Lecture 3: Parallel Architectures and Shared Memory Programming

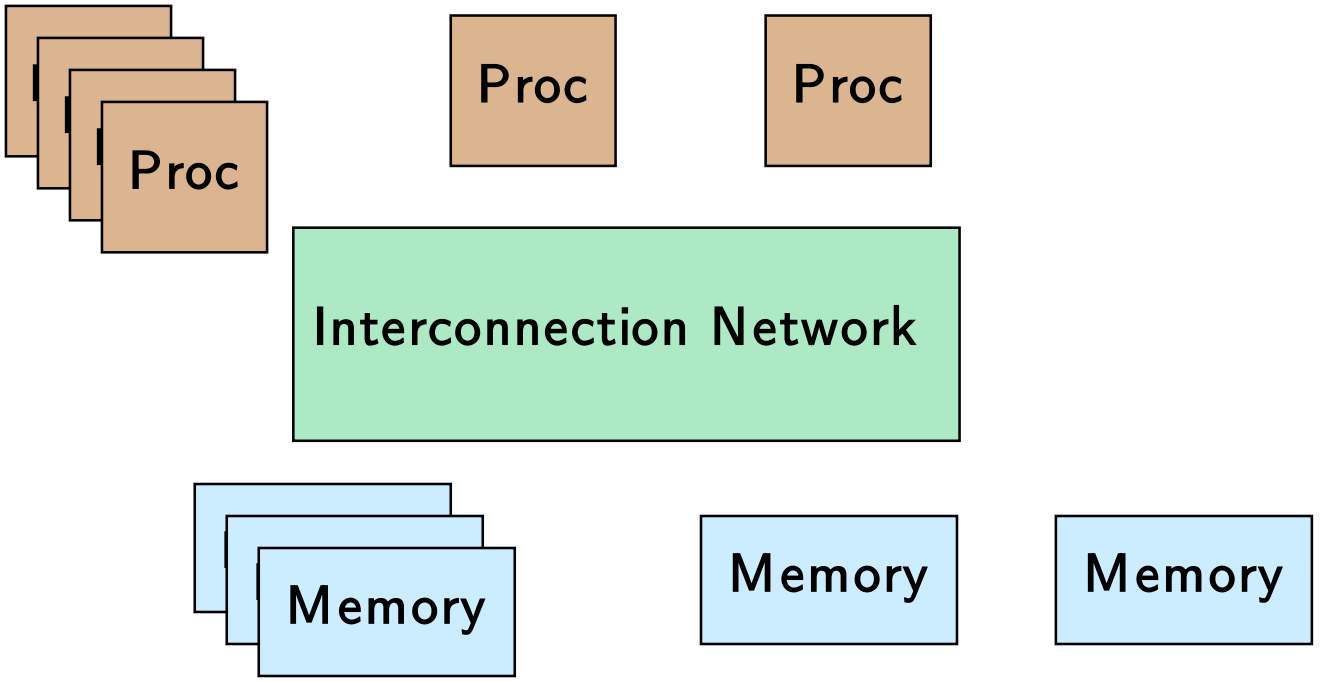
Part I: Parallel Machines and Programming Models

Outline

- Overview of programming models, APIs, and machines*
  - Programming models:
    - Shared memory
    - Shared address space
    - Message passing/distributed memory
    - Data parallel
    - Hybrid

- *Note: Parallel machine may or may not be tightly coupled to programming model
  - Historically, tight coupling
  - Today, portability is important

A generic parallel architecture



- Where is the memory physically located?
- Is it connected directly to processors?
- What is the connectivity of the network?

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?

Simple Example

- Consider applying a function f to the elements of an array A and then computing its sum:

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

- Questions:
  - Where does A live? All in single memory? Partitioned?
  - What work will be done by each processors?
  - They need to coordinate to get a single result, how?

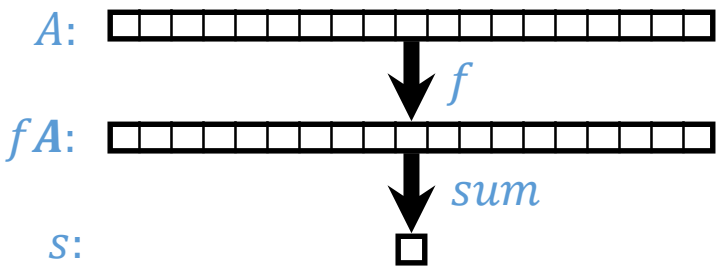
"map"

A = array of all data

$fA = f(A)$

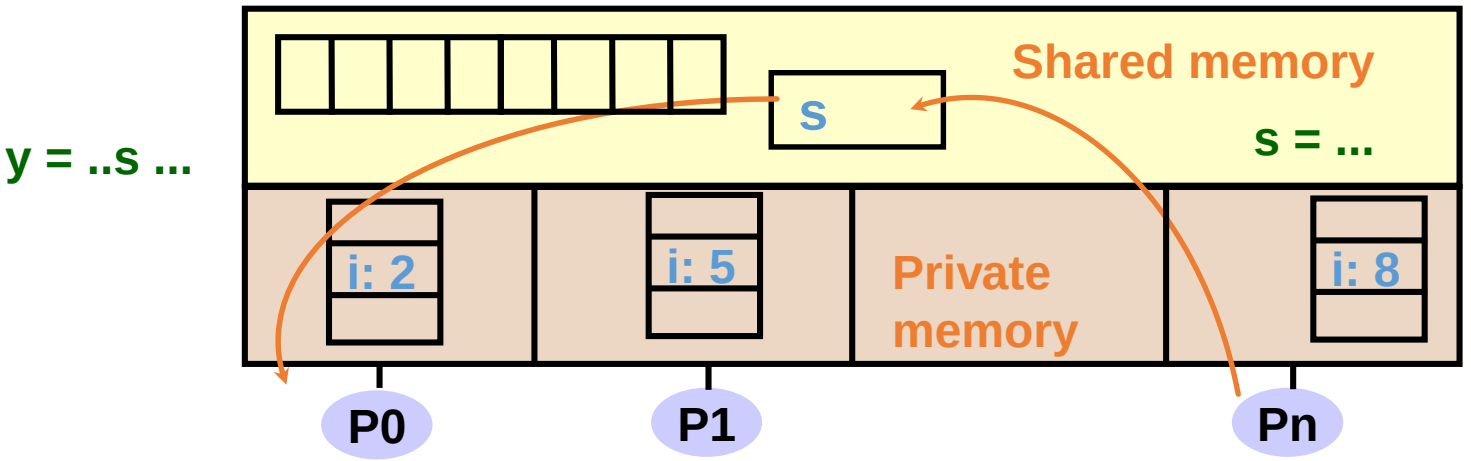
$s = sum(fA)$

"reduce"



