Chapter 1

SORTING

1.1 Sorting Generals

In computer programs sorting is one of the most frequently used algorithms. Sorted data are easier to maintain and reduces the computation need in many cases. The best examples are the database applications, where the so called 'indexes' are applicated to keep the sorted order of the database records. It is a very famous topic in algorithm theory because its formulation is very simple and it still leads to deep results. Many techniques, algorithms, theoretical results were devised in the last few decades. In the parallel algorithm's point of view sorting is one of the best problems for understanding how the data and work should be divided among processing units to get optimum performance.

Definition

Generally, the problem is to arrange an unordered collection of elements in order to get a monotonically increasing/decreasing collection. To be more formal, let

$$S = (a_1, a_2, a_3, \ . \ . \ ., a_n)$$

be a sequence of n elements in arbitrary order. The sorting operation transforms S into a monotonically increasing sequence

$$S' = (a'_1, a'_2, a'_3, \ldots, a'_n)$$

where $a'_i \leq a'_j$, $1 \leq i \leq j \leq n$ and S' is a permutation of S.

We can categorize the sorting based upon several attributes:

- **internal/external** This attribute reflects to the size of data compared to the applicable memory size. When the data is small enough to fit into the memory then we use an *internal* sorting, otherwise some external devices (e.g. hard disk) must be used to maintain the data. In this case the *external* sorting algorithm has to maintain reading/writing of the external devices which causes extra algorithmical problems. For the case of simplicity we focus our attention to the internal sorting algorithms.

- **comparison-based/non comparison-based** The *comparison-based* algorithm selects **two** elements and calls a comparison operation. When they are in the wrong order it exchanges them. The algorithm repeats this operation until the sequence is completely sorted. In contrast the *non comparison-based* algorithm selects only **one** element and based on the elements properties (such as binary representation) puts it into the right place in the sorted sequence.

  The computation complexity is different because of the extra sorting information what non comparison-based algorithms use:

  Computational complexity

  - comparison based: $O(n \cdot \log n)$
  - non comparison based: $O(n)$

  The first type is more general while the latter is applicable only when the sorted elements has some special properties. We treat only the comparison based algorithms here.

Concerning the input of the sorting there are two possibilities: either the input sequence is stored in a single processor's memory or the elements of the sequence is distributed among the processors.

In most of the cases the input of the sorting algorithm, (i.e. the sequence of the elements to be sorted) are distributed over the processors. The reason is that the sorting frequently appears inside a more complex parallel algorithm and the elements are created by different processors. The same happens after the sorting: the output, the sorted sequence is the input of another parallel algorithm, which also has to handle a distributed sequence.

The simplest case is if we have only one element per processor, but generally the sorting has to maintan much more element than the number of the processor. In this case we store more elements per processor.

The output will be a sorted sequence which is also distributed among the processors. Practically, we arrange the processors, and each processor stores the elements in sorted order. The sorted sequence will contain first the elements of the processor P_1 $a_1, a_2, ...$ then the elements of the processor P_2 and so on.

To enumerate the processor we can use any of the known algorithms, but the enumeration can have an effect on the performance of the algorithm. For example, in the case of ring architecture an enumeration which follows the links yields a better performance.

The key step in the sorting is the comparison of elements. If the elements appear on the same processor the operation simply compares and exchanges them if needed. If the elements stored on different processors they have to communicate.

1.1.1 Compare and exchange operation

Let we suppose that $i \leq j$, a_i element is stored by the P_i processor and the a_j element stored by the P_j processor. (See Fig. 0.1.) In he first step both processor sends its elements to the other processor. In the second step both processor compares it's own and the foreign element and the former stores the minimal, the latter stores the maximal element.

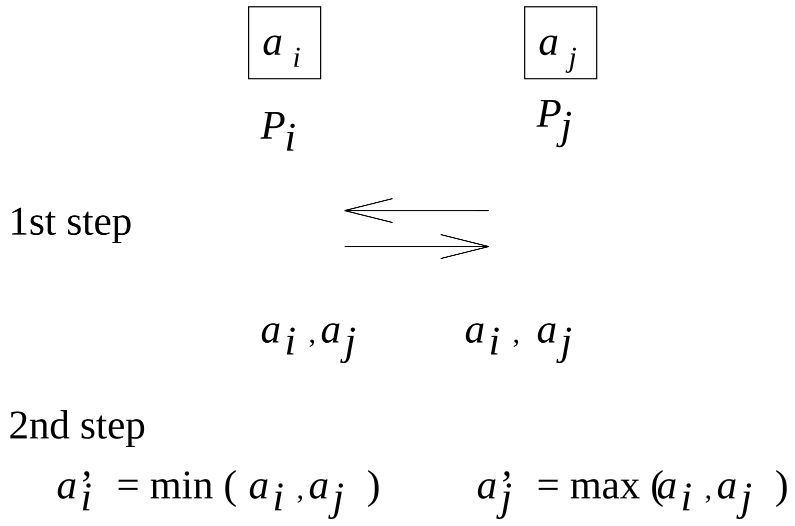

Figure 1.1:

1.1.2 Performance considerations

Let we denote the time elapsed for mesage startup by t_s the per word transfer time by t_w and the time for comparison by t_c.

If we have only one element per processor then every coparison operation between the neighboring processors requires $(t_s + t_w)$ time for communication and t_c for comparison. In the practice the t_s time is much larger than the t_w or t_c. The time elapsed by the sorting can be estimated by the number of comparisons times t_s. We can conclude now that the one element per processor scheme leads to a very inefficient usage of the multiprocessor architecture while most of the time is spent to the interprocess communication.

