Parallel Computations & Applications

National Tsing Hua University

2018, Fall Semester

Outline

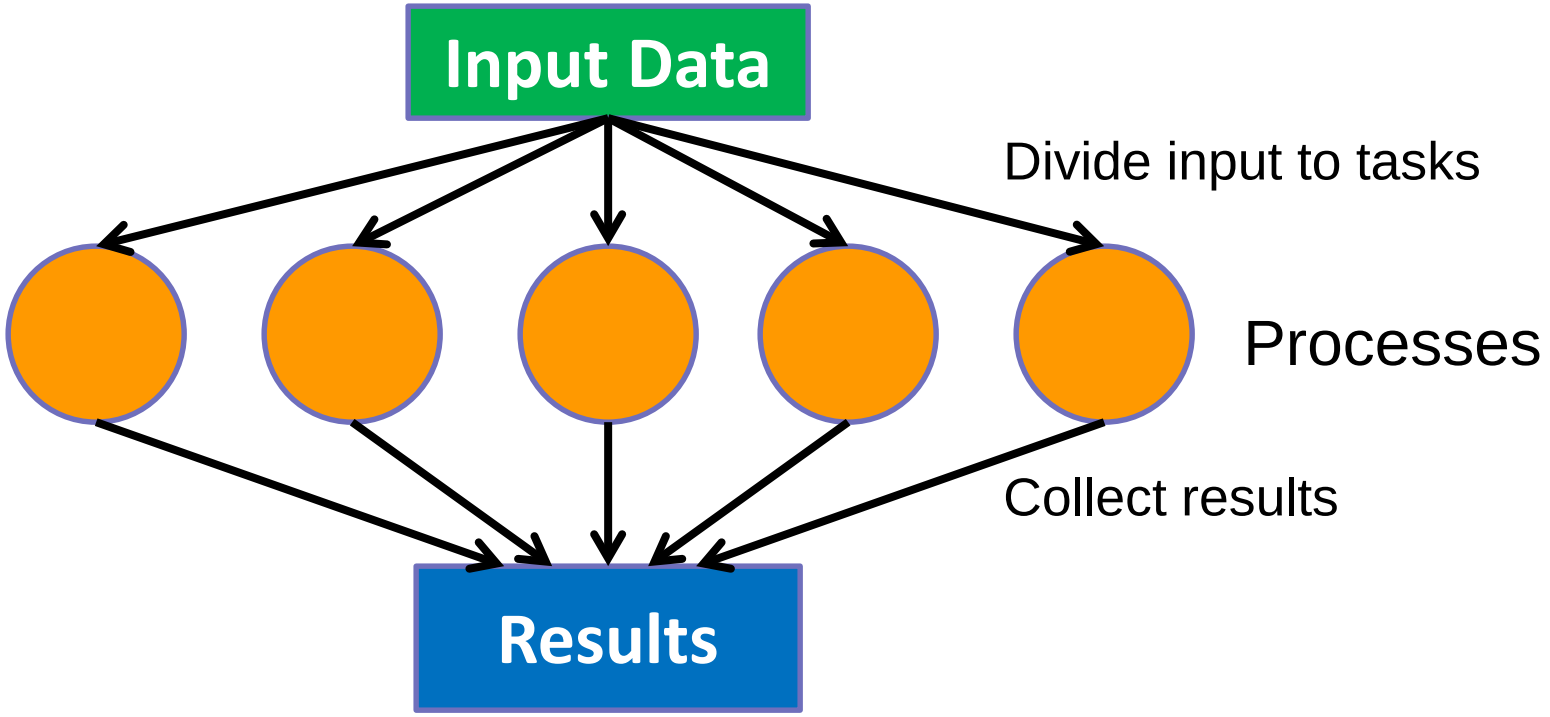
- Embarrassingly Computations
- Load Balancing & Termination
- Divide-And-Conquer Computations
- Pipelined Computations
- Synchronous Computations

Outline

- Embarrassingly Computations
  - Image Transformations
  - Monte Carlo Methods
  - Mandelbrot Set
- Load Balancing & Termination
- Divide-And-Conquer Computations
- Pipelined Computations
- Synchronous Computations

What is Embarrassingly Parallel

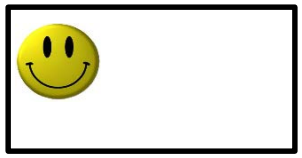
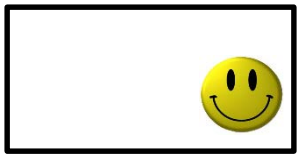
- A computation that can be divided into a number of completely **independent** tasks



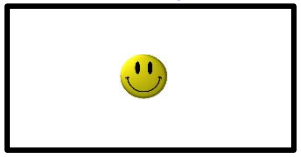
Example 1: Image Transformations

- Low-level image operations:
  - Shifting: object shifted by Δx in the x-dimension and Δy in the y-dimension:

    $$x' = x + \Delta x, \quad y' = y + \Delta y$$

    
    
  - Scaling: object scaled by a factor of Sx in the x-direction and Sy in the y-direction;

    $$x' = xS_x, \quad y' = yS_y$$

    
    
  - Rotation: object rotated through the angle θ about the origin of the coordinate system:

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

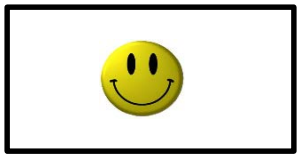
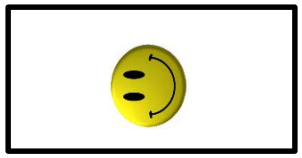
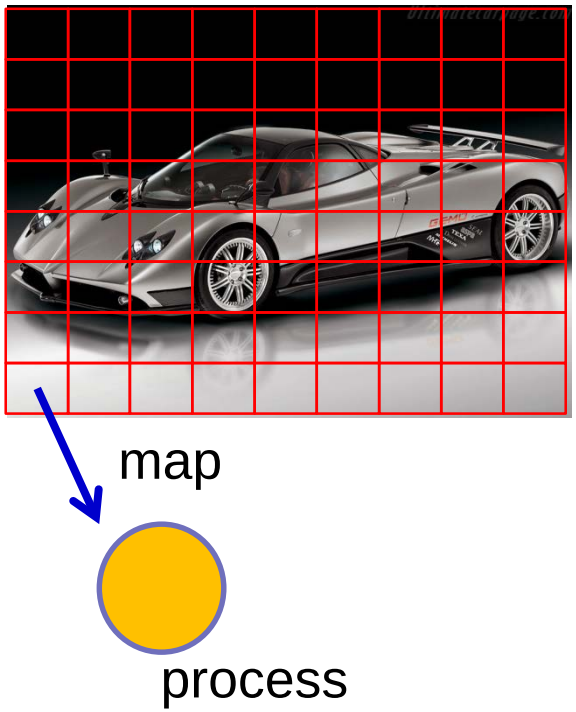
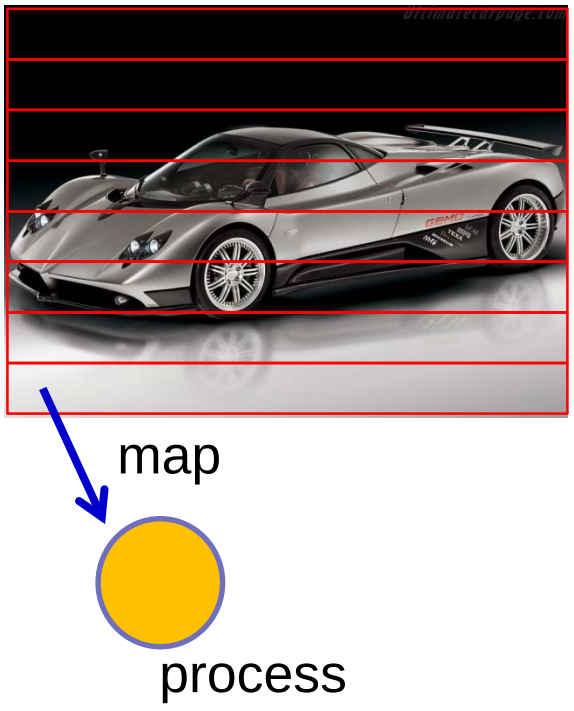
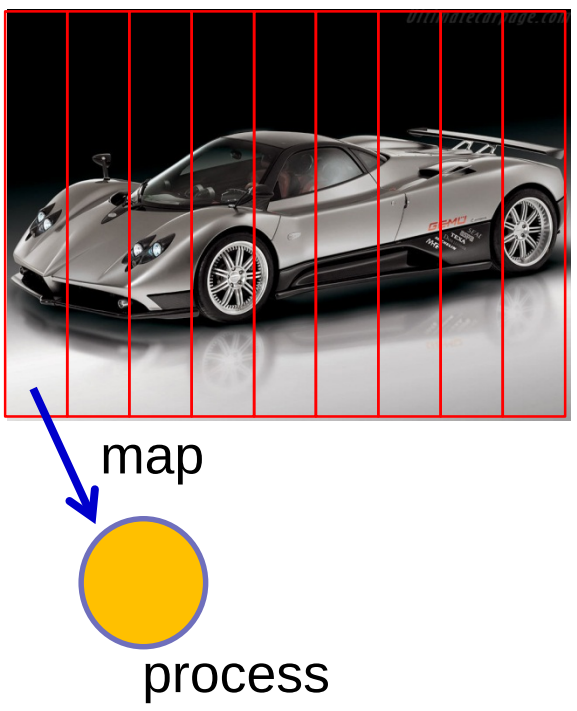
    
    

Image Region Partitioning

- Square region partition
- Row region partition
- Column region partition

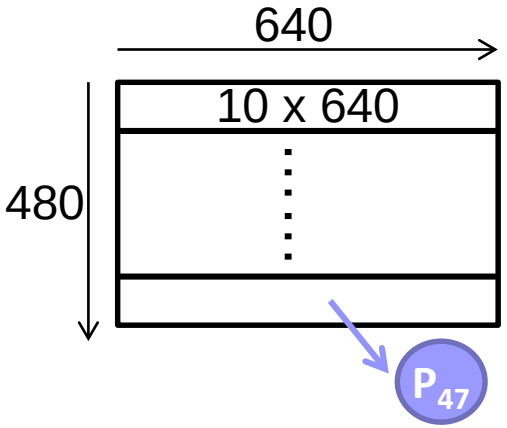






Pseudo-code for Image Shift

- Partition region by ROW with width 10




```
//master process
for(i=0, row=0; i<48; i++, row+=10)   // for each of 48 processes
      send(row, Pi);                  // send row no.

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

for(i=0; i<(480*640); i++) {                                    // for each pixel
      recv(oldrow, oldcol, newrow, newcol, PANY);               // accept new coordinates
      if !((newrow<0)||((newrow>=480)||(newcol<0)||((newcol>=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];   // update map
```

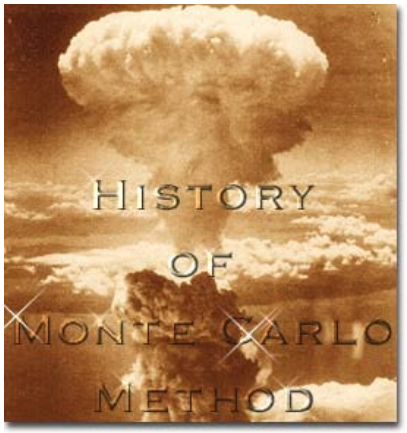
```
// slave process
recv (row, Pmaster);
for (oldrow = row; oldrow < (row+10); oldrow++)      // for each row in the partition
      for (oldcol = 0; oldcol < 640; oldcol++) {     // for each column in the row
            newrow = oldrow + delta_x;               // shift along x-dimension
            newcol = oldcol + delta_y;               // shift along y-dimension
            send(oldrow, oldcol, newrow, newcol, Pmaster); // send out new coordinates
      }
```

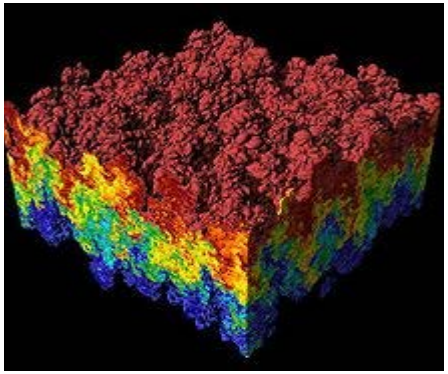
Example 2: Monte Carlo Methods

- **Monte Carlo methods**: a class of computational algorithms that rely on **repeated random sampling** to compute their results

  - Invented in 1940s by *John von Neumann*, *Stanislaw Ulam* and *Nicholas Metropolis*, while they were working on nuclear weapon (Manhattan Project)

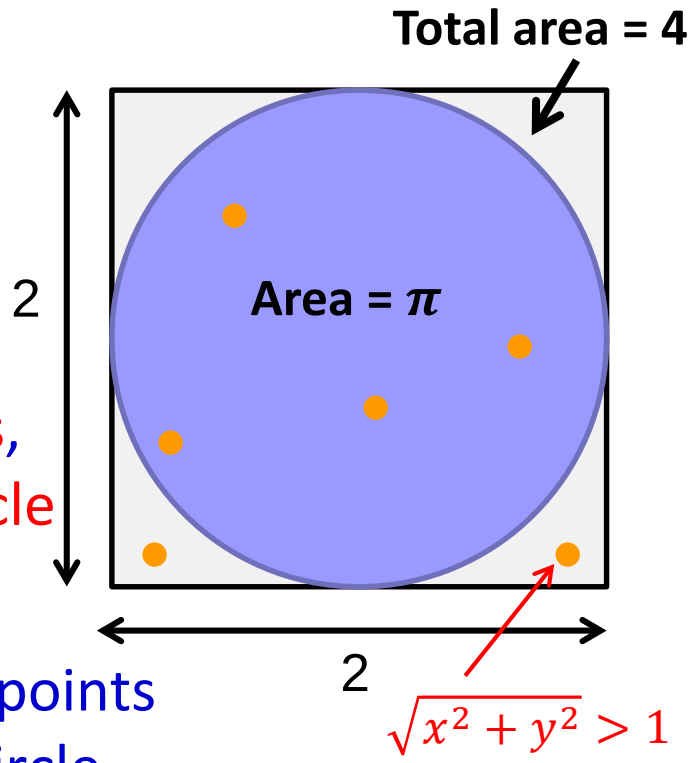
  - Especially useful for **simulating systems** with many coupled degrees of freedom, such as fluids, disordered material





Monte Carlo Methods --- π calculation

- How to compute π ???
  - Definition of π: the area of a circle with unit radius
  - We know: $\frac{\textbf{Area of circle}}{\textbf{Area of square}} = \frac{\pi}{4}$
  - Randomly choose points from the square
  - Giving sufficient number of samples, the fraction of points **within** the circle will be $\pi/4$!!!
  - E.g.: With 10,000 randomly sample points we expect 7854 points within the circle
    ➔7854/10000 =π/4 ➔π = 7854/10000*4 = 3.1416

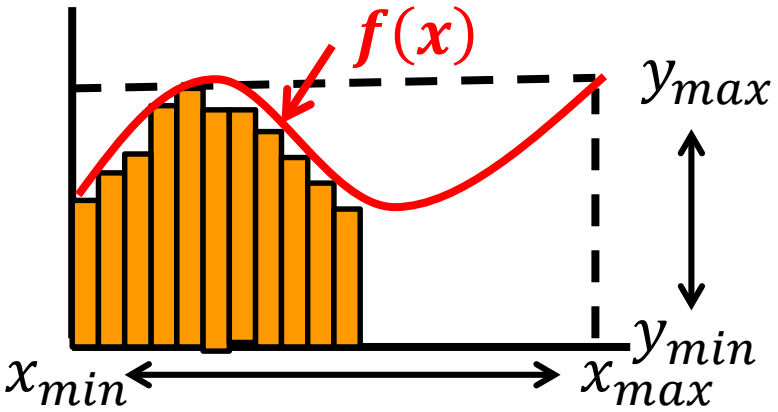


Monte Carlo Methods --- Integral

- Monte Carlo Method can compute ANY **definite** integral!
  - max and min values of the integral must be known
  - Very inefficient….

$$Area = \int_{x_{min}}^{x_{max}} f(x)dx$$

- Method:
  - Randomly choose point (x, y):
    - $x_{max} \leq x \leq x_{min}$
    - $y_{max} \leq y \leq y_{min}$
  - Compute the area (integral) according to the ratio of points inside and outside the area
    ➔ just like the computation of π



  - Given any point (x, y), outside means : $y > f(x)$

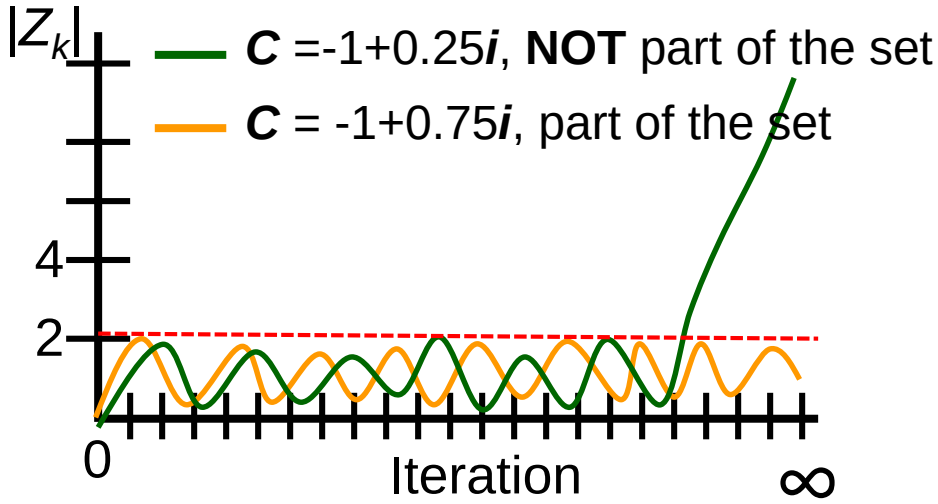
Example 3: Mandelbrot Set

- The Mandelbrot Set is a set of **complex numbers** that are quasi-stable when computed by iterating the function:

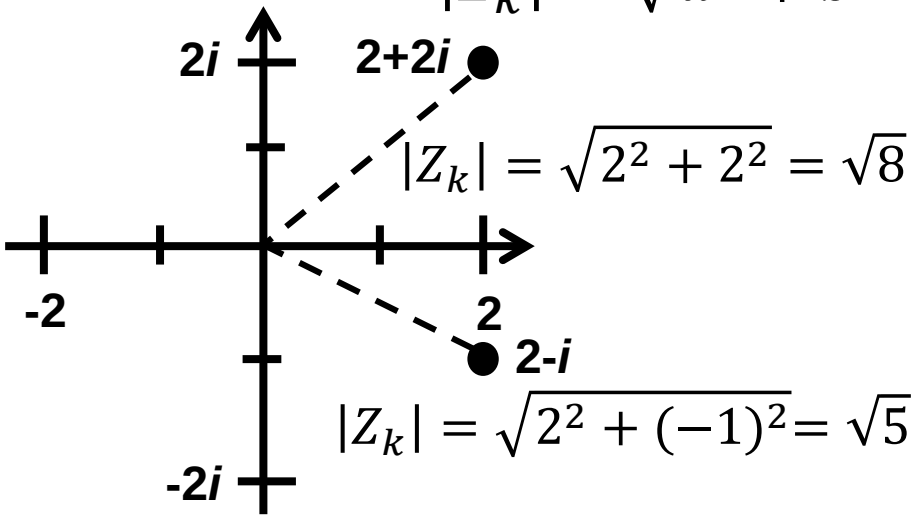
$$Z_0 = C, \qquad Z_{k+1} = {Z_k}^2 + C$$

  - $\boldsymbol{C}$ is some complex number: $\boldsymbol{C} = a + b\boldsymbol{i}$
  - $\boldsymbol{Z_{k+1}}$ is the **(k+1)**th iteration of the complex number
  - If $|\boldsymbol{Z_k}| \leq 2$ for **ANY** $\boldsymbol{k}$ ➔ $\boldsymbol{C}$ belongs to Mandelbrot Set

Once $|Z_k| > 2$, it will increase forever!



$$|Z_k| = \sqrt{a^2 + b^2}$$

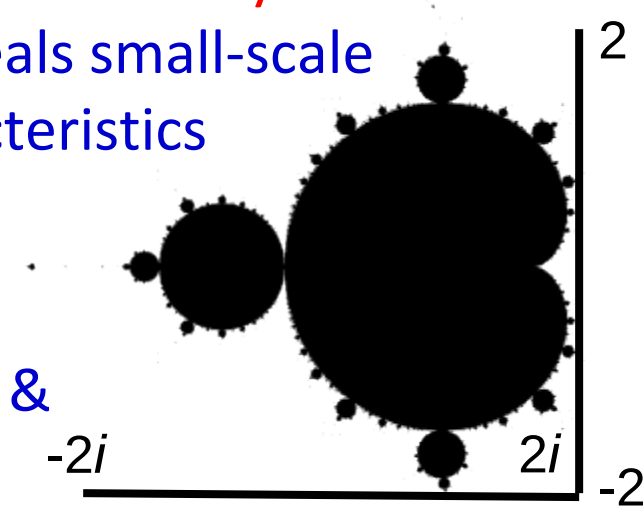


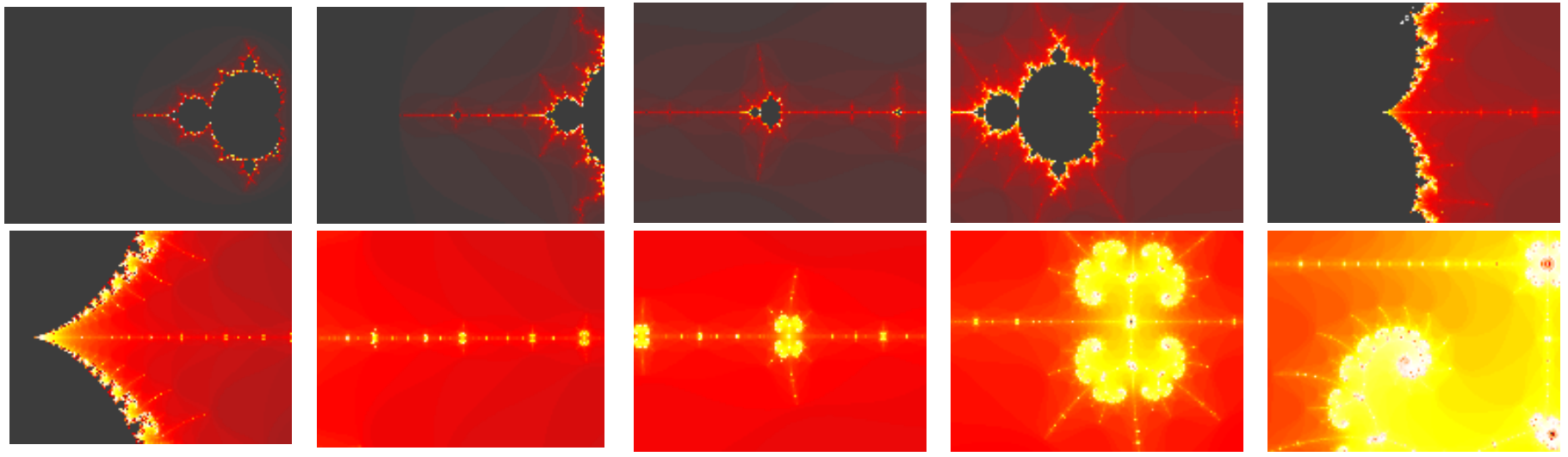
$$|Z_k| = \sqrt{2^2 + 2^2} = \sqrt{8}$$

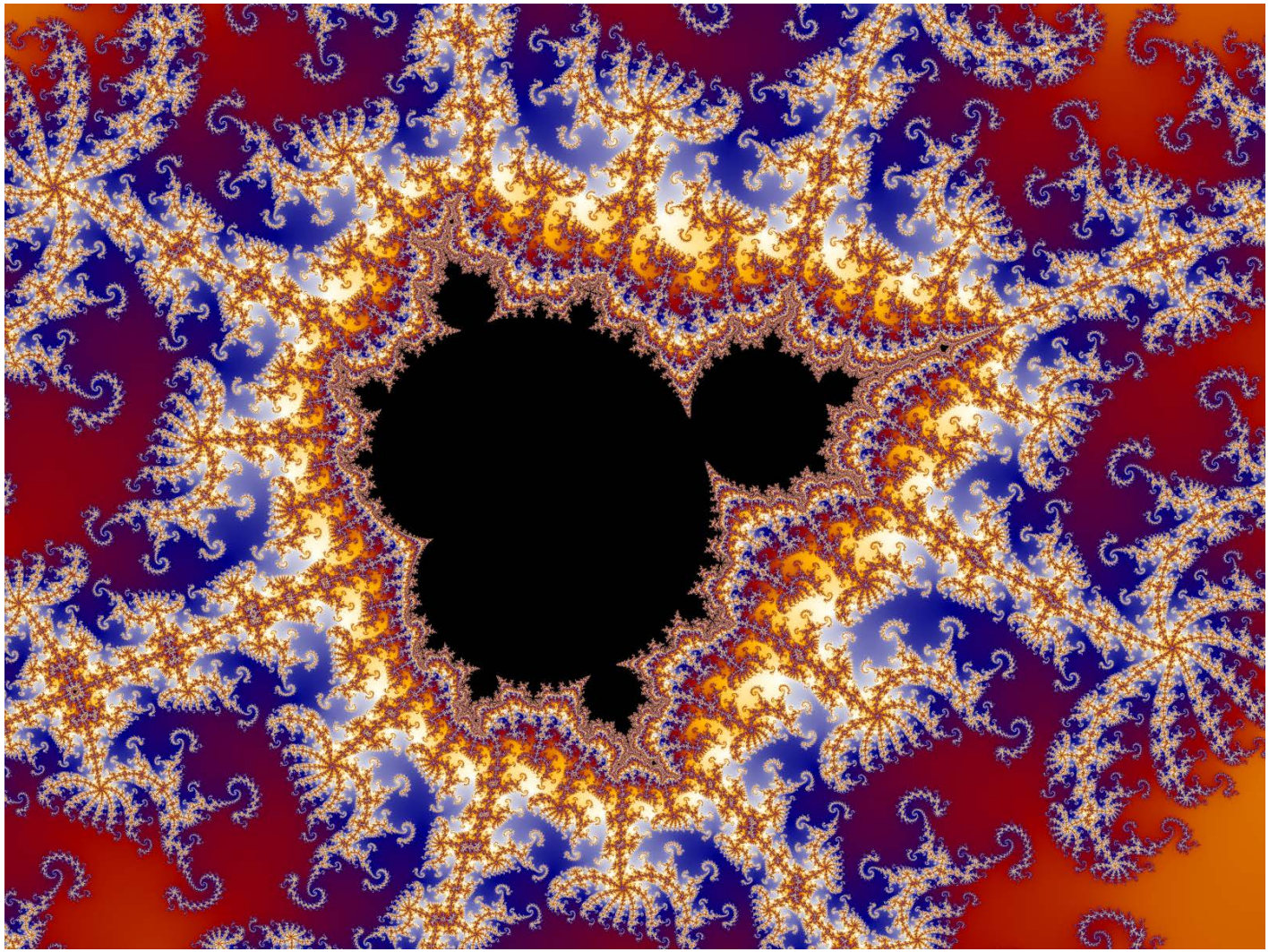
$$|Z_k| = \sqrt{2^2 + (-1)^2} = \sqrt{5}$$

Fractal

- What exact is Mandelbrot Set?
  - It is a **fractal**: An object that display self-similarity at various scale; Magnifying a fractal reveals small-scale details similar to the large-scale characteristics
  - After plotting the Mandelbrot Set determined by thousands of iteration:
  - Add color to the points outside the set & zoom in at the center of the image:







Mandelbrot Set Program

- Compute $Z_{k+1} = {Z_k}^2 + C$
  - Let $C = C_{real} + C_{imag}i$, $Z_k = Z_{real} + Z_{imag}i$
  - $Z_{k+1} = \left({Z_{real}}^2 - {Z_{imag}}^2 + 2Z_{real}Z_{imag}i\right) + \left(C_{real} +\right.$

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

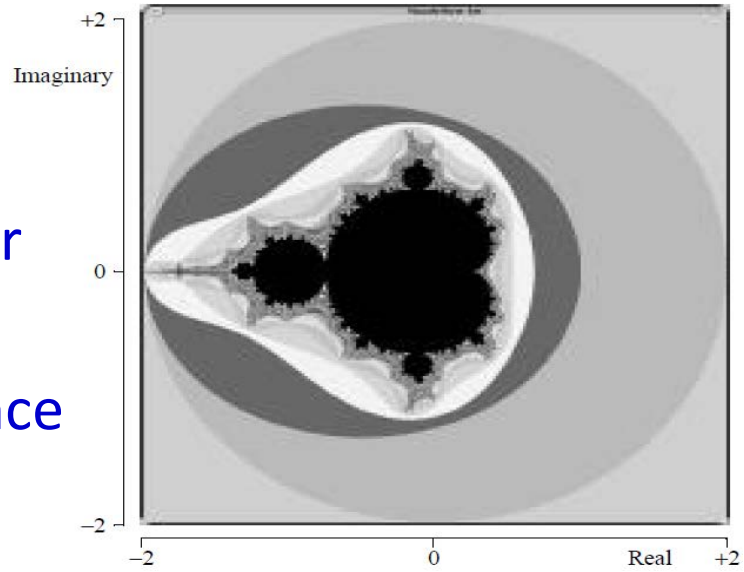
Sequential Mandelbrot Set Program

- Testing program:
  - Giving a complex number
  - Return the iteration number when $|Z_k| > 2$
  - Let the maximum iteration is 256

```
int cal_pixel (complex c) {
   int count = 0;                          // number of iterations
   int max= 256;                           // maximum iteration is 256
   float temp, lengthsq;
   complex z;                              // initialize complex number z
   z.real = 0; z.imag = 0;
   do {
      temp = (z.real * z.real) – (z.imag * z.imag) + c.real;   // compute next z.real
      z.imag = (2 * z.real * z.imag) + c.imag;                 // compute next z.imag
      z.real = temp;
      lengthsq = (z.real * z.real) + (z.imag * z.imag);
      count++;                                                 // update iteration counter
   } while ((lengthsq < 4.0) && (count < max));
   return count;
}
```

