1st Steps Toward Parallel Programming

Alexandre David
1.2.05
adavid@cs.aau.dk

Data and Task Parallelism

- Do we parallelize the data or the code?
  - Data parallelism: same operation to different data items at the same time. Parallelism grows with data.
    E.g. on GPUs.
  - Task parallelism: do different tasks at the same time. Number of tasks *may* be fixed and not scalable.
    E.g. pipelines.

Peril-L Notation

- Pseudo-code for parallelism.
  - simplified notation for describing algorithms
  - easy to go from pseudo-code to a programming language
  - conceptually complete and unambiguous (for us)
  - possible to reason about performance
  - here for parallelism
  - execute on a CTA – locality awareness
  - C look & feel
  - Important: Not Peril-L notation itself but the concepts that go with it.

Parallel Threads

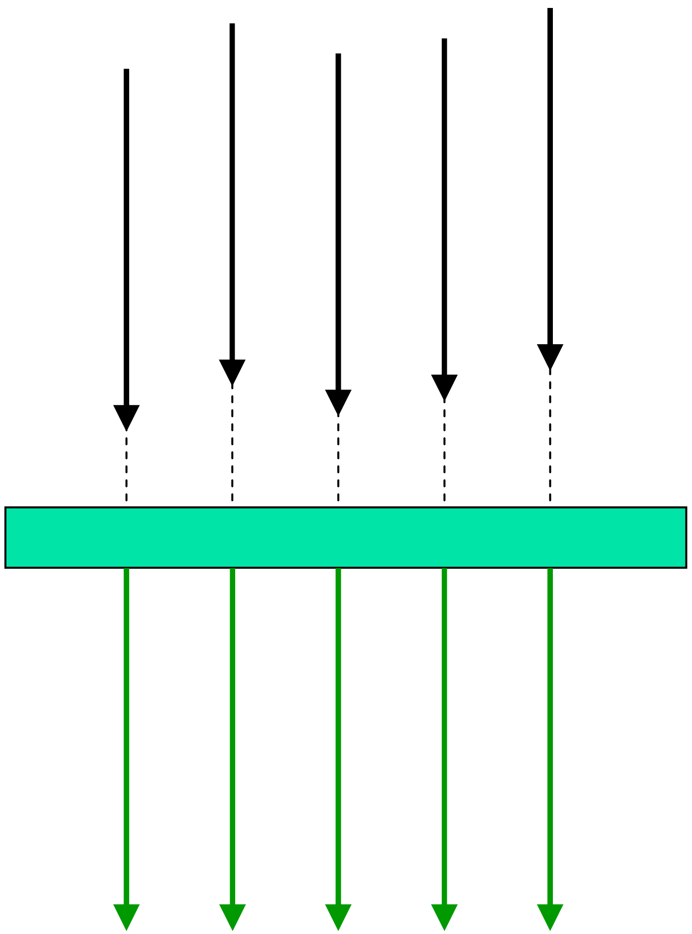
```
forall(<integer variable> in (<index range>))
{
        <body>
}
```

- Conceptually
  - Consider an unordered set of indices (range specified).
  - Execute the code over that set.
- Parallelism:
  - The index range = a set S of indices.
  - A *logical* thread per index of S executes the code with that index value.
  - No order is enforced.
  - Synchronization is not specified.

Synchronization

- If you do not enforce order between threads there will be no order.
  - Corollary: Threads are evil, if they can behave in a bad way, they will.
- Mutual exclusion: *exclusive* { <body> }
- Barrier synchronization: barrier

Barrier



```
forall(index in (1..12))
{
  printf("tweedle dee\n");
  barrier;
  printf("tweedle dum\n");
}
```

Memory Model

- Local variables distinguished from global variables.
  - Locality is defined by scope.
  - Global variables are underlined.
  - Be careful to concurrent writes.

```
int data[n];
forall(index in (0..n-1))
{
  if (data[index]<0)
  {
    data[index]++;
  }
}
```

OK

Global -> Local Memory

- Recall: CTA has no global memory.
  - Structures are distributed.
  - We need a way to map to local memory.
  - localize makes the mapping.

```
int allData[n];
forall(t in (0..p-1))
{
  int size=n/p;
  int localData[size] = localize(allData[]);
...
```

Accesses of thread t are local to localData. Accesses to allData is still global.

Abstract from real architecture but keep it meaningful.

Local <-> Global

- Do not mix global and local accesses.
  - Good policy.
- Protect global accesses.
- Local accesses do not need protection.
- Owner compute rule – very important:
  - A process owns some defined data and is responsible for its associated computations.
- mySize(allData[], i) instead of n/p for everyone.
- localToGlobal(local,i,j) gives the index in allData[] of the local index i for the thread j.

Synchronized Memory

- Full-empty variables
  - useful 1-place (blocking) queue
  - int t'=0; declares and fill an FE variable
  - important: accesses incur some overhead

Reduce and Scan (=Prefix)

- Useful collective operations used as steps in algorithms.
  - Associative and commutative operations.
  - Reduce: /   scan: \
  - *least=min/<u>dataArray</u>;* local min of the global array.
  - *total=+/count;* local total is the sum of all local counts.
  - *beforeMe=+\count;* local beforeMe contains the prefix (for this thread, with + operator) over counts.
  - Implicit barrier: all threads execute these statements.

?

Reduce

- Avoid
  - exclusive { total += priv_count; }
    serial code
- Use
  - total = +/priv_count;
    abstract code → parallel & scalable

Count 3s – Try 3

```
int array[length];                                        The data is global
int t;                                                    Number of desired threads
int total=0;                                              Result of computation, grand total
int lengthPer=ceil(length/t);
forall(index in(0..t-1))
{
  int priv_count=0;                                       Local accumulation
  int i, myBase=index*lengthPer;
  for(i=myBase; i<min(myBase+lengthPer, length); i++)
  {
    if(array[i]==3)                                       There's no concurrent read since
    {                                                     Array has been partitioned
      priv_count++;
    }
  }
  exclusive { total+=priv_count; }                        Compute grand total
}
```

How to Formulate Parallelism?

