PRO: A Model for the Design and Analysis of Efficient and Scalable Parallel Algorithms

Assefaw Hadish Gebremedhin, Jens Gustedt, Mohamed Essaïdi, Isabelle Guérin Lassous, Jan Arne Telle

PRO: A Model for the Design and Analysis of Efficient and Scalable Parallel Algorithms

Assefaw Hadish Gebremedhin
Old Dominion University, USA

Jens Gustedt
INRIA Lorraine & LORIA, France

Mohamed Essaïdi
INRIA Sophia-Antipolis, France

Isabelle Guérin Lassous
UFR Informatique / UCBL
LIP / ENS Lyon, France

Jan Arne Telle
University of Bergen, Norway

Abstract. We present a new parallel computation model called the Parallel Resource-Optimal computation model. PRO is a framework being proposed to enable the *design* of efficient and scalable parallel algorithms in an architecture-independent manner, and to simplify the *analysis* of such algorithms. A focus on three key features distinguishes PRO from existing parallel computation models. First, the design and analysis of a parallel algorithm in the PRO model is performed relative to the time and space complexity of a specific sequential algorithm. Second, a PRO algorithm is required to be both time- and space-optimal relative to the reference sequential algorithm. Third, the quality of a PRO algorithm is measured by the maximum number of processors that can be employed while optimality is maintained. Inspired by the Bulk Synchronous Parallel model, an algorithm in the PRO model is organized as a sequence of *supersteps*. Each superstep consists of distinct computation and communication phases, but the supersteps are not required to be separated by synchronization barriers. Both computation and communication costs are accounted for in the runtime analysis of a PRO algorithm. Experimental results on parallel algorithms designed using the PRO model—and implemented using its accompanying programming environment SSCRAP—demonstrate that the model indeed delivers efficient and scalable implementations on a wide range of platforms.

Key words: Parallel computers, Parallel computation models, Parallel algorithms, Complexity analysis

1. Introduction

As Akl [1997] notes, a model of computation should ideally serve two major purposes. First, it should describe a computer. In this role, a model should attempt to capture the essential features of an existing or contemplated machine while ignoring less important details of its implementation. Second, it should serve as a tool for analyzing problems and expressing algorithms. In this sense, a model need not necessarily be linked to a real computer but rather to an understanding of computation.

In the realm of sequential computation, the Random Access Machine (RAM) has been a standard model for many years, successfully achieving both of these purposes. It has served as an effective model for hardware designers, algorithm developers, and programmers alike. Only recently has the focus on external memory and cache issues uncovered a need for more refined models. When it comes to parallel computation, there has not been an analogous, universally accepted model that has been as successful. This is in part due to the complex set of issues inherent in parallel computation.

The performance of a sequential algorithm is adequately evaluated using its execution time, one of the reasons that made the RAM powerful enough for algorithm analysis and design. On the other hand, the performance evaluation of a parallel algorithm involves several metrics. Perhaps the most important metrics are *speedup*, *optimality* (or *efficiency*), and *scalability*. To enjoy similar success as that of the RAM, a parallel computation model should incorporate at least these metrics and be simple to use at the same time. In order to simplify the design and analysis of *resource-optimal*, *scalable*, and *portable* parallel algorithms, we propose the Parallel Resource-Optimal (PRO) computation model. The PRO model was briefly introduced in the conference paper Gebremedhin *et al.* [2002]. Here we describe the model in detail, and provide experimental results to help validate the model and demonstrate its practical relevance.

The PRO model is inspired by the Bulk Synchronous Parallel (BSP) model introduced by Valiant [1990] and the Coarse Grained Multicomputer (CGM) model of Dehne *et al.* [1996]. In the BSP model a parallel algorithm is organized as a sequence of *supersteps* with distinct computation and communication phases. The emergence of the BSP model marked an important milestone in parallel computation. The model introduced a desirable structure to parallel programming, and was accompanied by the definition and implementation of communication infrastructure libraries due to Bonorden *et al.* [1999] and Hill *et al.* [1998]. Recently, Bisseling [2004] has written a textbook on scientific parallel computation using the BSP model. From an algorithmic (as opposed to a programming) point of view, we believe that the relatively many and machine-specific parameters involved in the BSP model make the design and analysis of algorithms somewhat cumbersome. The CGM model partially addresses this limitation as it involves only two parameters, the input size and the number of processors. The CGM model is a specialization of the BSP model in that the communication phase of a superstep is required to consist of single long messages rather than multiple short ones. A drawback of the CGM model is the lack of an *accurate* performance measure; the number of communication rounds (supersteps) is usually used as a quality measure, but as we shall see later in this paper, this measure is sometimes inaccurate.

The PRO model inherits the advantages offered by the BSP and the CGM models. It also reflects a compromise between further theoretical and practical considerations in the design of optimal and scalable parallel algorithms. The model focuses on three key features, a fact that distinguishes PRO from existing parallel computation models. The foci of the model are: *relativity*, *resource-optimality*, and a new quality measure referred to as *granularity*.

Relativity pertains to the fact that the design and analysis of a parallel algorithm in PRO is done relative to the time and space complexity of a *specific* sequential algorithm. As a consequence, the parameters involved in the analysis of a PRO-algorithm are: the number of processors p, the input size n, and the time and space complexity of the reference sequential algorithm $\mathcal{A}$. Note that speedup and optimality are metrics that are relative in nature as they are expressed with respect to some sequential algorithm, and this forms the major reason for the focus on relativity. The notion of relativity is also relevant from a practical point of view, since a parallel algorithm is usually designed not from scratch, but rather starting from a sequential algorithm.

A PRO-algorithm is required to be time- and space-optimal, hence resource-optimal, with respect to the reference sequential algorithm. A parallel algorithm is said to be time- (or work-) optimal if the overall *computation and communication* cost involved in the algorithm is proportional to the time complexity of the sequential algorithm used as a reference. Similarly, it is said to be space-optimal if the overall memory space used by the algorithm is of the same order as the memory usage of the underlying sequential version. As a consequence of its time-optimality, a PRO-algorithm always yields *linear speedup* relative to the reference sequential algorithm. In other words, the ratio between the sequential and the parallel runtime is a linear function of the number of processors p. The resource optimality requirement set in the PRO model enables one to concentrate only on practically useful parallel algorithms. Here optimality is required only in an asymptotic sense, which leaves enough slackness for easy design and analysis of algorithms.

Before turning to the quality measure of a PRO algorithm, we wish to underscore the consequences of the novel notion of relativity. In PRO, instead of directly comparing algorithms that solve the same problem, a two-leveled approach is taken. First, a reference sequential algorithm $\mathcal{A}$ with a particular space and time complexity is selected. Then, parallel algorithms that are resource-optimal with respect to $\mathcal{A}$ are compared. The latter comparison, the quality of a PRO algorithm, is measured by the range of values the parameter p can assume while linear speedup is maintained. It is captured by an attribute of the model called the granularity function $\mathrm{Grain}(n)$. In particular, a PRO-algorithm with granularity $\mathrm{Grain}(n)$ is required to yield linear speedup for all values of p such that $p = O(\mathrm{Grain}(n))$. In other words, the algorithm is required to be fully *scalable* for $p = O(\mathrm{Grain}(n))$. The higher the function value $\mathrm{Grain}(n)$, the better the algorithm. The final evaluation of a PRO-algorithm for a given problem must of course take into account both the time and the space complexity of the reference sequential algorithm and the granularity function. A new result will typically be presented as follows: *Problem* Π *has a PRO-algorithm with* $Grain(n) = g(n)$ *relative to a sequential algorithm* $\mathcal{A}$ *with time complexity* $T_{\mathcal{A}}(n)$ *and space complexity* $S_{\mathcal{A}}(n)$. This simply means that for every number of processors p and input size n with $p = O(g(n))$, there is a parallel algorithm in the PRO model for problem Π where the parallel runtime is $O(\mathrm{T}_{\mathcal{A}}(n)/p)$ and each processor uses $O(\mathrm{S}_{\mathcal{A}}(n)/p)$ memory.

