

Parallel Techniques

- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Asynchronous Computations
- Load Balancing and Termination Detection

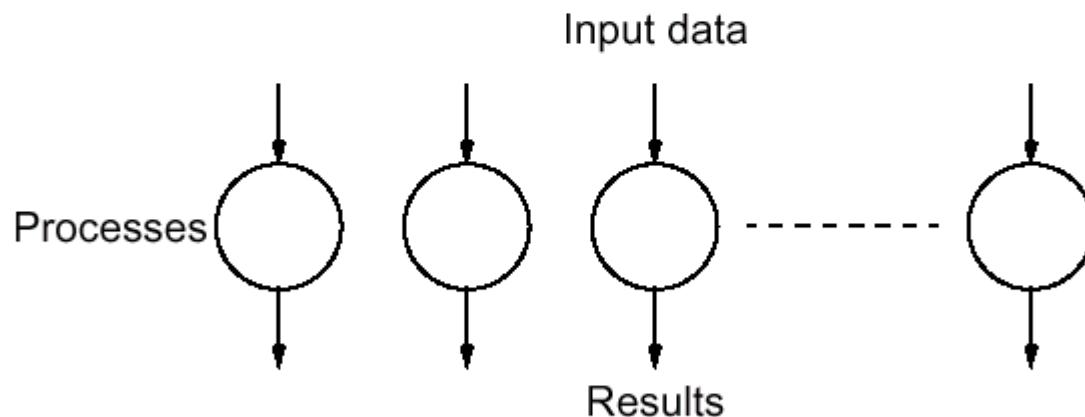
Not covered in 1st edition
of textbook

Chapter 3

Embarrassingly Parallel Computations

Embarrassingly Parallel Computations

A computation that can **obviously** be divided into a number of completely independent parts, each of which can be executed by a separate process(or).



No communication or very little communication between processes
Each process can do its tasks without any interaction with other processes

Embarrassingly (Pleasantly) Parallel Computation: “The ideal computation”

A computation that can be divided into a number of completely independent parts, each of which can be executed simultaneously by a separate processor.

- **parallelizing these problems are obvious**
 - no special techniques or algorithms: just distribute data & start processes
- **no communication between the separate processes**
 - each process need different data and produces results from its input w/o any need for results from other processes
- **gives maximum speedup**

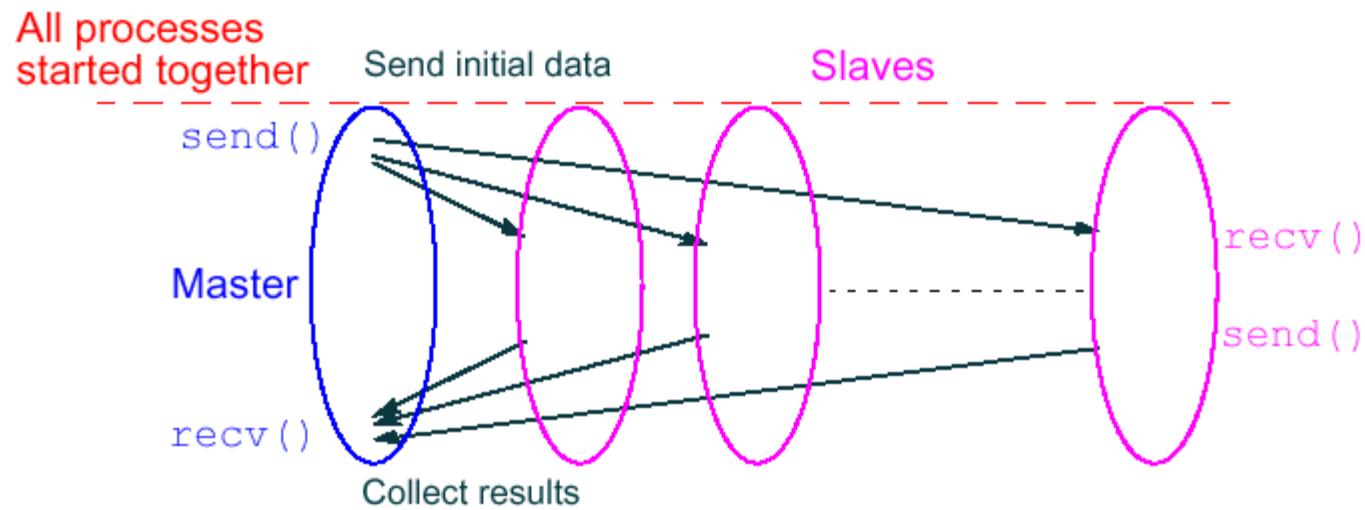
Often the independent parts are identical computations and SPMD model is appropriate

Data is not shared, but copied to each process if necessary

Nearly Embarrassingly Parallel Computation:

- requires results to be distributed and collected and combined in some way.
 - Initially and finally a single process must be operating alone
 - Master-Slave Organization (w/ dynamic or static process creation)
 - Even if the slave processes are all identical, we may not get the optimum solution if the processors are different.

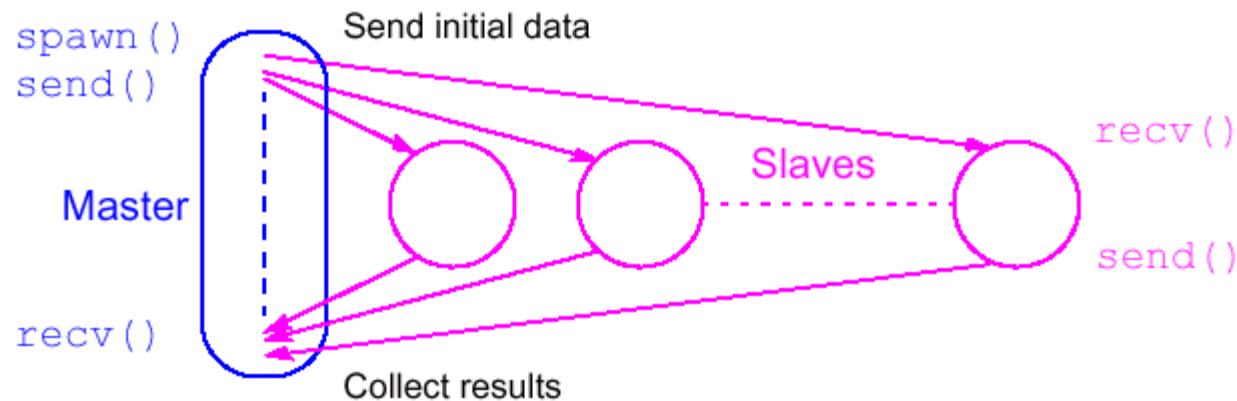
Practical embarrassingly parallel computation with static process creation and master-slave approach



MPI approach

Practical embarrassingly parallel computation with dynamic process creation and master-slave approach

Start Master initially



Embarrassingly Parallel Computation Examples

- Low level image processing
- Mandelbrot set
- Monte Carlo Calculations

Low level image processing

Many low level image processing operations only involve local data with very limited if any communication between areas of interest.

Geometrical Transformations of Images

Displayed images on a computer often originate in two ways:

- Images obtained from external sources.
 - image processing
- Images that are artificially created
 - computer graphics

Graphical operations can be performed upon a stored image:

- move, resize, rotate on regular images
- smoothing and edge detection on noisy images

Pixmap: the most basic way to store a 2-D image in which each pixel (picture element) is stored as a binary number in a 2-D array.

B/W Images: bitmap (a single bit is sufficient for each pixel)

Grayscale Images: 8 bits to represent 256 different monochrome intensities.

Color Images: Three primary colors, R/G/B, are stored as separate 8-bit numbers.
("tiff" format)

Geometric Transformations

Xformations on each pixel is totally independent from the xformations on other pixels.

- result of a xformation is simply an updated bitmap.

- Shifting:

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

- Scaling: (enlarge if $S>1$, reduce if $S<1$)

$$x' = xS_x$$

$$y' = yS_y$$

- Rotation: (through an angle θ about the origin)

$$x' = x \cos \theta + y \sin \theta$$

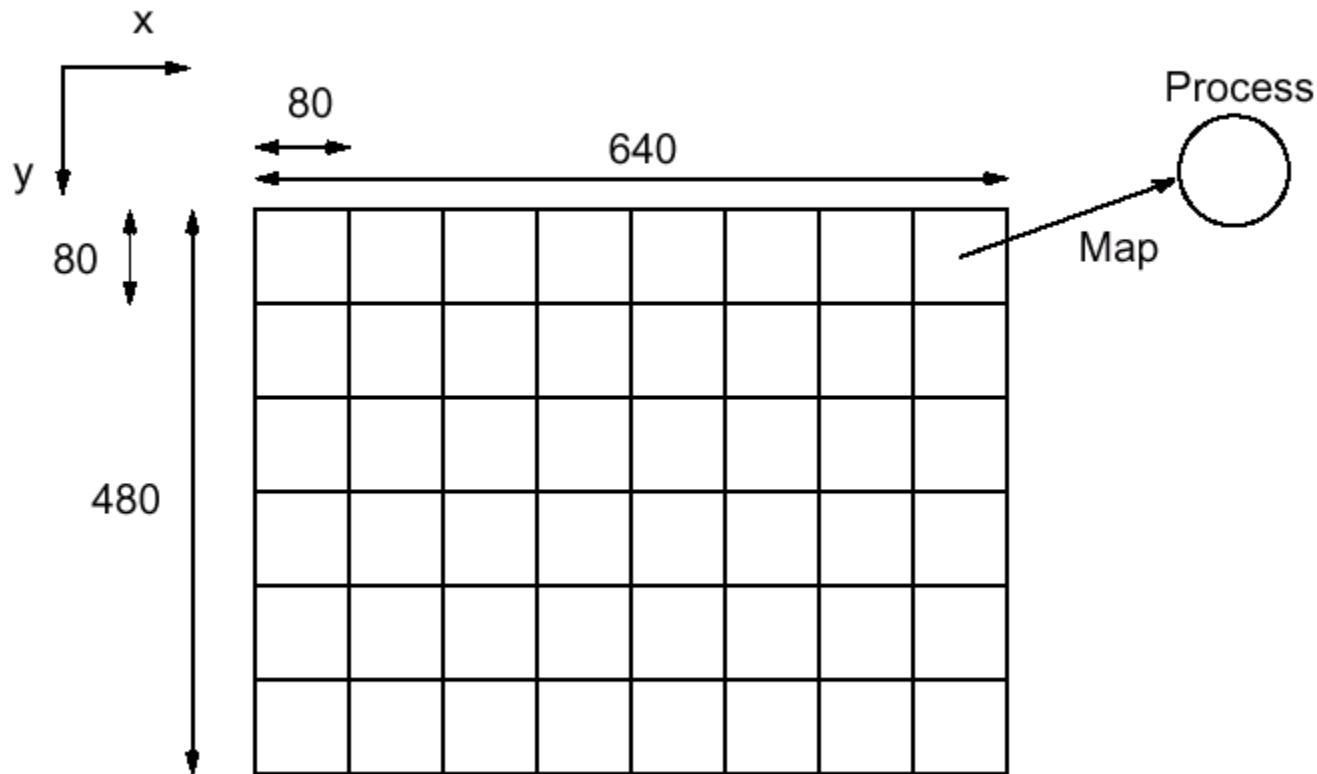
$$y' = -x \sin \theta + y \cos \theta$$

- Clipping: x_l, y_l (x_h, y_h) is the lowest (highest) values of x, y in the area to be displayed:

$$x_l \leq x' \leq x_h$$

$$y_l \leq y' \leq y_h$$

Partitioning into regions for individual processes.



Square region for each process (can also use strips)

Master

```
for (i=0; row=0; i<48; i++, row+=10)
    send(row, Pi);
                                //send row number

for (i=0; i<480; i++)
    for (j=0; j<640; j++)
        temp_map[i][j] = 0;

for (i=0; i < (640*480); i++) {
    recv(oldrow, oldcol, newrow, newcol, PANY); //accept new coords
    if (!((newrow<0) || (newrow>=480) || (newrow<0) || (newrow>=640)))
        temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i=0; i<480; i++)
    for (j=0; j<640; j++)
        map[i][j] = temp_map[i][j];
```

Slave

```
recv(row, Pmaster);
                                //receive row number

for (oldrow=row; oldrow<(row+10); oldrow++)
    for (oldcol=0; oldcol<640; oldcol++) {           // transform coords
        newrow = oldrow + delta_x;
        newcol = oldcol + delta_y;
        send(oldrow, oldcol, newrow, newcol, Pmaster); //coords to master
    }
```

Mandelbrot Set

Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

$$z_{k+1} = z_k^2 + c$$

where z_{k+1} is the $(k + 1)$ th iteration of the complex number $z = a + bi$ and c is a complex number giving position of point in the complex plane. The initial value for z is zero.

Iterations continued until magnitude of z is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of z is the length of the vector given by

$$z_{\text{length}} = \sqrt{a^2 + b^2}$$

Mandelbrot Set

Again a bit mapped image is manipulated, but this time it involves significant computation.

Mandelbrot Set is a set of points in complex plane that are quasi-stable when computed by iterating a function, such as

$$z_{k+1} = z_k^2 + c \text{ , initial value for } z \text{ is } 0.$$

Iterations continue until magnitude of $z > 2$ or iteration number > a threshold value.

$$z_{\text{length}} = \sqrt{(a^2 + b^2)}$$

In each iteration,

$$z_{\text{real}} = z_{\text{real}}^2 - z_{\text{imag}}^2 + c_{\text{real}}$$

$$z_{\text{imag}} = 2 z_{\text{real}} z_{\text{imag}} + c_{\text{imag}}$$

Mandelbrot set

Imaginary

+2

0

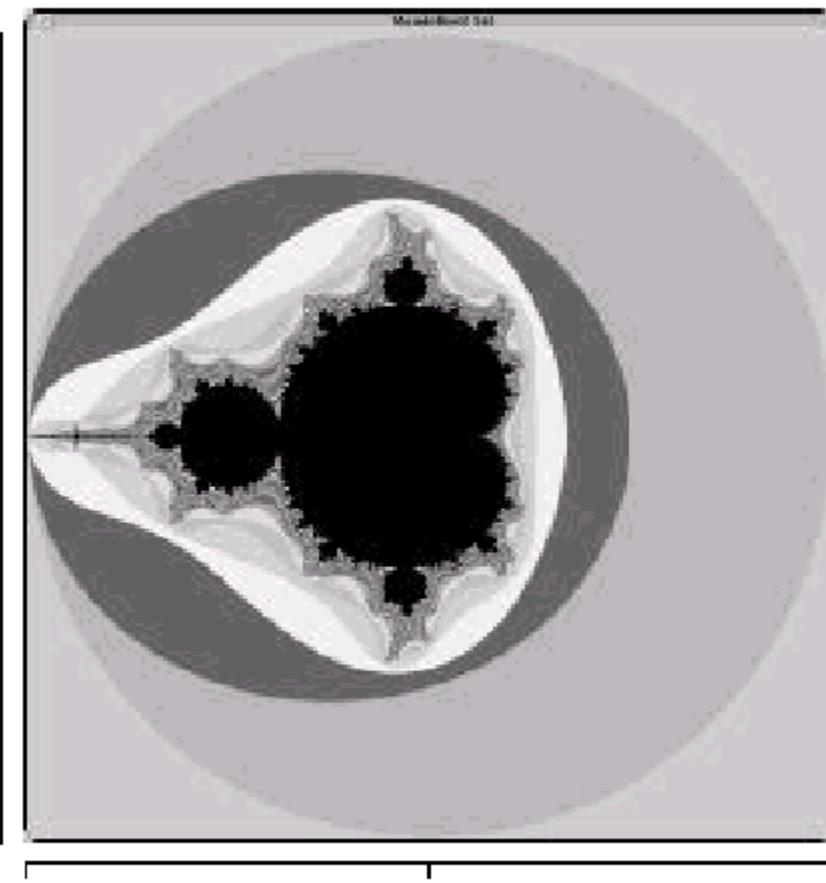
-2

-2

0

+2

Real



Sequential Code:

```
structure complex {
    float real;
    float imag;
};

int cal_pixel(complex c)
{
    int count, max;
    complex z;
    float temp, lengthsq;
    max = 256;
    z.real = 0;
    z.imag = 0;
    count = 0;
    do {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while ((lengthsq < 4.0) && (count < max));
    return count;
}
```

Obtain the actual complex plane coordinates by scaling:

```
c.real = real_min + x * (real_max - real_min)/disp_height;  
c.imag = imag_min + y * (imag_max - imag_min)/disp_width;
```

For computational efficiency:

```
scale_real = (real_max - real_min)/disp_height;  
scale_imag = (imag_max - imag_min)/disp_width;
```

Including scaling, the could could be of the form:

```
for (x=0; x<disp_width; x++) /* screen coordinates x & y */  
    c.real = real_min + ((float) x * scale_real);  
    c.imag = imag_min + ((float) y * scale_imag);  
    color = cal_pixel(c );  
    display(x, y, color);  
}
```

Parallelizing Mandelbrot Set Computation

Static Task Assignment

Simply divide the region in to fixed number of parts, each computed by a separate processor.

Not very successful because different regions require different numbers of iterations and time.

Dynamic Task Assignment

Have processor request regions after computing previous regions

Static Task Assignment

Master

```
for (i=0; row=0; i<48; i++, row+=10)
    send(row, Pi);
                                // send row number

for (i=0; i < (640*480); i++) {
    recv(&c, &color, PANY);
    display(c, color);
}
```

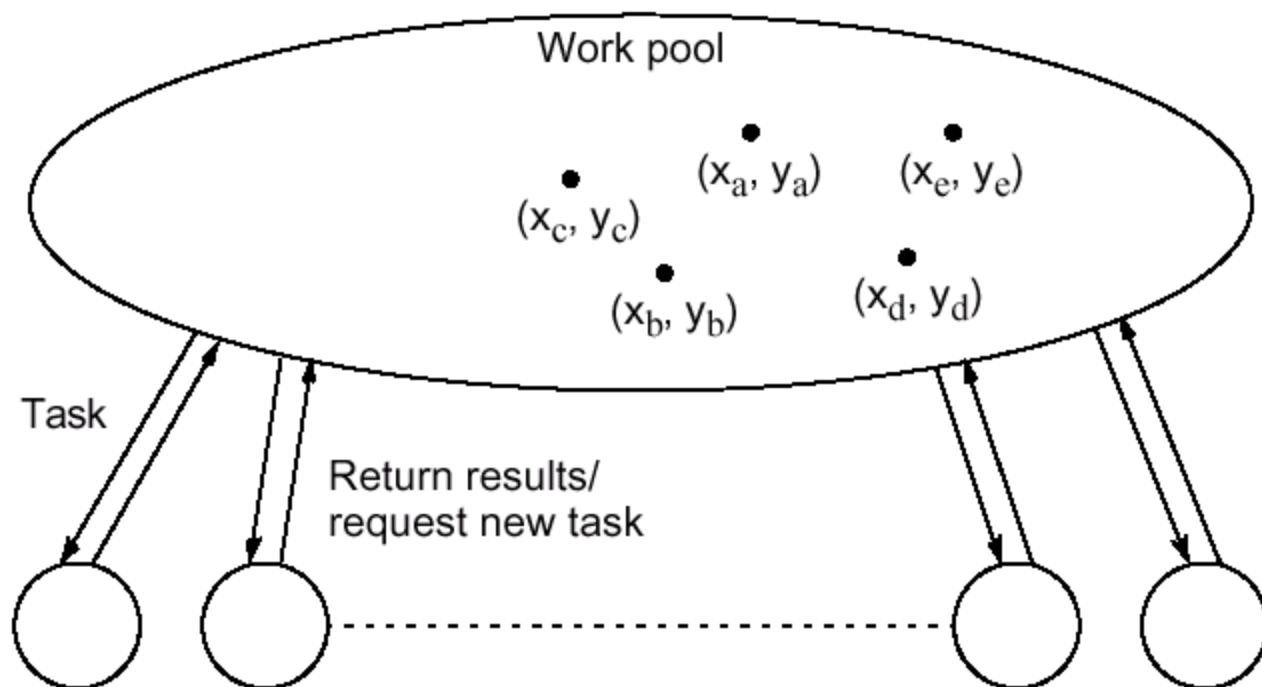
Slave

```
recv(row, Pmaster);
                                // receive row number

for (x=0; x<disp_width; x++)
    for (y=row; y<(row+10); y++) {
        c.real = real_min + ((float) x * scale_real);
        c.imag = imag_min + ((float) y * scale_imag);
        send(&c, &color, Pmaster);
                                // send coords & color to master
    }
```

Dynamic Task Assignment

Work Pool/Processor Farms



Dynamic Task Assignment - Work Pool/Processor Farms

Mandelbrot Set requires significant iterative computation for each pixel:

- # iterations will generally be different for each pixel.
 - computers may be of different type, or operate at different speeds.
- ⇒ **Hence some processors may complete their assignment before others**

Ideally we want all processors to finish together, achieving a system efficiency of 100%, which can be addressed using load balancing.

Different sizes of regions could be assigned to different processors, but this would not be satisfactory:

- we may not a priori each processor's computational speed,
- we would have to know the exact time it takes for each processor to compute each pixel.

Workpool Approach to Dynamic Task Assignment

- Processors are supplied with work when they become idle
 - Sometimes called processor farm, when all processors are the same type.
- Workpool holds a collection, pool, of tasks to be performed.
 - in our case, the set of pixels forms the tasks
 - when a processor has computed the color for the pixel, it returns the color and requests a further pair of pixel coordinates from the work pool
 - when all pixel coordinates have been taken, we than have to wait for all the processors to complete and report in for more pixel coordinates.

Sending pairs of coordinates of individual pixels will result in excessive communication \Rightarrow group them.

In workpool solution, some pixels will be generated before others.

Master

```
count = 0;
row = 0;
for (k=0; k<procno; k++) {
    send(&row, Pk, data_tag); //send row #
    count++;
    row++;
}
do {
    recv(&slave, &r, color, PANY, result_tag);
    count--;
    if (row < disp_height) {
        send(&row, Pslave, data_tag);
        row++;
        count++;
    } else
        send(&row, Pslave, terminator_tag);
    rows_recv++;
    display(r, color);
} while (count > 0);
```

Slave

```
recv(y, Pmaster, ANYTAG, source_tag); //receive
// 1st row to compute
while (source_tag == data_tag) {
    c.imag = imag_min + ((float) y *
scale_imag);
    for (x=0; x<disp_width; x++) {
        c.real = real_min + ((float)x*scale_real);
        color[x] = cal_pixel(c );
    }
    // send row colors to master
    send(&i, &color, Pmaster, result_tag);
    recv(y, Pmaster, source_tag);
}
```

Monte Carlo Methods

Another embarrassingly parallel computation.

Monte Carlo methods use of random selections.

Example - To calculate π

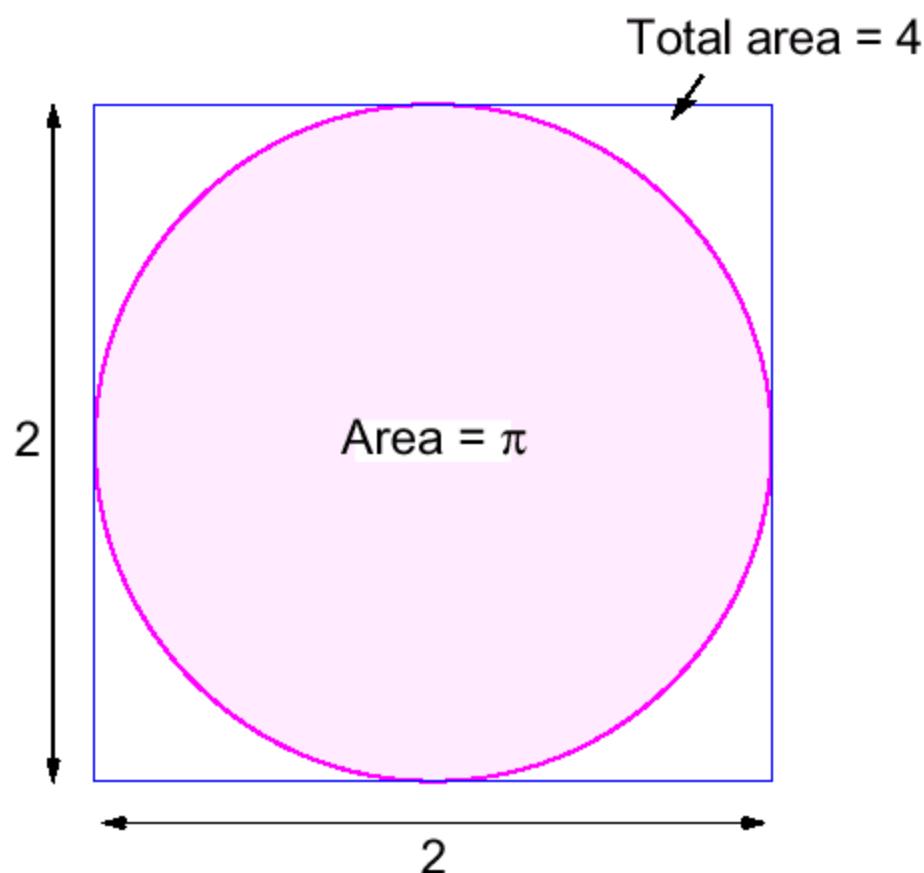
Circle formed within a square, with unit radius so that square has sides 2×2 . Ratio of the area of the circle to the square given by

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

Points within square chosen randomly.

Score kept of how many points happen to lie within circle.

Fraction of points within the circle will be $\pi/4$, given a sufficient number of randomly selected samples.



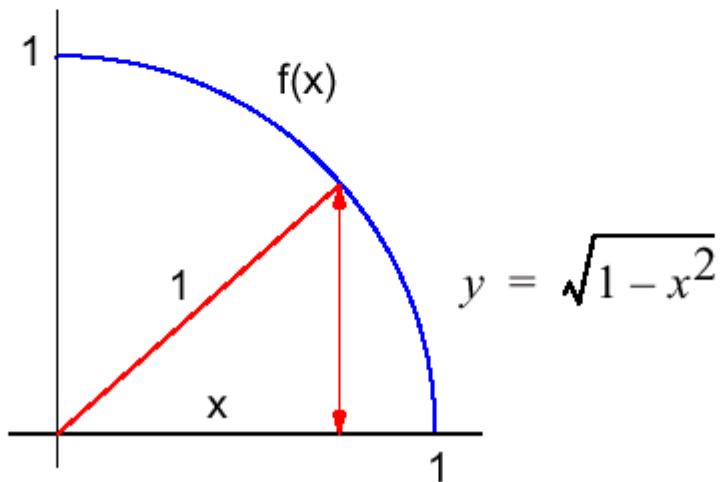
Computing an Integral

One quadrant of the construction can be described by integral

$$\int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$

Random pairs of numbers, (x_r, y_r) generated, each between 0 and 1.

Counted as in circle if $y_r \leq \sqrt{1-x_r^2}$; that is, $y_r^2 + x_r^2 \leq 1$.



Sequential Code: for computing $f(x) = x^2 - 3x$

```
sum = 0;  
for (i=0; i<N; i++) {           // N random samples  
    xr = rand_v(x1, x2);        // generate next random value  
    sum += xr * xr - 3 * xr;    // compute f( xr)  
}  
area = sum / N;
```

randv(x1, x2) returns a pseudorandom number between x1 and x2.

Parallel Random Number Generation

- The most popular way of creating a pseudorandom number sequence is by evaluating x_{i+1} from a carefully chosen function of x_i .
 - The key is to find a function that will create a very large sequence with the correct statistical properties:

$$x_{i+1} = (a x_i + c) \bmod m \quad \textit{Linear Congruential Generator}$$

where a , c , and m are constants chosen to create a sequence that has similar properties to truly random sequences.

Even though it appears that the pseudorandom number computation is sequential in nature, as each number is calculated from the previous number, a parallel formulation is possible.

$$x_{i+1} = (a x_i + c) \bmod m$$

$$x_{i+k} = (A x_i + C) \bmod m$$

where $A = a^k \bmod m$, $C = c(a^{k-1} + a^{n-2} + \dots + a^1 + a^0) \bmod m$, and k is a selected “jump” constant.

Chapter 4

Partitioning and Divide-and-Conquer Strategies

Chapter 4 – Partitioning and Divide-and-Conquer Strategies

Two fundamental techniques in parallel programming:

1. Partitioning

- divide problem into separate parts and compute each part separately.

2. Divide and conquer

- applies partitioning in a recursive manner by continually dividing the problem into smaller and smaller parts before solving the smaller parts and combining the results

Partitioning Strategies

- Partitioning divides the problem into parts.
 - it is the basis of all parallel programming.
 - most partitioning strategies require the results of the parts to be combined later on to obtain the desired result.
 - Data Partitioning or Domain Decomposition:
 - dividing the data and operating upon the divided data concurrently.
This is the main strategy for parallel programming.
 - Functional Decomposition:
 - dividing the program into independent functions and executing the functions concurrently.

Example: Adding Numbers

We are to add a sequence of numbers x_0, \dots, x_{n-1}

1. divide sequence into m parts of n/m numbers each
(distribute sequences to corresponding processors)
2. m processors can each add one sequence
independently to create partial sums
3. partial sums are added together on master to form
the final sum.

Master-slave approach.

- How to broadcast data?
 - broadcast whole list of numbers to every slave ?
 - send the specific numbers to each slave ?

Broadcast operation will have

a single startup time rather than separate startup times
when using multiple send routines and may be preferable.

The code using separate send() s and recv () s:

Master:

```
s = n / m;           /* number of items on each slave */
for (i=0, x=0; i<m; i++, x += s)
    send(&numbers[x], s, Pi);

result = 0;
for (i=0; i<m; i++) { /* wait for results from slaves */
    recv(&part_sum, PANY);
    sum += part_sum;      /* accumulate partial sums */
}
```

Slave:

```
recv(numbers, s, Pmaster); /* receive s items from master */
sum = 0;
for (i=0; i<s; i++)      /* add numbers */
    part_sum += numbers[i];
send(&part_sum, Pmaster); /* send result to master */
```

Remarks:

- Slaves are identified by a process ID in PVM, that can be usually obtained by calling a library routine, e.g., `pvm_spawn(...)`
- Slave number is the rank within the group in MPI
 - an integer from 0 to $m-1$, where m is the # processes in the group.

The code using a broadcast or multicast routine:

Master:

```
s = n / m;  
bcast(numbers, s, Pslave_group);  
  
result = 0;  
for (i=0; i<m; i++) {  
    recv(&part_sum, PANY);  
    sum += part_sum;  
}
```

Slave:

```
bcast(numbers, s, Pslave_group);  
start = slave_number * s;  
end = start + s;  
sum = 0;  
for (i=start; i<end; i++)  
    part_sum += numbers[i];  
send(&part_sum, Pmaster);
```

The code using scatter and reduce routines, if available:

Master:

```
s = n / m;  
scatter(numbers, &s, Pgroup, root=master);  
reduce_add(&sum, &s, Pgroup, root=master);
```

Slave:

```
scatter(numbers, &s, Pgroup, root=master);  
  
sum = 0;  
for (i=0; i<s; i++)  
    part_sum += numbers[i];  
reduce_add(&part_sum, &s, Pgroup, root=master);
```

Similar to adding numbers, we could do other operations as well:

- find maximum number
- find number of occurrences of a number

Analysis.

Sequential Implementation.

requires $n-1$ additions or $O(n)$.

Parallel Implementation.

- Phase 1 - Communication. m slave processes reads their n/m numbers.

$$t_{\text{comm1}} = m(t_{\text{startup}} + (n/m)t_{\text{data}}) \quad (\text{Using send/recv})$$

$$t_{\text{comm1}} = t_{\text{startup}} + n t_{\text{data}} \quad (\text{Using scatter})$$

- Phase 2 - Computation. Slaves concurrently add n/m number together.

$$t_{\text{comp1}} = n/m - 1$$

- Phase 3 - Communication. Slaves return partial sums to master

$$t_{\text{comm2}} = m(t_{\text{startup}} + t_{\text{data}}) \quad (\text{Using send/recv})$$

$$t_{\text{comm2}} = t_{\text{startup}} + m t_{\text{data}} \quad (\text{Using gather and reduce})$$

- Phase 4 - Computation. Final accumulation.

$$t_{\text{comp1}} = m - 1$$

Analysis (continued)

- Overall $t_p = O(n+m)$ and worse than t_s
- What about speedup?

$$S = \frac{t_s}{t_p} = \frac{(n - 1)}{\frac{n}{m} + m - 2}$$

The speedup tends to m for large n .

- The speedup will be quite low for increasing # slaves, as m slaves are idle in the 2nd phase forming the final result.
- Ideally we want all the processes to be active all of the time.

Divide and Conquer

1. Divide the problem into smaller subproblems
2. Divide subproblems into still smaller subproblems
recursively until the tasks cannot be broken down into smaller parts.
3. Combine the results of elementary tasks, continue combining results of larger and larger tasks recursively.

Example: Sequential Recursive definition for adding a list of numbers:

- *What is the termination condition below?*

```
int add(int *s)
{
    if (number(s) <= 2) return(n1 + n2);
    else {
        Divide(s, s1, s2); /* divide s into two parts: s1&s2*/
        part_sum1 = add(s1);
        part_sum2 = add(s2);
        return(part_sum1 + part_sum2);
    }
}
```

- **The same divide-and-conquer method can be used for**
 - sorting a list, for finding the maximum number in a list, etc.

When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree.

- The tree is traversed
 - downward as calls are made (preorder traversal)
 - and upward when the calls return

Consider the following case:

- The tree is not a complete binary tree (not perfectly balanced with all bottom nodes at the same level)
 - happens if number of parts is not a power of 2

Chapter 4

Partitioning and Divide-and-Conquer Strategies

Divide and Conquer

1. Divide the problem into smaller subproblems
2. Divide subproblems into still smaller subproblems
recursively until the tasks cannot be broken down into smaller parts.
3. Combine the results of elementary tasks, continue combining results of larger and larger tasks recursively.

Example: Sequential Recursive definition for adding a list of numbers:

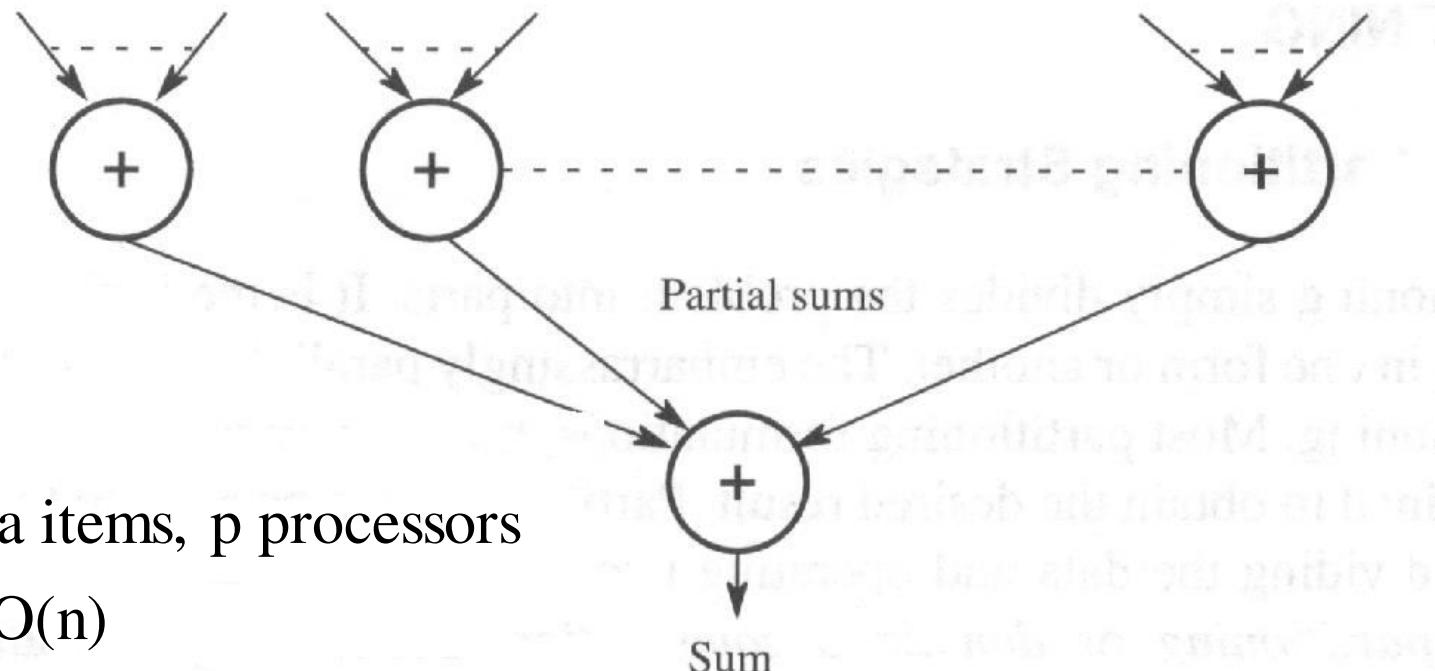
- *What is the termination condition below?*

```
int add(int *s)
{
    if (number(s) <= 2) return(n1 + n2);
    else {
        Divide(s, s1, s2); /* divide s into two parts: s1&s2*/
        part_sum1 = add(s1);
        part_sum2 = add(s2);
        return(part_sum1 + part_sum2);
    }
}
```

- **The same divide-and-conquer method can be used for**
 - sorting a list, for finding the maximum number in a list, etc.

Adding numbers

$x_0 \dots x_{(n/m)-1} \quad x_{n/m} \dots x_{(2n/m)-1} \quad \dots \quad x_{(m-1)n/m} \dots x_{n-1}$



n data items, p processors

$$t_s = O(n)$$

$$t_p = O(n/p) \text{ if data on each proc } \Rightarrow S = t_s / t_p = O(p)$$

$$t_p = O(n + n/p) \text{ if data needs broadcasting } \Rightarrow S = t_s / t_p = o(1)$$

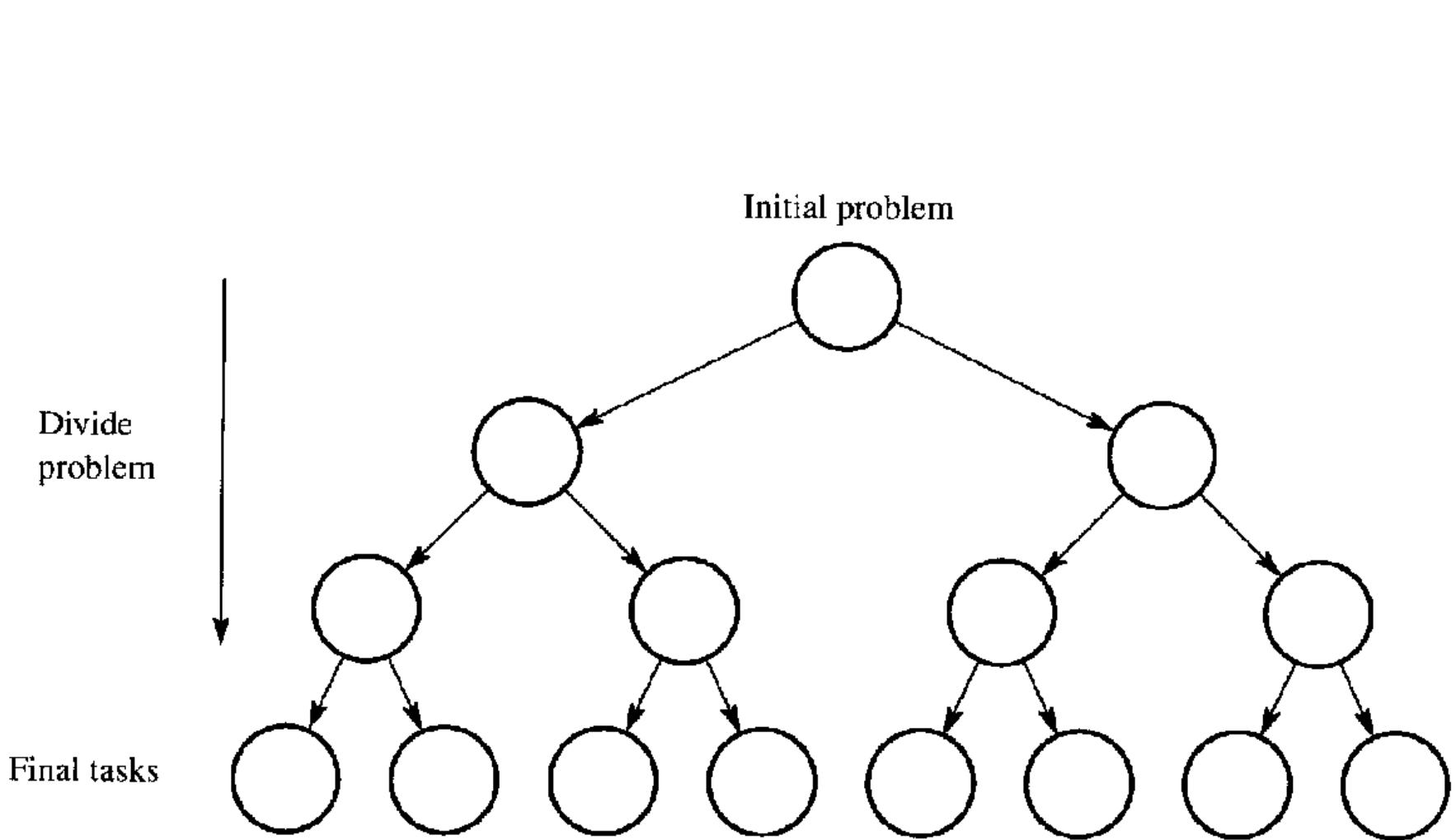
When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree.

- The tree is traversed
 - downward as calls are made (preorder traversal)
 - and upward when the calls return

Consider the following case:

- The tree is not a complete binary tree (not perfectly balanced with all bottom nodes at the same level)
 - happens if number of parts is not a power of 2

Initial problem



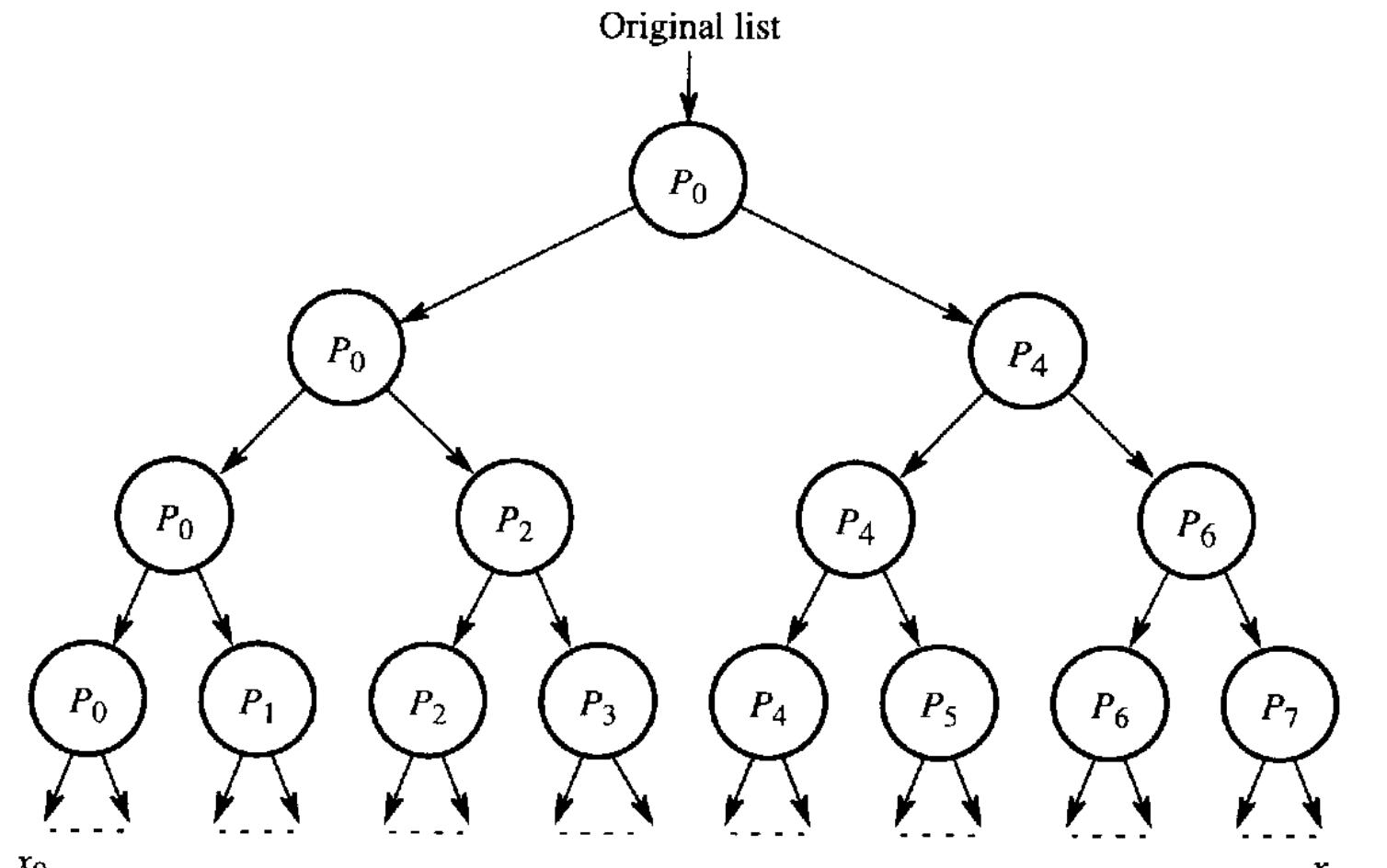
Parallel Implementation.

- In a serial implementation, only one node of the tree can be visited at a time.
- In parallel solution, several parts of the tree can be traversed simultaneously.
- Let's do it without using recursion.
 - The key point is that the construction is a tree.
 - 1. Inefficient Solution: Assign one processor to each node in the tree.
 - $2^{m+1} - 1$ processors to divide the task into 2^m parts.
 - Each processor would only be active at one level in the tree
(Very inefficient solution!)

2. A More Efficient Solution: reuse processors at each level of the tree (Figure 4.3)

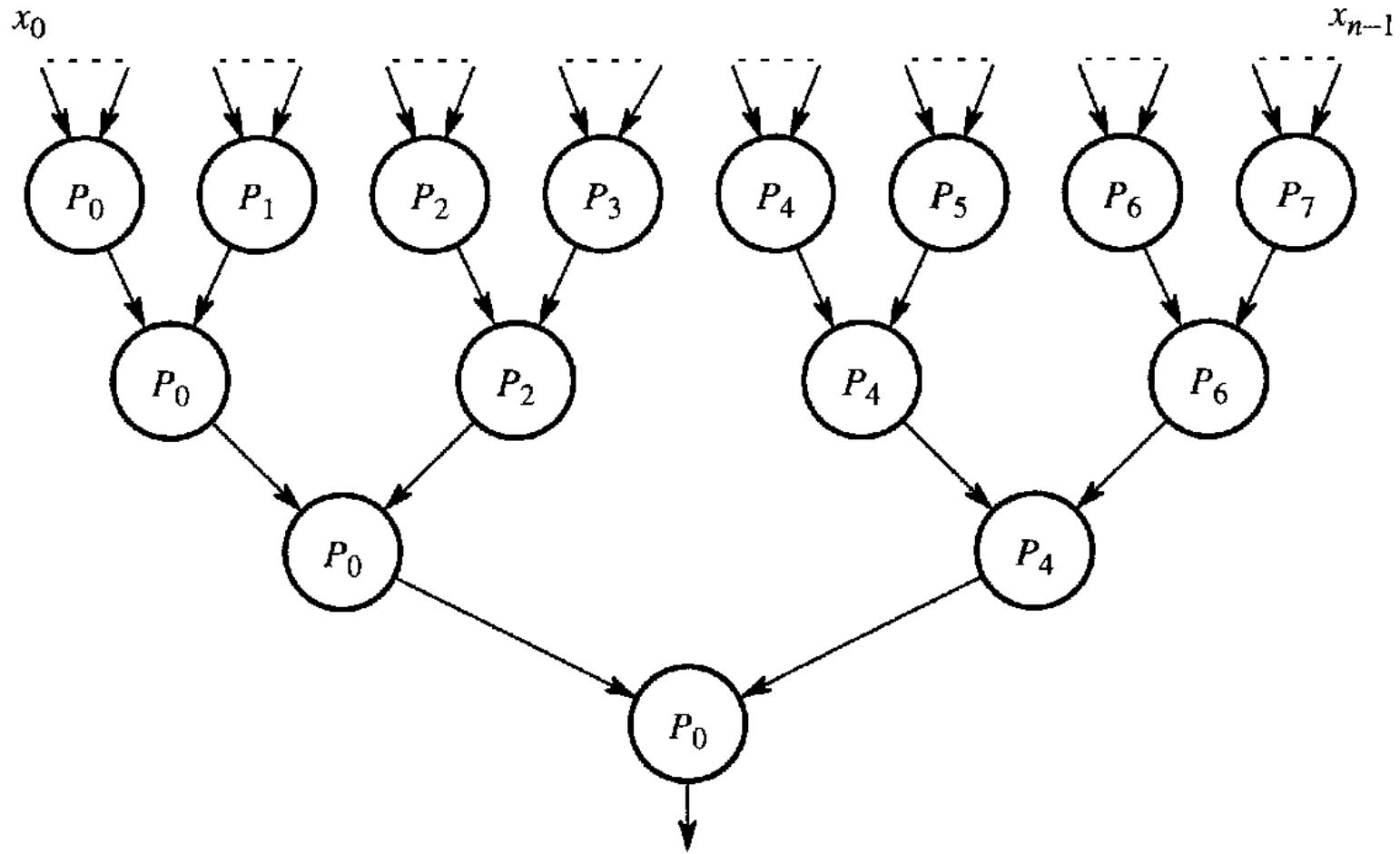
- division stops when the total # processors is committed
 - until then, at each stage each processor keeps the half of the list and passes on the other half.
 - Each list at final stage has n/p numbers
 - There are $\log p$ levels in the tree.
-
- The combining act of summation of the partial sums can be done similarly but in reverse order.
 - The constructions are the same as the binary hypercube broadcast and gather algorithms.
 - We use the communicating neighbors from their binary addresses.

Parallel Recursion



$$t_{\text{comm}} = O(n/2 + n/4 + \dots + n/p) = O(n)$$

$$t_{\text{comp}} = O(n/2 + n/4 + \dots + n/p) = O(n) \quad S=o(1)$$



$$t_{\text{comm}} = O(1 + 1 + \dots + 1) = O(\log p)$$

$$t_{\text{comp}} = O(1 + 1 + \dots + 1) = O(\log p)$$

$$S = O(n / \log p)$$

With 8 processors:

Process P0:

```
divide(s1, s1, s2); /*division*/  
send(s2, P4);      ──────────→  
divide(s1, s1, s2);  
send(s2, P2);  
divide(s1, s1, s2);  
send(s2, P1);
```

```
part_sum = *s1; /*combining*/
```

```
recv(&part_sum1, P1);  
part_sum += part_sum1;  
recv(&part_sum1, P2);  
part_sum += part_sum1;  
recv(&part_sum1, P4); ←  
part_sum += part_sum1;
```

With 8 processors:

Process P4:

```
recv(s1, P0); /*division*/  
divide(s1, s1, s2);  
send(s2, P6);  
divide(s1, s1, s2);  
send(s2, P5);
```

```
part_sum = *s1; /*combining*/
```

```
recv(&part_sum1, P5);  
part_sum += part_sum1;  
recv(&part_sum1, P6);  
part_sum += part_sum1;  
send(&part_sum, P0);
```

Analysis.

Assume n is a power of 2 and ignore the t_{startup} for simplicity.

The division phase only contains communication, required computation is minimal.

The combining phase requires both communication and computation to add the partial sums received and pass on the result.

Communication. $\log p$ steps with p processes.

$$t_{\text{comm1}} = (n/2) t_{\text{data}} + (n/4) t_{\text{data}} + (n/8) t_{\text{data}} + \dots + (n/p) t_{\text{data}} = (n(p-1)/p) t_{\text{data}}$$

$\Rightarrow t_{\text{comm1}}$ is marginally better than a simple broadcast.

Combining phase , only one data item (partial sum) is sent in each message

$$t_{\text{comm2}} = t_{\text{data}} \log p$$

Computation.

$$t_{\text{comp}} = (n/p) + \log p, \text{ with time complexity of } O(n) \text{ for constant } p.$$

For large n and variable p , we get $O(n/p)$.

M-ary Divide and Conquer

Divide and conquer can also be applied where a task is divided into more than two parts at each stage.

For example, let's divide a task into four parts:

```
int add(int *s)
{
    if (number(s) <= 4) return (n1 + n2 + n3 + n4);
    else {
        Divide(s, s1, s2, s3, s4);
        part_sum1 = add(s1);
        part_sum2 = add(s2);
        part_sum3 = add(s3);
        part_sum4 = add(s4);
        return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
    }
}
```

Tree Types

Quadtree: a tree in which each node has four children

- has particular applications in decomposing 2-D regions into 4 subregions.

Octtree: a tree in which each node has 8 children.

- has application for dividing a 3-D space recursively.

m-ary tree: a tree in which each node has *m* parts.

- suggests greater parallelism.

Sorting with Bucket Sort

- Most sequential algorithms are based upon the compare and exchange of pairs of numbers.
- In contrast, bucket sort is naturally a partitioning method:
 - works well if the original numbers are uniformly distributed across a known interval, say 0 to $a-1$.
 - This interval is divided into m equal regions and one “bucket” is assigned to hold numbers that fall within each region: m buckets.
 - We can use $m=n$ buckets, i.e., one bucket for each number
 - we can develop this into a divide-and-conquer method by continually dividing the buckets into smaller buckets:
 - here we will use a limited number of buckets.

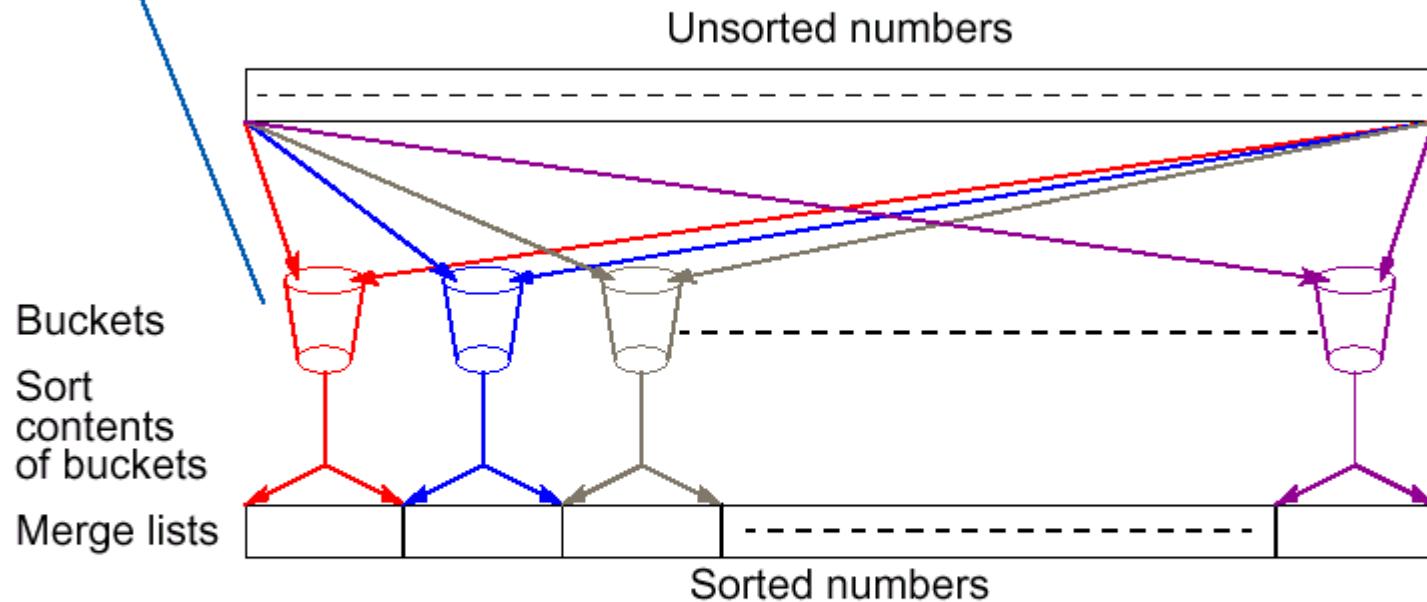
Parallel Algorithm

- Four phases:
 - Partition numbers.
 - Sort into small buckets.
 - Send to large buckets.
 - Sort large buckets.

- Phase 1. Computation and Communication.
 - $t_{comp1} = n$
 - $t_{comm1} = t_{startup} + t_{data} n$
- Phase 2. Computation.
 - $t_{comp2} = n/p$
- Phase 3. Communication.
 - $t_{comm3} = p(p-1)(t_{startup} + (n/p^2)t_{data})$
 - $t_{comm3} = (p-1)(t_{startup} + (n/p^2)t_{data})$
- Phase 4. Computation.
 - $t_{comp4} = (n/p)\log(n/p)$
- Overall.

Sequential Bucket sort

One “bucket” assigned to hold numbers that fall within each region.
Numbers in each bucket sorted using a sequential sorting algorithm.



m buckets , n numbers

Works well if the original numbers uniformly distributed across a known interval, say 0 to $a - 1$.

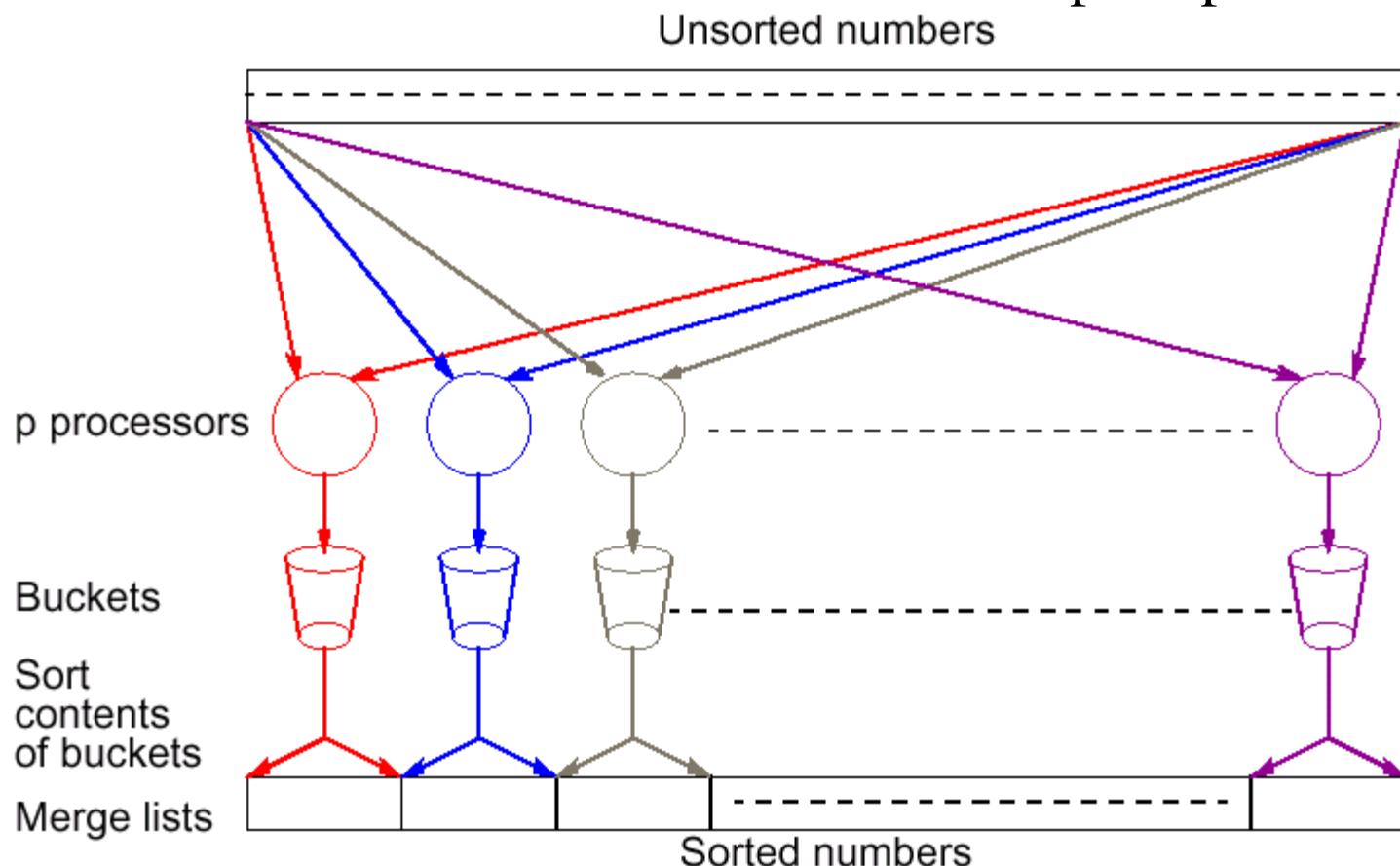
$$t_s = O(n + m((n/m) \log (n/m))) = O(n \log(n/m))$$

Parallel version of bucket sort

Simple approach

Assign one processor for each bucket.

m buckets , n numbers,
p=m processors



$$t_p = O(n + (n/p) \log (n/p))$$

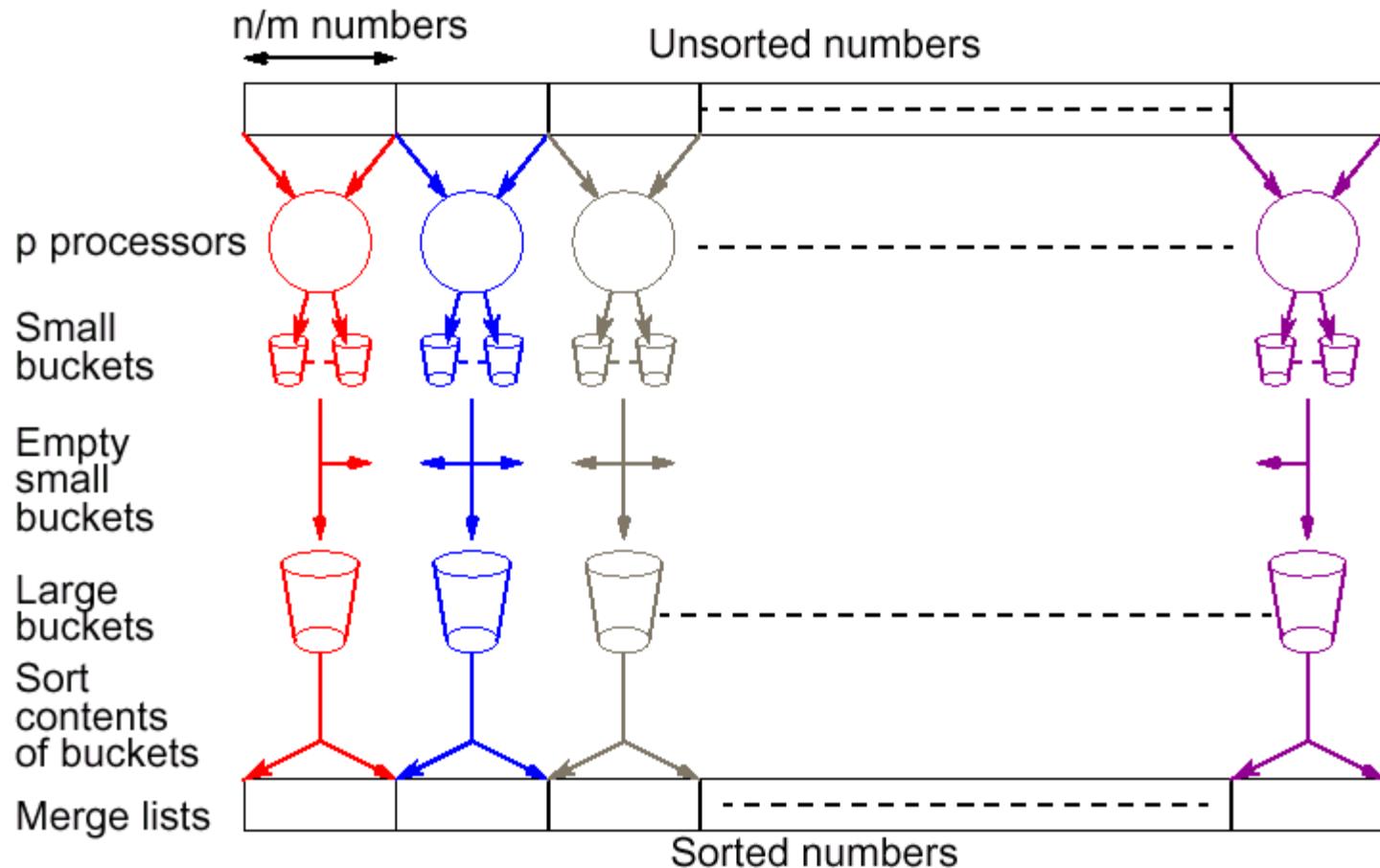
Further Parallelization

Partition sequence into m regions, one region for each processor.

Each processor maintains p “small” buckets and separates the numbers in its region into its own small buckets.

