ABSTRACT: Theoretical research on parallel algorithms has focused on NC theory. This motivates the development of parallel algorithms that are extremely fast, but possibly wasteful in their use of processors. Such algorithms seem of limited interest for real applications currently run on parallel computers. This paper explores an alternative approach that emphasizes the efficiency of parallel algorithms. We define a complexity class PE of problems that can be solved by parallel algorithms that are efficient (the speedup is proportional to the number of processors used) and polynomially faster than sequential algorithms. Other complexity classes are also defined, in terms of time and efficiency: A class that has a slightly weaker efficiency requirement than PE, and a class that is a natural generalization of NC. We investigate the relationship between various models of parallel computation, using a newly defined concept of efficient simulation. This includes new models that reflect asynchrony and high communication latency in parallel computers. We prove that the class PE is invariant across the shared memory models (PRAM's) and fully connected message passing machines. These results show that our definitions are robust. Many open problems motivated by our approach are listed.

1. INTRODUCTION

As parallel computers become increasingly available, a theory of parallel algorithms is needed to guide the design of algorithms for such machines. It is felt by many users of parallel machines that the existing theory of parallel computations offers only scant help. The bulk of this theory, which explores the complexity class NC, seems only remotely relevant to present day parallel computers. Models seem to ignore issues that practitioners feel to be very significant, such as communication and synchronization overheads, and granularity.

This paper outlines an approach to the analysis of parallel algorithms that we feel may alleviate this problem without sacrificing too much in terms of generality and abstractness. We propose in §3 new definitions of efficient parallel algorithms: these are algorithms that achieve ("almost") linear speedup (in the number of processors) for "moderate" problem sizes. These definitions are compared and contrasted with the definitions underlying the NC concept, namely algorithms that provide fast parallel solutions (i.e. algorithms that use a polynomial number of processors to achieve polylogarithmic time). We submit that research on the design and analysis of parallel algorithms should focus on efficient parallel algorithms.

One obstacle encountered in such research is the variety of existing parallel computer architectures and the even greater variety of proposed parallel computation models. Distinctions among parallel computation models partly reflect real technological constraints.
such as memory granularity and contention, asynchronism, large communication latency, and restricted communication geometry. In §4 we explore the relationships between various parallel computation models, focusing on those that reflect significant technological constraints. Although these models are distinct in their computing power, they define the same class of efficient parallel algorithms. Each weaker model can efficiently simulate any stronger model, provided that the stronger model has polynomially more processors. This implies that efficient parallel algorithms defined on a strong model yield efficient parallel algorithms for the weak model (although the degree of parallelism that can be achieved efficiently on the weak model is smaller). Thus, it is possible to develop a unified theory of parallel efficient algorithms that ignores distinctions among models. Finally we suggest in §5 open problems and research directions that are motivated by the approach developed in this paper.

The following are the major contributions of this paper:

- New classes of parallel algorithms are defined and shown to be robust:
  1. PE: the class of problems with efficient parallel algorithms.
  2. PE*: the class of problems with almost efficient parallel algorithms.
  3. GNC: a class of problems that generalizes NC.

- New parallel computation models are defined; efficient simulations across models are given:
  1. An efficient deterministic simulation of Concurrent Read, Concurrent Write (CRCW) PRAM's by Exclusive Read, Exclusive Write PRAM's.
  2. An efficient probabilistic simulation of CRCW PRAM's by message passing, fully connected machines (DCM model).

- Classes of problems are proved invariant under classes of parallel models:
  1. The class PE* is proved invariant under a wide class of models, including PRAM's (shared memory machines) and Butterfly machines.
  2. The class PE is proved invariant under the PRAM models.
  3. The class PE is proved invariant for probabilistic algorithms under the PRAM and DCM models.

2. A REVIEW OF PARALLEL COMPLEXITY

2.1. The PRAM or Paracomputer model

A strongest parallel computer model is the PRAM (Parallel Random Access Machine) or Paracomputer model [FW,Sc]. Each processor is a unit-cost RAM, as defined in [AH], with a few minor modifications; all access a common memory. Each processor is controlled by its own, separate program and has its own local set of registers (at least two are needed). Instructions are either zero-address (registers operands) or one-address (one operand in shared memory). Test (and jump) instructions have only register operands. We prohibit indirect addressing, using addresses in shared memory, but allow indexed addressing, using a local register as index.

There are many variants of this model. One of the strongest is the Synchronous Priority RMW PRAM (or, for short, the Strong PRAM): Processors proceed in lock-step, executing one instruction per time unit. For each basic binary operation of the RAM model (Add, Subtract, Multiply, and Divide) we have a corresponding Fetch&Op memory access operation that executes an atomic Read-Modify-Write cycle. The execution of an instruction Fetch&Op v R by a processor updates memory variable v to a new value v Op R, and sets (local) register R to the old value of v. If several processors concurrently access the same variable, all of the accesses execute at that time step;
the outcome is as if these accesses occurred sequentially in the order of the processor identifiers. For the noncommutative arithmetic operations (subtraction and division) we also include $\text{Fetch} \& \text{InvOp}$ operations; e.g. $\text{Fetch} \& \text{InvDiv} v R$ sets $v$ to $R/v$, and returns to $R$ the old value of $v$. One could also include other Read-Modify-Write operations; see [K1] for a discussion of such operations.

