Parallel Algorithm Design

CS121 Parallel Computing

Fall 2021

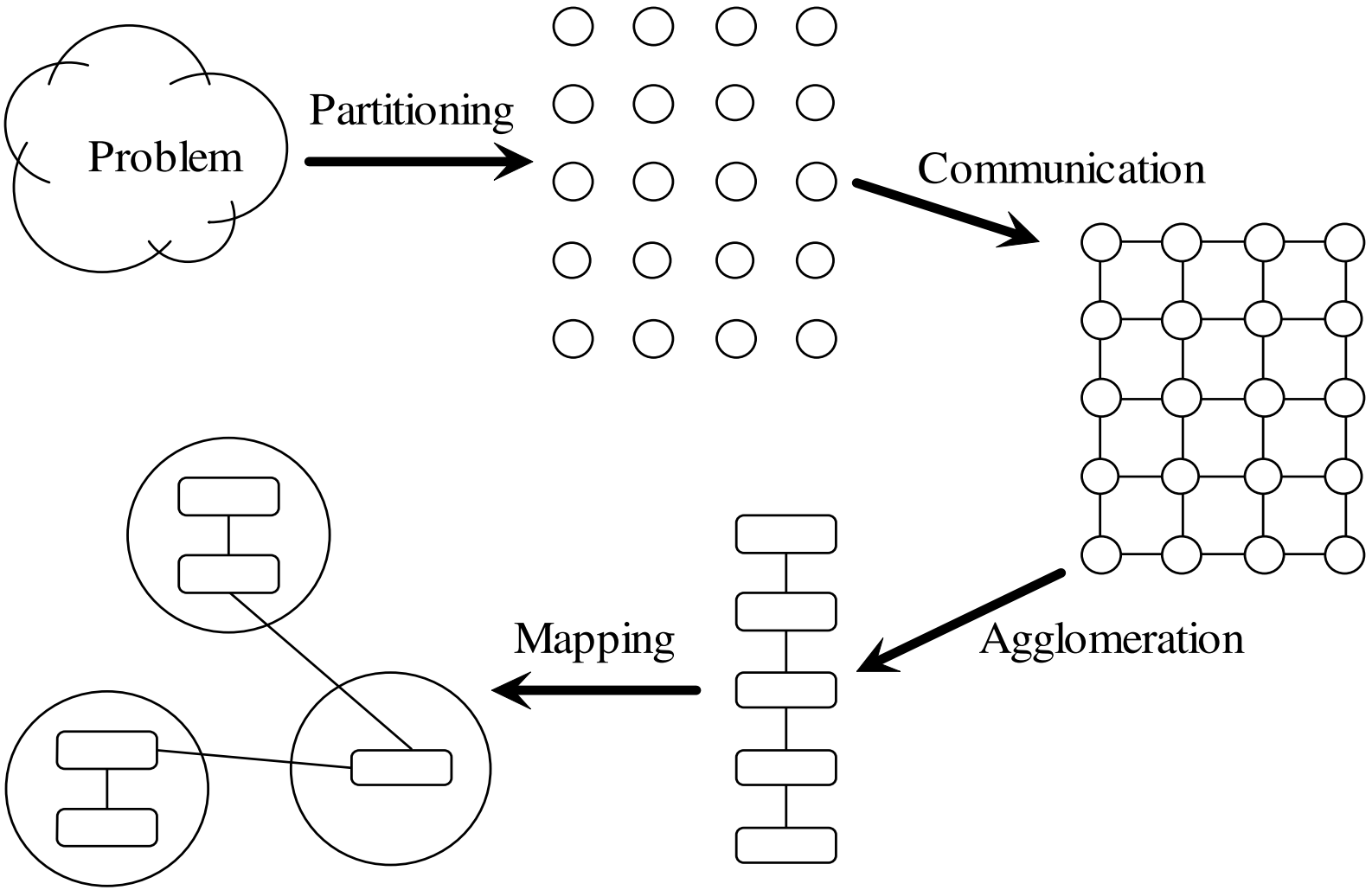
Foster's design methodology

- Encourages development of scalable algorithms by delaying machine-dependent considerations until later steps.
  - I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995, available online at http://www.mcs.anl.gov/~itf/dbpp/
- Methodology has four steps
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping

Outline

- Overview of Foster's methodology.
- Example using Floyd-Warshall algorithm.
- Data partitioning.
- Algorithmic partitioning.

Foster's design methodology

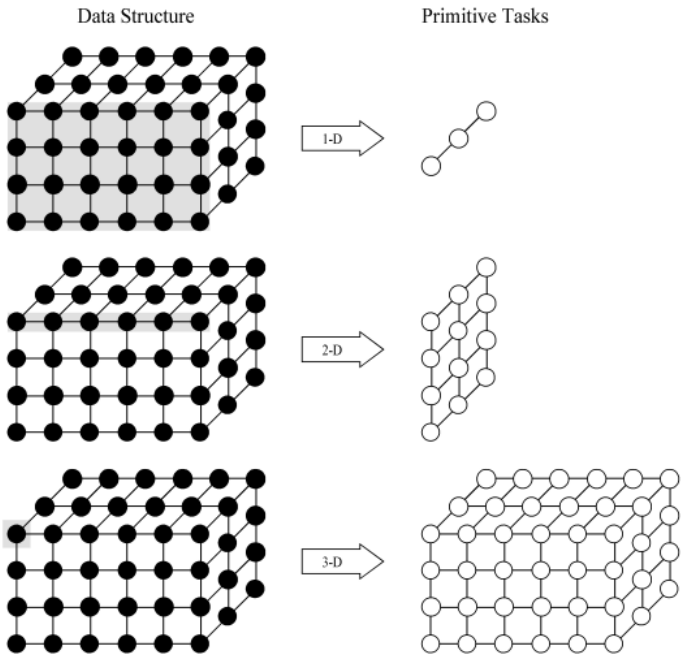


Partitioning

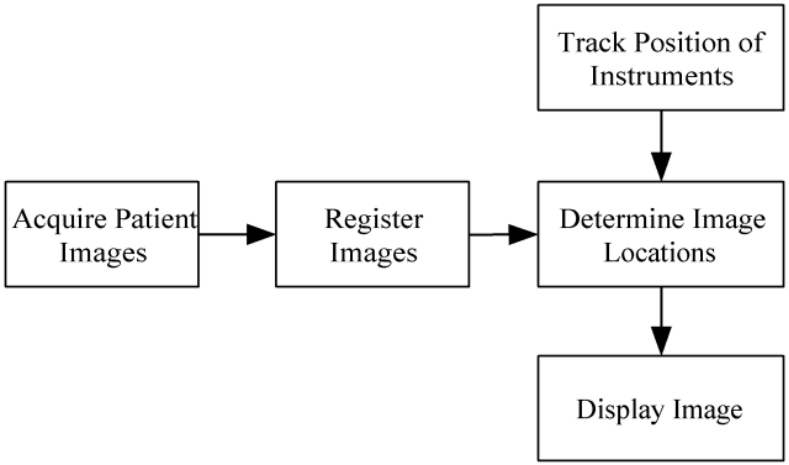
- The application must be partitioned into a number of tasks that can execute in parallel.
- Data partitioning
  - The application data is divided into parts and each task operates on a different part of the data.
  - Need to associate computations with the data.
  - Also known as domain decomposition.
- Algorithmic partitioning
  - The algorithm is divided into a number of tasks that can execute in parallel.
  - Need to associate data with the tasks.
  - This often yields tasks that can be pipelined.
  - Also known as functional decomposition.

Partitioning

- Data partitioning
  - The application data is divided into parts that can be operated on in parallel.



- Algorithmic partitioning
  - The algorithm is divided into parts that can be executed in parallel.



Partitioning checklist

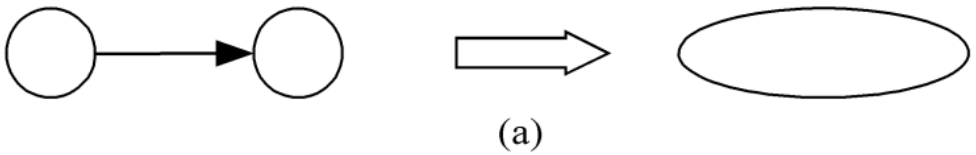
- There should be at least 10x more primitive tasks than processors in the target computer.
- Redundant computations and redundant data storage should be minimized.
- The primitive tasks should be roughly the same size.
- The number of tasks should be an increasing function of problem size.

Communication

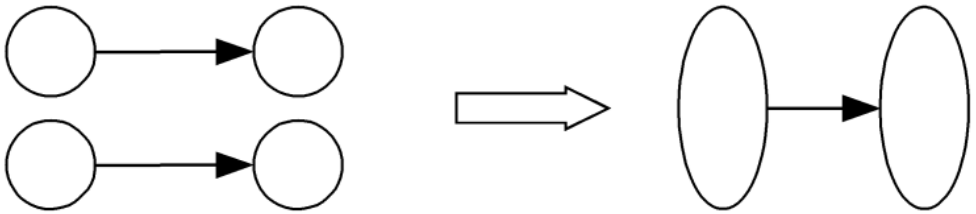
- Determine what data is passed between tasks.
- Two kinds of communication
  - Local communication: A task needs values from a small number of other tasks (point-to-point),
  - Global communication: A number of tasks require or contribute data (collective).
- Checklist
  - Communication should be balanced among tasks.
  - Each task should preferably communicate with only a small group of neighbours.
  - Tasks should be able to perform communications concurrently, i.e. no bottlenecks.
  - Task should be able to overlap computations and communication.

Agglomeration

- Grouping tasks into larger (super)tasks.
- Goal is to improve performance and simplify programming.
  - Eliminate communication between primitive tasks that are agglomerated into one task.
  - Combine groups of sending and receiving tasks so that fewer but larger messages are sent.