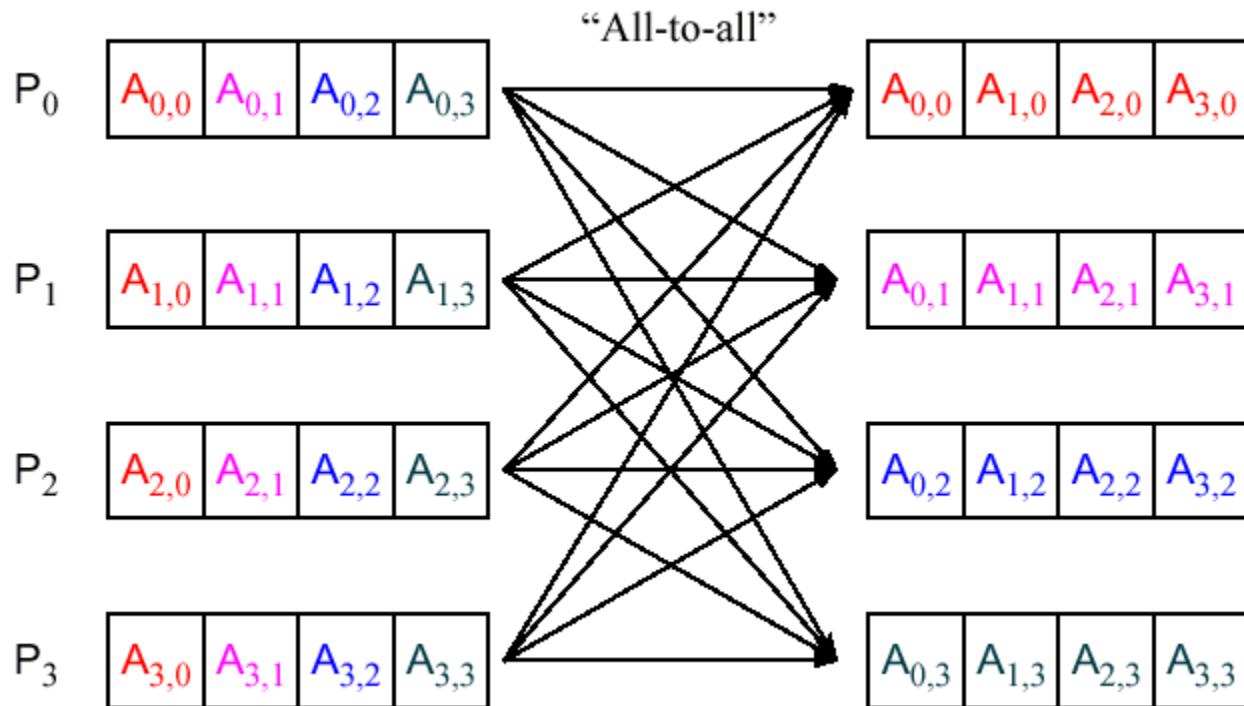
Small buckets then emptied into p final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket i to processor j).

Another parallel version of bucket sort

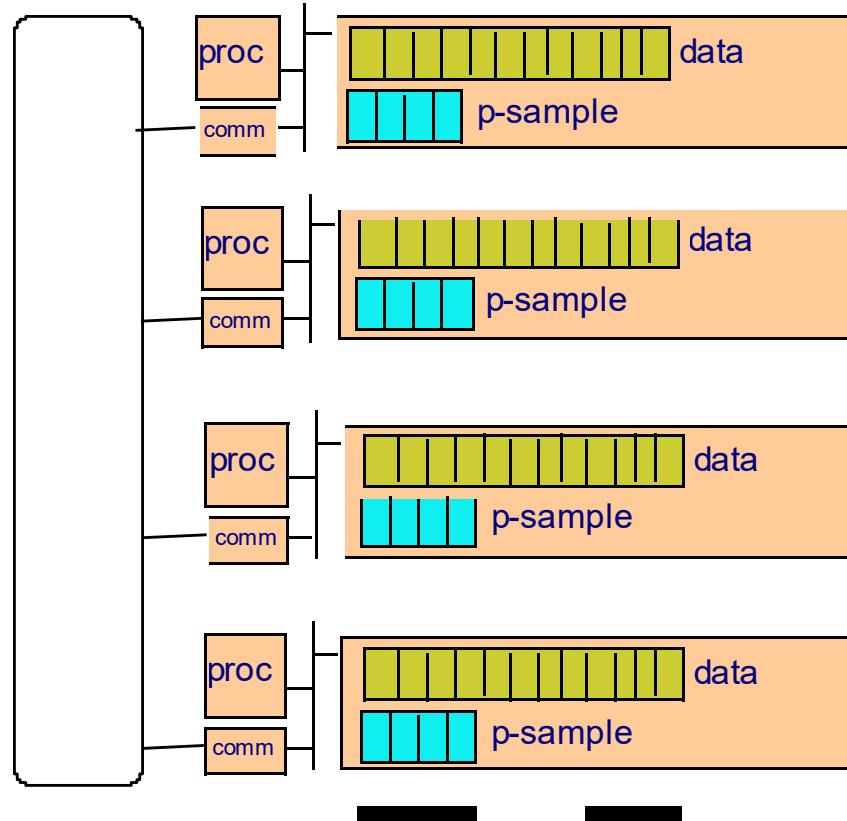


$$t_p = O(n/p + (n/p) \log (n/p)) = O((n/p) \log (n/p)) \Rightarrow S=O(p)$$

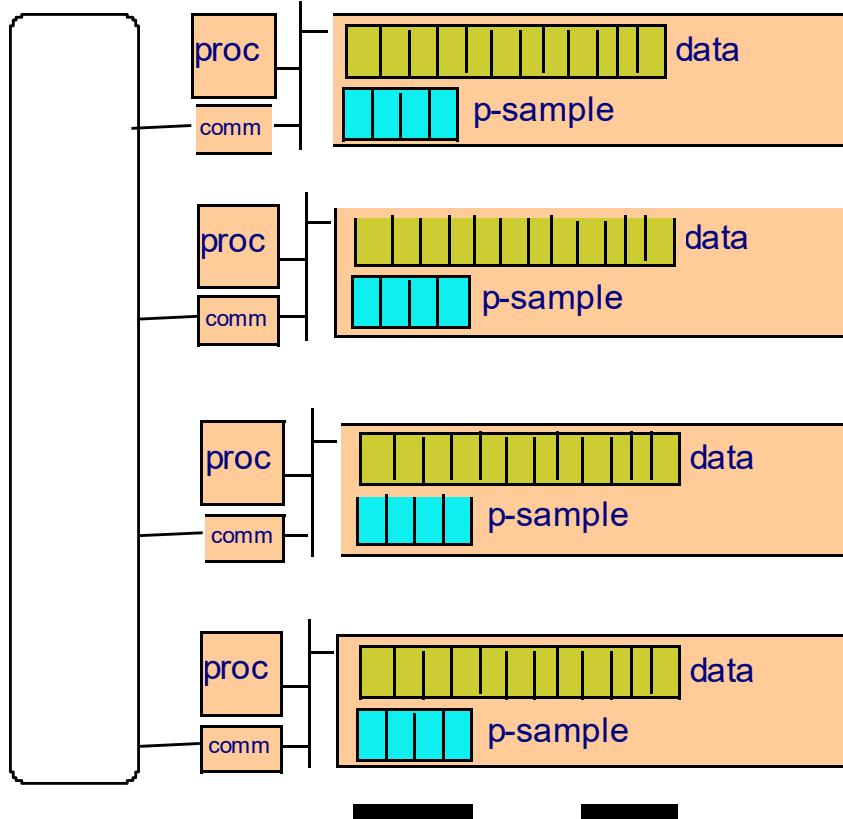
Introduces new message-passing operation - all-to-all broadcast.



Det. Sample Sort

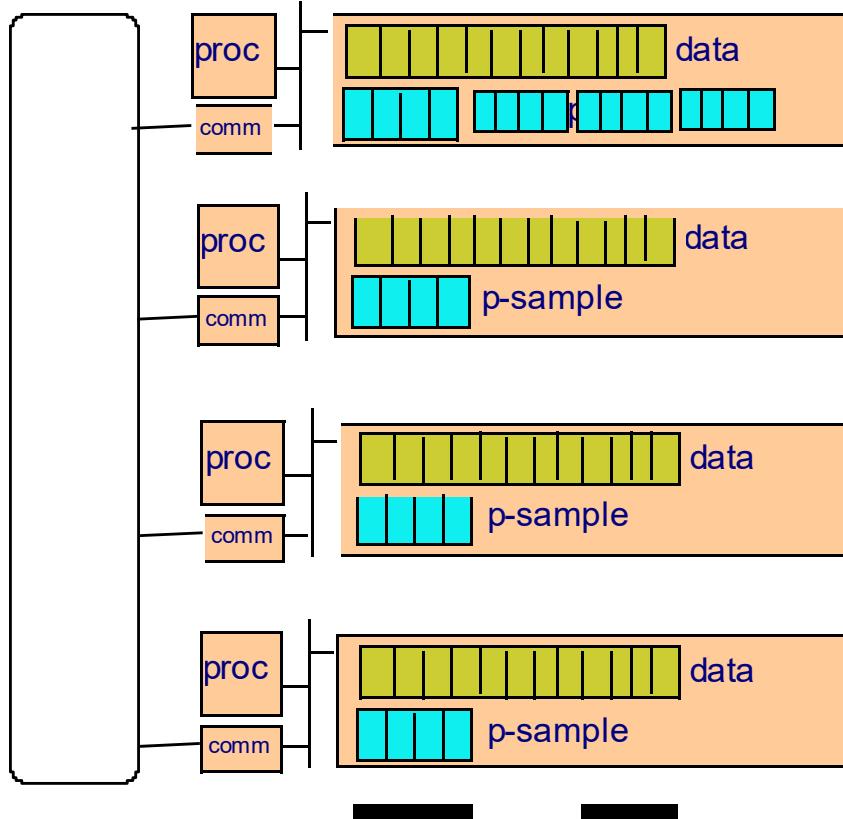


Det. Sample Sort



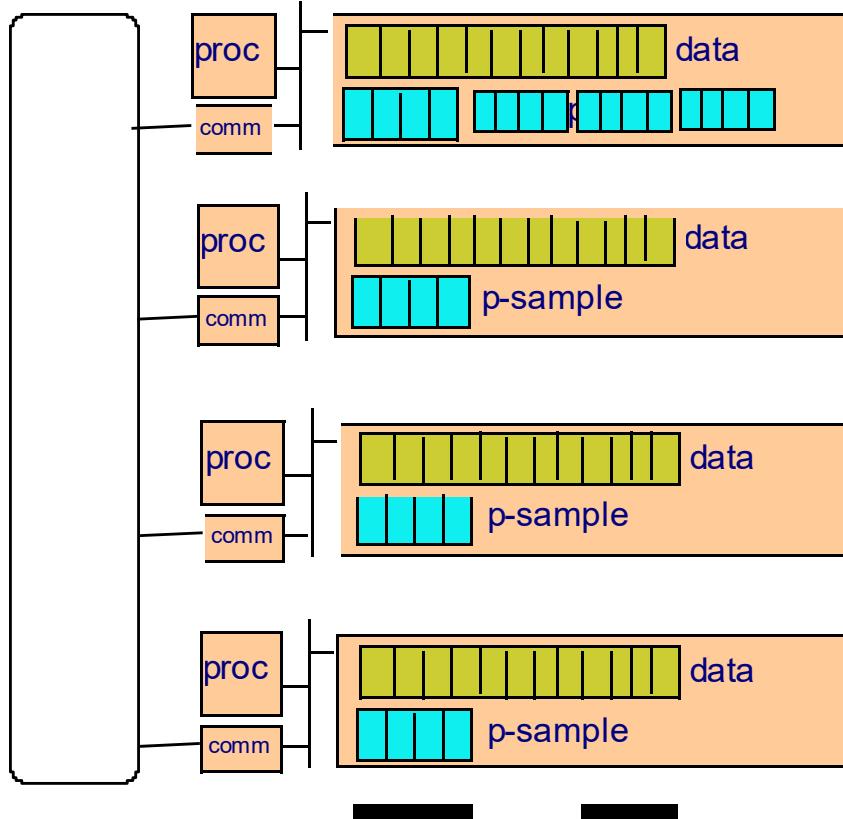
- sort locally and create p-sample

Det. Sample Sort



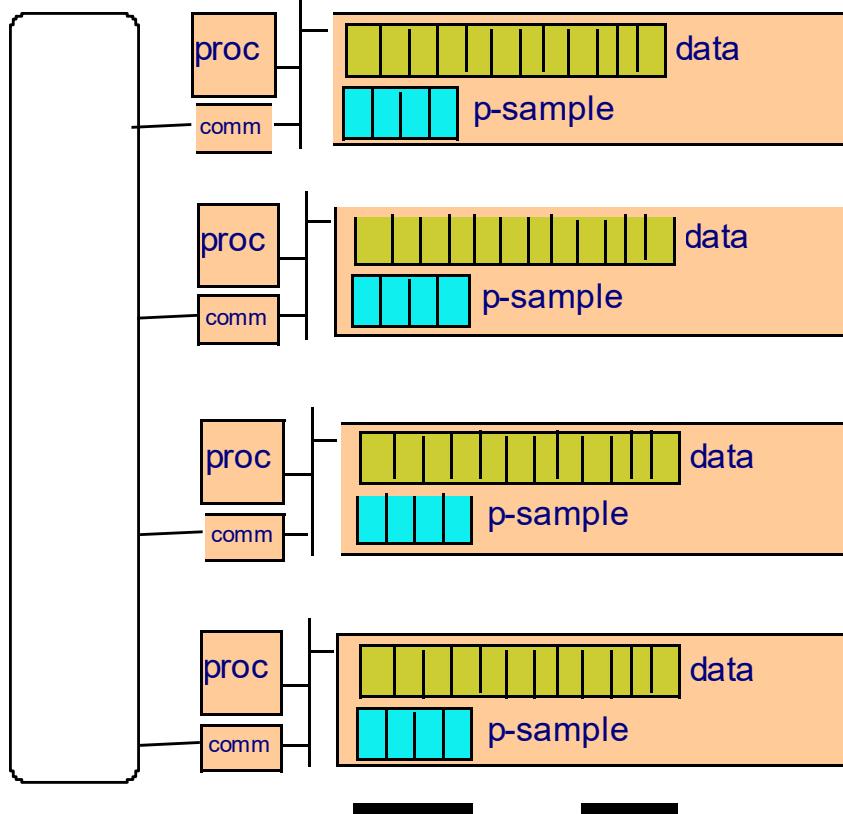
- send all p-samples to processor 1

Det. Sample Sort



- proc.1: sort all received samples and compute global p-sample

Det. Sample Sort



- broadcast global p-sample
- bucket locally according to global p-sample
- send bucket i to proc. i
- resort locally

Numerical Integration

- Sometimes simple partitioning will not give the optimal solution, especially if the amount of work in each part is difficult to estimate.
 - Bucket sort, for example, is only effective when each region has approximately the same # of numbers.

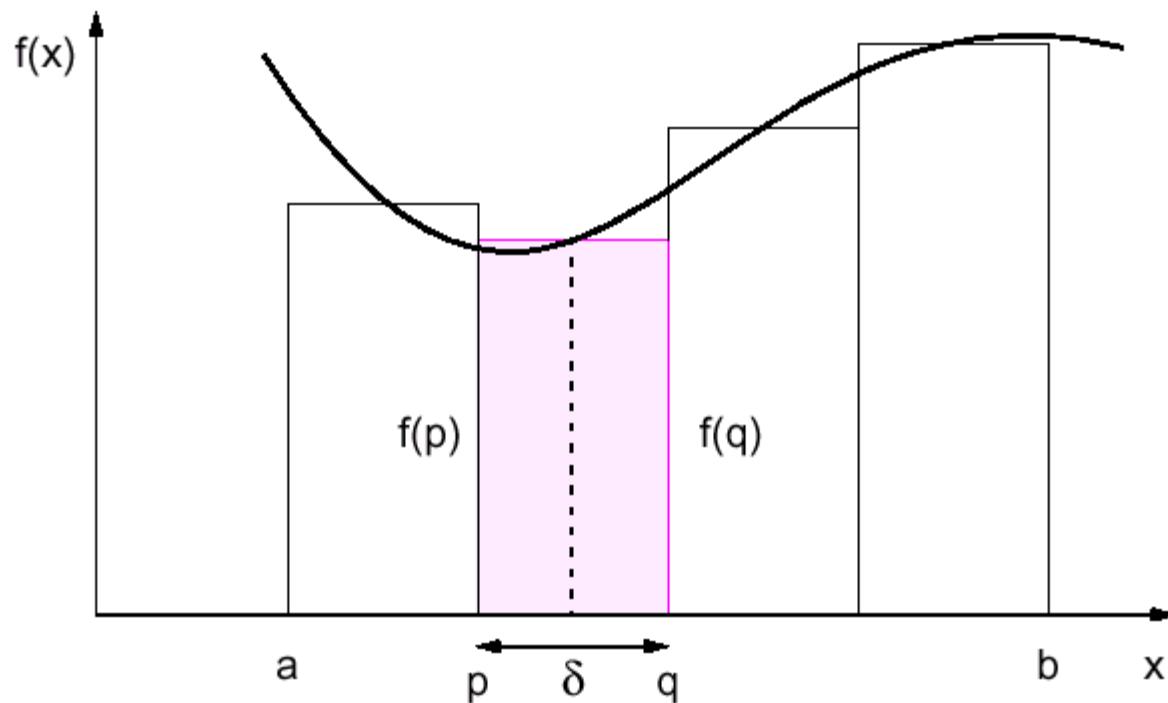
A general divide-and-conquer technique divides the region continually into parts and lets some optimization function decide when certain regions are sufficiently divided.

$$I = \int_a^b f(x) dx$$

To integrate this function (i.e., to compute the area under the curve,) we can divide the area into separate parts, each of which can be calculated by a separate process.

Numerical integration using rectangles.

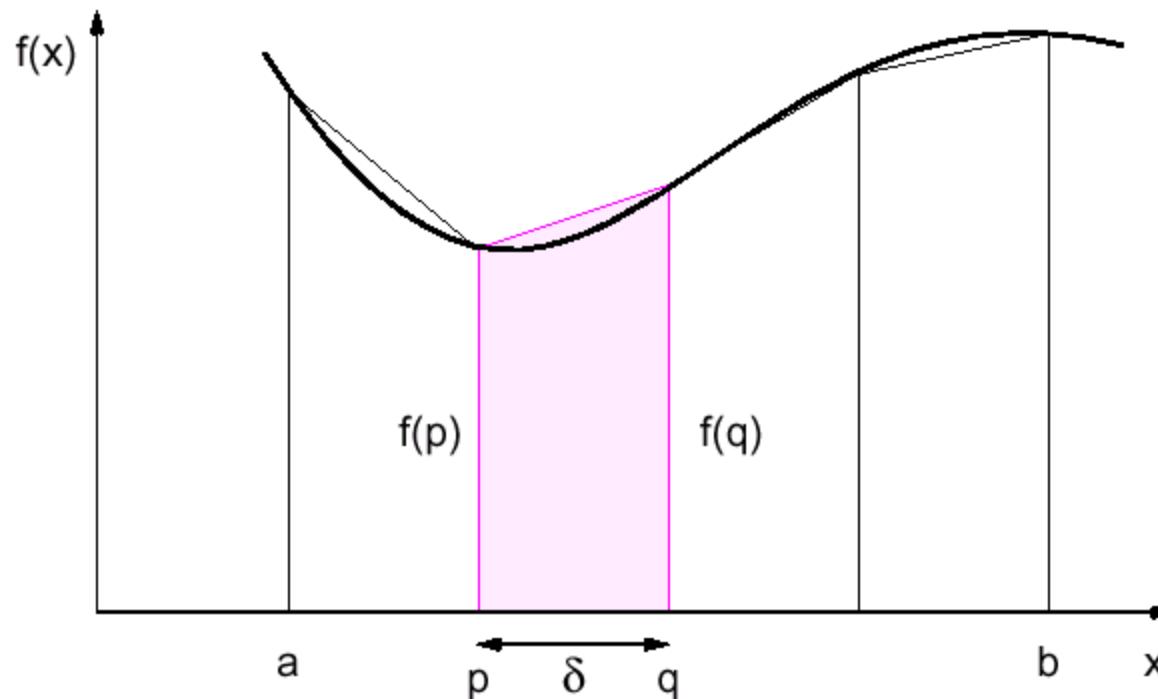
Each region calculated using an approximation given by rectangles:
static assignment of processors
to segments of $[a,b]$



$$\text{area} = \delta (f(p)+f(q))/2$$

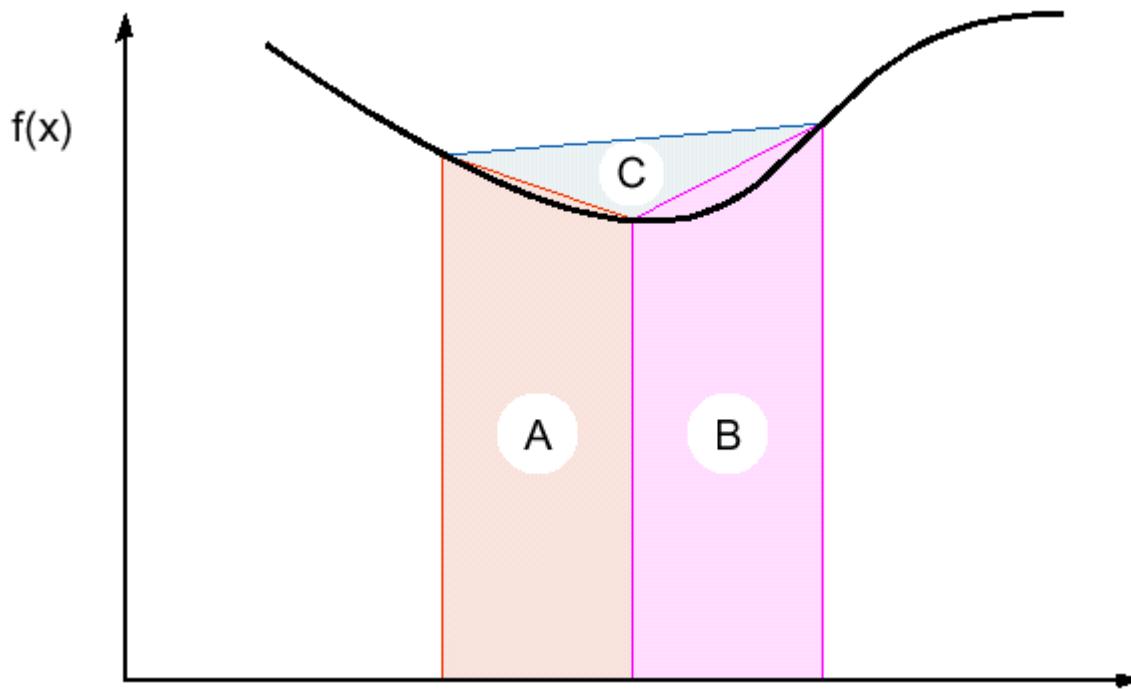
Problem: precision depends on curve's shape

Numerical integration using trapezoidal method



May not be better!

Adaptive Quadrature



Terminate when C is sufficiently small

Problem: different parts of the curve need different resolution

Quadrature Methods:

approximate numerical methods for computing a definite integral using a linear combination of values

- Rectangular regions
- Aligned rectangular regions whose middle points intersect with the function
- Trapezoidal regions

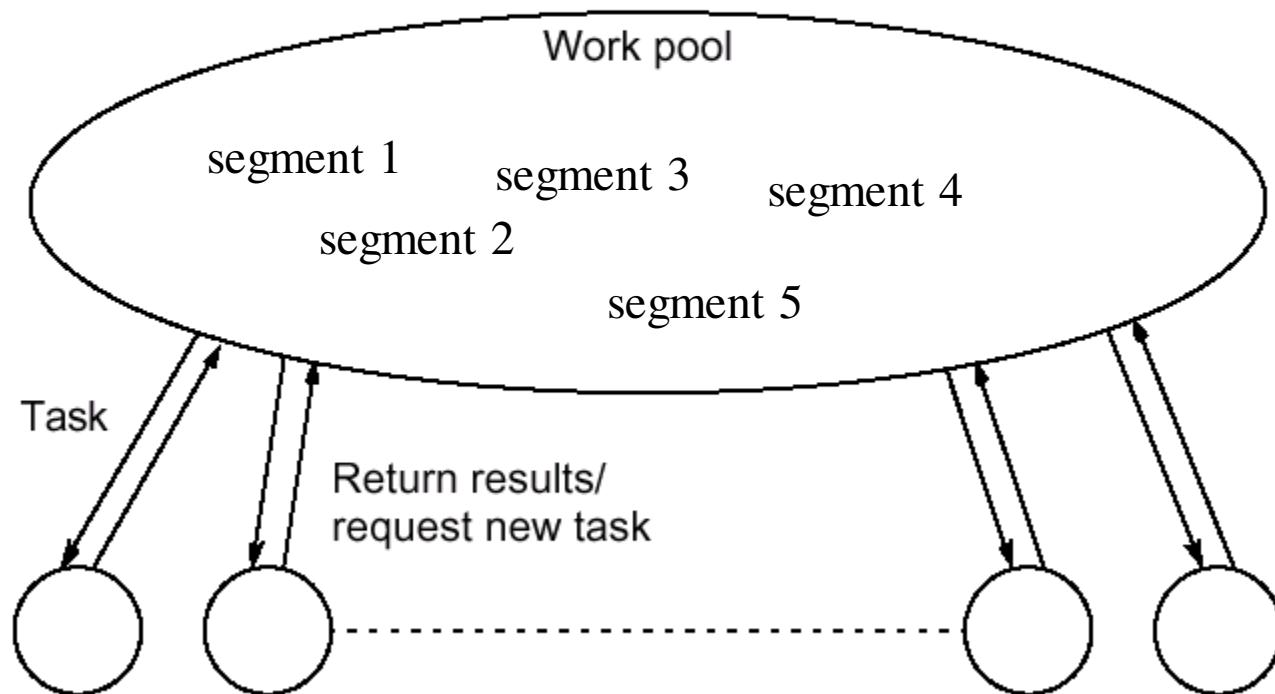
- Static Assignment.
 - Let's consider trapezoidal method.

Process Pi:

```
if (i == master) {  
    printf("Enter number of intervals: ");  
    scanf("%d", &n);  
}  
bcast(&n, P_group);  
region = (b-a)/p;  
start = a + region * i;  
end = start + region;  
d = (b-a)/n;  
area = 0.0;  
for (x=start; x<end; x += d)  
    area += 0.5 * ( f(x) + f(x+d) ) * d;  
reduce_add(&integral, &area, P_group);
```

Dynamic Task Assignment

Work Pool/Processor Farms



Gravitational N-Body Problem

Finding positions and movements of bodies in space subject to gravitational forces from other bodies, using Newtonian laws of physics.

Gravitational N-Body Problem Equations

Gravitational force between two bodies of masses m_a and m_b is:

$$F = \frac{Gm_a m_b}{r^2}$$

G is the gravitational constant and r the distance between the bodies. Subject to forces, body accelerates according to Newton's 2nd law:

$$F = ma$$

m is mass of the body, F is force it experiences, and a the resultant acceleration.

Details

Let the time interval be Δt . For a body of mass m , the force is:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

New velocity is:

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where v^{t+1} is the velocity at time $t + 1$ and v^t is the velocity at time t .

Over time interval Δt , position changes by

$$x^{t+1} - x^t = v\Delta t$$

where x^t is its position at time t .

Once bodies move to new positions, forces change. Computation has to be repeated.

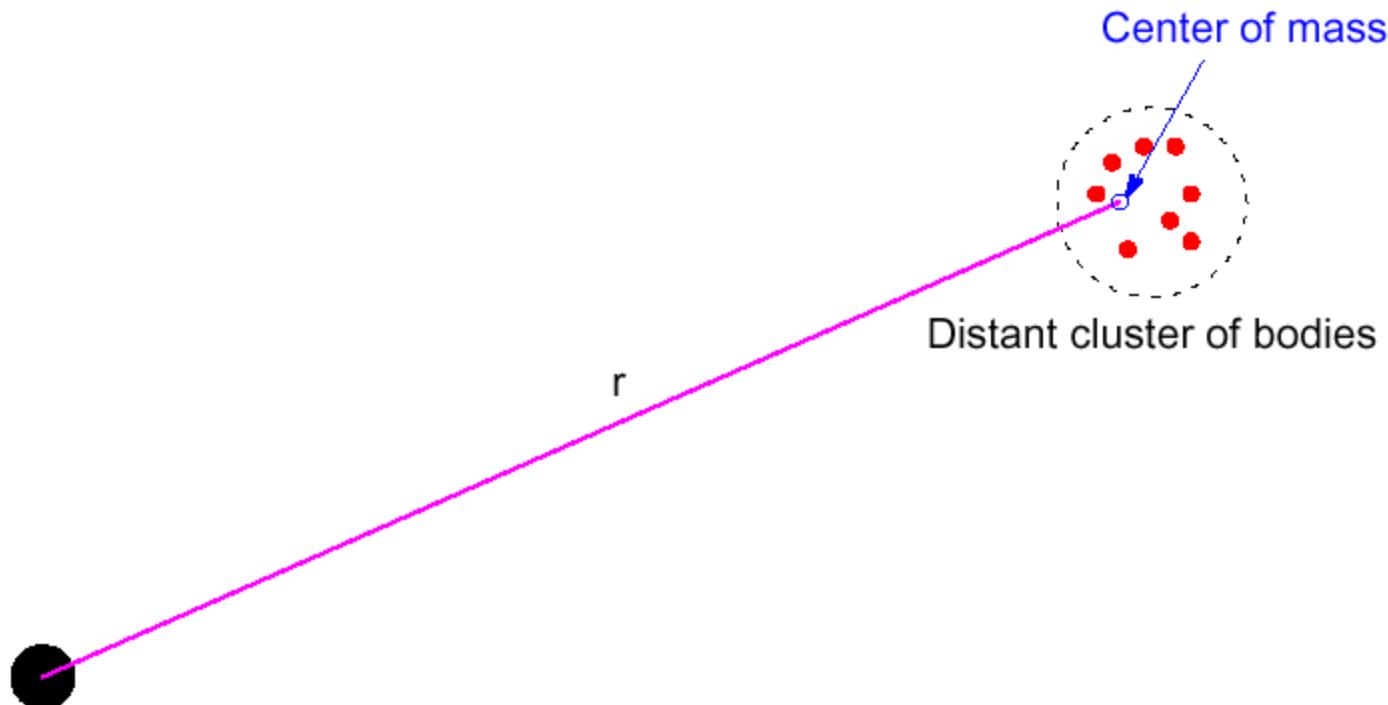
Sequential Code

Overall gravitational N -body computation can be described by:

```
for (t = 0; t < tmax; t++)      /* for each time period */
    for (i = 0; i < N; i++) {    /* for each body */
        F = Force_routine(i);    /* compute force on ith body */
        v[i]_new = v[i] + F * dt / m; /* compute new velocity */
        x[i]_new = x[i] + v[i]_new * dt; /* and new position */
    }
    for (i = 0; i < nmax; i++) {    /* for each body */
        x[i] = x[i]_new;           /* update velocity & position*/
        v[i] = v[i]_new;
    }
```

$$t_s = O(n^2) \text{ per time step}$$

Time complexity can be reduced using observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster sited at the center of mass of the cluster:

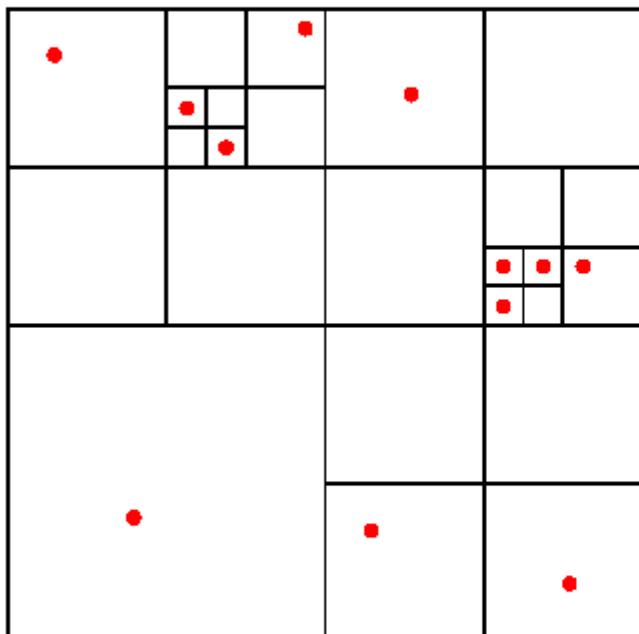


Barnes-Hut Algorithm

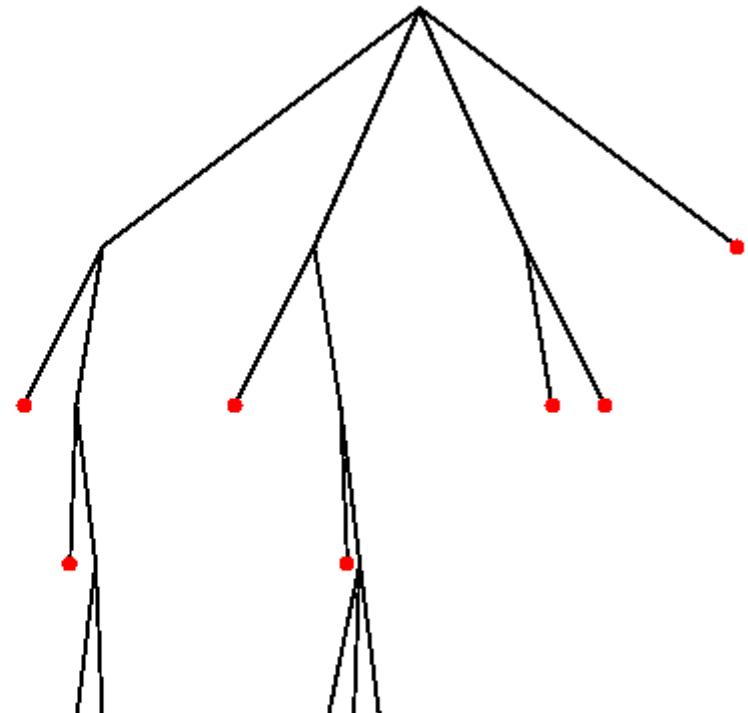
Start with whole space in which one cube contains the bodies (or particles).

- First, this cube is divided into eight subcubes.
- If a subcube contains no particles, the subcube is deleted from further consideration.
- If a subcube contains one body, this subcube retained
- If a subcube contains more than one body, it is recursively divided until every subcube contains one body.

Recursive division of two-dimensional space



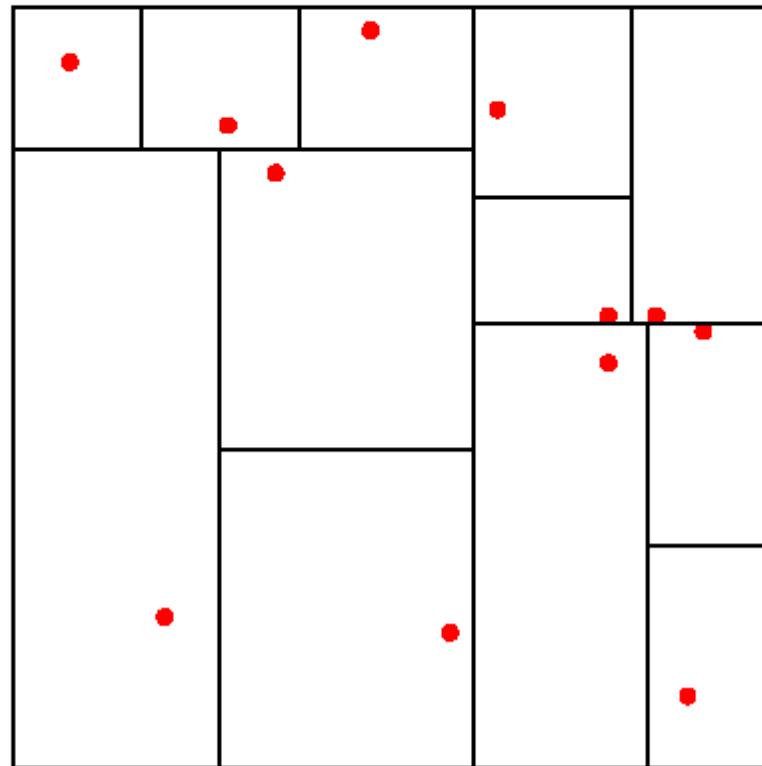
Particles



Partial quadtree

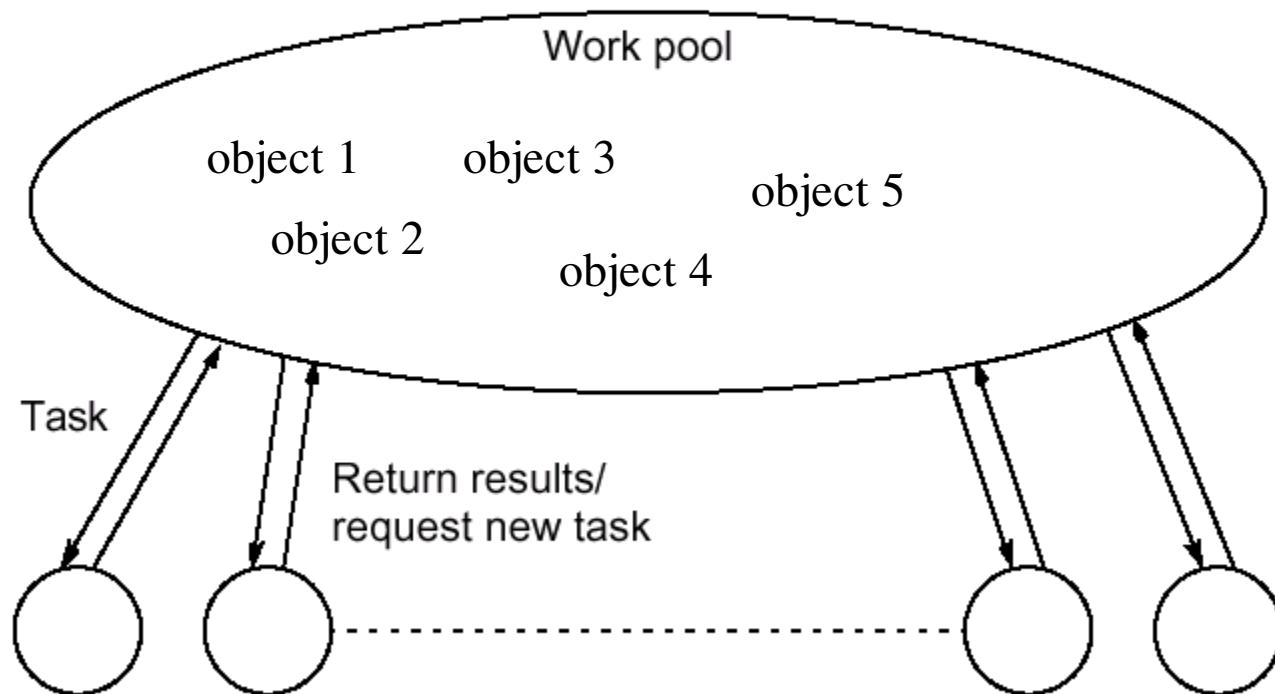
Orthogonal Recursive Bisection

(For 2-dimensional area) First, a vertical line found that divides area into two areas each with equal number of bodies. For each area, a horizontal line found that divides it into two areas each with equal number of bodies. Repeated as required.



Dynamic Task Assignment

Work Pool/Processor Farms



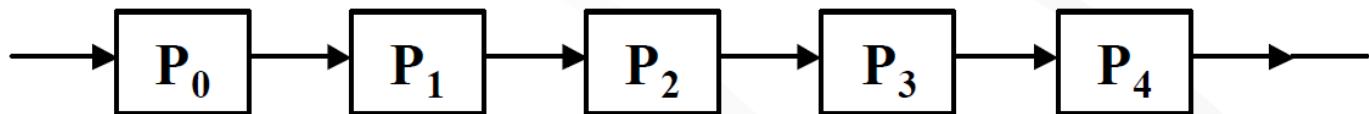
Pipelined Computations

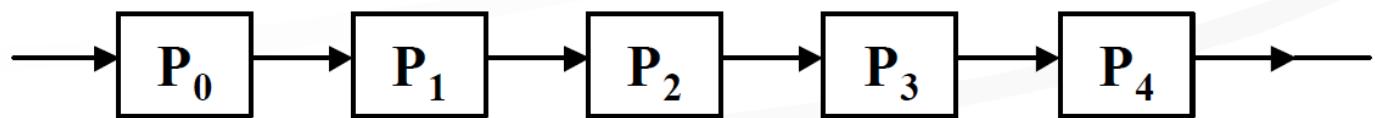
Pipelined computation

Applicable to a wide range of problems that are partially sequential in nature; i.e., sequence of steps must be undertaken.

In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other - a form of “functional parallelism”

Pipeline stage: each pipeline process (that contributes to the overall problem and pass on info needed for subsequent stages.)





```
for (i=0; i<n; i++)
```

```
    sum += a[i];
```

unfold

```
sum = sum + a[0];
```

```
sum = sum + a[1];
```

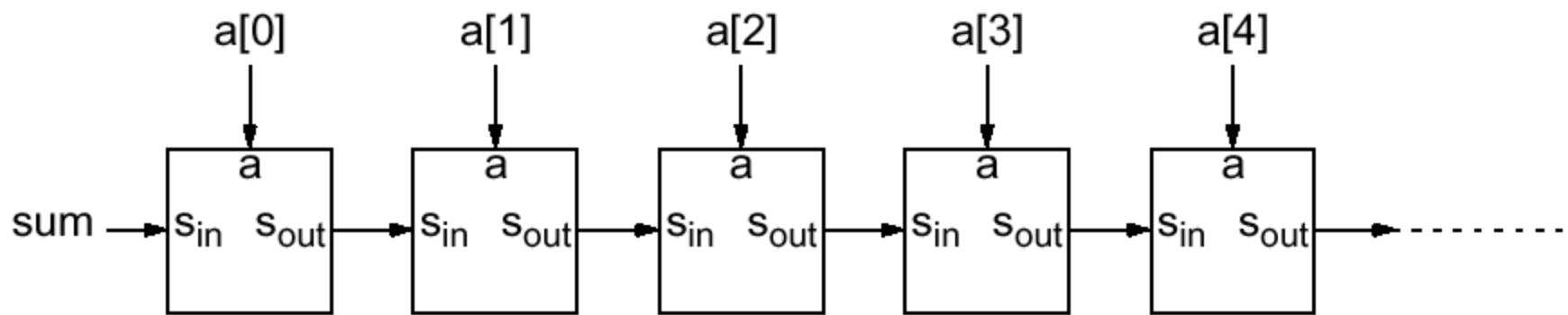
```
sum = sum + a[2];
```

```
sum = sum + a[3];
```

```
sum = sum + a[4];
```

Stage i performs: `sout = sin + a[i];`

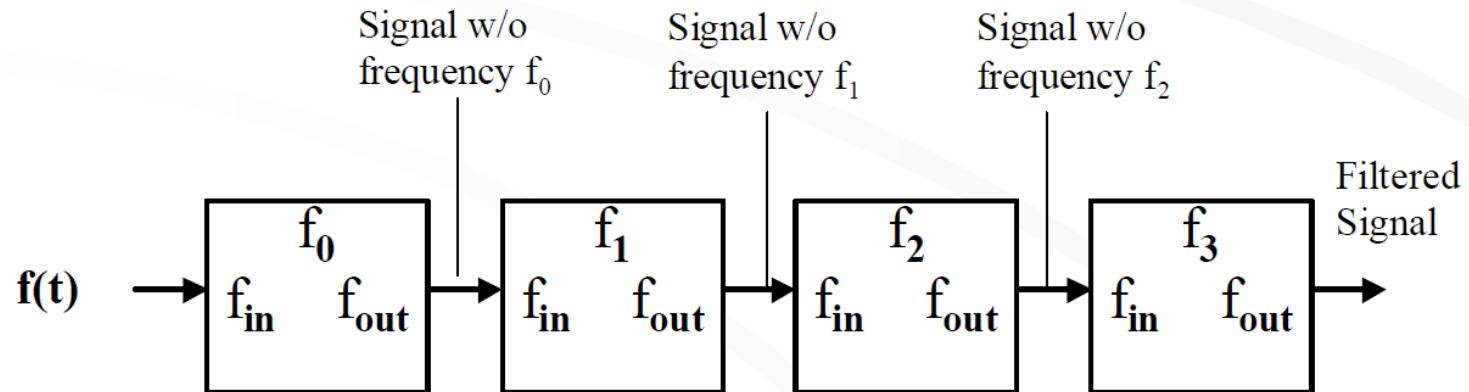
Pipeline for an unfolded loop



Example: Frequency Filter

- Problem is divided into a series of functions
- The objective is to remove specific frequencies (say f_1, f_2, f_3, \dots) from a (digitized) signal, $f(t)$.
- The signal enter the pipeline from the left
- Each stage is responsible for removing one of the frequencies.

Frequency-amplitude histogram in professional sound systems is a similar example.



Applicable Types of Computations

1. If more than one instance of the complete problem is to be executed.
 - ◆ internal hardware design, simulation exercises
2. If a series of data items must be processed, each requiring multiple operations.
 - ◆ array operations

With p processes and n data items,
overall execution time is $(p-1) + n$
3. If info to start the next process can be passed forward before the process has completed all its internal operations.

If number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor.

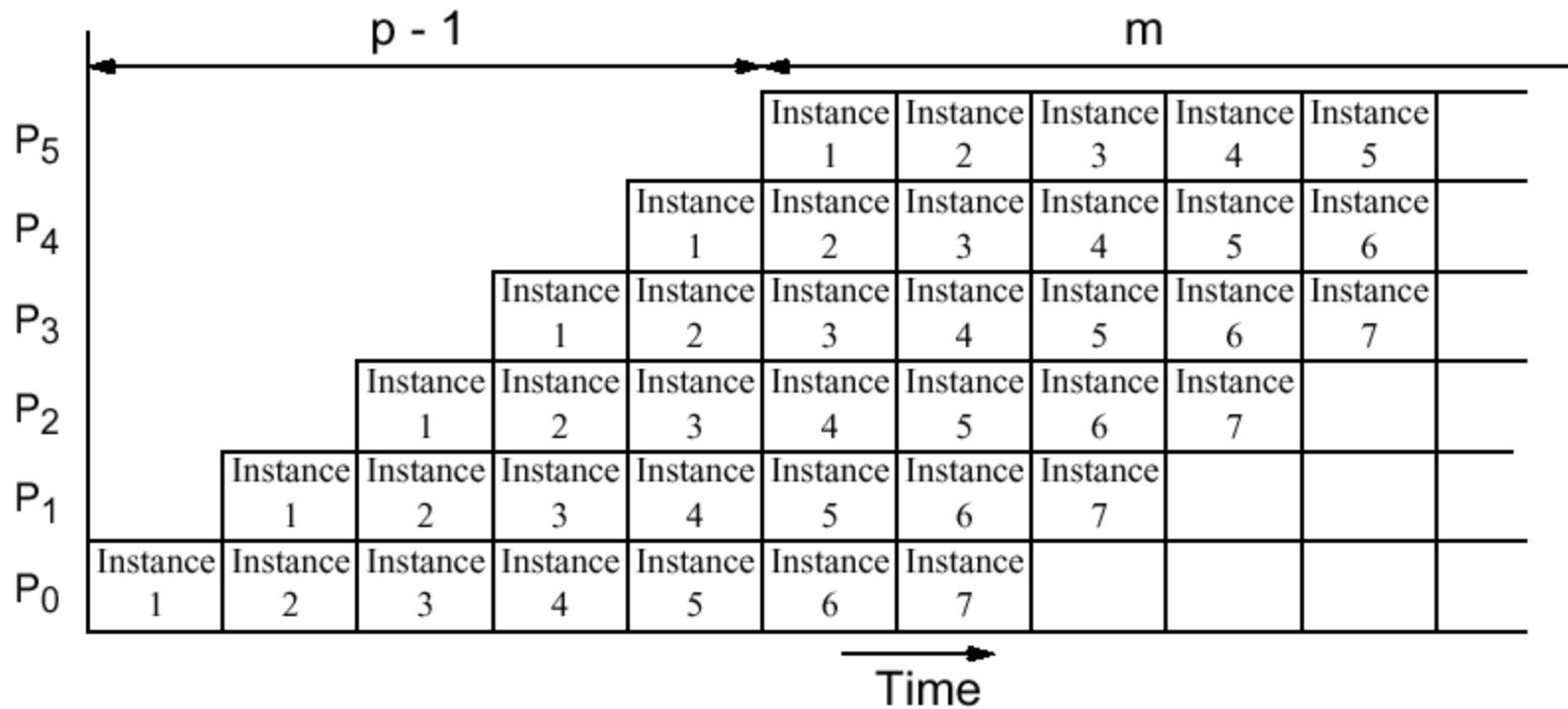
Where pipelining can be used to good effect

Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations:

1. If more than one instance of the complete problem is to be executed “data stream”
2. If a series of data items must be processed, each requiring multiple operations “vector”
3. If information to start the next process can be passed forward before the process has completed all its internal operations
 “tunnel”

“Type 1” Pipeline Space-Time Diagram

“data stream”

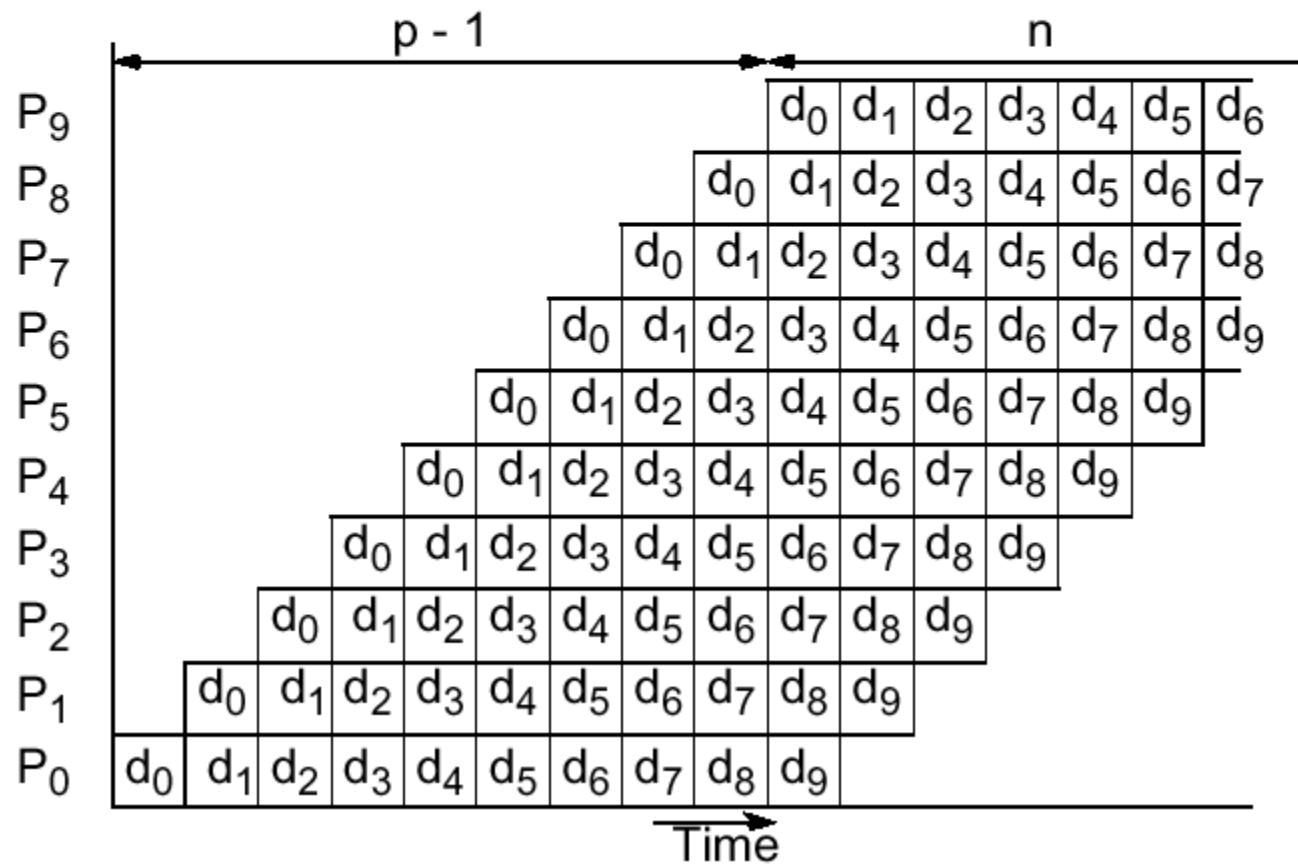
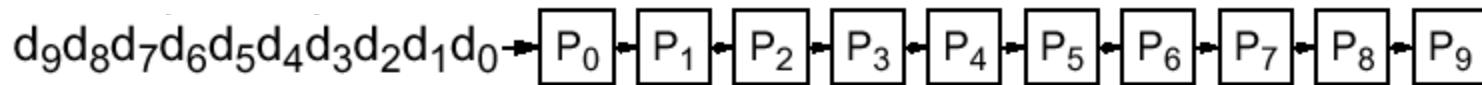


Execution time: $p-1+m$ (For p processors and m instances)

per instance: $1+(p-1)/m \rightarrow 1$ for large m

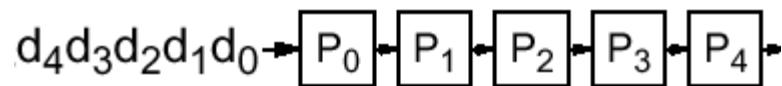
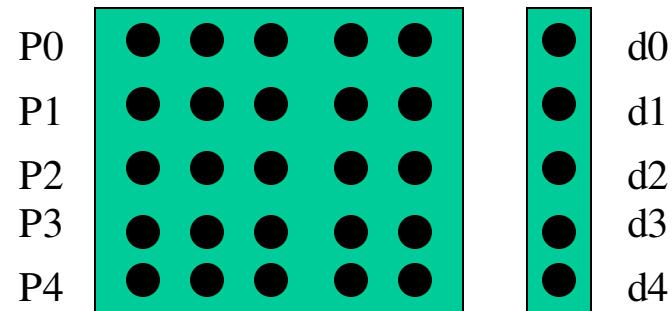
“Type 2” Pipeline Space-Time Diagram

“vector”



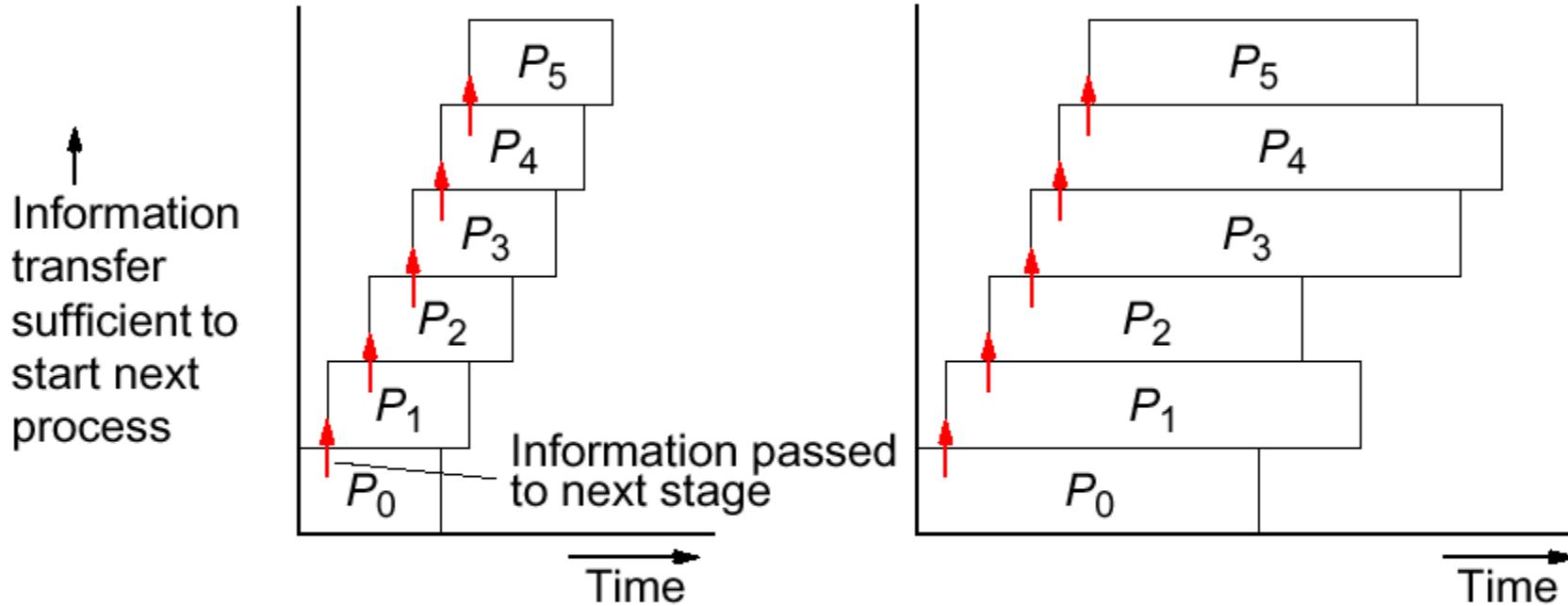
Time: $p-1+n$

Example: Matrix-Vector Product



“Type 3” Pipeline Space-Time Diagram

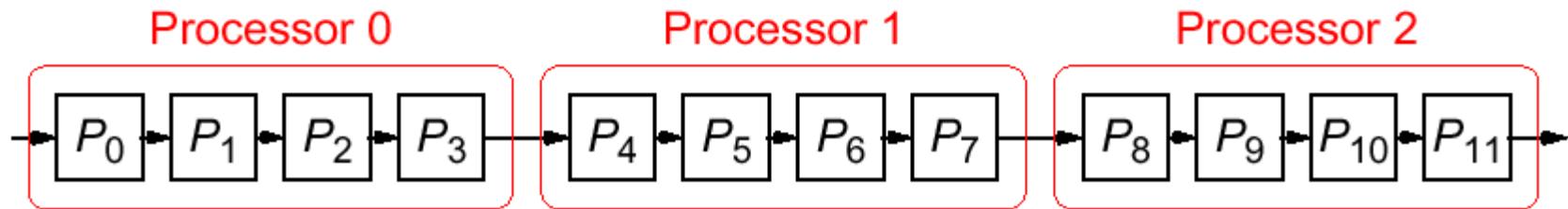
“tunnel”



Pipeline processing where information passes to next stage before end of process.

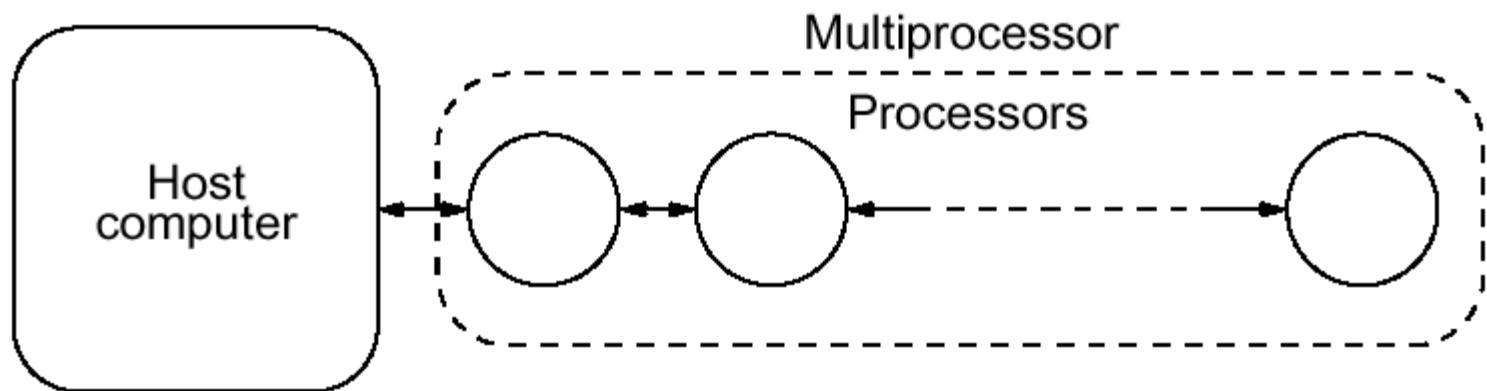
Example: Graphics Processor

If the number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor:



Computing Platform for Pipelined Applications

Multiprocessor system with a line configuration.



Type 1

Pipeline cycle: each time period in space-time diagram.

Note the staircase effect at the beginning.

With p processes constituting the pipeline and
 m instances of the problem:

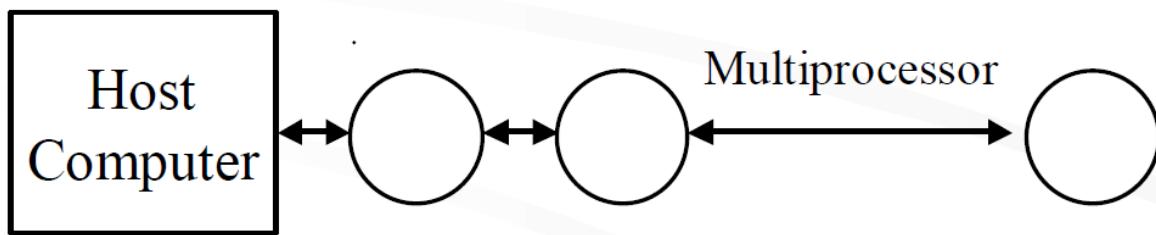
$m+p-1$ cycles are required to execute all m instances.

Average number of cycles: $(m+p-1)/m$

One instance of the problem is completed in each pipeline cycle after the $n-1$ cycles (pipeline latency)

Computing Platform for Pipelined Applications

- Ability to send messages between adjacent processes in the pipeline
 - direct comm. links -- ring or line is ideal structure.
- Networked workstations on Ethernet may not really be a suitable platform for the pipelined programs.



locally blocking `send()` operations can be used.

Adding Numbers (Type 1)

- Basic code for process P_i :

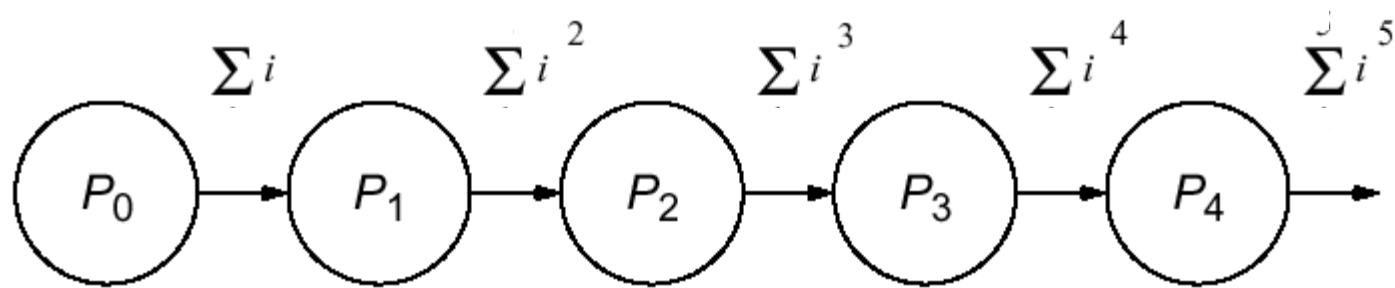
```
if (process > 0) {  
    recv(&partial_sum, Pi-1);  
    partial_sum += number;  
}  
if (process < n - 1)  
    send(&partial_sum, Pi+1);
```

Two ways of distributing data: (Fig.5.11 & 12)

1. Data being entered into the first process, the result is returned through the last process.
2. Data is fed into each process at the times that they are needed by the processes.

