Synchronous Parallel Computation

An Extreme: Cellular Automata

- Synchronous computation
- Infinitely-large grid (finite occupancy)
- Typically fine-grain
- If distributed, still need to communicate and boundaries, once per cycle.

Barriers: Used in MIMD, SPMD

- **Synchronize** all of a group of processes
- Used in both distributed and shared-memory
- Issue: Implementation & cost

Barriers

- Synchronize all of a group of processes
- Used in both distributed and shared-memory
- Issue: Implementation & cost

Counter Method for Barriers

- One-phase version
  - Use for distributed-memory
  - Each processor sends a message **to each** of the others when barrier reached.
  - When each processor has received a message from all others, the processors pass the barrier

Counter Method for Barriers

- Two-phase version
  - Use for shared-memory
  - Each processor sends a message to the **master** process.
  - When the master has received a message from all others, it sends messages to each processor indicating that it can pass the barrier.
  - Easily implemented with blocking receives, or semaphores (one per processor).

Tree Barrier

- Processors are organized as a tree, with each sending to its parent.
- **Fan-in phase:** When the root of the tree receives messages from both children, the barrier is complete.
- **Fan-out phase:**Messages are then sent down the tree in the reverse direction, and processes pass the barrier upon receipt.

Butterfly Barrier

- Adjacent pairs (in dimensions of a hypercube in sequence) notify each other.

- Advantage is that no separate fan-out phase is required.

Butterfly Barrier

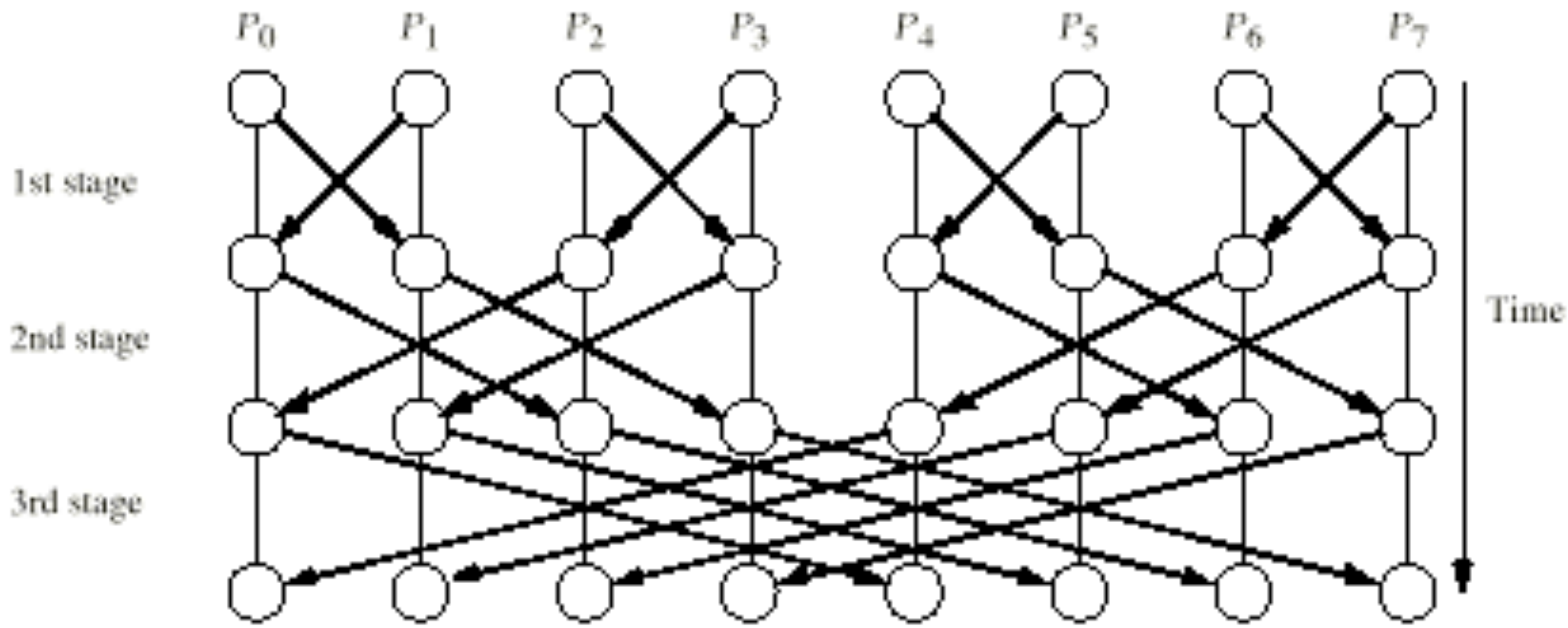

Figure 6.6 Butterfly construction.

Barrier Bonuses

- Barrier messages heretofore are communications with null content.
- By **adding content** to messages, barriers can have added utility.

