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.

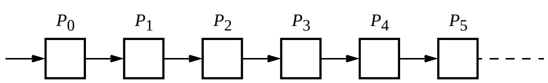

Figure 5.1 Pipelined processes.

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:

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

The loop could be “unfolded” to yield

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
    .
```

One pipeline solution:

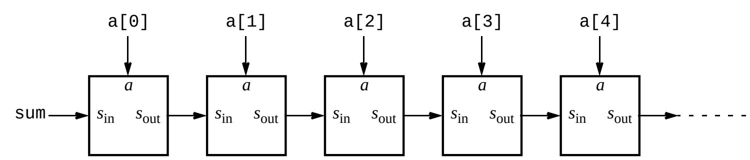

Figure 5.2 Pipeline for an unfolded loop.

Stage *i* performs

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

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

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:

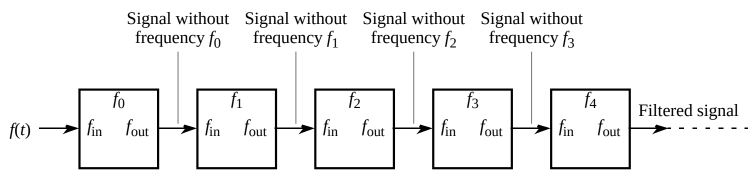

Figure 5.3 Pipeline for a frequency filter.

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

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

“Type 1” Pipeline Space-Time Diagram

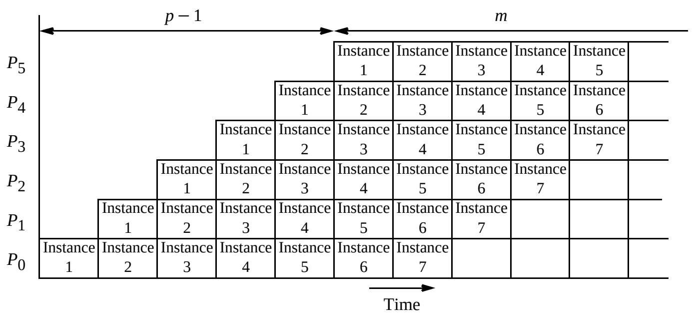

Figure 5.4 Space-time diagram of a pipeline.

