will define the *rake* operation for leaves whose parent is not the root. Let x be a leaf, let y be its parent (also called the *stem* of leaf x) and u be the parent of y, and let the other child of y (which is also the sibling of x) be z. Applying the **rake** operation to leaf x means the following: *delete x and y and have z become a child of u instead of y.* See Figure 1. Applying the rake operation to a leaf x of the binary tree T, whose parent is not the root yields a tree which is still binary.

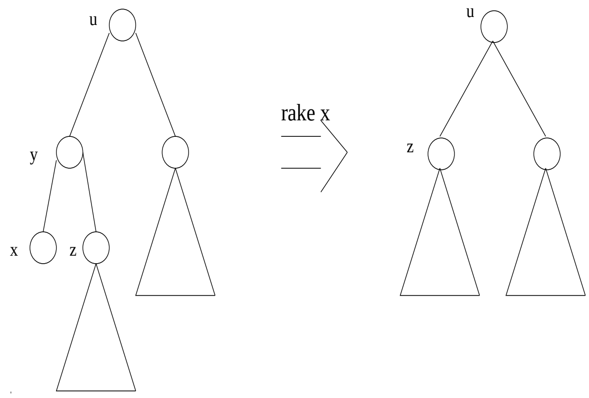

Figure 1: A rake operation

A **serial tree contraction scheme** applies rakes in several steps until the tree T becomes a 3-node binary tree (which consists of a root having two leaves as children).

Parallel rake. Applying the rake operation in parallel to several leaves is considered *legal* as long as the following two conditions hold: (i) no two of the raked leaves have the same parent (stem); and (ii) the parent of a stem cannot be itself the stem of a leaf which is concurrently being raked. Note that applying a legal parallel rake operation to leaves of the binary tree T yields a tree which is still binary.

3.1 First Attempt At Parallel Tree Contraction

A **parallel tree contraction scheme** applies rounds of legal parallel rakes until the tree T becomes a 3-node binary tree. Parallel tree contraction is an important paradigm in parallel algorithms. Specifying the exact order of rake operations gives a **parallel tree contraction algorithm**. We cited above several such known algorithms.

Efficient parallel tree contraction algorithms have typically relied on a preparatory stage. During that stage, data to enable scheduling of rake operations are computed. The Euler tour technique has been the main tool for collecting these data. The Euler tour technique is in turn based on parallel list ranking.

As background, a very simple tree contraction algorithm whose number of rounds is logarithmic in the number of leaves, and is similar to [ADKP89] and [KD88], is presented. Suppose T has n leaves and suppose that they are numbered from 1 to n in the same order in which they are visited in (say) a depth-first search of T.

The rakings stage The main iteration, in which the actual rakings are done, follows:
Main Iteration Pick L_{odd}, the subset of the leaves L consisting of the odd-numbered leaves of the binary tree. Let L_{left} be all leaves in L_{odd} which are left children of their parents.
Step 1 Apply parallel rakes to all leaves in L_{left}, with the exception of a leaf whose parent is the root.
Step 2 Apply parallel rakes to all leaves in $L_{odd} - L_{left}$, with the exception of a leaf whose parent is the root.
Step 3 $L := L_{even}$.
We repeat the main iteration until a 3-node binary tree is reached.

Complexity The main iteration is repeated $\log n$ times, and since each leaf is not raked more than once, this takes a total of $O(n)$ operations and $O(\log n)$ time.

The current paper addresses the following question: to what extent is a less synchronous algorithm possible? However, we have not succeeded to derive a no-busy-wait algorithm from the above raking stage. The problem is that steps 1 and 2 above require suspending "threads". See also the comment entitled "Revisiting the first (unsuccessful) attempt" at the end of this abstract.

3.2 A No-Busy-Wait Approach

We will show a way for implementing the raking stage using the balanced tree paradigm and thereby accomplish the desired no-busy-wait property. However, we were *able to do that only by deviating considerably from the above tree contraction algorithm.* In fact, in order to do that our starting point was a rather different tree contraction algorithm. The paper [CV88a][1] shows how to use a "centroid decomposition" of a tree in order to first derive an $O(\log^2 n)$ time tree contraction algorithm; it also shows how to use a so-called accelerated centroid decomposition for an $O(\log n)$ time algorithm. In both cases the work is $O(n)$.

