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

Assefaw Hadish Gebremedhin
Department of Computer Science
Old Dominion University
Norfolk VA 23529-0162 USA.
Email: assefaw@cs.odu.edu.

Isabelle Guérin Lassous
INRIA Rhne-Alpes & CITI
Lyon, France.
Email: Isabelle.Guerin-Lassous@inrialpes.fr

Jens Gustedt
LORIA & INRIA Lorraine
Nancy, France.
Email: Jens.Gustedt@loria.fr

Jan Arne Telle
Department of Informatics
University of Bergen
N-5020 Bergen, Norway.
Email: telle@ii.uib.no

**Abstract**

We present a new parallel computation model that enables the design of resource-optimal and scalable parallel algorithms and simplifies their analysis. The model rests on the following novel ideas: it incorporates optimality relative to a specific sequential algorithm as an integral part, and it measures the quality of a parallel algorithm in terms of granularity. Inspired by the BSP model, an algorithm in the PRO model is organized as a sequence of supersteps. The supersteps are not however required to be separated by synchronization barriers.

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

## I. Introduction

As Akl in his book on parallel computation [2] notes, a model of computation serves two major purposes. First, it is used to describe a computer. In this role, a model attempts to capture the essential features of an existing or contemplated machine while ignoring less important details of its implementation. Second, it is used as a tool for analyzing problems and expressing algorithms. In this sense, a model is not necessarily linked to any real computer but rather to an understanding of computation.

In the realm of sequential computation, the Random Access Machine (RAM) is a standard model that for many years has succeeded in achieving both of these purposes. It has served as an effective model for hardware designers, algorithm developers, and programmers alike, although one notes that the recent focus on external memory and cache issues has uncovered a need for more refined models. When it comes to parallel computation, there is no analogous, universally accepted model. 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, making the RAM powerful enough for analysis and design. On the other hand, the performance evaluation of a parallel algorithm involves several metrics, the most important of which 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 at the same time. In light of this and 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.

Research 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; and by the US National Science Foundation grant ACI 0203722. Part of this work has previously been published in [1]. Much of the work of the first author was carried out while completing a PhD at the University of Bergen.

The PRO model is inspired by the Bulk Synchronous Parallel (BSP) [3] and the Coarse Grained Multicomputer (CGM) [4] models. The introduction of the BSP model, in which a parallel algorithm is organized as a sequence of *supersteps* with distinct computation and communication phases, 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 [5]–[7]. Recently, Bisseling [8] has written a textbook on scientific parallel computation using the BSP model. From an algorithmic, as opposed to a programming, point of view, however, 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. In an interesting survey paper [9], Maggs *et al*. suggest that an ideal parallel computation model be designed within "the philosophy of *simplicity* and *descriptivity* balanced with *prescriptivity*". The PRO model is designed with this philosophy in mind.

Three key features distinguish the PRO model from existing parallel computation models. These 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 of this, 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 $A_{seq}$. 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 our focus on relativity. The notion of relativity is also relevant from a practical point of view since a parallel algorithm is usually not designed from scratch, but rather starting from a sequential algorithm.

A PRO-algorithm is required to be asymptotically both time- and space-optimal, hence resource-optimal. A parallel algorithm is said to be asymptotically 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 asymptotically 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, *i.e.*, the ratio between the sequential and 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. The fact that optimality is required only asymptotically leaves enough slackness for easy design and analysis of algorithms.

Before turning to the quality measure of a PRO algorithm, let us emphasize the consequences of the novel notion of relativity. In PRO, instead of directly comparing algorithms that solve the same problem, we employ a two-leveled approach. First, select a particular space and time complexity based on some sequential algorithm $A_{seq}$. Then, compare parallel algorithms that are resource-optimal with respect to $A_{seq}$. The latter comparison, *i.e.*, the quality of a PRO algorithm, is measured by the range of values parameter $p$ can assume while linear speedup is maintained. It is captured by an attribute of the model called the granularity function $\text{Grain}(n)$. In particular, a PRO-algorithm with granularity $\text{Grain}(n)$ is required to yield optimal speedup, within a constant factor, for all values of $p$ such that $p = O(\text{Grain}(n))$; in other words, the algorithm is fully *scalable* for $p = O(\text{Grain}(n))$. The higher the function value $\text{Grain}(n)$, 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 space complexity of the reference sequential algorithm and the granularity function. A

new result will typically be presented as follows: 'We have given a PRO-algorithm for problem $P$ relative to sequential time $T(n)$ and space $S(n)$ which has granularity Grain$(n)$.' This simply means that for any number of processors $p$ and input size $n$ with $p = O(\text{Grain}(n))$, there is a parallel algorithm in the PRO model for problem $P$ where the parallel runtime is $O(T(n)/p)$ and each processor uses $O(S(n)/p)$ memory.

