

SSCRAP : Soft Synchronized Computing in Rounds for Adequate Parallelization

Mohamed Essaïdi, Isabelle Guérin Lassous, Jens Gustedt

▶ **To cite this version:**

Mohamed Essaïdi, Isabelle Guérin Lassous, Jens Gustedt. SSCRAP : Soft Synchronized Computing in Rounds for Adequate Parallelization. [Research Report] RR-5184, INRIA. 2004. inria-00071404

HAL Id: inria-00071404
https://inria.hal.science/inria-00071404

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SSCRAP: Soft Synchronized Computing in Rounds for Adequate Parallelization

Mohamed Essaïdi — Isabelle Guérin Lassous — Jens Gustedt

N° 5184

Mai 2004

Thème NUM



ISSN 0249-6399 ISRN INRIA/RR--5184--FR+ENG



SSCRAP: Soft Synchronized Computing in Rounds for Adequate Parallelization

Mohamed Essaïdi , Isabelle Guérin Lassous , Jens Gustedt

Thème NUM — Systèmes numériques
Projets Algorille et ARES

Rapport de recherche n° 5184 — Mai 2004 — 22 pages

Abstract: The Soft Synchronized Computing in Round for Adequate Parallelization (SSCRAP) library is a C++ communication and synchronization library for coarse grained algorithms. SSCRAP is the first library supporting all known coarse grained models who are based on message passing. Indeed, it allows the implementation of BSP, GCM and PRO algorithms by supporting, at the same time, their respective execution models. Providing a high abstraction level, SSCRAP makes the real evolved communications transparent for the user and handles efficiently data exchanges and inter-process synchronizations. Thus it guarantees a good portability and ensures efficiency.

Key-words: SSCRAP, parallel and distributed computing, PRO, BSP, CGM, coarse grained communication library

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,

SSCRAP: Soft Synchronized Computing in Rounds for Adequate Parallelization

Résumé : SSCRAP (pour Soft Synchronized Computing in Round for Adequate Parallelization) est une bibliothèque développée en C++ qui offre des outils de communication et de synchronisation pour les algorithmes à gros grain. SSCRAP est la première bibliothèque à supporter tous les modèles de programmation à gros grain basés sur le passage de messages. En effet, elle permet l'implantation des algorithmes BSP, CGM et PRO en supportant simultanément leurs modèles d'exécution respectifs. En offrant un haut niveau d'abstraction, SSCRAP prend en charge efficacement les échanges et les synchronisation inter-processeurs effectifs tout en assurant leur transparence totale vis-à-vis de l'utilisateur. Elle garantie ainsi la portabilité et l'efficacité des programmes produits

Mots-clés : SSCRAP, calcul parallèle et distribué, PRO, BSP, CGM, bibliothèques de communication à gros grain

1 Introduction

To reduce the design complexity of parallel applications, several parallel models were proposed. They can be classified into two categories. The first class is referred to as fine grained models. In these models, we suppose that the size of the problem is linearly proportional to the number of processors P (i.e. $P = O(N)$ where N is the total input size). PRAM [1] [12] is the most known fine grained model. Due to its high abstraction level, the PRAM allows to point out the possible parallelism of a problem in an easy and intuitive way. However, speedup results for theoretical PRAM algorithms do not necessarily match the speedups observed on real machines and the aimed efficiency often cannot be reached [3].

With the aim to provide more realistic models than PRAM, several works tried to emphasize on so called coarse grained models. They take the real characteristics of existing platforms into account while covering at the same time as many existing parallel platforms as possible. In fact, on the one hand they assume that every processor has a substantial local memory, which is true for all commercially available multiprocessors. On the other hand, they consider communication costs (which is not the case in the PRAM) but without specifying the topology of the medium. The most prominent coarse grained models are BSP (Bulk Synchronous Parallel model) [24], LogP (Latency overhead gap Processors number) [5] and CGM (Coarse Grained Multicomputer) [7, 6] and they assume a distributed memory architectural model. That is why the communication steps required for the data distribution must be considered in the algorithmic design.

The first coarse grained model BSP introduced the *supersptep* execution model. In this model, a parallel computation consists of a sequence of phases called *superstep*. In every superstep, each processor can firstly perform computation on data present in its local memory , send and receive messages. Supersteps are separated by a synchronization barrier. At the beginning of each superstep, it guarantees that all messages sent in the previous superstep are already received.

1.1 Related work

BSP is by far the most known and used coarse grained model. Indeed, several tools and libraries were developed to simplify the implementation of BSP algorithms. The Oxford BSP library [20] is the first library implementing the BSP model. It basically provides *bulk synchronous message passing* (BSMP) routines, synchronization tools, and so-called *direct memory access* (DRMA) operations. BSMP routines implement point-to-point communications (fixed size packet to specified destination) and DRMA only allows a concurrent read access to static variables.

After several improvements and extensions, a new version of the Oxford BSP library called "BSPlib" was proposed in [19]. BSPlib provides BSMP as well as DRMA operations. Furthermore, DRMA access is not restricted to static variables. In fact, each processor can dynamicly register/unregister data which can be reached through DRMA operations. For

[1] Parallel Random Access Memory/Machine

the multi-point communications, a prototype of an interface for the collective communications [13, 2] was proposed by the BSPlib developers. However, until now no official BSPlib distribution includes this interface. Considering the success of BSPlib and the diversity of BSBlib like libraries, a standard inspired by BSPlib and called "the BSP Worldwide Standard Library" was proposed in 1996 [2].

