[MPS89] Charles Martel, Arvin Park, and Ramesh Subramonian. Optimal asynchronous algorithms for shared memory parallel computers. Computer Science and Engineering CSE-89-8, University of California at Davis, July 1989.

[MS90] Charles Martel and Ramesh Subramonian. Asynchronous pram algorithms for list ranking and transitive closure. UCDavis, manuscript, January 1990.

[MSP90] Charles Martel, Ramesh Subramonian, and Arvin Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science*, 1990.

[Nis90] Naomi Nishimura. Asynchronous shared memory parallel computation. In *Proceedings of 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 76–84, 1990.

[PF77] Gary L. Peterson and Michael J. Fischer. Economical solutions for the Critical Section Problem in a distributed system. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 91–97, May 1977.

[PU87] Christos H. Papadimitriou and Jeffrey D. Ullman. A commmunication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, August 1987.

[PY88] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–513, 1988.

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

[Wyl79] James C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell University, August 1979. Technical report number TR 79-387, Department of Computer Science.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[CV86] Richard Cole and Uzi Vishkin. Deterministic Coin Tossing with Applications to Optimal List Ranking. *Information and Control*, 70(1):32–53, 1986.

[CV87] Richard Cole and Uzi Vishkin. Approximate Parallel Scheduling. Part II: Application to Optimal Parallel Graph Algorithms in Logarithmic Time. Technical Report 291, New York University, 1987. To appear, Information and Computation.

[CV88] Richard Cole and Uzi Vishkin. Approximate Parallel Scheduling. Part I: The Basic Technique With Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM Journal on Computing*, 17(1):128–142, 1988.

[CZ89] Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[CZ91] Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In Preparation, 1990.

[CZ90b] Richard Cole and Ofer Zajicek. Asynchronous graph connectivity in $O(\log n)$ rounds. Manuscript, 1990.

[CZ90c] Richard Cole and Ofer Zajicek. The expected advantage of asynchrony. In *Proceedings of 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 85–94, 1990.

[Gib89] Phillip B. Gibbons. Towards better shared memory programming models. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[KRS88a] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 333–346. Springer-Verlag, July 1988.

[KRS88b] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC 13572, International Business Machines, 1988.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LF81] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

the RAM and PRAM models. It is introduced in order to study the synchronization costs of shared memory parallel computation, costs which are hidden by the synchronous PRAM model.

The APRAM does not provide a global clock by which all the processes can synchronize. Instead, the required synchronization must be coded into the algorithm. We have shown that careful design may lead to efficient asynchronous algorithms; one method demonstrated in this paper replaces global synchronization by local synchronization.

When a PRAM algorithm is run on an asynchronous machine, it is necessary to syncronize all the processes after each step. We have shown that this synchronization may increase the rounds complexity of the algorithm by a multiplicative factor of $O(\log n)$, where $n$ is the size of the input. Using an asynchronous model, such as the APRAM, favors algorithms with fewer synchronization requirements yielding algorithms which can be run faster in asynchronous environments.

# References


[AC88] Alok Aggarwal and Ashok K. Chandra. Communication complexity of PRAMs. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 1–17. Springer-Verlag, July 1988.

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[AG87] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, IT-33(3):315–322, May 1987.

[AM88] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In *Proceedings of the 3rd Aegean Workshop on Computing*, number 319 in Lecture Notes in Computer Science, pages 81–90. Springer-Verlag, 1988.

[Awe85] Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 230–240, May 1987.

**Theorem 4.2** *Given $n$ elements in an array, forming a collection of linked lists, the recursive doubling algorithm computes for each element the end of its list using $n$ processes in* $O(\log n)$ *rounds.*

The two algorithms analyzed above are simple and straightforwart. A companion paper, [CZ90b], presents an $n + e$ process, $O(\log n)$ rounds APRAM algorithm for computing the connected components of an undirected graph with $n$ vertices and $e$ edges; this algorithm is substantially different from the known PRAM algorithms, and its analysis is quite intricate. We remind the reader that the fastest PRAM algorithms for graph connectivity run in $O(\log(n + m))$ time on a CRCW PRAM. In Shiloach and Vishkin [SV82], $n + m$ processes are used; in Cole and Vishkin [CV87] this is reduced to $(n+m)\alpha(m, n)/\log(n+m)$ processes, but at the cost of large constants resulting from the presence of expander graphs.