In addition to describing—and arguing for the need for—the PRO model, a twin goal of this paper is to provide experimental evidence to help validate the model. To this end, we present results on parallel algorithms for the list ranking and sorting problems designed using the PRO model. These algorithms are implemented using SSCRAP, a C++ communication infrastructure library for implementing BSP-like parallel algorithms, developed by Essaïdi *et al.* [2002, 2004]. The experiments are run on several platforms, including an SGI Origin 3000 parallel computer and a PC cluster. The obtained results show that designing algorithms within the framework of the PRO model indeed offers linear speedup and a high degree of scalability across a variety of platforms.

The rest of the paper is organized as follows. In Section 2 we highlight the limitations of a few relevant existing parallel computation models, to help justify the need for the introduction of the new model PRO. In Section 3 the PRO model is presented in detail, and in Section 4 it is systematically compared with a selection of existing parallel computation models. In Section 5 we illustrate how the PRO model is used in the design and analysis of algorithms using three examples: matrix multiplication, list ranking and sorting. The latter two algorithms are used in our experiments, the setting and the results of which are discussed in Section 6. We conclude the paper in Section 7 with some remarks.

2. Existing models and their limitations

There exists a plethora of parallel computation models in the literature. Our brief discussion in this section focuses on just three of them, the Parallel Random Access Machine (PRAM), the BSP, and the CGM; we will also in passing mention a few other models. The PRAM is discussed not to reiterate its failure to capture real machine characteristics but rather to point out its limitations even as a theoretical model. The BSP and CGM models are discussed because the PRO model is derived from them. The models discussed in this section are in a loose sense divided into two groups as 'dedicated models' (to either software or hardware) and 'bridging models' (between software and hardware).

2.1 Dedicated models

2.1.1 The PRAM family of models

In its standard form, the PRAM model [Fortune and Wyllie 1978] consists of an arbitrarily large number of processors and a shared memory of unbounded size that is uniformly accessible to all processors. In this model, processors share a common clock and operate in lock-step, but they may execute different instructions in each cycle.

The PRAM is a model for *fine-grain* parallel computation as it supposes that the number of processors can be arbitrarily large. Usually, it is assumed that the number of processors is polynomial in the input size. However, practical parallel computation is typically *coarse-grain*. In particular, on most existing parallel machines, the number of processors is several orders of magnitude less than the input size. Moreover, the assumption that memory is uniformly accessible to all processors is in obvious disagreement with the reality of practical parallel computers.

Despite its serious limitation of being an 'idealized' model for parallel computation, the standard PRAM model still serves as a theoretical framework for investigating the maximum possible computational parallelism available in a given task. Specifically, on this model, the *NC* versus *P*-complete dichotomy [Greenlaw *et al.* 1995] is used to reflect the ease/hardness of finding a parallel algorithm for a problem.

Unfortunately, the *NC* versus *P*-complete dichotomy has several limitations. First, *P*-completeness does not depict a full picture of non-parallelizability since the runtime requirement for an *NC* parallel algorithm is so stringent that the classification is confined to the case where up to polynomial number of processors in the input size is available. For example, there are *P*-complete problems for which less ambitious, but still satisfactory, runtime can be obtained by parallelization in PRAM, see for example Vitter and Simons [1986]. In a fine-grained setting, since the number of processors p is a function of the input size n, it is customary to express speedup as a function of n. Thus the speedup obtained using an *NC*-algorithm is sometimes referred to as exponential. In a coarse-grained setting, speedup is expressed as a function of only p and some recent results show that this approach is practically relevant [Caceres *et al.* 1997, Dehne *et al.* 1996, Gebremedhin *et al.* 2003, Guérin Lassous *et al.* 2000].

Second, an *NC*-algorithm is not necessarily work-optimal (thus not resource-optimal), considering runtime and memory space as resources one wants to use efficiently.

Third, even if we consider only work-optimal *NC*-algorithms and apply the scheduling principle due to Brent [1974], which says a parallel algorithm in theory can be simulated on a machine with fewer processors by only a constant factor more work, implementations of PRAM algorithms often do not reflect this optimality in practice, see for instance Dehne [1999]. This is mainly because the PRAM model does not account for non-local memory access (communication), and a Brent-type simulation relies heavily on cheap communication.

To overcome the defects of the PRAM related to its failure of capturing real machine characteristics, the advocates of shared memory models propose several modifications to the standard PRAM model. In particular, they enhance the standard PRAM model by taking practical machine features such as memory access, synchronization, latency and bandwidth issues into account. Pointers to the PRAM family of models can be found in Maggs *et al.* [1995].

2.1.2 Distributed memory models

Critics of shared memory models argue that the PRAM family of models fails to capture the nature of existing parallel computers with *distributed* memory architectures. Examples of distributed memory computational models suggested as alternatives include the Postal Model [Bar-Noy and Kipnis 1992] and the Block Distributed Memory (BDM) model [JáJá and Ryu 1996]. Other categories of parallel models such as low-level, hierarchical memory, and network models are briefly reviewed in Maggs *et al.* [1995].

These models are very close to the architecture considered and the associated algorithms are often not portable from one architecture to another.

2.2 Bridging models

In a seminal work, Valiant [1990] underscored that a successful parallel computation model needs to act as an efficient 'bridge' between software and hardware. He introduced the Bulk Synchronous Parallel model as a candidate bridging model, and argued that it could serve as a standard model for parallel computation.

2.2.1 The Bulk Synchronous Parallel model

The BSP model consists of a collection of processor/memory modules connected by a router that can deliver messages in a point-to-point fashion. An algorithm in this model is divided into a sequence of *supersteps* separated by synchronization barriers. A superstep has distinct computation and communication phases. In a superstep, a processor may send (and receive) at most h messages. Such a communication pattern is called an *h-relation* and the basic task of the router is to realize arbitrary h-relations. The quantity h here is related to the total size of communicated data during a superstep.

The BSP model uses the four parameters, n, p, L, and g. Parameter n is the problem size, p is the number of processors, L is the minimum time between successive synchronization operations, and g is the ratio of overall system computational capacity per unit time divided by the overall system communication capacity per unit time.

The introduction of the BSP model initiated several subsequent studies suggesting various modifications. For example, Culler *et al.* [1993] proposed a model that extends the BSP model by allowing asynchronous execution and by better accounting for communication overhead. Their model is coined LogP, an acronym for the four parameters (besides the problem size n) involved. Models such as LogP involve many parameters making design and analysis of algorithms difficult. Analysis using the BSP model is not as difficult, but still not as simple as it could be. In fact, to simplify analysis while using the BSP model, one often neglects the latency L for problems of large enough size. Ideally, for the design of portable algorithms, it is important to abstract away specific architectural parameters. This issue is well-captured in the PRO model.

2.2.2 The Coarse-Grained Multicomputer model

The CGM model [Caceres *et al.* 1997] was proposed in an effort to retain the advantages of the BSP model while simultaneously aiming at simplicity. The CGM model consists of p processors, each with $O(n/p)$ local memory, interconnected by a router that can deliver messages in a point-to-point fashion. A CGM algorithm consists of an alternating sequence of *computation rounds* and *communication rounds* separated by synchronization barriers. A computation round is equivalent to the computation phase of a superstep in the BSP model. A communication round usually consists of a single h-relation with

$$h \approx n/p. \tag{1}$$

An important advantage of the CGM model compared to BSP is that all the information sent from one processor to another in one communication round is packed into one long message, striving to minimize communication overhead and latency. Thus, the only parameters involved in the CGM model are p and n, a fact that simplifies design and analysis of algorithms.

The assumption captured by Equation (1) has interesting implications on the design and analysis of algorithms. To make these implications more apparent, we first distinguish between parallel algorithms where the communication time to computation time ratio is a constant and those algorithms where the ratio is some function of the input size.

Suppose we have a CGM algorithm where the communication time to computation time ratio is a constant. Suppose also that Equation (1) holds. Then, since each superstep has a complexity of $\Theta(h) = \Theta(n/p)$, the only parameter of the model that distinguishes one algorithm from another is the number of supersteps. This direction was followed for instance by Caceres *et al.* [1997] where a long list of algorithms that are designed under these assumptions is given.

However, there exists a large class of problems for which there are no known CGM algorithms with constant communication time to computation time ratio. Problems with super-linear time sequential algorithms, such as sorting and matrix multiplication, belong to this class. For such problems and their corresponding parallel algorithms, communication alone cannot be a complexity measure and hence one needs to consider computation as well. Furthermore, even for problems whose algorithms are such that the stated ratio is constant, the assumption in Equation (1) turns out to be quite restrictive. We shall illustrate this in Section 5.2 using the *list ranking* problem as an example. In particular, we will show that the CGM model fails to identify competitive algorithms when using the number of supersteps as a quality measure.

