External Memory Algorithms using a Coarse Grained Paradigm

Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

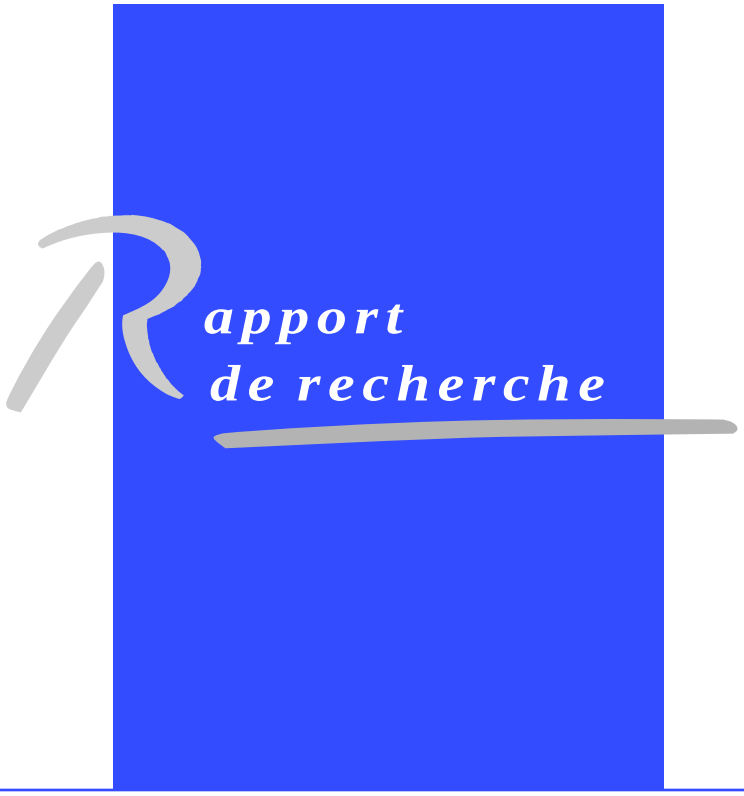
External Memory Algorithms using a Coarse Grained Paradigm

Jens Gustedt

No 5142

Mars 2004

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--5142--FR+ENG

External Memory Algorithms using a Coarse Grained Paradigm

Jens Gustedt*

Thème 1 — Réseaux et systèmes
Équipe AlGorille

Rapport de recherche n°5142 — Mars 2004 — 18 pages

Abstract: We present a simple framework that allows for the use of algorithms in external memory settings that originally were designed for coarse grained parallel architectures. This framework is an extension of the model PRO as described by Gebremedhin et al. (2002). Compared to the commonly used IO model it is trading a slight (but practically not important) restriction on the internal versus external memory size for an independence of the latency of the underlying hardware. Thereby the performance of an algorithm that is described in this model will be bound to only two parameters, namely computing time and bandwidth requirements. To prove the usefulness of this setting we also describe an extension to SSCRAP, our C++ environment for the development of algorithms on coarse grained architectures, that allows for easy execution of programs in an external memory setting. Our environment is well suited for regular as well as irregular problems and scales from low end PCs to high end clusters and mainframe technology. It allows running algorithms designed on a high level of abstraction in one of the known coarse grained parallel models without modification in an external memory setting. The first tests presented here in this paper show a very efficient behavior in the context of out-of-core computation (mapping memory to disk files).

Key-words: Coarse grained parallel models, external memory algorithms, experiments.

(Résumé : tsvp)

* INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr – Phone: +33 3 83 59 30 90

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : 03 87 20 35 00 - International : +33 3 87 20 35 00
Télécopie : 03 87 76 39 77 - International : +33 3 87 76 39 77

Algorithmes à mémoire externe utilisant un paradigme à gros grain

Résumé : Nous présentons un simple cadre qui permet l'utilisation d'algorithmes qui ont étés conçus pour des architectures à gros grain pour des environnements à mémoire externe. Ce cadre étend le modèle PRO comme il a été présenté par Gebremedhin et al. (2002). Comparé au modèle ES habituel, il impose une légère restriction sur le quotient entre mémoire interne et externe, mais qui est sans importance pratique. Par contre, il permet de négligé la latence du dispositif externe, qui est le paramètre le plus contraignant d'une telle architecture. Pour prouvé l'utilité de cet approche nous présentons une extension de SSCRAP, notre environnement C++ pour le développement d'algorithmes pour des architectures à gros grain, qui permet une exécution facile de programmes dans le cadre d'utilisation de mémoire externe. Notre environnement est bien adapté à des problèmes réguliers et irréguliers et fonctionne aussi bien au PC bas de gamme que sur des grappes haut de gamme et des super-calculateurs. Sans aucune modification du code, il permet de faire tourner des algorithmes qui ont étés conçus pour un des modèles parallèles à gros grain avec de la mémoire externe. Les premiers tests présentés ici montrent un comportement très efficace dans le contexte du calcul « out of core » (affecter de la mémoire à des fichiers sur disque dur).

Mots-clé : modèles parallèles à gros grain, algorithmes à mémoire externe, expérimentation

1 Introduction an Overview

So-called external memory (or out-of-core) settings describe problem solving environments that are specialized in treating problems as large that they don't fit into the memory of conventional workstations and maybe even mainframes. Their main feature is to externalize the data on some large storage device, usually a big disk array, and to only access parts this data at any moment during the execution of an algorithm. In such a setting IO efficiency of an algorithm that is to be executed is crucial : the running time is easily dominated by the access to the external storage media.

In the mid-nineties several authors, see e.g Cormen and Goodrich [1996], Dehne et al. [1997], developed a connection between this type of computation and BSP-like models of parallel computation. The main idea is to *simulate a parallel computation* of several processors on one single processor. By that they are able to enforce a good data locality and to give good estimates for the amount of communication that such a simulated parallel algorithm produces.