If we have more elements per processor we use the modified version of the compare and exchange operation. (See Fig. 0.2.)

Let we denote the number of processors by p and the set of elements stored on P_i by $A_i = \{a_{i1}, a_{i2}, ...\}$. We denote $A_i \leq A_j$ if every element in A_i is smaller than every element in A_j.

We have a properly sorted set of all elements when $A_1' \leq A_2' \leq \ \ldots \leq$

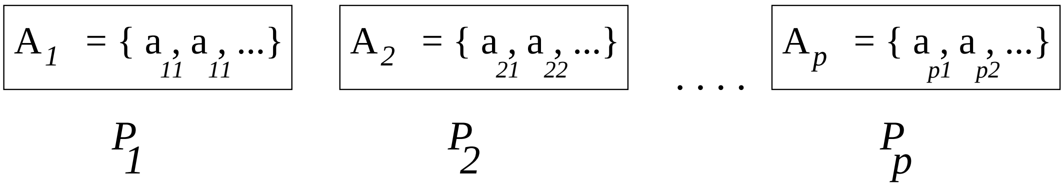

Figure 1.2:

A'_p and A'_i is sorted. The elements having the same processor can be easily sorted by one of the sequential sorting algorithms but to sort all of them we have to have a so called *compare and split* step described as follows. (See Fig. 0.3.) In the first step each processor sends its set (A_i, A_j) to the other processor. In the second step the processors perform merging of the sets. The resultant sets, $B_{j,i}$ and $B_{j,i}$ are sorted while the original sets were sorted. In the third step the processors take the lower or the upper half of the merged sets respectively. It is easy to see that for the new sets $A'_i \leq A'_j$.

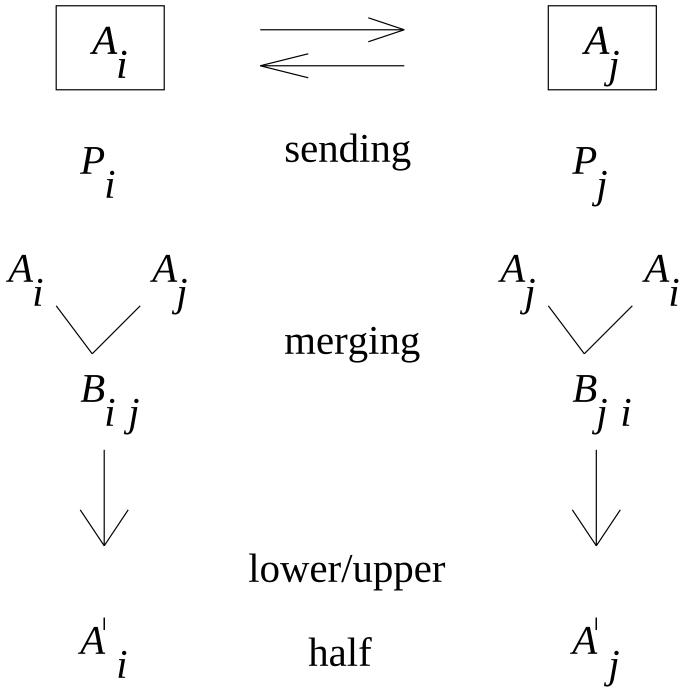

Figure 1.3:

1.2 Sorting Networks

As the first possibility to make sorting parallel we describe the sorting networks. Using a large number of *comparators* the sorting time is significantly smaller than $O(nlogn)$. These sorting networks are based on a comparison network model, in which many comparison operations are performed simultaneously.

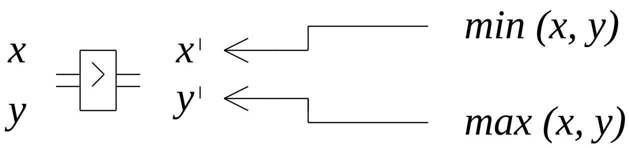

Figure 1.4:

A comparator (see Fig. 0.4) is a device with two inputs x and y and two outputs x' and y'.

There are two types of it:

- increasing comparator, $x' = minx, y$ and $y' = maxx, y$, we denote it by $\oplus$;
- decreasing comparator $x' = maxx, y$ and $y' = minx, y$, we denote it by $\ominus$;

As the two elements enter the input wires of the comparator, they are compared and if necessary exchanged before they go to the output wires. The comparators arrange in several columns, see Fig. ref5. The comparators having the same columns executed at the same time and – by the help of the connections between the columns – fill the comparator's input of the next column. Finally, the last column is filled with the elements which are properly sorted. The time elapsed by the sorting is proportional of the number of the columns which is called *depth*. In the next chapter we will see how the interconnections between the columns are arranged. The Fig. 0.6 shows a typical sorting network.

1.3 Bitonic Sort

The bitonic sorts is one application of the sorting network. The name is coming from the *bitonic sequence* which the key element of the algorithm.

A bitonic sequence is a sequence of elements $(a_0, a_1, a_2, \ldots, a_{n-1})$ with the property that either exists an $1 \leq i \leq n-1$ index:

$(a_0, a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_{n-1})$

$\leq \leq \leq \ \ldots \leq \Uparrow \geq \geq \ \ldots \geq$

such that $(a_0, a_1, a_2, \ldots, a_i)$ is monotonically increasing and $(a_{i+1}, \ldots, a_{n-1})$ monotonically decreasing; or, exists a circular shift