Parallel out-of-core sorting and fast accesses to disks

Christophe Cérin*
LIPN – UMR CNRS 7030,
Institut Galilée, Université Paris-Nord,
99, Avenue Jean-Baptiste Clément,
93430 Villetaneuse, France
Fax: +33-(0)148 26 07 12 E-mail: christophe.cerin@lipn.univ-paris13.fr
*Corresponding author

Olivier Cozette and Gil Utard
Université de Picardie Jules Verne LaRIA,
Bat Curi, 5 rue du moulin neuf, F-80000 Amiens, France
E-mail: cozette@laria.u-picardie.fr E-mail: utard@laria.u-picardie.fr

Hazem Fkaier and Mohamed Jemni
Ecole Superieure des Sciences et Techniques de Tunis,
Département d'informatique, 5AV Taha Hussein Montfluery,
1008 Tunis, Tunisie
E-mail: hazem.fkaier@fst.rnu.tn E-mail: mohamed.j emni@fst.rnu.tn

Abstract: The paper addresses two problems. We investigate the problem of parallel external sorting in the context of a form of heterogeneous clusters then we investigate the impact of efficient disk remote accesses on the performance of external sorting. We explore three techniques to show how they can be deployed for clusters with proportional processor performances. We also validate the READ^2 library, an efficient implementation of remote SCSI disk accesses. We derive a new parallel sorting algorithm that is adapted to the READ^2 interface. The expected gain of using READ^2 is compared to the measured gain for one external sorting implementation.

Keywords: out of core, parallel sorting algorithms; performance evaluation and modelling of parallel integer sorting algorithms; sorting by regular sampling and by over partitioning; data distribution; load balancing strategies; I/O bandwidth; disk bandwidth; remote I/O; system area network.

Reference to this paper should be made as follows: Cérin, C., Cozette, O., Utard, G., Fkaier, H. and Jemni, M. (2005) 'Parallel, out of core, sorting and fast accesses to disks', *Int. J. High Performance Computing and Networking*, Vol. 3, Nos. 2/3, pp.188–202.

Biographical notes: Christophe Cérin is a Professor at the University of Paris 13, Laboratoire d'informatique de Paris Nord (LIPN) in 2005. He received his PhD in University of Paris XI, Laboratoire de recherche en informatique d'Orsay (LRI) in 1992. His research interests are in the area of Parallel and Distributed Processing (Grid, Cluster), and Parallel IO. His research in Parallel and Distributed Processing focuses on fundamental techniques for partitioning data in heterogeneous environments (both in terms of CPU and network bandwidth). He is also concerning with the programmability of parallel machines, for instance under the framework of GridExplorer project (http://www.lri.fr/~fci/GdX). Finally, he is also contributing to the research on Parallel IO for clusters and its usage in industrial projects. His contact website is http://www-lipn.univ-paris13.fr/~cerin/.

Olivier Cozette received his Master degree in Mathematics and PhD in Computer Sciences with Laboratoire de Recherche en Informatique d'Amiens (LaRIA -Amiens). His main interests include memory management on Linux (swapping policies) and remote disk access (with Myrinet and LSI SCSI adapter). Currently he is a software enginneer on storage technologies at Seanodes http://www.seanodes.com (Toulouse/France).

Copyright © 2005 Inderscience Enterprises Ltd.

Gil Utard has been a Associate Professor at the University of Picardie Jules Verne since 1997 and received his PhD from the University of Lyon in 1995. He is involved in many projects in parallel IO (file system and software library interfaces), data-Grid and Peer-to-peer systems. He is currently involved with the USA (Ubiquitus Storage – http://www.iut-amiens.fr/~olivier.soyez/ustorage/) which is devoted to a secure service for storing data over the Internet.

Hazem Fkaier has been a PhD Student at Faculté des Sciences de Tunis in 2003 and Engineer in Computer Science from Faculté des Sciences de Tunis in 2000, Master in Computer Science from Faculté des Sciences de Tunis in 2003. He is a member of he Research Unit of Technologies of Information and Communication – University of Tunis. His Research Projects Involvement: High performance Computing; Algorithmic and tools for Grid and Cluster computing.

Mohamed Jemni is a Professor at Ecole supérieure des Sciences et Techniques de Tunis, University of Tunis in Tunisia. His habilitation to supervise research in Computer Science, University of Versailles, France in 2004. He received his PhD in Computer Science, University of Tunis in 1997 and Engineer Diploma in Computer Science, Faculty of Sciences of Tunis in 1991. He has been a Head of the Research Unit of Technologies of Information and Communication – University of Tunis and Head of Research team on e-learning tools and environments. His Research Projects Involvement: High performance Computing; Algorithmic and tools for Grid computing; Advanced e-learning environments.

1 INTRODUCTION

It is often said that 25–50% of all the work performed by computers is achieved by sorting algorithms (Akl, 1985). One reason, among others, for the popularity of sorting is that sorted data are easier to manipulate than unordered data. For instance a sequential search is much less costly when the data are sorted. The quasi non predictable aspects of memory references in sorting algorithms make them good candidates to appreciate the performance of processors in real situations.

The advent of parallel processing, particularly in the context of *cluster computing* is of high interest with the available technology. A special class of *non homogeneous clusters* is under consideration in the paper. We mean clusters whose global performance is correlated by a multiplicative factor.

This class of machines is of particular interest for two kinds of customers:

- for those who cannot instantaneously replace the whole components of their cluster with a new processor or disk generation, but will compose with old and new processors or disks
- second for people sharing cpu-time, because the cluster is not a dedicated one.

We focus here, on the ways that ensure good load balancing properties: if a processor is initially loaded with n integers and n is related to its performance, then the processor must never deal with more than $k.n$ integers with the requirement that k should be as low as possible.

Since our framework is related to external sorting, we also explore the ways of performing efficient I/O operations in a context larger than our sorting problem. Grand challenge applications often process large datasets that require high performance I/O systems. For example, the amount of data processed at the European particle accelerator (LHC/CERN), reaches several Petabytes/year (Hoschek et al., 2000) and the San Francisco museum uses 20 PCs with 368 disks to manage a 3.2 Terabytes digitised picture collection (Talagala et al., 2000). To deal with such datasets in a 'cluster architecture', disk drives are aggregated in order to provide a large parallel file system.

We demonstrate that the overall performance of our external sorting algorithms is increased significantly when using an efficient library, capable of transferring disk to disk data. The design of READ^2 library is introduced and we explain how to reduce memory transfers in the context of a cluster built of SCSI disks and Myrinet cards.

The remainder of the paper is organised as follows. In Section 2, we introduce the problem of incore and out of core sorting and the notion of heterogeneity. In Section 3, we relate known solutions for parallel sorting on homogeneous clusters and we introduce basic concepts for sorting on heterogeneous platforms. Section 4 is devoted to sorting on heterogeneous clusters. In Section 5 we introduce our library for implementing efficient disk to disk transfers. Section 6 is related to experiments and Section 7 concludes the paper.

2 RELATED WORK AND DISCUSSION ABOUT THE NOTION OF HETEROGENEITY

'Out of core' algorithms process data stored in external memories like disks, meanwhile data processed by incore algorithms is stored in main memory. Parallel sorting algorithms within the framework of 'out of core' computation is not new. The most valuable and recent publications, to our knowledge are Schikuta and Kirkovits (1996), Rajasekaran (1998), Cormen and Hirschl (1997), Pearson (1999) and Nodine and Vitter (1995). Since our work is based on sampling techniques, we shall mention DeWitt et al. (1991) that summarises all the work in the field prior to 1991.

The objective in these papers is to minimise the round disk trip or the number of disks we accesses as these are very costly, with current disk technology when compared with the memory to memory or cache access time. Salem and Garcia-Molina (1986), Knuth (1998) and Kim (1986) are examples of techniques that efficiently deal with disks of and presentation of sequential external sorting algorithms. From a model point of view, the papers of Vitter and Shriver (1994a, 1994b) and Aggarwal and Vitter (1988) offer, in our opinion, the best views of parallel disks systems. We introduce below Vitter's work in order to clarify ideas about the challenge of external algorithms.