Barrier Bonuses

- These can be accomplished along with a barrier:
  - **Reduce** using a binary operator (esp. good with a tree or butterfly barrier)
  - All-to-all **broadcast**

Data Parallel Programming Constructs

- **forall** statement:

```
forall( j = 0; j < n; j++ )
        {
        … body done in parallel for all j ...
        }
```

- **Barrier** implied at end among parallel bodies

Example: Prefix-Sum

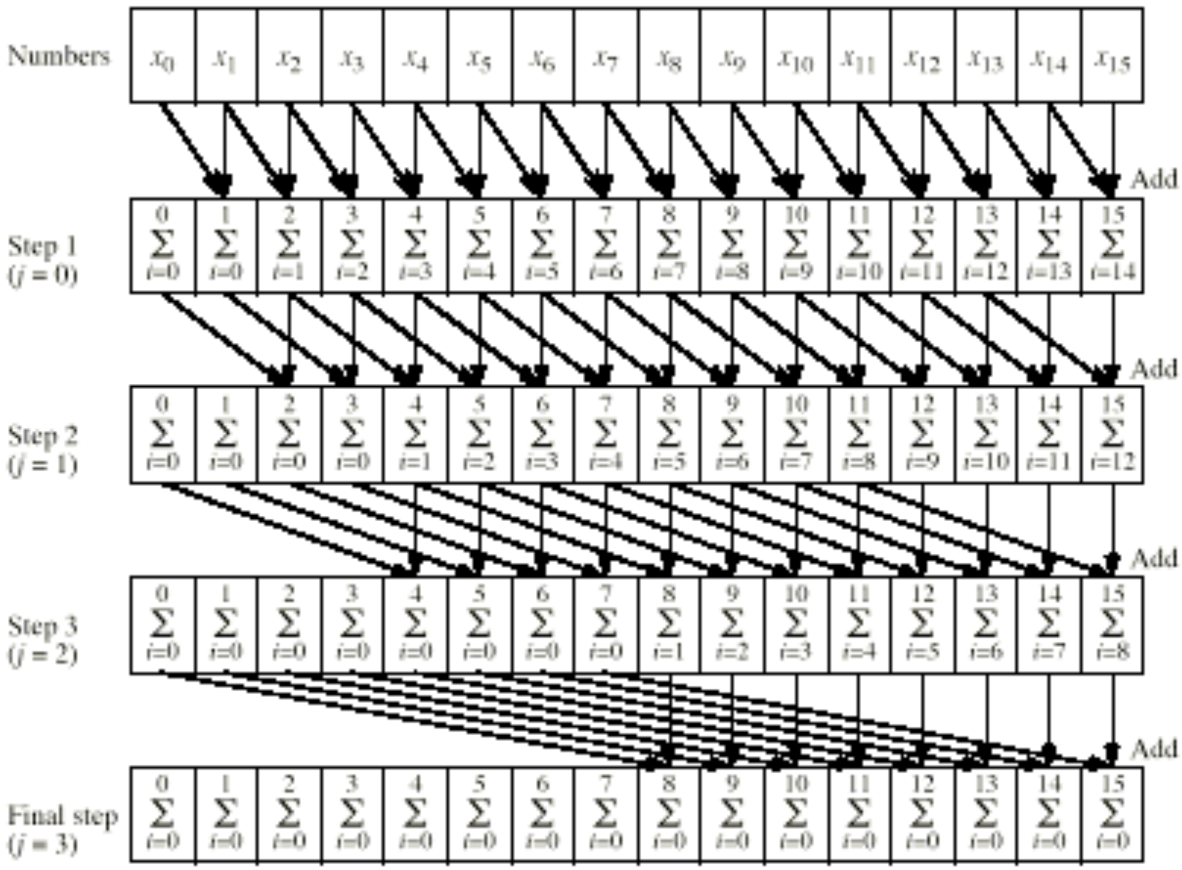

Figure 6.8 Data parallel prefix sum operation.

Example: Prefix-Sum

- Assume that n is a power of 2.
- Assume shared memory.
- for(j = 0; j < log(n); j++) // parallel steps

forall(i = 2^j; i < n; i++)

$$\underbrace{x[i]}_{\text{buffered new value}} \mathrel{+}= \underbrace{x[i - 2^j]}_{\text{old value}};$$

Implementing **forall** using SPMD: "Synchronous Iteration"

- ```
  for( j = 0; j < log(n); j++ )
        forall( i = 0; i < n; i++)
              Body(i);
  ```

implementable in **SPMD** as:

- ```
  for( j = 0; j < log(n); j++ )
        {
        i = my_process_rank();
        Body(i);
        barrier();
        }
  ```

Outer
forall processes
implicit due to SPMD

Example:
Iterative Linear Equation Solver

```
for( iter = 0; iter < numIterations; iter++ )

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

Note:
Local memory for each i.

Iterative Linear Equation Solver: **Translation** to **SPMD**

Outer forall processes implicit

```
for( iter = 0; iter < numIterations; iter++ )
    {
    i = my_process_rank();
    double sum = 0;
    for( j = 1; j < n; j++ )
            sum += a[i][j]*x[j];
    new_x[i] = sum;
    all gather new_x to x (implied barrier)
    }
```

Nested forall's

- forall(i = 0; i < m; i++)

  forall(j = 0; j < n; i++)

  Body(i, j)

Example of nested forall's: Laplace equation

- ```
  forall( i = 0; i < m; i++)
      forall( j = 0; j < n; i++)
          x[i][j] = (x[i-1][j] + x[i][j-1] + x[i+1][j] + x[i][j+1])/4.0;
  ```
  ```

Exercise

- How would you translate nested forall’s to SPMD?