Pipeline Program Examples

Adding Numbers



Type 1 pipeline computation

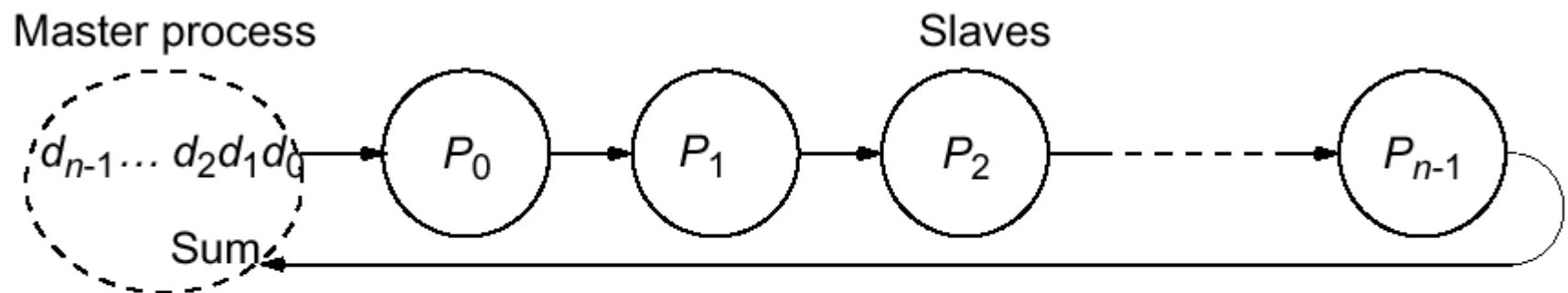


Fig. 5.11

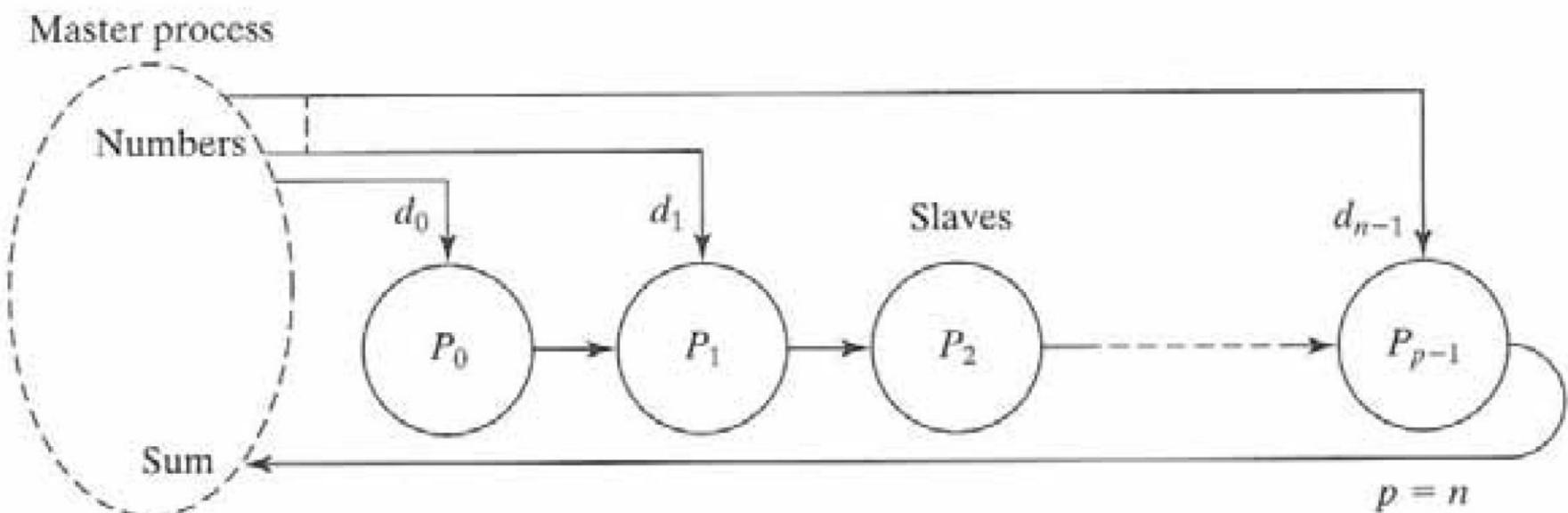


Fig. 5.12

Analysis

- Assume that each process performs similar actions in each pipeline cycle.
- We will work on comp. and comm. required in one cycle.

$$t_{\text{total}} = (\text{time for one pipeline cycle}) (\# \text{ of cycles})$$

$$t_{\text{total}} = (t_{\text{comp}} + t_{\text{comm}}) (m+p-1)$$

m: number of instances of the problem

p: pipeline stages (processes)

The average time for a computation: $t_a = t_{\text{total}} / m$

Single Instance of Problem: (Figure 5.11)

Single number is being added in each stage: $n==p$

$$t_{\text{comp}} = 1$$

$$t_{\text{comm}} = 2 (t_{\text{startup}} + t_{\text{data}})$$

When only one set of numbers, $m=1$:

$$t_{\text{total}} = (2 (t_{\text{startup}} + t_{\text{data}}) + 1) n$$

Multiple Instances of Problem:

m groups of n numbers are being added:

$$t_{\text{total}} = (2 (t_{\text{startup}} + t_{\text{data}}) + 1) (m+n-1)$$

For large m , $t_a = t_{\text{total}}/m = 2 (t_{\text{startup}} + t_{\text{data}}) + 1$

Data Partitioning with Multiple Instance of Problem:

Each stage will process a group of d numbers

$$p = n/d$$

$$t_{\text{comp}} = d$$

$$t_{\text{comm}} = 2 (t_{\text{startup}} + t_{\text{data}})$$

$$t_{\text{total}} = (2 (t_{\text{startup}} + t_{\text{data}}) + d) (m + n/d - 1)$$

As d increases,

- the impact of the comm. on the overall time diminishes
- parallelism decreases and increases the execution time.

Sorting Numbers (Type 2)

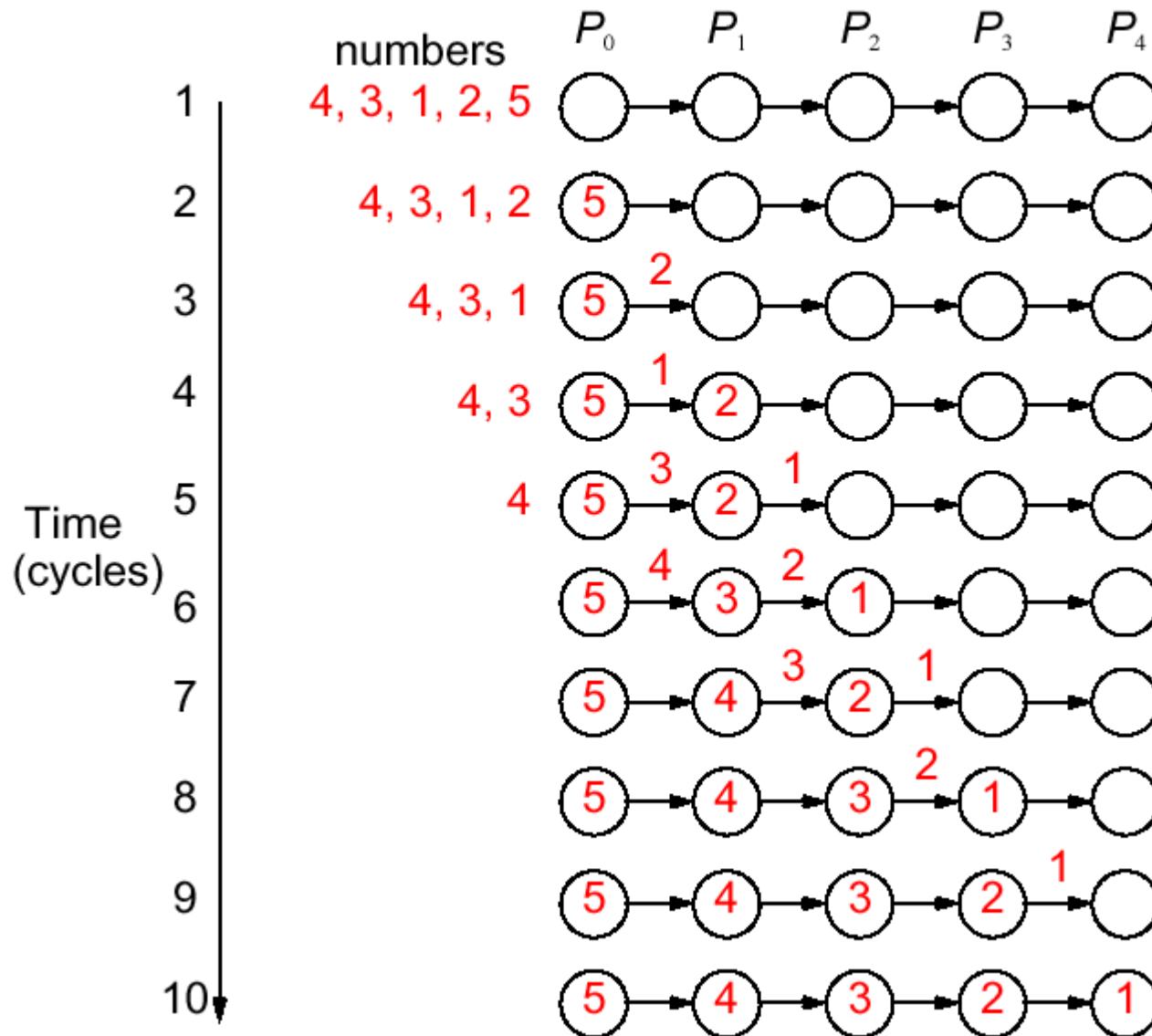
reorder a set of numbers in increasing (or decreasing) numeric order.

A pipeline solution: (Parallel Insertion Sort)

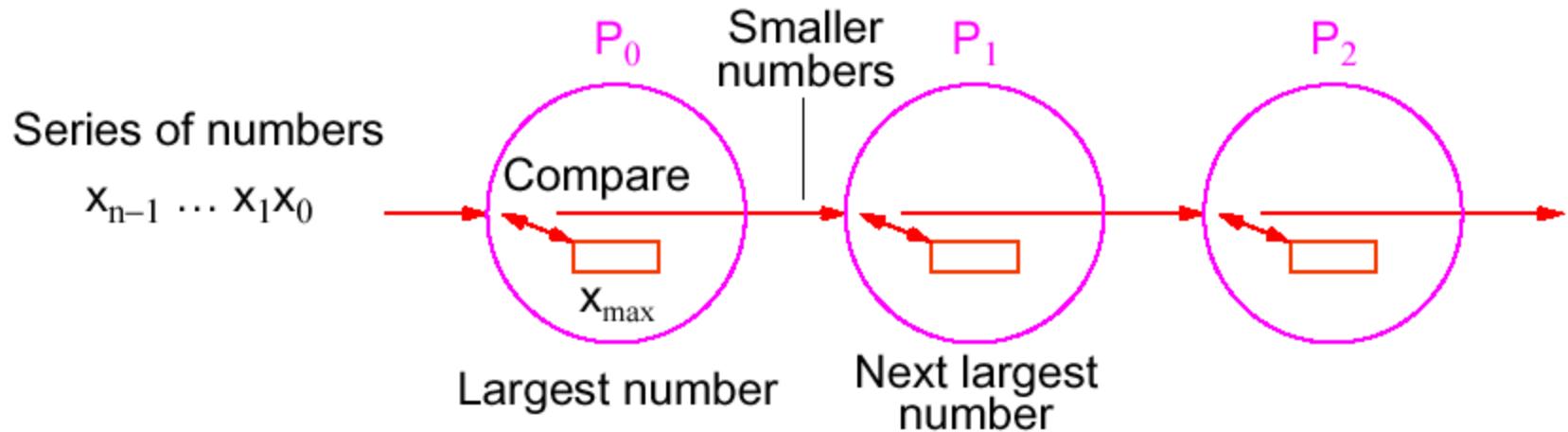
- The first process, P_0 , accepts the series of numbers one at a time.
- Stores the largest number so far received, and pass onward all numbers smaller than the stored number.

Sorting Numbers

A parallel version of *insertion sort*.



Pipeline for sorting using insertion sort



Type 2 pipeline computation

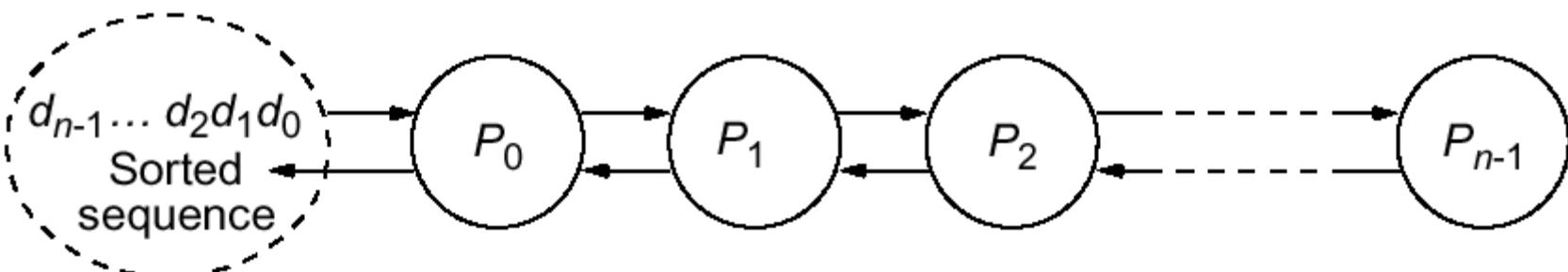
The basic algorithm for process P_i is

```
recv(&number, Pi-1);  
if (number > x) {  
    send(&x, Pi+1);  
    x = number;  
} else send(&number, Pi+1);
```

With n numbers, how many the i th process is to accept is known; it is given by $n - i$. How many to pass onward is also known; it is given by $n - i - 1$ since one of the numbers received is not passed onward. Hence, a simple loop could be used.

Insertion sort with results returned to the master process using a bidirectional line configuration

Master process



Parallel Code for P_i.

```
recv (&number, Pi-1);
if (number > x) {
    send (&x, Pi+1);
    x = number;
} else
    send (&number, Pi+1);
```

With n numbers:

```
right_procno = n - i - 1;
recv (&x, Pi-1);
for (j=0; j<right_procno; j++) {
    recv (&number, Pi-1);
    if (&number > x) {
        send (&x, Pi+1);
        x = number;
    }
    else
        send (&number, Pi+1);
```

A message-passing program using an SPMD or a master-slave approach is straightforward

- especially since each pipeline process executes essentially the same code.

Results can be extracted from the pipeline using

- either the ring configuration of Fig.5.11 or
- the bidirectional line configuration of Fig.5.15.

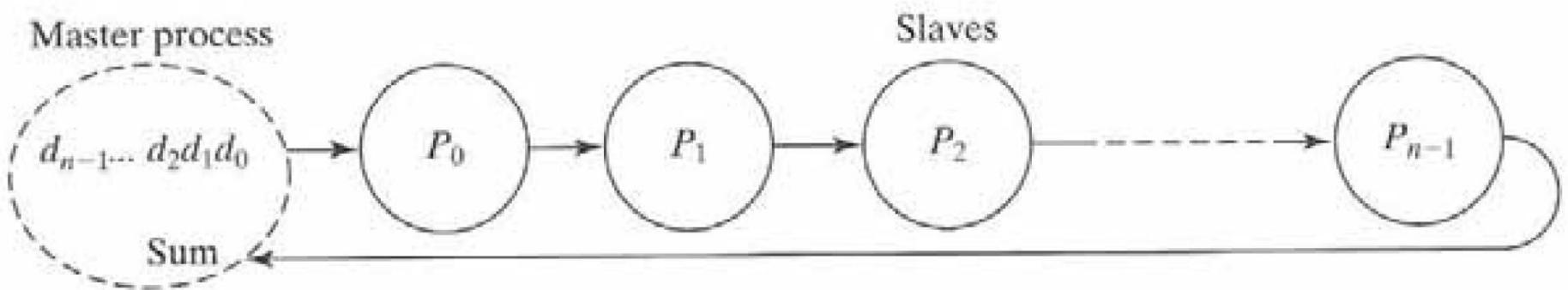


Fig. 5.11

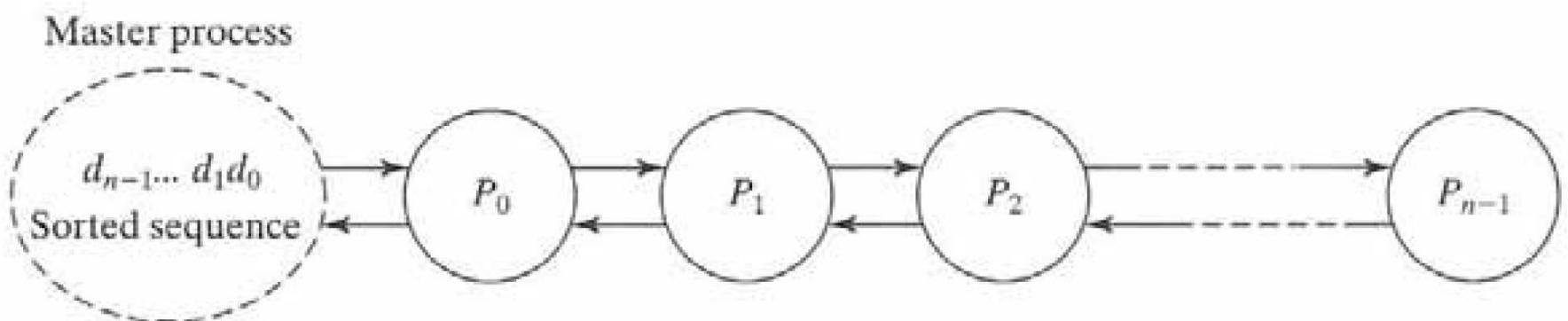


Fig. 5.15

```
right_procno = n - i -1;
recv (&x, Pi-1);
for (j=0; j<right_procno; j++) {
    recv (&number, Pi-1);
    if (&number > x) {
        send (&x, Pi+1);
        x = number;
    }
    else send (&number, Pi+1);
}
send (&number, Pi-1);
for (j=0; j<right_procno; j++) {
    recv (&number, Pi+1);
    send (&x, Pi-1);
}
```

Analysis.

Assuming that compare-and-exchange is one computational step:

$$t_s = (n-1) + (n-2) + \dots + 2 + 1 =$$

The parallel impl. has $n+n-1 = 2n-1$ pipeline cycles during the sorting if there are n pipeline processes and n numbers to sort.

$$t_{comp} = 1$$

$$t_{comm} = 2 (t_{startup} + t_{data})$$

If the results are returned by comm. to the left through the master, $3n-1$ pipeline cycles are needed.

Prime Number Generation

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

To find the primes up to n , it is only necessary to look at numbers up to \sqrt{n} .

Sequential Code.

```
for (i=2; i<n; i++)
    prime[i] = 1;
        for (i=2; i<= sqrt_n; i++)
            if (prime[i] == 1)
                for (j=i+i; j<n; j+=i)
                    prime[j] = 0;
```

There are $[n/2-1]$ multiples of 2, $[n/3-1]$ multiples of 3, so on.

Hence, total sequential time is ... with $O(n^2)$ complexity

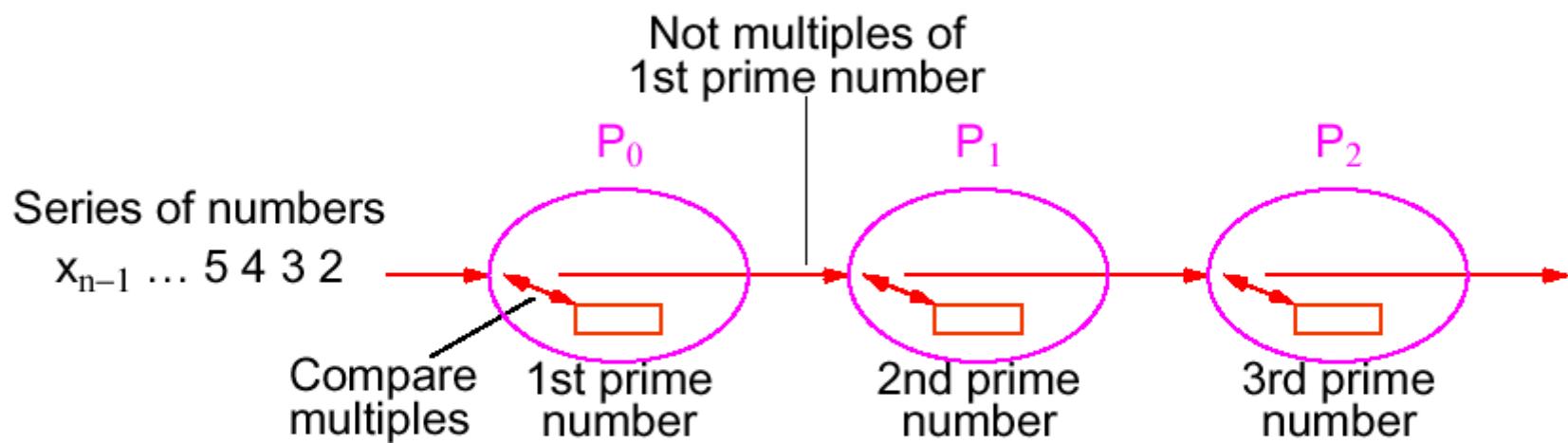
The code for a process, P_i , could be based upon

```
recv(&x, Pi-1);
/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0) send(&number, iP1);
```

Each process will not receive the same amount of numbers and the amount is not known beforehand. Use a “terminator” message, which is sent at the end of the sequence:

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
    recv(&number, Pi-1);
    if (number == terminator) break;
    if (number % x) != 0) send(&number, iP1);
}
```

Pipeline for Prime Number Generation



Type 2 pipeline computation

Solving a System of Linear Equations (Type 3)

A process can continue with useful work after passing on information.

Objective: solving a system of linear equations of the *upper-triangular* form.

Method: simple repeated “*back*” substitution.

The i th process ($0 < i < n$) receives the values x_0, x_1, \dots, x_{i-1} and computes x_i from the equation:

$$x_i = \frac{(b_i - \sum_{j=0}^{i-1} a_{i,j}x_j)}{a_{i,i}}$$

Solving a System of Linear Equations

Upper-triangular form

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

where the a 's and b 's are constants and the x 's are unknowns to be found.

Back Substitution

First, the unknown x_0 is found from the last equation; i.e.,

$$x_0 = \frac{b_0}{a_{0,0}}$$

Value obtained for x_0 substituted into next equation to obtain x_1 ; i.e.,

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

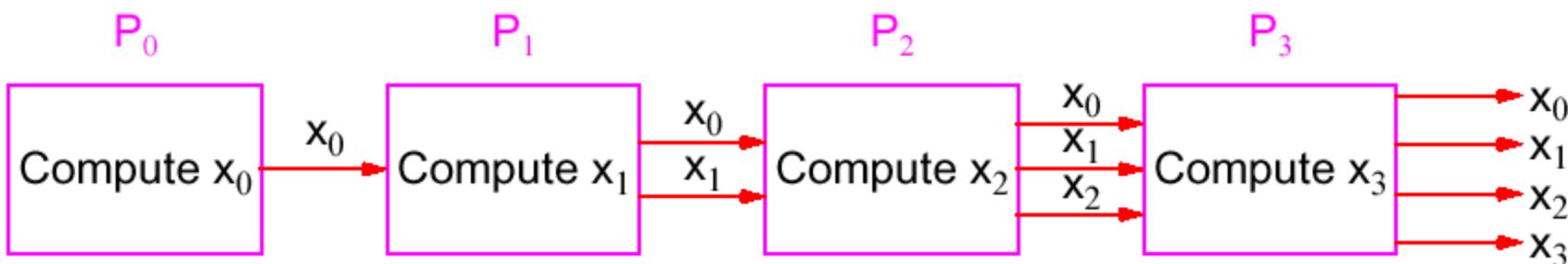
Values obtained for x_1 and x_0 substituted into next equation to obtain x_2 :

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

and so on until all the unknowns are found.

Pipeline Solution

First pipeline stage computes x_0 and passes x_0 onto the second stage, which computes x_1 from x_0 and passes both x_0 and x_1 onto the next stage, which computes x_2 from x_0 and x_1 , and so on.



Type 3 pipeline computation

The i th process ($0 < i < n$) receives the values $x_0, x_1, x_2, \dots, x_{i-1}$ and computes x_i from the equation:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

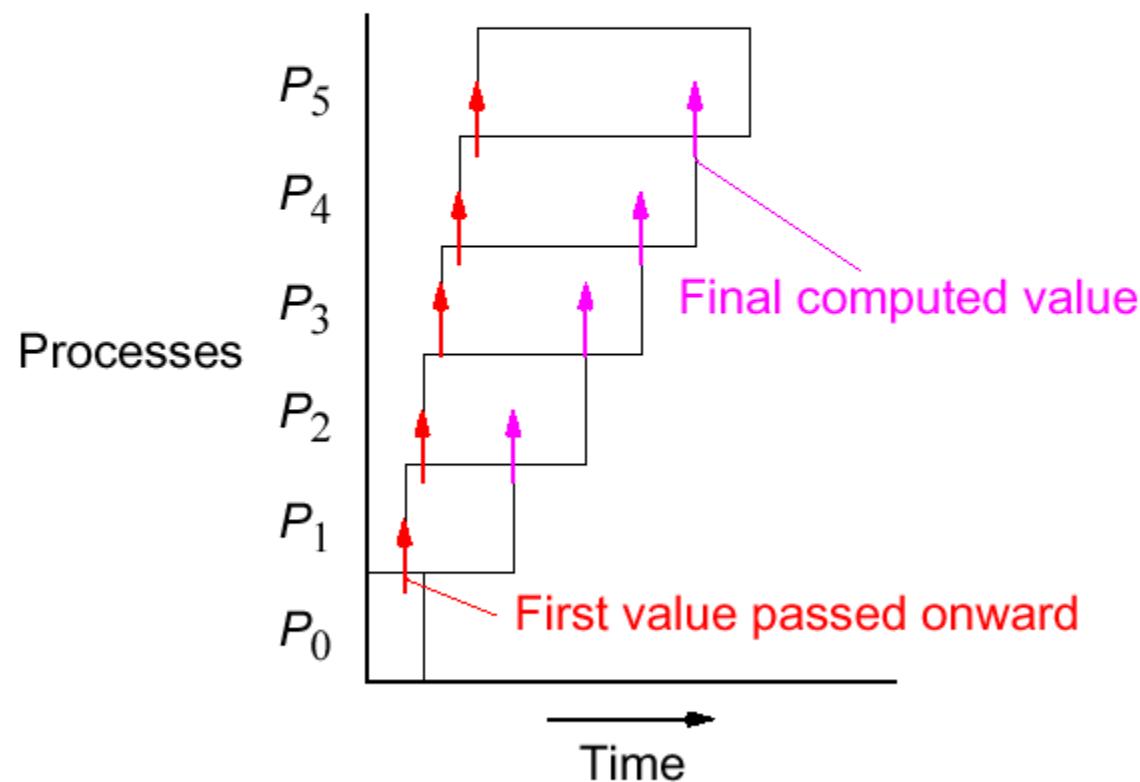
Sequential Code.

```
x[0] = b[0]/a[0][0];
for (i=1; i<n; i++) {
    sum = 0;
    for (j=0; j<i; j++) {
        sum += a[i][j] * x[j];
    }
    x[i] = (b[i] - sum) / a[i][i];
}
```

Parallel Code.

```
for (j=0; j<i; j++) {
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
}
sum = 0;
for (j=0; j<i; j++) {
    sum += a[i][j] * x[j];
}
x[i] = (b[i] - sum) / a[i][i];
send(&x[i], Pi+1);
```

Pipeline processing using back substitution



Analysis.

Computational effort at each pipeline stage is not the same.

P_0 performs one divide and one `send()`.

P_i performs

- $i \text{ recv}()$
- $i \text{ send}()$
- $i \text{ multiply/add}$
- $1 \text{ divide/subtract}$
- $1 \text{ final send}()$

a total of $n-1$ comm.
and $2n-1$ comp. steps

Figure 5.20: a perfect sync. of the sends and recvs

$T_p = \text{time of final process} + n-1 \text{ sends and } 1 \text{ divide.}$

$O(n)$ - parallel time complexity

$O(n^2)$ - serial time complexity

Actual speedup is $0.37n$ (Lester, 1993)

↓

	P_0	P_1	P_2	P_3	P_4
divide					
$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$				
end	$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$			
	multiply/add	$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$		
	divide/subtract	multiply/add	$\text{send}(x_0) \Rightarrow$	$\text{recv}(x_0)$	$\text{recv}(x_0)$
	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$	multiply/add	$\text{send}(x_0) \Rightarrow$	$\text{send}(x_1) \Rightarrow$
	end	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$	multiply/add	multiply/add
		multiply/add	$\text{send}(x_1) \Rightarrow$	$\text{recv}(x_1)$	$\text{recv}(x_1)$
		divide/subtract	multiply/add	$\text{send}(x_1) \Rightarrow$	$\text{send}(x_1) \Rightarrow$
		$\text{send}(x_2) \Rightarrow$	$\text{recv}(x_1)$	multiply/add	multiply/add
		end	$\text{send}(x_2) \Rightarrow$	$\text{recv}(x_2)$	$\text{recv}(x_2)$
			multiply/add	$\text{send}(x_2) \Rightarrow$	$\text{send}(x_2) \Rightarrow$
			divide/subtract	multiply/add	multiply/add
			$\text{send}(x_3) \Rightarrow$	$\text{recv}(x_3)$	$\text{recv}(x_3)$
			end	$\text{send}(x_3) \Rightarrow$	$\text{send}(x_3) \Rightarrow$
				multiply/add	multiply/add
				divide/subtract	divide/subtract
				$\text{send}(x_4) \Rightarrow$	$\text{send}(x_4) \Rightarrow$
				end	end

- Nonblocking sends and blocking receives are used in implementation.

Chapter 6

Synchronous Computations

Synchronous Computations

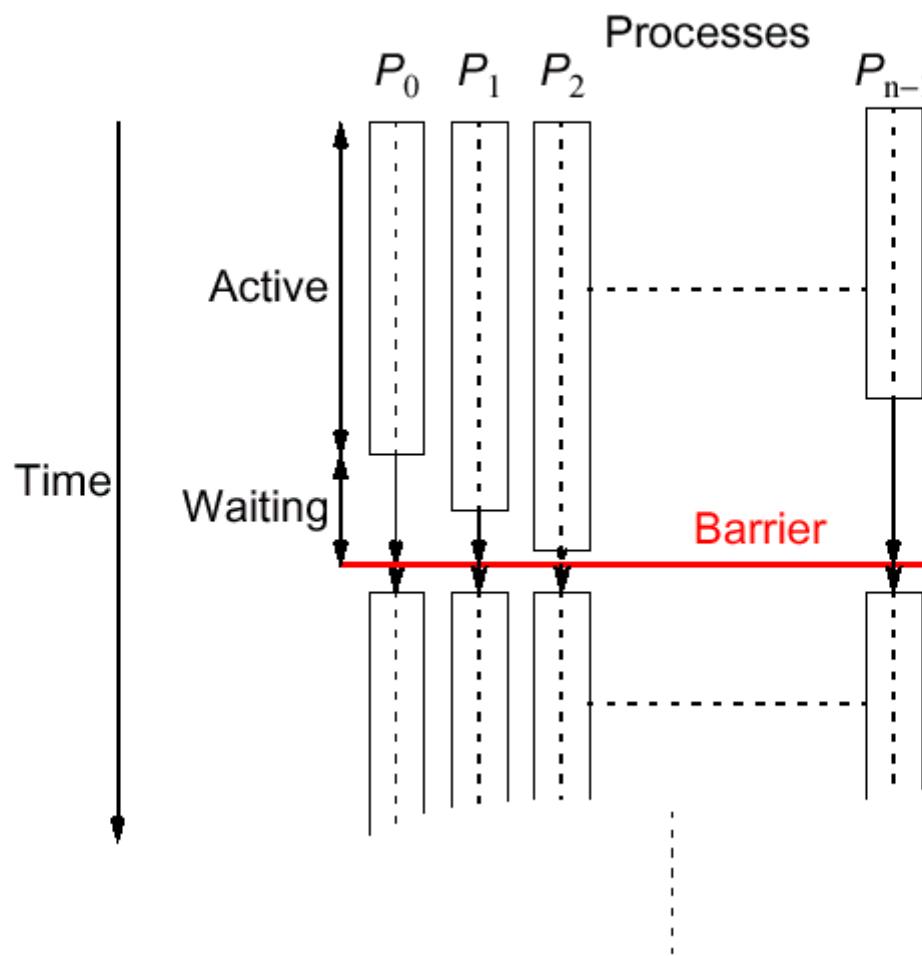
In a (fully) synchronous application, all the processes synchronized at regular points.

Barrier

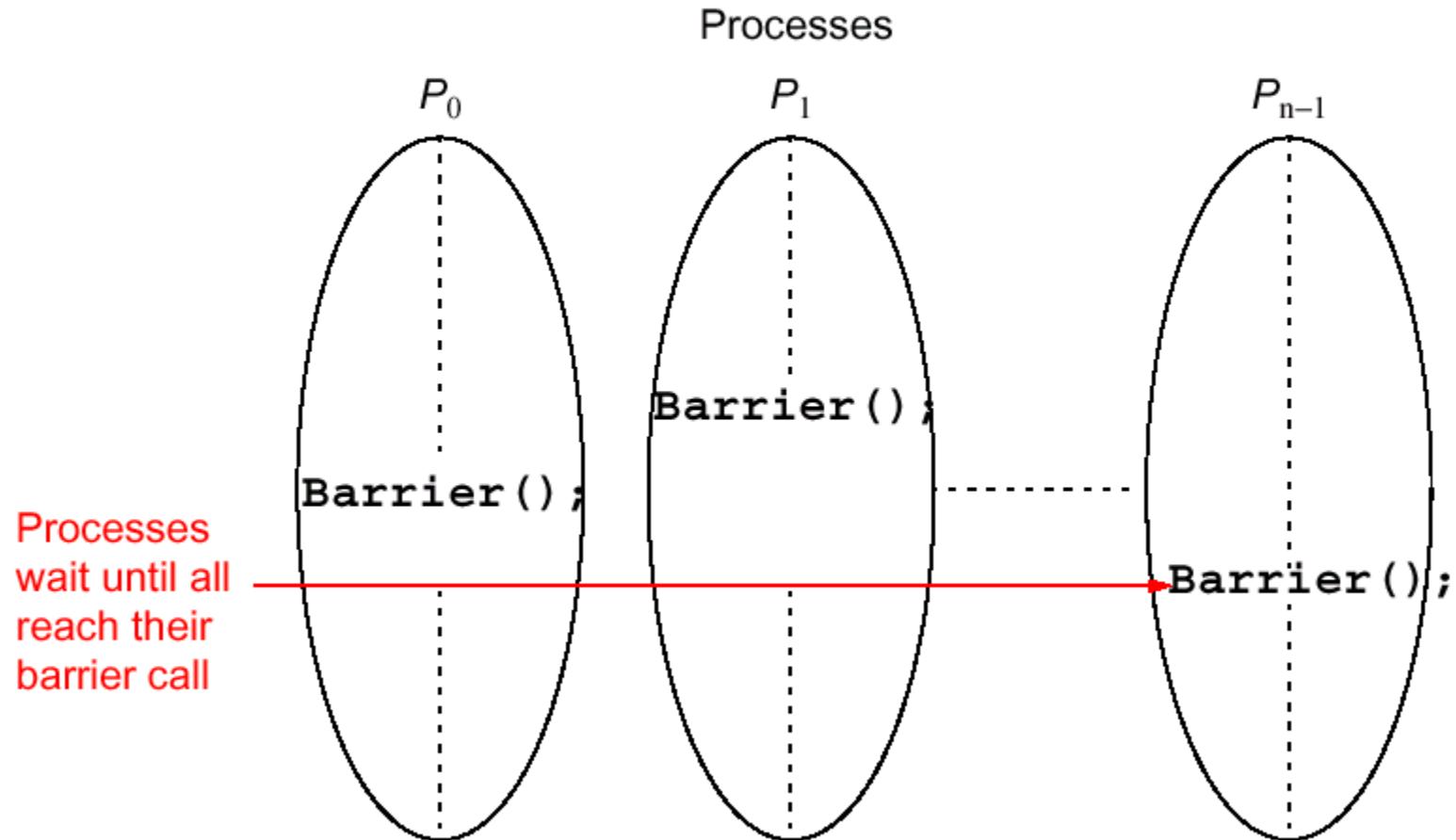
A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait.

All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).

Processes reaching barrier at different times



In message-passing systems, barriers provided with library routines:



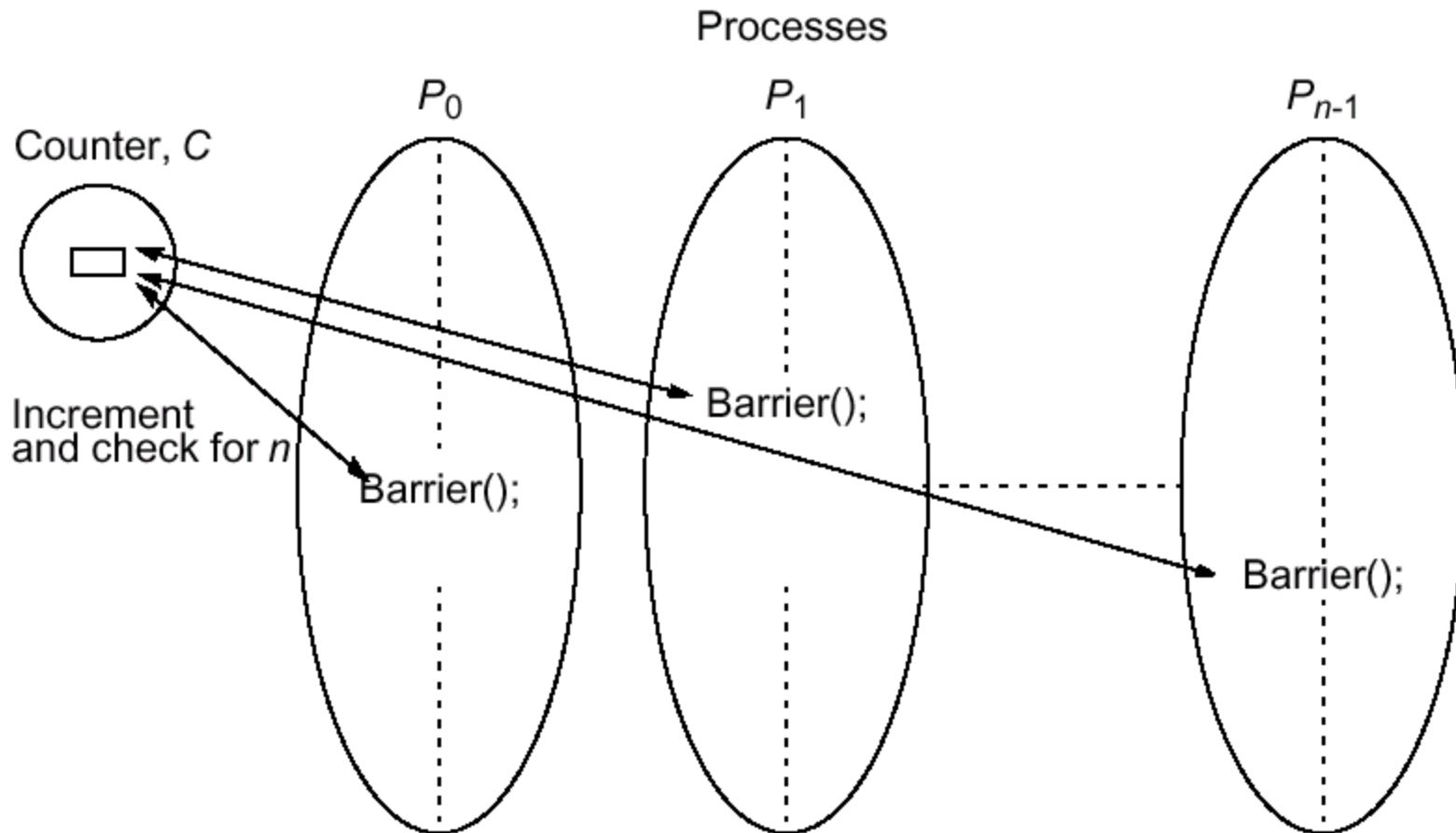
MPI

MPI_Barrier()

Barrier with a named communicator being the only parameter.
called by each process in the group, blocking until all members of
the group have reached the barrier call and only returning then.

Barrier Implementation

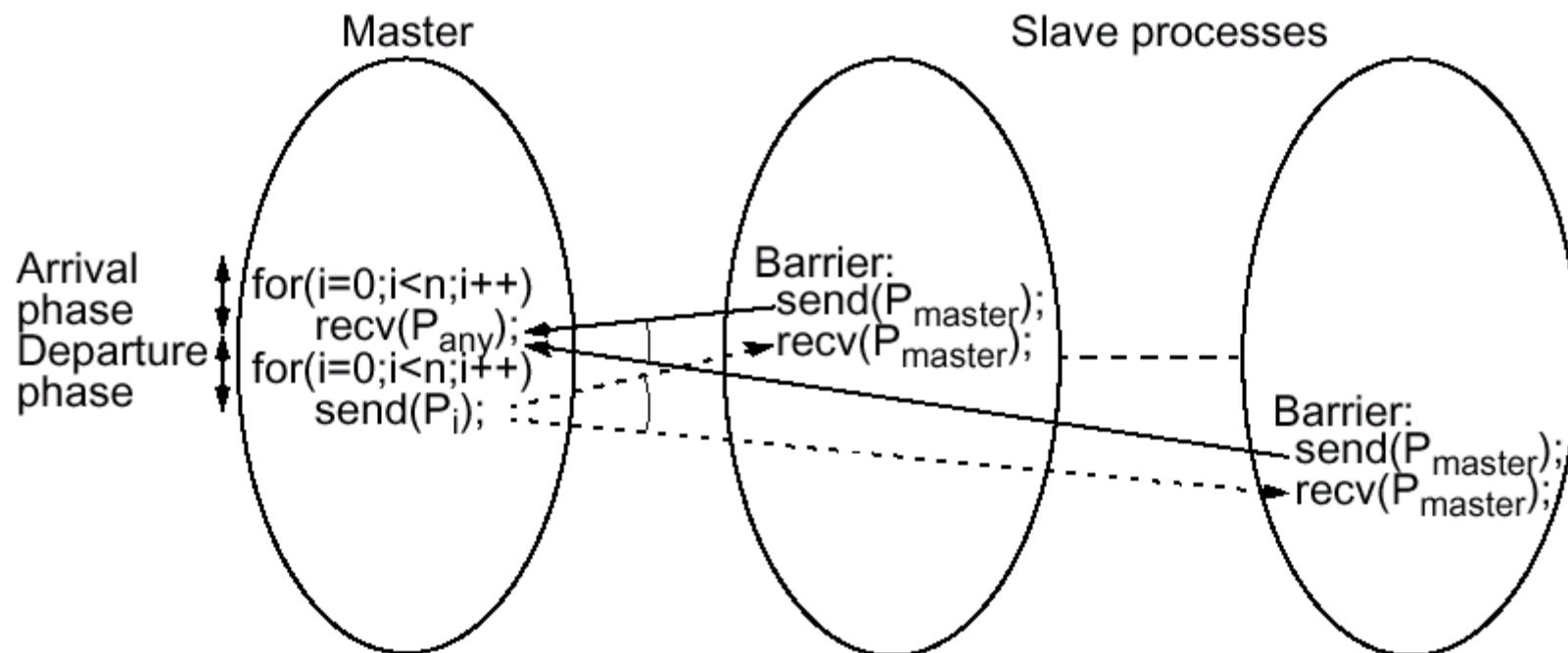
Centralized counter implementation (*linear barrier*):



Counter-based barriers often have two phases:

- A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.
- Then processes move to departure phase and are released.

Barrier implementation in a message-passing system



Example code:

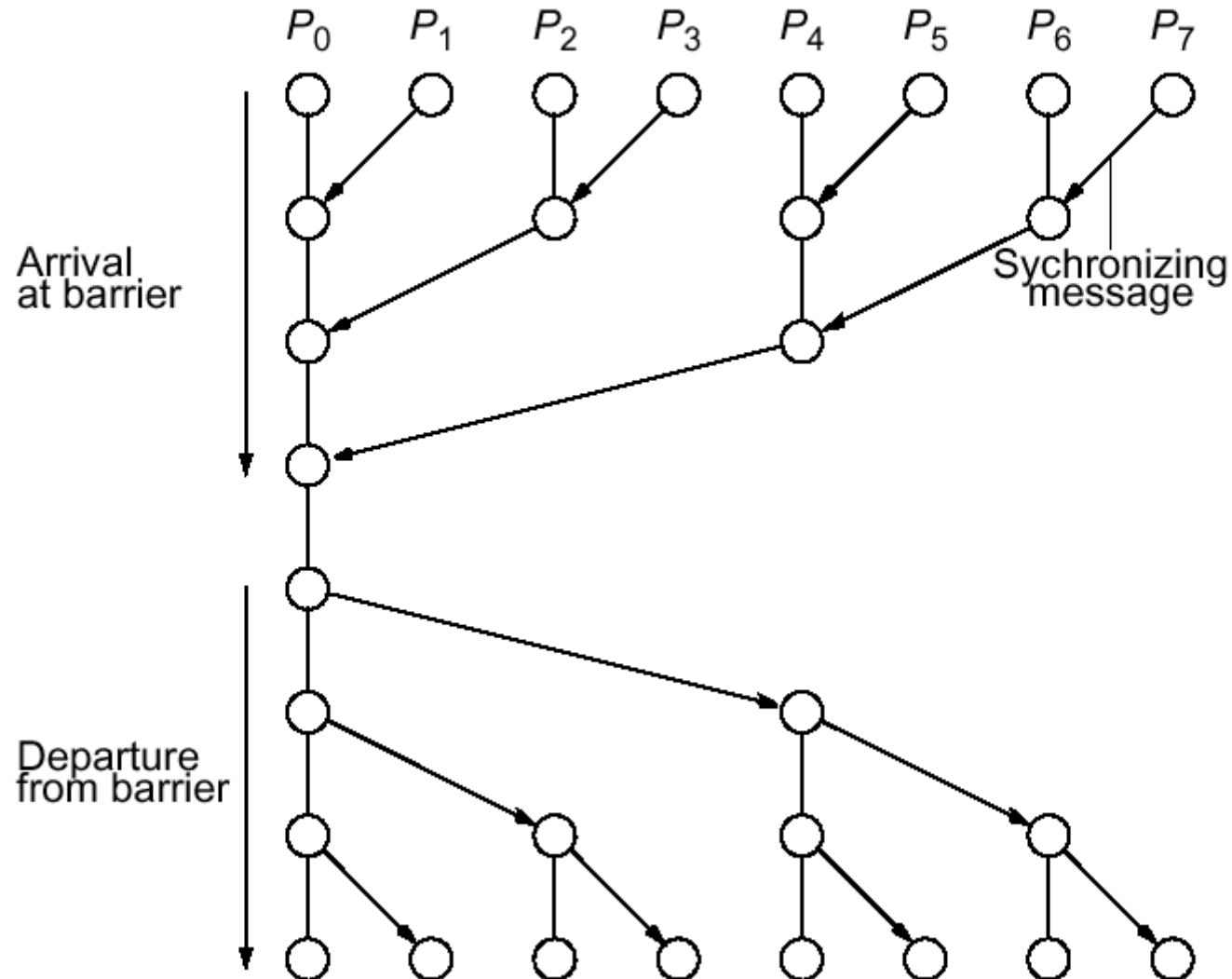
Master:

```
for (i = 0; i < n; i++)/*count slaves as they reach  
barrier*/  
    recv(Pany);  
for (i = 0; i < n; i++)/* release slaves */  
    send(Pi);
```

Slave processes:

```
send(Pmaster);  
recv(Pmaster);
```

Tree Barrier



Tree Implementation

More efficient. Suppose 8 processes, $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$:

First stage: P_1 sends message to P_0 ; (when P_1 reaches its barrier)

P_3 sends message to P_2 ; (when P_3 reaches its barrier)

P_5 sends message to P_4 ; (when P_5 reaches its barrier)

P_7 sends message to P_6 ; (when P_7 reaches its barrier)

Second stage: P_2 sends message to P_0 ; (P_2 & P_3 reached their barrier)

P_6 sends message to P_4 ; (P_6 & P_7 reached their barrier)

Third stage: P_4 sends message to P_0 ; (P_4, P_5, P_6 , & P_7 reached barrier)

P_0 terminates arrival phase;

(when P_0 reaches barrier & received message from P_4)

Release with a reverse tree construction.

Butterfly Barrier

First stage

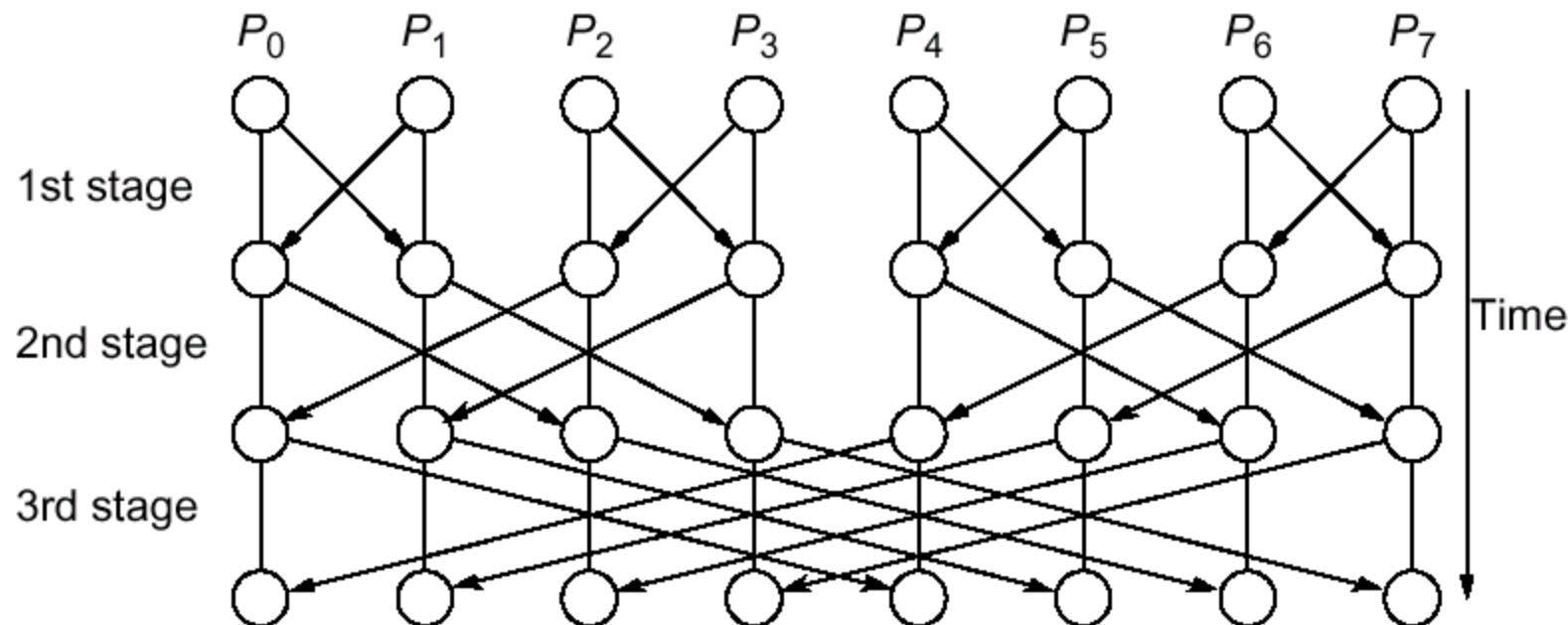
$$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$$

Second stage

$$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$$

Third stage

$$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$$



Deadlock

When a pair of processes each send and receive from each other, deadlock may occur.

Deadlock will occur if both processes perform the send, using synchronous routines first (or blocking routines without sufficient buffering). This is because neither will return; they will wait for matching receives that are never reached.

A Solution

Arrange for one process to receive first and then send and the other process to send first and then receive.

Example

Linear pipeline, deadlock can be avoided by arranging so the even-numbered processes perform their sends first and the odd-numbered processes perform their receives first.

Combined deadlock-free blocking `sendrecv()` routines

MPI provides `MPI_Sendrecv()` and `MPI_Sendrecv_replace()`.

Example

Process P_{i-1}

Process P_i

Process P_{i+1}

```
sendrecv( $R_i$ ) ;  $\leftrightarrow$  sendrecv( $R_{i-1}$ ) ;  
                    sendrecv( $R_{i+1}$ ) ;  $\leftrightarrow$  sendrecv( $R_i$ ) ;
```

sendrev()s have 12 parameters!

Synchronized Computations

Can be classified as:

- Fully synchronous

or

- Locally synchronous

In fully synchronous, all processes involved in the computation must be synchronized.

In locally synchronous, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation

Fully Synchronized Computation Examples

Data Parallel Computations

Same operation performed on different data elements simultaneously; i.e., in parallel.

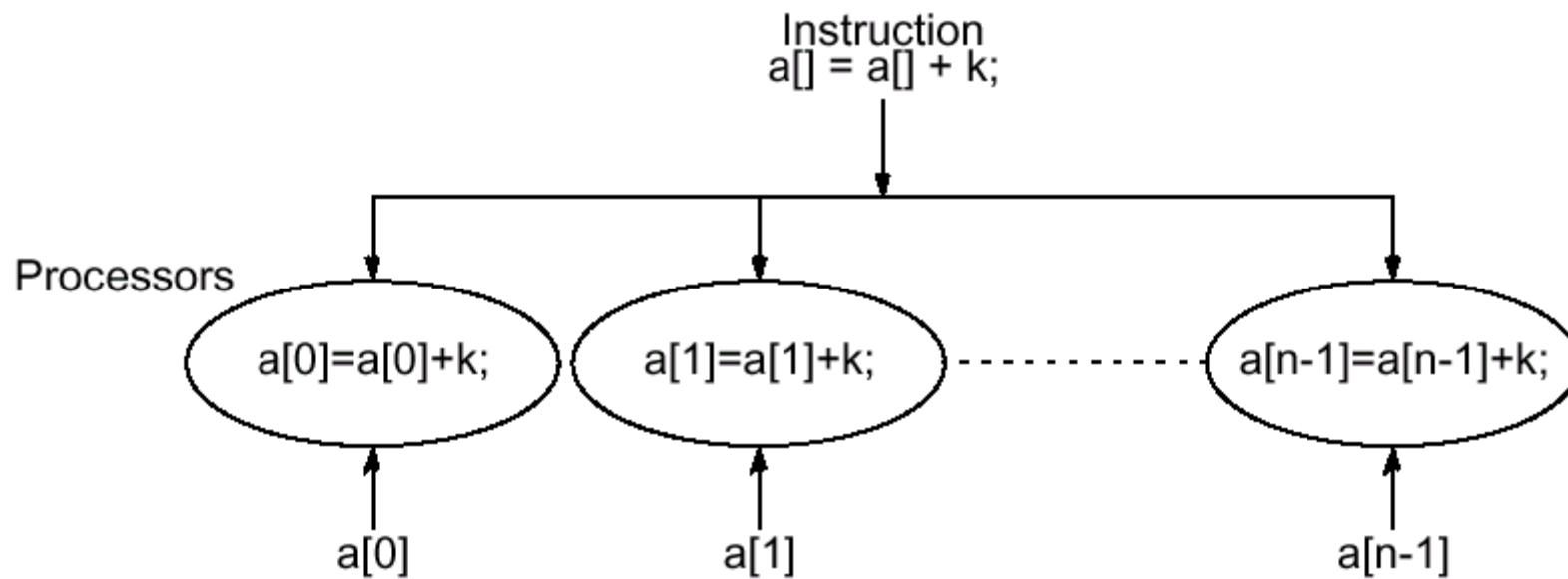
Example

To add the same constant to each element of an array:

```
for (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

The statement $a[i] = a[i] + k$ could be executed simultaneously by multiple processors, each using a different index $i (0 < i \leq n)$.

Data Parallel Computation



Forall construct

Special “parallel” construct in parallel programming languages to specify data parallel operations

Example

```
forall (i = 0; i < n; i++) {  
    body  
}
```

states that n instances of the statements of the body can be executed simultaneously.

One value of the loop variable i is valid in each instance of the body, the first instance has $i = 0$, the next $i = 1$, and so on.

To add **k** to each element of an array, **a**, we can write

```
forall (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

Data parallel technique applied to multiprocessors and multiccomputers.

Example

To add **k** to the elements of an array:

```
i = myrank;
a[i] = a[i] + k; /* body */
barrier(mygroup);
```

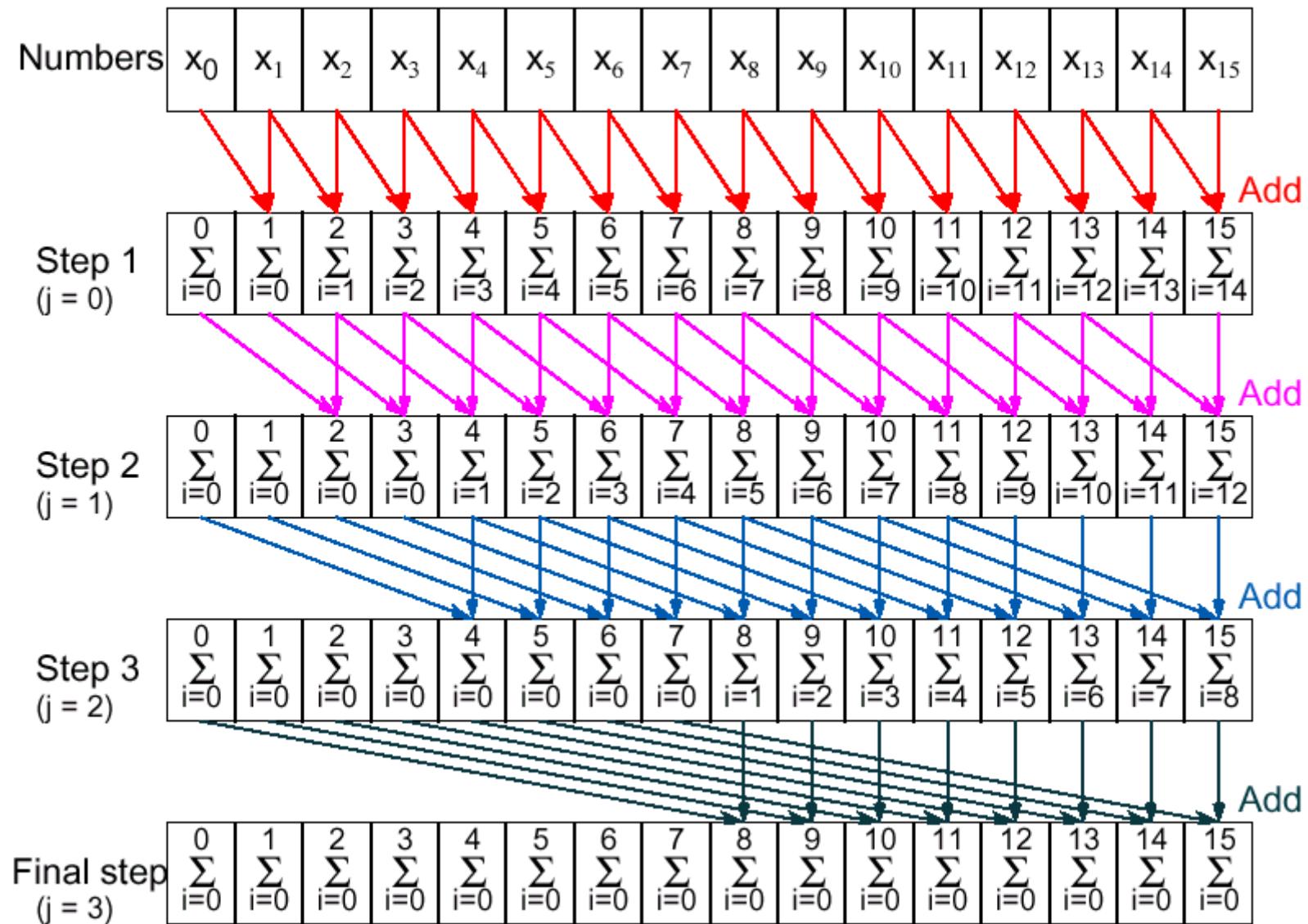
where **myrank** is a process rank between 0 and $n - 1$.

Data Parallel Example - Prefix Sum Problem

Given a list of numbers, x_0, \dots, x_{n-1} , compute all the partial summations (i.e., $x_0 + x_1; x_0 + x_1 + x_2; x_0 + x_1 + x_2 + x_3; \dots$).

Can also be defined with associative operations other than addition.
Widely studied. Practical applications in areas such as processor allocation, data compaction, sorting, and polynomial evaluation.

Data parallel prefix sum operation



Sequential code

```
for (j = 0; j < log(n); j++)/* at each step, add*/
  for (i = 2j; i < n; i++)/* to accumulating sum */
    x[i] = x[i] + x[i - 2j];
```

Parallel code

```
for (j = 0; j < log(n); j++) /* at each step, add */
  forall (i = 0; i < n; i++)/*to sum */
    if (i >= 2j) x[i] = x[i] + x[i - 2j];
```

Synchronous Iteration (Synchronous Parallelism)

Each iteration composed of several processes that start together at beginning of iteration. Next iteration cannot begin until all processes have finished previous iteration. Using **forall**:

```
for (j = 0; j < n; j++) /*for each synch. iteration */
    forall (i = 0; i < N; i++) /*N procs each using*/
        body(i);                /* specific value of i */
    }
```

or:

```
for (j = 0; j < n; j++) /*for each synchr. iteration */
    i = myrank;           /*find value of i to be used */
    body(i);
    barrier(mygroup);
}
```

Another fully synchronous computation example

Solving a General System of Linear Equations **by Iteration**

Suppose the equations are of a general form with n equations and n unknowns

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

.

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \dots + a_{2,n-1}x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \dots + a_{1,n-1}x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \dots + a_{0,n-1}x_{n-1} = b_0$$

where the unknowns are $x_0, x_1, x_2, \dots, x_{n-1}$ ($0 \leq i < n$).

By rearranging the i th equation:

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots + a_{i,n-1}x_{n-1} = b_i$$

to

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots + a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \dots + a_{i,n-1}x_{n-1})]$$

or

$$x_i = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j \right]$$

This equation gives x_i in terms of the other unknowns and can be used as an iteration formula for each of the unknowns to obtain better approximations.

Jacobi Iteration

All values of x are updated **together**.

Can be proven that the Jacobi method will converge if the diagonal values of a have an absolute value greater than the sum of the absolute values of the other a 's on the row (the array of a 's is *diagonally dominant*) i.e. if

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

This condition is a sufficient but not a necessary condition.

Termination

A simple, common approach. Compare values computed in one iteration to values obtained from the previous iteration.

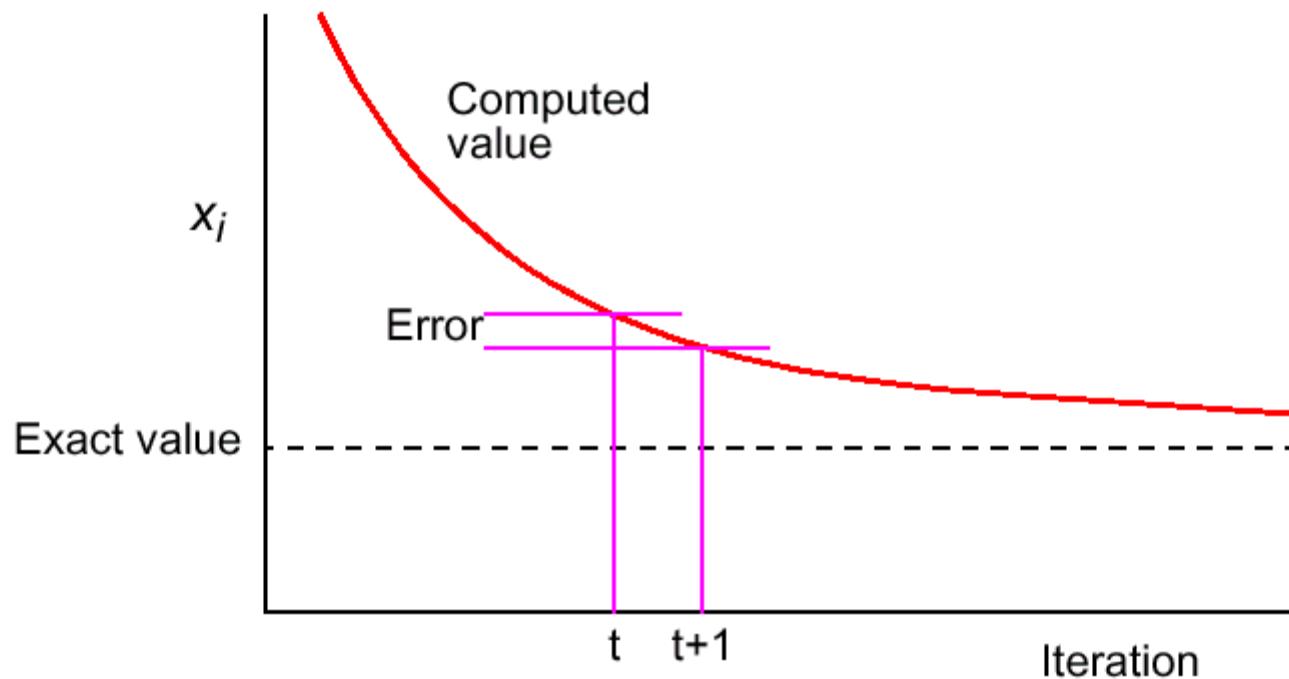
Terminate computation when all values are within given tolerance;
i.e., when

$$|x_i^t - x_i^{t-1}| < \text{error tolerance}$$

for all i , where x_i^t is the value of x_i after the t th iteration and x_i^{t-1} is the value of x_i after the $(t-1)$ th iteration.

However, this does not guarantee the solution to that accuracy.

Convergence Rate



Sequential Code.