3. The PRO model definition

The PRO model is an algorithm *design* and *analysis tool* used to deliver a practical, optimal, and scalable parallel algorithm relative to a specific sequential algorithm whenever this is possible. Let $\mathrm{T}_{\mathcal{A}}(n)$ and $\mathrm{S}_{\mathcal{A}}(n)$ denote the time and space complexity of a specific sequential algorithm $\mathcal{A}$ for a given problem with input size n. Let $\mathrm{Grain}(n)$ be a function of n. The PRO model is defined to have the attributes given in Table I. In the following we will argue for each of these attributes turn by turn.

Attribute A states that optimality in PRO is a relative notion. Thus in PRO we could speak of an optimal parallel algorithm for a problem even if an optimal sequential algorithm for the problem is unknown. We will illustrate this point using the matrix multiplication problem as an example in Section 5. An implication of attribute A is that PRO does not define a complexity class.

As discussed in the LogP paper by Culler *et al.* [1993], technological factors are forcing parallel systems to converge towards systems formed by a collection of essentially complete computers connected by a robust communication network. The *machine* model assumption of PRO (attribute B) is consistent with this convergence and maps well on several existing parallel computer architectures. The memory requirement $M = O(\frac{\mathrm{S}_{\mathcal{A}}(n)}{p})$ ensures that the space utilized by the underlying sequential algorithm is uniformly distributed among the p processors. Since we may, without loss of generality, assume that $\mathrm{S}_{\mathcal{A}}(n) = \Omega(n)$, the implication is that the private memory of each processor is large enough to store its 'share' of the input and any additional space the sequential algorithm might require. When $\mathrm{S}_{\mathcal{A}}(n) = \Theta(n)$, the input data needs to be uniformly distributed among the p processors. In this case the machine model assumption of PRO is similar to the assumption in the CGM model of Dehne *et al.* [1996].

The *coarseness* assumption $p \leq M$ (attribute C) is consistent with the structure of existing parallel machines and those to be built in the foreseeable future. The assumption

TABLE I: Attributes of the PRO Model.

A. *Relativity:*

The time and space requirements of a PRO-algorithm for a problem (of input size n) are measured relative to the time and space requirements $\mathrm{T}_{\mathcal{A}}(n)$ and $\mathrm{S}_{\mathcal{A}}(n)$ of a specific sequential algorithm $\mathcal{A}$ that solves the same problem.

B. *Machine Model:*

The underlying machine is assumed to consist of p processors each of which has a private memory of size $M = O(\frac{\mathrm{S}_{\mathcal{A}}(n)}{p})$. The processors are assumed to be interconnected by some communication device (such as an interconnection network or a shared memory) that can deliver messages in a point-to-point fashion. A message can consist of several machine words.

C. *Coarseness Assumption:*

The size of the local memory of each processor is assumed to be big enough to store p words. That is, the relationship $p \leq M$ is assumed to hold.

D. *Execution Model:*

A PRO algorithm is organized as a sequence of *supersteps,* each consisting of a local computation phase and an interprocessor communication phase. In particular, in each superstep, each processor

D.1. sends at most one message to every other processor,
D.2. sends and receives at most M words in total, and
D.3. performs local computation.

E. *Runtime Analysis:*

Both *computation* and *communication* are accounted for in the runtime analysis of a PRO algorithm. In particular,

E.1. a processor is charged a unit of time per operation performed locally, and
E.2. a processor is charged a unit of time per machine word sent or received.

F. *Optimality Requirement:*

For every value $p = O(\mathrm{Grain}(n))$, a PRO algorithm is required to have

F.1. a number of supersteps $\mathrm{Steps}(n, p) = O(\frac{\mathrm{T}_{\mathcal{A}}(n)}{p^2})$, and
F.2. a parallel runtime $\mathrm{T}(n, p) = O(\frac{\mathrm{T}_{\mathcal{A}}(n)}{p})$.

G. *Quality Measure:*

The *granularity* function $\mathrm{Grain}(n)$ measures the *quality* of the algorithm.

is required to simplify the implementation of gathering messages on or broadcasting messages from a single processor.

In terms of *execution*, a PRO-algorithm consists of a sequence of *supersteps* (see attribute D). A superstep has distinct local computation and inter-processor communication phases. The *length* of a superstep on each processor is determined by the sum of the time used for communication and the time used for local computation (see attributes E1 and E2). The length of a superstep in the parallel algorithm seen as a whole is the maximum over the lengths of the superstep on all processors. The parallel runtime $\mathrm{T}(n, p)$ of the algorithm is the sum of the lengths of all the supersteps.

Conceptually, we can think of the supersteps as being synchronized by a barrier set at the end of the longest superstep across the processors. In reality, however, PRO does not require the processors to be synchronized at the end of each superstep. The assumption made instead is that prior to a computation phase of a superstep, all the messages a processor awaits for has been completely received. This way the processors are 'soft synchronized' via communication, and processors may thus differ in a maximum of two supersteps. Hence the hypothetical barriers introduce only a multiplicative factor of two in comparison with an analysis that does not assume the barriers.

In PRO, since a processor sends at most one message to every other processor in each superstep (attribute D1), each processor is involved in at most $2(p-1)$ messages per superstep. Hence the total amount of messages that a PRO-algorithm will be involved in is at most $2(p-1)\cdot\mathrm{Steps}(n, p)$. The overall contribution of *latency* is thus at most $2l(p-1)\cdot\mathrm{Steps}(n, p)$, where l is an appropriate constant capturing the network latency. Therefore, the requirement that the number of supersteps be bounded by $O(\frac{\mathrm{T}_{\mathcal{A}}(n)}{p^2})$ (attribute F1) implies that the overall time paid per processor for latency is $O(\mathrm{T}_{\mathcal{A}}(n)/p)$ and hence within the same bound as the parallel runtime $O(\mathrm{T}_{\mathcal{A}}(n)/p)$ we would like to achieve. Note that in the extreme case where

$$\mathrm{Steps}(n, p) = \Theta(\frac{\mathrm{T}_{\mathcal{A}}(n)}{p^2}), \tag{2}$$

latency could be the practically dominant term in the overall parallel runtime expression. Since latency is a parameter primarily determined by physical restrictions such as the speed of light—and hence unlikely to improve as other architectural features—an algorithm with such a number of supersteps may no longer be portable across architectures.

The *bandwidth* of an architecture, on the other hand, is not subject to such physical restrictions. Computation and communication contribute to the overall runtime in fairly similar ways as far as bandwidth is concerned. To account for bandwidth restrictions of an architecture, in PRO, each processor is charged a unit of time per word sent and received (attribute E2). This assumption is fairly realistic since the network throughput on modern architectures such as high performance clusters is quite close to the CPU frequency and to the CPU/memory bandwidth.

These properties imply that the BSP-cost of a PRO algorithm is proportional to the following expression:

$$g \cdot \mathrm{T}(n, p) + L \cdot \mathrm{Steps}(n, p), \tag{3}$$

where g and L are the BSP parameters defined in Section 2.2; the parameter L accounts for the sum of latency l and the cost for synchronization.

The condition $\mathrm{T}(n, p) = O(\frac{\mathrm{T}_{\mathcal{A}}(n)}{p})$ in attribute F2 of the PRO model *requires* that a PRO-algorithm be optimal. That is to say a PRO-algorithm is required to yield linear speedup relative to the sequential algorithm used as a reference. This requirement ensures the potential practical use of the parallel algorithm. Except for the extreme case represented by the expression (2), in general, the first term of the expression (3) is the dominant term. Hence latency can be neglected in the analysis of a PRO-algorithm.

The function Grain(n) is a *quality measure* for a PRO algorithm (attribute G). In particular, for every number of processors p such that $p = O(\text{Grain}(n))$, a PRO algorithm gives a linear speedup with respect to the reference sequential algorithm. One of the objectives in designing a PRO algorithm is to make Grain(n) as high as possible so as to increase scalability. As the following observation shows, there is an upper bound on Grain(n) set by the complexity of the reference sequential algorithm.

