

Lecture #16: Parallel Sorting-I

Professor: S. Lennart Johnsson

TA: Wei Ding

1 Sorting

Sorting is one of the most common applications on computers. It is not only used in commercial applications, such as those in the banking and insurance industries, but also in science applications.

Sorting has found a new use in parallel computing. Routing messages from their sources to destinations can be made by sorting the messages by their destination. Though this idea is feasible, it is often less efficient than dedicated routing algorithms.

Another application of sorting for parallel computation is load-balancing. For instance, consider a collection of particles distributed in arbitrary ways in a three-dimensional space. With the work per particle approximately the same, one approach would be to simply divide the particles evenly among the processing nodes. Though this approach yields load-balance, it may result in excessive communication since there is no guarantee for locality of reference. Depending upon how the particles are ordered, particles assigned to the same processing node may be far apart, while particles close in physical space may have been assigned to nodes that are far apart. A simple solution is to sort the particles by their coordinates. In one dimension, the particle coordinates can be used directly to form the keys for sorting. But, in two or more dimensions there are many ways in which the keys can be constructed from the coordinates. First, with coordinates represented as floating-point numbers they can be converted into integers for each coordinate. Then, for say a three-dimensional problem, we do *not* want to order particles by forming the key simply concatenating the integers for a particles x , y and z coordinates. In this ordering, particles close with respect to the coordinate chosen as the least significant part of the key will be close also in the sorted order, while particles close with respect to the coordinate chosen as the most significant may be far apart in the sorted order. Instead, what is often made in practice is forming keys by interleaving the bits from the integers representing the coordinates of a particle. Such an ordering is known as Morton ordering [6]. Once the particles are sorted in a suitable ordering, load-balancing can be accomplished, for example, through the use of *orthogonal coordinate bisection*. Another popular load-balancing technique relies on *recursive spectral bisection*, which employs sorting of components of eigenvectors for partitioning of grids.

1.1 1-D Arrays

We will first consider sorting on multi-processor systems connected as linear arrays, then on systems with mesh interconnect and later other forms of interconnect. Figure 1 shows a linear

array of nodes with an ability to accept and deliver data at the ends of the array.

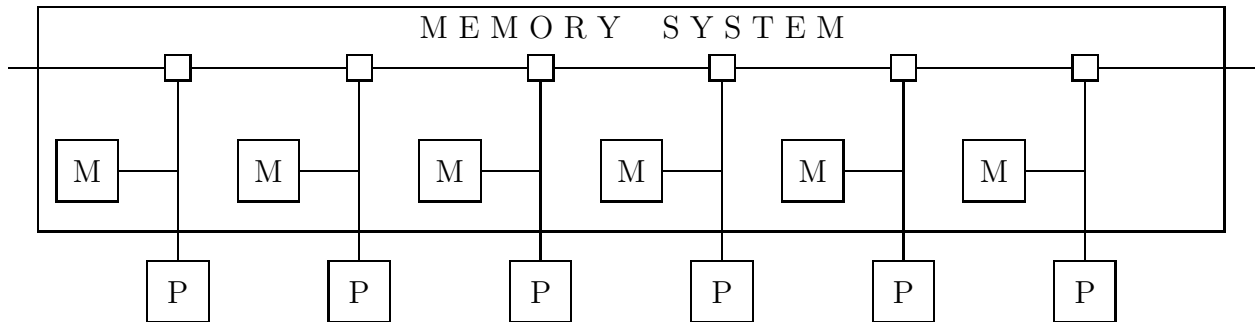


Figure 1: The memory system for distributed memory architectures.

For simplicity, we will draw the linear arrays as shown in Figure 2, which shows one linear array without wraparound. A linear array with wraparound connection, i.e., a ring, can be created by adding the dashed connection.

The data distribution has a profound impact on the choice of algorithms and performance, which can be expected given what we know of the performance for banked and interleaved memory systems.

In a linear array of nodes, each with its own local memory, one possibility is that data can enter and leave only at the ends of the array. In this scenario, no computation can finish in less time than it takes to enter the data and output the result. Another scenario is a linear array in which each node can accept and output data from outside the array. In this case, the data input and output times may be reduced significantly at the expense of a considerably increased input/output system. Its capacity is now required to be proportional to the number of nodes N instead of a constant.

We will first consider sorting with data entering and leaving at the ends of the linear array, then consider sorting with data already distributed within the array.

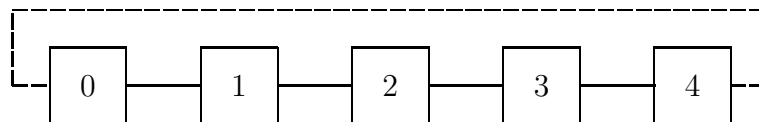


Figure 2: A linear array and a ring.

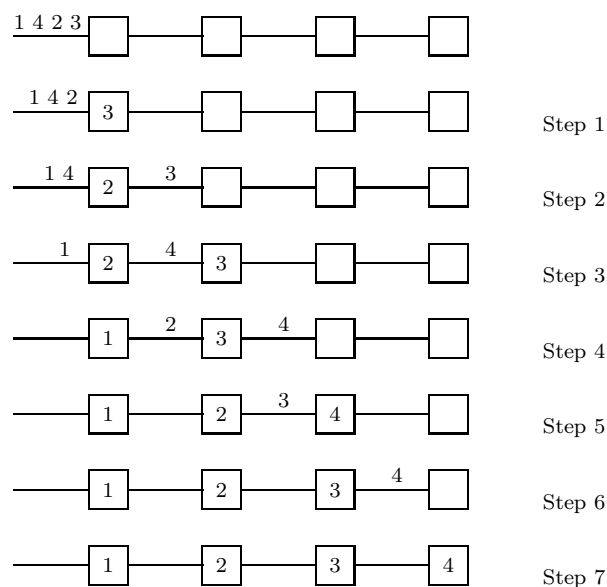


Figure 3: The first phase of the up-down sorting algorithm

1.1.1 Sorting with data external to the array

With a linear array of nodes with input and output restricted to the end points, it is clear that the time to sort N numbers on N nodes is at best $O(N)$. The numbers can simply not be accepted or delivered in a time less than what it takes to read and deliver the numbers one at a time.

A simple algorithm is shown in Figure 3. Each node executes the following algorithm:

1. Phase One: Enter and sort values
 - (a) Accept a value from the left neighbor,
 - (b) Compare the input value with the value it retained from the previous comparison,
 - (c) Output the larger value to the right neighbor, and store the smaller value.
2. Phase Two: Output sorted values
 - (a) Output the stored value to the left neighbor,
 - (b) Accept and store a value received from the right neighbor.

In the first phase, the N input values are received and sorted. In the second phase, the sorted values are output. Figure 3 shows the first phase for 4 numbers.

