Parallel Sorting

Leiserson's Merge Sort using Cilk

- See Cilk Lecture 2 slides

Review of Master Recurrence-Solving Method

The ***Master Method*** for solving recurrences applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,^{*}$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

IDEA: Compare $n^{\log_b a}$ with $f(n)$.

*The unstated base case is $T(n) = \Theta(1)$ for sufficiently small n.

source: Leiserson, Cilk Lec. 2

Master Method — Case 1

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \gg f(n)$$

Specifically, $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

Solution: $T(n) = \Theta(n^{\log_b a})$.

source: Leiserson, Cilk Lec. 2

Master Method — Case 2

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \approx f(n)$$

Specifically, $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

source: Leiserson, Cilk Lec. 2

Review of Master Recurrence-Solving Method

Master Method — CASE 3

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \ll f(n)$$

Specifically, $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ *and* $f(n)$ satisfies the ***regularity condition*** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.

Solution: $T(n) = \Theta(f(n))$.

source: Leiserson, Cilk Lec. 2

Master Method Summary

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$,
and regularity condition
$\Rightarrow T(n) = \Theta(f(n))$.

source: Leiserson, Cilk Lec. 2

Master Method Examples

- $T(n) = 4\,T(n/2) + n$

  $n^{\log_b a} = n^2 \gg n \Rightarrow$ **CASE 1:** $T(n) = \Theta(n^2)$.

- $T(n) = 4\,T(n/2) + n^2$

  $n^{\log_b a} = n^2 = n^2 \lg^0 n \Rightarrow$ **CASE 2:** $T(n) = \Theta(n^2 \lg n)$.

- $T(n) = 4\,T(n/2) + n^3$

  $n^{\log_b a} = n^2 \ll n^3 \Rightarrow$ **CASE 3:** $T(n) = \Theta(n^3)$.

- $T(n) = 4\,T(n/2) + n^2/\lg n$

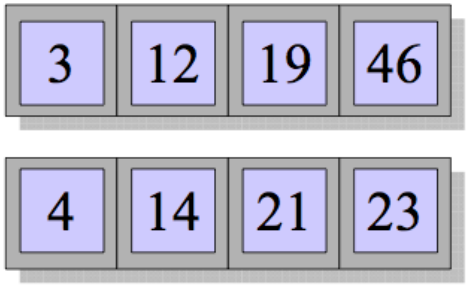
  Master method does not apply!

source: Leiserson, Cilk Lec. 2

Sequential Merging

```
void Merge(int *C, int *A, int *B, int na, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  }
}
```

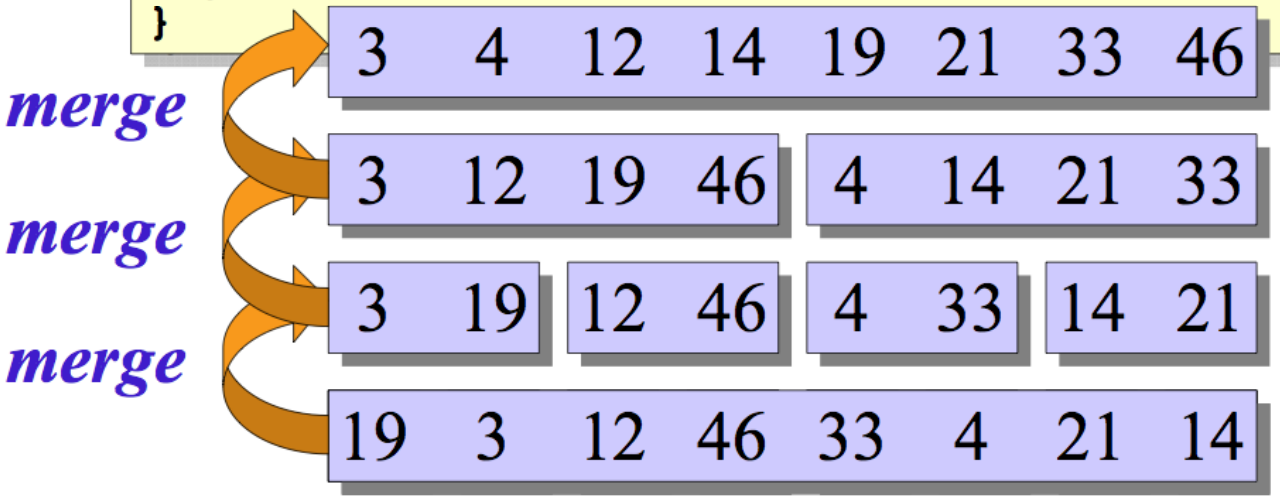
Time to merge n elements $= \Theta(n)$.



source: Leiserson, Cilk Lec. 2

Parallel Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```



source: Leiserson, Cilk Lec. 2

Work

```
cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$ — **CASE 2** of Master Formula

$n^{\log_b a} = n^{\log_2 2} = n \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^0 n)$.

source: Leiserson, Cilk Lec. 2

Span

```
cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

$$\textbf{\textit{Span:}}\ T_\infty(n) = T_\infty(n/2) + \Theta(n)$$
$$= \Theta(n) \quad \text{— CASE 3}$$

of Master Formula

$$n^{\log_b a} = n^{\log_2 1} = 1 \ll \Theta(n)\ .$$

source: Leiserson, Cilk Lec. 2

Parallelism

$$\textbf{\textit{Work:}} \quad T_1(n) = \Theta(n \lg n)$$

$$\textbf{\textit{Span:}} \quad T_\infty(n) = \Theta(n)$$

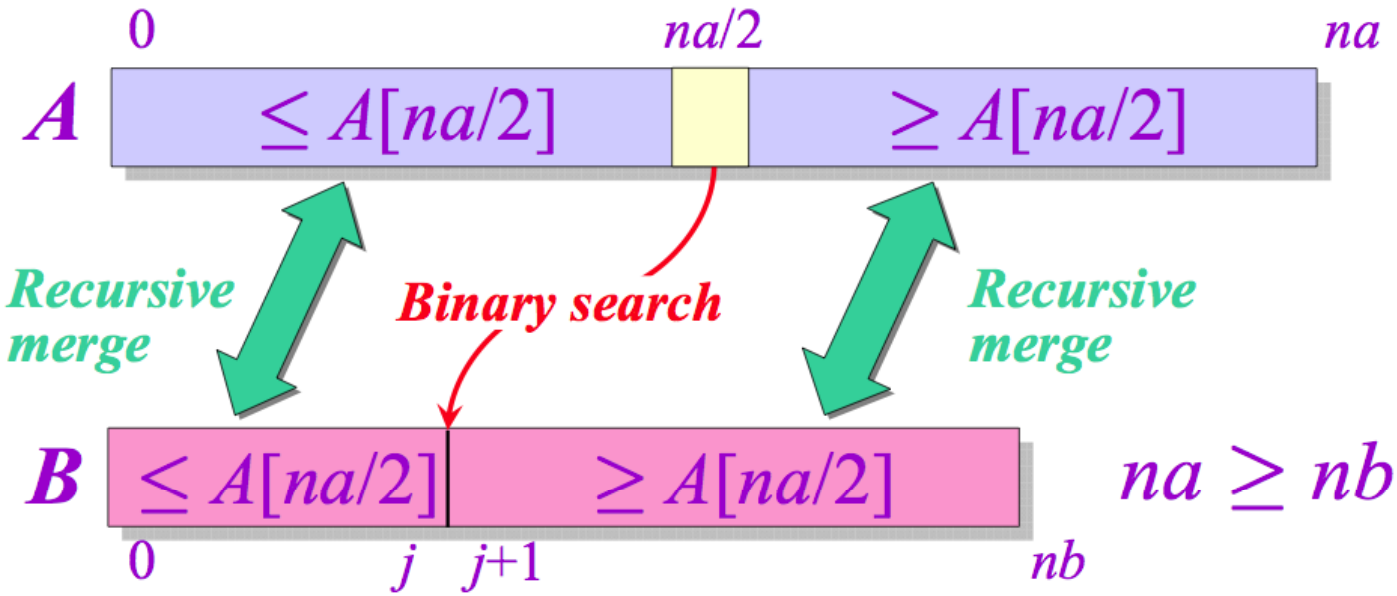
PUNY!

$$\textbf{\textit{Parallelism:}} \quad \frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$$

*We need to parallelize the **merge**!*

source: Leiserson, Cilk Lec. 2

Parallelizing Merge



KEY IDEA: If the total number of elements to be merged in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)n$.

source: Leiserson, Cilk Lec. 2

Parallel Merge Algorithm

```
cilk void P_Merge(int *C, int *A, int *B,
                                  int na, int nb) {
  if (na < nb) {
    spawn P_Merge(C, B, A, nb, na);
  } else if (na==1) {
    if (nb == 0) {
      C[0] = A[0];
    } else {
      C[0] = (A[0]<B[0]) ? A[0] : B[0]; /* minimum */
      C[1] = (A[0]<B[0]) ? B[0] : A[0]; /* maximum */
    }
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    spawn P_Merge(C, A, B, ma, mb);
    spawn P_Merge(C+ma+mb, A+ma, B+mb, na-ma, nb-mb);
    sync;
  }
}
```

Coarsen base cases for efficiency.

source: Leiserson, Cilk Lec. 2

P_Merge Span

```
cilk void P_Merge(int *C, int *A, int *B,
                                  int na, int nb) {
  if (na < nb) {
  :
  :
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    spawn P_Merge(C, A, B, ma, mb);
    spawn P_Merge(C+ma+mb, A+ma, B+mb, na-ma, nb-mb);
    sync;
  }
}
```

$$\textbf{\textit{Span:}}\ T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$$
$$= \Theta(\lg^2 n) \quad \text{— CASE 2}$$

$$n^{\log_b a} = n^{\log_{4/3} 1} = 1 \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^1 n)\ .$$

source: Leiserson, Cilk Lec. 2

P_Merge Work

```
cilk void P_Merge(int *C, int *A, int *B,
                  int na, int nb) {
  if (na < nb) {
  :
  :
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    spawn P_Merge(C, A, B, ma, mb);
    spawn P_Merge(C+ma+mb, A+ma, B+mb,
    sync;
  }
}
```

HAIRY!

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \leq \alpha \leq 3/4$.

CLAIM: $T_1(n) = \Theta(n)$.

analysis in Leiserson, Cilk Lec. 2

source: Leiserson, Cilk Lec. 2

P_Merge Parallelism

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$

source: Leiserson, Cilk Lec. 2

Parallel Merge Sort

```
cilk void P_MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn P_MergeSort(C, A, n/2);
    spawn P_MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    spawn P_Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

source: Leiserson, Cilk Lec. 2

Work of Parallel Merge Sort

```
cilk void P_MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn P_MergeSort(C, A, n/2);
    spawn P_MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    spawn P_Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