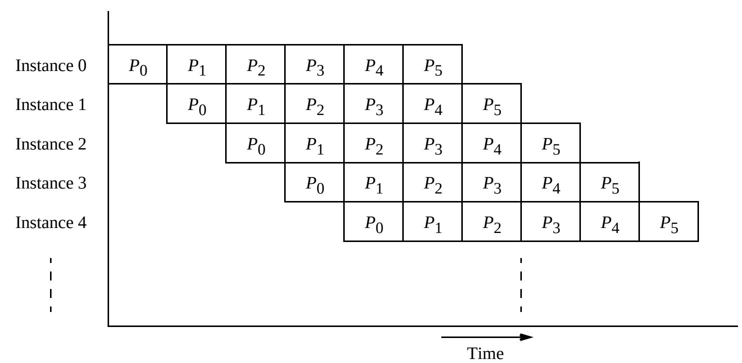

Figure 5.5 Alternative space-time diagram.

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

“Type 2” Pipeline Space-Time Diagram

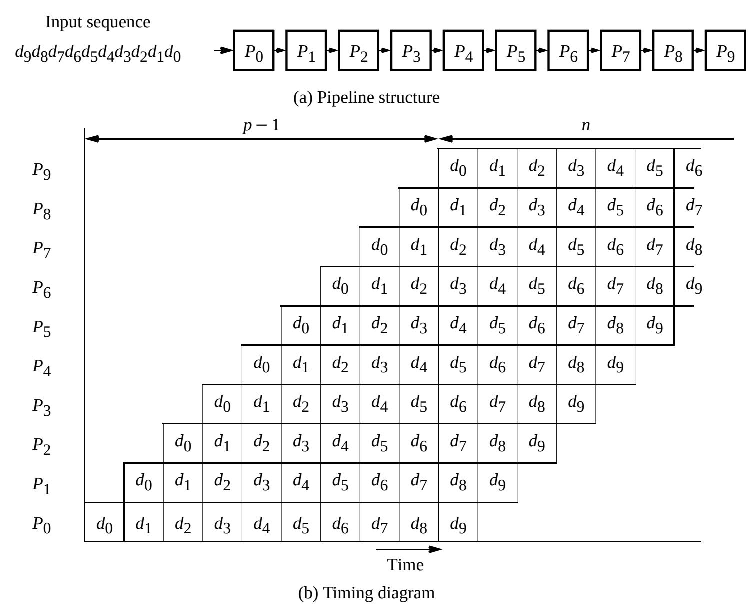

Figure 5.6 Pipeline processing 10 data elements.

“Type 3” Pipeline Space-Time Diagram

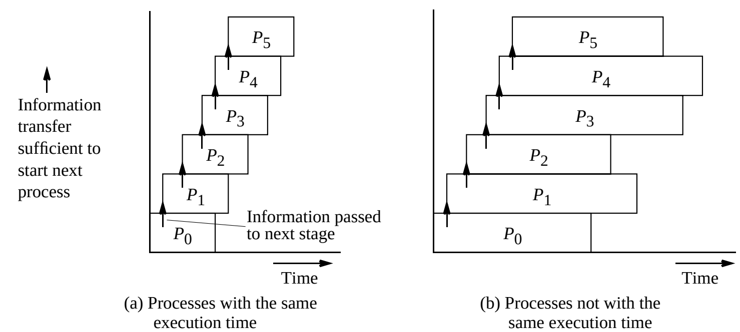

Figure 5.7 Pipeline processing where information passes to next stage before end of process.

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

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:

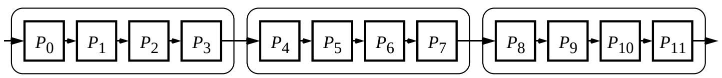

Figure 5.8 Partitioning processes onto processors.

Computing Platform for Pipelined Applications

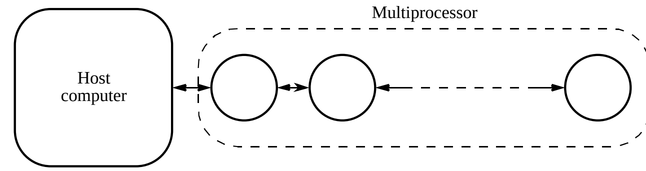

Figure 5.9 Multiprocessor system with a line configuration.

Pipeline Program Examples

Adding Numbers

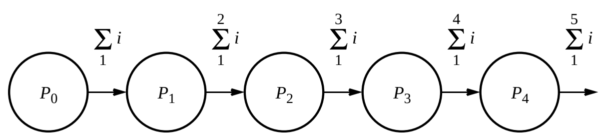

Figure 5.10 Pipelined addition.

The basic code for process P_i :

