An Out-of-Core Sorting Algorithm for Clusters with Processors at Different Speed

Christophe Cérin
Université de Picardie Jules Verne,
LaRIA, Bat CURI, 5 rue du moulin neuf, 80000 AMIENS - France
cerin@laria.u-picardie.fr

Abstract

The paper deals with the problem of parallel external integer sorting in the context of a class of heterogeneous clusters. We explore some techniques inherited from the homogeneous and in-core cases to show how they can be deployed for clusters with processor performances related by a multiplicative factor.
Keywords: *Out-of-Core Parallel Sorting Algorithms, Performance Evaluation, Parallel Integer Sorting Algorithms, Sorting by Regular Sampling, Data Distribution, Load Balancing Strategies.*

1. Introduction and motivations

Parallel in-core sorting records whose keys come from a linearly ordered set has been studied for many years. It is often said that 25 to 50 percent of all the work performed by computers is being accomplished by sorting algorithms [2]. For sorting (in parallel), memory references are not predictable and sufficient amount of communication are involved in a communication step. Thus sorting is a good candidate for benchmarking parallel machines.

The advent of parallel processing, in particular in the context of *cluster computing* is of particular interest with the available technology. 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. Alternatively, the aim is to sort when processors of the homogeneous cluster are loaded differently - but the initial loads stay constant during the experiment.

This class of machines is of particular interest for two kinds of customers: first, for those who cannot replace instantaneously the 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 also 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 with more than $k.n$ integers with the requirement that k should be as low as possible.

Parallel sorting algorithms under the framework of out-of-core computation is not new. Among the most valuable and recent publications we can cite [13], [6], [11], and since our work is based on sampling techniques, we shall also mention [7]. The objectives is 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 times.

The organization of the paper is along five sections not included this one. In section 2 we fix the additional vocabulary of external sorting and we recall the parallel disk model (PDM) that we use. In section 3 we review one in-core technique based on sampling for sorting in parallel and in the heterogeneous case for which we have obtained good results in the past. In section 4 we discuss step by step all the necessary ingredients that we have to combine to drive performance and we introduce our algorithm. Section 5 is about implementation issues and sketches experimental results. Section 6 concludes the paper.

2. The parallel disk model

From an algorithmic and model point of view, the papers of Vitter and al. [18, 19], [1] offer the best view, to our opinion of parallel disks systems. The I/O model measures the complexity of an algorithm not by the number of processing instruction but by the number of I/O operations required. Vitter captures the main properties of disk systems by the commonly used *parallel disk model* (PDM) through the following parameters (see Figure 1):

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. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items. It is convenient to refer to the following shortcuts:

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

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

Theorem 1 ([1]) *The average and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is:*

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

Note that in practice the $\log_m n$ term is a small constant.

3. Related work on parallel in-core sorting

In this section we recall some well known strategies for in-core sorting in parallel. We focus on a specific technique. It is now well understood that two generic approaches for in-core-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 algorithms, the different steps are summarized as follows: (1) each processor contains a portion of the list to be sorted (2) sort the portions and exchange them among all the processors (3) merge portions in one or many steps;

QUICKSORT-BASED: for this kind of algorithms, the different steps are summarized as follows: (1) the unsorted list is partitioned into a number of progressively smaller sublists defined by selected pivots (2) sort the sublists for which processors are responsible.

We consider only *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.

3.1. Regular sampling: an efficient technique for in-core sorting

"Sorting by regular sampling" (PSRS) [15, 16]. is an efficient technique for in-core sorting. It is based on the following ideas that refine the merge based approach we have cited previously: initially, each processor contains a portion of the list to be sorted. Then a 'merge based' approach is set up in order to sort the portions and exchange them among all the processors and merged. The merging of the sorted portions in a merge based approach can be achieved either in one step or many steps. We assume that in the remainder of the paper n denotes the input size and p the processor number.

Theoretically speaking, it can be shown that the computational cost of PSRS matches the optimal $\mathcal{O}(n/p \log n)$ bound. To obtain this result we must ensure that all the data are unique. In the case of no duplicates, PSRS *guarantees* to balance the work within a factor of two of optimal in theory, regardless of the value distribution. In practice we observe a few percent of optimal. The PSRS algorithm is as follows:

Step 1: Perform a local sort; each processor selects p samples which are gathered onto process zero;

Step 2: Perform a local sort of p^2 samples; pick $p-1$ regular pivots k from the sorted p^2 samples; (pivots are picked at $ip + p/2$, $(1 \leq i \leq (p-1)$ intervals); broadcast these pivot values to all the processors from processor zero.