2.1 The I/O model

The I/O model measures the complexity of an algorithm not by the number of processing instructions but by the number of required I/O operations required. Vitter captures the main properties of disk systems by the commonly used *parallel disk model* (PDM) through the following parameters:

- N: Problem size (in units of data items)
- M: Internal memory size (in units of data items)
- B: block transfer size (in units of data items)
- D: Number of independent disk drives
- P: Number of CPUs

where $M < N$, and $1 \leq DB \leq M/2$ for practical reasons and to match existing systems. Figure 1 depicts PDM. In a single I/O, each disk can simultaneously transfer a block of B contiguous data items. It is convenient to refer to the following shortcuts:

$$n = \frac{N}{B}, \quad m = \frac{M}{B}.$$

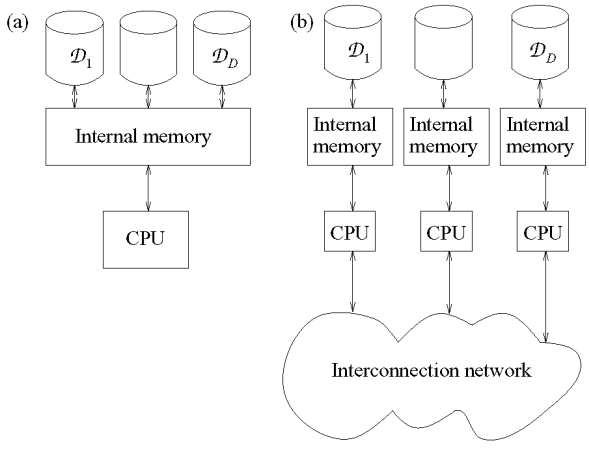


Figure 1 *Parallel disk model (Vitter and Shriver, 1994a) (a) P = 1, in which the D disks are connected to a common CPU and (b) P = D, where each of the D disks is connected to a separate processor. This last organisation is realistic for a cluster system*

Ideally, algorithms should use linear storage space i.e., $O(N/B) = O(n)$ disk blocks of storage. It can be proved that the I/O bound on sorting N data items with $D \geq 1$ disk is given by:

$$Sort(N) = \Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\log_m n\right).$$

Note that, in practice, the $\log_m n$ term is a small constant. The main theorem for sorting within the framework of PDM is the following:

Theorem 1 (Aggarwal and Vitter, 1988; Nodine and Vitter, 1995)**:** *The average and worst case number of I/Os required for sorting N = nB data items using D disks is:*

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right). \tag{1}$$

To reach the bound of equation (1), techniques named *distribution* or *merge* based should be devised. These techniques access the D disks independently during parallel read operations, but in a striped manner during the parallel write operations. Let us examine *distribution sort* (Knuth, 1998) which is very close in spirit to the sampling algorithms we will discuss in a forthcoming section.

Distribution Sort is a recursive algorithm where the inputs are partitioned by a set of $S - 1$ splitters into S buckets. The individual buckets are recursively sorted. There are $\log_S(n) = \log_m n$ recursion levels and the bucket sizes decrease by a factor of $\Theta(S)$ from one level of recursion to the next. If each level of recursion uses $\Theta(N/DB) = \Theta(n/D)$ I/Os, then distribution sort performs with I/O complexity of $\mathcal{O}(n/D)\log_m n)$ which is optimal.

When researchers study performances, the parallel execution time is not the only metric used. Blelloch et al. (1991) defines the concept *of sublist expansion metric* as the ratio of the size of the largest list treated by a processor in one moment of the algorithm on the expected average size. In other words, this metric accounts for load balancing; ideally, a value one is suspected. In this case, load balancing is optimal.

2.2 What is a heterogeneous cluster?

The previous discussion was made in the context of sorting on a sequential machine or on homogeneous clusters, i.e., clusters based on identical motherboards, identical disks, identical CPUs.

We are now interested in a particular class of *non homogeneous clusters* that we define here as usual clusters (network based on unique communication layer, same installed operating system, same CPU, same memory size and disks) but microprocessors may have *different speeds*. It is a first level of heterogeneity and it introduces new challenges.

In a parallel system the notion of heterogeneity potentially covers all the components of the system:

- *Network*: heterogeneity concerns here, the possibilities of using different media of communication, various methods of signalling, various interfaces and protocols
- *Memory*: it concerns the possibilities of using local memory as well as distant memory, of managing various levels of hierarchy with various management policies
- *Processors*: it concerns the possibilities of using processors of various manufacturers, with various speeds, with different internal architectures i.e., RISC (Reduced Instruction Set Computer), VLIW (Very Long Instruction Width architecture), multithreadead architecture
- *Software running on the systems*: it concerns the possibilities of using several operating systems or different binary codes of the same program
- *Disks*: it concerns the possibilities of using several file systems, different storage media (hard disks, floppy, cartridge), various protocols (IDE ATA, SCSI, PCMCIA).

A parallel heterogeneous environment, as we have just defined it, establishes interesting and particularly useful problems if the user of a cluster cannot change all the processors of the cluster but has to compose with several versions of one processor, with various speeds. It also seems to us that there is not a lot of literature (to our knowledge) on this class of architecture; many papers on sorting, treat the homogeneous case only.

For the problem of effectively measuring the relative processor speeds, we assume the availability of precise techniques to realise it.

3 RELATED WORKS ON PARALLEL SORTING ON HOMOGENEOUS CLUSTERS

One algorithm using parallel sampling techniques and for the *D* disk model is the work of DeWitt et al. (1991) which is a randomised two steps distribution sort algorithm.

- They define *N* buckets for an N-process program. Then, each program reads its initial data segment and sends each element to the appropriate bucket (other process). All elements received are written on disks as small sorted runs.
- Each process merge-sorts its run.[1]

The other type of strategy for external sorting is 'sorting by merging' which is orthogonal to the previous paradigm. The principle can by depicted as follows:

- In the 'run's formation' the n blocks of data are streamed into memory, one memory load at a time; each memory load is sorted into a 'single run', which is then output to the disk(s). There are n/m sorted runs.
- *Merging phase*: the groups of R runs are merged together. During each merge, one block from each run resides in RAM. Some recent and later refinements 'in parallel' of this strategy are presented by Rajasekaran (1998) and implemented by Pearson (1997).

Sorting is also present in several test sets (see the NASA website, http://www.nas.nasa.gov/). The NOW (http://now.cs.berkeley.edu/NowSort/index.html/) at Berkeley is, without any doubt, the first project (1997) concerning sorting on clusters. The project considers a homogeneous cluster of SPARC processors and authors measure time performances only and not the quality of load balancing.

We also find people from industry who are specialists in sorting. For example, we may mention here, the Ordinal company (http://www.ordinal.com/). They distribute `Nsort` for various platforms, generally multiprocessor machines.

The goal is to sort as much data as possible in one minute (minute sort). However, the sorting algorithm has little interest from a scientific point of view: it is a question of reading the disks, of sorting in memory and writing the results on disks. It is a brute force, but efficient! It will be noted that the total size of the main memory is always higher than the size of the problem!

In March 2004, the Nsort program was able to sort 34 gigabytes of data (340,000,000 100-byte records) in 58 seconds on a 32 processor Itanium 2 NEC Express5800/1320Xd running Microsoft Windows Server 2003 Datacenter Edition. This set new records for the MinuteSort (http://research.microsoft.com/barc/SortBenchmark/) benchmark.

3.1 General principles

In this section we recall some well known strategies for incore sorting in parallel. We focus on a *specific technique for the homogeneous case.* It is now well understood that two generic approaches for incore sorting, in parallel, are of particular interest and work in practice (implemented algorithms are efficient on a variety of multiprocessor architectures).

Merge based: for this kind of algorithm, the different steps may be summarised as follows:

- each processor involves a portion of the list to be sorted
- each processor sorts the portions and exchanges them among all the processors
- each processor merges portions in one or several steps.

Quicksort based: for this kind of algorithm, the different steps may be summarised as follows:

- the unsorted list is partitioned into a number of progressively smaller sublists defined by selected pivots
- sort the sublists for which processors are responsible.

Only a merge based algorithm is under consideration in this paper, principally because the bound on load balancing for heterogeneous clusters is easier to obtain and experimental

results are good. We specifically focus on *one step communication* algorithms because they match the requirement of using a limited number of short messages in message passing, programming languages in order to obtain good performance.

