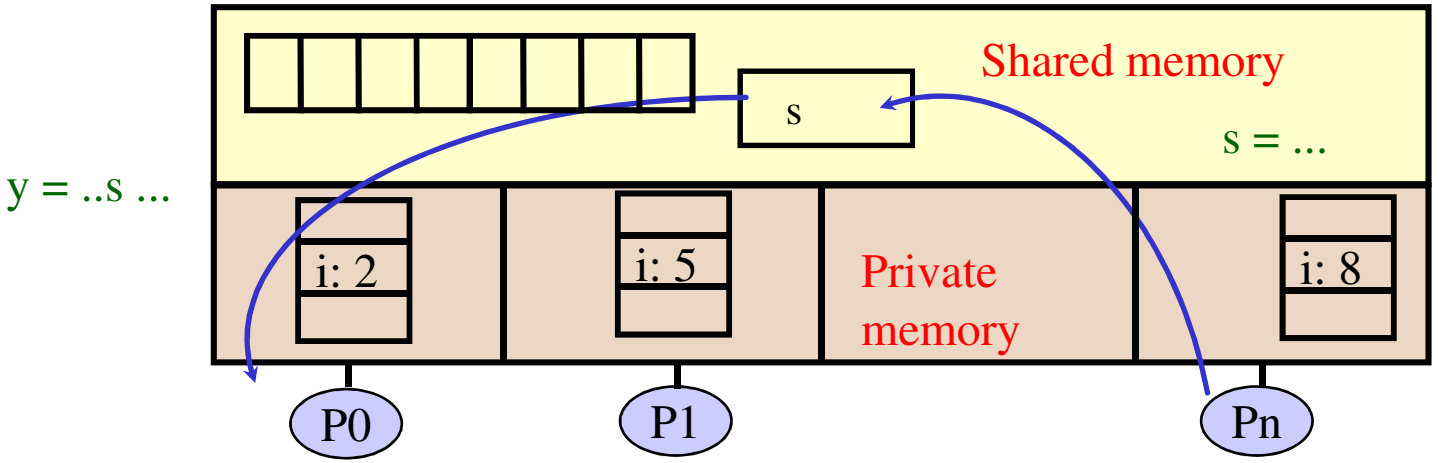
Parallel Programming Models

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?

Programming Model 1: Shared Memory

- Program is a collection of **threads** of control
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

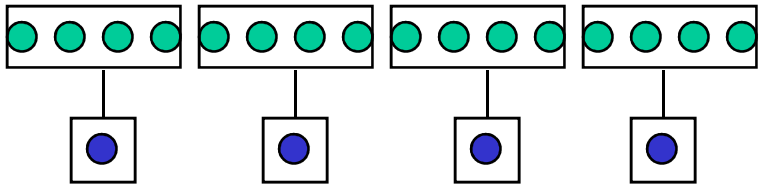


A Simple Example

- Shared memory strategy:
  - small number p << n=size(A) processors
  - attached to single memory

$$\sum_{i=0}^{n-1} f\ (\ A\ [\ i\ ])$$

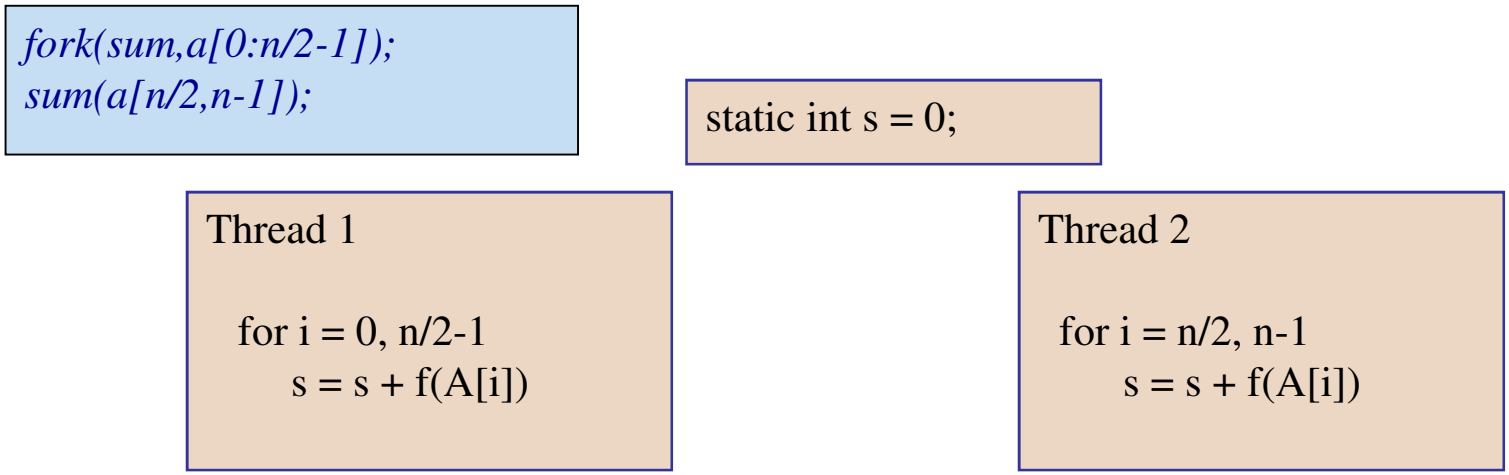
- Parallel Decomposition:
  - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each proc
  - Each computes independent "private" results and partial sum.
  - Collect the p partial sums and compute a global sum.