(a)



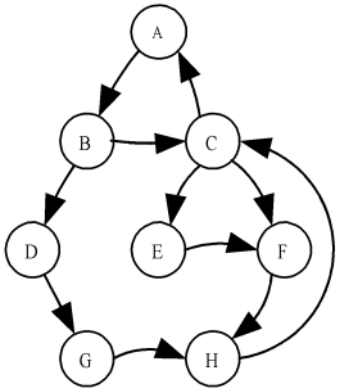
(b)

Agglomeration checklist

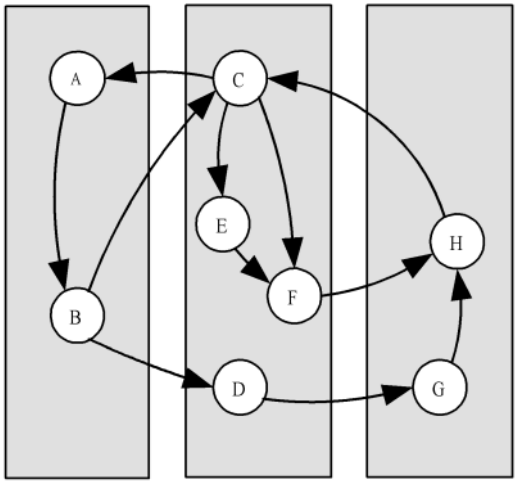
- Locality of the parallel algorithm should increase.
- Replicated computations can sometimes replace communications and often take less time.
- The amount of replicated data should be small so that it does not affect scalability.
- Agglomerated tasks should have similar computational and communication costs.
- The number of tasks should be suitable for likely target systems (at least equal to the number of processors).
- The modifications to the sequential code should be reasonable.

Mapping

- Assigning (super)tasks to processors
  - With a distributed memory system, mapping can be done by the user.
- Goals
  - Balance the load to maximize processor utilization.
  - Minimize interprocessor communication.
  - Sometimes two goals conflict. Look for best tradeoff.



(a)



(b)

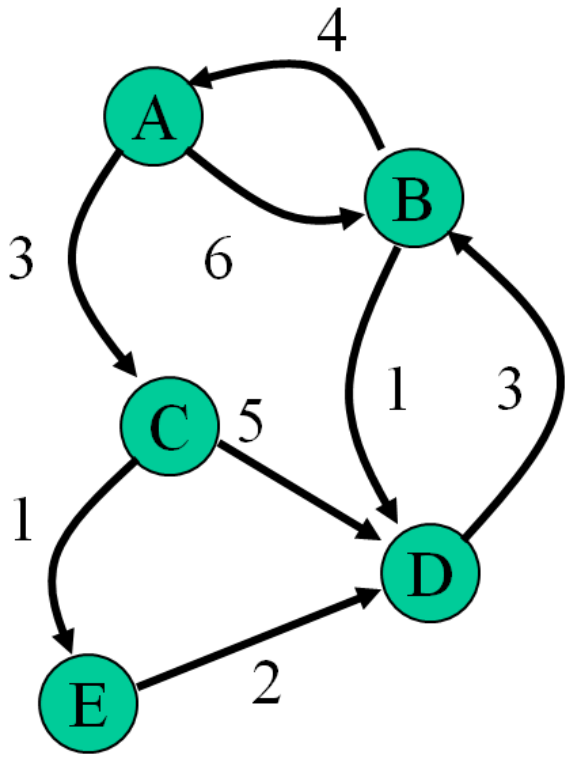
- If all tasks require the same amount of time, the above mapping would mean the middle processor takes twice as long as the other two.

Mapping checklist

- Static task allocation
  - If tasks have same computation time, can agglomerate tasks to minimize communication and create one supertask per processor.
  - If tasks have varying computation times, can cyclically map tasks to processors so each processor receives a set of tasks with same average load.
- Dynamic task allocation
  - If tasks are created dynamically, or they have unknown computation times, can allocate dynamically.
  - Ex Work stealing. Each processor has a work queue. Processors finished with own work queue take tasks from another processor's queue.
  - Should ensure the task allocator (manager) is not a bottleneck to performance.

All pairs shortest path problem

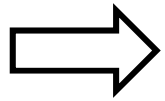
- Given a weighted directed graph of vertices and edges.
  - There may be an edge from vertex i to j, but not from j to i.
  - Edge weight from vertex i to j may be different to from j to i.
- Find the shortest path between every pair of vertices.
- Applications
  - Shortest route on a road / communication network.
  - Efficient solution in a state space graph.



Example

- Represent graph using adjacency matrix.
  - a[i,j] = 0 if i = j.
  - For i ≠ j, a[i,j] = length of edge from i to j, or ∞ if no edge.
- Output matrix of shortest path lengths, or ∞ if no path.
- Assume no negative weight cycles for simplicity.

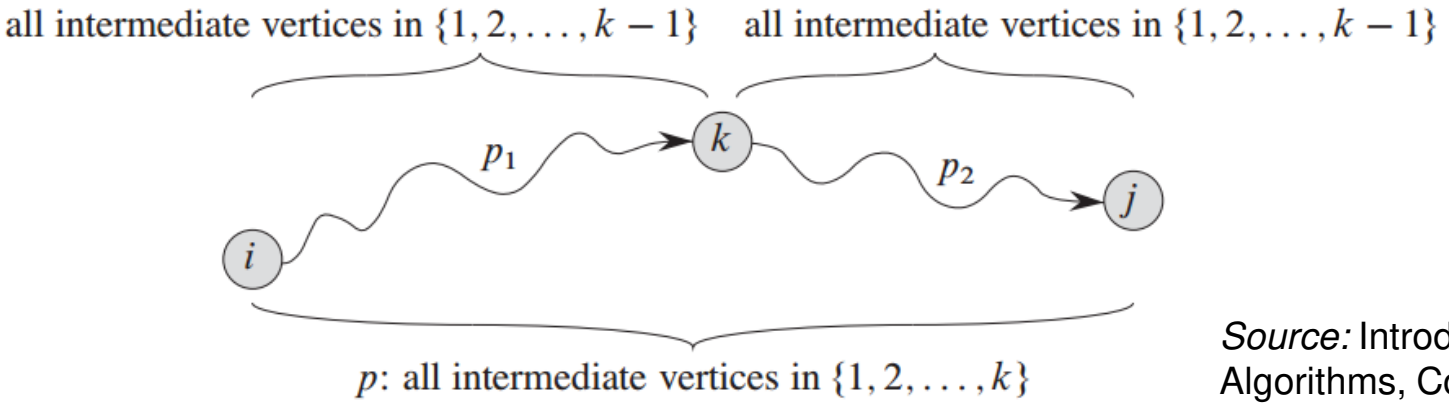
	A	B	C	D	E
A	0	6	3	∞	∞
B	4	0	∞	1	∞
C	∞	∞	0	5	1
D	∞	3	∞	0	∞
E	∞	∞	∞	2	0



	A	B	C	D	E
A	0	6	3	6	4
B	4	0	7	10	8
C	10	6	0	3	1
D	7	3	10	0	11
E	9	5	12	2	0

Floyd-Warshall algorithm

- Consider the shortest path from i to j using only nodes 1,...,k as intermediate nodes.
  - Intermediate node is a node on the path besides i and j.
  - If the path doesn't exist, indicate its distance by ∞.
- Assume a path exists. Then it either uses node k or not.
  - In the first case, k occurs once in path from i to j.
    - The path consists of a subpath from i to k using 1, ..., k-1 as intermediate nodes, and a path from k to j using 1, ..., k-1 as intermediate nodes.
  - In the second case, the path only uses nodes 1,...,k-1.



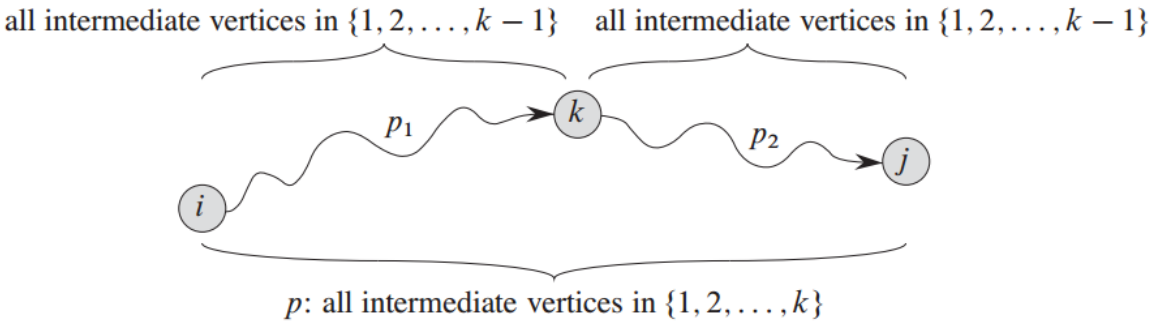
Source: Introduction to Algorithms, Cormen et al.

Floyd-Warshall algorithm

- Let $d_{i,j}^{k}$ = min distance from i to j using 1, ..., k as intermediate nodes.
- For a graph with n nodes, we compute the shortest paths using a dynamic programming formulation that increases k from 0 to n.
  - Represents more and more general paths.
  - Shortest path from i to j equals $d_{i,j}^{n}$.
  - Assume in k'th stage, have already found $d_{i,j}^{k-1}$ for all i,j. Then

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

  - Expresses observation that shortest path from i to j using 1,...,k as intermediate nodes either does not use node k, or uses it once.



Parallelization: partitioning

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases}$$

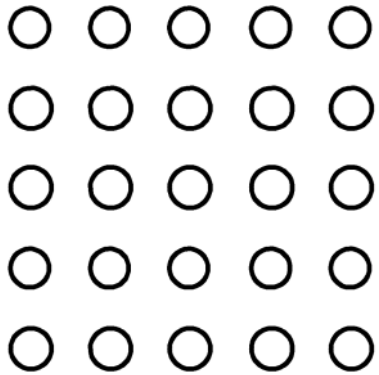
```
for (k = 0; k < n; k++)
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        a[i, j] = min(a[i, j], a[i, k] + a[k, j]);
```

- To parallelize this algorithm, observe for each k, all assignments can be done independently.
  - I.e. a[i,j] and a[i',j'] can be computed in either order, for all i,i',j,j'.
  - a[i,j] depends on a[i,k] and a[k,j].
  - But a[i,k]=min(a[i,k], a[i,k] + a[k,k]), where a[k,k]=0.
  - So a[i,k] doesn't change in iteration k. Similarly for a[j,k].
  - So no matter when a[i,j] is computed during phase k, it has same value. Same for a[i',j']. So a[i,j] and a[i',j'] are independent.
- Parallel algorithm runs n phases sequentially (one for each k). In each phase, all n^2 assignments done in parallel.

