

- [DSB88] M. Dubois, C. Scheurich, and F. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, 9–21, 1988.
- [FLP85] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus," *J. of the ACM*, 32(2):374–382, 1985.
- [FW78] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Proc. 10th ACM Symp. on Theory of Computing*, 114–118, 1978.
- [Gi89] P. Gibbons, "A More Practical PRAM Model," *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures*, 158–168, 1989.
- [He88] M. Herlihy, "Impossibility and Universality results for Wait-free Synchronization," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 276–290, 1988.
- [He90] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures," *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 197–206, 1990.
- [He91a] M. Herlihy, "Wait-free Synchronization," *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, 1991.
- [He91b] M. Herlihy, "Impossibility Results for Asynchronous PRAM," *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, 327–336, 1991.
- [He91c] M. Herlihy, "Randomized Wait-Free Concurrent Objects," *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 11–21, 1991.
- [KS89] P. Kanellakis and A. Shvartsman, "Efficient Parallel Algorithms Can be Made Robust," *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 211–222, 1989.
- [KS91] P. Kanellakis and A. Shvartsman, "Efficient Parallel Algorithms on Restartable Fail-stop Processors," *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 23–36, 1991.
- [KPS90] Z. Kedem, K. Palem, and P. Spirakis, "Efficient Robust Parallel Computations," *Proc. 22nd ACM Symp. on Theory of Computing*, 138–148, 1990.
- [KPRS91] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis, "Combining Tentative and Definite Algorithms for Very Fast Dependable Parallel Computing," *Proc. 23rd ACM Symp. on Theory of Computing*, 381–390, 1991.
- [La78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. of the ACM*, 21(7):558–565, 1978.
- [LL84] J. Lundelius and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," *Information and Control*, 62:190–204, 1984.
- [LS85] L. Lamport and P. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. of the ACM*, 32(1):52–78, 1985.
- [MPS89] C. Martel, A. Park, and R. Subramonian, "Fast Asynchronous Algorithms for Shared Memory Parallel Computers," *Tech. Rep. CSE-89-8*, Univ. of California–Davis, 1–17, July 25, 1989.
- [MS91] C. Martel and R. Subramonian, "On the Complexity of Certified Write All Algorithms," *Manuscript*, 1991.
- [MSP90] C. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, 590–599, 1990.
- [Ni91] N. Nishimura, "Asynchrony in Shared Memory Parallel Computations," TR-253/91, University of Toronto, 1991.
- [Pl89] S. Plotkin, "Sticky Bits and Universality of Consensus," *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 159–176, 1989.
- [Ra82] M. Rabin, "The Choice Coordination Problem," *Acta Informatica*, 17:121–134, 1982.
- [Ra83] M. Rabin, "Randomized Byzantine Generals," *Proc. 24th IEEE Symp. on Foundations of Computer Science*, 403–409, 1983.
- [Ra89] M. Rabin, "Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance," *J. of the ACM*, 36(2):335–348, 1989.
- [Su91] R. Subramonian, *Asynchronous Algorithms for Shared-Memory Parallel Computers*, Ph.D. Dissertation, University of California, Davis, 1991.
- [Va90] L. Valiant, "A Bridging Model for Parallel Computation," *Comm. of the ACM*, 33(8):103–111, 1990.

ability is bounded above by $n^{-\sigma}$, for constant c and σ . We conclude that if $\mu' = c \log n / \log \log n$, the probability that μ' dubious writes went to *odd* for a given value of *even* is at most $n^{-\sigma}$.

We now set $\mu = 10\mu' = 10c \log n / \log \log n$. From the above arguments and conditioned on the fact that with probability at least $1 - n^{-\sigma}$ more than $4/5$ of the copies of *even* are unmarked with value τ at some point in time, it follows that *even* is read as a class 0 variable with probability at most $n^{-\sigma}$. A similar statement would hold for *odd* as well.

We now estimate the amount of work done in the ensuring a progressive clock. Theorem 2 from Section 2.2 helps us out here, by asserting that provided certain premises are met and at some critical time $t(\tau)$, *even* had the value of τ , then later at critical time $t(\tau + 2)$ it has the value of $\tau + 2$. Based on the discussion in the previous paragraphs, throughout the rest of this section, we will assume that no class 0 variables are encountered. It is easy to see that the remaining two premises of Theorem 2 are satisfied by the readers and writers working on the counter. Therefore, we only need to estimate the work involved in advancing the counter, i.e., the work done between $t(\tau)$ and $t(\tau + 2)$.

We now use a standard result from probability, the *coupon collector* result. It states that in order to pick all n coupons by randomly picking 1 out of n in each trial, we need no more than $X = c_1 n \log n$ trials with high probability, with c a suitable constant. Here and in the rest of this section, “high probability” stands for probability at least $1 - n^{-\sigma'}$. From this, it easily follows that

Theorem 3 Let $\mu = \Omega(\log n / \log \log n)$. For polynomial (parallel-)time PRAM programs, the probability that *Rollback* is ever set is inverse polynomial in n . Furthermore, if *Rollback* is never set, the probability that the amount of work between two consecutive critical times of *even* is $\Omega(n \log^3 n / \log \log^2 n)$ is inverse polynomial in n .

The work bound of the methodology of Section 2 is actually better than the previous theorem indicates. Martel and Subramonian [MS91] showed that a slight variation of the coupon collector algorithm for CWA does $O(\max\{n, p \log n\})$ expected work. This variation can be used in our case so that our methodology will increase the expected work complexity of the CWA algorithm from [MS91] by a factor of $O(\log^2 n)$, since each write translates into writing $O(\log n)$ copies and $O(\log n)$ copies of the global counter are read after writing each copy. Furthermore, the use of fuzzy variables increases the space complexity by a factor of $O(\log n)$.

When *Rollback* is set, we simply run n sequential computations. The probability of rollback is so small that this would not add to the asymptotic work and space complexity results.

Theorem 4 Our methodology increases the space com-

plexity of the ideal PRAM program by a factor of $O(\log n)$. It increases with high probability the expected work complexity of the ideal PRAM program by a factor of $O(\log^2 n)$, assuming up to $n / \log n$ asynchronous processors.

Acknowledgement

We wish to thank Yonathan Aumann for very helpful discussions and for uncovering a difficulty in an earlier version of this paper.