We are going to use the strong PRAM as our basic model in order to analyze parallel algorithms for several reasons: (1) It is a natural generalization of the RAM model. (2) It is a reasonable model: the number of arithmetic operations done by $p$ processors per cycle is at most $p$; this includes Read-Modify-Write accesses to shared memory. (3) It is the strongest reasonable model that has been considered in the literature; all other models can be seen as restricted versions of this model. Thus, a complexity theory based on this model has some applicability to all other models (e.g., lower bounds for this model apply to all models). Other models and several variants of the PRAM model are discussed in §4. As we shall see later, the actual choice of (PRAM) models makes little difference to our theory.

2.2. Parallel Algorithms

For a sequential algorithm $A$, we write $t^A(N)$ for the running time on a problem of size $N$, but shorten it to $t(N)$ when the algorithm is clear from context. For a parallel algorithm $B$, we write $T^B_p(N)$ for the running time on a problem of size $N$ on a $P$ processor machine, but shorten it to $T_p(N)$ when the algorithm is clear from context. Note that the parallel algorithm when run on one processor may be different than the sequential algorithm, so it may be that $T_1(N) \neq t(N)$; we assume, however, that $T_1(N) \geq t(N)$.

It is not the performance per se of a parallel algorithm that interests us, but, rather, the relative improvement as compared to a sequential algorithm, i.e. the value of $T_p(N)$ as compared to the value of $t(N)$. We shall usually choose our yardstick to be the best existing sequential algorithm. Note, however, that we must relativize our definition to a fixed sequential algorithm, as there may not be a sequential algorithm with a “best” performance, not even up to a constant factor (this follows from Blum’s speedup theorem [Bl]).

The two figures of merit of the parallel algorithm are its speedup $S_p(N) = t(N)/T_p(N)$ and its efficiency $E_p(N) = t(N)/(P \cdot T_p(N))$. The first measures the improvement in running time achieved by $P$ processors, and the second measures the ratio between the amount of work done by the sequential algorithm and the total amount of work done by the parallel algorithm.

3. EFFICIENCY CLASSES

In this section, we define weak and strong efficiency, giving several different but provably equivalent definitions in each case.

3.1. Definitions

Definition. A parallel model is self-simulating if a $Q$ processor version can simulate one step of a $P$ processor version in time $O\left(P/Q\right)$ for $P > Q$ and in time $O\left(1\right)$ for $P \leq Q$. (This simplifies to time $O\left(\max(1,P/Q)\right)$.)

Assumption. The theorems in this section assume a self-simulating model of parallel computation.

Definition [GK]. In a size-independent parallel algorithm the problem size and number of processors are independent parameters. In a size-dependent parallel algorithm the number of processors used is a function of the input size. We shall then denote by
$P(N)$ the number of processors used to solve a problem of size $N$; $T(N)$ is the running time (with $P(N)$ processors).

**Definition (Efficient).** A size-dependent parallel algorithm is *efficient* (relative to a sequential algorithm with time $t(N)$) if its efficiency is bounded away from zero, i.e. $E_{P(N)}(N) = t(N)/(P(N) \cdot T(N)) \geq c$, for some constant $c > 0$.

This definition does not say anything about the amount of parallelism; in particular, a sequential algorithm is always efficient relative to itself. We want a parallel algorithm to use increasing amounts of parallelism as the problem size increases:

**Definition (Weak Parallel Efficient) [GK, VS].** A size-dependent parallel algorithm is *weakly parallel efficient* (relative to a sequential algorithm with time $t(N)$) if it is efficient ($P(N) \cdot T(N) = O(t(N))$) and $\lim_{N \to \infty} P(N) = \infty$.

Thus, a size-dependent parallel algorithm is weakly parallel efficient if it achieves speedups proportional to the number of processors used, and the speedup grows to infinity as the input size goes to infinity.

**Lemma 3.1 [K2].** The following are equivalent:

1. A size-dependent parallel algorithm is weakly parallel efficient (relative to a serial algorithm).
2. For some positive constant $c$ and monotone nondecreasing function $f$, the associated size-independent parallel algorithm has running time $T_P(N) \leq c t(N)/P + f(P)$.
3. For some positive constant $c$ and monotone nondecreasing function $f$, the associated size-independent parallel algorithm has running time $T_P(N) \leq c t(N)/P$ for $t(N) \geq f(P)$.

