Designing synchronous algorithms for asynchronous processors

Ramesh Subramonian ,
University of California, Berkeley

Abstract

The PRAM model has proven to be a fertile ground for algorithm development. However, it assumes that processors operate synchronously, whereas most shared-memory multiprocessors are asynchronous and are likely to remain so. This has motivated the development of simulations of PRAM programs on asynchronous PRAMs. However, such simulations induce either a time or work penalty.

Avoiding this penalty has meant designing specifically asynchronous algorithms. To date, the design of these asynchronous algorithms has been ad-hoc and non-intuitive. We show how many algorithms, designed and analyzed assuming synchrony, can be easily and systematically converted so that the same work and time bounds are maintained under arbitrary asynchrony. The existence of lower bounds indicates that there exist problems for which the same work and time bounds cannot be maintained. However, this paper shows that in far more cases than hitherto thought possible, asynchrony does not induce a time or work penalty.

We suggest a radically new approach to the problem of cache coherence. We show how appropriate architectural support motivates the design of algorithms which are immune to cache incoherence. [1]

1 Introduction

In the commonly used PRAM model of shared memory parallel computation processors are assumed to operate synchronously. However, variations in processor execution speed can arise from a number of sources such as clock skew, varying processor load, multi-programming, interrupts, page faults, cache misses and differences in processor speeds. These variations make it impractical to enforce synchrony for a large number of parallel processors between each parallel step. Hence, a number of recent papers have considered relaxing the PRAM assumption of synchrony (see [Sub91] for a detailed survey).

We measure the performance of an algorithm in terms of *work* which is the total number of instructions executed by all processors during the execution of the program. Work is simply a generalization of processor-time product. In an asynchronous environment, time has no meaning unless one defines the rate at which work is being done. Purely for comparison purposes, we shall assume that the rate of work is comparable to a synchronous evaluation.

Several researchers have shown how PRAM programs can be simulated on a variety of asynchronous and fault-prone PRAMs [KPS90, MSP90, KS91, KPRS91, SM90]. The drawback of these simulations is that they impose a penalty in terms of time or in terms of work compared to the bounds for the simulated PRAM program. Improving on the bounds obtained using the general simulation has required designing specifically asynchronous algorithms. Specific algorithms for list ranking and transitive closure [MS90], connected components [CZ90], and union-find [AW91] have been developed on asynchronous PRAMs.

It is a general principle that the more powerful the machine model, the simpler the algorithm design. The weaker the model, the more practical and less expensive to implement. Providing simulations of more powerful models on weaker models usually entails accepting a loss in performance. Designing specifically asynchronous algorithms to avoid this penalty is unattractive. This paper shows that for a large class of algorithms, one can design and analyze with a more powerful model in mind yet suffer no performance degradation on the weaker model.

The key idea behind this paper is that there is often a certain structure to the way algorithms are designed which can be gainfully exploited. Recognizing this structure allows us to design and analyze algorithms as if they were synchronous and yet, with minor modification, have them run on asynchronous machines with the same work and time bounds. This is a radically dif-

[1]

ferent approach from most research in this field which has concentrated either on general simulations or on specific algorithms.

The main contribution of this paper is providing a cookbook for the design of synchronous algorithms so as to make them immune to asynchrony. We claim that the discipline that needs to be enforced is not burdensome. We validate this claim by showing how fundamental algorithms designed by other researchers, with a synchronous machine as the intended target, can be trivially modified to conform to the discipline necessary for their asynchronous transformation. For instance, in the following examples, our techniques improve on the bounds imposed by the general simulations by a logarithmic factor. These include list ranking using deterministic coin-tossing, tree contraction, maximal matching, maximal independent set (on $O(1)$ degree graphs), Euler tours to compute a variety of tree functions such as level, preorder number, number of descendants.

We use three fundamental paradigms to make systematic transformations from synchronous programs to asynchronous ones: knowledgeable, indifferent and approximately synchronous algorithms. In a knowledgeable algorithm, we can predict the last step in which the variable being read was written to. In an indifferent algorithm, we abandon the notion of a "step". Instead, we use tags both to guarantee progress and to prevent errors arising from an old version of a variable being used. In an approximately synchronous algorithm, the notion of a step is probabilistic: it occurs after a certain amount of work is expected to happen. We prevent a processor with an incorrect idea of the current step from doing damage by making only tentative decisions and deferring final decisions to a synchronization point that is executed when the algorithm is expected to have terminated.

A significant contribution of this paper is conserving space in an asynchronous environment. In [Her90] and [AW91], rather than changing the value of an object in place, a new version of the object is created and a pointer to it returned. In [SM90] and [KPRS91], to avoid the overhead of synchronizing at every step, space is allocated for all variables that will be written to between two synchronization steps. In contrast, our techniques at most double the space requirement.

A processor could read an old value of a variable if it suffered a delay. Equivalently, it could read an old value if it had a cached version of the variable which had not been updated. We delineate the architectural support needed to enable one to design algorithms that are immune to cache incoherence. This enables one to do away with expensive hardware and time-consuming protocols to maintain exact cache coherence.

A motivation for this study has been to get an insight into the inherent limitations imposed by asynchrony. It has been shown that not all problems are amenable to the techniques in this paper. An $\Omega(n + p \log n)$ work bound has been proved for the CWA problem (Lemma A.4) in [BR90, KPRS91, MS91]. An important open problem is to provide a precise characterization of problems for which asynchrony does not impose a penalty.

