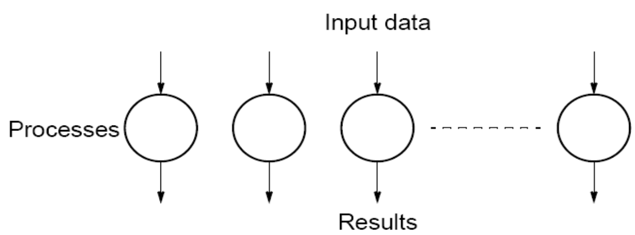
Parallel Techniques

- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Asynchronous Computations
- Strategies that achieve load balancing

Embarrassingly Parallel Computations

Embarrassingly Parallel Computations

A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or).

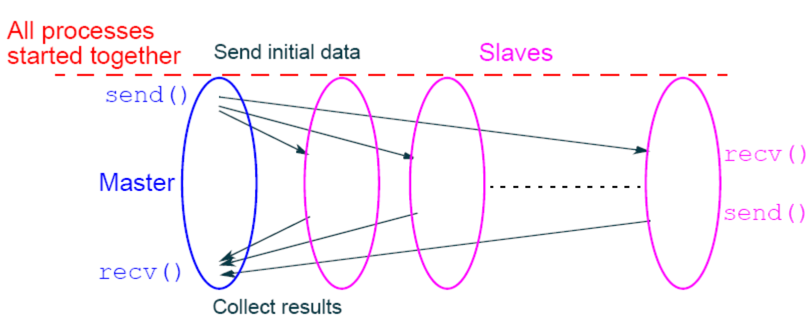


No communication or very little communication between processes

Each process can do its tasks without any interaction with other processes

3.3

Practical embarrassingly parallel computation with static process creation and master-slave approach



Usual MPI approach

3.4

Embarrassingly Parallel Computation Examples

- Low level image processing
- Mandelbrot set
- Monte Carlo Calculations

Parallelizing Mandelbrot Set Computation

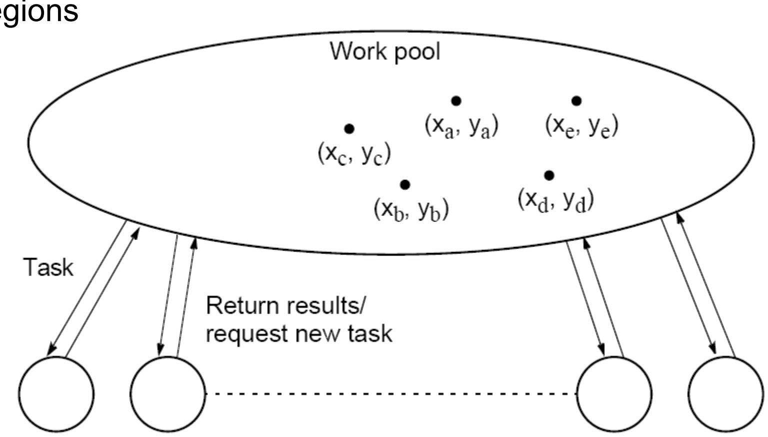
Static Task Assignment

Simply divide the region in to fixed number of parts, each computed by a separate processor.

Not very successful because different regions require different numbers of iterations and time.

Dynamic Task Assignment

Have processor request regions after computing previous regions



3.14

Chapter 4

Partitioning and Divide-and-Conquer Strategies

Partitioning

Partitioning simply divides the problem into parts.

Divide and Conquer

Characterized by dividing problem into sub-problems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.

Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts. Also usually data is naturally localized.

4.1

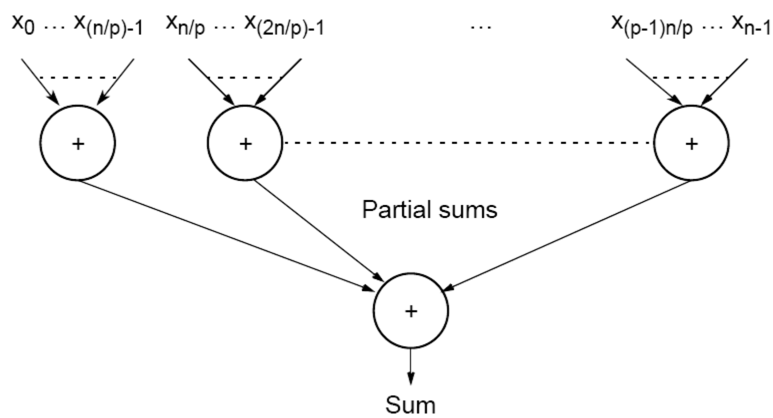
Partitioning/Divide and Conquer Examples

Many possibilities.

