Chapter 5

Pipelined Computations
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:

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

The loop could be “unfolded” to yield

```c
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


s_in  → s_out  → s_in  → s_out  → s_in  → s_out  → s_in  → s_out
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 the next process can be passed forward before the process has completed all its internal operations
Execution time = m + p - 1 cycles for a p-stage pipeline and m instances.
Alternative space-time diagram

Instance 0

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
</table>
Instance 1

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
</table>
Instance 2

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
</table>
Instance 3

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
</table>
Instance 4

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
</table>

Time
"Type 2" Pipeline Space-Time Diagram

Input sequence
\[ d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 \]

(a) Pipeline structure

(b) Timing diagram
“Type 3” Pipeline Space-Time Diagram

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

(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:
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 $P_i$:

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

except for the first process, $P_0$, which is

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

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

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

if (process > 0) {
    recv(&accumulation, P_{i-1});
    accumulation = accumulation + number;
}
if (process < n-1) send(&accumulation, i_{P_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
Sorting Numbers

A parallel version of *insertion sort*.

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>Numbers</th>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4, 3, 1, 2, 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4, 3, 1, 2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4, 3, 1</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4, 3</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Numbers are sorted in parallel using a network of processors.
Pipeline for sorting using insertion sort

Series of numbers $x_{n-1} \ldots x_1 x_0$

Type 2 pipeline computation
The basic algorithm for process $P_i$ is

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

With $n$ numbers, how many the $i$th process is to accept is known; it is given by $n - i$. How many to pass onward is also known; it is given by $n - i - 1$ since one of the numbers received is not passed onward. Hence, a simple loop could be used.
Insertion sort with results returned to the master process using a bidirectional line configuration.

Master process

\[ d_{n-1} \ldots d_2 d_1 d_0 \]

Sorted sequence

\[ P_0 \quad P_1 \quad P_2 \quad P_{n-1} \]
Insertion sort with results returned

Sorting phase

$2n - 1$

Returning sorted numbers

$n$

Shown for $n = 5$
Prime Number Generation

Sieve of Eratosthenes

Series of all integers is generated from 2. First number, 2, is prime and kept. All multiples of this number are deleted as they cannot be prime. Process repeated with each remaining number. The algorithm removes nonprimes, leaving only primes.
Pipeline for Prime Number Generation

Series of numbers
\[ x_{n-1} \ldots 5 4 3 2 \]

Compare multiples
1st prime number

Not multiples of 1st prime number

2nd prime number

3rd prime number

Type 2 pipeline computation
The code for a process, $P_i$, could be based upon

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

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

```c
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, iP_{i+1});
}
```
Solving a System of Linear Equations

Upper-triangular form

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

\[ \vdots \]

\[ a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2 \]

\[ a_{1,0}x_0 + a_{1,1}x_1 = b_1 \]

\[ a_{0,0}x_0 = b_0 \]

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.,

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

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 $i$th process ($0 < i < n$) receives the values $x_0, x_1, x_2, \ldots, x_{i-1}$ 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

```c
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

```c
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 additional computations after receiving and resending values.
Pipeline processing using back substitution

Processes

Time

First value passed onward

Final computed value