The rest of the paper is organized as follows. In Section II we highlight the limitations of a few relevant existing parallel computation models, to justify the need for the introduction of the new model PRO. In Section III the PRO model is presented in detail and in Section IV it is systematically compared with a selection of existing parallel models. In Section V we illustrate how the model is used in the design and analysis of an algorithm using the matrix multiplication and list ranking problems as concrete examples. In Section VI we give PRO-algorithms for two commonly used communication primitives, one-to-all broadcast and all-to-all communication. In Section VII we discuss observations made from related experimental studies. We conclude the paper in Section VIII with some remarks.

## II. 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 in two groups as 'dedicated' (to either software or hardware) and 'bridging' (between software and hardware) models.

### A. Dedicated models

*1) PRAM:* In its standard form, the PRAM model [10], [11] 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 a 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 [12] is used to reflect the ease/hardness of finding a parallel algorithm for a problem.

Recall that, on the PRAM model, a parallel algorithm is considered 'efficient' if its runtime is polylogarithmic, *i.e.*, $O(\log^k n)$ for some fixed constant $k$, while the number of processors it uses is polynomial in the input size $n$. The class $NC$ consists of problems that can be solved by such efficient parallel algorithms. By simulation, an $NC$-algorithm can be converted into a polynomial time sequential algorithm. Thus, the class $NC$ is included in the class $P$, *i.e.*, $NC \subseteq P$. On the other hand, whether or not $P \subseteq NC$ is an open problem in complexity theory. The general belief is that $P \nsubseteq NC$, and hence that there are problems in $P$ that do not have $NC$-algorithms. The class of $P$-complete problems consists of the most likely candidates for such problems. Informally, a problem is said to be $P$-complete if an $NC$-algorithm for it implies that all problems in $P$ would have $NC$-algorithms.

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 [13]. 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 [4], [14]–[16] show that this approach is practically relevant.

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

Third, even if we restrict ourselves to work-optimal $NC$-algorithms and apply Brent's scheduling principle, which says an 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 [17]. 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.

*a) Enhanced PRAM's:* 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 [9].

*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 [18] and the Block Distributed Memory (BDM) model [19]. Other categories of parallel models such as low-level, hierarchical memory, and network models are briefly reviewed in [9].

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

### *B. Bridging models*

Valiant in his seminal paper [3] underscored that a successful parallel computation model needs to act as an efficient 'bridge' between software and hardware. He introduced the BSP as a candidate bridging model and argued that it could serve as a standard model for parallel computation.

*1) BSP:* 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 barrier synchronizations. 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. Note that, here, $h$ relates 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.* [20] 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. A common feature of the BSP, LogP, and other related models is their lack of simplicity: each model involves relatively many parameters making analysis and design of algorithms cumbersome.

*2) CGM:* The CGM model [4], [14] was proposed in an effort to retain the advantages of BSP while keeping the model simple. 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 barrier synchronizations. 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.

Assumption (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 assumption (1) holds. Then, since each superstep has a complexity of $\Theta(h) = \Theta(n/p)$, the only parameter of the model that distinguishes different algorithms is the number of supersteps. This direction was followed for instance in [14] where a long list of algorithms that are designed under these assumptions is given.

Nevertheless there exists a large class of problems for which these assumptions are not known to hold in that we do not know CGM algorithms where the communication time to computation time ratio is a constant. 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 therefore one needs to consider computation as well.

Moreover, even for problems whose algorithms are such that the stated ratio is constant, assumption (1) turns out to be quite restrictive. We will illustrate this with the *list ranking* problem which we will discuss in more details in Section V-B. There we will see that CGM fails to identify competitive algorithms when using the number of supersteps as a quality measure.

## III. 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 Time$(n)$ and Space$(n)$ denote the time and space complexity of a specific sequential algorithm for a given problem with input size $n$. Let 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.

As discussed in the LogP paper [20], 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 is consistent with this convergence and maps well on several existing parallel computer architectures. The memory requirement $M = O(\frac{\text{Space}(n)}{p})$ 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 Space$(n) = \Omega(n)$, 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 Space$(n) = \Theta(n)$, note that the input data must be uniformly distributed on the $p$ processors. In this case the machine model assumption of PRO is similar to the assumption in the CGM model [4].

The *coarseness* assumption $p \leq M$ is consistent with the structure of existing parallel machines and those to be built in the foreseeable future. The assumption is required to simplify the implementation of gathering messages on or broadcasting messages from a single processor.