- Operations on sequences of number such as simply adding them together
- Several sorting algorithms can often be partitioned or constructed in a recursive fashion
- Numerical integration
- *N*-body problem

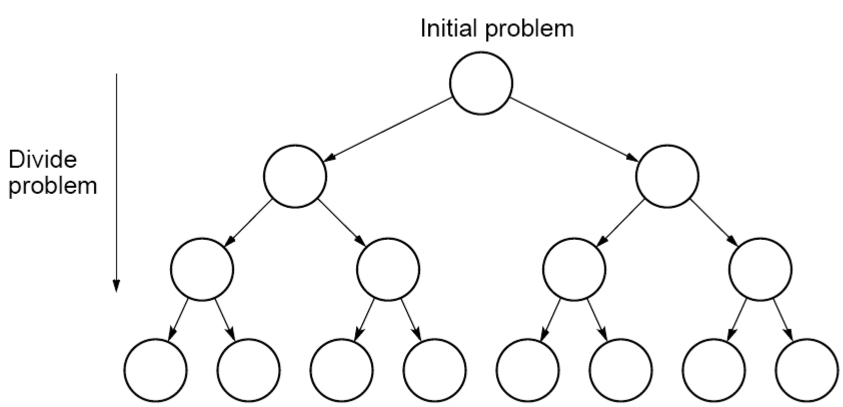
4.2

Partitioning a sequence of numbers into parts and adding the parts



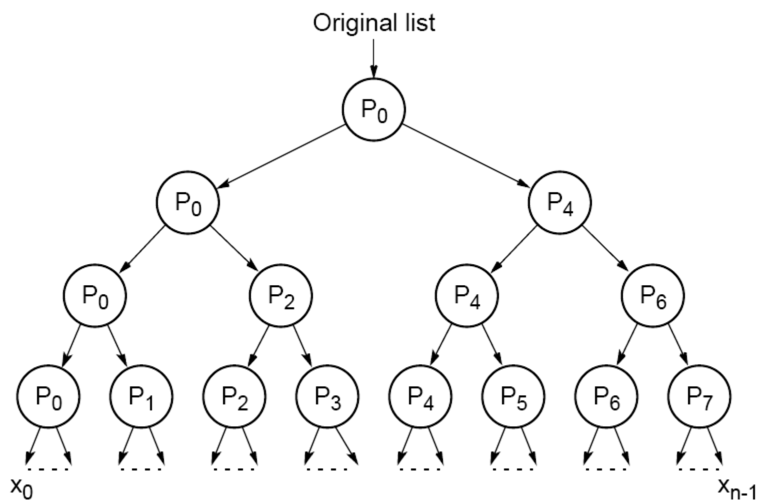
4.3

Tree construction



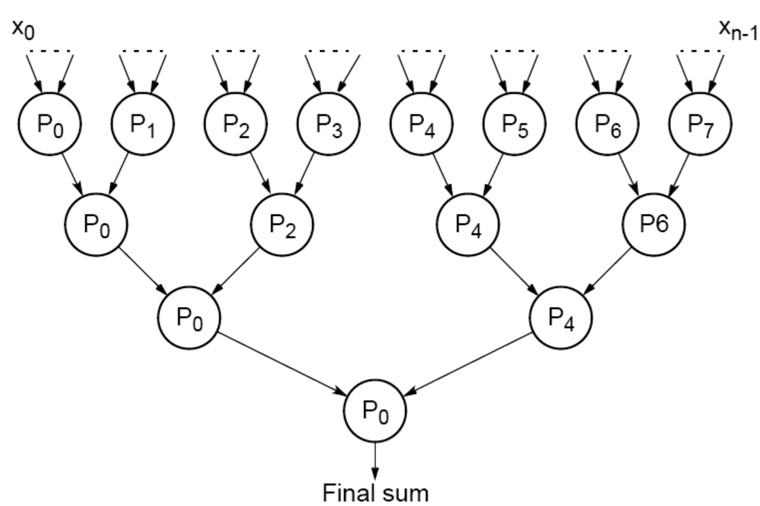
4.4

Dividing a list into parts



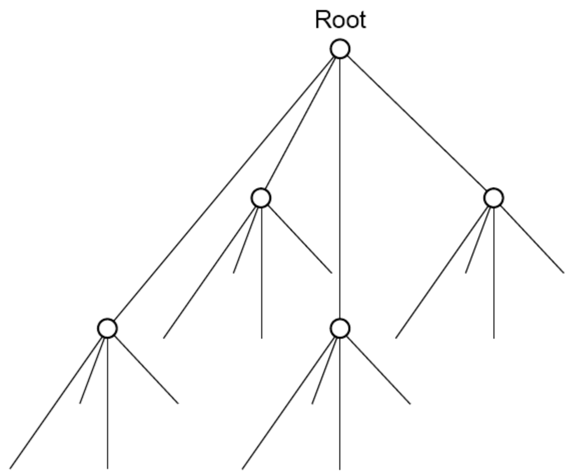
4.5

Partial summation



4.6

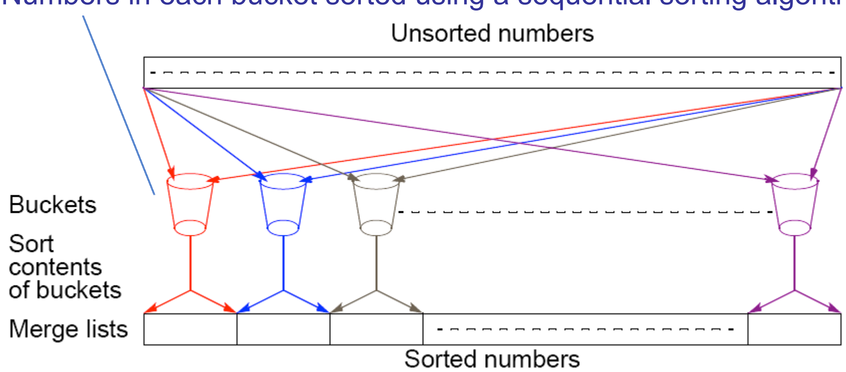
Quadtree



4.7

Bucket sort

One "bucket" assigned to hold numbers that fall within each region.
Numbers in each bucket sorted using a sequential sorting algorithm.



