Parallel Programming Models

HPC

Prof. Robert van Engelen

Overview

- Basic concepts
- Programming models
- Multiprogramming
- Shared address space model
  - UMA versus NUMA
  - Distributed shared memory
  - Task parallel
  - Data parallel, vector and SIMD
- Message passing model
- Hybrid systems
- BSP model

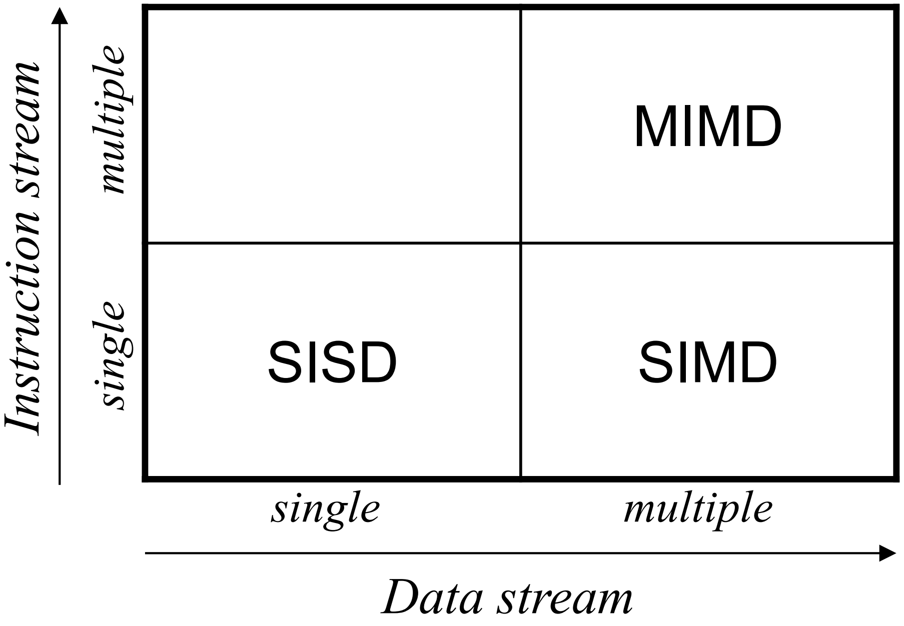
Parallel Programming: Basic Concepts

- Control
  - □ How is *parallelism* created, implicitly (hardwired) or explicitly?
  - □ What *orderings* exist between operations?
  - □ How do different threads of control *synchronize?*
- Naming data
  - □ What data is *private* and what is *shared?*
  - □ How is *logically shared data* accessed or communicated?
- Operations on data
  - □ What are the basic *operations* on shared data?
  - □ Which operations are considered *atomic?*
- Cost
  - □ How do we account for the *cost* of each of the above to achieve parallelism (man hours spent, software/hardware cost)

Parallel Programming Models

- *Programming model* is a conceptualization of the machine that a programmer uses for developing applications
  - *Multiprogramming model*
    - A set of independence tasks, no communication or synchronization at program level, e.g. web server sending pages to browsers
  - *Shared address space* (shared memory) programming
    - Tasks operate and communicate via shared data, like bulletin boards
  - *Message passing* programming
    - Explicit point-to-point communication, like phone calls (connection oriented) or email (connectionless, mailbox posts)

Flynn's Taxonomy



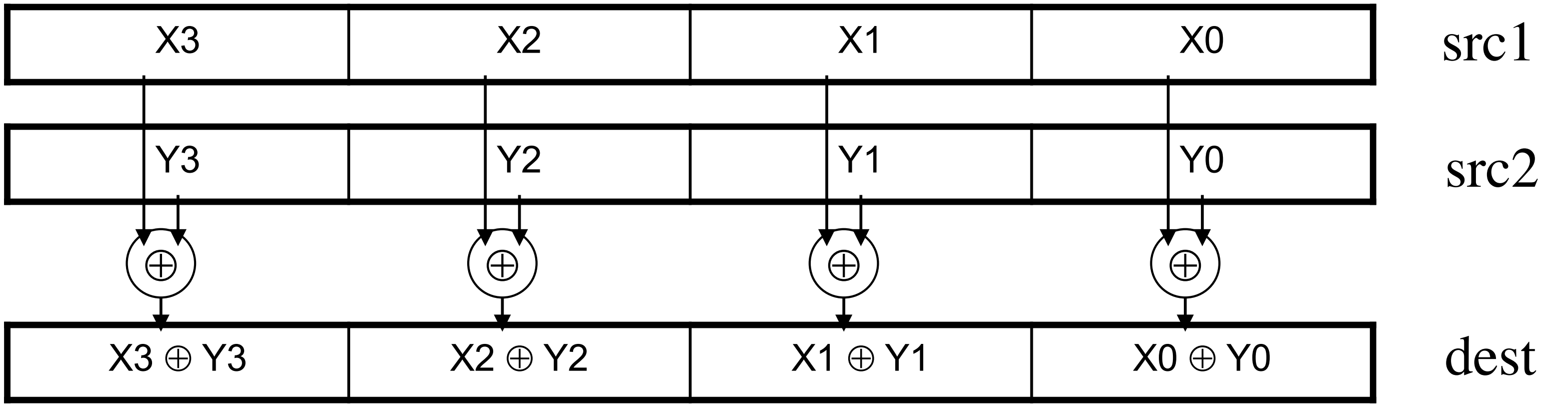
Instruction stream		
multiple		MIMD
single	SISD	SIMD
	single	multiple
	Data stream	

- *Single instruction stream single data stream* (SISD)
  - □ Traditional PC system
- *Single instruction stream multiple data stream* (SIMD)
  - □ Similar to MMX/SSE/AltiVec multimedia instruction sets
  - □ MASPAR
- *Multiple instruction stream multiple data stream* (MIMD)
  - □ *Single program, multiple data* (SPMD) programming: each processor executes a copy of the program

MIMD versus SIMD

- *Task parallelism, MIMD*
  - □ Fork-join model with thread-level parallelism and shared memory
  - □ Message passing model with (distributed processing) processes
- *Data parallelism, SIMD*
  - □ Multiple processors (or units) operate on segmented data set
  - □ SIMD model with vector and pipeline machines
  - □ SIMD-like multi-media extensions, e.g. MMX/SSE/Altivec