- Fixed parallelism – fix the number of threads
  - not scalable, not portable → avoid
- Unlimited parallelism may be missleading.

```
int count = 0;
forall(i in (0..n-1))
{
  count = +/(array[i]3?1:0);
}
```

Elegant and smart, suggests $O(\lambda\log n)$
but $P \ll n$ in practice so $O(\lambda\log P + n/P)$.
In practice simulation of the missing processes is expensive.

Goal: Identify parallelism and structure it to minimize interations.

Scalable Parallelism

- Respect locality.
  - Find right granularity for the decomposition = find the right size of sub-problems.
- Minimize interactions.
  - Keep tasks as independent as possible.

- May be contradictory w.r.t. concurrency.

Example Revisited

```
int array[length];
int t;
int total;
forall(j in(0..t-1))
{
  int size=mySize(array,0);
  int myData[size]=localize(array[]);
  int i, priv_count=0;
  for(i=0; i<size; i++)
  {
    if(myData[i]==3)
    {
      priv_count++;
    }
  }
  total =+/priv_count;
}
```

The data is global
Number of desired threads
Result of computation, grand total

Figure size of local part of global data

Associate my part of global data with local variable
Local accumulation

ompute grand total

Sorting

Problem

Arrange an unordered collection of elements into monotonically increasing (or decreasing) order.

Let $S = \langle a_1, a_2, \ldots, a_n \rangle$.

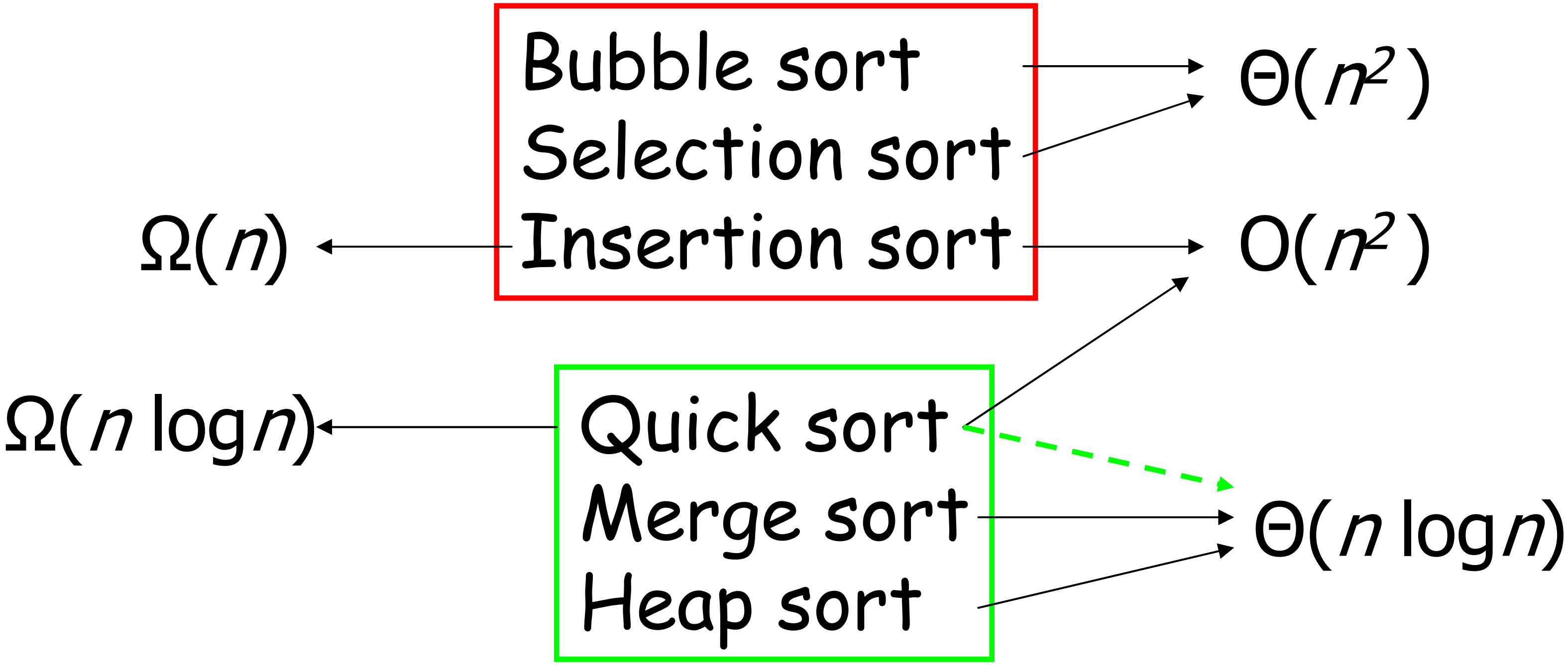
Sort S into $S' = \langle a_1', a_2', \ldots, a_n' \rangle$ such that

$a_i' \leq a_j'$ for $1 \leq i \leq j \leq n$

and S' is a permutation of S.

Here the elements are words.

