Lecture 7: Sorting

Lecture Outline

(Reading: Chapter 10 of textbook)

```
1. Introduction
   (1) Problem definition
   (2) Sorting related issues
   (3) Enumeration sort on CRCW PRAM model
   (4) Lower bounds on parallel sorting
2. Bitonic sequence and binotic merge
3. Parallel sorting on processor arrays
   (1) On SIMD 1-D-mesh
   (2) On SIMD 2-D-mesh
   (3) On SIMD hypercubes
   (4) On SIMD shuffle-exchange networks
4. Multiprocessor implementation of quicksort
   (1) Parallel quicksort on UMA multiprocessors
   (2) Hyperquicksort -- parallel quicksort on hypercube
       multiprocessors
```

1. Introduction

(1) Problem definition.

Definition. Let D be a domain with a linear order, i.e. for any $a, b \in D$, either $a < b$, or $a = b$, or $a > b$. The sorting problem is the following: Given a collection of n elements $a_0, a_1, \cdots, a_{n-1}$, where each $a_i \in D$, find a permutation $(\pi_1, \pi_2, \cdots, \pi_n)$ such that

$$a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n}$$

(2) Sorting related issues

a. One of the most widely used applications (if not the most).

b. *External sorting* and *internal sorting*

c. Any sequential algorithm for sorting n elements takes time $\Omega(n \log n)$

(3) Enumeration sort on CRCW PRAM model

a. Basic ideas

(a) Spawn $n \times n$ processors $P_{i,j}$

(b) Let each $P_{i,j}$ calculate if $a_i < a_j$, or $a_i = a_j$ with $i < j$. If yes, it records a value 1, otherwise a value 0.

```
ENUMERATION SORT(SPECIAL CRCW PRAM)
    Parameter   n                       {Number of elements}
    Global      a[0...(n − 1)]          {Elements to be sorted}
                position[0...(n − 1)]   {Stored positions}
                sorted[0...(n − 1)]     {Contains sorted elements}
begin
    spawn (P_{i,j}, for all 0 ≤ i, j < n)
    for all P_{i,j}, where 0 ≤ i, j < n do
        position[i] ← 0
        if a[i] < a[j] or (a[i] = a[j] and i < j) do
            position[i] ← 1
        endif
    endfor
    for all P_{i,0}, where 0 ≤ i < n do
        sorted[position[i]] ← a[i]
    endfor
end
```

Figure 81: A set of n elements can be sorted in $\Theta(\log n)$ time using n^2 processors, given a CRCW PRAM model in which simultaneous writes to the same memory location cause the sum of the values to be assigned. If the time needed to spawn the processors is not counted, the algorithm executes in constant time (Fig.10-1, p.257)

(c) Each $P_{i,j}$ writes, concurrently, the value recorded in last step. This concurrently writing guarantees that if two processors write 1 to the same variable, the final written value will be 2.

(d) Once knowing the number of values that is larger than particular a_i, the algorithm then places a_i in the correct final location in array a.

b. The pseudo code: Fig.10-1, p.257.

c. Problems: the CRCW assumption is not practical

(4) Lower bounds on parallel sorting

a. Lower bounds on parallel sorting on 1-D mesh

Theorem 10.1. Assume that n elements are to be sorted on a processor array organized as a one-dimensional arrary. Also assume that before and after the sort the elements are to be distributed evenly, one element per processor. Then a lower bound on the time complexity of any sorting algorithm is $\Theta(n)$.

Proof: Based on the facts that (a) the bisection width is 1; (b) diameter is $n-1$. If all elements at one side of the bisection must be moved to the other side, it will take $\Theta(n)$ time to move them.

b. Lower bounds on parallel sorting on 2-D mesh

Theorem 10.2. Assume that n elements are to be sorted on a processor array organized as a two-dimensional arrary. Also assume that before and after the sort the elements are to be distributed evenly, one element per processor. Then a lower bound on the time complexity of any sorting algorithm is $\Theta(\sqrt{n})$.

Proof: Based on the facts that (a) the bisection width for a 2-D mesh is $O(\sqrt{n})$; and (b) the diameter is $2(\sqrt{n}-1)$. A sort may have to sway the elements originally stored in nodes at diagonally opposite corners of the mesh. Hence a lower bound on the number of data routings needed is $4(\sqrt{n}-1)$. Therefore any algorithm to sort n^2 elements on the SIMD-MC^2 model has time complexity $\Omega(\sqrt{n})$.

c. Lower bounds on parallel sorting shuffle-exchange networks

Theorem 10.3. Assume that $n = 2^k$ elements are to be sorted on a processor array organized as a shuffle-exchange network. Also assume that before and after the sort the elements are to be distributed evenly, one element per processor. Then a lower bound on the time complexity of any sorting algorithm is $\Theta(\log n)$.

Proof: Based on the fact that the diameter of a shuffle-exchange network is $\log n$.

2. *Bitonic sequences* and *bitomic merge*: basis for sorting algorithms on several parallel models

(1) Basic operation: compare and exchange. Comparators: low-to-high (+), and high-to-low (−), Fig.10-4,p.260.

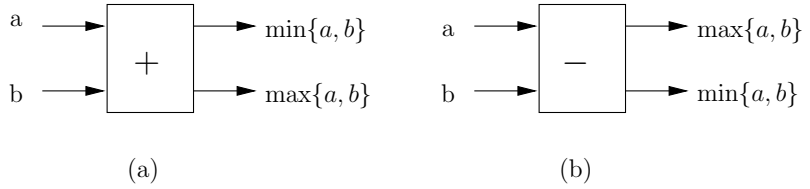


Figure 82: Two comparators. (a) Low-to-high comparator. (b) High-to-low comparator (Fig.10-4, p.260)

(2) **Definition.** (Bitonic sequences) A **bitonic sequence** is a sequence of numbers $a_0, a_1, \cdots, a_{n-1}$ with the property that (a) there exists an index i, $0 \le i \le n-1$,

such that a_0 through a_i is monotonically increasing and a_i through a_{n-1} is monotonically decreasing, or else (b) there exists a cyclic shift of indices so that condition (a) is satisfied.

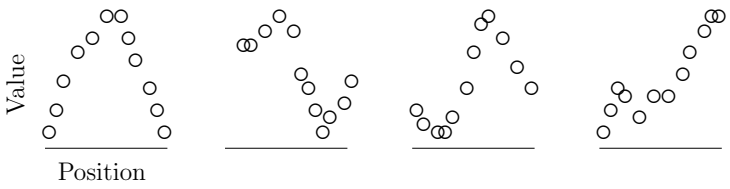


Figure 83: The first three sequences are bitonic sequences; the last sequence is not comparator (Fig.10-5, p.260)

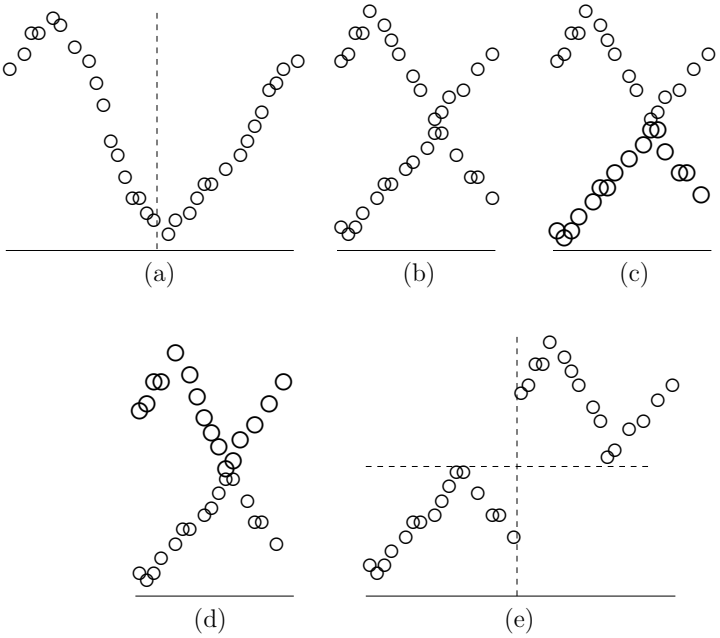


Figure 84: First part of informal proof of Lemma 10.1. (a) Original bitonic sequence. (b) First half of sequence overlayed on second half of sequence. (c) Minimum values. (d) Maximum values. (e) Transformed sequence (Fig.10-6, p.261)

(3) **Lemma 10.1.** A single compare-exchange step can split a single bitonic sequence into two bitonic sequences. Moreover, each element from one of the two sequences is smaller than any elements of the other sequence. Specifically, let $a_0, a_1, \cdots, a_{n-1}$ be a bitonic sequence with property that $a_0 \leq a_1 \leq \cdots \leq a_{n/2-1}$, and $a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$. Then each of the following two sequences is a bitonic sequence:

$$min(a_0, a_{n/2}),\ min(a_1, a_{n/2+1}), \cdots, min(a_{n/2-1}, a_{n-1})$$

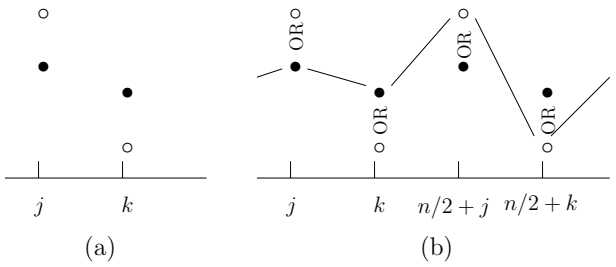


Figure 85: Second part of informal proof of Lemma 10.1. (a) Suppose the jth element of the first half of the final sequence is greater than the kth element of the last half of the final squence. (b) If that is true, the original sequence could not be a bitonic sequence (Fig.10-7, p.262)

$$max(a_0, a_{n/2}),\ max(a_1, a_{n/2+1}), \cdots, max(a_{n/2-1}, a_{n-1})$$

Furthermore, every element in the first sequence is smaller than every element in the second sequence.