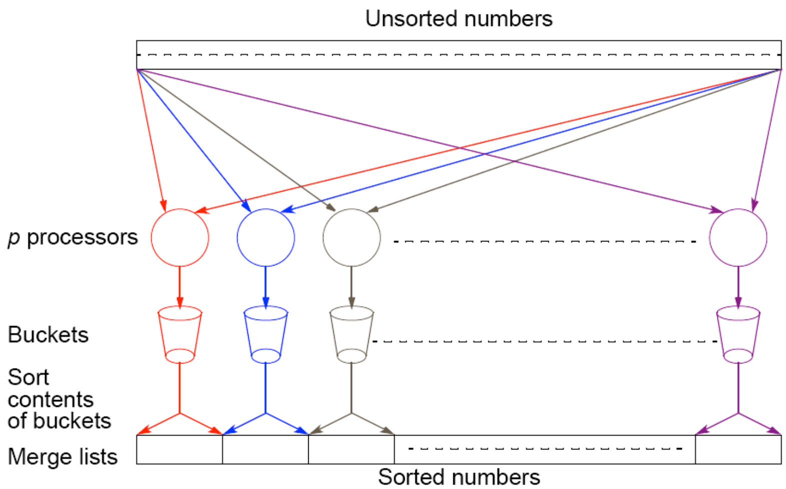
Sequential sorting time complexity: O(nlog(n/m)).
Works well if the original numbers uniformly distributed across a known interval, say 0 to *a* - 1.

4.9

Parallel version of bucket sort

Simple approach

Assign one processor for each bucket.



4.10

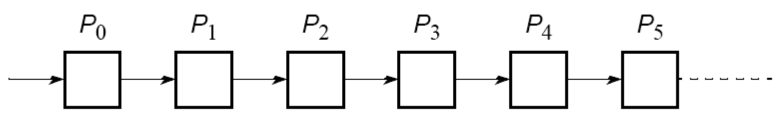
Chapter 5

Pipelined Computations

5.1

Pipelined Computations

Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.



5.2

Example

Add all the elements of array **a** to an accumulating sum:

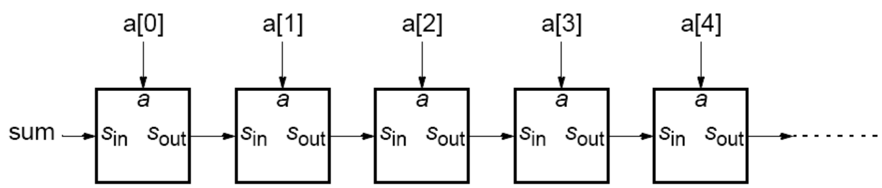
```
for (i = 0; i < n; i++)
sum = sum + a[i];
```