Sequential Mandelbrot Set Program

- Scaling Coordinate Display Program:
  - Plot the Mandelbrot Set from the coordinate system
  - Color indicate the iteration number black=256, white=0
  - Points are apart with a fixed distance read_disk, imag_dist




```
for (x=real_min; x < real_max; x += real_dist) {
   for (y=imag_min; y < imag_max; x += imag_dist) {
         c.real = x; c.img = y;
         color = cal_pixel (c);
         display(x, y, color);
   }
}
```

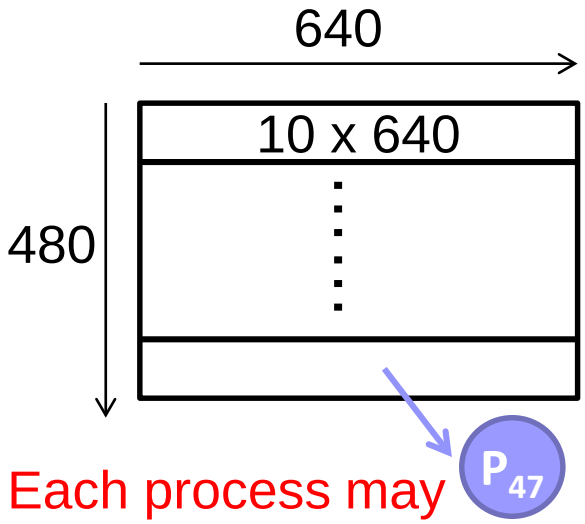
Parallelizing Mandelbrot Set Program

- Partition screen 640*480 by row using 48 processes

```
//master process
for(i=0, row=0; i<48; i++, row+=10)      // for each process
      send(row, Pi);                     // send row no.

for(i=0; i<(480*640); i++) {             // for each pixel point
      recv(&x, &y, &color, PANY);        // receive coordinate/colors
      display(x, y, color);              // display pixel
}
```

```
//slave process
recv (&row, Pmaster);
for (x=0; x < 640; x++) {
   for (y=row; y < (row+10); y++) {
         c.real = min_real + (x * scale_real);
         c.imag = min_imag + (y * scale_image);
         color = cal_pixel (c);
         send(x, y, &color, Pmaster);
   }
}
```

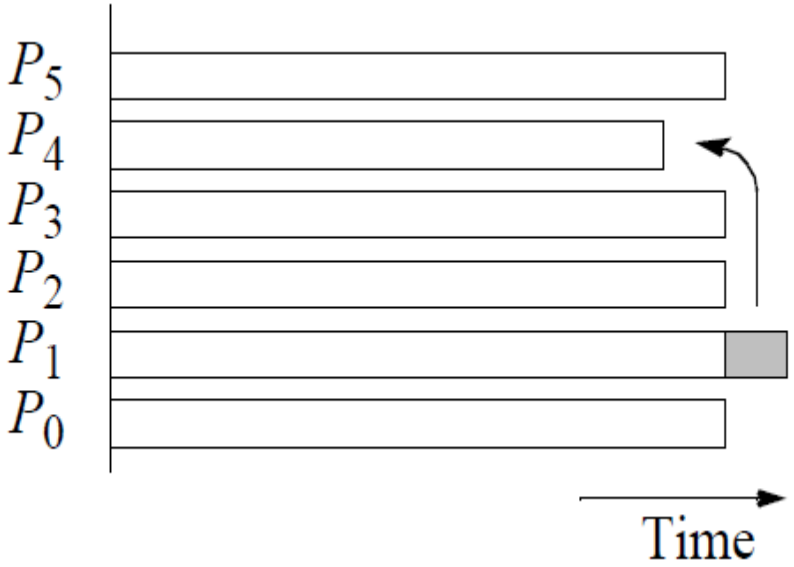


Outline

- Embarrassingly Computations
- Load Balancing & Termination
- Divide-And-Conquer Computations
- Pipelined Computations
- Synchronous Computations

Load-Balancing & Termination

- *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



Static Load-Balancing

- ***Static*** *means …*
  - *Pre-determine assignment between tasks and processes*

- *Round robin algorithm* — selects processes in turn
- *Randomized algorithms* — selects processes randomly
- *Recursive bisection* — recursively divides the problem into sub-problems of equal computational effort

Static Load-Balancing

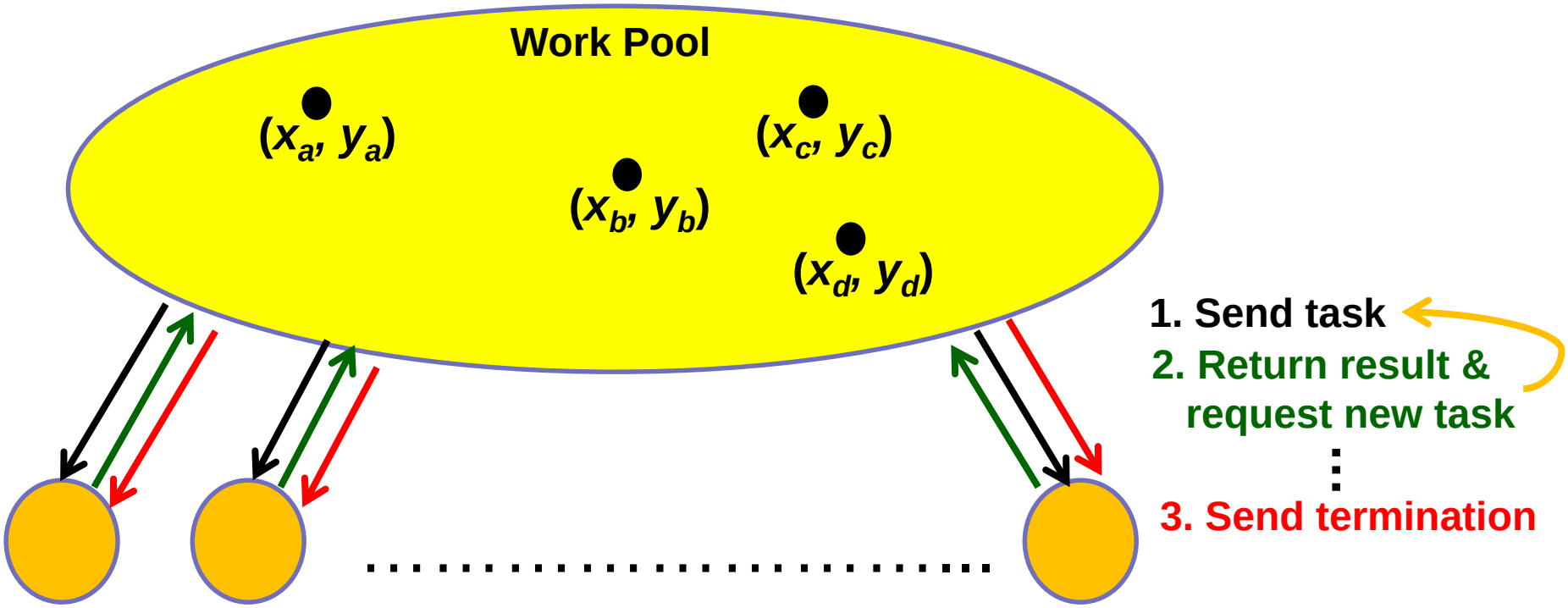
- Several fundamental flaws with static load balancing even if a mathematical solution exists:
  - Very difficult to estimate accurately the execution times of various parts of a program without actually executing the parts
  - Communication delays that vary under different circumstances
  - Some problems have an indeterminate number of steps to reach their solution (e.g. Manderbrot set)

Dynamic Load Balancing

- ***Dynamic*** *means …*
  - *Assign tasks to processes during the* ***execution***
  - Does incur an additional overhead during execution, but it is much more effective than static load balancing
- Approaches
  - Centralized Work Pool
  - Decentralized Work Pool
  - Fully Distributed Work Pool

Centralized Work Pool

- Work pool / Processor Farm
  - Useful when tasks require different execution time
  - Dynamic load balancing



Coding for Work Pool Approach

```
//master process
count = 0;                              // # of active processes
row = 0;                                // row being sent
for (i=0; i<num_proc; i++) {            // send initial row to each processes
      send(row, P_i , data_tag);
      count++;
      row++;
}
do {
      recv(&slave, &r, color, P_ANY , result_tag);
      count--;
      if (row < num_row) {                    // keep sending until no new task
           send(row, P_slave , data_tag);     // send next row
           count++;
           row++;
      } else {
           send(row, P_slave , terminate_tag);   // terminate
      }
      display(r, color);                        // display row
} while(count > 0);
```

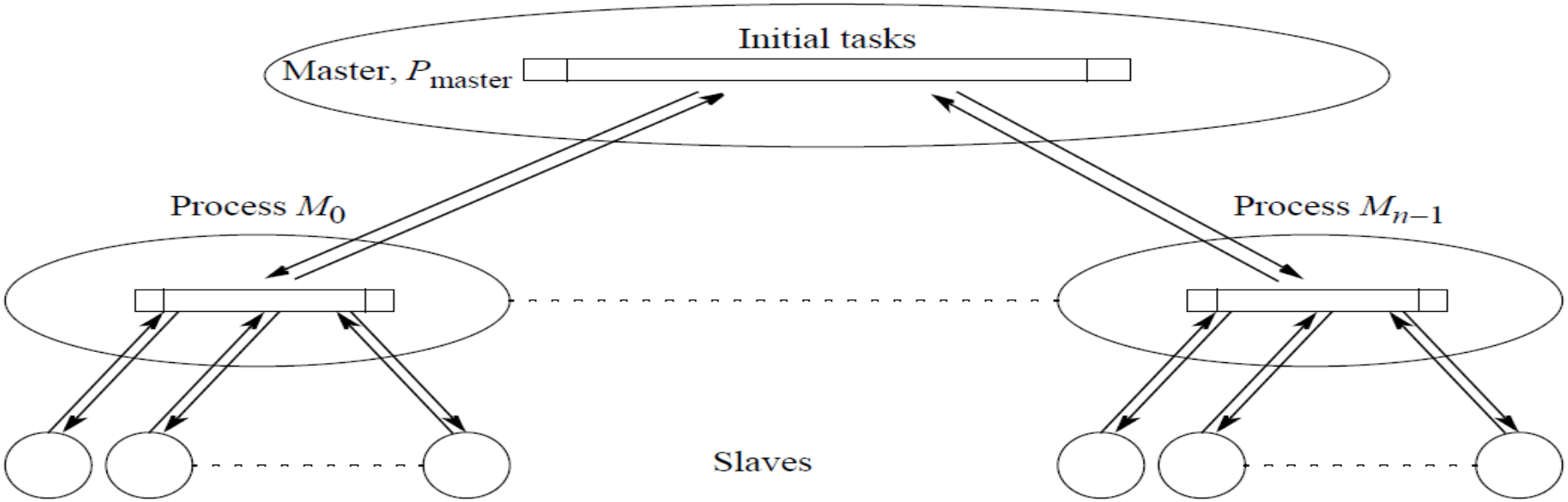
Tag is needed to distinguish between **data** and **termination** msg

Coding for Work Pool Approach

```
//slave process P ( i )
recv(&row, P_master , source_tag);
while (source_tag == data_tag) {          // keep receiving new task
      c.imag = min_imag + (row * scale_image);
      for (x=0; x<640; x++) {
            c.real = min_real + (x * scale_real);
            color[x] = cal_pixel (c);          // compute color of a single row
      }
      send(i, row, color, P_master , result_tag); // send process id and results
      recv(&row, P_master , source_tag);
}
```

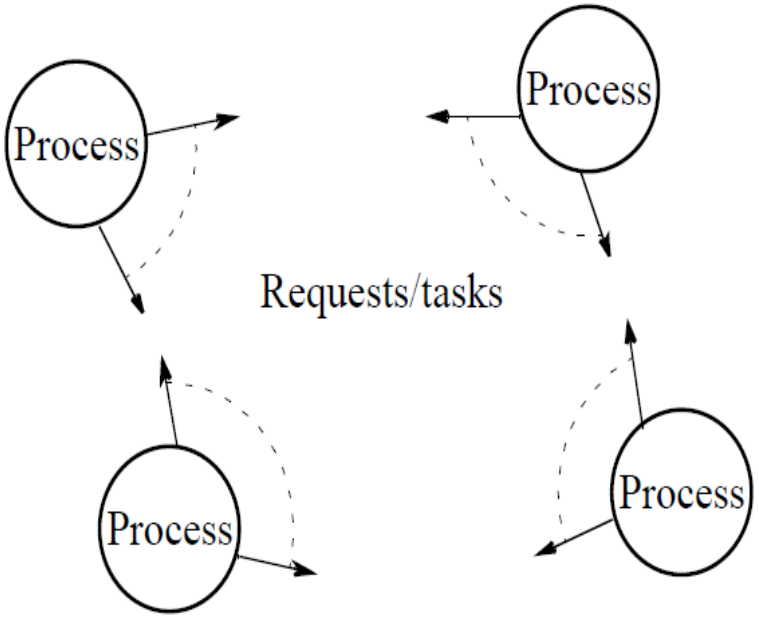
Decentralized Work Pool

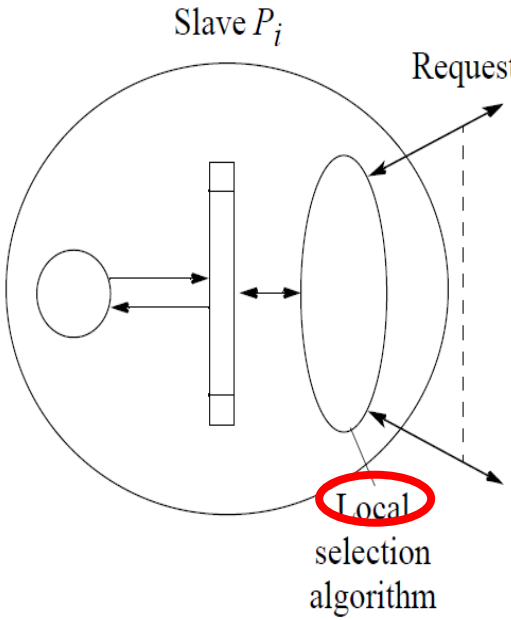
- Hierarchical structure
- The master divides the initial work pool into parts to each of the “mini-masters”
- Each mini-master controls one group of slaves

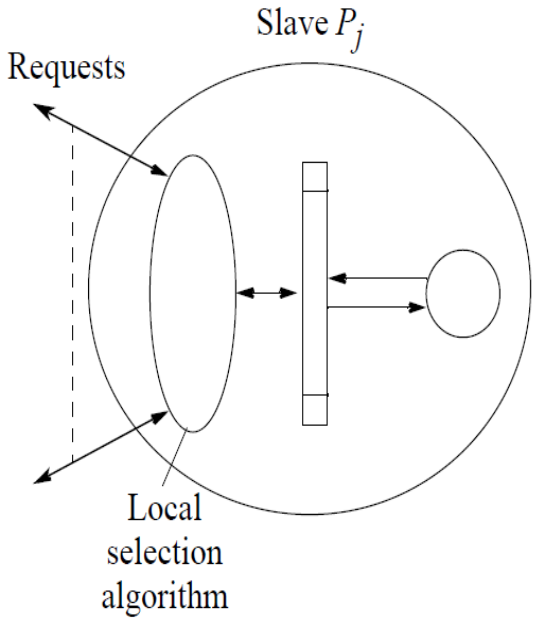


Fully Distributed Work Pool

- Processes have or generate their own tasks
- Tasks may be transferred among processes
- Processes make their own local decision





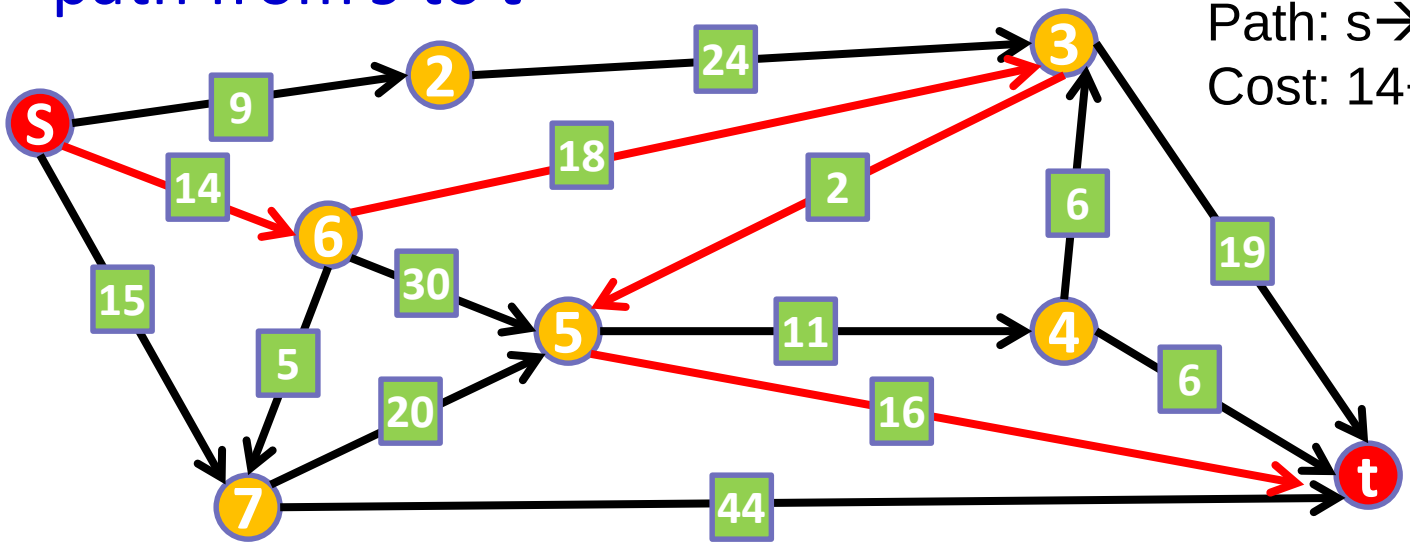


Fully Distributed Work Pool

- Receiver-initiated method (Receiver has lighter load)
  - Request tasks from heavy loaded processes
  - More suitable (less request) for heavy loaded system
- Sender-initiated method (sender has higher load)
  - Send tasks to light loaded processes
  - More suitable (less request) for light loaded system
- Challenges:
  - it could be difficult or expensive to determine process loads
  - may NOT provide immediate load balance

Shortest Path in a Weighted Diagraph

- Given a weighted graph, find the shortest directed path from ***s*** to ***t***



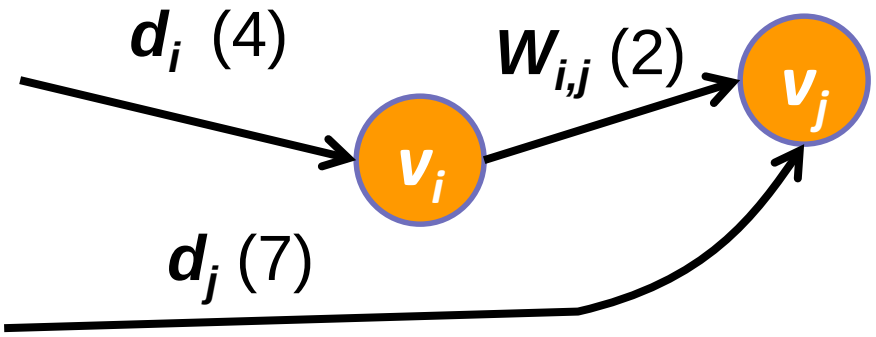
Path: s→6→3→5→t
Cost: 14+18+2+16=50

- Note: weight are arbitrary numbers
  - Not necessary distance
  - Need not satisfy the triangle inequality

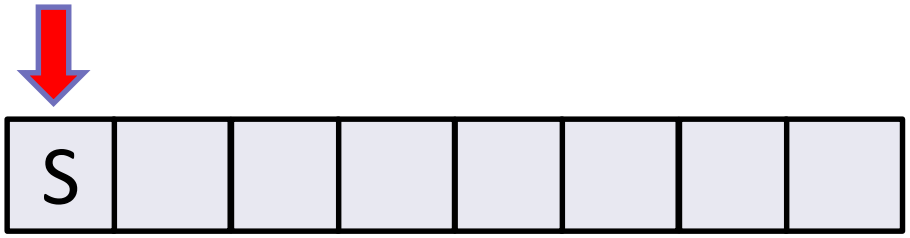
Moore's Shortest Path Algorithm

- Let d_i be the shortest distance from source to v_i
- Let $w_{i,j}$ be the weight between v_i and v_j
- Iteratively search over the graph
  - If d_i be the new shortest distance from source to v_i
  - For each v_j adjacent to v_i, update the shortest distance to v_j

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



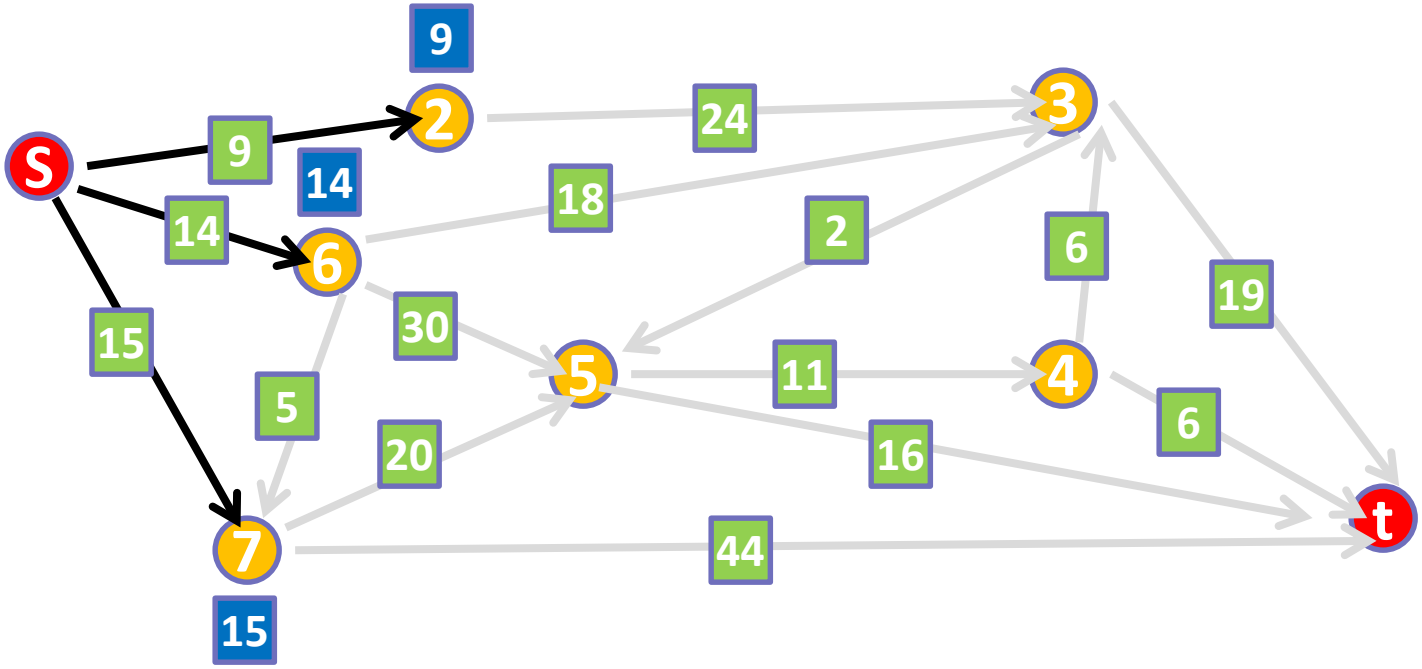
Moore's Shortest Path Algorithm



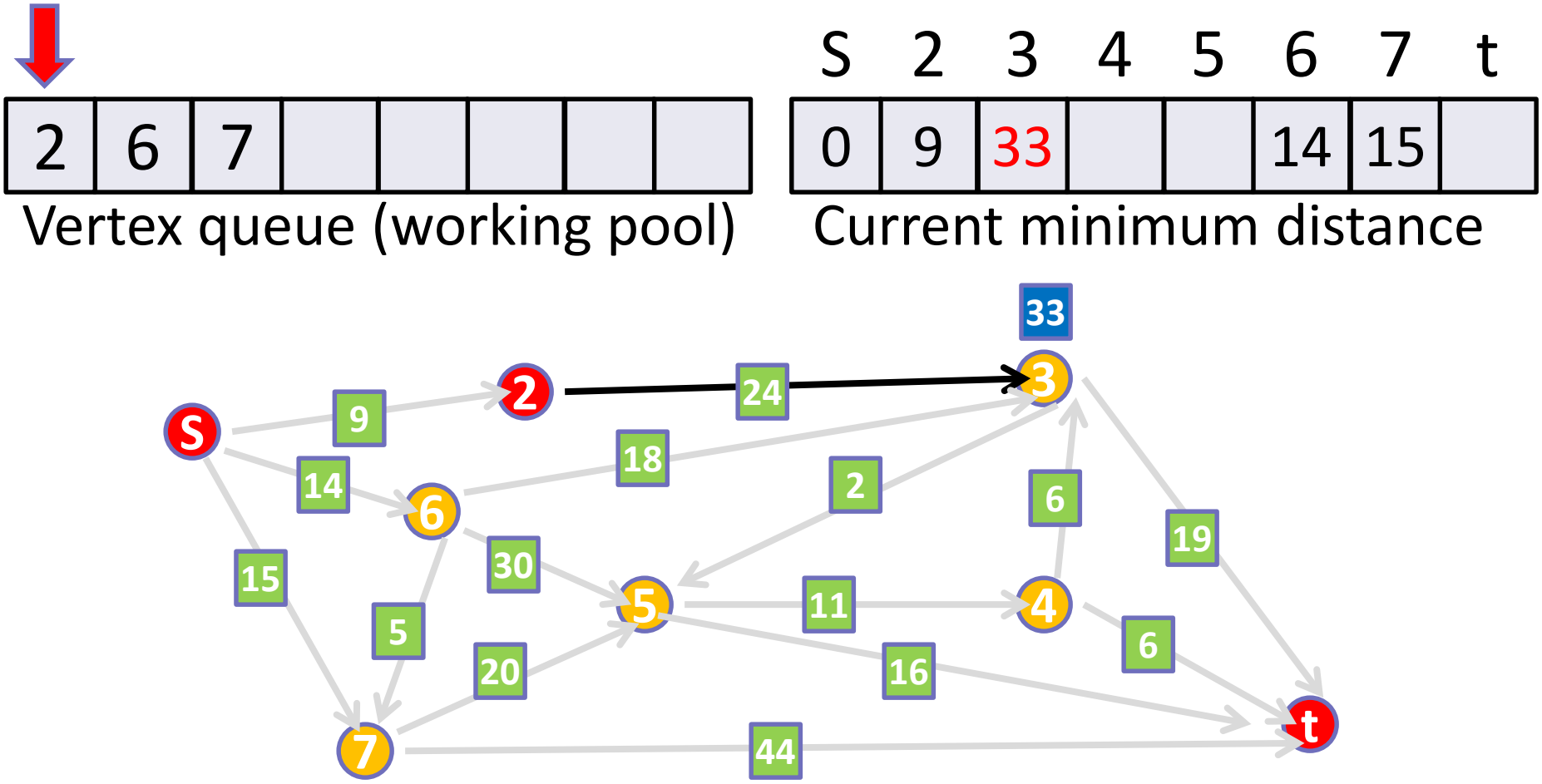
Vertex queue (working pool)

