Theoretical Computer Science 71 (1990) 95-132
North-Holland

A COMPLEXITY THEORY OF EFFICIENT PARALLEL ALGORITHMS *

Clyde P. KRUSKAL

Computer Science Department, Institute for Advanced Computer Studies, University of Maryland College Park, MD 20742, USA

Larry RUDOLPH

Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel

Marc SNIR

IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA

Abstract. This paper outlines a theory of parallel algorithms that emphasizes two crucial aspects of parallel computation: *speedup* the improvement in running time due to parallelism, and *efficiency*, the ratio of work done by a parallel algorithm to the work done by a sequential algorithm. We define six classes of algorithms in these terms; of particular interest is the class, EP, of algorithms that achieve a polynomial speedup with constant efficiency. The relations between these classes are examined. We investigate the robustness of these classes across various models of parallel computation. To do so, we examine simulations across models where the simulating machine may be smaller than the simulated machine. These simulations are analyzed with respect to their efficiency and to the reduction in the number of processors. We show that a large number of parallel computation models are related via efficient simulations, if a polynomial reduction of the number of processors is allowed. This implies that the class EP is invariant across all these models. 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. To be useful, such a theory must address two major concerns in parallel computation, namely speedup and efficiency. It should classify algorithms and problems into a few, meaningful classes that are, to the largest extent possible, model independent. This paper outlines an approach to the analysis of parallel algorithms that we feel answers these concerns without sacrificing too much generality or abstractness.

We propose a classification of parallel algorithms in terms of parallel running time and inefficiency, which is the extra amount of work done by a parallel algorithm as compared to a sequential algorithm. Both running time and inefficiency are measured as a function of the sequential running time, which is used as a yardstick

* A preliminary version of this paper was presented at 15th International Colloquium on Automata, Languages and Programming, Tampere, Finland, July 1988.

to evaluate parallel algorithms. One desires a small running time with a small inefficiency. To be in the class NC, an algorithm must have (at most) polylogarithmic running time with (at most) polynomial inefficiency. These requirements seem to overemphasize the importance of speedup, while underemphasizing the importance of efficiency. It will be argued that a more modest "polynomial" reduction of running time from sequential time $t(n)$ to parallel time $O(t(n)^{\varepsilon})$, for some constant $\varepsilon < 1$, is a more reasonable and practical goal. The combinations of these two requirements on running time and of three natural bounds on inefficiency define six classes of algorithms.

(1) ENC (Efficient, NC fast): the class of algorithms that achieve polylogarithmic running time with constant inefficiency;

(2) ANC (Almost efficient, NC fast): the class of algorithms that achieve polylogarithmic running time with polylogarithmic inefficiency;

(3) SNC (Semi-efficient, NC fast): the class of algorithms that achieve polylogarithmic running time with polynomial inefficiency;

(4) EP (Efficient, Polynomially fast; or Efficient Parallel): the class of algorithms that achieve a "polynomial" reduction in running time with constant inefficiency;

(5) AP (Almost efficient, Polynomially fast; or Almost efficient Parallel): the class of algorithms that achieve a "polynomial" reduction in running time with a polylogarithmic inefficiency; and

(6) SP (Semi-efficient, Polynomially fast; or Semi-efficient Parallel): the class of algorithms that achieve a polynomial reduction in running time with a polynomial inefficiency.

These classes naturally categorize almost all extant parallel algorithms. This classification extends and refines classes defined in [77].

The most practically interesting of these classes is EP. It seems to best capture the type of algorithm that is actually used on real parallel computers: p processors yield a p-fold reduction in running time, for a problem with total computational requirements of $p^{O(1)}$. Associated with each class of algorithms is a class of problems. For example, EP is (essentially) the class of problems that have EP algorithms. EP, as a class of problems, seems to best capture the type of problem that is solved well by parallel computers.

One obstacle encountered in the investigation of parallel algorithms is the variety of existing parallel computer architectures and the even greater variety of proposed parallel computation models. Some of the distinctions among parallel computation models reflect real technological constraints such as memory granularity and contention, asynchronism, large communication latency, and restricted communication geometry. We examine some of these models. We consider simulations across models where the number of simulating processors may be smaller than the number of simulated processors. Such simulations are analyzed in terms of the reduction in the number of processors and their inefficiency (the ratio between the work of the simulated machine and the work of the simulating machine). We can trade off these two parameters, obtaining more efficient simulations at the cost of a larger reduction in parallelism. It is known that a PRAM can be simulated by a complete network (or even a fixed degree network) with the same number of processors, and a polylogarithmic inefficiency. This implies that the classes of algorithms that tolerate a polylogarithmic inefficiency are invariant across these models. More surprisingly, we show that a $p^{1-\varepsilon}$ processor complete network ($\varepsilon > 0$) can, with constant inefficiency, probabilistically simulate a p processor PRAM. This result cannot be extended to constant degree networks. It does, however, extend to weaker complete network models, taking into account both high (polynomial) start-up time and asynchronism. This shows that the class EP is invariant across all these models. Thus, it is possible to develop a unified theory of EP (efficient parallel) algorithms that ignores distinctions among these models.

Section 2 very briefly reviews sequential complexity analysis and parallel computer models, and then critiques the parallel complexity class NC. Section 3 defines classes of algorithms in terms of parallel running time and inefficiency. Problems are classified according to the types of parallel algorithms that can be used to solve them, and the relations among these classes are explored. New problems, motivated by this approach, are proposed. The section concludes by comparing and contrasting our definitions of parallel classes with some other suggested definitions. Section 4 describes the relationship between parallel computer models and our parallel classes. We first discuss parallel computer models, and then present separation theorems that show that these models differ in their computing power. Simulation results are presented that show that, despite these differences, the classes of problems are invariant across large classes of models. The section concludes by summarizing the section and listing new problems on the relative power of various parallel computing models. Section 5 summarizes the paper and lists more open problems.

2. Review

2.1. Sequential complexity

The theory of sequential computer algorithms is both of significant intrinsic interest and useful as a guide to the implementation of algorithms on real sequential computers. It will be useful to consider the reasons for its success before discussing parallel algorithms.

The study of algorithms proceeds in the following three conceptual stages.

(1) Establish a formal computing model. A widely used abstract computation model is the unit-cost RAM model [5]. One assumes a standard set of operations, each taking unit time, and a random access memory of unlimited size with unit access time.

(2) Study the complexity of various algorithms in this model; i.e., analyze the amount of computing resources needed to solve a problem as a function of its size—the most important resource being *time*.

(3) Classify problems as *tractable*, i.e., those that can be solved within reasonable time, or *intractable*. The class P of problems solvable in polynomial time is taken to be the formal definition of tractability.

A computing model (such as the RAM model) is chosen so as to be fairly abstract, hence simple and general. On the other hand, it should be sufficiently close to real machines so that complexity results for algorithms on the abstract model are indicative of the resources used by programs running on real machines. The model captures a machine organization paradigm; the RAM model is an abstraction of the Von Neumann computer architecture. One should remember that computer architectures are themselves abstractions that are separated from the underlying physical reality of circuits and devices by many layers of translation and emulation. The model that most appropriately describes the programmer model of a machine bears little relation to the physical realities of circuit technology.

While the RAM is the predominant model used to analyze sequential algorithms, it is by no means the unique model. The RAM model provides a good approximation of real machines for a certain family of problems within a given range (CPU bound problems that use a moderate amount of memory and a reasonable word size). Outside this range, other models may be more appropriate (e.g., to describe I/O bound computations). The definition of the class P of tractable problems is *robust*, and is preserved across all "reasonable" models of sequential computations. (Circularly, once a concept like P is established, it serves to define what is a reasonable model.)

2.2. Parallel computers and computational models

A "sequential" computer is highly parallel when examined at the device level; many circuits are concurrently active. This concurrency is hidden from the user, who is presented with an architecture model where instructions are executed sequentially. The fiction of sequential execution provides the user with a computation model that is simple to comprehend; it simplifies control of execution. On the other hand, the obvious way of increasing computer performance with a given hardware technology is to increase the amount of concurrency. In order to increase concurrency while keeping the fiction of sequential execution, one needs increasingly complex control mechanisms to enforce data and control dependencies, and increasingly complex compiler analysis to extract concurrency from serial code. Increasing the amount of concurrency in a sequential machine offers diminishing returns and it seems we are reaching the limits of what can be done at reasonable cost within the Von Neumann paradigm. On the other hand, the rapid miniaturization of circuits and the availability of cheap, mass-produced computing elements offer the opportunity for massive parallelism. Thus, increasingly one sees large-scale parallel computers, i.e., computers where parallelism is evident at the user interface. (By large-scale we mean machines with hundreds or thousands of processing elements, each as complex as a microprocessor.) Efficient use of such machines requires the development of a complexity theory of parallel computations.

The first step is to choose a formal parallel computing model. Our choice is complicated by the lack of consensus on parallel computer architectures. Existing parallel computers differ widely in many essential aspects, such as communication and control mechanisms. As a result, many different parallel computation models have evolved, each reflecting different types of parallel computer architectures. We are going to use the PRAM model of a shared memory machine as our basic model for several reasons: (1) It is a natural generalization of the RAM model. (2) It is a *reasonable* model: the number of operations done by p processors per cycle is at most p. (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). (4) It is a *practical* model: large shared memory multiprocessors are actually being built. (5) The model is simple and convenient.

Other parallel computer models, and their relations to the PRAM model, are discussed in Section 4. Parallel complexity and derived concepts can be defined using any reasonable parallel computer model. The choice of model makes little difference to our theory; the classes we define are invariant over large families of parallel computing models.

Formally, a PRAM (Parallel Random Access Machine) or *paracomputer* [30, 68] consists of a set of autonomous processors, all sharing a common memory. Each processor is a unit-cost RAM, as defined in [5], with a few minor modifications. Each processor has its own instruction counter and its own local set of registers; one of these registers stores the processor number. All processors execute the same program; they can, however, branch on processor number. The processors execute their instruction streams in lock-step, one instruction per time step. Instructions are executed on operands in local registers. In addition, a processor may load a variable from shared memory to a local register, or store from a local register to shared memory. Several processors may concurrently access the same shared location; the outcome is as if the accesses occurred sequentially, in the order defined by the processor ids (this is the *priority* CRCW PRAM model). The inputs are stored initially in shared memory, and the outputs are stored there at the end of the computation.

Having chosen the PRAM model, our study of the complexity of parallel algorithms can begin. We can define complexity classes in terms of the two main resources used by parallel algorithms, namely the number of processors p and the time T.

2.3. *Critique of NC*

Analogous to the class P for serial algorithms, one seeks a complexity class that captures the notion of problems that have "good" parallel algorithms. This class has usually been identified with NC, the class of problems that can be solved in polylogarithmic time using polynomially many processors. We have $NC \subseteq P$, so that a problem has an efficient parallel solution only if it is tractable. It is widely conjectured that this inclusion is proper. Hence, the class of problems that are P-complete (under log-space reductions) is conjectured to be disjoint from NC. A proof that a problem is P-complete is taken as evidence that it is "inherently serial" and cannot be solved fast in parallel.

Undoubtedly, NC is a useful concept: it has led to the development of a deep and rich theory (see [22, 12, 40] for surveys of results). However, it is not clear that it captures the informal notion "amenable to a parallel solution". Vitter and Simons [77] have shown that P-complete problems (i.e., problems presumed not be in NC) may be solved by efficient parallel algorithms, using a reasonable definition of efficient parallel algorithms. On the other hand, a problem like search (on an ordered list), which runs in logarithmic serial time, is in NC, irrespective of the existence of efficient parallel algorithms. In fact, searching does not admit efficient parallel algorithms [48, 70].

Efficiency is a prime consideration in the design of parallel algorithms: one desires to solve a problem roughly p times faster when using p processors. This consideration is missing from the definition of NC. An "NC algorithm" can be "polynomially wasteful": it can perform polynomially more operations than a serial algorithm that solves the same problem.

