Lesson 5: Pipeline computations

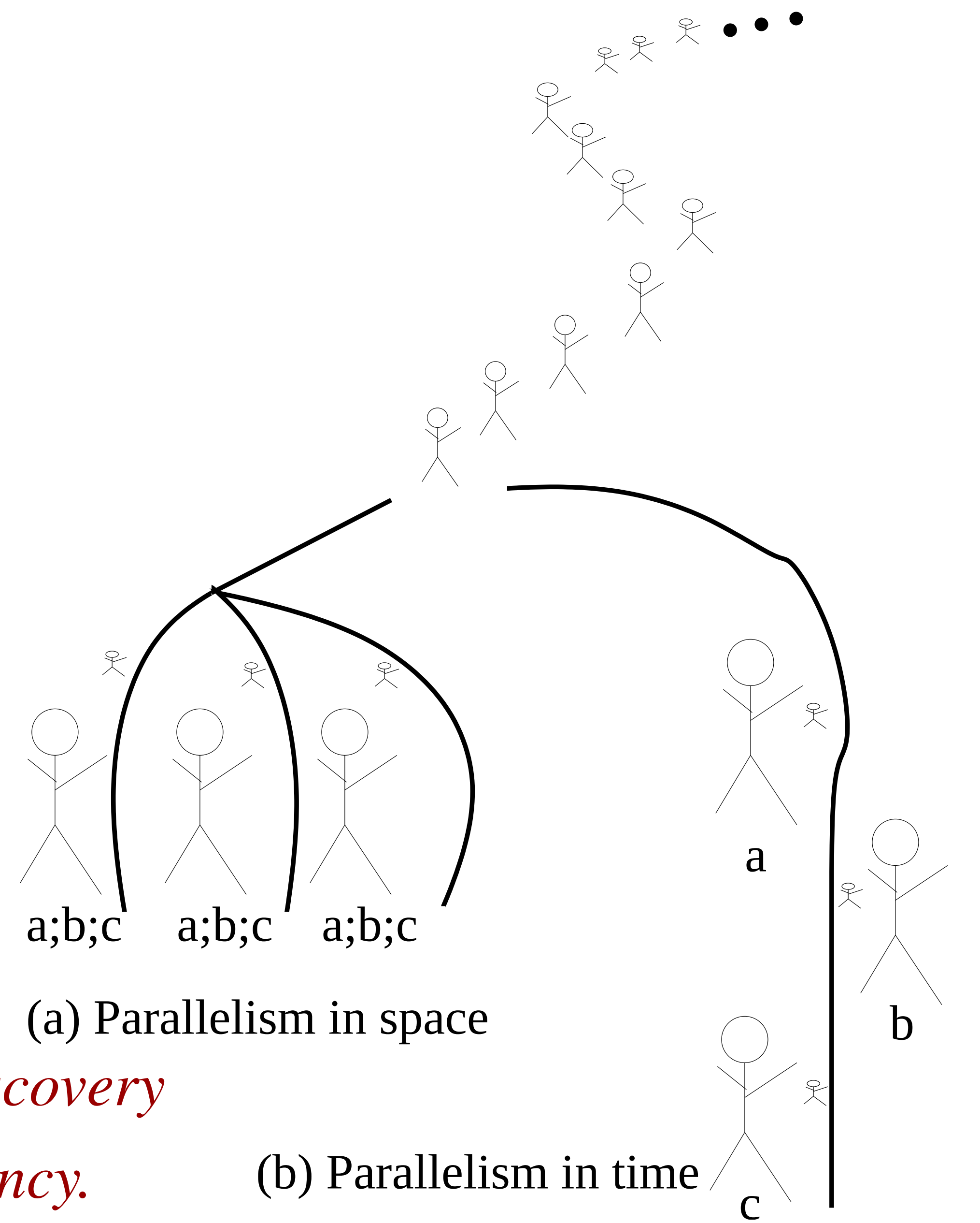
G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Parallelism in space and in time

One may identify two kinds of parallelism:
—*in space* and
—*in time*.

In the former case (a), the full sequence of operations `a;b;c;` is done by a single process, while in the latter case (b) one has specialized processes for each action `a`, `b`, and `c`.



(a) Parallelism in space

(b) Parallelism in time

Notice: Actually, (b) reflects Ford's discovery that specialization often increases efficiency.

Applications

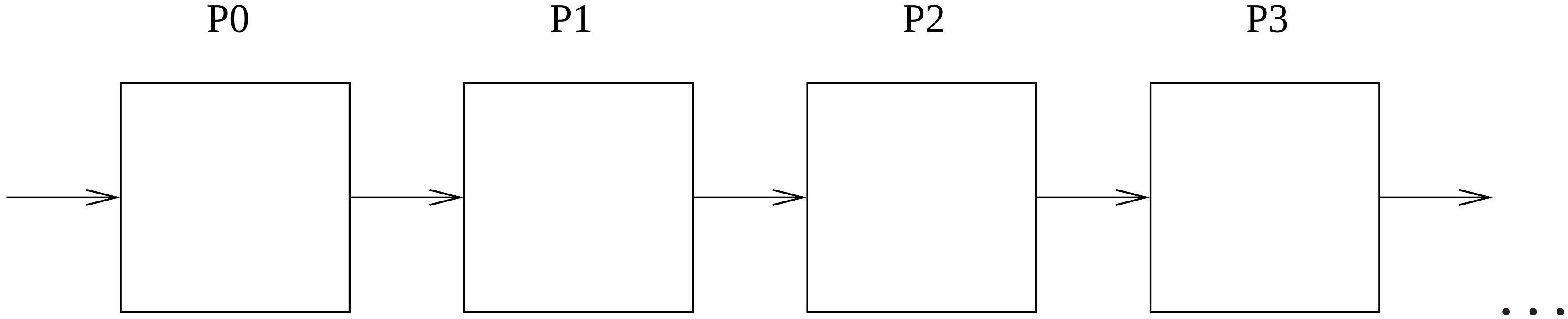
Applications:

- The "*pipeline*" approach was the main source of increasing *processor speed* in the last decade. (Currently, there is a shift to instruction-level parallelism in processor design.)
- It was also a key part in the design of *parallel data-flow architectures* aiming as an alternative to classical Von Neumann architecture.