TABLE I
ATTRIBUTES OF THE PRO MODEL

| | |
|---|---|
| Machine Model: | The underlying machine is assumed to consist of $p$ processors each of which has a private memory of size $M = O(\frac{\text{Space}(n)}{p})$. 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. |
| Coarseness Assumption: | The size of the local memory of each processor is assumed to be big enough to store $p$ words (*i.e.* the relationship $p \leq M$ is assumed to hold). |
| 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<br>• sends at most one message to every other processor,<br>• sends and receives at most $M$ words in total,<br>• performs local computation. |
| Runtime Analysis: | Both *computation* and *communication* are accounted for in the runtime analysis of a PRO algorithm. In particular,<br>• a processors is charged a unit of time per operation performed locally, and<br>• a processor is charged a unit of time per machine word sent or received. |
| Optimality Requirement: | For any value $p = O(\text{Grain}(n))$, a PRO algorithm is required to have<br>• at most $o(\frac{\text{Time}(n)}{p^2})$ supersteps, and<br>• a parallel runtime $\text{Time}(n,p) = O(\frac{\text{Time}(n)}{p})$.<br>The *granularity* function $\text{Grain}(n)$ measures the *quality* of the algorithm. |

In terms of *execution*, a PRO-algorithm consists of a sequence of *supersteps* (or rounds). 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. The length of a superstep $s$ in the parallel algorithm seen as a whole, denoted by $\text{Time}_s(n,p)$, is the maximum over the lengths of the superstep on all processors. We can conceptually think as if the supersteps are synchronized by a barrier set at the end of the longest superstep across the processors. However, in PRO the processors are not in reality required to synchronize at the end of each superstep. The parallel runtime $\text{Time}(n,p)$ of the algorithm is the sum of the lengths of all the supersteps. Notice that the hypothetical barriers result in only a constant factor more time compared 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, each processor is involved in at most $2(p-1)$ messages per superstep. Therefore, the requirement $\text{Steps} = o(\frac{\text{Time}(n)}{p^2})$ on the number of supersteps implies that the overall time paid per processor for *communication*

*overhead* and *latency* is $o(\text{Time}(n)/p)$ and hence can be neglected from the analysis since our goal is to achieve an $O(\text{Time}(n)/p)$ parallel runtime. Such a statement would not hold if we were to replace the little-*o* in the requirement for the number of supersteps by a big-*O*. Had we allowed $O(\text{Time}(n)/p^2)$ supersteps, the corresponding latency could be of the same order of magnitude as the computation or communication time and the success of an execution would then depend on the particular parameters of the concrete machine. Since latency is a parameter that is determined by fundamental physical restrictions (the speed of light) and will not as easily improve as other architectural parameters, such a possible dependency on latency of an algorithm is not desirable.

On the other hand, the bandwidth of the underlying architecture is not subject to such principal restrictions and in fact computation and communication would contribute to the overall runtime in similar terms. Therefore, it is important that bandwidth is accounted for. To this end, in PRO each processor pays a unit of time per word sent and received. This is not an unrealistic assumption noting that the network throughput (accounted in machine words) on modern architectures such as high performance clusters is relatively close to the CPU frequency and to the CPU/memory bandwidth.

The condition $\text{Time}(n,p) = O(\frac{\text{Time}(n)}{p})$ requires that a PRO-algorithm be optimal and yield linear speedup relative to the sequential algorithm used as a reference. This requirement ensures the potential practical use of the parallel algorithm.

The function $\text{Grain}(n)$ is a *quality measure* for a PRO algorithm, *i.e.*, for any 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. The objective in designing a PRO algorithm is to make $\text{Grain}(n)$ as high as possible, thereby increasing its scalability. As the following observation shows, there is an upper bound on $\text{Grain}(n)$ set by the complexity of the reference sequential algorithm.

*Observation 1:* A PRO algorithm relative to a sequential algorithm with runtime $O(\text{Time}(n))$ and space requirement $O(\text{Space}(n))$ has maximum granularity

$$\text{Grain}(n) = o\left(\sqrt{\text{Space}(n)}\right). \tag{2}$$

A PRO algorithm that achieves this is said to have optimal grain.
Observation 1 is due to

(i) the limit on the memory size of each processor,
(ii) the coarseness assumption, and
(iii) the bound on the number of supersteps.

The limit on the size of the private memory of each processor ($M = O(\frac{\text{Space}(n)}{p})$) together with the coarseness assumption $p \leq M$ imply $p = O(\sqrt{\text{Space}(n)})$. The fact that the number of supersteps of a PRO-algorithm should be $\text{Steps} = o(\text{Time}(n)/p^2)$, gives $p = o(\sqrt{(\text{Time}(n)/\text{Steps})})$ upon resolving and we clearly have $\text{Steps} \geq 1$. Finally, note that $\text{Time}(n) \geq \text{Space}(n)$, since we may reasonably assume that all memory is initialized.

