UiT

THE ARCTIC
UNIVERSITY
OF NORWAY

Lecture 6: Pipelined Computations

Parallell Programming (INF-3201)
University of Tromsø
Autumn 2013

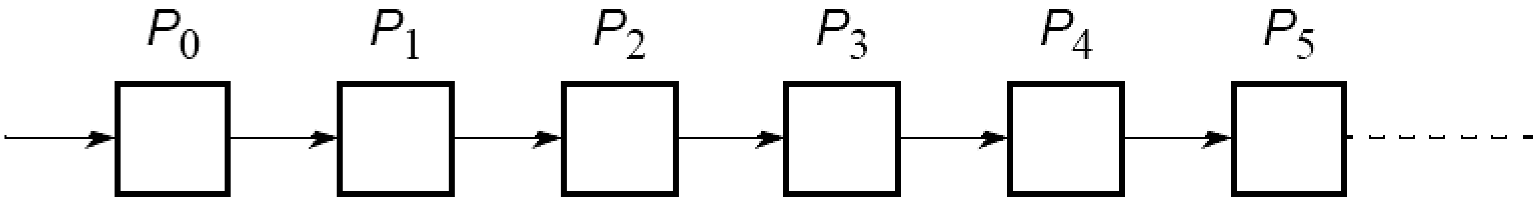
John Markus Bjørndalen

Outline

- Pipeline technique
- Examples
  - Adding numbers
  - Sorting numbers
  - Prime number generation
  - Solving a System of Linear Equations

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

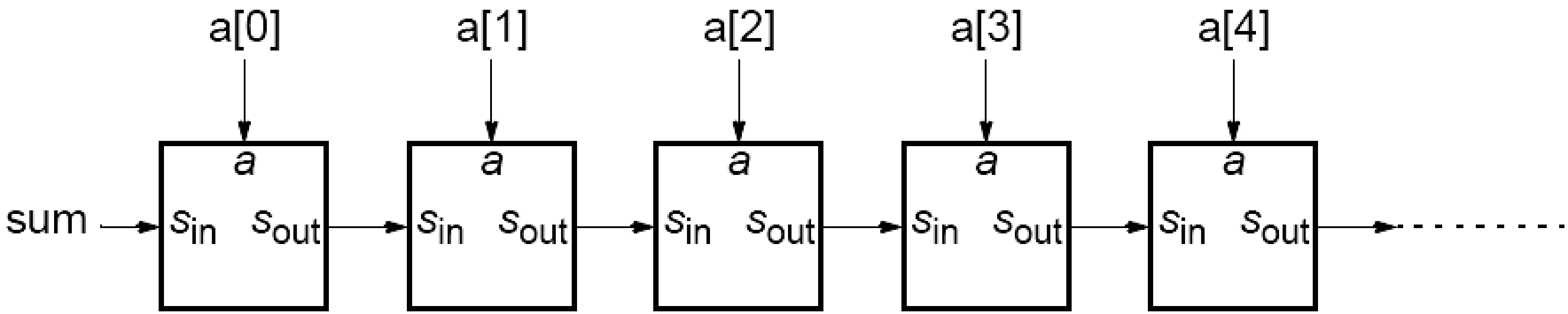
Add all the elements of array **a** to an accumulating sum:

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

The loop could be “unfolded” to yield

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

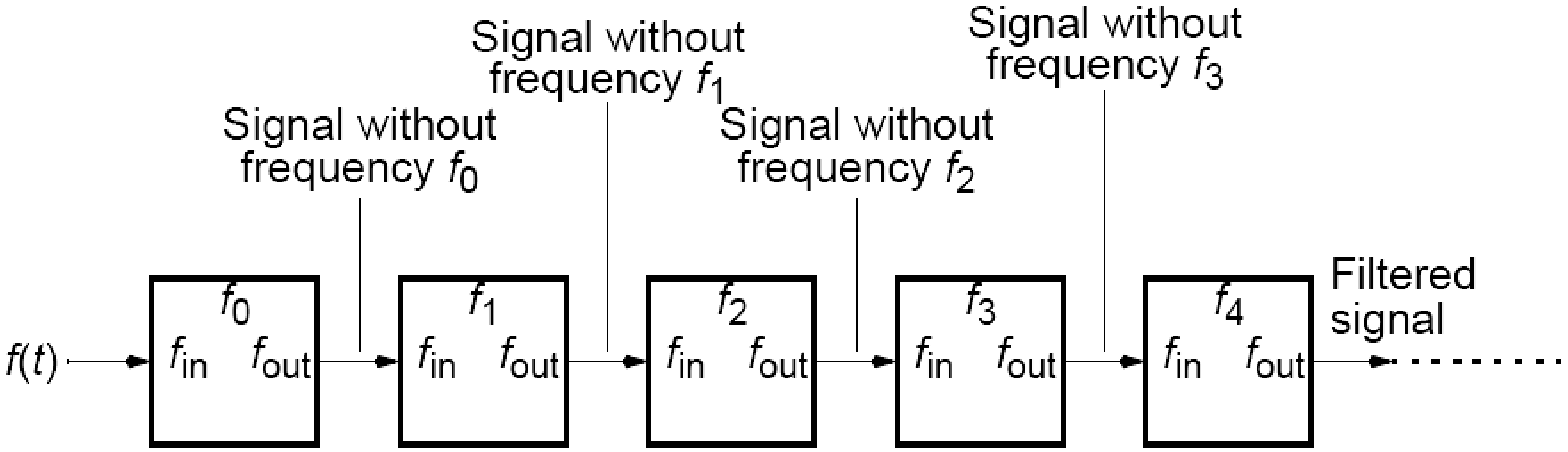
Pipeline for an unfolded loop



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:

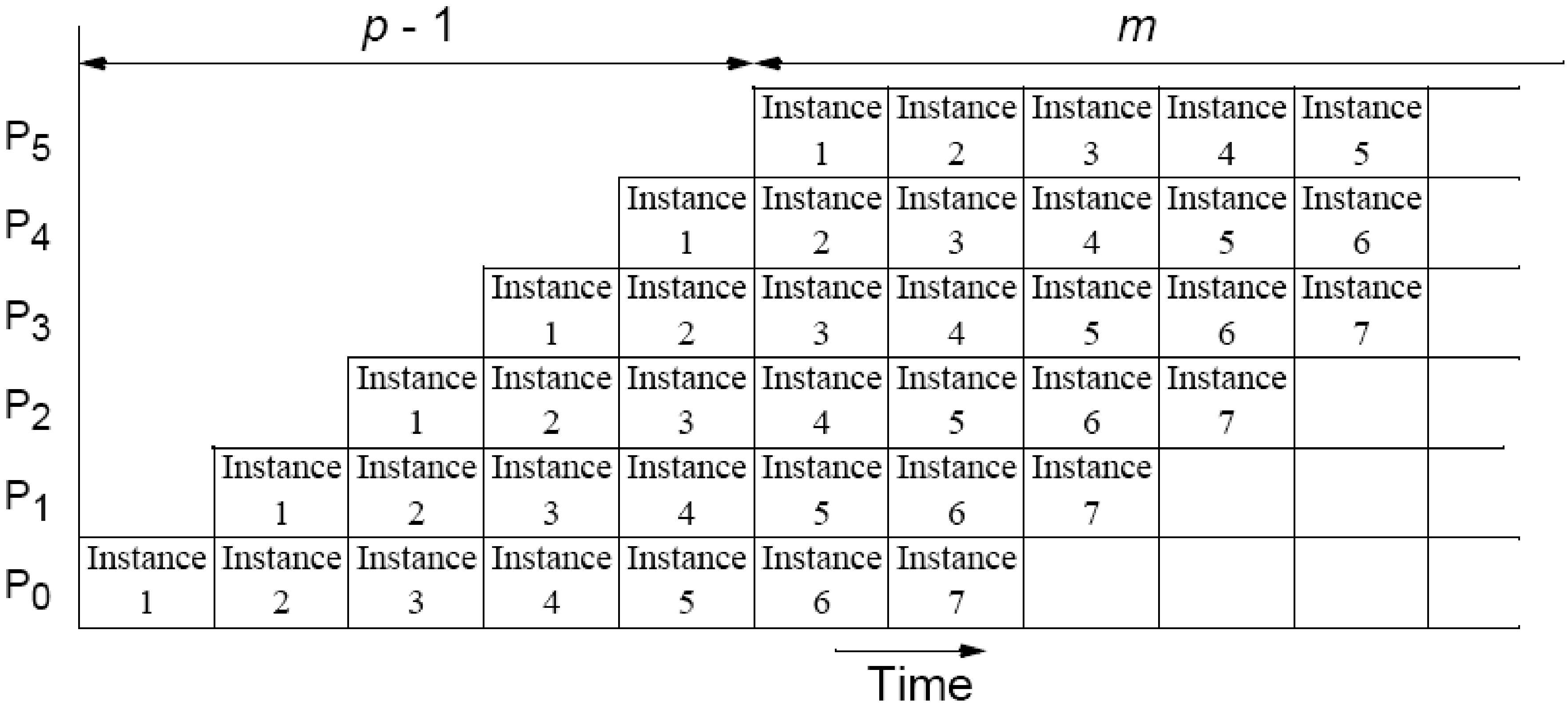


Where can pipelining be used

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

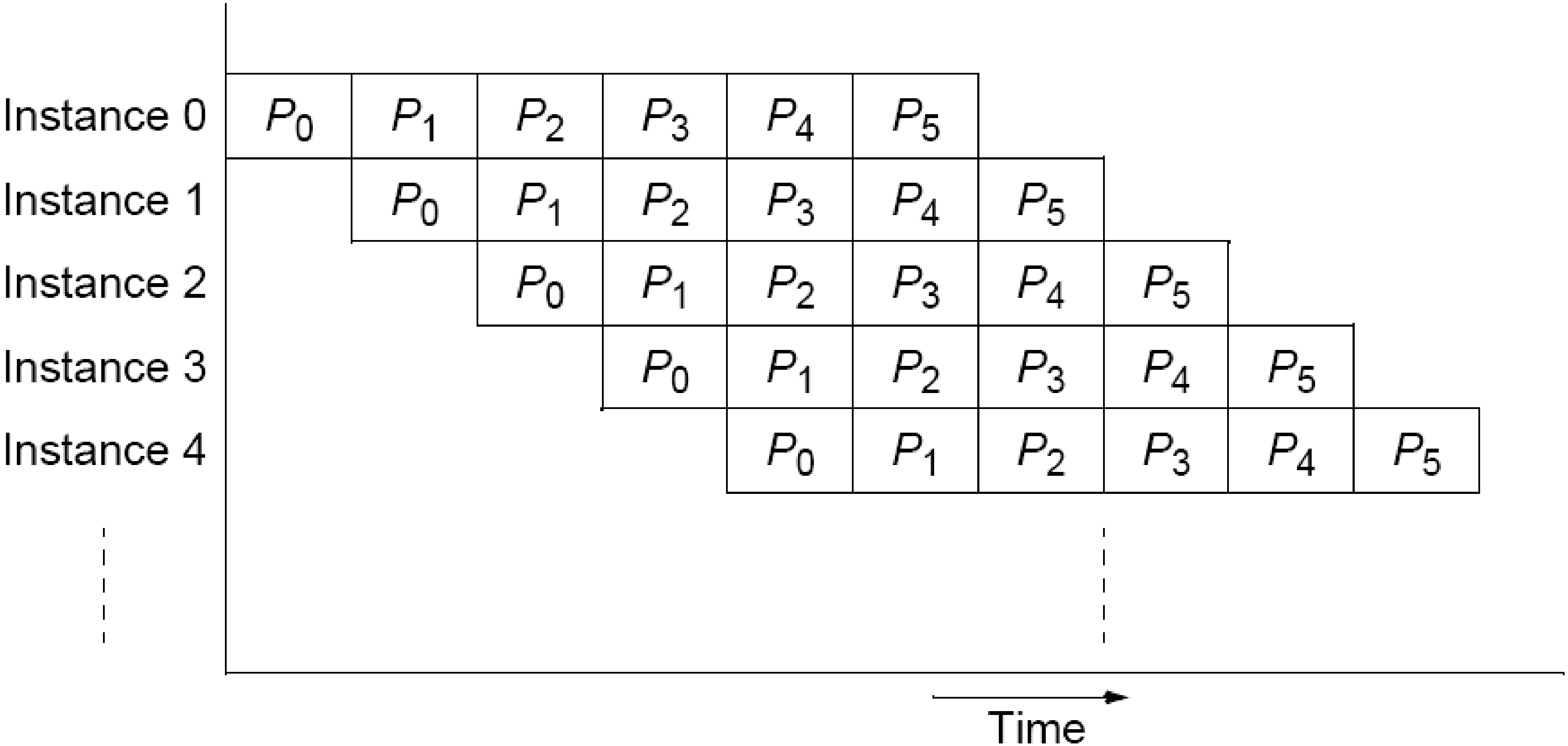
“Type 1” Pipeline Space-Time Diagram



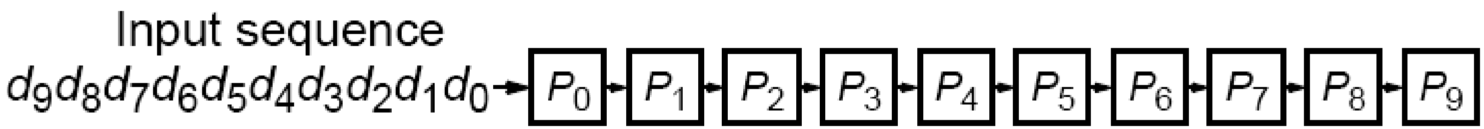
p processes (or stages), m instances:

- average #cycle per instance $t_a = (m+p-1)/m$. $t_a \rightarrow 1$ for large m

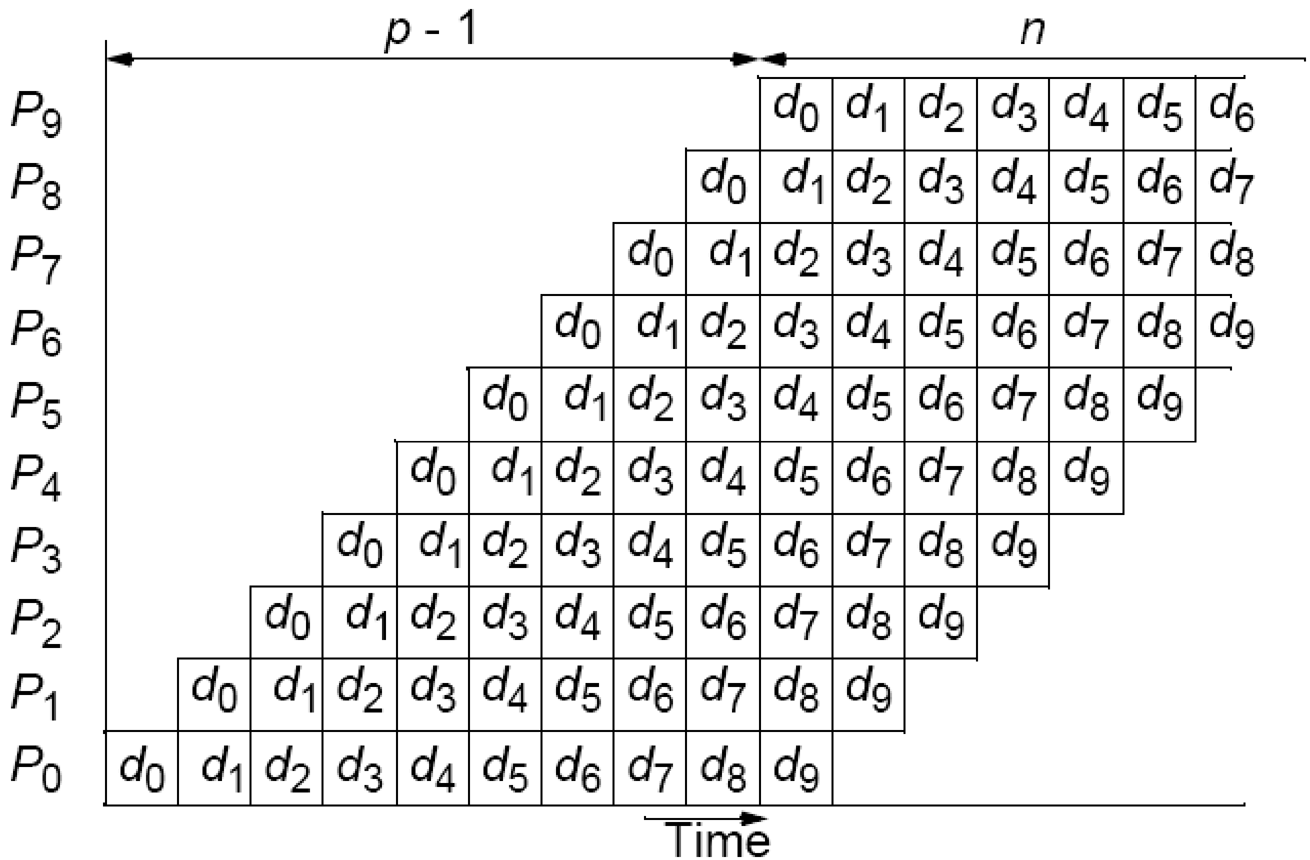
Alternative space-time diagram



“Type 2” Pipeline Space-Time Diagram

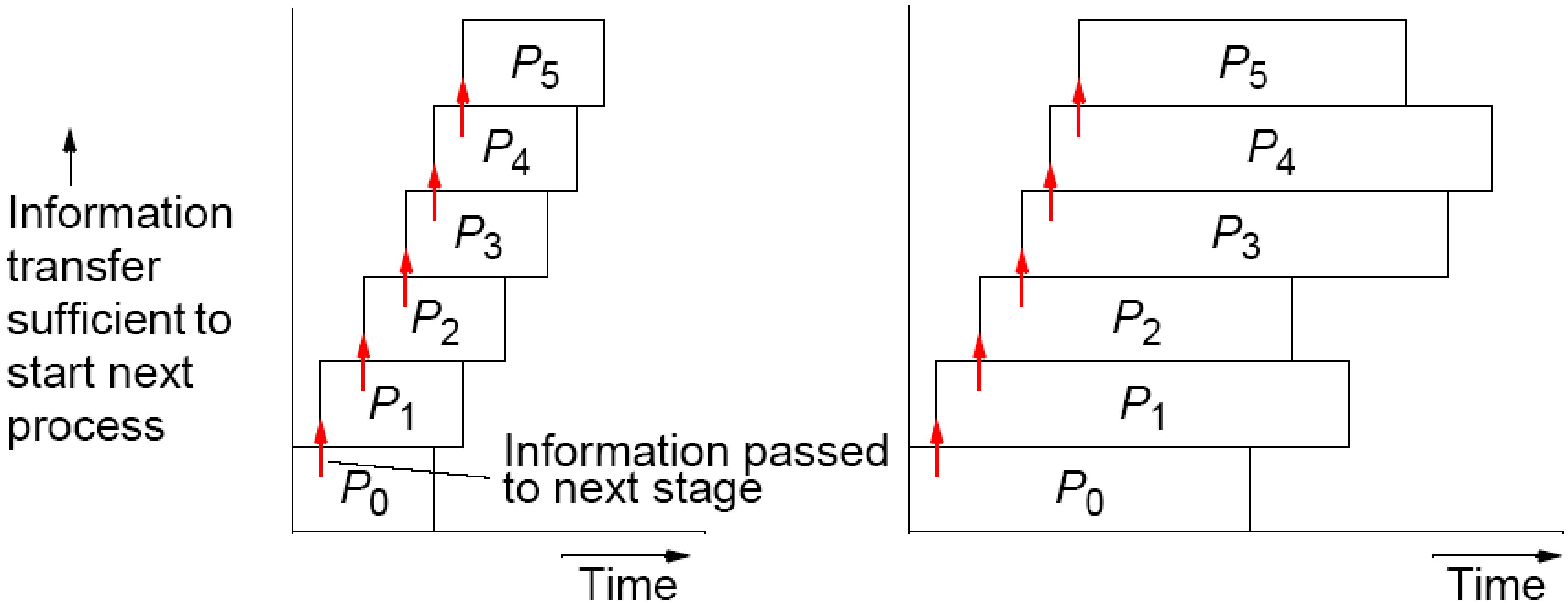


(a) Pipeline structure



(b) Timing diagram

“Type 3” Pipeline Space-Time Diagram

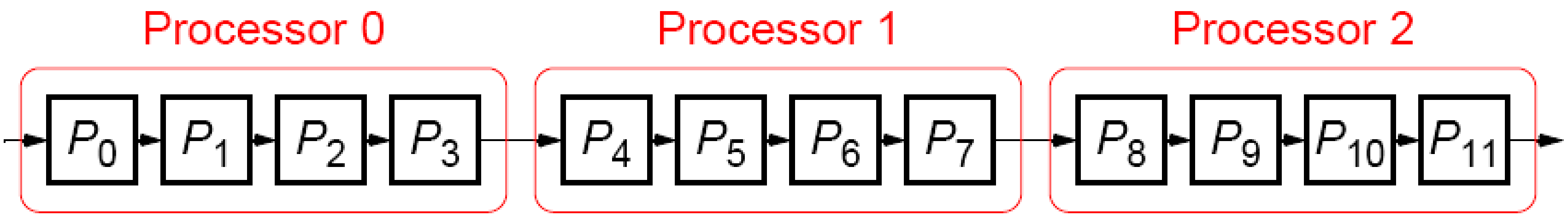


(a) Processes with the same execution time

