Improving Parallel Execution Time of Sorting on Heterogeneous Clusters

Christophe Cérin
Université de Picardie Jules Verne
LaRIA, Bat Curi, 5 rue du moulin neuf
F-80039 Amiens cedex 1- France
cerin@laria.u-picardie.fr

Michel Koskas
Université de Picardie Jules Verne
LaMFA/CNRS UMR 6140, 33 rue St Leu
F-80039 Amiens cedex 1- France
koskas@laria.u-picardie.fr

Hazem Fkaier, Mohamed Jemni
École supérieure des Sciences et Techniques de Tunis,
Unité de recherche UTIC
5, avenue Taha Hussei, B.P. 56 Bab Menara, Tunis - Tunisie
hazem.fkaier@fst.rnu.tn,
mohamed.jemni@fst.rnu.tn

March 17, 2004

Abstract

The aim of the paper is to introduce techniques in order to optimize the parallel execution time of sorting on heterogeneous (processors speeds are related by a constant factor) platforms. We developp a constant time technique for mastering processor load balancing and execution time in an heterogenous environment. In order to illustratre the difficulties and solutions, we develop an analytical model for the execution time, sustained by experimental results in the case of a 2-processors systems. The solution is a constant one, independent of the problem size and is not based on dynamic programming technique. Consequently, there is no overhead regarding the sorting problem.
Keywords: in-core parallel sorting algorithms, heterogenous computing, complexity of parallel algorithms.

1 Introduction

Sorting is a fundamental problem in computer science and many works have been published in the past. 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. Applications of sorting are so numerous that it will be fastidious to list all of them. We shall say that sorting appears in the convex hull problem; the join operation in database systems can be implemented with a sort, and so on.

Moreover, the quasi non predictable aspects of memory references in sorting algorithms makes them good candidates to appreciate the performance of processors in real situations.

The advent of parallel processing, in particular in the context of *cluster computing*[1] 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 whole 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 depict a cluster by the mean of a vector (the so call *performance vector*) set by the relative speeds of each processor.

In this paper we develop techniques in order of mastering the execution time and the load balancing of each node. The paper organization is the following. In Section 2 we summarize our previous results. In Section 3 we introduce the new problems. Section 4 exposes the technique and solution for the computation of optimal sizes for data redistribution and for the case of 2 processors. Section 5 generalizes the results exposed previously. Section 6 is devoted to experimental results and Section 7 concludes the paper.

2 Our previous results

Our previous papers [CG00a, CG00b, CG00c, CG02, Cér02] deal with internal and external sorting algorithms on this particular class of clusters and they are innovative since all the papers (to our knowledge) about external parallel sorting algorithms for instance, that work (we mean "implemented") always consider the special case of homogeneous computing platforms. For instance, the sorting algorithm implemented for NAS parallel benchmark considers the homogeneous case.

In these papers [CG00a, CG00b, CG00c, CG02, Cér02] we focused 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 $k.n$ data with the requirement that k should be as low as possible.

The main difference with our previous works is that we are now interested by the best way to guaranty both load balancing properties and the best execution time. We will explain that load balancing does not necessary implies the best execution time when we deal with heterogeneous clusters.

We do not explain here in details the properties that we have used in our past results but we do prefer to expose the overall strategy. We focus on a specific technique. The technique, namely a meta schema for partitioning elements located on different nodes, is revisited in this paper to match an upper bound on the execution time.

2.1 Generic approaches

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

[1] See the IEEE task force on cluster computing on `http://www.ieeetfcc.org`

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.

Only a MERGE-BASED algorithm is under concern in papers above mentioned, 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 limited number of long messages of message passing programming languages in order to get performances. 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 'to be slowdown' by the bandwidth of the network.

To summarize, *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 don't care about it: it is difficult to derive a bound to partition data into sublists of 'equal sizes'.

Let us explain the spirit of such algorithm in the case of homogeneous clusters.