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];
        .
        .
        .
```

5.3

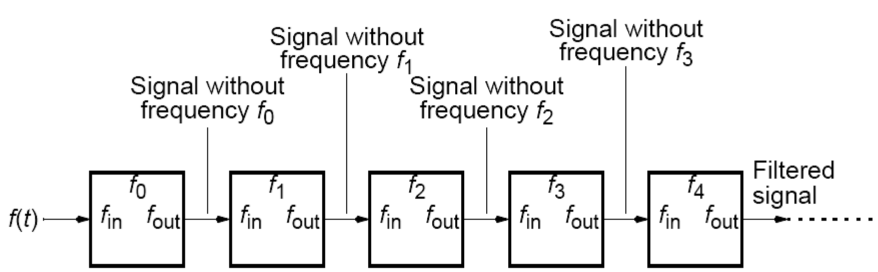
Pipeline for an unfolded loop



5.4

Another Example

Frequency filter - Objective to remove specific frequencies (f_0, f_1, f_2,f_3, etc.) from a digitized signal, $f(t)$.
Signal enters pipeline from left:



5.5

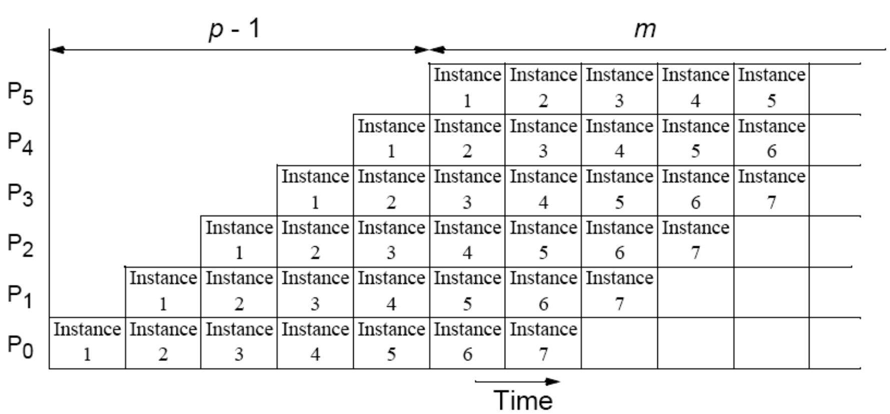
Where pipelining can be used to good effect

Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution 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 next process can be passed forward before process has completed all its internal operations

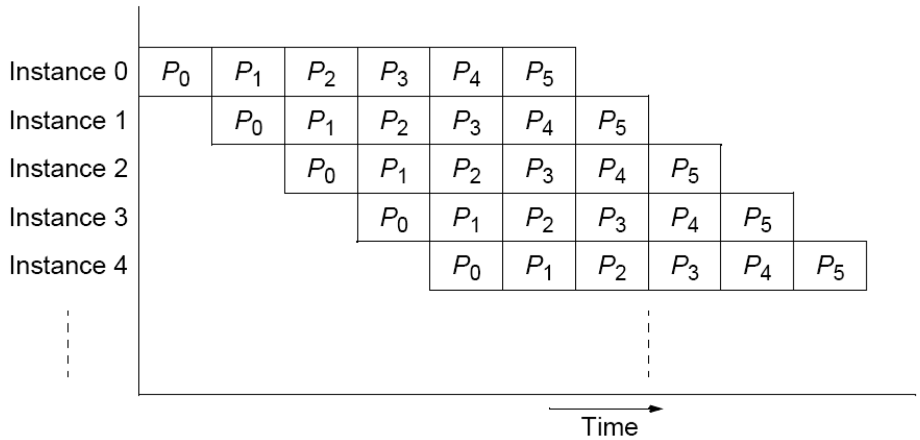
5.6

"Type 1" Pipeline Space-Time Diagram



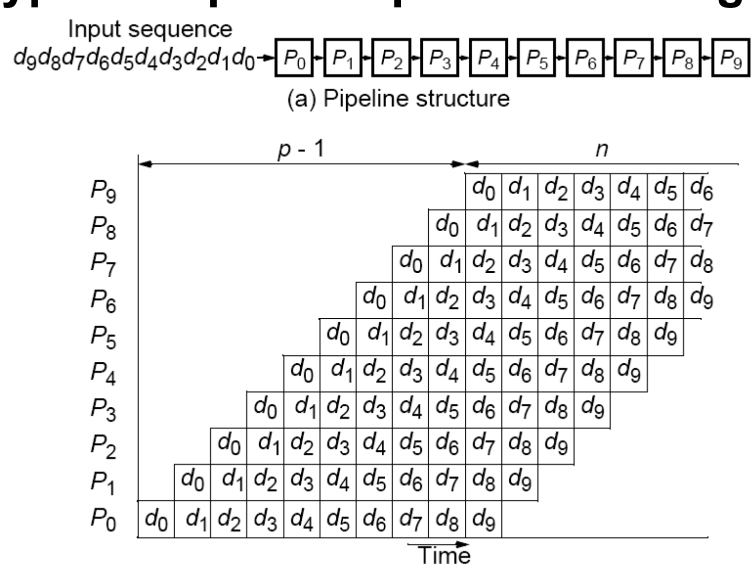
5.7

Alternative space-time diagram



5.8

"Type 2" Pipeline Space-Time Diagram

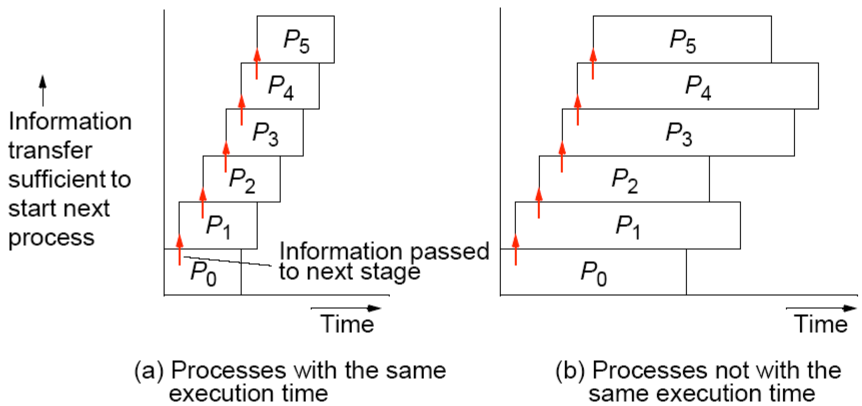


(a) Pipeline structure

(b) Timing diagram

5.9

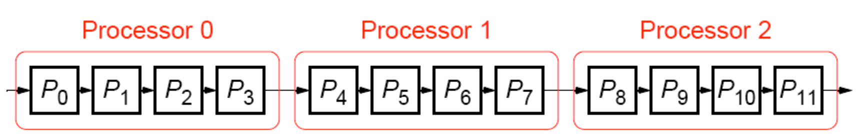
"Type 3" Pipeline Space-Time Diagram



Pipeline processing where information passes to next stage before previous state completed.

5.10

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:



5.11

Chapter 6

Synchronous Computations

6.1

Synchronous Computations

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

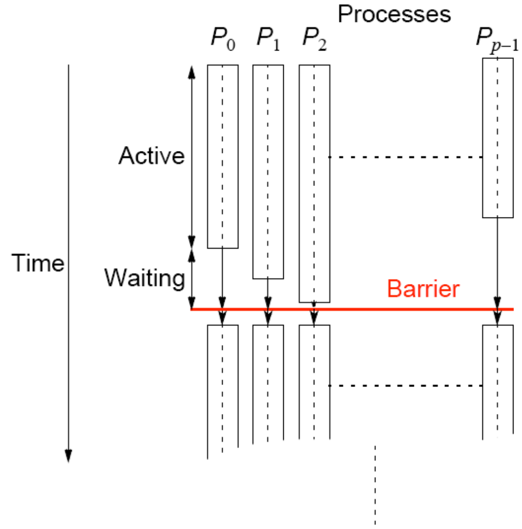
Barrier