Please see [Sub92] for detailed proofs and programs.

2 Model

The TPRAM [SM90] is a Tagged version of the A-PRAM (Asynchronous Parallel Random Access Machine) [MSP90]. It consists of a collection of anonymous, asynchronous unit cost RAMs that can perform atomic read and write operations to shared memory without contention. For ease of exposition, we present a model which is close to the standard CRCW PRAM model. In this model, time is divided into unit length slots in which a single low level RAM instruction (read, jump, write, ...) can be executed. It is equivalent to an apparently more asynchronous model where atomic reads and writes occur at arbitrary real points in time.

To prevent a slow processor from writing an old, and possibly incorrect, value, we design a tagged architecture. Every memory cell contains a ordered pair of the form $(value, tag)$. A Write can alter the value of a variable only if its tag is greater than the existing tag. A write alters the value of the variable only after all the reads have occurred. A read returns the value of a variable at the end of the previous slot.

Write semantics. Let the write $X \leftarrow (v_1, t_1)$ be executed by processor p in slot s. Let X contain (v_0, t_0) at the end of slot $s-1$. Assume this is the only Write to X in slot s. If $t_1 > t_0$, then the Write is said to succeed and X contains (v_1, t_1) at the end of slot s; else, X continues to contain (v_0, t_0). If there is more than one Write to X in slot s with a tag greater than t_0, then an *arbitrary* Write with a tag greater than t_0 succeeds. Tagged writes are a masked equivalent of *compare&swap* and hence are both practical and "universal"[Her88]. In the Priority TPRAM model, the write with the highest tag in a slot succeeds, not just an arbitrary Write with a higher tag. Equivalently, we could assume, as in [KRS90], that the outcome of the concurrent execution of several memory accesses is as if these accesses occurred in some serial order (however, all accesses are executed in one slot).

Speed functions. We assume the most general form of asynchrony: processors can proceed at arbitrarily varying rates of speed and can fail at any time except

during a write instruction. Precisely, each processor, j, is assigned, by an adversary, a speed function which is a possibly infinite ordered list $t_1^j, t_2^j, \ldots$ where t_i^j is the slot in which the jth processor *executes* its ith instruction.

Expected Work. We assume this *worst* case choice of speed functions and average over the random choices made by the processors. However, we require that the processor speed functions should not be correlated with the random numbers generated.

Dealing with cache incoherence. The class of algorithms we design are immune to cache incoherence. More precisely, we define the notion of *historical consistency* as follows. Let X be a variable and let X_s be the contents of X at the end of slot s. A read that occurs in slot s can return any element of the set $\{X_0, X_1, \ldots, X_{s-1}\}$. This is necessary for correctness. As long as the memory system is historically consistent, the algorithms present here will be correct. However, to maintain the performance bounds claimed, we require at least a constant probability that a read that occurs in slot s returns X_{s-1}.

3 Notation

A variable consists of a $(value, tag)$ tuple. Variables starting in uppercase (eg., X) are global variables and variables in lowercase (eg., x) are local versions of the corresponding global variable. We will often deal with variables whose value and tag are the same. In that case, we shall use just the variable name, with no subscripts. (An example of this is our use of loop counters where all we want to ensure is that the new loop counter is not clobbered by an old value.)

$x \leftarrow\in_R X$ means that x is assigned a value chosen uniformly at random from the set X.

$EO(f(n))$ is shorthand for *Expected* $O(f(n))$.

For simplicity of exposition, we adopt the following conventions.

1. The algorithm is written for a named set of PRAM processors.
2. The program for each processor is of the form
   repeat
   body of loop { consisting of $O(1)$ work }
   while computation has not terminated
3. Simulating this algorithm on anonymous, asynchronous TPRAM processors is done as follows. A physical processor picks a virtual processor to simulate at random and performs one iteration of the loop for that virtual processor. It continues to do this until the computation terminates.
4. With each processor, p, we keep an indicator of how many logical steps have been performed for it in $Done[p]$. When a physical processor chooses to simulate p, it first determines which step, s, it needs to simulate by $s \leftarrow Done[p] + 1$. On successfully completing that step, it updates the progress indicator by $Done[p] \leftarrow s$.
5. When a physical processor picks a virtual processor to simulate, it knows exactly how much of the loop has been processed and where it should pick up from. (This is easily implemented using tagged program counters. It is not necessary but simplifies exposition.)
6. **Premature reads.** Consider a statement of the form $x \leftarrow Y[s]$. Unless otherwise stated, ff the value of Y returned by the read is $\perp$, the simulating processor abandons the action and picks another virtual processor to simulate at random.
7. **Unique random numbers.** Often we will need to generate a random number for an element. To ensure that all processors use the same random number, we will first write to the global location and then read from it. Only one write with a given tag can succeed.
8. **Termination detection.** The program terminates when a pre-determined set of locations all have one of a set of possible final values. Determining when this has happened is an instance of the CWA problem (Lemma A.4).

   A node is a *frontier* node if it is unevaluated and at least one of its inputs is evaluated.

   (a) A frontier node is said to be *ready* if all its inputs are evaluated.

   (b) A frontier node is said to be *unready* if not all its inputs are evaluated.