Let $SIZE(v)$ be the number of nodes in the subtree of T rooted at v. The *centroid level* of v, denoted $C\_LEVEL(v)$, is[2] $\lceil \log SIZE(v) \rceil$. Clearly, each node v has at most one child u such that $C\_LEVEL(u) = C\_LEVEL(v)$. The *centroid path* of node v is the longest directed path, passing through v, where all the nodes on the path have the same centroid level. The *tail* of the path is the node which has no children in the path and the *head* of the path is the node whose parent is not in the path. See Figure 2. Each centroid level may have several disjoint paths. The partition of the nodes into centroid paths is called the *centroid decomposition* of T.

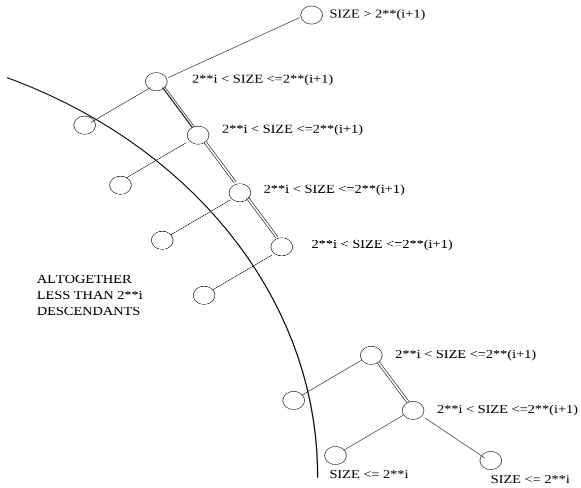

Figure 2: Centroid path at level $i+1$

We already explained what we mean by applying the rake operation to a leaf of the tree. Given a non-leaf node of the tree that has a child which is a leaf, we say that we apply a rake operation to the node whenever we apply it to its leaf child. If the node has two leaf children, apply rake (symbolically) to one of them. No confusion will arise.

For motivation purpose it will be useful to first describe a synchronous $O(\log^2 n)$ time tree contraction algorithm which is similar to the $O(\log^2 n)$ time algorithm in [CV88a]. The idea is to first apply the rake operations to the nodes of centroid level 2, then centroid level 3 and so on. Since each centroid level consists of (disjoint) paths, this could be done for all paths in parallel, as follows. Rake odd numbered nodes in each path; then rake odd numbered nodes in the remaining path, and repeat in up to $O(\log n)$ rounds.

We now proceed to describe our no-busy-wait $O(\log n)$ rounds rake algorithm using the balanced tree paradigm. The idea is to build a balanced tree B which will guide the rake instructions in a way which is reminiscent of the less synchronous algorithms in the previous sections.

Determining the parent in B For each internal (non-leaf) node of T, we show how to determine its parent in B. Consider a centroid path at level $i+1$. Let t be its tail and d its head. In case the centroid path has $k > 1$ nodes, the *idea will be to "partially order" the nodes within the path in a binary tree form.* Denote the nodes of the path $v_1, v_2, \ldots, v_k$ where v_1 is t and v_k is d. Let $s_1, s_2, \ldots, s_k$, respectively, denote $SIZE(v_1), SIZE(v_2), \ldots, SIZE(v_k)$, respectively. We specify next how most nodes in the centroid path determine their parent in B. As can be seen later, determining the parents of t and one more node of the centroid path requires more attention. Consider the binary rep-

[1] What corresponds to the raking operation is defined a bit differently in [CV88a].