These proposals for extending a parallel computation model for external memory problems are in contrast to attempts of formulating native computation models for this context. The most important native model that is now well accepted is the one of Vitter and Shriver [1994]. Its main characteristic is that it tries to account for IO of larger units than classical computation models, named blocks. These blocks represent the minimal unit of data that is transfered during an IO and by that this model covers an important feature of modern architectures.

On the other hand this IO model neglects another important issue that is already crucial in existing hardware and that will probably become even more important in the future : the distinction of *latency* and *bandwidth* as restrictions for communication. Whereas hardware with high bandwidth is commonly available for a moderate cost, low latency hardware is only available in specialized equipment that is orders of magnitude more expensive. Also, latency has a physical bound that is imposed by the finiteness of the speed of light : the wavelength of a signal at 1 GHz is of about 30 cm so any (physically) external storage device will be bound to a latency in the order of magnitude of a nanosecond. As far as we know, no such principal restrictions apply to bandwidth.

One of the issues in this paper is to prove that latency issues can be neglected when using a fairly general model for coarse grained parallel computing (named PRO, see Gebremedhin et al. [2002]) for which we will simulate the parallel algorithms in an external memory environment. This approach is not as general as the native IO model since it imposes a restriction about the relative sizes between internal and external memory, namely that the external memory is bound to the square of the internal memory. But in reality, this restriction is not very much relevant if we compare the sizes of existing hardware : internal memory is at some hundred Mega to some GigaBytes ($2^{29} - 2^{32}$ bytes) and external memory up to some Tera or PetaBytes (2^{50} bytes).

Such a setting also has the advantage that lower bounds on IO as they were given by Chiang et al. [1995] do not apply : in the setting we consider the three basic measurement functions for IO complexity *scan(n)*, *perm(n)* and *sort(n)* coincide[1]. This coincidence is not accidental, but build into PRO : PRO enforces that any processor has sufficiently many resources to communicate with all of its colleagues at any given time.

The other issue of this paper is to advance in the unification approach as it was suggested by Cormen and Goodrich [1996] : We give a modelling framework in terms of simulation of parallel computation that covers a large variety of known tractability results. At the same time we take care that the parameter space that an algorithm designer has to handle doesn't explode. Again in following

[1] *perm(n)* and *sort(n)* will still be different in terms of their computational complexity, though.

PRO, we give some simple to follow constraints. Once these constraints are fulfilled, essentially two parameters remain that describe the performance of the resulting external memory algorithm :

- computing cost of the parallel algorithm
- bandwidth on the storage device.

This theoretical framework is presented in Section 2.

Whereas such an approach might be convincing on a theoretical level, its efficient and competitive implementation is quite challenging in practice. In particular, it needs software that induces as less overhead as possible by itself. Up to now, it seems that this has only been provided by software specialized in IO efficient implementations.

Currently, with our library SSCRAP, see Essaïdi et al. [2002], we reached a level of scalability that let us hope that it could now be possible to attain high efficiency in both (or even mixed) contexts :

- SSCRAP can run hundreds of "processors" (as POSIX threads) on a single machine (mainframe or not) without loosing on its performance,
- It can handle problem instances efficiently that may exceed the size of the address space of an individual hardware processor.

In fact, with some relatively small add-ons to SSCRAP we were able to provide such a framework. It was tested successfully on some typical hardware, PC with some GB of free disk.

The main add-on that was integrated into SSCRAP was a consequent implementation of an abstraction between the *data* of a process execution and the memory of a processor. The programmer acts upon these on two different levels :

- with a sort of *handle* on some data array which is an abstract object that is common to all SSCRAP processors.
- with a map of its (local) part of that data into the address space of the SSCRAP processor, accessible as a conventional pointer.

Another add-on was the possibility to fix a maximal number of processors (ie threads) that should be executed concurrently. With these add-ons, simple environment variables `SSCRAP_MAP_FILE` and `SSCRAP_SERIALIZE` allow for a runtime control of the program behavior.

In the present paper we present promising test with three different problems and algorithms :

sorting, for a problem that has a non-linear lower bound in terms of computation, but allows for only linear amount of streaming IO when doing reasonable assumptions on the granularity on the setting.

permutation generation at random, for a problem that is linear, has a highly random memory access pattern, but where the needs of computation (random numbers) and memory access equilibrate to a certain extent.

list ranking, for a problem that is linear, has a highly random memory access pattern and where this memory access pattern completely dominates execution.

2 PRO and the Simulation of Parallel Computation

In this section will present PRO and its simulation in an external memory setting. From that description we will deduce a main feature of such a simulation, namely that it permits to abstract from latency problems of the underlying hardware and lets us concentrate on the bandwidth as the main characteristic (and restriction).

The following theorem gives the general framework in which our approach is guaranteed to work. Such a statement would not hold for every parallel model but is due to the special features of the model

PRO, of which we will recall the essentials needed in our context some paragraphs below. In addition to PRO, we also need to bind the amount of random access address space that the computation will need. This notion of "RAM-awareness" will also be explained further down.

Theorem 1 *Let $\mathcal{P}$ be a problem that has a RAM-aware PRO algorithm $\mathcal{A}$ on $p = Grain(n)$ processors that solves $\mathcal{P}$ for input size n with a total cost over all processors of $T(n)$ CPU time and $S(n)$ space. Then $\mathcal{A}$ can be simulated on an external memory machine with the following requirements of system resources :*

$$\textit{computing unit} \begin{cases} O(T(n)) & \textit{computation time} \\ O\left(\frac{S(n)}{Grain(n)} + Grain(n)\right) & \textit{memory} \end{cases}$$

$$\textit{external storage unit} \begin{cases} O(S(n)) & \textit{memory} \\ O(T(n)) & \textit{bandwidth} \\ o(T(n)) & \textit{latency.} \end{cases}$$

The precision of the terms in this theorem, the details of the simulation and finally the proof of the theorem will cover the remainder of this section.

