A Synthesis of Parallel Out-of-core Sorting Programs on Heterogeneous Clusters*

Christophe Cérin,

Université de Picardie Jules Verne
LaRIA, Bat Curi, 5 rue du moulin neuf
F-80000 Amiens - France
cerin@laria.u-picardie.fr

Hazem Fkaier,
Mohamed Jemni
Université de Tunis EL MANAR, Faculté des Sciences de Tunis, Laboratoire LIP2
1060 Belvédère Tunis - Tunisie
hazem.fkaier@fst.rnu.tn,
mohamed.jemni@fst.rnu.tn

Abstract

The paper considers the problem of parallel external sorting in the context of a form of heterogeneous clusters. We introduce two algorithms and we compare them to another one that we have previously developed. Since most common sort algorithms assume high-speed random access to all intermediate memory, they are unsuitable if the values to be sorted don't fit in main memory. This is the case for cluster computing platforms which are made of standard, cheap and scarce components. For that class of computing resources a good use of I/O operations compatible with the requirements of load balancing and computational complexity are the key to success. We explore three techniques and show how they can be deployed for clusters with processor performances related by a multiplicative factor. We validate the approaches in showing experimental results for the load balancing factor.

Keywords: *Out-of-Core parallel sorting algorithms on clusters, Performance Evaluation and Modeling of Parallel Integer Sorting Algorithms, Sorting by Regular Sampling and by Over-partitioning, Data Distribution, I/O and resource management, Load Balancing.*

1 Introduction

It is often said that 25 to 50 percent of all the work performed by computers is being accomplished by sorting algorithms [Akl85]. 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 and parallel systems in real situations.

A special class of *non homogeneous clusters* is under concern in the paper. We mean clusters whose global performances are correlated by a multiplicative factor.

This class of machines is of particular interest for two kinds of customers: first, for those who cannot replace instantaneously the whole components of its cluster with a new processor or disk generation but shall compose with old and new processors or disks and second for people sharing cpu-time because the cluster is not a dedicated one.

We focus on the ways to 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, during the execution, with more than $k.n$ integers with the requirement that k should be as low as possible and must be as close as possible to the optimal 1.

In our algorithms, a vector containing the relative "speeds" of nodes is used to partition the work evenly. The vector management is the key of the success. Representing the relative performance of participating nodes according to a simple integer vector is certainly weak. What should be shown in the future is the relevance of the factor compared with performance of the node including CPU power, local disk speed and bandwidth as well as network bandwidth. However, we consider that our methodology is the first attempt to deal with sorting in the heterogeneous case and we exhibit bounds on the load balancing factor.

The organization of the paper is the following. In Section 2 we recall some techniques of common use for external sorting on homogeneous clusters. Section 3 is related to an algorithm we introduced in the past for sorting on heterogeneous clusters. Section 4 introduces two new algorithms and we compare them in

terms of load balance and resources. Section 5 is devoted to experiments to validate our implementations. Section 6 concludes the paper.

2 External Parallel Sorting

Parallel sorting algorithms under the framework of out-of-core computation is not new. The most valuable and recent publications to our opinion are [SK96], [Raj98], [CH97], [Pea99], [NV95] and since our work is based on sampling techniques, we shall mention the 'ante-cluster' reference [DNS91] which summarize all the work in the field prior to 1991.

The objectives in these articles are to minimize the round disk trip or the number of times we access the disks which is very costly with current disk technology comparing to the memory to memory or cache access time. References [SGM86], [Knu98], [Kim86] are examples of techniques to deal efficiently with disks and they investigate the case of sequential external sorting algorithms.

When researchers study performances, the parallel execution time is not the only center of interest. Blelloch in [BLM91] defines the concept *sublist expansion metric* as being the ratio between the size of the largest list treated by a processor in one moment of the algorithm to the average expected size. In other words, this metric accounts for load balancing: ideally, a value 1 is awaited. In this case load balancing is optimal. In this paper, the *load expansion metric* is the primary metric of interest.

We are interested in a particular case 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 can have *different speeds*. It is a first level of heterogeneity and it introduces new challenges.

It also seems to us that there is not a lot of literature (in our knowledge) for this class of architecture, many articles (on sorting) on clusters treat homogeneous case only.

Concerning the problem of measuring effectively the relative speed of processors, we suppose the existence of precise techniques to realize it.

