L7 - S5 Pipelined Computations

什么辣鸡？？

Pipeline Computations

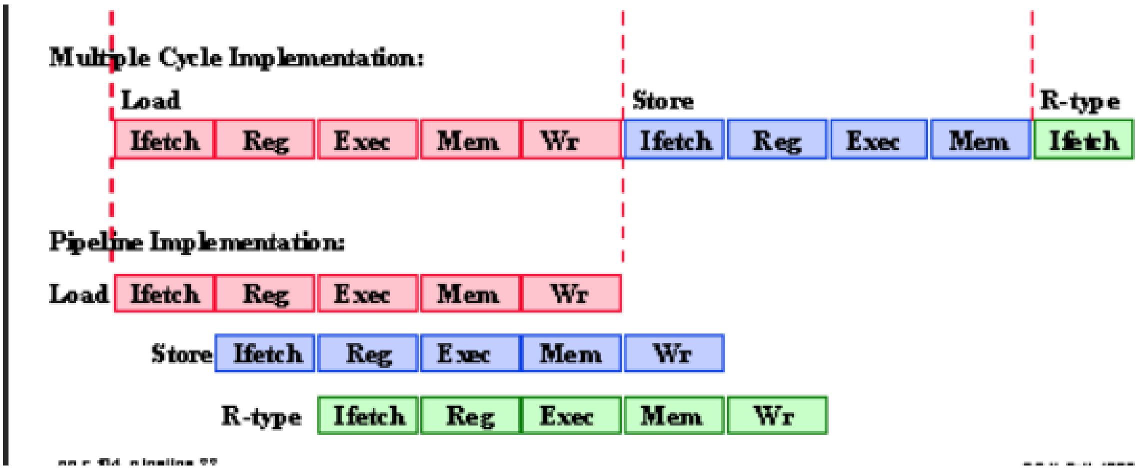
- In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other.

Three Types of Pipelined Computations

- 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

Type -1

- If more than one instance of the complete problems is to be executed



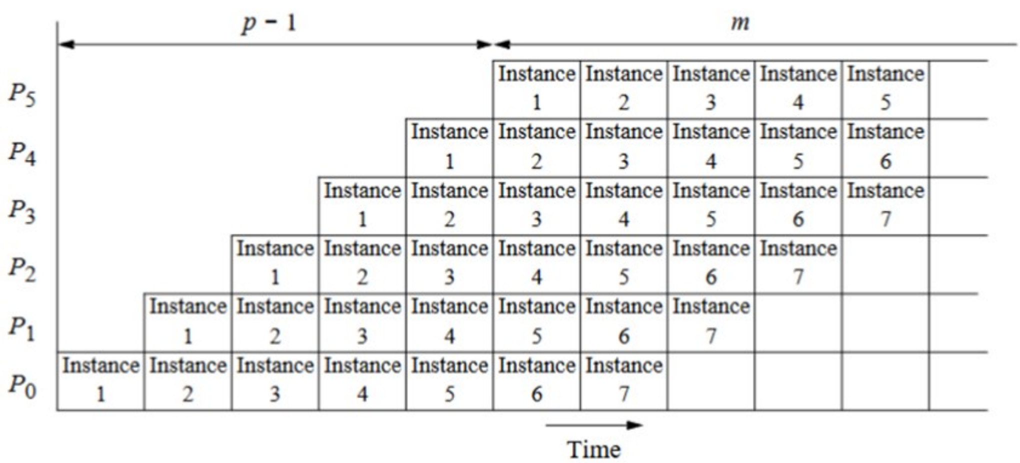


Figure 5.4 Space-time diagram of a pipeline.

