SOLVING INTRACTABLE PROBLEMS BY RANDOMIZED STEPS

and

CLOCK CONSTRUCTION IN FULLY ASYNCHRONOUS PARALLEL SYSTEMS AND PRAM SIMULATION

M O RABIN

Rapporteurs: Ann Petrie and John S Fitzgerald

Clock Construction in Fully Asynchronous Parallel Systems and PRAM simulation

(Extended Abstract)

Yonatan Aumann* Michael O. Rabin†

Abstract

In this paper we return to the question of simulating synchronous computations on asynchronous systems. We consider an asynchronous system with very weak, or altogether lacking any, atomicity assumptions.

The first contribution of this paper is a novel *clock* for asynchronous systems. The clock is a basic tool for synchronization in the asynchronous environment. It is a very robust construction and can operate in a system with no atomicity assumptions, and in the presence of a *dynamic scheduler*. The behavior of the clock is obtained with *overwhelming probability* ($1 - 2^{-\alpha n}$, $\alpha > 0$).

We then show how to harness this clock to drive a PRAM simulation on an asynchronous system. The resulting simulation scheme is more efficient then existing ones, while actually relaxing the assumptions on the underlying asynchronous system.

1 Introduction

Parallel algorithms and programs are most commonly designed and described for systems of tightly coupled processors working in almost complete synchrony. A typical example of such a system is the PRAM, in which all processors work step by step in complete synchrony. In less extreme models (e.g. the BSP model [Val90]), synchronization is not assumed to exist at each and every step, but is still an indispensable ingredient of the overall structure. Synchronization assumptions are convenient from the program development point of view as they free the programmer from the need to consider actual processor and network timings and let him or her focus on the major task of parallelization. These assumptions do not however correspond to the way actual parallel systems operate. Typically, processors working on separate parts of the same program may do so asynchronously and at considerably different speeds, for a multitude of reasons: interrupts, context switches, network congestion, page faults, etc..

Handling asynchrony has thus attracted much research activity and is the topic of a large body of work. One important issue is how to simulate the execution of a PRAM program on an asynchronous parallel system, but even implementing particular algorithms in an asynchronous setting is a challenging task. In this paper we focus on asynchronous systems with shared memory. Previous work regarding this setting typically assumes some sort of atomicity in carrying out basic instructions. A minimalistic assumption would be that single reads and writes are atomic, and more frequently it is assumed that some compound instruction of the form "read & write" (e.g. *test & set, fetch & add, compare & swap*) is atomic. Bootstrapping on such atoms, consensus protocols and synchronization mechanisms are then developed. Carrying out the more complex computations, such as PRAM simulations, generally required the stronger primitives. In fact, Herlihy [Her88] describes a full hierarchy of atomicity assumptions, and proves that atoms of a higher class cannot be implemented by those of a lower class, in a wait-free fashion in the deterministic setting. In particular, complex computations require the most powerful atoms such as *compare & swap* (see also [Her91]). Recently, Palem, Kedem, Rabin and Raghunatan [KPRR92] gave for the first time a PRAM simulation scheme on an asynchronous PRAM for which only individual reads and writes are assumed to be atomic.

*Institute of Mathematics and Computer Science, The Hebrew University of Jerusalem, Israel; aumann@cs.huji.ac.il

†Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138, USA, and The Institute of Mathematics and Computer Science, The Hebrew University of Jerusalem, Israel; rabin@das.harvard.edu, rabin@cs.huji.ac.il. The research of this author was supported in part by ONR contract number N0001491-J-1981, and NSF grant number CCR-90-07677, at Harvard University.

Appeared in 33rd annual Symposium on Foundations of Computer Science, October 25-27, 1992, Pittsburg, PA

In this paper we describe new and improved results for asynchronous settings with weak, or altogether lacking, atomicity assumptions. Our first contribution is clock construction for shared memory asynchronous systems. The clock is a basic tool to keep track of the overall amount of work performed by the system. Thus, if we have an estimate of the amount of work between synchronization barriers, then the clock can function as a synchronization mechanism (the notion of "work" is formally defined later). This enables to perform the synchronization on a purely computational basis without reference to actual time (which is ill-defined in asynchronous systems).

The clock is a general construction for asynchronous systems and can function in the most extreme asynchronous setting, in which even individual reads and write (even of a single bit) are not assumed to be atomic. Individual read and write operations are viewed as occupying physical time intervals of varying lengths, and the intervals of different processors may overlap without coinciding. A formal description of this Fully Asynchronous Parallel system (FAPS) is given in Section 2. The clock is a very robust construction and works, even in this extreme form of asynchrony, with *overwhelmingly high probability* $(1 - 2^{-\theta(n)})$.

In the second part of the paper we show one possible application of the clock. Using the clock as a synchronization mechanism, we show how to efficiently simulate a synchronous PRAM on an asynchronous system. The asynchronous system we assume for the PRAM simulation is not the full asynchronous model, as we assume atomic reads and writes. No further atomicity (e.g. *read & write*) is assumed. For a synchronous PRAM with n processors we obtain the simulation on a n processor asynchronous system with an $O(\log^2 n)$ work overhead, and an $O(\log n)$ work overhead if the asynchronous systems contains $n/\log n$ actual processors. This is a $\log n$ factor improvement over the previous results ([KPRR92]). We also relax the assumption regarding the concurrency allowed by the system. Our scheme requires $O(\log n)$ concurrent memory accesses (instead of $O(n)$ in the previous works).