2.1 Related works on homogeneous clusters

Sorting is also present in several test sets which one can find on the site of NASA[1]. The NOW[2] in Berkeley is without any doubt the first project (1997) concerning sorting on clusters. It is interested in a homogeneous cluster of SPARC processors and in the performances in time only. In this work, one is not interested in the problems of load balancing. The project `NowSort` currently rebounds around the project `Millennium`[3] which proposes to develop a cluster of clusters on the campus of Berkeley in order to support applications of scientific computation, simulation and modeling.

We also find people from industry that are specialists in sorting. For example, the `Ordinal` company [4] distributes `Nsort` for various platforms, generally multiprocessors machines. As title of example, in 1997, an Origin2000 with 14 R10000 processors at 195Mhz, 7GB of RAM and 49 disks was able to sort 5.3GB of data in 58 seconds.

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

Year 2002 results show that Minute sort performance is 21.8 GB (218 million records) in 56.51s on a cluster.

3 Related work on heterogeneous clusters

The strategy of sampling (of pivots) is studied in this section for sorting on an heterogeneous cluster.

Intuitively it is a question of insulating in the input vector the pivots which would partition it in segments of equal sizes. This phase selects pivots so that between two consecutive pivots, there is the same number of objects. After a phase of redistribution of the data according to values of the pivots, it does not remain any more only to sort the values locally.

A first attempt for this strategy was successfully introduced and implemented in [Cér02]. The underlying algorithm involves only one communication step: when data move from one disk to another disk, they necessary go to the right place! Such property is important for clusters made of affordable disk systems because it reduces the overheads of data movements across disks. Thus we need a limited number of communication steps in order to avoid also 'to be slowdown' by the bandwidth of the network. We claim again that our algorithms are adequate for cheap clusters.

The problem is introduced as follows: n data (without duplicates) are physically distributed on p processors.

[1] `http://www.nas.nasa.gov/`
[2] `http://now.cs.berkeley.edu/NowSort/`
[3] `http://www.millennium.berkeley.edu/`
[4] See: `http://www.ordinal.com/`

The processors are characterized by their speeds and their relative differences are stored in the `perf` vector. 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. for the case where the size of the problem can be expressed as p sums. The concept of *lowest common multiple* is useful in order to specify the things here in a mathematical way. In other words, we ask so that the problem size n be expressed in the form:

$$n = k * perf[0] * \mathrm{lcm}(perf, p) + ...+ \\ k * perf[p-1] * \mathrm{lcm}(perf, p) \qquad (1)$$

where k is a constant in $\mathbb{N}$ which represents a number of integers, $perf$ is a vector of size p which contains the relative performances of the p the processors of the cluster, $\mathrm{lcm}(perf, p)$ is the smallest common multiple of the p values stored in the $perf$ vector.

For example, with $k = 1, perf = \{8, 5, 3, 1\}$, we describe a processor who runs 8 times more quickly than the slowest, the second processor is running 5 times more quickly than the slowest processor, the third processor is running 3 times more quickly than the slowest and we obtain that $\mathrm{lcm}(\{8, 5, 3, 1\}, 4) = 120$ and thus $n = 120 + 3 * 120 + 5 * 120 + 8 * 120 = 2040$ is acceptable.

In an equivalent way, note that n should be a multiple of the sum of the values stored in the `perf` vector.

Thus, with problem sizes of the form of equation 1, it is very easy for us to assign to each processor a quantity of data *proportional to its speed*. It is the intuitive initial idea and it characterizes the precondition of the problem. If n cannot be expressed itself as with the equation 1, different techniques as those presented in [ARM95] can be used in order to ensure balancing.

4 New results

In this section we introduced two new results for sorting on heterogeneous clusters. Let us introduce first, on Table 7 page 8, a comparison of resources used by the three algorithms. We focus on memory usage, the number of files and the load balance factor. Then we will focus on the technical details of the two algorithms.

On Table 7 page 8 H-PSS means "Heterogeneous Parallel Sample Sort", H-PSOP means "Heterogeneous Parallel Sorting by Over-Partitioning" and H-PSRS means "Heterogeneous Parallel Sorting by Regular Sampling". H-PSRS was developed in [Cér02].

Reader should notice in particular that the memory (RAM) usage is very low as well as the number of files open simultaneously and used in the implementations. On Table 7, constant 15 is the number of temporary files used by the polyphase merge sort we have employed, 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 of the success in sorting is dependent of the pivots that must partition the initial bucket into roughly equal sizes.

The Parallel Sample Sort (PSS) algorithm [HC83] and its improvement [LS94] does not sort the portions first but it uses *oversampling* to select pivots. It picks $p-1$ pivots by randomly choosing $p * s$ 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:

1. Pick pivots in a proportional way to the `perf` vector; See below for details;
2. Send pivots to a master node that sorts them; keep $p-1$ pivots and broadcasts them to each node;
3. Each node partitions its input according to the pivots and sends the partitions to appropriate processors;
4. Sort the received partitions (in our case we use a polyphase merge sort).

