Partitioning, Divide-and-Conquer and Pipelining

Partitioning

Partitioning simply divides the problem into parts.

Divide and Conquer

Characterized by dividing problem into sub-problems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.

Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts. Also usually data is naturally localized.

Partitioning/Divide and Conquer Examples

Many possibilities.

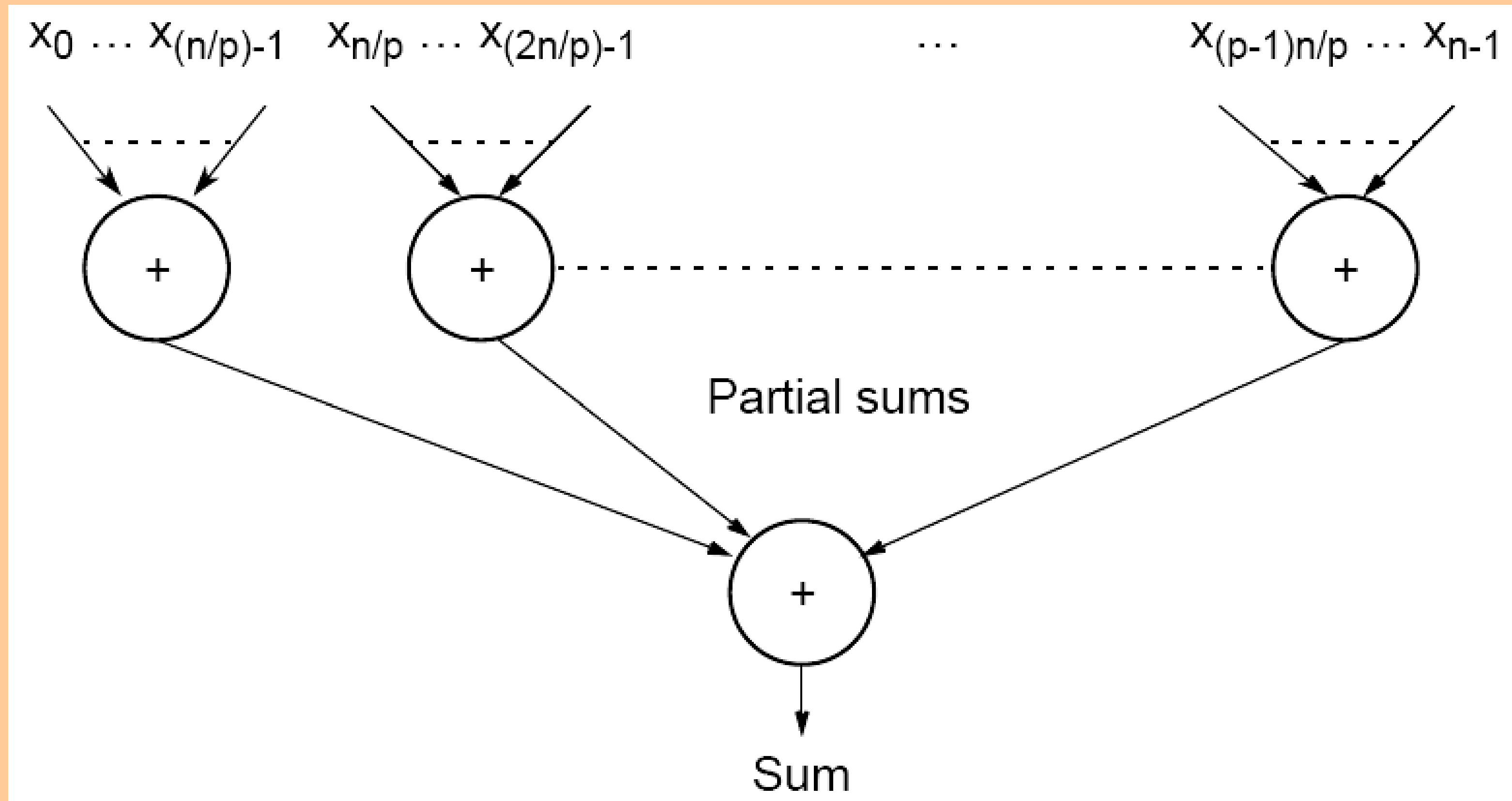
- Operations on sequences of number such as simply adding them together

- Several sorting algorithms can often be partitioned or constructed in a recursive fashion