We guess that our programs should perform well on clusters with a typical network such as fast ethernet. Thus, we need a limited number of communication steps in order to avoid slowdown by the network bandwith.

To summarise, *one step merge based algorithms* have low communication cost because they move each element at most once (and at the 'right place') and they ensure regular communication requirements invariant with respect to the input distribution as we will see later in the paper. Their main drawback is that they have poor load balancing if we do not care about it; it is difficult to derive a bound to partition data into equal size sublists.

Let us denote by *n,* the size of the problem and by *p,* the number of processors. The four canonical stages of the sorting by regular sampling algorithm are the following. It is the algorithm of Shi and Schaeffer (1992) called PSRS (Parallel Sorting by Regular Sampling) which was designed for the homogeneous case:

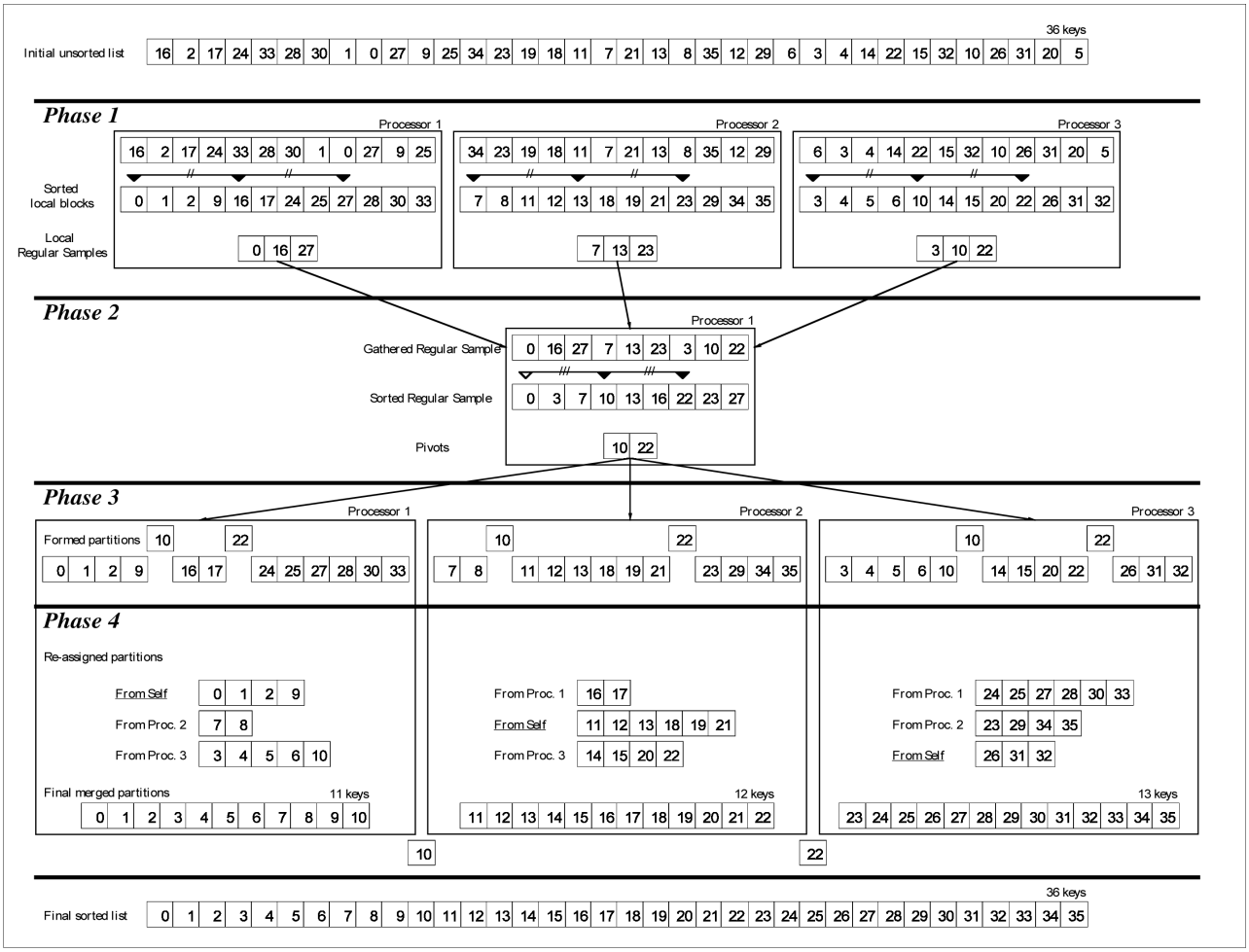
Step 1: One starts by sorting its n/p data locally, then each processor selects p pivots which are gathered on processor 0 (one passes the details concerning the choice of pivots itself).

Step 2: Sort on processor 0 of p^2 pivots; we keep $p-1$ pivots (they are selected at $ip + p/2$, $(1 \le i \le (p-1)$ intervals); all the pivots are broadcasted to the other processors.

Step 3: Each processor produces p sorted partitions according to the $p-1$ pivots and sends the partition i (marked by pivots k_i and k_{i+1}) to processor i.

Step 4: The processors received sorted partitions; merge them.

Figure 2 presents an example of unfolding such an algorithm. Notice the initial local sorting on each processor (which can be implemented with an incore or 'out of core' sequential algorithm), the choice of the pivots which is done at regular intervals, the centralisation of the pivots and their sorting and finally the redistribution of the data according to values of the pivots.



Source: Shi and Schaeffer (1992)

Figure 2 *An example of PSRS execution*

The key of success depends on the splitters that must partition the bucket into roughly equal sizes. As noted in Vitter and Shriver (1994a) for external sorting,

> "It seems difficult to find $S = \Theta(m)$ splitters using $\Theta(n/D)$ I/Os (the formation of the buckets must be done akin to this bound to guarantee an optimal algorithm) and guarantee that the bucket sizes are within a constant factor of one another."

However, what makes the force of this algorithm is that, by sampling on all the processors, we can consider that information on data order is captured. It was shown by Shi and Schaeffer (1992) that the computational cost of PSRS matches the optimal bound of $\mathcal{O}(n/p \log n)$. To obtain this result, it is necessary to be ensured of the unity of the data. In this case, one can show that, under the assumption that $n > p^3$, PSRS guarantees a load balancing which differs from the optimum by a ratio of two, which is very satisfactory. Indeed, that means that, at stage four of the algorithm presented above, none of the processors sorts more than twice the number of items it had at the beginning. In practice, the constant is rather close to the optimal one.

3.2 Related work on heterogeneous clusters

The strategy of sampling is studied in this section for sorting on a heterogeneous cluster.

Intuitively it is a question of insulating in the input vector, the pivots which would partition it into segments of equal size. This phase selects pivots so that between two consecutive pivots, there is the same number of objects. After a redistribution phase of the data according to values of the pivots, it no longer remains to sort the values locally.

A first attempt for this strategy was successfully introduced and implemented in Cérin (2002) and Cérin et al. (2003). Only the principle of choosing pivots has been adapted, the others steps of PSRS framework, namely initial sort, distribution and merging are unmodified.

When different numbers of array elements are assigned to processors, one may think that it will create a communication imbalance during the distribution of the array elements. This may result in endpoint contentions (i.e., multiple processors sending data to the same processor concurrently). This is true only if we consider that we have a point to point network (Myrinet) and not an ethernet network which is a shared bus with contention arbitration done in hardware and not under the programmer's responsibility.

In addition, if the processor I/O subsystem is not identical from one node to another, data distribution based on processor speeds may cause some processors to slowdown the total execution.

As noted previously, we do not study here, the impact of the network and I/O subsystems performance at each node. Elaborating on the impact of such issues on performance is a challenging task.

3.2.1 A general framework for sorting on heterogeneous clusters

The problem is introduced as follows: n data (without duplicates) are physically distributed on p processors. Here the processors are characterised by their speeds denoted by s_i, $(1 < i < p)$. The rates of transfers with the disks as well as the bandwidth of the network are not captured in the model which follows.

