Parallel Merging

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

The main goal of this paper is to present a highly parallel program used to merge any two sorted lists into a single one ordered by the same rule. The program has to be as efficient as possible in order to maximize the speedup and to minimize the parallel execution time.
Keywords: *parallel algorithms, merging, sorting, binary search, parallelization techniques.*

Basically, a merging algorithm takes two sorted sequences as inputs and combines them into a single sorted sequence. Such an algorithm generally runs over multiple sorted lists in a time proportional to the sum of the lengths of the lists.
The classical merging algorithm outputs, at each step, the data item having the lowest key. Given some sorted lists, it produces a sorted list containing all the elements in any of the input lists, and it does so in a time proportional to the sum of the lengths of the input lists. We can say with no doubt that the time complexity of the sequential version of the merging algorithm is equal with *O(n)*. The Pascal source code of this version of sequential merging algorithm is presented below (Alg.1).

```
procedure SequentialMerging
(var x:int_array;nx:integer;
 var y:int_array;ny:integer;
 var z:int_array);
var     crt_pos_x:integer;
        crt_pos_y:integer;
        crt_pos_z:integer;
        i:integer;
begin   crt_pos_x:=1;
        crt_pos_y:=1;
        crt_pos_z:=1;
        while (crt_pos_x<=nx) and (crt_pos_y<=ny) do
        begin  if x[crt_pos_x]<y[crt_pos_y] then
                begin   z[crt_pos_z]:=x[crt_pos_x];
                        crt_pos_x:=crt_pos_x+1;
                        crt_pos_z:=crt_pos_z+1;
                end
                else
                begin   z[crt_pos_z]:=y[crt_pos_y];
                        crt_pos_y:=crt_pos_y+1;
                        crt_pos_z:=crt_pos_z+1;
                end;
        end;
        if crt_pos_x<=nx then
        begin  for i:=crt_pos_x to nx do
                begin   z[crt_pos_z]:=x[i];
                        crt_pos_z:=crt_pos_z+1;
                end;
        end
        else
        begin   for i:=crt_pos_y to ny do
                begin   z[crt_pos_z]:=y[i];
                        crt_pos_z:=crt_pos_z+1;
                end;
        end;
end;
```

Alg.1. Sequential merging

Analyzing the algorithm we can observe that merging sorted lists is a very sequential activity with little opportunities for parallelism. The algorithm compares pairs of two elements, the minimum value is written in the final array and the corresponding indexes are incremented. There is no room for parallelism and concurrency.
As like in the *Rank Sort* algorithm, we can notice that the final position of an element

X[i] can be directly computed based on its *rank* that can be obtained using the current position *i* and the corresponding position *j* in the second list *Y*. In such a way, the final position of the element will be *Z[i+j-1]*.
This observation can be easily justified in the following way: the first *i-1* items from the list *X* are less than *X[i]* and must appear before this element in the final sorted list *Z*. Also, the first *j-1* items from the second list are less than *X[i]* and must also appear before our element in the final list. So, by total, there are *i-1+j-1* items that must appear prior to *X[i]* in the *Z* list. This is why the rank of the element *X[i]* will be equal with *i+j-1* and the element will be placed directly into position *Z[i+j-1]*. The same observation can be applied over the elements of the second list, *Y*. At the end, for every item from the initial lists we will obtain the rank that will be used to put in place the element at the right position.
Due to the fact the lists are already sorted, the binary search can be used to compute the corresponding position of the element in the other list.
We still have an opened issue regarding the duplicate items in the two lists. Such elements will produce the same value of rank and therefore the final list will not be complete. To avoid such a situation we need to create an asymmetry in the way in which the elements of the two lists are processed. When performing the binary search of list *Y* to find the location of *X[i]*, all the elements that are equal with *X[i]* will be treated as if they were greater. In the same way, when the binary search of list *X* for *Y[k]* is performed, the elements equal with *Y[k]* will be managed as if they were lower. This asymmetry will prevent the collision between the equal items from the two initial lists.
The source code of the procedure that performs the rank computation of a given element using the binary search is presented below (Alg.2). The *PutInPlace* routine has a complexity that is equal with the complexity of binary search operation, *O(log n)*. The *source* parameter is used to generate the asymmetry.

```
procedure PutInPlace
(value:integer;pos:integer;
 x:int_array;n:integer;
 var z:int_array;source:integer);
var     start_pos:integer;
        end_pos:integer;
        crt_pos:integer;
begin   start_pos:=1;
        end_pos:=n;

        while (end_pos-start_pos>1) do
        begin   crt_pos:=(start_pos+end_pos) div 2;
                if value<x[crt_pos] then end_pos:=crt_pos
                else if value>x[crt_pos] then start_pos:=crt_pos
                     else if source=1 then start_pos:=crt_pos
                          else end_pos:=crt_pos;
        end;
        if value>x[end_pos] then crt_pos:=end_pos+1
        else if value<x[start_pos] then crt_pos:=start_pos
             else crt_pos:=end_pos;
        z[pos+crt_pos-1]:=value;
end;
```