```
for (i=0; i<n; i++)
    x[i] = b[i]; /*initialize unknowns*/

for (it=0; it<limit; it++) {
    for (i=0; i<n; i++) /* for each unknown*/
        sum = 0;
        for (j=0; j<n; j++) /*compute summation of a[][]x[]*/
            if (i != j) sum += a[i][j] * x[j];
        new_x[i] = (b[i]-sum)/a[i][i]; /*compute unknown*/
    }
    for (i=0; i<n; i++)
        x[i] = new_x[i]; /*update values*/
}
```

Can be more written in a more efficient way!

Parallel Code

Process P_i could be of the form

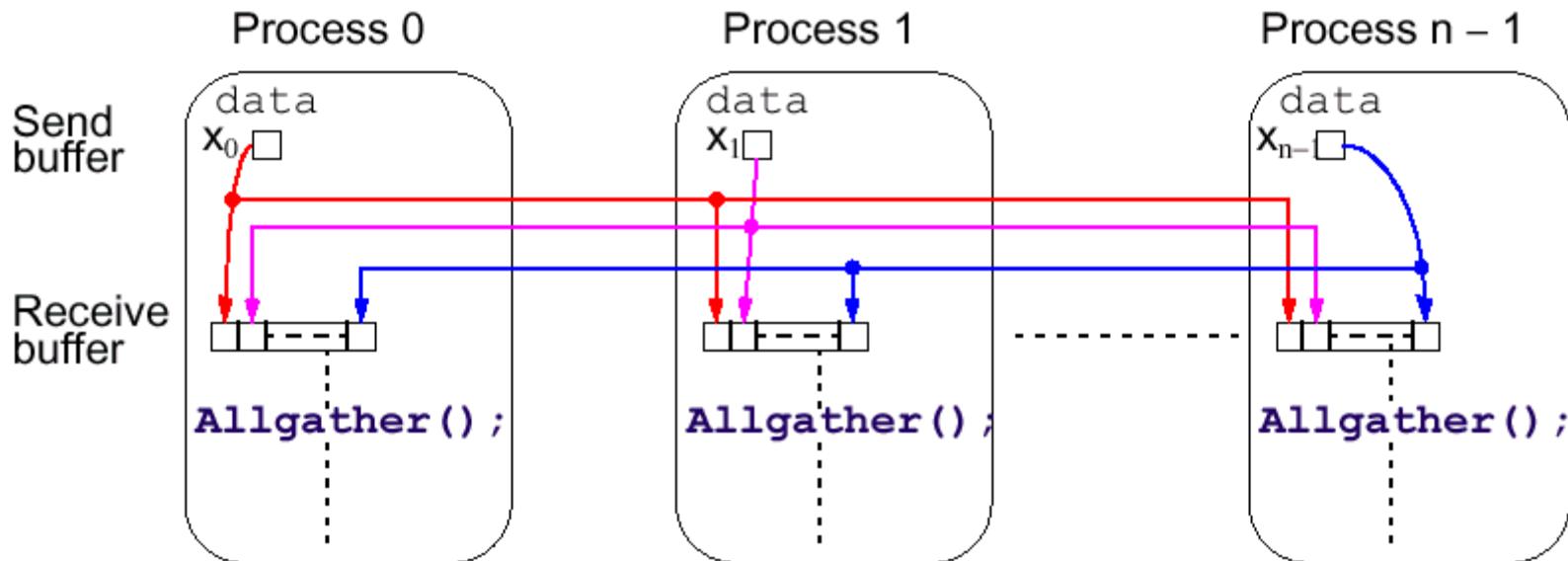
```
x[i] = b[i];                                /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 0; j < n; j++)                  /* compute summation */
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];/* compute unknown */
    allgather(&new_x[i]);                  /*bcast/rec values */
    global_barrier();                         /* wait for all procs */
}
```

allgather() sends the newly computed value of **x[i]** from process i to every other process and collects data broadcast from the other processes.

Introduce a new message-passing operation - Allgather.

Allgather

Broadcast and gather values in one composite construction.



Partitioning

Usually number of processors much fewer than number of data items to be processed. Partition the problem so that processors take on more than one data item.

block allocation – allocate groups of consecutive unknowns to processors in increasing order.

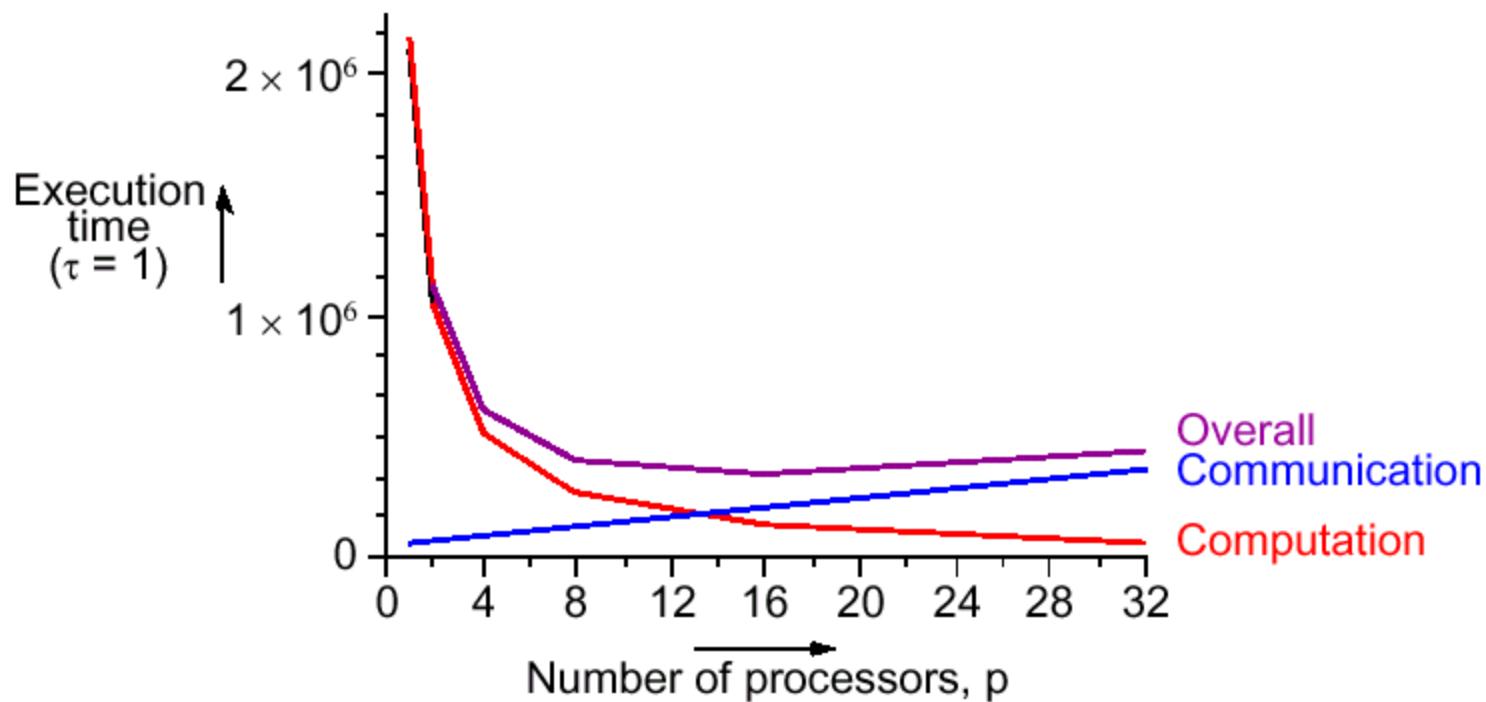
cyclic allocation – processors are allocated one unknown in order; i.e., processor P_0 is allocated $x_0, x_p, x_{2p}, \dots, x_{((n/p)-1)p}$, processor P_1 is allocated $x_1, x_{p+1}, x_{2p+1}, \dots, x_{((n/p)-1)p+1}$, and so on.

Cyclic allocation no particular advantage here (Indeed, may be disadvantageous because the indices of unknowns have to be computed in a more complex way).

Effects of computation and communication in Jacobi iteration

Consequences of different numbers of processors done in textbook.

Get:



Analysis.

Suppose p processors and n equations (unknowns):

- Each processor operates upon n/p unknowns
- t iterations per processor

Computation.

$$\underline{t_{\text{comp}}} = n/p (2n+4) t$$

$\underline{1 *}$ and $1 +$ in inner loop --- $1 ^*$, $2 -$, and $1 /$ in outer loop.

Communication.

$$t_{\text{comm}} = p (t_{\text{startup}} + (n/p) t_{\text{data}})t = pt_{\text{startup}} + nt_{\text{data}})t$$

The resulting total execution time has one component that is decreasing function of p and another that is increasing function of p .

We can find the minimum by differentiation.

Locally Synchronous Computation

Heat Distribution Problem

An area has known temperatures along each of its edges. Find the temperature distribution within.

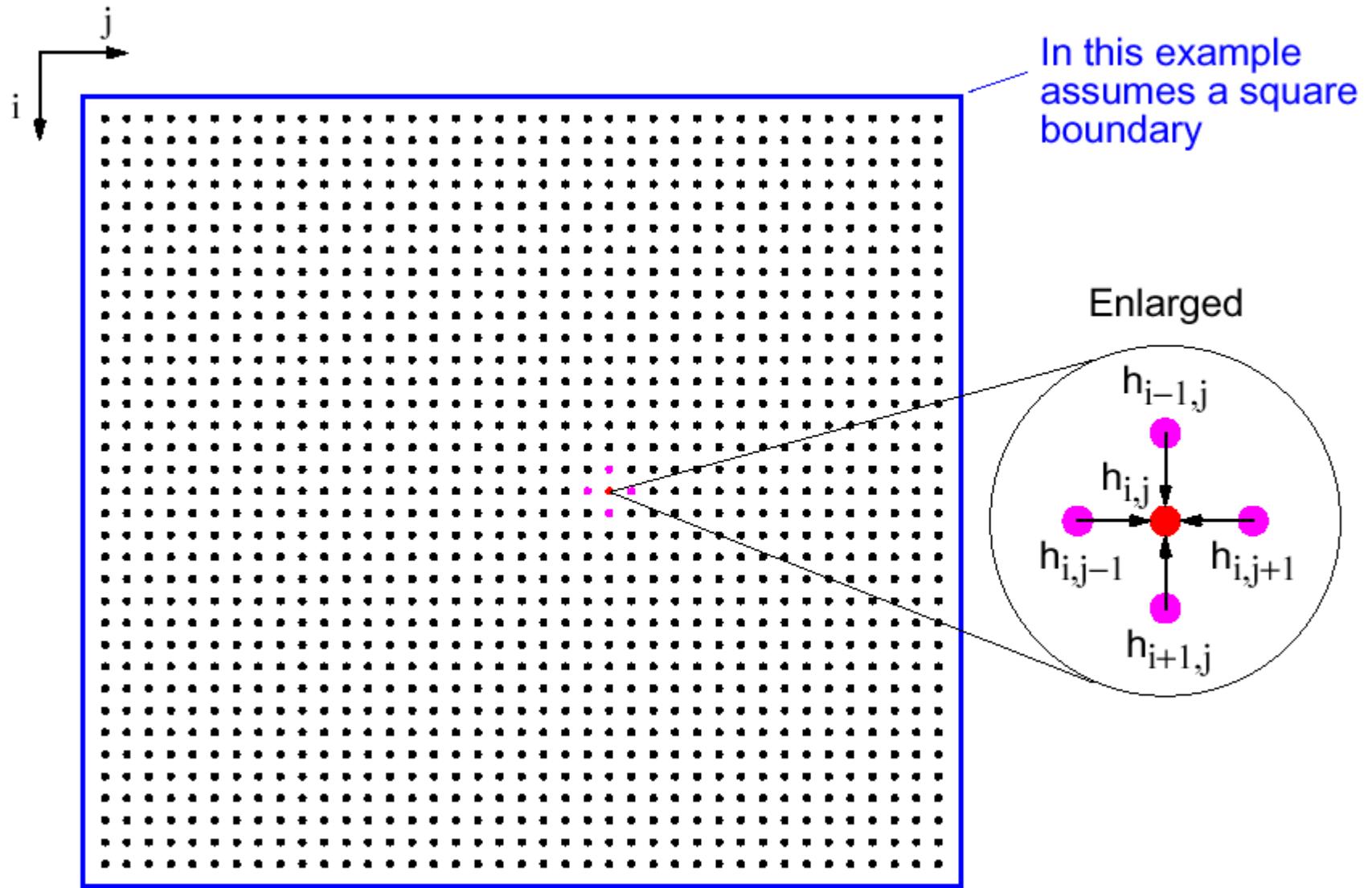
Divide area into fine mesh of points, $h_{i,j}$. Temperature at an inside point taken to be average of temperatures of four neighboring points. Convenient to describe edges by points.

Temperature of each point by iterating the equation:

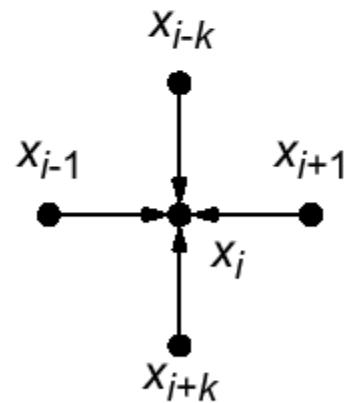
$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

$(0 < i < n, 0 < j < n)$ for a fixed number of iterations or until the difference between iterations less than some very small amount.

Heat Distribution Problem



Natural ordering of heat distribution problem



Number points from 1 for convenience and include those representing the edges. Each point will then use the equation

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-k} + x_{i+k}}{4}$$

Could be written as a linear equation containing the unknowns x_{i-k} , x_{i-1} , x_{i+1} , and x_{i+k} :

$$x_{i-k} + x_{i-1} - 4x_i + x_{i+1} + x_{i+k} = 0$$

Notice: solving a (sparse) system of linear equations.

Also solving [Laplace's equation](#).

Sequential Code

Using a fixed number of iterations

```
for (iteration = 0; iteration < limit; iteration++) {  
    for (i = 1; i < n; i++)  
        for (j = 1; j < n; j++)  
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-  
1]+h[i][j+1]);  
    for (i = 1; i < n; i++)/* update points */  
        for (j = 1; j < n; j++)  
            h[i][j] = g[i][j];  
}
```

To stop at some precision:

```
do {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    for (i = 1; i < n; i++)/* update points */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];
    continue = FALSE;          /* indicates whether to continue */
    for (i = 1; i < n; i++)/* check each pt for convergence */
        for (j = 1; j < n; j++)
            if (!converged(i,j) /* point found not converged */
                continue = TRUE;
                break;
}
} while (continue == TRUE);
```

Parallel Code

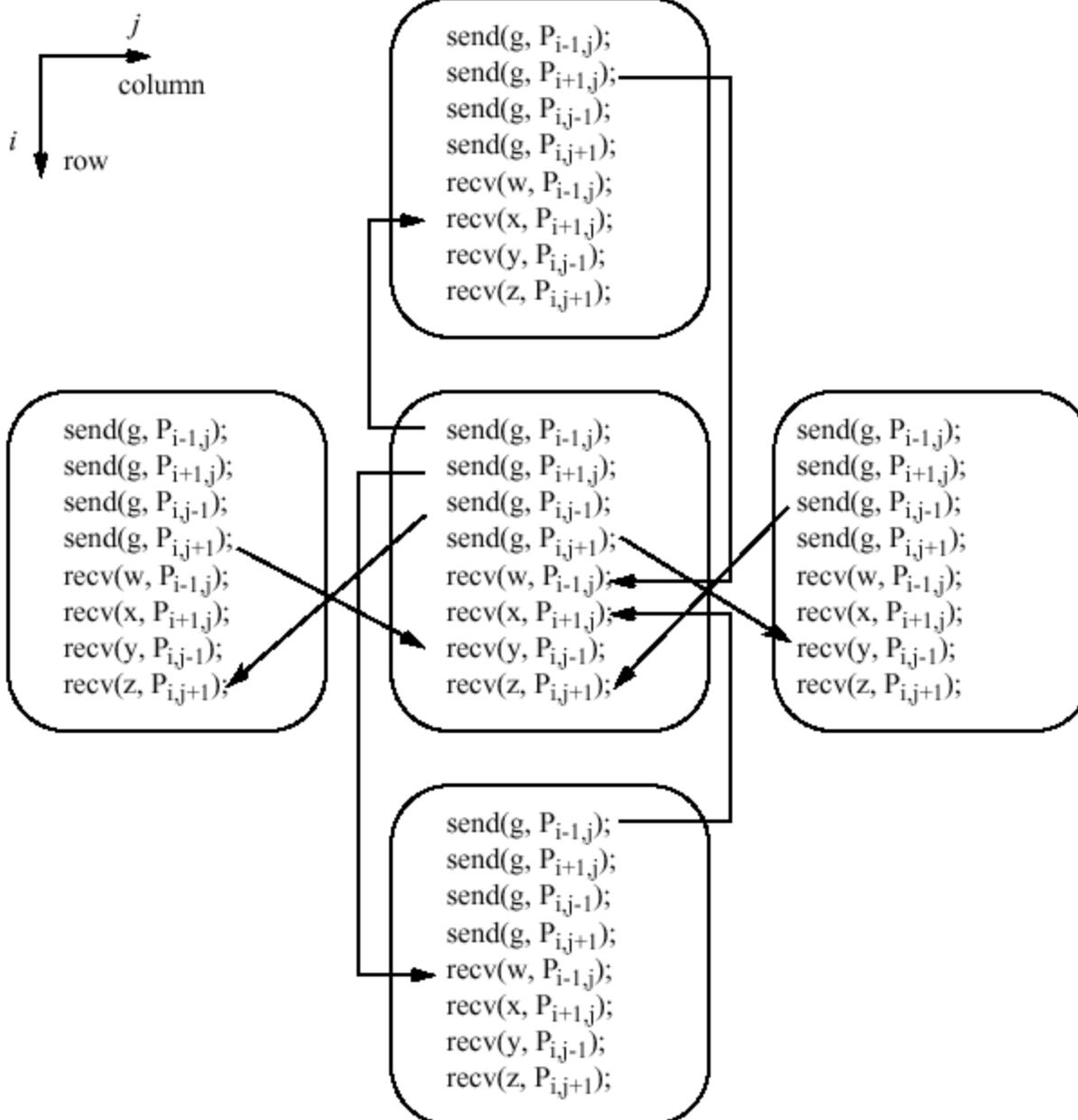
With fixed number of iterations, $P_{i,j}$ (except for the boundary points):

```
for (iteration = 0; iteration < limit; iteration++) {  
    g = 0.25 * (w + x + y + z);  
    send(&g, Pi-1,j); /* non-blocking sends */  
    send(&g, Pi+1,j);  
    send(&g, Pi,j-1);  
    send(&g, Pi,j+1);  
    recv(&w, Pi-1,j); /* synchronous receives */  
    recv(&x, Pi+1,j);  
    recv(&y, Pi,j-1);  
    recv(&z, Pi,j+1);  
}
```

Local barrier

Important to use **send()**s that do not block while waiting for the **recv()**s; otherwise the processes would deadlock, each waiting for a **recv()** before moving on - **recv()**s must be synchronous and wait for the **send()**s.

Message passing for heat distribution problem



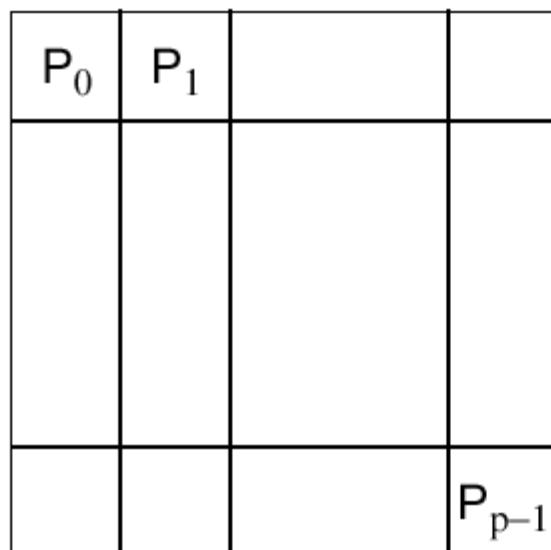
Version where processes stop when they reach their required precision:

```
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    send(&g, Pi-1,j);
    send(&g, Pi+1,j);
    send(&g, Pi,j-1);
    send(&g, Pi,j+1);
    recv(&w, Pi-1,j);
    recv(&x, Pi+1,j);
    recv(&y, Pi,j-1);
    recv(&z, Pi,j+1);
} while((!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, Pmaster);
```

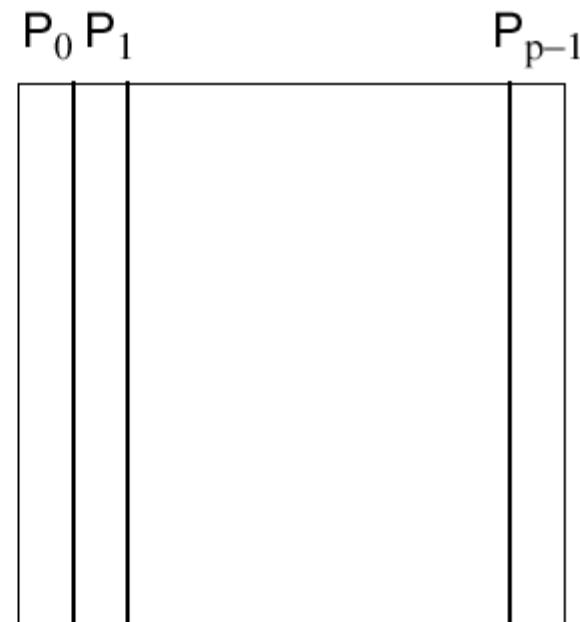
Partitioning

Normally allocate more than one point to each processor, because many more points than processors.

Points could be partitioned into square blocks or strips:



Blocks

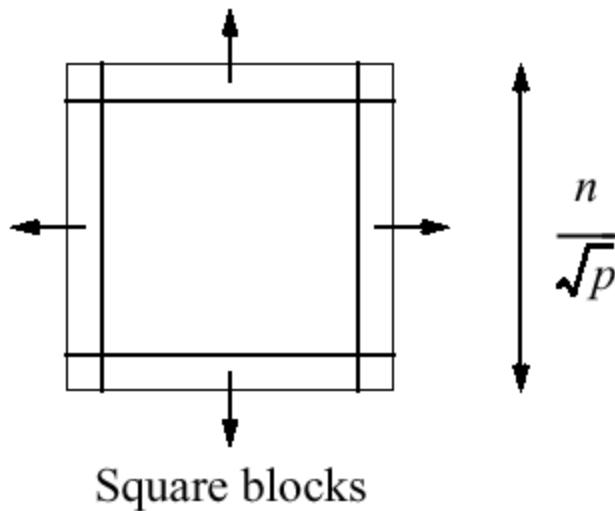


Strips (columns)

Block partition

Four edges where data points exchanged. Communication time given by

$$t_{\text{commsq}} = 8 \left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}} \right)$$

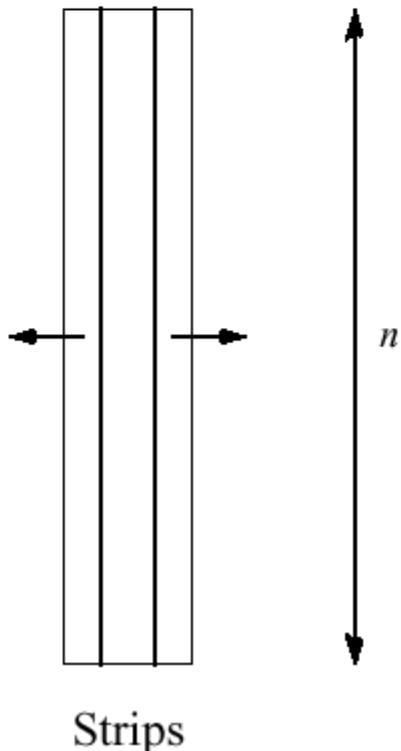


Communication consequences of partitioning

Strip partition

Two edges where data points are exchanged. Communication time is given by

$$t_{\text{commcol}} = 4(t_{\text{startup}} + nt_{\text{data}})$$



Optimum

In general, the strip partition is best for a large startup time, and a block partition is best for a small startup time.

With the previous equations, the block partition has a larger communication time than the strip partition if

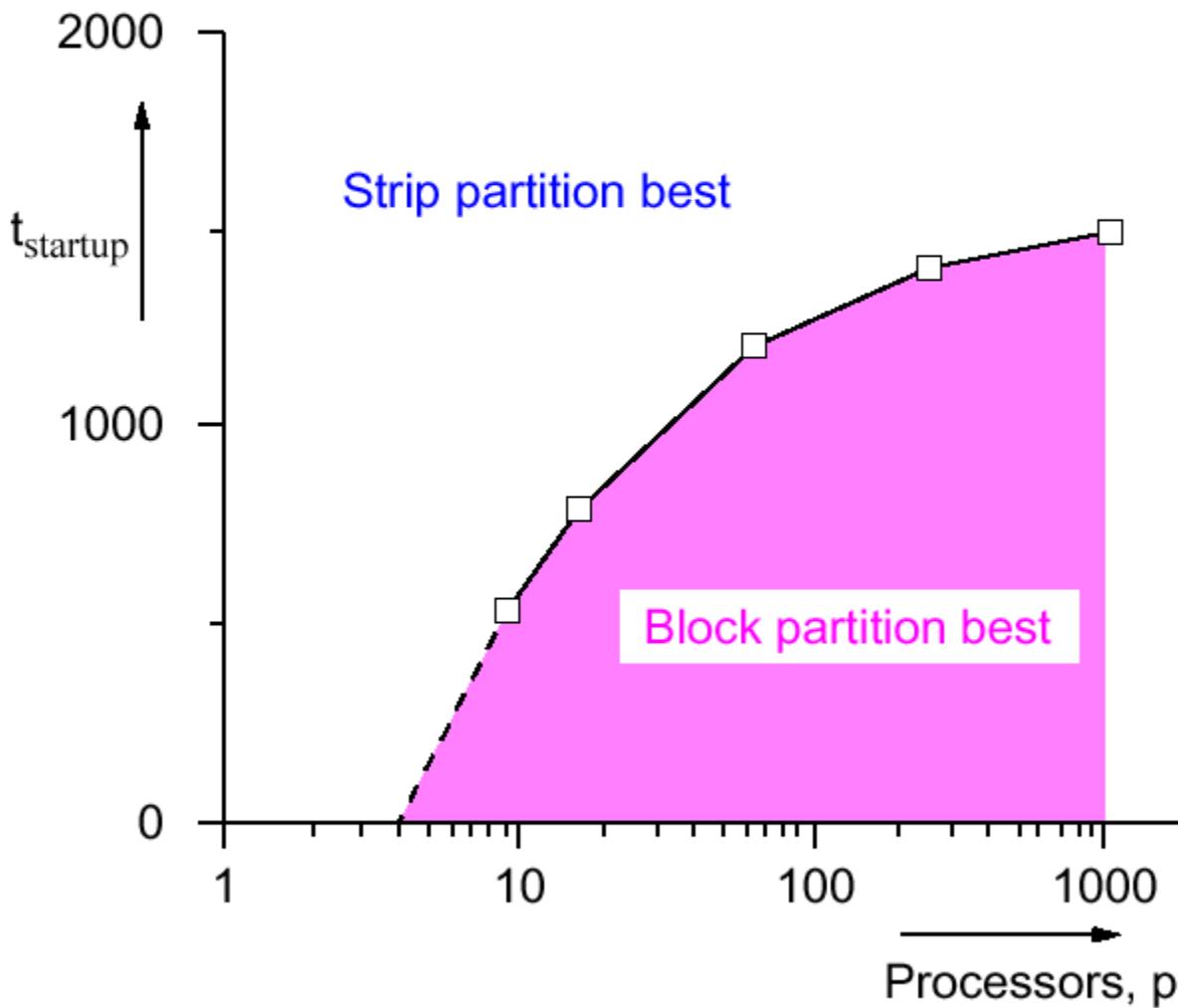
$$8\left(t_{\text{startup}} + \frac{n}{\sqrt{p}}t_{\text{data}}\right) > 4(t_{\text{startup}} + nt_{\text{data}})$$

or

$$t_{\text{startup}} > n\left(1 - \frac{2}{\sqrt{p}}\right)t_{\text{data}}$$

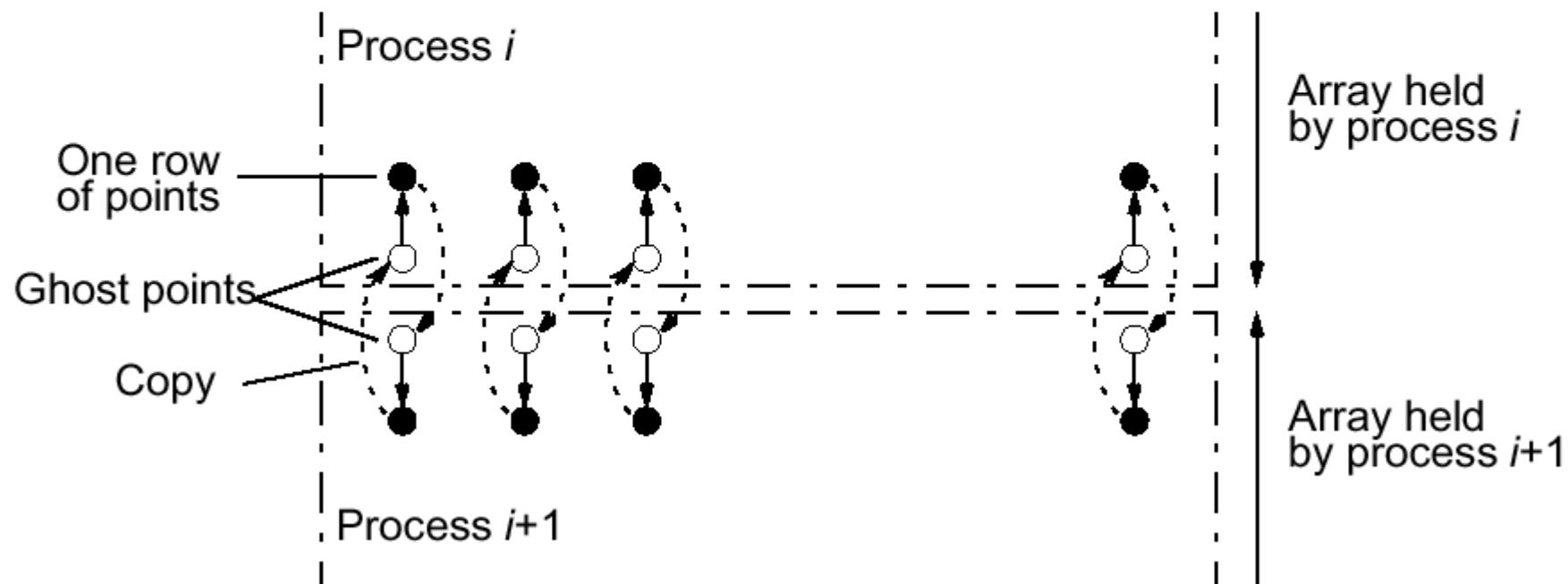
($p \geq 9$).

Startup times for block and strip partitions



Ghost Points

An additional row of points at each edge that hold the values from the adjacent edge. Each array of points is increased to accommodate the ghost rows.



Safety and Deadlock

When all processes send their messages **first** and then receive all of their messages is “**unsafe**” because it relies upon buffering in the **send()**s. The amount of buffering is not specified in MPI.

If insufficient storage available, send routine may be delayed from returning until storage becomes available or until message can be sent without buffering.

Then, a locally blocking **send()** could behave as a synchronous **send()**, only returning when the matching **recv()** is executed. Since a matching **recv()** would never be executed if all the **send()**s are synchronous, **deadlock would occur**.

Making the code safe

Alternate the order of the `send()`s and `recv()`s in adjacent processes so that only one process performs the `send()`s first:

```
}
```

Then even synchronous `send()`s would not cause deadlock.

Good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

MPI Safe Message Passing Routines

MPI offers several alternative methods for safe communication:

- Combined send and receive routines:

`MPI_Sendrecv()`

which is guaranteed not to deadlock

- Buffered send():

`MPI_Bsend()`

here the user provides explicit storage space

- Nonblocking routines:

`MPI_Isend()` and `MPI_Irecv()`

which return immediately. Separate routine used to establish whether message has been received - **`MPI_Wait()`**,
`MPI_Waitall()`,**`MPI_Waitany()`**,**`MPI_Test()`**,**`MPI_Testall()`**,
or `MPI_Testany()`.

Other fully synchronous problems

Cellular Automata

The problem space is divided into cells.

Each cell can be in one of a finite number of states.

Cells affected by their neighbors according to certain rules, and all cells are affected simultaneously in a “generation.”

Rules re-applied in subsequent generations so that cells evolve, or change state, from generation to generation.

Most famous cellular automata is the “Game of Life” devised by John Horton Conway, a Cambridge mathematician.

Also good assignment for graphical output.

The Game of Life

Board game - theoretically infinite two-dimensional array of cells. Each cell can hold one “organism” and has eight neighboring cells, including those diagonally adjacent. Initially, some cells occupied.

The following rules apply:

1. Every organism with two or three neighboring organisms survives for the next generation.
2. Every organism with four or more neighbors dies from overpopulation.
3. Every organism with one neighbor or none dies from isolation.
4. Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.

These rules were derived by Conway “after a long period of experimentation.”

Simple Fun Examples of Cellular Automata

“Sharks and Fishes”

An ocean could be modeled as a three-dimensional array of cells.
Each cell can hold one fish or one shark (but not both).

Fish and sharks follow “rules.”

Fish

Might move around according to these rules:

1. If there is one empty adjacent cell, the fish moves to this cell.
2. If there is more than one empty adjacent cell, the fish moves to one cell chosen at random.
3. If there are no empty adjacent cells, the fish stays where it is.
4. If the fish moves and has reached its breeding age, it gives birth to a baby fish, which is left in the vacating cell.
5. Fish die after x generations.

Sharks

Might be governed by the following rules:

1. If one adjacent cell is occupied by a fish, the shark moves to this cell and eats the fish.
2. If more than one adjacent cell is occupied by a fish, the shark chooses one fish at random, moves to the cell occupied by the fish, and eats the fish.
3. If no fish in adjacent cells, the shark chooses an unoccupied adjacent cell to move to in a similar manner as fish move.
4. If the shark moves and has reached its breeding age, it gives birth to a baby shark, which is left in the vacating cell.
5. If a shark has not eaten for y generations, it dies.

Similar examples: “foxes and rabbits” -Behavior of rabbits to move around happily whereas behavior of foxes is to eat any rabbits they come across.

Sample Student Output



Chapter 7

Load Balancing and Termination Detection

Load balancing – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

Termination detection – detecting when a computation has been completed. More difficult when the computation is distributed.

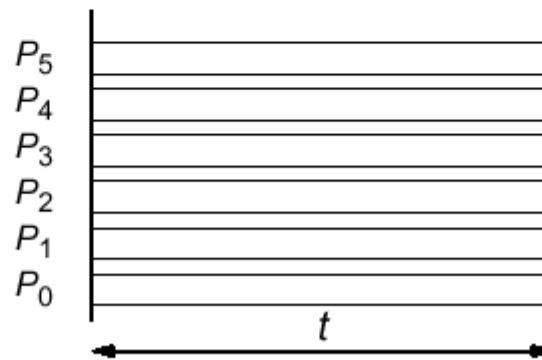
Load balancing

Processors



(a) Imperfect load balancing leading to increased execution time

Processors



(b) Perfect load balancing

Static Load Balancing

Before the execution of any process. Some potential static load-balancing techniques:

- *Round robin algorithm* — passes out tasks in sequential order of processes coming back to the first when all processes have been given a task
- *Randomized algorithms* — selects processes at random to take tasks
- *Recursive bisection* — recursively divides the problem into subproblems of equal computational effort while minimizing message passing
- *Simulated annealing* — an optimization technique
- *Genetic algorithm* — another optimization technique, described in Chapter 12

Static Load Balancing

Balance load prior to the execution. Various static load-balancing algorithms.

Several **fundamental flaws** with static load balancing even if a mathematical solution exists:

- Very difficult to estimate accurately the execution times of various parts of a program without actually executing the parts.
- Communication delays that vary under different circumstances
- Some problems have an indeterminate number of steps to reach their solution.

Dynamic Load Balancing

Vary load during the execution of the processes.

All previous factors are taken into account by making the division of load dependent upon the execution of the parts as they are being executed.

Does incur an additional overhead during execution, but it is much more effective than static load balancing

Processes and Processors

Computation will be divided into *work* or *tasks* to be performed, and **processes** perform these tasks. **Processes** are mapped onto **processors**.

Since our objective is to keep the processors busy, we are interested in the activity of the processors.

However, we often map a single process onto each processor, so we will use the terms **process** and **processor** somewhat interchangeably.

Chapter 7

Load Balancing and Termination Detection

Load balancing – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

Termination detection – detecting when a computation has been completed. More difficult when the computation is distributed.

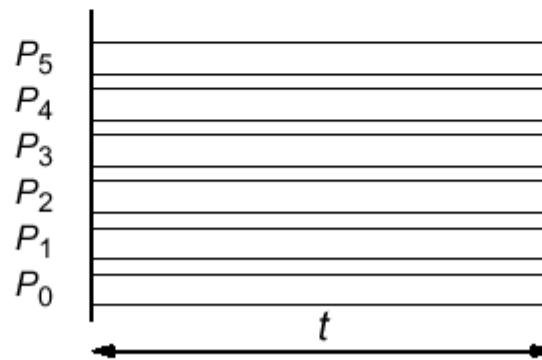
Load balancing

Processors



(a) Imperfect load balancing leading to increased execution time

Processors



(b) Perfect load balancing

Static Load Balancing

Before the execution of any process. Some potential static load-balancing techniques:

- *Round robin algorithm* — passes out tasks in sequential order of processes coming back to the first when all processes have been given a task
- *Randomized algorithms* — selects processes at random to take tasks
- *Recursive bisection* — recursively divides the problem into subproblems of equal computational effort while minimizing message passing
- *Simulated annealing* — an optimization technique
- *Genetic algorithm* — another optimization technique, described in Chapter 12

Static Load Balancing

Balance load prior to the execution. Various static load-balancing algorithms.

Several **fundamental flaws** with static load balancing even if a mathematical solution exists:

- Very difficult to estimate accurately the execution times of various parts of a program without actually executing the parts.
- Communication delays that vary under different circumstances
- Some problems have an indeterminate number of steps to reach their solution.

Dynamic Load Balancing

Vary load during the execution of the processes.

All previous factors are taken into account by making the division of load dependent upon the execution of the parts as they are being executed.

Does incur an additional overhead during execution, but it is much more effective than static load balancing

Processes and Processors

Computation will be divided into *work* or *tasks* to be performed, and **processes** perform these tasks. **Processes** are mapped onto **processors**.

Since our objective is to keep the processors busy, we are interested in the activity of the processors.

However, we often map a single process onto each processor, so we will use the terms **process** and **processor** somewhat interchangeably.

Dynamic Load Balancing

Can be classified as:

- Centralized
- Decentralized

Centralized dynamic load balancing

Tasks handed out from a centralized location. Master-slave structure.

Decentralized dynamic load balancing

Tasks are passed between arbitrary processes.

A collection of worker processes operate upon the problem and interact among themselves, finally reporting to a single process.

A worker process may receive tasks from other worker processes and may send tasks to other worker processes (to complete or pass on at their discretion).

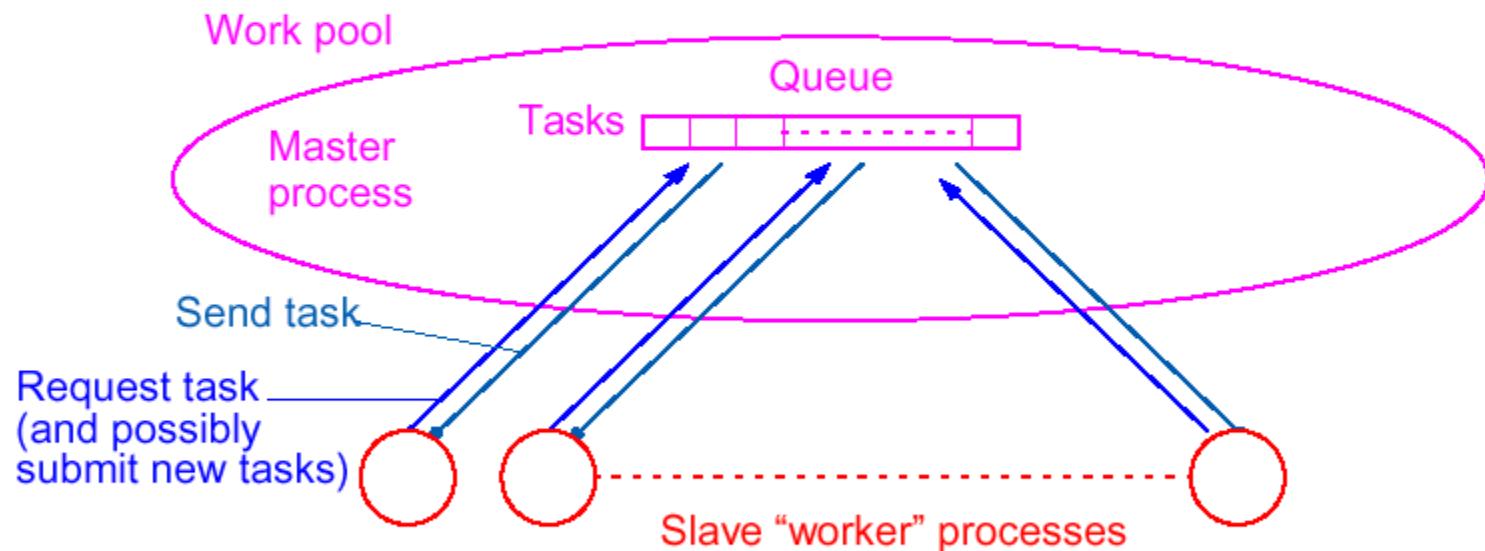
Centralized Dynamic Load Balancing

Master process(or) holds the collection of tasks to be performed.

Tasks are sent to the slave processes. When a slave process completes one task, it requests another task from the master process.

Terms used : *work pool, replicated worker, processor farm.*

Centralized work pool



Termination

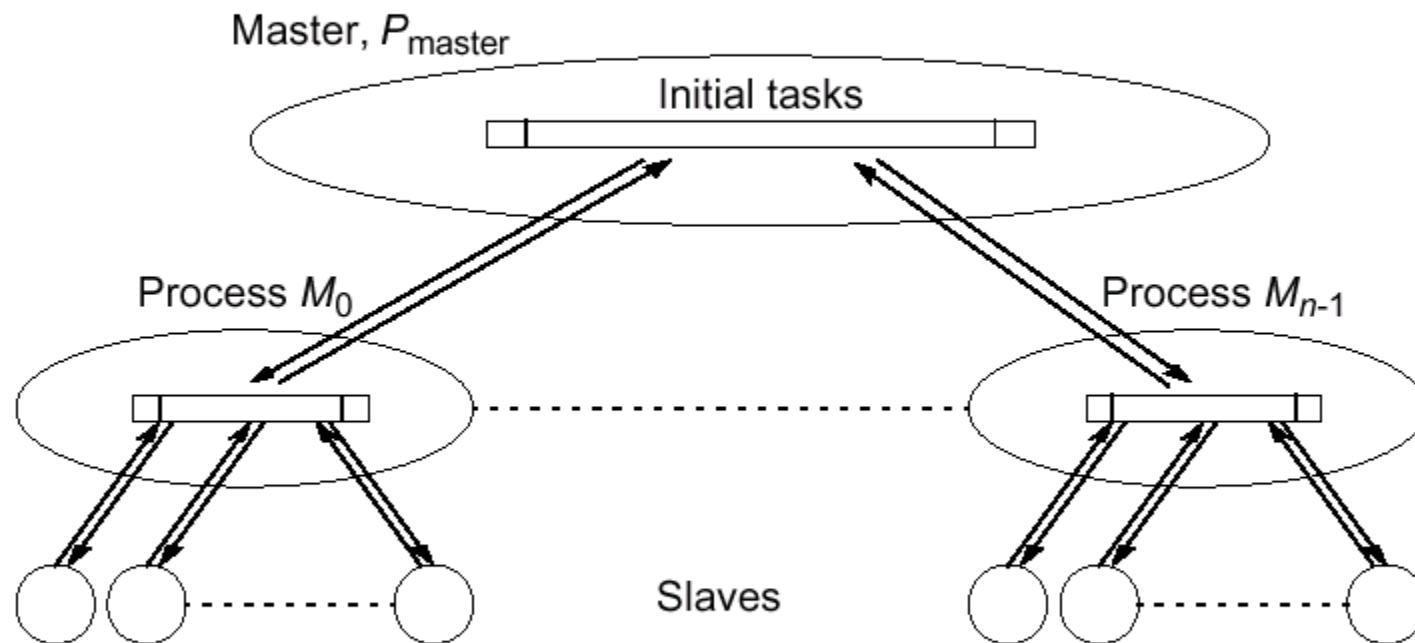
Computation terminates when:

- The task queue is empty and
- Every process has made a request for another task without any new tasks being generated

Not sufficient to terminate when task queue empty if one or more processes are still running if a running process may provide new tasks for task queue.

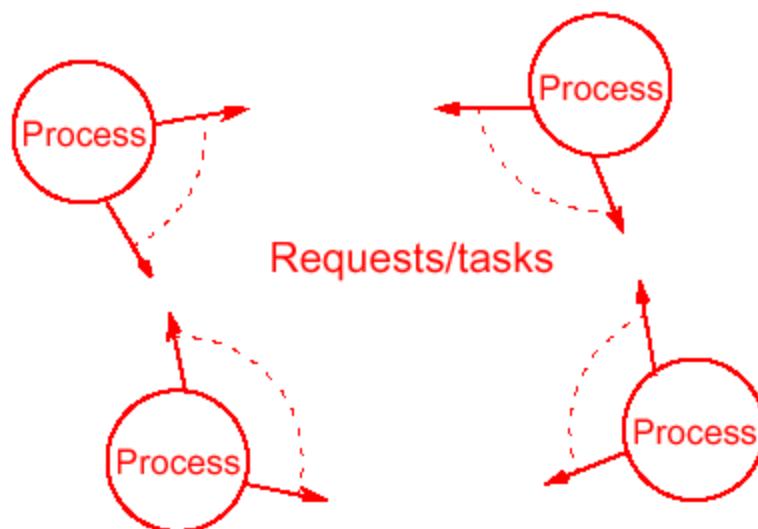
Decentralized Dynamic Load Balancing

Distributed Work Pool



Fully Distributed Work Pool

Processes to execute tasks from each other



Task Transfer Mechanisms

Receiver-Initiated Method

A process requests tasks from other processes it selects.

Typically, a process would request tasks from other processes when it has few or no tasks to perform.

Method has been shown to work well at high system load.

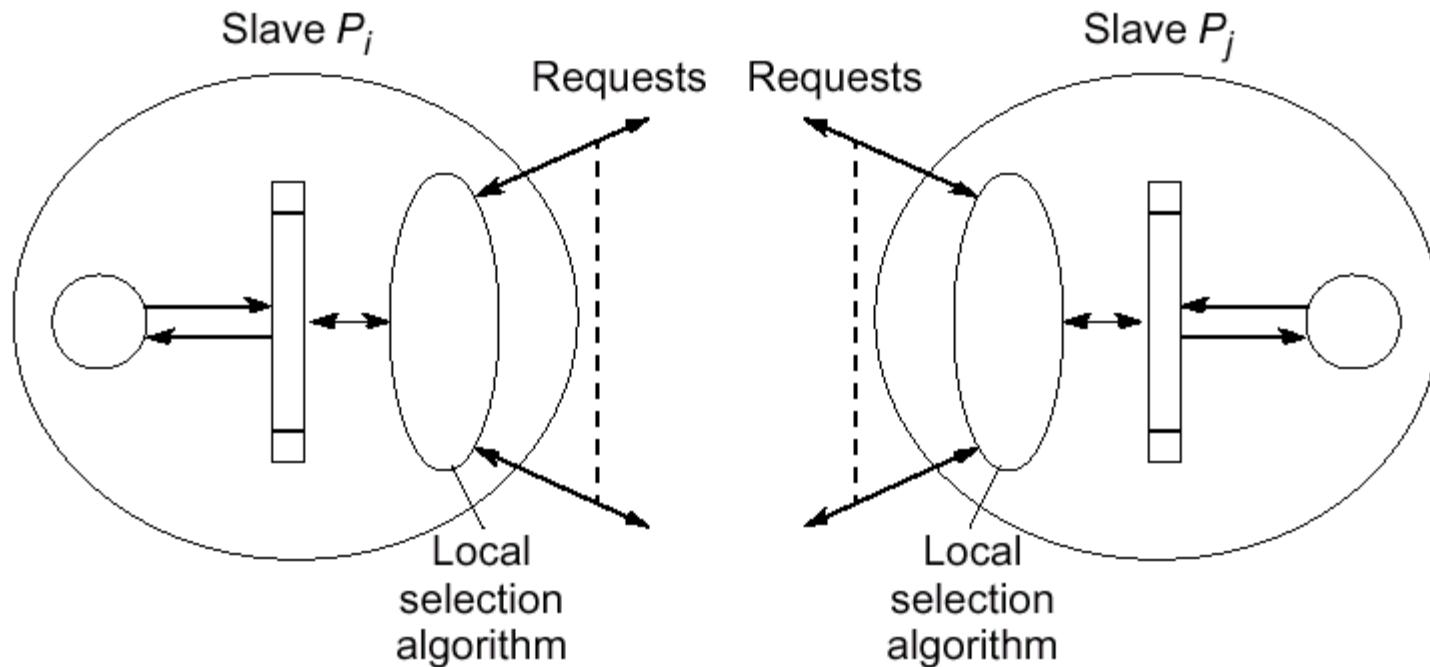
Sender-Initiated Method

A process sends tasks to other processes it selects.

Typically, a process with a heavy load passes out some of its tasks to others that are willing to accept them.

Method has been shown to work well for light overall system loads.

Decentralized selection algorithm requesting tasks between slaves



whom shall I ask next ...?

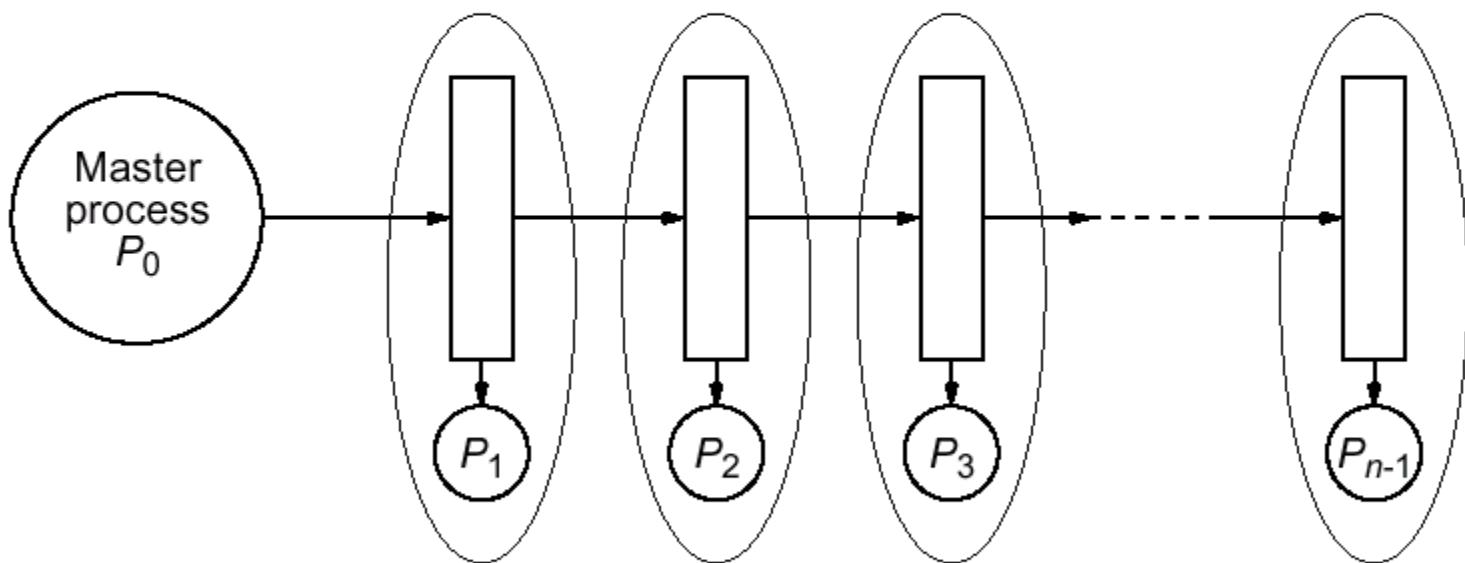
Process Selection

Algorithms for selecting a process:

Round robin algorithm – process P_i requests tasks from process P_x , where x is given by a counter that is incremented after each request, using modulo n arithmetic (n processes), excluding $x = i$.

Random polling algorithm – process P_i requests tasks from process P_x , where x is a number that is selected randomly between 0 and $n - 1$ (excluding i).

Load Balancing Using a Line Structure



The master process (P_0 in Figure 7.6) feeds the queue with tasks at one end, and the tasks are shifted down the queue.

When a “worker” process, P_i ($1 \leq i < n$), detects a task at its input from the queue and the process is idle, it takes the task from the queue.

Then the tasks to the left shuffle down the queue so that the space held by the task is filled. A new task is inserted into the left side end of the queue.

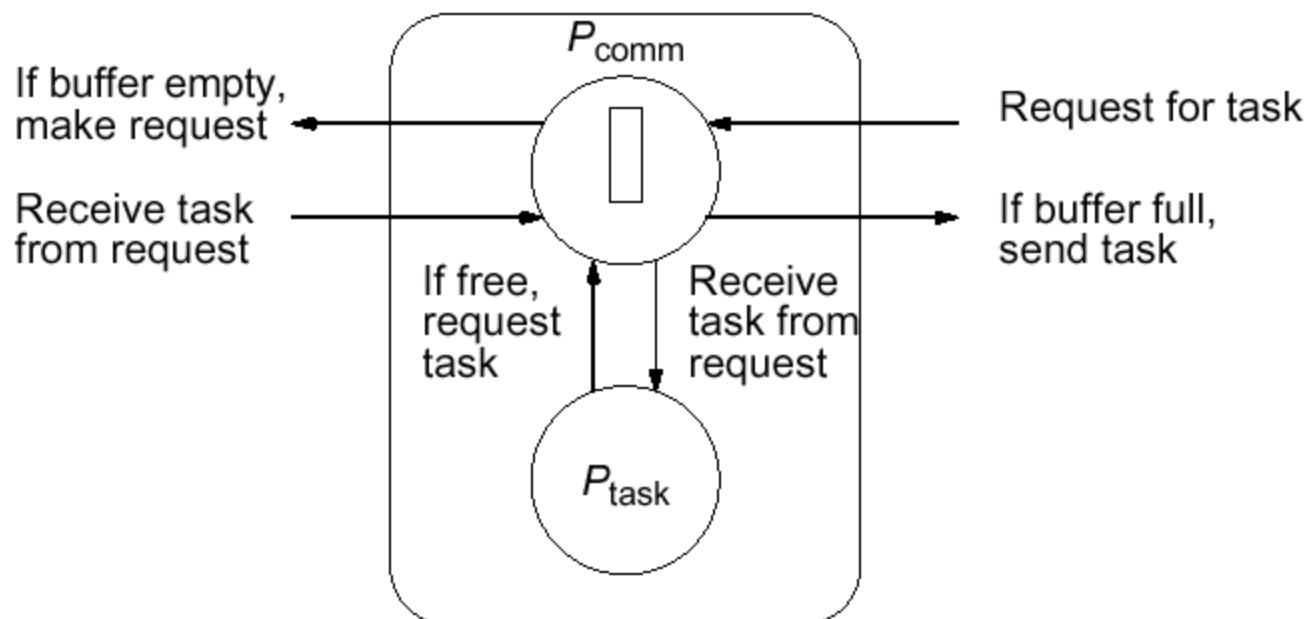
Eventually, all processes will have a task and the queue is filled with new tasks.

High-priority or larger tasks could be placed in the queue first.

Shifting Actions

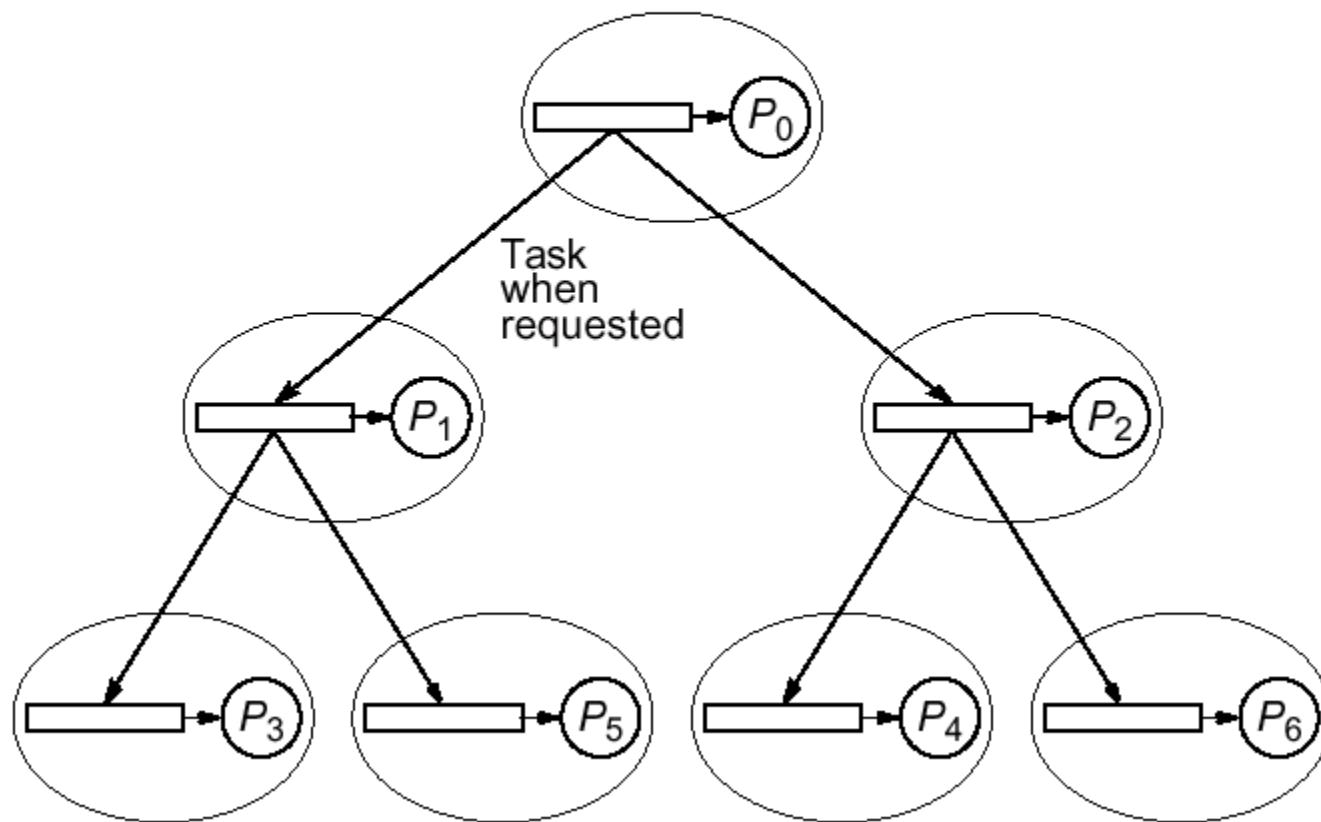
could be orchestrated by using messages between adjacent processes:

- For left and right communication
- For the current task



Load balancing using a tree

Tasks passed from node into one of the two nodes below it when node buffer empty.



Distributed Termination Detection Algorithms

Termination Conditions

At time t requires the following conditions to be satisfied:

- Application-specific local termination conditions exist throughout the collection of processes, at time t .
- There are no messages in transit between processes at time t .

Subtle difference between these termination conditions and those given for a centralized load-balancing system is having to take into account messages in transit.

Second condition necessary because a message in transit might restart a terminated process. More difficult to recognize. The time that it takes for messages to travel between processes will not be known in advance.

One very general distributed termination algorithm

Each process in one of two states:

1. Inactive - without any task to perform
2. Active

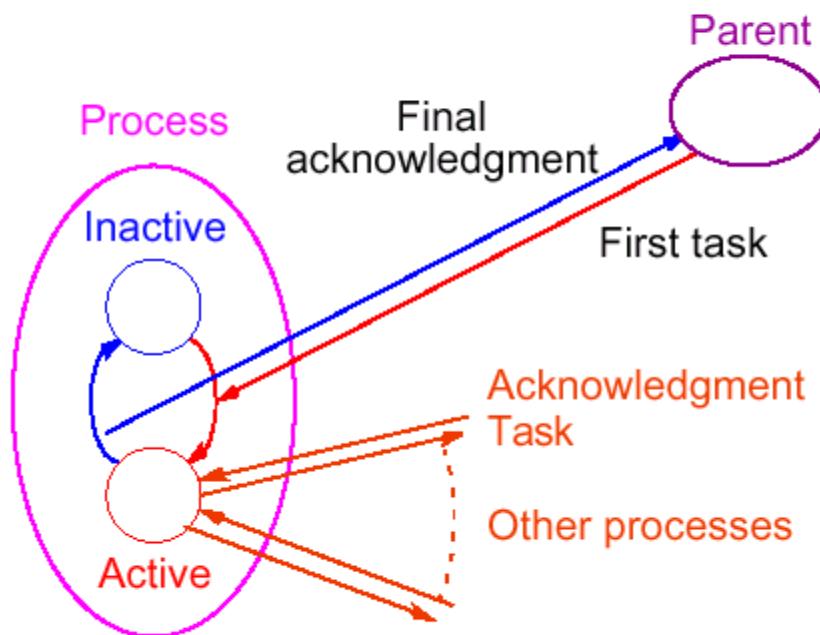
Process that sent task to make it enter the active state becomes its “parent.”

When process receives a task, it immediately sends an acknowledgment message, **except if the process it receives the task from is its parent process**. Only sends an acknowledgment message to its parent when it is ready to become inactive, i.e. when

- Its local termination condition exists (all tasks are completed, *and*)
- It has transmitted all its acknowledgments for tasks it has received, *and*
- It has received all its acknowledgments for tasks it has sent out.

A process must become inactive before its parent process. When first process becomes idle, the computation can terminate.

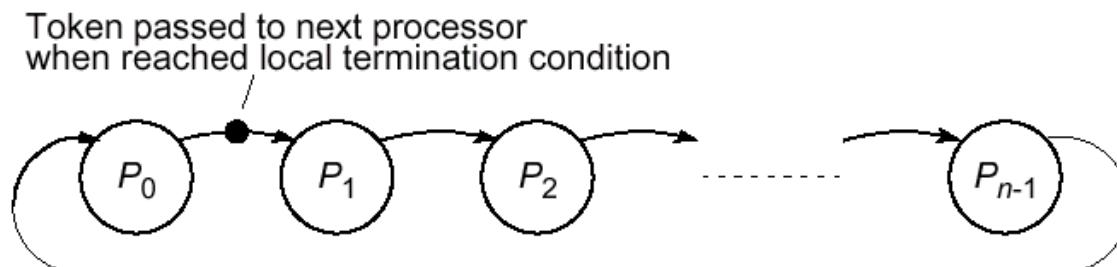
Termination using message acknowledgments



Ring Termination Algorithms

Single-pass ring termination algorithm

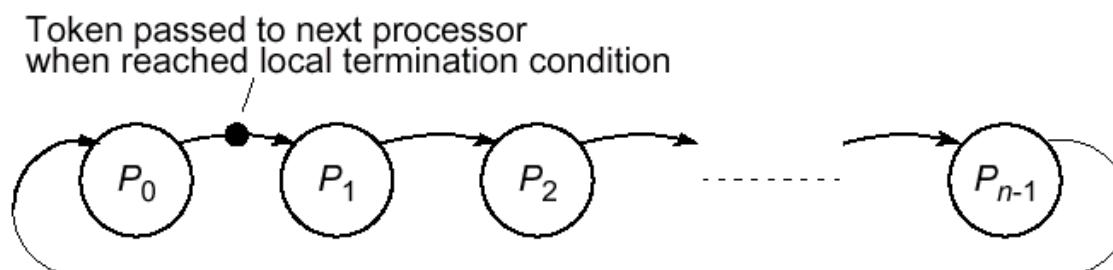
1. When P_0 has terminated, it generates a token that is passed to P_1 .
2. When P_i ($1 \leq i < n$) receives the token and has already terminated, it passes the token onward to P_{i+1} . Otherwise, it waits for its local termination condition and then passes the token onward. P_{n-1} passes the token to P_0 .
3. When P_0 receives a token, it knows that all processes in the ring have terminated. A message can then be sent to all processes informing them of global termination, if necessary.



Ring Termination Algorithms

Single-pass ring termination algorithm

1. When P_0 has terminated, it generates a token that is passed to P_1 .
2. When P_i ($1 \leq i < n$) receives the token and has already terminated, it passes the token onward to P_{i+1} . Otherwise, it waits for its local termination condition and then passes the token onward. P_{n-1} passes the token to P_0 .
3. When P_0 receives a token, it knows that all processes in the ring have terminated. A message can then be sent to all processes informing them of global termination, if necessary.



The algorithm assumes that a process cannot be reactivated after reaching its local termination condition. Does not apply to work pool problems in which a process can pass a new task to an idle process

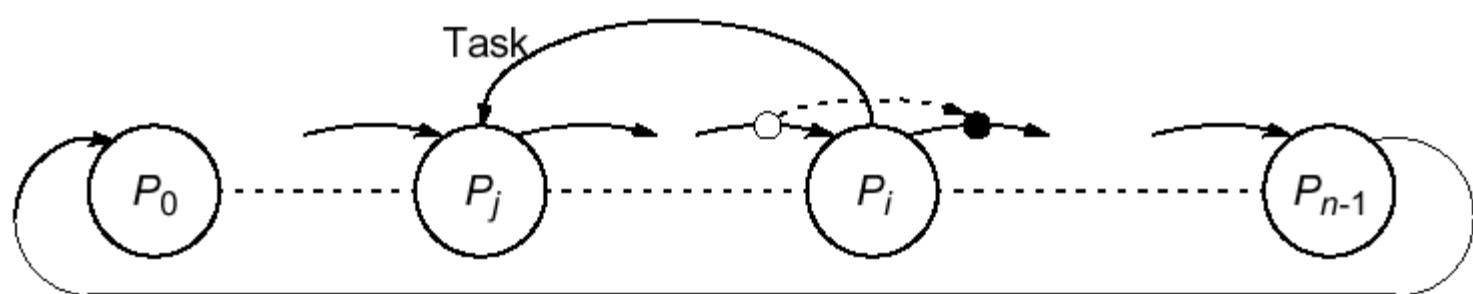
Dual-Pass Ring Termination Algorithm

Can handle processes being reactivated but requires two passes around the ring. The reason for reactivation is for process P_i , to pass a task to P_j where $j < i$ and after a token has passed P_j . If this occurs, the token must recirculate through the ring a second time.

To differentiate these circumstances, tokens colored white or black. Processes are also colored white or black.

Receiving a black token means that global termination may not have occurred and token must be recirculated around ring again.

Passing task to previous processes

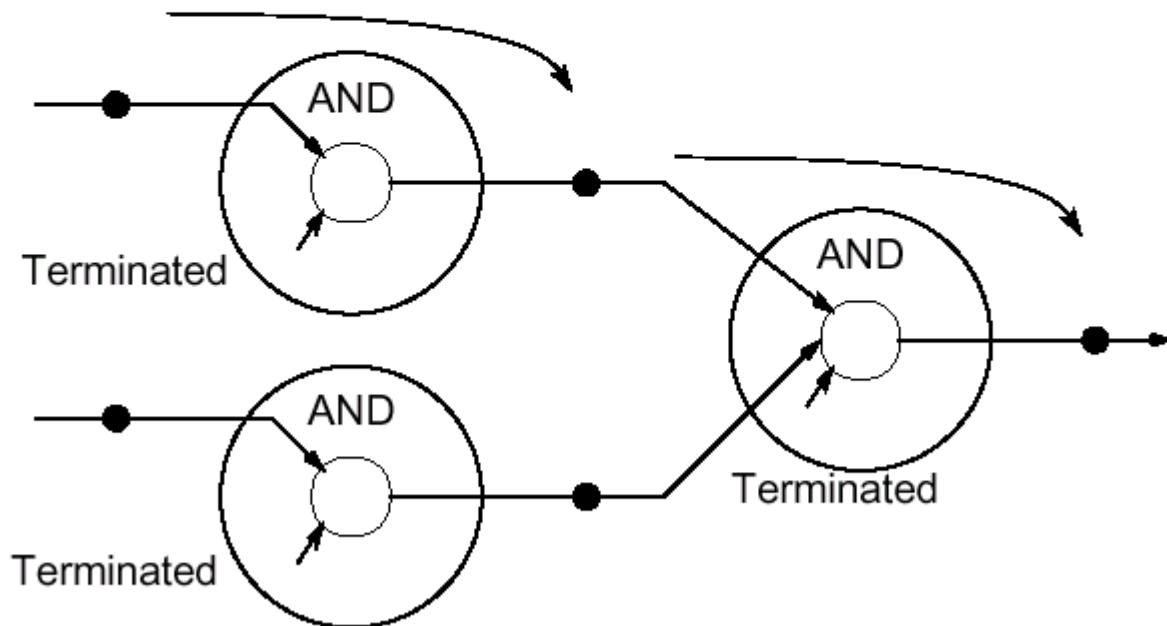


The algorithm is as follows, again starting at P_0 :

1. P_0 becomes white when it has terminated and generates a white token to P_1 .
2. The token is passed through the ring from one process to the next when each process has terminated, but the color of the token may be changed. If P_i passes a task to P_j where $j < i$ (that is, before this process in the ring), it becomes a *black process*; otherwise it is a *white process*. A black process will color a token black and pass it on. A white process will pass on the token in its original color (either black or white). After P_i has passed on a token, it becomes a white process. P_{n-1} passes the token to P_0 .
3. When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

Tree Algorithm

Local actions described can be applied to various structures, notably a tree structure, to indicate that processes up to that point have terminated.



Load balancing/termination detection Example

Shortest Path Problem

Finding the shortest distance between two points on a graph.
It can be stated as follows:

Given a set of interconnected nodes where the links between the nodes are marked with “weights,” find the path from one specific node to another specific node that has the smallest accumulated weights.

The interconnected nodes can be described by a *graph*.

The nodes are called *vertices*, and the links are called *edges*.

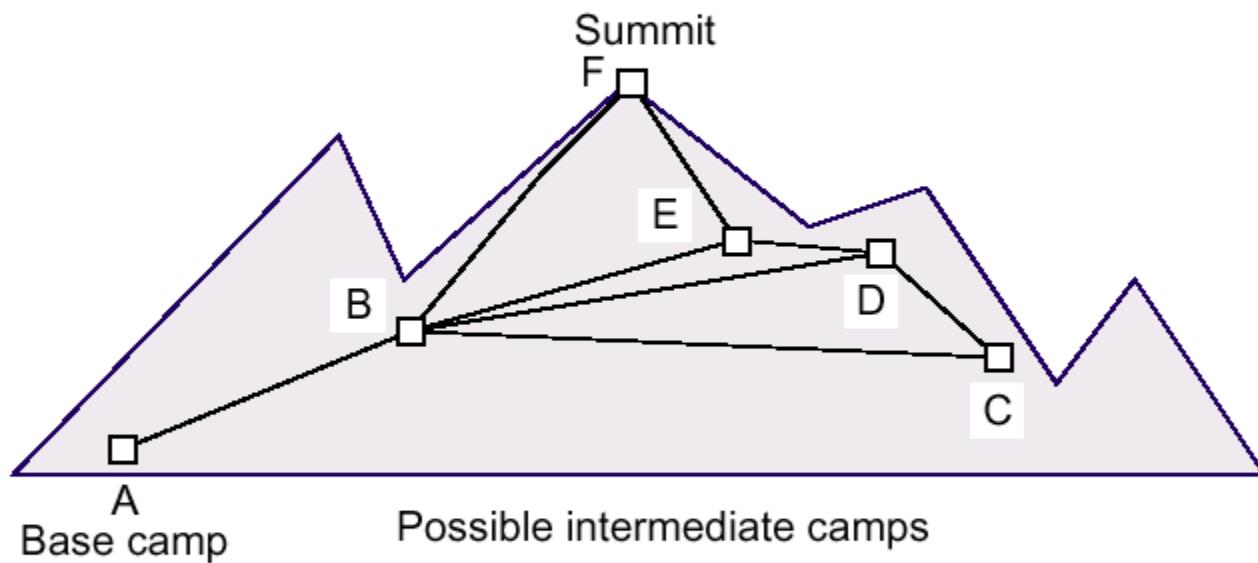
If the edges have implied directions (that is, an edge can only be traversed in one direction, the graph is a *directed graph*.

Graph could be used to find solution to many different problems; eg:

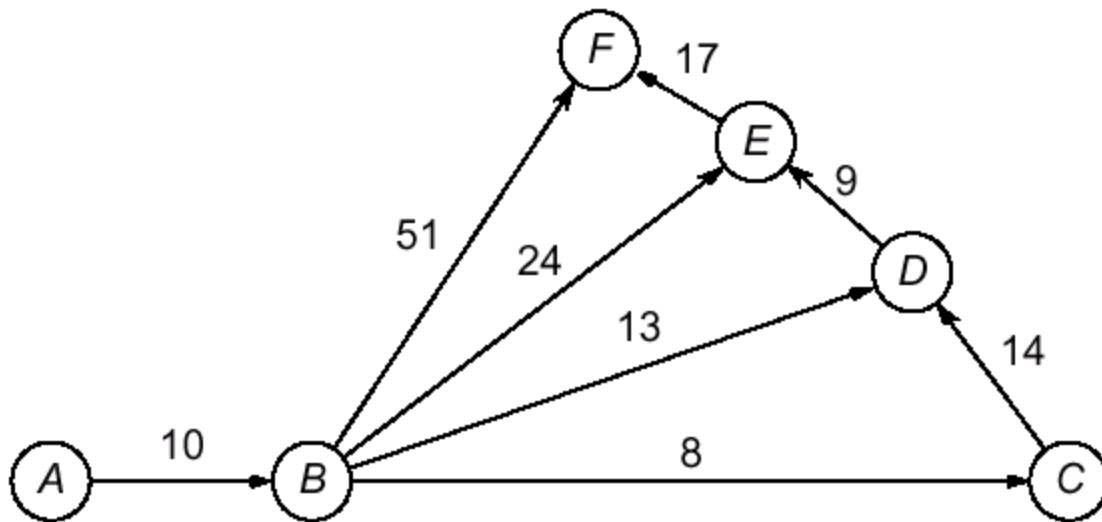
1. The shortest distance between two towns or other points on a map, where the weights represent distance
2. The quickest route to travel, where the weights represent time (the quickest route may not be the shortest route if different modes of travel are available; for example, flying to certain towns)
3. The least expensive way to travel by air, where the weights represent the cost of the flights between cities (the vertices)
4. The best way to climb a mountain given a terrain map with contours
5. The best route through a computer network for minimum message delay (the vertices represent computers, and the weights represent the delay between two computers)
6. The most efficient manufacturing system, where the weights represent hours of work

“The best way to climb a mountain” will be used as an example.

Example: The Best Way to Climb a Mountain



Graph of mountain climb



Weights in graph indicate amount of effort that would be expended in traversing the route between two connected camp sites.

The effort in one direction may be different from the effort in the opposite direction (downhill instead of uphill!). (*directed graph*)

Graph Representation

Two basic ways that a graph can be represented in a program:

1. **Adjacency matrix** — a two-dimensional array, a , in which $a[i][j]$ holds the weight associated with the edge between vertex i and vertex j if one exists
2. **Adjacency list** — for each vertex, a list of vertices directly connected to the vertex by an edge and the corresponding weights associated with the edges

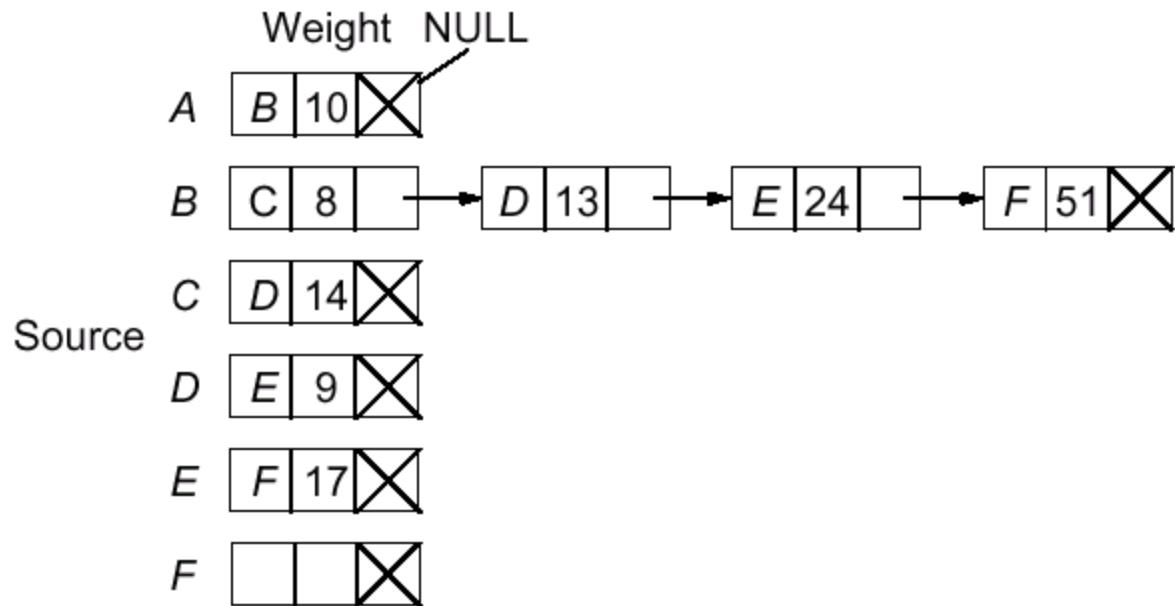
Adjacency matrix used for dense graphs. The adjacency list is used for sparse graphs.

Difference based upon space (storage) requirements. Accessing the adjacency list is slower than accessing the adjacency matrix.

Representing the graph

		Destination					
		A	B	C	D	E	F
Source	A	•	10	•	•	•	•
	B	•	•	8	13	24	51
	C	•	•	•	14	•	•
	D	•	•	•	•	9	•
	E	•	•	•	•	•	17
	F	•	•	•	•	•	•

(a) Adjacency matrix



Searching a Graph

Two well-known single-source shortest-path algorithms:

- Moore's single-source shortest-path algorithm (Moore, 1957)
- Dijkstra's single-source shortest-path algorithm (Dijkstra, 1959)

which are similar.

Moore's algorithm is chosen because it is more amenable to parallel implementation although it may do more work.

The weights must all be positive values for the algorithm to work.

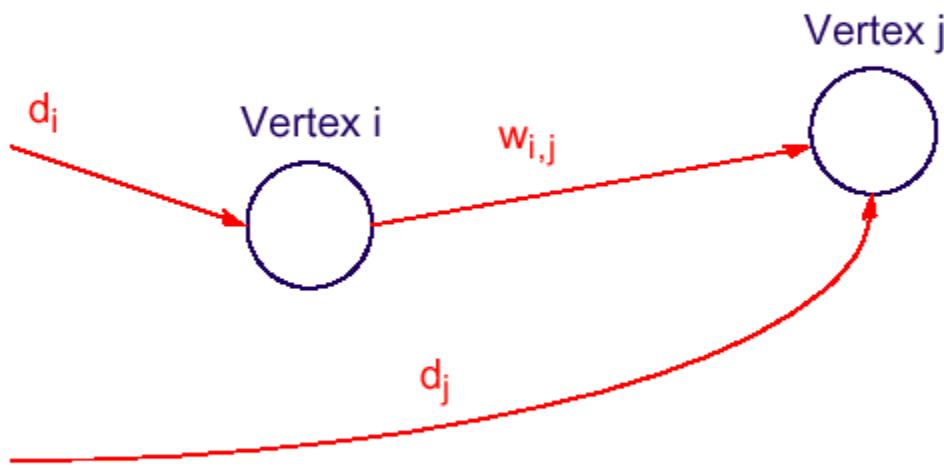
Moore's Algorithm

Starting with the source vertex, the basic algorithm implemented when vertex i is being considered as follows.

Find the distance to vertex j through vertex i and compare with the current minimum distance to vertex j . Change the minimum distance if the distance through vertex i is shorter. If d_i is the current minimum distance from the source vertex to vertex i and $w_{i,j}$ is the weight of the edge from vertex i to vertex j :

$$d_j = \min(d_j, d_i + w_{i,j})$$

Moore's Shortest-path Algorithm



Date Structures

First-in-first-out vertex queue created to hold a list of vertices to examine. Initially, only source vertex is in queue.

Current shortest distance from source vertex to vertex i stored in array `dist[i]`. At first, none of these distances known and array elements are initialized to infinity.

Suppose $w[i][j]$ holds the weight of the edge from vertex i and vertex j (infinity if no edge). The code could be of the form