[2] The base of all logarithms in the paper is two.

resentation of $s_1, s_2, \ldots, s_k$, and let a_i, $2 \le i \le k$, denote the (index of the) most significant bit in which s_i and s_{i-1} differ. For each a_i, $2 \le i \le k$, find the smaller among its two nearest larger elements. That is, first find the two nearest elements among $a_2, \ldots, a_k$ that are larger than a_i (if such exist): the left nearest larger element $a_j > a_i$ (with $j < i$) and the right nearest larger element $a_f > a_i$ (with $f > i$); second, among these two find the one which is smaller; denote it by b_i. Node b_i will be the parent of node a_i in B. Two nodes in the centroid path are yet without parents. Node t and the node v_i whose a_i value is the largest for $2 \le i \le k$. The parent of v_i is t and the parent of t is $PARENT(v_k)$. A node in the centroid path may have up to 3 children in B. Two from the centroid path itself and one through its child which is outside its centroid path.

Lemma Let $h \le \log n$ be a non-negative integer and let v be a node in T for which $h = C - LEVEL(v)$. Let $H(h)$ denote an upper bound on the height of v in B. Then $H(h) \le H(h-1) + 2$.

Corollary $H(i) \le 2i - 3$. This is since $H(2) \le 1$ and there is equality if $n = 4$.

Time complexity: The number of operations in the longest thread is $O(\log n)$. *Work complexity:* The total number of operations over all threads is $O(n)$.

Synchronization. The raking above is done using a single Spawn-Join pair.

Comment In view of the kind of superscalar architectures that we have seen in recent years and their "hunger" for instruction-level parallelism (ILP), we note an interesting property of the tree contraction algorithm (and possibly other applications of the balanced tree paradigm). Suppose that not only the node currently visited by a thread is ready for processing, but also the node which is its parent. In this case, some "look-ahead" contraction could be used to cut further the processing time by way of more ILP.

The preparation stage Following [CV88a], we note that all the data needed for the raking algorithm could be found through simple reduction to parallel list ranking algorithms, using the Euler tour technique with one exception: the computation of the right (and left) nearest larger elements, which is is an interesting instance of the all nearest smaller values (ANSV) problem [BSV93]. Practical XMT list ranking techniques have been discussed in [DS99]. We have nothing new to add about this stage beyond [CV88a] and [DS99].

A no-busy-wait parallel algorithm for our ANSV problem We refer the reader to [BSV93] for background and PRAM algorithms for the ANSV problem. Next, we jump directly to the no-busy-wait algorithm. Given are an array $s_1, s_2, \ldots, s_k$ of size k, where the s_i values, k, a_i, $2 \le i \le k$, and n are all as above. For simplicity we assume that $\log n$ is an integer which divides k. We partition the array of size k into $k/\log n$ blocks of size $\log n$ each. Each block "defines" a thread. *First,* the thread finds the largest a_i in its block (by serially visiting the $\log n$ elements), denoted a_b. *Second,* in $\log k$ steps the right nearest larger element of a_b, denoted a_x, is found; this is done using binary search; each binary search step comprises of comparing a_b with the most significant bit in which s_b and s_i differ for some $i > b$. The underlying *observation* is that the minimum value over the interval $a_{b+1}, a_{b+2} \ldots a_{i-1}$ is the (index of the) most significant bit in which s_b and s_i differ. (Note that this observation is the only difference between the general ANSV problem and the instance considered in the current paper.)
Third, the thread continues to "merge" elements to the right of a_b (producing their right nearest larger element) with elements from a_x and to it left (producing their left nearest larger element). Following each round of this merge the length of the interval decreases by one and its maximum element is computed (again, based on the above observation). The merging will stop once the value of the maximum element exceeds the value of any one of its end points. (Of course, if a_x belong to the successive block of a_b, the merging will end when the work on the interval is finished.) The same (or another) thread does something similar to the second stage for finding the left nearest larger element of a_b, denoted a_y and then proceeds to a merging stage. This ends the description of a thread.
Correctness Lemma . For every a_i, $2 \le i \le k$, exactly one of the threads will find its right nearest larger element (if exists) and its left nearest larger element (if exists).