# 5 APRAM Simulations

The tree computation used in the summation algorithm, as well as the list based recursive doubling algorithm, can be used to execute a barrier-like synchronization among the processes in $O(\log n)$ rounds. From this it follows that:

**Theorem 5.1** *Any PRAM algorithm, $A$, with parallel running time $T(n)$ using $P(n)$ processes can be simulated by an APRAM algorithm which uses $P(n)$ processes and has round complexity* $O(T(n)\log n)$.

By performing a simulation on $P' \leq P(n)/\log P(n)$ processes, and performing the barrier-like synchronization every $\log(P(n)/P')$ rounds, one can obtain:

**Theorem 5.2 (Optimal Simulation)** *Any PRAM algorithm, $A$, with parallel running time $T(n)$ using $P(n)$ processes can be simulated by an APRAM algorithm which uses $P'$ processes, $P' \leq P(n)/\log P(n)$, and has round complexity* $O(T(n)P(n)/P')$.

For any PRAM algorithm, $A$, the APRAM algorithm obtained by inserting a barrier-like synchronization after each statement is called *the APRAM simulation of $A$*. The multiplicative factor of $O(\log n)$, introduced by the simulation, can often be avoided, as we have already seen for the summation and recursive doubling problems.

# 6 Conclusions

This paper gives a formal definition of the APRAM model and its associated complexity measure, the rounds complexity. The APRAM model is an asynchronous generalization of

**Theorem 4.1** *There is an APRAM algorithm which computes the sum of $n$ elements given in an array in $O(\log n)$ rounds using $n-1$ processes.*

The above algorithm can easily be modified to use only $n/\log n$ processes while achieving the same asymptotic bound. It is also straightforward to extend this result to obtain a parallel prefix algorithm with the same complexity.

### 4.3 Recursive Doubling

The recursive doubling algorithm takes as input $n$ elements given in an array. Each element, $u$, has a pointer, $p(u)$, which points to its successor, an element in the array. An element is said to be the head of a list if $p(u) = u$; it is assumed that $p$ does not have non-trivial cycles. The recursive doubling algorithm is the heart of most, if not all, of the known parallel list ranking algorithms [Wyl79,AM88,CV86,CV88,MS90]. We do not present a list ranking algorithm here, for it involves questions regarding atomicity which we do not intend to discuss presently.

The APRAM algorithm for recursive doubling requires no synchronization and is identical to the simple PRAM algorithm due to Wyllie [Wyl79].

Algorithm for process $i$

```
1   while (p(p(i)) ≠ p(i)) do
2         p(i) := p(p(i))
```

Statement 1 checks for termination and Statement 2 updates the value of $p(v)$. The correctness of the algorithm follows from the invariant that for any element, $u$, $p(u)$ is an element that was an ancestor of $u$ in the input list.

For any two elements, $u$ and $v$, define the *distance between $u$ and $v$* to be the number of edges that must be traversed in the input list to get from $u$ to $v$. Let $C$ be any computation and define a superevent to comprise a complete iteration of the while loop. For any two elements, $u$ and $v$, of the list and any number, $i$, let $p_i(u)$ be the successor of $u$ after round $i$ and let $D(u,v)$ be the distance from $u$ to $v$. The recursive doubling algorithm maintains the invariant that for any number $i$ and for any element $u$, if after round $i$ the successor of $u$ is not a head of a list, then

$$D(u, p_{i+1}(u)) \geq D(u, p_i(u)) + D(p_i(u), p_i(p_i(u))).$$

Since $D(u, p_0(u)) = 1$, after round $i$ every element which does not point to the end of its list points to an element at distance at least $2^i$. We conclude:

for $r$ a constant is $O(R)$ rounds. We ignore constant factors, as is often done in asymptotic analysis, and henceforth use superevents and superrounds. When no ambiguity arises, we use the term round to refer to a superround.

The complexity of an APRAM algorithm is measured in terms of the pair [round complexity, number of processes]. This is intended to correspond to the measure [time, processes] used for the PRAM model. The aim is that this measure of complexity should correspond roughly to a [time, processors] complexity measure on more realistic machines. The notion of rounds is far from new; it is used extensively when analyzing distributed (asynchronous) algorithms; however, in distributed algorithms, typically, the other component of the complexity is the number of messages transmitted. We are interested in a more tightly coupled form of processing, which is characteristic of parallel computation. We note that the virtual clock is used solely for the round analysis; the correctness arguments must prove the algorithm correct regardless of the division into segments.