As BSPlib, the Paderborn University BSP-Library (PUB) [1] implements the BSP Worldwide Standard. Indeed, it provides BSMP, DRMA, and bulk synchronization tools. In addition, PUB offers several other features which can be classified in four categories: collective communications, processors sub-groups management, weak synchronization, and virtual execution processors. As collectives communication tools, PUB implements broadcasting routines. For parallel algorithms allowing problem decomposition the global BSP synchronization is a real bottleneck. Even if a sub-group of processors can be affected for each sub-problem, global synchronization forbids sub-groups to evolve independently. The PUB library enables the distribution of processors in several sub-groups. Each sub-group behaves like an independent BSP machine. In several algorithms, for a given processor in a given superstep it is possible to know in advance the exact number of expected messages. PUB implements low-cost synchronization allowing blocking processors until a specified number of awaited messages is received. To make the scalability analysis and debugging of BSP program easier, the PUB library provides a set of tools allowing creation of virtual BSP processors.

The CGM model, more recent and designed as a simplification of the BSP model, allows efficient design of parallel algorithms [3, 8, 6, 9]. However, until recently and besides the work presented in this paper, CGM had no dedicated realization. It newly lays out of its own library: the CGM-lib[4]. According to the authors, CGM-lib can be also used for the implementation of BSP algorithms. CGM-lib was initially designed to handle communication for a high level interface called cgm*graph*. The cgm*graph* interface provides a tool set dedicated to the implementation of CGM graph algorithms. However, the CGMlib communication interface can be used for the implementation of other types of CGM algorithms. The CGM-lib provides point-to-point communications (BSMP like), collective communications, and CGM communication also called hRelation. In a hRelation communication, each processor indicates for each local data the destination processor for the next super step. As the user does not need to handle messages, the hRelation communication provides a higher abstraction level than classic message passing (see section 2.3 for more details).

1.2 Overview

In this paper we present the Soft Synchronized Computing in Rounds for Adequate Parallelization (SSCRAP) library which is the first library implementing CGM model. Starting from the observations collected during the development of SSCRAP and GCM algorithmic experiments, a parallel programming model was proposed to allow the design and analyzes of extensive and resource-optimal algorithms. This model called PRO [16] (Parallel Resource-Optimal computation) is based on the following ideas:

- to incorporate relative optimality as an integral part of the model;
- to measure the quality of the parallel algorithm according to its granularity.

A PRO algorithm is required to be both time- and space-optimal. As a consequence of its optimality, a PRO-algorithm always yields linear speed-up relative to a reference sequential algorithm. In addition, PRO assumes that the coarse grained communication cost only depends on P (number of processors) and the bandwidth of the considered interconnection network. In fact, the experiments based on SSCRAP show that the communication start-up and latency can both be neglected from the cost analysis as has been predicted in [16].

Considering the several similarities between the known coarse grained models based on message passing (BSP, CGM and PRO) and taking in account the various algorithmic and experimental analysis [10, 11], the main purpose of the newest SSCRAP releases is to provide a development framework covering these various models. Unlike the model dedicated library (BSPlib, PUB and CGM-lib), SSCRAP allows the implementation of the BSP, CGM and PRO algorithms and guarantees the portability and efficiency of the produced programs on a large variety of parallel and distributed platforms.

In order to describe the SSCRAP library, we first enumerate its main features in section 2. We mainly describe communication and synchronization tools and we illustrate the flexibilities which they introduce in the SSCRAP execution model possibilities. We also present the group management and other supplementary features provided by SSCRAP. Section 3 gives an overview of the implementation and the practical results of the library.

2 Features of SSCRAP

The library SSCRAP features can be classified into four categories: communication routines, synchronizations tools, process management and instrumentation. In this paper we present the fonctionnalities of SSCRAP (see [23] for details of SSCRAP's API). To highlight SSCRAP capability to support various coarse grained execution models, we first describe the considered communication and synchronization models.

2.1 Synchronization

To implement BSP and CGM models, SSCRAP provides bulk synchronization. It also implements data synchronizations which can be considered as one of its main innovations. Indeed, it provides two new synchronization types: the send synchronization and the receive synchronization. Through data synchronizations SSCRAP implicitly provides a model of data exchange based on the handover of data control. This model requires separation between the algorithmic aspect relative to only the data processing and the remaining part where communication is really carried out.

send synchronization: Each processor is blocked until the communication layer has taken over all sending messages. Thereafter local memory allocated for send buffers can be reused: data control is handed over to the communication layer.

receive synchronization: each processor is blocked until all the messages coming from the communication layer are loaded in their respective reception buffers. Then processors access to the data for their computation: data control is handed over to the processors.

Being data based, send and receive synchronizations allow "soft synchronization". Indeed, compared to bulk synchronization as used in the BSP libraries where every processor has to wait until *all* communications of *all* processors is accomplished, they avoid unnecessary idle time. Even the receive synchronization is still less restrictive than the low-cost synchronization of PUB. In fact, to be able to use the PUB low-cost synchronization, users mostly provide the number (known staticly) of awaited messages whereas SSCRAP ensures dynamically (during the execution) and in a transparent way, the management (and mainly the counting) of the messages exchanged during the *superstep*.

2.2 Communications

SSCRAP implements three different communication types: message passing, bulk communications and DRMA exchanges. For message passing, SSCRAP provides both point-to-point and collective communications routines. Thanks to data synchronization, the point-to-point interface is simplified to only asynchronous operations. In fact, if blocking receive is required, it can be carried out using an asynchronous receive followed by receive synchronization. As collective communications, SSCRAP implements broadcast (one-to-all), general broadcast (all-to-all-broadcast) and a vectorial version of all-to-all where each processor sends to all the others a data table or vector (like all-to-allv in MPI). With the aim to reduce their execution time, all collective communications integrate data synchronizations. Indeed, when such communication is evolved, a user does not have to ensure synchronization explicitly except if global synchronization is needed.