It can be shown [LS94] that for an unsorted list of size n, $(pk-1)$ pivots (with $k \geq 2$) partition the list into $p * k$ sublists such that the size of the maximum sublist is less or equal to n/p with probability at least $1 - 2p(1 - (1/(2p)))^{pk}$

In the case of an heterogeneous cluster (processors at different speeds) we simulate a p' machine with p' equals the sum of coefficients in the performance vector.

Example: for perf= $\{1, 1, 4, 4\}$, if $k = 3\log_2 10$ we obtain that the size of the maximum sublist is less or equal to n/p' with probability at least $1 - 2 * 10(1 - (1/(2 * 10)))^{10*3*log_2 10}$ that is to say with probability $1 - 20(0.95)^{99.6578} = 1 - 0.12 = 88\%$.

Now, if we set $k = 6 * \log_2 p'$ we get a probability of 99.92739%. For an out-of-core point of view, the

increase (sustained by $k = 3$ becomes $k = 6$) on the number of pivots is acceptable because the memory usage stays low!

We set k with $6 \log_2 p'$ to mimic the framework of Li and Sevcik [LS94].

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: $p' * 6 \log p'$. Note that this number is quiet low for an out-of-core point of view, so the corresponding integers fit in main memory. Moreover, it is necessary a divisor of p'. Note also that more the cluster is unbalanced (for instance a processor is 1000 times more powerful than the others) 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 Over-Partitioning revisited

4.2.1 Introduction

Li and Sevcik in [LS94] proposed an algorithm for in-core sorting on homogeneous platforms with no sequential sort in the beginning. The choice and the number of pivots is done according to the discussion of the previous section: for an unsorted list of size n, $(pk - 1)$ pivots (with $k \geq 2$) partition the list into $p * k$ sublists such that the size of the maximum sublist is less or equal to n/p with probability at least $1 - 2p(1 - (1/(2p)))^{pk}$.

The algorithm presented in [LS94] for sorting on homogeneous platforms with the over-partitioning technique is as follows:

Algorithm 1 (PSOP [LS94])

Step 1 *initially, processor i has l_i, 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 over partitioning ratio. Each processor picks $s.k$ candidates and passes them to a designated processor. These candidates are sorted and then $p.k-1$ pivots are selected by taking (in a 'regular way') $s^{th}, 2.s^{th}, \cdots, (pk-1)^{sh}$ candidates from the sample. The selected pivots $d_1, d_2, \cdots, d_{pk-1}$ 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 i decomposes l_i according to the pivots. It produces pk sublists per processor denoted l_{mj} where m, j stands for two consecutive pivots (except for the initial an final case). A sublist S_j is the union of l_{mj} with m ranging over all processors. There is pk sublists.*

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

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

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

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

A task queue is built with the tasks ordered from the largest sublists size to smallest. Each processor repeatedly takes one task $T(S_j)$ at a time from the queue. It processes the task by (a) copying the p parts of the sublist into the final array at position σ_j to $\sigma_j + \mid S_j \mid -1$, and (b) 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 in the way we select pivots. First, the number of candidates is calculated according to $4 * p' * p' * \log_2(p')$ where p' is the sum of the values stored in the performance vector. After a sorting stage, we keep $4 * p' * \log_2(p') - 1$ pivots among the candidates since we have set $k = 4 * \log_2(p')$ and $s = p'$ according to the probability formula given above.

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 amortize the risk of unbalanced partitions.

Second, Step 4 of Algorithm 1 is modified as follows: the partition sizes of task $T_j, (1 \leq j \leq$ number of partitions) are broadcasted to processors and sorted. In order to decide if processor i keeps or rejects the task of sorting partition T_j, processor i computes $T_j.size$ divided by its performance where $T_j.size$ represents the number of elements in partition T_j. The ratio of $T_j.size$ and the performance gives an estimate of the "execution time of task T_j". We allocate task T_j on processor with the smallest 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 execution time of all previous tasks $T_k, (1 \leq k < j)$ that has been allocated to processor i.

5 Preliminary results

In this section we focus mainly on the load balance factor. In doing this we validate our approaches. We do not yet investigate the execution times on an heterogeneous cluster. We use a small homogeneous cluster composed of one Pentium III (Katmai), 451 Mhz, cache: 512KB, RAM: 261668 kB and 3 Celerons (Mendocino), 400 Mhz, cache: 128MB, RAM: 64MB. Disks were FUJITSU MPD3064AT disks with 512KB of cache.