Recall on Comparison Based Sorting Algorithms



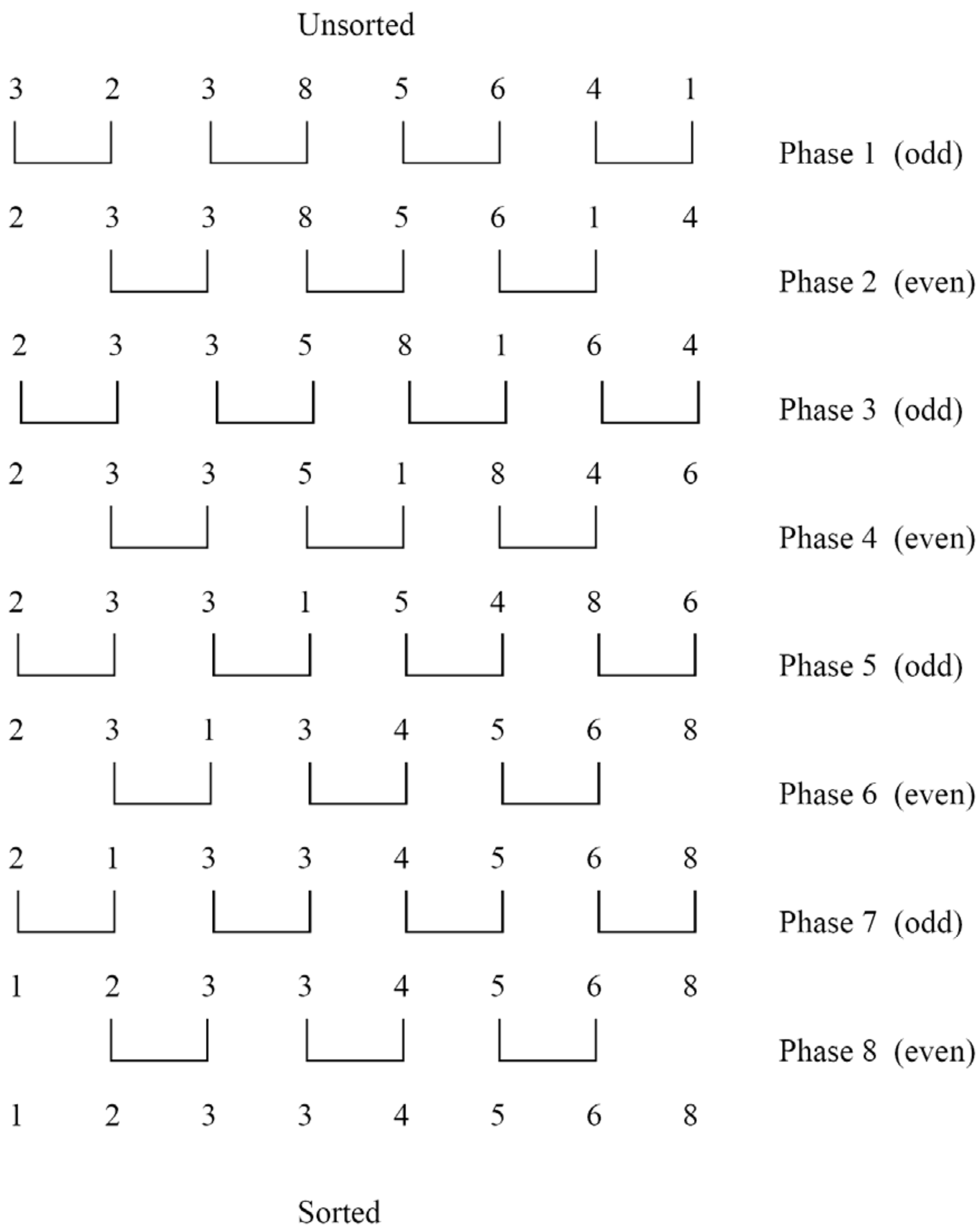
Fundamental Distinction

- **Comparison based** sorting:
  - *Compare-exchange* of pairs of elements.
  - Lower bound is $\mathbf{\Omega}(n \log n)$ (proof based on decision trees).
  - Merge & heap-sort are optimal.
- **Non-comparison based** sorting:
  - Use information on the element to sort.
  - Lower bound is $\mathbf{\Omega}(\mathbf{n})$.
  - Counting & radix-sort are optimal.

Sorting Example
Alphabetizing

- Unlimited parallelism
  - odd/even interchange
  - lots of copies
- Fixed parallelism over the letters of the alphabet
  - by batch
  - load balancing problem, not scalable
- Scalable parallelism
  - Batcher’s sort – idea from sorting networks

Odd/even interchange



Non-Peril-L Pseudo-code

```
procedure ODD-EVEN(n)
begin
    for i := 1 to n do
    begin
        if i is odd then
            for j := 0 to n/2 - 1 do
                compare-exchange(a_{2j+1}, a_{2j+2});
        if i is even then
            for j := 1 to n/2 - 1 do
                compare-exchange(a_{2j}, a_{2j+1});
    end for
end ODD-EVEN
```

$\Theta(n^2)$

$(a_1,a_2),(a_3,a_4)\ldots$

$(a_2,a_3),(a_4,a_5)\ldots$

Algorithm 9.3 Sequential odd-even transposition sort algorithm.

```
bool continue=true;
rec L[n];                                    The data is global
while(continue) do
{
  forall(i in(1:n-2:2))                      Stride by 2
  {
    rec temp;
    if(strcmp(L[i].x,L[i+1].x)>0             Is odd/even pair misordered?
    {
      temp=L[i];                             Yes, fix
      L[i]=L[i+1];
      L[i+1]=temp;
    }
  }
  forall(i in(0:n-2:2))                      Stride by 2
  {
    rec temp;
    bool done = true;                        Set up for termination test
    if(strcmp(L[i].x,L[i+1].x)>0)  Is even/odd pair misordered?
    {
      temp=L[i];                             Yes, interchange
      L[i]=L[i+1];
      L[i+1]=temp;
      done=false;                            Not done yet
    }
    continue=!(&&/done);                     Were any changes made?
  }
}
```

Hidden Communication of Odd/Even sort

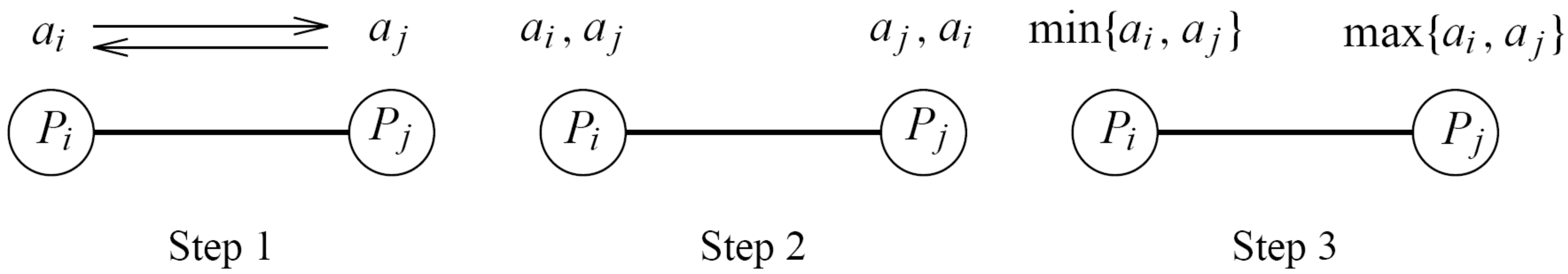


Figure 9.1 A parallel compare-exchange operation. Processes P_i and P_j send their elements to each other. Process P_i keeps $\min\{a_i, a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.

