Dynamic Load Balance in Parallel Merge Sorting over Homogeneous Clusters

De Giusti Armando
Chair Professor.Main Researcher CONICET.
III-LIDI Instituto de Investigación en Informática LIDI.
Computer Science School.
National University of La Plata.
degiusti@lidi.info.unlp.edu.ar

Naiouf Marcelo
Chair Professor.
III-LIDI Instituto de Investigación en Informática LIDI.
Computer Science School.
National University of La Plata.
mnaiouf@lidi.info.unlp.edu.ar

Chichizola Franco
Becario de Iniciación UNLP.
III-LIDI Instituto de Investigación en Informática LIDI.
Computer Science School.
National University of La Plata.
francoch@lidi.info.unlp.edu.ar

De Giusti Laura
Becaria de Perfeccionamiento UNLP.
III-LIDI Instituto de Investigación en Informática LIDI.
Computer Science School.
National University of La Plata.
ldgiusti@lidi.info.unlp.edu.ar

Abstract

Sorting is one of the most usual and important operations carried out in a computer. The time required by the sequential sorting algorithms is an important problem when working with large sized sequences. As solution, we can consider parallelization and attaining a near optimal performance achieving a balance in the work to be done by processors. Notice that the work does not depend only on the amount of data but also on the sequence disordering, and on the relative computer power of processors.

This paper develops a technique of redistributing dynamically the data load from the prediction of work to be carried out by each processor, balancing the load among the different processes. The method is proved to reach the theoretical optimum for the parallel algorithm performance.

Keywords: *Sorting, Algorithm Parallelization, Performance Prediction, Load Balancing, Dynamic Redistribution.*

1. Introduction

Parallel processing has been increasing practically from the very beginning of digital computers. The axes encouraging the topics of concurrence in software and multiprocessing in hardware are multiple, but can be mentioned mainly the need of reducing the processing time of large data quantities. This evolution leads to a great effort to transform sequential processing into parallel.

One of the operations that are usually required as a part of the solution of more complex algorithms is that of "sorting" a data sequence in order to, for instance, access the information more efficiently [6]. The best sequential sorting algorithms have times of O(n x Log n), where n is the number of elements in the sequence so execution time is very important with increasing data. [5][7].

The solution to the increasing processing time with *n* is the parallelization of the sorting algorithm. Given a sequence of *n* data elements, these are distributed among the different processors, where they are sorted in order to merge them, thus achieving the ordering of the whole sequence.

The use of multiple processors working on subsequences of the total *n* data elements may get an optimal performance. In order to obtain it, the work to be done by each processor should be balanced. When the architecture is homogeneous, the balance only depends on the characteristics of the data to be sorted, while in the heterogeneous case, the relation between computing power of the different processors should be added[8][2][10][1].

One way of attaining load balancing between the different processes is to distribute dynamically the data among them. To do it, a part of the data elements (*block 1*) is distributed among the different tasks, then a percentage of the work is carried out in parallel, and the remaining work is estimated. From this prediction, the rest of the data

Proceedings of the 19th International Conference on Advanced Information Networking and Applications (AINA'05)
1550-445X/05 $20.00 © 2005 **IEEE**

(*block 2*) is distributed trying to balance the work of all the processes.

2. PSDR Method

In this section the Parallel Sorting with Dynamic Redistribution (PSDR) method is presented, including the definition of a work prediction function for the parallel algorithm.

The swap method with sentinel is analyzed, which carries out a series of iterations comparing in each of them all the adjacent data pairs, and swapping them if they are disordered [5]. The algorithm finishes when all the data are sorted. When parallelizing the algorithm, each process sorts its subsequence using this method and then the sorted parts are merged (parallel merge sort) [7][2][8].

In order to obtain a good global performance in the data sequence sorting. So as to obtain this equity, a dynamic data redistribution is used from the work carried out by each processor up to a certain moment of the ordering, and the prediction of the remaining work for each process.

We will analyze four stages of the research carried out: *Sequential Work Prediction, Sorting Algorithm Parallelization, Work Prediction in the Parallel Model* to be carried out by each process, and *Dynamic Redistribution or Balance* of the remaining data among processors.