(b) Processes not with the same execution time

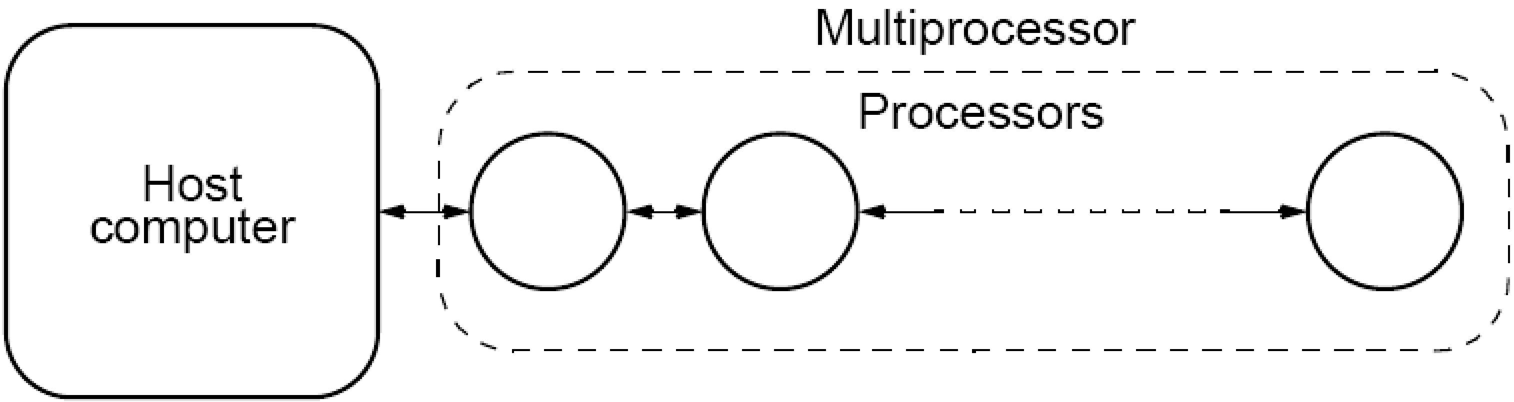
Pipeline processing where information passes to next stage before previous state completed.

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

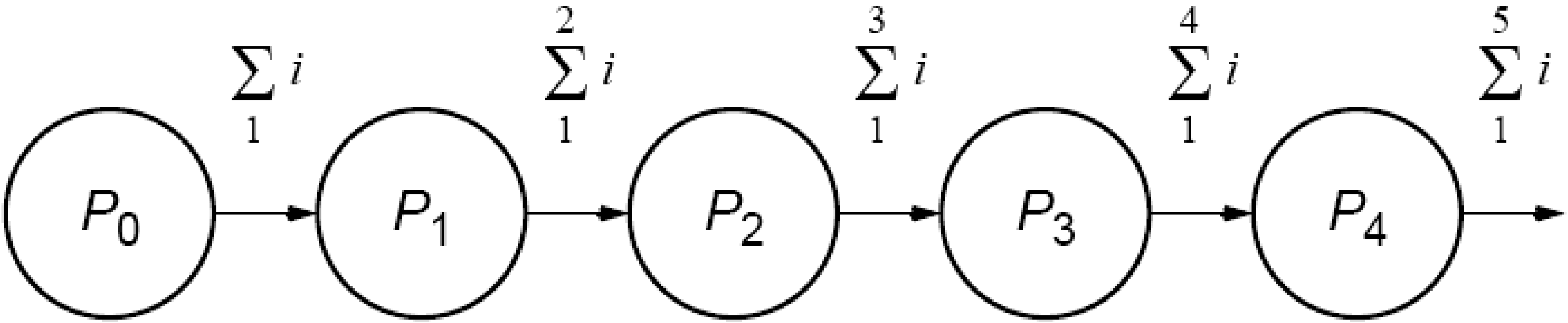


Outline

- Pipeline technique
- Examples
  - Adding numbers
  - Sorting numbers
  - Prime number generation
  - Solving a System of Linear Equations