1.1 Previous and Related Work

Several studies, over the past years, have addressed the issue of incorporating asynchrony into the shared memory and PRAM models, and designing methods for handling the difficulties it introduces ([MSP90,CZ89,Gib89,Nis90,Her88] and others). Among these studies, there is a great diversity both in the formulation of the model and the complexity measures, and in the "target" algorithms to be implemented on the asynchronous system (e.g. PRAM simulation, FFT, graph connectivity). For a detailed overview the reader is referred to the introduction in [KPRR92] (notice there especially the important distinction between our notion of *progressive* computation and that of *wait-free* computation of [Her88,Her91]).

Martel, Subramonian and Park [MSP90] give an $O(Tn)$ work simulation scheme for a T step n processor PRAM program, on an $n/\log n \log^* n$ processor actual asynchronous system, which is work-optimal. With n processors in the actual system, the scheme gives an $O(\log n \log^* n)$ work overhead. This was later improved to eliminate the $\log^* n$ factor in [MS]. Reads and writes are assumed to be atomic. In addition there is a "loose atomicity" assumption which states that no more than $O(n)$ work units are completed in the system between specified read and write instructions. (this prevents tardy processors from clobbering correct results).

Kedem, Palem, Rabin and Raghunathan [KPRR92], give a scheme for the simulation of an EREW PRAM on an asynchronous system, assuming only atomic reads and writes. For an n processor PRAM program and an n processor system, the scheme gives an $O(\log^3 n)$ work overhead. If the actual system consists of $n/\log n$ actual processors then the overhead reduces to $O(\log^2 n)$.

Both of the above schemes assume that the simulating asynchronous system allows up to $O(n)$ concurrent reads, even for an EREW program. Both schemes are randomized and work w.h.p. in the presence of an oblivious adversary.

For the PRAM simulation scheme in this paper we assume a system with atomic reads and writes (no further atomicity assumptions) and $O(\log n)$ concurrency. We obtain the simulation with $O(\log^2 n)$ work overhead for the n processor system, and $O(\log n)$ overhead for the $n/\log n$ processor system. We note, however, that our scheme is Monte-Carlo, while the previous ones are Las-Vegas.

The idea of keeping several copies of each program variable used in this work, was introduced in the simulation context in [KPRR92]. The latter paper also has a clock. Our clock is, however, rather different, both in structure and in usage, from that of [KPRR92]. We use the clock to *drive* the computation, rather than for performing lateness tests. The possibility of doing so is unique to our new clock.

A very particular form of asynchrony is the fail-stop behavior. PRAM simulation on fail-stop PRAM is dealt in [KS91,KPRS91,KPS90] and others. Clearly

our results hold for this restricted model as well.

L. Lamport in [Lam86] dealt extensively with the delicate issue of atomicity of reads and writes, or the lack thereof. In the first part of [Lam86], a general definition of asynchronous systems is described. Our FAPS model is similar to the "global time model" described there.

1.2 Outline and Terminology

This paper is organized as follows. In Section 2 we give a formal description of the Fully Asynchronous Parallel System, which is the general model for the clock. In Section 3 we give the description of the clock and prove its strong properties. In Section 4 we give the application of the clock in the PRAM simulation scheme.

We, loosely, say that an event E occurs *with high probability (w.h.p.)* if for all $\alpha > 0$ there exists a proper choice of the relevant parameters such that $\Pr(E) \geq (1 - n^{-\alpha})$. We say that the event occurs *with overwhelming probability* if for α as above $\Pr(E) \geq (1 - 2^{-\alpha n})$.

2 The Fully Asynchronous Parallel System

In the definition of the asynchronous system we want to formulate the idea that each processor can have a completely non-correlated idea of "time"; non-correlated to that of other processors, and non- correlated to actual (physical) time. It is important to emphasize that not only can different processors disagree on the question "what time is it?" but also on how "fast" time passes by. In order to formulate this we express each processor's internal, subjective, view of time with regards to the actual (physical) continuous time axis. Note, however, that actual time does not exist for the processors; it is only for our convenience in formulation and analysis that it is introduced.

The asynchronous parallel computer model we assume is the following:

- The system consists of n independent parallel processors, $\{P_i\}_{i=1}^{n}$, and shared memory. Processors may also have private memory.
- Processors act by reading from and writing to shared memory, and by performing internal computations. We postulate a set of *basic actions* which include: reading or writing a single memory cell, or performing one of a predefined set of internal computations.
- Processors have an *internal* view of time. Internal time is discrete, ranging over the natural numbers, $\mathcal{N}$. At each internal time point, $\tau \in \mathcal{N}$, a processor performs exactly one of the above basic actions.
- To each processor P_i there corresponds a *schedule* T_i mapping discrete internal time into actual continuous-time intervals. Formally: let $\text{Int} = \{[a,b) | 0 \leq a < b \leq \infty\}$, the function $T_i : \mathcal{N} \rightarrow \text{Int} \cup \{\text{Failure}\}$ is a mapping with the following properties:
  1. Non overlapping: For $\tau, \sigma \in \mathcal{N}, \tau \neq \sigma$, if $T_i(\tau) \in \text{Int}$ and $T_i(\sigma) \in \text{Int}$ then $T_i(\tau) \cap T_i(\sigma) = \phi$.
  2. Order preserving: for $\tau < \sigma$ if $T_i(\tau) = [a,b)$ and $T_i(\sigma) = [c,d)$ then $b \leq c$. And if $T_i(\tau) = \text{Failure}$ then $T_i(\sigma) = \text{Failure}$[1].

  The mapping T_i is called the *schedule* of processor P_i, and the sequence $T = (T_1, T_2, \ldots, T_n)$ is the *total schedule.*
- An interval, $T_i(\tau)$, in the range of T_i is called an *action interval.* The action interval $T_i(\tau)$ is the actual time interval required by processor P_i to perform the basic action taking place at its internal time point τ. Throughout this interval, and nowhere else, the internal time for P_i is τ.