Observe that the function $Grain(n)$, "*grain*" or "*granularity*", that in the parallel setting describes the relation between the number of processors and the input size plays a crucial role in the quality of the simulation as an external memory algorithm. It achieves an optimum (imposes the least restrictions) in case $Grain(n) \approx \sqrt{S(n)}$ which also corresponds to the optimal value in the parallel setting.

2.1 PRO and random memory access

We place ourselves in the framework of coarse grained parallel algorithms, or to be more precise, of algorithms for coarse grained *architectures*. This family of models all inherit from Valiant [1990], which was the first and most successful to propose such a framework. The variant that we will mainly consider in this paper is PRO, Gebremedhin et al. [2002], since it has the advantage of being relatively simple and allows very efficient implementations with accurate predictions of running times, Essaïdi et al. [2002].

This "coarseness" has to be clearly distinguished from algorithmic coarseness which is usually understood as the quotient between computation and communication time of an algorithm. Usually algorithms that are hard in an external memory context are fine grained in this algorithmic sense, in that they achieve sequential RAM solutions that have a low (mainly constant) ratio between pure computation and data access.

A PRO algorithm is defined in view of
- an algorithmic problem $\mathcal{P}$
- a problem instance of size n
- a sequential algorithm $\mathcal{A}$ that solves $\mathcal{P}$ on all instances of size n
  - in time $T_{\mathcal{A}}(n)$
  - within a space of $S_{\mathcal{A}}(n)$.

It can be described as the decomposition of different submodels :

architecture : A machine is homogeneously composed of $p = Grain(n)$ distributed processors each having an amount of local memory of

$$M(n) = O\left(\frac{S_{\mathcal{A}}(n)}{p}\right). \tag{1}$$

execution : An algorithm runs in *supersteps* in which each processor may compute as much as necessary and in which he will send out and receive at most one (potentially large) message from each other processor. The amount of data that is sent and received in each superstep by each processor is limited by $M(n)$. The amount of supersteps $\lambda(n)$ the algorithm performs is bound by

$$\lambda(n) = o\left(\frac{T_{\mathcal{A}}(n)}{p^2}\right). \tag{2}$$

The duration of a superstep is given by the maximal duration on any of the processors. In contrast to the CGM model, see Dehne et al. [1996], different supersteps can have a different duration and in particular the duration may be smaller than $M(n)$.

cost : All use of system resources *must* be asymptotically optimal, in particular it must have a linear speed up and the sum of all running times must be

$$T(n) = O(T_{\mathcal{A}}(n)). \tag{3}$$

Different PRO algorithms that solve the same problem can then be compared by means of $Grain(n)$: the higher we may choose $Grain(n)$, the better. But this comparison aspect is of minor importance for this paper, we will focus on PRO being a *performance certificate* for algorithms.

Also, a detailed discussion of the different parts of the model goes beyond the scope of this paper, but observe that the conditions on the supersteps impose that in each superstep each processor is involved in at most $2p-2$ communications. Together with (2) this implies in particular that we may neglect any overhead that might be generated by the supersteps themselves.

In order to capture the behavior of algorithms with respect to random memory access, we introduce an additional parameter $R(n)$. Let $R_{i,j}(n)$ be the maximum amount of memory that the system has to map into the address space of processor j in superstep i. If we suppose that the algorithm has random access to the whole memory we have that $R_{i,j}(n) = M(n)$ but if we are able to reduce the size of the memory to which we access during a particular superstep i this value may be much smaller.

$R(n)$ is now defined as the sum over all $R_{i,j}$ for all i and j and we refer to it as the *random access penalty* or *RAM-penalty*. We call a PRO algorithm *RAM-aware* if $R(n) = O(T(n)) = O(T_{\mathcal{A}}(n))$.

This cost function $R(n)$ (which is a bit special) measures in a sense the sum of random memory accesses that could correspond to page faults of a coarse grained algorithm over time. This is a finer measure than the classical memory requirement $M(n)$ which is just the maximum of such simultaneous memory requirements and also as the "context size" μ from the BSP/CGM simulation of Dehne et al. [1997].

RAM-awareness is fulfilled by an important class of the known coarse grained parallel algorithms, namely

Algorithms with constant number of supersteps. Clearly if $\lambda(n) = c$ for some c all overhead is bound by the number of supersteps. In the test set that we use below, two algorithms fall into that category : for *sorting* we use ideas of Gerbessiotis and Valiant [1994] and for *randomized permutations* we use Gustedt [2003].

If we are not able to do bind $\lambda(n)$ and we potentially access all memory during each superstep we have that $R(n) = \lambda(n) \cdot p \cdot M(n) = \lambda(n) \cdot S_{\mathcal{A}}(n)$. In general such an algorithm will not be competitive if $T_{\mathcal{A}}(n) \approx S_{\mathcal{A}}(n)$. But there is an important class of algorithms that is in fact captured by this definition :

Proposition 2.1 *All CGM-algorithms as defined by Dehne et al. [1996] are RAM-aware and have a running time and RAM-penalty of* $\lambda(n) \cdot M(n)$ *per processor.*

Proof: CGM makes the simplified assumption that each processor exchanges all its data during each superstep. Thereby for such an algorithm $R_{i,j}(n) = M(n)$ for all processors and supersteps. □

Since they are not necessarily resource optimal, not all CGM algorithms are PRO. But any CGM-algorithm that is also PRO immediately falls under Theorem 1.