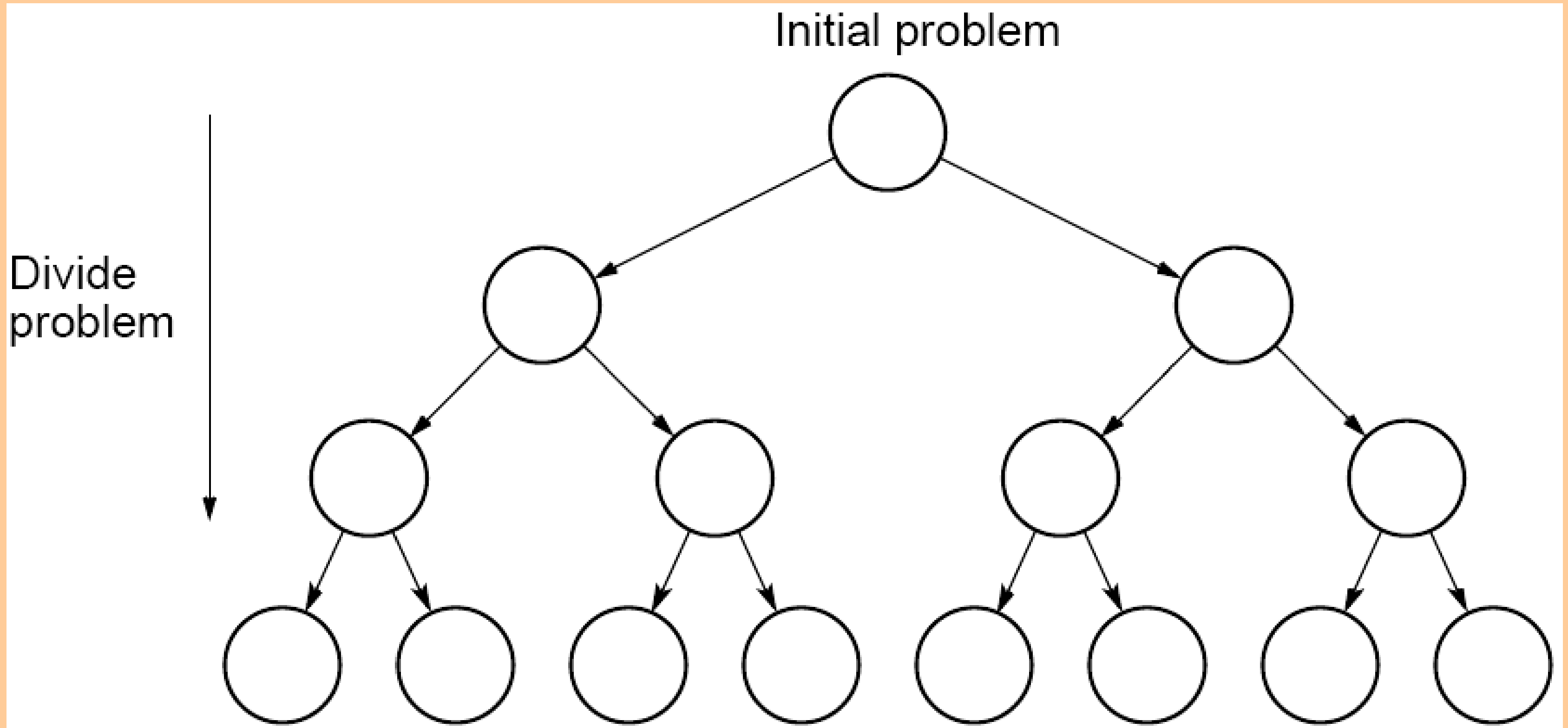
- Numerical integration

- *N*-body problem

Partitioning a sequence of numbers into parts and adding the parts

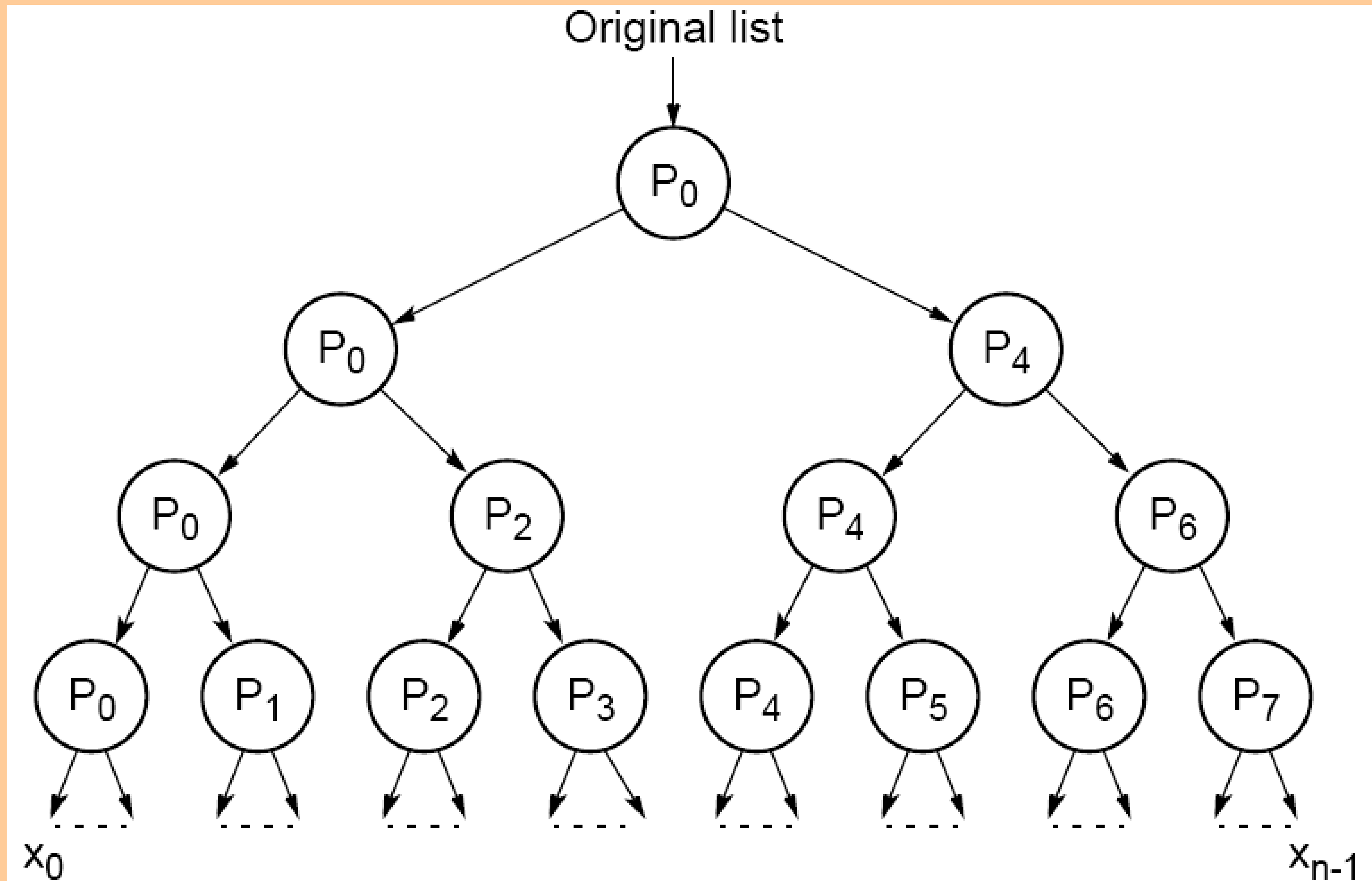


Tree construction

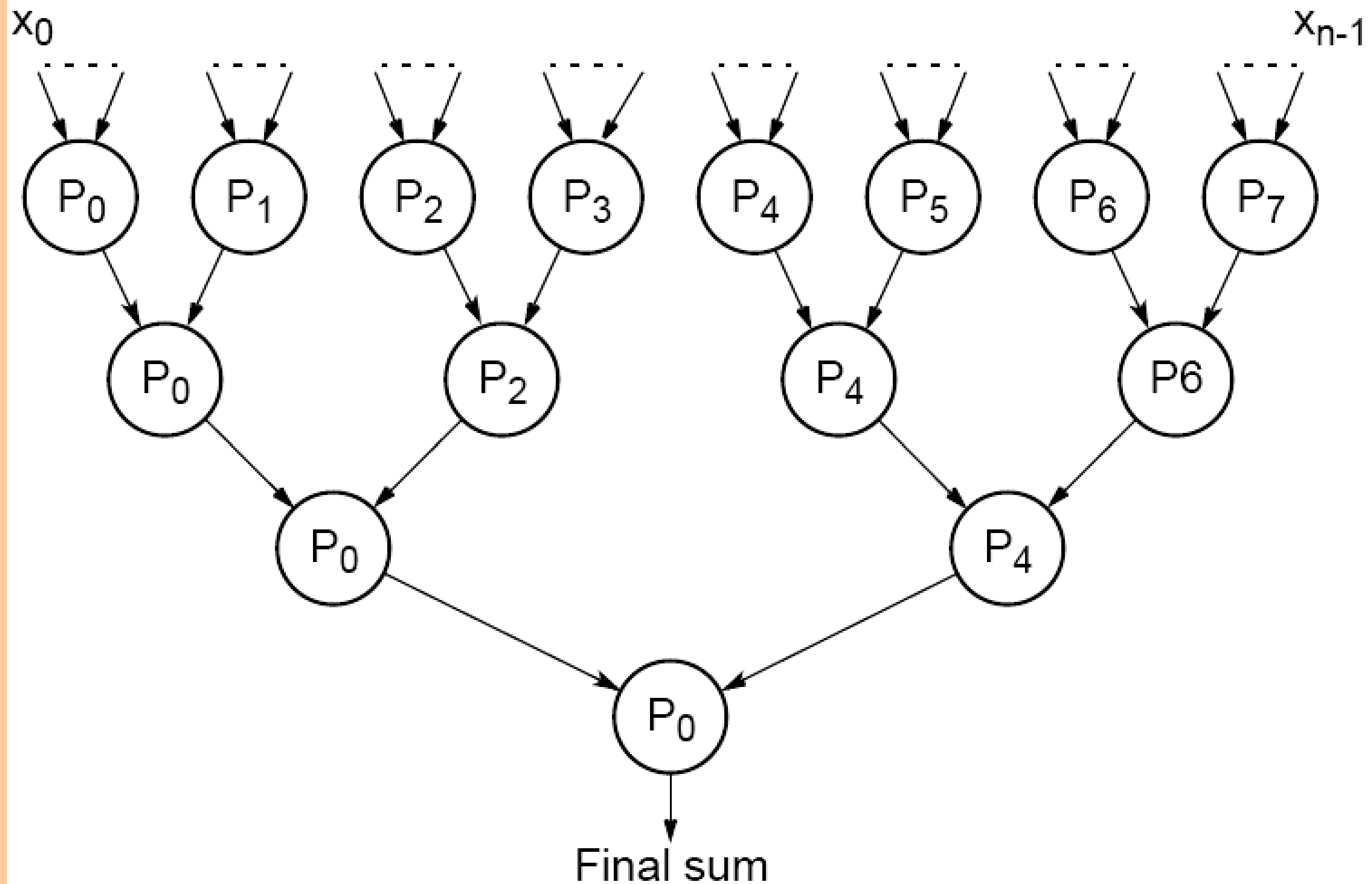


One CPU per node?