We guess it is sufficient in order to study the load balancing factor and to isolate the main properties of our codes. We show that we obtain good load balancing factors both in the case of heterogeneous clusters and homogeneous clusters (simulated).

Tabular 1 and also 2, 3, 4 are divided into five columns. From left to right, we have the mean size of data in the last step of the algorithm (column Mean), the standard deviation of the mean (SD), the ratio of the mean over the optimal size (the values in the columns should be close to 100% and represent the quality of the load balance), the ratio of the mean over the standard deviation and, at least the maximal and minimal observed sizes over the 35 experiments (these values can be compared to the mean size to appreciate how algorithms capture the extreme cases).

5.1 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). A first result is presented on Table 1. We sort $n = 2088960$ integers and we use our benchmark numbered 0 (random generated data using the linear congruential generator $x_{k+1} = ax_k (\text{mod } 2^{46})$).

After that, we set again the performance vector to $\{1, 3, 5, 8\}$ and we observe the load balancing factor (in the "Mean/opt" column) of Table 2. Here, we sort $n = 16711680$ integers.

When we compare the results about the load balancing factors of Tables 1 and 2 we observe very good metrics, close to 100% for all cases. The choice made for the number of pivots is appropriate.

5.1.1 Sample Sort with a performance vector configuration as an homogeneous cluster

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

Mean	SD	Mean/opt	Mean/SD	$\frac{Max}{Min}$
PID0				
115632	10847	93.88%	9.38%	$\frac{96144}{10847}$
PID1				
371959	15981	100.09%	4.29%	$\frac{392898}{335504}$
PID2				
615038	20479	100.10%	3.33%	$\frac{652183}{571873}$
PID3				
986599	20970	100.36%	2.12%	$\frac{1033844}{947255}$

Table 1. Heterogeneous Sample Sort (2MB of data, heterogeneous configuration of perf vector)

A first result is presented on Table 4. We sort $n = 16711680$ integers (the optimal amount of data per processor is 4177920 integers). We start 35 experiments and we observe a mean execution time of 82.15 seconds (the standard deviation is 6.75 seconds).

A second result is presented on Table 3. Here, we sort $n = 2088960$ integers (the optimal amount of data per processor is 522240 integers). We start 35 experiments and we observe a mean execution time of 6.25 seconds (the standard deviation is 0.56 seconds).

Again, the results (see again Tables 1 to 4) about the load expansion metric are good. All the results validate in terms of load balancing the approach both for the heterogeneous case and for 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 said previously that a vector filled with the same value (1) represents the "homogeneous case". If we set entirely 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!

5.2 Heterogeneous Parallel Sorting by Over-Partitioning

We set the performance vector to $\{8, 5, 3, 1\}$. Tabular 5 and 6 propose two experiments for input sizes of 1973785 and 16777215 integers stored initially on disks of a cluster of four processors. We notice that the load expansion metric (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 ($4*17*log_2(17)-1 = 271$ is

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

Table 2. Heterogeneous Sample Sort (16MB of data, heterogeneous configuration of perf vector)

very low comparing to the input size) is good enough to ensure a quasi-perfect 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 Conclusion

In this paper we introduced two new algorithms for sorting on heterogeneous clusters and we compared them to a third algorithm that we introduced in [Cér02]. We obtain a family of three algorithms that are efficient for sorting on clusters: all algorithms have good properties when we consider the load balancing factor and they can be compared (see Table 7) in terms of resources that they use. We control explicitly the number of intermediate files, memory size and messages size. A simple modification of our codes concerning a buffer declaration allows us to manage both homogeneous and heterogeneous clusters. Codes are freely available at `http://www.laria.u-picardie.fr/~cerin/=paladin`.

We are currently experimenting with READ^2 [CRU02] library, an efficient implementation of remote SCSI disk accesses. In READ^2 any cluster's node directly access to a remote disk of a distant node: the distant processor and the distant memory are removed from the control and data path. With this technique, a cluster can be considered as a *shared disk* architecture instead of a *shared nothing* one, and may inherit works from the SAN community.

Our objective is to validate the READ^2 library starting with a real application: at present time, only preliminary results are available for READ^2. In the current implementation (oct 2002), a program interface allows the programmer to explicitly write from the local memory of a processor to a distant disk. The expected gain of using READ^2 will be compared to the measured gain for our H-PSS implementation over that we are re-coding to use READ^2.