Another important class of algorithms that extends the CGM-setting had already been identified by Chiang et al. [1995] as candidates for efficient IO algorithms. Namely the class of algorithms for which we are able to bound the memory access for each superstep individually :

Algorithms with a recursive data reduction. These are algorithms that use recursion on a problem of reduced size to achieve their goal. If the data size in the recursion is some ε-factor smaller, $\varepsilon < 1$, then usually a bound with a geometric series proves a constant bound on the overhead. From our test set the *list ranking* algorithm falls into that setting. We use the algorithm and implementation described in Guérin Lassous and Gustedt [2002].

2.2 The simulation and proof of Theorem 1

In Algorithm 1 we present a general framework for a simulation of any PRO-Algorithm $\mathcal{B}$ on a sequential machine with an external storage capacity, here used as a "disk" which is equipped with a file system. For each superstep of $\mathcal{B}$ it simulates one processor after the other. For processor j it first fetches all necessary data (RAM and received messages) from disk, performs computation and then writes out to memory again.

The proof of Theorem 1 will be a direct consequence of the following lemma.

Lemma 2.2 *The following invariants hold for Algorithm 1 :*

line *empty* : *The main memory of the simulation is empty.*

line *simulate* : *The main memory of the simulation contains all data that is necessary to perform superstep i on processor j. The size of this data is less than or equal to $4R_{i,j}(n)$.*

line *saved* : *All data to perform superstep i is saved on disk. The maximal size of this data is $O(p \cdot M(n))$.*

Proof: Statements *empty* and the first part of *simulate* are straight forward. For the second part of *simulate* we may assume that $\sum_{k=0}^{p-1} m_{k,j}$ and $\sum_{k=0}^{p-1} m'_{j,k}$ are both bound by $R_{i,j}(n)$ since all data that is received or send should be touched at least once during the simulation phase. We also may assume that we are able to describe $D_j^{(i)}$ in phase map within a size that is bound to $R_{i,j}(n)$.

Statement *saved* follows from the assumption that all messages that are received or send during a superstep must fit into memory of the PRO-processor, that is $\sum_{k=0}^{p-1} m_{k,j}$ and $\sum_{k=0}^{p-1} m'_{j,k}$ are bound to $M(n)$. □

Proof of Theorem 1: The statements concerning the computing time and latency follow directly from the definitions of PRO and from the fact that in each superstep i and for each processor j the simulation only adds a constant amount of computing time for each processor when it is communicating. The statement about the memory of the computing unit follows directly from *empty* and *simulate* of Lemma 2.2.

For the external storage the bound on the size follows directly from *saved*. The bandwidth requirement is the sum of all read and write operations to files. By *simulate* these are bound to

$$p \cdot 4 \sum_{j=0}^{p-1} R_{i,j}(n) \leq 4pT_i \tag{4}$$

Algorithm 1: External Memory Simulation of PRO-Algorithm $\mathcal{B}$

Input: For each processor j a disk file D_j holding the initial data.

Output: For each processor j disk file D_j holds the final data.

init **foreach** *processor* j **do**
  for $k = 0, \ldots, p-1$ **do** Initialize disk files $m_{j,k}$ and $m'_{j,k}$ (representing messages from j to k) and L_j to be empty;

saved **foreach** *superstep* i *of* $\mathcal{B}$ **do**

empty **foreach** *processor* j **do**

map Read the data $D_j^{(i)}$ as part of D_j that has to be mapped for j in step i from disk;

receive **foreach** *processor* k *that sent a message to* j *in step* $i-1$ **do**
    read $m_{k,j}$ from disk;
    re-initialize the file $m_{k,j}$ to empty;

simulate Perform all computation that j performs in step i;

send **foreach** *processor* k *that receives a message from* j *in step* i **do**
    Write $m'_{j,k}$ to disk;
    Append k to file L_j;

unmap **begin**
    if $D_j^{(i)}$ *has changed during computation* **then** write $D_j^{(i)}$ into D_j on disk;
    Determine the part $D_j^{(i+1)}$ of D_j that has to be mapped in the next superstep and store this information on disk;
  end
  Free the memory occupied by all $m_{j,k}$, $m'_{j,k}$ and $D_j^{(i)}$;

rename **foreach** *processor* j **do**
    Read file L_j;
    foreach *processor* k *in* L_j **do** rename file $m'_{j,k}$ to $m_{j,k}$;
    Truncate L_j to be empty;

for superstep i. □

2.3 Extensions

In view of this proof we may now even give a stronger version of Theorem 1 as a corollary :

Corollary 2.3 *Let the notations be the same as in Theorem 1 and let $R(n)$ be the RAM-penalty of $\mathcal{A}$. Then $\mathcal{A}$ can be simulated on an external memory machine such that the bandwidth requirements on the external storage medium are $O(R(n))$.*

This distinction between $T(n)$ for computation and $R(n)$ for IO is particularly interesting for the sorting problem. For this problem we have essentially that $T(n) \approx R(n) \log M(n)$. In conclusion this confirms the known fact that the sorting lower bound can be banned from the top-level of the memory hierarchy to lower levels which makes it occur much less restrictive in practical settings.

Another possibility of extension doesn't concern the theorem but the machine abstraction and simulation. First observe that the simulation itself could easily be done on a multiprocessor machine with $q \leq p$ processors. In fact, since the simulation only adds a constant factor to the resources that are needed we immediately conclude :

Corollary 2.4 *For every PRO-algorithm $\mathcal{B}$ Algorithm 1 again describes a PRO-algorithm with granularity function $Grain'(n) = \Omega(Grain(n))$ and RAM-penalty $R'(n) = O(R(n))$.*

With this observation it is in fact easy to see that the statement also holds for a setting with multiple disks.