4 Knowledgeable and Indifferent Algorithms

Limitation of C-Circuit Theorem. If a computation can be cast as a C-Circuit (Section A.1), it can be performed asynchronously with no loss in work or time compared to a PRAM evaluation of the circuit. However, many computations cannot be cast as circuits. To get around this problem, we show that the actions of the asynchronous algorithm can be construed *as if* they were evaluating the frontier nodes of a circuit. The circuit is defined by the data dependencies and is used to

derive efficiency bounds. The key points are (i) the circuit need not be known in advance; and (ii) the circuit need not be specified explicitly.

What is "knowledgeable" ? In presenting the asynchronous equivalent, every variable of the synchronous program is replaced by a stream of variables, indexed by step. A write: "$X \leftarrow v$" in step s is replaced by $X[s] \leftarrow v$. Reads pose a problem. A Read : "$r \leftarrow X$" in step s must be replaced by $r \leftarrow X[s']$, $s' < s$. It is the algorithm's responsibility to determine, to "know", s'.

Preserving space optimality. Maintaining multiple versions of each variable increases the space requirements. We first design a space-inefficient algorithm. Then, we conserve space by maintaining only the last successful write to each variable. The tagged write semantics ensure that this will be the write with the highest step number. A write: "$X \leftarrow v$" in step s is replaced by $X \leftarrow (v, s)$.

What about a read: "$x \leftarrow X[s']$" ? This is replaced by $x \leftarrow X$. If $x.tag = s'$, all is well. If $x.tag < s'$, this is treated as a premature read. If $x.tag > s'$, it appears that we have lost necessary information. So, we provide a means of inferring the value of $X[s']$ or show how to make do with what we have.

The need for tags. We use $Done[v]$ to record how many steps of virtual processor v have been simulated. After the sth step of v has been simulated, $Done[v]$ is set to s. Tags are necessary to prevent a slow processor from over-writing the correct value of $Done[v]$. ($Done$ is used to ensure that processors pick nodes along the frontier of the circuit used for analysis.) In the space inefficient version of the algorithm, tags are necessary to guarantee that the value of a given version of a given variable is unique. In the space-conserving version, tags are used to guarantee that the latest version of a variable is not over-written by an earlier version.

Consider the processing of a given loop of a virtual processor. If more than one processor is simulating that loop, they may be at different stages of the loop. It is easy to show that the writes issued by a processor further behind in the loop or executing a previous iteration of the loop must fail. In subsequent discussions, we will ignore the actions of laggard processors.

A drawback of knowledgeable algorithms is their inability to exploit immature data i.e., if X is requested and is expected to have tag s, the knowledgeable algorithm abandons the action if the tag of the value returned is less than s. In an *indifferent* algorithm, we can often make some progress with $X_{s'}$, $s' < s$, although not as much as if we had X_s. Tags are used to ensure progress and to prevent a processor using an old version of a variable from doing damage.

Program for sth step of jth virtual PRAM processor
repeat
  $k \leftarrow Pointer[s-1, j]$ **then**
  if $rank[k] = \perp$
    $l \leftarrow Pointer[s-1][k]$
    $Pointer[s][j] \leftarrow l$
  else { compute $rank[j]$ }
    $rank[j] \leftarrow 2^{s-1} + rank[k]$
until $\forall i : rank[i] \neq \perp$

Figure 1: List Ranking: Synchronous Algorithm and Knowledgeable Equivalent

5 Hidden Circuit

We show how a synchronous algorithm can be performed asynchronously by (i) demonstrating the existence of a *hidden* circuit and (ii) restricting the asynchronous algorithm to perform actions equivalent to evaluating the hidden circuit. A hidden circuit has the property that it can be constructed even before the input is presented. However, the mapping of input elements to input nodes is done implicitly only after the input is presented.

The example we shall use is Wyllie's list ranking algorithm [Wyl81]. We start with an array *link* which contains the initial links and an array *rank*, which is set to $\perp$ except for the end of list element, which has rank 0. For the asynchronous equivalent, we use an array $Pointer[0..\log n][1..n]$ such that $Pointer[s][j]$ will contain the pointer from j of length 2^s.

Hidden Circuit Construction. The precedence constraints of Wyllie's algorithm can be viewed as hidden circuit, the top layer of which is the links of length 1, the second the links of length 2, then 4,..., up to $n/2$. The ith column represents the links computed for the element with rank i. The knowledgeable algorithm evaluates the entries of $Pointer$. Each data node of the hidden circuit corresponds to an entry of $Pointer$. The key point is that evaluating an entry of $Pointer$ does not require knowing which node in the circuit it corresponds to. The circuit has width $O(n)$, depth $\log n$, and $O(n^2)$ paths. Theorem 5.1 follows from Lemma A.5.

Theorem 5.1 *List ranking requires $EO(n \log n)$ work ($O(n \log n)$ space) using $p \leq n$ processors.*

Other examples. The [CV86b] algorithm to compute a 2-ruling set in $O(\log^* n)$ time can be modified so that a hidden circuit can be constructed for it. Parallel prefix can be computed in $EO(n)$ work using $p \leq \frac{n}{\log n}$ processors.