The domain of application of such NC algorithms seems to be mostly disjoint from the domain of application of large-scale parallel computers. In practice, one uses a moderately sized machine (tens, hundreds, or thousands of processors) to efficiently solve a large problem (thousands or millions of input variables). NC theory is concerned with the situation where one would use a huge machine (billions of processors) to quickly solve a moderately sized problem (hundreds or thousands of input variables). Furthermore, some problems that are not in NC have parallel sublinear algorithms. For algorithms with sublinear, slow growing running time, the asymptotic behavior becomes relevant only for impractically large input sizes (and impractically large processor counts). For example, an algorithm with running time $\sqrt{n}$ is superior to an algorithm with running time $\log^3 n$ up to input size (in excess of) one half billion.

Finally, a computer system is likely to have serial bottlenecks, such as I/O. It does not help to solve a problem in polylogarithmic time, if the time required to input and output data is linear, or worse.

3. Parallel complexity

3.1. *Definitions*

While the time for an algorithm to solve a problem depends directly on the input, usually the running time is measured with respect to problem size n. For a sequential algorithm A, we write $t^A(n)$ for the (worst-case) running time on a problem of size n, but shorten it to $t(n)$ when the algorithm is clear from context. Parallel algorithms have one additional free parameter: the number p of processors. For parallel algorithms there is at least one resource that must be considered in addition to time, namely the number p of processors used (which we assume to be a simple, easily computable function of n). We can take both time and the number of processors to be functions of the probiem size; we call such an algorithm *size-dependent* [36]. We denote then by $P^B(n)$ the number of processors and by $T^B(n)$ the time used by algorithm B on inputs of size n. Alternatively, we may assume that p (the number of processors) is a free parameter, and take time to be a function of n and p. We call such an algorithm *size-independent* [36], and denote $T_p^B(n)$ the running time of algorithm B with p processors on inputs of size n. The superscript B is omitted when the algorithm is clear from context. The former approach (size-dependent algorithms) is more convenient to use in formal definitions; the latter is more natural. In any case, there is an easy translation from one type of algorithm to the other.

Definition. A parallel model is *self-simulating* if a p processor version can simulate one step of a q processor version in time $O(q/p)$ for $p < q$ and in time $O(1)$ for $p \geq q$. (This simplifies to time $O(\lceil q/p \rceil)$.)

All of the PRAM models are self-simulating, as are meshes, trees, and hypercubes. Butterfly networks (with wrap-around) are not self-simulating since a large machine cannot (necessarily) simulate a smaller version of itself in constant time. However, it is fairly easy to modify the butterfly topology so as to obtain a family of constant degree self-simulating networks that contains the butterfly networks. Most models fulfill a stronger requirement: q processors can simulate with constant overhead $\Omega(q/p)$ independent p processor machines, for $q > p$. This holds true for PRAM models, meshes, and hypercubes, but not for butterfly or related networks. An argument similar to that used in [58] shows that a constant degree network that permutes in logarithmic time cannot fulfill the stronger requirement.

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

This assumption implies that a size dependent algorithm with time and processor bounds $T(n)$ and $P(n)$, respectively, can be simulated by a size independent algorithm with running time

$$T_p(n) = \Theta(\lceil P(n)/p \rceil \cdot T(n)) = \Theta(\max(T(n)P(n)/p, T(n))). \tag{3.1}$$

The running time of this simulating algorithm is of the form

$$T_p(n) = \Theta(\max(f(n)/p, g(n))) = \Theta(f(n)/p + g(n)). \tag{3.2}$$

We shall generally assume that the running time of a parallel algorithm is given in such form, with $f(n) > g(n)$. The function $f(n)$ is the work done by the algorithm; it is equal (up to a constant factor) to the total number of steps executed by all processors, provided that $p = O(f(n)/g(n))$. The function $g(n)$ is the best running time that the algorithm can achieve (up to a constant factor). To achieve this minimal running time (up to a constant factor), it suffices to use $f(n)/g(n)$ processors. The algorithm *saturates* at this level; increasing the number of processors beyond this limit will decrease the running time by at most a constant factor.

3.2. *Performance of parallel algorithms*

It is not the performance per se of a parallel algorithm that interests us here, but rather the performance relative 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.

The main goal for parallelism is reduction in running time. Thus, the main criterion for evaluating a parallel algorithm is *speedup*, i.e., the ratio $t(n)/T(n)$ of sequential running time to parallel running time. The weakest nontrivial requirement is that speedup be unbounded. For a size-dependent algorithm this is expressed as

$$\lim_{n\to\infty} \frac{T(n)}{t(n)} = 0.$$

This is an extremely weak requirement: for example, a parallel algorithm that reduces running time from $t(n) = \Theta(n)$ to $T(n) = \Theta(n/\log\log n)$ is unlikely to be worthwhile. We generally want to claim that a significant reduction in running time is achieved by parallelism; parallel running time should be a fast decreasing function of sequential running time. There are two obvious choices for such a function.

Definition. A size-dependent parallel algorithm is *polynomially fast* if $T(n) \leq t(n)^\varepsilon$, for some constant $\varepsilon < 1$.

Definition. A size-dependent algorithm is *polylogarithmically fast* (or, for short, *polylog fast*) if

$$T(n) = \log^{O(1)}(t(n)).$$

Reduction in running time has a cost. The number of processors must increase at least as fast as the speedup; generally it increases faster. The *inefficiency* of a parallel algorithm is the ratio $T(n) \cdot P(n)/t(n)$. This is the ratio between the work of the parallel algorithm (the total number of instruction cycles executed by all processors), and the work of the sequential algorithm. The *efficiency*, which is the inverse ratio $t(n)/(T(n) \cdot P(n))$, is more commonly used.

One typically desires an order P reduction in running time when using p processors.

Definition. A size-dependent algorithm has (at most) *constant inefficiency* if

$$T(n) \cdot P(n) = O(t(n)).$$

For convenience, we will say that an algorithm with constant inefficiency has (at least) *constant efficiency* or simply is *efficient.*

As we shall see later, the requirement that a parallel algorithm be efficient is model dependent. A weaker, but more robust, requirement is polylog inefficiency.

Definition. A size-dependent algorithm has *polylogarithmically bounded inefficiency* or, for short, *polylog inefficiency* if

$$T(n)\cdot P(n)=t(n)\cdot \log^{O(1)}(t(n)).$$

Finally, the weakest reasonable requirement is polynomial inefficiency.

Definition. A parallel algorithm has *polynomially bounded inefficiency* or, for short, *polynomial inefficiency* if

$$T(n)\cdot P(n)=t(n)^{O(1)}.$$

Six interesting classes of algorithms can be defined by combining the two requirements on speedup with the three constraints on inefficiency. The strictest class is obtained by insisting on polylog speedup with constant efficiency.

Definition. ENC (Efficient, NC fast) is the class of algorithms that are polylog fast with constant efficiency (relative to $t(n)$).

For example, the problem of computing the sum of n numbers stored in consecutive locations has sequential complexity $t(n)=\Theta(n)$. Summing can be computed in parallel with $T(n)=n/\log(n)=\Theta(t(n)/\log(t(n))$ processors, in time $T(n)=\Theta(\log n)=\Theta(\log(t(n)))$. This parallel summing algorithm is therefore in ENC. In the future, we will typically not bother distinguishing $\Theta(t(n))$ from $\Theta(n)$ when $t(n)$ is a linear function, or $\Theta(\log(t(n)))$ from $\Theta(\log(n))$ when $t(n)$ is a polynomial function.

Theorem 3.1. *The following assertions are equivalent*:

(1) *A problem can be solved by a (size-dependent) parallel algorithm from the class* ENC (*i.e.*, $T(n)=\log^{O(1)}(t(n))$ *and* $T(n)\cdot P(n)=O(t(n))$).

(2) *A problem can be solved by a (size-independent) parallel algorithm with running time*

$$T_p(n)=O(t(n)/p)+\log^{O(1)}(t(n)).$$

(3) *A problem can be solved by a (size-independent) parallel algorithm with running time*

$$T_p(n)=O(t(n)/p)+\log^{O(1)}(p).$$

Because of the self-simulation property of our model, it is always possible to simulate an efficient parallel algorithm by an efficient parallel algorithm that uses fewer processors. Thus, it is convenient to derive a constant efficiency algorithm that uses as many processors as possible, i.e., is as fast as possible. However, we have already argued that algorithms that use processors inefficiently are of limited practical interest. Thus, for practical purposes, it is sufficient to derive constant efficiency algorithms that have more modest speedup.

Definition. EP (Efficient, Polynomially fast; or Efficient Parallel) is the class of algorithms that are polynomially fast and have constant inefficiency (relative to $t(n)$).

This will be the main class of algorithms investigated in this paper. For example, assume that we take the Gaussian elimination algorithm (which has $t(n)=\Theta(n^3)$ running time) to be the yardstick for the sequential time of matrix inversion. Then a parallel version of Gaussian elimination that runs in time $T(n)=O(n)$ with $P(n)=O(n^2)$ processors, or even time $T(n)=O(n^2)$ with $P(n)=O(n)$ processors is in EP (with respect to our sequential standard). On the other hand, a parallel algorithm that runs in time $T(n)=\Theta(\log^2 n)$ using $\Omega(n^3)$ processors (e.g., Csanky's algorithm [24]), is not parallel efficient (since $P(n)\cdot T(n)=\Omega(n^3\log^2 n)$, which is not $O(t(n))$).

Theorem 3.2. *The following assertions are equivalent:*

(1) *A problem can be solved by a (size-dependent) algorithm from the class* EP (*i.e.*, $T(n)=O(t(n)^\varepsilon)$ *with* $\varepsilon<1$ *and* $T(n)\cdot P(n)=O(t(n))$).

(2) *A problem can be solved by a (size-independent) algorithm with running time* $T_p(n)=O(t(n)/p+t(n)^\varepsilon)$, *for some constant* $\varepsilon<1$.

(3) *A problem can be solved by a (size-independent) algorithm with running time* $T_p(n)=O(t(n)/p)+p^{O(1)}$.

Two natural classes are obtained, if polylog inefficiency is accepted.

Definition. ANC (Almost efficient, NC fast) is the class of (size-dependent) algorithms that are polylog fast with polylog inefficiency (relative to $t(n)$).

For example, the connected components of a graph with n vertices and m edges can be computed on a sequential machine in time $t(n)=\Theta(m+n)$ and can be computed on a parallel machine (CRCW PRAM) in time $T(n)=\Theta(\log n)$ with $\Theta(m+n)$ processors [69]. This algorithm is $\Theta(\log n)$ fast and it has $T(n)\cdot P(n)/t(n)=\Theta(\log n)$ inefficiency. Thus, the algorithm is in ANC.

Theorem 3.3. *The following assertions are equivalent*:

(1) *A problem can be solved by a (size-dependent) parallel algorithm from the class* ANC (*i.e.*, $T(n) = \log^{O(1)}(t(n))$ *and* $T(n) \cdot P(n) = t(n) \cdot \log^{O(1)}(t(n))$).

(2) *A problem can be solved by a (size-dependent) parallel algorithm with* $T(n) = \log^{O(1)}(t(n))$ *time and* $P(n) = O(t(n))$ *processors.*

(3) *A problem can be solved by a (size-independent) parallel algorithm with running time*

$$T_p(n) = \frac{t(n) \cdot \log^{O(1)}(t(n))}{p} + \log^{O(1)}(t(n)).$$

(4) *A problem can be solved by a (size-indepenent) parallel algorithm with running time*

$$T_p(n) = \frac{t(n) \cdot \log^{O(1)}(t(n))}{p} + \log^{O(1)}(p).$$

Definition. AP (Almost efficient, Polynomially fast; or Almost efficient Parallel) is the class of algorithms that are polynomially fast with polylog inefficiency (relative to $t(n)$).

