Parallel Bubble Sort

Assistant Lecturer Felician ALECU
Economic Informatics Department, A.S.E. Bucharest

Abstract

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, such as the bubble sort, but others, like quick sort, are extremely complicated but produce lightening-fast results.

The sequential version of the bubble sort algorithm is considered to be the most inefficient sorting method in common usage. In this paper we want to prove that the parallel bubble sort algorithm has a linear complexity, much better than the complexity level of the fastest known sequential sorting algorithm.

Keywords

Sorting algorithms, bubble sort, parallel processing, complexity level, big *O* notation, efficiency, odd-even transposition.

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 sorts and $O(n \log n)$, which includes the heap, merge, quick sorts.

The sequential version of the bubble sort is the oldest, the simplest and the slowest sorting algorithm in use having a complexity level of $O(n^2)$. This is why the method is considered to be the most inefficient sorting algorithm in common usage.

The bubble sort works by comparing each item in the list with the item next to it and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. Such a situation means that all items are in the correct order. This causes larger values to be moved (*bubbled*) to the end of the list while smaller values remain towards the beginning of the list.

The source code of the sequential bubble sort algorithm can be found 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 sorting algorithm ends its execution after a number of maximum *n* iterations of the main loop. The number of comparisons will be *n-1* and the number of swaps can be approximated by *(n-1)/2*. The minimum number of iterations is *1*, in the case when the elements of the array are already sorted. In this case the algorithm will perform *n-1* comparisons and *0* swaps. Based on these results we can conclude that the complexity level of the algorithm for a common array is $O(n^2)$.

```
procedure BubbleSort_Sequential(var x:vector;n:integer);
var
     i:integer;
     sort_ok:boolean;
begin
     repeat
           sort_ok:=true;
           for i:=1 to n-1 do
                if x[i]>x[i+1] then
                begin
                     Exchange(x[i],x[i+1]);
                     sort_ok:=false;
                end;
      until sort_ok;
end;
```

Alg. 1 – The sequential version of bubble sort algorithm

The source text of the *Exchange* routine is the following (Alg. 2). Using a temporary location, the procedure swaps two elements of the array given as parameters.

```
procedure Exchange(var x:integer;var y:integer);
var
     temp:integer;
begin
     temp:=x;
     x:=y;
     y:=temp;
end;
```

Alg. 2 – The *Exchange* procedure

An improved version of the bubble sort algorithm can be obtained if the *repeat* loop is replaced by a *for* one. The sorting procedure can be speeded up if we notice that at the end of one loop, a part of the array, starting with the 2nd item from the last modified pair, is already sorted. This is why we do not need to analyze again all these elements in the next loops. Basically, at the first loop the maximum element is moved on the first position, at the 2nd loop the maximum element from the remaining array will be swapped to its final position and so on. After maximum *n-1* iterations the array will be fully sorted.

The improved version of the sequential bubble sort algorithm can be found below (Alg. 3). For a common array, this version of the algorithm will perform *n-1* iterations of the outer loop. At every iteration, the number of comparisons will be equal with *n-i*, where *i* represent the index of the current iteration. The total number of comparisons can be computed using the following formula: $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$. This is why we can conclude that the complexity level of this version of bubble sort method is $O(n^2)$ too but the algorithm is running faster than the previous one..

```
procedure BubbleSort_ImprovedSeq(var x:vector;n:integer);
var
     i,j:integer;
begin
      for i:=1 to n-1 do
           for j:=1 to n-i do
                if x[j]>x[j+1] then
                     Exchange(x[j],x[j+1]);
end;
```

Alg. 3 – Improved sequential bubble sort algorithm

It is not very easy to transform this algorithm in a parallel one because it compares pairs of consecutive elements. The parallel version can be obtained if we use the *odd-even transposition method* (Alg. 4) that implies the existence of *n* phases, each requiring *n/2* compare and exchanges. In the first phase, called *odd phase*, the elements having odd indexes are compared with the neighbors from the right and the values are swapped when necessary. In the *even phase*, the elements having even indexes are compared with the elements from the right and the exchanges are performed only if necessary.

```
procedure OddEvenTranspositionSort_Parallel
                                          (var x:vector;n:integer);
var
     i,j:integer;
begin
     for i:=1 to n do
           if odd(i) then
           begin
                forall j:=1 to n div 2 do
                     if x[2*j-1]>x[2*j] then
                          Exchange(x[2*j-1],x[2*j]);
           end
           else
           begin
                forall j:=1 to (n div 2)+(n mod 2)-1 do
                     if x[2*j]>x[2*j+1] then
                          Exchange(x[2*j],x[2*j+1]);
           end;
end;
```

Alg. 4 – Parallel version of the odd-even transposition method

The array will be fully sorted after maximum *n* phases (*n/2* iterations), where *n* represents the number of elements. The inner loops iterations will be performed in parallel on the processors available in the system.

In order to adapt the odd-even transposition method to our bubble sort algorithm, we can put both phases, odd and even, inside the main loop by using a boolean flag to indicate that the sorting operation is complete. The source code of the parallel bubble sort algorithm can be found below (Alg. 5).

```
procedure BubbleSort_Parallel(var x:vector;n:integer);
var
     i:integer;
     temp:integer;
     sort_ok:boolean;
begin
     repeat
          sort_ok:=true;
          forall i:=1 to (n div 2) do
               if x[2*i-1]>x[2*i] then
               begin
                    Exchange(x[2*i-1],x[2*i]);
                    sort_ok:=false;
               end;
          forall i:=1 to (n div 2)+(n mod 2)-1 do
               if x[2*i]>x[2*i+1] then
               begin
                    Exchange(x[2*i],x[2*i+1]);
                    sort_ok:=false;
               end;
     until sort_ok;
end;
```

Alg. 5 – The parallel version of the bubble sort algorithm

For a common array, the parallel version of bubble sort will perform *n* iterations of the main loop and for each iteration a number of *n-1* comparisons and *(n-1)/2* exchanges will be performed in parallel.

If the number of processors is lower than *n/2*, every processor will execute *(n/2)/p* comparisons for each inner loop. In this case the complexity level of the algorithm will be $O(n^2/2p)$.

If the parallel system has a number of processors *p* higher than *n/2*, the complexity level of the inner loops is *O(1)* because all the iterations are performed in parallel. The outer loop will be executed for maximum *n* times so we can conclude that the complexity level of the algorithm is equal with *O(n)*, much better than *O(n log n)*, the complexity of the fastest known sequential sorting algorithm.

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