CHAPTER 5 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.

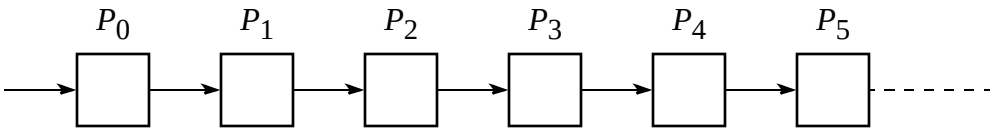


Figure 5.1 Pipelined processes.

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];
    .
```

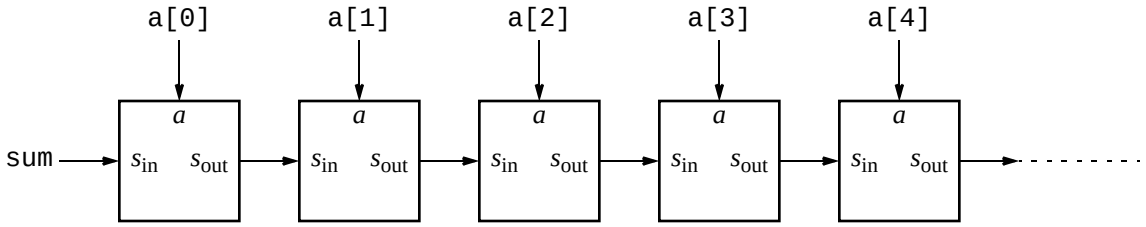


Figure 5.2 Pipeline for an unfolded loop.

Another 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:

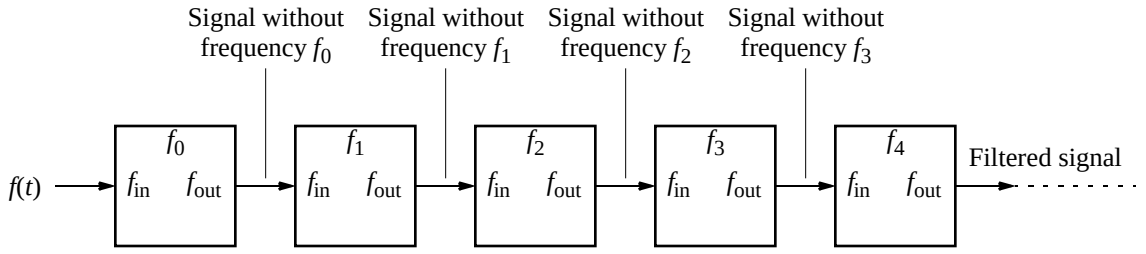


Figure 5.3 Pipeline for a frequency filter.

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

"Type 1" Pipeline Space-Time Diagram

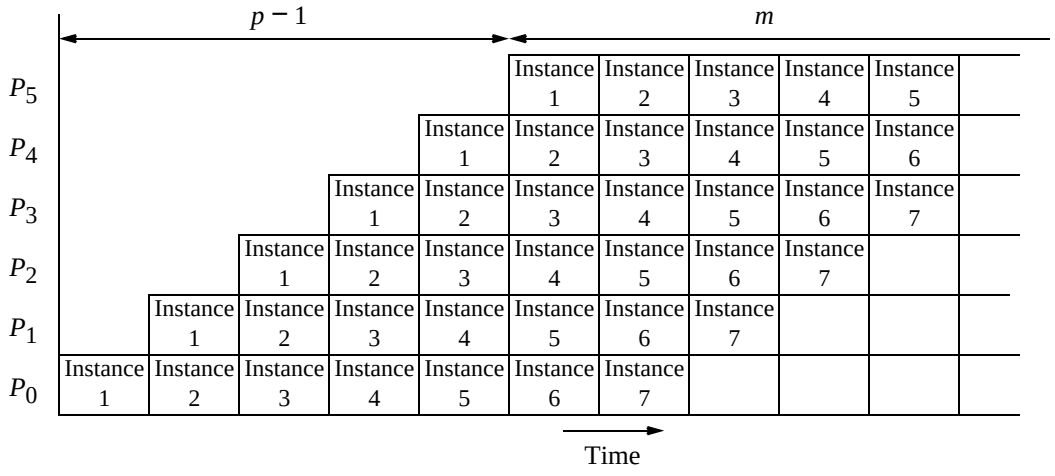