  This formulation implies:
  - There can be arbitrary long actual time gaps between actions of any given processor. This allows the n processors to behave in complex forms of interleaving and overlapping actions.
  - The actual time it takes to perform any action, including the basic actions of reading and writing shared memory cells, may vary from one processor to another, as well as for the same processor from action to action.
  - The model does not assume atomicity of any sort, not even of the basic actions of reading or writing single memory cells. Even a single read or write action of a processor to a single cell is spread over a time interval (rather then occupying an idealized discrete time point), and during this interval the state of the memory cell is not determined. Moreover, two processors may access the same cell during action intervals that overlap, but do not coincide.
- Concurrent memory accesses produce non-deterministic results. If two processors perform a read or write action involving the same

[1] The first property can actually be deduced from the second.

memory cell and their action intervals overlap then the outcome is non deterministic, and can produce any value (or no value at all). Later we refer to such overlapping accesses as *interfering* with each other (Definition 3.1).

An internal computation by a processor, however, never interferes with any other action, and cannot be interfered with, even if the physical time intervals overlap.

- The total schedule is determined by an adversary. We consider two types of adversary:

  1. Dynamic adversary: At any actual time instance the dynamic adversary may view the entire state of the computation and determine the continuation of the schedule. The adversary cannot, however, prescribe what actions processors choose to perform in the action intervals granted to them.
  2. Oblivious adversary: The oblivious adversary determines the entire schedule before the parallel computation starts. The adversary has full knowledge of the computation to be executed, but cannot make schedule changes during the course of the actual computation.

- The only time-based relations available to the system are: before, after and concurrent. In particular this means that the system is insensitive to strictly monotonic transformations of the time axis onto itself.

We call a system thus described a *Fully Asynchronous Parallel System* (FAPS).

Definition 2.1 *A FAPS, M, is a triplet $M = (n, A, T)$, where n is the number of processors, A the set of basic actions and T the total schedule.*

Complexity and efficiency in the FAPS clearly cannot be assessed by the standard measures of time or number of steps. For a FAPS it is natural to measure work in number of action intervals. Hence the following definition:

Definition 2.2 *Let M be a FAPS and $I = [t_0, t_1]$ a physical time interval. We say that I contains k work units if, summed up over all processors, there are k complete action intervals in I; i.e. $k = \sum_{i=1}^{n} |\{\tau | T_i(\tau) \subseteq I\}|$ (where $|S|$ is the cardinality of the set S).*

3 The Clock

Our first goal is to construct a robust clock which functions in this highly asynchronous environment. Clearly such a clock cannot measure actual time in the physical sense, rather it will give a good measure of the amount of work performed. For a system with n asynchronous physical processors the clock advances from π to $\pi + 1$ after $\Theta(n \log n)$ work units. In subsequent sections this clock will be harnessed to drive the entire parallel asynchronous computation. The clock can function in the *dynamic adversary* setting, with *arbitrary outcomes* for interfered actions (for reads as well as for writes).

The clock is composed of three arrays, of $k = cn \log n$ locations each (c to be determined later), $\text{Clock}^l = (C_1^l, C_2^l, \ldots, C_k^l)$, $l = 0, 1, 2$, which drive each other in a circular fashion. Before going into the technical details let us first outline the general behavior of the clock. Later on we give exact meaning to the somewhat "fuzzy" notions first used.

Locations of Clock^l hold values π such that $\pi \equiv l \bmod 3$. Initially, the value 0 is written in all locations of Clock^0, 1 in all locations of Clock^1 and 2 in those of Clock^2. Now the value 2 in Clock^2 will start driving the value of Clock^0 to 3, which in turn drives the value of Clock^1 to 4, and so forth in a circular fashion (for simplicity, in the following all operations in clock superscripts are taken $\bmod 3$, i.e. $2 + 1 = 0$ etc.). We insure that Clock^{l+1} does not start driving Clock^{l+2} from $\pi - 1$ to $\pi + 2$ until Clock^{l+1} itself has the value $\pi + 1$ "firmly" written in it. And by the time Clock^{l+2} starts driving Clock^l from π to $\pi + 3$, the value $\pi + 1$ is written in Clock^{l+1} in an extremely robust form, durable in face of any number "clobbers" by tardy processors. The actual clock value is obtained by taking the value of Clock^0 and dividing it by 3. Since the clock is of size $\Theta(n \log n)$, obtaining the value of the clock is actually achieved by sampling the clock arrays.

We now give an exact formulation of the above outline. Let X be a set of memory locations (e.g. $X = \text{Clock}^l$). A *d-sample*, S, of X is a reading of d randomly chosen locations of X. For a sample S and a value π, denote by $\text{Count}(S, \pi)$ the *fraction* of the locations in the sample S which gave the reading π. The protocol for a processor participating in updating the clock is:

Protocol 1: Clock Update

1. Choose $l \in \{0, 1, 2\}$ at random.
2. d-sample Clock^l, let S be the sample. If for all values π, $\text{Count}(S, \pi) < .7$ then exit.

3. Let π be such that $\text{Count}(S,\pi) \geq .7$ (π is unique). Choose one location of Clock^{l+1} at random and write $\pi+1$ in it.

We prove that, with overwhelming probability, once the arrays are initialized as above, the values appearing in the vast majority of the cells of the three clock arrays advance monotonically in a circular fashion, in FAPS phases consisting of $\Theta(n \log n)$ work units.

Before we analize the overall dynamic behavior of the clock we must address the impact of concurrent memory accesses, i.e. overlapping read or write actions interfering with each other. We prove that these have a negligible effect on the overall behavior.

Denote by $A_i(\tau)$ the basic action performed by P_i in internal time point τ (i.e. occurring during actual time interval $T_i(\tau)$).