An algorithm may be parallel efficient according to the last definition, yet allow efficient use of $P$ processors only for prohibitively large problems (e.g. if $f(P) = 2^P$); we want the algorithm to start being efficient at moderate problem sizes. We therefore adopt the following amended definition:

**Definition (Strong Efficiency).** A (size-dependent) parallel algorithm is *strongly parallel efficient* if it is efficient ($P(N) \cdot T(N) = O(t(N))$) and $T(N) = O((t(N))^\alpha)$ for some constant $0 < \alpha < 1$.

A strongly parallel efficient algorithm achieves a polynomial speedup, reducing the running time from $t(N)$ to $(t(N))^{\alpha}$. It efficiently uses $P(N) = \Omega((t(N))^{1-\alpha})$ processors.

**Lemma 3.2 [K2].** The following are equivalent:

1. A size-dependent parallel algorithm is strongly parallel efficient (relative to a serial algorithm).
2. For some constant $k$, the associated size-independent parallel algorithm has running time $T_P(N) = O(t(N)/P + P^k)$.
3. For some constant $k$, the associated size-independent parallel algorithm has running time $T_P(N) \leq O(t(N)/P)$ for $t(N) \geq P^k$.

We will henceforth denote “strongly parallel efficient” algorithms simply as “parallel efficient” algorithms.

### 3.2. The class PE

Now that we have defined efficient algorithms, a corresponding complexity class can be defined.
Definition. A problem is in the class PE (Parallel Efficient) if, relative to any sequential algorithm that solves the problem, there exists a strongly parallel efficient algorithm that solves the same problem. That is, a problem is in the class PE if for any sequential algorithm $A$ that solves this problem there exists a parallel algorithm $B$ that solves the same problem and positive constants $c$ and $k$ such that $T_B^P(N) \leq ct^A(N)/P + P^k$.

The last definition can be simplified for problems that have a well defined sequential complexity: The sequential complexity of a problem is $t(N)$ if the problem can be solved by a sequential algorithm with running time $t(N)$; and for any sequential algorithm $A$ that solves the problem $t^A(N) = \Omega(t(N))$. When this obtains, then the problem is in PE if it has a parallel algorithm with running time $O(t(N)/P + P^k)$. Note, however, that not every problem has a well defined sequential complexity $|\mathcal{B}|$.

Problems are in PE if they can be solved in parallel polynomially faster than sequentially, with a number of processors proportional to the speedup achieved. To illustrate the difference between the class PE and the class NC, consider a hypothetical problem that has sequential complexity $t(N) = \Theta(N^2)$. The problem is in NC if there is an algorithm that achieves parallel running time $T(N) = O(\log^{O(1)}(N))$, using $P(N) = N^{O(1)}$ processors; the problem is in PE if there is a parallel algorithm that achieves, for some $\alpha > 0$, parallel running time $T(N) = O(N^{2-\alpha})$ using $P(N) = N^\alpha$ processors. The improvement in running time we require for PE is modest compared to NC; on the other hand there is a more stringent constraint on the number of processors used.

Since the definition is relative to serial running time, it is not restricted to problems in P; a problem with exponential serial complexity is in PE if parallelism can efficiently decrease the running time by a polynomial amount; e.g. if running time can be reduced from $\Theta(2^N)$ to $\Theta(2^{N/2})$, using $\Theta(2^{\sqrt{N}})$ processors.

3.3. Other Classes

Our definition of an efficient parallel algorithm is very stringent. In some cases slight inefficiency can be tolerated, when a significant reduction in running time is essential. Also, less stringent requirements result in a definition that is robust across a wider range of models. This motivates the following extension to our previous definition.

Definition. A (size-dependent) parallel algorithm is $(f,g)$-efficient relative to a sequential algorithm if $T(N) \leq f(t(N))$ and $E_{P,N}(N) \geq 1/g(t(N))$. I.e., the parallel algorithm achieves an $f$-reduction in running time with a $g$-inefficiency, as compared to the sequential algorithm.

A complexity class can be accordingly defined:

Definition. The class PE$(f,g)$ consists of problems such that, relative to any sequential algorithm that solves the problem, there is an $(f,g)$-efficient parallel algorithm that solves the problem. As usual, PE$(F,G)$ denotes the union of classes PE$(f,g)$, for $f \in F$, $g \in G$.

The definition of PE given in the previous section corresponds to the case where $f(x) = x^\alpha$, with $0 < \alpha < 1$, and $g$ is a constant. Thus, PE = PE$(x^{1-O(1)},O(1))$.

We relax the efficiency requirement slightly to allow a polylogarithmic inefficiency:

Definition. The class PE$^*$ consists of problems such that, relative to any sequential algorithm that solves the problem, there exists an $(x^\alpha,\log^{O(1)}x)$-efficient parallel algorithm that solves the problem, where $0 < \alpha < 1$; i.e., PE$^* = PE(x^{1-O(1)},\log^{O(1)}x)$.