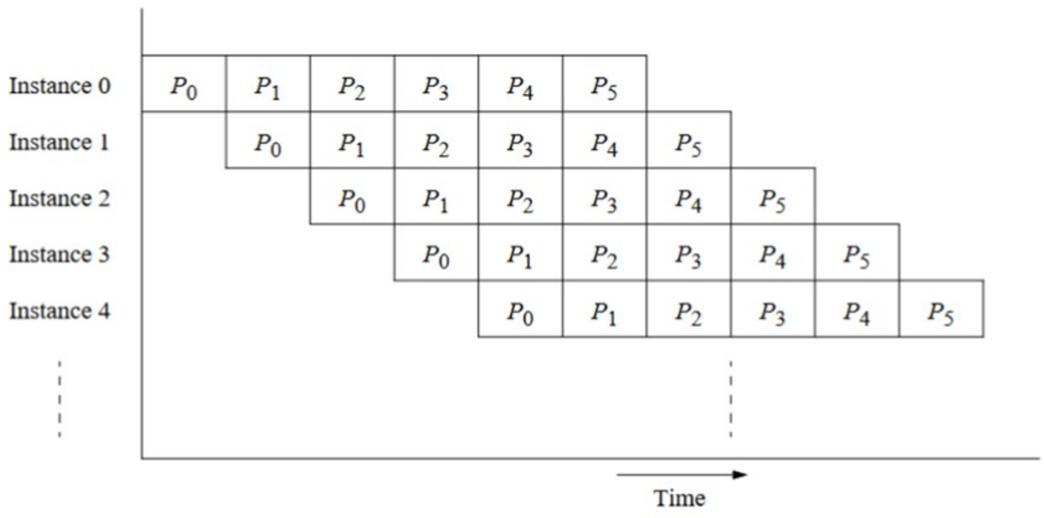


Figure 5.5 Alternative space-time diagram.

- Analysis

Total execution time

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

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

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

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

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

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.

Type - 2

- If a series of data items must be processed, each requiring multiple operations

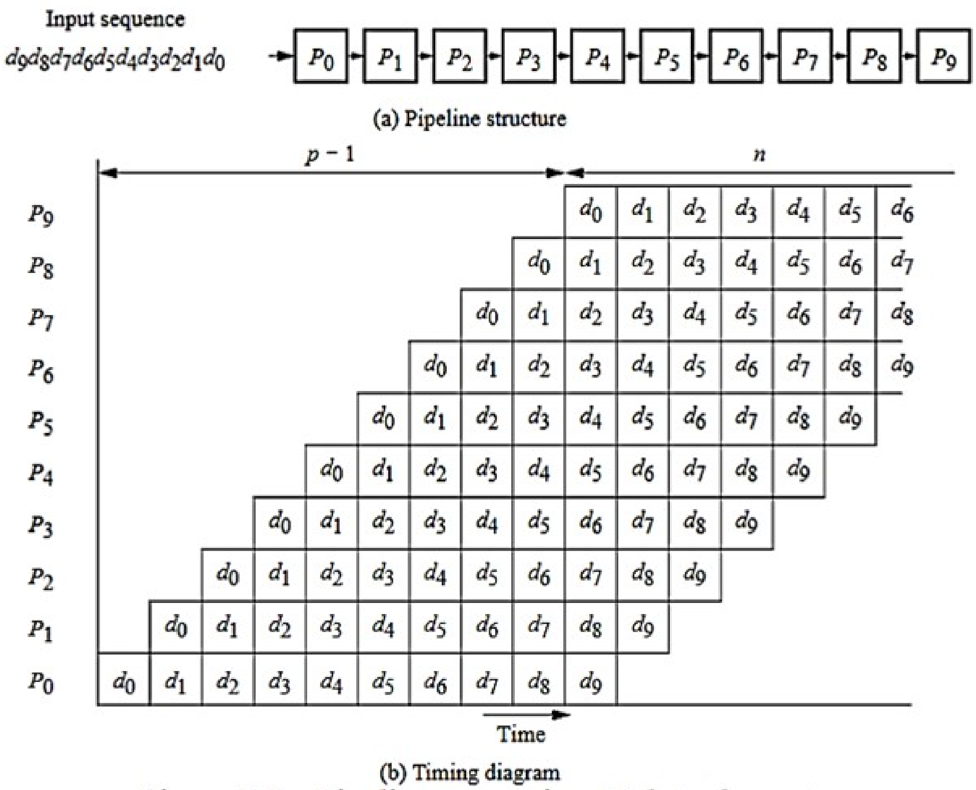


Figure 5.6 Pipeline processing 10 data elements.