S	2	3	4	5	6	7	t
0	9				14	15	

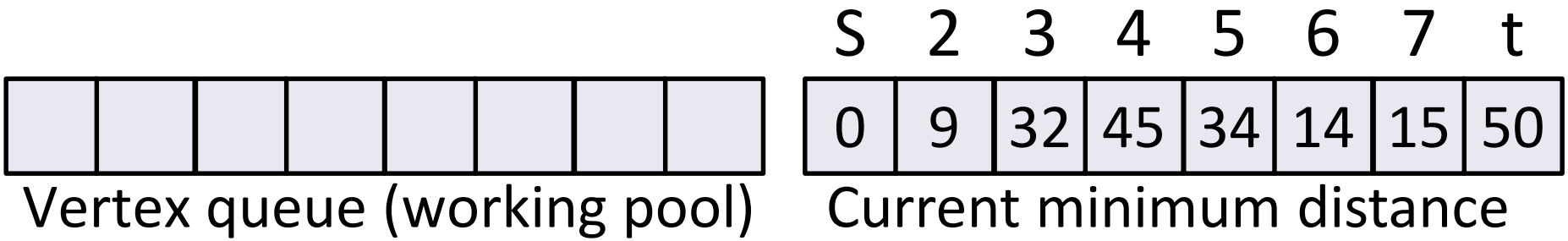
Current minimum distance

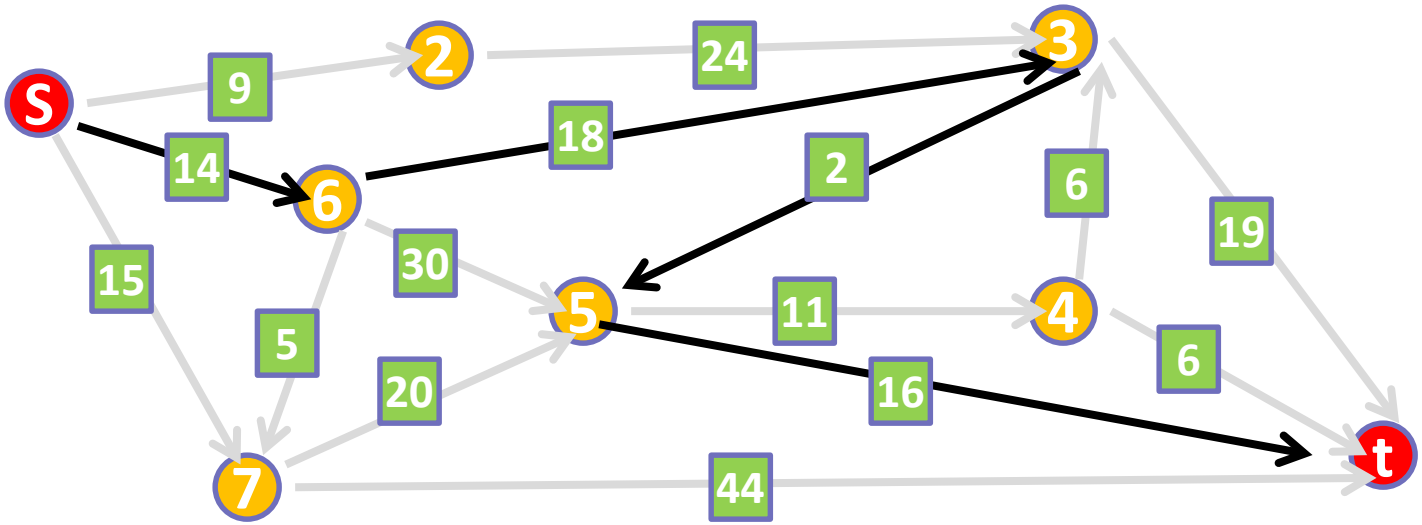


Moore's Shortest Path Algorithm



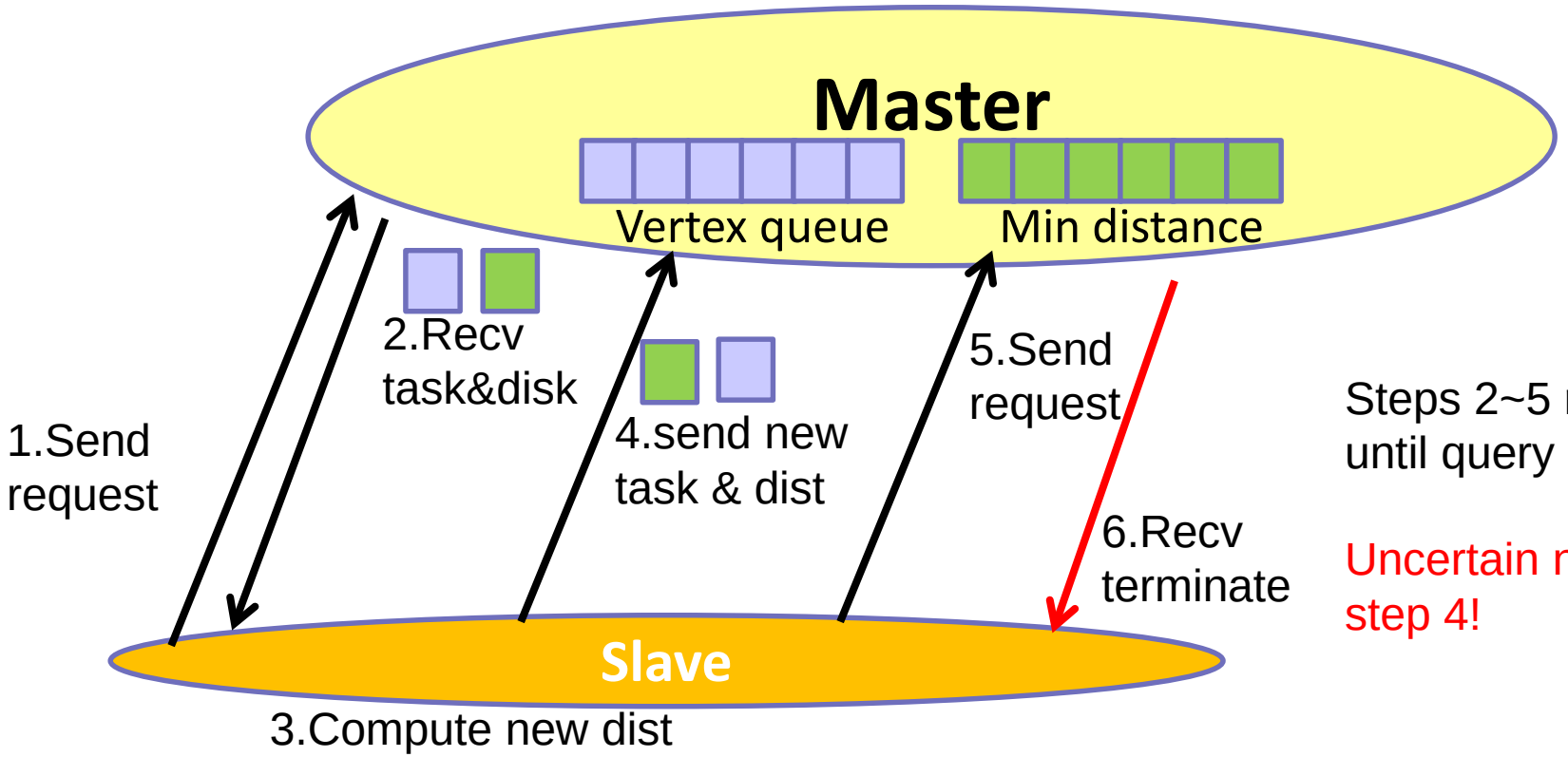
Moore's Shortest Path Algorithm





Centralized Pool Parallel Code

- Centralized work pool holds the vertex queue and distance array
- Each slave takes a vertex and returns new vertices and distances
- Since graph weights is fixed, it could be copied into each slave.

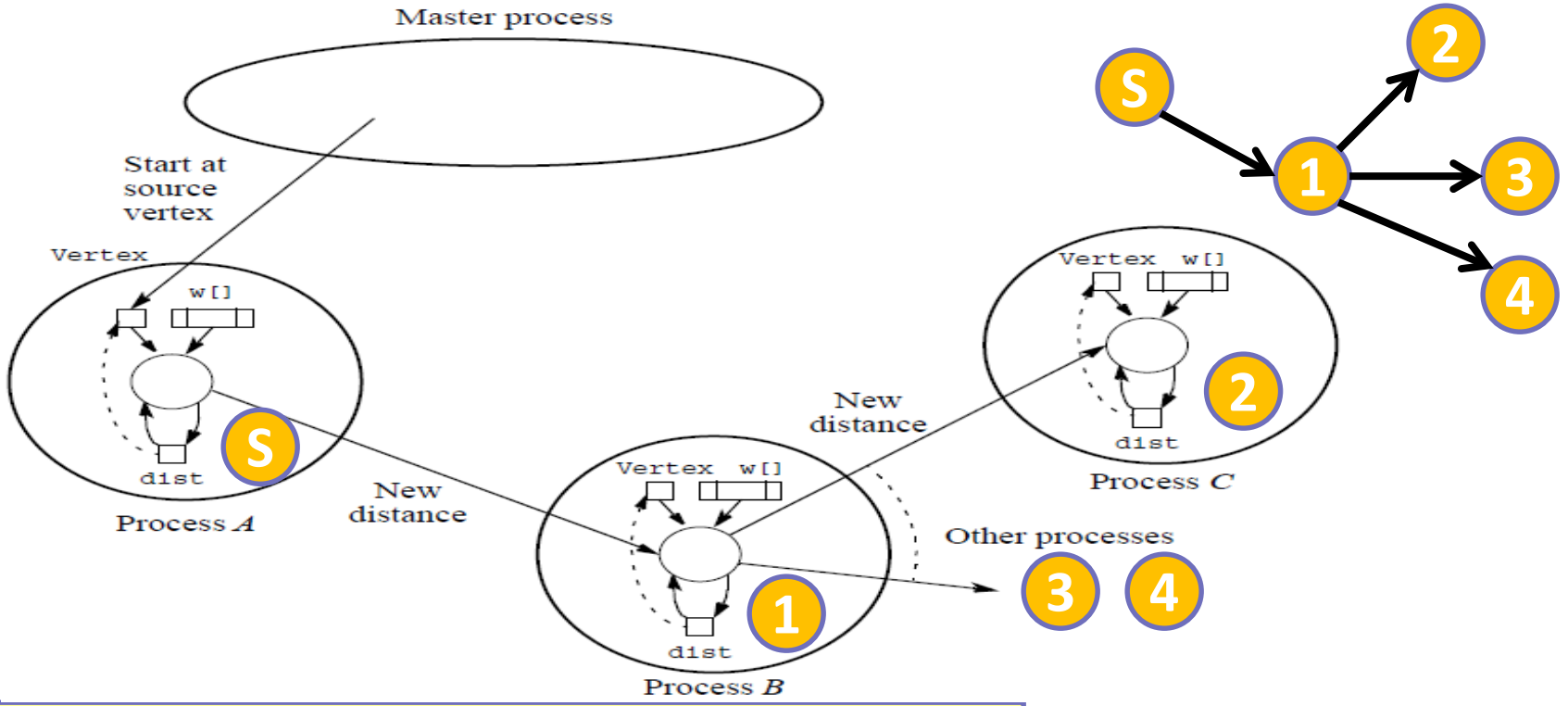


Distributed Pool Parallel Code

- Distribute vertex queue and distance array
  - Each process holds all the information of a vertex
  - Including queue, distance and adjacent weight

- Algorithm
  - For each new distance to a vertex V_i is computed
    - ➔ send the distance to P_i and re-activate P_i
    - ➔ if the new distance is shorter:
      - compute new distance for each adjacent vertex V_j
      - send the new distance to P_j

Distributed Pool Parallel Code




```
recv(newdist, PANY);
if (newdist < dist)
        dist = newdist; /* 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 */
                }
```

Q: How do we know when to terminate?

A: Use dual-pass ring algorithm…

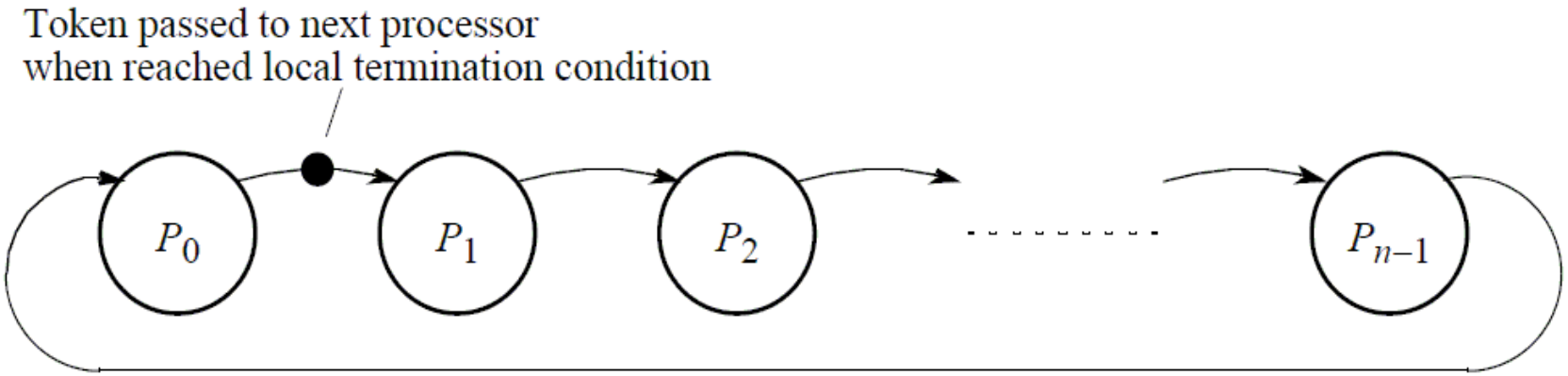
Distributed Termination Detection

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

  - 2nd condition is necessary because a message in transit might **reactivate** a terminated process
  - However, it is difficult to recognize because the message transfer time is not known in advance

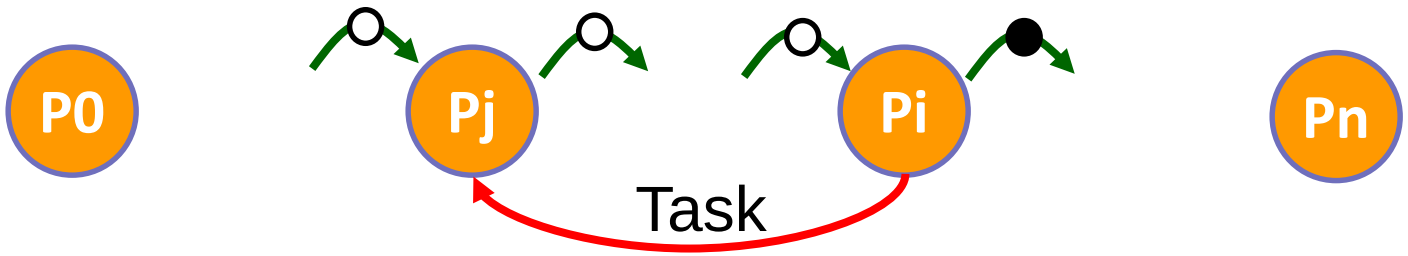
Single-Pass Ring Termination Algorithm

- Steps:
  - When P0 terminates, it generates a token that is passed to the next process P1
  - When Pi ($1 \leq i \leq n$) receives the token, it waits for its location termination condition and then passes the token to the next process Pi+1
  - When P0 receives back the token, a global termination message can be sent to all processes
- Must assume processes can **NOT** be re-activated



Dual-Pass Ring Termination Algorithm

- It can handle processes being reactivated but requires two passes around the ring
- Reactivation occurs when …
  - P_i, to pass a task to P_j where $j < i$ and after a token has passed P_j
- To handle reactivation, the token must recirculate through the ring a second time
  - The token is colored BLACK when it goes through Pi
  - The token becomes WHITE again once it goes through P0
  - If P0 receive WHITE tokens, all processes are terminated

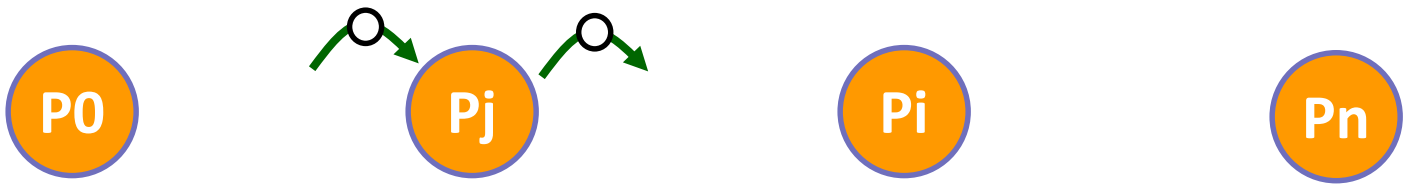


Dual-Pass Ring Termination Algorithm

1. *P*0 becomes white when it has terminated and generates a white token to *P*1.

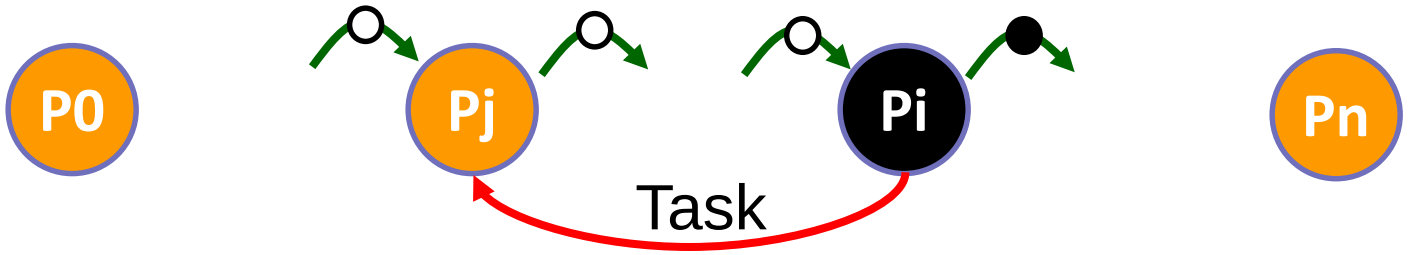
Dual-Pass Ring Termination Algorithm

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.
   - A white process will pass on the token in its original color (either black or white). After *Pi* has passed on a token, it becomes a white process. *Pn*-1 passes the token to *P*0.



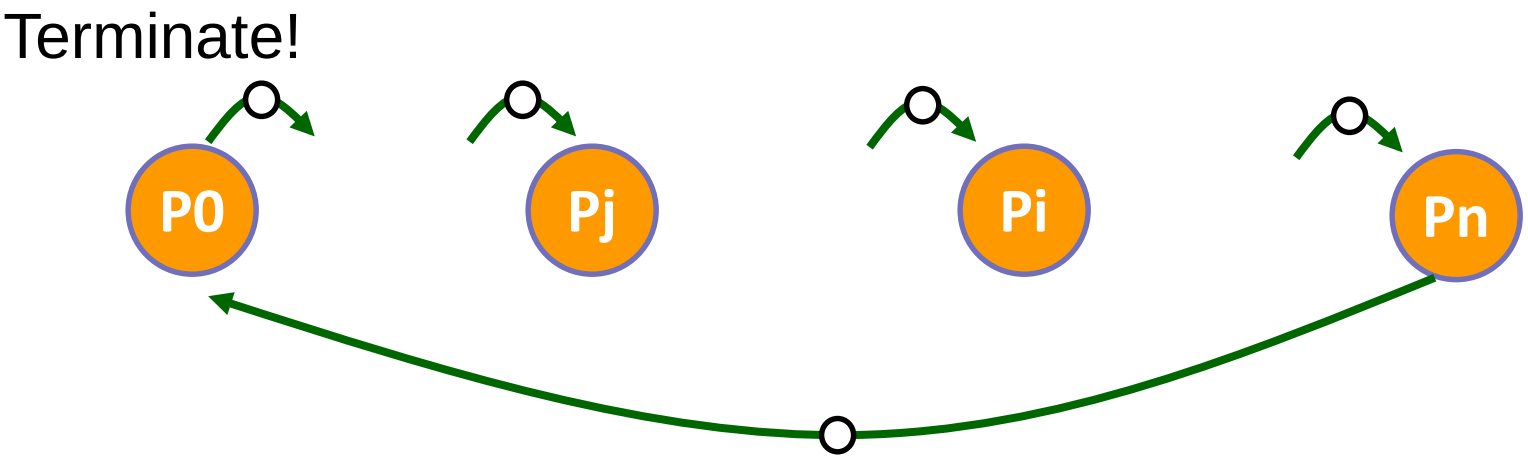
Dual-Pass Ring Termination Algorithm

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 *Pi* passes a task to *Pj* 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.

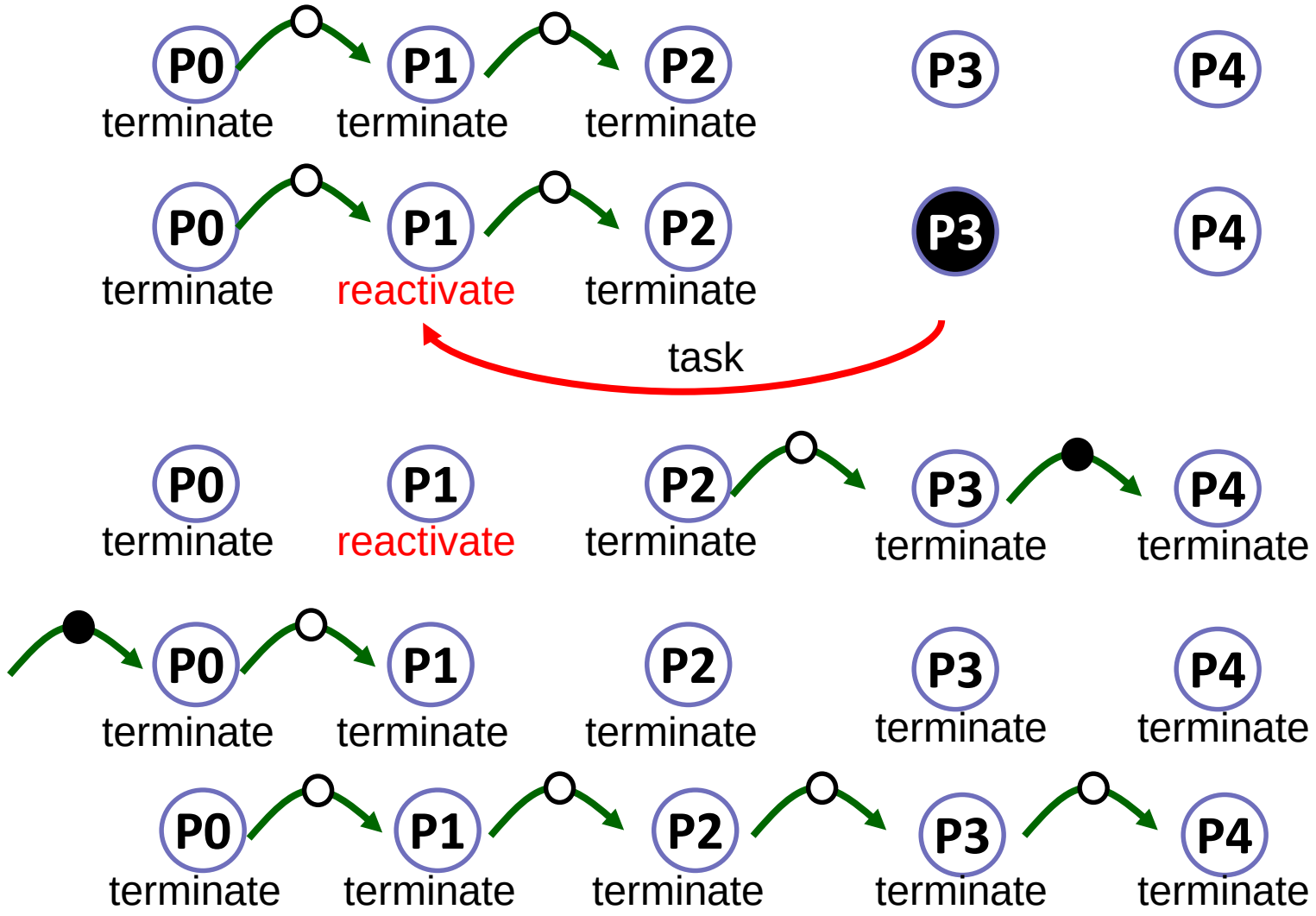


Dual-Pass Ring Termination Algorithm

3. When *P*0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.



Dual-Pass Ring Termination Example



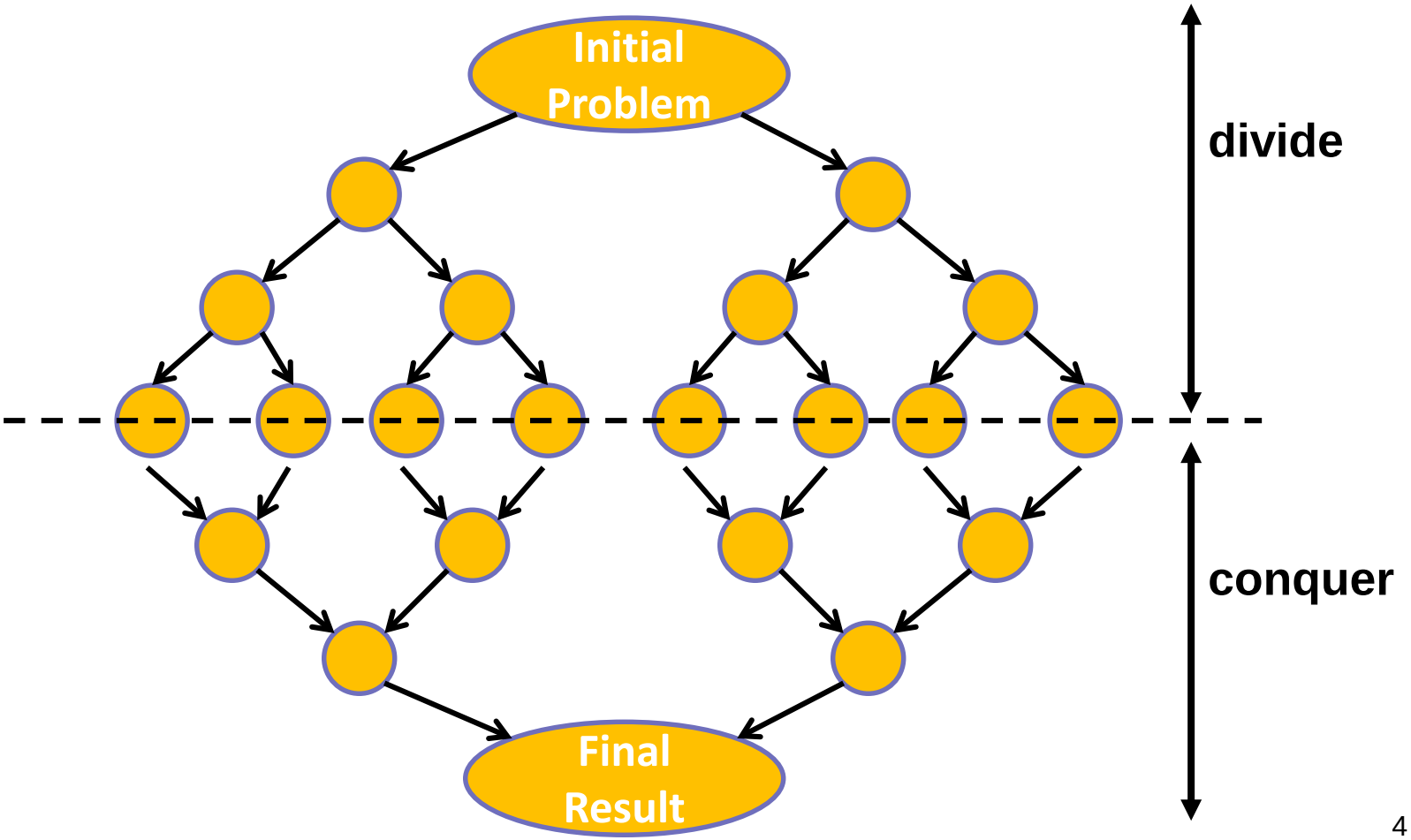
*P*0 is the central point for global termination

Outline

- Embarrassingly Computations
- Load Balancing & Termination
- Divide-And-Conquer Computations
  - Sorting Algorithms
  - N-Body Simulation
- Pipelined Computations
- Synchronous Computations

What is Divide & Conquer

- **Recursively** divide a problem into sub-problems that are of the same form as the larger problem



Sorting Algorithms

- Sorting
  - Re-arranging a list of numbers into increasing (strictly non-decreasing) order
  - One of the most common and critical operation in large data processing

Sorting in Parallel

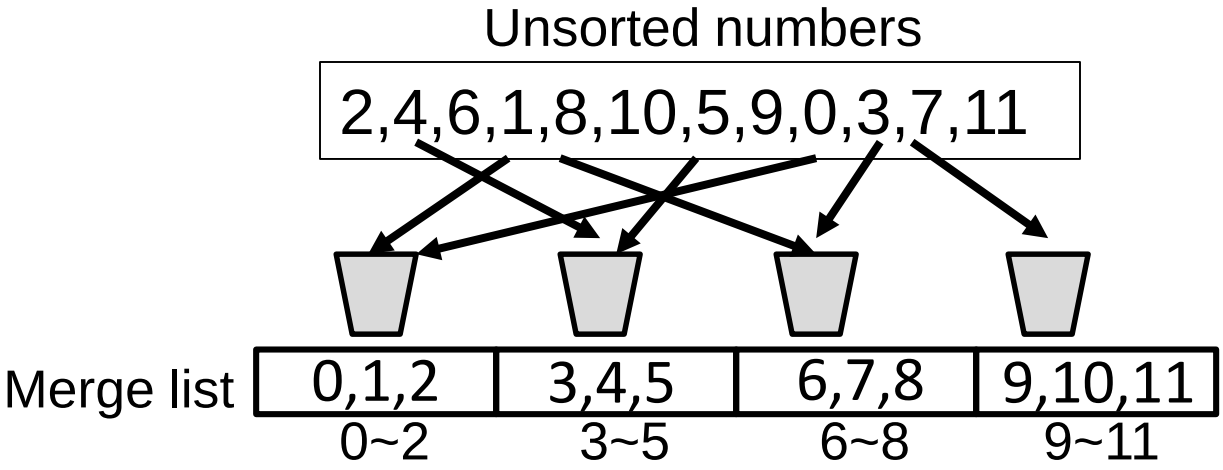
- In sequential
  - We all know it is $n \log n$
  - i.e. n is the number of elements
- In parallel with n processors
  - Optimal parallel time complexity

$$\frac{O(n \log n)}{n} = O(\log n)$$

Has been obtained but the constant hidden in the order notation extremely large

Bucket Sort

- Algorithm
  1. Range of numbers is divided into *m* **equal regions**
  2. One bucket is assigned for each region
  3. Place numbers to buckets based on the region
  4. Use sequential sort for each bucket