We illustrate the round complexity by means of two simple examples: computing the sum of $n$ integers given in an array and a recursive doubling algorithm.

### 4.2 Summation Algorithm

The APRAM summation algorithm given below is very similar to the known PRAM algorithm. It uses $2n-1$ memory cells arranged in an implicit complete binary tree with the input elements at the leaves. A process is associated with each internal node of the tree. Each memory cell has a valid bit which is initialized to true for all the input elements (the leaves) and false for all internal nodes. For each process, $i$, $V_i$ is the internal node associated with $i$, and $L_i$ and $R_i$ are the left child and right child of $V_i$, respectively.

The algorithm iterates the following superevent comprising a condition test plus a possible execution of the if statement.

Algorithm for process $i$:

```
1   while (V_i is not valid) do
2       if R_i and L_i are valid then
3           set V_i := L_i + R_i
4           set the tag of V_i to valid
5       end if
6   end while
```

Once both children of a node are valid the process associated with the node executes only one more iteration, during which its associated node is validated. As the depth of the tree is $\lceil \log_2 n \rceil$, and each process executes at least one iteration at each round, the number of rounds required is $\lceil \log_2 n \rceil$. We have shown:

are presented in [CZ91].

Let $C$ be any computation and $\mathcal{T}$ any virtual clock of $C$. For each event, $e$, the time of $e$, $t(e)$, is the time assigned to $e$ by $\mathcal{T}$. We call the integer part of the time of $e$ its round number, $rd(e)$, $rd(e) = \lfloor t(e) \rfloor$. The virtual clock in effect divides the computation into contiguous segments, called rounds, so that each segment contains at least one event from each process. A round is called *effective* if it has at least one effective event.

The *rounds complexity* of a computation is the number of effective rounds in a virtual clock of the computation maximized over all possible virtual clocks. The rounds complexity of an algorithm for a given input, $I$, is the maximum rounds complexity over all possible computations on input $I$. The rounds complexity of an algorithm for inputs of size $n$ is then defined to be the maximum rounds complexity over all inputs of size $n$. Note that each time step of a synchronous computation is a round which contains exactly one event from each process.

We wish to stress that our model refers to processes and not processors. Brent's theorem [Bre74] justifies the use of processes, when designing PRAM algorithms, without regard to the actual number of physical processors at hand. An analogue of Brent's theorem applies to the APRAM model:

**Lemma 4.1** *A $t$ round, $p$ process algorithm can be simulated using $q \leq p$ processes in $O(tp/q)$ rounds.*

**Proof.** The processes of the $p$ process algorithm, $A_p$, are distributed evenly among the $q$ processes of the simulating algorithm, $A_q$. Each process of $A_q$ receives either $\lceil p/q \rceil$ or $\lfloor p/q \rfloor$ processes of $A_p$. Each process of $A_q$ simulates its assigned processes in round robin fashion. Consider a computation, $C$, of $A_q$. There is a one to one mapping from the events of $C$ to a computation of $A_p$. Partition $C$ into phases of $\lceil p/q \rceil$ rounds. Clearly, each phase is a round of $A_p$. Hence, $A_q$ uses at most $t\lceil p/q \rceil$ rounds. □

So far we have defined the model and the complexity measures in terms of events. Requiring this low level detail may make the design process tedious and the analysis unduly complicated. Instead, we group statements into larger constructs called *superevents*. A process, $p$, is said to execute a superevent, $E$, if the events comprising $E$ appear as a segment of $p$. We allow each superevent to contain up to $r$ events, for some fixed number, $r$, independent of the input. We then define a *superround* to be a segment in which each process executes at least one superevent.

Every segment comprising $2r - 1$ rounds contains at least one superevent. Therefore, an algorithm which has complexity $O(R)$ superrounds has complexity $O(rR)$ rounds, which

# 4 The Rounds Complexity Measure

The PRAM is a synchronous model. In using the PRAM as a programming model, one in effect assumes the existence of a global clock to which all the processes synchronize. It is natural to use this clock as a measure of algorithmic complexity. The APRAM does not assume the existence of a global clock; therefore, we need complexity measures that can replace the running time complexity of synchronous models. These new measures should reflect the elapsed real time from the start of the algorithm until its termination if it were implemented in an asynchronous environment.