Work: $T_1(n) = 2\,T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$ — **CASE 2**

source: Leiserson, Cilk Lec. 2

Span of Parallel Merge Sort

```
cilk void P_MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn P_MergeSort(C, A, n/2);
    spawn P_MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    spawn P_Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

$$\textbf{\textit{Span:}}\ T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$$
$$= \Theta(\lg^3 n) \quad \text{— CASE 2}$$

$$n^{\log_b a} = n^{\log_2 1} = 1 \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^2 n)\ .$$

source: Leiserson, Cilk Lec. 2

Parallelism of Parallel Merge Sort

Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$

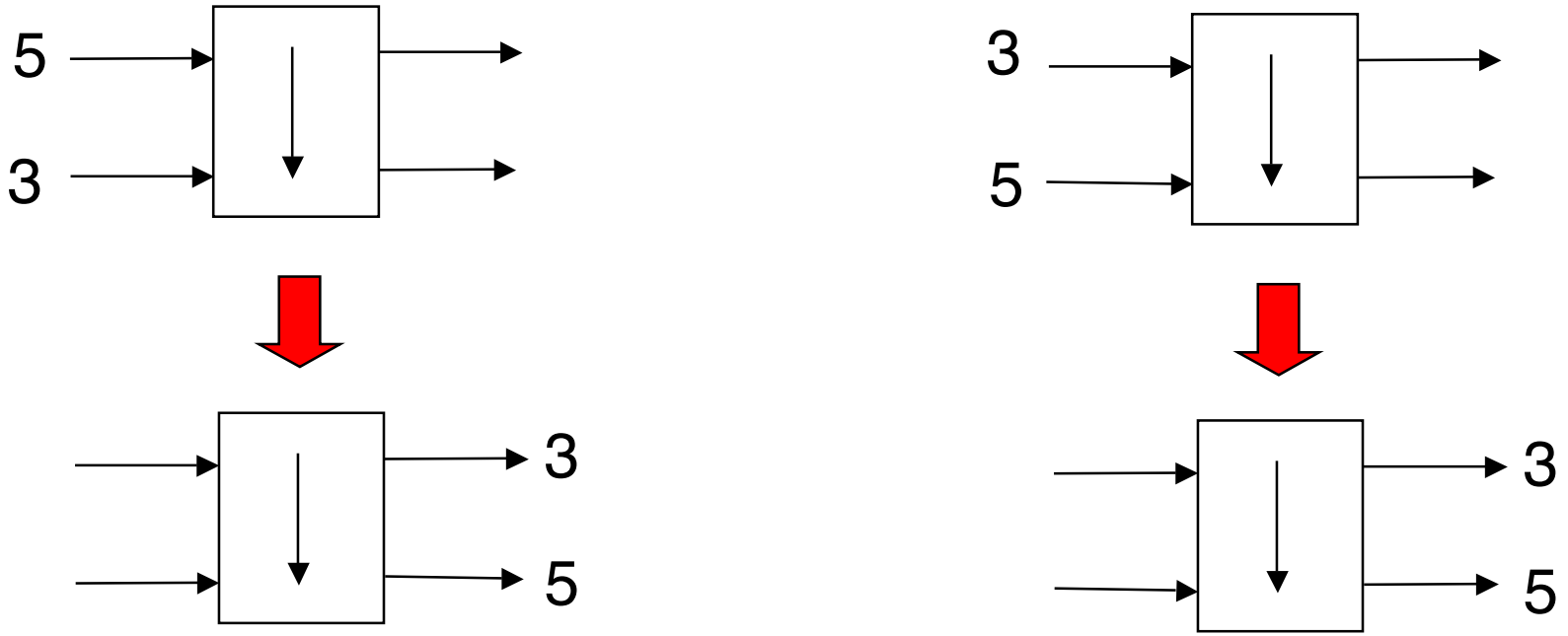
source: Leiserson, Cilk Lec. 2

Sorting Networks

- **Sorting networks** were first proposed by Kenneth Batcher in the 1960's.
- These networks can be constructed from a number of simple devices, called **comparators**.
- Software versions are also possible.

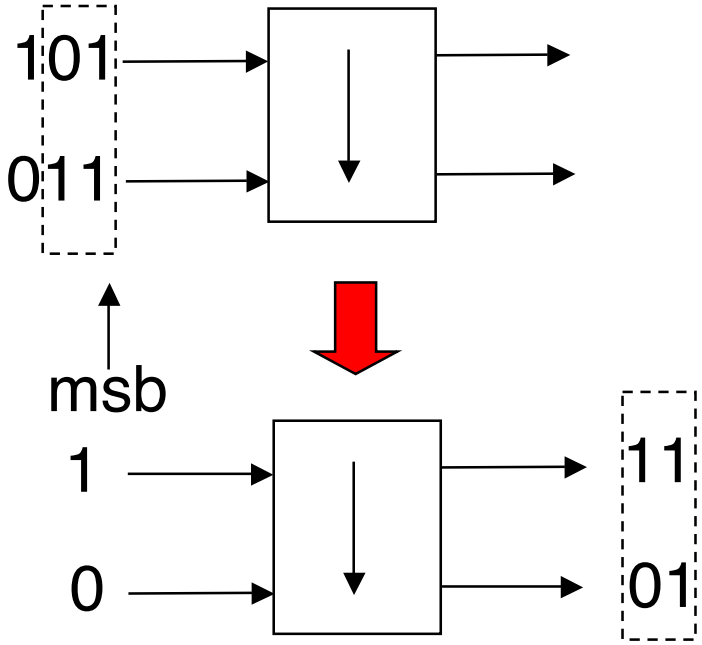
Sorting Network Comparator

- A single comparator inputs two numbers and outputs them in sorted order.



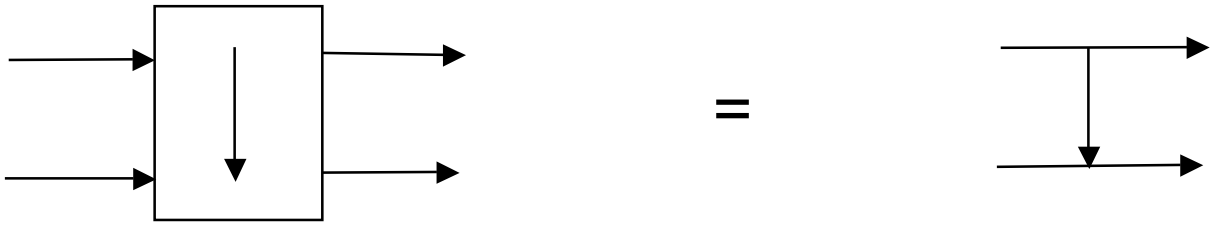
Sorting Network Comparator

- If desired, a comparator *can* be **pipelined** at the *bit-level*, as a finite-state machine accepting inputs MSB first (assuming the same number of bits in both numbers:

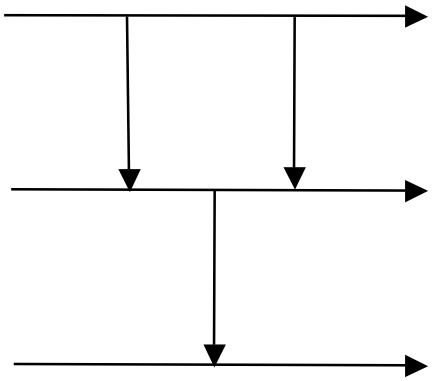


Sorting Networks

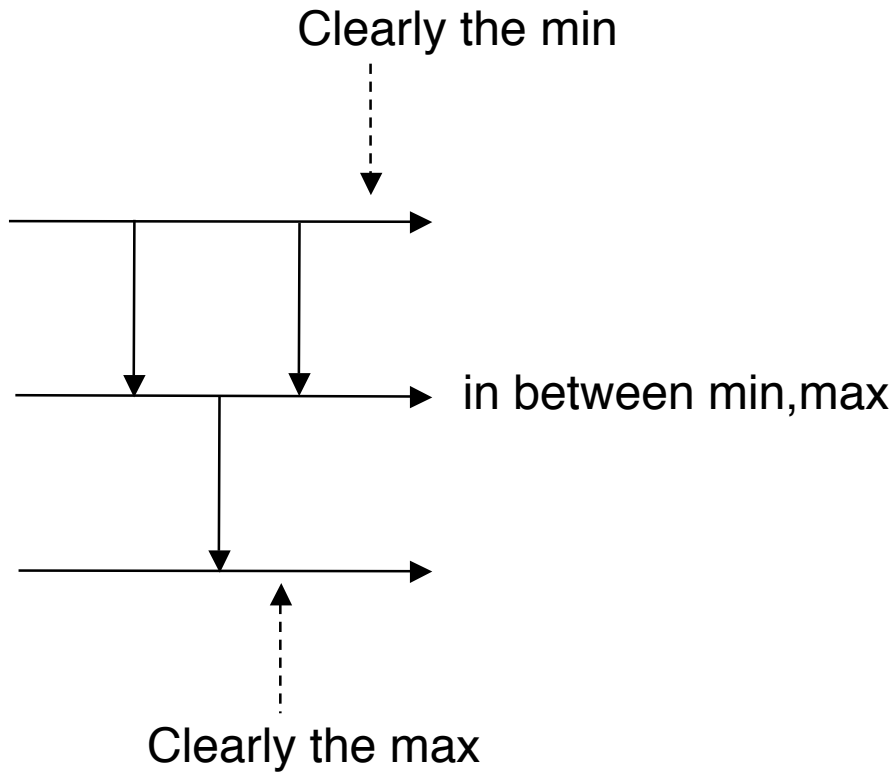
- Comparator abstraction (for drawing networks)



- A network that sorts 3 numbers



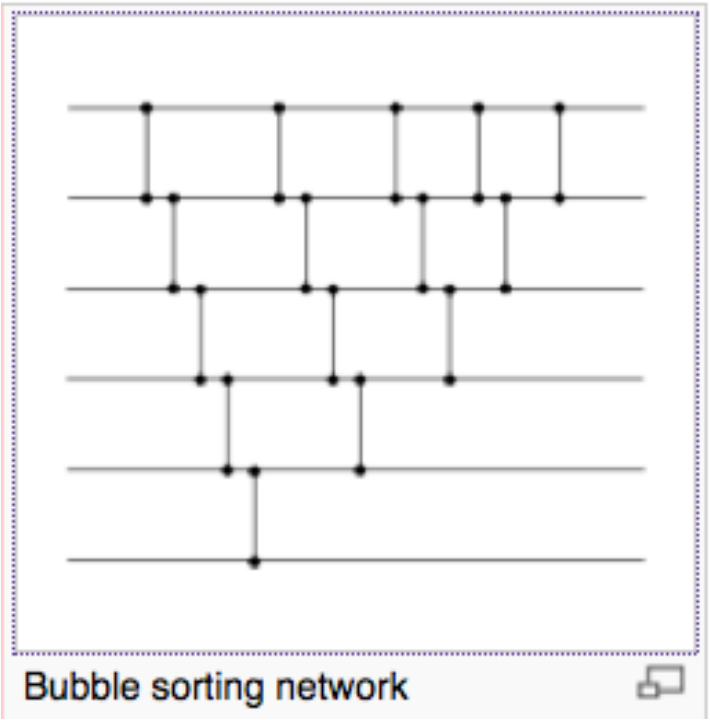
Analysis



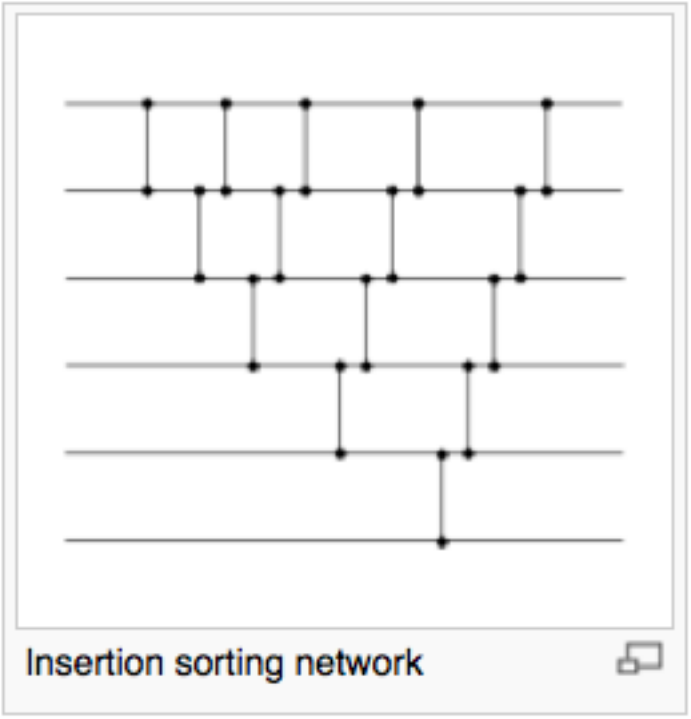
Exercises

- Construct a sorting network for 4 numbers
- Construct a sorting network for 8 numbers

(Non-Optimal) Sorters for Any Number of Lines



Bubble sorting network



Insertion sorting network

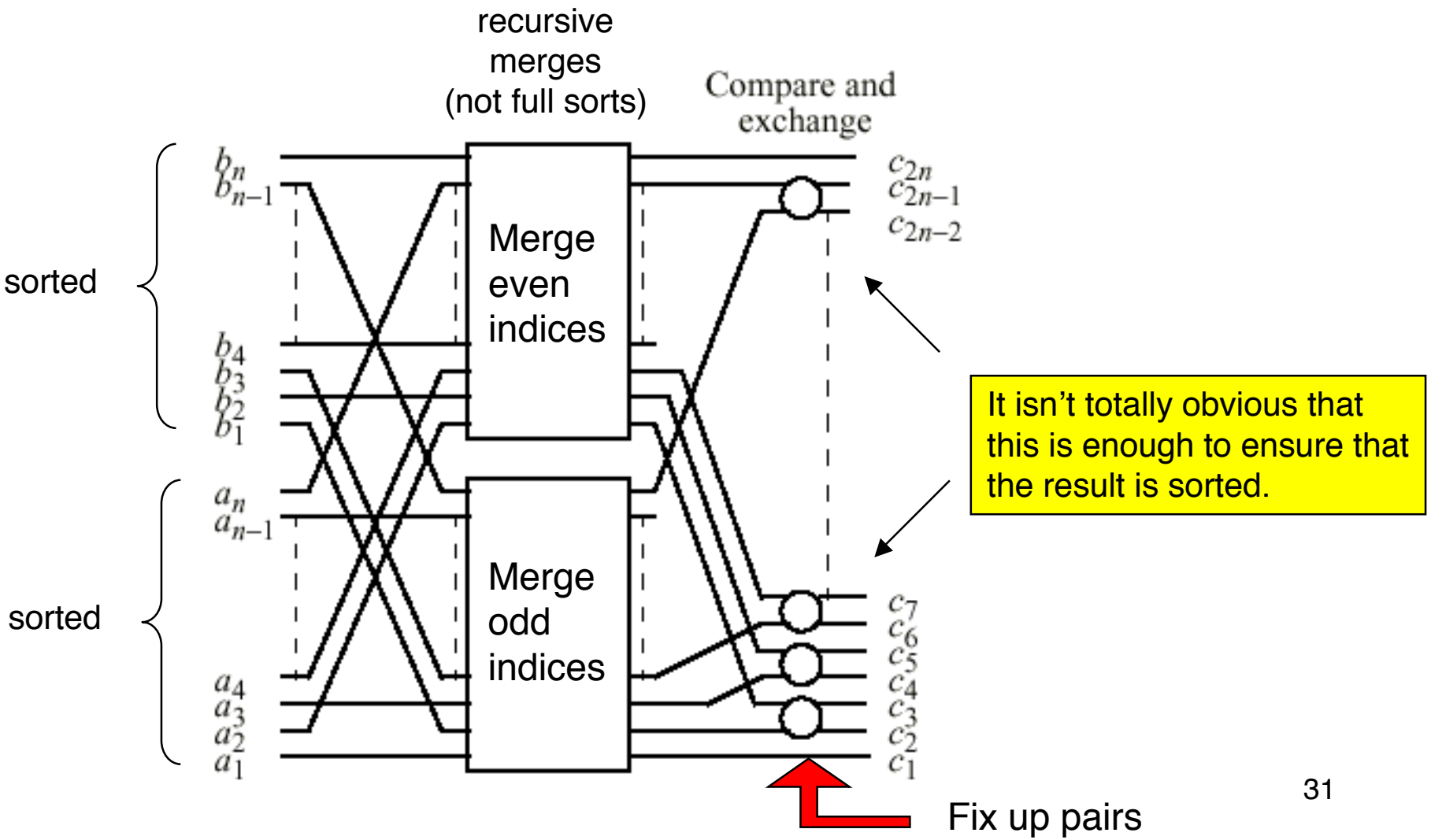
Superior Constructions

- Odd-Even Merging
  - Assume 2n elements to be sorted.
  - Split the elements into two groups.
  - Sort each half recursively.
  - Merge the odd-indexed components of the result.
  - Merge the even-indexed components of the result.
  - Merge the output of the merges.

parallel (Merge the odd-indexed components of the result. / Merge the even-indexed components of the result.)

can be done in 1 stage (Merge the output of the merges.)

Odd-Even Merging



Ways to Verify Sorted

- 0-1 Principle
- View as symmetric functions (Levy)
- Partial-order analysis (Karp)

Timing Analysis of Sorting by Odd-Even Merging

- Let $S_\infty(n)$ = time to sort n elements
- Let $M_\infty(n)$ = time to merge n/2 with n/2 elements
- Recurrences:
  - $S_\infty(1) = 0$
  - $S_\infty(2n) = S_\infty(n) + M_\infty(2n)$
  - $M_\infty(2) = 1$
  - $M_\infty(2n) = M_\infty(n) + 1$
- assuming we don't charge time for splitting indices into 2 groups

Timing Analysis of Sorting by Odd-Even Merging

- $M_\infty(2) = 1$
- $M_\infty(2n) = M(n) + 1$

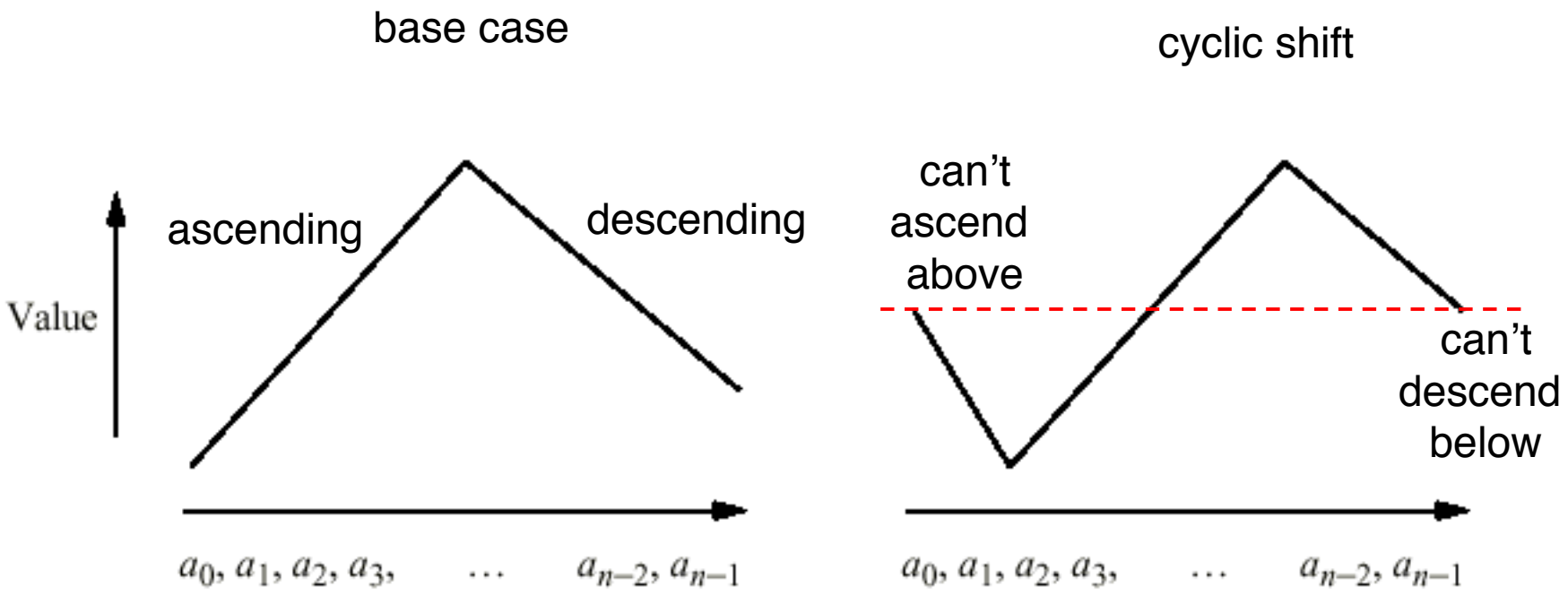
- Therefore
  - $M_\infty(2^n) = n-1$
  - $S_\infty(2^n) = S_\infty(2^{n-1}) + n-1$
- Giving $S_\infty(2^n) = (n-1) + (n-2) + \ldots + 1$

or $\boxed{S_\infty(N) = O(\log^2 N)}$

Bitonic Sorting

- Define a *bitonic* sequence to be one in which either:
  - There is a strictly ascending portion, followed by a strictly descending portion, OR
  - is a **cyclic shift** of a sequence with the above property.

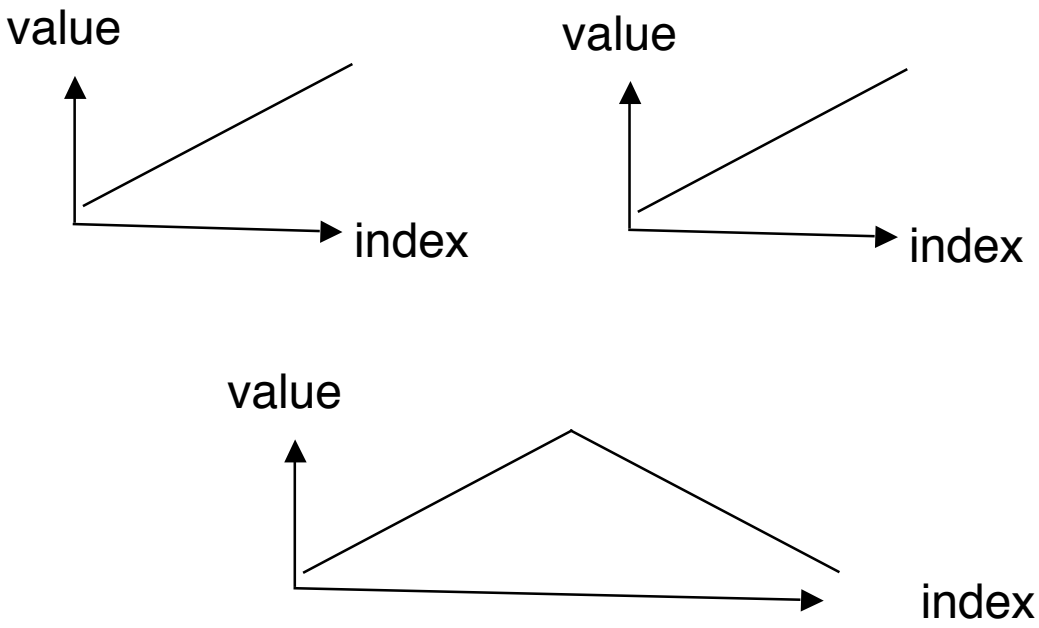
Bitonicity


base case
cyclic shift
Value
ascending
descending
can't ascend above
can't descend below
$a_0, a_1, a_2, a_3, \ldots a_{n-2}, a_{n-1}$
$a_0, a_1, a_2, a_3, \ldots a_{n-2}, a_{n-1}$

Bitonic Sorting

- Two *already-sorted* sequences can be made into a bitonic sequence by reversing the second one and abutting them, which takes no time, just index shifting.

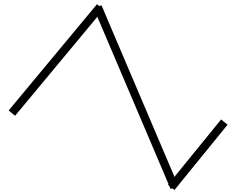
- Feeding a bitonic sequence to a “Bitonic Merge” sorts the original sequence.



Bitonic Merging

- A sequence **known to be bitonic** can be sorted by the following scheme:
  - Do compare-exchanges between elements j and j+n/2 for j = 1, 2, …, n/2.
  - This gives two sequences that (this needs to be shown):
    - are both bitonic
    - all elements of one are $\leq$ all elements of the other
  - Repeating this scheme recursively with the resulting two bitonic sequences gives a sorted sequence.

Illustration of Bitonic Merge



bitonic: 4 5 7 9 8 6 3 0 1 2

split: 4 5 7 9 8 6 3 0 1 2

compare-exchange:

4 3 0 1 2 6 5 7 9 8

(both are still bitonic)

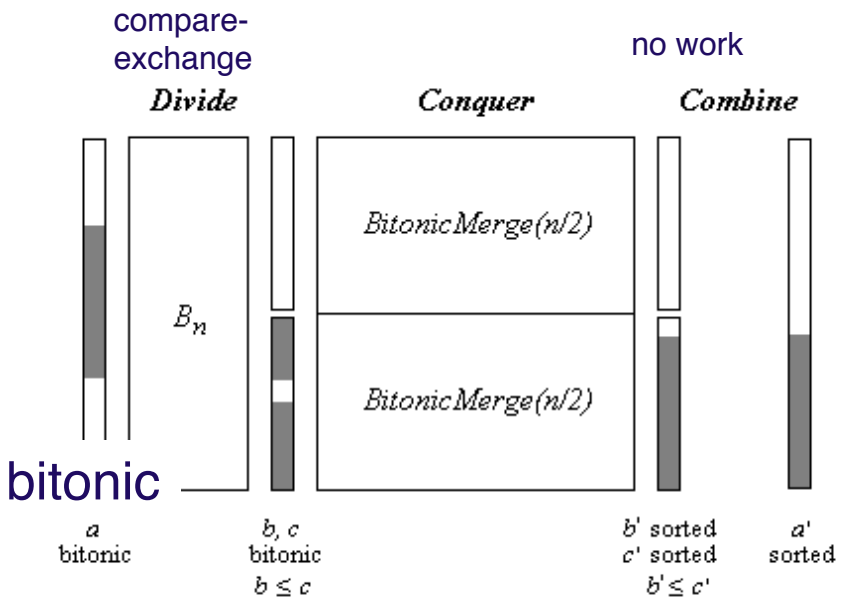
bitonic merge each (recursively):

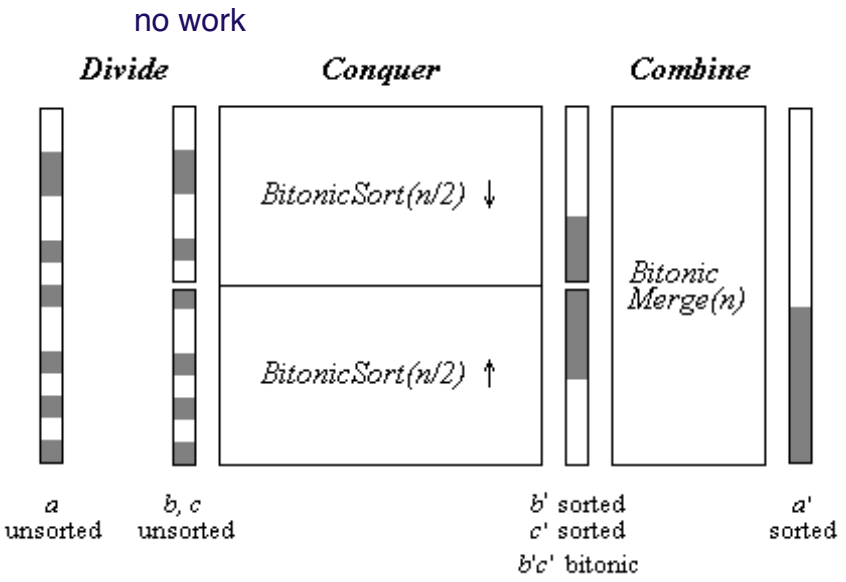
0 1 2 3 4 5 6 7 8 9

Bitonic Sort Schematic

BitonicMerge(n):
(requires bitonic input,
produces sorted output)

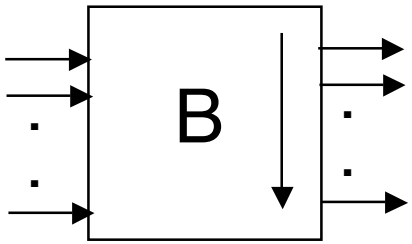
BitonicSort(n):
(arbitrary input, sorted output)





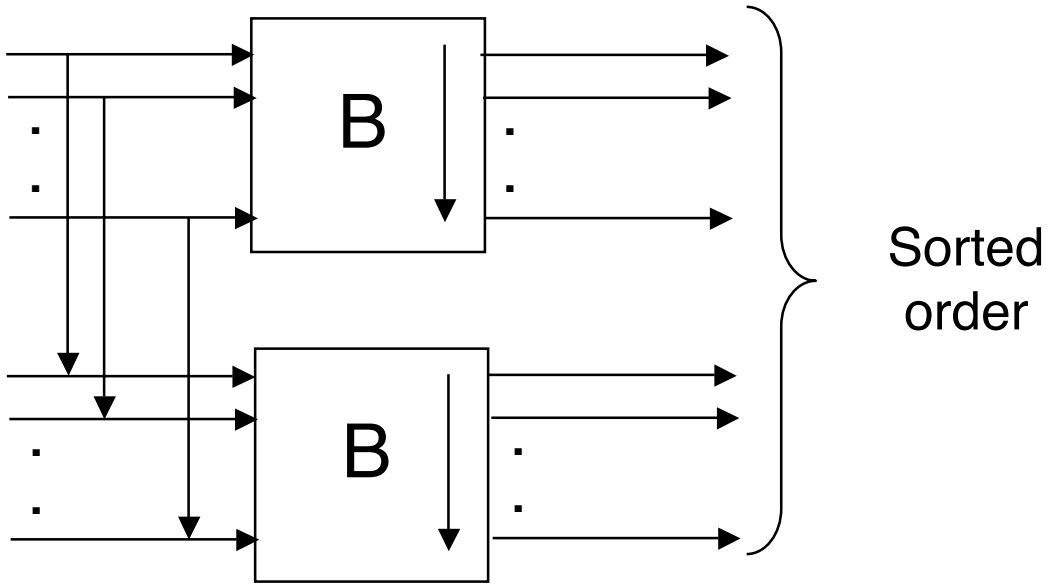
B_n is a rank of compare-exchange, comparing elements offset by n/2 in their indices.

Sorting a Bitonic Sequence

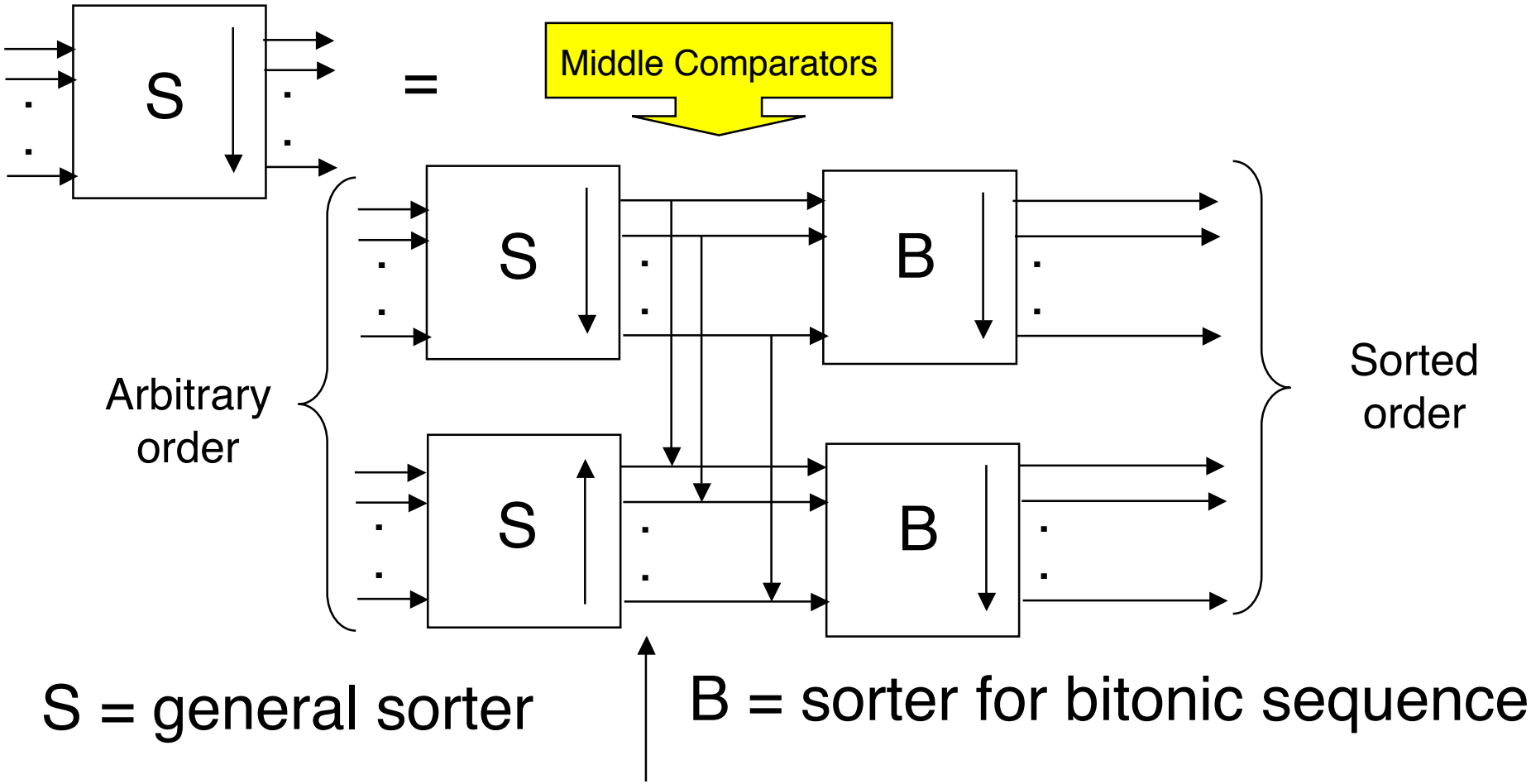


=

B = sorter for bitonic sequence



Bitonic Sort Recursion

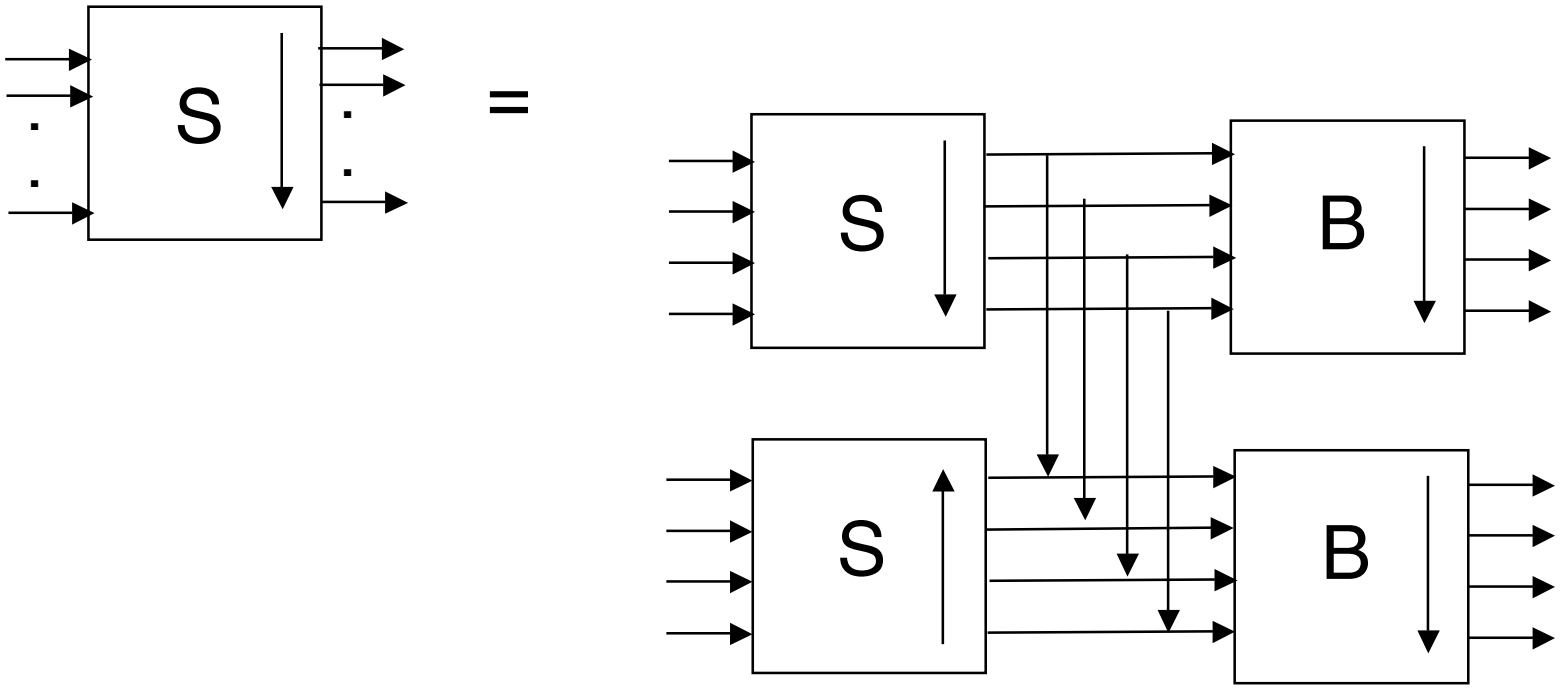


S = general sorter

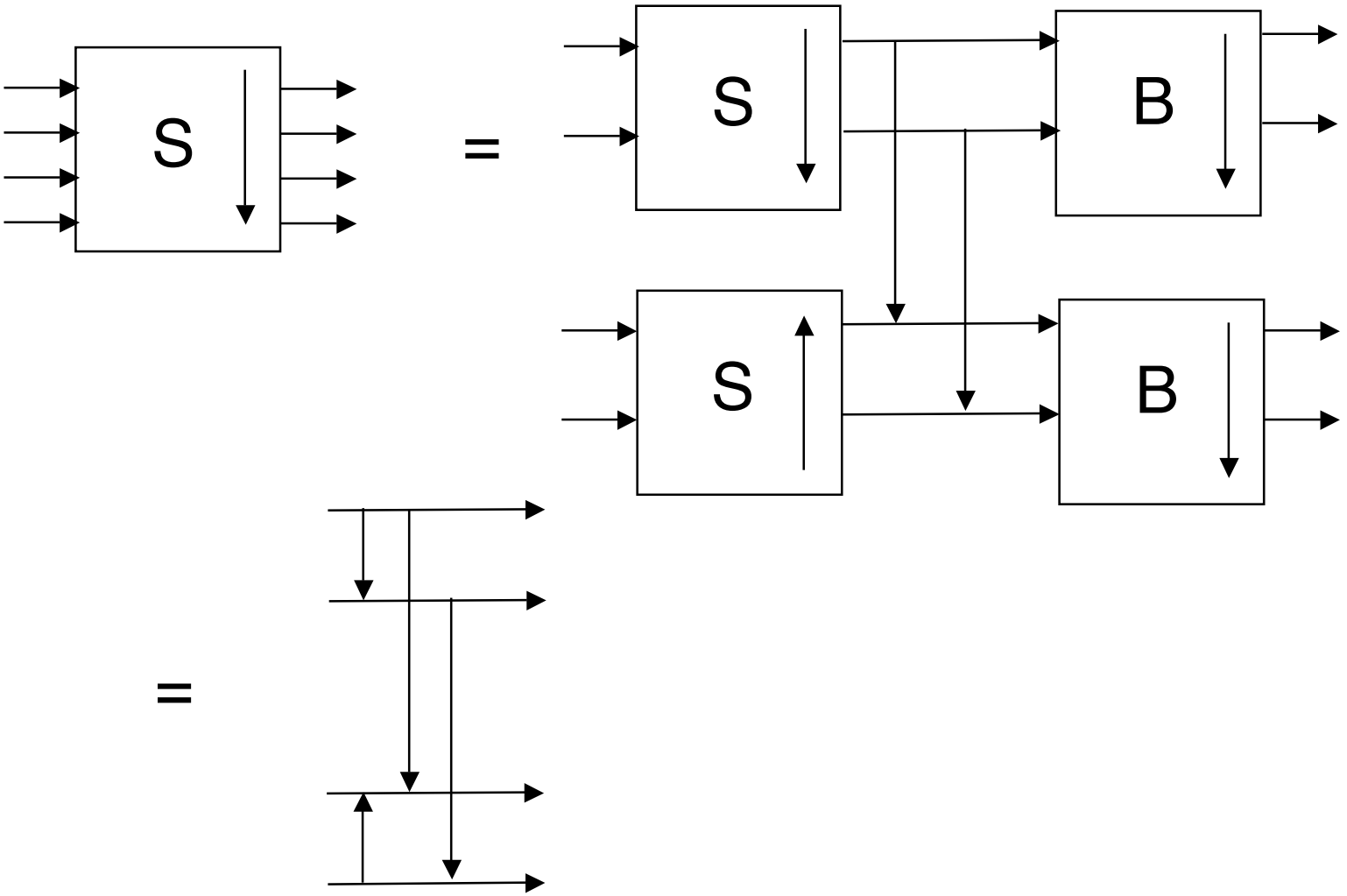
B = sorter for bitonic sequence

bitonic sequence, by abutting

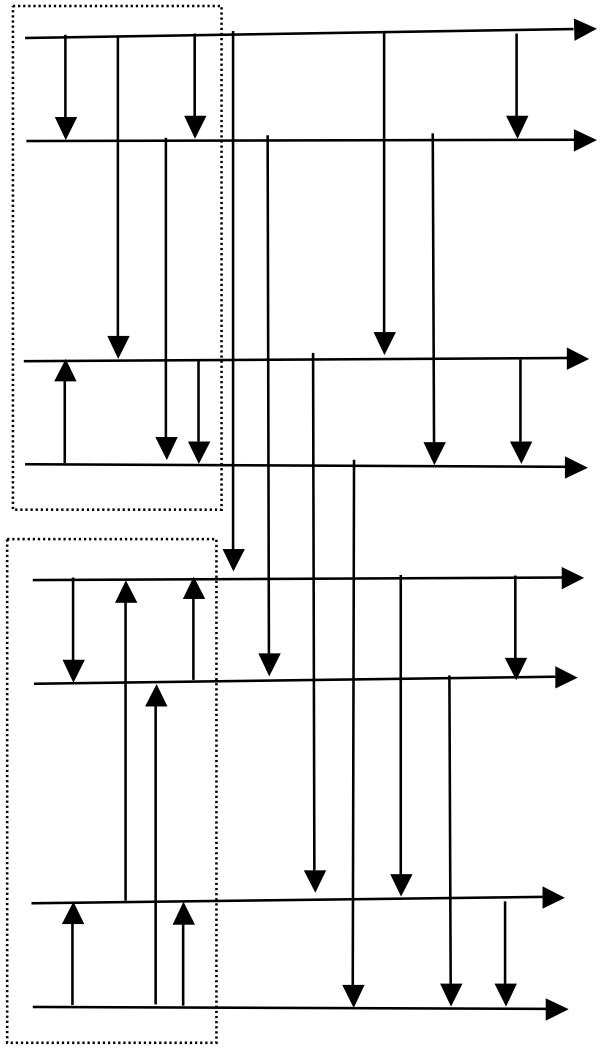
Example: 8 Elements



4-Element



8-Element



Python Implementation