In coarse grained algorithm execution, we distinguish two types of exchanged data. The first corresponds to the local data directly used in the algorithmic treatment. In this case, the data size is proportional to the input size and relatively important. Their performance is mainly restricted by the bandwidth of the communication layer. The second type of exchanged data corresponds to control messages. These messages inform the processors about the state of the coarse grain machine, the exchanges and the distribution of the data dynamicly during the execution. Having generally a fixed and reduced size (about $O(p)$ or of $O(p^2)$), their performance is usually dominated by the latency of the communication layer. To simplify the exchange of the control data, SSCRAP implements DRMA mechanisms. Using DRAM users can specify and use shared data structures used to make the control data remotely accessible without message passing. These shared structures are managed by an CREW access control protocol to guarantee a total coherence between the various processors. Since these tools are only meant to be used for control and only once per superstep, their overhead should be small compared to the data communication.

2.3 Bulk communications

The main SSCRAP contribution in coarse grained exchange is the new communication tool called *bulk communication*. The bulk communication can be considered as abstraction level which really fits with exchange requirements of coarse grained models. The main purpose of the bulk communication is to provide a *simple* and *unique* tool to cover all the different types of data exchange.

Considering the real contribution of bulk communication (the encapsulation and the high level abstraction) in coarse grained algorithm development, CGM-lib, subsequent to SSCRAP, implements a similar feature called *hRelation* [4]. As CGM-lib ensures data exchanges through the so called "CommObject", to be able to use *hRrelation*, users must indicate for each processor which list of objects are to be sent to which processor. However, as we don't have enough information about *hRelation*, we cannot make detailed comparison between SSCRAP bulk communication and CGM-lib *hRelation*.

Implementation:

In SSCRAP bulk communications, data is represented as a global array distributed among the processors. Each processor can directly access its own sub-array. As data exchange, bulk communication allows the efficient redistribution of the whole distributed array. To use bulk communication users have just to provide two specialized methods, **identify** and **extract**, that cover the part of the communication that is application or algorithm specific. With **identify**, each processor determines the destination processor of each individual data item. **Extract** is used to transform the data item as it is on the sending side to what should be visible on the receiving side. Often, **extract** is just a simple copy operation. Commonly, these two methods are easily and directly deduced from the algorithm under consideration.

Once the **identify** and the **extract** routines are defined, SSCRAP ensures the bulk communications according to the following process:

Pre-communication:

- Using **identify**, each processor identifies the destination processor for each of its data items.
- Each processor computes the size of the data to be sent to each destination.
- The processors carry out an all-to-all exchange of the data sizes.
- Each processor, knowing the various sizes of sent and receive data, dynamically allocates its send and receive buffers.
- Each processor extracts the data to be sent to the emission buffers. Both of the **identify** and the **extract** routines are used in this step.

Communication: During this step, all the processors carry out the effective data exchange by issuing an `all-to-allv`: the sending of the data on the physical layer as well all necessary higher level operations to have the data in the desired location on the reception side.

Post-communication: At the end of the exchanges, the send and receive buffers are deallocated.

Defined in such an abstract way, the bulk communication considerably reduces the development difficulty. Indeed, after specifying **identify** and **extract**, the programmer must just invoke the bulk communication. He needs neither to specify the aimed exchange type (one-to-one, one-to-all, etc.) nor has to handle data, messages, buffers and synchronization directly as he would have to do with all of the BSP libraries or with MPI.

Example

In the following example, we highlight separation between computation and communication that is generated by the use of the general communications and which allows to isolate the algorithmic parts corresponding to the routines **identify** and **extract** easily. We underline also the algorithmic simplification introduced by the bulk communications which provides a higher abstraction level compared to the traditional message passing communications. The algorithm allows the distributed computing of tree depth. Its entry is a tree having n nodes that are randomly distributed on p processors. A parallel intuitive solution to solve this problem is described by Algorithm-1 executed by each processor in SPMD mode. We do not describe the used algorithm to compute the in-degree (detailed in [11]).

For the given example, it's easy to distinguish the algorithmic parts ensuring to the **identify** and **extract** routines rules. In fact, the two lines starting by "Indentify" and indicating for each local node the destination processor, correspond to the **identify** routine. The function providing for a given node N_i its father N_j can be considered as the **extract** routine.

The use of the bulk communications in Algorithm 2 does not only simplify the algorithm but also enables the encapsulation of all the algorithmic parts dedicated to the communication and the distribution. Indeed, in Algorithm 2, the visibility and thus the treatment which each process can carry out are restricted on its own local data. As no processor reference is used in Algorithm 2, the distribution aspects (global operations) become transparent.

2.4 Execution models

To allow the implementation of coarse grained algorithms designed in BSP, CGM and PRO models, SSCRAP must support their respective execution models. Thanks to the experimental studies of CGM algorithms implementations over SSCRAP, we note that we can introduce a real flexibility in the coarse grained execution model (compared to BSP and CGM) implicitly integrated to the PRO model [16, 11]. To reproduce such possibilities in the implementation process, we defined the communication model which enables the description of the execution model of SSCRAP. It is described as follow:

- Separation between sending and receiving operation: unlike the BSP and CGM models which describe data exchange as an atomic operation.
- All exchange operations are asynchronous.

