World Academy of Science, Engineering and Technology
International Journal of Electrical and Computer Engineering
Vol:2, No:9, 2008

A Message Passing Implementation of a New Parallel Arrangement Algorithm

Ezequiel Herruzo, Juan José Cruz, José Ignacio Benavides, and Oscar Plata

Open Science Index, Electrical and Computer Engineering Vol:2, No:9, 2008 publications.waset.org/13070/pdf

***Abstract*—**This paper describes a new algorithm of arrangement in parallel, based on Odd-Even Mergesort, called division and concurrent mixes. The main idea of the algorithm is to achieve that each processor uses a sequential algorithm for ordering a part of the vector, and after that, for making the processors work in pairs in order to mix two of these sections ordered in a greater one, also ordered; after several iterations, the vector will be completely ordered. The paper describes the implementation of the new algorithm on a Message Passing environment (such as MPI). Besides, it compares the obtained experimental results with the quicksort sequential algorithm and with the parallel implementations (also on MPI) of the algorithms quicksort and bitonic sort. The comparison has been realized in an 8 processors cluster under GNU/Linux which is running on a unique PC processor.

***Keywords*—**Parallel algorithm, arrangement, MPI, sorting, parallel program.

I. Introduction

THE arrangement of the elements of a vector is a process very common in the present computer systems. It is used in many algorithms so that the access to the information that these need can more efficiently be made. It is used frequently to execute general exchanges of data, including random access as much for reading as for writing. These operations of data movement can be used to solve problems in theory of graphs, computational geometry and image processing in an optimal or almost optimal time.

The present document defines a new parallel algorithm of arrangement, denominated of division and concurrent mixes; originally conceived for its application in multiprocessor system (with shared memory). Here it is described its implementation by Message Passing, MPI [6][7] and is compared with the sequential algorithm of QuickSort [5][5][9], Quicksort arrangement [8][9][10] and the Bitonic Sort [1][1][4] algorithm of arrangement in parallel.

Ezequiel Herruzo, Juan José Cruz and José Ignacio Benavides are with the Department of Computer Architecture and Electronic, University of Córdoba., Spain (corresponding author to provide phone: +34 957 218375; fax: +34 957 218316; e-mail: eze@uco.es).

Oscar Plata is with the Department of Computer Architecture, University of Málaga, Spain (e-mail: oscar@ac.uma.es).

II. Description of the Algorithm

A. Basic Idea

The main idea of the algorithm is to achieve that each processor uses a sequential algorithm to order a part of the vector, and once this has been done, to make the processors work in pairs so as to mix two of these sections ordered in a greater one, also ordered; after several iterations, the vector will be completely ordered.

Our algorithm is divided in two parts: one part of division and mixes contender of the elements of a vector, and another one that makes the ordered mixture of subvectors already sorted (algorithm PREZ).

B. Division and Mixes Contender Algorithm

We suppose a system with N processors, with a system of shared memory and capacity of concurrent and denoted reading and writing like P_1, P_2, ... P_N, being N an even number. We also suppose a vector of data S with n elements initially jumbled. This vector is divided in subvectors of length n/N, where n must be divisible by N[1], and the handling of each one of them is assigned to the processor P_i, as shown in the Fig. 1.

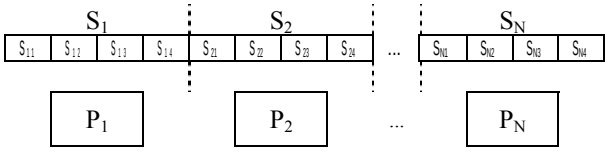


Fig. 1 Scheme of the distribution of the vector between the processors

The first step of the division and concurrent mixes algorithm is that each processor P_i may order the S_i subvector sequentially using the sequential algorithm quicksort. In the second step, the processors P_i and P_{i+1}, where i is odd, must jointly sort their subvectors S_i and S_{i+1} to a sorted vector $S_{i'}$, keeping in S_i the lower half of $S_{i'}$ and the higher half in S_{i+1}. The third step repeats the procedure with each P_i and P_{i+1} with i even (processors 1 and N, are in "stand by" in this step). After $N/2$ iterations of the second and third steps, the algorithm finishes with a sorted vector S.