Figure 5.4 Space-time diagram of a pipeline.

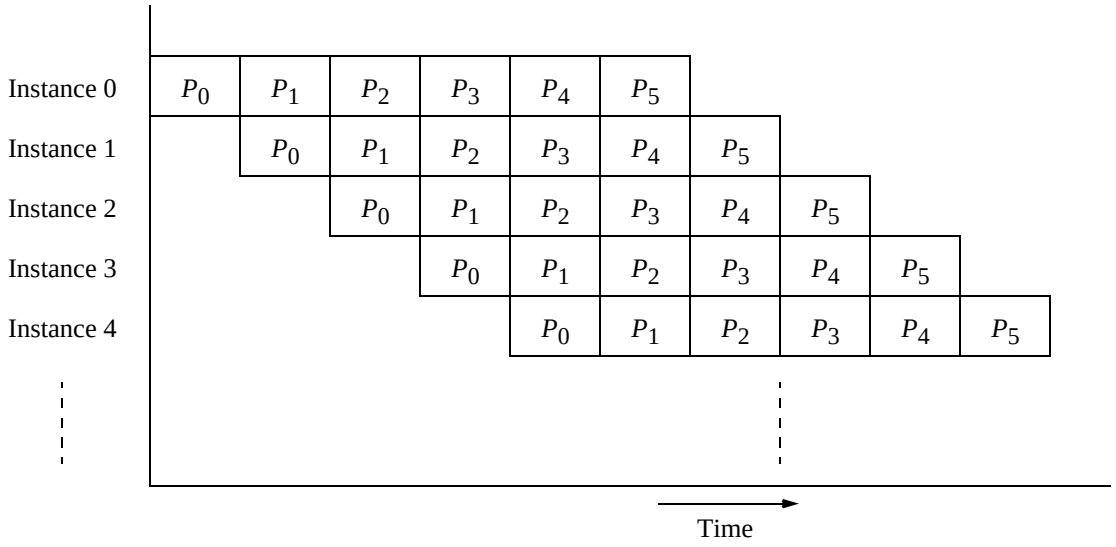
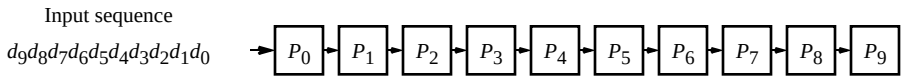
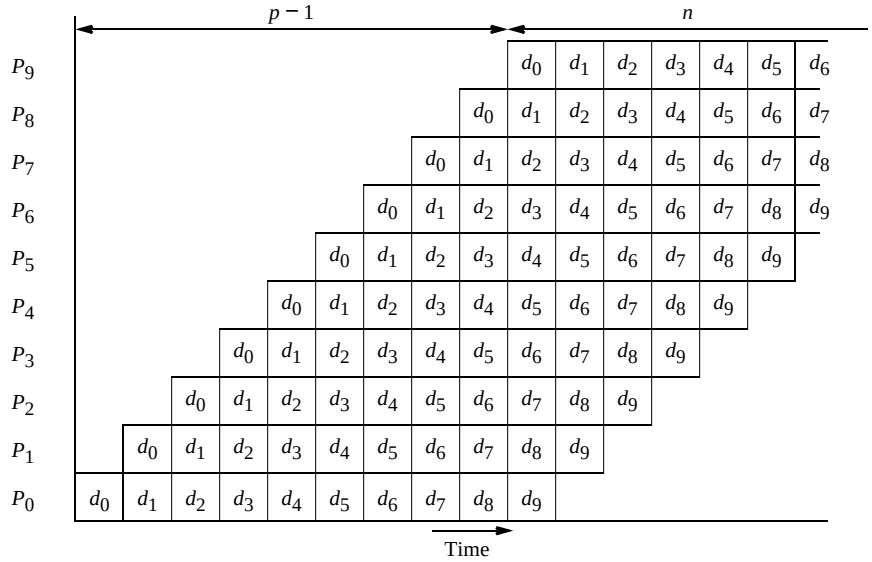


Figure 5.5 Alternative space-time diagram.

"Type 2" Pipeline Space-Time Diagram



(a) Pipeline structure



(b) Timing diagram

Figure 5.6 Pipeline processing 10 data elements.

"Type 3" Pipeline Space-Time Diagram

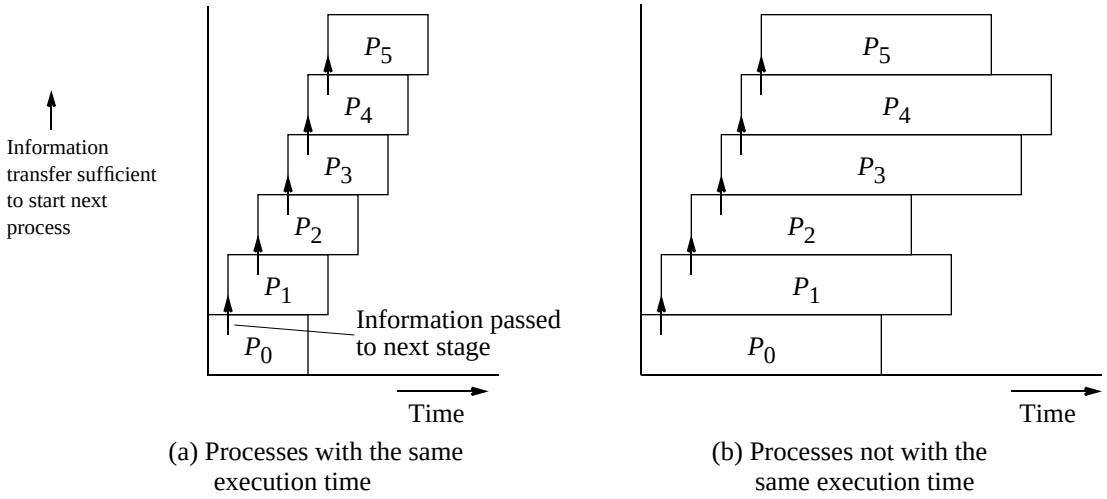


Figure 5.7 Pipeline processing where information passes to next stage before end of process.

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:

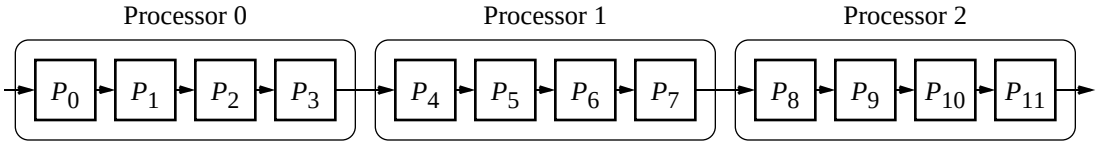


Figure 5.8 Partitioning processes onto processors.

Computing Platform for Pipelined Applications

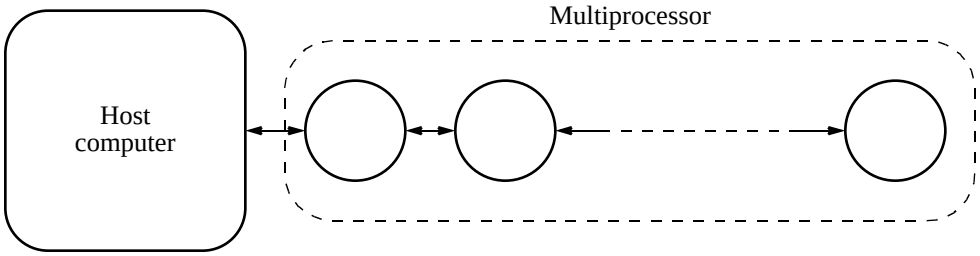


Figure 5.9 Multiprocessor system with a line configuration.

Pipeline Program Examples

Adding Numbers

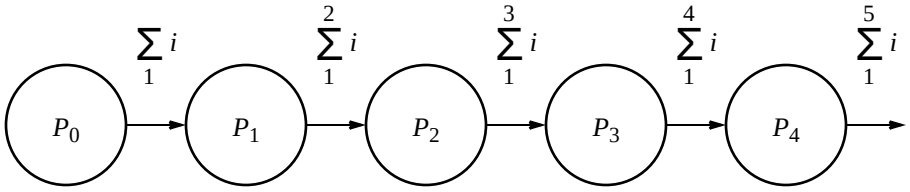


Figure 5.10 Pipelined addition.

The basic code for process P_i :

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

except for the first process, P_0, which is

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

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