For H-PSS, it is not hard to observe that we have about $((p-1)N_i)/p$ data (in the most favorable case for the partitioning), where N_i is the initial amount of data on the local disk of processor i, that move from one node to another disk during the redistribution of data. It is an important amount of information. In this case, READ^2 should bring us a significant speedup both in terms of io-bus usage but also in terms of memory-bus usage. Since the memory bus usage is expected to be reduced, a supplementary question arises: is it possible to use more memory bandwidth to start the final sequential external sort concurrently with the redistribution of data in order to overlap communication and computation more efficiently?

References

[Akl85] Selim Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[ARM95] Behrooz A.Shirazi, Ali R.Hurson, and Krishna M.Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE CS press, 1995.

[BLM91] G. Blelloch, C. Leiserson, and B. Maggs. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

[Cér02] Christophe Cérin. An out-of-core sorting algorithm for clusters with processors at different speed. In *16th International Parallel and Distributed Processing Symposium (IPDPS), Ft Lauderdale, Florida, USA*, page Available on CDROM from IEEE Computer Society, 2002.

[CH97] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the parallel disk model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, May 1997.

[CRU02] Olivier Cozette, Cyril Randriamaro, and Gil Utard. Improving cluster io performance with remote efficient access to distant device. In *Proceedings of the Workshop on High-Speed local Netwoks, held in conjunction with the 27th Annual IEEE Conference on Local Computer Networks (LCN)*, Embassy Suites Hotel, Tampa, Florida, november 2002.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291, December 1991.

[HC83] J. S. Huang and Y. C. Chow. Parallel Sorting and Data Partitioning by Sampling. In *IEEE Computer Society's Seventh International Computer Software & Applications Conference (COMPSAC'83)*, pages 627–631, November 1983.

[Kim86] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers, Vol.C-35*, (11), November 1986.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[LS94] Hui Li and Kenneth C. Sevcik. Parallel sorting by overpartitioning. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, pages 46–56, New York, NY, USA, June 1994. ACM Press.

[NV95] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[Pea99] Matthew D. Pearson. Fast out-of-core sorting on parallel disk systems. Technical Report PCS-TR99-351, Dept. of Computer Science, Dartmouth College, Hanover, NH, June 1999.

[Raj98] Rajasekaran. A framework for simple sorting algorithms on parallel disk systems (extended abstract). In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk Striping. In *Proceedings of the 2^{nd} International Conference on Data Engineering*, pages 336–342. ACM, February 1986.

[SK96] Erich Schikuta and Peter Kirkovits. Analysis and evaluation of sorting for parallel database systems. In *In Proc. Euromicro 96, Workshop on Parallel and Distributed Processing, Braga, Portugal, IEEE Computer Society Press*, pages 258–265, January 1996.

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

Table 3. Heterogeneous Sample Sort (2MB of data, homogeneous configuration of perf vector)

Mean	SD	Mean/opt	Mean/SD	$\frac{Max}{Min}$
PID0				
3918862	584064	93,78%	14,90%	$\frac{5744959}{3045409}$
PID1				
4150519	625341	99,34%	15,06%	$\frac{6083675}{2942227}$
PID2				
3935862	579492	94,2%	14,72%	$\frac{5423733}{2755896}$
PID3				
4706443	505979	112,65%	10,75%	$\frac{5719919}{3699471}$

Table 4. Heterogeneous Sample Sort (16MB of data, homogeneous configuration of perf 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

Table 5. Heterogeneous PSOP (1973785 integers, heterogeneous configuration of perf 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, 983598

Table 6. Heterogeneous PSOP (16777215 integers, heterogeneous configuration of perf vector)

Criteria	**H-PSS**	**H-PSRS**	**H-PSOP**
Number of candidates	$6 * p' * log_2(p')$	$p' * (p-1)$	$4 * p' * p' * \log_2(p')$
Number of pivots	$P-1$	$P-1$	$4 * p' * \log_2(p') - 1$
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)	$\pm 15\%$ of the optimal and in the worst case and on one processor.	$\pm 0.1\%$ of the optimal value	$\pm 0.01\%$ of the optimal value
Number of files created / proc	$\max\{15, P\} + 1$	$\max\{15, P\} + 1$	$15 + 4 * p' * \log_2(p')$
Sensitivity to duplicates	?	No, until a bound of n/p' duplicates	?
Message sizes	32KB	32KB	32KB
Allocated memory	$8K * sizeof(int) + 6 * p' * \log_2(p')$	$8K * sizeof(int) + p' * (p-1)$	$8K * sizeof(int) + \mathcal{O}(p')$

Table 7. Summary of main properties of algorithms