Type - 3

- If information to start the next process can be passed forward before the process has completed all its internal operations

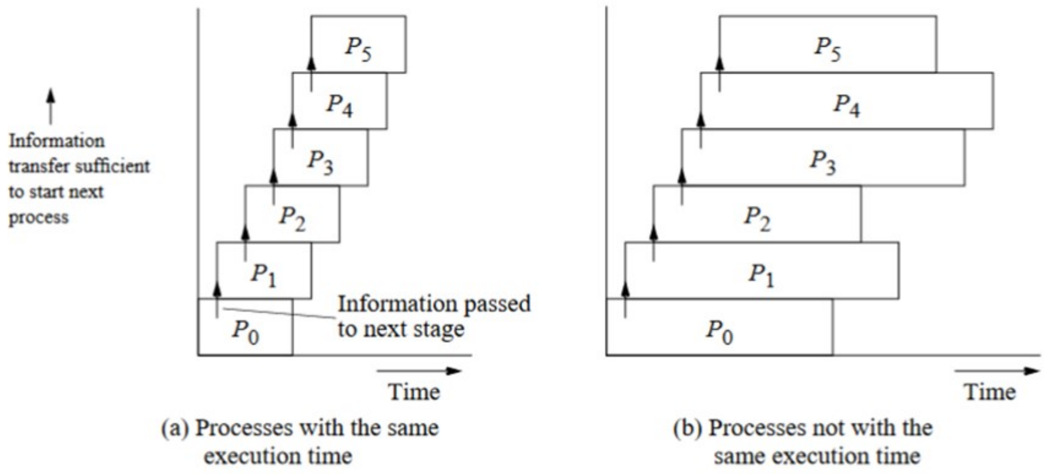


Figure 5.7 Pipeline processing where information passes to next stage before end of process.

Partitioning of Pipelined Computations

- 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

Pipeline Program Examples - Adding Numbers

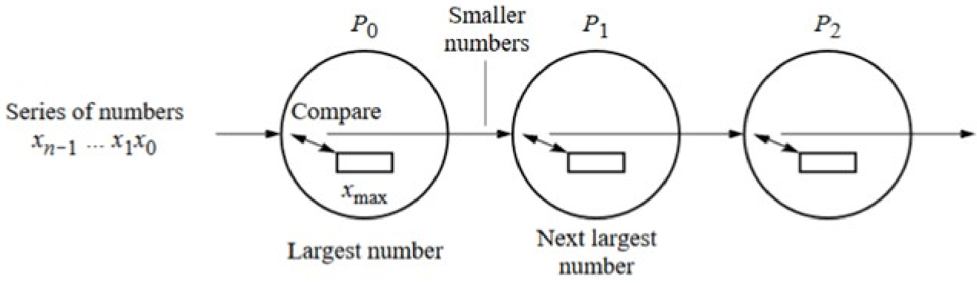
SPMD program

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

Sorting Numbers

- basic algorithm

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



Prime Number Generation

L8 - S6 Synchronous Computations

Synchronous Computations

- In a (fully) synchronous application, all the processes synchronized at regular points.

MPI

- `MPI_Barrier()` with a named communicator

PVM

- `pvm_barrier()` with a named group of processes
  - unusual features: specifying the number of processes that must reach the barrier to release the processes

Counter Implementation

- centralized counter implementation (sometimes called a *linear barrier*)
- two phases:
  - arrival phase (does not leave the phase until all processes have arrived), and

- departure phase (are released)