Algorithm 1: Tree Depth in parallel SPMD execution

```
Input: Each processor P handles n/p nodes of tree T (nodes are randomly distributed)
Output: depth of T (initialized to 0)
begin
    Compute the in-degree deg_in(v) for each local node v;
    while ∃ local node v having deg_in(v) = 0 do
        foreach local node v having deg_in(v) = 0 do
            Identify node u father of v;
            Identify processor P_i handling u;
            Add the identifier u to the message Emission_i;
            Delete local node v;
        end
        foreach P_i, i = 1, ..., p do
            Send Emission_i to processor P_i;
            Receive Reception_i from processor P_i;
            Concatenate Reception_i to Reception;
        end
        foreach local node v in Reception do
            deg_in(v) = deg_in(v) − 1;
        end
        depth = depth + 1;
    end
end
```

- In each superstep, processors can send at most one message to each destination processor (according to PRO model).
- During send operations, each message is stamped with the number of the current superstep.
- Processors can only receive messages stamped with the same number as their current superstep.

The SSCRAP execution model is then described as follows:

- Execution is performed according to Supersteps.
- Superstep contains in any order: one computation step, one sending step and one receiving step
- Global synchronization is not obligatory.

Algorithm 2: Tree Depth with bulk communications

Input: Each processor P handles n/p nodes of tree T (nodes are randomly distributed)

Output: *depth* of T (initialized to 0)

begin

  Compute the in-degree $deg_in(v)$ for each node v;

  while $\exists$ *local node* v *having* $deg_in(v) = 0$ **do**

    Bulk_Communication(Input: local nodes;
    Output: *Reception* (the same array than algorithm 1);
)

    foreach *local node* v *in Reception* **do**

      $deg_in(v) = deg_in(v) - 1$;

    $depth = depth + 1$;

end

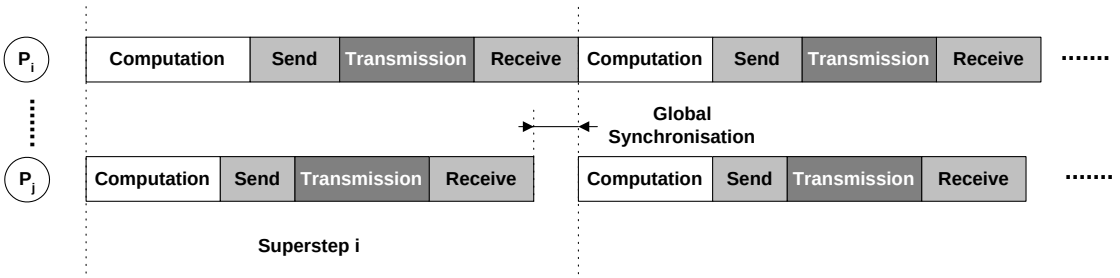


Figure 1: BSP/CGM execution model

- Data synchronization (send and receive) must be used before the end of each superstep.
- Data received during a superstep are not available (for local computation) until the following superstep.

The execution model as defined above is not only able to implement the CGM and BSP execution model (see Figure 1) but also allows the implementation of new execution models being able, for example, to ensure a covering between communication and computation (see Figure 2). Unlike the BSP and CGM models where superstep is an exact sequence of local computation and exchange operation, it enables a free composition (in any order) of one computation step, one sending step and one receiving step for a superstep execution. In

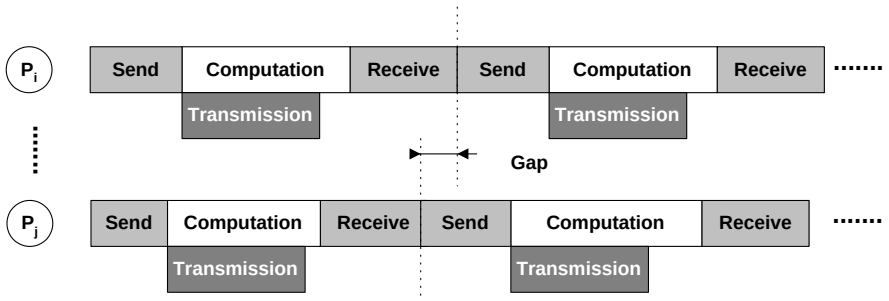


Figure 2: Execution model with data synchronization and computation/transmission recovering

both Figures 1 and 2 we distinguish between the required CPU time for send and receive operations, and the network time needed for data transmission.

If the aimed algorithm and the considered architecture allow the use of the execution model described in figure 2, the execution time required by superstep j over the processor i is:

$$T^{ij}_{SupS} = T^{ij}_{Send} + \max(T^{ij}_{Comm}, T^{ij}_{Comp}) + T^{ij}_{Recv} = T^{ij}_{CGM} - \min(T^{ij}_{Comm}, T^{ij}_{Comp})$$

As CGM model requires a global synchronization, the total execution time of CGM algorithm is $T_{pCGM} = \sum_{j=1}^{\lambda} \max_{i=1}^{p} \{T^{ij}_{CGM}\}$ (where λ is the number of supersteps). Using data synchronization, the total execution time of the proposed execution model is given by $T_{PROP} = \max_{i=1}^{p} \{\sum_{j=1}^{\lambda} T^{ij}_{SupS}\}$. Indeed we obtain:

$$T_{PROP} \leq T_{CGM} - \min_{i=1}^{p} \left\{ \sum_{j=1}^{\lambda} \min(T^{ij}_{Comm}, T^{ij}_{Comp}) \right\}$$

2.5 Group management

BSP and CGM models define an execution model where all processors communicate and must be synchronized at the end of each superstep. Being able to break up a BSP machine, for example, in several sub-BSP machines each affected to a part of total treatment, has several advantages:

- Several algorithmic approaches are based on initial problem decomposition in various subproblems which can be treated separately. The introduction of the possibility of sub-division, eases the implementation of these approaches.