- Compare-exchange operation
  - possibly in parallel
  - communication time comparable (or greater) to the comparisons

```
rec L[n];                                     The data is global
forall(j in(0..25))                           A thread for each letter
{
  int myAllo=mySize(L, 0);                    Number of local items
  rec LocL[]=localize(L[]);                   Make data locally referenceable
  int counts[26]=0;                           Count number of each letter
  int i, j, startPt, myLet;
  for(i=0; i<myAllo; i++)                     First, count number w/each letter; need this
  {
    counts[letRank(charAt(LocL[i].x,0))]++;
  }
  counts[index]=+/counts[index];              Figure how many of each letter
  myLet=counts[index];                        Number of records of my letter
  rec Temp[myLet];                            Allocate local storage for records
  j=0;                                        Index for local array
  for(i=0; i<n; i++)                          Move records locally for local alphabetize
  {
    if(index==letRank(charAt(L[i].x,0)))
    {
      Temp[j++]= L[i];                        Save record locally
    }
  }
  alphabetizeInPlace(Temp[]);                 Alphabetize within this letter locally
  startPt=+\myLet;                            Scan counts # records ahead of these; scan
                                              synchs, so okay to overwrite L, once sorted
  j=startPt-myLet;                            Find my starting index in global array
  for(i=0; i<count; i++)                      Return records to original global memory
  {
    L[j++]=Temp[i];
  }
}
```

local batch

size of the batch

reduce

copy global to local

local sort

prefix=where to start

copy local to global

Sorting Networks

- Mostly of theoretical interest.
- Key idea: Perform many comparisons in parallel.
- Key elements:
  - Comparators: 2 inputs, 2 outputs.
  - Network architecture: Comparators arranged in columns, each performing a permutation.
  - Speed proportional to the depth.

Comparators

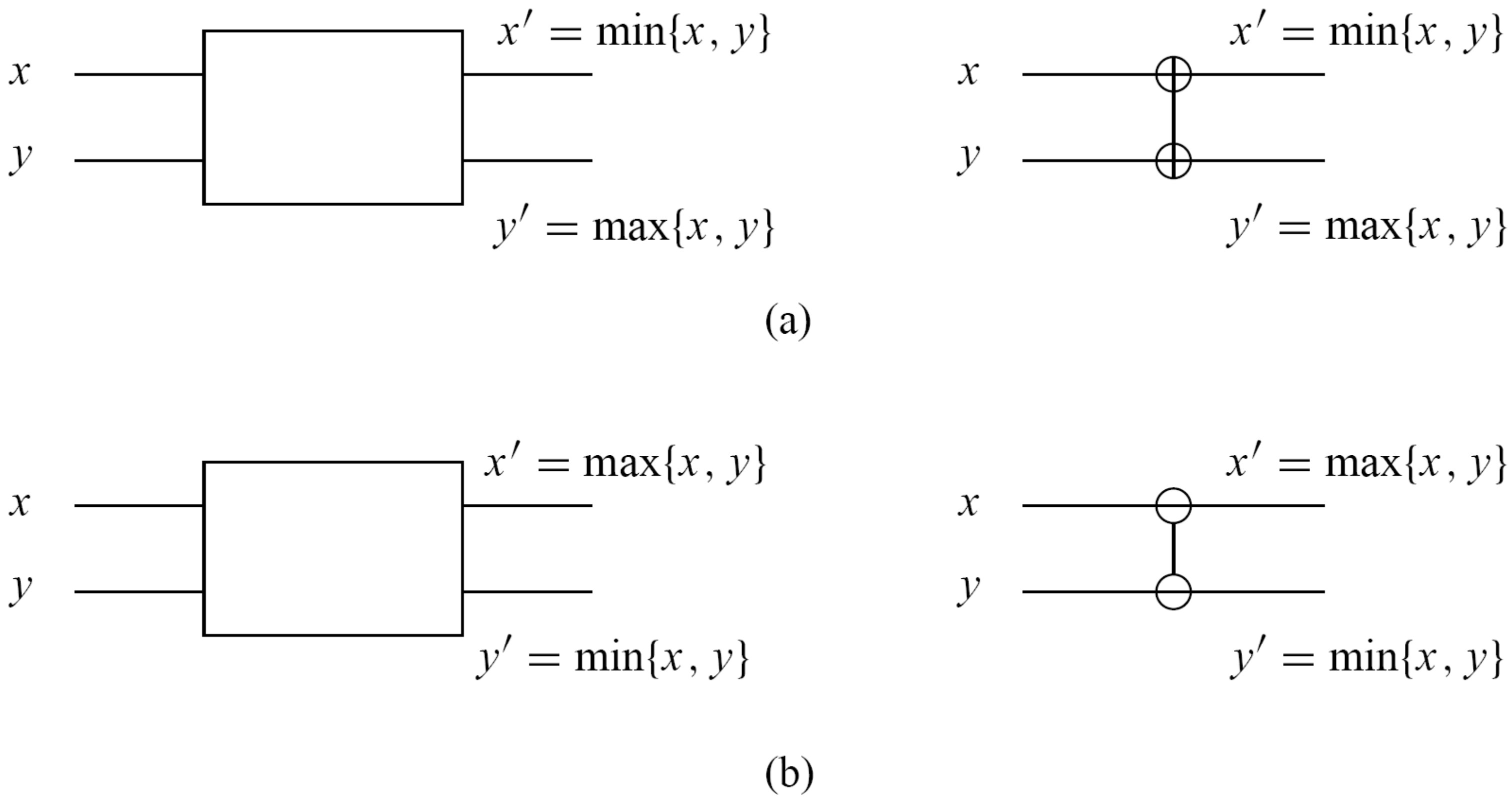


Figure 9.3 A schematic representation of comparators: (a) an increasing comparator, and (b) a decreasing comparator.

