A New Parallel Sorting Algorithm based on Odd-Even Mergesort

Ezequiel Herruzo, Guillermo Ruíz, J. Ignacio Benavides
Dept. Computer Architecture and Electronics
University of Córdoba, Spain
{eze,i22rurag,el1bebej}@uco.es

Oscar Plata
Dept. Computer Architecture
University of Málaga, Spain
oscar@ac.uma.es

Abstract

This paper describes a new parallel sorting algorithm, derived from the odd-even mergesort algorithm, named "partition and concurrent merging" (PCM). The proposed algorithm is based on a divide-and-conquer strategy. First, the data sequence to be sorted is decomposed in several pieces that are sorted in parallel using Quicksort. After that, all pieces are merged using a recursive procedure to obtain the final sorted sequence. In each iteration of this procedure pairs of sequence pieces are selected and sorted concurrently. The paper analyzes the computational complexity of the new algorithm and compares it with that of other well-known parallel sorting algorithms. We implemented the PCM algorithm on a SGI Origin2000 multiprocessor using OpenMP, sorting different benchmark sets of data sequences. Experimental results are compared with those of the Quicksort sequential algorithm and parallel implementations of other sorting algorithms, obtaining that our proposal outperforms the other solutions.

1. Introduction

Sorting the elements of a data sequence is a very common task in many application areas. It is important for optimizing the use of many algorithms, such as searching algorithms, that require sorted data lists to work efficiently. It is frequently used, for instance, to execute general data permutations, including random read and write accesses. These operations of data exchange can be used to solve problems in graph theory, computational geometry and image processing in (near) optimal time. Their importance is also represented by the interest to develop efficient parallel implementations [4][11][12].

This paper presents a new parallel sorting algorithm, called *partition and concurrent merging*. Its computational complexity is analyzed, and some comparisons with other well-known sorting algorithms, both sequential versions, like with Quicksort [3][6][8], and parallel versions, like with Odd-Even Mergesort [1][5][8], Bitonic Sort [1][3][5], Shellsort [5][7][9] and PSRS (Parallel Sorting by Regular Sampling) [10], are made. The rest of the paper is organized as follows. Section 2 describes the proposed parallel sorting algorithm PCM. Section 3 presents the analysis of the computational complexity of PCM. In section 4 we show the experimental evaluation of the algorithm compared with other implementations. Finally we draw some conclusions.

2. Description of the algorithm

The proposed parallel sorting algorithm is based on a divide-and-conquer strategy. In a first stage, the data sequence to be sorted is partitioned into several pieces that are sorted concurrently (fully parallel). In a second stage, all sorted pieces are merged using a recursive procedure to finally obtain the full sorted data sequence. In each iteration of this procedure pairs of previously sorted pieces are selected, merged and sorted in a concurrent way. This merging and sorting task of two sequence pieces is made by a new algorithm we have developed called PREZ. Eventually, in the last iteration of the procedure, two large pieces remain, which are merged and sorted to produce the complete sorted data sequence.

We call *partition and concurrent merging* (PCM) the complete sorting algorithm composed of the above two stages. Though the complete process is similar to other sorting algorithms, however its efficiency depends to a large extent on the PREZ algorithm, that constitutes the main contribution of the paper.

2.1. Partition and concurrent merging

Let us assume a shared-memory multiprocessor with N processors, denoted by P_1, P_2 ... P_N. To simplify the explanation let us consider N an even number. Let us also assume a data sequence D of size S initially unordered. In our parallel sorting algorithm the sequence D is partitioned into subsequences of size S/N, being S divisible by N[(1)]. Each subsequence is denoted by D_i, and assigned to the processor P_i.