Two Classes of Data:

- Which are logically shared data?
- Which are logically private data?

Shared Memory "Code"



- What is the problem with this program?

Shared Memory "Code"

```
static int s = 0;
```

```
Thread 1

    local_s1= 0
    for i = 0, n/2-1
```

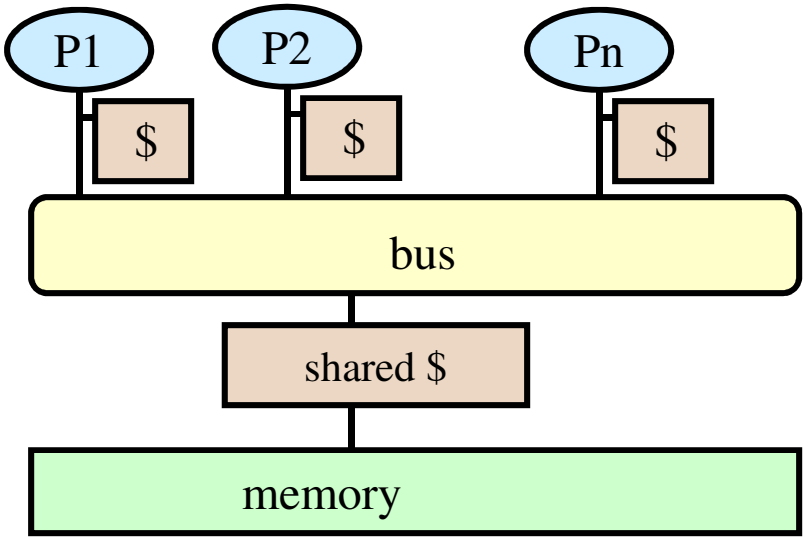
```
Thread 2

    local_s2 = 0
    for i = n/2, n-1
```

- How to solve the race condition?