[1] In case of n not divisible by N, false elements are introduced to complete the subvectors, or the leftover elements are excluded to insert them sequentially. For the sake of simplicity, we have avoided the treatment of this case in the description of the algorithm.

Open Science Index, Electrical and Computer Engineering Vol:2, No:9, 2008 publications.waset.org/13070/pdf

```
Procedure DIVISION_AND_CONCURRENT_MIXING (S)
  for i=1 to N do in paralel
    quicksort(Si)
  end for

  for j=1 to [N/2] do
    for i=1,3 ... 2*[n/2]-1 do in parallel
      PREZ(Si, Si+1, Si')
      Si<-{Si'(1), Si'(2),..., Si'(n/N)}
      Si+1<-{Si'((n/N)+1), ... Si'(2-n/N)}
    end for

    for i=2, 4, ..., 2*[(N-1)/2] do in parallel
      PREZ(Si, Si+1, Si')
      Si<-{Si'(1), Si'(2),..., Si'(n/N)}
      Si+1<-{Si'((n/N)+1), ... Si'(2-n/N)}
    end for
  end for
end procedure
```

C. PREZ Algorithm

The innovation of the method proposed here is the mixing algorithm: PREZ. PREZ involves in its execution two processors working in parallel to mix two sorted subvectors of n size, giving as a result a 2n-size vector, also sorted.

There are two processors, P1 and P2; two sorted subvectors, S1 and S2 and a vector, SR, which finally will contain the mix of S1 and S2. The algorithm uses four pointers, c11, c12, c21 and c22, two for each processor. The pointer c11 determines the position to be read in S1 by P1; c21 indicates the position of S2 which is treated by P1; in the same way, c12 and c22 are used by P2.

Initially, the pointers of P1 points to S1(1) and S2(1). These elements are compared to assign the minor of them to SR(1); afterwards it is increased the pointer which points to SR and the one that points to the subvector with the smallest of the two elements. This process is repeated n times. The behaviour of P2, working in parallel, is the opposite: the pointers initially direct to S1(n) and S2(n), the greatest one is assigned to Sr(n), and the corresponding pointers are decreased.

After n iterations, SR will contain the sequence of 2n sorted elements.

The pseudocode for the PREZ algorithm is:

```
Procedure PREZ (S1, S2, SR)
  C11=1; c12=n; c21=1; c22=n
  /* 2 is the number or processors */
  for j=1 to 2 do in parallel
    for i=1 to n do
      P1: if(s1[c11]<=S2[c22]) then
        SR[i]=S1[c11]
        C11++
      else
        SR[i]=S2[c21]
      end if
      P2: if(S2[c22]>S1[c12]) then
        SR[(2*n+1)-1]=S2[c22]
        C22–
      else
        SR[(2*n+1)-1]=S1[c12]
        C12–
      end if
    end for
  end for
end procedure
```

Fig. 2, shows an example of the sorting procedure, for a vector S={7, 0, 9, 1, 5, 6, 5, 2, 8, 4, 3, 1} with N=4 processors. For each step, the pairs of processors working in their assigned subvectors are marked in bold. The arrows indicate each pair of processors working jointly, performing the PREZ algorithm.

At the beginning	{7,0,9}	{1,5,6}	{5,2,8}	{4,3,1}
	P1	P2	P3	P4
Quicksort	{0,7,9}	{1,5,6}	{2,5,8}	{1,3,4}
	P1	P2	P3	P4
Main loop:				
j=1, Part 1	{0,1,5}	{6,7,9}	{1,2,3}	{4,5,8}
	P1 ← → P2		**P3 ← → P4**	
j=1, Part 2	{0,1,5}	{1,2,3}	{6,7,9}	{4,5,8}
	P1	**P2 ← → P3**		P4
j=2, Part 1	{0,1,1}	{2,3,5}	{4,5,6}	{7,8,9}
	P1 ← → P2		**P3 ← → P4**	
j=2, Part 2	{0,1,1}	{2,3,4}	{5,5,6}	{7,8,9}
	P1	**P2 ← → P3**		P4

Fig. 2 Example of the sorting of a 12-element vector

III. Implementation on MPI

A. Implementation of the Division and Concurrent Mixing

The main handicap to implement the PREZ algorithm in a Message Passing environment is the need to make two tasks working with the same data as in a shared memory system. To solve this problem we must implement a relatively complex mechanism of intercommunication between pairs of tasks in order to share their data in every iteration of the algorithm.