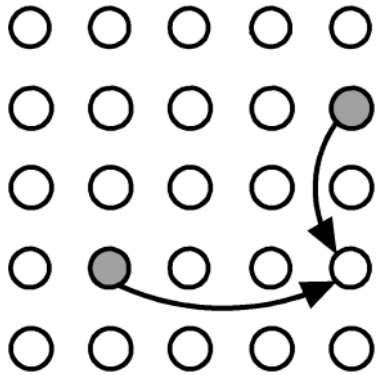
Parallelization: communication

- Communication in each phase

The primitive tasks



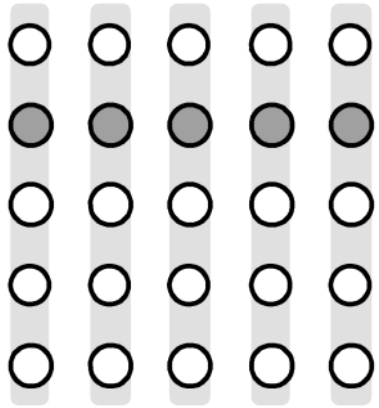
(a)



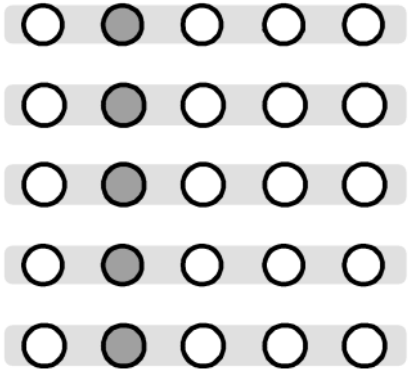
(b)

- Updating a[3,4] when k=1.
- In general, every entry in row i needs a[i,k], and every entry in column j needs a[k,j].

So in phase k, a[k,j], for each j, broadcasts its value to the j'th column.



(c)



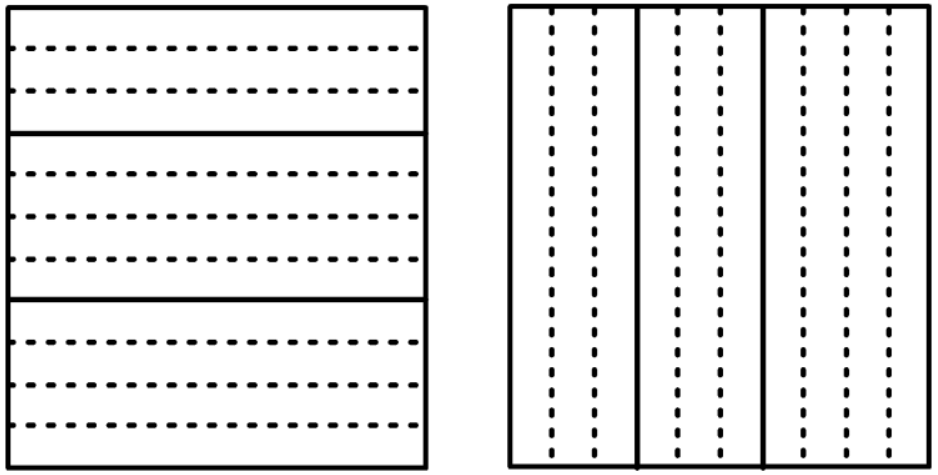
(d)

Also, a[i,k], for each i, broadcasts its value to the i'th row.

Agglomeration and mapping

- Number of tasks is static, always n^2.
- Computation time per task is constant.
- Agglomerate tasks to minimize communication.
- Create one task per MPI process.
- Can choose either (a) row striping, or (b) column striping.
  - Row striping more convenient in C.

Each process has a strip of rows, avoid communication within same strip (row).



(a) (b)

Each process has a strip of columns, avoid communication within same strip (column).

MPI pseudocode

- Assume n × n matrix on p processors (n exactly divisible by p), where part[i,j] holds strip of rows of i'th processor.
- We're ignoring code to distribute matrix and collect results.

```
s = n/p;                                    /* rows in strip */
for (k = 0; k < n; k++) {
    root = k/s;                             /* owner of row */
    if (myrank == root) {
        offset = k – myrank*s;              /* offset of row */
        for (j = 0; j < n; j++)             /* copy row */
            temp[j] = part[offset][j];
    }
    bcast(temp, n, root, P_group);          /* broadcast row */
    for (i = 0; i < s; i++)                 /* update strip */
        for (j = 0; j < n; j++)
            part[i][j] =
                min(part[i][j], part[i,k] + temp[j]); }
```

Analysis of algorithm

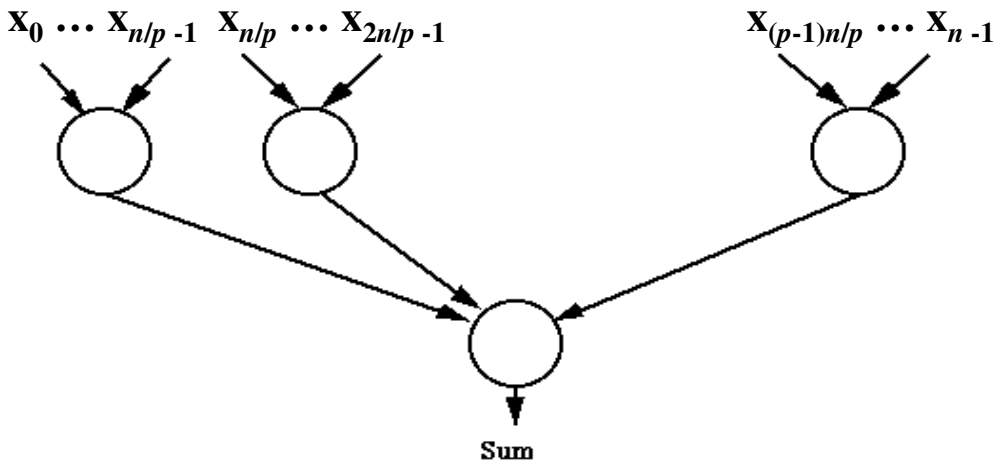
- Sequential running time $t_s = O(n^3)$.
- Parallel execution
  - In each iteration of k, there is a communication phase and a computation phase.
  - Communication phase
    - Broadcast row to other processors
    - $t_{comm} = (t_{startup} + 4\ n\ t_{data})\ \log p$, assuming each element 4 bytes.
  - Computation phase $t_{comp} = cn^2 / p$.
  - Total time
    - $t_p = n\ (t_{startup} + 4\ n\ t_{data}\ \log p + cn^2 / p\ ) = O(n^2 \log p + n^3 / p)$.
- Speedup
  - $t_s / t_p = O(n^3) / O(n^2 \log p + n^3/p) = O(p)$.

Data partitioning

- Simple data partitioning
  - Divide data into parts of fixed size based on data size and number of processors.
  - Cost includes computation cost, and communication cost for data distribution and collection.
- Divide and conquer
  - Recursively divide data into smaller and smaller parts, until a small enough size is reached.
  - Each part solved on a processor.
  - Number of parts produced not known a priori.
  - This results in a tree structure with a number of levels.
  - Better load balancing, solution quality, but higher overhead.

Sum of numbers

- To add a list of n numbers, divide it into sublists that are added in parallel by different tasks and the results are then combined.
- Initially master holds all values. It scatters values to slaves.
- Convenient to have one task (sublist) per processor.



Master P_0 and slaves P_i $(1 \le i < p)$

MPI pseudocode

- Assume for simplicity n is divisble by p.

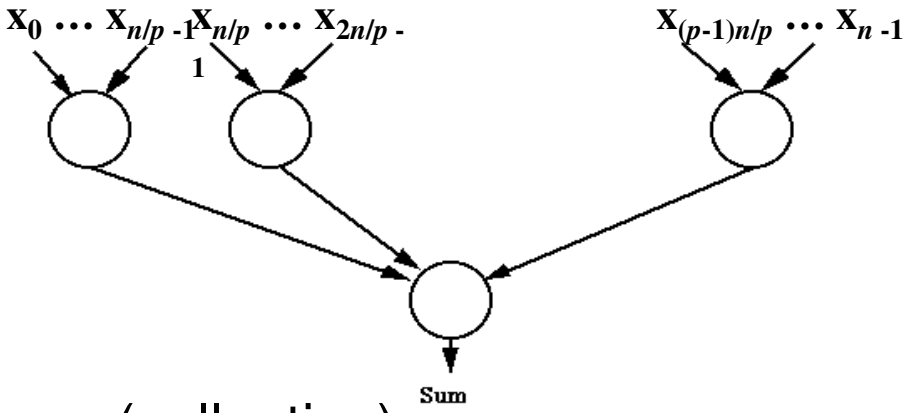
```
s = n/p;                        /* size of sublist */

/* scatter list to slaves (incl. master) */
scatter(numbers, part, s, P0, Pgroup);

res = 0;
for (j=0; j<s; j++)             /* add sublist */
      res = res + part[j];
}

/* add results from slaves (incl. master) */
reduce(res, &sum, ADD, P0, Pgroup);
```

Analysis



- Sequential execution
  - Time $t_s = n - 1$
- Parallel execution
  - Phase 1: send data to slaves (collective)
    - $t_{comm1} = t_{startup} + n\, t_{data}$
  - Phase 2: computation in slave
    - $t_{comp1} = n / p - 1$
  - Phase 3: receive results from slaves (collective)
    - $t_{comm2} = t_{startup} + p\, t_{data}$
    - Can make more efficient using reduction tree.
  - Phase 4: computation in master
    - $t_{comp2} = p - 1$
  - Total: $t_p = p + n / p - 2 + 2t_{startup} + (p + n)\, t_{data} = O(n / p) + O(p + n)$.
  - No speedup over sequential execution, due to need to transfer data in O(n) time.