For any PRAM algorithm, $A$, let $T(n)$ and $P(n)$ denote the parallel running time and the number of processes used by the algorithm on inputs of size $n$. The number of effective events in any synchronous computation of $A$ on inputs of size $n$ is at most $P(n) \cdot T(n)$. In this respect, the length of a synchronous computation gives some indication of the algorithm's complexity. This need not be the case for asynchronous algorithms. For example, if one process, $P_1$, is assigned to set a global variable, $x$, (which is initially reset), and all other processes wait for the variable to be set (e.g. by repeatedly checking the value of $x$) one can create arbitrarily long computations by delaying the step of $P_1$ that sets $x$. One can view the frequency of events from a process $P$ in a segment of a computation as the process's *speed*. This example shows that it is possible for the length of a computation to increase when the speed of a subset of the processes is increased.

## 4.1 The Virtual Clock

The running time, or parallel time, complexity used in analyzing PRAM algorithms corresponds to the number of time steps in a synchronous computation. One approach in asynchronous models is to use a *virtual clock* (or *logical clock* [Lam78]). This approach was introduced in [PF77] and used in [AFL83,LF81] and is common in the area of distributed computing (see [Awe87,Awe85,AG87]). Consider a computation, $C$. A *virtual clock of* $C$ is an assignment of unique virtual times to the events of $C$; the times assigned are a non-decreasing function of the event number. The time difference between two consecutive events of a process is called the *duration* of the later event. The complexity of a computation is the time assigned to the last effective event in the computation.

One can obtain variations in the complexity measure by restricting the allowable virtual clocks. One such variation, called the *rounds complexity*, requires the duration of each event to be at most 1. In effect, it assumes that each operation takes at most one unit of time, but is allowed to take less. Other variations for measuring the implicit costs of synchronization

**T4.** $e_2$ is an event in the time step immediately following the time step containing $e_1$.

**T5.** $e_1$ is executing a Load operation and $e_2$ is executing a Store operation to the same address and $e_1$ and $e_2$ are in the same time step.

Conditions T4 and T5 correspond to the synchronization implicit in the PRAM model. Two distinct events, $e_1$ and $e_2$, are called *concurrent* if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$.

We say that an event *accesses* a variable, $x$, if it either writes to $x$ or reads from $x$. A computation is said to be *exclusive write* (resp. *exclusive read*) if for any variable, $x$, the events writing to (resp. accessing) $x$ are totally ordered with respect to the relation $\rightarrow$. A program is called *exclusive write* (resp. *exclusive read*) if every computation of the program is exclusive write (resp. exclusive read). A program is called *synchronously exclusive write* (resp. *synchronously exclusive read*) if every synchronous computation is exclusive write (resp. exclusive read). The CREW PRAM (resp. the EREW PRAM) is a synchronously exclusive write (resp. synchronously exclusive read) RAM.

When using the CRCW PRAM model one must define how write conflicts are resolved. Several submodels appear in the literature; the more popular ones are the *Priority*, *Arbitrary* and *Common*. In the Priority model, each process is assigned a priority. When several processes attempt to write to the same memory location at the same time the process with highest priority succeeds. In the Arbitrary model, when several processes attempt to write into the same memory location at the same time one of them succeeds but it is not known which; this introduces non-determinism to the computation. In the Common model, processes are allowed to write simultaneously to the same location only if they all write the same value.

Define the *synchronous APRAM* to be an APRAM restricted to synchronous computations. The same variations can be carried over to the synchronous APRAM model. For the synchronous APRAM priority model, each process is assigned a priority. A synchronous APRAM computation is called *Prioritized* if for any two events, $e_1$ and $e_2$, which write to the same location in the same time step, if the priority of the process executing $e_1$ is higher than that of the process executing $e_2$, then $e_1$ appears after $e_2$ in the computation. For the common model, we require that for any two events, $e_1$ and $e_2$, if they both write to the same location in the same time step, they write the same value.

**Remark.** We are not showing how to implement the various PRAM models in the APRAM model. Rather, we are showing that appropriate restrictions on the APRAM model yield the various PRAM models.

replace $e_i$ and $e_{i+1}$, respectively. $C_k$ is called a *process of* $C$. It is convenient to assume that the identifiers of the processes of a computation with $p$ processes are numbered from 0 to $p-1$. A process is said to have terminated at event $e$, if $e$ is the last effective event of the process. Note that for any process, $P$, and any event $e \in P$, if $op(e) = terminate$ then for any $e' \in P$ such that $e' \geq e$, $op(e') = terminate$.