2.1. Sequential Work Prediction

In the algorithm used, the work necessary to sort *n* data elements not only depends on the data quantity (*n*), but also on the their distribution within the sequence. For this reason, it is really difficult to determine "a priori" the work and, thus, the necessary time for carrying out the sorting.

As the sorting algorithm is based on executing comparisons and swaps, the work is considered to be given by a combination of the quantity of these operations:

$$W = NC + (NS * Co)$$, where:

W is the work - NS is the number of swaps.

NC is the number of comparisons.

Co is the coefficient that indicates the relation between the necessary work for a swap and a comparison.

As the algorithm progresses, the work to be done in each iteration decreases [9]. Due to this fact, when the percentage $K1$ (with $K1 << 50$ %) of iterations has been executed, the work necessary to carry out the complete data sorting can be estimated. This iteration quantity represents a percentage $K2$ of the total work[3]. In consequence, the work can be estimated by the following formula:

$$TEW = DW*100 / Cp$$, where

TEW is the total estimated work.

DW is the work carried out in $K1$% of the iterations.

Cp represents the work percentage performed in the $K1$% of the iterations.

2.2. Sorting Algorithm Parallelization

There exist several techniques for Parallel Sorting, one of them being the Sorting by Merging, which uses the master-slave paradigm and consider the following steps:

- The *n* items are divided in *k* subsequences of equal size.
- Each subsequence is sent to a different processor, which sorts it using the previously explained "swap method" with sentinel.
- The Merge of the *k* subsequences sortered by each processor is carried out, obtaining a complete, ordered sequence. A single stage merge is implemented, instead of a multi-step merge, since the latter solution requires more communication among processors, affecting the algorithm performance in a cluster architecture [3].

In this way, the sorting algorithm performance is determined by the process which should carry out most of the work.

2.3. Work Prediction in the Parallel Model

As was previously explained, each process carries out the sorting of n/k data elements, and in order to estimate the work to be performed by each of them, the formula detailed in step 2.1 is used. As processes are being executed in parallel, the work considered for this stage is determined by that of the process which will carry out most of the work.

In order to estimate the parallel work, we should add - to the previous- the work necessary to perform the merge among the ordered subsequences, thus obtaining the final sequence.

The estimate is represented in the following formula:

$$PW = \max_{i \in [1..k]} (TEWi) + MW$$, where:

PW is the parallel work.

TEWi estimated work for process i by means of the formula explained in 2.1.

MW is the work for carrying out the merge.

2.4. Redistribution or Dynamic Balance

From the work estimate to be carried out in each processor through the formula in ***stage 2.2***, the data can be redistributed, thus balancing the work of each process.

The redistribution is carried out as follows:

- A percentage of data (*block 1*) is equally distributed into each process, thus obtaining a rest without being distributed.
- The work to be done is estimated for each process from the performed in the $K1$% of the iterations, according to the formula mentioned in 2.2.

• The rest of the data (*block 2*) is redistributed, trying to balance the total work to be carried out by each process (total work = work (block 1) + work (block 2)).
• Each process merges sorted blocks 1 and 2.
• Each process sorted subsequences are gathered, and the merge is carried out in order to obtain the final ordered sequence.

3. Experimentation and obtained results

The support for carrying out the experimentation was C language with the MPI library for communications, over a cluster of 20 homogeneous PCs (Pentium IV) networked by an Ethernet network of 100 Mbytes [4].

Two additional algorithms were implemented in order to compare the load balance:
• Parallel Sorting without redistribution (PSWR), in which all the data are equally distributed (in terms of data quantity) among all the processes.
• Parallel Sorting with fixed redistribution (PSFR), in which a percentage of data is initially and equally distributed (in terms of quantity) and then the rest is distributed in the same way.

Several tests were carried out including different sizes of sequences to be ordered (50000, 100000, 500000, and 1000000 of data elements) as well as different quantities of tasks to be used (4, 8 and 16 tasks).

The initially undistributed data percentage (block2) was 10%, 20%, 30%, and 40%.