Observation 1. A PRO algorithm relative to a sequential algorithm $\mathcal{A}$ with time complexity $\mathrm{T}_{\mathcal{A}}(n)$ and space complexity $\mathrm{S}_{\mathcal{A}}(n)$ has maximum granularity

$$\text{Grain}(n) = O\left(\sqrt{\mathrm{S}_{\mathcal{A}}(n)}\right).$$

A PRO algorithm that achieves this is said to have *optimal grain*.

Proof. One arrives at this result from two different sets of PRO-attributes. The bound $M = O(\frac{\mathrm{S}_{\mathcal{A}}(n)}{p})$ on the size of the private memory of each processor (attribute B) and the coarseness assumption $p \leq M$ (attribute C) taken together imply the bound $p = O(\sqrt{\mathrm{S}_{\mathcal{A}}(n)})$. Further, the requirement Steps $= O(\mathrm{T}_{\mathcal{A}}(n)/p^2)$ on the number of supersteps of a PRO-algorithm (attribute F1) gives the expression $p = O(\sqrt{(\mathrm{T}_{\mathcal{A}}(n)/\text{Steps})})$ upon resolving, and since Steps ≥ 1 holds, the expression reduces to $p = O(\sqrt{\mathrm{T}_{\mathcal{A}}(n)})$. Moreover, since we may reasonably assume that all memory is initialized, the inequality $\mathrm{T}_{\mathcal{A}}(n) \geq \mathrm{S}_{\mathcal{A}}(n)$ holds. Thus the bound $p = O(\sqrt{\mathrm{S}_{\mathcal{A}}(n)})$ set by attributes B and C is more restrictive and the result follows. □

Since a PRO-algorithm yields linear speedup for every $p = O(\text{Grain}(n))$, a result like Brent's scheduling principle (discussed in Section 2.1.1) is implicit for these values of p. But Observation 1 shows that we cannot start with an arbitrary number of processors and efficiently simulate on fewer processors. So Brent's scheduling principle does not hold with full generality in the PRO model, which is in accordance with practical observations.

The design of a PRO-algorithm may sometimes involve subroutines for which no natural sequential counterparts exist. Examples of such tasks include communication primitives such as broadcasting, data (re)-distribution routines, and load balancing routines. Such routines are often required in various parallel algorithms. With a slight abuse of terminology, we call a parallel algorithm for one of such routines a PRO-algorithm if the overall computation and communication cost is linear in the input size to the routine.

4. Comparison with other models

In this section we compare the PRO model with the PRAM, BSP, LogP, CGM, and the Queuing Shared Memory (QSM, Gibbons *et al.* [1999]) models. The QSM model is interesting since it is a shared memory model based on some BSP principles. Our tabular format for comparison is inspired by a similar presentation in Gibbons *et al.* [1999]. The columns of Table II are labeled with names of models and some relevant features of a model are listed along the rows.

The synchrony assumption of a model is indicated in the row labeled *synch.* Lock-step indicates that processors are fully synchronized at each step (of a universal clock), without accounting for synchronization. Bulk-synchrony indicates that there can be asynchronous operations between synchronization barriers. The row labeled *memory* shows how a model views the memory of the parallel computer: 'sh.' indicates globally accessible shared memory, 'dist.' stands for distributed memory and 'priv.' is an abstraction for the case where the only assumption is that each processor has access to private (local) memory. In the last variant the whole memory could be either distributed or shared. The row labeled

TABLE II: Comparison of parallel computational models.

	PRAM	QSM	BSP	LogP	CGM	PRO
synch.	lock-step	bulk-synch.	bulk-synch.	asynch.	asynch.	asynch.
memory	sh.	sh.	dist.	dist.	priv.	priv.
commun.	SM	SM	MP	MP	MP/SM	MP/SM
parameters	n	p, g, n	p, g, L, n	p, g, l, o, n	p, n	$p, n, \mathcal{A}$
granularity	fine	fine	coarse	fine	coarse	Grain(n)
speedup	NA	NA	NA	NA	NA	$\Theta(p)$
optimal	NA	NA	NA	NA	NA	rel. $\mathcal{A}$
quality	time	time	time	time	rounds	Grain(n)

commun. shows the type of interprocessor communication assumed by a model. Shared memory (SM) indicates that communication is effected by reading from and writing to a globally accessible shared memory. Message-passing (MP) denotes the situation where processors communicate by explicitly exchanging messages in a point-to-point fashion. The MP abstraction hides the details of how the message is routed through the interprocessor communication network.

The parameters involved in a model are indicated in the row labeled *parameters*. The number of processors is denoted by p, n is the input size, $\mathcal{A}$ is the reference sequential algorithm, l is the communication cost (latency), L is a single parameter that accounts for the sum of latency (l) and the cost for a barrier synchronization, *i.e.* the minimum time between successive synchronization operations, g is the bandwidth gap, and o is the overhead associated with sending or receiving a message. Note that the machine characteristics l and o are taken into account in PRO, even though they are not explicitly used as parameters. Latency is taken into consideration since the length of a superstep is determined by the sum of the computational and communication costs. Communication overhead is hidden by the PRO-requirement that number of supersteps is bounded by $O(\frac{\mathrm{T}_{\mathcal{A}}(n)}{p^2})$.

The row labeled *granularity* indicates whether a model is fine-grained, is coarse-grained, or uses a more precise measure. A model is coarse-grained if it applies to the case where $n \gg p$. A model is fine-grained if it relies on using up to a polynomial number of processors in the input size. In PRO, granularity is a quality measure, captured by the model attribute Grain(n).

The rows labeled *speedup* and *optimal* indicate the speedup and resource optimality requirements imposed by a model. Whenever these issues are not directly addressed by the model or are not applicable, the word 'NA' is used. Note that these requirements are 'hard-wired' in the model in the case of PRO. The label 'rel.$\mathcal{A}$' means that the parallel algorithm is optimal relative to the time and space complexity of the sequential algorithm $\mathcal{A}$. We point out that the goal in the design of algorithms using the CGM model—such as in the works of Caceres *et al.* [1997] and Dehne *et al.* [1996]—is often stated as that of achieving optimal algorithms, but the model *per se* does not impose an optimality requirement.

The last row indicates the *quality* measure of an algorithm designed using the different models. For every model except CGM and PRO, the quality measure is runtime. In CGM, the number of supersteps (rounds) is usually presented as a quality measure. In PRO the quality measure is granularity, one of the features that makes PRO fundamentally different from all existing parallel computation models.

5. Algorithm design in PRO

In this section, using three examples, we illustrate how the PRO model is used to design efficient parallel algorithms. In each example, we start with a specific sequential time and space complexity, and then design and analyze a parallel algorithm relative to these.

Our first example is the standard matrix multiplication algorithm with three nested for-loops. This example is chosen for two reasons: its simplicity and its suitability to emphasize the importance of explicitly stating the sequential time and space complexity against which a parallel algorithm is compared. The complexity of an optimal sequential matrix multiplication algorithm is still unknown, and many algorithms that are theoretically known to be faster than the standard cubic-time algorithm are impractical.

Our second example is list ranking, a basic routine used in many parallel graph algorithms. List ranking is an interesting example in our context as a CGM-analysis of one of its parallel algorithms suggests inefficiency, despite the fact that the algorithm is efficient in practice. The third example is sorting. This example has a known BSP algorithm that also satisfies all of the requirements of the PRO-model. The parallel list ranking and sorting algorithms discussed here will be used in the experimental study reported in the next section.

5.1 Matrix multiplication

Consider the problem of computing the product C of two $m \times m$ matrices A and B (input size $n = m^2$). We want to design a PRO-algorithm relative to the standard sequential matrix multiplication algorithm $\mathcal{M}^3$ which has $\mathrm{T}_{\mathcal{M}^3}(n) = O(n^{\frac{3}{2}})$ and $\mathrm{S}_{\mathcal{M}^3}(n) = O(n)$.

