[*]

Pipelined Computations

[*]Material based on B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

Pipeline Technique

To use this method, problem must be divided into sequence of tasks.

Each task will be executed on a different processor, in the specified order. See Figure 5.1.

Refer to each process as a *pipeline stage.*

Each stage performs part of problem, and passes on information to the next stage.

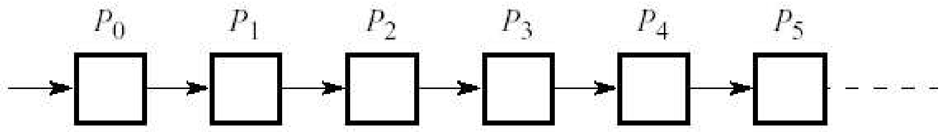


Figure 5.1 Pipelined processes.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Frequency Filter Example

An example of how a problem could be decomposed into stages is a frequency filter. Goal is to remove specific frequencies (f_0, f_1, f_2, and f_3) from a digitized signal $f(t)$

Signal $f(t)$ represents sequence of values, $f(0)$, $f(1), \ldots$ Each value must be processed in turn by each stage. Each stage removes one frequency.

Once pipeline is filled, will output results once per *time unit* (time unit is time for one stage to process a value).

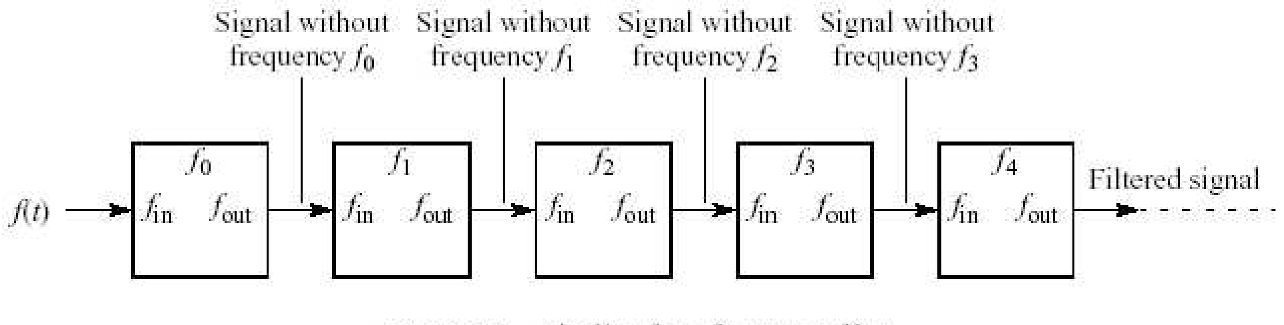

Signal without frequency f_0
Signal without frequency f_1
Signal without frequency f_2
Signal without frequency f_3
$f(t)$
f_0
f_1
f_2
f_3
f_4
f_{in}
f_{out}
Filtered signal

Figure 5.3 Pipeline for a frequency filter.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Pipeline Structural Conditions

Assuming that our problem can be decomposed into a sequence of tasks...

Potential for speedup in the following cases (types):

1) When more than one instance of a complete problem must be processed. *ie. an instruction pipeline of a computer where multiple instructions (problems) must be processed.*

2) When a series of data elements must be processed, and multiple operations must be applied to each element. *ie. image transformations for computer graphics.*

3) When the information for the next stage to begin executed is available, before the current stage has completed. *See Section 5.3.4 of text for an example.*

Type 1 Problem

Figure 5.4 illustrates a *type 1* pipeline using a *space-time diagram.*

In this example, each instance of the problem requires 6 stages to be processed.

Diagram assumes process for each stage requires same time to complete task.