Sorting Networks

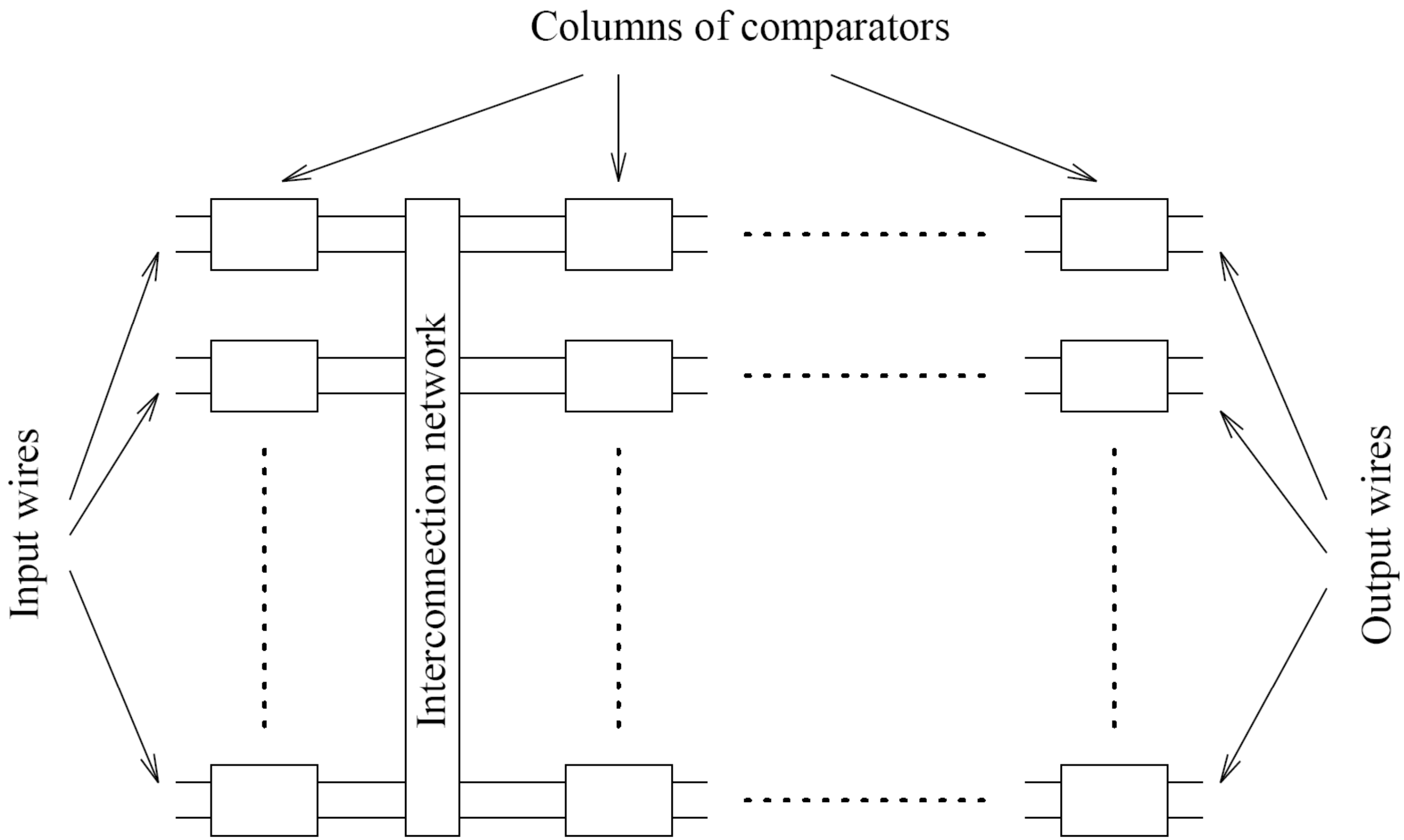


Figure 9.4 A typical sorting network. Every sorting network is made up of a series of columns, and each column contains a number of comparators connected in parallel.

Bitonic Sequence

Definition

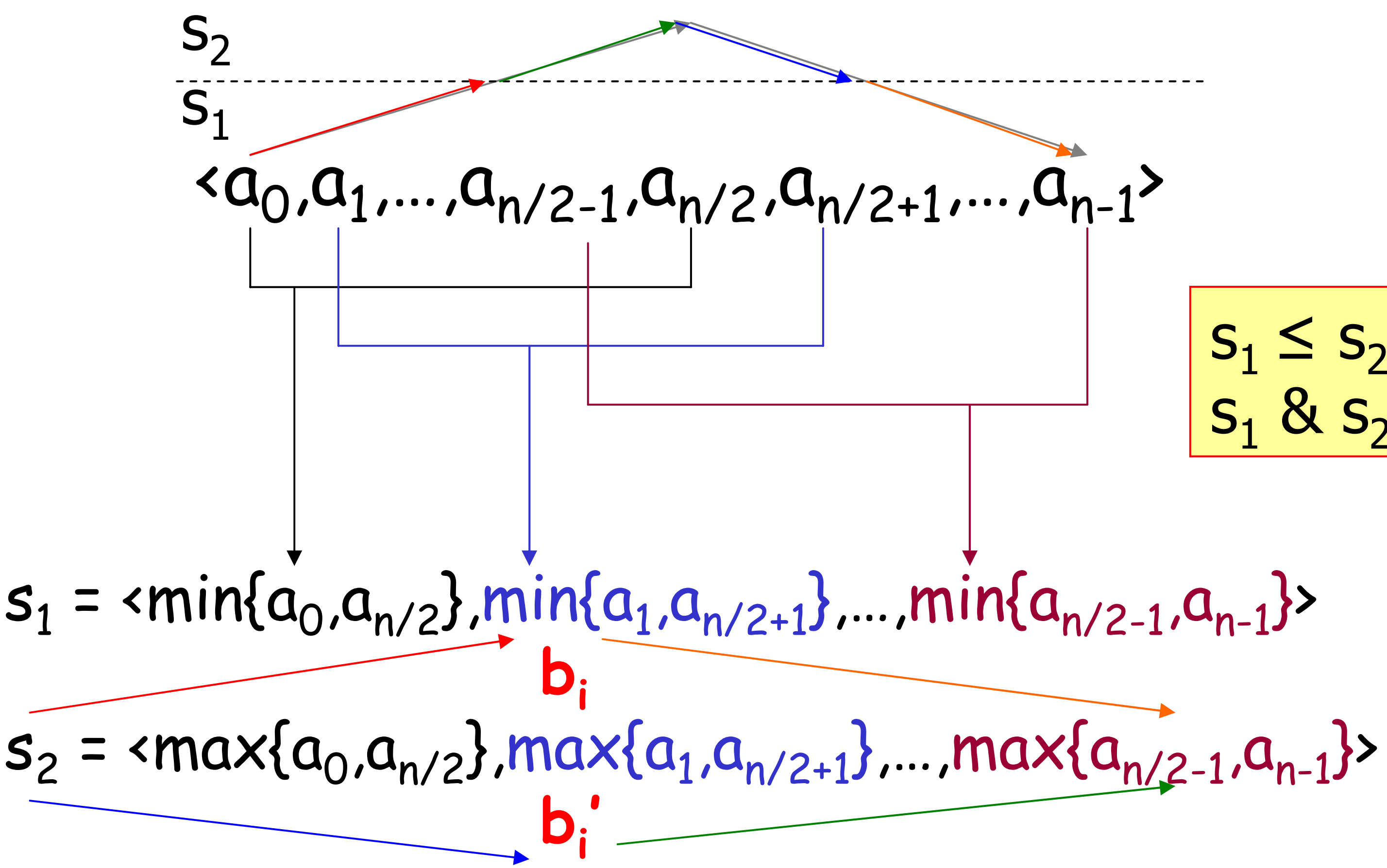
A **bitonic sequence** is a sequence of elements $\langle a_0, a_1, \ldots, a_n \rangle$ s.t.