Vector operation X[0:3] ⊕ Y[0:3] *with SSE instruction on Pentium-4*



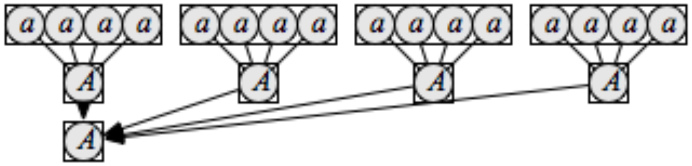
X3	X2	X1	X0	src1
Y3	Y2	Y1	Y0	src2
⊕	⊕	⊕	⊕	
X3 ⊕ Y3	X2 ⊕ Y2	X1 ⊕ Y1	X0 ⊕ Y0	dest

Task versus Data Parallel

- Task parallel (maps to high-level MIMD machine model)
  - □ Task differentiation, like restaurant cook, waiter, and receptionist
  - □ Communication via shared address space or message passing
  - □ Synchronization is explicit (via locks and barriers)
  - □ Underscores operations on private data, explicit constructs for communication of shared data and synchronization
- Data parallel (maps to high-level SIMD machine model)
  - □ Global actions on data by tasks that execute the same code
  - □ Communication via shared memory or logically shared address space with underlying message passing
  - □ Synchronization is implicit (lock-step execution)
  - □ Underscores operations on shared data, private data must be defined explicitly or is simply mapped onto shared data space

A Running Example: $A = \sum_{i=1}^{N} f(a_i)$

- Parallel decomposition
  - □ Assign N/P elements to each processor



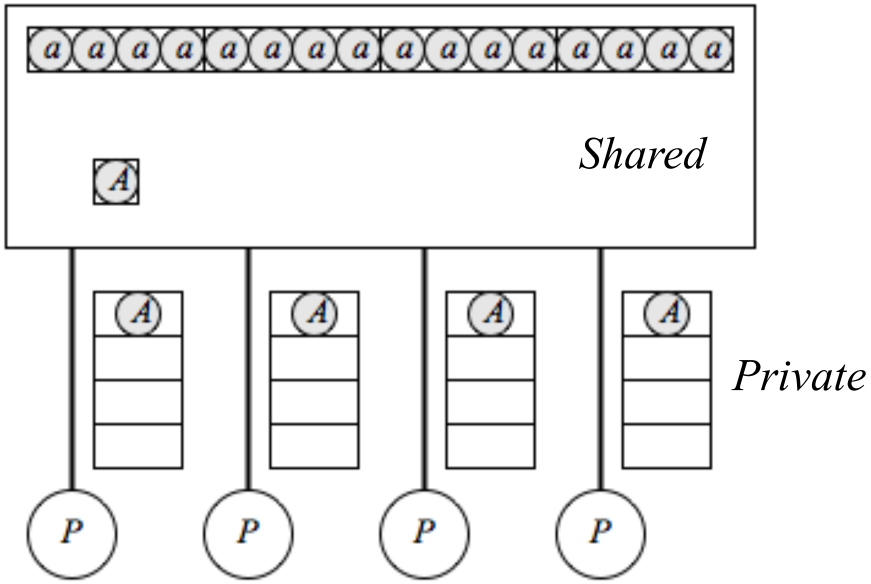
  - □ Each processor computes the partial sum $$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$
  - □ One processor collects the partial sums $$A = \sum_{i=1}^{P} A_i$$
- Determine the data placement:
  - □ Logically shared: array a, global sum A
  - □ Logically private: the function $f(a_i)$ evaluations
  - □ Either logically shared or private: partial sums A_j

Programming Model 1

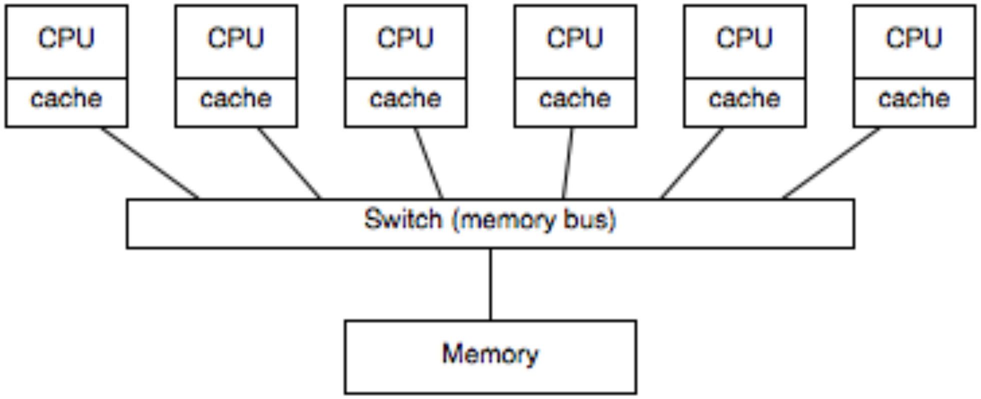
- *Shared address space* (*shared memory*) programming
- Task parallel, thread-based MIMD
  - □ Program is a collection of threads of control
- Collectively operate on a set of *shared data* items
  - □ Global static variables, Fortran common blocks, shared heap
- Each thread has *private variables*
  - □ Thread state data, local variables on the runtime stack
- Threads coordinate explicitly by synchronization operations on shared variables, which involves
  - □ Thread creation and join
  - □ Reading and writing flags
  - □ Using locks and semaphores (e.g. to enforce mutual exclusion)

Programming Model 1

- *Uniform memory access* (UMA) shared memory machine
  - Each processor has uniform access to memory
  - Symmetric multiprocessors (SMP)
- No local/private memory, private variables are put in shared memory
- Cache makes access to private variables seem "local"