Will refer to the time for a process to finish with one instance as one *pipeline cycle.*

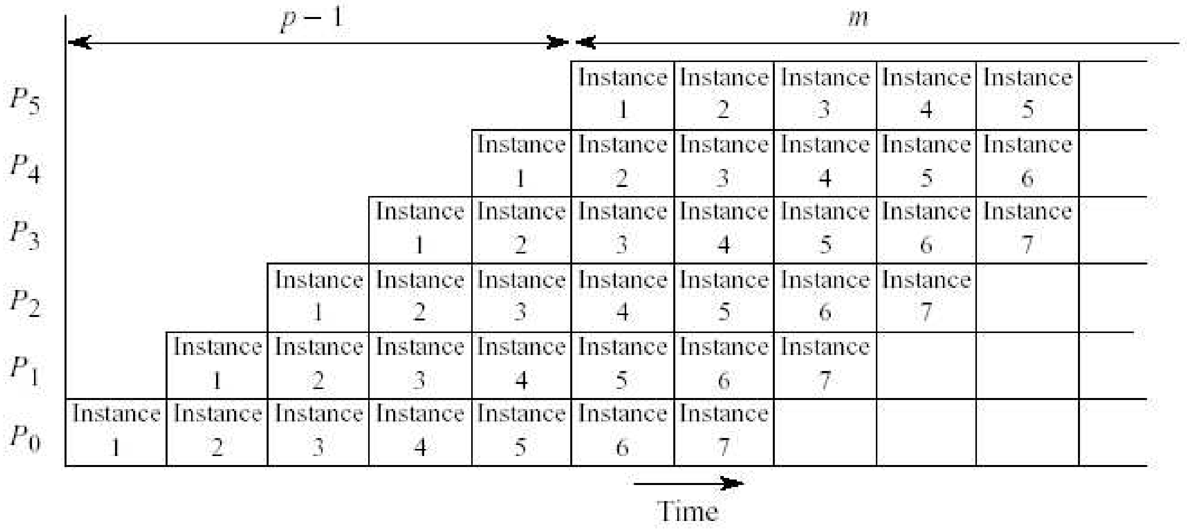


Figure 5.4 Space-time diagram of a pipeline.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Type 1 Problem Cont.

After 5 cycles, we get one instance completed per cycle.

Let p be the number of processes in pipeline, and m be the number of instances of the problem.

Cycles required to complete all m instances is:

$$\mathit{Cycles} = m + p - 1$$

$$\mathit{Average\ cycle\ time} = (m + p - 1)/m$$

Type 2 Problem

For this type of problem , multiple operations must be applied sequentially to each element in a series of data.

We have n data elements and p processes in our pipeline, as shown in Figure 5.6.