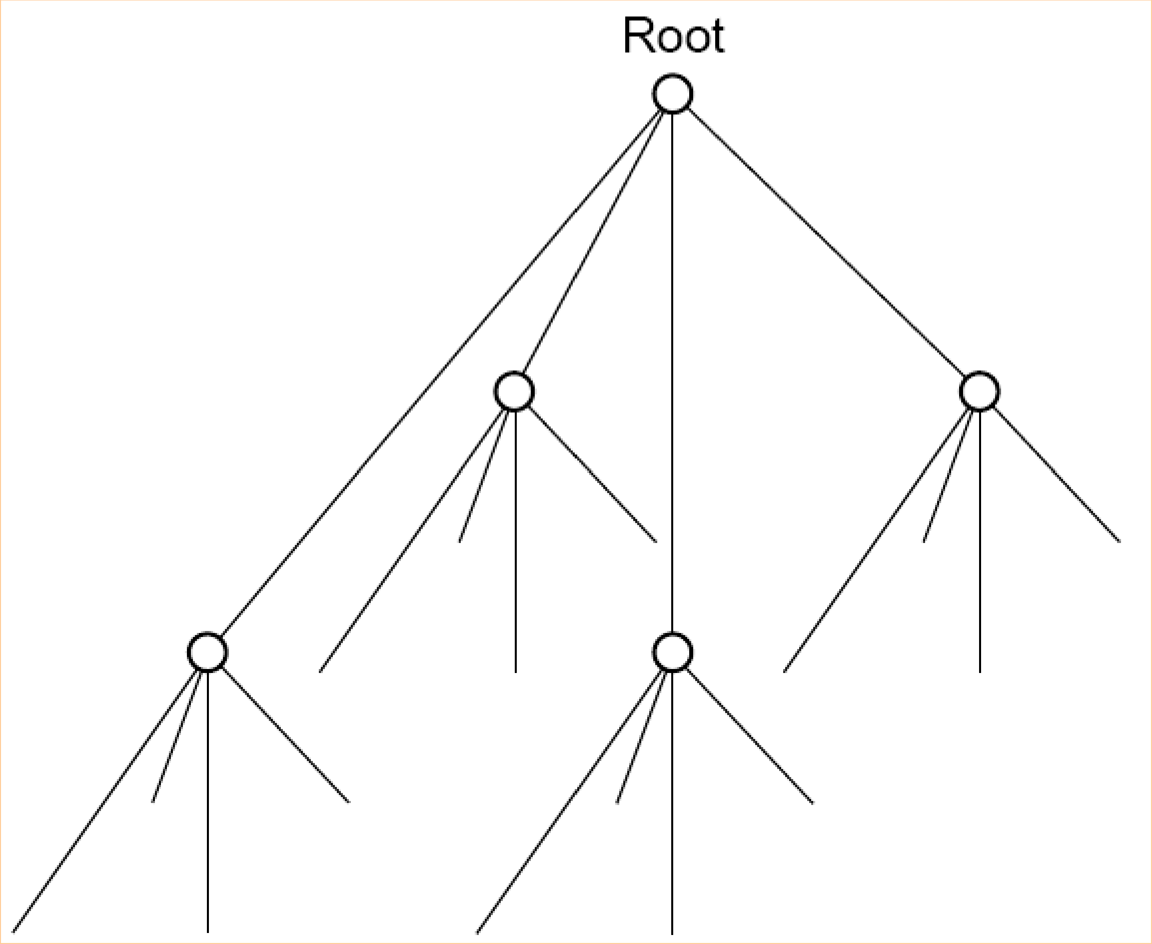
Dividing a list into parts



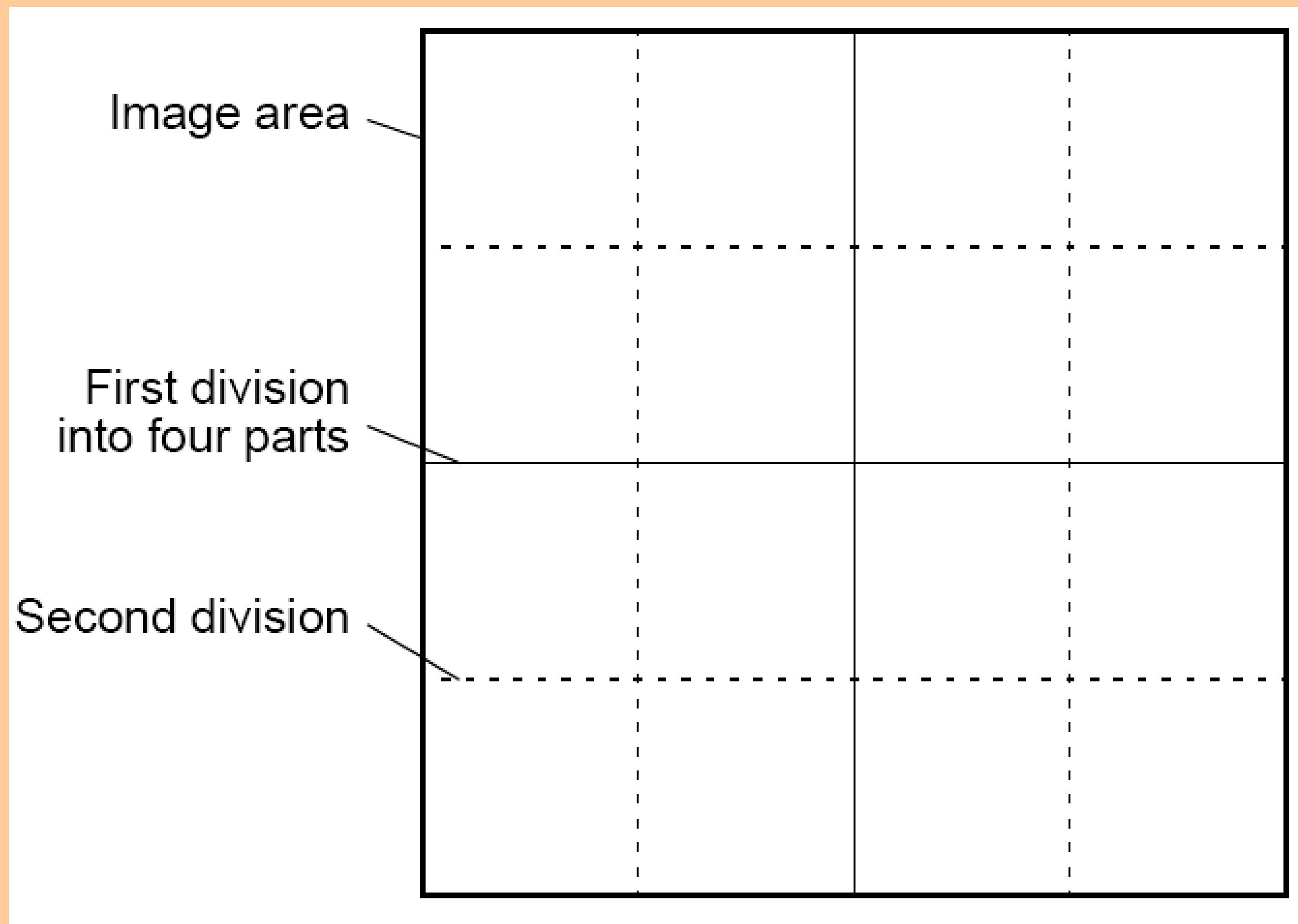
Partial summation



Quadtree

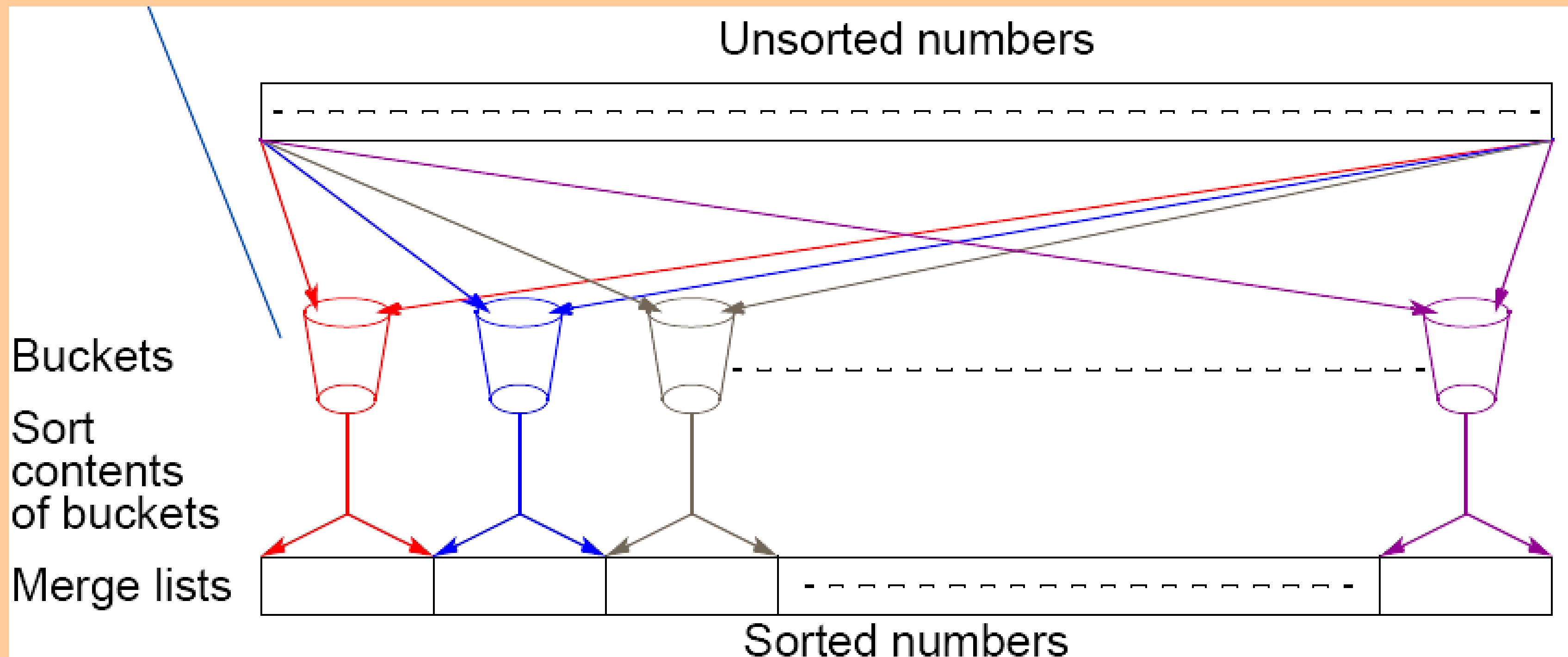


Dividing an image



Bucket sort

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

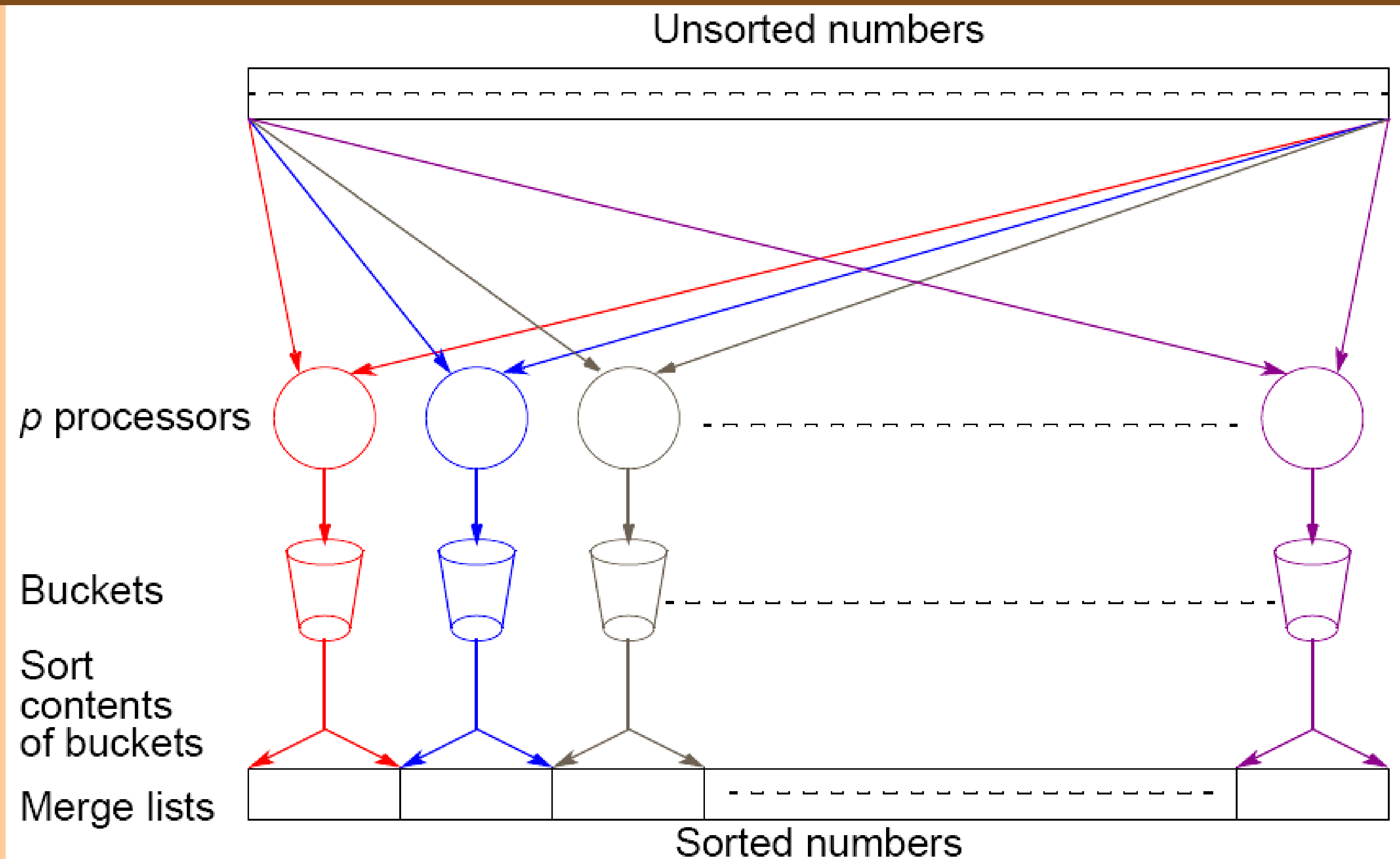


Sequential sorting time complexity: O(nlog(n/m)).
Works well if the original numbers uniformly distributed across a known interval, say 0 to *a* - 1.

Parallel version of bucket sort

Simple approach

Assign one processor for each bucket.