Bibliography

- [Be83] M. Ben-Or, “Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols,” *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, 27–30, 1983.
- [AB91] Y. Aumann and M. Ben-Or, “Asymptotically Optimal PRAM Emulation on Faulty Hypercube,” *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pp. 440–446, 1991.
- [AB92] Y. Aumann and M. Ben-Or, “Computing with Faulty Arrays,” *Proc. 24th ACM Symposium on Theory of Computing*, 1992 (to appear).
- [BBKTW90] S. Ben-David, A. Borodin, R. Karp, E. Tardos, and A. Wigderson, “On the Power of Randomization in Online Algorithms,” *Proc. 22nd ACM Symposium on Theory of Computing*, 1990.
- [BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- [C89] F. Christian, “Probabilistic Clock Synchronization,” *Distributed Computing*, 3:146–158, 1989.
- [CZ89] R. Cole and O. Zajicek, “The APRAM: Incorporating Asynchrony into the PRAM Model,” *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures*, 170–178, 1989.
- [CZ90] R. Cole and O. Zajicek, “The Expected Advantage of Asynchrony,” *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 85–94, 1990.
- [DHSS84] D. Dolev, J. Halpern, B. Simons, and R. Strong, “Fault-tolerant Clock Synchronization,” *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 89–102, 1984.

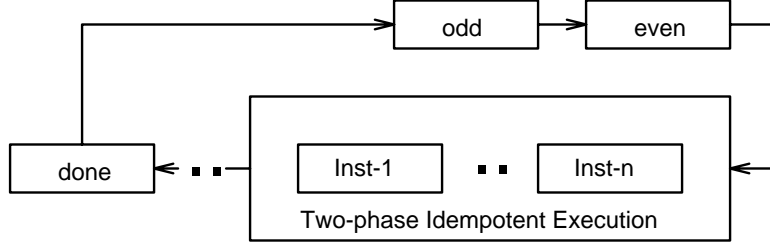


Figure 3: The management of TIES in detail

be incremented up to τ using an algorithm that depends on the specific certification scheme chosen.

- Node *odd*. The actor behaves as the writer who chose *odd* in Section 2.2 but it refers to *done* instead of *even*.
- Node *even*. The actor behaves as the writer who chose *even* in Section 2.2.

3 The complexity

In this section, we describe the analysis of our randomized technique. First we consider the power of the adversary that sets the processor speeds. Clearly, if the speeds can depend upon the random choices made, the power of randomization is completely lost, and our algorithm can be no better than the best deterministic algorithm. We adopt the notion of an *oblivious adversary* [BBKTW90]. For our purposes, an oblivious adversary is one that determines the processor speeds without adapting to the random choices by the algorithm. It is however, permitted to use knowledge regarding the *history* of the processors' speeds.

We now review those features of our methodology that are the most relevant for the analysis. For the sake of brevity, we will only analyze the case where the global counter operates in isolation. We will act like there are n variables, $var_1, var_2, \dots, var_n$, where var_1 is *odd*, var_n is *even*, and $var_i, 2 \leq i \leq n-1$ are dummy fuzzy variables. For concreteness, when we land on a dummy variable $var_i, 2 \leq i \leq n/2$, we use the same protocol to update it as if it were *odd*. When we land on a dummy variable $var_i, n/2 + 1 \leq i \leq n-1$, we use the same protocol to update it as if it were *even*.

1. Each time a processor chooses one of var_1, \dots, var_n , it does so with probability $1/n$; furthermore, this choice is completely independent of all previous choices.
2. All of $var_1, var_2, \dots, var_n$ have μ copies, with μ to be specified later.
3. Let processor $P[pid]$ participate in setting $var_i, 1 \leq i \leq n/2$, to value τ . While so participating, let

$P[pid]$ write into some copy of var_i ; such a write is called a τ -write. Our technique ensures that processor $P[pid]$ can make at most *one* copy of any of $var_1, \dots, var_{n/2}$ dubious by any τ -write when *even* has value τ' , with $\tau < \tau'$.¹⁶ Thus, in a given value τ' of *even*, at most p τ -writes can occur (one per processor), and at most p copies over all the variables $var_1, var_2, \dots, var_{n/2}$ can get dubious.

Let us first calculate the number of ways to distribute p different trials to n variables, such that at least one variable gets μ' trials or more (μ' to be specified later). This is to calculate the probability that out of the p dubious writes to $var_1, \dots, var_{n/2}$ that occurred for a given value of *even*, *odd* gets μ' or more trials.

Let P_1, P_2, \dots, P_n be properties where P_i is the property that var_i gets at least μ' trials. Then by the inclusion-exclusion principle, the number of ways to distribute trials to variables such that at least one of the properties holds is $\sum_{i=1}^n N(P_i) - \sum_{i \neq j} N(P_i P_j) + \dots + (-1)^{n+1} N(P_1 P_2 \dots P_n)$, where, as usual, $N(P_{i_1} P_{i_2} \dots P_{i_r})$ represents the number of ways to distribute trials to variables, while maintaining properties $P_{i_1}, P_{i_2}, \dots, P_{i_r}$.

Thus, the number of ways to distribute trials to variables such that at least one of the properties holds is bounded above by $\sum_{i=1}^n N(P_i)$. Trivially, $N(P_i) = \sum_{j=\mu'}^p \binom{p}{j} (n-1)^{p-j}$. Now, $\binom{p}{j+1} (n-1)^{p-j-1} = \binom{p}{j} (n-1)^{p-j} (p-j) / ((j+1)(n-1))$. Since $n-1 \geq p-j$, we have that $\binom{p}{j+1} (n-1)^{p-j-1} \leq (1/(j+1)) \binom{p}{j} (n-1)^{p-j}$. Extending this argument for $j = \mu', \mu'+1, \dots, p-1$, we have that $N(P_i) = \sum_{j=\mu'}^p \binom{p}{j} (n-1)^{p-j} \leq 2 \binom{p}{\mu'} (n-1)^{p-\mu'}$.

The number of ways to distribute trials to variables such that at least one of the properties holds is now bounded from above by $2n \binom{p}{\mu'} (n-1)^{p-\mu'}$. The probability that at least one of the properties holds is at most $2n \binom{p}{\mu'} \frac{(n-1)^{p-\mu'}}{n^p} \leq \left(\frac{p}{n-1}\right)^{\mu'} \frac{1}{\mu'!}$. It is easily seen that when $p = n$, and $\mu' = c \log n / \log \log n$, this prob-

¹⁶A similar statement will hold for $var_{n/2+1} \dots var_n$, with *odd* playing the role of *even*.

time t_2 *odd* had a class 1 or a class 2 value of $\tau - 1$. Then the value of the counter was τ during the time interval $[t_1, t_2]$.