1. $\exists i$, $0 \leq i \leq n-1$ s.t. $\langle a_0, \ldots, a_i \rangle$ is monotonically increasing and $\langle a_{i+1}, \ldots, a_{n-1} \rangle$ is monotonically decreasing,
2. or there is a cyclic shift of indices so that 1) is satisfied.

Bitonic Sort

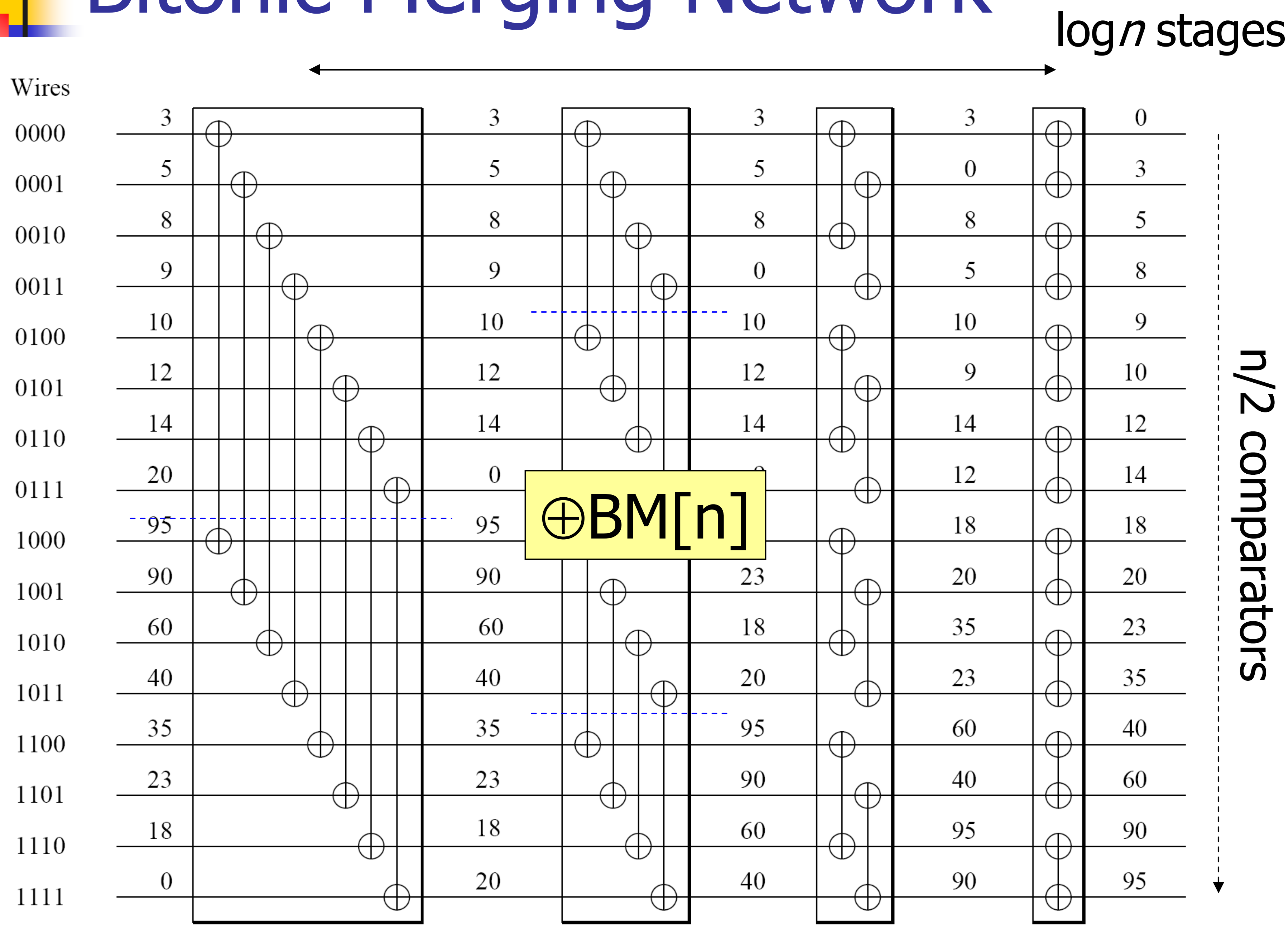
- Rearrange a bitonic sequence to be sorted.
- Divide & conquer type of algorithm (similar to quicksort) using **bitonic splits**.
  - Sorting a bitonic sequence using bitonic splits = bitonic merge.
  - But we need a bitonic sequence…

Bitonic Split



$s_1 \leq s_2$
s_1 & s_2 bitonic!

