Parallel Rank Sort

Assist. Lecturer Felician ALECU
Economy Informatics Department, A.S.E. Bucharest

One of the fundamental problems of computer science is ordering a list of items. There are a lot of solutions for this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive but others are extremely complicated but produce lightening-fast results. The purpose of this paper is to present a parallel bubble sort algorithm that has a linear complexity, much better than the complexity level of the fastest known sequential sorting algorithm.
Keywords: *Parallel and sequential rank sort algorithm, parallel processing, efficiency, complexity level, big-O notation.*

Sorting is one of the most common operations performed by a computer. Basically, it is a permutation function which operates on *n* elements. Internal sorting can be used when the number of elements is small enough to fit into the main memory. If *n* is very large and it doesn't fit the main memory then auxiliary storage must be used in order to complete the sorting operation.
The sorting methods can be divided into two classes by the complexity of the algorithms used. The complexity of a sorting algorithm is generally written in the *big-O* notation and it is expressed based on the size of sets the algorithm is run against. The *big-O* notation represents a theoretical framework upon which we can compare two or more algorithms. The two classes of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection, shell, rank sorts and $O(n \log n)$, which includes the heap, merge, quick sorts.
For each element of the list to be sorted, the *Rank Sort* algorithm is computing the total number of elements that are lower than that number. This value is called the *rank* of the element and to compute it the algorithm needs to compare the element with all other values form the list. In a fully sorted list in increasing numerical order, the rank of each element will just be its actual position in the list. Finally, the algorithm uses the rank of each element to place it in its proper sorted position.
The following table (Tab. 1) shows the rank of each element form a list (unsorted and sorted).

Unsorted List	Rank	Sorted List	Rank
55	*3*	33	*1*
99	*6*	44	*2*
44	*2*	55	*3*
33	*1*	77	*4*
88	*5*	88	*5*
77	*4*	99	*6*

Tab. 1 – Rank of the elements

The sequential version of the *Rank Sort* algorithm is presented below (Alg. 1). The programming language used to describe the algorithm is *MultiPascal*, a parallel version of the classical *Pascal* language. The *MultiPascal* language was developed by *Bruce P. Lester*.
The *PutInPlace* procedure finds the rank of an element by looping through the values from the list. After the rank is computed, the procedure puts the element in its final sorted position.

```
procedure PutInPlace
(var x:vector;var y:vector;n:integer;pos:integer);
var
      i,rank:integer;
begin
      rank:=1;
      for i:=1 to n do
            if x[i]<x[pos] then
                  rank:=rank+1;
      y[rank]:=x[pos];
end;

procedure RankSort_Sequential
(var x:vector;var y:vector;n:integer);
var
      i:integer;
begin
      for i:=1 to n do
            PutInPlace(x,y,n,i);
end;
```

Alg. 1 – The sequential version of the *Rank Sort* algorithm

The sorting algorithm ends its execution after a number of *n* iterations of the main loop. At each step, the program performs a number of comparisons equal with *n*, the number of the elements from the list. Based on these results we can conclude that the complexity level of the algorithm for a common array is $O(n^2)$, no matter of the initial existing order of the elements from the list. The algorithm doesn't need any kind of exchanges because the final position of an element is computed based on its rank.

The parallel version of the *Rank Sort* algorithm can be easily obtained by computing the rank of each element from the list independently on a different processor. Processor number 1 can compute the rank of the first element by comparing it with every other element in the list. In the same time, processor number 2 can simultaneously compute the rank of the 2nd element from list and so on. If there are *n* processors in the parallel system, each processor *i* can be assigned to compute the rank of the element number *i* in the list.

Also, the parallel *Rank Sort* algorithm is using two arrays (unsorted and sorted lists) stored in the shared memory area. In such a way the lists will be available to all the processors from the system in the same time (Fig. 1).