6 Prior circuit

In this section, we show how an asynchronous algorithm can be analyzed by constructing a *prior* circuit. A prior circuit can be constructed in entirety before the computation starts but only after the input is presented. The example we shall use to illustrate this is Luby's [Lub86] algorithm for Maximal Independent Set (MIS), *restricted to graphs of constant degree.*

In each step of the synchronous algorithm, each node, v, picks a random number, stored in $\Pi[v]$, such that with high probability, the numbers picked are unique. A node is put into the IS if its number is strictly smaller than that of its neighbours. All nodes adjacent to a node in the IS are then deleted and we repeat the algorithm on the sub-graph induced on the remaining nodes.

Intuition behind asynchronous transformation. Consider the processing of node v in step s. If v's membership in the IS has been decided ($Mis[v] \neq \perp$), we have nothing to do. Else, we need the random numbers generated by v's neighbours in step s. Consider a neighbour w. If w's membership in the IS has not been decided prior to step s, then we need w's sth random number. If w's membership in the IS has been decided, then if w is in, v is out and if w is out, we can ignore it. In short, in processing v in step s, we need either $Mis[w][s-1]$ or $\Pi[w][s]$ to be defined and only one of them can be $\perp$. If both are $\perp$, the read was premature and is abandoned. The dependency constraints so defined induce a prior circuit, to which we can apply Lemma A.5. The prior circuit has an infinite number of identical levels. The vth computation node in the sth row, $C[v][s]$, corresponds to the actions performed by the vth virtual PRAM processor in step s and is described in Figure 2.

Conserving space. To conserve space, we replaced $p_w \leftarrow \Pi[w][s]$ with $p_w \leftarrow \Pi[w]$. If $p_w.tag > s$, it can be inferred that w does not pose an impediment to v's induction into the IS in step s. Further, if v is inducted into IS in step s, then w cannot be placed in MIS. Similarly, we replaced the read of $Mis[w][s-1]$ with $mis_w \leftarrow Mis[w]$. If $Mis[w] \neq \perp$, then that is its final value. If $Mis[w] = \perp$, this might not be the correct value of $Mis[w]_{s-1}$ because the write to it might be delayed. However, we show that assuming that $Mis[w]_{s-1} = \perp$ does not cause an error.

Theorem 6.1 *MIS for constant degree graphs requires $O(n+m)$ space and $EO((n+m)\log n)$ work using $p \leq (n+m)$ processors.*

Proof Sketch. Since the algorithm is knowledgeable, the asynchronous equivalent evaluates a prior circuit induced by the synchronous algorithm. The circuit has depth $EO(\log n)$ [Lub86]. It has width $O(n)$, degree $= \Delta = O(1)$, and $EO(n\Delta^{\log n})$ paths. The proof follows from Lemma A.5. □

Program for sth step of vth virtual PRAM processor.
repeat
  if $Mis[v] = \perp$ **then**
    $\Pi[v][s] \leftarrow_R \{1, .., n^4\}$; $p_v \leftarrow \Pi[v][s]$
    { Each node picks a random number }
    $in\_mis \leftarrow true$
    for $w \in N(v)$ **do** { for all neighbours of v }
      case $Mis[w][s-1]$ **of**
        $false$: NOP
        $true$: $Mis[v][s] \leftarrow false$
        $\perp$: $p_w \leftarrow \Pi[w][s]$; **if** $p_w \leq p_v$ **then**
          $in\_mis \leftarrow false$
    if $in\_mis = true$ **then**
      $Mis[v] \leftarrow true$
  else
    $Mis[v][s] \leftarrow Mis[v][s-1]$
until $\forall v : Mis[v] \neq \perp$

Figure 2: Knowledgeable MIS Algorithm

Other examples. One can construct a prior circuit with width $O(1)$ and depth $O(1)$ for the [TV85] algorithm for finding an Euler tour for a tree. By concatenating this with a hidden circuit for list ranking, (with appropriate initial weights) we can obtain other tree functions such as depth-first search, pre-order numbering, level numbers, number of descendants. Therefore, for these algorithms, the same work and time bounds hold on a TPRAM as on a PRAM.

7 Dynamic Circuit

There are algorithms for which we can construct only the edges into and out of frontier nodes of the circuit induced by the synchronous computation. Such a circuit is called *dynamic*. We explicate the idea of dynamic circuits by designing a new asynchronous algorithm for connected components. This is a significant departure from the [SV82] paper based on tree-hooking and asynchronous variations thereof [CZ90]. It illustrates how knowing that a knowledgeable algorithm is the target *influences, but does not hamper,* the design of the synchronous algorithm.

Problem description. The input is an undirected graph $G = (V, E)$, presented as a list of edges in some arbitrary order, $E[1..m]$. The eth edge is the arc $(e.u, e.v)$. The output of the algorithm is an array $P[1..n]$ such that

if u and v are in the same component, then $P[u] = P[v]$. Initially, $\forall u : P[u] = u$. Without loss of generality, we assume that the graph is connected and that each edge is stored twice, once as (u, v) and once as (v, u).