Applications of tree contraction The most known applications are in the domain of expression tree evaluation. The case where only additions and multiplication are allowed can be generalized to allowing divisions. Tree contraction works also for several graph problems, such as minimum vertex cover, which have a linear time leaves-to-root serial algorithm for inputs which are trees. See [ADKP89], [CV88a], [KD88], [MR89], [MR91], and [RMM] for these and other tree contraction applications.

Revisiting the first (unsuccessful) attempt Simply running once the synchronous raking stage (in a typical tree contraction algorithm) would yield a balanced tree. The balanced tree paradigm could have then been applied. However, in the current paper we did not allow adding one round of synchronous raking to the preparation stage. This could have nevertheless made sense for recomputations of the same expression tree.

4 An XMT Programming Methodology

One of the SPAA reviewers asked to see a methodology which would extend beyond relatively simple and concrete examples. Sections 4.2 and 4.3 try to address that. But, first, we saw a need to extend the algorithmic model by allowing a nested fork instruction. This is a new element for XMT. Second, we present the programming methodology. We chose to do that by way of analogy to the "parallel architecture programming methodology" in the book [CS99]. Modeling of reduced synchrony is discussed in Section 4.3.

4.1 Adding Fork to the XMT model

We extend the algorithmic model by allowing the *Fork* instruction in the XMT SPMD language-based model.

The basic XMT model, in [VDBN98], allowed switching from serial state to parallel state using a Spawn instruction, but did not allow nesting of Spawn instructions. The Fork instruction *allows an existing thread to fork off another thread and proceed.* The program for the forked thread will be the same SPMD program as of the forking thread. The forking thread will initialize the needed data for the forked thread. While possible implementations, even at the level discussed in [VDBN98], are beyond the scope of the current paper, we note that: (i) the figure of the number of threads currently executed under the current Spawn-Join pair will be incremented (using prefix-sums); (ii) the forking thread will assign an ID to the forked thread (again, using prefix-sums); the forked thread will be executed when its turn will arrive based on its ID; (iii) execution of the current Spawn-Join pair ends once all the threads initiated by the original Spawn command, or through a chain of Fork commands, terminated. Enhancing XMT using nestable Fork commands looks to the author preferable to two alternative options: (i) leave the XMT model without nesting, and (ii) nesting of Spawn commands (as proposed in the EMT model mentioned in [VDBN98]); the reasons being that implementation could use mechanisms similar to the basic XMT Spawn, and simple programming (and efficient implementation) of many more algorithms becomes possible. More work is needed to evaluate this assessment.

4.2 The Programming Methodology

in [CS99]. The "parallel architecture programming methodology" in [CS99] was designed for reducing data access, communication and synchronization cost for current multi-processors. As described in [CS99], it comprises not only a parallel programming methodology but also performance evaluation. The book previews the methodology in its Chapter 2.2. The job of creating a parallel program from a sequential one consists of four steps: (i) *decomposition* of the computation into tasks; (ii) *assignment* of tasks to processes (the books refers to processes and threads interchangeably); (iii) *orchestration* of the necessary data access, communication and synchronization among processes; (iv) *mapping* or binding of processes to processors. The book also comprises a broad experimental evaluation of various programs that have been developed using this methodology; it focuses on two main *measurements*: (i) Speedups of parallel programs on p-processor machines relative to the performance of the *same* programs when run on a machine employing one processor. (ii) Comparison relative to the program's performance on a p-processor PRAM model. The book reasons that the first measurement conveys the overall success in *parallelizing a program*, while the second conveys difficulties due to the cost of communication (as communication costs are abstracted away in the PRAM modeling).

The proposed programming methodology The analogous methodology for the Explicit Multi-Threaded (XMT) approach is fundamentally different (as illustrated in the second column of Figure 3). In fact, it starts with considering which algorithm to pick, and has two more steps: (i) *step-by-step parallelism:* reduce the step count of the algorithm by executing in each step as many parallel operations as possible (in a synchronous, step-by-step PRAM-like manner). (ii) *threading:* assignment of operations to threads; **long threads are sought to reduce synchronization** and to increase locality. The rest is to be done automatically by the XMT software and hardware components.
The current paper quantified the extent to which the balanced tree paradigm–the main contribution of the current paper–helped the threading effort. However, while the programming methodology is described above in general terms we don't see a general way for quantifying the threading effort. The DAG search example, presented later, demonstrates the problem.