Step 3: Each processor produces p partitions of its local block using the $p-1$ pivots; each processor sends its partition i (marked by pivots k_i and k_{i+1}) to processor i;

Step 4: Perform a merge of all the received partitions;

4. Our framework for out-of-core sorting

PSRS technique has been modified in [5] and [4] for in core sorting with processors running at different speeds. The *lowest common multiple* (lcm for short) is necessary to express some properties in a mathematical way and for shorter explanations. In another words, we require that the input size is as follows:

$$n = k * P[0] * \mathrm{lcm}(P, p) + \ldots + k * P[p-1] * \mathrm{lcm}(P, p) \tag{2}$$

where k is a constant in N, P is an array of size p of integers that denotes the relative performances of the p processors in the machine and $\mathrm{lcm}(P, p)$ is the least common multiple of the p integers stored in array P. For instance with

$k = 1, P = \{8, 5, 3, 1\}$ (we have one processor running 8 times faster than the slower processor, one processor running 5 times faster, one processor running 3 times faster in the cluster) we have that $\mathrm{lcm}(\{8, 5, 3, 1\}, 4) = 120$ and thus n could be equal to $120 + 3*120 + 5*120 + 8*120 = 2040$. Then, in the heterogeneous case, we will be able to assign easily to each processor an amount of data that will be inversely proportional to its speed. Note also that if the P buffer contains only one values, we get the homogeneous case. Our code is implemented in such a way that we have to modify only the declaration for the `P` array to experiment (simulate) with different situations.

Thus, we follow the four canonical phases of PSRS. The first step is a sequential sort. We implement it with a Polyphase Merge Sort [8]. Polyphase merging uses $2m$ files to get a $2m - 1$ way merge without a separate redistribution of runs after every pass. It thus has the advantages of an unbalanced sort without the disadvantage of redistribution. Polyphase merge sort is known to be efficient and matches the bound on sequential sorting. The whole algorithm is now depicted as follows:

Algorithm 1 (A PSRS scheme for external sorting)

Preconditions and initial configuration: *we deal with a cluster of $p = 2^q$ heterogeneous nodes. The heterogeneous notion is coded in the array 'P' of size p of integers that denotes the relative performances of the p nodes in the machine. The input size is n and n verifies Equation 2. Initially, disk (or processor) i has l_i, a portion of size $(n/(\Sigma_{i=1}^{p} P[i])) * P[i]$ of the unsorted list l.*

Step 1: (sequential sorting) *in using polyphase merge sort we get $2.l_i(1 + \lceil \log_m l_i \rceil)$ IO operations on each processor (since we have one disk attached per processor).*

Step 2: (selecting pivots) *a sample of candidates are randomly picked from the list. Each processor i picks $L = (p - 1) * P[i]$ candidates and passe them to a designated processor. This step requires L IO operations that is very inferior, in practice, to the IO operations of step 1. These candidates are sorted and then $p - 1$ pivots are selected by taking (in a 'regular way') $s^{th}, 2.s^{th}, \cdots, (p-1)^{th}$ candidates from the sample. Equipped with Equation 2, it can be checked that $s = k * \left\lfloor \dfrac{n * P[i] * lcm(P,p)}{p} \right\rfloor$. The selected pivots $d_1, d_2, \cdots, d_{p-1}$ are made available to all the processors; All these computations can be done in-core since the number of pivots is very small in practice (it is not an order of the internal memory size);*

Step 3: (partitioning) *since the pivots have been sorted (and fit in the main memory), each processor performs binary partitioning on its local portion. Processor i decomposes l_i according to the pivots. It produces p sublists (files) per processor. Since there are $Q = (n/(\Sigma_{i=1}^{p} P[i])) * P[i]$ data on processor i, there is no more than $2 * Q/B$ IOs per processor to accomplish the work in this step (we count read and write of data).*

Step 4: (redistribution of the sublists): *we form messages in order that the message sizes can fit in the local and distant memory. The size is also a multiple of the block size B. If we have an hardware which is able to transfer data from disk to disk, it will be more efficient. The number of IOs is no more than $2 * l_i/B$ (we also count read (sender side) and write (receiver side) of data).*

