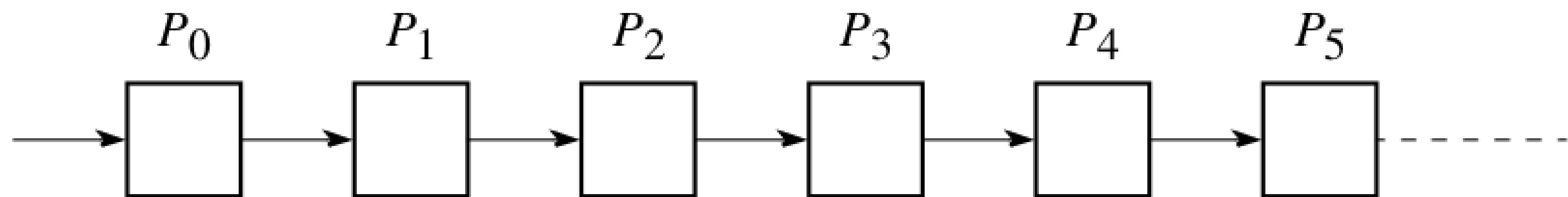
Pipelined Computations

Based on chapter 5 from Wilkinson & Allen

In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other.

In fact, this is the basis of sequential programming.

Each task will be executed by a separate process or processor.



Example

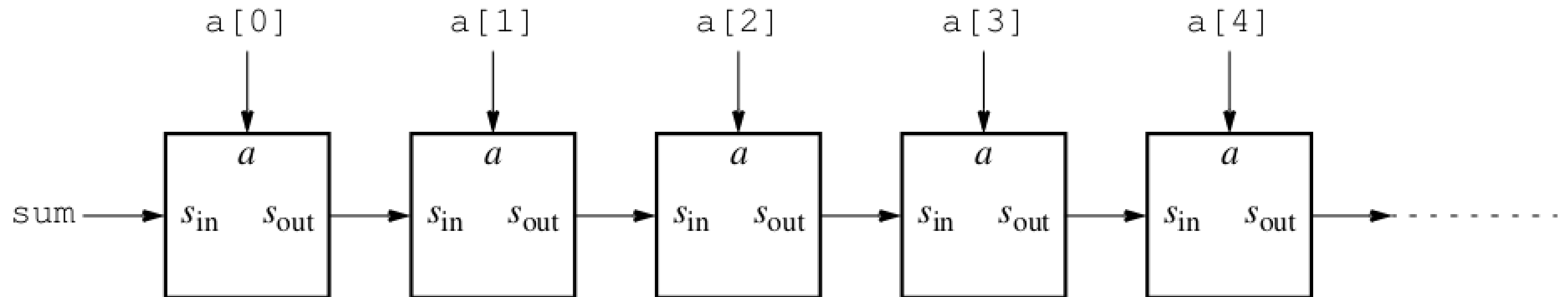
Add all the elements of array **`a`** to an accumulating sum:

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

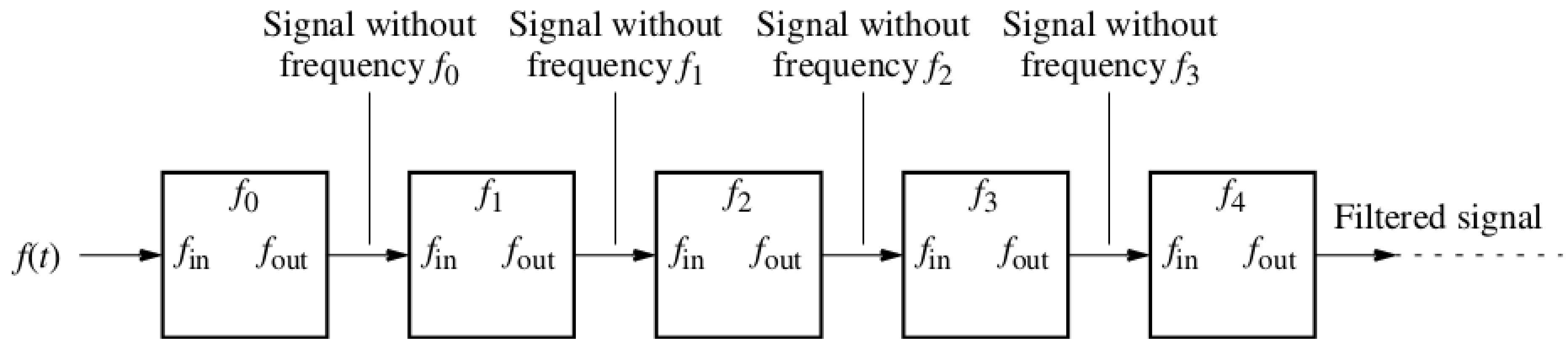
The loop could be “unfolded” to yield

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

Unfolded loop



Frequency Filter

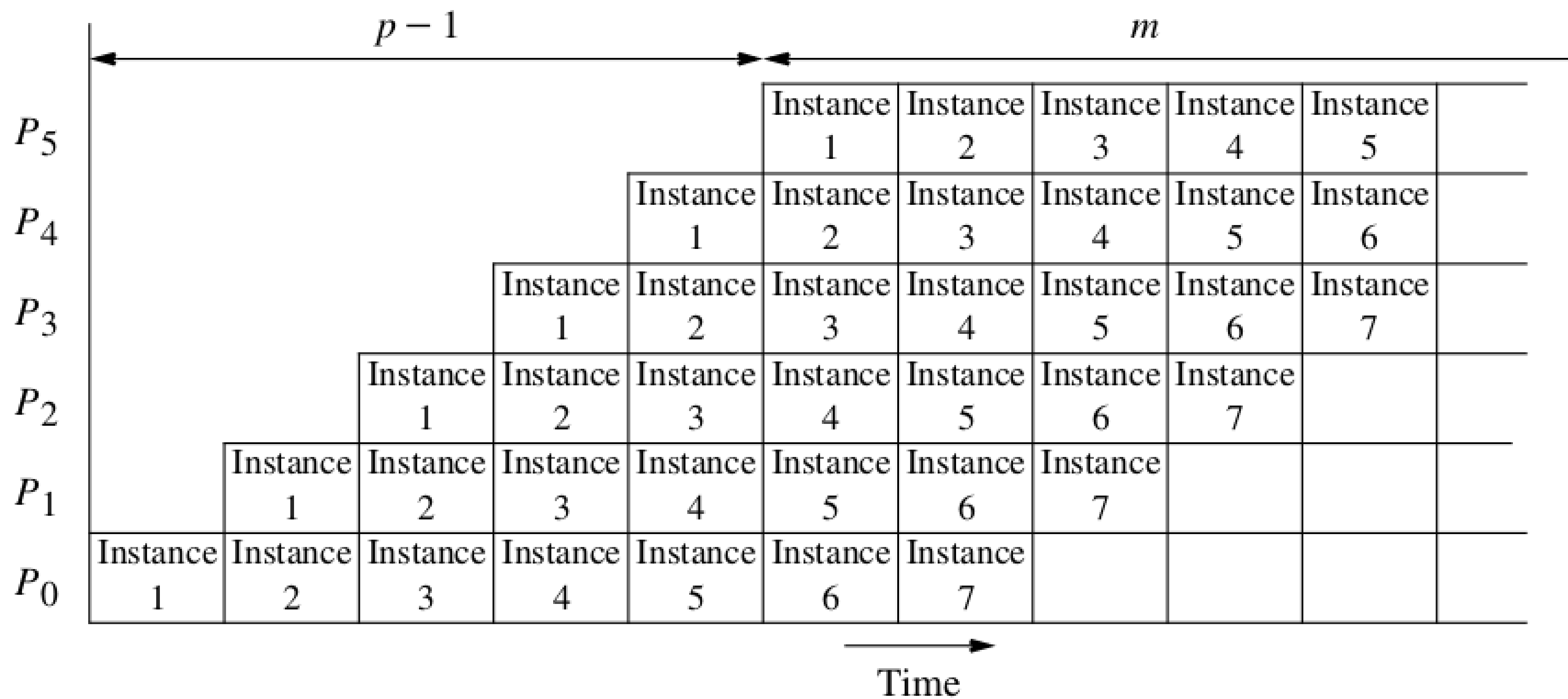


Conditions for a potential increased speed

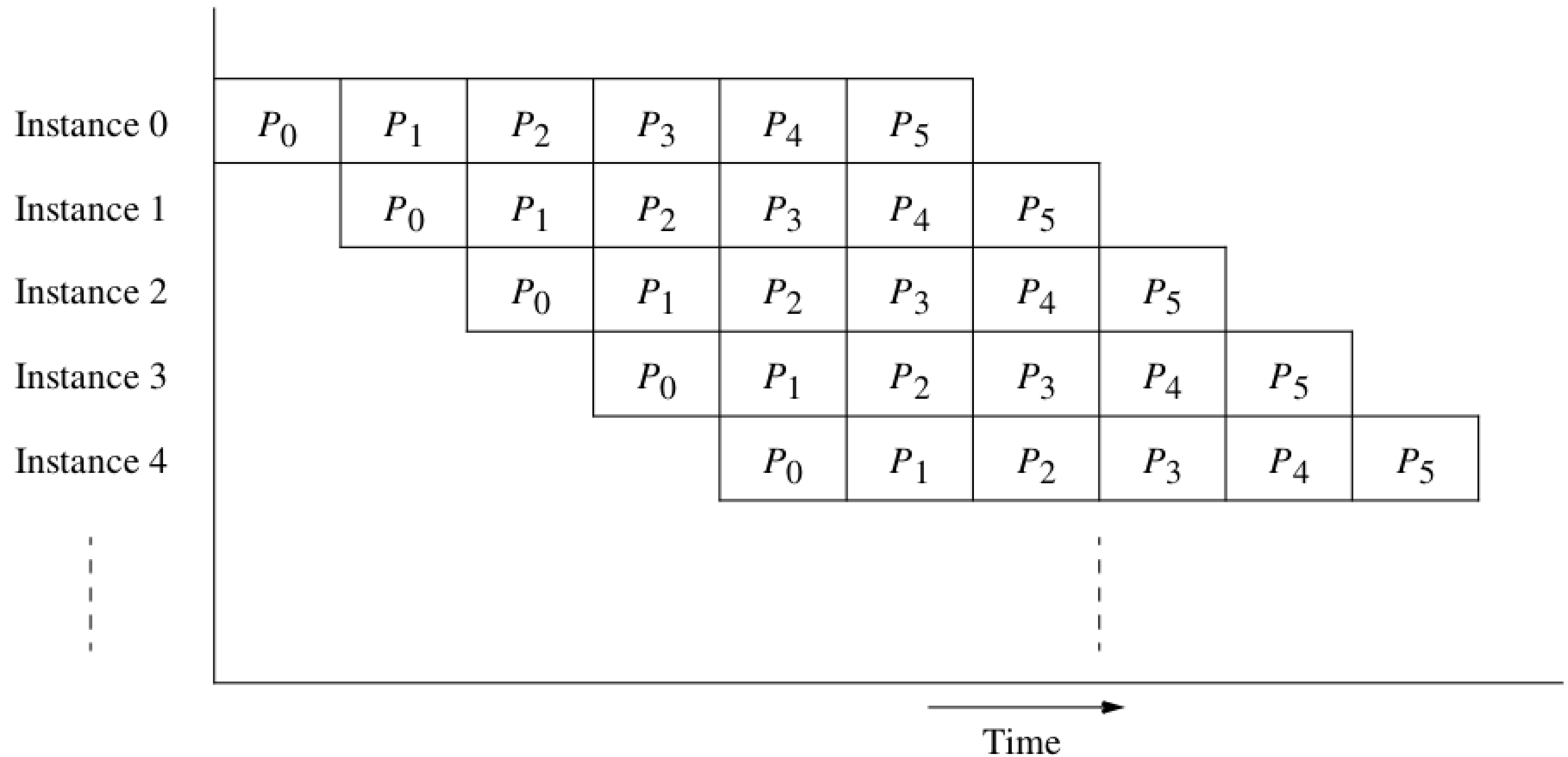
Given that the problem can be divided into a series of sequential tasks, the pipelined approach can provide increased speed under the following three types of computations:

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 the next process can be passed forward before the process has completed all its internal operations

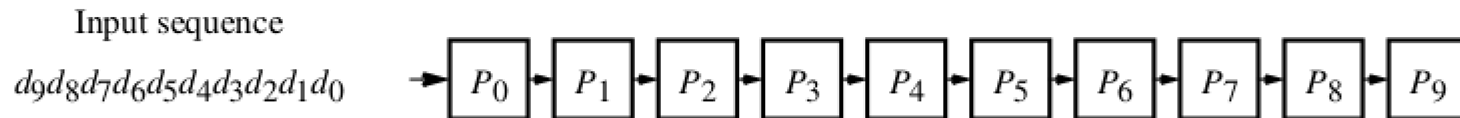
"Type 1" Pipeline Space-Time Diagram



Alternative space-time diagram



“Type 2”



(a) Pipeline structure

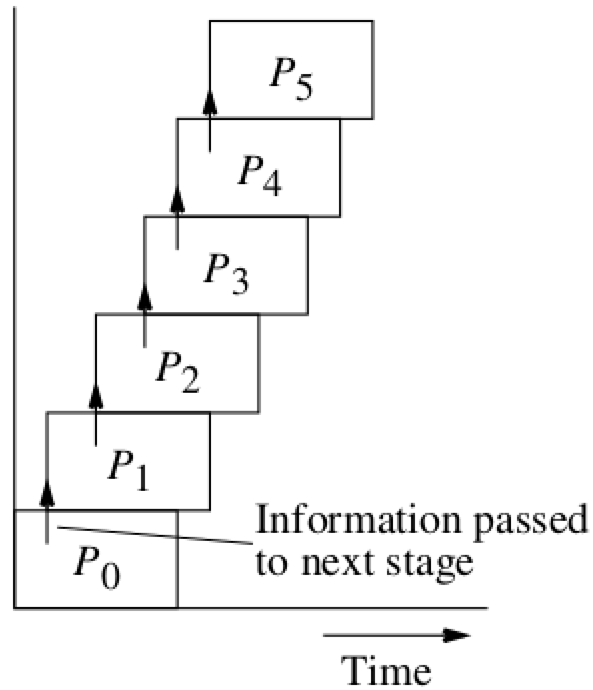
$p-1$ | n

P_9										d_0	d_1	d_2	d_3	d_4	d_5	d_6
P_8									d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
P_7								d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
P_6							d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
P_5						d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	
P_4					d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9		
P_3				d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9			
P_2			d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9				
P_1		d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9					
P_0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9						