Again, we have: $\mathit{Cycles} = n + p - 1$

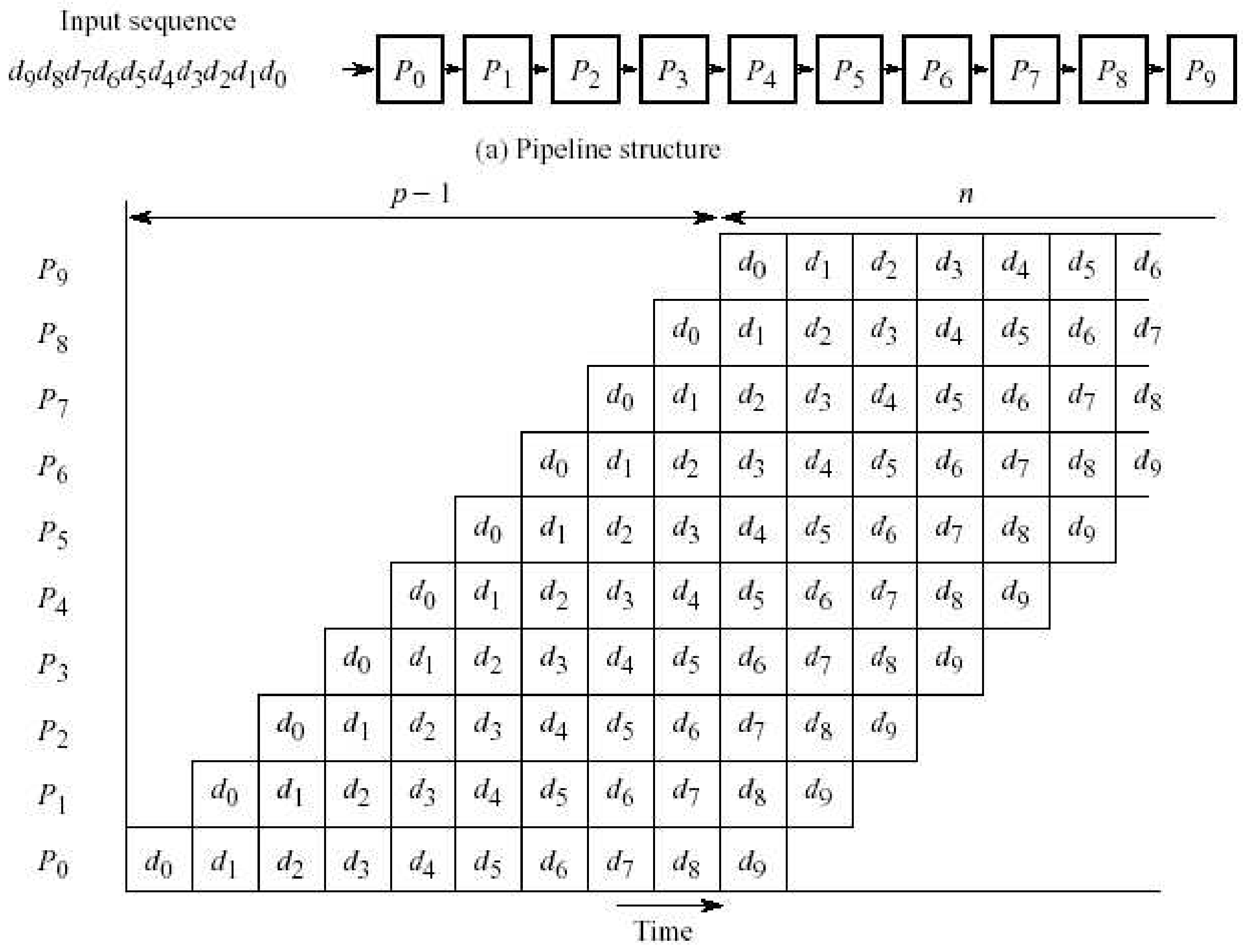

Input sequence
$d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9
(a) Pipeline structure
$p-1$
n
Time
(b) Timing diagram

Figure 5.6 Pipeline processing 10 data elements.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Type 3 Problem

This type used when one instance of problem, but information can be passed forward before current processing is complete.

Figure 5.7 shows space-time diagram illustrating this.