A problem is in PE$^*$ if for any sequential algorithm for the problem with running time $t(N)$ there is a (size-dependent) parallel algorithm and positive constants $\alpha < 1$ and $k$ such that $T(N) = O((t(N))^{\alpha})$ and $E_{P,N}(N) = \Omega(\log^{-k}(t(N)))$. 


The class NC can be generalized in a similar manner:

**Definition.** GNC is the class $\text{PE}(\text{polylog}, \text{polynomial})$. I.e., GNC is the class of problems where a quasi-exponential reduction in running time can be achieved, with a polynomially bounded inefficiency. (We use the term *quasi-exponential* to mean the inverse of a polylogarithmic function.) Formally, a problem is in GNC if for every sequential algorithm there is a parallel algorithm and a positive constant $k$ such that $T(N) = O(\log^k(t(N)))$ and $E_p(N) = \Omega((t(N))^{-k})$. The last condition can be replaced by the condition $P(N) = O((t(N))^{1})$. Thus, a problem is in GNC if a quasi-exponential reduction can be achieved in the sequential running time, using a number of processors polynomial in the sequential running time.

For such problems that are “strictly” polynomial time our definition of GNC agrees with the definition of NC. Note, however, that a problem that has sequential logarithmic complexity is in NC, even if no further speedup can be achieved by parallelism; our definition of GNC disallows such anomalies. Consider the problem of searching in a sorted list. The parallel complexity is $T_p(N) = \Theta(\log_{p+1}N)$ [k2,S1]; so, $P$ processors achieve a speedup of $\Theta(\log(P+1))$. Thus, a number of processors that is polynomial in the sequential time $t = \Theta(2^N)$ cannot achieve running time polylogarithmic in $t$. Similar trade-offs have been exhibited for other search problems [KW]. Furthermore (unlike the class NC), the class GNC is not restricted to problems in P. For example, a problem that requires exponential serial time $t = \Theta(2^N)$ is in GNC if it can be solved in parallel in polynomial time $N^{O(1)}$ using exponentially many processors ($P = 2^N$).

4. OTHER PARALLEL COMPUTATIONAL MODELS

We examine some of the main models of parallel computation in §4.1. These models are all truly distinct in their computing abilities – §4.2 lists some of the separating theorems. While such distinctions may be of theoretical interest, they do not necessarily coincide with real technological alternatives. Luckily, it turns out that many of these distinctions can be ignored in the research of parallel efficient algorithms: §4.3 shows that these models do not differ by much in computing power, so that the class PE is invariant across the models. §4.4 shows that despite the fact that the models differ in computing power, the class PE is invariant across the more powerful ones.

4.1. The Models

**PRAM Models:** The Synchronous PRAM model admits many variants, differing in the power of their memory access mechanism. The EREW (Exclusive Read Exclusive Write) model, which is the weakest variant, does not allow concurrent accesses to the same location. The CREW (Concurrent Read Exclusive Write) model allows concurrent reads from the same location, whereas exclusive access is required for writes. The CROW (Concurrent Read Owner Write) model sets a fixed partition of memory; a processor can write only on memory cells in the partition it owns. The CRCW (Concurrent Read Concurrent Write) model not only allows concurrent reads from the same location, but multiple processors may write to the same location at the same time. Various rules are used to determine the outcome of concurrent writes (Common, Arbitrary, Priority, etc.)

Even stronger PRAM models are obtained by the addition of Read-Modify-Write operations, as described in §2.3.

**Bibliographical Note**

The current terminology for PRAM models (CRCW, CREW, and EREW) was coined in [S1]; these models, however, were used long before. The CREW model in implicit in the earliest research on parallel numerical algorithms (see, e.g. [BM, ch. 6], and references
therein). The PRAM of Fortune and Wyllie [FW] is CREW. The EREW model (called PRAC) is used by Lev, Pippenger, and Valiant [LP]. The COMMON CRCW model is used by Kucera [K]; the priority CRCW model is used by Goldschlager [Go] (his SIMD-DAG is a SIMD CRCW Priority PRAM). The strong PRAM model (with Fetch&Op operations) and the arbitrary model originate from the work on the NYU Ultracomputer [GG,kl,Ru]. The CROW model was studied by Dymond and Ruzzo [DR].

Memory Partition: A Direct Connection Machine (DCM) consists of autonomous unit-cost RAM's, each with its own local memory, that communicate by message passing – there is no shared memory. In addition to the usual (local) operations, processors can send and receive messages. Each processor can buffer at most one incoming message. The instruction \( \text{SEND}(v,i) \) stores in the input buffer of processor \( i \) the value \( v \). The instruction \( \text{RECEIVE}(v) \) sets \( v \) to the value in the input buffer and clears the buffer. A message may be sent to a processor only if its input buffer is clear, and only one message at a time may be sent to a processor. (The model of a PRAM with an equal number of processors and memory modules is studied in [tI], where it is called Fully Connected Direct Connection Machine, and in [MV], where it is called Module Parallel Computer (MPC).)