Corollary 2.5 *For $k \leq p = Grain(n)$, the statements of Theorem 1 and Corollary 2.4 hold for environments with k different disks of equal size.*

3 The SSCRAP library as an starting point

The main purpose of SSCRAP, see Essaïdi et al. [2002], is to provide a development framework for the various coarse grain models and to guarantee the portability and efficiency of the produced programs on a large variety of parallel and distributed platforms. It is not the first library that provides such a framework : several libraries implementing communications for coarse grained models were developed, see Miller [1993], Goudrau et al. [1995], Hill et al. [1997], Bonorden et al. [1999]. One of its particularities is the greater range of models for which it implements algorithms, see Essaïdi [2004], among them PRO.

PRO algorithms with some easy-to-follow "*golden rules*" and their implementation as SSCRAP programs ensure the efficient use of nowadays parallel and distributed hardware. A simulation of a multi-processor algorithm on a machine with a small number of real processors (or just one) behaves almost the same in terms of work load of the machine as a whole.

For the SSCRAP design, the main goals were portability, efficiency and scalability. The most difficult task was to balance between efficiency and portability. Indeed, to be portable, a program must consider a generic machine model disregarding the specificity of the physical architecture. However to be efficient, a program must get the full benefit from the available resources which differ enormously from one architecture to another. SSCRAP ensures this difficult task by introducing an

abstract communication layer between the "user" routines of SSCRAP and the target platform. Currently, SSCRAP is interfaced with the two main types of parallel architectures : distributed memory (as e.g. cluster of PC or workstations) and shared memory.

For the context of this study only the shared memory interface is of relevance. That interface is build upon *PosiX threads* which ensures portability and efficiency on most modern platforms. In fact, usually SSCRAP can run with some hundred SSCRAP processors (ie threads in that case) without significant performance degeneration on any of the platforms that are supported : ranging from mono-processor PC or workstations, over few-processor machines like quadri-processor PCs to multiprocessor mainframes.

3.1 Serializing execution

The first thing to add to the shared memory interface was a tool to allow for the sequential "in-order" execution of the SSCRAP processors. Since each processor to be simulated corresponds to a thread, this was relatively simple to achieve with a semaphore. Whenever a SSCRAP processor enters one of the functions that define the end of a superstep it `post`s (increases) that semaphore, whenever it starts a new superstep it `wait`s for the same semaphore. At program startup the semaphore is initialized by the value κ found in environment variable `SSCRAP_SERIALIZE` (or infinity if doesn't exist).

As a consequence, at most κ SSCRAP processors will execute concurrently inside a superstep at any time. Care is taken such that this statement holds for any of the time consuming operations, especially interprocessor copying of data. We can simply think of the SSCRAP processors passing κ execution tokens to each other. Each processor keeps its token as long as it can during a superstep. It only passes it to the next processor when it is obliged to wait that its communications are terminated. Then it waits for its next turn for a token and enters the next superstep.

On the other hand, in the very brief period in between the supersteps when SSCRAP handles its control datastructures all SSCRAP processors are active and no deadlock occurs.

3.2 Communication Abstraction

Probably the most important feature of SSCRAP in the present context is the abstraction that it enforces from the underlying communication. In each superstep, the program may perform any sort of local computation for a given SSCRAP processor. In addition each such processor may issue one `send` and one `receive` operation to and from any processor. So in each superstep each processor issues at most p `sends` and `receives`, for p the amount of SSCRAP processors.

These two types of operations are non-blocking and unbuffered such that the processor can continue its operation until the end of the superstep. On the other hand it may not rely on the contents of the underlying send or receive buffers during the same superstep. In that way, the specific communication library can handle communications efficiently according to the specific requirements of the setting, without making unnecessary copies of the data in buffers or blocking the processor while it might have useful work to perform.

In an external memory context this means that write or read operations (here `send` and `receive`) are scheduled long before the data will effectively be used by the program in the next superstep.

```
void identity( sscrap::step::array< long > &A)
{
  sscrap::mem::apointer< long > ap(A);
  for ( long i=0;
        i<A.localSize();
        ++i)
      {
        ap[i]=i;
      }
}
```

TAB. 1 – Programming example

3.3 Data Abstraction

The main add-on that was integrated into SSCRAP was a consequent implementation of an abstraction between the *data* of a process execution and the memory of a processor. This is to ensure the transparent implementation of the steps *map* and *unmap* of Algorithm 1.

The programmer acts upon these on two different levels :

`sscrap : :step : :array` gives a kind of *handle* on some data array which is common to all SSCRAP processors.

`sscrap : :mem : :apointer` maps its (local) part of that data into the address space of the SSCRAP processor and lets the programmer access this part like she would with a conventional pointer.

Such a two level model is in contrast to the usual "C" approach where arrays and pointers are (almost) the same type of object.

It has the big advantage to free the run-time library from certain obligations. E.g two different `apointers`, `ap` and `bp`, that are constructed from the same `array` object at different points of the program may see the same data *mapped* to completely different addresses in the address space of the processor. In that way the system may re-use addresses and physical memory according to the needs of the execution.

In the implementation the mapping and unmapping into the address space is done with the system calls `mmap` and `munmap`, see IEEE 1003.1.

3.4 Design

To guarantee portability and efficiency SSCRAP is written in C++. This allows us to have a design organized in different levels of abstraction, from the interface to the communication library (threads or MPI) over classes organizing the supersteps to user interfaces such as `sscrap : :step : :array` as described above. On the other hand, care is taken that modern C++ compilers are able to transform this code into efficient assembler.

For the use of external memory a demand for a pointer (address) to the local part of an array can be translated into a file mapping request by means of the system call `mmap`. This is done without changing the code or even recompiling it. Upon *creation* of a `sscrap : :step : :array` the environment variable `SSCRAP_MAP_FILE` is inspected. If it is set, the memory for the array is allocated in a disk file and otherwise on the heap (with `malloc`). Since such events of creation are quite rare, the fact of deciding upon the memory model at runtime is not noticeable.