Step 5: (final merge) *each processor has in its local disk the p final portions (files) that can be merged with an external merged algorithm for mono-processor system. We re-use the* `mergeSort()` *procedure of the polymerge sort algorithm used in step 1. The number of IO operations is lower than $2.l_i(1 + \lceil \log_m l_i \rceil)$. Note that the constant 2 here stands for the bound of data given by the 'PSRS Theorem' that guarantee that in the last step of the algorithm, no processor has to deal with more than two times the initial amount of data. This theorem is still true for the heterogeneous case (see [5]) and we apply it.*

As we can see, the problem has been decomposed in such a way that is requires only basic blocks for the PDM with $D = 1$ and we use disks independently. The number of selected pivots is a modification of the result in [9] and it ensures that the load is well balanced across the heterogeneous processors. Concerning processor i with the amount of initial data of l_i, the optimal bound of $\Theta\left(\frac{l_i}{B} \log_m \frac{l_i}{B}\right)$ for parallel IO operations is also obtained with the algorithm. Remember that $l_i = n/p$ in the homogeneous case.

Let us give a sketch of the proof about the load balancing bound of 2. Our strategy to get the same bound than the PSRS algorithm is to respect the condition required by the PSRS... and then we apply the theorem. Basically, we have to pay attention to the number of pivots that we choose after the first initial sort. Since the input verifies Equation 2 we can pick pivots, in a regular way that is proportional both to the least common multiple of the performance and to the performance array itself. From a programming point of view, the code that each processor execute to select pivots is the following (the `blocksize` variable is the number of integers local to a processor - note that the value of `i` is the same on all processors due to Equation 2):

```
i=(int)(blocksize/(P[mypid]*nprocs))- 1;
off = i + 1; k = 0;
while (i <= (blocksize - off - 1)) {
   fseek(MYfpIN, (long) (i *
            sizeof(MPI_INT)), SEEK_SET);
   fread(&pivot[k], sizeof(MPI_INT), 1,
          MYfpIN);
   k++; i += off;
}
```

The consequence is that we pick a number of pivots that is proportional to the performance of each processor but we ensure that between any two consecutive pivots there is the same number of (sorted) elements. This is the main property of the PSRS algorithm that we have adapted here. Our framework is a generalization of the PSRS algorithm.

5. Implementation issues

Our implementation has been tuned with MPI whereas our previous codes were developed under the framework of BSP [17]. Our codes are freely available at `http://www.laria.u-picardie.fr/~cerin/=paladin`. Eight different benchmarks corresponding to eight different inputs are available. Since there is no data movement before sorting (we guess that we have a parallel file system with data distributed among nodes in a way proportional to the speeds) we measure only the effective sorting steps (steps 1 to 5 of Algorithm 1) and also a load-balancing factor.

Concerning the technical question about the way to fill the `P` array we decided to proceed according to the following protocol: for an input size of N integers on a $p > 1$ processors machine, we first execute the sequential external sort used in the parallel code on N/P data. We guessed that since the external sort performs both in and out operations and since with an homogeneous cluster each processor shall receive N/P data, external sorting is a good indicator of the relative performances. The ratio to the slower execution time allow us to fill the `P` array. We will see later, through experiments, the pertinence of the approach.

On Table 1 we have a synthetic view of our small cluster. The `/work` partition is used as the storage area and it is an SCSI disk drive.

On Table 2 we show the results for the sequential sort (*polyphase merge sort*) to fill the `P` array. Note that an input size of 33554432 integers corresponds to $33554432 * 4 = 134217728$ bytes, that is to say $134Mb$. We conclude that helmvige and grimgerde are 4 times faster than siegrune and rossweisse. This conclusion contradicts what we have on Table 1: in fact we have forked processes on siegrune and rossweisse to obtain loaded processors. We decide to configure our `P` vector with $\{1, 1, 4, 4\}$ values and we keep, in the remainder of our experiments, our initial loads!

With a performance array containing only 1 values (we configure our machine as an homogeneous cluster) we are able to find (with fast-Ethernet) experimental results that are worst than the sequential execution. For instance, with packet size of 8 integers, we need 133.61 seconds to sort 2097152 (see also Table 2). But, with message size of $8K$ integers we sort in $32.6s$ the 2097152 integers. It seems that $8K$ gives the best time performance.

