Chapter 6 – Synchronous Computations

A group of separate computations must wait for each other before proceeding, thereby becoming synchronized.

Fully synchronous applications require all processes to be synchronized at regular points:

- Generally the same computation is applied to a set of data points.
- All operations start the same time in a lock-step manner anologous to SIMD computations.

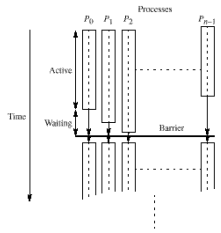

Figure 6.1 Processes reaching the barrier at different times.

Synchronization with Barrier

- When processes need to exchange data between themselves and then continue from a known state together
- Each process must wait until all others have reached a particular reference point in their computations.
- In dynamic process creation:
  - exit and respawn -- but costly
  - barrier
  - All processes must wait in a barrier and placed in an inactive state and they wait others to reach the same point.
    - potential race condition

- MP systems: as library routines
  - `MPI_Barrier()`
  - `pvm_barrier()` -- all or a subset of processes
- Synchronous and message tags are not used.
- Let's review some of the common implementations of a barrier:
  - counter
  - tree
  - butterfly

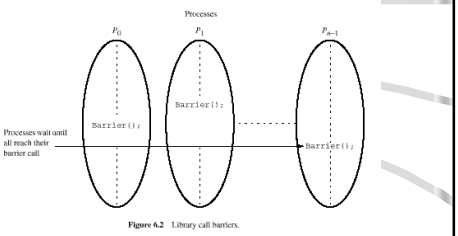

Figure 6.2 Library call barriers.

Counter Implementation (Linear Barrier)

- A centralized counter is used to count the number of processes reaching the barrier.
- When the correct number is reached, all other processes waiting for the counter are released.
- Implementation in two phases:
  - an arrival (or trapping) phase
  - a departure (or release) phase
- Consider the case a barrier might be used more than once in a process, i.e., a process enters the barrier for a second time before previous processes have left the barrier for the first time.

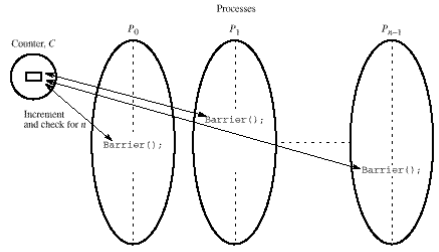

Figure 6.3 Barrier using a centralized counter.

Master Process:

```
for (i=0; i<n; i++)   /* arrival phase */
   recv(P_any);
for (i=0; i<n; i++)   /* departure phase */
   send(P_i);
```

Slave Process:

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

- All processes must reach the arrival phase before continuing on to a clearly defined departure phase.
- Blocking `recv()` and locally blocking `send()` operations are used and this implementation has a time complexity of O(*n*) with *n* processes.
- An implementation using `gather()` and `broadcast()` routines is possible.
- Even if system-supplied `barrier()` is available, user implemented barriers may be required.

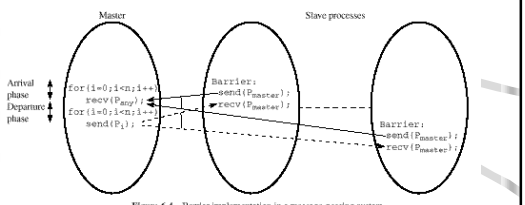

Figure 6.4 Barrier implementation in a message-passing system.

Tree Implementation

- With *n* processes, we can implement a barrier with a counter that has *2 log n* steps and a time complexity of O(*log n*).

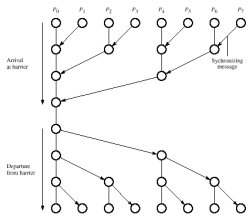

Figure 6.5 Tree barrier.

Butterfly Barrier

- Pairs of processors synchronize at each stage.
- Each sync requires only a single pair of `send()`/`recv()`. After all sync stages, all processes can continue.
- At stage *s*, process *i* syncs with process $i+2^{s-1}$ if *n* is a power of 2.
- With *n* processes, it has *n* steps, and complexity of O(*log n*).