Definition 3.1 *Say that actions $A_i(\tau)$ and $A_j(\sigma)$ ($i \neq j$)* interfere *with each other, if they both access (read or write) the same shared memory cell and the corresponding action intervals overlap ($T_i(\tau) \cap T_j(\sigma) \neq \phi$). When $A_i(\tau)$ and $A_j(\sigma)$ interfere with each other we also say that $A_j(\sigma)$ interferes with $A_i(\tau)$, and vise versa.*

Lemma 3.1 *Let M be a FAPS, and $I = [t_0, t_1]$ a physical time interval containing $b \cdot n \log n$ action intervals. Assume M is running Protocol 1. Then, with overwhelming probability, no more than $O(n)$ actions are interfered with in I.*

Proof: Consider the following ordering of actions: for $A_i(\tau)$ and $A_j(\sigma)$ with $T_i(\tau) = [a,b)$ and $T_j(\sigma) = [c,d)$, $A_i(\tau) \prec A_j(\sigma)$ iff $a < c$, or $a = c$ and $i < j$. This is a complete ordering of the actions. Say that $A_j(\sigma)$ *injures* $A_i(\tau)$ if:

1. $A_j(\sigma)$ interferes with $A_i(\tau)$.
2. $A_i(\tau) \prec A_j(\sigma)$.
3. For all $A_k(\lambda)$, $A_i(\tau) \prec A_k(\lambda) \prec A_j(\sigma)$, $A_k(\lambda)$ does not interfere with $A_i(\tau)$.

The idea behind the injuring relation is that while an action can interfere with several actions, it can injure at most one. Also, every action which is interfered by a later action, is also injured by some action.

We now count the number of injuries in I. Let A^i be the *i-th* action in I, according to the above ordering. Let X_i be a Bernoulli random variable getting the value 1 if A^i injures another action in I, and 0 otherwise. Let t_i be the beginning time of action A^i. Let L_i be the set of cells in the clock arrays which are accessed at t_i by some action previous to A^i (according to the defined ordering). At any given physical time instance there are at most n read or write actions in progress, thus $|L_i| \leq n$. Each clock array contains $cn \log n$ cells and the processors randomly choose which cell to access. Thus $\Pr(X_i = 1) \leq 1/c \log n$. The X_i's, however, are not independent. We define another set of Bernoulli random variables, Y_i's, as follows. For all L_i choose a set $\bar{L}_i$ of cells of the clock, such that $L_i \subseteq \bar{L}_i$ and $|\bar{L}_i| = n$. Let Y_i be a random variable such that $Y_i = 1$ iff action A^i chooses to access a cell $\bar{L}_i$. Now, the Y_i's are independent and $\Pr(Y_i = 1) = 1/c \log n$. Thus in a total of $bn \log n$ actions occurring during the interval I, with overwhelming probability $\sum Y_i \geq 2bn/c$. Clearly $\sum X_i \leq \sum Y_i$. Thus, with overwhelming probability, there are no more then $2bn/c$ injuries in I. It remains only to notice that number of actions interfered with in I is at most double the number of injuries. Thus the total number of interfered actions is at most $2 \cdot 2bn/c = O(n)$. ■

Denote by $\text{Count}(\text{Clock}^l, \pi)$ the fraction of location of Clock^l which hold the value π. Say that a sample, S, of Clock^l is *ϵ-distorted* if there exists a π such that throughout the physical time interval in which the sample was taken $\text{Count}(\text{Clock}^l, \pi) \geq .3$ and $|\text{Count}(S,\pi) - \text{Count}(\text{Clock}^l, \pi)| \geq \epsilon$. A sample is *interfered* if at least one of the reading actions involved in it is interfered. The following fact is proved by standard statistical arguments:

Fact 3.2 *For any $p < 1, \epsilon < 1$, there exists a d such that the probability that a non-interfered d-sample is ϵ-distorted is $\leq p$.*

Definition 3.2 *Let I be a real time interval. An execution of Protocol 1 is said to* transcend *I if the execution partially, but not fully, overlaps I.*

Fact 3.3 *For any physical time interval I there are at most $2n$ protocol executions transcending it.*

An execution of a protocol is *interfered* if the sample or the write action are interfered.

Using the above definitions and lemmas we can now prove the following *main lemma.*

Lemma 3.4 *There exist constants $c > 1, 0 < \epsilon < 1, 1 < d_1 < d_2$, such that if at some time instance, t_0, the state of the clock is the following:*

- $\text{Count}(\text{Clock}^j, \pi) > .5$,
- $\text{Count}(\text{Clock}^{(j+1)}, \pi+1) > .7 + \epsilon$
- $\text{Count}(\text{Clock}^{(j+2)}, \pi+2) > .7 + \epsilon$,

then, with overwhelming probability, after a time interval I containing w work units, with $d_1 n \log n < w < d_2 n \log n$, the following state will be reached:

- $\text{Count}(\text{Clock}^j, \pi+3) > .7+\epsilon$,
- $\text{Count}(\text{Clock}^{(j+1)}, \pi+1) > .5$
- $\text{Count}(\text{Clock}^{(j+2)}, \pi+2) > .7+\epsilon$.

Proof: W.l.o.g. assume $j=0$. Let us first concentrate on non-interfered protocol executions originating after t_0. The progress of the clock will take place in two stages. Initially, protocol executions for which the sample is at most ϵ-distorted have the following outcomes, depending on the choice of l is stage 1 of the protocol:

- $l=0$: Write $\pi+1$ in Clock^1 or exit.
- $l=1$: Write $\pi+2$ in a random location of Clock^2 (either overwriting a previous value or rewriting $\pi+2$).
- $l=2$: Write $\pi+3$ in a random location of Clock^0.