Our description of the algorithm ignores several important practical details. For instance, since in each step of the first phase a node compares the value it receives with the value it retained

from the previous comparison, initializing the local data must be done such that it does not interfere with the desired result. Second, a processor must also either be busy waiting for inputs, or be started somehow through some global control signal. A processor must also be able to know when it has completed Phase One. This can be accomplished through an end of sequence symbol, through a global completion signal, or through a local counter if it is known how many numbers shall be sorted. Finally, a processor must also be able to determine when Phase Two is over, either through a marker, global signal, or through counting, or by some other means.

The sorting time for the linear array is $2N$ steps, where each step must be sufficiently long for a node to receive data, send data, and perform a comparison for Phase One. A step in Phase Two only requires data to be sent and received.

If the linear array has input/output capabilities at both ends, it is possible to sort in both directions. One input stream use the left port, the other input stream use the right port. By skewing the operations in time by N steps, no additional communication or processing demand is placed on the nodes.

A good sequential sorting algorithm has a worst case sorting time of $O(N \log_2 N)$. Thus, for sorting a single sequence we have

$$\textit{Speedup: } S_N = O(N \log_2 N) / 2N = O(\log_2 N),$$

and

$$\textit{Efficiency: } E_N = O\left(\frac{\log_2 N}{N}\right),$$

which is very poor for large N . For instance, for $N = 1024$, $\frac{\log_2 N}{N} = \frac{10}{1024} \approx 1\%$. Sorting two sequences only improves the efficiency, and speedup, by a factor of two. Thus, a linear array is not a very efficient processor arrangement for sorting data. If the data is arriving sequentially, then the time cannot be reduced much, but other processor arrangements and algorithms may result in less resources with no increase, or only a modest increase in the sorting time.

Correctness of the sorting algorithm is easy to prove. Each node compares the number it stores and with the number it receives from the left and keeps the smaller. Thus, for each node numbers to the right of the node are at least as large as the number it keeps. The rule to keep the smaller number results in the numbers being sorted in a nondescending order.

Using a linear array to sort a sequence of elements of the same size is realistic only for small to moderate size sequences. In most cases a parallel computer has fewer nodes than there are elements to be considered. One approach in such a case is that each node emulates a number of nodes of a *virtual array* with as many nodes as there are elements of the data array. Thus, assume that the data array has R elements, while the physical array has N nodes, $N < R$. The situation is shown in Figure 4.

Each physical node must have sufficient memory to store all values associated with $V = R/N$ virtual nodes. Thus, the storage requirements are the same as for a physical array of size R . The savings for $N < R$ is in the processing logic and the communication circuitry.

Considering our up–down sorting algorithm with each physical node emulating V virtual nodes,

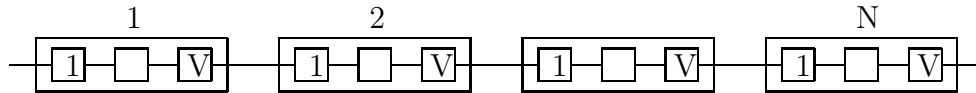


Figure 4: Emulation of a VN array on an N node array.

each step will take approximately V times as long, since upon input of a data element, it must be compared with V elements, except for the first V steps when fewer comparisons are required for each input. Upon input of the second element, one comparison suffice. Upon input of the third element, two comparisons suffice, etc., until V elements have been input. The reason why each step thereafter is not exactly V times as long for the physical array emulating V virtual nodes as a physical array with R nodes is that $V - 1$ communication steps are replaced by local memory references, which, in general, are considerably faster than communication between nodes. V comparisons must be performed in sequence, each of which requires the same time as when the physical array is of the same size as the data array. Ignoring the fact that $V - 1$ communication steps are replaced by local memory references for V comparisons, the time for sorting a sequence of R elements on N physical nodes is

$$1, 2, 3, \dots, V, V, \dots, V = V(V + 1)/2 + (R - V)V = V(2R - V + 1)/2$$

We now have

$$\text{Speedup } S_N = \frac{O(R \log_2 R)}{\frac{R}{N}(2R - \frac{R}{N} + 1)/2}$$

and

$$\text{Efficiency } E_N = \frac{O(\log_2 R)}{(2R - \frac{R}{N} + 1)/2}.$$

The efficiency decreases by about a factor of two as N increases from 1 to R , but will never be better than $O(\log_2 R)$. The speedup decreases from $O(\log_2 R)$ for $N \approx R$ to $O(\frac{\log_2 R}{R})$ for $N = 1$. Thus, in fact, as N approaches 1, our linear sorting algorithm will in fact cause a slowdown compared to a good sequential sorting algorithm. This should be no surprise, since if we consider the total sorting time for $N = 1$ it is $R(R + 1)/2$. In the emulation mode, the up-down sorting algorithm degenerates to bubble sort for $N = 1$.

How can the emulation be changed to produce an algorithm that operates in time $O(R \log_2 V)$?

In our analysis of the linear array, we have assumed that all word operations are constant time operations. However, as N increases it is likely that the width of a word increases as well. For instance, if it is anticipated that all N numbers may be distinct, then at least $\log_2 N$ bits are required to represent the numbers. Thus, for fixed width data paths, there is likely to be a dependence upon N for each communication step. With path width w and k -bit wide numbers, the communication time must be multiplied by a factor $\lceil \frac{k}{w} \rceil$. Furthermore, a comparison of two k -bit numbers cannot be made in constant time, but requires at least a time proportional to $\log_2 k$ for limited fan-in circuitry. Moreover, each node must be able to store k bits of at least one number, even if $w < k$. Thus, the total memory of the array is $O(kN)$ bits. The total wiring area is wN , and the time is at least $\frac{k}{w}N$.

We will now review the linear sorting algorithm at the bit-level. A one-bit comparator can perform the comparison in time k . Simply compare the bits in order of decreasing significance until a bit that differs in the two numbers is found. At that time, it is clear which number of the two is the larger number. Until a bit that distinguishes the two numbers is found, a copy of the compared bits can always be output, since it is part of the larger (as well as the smaller) number. Thus, assuming each comparison is taking time k , the time for the sort is $O(kN)$ and it suffices to have channels of width 1-bit for this time and a single-bit comparator in each node. Each node has $O(1)$ hardware, except for memory, which is $O(k)$.

The function of each node in bit-serial operation is

```

FOR  $j = k - 1$  STEP -1 UNTIL ( $input - bit \neq stored - bit$ ) DO
  Output the input bit
ENDFOR
IF  $input - bit > output - bit$  THEN
  Output the remaining input bits
ELSE
  Output the remaining memory-bits and
  Store the remaining input-bits in place of
  the memory-bits being output.
ENDIF

```

If we are in a position to input all k bits of each word, then a one-bit comparator is inconsistent with the communication bandwidth. By employing pipelining, k comparison cells in each node suffice to have the entire array complete the sorting in time $O(N + k)$. Figure 5 shows the first stage in sorting the sequence 2, 1, 5, 0, 3 on an array of five nodes. Since the range is less than eight, we assume that $k = 3$.

The time for data input is $N + k - 1$ steps. Another $N - 1$ steps may be required for the last item to propagate through the array. Thus, input and sorting require a total time of $2(N - 1) + k$ steps. Compared to the bit-serial algorithm, increasing the number of comparators by a factor of k results in a speedup by the same factor, ignoring lower order terms.