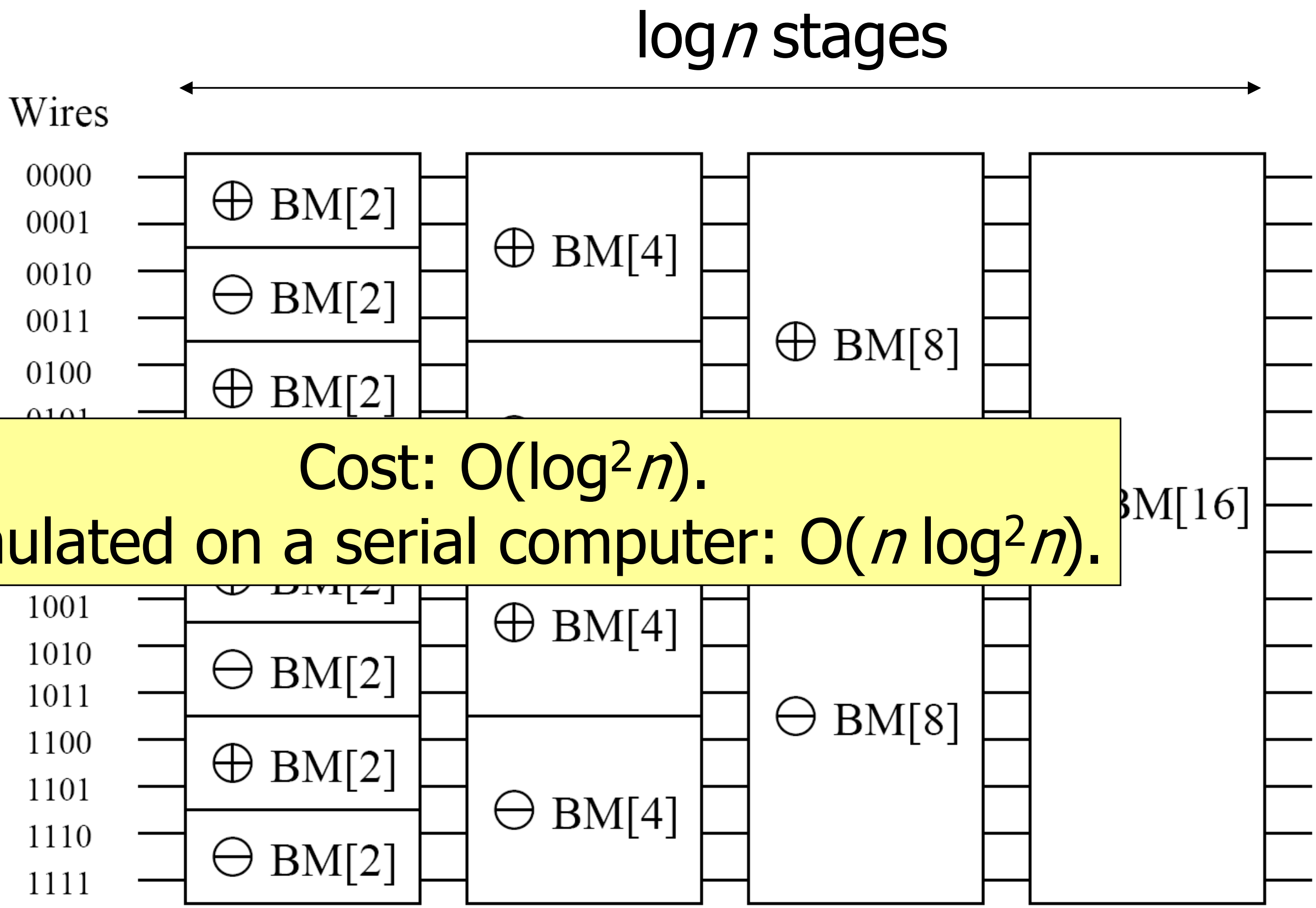
Bitonic Merging Network



Bitonic Sort

- Use the bitonic network to merge bitonic sequences of increasing length… starting from 2, etc.
- Bitonic network is a component.

Bitonic Sort



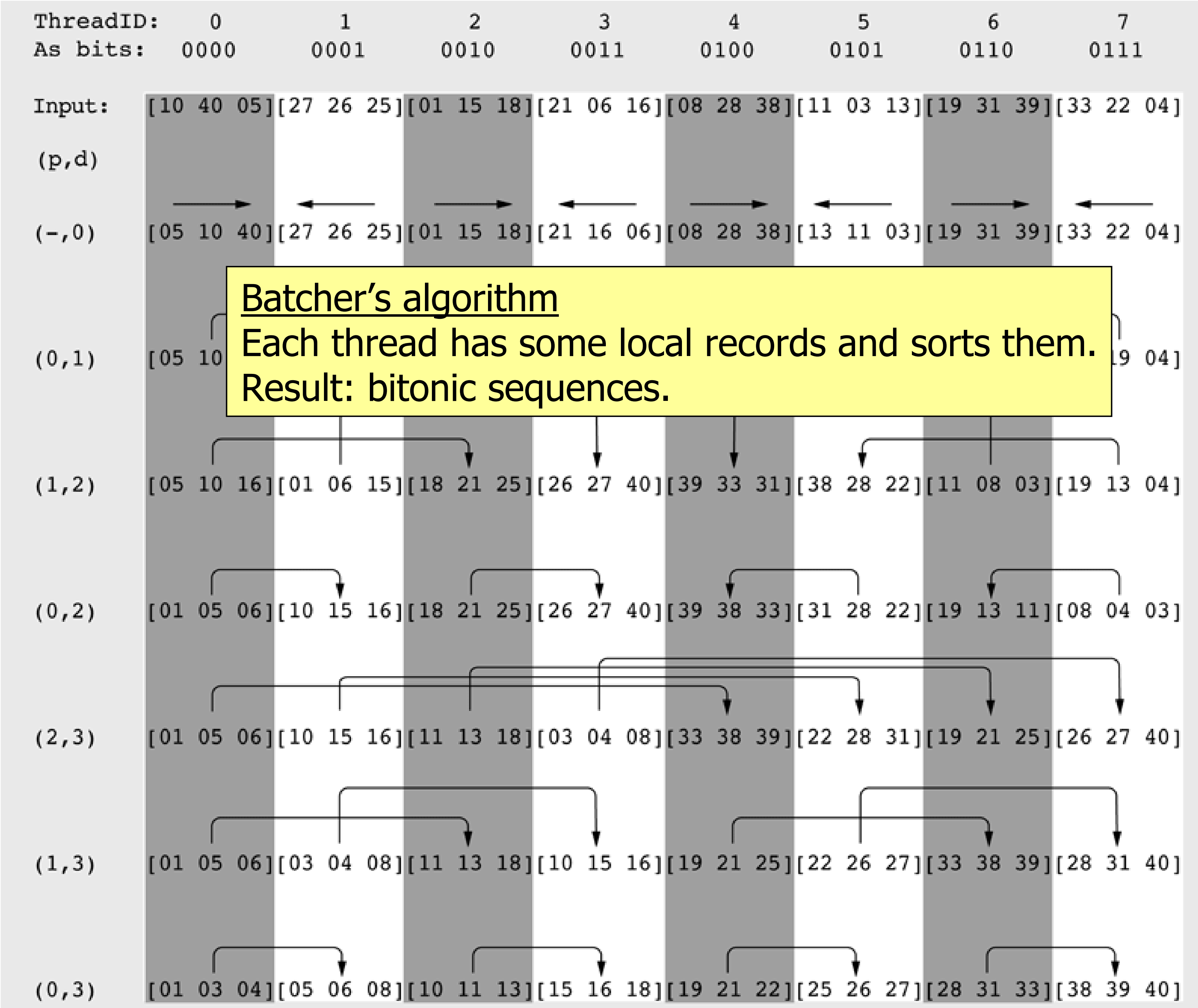
Cost: $O(\log^2 n)$.