- In some algorithms, it is possible to distinguish groups of processors which frequently communicate with each other. Separation in sub-groups allows the simultaneously use of various exchange types (point-to-point, one-to-all, bulk, etc.) overall processors.
- The sub-group subdivision introduces a relative asynchronism. Indeed, the range of bulk synchronization being limited to processors sub-group, processors having a high computation load do not have to be synchronized with those requiring frequent exchanges.

Unlike CGM and BSP, PRO implicitly integrates the possibility of hierarchical decomposition of a PRO machine in several sub-machines. To discharge the user from the management aspects relative to such decompositions, SSCRAP provides tools allowing mainly the subdivision of a group into two or several sub-groups and the fusion of the sub-groups to reconstitute the original one. These operations are carried out according to the following rules:

- A group can correspond to the initial processor group or a sub-group of processors.
- During execution, states of all effective groups can be represented by a tree in which the root corresponds to the initial processors group. Initially, the tree only contains the root.
- Only leaves groups can be partitioned into several sub-groups. New sub-groups are represented by leaves that are children of the original group node.
- Merg allows the reconstitution of a parent group from its sub-groups. Fusion is only possible when all sub-groups are leaves.
- The visibility of each processor is limited to its current leaf group. Each leaf group behaves like an independent coarse grain machine. Thus, a processor of a given group can communicate or be synchronized only with the members of its current group.

All group management aspects are completely transparent to the user. Indeed, to be able to distribute processors in groups, the user must only give an identifier for each group and specify for each processor of the initial group its new sub-group identifier. Thus, all the communications and synchronizations are automatically restricted on the sub-groups level.

The processor group management provided by SSCRAP is more flexible and easier to use than the one proposed by PUB. In fact, all PUB group subdivision is based on the initial groups of processors, thus hierarchical distributions (subdivision of sub-group in groups even smaller) are not permitted. In addition, PUB imposes a preliminary (static) knowledge of the processors distribution whereas with SSCRAP, it can be both, hierarchical and conditioned, according to the algorithmic behavior during the execution (dynamic distribution).

2.6 Supplementary features

SSCRAP also includes a tool set allowing for the measurement of various execution parameters. The main parameters for each processor are provided: the total execution time, the total number of effective CPU cycles, the computing time (algorithm), the communication time (SSCRAP), the number of exchanged messages, the consumed bandwidth and the used swap size.

Recently, SSCRAP now extends to so-called external memory architectures [17]. Such a setting allows treating problems as large that they don't fit into the memory of conventional workstations and maybe even mainframes. Providing tools (based on memory mapping) to efficiently externalize the data on large storage devices, SSCRAP allowes us to extent the use of certain types of parallel programs to the setting of out-of-core computation [17].

SSCRAP also provides tools to carry out a random distribution of the program input. Based on an algorithm of distributed generation of permutations that is also implemented with SSCRAP, it facilitates the generation of test sets for coarse grained programs.

2.7 Features summary and comparison

Features	BSPlib	PUB	CGM-lib	SSCRAP
Comm. one-to-one	BSMP	BSMP	Comm. Object	MPI like
Comm. one-to-all	NA	$\surd$	$\surd$	$\surd$
Comm. all-to-all	NA	NA	$\surd$	$\surd$
Comm. all-to-allv	NA	NA	*hRelation*	bulk comm.
Comm. DRMA	$\surd$	$\surd$	NA	$\surd$
Global Sync.	$\surd$	$\surd$	$\surd$	$\surd$
Data Sync.	NA	receive(param)	NA	receive+send
Sub-groups	NA	simple	simple	Hierarchic
External memory	NA	NA	NA	$\surd$

Table 1: Coarse grained library features

Table 1 summarizes the features of the various coarse grained libraries that we considered. The libraries are classified (from the left) according to the number and to the nature of their features. We note that SSCRAP is characterized by its MPI like interface for message passing, its bulk communications, its data synchronization and its hierarchical group management. Although CGM-lib provides the *hRelation* function to ensure the global exchanges, the SSCRAP bulk communication has a higher abstraction level. In fact, as the user must provide a correspondence array between data and destination processors, *hRelation* does not allow the real algorithmic separation between local computations and global operations. Considering data synchronization, PUB requires a preliminary knowledge of the number awaited messages to ensure the synchronization receive. In the contrary, SSCRAP internally manages the number of exchanged messages and thus ensures and simplifies the use of data synchronization (send and receive).

3 Library design, implementation and performance

The SSCRAP library provides development tools for the algorithms described in the CGM, BSP and PRO models. It is developed in C++ language and takes advantage of objects programming assets: classes, overloading, inheritance and genericity. Considering the ceaseless improvement of the C++ compilers, it benefits from the most recent functionalities, and uses available optimizations. This mainly explains the good experimental results highlighted in the following.

To ensure the basic communication functions, and to grant SSCRAP good portability and efficiency, we implement low level communication tools using MPI libraries and POSIX threads. These tools are located at the deepest abstraction level of the library and are completely transparent and inaccessible to the user. To ensure the separation between the user interface and the low level communication interfaces we introduced a hybrid communication layer. Its main role is to allow grafting new communication interfaces without having to modify upper library layers while benefitting from the specific capacity of each new combination of interface and architecture. Indeed, using shared memory interface (based on POSIX threads), the hybrid layer enabled us to introduce new mechanisms of buffer management. Compared to MPI, it reduces the number of copies considerably. In bulk communications, for example, a direct extraction of the data into the remote memory of a destination processor is provided.

The use of the standards and portable libraries as MPI and POSIX threads, confers to SSCRAP a good portability. Indeed, SSCRAP is likely to function on practically all *Unix-like* platforms. From the point of view of efficiency, SSCRAP provides good performances on both shared memory (POSIX threads) and distributed memory (MPI) architectures.