Pipeline technique

In the *pipeline technique*,

- The problem is divided into *a series of tasks* that have to be completed *one after the other*. [Actually, this is the basis of sequential programming.]
- Then, *each task* will be executed by a *separate process* (or processor).



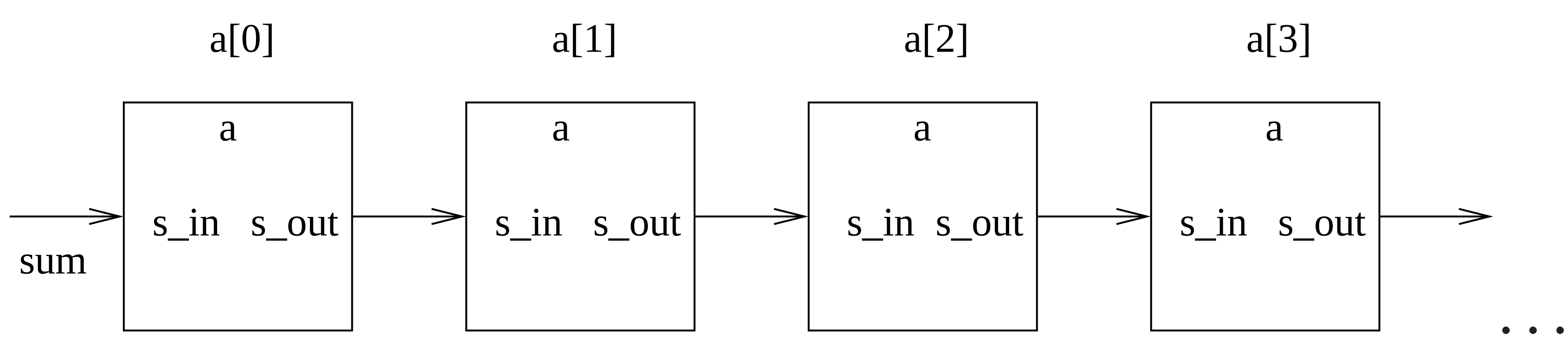
..Pipeline technique

Example 1 (adding numbers): Add all the elements of array `a`:

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

The loop may be *unfolded* to yield

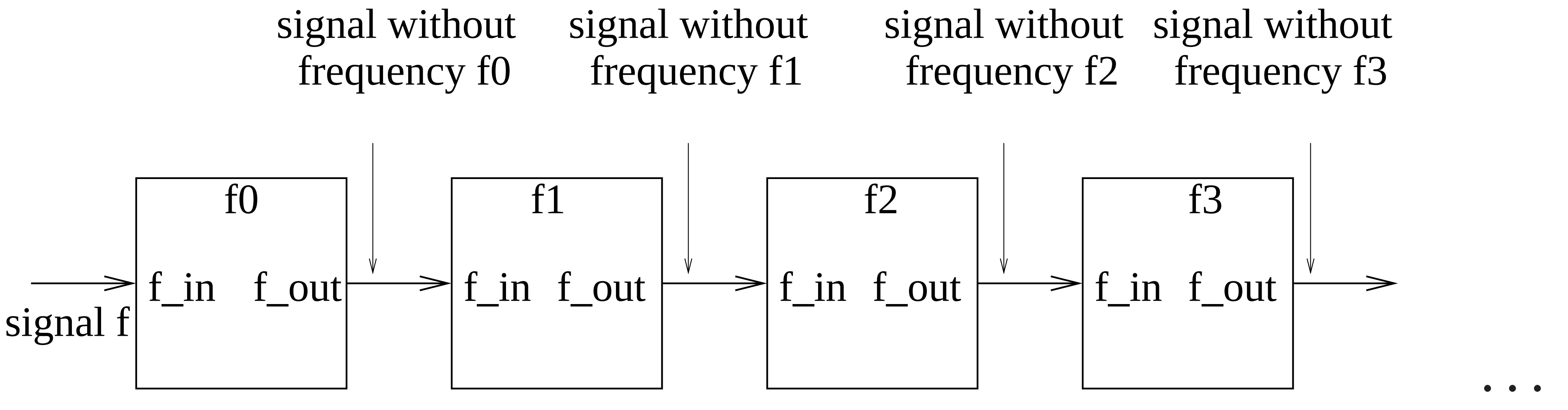
```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
⋮
```



..Pipeline technique

Example 2 (frequency filter):

- *Frequency filter*: The objective here is to remove specific frequencies, say $f0, f1, f2, f3, \ldots$ from a given digital signal $f(t)$.
- The pipeline is described in the figure below