A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait.

All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).

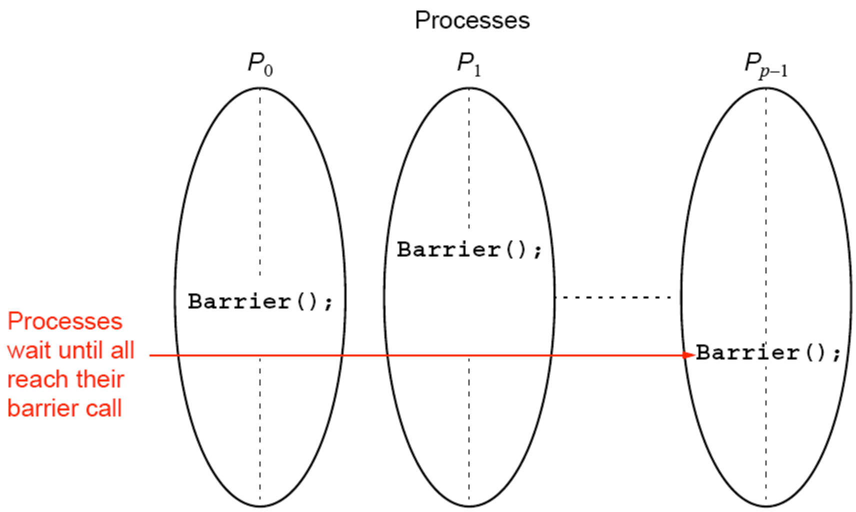
6.2

Processes reaching barrier at different times



6.3

In message-passing systems, barriers provided with library routines:



6.4

MPI
MPI_Barrier()

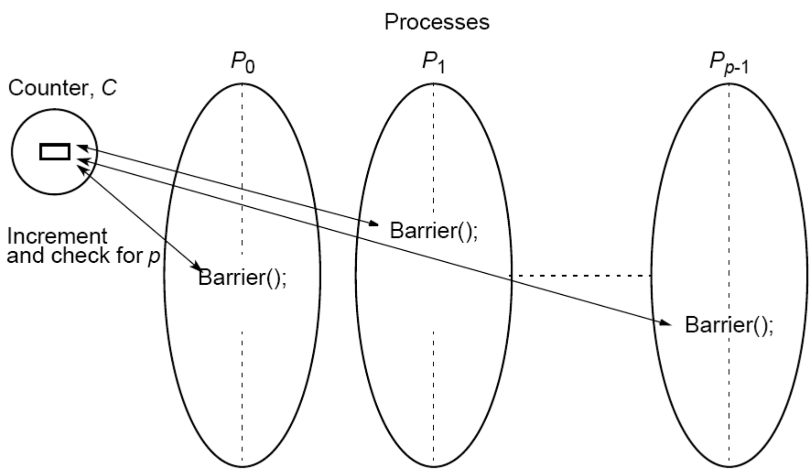
Barrier with a named communicator being the only parameter.

Called by each process in the group, blocking until all members of the group have reached the barrier call and only returning then.

6.5

Barrier Implementation

Centralized counter implementation (a *linear barrier*):



6.6

Good barrier implementations must take into account that a barrier might be used more than once in a process.

Might be possible for a process to enter the barrier for a second time before previous processes have left the barrier for the first time.

6.7

Counter-based barriers often have two phases:

- A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.

- Then processes move to departure phase and are released.

Two-phase handles the reentrant scenario.

6.8

Example code:

Master:

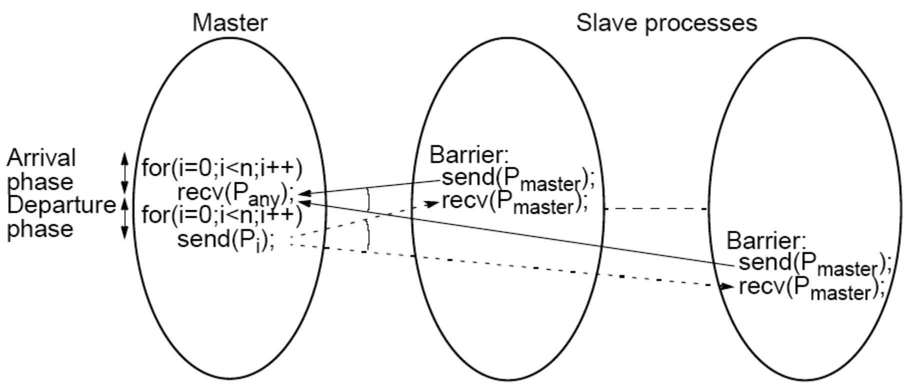
```
for (i = 0; i < n; i++)       /*count slaves as they reach barrier*/
        recv(Pany);
for (i = 0; i < n; i++)      /* release slaves */
        send(Pi);
```

Slave processes:

```
send(Pmaster);
recv(Pmaster);
```

6.9

Barrier implementation in a message-passing system



6.10

Tree Implementation

More efficient. O(log *p*) steps
Suppose 8 processes, *P*0, *P*1, *P*2, *P*3, *P*4, *P*5, *P*6, *P*7:

1st 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)

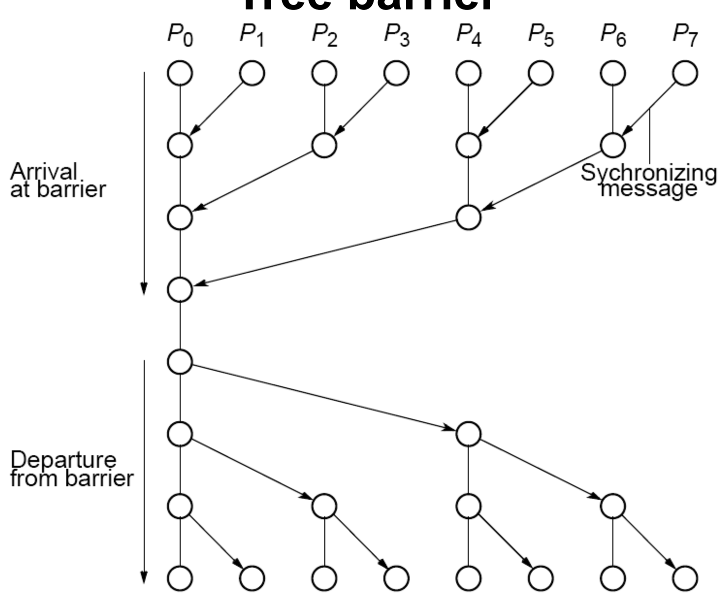
2nd stage: *P*2 sends message to *P*0; (*P*2 & *P*3 reached their barrier)
*P*6 sends message to *P*4; (*P*6 & *P*7 reached their barrier)

3rd stage: *P*4 sends message to *P*0; (*P*4, *P*5, *P*6, & *P*7 reached barrier)
*P*0 terminates arrival phase;
(when *P*0 reaches barrier & received message from *P*4)

Release with a reverse tree construction.

6.11

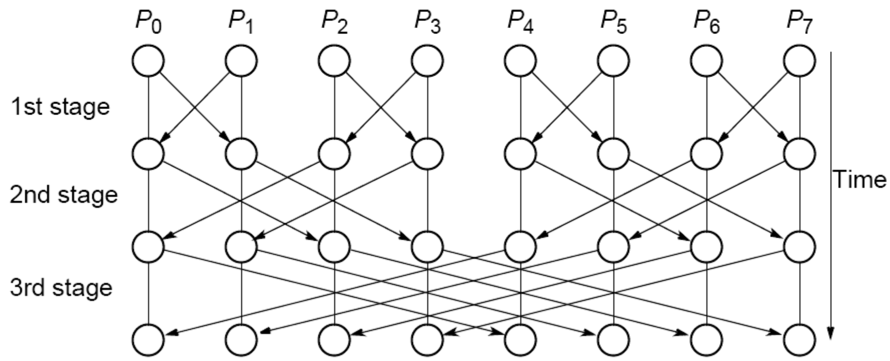
Tree barrier



6.12

Butterfly Barrier

1st stage	$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
2nd stage	$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
3rd stage	$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



6.13

Local Synchronization

Suppose a process *Pi* needs to be synchronized and to exchange data with process *Pi*-1 and process *Pi*+1 before continuing:

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

Not a perfect three-process barrier because process *Pi*-1 will only synchronize with *Pi* and continue as soon as *Pi* allows. Similarly, process *Pi*+1 only synchronizes with P*i*.

6.14

Deadlock

When a pair of processes each send and receive from each other, deadlock may occur.

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

6.15

Combined deadlock-free blocking sendrecv() routines

Example

Process P_{i-1}	Process P_i	Process P_{i+1}
`sendrecv(Pi);` ↔	`sendrecv(Pi-1);`	
	`sendrecv(Pi+1);` ↔	`sendrecv(Pi);`

MPI provides **MPI_Sendrecv()** and **MPI_Sendrecv_replace().**
MPI sendrev()s actually has 12 parameters!

6.17

Partitioning

Usually number of processors much fewer than number of data items to be processed. Partition the problem so that processors take on more than one data item.

6.38

block allocation – allocate groups of consecutive unknowns to processors in increasing order.

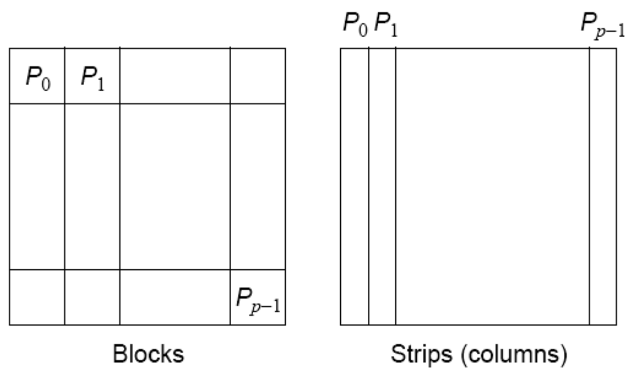
cyclic allocation – processors are allocated one unknown in order; i.e., processor P_0 is allocated x_0, x_p, x_{2p}, …, $x_{((n/p)-1)p}$, processor P_1 is allocated x_1, x_{p+1}, x_{2p+1}, …, $x_{((n/p)-1)p+1}$, and so on.

Cyclic allocation no particular advantage here (Indeed, may be disadvantageous because the indices of unknowns have to be computed in a more complex way).

6.39

Partitioning

Normally allocate more than one point to each processor, because many more points than processors.