Communication Latency and Granularity: In many parallel systems there is a significant overhead for establishing a communication between two processors; once established, large amounts of information can be transferred at low cost. This is modeled by assuming that information can be transferred across processors only in large blocks. A Direct Connection Machine (DCM) has granularity \( g \) if each message consists of \( g \) words and takes \( g \) cycles to transfer. The usual DCM model corresponds to a model with granularity one. Other reasonable assumptions are logarithmic granularity \( (g = \Theta(\log P)) \), and polynomial granularity \( (g = \Theta(P^a)) \). Granularity constraints can be introduced into shared memory models in a similar manner: Shared memory is accessed by copying blocks of size \( g \) to or from local memory; local memory is accessed on a word by word basis.

Synchronism: Real MIMD parallel systems are not synchronous, in general. Each processor independently executes its own instruction stream, at its own pace. The semantics of asynchronous execution of parallel code is easy to define, using interleaving semantics: The outcome of a computation is that would obtain in a sequential execution of a sequence of instructions obtained by arbitrarily interleaving the instruction streams of each processor. Processors can progress at arbitrary relative speeds. A time measure is obtained by assuming that each processor executes at least one instruction per time cycle; it may execute more. The precise definitions for the DCM model are given in [K2]; they are adapted from [FL]. In particular, we show that two processors can synchronize in constant time, using busy waiting; this can be generalized to a barrier synchronization routine that synchronizes \( P \) processors in time \( O(\log P) \). It is also possible to define timing for asynchronous PRAM's, with various assumptions on memory access mechanisms [S2]. The PRAM models can also synchronize in \( O(\log P) \) time.

Sparse Communication Networks: The DCM model corresponds to the case of complete communication graphs. In a sparse network the degree of the communication graph is a slowly increasing function of the number of processors \( P \) (e.g. \( d = 2\log P \), for a Hypercube), or is bounded (e.g. \( d = 4 \), for a Butterfly).

4.2. Relations Between Models

The various models presented differ in their computing power. Logarithmic separation theorems are known for most pairs of models; these theorems are of the form:
There is a problem that can be solved in constant time with \( P \) processors in the strong model but requires \( \Omega(\log P) \) time in the weak model.

Examples of pairs of models that can be thus separated are

- CRCW PRAM vs CREW PRAM [CD],
- CREW PRAM vs EREW PRAM [S1],
- EREW PRAM vs DCM [K2],
- Synchronous DCM vs Asynchronous DCM [S2],
- DCM with granularity 1 vs DCM with granularity \( \log P \) [K2],
- DCM vs bounded degree network [K2].

The last results are optimal: the weak model can simulate the stronger model with the same number of processors and a logarithmic overhead. For example:

**Theorem 4.1** [Vi,K2]. A \( P \) processor EREW PRAM can simulate \( T \) steps of a \( P \) processor strong PRAM in time \( O(T \log P) \).

**Theorem 4.2** [A,KU,Ra]. \( T \) steps of a strong PRAM with \( P \) processors and \( P^{O(1)} \) memory can be simulated on a \( P \) processor Butterfly network deterministically in time \( O(T (\log P)^2) \) and probabilistically in time \( O(T \log P) \).

As previously stated we can assume that the simulating machine is asynchronous; if it has \( \log P \) granularity then the simulation overhead is at most worse by another logarithmic factor. This implies the following:

**Corollary 4.3.** (i) Given a parallel algorithm in the strong PRAM model that is \((\text{polynomial}, \text{polylog})\)-efficient relative to a sequential algorithm, it can be simulated by a \((\text{polynomial}, \text{polylog})\)-efficient algorithm in any of the other models (PRAM's, DCM's and Butterfly networks, synchronous or asynchronous, and with granularity \( g = \log^{O(1)} p \)).

(ii) The class \( \text{PE}^* \) is invariant under the various models defined above.

The logarithmic gaps between the various models indicate that \( P \) processors of a weaker model cannot efficiently simulate \( P \) processors of a stronger model. This would seem to entail that the more stringent requirement of strong efficiency cannot be met when moving from one model to another. Surprisingly, this is not the case.

**Definition.** A parallel computing model \( M_2 \) efficiently simulates model \( M_1 \) if there is a constant \( 0<\delta<1 \) such that \( T \) steps of a computation with \( Q \) processors in model \( M_1 \) can be simulated in \( O(TQ/P) \) steps with \( P = Q^\delta \) processors in model \( M_2 \). Two models are equivalent if each efficiently simulates the other.

**Theorem 4.4.** Assume that \( M_2 \) efficiently simulates \( M_1 \). Then a parallel efficient algorithm on model \( M_1 \) can be simulated by a parallel efficient algorithm on model \( M_2 \).