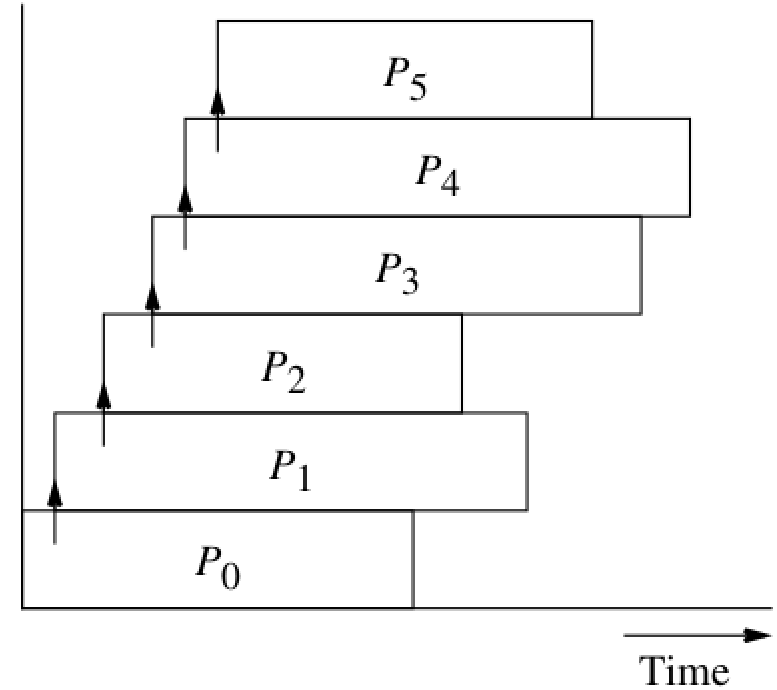
Time →

“Type 3”