```
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, P(i+1));
```

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

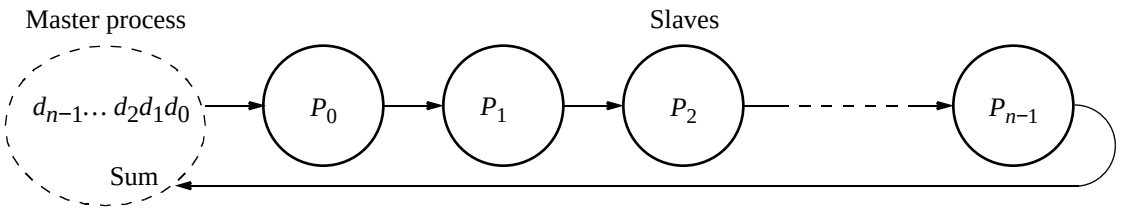


Figure 5.11 Pipelined addition numbers with a master process and ring configuration.

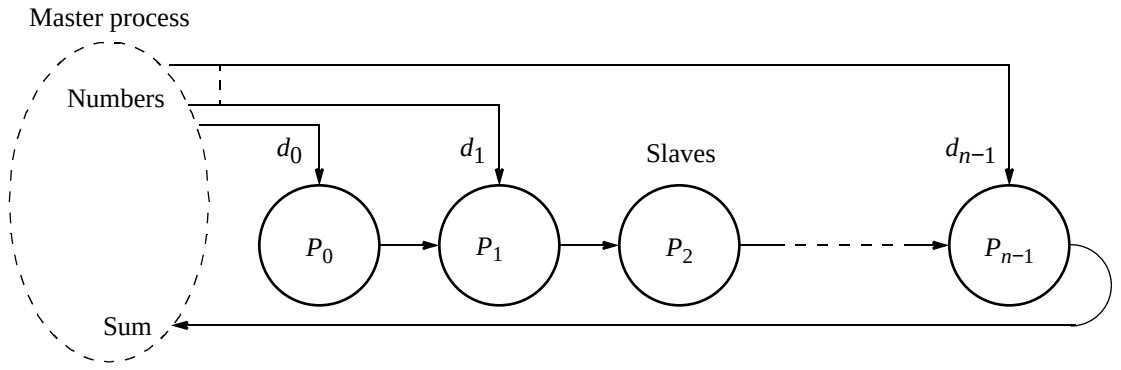


Figure 5.12 Pipelined addition of numbers with direct access to slave processes.

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.

Total execution time

t_{total} = (time for one pipeline cycle)(number of cycles)

$$t_{total} = (t_{comp} + t_{comm})(m + p - 1)$$

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

The average time for a computation is given by

$$t_a = \frac{t_{total}}{m}$$

Single Instance of Problem

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

$$t_{total} = (2(t_{startup} + t_{data}) + 1)n$$

Tme complexity = $O(n)$.

Multiple Instances of Problem

$$t_{total} = (2(t_{startup} + t_{data}) + 1)(m + n - 1)$$

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

That is, one pipeline cycle

Data Partitioning with Multiple Instances of Problem

$$t_{comp} = d$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

$$t_{total} = (2(t_{startup} + t_{data}) + d)(m + n/d - 1)$$

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)

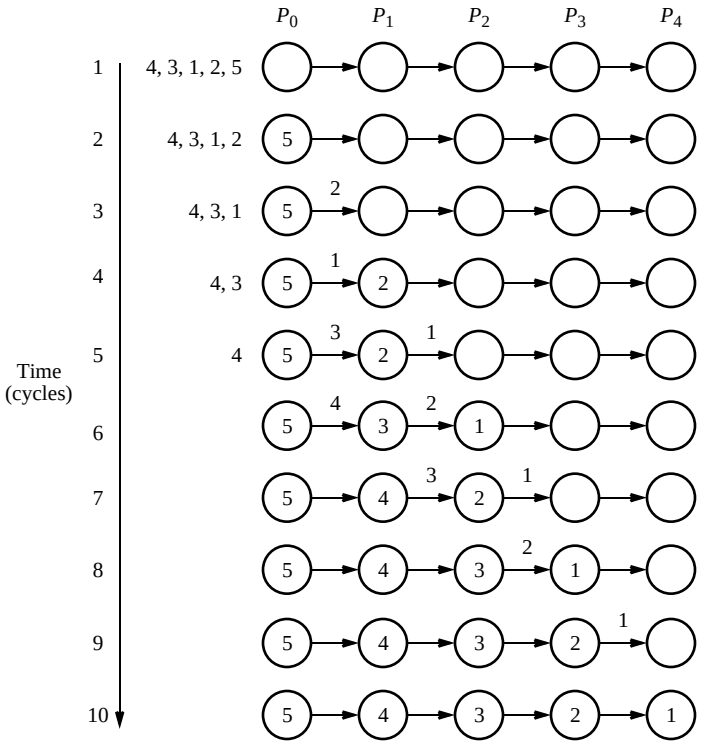


Figure 5.13 Steps in insertion sort with five numbers.

The basic algorithm for process P_i is