We assume that the input matrices A and B are distributed among the p processors $P_0, ..., P_{p-1}$ so that processor P_i stores *rows* $\frac{m}{p} \cdot i + 1$ to $\frac{m}{p} \cdot (i+1)$ of the matrix A and a similar chunk of *columns* of the matrix B. The output matrix C will be row-partitioned among the p processors in a similar fashion. Note that with this data distribution, each processor can compute a block of $\frac{m^2}{p^2}$ of the $\frac{m^2}{p}$ entries of C expected to reside on it without any communication. In order to compute the next block of $\frac{m^2}{p^2}$ entries, processor P_i needs the columns of matrix B that reside on processor P_{i+1}. Therefore, in each superstep of the PRO algorithm, processors exchange columns in a round-robin fashion and then each processor computes a new block of results. Note that each column exchanged in a superstep constitutes one single message. Note also that the initial distribution of the rows of matrix A remains unchanged. The sequence of computation and communication steps we have sketched is outlined in Algorithm 1 in a manner that meets the requirements of the PRO model.

Algorithm 1 has p supersteps (Steps = p). In each superstep, the time spent in locally computing each of the m^2/p^2 entries is $\Theta(m)$ resulting in local computing time of $\Theta(m^3/p^2) = \Theta(n^{\frac{3}{2}}/p^2)$ per superstep. Likewise, the total size of data (words) exchanged per processor in a superstep is $\Theta(m^2/p) = \Theta(n/p)$. Thus, the length of a superstep σ is $\mathrm{T}_\sigma(n,p) = \Theta(n^{\frac{3}{2}}/p^2 + n/p)$. Note that for $p = O(\sqrt{n})$, $\mathrm{T}_\sigma(n,p) = \Theta(n^{\frac{3}{2}}/p^2)$. Hence, for $p = O(\sqrt{n})$, the overall parallel runtime of the algorithm is

$$\mathrm{T}(n,p) = \sum_{\text{Steps}} \Theta(n^{\frac{3}{2}}/p^2) = \Theta(n^{\frac{3}{2}}/p) = \Theta(\mathrm{T}(n)/p). \tag{4}$$

Since $\mathrm{S}(n) = \Theta(n)$, the memory restriction of the PRO model is respected. That is, each processor has enough memory to handle the transactions. In order to be able to neglect communication overhead, the condition F1 on the number of supersteps, which in this case is just p, should be met. In other words, we need to choose p such that $p = O(\mathrm{T}_{\mathcal{A}}(n)/p^2) =$

Algorithm 1: Matrix multiplication

Input: Two $m \times m$ matrices A and B. The rows of A and the columns of B are divided into m/p contiguous blocks, and stored on processors $P_0, P_1, \ldots P_{p-1}$.
Output: The product matrix C where the rows are stored in contiguous blocks across the p processors.

for *superstep* $s = 1$ *to* p **do**
  foreach *processor* P_i **do**
    P_i computes a local sub-matrix, a product involving the rows that belong to P_i and a current block of columns on P_i;
    $P_{(i+1) \ mod \ p}$ sends its current block of columns to P_i;
    P_i receives a new block of columns from $P_{(i+1) \ mod \ p}$;

$O(n^{\frac{3}{2}}/p^2)$, which is true for $p = O(\sqrt{n})$. The optimality requirement F2 is satisfied as we have already shown in Equation (4). Thus the granularity function of the PRO-algorithm is $\text{Grain}(n) = O(\sqrt{n})$. Note that the sequential reference algorithm $\mathcal{M}^3$ used in our analysis can be replaced by any other algorithm $\mathcal{A}$ that has the same complexity bounds. The following lemma summarizes this result.

LEMMA 1. *Multiplication of two $\sqrt{n} \times \sqrt{n}$ matrices has a PRO-algorithm with $Grain(n) = O(\sqrt{n})$ relative to a sequential algorithm $\mathcal{A}$ with $T_{\mathcal{A}}(n) = O((\sqrt{n})^3)$ and $S_{\mathcal{A}}(n) = O(n)$.*

From Observation 1 we note that Algorithm 1 has optimal grain. On a more relaxed model, where the assumption that $p \leq M$ is not present, the strong regularity of matrix multiplication and the exact knowledge of the communication pattern allow for algorithms that have an even larger granularity than $O(\sqrt{n})$. For example, a systolic matrix multiplication algorithm has a granularity of $O(n)$. However, PRO is intended to be applicable for general problems (including those with irregular communication pattern) and practically relevant parallel systems, hence the result in Lemma 1.

5.2 List Ranking

In the *list ranking* problem (LR) we are given a vector $next[1..n]$ representing a linked list of n elements, and a vector $length[1..n]$ where $length[e]$ stores the 'length' of the link from element e to element $next[e]$. For each element e, the goal is to compute the sum of the lengths of all the links from element e to the end of the list. There is a trivial linear time sequential algorithm for this problem, and designing a good parallel algorithm for it has been a classical question since the early days of parallel computing; see for example Cole and Vishkin [1989], Guérin Lassous and Gustedt [2002], and Sibeyn [1999].

One of the known parallel algorithms for LR uses the technique of *pointer jumping*. This algorithm has $\log n$ phases and is based on the simple observation that in a directed graph where each node (element) e has a single outgoing arc $next[e]$, the diameter of the graph can be halved in a single phase via 'pointer jumping'—by setting $next[e] \leftarrow next[next[e]]$ in parallel for all nodes. Algorithm 2 outlines this approach.

Algorithm 2 can easily be translated into a CGM or a PRO algorithm. Here each phase corresponds to a superstep in which processors exchange their respective values $length[next[e]]$ and $next[next[e]]$. There are of course some details in doing this: Each processor needs to send a request to those processors that store the $next[e]$ indices it needs,

Algorithm 2: List Ranking using Pointer Jumping

Input: A set S of elements; and vectors `next`$[1..n]$ and `length`$[1..n]$ with indices/elements distributed evenly among p processors.

Output: Vector `length`$[1..n]$ with `length`$[e]$ equal to the sum of lengths from element e to the end of the list `next`.

```
for phase s = 1 to log n do
    foreach processor P_i do
        foreach element e that belongs to P_j do
            length[e] ← length[e] + length[next[e]];
            next[e] ← next[next[e]];
```

receive similar requests for values from other processors, send the values requested from it, and finally receive the values it requested. The number of supersteps in this algorithm is $O(\log n)$, or $O(\log p)$ after some refinement. In any case, the number of supersteps reflects a super-linear computation cost for the entire algorithm, which captures very well the fact that this algorithm is not efficient.

There exist other, more sophisticated parallel algorithms for LR that are efficient when compared to the linear-time sequential algorithm. One class of such algorithms relies on computing *independent sets*, and another relies on computing *dominating sets*. The analysis of these classes of algorithms in terms of CGM or PRO is similar. Here we will briefly present an independent set-based algorithm, which we outline in Algorithm 3.

Algorithm 3 is recursive and starts by computing a large independent set I of nodes (having the property that if $i \in I$ then $next[i] \notin I$), for example by a variant of a so-called *random mating* algorithm. This set can be guaranteed to be maximal, of size εn, for $\frac{1}{3} \leq \varepsilon \leq \frac{1}{2}$. For each element i in the set I, the algorithm then computes i's closest successor in the linked list, the value $next_I[i] \in I$, and the accumulated distance from i to that element, $length_I[i]$. When coming back from the recursion with a solution for this intermediate list I, the obtained information can easily be propagated to the elements that are not in I.

The translation of this algorithm to CGM is straightforward and the analysis with CGM is simple. Obviously, the recursion introduces a logarithmic number of supersteps and hence the total processing cost *in terms of the CGM model* is super-linear.

However, the overall work in each recursion level can be made linear in the actual size of the list; hence the work load decreases with every step of the recursion. Since the auxiliary list for the recursion is substantially smaller than the original list (εn), a geometric series argument can be applied to show that the overall resource utilization is linear. Hence, contrary to what a CGM-analysis suggests, this second family of algorithms is in fact efficient. Thus the LR example exhibits a case where a CGM-analysis is not able to distinguish between a "bad" algorithm (Algorithm 2) and a "good" one (Algorithm 3).

The PRO model provides a different view of Algorithm 3. Assuming that the chosen independent set at each recursion level is well balanced among the processors, it is easy to show that $\mathrm{T}(n, p) = \Theta(n/p) = \Theta(\mathrm{T}_{\mathcal{A}}(n)/p)$. In order to be able to neglect communication overhead, we need to meet the condition F1 on the number of supersteps $\log n$. This means we need $\log n = O(\frac{n}{p^2})$, which upon resolving gives $p = O(\sqrt{\frac{n}{\log n}})$. With $\mathrm{S}_{\mathcal{A}}(n) = \Theta(n)$, the PRO memory restriction is respected. The following lemma summarizes these results.