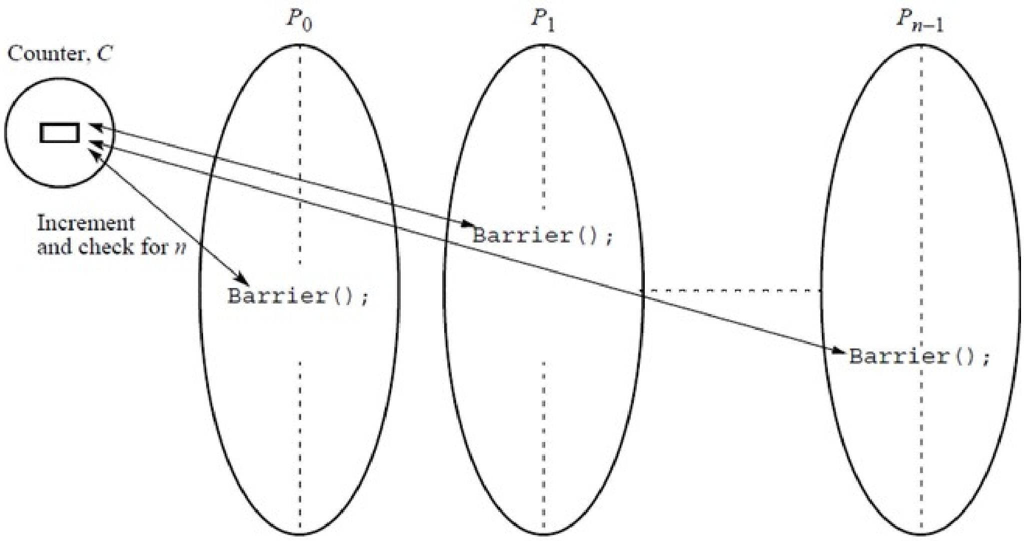


Figure 6.3 Barrier using a centralized counter.

Master:

```
for (i = 0; i < n; i++)/*count slaves as they reach barrier*/
    recv(P_any);
for (i = 0; i < n; i++)/* release slaves */
    send(P_i);
```

Slave processes:

```
send(P_master);
recv(P_master);
```

Tree Implementation

- suppose 8 processes.

First stage: P_1 sends message to P_0; (when P_1 reaches its barrier)
P_3 sends message to P_2; (when P_3 reaches its barrier)
P_5 sends message to P_4; (when P_5 reaches its barrier)
P_7 sends message to P_6; (when P_7 reaches its barrier)
Second stage: P_2 sends message to P_0; (P_2 and P_3 have reached their barrier)
P_6 sends message to P_4; (P_6 and P_7 have reached their barrier)
Third stage: P_4 sends message to P_0; (P_4, P_5, P_6, and P_7 have reached their barrier)
P_0 terminates arrival phase; (when P_0 reaches barrier and has received message from P_4)

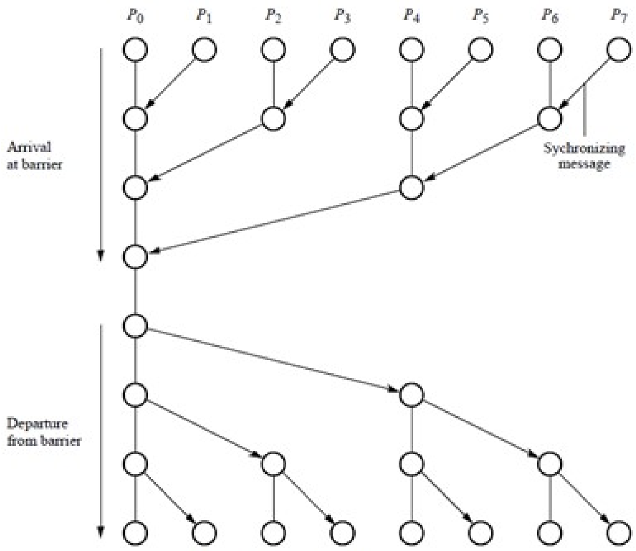


Figure 6.5 Tree barrier.

Bufferfly Barrier

First stage $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
Second stage $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
Third stage $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

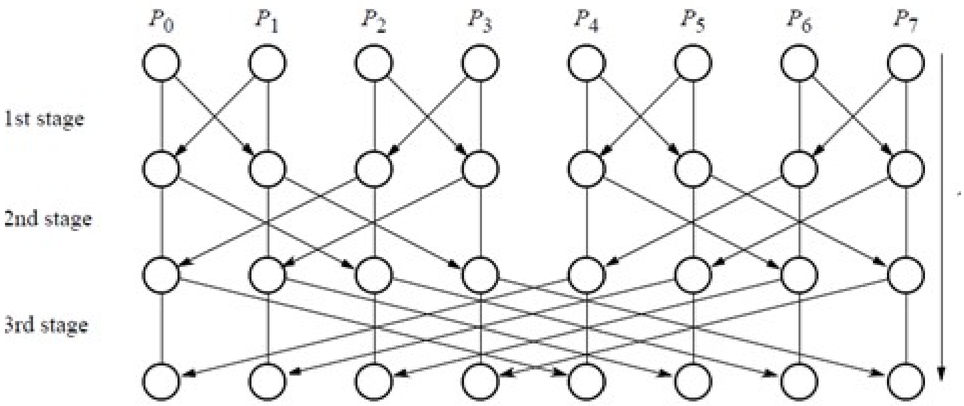


Figure 6.6 Butterfly construction.