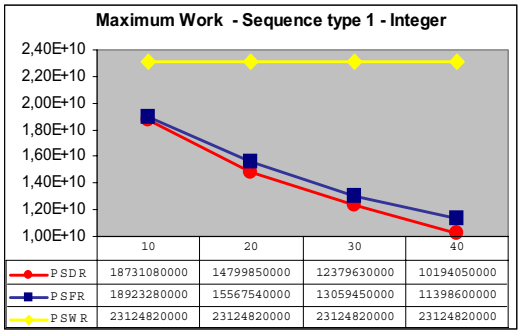
In addition, the tests included different types of sequences according to the distribution of the data initially sent to each process :
• the data sent to a task are completely inverted, and the rest are random (type 1).
• the data sent to half of the tasks are inverted and the rest are random (type 2).

The load balance was established as the difference between the maximum and minimum work divided by the average work performed among all the processes involved in the execution.

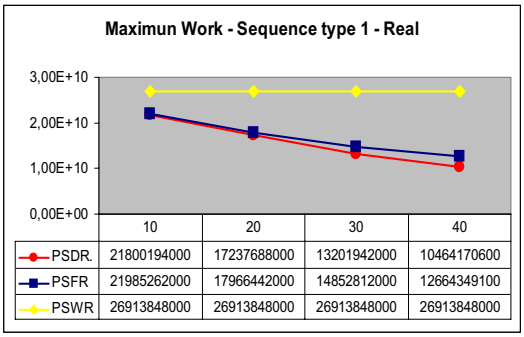
In a first instance, we experimented with integer sequences (integer of 4 bytes), then with real numbers (long double of 12 bytes) in order to perform the behavior of the algorithm when working with structures of larger sizes. In both cases, the percentage of iterations used in order to predict the work to be done was 5% ($K1 = 5$). And the work percentage performed in that 5% of the iterations ($K2$) was of approximately 11% for integers and 13% for reals [3].

3.1. Maximum Work

The graphics 1 and 2 show the maximum work carried out by each of the three algorithms with 1000000 of data, 8 tasks, and using the type 1 of sequences.



Graphics 1. Maximun work with integer sequences.

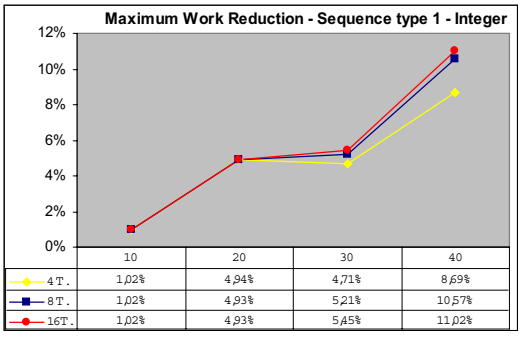


Graphics 2. Maximun work with real sequences.

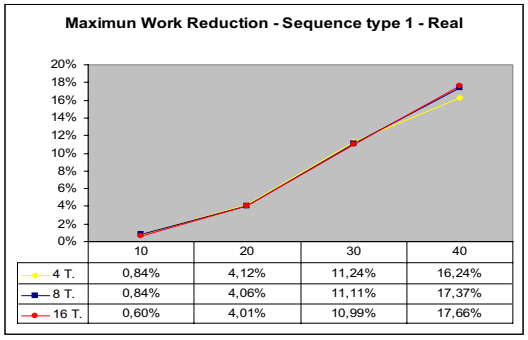
3.2. Maximum Work Reduction Percentage

The graphics 3 and 4 show the reduction relation among the maximum works carried out in the methods with dynamic and fixed redistribution, for 1000000 of data and with 4, 8, and 16 tasks, using sequences of the two types. The reduction is computed through the following count:

R= ((MWFR – MWDR) / MWFR) * 100, where
R is the reduction percentage obtained.
MWFR is the maximum work with fixed redistribution.
MWDR is the maximum work with dynamic redistribution.



Graphics 3. Maximun work reduction with integer sequences.