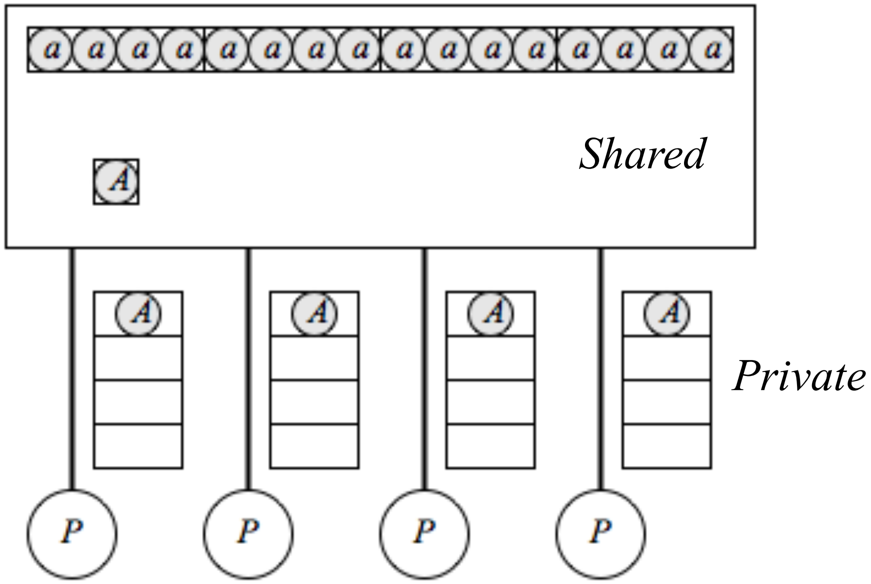
Programming model



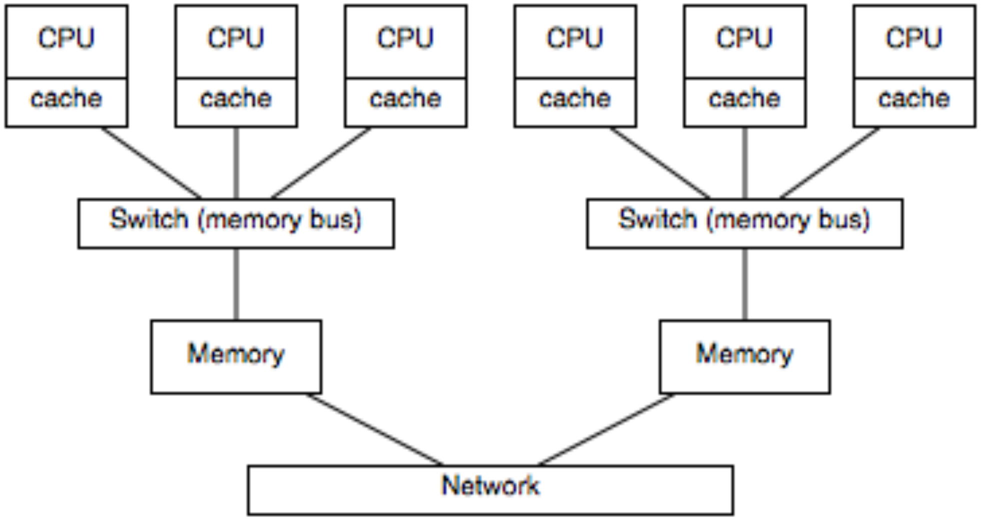
Machine model

Programming Model 1

- *Nonuniform memory access* (NUMA) shared memory machine
  - Memory access time depends on location of data relative to processor
  - Local access is faster
- No local/private memory, private variables are put in shared memory



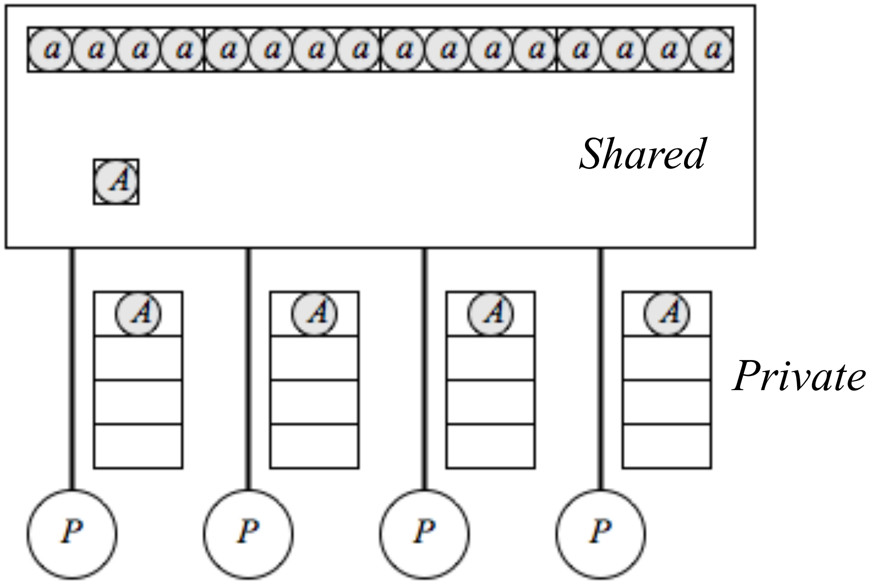
Programming model



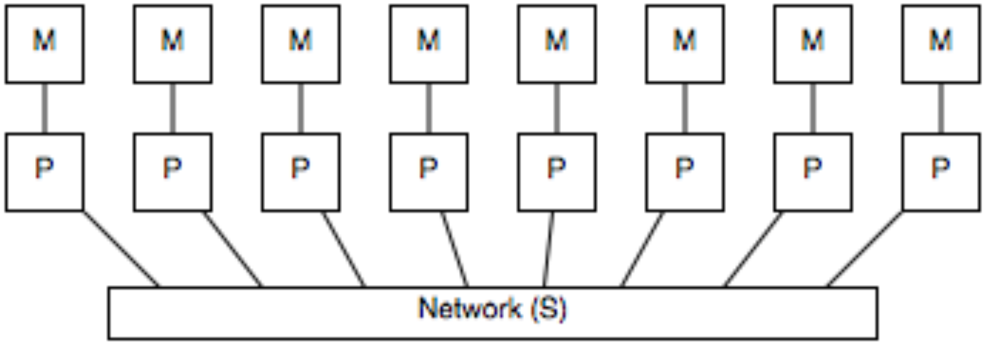
Machine model

Programming Model 1

- *Distributed shared memory machine* (DSM)
- Logically shared address space
  - Remote memory access is more expensive (NUMA)
  - Remote memory access requires communication, automatic either done in hardware or via software layer



Programming model



Machine model

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
shared A[1..2]
private i

A[1] := 0
for i = 1..N/2
  A[1] := A[1]+f(a[i])
A := A[1] + A[2]
```

Thread 2

```
shared A
shared A[1..2]
private i

A[2] := 0
for i = N/2+1..N
  A[2] := A[2]+f(a[i])
```

What could go wrong?

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
A[1]  := A[1]+f(a[0])
A[1]  := A[1]+f(a[1])
A[1]  := A[1]+f(a[2])
…
A[1]  := A[1]+f(a[9])
A  := A[1]  + A[2]
```