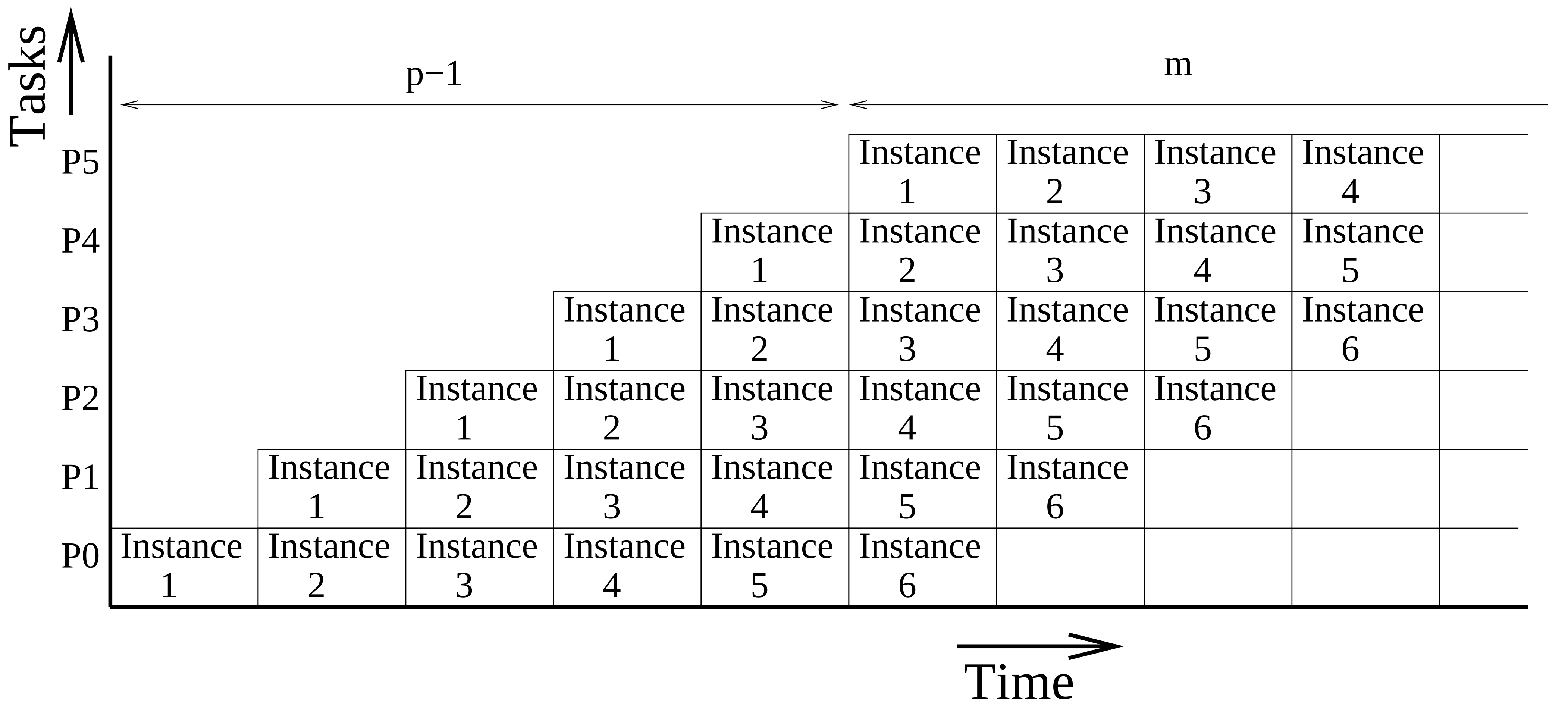
..Pipeline technique

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

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

Space-time diagrams

Type 1: Multiple instances of the same problem.



..Space-time diagrams

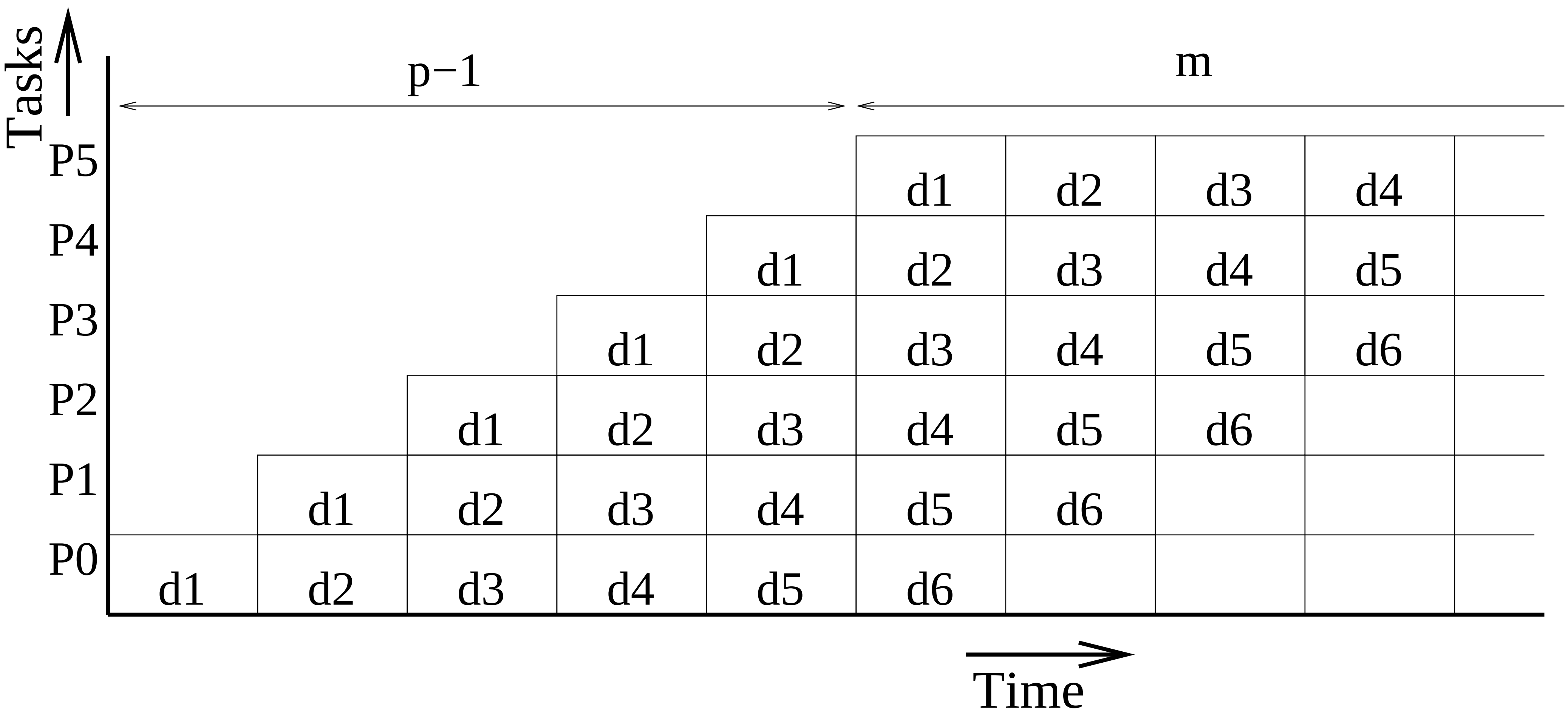
Type 1:

- mostly used in hardware design, particularly for processor design
- a staircase effect at the beginning; after that, one instance of the problem is completed at each pipeline cycle.
- $p-1$ is the *pipeline latency*

..Space-time diagrams

Type 2: Pipeline structure

`d6 d5 d4 d3 d2 d1 d0` $\rightarrow \boxed{P0} \rightarrow \boxed{P1} \rightarrow \boxed{P2} \rightarrow \boxed{P3} \rightarrow \boxed{P4} \rightarrow \boxed{P5}$



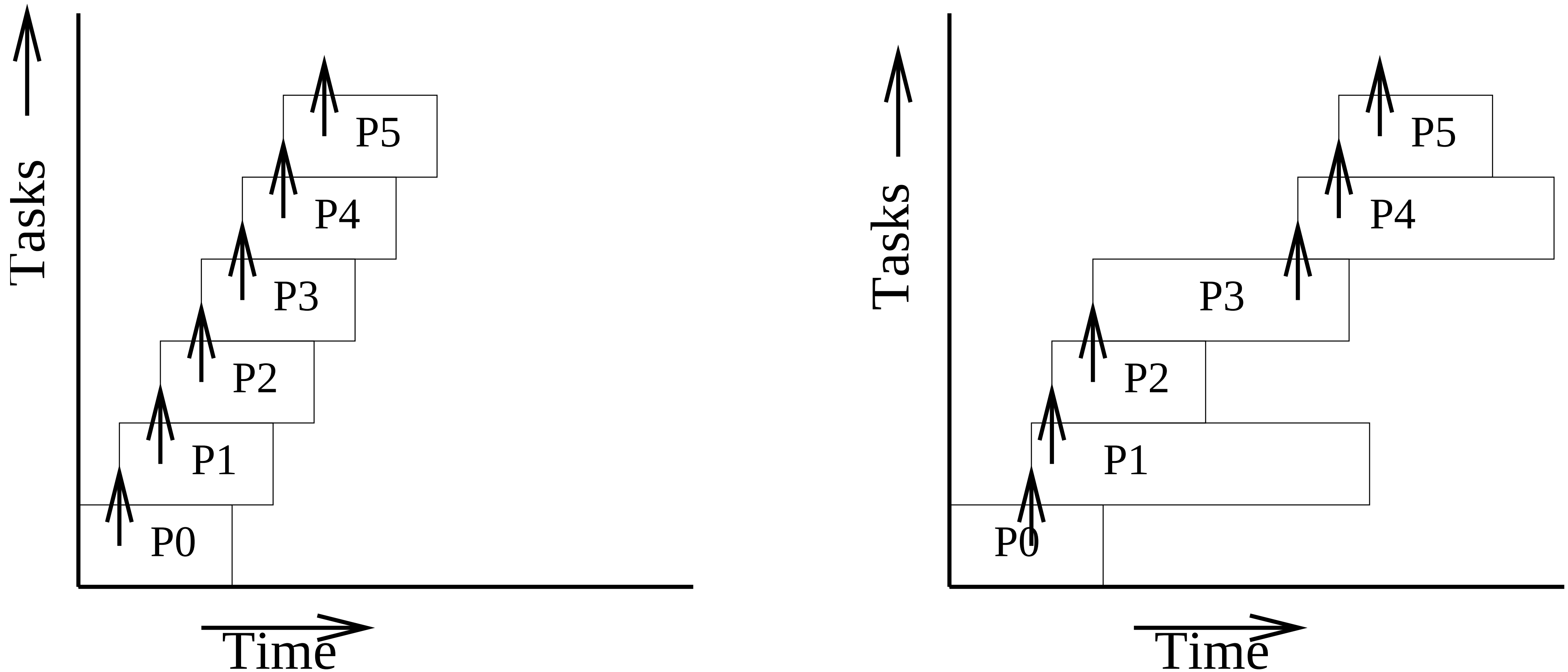
..Space-time diagrams

Type 2:

- a single instance of a problem, but with a series of data items to be processed
- examples: multiplications, sortings, etc.

..Space-time diagrams

Type 3: Pipeline processing where relevant *information for starting a next stage is passed before the end of the current task*.



(a) Processes with the same execution time

(b) Processes with possible different execution time

..Space-time diagrams

Type 3:

- a single instance of a problem, but now each process can pass on information to the next process before it has completed
- example: triangular systems of linear equations
- generally difficult to analyze

..Space-time diagrams

Stages vs. processes (processors):

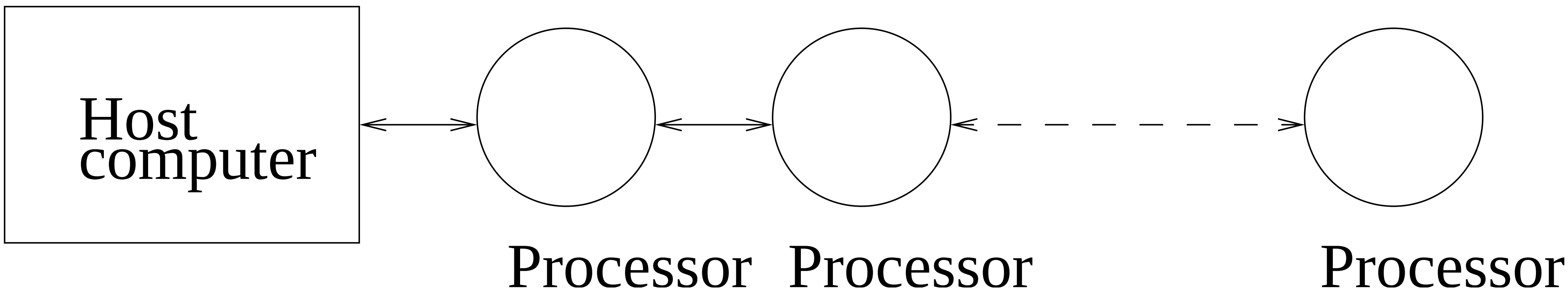
- What we have counted before was the number of *logical stages* in a pipeline solution of a problem.
- If their number is larger than the number of processors, then a *group of stages* can be assigned to each processor.