**Corollary 4.5.** The class \( \text{PE} \) is invariant across equivalent parallel models.

We now proceed to show that most previously defined parallel models are equivalent (and, therefore, define the same class of efficient algorithms). To do so, it is sufficient to show that the strong PRAM model can be simulated efficiently by the weaker models. We assume the simulated machine \( M_s \) is a strong PRAM with \( Q \) processors whereas the simulating machine \( M_t \) is a synchronous DCM with \( P \) processors and granularity one; later we extend the result to the other models.

Each of the \( P \) processors of \( M_t \) is allocated the job of simulating \( Q/P \) processors of \( M_s \). The memory of \( M_s \) is also distributed across the simulating processors, according to some hashing function \( h; h(\text{addr}) \) is the id of the processor storing location \( \text{addr} \). The simulation proceeds by rounds; each round simulates one step of \( M_s \).
The only nontrivial part of a round is to satisfy memory requests. The algorithm is described below. Arrays are assumed to be stored with blocks of consecutive locations at each processor; an array of length \( N \) is stored with the first \( N/P \) locations at first processor, the next \( N/P \) locations at second processor, etc.

1. **Issue requests**: Each processor creates for each memory request a tuple \(<id, addr, A>\).

2. **Sort requests**: Sort requests by keys \(<addr, id>\). The sorted tuples are stored in an array of length \( Q \).

3. **Route requests**: For each first tuple in a group of tuples with same address, route a message to the processor containing the variable accessed (processor \( h(addr) \)); the message is tagged by the "return address" of the processor sending that request.

4. **Read Memory**: Read the value in memory for each message received at the previous step.

5. **Route Read Answers**: Route the values read back to the processors that initiated the read messages, according to the return address of each message.

6. **Simulate Accesses**: Combine requests within each group, and compute the value returned by each access, and the final value in memory (using parallel prefix in groups) – see [K2].

7. **Route Write Answers**: Send the final value produced by combining requests within each group to the processor storing that variable.

8. **Write Memory**: Write back in memory the values sent at the previous step.

9. **Route Replies**: Send the outcome of each request to the processor that originated the request.

Assume, for the time being, that memory accesses are well distributed, so that each processor receives at most \( 2Q/P \) memory access requests. We also assume that the memory size of the simulated machine is \( Q^{O(1)} \). Steps (1,4,6) and (8) are executed locally at each processor, in time \( O(Q/P) \); the remaining steps can be implemented using the following three routines in time \( O(Q/P) \), if \( P \leq Q^{1/3} \).

**Parallel Prefix**: Input: an array \( x_1, \ldots, x_N \) of variables; and an associative binary operation * that can be computed in constant time. Output: The array \( y_1, \ldots, y_N \), where \( y_i = x_1 * \cdots * x_i \). Parallel prefix on a DCM takes time \( O(N^2 + \log P) \).

**Balanced Routing**: Input: an array \( x_1, \ldots, x_N \) of records; a destination vector \( \sigma(1), \ldots, \sigma(N) \); \( \sigma(i) \in \{0,1, \ldots, P\} \). For each \( i, 1 \leq i \leq P \), \( \sigma^{-1}(i) = O(N/P) \). Output: record \( x_i \) is moved to processor \( \sigma(i) \), if \( \sigma(i) \neq 0 \). Balanced routing can be performed deterministically on a DCM in time \( O(N/P + P^2) \) [KM]. Using a slightly stronger model where conflicting Send operations are allowed and the Send operations fail when such conflict occurs, the running time can be improved to \( O(N/P + \log P) \), using a probabilistic algorithm [AM].

**Radix Sort**: Input: \( x_1, \ldots, x_N \) in the range \( 1, \ldots, R \). Output: A stable sort of the inputs. Radix sort can be performed deterministically on a DCM in time \( O((N/P + P^2)\log R / \log(N/P)) \) and probabilistically in the stronger conflict model in time \( O((N/P + \log P)\log R / \log(N/P)) \) [K2].

We still need to ensure that memory accesses are well distributed across the simulating processors. To do so, we distribute memory variables across the processors using a Universal Hashing Function [CW]. This is similar to the technique used by Karlin and Upfal [KU] to simulate PRAM's by Ultracomputers. However, in order to get an
efficient simulation, a hashing function is needed that can be computed in constant time, so a different analysis is required.

Let \( p \) be a prime number larger than the highest memory address in the simulated machine; \( p = Q^{O(1)} \). The memory variables are distributed across the \( P \) simulating processors according to a hash function randomly chosen from the set

\[
H_d = \{ h : h(x) = ((\sum_{i=0}^{d-1} a_i x^i) \mod p) \mod P, a_i \in \{0, \ldots, p-1\} \}.
\]

A family of discrete random variables \( X_1, X_2, \ldots \) is \( d \)-independent if for each choice of \( d \) variables and \( d \) values

\[
\text{Prob}(X_{i_1} = a_1, \ldots, X_{i_d} = a_d) = \text{Prob}(X_{i_1} = a_1) \times \cdots \times \text{Prob}(X_{i_d} = a_d).
\]

We have the following result:

**Lemma 4.6** [CW]. The set of random variables \( \{h(i) : 0 \leq i < p\} \) is \( d \)-independent.

We use a generalization of Chebychev’s inequality, which is derived from the following theorem.