Information transfer sufficient to start next process

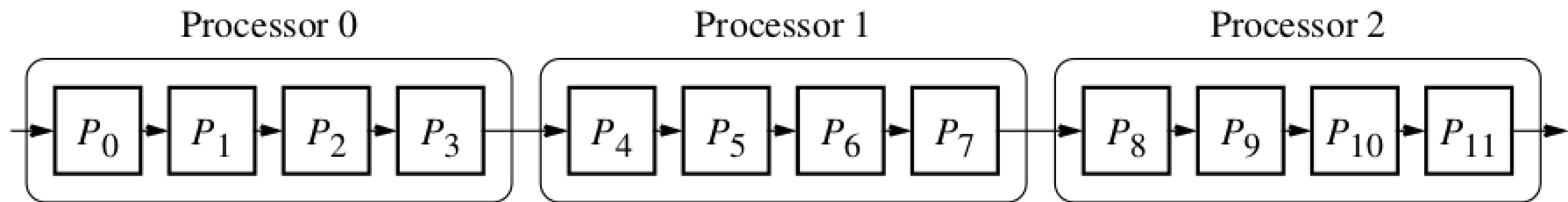


(a) Processes with the same execution time

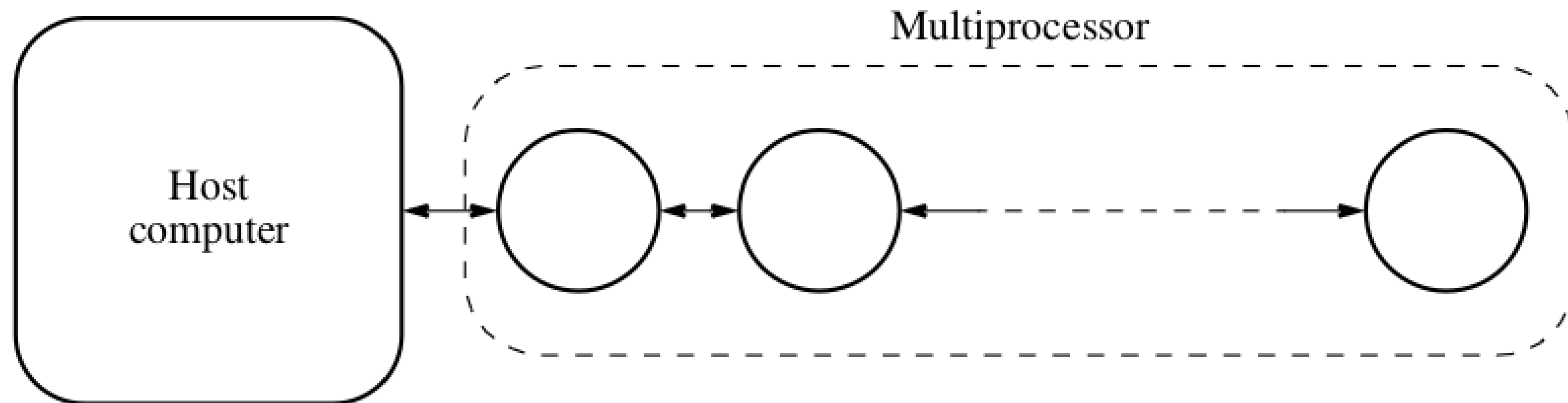


(b) Processes not with the same execution time

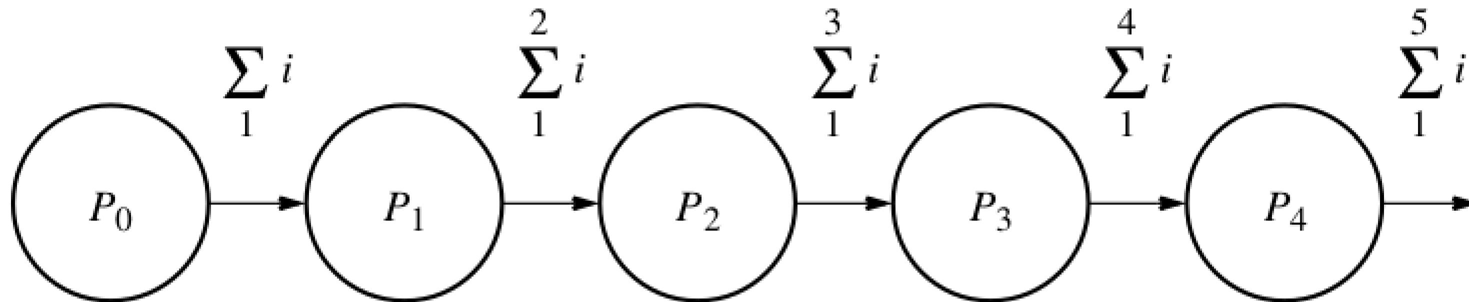
Partitioning processes onto processors



Computing platform for pipelined applications



Example: adding numbers



The basic code for process P_i :

```
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, P_{n-1}, which is

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

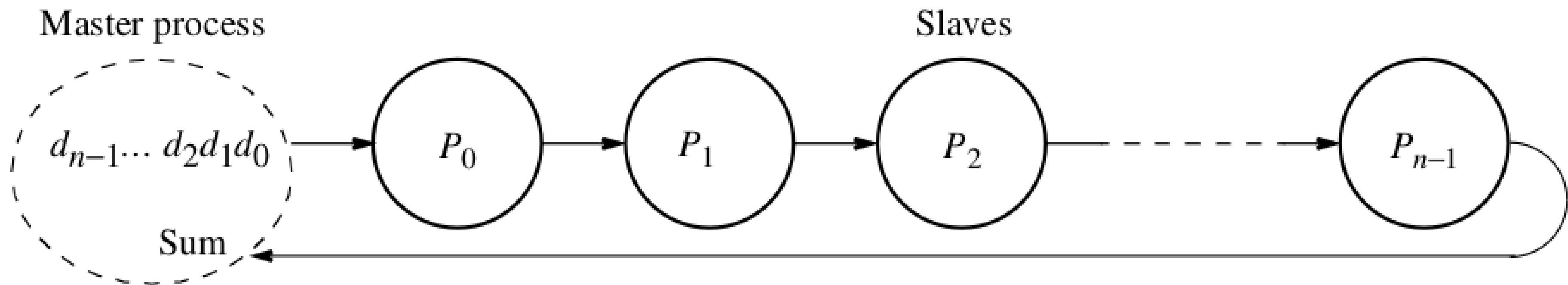
SPMD program

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

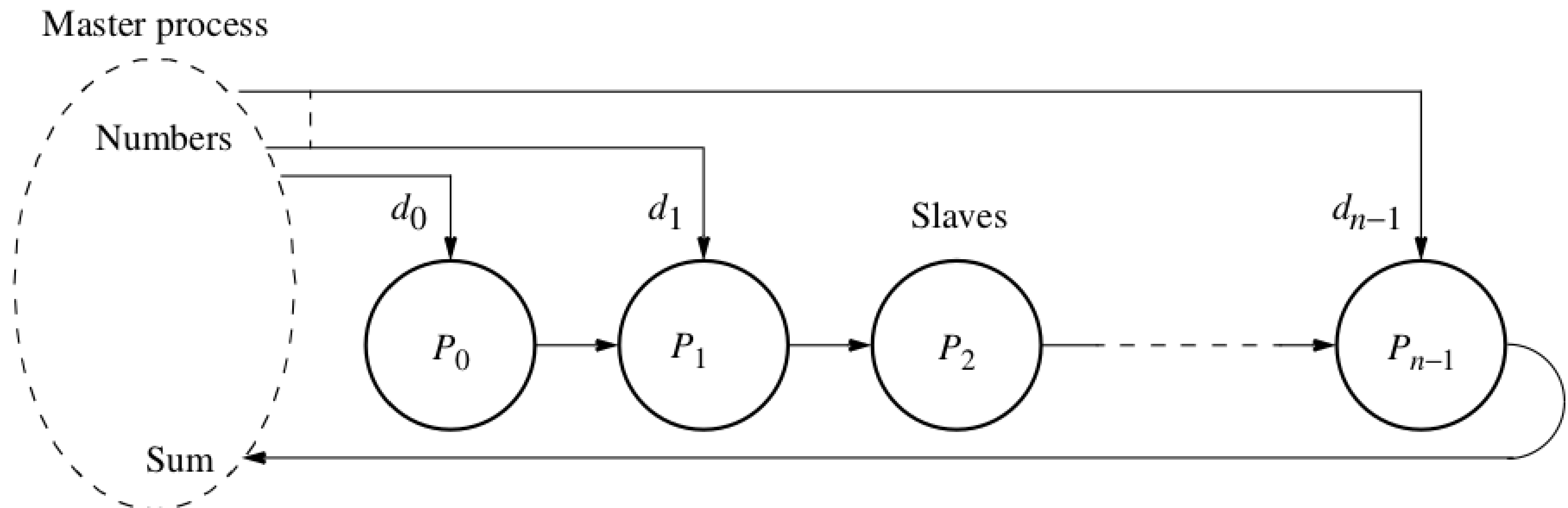
The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

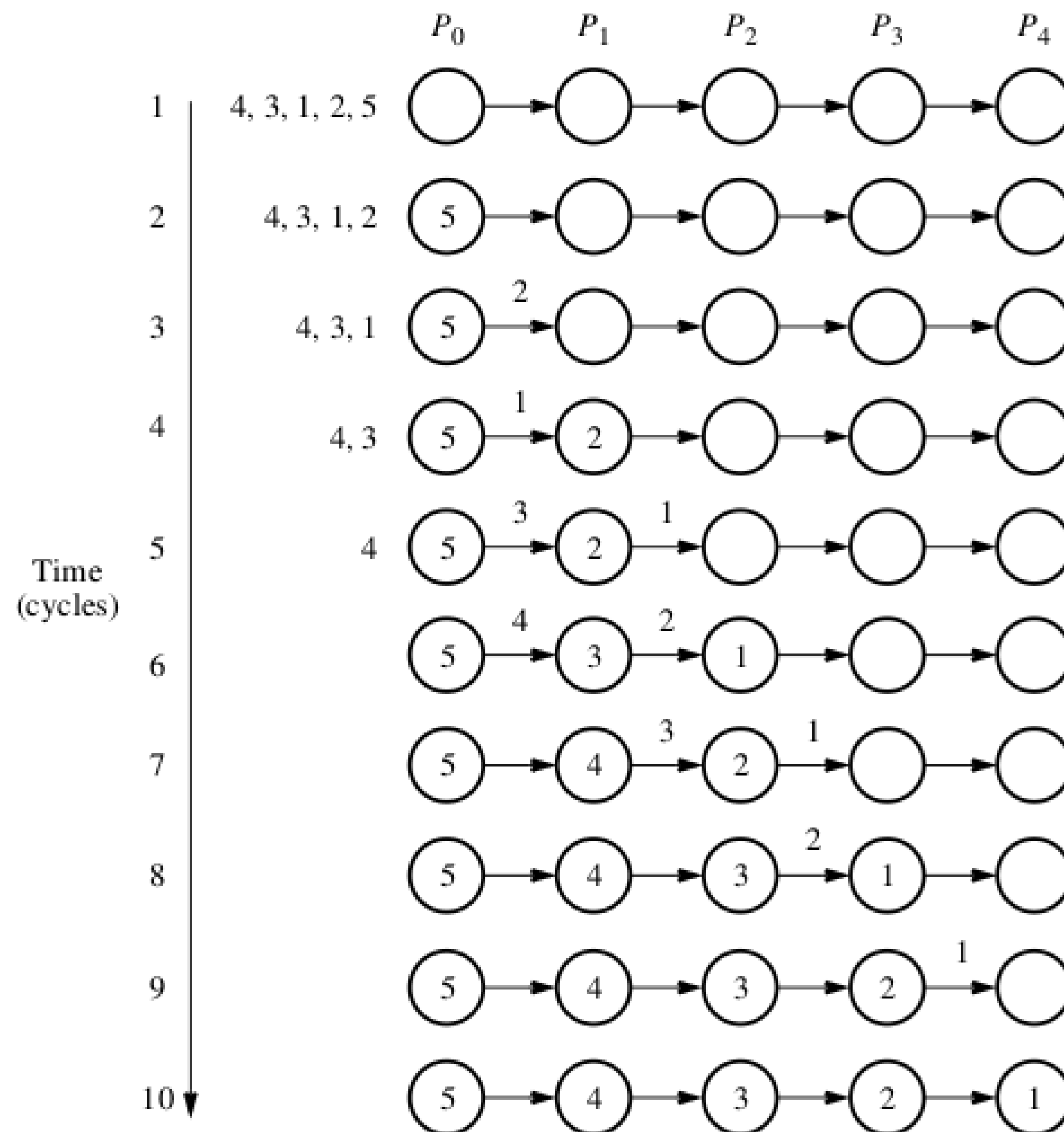
Pipelined addition numbers with a master process and ring configuration.



Pipelined addition of numbers with direct access to slave processes.

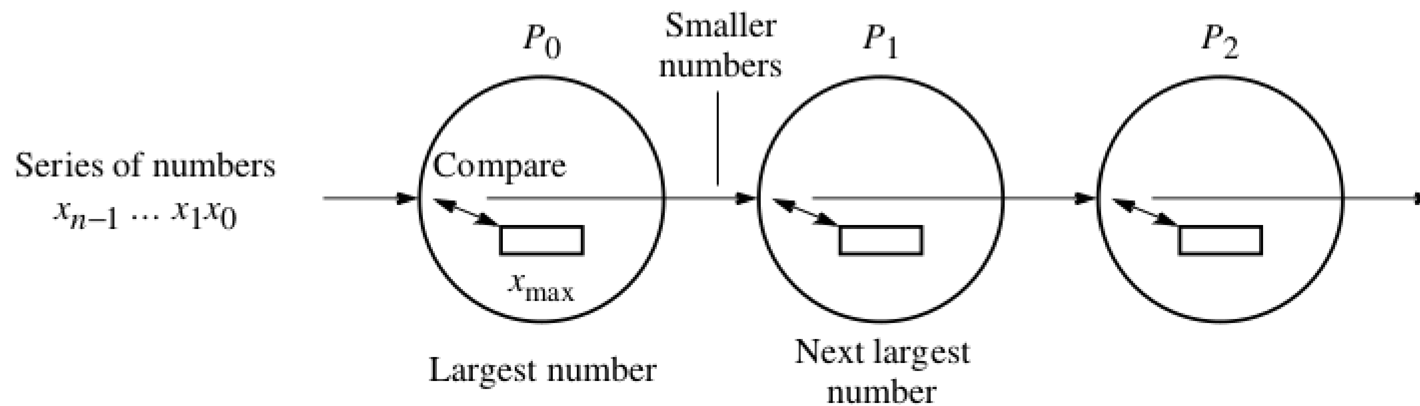


Sorting numbers – Insertion sort



The basic algorithm for process P_i is

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



Pipeline for sorting using insertion sort.

The assumption here: n numbers with n processors. Basic work unit is 1 compare&exchange

Analysis

Sequential

$$t_s = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Obviously very poor sequential sorting algorithm and unsuitable except for very small n.

Parallel