Algorithm 3: Recursive List Ranking

Input: A set S of elements, and vectors `next` and `length` as in Algorithm 2.
Output: The vector `next` as in Algorithm 2.
if *input list size is small* **then** solve by Pointer Jumping;
else
  Find a maximal independent set I in the input list;
  foreach *element* $e \in S$ **do**
    Compute $\texttt{next}_I[e]$, the closest successor of e in I;
    Compute $\texttt{length}_I[e]$, the sum of lengths from e to $\texttt{next}_I[e]$;
  RecursiveListRanking($I, \texttt{next}_I, \texttt{length}_I$);
  Propagate the partial solution for I to elements in the set $S \setminus I$;

LEMMA 2. *List ranking on n elements has a PRO-algorithm with Grain(n) $= O(\sqrt{\frac{n}{\log n}})$ relative to a sequential algorithm $\mathcal{A}$ with $T_{\mathcal{A}}(n) = O(n)$ and $S_{\mathcal{A}}(n) = O(n)$.*

Note that the granularity function in this PRO-algorithm is less than $O(\sqrt{n})$ and thus the granularity is not optimal.

5.3 Sorting

Like list ranking, sorting problems occur frequently in sequential as well as distributed computing. For our experimental studies, we chose the randomized and distributed sorting algorithm described in Gerbessiotis and Valiant [1994] in the context of the BSP model. The algorithm is based on an over-sampling technique, and is outlined in Algorithm 4.

Algorithm 4 starts by computing a small random sample of the items that are to be sorted (phase Φ_1). Then, the random sample is used to determine $p-1$ *splitters* that are approximately equidistant in the set of sorted items (phase Φ_2). The computation of the splitters can be seen as a generalization of the computation of a p-median. In fact, for $p = 2$ the (unique) splitter is expected to be close to a median value. The splitters are then used by the processors to partition their values into p different buckets (phase Φ_3), and to redistribute them to appropriate target processors (phase Φ_4). The algorithm terminates with a parallel local sorting on all processors (phase Φ_5).

The performance of Algorithm 4 depends on the choice of a value k for the size of the local sample that is computed in phase Φ_1. The choice of k has to ensure that the overall sample is a good representative of the input array such that the final share of data for each processor that is to be received in phase Φ_4 and to be sorted in phase Φ_5 is not too large.

To simplify analysis, we choose $k = n/p^2$. For more subtle discussions on the requirements for the choice of k that guarantee a good expected behavior, see Gerbessiotis and Valiant [1994].

We will now look at the complexity of each of the five phases in Algorithm 4.

Φ_1: This phase can be done in $O(m)$ time, where $m = n/p$.

Φ_2: Since the sample size that processor P_0 has to handle is $pk = n/p = m$, the computation cost of this phase is at most $O(\mathrm{T}_{\mathcal{A}}(m))$.

Φ_3: Using binary search to determine the bucket for each element, this phase can be done in $O(m \log p)$ time.

Algorithm 4: Parallel Sorting

Input: A number $0 \leq \rho < p$ identifying this processor, and a distributed array A of some values; A_ρ denotes the local sub-array on this processor.
Output: The array A is globally sorted.

```
begin
Φ1    Randomly extract a sample E_ρ of k values from A_ρ;
      Send the sample E_ρ to Processor P_0;
      if ρ = 0 then
          E ← ⋃_{0≤i<p} E_i;
Φ2        LocalSort(E);
          Let S be an array (of size p) of splitters.
          Initialize S: S[0] ← −∞ and S[p] ← +∞;
          foreach i = 1, ..., p − 1 do S[i] ← E[i · k];
          Broadcast S to all other processors;
      Receive the splitters S from Processor P_0;
      Let M be an array (of size p) of messages.
      Initialize M: for 0 ≤ i < p, M_i ← ∅ ;
Φ3    foreach value v ∈ A_ρ do
          Find ℓ with S[ℓ] ≤ v < S[ℓ + 1];
          M_ℓ ← M_ℓ ∪ {v};
Φ4    foreach i = 0, ..., p − 1 do
          Send local message M_i to Processor P_i;
          Gather messages from relevant processors into M'_i;
Φ5    A_ρ ← ⋃_{0≤i<p} M'_i;
      LocalSort(A_ρ);
end
```

Φ_4: This phase is by far the most expensive task in terms of communication: the initial global array A of size n is completely redistributed through the interconnection network. So this phase accounts for $O(m)$ time.

Φ_5: Assuming that the first three phases provide a balanced redistribution of the initial array ($\Theta(m)$ values per processor), the computation cost of this phase is again $O(\mathrm{T}_{\mathcal{A}}(m))$.

Thus the overall computation cost of Algorithm 4 is $O(\mathrm{T}_{\mathcal{A}}(m) + m \log p)$, and because of the lower bound $\Omega(n \log n)$ on comparison-based sorting, the expression reduces to $O(\mathrm{T}_{\mathcal{A}}(m))$. Since we may also assume that $\mathrm{T}_{\mathcal{A}}(n)$ is a concave-upwards function, we have $\mathrm{T}_{\mathcal{A}}(m) = O(\mathrm{T}_{\mathcal{A}}(n)/p)$ and thus the speedup relative to $\mathcal{A}$ is $\Omega(p)$. The overall communication volume required by the algorithm is bounded by $O(n)$.

Since the number of supersteps of the algorithm is also bounded by a constant, this algorithm fulfills all of the PRO-requirements and in fact has optimal granularity.

Lemma 3. *Sorting of n elements has a PRO-algorithm with Grain$(n) = O(\sqrt{n})$ relative to any comparison-based sequential algorithm $\mathcal{A}$ with $T_{\mathcal{A}}(n) = \Theta(n \log n)$ and $S_{\mathcal{A}}(n) = O(n)$.*

TABLE III: Platforms used in the experiments.

Platform name	Type	Nr. Proc.	Freq. (MHz)	Mem. (GB)	Network type	BW (Mb/s)	OS
SGI Origin3000	DSM NUMA	56	700	42	SGI NUMA-Link		IRIX
SunFire 6800	DSM NUMA	24	900	24	Sun Fireplane Interconnect		Solaris
Icluster	Cluster	200	733	51.2	Ethernet	100	Linux
Albus	SMP Cluster	16	1333	8	Ethernet Myrinet	100 4000	Linux

6. Experimental validation

The aim of this section is to provide experimental evidence to help validate the PRO model. We use the list ranking and sorting problem as test-cases, and report results on their corresponding PRO-algorithms discussed in the previous section (Algorithms 3 and 4, respectively). We chose the list ranking and sorting problems for our experiments since they are good representatives of two different classes of problems: list ranking uses a highly irregular data structure (linked list) and sorting uses a highly regular data structure (array). In general, the relative communication cost associated with irregular data structures is higher than that associated with regular data structures.

6.1 Experimental setup

Both the list ranking and the sorting algorithms were implemented using the programming environment *SSCRAP* developed by Essaïdi *et al.* [2002, 2004]. *SSCRAP* (Soft Synchronized Computing in Rounds for Adequate Parallelization) is a C++ communication and synchronization library for implementing coarse-grained parallel algorithms on different platforms, including clusters and parallel machines. In addition to the PRO model, *SSCRAP* supports other variants of coarse-grained models. By providing a high level of abstraction, *SSCRAP* makes complex communication tasks transparent to the user and handles inter-processor data exchanges and synchronizations efficiently. Due to its efficiency, low overhead, and architecture-independence, *SSCRAP* can be used to carry out reproducible experimental studies.

In our experiments we considered four variants of platforms. The main features of these platforms are summarized in Table III. Starting with the leftmost column, the table lists the platform name, the architecture type, the number of available processors, the processor frequency, the total memory size, the interconnection type, the communication bandwidth and the operating system used in each case. We used two essentially different types of platforms: distributed-shared-memory (DSM) parallel machines and clusters. For DSM, we used two different 64 bit machines. The first one is an SGI Origin 3000 and the second is a SunFire 6800. In addition, we experimented with the SGI machine using two different sets of processors, the first of type R 12000 and the second of type R 16000. We refer to these as R12M and R16M, respectively. Table III also presents two different clusters, named Icluster and Albus. Icluster is a large PC cluster with about 200 common desktops powered by PIII processors. Albus is a cluster composed of 8 biprocessor-AMD Athlon MP SMP nodes. Albus has two different interconnections, a standard 100 Mb/s switched ethernet and a high speed Myrinet.