In a first step of our algorithm each processor sorts its assigned subsequence using a fast (sequential) sorting algorithm, like Quicksort. After that, in a second step, each odd-tagged processor P_i (with an odd i) executes the PREZ algorithm to merge and sort the subsequences assigned to P_i and P_{i+1}, that is, subsequences D_i and D_{i+1}, resulting in a new sorted subsequence R of double size. This new subsequence is later in-place stored, the first half or R replacing the original D_i and the second half of R replacing D_{i+1}. In a third step, the same process is executed but by even-tagged processors. Both second and third steps are iterated $N/2$ times, doubling each time the size of the resulting subsequence R. At the end, the parallel sorting algorithm ends with a completely sorted sequence D.

A pseudocode of the PCM algorithm is as follows:

```
Procedure PCM
  do parallel
    $P_i$:{ quicksort($D_i$) }
  end do parallel
  for $k = 1$ to $N/2$ do
    do parallel
      $P_i$, $i$ odd:{ PREZ($i, D_i, D_{i+1}, R$)
                   $D_i \leftarrow R[1, ..., S/N]$
                   $D_{i+1} \leftarrow R[S/N+1, ..., 2*S/N]$ }
    end do parallel
    do parallel
      $P_i$, $i$ even:{ PREZ($i, D_i, D_{i+1}, R$)
                    $D_i \leftarrow R[1, ..., S/N]$
                    $D_{i+1} \leftarrow R[S/N+1, ..., 2*S/N]$ }
    end do parallel
  end for
end Procedure
```

2.2. The PREZ algorithm

The parallel sorting algorithm described in the previous subsection is similar to other well-known algorithms appearing in the literature. The main difference between those algorithms and ours lies in the merging and sorting process of two subsequences, what we called PREZ algorithm. This algorithm works in parallel involving two processors that cooperatively merge and sort two subsequences (previously sorted in an independent way), obtaining a single, fully sorted subsequence of double size.

The PREZ algorithm takes a tag of a processor (i) and the two subsequences to be merged and sorted (D_1 and D_2) as the input arguments, returning the merged, sorted double-sized subsequence (R) as the output. It uses four indices, c_{11}, c_{21}, c_{12} and c_{22}, to traverse both input subsequences. Initially, processor P_i takes the indices c_{11} and c_{21}, that point to the first entry of D_1 and D_2. The smallest of these two entries are assigned to the first entry of R. Afterwards, the index pointing to the smallest entry is increased and the process is repeated, assigning the result to the next entry of R. This process is iterated until the first half of R is filled. At the same time, in parallel, processor P_{i+1} takes the indices c_{12} and c_{22}, that point to the last entry of D_1 and D_2. The greatest of both entries is assigned to the last entry of R. The index pointing to the greatest entry is decreased and the process is repeated as well until the second half of R is filled. When the full procedure finishes R contains all entries of D_1 and D_2 sorted.

A pseudocode for PREZ is shown below.

```
Procedure PREZ($i, D_1, D_2, R$)
  $c_{11} = 1$; $c_{21} = 1$; $c_{12} = s$; $c_{22} = s$      // $s$ is $S/N$
  do parallel      // Only two processors, $P_i$ and $P_{i+1}$
    $P_i$:{
          for $k = 1$ to $s$ do
            if ($D_1[c_{11}] <= D_2[c_{21}]$) then
              $R[k] = D_1[c_{11}]$;  $c_{11} = c_{11}+1$
            else
              $R[k] = D_2[c_{21}]$;  $c_{21} = c_{21}+1$
            end if
          end for    }
    $P_{i+1}$:{
          for $k = 1$ to $s$ do
            if ($D_2[c_{22}] > D_2[c_{12}]$) then
              $R[(2*s+1) - k] = D_2[c_{22}]$;  $c_{22} = c_{22}-1$
            else
              $R[(2*s+1) - k] = D_1[c_{12}]$;  $c_{12} = c_{12}-1$
            end if
          end for    }
  end do
end Procedure
```

2.2. Example

Figure 1 shows an application example of the parallel PCM sorting algorithm, as described in the