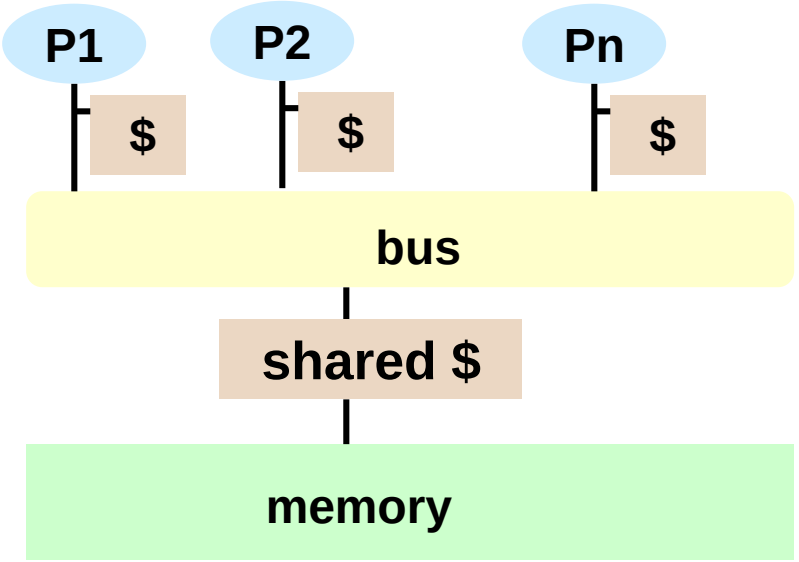
Programming Model 1: Shared Memory

- Program is a collection of threads of control.
- 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



Typical Shared Memory Machine Model

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



Note: $ = cache

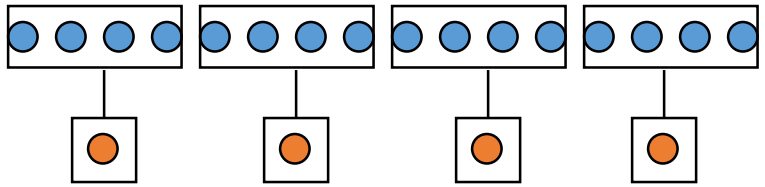
Simple Example

- Shared memory strategy:
  - small number $p \ll n = \text{size}(A)$ processors
  - attached to single memory

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

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

Two Classes of Data:



- Logically Shared
  - e.g., original n numbers, the global sum
- Logically Private
  - e.g., the individual function evaluations

Shared Memory "Code" for Computing a Sum

```
fork(sum(a[0:n/2-1]),
sum(a[n/2,n-1]));
```

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

Thread 1

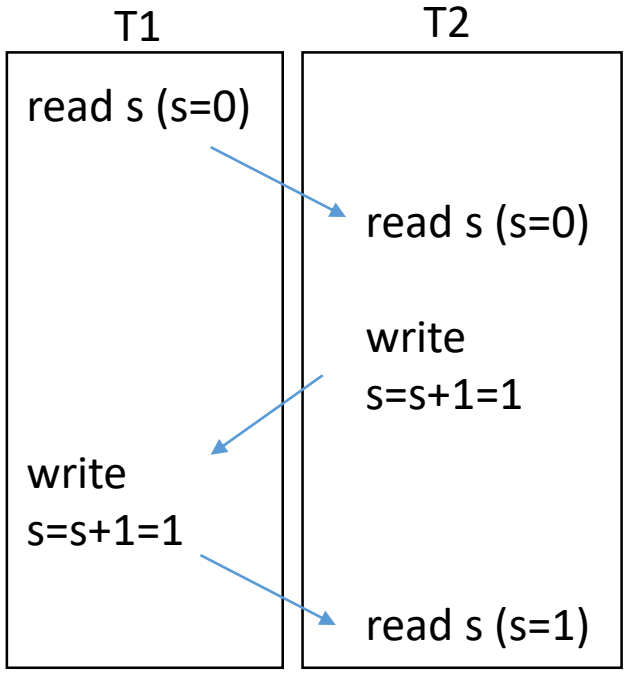
```
for i = 0, n/2-1
    s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1
    s = s + f(A[i])
```

- What is the problem with this program?

- A race condition or data race occurs when:
  - Two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously



Shared Memory "Code" for Computing a Sum

$A =$ | 3 | 5 | $f(x) = x^2$

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

Thread 1

```
….
 compute f([A[i]) and put in reg0    9
 reg1 = s                            0
 reg1 = reg1 + reg0                  9
 s = reg1                            9
…
```

Thread 2

```
…
 compute f([A[i]) and put in reg0   25
  reg1 = s                           0
  reg1 = reg1 + reg0                25
  s = reg1                          25
 …
```

- Assume $A = [3,5]$, $f(x) = x^2$, and $s = 0$ initially
- For this program to work, s should be $3^2 + 5^2 = 34$ at the end
  - but it may be 34, 9, or 25
- The *atomic* operations are reads and writes
  - += operation is *not* atomic
  - All computations happen in (private) registers

Improved Code for Computing a Sum

```
static int s = 0;
static lock lk;
```

Why not do lock Inside loop?

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1
unlock(lk);
```

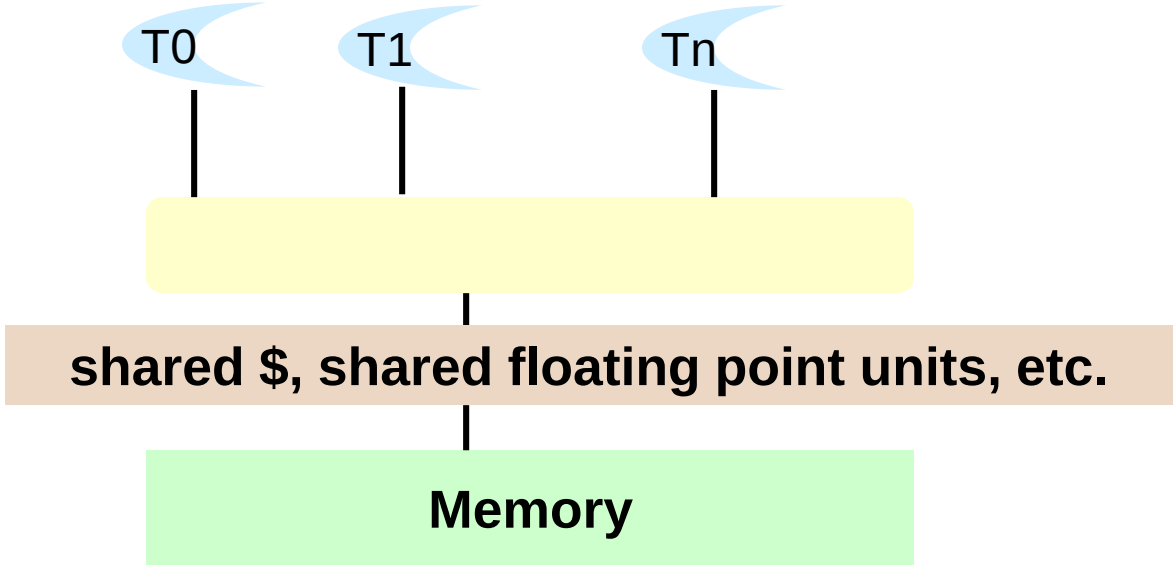
Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + f(A[i])
lock(lk);
s = s +local_s2
unlock(lk);
```

- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

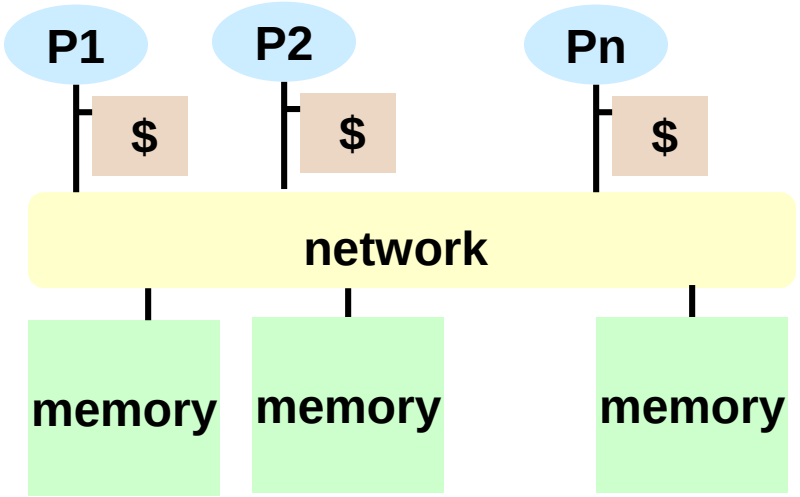
Another Machine Model: Multithreaded Processor

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



Another Machine Model: 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
- SGI was canonical example (+ research machines)
  - Scaled to 512 procs (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is *cache coherency protocols* – how to keep cached copies of the same address consistent



Cache lines (pages) must be large to amortize overhead

➔

locality still critical to performance

Shared Memory Programming APIs

- POSIX threads (pthreads)
  - C library
  - allows a program to control multiple different flows of work (threads) that overlap in time
  - functions:
    - Thread management - creating, joining threads etc.
    - Mutexes
    - Synchronization between threads using read/write locks and barriers
- Intel TBB (Thread Building Blocks)

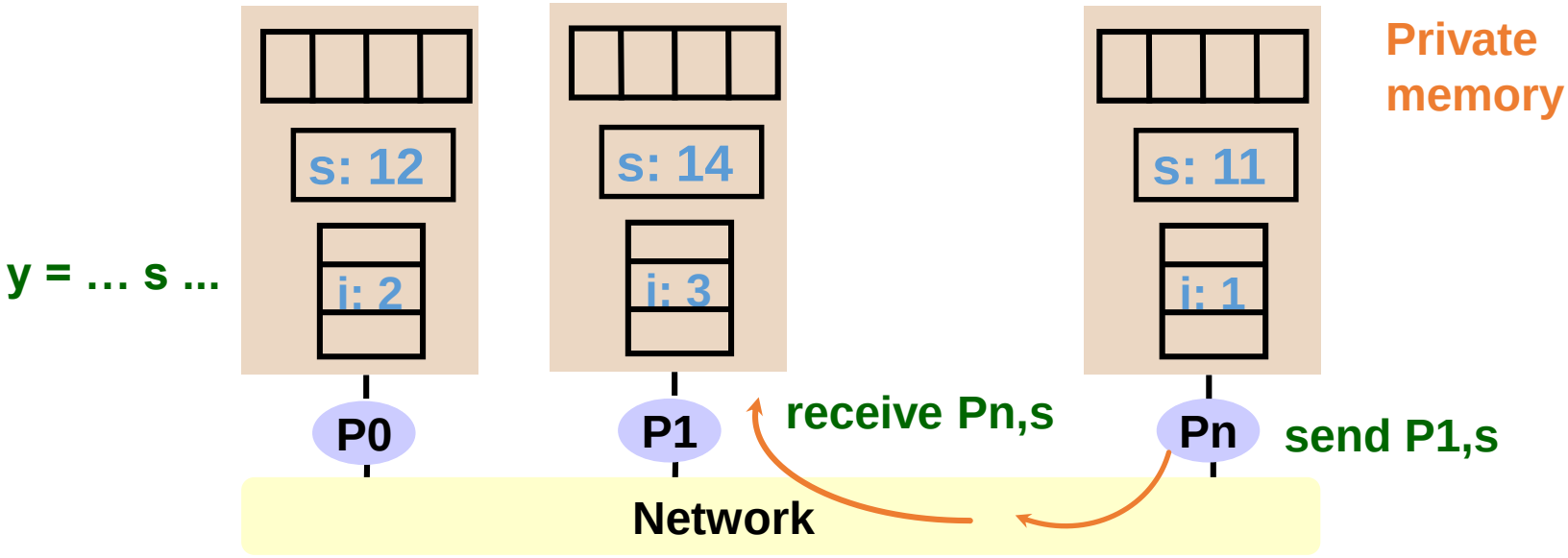
Shared Memory Programming APIs

- OpenMP (http://www.openmp.org/)
  - supports multi-platform shared-memory parallel programming in C/C++ and Fortran.
  - portable, scalable model with a simple and flexible interface
  - section of code that is meant to run in parallel is marked accordingly, with a compiler directive ("pragma")

```c
#include <stdio.h>
#include <omp.h>
int main(void)
{
        #pragma omp parallel
        printf("Hello, world.\n");
        return 0;
}
```

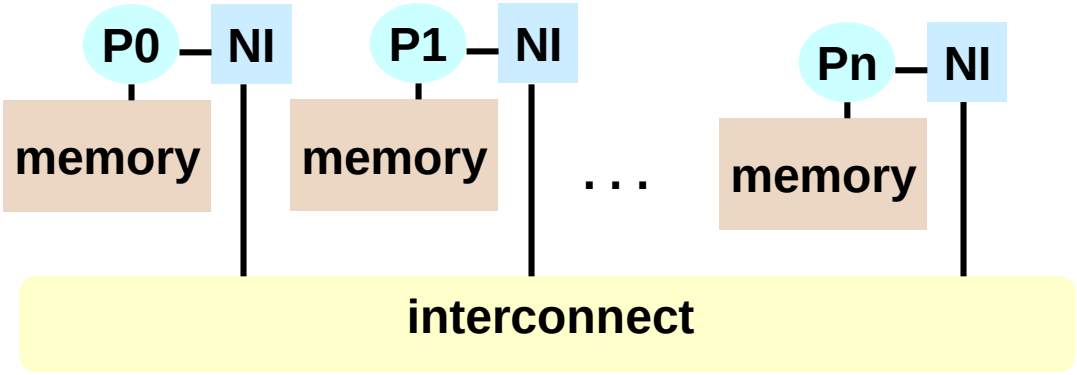
Programming Model 2: Message Passing

- 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.
  - MPI (Message Passing Interface) is the most commonly used SW



Typical Distributed Memory Machine Model

- PC Clusters (like Karlín cluster)
- Most or all of the Top500 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.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Computing $s = f(A[1]) + f(A[2])$ on each processor

- First possible solution – what could go wrong?

Processor 1

```
xlocal = f(A[1])
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

Processor 2

```
xlocal = f(A[2])
send xlocal, proc1
receive xremote, proc1
s = xlocal + xremote
```

- If send/receive acts like the (old) telephone system? The post office?
- Second possible solution

Processor 1

```
xlocal = f(A[1])
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

Processor 2

```
xlocal = f(A[2])
receive xremote, proc1
send xlocal, proc1
s = xlocal + xremote
```

- What if there are more than 2 processors?

MPI – the de facto standard

MPI has become the de facto standard for parallel computing using message passing (www.mpi-forum.org)

Pros and Cons of standards

- MPI created finally a standard for applications development in the HPC community → portability
- The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation

Programming Model reflects hardware!