Outline of Synchronous Algorithm. Initially, all nodes have a pointer to themselves and are called *leaders*. A node which points to a node other than itself is called a *follower*. In an iteration, each edge tries to make one of its endpoints the "owner" of the other. Write conflicts are resolved arbitrarily. After all edges have written, each leader has an owner. We would like each leader to create a pointer to an owner such that there is no cycle in the pointers created. We use the Random Mate strategy [MR85]. Each leader picks a sex at random. If it is male and if its owner is female, it creates a pointer to its owner. We expect a quarter of the leaders to create such a pointer and become followers. Followers do not perform the above actions. Instead, after the leaders have done the above, they "pointer-jump" so that they always point to a leader. At the end of an iteration, we update all edges so that the endpoints of the edges are leaders. This is done by replacing each endpoint with the node it points to, which is necessarily a leader. As a consequence, some edges may become *redundant* (both endpoints equal) and will not be used thereafter.

Lemma 7.1 *No node points to a follower. Precisely,* $\forall u : P[P[u]] = P[u]$

Lemma 7.2 *All edges are between leaders. Precisely,* $\forall e = (x, y) : P[x] = x \wedge P[y] = y$

Lemma 7.3 *The number of leaders can only decrease.*

Lemma 7.4 *In a step, the probability that a leader becomes a follower is* $\frac{1}{4}$.

Lemma 7.5 *The number of steps before all edges are redundant is* $EO(\log n)$

Proof. From Lemma 7.3 and Lemma 7.4, we can assert the following probabilistic recurrence relation on the steps before all edges are redundant. $T(n) = 1 + T(h(n))$, where $T(1) = 1$, $E[h(n)] = \frac{3n}{4}$. By Theorem 1 of [Kar91], it follows that $P[T(n) \geq w + 1 + \lfloor \log_{4/3} n \rfloor] \leq \frac{3}{4}^{(w-1)} \frac{n}{4/3^{(\lfloor \log_{\frac{4}{3}} n \rfloor + 1)}}$. A little manipulation yields $P[T(n) \geq (w+2)(\lfloor \log_{4/3} n + 1 \rfloor)] \leq \frac{1}{n^w}$ The proof follows. □

Theorem 7.1 *Computing the connected components of an undirected graph requires* $EO(\log n)$ *work using* $p \leq n$ *processors on a TPRAM.*

Proof Sketch. Since the algorithm is knowledgeable, the asynchronous equivalent, evaluates some (based on how writes are arbitrated) dynamic circuit induced by the synchronous algorithm. The circuit has depth $EO(\log n)$, width $O(n + m)$, degree $= \Delta = O(1)$ and $EO((n+m)\Delta^{\log n})$ paths. By Lemma A.5, the theorem follows. □

Conserving space. In the following discussion, we assume that s is the tag expected and that s' is the tag returned. Since we store only the latest version (with the highest tag), we must specify what happens when $s' > s$. (Recall that if $s' < s$, it is treated as a premature read and the loop abandoned.) If $s' > s$, it would appear that we have lost X_s. In that case, we resort to one of the following approaches. (i) We prove *impossibility*: $s' \not> s$, unless the simulating processor had suffered a delay. As discussed in Section 3, this case can be ignored. (ii) We provide a means of *inferring* the lost value, X_s. (ii) We show how using $X_{s'}$ has the same effect as if the computation had had all necessary intermediate values. This speeds up circuit evaluation and is called *short-circuiting*. Starting with the first short-circuit and working backwards, we show that the effect of short-circuiting is the same as if we had used the intermediate values and performed the steps one at time. The following discussion pertains to asynchronous program in Figure 3.

1. We replace $s_u \leftarrow Sex[u][s]$ with $s_u \leftarrow Sex[u]$. But $s' \not> s$ unless the processor fell behind.

2. Inferring lost value. We replace $s_o \leftarrow Sex[o][s]$ with $s_o \leftarrow Sex[o]$. Sex was used to ensure that if u became a follower of o in step s, then o did not become a follower in step s. If $Sex[o].tag > s$, we show that o did not become a follower in step s. Hence, u *infers* that it can point to o in step s.

3. Impossibility. We replace $o \leftarrow Owner[o][s]$ with $o \leftarrow Owner[u]$. $Owner[u] = (o, s') \Rightarrow \exists e = ((u, o), s' - 1)$ (which wrote $(o, s' - 1)$ to $O[u]$) (Line C of Figure 3). But this implies that $P[u].tag \geq s' - 1 > s - 1$ (Line E) which implies that the processor has fallen behind.

4. Short-circuiting.
   We replace $gp_u \leftarrow P[p_u, s]; P[u, s] \leftarrow gp_u$ with $gp_u \leftarrow P[p_u]; P[u] \leftarrow gp_u$.

5. Short-circuiting. In program for eth processor, we replace $p_u \leftarrow P[u]$ and $p_v \leftarrow P[v]$ by:
   $gp_u \leftarrow P[u]; gp_v \leftarrow P[v]; E[e] \leftarrow ((gp_u, gp_v), s')$ where $s' = min(gp_u.tag, gp_v.tag) \geq s$. (W.l.o.g., we assume that writing the two endpoints of the edge can be done atomically.)