Each pipeline cycle requires at least

$$t_{comp} = 1$$

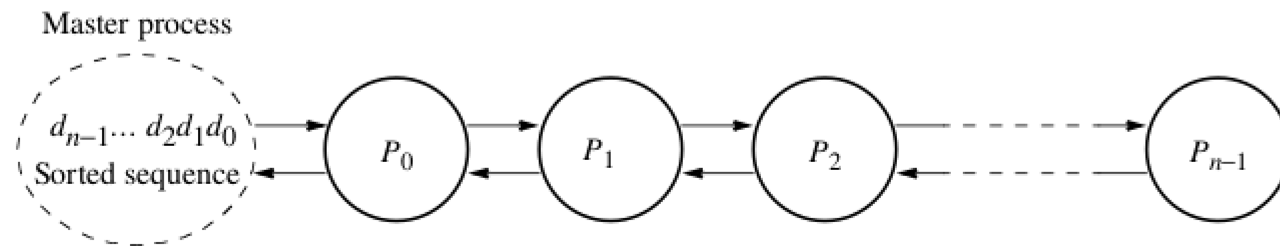
$$t_{comm} = 2(t_{startup} + t_{data})$$

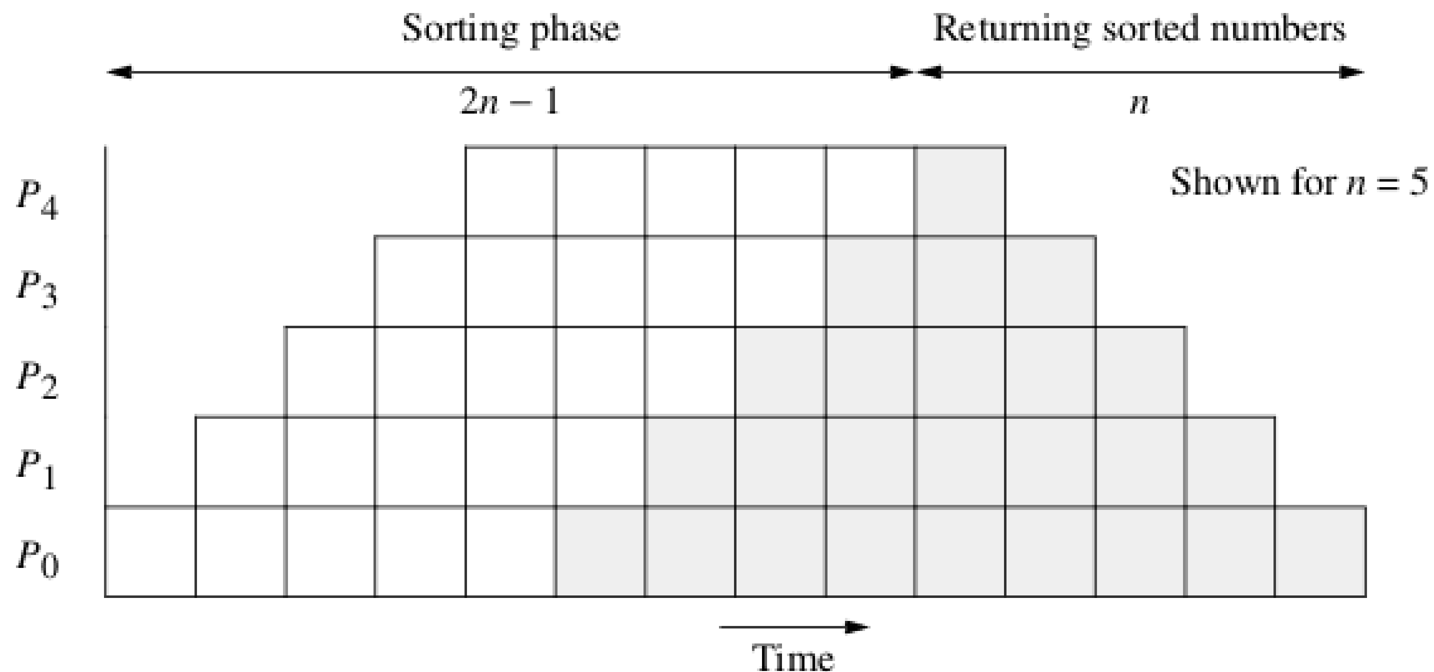
The total execution time, t_{total}, is given by

$$t_{total} = (t_{comp} + t_{comm})(2n-1) = (1 + 2(t_{startup} + t_{data}))(2n-1)$$

See next slide

Insertion sort with result returned





Solving a System of Linear Equations — Special Case

Type 3 example - process can continue with useful work after passing on information.

To solve system of linear equations of the so-called *upper-triangular* form:

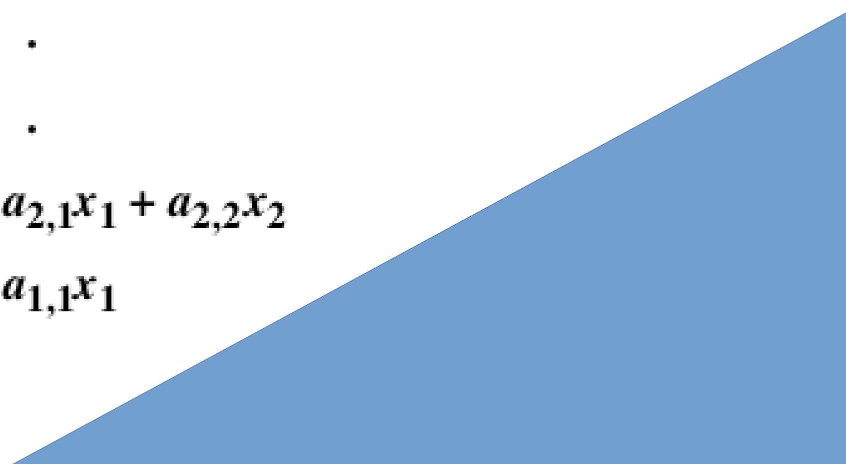
$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \ldots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

$$\vdots$$

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

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

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



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.,

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

The value obtained for x_0 is substituted into the next equation to obtain x_1; i.e.,

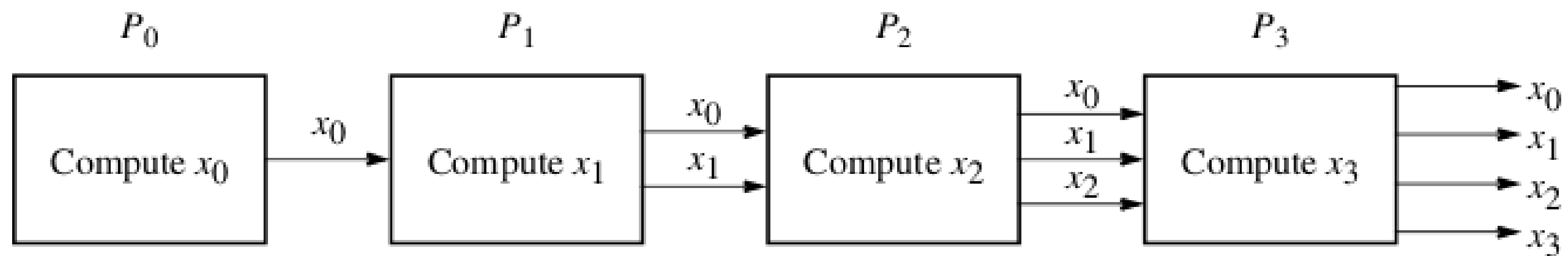
$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

The values obtained for x_1 and x_0 are substituted into the 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.

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.



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