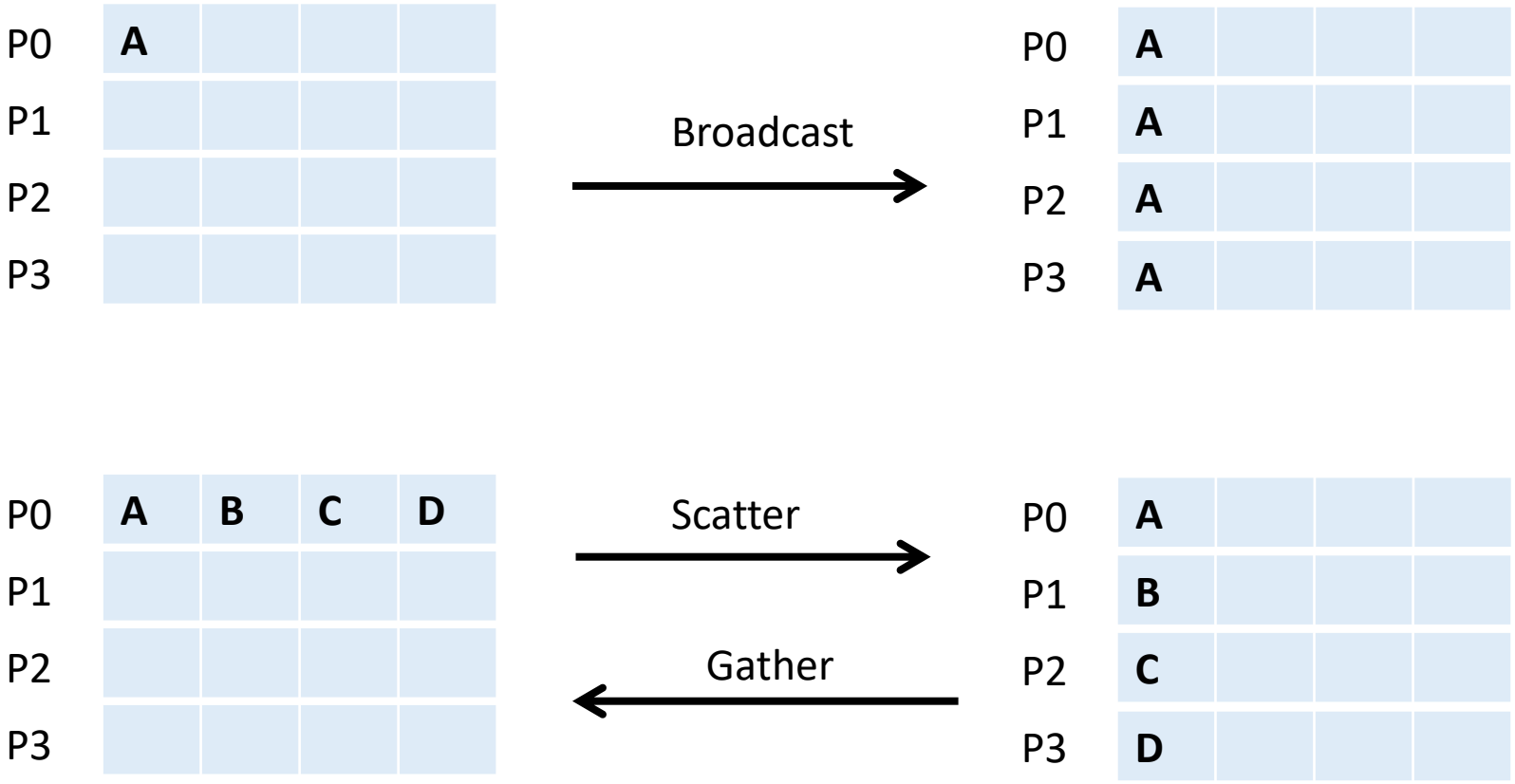
"I am not sure how I will program a Petaflops computer, but I am sure that I will need MPI somewhere"

– Horst Simon 2001

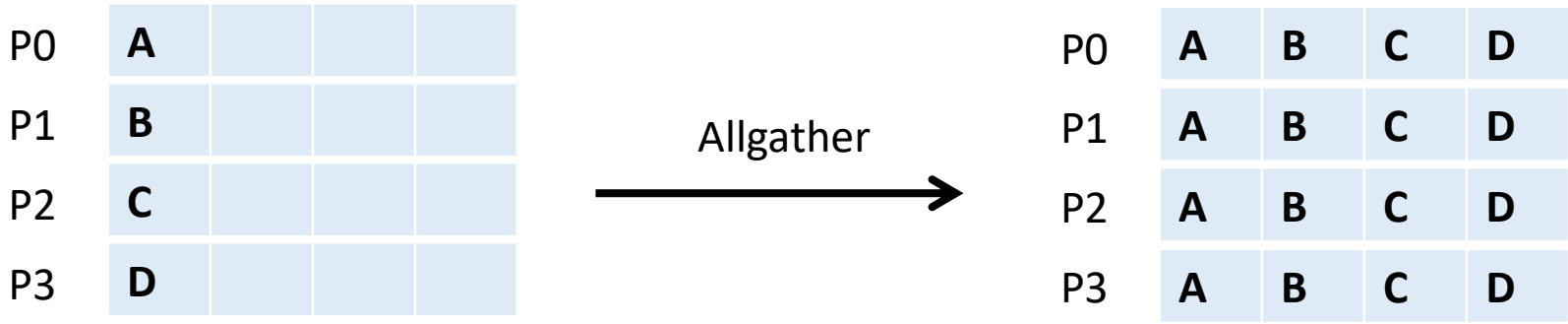
Message Passing Interface (MPI)

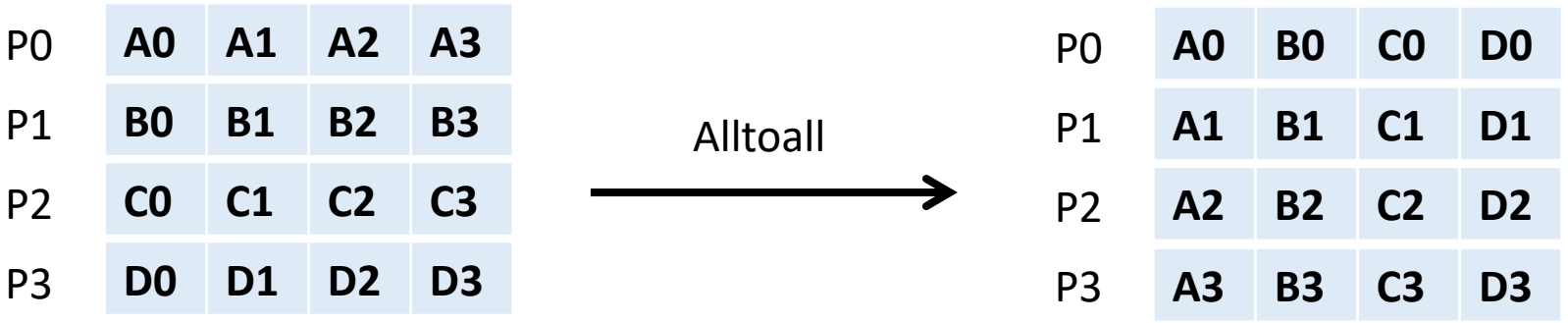
- MPI Standard defines syntax and semantics of a core of library routines in C, C++, and Fortran
  - Many implementations exist, many open source (Open MPI on Karlín cluster)
- Both "point to point" and "collective" communication supported
  - Point to point: communication between two specific processes (e.g., proc. 1 sends a message to proc. 2, uses MPI_Send(), MPI_Recv())
  - Collectives: communication among all processes (e.g., MPI_Reduce())
- Bindings available in higher level languages (e.g., python implementations of MPI include pyMPI, mpi4py, pypar, MYMPI, etc.)