```
newdist_j = dist[i] + w[i][j];  
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

When a shorter distance is found to vertex j , vertex j is added to the queue (if not already in the queue), which will cause vertex j to be examined again - **Important aspect of this algorithm, which is not present in Dijkstra's algorithm.**

Example

The initial values of the two key data structures are

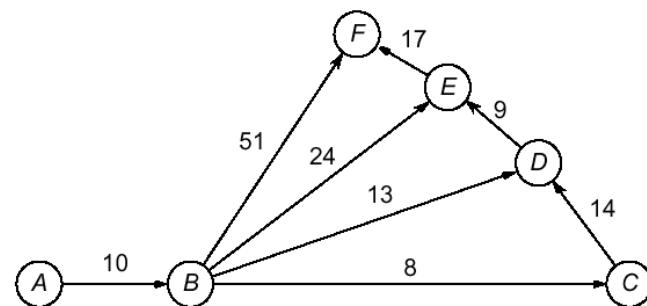
Vertices to consider						
A						

vertex_queue

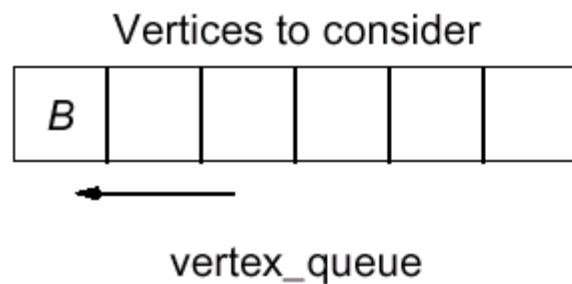
Current minimum distances						
0	•	•	•	•	•	•

vertex A B C D E F

dist[]



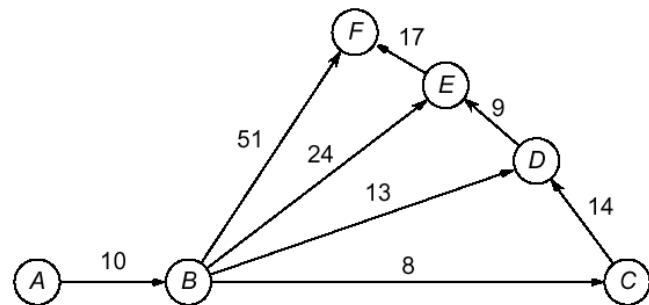
After examining A to



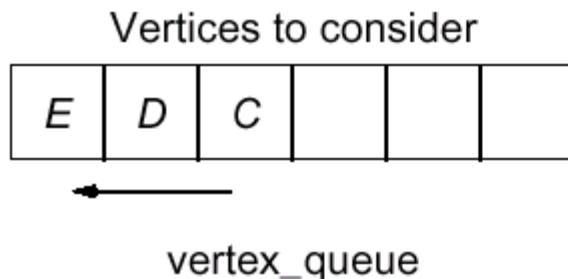
Current minimum distances

0	10	•	•	•	•	
vertex	A	B	C	D	E	F

dist[]



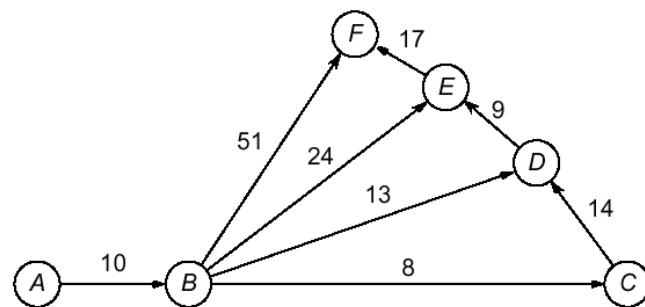
After examining B to F, E, D , and C :



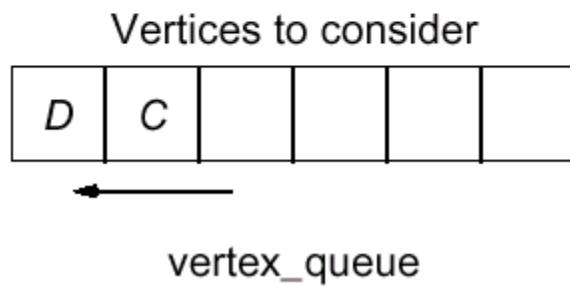
Current minimum distances

0	10	18	23	34	61
vertex A	B	C	D	E	F

dist[]

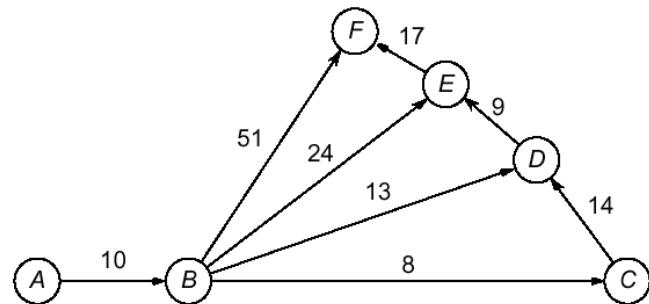


After examining E to F



Current minimum distances

vertex	A	B	C	D	E	F
dist[]	0	10	18	23	34	50

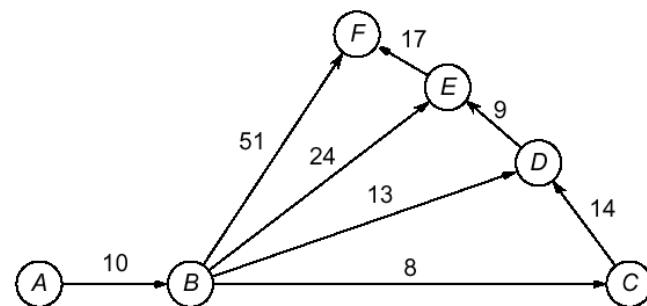


After examining D to E :

Vertices to consider					
C	E				

vertex_queue

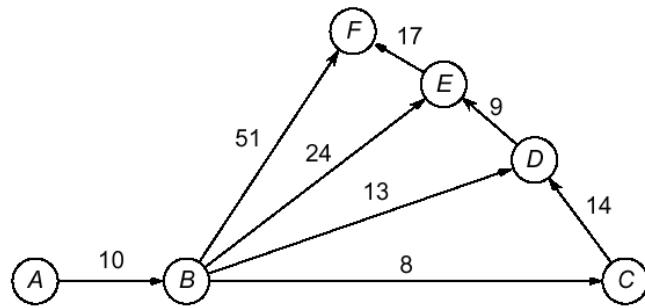
Current minimum distances						
vertex	A	B	C	D	E	F
dist[]	0	10	18	23	32	50



After examining C to D : No changes.

After examining E (again) to F :

Vertices to consider						Current minimum distances							
vertex	A	B	C	D	E	F	dist[]	0	10	18	23	32	49
vertex_queue													



No more vertices to consider. We have the minimum distance from vertex A to each of the other vertices, including the destination vertex, F .

Parallel Implementations

Centralized Work Pool

Centralized work pool holds vertex queue, `vertex_queue []` as tasks.

Each slave takes vertices from the vertex queue and returns new vertices.

Since the structure holding the graph weights is fixed, this structure could be copied into each slave, say a copied adjacency matrix.

Master

```
while (vertex_queue() != empty) {
    recv(PANY, source = Pi);      /* request task from slave */
    v = get_vertex_queue();
    send(&v, Pi);                  /* send next vertex and */
    send(&dist, &n, Pi);          /* current dist array */

    .
    recv(&j, &dist[j], PANY, source = Pi);/* new distance */
    append_queue(j, dist[j]);   /* append vertex to queue */
                                /* and update distance array */
};

recv(PANY, source = Pi);      /* request task from slave */
send(Pi, termination_tag); /* termination message*/
```

Slave (process i)

```
send(Pmaster);           /* send request for task */
recv(&v, Pmaster, tag);    /* get vertex number */
if (tag != termination_tag) {
    recv(&dist, &n, Pmaster); /* and dist array */
    for (j = 1; j < n; j++) /* get next edge */
        if (w[v][j] != infinity) /* if an edge */
            newdist_j = dist[v] + w[v][j];
            if (newdist_j < dist[j]) {
                dist[j] = newdist_j;
                send(&j, &dist[j], Pmaster); /* add vertex to queue */
                                         /* send updated distance */
            }
}
}
```

Decentralized Work Pool

Convenient approach is to assign slave process i to search around vertex i only and for it to have the vertex queue entry for vertex i if this exists in the queue.

The array `dist []` will also be distributed among the processes so that process i maintains the current minimum distance to vertex i .

Process also stores an adjacency matrix/list for vertex i , for the purpose of identifying the edges from vertex i .

Search Algorithm

Vertex A is the first vertex to search. The process assigned to vertex A is activated.

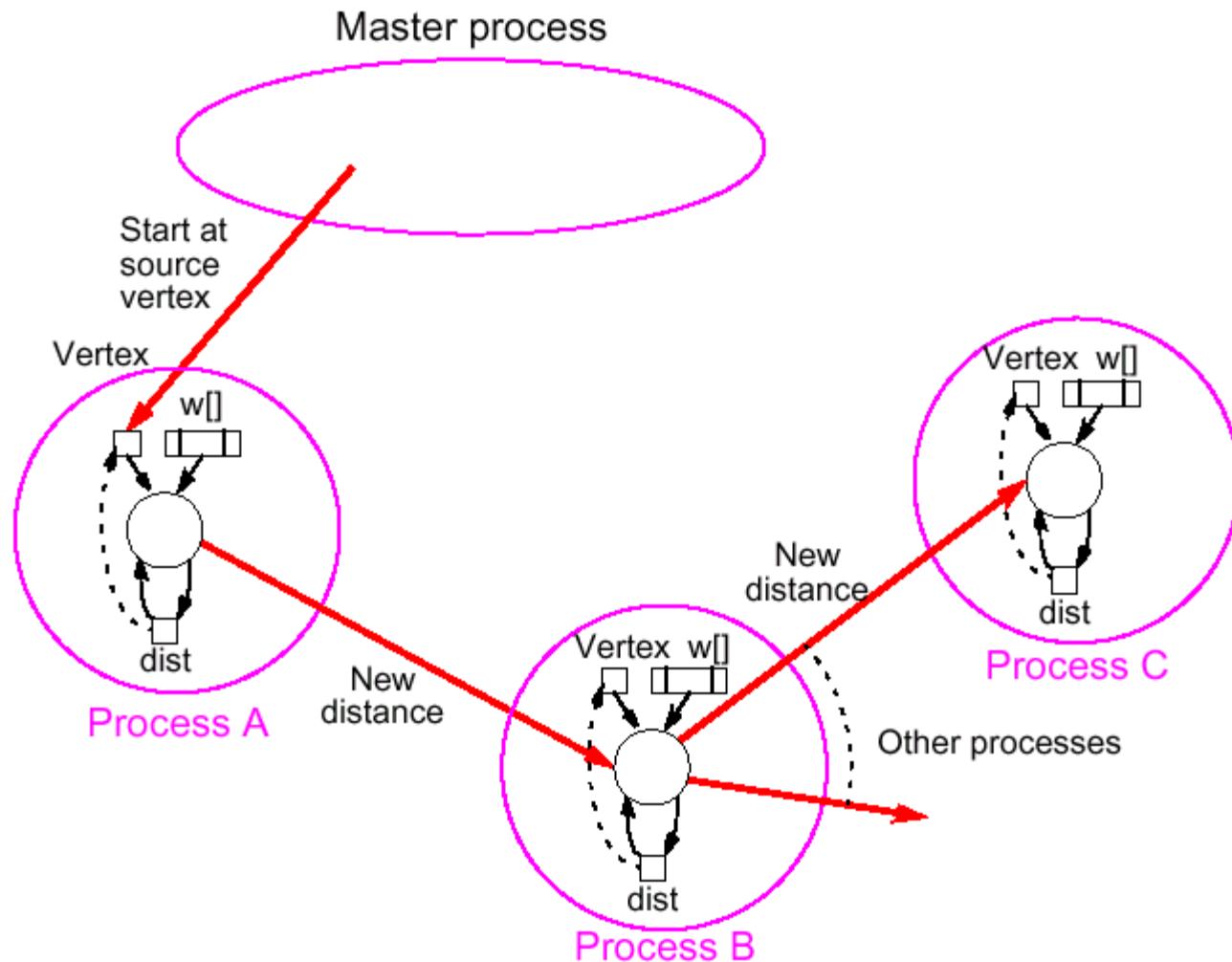
This process will search around its vertex to find distances to connected vertices.

Distance to process j will be sent to process j for it to compare with its currently stored value and replace if the currently stored value is larger.

In this fashion, all minimum distances will be updated during the search.

If the contents of $d[i]$ changes, process i will be reactivated to search again.

Distributed graph search



Slave (process i)

```
recv(newdist, PANY);
if (newdist < dist) {
    dist = newdist;
    vertex_queue = TRUE;           /* add to queue */
} else vertex_queue == FALSE;
if (vertex_queue == TRUE)/*start searching around vertex*/
    for (j = 1; j < n; j++)      /* get next edge */
        if (w[j] != infinity) {
            d = dist + w[j];
            send(&d, Pj);       /* send distance to proc j */
        }
```

Algorithms and Applications

Parallel Algorithm Time Complexity

Can derive the time complexity of a parallel algorithm in a similar manner as for a sequential algorithm by counting the steps in the algorithm (worst case) .

Following from the definition of cost-optimal algorithm

(Cost) optimal parallel time complexity = $\frac{\text{sequential time complexity}}{\text{number of processors}}$

- Sorting Algorithms
- Numerical Algorithms
- Image Processing
- Searching and Optimization

Sorting Algorithms

- rearranging a list of numbers into increasing (strictly non-decreasing) order.

Potential Speedup

$O(n \log n)$ optimal for any sequential sorting algorithm without using special properties of the numbers.

Best we can expect based upon a sequential sorting algorithm using n processors is

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

Rank Sort

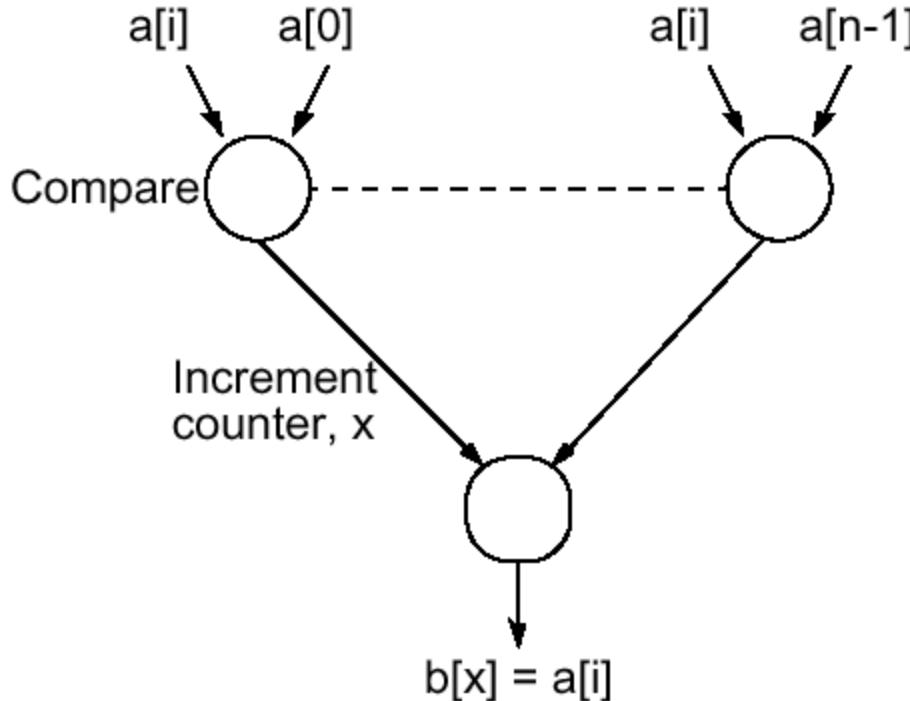
The number of numbers that are smaller than each selected number is counted. This count provides the position of selected number in sorted list; that is, its “rank.”

First $a[0]$ is read and compared with each of the other numbers, $a[1] \dots a[n-1]$, recording the number of numbers less than $a[0]$. Suppose this number is x . This is the index of the location in the final sorted list. The number $a[0]$ is copied into the final sorted list $b[0] \dots b[n-1]$, at location $b[x]$. Actions repeated with the other numbers.

Overall sequential sorting time complexity of $O(n^2)$ (not exactly a good sequential sorting algorithm!).

Using n^2 Processors

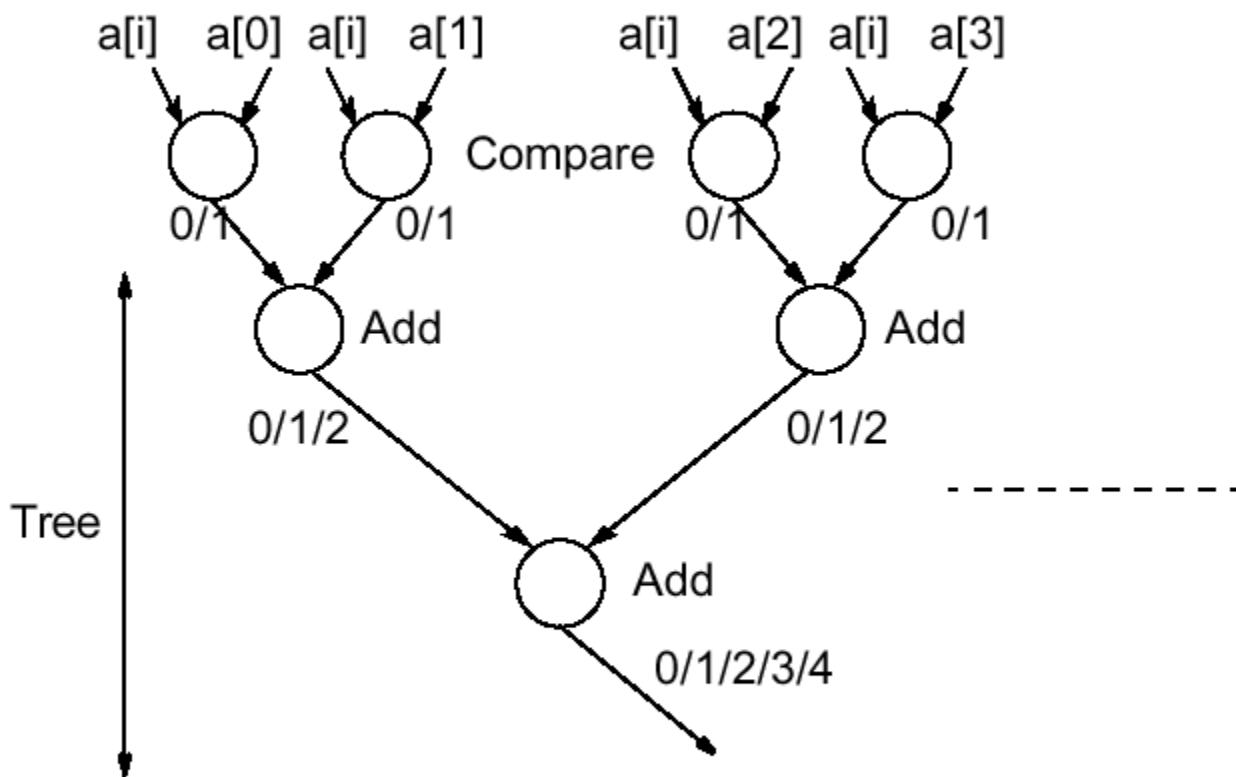
Comparing one number with the other numbers in list using multiple processors:



$n - 1$ processors used to find rank of one number. With n numbers, $(n - 1)n$ processors or (almost) n^2 processors needed. Incrementing the counter done sequentially and requires maximum of n steps.

Reduction in Number of Steps

Tree to reduce number of steps involved in incrementing counter:



$O(\log n)$ algorithm with n^2 processors.

Processor efficiency relatively low.

Compare-and-Exchange Sorting Algorithms

Compare and Exchange

Form the basis of several, if not most, classical sequential sorting algorithms.

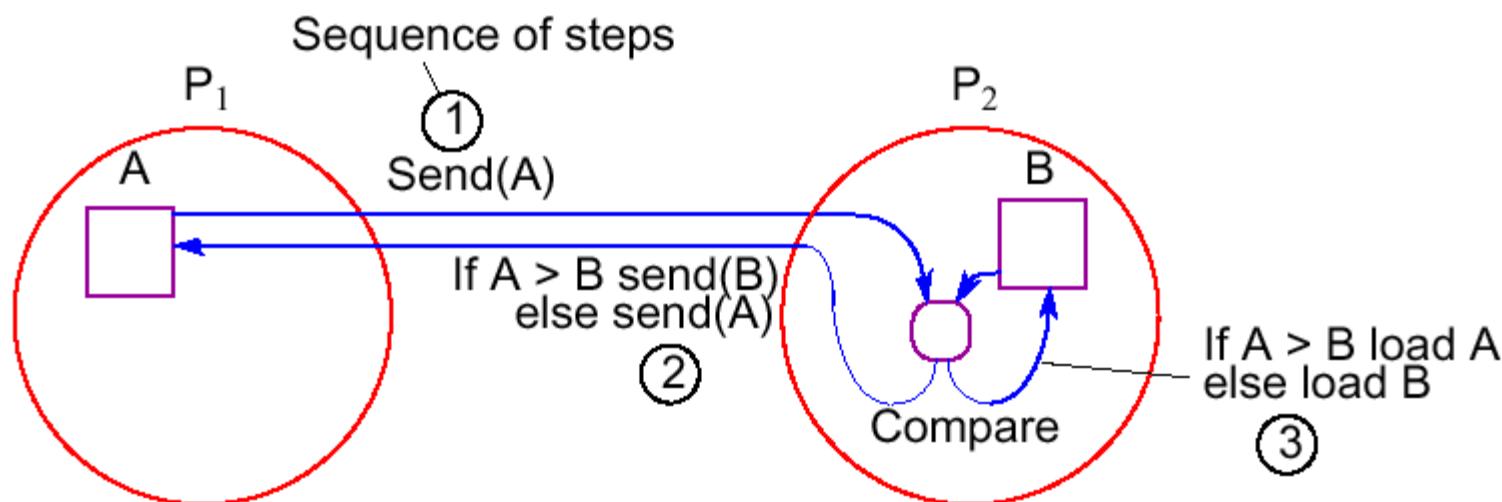
Two numbers, say A and B , are compared. If $A > B$, A and B are exchanged, i.e.:

```
if (A > B) {  
    temp = A;  
    A = B;  
    B = temp;  
}
```

Message-Passing Compare and Exchange

Version 1

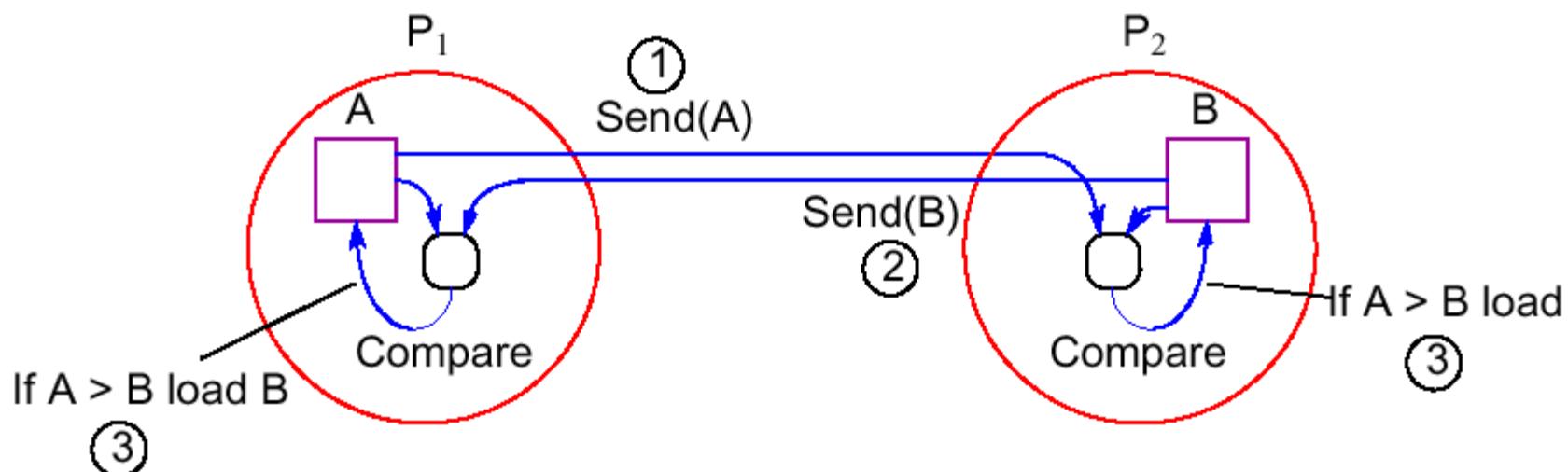
P_1 sends A to P_2 , which compares A and B and sends back B to P_1 if A is larger than B (otherwise it sends back A to P_1):



Alternative Message Passing Method

Version 2

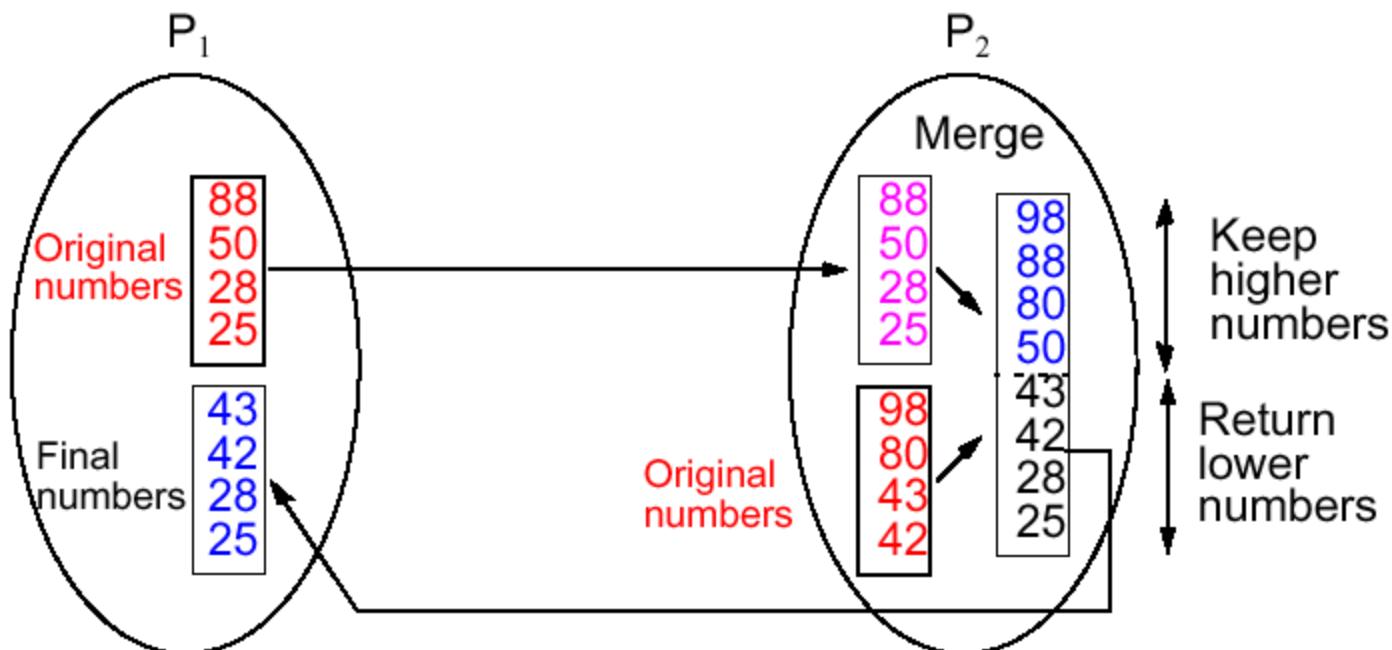
For P_1 to send A to P_2 and P_2 to send B to P_1 . Then both processes perform compare operations. P_1 keeps the larger of A and B and P_2 keeps the smaller of A and B :



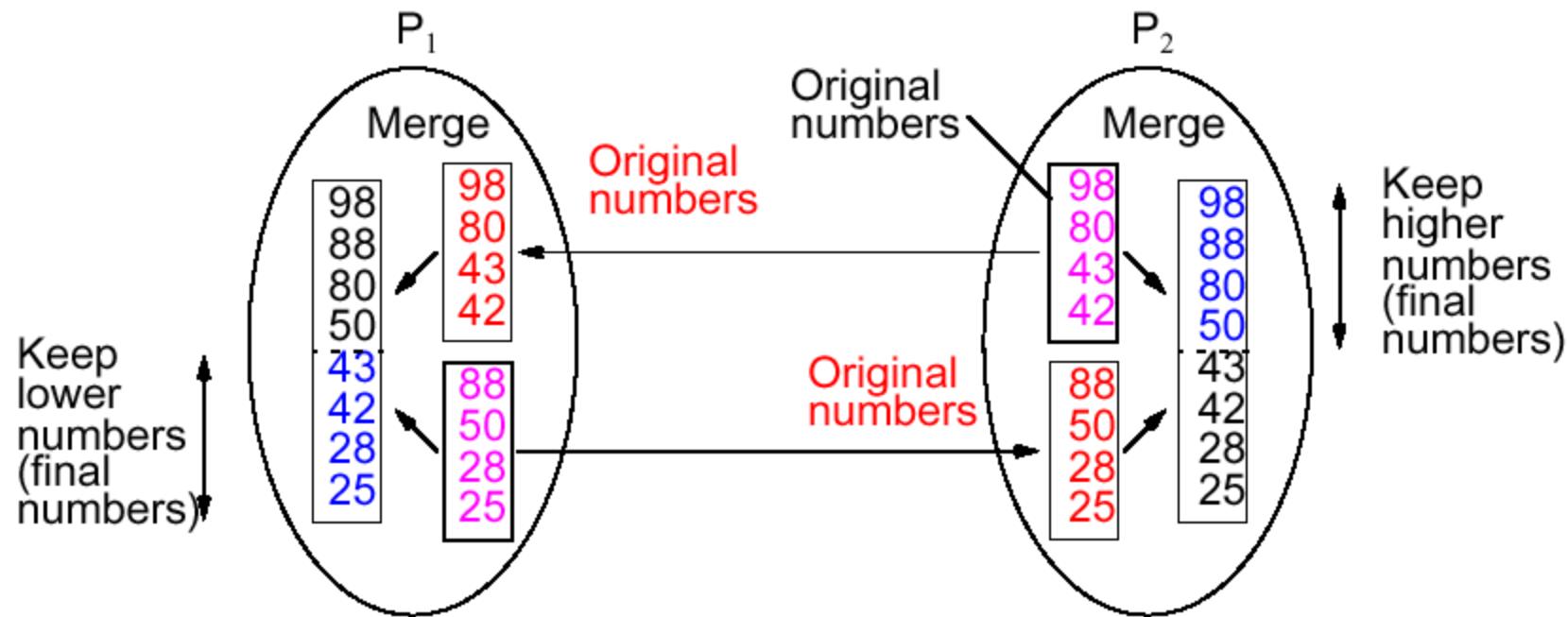
Data Partitioning

(Version 1)

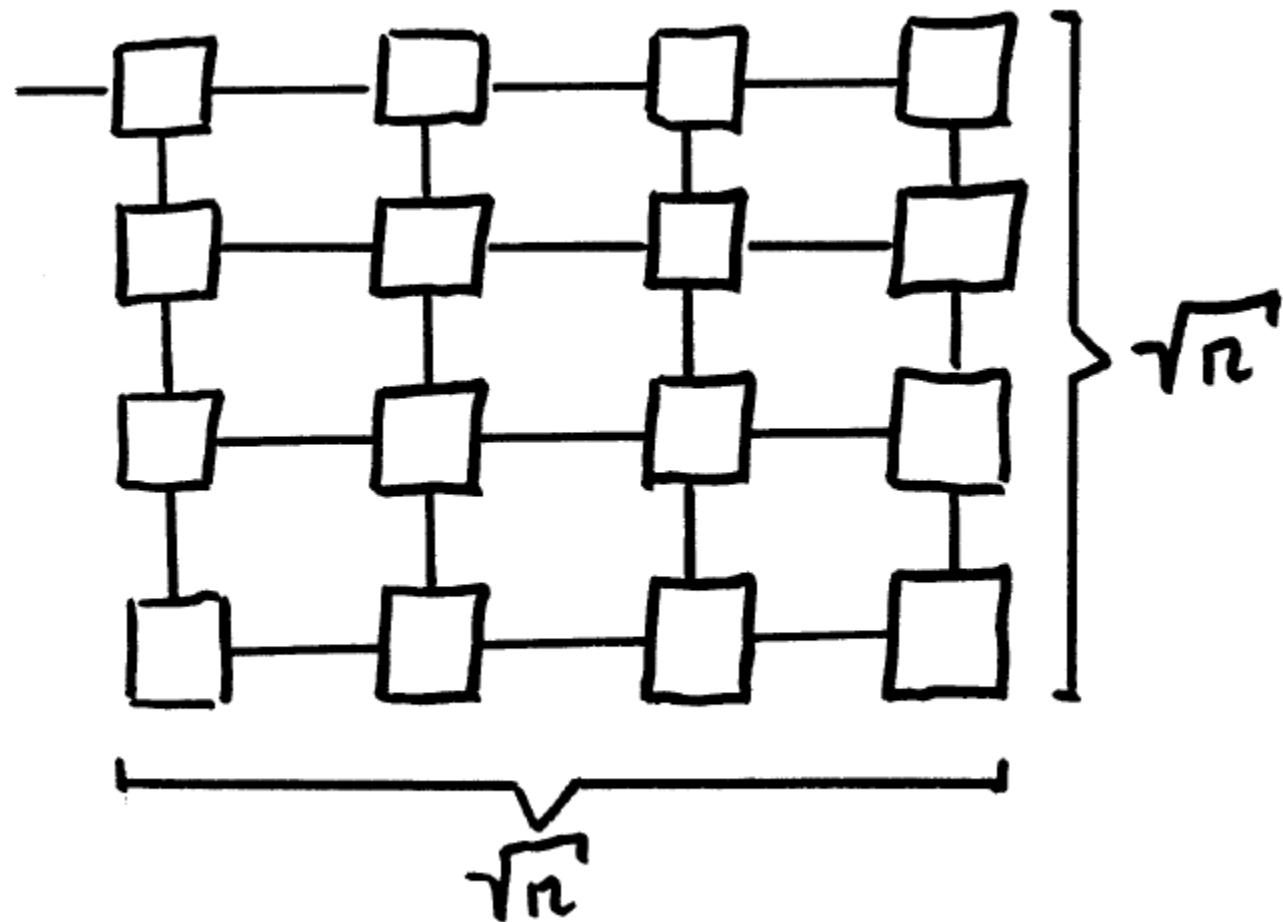
p processors and n numbers. n/p numbers assigned to each processor:



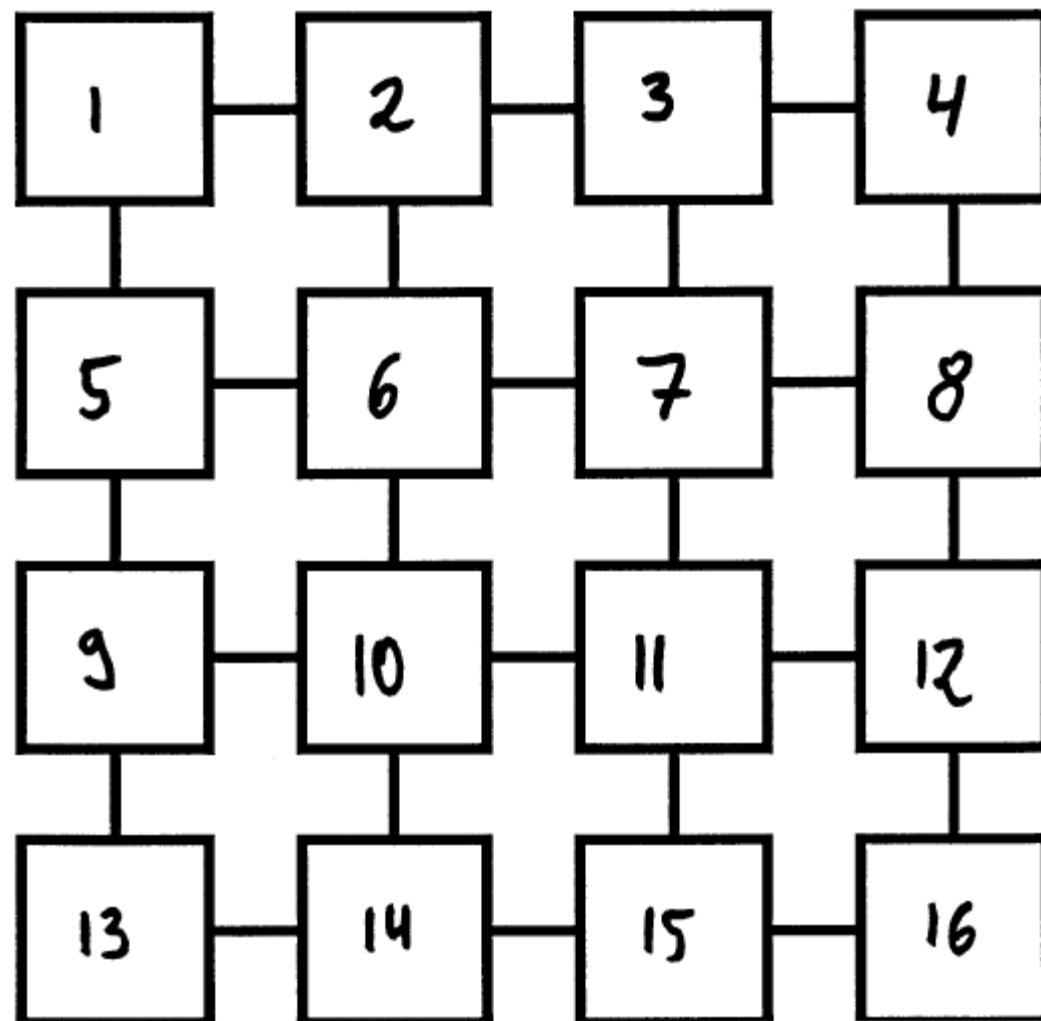
Merging Two Sublists — Version 2



Array Sort

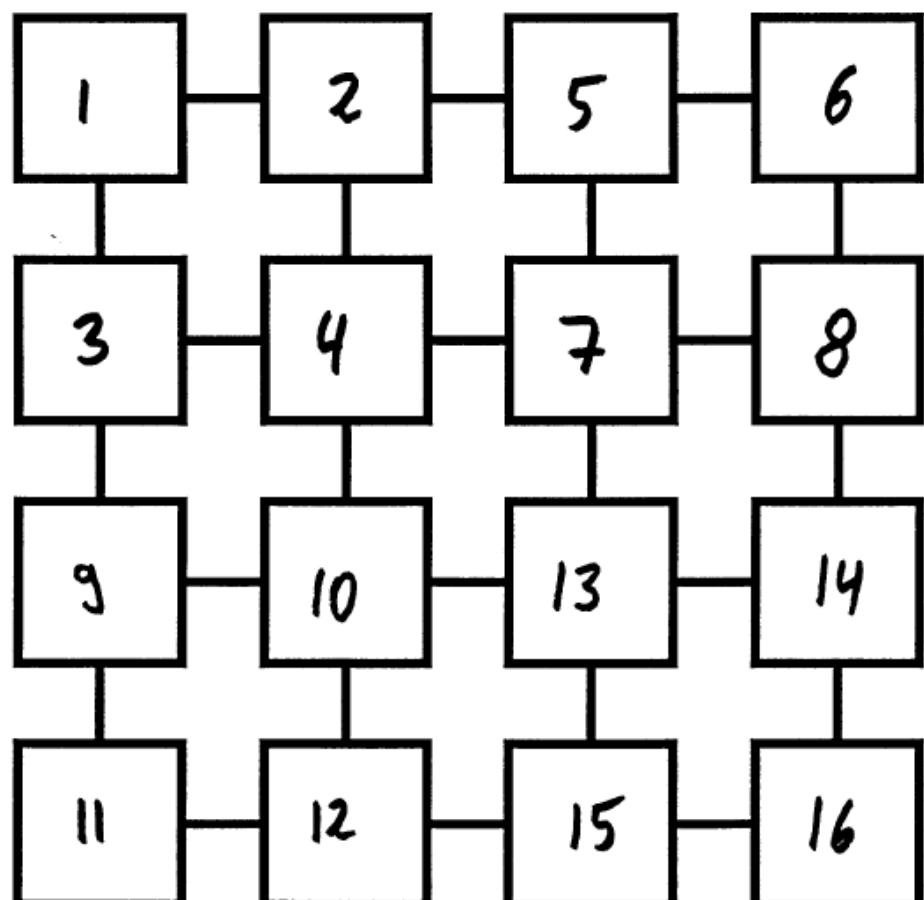


ROW MAJOR

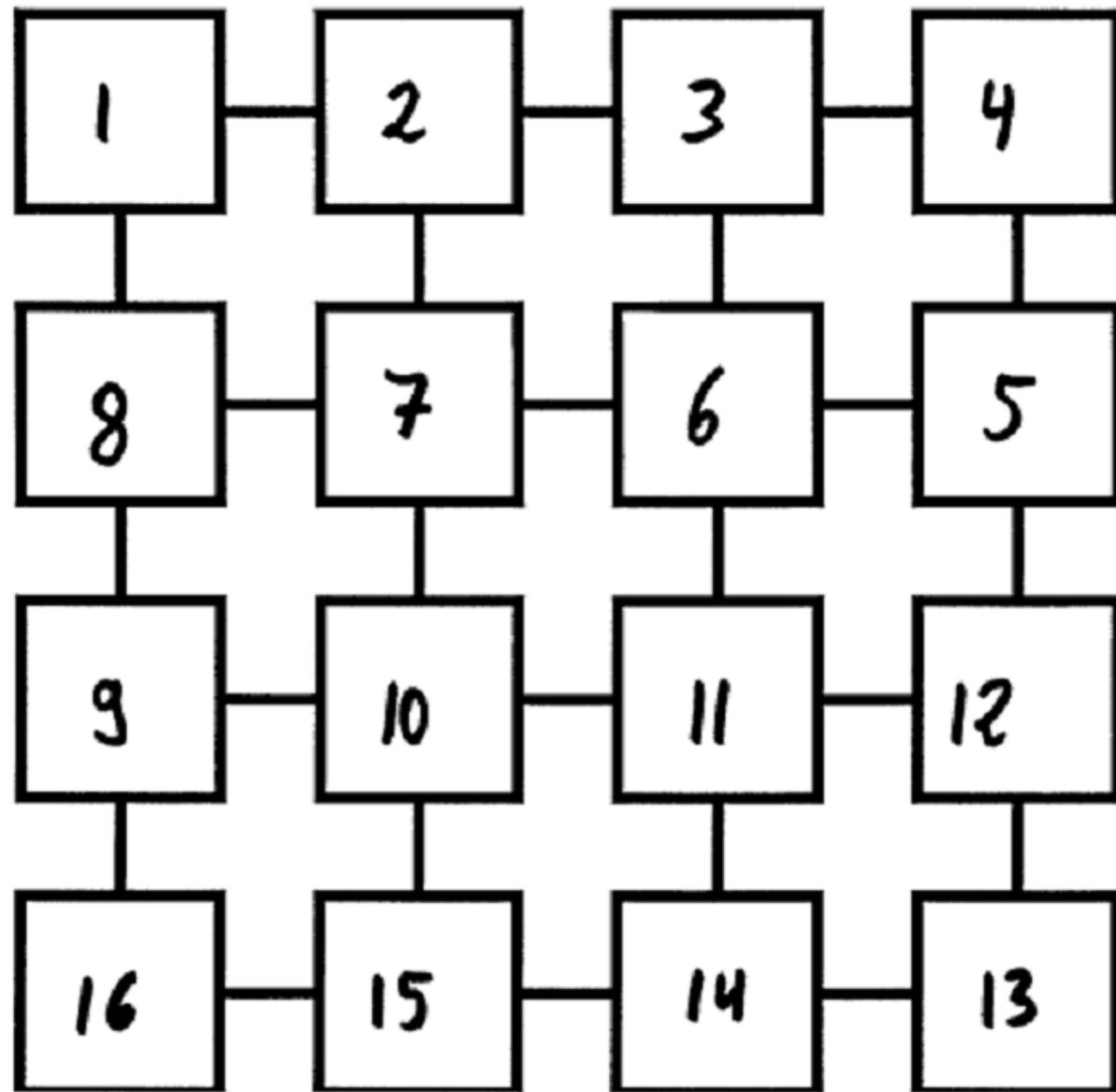


SHUFFLED ROW MAJOR

1	2
3	4



SNAKE - LIKE

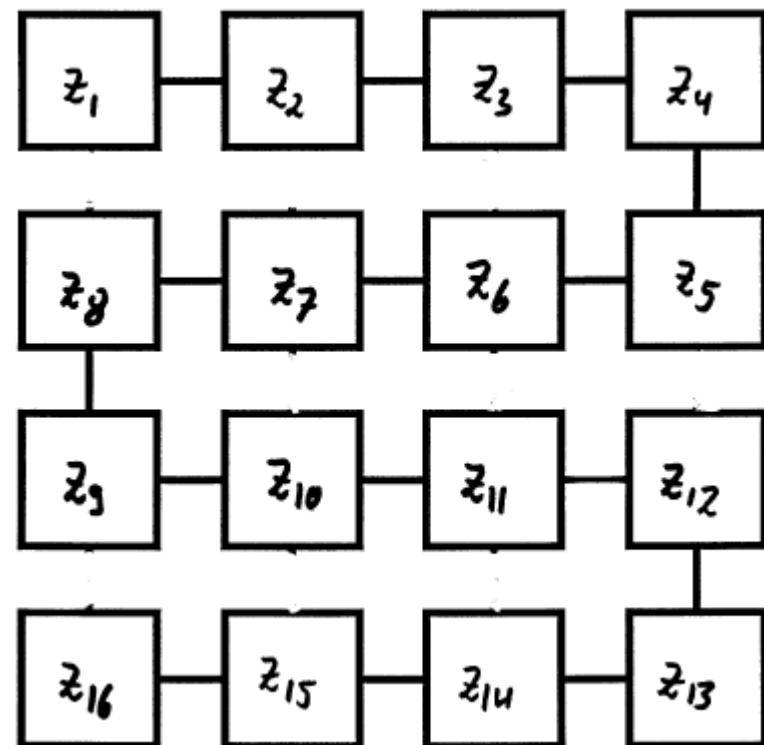


A $\sqrt{n} \times \sqrt{n'}$ PROCESSOR ARRAY
CAN BE SORTED IN TIME $O(\sqrt{n'})$.