A *synchronous computation* is a parallel computation which can be divided into contiguous segments called *time steps*, satisfying:

1. Each segment contains exactly one event from each process.

2. If there are two events in a segment, one executing a load (read) operation and the other a store (write), both using the same address, then the event performing the load operation precedes the event performing the store operation.

The *sequential* RAM is equivalent to a RAM restricted to sequential computations. For parallel computations, an algorithm description specifies the number of processes used; this can be a function of the input. If an algorithm is defined to use $p$ processes this is equivalent to the restriction that a computation is valid only if it has at most $p$ processes. The PRAM is equivalent to a RAM restricted to synchronous computations. We define the APRAM to be a RAM restricted to parallel computations. We note that the PRAM is a restriction of the APRAM.

### 3.3 Parallel Submodels

The PRAM model is subdivided into submodels which differ in their restrictions on access to shared memory. Corresponding subdivisions can be defined in the APRAM model.

Consider any computation, $C$. The *executed before* relation defined below captures the notion that the execution of an event had an effect, direct or indirect, on the execution of another event. For any event, $s$, let $P(s)$ denote the process to which $s$ belongs. For any two events, $e_1$ and $e_2$, $e_1 \rightarrow e_2$, if one of the following holds:

**T1.** $P(e_1) = P(e_2)$ and $e_1$ occurs before $e_2$ in $S$.

**T2.** Event $e_2$ executes a load operation with address $a$, and $e_1$ is the most recent event in $S$ which executes a store with address $a$.

**T3.** There is an event, $e_3$, for which $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$.

For synchronous computations, in addition to the above three conditions, $e_1 \rightarrow e_2$ if

where $V$, $A$, and $B$ are variables corresponds to four events executing $\langle Load, A\rangle$, $\langle Load, B\rangle$, $\langle Local\rangle$ and $\langle Store, V\rangle$. The arithmetic operation, in this case the addition of $A$ and $B$, could be computed as part of the Store event, but it is usually more convenient to consider it to be performed by a separate event executing a local operation.

Programs are designed to solve problems. A problem definition comprises a class of inputs and a desired outcome; the outcome can be thought of as a property. For instance, an algorithm that sums all the elements in the input can be defined to accept as its input an array of integers. We can define a computation of the algorithm to be correct if the last event that writes to address 0 writes the sum of all the input elements.

More formally, for any computation, $S$, and any property $F$, *property $F$ holds for $S$* if there is some event $e \in S$ such that for any event $e'$ subsequent to event $e$ property $F$ holds after $e'$.

A RAM *algorithm* is defined to be a RAM program, $N$, together with a class of allowable inputs, $\mathcal{I}$, a desired property, $F$, and a class of allowable computations, $\mathcal{C}$. For instance, one natural restriction on the computations is that the initial events should have their current state equal to some initial value (the specification of initial events is made precise later). An algorithm is said to be *correct* if for any input, $I \in \mathcal{I}$, property $F$ holds for every computation, $C \in \mathcal{C}$, of $N$ on input $I$.

In the following sections we define a number of common RAM submodels by specifying their corresponding classes of allowable computations.

### 3.2 The RAM Family

The RAM has a predefined initial state called $InitState$ and a predetermined initial value for the communication register called $InitComm$ (e.g. 0). A *sequential computation* is a computation whose events are chained. More formally, a computation is sequential if for any $i \geq 1$:

**(UNIQ)** $Id(e_{i+1}) = Id(e_i)$.

**(SRAM-I)** $CurState(e_1) = InitState$, and $Comm1(e_1) = InitComm$.

**(SRAM-C)** $CurState(e_{i+1}) = NewState(e_i)$ and $Comm1(e_{i+1}) = Comm2(e_i)$.

Next, we define several parallel RAM models. For any computation, $C$, and for any number, $k$, let $C_k$ be the subsequence of $C$ containing all the events with $Id = k$; so $C_k = e_{k_1}, e_{k_2}, \ldots$. A computation, $C$, is called a *parallel computation* if for any $k$, $C_k$ satisfies (SRAM-I) and (SRAM-C), where in SRAM-I $e_{k_1}$ replaces $e_1$, and in SRAM-C $e_{k_i}$ and $e_{k_{i+1}}$

one of the four RAM operations named above. We note that the operation $Op$ may comprise several fields; for example, the load operation requires an address field. The values of the required fields are specified by the program. Constraints on $Comm2$ will be specified subsequently. For any event, $e$, let $Id(e)$, $CurState(e)$, $Comm1(e)$, $Op(e)$, $NewState(e)$, and $Comm2(e)$ be the corresponding members of the event tuple.