MPI Collectives

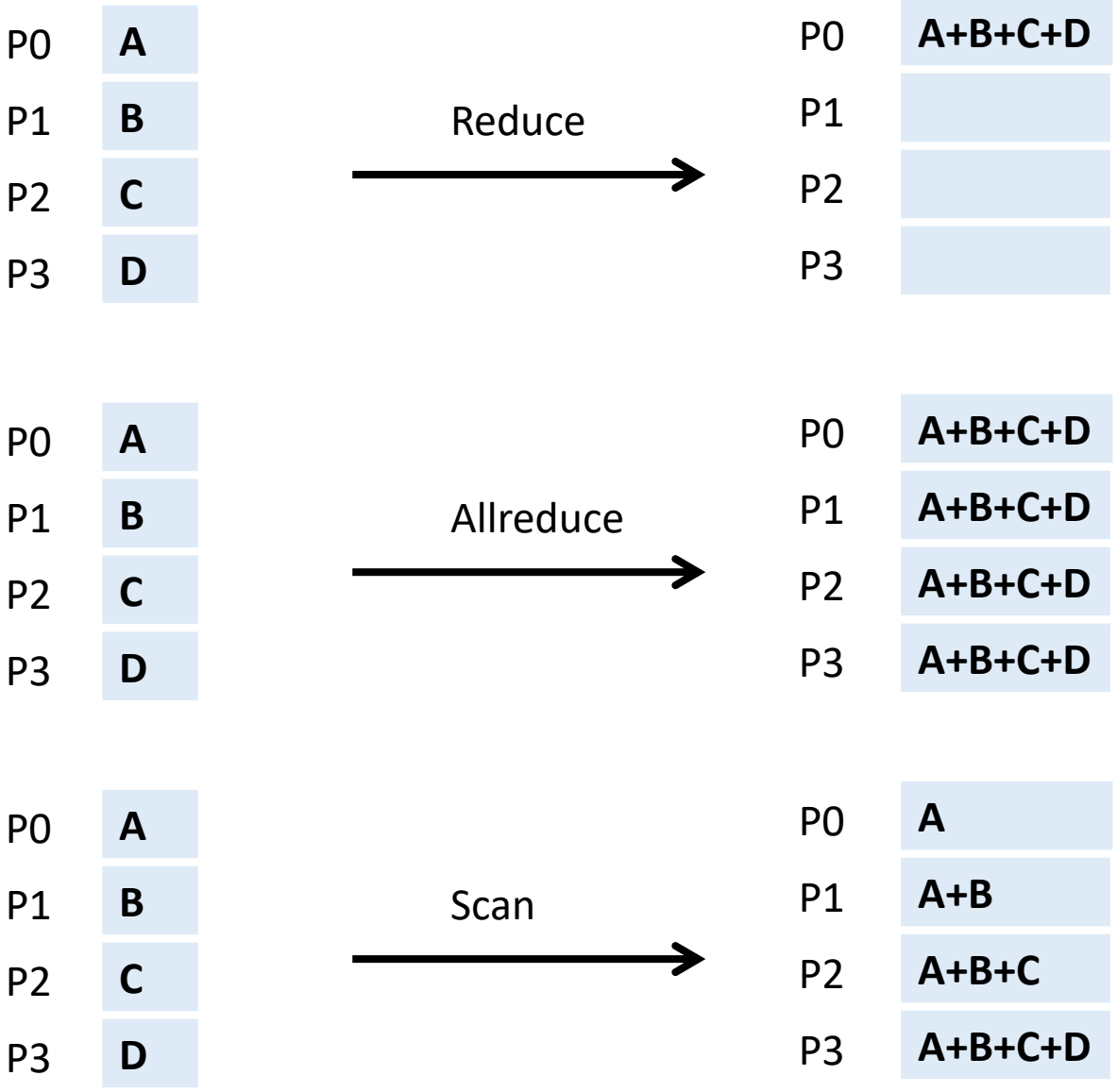


MPI Collectives



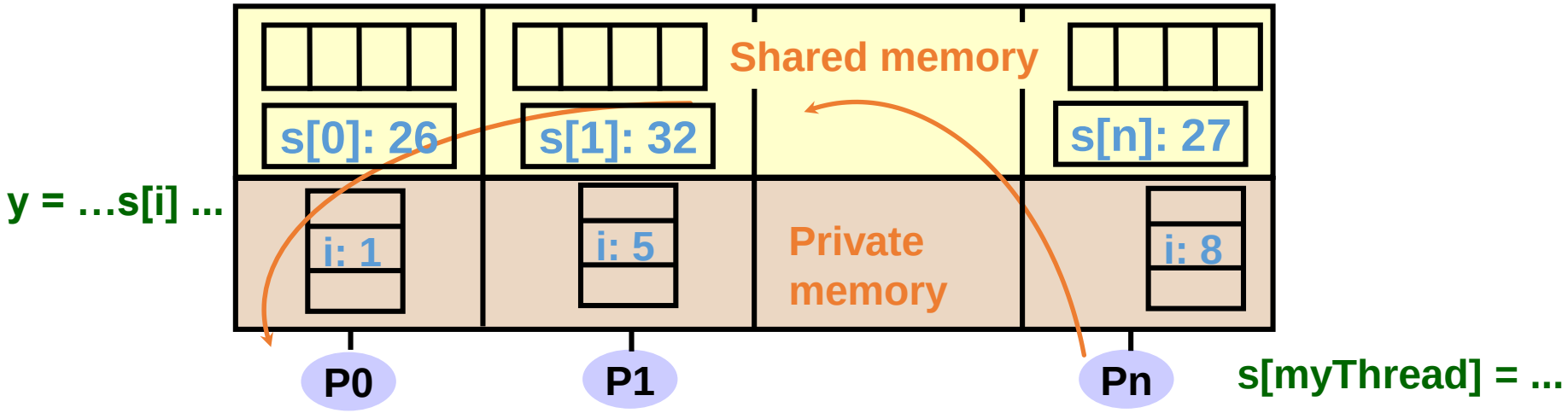


MPI Collectives



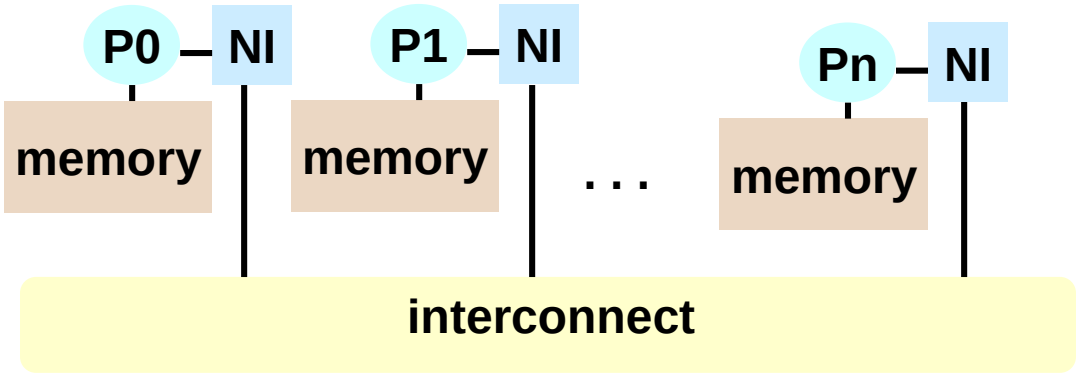
Programming Model 2a: Global Address Space