Program for uth processor, $1 \leq u \leq n$
repeat
1. $p_u \leftarrow P[u][s-1]$
2. **if** $p_u \neq u$ **then**
3. $\quad gp_u \leftarrow P[p_u][s]$
4. $\quad P[u][s] \leftarrow gp_u$
5. **else**
6. $\quad o \leftarrow Owner[u][s]$
7. $\quad Sex[u][s+1] \leftarrow_{\in R} \{M, F\}$; $s_u \leftarrow Sex[u][s]$
8. $\quad s_o \leftarrow Sex[o][s]$
9. $\quad$ **if** $s_u = M$ **and** $s_o = F$ **then**
10. $\quad\quad P[u][s] \leftarrow o$
11. $\quad$ **else**
12. $\quad\quad P[u][s] \leftarrow u$

until $\forall e \in E : E[e].u = E[e].v$ { all edges redundant }
Program for eth processor, $1 \leq e \leq m$
repeat
A. $u \leftarrow E[e][s-1].u$; $v \leftarrow E[e][s-1].v$;
B. **if** $u \neq v$ **then**
C. $\quad Owner[u] \leftarrow (v, s)$
E. $p_u \leftarrow P[u][s]$; $p_v \leftarrow P[v][s]$
F. $E[e, s] \leftarrow (p_u, p_v)$

until $\forall e \in E : E[e].u = E[e].v$ { all edges redundant }

Figure 3: Algorithm ACC

Other examples. We mention a few fundamental algorithms for which the dynamic circuit technique is applicable. Therefore, these algorithms, with minor modifications, can be simulated on a TPRAM with the same work and time bounds as on a PRAM. Tree contraction of an n-leaf binary tree using $O(n)$ work and $p \leq \frac{n}{\log n}$ processors [KR90]. Finding a random permutation in $EO(n)$ work using $p \leq \frac{n}{\log n}$ processors [MR85]. Using the prior circuit for constructing the 2-ruling set and the hidden circuit for list-ranking and the dynamic circuit for the bucket-sort, list-ranking using $EO(n)$ work and $p \leq \frac{n}{\log n \log^* n}$ processors [CV86b].

8 Approximate synchrony: using negative information

A limitation of the techniques we have presented so far is their inability to make use of negative information. To illustrate this, consider the algorithm for maximal matching, based on Luby's Maximal Independent Set algorithm [Lub86] on the Max-PRAM model. This is a strong PRAM model which assumes that if there is more than one write to a location in a step, the write with the highest value succeeds. We use the more powerful Priority TPRAM model. Each edge is assigned a unique random number with high probability. Each edge writes its number into its endpoints. The highest write into a node succeeds. An edge is put into the matching if both its writes are successful. All edges incident on an endpoint of a matched edge are deleted and the procedure repeated on the residual graph. It requires $EO(\log m)$ steps using m processors.

Under asynchrony, an edge cannot put itself in the matching until it is sure that all competing edges have committed their writes. This is equivalent to the CWA problem. Hence, an optimal asynchronous equivalent seemed unlikely, but for the following key observation.

Observation 8.1 *If in each step, an edge that successfully writes into both its endpoints puts itself in the matching with at least a constant probability, the asymptotic complexity is unchanged.*

An approximate step. We assume that we have a bell which goes off after m expected work has been performed. A processor becomes aware that the bell has gone off only after completing an iteration of the loop it is executing. Each time the bell goes off, the processors move on to the next step, as specified by the bell. Such a bell can be simulated by a coin toss.

Outline of algorithm. An edge writes into a node with the value field as its identity and the tag field consisting of two sub-fields, (s, r), s for the step and r for the random number. A write of step s must necessarily overwrite a write of step $s-1$.

An edge $e = (u, v)$ writes into either u or v only if they have not been *touched* by a matched edge. Let $\Pi[u].val = e = (u, w)$. If $\Pi[w].val = e = (u, w)$ then u is touched by the matched edge (u, w). It is possible that an edge that *seems* matched at time t may be unmatched at time $t' > t$.

Consider an approximate step in which *some* m edges, chosen at random, were simulated. Consider an edge that would have placed itself in the matching had it been selected. The probability that it is selected is $(\geq \frac{e-1}{e})$ (Lemma A.1). Hence, by Observation 8.1, the asymptotic rate of progress is the same.

All judgements as to whether an edge is matched are tentative. When the probability of termination is at least a half, we freeze $\Pi[1..n].val$ by copying it into an auxiliary array and then determine whether the matching is maximal. If not, we repeat.

Theorem 8.1 *Finding a maximal matching requires $O(m+n)$ space and $EO((m+n)\log n)$ work using $(m+n)$ processors on a TPRAM.*

Other examples. This idea can be extended to construct a spanning tree in $EO((m+n)\log n)$ work using $p \leq (m+n)$ processors.

9 Possible Circuits

The key idea here is not to restrict the *computation* to follow a particular path but to restrict the *analysis* to count only progress made along one fixed *possible* path. Not counting progress made by actions not represented in the possible circuit can only over-estimate the work and does not cause an error because the final answer is unique. The example we shall use is transitive closure. In the synchronous program, the (i,j,k) PRAM processor performs "**if** $A[i,j] = 1 \wedge A[j,k] = 1$ **then** $A[i,k] \leftarrow 1$" $\log n$ times. In the indifferent algorithm, the (i,j,k) virtual PRAM processor performs "**if** $A[i,j] = 1 \wedge A[j,k] = 1$ **then** $A[i,k] \leftarrow 1$" until the computation terminates.

Constructing a possible circuit. We construct a layered circuit, L, which represents one of the possible ways in which the computation could have proceeded. To distinguish between the digraph A and the circuit being constructed, we shall refer to nodes in the circuit as *gates* and edges in the circuit as *wires*. A gate in the circuit L represents an edge in the transitive closure A^*. Therefore there are at most n^2 gates in this circuit. A wire from (a,b) to (c,d) in L means that (a,b) was one of the edges in A used to construct (c,d) in A. We define l, the length of edge (i,j), to be the length of the *shortest* path from i to j in the graph represented by A. If edge (i,j) has length l, then gate (i,j) is in layer $k = \lceil \log l \rceil$.