- **Proof**: The proof of the first half of the lemma is illustrated by Fig.10-6. The second half is proved by contradiction. Fig.10-7 shows the ideas.
- By this lemma, given a bitonic sequence of length n, a single compare-exchage step divides the sequence into two bitonic sequences of length $n/2$ each, as illustrated by Fig.10-8, p.263.
- The above idea can be applied recursively to the length $n/2$ bitonic sequences, and so on. This leads to the next theorem.

(4) Batcher's theorem and bitonic merge sort

Theorem 10.5. (Batcher [1968]) A list of $n = 2^k$ unsorted elements can be sorted by using a network of $2^{k-2}k(k+1)$ comparators in time $\Theta(\log^2 n)$.

Proof: Fact: a bitonic sequence of length 2^m can be sorted in time $\Theta(m)$.

- A given unsorted sequence of $n = 2^k$ elements can be thought as $n/2$ bitonic sequences of 2 elements each. Each of these 2-element binotic sequences can be sorted in 1 time unit. We then get $n/4$ bitonic sequence of 4 elements each. Sorting each of these 4-element sequences takes $\log 4 = 2$ time units. And so on.
- We eventually get 2 bitonic sequences of $n/2$ elements each. Sorting each of them needs time $\log n - 1 = k - 1$, resulting a single bitonic sequence, which we need $\log n = k$ times to sort. Total level of comparators is $1+2+\cdots k = k(k+1)/2 = \log n(\log n+1)/2$, i.e. $\Omega(log^2 n)$. Total number of comparators is $(n/2)(k(k+1)/2) = 2^{k-2}k(k+1)$.

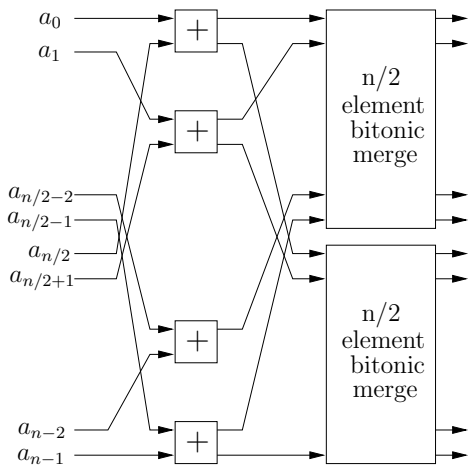


Figure 86: The recursive nature of bitonic merge. Given a bitonic sequence, a single compare-exchange step divides the sequence into two bitonic sequences of half the length. Applying this step recursively yields a sorted sequence, which can be thought of as half of a bitonic sequence of twice the length (Fig.10-8, p.263)

Example. Fig. 10-12, page 265.

(5) **Theorem 10.6.** (Stone [1971]) A list of $n = 2^k$ unsorted elements can be sorted in time $\Theta(\log^2 n)$ with a network of $2^{k-1}[k(k-1)+1]$ comparators using the perfect shuffle interconnections.

a. Fig.10-13: sorting a bitonic sequence of length 16 using Stone's perfect shuffle interconnections.

b. Fig.10-14: bitonic merge sort of an unsorted list of eight elements using Stone's perfect shuffle interconnections.

c. Fig.10-15: Sorting machine based on perfect shuffle connection. It takes advantages of the regular interconnection pattern between steps.

3. Parallel sorting on processor arrays

(1) On SIMD-MC^1 model

a. SIMD-MC^1 model: one-dimensional mesh (array)

b. Odd-even transposition sort.

(a) Array $A = (a_0, a_1, \cdots, a_{n-1})$ contains the n elements to be sorted; arrays $B = (b_0, b_1, \cdots, b_{n-1})$ and $T = (t_0, t_1, \cdots, t_{n-1})$ are used to store temporary values.

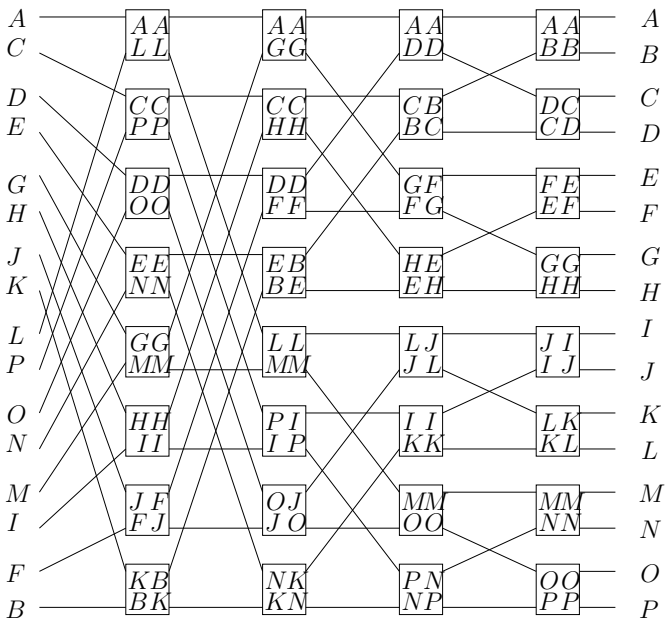


Figure 87: Sorting a bitonic sequence of length 16 by using bitonic merge (Fig.10-10, p.264)

(b) The whole algorithm consists of $n/2$ iterations, and each iteration has two phases.

i. In the first phase, called *odd-even exchange*, the value of a_i in every odd-numbered processor i (except processor $n-1$) is compared with the value a_{i+1} stored in even-numbered processor $i+1$. The values are exchanged, if necessary, so that the lower-numbered processor contains the smaller value.

ii. The second phase is called *even-odd exchange*, the value of a_i in every even-numbered processor i is compared with the value a_{i+1} stored in odd-numbered processor $i+1$. The values are exchanged, if necessary, so that the lower-numbered processor contains the smaller value. After $n/2$ iterations, the values are sorted in ascending order.

c. The parallel algorithm code: Fig.10-2, p.259

ODD-EVEN TRANSPOSITION SORT(ONE-DIMENSIONAL MESH PROCESSOR ARRAY):

Parameter	n	
Global	i	
Local	a	{Element to be sorted}
	t	{Element taken from adjacent processor}

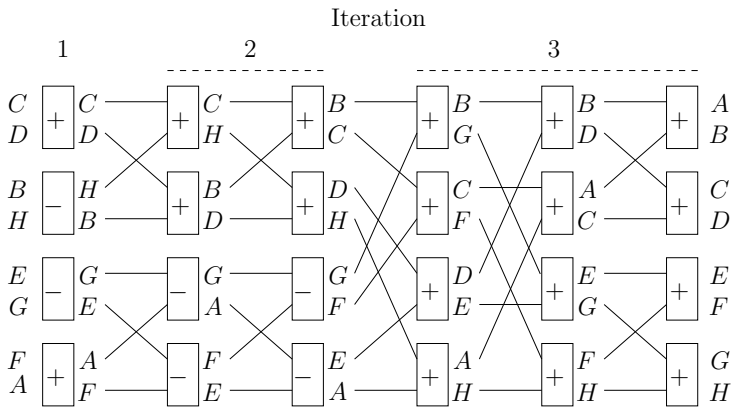


Figure 88: Bitonic merge-sort of an unsorted list of eight elements (Fig.10-12, p.265)

```
begin
   for i ← 1 to n/2 do
      for all P_j, 0 ≤ j ≤ n − 1, do
         if j < n − 1 and odd(j) then   {Odd-even exchange}
            t ⇐ successor(a)            {Get value from successor}
            successor(a) ⇐ max(a, t)    {Give away larger value}
            a ← min(a, t)               {Keep smaller value}
         endif
         if even(j) then                {Even-odd exchange}
            t ⇐ successor(a)            {Get value from successor}
            successor(a) ⇐ max(a, t)    {Give away larger value}
            a ← min(a, t)               {Keep smaller value}
         endif
      endfor
   endfor
end
```

Fig.10-2. Odd-even transposition sort algorithm for the one dimensional mesh prcessor array model (p.259)

d. **Example**: Fig.10-3, p.259

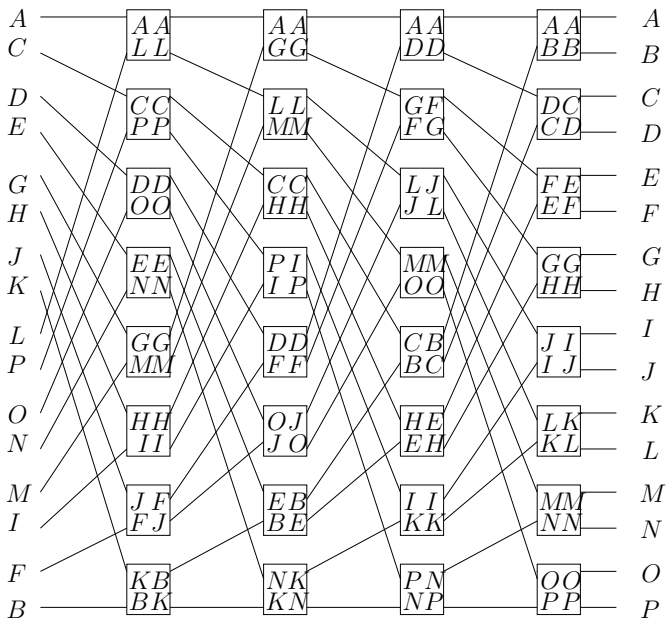


Figure 89: Sorting a bitonic sequence of length 16 by using Stone's perfect shuffle (Fig.10-13, p.266)