Though a comparison can be made in time $O(\log_2 k)$, it will not improve upon the execution time, and will actually require about twice as many comparators.

1.1.2 Sorting with data internal to the array

If the data is already present in the array, then *odd-even transposition sort* [3] can be used. This algorithm is a fairly straightforward parallelization of *bubble sort* and works as follows:

```

FOR  $j=1$  TO  $\lceil N/2 \rceil$  DO
  Processor  $i$  compares its value with processor  $i + 1$  for odd  $i$ ,
  keeping the smaller value and passing the larger value to processor  $i + 1$ .

```

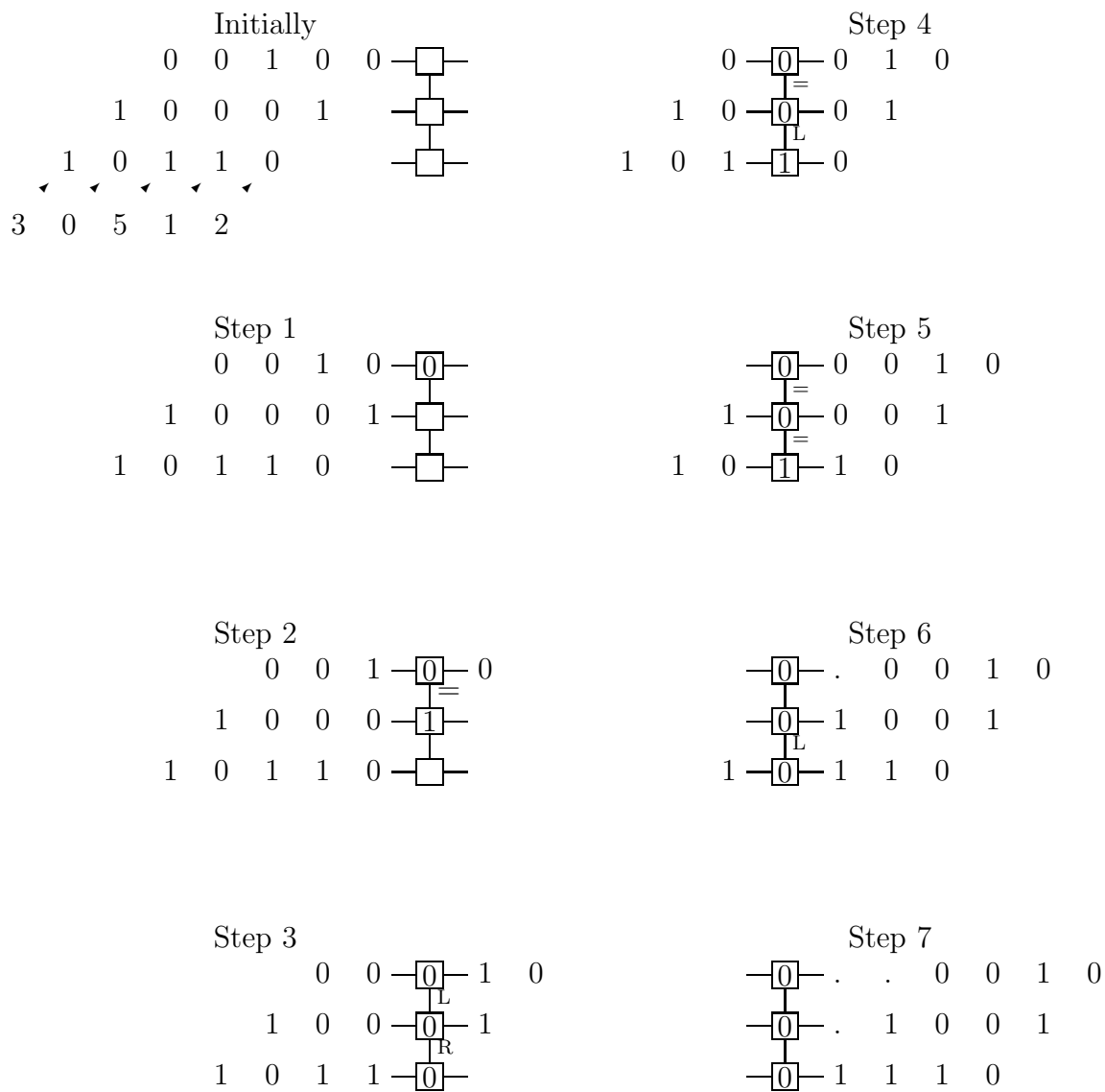


Figure 5: Simulation of the first stage of a bit-level pipelined sorting on a linear array.

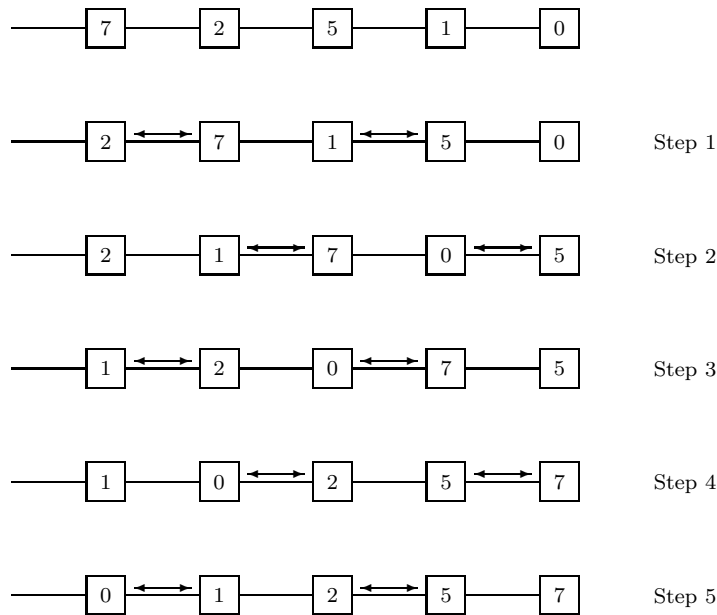


Figure 6: Odd–even transposition sort

Processor i compares its value with processor $i + 1$ for even i ,
 keeping the smaller value and passing the larger value to processor $i + 1$.

ENDFOR

Figure 6 shows an example of odd–even transposition sort. With five nodes and numbers, the result is sorted after five steps.

In our example it took precisely N steps to sort N numbers. To prove that odd–even transposition sort always yields a correct result after N steps for any sequence we will use the so–called Sorting Lemma. This lemma implies that it suffices to consider an arbitrary string of 0’s and 1’s in proving the correctness of a sorting algorithm.