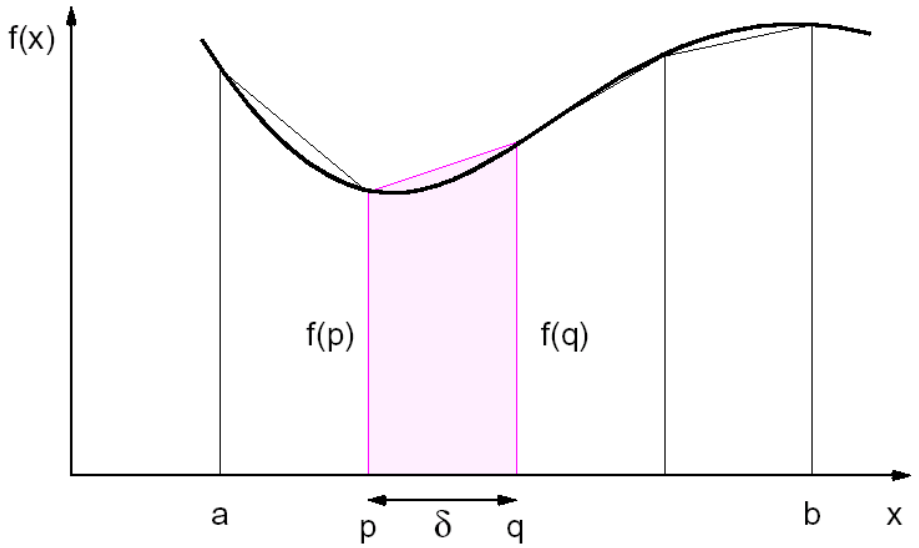
Numerical integration

- Compute area under curve given by f(x) for $x \in [a,b]$.
- Split area into small intervals of size δ and give a number of consecutive intervals to each processor.
  - There are $n=(b-a)/\delta$ intervals.
  - If there are p processors, give n/p intervals to each processor.
- Compute area of each interval using trapezoid approximation.
  - Smaller δ leads to higher n and better approximation.

MPI pseudocode

```
s = (b - a) / p;                    /* size of region */
d = (b - a) / n;                    /* size of interval */
low = a + s*myrank;                 /* my region low */
high = low + s;                     /* my region high */
area = 0.0;
for (x=low; x<high; x = x + d)
    area = area + 0.5*(f(x) + f(x+d))*d;
/* add partial results from slaves */
reduce(area, &integral, ADD, P0, Pgroup);
```

Only need to transfer final value from each processor.



Analysis

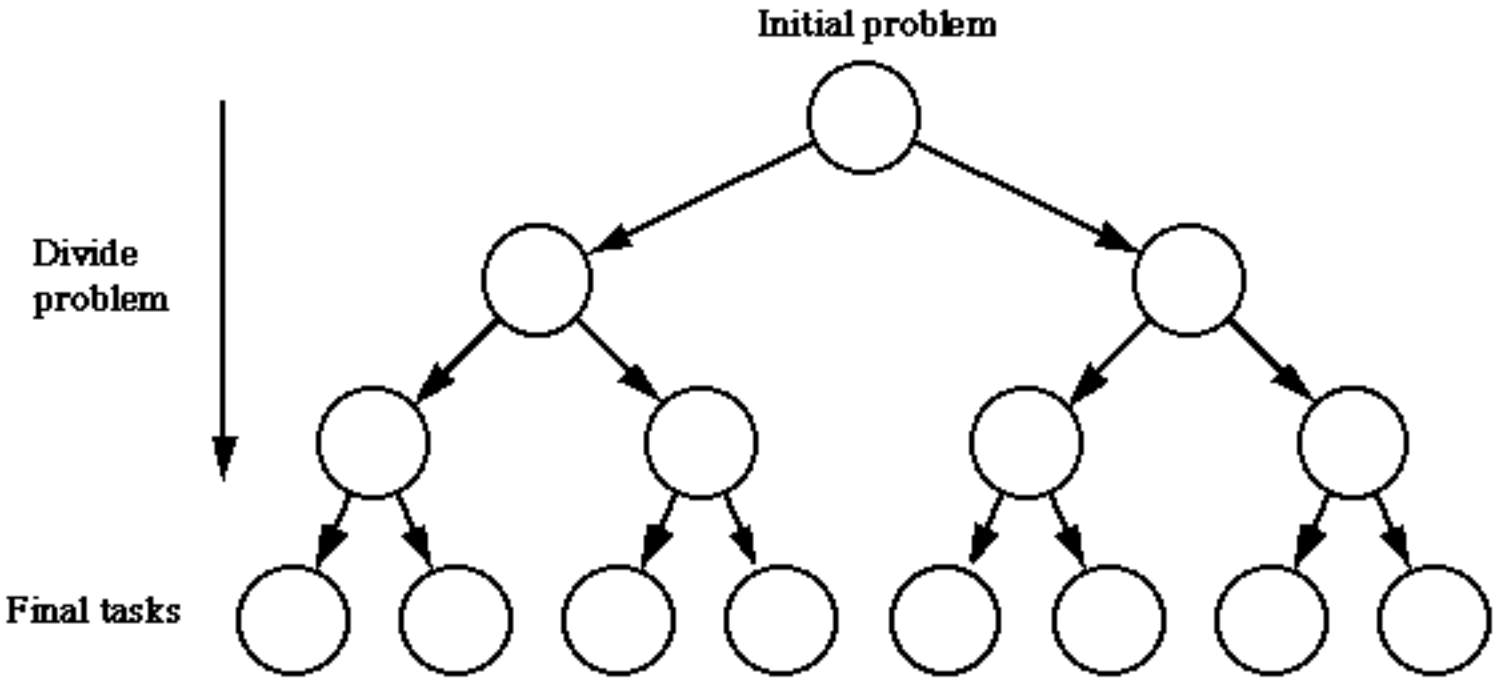
- Sequential execution
  - Time $t_s = c*n$, where c is time for one interval.
- Parallel execution
  - Phase 1: computation in slave
    - $t_{comp1} = c*n / p$
  - Phase 2: receive results from slaves (reduce)
    - $t_{comm} = t_{startup} + p\ t_{data}$
  - Phase 3: computation in master
    - $t_{comp2} = p - 1$
  - Total: $t_p = c*n / p + p - 1 + t_{startup} + p\ t_{data} = O(p + n / p)$.
  - O(p) speedup over sequential, because only needed O(p)time for communication.

Divide and conquer

- Subdivide a problem into smaller parts until parts are small enough to solve directly.

```
f(problem)
{
    if (size(problem) > k) {          /* threshold k */
        divide(problem, part0, part1);
        res0 = f(part0);              /* recursive call */
        res1 = f(part1);              /* recursive call */
        res = combine(res0, res1)     /* combine */
    }
    else                  /* solve directly if small enough */
        res = solve(problem)
    return (res);
}
```

Tree representation



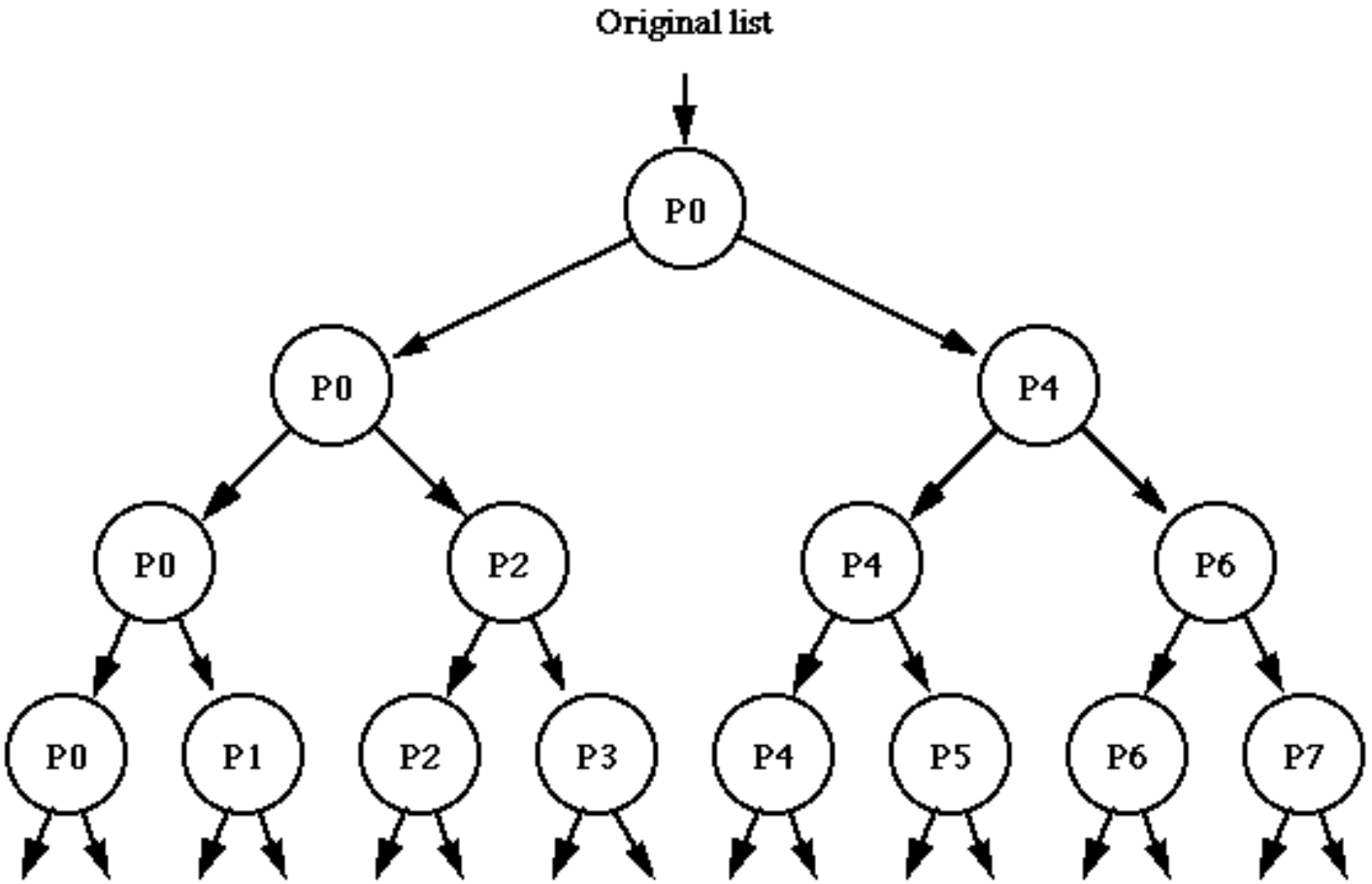
Sum of numbers

```
int add(int *numbers, n)
{
    if (n > 2) {
       s = n/2;
       res0 = add(numbers, s);          /* add 1st part */
       res1 = add(&numbers[s], n-s);    /* add 2nd part */
       res = res0 + res1;
     }
    else if (n == 2)
       res = numbers[0] + numbers[1];
     else                                    /* n == 1 */
       res = numbers[0];
     return (res);
}
```