```
def bitonic_sort(up,x):
    if len(x)<=1:
        return x
    else:
        first = bitonic_sort(up,x[:len(x)/2])
        second = bitonic_sort(not up,x[len(x)/2:])
        return bitonic_merge(up,first+second)

def bitonic_merge(up,x):
    # assume input x is bitonic, and sorted list is returned
    if len(x) == 1:
        return x
    else:
        bitonic_compare(up,x)
        first = bitonic_merge(up,x[:len(x)/2])
        second = bitonic_merge(up,x[len(x)/2:])
        return first + second

def bitonic_compare(up,x):
    dist = len(x)/2
    for i in range(dist):
        if up and x[i] > x[i+dist] or not up and x[i] < x[i+dist]:
            x[i], x[i+dist] = x[i+dist], x[i] #swap
```

Exercise

- Comment on the pipelinability of bitonic sorting.
- Analyze the time taken for bitonic sorting.

Other facets of sorting networks

- The **number** of comparators for **odd-even merging** is $O(n \log^2 n)$, which is typical of the most accessible constructions.

- An upper bound of $O(n \log n)$ comparators and $O(\log n)$ time has been shown, but the constants are large (in the 1000's) [AKS network after its discoverers Ajtai, Komlós, and Szemerédi].

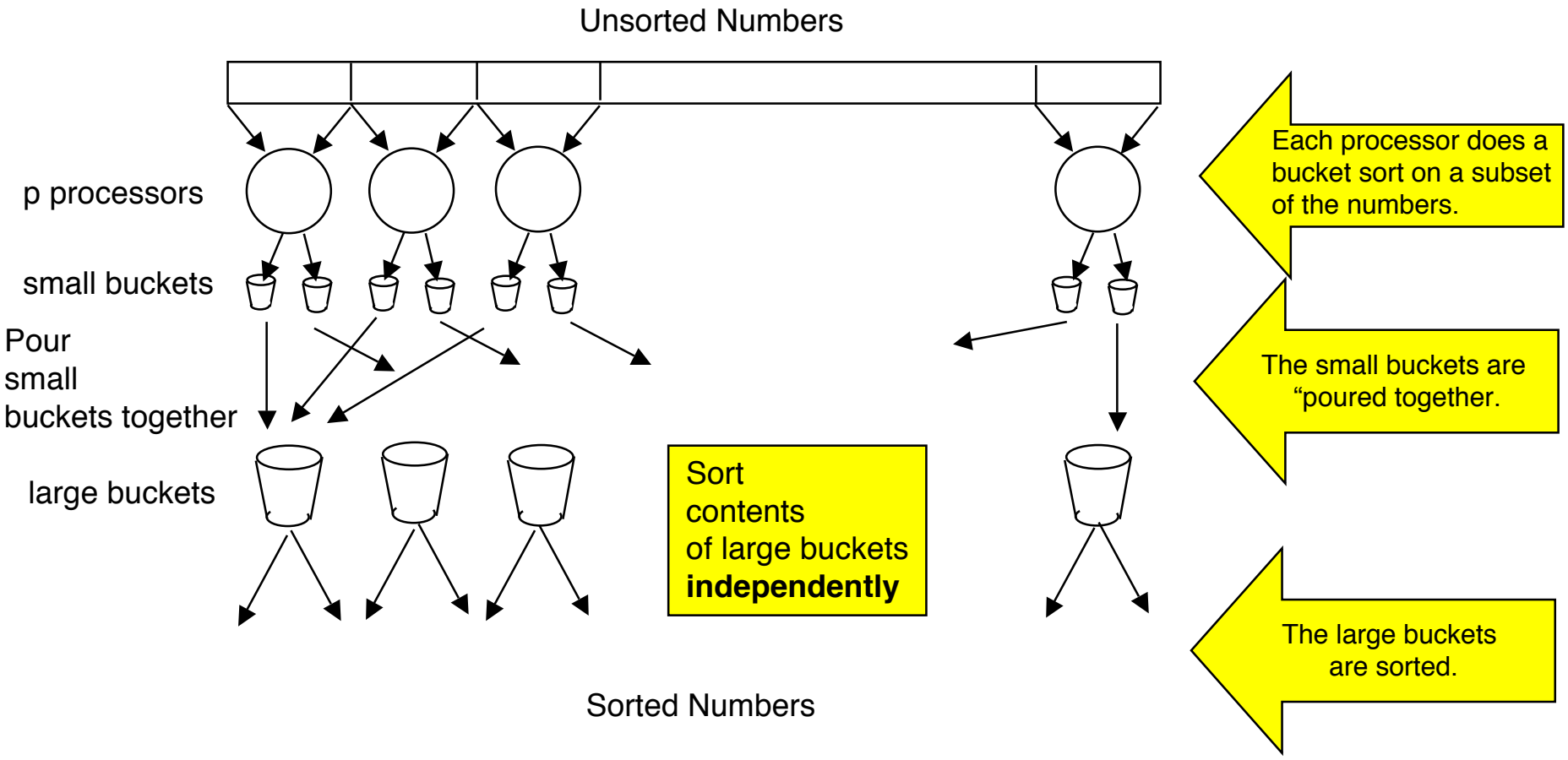
- See http://en.wikipedia.org/wiki/Sorting_network

Parallel Sorting on a Cluster?

A Parallel Bucket Sort

(Wilkinson and Allen, Parallel Programming, sec. 4.2.1, Prentice-Hall, 2005)

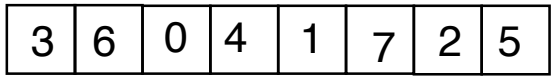
A *bucket* receives numbers in a pre-specified range.



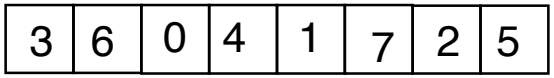
Application of In-Order Traversal

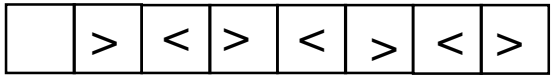
- A PRAM version of Quicksort
- Construct a tree indicating how the nodes partition (without actually moving any data)
- An in-order traversal of the tree gives the nodes in sorted order.

Quicksort Partition Tree



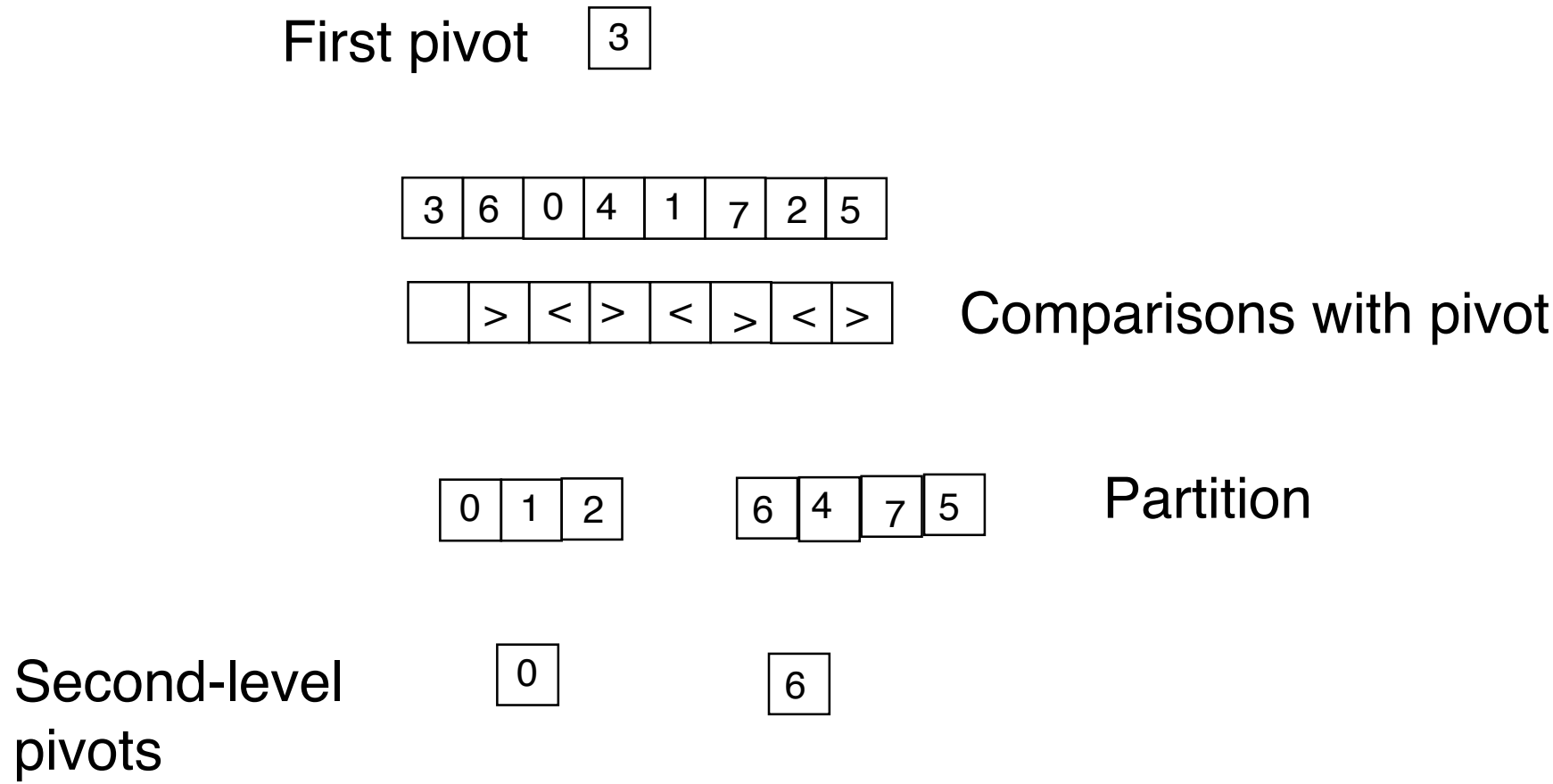
First pivot 



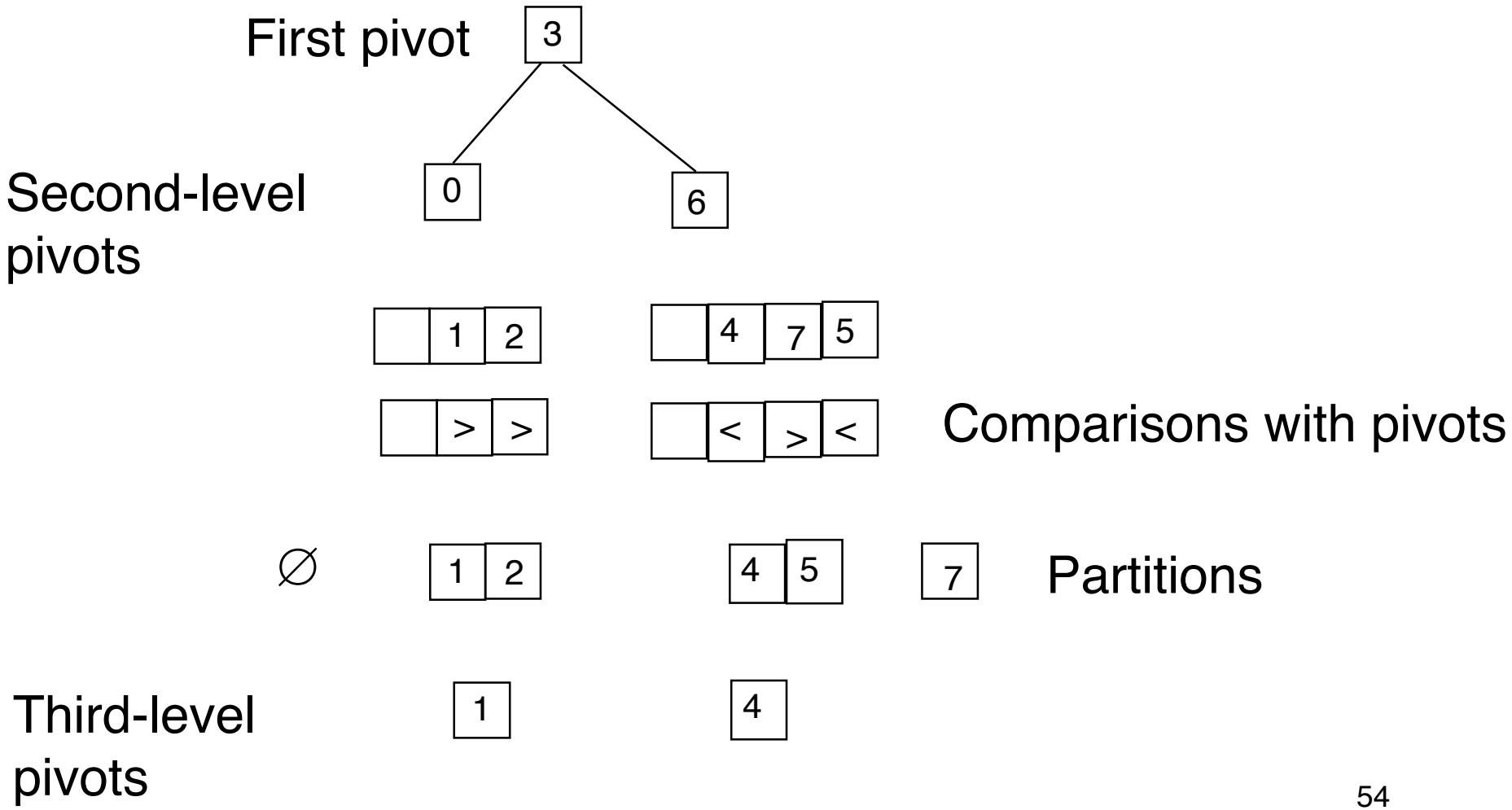


Comparisons with pivot
(Each node remembers which half it is in.)

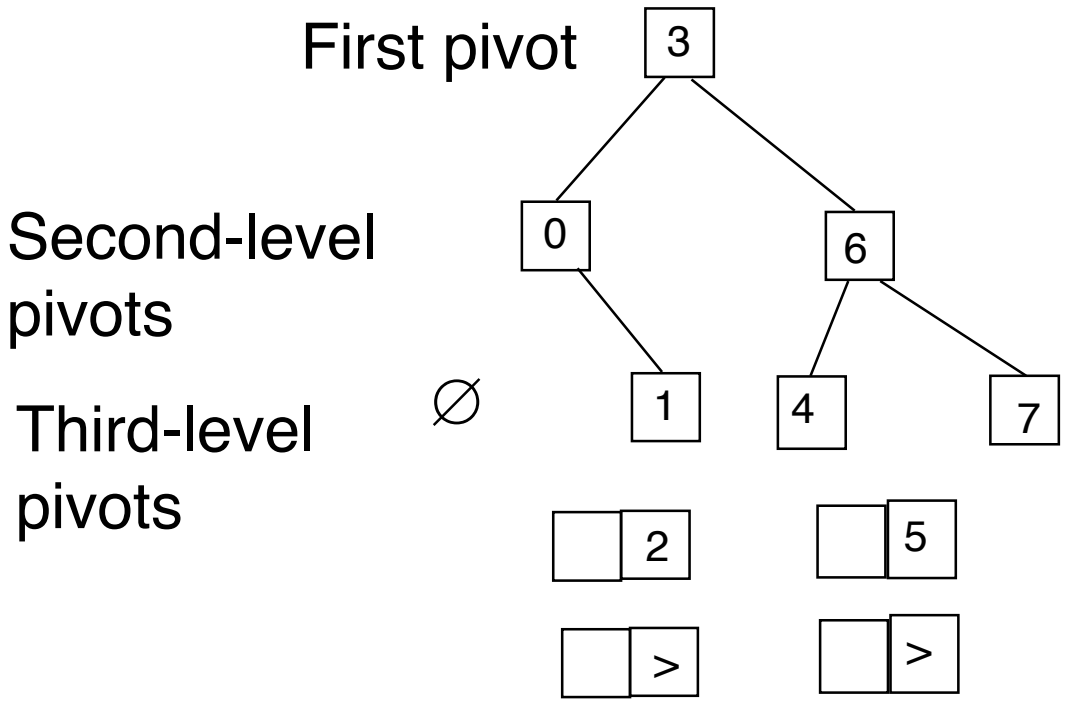
Quicksort Partition Tree



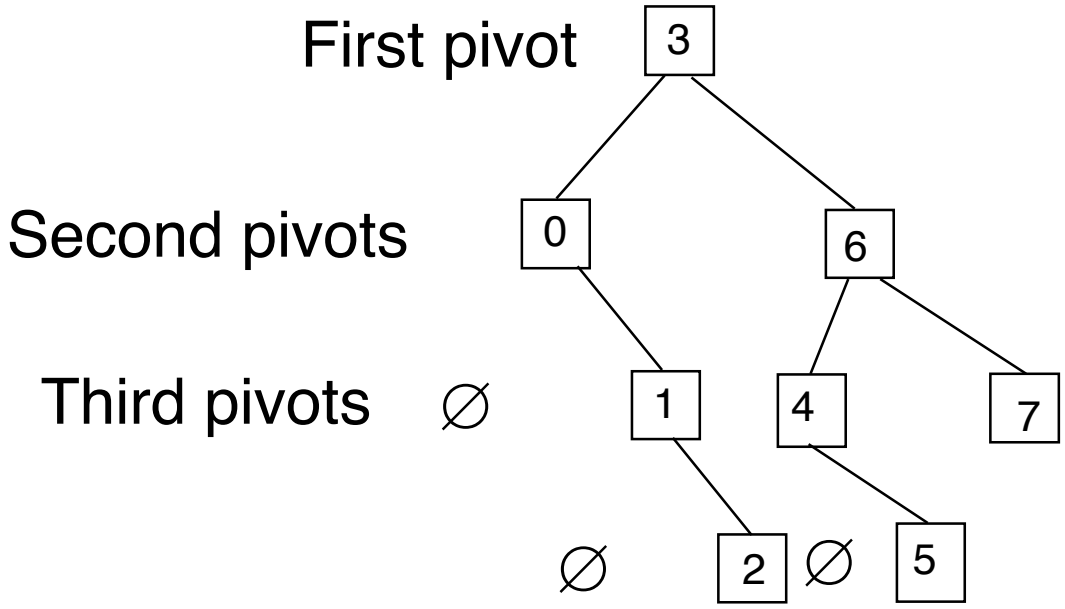
Quicksort Partition Tree



Quicksort Partition Tree



Quicksort Partition Tree



In-order traversal is sorted array

Extreme Sorting

- On an additively-combining PRAM, n numbers can be sorted in O(1) time using n^2 processors.

- Catch:
  - Lots of processors, not cost optimal
  - Additive combining sweeps a lot under the rug.

“Parallel sorting in O(1)”

(from http://www.cs.uku.fi/~penttone/parallel/sort.html, has demo)