Pipeline Program Examples

Adding Numbers



Type 1 pipeline computation

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

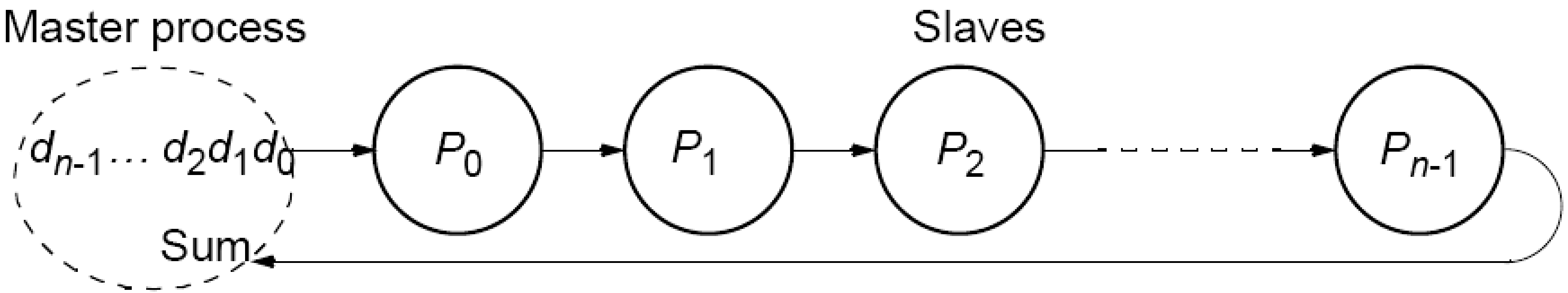
```
if (process > 0) {
   recv(&accumulation, Pi-1);
   accumulation = accumulation + number;
}
if (process < n-1)
   send(&accumulation, Pi+1);
```