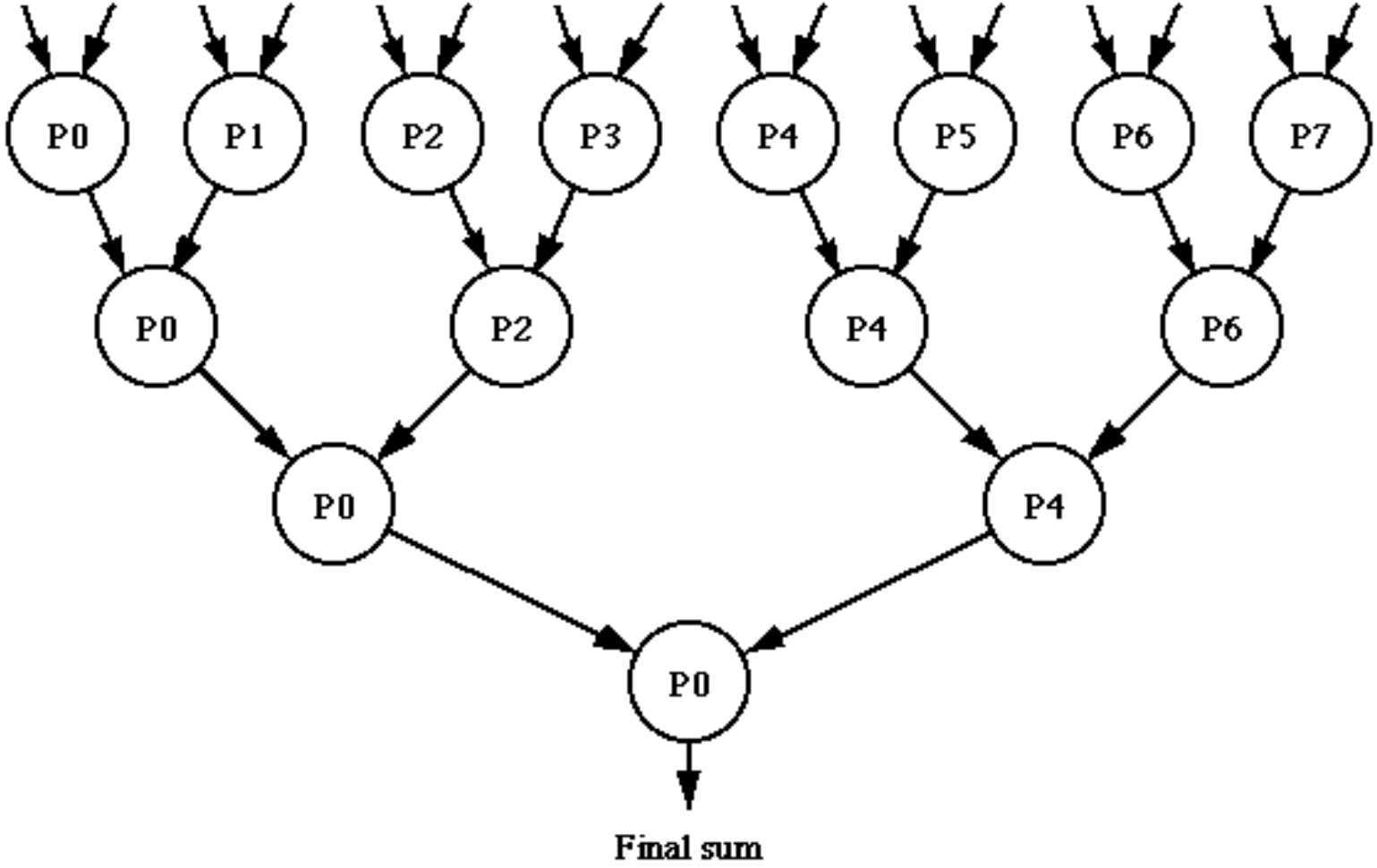
Parallel divide and conquer

- Subdivide problem into smaller parts until each part small enough to solve on separate processor.
  - Divide phase is executed in parallel.
  - Final small tasks are executed in parallel.
  - Combine phase is executed in parallel.
  - Still uses a recursive formulation.
- Assignment
  - Each node in the tree structure can be assigned to a separate processor.
  - Better to reuse processors at different levels of the tree.

Divide phase

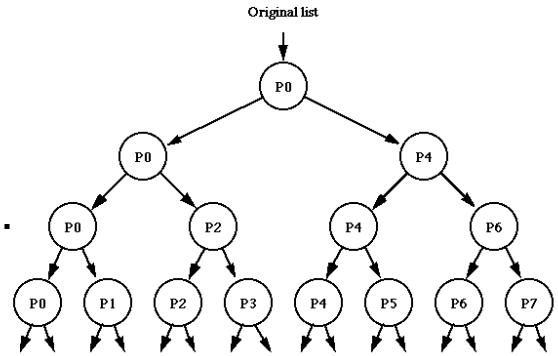


Combine phase



Parallel sum of numbers

- Sum n numbers by divide and conquer with p processors.
  - Recurse until adding at most 2 numbers, i.e. need $n/2 \leq p \leq n$.




```
int main(int *numbers) {
     /* turn off rightmost set bit in
myrank */
     recv_from = myrank & (myrank - 1);
     /* isolate rightmost set bit */
     recv_count = myrank & (-myrank);
     if (myrank == 0)
          res = add(numbers, n, myrank);
     else {
          recv(numbers, recv_count,
recv_from);
          res = add(numbers, recv_count,
myrank);
          send(res, 1, recv_from);
     }
     return res;
}
```

```
int add(int *numbers, int count, int rank)
{
     if (n > 2) {
          s = count / 2;
          /* send half numbers to slave */
          send(&numbers[s], count-s,
rank+s);
          /* calculate first part myself */
          res0 = add(numbers, s, rank);
          /* get result from slave */
          recv(&res1, 1, rank+s);
          res = res0 + res1;
     }
     else if (n == 2)
          res = numbers[0] + numbers[1];
     else       /* n == 1 */
          res = numbers[0];
     return (res);
}
```

Sum of numbers

- If p < n/2, we can assign a sublist of n/p numbers to each processor.

```
/* list of size n on p processors */
int add(int *numbers, n, p)
{
    if ((n > 2) && (p > 1)) {
        / * continue to divide */
        … …
    }
    else {
        / * add numbers sequentially */
        … …
    }
    return (res);
}
```

Analysis

- Assume for simplicity both n and p are powers of 2.
- Parallel execution
  - Phase 1: Divide (log p steps, message size halves each step)
    - $t_{comp1} = \log p$
    - $t_{comm1} = (t_{startup} + (n / 2)\, t_{data}) + (t_{startup} + (n / 4)\, t_{data}) + \ldots + (t_{startup} + (n / p)\, t_{data})$

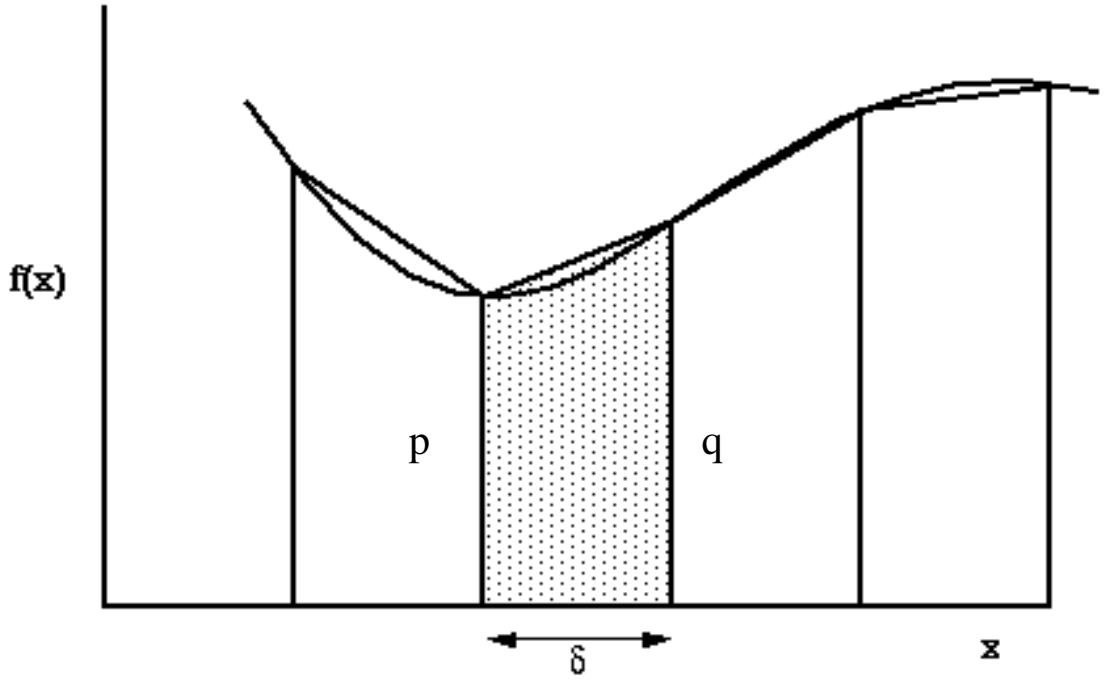
      t_{data} terms form geometric series $n\,(1/2 + 1/4 + 1/8 + \ldots)$
    - $t_{comm1} = (\log p)\, t_{startup} + (n\,(p - 1) / p)\, t_{data})$

Analysis

- Parallel execution
  - Phase 2: computation of final tasks
    - $t_{comp2} = n / p$
  - Phase 3: Combine (log p steps, but message size 1)
    - $t_{comm3} = (\log p)\ (t_{startup} + t_{data})$
    - $t_{comp3} = \log p$
  - Total: $t_p = n / p + 2 \log p + 2\ (\log p)\ t_{startup} + (n\ (p - 1) / p + \log p)\ t_{data} = O(n) + O(\log p) + O(n / p)$
- For speedup, t_{data} must be small compared to computation.

Numerical integration

- Using a fixed size interval may not give accurate results as different intervals may give different errors.