The practical results obtained with the implementation of several typical algorithms emphasize the real efficiency and the high scalability of SSCRAP. Considered as a valid implementation tool, it is used to carry out fine experimental analyses of several coarse grained algorithms. This section provides an overview of the obtained practical results. First, we introduce the experimental environment: algorithms and platforms. Next, we illustrate and analyze the experimental results.

3.1 Experimental environment

To highlight the contribution of SSCRAP to coarse grained parallel programming, we used it to efficiently implement several algorithms [11, 18]. In this paper, we only consider two typical problems: *List Ranking* and *Sorting*. Many applications are based on routines that rank input elements using either variants of Sorting or List Ranking. List Ranking has a chained list of nodes as input. Each node knows its successor node as well as the distance which separates these two nodes. Solving the List Ranking problems consists in computing for each node the distance which separates it from the last node in the list. In contrast to the known theoretical complexity this problem is notoriously difficult to implement with acceptable speedups on few processors, see e.g. [22].

For tests, we implemented the nondeterministic CGM algorithm proposed in [15]. It is based on a recursive computation of independent subsets. A independent subset I of the element list L is a part of L for which no couple of elements are neighbours on L. For sorting we implemented the algorithm proposed in [14], based on over-sampling and using a variant of QuickSort for local sorting.

With the above mentioned POSIX threads interface, SSCRAP practically supports SMP (Symmetric Multi-Processors) platforms as well as DSM (Distributed Shared Memory) platforms. In this paper we consider two different DSM machines. The first one is an SGI Origin 3000 powered with 56 RISC R1600 64 bit processors (at 700Mhz) and managing 42 GB of main memory. The second is a SunFire 6800 machine having 24 ULTRA SPARC III 64 bit processors (at 900Mhz) and 24 GB of main memory. The SGI machine runs the IRIX64 v6.5. system and the SunFire is under Solaris Operating Environment V8.

For distributed memory tests, we consider two different types of cluster. Both of these clusters use Linux 2.4.2 as their operating system. The first one is a large PC cluster (Icluster at Grenoble) with about 200 nodes. The nodes are fairly distributed among five 100 Mb Ethernet branches which are interconnected by five 1 Gb switches mesh. Each node consists of a PC desktop powered by a 733 MHz Coppermine INTEL Pentium III processor with 256 MB SDRAM PC100 local memory at 800 MB/s bandwidth and 10 ns latency. The second architecture ("Albus" at Nancy) is a cluster composed of 8 bi-AMD Athlon MP 1500+(1333 MHz) SMP nodes. Every of these nodes has 1.0 GB 2100 DDR-SDRAM memory providing 2,1 Gb/s bandwidth at 6 ns latency. For the interconnection, each node is equipped with two different interfaces: Ethernet 100 Mb and Myrinet 2000 M3F. Both of these interfaces ensure interconnection respectively through a switched Ethernet network and a switched optical Myrinet network. The Myrinet card installed on 64b PCI port at 66 MHz provides 528 MB/s (half-duplex) at 9 μs latency on DMA (Direct Memory Access).

For the framework of our experimental studies and for the result analysis that follows we note:

1. The executions on one processor correspond to those of the optimal sequential algorithm and not to those of the parallel algorithm on mono-processor configuration. Then all the obtained accelerations are not relative accelerations but absolute ones.
2. The number of items qualify:
   - For list ranking: the number of the list elements,
   - For Sorting: the number of elements to be sorted.
3. To be less dependent on the particular architecture and to ease the comparison to the sequential setting, all provided results are given in the number of clock cycles per item. The measured times are wall clock times of the parallel execution. To facilitat the comparaison for different order o magnitude, the scales are algorithmic.
4. For each quadruplet (algorithm, platform, number of items, number of processors), the given results correspond to the average of 10 effective tests. We note here, that in all cases, the variance is very slight.

3.2 Result analysis

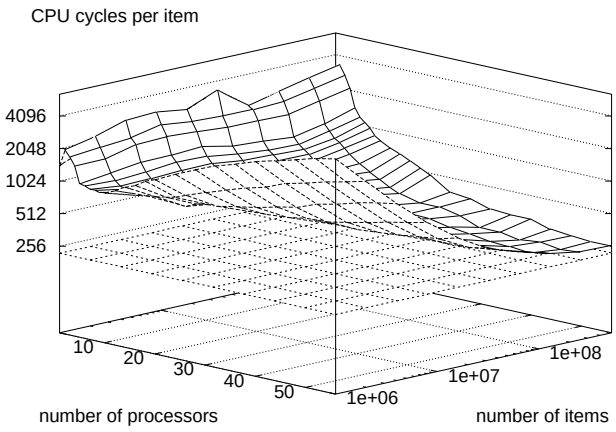


Figure 3: List ranking on Origin 3000 DSM architecture: Overall results

Figure 3 shows an overview of the results of List Ranking obtained with the SGI machine. We used all available processors (from 1 to 56 processors) to rank lists having 1 to 512 million elements. According to Figure 3, the algorithm implemented with SSCRAP has a typical coarse grained behavior. In fact we notice that:

1. For a given number of items, there is a number of processors beyond which we can not obtain acceleration any more: For example for 10^6 items, the use of more than 7 processors does not reduce the running time but it grows considerably.

2. For a given number of processors, increasing the number of items noticeably reduces the number of required cycles: for 56 processors, the number of cycles is inversely proportional to the number of items.

