Pipelined Computations

In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other.

In fact, this is the basis of sequential programming.

Each task will be executed by a separate process or processor.

![Pipeline Diagram](image)

**Figure 5.1** Pipelined processes.

This parallelism can be viewed as a form of *functional decomposition*.

The problem is divided into separate functions that must be performed, but in this case, the functions are performed in succession.

As we shall see, the input data is often broken up and processed separately.
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];
```

One pipeline solution:

![Pipeline for an unfolded loop.](image)

**Figure 5.2** Pipeline for an unfolded loop.

Stage $i$ performs

$$s_{\text{out}} = s_{\text{in}} + a[i];$$
Example

A frequency filter - The objective here is to remove specific frequencies (say the frequencies $f_0, f_1, f_2, f_3$, etc.) from a (digitized) signal, $f(t)$. The signal could enter the pipeline from the left:

![Diagram of frequency filter pipeline](image)

**Figure 5.3** Pipeline for a frequency filter.
Given that the problem can be divided into a series of sequential tasks, the pipelined approach can provide increased 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

Figure 5.4  Space-time diagram of a pipeline.
Figure 5.5 Alternative space-time diagram.
“Type 2” Pipeline Space-Time Diagram

Input sequence $d_0d_1d_2d_3d_4d_5d_6d_7d_8d_9$

(a) Pipeline structure

(b) Timing diagram

Figure 5.6  Pipeline processing 10 data elements.
“Type 3” Pipeline Space-Time Diagram

Information transfer sufficient to start next process

(a) Processes with the same execution time

(b) Processes not with the same execution time

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

![Diagram of processes partitioned onto processors](image)

**Figure 5.8** Partitioning processes onto processors.
Computing Platform for Pipelined Applications

![Diagram showing a multiprocessor system with a line configuration.](image)

**Figure 5.9** Multiprocessor system with a line configuration.
Pipeline Program Examples

Adding Numbers

The 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, P_{i-1});
    accumulation = accumulation + number;
}
if (process < n-1) send(&accumulation, P_{i+1});
```

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.
Figure 5.11  Pipelined addition numbers with a master process and ring configuration.
Figure 5.12  Pipelined addition of numbers with direct access to slave processes.
Analysis

Our first pipeline example is Type 1. We will assume that each process performs similar actions in each pipeline cycle. Then we will work out the computation and communication required in a pipeline cycle.

The total execution time:

\[
t_{\text{total}} = (\text{time for one pipeline cycle})(\text{number of cycles})
\]

\[
t_{\text{total}} = (t_{\text{comp}} + t_{\text{comm}})(m + p - 1)
\]

where there are \( m \) instances of the problem and \( p \) pipeline stages (processes).

The average time for a computation is given by

\[
t_a = \frac{t_{\text{total}}}{m}
\]

**Single Instance of Problem**

\[
t_{\text{comp}} = 1
\]

\[
t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})
\]

\[
t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)n
\]

The time complexity = \( O(n) \).

**Multiple Instances of Problem**

\[
t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)(m + n - 1)
\]

\[
t_a = \frac{t_{\text{total}}}{m} \approx 2(t_{\text{startup}} + t_{\text{data}}) + 1
\]

That is, one pipeline cycle

**Data Partitioning with Multiple Instances of Problem**

\[
t_{\text{comp}} = d
\]

\[
t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})
\]

\[
t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + d)(m + n/d - 1)
\]

As we increase the \( d \), the data partition, the impact of the communication diminishes. But increasing the data partition decreases the parallelism and often increases the execution time.
Sorting Numbers

A parallel version of *insertion sort*. (The sequential version is akin to placing playing cards in order by moving cards over to insert a card in position)

![Diagram of steps in insertion sort with five numbers](image)

**Figure 5.13** Steps in insertion sort with five numbers.

The basic algorithm for process $P_i$ is

```plaintext
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.
Figure 5.14  Pipeline for sorting using insertion sort.
Figure 5.15 Insertion sort with results returned to the master process using a bidirectional line configuration.

Incorporating results being returned, process $i$ could have the form