```
x[0] = b[0]/a[0][0];      /* x[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

SPMD

The pseudocode of process P_i $(1 < i < n)$ of one pipelined version could be

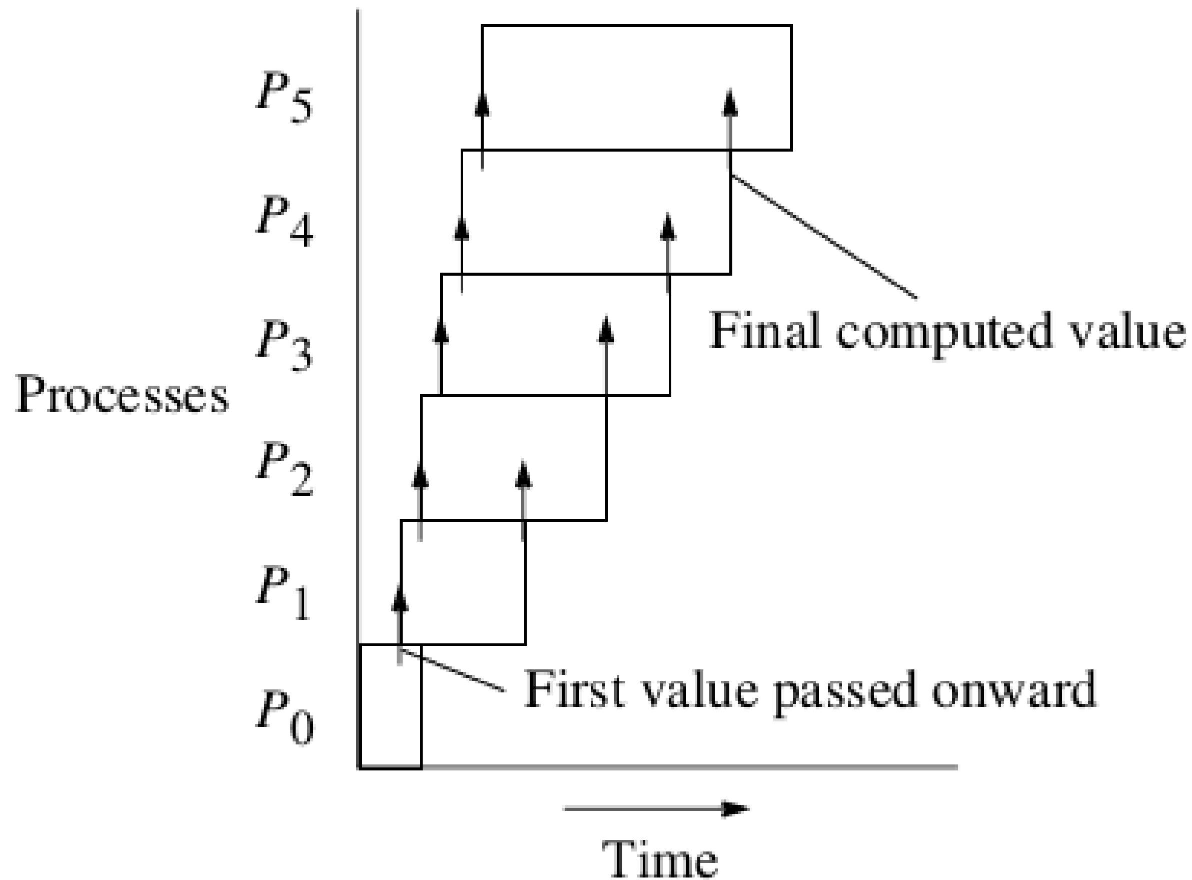
```
for (j = 0; j < i; j++) {
  recv(&x[j], P_{i-1});
  send(&x[j], P_{i+1});
}
sum = 0;
for (j = 0; j < i; j++)
  sum = sum + a[i][j]*x[j];
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], P_{i+1});
```

תקשורת: העברה
משמאל לימין

תקשורת: העברת
ה- x האחרון ימינה

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

Pipeline processing using back substitution



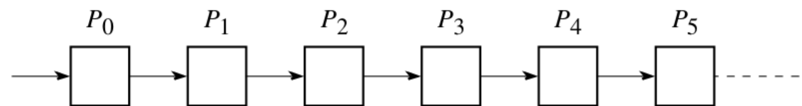
Pipelined Computations

Based on chapter 5 from Wilkinson & Allen

In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other.

In fact, this is the basis of sequential programming.

Each task will be executed by a separate process or processor.