**Theorem 4.7.** Let \( X \) be a nonnegative random variable such that \( E(X) = 0 \) and \( E(X^d) = c < \infty \). Then there exists a constant \( \alpha \) such that the following holds. If \( X_1, \ldots, X_n \) are \( d \)-independent variables equidistributed as \( X \) then

\[
E((\sum_{i=1}^n X_i)^d) \leq \alpha cn^{d/2}.
\]

**Corollary 4.8.** Let \( X_1, \ldots, X_n \) be nonnegative, \( d \)-independent, equidistributed random variables, such that \( E(X^d) = c < \infty \). Let \( \mu = E(X) \). Then there exists a constant \( \beta \) (that depends on \( c \) and \( d \), but not on \( n \)) such that

\[
\text{Prob}\left(\sum_{i=1}^n (X_i - \mu) > \epsilon\right) \leq \alpha cn^{d/2} \epsilon^{-d}.
\]

**Proof:** We have

\[
E(X - \mu) = 0 \quad \text{and} \quad E((X - \mu)^d) \leq E(X^d) = c.
\]

Thus,

\[
\text{Prob}\left(\sum(X_i - \mu) > \epsilon\right) = \text{Prob}\left(\left(\sum(X_i - \mu)\right)^d > \epsilon^d\right) \\
\leq E\left(\left(\sum(X_i - \mu)\right)^d\right)/\epsilon^d \leq \alpha cn^{d/2} \epsilon^{-d}.
\]

We now apply these results to the memory distribution problem.

**Theorem 4.9.** Any \( T \) steps of a PRAM with \( Q \) processors and \( Q^k \) memory locations can be simulated by a DCM with \( P = Q^{1/3} \) processors in time \( O(kTQ/P) \) with probability tending to 1 as \( T \) and/or \( Q \) goes to infinity.

**Proof:** We shall use a universal hash functions from \( H_d \), for some constant \( d \) to be chosen later. Note that the hash value of an address is computed in time \( O(d) \). Whenever more than \( 2Q/P \) memory requests are generated in one step for the same simulating processor we rehash the entire memory, using a new randomly chosen function from \( H_d \). This rehashing takes time \( O(Q^k d/P) \). Otherwise, a step of the simulated machine is executed in time \( O(Qd/P) \). We have to show that the probability of rehashing is significantly smaller than \( (Qd/P)/(Q^k d/P) = 1/Q^{k-1} \). Let \( X_i^j \) be the
random variable that equals one if the $i$-th memory access hashes to processor \( j \), zero otherwise. The random variables \( X_1, \ldots , X_Q \) are \( d \)-independent, according to Lemma 4.6; \( E(X_i) = E((X_i)^d) = 1/P \). The number of accesses destined to processor \( j \) equals \( \sum_{i=1}^Q X_i^j \). According to Corollary 4.8 the probability of more than \( 2Q/P \) accesses at processor \( j \) is bounded by \( \alpha Q^{d/2} / (P(Q/P)^d) \). The probability that more than \( 2Q/P \) accesses occur at any of the \( P \) simulating processors is bounded by \( \alpha Q^{d/2} / (Q/P)^d = \alpha P^d Q^{-d/2} = \alpha Q^{-d/6} \). We choose \( d > 6k \).

The last result can be slightly improved, using the balanced routing algorithm of Anderson and Miller (and their more powerful DCM model): it is sufficient to take \( P = Q^{k-\epsilon} \), for some fixed \( \epsilon > 0 \).

A similar result holds for weaker models:

**Theorem 4.10.** An asynchronous DCM with polynomial granularity can simulate efficiently a synchronous DCM with granularity one.

**Proof Outline:** It is easy to modify the balanced routing algorithm of Kruskal, Madej, and Rudolph [KM] so that it works on an asynchronous DCM with granularity \( g \) in time \( O(N/P + \sqrt{NPg} + Pg) \). If \( g = O(P^k) \) then this running time is \( O(N/P + P^{k+3}) \). It follows that \( P = Q^{1/(k+4)} \) processors of an asynchronous DCM with granularity \( O(P^k) \) can simulate a communication step of \( Q \) processors of a synchronous DCM with granularity one in time \( O(Q/P) \). We use the same simulation as before, with the modified balanced routing and parallel prefix routines.

**Corollary 4.11.** The class \( PE \) is invariant under the PRAM and DCM models for probabilistic algorithms.

**Corollary 4.12.** The class \( PE \) is invariant under the PRAM models (for deterministic algorithms).

**Proof:** Use the same simulation as for the DCM model, except the routing is not needed for PRAMs. This becomes a deterministic algorithm.