Alg.2. The *PutInPlace* procedure

The sequential version of the merging algorithm based on rank computation is presented below (Alg.3). The *PutInPlace* procedure is executed repeatedly by *n* times, where *n* is the sum of the lengths of the input lists. This is why the time complexity of this algorithm is equal with *O(n log n)*, higher than the value obtained for the previous version.
This rank based merging algorithm can be easier parallelized because the rank of elements can be concurrently computed on the processors from a parallel configuration. Every processor will calculate independently one or more ranks, depending on the number of processors from the parallel system.
The parallel version of the merging algorithm is presented below (Alg.4). The source code was written using the Multi-Pascal language.

for a multiprocessor parallel architecture

```
procedure SequentialMerging
(x:int_array;nx:integer;
 y:int_array;ny:integer;
 var z:int_array);
var    i:integer;
begin  for i:=1 to nx do
             PutInPlace(x[i],i,y,ny,z,1);
       for i:=1 to ny do
             PutInPlace(y[i],i,x,nx,z,2);
end;
```

Alg.3. Sequential merging based on rank computation

This version is obtained by running in parallel the successive calls of the *PutInPlace* procedure. Due to the fact that all the loops are executed in parallel, the time complexity of the algorithm is equal with *O(log n)*, much better compared with the sequential version.

```
procedure Merging_Parallel
(x:int_array;nx:integer;
 y:int_array;ny:integer;
 var z:int_array);
var    i:integer;
begin  forall i:=1 to nx do
             PutInPlace(x[i],i,y,ny,z,1);
       forall i:=1 to ny do
             PutInPlace(y[i],i,x,nx,z,2);
end;
```

Alg.4. Sequential merging

The main reason of parallelization a sequential program is to run the program faster. The first criterion to be considered when evaluating the performance of a parallel program is the speedup used to express how many times the parallel program runs faster than the sequential one, where both programs are solving the same problem. If the parallel program is executed on a computer with *p* processors, the highest value that can be obtained for the speedup is equal with the number of processors from the system. The maximum speedup value could be achieved in an ideal multiprocessor system where there are no communication costs and the workload of processors is balanced.
If the T_s is the execution time of the sequential merging algorithm and T_p is the running time of the parallel version, the *speedup* can be computed according with the following formula:

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

The system *efficiency* can be calculated by dividing the speedup value to the number of the processors from the system:

$$E = \frac{S}{p} = \frac{O\left(\frac{n}{\log n}\right)}{O(n)} = O\left(\frac{1}{\log n}\right)$$

The *sequential cost* is equal with the T_s. The *parallel cost* represents the total time consumed by all the processors from the system to solve the problem:

$$C_p = p \cdot T_p = O(n) \cdot O(\log n) = O(n \log n)$$

The sum of the times dedicated to communication and synchronization activities is called *supplementary cost* and its value can be obtained using the following formula:

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

Bibliography

[Wyr04] R. Wyrzykowski, *Parallel Processing and Applied Mathematics*, Springer, 2004
[Lad04] S. Ladd, *Guide to Parallel Programming*, Springer-Verlag, 2004
[Dod02] Gh. Dodescu, B. Oancea, M. Răceanu, *Procesare paralelă*, Editura Economică, Bucureşti, 2002
[Sed98] R. Sedgewick, *Algorithms*, Addison-Wesley, 1998