Each process sees all the numbers

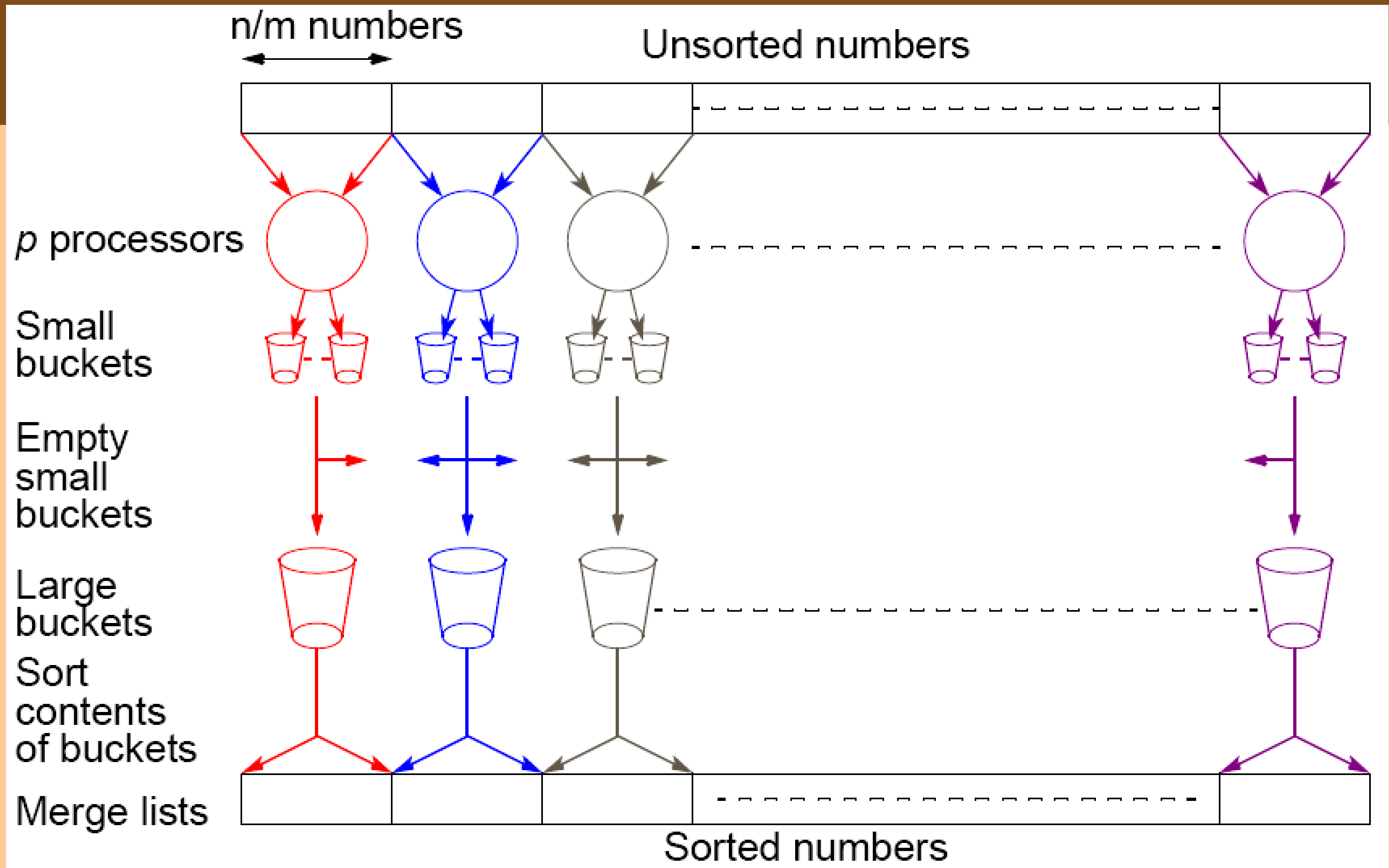
Further Parallelization

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

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

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

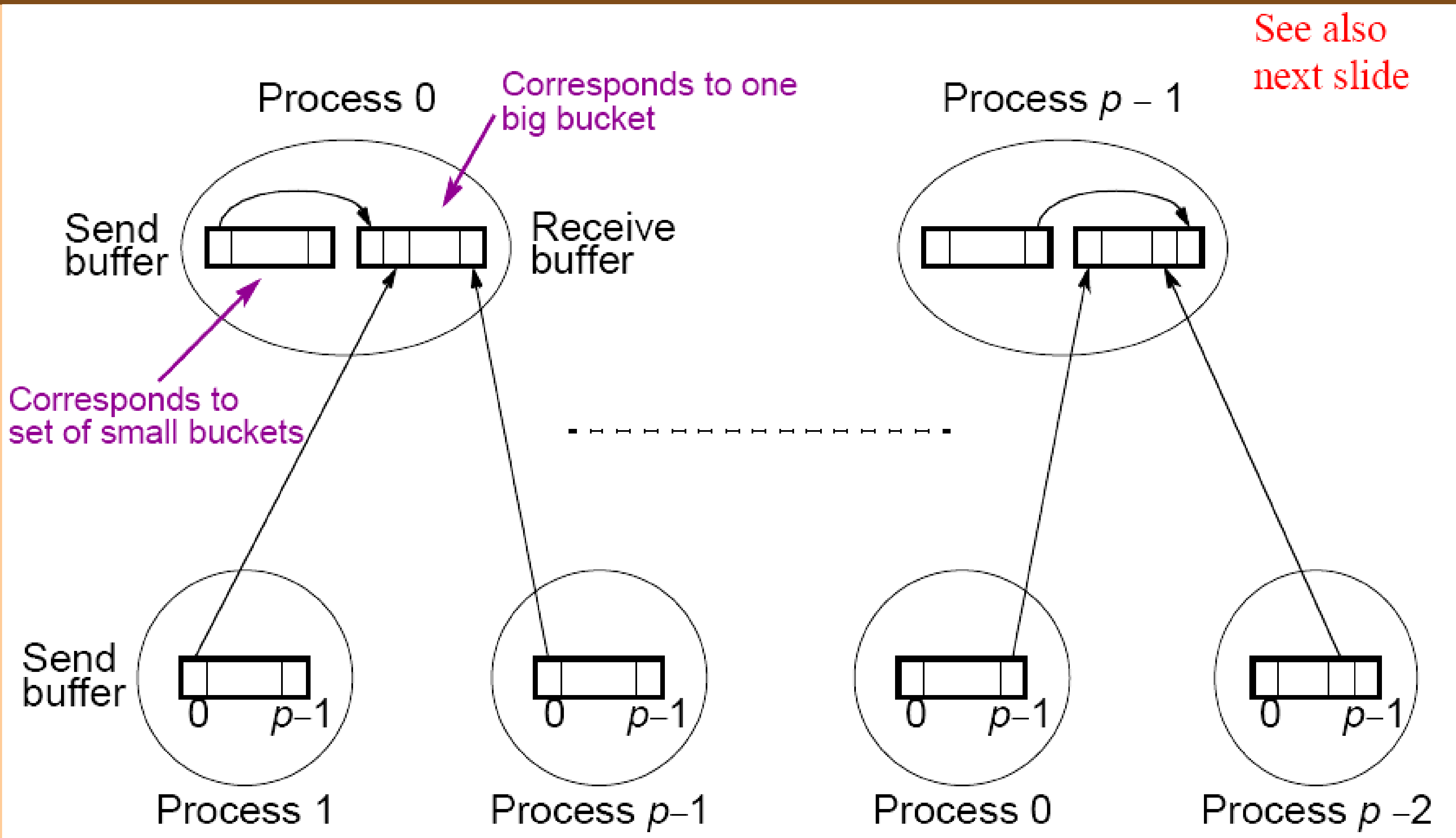
Another parallel version of bucket sort



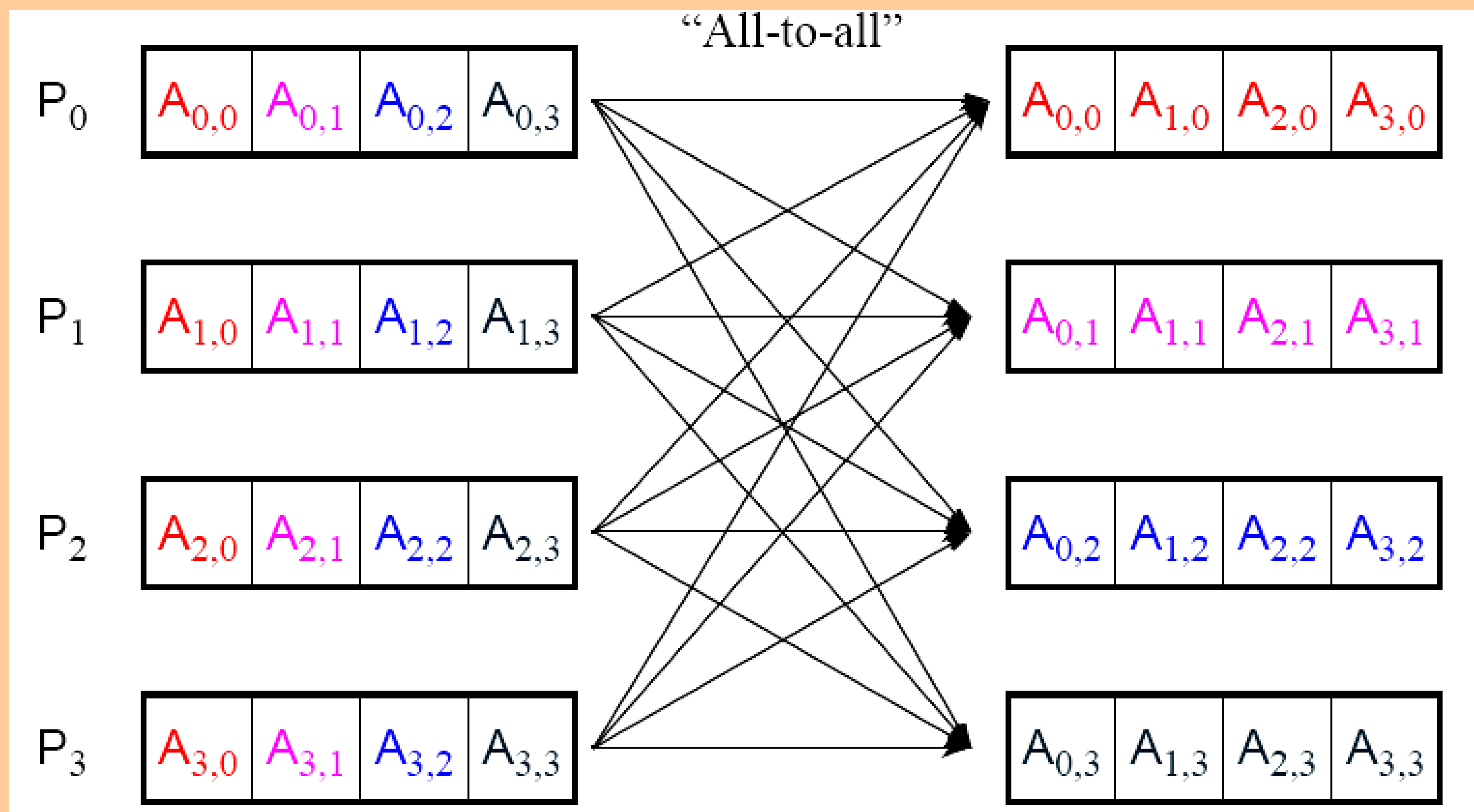
Can use collective communication - all-to-all broadcast.

“all-to-all” broadcast routine

Sends data from each process to every other process



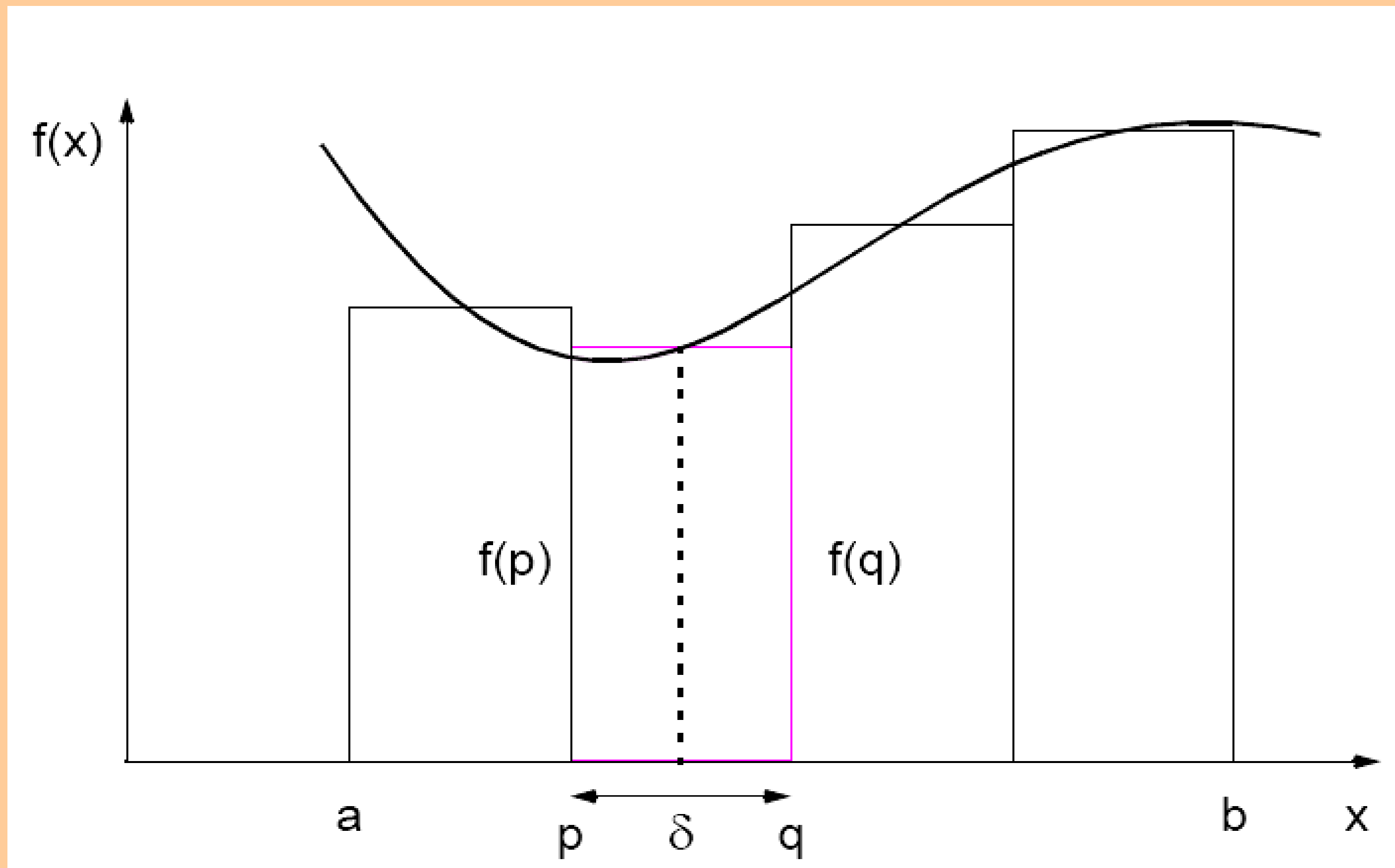
"all-to-all" routine actually transfers rows of an array to columns: Transposes a matrix.



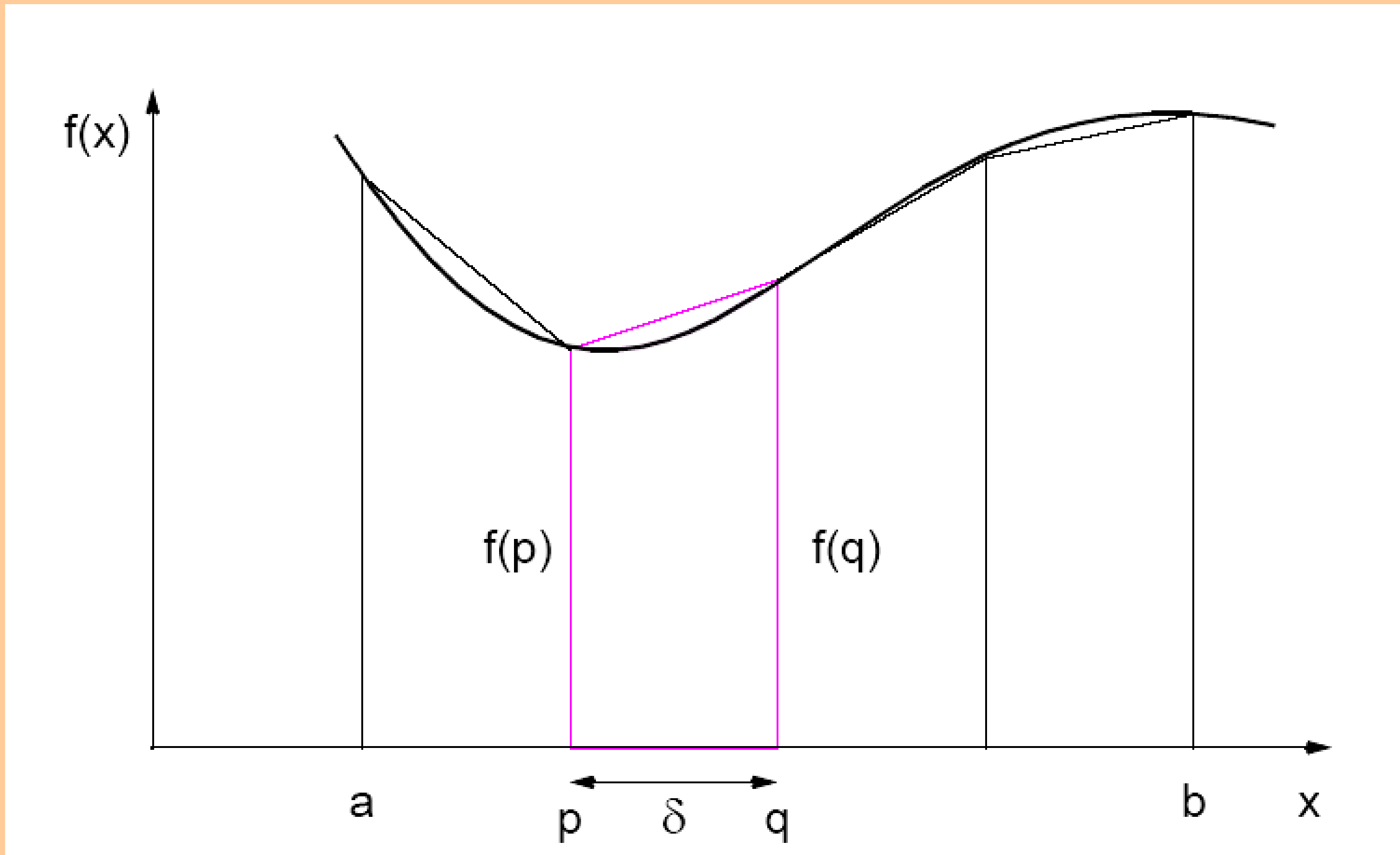
Numerical integration using rectangles

Each region calculated using an approximation given by rectangles:

Aligning the rectangles:



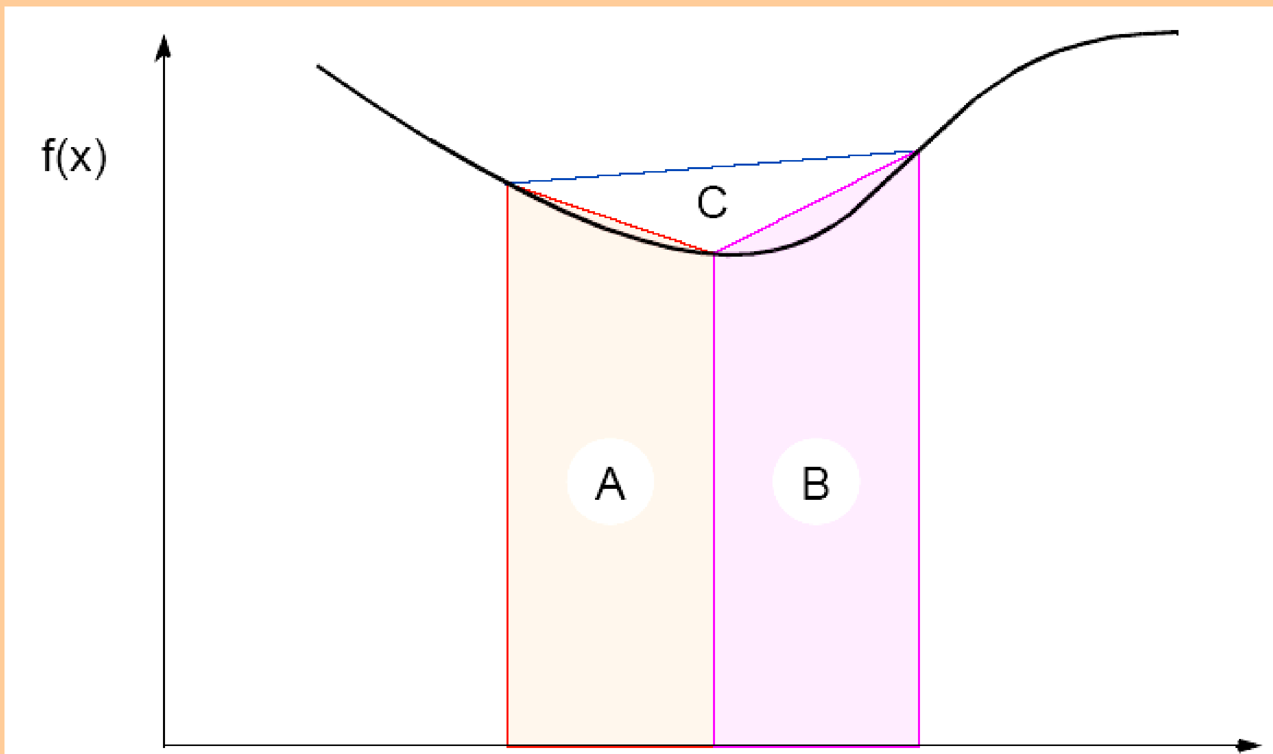
Numerical integration using trapezoidal method



May not be better!

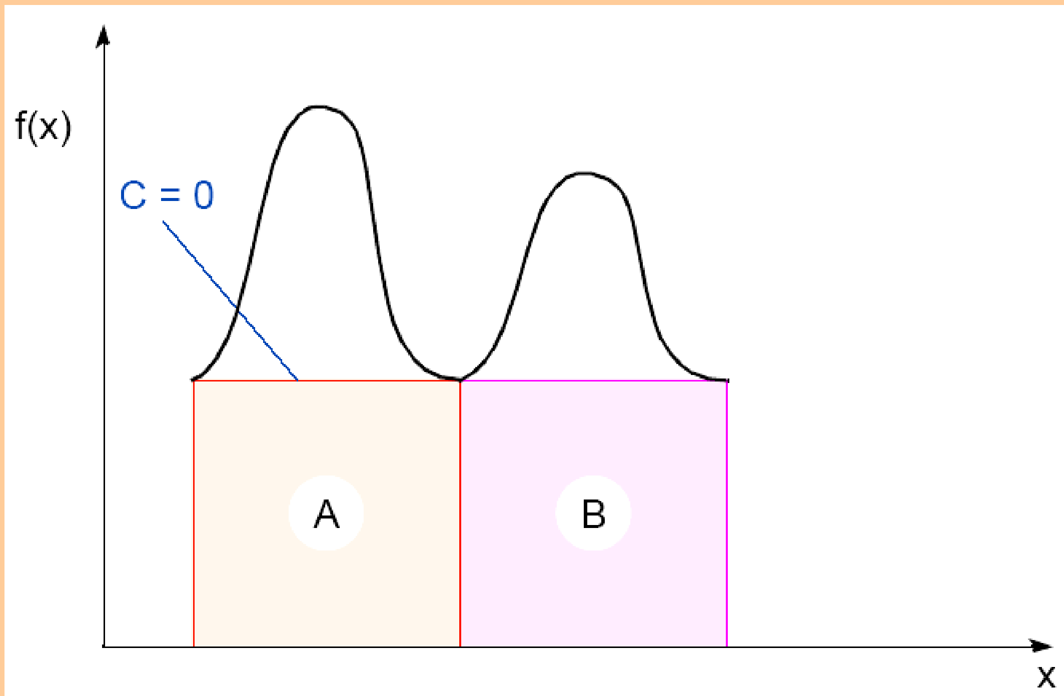
Adaptive Quadrature

Solution adapts to shape of curve. Use three areas, *A*, *B*, and *C*. Computation terminated when largest of *A* and *B* sufficiently close to sum of remain two areas .



Adaptive quadrature with false termination.

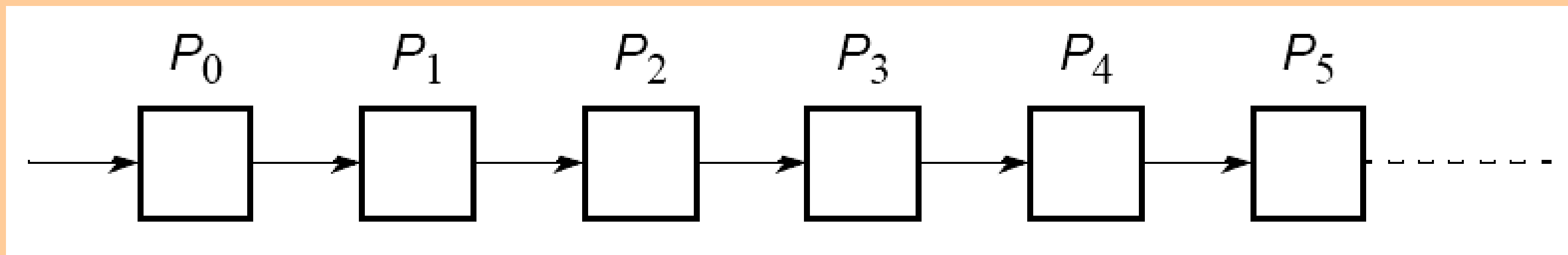
Some care might be needed in choosing when to terminate.



Might cause us to terminate early, as two large regions are the same (i.e., $C = 0$).

Pipelined Computations

Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.



Example

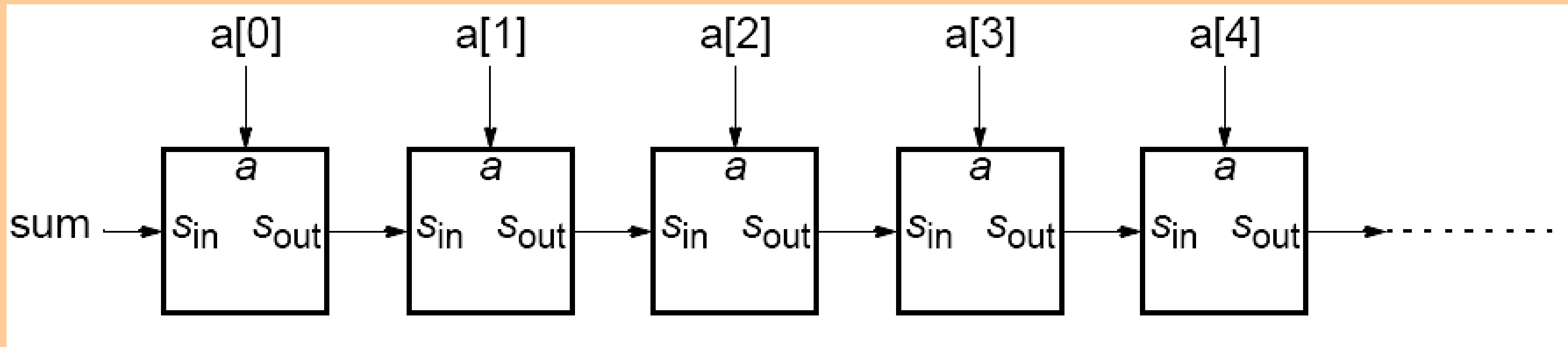
Add all the elements of array **a** to an accumulating sum:

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

The loop could be “unrolled” to yield

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
          .
          .
          .
```

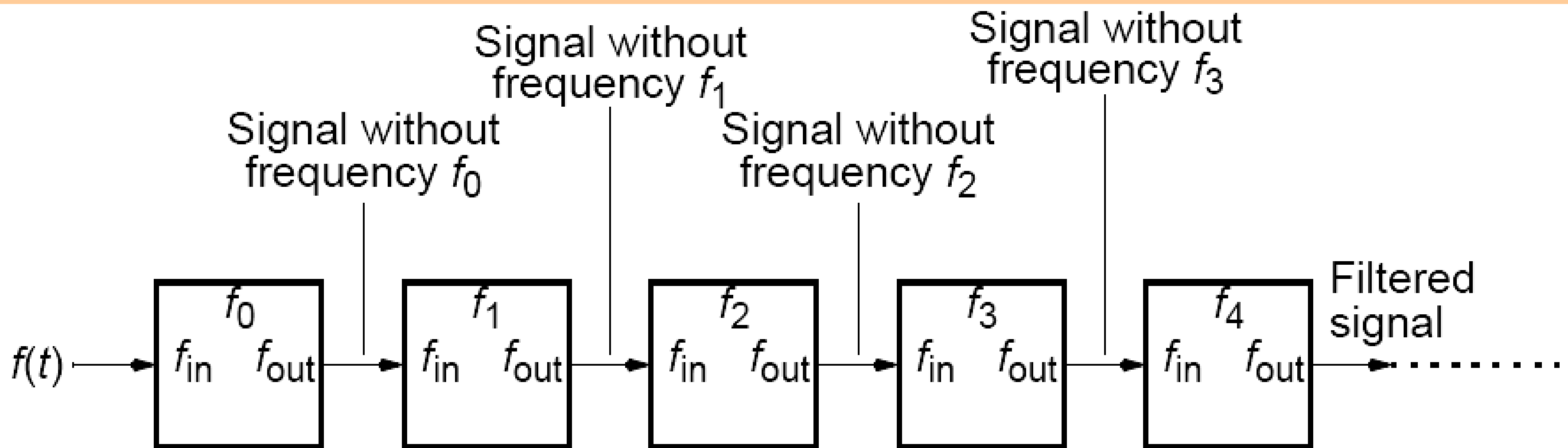
Pipeline for an unrolled loop



Another Example

Frequency filter - Objective to remove specific frequencies (f_0, f_1, f_2,f_3, etc.) from a digitized signal, $f(t)$.

Signal enters pipeline from left:



Where pipelining can be used to good effect

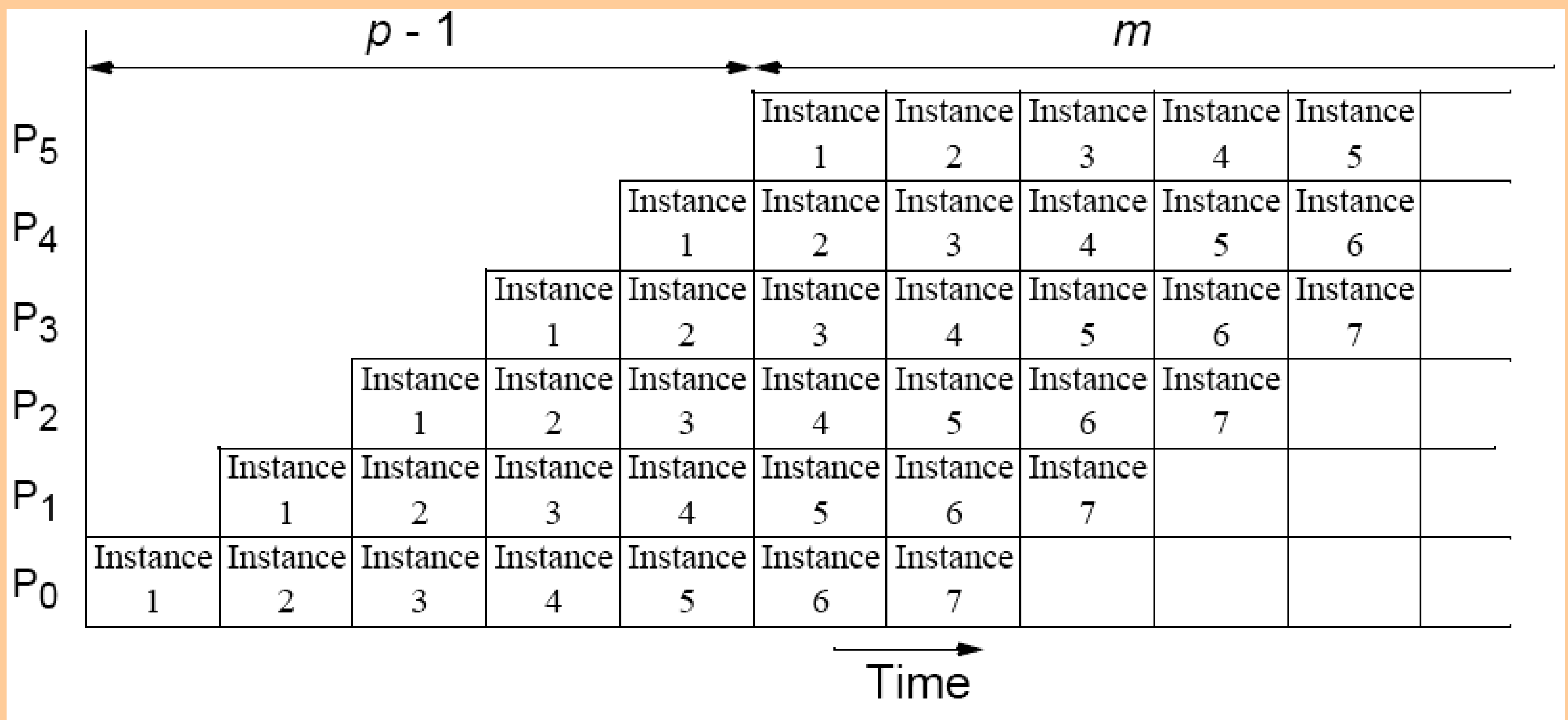
Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations:

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

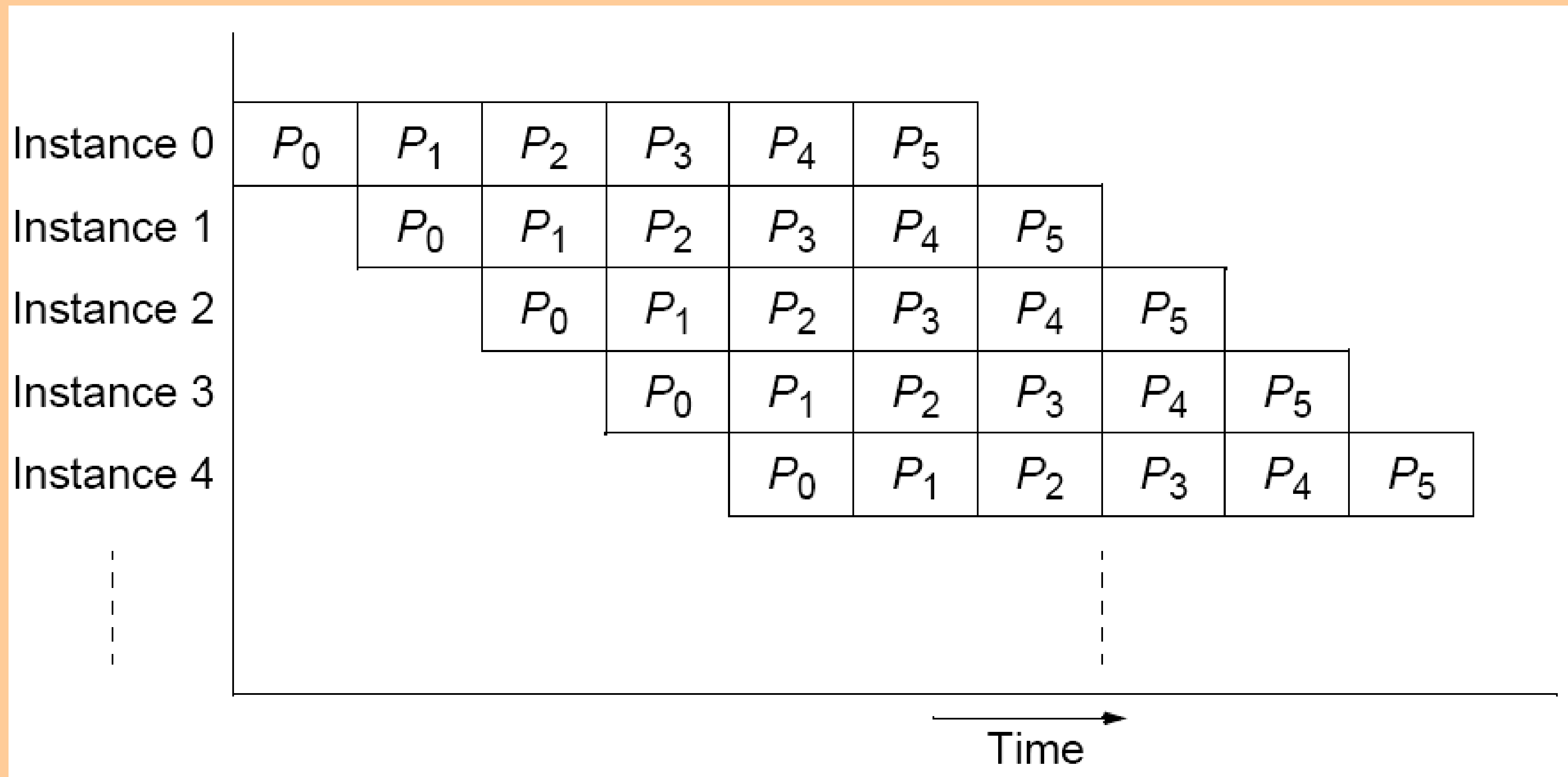
2. If a series of data items must be processed, each requiring multiple operations

3. If information to start next process can be passed forward before process has completed all its internal operations

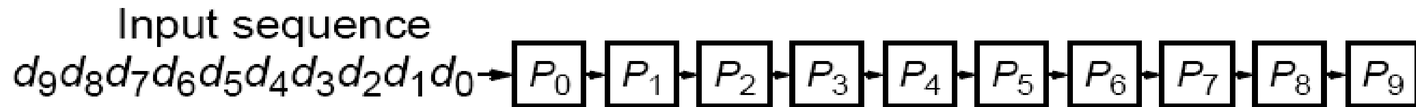
“Type 1” Pipeline Space-Time Diagram



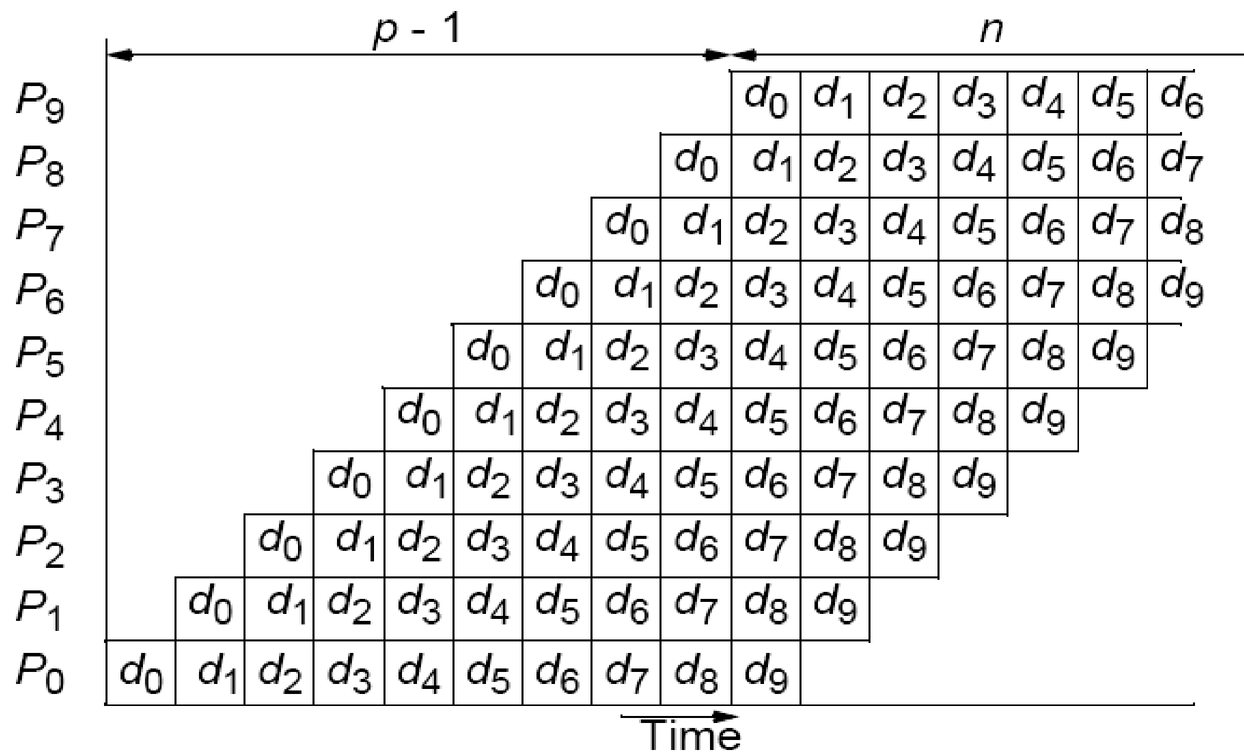