Theorem 3.4. *The following assertions are equivalent*:

(1) *A problem can be solved by a (size-dependent) algorithm from the class* AP (*i.e.*, $T(n) = O(t(n)^\varepsilon)$, *with* $\varepsilon < 1$, *and* $P(n) \cdot T(n) = t(n) \cdot \log^{O(1)}(t(n))$).

(2) *A problem can be solved by a (size-independent) algorithm with running time*

$$T_p(n) = \frac{t(n) \log^{O(1)}(t(n))}{p} + O(t(n)^\varepsilon)$$

for some constant $\varepsilon < 1$.

(3) *A problem can be solved by a (size-independent) algorithm with running time*

$$T_p(n) = \frac{t(n) \log^{O(1)}(t(n))}{p} + p^{O(1)}.$$

On a more theoretical level, one might tolerate algorithms with polynomial inefficiency. Again, there are two natural classes.

Definition. SNC (Semi-efficient, NC fast) is the class of algorithms that are polylog fast with polynomial inefficiency (relative to $t(n)$).

This class captures the type of algorithms whose study is motivated by the theory of NC. However, this definition is meaningful even when sequential running time is not polynomial. We can speak of an SNC algorithm with respect to an exponential sequential running time $t(n) = 2^n$; such an algorithm should achieve polynomial parallel running time $T(n) = n^{O(1)}$, using exponentially many processors ($P(n) = 2^{O(n)}$).

Theorem 3.5. *The following assertions are equivalent:*

(1) *A problem can be solved by a (size-dependent) algorithm from the class* SNC (*i.e.,* $T(n) = \log^{O(1)}(t(n))$ *and* $P(n) = t(n)^{O(1)}$).

(2) *A problem can be solved by a size-independent algorithm with running time*

$$T_p(n) = \frac{t(n)^{O(1)}}{p} + \log^{O(1)} t(n).$$

(3) *A problem can be solved by a size-dependent algorithm with running time*

$$T_p(n) = t(n)^{O(1)}/p + \log^{O(1)}(p).$$

Definition. SP (Semi-efficient, Polynomially fast; or Semi-efficient Parallel) is the class of (size-dependent) algorithms that are polynomially fast with polynomial inefficiency (relative to $t(n)$).

For example, a depth-first tree of an undirected graph with n vertices and m edges can be computed on a sequential machine in time $t(n) = \Theta(m+n)$ and can be computed on a parallel machine (CRCW PRAM) in $T(n) = \Theta(\sqrt{n}\log^5 n)$ time with $\Theta(m+n)$ processors [33]. This algorithm is polynomially fast and has polynomial inefficiency; thus, it is in SP. Given the asymptotics of slowly growing functions, as discussed in the critique of NC, the class SP may well capture those problems that are amenable to fast parallel solution better than the class SNC (or the class NC).

Theorem 3.6. *The following assertions are equivalent:*

(1) *A problem can be solved by a (size-dependent) parallel algorithm from the class* SP (*i.e.,* $T(n) = O(t(n)^\varepsilon)$ *with* $\varepsilon < 1$ *and* $T(n) \cdot P(n) = t(n)^{O(1)}$).

(2) *A problem can be solved by a (size-dependent) parallel algorithm with* $T(n) = t(n)^\varepsilon$ *time with* $\varepsilon < 1$ *and* $P(n) = O(t(n)^{O(1)})$ *processors.*

(3) *A problem can be solved by a (size-independent) parallel algorithm with running time*

$$T_p(n) = t(n)^{O(1)}/p + t(n)^\varepsilon,$$

for some constant $\varepsilon < 1$.

(4) *A problem can be solved by a (size-independent) parallel algorithm with running time*

$$T_p(n) = t(n)^{O(1)}/p + p^{O(1)}.$$

We illustrate our notation with a simple parallel sorting algorithm, due to Baudet and Stevenson [11]. They show that, given a sorting network of depth $D(p)$ that sorts $2p$ keys, one can obtain a sorting algorithm that sorts $n \geq 2p$ keys in time

$$T_p(n) = \Theta\left(\frac{n}{p}\left(\log\left(\frac{n}{p}\right) + D(p)\right)\right).$$

If one uses a bitonic sorting network [10] with depth $D(p)=\Theta(\log^2 p)$, then

$$T_p(n)=\Theta\left(\frac{n}{p}\left(\log\left(\frac{n}{p}\right)+\log^2 p\right)\right)=\Theta\left(\frac{n}{p}(\log n+\log^2 p)\right).$$

The algorithm is efficient provided that $\log^2 p=O(\log n)$ or $p=n^{O(1/\sqrt{\log n})}$; it is polynomially fast provided that $p=n^{\Omega(1)}$. These two ranges are disjoint, and the algorithm cannot efficiently achieve polynomial reduction in running time; the algorithm is not in the class EP. To see the implications of this concretely, assume that one desires the parallel algorithm to perform at most twice the number of comparisons of the sequential algorithm. Only about one hundred processors can be used to sort one million keys, and only about three hundred processors (three times the previous number) can be used to sort one billion keys.

On the other hand, the bitonic sorting network does yield an algorithm in the class ANC (almost efficient, NC fast): Taking $P(n)=n$, we obtain running time $T(n)=\Theta(\log^2 n)=\Theta(\log^2(t(n)))$. The algorithm is polylog fast, and has an inefficiency of

$$T(n)\cdot P(n)/t(n)=\Theta(n\cdot\log^2 n/(n\log n))=\Theta(\log n)=\Theta(\log(t(n))).$$

One could base our algorithm on the AKS sorting network, which has depth $D(p)=\Theta(\log p)$ [7]. The running time will now be

$$T_p(n)=\Theta\left(\frac{n}{p}\left(\log\left(\frac{n}{p}\right)+\log p\right)\right)=\Theta\left(\frac{n\log n}{p}\right)$$

for $n\geq 2p$. The algorithm is efficient for $p=O(n)$ and polylog fast for $p=n/\log^{O(1)} n$. These two ranges do intersect. Thus, the AKS sorting network yields an ENC algorithm for sorting. In practice, the "bad" ANC algorithm, based on the bitonic network, will perform better than the "good" ENC algorithm, based on the AKS network, for any reasonable values of p and n. The constant hidden in the O( ) notation is very small for the former and very large for the latter.

3.3. Classes of problems

It will be convenient to classify problems according to the type of parallel algorithms that can be applied to solve them. Informally, we say that a problem belongs to the class EP if it can be solved by a parallel algorithm that is in the class EP relative to the "best" sequential algorithm. A formal definition requires more care. We need to avoid the dependency on a specific sequential algorithm in our classification of problems: not every problem has a well defined sequential complexity, not even up to a constant factor; this follows from Blum's speedup theorem [14].

Definition. A problem is in the class EP (*Efficient Parallel*) if, with respect to any sequential algorithm that solves the problem, there exists an EP algorithm that solves the same problem.

Note that we use the same name EP to define a class of algorithms (relative to a particular sequential running time) and to define a class of problems. One should be careful not to confuse these two notions.

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 $t^A(n)=\Omega(t(n))$.) A problem with sequential complexity $t(n)$ is in EP if it has a parallel algorithm with running time $O(t(n)/p)+p^{O(1)}$. For example, if a problem has linear sequential complexity, then it is in the class EP if it can be solved in parallel in time $T_p(n)=O(n/p)+p^{O(1)}$. Similar definitions can be used to define classes of problems for the other classes of algorithms (ENC, ANC, SNC, AP, and SP).

3.4. *Discussion*

The classes of problems just defined are not defined as complexity classes in the sense of Blum [14]. It is not clear to what extent they can be used to develop a structural theory (reductions, complete problems, etc.). Nevertheless, they provide a useful organization, draw distinctions that have significant practical importance, and motivate many open research problems.

3.4.1. *Classifying problems*

The foremost question is to try to classify problems according to the parallel classes we have defined.

ENC: ENC algorithms are known for many important problems. A very partial list is given below:

(1) Sorting, merging, selection [15, 13, 20, 25, 47, 48, 63, 74].

(2) Evaluating rational recurrences of degree one; in particular computing sums, partial sum computations (initial prefix computations), evaluating polynomials, or Horner expressions, evaluating continued fractions, etc. (see [54, Section 2.3]).

(3) Formula evaluation [46].

(4) FFT [62].

(5) Connected and biconnected components on dense graphs, where *dense* means that the number m of edges is significantly larger than the number n of nodes. Depending on the precise assumptions made on the computing model, one needs $m=\Omega(n\log^* n)$ [21] or $m=\Omega(n\log n)$ [81].

EP: It is fairly easy to show that one can traverse a graph with n nodes and m edges, in breadth first or depth first order, in $O(m/p+n)$ time [29] (or $O(m/p+n\log p)$ time for weaker PRAM models [77]). These algorithms have a polynomial speedup on polynomially dense graphs, i.e., where the number of edges is $m=\Omega(n^{1+\varepsilon})$ for some constant $\varepsilon>0$. This implies EP algorithms for most graph computations on dense graphs: strongly connected components, topological sorting, etc. The single source shortest path and minimal spanning tree problems also have EP algorithms on dense graphs [27]. EP algorithms are not known for any of these problems on sparse graphs. EP algorithms are known for the "dense" case, but not for the "sparse" case of directed graph reachability, and monotone circuit value [77].

ANC: The connectivity and biconnectivity problems can be solved on sparse graphs $m = O(n)$ in logarithmic time, with a logarithmically bounded inefficiency; these problems are in ANC, but are not known to be in ENC [69].

AP: We do not know of any interesting, natural problems that are in AP but not either in ANC or in EP.

SNC: For problems strictly in P, the class SNC coincides with NC. (We say that a problem is strictly in P if it requires time $n^{\Theta(1)}$ to solve sequentially.) The issue of membership in NC has been widely studied. It is conjectured that the class NC is properly contained in P. Assuming this conjecture, a P-complete problem is not in NC and therefore also not in SNC. An example of a P-complete problem is the generation of the lexicographically first depth-first-search tree for an undirected graph (this is the depth-first-search tree generated by a sequential depth-first-search that scans edges in the natural order) [65].

The definitions of SNC and NC do not, however, coincide on problems with small, e.g., logarithmic sequential complexity. Consider the problem of searching in a sorted list, and assume we restrict ourselves to comparison based algorithms (i.e., to a restricted RAM model; see, e.g., [48, 70]). The parallel complexity of searching is $T_p(n) = \Theta(\log_{p+1} n)$; so, p processors achieve a speedup of $\Theta(\log(p+1))$. Thus, a number of processors that is polynomial in the sequential time $t(n) = \Theta(\log n)$ cannot achieve running time polylogarithmic in $t(n)$. The problem is in NC simply because it can be solved fast sequentially and therefore in parallel; it is not in SNC because no parallel algorithm can significantly speed up the sequential algorithm. Similar trade-offs have been exhibited for other search problems [43].

Furthermore (unlike the class NC), the class SNC is not restricted to problems in P. For example, a problem that requires exponential serial time $t(n) = \Theta(2^n)$ is in SNC if it can be solved in parallel in polynomial time ($T(n) = n^{O(1)}$) using exponentially many processors ($P(n) = 2^{O(n)}$). In particular, P-space algorithms have straightforward parallelizations that yield polynomial time parallel algorithms with an exponential number of processors, so P-space problems that require exponential time are automatically in the class SNC. (This follows from the "parallel computation thesis", see [22].) A similar remark applies to log-space problems that require polynomial time.

SP: Depth-first-search in undirected graphs, the weighted bipartite matching problem (assignment problem), and flows in zero-one networks are in SP [33]. It is not obvious, however, that these problems are not in SNC, especially since they are in RNC (random NC) [1, 44].