Therefore, layer 0 consists of the gates corresponding to the initial edges given. These are edges of length 1. Now, each edge of length l could be constructed in at least $l-1$ ways. However, our circuit will only allow an edge in A to be constructed in a unique fashion. We insist that edge (i,k) of length l, be constructed from two edges of length $\lceil l/2 \rceil$ and $\lfloor l/2 \rfloor$. Let us say that these edges are (i,j) and (j,k). So, in L we shall construct a wire from (i,j) to (i,k) and from (j,k) to (i,k). There may be several values of j which satisfy the above constraint but an arbitrary single j value is chosen. These are the edges from layer $\lceil \log l \rceil - 1$ or $\lceil \log l \rceil - 2$ to layer $\lceil \log l \rceil$. This means that each gate in L has a constant in-degree of 2. Therefore, L has $O(n^3)$ paths. It is easy to see that the circuit has depth $O(\log n)$.

Lemma 9.1 *The work done before TC is computed is* $EO(n^3 \log n)$

Proof Sketch. If gate (i,k) in L has incoming wires from (i,j) and (j,k), then selecting triple (i,j,k) is equivalent to selecting gate (i,k). We can define a block of $O(n^3)$ work consisting of selecting n^3 triples such that each gate of L has at least a constant probability of being selected. L has $O(n^3)$ paths and a depth of $O(\log n)$. The proof follows from Lemma A.3. □

We check for termination at the end of an approximate step of $\Theta(n^3 \log n)$ work. We freeze the matrix and count the number of 1 entries. We then perform $A \leftarrow A + A^2$ which requires $EO(n^3 \log n)$ work using n^3 processors. We re-count the 1 entries in A and we know if that the count has not changed, the computation has terminated. Else, we repeat.

10 Hypothetical circuits

We now present an algorithm for which constructing any kind of circuit seemed hopeless. This was because it left behind no trace which we could follow as in previous examples. So, we created a hypothetical circuit, which was the trace the algorithm would have left behind *had it operated synchronously*. Then, rather than mapping the *actions* of the algorithm to a colouring of the circuit, (see Lemma A.5), we mapped the *progress* made to a colouring. We then lower bounded the expected rate at which the colouring occurred, thereby lower bounding the expected rate at which progress was made. The example we use is list ranking.

We store only one link for each item. Each link is tagged with its length. Initially, $\forall i : link[i].tag = 1$, except the end of list element which has a tag of 0. During the computation, if $link[i] = (j,l)$, it means that list item i has a pointer of length l to item j.

The algorithm picks items at random. If $link[i] = (j,l_i)$, then item i has a pointer of length l_i to j. If $link[j] = (k,l_j)$, j has pointer to k of length l_j. We update i's pointer to point to k and to be of length $l_i + l_j$ by $link[i] \leftarrow (k, l_i + l_j)$. The tags ensure that the length of an item's pointer never decreases. Since the pointer and its length are written atomically, consistency is maintained.

Theorem 10.1 *List ranking requires* $O(n)$ *space and* $EO(n \log n)$ *work using* $p \leq n$ *processors.*

Intuition behind proof. The main difficulties in proving what seemed an obvious result was (i) the number of hops from a node to the tail of the list could increase and (ii) the effect of different pointer jumps was not independent. So, we need to define a *ready* node (see Section A.1) in the hypothetical circuit in such a

manner that Lemma 10.1 holds.

$P[j,i]$ is a frontier node if $2^{j-1} \leq link[i].tag < 2^j$ Let $link[i] = (k, l_i)$. This means that i has a pointer to k of length l_i. Let the last successful write to $link[i]$ have occurred at time t_i. Let $link[k] = (m, l_k)$ at time t_i. $P[j,i]$ is *ready* if $l_i + l_k \geq 2^{j+1}$; and $P[j,i]$ is *unready* if $l_i + l_k < 2^{j+1}$.

Lemma 10.1 *A ready node can never become unready.*

Theorem 10.2 *List ranking, using TLR2, requires $EO(n \log n)$ work using up to n processors.*

11 Conclusions

We have outlined simple and practical yet powerful architectural support for asynchrony using tags. We have shown how a large class of synchronous PRAM algorithms can be designed so that fault-tolerant simulations on asynchronous PRAMs do not impose a time, work or space penalty. A fundamental open question that remains is the precise characterization of problems for which asynchrony does not impose a penalty compared to the best PRAM algorithm.

Acknowledgements. I would like to thank Richard Karp, Charles Martel and Abhiram Ranade for many fruitful discussions.

References

[AW91] R. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. of 23rd STOC*, pages 370–380, 1991.

[BR90] J. Buss and P. Ragde. Certified write-all on a strongly asynchronous pram. Technical report, U of Waterloo, 1990. manuscript.

[CV86a] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proc. of 18th STOC*, pages 206–219, 1986.

[CV86b] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[CZ90] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proc. of 2nd SPAA*, pages 85–94, 1990.

[Fel57] W. Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley & Sons, 1957.