(1) In case that S is not divisible by N, null data entries may be included to complete the subsequences, or some data entries may be excluded from the subsequences (and sorted sequentially at the end of the algorithm). For the sake of simplicity, we do not consider this situation in the description of the algorithm.

previous subsections. The data sequence to be sorted contains 12 entries, as follows, D = {7, 0, 9, 1, 5, 6, 5, 2, 8, 4, 3, 1}, that will be sorted using four processors (N = 4). The sequence D is initially partitioned into four 3-entry subsequences that are sorted individually in parallel using Quicksort. Afterwards, it starts the main loop that iterates two times. At each iteration, subsequences are merged and sorted, working in parallel first the odd-tagged processors and next the even-tagged processors. Finally, the sequence D gets sorted, distributed among all processors.

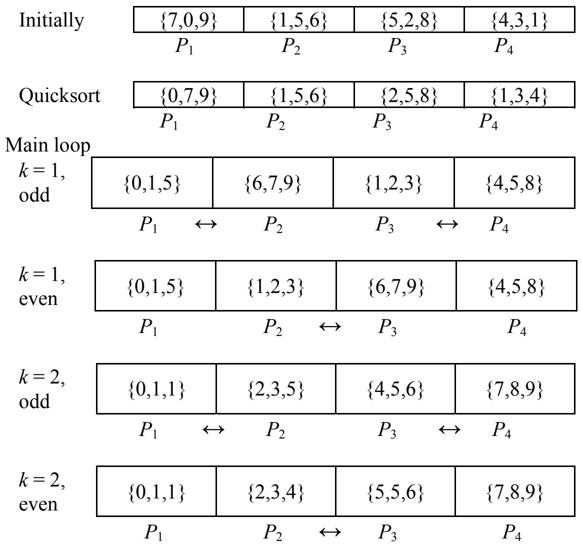

Figure 1. Sorting a 12-entry sequence.

3. Computational complexity

3.1. Complexity analysis of the PCM algorithm

This section is devoted to a detailed analysis of the computational complexity of the proposed parallel sorting algorithm. We split the analysis in two parts, the initial fully parallel phase, based on the Quicksort algorithm, and the main iterative loop where a concurrent merging a sorting phase is executed, based on the PREZ algorithm.

3.1.1. Parallel quicksort-based phase complexity. The first phase of our algorithm is a quicksort-based fully parallel sorting of subsequences. Quicksort [6] is a very well-known sorting algorithm that has been extensively studied in the literature. On average, the computational complexity of Quicksort is $\Theta(n \log n)$, that corresponds to the number of comparisons to sort n values. However, the complexity reaches $\Theta(n^2)$ in the worst case, though is significantly faster in practice.

In the case of the PCM algorithm, Quicksort is used in parallel to sort subsequences of size S/N, so as the computational complexity of this first phase is $\Theta(S/N \log (S/N))$, or $\Theta((S/N)^2)$ in the worst case.

3.1.2. Main iterative loop complexity. The main iterative loop of the PCM algorithm basically consists of a series of calls to the PREZ routine. In each call to PREZ two (previously) sorted subsequences of size S/N are merged and sorted. In the PREZ algorithm two processors work in parallel, each one executing a total of S/N comparisons. So, although in PREZ a total of $2S/N$ comparisons are executed, however as both processors work in parallel, the effective computational complexity is $\Theta(S/N)$.

On the other hand, the body of the main iterative loop contains two parts executed sequentially. In the first one, every odd-tagged processor executes one instance of the PREZ algorithm. The second step is similar but involving all the even-tagged processors except the last one P_N (because D_{N+1} does not exist). Therefore, PREZ is called a total of N–1 times in each main loop iteration. However, as each part is concurrent, that is, all odd-tagged processors work in parallel (and all even-tagged processors as well), the effective number of calls to PREZ is 2 (the first $N/2$ calls are concurrent, and the rest $N/2–1$ calls are also concurrent). Hence, the effective number of comparisons of the main loop body is $2S/N$.