Moreover, we are interested in the '*perfect case*' i.e., the case where the problem size can be expressed as p sums. The concept of *lowest common multiple* (l cm) is useful in order to specify the matter in a mathematical way. In other words, we ask that the problem size n be expressed as follows:

$$n = k \times \text{lcm}(per\ f, p) \times (per\ f[0] + \cdots + per\ f[p-1]) \quad (2)$$

where k is a constant in $\mathbb{N}$, *per f* is a vector of size p containing the relative performances of the p processors of the cluster and l cm(*per f*, p) is the smallest common multiple of the p values stored in the *per f* vector.

The property can be also expressed by the fact that the size of the problem must be divisible by the sum of the values of the *per f* vector. If n cannot be written according to equation (2), different techniques as those presented in Shirazi et al. (1995) can be used in order to ensure balancing.

For example, with $k = 1$, *per f* = {8,5,3,1}, we describe a processor which is eight times faster than the slowest, the second processor is five times faster than the slowest processor, the third processor is three times faster than the slowest and we obtain that l cm({8,5,3,1}, 4) = 120. Therefore, $n = 120 + 3 \times 120 + 5 \times 120 + 8 \times 120 = 2{,}040$ is acceptable.

With problem sizes as given by equation (2), it is very easy for us to assign to each processor, an amount of data *proportional to its speed.* It is the intuitive initial idea and characterises the precondition of the problem.

3.2.2 The spring of partitioning

The key point to obtain good performances for load balancing is again the choice of the pivots. Indeed, this choice allows the partitioning of the input in portions of roughly identical size in the homogeneous case. Here, for the heterogeneous case, it is necessary to arrange the pivots so that they constitute portions of sizes proportional to the speed of each processor.

Let us consider Figure 3 to explain, without giving too complex technical details, what are the deep springs of partitioning for the heterogeneous case and according to the PSRS's manner. On Figure 3 we have three processors which have three different speeds represented by three different rectangles of different dimensions. The pivots (not the candidate) are taken with regular intervals here within the meaning of PSRS and this is materialised by the ←/→ symbols.

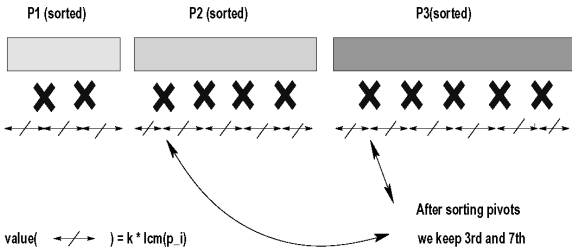


Figure 3 *The springs of partitioning on heterogeneous clusters*

The pivots on Figure 3 come from the three processors and they are supposed to be sorted. We keep here, in particular, the third pivot (on the basis of the left part of the figure). The intuitive justification is as follows: in operating that way, we notice that the number of data on the left of this first pivot is necessarily smaller than twice the size of the segment of data contained initially on the left processor (the slower one). While maintaining this invariant for the choice of the other pivots, one arrives at a result of the type of PSRS: none of the processors treats more twice what it has at the beginning.

In fact, we have adapted the proof of the PSRS algorithm which is based on similar observations. We obtain a general framework, based on sampling and which offer guarantees in terms of load balancing.

We call this algorithm H-PSRS for "Heterogeneous Parallel Sorting by Regular Sampling" technique.

4 NEW INVESTIGATIONS

In this section, we introduce two new strategies for sorting on heterogeneous clusters. Let us begin by a comparison of resources used by the three algorithms. Then, we will focus on the technical details of the algorithms.

Table 1 summarises the main properties of the different algorithms presented in this paper. H-PSS is for "Heterogeneous Parallel Sample Sort", H-PSRS is for "Heterogeneous Parallel Sorting by Regular Sampling" (see above) and H-PSOP is for 'Heterogeneous Parallel Sorting by OverPartitioning'.

Table 1 *Summary of main properties of algorithms*

Criteria	*H-PSS*	*H-PSRS*	*H-PSOP*
Number of candidates	$6 \times p' \times \log_2(p')$	$p' \times (p-1)$	$4 \times p' \times p' \times \log_2(p')$
Number of pivots	$P-1$	$P-1$	$4 \times p' \times \log_2(p') - 1$
Initial sort	No	Yes	No
Load balance (theory)	Algorithm manages partitions of size n/p' with a probability which is a function of the number of candidates	No processor has more that two times its initial load	Algorithm manages partitions of size n/p' with a probability which is a function of the number of candidates
Load balance (measured)	±15% of the optimal in the worst case on one processor	±0.1% of the optimal value	±0.01% of the optimal value
Number of files created/proc	$15 + P + 1$	$15 + P + 1$	$15 + 4 \times p' \times \log_2(p')$
Sensitivity to duplicates	?	No, until a bound of n/p' duplicates	?
Message sizes	32KB	32KB	32KB
Allocated memory	$8K \times$ sizeof(int) + $6 \times p' \times \log_2(p')$	$8K \times$ sizeof(int) + $p' \times (P-1)$	$8K \times$ sizeof(int) + $O(p')$

The reader should notice in particular, that the memory (RAM) usage as well as the number of files used in the implementations is very low. On Table 1, the constant 15 is the number of temporary files used by the polyphase merge sort we have used, *p'* is the sum of values stored in the performance vector, P is the number of processors and n is the input size.

4.1 Parallel sample sort revisited

4.1.1 Introduction

The key to success in sorting is dependent on the splitters that must partition the initial bucket into roughly equal sizes. As noted in Vitter and Shriver (1994a)

> "It seems difficult to find $S = \Theta(m)$ splitters using $\Theta(n/D)$ I/Os[2] (the formation of the buckets must be done akin to this bound to guarantee an optimal algorithm) and guarantee that the bucket sizes are within a constant factor of one another."

The Parallel Sample Sort (PSS) algorithm (Huang and Chow, 1983) and its improvement (Li and Sevcik, 1994) do not sort the portions first but use *oversampling* to select pivots. They pick $p-1$ pivots by randomly choosing $p \times s$ candidates from the entire input data, where s is the oversampling ratio, and then selecting $p-1$ pivots from the sorted candidates. Intuitively, a larger oversampling ratio results in better load balancing but increases the cost of selecting pivots.

We propose the following framework for external sorting which is based on PSS:

- pick pivots
- broadcast pivot to a master node that sorts them; keep $p-1$ pivots and broadcast them to each node
- each node partitions its input according to the pivots; broadcast the partitions
- sort the received partitions (in our case we use again polyphase merge sort).

It can be shown (Li and Sevcik, 1994) that for an unsorted list of size n, $(pk-1)$ pivots (with $k \geq 2$) partition the list into $p \times k$ sublists such that the size of the maximum sublist is less or equal to n/p with a probability of at least $1 - 2p(1 - (1/(2p)))^{pk}$.

In the case of a heterogeneous cluster (processors running at different speeds), we simulate a p' machine where p' is the sum of coefficients in the performance vector. We also set $k = 6 \log_2 p'$ to mimic the framework of Li and Sevcik (1994).

Example: For *per* $f = \{1,1,4,4\}$, $k = 3$ we obtain that the size of the maximum sublist is less or equal to n/p' with probability at least $1 - 2 \times 10(1 - (1/(2 \times 10)))^{10 \times 3 \times \log_2 10}$, that is to say, with probability $1 - 20(0.95)^{996578} = 1 - 0.12 = 88\%$.

Now, if we set $k = 6 \times \log_2 p'$ we get a probability of 99.92739%. From an 'out of core' point of view, the increase (sustained by $k = 3$ becomes $k = 6$) in the number of pivots is acceptable because the memory usage remains low!

Finally, let p' be the sum of values in the performance vector. Thus, the total number of pivots selected in our implementation of external PSS is $p' \times 6 \log p'$. Note that this number is quite low from an 'out of core' point of view, so it fits in main memory. Moreover, it is necessary a divisor of p'. Note also, that the more the cluster is unbalanced (for instance a processor is 1,000 times faster than the others) the more the probability is high. That is to say, we will have more chance to get balanced sublists. The choice of p' is thus justified to capture the heterogeneity of the machine.