6.2 Experimental Results

6.2.1 Execution time

Figures 1(a) and 1(b) show execution time plots for the list ranking and sorting algorithms, respectively. The plots are on log-log scale, where the horizontal axis corresponds to *number of processors* and the vertical axis to *normalized execution times* per *number of items*. The term "number of items" here is a generic description for the number of list elements in the list ranking algorithm or the number of elements to be sorted in the sorting algorithm. The curves in Figure 1 show results for the *largest* number of items we were able to solve on the various platforms. To be able to compare behaviors across different architectures, the execution times have been normalized by the CPU frequency; thus the normalized quantities appear as clock cycles of the underlying architecture. In some sense this normalization also hides efficiency-differences across the platforms that would have appeared if pure running times were to be used. Should the actual runtimes be of interest, they can easily be obtained using the clock frequencies given in Table III.

For each four-tuple (algorithm, platform, number of items, number of processors), the result shown in Figure 1 is an average of 10 runs. In these runs, the variance was consistently observed to be very low. On the DSM machines, the execution times on one processor correspond to the execution times of an optimized *sequential implementation* and not to those of the parallel algorithm run on a single processor. Hence, the speedups observed on these machines are absolute, as opposed to relative. On the clusters, problem instances of the sizes reported here could not be solved on a single machine. Hence, sequential runtimes are not available for comparison on these platforms (the corresponding curves in Figure 1 start at a number of processors larger than one).

The following observations can be made from Figure 1.

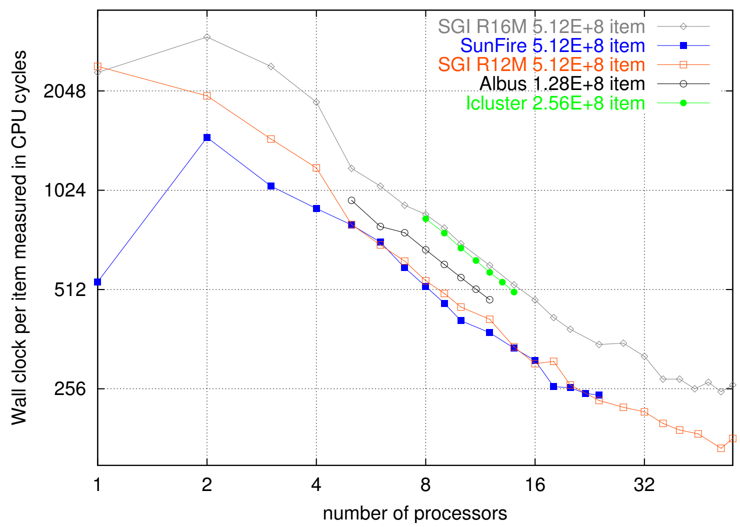
- The curves are to a large extent close to straight lines, indicating that the speedup is linear over a wide range of processors.
- A comparison between the execution time of the optimal sequential algorithm (the case where $p = 1$) and the parallel runtime on two processors reveals that the overhead for parallelization is small.
- The execution time curves are nearly parallel to each other.
- For both the list ranking and sorting algorithms, the behavior is remarkably similar on the various platforms.

6.2.2 Memory usage

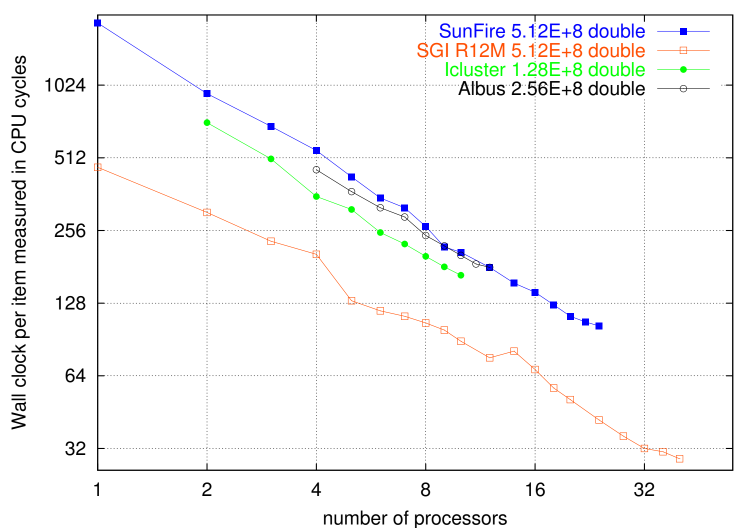
Let N_{seq} be the maximum input size (for the list ranking or the sorting algorithm) that can be computed sequentially in memory of size M_{seq}. The corresponding PRO algorithms using p processors would then solve inputs of size $\Omega(p \cdot N_{seq})$ in memory volume of $\Theta(p \cdot M_{seq})$.

To show that this behavior is observed in practice we present Figure 2. The figure shows the maximum input size that could be computed on the platform Icluster, for both the list ranking and the sorting algorithms. Each node of the cluster has 256 MByte local memory and a sequential version of the list ranking algorithm could only rank 5 million elements (resp. 10 million doubles for sorting). For the parallel PRO algorithm, by employing more nodes, larger input sizes could be computed. Figure 2 shows that, disregarding irregularities due to discretization, the maximum input sizes to the PRO algorithms scale linearly for a wide range of processors. Figure 2 also shows that sorting scales slightly better than list ranking. This is due to the recursion involved in and the memory overhead associated

Fig. 1: Computational results on all platforms. Ideal speedups correspond to curves of slope -1.

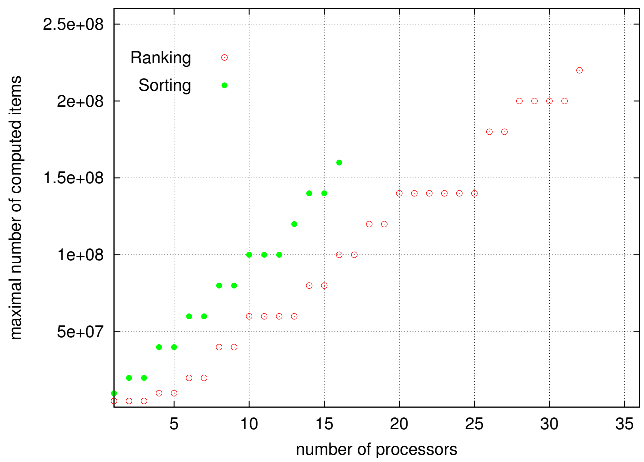


(a) List Ranking



(b) Sorting

Fig. 2: List Ranking and Sorting: Maximum input size computed on Icluster.



with the independent set construction in the list ranking algorithm; both of these aspects make the list ranking algorithm require more memory than the sorting algorithm.

The experimental set up used here (the model PRO, the programming environment *SSCRAP*, and the various test platforms) has also been successfully applied on a large variety of other algorithms, including algorithms for matrix multiplication (using BLAS routines), combinatorial problems on trees and lists, and problems on large cellular networks; see Essaïdi [2004], Essaïdi and Gustedt [2006] and Gustedt *et al.* [2006] for more information.

7. Conclusion

We have introduced a new parallel computation model (PRO) that enables the development of efficient and scalable parallel algorithms and simplifies their complexity analysis.

The distinguishing feature of the PRO model is the novel focus on relativity, resource-optimality, and a new quality measure (granularity). The PRO model requires a parallel algorithm to be both time- and space-optimal relative to an underlying sequential algorithm. Having optimality as a built-in requirement, the quality of a PRO-algorithm is measured by the maximum number of processors that could be used while the optimality of the algorithm is maintained.

The focus on relativity has theoretical as well as practical justifications. From a theoretical point of view, the performance evaluation metrics of a parallel algorithm include speedup and optimality, both of which are always expressed relative to some sequential algorithm. In practice, a parallel algorithm is often developed based on some known sequential algorithm. The fact that optimality is incorporated as a requirement in the PRO model enables one to concentrate only on parallel algorithms that are practically useful.