```
procedure Sort(modifies A: array 1..n of integer)
for i in 1..n pardo K[i]:=0

for i in 1..n pardo
  for j in 1..n pardo
    if A[i]<=A[j] then K[j]:=K[j]+1

for i in 1..n pardo A[K[i]]:=A[i]
```

How many processors?
What kind of conflict resolution?
How much effort?

Not exactly sorting, but …

N-Body Computation

- Simulate movement of N bodies with point mass m_1, m_2, …
- "Point mass" means the bodies are approximated as single points with the given mass.
- Each *pair* of bodies are attracted by gravitational force of magnitude:

$$F = G\, m_i\, m_j / d^2$$

- where d is distance between the points.

Variations

- Charged particles, coulomb force
- Molecular dynamics,
  particles are atoms
- Fluid dynamics,
  particles are droplets

N-body: Euler's Method

- Knowing the acceleration, we can compute the change in vector velocity for a small time-step.

- Knowing the vector velocity, we can compute the new position for a small time-step.

N-Body Computation

- Force is mass x acceleration.
- Acceleration is force/mass.
- Acceleration is also change in velocity.
- So the **net acceleration** of the points can be obtained by:
  - Computing for each point i, the vector *sum* over all *other* points j of

$$G\, m_j * (x_j - x_i)/d^3$$

to account for x_j-x_i in numerator

N-body: Euler's Method

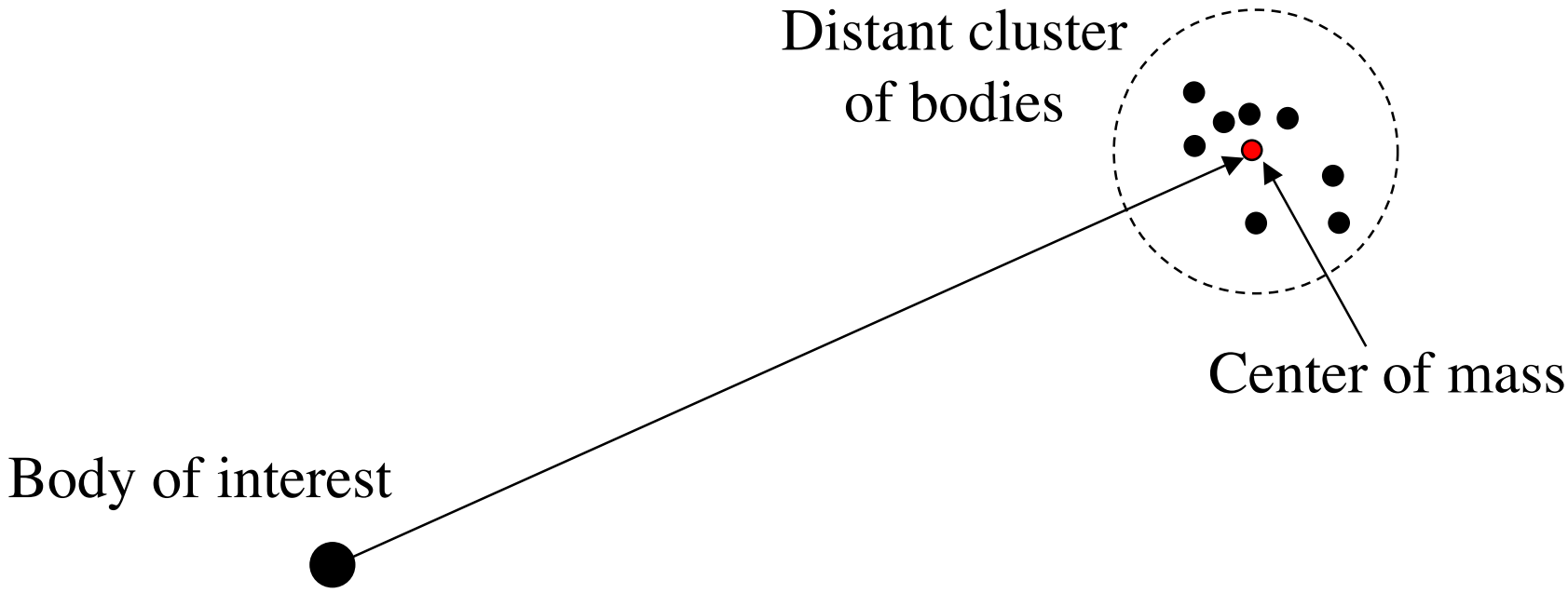
- The computation of the acceleration for a single point is O(N).
- The computation for N points is $O(N^2)$.
- This computation must be repeated each time step.
- For large N, this can be significant.

N-body: Euler's Method

- Each point's accelerations can be computed independently of the others.
- Is this not a pleasantly-parallel problem?
- Actually is an embarassingly *less*-parallel problem, because a significant ***algorithmic*** speedup is possible.

Barnes-Hut Algorithm for the N-body problem

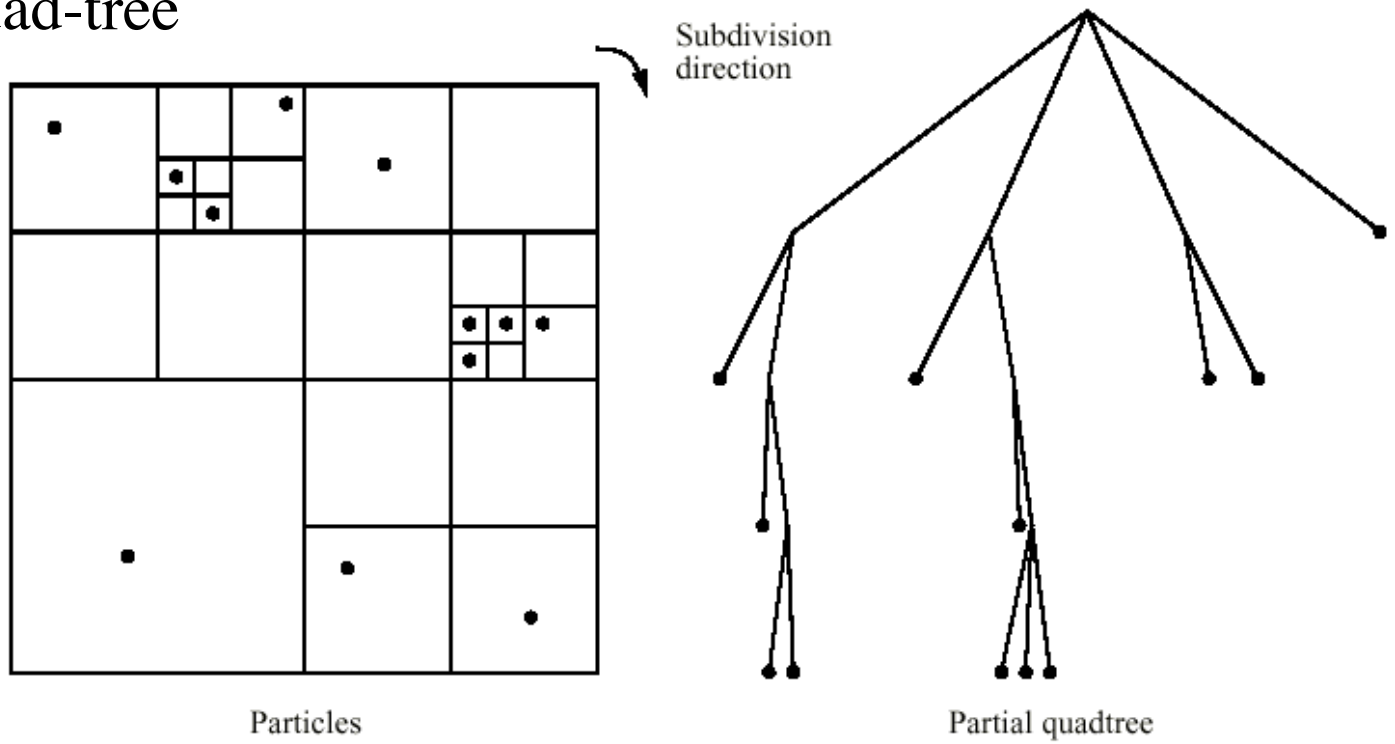
If a cluster of bodies is a long distance away from a given point, the effect of the entire cluster on the point is **well-approximated by a composite mass** located at the center of mass of the cluster.



Barnes-Hut Algorithm
(*Nature*, **324**, December 1986)

Adaptively divide space up into an oct-tree (3D) or quad-tree (2D).
Stop when a cell contains 0 or 1 point masses.

quad-tree



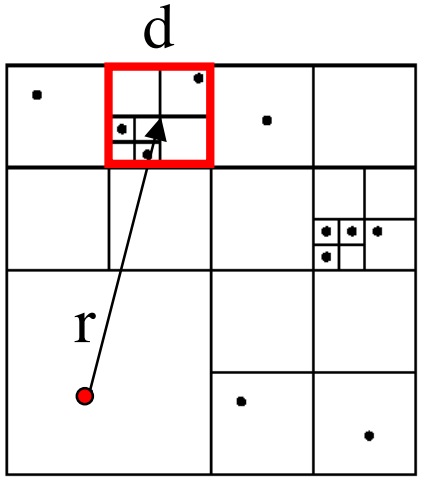
Assumption

- In the following, we make the **assumption** that the points are **distributed evenly enough** that **the height of the quad- or oct-tree is bounded by log N**.

- There might be some fixed precision, b, of bits that are used to index the subtrees, in which case log N could be replaced with b.

Barnes-Hut Algorithm

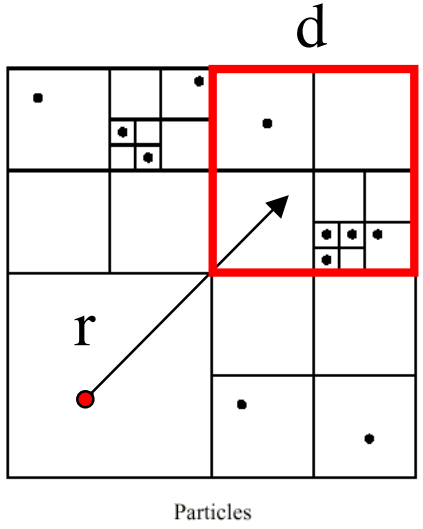
Each cell is approximated, for *distant* points, by the contained mass located at its center of gravity.
A criterion is needed for determining what is *distant*.

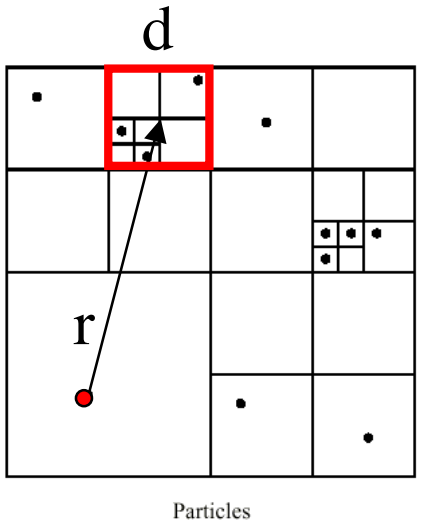


Let **d** be the dimension of one side of a box.
Let **r** be the **distance** of this point to the centroid of the box.
The **smaller d/r** is, the less error in this approximation.

Barnes-Hut Algorithm

The *smaller* d/r is, the *less error* in this approximation.
For an arbitrary point and box, d/r may be **too large**.





We generally must sub-divide the boxes (walk down the tree)
until d/r becomes acceptable, ***then*** use the approximation on that box.
Computing the net acceleration is thus **recursive**.

Barnes-Hut Algorithm

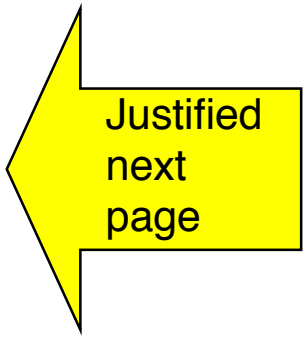
- The acceptability criterion is called the MAC:
  Multipole Acceptability Criterion.
- Recommended value is
  $d/r < 1/\sqrt{3} = 0.57735$.
- Many other criteria exist.

Barnes-Hut Algorithm

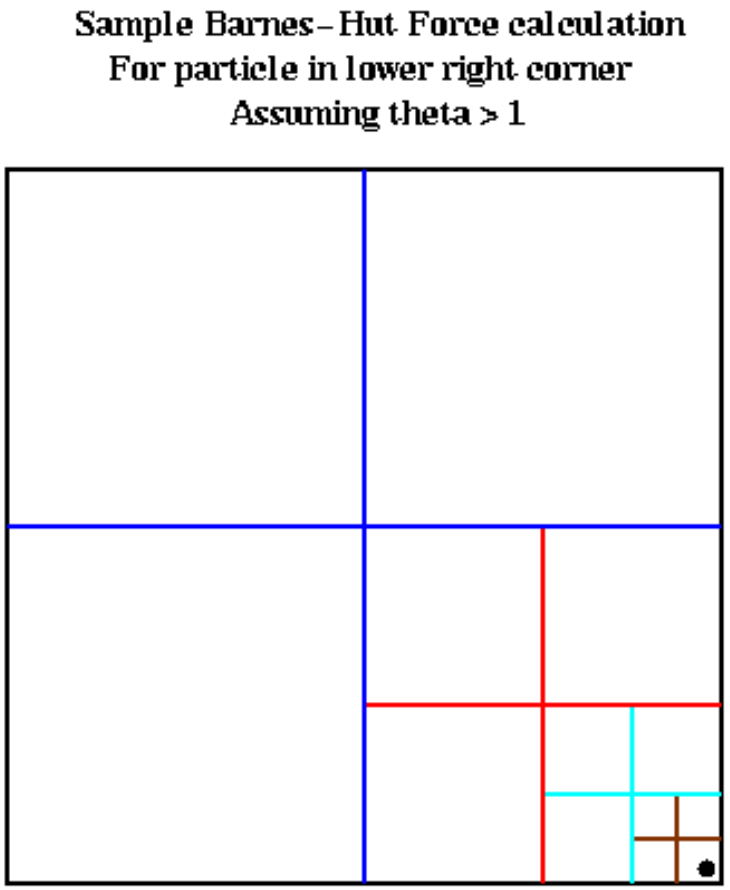
- On the preceding slides we indicated how the accelerations can be computed for one state.
- The particles will be in *different positions* in the next time step.
- Therefore the **tree** might need to be **reconstructed** on ***every*** time step.

Complexity of One Time-Step of Barnes-Hut Algorithm

- The height of the tree was assumed to be O(log N).
- The number of steps to compute the tree, including the centroids of each sub-box, will be O(N log N).
- The cost of computing the acceleration on ***one*** particle is O(log N).
- Therefore the total cost of one time step is O(N log N).



Justification for Force-Computation Bound



Number of cells examined =
3*__depth__ of point
$\in O(\log N)$

Pseudo-code for tree construction

(source: http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html)