2.2 Regular sampling: an efficient technique for in-core sorting in parallel

"Sorting by regular sampling" (PSRS) [RV87, RV83, SS92, LLS+93, HJB96]. 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 merge them. The merging of the sorted portions in a merge based approach can be achieved either in one step or many steps. References [Qui89, LLS+93, SS92, LS94] belong to the category of *one step merge based algorithms*; references [Bat68, BR93, BH82, Col88, Lei84, NS82, Pla89, TB93] belong to *multi steps merge-based algorithms.*

We assume that in the remainder of the paper N denotes the input size and p the processor number. The implementation of PSRS, in the homogeneous case, is as follows:

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

In order to observe why the algorithm is correct we shall note that first, in sampling the locally sorted chunks of all the processors and not just a subset, the entire data is represented and second that in sampling after the first local sort, the order information of the data is captured.

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.

Note also that, because it will be our main disucusion later in the paper, the initial condition for this sort is that each processor has n/p data locally before starting any operation and that we preserve this amount of data on each processor during the different steps. Theses facts are still valid in the heterogeneous case that we do not introduce here in order to restrict the page number to the desired value, but the reader may consult [Cér02] for instance.

We introduce now yet another technique that we have developped in [CG00b] for sorting on heterogeneous clusters.

2.3 Parallel Sorting by Over-partitioning (PSOP)

2.3.1 The homogeneous case

Other approaches than PSRS to load balance the work in sorting algorithms such as the Li and Sevcik algorithm [LS94] can potentially handle nodes that do not make uniform progress. The key ideas of Li and Sevcik [LS94] for the homogeneous case is to replace the initial sorting step that is used for the selection of pivots that partition the input into equal size chuncks by a 'sufficient number' of random pivots that also have to partition the input into chunks of approximatively equal sizes.

The key technical discussion is about the number of pivots. The trick used in [LS94] is related to the following result:

Theorem 1 (See [LS94]) $(p.k-1)$ *pivots partition the input into* $p.k$ *chuncks such that the size of the greatest chunck is lower ou equal to* n/p *with probability at least*

$$1 - 2p\left(1 - \frac{1}{2p}\right)^{p.k}$$

The other canonical steps after this initial step of the PSRS technique is identical in the Li and Sevcik algorithm. Here it is:

Algorithm 1 (Code as it is found in [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,*[2] *where* s *is the oversampling ratio and* k *the over-partitioning ratio. Each processor picks* $s.k$ *candidates and passe them to a designated processor. These candidates are sorted and then* $p.k-1$ *pivots are selected by taking (in a 'regular way')* $s^{th}, 2.s^{th}, \cdots, (pk-1)^{th}$ *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_{ij} *where* i, j *stands for two consecutive pivots (except for*

[2] Note here that we bypass the initial sort of the PSRS technique in order to improve (intuitively speaking) performances

the initial an final case). A sublist S_j *is the union of* l_{ij} *with* i *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:*

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

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 sublists size to the 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* $\sigma_j + | S_j | -1$*, and (b) applying a sequential sort to the elements in that range. The process continues until the task queue is empty.*

2.3.2 The heterogeneous case

We have introduced in [CG00b] and adaptation of the previous algorithm for an heterogeneous platform. The key for load balancing in this algorithm is the tasks scheduling step (step 4). The challenge is to schedule tasks with different sizes on processors having different speeds. The number of pivots required to ensure a good load balancing factor is computed according to theorem 1 but in considering that we have virtually a number of processor equal to the sum of the performance vector instead of p processors.

Concerning the scheduling step, we propose now the following strategy for implementing step 4:

- As with the original version, we compute the sizes of the tasks and then we sort them according to their sizes.
- We proceed task by task, in the decreasing order.
- We estimate the execution time of the current task on each processor as follows. Let TS_i be the size of the ith task, then the execution time on the jth processor will be: Exec_time(current task) $= (TS_i \log TS_i)/perf_j$.

  We compute, for each processor, the execution time of the tasks already scheduled on it plus the execution time of the current task: Let S_j be the sum of the execution times of the tasks aldeary scheduled to the jth processor: $S_j = S_j +$ Exec_time(current task)
- We compare the new values of S_j relative to the different processors, and we detain the lowest value. The task will be definitely scheduled on to the processor having the lowest value of S_j and the others will ignore this task. And we pass to the next task.

The strategy we presented here is well known in production management for instance. In fact, the "Longest Processing Time" algorithm is used in the case of scheduling independent tasks having different sizes to machines having different speeds. The goal is to minimize the global execution time (*makespan*) by using all the allowed resources and beginning in the same time.

Note that in [CG00b], the task scheduling is not made according to the execution time of sorting n_i data but according to the n_i sizes.