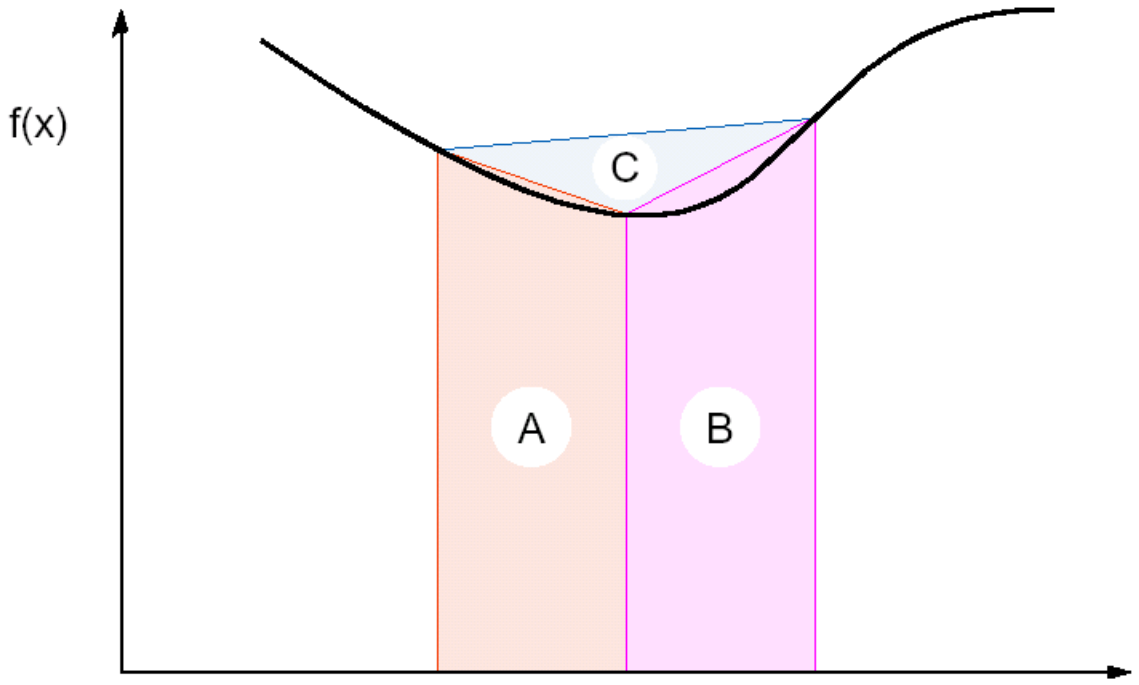


Adaptive quadrature

- Use divide and conquer to improve accuracy.
- Divide region into two intervals.
  - Continue to subdivide each interval in two until error is small enough.
- Some regions divided more finely than others.
  - Algorithm adapts to shape of curve.
  - Can represent subdivisions using a tree.
  - But tree is not a complete binary tree.
  - Assigning regions to processors is a load balancing problem. More on this in later lecture.

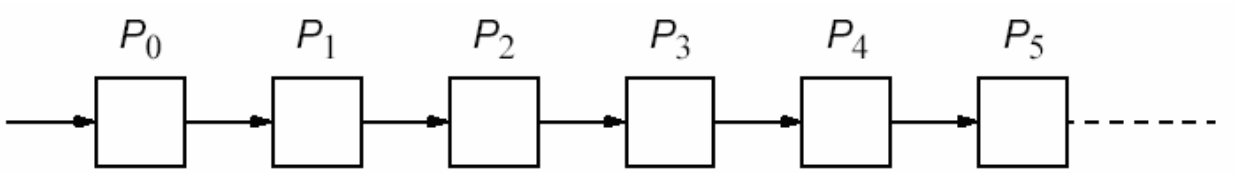
Adaptive quadrature

- Terminate when two successive approximations are close enough.
  - C is sufficiently small.
  - Or equivalently, A+B is approximately equal to area of large trapezoid.

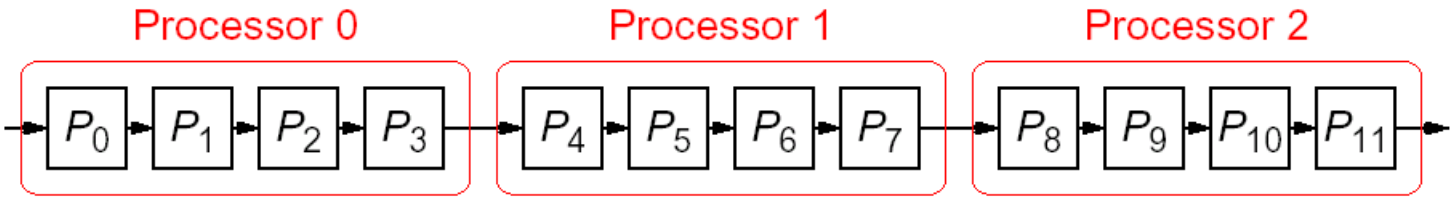


Algorithmic partitioning

- Divide an algorithm into several parts, each taking turns processing the data.
- One example is pipelined computations.
  - The problem is divided into a sequence of steps.
  - The steps must be executed one after another.
  - Each step is executed by a separate task.



  - If number of stages in pipeline is greater than number of processors, assign group of steps to each processor.



Example: sum of sequence

- Power series for sin $\theta = \theta - \theta^3/3! + \theta^5/5! - \theta^7/7! + \ldots$

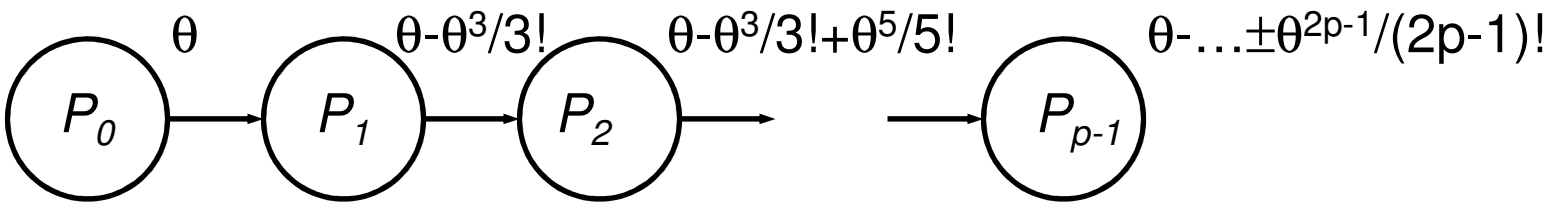
```
r = theta;
sum = theta;
for (i=1; i < p; i++) {
      r = – r*theta*theta/(2*i*(2*i+1));
      sum = sum + r;
}
```

- The loop could be unfolded to give.

```
sum = theta;
sum = sum - theta^3/3!;
sum = sum + theta^5/5!;
sum = sum - theta^7/7!;
…
```

Pipeline for unfolded loop

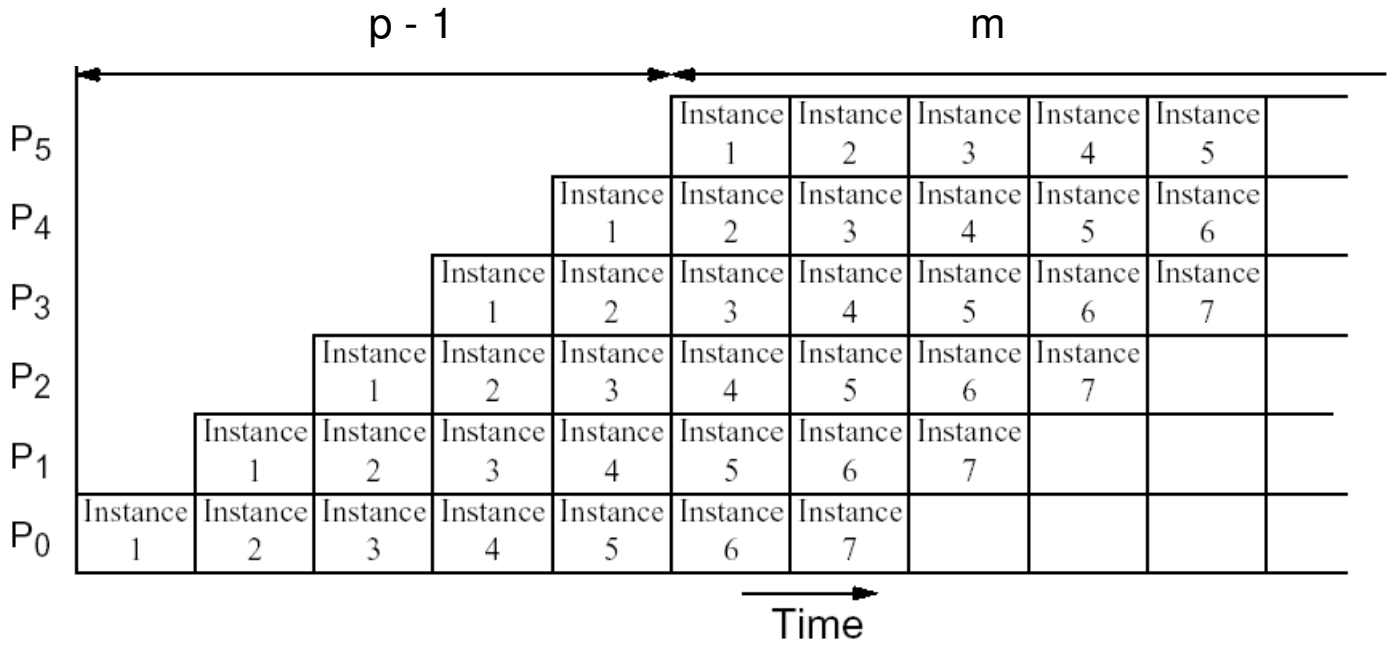
- Use as many stages (tasks) as required by the precision of the result.
- Can pass θ, partial product and partial sum to next processor in pipeline.
  - More details in later slide.