Indices:	0	1	2	3	4	5	6	7
Initial values:	G	H	F	D	E	C	B	A
After odd-even exchange:	G	F	<H	D	<E	B	<C	A
After even-odd exchange:	F	<G	D	<H	B	<E	A	<C
After odd-even exchange:	F	D	<G	B	<H	A	<E	C
After even-odd exchange:	D	<F	B	<G	A	<H	C	<E
After odd-even exchange:	D	B	<F	A	<G	C	<H	E
After even-odd exchange:	B	<D	A	<F	C	<G	E	<H
After odd-even exchange:	B	A	<D	C	<F	E	<G	H
After even-odd exchange:	A	<B	C	<D	E	<F	G	<H

Fig.10-3. Odd-even transposition sort of eight values on the one-dimensional mesh prcessor array model (p.259)

e. **Theorem 10.4.** The complexity of sorting n elements on an SIMD-MC^1 model with n processors using odd-even transposition sort is $\Theta(n)$.

(2) On SIMD-MC^2 model

a. Lower bound – see Theorem 10.2.

b. A $\Theta(\sqrt{n})$ algorithm based on bitonic merge.
Theorem 10.7. An algorithm exists to sort $n = m^2 = 2^k$ elements on the SIMD-MC^2 model in time $\Theta(m) = \Theta(\sqrt{n})$. (Thompson and Kung 1977)

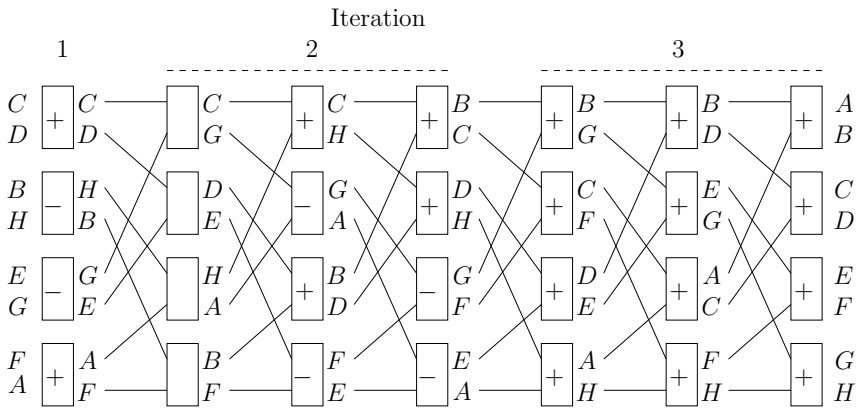


Figure 90: Bitonic merge-sort of an unsorted list of eight elements, by using Stone's perfect shuffle interconnection (Fig.10-14, p.267)

Proof: Given $n = 2^k$ elements, the bitonic merge sort consists of k iterations, where iteration i has i compare-exchange steps.

(a) Each compare-exchange steps requires two data routings: the 1st routing brings together the elements to be compared, and the 2nd routing redistribute them. See Fig. 10-18,p.269 for an illustration for $n = 16$. Each row represents the position of an element. Arrows represent compare-exchange operations. To perform a compare-exchange, the element at the position marked by the tail of an arrow is routing to the position marked by the arrowhead. After the two elements are compared, the smaller is routed back to the tail position.

(b) Notice that elements in positions whose representations differ in their least significant bit are compared every iteration, while elements in positions whose representations differ in their most significant position are compared only on the last iteration. An efficient implementation of bitonic merge on the SIMD-MC^2 model must have the property that if bit i is less significant than bit j, then a compare-exchange on bit i cannot require more data routings than a compare-exchange on bit j. One way to satisfy this condition is to use a "shuffled row-major" addressing scheme, as shown in Fig. 10-19(a). The advantage of this scheme is that "shuffling" operations occur on square subsections of the mesh, reducing the number of routing operations.

(c) The direction of routing necessary at each compare-exchange step depends upon the index of the particular node. See Fig. 10-20 for an illustration. To sort $n = m^2 = 2^k$ elements requires $\log n$ phases. The total number of routing steps performed is

$$\Sigma_{i=1}^{\log n} \Sigma_{j=1}^{i} 2^{\lfloor (j-1)/2 \rfloor}$$

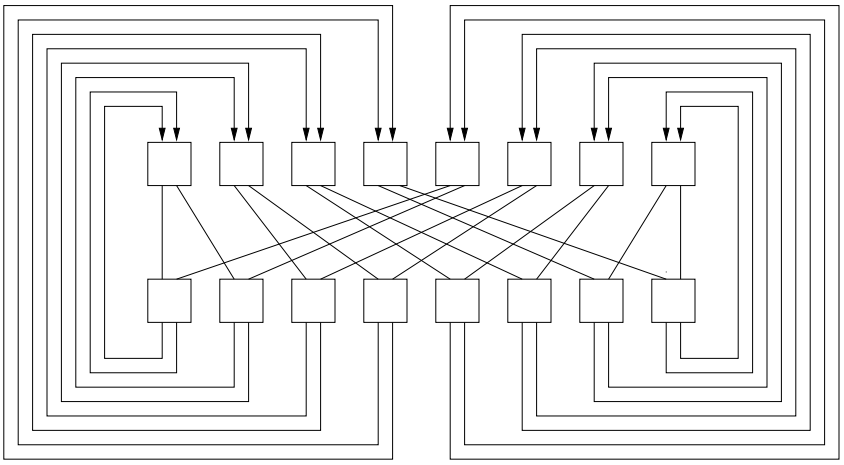


Figure 91: Sorting machine based upon perfect shuffle connection (Sedgewich, 1983) (Fig.10-15, p.267)

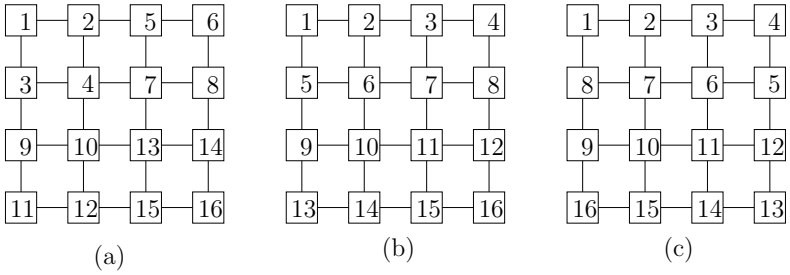


Figure 92: Three index functions mapping list elements into a two-dimensional mesh. (a) Shuffle row-major order. (b) Row-major order. (c). Snakelike row-major order (Fig.10-19, p.270)

which is $\Theta(\sqrt{n})$. The total number of comparison steps is $\Sigma_{i=1}^{\log n}$, which is $\Theta(\log^2 n)$. Thus the worst-case time complexity is $\Theta(\sqrt{n})$, making it optimal for this model.

c. **Example**. Fig.10-20, p.271.
Note: the compare-exchange operations in Stage 1 between P_5 and P_6, P_7 and P_8, P_{13} and P_{13}, P_{15} and P_{16}, are different in this lecture notes than in the textbook. The operations in this lecture notes are consistent with Fig.10-18.

(3) Summary of compare-exchange steps by using bitonic merge-sort stragety: Fig.10-18, p.269

(4) On SIMD-CC model.