Furthermore, we can also prove that:

Theorem 2 Assume that:

1. No variable ever has a class 0 value.
2. For any instance of time t and for any writer that does not fail permanently, there exist instances $t', t'' > t$ such that at time t' it picks *odd* and at time t'' it picks *even*.
3. At least one writer does not fail permanently.

Then, for any $\tau > 2$, there exists a finite time instance at which the value of *parity*(τ) became τ .

This theorem is of great importance as it shows that provided some conditions are satisfied the counter will not “starve.” In Section 3 we show that the appropriate conditions hold with very high probability and that the computation (including the counter) proceeds efficiently.

2.3 Implementing one phase

Having described all the necessary tools, we now expand the view of the system and describe it in Fig. 3. For definiteness we will consider the system as a directed graph. There are n action nodes labeled *Inst-1, Inst-2, ..., Inst-n*. They correspond to the n threads of one phase of an arbitrary instruction in our general simulation.¹² Finally, the termination is indicated by modifying the value of *done*.

What we need to assure is that an action of a processor does not start too early or terminate too late. To assure this, we rely on the following two facts:

Fact 1 If a processor while reading *even*,

1. encountered more than $4/5$ unmarked copies with the the value of τ , or
2. encountered more than $1/2$ unmarked copies with the the value of τ and this processor itself had previously wrote and unmarked τ in all the copies of *even*,

then the value of the counter at the instance the processor finished reading *even* was at least τ .

Fact 2 If a processor while reading *odd* encountered more than $1/2$ unmarked copies with the the value of $\tau - 1$, then the value of the counter at the instance the processor started reading *odd* was at most τ .

¹²In addition to these actions, various internal data structures monitoring the progress of the phase are acted on. The extensions to handle them are straightforward and ignored in the sequel.

The key idea is that if a possibly non-atomic action needs to be completed within one value of the global counter, the processor relies on Facts 1 and 2 to determine whether the action was done on time. For instance, consider the case of writing of a copy (see *Writer Protocol*). To make sure that the action performed by the processor in step 2 (of the writer protocol) completed while the value of counter has not been incremented beyond even τ , *odd* is checked to verify if its value has not exceeded $\tau - 1$. We will refer to this type of test as a *lateness test*. If writing to *var*^(copy) succeeded on time, the processor will write the same value to *var*^(copy+1), and so on, until either the lateness test fails, or *var*^(μ) is written. Here we considered only the “application program variables” and not “control” variables used for certification itself. These control variables and their values have to be treated differently, and in a manner similar to the way the counter’s variables are treated.

To execute all of *Inst-1, Inst-2, ..., Inst-n*, processors randomly choose nodes to execute. When all n executions complete, this fact will be certified by having *done* reach a value equal to the current value of the counter, say τ .¹³ During the execution, the *certification* variables (including *done*) have values with classes 0, 1, 2, or 3; writers on the control variables are triggered by appropriate values in other variables they depend on, as sketched below.

Apart from the n *Inst-i* nodes, there are three other nodes, one for each of *done*, *odd*, and *even*.¹⁴ We describe briefly what a processor does depending on the type of the node it picks.¹⁵ Our description is informal and grossly oversimplified to save on space.

- Node *Inst-i*. The actor reads the global *Rollback* flag. If it is raised, it will start the rollback, otherwise it proceeds as follows. It reads the copies of *even*. Using Fact 1 it attempts to estimate the current value; if it succeeds, let it be τ ; if it does not succeed, it exits the procedure. If the processor does succeed, it then executes the current phase for *Thread*[i]. It can do so if at least one copy of each of the (program) variables it has to read to execute the phase is unmarked. If this condition does not hold and the counter has not yet been advanced it will raise a global *Rollback* flag. Otherwise, it executes the appropriate writes in the variables, while performing the lateness test by reading *odd* and relying on Fact 2.
- Node *done*. Its value is (ultimately) supposed to

¹³This can be implemented by using a certification scheme such as in [MSP90].

¹⁴As stated above, one generally needs additional nodes to act on auxiliary data structures to *certify* that all of *Inst-1, Inst-2, ..., Inst-n* are done.

¹⁵It is easily seen that the two nodes *done* and *odd* can in fact be replaced by a single node.

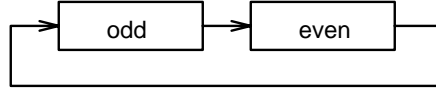


Figure 2: The Counter in isolation

2.2.2 Crucial properties

Intuitively, the idea is that the (most of the unmarked) values of the copies of *even* progress through the sequence of positive even numbers, while (most of the unmarked) values of the copies of *odd* progress through the sequence of positive odd numbers. The actual behavior especially the interaction between the values in *odd* and *even* is quite subtle. (The counter can “flicker”.) As we will see, the value of the counter is defined by its historical behavior and *not* by a snapshot in physical time.

Lemma 2 Only even values can be written in *even* and only odd values can be written in *odd*.

One important thing to prove is that a copy in which the value written is “too small” is never unmarked if a higher value had already been written into it.

Lemma 3 If a copy of one of the two variables is unmarked with some value at some time instance, then no higher value has ever been written in it previously.

Proof Assume otherwise. Let t_1 be the first instance in time in which the lemma is violated. Then, without loss of generality, for some copy u of *odd* there exist values σ and ρ and time instances t_3 and t_2 such that: at time t_3 the value ρ was written in u , at time $t_2 > t_3$ the value $\sigma < \rho$ was written in u , at time $t_1 > t_2$ the variable u was unmarked, and no value was written in u during the interval $(t_2, t_1]$. (If some some value was written, by another processor, during the interval $(t_2, t_1]$ in u unmarking does not apply as $u.pid$ no longer points to the processor being considered.)

As ρ was written in u at time t_3 , then at some time instance $t_4 < t_3$, the value $\rho - 1$ was written in at least $4/5$ copies of *even*. Now, for u to be unmarked with the value of σ at time t_1 , during the time interval (t_2, t_1) , the writing processor found more than $1/2$ unmarked copies of *even* with values of less or equal to $\sigma - 1$ and therefore less than $\rho - 1$. Thus in at least one copy of *even* the lemma was violated before t_1 , contradicting the definition of t_1 . \square

We will use *parity* to denote one of the variables *odd* or *even*.