As shown in the code below, for each iteration and step, every pair of tasks shares its subvectors by passing the data to each other.

In this way, both tasks know the subvectors S_i and S_{i+1}. P_i calls the PREZ algorithm to sort the first half of the S_i vector, while P_{i+1} will sort the second half.

The below code fragment implements the procedure described above, where the used variables are:

- *v* is the entire vector *S*.
- *subvsz* is the size of subvectors *n/N*.

Open Science Index, Electrical and Computer Engineering Vol:2, No:9, 2008 publications.waset.org/13070/pdf

- *myv* is each subvector S_i.
- *recv* is the subvector received by the other task.
- *rv* is, for each task, the corresponding half of the resulting vector $S_{i'}$.
- *nproc* is the number of tasks.

```
/* scatters the work */
MPI_Scatter(v, subvsz, MPI_INT, myv, subvsz,
  MPI_INT, 0, MPI_COMM_WORLD);

/* First step: quicksort ordering */
quicksort(myv, 0, subvsz-1);

/* PREZ mixing */
for(i=0; i<(nproc/2); i++) {
  /* Second step: even mixing ([1/2][3/4]...) */
  recv = (int *) malloc(subvsz * sizeof(int));
  if(myid & 0x01) { /* odd task sends first and
                       then receives */
    MPI_Send(myv, subvsz, MPI_INT, myid-1, 0,
            MPI_COMM_WORLD);
    MPI_Recv(recv, subvsz, MPI_INT, myid-1, 0,
            MPI_COMM_WORLD, NULL);
    /* Calls the PREZ algorithm */
    rv = PREZ(myv, recv, subvsz, 0);
    }
  else { /* even receives first and then sends */
    MPI_Recv(recv, subvsz, MPI_INT, myid+1, 0,
            MPI_COMM_WORLD, NULL);
    MPI_Send(myv, subvsz, MPI_INT, myid+1, 0,
            MPI_COMM_WORLD);
    /* Calls the PREZ algorithm */
    rv = PREZ(myv, recv, subvsz, 1);
    }
  free(myv);
free(recv);
myv = rv; /* Now, S(i) is the corresponding half of
S(i') */

/* Third step: odd mixing ([2/3][3/4]...) */
if(myid > 0 && myid < (nproc-1)
    && i < ((nproc/2)-1)) {
  recv = (int *) malloc(subvsz * sizeof(int));
  if(myid & 0x01) {
    MPI_Send(myv, subvsz,MPI_INT, myid+1, 0,
          MPI_COMM_WORLD);
    MPI_Recv(recv, subvsz,MPI_INT, myid+1, 0,
          MPI_COMM_WORLD, NULL);
    rv = PREZ(myv, recv, subvsz, 1);
    }
  else {
    MPI_Recv(recv,subvsz, MPI_INT, myid-1, 0,
          MPI_COMM_WORLD, NULL);
    MPI_Send(myv, subvsz, MPI_INT, myid-1, 0,
          MPI_COMM_WORLD);
    rv = PREZ(myv, recv, subvsz, 0);
    }
  free(myv);
  free(recv);
  myv = rv;
  }
} /* end of the PREZ mixing */
/* gathers the work */
MPI_Gather(myv, subvsz, MPI_INT, v, subvsz, MPI_INT,
    0, MPI_COMM_WORLD);
```