```
procedure QuadtreeBuild
   Quadtree = {empty}
    For i = 1 to n                ... loop over all particles
      QuadInsert(i, root)         ... insert particle i in quadtree
    end for
    ... at this point, the quadtree may have some empty
    ... leaves, whose siblings are not empty
    Traverse the tree (via, say, breadth first search),
      eliminating empty leaves
```

```
procedure QuadInsert(i,n)
   ... Try to insert particle i at node n in quadtree
   ... By construction, each leaf will contain either
   ... 1 or 0 particles
   if the subtree rooted at n contains more than 1 particle,
     determine which child c of node n particle i lies in
       QuadInsert(i,c)
   else if the subtree rooted at n contains one particle
     ... n is a leaf
     add n's four children to the Quadtree
     move the particle already in n into the child
       in which it lies
     let c be child in which particle i lies
     QuadInsert(i,c)
   else if the subtree rooted at n is empty
     ... n is a leaf
     store particle i in node n
   endif
```

Pseudo-code for force computation

(source: http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html)

```
... For each particle, traverse the tree
    ... to compute the force on it.
    For i = 1 to n
       f(i) = TreeForce(i,root)
    end for

    function f = TreeForce(i,n)
       ... Compute gravitational force on particle i
       ... due to all particles in the box at n
       f = 0
       if n contains one particle
          f = force computed using the formula
       else
          r = distance from particle i to
                center of mass of particles in n
          D = size of box n
          if D/r < theta
             compute f using the formula
          else
             for all children c of n
                f = f + TreeForce(i,c)
             end for
          end if
       end if
```

Parallel Computation of Barnes-Hut Algorithm

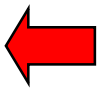
- Can Barnes-Hut Exploit Parallelism?
  - Steps are:
    - O(N) finding outer bounds of set of particles
    - O(N log N) tree-construction computation
    - O(N log N) point acceleration computation
  - Which steps can be parallelized?

Parallel Computation of Barnes-Hut Algorithm

- Conjectured parallel versions for large N, p processors, **ignoring communication costs**
  - Each step is:
    - O(N/p) finding outer bounds of set of particles
    - O(N log N/p) tree-construction computation
    - O(N/p) center of mass computation (for tree)
    - O((N log N)/p) point acceleration computation
  - O((N logN)/p) overall

Parallel Computation of Barnes-Hut Algorithm

- What if we can't ignore communication costs (e.g. on distributed-memory system)?
  - One problem is **load-balancing**:
    - How do we distribute the computational work onto p processors so that each one is approximately equally busy?
      - finding outer bounds of set of particles
      - tree-construction computation
      - center of mass computation (for tree)
      - point acceleration computation



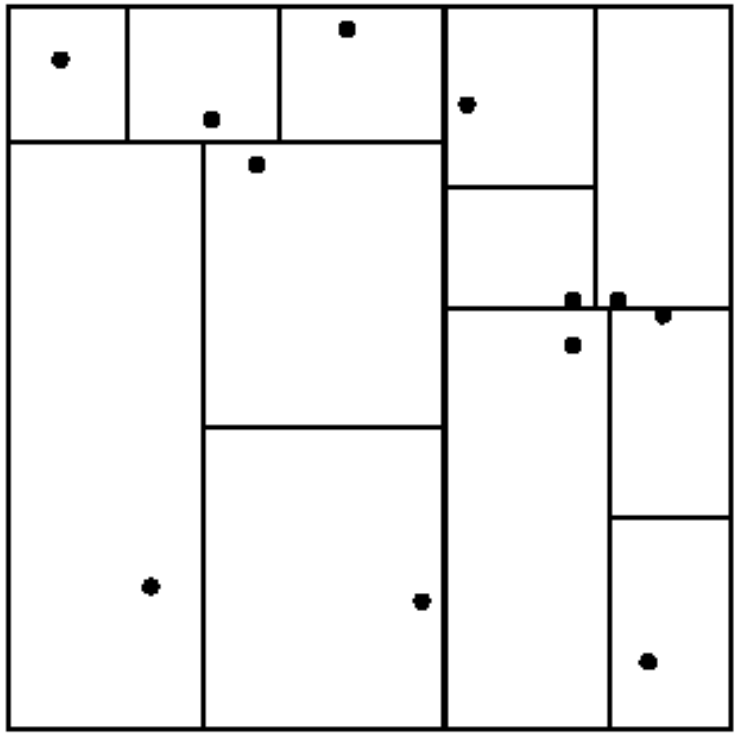
dominant

Orthogonal Recursive Bisection for load-balancing

Position vertical line so as to divide number of points in half.
Within each half, position horizontal line so as to divide those numbers of points in half, etc., until there are as many divisions overall as there are processors.

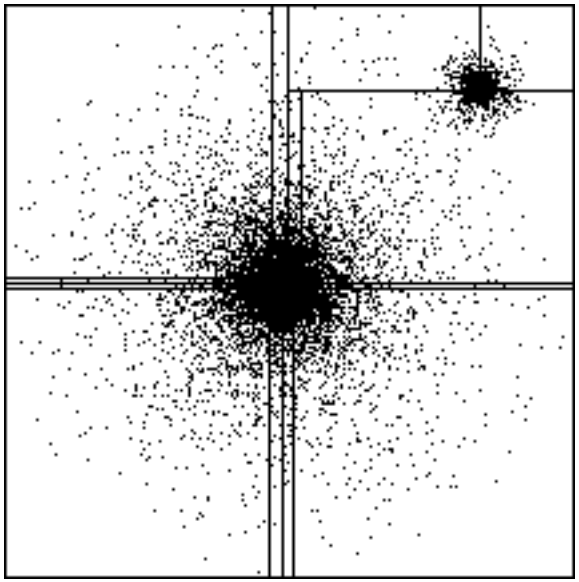
This is ***separate*** from the partitioning used in the quad-tree.

Load-balancing does not demand that spatially-close points be on the same processor.



Orthogonal Recursive Bisection

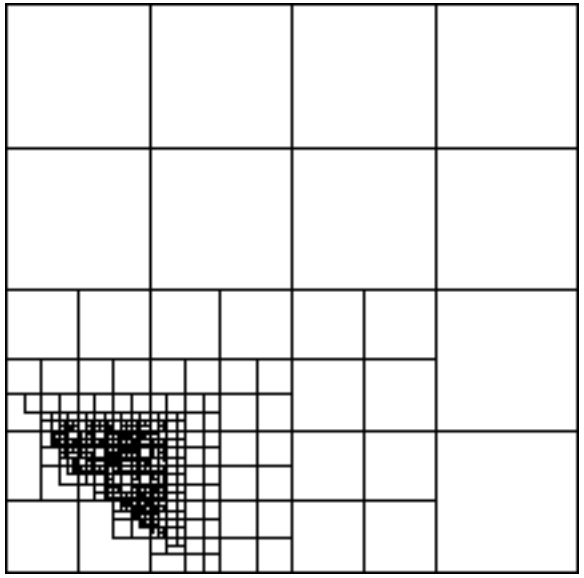
Example: Decomposition Resulting from Orthogonal Recursive Bisection of a System with Two Galaxies



From: http://www.npac.syr.edu/copywrite/pcw/node281.html

A Related Issue: Locality of Data can be Used to Advantage

Optimizing the acquisition of **locally-essential** data.



Each square represents a datum required to compute force on a point in the lower-left quadrant.

See: http://www.npac.syr.edu/copywrite/pcw/node282.html

N-Body Case Study, 1992

(from http://www.npac.syr.edu/copywrite/pcw/node283.html)

- 512-processor Intel Delta at Caltech
- 17.15 million bodies for approximately 600 time steps
- simulated regions of the universe 100 megaparsec
- ran at an aggregate speed exceeding 5000 MFLOPS/sec, generating 25 GB of data
- initialized with random-density fluctuations consistent with the “cold dark matter” hypothesis
- 1992 Gordon Bell Price for performance in practical parallel processing research

Fast Multipole Method (FMM)

(L.. Greengard and V. Rokhlin, J. Comp. Phys. 73 (1987) 325.)

- An alternate method for the N-body problem.
- It is based on establishing a **vector field** in which the force on a particle can be evaluated.
- (A multipole expansion is akin to a Taylor's series.)
- Like Barnes-Hut, it uses a quad or oct-tree.
- It is O(N) per step, but the constant may be higher than in the O(N log N) Barnes-Hut algorithm.
- O(N/p) parallelization is possible.

FMM Cursory Outline

(http://www.eecs.berkeley.edu/~demmel/cs267/lecture27/lecture27.html)

(1) Build the quadtree containing all the points.

(2) Traverse the quadtree from bottom to top, computing **Outer(n)** [potential away from n] for each square n in the tree.

(3) Traverse the quadtree from top to bottom, computing **Inner(n)** [potential inside n] for each square in the tree.

(4) For each leaf, add the contributions of nearest neighbors and particles in the leaf to **Inner(n)**.

References

- http://www.cs.cmu.edu/~scandal/alg/nbody.html (includes an applet for Barnes-Hut)
- http://www.npac.syr.edu/copywrite/pcw/node303.html
- http://www2.epcc.ed.ac.uk/~mario/nbody.html (*many* additional links)
- http://www.eecs.berkeley.edu/~demmel/cs267/
- http://www.umiacs.umd.edu/~wpwy/fmm/ (FMM applets)