- Only effective if number of items per bucket is similar!!
  - Numbers should have a known interval ([max, min])
  - Numbers better to be uniformly distributed

Complexity Analysis

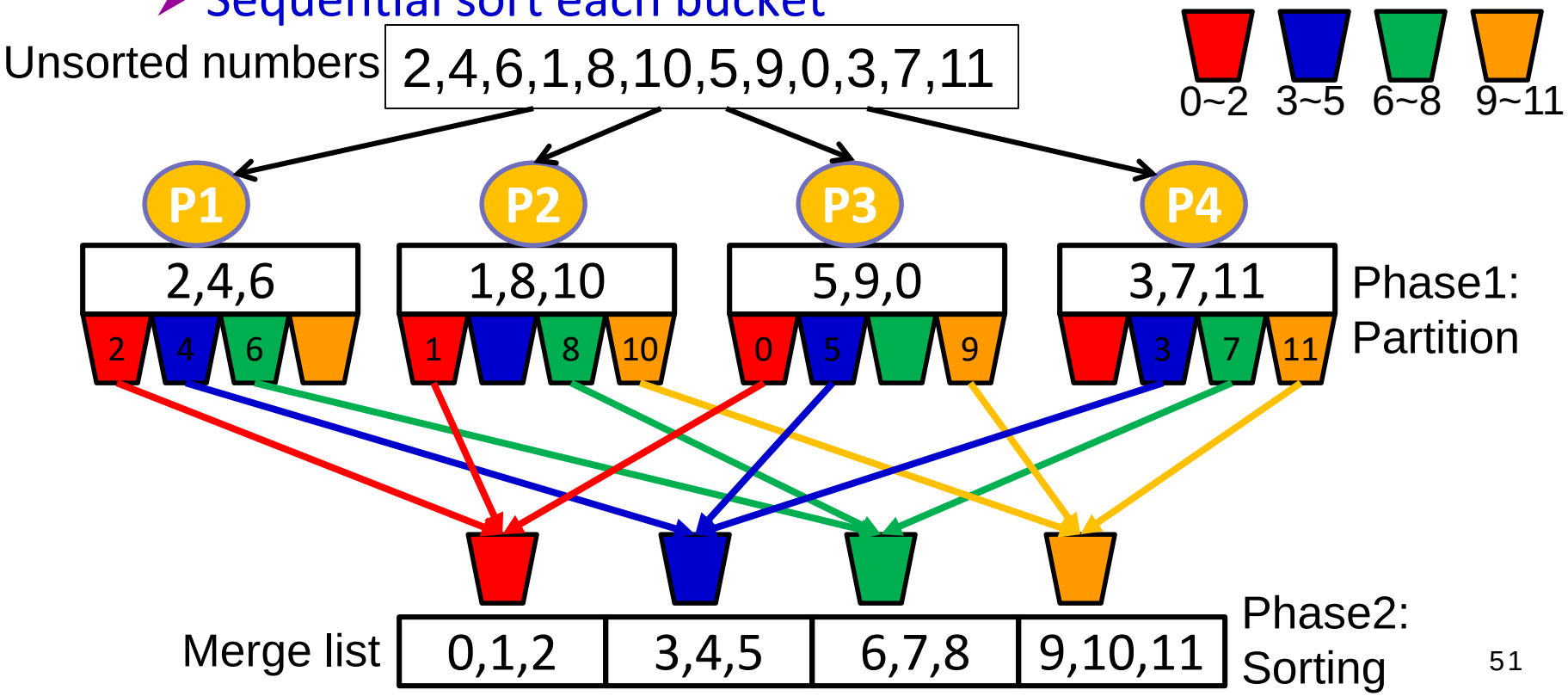
- Sequential:
  1. Distribute numbers to bucket: O(*n*)
  2. Sequential sort each bucket: (n/m)log(n/m) x m
  - Overall: O(n log(n/m))

- Parallelize sorting: one process per bucket
  1. Distribute numbers to bucket: O(*n*)
  2. Sequential sort each bucket: (n/m)log(n/m)
  - Overall: O(n + n/m log(n/m))

  - A single process must scan through all numbers in step1

Further Parallelized Bucket Sort

- Parallelize partitioning and sorting:
  - Partition numbers to ***m*** parts/processes
  - Each process divides its numbers to small buckets
  - Merge small buckets to large bucket
  - Sequential sort each bucket



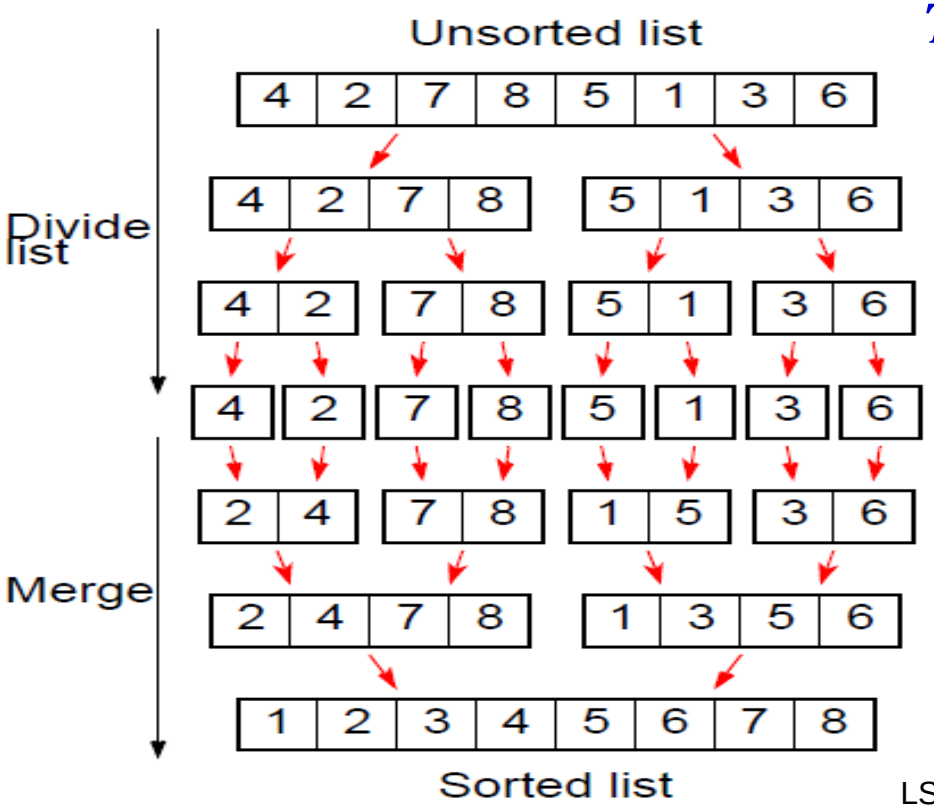
Merge Sort

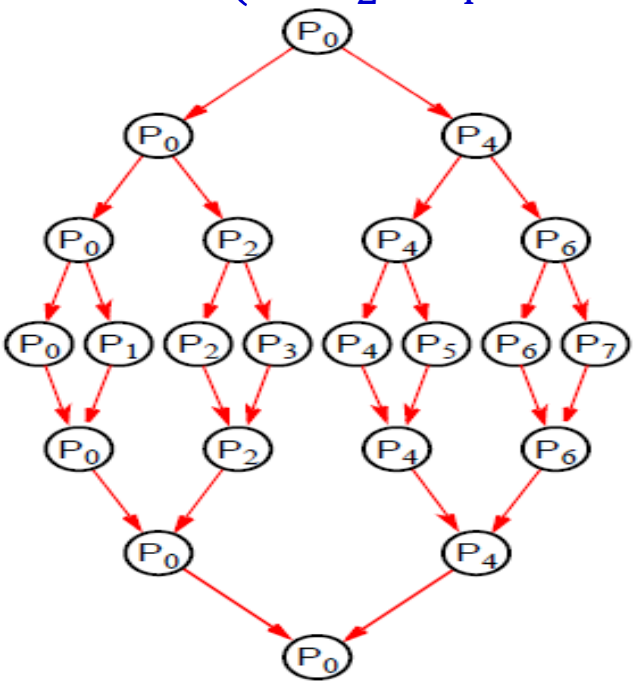
- Divide & Conquer
  - Sequential: $O(n \log n)$
  - Parallel: $O(n)$

#elements

$$T_{comm} = O\left(2\left(\frac{n}{2} + \frac{n}{4} + \cdots + 1\right)\right) = O(n)$$

$$T_{comp} = O\left(n + \frac{n}{2} + \frac{n}{4} + \cdots 2\right) = O(n)$$



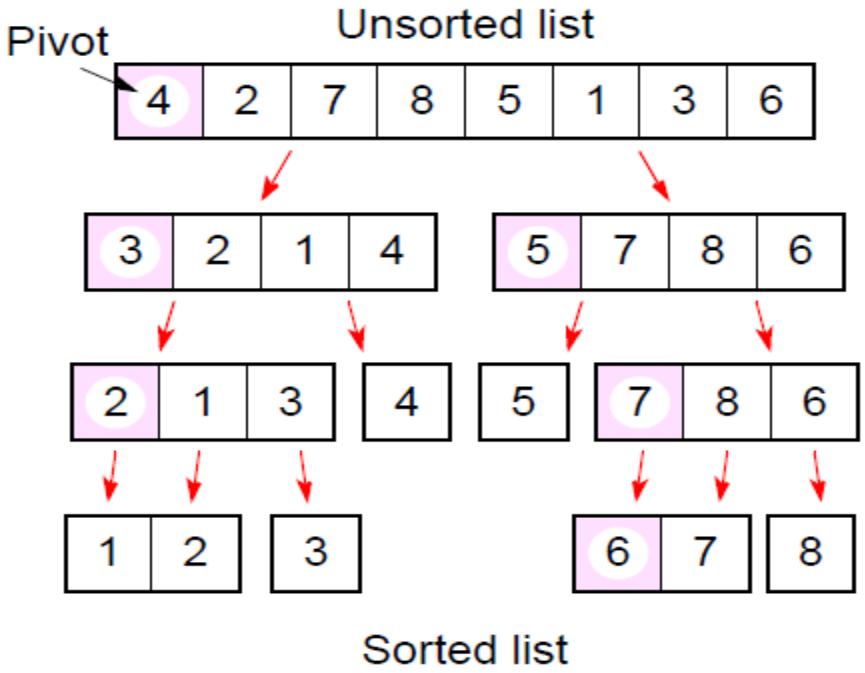


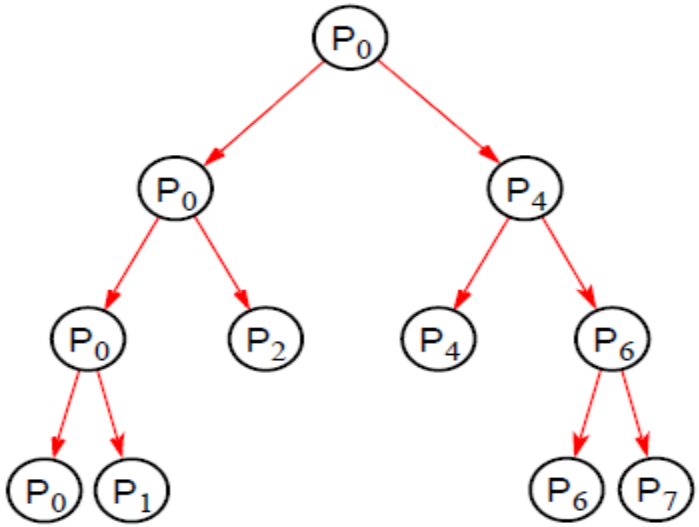
Process allocation

Quick Sort

- Most popular sequential sorting algorithm
  - Parallel: Iteratively pick pivot and partition numbers
- Complexity:
  - Sequential: $O(n \log n)$
  - Parallel: $O(n)$

Still best choose in parallel?
Not really & load might not be balanced



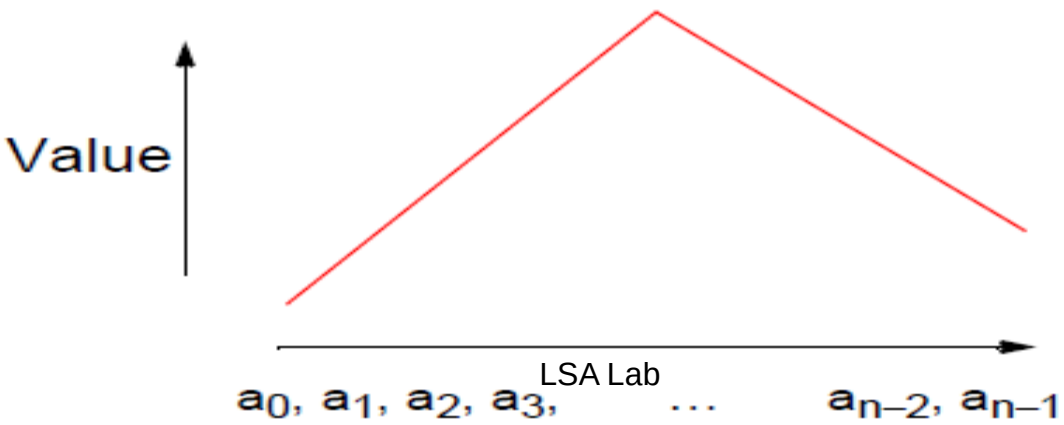


Bitonic Mergesort

- A parallel sorting algorithm based on **bitonic sequence**
  - **Remove the O(N) bottleneck for merging**
- Monotonic sequence:
  - A sequence of increasing numbers

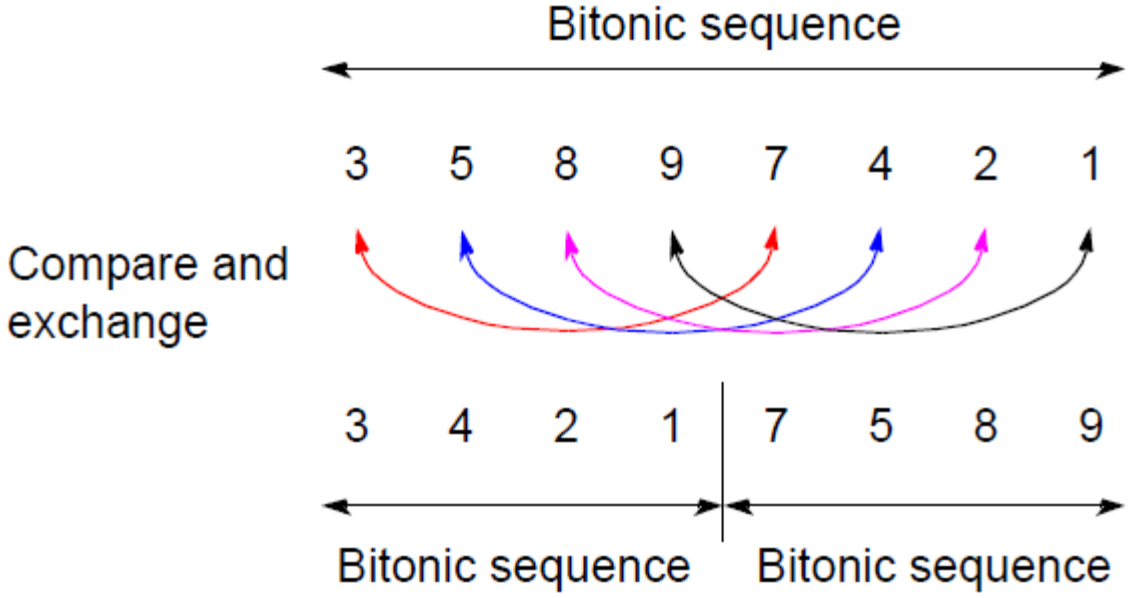
    <1, 2, 3, 4, 5, 7, 8, 9>
- Bitonic sequence:
  - Two sequences, one increasing and one decreasing

    <3, 5, 8, 9, 7, 4, 2, 1, >



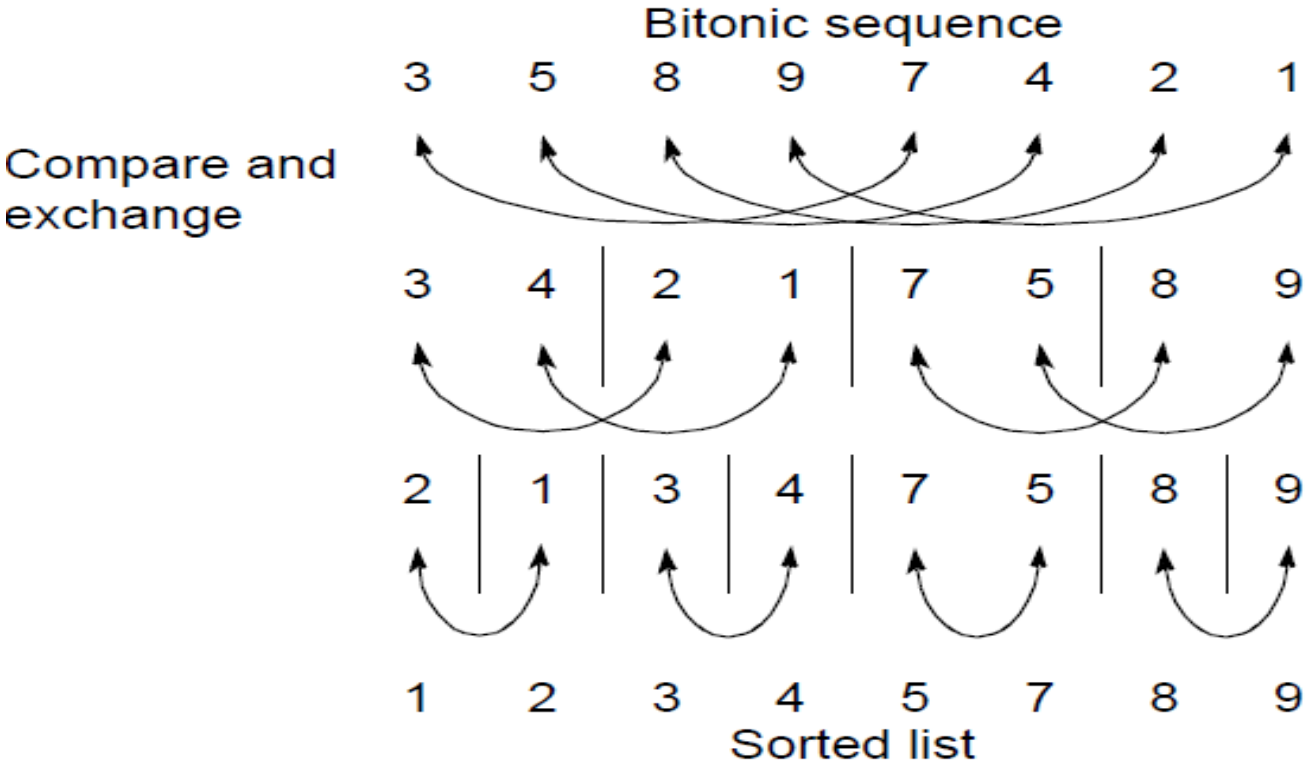
Characteristic of Bitonic Sequence

- If we perform a compare-and-exchange operation on a_i with $a_{(i+n/2)}$ for all i
  - Get **TWO** bitonic sequences
  - The numbers in one sequence are **all less than the numbers in the other sequence**
- Example: <3,5,8,9,7,4,2,1>



Bitonic Sorting Operation

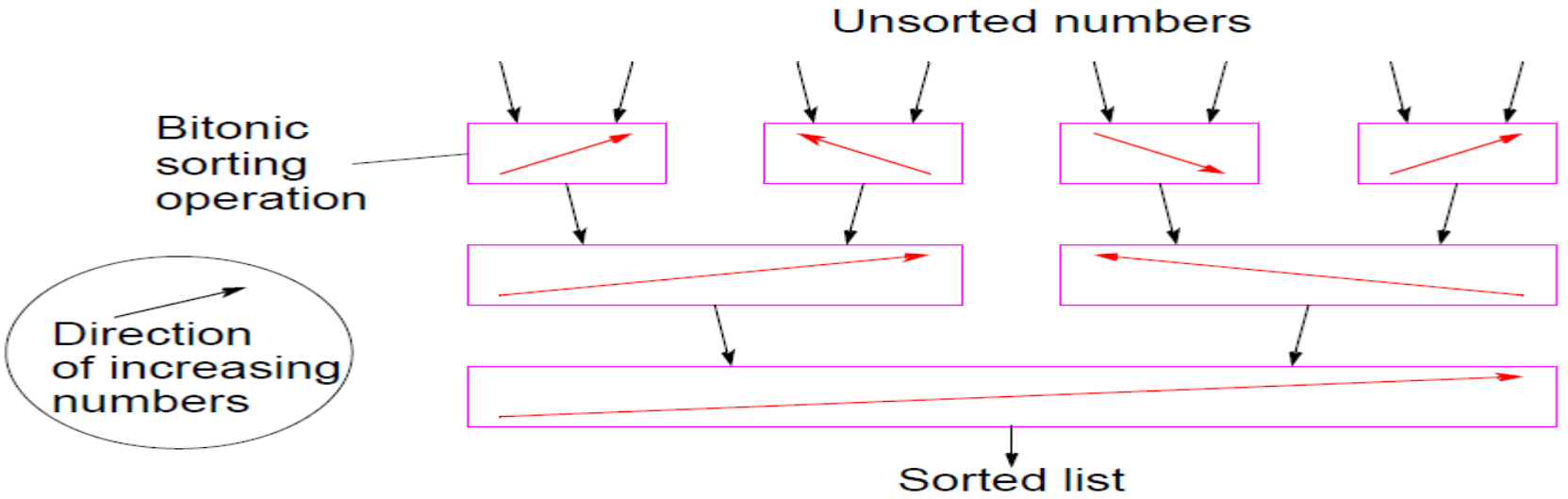
- Given a bitonic sequence, recursively performing compare-and-exchange will sort the list
- Complexity is only $\boldsymbol{O(log\ n)}$ instead of $O(n)$



Bitonic Mergesort

- Combine bitonic sorting into merge sorting
- Complexity:
  - Let $n = 2^k$: there will be k phases
  - k[th] phase has size 2^k ➔need only k steps with bitonic sorting

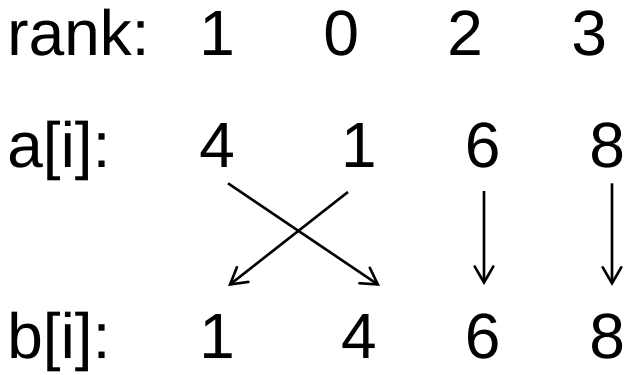
$$T_p = O\left(\sum_{i=1}^{k} i\right) = O\left(\frac{k(k+1)}{2}\right) = O\left(\frac{\log n(\log n)}{2}\right) = O(\log^2 n)$$



Rank Sort

- Count the number of numbers that are smaller than each of the selected number
- Sequential: $O(n^2)$

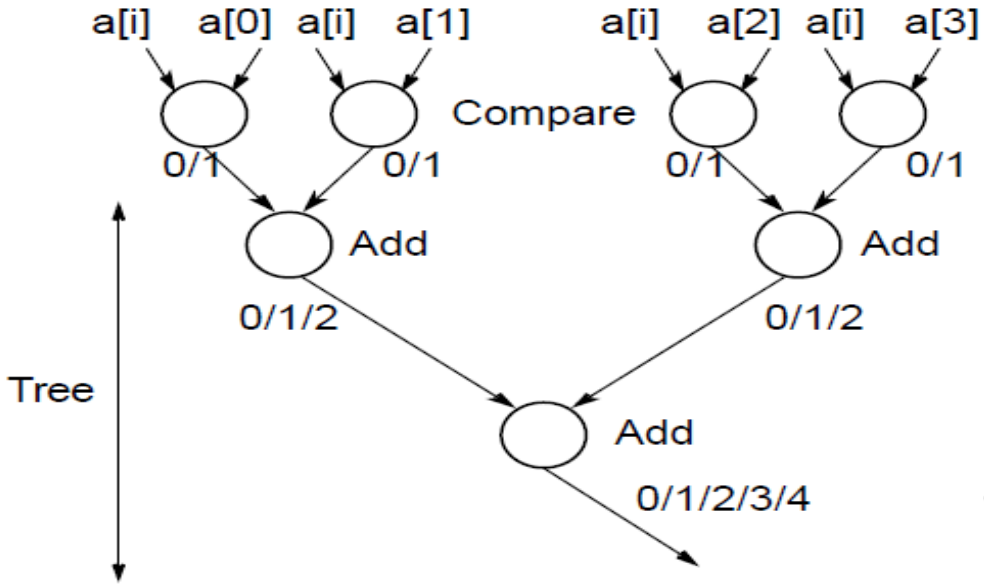
```
for(i=0; i<n; i++) {
        rank=0;
        for(j=0; j<n; j++) {
                if (a[i] < a[j]) rank++;
         }
         b[rank] = a[i];
}
```

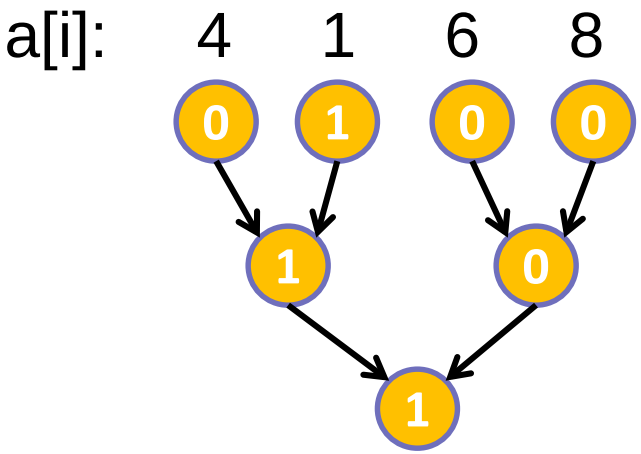


- Parallel: $O(n)$
  - use one processor for each number

Rank Sort with n^2 Processors

- We can use n processor to compute the rank of a number in parallel (summation parallel algo.)
- Time complexity? $O(\log n)$
- How about parallel efficiency? Actually decreased!