Since a PRO-algorithm yields linear speedup for any $p = O(\text{Grain}(n))$, a result like Brent's scheduling principle 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.

We note that there exists a slightly similar notion to $\text{Grain}(n)$ in the literature. This notion, called *iso-efficiency* function, is suggested as an analytical tool for evaluating the scalability of a parallel system (parallel algorithm together with a parallel architecture) [21], [22]. The iso-efficiency function, which is defined only for scalable algorithms, determines the ease with which a parallel system achieves speedups increasing in proportion to the number of processors involved. A small iso-efficiency function implies that a small increase in problem size is sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel system is highly scalable. A large iso-efficiency function, on the other hand, indicates a poorly scalable parallel system. Thus, the iso-efficiency function, which is

TABLE II
COMPARISON OF PARALLEL COMPUTATIONAL MODELS

| | PRAM [10] | QSM [23] | BSP [3] | LogP [20] | CGM [14] | 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 | $n$ | $p, g, n$ | $p, g, L, n$ | $p, g, l, o, n$ | $p, n$ | $p, n, A_{seq}$ |
| granularity | fine | fine | coarse | fine | coarse | Grain$(n)$ |
| speedup | NA | NA | NA | NA | NA | $\Theta(p)$ |
| optimal | NA | NA | NA | NA | NA | rel. $A_{seq}$ |
| quality | time | time | time | time | rounds | Grain$(n)$ |

expressed as a function of $p$, is in a sense "inversely" related to Grain$(n)$ in PRO. Despite this slight resemblance, the two notions are fundamentally different. Grain$(n)$ is an attribute of a model; it is in fact a quality measure for a PRO-algorithm. The iso-efficiency function is not related to any computation model. Moreover, it is not always possible to get an analytic expression for the iso-efficiency function, even if the parallel algorithm is highly scalable in practice [21].

The design of a PRO-algorithm may sometimes involve subroutines for which there do not exist sequential counterparts. 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 notation, we will call such parallel routines PRO-algorithms if the overall computation and communication cost is linear in the input size to the routines.

## IV. COMPARISON WITH OTHER MODELS

In this section we compare the PRO model with PRAM, BSP, LogP, CGM, and the Queuing Shared Memory (QSM) model [23] (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 [23]. 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 the model is indicated in the row labeled *synch.* Lock-step indicates that the 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 the 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 either be distributed or shared. The row labeled *commun.* shows the type of interprocessor communication assumed by the 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 the model are indicated in the row labeled *parameters*. The number of processors is denoted by $p$, $n$ is the input size, $A_{seq}$ 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 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 cost. Communication overhead is hidden by the PRO-requirement that states Steps $= o(\frac{\text{Time}(n)}{p^2})$.

The row labeled *granularity* indicates whether the model is fine-grained, coarse-grained or a more precise measure is used. We say that a model is coarse-grained if it applies to the case where $n \gg p$ and call it fine-grained if it relies on using up to a polynomial number of processors in the input size. In PRO granularity is exactly the quality measure Grain$(n)$, and appears as one of the attributes of the model.

The rows labeled *speedup* and *optimal* indicate the speedup and resource optimality requirements imposed by the 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. $A_{seq}$' means that the algorithm is optimal relative to the time and space complexity of $A_{seq}$. We point out that the goal in the design of algorithms using the CGM model [4], [14] is usually 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 all other models 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.

## V. Algorithm examples

In this section we use two examples to illustrate how the PRO model is used. In each example, we start with a given sequential algorithm and then design and analyze a parallel algorithm relative to it. Our first example is the standard matrix multiplication algorithm with three nested for-loops. This example is chosen for its simplicity, since our objective at this stage is to illustrate the use of a new model rather than solving a "difficult" problem. The second example is an algorithm for list ranking, a basic routine used in many parallel graph algorithms. This example is interesting as a CGM-analysis would regard it inefficient, despite the fact that it is efficient in practice.

### A. 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 which has Time$(n) = O(n^{\frac{3}{2}})$ and Space$(n) = O(n)$.