Lemma 1 The 0–1 Sorting Lemma. *If an oblivious comparison–exchange algorithm sorts all sets of 0s and 1s, then it sorts all input sets of arbitrary values.*

Proof: The proof is by contradiction. Thus, we assume that the algorithm correctly sorts all sequences of 0s and 1s, but fail to sort at least one sequence of arbitrary values. Let that sequence be x_1, x_2, \dots, x_N and let π be the permutation that makes the elements appear in nondescending order, i.e., $x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)} \leq \dots, x_{\pi(N)}$. Furthermore, let ψ be the permutation that results from the sorting algorithm: $x_{\psi(1)}, x_{\psi(2)}, x_{\psi(3)}, \dots, x_{\psi(N)}$. Since this sequence is not sorted according to our assumption, there exists some k , $1 \leq k < N$, such that $x_{\pi(k)} \neq x_{\psi(k)}$ and $x_{\pi(i)} = x_{\psi(i)}$, $i < k$. There also exists a value $r > k$ such that $x_{\psi(r)} = x_{\pi(k)}$. Moreover, $x_{\psi(k)} > x_{\pi(k)}$, since the sorted sequence is nondescending. Now, define a 0–1 sequence

such that all elements x_i greater than $x_{\pi(k)}$ are equal to one, and all others equal to zero. We will now show that this sequence cannot be sorted, which contradicts our assumption that the 0–1 sequence is sorted, but the sequence x_1, x_2, \dots, x_N is not. Hence, let

$$y_i = \begin{cases} 0 & \text{if } x_i \leq x_{\pi(k)} \\ 1 & \text{if } x_i > x_{\pi(k)} \end{cases}$$

Now, consider the result of applying the algorithm to the sequence y_1, y_2, \dots, y_N . Since,

$$x_i \geq x_j \longrightarrow y_i \geq y_j,$$

the comparison–exchange operations result in the same permutation on the y sequence as it does on the x sequence. The effect of the sorting algorithm on the y sequence is

$$\begin{aligned} & y_{\psi(1)}, y_{\psi(2)}, \dots, y_{\psi(k-1)}, y_{\psi(k)}, y_{\psi(k+1)}, \dots, y_{\psi(r-1)}, y_{\psi(r)}, y_{\psi(r+1)}, \dots, y_N = \\ & = 0, 0, \dots, 0, 1, 1, \dots, 1, 0, 1, \dots, 1 \end{aligned}$$

This sequence is clearly not sorted and we have a contradiction. ■

Consider that each node in the linear array holds either a 0 or a 1, and that the total number of 1's is m . Then, when the array is sorted, nodes $N - m + 1$ through N will contain 1's. It is easy to see that the rightmost 1 in the initial distribution moves in the first comparison–exchange step if it is located in an odd node, otherwise it starts to move in the second step. Since all elements to the right of it initially are 0, the rightmost 1 moves every step until it reaches the destination. A 1 to the left of the rightmost 1 can only be blocked from moving right if there is a block of 1's. The rightmost 1 cannot block any 1 to its left, since it moves every step until it reaches its destination. Hence, the first 1 to its left will start to move no later than one step after the rightmost 1 moves. Since a 1 will only be prevented initially from moving right if there is a block of 1's to its right, the leftmost 1 will start to move right no later than step $m + 1$. It will move every step once started, and has destination $N - m + 1$. But, since the leftmost 1 can move right up to $N - m$ steps, it is guaranteed to reach its destination, even if it starts in node 1. Thus, odd–even transposition sorts yields correct result for any input sequence by the 0–1 Sorting Lemma.

Compared to sorting on a linear array with data external to the array, the number of steps is reduced to N compared to $3N - 1$. Each step now involves a get, compare, and send. Many systems are designed to carry out an exchange in the same time as sending data in only one direction between a pair of nodes. In such systems, the comparison–exchange step can also be implemented as an exchange followed by a comparison in all nodes. No communication is required after the comparison in this organization. But, twice as many comparisons are performed compared to communicating both before and after the comparison. Note however, that even though twice as many comparisons are made, the total time for carrying out the comparisons is the same as in the original algorithm, since for it only half of the comparators is used at any time.

If two or more sequences shall be sorted, then all nodes can be used for every step by starting the comparison for a pair of sequences in two successive steps. Instead of a get, compare, send sequence for each step we now have the sequence exchange, compare, exchange for each step, or exchange followed by one comparison for sequence 1 and one for sequence 2.

As in the case of the data external to the linear array, we have so far ignored the dependence of the comparison time upon the word width k , as well as the channel width w . With external data we reduced the sorting time to $2(N - 1) + k$ by pipelining the data through kN 1-bit comparators that, in fact, were connected as a two-dimensional mesh. Note, that by emulating a ring on the linear array, if it does not have a wraparound connection, we can use the same algorithm when data is internal to the array.

If comparisons are performed in time $c \log_2 k$, then odd-even transposition sort would require a time of $cN \log_2 k$, which only would compare favorably with the up-down sorting algorithm for small c and $\log_2 k$. Is there a way to perform odd-even transposition sort in time $N + k$, or $O(N + \log_2 k)$?

The speedup and efficiency of odd-even transposition sort is somewhat better than when data is external to the array. Assuming each step is a unit time operation, the improvement of odd-even transposition sort is approximately a factor of three over the up-down sorter. The same is true for the efficiency. However, both speedup and efficiencies are low for large arrays.

We will now briefly consider sorting of data when there are multiple elements per node in the array. Assume that there are R elements to be sorted, and that $N < R$. Clearly, at least R memory locations are needed in the array. We assume that the memory is evenly divided among the nodes, i.e., we assume that each node stores R/N elements of the data array.

As in the case with data external to the array, we can have each node in the physical array of size N emulate R/N nodes of a virtual array of size R . The number of comparisons each node must perform for one step increases by a factor of R/N . Communication between virtual nodes mapped to the same physical node is replaced by local memory references. The time per step is proportional to R/N . Since there still is R steps, the time is proportional to R^2/N . As N approaches one, the linear array sort may actually be slower than a good single node sort.

An improved algorithm for $N < R$ can be obtained by replacing the local comparison-exchange sequence by a local merge, preceded by a local sort. The local merge requires $2R/N - 1$ comparisons, and the initial sort requires $O(\frac{R}{N} \log_2 \frac{R}{N})$ comparisons. There are N merge steps. A simple algorithm is as follows

Perform a local sort on the R/N values in each node.

FOR $j=1$ **TO** $\lceil N/2 \rceil$ **DO**

Processor i merges its sorted sequence with processor $i + 1$ for odd i ,
 keeping the smaller half of the merged sequences
 and passing the larger half to processor $i + 1$.

Processor i merges its sorted sequence with processor $i + 1$ for even i ,
 keeping the smaller half of the merged sequences
 and passing the larger half to processor $i + 1$.

ENDFOR

The sorting time is $O(\frac{R}{N} \log_2 \frac{R}{N}) + (2\frac{R}{N} - 1)N$. We have

$$\text{Speedup: } \frac{O(R \log_2 R)}{O(\frac{R}{N} \log_2 \frac{R}{N}) + 2R - N},$$

which for $N = R$ is of order $O(\log_2 R)$, but for $N = 1$ is $O(1)$. Similarly, we have the

$$\text{Efficiency: } \frac{O(R \log_2 R)}{O(R \log_2 \frac{R}{N}) + N(2R - N)},$$

which for $N = R$ is $O(\frac{\log_2 R}{R})$. But, for $N = 1$ the efficiency is $O(1)$.