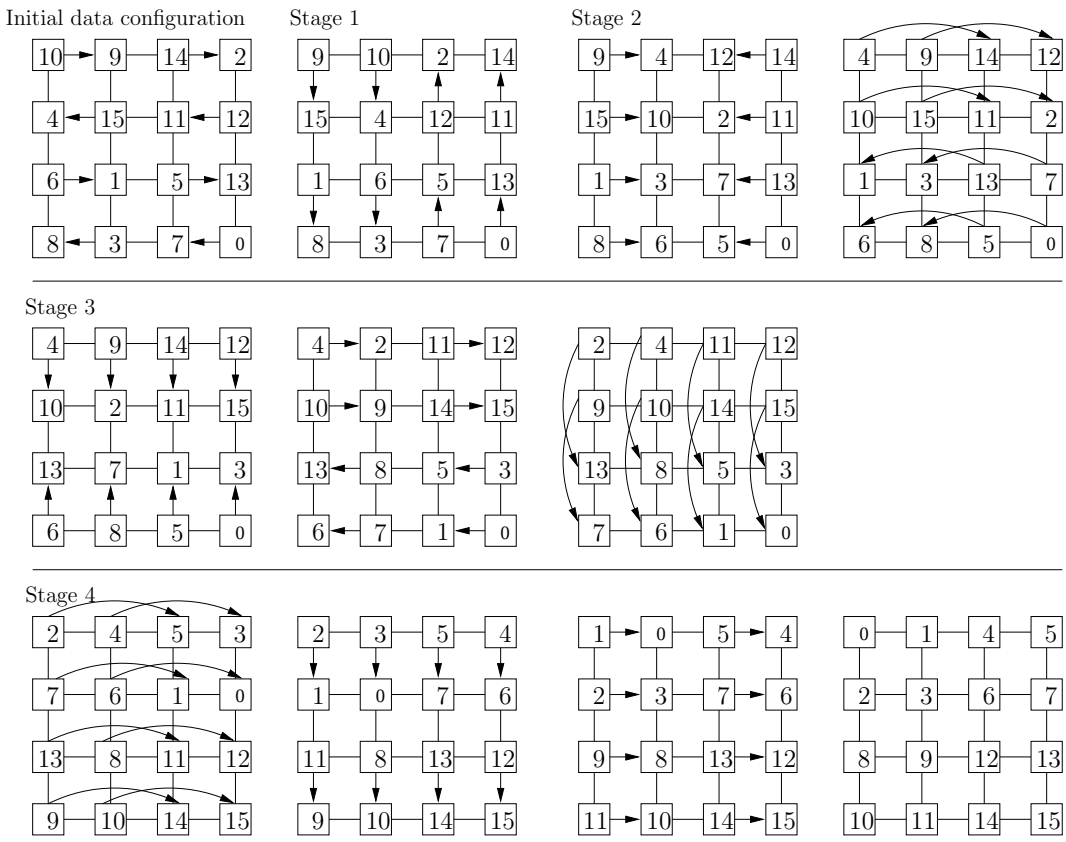


Figure 93: Sorting values into shuffled row-major order on the two-dimensional mesh processor array model (Fig.10-20, p.271)

a. Basic ideas. Since bitonic merge always compare elements whose indices differ in exactly one bit, and nodes in SIMD-CC model are connected if their indices differ in exactly one bit, bitonic merge is easily implemented on SIMD-CC model. Processors replace comparators. instead of routing pairs of elements to comparators, processors route data to adjacent processors, where the elements are compared. Assume that $n = 2^m$ elements are to sorted, $m > 0$.

b. **Example**. (Two self-prepared diagrams) Fig. 95 and 96.

c. The algorithm.

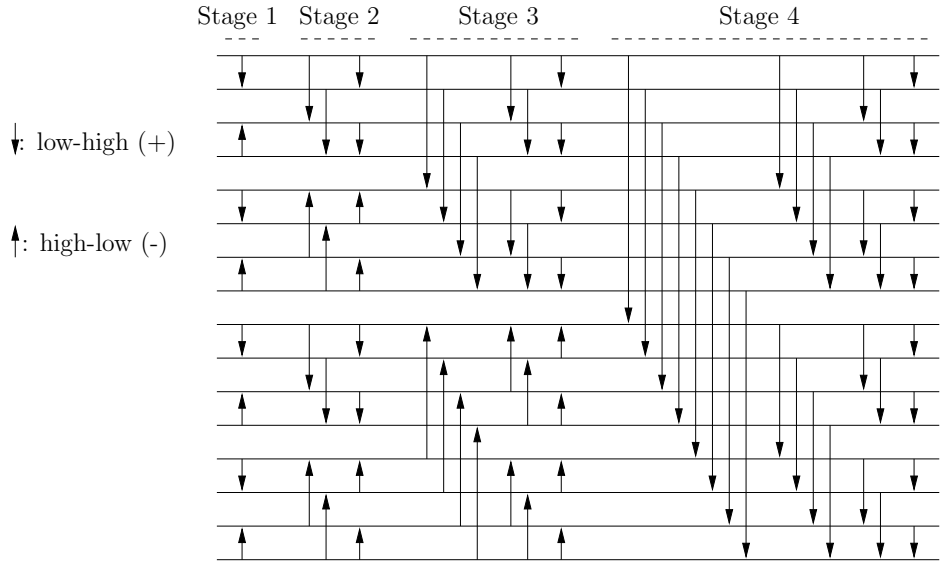


Figure 94: A sorting network based on bitonic merge (Knuth 1973) (Fig.10-18, p.269).

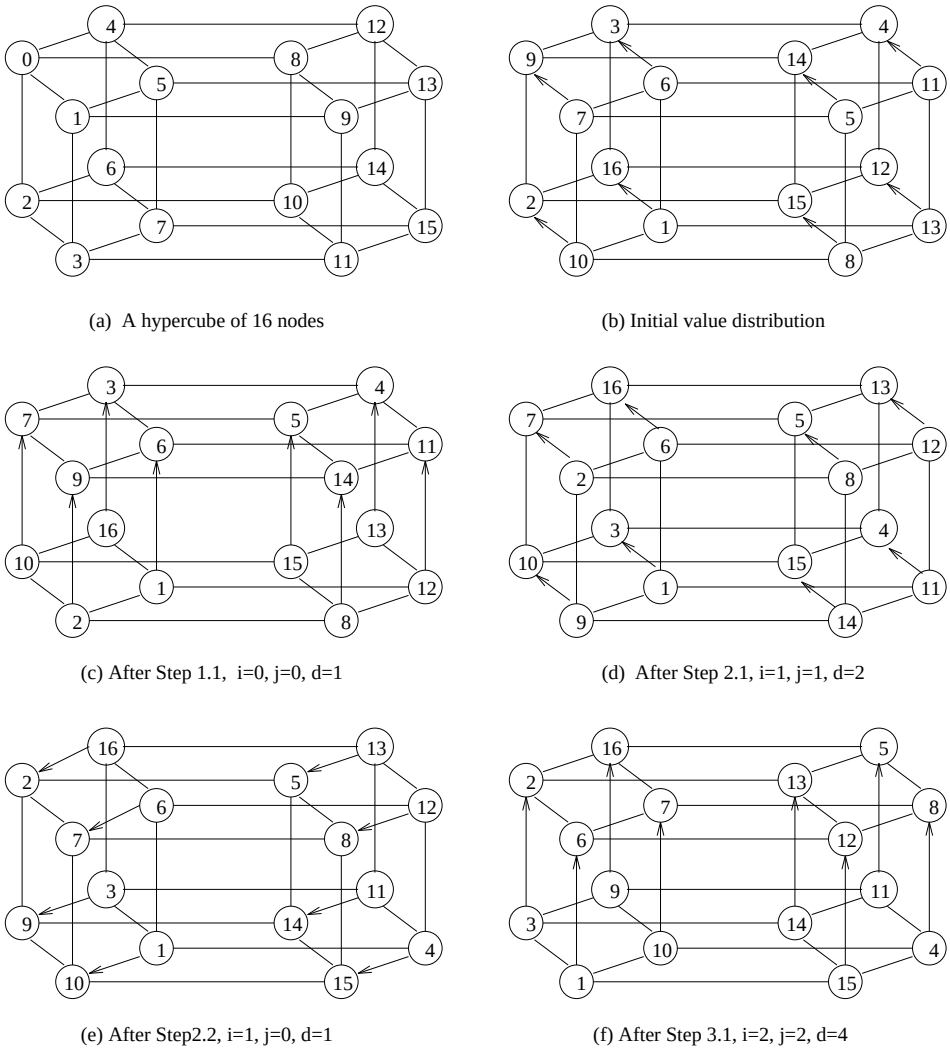
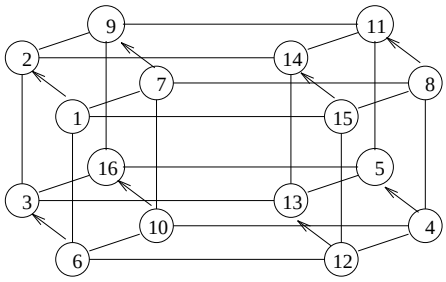
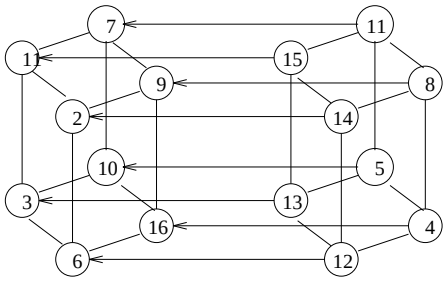


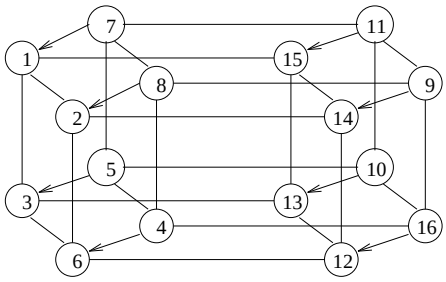
Figure 95: Part 1 of an example of sorting sixteen elements on a 4-dimension hypercube.