Thread 2

```
…
A[2]  := A[2]+f(a[10])
A[2]  := A[2]+f(a[11])
A[2]  := A[2]+f(a[12])
…
…
A[2]  := A[2]+f(a[19])
```

Thread 2 has not completed in time

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
shared A[1..2]
private i

A := 0
A[1] := 0
for i = 1..N/2
  A[1] := A[1]+f(a[i])
A := A + A[1]
```

Thread 2

```
shared A
shared A[1..2]
private i

A := 0
A[2] := 0
for i = N/2+1..N
  A[2] := A[2]+f(a[i])
A := A + A[2]
```

What could go wrong?

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
A[1] := A[1]+f(a[0])
A[1] := A[1]+f(a[1])
A[1] := A[1]+f(a[2])
…
A := A + A[1]
```

Thread 2

```
A[2] := A[2]+f(a[10])
A[2] := A[2]+f(a[11])
A[2] := A[2]+f(a[12])
…
A := A + A[2]
```

↔ *Race condition*

```
reg1 = A
reg2 = A[1]
reg1 = reg1 + reg2
A = reg1
```

```
reg1 = A
reg2 = A[2]
reg1 = reg1 + reg2
A = reg1
```

Instructions from different threads can be interleaved arbitrarily: the resulting value of **`A`** *can be* **`A[1]`**, **`A[2]`**, *or* **`A[1]+A[2]`**

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
shared A[1..2]
private i

A[1] := 0
for i = 1..N/2
  A[1] := A[1]+f(a[i])
atomic A := A + A[1]
```

Thread 2

```
shared A
shared A[1..2]
private i

A[2] := 0
for i = N/2+1..N
  A[2] := A[2]+f(a[i])
atomic A := A + A[2]
```

Solution with atomic operations to prevent race condition

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
shared A[1..2]
private i

A[1] := 0
for i = 1..N/2
  A[1] := A[1]+f(a[i])
lock
A := A + A[1]
unlock
```

Critical section

Thread 2

```
shared A
shared A[1..2]
private i

A[2] := 0
for i = N/2+1..N
  A[2] := A[2]+f(a[i])
lock
A := A + A[2]
unlock
```

Solution with locks to ensure mutual exclusion

(But this can still go wrong when an FP add exception is raised, jumping to an exception handler without unlocking)

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
private Aj
private i

Aj := 0
for i = 1..N/2
  Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
```

Critical section

Thread 2

```
shared A
private Aj
private i

Aj := 0
for i = N/2+1..N
  Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
```

Note that the A[1] and A[2] are just local, so make them private

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
private Aj
private i

Aj := 0
for i = 1..N/2
  Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
… := A
```

Critical section

Thread 2

```
shared A
private Aj
private i

Aj := 0
for i = N/2+1..N
  Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
… := A
```

What could go wrong?

Programming Model 1

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Thread 1

```
shared A
private Aj
private i

Aj := 0
for i = 1..N/2
  Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
barrier
… := A
```

Thread 2

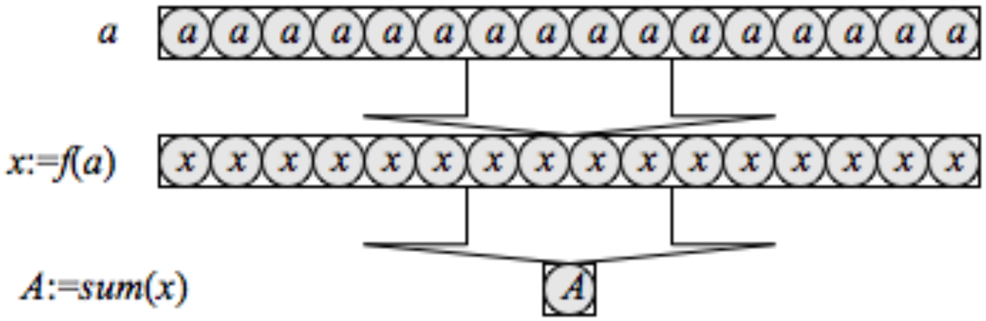
```
shared A
private Aj
private i

Aj := 0
for i = N/2+1..N
  Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
barrier  } All procs synchronize
… := A
```

With locks, private A_j, and barrier synchronization

Programming Model 2

- *Shared address space* (shared memory) programming
- *Data parallel programming*
  - Single thread of control consisting of parallel operations
  - Parallel operations are applied to (a specific segment of) a data structure, such as an array
- Communication is implicit
- Synchronization is implicit

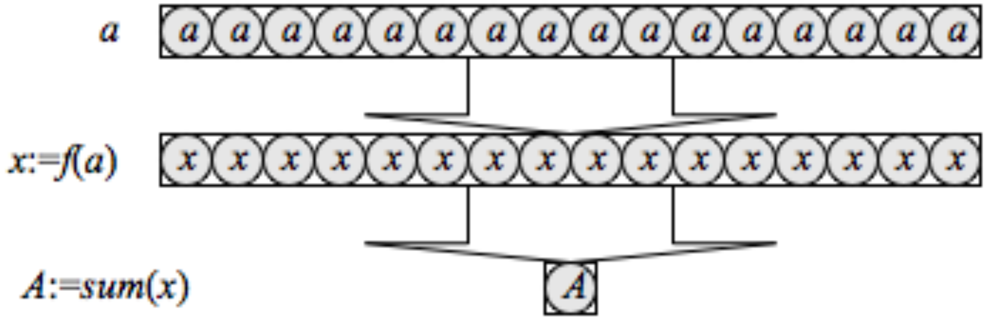
```
shared array a, x
shared A
a := array of input data
x := f(a)
A := sum(x)
```


a
a a a a a a a a a a a a a a a a
x:=f(a)
x x x x x x x x x x x x x x x x
A:=sum(x)
A