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

Pipelined addition numbers

Master process and ring configuration

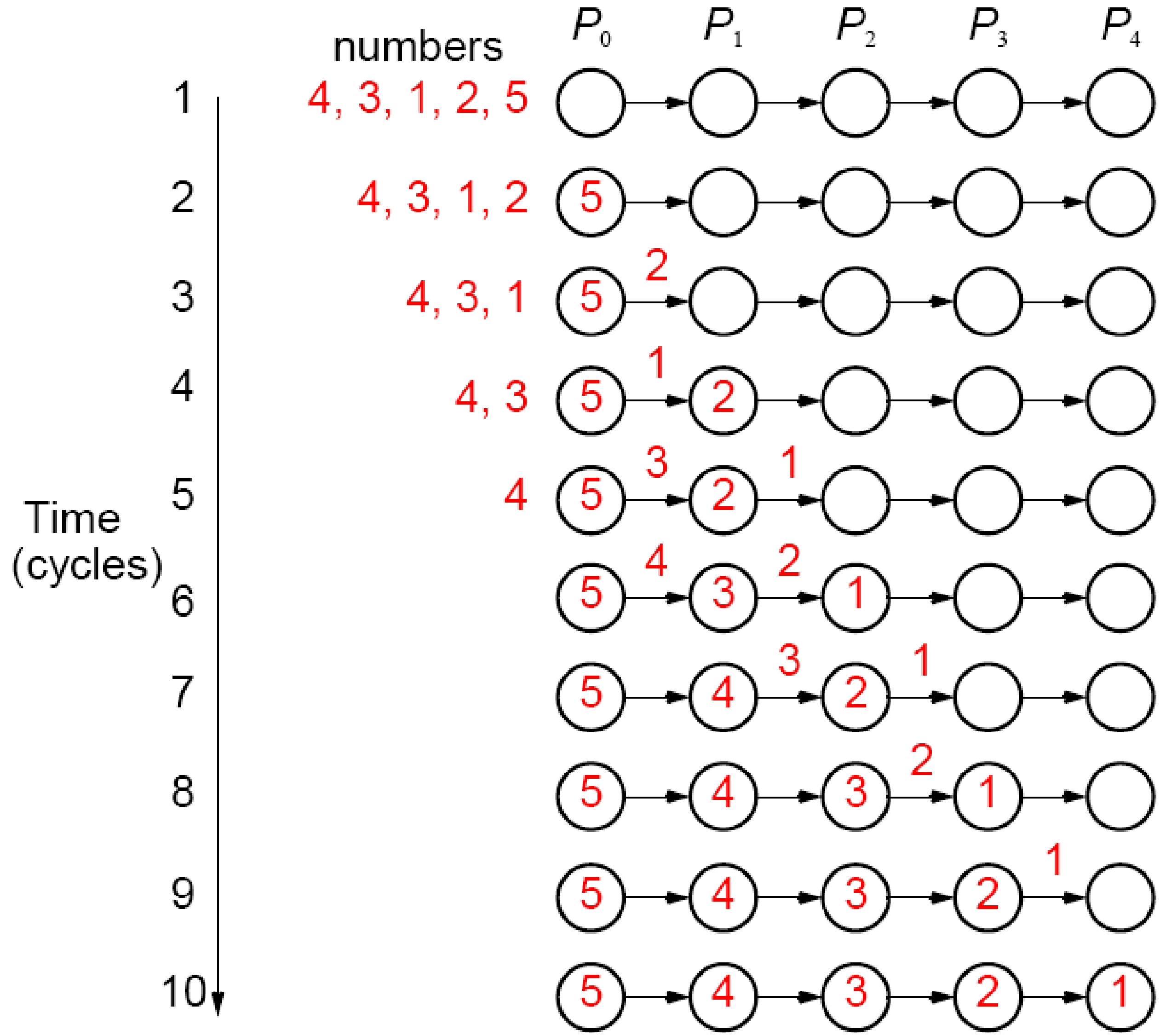


Outline

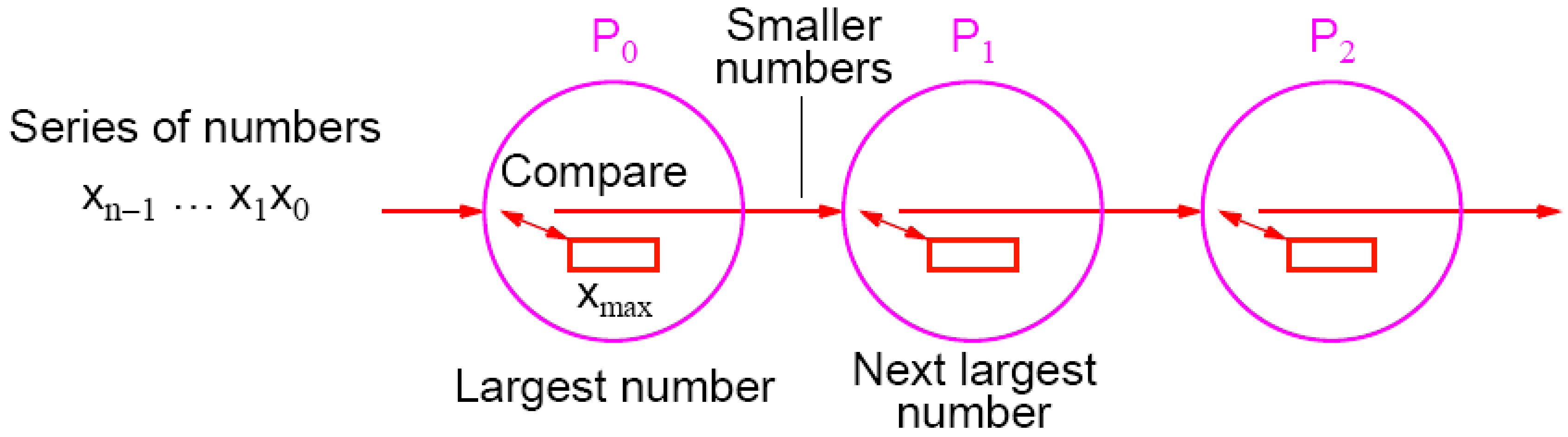
- Pipeline technique
- Examples
  - Adding numbers
  - Sorting numbers
  - Prime number generation
  - Solving a System of Linear Equations

Sorting Numbers

A parallel version of *insertion sort*.



Pipeline for sorting using insertion sort



Type 2 pipeline computation

The basic algorithm for process P_i is

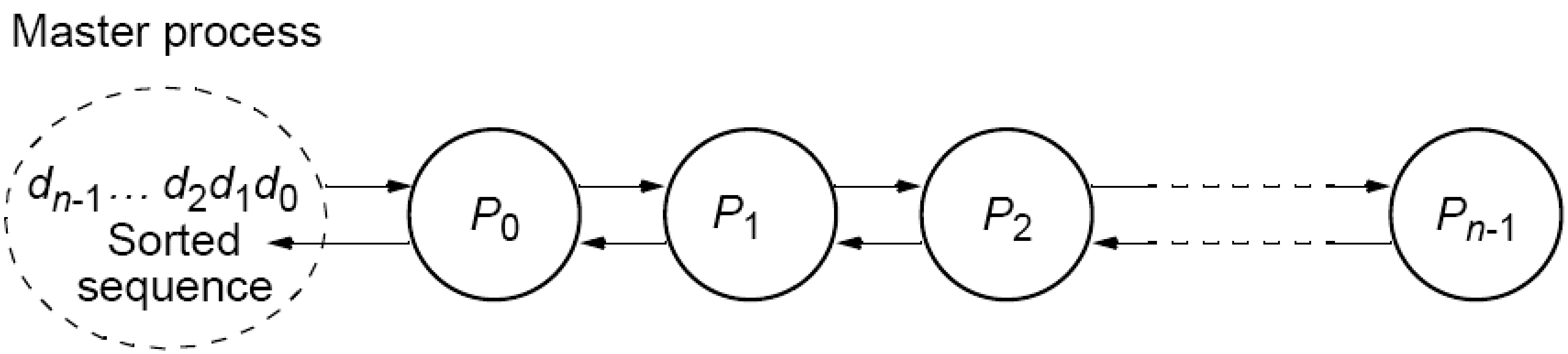
```
recv(&number, P_{i-1});
if (number > x) {
   send(&x, P_{i+1});
   x = number;
} else {
  send(&number, P_{i+1});
}
```

With n numbers, number P_i is to accept = n - i.