[Her88] M. Herlihy. Wait-Free Synchronization, In *TOPLAS*, Jan 1991.

[Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. of POPL*, March 1990.

[Kar91] R. M. Karp. Probabilistic recurrence relations. In *Proc. of 23rd STOC*, pages 190–197, 1991.

[KPRS91] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining definite and tentative executions for dependable parallel computing. In *Proc. of 23rd STOC*, pages 381–390, 1991.

[KPS90] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proc. of 22nd STOC*, pages 138–148, 1990.

[KR90] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared memory machines. In *Theoretical Computer Science.* North Holland, 1990.

[KRS90] C. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[KS91] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms on restartable fail-stop processors. In *Proc. of 10th PODC*, 1991.

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4):1036–1053, 1986.

[MPS89] C. U. Martel, A. Park, and R. Subramonian. Work-optimal asynchronous algorithms for shared memory parallel computers. *SIAM Journal of Computing*, Dec 1992.

[MS90] C. U. Martel and R. Subramonian. Asynchronous algorithms for list ranking and transitive closure. In *Proc. of ICPP*, pages 60–63, 1990.

[MS91] C. U. Martel and R. Subramonian. On the complexity of certified write all algorithms. Technical Report CSE-91-24, UC Davis, 1991.

[MSP90] C. U. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proc. of 30th FOCS*, pages 590–599, 1990.

[MR85] G. Miller and J. Reif. Parallel tree contraction and its applications. In *Proc. of 26th FOCS*, pages 478–489, 1985.

[SM90] R. Subramonian and C. U. Martel. How to emulate synchrony. Technical Report CSE 90-26, UC Davis, 1990.

[Sub91] R. Subramonian. *Asynchronous Algorithms for Shared Mmeory Parallel Computers.* PhD thesis, UC Davis, 1991.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Sub92] R. Subramonian. Designing synchronous algorithms for asynchronous processors. Technical report, UC Berkeley, 1992.

[TV85] R. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal of Computing*, 14:862–874, 1985.

[Wyl81] J. C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell U, 1981.

A Background results

Lemma A.1 *[Fel57] If m selections are made with replacement from m items, the probability that a given item is not selected is $(1 - \frac{1}{m})^m$. $\forall m \geq 2, \frac{1}{4} \leq (1 - \frac{1}{m})^m \leq \frac{1}{e}$*

Lemma A.2 *[MS91] Let A be an algorithm which consists of a main loop such that at most k instructions are executed in a complete loop iteration, and let p be the number of processors executing A. Then for any $W \geq pk$, the parallel execution of A can be broken up into consecutive blocks of $\Theta(W)$ work such that each block contains at least $\frac{W}{k}$ complete iterations of the main loop.*

Lemma A.3 *[MPS89] Consider a circuit of depth d with P paths from the inputs to the outputs. In each step any gate which has all its inputs computed has at least a constant probability $c > 0$ of computing its output. Then, the expected number of steps until all outputs are computed is $\leq \frac{6}{p}(d + \log P)$ and the probability that more than $\frac{5}{p}(d + \log P)$ steps are used is less than $\frac{1}{P}$.*

The CWA problem The problem of providing a synchronization primitive is well abstracted by the CWA problem: "given an array $A[1..n]$ and a flag f, all initialized to 0, set all entries of A to 1 and then set f to 1."

Lemma A.4 *[MS91] The CWA problem can be solved in $E\Theta(n + p \log n)$ work using p processors.*

A.1 C-Circuits

We define a class of highly structured computations, which can be represented by C-Circuits. A C-Circuit is a directed acyclic graph with two types of nodes, *data nodes* and *computation nodes.* A subset of the data nodes are *input nodes* which have indegree zero, and *output nodes* which have outdegree zero. All arcs in the graph go from data nodes to computation nodes or from computation nodes to data nodes. Each data node, input and output nodes excepted, has arbitrary indegree and outdegree. Each computation node has bounded indegree and outdegree. Each computation node N has associated with it a constant length sequence of RAM instructions. These instructions read from the data nodes which have arcs directed into N. They write a single value to each data node which has an edge directed from N.

The *size*, S, of a C-Circuit is the number of computation nodes. The *depth*, D, is the longest path from an input to an output. We provide an intuitive description of the *width* of a circuit. Initially, we consider the inputs of the circuit as its *data items.* As a computation node in the circuit is evaluated, the data items are said to move from the inputs of the computation node to its outputs. If the fan-in is greater than the fan-out, data items are said to be destroyed. If the fan-in is smaller, data items are created. The width of a is the maximum number of data items present at any time, for any legal order of evaluation.

Lemma A.5 *[C-Circuit Lemma] [SM90] Any C-Circuit of width $O(n)$ and depth $d > \log n$ can be evaluated on a TPRAM using $EO(nd)$ work and up to n processors i.e., with no asymptotic loss in work or time compared to a PRAM evaluation.*

Note. Lemma A.5 holds for circuits whose depth D is a random variable such that $E[D] = \frac{d}{2}$ and $P[D > kd] < \frac{1}{2^k}$ It might appear that Lemma A.5 is inapplicable for circuits with depth $o(\log n)$ (eg., $O(\log^* n)$ depth circuit for a 2-ruling set [CV86a]). However, it is still useful because one can concatenate many shallow circuits so that the resultant circuit has the necessary depth.