```
recv(&accumulation, Pi-1);
accumulation = accumulation + number;
send(&accumulation, Pi+1);
```

except for the first process, P_0, which is

```
send(&number, P1);
```

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

```
recv(&number, Pn-2);
accumulation = accumulation + number;
```

SPMD program

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

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

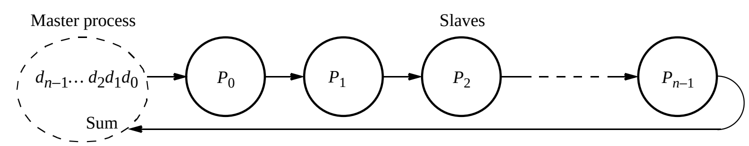

Figure 5.11 Pipelined addition numbers with a master process and ring configuration.

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

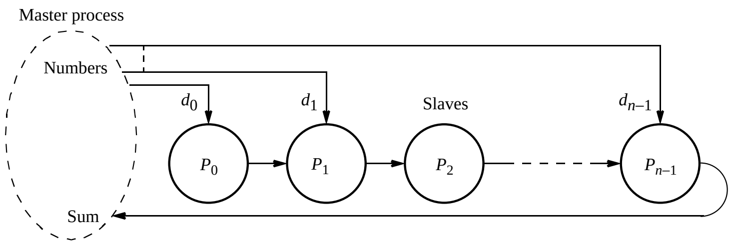

Figure 5.12 Pipelined addition of numbers with direct access to slave processes.

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.

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

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)

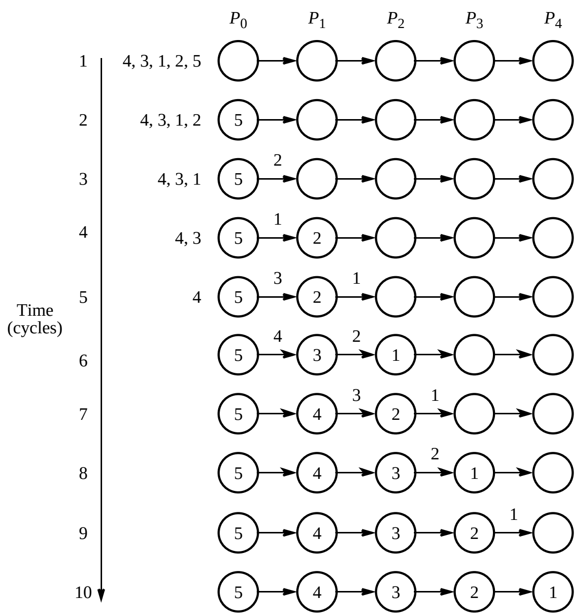

Figure 5.13 Steps in insertion sort with five numbers.

The basic algorithm for process P_i is

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

With n numbers, how many the ith 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.

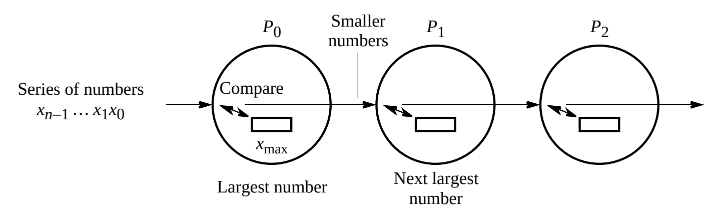

Figure 5.14 Pipeline for sorting using insertion sort.

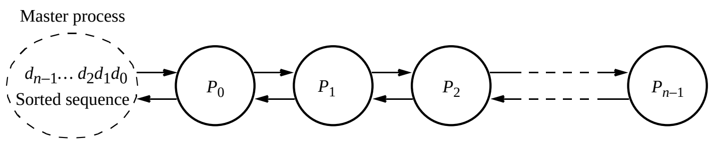

Figure 5.15 Insertion sort with results returned to the master process using a bidirectional line configuration.

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

```
right_procno = n - i - 1;            /* no of processes to the right */
recv(&x, Pi-1);
for (j = 0; j < right_procno; j++) {
   recv(&number, Pi-1);
   if (number > x) {
      send(&x, Pi+1);
      x = number;
   } else send(&number, Pi+1);
}
send(&number, Pi-1);                        /* send number held */
for (j = 0; j < right_procno; j++) {      /* pass on other numbers */
   recv(&x, Pi+1);
   send(&x, Pi-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_{total}, is given by

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

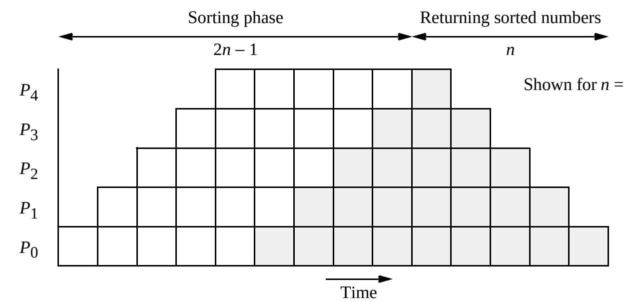

Figure 5.16 Insertion sort with results returned.

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

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, $\cancel{12}$, 13, $\cancel{14}$, 15, $\cancel{16}$, 17, $\cancel{18}$, 19, $\cancel{20}$

where the numbers with / are marked as not prime and not to be considered further. After considering 3, we get

2, 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, $\cancel{20}$

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

```
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 $\lfloor n/2 - 1 \rfloor$ multiples of 2, $\lfloor n/3 - 1 \rfloor$ 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, 1998

Pipelined Implementation

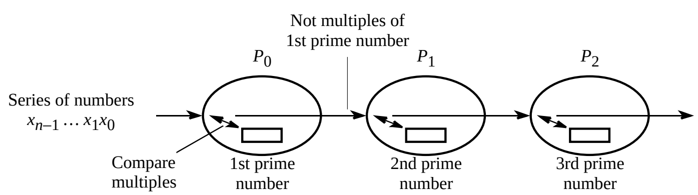

Figure 5.17 Pipeline for sieve of Eratosthenes.

The code for a process, P_i, could be based upon

```
recv(&x, Pi-1);
/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0) send(&number, Pi+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

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
  recv(&number, Pi-1);
  if (number == terminator) break;
  if (number % x) != 0) send(&number, Pi+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:

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

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.