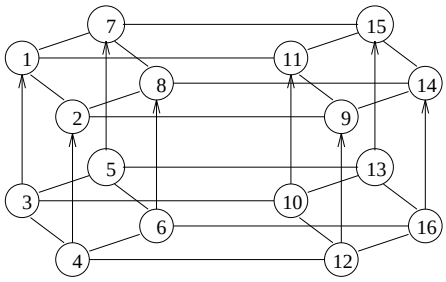
(g) After Step 3.2, i=2, j=1, d=2



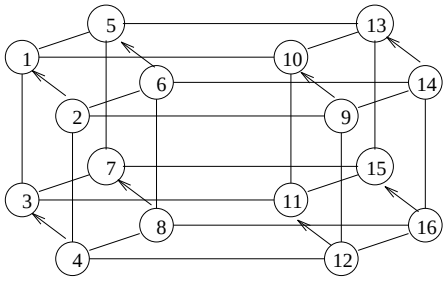
(h) After Step 3.3, i=2, j=0, d=1



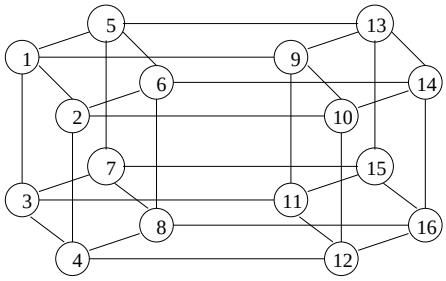
(i) After Step 4.1, i=3, j=3, d=8



(j) After Step 4.2, i=3, j=2, d=4



(k) After Step 4.3, i=3, j=1, d=1



(l) The final distribution, After Step 4.4, i=3, j=0, d=1

Figure 96: Part 2 of an example of sorting sixteen elements on a 4-dimension hypercube.

```
BITONIC MERGE SORT(HYPERCUBE PROCESSOR ARRAY):
    Global      d {Distance between elements being compared}
    Local       a {One of the elements to be sorted}
                t {Element retrieved from adjacent processor}
begin
    for i ← 0 to m − 1 do
        for j ← i downto 0 do
            d ← 2^j
            for all P_k, 0 ≤ k ≤ 2^m − 1, do
                if k mod 2d < d then
                    t ⇐ [k + d]a                    {Get value from adjacent processor}
                    if k mod 2^(i+2) < 2^(i+1) then
                        [k + d]a ⇐ max(t, a)        {Sort low to high}
                        a ← min(t, a)
                    else
                        [k + d]a ⇐ min(t, a)        {Sort high to low}
                        a ← max(t, a)
                    endif
                endif
            endfor
        endfor
    endfor
end
```

d. **Complexity:** The **for all** statement takes time $\Theta(1)$. Hence the complexity of this algorithm is $\Theta(m^2) = \Theta(\log^2 n)$.

(5) On SIMD-PS model – Stone's algorithm

a. Basic ideas.

* It again is easy to implement bitonic merge with the SIMD-PS model, because the the elements are shuffled to a pair of processors connected with an "exchange" link, instead of using comparators (the exchange connections in SIMD-PS model link processors whose bit representation differ in the least significant bit).
* The tricky part is to determine whether a particular pair of elements being compared is to be sorted low to high or high to low. Stone's algorithm uses a mask vector M to indicate the kind of sort to be done by a particular processor. A value 0 corresponds to a plus comparator, and a value 1 corresponds to a minus comparator.

b. The pseudo code: Fig.10-16, Fig.10-17, p.268, p.269

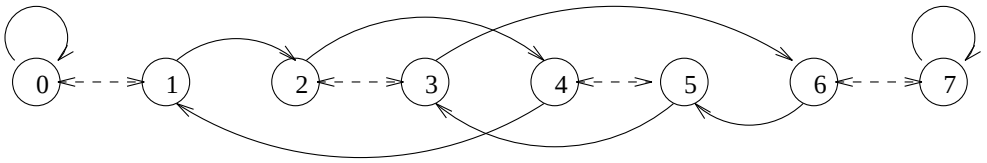
```
BITONIC MERGE SORT(SHUFFLE-EXCHANGE PROCESSOR ARRAY):
begin
    {Compute initial values of the mask M}
    for all P_i, 0 ≤ i ≤ n − 1, do
        r_i ← i mod 2
        m_i ← r_i
    endfor
    for i ← 1 to m do
        for all P_i, 0 ≤ i ≤ n − 1, do
            m_i ← m_i ⊕ r_i                {Exclusive OR}
            shuffle(m_i)
        endfor
    endfor
    COMPARE-EXCHANGE(A, M)
    for i ← 1 to m − 1 do
        for all P_i, 0 ≤ i ≤ n − 1, do
            shuffle(r_i)
            m_i ← m_i ⊕ r_i                {Exclusive OR}
            for j ← 1 to m − 1 − i do
                shuffle(a_i)
                shuffle(m_i)
            endfor
        endfor
        for j ← m − i to m do
            for all P_i, 0 ≤ i ≤ n − 1, do
                shuffle(a_i)
                shuffle(m_i)
            endfor
            COMPARE-EXCHANGE(A, M)
        endfor
    endfor
end
```

Figure 97: Implementation of bitonic merge-sort algorithm on the shuffle-exchange SIMD model (Fig.10-16, p.268).

```
COMPARE-EXCHANGE(A, M)
begin
    for all P_i, 0 ≤ i ≤ n − 1, do
        if even(i) then
            t_i ⇐ a_{i+1}
            if m_i = 0 then                {Sort low to high}
                b_i ← max(a_i, t_i)
                a_i ← min(a_i, t_i)
            else                           {Sort high to low}
                b_i ← min(a_i, t_i)
                a_i ← max(a_i, t_i)
            endif
        endif
        if odd(i) then
            a_i ⇐ b_{i−1}
        endif
    endfor
end
```

Figure 98: Compare-exchange routine called by bitonic merge sort algorithm for shuffle-exchange processor array. The even numbered processing elements assume the role of comparators (Fig.10-17, p.269).

c. An example:



15 8 3 12 14 16 11 9

(a) Initially 8 elements reside in a SIMD-PS mdel

$n = 8, m = \log n = 3$

Initialization, the first **for all** loop sets:

$r_0 = r_2 = r_4 = r_6 = 0,\ r_1 = r_3 = r_5 = r_7 = 1$

$m_0 = m_2 = m_4 = m_6 = 0,\ m_1 = m_3 = m_5 = m_7 = 1$

Initialization, the second **for** loop sets:

$i = 1$: $m_i \leftarrow m_i \oplus r_i$

$m_0 \leftarrow 0,\ m_1 \leftarrow 0,\ m_2 \leftarrow 0,\ m_3 \leftarrow 0,\ m_4 \leftarrow 0,\ m_5 \leftarrow 0,\ m_6 \leftarrow 0,\ m_7 \leftarrow 0$

$m_0 = 0,\ m_1 = 0,\ m_2 = 0,\ m_3 = 0,\ m_4 = 0,\ m_5 = 0,\ m_6 = 0,\ m_7 = 0$

$i = 2$: $m_0 \leftarrow 0,\ m_1 \leftarrow 1,\ m_2 \leftarrow 0,\ m_3 \leftarrow 1,\ m_4 \leftarrow 0,\ m_5 \leftarrow 1,\ m_6 \leftarrow 0,\ m_7 \leftarrow 1$

$m_0 = 0,\ m_1 = 0,\ m_2 = 1,\ m_3 = 1,\ m_4 = 0,\ m_5 = 0,\ m_6 = 1,\ m_7 = 1$

$i = 3$: $m_0 \leftarrow 0,\ m_1 \leftarrow 1,\ m_2 \leftarrow 1,\ m_3 \leftarrow 0,\ m_4 \leftarrow 0,\ m_5 \leftarrow 1,\ m_6 \leftarrow 1,\ m_7 \leftarrow 0$

$m_0 = 0,\ m_1 = 0,\ m_2 = 1,\ m_3 = 1,\ m_4 = 1,\ m_5 = 1,\ m_6 = 0,\ m_7 = 0$

Step 1: the first COMPARE-EXCHANGE(A, M)

$t_0 \leftarrow 8,\ t_2 \leftarrow 12,\ t_4 \leftarrow 16,\ t_6 \leftarrow 9$

$b_0 \leftarrow 15,\ b_2 \leftarrow 3,\ b_4 \leftarrow 14,\ b_6 \leftarrow 11$

$a_0 \leftarrow 8,\ a_2 \leftarrow 12,\ a_4 \leftarrow 16,\ a_6 \leftarrow 9$

8 15 12 3 16 14 9 11

(b) After Step 1 (first COMPARE EXCHANGE)

Step 2: $i = 1$ (for $i \leftarrow 1$ to $m - 1$ do)

The first **for all** loop in the main **for** loop:

Shuffle(r_i):

$r_0 = 0,\ r_1 = 0,\ r_2 = 1,\ r_3 = 1,\ r_4 = 0,\ r_5 = 0,\ r_6 = 1,\ r_7 = 1$

$m_i \leftarrow m_i \oplus r_i$

$m_0 = 0,\ m_1 = 0,\ m_2 = 0,\ m_3 = 0,\ m_4 = 1,\ m_5 = 1,\ m_6 = 1,\ m_7 = 1$

$j = 1$, (for $j \leftarrow 1$ to $m - 1 - i$ do)

Shuffle(a_i):

8 16 15 14 12 9 3 11

Shuffle(m_i)

$m_0 = 0,\ m_1 = 1,\ m_2 = 0,\ m_3 = 1,\ m_4 = 0, m_5 = 1,\ m_6 = 0,\ m_7 = 1$

Step2.1, the (**for** $j \leftarrow m - i$ to m do) loop in the main **for** loop, $j = 2$:

The inner **for all** loop:

Shuffle(a_i):

8 12 16 9 15 3 14 11

Shuffle(m_i)