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 (respectively columns) $\frac{m}{p} \cdot i + 1$ to $\frac{m}{p} \cdot (i + 1)$ of $A$ (respectively $B$). The output matrix $C$ will be row-partitioned among the $p$ processors in a similar fashion. Notice that with this data distribution each processor can, without communication, compute a block of $\frac{m^2}{p^2}$ of the $\frac{m^2}{p}$ entries of $C$ expected to reside on it. 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 $P_{i+1}$. In each superstep the processors in the PRO algorithm will therefore exchange columns in a round-robin fashion and then each will compute 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. In Algorithm 1, we have organized this sequence of computation and communication steps 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 $\Theta(m^3/p^2) = \Theta(n^{\frac{3}{2}}/p^2)$ per superstep. Likewise, the total size of data (words) exchanged by each processor in a superstep is $\Theta(m^2/p) = \Theta(n/p)$. Thus, the length of a superstep $s$ is Time$_s(n, p) = \Theta(n^{\frac{3}{2}}/p^2 + n/p)$. Note that for $p = O(\sqrt{n})$, Time$_s(n, p) = \Theta(n^{\frac{3}{2}}/p^2)$. Hence, for $p = O(\sqrt{n})$, the overall parallel runtime of the algorithm is

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

**Algorithm 1**: Matrix multiplication

**Input**: Two $m \times m$ matrices $A$ and $B$. The rows (columns) of $A$ ($B$) are divided into $m/p$ contiguous blocks, and stored on processors $P_0, P_1, \ldots P_{p-1}$, respectively.

**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 the local sub-matrix product of its rows and current columns;

    $P_{(i+1) \ mod \ p}$ sends its current block of columns to $P_i$;

    $P_i$ receives a new current block of columns from $P_{(i+1) \ mod \ p}$;

Noting that Space$(n) = \Theta(n)$, we see that the memory restriction of the PRO model is respected, *i.e.*, each processor has enough memory size to handle the transactions. In order to be able to neglect communication overhead, the condition on the number of supersteps, which in this case is just $p$, should be met. In other words, we need $p = o(\text{Time}(n)/p^2) = o(n^{\frac{3}{2}}/p^2)$, which is true for $p = o(\sqrt{n})$. Thus the granularity function of the PRO-algorithm is Grain$(n) = o(\sqrt{n})$.

The following lemma summarizes this result.

*Lemma 1:* Multiplication of two $m \times m$ matrices has a PRO-algorithm with Grain$(n) = o(m)$, relative to a sequential algorithm with Time$(n) = m^3$ and Space$(n) = m^2$ (input size $n = m^2$).

From Observation 1, we note that Algorithm 1 has optimal grain. Note that on a 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 finer granularity than $m$. For example, a systolic matrix multiplication algorithm has a granularity of $m^2$. However, PRO is intended to be applicable for general problems (including those with irregular communication pattern) and practically relevant parallel systems.

### *B. List Ranking*

The *list ranking* problem (LR) is the following: given a linked list as an input, determine the distance of every item in the list from the *end* of the list. This problem has an obvious linear time sequential solution and a classical question in the beginning of parallel computing has been to investigate how well it can be solved in parallel (see for example [24]–[26]).

**Algorithm 2**: List Ranking using Pointer Jumping

**Input**: An integer $n$ and a list of length $n$ given by a vector `next`; A vector `dist` (initialized with 1) where `dist[i]` stores the distance from item $i$ in the list to item `next[i]`.

**foreach** *Processor* $P_i$ **do**

  **while** $n > 0$ **do**

    dist[i] = dist[i] + dist[next[i]];

    next[i] = next[next[i]];

    $n = n/2$;

One parallel solution for LR uses *pointer jumping*; Algorithm 2 uses this technique. The algorithm has a logarithmic number of phases. In each phase, each list item updates its distance with the distance that is known to its neighbor and then updates the neighbor.

This algorithm is easily translated into a CGM algorithm: phases correspond to supersteps in which the processors communicate their respective values `dist[i]` and `next[i]`. The number of supersteps

is $\log n$ (or $\log p$ 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.

**Algorithm 3**: Recursive List Ranking

**Input**: An integer $n$ and a list of length $n$ given by a vector `next`; A vector `dist` (initialized with 1) where `dist[i]` stores the distance from item $i$ in the list to item `next[i]`.
Find a large independent set $I$;
Compute the sublist $next'$ that is generated by $I$ and the relative distances $dist'$ between neighboring elements in $I$;
Recurse on $next'$ and $dist'$;
Using the values obtained from the recursion, update the remainder of the lists $next$ and $dist$ appropriately;

Other, more sophisticated, solutions for LR use the *random mating* technique. Algorithm 3 illustrates such an approach. The main idea here is to determine a large independent set in the list and then to apply recursion on either this independent set or its complement. The translation of such algorithms to CGM is usually straightforward and the analysis with CGM is simple. 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 and so the work load decreases with every step of the recursion. 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 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 (pointer jumping) and a "good" one (random mating).

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 $\text{Time}(n,p) = \Theta(n/p) = \Theta(\text{Time}(n)/p)$. In order to be able to neglect communication overhead, we need to meet the condition on the number of supersteps, which for this algorithm is $\log n$. This means we need $p^2 = o(\frac{n}{\log n})$, or taking square roots of both sides, $p = o(\sqrt{\frac{n}{\log n}})$. With $\text{Space}(n) = \Theta(n)$, the PRO memory restriction is respected.