A RAM computation is defined to comprise an input and a sequence of events satisfying two constraints. First, the value returned by a load operation is the most recent value stored in the location addressed, if any. Otherwise, the value returned is the initial value assigned by the input to that location. More formally, a computation is a tuple comprising a partial mapping, $I : addr \mapsto value$, called the *input*, and an infinite sequence of events, $e_1, e_2, \ldots$ satisfying the following for any $i \geq 1$:

**(RAM)** If $Op(e_i) = \langle Load, addr_i \rangle$ then let $j$ be the largest number smaller than $i$, if any, for which $Op(e_j) = \langle Store, addr_i, data \rangle$ for some value $data$. If there is such a $j$ then $Comm2(e_i) = data$, otherwise $Comm2(e_i) = I(addr_i)$.

If $Op(e_i)$ is not a load operation, $Comm2(e_i) = Comm1(e_i)$.

This specifies the semantics of the load and store operations.

Second, the terminate operation indicates that the RAM terminated and its semantics is as follows. For any RAM program, $N$, any state, $s$, any communication register value, $c$, and any identifier, $id$, if $N(id, s, c) = \langle terminate, s' \rangle$ then $s = s'$. In other words, for any event, $e$, if $Op(e) = terminate$ then $NewState(e) = CurState(e)$. An event that executes a terminate operation is called a *terminate* event, all other events are called *effective* events.

For any computation and any two events, $e$ and $e'$, in the computation, let $e' > e$ denote that $e'$ is subsequent to $e$. A computation, $C$, is said to have *terminated at* $e$ if $e \in C$, $e$ is an effective event, and for any event $e' \in C$ such that $e' > e$, $e'$ is a terminate event.

We consider RAM models which restrict the size of the state block and the size of the communication register. First, we assume that the state register is finite and that the state block has a fixed number of computation registers of $c \log n$ bits each, where $c$ is a constant and $n$ is the size of the input. In addition, the communication register is assumed to have $c \log n$ bits; this limits the number of bits that can be read from memory in a single operation.

In algorithm design it is common to specify the program using a high level language such as Pascal. The state register would then correspond to the program counter. Reading a variable translates to a load operation while writing a variable translates to a store operation. Arithmetic and logical operations translate to local RAM operations. A statement such as

$$V := A + B$$

complexity of Section 4 with a probabilistic analysis: the duration of an event is assumed to be a random variable with a known probability distribution.

# 3 The APRAM

The random access machine, or the RAM, is the standard model for sequential computation. It is therefore natural to model parallel computation by extensions of the RAM. One such extension, the PRAM, has become a widely used model for parallel computation. In this thesis we consider another generalization of the RAM model. The generalization incorporates both the standard RAM model and the PRAM model, as well as the APRAM model, the subject of this research. Henceforth the standard RAM model is called the *sequential* RAM; the term RAM is reserved for the new generalized model.

## 3.1 The (Generalized) RAM

A computation model is an abstraction whose purpose is to model our notion of a computer. Intuitively, a computer is a machine with memory and registers, which when presented with a program performs actions. The actions may fetch a value from memory, modify a memory location or perform a calculation on the machine's registers. With this in mind, we view the RAM as a model which when presented with a *program* and an initial *input* produces computations; this is made more precise shortly. The RAM comprises a *state register*, several *computation registers*, a *communication register* and an *address space*. The state register together with the computation registers form a RAM *state*.

The RAM performs actions which are described in terms of four basic operations: A *load* operation, $\langle Load, addr\rangle$, a *store* operation, $\langle Store, addr, data\rangle$, a *local* operation, $\langle Local\rangle$, and a *terminate* operation, $\langle Terminate\rangle$. The address space corresponds to the computer memory. The load operation corresponds to a fetch from memory and a store operation corresponds to a write to memory. A local operation modifies the state of the RAM but does not affect the address space (i.e. the memory). The terminate operation is a special local operation which indicates that the RAM has terminated. The details of these operations as well as the purpose of the various registers is made precise shortly.