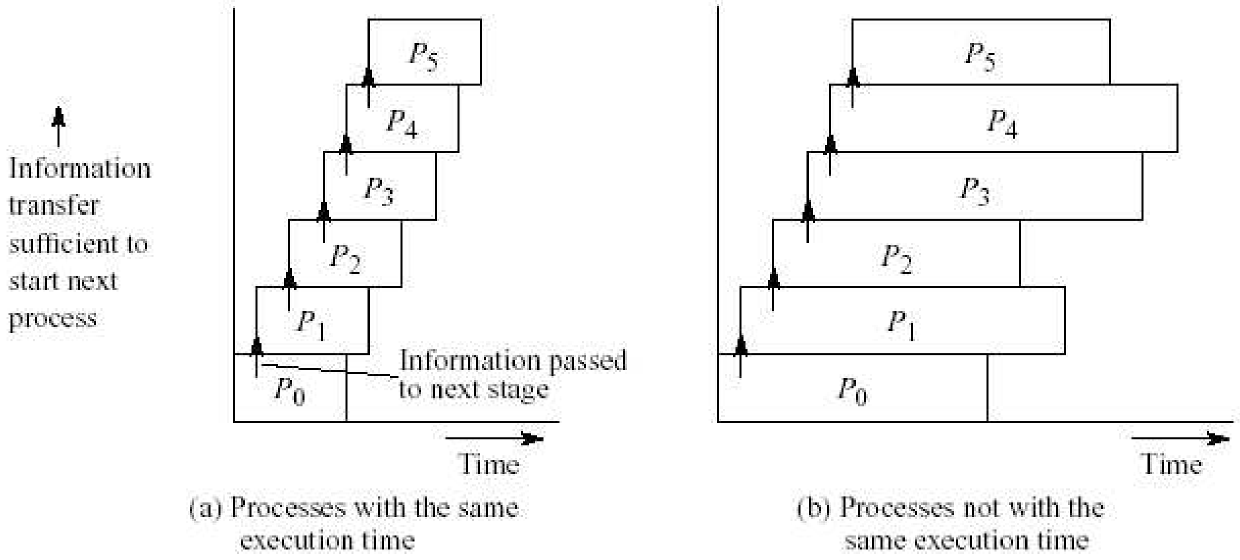

Information transfer sufficient to start next process
P_0
P_1
P_2
P_3
P_4
P_5
Information passed to next stage
Time
(a) Processes with the same execution time
P_0
P_1
P_2
P_3
P_4
P_5
Time
(b) Processes not with the same execution time

Figure 5.7 Pipeline processing where information passes to next stage before end of process.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Partitioning Stages

If more stages than processors, then assign groups to each processor as in Figure 5.8.

Stages wthin a given processor would be performed sequentially.

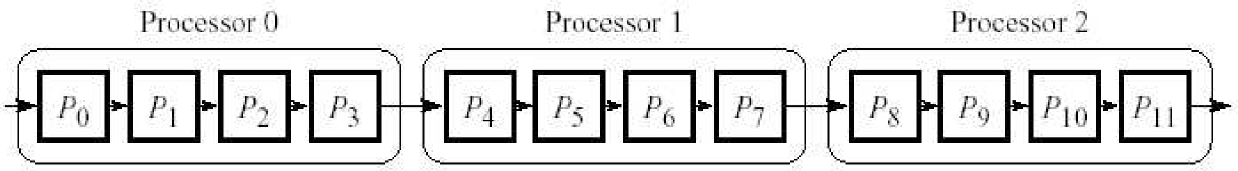


Figure 5.8 Partitioning processes onto processors.
Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Computing Platforms

For pipelines to work well, we need to be able to send messages between adjacent processors.

Implies we need direct links between adjacent processors.

Best interconnection structure is host system connected to processors organized in a line or ring such as Figure 5.9.

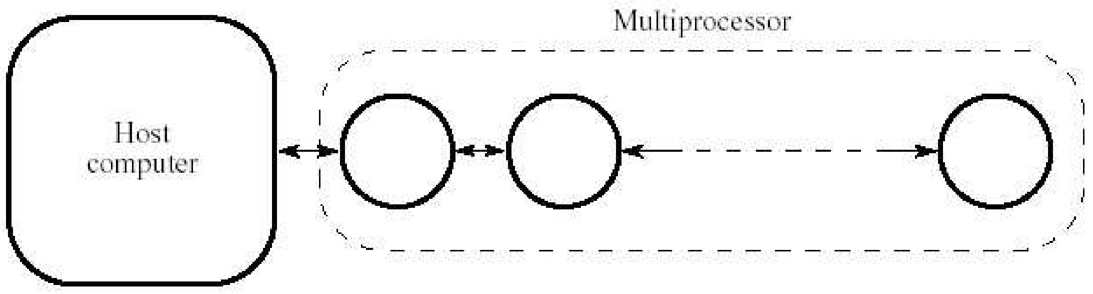


Figure 5.9 Multiprocessor system with a line configuration.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Computing Platforms Cont.

As lines/rings can be perfectly embedded into meshes/toruses and hypercubes, these work well also.

Bus structure topology doesn't work well, as only one processor could transmit at a time.