$m_0 = 0,\ m_1 = 0,\ m_2 = 1,\ m_3 = 1,\ m_4 = 0,\ m_5 = 0,\ m_6 = 1,\ m_7 = 1$

COMPARE-EXCHANGE(A,M):

8 12 16 9 3 15 14 11

Step2.2, the (**for** $j \leftarrow m - i$ to m do) loop in the main **for** loop, $j = 3$:

The inner **for all** loop:

Shuffle(a_i):

8 3 12 15 16 14 9 11

Shuffle(m_i)

$m_0 = 0,\ m_1 = 0,\ m_2 = 0,\ m_3 = 0,\ m_4 = 1,\ m_5 = 1,\ m_6 = 1,\ m_7 = 1$

COMPARE-EXCHANGE(A,M):

⓪	①	②	③	④	⑤	⑥	⑦
3	8	12	15	16	14	11	9

(c) After Step 2

Step 3: $i = 2$

The first **for all** loop in the main **for** loop:

Shuffle(r_i):

$r_0 = 0,\ r_1 = 0,\ r_2 = 0,\ r_3 = 0,\ r_4 = 1,\ r_5 = 1,\ r_6 = 1,\ r_7 = 1$

$m_i \leftarrow m_i \oplus r_i$

$m_0 = 0,\ m_1 = 0,\ m_2 = 0,\ m_3 = 0,\ m_4 = 0,\ m_5 = 0,\ m_6 = 0,\ m_7 = 0$

The **for** loop inside the **for all** loop is bypassed

The (**for** $j \leftarrow m - i$ to m do) loop in the main **for** loop:

Step 3.1, $j = 1$, the inner **for all** loop:

Shuffle(a_i):

			14				
3	16	8		12	11	15	9

Shuffle(m_i): all m_i unchanged

COMPARE-EXCHANGE(A,M):

3	16	8	14	11	12	9	15

Step 3.2, $j = 2$, the inner **for all** loop:

Shuffle(a_i):

3	11	16	12	8	9	14	15

Shuffle(m_i): all m_i unchanged

COMPARE-EXCHANGE(A,M):

3	11	12	16	8	9	14	15

Step 3.3, $j = 3$, the inner **for all** loop:
Shuffle(a_i):

3	8	11	9	12	14	16	15

Shuffle(m_i): all m_i unchanged
COMPARE-EXCHANGE(A,M):

(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)
3	8	9	11	12	14	15	16

(d) After Step 3: the final result

d. **Complexity:** The algorithm requires $m(m+1)/2$ compare-exchange steps, $m(m-1)$ shuffle steps of the vector A, and $2m-1$ shuffle steps of the vector M and R. Since $m = \log n$, the time complexity of the algorithm is $\Theta(\log^2 n)$ with n processors.

4. Multiprocessor implementation of quicksort

(1) Sequential quicksort algorithm

a. Basic ideas of sequential quicksort algorithm

procedure Sequential-QuickSort(S)
begin
  if S contains at most one element **then**
    return
  endif
  Choose an element a randomly from S
  Let S_1, S_2, and S_3 be the sequences of elements in S less
    than, equal to, and greater than a, respectively
  Sequential-QuickSort(S_1)
  Sequential-QuickSort(S_3)
end

b. **Example.** $S = 2, 9, 4, 1, 3, 7, 6, 8, 10$

c. **Time complexity:** Assume that $n = 2^m - 1$ distinct values to be sorted. The number of comparisons (equal to the time needed in sequential algorithms) $T(n)$ is

$$T(n) = \begin{cases} 2T(\frac{n-1}{2}) + n - 1 & \text{for } n = 7, 15, 31, \cdots \\ 2 & \text{for } n = 3 \end{cases}$$

The solution to this recursive relation is

$$T(n) = (n+1)\log(n+1) - 2n$$

$$\begin{aligned}
T(n) &= 2T(\tfrac{n-1}{2}) + (n-1) \\
&= 2T(\tfrac{n}{2} - \tfrac{1}{2}) + (n-1) \\
&= 2^2 T(\tfrac{n}{2^2} - \tfrac{1}{2^2} - \tfrac{1}{2}) + 2(\tfrac{n}{2} - \tfrac{1}{2} - 1) + (n-1) \\
&= 2^2 T(\tfrac{n}{2^2} - \tfrac{1}{2^2} - \tfrac{1}{2}) + (n-1-2) + (n-1) \\
&= 2^3 T(\tfrac{n}{2^3} - \tfrac{1}{2^3} - \tfrac{1}{2^2} - \tfrac{1}{2}) + 2^2(\tfrac{n}{2^2} - \tfrac{1}{2^2} - \tfrac{1}{2} - 1) + (n-1-2) + (n-1) \\
&= 2^3 T(\tfrac{n}{2^3} - \tfrac{1}{2^3} - \tfrac{1}{2^2} - \tfrac{1}{2}) + (n-1-2-2^2) + (n-1-2) + (n-1) \\
&= 2^4 T(\tfrac{n}{2^4} - \tfrac{1}{2^4} - \tfrac{1}{2^3} - \tfrac{1}{2^2} - \tfrac{1}{2}) + (n-1-2-2^2-2^3) + (n-1-2-2^2) + \\
&\quad (n-1-2) + (n-1) \\
&\cdots\cdots \\
&= 2^j T(\tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i}) + (n - \Sigma_{i=0}^{j-1} 2^i) + (n - \Sigma_{i=0}^{j-2} 2^i) + \cdots + (n - \Sigma_{i=0}^{1} 2^i) + \\
&\quad (n - \Sigma_{i=0}^{0} 2^i) \\
&= 2^j T(\tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i}) + (n - (2^j - 1)) + (n - (2^{j-1} - 1)) + \cdots + \\
&\quad (n - (2^2 - 1)) + (n - (2^1 - 1)) \\
&= 2^j T(\tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i}) + (n + 1 - 2^j) + (n + 1 - 2^{j-1}) + \cdots + (n + 1 - 2^2) + \\
&\quad (n + 1 - 2^1) \\
&\quad \cdots\cdots \\
&\{\text{Let } q = \log(n+1) - 2, \text{ since let } \tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i} = 3,\ j = q\} \\
&= 2^q T(3) + (n + 1 - 2^q) + (n + 1 - 2^{q-1}) + \cdots + (n + 1 - 2^j) + \\
&\quad (n + 1 - 2^{j-1}) + \cdots + (n + 1 - 2^2) + (n + 1 - 2^1) \\
&= \tfrac{n+1}{2} + \Sigma_{i=1}^{q} [(n+1) - 2^i] \\
&= \tfrac{n+1}{2} + (n+1)q - \Sigma_{i=0}^{q} 2^i \\
&= \tfrac{n+1}{2} + (n+1)q - 2(2^q - 1) \\
&= \tfrac{n+1}{2} + (n+1)(\log(n+1) - 2) - 2(2^{\log(n+1)-2} - 1) \\
&= \tfrac{n+1}{2} + (n+1)(\log(n+1) - 2) - \tfrac{n+1}{2} + 2 \\
&= (n+1)\log(n+1) - 2n \ \square
\end{aligned}$$

(2) A parallel quicksort algorithm on UMA multiprocessors

a. Basic ideas: Fig.10-22, p.274.

(a) Assume $p = 2^m$ processors. A number of identical processes, one per processor, execute the parallel algorithm.

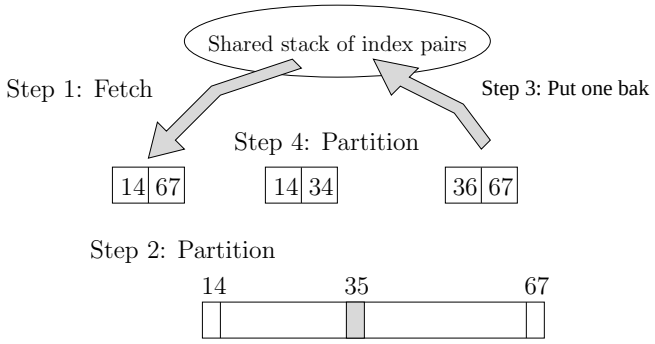
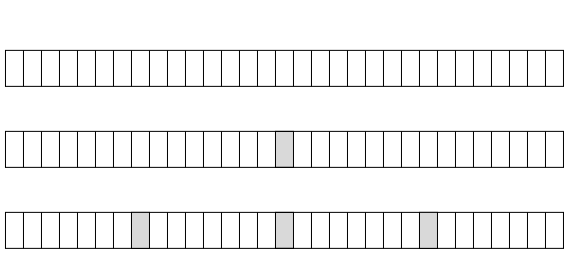


Figure 99: Illustration of a parallel quicksort algorithm for UMA multiprocessors. A process fetches the indices of an unsorted subarray from a shared stack of index pairs. The process uses the quicksort partitioning step to divide the interval into two subintervals and a median. The process puts the indices of one unsorted subinterval back on the stack and keeps the other index pair, repeating the partitioning step (Fig.10-22, p.274).



	Comparisons	Processors	Time
	$n-1$	1	$n-1$
	$n-3$	2	$(n-3)/2$
	$n-7$	4	$(n-7)/4$

Figure 100: A view of the beginning of parallel quicksort, when the number of unsorted intervals is less than or equal to the number of processes. The analysis assume $n = 2^k - 1$ and every partitioning step divides an interval exactly in half (Fig.10-23, p.274).

(b) The elements to be sorted are stored in a global array.