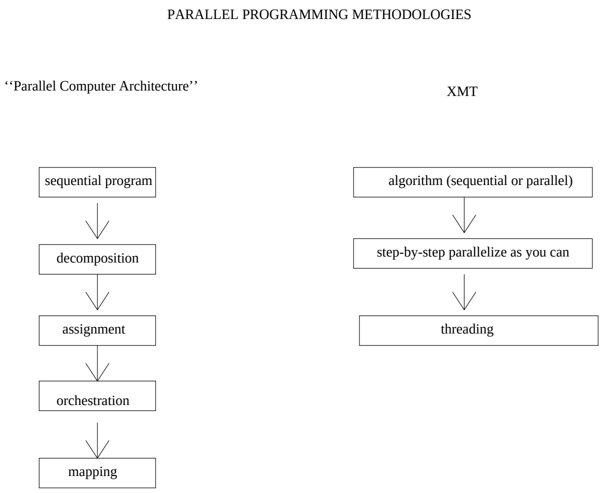

Figure 3: Culler-Singh's Parallel Computer Architecture programming methodology versus the XMT one.

The Level of Parallelism denoted P is (for an ILP architecture) *the number of instructions whose execution can overlap in time.* P cannot exceed the product of the number of instructions that can be issued in one clock with the number of stages in the execution pipeline. (The popular text [HP96] states in p. 330 that "realistic hardware support that might be attainable in the next five to 10 years" [from 1995, when that text came out] includes "Up to 64 instruction issues per clock with no issue restrictions"; so, a standard 5- or 6-stage pipeline leads to P exceeding 300. It is hard to guess why, in spite of this statement, P has only moderately increased since 1995; perhaps, this is because vendors have not found a way to harness such an increase for cost effective performance improvement of existing code.)

Our typical measurement (in [VDBN98]) has been speedups on an XMT execution model relative to the performance of the *fastest serial* program for the *same problem* (which could be completely different than the parallel program explored) on a single processor. The single processor single thread ILP execution model is similar to the intra-thread ILP of a single XMT thread. See [VDBN98].

4.3 Implication for Modeling Reduced Synchrony

Adding nested Fork commands to XMT makes counting the number of Spawn-Join pairs (plus the number of serial steps) far less relevant. It leaves us only with the partial order criterion (of Section 1.1) for evaluating synchronization. As indicated, this criterion is meaningful only if two algorithms that perform a sufficiently similar set of operations are compared. The following example, together with the later comment on the literature, demonstrate that the modeling problems are not limited to XMT.

Example: *Search of directed acyclic graphs (DAGs).* Given a DAG, consider the problem of finding the length of its longest path, counting edges. A rather natural "topological sort" algorithm will first initialize an $INDEGREE$ counter of each vertex to the number of its incoming edges. Then, the algorithm will start with "processing" in parallel all vertices which have no incoming edges (i.e., whose indegree is 0 initially). Processing a vertex v means the following: for each edge (v, w) whose tail is v and head is w (i.e., (v, w) is an outgoing edge of v and an incoming edge of w) "decrement" $INDEGREE(w)$; once $INDEGREE(w)$ is decremented to 0, vertex w is also processed. Incorporating path length computation into the topological sort algorithm is standard.
Programming alternatives: (1) [CLR90] shows how a (serial) depth-first search could be used to implement such an algorithm. (2) A Spawn-Join parallel approach could start in a first stage with a Spawn to process all (the outgoing edges of) the vertices whose $INDEGREE$ is 0. A second stage processes all vertices whose $INDEGREE$ was decremented to 0 in the first stage, and so on.
The above topological sort description suggests starting a Spawn at each vertex to be processed. Since this could be hard to implement, several alternatives can be considered such as the one that follows. (3) Use Spawn to instruct processing all the vertices whose $INDEGREE$ is 0. Once the $INDGEREE$ of any other vertex is decremented to 0, start processing that vertex; the instruct to process such vertex starts with a single thread; using repeated forking a thread will be created for each of the, say, d outgoing edges of the vertex; to reduce parallel time, we will use a standard "balanced binary tree" of height $\log d$ to guide the forkings; that is, the first thread will Fork another thread; each of the two threads will Fork a thread, then each of the four threads will Fork a thread, and so on, until we get d threads. Note that program 3 is fully encompassed within one Spawn-Join pair. However, the requirement that threads terminate prior to processing a vertex, to be followed by later starting other threads is a form of synchronization. The following comment on *modeling reduced synchrony* for the above DAGs search programs elaborates further: Counting Spawn-Join pairs is still valid for comparing the synchronization requirements in programs 1 and 2 above. However, only the partial-order criterion could help for comparing the synchronization requirements in programs 2 and 3 or alternatives similar to program 3.