On Table 3 we have the experimental results for sorting with disks $2^{24} = 16777216$ integers on our 4 processors cluster. We set the performance vector either with $\{1, 1, 1, 1\}$ or with $\{1, 1, 4, 4\}$. We have also mixed two communication libraries: Fast-Ethernet and Myrinet.

The six columns of Table 3 correspond to (from left to right): problem size (number of integers), mean execution time in seconds, standard deviation, mean of the size of partitions in the last step of the algorithm (the optimal for the case where the performance array contains only one values is: 4194304), maximal partition size, the sublist expansion metric (the ratio between column 5 and value 4194304 in the case of performance buffer filled with 1 values).

In any case, we note that the sublist expansion metric is very close to 1. So, we master the load very efficiently. However, regarding to the experiment with the performance buffer filled with 1 values, we find that the execution time is greater than the sequential execution time (235s) to sort the same amount of data on Helmvige. It is better than the sequential execution time on Siegrune (909s). In this case, the gain with four processors is 3.

Since the least common multiple of $\{1, 1, 4, 4\}$ is 4, we are able to choose the size of 16777220 as the problem size for the two last lines of Table 3. These two lines correspond to an experiment with fast-Ethernet and the other one with Myrinet. The optimal size on the two slowest processors (Siegrune and Rossweisse) is 1677722 whereas the optimal size on the two fastest processors (Helmvige and Grimgerde) is 6710888 integers. On Table 3, the column entitled 'Mean' of the heterogeneous cases gives the mean sizes handled on the two fastest processors and the column entitle S(Max) gives the sublist expansion metric for the two fastest processors. We also note a value close to the optimal which is once again 1.

Comparing to the most favorable sequential execution time (212s) we have a gain of 1.37; and comparing to the most unfavorable sequential execution time (951s) we have a gain of 6.13... on a 4 processors machine. The results on Table 3 validate our approaches since we note an improvement of the execution time comparing to the 'homogeneous' configuration.

At least, we note that the execution times with Myrinet do not improve performance. The explanations are the following: first the application does not communicate so much (when data move, they go to the right place) and second, the execution time does not comprise neither the initial distribution of data (since they are generated on a sole node) nor the gather time. The results was expected and confirm that our sort performs well when the communication layer is not 'the best we can use'.

6. Conclusion

In this paper we have introduced our approach to tackle the problem of external sorting on non homogeneous clusters. The algorithm that we describe combines very good properties for load balancing. The algorithm is an adaptation of a known technique called PRSR (Parallel Sorting by Regular Sampling).

To our knowledge, little work has been performed about external sorting on non homogeneous network. If we try to reuse the main results [12, 10, 16, 9] about sorting on homogeneous clusters as a general strategy, the main difficulty is to show that it will lead to good properties for load balancing, for execution time and also that the underlying algorithm is suitable for a concrete implementation. We are not yet convinced that other technique than the sampling technique will have so much properties. It is still challenging to explore in deep quicksort based approaches and randomized parallel sorting algorithms (external and not) in the context of non homogeneous clusters. We think that our approach is promising in many ways.

Today, accessing a disk is an order of magnitude greater than accessing RAM. Thus, IO intensive algorithms are designed to minimize the number of times we access disks. If we guess first that in a very near future the gap between accessing main memory and accessing cache stay identical to nowadays performance and second that disks will be accessed much faster than today by the use of an appropriate technology (InfiniBand or RapidIO or bigger caches) making a memory access time 'comparable' to a disk access time then the design of cache conscious algorithms will play a central role. For the case of sorting, the papers [14] and [3] tackle the problems. However, no one of the two papers consider the case of heterogeneous platforms, in particular heterogeneous caches.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[2] S. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[3] L. Arge, J. Chase, J. S. Vitter, and R. Wickremesinghe. Efficient sorting using registers and caches. In *Proceedings of the Workshop on Algorithm Engineering (WAE'00, to be published by Springer Verlag as LNCS (not yet available)*, 2000. See: http://www.cs.duke.edu/˜large/ and http://www.mpi-sb.mpg.de/˜conf2000/wae2000/.