We made an estimation of the execution time based on the time complexity of the sorting algorithm. But, and this point will be the main discusion point in the next section, the partitioning step considers portions of size $n/perf_i$, where $perf_i$ is the speed of processor i.

3 Discussion about the parallel execution time

We have seen that in the case of heterogeneous platforms, data are initially distributed proportionally to the speed of processors. This is the precondition of the problem. We guaranty the load balancing of work on each processor either by the mean of PSRS or PSOP techniques.

We now examine the impact of the initial distribution or, more precisely the impact of the redistribution of data, on the parallel execution time. We determine the impact in terms of the way of restructuring the code of the meta partitioning scheme that we have introduced above. Then we justify the approach rather than processing to a redistribution of data when we start a new execution.

In previous section, when we had N data to sort on p processors depicted by their respective speeds $k1, \cdots, k_p$, we had needed to distribute to processor p_i an amount n_i of data such that:

$$n_1/k_1 = n_2/k_2 = ... = n_p/k_p \tag{1}$$

and

$$n_1 + n_2 + ... + n_p = N \tag{2}$$

The solution is:

$$\begin{array}{rcl} n_1 & = & N * k_1/(k_1 + k_2 + ... + k_p) \\ n_2 & = & N * k_2/(k_1 + k_2 + ... + k_p) \\ \cdots & = & ..... \\ n_p & = & N * k_p/(k_1 + k_2 + ... + k_p) \end{array}$$

Now, since the sequential sorts are executed on n_i data at $n_i \log n_i$ time cost (approximatively since there is a constant in front of this term), there is no reason that the nodes terminate at the same time since $n_1/k_1 \log n_1 \neq n_2/k_2 \log n_2 \neq \cdots \neq n_p/k_p \log n_p$ in this case.

The main idea that we develop now is to distribute to each processor an amount of data proportional to $n_i \log n_i$ in order to be processed by the sequential sorts. The problem is now to compute the new data sizes $n'_1, \cdots, n'_p$ such that:

$$n'_1 + n'_2 + \cdots + n'_p = N \tag{3}$$

and such that

$$(n'_1/k_1) \log n'_1 = (n'_2/k_2) \log n'_2 = \cdots = (n'_p/k_p) \log n'_p \tag{4}$$

First of all, we show that this new distribution converges to the initial distribution when N tend to infinity. The fundamental reason is that $\lim_{n\to\infty} \frac{n}{\log n} = \lim_{n\to\infty} (n)$.

Let us simply consider the case of 2 processors at speed k_1, k_2 to get the intuition that the distributions converge. The first (when we consider the initial method) distribution is:

$$\begin{array}{rcl} n_1 & = & N * k1/(k1 + k2) \\ n_2 & = & N * k2/(k1 + k2) \end{array}$$

and consequently $(k1/k2) = (n_1/n_2)$.

The new distribution is:

$$\begin{array}{rcl} n'_1 + n'_2 & = & N \\ (n'_1/k_1)\log n'_1 & = & (n'_2/k_2)\log n'_2 \end{array}$$

and consequently $(k_1/k_2) = (n'_1 \log n'_1)/(n'_2 \log n'_2) = (n'_1/n'_2) * (\log n'_1/\log n'_2)$

If N increases, then n_1 and n_2 will also increase. And more they increase, more the ratio of log terms tends to 1. Consequently, the ratio n'_1/n'_2 tends to k_1/k_2 which is equal to n_1/n_2. Consequently, more N is high, more the execution times of sorts will be close each together.

Second, let us consider some numerical examples to see if some practical cases could be solved efficiently by our new method despite the fact that asymptotically speaking it is equivalent to the previous one.

Let P_1, P_2 two processors caracterized by their speeds $k_1 = 1, k_2 = 6$ respectively and let $N = 321$. The first method gives $n_1 = 46$ and $n_2 = 275$ whereas the second method gives (approximatively):

$$\begin{array}{rcll} n'_1 & = & 64 & ((64\log 64)/1 = 64 * 6 = 384) \\ n'_2 & = & 257 & ((257\log 257)/6 = (257 * 9)/6 = 385,5) \end{array}$$

The ratio $k_1/k_2 = 0,1666$ whereas the ratio $n'_1/n'_2 = 0,249$. The unbalance is $1 - 0.1666/0.249 = 37\%\ldots$ which is important.