By fact 3.2 at most a p fraction of the total amount of executions are with distorted samples, and will produce other values. Thus, for p small enough, with overwhelming probability, $\text{Count}(\text{Clock}^2, \pi+1)$ and $\text{Count}(\text{Clock}^3, \pi+2)$ will not decrease, while $\text{Count}(\text{Clock}^0, \pi+3)$ will be constantly growing. This state of affairs continues at least until $\text{Count}(\text{Clock}^0, \pi+3) \approx .7-\epsilon$. Which means a change of at least 20% of the locations of Clock^0. Thus, this stage should take roughly no less than $3(.2dcn\log n)$ work units and no more than $d'n\log n$, for some d'.

Once $\text{Count}(\text{Clock}^0, \pi+3) \geq .7-\epsilon$ then Clock^0 can start driving Clock^1 to $\pi+4$. At this point, for non-interfered and at most ϵ-distorted samples, the different choices of l have the following outcomes:

- $l=0$: Write $\pi+4$ in a random location of Clock^1, or exit.
- $l=1$: Write $\pi+2$ in a random location of Clock^2, or exit.
- $l=2$: Write $\pi+3$ in a random location of Clock^0, thus driving up $\text{Count}(\text{Clock}^0, \pi+3)$.

As above, only a p fraction of the samples are ϵ-distorted. Thus, with overwhelming probability, $\text{Count}(\text{Clock}^3, \pi+2)$ will not decrease, and $\text{Count}(\text{Clock}^0, \pi+3)$ will continue growing. This state of affairs will continue at least as long as $\text{Count}(\text{Clock}^1, \pi+2) > .5$. Initially we had $\text{Count}(\text{Clock}^1, \pi+2) > .7$, thus, for it to fall under .5 will require at least $0.2dcn\log n$ work units. Due to the randomization roughly the same amount of work is performed with the choice $l=2$. For a proper choice of ϵ this will be sufficient to drive $\text{Count}(\text{Clock}^0, \pi+4)$ to at least $.7+\epsilon$.

Interfered protocol executions, and protocol execution transcending from before t_0 may result in outcomes other than those presumed above. However, by Lemma 3.1 and Fact 3.3 there are only $O(n)$ such writes, and thus they have a negligible effect on a $cn\log n$ size clock. ∎

At any time instance t let the value of the clock be:

$$\text{Clock}(t) = \begin{cases} \pi/3 & \text{if } \text{Count}(\text{Clock}^0, \pi) \geq .6 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since the initial state of the clock obeys the conditions of the lemma we get by induction the following as a simple corollary:

Theorem 1 *Let M be a FAPS with a dynamic adversary. For all α, $m = 2^{\alpha n\log n}$ and $s>0$ there exist constants d, c, such that if M is operating by Protocol 1, then with probability $\geq 1-2^{-sn\log n}$ the following holds:*

- *The value of the clock propagates monotonically through all integer values $\{1,2,\ldots,m\}$.*
- *There exist constants d_1, d_2, d_3, d_4 such that for each integer value π the value of the clock is π for w work units, with $d_1 n\log n \leq w \leq d_2 n\log n$, and between the time the value is π and the time the value is $\pi+1$ there are w' work units, $d_3 n\log n \leq w' \leq d_4 n\log n$.*

Thus the clock gives us a good measure of the amount of work performed on it.

Reading the clock is performed by $d\log n$-sampling Clock^0. Let S be such a sample. The value of the clock is taken to be π if $\text{Count}(S, 3\pi) \geq .7$, and undefined if this does not hold for any π. Theorem 1 tells us that this form of reading gives us a "clock-like" behavior.

4 PRAM Simulation

In this section we show how to use the clock to obtain an efficient PRAM simulation scheme. In the simulation, however, we cannot cope with full asynchrony and must introduce some minimal atomicity assumptions. We assume an Asynchronous Parallel System (APS) with an oblivious adversary, for which only single reads and writes are guaranteed to be atomic. In addition we assume the system allows up to $O(\log n)$ concurrent reads and writes to the same memory location (with some arbitration policy for conflicting writes). A formal description of a similar model with $O(n)$ concurrency may be found in [MSP90,KPRR92].

4.1 PRAM Computation and Simulation

Our overall objective is to enable execution of computations programmed for a synchronous PRAM

(without failures) on an asynchronous system. This is achieved by means of *program transformation*. We give a uniform method for transforming any given PRAM program into a APS program, which yields the same results (under an appropriate interpretation). The PRAM model we consider is the EREW (Exclusive Read Exclusive Write) PRAM. For the sake of completeness let us state the characteristics of a EREW PRAM program:

- The program is written in parallel *steps*. In every step each PRAM processor is to perform one instruction of the form $x \leftarrow f(y,z)$. It is postulated that each of the variables x, y, z, occupies a single shared memory cell.
- All instructions in a step are assumed to be performed concurrently and completed together. In particular no processor has to await the output, in the same step, of any other processor.
- It is assumed that all reads in the parallel step occur before all writes. Thus if a processor reads a variable it will obtain the value last written in it *before the current step*.
- The program is written is such a way so as to guarantee that during no one parallel step more than one processor attempts to access (read or write) the same memory cell.

Each PRAM step is translated into a *phase* in the operation of the APS, consisting of $\Theta(n \log^2 n)$ atomic actions. The transformation guarantees that, w.h.p., the computation is:

1. Correct: produces the same results (under suitable interpretation) as the original PRAM program.
2. Progressive: If $\Theta(n \log^2 n)$ work units are devoted to a phase then the corresponding PRAM step is completed.

A single step would take the synchronous PRAM $\Theta(n)$ work units. Thus, the complexity overhead is $\Theta(\log^2 n)$.