4.2 Parallel sorting by overpartitioning revisited

4.2.1 Introduction

Li and Sevcik (1994) proposed an algorithm for incore sorting on homogeneous platforms with no sequential sort in the beginning. The choice and the number of pivots is done according to the discussion done in the previous section: for an unsorted list of size n, $(pk-1)$ pivots (with $k \geq 2$) partition the list into $p \times k$ sublists such that the size of the maximum sublist is less than or equal to n/p with probability at least $1 - 2_p(1 - (1/(2p)))^{pk}$.

The algorithm presented in Li and Sevcik (1994) for sorting on homogeneous platforms with the overpartitioning technique is as follows:

Algorithm 1 (PSOP (Li and Sevcik, 1994)):

Step 1: Initially, processor i has k, a portion of size n/p of the unsorted list l

Step 2: (Selecting pivots) a sample of p.k.s candidates are randomly picked from the list, where s is the oversampling ratio and k the overpartitioning ratio. Each processor picks s.k candidates and passes them to a chosen processor. These candidates are sorted and then p.k – l pivots are selected by taking (in a 'regular way') the sth, 2.sth, … ,(pk – 1)th candidates from the sample. The selected pivots d_1, d_2, … ,$d_P k - i$ are made available to all the processors

Step 3: (Partitioning) since the pivots have been sorted, each processor performs binary partitioning on its local portion. Processor j decomposes l_j according to the pivots. It produces pk sublists per processor denoted l_{jk} where jk stands for two consecutive pivots (except for the initial and final case). A sublist S_j is the union of l_{ij} with i ranging over all processors. There are pk sublists.

Step 4: (building a task queue and sorting sublists) let $T(S_j)$ denote the task of sorting S_j. The size of each sublist can be computed:

$$|S_j| = \sum_{i=1}^{p} |l_{ij}|.$$

Also, the starting position of sublist S_j in the final sorted array can be calculated:

$$\sigma_j = 1 + \sum_{h=1}^{j-1} |S_h|.$$

A task queue is built with the tasks ordered from the largest sublist size to the smallest. Each processor repeatedly takes one task $T(S_j)$ at a time from the queue. It processes the task by

- copying the p parts of the sublist into the final array at position σ_j to $\sigma_j + |S_j| - 1$
- applying a sequential sort to the elements in that range.

The process continues until the task queue is empty.

4.2.2 The heterogeneous case

The main difference in the heterogeneous case is in the way we manage partitions and select pivots.

- The number of candidates is calculated according to $4 \times p' \times p' \times \log_2(p')$ where p' is the sum of the values stored in the performance vector. After a sorting stage, we keep $4 \times p' \times \log_2(p') - 1$ pivots among the candidates. Note that this number is independent of the problem size and also that if p' grows (the cluster is more 'unbalanced'), the number of pivots grows and we amortise the risk of unbalanced partitions.

- Step 4 of Algorithm 1 is modified as follows: the partition sizes of task T_j,($1 \le j \le$ number of partitions) are broadcasted to processors and sorted. In order to decide whether processor i keeps or rejects the task of sorting partition T_j, processor i computes $T_j.size$ divided by its performance where $T_j.size$ is the number of elements in partition T_j. The obtained ratio gives an estimate of the 'execution time of task T_j'. We allocate task T_j to the processor with the smallest corresponding execution time. A special protocol is also deployed in case of a draw but we ignore such details here. We also keep, when we visit task T_j in order to decide which processor will execute it, the sum of the execution times of all previous tasks T_k, ($1 \le k \le j$) that have been allocated to processor i.

5 EFFICIENT REMOTE DISK ACCESSES

The aim of this section is to introduce issues about the way in which we can implement efficient remote disk accesses, in particular, for our sorting application. We provide details of the READ^2 library (Cozette et al., 2002, 2003) and explain why we have to devise a new implementation of external parallel sorting in order to cope with the library.

In order to share data, each cluster's node must behave like a server for other nodes. For example, in distributed parallel file systems such as PVFS (Parallel Virtual File System (Ligon and Ross, 1999)) or PPFS (Portable Parallel File System (Huber et al., 1995)), each cluster's node has the burden of serving local data requested by distant node. Whereas good performance may be obtained for homogeneous collective parallel I/O by using adequate placement and redistribution schemes (Ilroy et al., 2001), the overhead is not negligible for general access patterns. The overhead can be decomposed into two parts: first, the overhead caused by the operating system running on each node, and second, the overhead due to the architecture hardware. In this paper we only focus on the hardware overhead.

Technological advancements in the last decade made disk drives able to achieve up to 90 MByte/s of sustained bandwidth. Disk controllers can also achieve several hundreds MByte/s of bandwidth, and network cards can achieve several Gigabit/s of bandwidth! By adding several I/O components to each node, it is possible to get plenty of I/O bandwidth for data intensive applications. However, an increase of the I/O bandwidth puts more pressure on the I/O and memory buses of each node. Unfortunately, these buses cannot be scaled, so the maximum bandwidth is bounded. An alternative to the technique of using multiple disks on the same node is to use NAD (Network Attached Devices) where the disks are directly plugged into the network. Many works that include GFS (Global File System (Preslan et al., 1999)) or NASD (Network Attached Secure Disk (Gibson and Van Metter, 2000)) have proven the effectiveness of such technology. Unfortunately, this approach implies the deployment of expensive network infrastructure, e.g., Fiber Channel technology.

We are currently developing READ^2 (Remote Efficient Access to Distant Device) as an alternative to NAD, which is based on affordable network technology. In READ^2, we exploit the capability of modern network interface cards to directly drive and access I/O devices plugged in the I/O bus (usually a PCI bus). For instance, in Walton et al. (1998) the authors combine two cooperative Myrinet cards on the same I/O bus for efficient IP forwarding: data flow directly from one Myrinet card to the second one and the processor is not involved in the data path. In READ^2, we extend this technique for remote disk access.

In this paper, we study the benefit of using the READ^2 library for sorting applications involving parallel I/Os. In the next subsection, we present our architectural cluster model and describe how READ^2 I/O accesses are implemented.

5.1 Architectural model

A cluster may be considered to be an interconnection of different buses. A node is made up of:

An I/O bus: (Input/Output bus) to connect network card, disk controller and I/O bridge; it is usually the PCI bus.

A memory bus: to connect memory to I/O bridge.

A processor bus: to connect processor to I/O bridge.

The interconnection of buses is depicted on Figure 4. These buses are also characterised by the following constant parameters:

M_d:	maximum disk throughput
M_{io}:	maximum I/O bus throughput
M_m:	maximum memory bus throughput
M_c:	maximum instruction processing time.

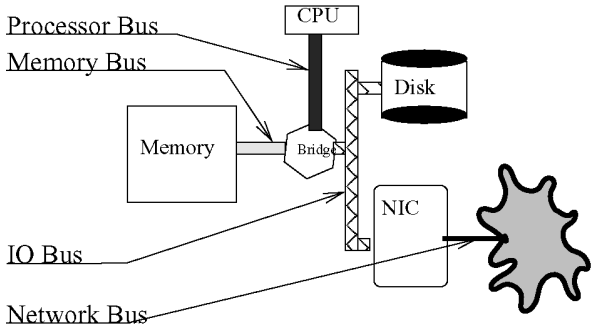


Figure 4 *Architecture and variable description*

Usually, disks are attached to the disk controller by another bus such as a SCSI bus: the global disk I/O bandwidth is the aggregated bandwidth of disks. For the sake of simplicity, we do not consider this class of busses in this paper.

In a cluster, nodes are interconnected by a network interface card plugged into the I/O bus: the network glues I/O busses to build a parallel machine. From a logical point of view, the network may be considered to be another bus

level. The network is characterised by the constant parameter M_n which is the maximum network throughput.

5.2 READ^2: remote efficient access to distant device

As pointed out previously, a cluster is usually considered to be a *shared nothing* architecture: all nodes are independent and collaborate by message passing. The processor, memory and disk of each node interact with each other. In fact, this is a high level point of view. If we consider the transport level, some node components (memory, disk) may be shared by all nodes. For instance, the SCI network technology (Bhoedjang et al., 1998) or some Virtual Shared Memory implementation based on remote DMA, allow nodes to share memory.