- Program consists of a collection of named threads.
  - **Usually fixed at program startup time**
  - **Local and shared data, as in shared memory model**
  - **But, shared data is partitioned over local processes**
  - **Cost models says remote data is expensive**
- Examples: UPC, Titanium, Co-Array Fortran
- Global Address Space programming is an intermediate point between message passing and shared memory



Typical Global Address Space Machine

- The network interface supports RDMA (Remote Direct Memory Access)
  - NI can directly access memory without interrupting the CPU
  - One processor can read/write memory with one-sided operations (put/get)
  - Not just a load/store as on a shared memory machine
    - Continue computing while waiting for memory op to finish
  - Remote data is typically not cached locally



Global address space may be supported in varying degrees

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:
  - **Not all problems fit this model**
  - **Difficult to map onto coarse-grained machines**

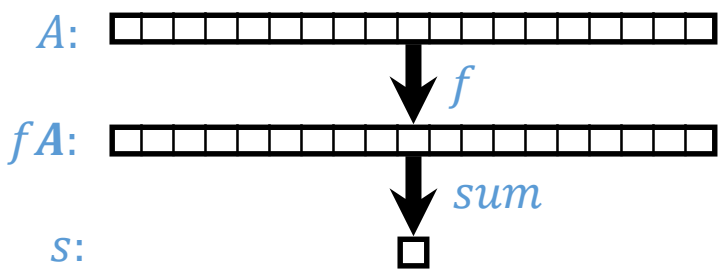
"map"

A = array of all data

$fA = f(A)$

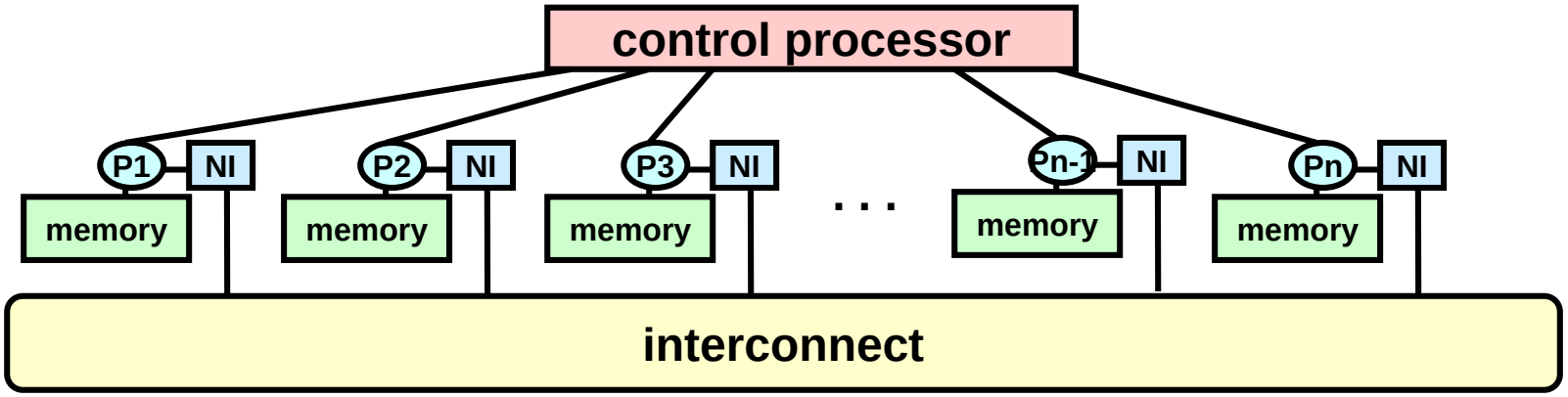
$s = sum(fA)$

"reduce"



"SIMD" Data Parallel Machine

- 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 manufactured
- Programming model can be implemented in the compiler
  - **mapping n-fold parallelism to p processors, n >> p, but it's hard**