3.4.2. Relations between classes

We now turn to the question of how the six classes relate. Figure 1 indicates the obvious inclusions. We conjecture that all these inclusions are proper, and that no

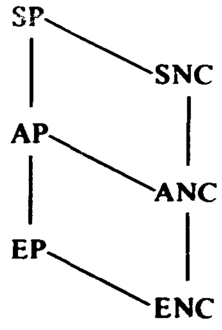

Fig. 1. Inclusion relations among classes.

other inclusions occur. Some partial results to this effect are known. For example, we observed that the problem of finding a lexicographic first depth-first-search tree for dense graphs is in EP. This problem is P-complete even when restricted to dense graphs; hence, assuming NC$\neq$P, SNC (and NC) are not contained in EP; this observation is due to Vitter and Simons [77], who also obtain EP algorithms for several other P-complete problems.

To prove that the reverse inclusion does not occur one would have to exhibit a problem that can be solved polylog fast with polynomial inefficiency, but cannot be solved even polynomially fast if one insists on an efficient algorithm. A possible candidate is the single source shortest path problem. By reduction to matrix multiplication over the (min, +) semiring (transitive closure) one can solve the all pairs shortest path problem in time $O(\log^2 n)$. But such an algorithm executes $\Omega(n^3)$ operations, rather than $O(m+n\log n)$, for a sequential algorithm. The algorithm has polynomial inefficiency. It seems similarly hard to obtain fast efficient algorithms for other path finding problems that are solved by reduction to matrix multiplication: topological ordering of a DAG, maximum matching, finding strong components in a directed graph, detecting cycles, etc. [53, 42, 60]. Each of these problems is solved sequentially in time $O(m+n)$, and in parallel in time $O(\log^2 n)$ with $M(n)$ processors, where $M(n)$ is the number of operations to multiply two $n\times n$ matrices ($M(n)\geq n^2$). A major open problem is to find fast, efficient algorithms for these problems, or to disprove their existence. It would be of utmost interest to show that these problems cannot be solved by ANC algorithms (or even AP algorithms).

Another promising approach is to consider algebraic computations (using only the four arithmetic operations). It is known that if a polynomial of degree d can be computed sequentially with C operations, it can be computed in parallel in time $O(\log C \log d)$, with $C^{O(1)}$ operations [76]. Thus the problem of computing polynomials of moderate degree ($d=C^{O(1)}$ or even $d=2^{\log^k C}$) is in algebraic SNC. However, the reduction to polylogarithmic depth increases the number of operations polynomially. We conjecture that such an increase is unavoidable for some polynomials, which, if true, implies that computing polynomials is not in algebraic ANC.

Such a result has been proven in the much weaker model of monotone algebraic computations (computations using only $+$, $\times$) [72].

3.5. Related work

The fact that efficiency is important in parallel computations has been obvious to the users of parallel computers for a long time. This fact has attracted less attention in theoretical work. The idea that one should consider n large relative to p is formalized in [36]. The problems that can be solved by parallel algorithms with unbounded speedup and constant efficiency are called there "completely parallelizable". Speedup is defined with respect to the "fastest known sequential algorithm".

Vitter and Simons [77] define the class PC to be those problems in the class P that can be solved by a parallel algorithm that has unbounded speedup and polynomial inefficiency, when compared to any sequential algorithm. They define the class PC* to be those problems in the class P that can be solved by a parallel algorithm that has unbounded speedup and constant efficiency, when compared to any sequential algorithm. The last definition implies that the problems in PC* have asymptotically optimal sequential algorithms, up to a constant factor.

Karp and Ramachandran [42] define a parallel algorithm to be "optimal" if it has polylogarithmic running time and is efficient; this class is closely related to ENC. They define an algorithm to be "efficient" if it has polylogarithmic running time, and polylogarithmic inefficiency. This class is closely related to ANC. However, running time is defined as a measure of the problem size n, rather than as a function of $t(n)$, and a parallel algorithm is compared to the "best sequential algorithm".

Finally, Bertoni et al. [12] define OC to be the class of problems solvable with polynomial number of processors in time $o(n^{\epsilon})$ for every constant $\epsilon > 0$.

4. Other parallel computational models

The literature on parallel complexity contains definitions of numerous parallel computing models; it is easy to obtain hundreds of distinct models by systematically varying all the various parameters of model definitions. We examine a small part of this space in Section 4.1. Most models are all truly distinct in their computing abilities; Section 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. For example, the distinction between models that allow concurrent writes and models that disallow them is a great theoretical issue. In practice, it seems simpler to support concurrent writes than concurrent reads: in the first case, one needs an arbitration mechanism to select one among several conflicting writes; in the second case, one needs both an arbiter and a broadcast mechanism to return the same answer to an arbitrary set of processors. Luckily, it turns out that many of these distinctions can be ignored in the research of parallel efficient algorithms. Explicit simulations of one model by another show that these models do not differ much in computing power. This implies that the classes of algorithms and problems we defined are robust across these models. The simulation theorems are listed in Section 4.3 and proved in Section 4.5. The results are discussed in Section 4.4.

4.1. The models

We first describe some variants of the PRAM model. The most powerful one, which we call the "strong PRAM model" will be used in the later simulations. We then look at weaker models. Issues we consider are memory partition, communication latency, synchronism, and sparse network topologies.

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. It is often assumed that alternating read and write cycles are executed. In the COMMON CRCW model, all of the processors must write the same value, so that no conflict occurs. Otherwise, a rule must be used to determine the value stored by conflicting writes. There are many possibilities, including that an arbitrary processor writes (*arbitrary* model), the processor with the lowest identifier writes (*priority* model), or the value written is some combination of all values, e.g., their Boolean OR. Another possibility is that *all* concurrent accesses (reads or writes) are executed, as implied by the usual interleaving semantics of concurrent processes. The outcome of the concurrent execution of several memory accesses is as if these accesses occurred in some serial order (however, all accesses are executed in one time step). In this case, an order must be selected for conflicting accesses. Possibilities include, they occur in an arbitrary order (*arbitrary* model), or they occur in the order of their processor identifiers (*priority* model).

Even stronger PRAM models are obtained by the addition of Read-Modify-Write operations. 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. Suppose, for example, that three processors simultaneously access the same variable v: Processor 1 executes *Store* v R_1, and $R_1 = 5$; Processor 2 executes *Fetch&Add* v R_2, and $R_2 = 1$; and Processor 3 executes *Fetch&Add* v R_3, and $R_3 = 2$. Then the value 5 is returned to register R_2, the value 6 is returned to register R_3, and the final value of v is 8. For the noncommutative arithmetic operations (subtraction and division) we also include *Fetch&InvOp* operations; e.g., *Fetch&InvDiv* v R sets v to R/v, and returns to R the old value of v. We call this model (with priority conflict resolution) the *strong* PRAM model. One could also include other Read-Modify-Write operations; see [51] for a discussion of such operations. Conflicts can be resolved in any of the ways described for concurrent writes.

Bibliographical note

The current terminology for PRAM models (CRCW, CREW, and EREW) was coined by Snir [70]; these models, however, were used long before. The CREW model is implicit in the earliest research on parallel numerical algorithms (see, e.g., [16, Chap. 6], and references therein). The PRAM of Fortune and Wyllie [30] is CREW. The EREW model (called PRAC) is used in [56]. The COMMON CRCW model is used in [53]; the priority CRCW model is used in [34] (his SIMDAG 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 [35, 47, 67]. The CROW model was studied in [28].

4.1.1. *Memory partition*

The PRAM model assumes an extremely powerful memory access mechanism: the weakest model allows any set of p distinct locations to be accessed simultaneously. In practice, many locations will be in the same memory bank; each memory bank supports a constant number of accesses per cycle. A more realistic model is obtained by assuming a fixed number of memory modules, each of which can accept at most one request per cycle. When the number of memory modules is proportional to the number of processors, one obtains a model that has the same computing power (up to a constant factor) as the message passing model described below.

A *Direct Connection Machine* (DCM) consists of autonomous unit-cost RAMs, 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 *SEND*(v, i) stores in the input buffer of processor i the value v. The instruction *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 may be sent at a time to a processor. (The model of a PRAM with an equal number of processors and memory modules is studied in [50], where it is called a *Fully Connected Direct Connection Machine* (FCDCM), in [59] where it is called a *Module Parallel Computer* (MPC), and in [75] where it is called a *Seclusive* PRAM.)

4.1.2. *Communication latency*

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. A Direct Connection Machine (DCM) has *latency* l if m consecutive words in local memory can be transferred to consecutive locations in the local memory of another processor in time $l+m$. The usual DCM model corresponds to a model with zero (or constant) latency. Other reasonable assumptions are polylogarithmic latency ($l=\log^{O(1)} p$) and polynomial latency ($l=p^{O(1)}$).

Latency can be introduced into shared memory models in a similar manner: A block of m consecutive words in shared memory can be copied into local memory in time $l+m$, and vice versa. This model, called BPRAM, is studied in [4].

4.1.3. Synchronism

Real MIMD parallel systems are not in general synchronous. 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 as 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 (varying) speeds. A time measure is defined on such computations by using the slowest processor clock for measure. Equivalently, we assume that each processor executes at least one instruction per time cycle; it may execute more. A precise definition appears in [57] and [9]; their definition captures the concept of an asynchronous CRCW PRAM (where concurrent accesses to the same memory location do not slow down processors). Similar definitions may be given to capture the notion of an asynchronous EREW PRAM, or an asynchronous DCM model.

In any of these models 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 $O(\log p)$. This can be shown optimal for the DCM model [9], the EREW model, or the CRCW model (with no "cumulative" Read-Modify-Write operations such as *Fetch&Add*) [71].

4.1.4. Sparse communication networks

The DCM model assumes that each processor can communicate with any other processor in constant time. Some parallel systems provide only partial connectivity; a processor can send messages only to a (small) subset of "neighbors." Such a system can be represented as a network of communicating RAMs. The connectivity is represented by a directed *communication graph*. A unit cost RAM model is associated with each node, and a unidirectional communication channel is associated with each edge. Processor i can send a message to processor j only if (i,j) is an edge in the communication graph. A computation model of this type consists of a sequence of communication graphs with increasing numbers of nodes; the communication graph is a uniform function of the number of nodes.

The DCM model corresponds to complete communication graphs. In a sparse network the degree d of the communication graph is bounded (e.g., $d=4$ for a butterfly) or is a slowly increasing function of the number of processors p (e.g., $d = 2\log p$ for a hypercube).

For bounded degree graphs, one can drop the assumption of synchronous execution. An asynchronous system can simulate a synchronous one with the same communication graph, with a constant factor overhead. Neighboring processors periodically exchange messages so as to stay nearly synchronized (see, e.g., [31]).

Lower bounds presented in this work for sparse networks are valid for any family of networks with constant (or slowly increasing) degree. Upper bounds are valid for the butterfly and hypercube networks; they also apply to other networks with similar algorithmic properties.

4.2. *Separation theorems*

The various models presented differ in their computing power. Separation theorems are known for most pairs of models, and usually show the existence of a logarithmic gap between the stronger and the weaker model. Below are several theorems of this form. The next subsection shows that, despite these differences, the classes of problems are invariant across large classes of models.

Theorem 4.1 (Cook et al. [23]). *The* OR *of p inputs can be computed in constant time on a* CRCW PRAM *with p processors, but requires $\Omega(\log p)$ time on a* CREW PRAM.

Theorem 4.2 (Snir [70]). *Searching for a key in a sorted table of size n can be done in $O((\log n)/\log p)$ time in a* CREW PRAM *with p processors, but requires $\Omega(\log n - \log p)$ time on an* EREW PRAM *with p processors.*

A simulation is conservative if it executes exactly the same operations as the simulated machine; it may keep several copies of each variable, which are distributed throughout the system.