Usually, in a parallel file system (e.g., PVFS), data sharing is made according to a data server model: it is a *peer to peer server* approach (P2PS). A consequence is that when a node accesses data on a remote node, the remote memory bus and the distant I/O bus are involved two times in the data path. Arpaci-Dusseau et al. (1998) have demonstrated that cluster architectures are penalised by this overuse of the different buses. In particular for streaming applications, the I/O bus is usually the bottleneck. Moreover, more stress is put on the memory bus by the file system, which may involve buffer copying policy. Some works such as DAFS (Collaborative, 2001) try to remove the overhead of copies by using a remote memory access. But DAFS always involves two traversals of the I/O bus for the data path.

Here we consider another technique that we call READ^2 (Cozette and Randriamaro, 2002, 2003) for Remote Efficient Access to Distant Device. In READ^2 each node is able to directly access and control any remote disk; nodes share disks. A first consequence is that the memory is no more involved in the data path; a second consequence is that the I/O bus is involved only one time in the data path. Figures 5 and 6 are illustrations of this fact.

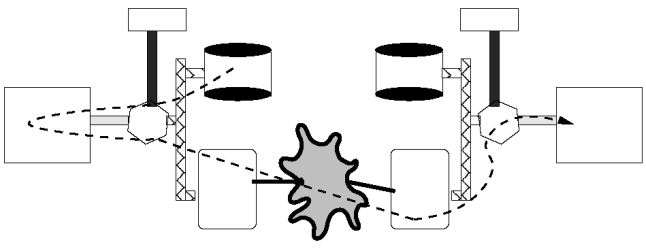


Figure 5 *Remote read data path with READ^2*

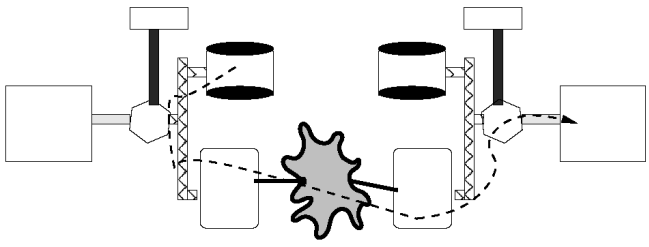


Figure 6 *Remote read data path with READ^2*

A work close to ours is OPIOM (Geoffray, 2001; Bonhomme and Geoffray, 2000). It was designed for the implementation of a distributed video on demand server on a Myrinet cluster. In OPIOM data transfers also go from disk to network directly through the Myrinet card. OPIOM uses the standard kernel driver, but in READ^2 the disk driver is embedded in the Myrinet card. Reading a remote disk does not involve the remote processor. Consequently, the remote processor is 'less' loaded: it can do more work.

We investigate now, the benefit of such an approach for external parallel sorting application on heterogeneous clusters.

5.3 External parallel sample sort

We have seen in Table 1 that PSS (Parallel Sample Sort) provides a good compromise between load balancing and execution time. So, we adapt it to fully exploit the characteristic of READ^2. Let us recall the six different parts of the algorithms.

1 choice of pivots
2 distribution of pivots in order to be sorted
3 pivot sorting stage
4 redistribution of pivots onto processors
5 partitioning and redistribution of data
6 final sorting stage.

The cost of the algorithm is the sum of all costs involved in the six parts. Our problem is to insert the functionalities of READ^2 in order to speedup the execution time. The first four steps involve few data: it is not significant to use READ^2 in these cases.

Step 5 and 6 can now be achieved in the following way. Instead of partitioning data according to pivots on the local disks and then distributing partitions, we propose to read a portion of data, then to partition data into buffers and when a buffer is full, to distribute it (thanks to READ^2) to the final disk in order to be merged later (during the last step) with other chunks of data.

Note that, in doing this we have transformed the last step into a pure (out of core) merge step instead of a sort step: a data reorganisation that sorts them during step 5 avoids at least a full file reading that corresponds to the first part of any polyphase mergesort program (see the MAKERUN procedure inside our codes) and thus we relax the memory bus usage.

We remind also that the sequential 'out of core' sorting program that we have used until now in step 5 is polyphase mergesort. It requires (during the first step) the building of auxiliary files such that the sizes of the files follow a Fibonacci series. It is too complicated to ensure such a property when using READ^2 because the latter uses only one flow i.e., only one pseudofile descriptor can be used at a time. This restriction is due to the fact that the READ^2 library is implemented below the file system layer.

Thus, we have revisited our sequential sorting brick and we have developed a 2-way like mergesort. To be short, the

principle is depicted according to the following example that uses two files and we consider blocks of 1 (sorted) integer, then blocks of 2, 4 … data:

Initial files:

[3, 8, 2, 4, 1, 10]
[5, 0, 6, 7, 9]

After the first step, we have the following auxiliary files:

[3 5, 2 6, 1 9]
[0 8, 4 7, 10]

After the second step, we have the following auxiliary files:

[0 3 5 8, 1 9 10]
[2 4 6 7]

After the third step, we have the following auxiliary files:

[0 2 3 4 5 6 7 8]
[1 9 10]

After the last step, we have the following auxiliary files:

[0 1 2 3 4 5 6 7 8 9 10]
[]

In our implementation, we work with only one file and we form (to be read or written) blocks of B data. It is just a matter of pointer management and matching of the one pseudofile descriptor requirement imposed by the library interface.

5.4 Overlapping of I/O operations and computation

To fully exploit the gain in bus usage offered by the library, we should also count on the overlapping of computation and disk transfers.

In a parallel implementation that will follow the six previous steps, it is still possible to overlap I/O operations done for the distribution of data and the merging of sorted data. The key idea is to exploit the memory bus that is free when a transfer of sorted data is done from one processor to a distant disk.

Imagine that a buffer becomes full during step 5 of our modified algorithm. Then, a READ^2 operation occurs in order to transfer the buffer to one disk. Once a second READ^2 operation occurs for the same disk, the processor attached to the disk can potentially merge the two chunks concurrently with the reception of yet another chunk.

We plan to implement this optimisation in the near future as we still encounter difficulties in multithreading the code. One difficulty in using the library is that we have to manage potentially concurrent writings to the same disk... and the READ^2 library has no support for this. One trick could be to force each node working in parallel to write on a distinct remote disk and to proceed by phases. A processor could possibly write, but not necessarily during each phase. One difficulty is to compute a bound on the number of phases and thus on the number of synchronisation points.

6 EXPERIMENTAL RESULTS

This section is divided into three parts. In the first and second parts we experiment without READ^2. We are mainly interested in the computation of the load balancing. In doing this we validate our approaches for partitioning data in the heterogeneous case. We use a small cluster composed of one Pentium III (Katmai), 451 Mhz, cache: 512 KB, RAM: 2,61,668 kB and 3 Celerons (Mendocino), 400 Mhz, cache: 128 MB, RAM: 64 MB. Disks were FUJITSU MPD3064AT disks with 512 KB of cache. Execution times and speedups are not our main concern and problem sizes are not so important here because we want to demonstrate that load balancing is under control.

Tables 2–5 are divided into five columns. From left to right, we have the mean size of data in the last step of the algorithm (Mean), the standard deviation of the mean (SD), the ratio of the mean over the optimal size, the ratio of the mean over the standard deviation and, at least the maximal and minimal observed sizes over the 35 experiments.

In the third part we validate READ^2 and exhibit the gain in using our library.