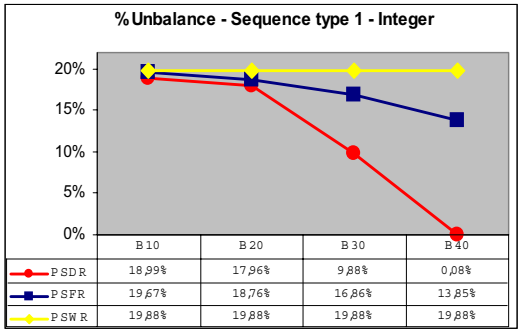


Graphics 4. Maximun work reduction with real sequences.

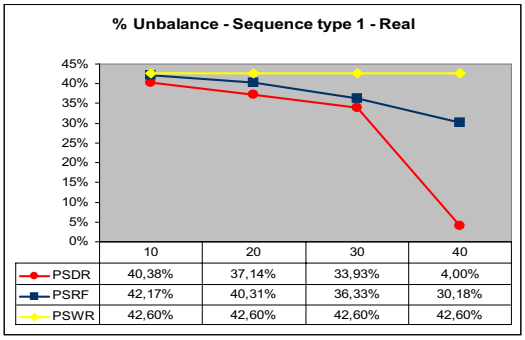
3.3. Load Balance

In order to analyze the load balance in the system, the difference between the maximum work and the minimum carried out in each algorithm is presented. On the other hand, the unbalance percentage, which represents such difference, is also computed in relation to the mean work.

The graphics 5 and 6 show the previously mentioned unbalance percentage in the three algorithms, for 1000000 of data, with 8 tasks, and with the two types of sequences.



Graphics 5. Unbalance percentage with integer sequences.



Graphics 6. Unbalance percentage with real sequences.

4. Conclusions and future work

There exists a great number of load balance techniques, both static and dynamic. Even though the static techniques are easier to carry out, they cannot be used when the work load is not known " a priori". In these cases, dynamically redistributing the load can attain a better load balance, without producing a high overhead.

This paper shows a way of estimating *dynamically* the work to be done in order to sort a data sequence by means of the swap method with sentinel.

When the ordering performance in parallel can be predicted from the work estimated for each process, it is possible correct and optimize the data distribution per processor, improving in this way the performance of the complete operation.

The graphics have shown that the proposed dynamic redistribution method balances in a better manner the work load among processors, reducing the maximum work to be done and, thus, the final work, optimizing the system performance as well.

It can also be noticed that the method can be used with different structures of data to be ordered, and that the greater this structure is, the greater will be the reduction in the maximum work to be done, which, in turn, leads to an more significant improvement in the final system's performance.

As future work, there exists an attempt to include processors heterogeneity as another factor in determining the quantity of work that each processor should perform. In addition, the considerations and necessary changes for using multicluster environments - communicating a cluster in Argentina (UNLP), other in Spain (UAB), and another in Brazil - are being analyzed.

Also, the algorithm is being modified so it can be executed in different architecture models, in particular, distributed-shared memory architectures (SGI Origin 2000).

5. References

[1], Amato N., Ravishankar Iyer, Sharad Sundaresan, Yan Wu, "A Comparison of Parallel Sorting Algorithms on Different Architectures", Technical Report No. 98/029, Department of Computer Science. Texas A\&M University, 1996.

[2], Cole R., "Parallel Merge Sort", Siam Journal of Computing, Vol.17 Nº4, 1998.

[3], De Giusti Laura, Chichizola Franco. Informe Técnico.

[4], IBM. Informe técnico, 2003.

[5], Knuth, "The Art of Computer Programming, Vol 3: Sorting and Searching", Addison-Wesley, 1973.

[6], Kumar V., Grama A., Gupta A., Karypis G., "Introduction to Parallel Computing. Design and Analysis of Algorithms", Benjamin/Cummings, 1994.

[7],Lang, "Sequential and Parallel Sorting Algorithms", www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm.

[8], Minsoo Jeon, Dongseung Kim, "Parallel Merge Sort With Load Balancing", International Journal of Parallel Programming, Vol.31 Nº1, 2003.

[9], Naiouf Marcelo, "Procesamiento paralelo. Balance de carga dinámico en algoritmos de sorting.", Tesis Doctoral, Julio 2004.

[10], Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, Hanmao Shi, "On the versatility of Parallel Sorting by Regular Sampling", Parallel Computing Vol.19, 1993.