The following lemma summarizes these results.

*Lemma 2:* List ranking on $n$ elements has a PRO-algorithm with $\text{Grain}(n) = o(\sqrt{\frac{n}{\log n}})$, relative to a sequential algorithm with $\text{Time}(n) = O(n)$ and $\text{Space}(n) = O(n)$.

Note that this algorithm does not have an optimal grain; yet it is a PRO algorithm.

## VI. Communication primitive examples

A good parallel computation model should have a selection of algorithms for primitive communication tasks available in its algorithm design tool-box. The PRO model is intended to meet this demand. In this section we present two such primitives: one-to-all broadcast and all-to-all communication.

### A. One-to-all broadcast

Algorithm 4 outlines a PRO-algorithm for the basic communication primitive one-to-all broadcast. Since there is no sequential basis algorithm in this case, we want an algorithm whose overall communication and computation cost is linear in the input and output sizes. More precisely, we consider the situation where the input consists of a vector of size $m$ on a single processor and the output should be a copy of this vector on each of the $p$ processors, and we want an algorithm that achieves this in $O(m)$ time using $O(m)$ memory on each processor.

**Algorithm 4**: One-to-All Broadcast

**Input**: A vector $V$ of size $m$ on processor $P_0$
**Output**: A copy of $V$ on each processor

S1 $P_0$ divides $V$ into $p$ equal sized parts;
$P_0$ sends the $i^{th}$ part of $V$ to processor $P_i$, for each $0 < i < p$;
**foreach** *processor* $P_i$, $i > 0$ **do**
  processor $P_i$ receives the $i^{th}$ part from $P_0$;

S2 **foreach** *processor* $P_i$ **do**
  $P_i$ sends out the $i^{th}$ part to $P_j$, for each $j \neq i$ and $0 < j < p$.

**foreach** *processor* $P_j$,$j \neq 0$ **do**
  $P_j$ receives the $i^{th}$ part from $P_i$, for each $i \neq j$ and $0 < i < p$

**Algorithm 5**: All-to-All communication

**Input**: Each processor has a vector $V$ of size $m$.
**Output**: Data is redistributed among the processors. Each processor collects the new data in a vector $W$ of size $m$.

S1 **foreach** *processor* $P_i$, $i \geq 0$ **do**
  $P_i$ groups and calculates the size of the data in $V$ destined to each processor $P_j$, $0 \leq j < p$;
  $P_i$ sends out to each $P_j$, for $0 \leq j < p$, the size of data in $V$ that it will receive from $P_i$.

**foreach** *processor* $P_j$,$j \geq 0$ **do**
  $P_j$ receives from each $P_i$, $0 \leq i < p$, the size of data that $P_i$ will send to it at the next step;
  $P_j$ calculates the total size of data it will receive;
  $P_j$ allocates the required size for $W$.

S2 **foreach** *processor* $P_i$, $i \geq 0$ **do**
  $P_i$ sends to each processor $P_j$,$0 \leq j < p$ the data destined to $P_j$

**foreach** *processor* $P_j$,$j \geq 0$ **do**
  $P_j$ receives from each $P_i$, $0 \leq i < p$, the data it stores in $W$

*Lemma 3:* The PRO Algorithm 4 implements one-to-all broadcast of $m$ memory words in two supersteps using $O(m)$ time and $O(m)$ space per processor, for any number of processors $p \leq m$.

*Proof:* First, we note that the algorithm correctly broadcasts the desired vector $V$, while observing the space restriction, in two supersteps. We turn to the timing. In step S1 processor $P_0$ in total sends out $(p-1)m/p$ words and each of the other processors receives a message of size $m/p$. In step **S2** processor $P_i$ in total sends out $\frac{p-2}{p}m$ words. Processor $P_j$, $j \neq 0$, in total receives $\frac{p-1}{p}m$ words.

The total time is dominated by communication which can be bounded as follows:

$$(p-1)m/p + m/p + \frac{p-2}{p}m + \frac{p-1}{p}m = \tag{4}$$

$$m/p(p+p-2+p-1) < 3m \tag{5}$$

for total time $O(m)$ as claimed. □

### *B. All-to-all communication*

The primitive all-to-all communication redistributes data among processors. Each processor has an input vector of size $m$ and it sends each of its elements to every other processor. Algorithm 5 outlines a PRO algorithm for such a primitive.

*Lemma 4:* The PRO Algorithm 5 implements all-to-all communication of $m$ memory words in two supersteps using $O(m)$ time and $O(m)$ space per processor, for any number of processors $p \leq m$.