Local Synchronization

- $P_{i-1}4willonlysynchronizewithP_i,P_{i+1}, too

Process P_{i-1}	Process P_i	Process P_{i+1}
recv(P_i);	send(P_{i-1});	recv(P_i);
send(P_i);	send(P_{i+1});	send(P_i);
	recv(P_{i-1});	
	recv(P_{i+1});	

Deadlock

- It will occur if both processes perform the send, using synchronous routines first (or, blocking routines without sufficient buffering). This because neither will return, they will wait for matching receives that are never reached

- Solution
    - arrange for one process to receive first and then send and other process to send first and then receive.

Combined deadlock-free blocking sendrecv() routines

- `MPI_Sendrecv()`

Data Parallel Computations

- Same operation performed on different data elements simultaneously.

Forall Construction

- `forall (i = 0; i < n; i++) {code}`
    - n instances of the statements of the body can be executed simultaneously

Prefix Sum Problem

- sequential code

```
for(i = 0; i < n; i++) {
  sum[i] = 0;
  for (j = 0; j <= i; j++)
    sum[i] = sum[i] + x[j];
}
```

Data parallel method of adding all partial sums of 16 numbers

- 用 `forall`

Synchronous Iteration (Synchronous Parallelism)

- Each iteration composed of several processes that start together at beginning of iteration and next iteration can't begin until all processes have finished previous iteration.

```
for (j = 0; j < n; j++)     /*for each synch. iteration */
    forall (i = 0; i < N; i++) { /*N procs each executing */
      body(i);                  /*using specific value of i */
    }
```

or:

```
for (j = 0; j < n; j++) {  /*for each synchr.iteration */
    i = myrank;             /*find value of i to be used */
    body(i);                /*using specific value of i */
      barrier(mygroup);
    }
```

Safety and Deadlock

When all processes send their messages first and then receive all of their messages, as in all the code so far, is described as "unsafe" in the MPI literature because it relies upon buffering in the `send()`s. The amount of buffering is not specified in MPI.

If a send routine has insufficient storage available when it is called, the implementation should be such to delay the routine from returning until storage becomes available or until the message can be sent without buffering.

Hence, the locally blocking `send()` could behave as a synchronous `send()`, only returning when the matching `recv()` is executed. Since a matching `recv()` would never be executed if all the `send()`s are synchronous, deadlock would occur.

Make the code safe

分为奇偶myid决定send和recv的顺序

```
if ((myid % 2) == 0) {
   send(&g[1][1], &m, P-1);
   recv(&h[1][0], &m, P-1);
   send(&g[1,m], &m, P+1);
   recv(&h[1][m+1], &m, P+1)
} else {
   recv(&h[1][0], &m, P-1);
   send(&g[1][1], &m, P-1);
   recv(&h[1][m+1], &m, P+1)
   send(&g[1,m], &m, P+1);
}
```

MPI Safe message Passing Routines

- `MPI_Sendrecv()`

- `MPI_Bsend()` buffered, provides explicit storage
- `MPI_Ised()` `MPI_Irecv()` nonblocking, returns immediately, 配合 `wait` 等使用

  A pseudocode segment using the third method is

  ```
  isend(&g[1][1], &m, P_{i-1});
  isend(&g[1,m], &m, P_{i+1});
  irecv(&h[1][0], &m, P_{i-1});
  irecv(&h[1][m+1], &m, P_{i+1});
  waitall(4);
  ```

Cellular Automata

- Problem space is first divided into cells, each cell can be in one of a finite number of states.
- Cells are affected by their neigbors according to certain rules, and all cells are affected simultaneously in a "generation"