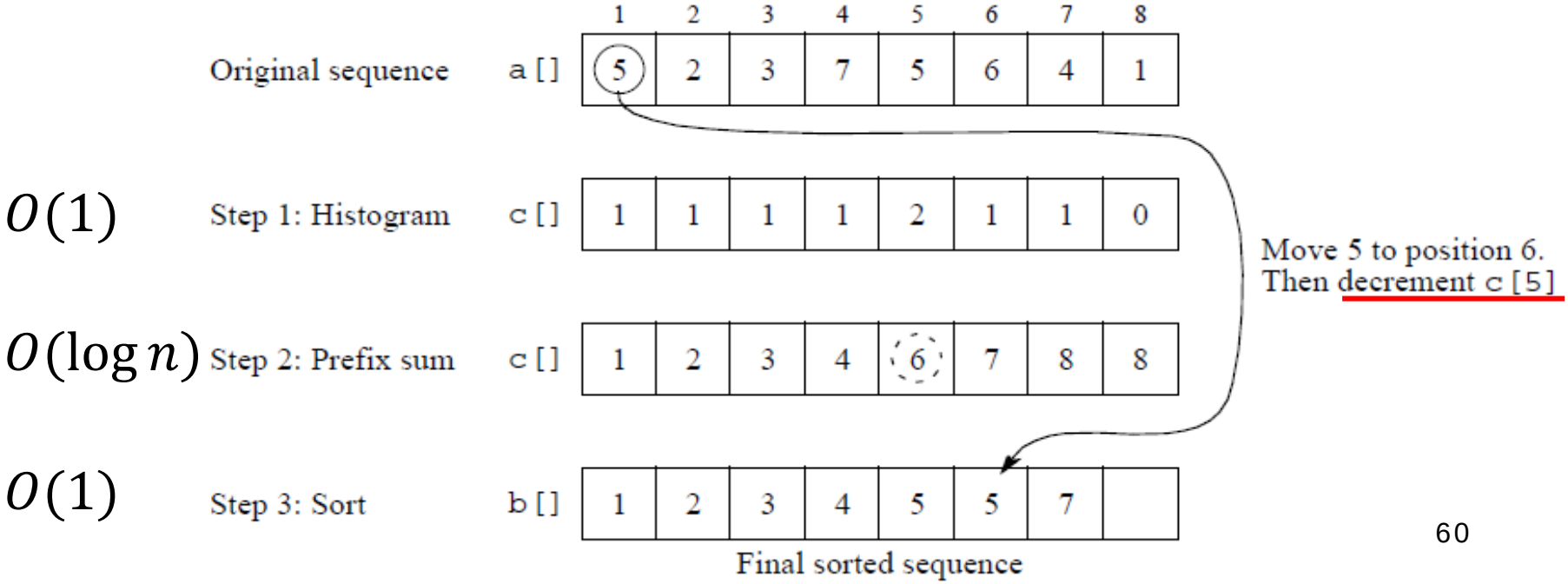




Compute the rank for a[0]
in O(log n) steps with n processors

Counting Sort

- If the numbers are integers and the range is known
  - Coding the rank sort algorithm to reduce the sequential time from $O(n^2)$ to $O(n)$
- Use array c[] to count the histogram of values
- Complexity
  - Sequential: $O(n+m)$; Parallel with n processors: $O(\log n)$



Summary

- With #processors = n

	Sequential	Parallel
Bucket Sort	$O(n \log n)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n)$
Bitonic mergesort	$O(n \log n)$	$O(\log^2 n)$
Rank Sort	$O(n^2)$	$O(n)$ $O(\log n)$
Counting Sort*	$O(n)$	$O(\log n)$

With n^2 processors

*Special case with known value range

N-Body Problem

- Newtonian laws of physics
  - The gravitational force between two bodies of masses m_a & m_b :

$$F = \frac{G m_a m_b}{r^2}$$

  - Subject to the force, acceleration occurs

$$F = m \times a$$

- Let the time interval be Δt & current velocity v^t, position x^t
  - New velocity v^{t+1} :

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

  - New position x^{t+1}:

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

Three-Dimensional Space

- Considering 2 bodies at (x_a, y_a, z_a)& (x_b, y_b, z_b)

$$r = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2}$$

- The forces, velocities and positions can be resolved in the three direction independently

$$F_x = \frac{G m_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

$$F_y = \frac{G m_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$

$$F_z = \frac{G m_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

N-Body Sequential Code

- Assume all bodies have the same mass ***m***

```
for (t=0; t<T; t++) {
    for (i=0; i<N; i++) {
        F = Compute_Force(i);          // compute force in O(N^2)
        v_new[i] = v[i] + F *dt / m;   // compute new velocity
        x_new[i] = x[i] + v_new[i] * dt;  // compute new position
    }
    for(i=0; i<N; i++){
        x[i] = x_new[i];               // update position
        v[i] = v_new[i];               // update velocity
    }
}
```

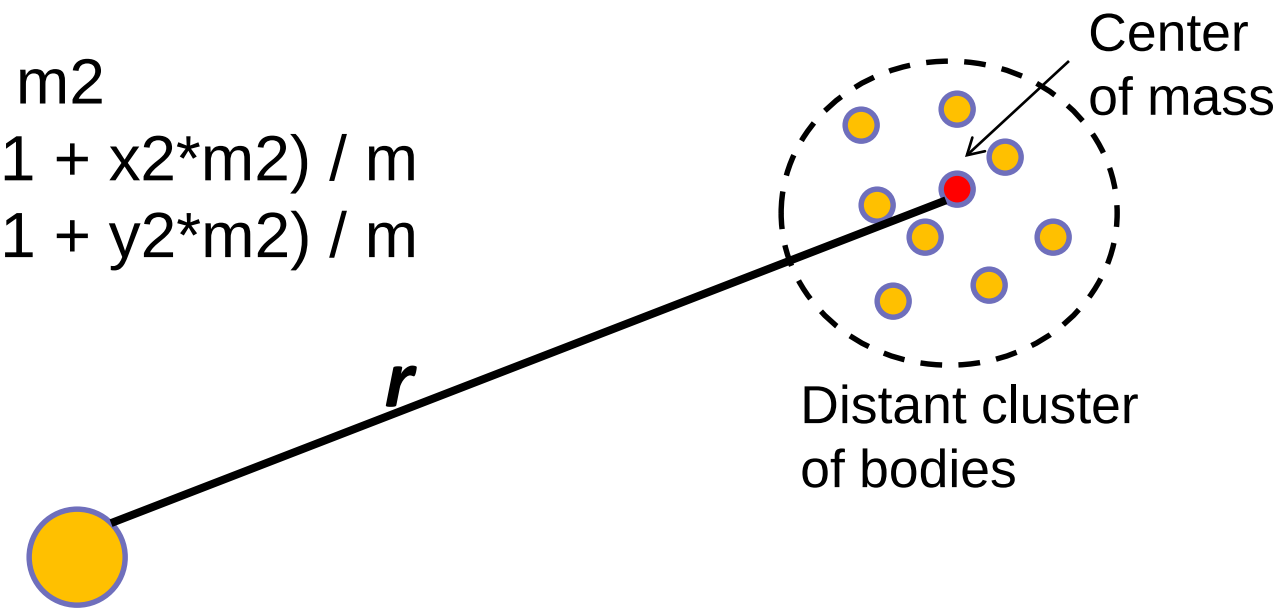
- Non-feasible as *N* increases due to $O(N^2)$complexity

Approximate Algorithms

- Reduce time complexity by approximating a cluster of bodies as a single distant body

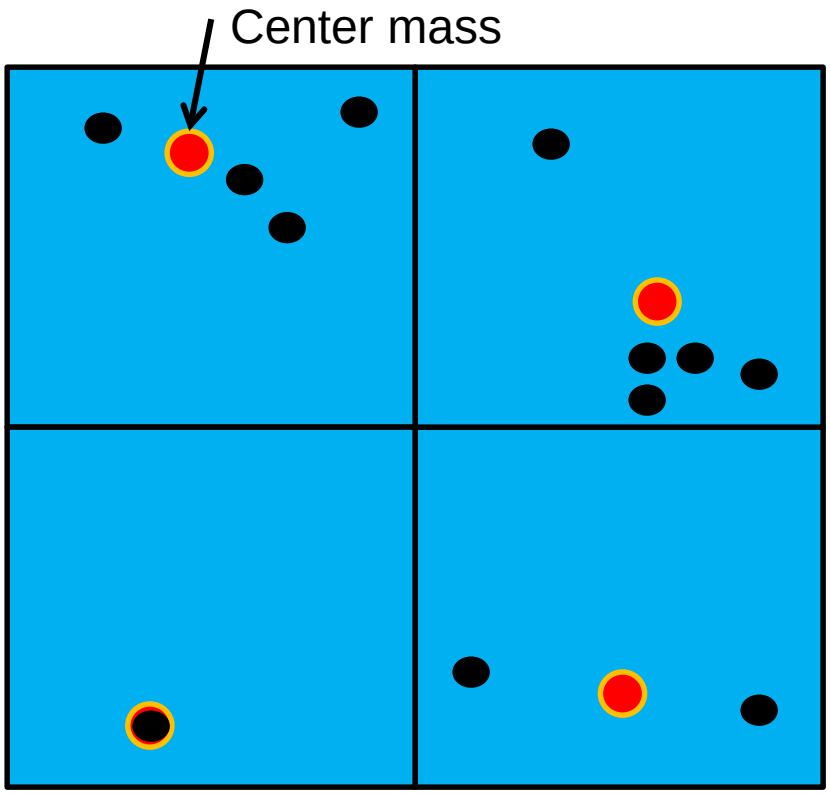
How to find those clusters of bodies?

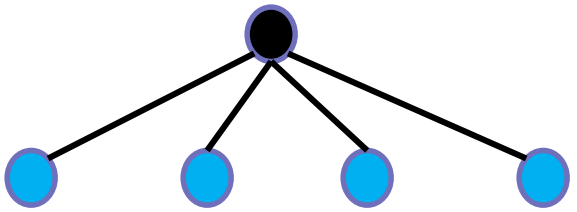
m = m1 + m2
x = (x1*m1 + x2*m2) / m
y = (y1*m1 + y2*m2) / m



Barnes-Hut Algorithm

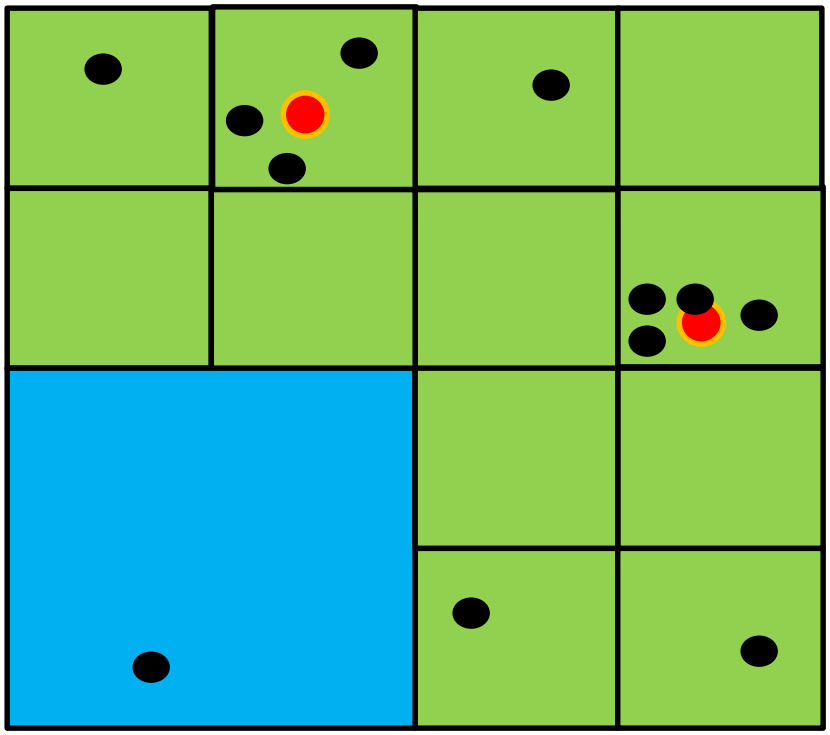
- Step1: Recursively divide space by two in each dimensions
  - Record the center mass and position of each internal node

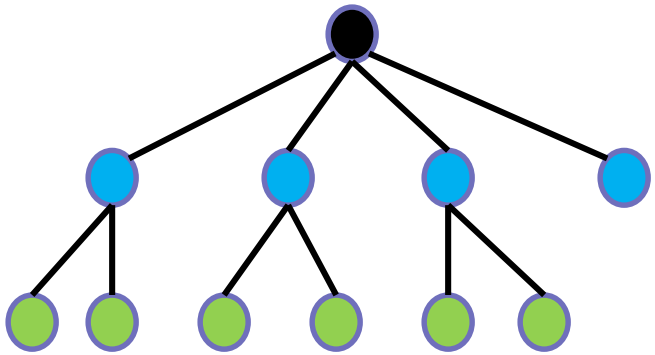




Barnes-Hut Algorithm

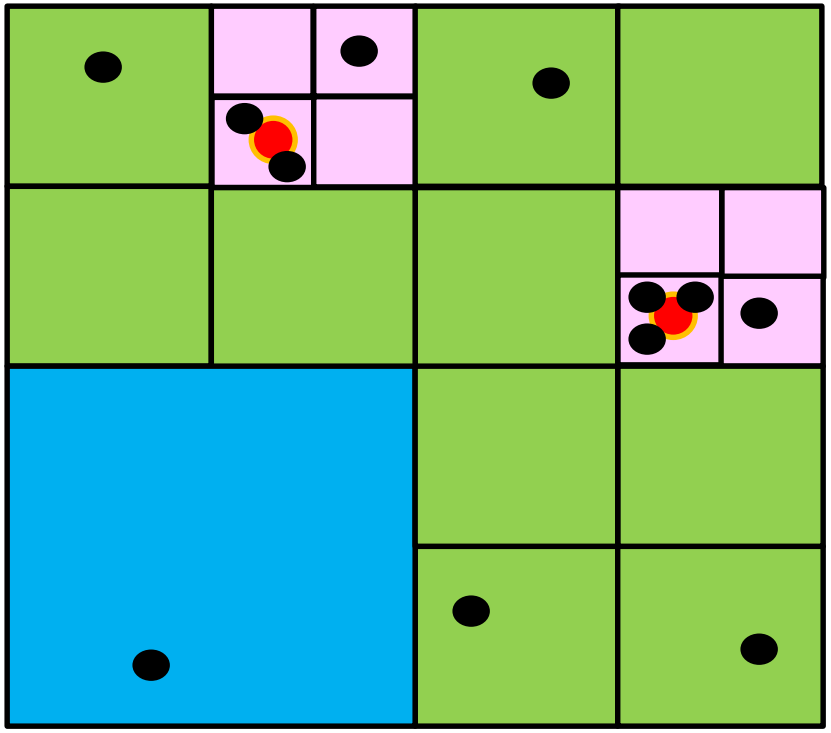
- Step1: Recursively divide space by two in each dimensions
  - Record the center mass and position of each internal node

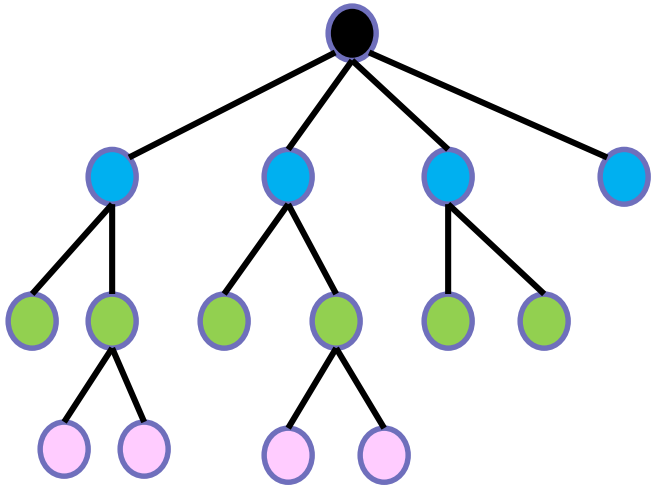




Barnes-Hut Algorithm

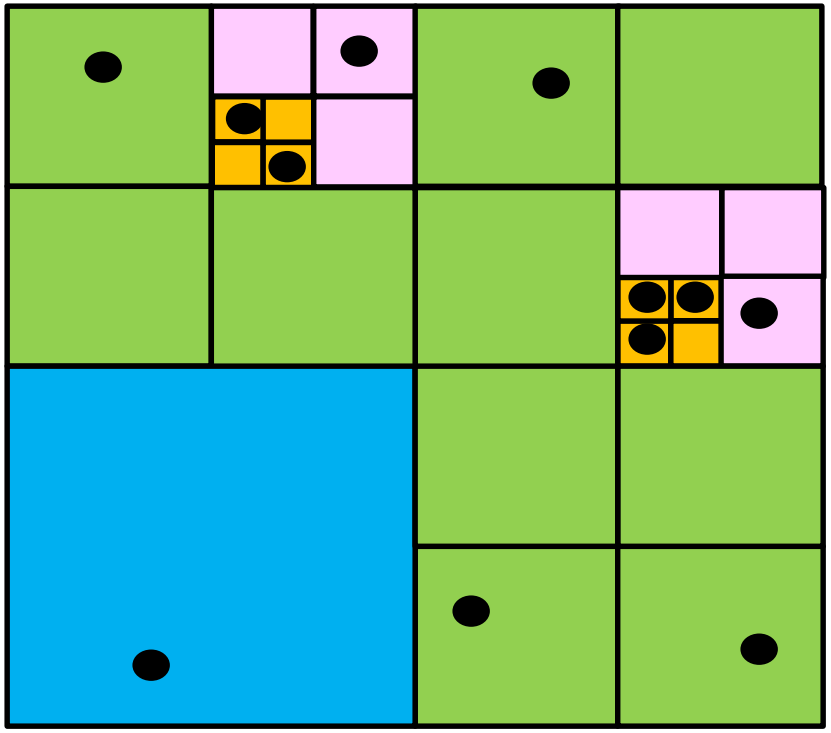
- Step1: Recursively divide space by two in each dimensions
  - Record the center mass and position of each internal node

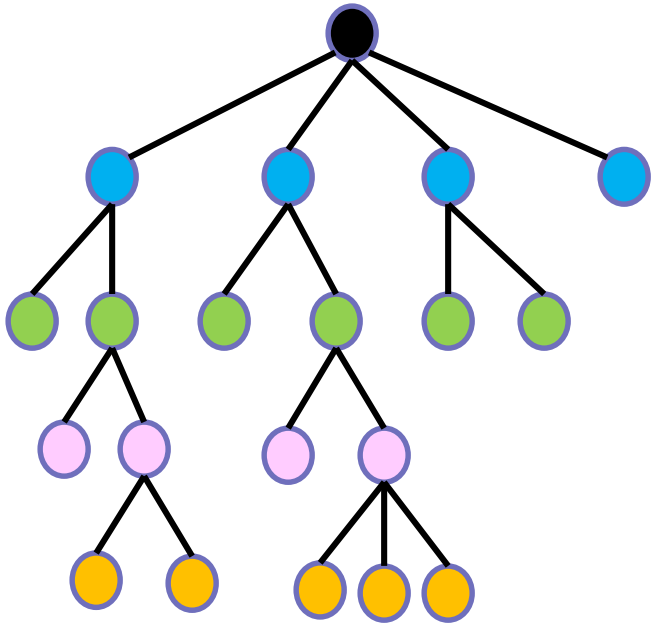




Barnes-Hut Algorithm

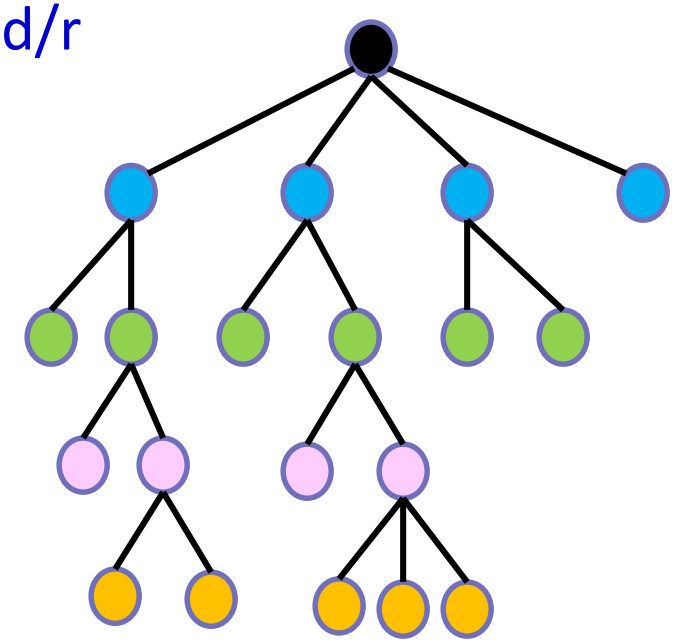
- Step1: Recursively divide space by two in each dimensions
  - Record the center mass and position of each internal node



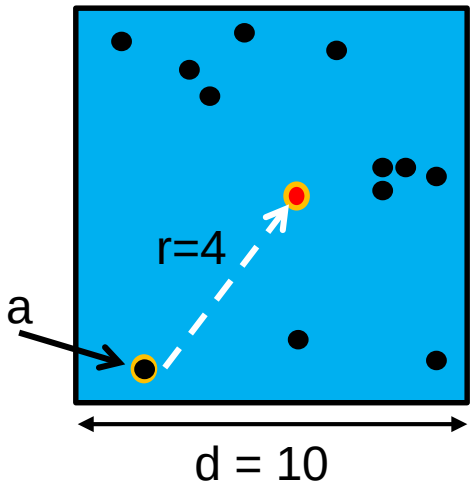


Barnes-Hut Algorithm

- Step2: Compute approximate forces on each object
  1. traverse the nodes of the tree, starting from the root.
  2. If the center-of-mass of an **internal node** is **sufficiently far** from the body, approximate the internal node as a single body
  - Far is determined by a parameter: θ=d/r
    - ***r***: the distance between the body and the node's center-of-mass
    - ***d***: the width of the region

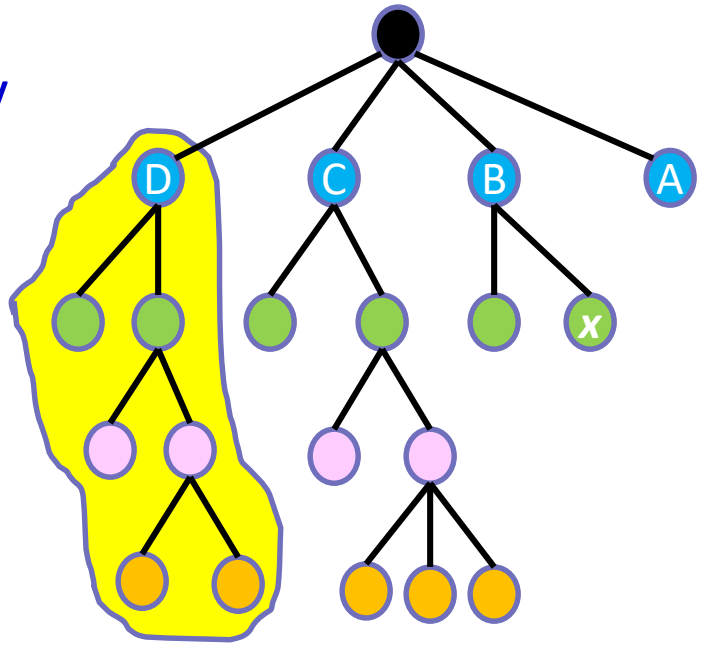


Example: θ=0.5
d/r=2.5 > θ



Barnes-Hut Algorithm

- Step2: Compute approximate forces on each object
  1. Traverse the nodes of the tree, starting from the root.
  2. If the center-of-mass of an **internal node** is **sufficiently far** from the body, approximate the internal node as a **single body**
  - Far means $d/r < \theta$ (e.t. $0 < \theta < 1$)
    - ***r***: the distance between the body and the node's center-of-mass
    - ***d***: the width of the region



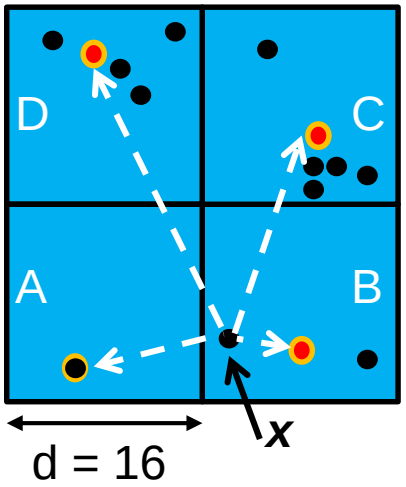
Example: θ=1

$d/r_A = 16/10 > \theta$

$d/r_B = 16/2 > \theta$

$d/r_C = 16/15 > \theta$

$d/r_D = 16/20 < \theta$



Barnes-Hut Algorithm

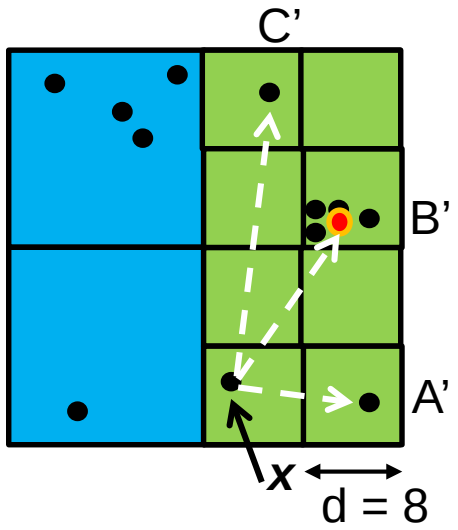
- Step2: Compute approximate forces on each object
  3. If it is a leaf node, calculate the force and add to the object.
  4. Otherwise, recursively compute the force from children of the internal node.

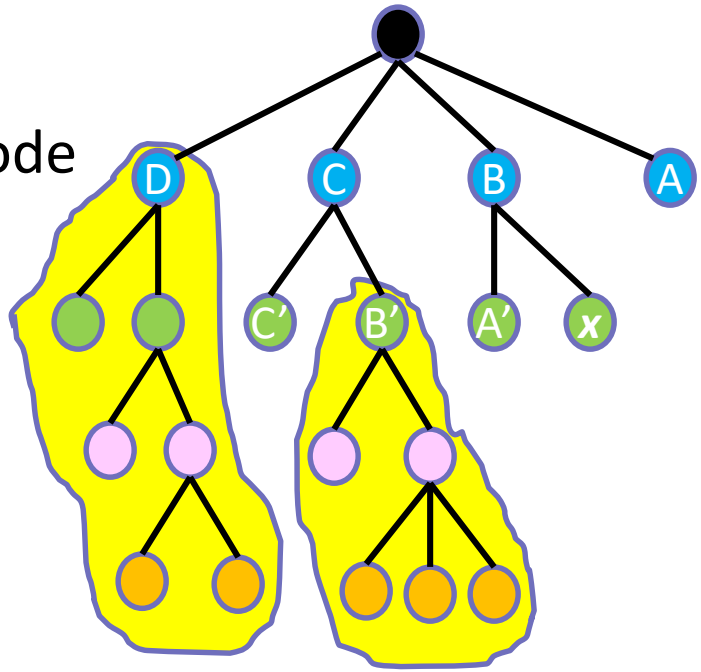
Example: θ=1

$d/r_{A'}=8/7 > \theta$ ➔ A' is a leaf node

$d/r_{B'}=8/15 < \theta$ ➔ B' treated like a single node

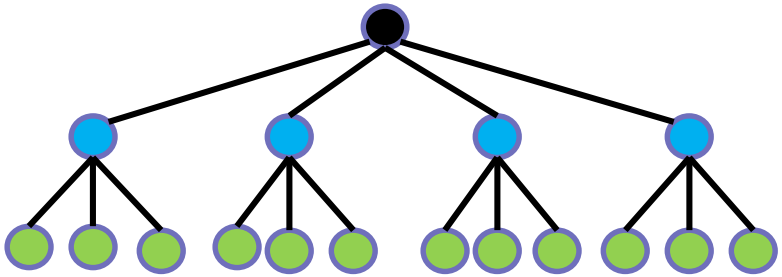
$d/r_{C'}=8/20 < \theta$ ➔ C' is a leaf node

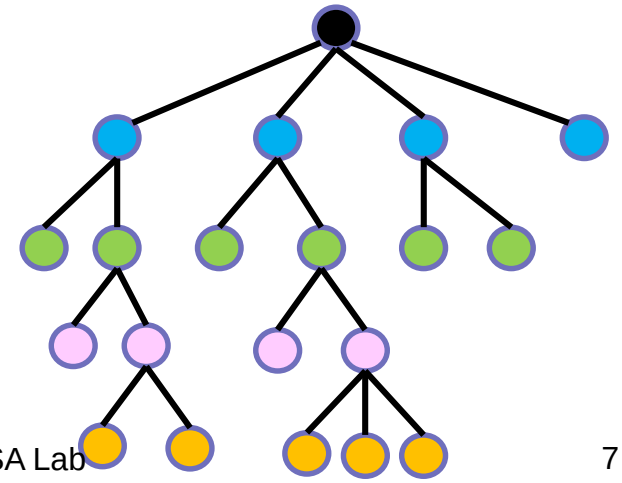




Barnes-Hut Algorithm

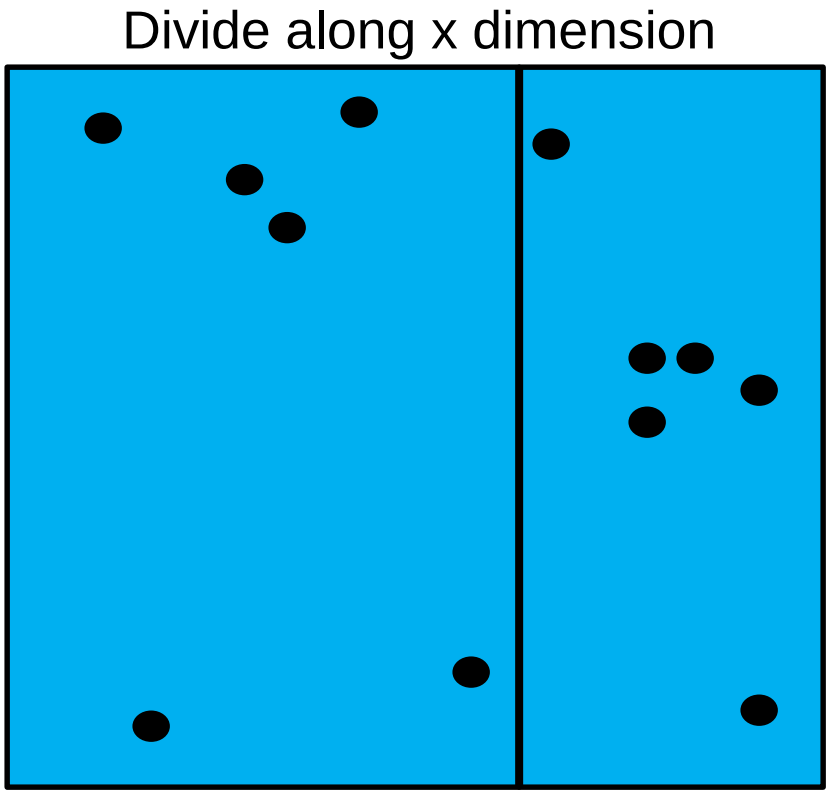
- θ controls the accuracy and approximation error of the algorithm
  - θ = 0 ➔ ***d*/*r* ALWAYS** larger than θ ➔ same as brute force
  - θ = 1 ➔ most likely only need to consider the object within the same cluster/region
- If the tree is balanced, the complexity is O(***n***log***n***)
  - But in general , the tree could be very unbalanced ……..
- The tree must be re-built for each time interval