What we have learnt from these sorting algorithms is that though emulating a large virtual array on a small array is a universal tool for handling problems larger than the physical array, given sufficient memory, the speedup and efficiency may be unacceptable. The underlying reason is that often good parallel algorithms have a higher operation count than a good sequential algorithm. Thus, emulating a parallel algorithm on a single node often leads to inefficient code. For real systems, algorithms must be designed to combine the best of sequential and parallel algorithms for a competitive result.

The linear array algorithms are linear time algorithms, which corresponds to $O(R^2)$ sequential algorithms and requires time R for R nodes. Though the algorithms are not particularly efficient with respect to the total operations count, they are adequate for linear arrays of the same size as the data set. The reason for this conclusion is that the minimum time for sorting on a linear array of N nodes cannot be expected to be less than N . This lower bound is due to the potential need for one element to move from one end of the array to the opposite end.

Another lower bound can be based on bisection width. In a worst case scenario the elements in the left half would need to move to right half, and vice versa. If elements can flow concurrently in both directions on any channel, then the bisection width gives a lower bound of $N/2$.

For sorting when N is comparable to R , a sorting time less than $O(R)$ requires both other algorithms, as well as other networks with a smaller diameter, and a larger bisection width. The best known parallel sorting algorithm for many years was Batcher sort also known as bitonic sort [2] with a parallel running time of $O(\log_2^2 R)$ for R elements, and a total comparison complexity of $O(R \log_2^2 R)$. The currently best known probabilistic sort requires time $O(\log_2 R)$ and is known as AKS sort [1]. We will discuss these algorithms later.

Remark 1: We know that sorting requires area $O(N)$. What is the lower bound on time for sorting N numbers in area $O(N)$?

Remark 2: The parallel aspect of the linear array sorting algorithms we described is oblivious, i.e., the algorithms require the same time whether the data is already sorted, or nearly sorted, as if the data is totally out of order.

1.2 2-D square arrays - Shear Sort

As our first example in computations on meshes we will give a simple algorithm that will make use of the sorting algorithms for linear arrays that we presented earlier. We first present the algorithm for a square array of nodes with each axis being a power of two in length, and with each node holding one element. We then discuss sorting on rectangular arrays, and sorting on such arrays when each node stores several elements. We also address the issue of ideal array shape.

The algorithm we will describe is known as *Shearsort* [5, 7]. The idea is to sort rows and columns in alternating order. After a total of $\log_2 \sqrt{N} + 1$ such steps, the entire array of N numbers are sorted in snakelike order, i.e., the numbers appear as if a linear array had been embedded in the two-dimensional mesh using the modified step embedding technique. The algorithm is illustrated in Figure 7 for a 4×4 array. The arrows point in the direction of nondecreasing values.

The algorithm can be described as follows with rows and columns labeled from zero::

```

FOR  $I = 0$  TO  $\log_2 \sqrt{N} - 1$  DO
    Sort even rows in nondescending order in a left to right order.
    Sort odd rows in nondescending order in a right to left order.
    Sort all columns in a nondescending order from top to bottom.
ENDFOR
Sort even rows in nondescending order in a left to right order.
Sort odd rows in nondescending order in a right to left order.

```

The total number of steps is $2 \log_2 \sqrt{N} + 1 = \log_2 N + 1$, where each step is a row sort or a column sort. Using odd–even transposition sort, we know that each such sort can be carried out in \sqrt{N} steps. Thus, Shearsort completes in a time of $\sqrt{N}(\log_2 N + 1)$. We have

$$\text{Speedup: } \frac{O(N \log_2 N)}{\sqrt{N}(\log_2 N + 1)} = O(\sqrt{N})$$

$$\text{Efficiency: } \frac{O(\sqrt{N})}{N} = O\left(\frac{1}{\sqrt{N}}\right)$$

Compared to odd–even transposition sort Shearsort yields considerably improved speedup and efficiency. However, compared to a good sequential sort, the efficiency is asymptotically poor. We will later see how Shearsort can be improved to complete in $3\sqrt{N} + o(\sqrt{N})$ time. The lower bound for sorting on a square mesh is $O(\sqrt{N})$ based either on the diameter of the mesh, or based on the bisection width. Thus, Shearsort is nonoptimal by a factor of $O(\log_2 N)$. We will later show that a lower bound for sorting on a mesh is $3\sqrt{N} - o(\sqrt{N})$.

Theorem 1 *Shearsort sorts N elements on an N node square mesh with one element per node in time $\sqrt{N}(\log_2 N + 1)$.*

Proof: We will now show that Shearsort does sort correctly in $\log_2 N + 1$ steps. The proof is based on the 0–1 Sorting Lemma. The proof is based on recursion, showing that for each