Theorem 4.3 (Upfal and Widgerson [73]). *A deterministic, conservative on-line simulation of T steps of a p processor* EREW PRAM *on a p processor* DCM *requires time $\Omega(T\log p/\log\log p)$.*

The *token detection problem* is to distinguish an input that consists of all zeroes from an input that has exactly one nonzero entry.

Theorem 4.4 (Snir [71]). *The token detection problem for p inputs can be solved in constant time on a synchronous* DCM *with p processors, but requires time $\Omega(\log p)$ on an asynchronous* DCM.

The same $\Omega(\log p)$ lower bound is valid for asynchronous EREW PRAMs or asynchronous CRCW PRAMs with no cumulative Read-Modify-Write operations.

Theorem 4.5. *The time required to compute the sum of n inputs on a synchronous* DCM *with latency l is $\Theta(l\log(n/l))$.*

Proof. Let $S(t)$ be the maximum number of inputs a processor may have accessed, either directly or indirectly, after t steps. Then

$$S(0)=0, \qquad S(t)\le \max((S(t-1))+1, 2S(t-l)).$$

It follows that $S(t)\le l2^{t/l}$. If summing of n inputs is done in time t_n then $S(t_n)\ge n$. This implies that

$$t_n \ge l\log(n/l). \qquad \square$$

The last result essentially shows a gap of l between a model with latency one and a model with latency l; in particular, we obtain a logarithmic gap for models with logarithmic latency. A matching upper bound is achieved by an obvious algorithm.

A permutation algorithm is *conservative* if it permutes by moving the inputs without computing any new values on them. Any static permutation can be computed on a DCM in time $O(n/p)$. The following theorem shows a $\Theta(\log p)$ separation between the DCM model and fixed degree networks. (A similar theorem is proven in [36], with a much more complicated proof.)

Theorem 4.6. *A conservative algorithm that computes all circular shift permutations on n inputs $x_1,\ldots,x_n$ on a fixed degree network with p processors has running time $\Omega((n\log p)/p)$.*

Proof. Let $d(i,j)$ be the distance in the network from the node containing x_i to the node containing x_j. If some node contains $(n\log n)/p$ inputs, then we are done. Otherwise, a simple counting argument shows that for any i

$$\sum_{j=1}^{n} d(i,j)=\Omega(n\log p).$$

A k-shift requires time at least

$$t_k=\frac{1}{p}\sum_{i=1}^{n} d(i,i+k).$$

The total time for all possible k-shifts is

$$\sum_{k=1}^{n} t_k=\frac{1}{p}\sum_{k=1}^{n}\sum_{i=1}^{n} d(i,i+k)=\frac{1}{p}\sum_{i=1}^{n}\sum_{j=1}^{n} d(i,j)=\Omega\left(\frac{n^2\log p}{p}\right).$$

Thus, some k-shift requires time $\Omega((n\log p)/p)$. $\square$

The same argument holds for any family of n permutations $\sigma_1,\ldots,\sigma_n$ on $x_1,\ldots,x_n$ such that for any pair i,j there is exactly one permutation σ_k such that

$\sigma_k(x_i) = x_j$. Many computation problems can be shown to require the computation of such a transitive family of permutations; see [78].

4.3. *Simulations*

Lower bounds that separate distinct computation models are obtained by exhibiting specific problems that one model solves better than the other. Upper bounds on the separation between models are obtained by simulating one model by the other. We shall usually consider simple simulations whereby each variable of the simulated machine is represented by a unique variable on the simulating machine; the simulating machine runs a step-by-step simulation, executing at each successive simulation phase the operations executed at the corresponding step by the simulated machine.

We are interested in simulations that preserve the various classes of algorithms we defined, thus showing the definitions to be robust. Accordingly, we have to consider two issues: how fast the simulation is, and how efficient it is. The two issues are distinct. Typically one considers the simulation of a p-processor machine of one type by a p-processor machine of another type. However, this is not necessary. We shall see that it is often expedient to reduce the number of processors of the simulating machine, so as to overcome the limitations of the weaker model. This reduction in parallelism will provide an increase in efficiency; this is acceptable as long as the slow down is within the boundaries of the class definition.

We assume a machine with p processor is simulating a machine with $q \geq p$ processors. The first parameter of the simulation is the relation between p and q.

Definition. A simulation has a

(1) *constant reduction* (in parallelism) if $p = \Theta(q)$;

(2) *polylog reduction* (in parallelism) if $p = q/\log^{\Omega(1)} q$; and

(3) *polynomial reduction* (in parallelism) if $p \leq q^{\varepsilon}$ for some $\varepsilon < 1$.

The next parameter is the *inefficiency* of the simulation, i.e., the extra amount of work done by the simulating machine. Formally, the simulation has inefficiency v if each step of the simulated machine is simulated in $O(v \cdot q/p)$ steps.

Definition. A simulation has

(1) (at most) *constant inefficiency* if $v = O(1)$; and

(2) *polylog inefficiency* if $v = \log^{O(1)} q$.

For convenience, we will say that a simulation with constant inefficiency has (at least) *constant efficiency* or simply is *efficient.*

To illustrate these definitions, consider the results claimed in the following theorems (proofs and references are given in Section 4.5).

Theorem 4.7. *T steps of a p processor strong* PRAM *can be simulated by a p processor* EREW PRAM *with latency l in time* $O(Tl \log p)$.

Theorem 4.8. *T steps of a q processor strong* PRAM *with q^k memory locations can be simulated by a $p \leqslant q^{1-\varepsilon}$ $(\varepsilon > 0)$ processor* EREW PRAM *in time* $O(kTq/p)$.

Theorem 4.9. *T steps of a q processor strong* PRAM *with q^k memory locations can be simulated probabilistically by a $p \leqslant q^{1-\varepsilon}$ $(\varepsilon > 0)$ processor* DCM *in time* $O(kTq/p)$.

Theorem 4.10. *T steps of a q processor strong* PRAM *with q^k memory locations can be simulated probabilistically by a p processor* DCM *with latency l in time* $O(kTq/p)$ *if* $p \cdot l \leqslant q^{1-\varepsilon}$ $(\varepsilon > 0)$.

Theorem 4.11. *T steps of a p processor strong* PRAM *can be simulated by a p processor butterfly network deterministically in* $O(T \log^2 p)$ *time and probabilistically in* $O(T \log p)$ *time.*

Theorem 4.12. *T steps of a q processor, synchronous arbitrary* PRAM *can be simulated by a $p = q/\log q$ processor, asynchronous (arbitrary)* PRAM *in time* $O(T \log p)$.

The first simulation (Theorem 4.7) has a constant reduction in parallelism but has a polylog inefficiency. The same holds true of the simulation of a strong PRAM by a butterfly (Theorem 4.11). Theorem 4.8 shows that the simulation of a strong PRAM by an EREW PRAM can be done efficiently (constant inefficiency), at the cost of a polynomial reduction in parallelism. The same holds true of the simulation of a strong PRAM by a DCM in Theorems 4.9 and 4.10. Finally, the last simulation of a synchronous, arbitrary PRAM by an asynchronous, arbitrary PRAM (Theorem 4.12) is efficient with a polylog reduction in parallelism.

The motivation for this classification of simulations is given by the next theorem.

Theorem 4.13. (1) *Efficient simulations with a polynomial reduction of parallelism map* EP *algorithms into* EP *algorithms.*

(2) *Polylog inefficient simulations with a polynomial reduction of parallelism map* AP *algorithms into* AP *algorithms.*

(3) *Efficient simulations with polylog reduction in parallelism map* ENC *algorithms into* ENC *algorithms.*

(4) *Polylog inefficiency simulations with polylog reduction in parallelism map* ANC *algorithms into* ANC *algorithms.*

Theorem 4.13, coupled with the simulations given above, immediately implies the robustness of the various classes of problems.

Theorem 4.14. *The class* EP *is invariant under the* PRAM *models (both synchronous and asynchronous) for deterministic algorithms; it is invariant under the* PRAM *and* DCM *models for probabilistic algorithms. This holds for any latency* $p^{O(1)}$.

Theorem 4.15. *The classes* AP *and* ANC *are invariant under the* PRAM *and* DCM *models and the butterfly and hypercube networks. The former holds for any latency* $p^{O(1)}$, *and the latter holds for any latency* $\log^{O(1)} p$.

4.4. Discussion

Define two models to be equivalent with respect to a class of simulations if each can simulate the other by a simulation in this class. The simulation results show that, if polylogarithmic inefficiencies are tolerated, all the major models presented in this paper are equivalent: they are equivalent with respect to simulations with polylog inefficiency and polylog reduction in parallelism. Thus, the classes AP and ANC are invariant across these models. Algorithms developed on one model yield algorithms of the same class for any other model; a unique complexity theory can be developed.

If one insists on efficient algorithms then distinctions appear among these models. The various PRAM and DCM models are equivalent with respect to efficient (probabilistic) simulations with polynomial reduction in parallelism: the class EP is invariant across these models. The significant feature of these models is that they offer "constant bandwidth" per processor (with various restrictions); for example, processors can exchange messages according to an arbitrary permutation at the rate of one word per processor per time unit. This holds true for asynchronous models as well, and extends to models with large latency. For example, a DCM with polynomial latency can simulate a PRAM with constant efficiency and polynomial reduction of parallelism. If one allows the nodes in a hypercube to send or receive simultaneously a message on each of their incident edges then the hypercube can probabilistically simulate a DCM with constant efficiency and a logarithmic reduction in parallelism [75]. The same result holds for a DCM machine where the p processors are connected by a network of $p \log p$ constant degree switches, arranged in a butterfly configuration (this follows from the results of Ranade [64]). The last two models still have constant bandwidth per processor, albeit logarithmic latency, in a probabilistic sense; this allows them to be (probabilistically) equivalent to a PRAM model.

One can compensate for large latency by decreasing the level of parallelism, without losing efficiency; one cannot similarly compensate for small bandwidth. In a fixed degree network there is a logarithmic degradation of bandwidth. Thus, a fixed degree network cannot simulate a PRAM without incurring a logarithmic overhead. For example, Theorem 4.6 implies that for any $q < p$ there is a problem that can be solved on a p processor PRAM (or DCM) in time $O(n/p)$, but requires time $\Omega(n(\log q)/q)$ to solve on a fixed degree network with q processors. Fixed degree networks are not equivalent, with respect to efficient simulations, to PRAM models.

There are still large gaps in our understanding of the relative powers of the various models. We conjecture that all pairs of models are distinct in the sense that they cannot simulate each other with constant reduction in parallelism and constant efficiency. The separation theorems show this conjecture to be true for a PRAM vs. a DCM (Theorem 4.3), a DCM with latency one vs. a DCM with latency $\log p$ (Theorem 4.5), and a DCM vs. a fixed degree network (Theorem 4.6). We do not have a proof of this seemingly obvious conjecture for the various pairs of PRAM models (CRCW vs. CREW, CREW vs. EREW); it is conceivable (but unlikely) that a p processor EREW PRAM can simulate T steps of a p processor CRCW PRAM in time $O(T+\log p)$.

One would also like to improve the simulation results. Can one show, for example, that the various PRAM models and the DCM model are equivalent with respect to efficient simulations with polylog reduction of parallelism? This would imply that the class ENC is invariant across these models. The two issues to resolve are (1) simulating concurrent memory accesses on a model that forbids it (CRCW and CREW vs. EREW); and (2) distributing memory across modules (PRAM vs. DCM). The simulation of concurrent memory accesses uses sorting. To achieve an efficient simulation with polylog reduction of parallelism one would need an integer sorting algorithm for the EREW PRAM or DCM model that sorts p numbers in range $1, \ldots, p^{O(1)}$ in time $\log^k p$ with $p/\log^k p$ processors. The PRAM memory is distributed in a DCM using random universal hashing function. We presently use constant degree polynomials over finite fields; these can be computed in constant time, but do not give a sufficiently even distribution of memory request across modules, unless q is polynomially larger than p. If one uses polynomials of degree $\log p$ then, in order to obtain a good distribution of memory requests in the simulation of Theorem 4.9, it is sufficient to have $p \log p \leq q$ [41]. If the time for computing the hashing function is ignored, then a p processor DCM can efficiently simulate a $p \log p$ processor EREW PRAM. Ignoring the cost of memory hashing may be justified in certain cases (e.g., if the computation of the hashed address is done in hardware).