ODD-EVEN TRANSPORTATION SORT:

[ODD: COMPARE/EXCHANGE
 z_i, z_{i+1} FOR ALL ODD i
EVEN: COMPARE/EXCHANGE
 z_i, z_{i+1} FOR ALL EVEN i]

$\frac{n}{2}$ TIMES



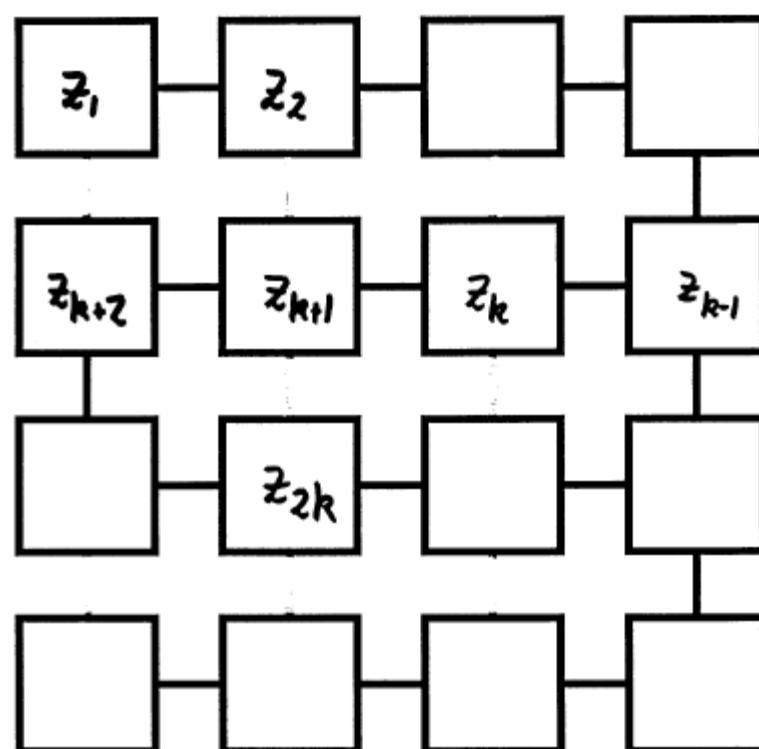
odd	6-5	2-3	4-1			
even	5	6-2	3-1	4		
odd	5-2	6-1	3	4		
even	2	5-1	6-3	4		
odd	2-1	5	3	6		
even	1	2-3	5-4	6		
	1	2	3	4	5	6

SHUFFLE :

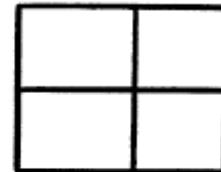
$z_1 z_2 \dots z_k z_{k+1} \dots z_{2k}$

$\rightarrow z_1 z_{k+1} z_2 z_{k+2} \dots z_k z_{2k}$

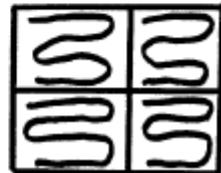
time : $O(2k)$



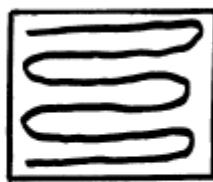
1) SPLIT ARRAY
INTO 4 SUBARRAYS
OF EQUAL SIZE

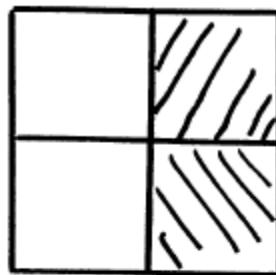


2) RECURSIVELY,
IN PARALLEL SORT
ALL SUBARRAYS



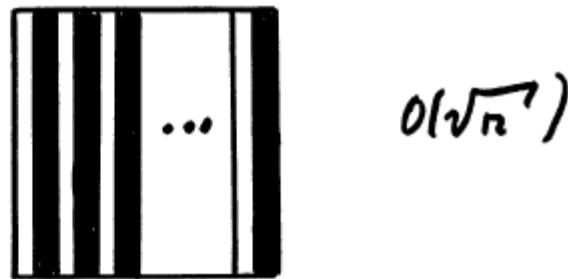
3) MERGE SORTED
SUBSEQUENCES
INTO ONE SORTED
SEQUENCE



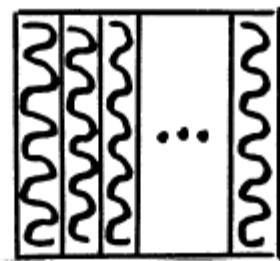


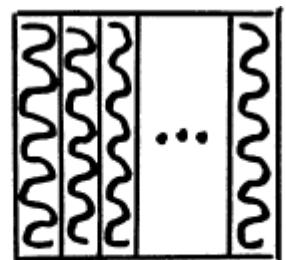
3A) SHUFFLE EACH ROW
(INTERCHANGE COLUMNS)



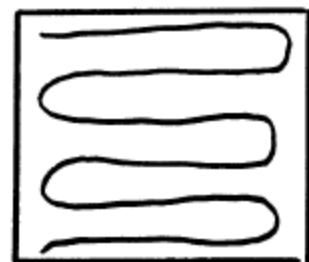


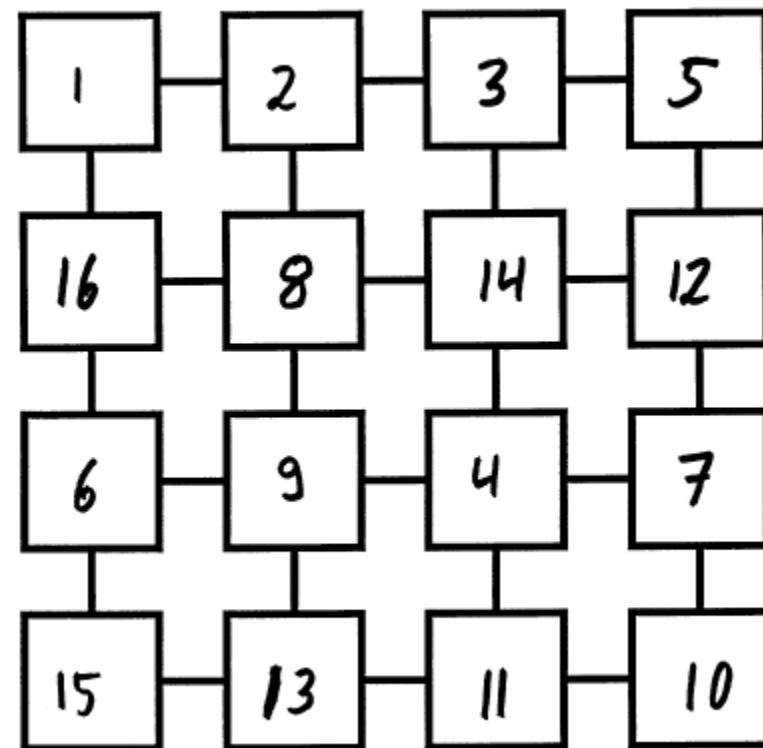
3B) SORT ALL DOUBLE COLUMNS
USING $2\sqrt{n}^2$ STEPS OF ODD-EVEN
TRANSPORTATION SORT





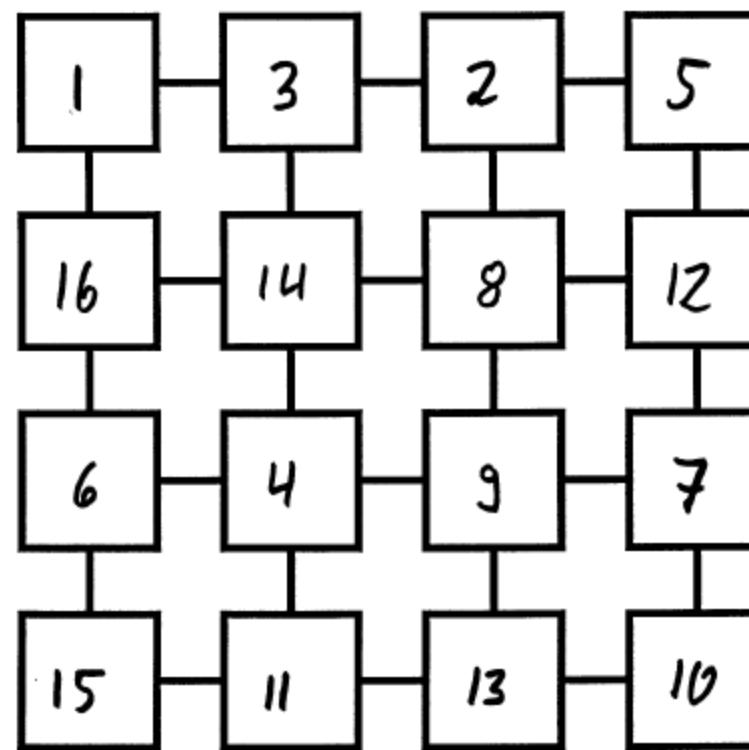
3C) APPLY $2\sqrt{n}$ STEPS OF
ODD-EVEN TRANSPORTATION
SORT TO THE WHOLE ARRAY



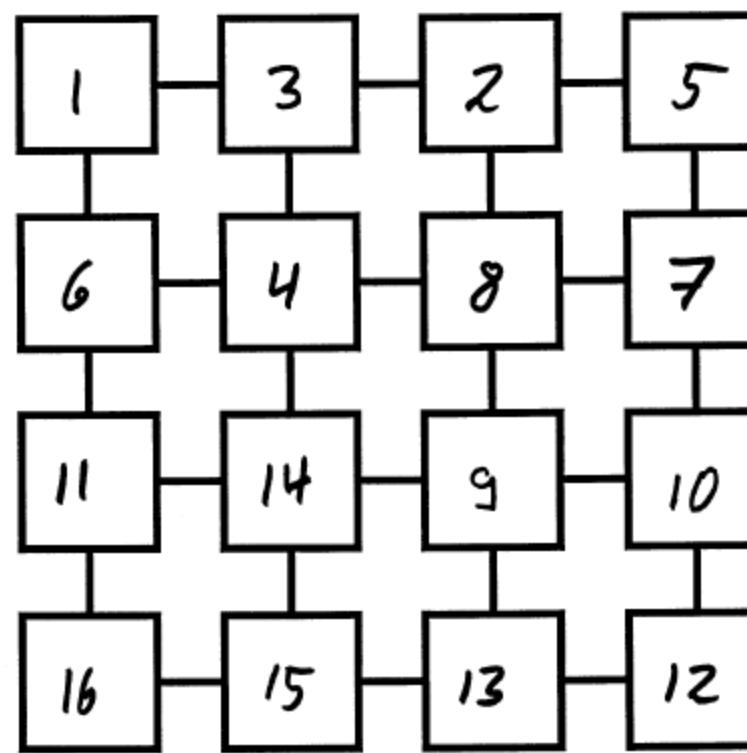


1

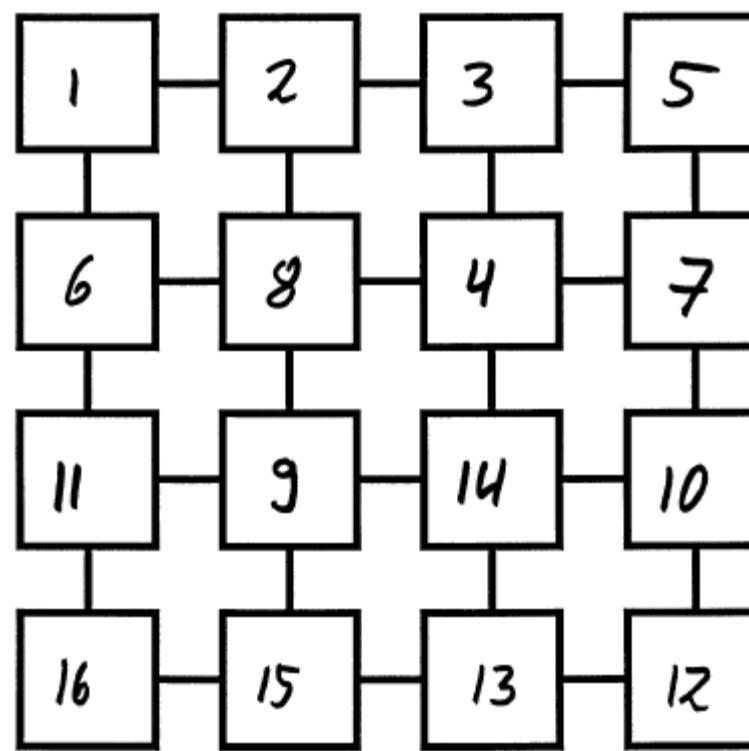
$\Downarrow A$



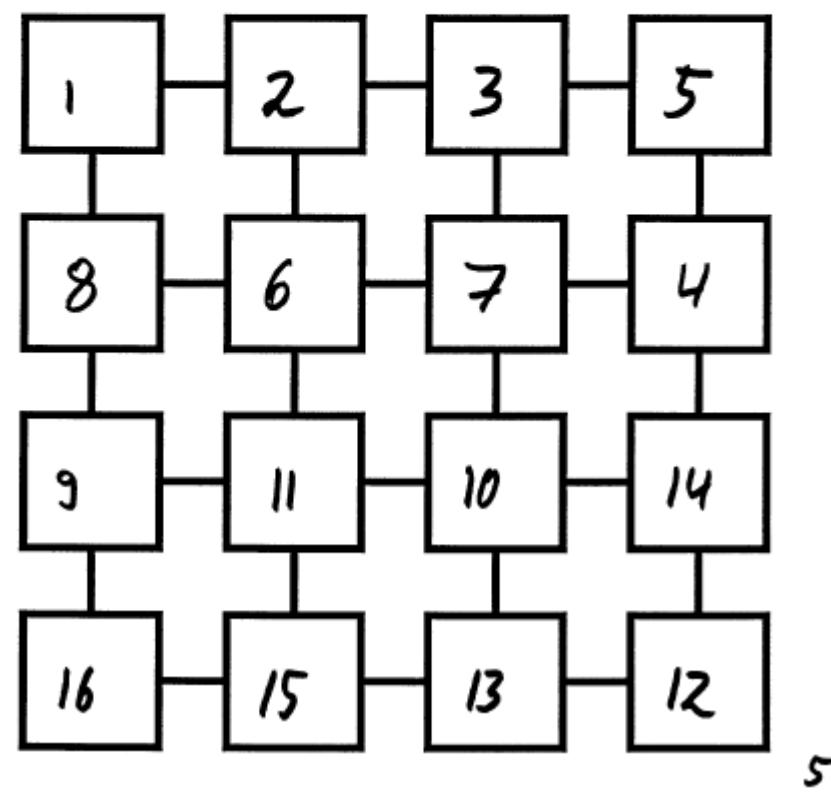
$\Downarrow \beta$



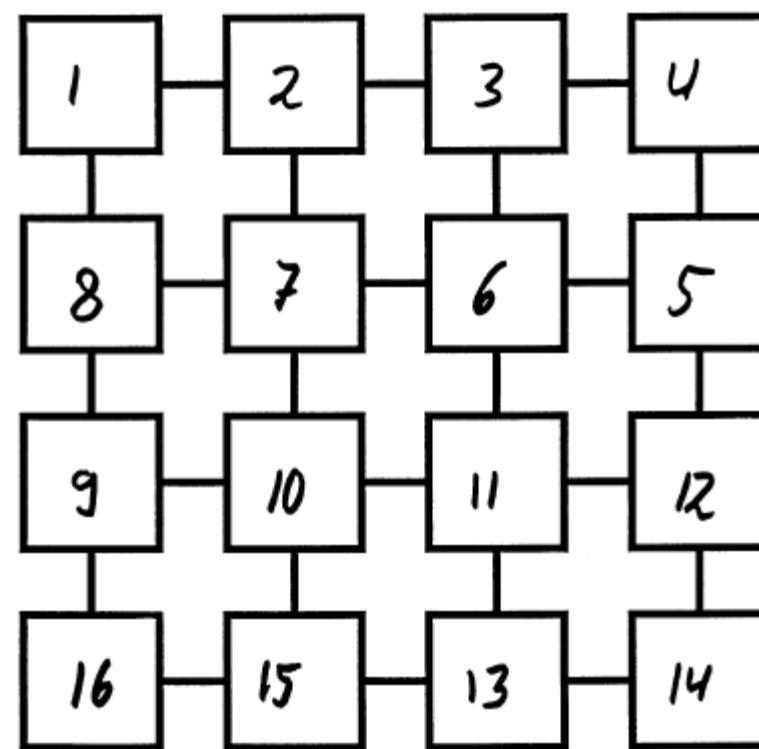
$\Downarrow C, EVEN$



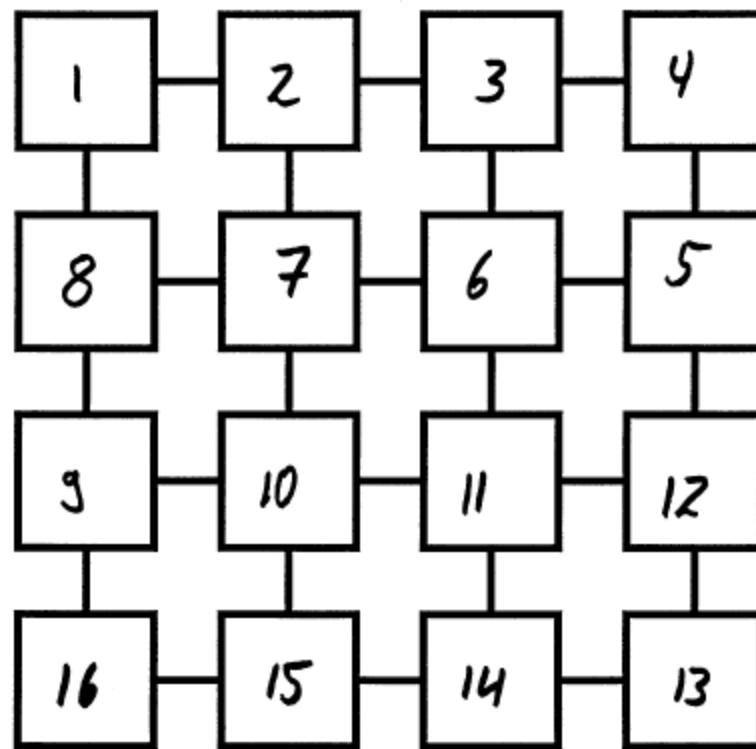
↓ odd



↓ EVEN



↙ ODD



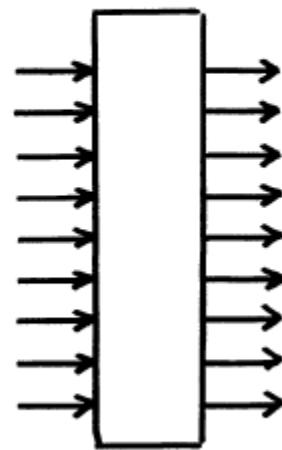
2

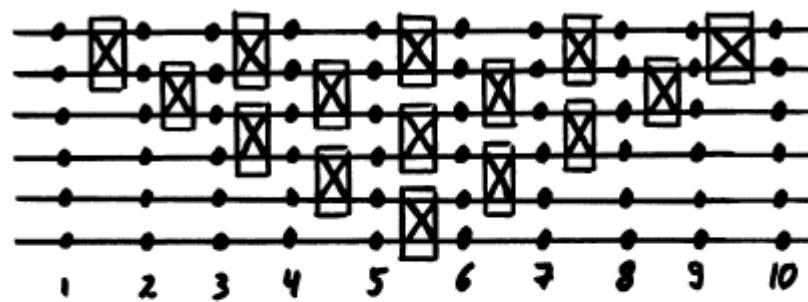
TIME $O(\sqrt{n'})$

0-1 PRINCIPLE :

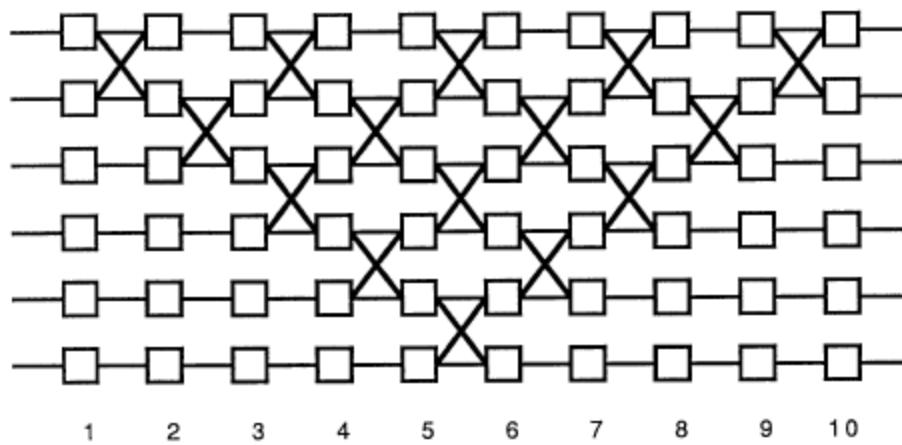
IF A NETWORK SORTS ALL
SEQUENCES OF 0's AND 1's,
THEN IT WILL ALSO SORT
ANY ARBITRARY SEQUENCE

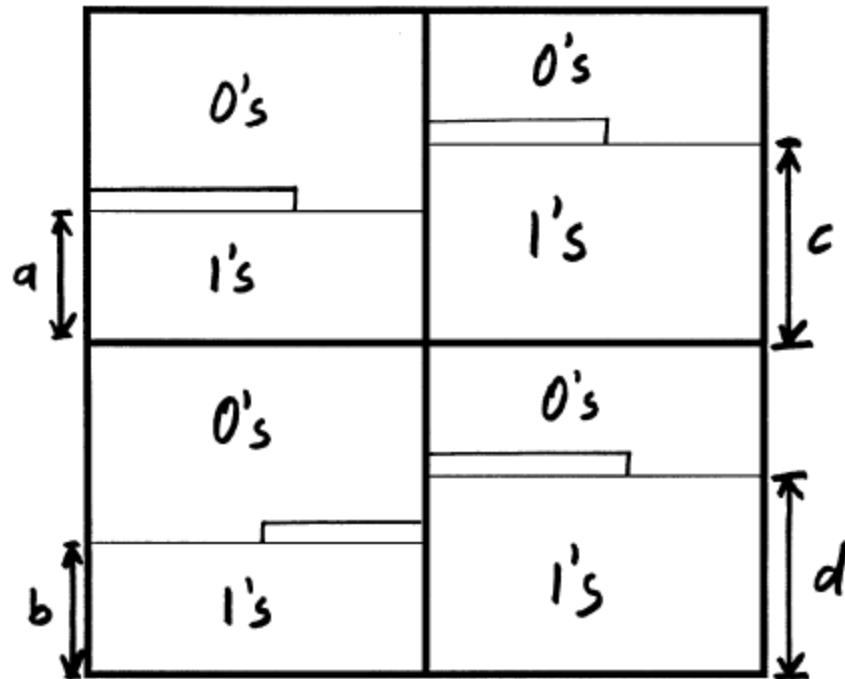
SORTING NETWORKS

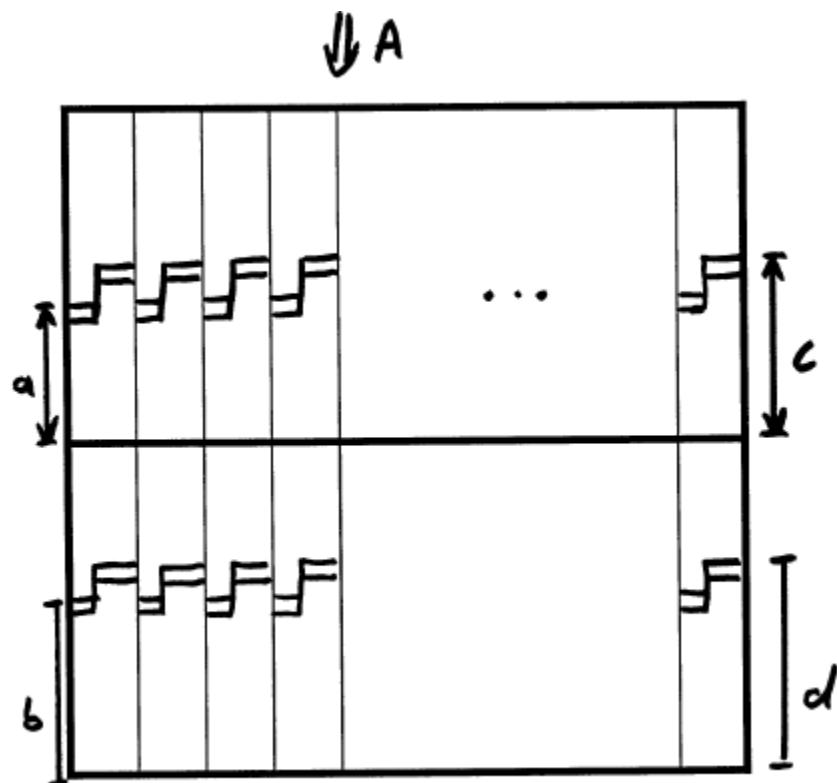


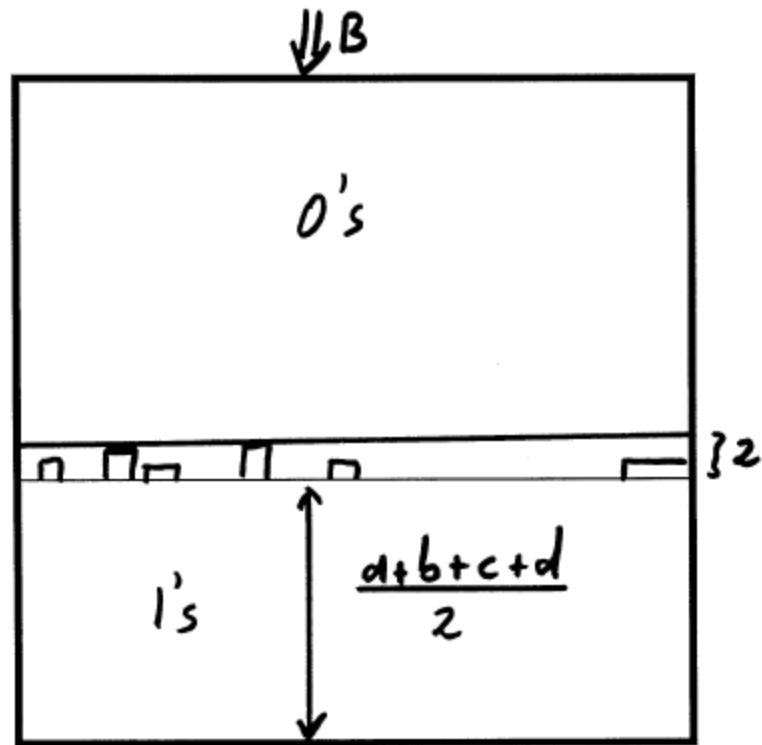


x ————— $\min\{x, y\}$
 y ————— $\max\{x, y\}$





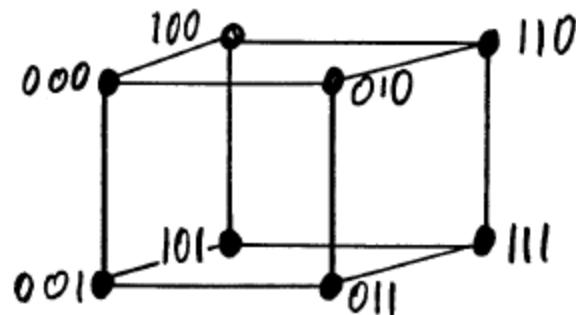




A $\sqrt{n} \times \sqrt{n'}$ PROCESSOR ARRAY
CAN BE SORTED IN TIME $O(\sqrt{n'})$.

Hypercube Sort: *Bitonic Sort*

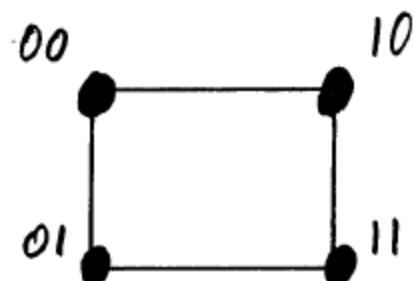
$d = 3$



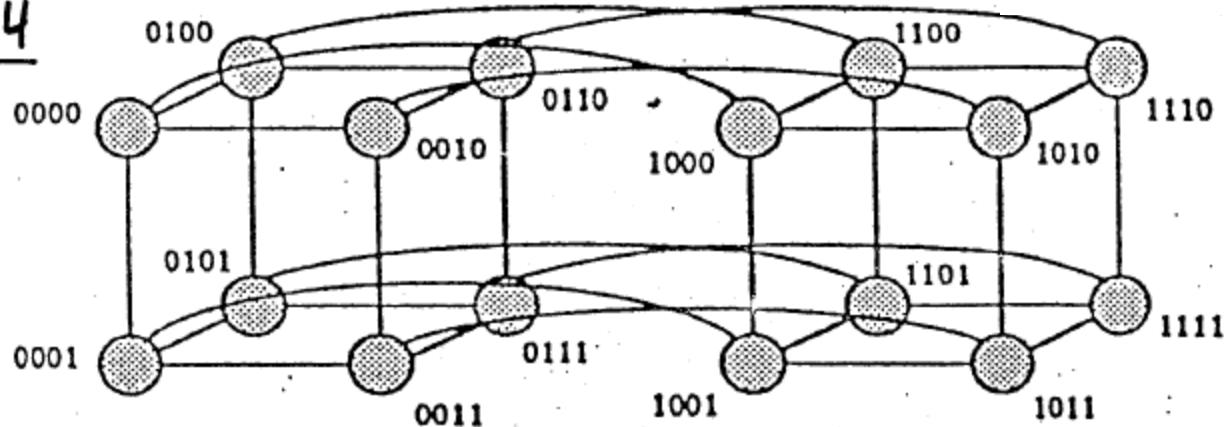
$d = 1$



$d = 2$



$d = 4$



UNIMODAL SEQUENCE:



9 10 13 17 21 19 16 15

BITONIC SEQUENCE:

CYCLIC SHIFT OF A UNIMODAL
SEQUENCE

16 15 9 10 13 17 21 19



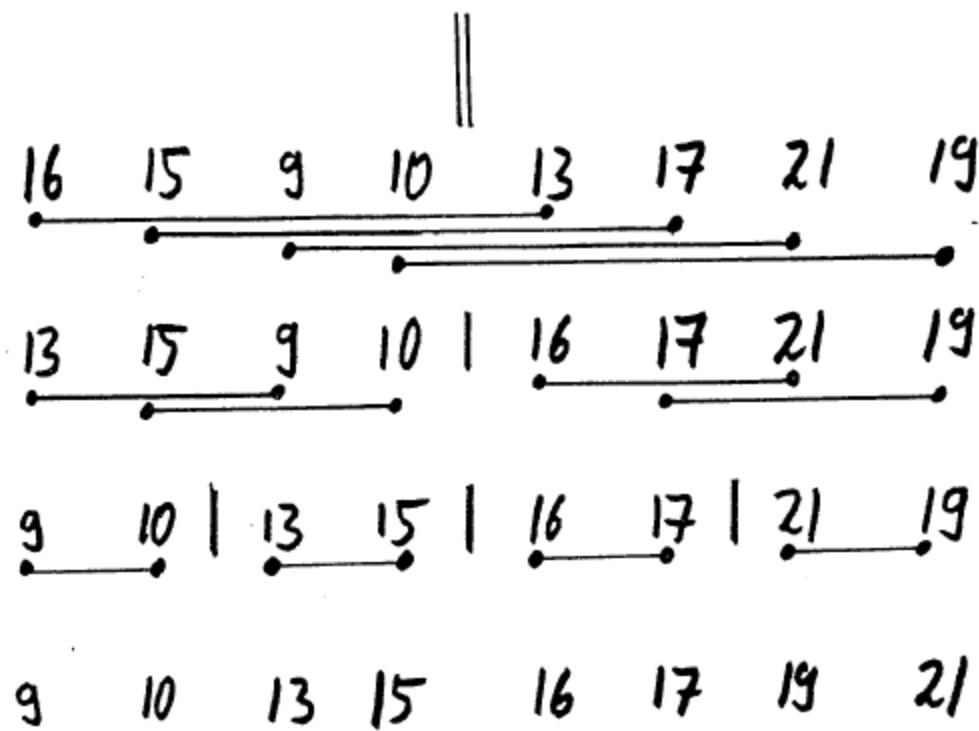
PROPERTIES OF BITONIC SEQUENCES

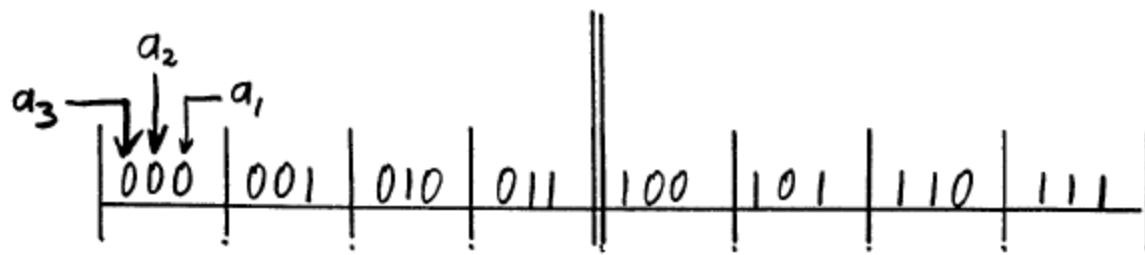
$\underline{x} = x_1, x_2, \dots, x_{2n}$ BITONIC SEQUENCE

$$L(\underline{x}) = y_1, \dots, y_r \quad y_i = \min \{x_i, x_{n+i}\}$$

$$U(\underline{x}) = z_1, \dots, z_n \quad z_i = \max \{x_i, x_{n+i}\}$$

- \Rightarrow (1) $L(\underline{x})$ and $U(\underline{x})$ ARE BITONIC
- (2) EVERY ELEMENT OF $L(\underline{x})$ IS SMALLER
THAN (OR EQUAL) EVERY ELEMENT
OF $U(\underline{x})$





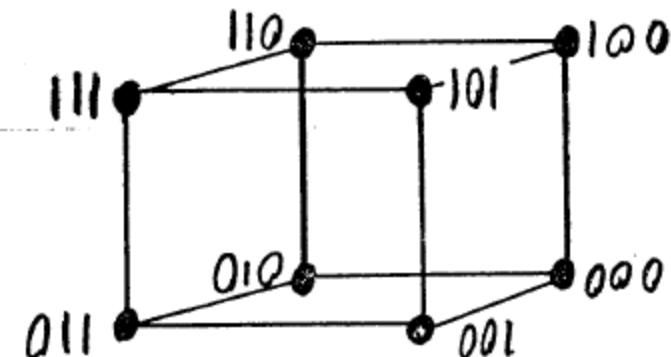
(a_3) 16 15 9 10 13 17 21 19

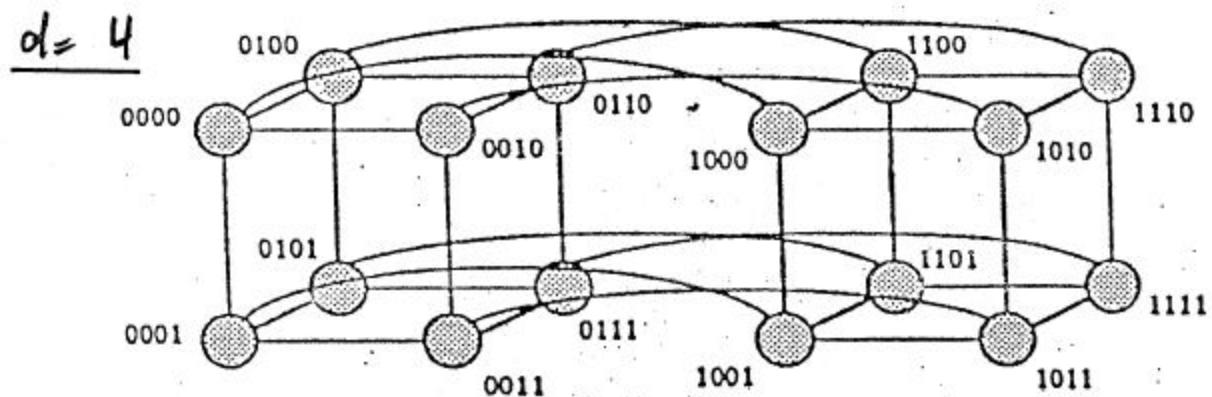
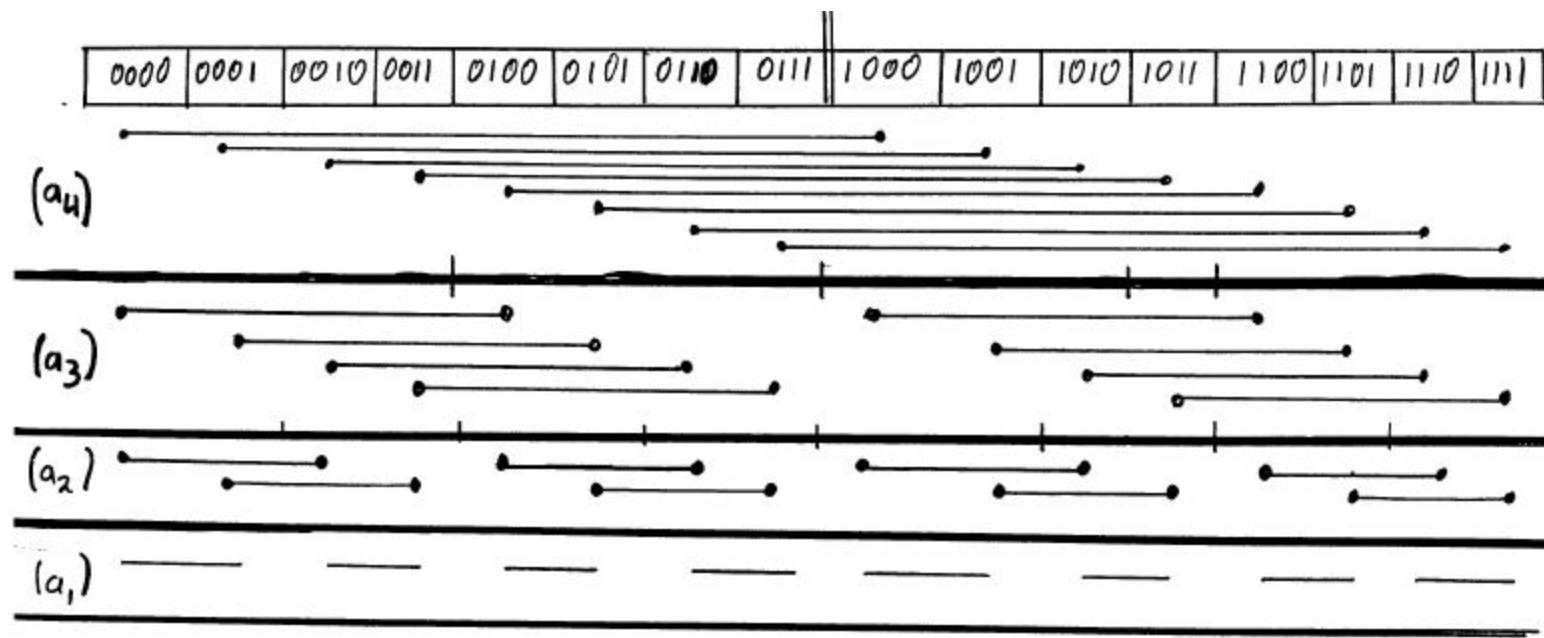
(a_2) 13 15 9 10 | 16 17 21 19

(a_1) 9 10 | 13 15 | 16 17 | 21 19

9 10 13 15 16 17 19 21

$n=8$
$d=3$





HOW TO SORT AN ARBITRARY SEQUENCE a_1, \dots, a_n
OF NUMBERS? (ASSUME: $n = 2^k$)

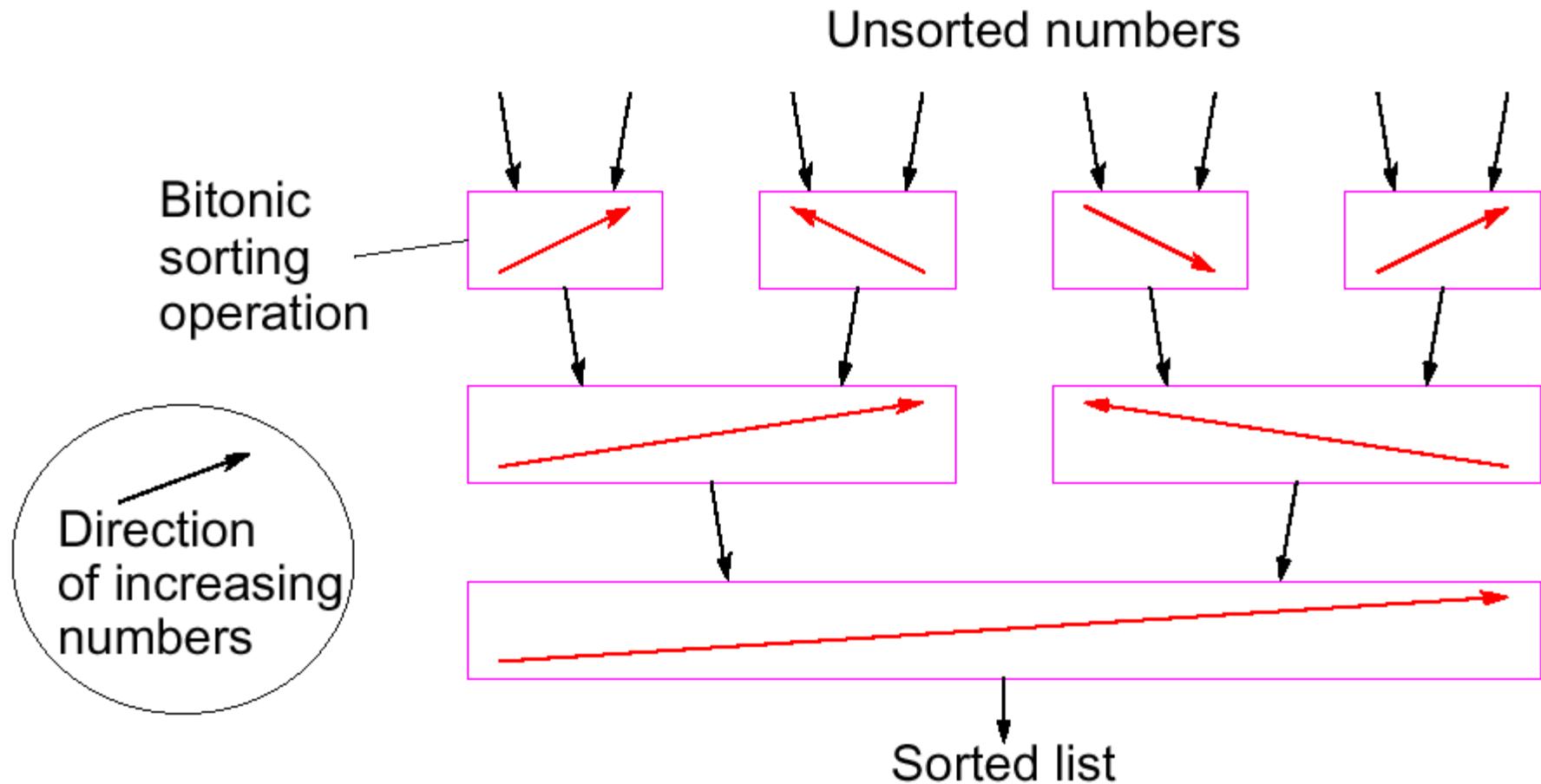
- 1) SPLIT $a_1 \dots a_n$ INTO $a_1 \dots a_{\frac{n}{2}}$ AND
 $a_{\frac{n}{2}+1} \dots a_n$
- 2) RECURSIVELY SORT EACH HALF
OF THE SEQUENCE (IN PARALLEL),
USING $\frac{P}{2}$ PROCESSORS, EACH.
- 3) MERGE THE TWO SORTED SUBSEQUENCES
HOW ?

IF $X \leq Y$ ARE INCREASING SEQUENCES,

THE $X \leq Y^R$ IS BITONIC

\Rightarrow SORT $X \leq Y^R$

Bitonic Mergesort



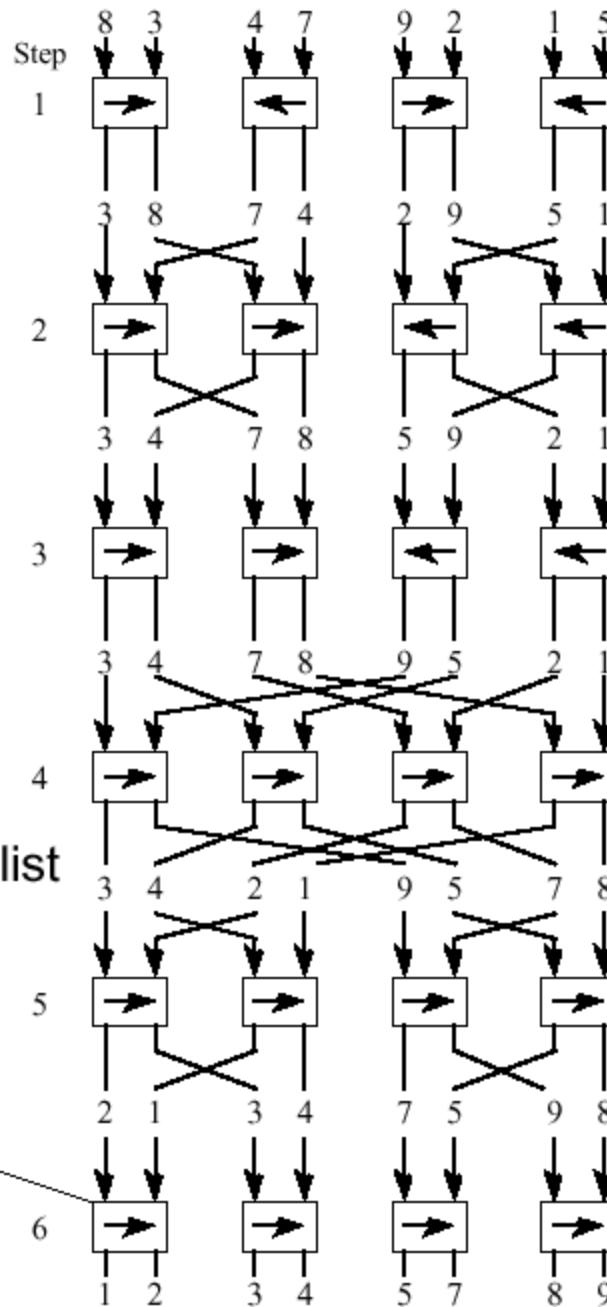
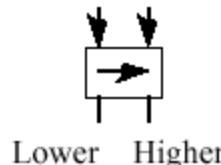
Bitonic Mergesort on Eight Numbers

Form bitonic lists of four numbers

Form bitonic list of eight numbers

Sort bitonic list

Compare and exchange



$$T(n) = T\left(\frac{n}{2}\right) + O(\log n)$$

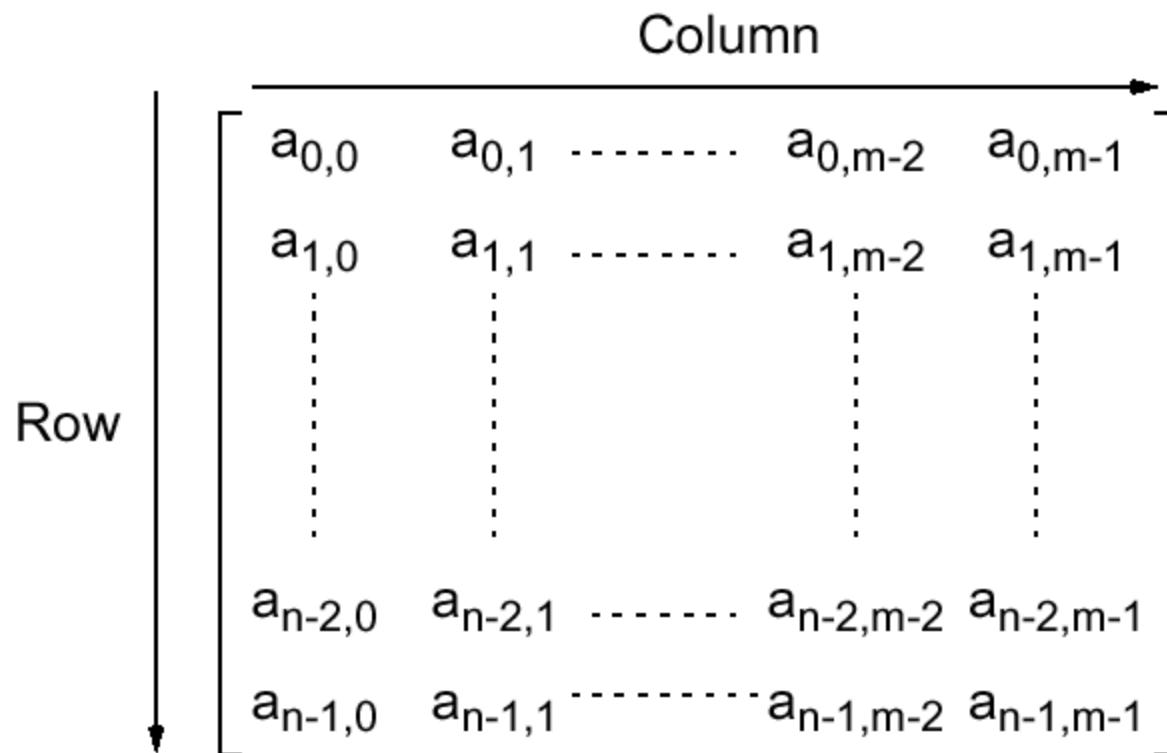
$$\Rightarrow T(n) = O(\log^2 n)$$

Numerical Algorithms

- Matrix multiplication
- Solving a system of linear equations

Matrices — A Review

An $n \times m$ matrix



Matrix Addition

Involves adding corresponding elements of each matrix to form the result matrix.

Given the elements of **A** as $a_{i,j}$ and the elements of **B** as $b_{i,j}$, each element of **C** is computed as

$$c_{i,j} = a_{i,j} + b_{i,j}$$

$$(0 \leq i < n, 0 \leq j < m)$$

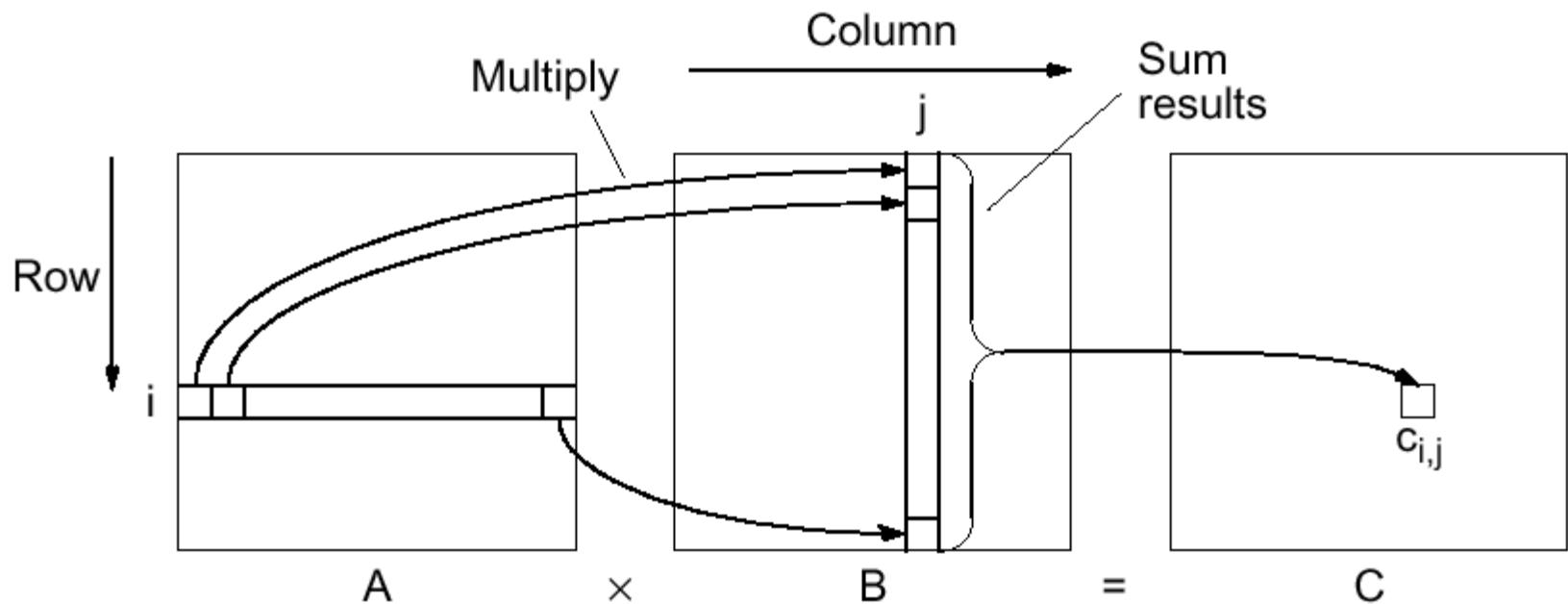
Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements, $c_{i,j}$ ($0 \leq i < n$, $0 \leq j < m$), are computed as follows:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

where **A** is an $n \times l$ matrix and **B** is an $l \times m$ matrix.

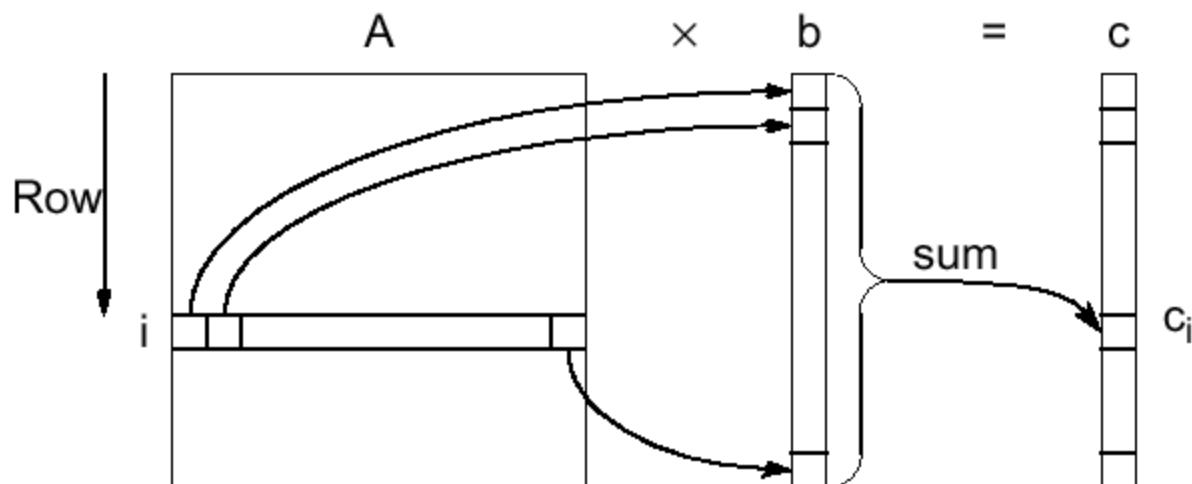
Matrix multiplication, $C = A \times B$



Matrix-Vector Multiplication

$$\mathbf{c} = \mathbf{A} \times \mathbf{b}$$

Matrix-vector multiplication follows directly from the definition of matrix-matrix multiplication by making \mathbf{B} an $n \times 1$ matrix (vector). Result an $n \times 1$ matrix (vector).



Relationship of Matrices to Linear Equations

A system of linear equations can be written in matrix form:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Matrix **A** holds the *a* constants

x is a vector of the unknowns

b is a vector of the *b* constants.

Implementing Matrix Multiplication

Sequential Code

Assume throughout that the matrices are square ($n \times n$ matrices).

The sequential code to compute $\mathbf{A} \times \mathbf{B}$ could simply be

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        c[i][j] = 0;
        for (k = 0; k < n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

This algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$. Very easy to parallelize.

Parallel Code (Shared Memory)

With n processors (and $n \times n$ matrices), can obtain:

- Time complexity of $O(n^2)$ with n processors
Each instance of inner loop independent and can be done by a separate processor
- Time complexity of $O(n)$ with n^2 processors
One element of A and B assigned to each processor.
Cost optimal since $O(n^3) = n \times O(n^2) = n^2 \times O(n)$.
- Time complexity of $O(\log n)$ with n^3 processors
By parallelizing the inner loop. Not cost-optimal since $O(n^3) \neq n^3 \times O(\log n)$.

$O(\log n)$ lower bound for parallel matrix multiplication.

Partitioning into Submatrices

Suppose matrix divided into s^2 submatrices. Each submatrix has $n/s \times n/s$ elements. Using notation $A_{p,q}$ as submatrix in submatrix row p and submatrix column q :

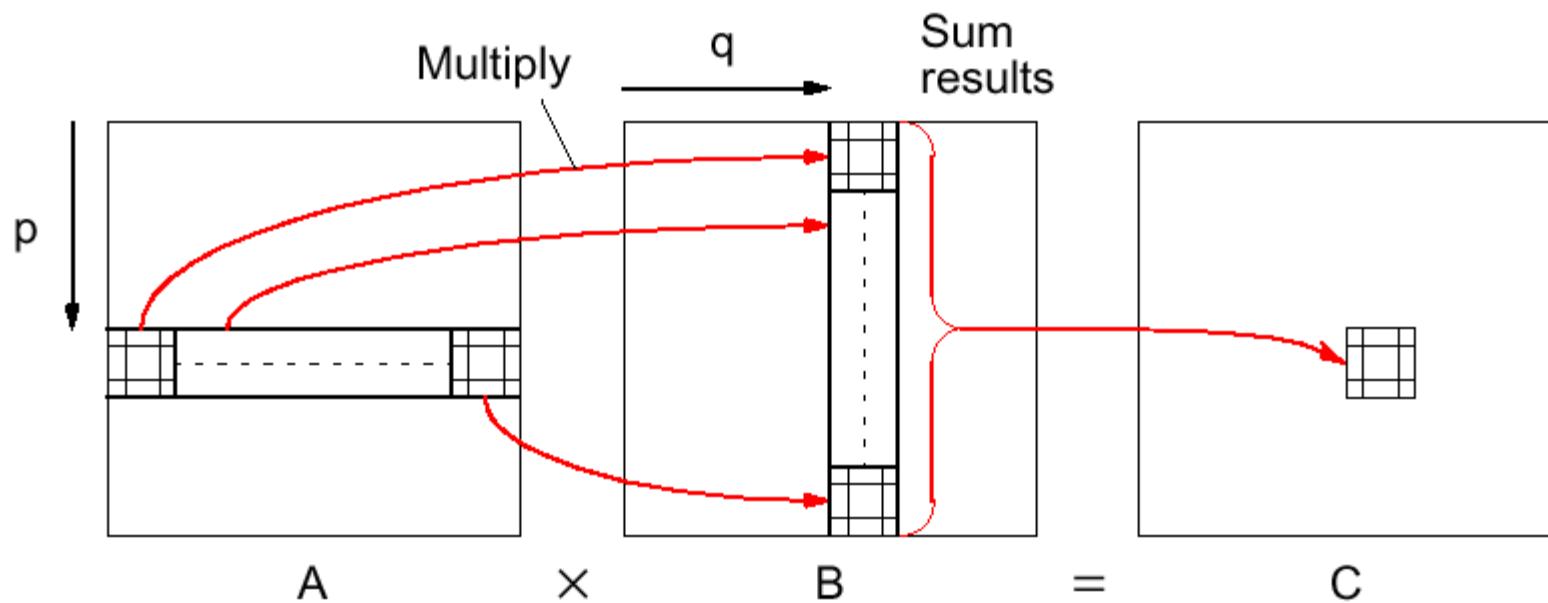
```
for (p = 0; p < s; p++)
    for (q = 0; q < s; q++) {
        Cp,q = 0; /* clear elements of submatrix */
        for (r = 0; r < m; r++)/* submatrix multiplication &*/
            Cp,q = Cp,q + Ap,r * Br,q; /*add to accum. submatrix*/
    }
```

The line

$$C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};$$

means multiply submatrix $A_{p,r}$ and $B_{r,q}$ using matrix multiplication and add to submatrix $C_{p,q}$ using matrix addition. Known as *block matrix multiplication*.

Block Matrix Multiplication



Submatrix multiplication

(a) Matrices

$$\begin{bmatrix} a_{0,0} & a_{0,1} & | & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & | & a_{1,2} & a_{1,3} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} & | & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & | & b_{1,2} & b_{1,3} \end{bmatrix}$$

$$\begin{bmatrix} a_{2,0} & a_{2,1} & | & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & | & a_{3,2} & a_{3,3} \end{bmatrix} \times \begin{bmatrix} b_{2,0} & b_{2,1} & | & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & | & b_{3,2} & b_{3,3} \end{bmatrix}$$

(b) Multiplying $A_{0,0} \times B_{0,0}$
to obtain $C_{0,0}$

$$A_{0,0} \quad B_{0,0} \quad A_{0,1} \quad B_{1,0}$$

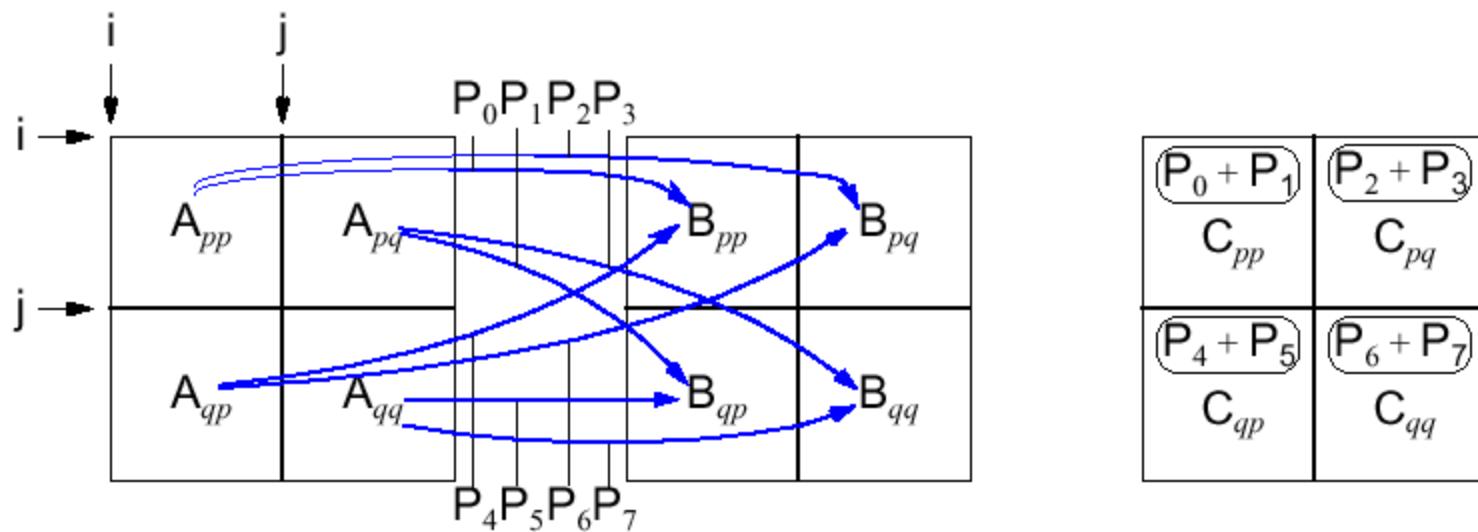
$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix} \times \begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0} + a_{0,3}b_{3,0} & a_{0,2}b_{2,1} + a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0} + a_{1,3}b_{3,0} & a_{1,2}b_{2,1} + a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} + a_{1,2}b_{2,0} + a_{1,3}b_{3,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

Recursive Implementation



Apply same algorithm on each submatrix recursively.

Excellent algorithm for a shared memory systems because of locality of operations.