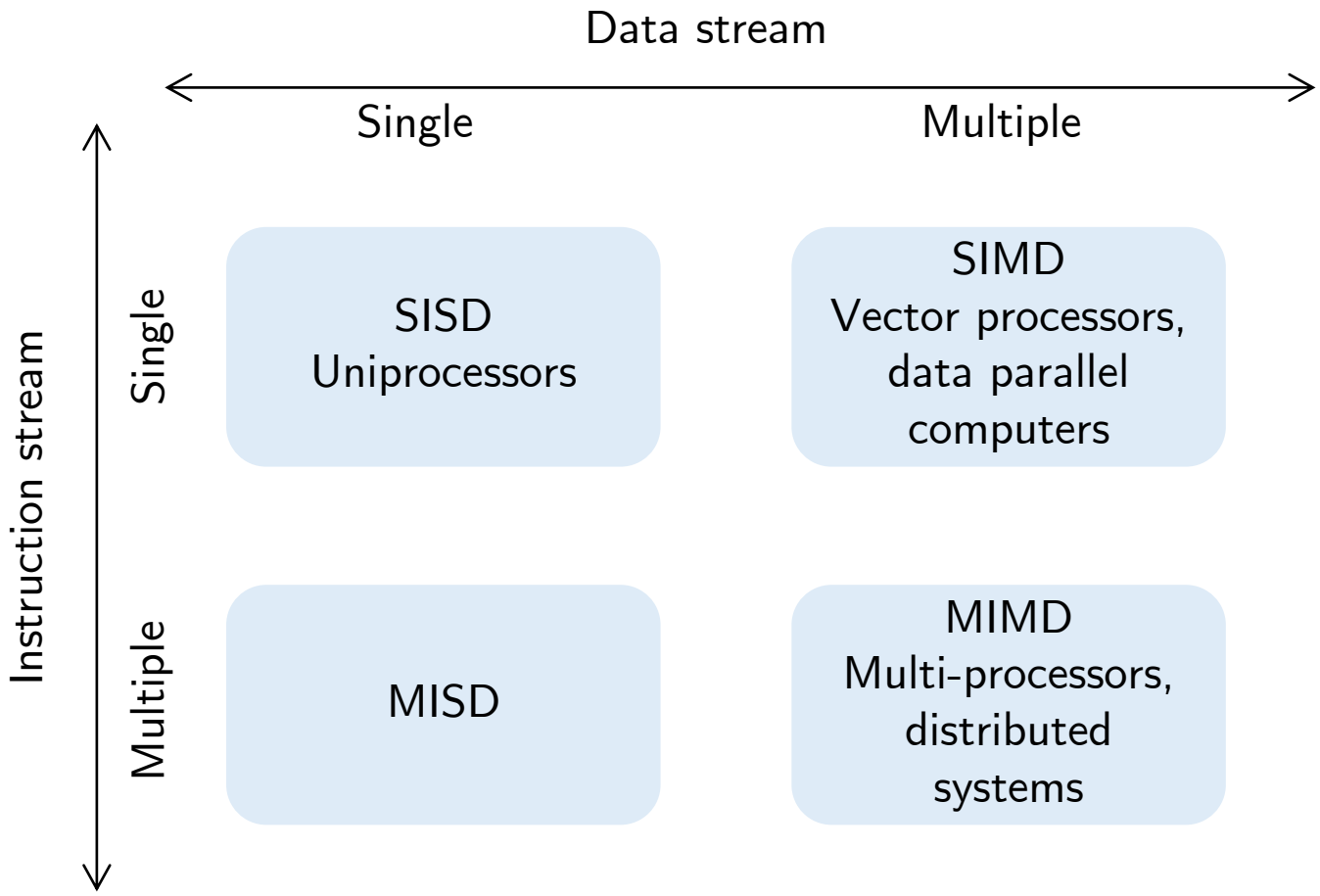


Vector Machines

- Vector architectures are based on a single processor
  - **Multiple functional units**
  - **All performing the same operation**
  - **Instructions may specify large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel**
- Historically important
  - **Overtaken by MPPs (massively parallel processors) in the 90s**
- Re-emerging in recent years
  - **At a large scale in the Earth Simulator (NEC SX6) and Cray X1**
  - **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

Flynn's Taxonomy

- classification of computer architectures, proposed by Michael J. Flynn in 1966.



Flynn's Taxonomy: MIMD

- MIMD can be further divided into two categories:

- **Single program, multiple data streams (SPMD)**
  - Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data.
  - The most common style of parallel programming

- **Multiple programs, multiple data streams (MPMD)**
  - Multiple autonomous processors simultaneously operating at least 2 independent programs
  - Typical example: one node is "manager", which runs one program that farms out data to all other nodes which all run a second "worker" program and return their results to the "manager"

Hybrid machines

- Multicore/SMPs are a building block for a larger machine with a network
- Old name:
  - CLUMP = Cluster of SMPs
- Many modern machines look like this (most of Top500)
- What is an appropriate programming model ???
  - Treat machine as "flat", always use message passing, even within a node (simple, but ignores an important part of memory hierarchy).
  - Shared memory within node, but message passing outside of an node.
- GPUs may also be building block
  - June 2022 Top500: 34% have accelerators

Programming Model 4: Hybrids

- Programming models can be mixed
  - Message passing (MPI) at the top level with shared memory within a node is common

- For better or worse
  - Supercomputers often programmed this way for peak performance

Caution

- Not all programming models work equally well on all machine architectures
- Not all problems are suited to all programming models

What about GPU and Cloud?

- GPU's big performance opportunity is data parallelism
  - Most programs have a mixture of highly parallel operations, and some not so parallel
  - GPUs provide a threaded programming model (CUDA) for data parallelism to accommodate both
  - Current research attempting to generalize programming model to other architectures, for portability (OpenCL)
- Cloud computing lets large numbers of people easily share $O(10^5)$ machines
  - MapReduce was first programming model: data parallel on distributed memory
  - More flexible models (Hadoop, Spark, ...) invented since then

Take-Away

- Three basic conceptual models
  - Shared memory
  - Distributed memory
  - Data parallel

  and hybrids of these

- All of these machines rely on dividing up work into parts that are:
  - Mostly independent (little synchronization)
  - About same size (load balanced)
  - Have good locality (little communication)

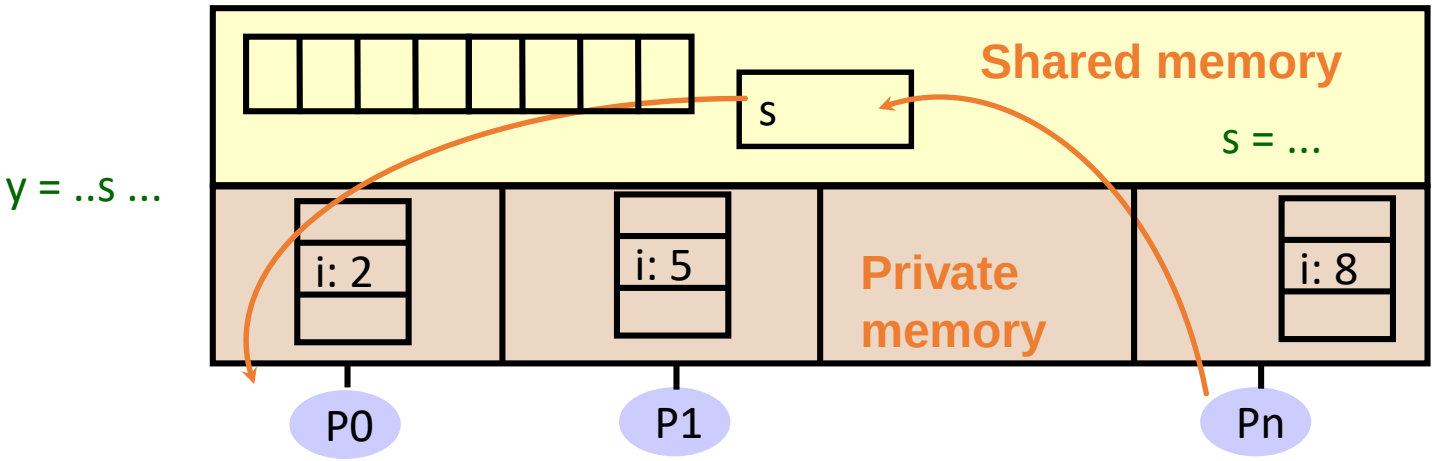
Part II: Shared Memory Programming

Outline

- Shared memory parallelism with threads
- What and why OpenMP?
- Parallel programming with OpenMP
- Introduction to OpenMP
  1. Creating parallelism
  2. Parallel Loops
  3. Synchronizing
  4. Data sharing

Recall 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



What's a thread? A process?

Processes are independent execution units that contain their own state information and their own address space. They interact via interprocess communication mechanisms (generally managed by the operating system). One process may contain many threads. Processes are given system resources.

All threads within a process share the same address space, and can communicate directly using shared variables.

What is state?

- instruction pointer
- Register file (one per thread)
- Stack pointer (one per thread)

Shared Memory Languages

- pthreads - POSIX (Portable Operating System Interface for Unix) threads; heavyweight, more clumsy
- PGAS languages - Partitioned Global Address Space: UPC, Titanium, Co-Array Fortran; not yet popular enough, or efficient enough
- OpenMP - newer standard for shared memory parallel programming, lighter weight threads, not a programming language but an API for C and Fortran

OpenMP Overview

OpenMP is an API for multithreaded, shared memory parallelism.

OpenMP = Open specifications for MultiProcessing

- A set of **compiler directives** inserted in the source program
  - pragmas in C/C++ (pragma = compiler directive external to prog. lang. for giving additional info)
- **Library functions**
- **Environment variables**

Goal is standardization, ease of use, portability.