If one allows only one message to be sent or received per cycle at each node of a hypercube then the lower bound of Theorem 4.6 applies and the hypercube is strictly weaker than a PRAM, with respect to the classes ENC or EP. The same holds true for any sparse network. This still leaves open the issue of the relations among the various fixed degree networks. Efficient simulations with logarithmic reduction in parallelism across various fixed degree networks, e.g., shuffle-exchange networks and butterfly networks have been recently obtained [45]. Thus, these two types of networks define the same ENC class.

4.5. *Proofs of simulation results*

We present here the proofs and references to the simulation theorems listed in Section 4.3.

The proof of Theorem 4.7 is well known, and has appeared several times in the literature. We present the full proof here, primarily because we will build on it in the proofs of the later theorems, and secondarily because our construction is slightly more general than usual (since we are simulating a strong PRAM). The simulation uses well-known techniques introduced by Nassimi and Sahni [61].

Proof of Theorem 4.7. Assume temporarily that the latency l is constant. Note that each operation of the form $x \to x\ op\ e$, or $x \to e\ op\ x$, where op is addition, subtraction, multiplication, or division, and e is a constant, can be written in the form

$$x \to \frac{ax+b}{cx+d}, \tag{4.1}$$

for some suitable choice of constants a, b, c, d. If A is the 2×2 matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

then we denote by f_A the mapping defined in (4.1). Note that the coefficients of the composition of two such mappings are obtained by multiplying the two matrices

$$f_A \circ f_B = f_{AB}.$$

Note, too, that the numerator $ax+b$ and denominator $cx+d$ of the quotient $(ax+b)/(cx+d)$ can be obtained by computing the product

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ 1 \end{pmatrix}.$$

Thus, to compute the outcomes of a sequence of *Fetch&Op* operations applied to a location with value v, one can compute the partial products

$$\vec{v}, A_1\vec{v}, A_2A_1\vec{v}, \ldots, A_n \cdots A_2A_1\vec{v},$$

where $\vec{v} = \begin{pmatrix} v \\ 1 \end{pmatrix}$ and A_i is the 2×2 matrix associated with the ith operation (this construction is due to Brent [17]). Load and Store operations can also be handled this way: A Load is merely a *Fetch&Add* with zero increment; a Store corresponds to a mapping $x \to e$, which is also a particular case of (4.1). (We shall actually implement a *swap* that stores the value e in memory and returns the old value of x; a store is obtained by ignoring the value returned.)

The concurrent execution of a sequence of memory requests proceeds in the following stages:

(1) *Issue Requests*: Each processor creates a record $\langle id, addr, A\rangle$, where *id* is the processor identifier, *addr* is the address of the memory location to be accessed, and A is the matrix encoding the *Fetch&Op* operation (a processor that need not access memory at that cycle issues a request to a dummy location). The records are stored in an array of size p; this takes constant time.

(2) *Sort Requests*: Sort the records by keys $\langle addr, id\rangle$, which groups requests for the same location together (*id* is used as a secondary key since we are simulating the priority model); this takes time $O(\log p)$ [20].

(3) *Read Memory*: For each first record in a group of requests to the same location (with same *addr* value), read the value v at address *addr*; this takes constant time.

(4) *Compute Accesses*; Combine requests within each group; i.e., for each sequence of records with the same address, and with associated matrices $A_j, A_{j+1}, \ldots, A_k$, compute the partial products $\vec{v}_{j-1}, \vec{v}_j, \ldots, \vec{v}_k$, where

$$\vec{v}_i = \begin{cases} \begin{pmatrix} v \\ 1 \end{pmatrix} & \text{for } i = j-1 \\ A_i\vec{v}_{i-1} & \text{for } i = j, j+1, \ldots, k; \end{cases}$$

compute v_i, the quotient of the two coefficients of $\vec{v}_i$, for $i = j-1, \ldots, k$; store v_{i-1} in the record of the ith request. This takes $O(\log p)$ time using parallel prefix by groups [68].

(5) *Write Memory*: Write each final answer v_k back at address *addr*; this takes constant time.

(6) *Route Answers*: Sort records by *id*, which returns to each processor its desired value; this takes time $O(\log p)$ [20].

If the latency l is not constant, just multiply the above execution times by l. Thus, the simulation of one memory access step takes $O(l \log p)$ time on an EREW model. □

The proof of Theorem 4.8 is very similar. It relies on the ability to sort memory addresses efficiently.

Lemma 4.16 (Kruskal et al. [52], Reif [66], Wagner and Han [80]). *A p processor* EREW PRAM *can sort n integers in the range* $0, \ldots, R-1$ *in time*

$$O\left(\left(\frac{n}{p} + \log p\right)\frac{\log R}{\log(n/p)}\right).$$

Proof of Theorem 4.8. Each processor of the p processor simulating EREW PRAM simulates q/p processors of the q processor strong PRAM. We execute essentially the same simulation as in the proof of Theorem 4.7, except each processor of the EREW PRAM issues q/p requests to simulate one memory access step of the strong PRAM. The dominant time is the sorting in steps (2) and (6), which by Lemma 4.16 is

$$O\left(\left(\frac{q}{p} + \log p\right)\frac{\log(q^k)}{\log(q/p)}\right).$$

We have assumed that p is (at least) polynomially smaller than q, so the sorting time simplifies to $O(kq/p)$. □

We now consider Theorem 4.9, which states that a DCM can efficiently simulate a strong PRAM with a polynomial reduction in parallelism. We use essentially the same simulation as in the previous theorem. Each processor of the p processor simulating DCM simulates q/p processors of the q processor strong PRAM. There are several technical difficulties that must be handled: integer sorting on the DCM must be efficient; memory accesses must be well distributed across the simulating processors; and processors must not conflict as they access the models.

Integer sorting is taken care of by the following lemma.

Lemma 4.17 (Han [38]). *A p processor* DCM *can sort n integers in the range* $0, \ldots, R-1$ *in time*

$$O\left(\left(\frac{n}{p}+p^{\varepsilon}\right)\log\frac{R}{\log n}\right),$$

for a constant $\varepsilon > 0$.

We now show how to distribute memory. Hashing is used to distribute memory locations randomly across the memory modules so that it is almost surely the case that no module will receive more than $2q/p$ requests. Whenever some module receives more than $2q/p$ requests, the memory locations are completely rehashed. Universal hashing [18] is used to ensure that the hash function is truly random. This is similar to the technique used in [41] to simulate PRAMs by Ultracomputers. However, in order to obtain an efficient simulation, the hash function must be computable in constant time, so a different analysis is required.

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

$$H_d = \left\{h: h(x) = \left(\left(\sum_{i=0}^{d-1} a_i x^i\right) \bmod r\right) \bmod p,\ a_i \in 0, \ldots, r-1\right\}.$$

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 4.18 (Carter and Wegman [18]). *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.

Lemma 4.19 (Dietzfelbinger [26]). *Let X be a 0-1 valued random variable such that $\mathrm{E}(X)=\mathrm{Prob}(X=1)=\mu$. Then there exists a constant α (that depends on d, but not on n) such that the following holds. If $n \geq d/(2\mu)$ and $X_1, \ldots, X_n$ are d-independent variables equidistributed as X then*

$$\mathrm{E}\left(\left(\sum_{i=1}^{n}(X_i-\mu)\right)^d\right) \leq \alpha(\mu n)^{d/2}.$$

Proof. Let $Y=X-\mu$ and $Y_i=X_i-\mu$. Then, for $j \geq 2$,

$$\begin{aligned}\mathrm{E}(|Y_i|^j) &= \mathrm{Prob}(X_i=1)\cdot(1-\mu)^j+\mathrm{Prob}(X_i=0)\cdot\mu^j\\ &= \mu(1-\mu)((1-\mu)^{j-1}+\mu^{j-1}) \leq \mu(1-\mu) \leq \mu.\end{aligned}$$

We have

$$\begin{aligned}\mathrm{E}\left(\left(\sum_{i=1}^{n}Y_i\right)^d\right) &= \sum_{r=1}^{d}\sum_{1\leq i_1<\cdots<i_r\leq n}\sum_{\substack{j_1,\ldots,j_r\geq 1\\ j_1+\cdots+j_r=d}}\binom{d}{j_1,\ldots,j_r}\mathrm{E}(Y_{i_1}^{j_1}\cdots Y_{i_r}^{j_r})\\ &= \sum_{r=1}^{d}\sum_{1\leq i_1<\cdots<i_r\leq n}\sum_{\substack{j_1,\ldots,j_r\geq 1\\ j_1+\cdots+j_r=d}}\binom{d}{j_1,\ldots,j_r}\mathrm{E}(Y_{i_1}^{j_1})\cdots\mathrm{E}(Y_{i_r}^{j_r})\end{aligned}$$

(since the variables are d-independent). Since $E(Y_i)=0$ we obtain

$$\begin{aligned}\mathrm{E}\left(\left(\sum_{i=1}^{n}Y_i\right)^d\right) &= \sum_{r=1}^{d/2}\sum_{1\leq i_1<\cdots<i_r\leq n}\sum_{\substack{j_1,\ldots,j_r\geq 2\\ j_1+\cdots+j_r=d}}\binom{d}{j_1,\ldots,j_r}\mathrm{E}(Y_{i_1}^{j_1})\cdots\mathrm{E}(Y_{i_r}^{j_r})\\ &\leq \sum_{r=1}^{d/2}\sum_{1\leq i_1<\cdots<i_r\leq n}\sum_{\substack{j_1,\ldots,j_r\geq 2\\ j_1+\cdots+j_r=d}}\binom{d}{j_1,\ldots,j_r}\mathrm{E}(|Y_{i_1}|^{j_1})\cdots\mathrm{E}(|Y_{i_r}|^{j_r})\\ &\leq \sum_{r=1}^{d/2}\sum_{1\leq i_1<\cdots<i_r\leq n}\sum_{\substack{j_1,\ldots,j_r\geq 2\\ j_1+\cdots+j_r=d}}\binom{d}{j_1,\ldots,j_r}\mu^r\\ &< \sum_{r=1}^{d/2}\binom{n}{r}r^d\mu^r < (d/2)^d\sum_{r=1}^{d/2}\binom{n}{r}\mu^r.\end{aligned}$$

Since $n\mu \geq d/2$, the terms in the last sum are increasing, and we use the last term to estimate the sum. We obtain

$$\mathrm{E}\left(\left(\sum_{i=1}^{n}Y_i\right)^d\right) < (d/2)^{d+1}\binom{n}{d/2}\mu^{d/2} \leq \frac{(d/2)^{d+1}}{(d/2)!}(n\mu)^{d/2}.$$

The result follows, with $\alpha=(d/2)^{d+1}/(d/2)!$. □

Corollary 4.20. *Let $X_1, \ldots, X_n$ be 0-1 valued, d-independent, equidistributed random variables. Let $\mu=\mathrm{E}(X_i)$. Then, for $n \geq d/(2\mu)$,*

$$\mathrm{Prob}\left(\sum_{i=1}^{n}(X_i-\mu)>\varepsilon\right) \leq \frac{\alpha(n\mu)^{d/2}}{\varepsilon^d}.$$

Proof. We have

$$\begin{aligned}\mathrm{Prob}(\textstyle\sum (X_i-\mu)>\varepsilon) &= \mathrm{Prob}((\textstyle\sum (X_i-\mu))^d>\varepsilon^d)\\ &\leqslant \mathrm{E}((\textstyle\sum (X_i-\mu))^d)/\varepsilon^d \leqslant \frac{\alpha(n\mu)^{d/2}}{\varepsilon^d}. \qquad \square\end{aligned}$$

We now apply these results to the memory distribution problem.