The granularity that is considered by the coarse grained models, corresponds to an architectural granularity which is defined as the relationship between memory size (or input size) and the number of processor. The performance taking down observed for small entry size represents borders of coarse grained context.

Figure 4 represents the overall results of List Ranking obtained on Icluster using 1 to 32 processors and on 5 to 220 million elements. We notice that we obtain good performance and also that we achieve a good scalability: the algorithm scales up to efficiently use the

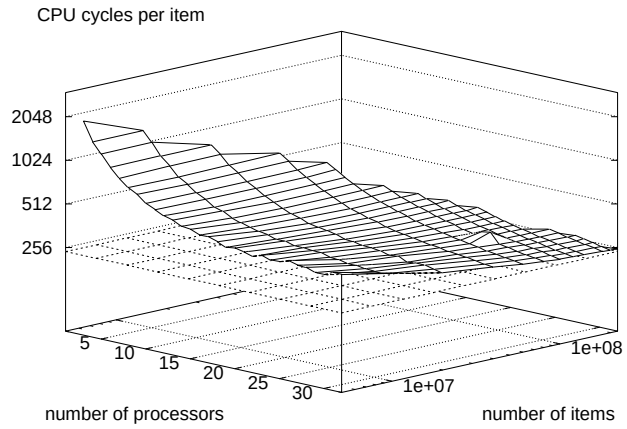


Figure 4: List ranking on Icluster: Overall results

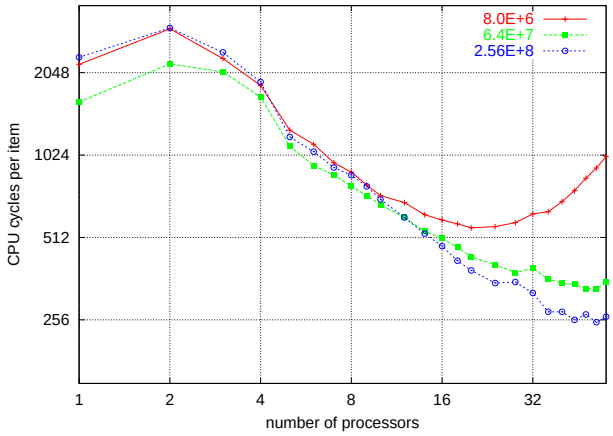


Figure 5: List ranking on Origin 3000 DSM architecture: 8, 64 and 256 millions input size

available memory on all machines. The performances decrease due to the exit of the coarse grained context is less visible than on the Origin.

To carry out finer observation, in Figure 5 we consider only three input sizes: 8, 64 and 256 million list elements. In this figure, we can distinguish three different areas:

- From 1 to 2 processors, an increase of the running time occures between the mono-processor execution (sequential algorithm) and the execution on 2 processors (parallel algorithms) and represents the additional cost of the parallelization.
- There is an interval of a number of processors where the number of cycles is independent of the input size and linearly decreases in $1/p$: Since we are here in a coarse grained context, we obtain a regular and linear behaviors and we achieve a very good acceleration.
- There is a number of processors from which the number of cycles by item does not decrease linearly any more: Depending on the considered input size and mainly visible for small one (8 millions), this area corresponds to executions that leave the coarse grained context.

With Figure 5 we can also deduce that the implementation of the list ranking algorithm with SSCRAP provides a very good acceleration and maybe the best results ever reached. In fact, with the other available list ranking implementations, at least 10 processors are required to reach the best sequential execution time but SSCRAP only needs at most 4 processors. The only exception to that seems to be the work of Sibeyn [21] which emphasizes on optimizing for small numbers of processors. SSCRAP providing better acceleration and scalability without requiring these optimizations could perhaps take advantage of this work for this range of processors.

Figures 6 and 7 show the combined results of respectively list ranking and sorting algorithms. Each figure gathers the results of all considerd platforms for the largest computed input size. For both algorithms, we can clearly notice that behavior is noticibly similar on the various platforms. Indeed we can deduce that:

1. behavior of SSCRAP is independent on both implemented algorithm and used platform.
2. The overhead that SSCRAP imposes is neglectable and the performance is linear compared to th input size.

According to these observation, SSCRAP allows to make realistic predictions of the algorithmic behaviors on other platforms. Providing an architectural and algorithmic independent behavior, SSCRAP can also be used as a valid tool to carry out fine experimental studies and comparison of coarse grained algorithms and parallel architectures.