Example of partitioning:

$$\rightarrow \boxed{\boxed{P0} \rightarrow \boxed{P1} \rightarrow \boxed{P2} \rightarrow \boxed{P3}} \rightarrow \boxed{\boxed{P4} \rightarrow \boxed{P5} \rightarrow \boxed{P6} \rightarrow \boxed{P7}} \rightarrow \boxed{\boxed{P8} \rightarrow \boxed{P9} \rightarrow \boxed{P10} \rightarrow \boxed{P11}} \rightarrow$$

A computing platform

A possible computing platform for pipeline applications is described below:

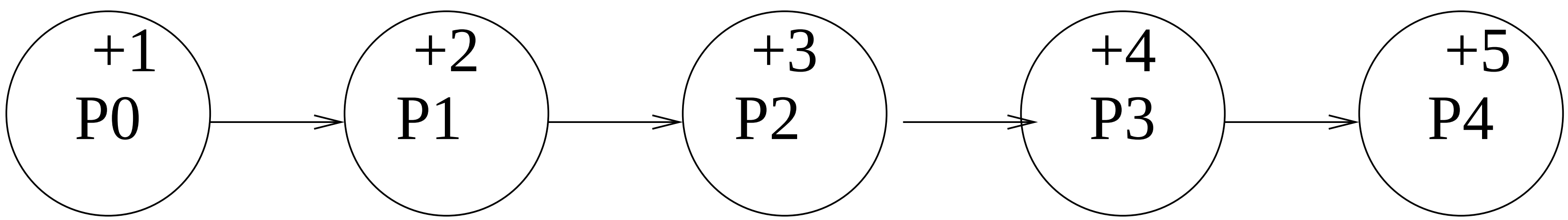


Data-flow architecture:

- Pipeline computation was one of the basic models used to develop *data-flow computation & machines*.
- Examples of programming languages based on this model are: *Lucid, Lustre, Signal*, etc.

Simple example

Adding numbers: A first, simple example is for adding some numbers (say, $\sum_1^k i$) using pipeline technique. The task is to add such numbers, each process holding a number. An illustration is below:



Processor Processor Processor Processor Processor

..example

Suppose we have n numbers/processes. The code is as follows:

- Code for process $P_i (0 < i < n)$:

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

- Code for process P_0:

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

- Code for process P_n:

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

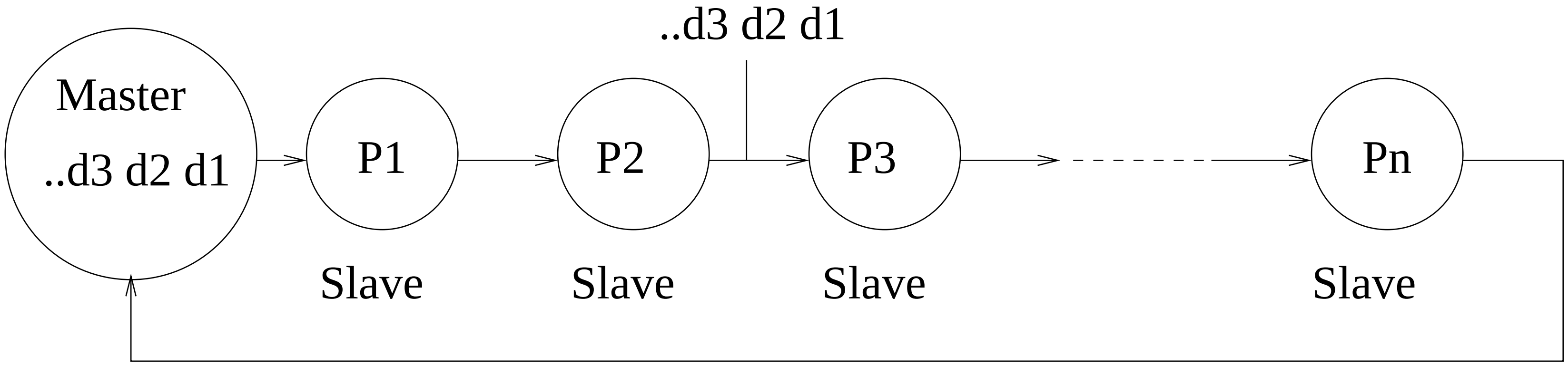
..example

One may write a SPMD program as follows

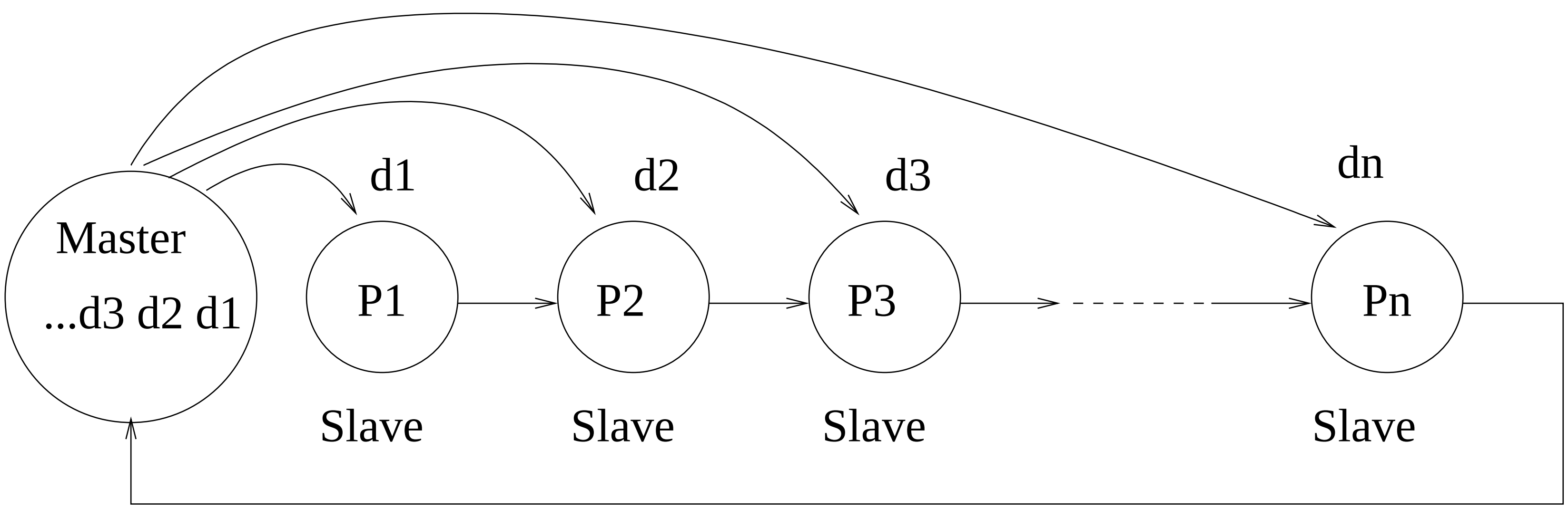
```
if (pid > 0){
      recv(&sum,P_{i-1});
      sum = sum + number;
}
if (pid < n-1){
      send(&sum,P_{i+1});
}
```

Concrete architectures:

- master process + ring configuration



- master process + direct access to slave processes



Case studies 1: Type 1, adding numbers

Analysis, adding numbers, Type 1: Suppose each process performs similar actions in each pipeline cycle. Then

- *(total time)* = *(time for one pipeline cycle)* × *(number of cycles)*
  If there are m instances and p pipeline stages, then the number of cycles is $m+p-1$, hence

$$t_{total} = (t_{comp} + t_{comm}) \times (m + p - 1)$$

- *(average time)* = *(total time) / (number of instances solved)*
  With the above notations,

$$t_a = \frac{t_{total}}{m}$$

..Type 1, adding numbers

Single instance of problem:

Suppose we have to add *one set ($m = 1$) of p data*.Then

$$t_{comp} = 1 \text{ (one addition)}$$
$$t_{comm} = 2(t_{startup} + t_{data})$$
$$\text{(left+right communications; only } \texttt{sum} \text{ is to be send)}$$
$$t_{total} = [2(t_{startup} + t_{data}) + 1]p.$$

Notice: Here (and in the following 2 slides) we suppose processes hold the numbers, hence only sum is communicated. If also numbers are to be sent, then one has to count the time to send these numbers. The result depends on the architecture - see Slide 16.

..Type 1, adding numbers

Multiple instances of problem: Suppose we have to add *m ($m > 1$) sets of p data*. Then

$$t_{comp} = 1 \text{ (one addition)}$$
$$t_{comm} = 2(t_{startup} + t_{data})$$
(left+right communications; only `sum` is to be send)
$$t_{total} = [2(t_{startup} + t_{data}) + 1](m + p - 1)$$

hence the average time is

$$t_a = \frac{t_{total}}{m} = [2(t_{startup} + t_{data}) + 1](1 + \frac{p-1}{m})$$

When m is large this is approximatively one pipeline cycle, i.e., $2(t_{startup} + t_{data}) + 1$. In other words, after a warming-up period the pipeline solves one instance per pipeline cycle.

..Type 1, adding numbers

Data partitioning with multiple instances of problem: Suppose we have to add *m* $(m > 1)$ *sets of n data, using p processes*, hence each process handle n/p data in a pipeline cycle. Then

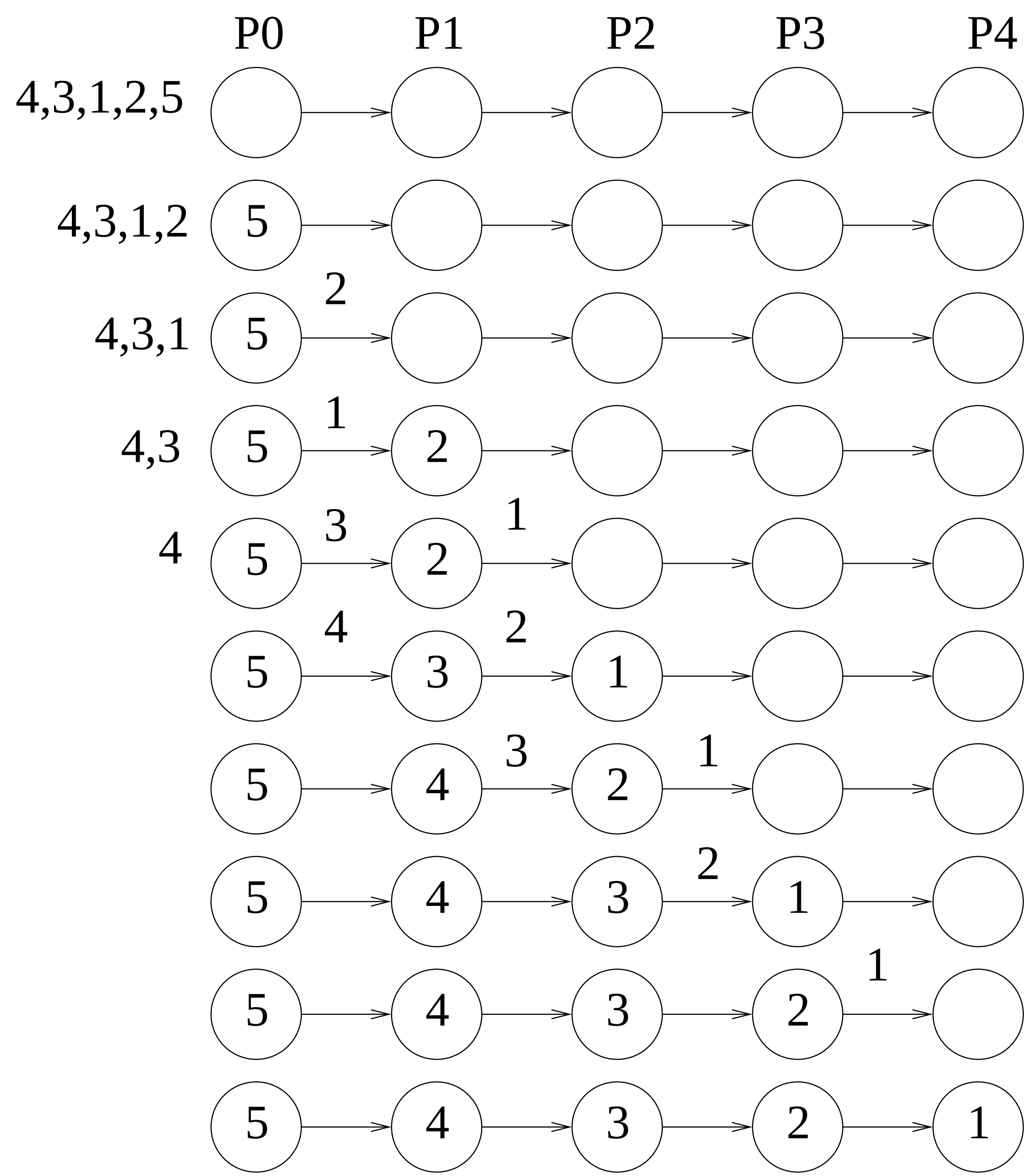
$$t_{comp} = n/p \text{ (additions)}$$
$$t_{comm} = 2(t_{startup} + t_{data})$$
(left+right communication; only `sum` is to be send)
$$t_{total} = [2(t_{startup} + t_{data}) + n/p](m + p - 1)$$