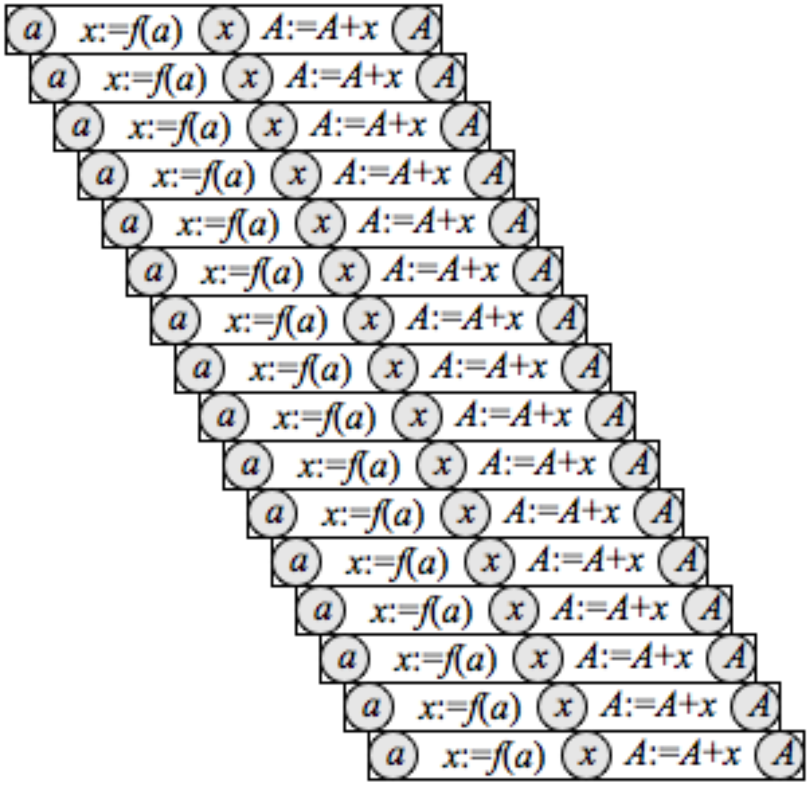
Programming Model 2

- E.g. data parallel programming with a vector machine
- One instruction executes across multiple data elements, typically in a pipelined fashion

```
shared array a, x
shared A
a := array of input data
x := f(a)
A := sum(x)
```



Programming model



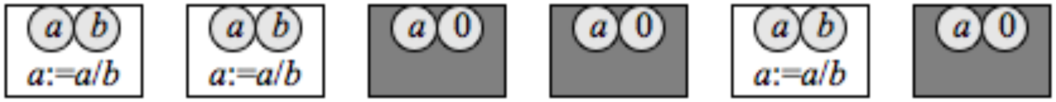
Machine model

Programming Model 2

- Data parallel programming with a SIMD machine
- Large number of (relatively) simple processors
  - □ Like multimedia extensions (MMX/SSE/AltiVec) on uniprocessors, but with scalable processor grids
- A control processor issues instructions to simple processors
  - □ Each processor executes the same instruction (in lock-step)
  - □ Processors are selectively turned off for control flow in program

```
REAL, DIMENSION(6) :: a,b
…
WHERE b /= 0.0
  a = a/b
ENDWHERE
```

Fortran 90 / HPF
(High-Performance Fortran)



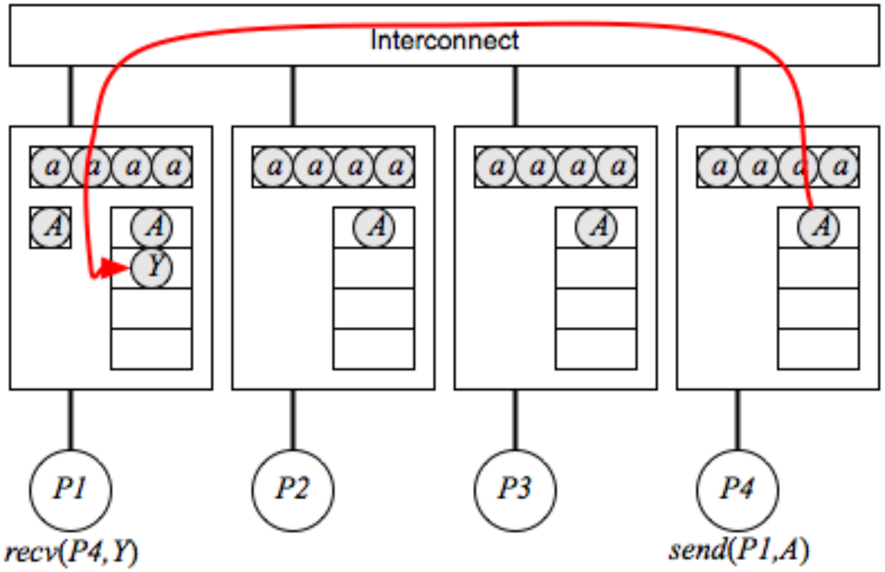
Lock-step execution by an array of processors with some processors temporarily turned off

Programming Model 3

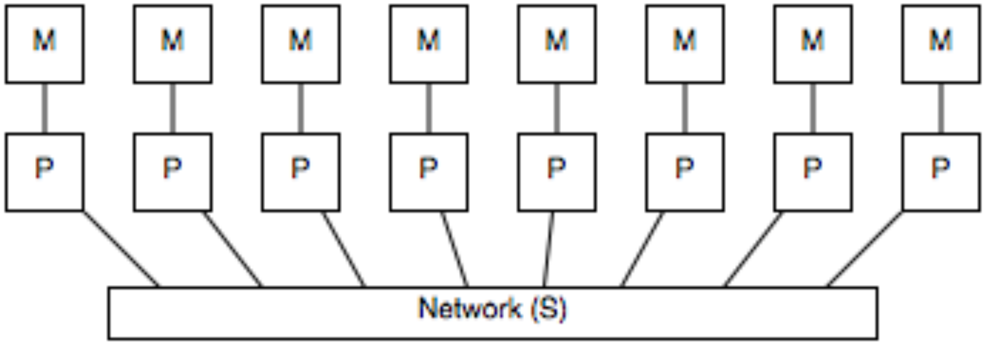
- *Message passing programming*
- Program is a set of *named* processes
  - □ Process has thread of control and local memory with local address space
- Processes communicate via explicit data transfers
  - □ Messages between source and destination, where source and destination are named processors P0…Pn (or compute nodes)
  - □ Logically shared data is explicitly partitioned over local memories
  - □ Communication with send/recv via standard message passing libraries, such as MPI and PVM