The C++ design of these classes hopefully guarantees that these constructs are easy to use. Consider the function definition in Table 1 as an example. It ensures that first the local part of parameter `A` is mapped into memory if necessary, that it can be used efficiently inside the `for`-loop, and that it is unmapped at the end. Then, the local part of the table can be accessed as usual inside the `for`-loop, here with the assignment `ap[i]=i`.

In general, compilers are able to optimize such a code as given here such that all overhead (calls to `mmap`, `munmap` etc) is done outside the loop and such that all information (here `i`, a pointer and the size of the `array`) are hold in registers.

4 Experiments and analysis of the result

The implementation as presented here runs successfully on different types of platforms. We tested it on PC (mono- and biprocessor), multiprocessor workstations (SUN) and a mainframe with 56 processors (SGI). But since the tests are quite time consuming (one run several hours) and quite sensible to perturbations we had to restrict to machines that could be dedicated during a whole week to these tests.

The experiments as presented were taken out on a modern PC with the following characteristics :

CPU	Pentium 4	2×2.0 GHz
RAM		1 GB
bus	speed	99 MHz
disk	swap	2 GB
	available file system	20 GB (software raid)
	bandwidth read/write	55/70 MB/sec
OS	GNU/linux	v. 2.4.18
C++	gcc	v. 3.2

Reading the whole contents of the RAM on from disk needs at least 18.6 seconds and a total exchange of RAM with disk 33.2 seconds. A streaming IO operation has a cost of

data type	memory	disk read	disk write
`long`	6.25 cc	149 cc ($22\times$)	117 cc ($18\times$)
`double`	12.5 cc	298 cc ($22\times$)	234 cc ($18\times$)

Since the machine had no other charge, variances between different runs of the same problem instances were low. Therefore, each experiment was only performed 7 to 10 times.

The results presented here are simply the average of the wall clock time that the experiments did take. They were scaled by the number of items in the experiment to obtain a cost factor of seconds per item. Curves are presented with doubly-logarithmic scales to better cover the different orders of magnitude of the experiments.

All parallel algorithms have the common property that they use much more memory than the sequential ones. Generally spoken they are not *in place* algorithms like the sequential ones, so they use at least twice as much memory. The maximal amount of items that could be placed in RAM is indicated by a vertical bar in each of the figures. In general, it was not possible to drive the in-core execution of the programs beyond that barrier.

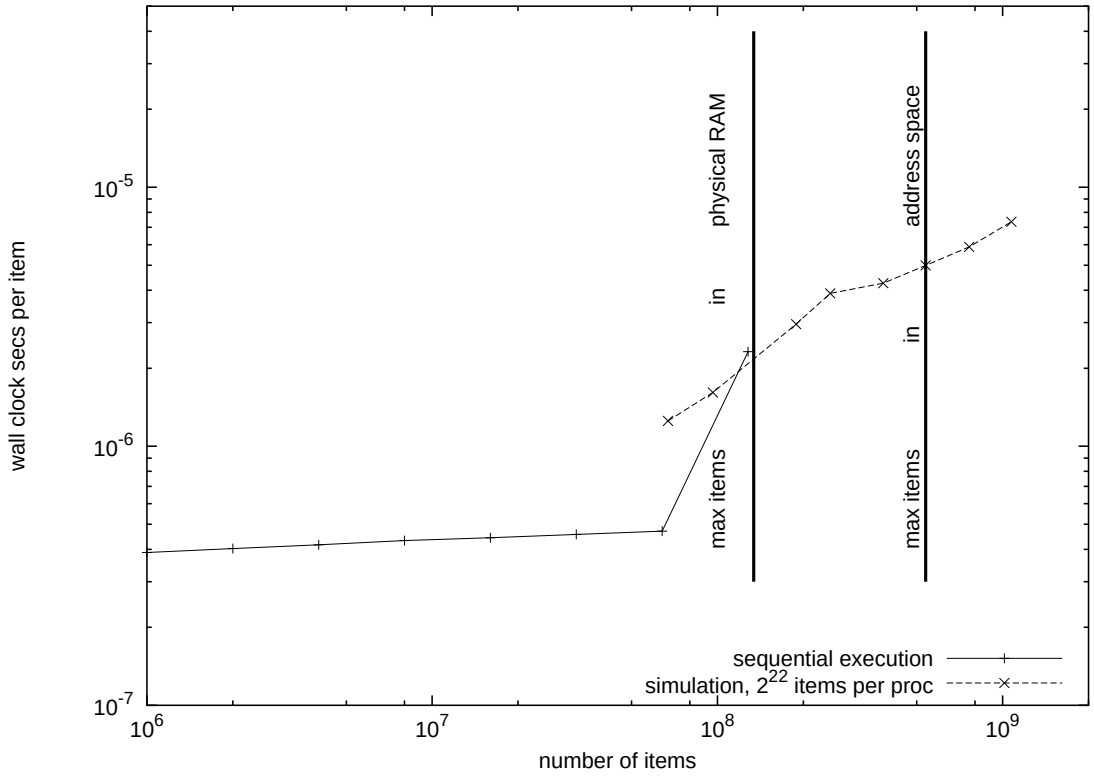


FIG. 1 – Sorting

4.1 Sorting

Besides being very useful in practice, sorting is a very interesting problem from a conceptional point of view. It has a non-linear lower bound in terms of computation, but this bound doesn't apply to memory access or communication operations.

In fact the implementation of the algorithm of Gerbessiotis and Valiant [1994] that we were using for the experiments takes advantage of this subtle difference. It has an overall computation time that is dominated by the time of two different phases

- performing a binary search operation to compute the processor number to which the item will be send, and
- locally sorting the received data on each processor.

Thus in general this time is $O(N/p(\log p + \log(N/p)))$, N the number of items, p the number of processors.

On the other hand it allows for only linear streaming IO. In terms of Theorem 1 and Corollary 2.3 it is a PRO algorithm that achieves $R(n) = O(n)$. So in fact the most expensive operation, memory access, can be avoided and substantially reduced.