Alternative space-time diagram



“Type 2” Pipeline Space-Time Diagram



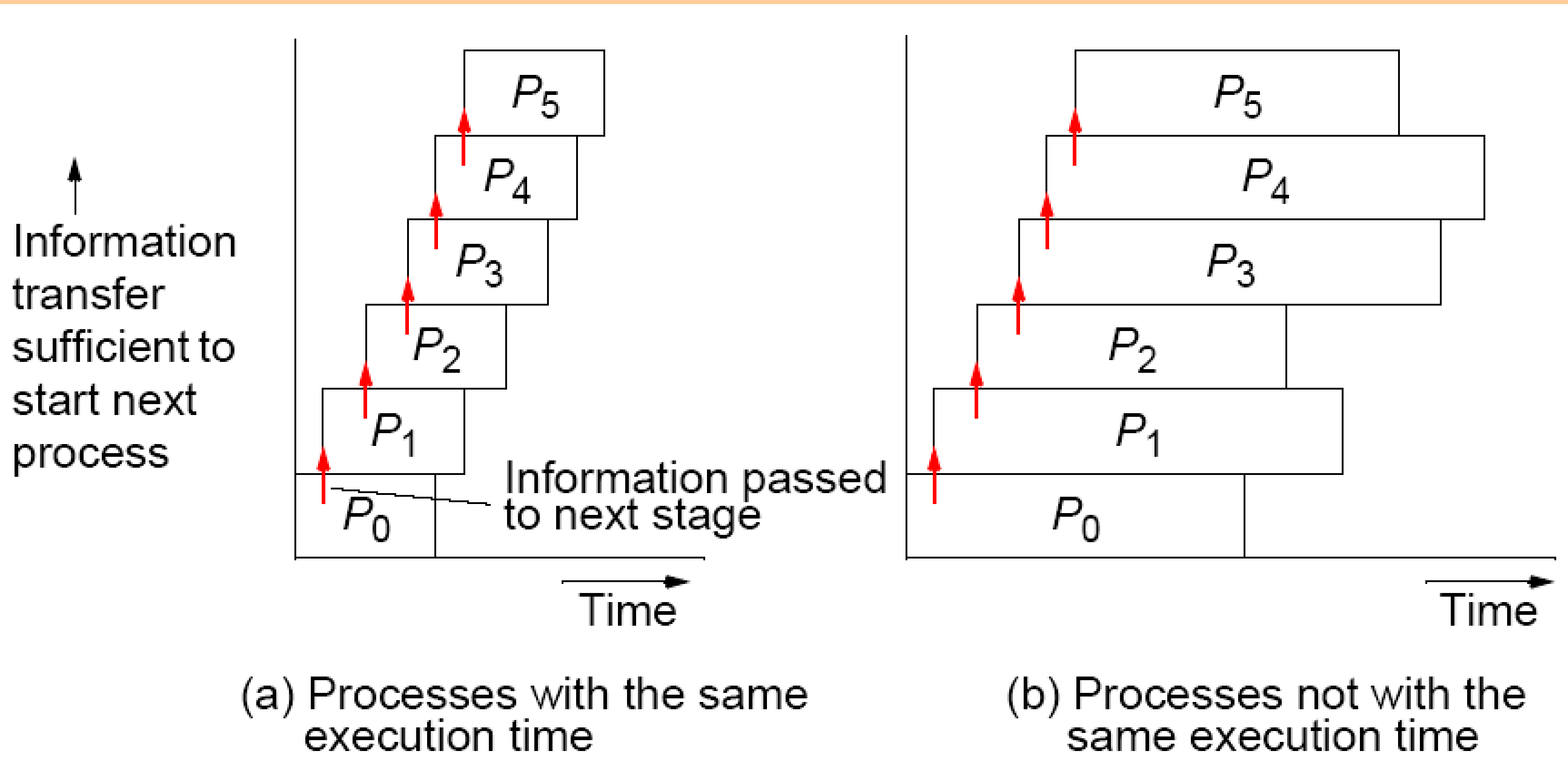
(a) Pipeline structure



(b) Timing diagram

“Type 3” Pipeline Space-Time Diagram

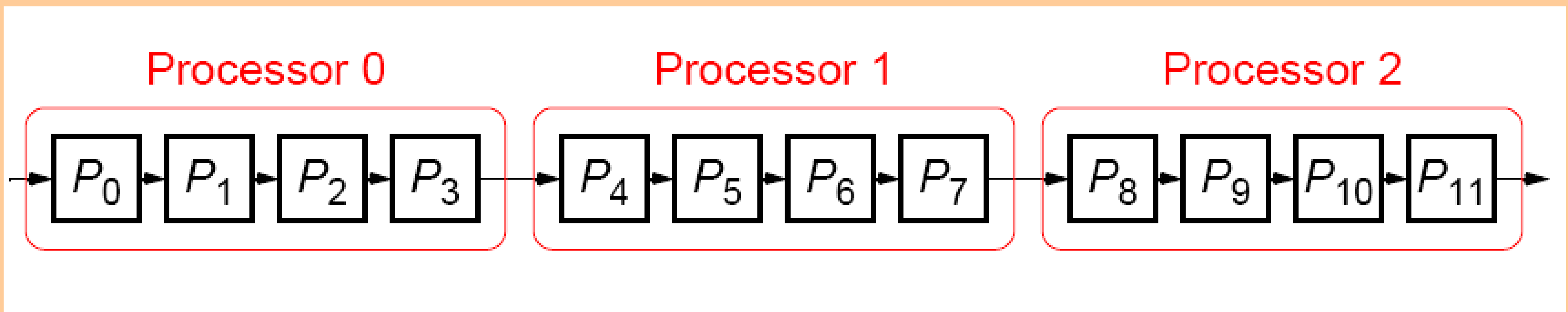
Pipeline processing where information passes to next stage before previous state completed.



(a) Processes with the same execution time

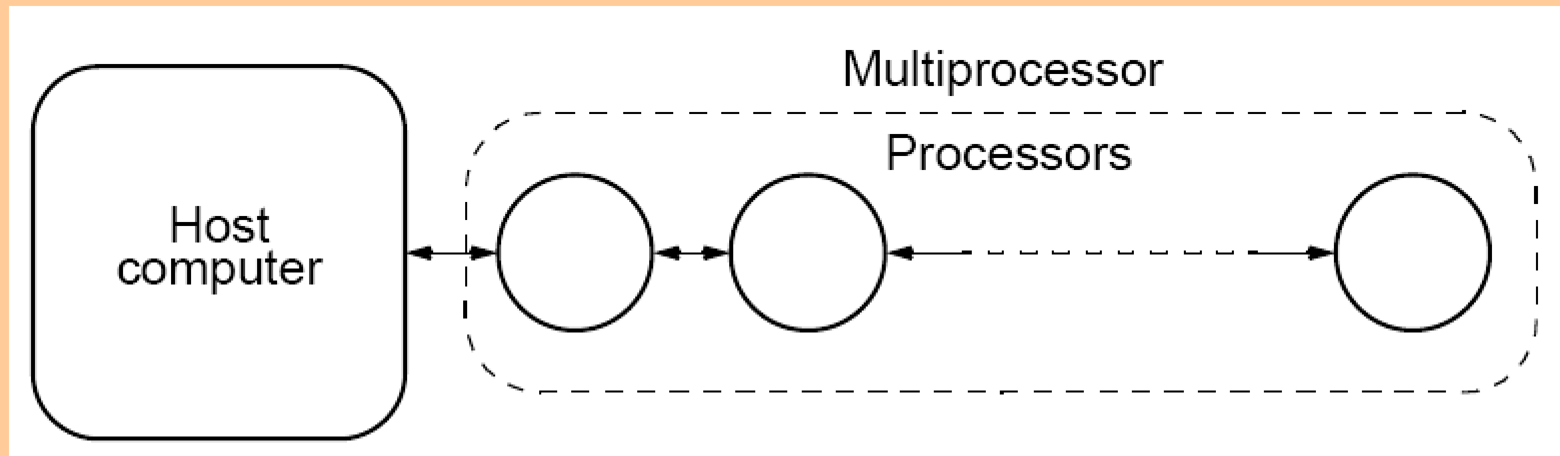
(b) Processes not with the same execution time

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



Computing Platform for Pipelined Applications

Multiprocessor system with a line configuration



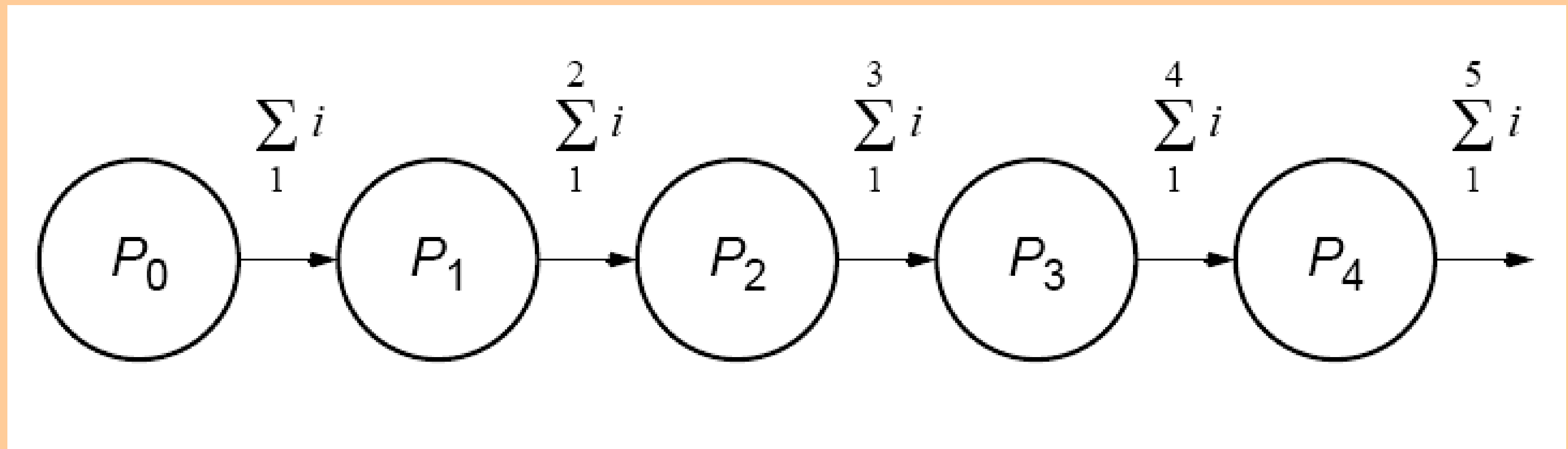
Strictly speaking pipeline may not be the best structure for a cluster - however a cluster with switched direct connections, as most have, can support simultaneous message passing.

Example Pipelined Solutions

(Examples of each type of computation)

Pipeline Program Examples

Adding Numbers



Type 1 pipeline computation

Basic code for process *Pi* :