Simulated on a serial computer: $O(n \log^2 n)$.

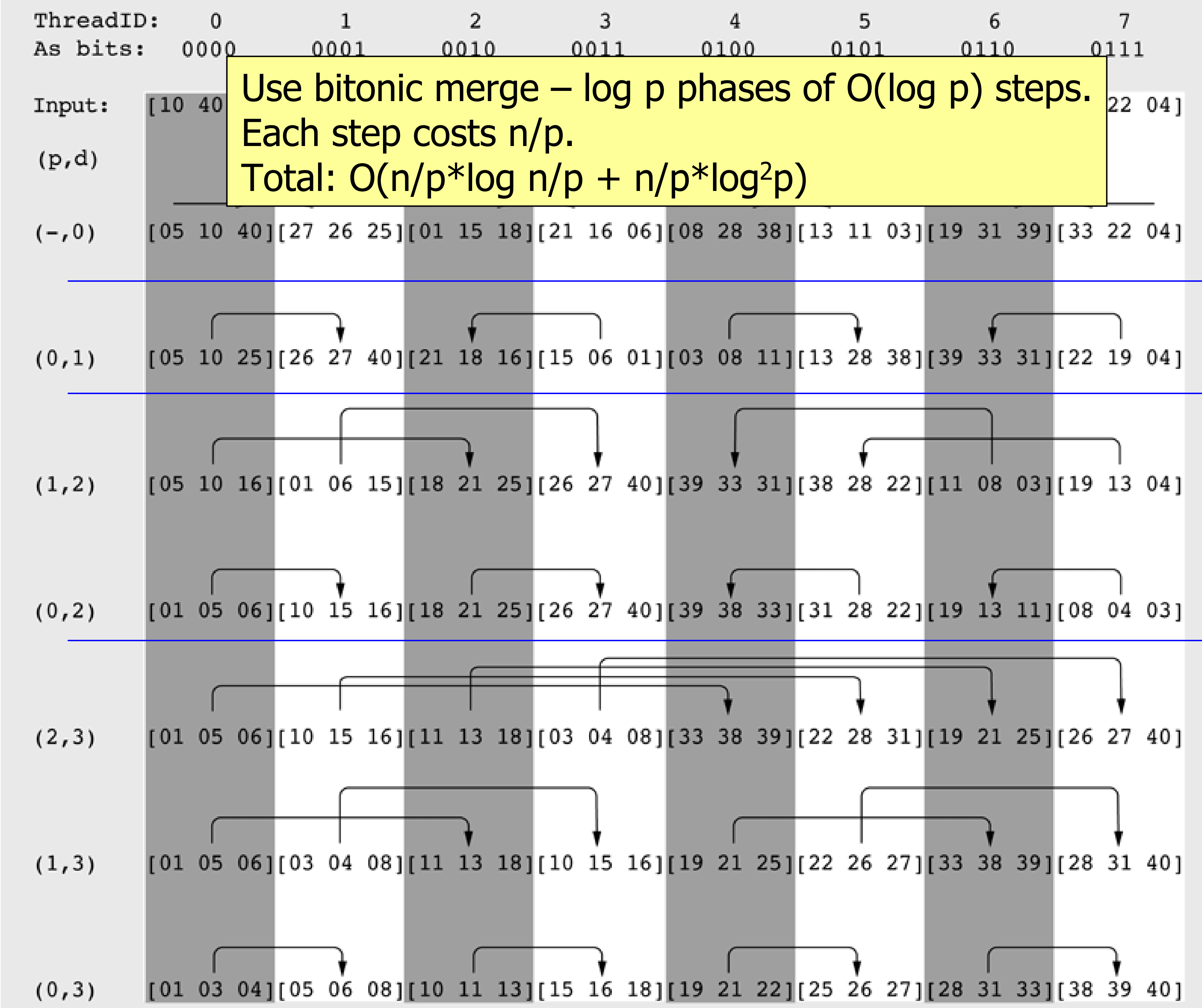
__Batcher's algorithm__

Each thread has some local records and sorts them.

Result: bitonic sequences.



Use bitonic merge – log p phases of O(log p) steps.
Each step costs n/p.
Total: $O(n/p*\log n/p + n/p*\log^2 p)$



Reflection

- Odd-even sort
  - lots of communication
  - bad complexity
- "Batch sort"
  - good complexity
  - bad scalability
- Bitonic sort
  - good complexity if p<<n ? Efficient?
  - still a lot of communication

Another Solution

- Partition the array over P.
- Use a good sorting algorithm locally.
- Use merge-sort in parallel.
- Good: simple with good complexity.
- Bad: the last step has limited parallelism.
- Still good: the last step costs nlog p.
- Even better: use tbb::parallel_for for recursive splitting and sorting (teaser).