Now, if we consider méga bytes, let $N = 408M$. The first distribution is $n_1 = 58M$ and $n_2 = 350M$. The second distribution is:

$$\begin{array}{rcll} n'_1 & = & 64M & ((64M\log 64M)/1 = 64M * 26 = 1744830464) \\ n'_2 & = & 344M & ((344M\log 344M)/6 = (344M * 29)/6 = 1743432362) \end{array}$$

The unbalance factor is now $1 - 0.166/0.186 = 11\%$.

Finally, let us consider the case of Giga bytes and let $N = 419G$. The first distribution gives $n_1 = 60G$ and $n_2 = 359G$. The second distributions gives:

$$\begin{array}{rcll} n'_1 & = & 64G & ((64G\log 64G)/1 = 64G * 36 = 2.473.901.162.496) \\ n'_2 & = & 355G & ((355G\log 355G)/6 = (355G * 39)/6 = 2.477.659.258.880) \end{array}$$

The n'_1/n'_2 ratio is $0,180$, thus the unbalance factor is now 8%.

We are now convinced that when N tends to infinity, the n'_1/n'_2 ratio tends to $1/6$, that is to say to n_1/n_2. Moreover, we think that we have many situations that could benefit from our optimization.

4 Computation of the optimal sizes (2 processor case)

We are going to compute the n'_i sizes by approximating the solution through a Taylor development for the log function. Note that with our initial method we have $n_i = N\dfrac{k_i}{(k_1 + \cdots + k_m)}$.

Let us find n'_i under the form $n'_i = N\dfrac{k_i}{k_1 + \ldots + k_m} + a_i\dfrac{N}{\log N}$.

The idea is to produce a new term of form $a_i\frac{N}{\log N}$ complementing the previous distribution.

In fact, we will now compute the a_i terms uniquely for the case of two processors. Moreover, it is clear that the sum of a_i terms is null since the sum of n_i terms is invariant and is equal to N.

Theorem 2 *The approximated sizes to distribute to each processor of a two processors cluster of speed* $k_1, k_2 \in \mathbb{N}$ *and for a problem size of* n *data are:*

$$n_1 = n\frac{k_1}{k_1+k_2} + \frac{n}{\log n}\log\left(\frac{k_2}{k_1}\right)\frac{k_1k_2}{(k_1+k_2)^2}$$

and

$$n_2 = n\frac{k_2}{k_1+k_2} + \frac{n}{\log n}\log\left(\frac{k_1}{k_2}\right)\frac{k_1k_2}{(k_1+k_2)^2}.$$

Proof: We need some shortcuts. Let

- $\alpha_1 = k_1/(k_1+k_2)$
- $\alpha_2 = k_2/(k_1+k_2)$
- $\alpha = k_1k_2/(k_1+k_2)$
- a_1 and a_2 be the two unknown terms

Let us recall the following Taylor developments:

- $\text{Taylor}(\log(1-x)) = -x - 1/2 * x^2 - 1/3 * x^3 - 1/4 * x^4 - 1/5 * x^5 - 1/6 * x^6 + \mathcal{O}(x^7)$
- $\text{Taylor}(\log(1+x)) = x - 1/2 * x^2 + 1/3 * x^3 - 1/4 * x^4 + 1/5 * x^5 - 1/6 * x^6 + O(x^7)$

We proceed by equivalence starting from the fact expressing that the execution times are identical:

$$\begin{aligned}
&(n_1\log n_1)/k_1 = (n_2\log n_2)/k_2 \iff \\
&((\alpha_1 n + a_1)\log(\alpha_1 n + a_1))/k_1 - ((\alpha_2 n + a_2)\log(\alpha_2 n + a_2))/k_2 = 0 \iff \\
&k_2(\alpha_1 n + a_1)\log[\alpha_1 n(1 + a_1/(\alpha_1 n)] - k_1(\alpha_2 n - a_1)\log[\alpha_2 n(1 - a_1/(\alpha_2 n)] = 0 \iff \\
&k_2\alpha_1 n\log(\alpha_1 n) + k_2 a_1\log(\alpha_1 n) + k_2\alpha_1 n\log(1 + a_1/(\alpha_1 n)) + \\
&k_2 a_1\log(1 + a_1/\alpha_1 n) - \\
&\text{(this term is close to 0 when we take the Taylor development, so we neglish it)} \\
&k_1\alpha_2 n\log(\alpha_2 n) + k_1 a_1\log(\alpha_2 n) - k_1\alpha_2 n\log(1 - a_1/(\alpha_2 n)) + k_1 a_1\log(1 - a_1/(\alpha_2 n)) \\
&\text{(this term is close to 0 when we take the Taylor development, so we neglish it)} \iff \\
&\alpha n\log(\alpha_1 n/(\alpha_2 n)) + \alpha\log((1 + a_1/(\alpha_1 n))/(1 - a_1/(\alpha_2 n))) + \\
&k_2 a_1\log(\alpha_1 n) + k_1 a_1\log(\alpha_2 n) = 0 \iff \\
&\text{(We use Taylor development here for logs)} \\
&\alpha n\log(k_1/k_2) + \alpha n(a_1/(\alpha_1 n) + a_1/(\alpha_2 n)) + k_2 a_1\log(\alpha_1 n) + k_1 a_1\log(\alpha_2 n) = 0 \iff \\
&\alpha n\log(k_1/k_2) + \alpha n a_1/n((k_1+k_2)^2/(k_1.k_2)) + a_1(k_1\log(\alpha_1 n) + k_1\log(\alpha_2 n)) \iff \\
&\alpha n\log(k_1/k_2) + a_1[(k_1+k_2) + k_2\log(\alpha_1 n) + k_1\log(\alpha_2 n)] = 0 \iff \\
&a_1 = \frac{k_1k_2/(k_1+k_2)n\log(k_2/k_1)}{(k_1+k_2) + k_2\log(\alpha_1 n) + k_1\log(\alpha_2 n)}
\end{aligned} \tag{5}$$

To observe the result, we have now to show that the denominator is close to $(k_1+k_2)\log(n)$ which is not very difficult to obtain. In this case, we obtain the aforementioned result:

$$a_1 = \frac{k_1 k_2}{(k_1+k_2)^2} \log(k_2/k_1) \frac{n}{\log n} \tag{6}$$

A similar computation leads to the value of n_2.

4.1 Summarize

To summarize, we propose to use the following values for n_1, n_2:

$$n_1 = n\frac{k_1}{k_1+k_2} + \frac{n}{\log n} \log\left(\frac{k_2}{k_1}\right) \frac{k_1 k_2}{(k_1+k_2)^2}$$

and

$$n_2 = n\frac{k_2}{k_1+k_2} + \frac{n}{\log n} \log\left(\frac{k_1}{k_2}\right) \frac{k_1 k_2}{(k_1+k_2)^2}.$$

4.2 Examples

To illustrate our choices, let's go to apply our formula for $k_1 = 1, k_2 = 6$ and for 3 different values differentes for n. The 5 columns of each array correspond to n_1, n_2 following to 3 metrics to estimate the bias. The fifth column correspond to the error comparing to the optimal solution, here $1/6$.

The second line corresponds to a compution of n_1, n_2 separately. The third line corresponds to the computation of n_1, then $n_2 = n - n_1$ and the fourth line correspond to the computation of n_2 first, then $n_1 = n - n_2$.

n= 4190.0

n_1	n_2	$(n_2+n_1-n)/n$	$1/6 \Rightarrow n_1/n_2$	$(n_2+n_1-n)/n - 1.0/6.0$
708.79	3583.02	0.02429984	$1/6 \Rightarrow 0.1978191$	-0.1423668
708.79	3481.20	0.0	$1/6 \Rightarrow 0.2036048$	0.03693820
606.97	3583.02	0.0	$1/6 \Rightarrow 0.1694028$	0.00273619

n= 419000.0

n_1	n_2	$(n_2+n_1-n)/n$	$1/6 \Rightarrow n_1/n_2$	$(n_2+n_1-n)/n - 1.0/6.0$
66958.24	359585.19	0.0180034	$1/6 \Rightarrow 0.1862096$	-0.14866
66958.24	352041.75	0.0	$1/6 \Rightarrow 0.1901997$	0.023533
59414.80	359585.19	0.0	$1/6 \Rightarrow 0.1652315$	-0.001435

n= 419000000.0

n_1	n_2	$(n_2+n_1-n)/n$	$1/6 \Rightarrow n_1/n_2$	$(n_2+n_1-n)/n - 1.0/6.0$
64487499.19	359441386.024	0.0117634	$1/6 \Rightarrow 0.17941$	-0.1549
64487499.19	354512500.806	0.0	$1/6 \Rightarrow 0.18190$	0.015238
59558613.9757	359441386.024	0.0	$1/6 \Rightarrow 0.16569$	-0.0009689