We performed sorting of a vector of `doubles`. The sequential sorting routine that is uses as a subroutine is an efficient in place *quicksort*. The maximal data size the machine could physically hold in memory were 134 mio items. The maximum number of items that can hold in the address space of the 32-bit architecture is 537 mio items. Figure 1 shows the results of the experiments.

The file mapping takes much more time than running the program entirely in RAM. Because of the average cost of some hundred clock cycles for reading one item from disk this is not very surprising. On the other hand the corresponding running times were reliable even far beyond the swap boundary.

We also see, that the factor in bandwidth of about 20 between RAM and disk access is maintained : per item, the out-of-core computation is not slower than 20 times the in-core computation.

4.2 Randomized Permutations

Randomized generation of permutations is a problem with a linear time complexity but where the most costly operation is random memory access. At least the standard sequential algorithm has a highly random memory access pattern and so in general most of these memory accesses will be cache or page misses.

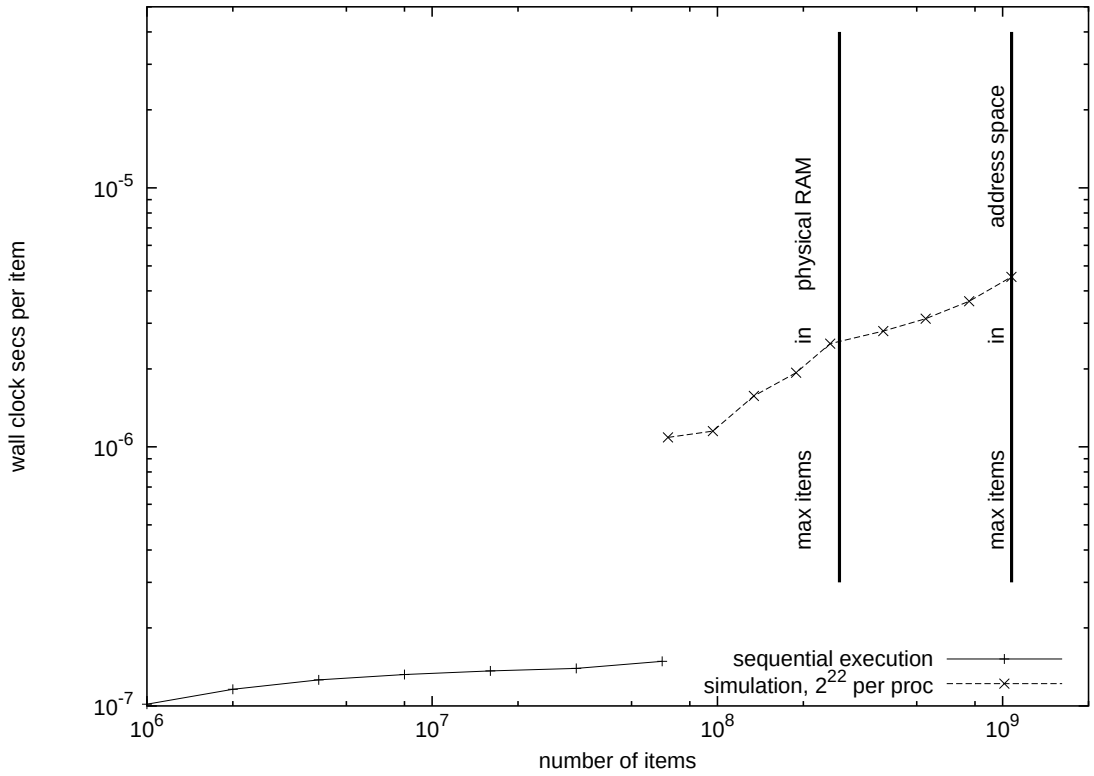


FIG. 2 – Randomized Generation of Permutations

But the needs of pure computation time for such a generation are also quite high, since we need (pseudo) random numbers. So memory access and computation should equilibrate to a certain extent.

For the tests we generated randomized permutations of `long int`s. The random number generation was done with linear congruent generators, one for each SSCRAP processor. The maximal data size the machine could hold in memory is 268 mio items.

Again, the mapping memory to files is a good strategy to handle problem sizes that otherwise could not be treated by the machine, and this by only being as slow as the bandwidth restriction of the external device enforces.

4.3 List Ranking

Given a collection of lists, the list ranking problem is to compute the distance to the tail of its component for every single item in these lists. This problem has a simple linear time solution, but

since we may not assume any particular organization of the list items in memory, it has a highly random memory access pattern.