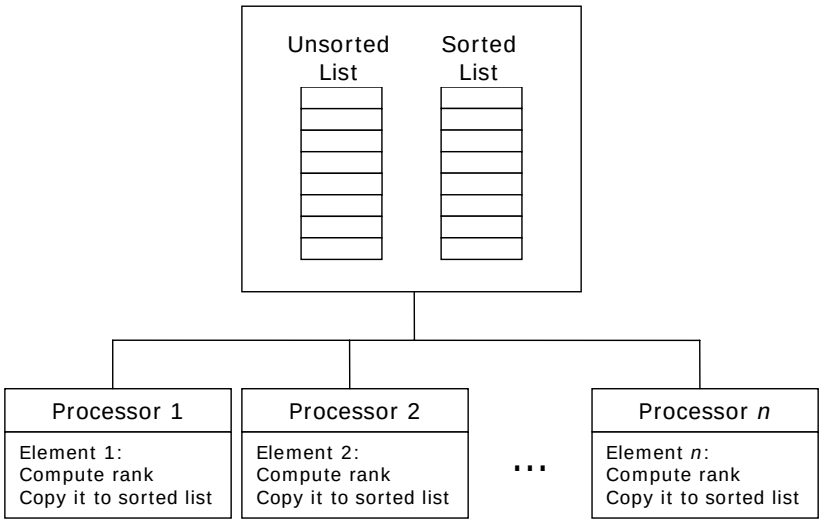

Fig. 1 – Parallel rank sort

The parallel version of the *Rank Sort* algorithm is presented below (Alg. 2).

```
procedure PutInPlace
(var x:vector;var y:vector;n:integer;pos:integer);
var
      i,rank:integer;
begin
      rank:=1;
      for i:=1 to n do
            if x[i]<x[pos] then
                  rank:=rank+1;
      y[rank]:=x[pos];
end;

procedure RankSort_Parallel
(var x:vector;var y:vector;n:integer);
var
      i:integer;
begin
      forall i:=1 to n do
            PutInPlace(x,y,n,i);
end;
```

Alg. 2 – The parallel version of the *Rank Sort* algorithm

The *PutInPlace* procedure is executed in parallel for *n* times and it has a complexity level of *O(n)*. Because the outer loop iterations are executed in parallel, the complexity level of the parallel version of the *Rank Sort* algorithm will become equal with *O(n)*, where *n* represents the number of the elements in the list. This is superior to the fastest known sequential sorting algorithms, which are all *O(n logn)*.

If the number *p* of processors in the system is less than the number of list elements *n*, the total execution time become $O(n^2/p)$ and it reduces to *O(n)* when $p = n$.

The values of the most important parameters measuring the performance of a parallel program (S – speedup, E – efficiency, Cp – parallel cost, Csupl – supplementary cost of parallel execution) are listed below:

$$S = \frac{T_s}{T_p} = \frac{O(n \log n)}{O(n)} = O(\log n)$$

$$E = \frac{S}{p} = \frac{O(\log n)}{O(n)} = O\left(\frac{\log n}{n}\right)$$

$$C_p = p \cdot T_p = O(n) \cdot O(n) = O(n^2)$$

$$C_{\sup l} = C_p - C_s = p \cdot T_p - T_s = O(n^2) - O(n \log n) = O(n^2)$$

Analyzing these parameters we can conclude the following: the sequential version of the *Rank Sort* algorithm is very useful because its simplicity, not its efficiency. The parallel version remains simple but it is also very fast, having a complexity level equal with *O(n)*.

References

1. A. Inselberg, *Parallel Coordinates*, Springer, 2004
2. R. Wyrzykowski, *Parallel Processing And Applied Mathematics*, Springer, 2004
3. J. Joseph, C. Fellenstein, *Grid Computing*, Prentice Hall, 2003
4. Gh. Dodescu, B. Oancea, M. Răceanu, *Parallel Processing*, Economic Publishing House, Bucharest, 2002
5. H. F. Jordan, H. E. Jordan, *Fundamentals of Parallel Computing*, Prentice Hall, 2002
6. R. Sedgewick, *Algorithms*, Addison-Wesley, 1998
7. G. W. Sabot, *High Performance Computing*, Addison-Wesley, 1995