Types of pipelines

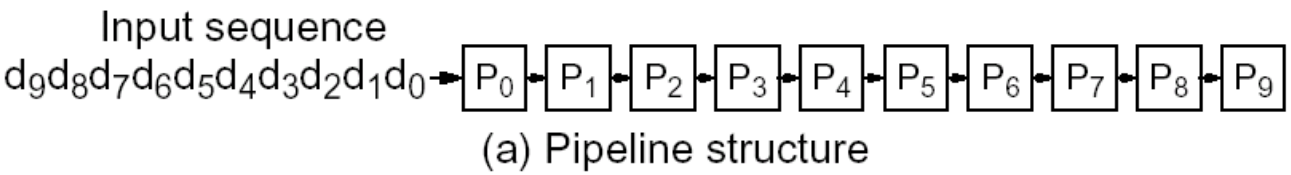
- Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations.
  1. If more than one instance of the problem is to be executed.
  2. If a series of data items must be processed, each requiring 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: Space-time diagram

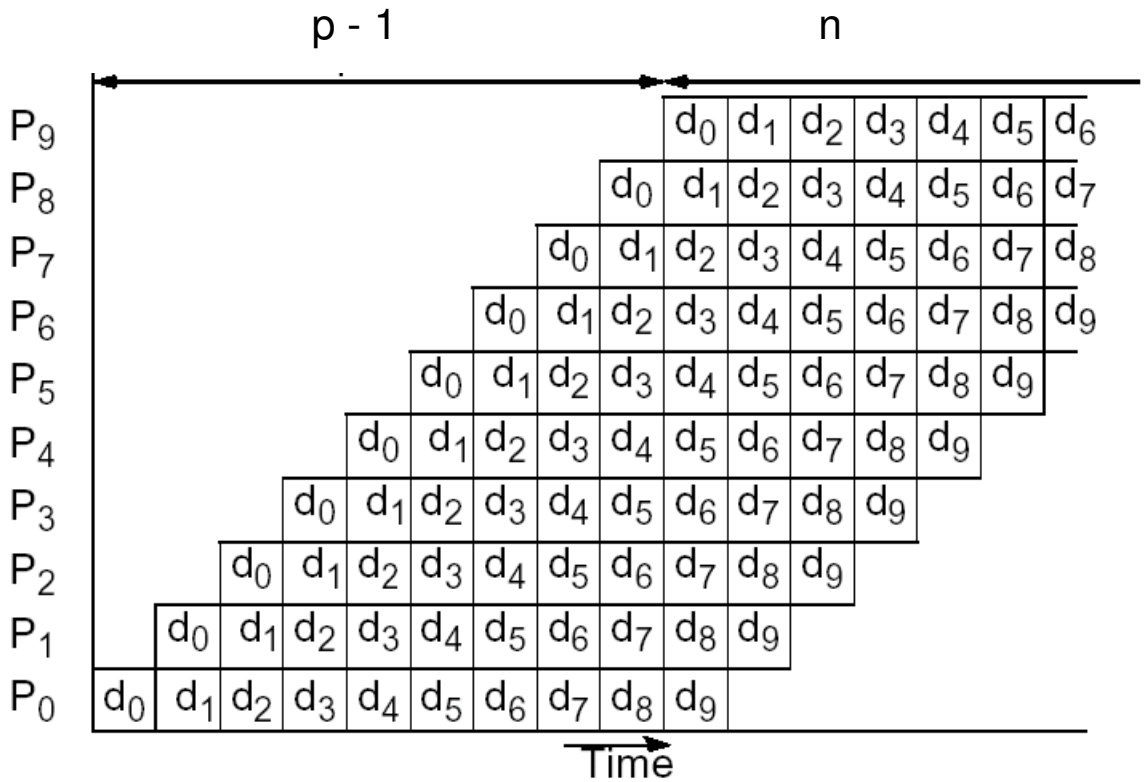


- p-stage pipeline on p processors.
- m instances of the problem.
- Execution time = m + p – 1.

Type 2: Space-time diagram



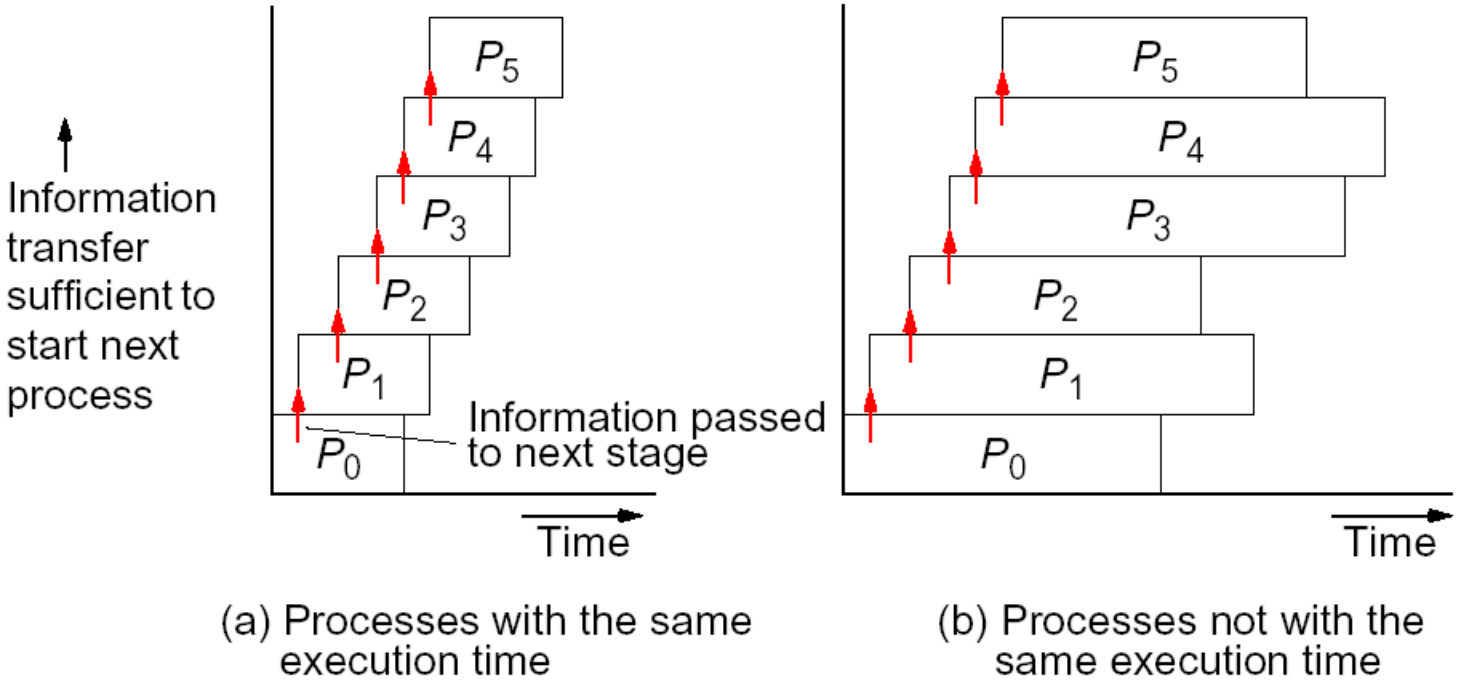
(a) Pipeline structure



(b) Timing diagram

- p-stage pipeline on p processors
- n data items
- Execution time = n + p - 1

Type 3: Space-time diagram



(a) Processes with the same execution time

(b) Processes not with the same execution time

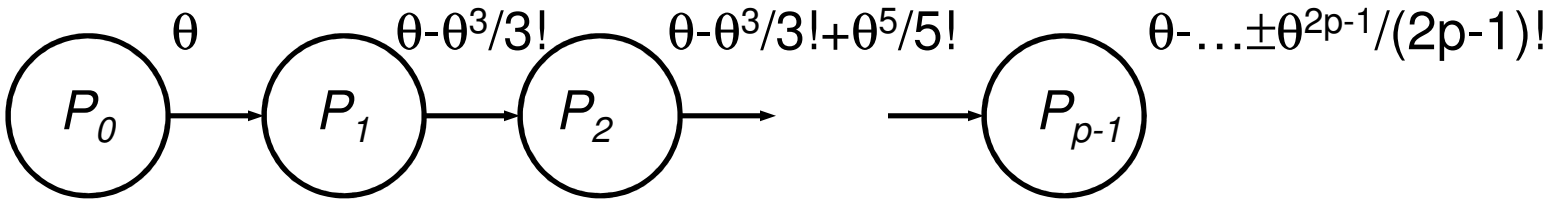
Pipeline processing where information passes to next stage before end of process.

Analysis of types 1 and 2

- Assumptions and model
  - p processors performing p stages (tasks).
  - Each task consists of a number of cycles, i.e. basic units of computation.
  - All cycles of all tasks take about same amount of time, say t_{cycle}.
- Parallel execution time $t_p = t_{cycle} * n_c$.
  - Time for one cycle $t_{cycle} = t_{comp} + t_{comm}$.
  - Number of cycles n_c given by space-time diagram.
    - For type 1, $n_c = m + p - 1$ for m instances.
    - For type 2, $n_c = n + p - 1$ for n data items.

Sum of sequence (type 1)

- Power series $\sin\theta = \theta - \theta^3/3! + \theta^5/5! - \theta^7/7! + \ldots$




```
if (myrank == 0)                          /* first */
      send(theta, theta, theta, P1);
else                                      /* not first */
      recv(&theta, &r, &sum, Pi-1);
      r = – r*theta*theta/(2*i*(2*i+1));
      sum = sum + r;
      if (myrank < p-1 )                  /* not last */
            send(theta, r, sum, Pi+1);
}
/* final result in Pp-1 */
```

Analysis

- Parallel execution
  - Use most complex cycle to upper bound computation time.

```
recv(&theta, &r, &sum, P_{i-1});
r = - r*theta*theta/(2*i*(2*i+1));
sum = sum + r;
send(theta, r, sum, P_{i+1});
```

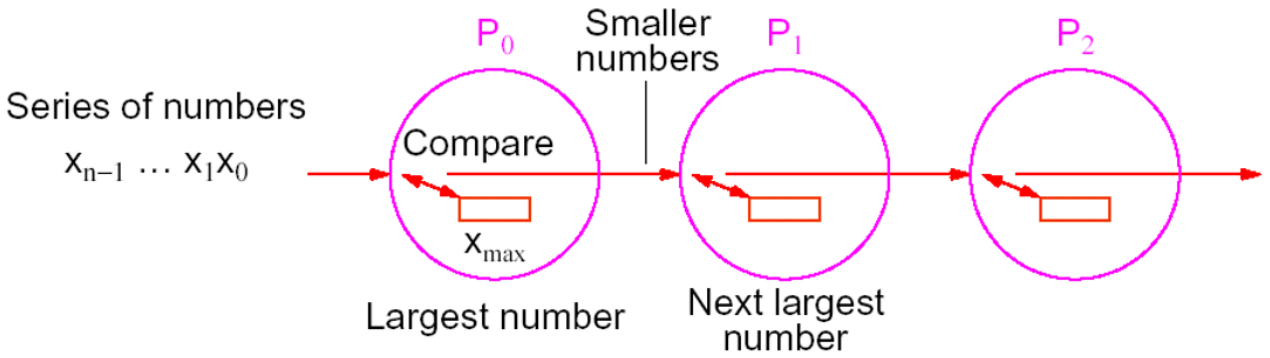
  - Computation time $t_{comp} = 4$.
  - Communication time $t_{comm} = 2(t_{startup} + 3\ t_{data})$.
  - Time for one cycle $t_{cycle} = t_{comp} + t_{comm}$.
- Single instance of problem
  - Number of cycles $n_c = p$ for single instance.
  - Time $t_p = t_{cycle} * n_c = p(2(t_{startup} + 3\ t_{data}) + 4)$.