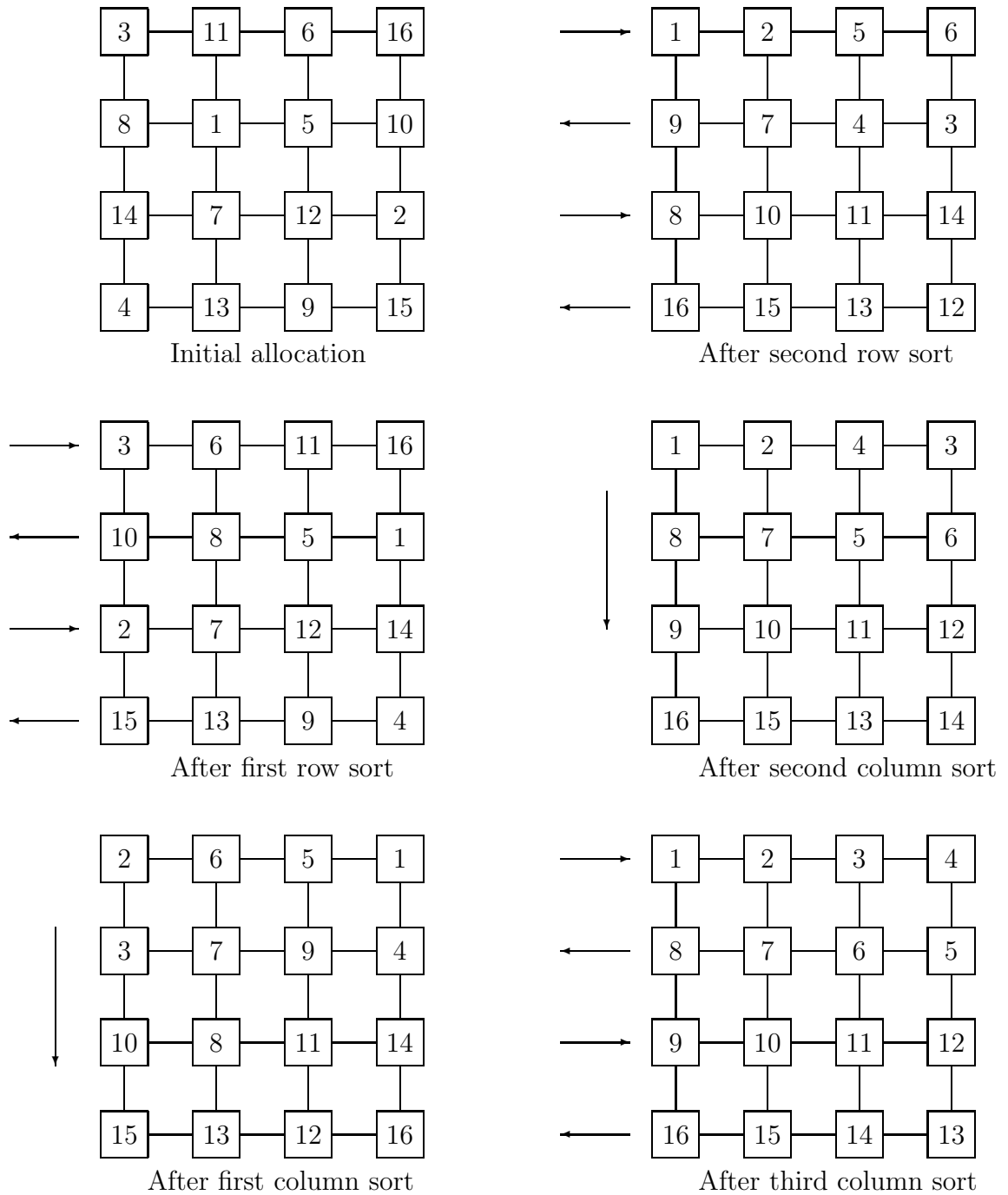


Figure 7: Shearsort on a 4×4 processor array.

Proof: Use the modified odd–even transposition sort that is based on exchange–merge instead of exchange–comparisons. Each such step can be completed in time $2\frac{M}{N}$. The time for a row and a column sort based on the exchange–merge transposition sort is $2\frac{M}{N}N_1$ and $2\frac{M}{N}N_0$, respectively. In order to prove that the sorting algorithm indeed sorts the numbers in snakelike ordering we need to prove that the number of unsorted nodal rows decreases by at least a factor of two in each step. For this, we consider a pair of nodal rows, just as in the case with one element per node. Clearly, if the top node contains all 0’s, then the vertical merge of two rows results in the top node still containing all 0’s, regardless of the content of the bottom node. Similarly, if the top node contains all 1’s, then the bottom node will receive all 1’s after the vertical merge regardless of its initial content. Thus, if all bottom nodes contain all 1’s up until the first node in the top row that have all 1’s, then the bottom row is clean. If on the other hand the bottom row has nodes with all 0’s starting with the node immediately to the right of the last node in the top row with all 0’s, then the vertical merge yields a clean top row. The remaining case to consider is the one where a node in the top row contains the sequence $0\dots 01\dots 1$ and the corresponding node in the bottom row contains the sequence $1\dots 10\dots 0$. The merge operation results in all 0’s in the top node if there are more 0’s in the top node than 1’s in the bottom node, and the top nodal row is clean since the two rows are sorted in opposite order. If there are fewer 0’s in the top node than 1’s in the bottom node, then the bottom row becomes clean. Thus, the number of clean nodal rows increases by at least a factor of two. ■

The optimal nodal array shape is a square, just as in case with one element per node. Since potentially the same amount of data must be communicated in either the row or column direction, minimizing the maximum axis extent is optimum.

1.2.3 Optimal Shear sort

Shear sort sorts N numbers on an N node square two–dimensional mesh in time $\sqrt{N}(\log_2 N + 1)$. A naive lower bound is the diameter, $2\sqrt{N}$. An improved lower bound of $3\sqrt{N} - 2\sqrt[4]{N} - 4$ for sorting in snake–like order on a square mesh is given in [4]. We will now describe an algorithm that sorts in time $8\sqrt{N}$ steps. The algorithm is a fairly simple modification of Shear sort.

The algorithm is as follows

1. Recursively sort each quadrant of the array into snake–like order.
2. Sort rows of the entire array in alternate order.
3. Sort the columns of the entire array.
4. Perform $2\sqrt{N}$ steps of odd–even transposition sort along the snake.

The four phases are illustrated in Figure 8.

To prove that this sorting algorithm indeed yields the correct result we use the 0–1 sorting lemma.

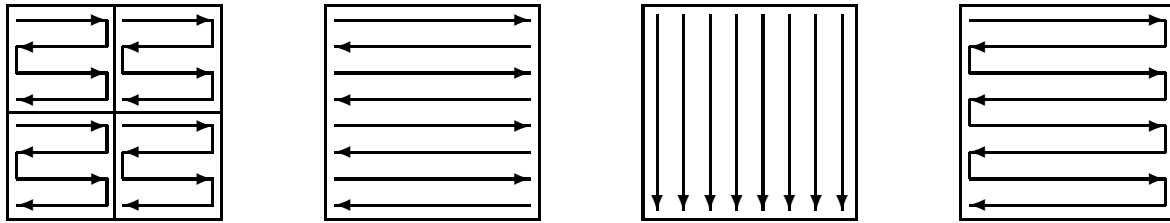


Figure 8: The four phases of sorting on a mesh.

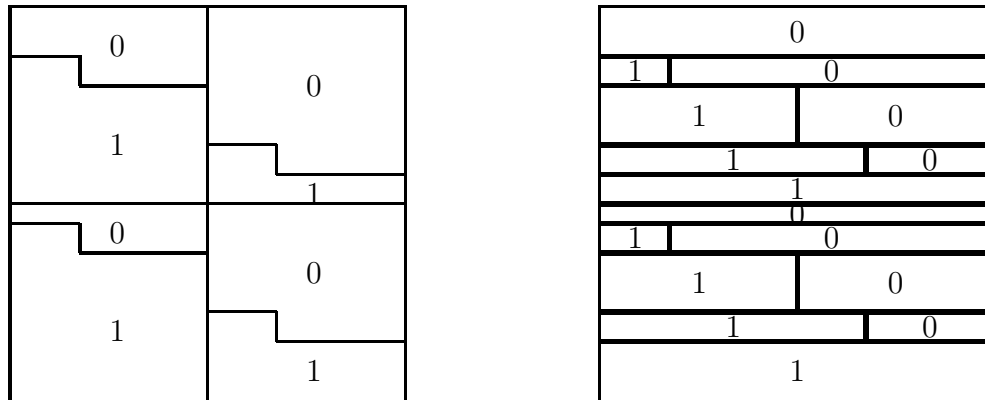


Figure 9: The result of the quadrant snake sorts.

The top and the bottom half of the mesh each consists of three regions: a clean top portion of all 0's; a bottom clean portion of all 1's. A contiguous dirty region of which all but the first and last rows have exactly as many 0's as 1's (since this region is made up of clean rows of the two quadrants making up the half). Figure 9 shows the distribution of 0's and 1's after the quadrant sort.