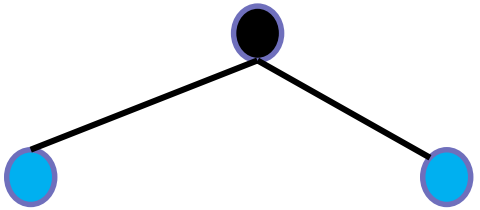




Orthogonal Recursive Bisection Method

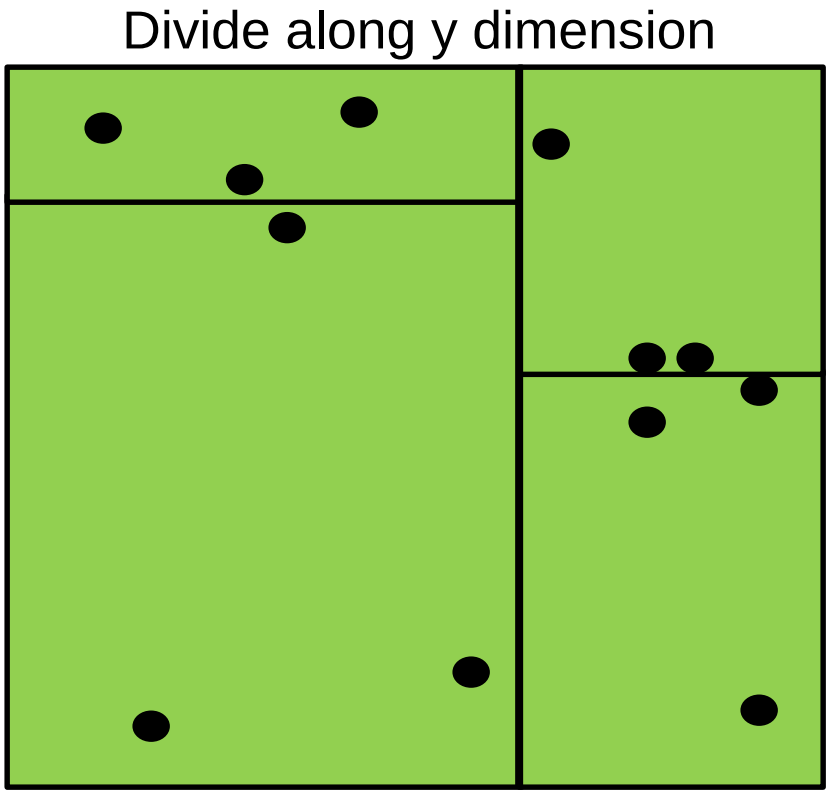
- Recursively evenly divide space with the same number of bodies in each of the dimensions

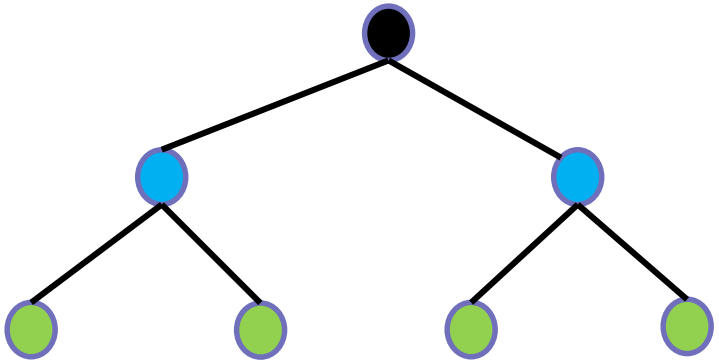




Orthogonal Recursive Bisection Method

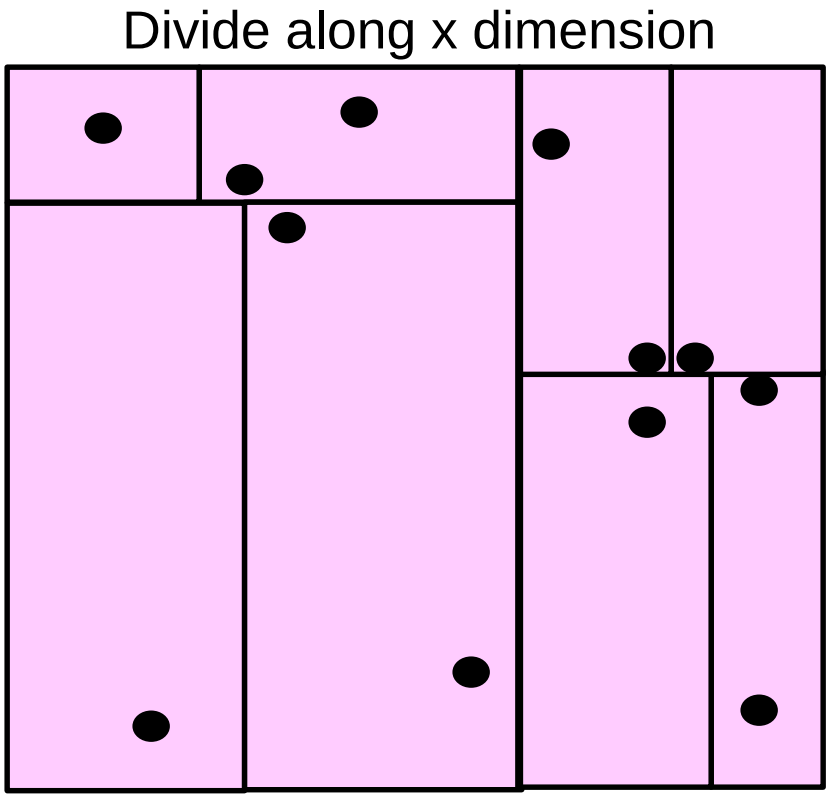
- Recursively evenly divide space with the same number of bodies in each of the dimensions

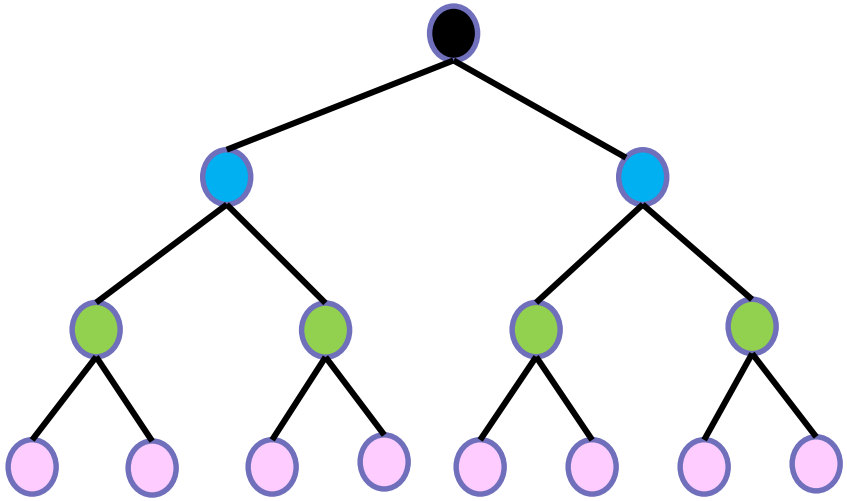




Orthogonal Recursive Bisection Method

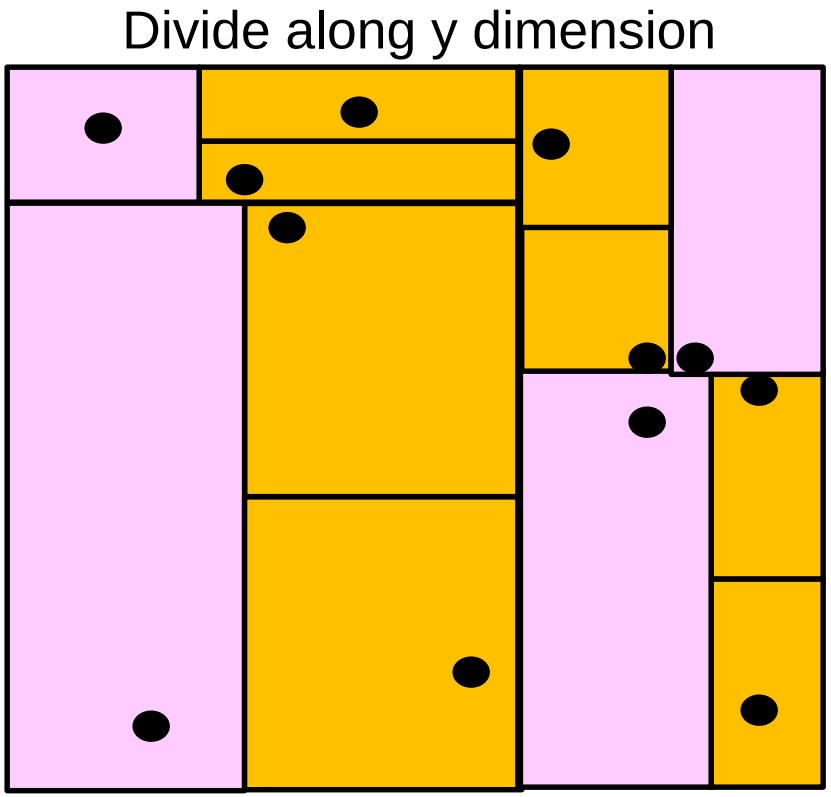
- Recursively evenly divide space with the same number of bodies in each of the dimensions

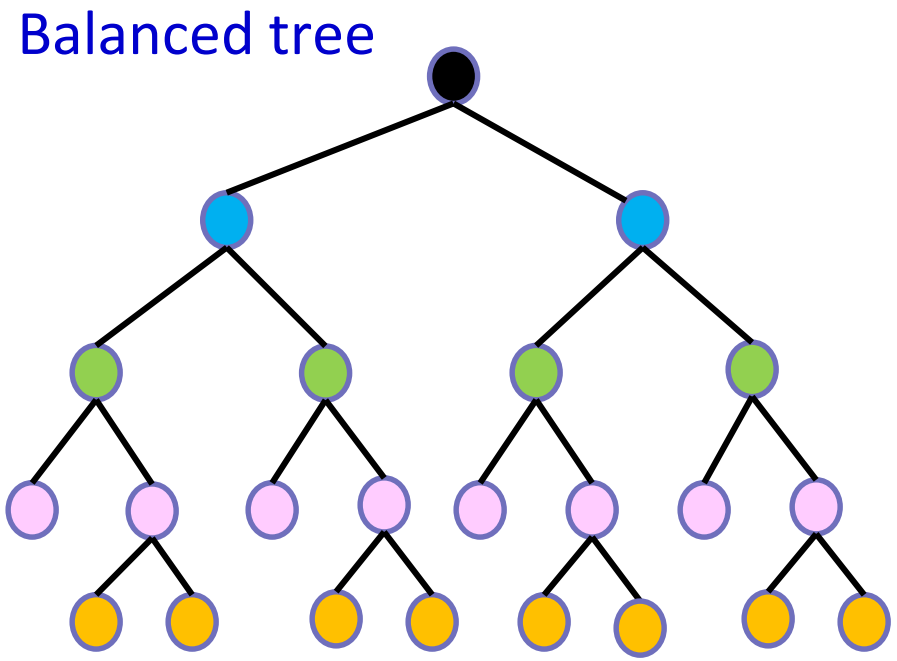




Orthogonal Recursive Bisection Method

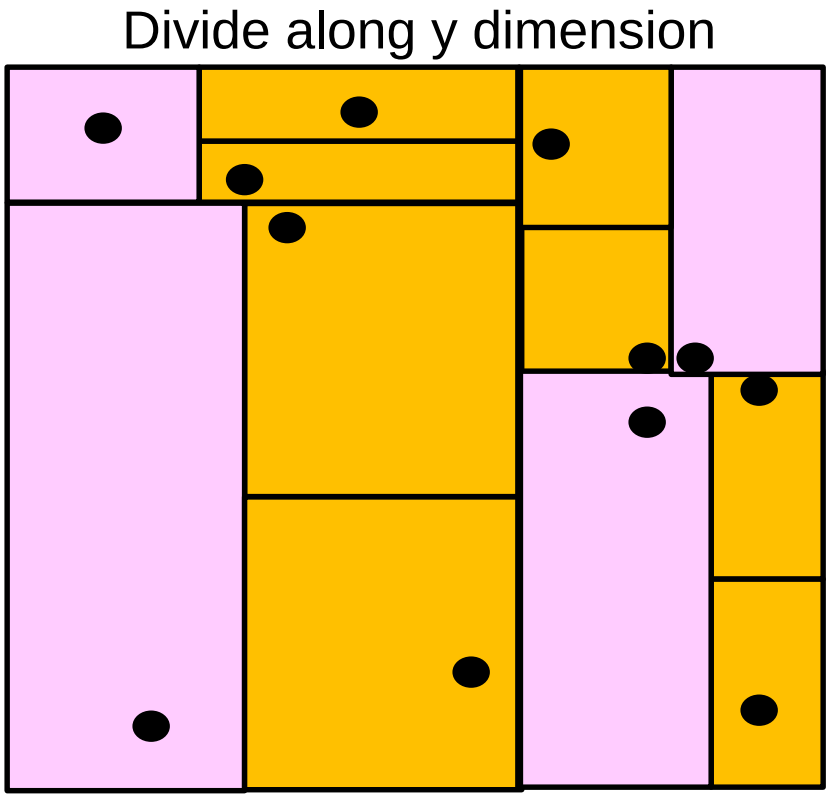
- Recursively evenly divide space with the same number of bodies in each of the dimensions

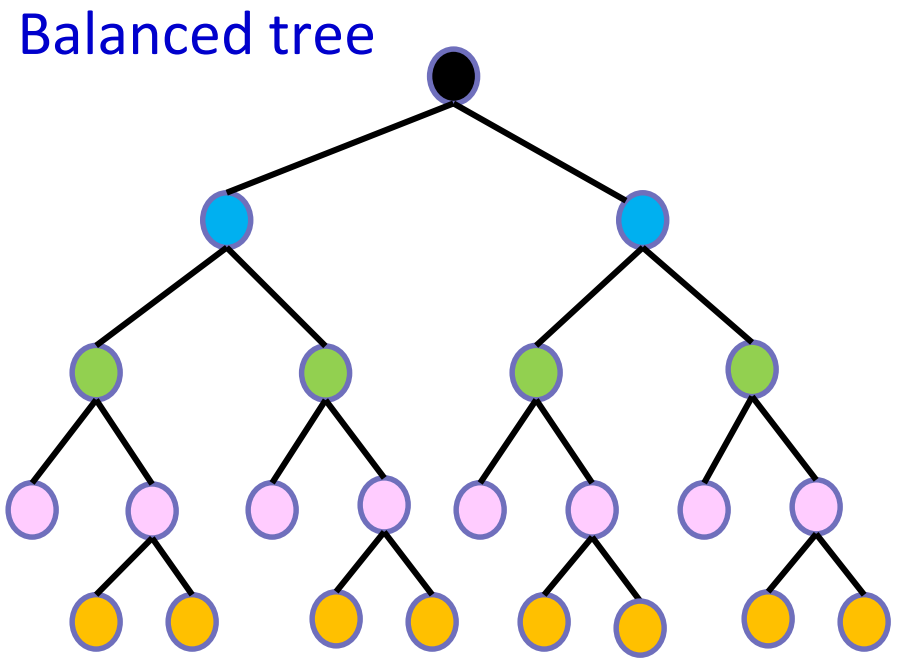




Orthogonal Recursive Bisection Method

- It is more balanced, but less accurate
  - Objects close to each other may not in the same cluster



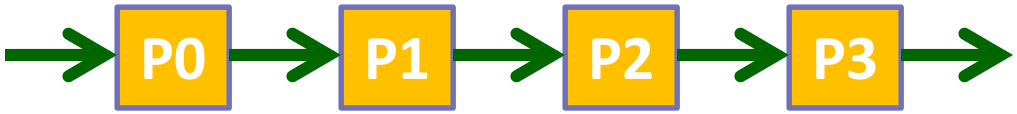


Outline

- Embarrassingly Computations
- Load Balancing & Termination
- Divide-And-Conquer Computations
- Pipelined Computations
  - Adding Numbers
  - Sorting Numbers
  - Linear Equation Solver
- Synchronous Computations

What is Pipelined Computations

- A problem is divided into a series of tasks
- Tasks have to be completed one after the other
- Each task will be executed by a separate process or processor



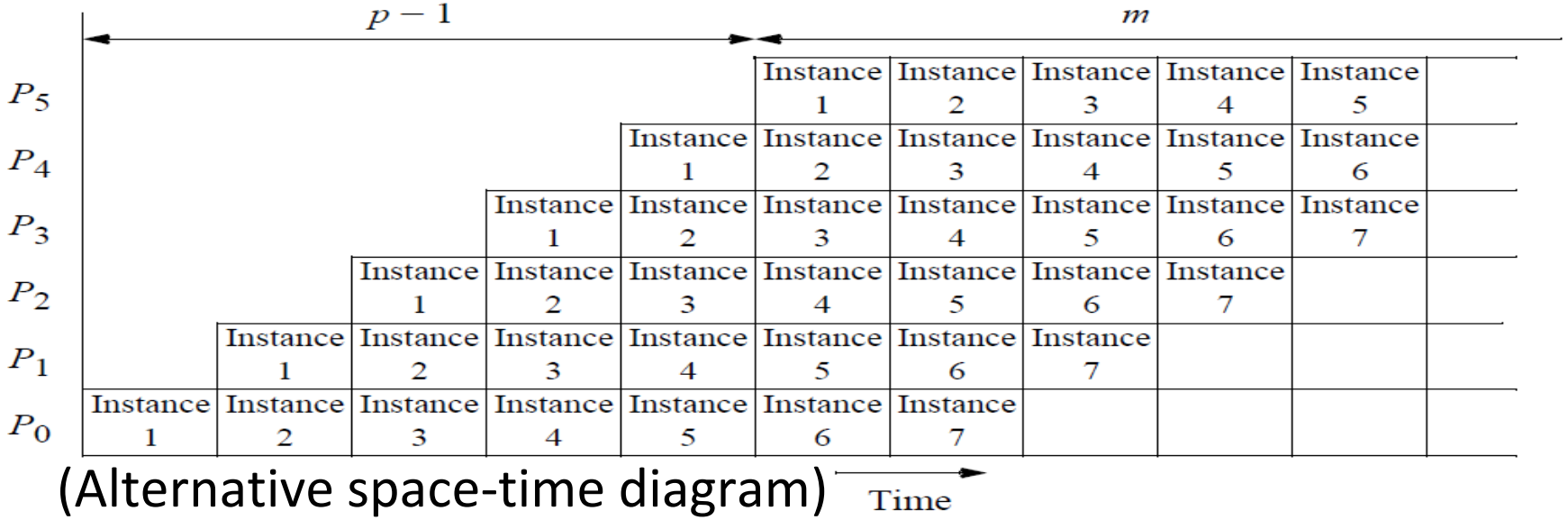
Types of Pipelined Computations

- Pipelined approach can provide increased speed under three types of computations:

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

Type 1 Pipelined Computations

1. If more than one **instance** of the complete problem is to be executed

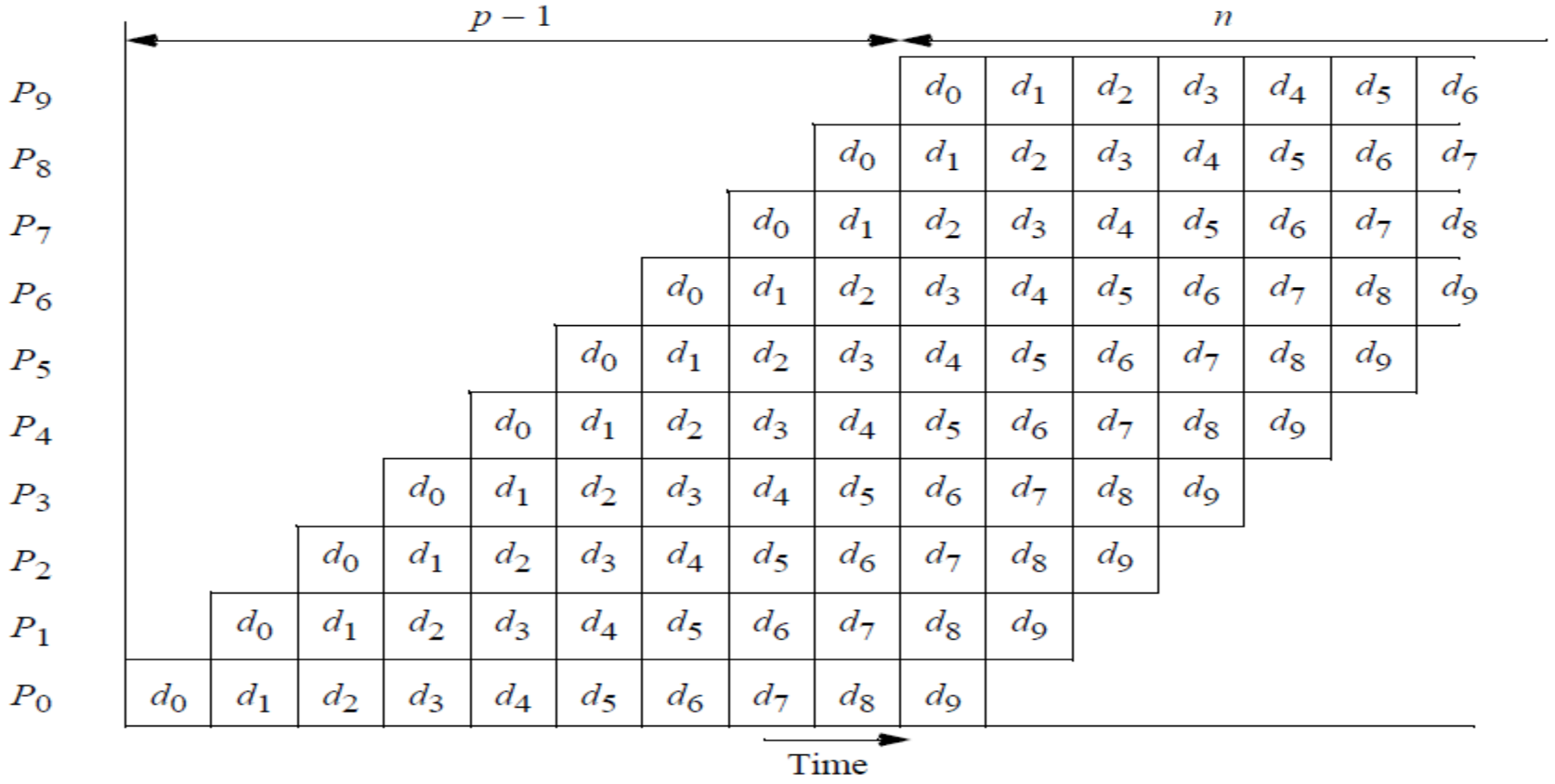


(Alternative space-time diagram)

- After the first (p-1) cycles, one problem instance is completed in each pipeline cycle
- The number of instance should be >> the number of processes

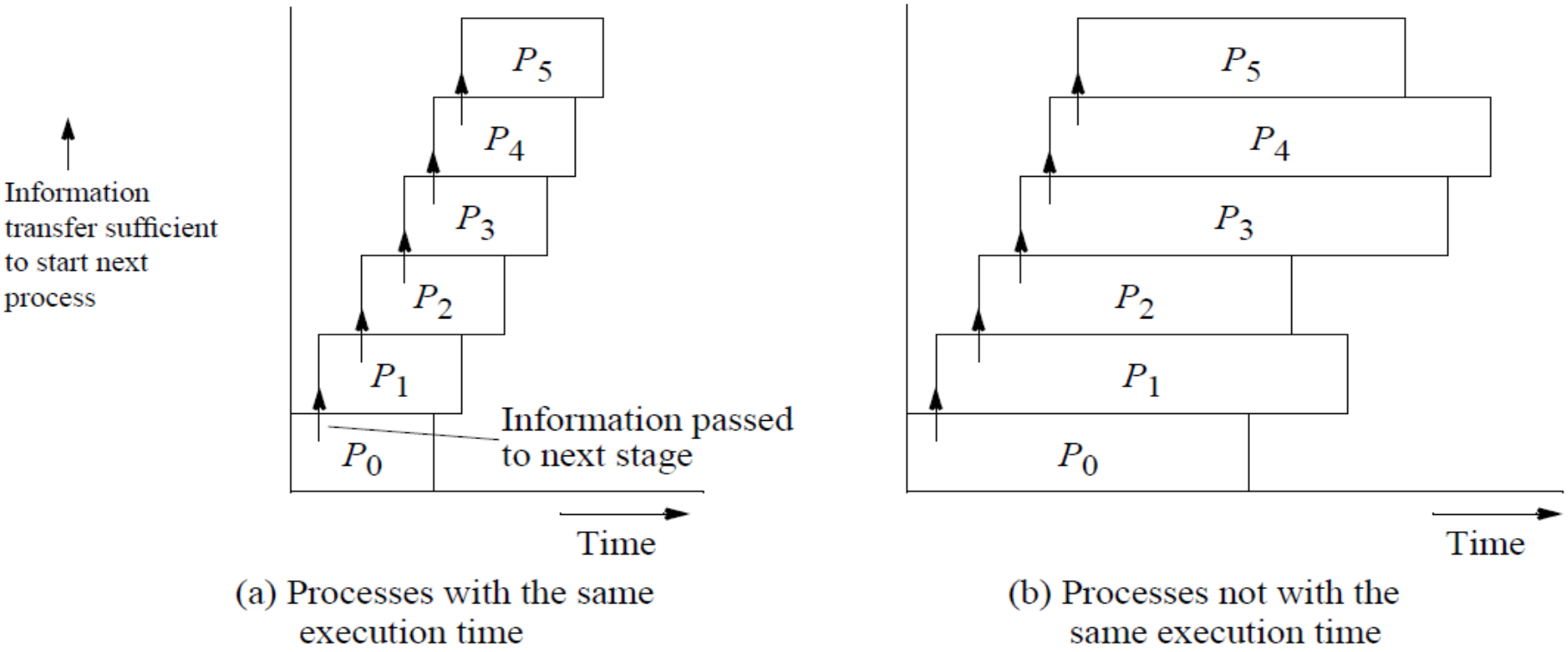
Type 2 Pipelined Computations

2. If a series of **data** items must be processed, each requiring multiple operations



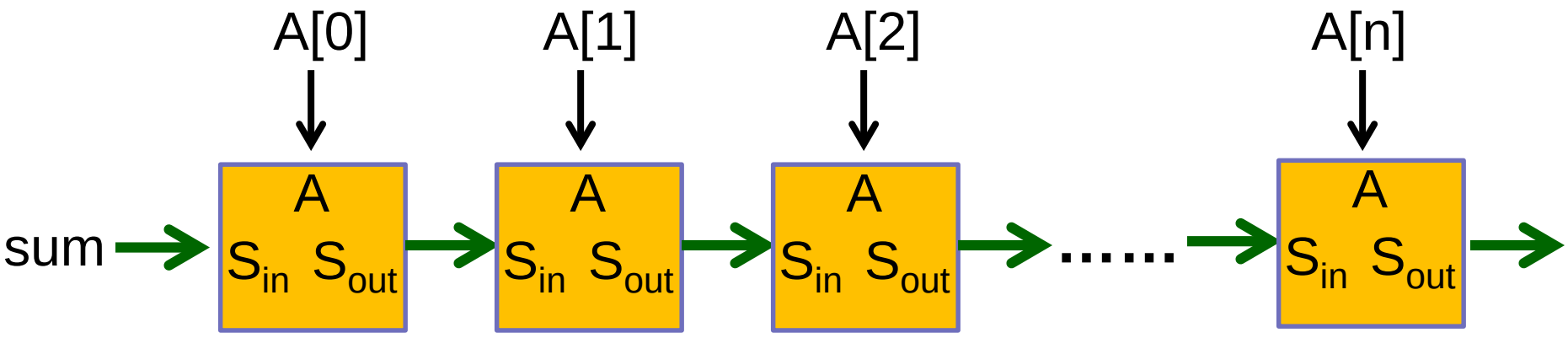
Types 3 Pipelined Computations

3. Only one problem instance, but each process can pass on information to the next process, **before it has completed**



Example1: Adding Numbers

- Compute sum of an array:
  - for(i=0; i<n; i++) sum += A[i]
- Pipeline for an unfolded loop:
  - sum += A[0], sum += A[1], sum += A[2], ……



Example1: Adding Numbers

- The basic code for Pi:

```
recv(&sum, P_{i-1});
sum += number;
send(&sum, P_{i+1});
```

- For the first process, P0:

```
send(&sum, P_{i+1});
```

- For the last process, Pn-1:

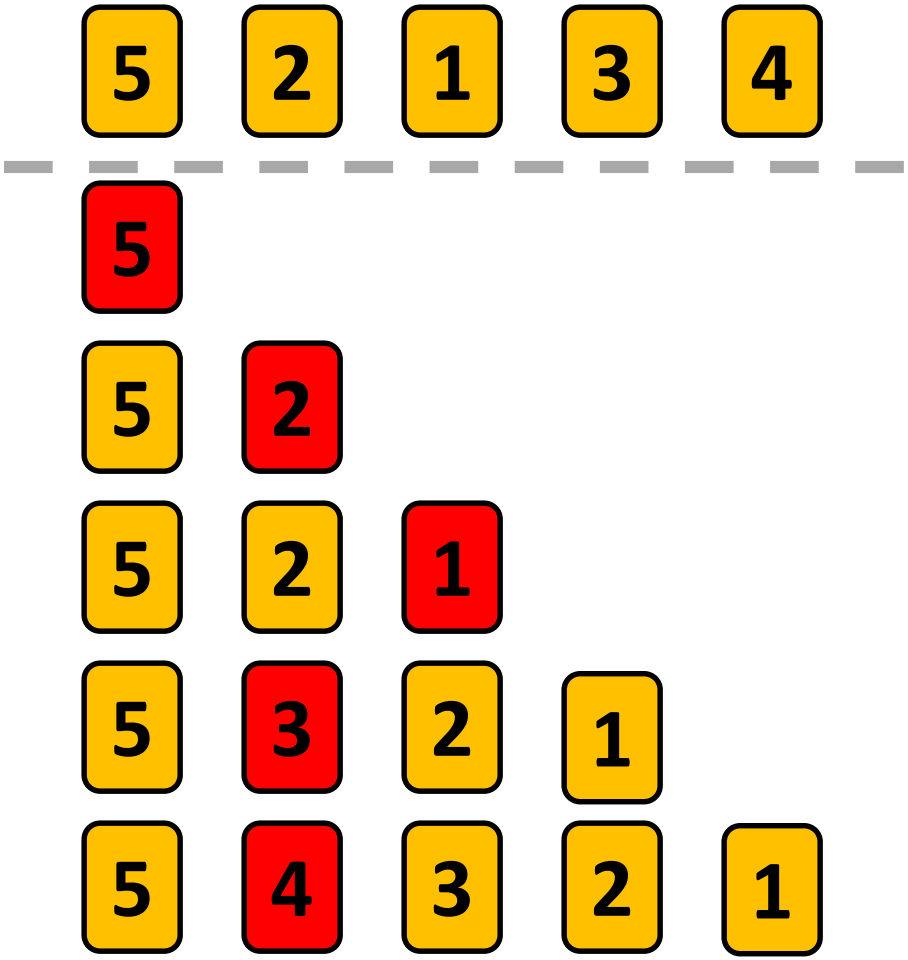
```
recv(&sum, P_{i-1});
sum += number;
```

- SPMD Program:

```
// code for process Pi
if (Pi != P0) {
    recv(&sum, P_{i-1});
    sum += number;
}
if (Pi != Pn) {
    send(&sum, P_{i+1});
}
```

Example2: Sorting Numbers

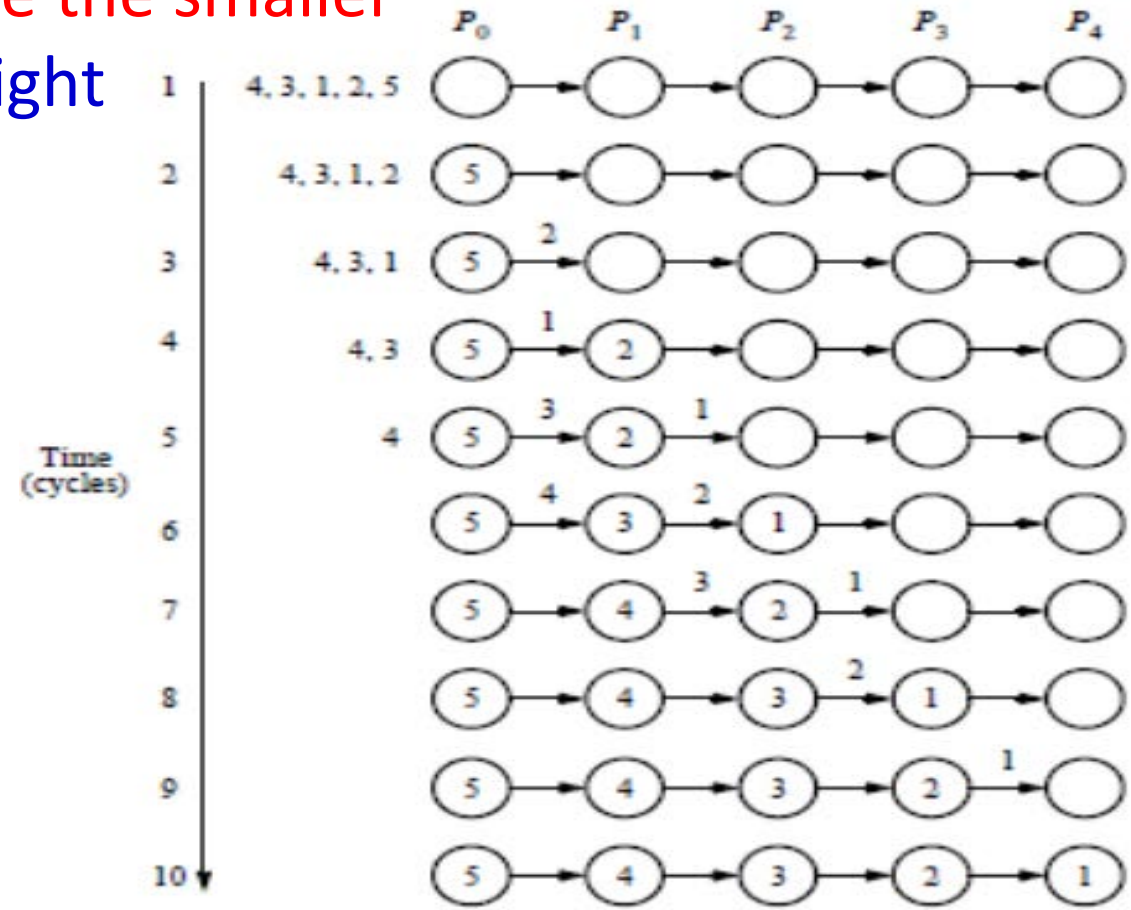
- Insertion Sort:



Example2: Sorting Numbers

- Insertion Sort:
  - Each process holds one number
  - Compare & move the smaller number to the right

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



Example 3: Linear Equation Solver

- Special linear equations of "upper-triangular" form
  - ***a***'s and ***b***'s are constants, ***x***'s are unknown to be found

$$\begin{pmatrix} a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots\cdots & a_{n-1,n-1} \\ a_{n-2,0} & a_{n-2,1} & \cdots\cdots & a_{n-2,n-2} & 0 \\ \vdots & \vdots & \vdots & 0 & 0 \\ a_{1,0} & a_{1,1} & 0 & 0 & 0 \\ a_{0,0} & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{pmatrix}$$

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \cdots + a_{n-1,n-1}x_{n-1} = \mathrm{b}_{n-1}$$

$$\vdots$$

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = \mathrm{b}_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = \mathrm{b}_1$$

$$a_{0,0}x_0 = \mathrm{b}_0$$

Example 3: Linear Equation Solver

- Back Substitution
  - x_0 is found from the last equation

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

  - Value for x_0 is substituted into the next equation

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

  - Values for x_0, x_1 are substituted into the next equation

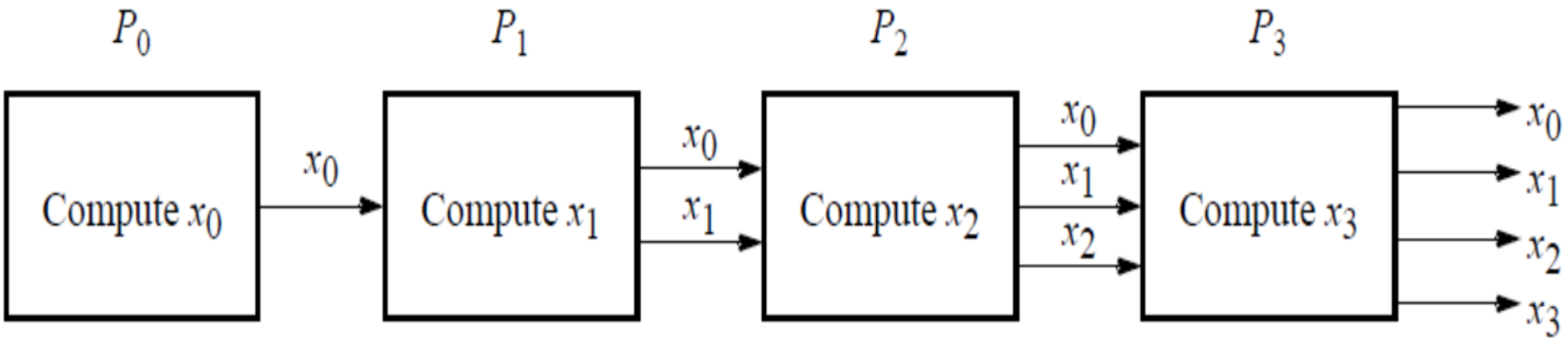
$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

  - So on until all unknowns are found …

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

Example 3: Linear Equation Solver

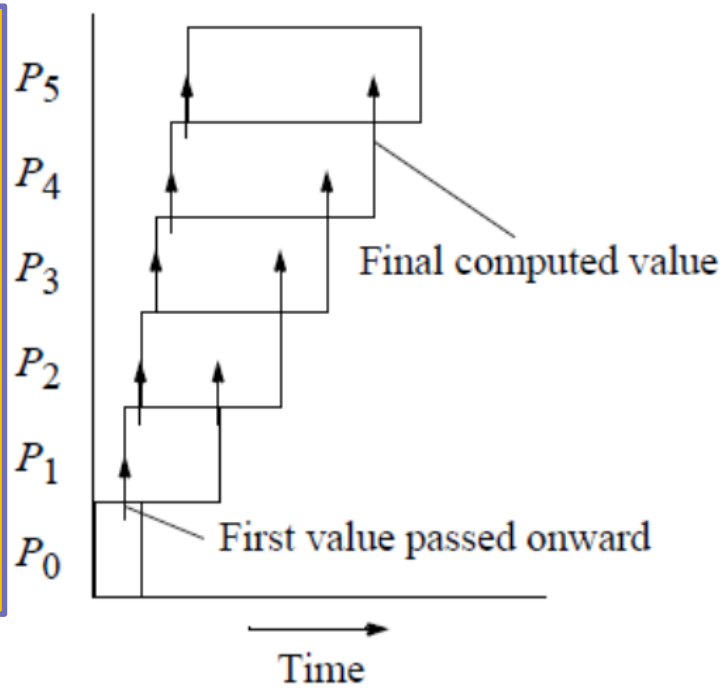
- First pipeline stage computes x_0 and passes x_0 onto the second stage, which computes x_1 from x_0 and passes both x_0 and x_1 onto the next stage, which computes x_2 from $x_0$and x_1, and so on



Example 3: Linear Equation Solver

- Parallel Code

```
// code for Pi
sum = 0;
for (j=0; j<i; j++) {          // compute partial result
    recv(&x[j], P_{i-1});      // once data is available
    send(&x[j], P_{i+1});
    sum += a[i][j]*x[j];
}
x[i] = (b[i] - sum) / a[i][j]; // send out final result to
send(&x[j], P_{i+1});          // next process
```



- Time complexity:
  - $O(n^2)$ without passing forward
  - $O(n)$ with passing forward

Outline

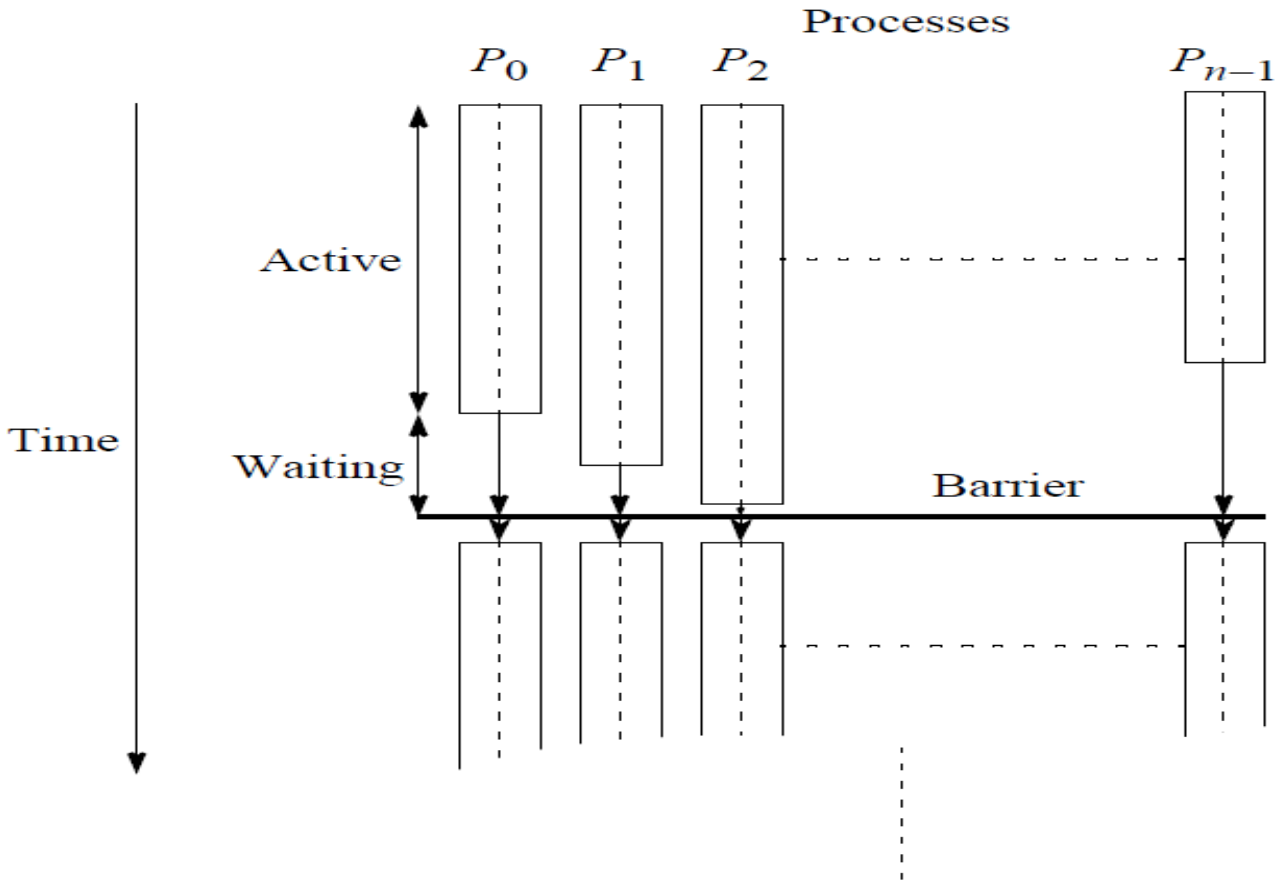
- Embarrassingly Computations
- Load Balancing & Termination
- Divide-And-Conquer Computations
- Pipelined Computations
- Synchronous Computations
  - Prefix Sum
  - System of Linear Equations
  - Heat Distribution

Synchronous Computations

- **Definition:** all the processes ***synchronized*** at regular points
- **Barrier**: Basic mechanism for synchronizing processes
  - Inserted at the point in each process where it must wait
  - Message (token) is passed among processes for synchronization
- **Deadlock**: Common problem occurs from synchronization
  - Two or multiple processes waiting for each other

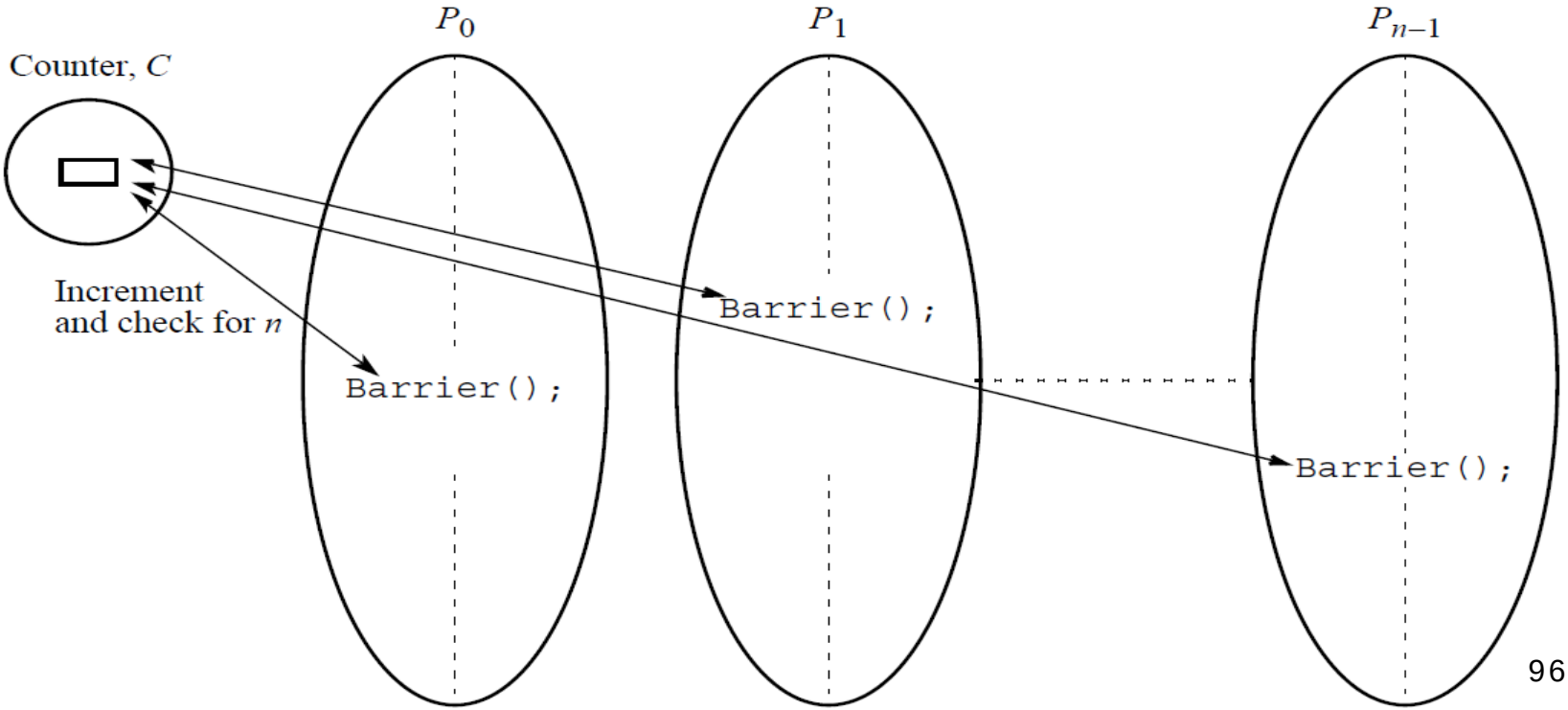
Barrier

- **All** processes can only continue from this *POINT* when **all** the processes have reached it



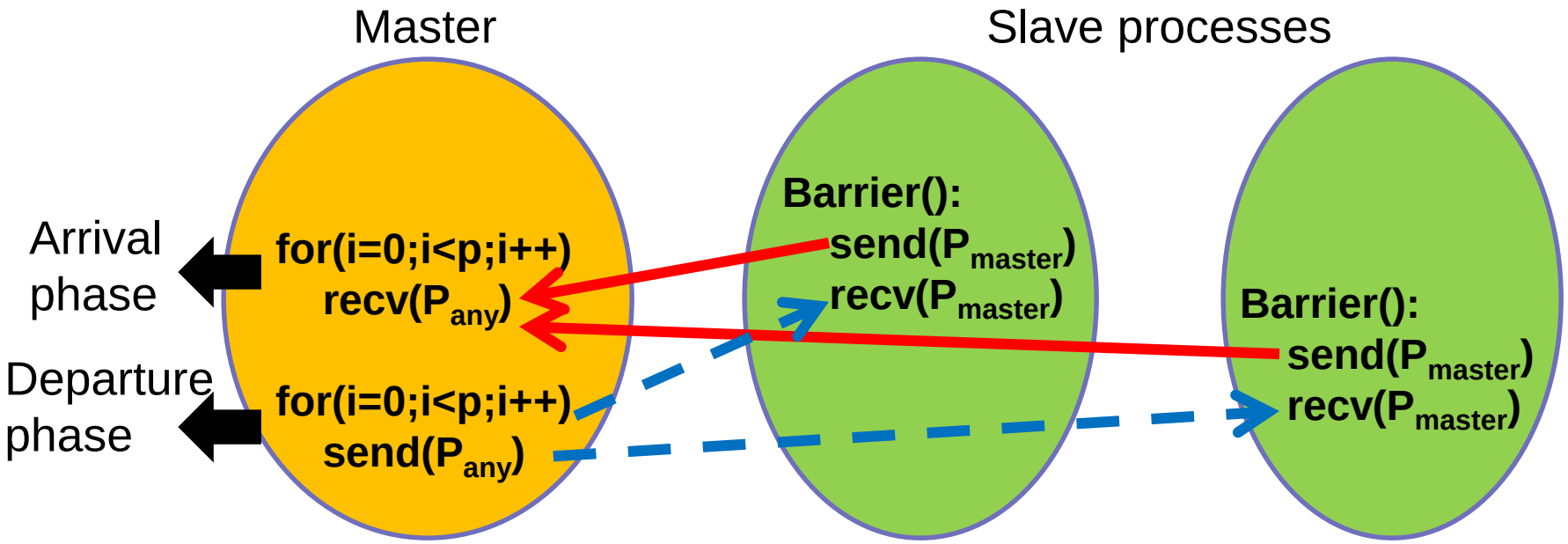
Counter Barrier Implementation

- A.k.a: Linear Barrier
  - Centralized counter: count # of processes reaching the barrier
  - Increase & check the counter for each barrier call
  - Processes is locked by the barrier call until counter == # processes



Counter Barrier Implementation

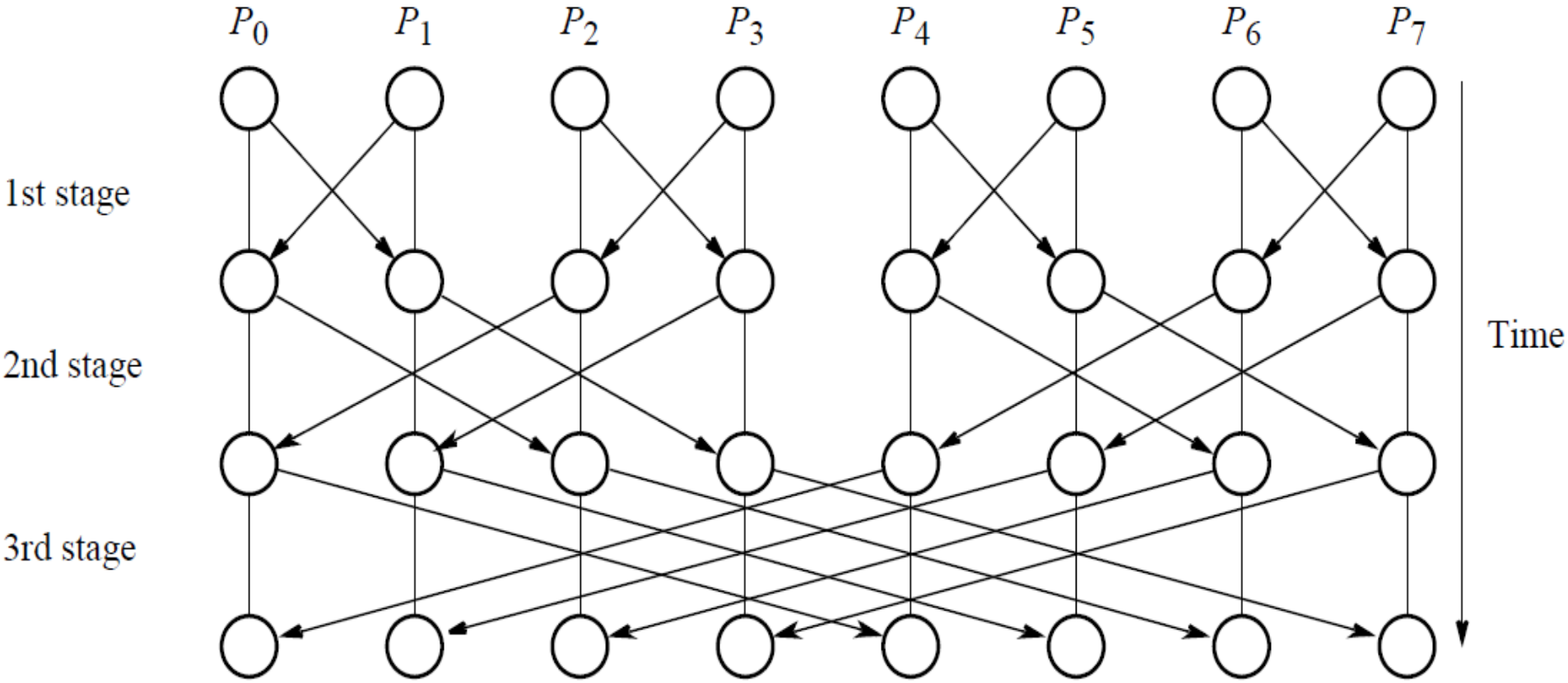
- Counter-based barrier often have two phases
  - **Arrival phase**: a process enters *arrival phase* and does not leave this phase until all processes have arrived in this phase
  - **Departure phase**: Processes are released after moving to the *departure phase*



- Slave processes is blocked by **recv()**
- Master could be a **bottleneck**

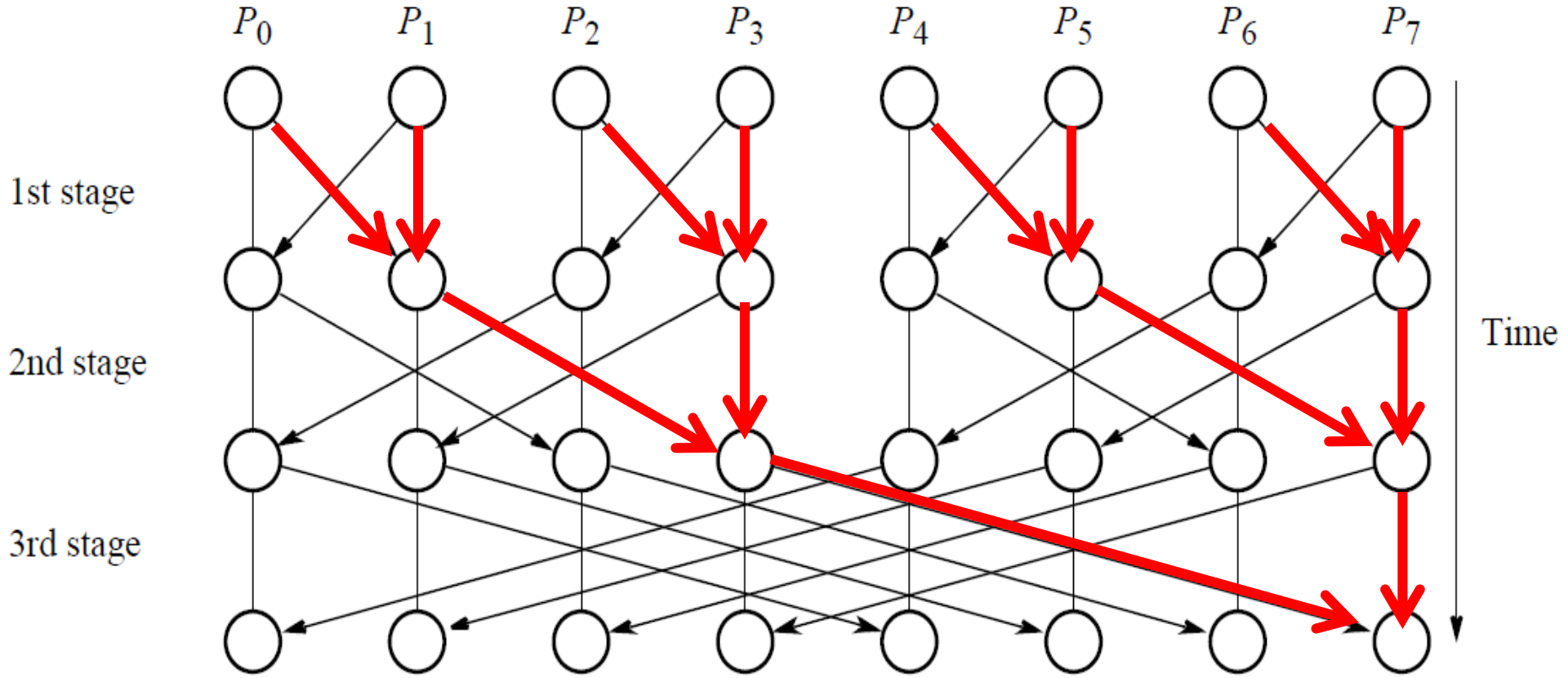
Butterfly Barrier Implementation

- At stage ***i***, each process passes a token to the process with 2^i distance away