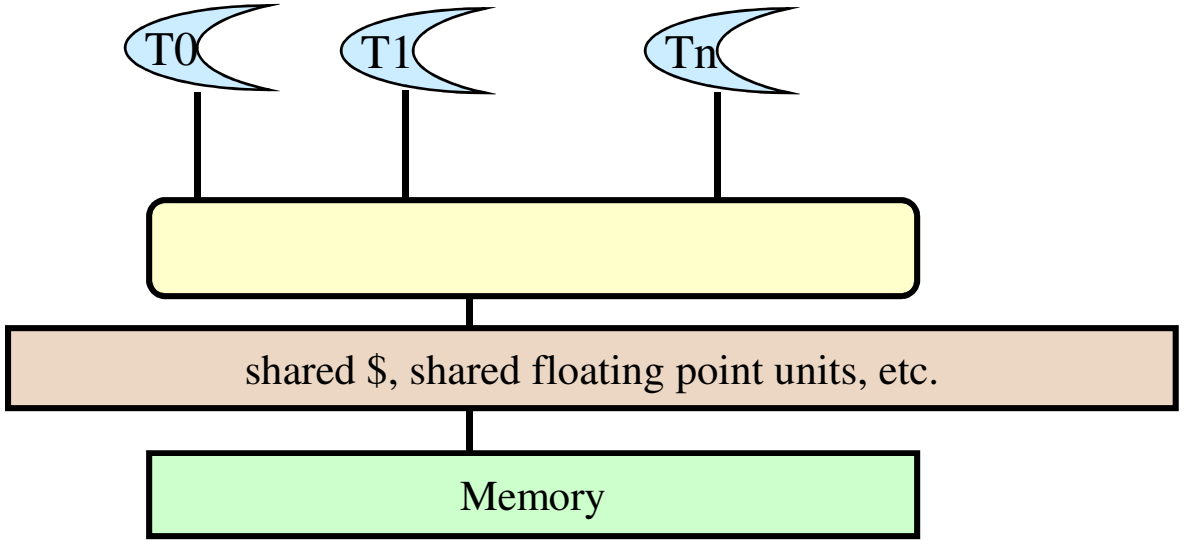
Machine Model 1a: Shared Memory

- Processors all connected to a large shared memory
  - Typically called Symmetric Multiprocessors (SMPs)
  - Multicore chips, except that all caches are shared
  - E.g., UMA
- Difficulty scaling to large numbers of processors
  - <= 32 processors typical
- Cost: much cheaper to access data in cache than main memory.



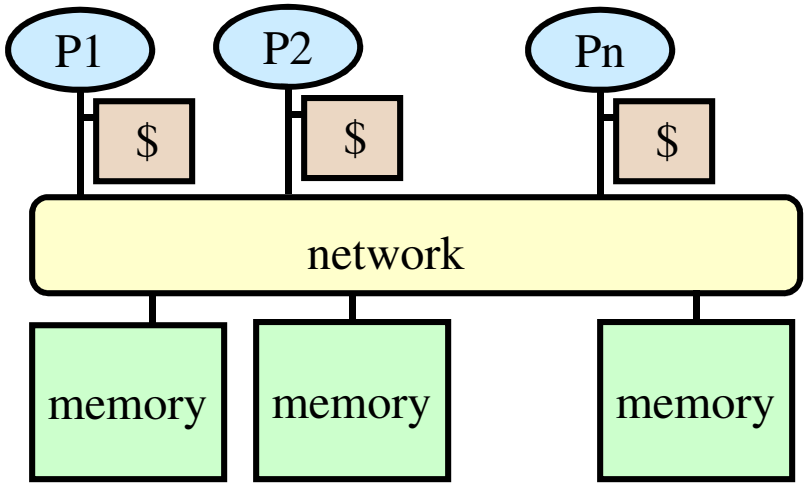
Machine Model 1b: Multithreaded Processor

- Multiple thread "contexts" without full processors
- Memory and some other state is shared
- Sun Niagra processor
  - Up to 64 threads all running simultaneously (8 threads x 8 cores)
  - In addition to sharing memory, they share floating point units
- Cray MTA and Eldorado processors (for HPC)



Machine Model 1c: Distributed Shared Memory

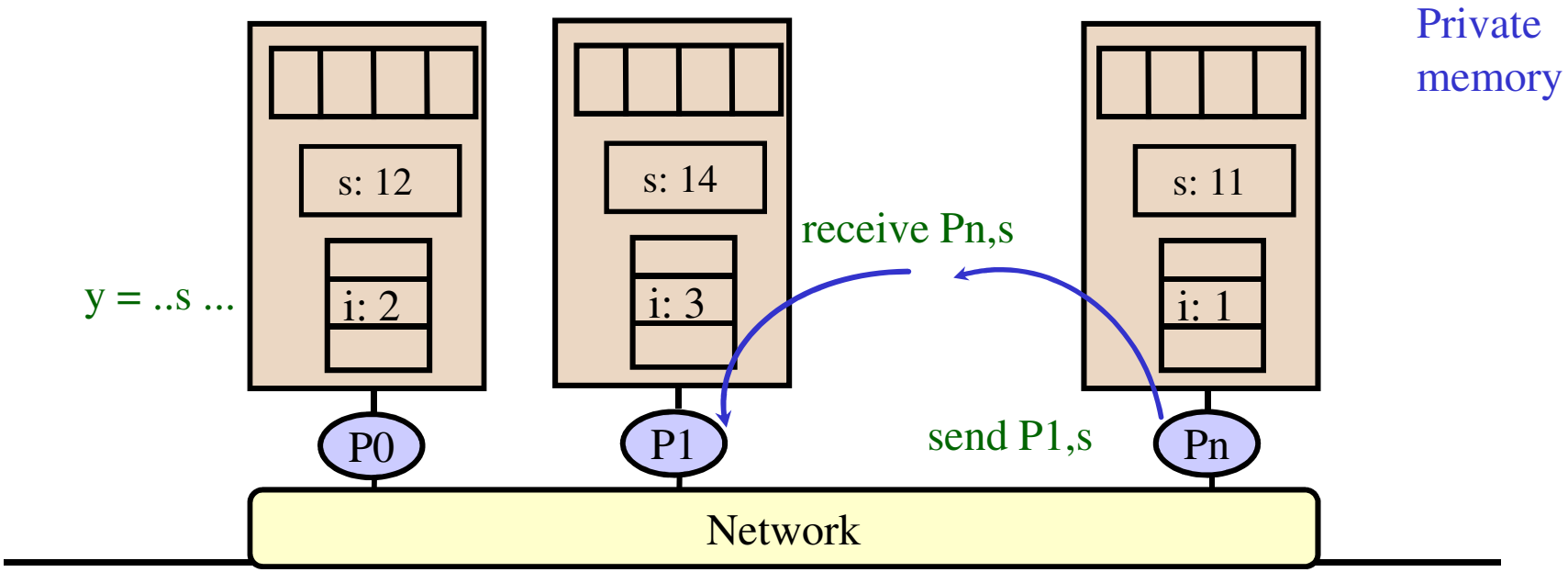
- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
  - E.g., NUMA
- SGI is canonical example (+ research machines)
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is *cache coherency protocols*