```
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 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.

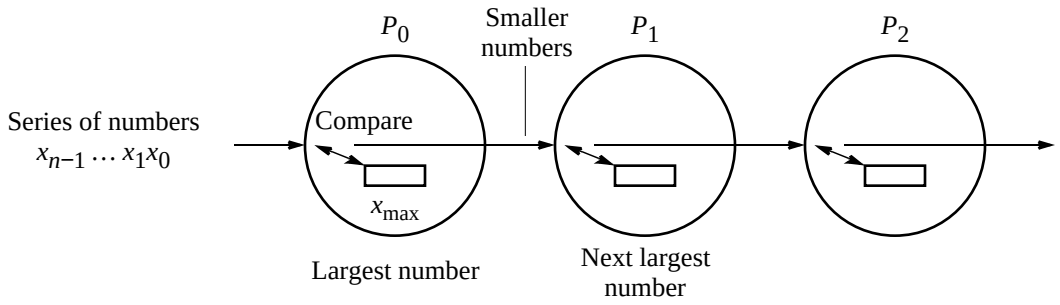


Figure 5.14 Pipeline for sorting using insertion sort.

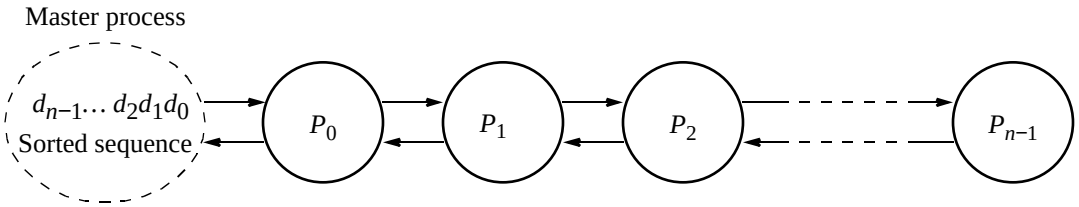


Figure 5.15 Insertion sort with results returned to the master process using a bidirectional line configuration.

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

```
right_procno = n - i - 1;       /*no of processes to the right */
recv(&x, 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 nos */
      recv(&x, P_{i+1});
      send(&x, P_{i-1});
    }
```

Analysis

Sequential

$$t_s = (n-1) + (n-2) + \quad + 2 + 1 = \frac{n(n-1)}{2}$$

Obviously very poor sequential sorting algorithm and unsuitable except for very small *n*.

Parallel

Each pipeline cycle requires at least

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

The total execution time, t_{total}, is given by

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

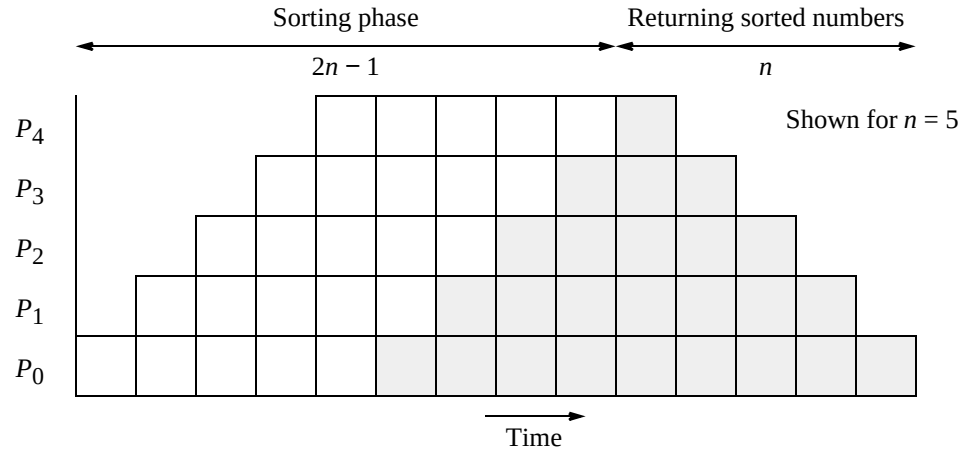


Figure 5.16 Insertion sort with results returned.