[4] C. Cérin and J.-L. Gaudiot. An over-partitioning scheme for parallel sorting on clusters running at different speeds. In *Cluster 2000. IEEE International Conference on Cluster Computing. Technische Universitt Chemnitz, Saxony, Germany. (Poster)*, 28Nov.-2Dec. 2000.

[5] C. Cérin and J.-L. Gaudiot. Parallel sorting algorithms with sampling techniques on clusters with processors running at different speeds. In *HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India*, Lecture Notes in Computer Science. Springer-Verlag, 17-20Dec. 2000.

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

[7] D. J. DeWitt, J. F. Naughton, and D. 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, Dec. 1991.

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

[9] H. Li and K. 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.

[10] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19:1079–1103, Oct. 1993.

[11] M. 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.

[12] M. Quinn. Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

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

[14] A. Ranade, S. Kothari, and R. Udupa. Register efficient mergesorting. In *HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India*, Lecture Notes in Computer Science. Springer-Verlag, 96-103Dec. 2000.

[15] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 10–16, Boston, Massachusetts, 25–27 Apr. 1983.

[16] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[17] L. Valiant. A bridging model for parallel computation. *Communiations of the ACM*, 33(8):103–111, Aug. 1990.

[18] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, Aug. and Sept. 1994.

[19] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, Aug. and Sept. 1994.

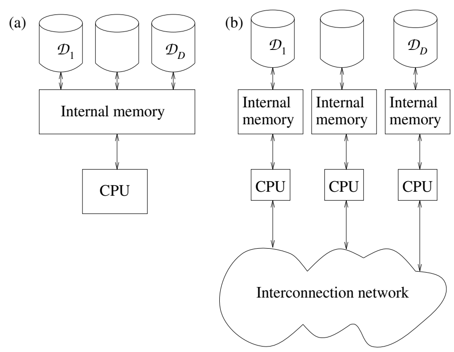

Figure 1. The picture comes from [18]. Parallel disk model: (a) $P = 1$, in which the D disks are connected to a common CPU; (b) $P = D$, in which each of the D disks is connected to a separate processor. This last organization is realistic for a cluster system.

Table 1: Configuration: 4 Alpha 21164 EV 56, 533Mhz - Fast Ethernet

Node	Cache L3/L2/L1	Disk	Kernel	`/work` size
helmvige	4Mo/96Ko/8Ko	4Go SCSI	Linux 2.2.13-0.9	1Go
grimgerde	4Mo/96Ko/8Ko	4Go SCSI	Linux 2.2.13-0.9	4Go
siegrune	4Mo/96Ko/8Ko	4Go SCSI	Linux 2.2.5-16	4Go
rossweisse	2Mo/94Ko/8Ko	8Go SCSI	Linux 2.2.5-16	4Go

Table 2: External sorting on architecture depicted on Table 1

Input size	Exe. Time (s)	Deviation	Input size	Exe. Time (s)	Deviation
Helmvige					Rossweisse
2097152	22.92146	0.45283	2097152	95.40269	1.09854
4194304	51.17832	1.99283	4194304	204.66360	4.59815
8388608	111.40898	1.48268	8388608	428.42470	3.35943
16777216	235.74163	2.67709	16777216	951.22738	77.4042
33554432	492.02380	9.74561	33554432	1998.72261	152.8972
Siegrune					Grimgerde
2097152	88.94593	1.85451	2097152	24.88658	10.20334
4194304	188.71978	3.41997	4194304	44.55758	0.86754
8388608	409.09711	36.13593	8388608	96.29102	1.46595
16777216	909.34783	81.67	16777216	212.82059	2.54191
33554432	1910.8261	160.60827	33554432	443.86681	10.12

Execution time metrics for benchmark 0
Algorithm: polyphase merge sort

Table 3: External Sorting on Cluster (see. Table 1), message size : 32Kb, 15 intermediate files, 30 experiments

Input Size	Exe Time (s)	Deviation	Mean	Max	S(max)
Performance : $\{1, 1, 1, 1\}$; Fast-Ethernet					
16777216	303.94	9.173	4193043.8	4204494	1.00273
Performance : $\{1, 1, 4, 4\}$; Fast-Ethernet					
16777220	155.41	3.645	6816502.4	7342910	1.094
Performance : $\{1, 1, 4, 4\}$; Myrinet					
16777220	155.43	3.465	6293368.5	7341545	1.093

Execution time metrics for benchmark 0
Algorithm: external PSRS