Points could be partitioned into square blocks or strips:

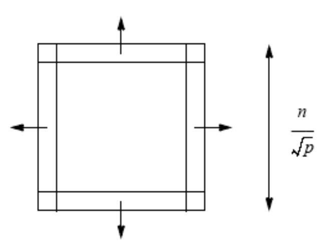


6.54

Block partition

Four edges where data points exchanged.
Communication time given by

$$t_{commsq} = 8\left(t_{startup} + \frac{n}{\sqrt{p}} t_{data}\right)$$



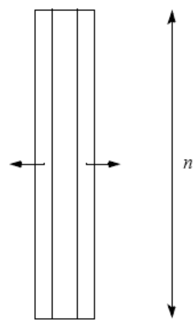
Square blocks

6.55

Strip partition

Two edges where data points are exchanged.
Communication time is given by

$$t_{commcol} = 4(t_{startup} + nt_{data})$$



Strips

6.56

Safety and Deadlock

When all processes send their messages first and then receive all of their messages is "unsafe" because it relies upon buffering in the **send()**s. The amount of buffering is not specified in MPI.

If insufficient storage available, send routine may be delayed from returning until storage becomes available or until the message can be sent without buffering.

Then, a 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.

6.60

Making the code safe

Alternate the order of the **send()**s and **recv()**s in adjacent processes so that only one process performs the **send()**s first.

Then even synchronous **send()**s would not cause deadlock.

Good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

6.61

MPI Safe Message Passing Routines

MPI offers several methods for safe communication:

- Combined send and receive routines:
  `MPI_Sendrecv()`
  which is guaranteed not to deadlock
- Buffered send()s:
  `MPI_Bsend()`
  here the user provides explicit storage space
- Nonblocking routines:
  `MPI_Isend()` **and MPI_Irecv()**

  which return immediately.
  Separate routine used to establish whether message has been received:
  `MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, or `MPI_Testany()`.

6.62

Chapter 7

Load Balancing and Termination Detection

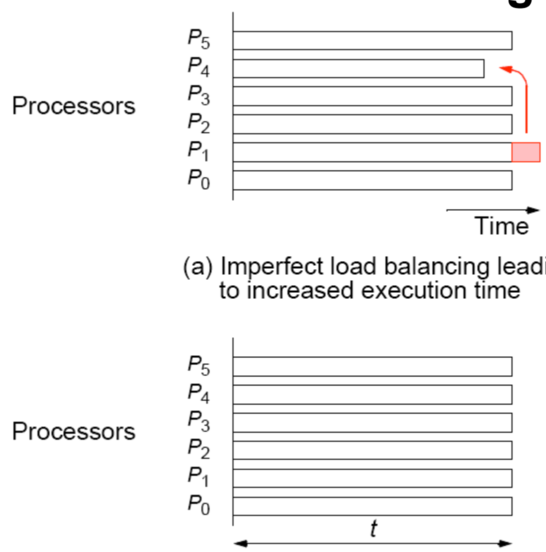
174

Load balancing – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

Termination detection – detecting when a computation has been completed. More difficult when the computation is distributed.

175

Load balancing



(a) Imperfect load balancing leading to increased execution time

(b) Perfect load balancing

176

Static Load Balancing

Before execution of any process.
Some potential static load balancing techniques:

- *Round robin algorithm* — passes out tasks in sequential order of processes coming back to the first when all processes have been given a task
- *Randomized algorithms* — selects processes at random to take tasks
- *Recursive bisection* — recursively divides the problem into sub-problems of equal computational effort while minimizing message passing
- *Simulated annealing* — an optimization technique