We note that in choosing first to compute n_2 then to set $n_1 = n - n_2$ we get the minimal bias and even we get n_1/n_2 ratios lower than 1% of the optimal $1/6$ value. In any cases, the approximations are good.

5 General case

In this section we compute the optimal sizes in the general of any number of processors. We reuse the previous assumptions and techniques. Let us assume that K is the sum of the relative speeds k_i of a p processor system. The problem is depicted by the following 3 equations:

$$n_i = \frac{k_i}{K}N + \epsilon_i, \quad (1 \leq i \leq p) \tag{7}$$

$$\sum_{i=1}^{p} \epsilon_i = 0 \tag{8}$$

$$\frac{n_i \log n_i}{k_i} = \frac{n_j \log n_j}{k_j}, \quad (1 \leq i, j \leq p) \tag{9}$$

Note again that Equation 8 says that the sum of the correcting factor should be null because the sum of the n_i is invariant and is equal to N. We develop Equation 9 and we reuse the same facts during the approximation that we have done in the previous section. In another words, we proceeed again by equivalence.

$$\begin{aligned}
& k_j \left(\frac{k_i}{K}N + \epsilon_i\right) \log\left(\frac{k_i}{K}N + \epsilon_i\right) = k_j \left(\frac{k_j}{K}N + \epsilon_j\right) \log\left(\frac{k_j}{K}N + \epsilon_j\right) \\
& \iff \\
& \frac{k_i k_j}{K}N \left[\log\left(\frac{k_i}{K}N + \epsilon_i\right) - \log\left(\frac{k_j}{K} + \epsilon_j\right)\right] + k_j \epsilon_i \log\left(\frac{k_i}{K}N + \epsilon_i\right) - k_i \epsilon_j \log\left(\frac{k_j}{K}N + \epsilon_j\right) = 0 \\
& \iff \\
& \frac{k_i k_j}{K}N \left[\log\left(\frac{k_i}{k_j}\right) + \frac{\epsilon_i K}{k_i N} - \frac{\epsilon_j K}{k_j N}\right] + k_j \epsilon_i \log\left(\frac{k_i}{K}N\right) - k_i \epsilon_j \log\left(\frac{k_j}{K}N\right) = 0 \\
& \iff \\
& \frac{k_i k_j}{K}N \log\left(\frac{k_i}{k_j}\right) + \epsilon_i \left[k_j + k_j \log\left(\frac{k_i}{K}N\right)\right] - \epsilon_j \left[k_i + k_i \log\left(\frac{k_j}{K}N\right)\right] = 0 \\
& \iff \\
& \epsilon_j = \frac{\frac{k_i k_j}{K}N \log\left(\frac{k_i}{k_j}\right) + \epsilon_i \left(k_j + k_j \log\left(\frac{k_i}{K}N\right)\right)}{k_i \left(1 + \log\left(\frac{k_j}{K}N\right)\right)}
\end{aligned} \tag{10}$$

Now, we have also that:

$$\epsilon_1 + \sum_{j=2}^{p} \epsilon_j = 0$$

Thus,

$$\epsilon_1 \left[\sum_{j=2}^{p} \frac{k_j \left(1 + \log\left(\frac{k_1}{K} N\right)\right)}{k_1 \left(1 + \log\left(\frac{k_j}{K} N\right)\right)} \right] + \frac{k_1}{K k_1} N \sum_{j=2}^{p} \frac{k_j \log\left(\frac{k_1}{k_j}\right)}{1 + \log\left(\frac{k_j}{K} N\right)} = 0$$