Proof of Theorem 4.9. As mentioned earlier, we use essentially the same simulation as in the previous theorem. Each processor of the p processor simulating DCM simulates q/p processors of the q processor strong PRAM. We hash memory using a universal hash function 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 hash 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 ith memory access hashes to processor j, zero otherwise. The random variables $X_1^j,\ldots,X_q^j$ are d-independent, according to Lemma 4.18; $E(X_i^j)=1/p$. The number of accesses destined to processor j equals $\sum_{i=1}^q X_i^j$. According to Corollary 4.20 the probability of more than $2q/p$ accesses at processor j is bounded by

$$\mathrm{Prob}\left(\sum_{i=1}^{q} X_i^j>\frac{2q}{p}\right)=\mathrm{Prob}\left(\sum_{i=1}^{q}\left(X_i^j-\frac{1}{p}\right)>\frac{q}{p}\right)\leqslant\frac{\alpha\cdot(q\cdot 1/p)^{d/2}}{(q/p)^d}=\alpha\left(\frac{p}{q}\right)^{d/2}.$$

The probability that more then $2q/p$ accesses occur at any of the p simulating processors is bounded by

$$\alpha\cdot\frac{p^{1+d/2}}{q^{d/2}}\leqslant\alpha\cdot\frac{q^{(1-\varepsilon)(1+d/2)}}{q^{d/2}}=\alpha q^{1-\varepsilon-\varepsilon d/2}.$$

Choose $d>2(k/\varepsilon-1)$.

By Lemma 4.17, sorting takes time $O(q/p)$ in the range of interest. It only remains to show how to Read Memory (Step (3)) without issuing conflicting requests. (Write Memory (Step (5)) can be handled in the same manner.)

We assume that the array containing the request records is stored so that the first processor has the first q/p records, the second has the next q/p records, and so on. We also assume that a hashed address is of the form *addr* = ⟨*destination*, *addr'*⟩, where *destination* is the number of the processor that stores this word, and *addr'* is the address within the local memory of the processor. Thus, records for requests with the same destination are consecutive in the request array.

(1) Mark as active the requests to be executed at Step (3) (the first request in a group of requests with the same address). This takes time $O(q/p)$. Each processor will execute the accesses on behalf of the active records in its segment of the request array. One needs to order these accesses so as to prevent conflicts.

(2) Rank the active requests within each group of active records with the same destination, using parallel prefix by groups. This takes time $O(q/p+\log p)$.

(3) At this point, every active request with a particular destination has a distinct positive integer ($\leqslant 2q/p$) assigned to it. This number can be used by the processor where the record currently resides as a distinct time in which to perform the access. The only minor problem is that a processor may have more than one access to execute at the same time if the processor holds records of active requests with several different destinations. For such a processor, we need be concerned only with the destinations of the first and last active requests in its segment of requests. Only these two destinations are shared by requests in other segments. Requests to other destinations do not conflict and can be handled at any time. Let d_f and d_l be, respectively, the destinations of the first and last active record in a processor segment. The processor executes three rounds, each consisting of $2q/p$ steps; in the first round the processor performs accesses for active records with destination d_f, at the time steps defined by the ranking; in the second round it performs accesses for active records with destination d_l (if $d_l \neq d_f$) at the times specified by the ranking; in the last round it performs in arbitrary order accesses for the remaining active records. This completes the implementation of Read Memory. □

The same simulation is used for the DCM model with latency. We have to show that the basic operations, in particular prefix computation and sorting, can be executed efficiently in this model.

Lemma 4.21. *A DCM with p processors and latency l can execute a parallel prefix computation on n items in time* $O(n/p+l\log p)$.

Proof. Obvious. □

Lemma 4.22. *A DCM with p processors and latency l can execute any fixed permutation on n items, where* $n>(l\cdot p)^{1+\varepsilon}(\varepsilon>0)$, *in time* $O(n/p)$.

Proof. This result was proven in [4] for an EREW PRAM with latency l (the BPRAM model). Their permutation algorithm transfers data from shared memory and back in blocks of size $\geqslant l$. With a constant increase in running time, the algorithm can be modified to access shared memory in blocks of size l, with fixed boundaries. The modified algorithm alternates between local computation phases and shared memory access phases. Each shared memory access phase takes time $\Theta(l)$; each processor accesses during such a phase a distinct block in shared memory. Let T

be the number of shared memory access phases. Vishkin and Widgerson [79] show that an oblivious EREW PRAM algorithm can be simulated on a DCM with the same number of processors, so that each PRAM step is simulated in a constant number of DCM steps. We apply the same transformation to our algorithm, treating each size l block as one "word". We obtain a DCM algorithm with $O(T)$ communication steps; at each communication step a processor either sends or receives a message of length l. Thus, the running time of the algorithm on the DCM model is the same, up to a constant factor, as on the EREW PRAM model. □

Lemma 4.23. *A DCM with p processors and latency l can sort n integers in range* $1, \ldots, n^k$, *where* $n > (l \cdot p)^{1+\varepsilon} (\varepsilon > 0)$, *in time* $O(kn/p)$.

Proof. Using Leighton's [55] "columnsort" algorithm recursively, one can sort n items by executing a constant number of phases, each consisting of $O(n^{1-\varepsilon/2})$ independent sorts on $O(n^{\varepsilon/2})$ items, and a constant number of fixed (static) permutations on n items. Using the previous lemma, the permutations are executed in time $O(n/p)$. Using radix sort, $O(n^{\varepsilon/2})$ items in range $1, \ldots, n^k$ can be sorted sequentially in time $O(kn^{\varepsilon/2})$. Since $n^{1-\varepsilon/2} > p$, all the sorts can be executed in parallel in time $O(kn/p)$. □

Proof of Theorem 4.10. We run the same simulation used in the proof of Theorem 4.9. According to Lemmas 4.21 and 4.23, sorting and parallel prefix are executed in time $O(q/p)$, in the range of interest. The only nontrivial issue left is to show that Read Memory (Step (3)) can be executed in time $O(q/p)$. The communications occur in three rounds. In each of the first two rounds each processor accesses at most one other processor. Using messages of length l, each of these two rounds are executed in time $O(q/p + l)$. In the last round a processor has exclusive access to a (possibly empty) set of up to p processors. Using a binary tree communication scheme, where communications are pipelined along the paths from the root to the leaves, the requests can be distributed to all the processors in time $O(q/p + l \cdot \log p) = O(q/p)$; by reversing the distribution schedule, the replies can be gathered back in the same amount of time. □

The results in Theorem 4.11 were proved in [73] and [8] for deterministic simulations; the simulations are not uniform (in p). Uniform probabilistic versions were proved in [41] and [64].

Proof of Theorem 4.12. Each processor of the p processor simulating DCM simulates q/p processors of the q processor strong PRAM. After each processor has executed its q/p memory access operations, it executes a synchronize routine. This last step takes $O(\log p)$ time. □

5. Summary and open questions

We have outlined in this paper a new approach to the classification of parallel algorithms, and defined several classes. Of particular importance is the class, EP, of efficient algorithms with polynomial reduction in running time. This seems to capture the type of parallel algorithm that is actually used on real parallel computers. The results on the invariance of the class EP under the various PRAM and DCM models allow us to develop parallel algorithms on a convenient (powerful) parallel model, knowing our results will port to all other models, while preserving the essential properties of the algorithm. Consider, for example, the probabilistic, efficient, logarithmic time parallel algorithm for connected and biconnected components, obtained in [32] for the CRCW model. Our results immediately imply that parallel efficient algorithms exist for these problems on all the other PRAM models, and on the DCM model. Such algorithms were not previously known.

We looked at simulations across various parallel computing models, with an emphasis on the efficiency of the simulation. An important paradigm in our results is that, rather than sacrificing the efficiency of a computation, one can sacrifice the level of parallelism. A weaker model is not necessarily a less efficient model; rather, it is a model where the ratio of the problem size to the number of processors used must be larger in order to obtain the same efficiency. These results reinforce the well-known pragmatic rule that any parallel machine can be used efficiencly, provided it is used to solve large enough problems [37, 49]. (Our analysis ignores practical obstacles to this approach, such as the finite amount of memory at each processor.)

Open problems motivated by our approach were proposed in the discussions in Sections 3 and 4. Another direction for further research is to consider restricted classes of algorithms. The simulation results for the various models were "worst-case" results where nothing is known on the type of problem at hand; better results may be obtained with more knowledge. For example, if the algorithm is *oblivious*, so that the pattern of memory accesses is fixed independently of the input value, then a DCM model can simulate a CREW PRAM with constant overhead and no reduction in parallelism [39, 79]. It is also easy to show that asynchronous models can simulate synchronous ones with constant overhead and no reduction in parallelism for oblivious computations. Another example is that of algorithms for PRAMs with local memories [3]: whenever these algorithms have a good ratio of local computations to global communications (good *locality*), they can be simulated on a fixed degree network with constant overhead.

In both cases listed above (oblivious routing and good locality), better simulation methods can be used if it is known in advance that the computation is of a particular form. Of more interest is to develop *adaptive* simulation methods, that yield the right worst-case performance, but perform better when the computation is of a particular, convenient form. This could be achieved by a combination of compiling techniques, that reorganize the algorithm code, and on-line resource management techniques, that adapt to the ongoing computation. The work of Aggarwal and Chandra [2] on memory management in uniprocessors provides the flavor of such an approach.

Acknowledgment

This work benefited from useful criticisms from Ron Fagin, Carl Ponder, Moshe Vardi, and Jeffrey Vitter, and help from Michael Benor, Pilar de la Torre, Bill Gasarch, Kathy Hilbers, Prabhakar Raghavan, and Eli Upfal. We thank Martin Dietzfelbinger for informing us of his improvement to Lemma 4.19.

References

[1] A. Aggarwal and R.J. Anderson, A random NC algorithm for depth first search, *Combinatorica* **8** (1988) 1-12.

[2] A. Aggarwal and A. Chandra, Virtual memory algorithms, in: *Proc. 20th ACM Symp. on Theory of Computing*, Chicago, IL, U.S.A. (1988) 173-185.

[3] A. Aggarwal and A.K. Chandra, Communication complexity of PRAM's, in: T. Lepistö and A. Salomaa (eds.), *Proc. 15th Int. Colloquium on Automata, Languages and Programming*, Tampere, Finland, Lecture Notes in Computer Science **317** (Springer, Berlin, 1988) 1-17.

[4] A. Aggarwal, A.K. Chandra and M. Snir, On communication latency in PRAM computations, *Proc. ACM Symp. on Parallel Algorithms and Architectures*, Santa Fe, New Mexico, USA (1989) 11-21.

[5] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[6] M. Ajtai, J. Komlos, W.L. Steiger and E. Szemeredi, Optimal parallel selection has complexity O(log log *n*), *J. Comput. System Sci.* **38** (1989) 125-133.

[7] M. Ajtai, J. Komlos and E. Szemeredi, Sorting in *c* log *n* parallel steps, *Combinatorica* **3** (1983) 1-19

[8] H. Alt, T. Hagerup, K. Mehlhorn and F.P. Preparata, Deterministic simulation of idealized parallel computers on more realistic ones, *SIAM J. Comput.* **16** (1987) 808-835.

[9] E. Arjomandi, M. Fisher and N. Lynch, Efficiency of synchronous versus asynchronous distributed systems, *J. Assoc. Comput. Mach.* **30** (1983) 449-456.

[10] K.E. Batcher, Sorting networks and their applications, in: *Proc. AFIPS* **32** (1968) 307-314.

[11] G. Baudet and G. Stevenson, Optimal sorting algorithms for parallel computers, *IEEE Trans. Comput.* **27** (1978) 84-87.