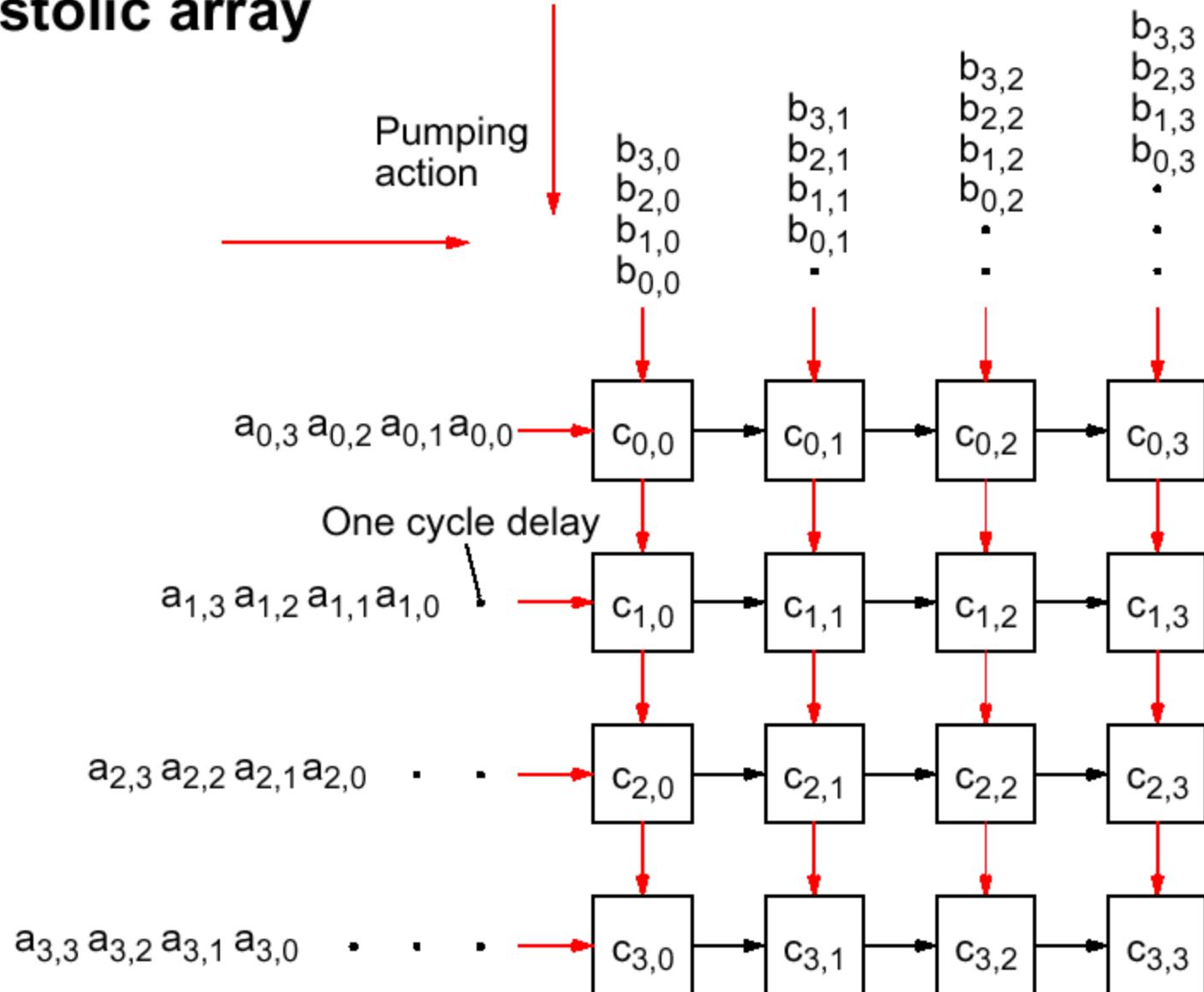
Recursive Algorithm

```
mat_mult(App, Bpp, s)
{
    if (s == 1)          /* if submatrix has one element */
        C = A * B;      /* multiply elements */
    else {
        /* continue to make recursive calls */
        s = s/2;         /* no of elements in each row/column */
        P0 = mat_mult(App, Bpp, s);
        P1 = mat_mult(Apq, Bqp, s);
        P2 = mat_mult(App, Bpq, s);
        P3 = mat_mult(Apq, Bqq, s);
        P4 = mat_mult(Aqp, Bpp, s);
        P5 = mat_mult(Aqq, Bqp, s);
        P6 = mat_mult(Aqp, Bpq, s);
        P7 = mat_mult(Aqq, Bqq, s);
        Cpp = P0 + P1;    /* add submatrix products to */
        Cpq = P2 + P3;    /* form submatrices of final matrix */
        Cqp = P4 + P5;
        Cqq = P6 + P7;
    }
    return (C);           /* return final matrix */
}
```

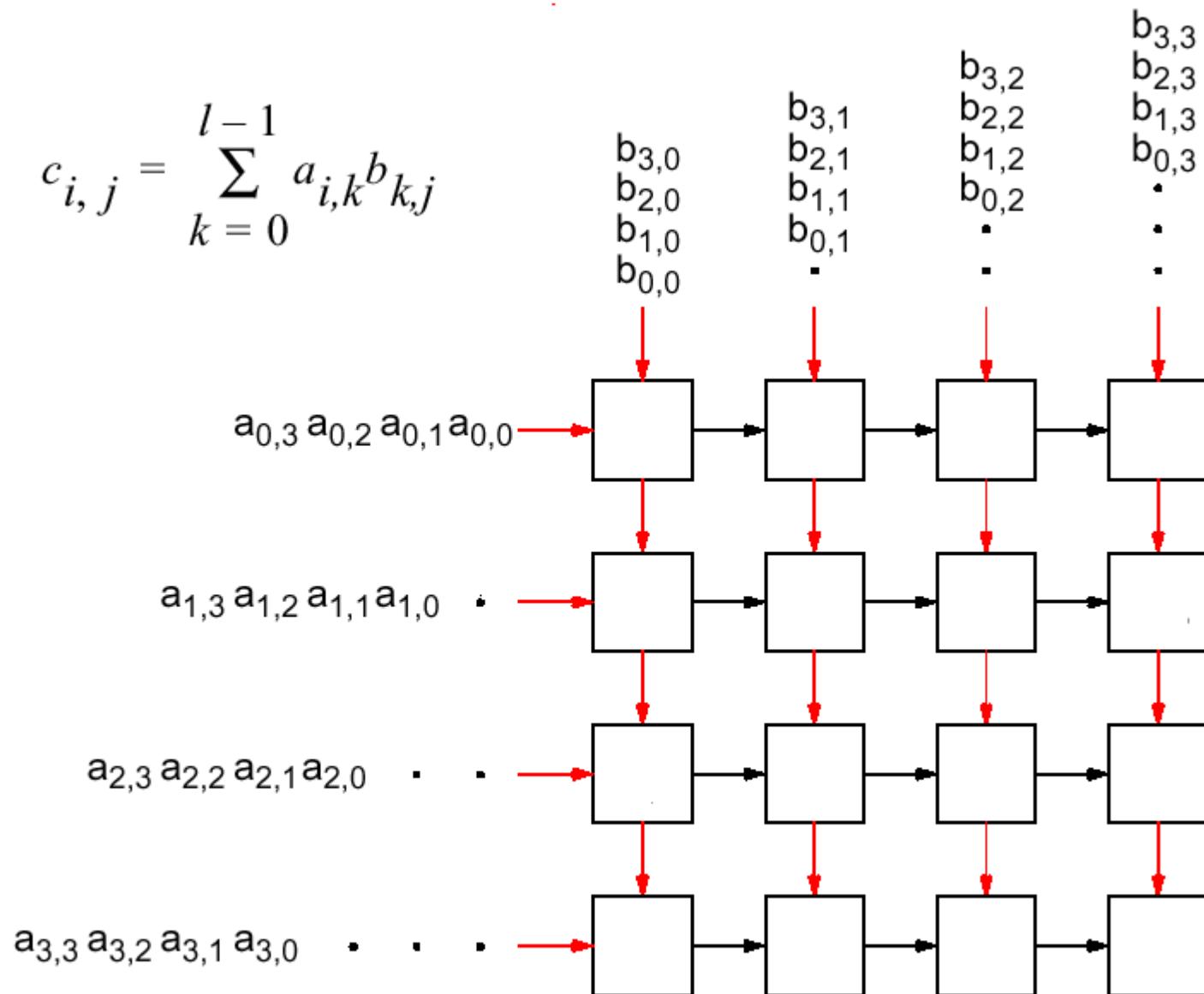
Mesh Implementations

- Systolic array
- Cannon's algorithm

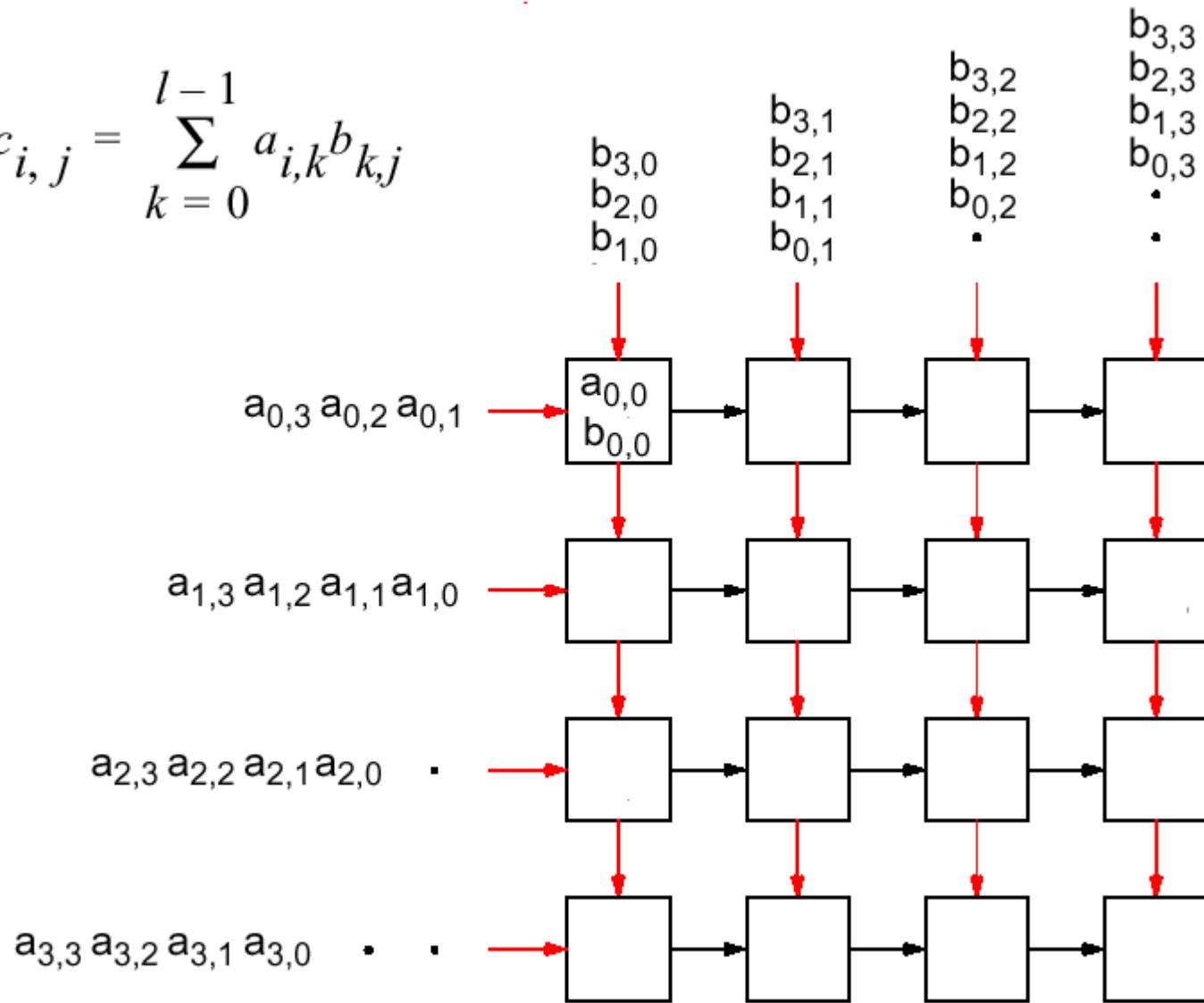
Systolic array



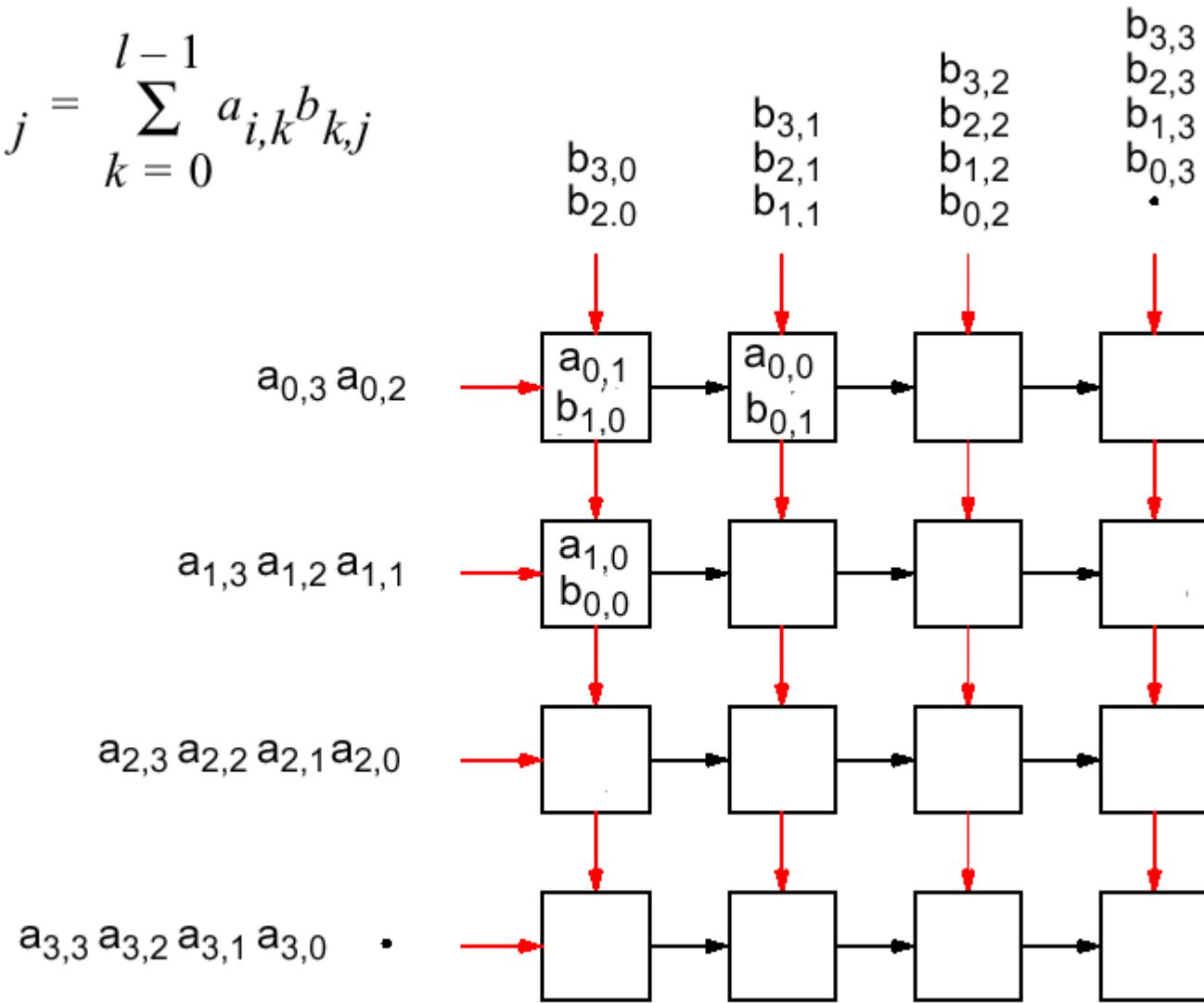
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



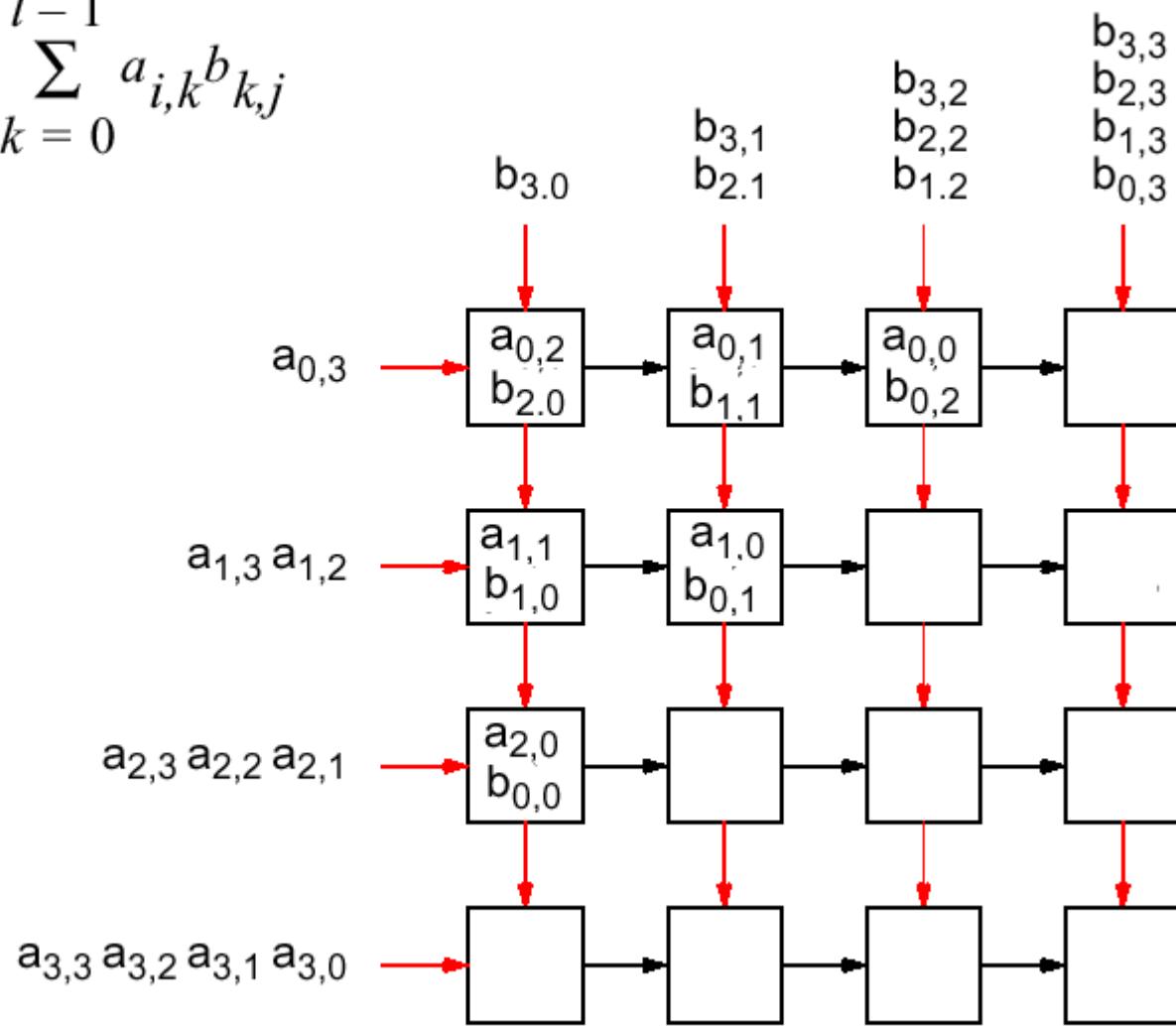
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



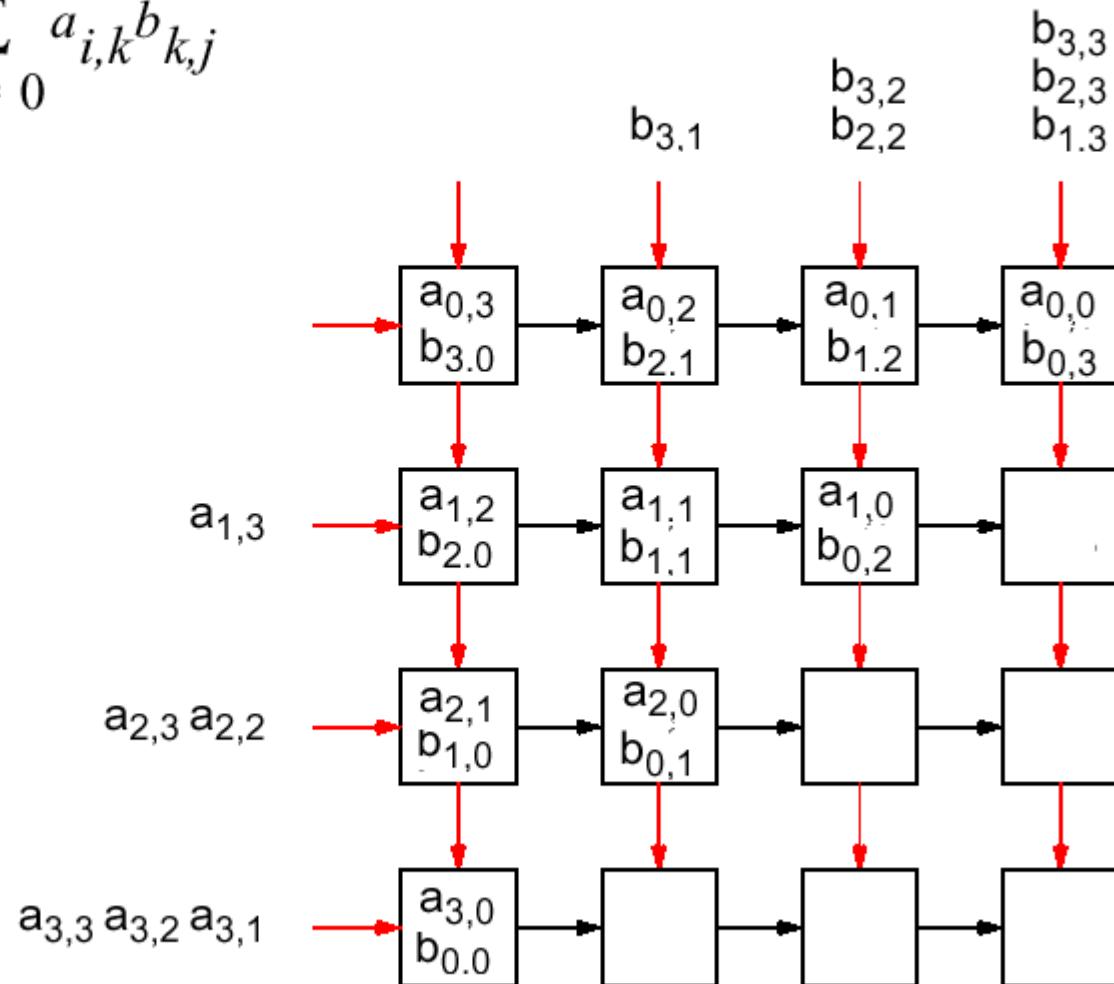
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



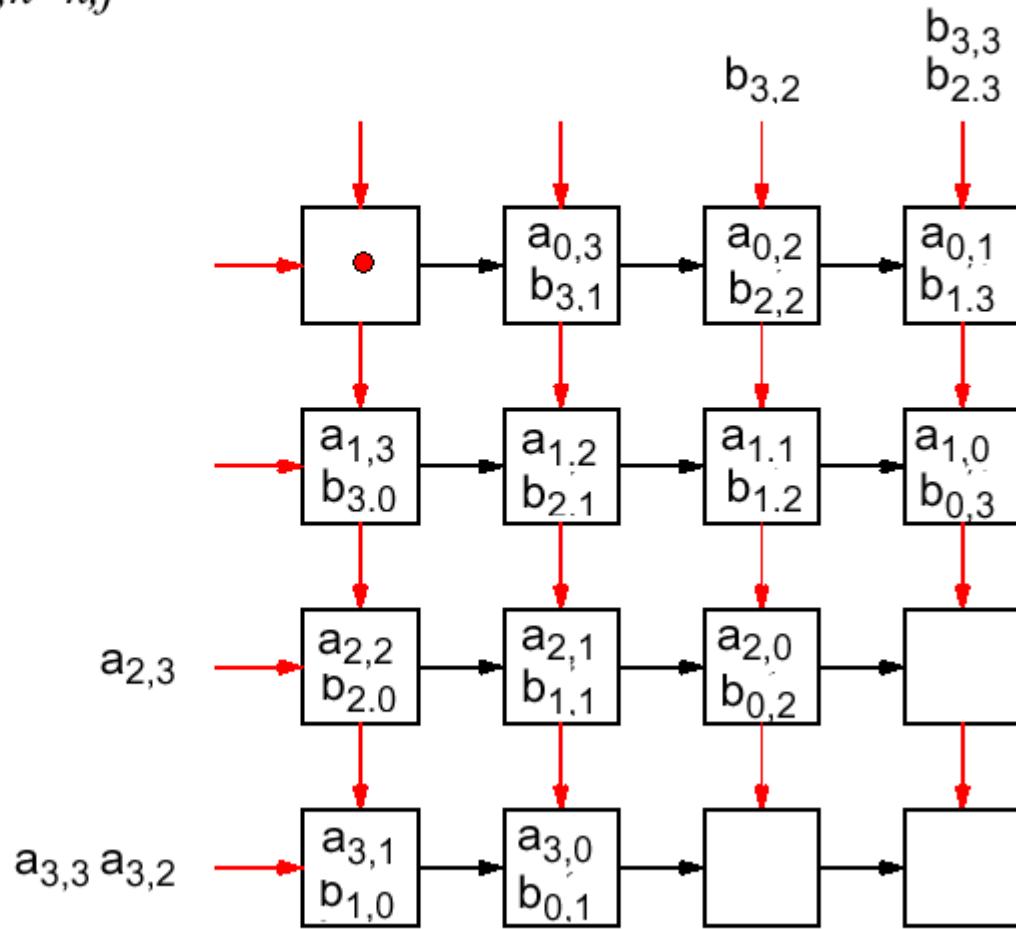
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



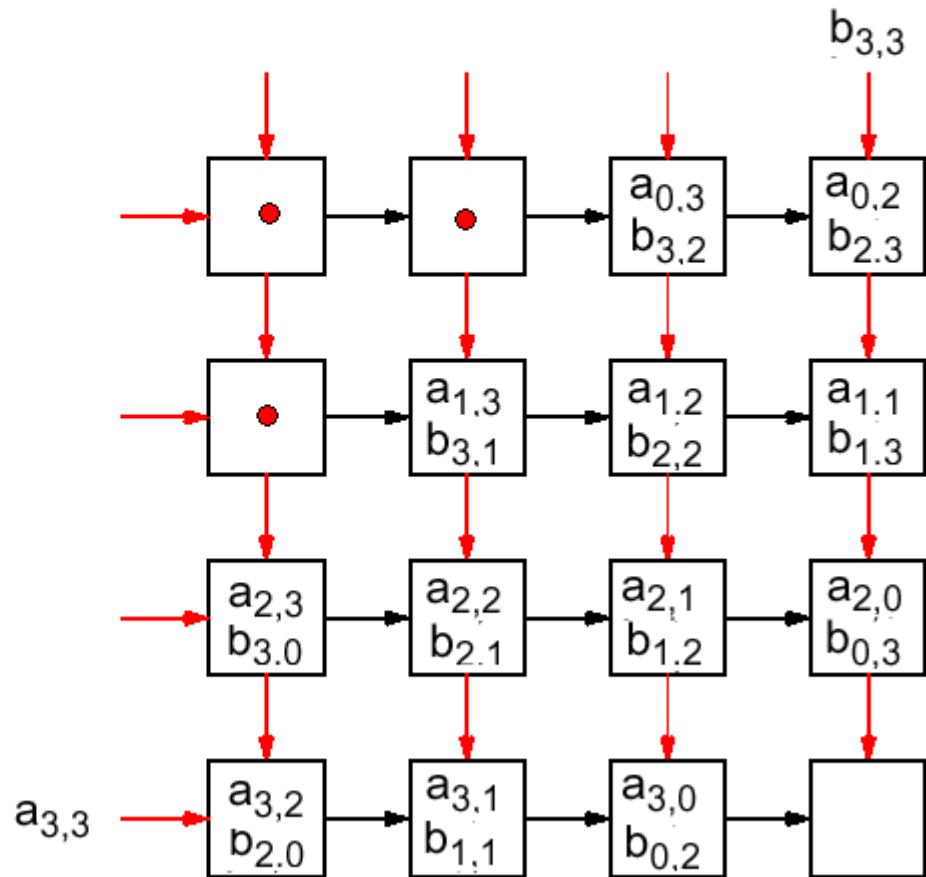
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



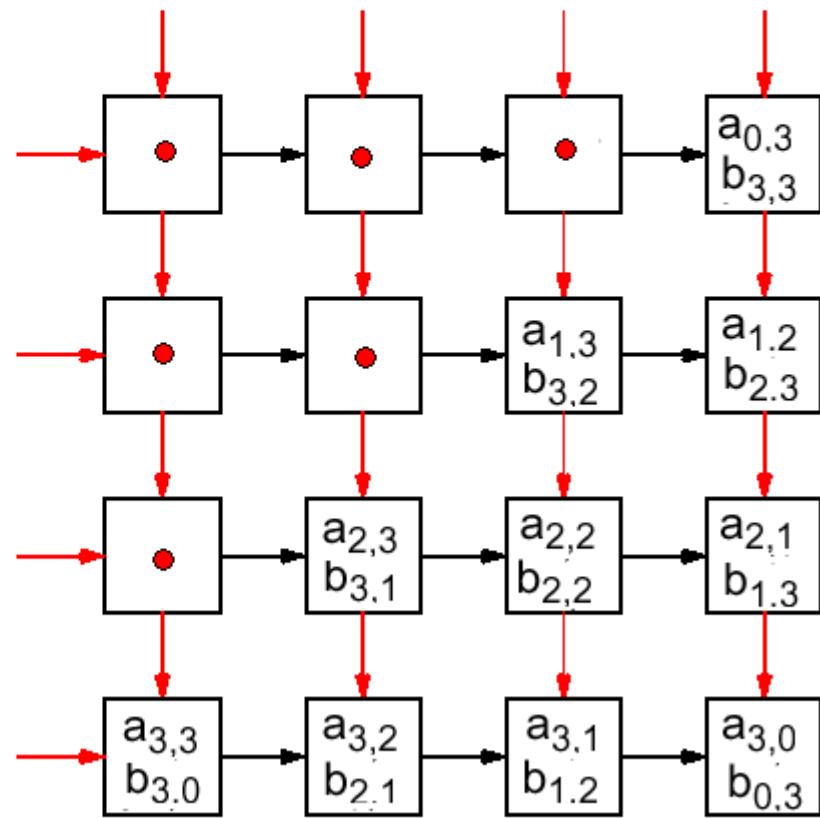
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



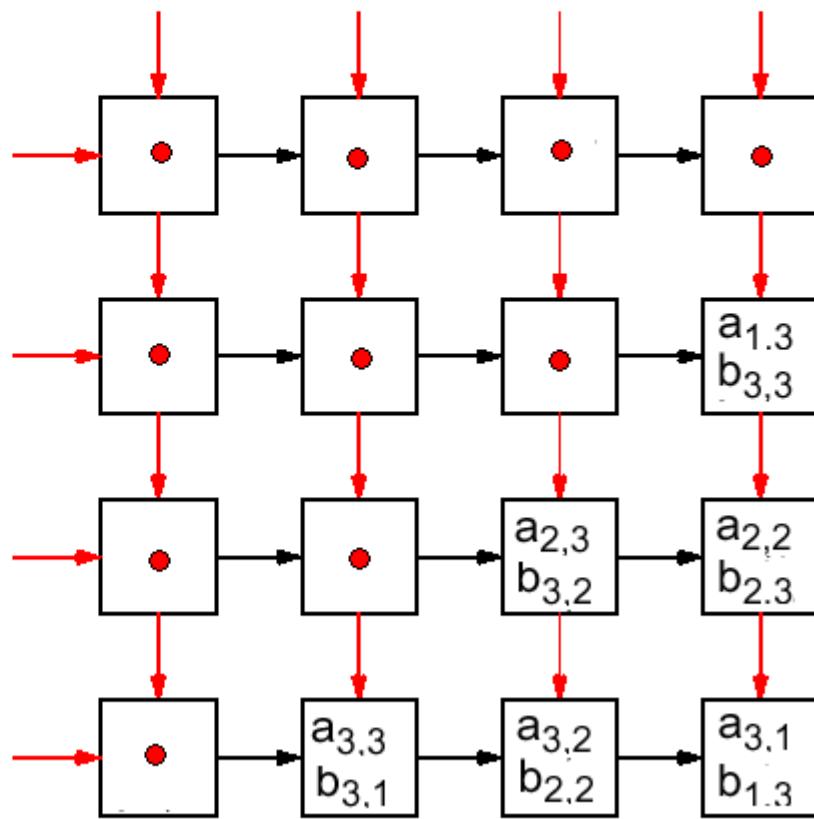
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



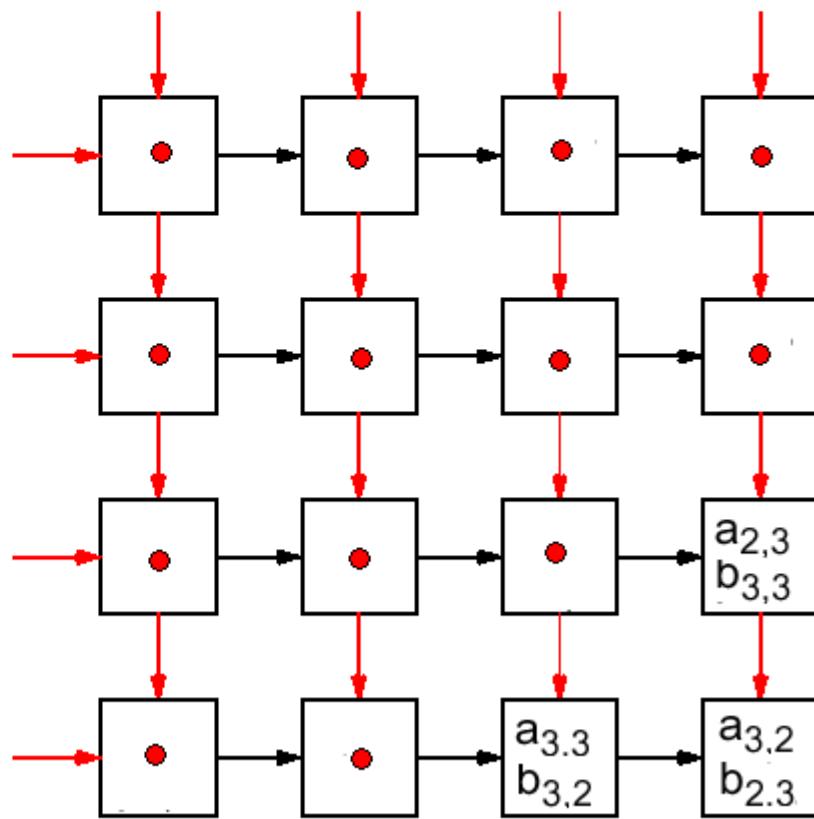
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



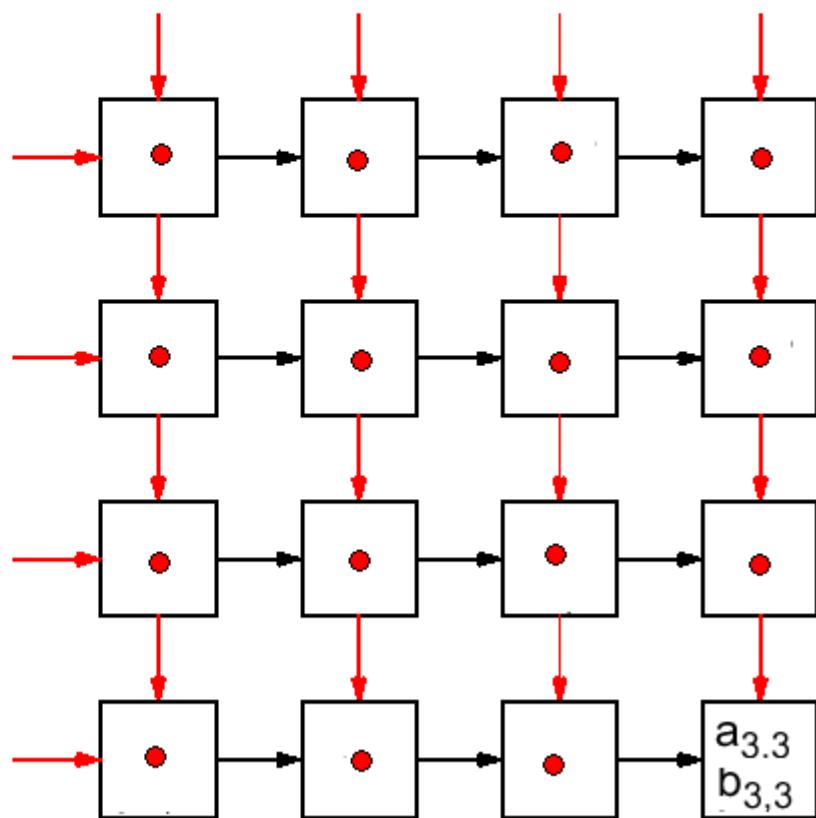
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



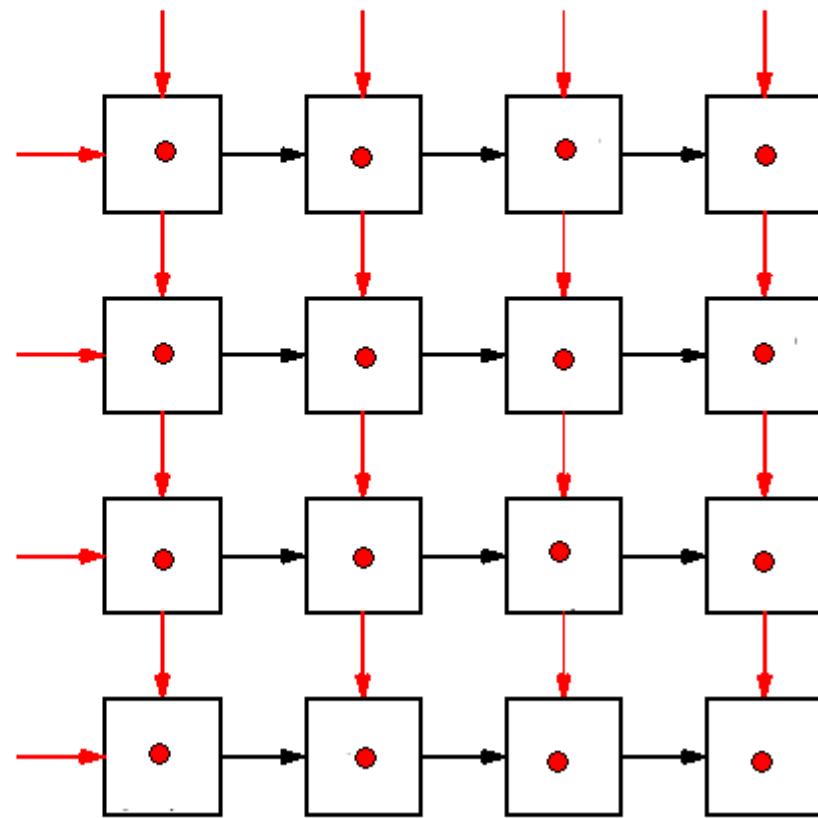
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{kj}$$



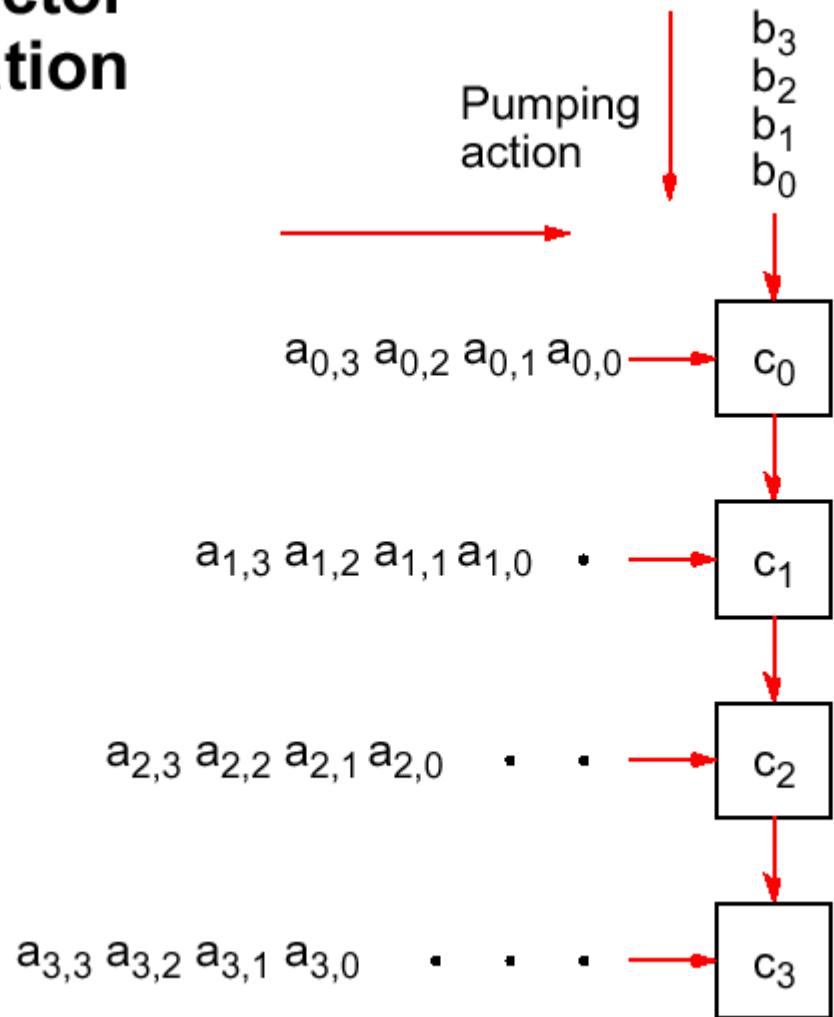
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

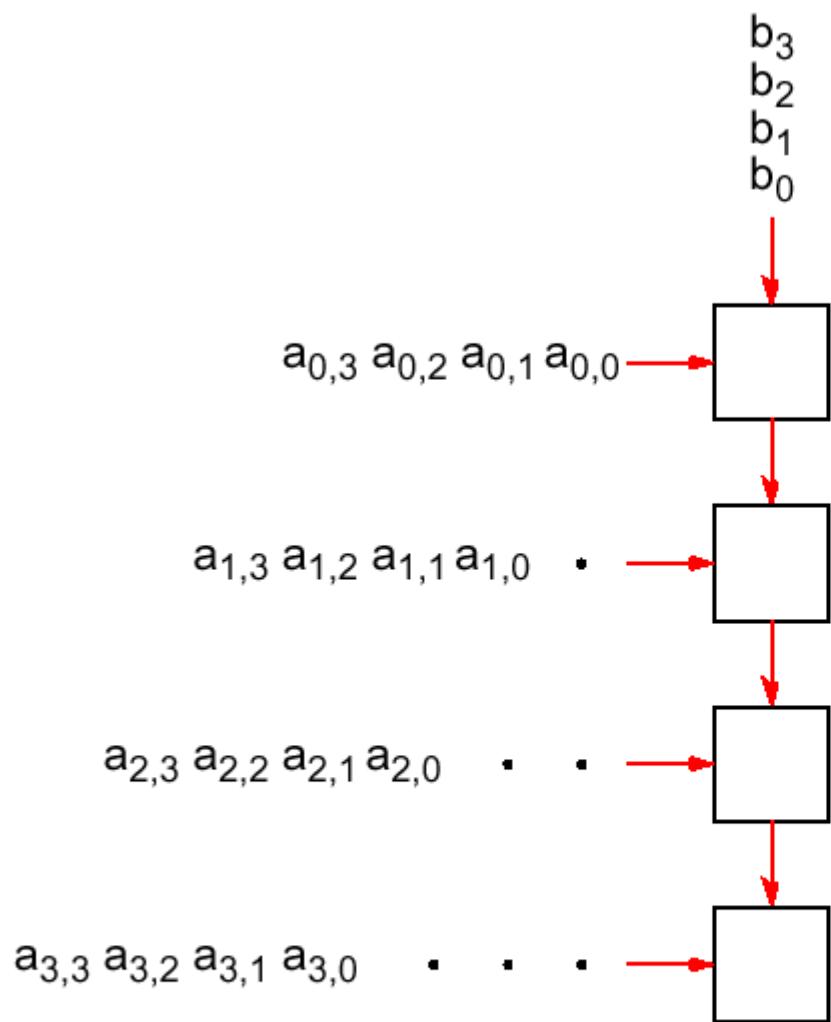


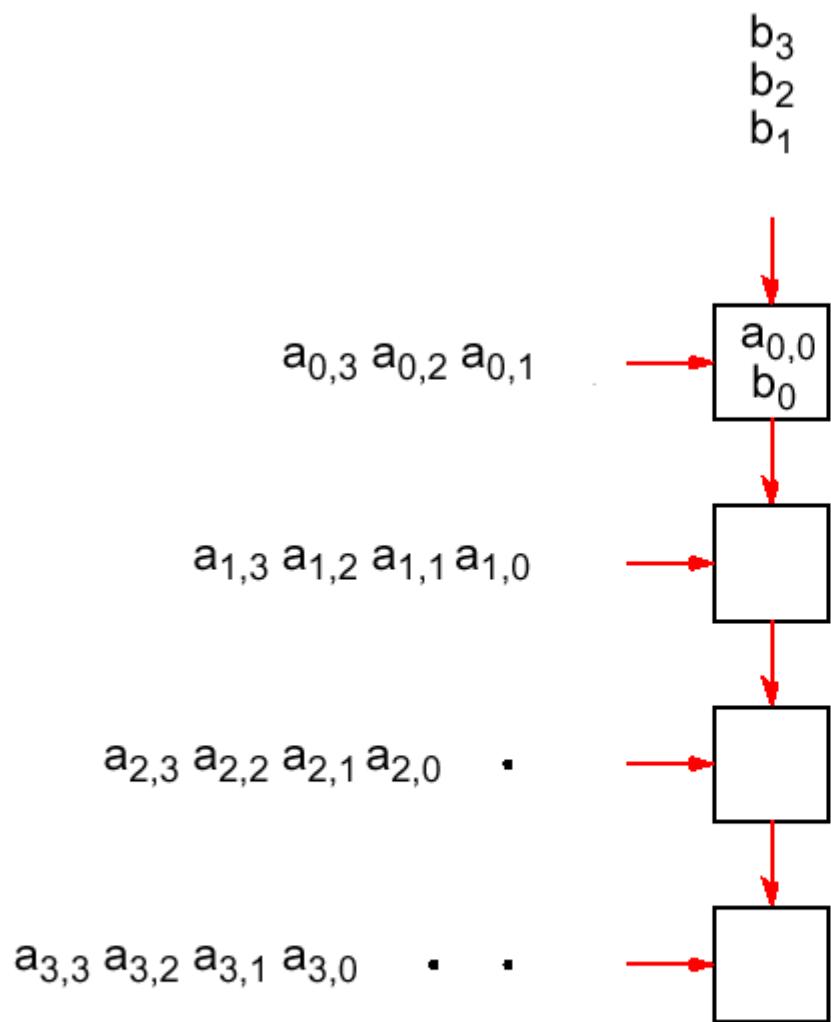
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{kj}$$



Matrix-Vector Multiplication







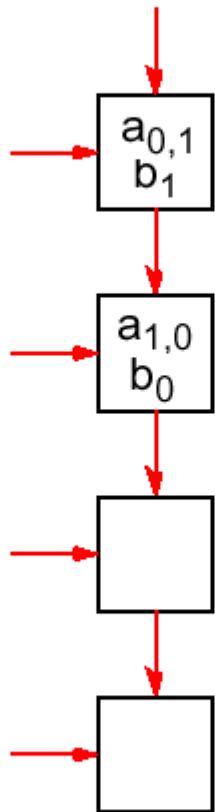
b_3
 b_2

$a_{0,3} \ a_{0,2}$

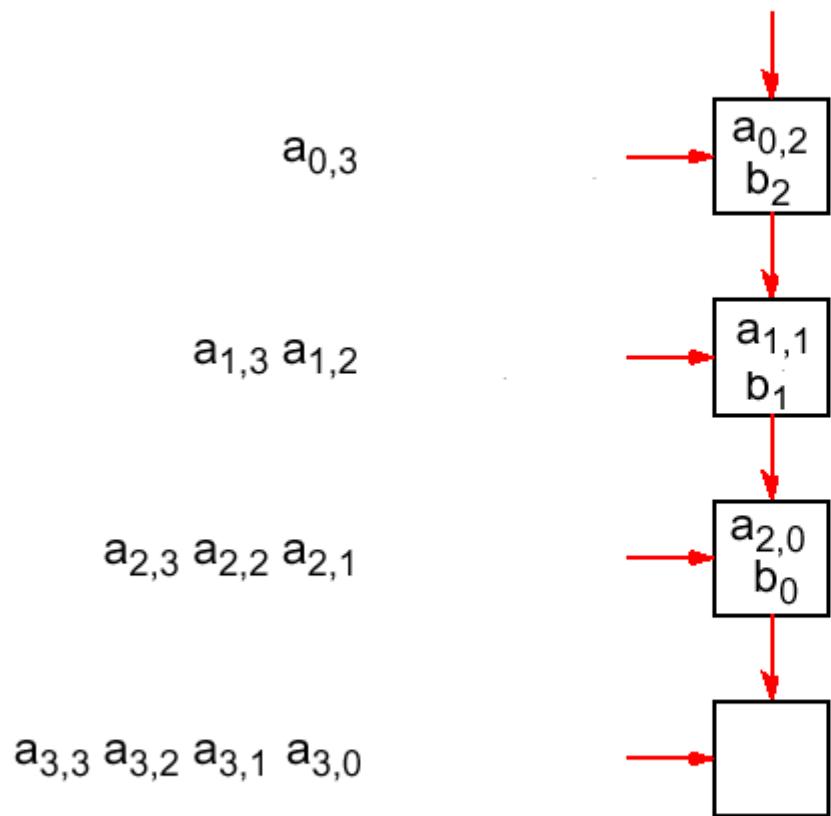
$a_{1,3} \ a_{1,2} \ a_{1,1}$

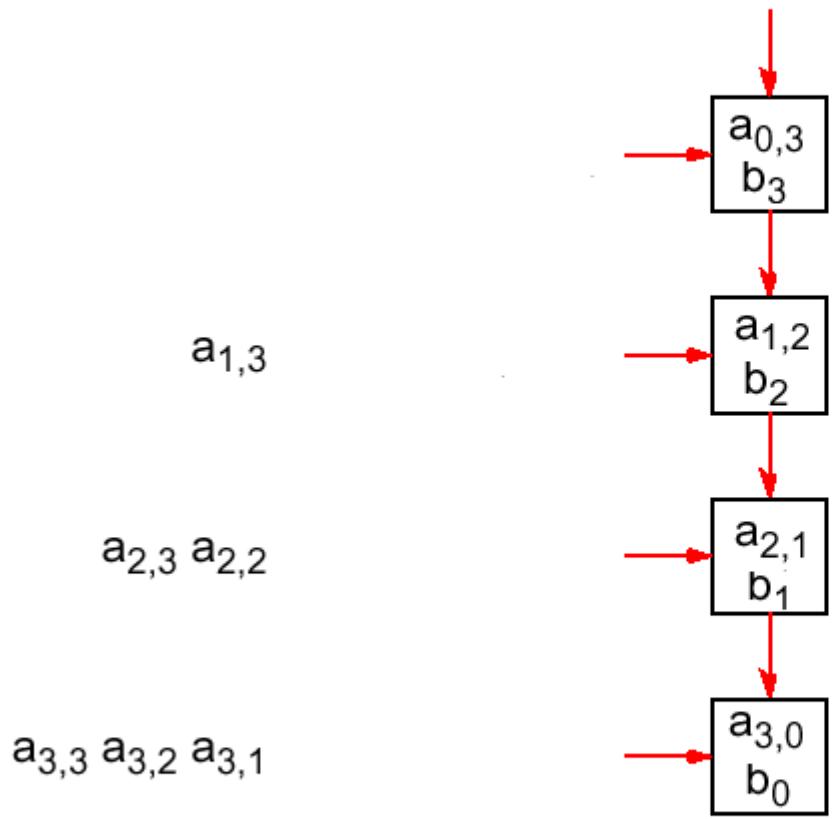
$a_{2,3} \ a_{2,2} \ a_{2,1} \ a_{2,0}$

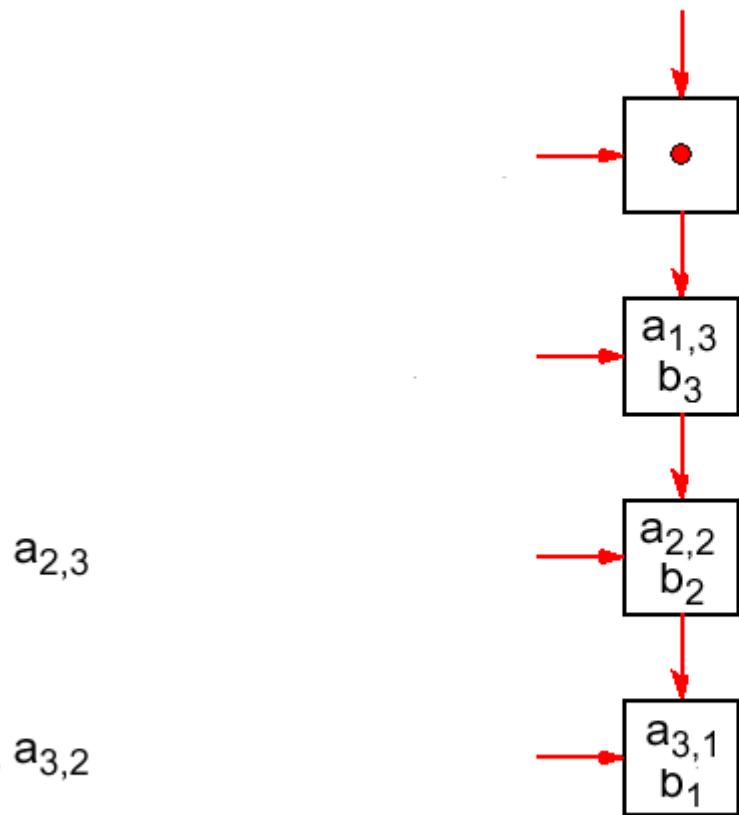
$a_{3,3} \ a_{3,2} \ a_{3,1} \ a_{3,0}$

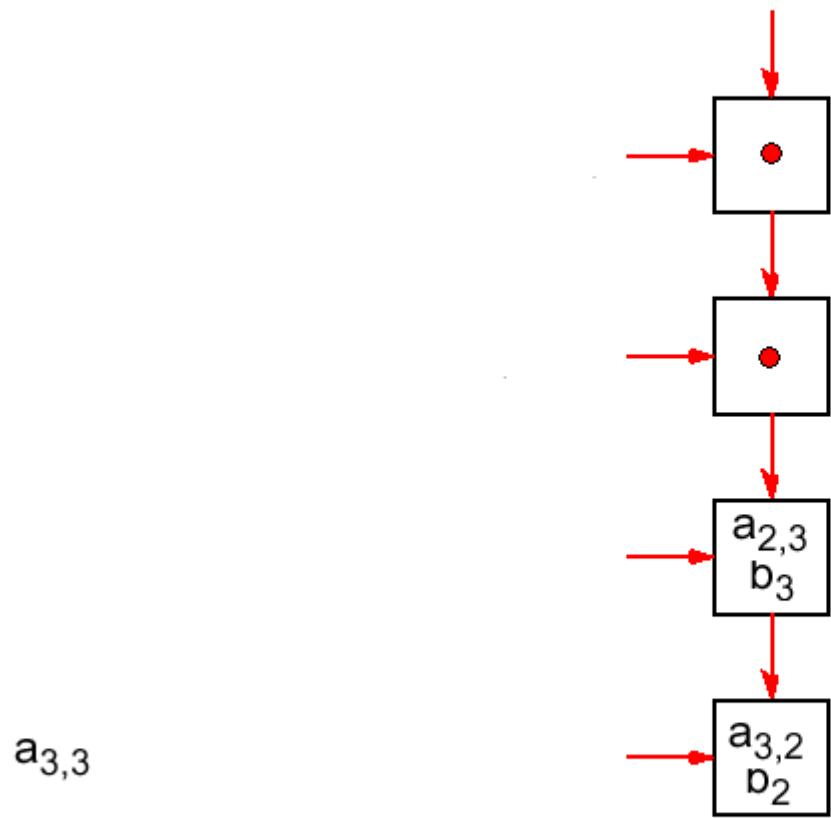


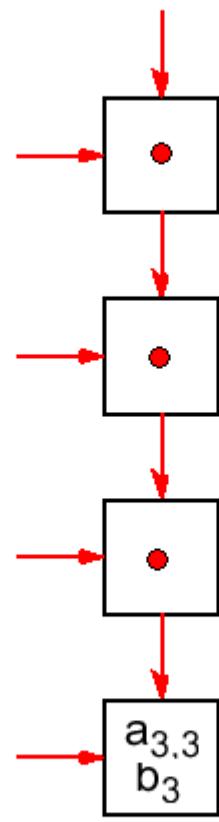
b_3

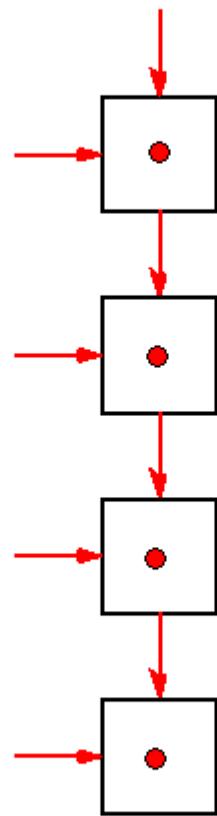












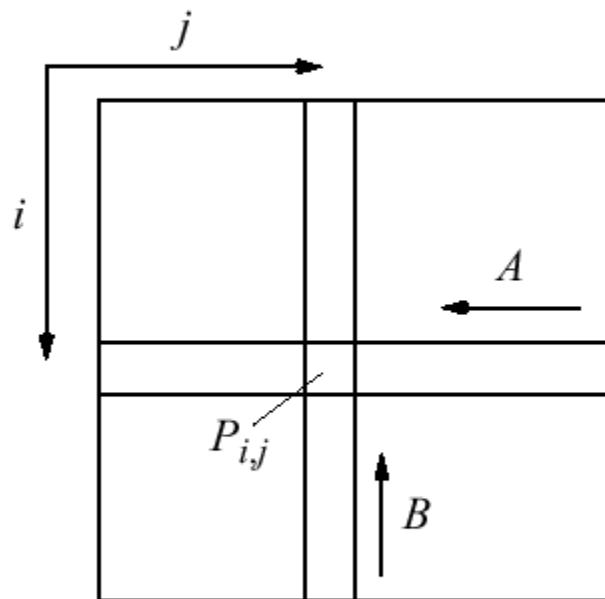
Mesh Implementations

Cannon's Algorithm

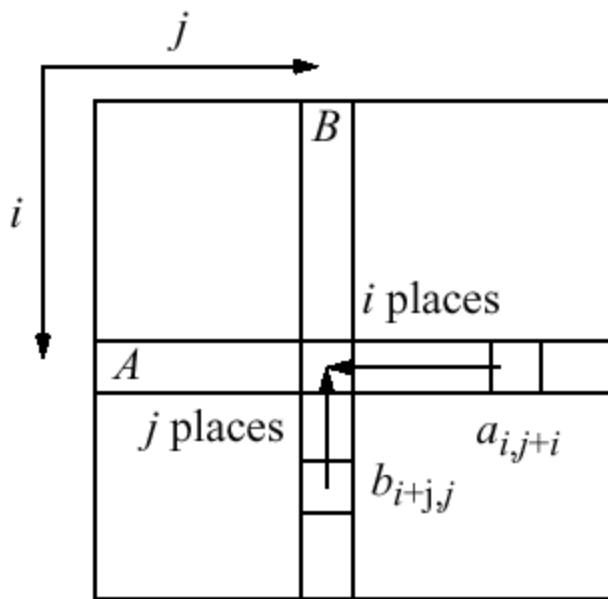
Uses a mesh of processors with wraparound connections (a torus) to shift the A elements (or submatrices) left and the B elements (or submatrices) up.

1. Initially processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ ($0 \leq i < n$, $0 \leq k < n$).
2. Elements are moved from their initial position to an “aligned” position. The complete i th row of A is shifted i places left and the complete j th column of B is shifted j places upward. This has the effect of placing the element $a_{i,j+i}$ and the element $b_{i+j,j}$ in processor $P_{i,j}$. These elements are a pair of those required in the accumulation of $c_{i,j}$.
3. Each processor, $P_{i,j}$, multiplies its elements.
4. The i th row of A is shifted one place right, and the j th column of B is shifted one place upward. This has the effect of bringing together the adjacent elements of A and B, which will also be required in the accumulation.
5. Each processor, $P_{i,j}$, multiplies the elements brought to it and adds the result to the accumulating sum.
6. Step 4 and 5 are repeated until the final result is obtained ($n - 1$ shifts with n rows and n columns of elements).

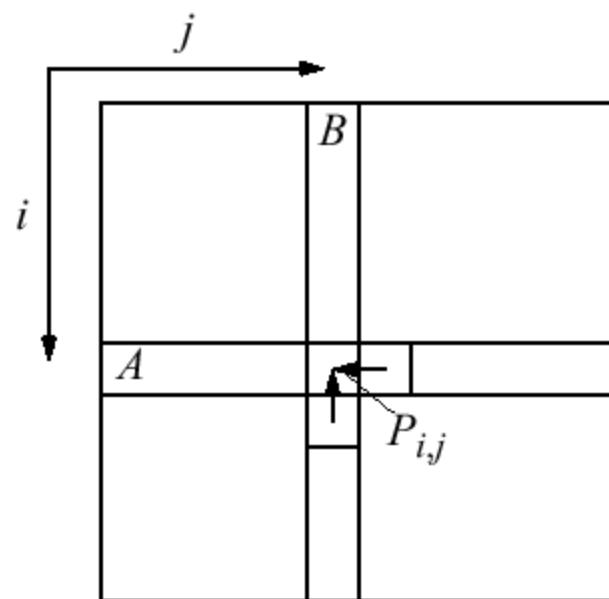
Movement of *A* and *B* elements



Step 2 — Alignment of elements of A and B

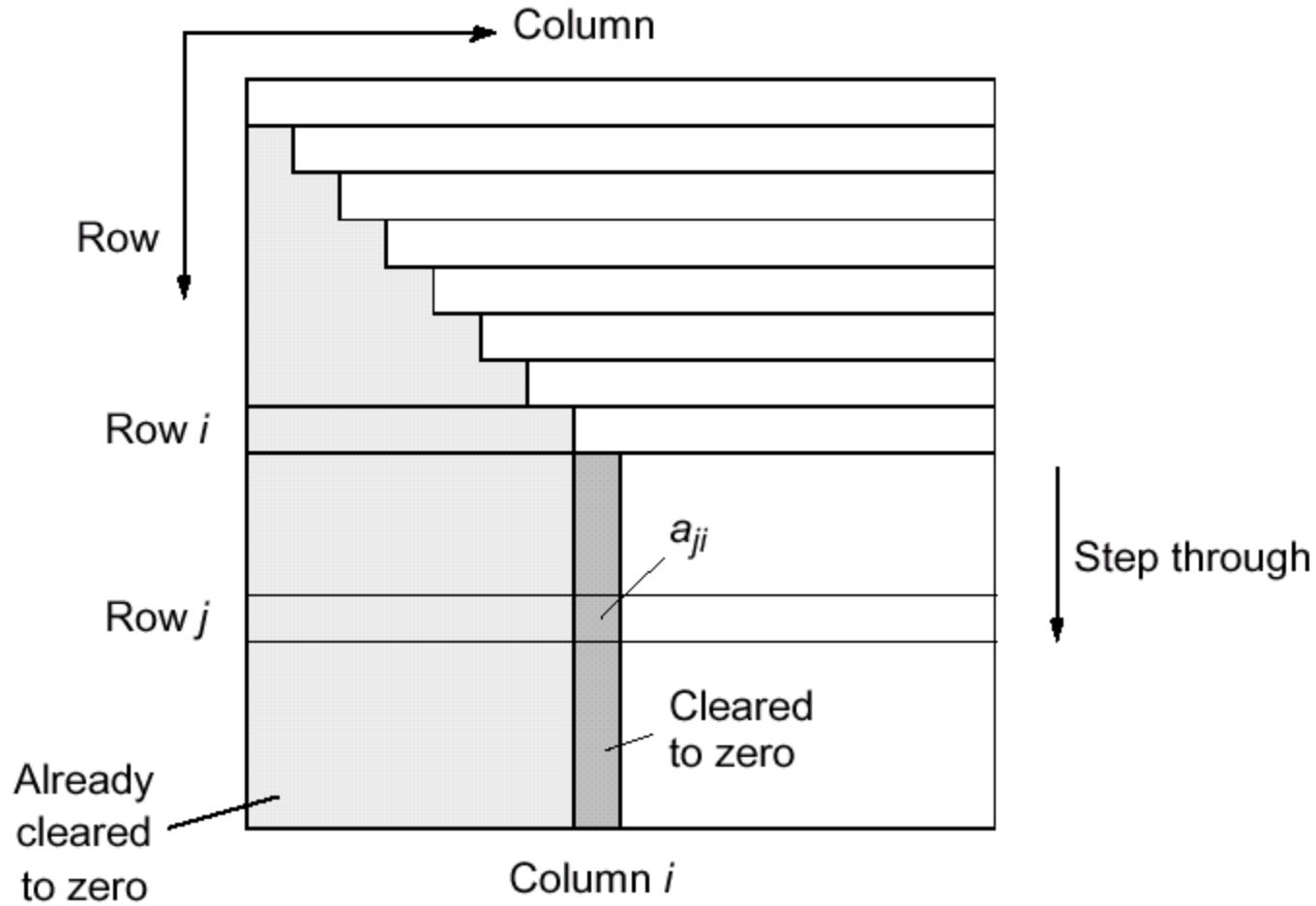


Step 4 — One-place shift of elements of A and B



Solving a System of Linear Equations

Gaussian elimination



Gaussian Elimination

Convert general system of linear equations into triangular system of equations. Then be solved by Back Substitution.

Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

Starts at the first row and works toward the bottom row. At the i th row, each row j below the i th row is replaced by row $j + (\text{row } i) \left(-\frac{a_{j,i}}{a_{i,i}} \right)$. The constant used for row j is $-\frac{a_{j,i}}{a_{i,i}}$. Has the effect of making all the elements in the i th column below the i th row zero because

$$a_{j,i} = a_{j,i} + a_{i,i} \left(\frac{-a_{j,i}}{a_{i,i}} \right) = 0$$

Partial Pivoting

If $a_{i,i}$ is zero or close to zero, we will not be able to compute the quantity $-a_{j,i}/a_{i,i}$.

Procedure must be modified into so-called *partial pivoting* by swapping the i th row with the row below it that has the largest absolute element in the i th column of any of the rows below the i th row if there is one. (Reordering equations will not affect the system.)

In the following, we will not consider partial pivoting.