Definition 1 We will say that the value of *parity* became some τ at time t , if t is the first (physical) time instance in which one of the following conditions hold:

1. At time t there are more than $4/5$ copies of *parity* so that each of these copies at some instance of time during the interval $[0, t]$ had an unmarked value of τ .
2. By time t , some writer wrote and unmarked all the copies of *parity* with the value of τ . Furthermore, at time t there are more than $1/2$ copies of *parity* so that each of these copies at some instance of time during the interval $[0, t]$ had an unmarked value of τ .

Definition 2 For any even value τ , we will say that the value of the counter became some τ at time t , if t is the (physical) time instance in which the value of *even* became τ . We will refer to such time t as critical and denote by $t(\tau)$.

Lemma 4 If *parity* has a value of $\rho > 2$ at some instance of time t , then for each $\sigma < \rho$ and for the variable *parity*(σ) (this stands for the variable, *odd* or *even* whose name is the parity of σ) there was an instance in time in which *parity*(σ) had the value of σ .

Proof Define

$$\overline{\text{parity}} = \begin{cases} \text{even} & \text{if } \text{parity} = \text{odd} \\ \text{odd} & \text{if } \text{parity} = \text{even} \end{cases}$$

Then in order for *parity* to have the value of ρ , the variable $\overline{\text{parity}}$ had to have the value of $\rho - 1$ previously. By formalizing this argument the lemma is proved by induction. \square

We note two points. First, we only associate even values to the counter. Second, of course (some of) the copies of *even* with the value of τ may be immediately overwritten by a lower value, thus making it impossible in general to determine that the counter’s value became τ previously.

Definition 3 We will say that the counter’s value at physical time t is τ if and only if $t(\tau) \leq t < t(\tau + 2)$.

In fact, we have shown that if the value of *odd* became some τ then the next different value it can reach must be $\tau + 2$. Meanwhile it can “flicker” between τ and undefined. (It does not go down or skip an odd value, but can possibly stay stuck.) Also note, that an outside observer may not be able to determine what the value of the counter is *even if the current values of all the locations of the memory are available instantaneously*.

Theorem 1 Let t_1 and t_2 be two instances in time such that: $t_1 \leq t_2$, at time t_1 *even* had a value of τ , and at

- 5.1. $[val'', status'', pid''] := x^{(i)}$
- 5.2. check if $val' = val''$ and $pid' = pid''$ and $status'' = good$
- 5.3. if yes then exit with success
- 5.4. if no then exit with failure

Lemma 1 If the lateness test in 3 of the writer protocol fails, then no processor will ever successfully read the value written in 2.

2.2 The global counter

We first describe the counter in isolation. We will build a counter from whose behavior the system’s logical time can be extracted. We refer the reader to Fig. 2. The counter consists of two variables: *odd*, whose copies are not memory-marked and initialized to 1; *even*, whose copies are not memory-marked and initialed to 2. The variables can be thought as nodes in a circular list.

2.2.1 How the counter changes in time

The system of course operates in *physical time*. This is time as seen by an outside observer for whom it “flows” naturally through the sequence of natural numbers. At each instance in time, the outside observer can see the complete state of the memory. The various endogenous actors start and complete their actions at integer time instances only, although they are not aware of the passage of time and the action can take an unbounded amount of time.

It is rather difficult in general to decide what the value of the counter is *even when all the variables and all the copies are observed at a single instance of physical time*. Therefore we will proceed by describing how the values of the copies change in time and then describe the meaning we attach to the values.

We describe informally the procedure a writer uses to act on the counter. For this it will be useful to classify variables according to the values in its copies.⁹

- A variable is of *class 0* if at most 7/10 of its copies are unmarked.
- A variable is of *class 1* if more than 4/5 of its copies have the same unmarked value; we will say that the variable has this class 1 value.
- A variable is of *class 2* if it is not of class 0 or class 1 and more than 1/2 of its copies have the same unmarked value; we will say that the variable has this class 2 value.
- Otherwise it is of *class 3*.

Behavior of a writer on the counter:

⁹In the following 1/2, 7/10, and 4/5 are acceptable parameter choices; others are possible

1. The writer randomly picks up one of the two variables *even* or *odd*. Without loss of generality, let it be *even*; if it is *odd*, exchange “even” with “odd.”
2. The writer reads one by one all the copies of *odd*. It decides what the class of the variable is, “as perceived by it.” If it is of class 0 or 3, it drops out.¹⁰ Otherwise it detects a value say τ , which is either of class 1 or class 2. (As we will see, all the copies of *odd* will always be odd.) The writer exits if it “remembers” writing all of *even* with $\tau + 1$ or if it aborted writing $\tau + 1$ before completing, because of a failed lateness test as described below (in item 2.1.2). Otherwise, we consider two cases based on the class of *odd*:

2.1. *odd* was of class 1. The writer proceeds through all the copies of *even*. For each copy it does the following (as long as it has not exited):

2.1.1. Prepares the ticket for writing the copy and writes $\tau + 1$ in the copy while processor-marking it.

2.1.2. It reads all the copies of *odd* and tests that *odd* is a variable of class 1 or 2 with the value of τ . If the condition holds, then it proceeds to the next copy of *even*, thus processor-unmarking the copy it wrote (which of course may have already been marked and also possibly unmarked by another processor thus making this unmarking irrelevant).¹¹ If the condition does not hold, it memory-marks the copy and exits the procedure.

2.2. *odd* was of class 2. If the writer wrote τ in all the copies of *odd* or if it now finds an unmarked value of $\tau + 1$ in *even* then it proceeds exactly as if *odd* were of class 1. Otherwise it exits.

Jumping ahead, we elaborate briefly and informally. We will say that the variable say *odd*, has reached a certain value (τ) if some processor wrote τ in all of its copies and more than 1/2 are unmarked with this value, or more than 4/5 of its copies had τ written and unmarked. A processor, if it is not too late can write $\tau + 1$ in say *even* if it has evidence that *odd* has reached the value of τ . This evidence can combine information about the states of the individual copies of *odd* and/or *even* with the information about the historical behavior of the processor itself.