[12] A. Bertoni, M. Goldwurm, G. Mauri and N. Sabatini, Parallel algorithms and the classification of problems, in: J.D. Becker and I. Eisele (eds.), *Proc. Workshop on Parallel Processing: Logic, Organization and Technology* (*WOPPLOT86*), Neubiberg, FRG, Lecture Notes in Computer Science **253** (Springer, Berlin, 1986) 206-226.

[13] G. Bilardi and A. Nicolau, Adaptive bitonic sorting: an optimal parallel algorithm for shared memory machines, *SIAM J. Comput.* **18** (1989) 216-228.

[14] M. Blum, A machine-independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.* **14** (1967) 322-336.

[15] A. Borodin and J.E. Hopcroft, Routing, merging and sorting on parallel models of computation, in: *Proc. 14th ACM Symp. on Theory of Computing*, San Francisco, CA, U.S.A. (1982) 338-344.

[16] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems* (Elsevier, New York, 1975).

[17] R.P. Brent, The parallel evaluation of general arithmetic expressions, *J. Assoc. Comput. Mach.* **21** (1974) 201-206.

[18] L. Carter and M. Wegman, New hash functions and their use in authentication and set equality, *J. Comput. System Sci.* **22** (1981) 265-279.

[19] F.Y. Chin, J. Lam and I.-N. Chen, Efficient parallel algorithms for some graph problems, *Comm. ACM* **25** (1982) 659-665.
[20] R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988) 770-785.
[21] R. Cole and U. Vishkin, Approximate and exact parallel scheduling with applications to list, tree and graph problems, in: *Proc. 27th Ann. Symp. on Foundations of Computer Science*, Toronto, Canada (1986) 478-491.
[22] S.A. Cook, Toward a complexity theory of synchronous parallel computations, *Enseign. Math.* **XXVII** (1981) 99-124.
[23] S.A. Cook, C. Dwork and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **15** (1986) 87-97.
[24] L. Csanky, Fast parallel inversion algorithms, *SIAM J. Comput.* **5** (1976) 618-623.
[25] R. Cypher and J.L.C. Sanz, Cubesort: an optimal sorting algorithm for feasible parallel computers, in: J.H. Reif (ed.), *3rd Aegean Workshop on Computing, AWOC 88*, Corfu, Greece, Lecture Notes in Computer Science **319** (Springer, Berlin, 1988) 456-464.
[26] Martin Dietzfelbinger, private communication.
[27] J.R. Driscoll, H.N. Gabow, R. Shrairman and R.E. Tarjan, Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation, *Comm. ACM* **31** (1988) 1343-1354.
[28] P.W. Dymond and W.L. Ruzzo, Parallel random access machines with owned global memory and deterministic context free language recognition, in: L. Kott (ed.), *Proc. 13th Int. Colloquium on Automata, Languages and Programming*, Rennes, France, Lecture Notes in Computer Science **226** (Springer, Berlin, 1986) 95-104.
[29] D.M. Eckstein and D. Alton, Parallel graph processing using depth-first search, in: *Conf. on Theoretical Computer Science at the University of Waterloo* (1977) 21-29.
[30] S Fortune and J. Wyllie, Parallelism in random access machines, in: *Proc. 10th ACM Symp. Theory of Computing*, San Diego, CA, U.S.A. (1978) 114-118.
[31] M. Furer, Generalized iterative arrays. Technical Report TR 79-07-03 CS, Univ. of Washington, Seattle, WA, U.S.A., 1979.
[32] H. Gazit, An optimal randomized parallel algorithm for finding connected components in a graph, in: *Proc. 27th Ann. Symp. on Foundations of Computer Science*, Toronto, Canada (1986) 492-501.
[33] A.V. Goldberg, S.A. Plotkin and P. M. Vaidya, Sublinear-time parallel algorithms for matching and related problems, in: *29th Ann. Symp. on Foundations of Computer Science*, White Plains, New York (1988) 432-443.
[34] L. Goldschlager, A unified approach to models of synchronous parallel machines, in: *Proc. 10th ACM Symp. on Theory of Computing*, San Diego, CA, U.S.A. (1978) 89-94.
[35] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer—designing an MIMD parallel machine, *IEEE Trans. Comput.* **32** (1983) 175-189.
[36] A. Gottlieb and C.P. Kruskal, Complexity results for permuting data and other computations on parallel processors, *J. Assoc. Comput. Mach.* **31** (1984) 193-209.
[37] J.L. Gustafson, G.R. Montry and R.E. Benner, Development of parallel methods for a 1024-processor hypercube, *SIAM J. Sci. Statist. Comput.* **9** (1988) 609-638.
[38] Y. Han, An optimal linked list prefix algorithm on local memory computer TR 111-88, Dept. of Computer Science, Univ. of Kentucky, Lexington, KY, U.S.A., Nov. 1987.
[39] H.J. Hoover, M.M. Klawe an N.J. Pippenger, Bounding fan-out in logical networks, *J. Assoc. Comput. Mach.* **31** (1984) 13-18.
[40] H.J. Hoover and W.L. Ruzzo, A compendium of problems complete for P, Manuscript.
[41] A.K. Karlin and E. Upfal, Parallel hashing—an efficient implementation of shared memory, in: *Proc. 18th ACM Symp. on Theory of Computing*, Berkeley, CA, U.S.A. (1986) 160-168.
[42] R.M. Karp and V. Ramachandran, A survey of parallel algorithms for shared-memory machines. Report no. UCB/CSD 88/408, Computer Science Division (EECS), University of California, Berkeley, CA, U.S.A.
[43] R.M. Karp, E. Upfal and A. Widgerson, The complexity of parallel search, *J. Comput. System Sci.* **36** (1988) 225-253.
[44] R.M. Karp, E. Upfal and A. Widgerson, Constructing a maximum matching is in random NC, *Combinatorica* **6** (1986) 35-48.
[45] R. Koch, T. Leighton, B. Maggs, S. Rao and A. Rosenberg, Work-preserving simulations of fixed-connection networks, Manuscript.

[46] S.R. Kosaraju and A.L. Delcher, Optimal parallel evaluation of tree-structured computations for raking (extended abstract), in: *Proc. 3rd Aegaen Workshop in Computing, AWOC 88*, Corfu, Greece, Lecture Notes in Computer Science **319** (Springer, Berlin, 1988) 101-110.
[47] C.P. Kruskal, Algorithms for replace-add based paracomputers, in: *Proc. Int. Conf. on Parallel Processing*, Michigan, U.S.A. (1982) 219-223.
[48] C.P. Kruskal, Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.* **32** (1983) 942-946.
[49] C.P. Kruskal, Performance bounds on parallel processors: an optimistic view, *Control Flow and Data Flow: Concepts of Distributed Programming*, NATO ASI series, Series F: Computer and System Sciences **14** (Springer, Berlin, 1985) 331-344.
[50] C.P. Kruskal, T. Madej and L. Rudolph, Parallel prefix on fully connected direct connection machine, in: *Proc. Int. Conf. on Parallel Processing*, Illinois, U.S.A. (1986) 278-283.
[51] C.P. Kruskal, L. Rudolph and M. Snir, Efficient synchronization in multiprocessors with shared memory, *ACM Trans. Programming Languages and Systems* **10** (1988) 579-601.
[52] C.P. Kruskal, L. Rudolph and M. Snir, Efficient parallel algorithms for graph problems, in: *Proc. Int. Conf. on Parallel Processing*, Illinois, U.S.A. (1986) 869-876.
[53] L. Kucera, Parallel computation and conflicts in memory accesses, *Inform. Process. Lett.* **14** (1982) 93-96.
[54] D.J. Kuck, *The Structure of Computers and Computations* (John Wiley, New York, 1978).
[55] T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Trans. Comput.* **34** (1985) 965-968.
[56] G. Lev, N. Pippenger and L. G. Valiant, A fast parallel algorithm for routing in permutation networks, *IEEE Trans. Comput.* **30** (1981) 93-100.
[57] N.A. Lynch and M.J. Fisher, On describing the behavior an implementation of distributed systems, *Theoret. Comput. Sci.* **13** (1981) 17-43.
[58] L. Meertens, Recurrent Ultracomputers are not log N-Fast, Ultracomputer Note 2, New York University, New York, NY, U.S.A., March 1979.
[59] K. Mehlhorn and U. Vishkin, Randomized and deterministic simulations of PRAMs by parallel machines with restricted granualarity of parallel memories, *Acta Inform.* **21** (1984) 339-374.
[60] K. Mulmuley, U.V. Vazirani and V.V. Vazirani, Matching is as easy as matrix inversion, *Combinatorica* **7** (1987) 105-114.
[61] D. Nassimi and S. Sahni, Data broadcasting in SIMD computers, *IEEE Trans. Comput.* **30** (1981) 101-107.
[62] M. C. Pease, An adaptation of the fast Fourier transform for parallel processing, *J. Assoc. Comput. Mach.* **15** (1968) 252-264.
[63] N. Pippenger, Sorting and selecting in rounds, *SIAM J. Comput.* **6** (1986) 1032-1038.
[64] A.G. Ranade, How to emulate shared memory, in: *Proc. 28th Ann. Symp. on Foundations of Computer Science*, Los Angeles, California (1987) 185-194.
[65] J.H. Reif, Depth-first search is inherently sequential, *Inform. Process. Lett.* **20** (1985) 229-234.
[66] J.H. Reif, An optimal parallel algorithm for integer sorting, in: *Proc. 26th Ann. Symp. on Foundations of Computer Science*, Portland, Oregon (1985) 496-504.
[67] L. Rudolph, Software structures for ultraparallel computing, Ph.D. Thesis, New York University, New York, NY, U.S.A., 1982.
[68] J. Schwartz, Ultracomputers, *ACM Trans. Programming Languages and Systems* **2** (1980) 484-521.
[69] Y. Shiloach and U. Vishkin, An O(log *n*) parallel connectivity algorithm, *J. Algorithms* **3** (1982) 57-67.
[70] M. Snir, On parallel searching, *SIAM J. Comput.* **14** (1985) 688-708.
[71] M. Snir, Asynchronous parallel computations, Manuscript.
[72] M. Snir, Size depth trade-offs for monotone arithmetic circuits, IBM Tech. Rep. RC 13742, IBM Research Division, Yorktown Heights, NY, U.S.A., May 1988.
[73] E. Upfal and A. Wigderson, How to share memory in a distributed system, *J. Assoc. Comput. Mach.* **34** (1987) 116-127.
[74] L.G. Valiant, Parallelism in comparison problems, *SIAM J. Comput.* **4** (1975) 348-355.
[75] L.G. Valiant, Structures for highly parallel computing, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. A* (North-Holland, Amsterdam, 1990) to appear.
[76] L.G. Valiant, S. Skyum, S. Berkowitz and C. Rackoff, Fast parallel computation of polynomials using few processors, *SIAM J. Comput.* **12** (1983) 641-644.

[77] J.S. Vitter and R.A. Simons, New classes for parallel complexity: a study of unification and other complete problems for P, *IEEE Trans. Comput.* **35** (1986) 403-418.
[78] J. Vuillemin, A combinatorial limit to the computing power of VLSI circuits, in: *Proc. 21st Ann. Symp. on Foundations of Computer Science*, Buffalo, New York (1980) 294-300.
[79] U. Vishkin and A. Widgerson, Dynamic parallel memories, *Inform. and Control* **56** (1983) 174-182.
[80] R.A. Wagner and Y. Han, Parallel algorithms for bucket sorting and the data dependent prefix problem, in: *Proc. Int. Conf. on Parallel Processing*, Illinois, U.S.A. (1986) 924-930.
[81] R.A. Wagner and Y. Han, An efficient and fast parallel connected component algorithm, Manuscript, Duke University, Raleigh, NC, U.S.A., 1986.