Programming Model 3

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms



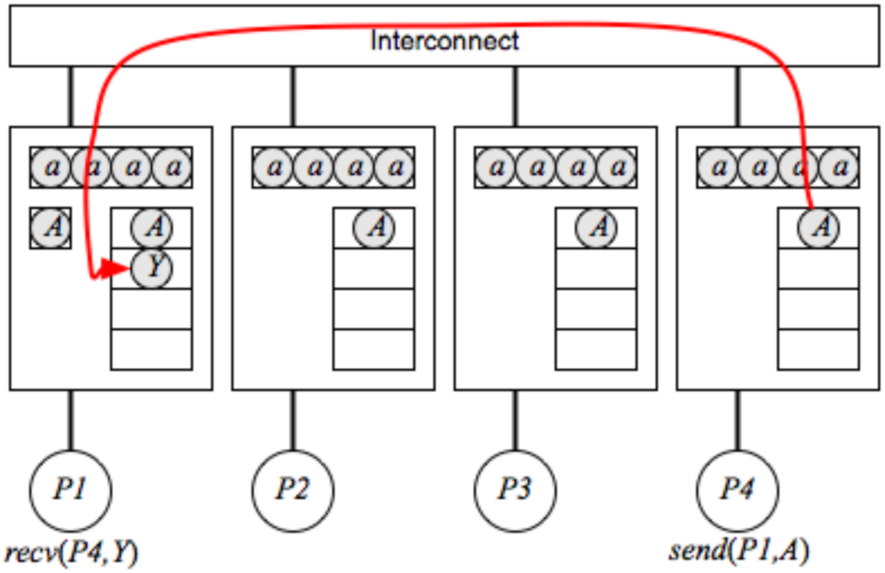
Programming model



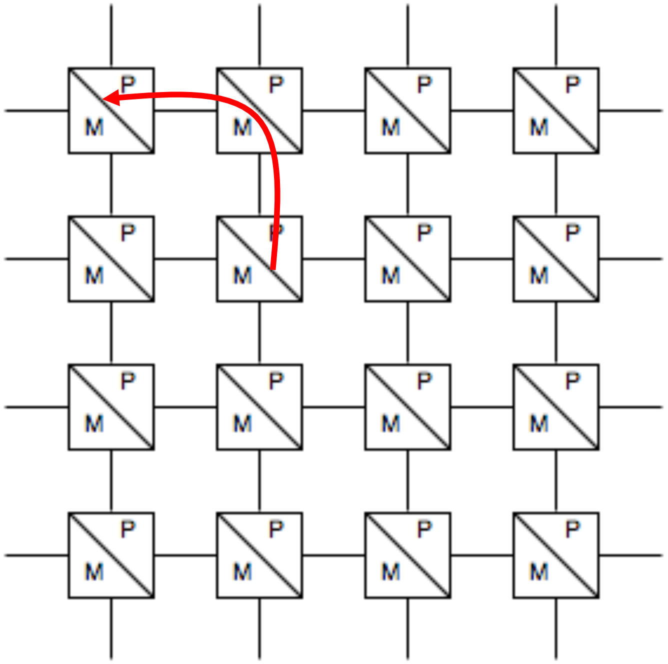
Machine model

Programming Model 3

- Message passing programming
- Each node has a network interface
  - □ Communication and synchronization via network
  - □ Message latency and bandwidth is dependent on network topology and routing algorithms



Programming model

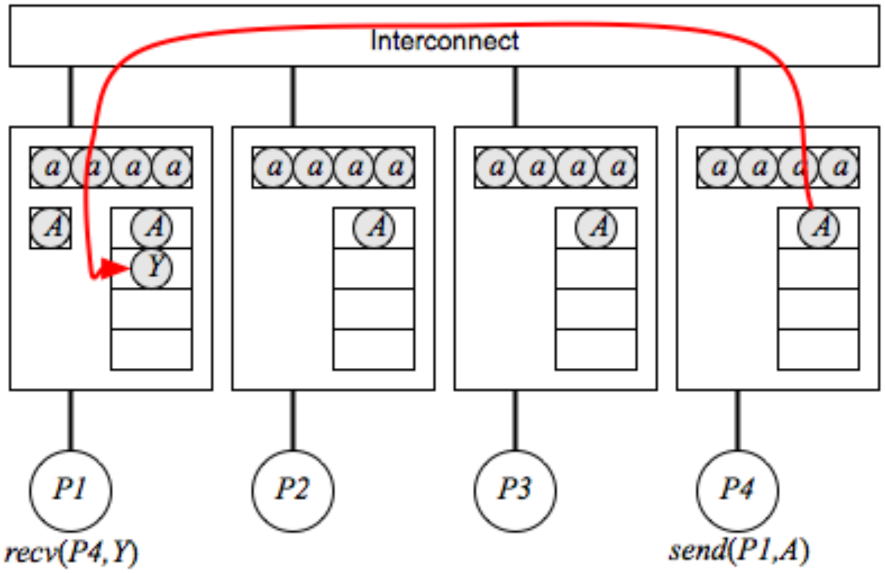


Machine model

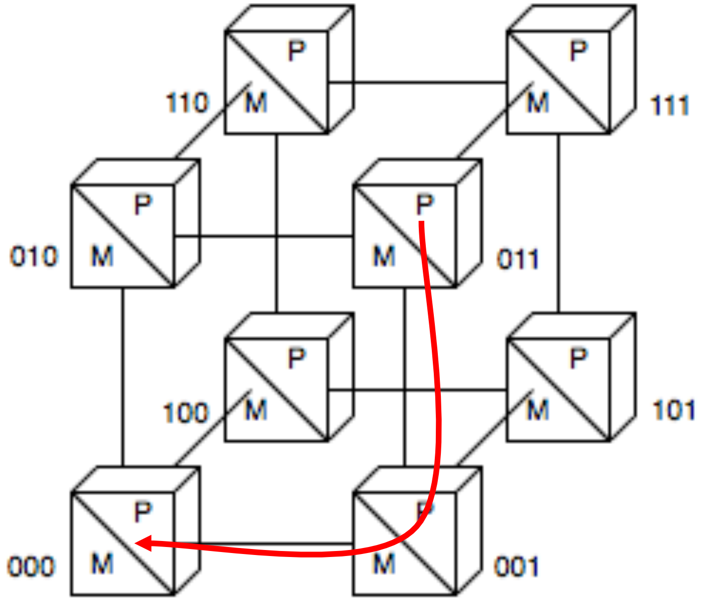
Message passing over mesh

Programming Model 3

- Message passing programming
- Each node has a network interface
  - □ Communication and synchronization via network
  - □ Message latency and bandwidth is dependent on network topology and routing algorithms