As the main loop executes a total of $N/2$ iterations, the number of effective comparisons for that loop is $(N/2)(2S/N) = S$. As a result, the computational complexity of the main iterative loop is $\Theta(S)$.

3.1.3. PCM complexity. According to the analysis in the above subsections, the computational complexity of the parallel PCM sorting algorithm is $\Theta(S/N \log S/N) + \Theta(S)$.

3.2. Comparison with other sorting algorithms

Table I shows a chart with the computational complexities of some classical sorting algorithms, that allows to compare them with that of our proposed algorithm.

The computational complexities have a first term that is the same for all sorting algorithms, which corresponds to a first phase of local sorting based on Quicksort. The rest of the terms in the complexities depend on the peculiarities of each algorithm.

Table I. Complexities of sorting algorithms

PCM	$\Theta(S/N \log(S/N)) + \Theta(S)$
Odd-Even Mergesort	$\Theta(S/N \log(S/N)) + \Theta(2S)$
Bitonic Sort	$\Theta(S/N \log(S/N)) + \Theta(2S/N \log^2 N)$
Shellsort	$\Theta(S/N \log(S/N)) + \Theta(S/N \log N) + \Theta(2S/N)$
Parallel Sorting by Regular Sampling	$\Theta(S/N \log(S/N)) + \Theta(N^2 \log N) + \Theta(N \log(S/N)) + \Theta(S/N)$

The complexity of our algorithm is smaller than that of the Odd-Even Mergesort due to the second term. Regarding the Bitonic Sort, PCM presents a lower complexity for a number of processors fulfilling $N < 2 \log^2 N$, that is, for a number of processors smaller than 128. In contrast, compared with Shellsort, our algorithm exhibits less computational complexity only when $N < (\log N + 2)$, that is, when $N < 4$. Finally, regarding PSRS, our proposal is less complex when $S < N^2 \log N + N \log(S/N) + S/N$, that occurs for small values of S and/or for large values of N.

4. Experimental evaluation

We have conducted an experimental evaluation of our parallel PCM sorting algorithm on a shared-memory multiprocessor system. We have implemented the algorithm in OpenMP [2] and made experimental executions on a SGI Origin2000 system, for different processor counts and data sequences of various sizes.

Specifically, a number of experimental setups were defined. The parallel execution time of the algorithm was measured for 2, 4 and 6 processors. Each experiment was carried out for input data sequences of various sizes, ranging from 10^6 to10^7 entries. For each combination of number of processors and sequence size, the experiment was repeated 30 times with the contents of the data sequences randomly generated.

4.1. Comparison with the sequential Quicksort

Table II shows the results of the average speed-up obtained for PCM using the described experimental setup compared with the sequential Quicksort algorithm. The size S of the input data sequences is specified in millions of entries (10^6).

Table II. Speed-up of our parallel algorithm compared with the sequential Quicksort

	QSort	*PCM, N*=2	*PCM, N*=4	*PCM, N*=6
$S = 1$	1	1,680	2,692	3,180
$S = 1.1$	1	1,664	2,704	3,234
$S = 1.2$	1	1,680	2,716	3,318
$S = 1.3$	1	1,674	2,720	3,342
$S = 1.4$	1	1,684	2,728	3,354
$S = 1.5$	1	1,694	2,736	2,970
$S = 2$	1	1,704	2,768	3,414
$S = 3$	1	1,704	2,752	3,090
$S = 4$	1	1,716	2,736	3,120
$S = 5$	1	1,724	2,752	3,168
$S = 7.5$	1	1,720	2,772	3,168
$S = 10$	1	1,728	2,780	3,294

Form the table we can conclude that the PCM algorithm obtains acceptable speed-ups compared with one of the fastest sorting algorithms. In addition, the performance does not depend much on the size of the data sequence to be sorted (at least in the range tested).