However, the PRO model is not just a collection of some 'ideal' features of parallel algorithms, it is also a means for achieving these. In particular, the attributes of the model capture the salient characteristics of a parallel algorithm that make its practical optimality and scalability highly likely. In this sense, it can also be seen as a parallel algorithm design scheme. We believe the experimental results reported in this paper go some distance in justifying this claim.

Acknowledgments

We thank the computing centers at LERI in Reims, IMAG in Grenoble, and LORIA in Nancy for generously allowing us to use their computing resources. This research is financially supported by IS-AUR 02-34 of The Aurora Programme, a France-Norway Collaboration Research Project of The Research Council of Norway, The French Ministry of Foreign Affairs and The Ministry of Education, Research and Technology; by the PRST *Intelligence Logiciel* of the Lorraine Region; by the *ACI ARGE* of the French Government; and by the US National Science Foundation grant ACI 0203722.

References

AKL, S. G. 1997. *Parallel Computation. Models and Methods.*. Prentice Hall, New Jersey, USA.

ALEXANDRAKIS, A. G., GERBESSIOTIS, A. V., LECOMBER, D. S., AND SINIOLAKIS, C. J. 1996. Bandwidth, space and computation efficient PRAM programming: The BSP approach. In *Proceedings of the SUP'EUR '96 Conference, Krakow, Poland*.

BAR-NOY, A. AND KIPNIS, S. 1992. Designing broadcasting algorithms in the Postal Model for message passing systems. In *The 4th annual ACM symposium on parallel algorithms and architectures*, 13–22.

BISSELING, R. H. 2004. *Parallel Scientific Computation: A structured approach using BSP and MPI*. Oxford.

BONORDEN, OLAF, JUURLINK, BEN H. H., VON OTTE, INGO, AND RIEPING, INGO. 1999. The Paderborn University BSP (PUB) Library—Design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*.

BRENT, R. P. 1974. The parallel evaluation of generic arithmetic expressions. *Journal of the ACM 21*, 2, 201–206.

CACERES, E., DEHNE, F., FERREIRA, A., LOCCHINI, P., RIEPING, I., RONCATO, A., SANTORO, N., AND SONG, S. W. 1997. Efficient Parallel Graph Algorithms For Coarse Grained Multicomputers and BSP. In *The 24th International Colloquium on Automata Languages and Programming*, Volume 1256 of *LNCS*. Springer Verlag, 390–400.

COLE, RICHARD AND VISHKIN, UZI. 1989. Faster optimal prefix sums and list ranking. *Information and Computation 81*, 3, 128–142.

CULLER, DAVID, KARP, RICHARD, PATTERSON, DAVID, SAHAY, ABHIJIT, SCHAUSER, KLAUS ERIK, SANTOS, EUNICE, SUBRAMONIAN, RAMESH, AND VON EICKEN, THORSTEN. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *4th ACM SIGPLAN Symposium on principles and practice of parallel programming, San Diego, CA*.

DEHNE, FRANK. 1999. Guest Editor's Introduction. *Algorithmica 24*, 3/4, 173–176.

DEHNE, FRANK K. H. A., FABRI, ANDREAS, AND RAU-CHAPLIN, ANDREW. 1996. Scalable parallel computational geometry for coarse grained multicomputers. *Int. J. on Comp. Geom. 6*, 3, 379–400.

ESSAÏDI, MOHAMED. 2004. *Echange de données pour le parallélisme à gros grain*. PhD thesis, Université Henri Poincaré.

ESSAÏDI, MOHAMED, GUÉRIN LASSOUS, ISABELLE, AND GUSTEDT, JENS. 2002. SSCRAP: An Environment for Coarse Grained Algorithms. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 398–403.

ESSAÏDI, MOHAMED, GUÉRIN LASSOUS, ISABELLE, AND GUSTEDT, JENS. 2004. SSCRAP: Soft Synchronized Computing in Rounds for Adequate Parallelization. Rapport de recherche, INRIA, `http://www.inria.fr/rrrt/rr-5184.html`.

ESSAÏDI, MOHAMED AND GUSTEDT, JENS. 2006. An experimental validation of the PRO model for parallel and distributed computation. In *14th Euromicro Conference on Parallel, Distributed and Network based Processing*. IEEE, The Institute of Electrical and Electronics Engineers.

FORTUNE, STEVEN AND WYLLIE, JAMES. 1978. Parallelism in random access machines. In *10th ACM Symposium on Theory of Computing*, 114–118.

GEBREMEDHIN, ASSEFAW HADISH, GUÉRIN LASSOUS, ISABELLE, GUSTEDT, JENS, AND TELLE, JAN ARNE. 2002. PRO: a model for Parallel Resource-Optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications*. IEEE, The Institute of Electrical and Electronics Engineers, 106–113.

GEBREMEDHIN, ASSEFAW HADISH, GUÉRIN LASSOUS, ISABELLE, GUSTEDT, JENS, AND TELLE, JAN ARNE. 2003. Graph Coloring on a Coarse Grained Multiprocessor. *Discrete Appl. Math. 131*, 1, 179–198.

GERBESSIOTIS, A. V., LECOMBER, D. S., SINIOLAKIS, C. J., AND SUJITHAN, K. R. 1998. PRAM programming: Theory vs. Practice. In *Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain*. IEEE Computer Society Press.

GERBESSIOTIS, A. V. AND SINIOLAKIS, C. J. 2001. A new randomized sorting algorithm on the BSP model. Tech. report, New Jersey Institute of Technology, `http://www.cs.njit.edu/~alexg/pubs/papers/rsort.ps.gz`.

GERBESSIOTIS, A. V. AND VALIANT, L. G. 1994. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing 22*, 251–267.

GIBBONS, P. B., MATIAS, Y., AND RAMACHANDRAN, V. 1999. Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? *Theory of Computing Systems 32*, 3, 327–359.

GREENLAW, R., HOOVER, H.J., AND RUZZO, W. L. 1995. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York.

GUÉRIN LASSOUS, ISABELLE AND GUSTEDT, JENS. 2002. Portable List Ranking: an Experimental Study. *ACM Journal of Experimental Algorithmics* 7, 7, `http://www.jea.acm.org/2002/GuerinRanking/`.

GUÉRIN LASSOUS, ISABELLE, GUSTEDT, JENS, AND MORVAN, MICHEL. 2000. Handling Graphs According to a Coarse Grained Approach: Experiments with MPI and PVM. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting*, Volume 1908 of *LNCS*. Springer-Verlag, 72–79.

GUSTEDT, JENS, VIALLE, STÉPHANE, AND DE VIVO, AMELIA. 2006. parXXL: A Fine Grained Development Environment on Coarse Grained Architectures. In *Workshop on state-of-the-art in scientific and parallel computing (PARA'06)*, `http://www.hpc2n.umu.se/para06/papers/paper_48.pdf`.

HAWICK, KEN ET AL. HPCC. High Performance Computing and Communications Glossary, `http://nhse.npac.syr.edu/hpccgloss/`.

HILL, JONATHAN M. D., MCCOLL, BILL, STEFANESCU, DAN C., GOUDREAU, MARK W., LANG, KEVIN, RAO, SATISH B., SUEL, TORSTEN, TSANTILAS, THANASIS, AND BISSELING, ROB. 1998. BSPlib: The BSP programming library. *Parallel Computing 14*, 1947–1980.

JÁJÁ, J. AND RYU, K. W. 1996. The Block Distributed Memory Model. *IEEE Transactions on Parallel and Distributed Systems 8*, 7, 830–840.

KARP, RICHARD M. AND RAMACHANDRAN, VIJAYA. 1990. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science*, Volume A, Algorithms and Complexity. Elsevier Science Publishers B.V., Amsterdam, 869–941.

KRUSKAL, CLYDE P., RUDOLPH, LARRY, AND SNIR, MARC. 1990. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science 71*, 1 (Mar.), 95–132.

MAGGS, B. M., MATHESON, L. R., AND TARJAN, R. E. 1995. Models of parallel computation: A survey and synthesis. In *28th HICSS*, Volume 2, 61–70.

SIBEYN, JOP F. 1999. Ultimate Parallel List Ranking? In *Proceedings of the 6th Conference on High Performance Computing*, 191–201.

VALIANT, LESLIE G. 1990. A bridging model for parallel computation. *Communications of the ACM 33*, 8, 103–111.

VITTER, J. S. AND SIMONS, R. A. 1986. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Transactions on Computers C-35*, 5, 403–418.