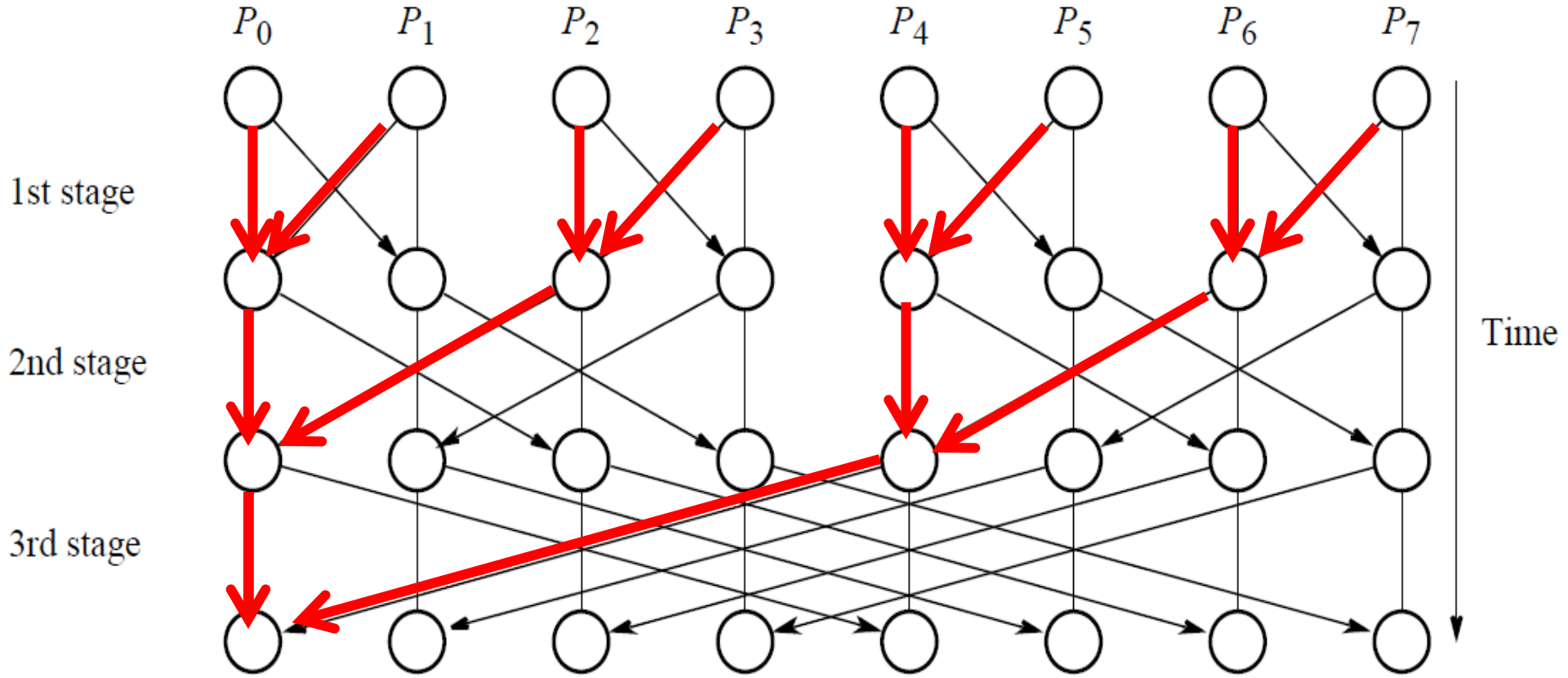
Butterfly Barrier Implementation

- At stage ***i***, each process passes a token to the process with 2^i distance away



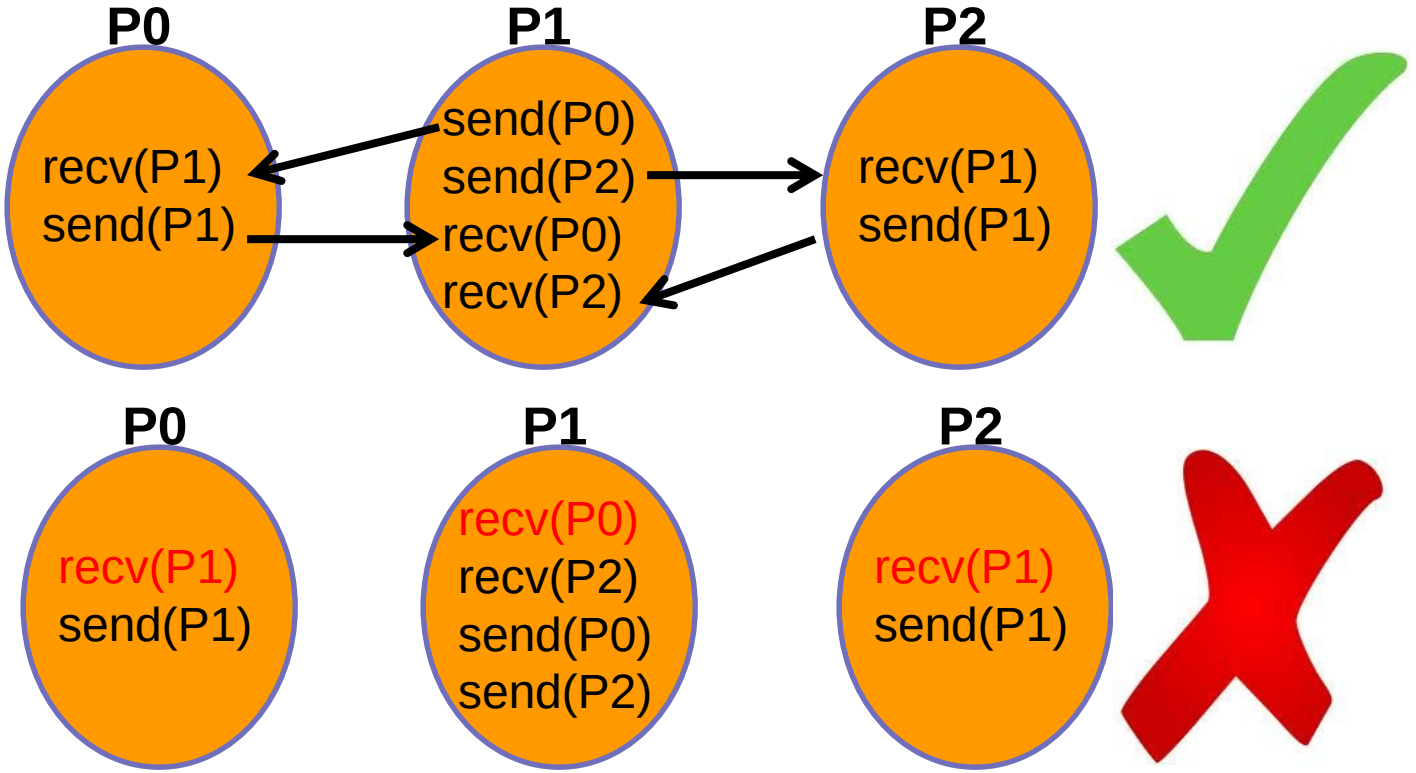
Butterfly Barrier Implementation

- At stage ***i***, each process passes a token to the process with 2^i distance away



Deadlock Problem

- A set of blocked processes each holding some resources and waiting to acquire a resource held by another process in the set
- Example:

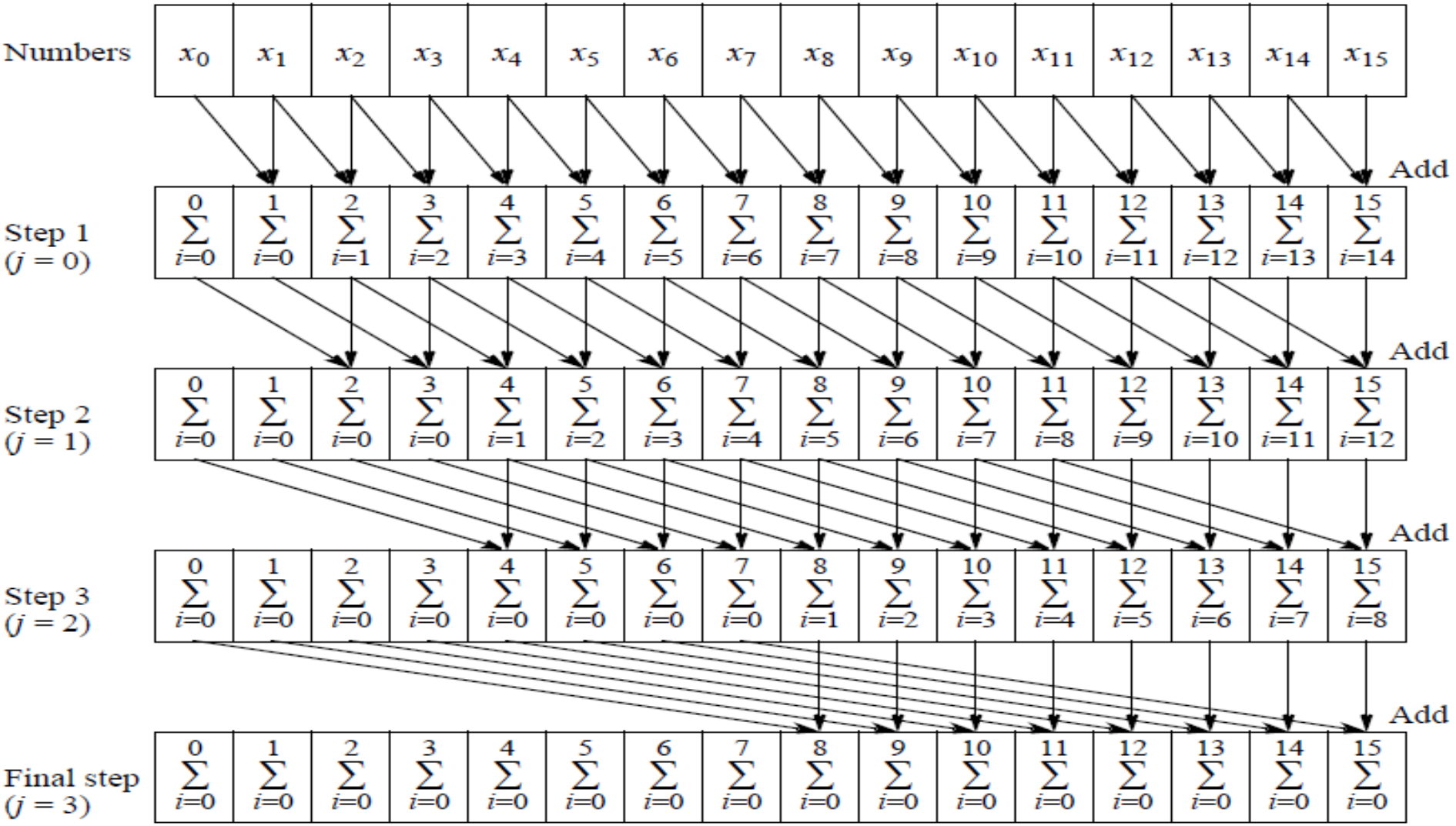


Example 1: Prefix Sum

- Given a list of numbers $x_0, x_1, \ldots . x_{n-1}$, compute all *partial* summations
  - $x_0; x_0 + x_1; x_0 + x_1 + x_2; \ldots \ldots .$
  - Could also replace operator + with AND, OR, *, etc.
- Example:
  - $x = 1,2,3,4,5$
  - Sum = 1,3,6,10,15
- Sequential code: $O(n^2)$

```
//sequential code
for(i = 0; i < n; i++) {
    sum[i] = 0;
    for (j = 0; j <= i; j++)
        sum[i] = sum[i] + x[j];
}
```

Data Parallelism Solution



Data Parallelism Code

- Sequential Code: $O(n^2)$, optimal: $O(n)$

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

- Parallel Code: $O(\log n)$

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

Synchronous Parallelism

- Each iteration composed of several processes that start together at beginning of iteration and next iteration cannot begin until all processes have finished previous iteration

- openMP

```
for (j=0; j<n; j++) {     // each iteration
    forall (i=0; i<N; i++) { // each process
        body(i);
    }
}
```

- MPI

```
for (j=0; j<n; j++) {     // each iteration
    i = myrank;
    body(i);
    barrier(mygroup);
}
```

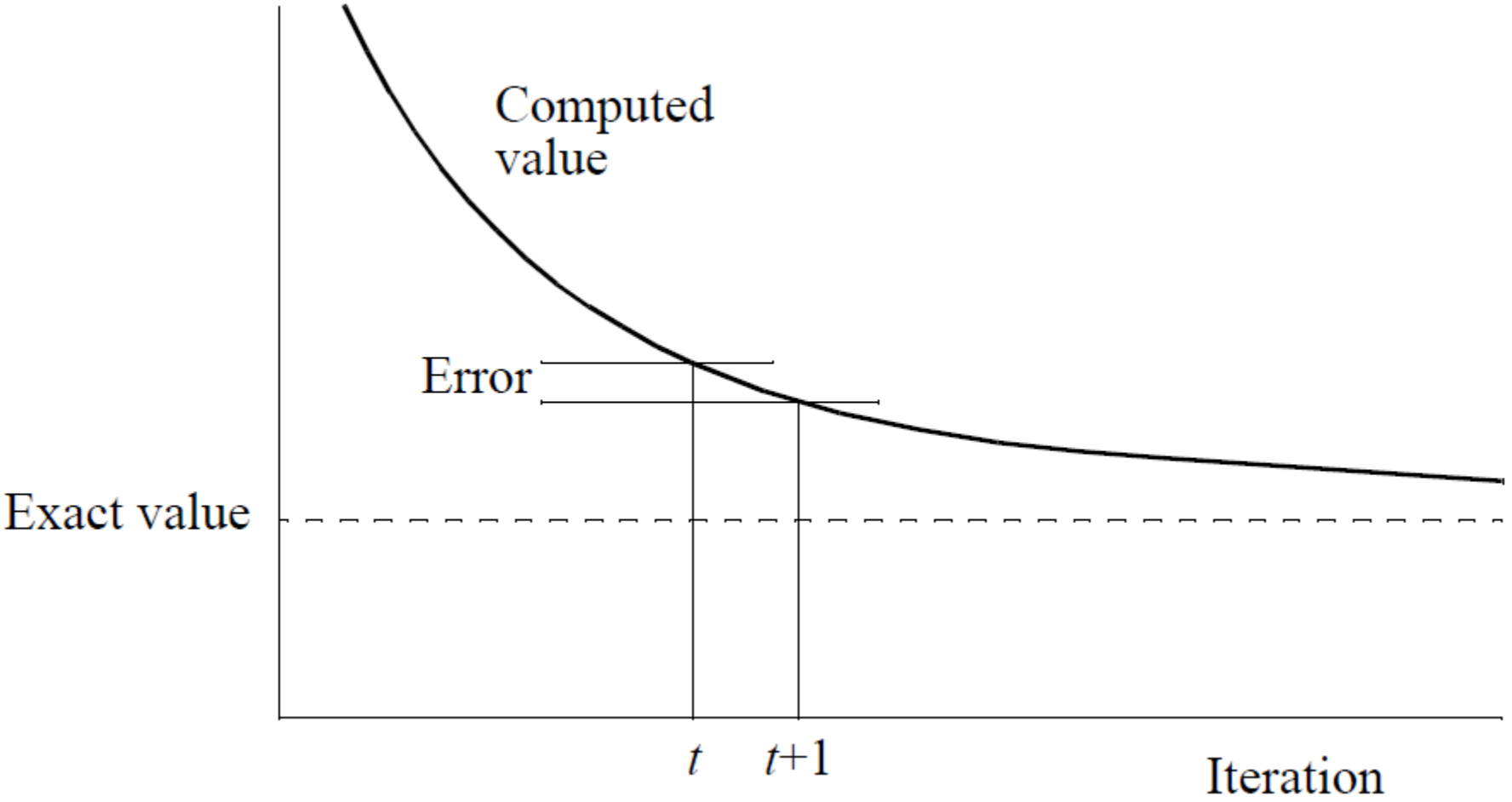
Example 2: System of Linear Equations

- System of linear equations

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots\cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots\cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-2,0} & a_{n-2,1} & a_{n-2,2} & \cdots\cdots & a_{n-2,n-1} \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots\cdots & a_{n-1,n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}$$

- Jacobi iteration algorithm:
  - Convert ***i***th iteration to $x_i = \frac{1}{a_{i,i}} [b_i - \sum_{i \neq j} a_{i,j} x_j]$
  - Initial guess with $x_i = b_i$, and calculate new x_i values
  - Repeat until $\left| x_i^{\,t} - x_i^{\,t-1} \right| <$ error tolerance

Jacobi iteration algorithm



Jacobi iteration algorithm example

$$\begin{cases} -x_0 + 2x_1 - x_2 = 2 \\ 2x_0 + x_1 - 2x_2 = 2 \\ 2x_0 - x_1 + 2x_2 = 2 \end{cases}$$

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

$$\begin{cases} x_0 = 2 - \dfrac{(2x_1 - x_2)}{-1} \\ x_1 = 2 - \dfrac{(2x_0 - 2x_2)}{1} \\ x_2 = 2 - \dfrac{(2x_0 - x_1)}{2} \end{cases}$$

- Iter1: $x_0^1 = 2,\ x_1^1 = 2,\ x_2^1 = 2$

  ➔ $x_0^2 = 2 - \frac{2x_1^1 - x_2^1}{-1} = 4,\quad x_1^2 = 2,\quad x_2^2 = 1$

  ➔ $e_0 = |2 - 4| = 2,\quad e_1 = 0,\quad e_2 = 1$

- iIter2: $x_0^2 = 2 - \frac{2x_1^2 - x_2^2}{-1} = 5,\ x_1^3 = -2,\ x_2^3 = -1$

  ➔ $e_0 = |4 - 5| = 1,\quad e_1 = 4,\quad e_2 = 2$

Jacobi iteration algorithm

- Sequential Code
  - **a[][]** and **b[]** holding constants in the equations
  - **x[]** holding unknowns
  - fixed number of iterations

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

```
for (i = 0; i < n; i++)  x[i] = b[i];          /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 0; i < n; i++) {                  /* for each unknown */
        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*/
    }
    for (i = 0; i < n; i++) x[i] = new_x[i];   /*update to new values*/
}
```

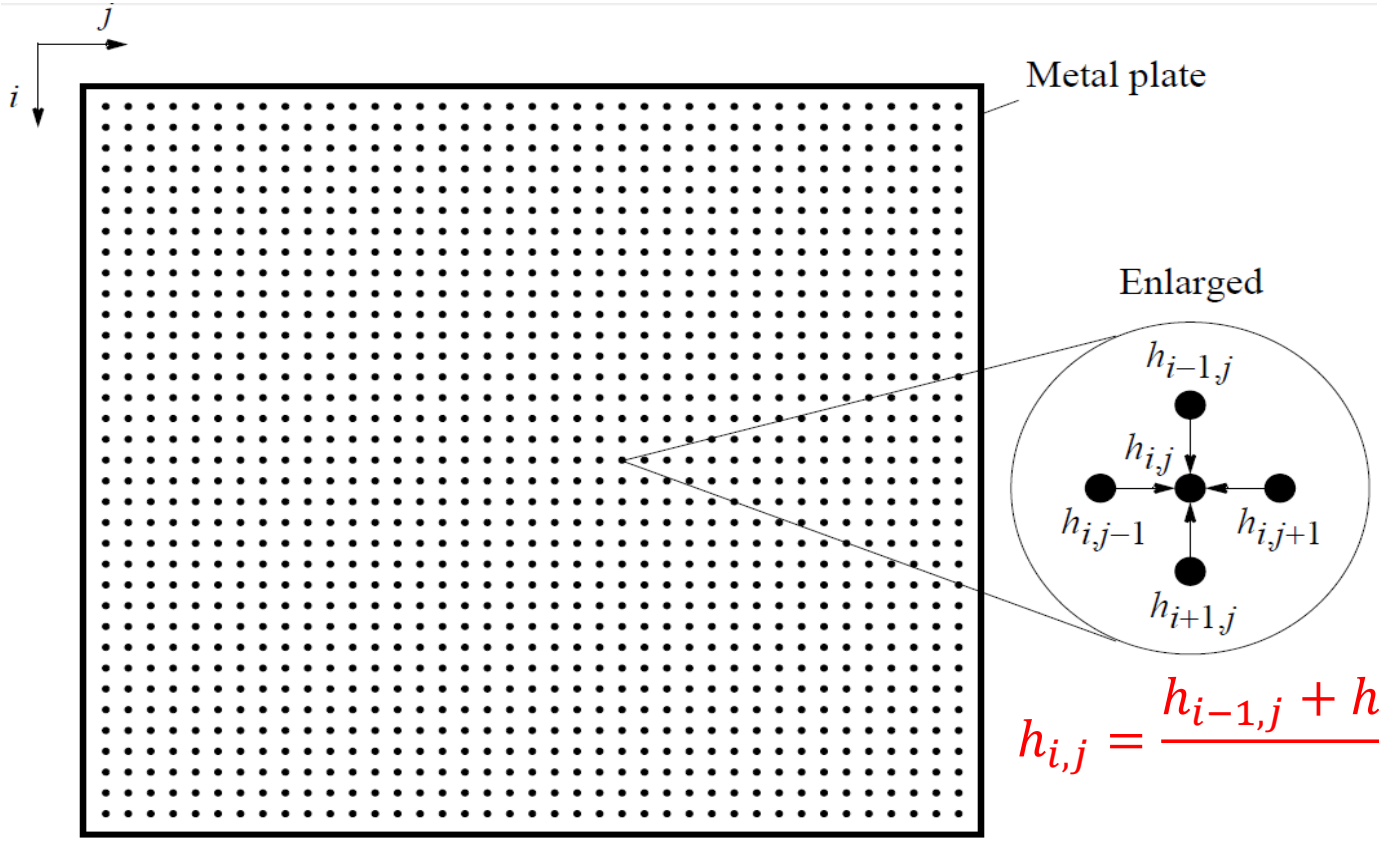
Jacobi iteration algorithm

- Parallel Code
  - Process ***i*** handles unknown ***x***[***i***]

```
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]); /* gather & broadcast new value */
    barrier();                          /* wait for all processes */
}
```

Problem3: Heat Distribution

- Consider an area as a mesh of points, $h_{i,j}$
  - Known temperatures along its edges
  - Heat is propagated until a fixed # of iterations or until temperature change is less than small pre-described amount



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

Problem3: Heat Distribution

- Sequential Code

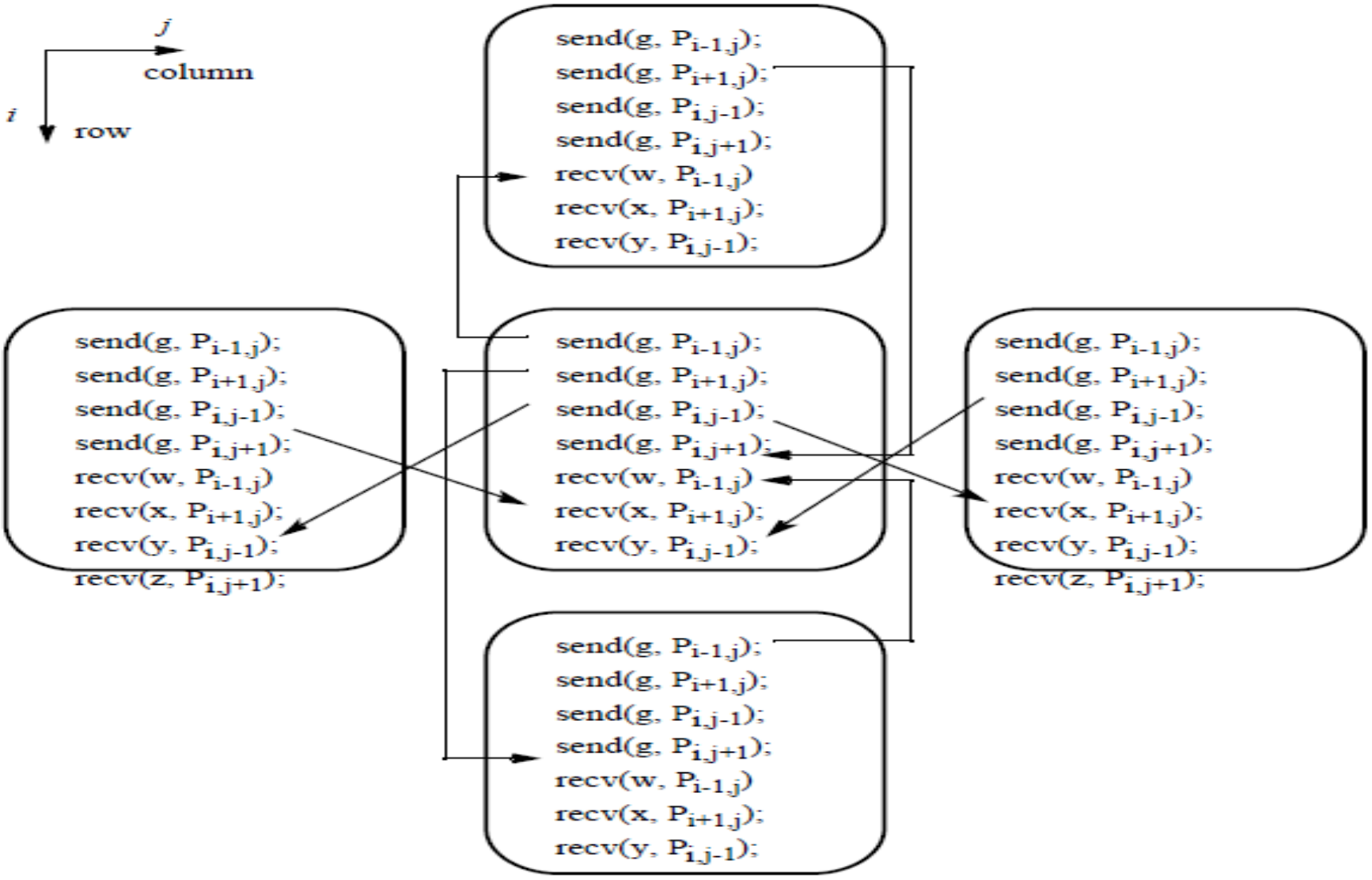
```
for (iteration = 0; iteration < limit; iteration++) {
        for (i = 1; i < n; i++)
            for (j = 1; j < n; j++)
                    g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
        for (i = 1; i < n; i++) /* update points */
            for (j = 1; j < n; j++)
        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

Problem3: Heat Distribution

- Parallel Code

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

Problem3: Heat Distribution



Partitioning

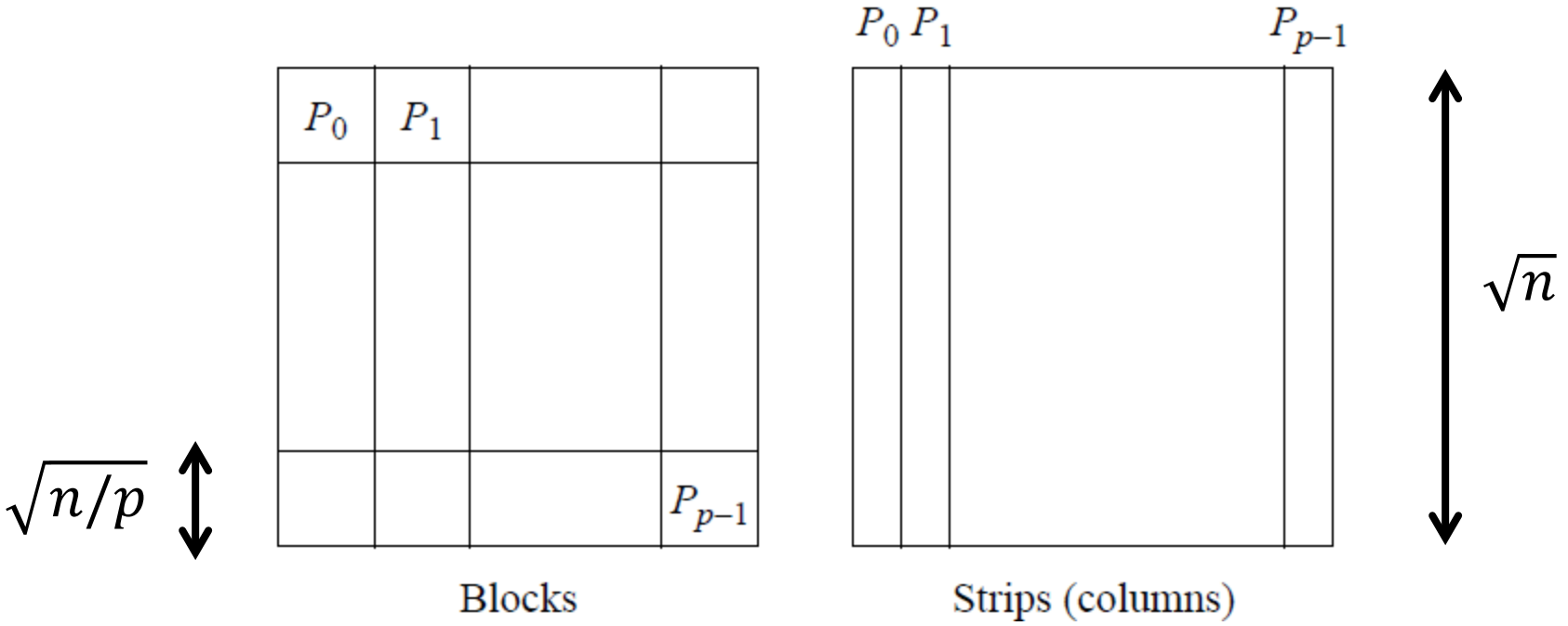
- Blocks:

Block partitioning requires lower bandwidth but higher latency

$$t_{comm} = 8(t_{startup} + \sqrt{n/p}\, t_{data})$$

- Strips:

$$t_{comm} = 4(t_{startup} + \sqrt{n}\, t_{data})$$



Partitioning

- Blocks partition has a larger communication time (**worse**) than strip partition if:

$$8\left(t_{startup} + \sqrt{n/p}\, t_{data}\right) > 4\left(t_{startup} + \sqrt{n}\, t_{data}\right)$$

$$\Rightarrow t_{startup} > \sqrt{n}(1 - 2/\sqrt{p})t_{data}$$