The computation of a RAM is a sequence of *events* defined by a *program*. A program is a total mapping, $N : \langle Id, State, Comm1\rangle \mapsto \langle Op, NewState\rangle$. An *event* is any tuple, $\langle Id, CurState, Comm1, Op, NewState, Comm2\rangle$, satisfying $N(Id, CurState, Comm1) = \langle Op, NewState\rangle$, where $Id$ is an integer called the *event identifier*, $CurState$ and $NewState$ are RAM states, $Comm1$ and $Comm2$ are values of the communication register, and $Op$ is

in this paper, we only discuss the rounds complexity measure.

In Section 4 we define the rounds complexity; it replaces the global clock used by the PRAM model by a *virtual clock*. This approach was introduced in [PF77] and used in [AFL83,LF81,KRS88b] and is common in the area of distributed computing (see [Awe87], [Awe85,AG87]). Consider a computation, $C$. A *virtual clock of* $C$ is an assignment of unique virtual times to the events of $C$; the times assigned are a non-decreasing function of the event number.

The virtual clock is meant to correspond to the "real" time at which the operations occurred in one possible execution of the algorithm, called a computation. The time difference between two consecutive events of a process is called the *duration* of the later event. The length of a computation is the time assigned to the last event in the computation.

The rounds complexity of an algorithm is the length of a computation maximized over all possible computations; i.e., maximized over all possible interleavings of the events of the processes. We assume that the duration of events is at most one. In effect, in the rounds complexity measure the slowest process defines a unit of time (a round); the complexity is expressed in rounds. Although this is inadequate for measuring the implicit costs of synchronization, using the rounds complexity we are able to measure the explicit costs of synchronization.

We analyze two algorithms under the rounds complexity measure: a tree based parallel summation algorithm and a list based recursive doubling algorithm. Both are fundamental algorithms and appear as subroutines in many parallel algorithms. We show that both algorithms have complexity $O(\log n)$ rounds, comparable to their PRAM parallel run time complexity (assuming a linear number of processes). Recall that one round of the virtual clock roughly corresponds to one parallel time step of the PRAM model.

In a companion paper we describe and analyze a more substantial algorithm; an algorithm for finding the connected components of an undirected graph [CZ90b]. This algorithm differs substantially from all known PRAM algorithms. We avoid the need to synchronize the processes, thereby obtaining an algorithm whose behavior appears somewhat chaotic. The description of the algorithm is relatively simple and straightforward; however, due to its apparently chaotic nature and the unpredictability of the asynchronous environment, its analysis is quite challenging. We show that the rounds complexity of the algorithm is $O(\log n)$ rounds assuming a linear number of processes.

In a second companion paper we study the implicit costs of synchronization by allowing the duration of events to be larger than 1 [CZ91]. We achieve this by generalizing the definition of a virtual clock due to [PF77,AFL83,LF81]. We also replace the worst case

spatial locality of reference. This approach implicitly assumes that the (virtual) machine is uniform, comprising a collection of similar processes. In addition to latency to memory, Gibbons [Gib89] studied the cost of process synchronization by considering an asynchronous model. Although he assumed that the machine is asynchronous, his analysis in effect assumes that the processes are roughly similar in speed. More precisely, he required that read and writes be separated by global synchronization; this results in a uniform environment from the process perspective.

## 2 Overview

The RAM is a very popular model in sequential computation and algorithm design. Therefore, it is not surprising that a generalization of the RAM, the PRAM, became a popular model in the field of parallel computation. We use another generalization of the RAM model, called the APRAM, suggested by Kruskal et al. [KRS88a]; a formal definition of the APRAM model is given in Section 3. It includes the standard PRAM as a submodel. Roughly speaking, the APRAM can be viewed as a collection of processes which share memory. Each process executes a sequence of basic atomic operations called *events*. An event can either perform a computation locally (by changing the state of the process) or access the shared memory; accessing the memory has a unit cost associated with it. An APRAM computation is a serialization of the events executed by all the processes.

Synchronization costs fall into two categories: *explicit costs* and *implicit costs*. By explicit costs we mean the overhead for achieving synchronization. This could be, for instance, the cost of executing extra code that must be added to the algorithm in order to synchronize. When processes proceed at different speeds, if the algorithm is required to proceed in lock step, the time required to execute a step is dictated by the slowest process. By implicit costs we refer to the cost associated with lock step execution apart from the explicit costs of synchronization. We do not define the notion of explicit and implicit costs formally, for we do not think we have enough experience to justify definitive definitions. Rather we present them as concepts which are exemplified in our analysis.