¹⁰In the full implementation of the protocol, sketched in Section 2.3, if the variable is of class 0 a global rollback will be initiated.

¹¹For brevity’s sake, we will not be careful in making this distinction in the rest of this writeup.

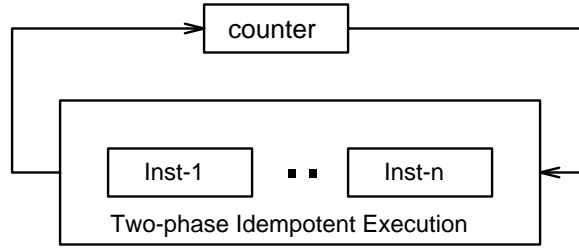


Figure 1: TIES execution of an n -way parallel step and its interaction with the counter

a one-copy variable called $ticket[pid]$. It is a pair of two fields: $ticket[pid].var$, $ticket[pid].copy$, where var is a name of a variable and $copy$ is a number identifying a copy of the variable. Thus, in fact the variable $ticket$ specifies a specific $var^{(copy)}$. We will say that $var^{(copy)}$ is processor-marked if and only if $ticket[var^{(copy)}.pid] = (var, copy)$, that is the copy and some processor “point” at each other. Note that a copy can be processor-marked by at most one processor.

Any processor will be able to read $ticket[pid]$ but only $P[pid]$ can write it. The purpose of $ticket[pid]$ is to record which $var^{(copy)}$ processor $P[pid]$ will write next.⁸

Let us describe informally how a processor $P[pid]$ writes some copy $var^{(copy)}$. We indicate by t_1, t_2, \dots the various time instances in which actions take place. At t_1 it writes $ticket[pid] := (var, copy)$. At t_2 it writes $var^{(copy)} := (val, good, pid)$. Note that the mere writing of the copy made it dubious. Now $P[pid]$ performs some lateness check to detect if the writing was too late. We consider two cases:

1. Assume that the the writing was not too late. Then the processor proceeds to another action. As a side effect it may modify $ticket[pid]$ thus making sure that $var^{(copy)}$ is no longer processor-marked by it. Note that if at some previous time instance $t_3 > t_2$ the copy was processor-marked by another processor $P[pid']$, then the changing of $ticket[pid]$ does not have any influence on the status of $var^{(copy)}$. Thus this (processor-)unmarking can only influence the value that the processor itself wrote.
2. Assume that the writing was too late. Then the processor can simply “hang on” permanently to the copy by not modifying its ticket. The copy will be processor-marked, and thus dubious, until another processor writes it. However, this prevents the processor from doing useful work. Instead, the processor writes $var^{(copy)} := (val, dubious, pid)$ thus memory-marking the copy and then proceeds to

another action while possibly modifying its ticket. Note that if at some previous time instance $t_3 > t_2$ the copy was written by another processor $P[pid']$, then by memory-marking the copy $P[pid]$ has possibly invalidated a correct copy, but that is acceptable.

The critical point is that a possibly incorrect copy is always marked as dubious. We need to specify more precisely how a variable, that is all its copies, is written, and how does a reader detect reliably that a copy is dubious. A major point that will be addressed later is how to perform the lateness test. This will require a subtle construction of a self-timed counter.

Writer Protocol

(The procedure for writing $x^{(i)}$ by $P[pid]$):

1. $ticket[pid] := [x, i]$
2. $x^{(i)} := [value, good, pid]$ (* The copy is *processor-marked* *)
3. perform a lateness test (* details depend on the writer’s task; specified later *)
4. if test succeeded then exit with success
5. if test failed then
 - 5.1. $x^{(i)} := [value, bad, pid]$ (* The copy is *memory-marked* *)
 - 5.2. exit with failure

We next proceed to describe how a processor reads some $x^{(i)}$ and determines whether it is dubious.

Reader Protocol

(The procedure for reading $x^{(i)}$ by $P[pid]$):

1. $[val', status', pid'] := x^{(i)}$
2. $[x', i'] := ticket[pid']$
3. check if $x' = x$ and $i' = i$
4. if yes then exit with failure
5. if no then

⁸However, $P[pid]$ does not indicate in its $ticket[pid]$ that it will write $ticket[pid]$ next. $ticket[pid]$ is only used before $P[pid]$ writes to *other* variables.

by positing that processor speeds are stochastic variables [CZ90], or that the instruction sets of the processors are augmented with powerful *read-modify-write* instructions [MSP90], [Su91].⁶ As mentioned earlier, the fine-grained implementation of these primitives involves busy-waiting and possible starvation [DSB88]. This should come as no surprise, since Herlihy [He88] has shown that these primitives are universal for wait-free implementations. We note that the atomic read and write instructions that we use in our schemes are exempt from this limitation for two reasons. *First*, since these instructions are *not universal* in the above sense, arbitration between conflicting reads and writes is much simpler than in the case of the read-modify-write primitives. *Second*, our computations will progress so long as some of the (replicated) reads and writes to a fuzzy variable succeed with high probability. The above condition is realized through memory replication as well as the possible use of randomization in the reader and writer protocols.

2 Execution of parallel programs on asynchronous machines

In [KPS90] the TIES methodology was introduced for transforming arbitrary PRAM program \mathcal{P} into a program \mathcal{P}' that executes correctly on a synchronous PRAM in which the processors may fail and possibly be restored. \mathcal{P}' is obtained by first restructuring each (parallel) instruction of \mathcal{P} into two phases. Second, an arbitrary algorithm for the CWA is used to reschedule the available processors during the execution of the phase to take over the work of processors that might not have done their work. At the end of the execution of the phase some flag variable *done* is assigned some value to indicate that the phase is completed. The execution of a phase was structured to assure a certain *idempotence* property: each atomic instruction of the phase can be executed arbitrary number of times as long as it is executed at least once. This is critical, because after it is detected that a processor has failed, it is still not possible to know exactly how much work it has done, so being conservative, possibly some work is done more than once.

The fact that the processors are synchronous made it trivial to assure that no processor will execute an atomic action too late. That is, if the action was started during a particular phase, then if the processor has not failed during its execution, the action was completed during one physical time unit and therefore still within the phase.⁷

⁶Examples of such instructions include the loose atomicity of *fetch-test-store*, *test&set*, *compare&swap* and *fetch&add*.