Programming model

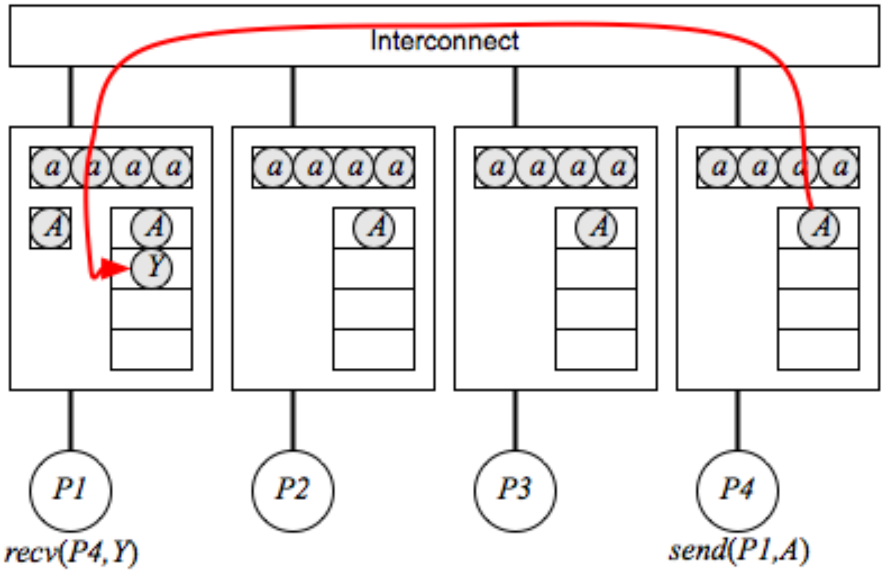


Machine model

Message passing over hypercube

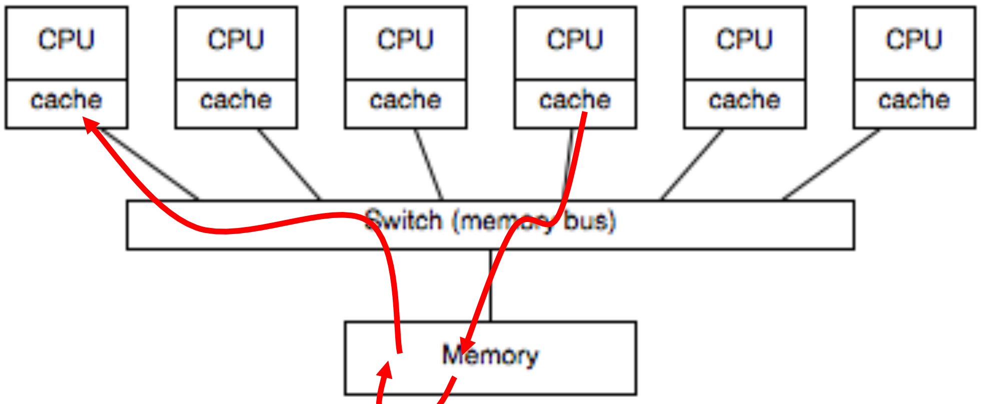
Programming Model 3

- Message passing programming
- On shared memory machine
  - □ Communication and synchronization via shared memory
  - □ Message passing library copies data (messages) in memory, less efficient (MPI call overhead) but portable



Programming model

Message passing on a shared memory machine



Copy data

Machine model

Programming Model 3

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Processor 1

```
A1 := 0
for i = 1..N/2
  A1 := A1+f(al[i])
receive A2 from P2
A := A1 + A2
send A to P2
```

Processor 2

```
A2 := 0
for i = 1..N/2
  A2 := A2+f(al[i])
send A2 to P1
receive A from P1
```

Solution with message passing, where global **`a[1..N]`** *is distributed such that each processor has a local array* **`al[1..N/2]`**

Programming Model 3

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Processor 1

```
A1  := 0
for i = 1..N/2
  A1  := A1+f(al[i])
send A1 to P2
receive A2 from P2
A  := A1 + A2
```

Processor 2

```
A2  := 0
for i = 1..N/2
  A2  := A2+f(al[i])
send A2 to P1
receive A1 from P1
A  := A1 + A2
```

Alternative solution with message passing, where global **`a[1..N]`** *is distributed such that each processor has a local array* **`al[1..N/2]`**

What could go wrong?

Programming Model 3

$$A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)$$

$$A = \sum_{i=1}^{P} A_i$$

Processor 1

```
A1  := 0
for i = 1..N/2
  A1  := A1+f(al[i])
send A1 to P2
receive A2 from P2
A  := A1 + A2
```

Synchronous blocking sends

Processor 2

```
A2  := 0
for i = 1..N/2
  A2  := A2+f(al[i])
send A2 to P1
receive A1 from P1
A  := A1 + A2
```

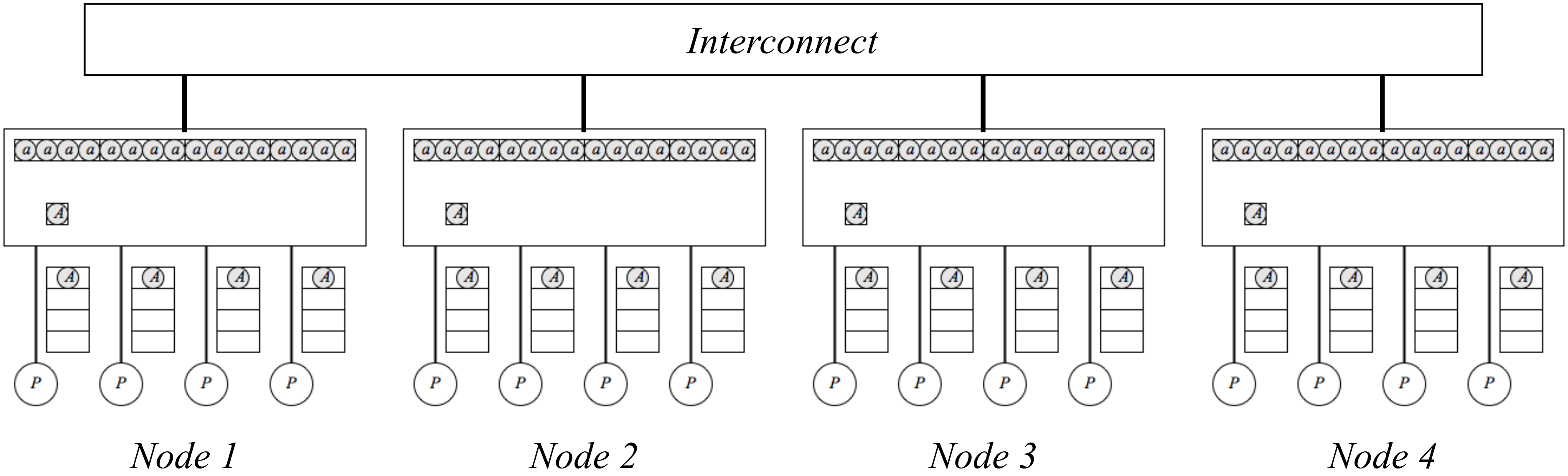
Deadlock with synchronous blocking send operations: both processors wait for data to be send to a receiver that is not ready to accept the message