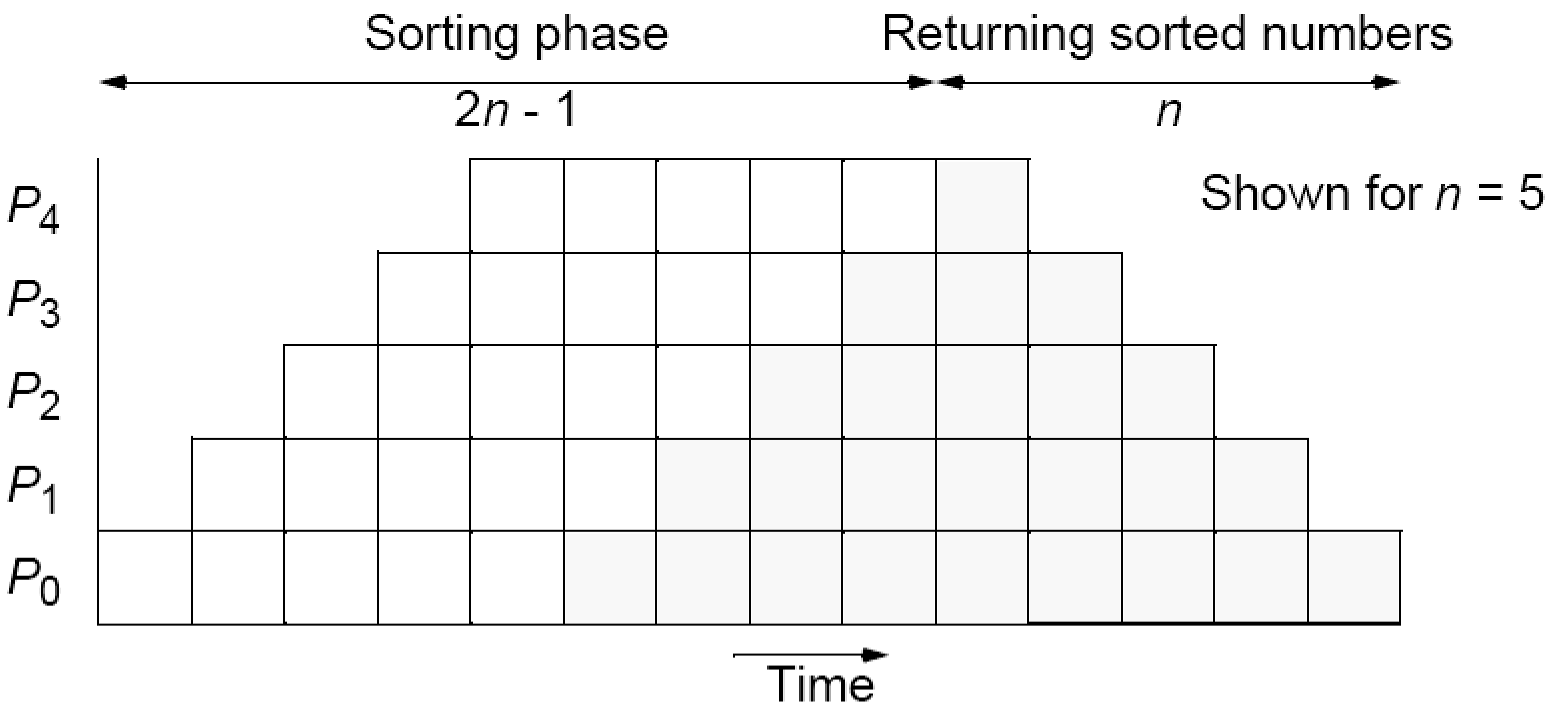
Number of passes onward = n - i - 1

Hence, a simple loop could be used.

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



Insertion sort with results returned



Analysis

- Assumption
  - Compare-and-exchange operation takes one step
- Sequential version
  - $t_s = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2$
  - Time complexity = $\Theta(n2)$
- Parallel version
  - For each pipeline cycle
    - $t_{comp} = 1$; $t_{comm} = 2(t_{startup} + t_{data})$
  - $T_{total} = (t_{comp} + t_{comm})(2n - 1)$

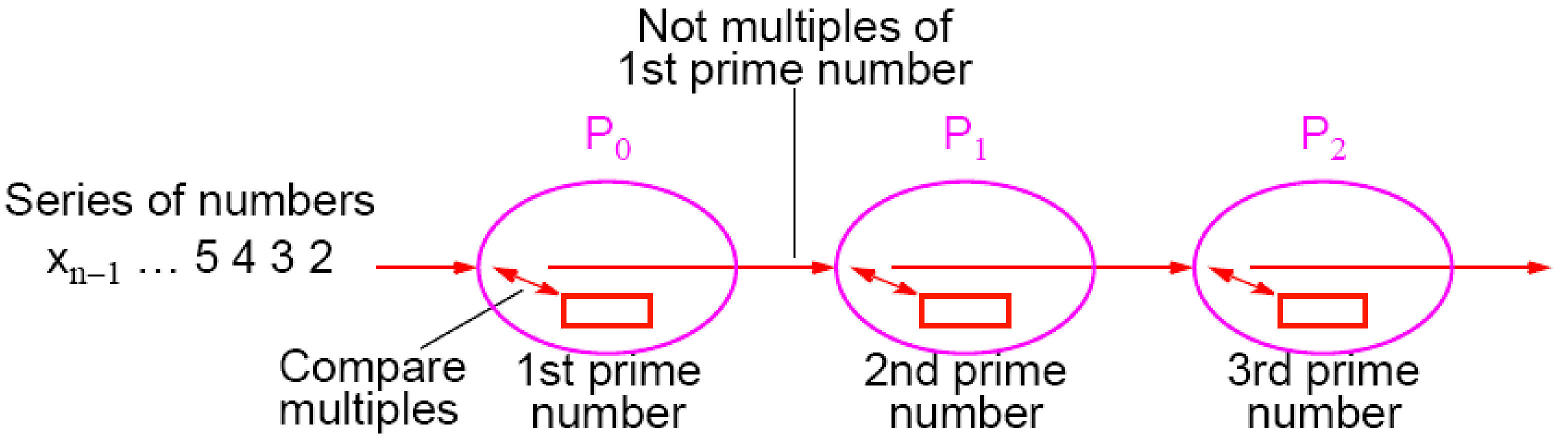
    $= (1 + 2(t_{startup} + t_{data}))(2n - 1)$
  - Time complexity = $\Theta(n)$

Outline

- Pipeline technique
- Examples
  - Adding numbers
  - Sorting numbers
  - Prime number generation
  - Solving a System of Linear Equations

Prime Number Generation
Sieve of Eratosthenes

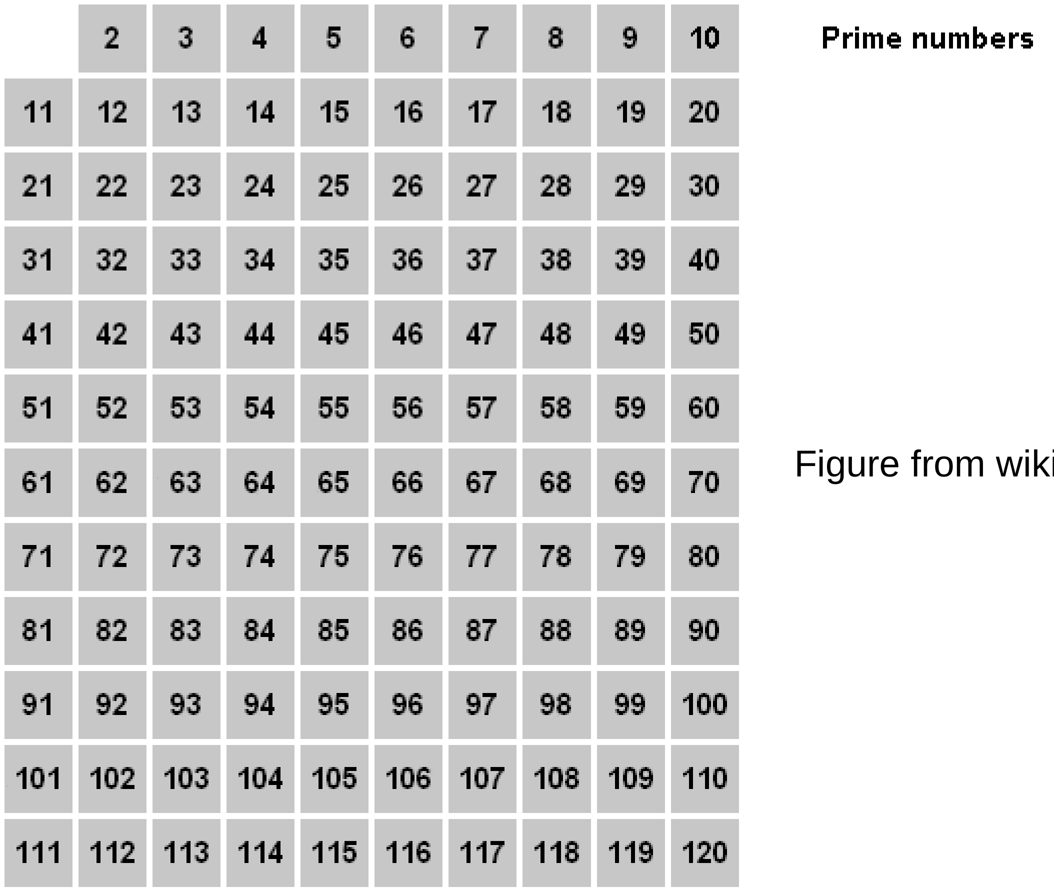
- Series of all integers generated from 2.
- First number, 2, is prime and kept.
- All multiples of this number deleted as they cannot be prime.
- Process repeated with each remaining number.
- The algorithm removes non-primes, leaving only primes.