Example

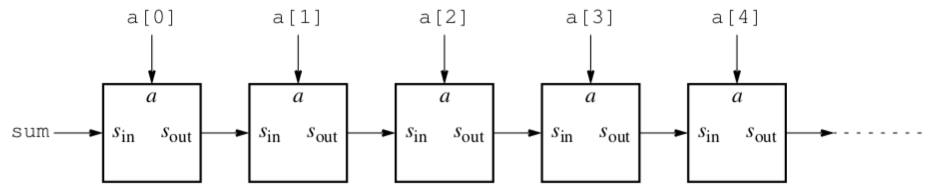
Add all the elements of array `a` to an accumulating sum:

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

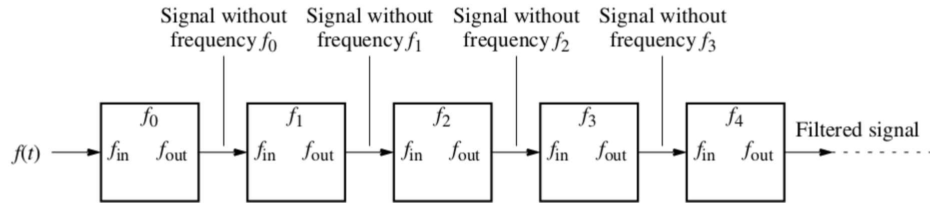
The loop could be “unfolded” to yield

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

Unfolded loop



Frequency Filter

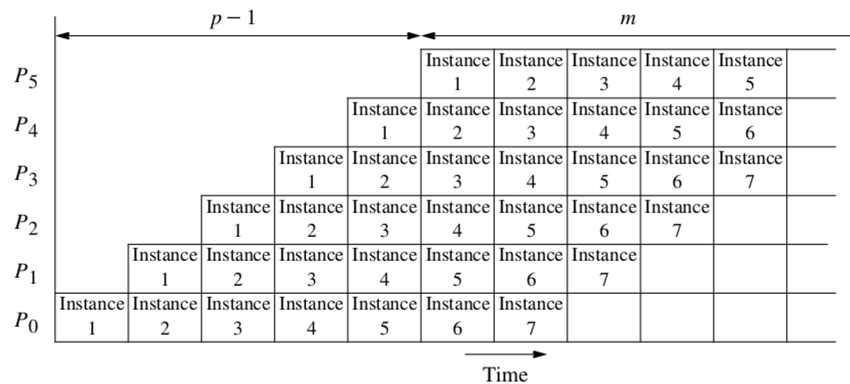


Conditions for a potential increased speed

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

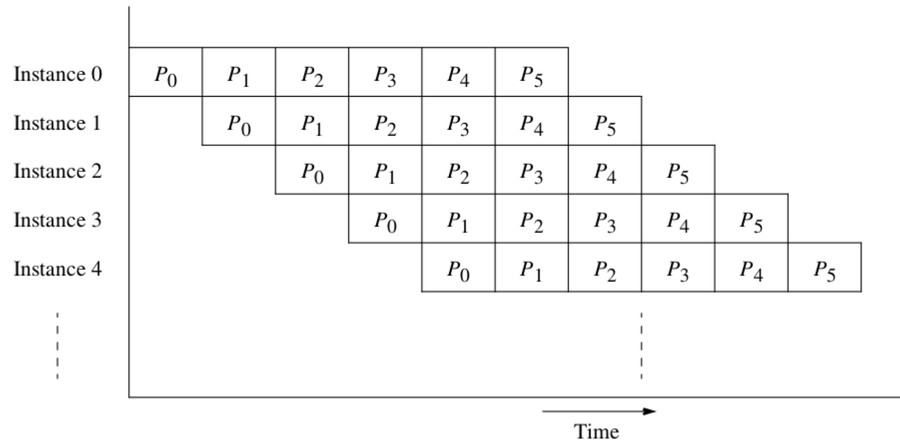
1. If more than one instance of the complete problem is to be executed
2. If a series of data items must be processed, each requiring multiple operations
3. If information to start the next process can be passed forward before the process has completed all its internal operations

“Type 1” Pipeline Space-Time Diagram

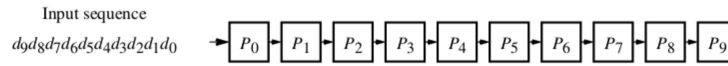


בסוג הראשון ההוראות (או עותק התכנית) זורם בצינור.
בסוג השני (עוד 2 שקפים) הנתונים זורמים בצינור.

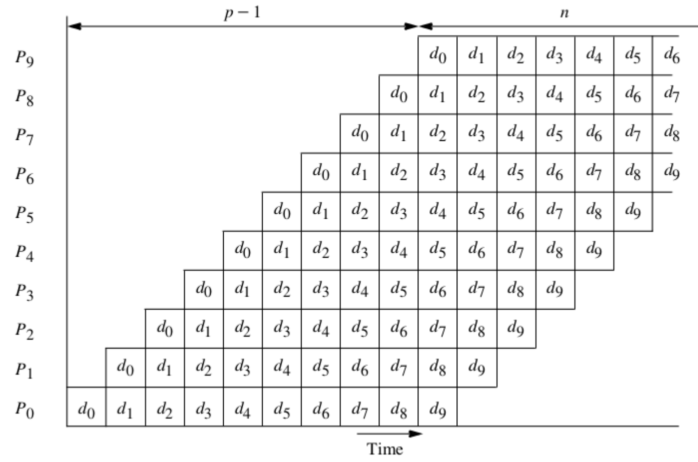
Alternative space-time diagram



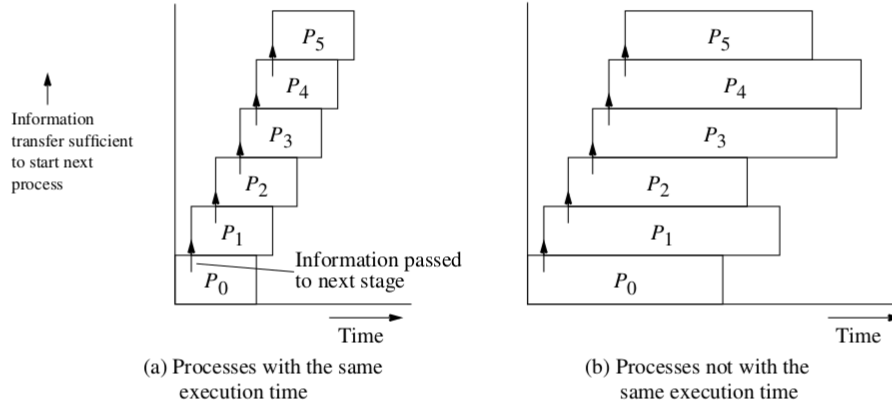
"Type 2"



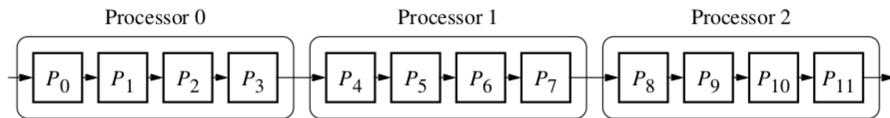
(a) Pipeline structure



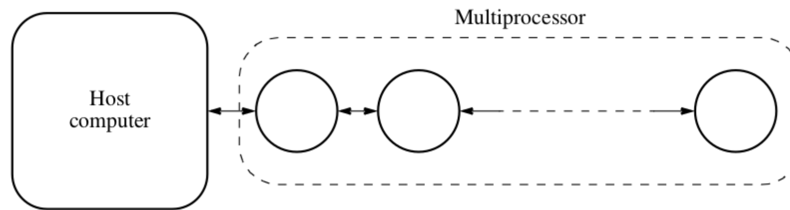
“Type 3”