⁷We can of course assume without loss of generality that on

Running \mathcal{P}' on a machine with asynchronous processors does not have to result in a correct execution. Although TIES can be made to guarantee that each atomic action that needs to be executed before the phase ends will indeed be executed at least once by that time, this is not sufficient. There may be slow processors who have also started to execute some atomic action, say a write, and then did not complete for a long time. At the time such a processor completes its write, it may overwrite a more recent value thus *clobbering* it. We need to devise a methodology for detecting such a clobberer and for preventing it from having an effect on the progress of the computation.

We must be able to detect that an action completed too late. This is done by means of consulting a global *counter*, which counts the number of phases executed so far. Somewhat oversimplifying, we can say that an actor consults the counter before and after executing an “important” action. If the value of the counter has been incremented, during the action, the action completed too late and its effects must be logically ignored. Fig. 1 describes the system model at the detail needed at this point.

2.1 Variables and copies

As discussed in the previous section, each variable will be stored fuzzily as μ copies, unless we state explicitly otherwise. The i th copy of variable x will be denoted by $x^{(i)}$. As discussed above a copy of a variable may store an incorrect value, which may have been written too late by a processor. Thus in keeping with our conservative approach, a value can be classified as *good* if it is guaranteed to be correct, or *dubious* that if it possibly incorrect (obsolete). We will utilize two ways of indicating that a variable is dubious. One way will utilize a specific field in the variable (which will be structured as a record). The other way will utilize a very simple linked list structure connecting the variable with the processor that wrote it last. This two ways of indicating that the variable is dubious are called respectively *memory marking* and *processor marking*. We will elaborate on this shortly, but first we describe the structure of a copy.

Each copy $x^{(i)}$ is a triple of three fields: $x^{(i)}.val$, $x^{(i)}.status$, $x^{(i)}.pid$. In effect, the first field indicates the “logical” value of the variable x as stored in copy $x^{(i)}$, the second field whose value is either *good* or *dubious* indicates whether $x^{(i)}.val$ is good or dubious, the third field states that this copy was written by processor $P[pid]$. We will say that $x^{(i)}$ is memory-marked if and only if $x^{(i)}.status = dubious$.

We now proceed to show how processor-marking is indicated. With each processor $P[pid]$ we associate

a synchronous machine all atomic actions take one time unit exactly.

the original program assuming up to $n/\log n$ asynchronous processors. Conditions for setting the *Rollback* bit are sketched in Section 2.3, and effect of rolling back on the overall complexity is discussed in Section 3.

1.2 Comparison to previous work

Asynchrony has been studied extensively, particularly in the distributed computing setting—the system is loosely-coupled, with each processor having private memory. Processors communicate only by passing messages. Fischer, Lynch and Paterson [FLP85] studied a fundamental problem, the *consensus* problem, in this model. They showed that consensus is deterministically impossible in this model, when reads and writes are the only atomic operations. Remarkably enough, the work of Ben-Or [Be83] and Rabin [Ra83] for example, demonstrated that by using randomization, these and other agreement problems can be solved in the same models. In this same setting, let a *wait-free* protocol be one where each thread makes progress independent of the speeds of the other threads.³ Herlihy [He88] showed that even simple data structures such as queues and stacks do not have wait-free implementations. On the other hand, he also showed that when the instruction sets are augmented with powerful *read-modify-write* primitives such as *test&set*, *compare&swap*, and *fetch&add*, every structure can be implemented in a wait-free manner. Thus, he terms these primitives *universal*. Furthermore, using randomization, Herlihy has constructed efficient implementations of “long-lived concurrent objects” [He91c].

Progressive computations are similar in spirit to the notion of a wait-free computation. However, in a wait-free computation, typically no distinction is made between the computation itself and the processors that execute it. In contrast, we treat the state of the *logical* program’s execution, as represented by the collection of processes, to be completely distinct from that of the *physical* processors that execute it. Thus, we differ from the settings traditionally investigated in the context of concurrent systems, where some form of “process-binding” is assumed.⁴ There are some exceptions to this observation, where “rescheduling” has been implicitly adopted such as [He91a] and [P189], for example.

In a progressive computation, a given processor can, over the course of the computation, be rescheduled across several different threads. Hence, this processor’s view of *time* is defined entirely by the (logical) state of the thread that it is currently executing. (Our notion of logical versus physical time is identical to that described

³Clearly, any protocol that makes use of *locks* is not wait-free since lock implementations involve mutual exclusion, and the processor holding the lock can be arbitrarily slowed down.

⁴We note that this assumption is not always made *explicitly*, but is quite often used *implicitly* in essential ways.

by Lamport [La78].) This independence from the physical time or memory associated with any one processor allows us to use them interchangeably, and achieve progressive computations.⁵ In using them interchangeably, we need to ensure that processors can interpret logical time correctly, so that their actions change the state of the computation only within the correct temporal context.

Let us now briefly summarize the past work in program transformations for tightly-coupled systems. As stated before, it was shown in [KPS90] that *arbitrary* PRAM programs can be efficiently transformed to execute in a progressive manner, on a synchronous, *fail-stop* PRAM. In this model, there are n initial processors, each of which can fail. The model was introduced by Kanellakis and Shvartsman [KS89], who designed *work-efficient*, robust algorithms for specific problems in this model, including the *Write-All* problem; informally, work is the sum of the number of live processors in each step of the algorithm’s execution. In the *Write-All* problem, there is an array $x[1..n]$ given as input. The job is to write the value 1 in each location of this array. In [KPS90], Kedem, Palem, and Spirakis showed that an arbitrary step of a PRAM program can be correctly executed in essentially the same time and work complexities as executing *any* algorithm for the *Certified Write-All* problem twice on a fail-stop PRAM. The *Certified Write-All* problem (or *CWA*) is like the *Write-All* problem, with the addition that when the work is completed, a *done* bit is set. In effect, they showed how to efficiently reschedule work to obtain progressive computations by “bootstrapping” on any algorithm for *CWA*. This rescheduling strategy, which they call the *Two-Phase Idempotent Execution Strategy*, or *TIES*, can be used to correctly transform arbitrary PRAM programs to run on fail-stop PRAMs. Later, Kedem, Palem, Raghunathan, and Spirakis [KPRS91] improved upon this scheme by combining probabilistic and deterministic *CWA* algorithms to obtain extremely efficient program transformations, including models with processor *restarts*. Deterministic variants of this model where fail as well as restart have been studied by Kanellakis and Shvartsman [KS91].