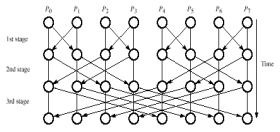

Figure 6.6 Butterfly construction.

Local Synchronization

- Sometimes processes need only be synchronized with a few other processes.
  - Ex: mesh or pipeline fashion processor organizations

```
Process P_i-1          Process P_i          Process P_i+1
recv(P_i);      <--    send(P_i-1);
send(P_i);      -->    send(P_i+1);    -->  recv(P_i);
                       recv(P_i-1);         send(P_i);
                       recv(P_i+1);
```

Note that this is not a perfect three-process barrier, but sufficient.

Deadlock

- When a pair of processes each send and receive from each other, deadlock may occur.
  - If both processes perform synchronous sends (or blocking sends without sufficient buffering)
  - Avoidance: arrange processes so that even-numbered processes perform their sends first and odd-numbered processes perform their receives first.
- `sendrecv()` routine: combined blocking operation for bidirectional data transfers is implemented so that deadlock cannot occur.

  `MPI_Sendrecv()/_replace()` having 12 parameters

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

Data Parallel Computations

- Have implicit sync. requirements
- The same operation is performed on different data elements simultaneously.
- Two reasons:
  - ease of programming (only one program)
  - can scale easily to larger problem sizes
- SIMD computers operate as data parallel computers
  - Synchronism is built into the hardware, the processors operate in lock-step fashion
  - same instruction is executed by different processors but on different data.

Ex: add the same constant to each element of an array:

```
for (i=0; i<n; i++)        ⇒   a[] = a[] + k; (SIMD)
    a[i] += k;
```

`forall` statement: a special parallel construct to specify data parallel operations.

```
forall (i=0; i<n; i++)
    a[i] += k;
```

- *n* instances of the body is executed simultaneously
- no iteration!
- whole construct will not be completed until all instances of the body have been executed.
  - Hence a barrier is implicit within the `forall` construct.

- On SIMD computers
- On a message-passing computer
  - explicit barrier is needed
  - SPMD style of programming is used

```
i = myrank;
a[i] += k;  /* body */
barrier(mygroup);
```

Other data parallel algorithms (Hillis and Steel, Jr., 1986)
- summing numbers
- sorting
- operating on linked lists
- etc.

Prefix Sum Problem

- Given a list of numbers x_0, x_1, ..., x_{n-1}, all the partial summations (i.e., x_0+x_1, $x_0+x_1+x_2$, $x_0+x_1+x_2+x_3$, ...) are computed.
  - Any associative operation can take place of +.
  - Processor allocation, data compaction, sorting, polynomial evaluation.

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

Time complexity is $O(n^2)$.

- Figure 6.8
- Adding all partial sums of 16 numbers as described in (Hillis and Steel, Jr., 1986)
  - original numbers are lost
  - a different number of computations occur in each step
  - requires *log n* steps, where there are *n* numbers (and *n* is a power of 2).
    - In step j ($0 \le j < log\ n$), $n-2^j$ additions occur.

Sequential Code:
```
for (j=0; i< log(n); j++)
   for (i=2^j; i<n; i++)
      x[i] += x[i-2^j];
```
$O(n^2)$

SIMD code:
```
for (j=0; i< log(n); j++)
   forall (i=0; i<n; i++)
      if (i>= 2^j) x[i]+=x[i-2^j];
```
not cost optimal! → With *n-1* processors, $O(n \log n)$

Synchronous Iteration

- Iterative method is a powerful method for solving numerical problems, where a calculation is repeated until convergence criteria is satisfied.
  - the result of one iteration is used in the next iteration.
- Parallel implementation can be successfully employed to iterative methods when there are multiple independent instances of the iteration.
- Synchronous iteration or synchronous parallelism is used to solving a problem by iteration where
  - each iteration is composed of several processes that start together at the beginning of each iteration and
  - the next iteration cannot begin until all processes have finished the previous iteration.


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

SPMD program:

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