4.2. Comparison with other parallel sorting algorithms

We have also implemented a selection of well-known parallel sorting algorithms, in order to compare their performance with the proposed PCM. Specifically, we have chosen the following algorithms: Odd-Even Mergesort [3], Bitonic Sort, Shellsort and PSRS [8].

Table III. Number of processor cycles for 2, 4 and 6 processors

	2 processors					**4 processors**					**6 processors**		
	PCM	***OE Msort***	***Bionic Sort***	***Shell sort***	***PSRS***	***PCM***	***OE Msort***	***Bitonic Sort***	***Shell Sort***	***PSRS***	***PCM***	***OE Msort***	***PSRS***
$S = 1$	0,322	0,553	0,543	0,542	0,347	0,201	0,552	0,543	0,543	0,193	0,170	0,552	0,130
$S = 1.1$	0,359	0,619	0,609	0,607	0,386	0,221	0,619	0,608	0,607	0,209	0,185	0,619	0,143
$S = 1.2$	0,391	0,663	0,658	0,656	0,420	0,242	0,663	0,657	0,656	0,220	0,198	0,663	0,156
$S = 1.3$	0,427	0,710	0,724	0,719	0,456	0,263	0,709	0,721	0,719	0,239	0,214	0,709	0,170
$S = 1.4$	0,460	0,775	0,779	0,775	0,492	0,284	0,774	0,778	0,775	0,259	0,231	0,774	0,184
$S = 1.5$	0,494	0,844	0,838	0,835	0,531	0,306	0,844	0,839	0,835	0,278	0,282	0,844	0,197
$S = 2$	0,669	1,146	1,141	1,140	0,714	0,412	1,146	1,147	1,140	0,372	0,334	1,146	0,265
$S = 3$	1,027	1,748	1,753	1,748	1,098	0,636	1,748	1,763	1,748	0,578	0,566	1,748	0,404
$S = 4$	1,390	2,411	2,400	2,395	1,485	0,872	2,411	2,419	2,394	0,778	0,765	2,411	0,545
$S = 5$	1,756	3,040	3,045	3,042	1,873	1,100	3,040	3,079	3,045	0,982	0,955	3,040	0,691
$S = 7.5$	2,700	4,579	4,646	4,638	2,884	1,676	4,579	4,699	4,637	1,497	1,465	4,579	1,051
$S = 10$	3,652	6,306	6,356	6,339	3,903	2,269	6,307	6,357	6,335	2,041	1,917	6,307	1,427

Table IV. Speed-up of various parallel sorting algorithms for 2, 4 and 6 processors

	Odd-Even MergeSort			***Bitonic Sort***		***ShellSort***		***PSRS***		
	N = 2	*N* = 4	*N* = 6	*N* = 2	*N* = 4	*N* = 2	*N* = 4	*N* = 2	*N* = 4	*N* = 6
S = 1	1,714	2,748	3,240	1,687	2,701	1,685	2,699	1,078	0,960	0,762
S = 1.1	1,724	2,800	3,348	1,697	2,750	1,691	2,747	1,076	0,944	0,774
S = 1.2	1,696	2,740	3,348	1,683	2,716	1,678	2,711	1,074	0,908	0,786
S = 1.3	1,662	2,692	3,312	1,695	2,740	1,683	2,733	1,068	0,908	0,792
S = 1.4	1,684	2,724	3,348	1,693	2,738	1,685	2,729	1,070	0,912	0,798
S = 1.5	1,708	2,760	3,354	1,697	2,741	1,691	2,730	1,074	0,908	0,696
S = 2	1,714	2,780	3,432	1,706	2,784	1,704	2,767	1,068	0,904	0,792
S = 3	1,702	2,748	3,144	1,707	2,772	1,702	2,749	1,070	0,908	0,714
S = 4	1,734	2,764	3,150	1,726	2,774	1,723	2,746	1,068	0,892	0,714
S = 5	1,732	2,764	3,186	1,734	2,799	1,732	2,768	1,066	0,892	0,726
S = 7.5	1,696	2,732	3,126	1,721	2,804	1,718	2,767	1,068	0,892	0,720
S = 10	1,726	2,780	3,288	1,740	2,802	1,736	2,792	1,068	0,900	0,744