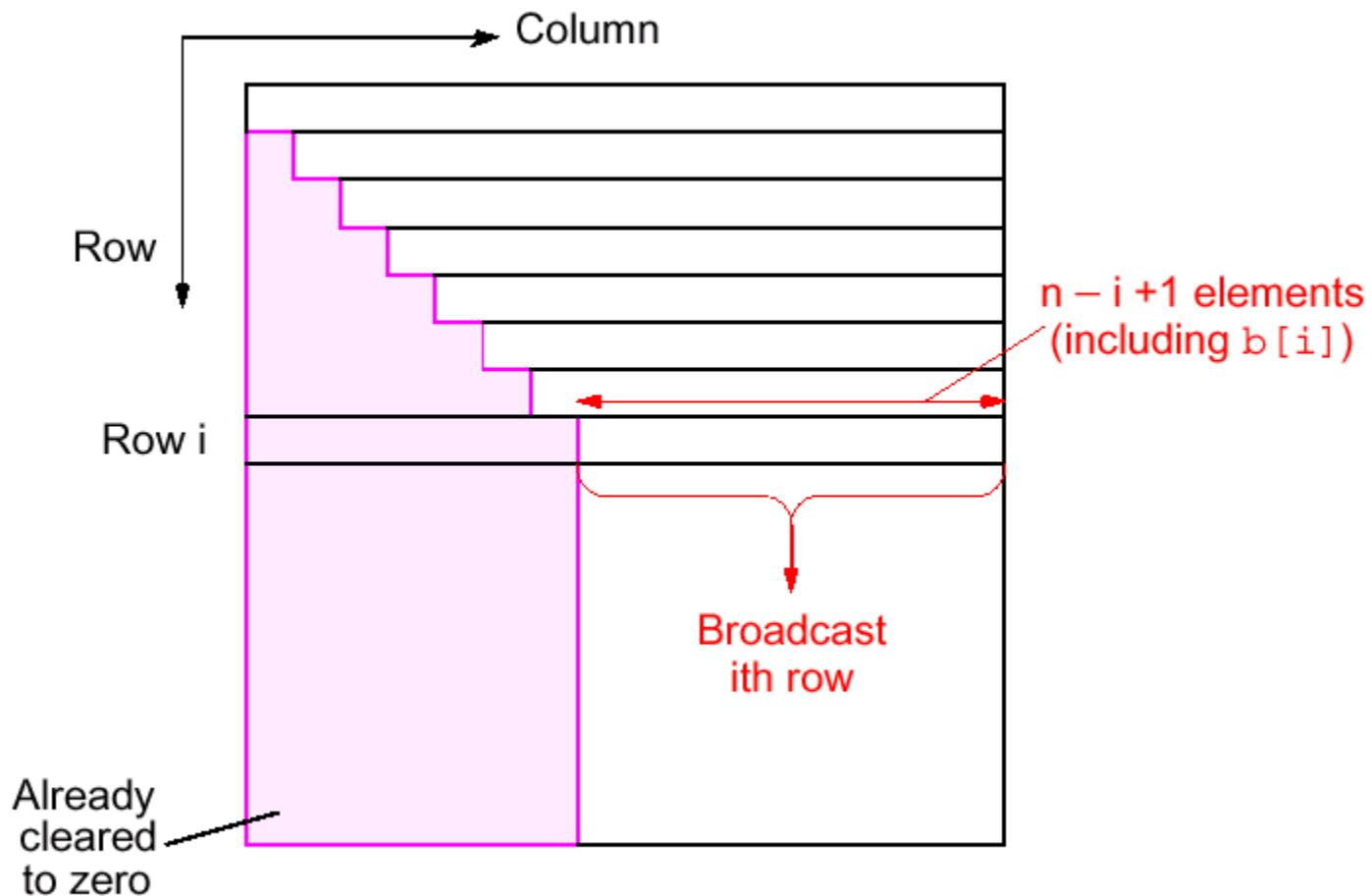
Sequential Code

Without partial pivoting:

```
for (i = 0; i < n-1; i++) /* for each row, except last */
    for (j = i+1; j < n; j++) /*step thro subsequent rows */
        m = a[j][i]/a[i][i]; /* Compute multiplier */
        for (k = i; k < n; k++)/*last n-i-1 elements of row j*/
            a[j][k] = a[j][k] - a[i][k] * m;
        b[j] = b[j] - b[i] * m;/* modify right side */
}
```

The time complexity is $O(n^3)$.

Parallel Implementation



Analysis Communication

$n - 1$ broadcasts performed sequentially. i th broadcast contains $n - i + 1$ elements.

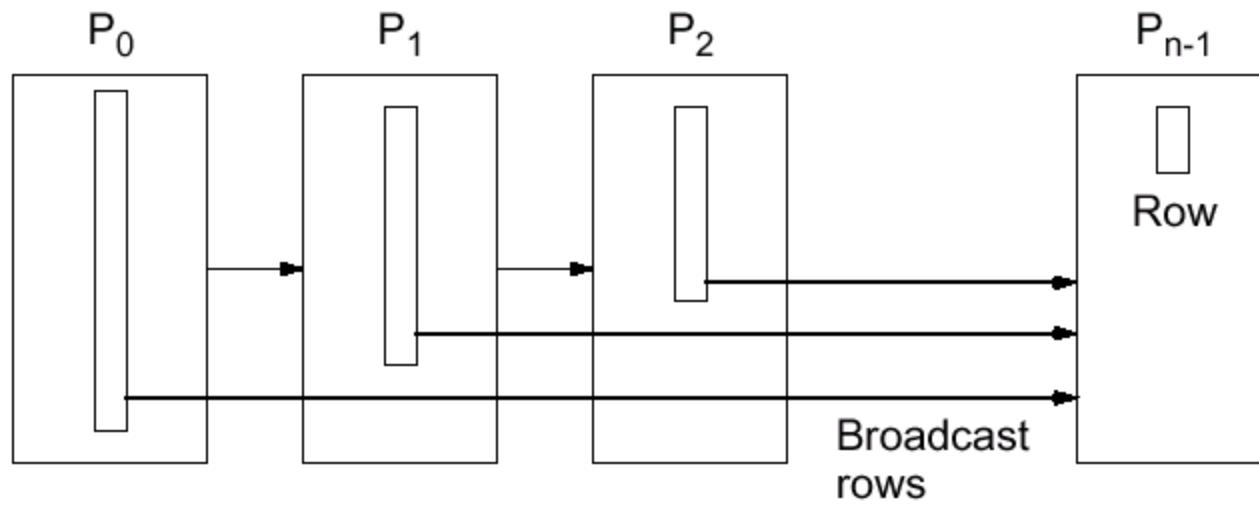
Time complexity of $O(n^2)$ (see textbook)

Computation

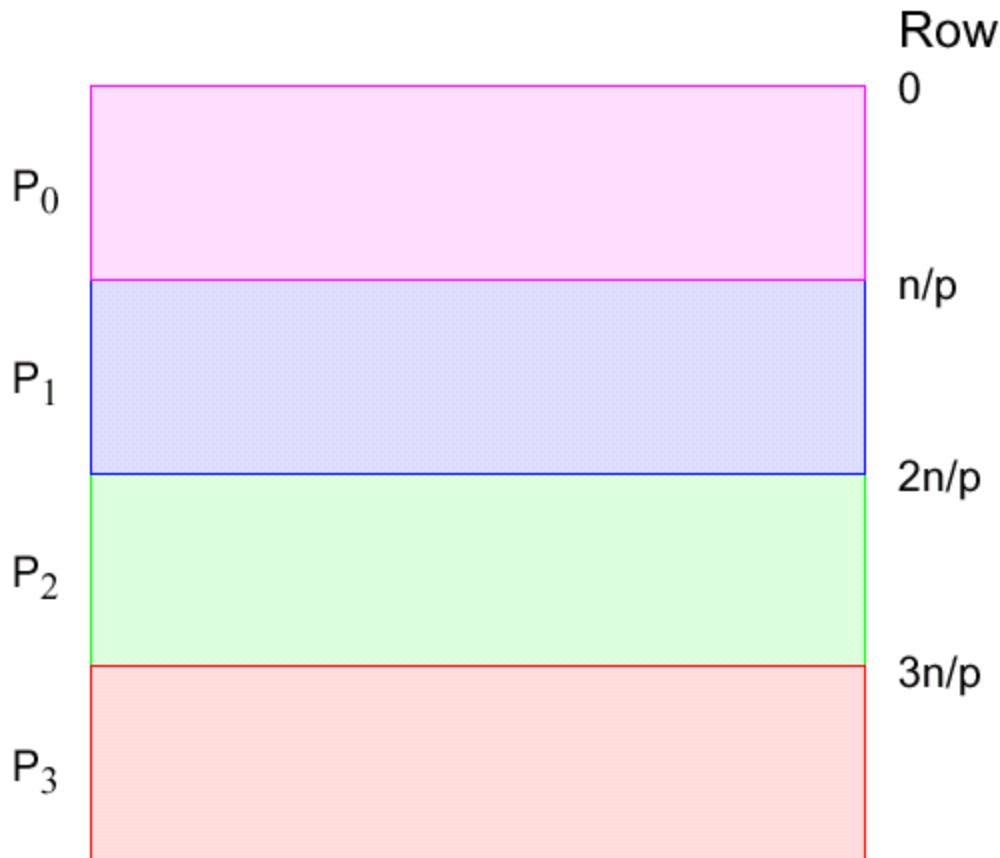
After row broadcast, each processor P_j beyond broadcast processor P_i will compute its multiplier, and operate upon $n - j + 2$ elements of its row. Ignoring the computation of the multiplier, there are $n - j + 2$ multiplications and $n - j + 2$ subtractions.

Time complexity of $O(n^2)$ (see textbook).

Pipeline implementation of Gaussian elimination



Strip Partitioning



Poor processor allocation! Processors do not participate in computation after their last row is processed.

Cyclic-Striped Partitioning

An alternative which equalizes the processor workload:

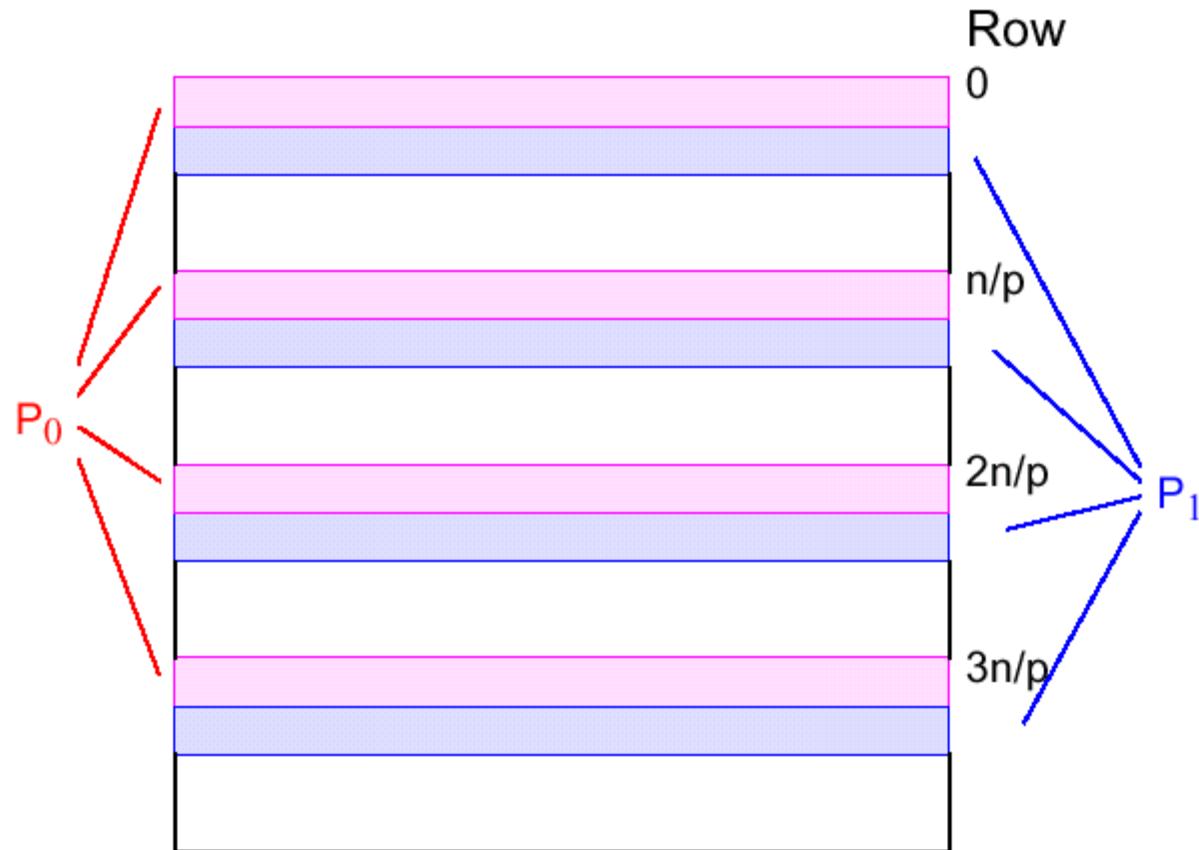


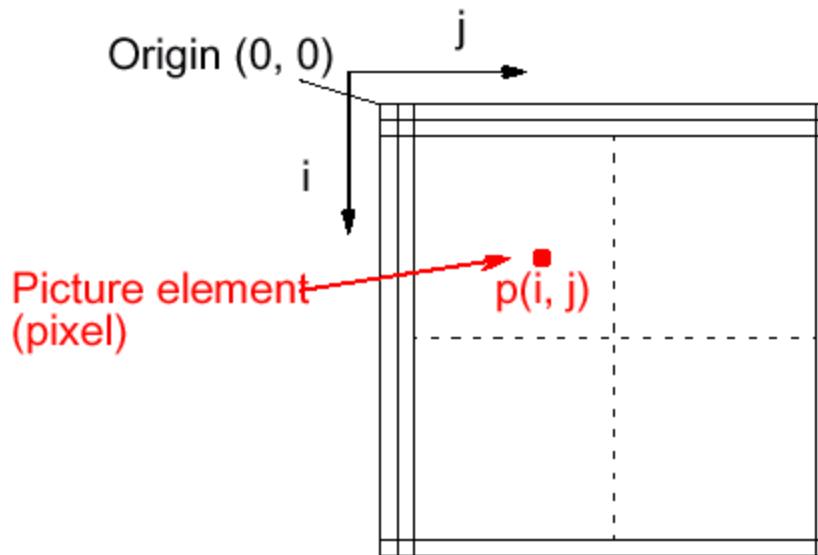
Image Processing

Application area chosen because it has very good parallelism and interesting output.

Low-level Image Processing

Operates directly on stored image to improve/enhance it.

Stored image consists of two-dimensional array of *pixels* (picture elements):



Many low-level image-processing operations assume monochrome images and refer to pixels as having *gray level* values or *intensities*.

Computational Requirements

Suppose a pixmap has 1024×1024 pixels and 8-bit pixels.

Storage requirement is 2^{20} bytes (1 Mbytes)

Suppose each pixel must be operated upon just once.

Then 2^{20} operations are needed in the time of one frame.

At 10^{-8} second/operation (10ns/operation), this would take 10 ms.

In real-time applications, the speed of computation must be at the frame rate (typically 60–85 frames/second).

All pixels in the image must be processed in the time of one frame; that is, in 12–16 ms.

Typically, many high-complexity operations must be performed, not just one operation.

Point Processing

Operations that produce output based upon value of a single pixel.

Thresholding

Pixels with values above predetermined threshold value kept and others below threshold reduced to 0. Given a pixel, x_i , operation on each pixel is

if ($x_i < \text{threshold}$) $x_i = 0$; else $x_i = 1$;

Contrast Stretching

Range of gray level values extended to make details more visible.
Given pixel of value x_i within range x_l and x_h , the contrast stretched to the range x_H to x_L by multiplying x_i by

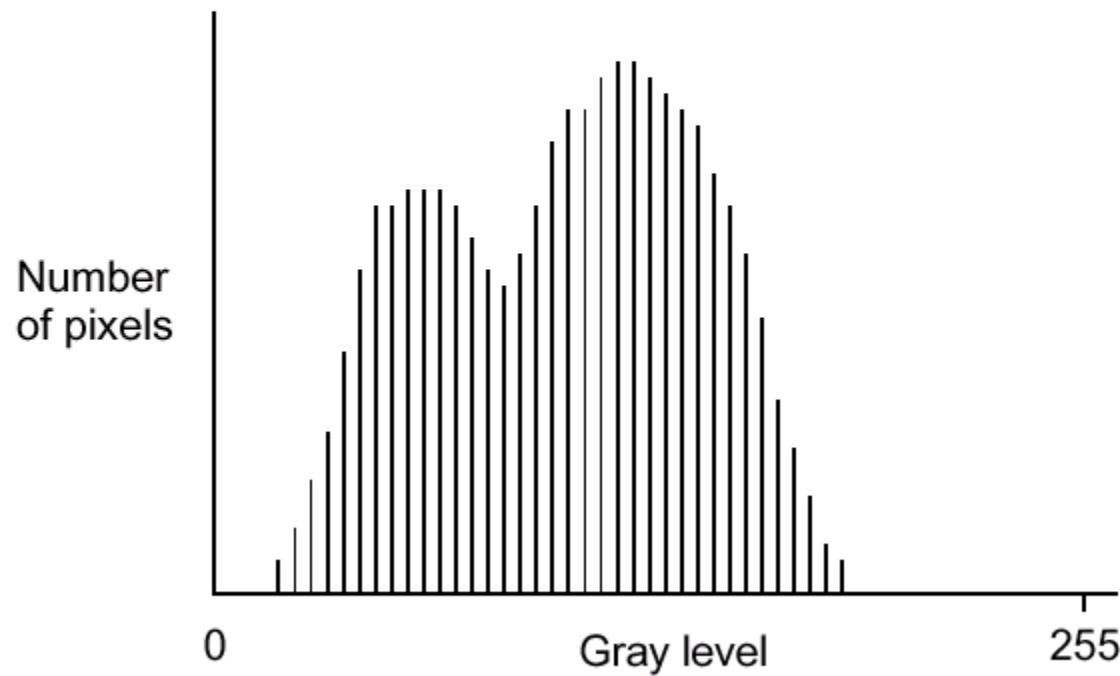
$$x_i = (x_i - x_l) \left(\frac{x_H - x_L}{x_h - x_l} \right) + x_L$$

Gray Level Reduction

Number of bits used to represent the gray level reduced. Simple method would be to truncate the lesser significant bits.

Histogram

Shows the number of pixels in the image at each gray level:



Sequential code

```
for(i = 0; i < height_max; x++)
    for(j = 0; j < width_max; y++)
        hist[p[i][j]] = hist[p[i][j]] + 1;
```

where the pixels are contained in the array `p[][]` and `hist[k]` will hold the number of pixels having the k th gray level.

Similar to adding numbers to an accumulating sum and similar parallel solutions can be used for computing histograms.

Smoothing, Sharpening, and Noise Reduction

Smoothing suppresses large fluctuations in intensity over the image area and can be achieved by reducing the high-frequency content.

Sharpening accentuates the transitions, enhancing the detail, and can be achieved by two ways.

Noise reduction suppresses a noise signal present in the image.

Often requires a local operation with access to a group of pixels around the pixel to be updated. A common group size is 3×3 :

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

Mean

A simple smoothing technique is to take the *mean* or *average* of a group of pixels as the new value of the central pixel.

Given a 3×3 group, the computation is

$$x_4' = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}$$

where x_4' is the new value for x_4 .

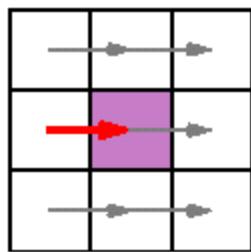
Sequential Code

Nine steps to compute the average for each pixel, or $9n$ for n pixels.

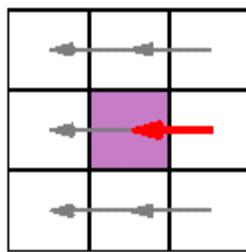
A sequential time complexity of $O(n)$.

Parallel Code

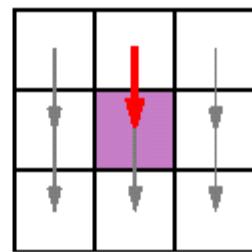
Number of steps can be reduced by separating the computation into four data transfer steps in lock-step data-parallel fashion.



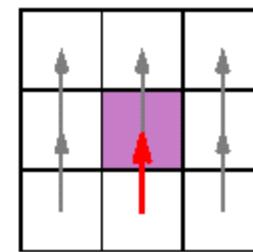
Step 1
Each pixel
adds pixel
from left



Step 2
Each pixel
adds pixel from
right

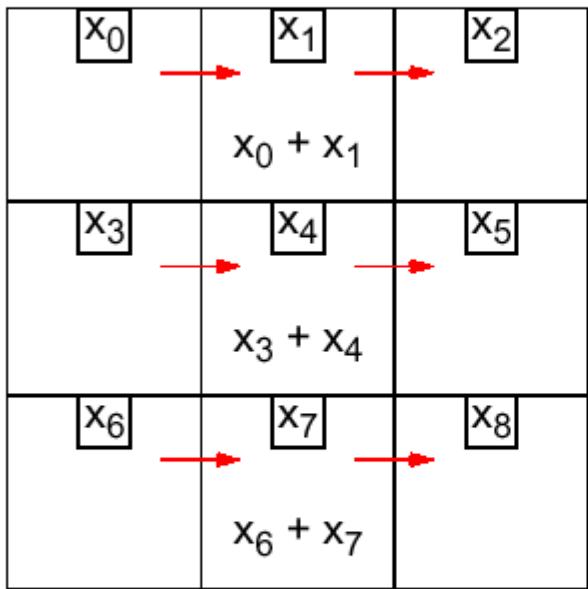


Step 3
Each pixel adds
pixel from above

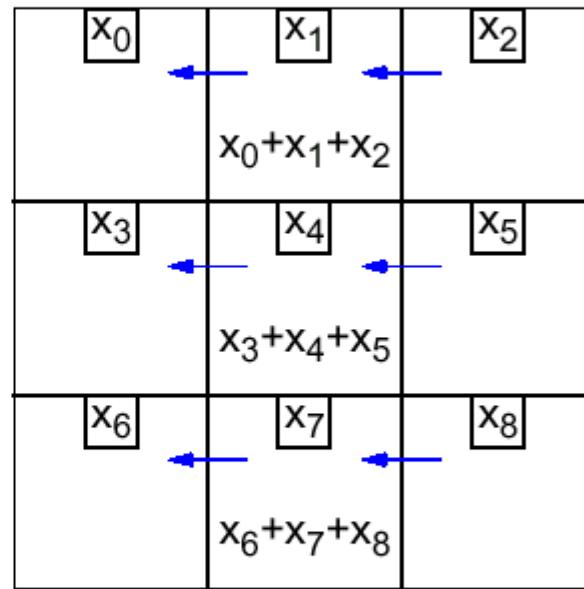


Step 4
Each pixel adds
pixel from below

Parallel Mean Data Accumulation



(a) Step 1



(b) Step 2

x_0	x_1	x_2
$x_0 + x_1 + x_2$		
x_3	x_4	x_5
$x_0 + x_1 + x_2$ $x_3 + x_4 + x_5$		
x_6	x_7	x_8
$x_6 + x_7 + x_8$		

(c) Step 3

x_0	x_1	x_2
$x_0 + x_1 + x_2$		
x_3	x_4	x_5
$x_0 + x_1 + x_2$ $x_3 + x_4 + x_5$ $x_6 + x_7 + x_8$		
x_6	x_7	x_8
$x_6 + x_7 + x_8$		

(d) Step 4

Median

Sequential Code

Median can be found by ordering pixel values from smallest to largest and choosing center pixel value (assuming an odd number of pixels).

With a 3×3 group, suppose values in ascending order are $y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7$, and y_8 . The median is y_4 .

Suggests that all the values must first be sorted, and then fifth element taken.

Using bubble sort, in which the lesser values found first in order, sorting could, in fact, be terminated after fifth lowest value obtained.

Number of steps given by $8 + 7 + 6 + 5 + 4 = 30$ steps, or $30n$ for n pixels.

Parallel Code

An Approximate Sorting Algorithm

First, a compare-and-exchange operation performed on each of the rows, requiring three steps. For the i th row, we have

$$p_{i,j-1} \leftrightarrow p_{i,j}$$

$$p_{i,j} \leftrightarrow p_{i,j+1}$$

$$p_{i,j-1} \leftrightarrow p_{i,j}$$

where \leftrightarrow means “compare and exchange if left gray level greater than right gray level”. Then done on columns:

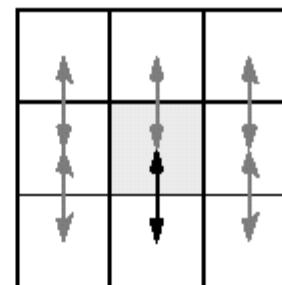
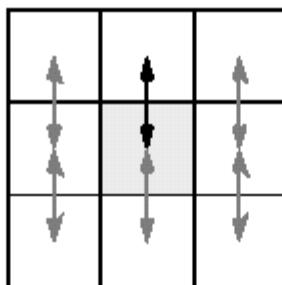
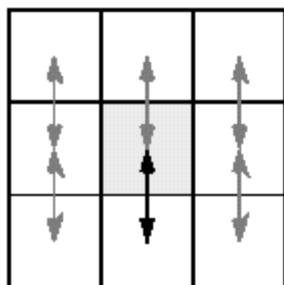
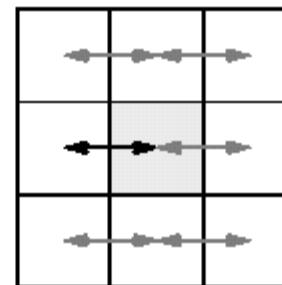
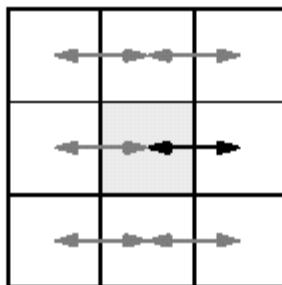
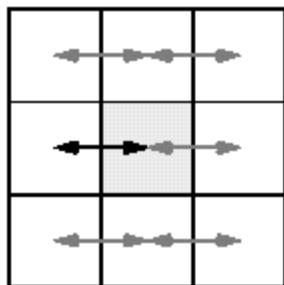
$$p_{i-1,j} \leftrightarrow p_{i,j}$$

$$p_{i,j} \leftrightarrow p_{i+1,j}$$

$$p_{i-1,j} \leftrightarrow p_{i,j}$$

Value in $p_{i,j}$ taken to be fifth largest pixel value. Does not always select fifth largest value. Reasonable approximation. Six steps.

Approximate median algorithm requiring six steps



Largest
in row

Next largest
in row

Next largest
in column

Weighted Masks

The mean method could be described by a weighted 3×3 mask.

Suppose the weights are $w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7$, and w_8 , and pixel values are $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$, and x_8 .

The new center pixel value, x'_4 , is given by

$$x'_4 = \frac{w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8}{k}$$

Scale factor, $1/k$, set to maintain correct grayscale balance.

Often, k is given by $w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7$.

Using a 3×3 Weighted Mask

Mask			Pixels			Result		
w_0	w_1	w_2	\otimes	x_0	x_1	x_2	$=$	
w_3	w_4	w_5		x_3	x_4	x_5		x_4'
w_6	w_7	w_8		x_6	x_7	x_8		

The summation of products, $w_i x_i$, from two functions w and x is the (discrete) *cross-correlation* of f with w (written as $f \otimes w$).

Mask to compute mean

$$x_4' = \frac{x_4 - x_0 - x_1 - x_2 - x_3 - x_5 - x_6 - x_7 - x_8}{9}$$

$k = 9$

1	1	1
1	1	1
1	1	1

A noise reduction mask

$$x_4' = \frac{8x_4 + x_0 + x_1 + x_2 + x_3 + x_5 + x_6 + x_7 + x_8}{16}$$

$k = 16$

1	1	1
1	8	1
1	1	1

High-pass sharpening filter mask

$$x_4' = \frac{8x_4 - x_0 - x_1 - x_2 - x_3 - x_5 - x_6 - x_7 - x_8}{9}$$

$k = 9$

-1	-1	-1
-1	8	-1
-1	-1	-1

Edge Detection

Highlighting edges of object where an edge is a significant change in gray level intensity.

Gradient and Magnitude

With a one-dimension gray level function, $f(x)$, first derivative, $\partial f / \partial x$, measures the gradient..

Edge recognized by a positive-going or negative-going spike at a transition.

Edge Detection using Differentiation

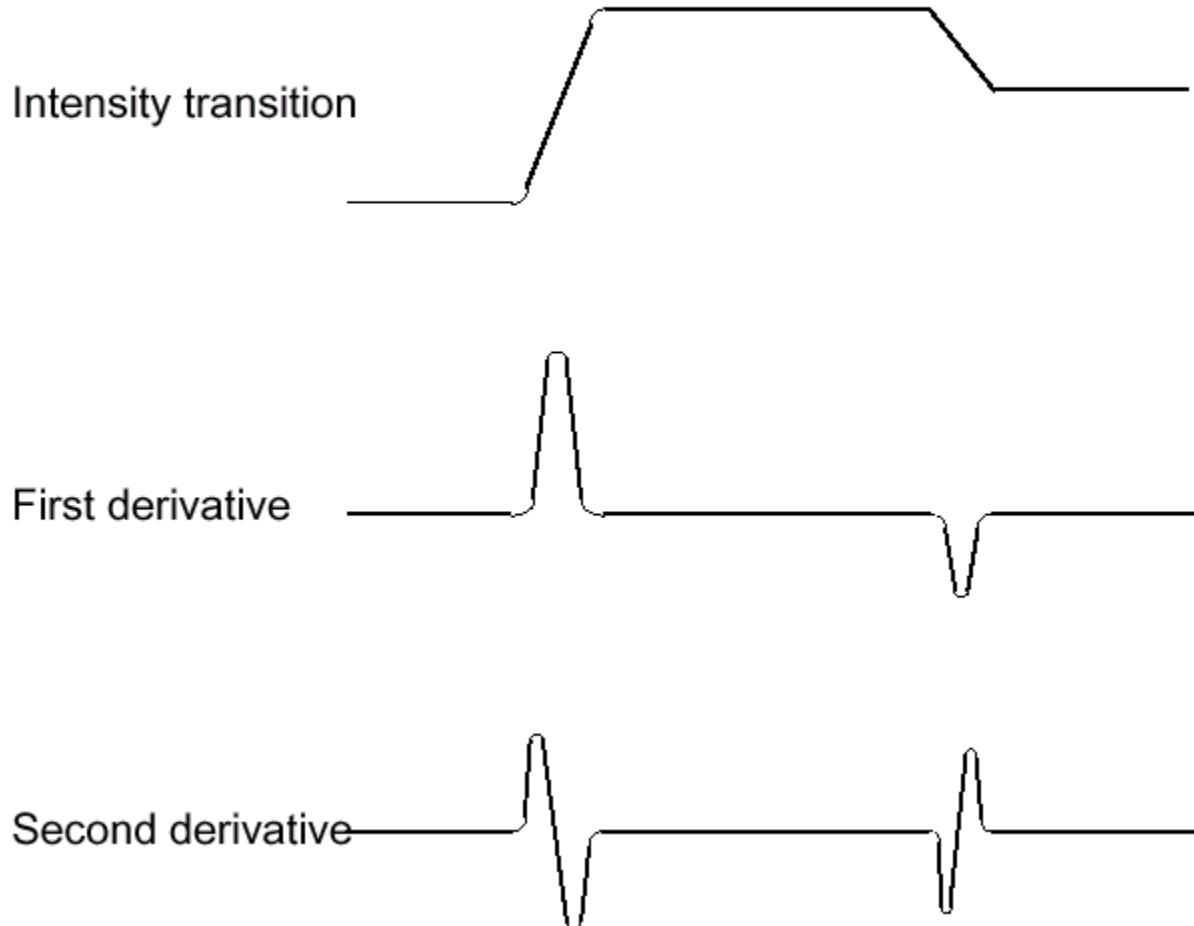


Image Function

A two-dimensional discretized gray level function, $f(x,y)$.

Gradient (magnitude)

$$\nabla f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

Gradient Direction

$$\phi(x, y) = \tan^{-1} \left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}} \right)$$

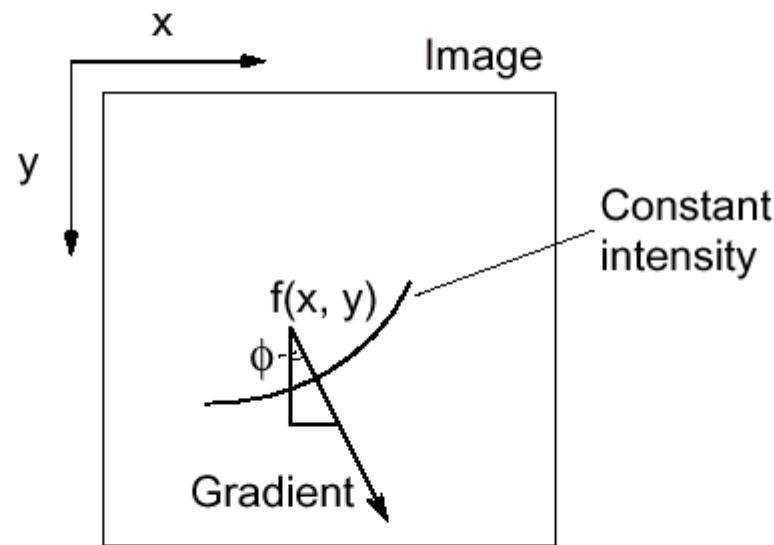
where ϕ is the angle with respect to the y -axis.

Gradient can be approximated to

$$\nabla f \approx \left| \frac{\partial f}{\partial y} \right| + \left| \frac{\partial f}{\partial x} \right|$$

for reduced computational effort

Gray Level Gradient and Direction



Edge Detection of Image Function

Image is a discrete two-dimensional function.

Derivative approximated by differences:

$\partial f / \partial x$ is difference in x-direction

$\partial f / \partial y$ is difference in y-direction

Edge Detection Masks

Might consider computing the approximate gradient using x_5 and x_3 (to get $\partial f / \partial x$) and x_7 and x_1 (to get $\partial f / \partial y$); i.e.,

$$\frac{\partial f}{\partial x} \approx x_5 - x_3$$

$$\frac{\partial f}{\partial y} \approx x_7 - x_1$$

so that

$$\nabla f \approx |x_7 - x_1| + |x_5 - x_3|$$

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

Two masks needed, one to obtain $x_7 - x_1$ and one to obtain $x_5 - x_3$.

The absolute values of results of each mask added together.

0	-1	0
0	0	0
0	1	0

0	0	0
-1	0	1
0	0	0

Prewitt Operator

The approximate gradient obtained from

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + (x_7 - x_1) + (x_8 - x_2)$$

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + (x_5 - x_3) + (x_8 - x_6)$$

x ₀	x ₁	x ₂
x ₃	x ₄	x ₅
x ₆	x ₇	x ₈

Then

$$\nabla f \approx |x_6 - x_0 + x_7 - x_1 + x_8 - x_2| + |x_2 - x_0 + x_5 - x_3 + x_8 - x_6|$$

which requires using the two 3×3 masks.

Prewitt operator

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

Sobel Operator

Derivatives are approximated to

$$\frac{\partial f}{\partial y} \approx (x_6 + 2x_7 + x_8) - (x_0 + 2x_1 + x_2)$$

$$\frac{\partial f}{\partial x} \approx (x_2 + 2x_5 + x_8) - (x_0 + 2x_3 + x_6)$$

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

Operators implementing first derivatives will tend to enhance noise.

However, the Sobel operator also has a smoothing action.

Sobel Operator

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Edge Detection with Sobel Operator



(a) Original image (Annabel)



(b) Effect of Sobel operator

Hough Transform

Purpose is to find the parameters of equations of lines that most likely fit sets of pixels in an image.

A line is described by the equation

$$y = ax + b$$

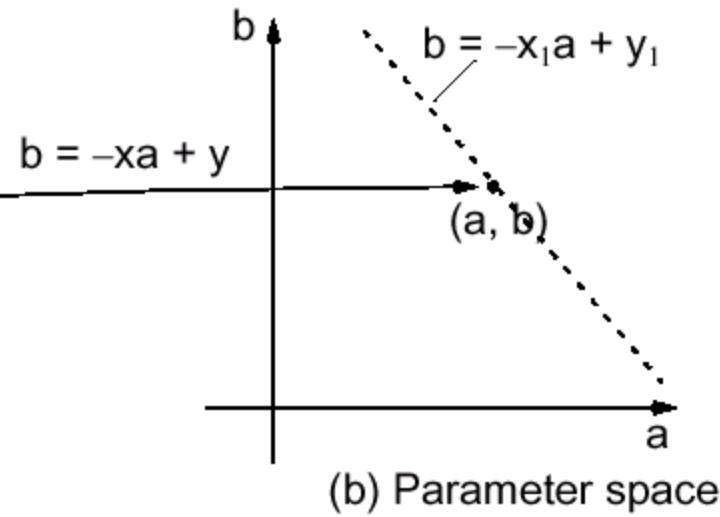
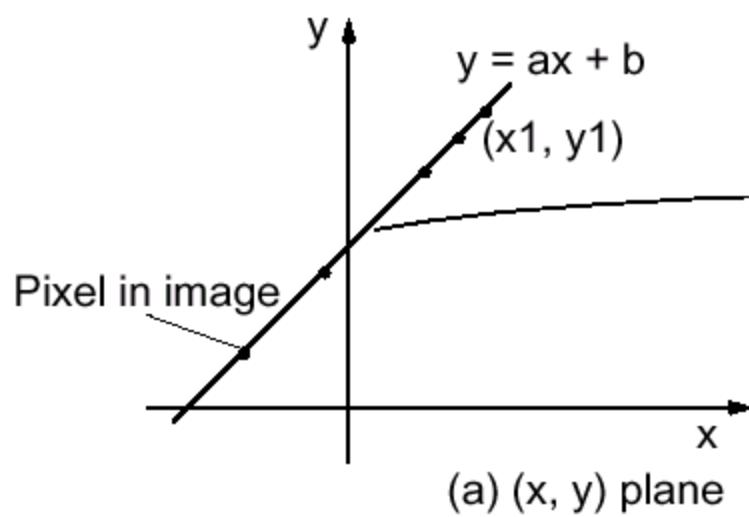
where the parameters, a and b , uniquely describe the particular line, a the slope and b the intercept on the y -axis.

A search for those lines with the most pixels mapped onto them would be computationally prohibitively expensive [$O(n^3)$].

Suppose the equation of the line is rearranged as:

$$b = -xa + y$$

Every point that lies on a specific line in the x - y space will map into same point in the a - b space (parameter space).



Finding the Most Likely Lines

In the mapping process, discrete values will be used to a coarse prescribed precision and the computation is rounded to the nearest possible a - b coordinates.

The mapping process is done for every point in the x - y space.

A record is kept of those a - b points that have been obtained by incrementing the corresponding accumulator.

Each accumulator will have the number of pixels that map into a single point in the parameter space.

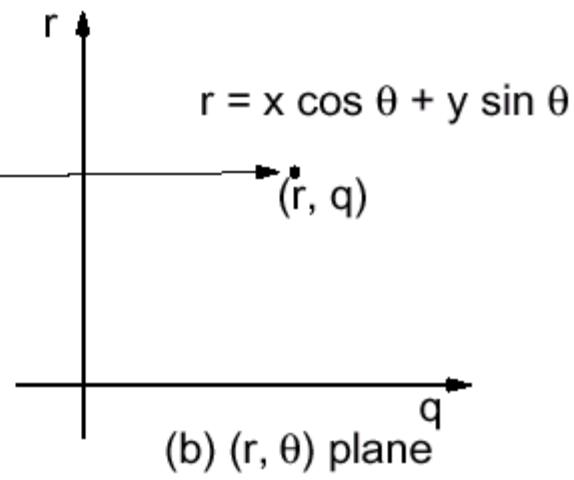
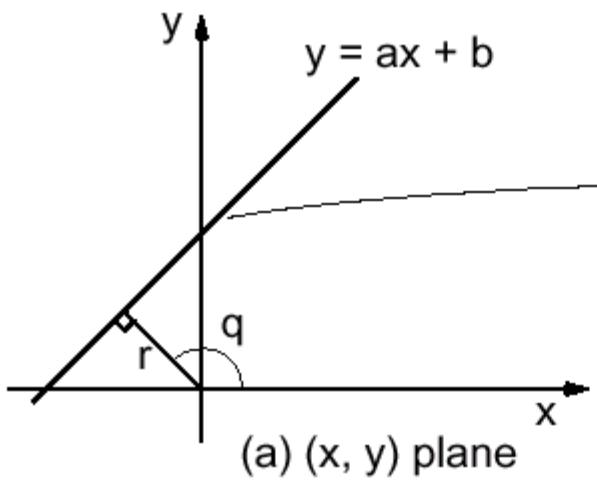
The points in the parameter space with locally maximum numbers of pixels are chosen as lines.

Unfortunately, this method will fail for vertical lines (i.e., with the slope, a , infinite and with the y intercept, b , infinite) and with lines that approach this extreme.

To avoid the problem, line equation rearranged to polar coordinates:

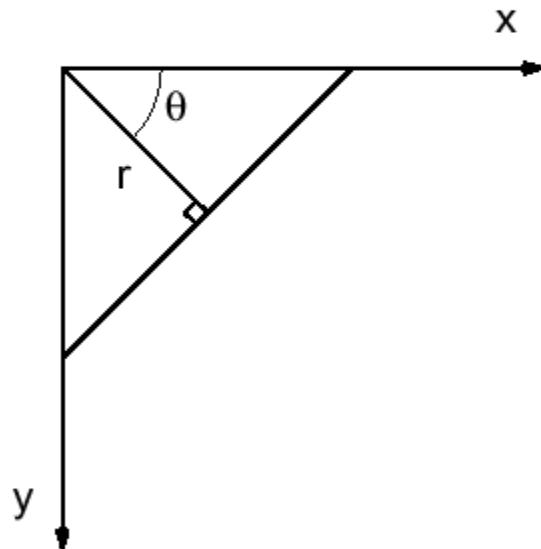
$$r = x \cos \theta + y \sin \theta$$

where r is the perpendicular distance to the origin in the original (x, y) coordinate system and θ is the angle between r and the x -axis. θ very conveniently the gradient angle of line (with respect to x -axis).



Implementation

Assume origin at the top left corner.



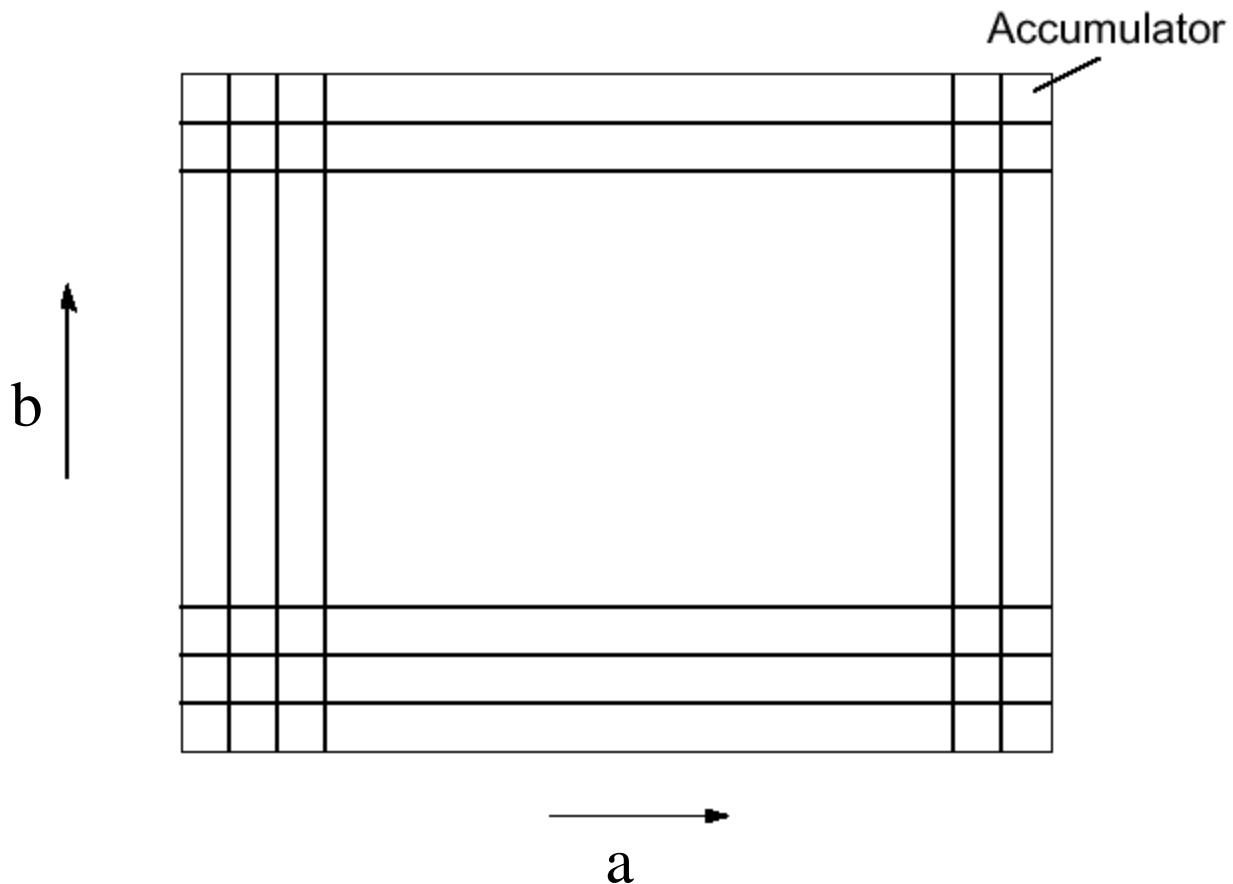
The parameter space divided into small rectangular regions. One accumulator for each region.

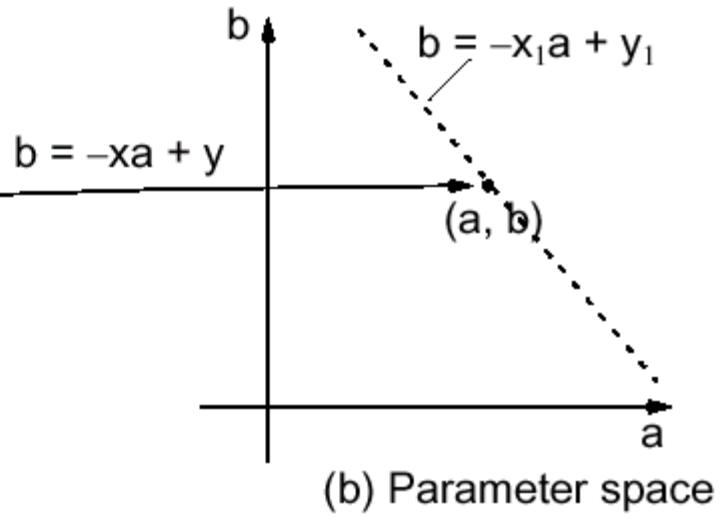
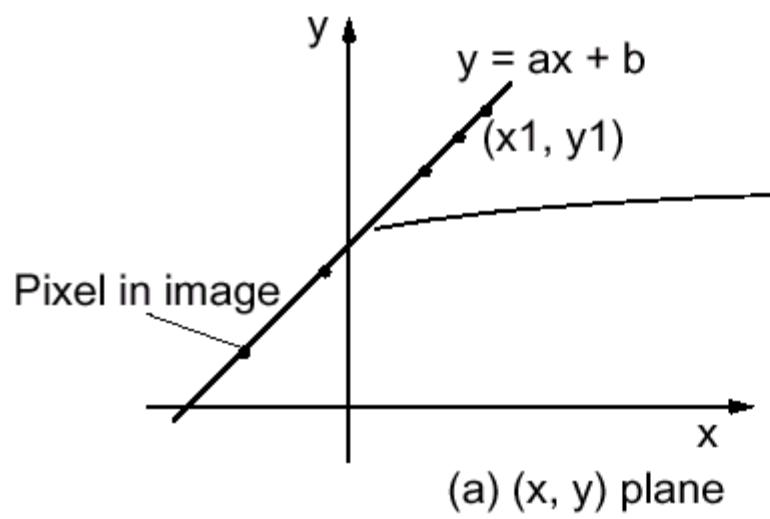
Accumulators of those regions that a pixel maps into incremented.
Process done for all pixels in image.

If all values of θ were tried (i.e., incrementing θ through all its values), computational effort would be given by the number of discrete values of θ , say k intervals. With n pixels the complexity is $O(kn)$.

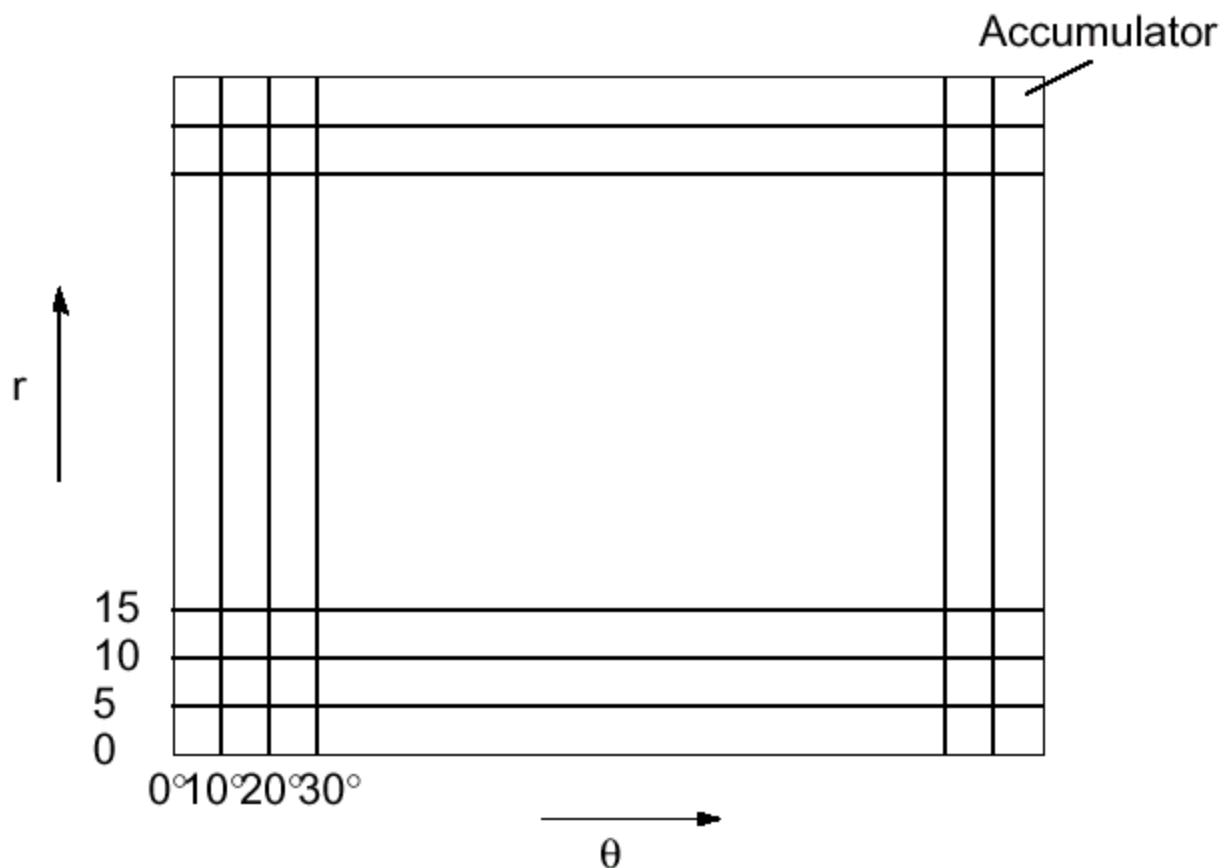
Computational effort can be reduced significantly by limiting range of lines for individual pixels using some criteria. A single value of θ could be selected based upon the gradient of the line.

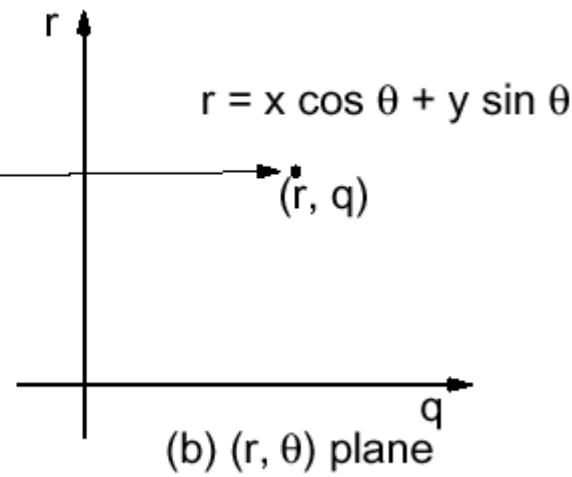
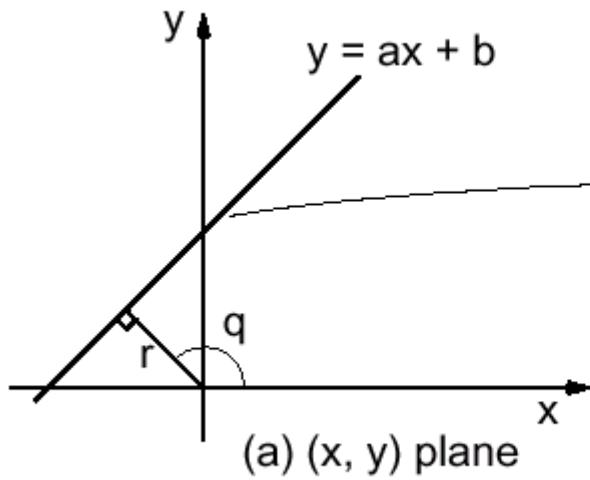
Accumulators





Accumulators, $\text{acc}[r][\theta]$, for Hough Transform





Sequential Code

Sequential code could be of the form

```
for (x = 0; x < xmax; x++)           /* for each pixel */
    for (y = 0; y < ymax; y++) {
        sobel(&x, &y, dx, dy);          /* find x and y gradients */
        magnitude = grad_mag(dx, dy);  /* find magnitude if needed */
        if (magnitude > threshold) {
            theta = grad_dir(dx, dy);  /* atan2() fn */
            theta = theta_quantize(theta);
            r = x * cos(theta) + y * sin(theta);
            r = r_quantize(r);
            acc[r][theta]++;
            append(r, theta, x, y);
        }
    }
```

Parallel Code

Since the computation for each accumulator is independent of the other accumulations, it could be performed simultaneously, although each requires read access to the whole image.

Transformation into the Frequency Domain

Fourier Transform

Many applications in science and engineering. In image processing, Fourier transform used for image enhancement, restoration, and compression.

Image is a two-dimensional discretized function, $f(x, y)$, but first start with one-dimensional case.

For completeness, let us first review results of Fourier series and Fourier transform concepts from first principles.

Fourier Series

The Fourier series is a summation of sine and cosine terms:

$$x(t) = \frac{a_0}{2} + \sum_{j=1}^{\infty} \left(a_j \cos\left(\frac{2\pi jt}{T}\right) + b_j \sin\left(\frac{2\pi jt}{T}\right) \right)$$

T is the period ($1/T = f$, where f is a frequency).

By some mathematical manipulation:

$$x(t) = \sum_{j=-\infty}^{\infty} X_j e^{2\pi i j \frac{t}{T}}$$

where X_j is the j th Fourier coefficient in a complex form and $i = \sqrt{-1}$.
(Fourier coefficients can also be computed from specific integrals.)

Discrete Functions

For functions having a set of N discrete values. Replace integral with summation, leading to the *discrete Fourier transform (DFT)*:

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{-2\pi i \left(\frac{jk}{N}\right)}$$

and *inverse discrete Fourier transform* given by

$$x_k = \sum_{j=0}^{N-1} X_j e^{2\pi i \left(\frac{jk}{N}\right)}$$

for $0 \leq k \leq N - 1$. The N (real) input values, $x_0, x_1, x_2, \dots, x_{N-1}$, produce N (complex) transform values, $X_0, X_1, X_2, \dots, X_{N-1}$.

Fourier Transforms in Image Processing

A two-dimensional Fourier transform is

$$X_{lm} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} x_{jk} e^{-2\pi i \left(\frac{jl}{N} + \frac{km}{M} \right)}$$

where $0 \leq j \leq N - 1$ and $0 \leq k \leq M - 1$.

Assume image is square, where $N = M$.

Equation can be rearranged into

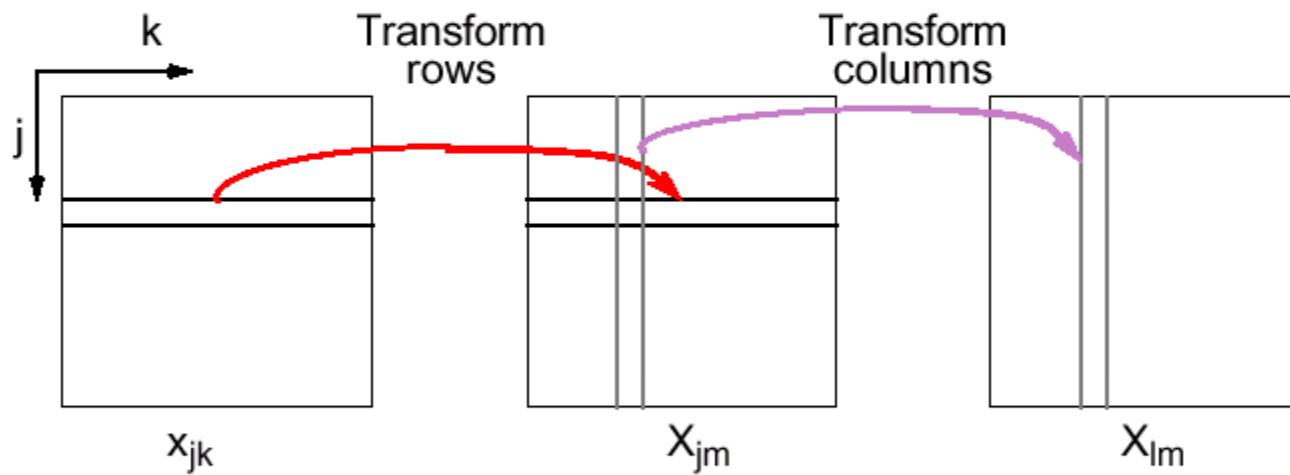
$$X_{lm} = \sum_{j=0}^{N-1} \left[\sum_{k=0}^{N-1} x_{jk} e^{-2\pi i \left(\frac{km}{N} \right)} \right] e^{-2\pi i \left(\frac{jl}{N} \right)}$$

Inner summation a one-dimensional DFT operating on N points of a row to produce a transformed row. Outer summation a one-dimensional DFT operating on N points of a column.

Can be divided into two sequential phases, one operating on rows of elements and one operating on columns:

$$X_{lm} = \sum_{j=0}^{N-1} X_{jm} e^{-2\pi i \left(\frac{jl}{N} \right)}$$

Two-Dimensional DFT



Parallelizing the Discrete Fourier Transform

Starting from

$$X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i \left(\frac{jk}{N}\right)}$$

and using the notation $w = e^{-2\pi i / N}$,

$$X_k = \sum_{j=0}^{N-1} x_j w^{jk}$$

w terms called *twiddle factors*. Each input multiplied by twiddle factor.

Inverse transform can be obtained by replacing w with w^{-1} .

Sequential Code

```
for (k = 0; k < N; k++) { /* for every point */
    X[k] = 0;
    for (j = 0; j < N; j++) /* compute summation */
        X[k] = X[k] + wj * x[j];
}
```

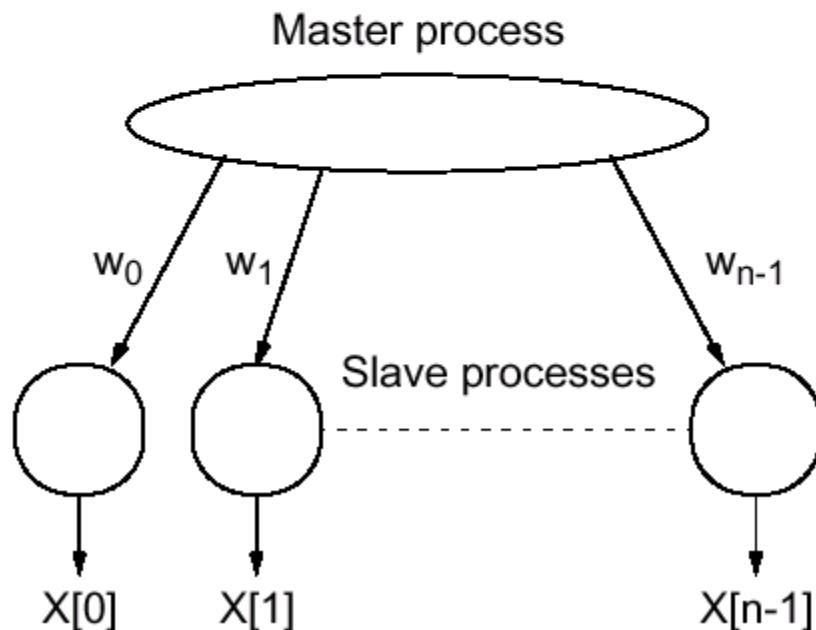
$X[k]$ is k th transformed point, $x[k]$ is k th input, $w = e^{-2\pi i/N}$.
Summation requires complex number arithmetic. Can be rewritten:

```
for (k = 0; k < N; k++) {
    X[k] = 0;
    a = 1;
    for (j = 0; j < N; j++) {
        X[k] = X[k] + a * x[j];
        a = a * wk;
    }
}
```

where a is a temporary variable.

Elementary Master-Slave Implementation

One slave process of N slave processes assigned to produce one transformed value; i.e., k th slave process produces $x[k]$. Parallel time complexity with N (slave) processes is $O(N)$.



$$X_k = \sum_{j=0}^{N-1} x_j w^{jk}$$

DFT as a Matrix-Vector Product

The k th element of discrete Fourier transform given by

$$X_k = x_0w^0 + x_1w^1 + x_2w^2 + x_3w^3 + \dots + x_{N-1}w^{N-1}$$

Whole transform can be described by a matrix-vector product:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_k \\ \vdots \\ X_{N-1} \end{bmatrix} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & w & w^2 & w^3 & \dots & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & \dots & w^{2(N-1)} \\ 1 & w^3 & w^6 & w^9 & \dots & w^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^k & w^{2k} & w^{3k} & \dots & w^{(N-1)k} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \dots & w^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \\ \vdots \\ x_{N-1} \end{bmatrix}$$

(Note $w^0 = 1$.) Hence, parallel methods for matrix-vector product as described in Ch. 10 can be used for discrete Fourier transform.

Fast Fourier Transform

Method of obtaining discrete Fourier transform with a time complexity of $O(N \log N)$ instead of $O(N^2)$.

Let us start with the discrete Fourier transform equation:

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j w^{jk}$$

where $w = e^{-2\pi i/N}$.

Each summation an $N/2$ discrete Fourier transform operating on $N/2$ even points and $N/2$ odd points, respectively.

$$X_k = \frac{1}{2} [X_{\text{even}} + w^k X_{\text{odd}}]$$

for $k = 0, 1, \dots, N - 1$, where X_{even} is the $N/2$ -point DFT of the numbers with even indices, x_0, x_2, x_4, \dots , and X_{odd} is the $N/2$ -point DFT of the numbers with odd indices, x_1, x_3, x_5, \dots .

Now, suppose k is limited to $0, 1, \dots N/2 - 1$, the first $N/2$ values of the total N values. Complete sequence divided into two parts:

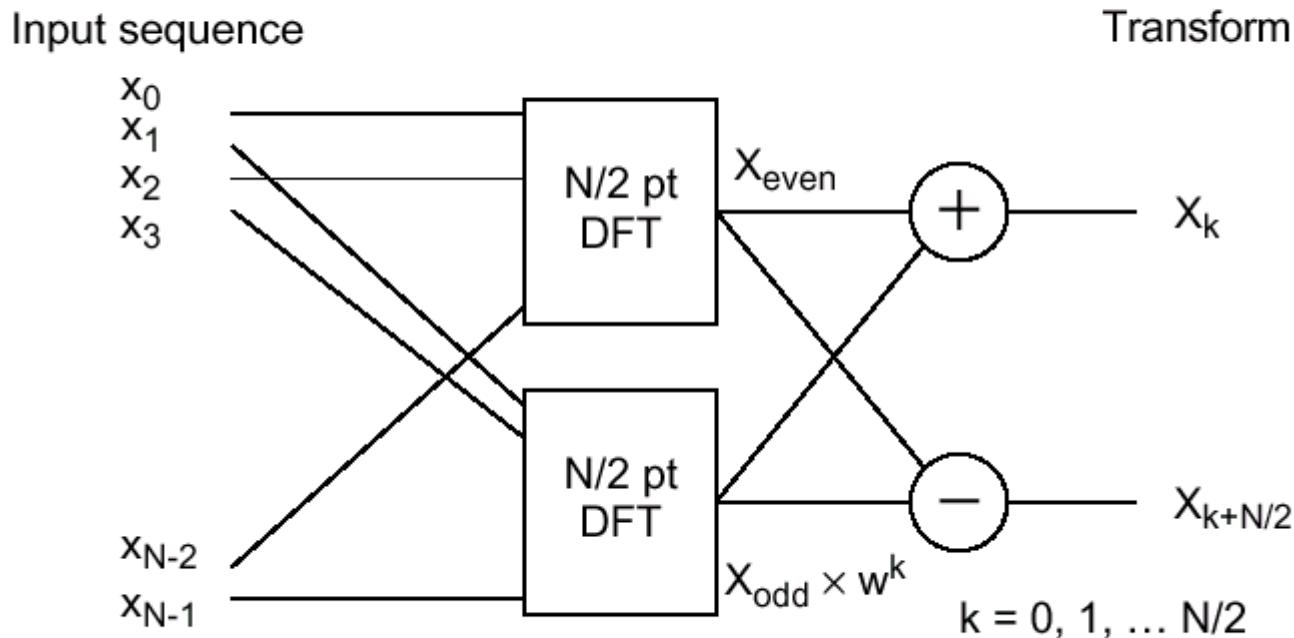
$$X_k = \frac{1}{2} [x_{\text{even}} + w^k x_{\text{odd}}]$$

and

$$X_{k+N/2} = \frac{1}{2} [x_{\text{even}} + w^{k+N/2} x_{\text{odd}}] = \frac{1}{2} [x_{\text{even}} - w^k x_{\text{odd}}]$$

since $w^{k+N/2} = -w^k$, where $0 \leq k < N/2$. Hence, we could compute X_k and $X_{k+N/2}$ using two $N/2$ -point transforms:

Decomposition of N -point DFT into two $N/2$ -point DFTs



\Rightarrow Hypercube !

sequential: $O(N \log N)$, parallel: $O(\log N)$