B. The PREZ Algorithm

To implement the PREZ algorithm, it is only necessary to add a new argument, the *pos* argument, which indicates the portion of the resulting vector $S_{i'}$ that will process the task: 0 for the first half and 1 for the last half.

```
int *PREZ(int *v1, int *v2, int n, int pos) {
  int i = 0;
  int c1, c2;
  int *r;

  r = (int *) malloc(n * sizeof(int));

  c1 = c2 = (n-1)* pos;
  if(pos)
    for(i=n-1;i>=0;i--)
      r[i] = v1[c1] > v2[c2] ? v1[c1--] : v2[c2--];
  else
    for(i=0;i<n;i++)
      r[i] = v2[c2] > v1[c1] ? v1[c1++] : v2[c2++];

  return r;
}
```

IV. Algorithm Testing

A. Comparison Respect to the Sequential Quicksort

In order to check the efficiency of the proposed algorithm and its implementation by Message Passing, several tests have been realized. They have consisted of the execution of a program that implements the algorithm in MPI. The program was run for 2, 4 and 8 tasks, with arrays of 2^{15}, 2^{17}, 2^{19}, 2^{21} and 2^{23} elements ([1] We have choosen vector sizes and number of tasks of 2^n due to the restriction of the bitonic sort algorithm, which only works with vectors of 2^n elements). For each combination of N and n, we realized 50 runs of the program, with different arrays, so that the result of each test was the average value of the 50 executions.

The results shown here have been performed by a PC running under SUSE Linux 10.0, with LAM MPI. The values

Open Science Index, Electrical and Computer Engineering Vol:2, No:9, 2008 publications.waset.org/13070/pdf

are provided by the *clock ()* function, which returns an approximation of processor time used by the program. Because the values returned by this function for array sizes smaller than 2^{15} aren't significants, the cases of study begins at this size.

Table I and Graphic 1 show the values of speed-up [3], defined as:

$$S = \frac{C_s}{C_p}$$

where *S* is the speed-up, *Cs* the number of cycles of the sequential algorithm execution and *Cp* the cycles employed in the parallel algorithm.

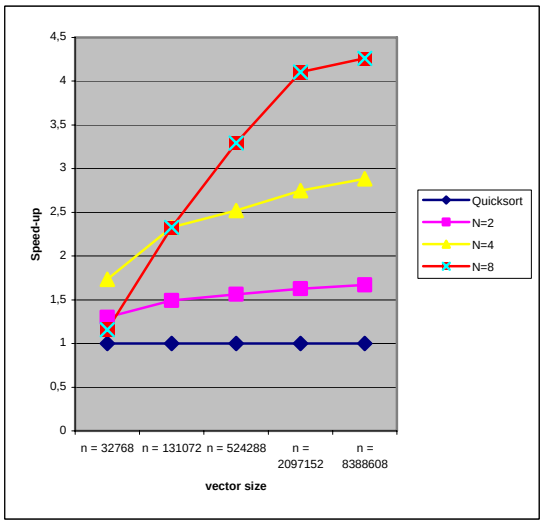
The percentage of gain, shown in Table II is defined as:

$$G = \left(\frac{S}{P} - 1\right) \cdot 100$$

being, *G* the percentage of gain, *S* the number of processor cycles for the sequential algorithm, and *P* the same for parallel execution.

TABLE I
SPEED-UP RESPECTING THE SEQUENTIAL QUICKSORT

	Qsort	PREZ N=2	PREZ N=4	PREZ N=8
n = 2^{15}	1	1,300	1,733	1,156
n = 2^{17}	1	1,491	2,327	2,327
n = 2^{19}	1	1,564	2,522	3,295
n = 2^{21}	1	1,628	2,749	4,106
n = 2^{23}	1	1,671	2,883	4,262



Graphic 1 Speed-up respecting the sequential quicksort

The best performance is achieved at greater values of *n*, which decrease when the number of tasks increases. Specially outstanding it is the result for N=8 with vector sizes under 2^{17}

The conclusion is that the algorithm obtains satisfactory results in speed-up, in relation to the sequential execution of the quicksort algorithm.