The parallel version of Quicksort was not implemented due to the specific implementation of OpenMP in the SGI machine, where nested parallelism is not allowed. This fact introduces a lot of complexities when encoding the parallel Quicksort. Table III shows the performance results (number of processor cycles measured in thousands of millions) from our experiments with all the considered parallel sorting algorithms. All experiments were executed for 2, 4 and 6 processors except for Bitonic Sort and Shellsort, that they were not tested for 6 processors, since both algorithms require a power of two number of processors.

Table IV shows the speed-up of the PCM algorithm with regards to the other four sorting algorithms. The table shows that our algorithm outperforms the other algorithms for the majority of the cases (speed-up greater than 1). Only PSRS performs better than our PCM algorithm for 4 and 6 processors.

5. Conclusions

The paper presents a new fast parallel sorting algorithm called PCM (*parallel and concurrent merging*). After an initial partition step of the data sequence, the algorithm works in two phases. The first phase sorts in parallel all the subsequences using Quicksort. The second phase is an iterative process that merges and sorts the subsequences by pairs. In the core of this second phase is the PREZ algorithm, the main contribution of the paper.

The PCM algorithm was implemented in OpenMP and evaluated in a SGI Origin2000 multiprocessor, comparing the results with other implementations of well-known parallel sorting algorithms. The results show that PCM outperforms in most cases sorting algorithms like Odd-Even Mergesort, Bitonic Sort and Shellsort. However, the PSRS algorithm performs better than PCM for 4 and 6 processors. As future work we intend to analyze in depth this situation in order to improve PCM. In addition, we are designing a new set of experiments using dual-core processors as the parallel platform.

In general, the proposed PCM algorithm represents an improvement with regards to classical algorithms, like Odd-Even Mergesort, Bitonic Sort and Shellsort.

6. References

[1] K.E. Batcher, “Sorting Networks and their Applications”, AFIPS Spring Joint Comput., 32:307-314, 1968.

[2] R. Chandra, D. Maydan, J. McDonald, R. Menon, L. Dagum, and D. Kohr, “Parallel Programming in OpenMP”, Morgan Kaufmann Pub., 2001.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, “Introduction to Algorithms”, The MIT Press, 1994.

[4] R.S. Francis and I.D. Mathieson, “A Benchmark Parallel Sort for Shared-Memory Multiprocessors”, IEEE Trans. on Computers, 37(2):1619-1626, 1988.

[5] A. Grama. et al., “Introduction to Parallel Computing”, 2nd Edition, Addison-Wesley, 2003.

[6] C. Hoare, “Quicksort”, Computer J., 5(1):10-15, 1962.

[7] V. Pratt, “Shellsort and Sorting Networks”, Garland, New York, 1979.

[8] R. Sedgewick, “Algorithms in Java”, Parts 1-4. 3. Auflage, Addison-Wesley, 2003.

[9] D.L. Shell, “A High-Speed Sorting Procedure”, Communications of the ACM, 2(7):30-32, 1959.

[10] H. Shi, “Parallel Sorting on Multiprocessor Computers”, M.Sc.Thesis, Dept. Computing Science, Univ. Alberta, 1990.

[11] G. Bilardi and A. Nicolau, “Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines”, SIAM J. of Computing, 18:216-228, 1989.

[12] K. Suehiro, H. Murai and Y. Seo, “Integer Sorting on Shared-Memory Vector Parallel Computers”, 12th Int. Conf. on Supercomputing (ICS’98), 1998.