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 \texttt{a} to an accumulating sum:

\begin{verbatim}
for (i = 0; i < n; i++)
    sum = sum + a[i];
\end{verbatim}

The loop could be “unfolded” to yield

\begin{verbatim}
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
\end{verbatim}
Pipeline for an unfolded loop

\[
\text{sum} \rightarrow a[n] \rightarrow a[n+1] \rightarrow a[n+2] \rightarrow a[n+3] \rightarrow a[n+4] \rightarrow \ldots
\]

\[
s_{\text{in}} \quad s_{\text{out}} \quad s_{\text{in}} \quad s_{\text{out}} \quad s_{\text{in}} \quad s_{\text{out}} \quad s_{\text{in}} \quad s_{\text{out}} \quad \ldots
\]
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
“Type 1” Pipeline Space-Time Diagram

Execution time = $m + p - 1$ cycles for a $p$-stage pipeline and $m$ instances.
Alternative space-time diagram

Instance 0

\[
\begin{array}{ccccccc}
& P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
P_0 & & & & & & \\
P_1 & & & & & & \\
P_2 & & & & & & \\
P_3 & & & & & & \\
P_4 & & & & & & \\
P_5 & & & & & & \\
\end{array}
\]

Instance 1

\[
\begin{array}{ccccccc}
& P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
P_0 & & & & & & \\
P_1 & & & & & & \\
P_2 & & & & & & \\
P_3 & & & & & & \\
P_4 & & & & & & \\
P_5 & & & & & & \\
\end{array}
\]

Instance 2

\[
\begin{array}{ccccccc}
& P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
P_0 & & & & & & \\
P_1 & & & & & & \\
P_2 & & & & & & \\
P_3 & & & & & & \\
P_4 & & & & & & \\
P_5 & & & & & & \\
\end{array}
\]

Instance 3

\[
\begin{array}{ccccccc}
& P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
P_0 & & & & & & \\
P_1 & & & & & & \\
P_2 & & & & & & \\
P_3 & & & & & & \\
P_4 & & & & & & \\
P_5 & & & & & & \\
\end{array}
\]

Instance 4

\[
\begin{array}{ccccccc}
& P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
P_0 & & & & & & \\
P_1 & & & & & & \\
P_2 & & & & & & \\
P_3 & & & & & & \\
P_4 & & & & & & \\
P_5 & & & & & & \\
\end{array}
\]

\cdots

Time
“Type 2” Pipeline Space-Time Diagram

Input sequence \( \text{d}_{9}\text{d}_{8}\text{d}_{7}\text{d}_{6}\text{d}_{5}\text{d}_{4}\text{d}_{3}\text{d}_{2}\text{d}_{1}\text{d}_{0} \)

(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...
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.

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

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

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

Master process: \( d_{n-1} \ldots d_2 d_1 d_0 \)

Slaves: \( P_0, P_1, P_2, \ldots, P_{n-1} \)
Sorting Numbers

A parallel version of *insertion sort*.

Numbers:

```
4, 3, 1, 2, 5
4, 3, 1, 2
4, 3, 1
4, 3
4
```

Time (cycles):

```
1
2
3
4
5
6
7
8
9
10
```
Pipeline for sorting using insertion sort

Series of numbers $x_{n-1} \ldots x_1x_0$

- $P_0$: Compare
  - $x_{\text{max}}$
  - Largest number

- $P_1$: Smaller numbers
- $P_2$: Next largest number

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

Sorted sequence

\( d_{n-1} \ldots d_2 d_1 d_0 \)
Insertion sort with results returned

Sorting phase

Returning sorted numbers

$2n - 1$

$n$

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.

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

\[
\begin{align*}
\text{recv}(&x, P_{i-1}); \\
/* \text{repeat following for each number} */ \\
\text{recv}(&\text{number}, P_{i-1}); \text{ \quad if } (\text{number} \mod x) \neq 0 \text{ send(&number, } P_{i+1});
\end{align*}
\]

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:

\[
\begin{align*}
\text{recv}(&x, P_{i-1}); \\
\text{for } (i = 0; i < n; i++) \{ \\
\text{recv}(&\text{number}, P_{i-1}); \\
\text{if } (\text{number} == \text{terminator}) \text{ break}; \\
\text{if } (\text{number} \mod x) \neq 0 \text{ send(&number, } P_{i+1});
\}
\end{align*}
\]
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 \( a \)'s and \( b \)'s are constants and \( x \)'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

\[
x[0] = b[0]/a[0][0]; \quad /* computed separately */
for (i = 1; i < n; i++) { /*for remaining unknowns*/
    sum = 0;
    for (j = 0; j < i; j++)
        sum = sum + a[i][j]*x[j];
    x[i] = (b[i] - sum)/a[i][i];
}
\]
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 we have additional computations to do after receiving and resending values.
Pipeline processing using back substitution

Processes

Time

First value passed onward

Final computed value