Analysis

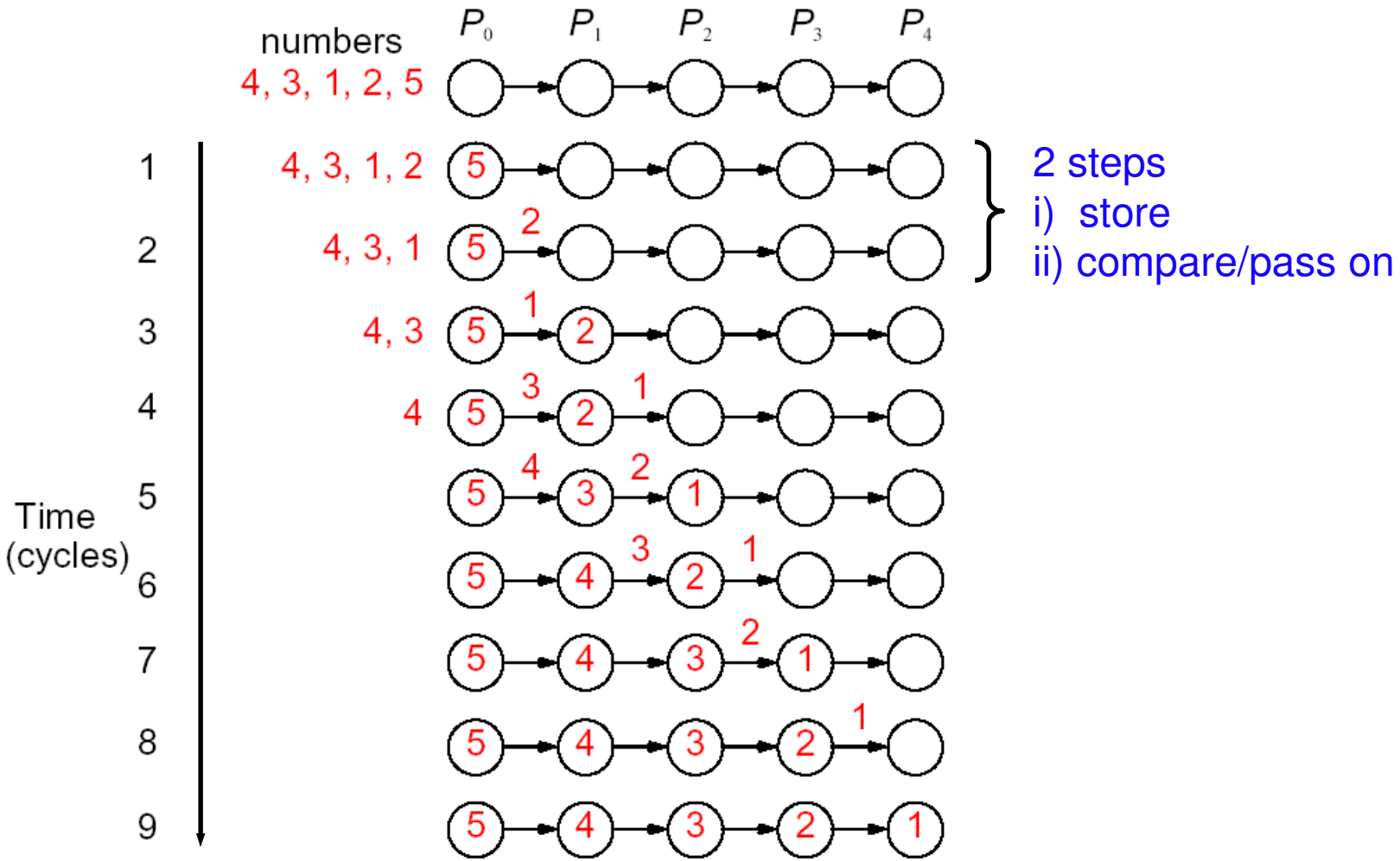
- Pipeline is efficient only if we have more than one problem to solve, i.e. more than one value of θ.
  - Assume we compute sin for m values of θ.
- Parallel execution
  - Number of cycles $n_c = m + p - 1$ for m instances.
  - Time $t_p = t_{cycle} * n_c = (2(t_{startup} + 3\ t_{data}) + 4) * (m + p - 1)$.
  - Average execution time per instance is $t_a = t_p / m = O((m + p) / m)$.
  - For large m (i.e. continuous supply of values of θ) average execution time per instance is
    - $t_a = 2(t_{startup} + 3t_{data}) + 4 = O(1)$.
    - A result is produced each constant number of pipeline cycles.

Insertion sort (type 2)

- Use a pipeline to sort n numbers in decreasing order using p = n processors.
- Each processor stores the largest number received so far and passes on all smaller numbers.
- If a new number is larger than the currently stored number, the stored number is passed on and replaced by the new number
- When all numbers have been passed through the pipeline, P_0 has the largest number, P_1 has next largest number, etc.



Insertion sort



Insertion sort

- Initialization and basic cycle for P_i.
  - x is current max value at P_i.

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

- Process P_i needs to pass on (n - i - 1) numbers, as there are (n - i - 1) processors to its right.
  - It executes the basic cycle (n - i - 1) times.

Analysis

- Sequential execution
  - $t_s = (n-1) + (n-2) + \ldots + 2 + 1 = n\,(n-1) / 2 = O(n^2)$.
- Parallel execution
  - Each basic cycle consists of one recv, one send and a compare/exchange operation.
  - Computation time $t_{comp} = 3$.
  - Communication time $t_{comm} = 2(t_{startup} + t_{data})$.
  - Time for one cycle $t_{cycle} = t_{comp} + t_{comm}$
  - Number of cycles $n_c = n + p - 1 = 2n - 1$.
  - Time $t_p = t_{cycle} * n_c = (2(t_{startup} + t_{data}) + 3) * (2n - 1) = O(n)$.

Solving linear equations (type 3)

- Solving linear equations in upper (or lower) triangular form.
  - Used in Gaussian elimination.
- Process P_i behaves as follows.
  - Receives partial solution from P_{i-1}.
  - Passes on partial solution to P_{i+1}.
  - Computes next part of solution and sends to P_{i+1}.
- This is a type 3 pipeline computation.
  - Each process has a different computation time.

Solving linear equations

Upper-triangular form

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \ldots \quad + a_{n-1,n-1}x_{n-1} &= b_{n-1} \\
&\vdots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 &= b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 &= b_1 \\
a_{0,0}x_0 &= b_0
\end{aligned}
$$

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.,

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

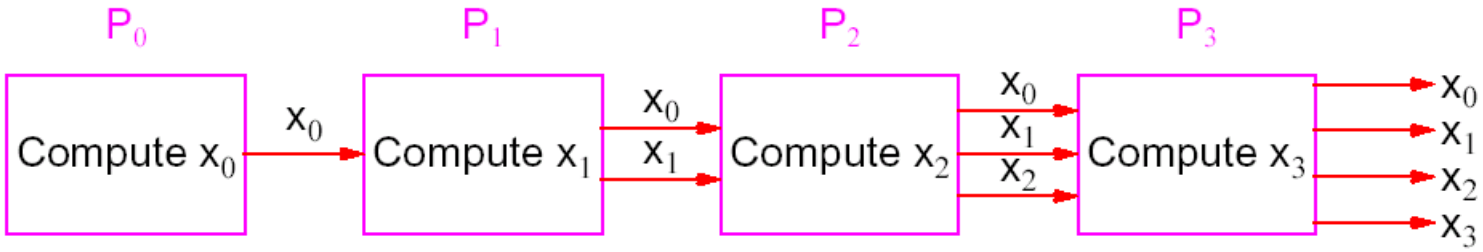
Values obtained for x_1 and x_0 substituted into next equation to obtain x_2:

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

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.



p = n
n equations
n unknowns

Type 3 pipeline computation

Parallel code

- Process P_i ($0 < i < n$)

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

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

- Process P_0

```
x[0] = b[0]/a[0][0];
send(x[i], P_{i+1});
```

- p = n stages, with final result on P_n.

Analysis

- Sequential execution
  - Iteration i of loop performs i multiplications and additions, one subtraction and one division.
  - $t_s = O(1+2+...+n-1) = O(n^2)$.
- Parallel execution
  - Process P_i performs i multiplications and additions, one subtraction and one division.
  - Process P_i has one recv and one send before process P_{i+1} can start, so time to pass data is $2(t_{startup} + t_{data})$.
  - Process P_{n-1} starts at time $(n - 1) * 2(t_{startup} + t_{data})$.

Analysis

- Parallel execution
  - Process P_{n-1} performs n - 1 multiplications and additions, one division and one subtraction.
  - Computation time $t_{comp} = 2(n - 1) + 2$.
  - Process P_{n-1} has (n - 1) recv and 1 send.
    - Communication time $t_{comm} = (2n - 1)(t_{startup} + t_{data})$.
  - Time $t_p = (n - 1) * 2(t_{startup} + t_{data}) + (2n - 1)(t_{startup} + t_{data}) + 2(n - 1) + 2 = O(n)$.
  - Speedup = $t_s / t_p = O(n)$ with n processors.