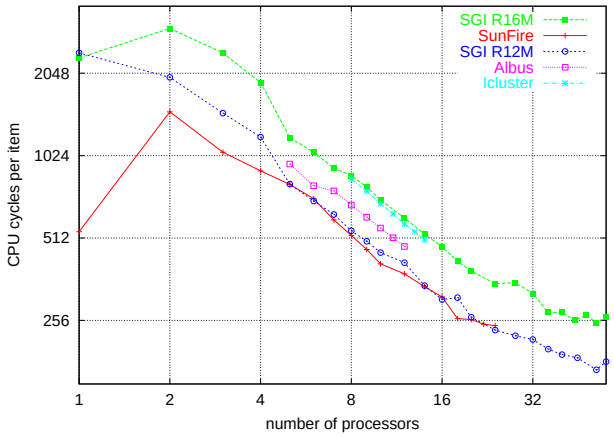


Figure 6: List Ranking: All platforms combined results

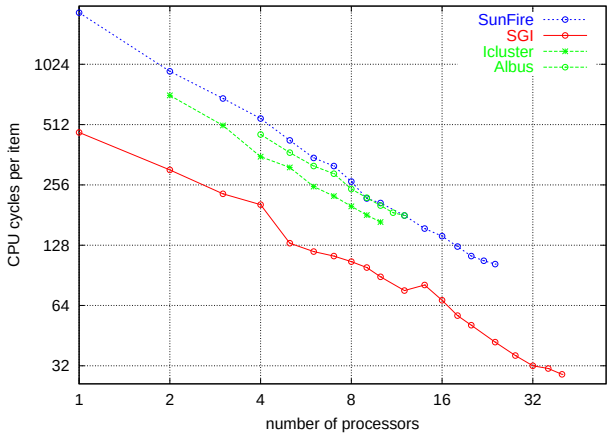


Figure 7: Sorting: All platforms combined results

4 Conclusion and futur work

SSCRAP enables the implementation of parallel algorithms designed in BSP, CGM or PRO model. It mainly ensures the inter-processors communication and synchronization by providing a complete and high level user interface. We consider bulk communication to be the most appropriate communication tool for the coarse grained paradigm. Thereby to our opinion, the main contribution of SSCRAP is the high level of abstraction and implementation performances provided by such communication. Practically, with SSCRAP we were able to efficiently implement communications for coarse grained algorithms. We support several platforms and we achieve real scalability, portability and predictability.

Thanks to its efficiency and its architectural and algorithmic independent behavior, SSCRAP is currently used to carry out accurate experimental studies for several coarse grained algorithms and for the coarse grained models them-self. The studies that have been realized until now will soon be subject to a separate publication.

Even if SSCRAP is validated as an environment for the development of coarse grained parallel algorithms we plan to promote it as an efficient and portable communication interface for real and tidy applications. Indeed, recently started works aim to implement the parallel language based on agents called PARCEL [25] over SSCRAP. This will ensures on the one hand the portability of PARCEL and the efficiency of its evolved communications. On the other hand, and as PARCEL implements several applications covering several domain (physics simulation, cortical networks, etc.), SSCRAP will be validated in a real application and multi-domain framework.

To improve portability of SSCRAP, the first we plan to interface SSCRAP with hierarchical architectures (networks of multiprocessors). As SSCRAP currently is restricted to homogeneous platforms, the second one is to extend its support to the heterogeneous distributed architecture.

References

[1] O. Bonorden, B. Juulink, I. von Otto, 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*, April 1999.

[2] BSP worldwide standard. `http://www.bsp-worldwide.org/bspwwact.htm`.

[3] E. Caceres, F. Dehne, A. Ferreira, and P. Flocchini. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Automata, Languages and Programming, 24th International Colloquium*, pages 390–400, Bologna, Italy, July 1997.

[4] A. Chan and F. Dehne. Cgmgraph/cgmlib: Implementating and testing cgm graph algorithms on pc cluster. private communication, 2003.

[5] D. E. Culler, R. M. Karp, D. A. D. A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, July 1993.

[6] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Khokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, Santa Barbara, California, July 1995. ACM SIGACT/SIGARCH and EATCS.

[7] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. Technical Report RR-1819, Inria, Institut National de Recherche en Informatique et en Automatique, December 1992.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 9th Annual Symposium on Computational Geometry (SCG '93)*, pages 298–307, San Diego, CA, USA, May 1993. ACM Press.

[9] F. Dehne, C. Kenyon, and A. Fabri. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. IEEE Symposium on Parallel and Distributed Processing*, pages 586–593. IEEE, 1994.

[10] M. Essaïdi, I. Guérin Lassous, and J. Gustedt. SSCRAP: An environment for coarse grained algorithms. In *Parallel and Distributed Computing and Systems (PDCS 2002)*, Cambridge, USA, November 2002.

[11] Mohamed Essaïdi. *Échange de données pour le parallélisme à gros grain.* PhD thesis, Université Henri Poincaré Nancy 1, Fevrier 2004.

[12] S. Fortune and J. Wyllie. Parallelism in random access machines. In ACM, editor, *Conference record of the tenth annual ACM Symposium on Theory of Computing: papers presented at the Symposium, San Diego, California, May 1–3, 1978*, pages 114–118, New York, NY, USA, 1978. ACM Press.

[13] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, August 1994.

[14] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[15] I. Guérin Lassous and J. Gustedt. Portable List Ranking: an experimental study. *ACM Journal of Exprimental Algorithmics*, 22, 2002.

[16] J. Gustedt, J. A. Telle, A. H. Gebremedhin, and I. Guérin Lassous. PRO: a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance*, pages 106–113. IEEE, 2002.

[17] Jens Gustedt. Towards realistic implementations of external memory algorithms using a coarse grained paradigm. In *International Conference on Computer Science and its Applications - ICCSA'2003, Montréal, Canada*, volume 2668 of *Lecture Notes in Computer Science*, pages 269–278. Springer, Feb 2003.

[18] Isabelle Guérin Lassous. *Algorithmes parallèles de traitement de graphes : une approche basée sur l'analyse expérimentale.* PhD thesis, Universitée Paris 7, Janvier 1999.

[19] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. santilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[20] R. Miller. A library for bulk-synchronous parallel programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.

[21] Jop F. Sibeyn. Better trade-offs for parallel list ranking. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 221–230, 1997.

[22] Jop F. Sibeyn. Ultimate Parallel List Ranking. In *Proceedings of the 6th Conference on High Performance Computing*, pages 191–201, 1999.

[23] Sscrap user guide. `http://www.loria.fr/ gustedt/sscrap/sscrap-doxy/`.

[24] L. G. Valiant. A bridging model for parallel computations. *Com. of the ACM*, 33(8):103–111, August 1990.

[25] Stéphane Vialle, Yannick Lallement, and Thierry Cornu. Design and implementation of a parallel cellular language for MIMD architectures. *Computer Languages*, 24(3):125–153, 1998.



Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399