```c
right_procno = n - i - 1; /* no of processes to the right */
recv(&x, P_{i-1});
for (j = 0; j < right_procno; 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}); /* send number held */
for (j = 0; j < right_procno; j++) { /* pass on other numbers */
    recv(&x, P_{i+1});
    send(&x, P_{i-1});
}
```
Analysis

Sequential

\[ t_s = (n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{n(n - 1)}{2} \]

Obviously a very poor sequential sorting algorithm and unsuitable except for very small \( n \).

Parallel

Each pipeline cycle requires at least

\[ t_{\text{comp}} = 1 \]
\[ t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}}) \]

The total execution time, \( t_{\text{total}} \), is given by

\[ t_{\text{total}} = (t_{\text{comp}} + t_{\text{comm}})(2n - 1) = (1 + 2(t_{\text{startup}} + t_{\text{data}}))(2n - 1) \]

Figure 5.16  Insertion sort with results returned.
Prime Number Generation

Sieve of Eratosthenes

A series of all integers is generated from 2. The first number, 2, is prime and kept. All multiples of this number are deleted as they cannot be prime. The process is repeated with each remaining number. The algorithm removes nonprimes, leaving only primes.

Example
Suppose we want the prime numbers from 2 to 20. We start with all the numbers:

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

After considering 2, we get

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

where the numbers with \( \cancel{ } \) are marked as not prime and not to be considered further. After considering 3, we get

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

Subsequent numbers are considered in a similar fashion. However, to find the primes up to \( n \), it is only necessary to start at numbers up to \( \sqrt{n} \). All multiples of numbers greater than \( \sqrt{n} \) will have been removed as they are also a multiple of some number equal or less than \( \sqrt{n} \).
Sequential Code

A sequential program for this problem usually employs an array with elements initialized to 1 (TRUE) and set to 0 (FALSE) when the index of the element is not a prime number.

Letting the last number be $n$ and the square root of $n$ be $\sqrt{n}$, we might have

```c
for (i = 2; i < n; i++)
    prime[i] = 1; /* Initialize array */
for (i = 2; i <= sqrt_n; i++) /* for each number */
    if (prime[i] == 1) /* identified as prime */
        for (j = i + i; j < n; j = j + i) /* strike out all multiples */
            prime[j] = 0; /* includes already done */
```

The elements in the array still set to 1 identify the primes (given by the array indices). Then a simple loop accessing the array can find the primes.

**Sequential time**

The number of iterations striking out multiples of primes will depend upon the prime. There are $\lfloor n/2 \rfloor - 1$ multiples of 2, $\lfloor n/3 \rfloor - 1$ multiples of 3, and so on.

Hence, the total sequential time is given by

$$t_s = \left\lfloor \frac{n}{2} - 1 \right\rfloor + \left\lfloor \frac{n}{3} - 1 \right\rfloor + \left\lfloor \frac{n}{5} - 1 \right\rfloor + \ldots + \left\lfloor \frac{n}{\sqrt{n}} - 1 \right\rfloor$$

assuming the computation in each iteration equates to one computational step. The sequential time complexity is $O(n^2)$. 

---

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1999
Pipelined Implementation

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, P_{i+1});
```

A simple `for` loop is not sufficient for repeating the actions because each process will not receive the same amount of numbers and the amount is not known beforehand.

A general technique for dealing with this situation in pipelines is to use a “terminator” message, which is sent at the end of the sequence. Then each process could be

```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, P_{i+1});
}
```
Solving a System of Linear Equations — Special Case

The final example is Type 3 in which the process can continue with useful work after passing on information.

The objective here is to solve a system of linear equations of the so-called *upper-triangular* form:

\[
\begin{align*}
  a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \cdots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \\
  \vdots \\
  a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 &= b_2 \\
  a_{1,0}x_0 + a_{1,1}x_1 &= b_1 \\
  a_{0,0}x_0 &= b_0 
\end{align*}
\]

where the \( a \)'s and \( b \)'s are constants and the \( x \)'s are unknowns to be found.

The method used to solve for the unknowns \( x_0, x_1, x_2, \ldots, x_{n-1} \) is a simple repeated “back” substitution. First, the unknown \( x_0 \) is found from the last equation; i.e.,

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

The value obtained for \( x_0 \) is substituted into the next equation to obtain \( x_1 \); i.e.,

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

The values obtained for \( x_1 \) and \( x_0 \) are substituted into the 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.

Clearly, this algorithm can be implemented as a pipeline. The 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.
Figure 5.18  Solving an upper triangular set of linear equation using a pipeline.

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]; /* x[0] computed separately */
for (i = 1; i < n; i++) {
    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

The pseudocode of process $P_i$ ($1 < i < n$) of one pipelined version 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});
```

($P_0$ simply computes $x_0$ and passes $x_0$ on.) Now we have additional computations to do after receiving and resending values.
Figure 5.19 Pipeline processing using back substitution.
Analysis

For this pipeline, we cannot assume that the computational effort at each pipeline stage is the same.

The first process, $P_0$, performs one divide and one \texttt{send()}.

The $i$th process ($0 < i < n - 1$) performs $i$ \texttt{recv}(), $i$ \texttt{send}(), $i$ multiply/add, one divide/subtract, and a final \texttt{send}(), a total of $2i + 1$ communication times and $2i + 2$ computational steps assuming that multiply, add, divide, and subtract are each one step.

The last process, $P_{n-1}$, performs $n - 1$ \texttt{recv}()s, $n - 1$ multiply/adds, and one divide/subtract, a total of $n - 1$ communication times and $2n - 1$ computational steps.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{back_substitution_pipeline.png}
\caption{Operations in back substitution pipeline.}
\end{figure}