Cache lines (pages) must be large to amortize overhead
➔
locality still critical to performance

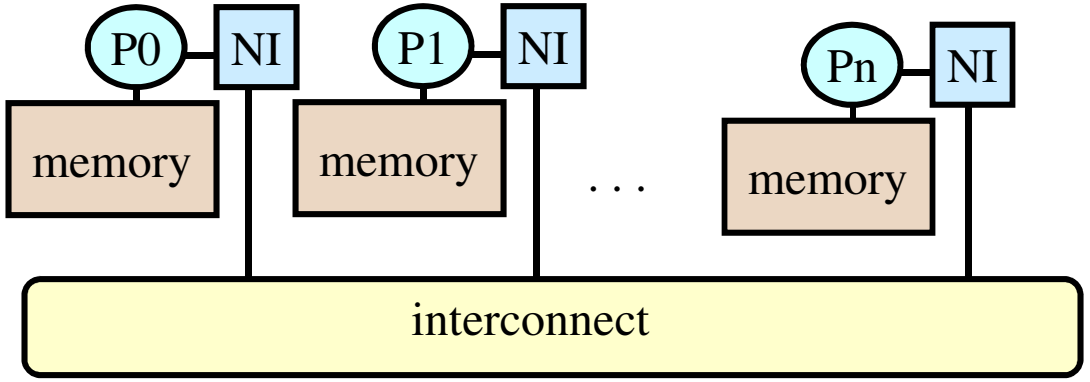
Programming Model 2: Message Passing

- Program consists of a collection of named processes
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event



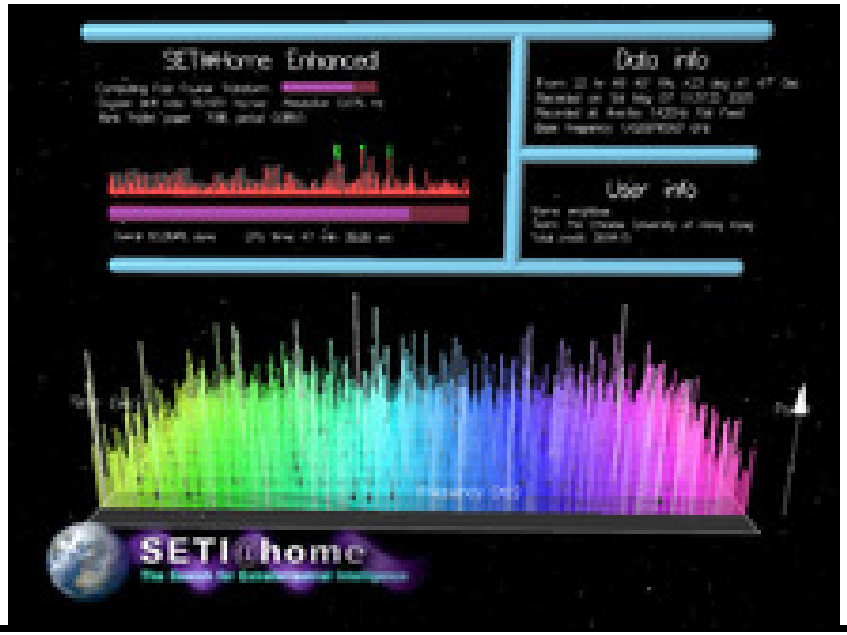
Machine Model 2a: Distributed Memory

- Cray XT4, XT 5
- PC Clusters (Berkeley NOW, Beowulf)
- IBM SP-3, Millennium, CITRIS are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor's memory
  - Aka "distributed address space machines"
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Machine Model 2b: Internet/Grids