Literature Ways to control parallel execution have attracted considerable attention since at least the early 1960s. For the sake of brevity, we refer the reader to p. 148 in [AG94], which reviews the multiprocessing literature and gives the needed background for the ways described in the current paper.

Future work It will be interesting to develop more reduced synchrony paradigms, such as the balanced tree paradigm presented in the current paper. The modeling part of this work had one main purpose: help quantify the reduced synchrony aspect of the balanced tree paradigm. We also showed, the above partial order criterion is potentially very useful when considering alternatives during the threading stage in the proposed programming methodology. However, for comparing programs that resulted from algorithms which are sufficiently different new models may be needed.

Acknowledgement Helpful comments by Omer Berkman, Vincenzo Liberatore, Joe Nuzman and the SPAA reviewers are gratefully acknowledged.

References

[AG94] G.S. Almasi and A. Gottlieb. Highly Parallel Computing. Benjamin/Cummings, 1994.

[ADKP89] K.N. Abrahamson, N. Dadoun, D.A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms*, 10(2):287–302, 1989.

[BSV93] O. Berkman, B. Schieber and U. Vishkin. Optimal doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values. *J. of Algorithms* 14:344–370, 1993.

[CV88a] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–348, 1988.

[CLR90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, Introduction to Algorithms, McGraw-Hill, 1990.

[CS99] D.E. Culler and J.P. Singh. *Parallel Computer Architecture.* Morgan Kaufmann, 1999.

[DL99] W.J. Dally and S. Lacy. VLSI Architecture: Past, Present, and Future, Proc. Advanced Research in VLSI Conference, Atlanta, March 1999.

[DS99] S. Dascal and U. Vishkin. Experiments with list ranking for Explicit Multi-Threaded (XMT) instruction parallelism (extended abstract). In Proc. 3rd Workshop on Algorithms Engineering (WAE'99), London, U.K., July 1999.

[Gi93] P.B. Gibbons. Asynchronous PRAM algorithms. In *Synthesis of Parallel Algorithms*, J.H. Reif (editor), Morgan-Kaufmann, 1993, 957–997.

[HP96] J.L. Hennessy and D.A. Patterson., *Computer Architecture - A Quantitative Approach*, 2nd Edition. Morgan/Kaufmann. 1996.

[KD88] S.R. Kosaraju and A.L. Delcher. Optimal parallel evaluation of tree-structured computations by ranking. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 101–110. Springer-Verlag, 1988.

[MR89] G.L. Miller and J.H. Reif. Parallel tree contraction part 1: fundamentals. Advances in Computing Research, Vol 5, 47–72, 1989.

[MR91] G.L. Miller and J.H. Reif. Parallel tree contraction part 2: further applications. SIAM J. Computing 20(6), 1128-1147, 1991.

[RMM] M. Reid-Miller, G.L. Miller and F. Modugno. List ranking and parallel tree contraction, J.H. Reif (editor), Morgan-Kaufmann, 1993, 115–194.

[VDBN98] U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) bridging models for instruction parallelism (extended abstract). In *Proc. SPAA'98*, 1998. For greater detail see the extended summary under `www.umiacs.umd.edu/~vishkin/XMT/`.