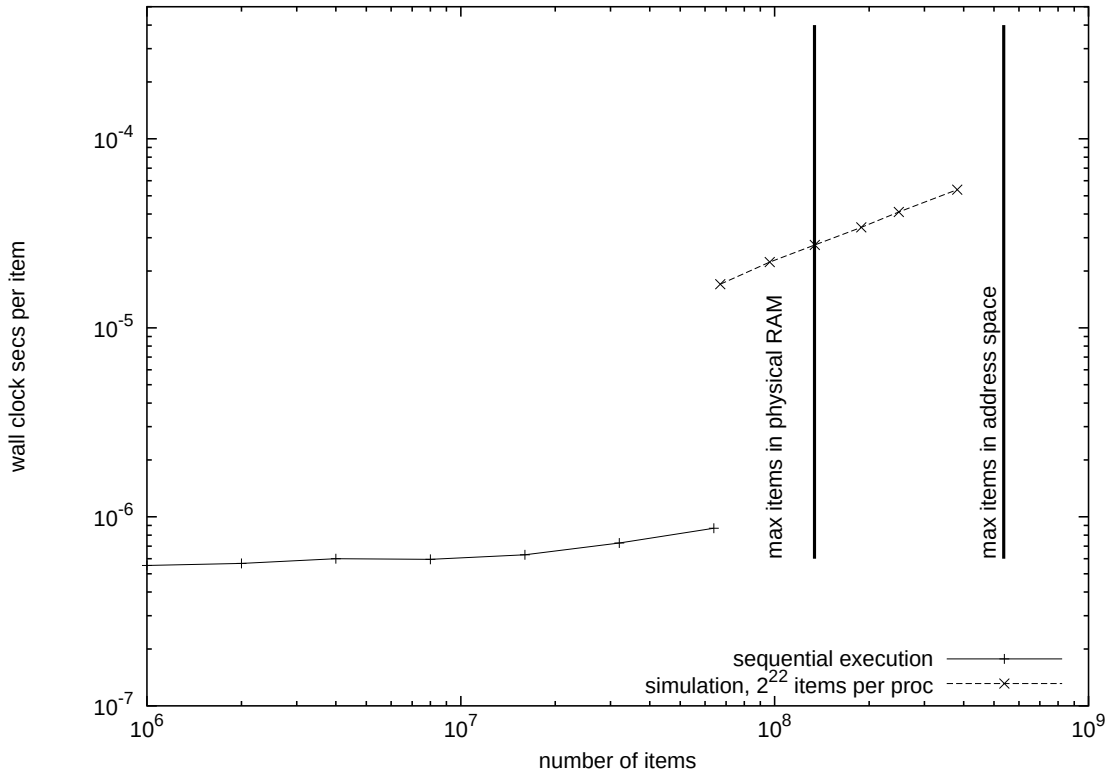


FIG. 3 – List Ranking

Even worse, since the principal operation here is the memory access itself, there are not much other operations that can be performed during a memory request. So the memory access pattern completely dominates execution time.

Since we need two `long` values to handle each item (one for the next in the list, and one for the distance), the maximal data size the machine could hold in memory is of 67 mio items.

List Ranking was the most demanding among the three problems. It has already a sequential running time which is quite hight and the implementation of an efficient parallel algorithm has not been simple at all. But the qualitative behavior of the results are the same as for the two other problems. We are able to handle large problems with a slow down that stays in the bounds of the reduced bandwidth of the disk.

5 Conclusion and Outlook

We have seen that coarse grained parallel models like PRO and their realizations in terms of the library SSCRAP enable us to extent the use of certain types of parallel programs to the setting of out-of-core computation. The principal bound in problem size that we encountered was related to the availability of a resource that is easily extensible and cheap, disk space, and the main bottleneck for the computation time as a whole is the bandwidth of this external storage device.

The principal restriction for an existing parallel algorithm is the amount of memory it needs to be mapped during each individual superstep of the computation. There is a large family of algorithms for which simple bounds on this resource can be estimated and that lead to efficient implementations in this setting.

We are able to handle problems that are larger than the address space (4 GB) of one individual (i86) processor. This can be extended to a cluster of k PC each having 4 GB of free disk could hold and sort a table of $\approx \frac{k}{2} \cdot 10^9$ `doubles`.

This would be particularly simple to undertake for sorting since it only accesses the data items in a table. For the other algorithms the address of an individual item has to be encoded. If we continue using `unsigned longs` for that, the problems size is bounded by $2^{32} \approx 4.2 \cdot 10^9$ items. Moving to `long longs` which most compilers (and architectures) implement nowadays this bound can be extended to $1.8 \cdot 10^{19}$.

Another direction to follow will be an optimized use of the existing code. The choice of the interrelation of the number of processors p that is simulated and the input size N could be important and should thus be investigated further.

Références

Olaf Bonorden et al. The Paderborn University BSP (PUB) Library—Design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, 1999.

Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149. Society for Industrial and Applied Mathematics, 1995. ISBN 0-89871-349-8.

Thomas H. Cormen and Michael T. Goodrich. A bridging model for parallel computation, communication, and I/O. *ACM Computing Surveys*, 28A(4), 1996.

F. Dehne, W. Dittrich, and D. Hutchinson. Effi cient external memory algorithms by simulating coarsegrained parallel algorithms. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3) :379–400, 1996.

Mohamed Essaïdi. *Échange de données pour le parallélisme à gros grain*. PhD thesis, Université Henri Poincaré, Nancy, France, February 2004.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP : An environment for coarse grained algorithms. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 398–403, 2002.

Assefaw Hadish Gebremedhin, Isabelle Guérin Lassous, Jens Gustedt, and Jan Arne Telle. PRO : a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 106–113. IEEE, The Institute of Electrical and Electronics Engineers, 2002.

Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2) :251–267, 1994.

M. W. Goudrau, K. Lang, S. B. Rao, and T. Tsantilas. The green BSP library. Technical Report TR-95-11, University of Central Florida, Orlando, 1995.

Isabelle Guérin Lassous and Jens Gustedt. Portable list ranking: an experimental study. *ACM Journal of Experimental Algorithmics*, 7(7), 2002. URL `http://www.jea.acm.org/2002/GuerinRanking/`.

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

Jonathan M. D. Hill et al. BSPlib : The BSP programming library. Technical report, Oxford University Computing Laboratory, 1997. URL `http://www.bsp-worldwide.org/standard/bsplib_C_examples.ps.Z`.

IEEE 1003.1. *The Open Group Base Specifi cations Issue 6 : IEEE Std 1003.1*. IEEE and Open Group, 2003. URL `http://www.opengroup.org/onlinepubs/007904975/mindex.html`.

Richard Miller. A library for bulk-synchronous parallel programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, 1993. URL `http://www.comlab.ox.ac.uk/oucl/oxpara/ppsg.ps.gz`.

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, 1990.

Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I : Two-level memories. *Algorithmica*, 12(2-3) :110–147, 1994. URL `citeseer.nj.nec.com/vitter92algorithms.html`.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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