$$\Longleftrightarrow$$

$$\epsilon_1 \left[1 + \frac{1 + \log\left(\frac{k_1}{K} N\right)}{k_1} \sum_{j=2}^{p} \frac{k_j}{1 + \log\left(\frac{k_j}{K} N\right)} \right] + \frac{N}{K} \sum_{j=2}^{p} \frac{k_j \log\left(\frac{k_1}{k_j}\right)}{1 + \log\left(\frac{k_j}{K} N\right)} = 0 \tag{11}$$

$$\Longleftrightarrow$$

$$\epsilon_1 \sum_{j=1}^{p} \frac{k_j}{1 + \log\left(\frac{k_j}{K} N\right)} + \frac{k_1 N}{K \left(1 + \log(\frac{k_1}{K} N)\right)} \sum_{j=1}^{p} \frac{k_j \log\left(\frac{k_1}{k_j}\right)}{1 + \log\left(\frac{k_j}{K} N\right)} = 0$$

Now we use the facts that $1 + \log(k_1/KN) \sim \log N$, $1 + \log(k_j/KN) \sim \log N$ and

$$\sum_{j=1}^{p} \frac{k_j}{1 + \log\left(\frac{k_j}{K} N\right)} \sim K / \log N$$

to derive that

$$\epsilon_1 = \frac{k_1 N}{K^2 \log N} \sum_{j=1}^{p} k_j \log\left(\frac{k_j}{k_1}\right)$$

Hence the formula:

$$\epsilon_i = \frac{N}{\log N} \left[\frac{k_i}{K^2} \sum_{j=1}^{p} k_j \log\left(\frac{k_j}{k_i}\right) \right] \tag{12}$$

Verification: for the 2 processors case we get the previous result $\epsilon_1 = \frac{N}{\log N} \log\left(\frac{k_2}{k_1}\right) \frac{k_1 k_2}{(k_1 + k_2)^2}$ as expected.

6 Experimental results

In order to get trends from our new partitioning schema, we have modified one of our code [CJF03] and we have experimented on a 2 processor system (2 Alpha 21164 EV 56 processors[3] at 533Mhz interconected with Fast Ethernet under MPI).

The code is an implementation of Parallel Sample Sort (PSS) for heterogeneous platforms. It is an out-of-core implementation because it uses only 8K of RAM (for any problem size) and Polyphase merge sort as our sequential sorting algorithm.

We set arbitrary the performance vector to $\{1, 3\}$ (processor 1 is 3 times faster than processor 0) and we measure the sizes of data on each node during the last step (the sequential sorting step) for our new

[3]The system is homogeneous but we measure the data size on each node and not the execution time

approach and for the previous one. We execute our codes for an input size of 1048576 integers with an oversampling ratio of 6 (see [CJF03] for an explanation of the role of this coefficient).

According to our new schema, we measure a deviation of $+6.7\%$ and -2.6% comparing to the optimal sizes (given as $262144 + 15580 = 277725$ integers and 770851 integers) and for processor 0 and 1 reciprocally. These results corresponds to means.

According to our previous schema, we measure a deviation of -13.4% and -1.5% comparing to the optimal sizes (given as 277725 integers and 770851 integers) and for processor 0 and 1 reciprocally. These results corresponds to means.

Thus, we find that our new schema does not improve significantly the load balance factor for PSS. It is partly expected because we know that PSS has, by nature, important deviation regarding the load balancing factor. Here the load balancing factor has a deviation of about 14% (in mean) ; we have measured 15% of deviation on a 4 processors system [CJF03] and according to a performance vector set to $\{8, 5, 3, 1\}$.

In fact, such deviation is due to the number of selected pivots and the way they are selected. We conclude that our new schema can not compensate totally the deviation of PSS. However, we notice that if we consider individually the results on each node, our schema provides less dispersion than for our previous schema. It is promising.

We are currently experimenting with Parallel Sort by Overpartitioning which has good properties in terms of load balancing and execution time [CJF03]. We are now convinced that improvements in performance are possible for such technique.

7 Conclusion

In this paper we have demonstrated how to optimize the data distribution in the case of sorting on heterogeneous platforms. Previous works on this subject have concentrated their efforts on load balancing but not on execution time criteria. This paper makes the bridge between the two required properties. The new technique is a constant time one. Consequently it introduces no overhead regarding the sorting problem.

We would like also to mention that the technique used in this paper forms a meta-partitioning schema, generalizing our previous works, and it can be reused in the case where the sequential brick of the parallel algorithm has a time complexity different than $n \log n$. Imagine for instance that we have a problem that can be divided into independent portions and that we have to use a sequential algorithm of time complexity $\sqrt{n}$.