```
recv(&accumulation, Pi-1);
accumulation = accumulation + number;
send(&accumulation, Pi+1);
```

except for the first process, *P*0, which is

```
send(&number, P1);
```

and the last process, *Pn*-1, which is

```
recv(&number, Pn-2);
accumulation = accumulation + number;
```

SPMD program

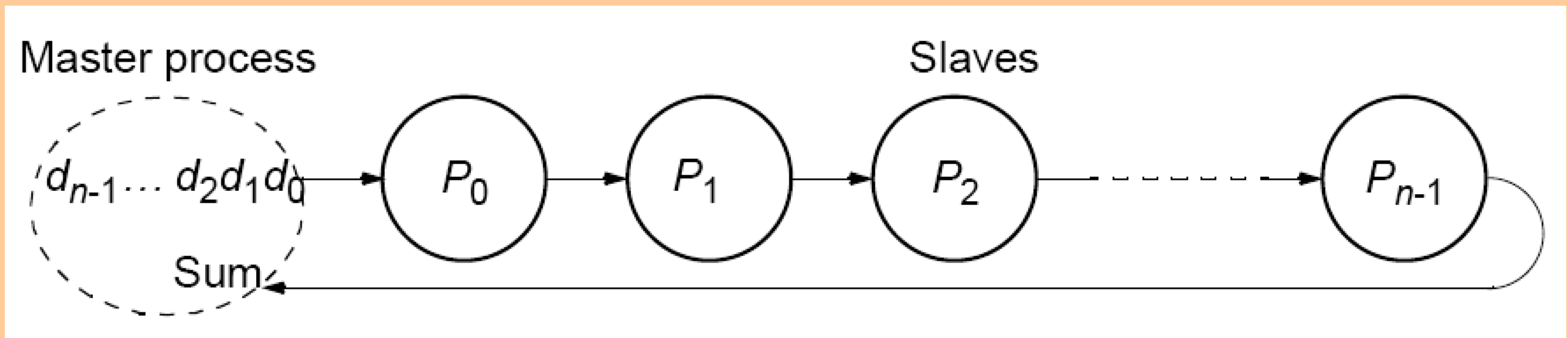
```
if (process > 0) {
        recv(&accumulation, Pi-1);
        accumulation = accumulation + number;
}
if (process < n-1)
        send(&accumulation, P i+1);
```

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

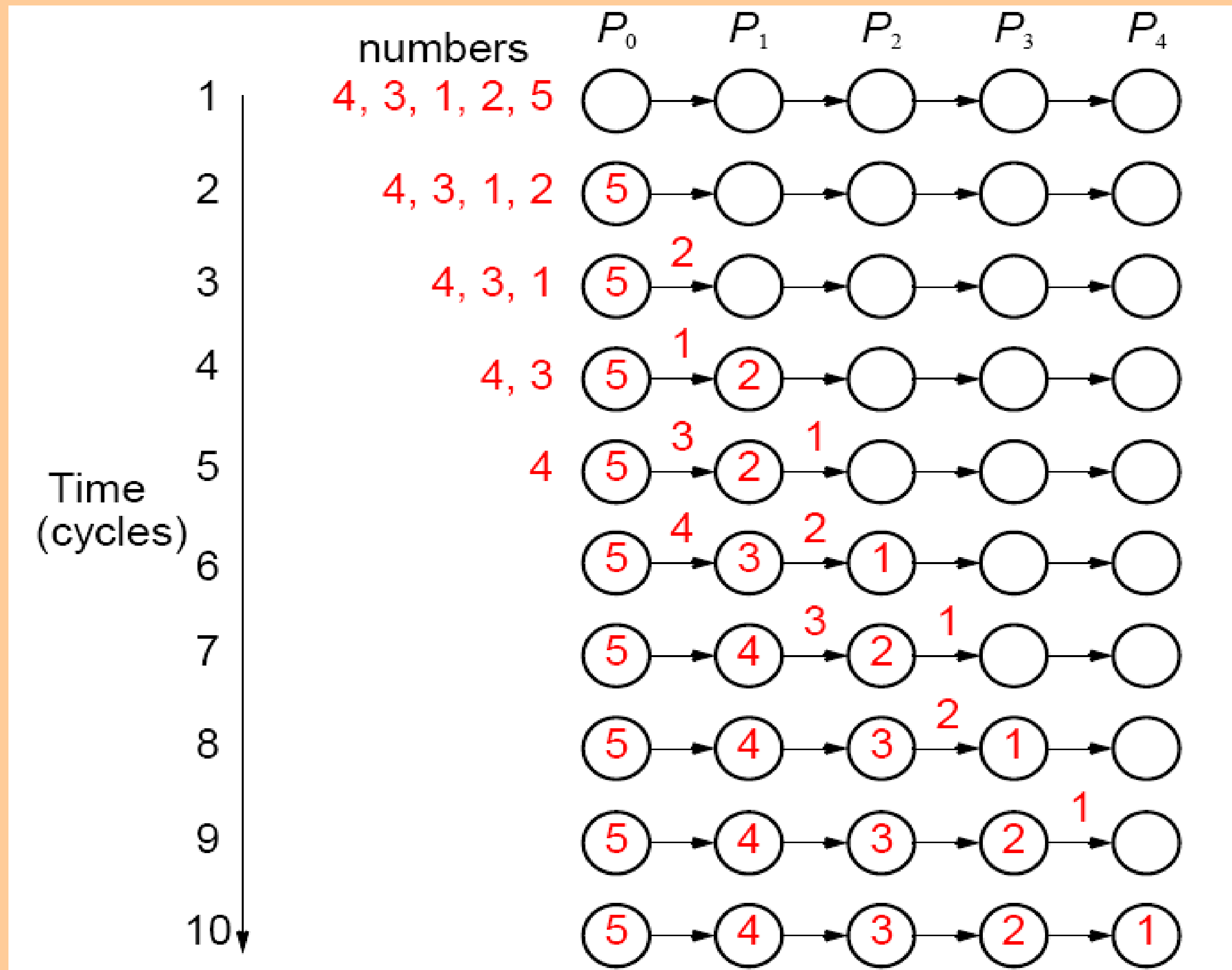
Pipelined addition of numbers

Master process and ring configuration

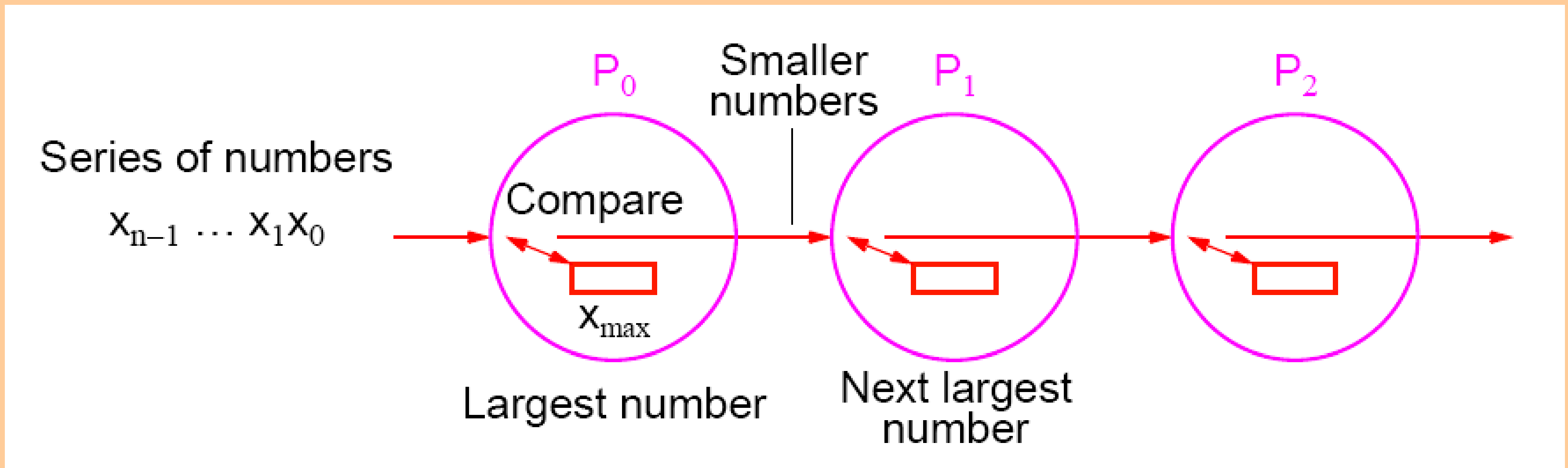


Sorting Numbers

A parallel version of *insertion sort*.



Pipeline for sorting using insertion sort



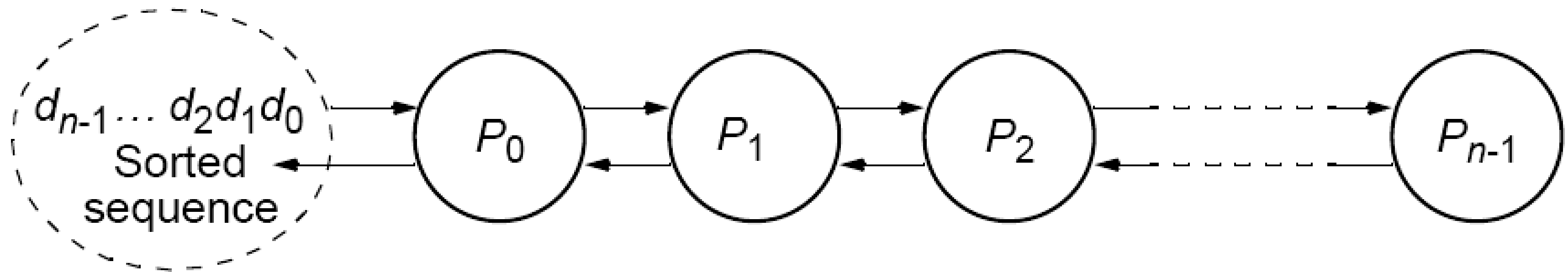
Type 2 pipeline computation

The basic algorithm for process *Pi* is

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

With n numbers, number ith process is to accept = $n - i$.
Number of passes onward = $n - i - 1$
Hence, a simple loop could be used.

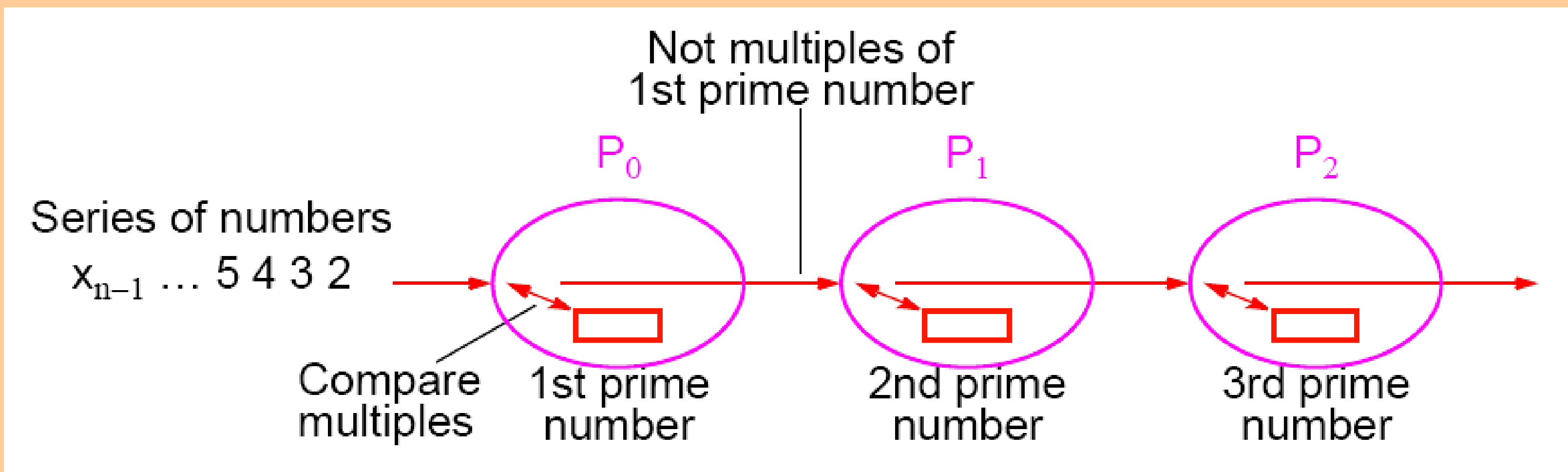
Insertion sort with results returned to master process using bidirectional line configuration



Insertion sort with results returned

Prime Number Generation
Sieve of Eratosthenes

- Series of all integers generated from 2.
- First number, 2, is prime and kept.
- All multiples of this number deleted as they cannot be prime.
- Process repeated with each remaining number.
- The algorithm removes non-primes, leaving only primes.



Type 2 pipeline computation

The code for a process, *Pi*, could be based upon

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

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

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

Solving a System of Linear Equations

Upper-triangular form

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

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

Back Substitution

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

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

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

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

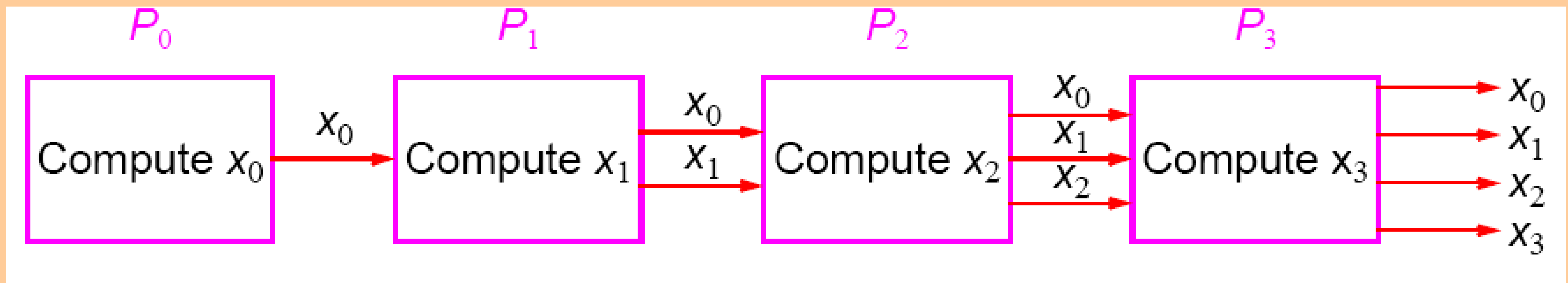
Values obtained for x_1 and x_0 substituted into next equation to obtain x_2:

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

and so on until all the unknowns are found.

Pipeline Solution

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



Type 3 pipeline computation

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

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

Sequential Code

Given constants $a_{i,j}$ and b_k stored in arrays **a[ ][ ]** and **b[ ]**, respectively, and values for unknowns to be stored in array, **x[ ]**, sequential code could be

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

Parallel Code

Pseudocode of process P_i ($1 < i < n$) of could be

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

Now have additional computations to do after receiving and resending values.

Pipeline processing using back substitution