Blocking and non-blocking versions of send/recv operations are available in message passing libraries: compare connection-oriented with rendezvous (telephone) to connectionless (mailbox)

Programming Model 4

- Hybrid systems: clusters of SMPs
- Shared memory within SMP, message passing outside
- Programming model with three choices:
  - □ Treat as “flat” system: always use message passing, even within an SMP
    - Advantage: ease of programming and portability
    - Disadvantage: ignores SMP memory hierarchy and advantage of UMA shared address space
  - □ Program in two layers: shared memory programming and message passing
    - Advantage: better performance (use UMA/NUMA intelligently)
    - Disadvantage: harder (and ugly!) to program
  - □ Program in three layers: SIMD (e.g. SSE instructions) per core, shared memory programming between cores on an SMP node, and message passing between nodes

Programming Model 4



```
shared a[1..N/numnodes]          } Shared part
private n = N/numnodes/numprocs
private x[1..n]                  }
private lo = (pid-1)*n           } Processor-local part
private hi = lo+n                }
x[1..n] = f(a[lo..hi])           } Vector (SIMD) part
A[pid] := sum(x[1..n])           }
send A[pid] to node1
```

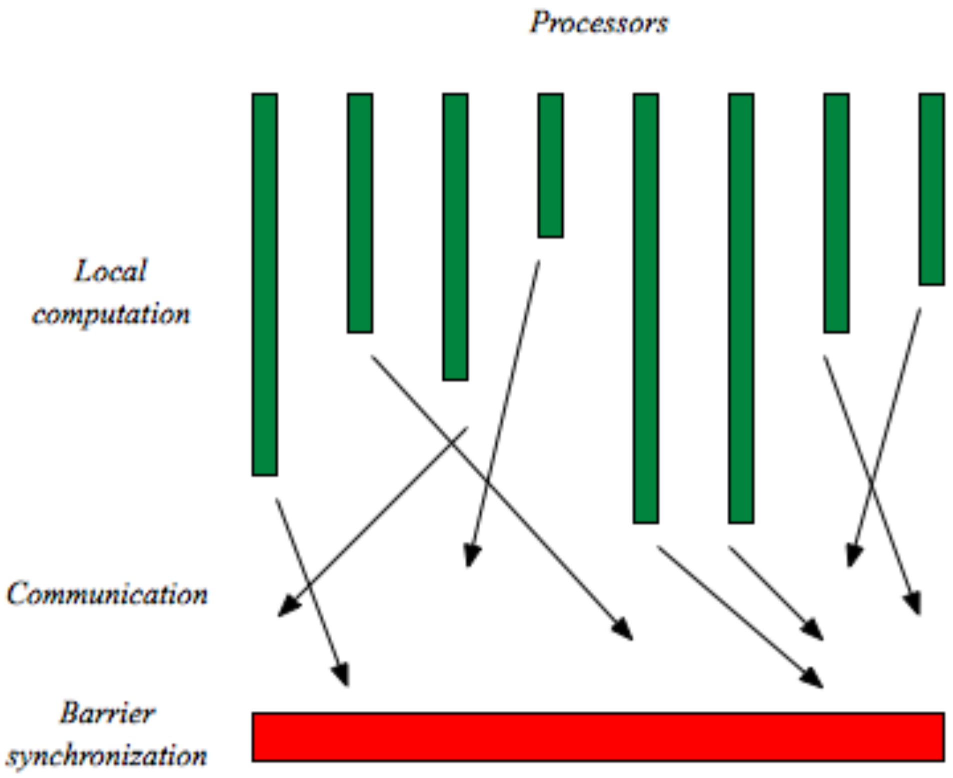
```
A := 0
if node=1 and pid=1
  for j = 1..numnodes
    for i = 1..numprocs
      receive Aj from node(j)
      A := A + Aj
```

Extra code for node 1 proc 1

Programming Model 5

- *Bulk synchronous processing* (BSP)
- A BSP *superstep* consists of three *phases*
  1. Compute phase: processes operate on local data (also read access to shared memory on SMP)
  2. Communication phase: all processes cooperate in exchange of data or reduction of global data
  3. Barrier synchronization
- A parallel program is composed of supersteps
  - Ensures that computation and communication phases are completed before the next superstep
- Simplicity of data parallel programming, without the restrictions

Programming Model 5



- The cost of a BSP superstep s is composed of three parts
  - □ w_s local computation cost of s
  - □ h_s is the number of messages send in superstep s
  - □ l is the barrier cost
- The total cost of a program with S supersteps is

$$W + Hg + Sl = \sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + Sl$$

where g is the communication cost such that it takes gh time to send h messages

Summary

- Goal is to distinguish the programming model from underlying hardware
- Message passing, data parallel, BSP
  - Objective is portable *correct* code
- Hybrid
  - Tuning for the architecture
  - Objective is portable *fast* code
  - Algorithm design challenge (less uniformity)
  - Implementation challenge at all levels (fine to coarse grain)
    - Blocking at loop and data level (compiler and programmer)
    - SIMD vectorization at loop level (compiler and programmer)
    - Shared memory programming for each node (OpenMP)
    - Message passing between nodes (MPI)