The parallel runtime complexity associated with the PRAM model describes the complexity of PRAM algorithms as a function of the global clock provided by that model. The APRAM model does not have such a clock. Consequently, it is necessary to define complexity measures to replace the running time complexity. We replace the role of the global clock by a virtual clock and we eventually investigate three complexity measures: the rounds complexity, the unbounded delays complexity and the bounded delays complexity; however,

The PRAM is a synchronous model and thus it strips away problems of synchronization. However, the implicit synchronization provided by the model hides the synchronization costs from the user. In many cases, an algorithm may have to be redesigned in order to allow it to run efficiently in an asynchronous environment. In this paper, we are concerned with the design of algorithms which perform well in the presence of the non-uniformity introduced by asynchrony. The work of Nishimura [Nis90] is similarly motivated. Another approach to the problem of asynchrony is to seek to efficiently compile PRAM algorithms to operate in asynchronous environments; this approach is followed by Martel, Park and Subramonian [MPS89,MSP90]. They give efficient randomized simulations of arbitrary PRAM algorithms on an asynchronous model, given certain architectural assumptions (e.g. the availability of a compare&swap instruction). It is not clear whether similar deterministic compilers exist; in the absence of such compilers, to obtain deterministic algorithms it appears necessary to design them in an asynchronous environment; this is the focus of our work.

Our algorithm design is targeted at the APRAM model, defined more precisely in Section 3. It assumes that each process can access each memory location in unit time. A nonuniform environment arises because processes may proceed at considerably different speeds; in addition, the same process may proceed at different speeds at different times during the execution of the algorithm. We study the effects of the nonuniformity on the complexity of algorithms, aiming to design algorithms which perform well in such environments.

Even if the underlying hardware is uniform and synchronous, the environment may appear nonuniform at the process level for several reasons. For example, it may appear nonuniform due to multitasking and consequent nonuniform task or process distribution, or because of interactions with external devices (e.g. interrupts to service disk I/O).

We hope that investigating the synchronization costs will further our understanding of some of the issues involved in parallel computation. The better we understand the issues involved the more likely we are to develop better abstract computational models, which will ultimately guide us in taking advantage of the power of parallelism.

Now, we briefly discuss some of the work concerned with uniform environments. [PU87], [PY88,AC88] show that the communication among the processes can be reduced by designing algorithms which take advantage of temporal locality of access; they assumed that each (global) memory access has a fixed cost associated with it. In a related paper [ACS89], Aggarwal et al. argue that typically it takes a substantial time to get the first word from global memory, but after that, subsequent words can be obtained quite rapidly. They introduce the BPRAM model which allows block transfers from shared memory to local memory. They show that the complexity of algorithms can be reduced significantly by taking advantage of

# The APRAM – The Rounds Complexity Measure and the Explicit Costs of Synchronization*

Richard Cole Ofer Zajicek

New York University

**Abstract**

This paper studies the explicit costs of synchronization by examining an asynchronous generalization of the PRAM model called the APRAM model. The APRAM model and its associated complexity measure, the rounds complexity, are defined and then illustrated by designing and analyzing two algorithms: a parallel summation algorithm which proceeds along an implicit complete binary tree and a recursive doubling algorithm which proceeds along a linked list. In both cases replacing global synchronization with local synchronization yields algorithms with reduced complexity.

## 1 Introduction

In this paper we consider the effect of process asynchrony on parallel algorithm design. As is well known, the main effort in parallel algorithm design has employed the PRAM model. This model hides many of the implementation issues, allowing the algorithm designer to focus first and foremost on the structure of the computational problem at hand – synchronization is one of these hidden issues.

This paper is part of a broader research effort which has sought to take into account some of the implementation issues hidden by the PRAM model. Broadly speaking, two major approaches have been followed. One body of research is concerned with asynchrony and the resulting non-uniform environment in which processes operate[1] [CZ89,CZ90c,Nis90,MPS89], [MSP90]. The other body of research has considered the effect of issues such as latency to memory, but assumes a uniform environment for the processes [PU87,PY88,AC88,ACS89], [Gib89].

*The work was supported in part by NSF grants CCR-8902221 and CCR-8906949, and by a John Simon Guggenheim Memorial Foundation Fellowship.

[1] We distinguish between processes and processors in order to emphasize that the APRAM is not a machine model but rather a programming model; it is the task of a compiler to implement the programming model on actual machines. The term processor will be used to refer to this component of a machine.