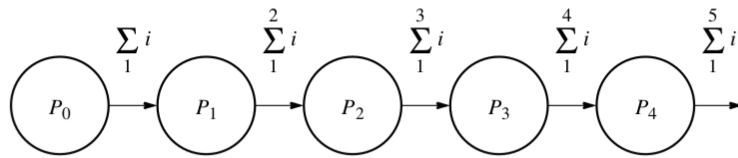
Partitioning processes onto processors



Computing platform for pipelined applications



Example: adding numbers



The basic code for process P_i :

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

except for the first process, P_0, which is

```
send(&number, P_1);
```

and the last process, P_{n-1}, which is

```
recv(&number, P_{n-2});
accumulation = accumulation + number;
```

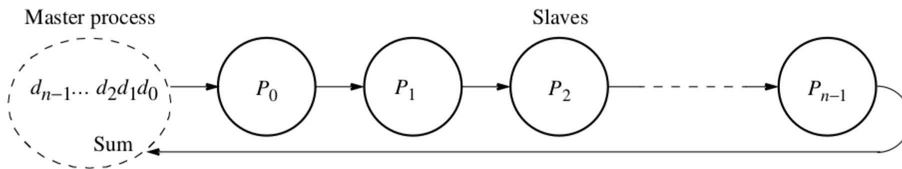
SPMD program

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

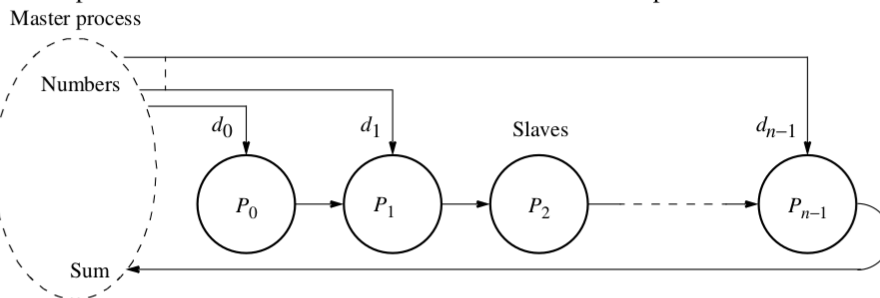
The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

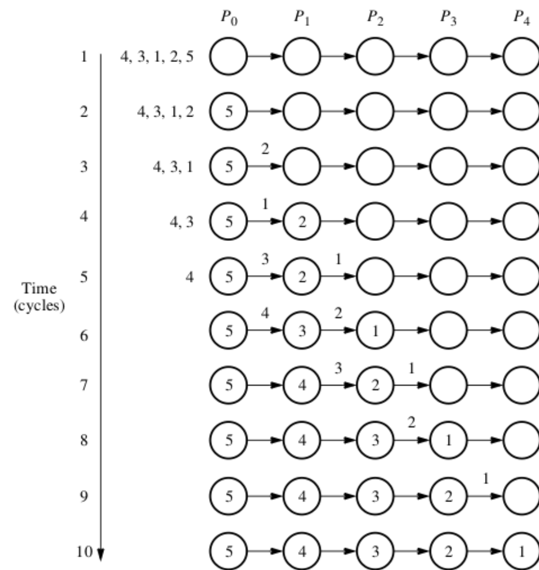
Pipelined addition numbers with a master process and ring configuration.



Pipelined addition of numbers with direct access to slave processes.

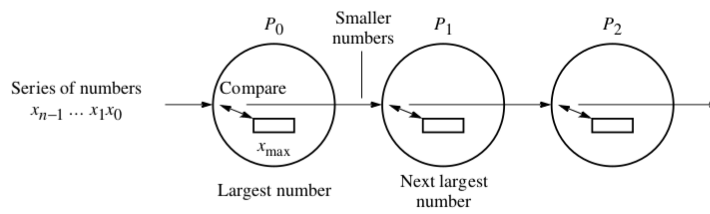


Sorting numbers – Insertion sort



The basic algorithm for process P_i is

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



Pipeline for sorting using insertion sort.

The assumption here:
n numbers with n processors.
Basic work unit is 1 compare&exchange

Analysis

Sequential

$$t_s = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Obviously very poor sequential sorting algorithm and unsuitable except for very small n.

Parallel

Each pipeline cycle requires at least

$$t_{comp} = 1$$

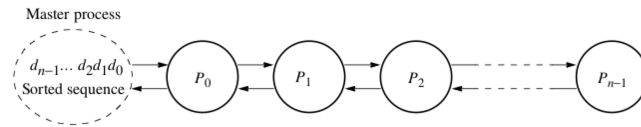
$$t_{comm} = 2(t_{startup} + t_{data})$$

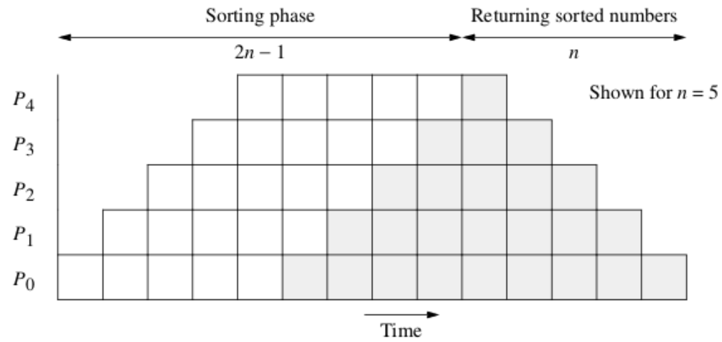
The total execution time, t_{total}, is given by

$$t_{total} = (t_{comp} + t_{comm})(2n - 1) = (1 + 2(t_{startup} + t_{data}))(2n - 1)$$

See next slide

Insertion sort with result returned





Solving a System of Linear Equations — Special Case

Type 3 example - process can continue with useful work after passing on information.

To solve system of linear equations of the so-called *upper-triangular* form:

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

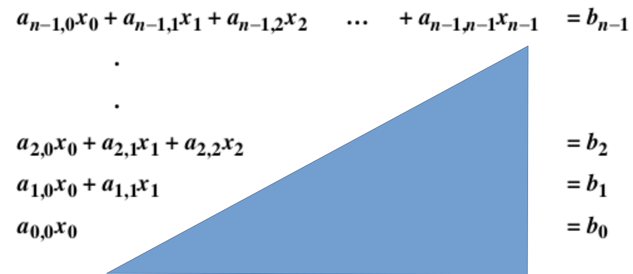
$$.$$

$$.$$

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

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

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



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

Back Substitution.

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

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

The value obtained for x_0 is substituted into the next equation to obtain x_1; i.e.,

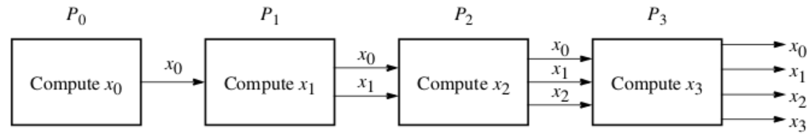
$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

The values obtained for x_1 and x_0 are substituted into the 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.

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.



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

```
x[0] = b[0]/a[0][0];      /* x[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

SPMD

The pseudocode of process P_i $(1 < i < n)$ of one pipelined version could be

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

תקשורת: העברה משמאל לימין

תקשורת: העברת ה- x האחרון ימינה

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

Pipeline processing using back substitution