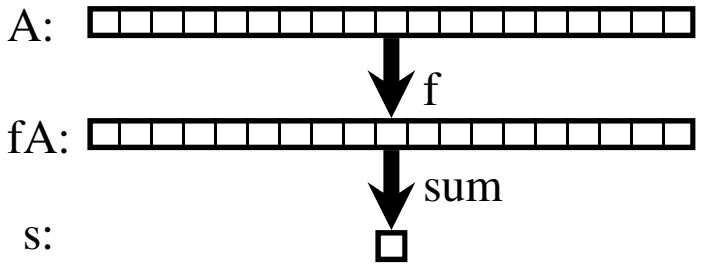
- SETI@Home: Running on 500,000 PCs
  - ~1000 CPU Years per Day
  - 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope



Programming Model 3: Data Parallel

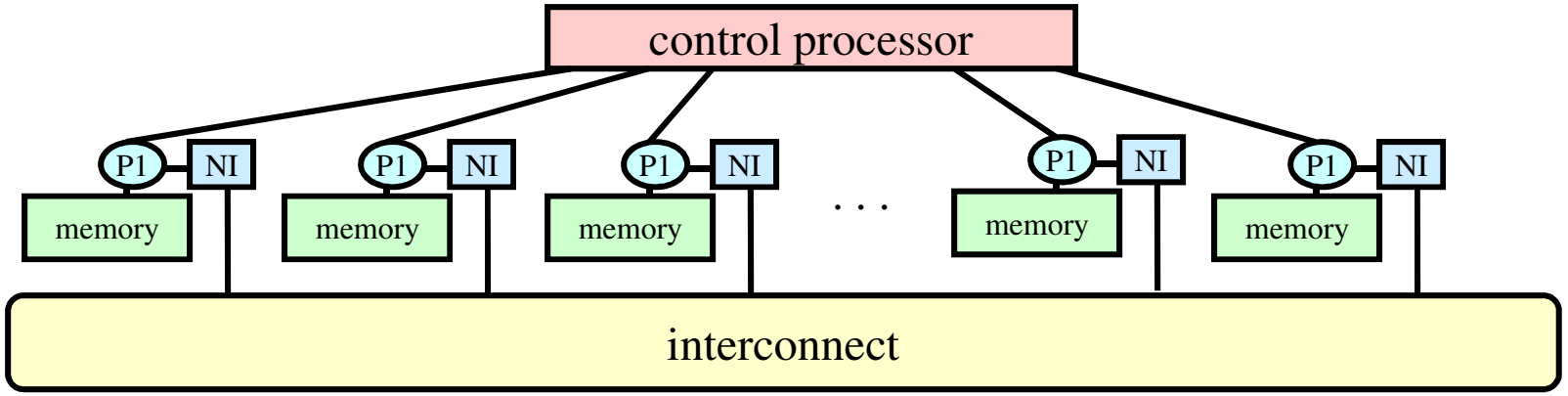
- Single thread of control consisting of parallel operations.
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit – statements executed synchronously
- Drawbacks:
  - ???

A = array of all data
fA = f(A)
s = sum(fA)



Machine Model 3a: SIMD System

- A large number of (usually) small processors.
  - A single "control processor" issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.
- Originally machines were specialized to scientific computing, few made (e.g.,CM2)
- Programming model can be implemented in the compiler



Machine Model 3b: Vector Machines

- Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specific large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel
- Historically important
  - Overtaken by MPPs in the 90s
- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1
  - At a small scale in SIMD media extensions to microprocessors
    - SSE, SSE2 (Intel: Pentium/IA64)
  - At a larger scale in GPUs
- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn't have to

Programming Model 4: Hybrids

- These programming models can be mixed
  - Message passing (MPI) at the top level with shared memory within a node is common
  - New DARPA HPCS languages mix data parallel and threads in a global address space
  - Global address space models can (often) call message passing libraries or vice verse
  - Global address space models can be used in a hybrid mode
    - Shared memory when it exists in hardware
    - Communication (done by the runtime system) otherwise
- For better or worse….