*Proof:* First, we note that the algorithm correctly redistributes the data stored in $V$ among the processors in two supersteps. Second, the space conditions are respected when $p \leq m$. Concerning timing, in step S1, each processor sends out $p-1$ words and each processor receives $p-1$ words. In step S2, each processor sends out at most $m$ words and receives $O(m)$ words. Therefore, for any number of processors $p \leq m$, the total time is $O(m)$. □

## VII. Related Work

This work is a culmination of several previous and concurrent studies [25], [27]–[30]. One of the common objectives in these studies has been to experimentally verify the relevance of the PRO model. Here we give a brief summary of the observations made in these studies.

- There is an ongoing project whose aim is to develop a library for implementing PRO algorithms. This library called *Soft Synchronized Computing in Rounds for Adequate Parallelization* (SSCRAP) [31] offers a framework that allows for implementations of PRO algorithms that are portable on a large variety of parallel and distributed platforms.
- Algorithms that *a posteriori* fit within the PRO model for such problems as permutation generation [32], sorting [33], list ranking [25], matrix multiplication [27], connected components, interval graphs, permutation graphs [29], [34] have been implemented using our tools. These algorithms have been found to offer almost linear speedup and a high degree of scalability.
- The aforementioned experiments demonstrated that the bound on the number of supersteps required in the PRO model effectively hides latency. In particular, contrary to the recommendation in CGM, algorithms with a number of supersteps that is a growing function in $n$ (and not in $p$) show satisfactory performance [25], [35]. The results also showed that performance is mainly determined by computing power and bandwidth restrictions.

## VIII. Conclusion

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

The distinguishing features of the PRO model are the novel focus on relativity, resource-optimality, and a new quality measure (granularity). In particular, the 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 includes speedup and optimality, both of which are always expressed relative to some sequential algorithm. Moreover, there is an inherent asymmetry between sequential and parallel computation. A parallel algorithm would always imply a sequential algorithm, whereas the converse is usually not true. Thus, in a sense, it is natural to think of an underlying sequential algorithm whenever one speaks of a parallel algorithm. From a practical point of view, one notes that the development of a parallel algorithm is often built 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 to achieve these features. 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. Moreover, the simplicity of the model eases analysis.

We believe that the PRO model is a step forward towards the identification of problems for which ‘practically good’ parallel algorithms exist. Much work remains to be done, and we hope that other members of the research community will join in.

# REFERENCES


[1] A. H. Gebremedhin, I. Guérin Lassous, J. Gustedt, and J. A. Telle, "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, 2002, pp. 106–113.

[2] S. G. Akl, *Parallel Computation. Models and Methods.* New Jersey, USA.: Prentice Hall, 1997.

[3] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[4] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable parallel computational geometry for coarse grained multicomputers," *International Journal on Computational Geometry*, vol. 6, no. 3, pp. 379–400, 1996.

[5] O. Bonorden, B. H. H. Juurlink, I. von Otte, and I. Rieping, "The Paderborn University BSP (PUB) Library—Design, Implementation and Performance," in *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, 1999.

[6] J. M. D. Hill, S. R. Donaldson, and A. McEwan, "Installation and user guide for Oxford BSP toolset (v1.4) implementation of BSPlib," Oxford University Computing Laboratory, Tech. Rep., 1998.

[7] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "BSPlib: The BSP programming library," *Parallel Computing*, vol. 24, pp. 1947–80, 1998.

[8] R. H. Bisseling, *Parallel Scientific Computation: A structured approach using BSP and MPI*. Oxford, 2004.

[9] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, "Models of parallel computation: A survey and synthesis," in *28th HICSS*, vol. 2, January 1995, pp. 61–70.

[10] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *10th ACM Symposium on Theory of Computing*, May 1978, pp. 114–118.

[11] J. Jájá, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[12] R. Greenlaw, H. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*. New York: Oxford University Press, 1995.

[13] J. S. Vitter and R. A. Simons, "New classes for parallel complexity: A study of unification and other complete problems for P," *IEEE Transactions on Computers*, vol. C-35, no. 5, pp. 403–418, 1986.

[14] E. Caceres, F. Dehne, A. Ferreira, P. Locchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song, "Efficient parallel graph algorithms for coarse grained multicomputers and BSP," in *The 24th International Colloquium on Automata Languages and Programming*, ser. LNCS, vol. 1256. Springer Verlag, 1997, pp. 390–400.

[15] A. H. Gebremedhin, I. Guérin Lassous, J. Gustedt, and J. A. Telle, "Graph coloring on a coarse grained multiprocessor," *Discrete Appl. Math.*, vol. 131, no. 1, pp. 179–198, 2003.