We would like to execute the parallel algorithm on an heterogeneous cluster of p processors. The problem here is to find the $n_i, (1 \leq i \leq p)$ such that $n = n_1 + \cdots + n_p$ and $\sqrt{n_1}/k_1 = \cdots = \sqrt{n_p}/k_p$. We are guessing that Taylor development of $\sqrt{n+1}$, i.e. $(1 + 1/2 * n - 1/8 * n^2 + 1/16 * n^3 - 5/128 * n^4 + 7/256 * n^5 + \mathcal{O}(n^6))$ could be used in this case.

So, we are planning to solve the problem of computation of the n_i values for a large scale of usual time complexity. In this way, one can imagine that our meta-partitioning schema will organise itself according to a sequential portion of code and for the purpose of distributing data in an efficient way.

In the future, we will also consider the problem of finding a solution for the partitioning problem when the complexity of the sequential algorithm cannot be expressed with rational fractions. Dynamic programming could certainly offer solutions with an overhead that should be maintain as low as possible. We think that the main interest in using dynamic programming for our problem resides in the fact that we

may assume that the cost function, here $n \log n$, is not necessary decomposable under rational fractions.

References

[Bat68] K. E. Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[BH82] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. In *ACM Symposium on Theory of Computing (STOC '82)*, pages 338–344, Baltimore, USA, May 1982. ACM Press.

[BR93] David T. Blackston and Abhiram Ranade. SnakeSort: A family of simple optimal randomized sorting algorithms. In P. Bruce Hariri, Salim; Berra, editor, *Proceedings of the 1993 International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 201–204, Syracuse, NY, August 1993. CRC Press.

[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.

[CG00a] Christophe Cérin and Jean-Luc Gaudiot. Evaluation of two bsp libraries through parallel sorting on clusters. In *Proceedings of WCBC'00 (The Second International Workshop on Cluster-Based Computing) in conjuction with ICS'00 (International Conference on Supercomputing, sponsored by ACM/SIGARCH)*, pages pp 21–26, Santa Fe, New Mexico, 6May 2000.

[CG00b] Christophe Cérin and Jean-Luc Gaudiot. An over-partitioning scheme for parallel sorting on clusters running at different speeds. In *Cluster 2000. IEEE International Conference on Cluster Computing. Technische Universität Chemnitz, Saxony, Germany. (Poster)*, 28November-2December 2000.

[CG00c] Christophe Cérin and Jean-Luc 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, 17-20December 2000.

[CG02] Christophe Cérin and Jean-Luc Gaudiot. On a scheme for parallel sorting on heterogeneous clusters. *FGCS (Future Generation Computer Systems*, 18(issue 4), 2002. The special issue is preliminary scheduled for publication in future vol.

[CJF03] Christophe Cérin, Mohamed Jemni, and Hazem Fkaier. A synthesis of parallel out-of-core sorting programs on heterogeneous clusters. In *3st International Symposium on Cluster Computing and the Grid, Tokyo*, May 2003.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal of Computer*, 17:770–785, aug. 1988.

[HJB96] David R. Helman, Joseph JáJá, and David A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Technical Report CS-TR-3670 and UMIACS-TR-96-54, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, August 1996.

[Lei84] T. Leighton. Tight bounds on the complexity of parallel sorting. In *ACM Symposium on Theory of Computing (STOC '84)*, pages 71–80, Baltimore, USA, April 1984. ACM Press.

[LLS+93] 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, October 1993.

[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.

[NS82] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *J. ACM*, (29):642–667, 1982.

[Pla89] C. G. Plaxton. Efficient computation on sparse interconnection networks. Technical Report STAN-CS-89-1283, Department of Computer Science, Stanford University, September 1989.

[Qui89] M.J. Quinn. Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

[RV83] John H. Reif and Leslie 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 April 1983.

[RV87] J. H. Reif and L. G. Valiant. A Logarithmic time Sort for Linear Size Networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[SS92] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[TB93] A. Tridgell and R.P. Brent. An implementation of a general-purpose parallel sorting algorithm. Technical Report TR-CS-93-01, Computer Science Laboratory, Australian National University, February 1993.