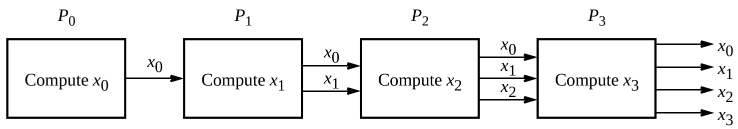

Figure 5.18 Solving an upper triangular set of linear equation using a pipeline.

The ith 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];              /* x[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 Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Parallel Code

The pseudocode of process P_i ($1 < i < n$) of one pipelined version could be

```
for (j = 0; j < i; j++) {
  recv(&x[j], Pi-1);
  send(&x[j], Pi+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], Pi+1);
```

(P_0 simply computes x_0 and passes x_0 on.) Now we have additional computations to do after receiving and resending values.

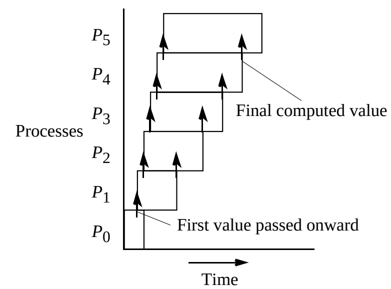

Figure 5.19 Pipeline processing using back substitution.

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

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 `send()`.

The ith process ($0 < i < n - 1$) performs i `recv()`s, i `send()`s, i multiply/add, one divide/subtract, and a final `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 $n - 1$ `recv()`s, $n - 1$ multiply/adds, and one divide/subtract, a total of $n - 1$ communication times and $2n - 1$ computational steps.

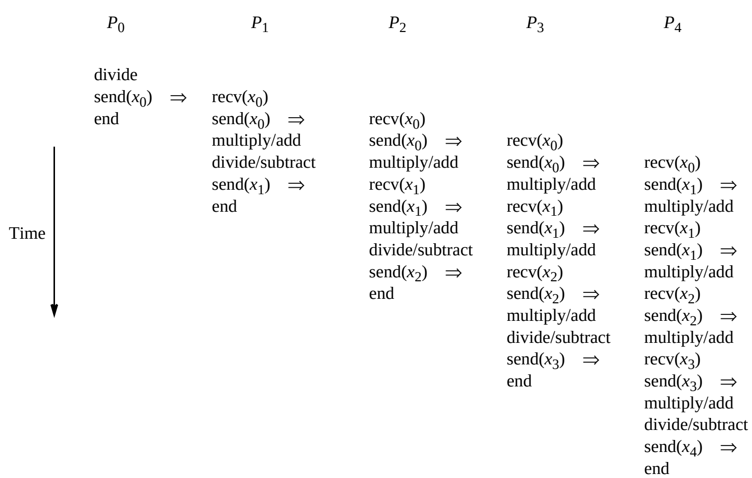

Figure 5.20 Operations in back substitution pipeline.

PROBLEMS

Scientific/Numerical

5-1. Write a parallel program to compute x^{16} using a pipeline approach. Repeat by applying a divide-and-conquer approach. Compare the two methods analytically and experimentally.

5-2. Develop a pipeline solution to compute $\sin\theta$ according to

$$\sin\theta = \theta - \frac{\theta}{3} + \frac{\theta}{5} - \frac{\theta}{7} + \frac{\theta}{9} - \ldots$$

A series of values are input, $\theta_0, \theta_1, \theta_2, \theta_3, \ldots$.

5-3. Modify the program in Problem 5-2 to compute $\cos\theta$ and $\tan\theta$.

5-4. Write a parallel program using pipelining to compute the polynomial

$$f = a_0x^0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

to any degree, n, where the a's, x, and n are input. Compare the pipelined approach with the divide-and-conquer approach (Problem 4-8 in Chapter 4).

5-5. Explore the trade-offs of increasing the data partition in the pipeline addition described in Section . Write parallel programs to find the optimum data partition for your system.

5-6. Compare insertion sort (Section 5.3.2) implemented sequentially and implemented as a pipeline, in terms of speedup and time complexity.

5-7. Rework the parallel code for finding prime numbers in Section 5.3.3 to avoid the use of the mod operator to make the algorithm more efficient.