4.2 Program Transformation

In each PRAM parallel step each PRAM processor, V_i, is to perform an instruction of the form $x \leftarrow f(y,z)$. Focusing on one such step, we designate the instruction that a specific processor V_i is to perform by $x_i \leftarrow f_i(y_i, z_i)$. Following [KPRS91] we split each parallel step into two sub-steps. First V_i reads the values of $y_i, z_i,$, computes $f_i(y_i, z_i)$ and writes the value in the *i-th* location of a special temporary array: $\mathrm{tmp}_i \leftarrow f_i(x_i, y_i)$. Then the new value is copied back from the temporary array into its location in memory, $x_i \leftarrow \mathrm{tmp}_i$. This two sub-step operation mode, known as TIES, is also enforced when dealing with control variables, such as the individual processor's program counters etc.. Having split each instruction this way, each sub-step becomes *idempotent*, that is: performing it several times has the same effect as performing it once. For the exact formulation and a full description of TIES the reader is referred to [KPRS91].

To avoid confusion we refer to the work to be performed by the PRAM processors as computation *threads*. Thus, there are n computation threads $\mathrm{Thread}_1, \ldots, \mathrm{Thread}_n$, corresponding to the n PRAM processors to be simulated.

In an asynchronous machine processors may "go to sleep" for long periods of time and then "wake up" in a later stage without knowing it. If the processor was about to write some variable before falling asleep then when waking up it might overwrite a new value by an obsolete one. In order to avoid losing the correct values, following [KPRR92], for each memory variable we keep μ actual copies, $\mu = \Theta(\log n)$. We shall see to it that w.h.p. at all times at least 3/4 of the copies of each variable hold the correct value. The temporary array variables also have μ copies. For a variable (or temporary variable) v, we denote the by $v^{(j)}$ the *j-th* copy of v.

Processors divide their effort between working on the actual program and advancing the clock, by randomly chosing between the two. Since we have definite bounds on the amount of work it takes to advance the clock, this will also give an accurate measure of the amount of work devoted to the program. We see to it that this amount is sufficient to guarantee that w.h.p. by the time the clock advances from one value to the next, the current program sub-step has been completed.

When a processor chooses to work on the program it could either be in a computing sub-step or in a copying back sub-step. There are separate protocols for each of these. Recall that reading the clock is performed by $d \log n$-sampling Clock^0, and dividing by 3 (Section 3). The overall protocol is thus ($0 < q < 1$ to be determined later):

Protocol 2: General Step

1. Choose $r \in \{0,1\}$ at random, with $\Pr(r=0) = q$, $\Pr(r=1) = 1-q$.
2. If $r = 0$ then perform clock update protocol (Protocol 1).
3. If $r = 1$ then read clock. If clock value is undefined then abort, else let π be the value.

4. If π is odd then perform computing sub-step protocol, else perform copying back sub-step protocol.

Definition 4.1 *The π-th* phase *is the actual time interval in which the clock value is π.*

Later we will see that the individual computing and copying back protocols consist of $O(\log n)$ basic actions each. Call the work performed on the clock *clock work* and that on the program (i.e. computing and copying protocols) *program work*. Recall that processors randomly choose if to perform program work or clock work using a $(q, 1-q)$ biased coin.

Fact 4.1 *For all b there exists a q such that with overwhelming probability in each phase the number of work units devoted to program work is w, with $w \geq bn\log^2 n$, $w = O(n\log^2 n)$.*

We now turn to describing the protocols for computing and copying back, starting with the former.

At each step there are n computation threads to be simulated. Corresponding to each thread Thread_i there is a value $f_i(y_i, z_i)$ to be computed and stored in the temporary array. Each such value must be written in $\mu = \beta\log n$ copies in the temporary array. Thus there are all in all $\beta n\log n$ tasks to be performed. Each time a processor is in a computing sub-step protocol it chooses one of these tasks at random and performs it. Obtaining the value of a variable is achieved by reading all the copies of the variable and taking the value that appears in most of them.

Protocol 3: Computing Sub-Step

Let π be the clock reading obtained in the general step protocol.

1. Choose $i \in \{1, \cdots, n\}$ and $j \in \{1, \ldots, \mu\}$ at random.
2. Read *i-th* thread program counter. Let $f_i(y_i, z_i)$ be the value to be computed by Thread_i at this step.
3. Read y_i and z_i.
4. Read clock. If value other than π then exit.
5. Write value of program counter $\text{PC}(i)$ in $\text{tmpPC}(i)^{(j)}$.
6. Compute $f_i(y_i, z_i)$ and write the value $\text{tmp}_i^{(j)}$.

The protocol for the copying back substep is analogous. Both protocols consists of $\Theta(\log n)$ action each.

We now give a brief analysis and prove the correctness of the scheme.

Definition 4.2 *An execution of a protocol is said to be* in sync *if its entire execution is within one phase. The execution is* out of sync *if it spans more than one phase.*

Lemma 4.2 *For all b' there exists a b such that if at least $bn\log^2 n$ work units are devoted to program work during each phase then w.h.p. each phase contains at least $b'n\log n$ complete in-sync protocol executions of program work.*

Proof: Assume each protocol takes at most $s\log n$ work units. By Fact 3.3 there are at most $2n$ out of sync protocol executions transcending the phase. At most $2ns$ work units are devoted to these out of sync executions. Thus there are at least $(bn\log^2 n - 2sn\log n)/s\log n$ in sync protocol executions in the phase. ∎

Thus, combining this with Fact 4.1, we can see to it that each phase contain "sufficient" amount of complete protocol executions.

Say that protocol execution *successfully terminated* if it is completed (does not exit in step 4). Note that only successfully terminating protocol execution perform write actions. For a successfully terminating protocol execution, say it is a *π-th phase* execution if it works with a clock reading of π. Recall that reading the clock is performed by a $d\log n$-sample.