If the number of rows in the dirty region with an equal number of 0's and 1's is even, then after the row sort in alternating order and the column sort, all these rows become clean. As for the other two dirty rows, the first and last of the dirty region, at least one of them become clean after the column sort. The arguments are the same as for Shear sort. Thus, after the row and column sort we have at most two dirty rows, one from the top half and the other from the bottom half.

For an odd number of 50/50 rows all but one become clean by the arguments above. Thus, let's consider three dirty rows as in Figure 10. It is easily seen that a clean row of 0's and a clean row of 1's will result from the column sort on these rows, and at most one dirty row remains.

Thus, regardless of whether there is an odd or even number of 50/50 rows at most two dirty rows remain after the row and column sort. Furthermore, these rows must be adjacent after the column sort.

Hence, the final odd–even transposition sort along the snake will sort these two dirty rows.

The time $T(N)$ to sort N numbers on an N node mesh by the algorithm is

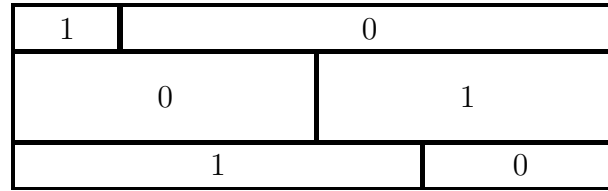


Figure 10: An odd number of 50/50 rows.

$$T(N) = T(N/4) + 2\sqrt{N} + \sqrt{N} + \sqrt{N} = T(N/4) + 4\sqrt{N} = 8\sqrt{N} - 1$$

1.3 Optimal mesh sort

See Leighton’s book pp. 148 – 151.

1.4 A tight lower bound for mesh sort

See Leighton’s book pp. 151 – 153.

1.5 Heapsort

Heapsort is easily implemented on a tree with the data entering and leaving through the root. A simple algorithm is as follows:

- Accept data from the parent node. If the item is smaller than the number stored, then pass it on to a child, with left and right child receiving numbers in alternating order. If the received number is greater than the stored number, then pass the stored number on and keep the new number. Numbers are always passed to left and right child in alternating order.
- When all numbers are received, output the number stored. Read both the left and right child numbers, keep the larger, then push the smaller back down to a child in left/right alternating order.

Figures 11 and 12 show the input and output phases for $N = 16$.

We will give additional algorithms for sorting on a complete binary tree later.

The lower bound for sorting on a complete binary tree is $O(N)$, i.e., the same as for sorting on a linear array. If the data is entering and leaving the tree through the root, it is clear that the time cannot be better than $2N$ steps for sorting N numbers on an N node complete binary

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

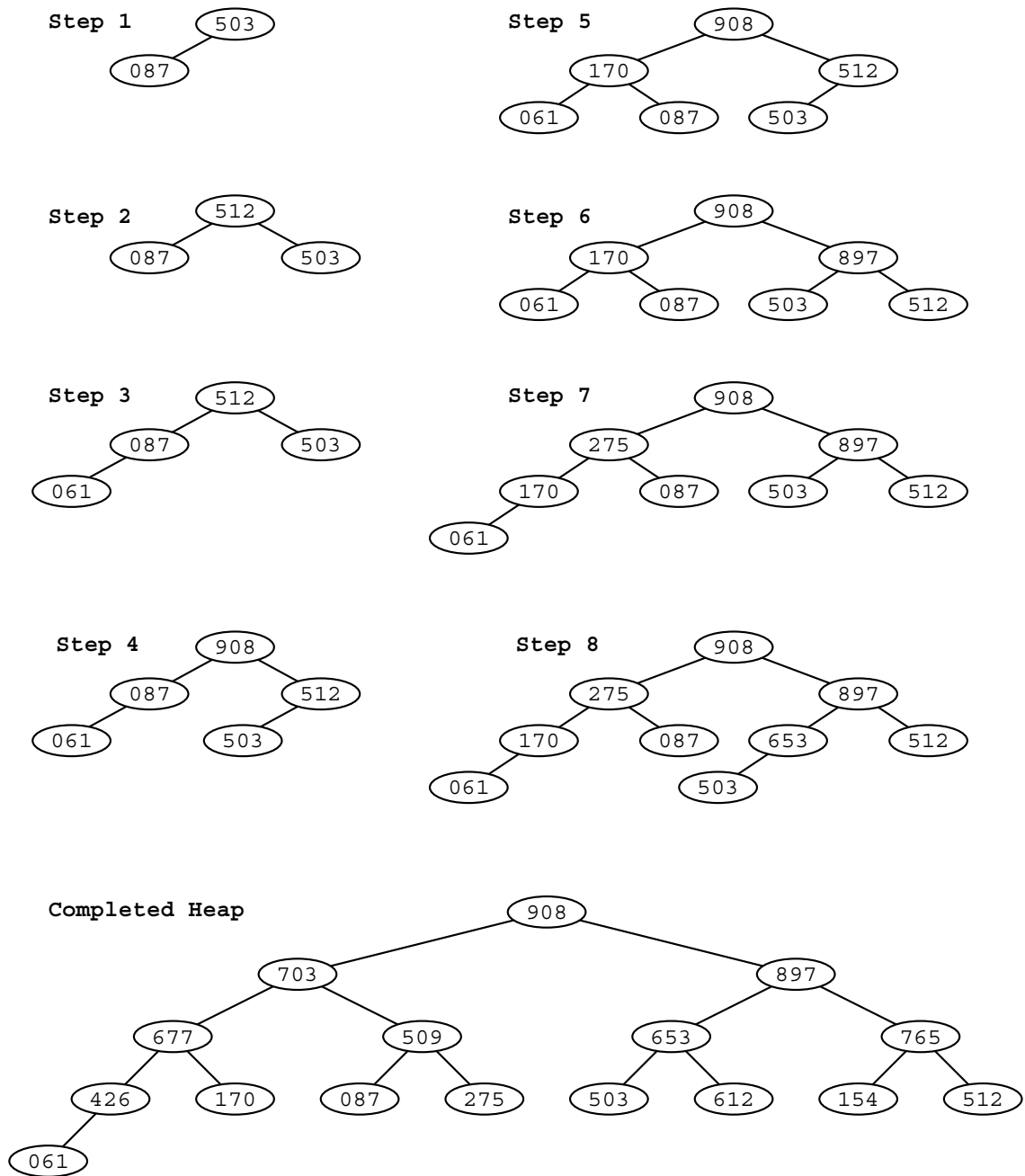


Figure 11: Constructing a heap ($N = 16$).