Type 2 pipeline computation

Illustration

Algorithm steps for primes below 120



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

Figure from wikipedia.org

In javascript:
http://www.hbmeyer.de/eratosiv.htm

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

```
recv(&x, P_{i-1});
/* repeat following for each number */
recv(&number, P_{i-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, P_{i-1});
for (i = 0; i < n; i++) {
   recv(&number, P_{i-1});
   if (number == terminator) break;
   if ((number % x) != 0) send(&number, P_{i+1});
}
```

Outline

- Pipeline technique
- Examples
  - Adding numbers
  - Sorting numbers
  - Prime number generation
  - Solving a System of Linear Equations

Solving a System of Linear Equations

Upper-triangular form

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

$$.$$

$$.$$

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

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

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

where *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.,

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

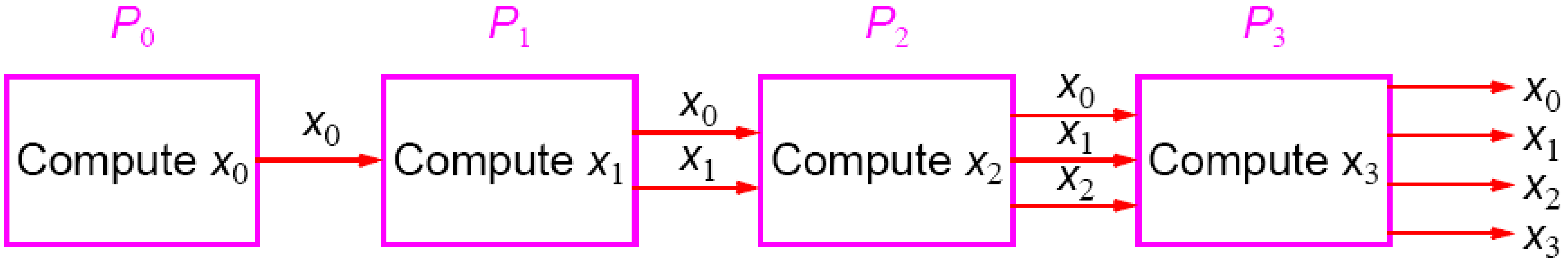
Values obtained for x_1 and x_0 substituted into next equation to obtain x_2:

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

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];
}
```

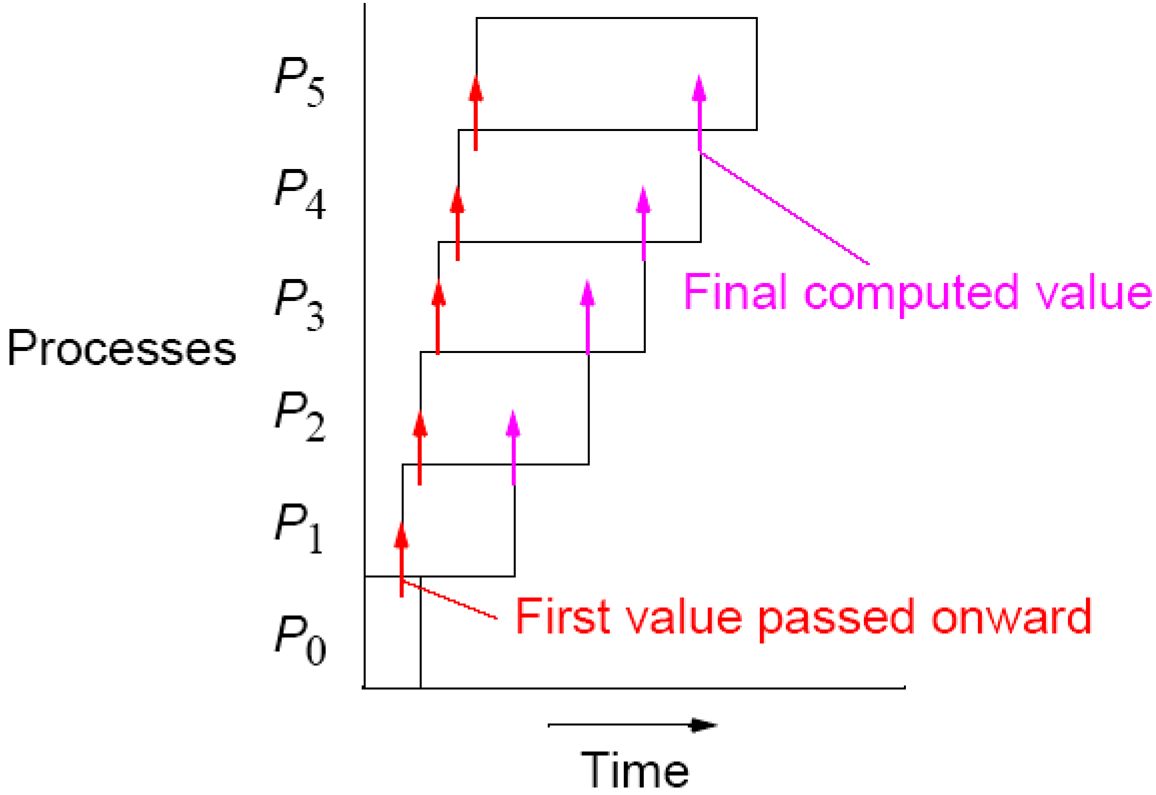
Parallel Code

Pseudocode of process P_i $(1 < i < n)$ 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});
```

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

Pipeline processing using back substitution



References

- Barry Wilkinson & Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.
- Blaise Barney, “Introduction to Parallel Computing”, Livermore Computing
- Algorithms in C, 3rd Edition, Parts 1-4 by Robert Sedgewick, Addison-Wesley, 1998. ISBN 0-201-31452-5