Fact 4.3 *With high probability, all clock reads are correct.*

Corollary 4.4 *Let E be a π-th phase successfully terminating protocol execution. W.h.p. all the readings of all variables in E where performed within the π-th phase.*

Proof: The clock is read before and after reading the variables. By Fact 4.3 w.h.p. the clock readings are correct. The value of the clock must be identical in these two readings for the execution to successfully terminate. ∎

Next we examine the write actions and the values they produce in the copies of the variables.

Definition 4.3 *Let I be a time interval. A variable v is said to be* λ-correct *in I if there exist at least a λ fraction of the copies of v that contain the correct (current) value throughout I. The entire memory is λ-correct in I if all variables which are not to be updated in I (according to the PRAM program) are λ-correct in I.*

We prove that at all times the entire memory is at least 3/4-correct, which has the following implication:

Fact 4.5 *Assume that the memory is 3/4-correct during the π-th phase. Then all copies written by a π-th phase protocol executions are in fact written with correct values according to the π-th PRAM step.*

Thus a copy updated by an in sync protocol executions holds the correct value (if that value was not overwritten later). There are two possible reasons why a copy does not hold the correct value.

- Old copy: the copy was not updated during the most recent update phase.
- Clobbered Copy: the copy was correctly updated, but was later overwritten by a protocol execution originating in a previous phase.

The following simple lemma bounds the probability of a copy to be old.

Lemma 4.6 *Assume that the memory is 3/4-correct during the π-th phase. Let v be a variable to be updated during this phase. For all $p < 1$ and β there exists a b' such that if there are at least $b'n \log n$ in sync protocol executions during the π-th phase then the probability that by the end of the π-th phase a specific copy of v, $v^{(j)}$, is not updated is $\leq p$.*

The next lemma bounds the probability of a copy to be clobbered.

Lemma 4.7 *The probability that a copy is clobbered is $\leq 1/\beta \log n$.*

Proof: Let $v^{(j)}$ be a copy of a variable and let π be the most recent phase that v was to be updated. The copy $v^{(j)}$ can only be clobbered by protocol executions originating before the π-*th* phase and terminating during or after it. There are at most n such protocol executions. Let us focus on one such execution. The PRAM step this execution is simulating is $\sigma < \pi$. If variable v was not to be written at the σ-*th* step then this protocol cannot cause a clobber in v. If variable v was to be written in the σ-*th* step then the probability that this specific execution chooses to update $v^{(j)}$ is $\leq 1/\beta n \log n$ (the scheduler is oblivious). Thus the total probability is $\leq n/\beta n \log n$. ∎

Each variable has $\mu = \beta \log n$ copies. Thus we obtain:

Lemma 4.8 *Let π be the most recent phase that a variable v was to be updated. For all ϵ there exist β and b', such that if v has $\beta \log n$ copies and at least $b'n \log n$ in sync protocol execution were completed in the π-th phase, w.h.p., following the π-th phase at most an ϵ fraction of the copies of v are old or clobbered.*

Thus we have:

Lemma 4.9 *Assume that the memory is 3/4-correct during the π-th phase, then w.h.p. the memory is 3/4-correct during the phase $\pi + 1$.*

Since initially the memory is completely correct we obtain the correctness of the entire simulation by simple induction. Finally we address the issue of concurrency.

Lemma 4.10 *W.h.p. no cell is accessed concurrently by more than $O(\log n)$ processors.*

Proof: The variables accessed by different threads are distinct. There are n processors and n threads. The processors randomly choose which thread to simulate. ∎

Putting this all together we obtain:

Theorem 2 *Let M be an n processor asynchronous system with atomic reads and writes, allowing up to $O(\log n)$ concurrency in memory access. Let $\mathcal{P}$ be an $m = poly(n)$ step EREW PRAM program. The above protocols are a transformation of $\mathcal{P}$ into a program for M, such that with overwhelming probability each step takes $\Theta(n \log^2 n)$ work units, and with high probability for each PRAM variable at all times at least 3/4 of the μ copies representing it hold the correct (current) value.*

For a system with $n/\log n$ actual processors we can reduce the complexity overhead to $O(\log n)$. However, the protocols and the analysis are somewhat more complex in this case, and we cannot give the details here. This is planned for the final version.

Finally, if we want to have some indication of the termination of the computation, then, w.l.o.g. the PRAM program can be augmented so as to include a control variable which will hold the value "Done", iff all the program counters, for the n PRAM processors reached "halt". Thus, by examining the copies of this variable, one can can determine that the simulation is completed. The output/results of the simulated PRAM computation can be then acquired by reading most copies of the relevant program variables.

5 Final Remarks

- The Clock described here is composed of three sub-arrays. It is also possible to construct such a clock with only two sub-array. (clearly, properly modifying the update protocols). We found the present construction to be the simplest to describe, and preferred presentation clarity over technical efficiency.

- The clock described here has size $\Theta(n \log n)$. We proved it works with probability $\geq 1-2^{-\alpha n}$. Actually we can prove a probability $\geq 1-n^{-\alpha n}$. A similar construction with an $\Theta(n)$ size clock (each array with cn locations), gives a probability $\geq 1-2^{-\alpha n}$.
- In the present work we made no assumptions regarding the actual or relative speeds of the processors. The results hold even with a schedule in which only one processor is doing all of the work, or any other schedule chosen by the adversary. This is due to the full randomization implemented by the processors. If we substitute this adversary scheduling by a stochastic timing model, then we can do away with some of the randomization. In particular, if the clock is composed of elements with fixed connection wires, but for which the timing is determined by a proper stochastic process, then a behavior similar to the one of the present clock should be expected. Thus, this clock may be significant and applicable in other domains as well, both within the world of asynchronous and distributed computing and elsewhere. This is the topic of a current study.