TABLE II
PERCENTAGE OF GAIN RESPECTING THE SEQUENTIAL QUICKSORT

	PREZ N=2	PREZ N=4	PREZ N=8
n = 2^{15}	30,000	73,333	15,556
n = 2^{17}	49,102	132,710	132,710
n = 2^{19}	56,379	152,212	229,480
n = 2^{21}	62,766	174,879	310,560
n = 2^{23}	67,099	188,339	326,229

B. Comparison Respect to the Arrangement in Parallel

A much more reliable way to determine the validity of our algorithm is to compare it with other methods of parallel arrangement. The chosen methods to contrast with PREZ have been quicksort and bitonic sort, both adapted to parallelism by message passing with MPI.

The test realized was the same as with PREZ: 50 executions for each N=2, 4 and 8, and each n= 2^{15}, 2^{17}, 2^{19}, 2^{21} and 2^{23}. The average values obtained are shown in Table III, expressed like numbers of cycles of execution in a scale of 10^5.

TABLE III
RESULTS FOR ALL THE TESTS REALIZED

N	*1*	*2*	*4*	*8*
PREZ				
n = 2^{15}		0,080	0,060	0,090
n = 2^{17}		0,334	0,214	0,214
n = 2^{19}		1,458	0,904	0,692
n = 2^{21}		6,306	3,734	2,500
n = 2^{23}		27,288	15,814	10,698
Bitonic Sort				
n = 2^{15}		0,308	0,198	0,138
n = 2^{17}		1,458	0,810	0,488
n = 2^{19}		6,884	3,592	1,966
n = 2^{21}		32,314	16,564	8,660
n = 2^{23}		150,914	76,936	39,882
Quicksort				
n = 2^{15}	0,104	0,116	0,124	0,166
n = 2^{17}	0,498	0,418	0,346	0,314
n = 2^{19}	2,280	1,788	1,422	1,210
n = 2^{21}	10,264	8,078	6,122	4,636
n = 2^{23}	45,598	35,936	27,376	19,850

1. Quicksort

Quicksort algorithm [5] does not need presentation, because it is possibly one of best known and most used sorting algorithms. Its function is based on the divide-and-win strategy, that is to say, it makes recursive partitions of the vector, using a value as pivot, where the values smaller o equals to the pivot go to one partition, and the values greater than the pivot go to the other one. The procedure is repeated with each one until getting partitions of 1 element. Then, the vector will be sorted.

The parallel implementation of quicksort may be, in some

Open Science Index, Electrical and Computer Engineering Vol:2, No:9, 2008 publications.waset.org/13070/pdf

cases, less efficient than the sequential version, as it is shown in Table V. The efficiency depends on the pivot chosen to do the partition. An inadequate pivot may cause, in most cases, an unbalanced workload.

Basically, the parallel algorithm realizes the first partition, then, it passes one of the subvectors to a dependent task. In the next iteration, both tasks will spawn a dependent process to which one of the new partitions of vectors passes. The process will continue until it is reached the maximum number of task, and after that, all of them will continue sequentially. When each task finishes its work, the result (the sorted subvector) returns to its parent-task recursively until the root task has the vector completely sorted.

Table IV shows the results obtained in tests of speed-up, for N=1 (sequential), 2, 4 and 8 tasks. Only for N=2, the values are acceptable. It can be seen that PREZ obtains better results in all the tests.

TABLE IV
SPEED-UP VALUES FOR QUICKSORT IN PARALLEL RESPECTING THE SEQUENTIAL ALGORITHM

	N=1	N=2	N=4	N=8
n = 2^{15}	1,000	0,897	0,839	0,627
n = 2^{17}	1,000	1,191	1,439	1,586
n = 2^{19}	1,000	1,275	1,603	1,884
n = 2^{21}	1,000	1,271	1,677	2,214
n = 2^{23}	1,000	1,269	1,666	2,297

Table V shows the gain values for 2, 4 and 8 tasks, where negative values can be seen for n=2^{15}, respect to the sequential quicksort.