**5. SUMMARY AND OPEN QUESTIONS**

**5.1. Parallel Efficient Algorithms and Complexity Classes**

Efficient parallel algorithms have been developed for many basic problems. Nevertheless, the field still offers many open problems: No parallel efficient algorithms are known for problems on sparse graphs, where the number of edges is linear in the number of nodes. This includes shortest path problems, connectivity, minimal spanning trees, transitive closure and cycle detection in directed graphs, and topological sort of directed acyclic graphs. Also, problems that are known to be \( \text{P-complete} \) have not received much attention. Parallel efficient algorithms for linear programming, network flow, and related problems have an obvious practical importance.

Table 1 classifies a few well-known problems according to their membership in the classes we defined. The complexity of searching, merging, and sorting is defined using comparison based models, and the complexity of arithmetic problems is defined using arithmetic circuit models (see [K2]). We assume that \( \text{P-complete problems are not in NC} \). Note that \( \text{PE} \subseteq \text{PE}^* \); hence membership in \( \text{PE} \) implies membership in \( \text{PE}^* \).

The example of lexicographically first depth first search [Re] shows that \( \text{PE} \) is not contained in \( \text{NC} \) (or GNC); there are problems where efficient polynomial speedups are possible, but where polylogarithmic running time is (apparently) not possible, when a polynomial number of processors is used. (This observation was made by Vitter and Simons [VS].)
We conjecture the reverse is also true: GNC is not contained in PE. I.e., there are problems where a quasi-exponential reduction in running time can be achieved using polynomially many processors, but where no polynomial reduction in running time can be achieved with a number of processors proportional to the speedup.

More generally, one would like examples of natural problems that are not in PE. Such problems exhibit nonlinear trade-offs between the number of processors applied to solve them, and the running time achieved. Thus, an interesting question is to exhibit such trade-offs, even for restricted models. Searching is such an example: we have $T_P(N) = \Omega(T(N)/\log(P+1))$ [k2,S1,KW]. This is an exponential trade-off and, hence, precludes the existence of $(\text{polylog}, \text{polynomial})$-efficient algorithms. Can one find examples of problems that exhibit nonlinear, polynomial trade-offs?

### 5.2. Efficient Simulations and Separation Theorems

An important issue is that of the relative power of the various parallel computation models. The separation theorems given in §4 are weak; e.g. is still possible that any problem that can be solved in time $T$ with $P$ processors in a CRCW PRAM, can be solved in time $O(T + \log P)$ with $P$ processors on an CREW PRAM. One would like, for each pair of models listed there, a strong separation theorem of the form:

For any $T$ and $P$, there is a problem that can be solved in the strong model with $P$ processors in time $T$, but requires time $\Omega(T \log P)$ to solve in the weak model with $P$ processors.
Such separation is only known to exist between PRAM's and constant degree networks [KU, K2]. A strong separation theorem would also imply that an efficient simulation of \( P \) processors of the strong model can use at most \( P / \log P \) processors on the weak model. Note that the simulations presented in this paper require a reduction from \( P \) to \( P^{1/3} \) in the number of processors used. It is an open problem to provide tight bounds on the reduction in the number of processors that is required to obtain an efficient simulation.

The (efficient) simulation of a CRCW PRAM by an EREW PRAM can be done deterministically; on the other hand, the simulation of a PRAM by a DCM makes heavy use of probabilistic techniques for memory distribution. We would like to know whether randomization is really needed.

The difference between the various models is, essentially, in terms of their ability to sustain communication and synchronization across processors. Most "natural" problems exhibit some "locality", so that when the ratio between the problem size and the number of processors increases, the ratio of communication to local processing decreases; for sufficiently large problems local computation dominates communication.

The notion of locality can be made precise as follows: Consider a PRAM model where each processor has its own local memory; processors execute either computation steps or communication steps to the shared memory. Let \( T(N) \) and \( C(N) \) be, respectively, the number of computation steps and communication steps executed by a size-dependent algorithm, with \( P(N) \) processors. We say that a size-dependent algorithm has locality \( f \) if \( T(N)/C(N) \geq f(P(N)) \). Tight bounds on the communication complexity of various problems are proven in [A2]. In particular, it is shown that FFT and sorting have logarithmic locality. A simulation overhead is incurred only for communication steps. It follows, for example, that a PRAM algorithm with logarithmic locality can be simulated efficiently on a Butterfly network. Thus, FFT and sorting can be computed efficiently on a Butterfly network.

The locality of a computation depends on the algorithm used and on the memory management (the allocation of physical locations to variables). The latter may be specified by the programmer, may be deduced by the compiler, or may be dynamically updated online. It is an open question to understand how much locality can be extracted by each of these three methods. In particular, can one devise (online) simulation algorithms that automatically take advantage of the locality that would obtain by a correct allocation of variables to local memories?

Acknowledgments

This work benefited from useful criticisms from Ron Fagin and Moshe Vardi, and help from Michael Benor, Bill Gasarch, Prabhakar Raghavan, and Eli Upfal.

References