(c) A stack S in global memory stores the indices of subarrays that are to be sorted. Initially, $S = 1, n$.

(d) When a process is without work it attempts to pop the indices for an unsorted subarray off the stack. It the popping action is successful, the process partitions the subarray into two smaller arrays. After the partitioning step, the process pushes the indices of one of the smaller arrays onto the stack. It then continues to partitioning steps to the other smaller array.

b. Code: Fig.10-24, p.276. Notes: processes must execute functions STACK.DELETE(), ADD.TO.SORTED(), and STACK.INSERT() inside critical sections.

```
QUICKSORT(UMA MULTIPROCESSOR):
    Global   n              {Size of array of unsorted elements}
             a[0...(n − 1)] {Array of elements to be sorted}
             sorted         {Number of elements in sorted position}
             min.position   {Smallest subarray that is partitioned rather
                            than stored directly}
    Local    bounds         {Indices of unsorted subarray}
             median         {Final position in subarray of partitioning key}
begin
    sorted ← 0
    INITIALIZE.STACK()
    for all P_i, where 0 ≤ i < p do
        while (sorted < n) do
            bounds ← STACK.DELETE()
            while (bounds.low < bounds.high) do
                if (bounds.high − bounds.low) < min.position) then
                    INSERTION.SORT(a, bounds.low, bounds.high)
                    ADD.TO.SORTED(bounds.high − bounds.low + 1)
                    exit while
                else
                    median ← PARTITION(bounds.low, bounds.high)
                    STACK.INSERT(median +1, bounds.high)
                    bounds.high ← median − 1
                    if (bounds.low = bounds.high) then
                        ADD.TO.SORTED(2)
                    else
                        ADD.TO.SORTED(1)
                    endif
                endif
            endwhile
        endwhile
    endfor
end
```

c. *Time complexity:* The entire computation can be divided into two phases:

   (a) In the 1st phase, there are more processes than subarrays (Fig.10-23);

   (b) In the 2nd phase, there are more subarrays than processes.

   (c) The first iteration needs time $n-1$ to perform $n-1$ comparisons, resulting two subarrays of size $(n-1)/2$ each.

   (d) In the second iteration, two processes can partition the two subarrays in time $(n-1)/2-1$, resulting four subarrays of size $[(n-1)/2-1]/2$.

The total number of comparisons is $n-3$.

(e) Similarly, the third iteration needs time $[(n-1)/2-1]/2-1$, and the total number of comparisions is $n-7$.

The total time needed in phase 1 is

$$T_1(n) = T_1(\frac{n-1}{2}) + n - 1, n > p$$

The solution to this recursive relation is

$$T_1(n) = 2(n+1)(1-\frac{1}{p}) - 2\log p$$

$$\begin{aligned}
T_1(n) &= T(\tfrac{n-1}{2}) + (n-1) \\
&= T_1(\tfrac{n}{2} - \tfrac{1}{2}) + (n-1) \\
&= T_1(\tfrac{n}{2^2} - \tfrac{1}{2^2} - \tfrac{1}{2}) + (\tfrac{n}{2} - \tfrac{1}{2} - 1) + (n-1) \\
&= T_1(\tfrac{n}{2^3} - \tfrac{1}{2^3} - \tfrac{1}{2^2} - \tfrac{1}{2}) + (\tfrac{n}{2^2} - \tfrac{1}{2^2} - \tfrac{1}{2} - 1) + (\tfrac{n}{2} - \tfrac{1}{2} - 1) + (n-1) \\
&\quad \cdots\cdots \\
&= T_1(\tfrac{n}{2^j} - \Sigma_{i=1}^{j}\tfrac{1}{2^i}) + (\tfrac{n}{2^{j-1}} - \Sigma_{i=0}^{j-1}\tfrac{1}{2^i}) + (\tfrac{n}{2^{j-2}} - \Sigma_{i=0}^{j-2}\tfrac{1}{2^i}) + \cdots + (\tfrac{n}{2} - \Sigma_{i=0}^{1}\tfrac{1}{2^i}) + \\
&\quad (\tfrac{n}{2^0} - \Sigma_{i=0}^{0}\tfrac{1}{2^i}) \\
&= T_1(\tfrac{n}{2^j} - \Sigma_{i=1}^{j}\tfrac{1}{2^i}) + (\tfrac{n}{2^{j-1}} - 2(1-\tfrac{1}{2^j})) + (\tfrac{n}{2^{j-2}} - 2(1-\tfrac{1}{2^{j-1}})) + \cdots + \\
&\quad (\tfrac{n}{2} - 2(1-\tfrac{1}{2^2})) + (n - 2(1-\tfrac{1}{2^1})) \\
&= T_1(\tfrac{n}{2^j} - \Sigma_{i=1}^{j}\tfrac{1}{2^i}) + (\tfrac{n}{2^{j-1}} - 2 + \tfrac{1}{2^{j-1}}) + (\tfrac{n}{2^{j-2}} - 2 + \tfrac{1}{2^{j-2}}) + \cdots + (\tfrac{n}{2} - 2 + \tfrac{1}{2^2}) + \\
&\quad (n - 2 + 1) \\
&\cdots\cdots \\
&= T_1(\tfrac{n}{2^m} - \Sigma_{i=1}^{m}\tfrac{1}{2^i}) + (\tfrac{n}{2^{m-1}} - 2 + \tfrac{1}{2^{m-1}}) + (\tfrac{n}{2^{m-2}} - 2 + \tfrac{1}{2^{m-2}}) + \cdots + \\
&\quad (\tfrac{n}{2} - 2 + \tfrac{1}{2^2}) + (n - 2 + 1)
\end{aligned}$$

$$\begin{aligned}
&(\tfrac{n}{2^{m-1}} - 2 + \tfrac{1}{2^{m-1}}) + (\tfrac{n}{2^{m-2}} - 2 + \tfrac{1}{2^{m-2}}) + \cdots + (\tfrac{n}{2} - 2 + \tfrac{1}{2^2}) + (n - 2 + 1) \\
&= n\Sigma_{i=0}^{m-1}\tfrac{1}{2^i} - 2m + \Sigma_{i=0}^{m-1}\tfrac{1}{2^i} \\
&= (n+1)\Sigma_{i=0}^{m-1}\tfrac{1}{2^i} - 2m \\
&= 2(n+1)(1-\tfrac{1}{2^m}) - 2m \\
&= 2(n+1)(1-\tfrac{1}{p}) - 2\log p
\end{aligned}$$

The number of comparisons $C_1(n)$ is

$$C_1(n) = 2C_1(\frac{n-1}{2}) + n - 1, n > p$$

The solution to this recursive relation is

$$C_1(n) = (n+1)\log p - 2(p-1)$$

$$
\begin{aligned}
C_1(n) &= 2C_1(\tfrac{n}{2} - \tfrac{1}{2}) + (n-1) \\
&= 2^2 C_1(\tfrac{n}{2^2} - \tfrac{1}{2^2} - \tfrac{1}{2}) + 2(\tfrac{n}{2} - \tfrac{1}{2} - 1) + (n-1) \\
&= 2^3 C_1(\tfrac{n}{2^3} - \tfrac{1}{2^3} - \tfrac{1}{2^2} - \tfrac{1}{2}) + (n-1-2-2^2) + (n-1-2) + (n-1) \\
&\cdots\cdots \\
&= 2^j C_1(\tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i}) + (n - \Sigma_{i=0}^{j-1} 2^i) + (n - \Sigma_{i=0}^{j-2} 2^i) + \cdots + (n - \Sigma_{i=0}^{1} 2^i) + \\
&\quad (n - \Sigma_{i=0}^{0} 2^i) \\
&= 2^j C_1(\tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i}) + (n - (2^j - 1)) + (n - (2^{j-1} - 1)) + \cdots + \\
&\quad (n - (2^2 - 1)) + (n - (2^1 - 1)) \\
&= 2^j C_1(\tfrac{n}{2^j} - \Sigma_{i=1}^{j} \tfrac{1}{2^i}) + (n + 1 - 2^j) + (n + 1 - 2^{j-1}) + \cdots + (n + 1 - 2^2) + \\
&\quad (n + 1 - 2^1) \\
&\quad \cdots\cdots
\end{aligned}
$$

{We Only need consider the first m items}

$$
\begin{aligned}
&(n + 1 - 2^m) + (n + 1 - 2^{m-1}) + \cdots + (n + 1 - 2^2) + (n + 1 - 2^1) \\
&\quad = \Sigma_{i=1}^{m} [(n+1) - 2^i] \\
&\quad = (n+1)m - \Sigma_{i=1}^{m} 2^i \\
&\quad = (n+1)m - 2(2^m - 1) \\
&\quad = (n+1)\log p - 2(p-1) \ \square
\end{aligned}
$$

$$C_2(n) = T(n) - C_1(n)$$

$$T_2(n) = \frac{C_2(n)}{p}$$

$$\text{Speedup} \ = \ \frac{T(n)}{T_1(n) + T_2(n)}$$

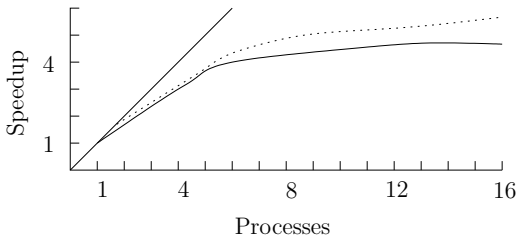


Figure 101: Predicted speedup (dashed line) and actual speedup (solid line) of parallel quicksort algorithm. Actual speedup data collected on a lightly loaded 20-processor Sequent Symmetry running the PTX operating systems (Fig.10-25, p.277).

d. Discussions.