TABLE V
GAIN VALUES FOR QUICKSORT IN PARALLEL RESPECTING TO THE SEQUENTIAL ALGORITHM

	N=2	N=4	N=8
n = 2^{15}	*-10,345*	*-16,129*	*-37,349*
n = 2^{17}	19,139	43,931	58,599
n = 2^{19}	27,517	60,338	88,430
n = 2^{21}	27,061	67,658	121,398
n = 2^{23}	26,887	66,562	129,713

2. Bitonic Sort

Bitonic sort [1] is a sorting network. Sorting networks are a special kind of sorting algorithms, where the sequence of comparisons is not data-dependent. This is why they are very suitable for implementation in hardware or parallel processor arrays.

Bitonic sort is based on the usage of bitonic sequences. A sequence a_1, a_2, ...a_k, a_{k+1}, ... a_n, is bitonic if it contains at least two changes of "tonic", i.e, initially increasing from a_1 to a_k, and decreasing from a_{k+1} to a_n, or vice versa.

The algorithm divides the vector into several bitonic sequences so as to recursively mix them two by two, making bitonic sequences of a double size in every iteration, until the entire vector is sorted.

To implement the algorithm on MPI in order to perform this test, it has been developed a procedure which splits the vector onto N subvectors. Each task will execute the bitonic sort; the odd tasks in ascending order and the even tasks in descending order, to get N/2 bitonic sequences. After this, the procedure will send the data to the corresponding task to continue the sorting. The explanation of the complete mechanism is too complex and it is not the aim of this article.

TABLE VI
SPEED-UP VALUES FOR BITONIC SORT IN PARALLEL RESPECTING THE SEQUENTIAL ALGORITHM

	Qsort	N=2	N=4	N=8
n = 2^{15}	1	-66,234	-47,475	-24,638
n = 2^{17}	1	-65,844	-38,519	2,049
n = 2^{19}	1	-66,880	-36,526	15,972
n = 2^{21}	1	-68,237	-38,034	18,522
n = 2^{23}	1	-69,785	-40,733	14,332

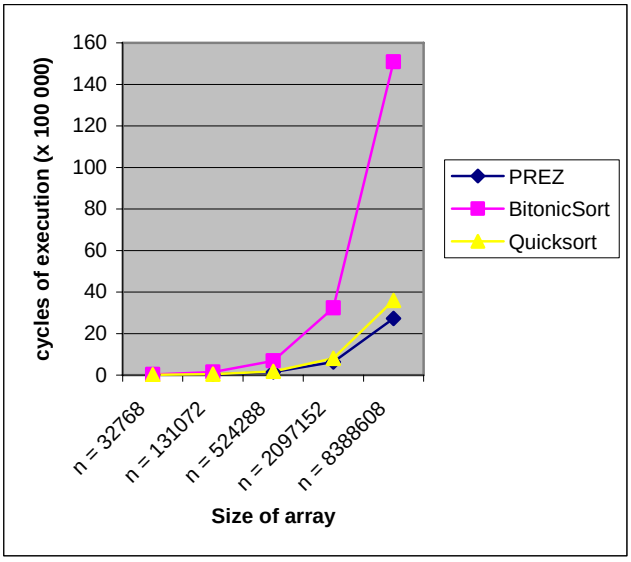
The results obtained in the tests (see Table VI for speed-up) are, in all cases, less efficient than PREZ.

3. Algorithms Comparisons

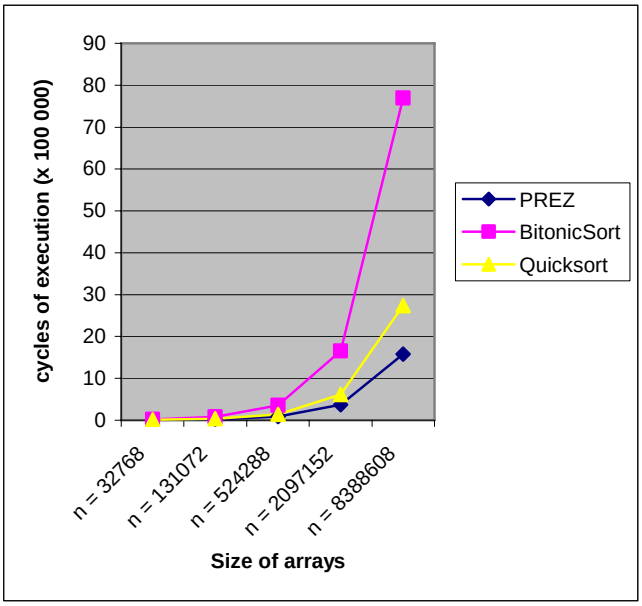
The following graphics are offered so as to compare the performance of PREZ respecting to the other algorithms. There are graphic representations of data on Table III:

- Graphic 2 shows the cycles of execution of three algorithms for 2 tasks.
- Graphic 3 shows the same data for 4 tasks, and
- Graphic 4 for 8 tasks.

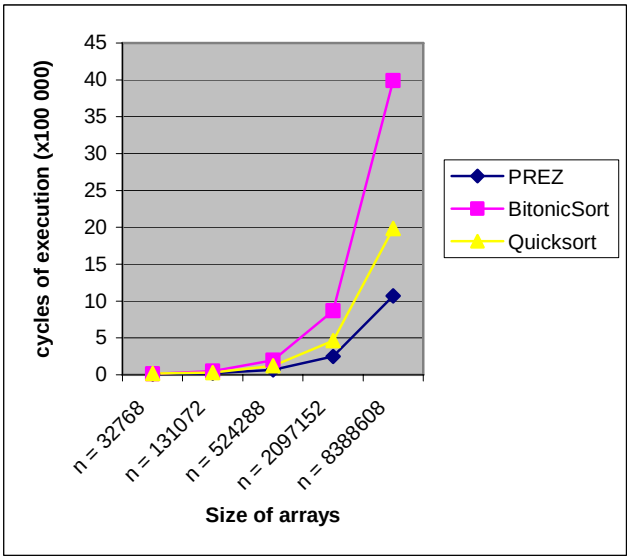
In all cases, PREZ lines indicate the best results, which are more significant as the number of elements to sort is growing.



Graphic 2 Cycles of execution for 2 tasks



Graphic 3 Cycles of execution for 4 tasks



Graphic 4 Cycles of execution for 8 tasks

V. Conclusion

The sorting algorithm presented here has proved to be powerful in terms of speed, in comparison with the others studied in this paper.

The algorithm presents a better performance with bigger arrays (2^{20} elements and more).

With respect to the Bitonic Sort, the improvement is really noteworthy, presenting notable differences in all cases.

In the results of the comparison with parallel quicksort, the contrast presents minor differences, but always better results for PREZ vs. quicksort. Once again, the advantage increased as the size of arrays did it.

Finally, our conclusion is that PREZ is a very interesting algorithm which presents good results on MPI, but it is necessary to realize more analysis, comparing it with other algorithms and, principally, executing the tests in a real distributed memory environment, in order to check how the transmission of data affects the performance.

References

[1] Batcher, K. E. "Sorting Networks an their Applications". Proc. AFIPS Spring Joint Comput. Conf., Vol. 32, 307-314, 1968.
[2] Cormen, T.H. et al. "Introduction to Algorithms". 2. Auflage, The MIT Press 2001.
[3] Culler, E., Singh, J.P. "Parallel Computer Architecture: A Hardware/Sotware approach". Morgan Kaufmann Publishers, Inc, San Francisco, 1999. ISBN 1-55860-343-3.
[4] Grama, A. Et al. "Introduction to Parallel Computing". Second Edition. Addison-Wesley 2003, SIBN 0-201-64865-2.
[5] Hoare, C. "Quicksort". Computer Journal, Vol. 5, 1, 10-15, 1962.
[6] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994.
[7] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org.
[8] Pratt, V. "*Shellsort and Sorting Networks*". Garland, New York, 1979.
[9] Sedgewick, R. "*Algorithms in Java*", Parts 1-4. 3. Auflage, Addison-Wesley, 2003
[10] Shell, D. L. "*A High-Speed Sorting Procedure*". Communications of the ACM, 2, 7, 30-32. 1959.

Open Science Index, Electrical and Computer Engineering Vol:2, No:9, 2008 publications.waset.org/13070/pdf