5-8. Radix sort is similar to the bucket sort described in Chapter 4, Section 4.2.1, but specifically uses the bits of the number to identify the bucket into which each number is placed. First the most significant bit is used to place each number into one of two buckets. Then the next most significant bit is used to place each number in each bucket into one of two buckets, and so on until the least significant bit is reached. Reformulate the algorithm to become a pipeline where all the numbers are passed reordered from stage to stage until finally sorted. Write a parallel program for this method and analyze the method.

5-9. A pipeline consists of four stages, as shown in Figure 5.21. Each stage performs the operation

$$y_{\text{out}} = y_{\text{in}} + a \times x$$

Determine the overall computation performed.

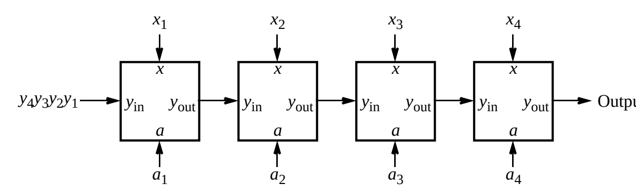

Figure 5.21 Pipeline for Problem 5-9.

5-10. The outer product of two vectors (one-dimensional arrays), A and B, produces a matrix (a two dimensional array), C, as given by

$$AB^T = \underset{A}{\begin{bmatrix} a_0 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}} \underset{B^T}{\begin{bmatrix} b_0 & \cdot & \cdot & b_{n-1} \end{bmatrix}} = \underset{C}{\begin{bmatrix} a_0b_0 & \cdot & \cdot & a_0b_{n-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a_{n-1}b_0 & \cdot & \cdot & a_{n-1}b_{n-1} \end{bmatrix}}$$

Formulate pipeline implementation for this calculation given that the elements of A (a_0, a_1, …, a_{n-1}) enter together from the left of the pipeline and one element of B is stored in one pipeline stage (P_0 stores b_0, P_1 stores b_1, etc.). Write a parallel program for this problem.

5-11. Compare implementing the sieve of Eratosthenes by each of the following ways:

(i) By the pipeline approach as described in Section 5.3.3

(ii) By having each process strike multiples of a single number

(iii) By dividing the range of numbers into m regions and assigning one region to each process to strike out multiples of prime numbers. Use a master process to broadcast each prime number as found to processes

Perform an analysis of each method.

5-12. (For those with knowledge of computer architecture.) Write a parallel program to model a five-stage RISC processor (reduced instruction set computer), as described in Hennessy and Patterson (1990). The program is to accept a list of machine instructions and shows the flow of instructions through the pipeline, including any pipeline stalls due to dependencies/resource conflicts. Use a single valid bit associated with each register to control access to registers, as described in Wilkinson (1996).

Real Life

5-13. As mentioned in Section 5.1, pipelining could be used to implement an audio frequency-amplitude histogram display in a sound system, as shown in Figure 5.22(a). This application could also be implemented by an embarrassingly parallel, functional decomposition, where each process accepts the audio input directly, as shown in Figure 5.22(b). For each method, write a parallel program to produce a frequency-amplitude histogram display using an audio file as input. Analyze both methods. (Some research may be necessary to develop how to recognize frequencies in a digitized signal.)

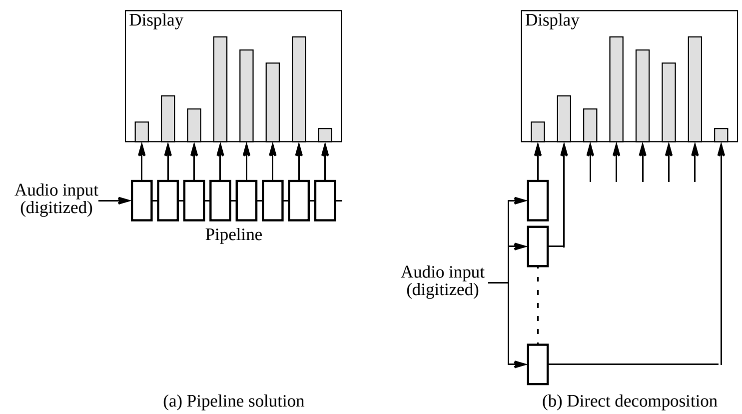

Figure 5.22 Audio histogram display.