(a) For the above estimated speedup, when for $n = 1023$ and $p = 8$, the best speedup is about 3.375. When for $n = 65,535$ and $p = 16$, the best speedup would be about 5.6.

(b) Fig.10-25 compares the estimated speedup and the actual speedup for the algorithm running on a lightly loaded 20-node Sequent Symmetry multiprocessor computer.

(c) Why is the speedup so low? Due to the way the processors work. This is an algorithm based on the divide-conquer strategy. It has a *cold start*. Too many processors remain idle at the initial stages.

(3) Hyperquicksort

a. It is a parallel sorting algorithm on hypercube multicomputers. It is an algorithm that will allow all processors to be active right at the beginning of the whole computation

b. Basic ideas. Assume that m elements are to be sorted on a d-dimension hypercube with $p = 2^d$ processors. Initially the m elements are distributed evenly across p processors, m/p elements per processor. The algorithm consists of two phases.

(a) In the first phase, each processor sorts its own list of elements using the sequential quicksort algorithm.

(b) The second phase is a repeated application of the following *split-and-merge* strategy, over *decreasing* dimensions of the hypercube.

i. The first processor P_0 broadcasts its median value α to every other processor in the whole hypercube.

ii. Upon receiving the median value from processor P_0, each processor in the lower half of the hypercube will send values greater than the median value to the corresponding processor in the upper half. Similarly each processor in the upper half will send values less than or equal to the median value to the corresponding processor in the lower half.

iii. Each processor will perform a sequential merge operation on the values it still keeps and the values it received from its corresponding processor. Therefore after the exchange-sort operations each processor again will have a list of sorted values. However, the number of elements each processor has can change.

vi. The above operations will be repeated in the two $d-1$ dimension hypercubes, and then the four $d-2$ dimension hypercubes.

The second phase will have a total d phases for a d-dimension hypercube.

c. Example: Fig.10-26, p.278. There are 32 values initialized distributed among 4 processors, 8 elements per processor. This is a dimension-2 hypercube.

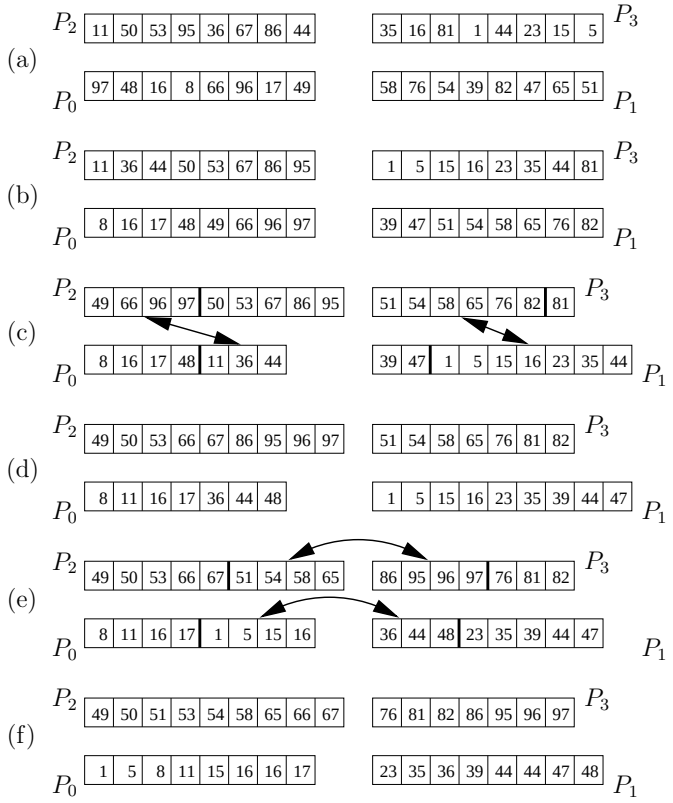


Figure 102: Illustration of the hyperquicksort algorithm. In this example 32 elements are being sorted on a two-dimesional hypercube. (a) Initially each processor has eight elements. (b) Each processor performs a sequential quicksort on its own list. Processor 0 broadcasts its median value, 48, to the other processors. (c) Processors in the lower half of the hypercube send values greater than 48 to processors in the upper half of the hypercube. The processors in the upper half send values less than or equal to 48. (d) Each processor merges the elements it kept with the elements it received. Processor 0 broadcasts its median value to processor 1, and processor 2 broadcasts its median value to processor 3. (e) Processors swap values across another hypercube dimension. (f) Each processor merges its elements it kept with the elements it received. At this point the list is sorted. (Fig.10-26, p.278)

d. Pseudo code: Fig.10-27, p.279

```
HYPERQUICKSORT(HYPERCUBE MULTICOMPUTER):
    Global   n            {Initial number of elements per processor}
             d            {Dimension of hypercube}
             i            {Dimension number of current hypercube}
    Local    logical.num  {Unique processor number}
             partner      {Processor's partner in the exchange}
             root         {Root processor of current hypercube}
             splitter     {Median of root processor's sorted list}
begin
    for all P_j, where 0 ≤ j < 2^d do
        Sort n values using sequential quicksort algorithm
        if d > 0 then
            for i ← d downto 1 do
                root ← root of the binary i-cube containing processor logical.num
                if (logical.num = root) then
                    splitter ← median of the stored list held by processor logical.num
                endif
                Processor root broadcasts splitter to other processors in binary i cube
                Use splitter to partition sorted values into low list and high list
                partner ← logical.num ⊗ 2^(i-1)    { Bitwise exclusive "or"}
                if (logical.num > partner) then
                    Send low list to processor partner
                    Receive another high list from processor partner
                else  {logical.num < partner}
                    Send high list to processor partner
                    Receive another low list from processor partner
                endif
                Merge two lists into a single sorted list of values
            endfor
        endif
    endfor
end
```

e. Complexity of the hyperquicksort

(a) Computational time

i. Assume that each processor initially is given n values. Then the initial sequential quicksort needs time $\Theta(n \log n)$.

ii. Assume that the chosen median value will always split values at each processor into two equal portions: half of them, $n/2$, are kept

and half are sent. Then the first merge operation will take time $\Theta(n)$.

iii. As the split-merge operations are repeated over hypercube of dimensions $d, d-1, \cdots, 2, 1$, the total time needed for split merge will be $n + n/2 + \cdots + 2 + 1$ which is $\Theta(nd)$. Hence the total time needed by the algorithm is $\Theta(n(\log n + d))$.

(b) Communication cost

i. Let λ denote the message latency (message initiation time) and β denote the time of transmitting a message between two adjacent nodes in a hypercube. For a d-dimensional hypercube, broadcast a median value (the splitter) induces communication time $d(\lambda + \beta)$.

ii. Assume that each processor passes half of its values. The time needed for sending $n/2$ values and receiving $n/2$ values from the partner processor is $2\lambda + n\beta$. The (total) expected communication time for the split-merge phase is

$$\Sigma_{i=1}^{d}(i(\lambda + \beta) + 2\lambda + n\beta)$$
$$= d(d+1)(\lambda + \beta)/2 + d(2\lambda + n\beta)$$

(c) Example: Fig.10-28.

(a) Sequential quicksort implemented in C on the nCUBE 3200 requires 12.2 microseconds per comparison. $\lambda = 500$ microseconds andn $\beta = 11$ microseconds.

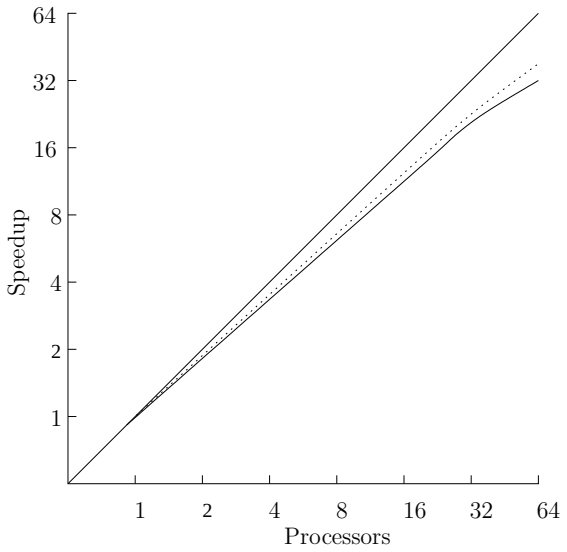


Figure 103: Predicted speedup (dashed line) and actual speedup (solid line) of hyperquicksort algorithm sorting 16,384 integers on an nCUBE 3200 hypercube multicomputer (Fig.10-28, p.280).

(b) The total number of integers $m = 16,384$.

(c) Fig.10-28 shows the expected speedup and the actual speedup with various number of processors used.

f. Modified hyperquicksort

(a) The selection of the right *splitter* is critical to the performance of the hyperquicksort algorithm.

(b) One method to select a good splitter value before each split-merge step is to compute the mean value of all median values in a subhypercube. This resulted in the *modified hyperquicksort* algorithm

(c) However, experimentations show that the modified hyperquicksort algorithm was outperformed by the original hyperquicksort algorithm, because the induced communication cost outweighs the the gain of choosing better splitter values.