As noted before, transforming PRAM programs to run correctly and efficiently on APRAMs is far more difficult to achieve. Of course, in studying the relative merits of different program transformation techniques, we need to consider not only the performance overhead incurred, but also the restrictions assumed about the target machine. Past results in this area have either assumed that the amount of asynchrony is limited,

⁵This independence also frees us from relying on *clock synchronization* techniques ([La78], [LS85], [DHSS84], [LL84], [C89]) that typically rely on underlying “physical” clocks that advance with limited *drift* relative to each other. Specifically, our computations will terminate correctly even if the processor clocks drift arbitrarily apart, since for a computation to be progressive does not depend on a notion of physical time.

achieving progressive transformations of ideal programs to run on asynchronous machines have assumed that the instruction sets of the processors are augmented with powerful *read-modify-write* instructions, such as the *test&set*, *compare&swap*, or *fetch&add*. For example, Martel, Subramonian and Park [MSP90] noted that the TIES strategy of [KPS90] for processor rescheduling will correctly transform PRAM programs to execute on an APRAM, *provided “loose atomicity” is assumed*. Specifically, they assumed that processors can issue a *Fetch-Test-Store* instruction that is *guaranteed* to complete before a total work of $O(n)$ is done by other processors, where n is the width of the parallel program. This enables them to interrupt and prevent a tardy processor from performing obsolete writes. Thus, in effect, each processor is able to resynchronize once during an execution of each step of the given ideal PRAM’s computation. An alternate assumption uses a modified *compare&swap* instruction to obtain *tagged* memories. There, every memory location holds a value together with a timestamp. Values associated with older timestamps are not allowed to overwrite locations that hold values associated with later timestamps; this ensures that there will be no clobber [Su91].

Given the fact that these read-modify-write primitives facilitate simpler program transformations that do not have to deal with clobbers, why do we seek a different solution? An important reason for doing so is a subtle yet serious hidden cost associated with deriving progressive computations, using the above-mentioned primitives. In current machines, the read-modify-write primitives are invariably implemented by enforcing some kind of mutual-exclusion among multiple writers [DSB88]. Specifically, when several writers attempt to access the same location by using one of these primitives, their requests are in effect serialized at the level of the fine-grained implementation by using locks. Thus, using these primitives would involve busy-waiting and possible starvation. Therefore, the resulting computations are in fact not progressive. (We will discuss the issue as to why read/write instructions are not prone to the above problems in Section 1.2.) Our goal in this paper is to realize efficient progressive computations without recourse to mutual exclusion as a primitive, at *any level* of the program’s execution.

1.1 Summary of main results

We now summarize our main constructs, methods and results.

1. *Fuzzy variables*. For correct and progressive computations, we must be able to overcome the clobber problem, where some temporally obsolete value is written by a slow processor. To do this, we maintain several copies, (say $\mu > 4$) of each variable, so that with extremely high probability at least one

copy will hold an up-to-date value of the variable. For details, see Section 2.1.

2. *Reader and Writer Protocols*. We design protocols that enable processors to read from and write to these fuzzy variable. If a fuzzy variable takes on an *incorrect value*, i.e., all of its copies are clobbered, this will *always* be recognized. The protocols *consult a global counter*, which indicates the total number of instructions of the ideal PRAM program that have been executed so far on the asynchronous machine, using which processors decide if they are tardy. These protocols are described in Section 2.1.
3. *Randomization*. Given the existence of such a global counter, we show that any scheme that randomly reschedules work across the n threads can be utilized to give the following important property: the reader and writer protocols assure that the probability that a fuzzy variable takes on a possibly incorrect value, is very small (inverse polynomial in n).² This property holds for every value of μ that is at least $c \log n$, with c a suitable constant. We will be presenting our results in terms of an *oblivious* adversary. For our purposes, this means that the adversary sets the processor speeds without ever looking at the outcomes of the random moves made by the algorithm [BBKTW90]. The interplay of randomization with the rest of the computation is sketched in Section 2.3. They are analyzed and the and the over complexity are derived in Section 3.
4. *Global Counter*. Items (2) and (3) above depended on the (assumed) existence of a global counter that correctly indicated the progress of the program’s execution. The value indicated by this counter (which itself will have to be implemented on an asynchronous machine) will allow us to divide the absolute (physical) time into intervals in which the logical clock of the ideal computation has a fixed value. We will design protocols using which the asynchronously operating processors can reliably *consult* and *update* this clock. The counter *itself* is implemented as a set of fuzzy variables. The counter’s implementation and correctness issues are discussed in Section 2.2.
5. *Rollback and Complexity*. If a processor ever detects a possibly incorrect fuzzy variable, it sets a flag called *Rollback*. When *Rollback* is set, processors will restart the computation to complete it correctly. We show that the progressive program, as obtained by our scheme, has space complexity $O(\log n)$ times that of the original program, and expected work complexity $O(\log^2 n)$ times that of

²In other words, we consider schemes that pick a thread at random with probability $1/n$. One such scheme has been described by Martel et al. [MSP90].

concreteness the PRAM), so that they run on an asynchronous (possibly virtual) machine, prone to slowdowns. (Failures and restarts can be viewed as a special case of slowdowns, where the processors stop and restart in a clean state without any memory of their state when they stopped.) The transformed program must be *correct* and should not incur a large overhead—it should remain (almost) as *efficient* as the original program in its work, time, and space complexities. Furthermore, a novel and crucial feature that we expect the transformed program to possess is that it be *continually progressive* (henceforth referred to simply as *progressive*)—the computation itself will make correct progress without waiting for the failed or very slow processors to complete their work. The notion of progressive computation is related to, but different from, that of a *wait-free* computation, as discussed in [He88], [He91b]. We will discuss this relationship at a greater length in Section 1.2.