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, 4̸, 5, 6̸, 7, 8̸, 9, 1̸0, 11, 1̸2, 13, 1̸4, 15, 1̸6, 17, 1̸8, 19, 2̸0

where numbers with / are not prime and not considered further. After considering 3:

2, 3, 4̸, 5, 6̸, 7, 8̸, 9̸, 1̸0, 11, 1̸2, 13, 1̸4, 1̸5, 1̸6, 17, 1̸8, 19, 2̸0

Subsequent numbers are considered in a similar fashion.

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

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 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 Code
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

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

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

Pipelined Implementation

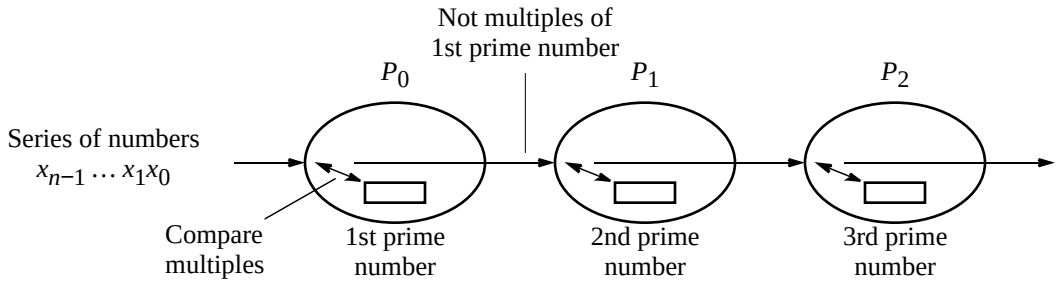


Figure 5.17 Pipeline for sieve of Eratosthenes.

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);
```

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:

```
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

Type 3 example - process can continue with useful work after passing on information.

To solve system of linear equations of the so-called *upper-triangular* form:

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

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.,

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

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.

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.

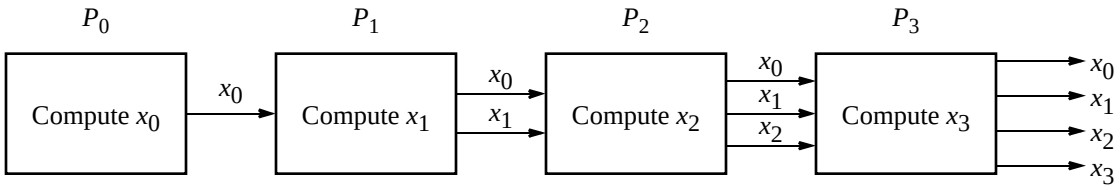


Figure 5.18 Solving an upper triangular set of linear equation using a pipeline.

The ith process ($0 < i < n$) receives the values x_0, x_1, x_2, …, 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 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);
```

Now we have additional computations to do after receiving and resending values.

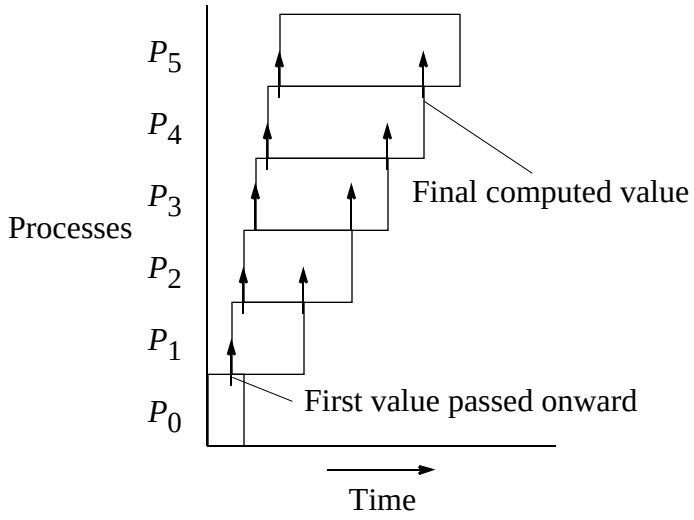


Figure 5.19 Pipeline processing using back substitution.

Analysis

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.

Intentionally blank