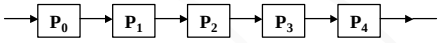
Chapter 5 – Pipelined Computations

Applicable to a wide range of problems that are partially sequential in nature; i.e., sequence of steps must be undertaken.

In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other - a form of "functional parallelism"

Pipeline stage: each pipeline process (that contributes to the overall problem and pass on info needed for subsequent stages.)

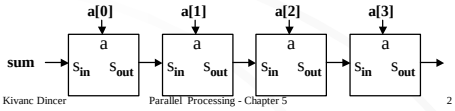



```
for (i=0; i<n; i++)
    sum += a[i];
```

unfold

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

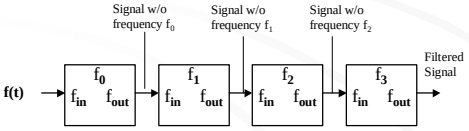
Stage i performs: sout = sin + a[i];



Example: Frequency Filter

- Problem is divided into a series of functions
- The objective is to remove specific frequencies (say f_1, f_2, f_3, …) from a (digitized) signal, f(t).
- The signal enter the pipeline from the left
- Each stage is responsible for removing one of the frequencies.

Frequency-amplitude histogram in professional sound systems is a similar example.



Applicable Types of Computations

1. If more than one instance of the complete problem is to be executed.
   - ♦ internal hardware design, simulation exercises
2. If a series of data items must be processed, each requiring multiple operations.
   - ♦ array operations

   With *p* processes and *n* data items, overall execution time is *(p-1) + n*
3. If info to start the next process can be passed forward before the process has completed all its internal operations.

If number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor.

Type 1

Pipeline cycle: each time period in space-time diagram.

Note the staircase effect at the beginning.

With *p* processes constituting the pipeline and *m* instances of the problem:

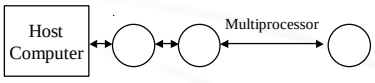
m+p-1 cycles are required to execute all *m* instances.

Average number of cycles: *(m+p-1)/m*

One instance of the problem is completed in each pipeline cycle after the n-1 cycles (pipeline latency)

Computing Platform for Pipelined Applications

- Ability to send messages between adjacent processes in the pipeline
  - direct comm. links -- ring or line is ideal structure.
- Networked workstations on Ethernet may not really be a suitable platform for the pipelined programs.



We will assume that an ICNW that can provide at least simultaneous transfers between adjacent processors.

- locally blocking **send()** operations can be used.

Adding Numbers (Type 1)

- Basic code for process P_i:

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

Two ways of distributing data: (Fig.5.11 & 12)

1. Data being entered into the first process, the result is returned through the last process.
2. Data is fed into each process at the times that they are needed by the processes.

Analysis

- Assume that each process performs similar actions in each pipeline cycle.
- We will work on comp. and comm. required in one cycle.

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

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

m: number of instances of the problem
p: pipeline stages (processes)

The average time for a computation: $t_a = t_{total} / m$

Single Instance of Problem: (Figure 5.11)

Single number is being added in each stage: *n==p*

$t_{comp} = 1$

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

When only one set of numbers, m=1:

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

Multiple Instances of Problem:

m groups of *n* numbers are being added:

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

For large *m*, $t_a = t_{total}/m = 2\ (t_{startup} + t_{data}) + 1$

Data Partitioning with Multiple Instance of Problem:

Each stage will process a group of *d* numbers

$p = n/d$

$t_{comp} = d$

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

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

As d increases,

- *the impact of the comm. on the overall time diminishes*
- *parallelism decreases and increases the execution time.*

Sorting Numbers (Type 2)

reorder a set of numbers in increasing (or decreasing) numeric order.

A pipeline solution: (Parallel Insertion Sort)

- The first process, P_0, accepts the series of numbers one at a time.
- Stores the largest number so far received, and pass onward all numbers smaller than the stored number.

Parallel Code for P_i.

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

With n numbers:

```
right_procno = n - i -1;
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);
```

A message-passing program using an SPMD or a master-slave approach is straightforward

- especially since each pipeline process executes essentially the same code.

Results can be extracted from the pipeline using

- either the ring configuration of Fig.5.11 or
- the bidirectional line configuration of Fig.5.15.


```
right_procno = n - i -1;
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);
for (j=0; j<right_procno; j++) {
  recv(&number, P i+1);
  send(&x, P i-1);
}
```


Analysis.

Assuming that compare-and-exchange is one computational step:

$t_s = (n-1) + (n-2) + \ldots + 2 + 1 =$

The parallel impl. has $n+n-1 = 2n-1$ pipeline cycles during the sorting if there are n pipeline processes and n numbers to sort.

$t_{comp} = 1$

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

If the results are returned by comm. to the left through the master, $3n-1$ pipeline cycles are needed.

Prime Number Generation

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

To find the primes up to n, it is only necessary to look at numbers up to $sqrt(n)$.

Sequential Code

```
for (i=2; i<n; i++)
  prime[i] = 1;
for (i=2; i<=sqrt(n); i++)
  if (prime[i] == 1)
     for (j=i+i; j<n; j+=i)
        prime[j] = 0;
```

There are [n/2-1] multiples of 2, [n/3-1] multiples of 3, so on. Hence, total sequential time is ... with O(n) complexity

Parallel Code.

```
recv(&x, P i-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 (Type 3)

A process can continue with useful work after passing on information.

Objective: solving a system of linear equations of the *upper-triangular* form.

Method: simple repeated *"back" substitution*.

The ith process ($0<i<n$) receives the values $x_0, x_1, \ldots, x_{i-1}$ and computes x_i from the equation:

$$x_i = \left(b_i - \sum_{j=0}^{i-1} a_{i,j} x_j \right) / a_{i,i}$$

Sequential Code.

```
x[0] = b[0]/a[0][0];
for (i=1; i<n; i++) {
  sum = 0;
  for (j=0; j<i; j++) {
      sum += a[i][j] * x[j];
      x[i] = (b[i] - sum) / a[i][i];
  }
}
```

Parallel Code.

```
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 += a[i][j] * x[j];
x[i] = (b[i] - sum) / a[i][i];
send(&x[i], P_{i+1});
```


Analysis.

Computational effort at each pipeline stage is not the same.

P_0 performs one divide and one send() .

P_i performs

- *i* recv()
- *i* send()
- *i* multiply/add
- *1* divide/subtract
- *1* final send()

} a total of *n-1* comm. and *2n-1* comp. steps

Figure 5.20: a perfect sync. of the sends and recvs

T_p = time of final process + *n-1* sends and *1* divide.

O(n) - parallel time complexity

$O(n^2)$ - serial time complexity

Actual speedup is 0.37n (Lester, 1993)

- Nonblocking sends and blocking receives are used in implementation.