Our main result is a methodology that transforms *any* parallel program written for an ideal machine with n processors, to execute in a progressive manner on an *arbitrary* asynchronous machine. We emphasize that in this asynchronous machine *any* of the instructions can take potentially unbounded amounts of time to complete. The *only atomic* instructions that we postulate for accessing memory in the asynchronous parallel machine, are *reads* and *writes*. For concreteness, we can view this asynchronous machine as being an APRAM introduced by Cole and Zajicek in [CZ89] and by Gibbons in [Gi89].¹ Our transformations yield progressive executions of the ideal programs, with the space complexity increased by a factor of $O(\log n)$, and with the expected work complexity increased by a factor of $O(\log^2 n)$, assuming up to $n/\log n$ asynchronous processors. In the case where the asynchronous machine has up to n processors, the space complexity remains the same, and the work overhead increases to $O(\log^3 n)$. For these complexity results we assume that the original program has the exclusive write property; thus it follows that any CRCW PRAM program will execute on the asynchronous machine with (asymptotically) no further space overhead, and an extra $O(\log n)$ work overhead.

To facilitate the description of our results, we first note that our methodology for achieving a progressive computation relies on the ability to *reschedule* processors. Informally, rescheduling allows a fast (or live) processor to subsume the work being done by a slow (possibly failed) processor. In effect, the overall computation does not wait for *tardy* (or slow) processors to complete their work. Research on the topic of transforming arbitrary ideal programs to run on “realistic” models, started with the work of Kedem, Palem and Spirakis [KPS90]. In [KPS90], a methodology is given—using

¹ For further details regarding the design of algorithms for the APRAM, the reader can refer to [Ni91].

what they refer to as the *Two Phase Idempotent Execution Strategy* or *TIES*—for correctly and efficiently transforming *arbitrary* PRAM programs to execute in a progressive manner on a (synchronous) *fail-stop* PRAM (introduced by Kanellakis and Shvartsman in [KS89]). By taking advantage of the fact that the states of all the processors and threads are globally accessible in a shared-memory machine, the TIES strategy reschedules faster processors to subsume the work of slower processors. We will adopt the TIES strategy in this work as well, as the basis for rescheduling.

Unfortunately, the problem with rescheduling in a completely asynchronous setting such as ours, is that a tardy processor might reenter the computation too late and *clobber* the memory by writing an obsolete value into it. This renders the memory inconsistent, leading to possibly incorrect results. Therefore, the TIES strategy of [KPS90] is not sufficient in itself to transform arbitrary PRAM programs to execute on APRAMs, since clobbers can occur.

We now sketch our main ideas for overcoming this clobber problem. (Recall that we make no assumptions about the degree of asynchrony or about the instruction set of the machine—read and write instructions are the only primitives.) First, we introduce a notion of *memory replication*. Specifically, we will “implement” each (atomic) variable as a *replicated* entity, resulting in *fuzzy variables*. In a fuzzy variable, several copies of the original variable’s value are stored. A correct value of the variable is obtained whenever at least one copy of its fuzzy implementation has an unclobbered value, *and* if this copy can be *reliably identified*. By randomizing the writes to memory, we will ensure that at least one unclobbered copy of each variable exists with very high probability. It is worth noting that replication of variables is used also in other settings such as distributed databases, see [BHG87]. However, replication is used in a very different way in this paper.

Our protocols for reading and writing will assure that only correct (or unclobbered) values of all the variables will be obtained. In order to distinguish correct copies of variables from clobbered (or obsolete) ones, we need to identify if a writer’s action took place too late, thereby leading to clobber. To detect this, we construct a global counter to which the various readers/writers can refer and get relative timing information. The counter’s implementation is reminiscent of the protocols due to Rabin [Ra82] for solving the *choice coordination problem*. We implement the counter asynchronously using fuzzy variables. However, unlike writes to program variables that can refer to the counter for timing information, there is no other exogenous agent that the protocols can use, to check their timing correctness while working on this counter. Thus, the implementation and updates to the counter will be *self-referential!*

In contrast to the present work, previous attempts at

Efficient Program Transformations for Resilient Parallel Computation via Randomization

(Preliminary Version)

Z. M. Kedem* K. V. Palem† M. O. Rabin‡ A. Raghunathan§

Abstract

In this paper, we address the problem of automatically transforming *arbitrary* programs written for an ideal parallel machine to run on a *completely asynchronous* machine. We present a *transformation* which can be applied to an ideal program such that the resulting program's execution on an asynchronous machine is work and space efficient, relative to the ideal program from which it is derived. Above all, the transformation will guarantee that the ideal program will execute in a *continually progressive* manner on the asynchronous machine: the computation itself will make progress *without waiting* for slow or failed processors to complete their work. We ensure the above properties by requiring that *only read and write* instructions be *primitives* in the asynchronous machine; these instructions *are not universal*. Furthermore, the individual processors can get delayed for arbitrary amounts of time while execut-

ing any instruction. In contrast, previous work relied either on the asynchronous machine having universal *read-modify-write* instructions as primitives, or on limited asynchrony by restricting the relative speeds of the processors.

1 Introduction

A widely used and convenient model for developing parallel programs is the shared-memory parallel machine. In order to provide a clear logical framework to the programmer, in this model, n *ideal* processing elements (PEs, or simply processors) cooperate to execute a parallel computation by reading from and writing to a global or shared memory. No information is private to individual PEs in this model. Corresponding to the n processors, the execution of a parallel program flows as n *threads* of computation, where each thread represents the work of one of the idealized PEs. Since the threads interact, barriers are created periodically to *synchronize* them. In the extreme case, synchronization barriers exist after each parallel step, such as in the PRAM [FW78].

Although the ideal model described above is desirable from the viewpoint of program development, it does not correspond to the way parallel machines behave in practice. Typically, the processors on which the individual program threads are executing can work *asynchronously* at very different speeds for a variety of reasons including clock skew, interrupts and context switches, as well as network congestion. Throughout the rest of this paper, we will refer to these asynchronous processors as if they were physical entities, for the sake of convenience. However, we note that the idealized logical computation can in fact be executing on a virtual asynchronous machine, that is derived from an underlying physical host. Additionally, such a virtual processor may behave asynchronously due to the failure of its host physical processor, or it may disappear for long periods of time (due to host overloading for example).

We address the problem of *automatically* transforming programs written for an ideal parallel machine (for

*Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, USA; kedem@cs.nyu.edu. The research of this author was supported in part by NSF/DARPA under grant number CCR-89-06949 and by NSF under grant number CCR-91-03953.

†IBM Research Division, T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA; kpalem@watson.ibm.com.

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

§Computer Science Division, University of California, Davis, CA 95616, USA; raghunat@iris.ucdavis.edu. The research of this author was partially conducted while he was visiting the IBM T. J. Watson Research Center, and was also supported in part by NSF under grant number CCR-91-07847.