- *Genetic algorithm* — another optimization technique

177

Several fundamental flaws with static load balancing even if a mathematical solution exists:

- Very difficult to estimate accurately execution times of various parts of a program without actually executing the parts.
- Communication delays that vary under different Circumstances
- Some problems have an indeterminate number of steps to reach their solution.

178

Dynamic Load Balancing

Vary load during the execution of the processes.

All previous factors taken into account by making division of load dependent upon execution of the parts as they are being executed.

Does incur an additional overhead during execution, but it is much more effective than static load balancing

179

Processes and Processors

Computation will be divided into *work* or *tasks* to be performed, and processes perform these tasks. Processes are mapped onto processors.

Since our objective is to keep the processors busy, we are interested in the activity of the processors.

However, often map a single process onto each processor, so will use the terms *process* and *processor* somewhat interchangeably.

180

Dynamic Load Balancing

Can be classified as:

- Centralized
- Decentralized

181

Centralized dynamic load balancing

Tasks handed out from a centralized location. Master-slave structure.

182

Decentralized dynamic load balancing

Tasks are passed between arbitrary processes.

A collection of worker processes operate upon the problem and interact among themselves, finally reporting to a single process.

A worker process may receive tasks from other worker processes and may send tasks to other worker processes (to complete or pass on at their discretion).

183

Centralized Dynamic Load Balancing

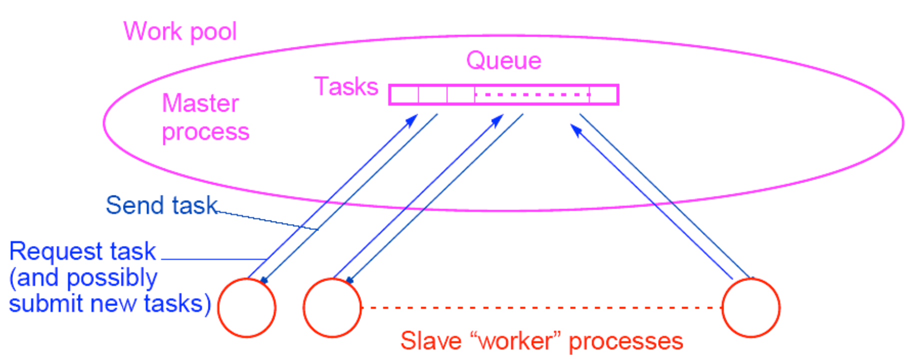
Master process(or) holds collection of tasks to be performed.

Tasks sent to slave processes. When a slave process completes one task, it requests another task from the master process.

Terms used : *work pool*, *replicated worker*, *processor farm*.

184

Centralized work pool



185

Termination

Computation terminates when:

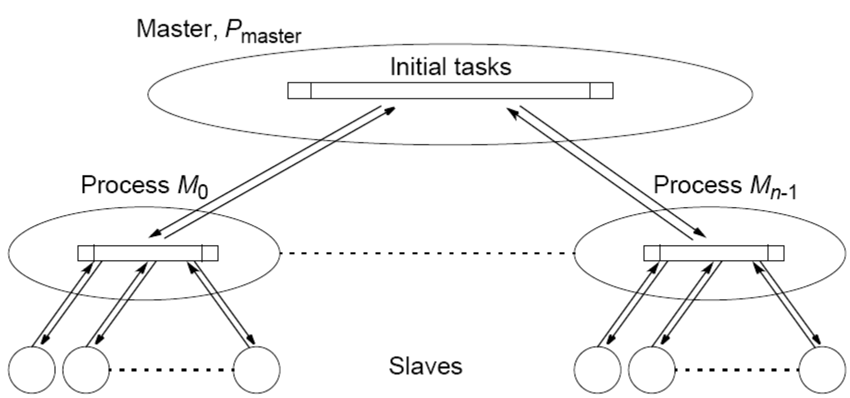
• The task queue is empty and

• Every process has made a request for another task without
any new tasks being generated

Not sufficient to terminate when task queue empty if one or more processes are still running if a running process may provide new tasks for task queue.

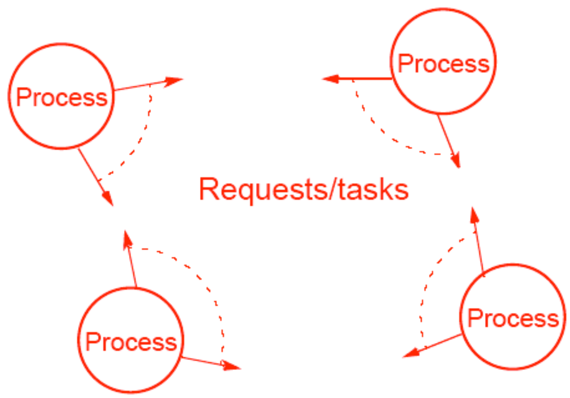
Decentralized Dynamic Load Balancing

Distributed Work Pool



Fully Distributed Work Pool

Processes to execute tasks from each other



188