Notice: By increasing the length of data partition n/p the impact of communication is diminished. Increasing the length too much will decrease the parallelism and the execution time may increase.

Case studies 2a: Type 2, sorting

Sorting numbers, Type 2:

A parallel version of the *insertion sort*. ((1) The basic step is to insert a new number in a previously ordered sublist. (2) An extra step requires to shift numbers, if necessary. (3) The final algorithm is obtained allowing such sequences of operations to interfere in a pipeline style.)



..Type 2, sorting

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})
```

Notice: If the number of numbers (say, n) and the process number (say i) are known, then the above basic step is repeated $n-i+1$ times.

..Type 2, sorting

The final code for process $P_i (i > 0)$ is (provided an extra requirement is to have all sorted numbers hold by master process P_0):

```
rightProcNo = n-i-1;
recv(&number,P_{i-1});
for (j=0; j<rightProcNo; j++){
    recv(&number,P_{i-1});
    if (number > x){
        send(&x,P_{i+1});
        x = number;
    } else send(&number,P_{i+1})
}
send(& number, P_{i-1});
for (j=0; j<rightProcNo; j++){
    recv(&number,P_{i+1});
    send(&number,P_{i-1})
}
```

..Type 2, sorting

Analysis:

- *Sequential*:

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

  (poor sequential sorting algorithm; good only for very small n)

- *Parallel*: (for n numbers there are $2n-1$ pipeline circles)

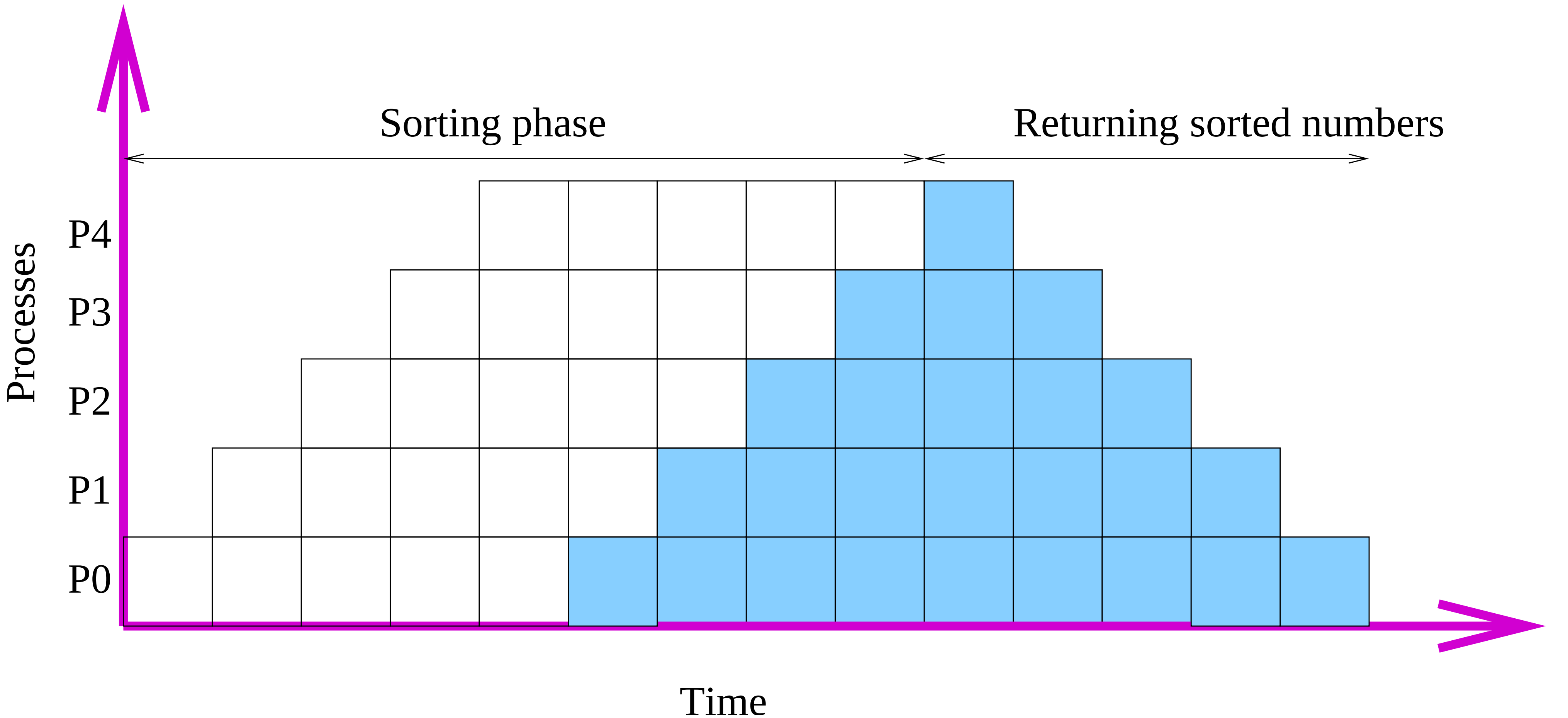
  $t_{comp} \leq 2$ (a comparison in `if` and sometimes an update of `x`)

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

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

..Type 2, sorting

Insertion sort with sorted numbers returned ($n = 5$)



(“sorting + returning” require $3n - 1$ cycles)

Case studies 2b: Type 2, prime numbers

Generate prime numbers using the Sieve of Eratosthenes.

Suppose we want to find the prime numbers form 2 to 20.
We start with all numbers

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$$

1st number 2 is prime; strike out the number and all its multiples

$$\cancel{2}, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}$$

1st (non-marked) number 3 is prime; strike out the number and all its multiples

$$\cancel{2}, \cancel{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}$$

... and so on.

To find all prime numbers up to n requires to repeat the marking procedure starting with (prime) numbers up to $\sqrt{n}$. (Each composite number up to n has at least one factor less than $\sqrt{n}$.)

..Type 2, prime numbers

Sequential code:

```
for (i=2; i<n; i++)
    prime[i] = 1;
for (i=2; i<=sqrt(n); i++){
    if (prime[i] == 1){
        for (j=i+i; i<n; j=j+i)
            prime[j] = 0;
    }
}
```

The program use an array `prime` *such that finally* `prime[i]` *is 1 if* `i` *is prime number, otherwise 0.*

..Type 2, prime numbers

Sequential time:

- The number of striking out steps depend on the prime number: $\lfloor n/2-1 \rfloor$ for 2, $\lfloor n/3-1 \rfloor$ for 3, and so on.
- Hence:

$$t_s = \lfloor n/2-1 \rfloor + \lfloor n/3-1 \rfloor + \lfloor n/5-1 \rfloor + \ldots + \lfloor n/p_k-1 \rfloor$$

where p_k is the greatest prime number $\leq \sqrt{n}$.

- Roughly, the growing of t_s is less than $\sqrt{n}n$ which is $O(n^{1.5})$.

Parallel code:

- A partitioning procedure may be quite inefficient.
- A pipeline solution acts as follows:
  - each process *keeps the 1st* received number, say p, as a prime number and
  - *passes forward* those received numbers which are *not multiple of* p.
- This procedure is more efficient as it avoids to multiple strike out the composite numbers for all their prime factors.

..Type 2, prime numbers

The basic piece of (pseudo)code for P_i may be

```
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});
}
```

We need a special "terminator" message, as the number of iteration steps is not a-priori known. The mod operator "%" is usually expensive and should be avoided.

Case studies 3: Type 3, systems of equations

Upper-triangular systems of linear equations:

These systems have the following shape:

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

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

..Type 3, systems

They are easily solved by backwards substitution method:

- compute x_0 from the last equation

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

- substitute the value of x_0 into the next equation to obtain x_1, i.e.,

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

- substitute the values of x_0, x_1 into the next equation to obtain x_2, i.e.,

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

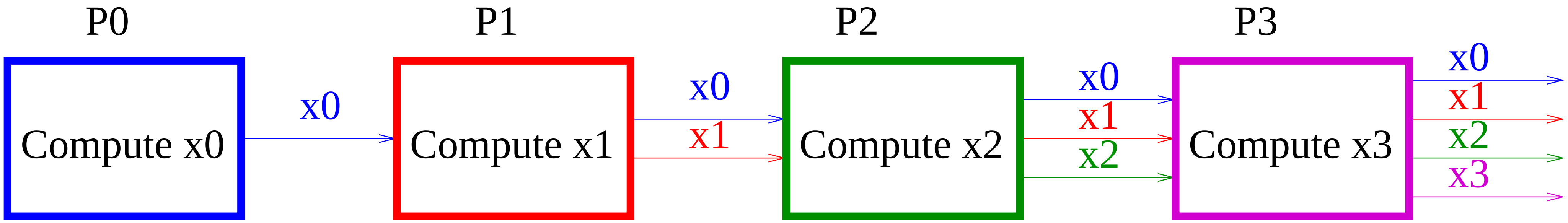
- ... and so on, till all unknowns are found.

..Type 3, systems

The general formula for solving x_i is:

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

A parallel *type 3, pipeline solution* comes naturally here: once x_i is obtained, it may be passed to all other upper processes to use it.



..Type 3, systems

The pseudocode of $P_i (0 < i < n)$ for a pipeline version is

```
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});
```

(The code of P_0 consists of the last 2 statements, only. The code of P_{n-1} consists of the above statements, except for the last `send`.)

..Type 3, systems

The analysis is quite difficult, as (1) one cannot assume the computational effort at each pipeline stage is the same and (2) the pipeline stages overlap. A rough analysis is as follows:

- Process P_0 performs one computation (division) and one `send`.
- The i-th process ($0 < i < n-1$) performs i sets of `send` / `recv` / multiply / addition, operations and finally one subtract / division / `send` set.
- Process P_{n-1} does similar operations as above, except for the last `send`.
- Finally, one has to consider the overlapping of the tasks/stages and to appropriately count the total execution time.