Machine Model 4: Hybrid machines

- Multicore/SMPs are a building block for a larger machine with a network
- Common names:
  - CLUMP = Cluster of SMPs
- Many modern machines look like this:
  - Millennium, IBM SPs, NERSC Franklin, Hopper
- What is an appropriate programming model #4 ???
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.
- Graphics or game processors may also be building block

Together…

- Parallel programming models
  - Shared memory (Pthreads and openMP)
  - Message passing (MPI)
  - Massive data parallelism (GPU programming)
  - Hybrid
- Distributed programming models
  - Cloud programming, MapReduce

Data Dependence

- Data dependence
  - True dependence
  - Anti-dependence
  - Output dependence
  - Loop-carried dependence

When can 2 statements execute in parallel?

- On one processor:

  statement 1;

  statement 2;

- On two processors:

processor1:	processor2:
statement1;	statement2;

When can 2 statements execute in parallel?

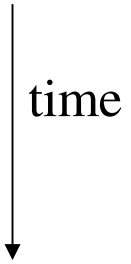
- Possibility 1

Processor1:

statement1;

Processor2:

statement2;



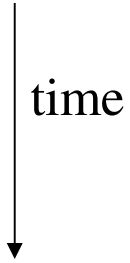
- Possibility 2

Processor1:

statement1;

Processor2:

statement2:



When can 2 statements execute in parallel?

- Their order of execution must not matter!
- In other words,

  statement1; statement2;

  must be equivalent to

  statement2; statement1;

Example 1

```
a = 1;
b = 2;
```

Example 2

```
a = 1;
b = a;
```

Example 3

```
a = f(x);
b = a;
```

Example 4

```
b = a;
a = 1;
```

Example 5

```
a = 1;
a = 2;
```

True dependence

Statements S1, S2

S2 has a true dependence on S1

iff

?

Anti-dependence

Statements S1, S2.

S2 has an anti-dependence on S1

iff

?

Output Dependence

Statements S1, S2.

S2 has an output dependence on S1

iff

?

When can 2 statements execute in parallel?

S1 and S2 can execute in parallel

iff

there are no dependences between S1 and S2

- true dependences
- anti-dependences
- output dependences

Some dependences can be removed.

Example 6

- Most parallelism occurs in loops.

```
for(i=0; i<100; i++)
    a[i] = i;
```

Example 7

```
for(i=0; i<100; i++) {
    a[i] = i;
    b[i] = 2*i;
}
```

Example 8

```
for(i=0;i<100;i++) a[i] = i;
for(i=0;i<100;i++) b[i] = 2*i;
```

Example 9

```
for(i=0; i<100; i++)
    a[i] = a[i] + 100;
```

Example 10

```
for( i=1; i<100; i++ )
  a[i] = f(a[i-1]);
```

Loop-carried dependence

- A loop-carried dependence is a dependence that is present only if the statements are part of the execution of a loop.
- Otherwise, we call it a loop-independent dependence.
- Loop-carried dependences prevent loop iteration parallelization.

Example 11

```
for(i=0; i<100; i++ )
  for(j=1; j<100; j++ )
        a[i][j] = f(a[i][j-1]);
```

Example 12

```
for( j=1; j<100; j++ )
  for( i=0; i<100; i++ )
        a[i][j] = f(a[i][j-1]);
```

Level of loop-carried dependence

- Is the nesting depth of the loop that carries the dependence.
- Indicates which loops can be parallelized.

Be careful … Example 13

```
printf(“a”);
printf(“b”);
```

Be careful … Example 14

```
a = f(x);
b = g(x);
```

Be careful … Example 15

```
for(i=0; i<100; i++)
  a[i+10] = f(a[i]);
```

Be careful … Example 16

```
for( i=1; i<100;i++ ) {
        a[i] = …;
        ... = a[i-1];
}
```

Be careful … Example 17

```
for( i=0; i<100; i++ )
        a[i] = f(a[indexa[i]]);
```

An aside

- Parallelizing compilers analyze program dependences to decide parallelization.
- In parallelization by hand, user does the same analysis.
- Compiler more convenient and more correct
- User more powerful, can analyze more patterns.

To remember

- Statement order must not matter.
- Statements must not have dependences.
- Some dependences can be removed.
- Some dependences may not be obvious.

Summary

1. Parallel platforms (~hardware) and programming models (~software)
   - Note: Parallel machine may or may not be tightly coupled to programming model

2. Data dependence

- Reading: Kumar – ch 1; Foster – ch 1