[16] I. Guérin Lassous, J. Gustedt, and M. Morvan, "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*, ser. LNCS, J. Dongarra, P. Kacsuk, and N. Podhorszki, Eds., vol. 1908. Springer-Verlag, 2000, pp. 72–79.

[17] F. Dehne, "Coarse grained parallel algorithms," *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, vol. 24, no. 3/4, pp. 173–176, 1999.

[18] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the Postal Model for message passing systems," in *The 4th annual ACM symposium on parallel algorithms and architectures*, July 1992, pp. 13–22.

[19] J. JáJá and K. W. Ryu, "The Block Distributed Memory model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 7, pp. 830–840, 1996.

[20] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *4th ACM SIGPLAN Symposium on principles and practice of parallel programming, San Diego, CA*, May 1993.

[21] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Addison Wesley, 2003.

[22] A. Gupta and V. Kumar, "Isoefficiency function: a scalability metric for parallel algorithms and architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, pp. 922–932, 1993.

[23] P. B. Gibbons, Y. Matias, and V. Ramachandran, "Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation?" *Theory of Computing Systems*, vol. 32, no. 3, pp. 327–359, 1999.

[24] R. Cole and U. Vishkin, "Faster optimal prefix sums and list ranking," *Information and Computation*, vol. 81, no. 3, pp. 128–142, 1989.

[25] I. Guérin Lassous and J. Gustedt, "Portable list ranking: an experimental study," *ACM Journal of Experimental Algorithmics*, vol. 7, no. 7, 2002. [Online]. Available: http://www.jea.acm.org/2002/GuerinRanking/

[26] J. F. Sibeyn, "Ultimate Parallel List Ranking?" in *Proceedings of the 6th Conference on High Performance Computing*, 1999, pp. 191–201.

[27] M. Essaïdi, "Echange de données pour le parallélisme à gros grain," Ph.D. dissertation, Université Henri Poincaré, Feb. 2004.

[28] M. Essaïdi, I. Guérin Lassous, and J. Gustedt, "SSCRAP: An environment for coarse grained algorithms," in *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 2002, pp. 398–403.

[29] I. Guérin Lassous, "Étude et implantation d'algorithmes parallèles de traitement de graphes sans circuit," Ph.D. dissertation, université Paris 7 - Denis Diderot, Jan 2000.

[30] J. Gustedt, "Towards realistic implementations of external memory algorithms using a coarse grained paradigm," in *International Conference on Computational Science and its Applications (ICCSA 2003), part II*, ser. LNCS, no. 2668. Springer, 2003, pp. 269–278.

[31] Http://www.loria.fr/ gustedt/sscrap/.

[32] J. Gustedt, "Randomized permutations in a coarse grained parallel environment [extended abstract]," in *Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'03)*. ACM, June 2003, pp. 248–249.

[33] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 251–267, 1994.

[34] I. Guérin Lassous, J. Gustedt, and M. Morvan, "Feasability, portability, predictability and efficiency: Four ambitious goals for the design and implementation of parallel coarse grained graph algorithms," INRIA, Tech. Rep., 00a.

[35] M. Essaïdi and J. Gustedt, "An experimental validation of the PRO model for parallel and distributed computation," in *14th Euromicro Conference on Parallel, Distributed and Network based Processing*, B. Di Martino, Ed. IEEE, The Institute of Electrical and Electronics Engineers, 2006.

[36] R. M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. A, Algorithms and Complexity. Elsevier Science Publishers B.V., Amsterdam, 1990, pp. 869–941.

[37] K. Hawick *et al.*, "High performance computing and communications glossary." [Online]. Available: http://nhse.npac.syr.edu/hpccgloss/

[38] C. P. Kruskal, L. Rudolph, and M. Snir, "A complexity theory of efficient parallel algorithms," *Theoretical Computer Science*, vol. 71, no. 1, pp. 95–132, Mar. 1990.

[39] R. P. Brent, "The parallel evaluation of generic arithmetic expressions," *Journal of the ACM*, vol. 21, no. 2, pp. 201–206, 1974.

[40] A. V. Gerbessiotis and C. J. Siniolakis, "A new randomized sorting algorithm on the BSP model," New Jersey Institute of Technology, Tech. Rep., 2001.

[41] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan, "PRAM programming: Theory vs. practice," in *Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain*. IEEE Computer Society Press, January 1998.

[42] A. G. Alexandrakis, A. V. Gerbessiotis, D. S. Lecomber, and C. J. Siniolakis, "Bandwidth, space and computation efficient PRAM programming: The BSP approach," in *Proceedings of the SUP'EUR '96 Conference, Krakow, Poland*, September 1996.