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

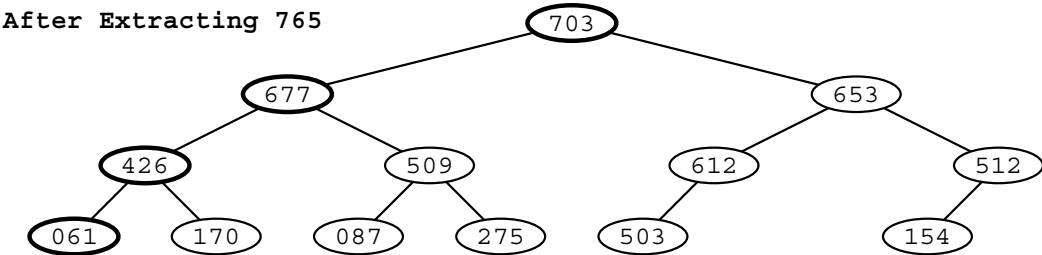
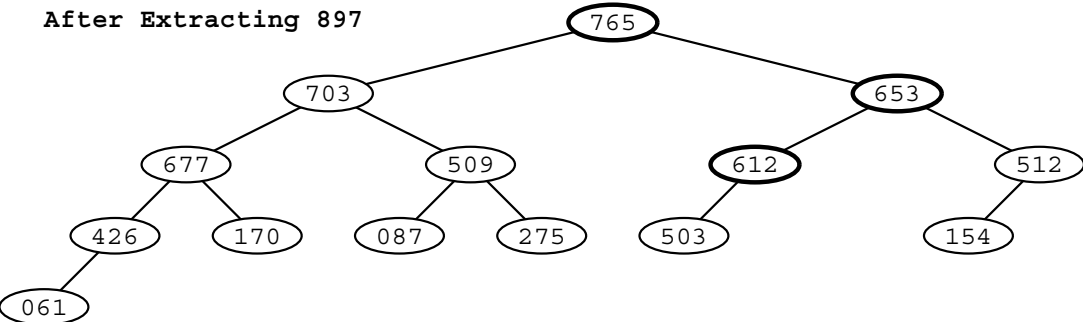
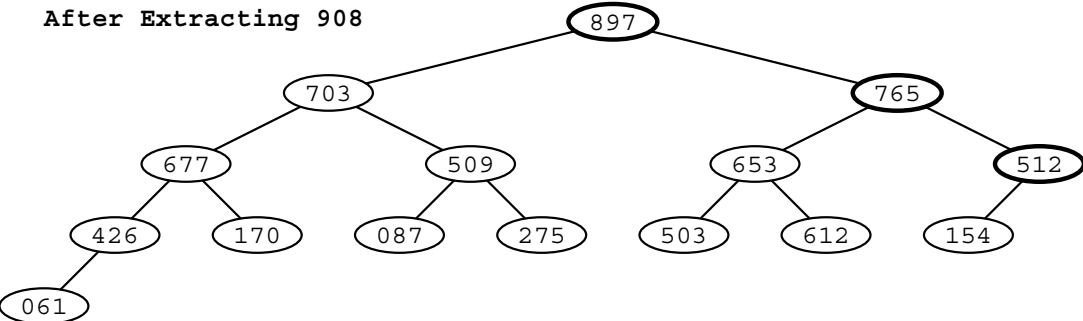


Figure 12: Extracting sorted values from a heap ($N = 16$).

tree. This bound is the same as our up–down sorting algorithm for a linear array. Moreover, the number of nodes in the complete binary tree is $2^{\lceil \log_2(N+1) \rceil} - 1$. Thus, whereas the linear array can easily be adjusted to fit any N , the complete binary tree cannot.

If the data is internal to the tree, then the bisection width at the root sets a lower bound of $(N - 1)/2$ if all elements in the subtrees of the root must be exchanged between the two subtrees. Again, this bound is the same as that for a linear array.

If the data enters through the leafs of the tree and exits through the leafs, then, in the worst case all elements entering the leafs of one subtree of the root may have to leave through the leafs of the other subtree. A total of $(N + 1)/2 + 2(\log_2(N + 1) - 1)$ steps are required. The first term is due to the congestion at the root and the second term is due to the propagation time for data moving from the leafs to the root and back to the leafs. If the data may exit through the root, one of the propagation phases is eliminated and the time becomes $N + \log_2(N + 1) - 1$ since all N elements must leave the root. Thus, at best, the sorting on a tree may improve upon sorting on a linear array by a factor of four.

Sorting on meshes of two or more dimensions requires fewer steps than sorting on a complete binary tree. Moreover, linear arrays and two–dimensional meshes can be laid out in the plane within area $O(N)$ and wires of length $O(1)$.

1.6 Discussion

Odd–even transposition sort is suitable for sorting on linear arrays in that its sorting time is compatible with the worst case lower bound for sorting on linear arrays. However, the running time of the algorithm is the same whether the data is already sorted or not. The up–down sorter is suitable for sorting on a linear array, with data originally and finally external to the array. But, as for the odd–even transposition sort, the sorting time is independent of the data distribution. With data external to the array, a sorting time proportional to the number of elements to be sorted is a lower bound, since the elements must all be inspected (entered into the array), regardless of whether the elements are already sorted or not.

Shear sort is a very elegant sort for two–dimensional meshes, with a sorting time in a fairly straightforward implementation suboptimal by a factor of about $\log_2 N$ that however can be improved to be optimal within at most a small constant factor. But again, the algorithm is oblivious, i.e., the running time is independent of whether the data is almost sorted or not.

Sorting on a mesh can be performed much faster than sorting on a linear array. This conclusion holds also if we take into account that the bandwidth of each communication channel may only be half of that of the linear array, since for the two–dimensional mesh four channels must share the perimeter (pins) instead of only two. However, if a module with fixed perimeter holds more than a single node, then for a $\sqrt{M} \times \sqrt{M}$ subgrid sharing the perimeter, each mesh channel is only allocated a $\frac{1}{4\sqrt{M}}$ fraction of the shared perimeter instead of $\frac{1}{2}$ for the linear array. Adjusting the time per step for a potential difference in channel width yields a time of $\frac{4\sqrt{M}}{w}\sqrt{N}$ per row and column sort for the mesh instead of $\frac{2}{w}N$ for the linear array. Even with this adjustment, since $\sqrt{M} < \sqrt{N}$, the mesh with Shearsort is still faster if $2\sqrt{M}\log_2 N < \sqrt{N}$.

Next we will consider sorting on tree connected networks, then discuss the “classical” parallel sort known as bitonic sort, or Batcher sort after Kenneth Batcher [2]. Then we will discuss Radix sort Sample Sort and Merge-Sort.

References

- [1] Michael Ajtai, J. Komlos, and E Szemerédi. An $o(n \log_2 n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing*, pages 1–9. ACM Press, 1983.
- [2] Kenneth E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314. IEEE, 1968.
- [3] Gerald M. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Trans. Computers*, C-27(1):84–87, 1978.
- [4] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
- [5] K. Sado and Y. Igarashi. Some parallel sorts on a mesh-connected processor array and their efficiency. *Journal of Parallel and Distributed Computing*, 3:398–410, September 1986.
- [6] H. Samet. *Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
- [7] I. Scherson, S. Sen, and A. Shamir. Shear-sort: A true two-dimensional sorting technique for VLSI networks. In *IEEE-ACM International Conference on Parallel Processing*, pages 903–908. IEEE Computer Society, 1986.