References

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

[Gib89] P. B. Gibbons. A more practical PRAM model. In *Proc. of the 1st ACM Symposium on Parallel Architectures and Algorithms*, pages 158–168, 1989.

[Her88] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on the Principles of Distributed Computing*, pages 276–290, 1988.

[Her91] M. Herlihy. Impossibility results for asynchronous PRAM. In *Proc. of the 3rd ACM Symposium on Parallel Architectures and Algorithms*, pages 327–336, 1991.

[KPRR92] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformation for resilient parallel computation via randomization. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, 1992.

[KPRS91] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P.G. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, 1991.

[KPS90] Z.M. Kedem, K.V. Palem, and P.G. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22rd Annual ACM Symposium on Theory of Computing*, pages 138–148, 1990.

[KS91] P Kanellakis and A. Shvartsman. Effecient parallel algorothms on restartable fail-stop processors. In *Proceedings of the 10th Annual ACM Symposium on the Principles of Distributed Computing*, pages 23–36, 1991.

[Lam86] L. Lamport. On interprocess communication. Part i: Basic formalism, and Part ii: Algorithms. *Distributed Computing*, 1(12):77–101, 1986.

[MSP90] C. Martel, R. Subramonian and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science*, pages 590–599, 1990.

[MS] C. Martel and R. Subramonian. On the complexity of certified write-all algorithms. Unpublished Manuscript.

[Nis90] M. Nishimura. Asynchronous shared memory parallel computations. In *Proc. of the 2nd ACM Symposium on Parallel Architectures and Algorithms*, pages 76–84, 1990.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

DISCUSSION

Rapporteur: Ann Petrie

Lecture One

Professor McCarthy said that Professor Rabin's pattern matching depended on the particular structure of multiplication and asked him if he had any idea what kinds of problem admitted of this structure. Professor Rabin agreed that he was making strong use of the distributivity and associativity of multiplication (not commutativity since matrix multiplication is not commutative). This makes his method look very particular. However he still believed that the method was useful and would answer the question by reminiscing about his own experience.

In 1975 he gave a lecture at CMU on randomized algorithms, not probabilistic ones that would work well in the average case, but ones where "you twist things around and bend them to your will and create an algorithm which, with exceedingly high probability is true in every case". The two examples he used were primality testing and finding the nearest pair in a very large set of points in n dimensions. The reaction of people there was that it was "all very nice" but rather specialized: he had used a variation of Fermat's theorem for the primality example and something else problem specific for the nearest pair example and consequently the method was not going to have wide utility.

Professor Rabin said that it turns out that there are many applications, as can be seen if you look at meetings in theoretical computer science and also in some actual applications. In theoretical computer science maybe a third of all papers at important conferences are devoted to randomized algorithms. Consequently, letting history be his guide, he considered that his approach merited study and that he would not like to say what its actual limitations were. He added that the parallelization of the pattern matching method works equally well for n dimensions as for two.

Professor Knuth said that he did not understand what was meant by the assertion that a program was correct "with very high probability" and asked for clarification. Professor Rabin said that he had not said that the program was correct. It was an important point that had been raised and he would explain what he did mean. He was not asserting that a program or procedure was correct - such a statement would have to be concerned with its semantics. What he was talking about was the following experiment.

Suppose we have a procedure for multiplying together two matrices - this is a black box. The experiment consists of choosing 10,000 random pairs of 1000x1000 {0,1} matrices A and B. (These are taken from a discrete domain so the problem of choosing randomly presents no problem.) For each pair we calculate A.B using the given procedure and then compare it with the result obtained using the ordinary (laborious) method of matrix multiplication which we assume to be correct. The question is, in the space of all 1000x1000 {0,1} matrices (which is enormous), what percentage of pairs will lead to results which disagree? If the the black box procedure is wrong (i.e. the results of the two methods for matrix multiplication disagree) in more than 1% of the cases, then the probability of this not being discovered in 10,000 trials is smaller than 1/exp(100). This is the assurance that we have without going into the semantics of the procedure.

DISCUSSION

Rapporteur: John S Fitzgerald

Lecture Two

Professor Dijkstra recalled that Professor Rabin's clock consisted of three lines of equal length and asked him to justify the number of lines and the equality of length.

Professor Rabin replied that the number three was not sacrosanct, but convenient in that it made the operation of the clock easier to explain than a two line case. All of that was, he said, "amenable to play". Filling the clock by threes allows an element of stability, whereas a two-line clock is more delicate in terms of the cross-actions between lines. A clock in a logical system could have a pulsator between two modes.

Professor Rabin commented that he liked the fact that we can reliably do things in this loose environment because of its potential for looking at neural systems. We would, for example, expect that if two neurons performed simultaneous writes, the result would be meaningless. This contrasts with the unrealism of synchrony, which forms the basis of current work in neural networks.

Professor Knuth commented that a difference between real neural networks and Professor Rabin's environment was that, in the latter, each processor can access any part of the shared memory with no restriction.

Professor Rabin thanked the questioner for an important remark, agreed and commented further that here he had adopted the randomised point of view. He would envision neural networks in nature in a statistical rather than randomised way. He would assume an array with fixed but random connections: this has statistical behaviour like random sampling. It is important to have some randomness of these connections and then the mathematical analysis becomes very similar.

Professor Wells suggested that this could be achieved by selecting cells randomly at the outset.

Professor Rabin pointed out that it would still be necessary randomly to select cells to be read in order to calculate the clock reading.

Professor Whitfield commented on Professor Rabin's "fuzzy variables". He pointed out that several processes were changing variable values and that the final value chosen was determined by the majority. He therefore proposed the terms "ballot variable" or "democratic variable". Professor Rabin favoured the former.