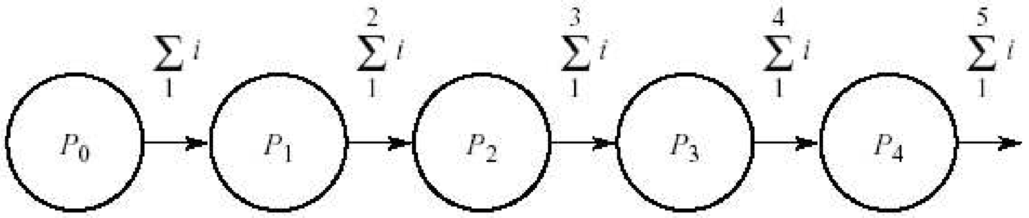
Will assume at least to be able to have simultaneous communication between adjacent processes.

Adding numbers Example

Want to add a set of n numbers with p processes. Type 1 problem.

One solution is when each processor adds one number (that it holds) to an accumulating sum. See Figure 5.10.

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



Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Adding numbers Example Cont.

We will assume that process 0 has two numbers, adds them, then sends the sum.

If we have more numbers than processes, each process will add a group together, then pass on sum.

We assume that each process performs similar actions so we calulate the number of steps required per cycle.

$m = \#$ of problem instances

Total execution time calculated as:

$$\begin{aligned} t_{\text{total}} &= \text{(time for one pipeline cycle)(number of cycles)} \\ t_{\text{total}} &= (t_{\text{comp}} + t_{\text{comm}})(m + p - 1) \end{aligned}$$

Average time to calculate a sum:

$$t_{\text{avg}} = \frac{t_{\text{total}}}{m}$$

Single Instance of Problem

Assume at first only one addition per process, thus $n = p + 1$.

Single instance, so $m = 1$.

Time for one cycle determined by one addition and two communications (to left and to right neighbour).

$$t_{\text{comp}} = 1; \quad t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})$$

$$\begin{aligned} t_{\text{total}} &= (2(t_{\text{startup}} + t_{\text{data}}) + 1)p \\ &= (2(t_{\text{startup}} + t_{\text{data}}) + 1)(n - 1) \end{aligned}$$

Time complexity is $\mathbf{O}(n)$.

Multiple Instances of Problem

If have m groups of n numbers to add.

Cycle time is the same, but now have more cycles.

$$\begin{aligned} t_{\text{total}} &= (2(t_{\text{startup}} + t_{\text{data}}) + 1)(m + p - 1) \\ &= (2(t_{\text{startup}} + t_{\text{data}}) + 1)(m + n - 2) \end{aligned}$$

For large m, average time for one problem instance:

$$t_{\text{avg}} = \frac{t_{\text{total}}}{m} \approx 2(t_{\text{startup}} + t_{\text{data}}) + 1$$

For serial algorithm, time for one sum is always:

$$t_s = n - 1 = p$$

To gain an advantage, requires:

$$2(t_{\text{startup}} + t_{\text{data}}) + 1 < p$$

For $t_{\text{startup}} = 8333$ and $t_{\text{data}} = 55$, we get:

$$16,777 < p$$

Data Partitioning and Multiple Instances

Now consider that each process does d additions.

That means each process has d numbers to add (plus incoming sum) except Process 0 which had $d+1$.

Total numbers are $n = p \cdot d + 1$

$$t_{\text{comp}} = d; \quad t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})$$

$$t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + d)(m + p - 1)$$

For large m, average time for one problem instance:

$$t_{\text{avg}} = \frac{t_{\text{total}}}{m} \approx 2(t_{\text{startup}} + t_{\text{data}}) + d$$

Data Partitioning and Multiple Instances Cont.

For serial algorithm, time for one sum is always:

$$t_s = n - 1 = p \cdot d$$

To gain an advantage, requires:

$$\begin{aligned} 2(t_{\text{startup}} + t_{\text{data}}) + d &< p \cdot d \\ \frac{2(t_{\text{startup}} + t_{\text{data}})}{d} + 1 &< p \end{aligned}$$

For $d = 5000$ $t_{\text{startup}} = 8333$ and $t_{\text{data}} = 55$, we get:

$$4.3552 < p$$