Table 2 *Heterogeneous sample sort (2 MB of data, heterogeneous configuration of performance vector)*

Mean	*SD*	*Mean/opt* (%)	*Mean/SD* (%)	*Max*	*Min*
PID0					
1,15,632	10,847	93.88	9.38	196144	100487
PID1					
3,71,959	15,981	100.09	4.29	392898	335504
PID2					
6,15,038	20,479	100.10	3.33	652183	571873
PID3					
9,86,599	20,970	100.36	2.12	1033844	947255

Table 3 *Heterogeneous sample sort (16 MB of data, heterogeneous configuration of performance vector)*

Mean	*SD*	*Mean/opt (%)*	*Mean/SD (%)*	*Max*	*Min*
PID0					
930196	87822	94.62	9.44	1108952	759477
PID1					
2935879	157911	99.51	5.38	3364108	2616418
PID2					
4974058	140542	101.2	2.82	5379087	4709106
PID3					
7871546	211648	100.36	2.69	8153092	7436021

Table 4 *Heterogeneous sample sort (16 MB of data, homogeneous configuration of performance vector)*

Mean	*SD*	*Mean/opt* (%)	*Mean/SD* (%)	*Max*	*Min*
PID0					
3918862	584064	93.78	14.90	5744959	3045409
PID1					
4150519	625341	99.34	15.06	6083675	2942227
PID2					
3935862	579492	94.2	14.72	5423733	2755896
PID3					
4706443	505979	112.65	10.75	5719919	3699471

Table 5 *Heterogeneous sample sort (2 MB of data, homogeneous configuration of performance vector)*

Mean	*SD*	*Mean/opt* (%)	*Mean/SD* (%)	*Max*	*Min*
PID0					
472349	59876	90.45	12.67	629023	377398
PID1					
526403	52376	100.79	9.95	604596	434675
PID2					
525042	62448	100.53	11.89	654503	384518
PID3					
564548	47235	108.10	8.36	645109	452583

6.1 Heterogeneous sample sort

We set the performance vector to {1,3,5,8} and we observe the load balancing factor (in the 'Mean/opt' column). The choice of those values is arbitrary and at the same time reflects a significant heterogeneity. Note that it does not correspond to the physical machine as we measure here, load balancing and not the execution time.

A first result is presented in Table 2. We sort n = 20,88,960 integers and we use our benchmark numbered 0 (random generated data using the linear congruent generator $x_{k+1} = ax_k (\mathrm{mod}\ 2^{46})$).

After that, we set again the performance vector to {1,3,5,8} and observe the load balancing factor (in the 'Mean/opt' column) of Table 3. Here we sort n = 1,67,11,680 integers.

When we compare the results of the load expansion metric of Tables 2 and 3, we observe a very good metric. The choice made for the number of pivots is appropriate.

6.1.1 Sample Sort with a performance vector configuration as a homogeneous cluster

We now configure our algorithm with the following setting for the performance vector: {1,1,1,1}.

A first result is presented on Table 4. We sort n = 1,67,11,680 integers (the optimal amount of data/processor is 41,77,920 integers). We start 35 experiments and we observe a mean execution time of 82.15 seconds (the standard deviation is 6.75 seconds).

Table 6 *Heterogeneous PSOP (1973785 integers, heterogeneous configuration of performance vector)*

Mean	*SD*	*Mean/opt* (%)	*Max*	*Min*
PID0				
929386	229	100.059	929858	929018
PID1				
580687	225	100.027	581129	580170
PID2				
347791	143	99.85	348123	347339
PID3				
115920	184	99.84	116121	115202

A second result is presented in Table 5. Here, we sort n = 20,88,960 integers (the optimal amount of data/processor is 5,22,240 integers). We start 35 experiments and we observe a mean execution time of 6.25 seconds (the standard deviation is 0.56 seconds).

Table 7 *Heterogeneous PSOP (16777215 integers, heterogeneous configuration of performance vector)*

Mean	*SD*	*Mean/opt* (%)	*Max*	*Min*
PID0				
7898307	1690	100.04	7901360	7895160
PID1				
4936858	1621	100.05	4939629	4933891
PID2				
2956149	1414	99.84	2959340	2953082
PID3				
985900	1115	99.89	987923	2953082

Again, the results of the load expansion metric are good. All the results validate the approach for both the heterogeneous case and the homogeneous case. So, the external parallel sample sort algorithm developed in this section is of general use.

Our last remark concerns the way we fill the performance vector. We have previously said that a vector filled with the same values (1) represents the 'homogeneous case'. If we entirely set the vector with value 10 we also model the 'homogeneous case'. But if we run the program according to this setting we will generate more pivots!

6.2 Heterogeneous parallel sorting by over partitioning

We set now the performance vector to {8,5,3,1}. Tables 6 and 7 present two experiments for input sizes of 19,73,785 and 1,67,77,215 integers stored initially on disks of a cluster of four processors. We notice that the load expansion metrics (columns Mean/opt) are very good, as well as the standard deviation of the observed values (column SD). The maximal and minimal values observed on processors are also good. We conclude that the number of pivots $(4 \times 17 \times \log_2(17) - 1 = 271$ is very low when compared

with the input size) and is good enough to ensure a quasiperfect load balance of the work. As a consequence, the overhead due to the processing of the partitions in the last step of the algorithm is kept low because we have fewer data to manage.

6.3 Validation of READ^2 through our sequential sorting algorithm

We estimate now, the cost of remote stores and we validate READ^2 with our sequential 'out of core' sorting program that requires an important traffic to/from the disks that is mixed with computation. The sequential 'out of core' sorting algorithm is part of the parallel 'out of core' algorithm which is not yet implemented in full detail.

The first column of Table 8 corresponds to the input size given in number of integers. The second column shows that we entirely sort with the local disk. The third column gives experiment results when using the remote disk over NFS and the fourth column gives results when we use the distant disk but our implementation is made over READ^2 i.e., with the READ^2 library implemented in C. The numbers in the three last columns correspond to seconds.

Table 8 *Comparison of systems*

#data to sort	*Local disk*	*Remote with NFS*	*Remote with READ^2*
17000000	195	624	232
19000000	221	613	258
21000000	240	613	289
23000000	253	665	319
25000000	290	703	791

We use two Alpha AXPs, 433 Mhz with 32 MB of memory, a Myrinet card and a 40 MB/s hard disk on each node. We fix at eight, the number of integer read or write to the disk with a `fread, fwrite` operation. A value of 512/4 = 128 corresponding to the disk buffer size in the system should be more adequate for performance. But we have made the choice to experiment in unfavourable conditions. Note that all the sizes in the table correspond to input sizes that are larger than the memory size available on the nodes:

We observe that the results for NFS are bad. It was expected. It is known that NFS does not perform well, when performance is a critical issue (especially, in high performance applications). Among other explanations, NFS uses a lot of processor resources and the data cross twice, the remote bus: the data come from the remote disk to the remote memory and then from the remote memory to the remote network card. After that, they are sent to the local host.

The results, when we use the READ^2 library to reduce these bus usages, are good when compared with the results obtained for sorting on the local disk. In fact, READ^2 includes the disk driver in the network card, so it bypasses the remote processor and the data are sent directly from the remote disk to the remote network card.

Except for the last line, we observe a penalty of about 15–21% in using the library vs. doing the sort on the local disk, which is good. We have no explanation for the result of the last line: a severe penalty is observed and even worth it; the result for NFS is better than the result for the library. Such observations have not been made previously (Cozette et al., 2002, 2003).

One assumption about the phenomenon is that NFS uses a cache that could improve the performance. Another one is that since the disk block size is small (eight integers), the overhead due to message headers in transfers becomes important. Further experiments in incrementing the block size to 128 and for an input size of 2,50,00,000 integers have shown that the penalty is about 5% between the execution on the local disk and execution with the use of the library, which is excellent. Moreover, we do not yet know if performance difference gets worse for larger arrays.

The performance of read^2 should be investigated more in depth in the future and should be compared also with a parallel file system (such as PVFS).

7 CONCLUSION

In this paper we devised algorithms for external parallel sorting. We depicted solutions when the platform is made of processors running at different speeds that remain constant during the execution. We also showed how to adapt these solutions to exploit efficiently the READ^2 library able to do fast remote read/write operations to disks. The difficulty lies mainly not in the parallel algorithm itself, because its principle is simple. It lies in the technical constraints imposed by the library interface and also in the choice of the 'out of core' sequential sorting implementation that can offer more or less computation/disk transfers overlapping.

We intend, when the READ^2 library will be completed to offer a 'full' file system interface, to experiment with our codes over such a file system in order to benchmark it and to compare our results with those of PVFS for instance.

ACKNOWLEDGEMENT

We would like to thank Dr. Zaher Mahjoub, University of Tunis El Manar, for his reading of this paper. We would like also thank reviewers for their valuable suggestions regarding the content and the structure of this paper.

REFERENCES

Aggarwal, A. and Vitter, J.S. (1988) 'The input/output complexity of sorting and related problems', *Communications of the ACM*, Vol. 31, No. 9, September, pp.1116–1127, [Online]. Available: http://www.acm.org/pubs/toc/Abstracts/0001-0782/48535.html

Akl, S. (1985) *Parallel Sorting* Algorithms, Academic Press, Orlando, Florida.

Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C., Culler, D.E., Hellerstein, J.M. and Patterson, D.A. (1998) *The Architectural Costs of Streaming I/O: A Comparison of Workstations, Clusters and SMPs*, HPCA, pp.90–101.

Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J. and Zagha, M. (1991) 'A Comparison of sorting algorithms for the connection machine CM-2', *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, July, pp.3–16.

Bonhomme, A. and Geoffray, P. (2000) 'High performance video server using myrinet', *1st Myrinet User Group (MUG),* Lyon, 28–29 September.

Cérin, C. (2002) 'An out-of-core sorting algorithm for clusters with processors at different speed', *16th International Parallel and Distributed Processing Symposium (IPDPS),* Available on CDROM from IEEE Computer Society, Ft Lauderdale, Florida, USA.

Cérin, C., Jemni, M. and Fkaier, H. (2003) 'A synthesis of parallel out-of-core sorting programs on heterogeneous clusters', *3st International Symposium on Cluster Computing and the Grid, Tokyo*, May.

Collaborative, (2001) *Direct Access File System version 1.0*, http://www.dafscollaborative.org/.

Cormen, T.H. and Hirschl, M. (1997) 'Early experiences in evaluating the parallel disk model with the ViC* implementation', *Parallel Computing*, Vol. 23, Nos. 4–5, May, pp.571–600.

Cozette, G.U.O. and Randriamaro, C. (2002) 'Improving cluster io performance with remote efficient access to distant device', *27th Annual IEEE Conference on Local Computer Networks (LCN'02),* IEEE CS Press, Tampa, Florida. November, pp.629–638.

Cozette, G.U.O. and Randriamaro, C. (2003) 'Read^2: put disks at network level', *3rd International Symposium on Cluster Computing and the Grid (CCGRID'03),* IEEE CS Press, May, Tokyo, Japan, pp.698–704.

Cozette, O., Randriamaro, C. and Utard, G. (2002) 'Improving cluster io performance with remote efficient access to distant devices', *Workshop on High Speed Local Network (HSLN'02)*, Tampa, USA, November.

Cozette, O., Randriamaro, C. and Utard, G. (2003) 'Read2: Put disks at network level', *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, p.698.

DeWitt, D.J., Naughton, J.F. and Schneider, D.A. (1991) 'Parallel sorting on a shared-nothing architecture using probabilistic splitting', *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December, pp.280–291 [Online] Available: http://www.cs.wisc.edu/Dienst/UI/2.0/Describe/ncstrl.uwmadison/CS-TR-1991-1043.

Geoffray, P. (2001) 'Opiom: off-processor io with myrinet', *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID'01)*, Brisbane, Australia, 15–18 May.

Gibson, G.A. and Van Metter, R. (2000) 'Network attached storage architecture', *Communication of the ACM*, Vol. 43, No. 11, November, pp.37–45.

Wolfgang Hoschek, A.S.H.S., Jaen-Martinez, J. and Stockinger, K. (2000) *Data Management in An Internationnal Data Grid Project, GRID 2000*, Bangalore, Inde, 17–20 December.

Huang, J.S. and Chow, Y.C. (1983) 'Parallel sorting and data partitioning by sampling', *IEEE Computer Society's Seventh International Computer Software and Applications Conference (COMPSAC'83)*, November, pp.627–631.

Huber, J., Elford, C.L., Reed, D.A., Chien, A.A. and Blumenthal, D.S. (1995) 'PPFS: a high performance portable parallel file system', *Proceedings of the 9th ACM International Conference on Supercomputing*, ACM Press, Barcelona, July, pp.385–394. [Online] Available: http://www-pablo.cs.uiuc.edu/Papers/ICS95-ppfs.html.

Ilroy, J., Randriamaro, C. and Utard, G. (2001) 'Improving MPI-IO performance on PVFS', *EuroPar '2001*, Manchester, UK, August, pp.911–915.

Kim, M.Y. (1986) 'Synchronized disk interleaving', *IEEE Transactions on Computers*, Vol. C–35, No. 11, November, pp.978–988.

Knuth, D.E. (1998) 'Sorting and searching', 2nd ed., *The Art of Computer* Programming, Reading, MA, Addison-Wesley, USA, Vol. 3.

Li, H. and Sevcik, K.C. (1994) 'Parallel sorting by overpartitioning', *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, ACM Press, New York, NY, USA, June, pp.46–56.

Ligon III, W. and Ross, R.B. (1999) 'An overview of the parallel virtual file system', *Proc. of the 1999 Extreme Linux Workshop*, June.

Nodine, M.H. and Vitter, J.S. (1995) 'Greed sort: Optimal deterministic sorting on parallel disks', *Journal of the ACM*, July, Vol. 42, No. 4, pp.919–933. [Online] Available: http://www.acm.org/pubs/toc/Abstracts/0004-5411/210343.html

Pearson, M.D. (1999) *Fast out-of-core sorting on parallel disk systems*, Dept. of Computer Science, Dartmouth College, Hanover, NH, Tech. Rep. PCS-TR99-351, June. [Online]. Available: ftp ://ftp. cs. dartmouth. edu/TR/TR99-3 51.ps.Z

Preslan, K.W., Barry, A.P., Brassow, J.E., Erickson, G.M., Nygaard, E., Sabol, C.J., Soltis, S.R., Teigland, D.C. and O'Keefe, M.T. (1999) 'A 64-bit, shared disk file system for Linux', *16th Mass Storage Systems Symposium*, IEEE, San Diego, 15–18 March.

Rajasekaran, S. (1998) 'A framework for simple sorting algorithms on parallel disk systems (extended abstract)', *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures,* Department of CISE, University of Florida, Gainesville, FL.

Bhoedjang, R.A.F., Ruhl, T. and Bal, H.E. (1998) 'User-Level Network Interface Protocols', *Computer*, Vol. 31, No. 11, November, pp.53–60.

Salem, K. and Garcia-Molina, H. (1986) 'Disk striping', *Proceedings of the 2nd International Conference on Data Engineering*, ACM, February, pp.336–342.

Schikuta, E. and Kirkovits, P. (1996) 'Analysis and evaluation of sorting for parallel database systems', *Proc. Euromicro 96, Workshop on Parallel and Distributed Processing, Braga, Portugal*, IEEE Computer Society Press, January, pp.258–265.

Shi, H. and Schaeffer, J. (1992) 'Parallel sorting by regular sampling', *Journal of Parallel and Distributed Computing*, Vol. 14, No. 4, pp.361–372 [Online] Available: http://web.cs.ualberta.ca/jonathan/Papers/psrs1.ps.Z.

Shirazi, B.A., Hurson, A.R. and Kavi, K.M. (1995) *Scheduling and Load Balancing in Parallel and Distributed Systems*, ISBN 0-8186-6587-4, Computer Society Press, Piscataway NJ, USA.

Talagala, N., Asami, S., Patterson, D., Futernick, B. and Hart, D. (2000) 'The art of massive storage: a web image archive', *IEEE Computer Journal*, Vol. 33, No. 11, November, pp.22–28.

Vitter, J.S. and Shriver, E.A.M. (1994a) 'Algorithms for parallel memory I: two-level memories', *Algorithmica*, Vol. 12, Nos. 2–3, August and September, pp.110–147.

Vitter, J.S. and Shriver, E.A.M. (1994b) 'Algorithms for parallel memory II: hierarchical multilevel memories', *Algorithmica*, Vol. 12, Nos. 2–3, August and September, pp.148–169.

Walton, S., Hutto, A. and Touch, J. (1998) 'Efficient high-speed data paths for IP forwarding using host based routers', *10th IEEE Workshop on Local and Metropolitan Area Networks*, Vol. 31, No. 11, November, pp.46–52.

NOTES

[1]A *run* is a sequence of sorted records.
[2]Remind that *m* and *D* are defined in Section 2.1.

WEBSITES

http://now.cs.berkeley.edu/NowSort/index.html/.
http://research.microsoft.com/barc/SortBenchmark/.
http://www.nas.nasa.gov/.
http://www.ordinal.com/.