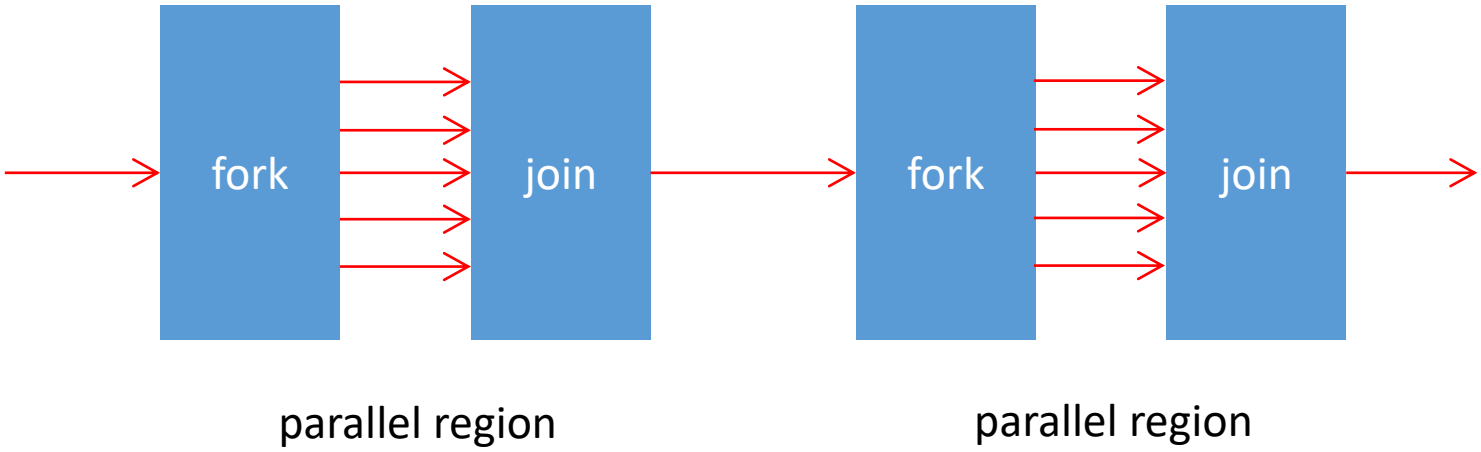
Significant parallelism possible with just 3 or 4 directives.

Works on SMPs (symmetric multiprocessors) and DSMs (distributed shared memory systems).

Allows fine and coarse-grained parallelism; loop level as well as explicit work assignment to threads as in SPMD (single program, multiple data).

Basic Idea

- Explicit programmer control of parallelization using fork-join model of parallel execution
  - all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
  - FORK: master thread creates team of parallel threads
  - JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread.



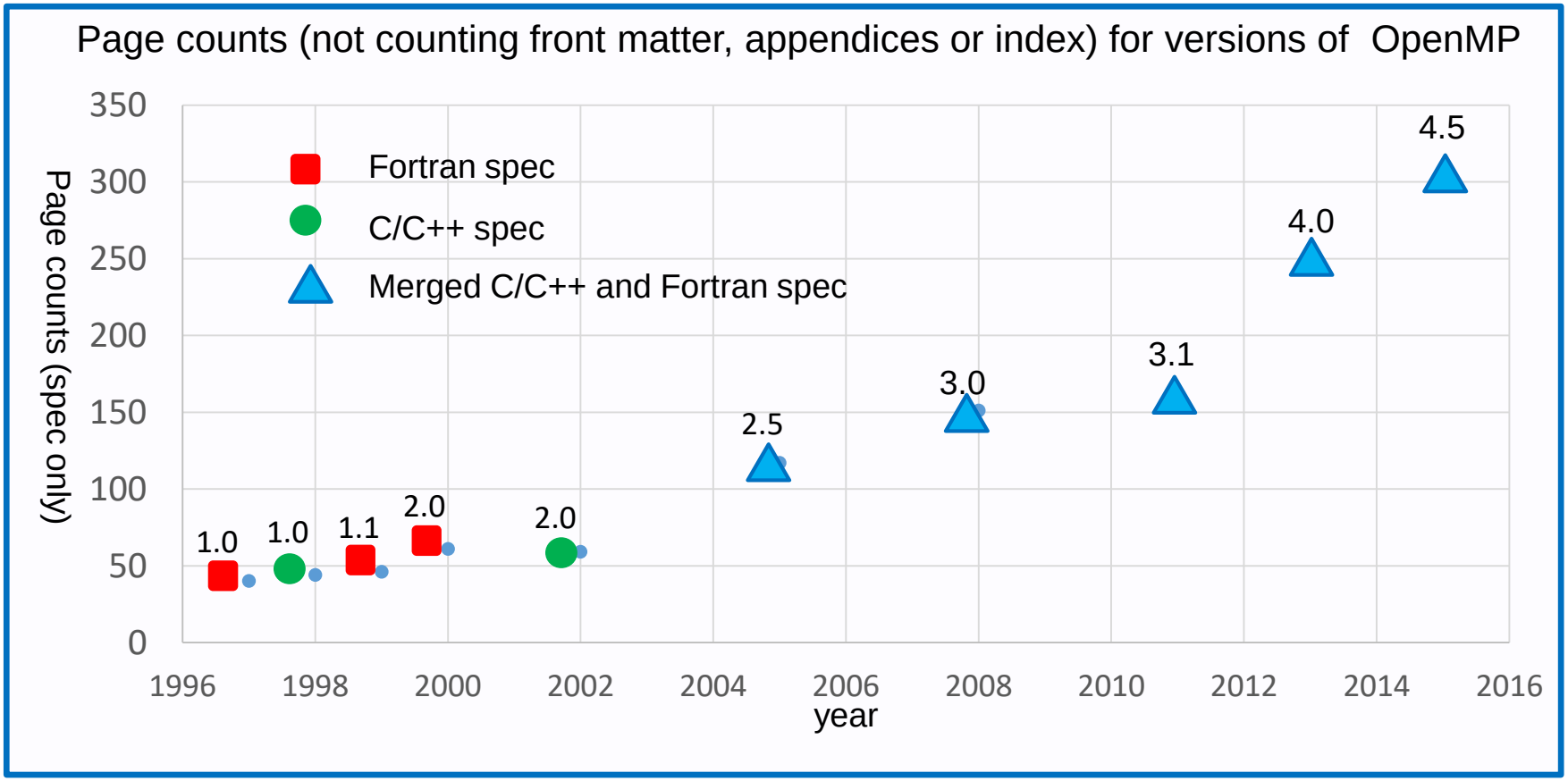
Basic Idea

- User inserts directives telling compiler how to execute statements
  - which parts are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads
  - #pragma omp in C and !$omp in Fortran
- Compiler generates explicit threaded code
- Rule of thumb: One thread per core (2 or 4 with hyperthreading)
- Dependencies in parallel parts require synchronization between threads

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!



OpenMP 5.2 (November 2021) is actually **669 pages**.

The OpenMP Common Core: Most OpenMP programs only use these 19 items

OpenMP pragma, function, or clause
#pragma omp parallel
int omp_get_thread_num() int omp_get_num_threads()
double omp_get_wtime()
setenv OMP_NUM_THREADS N
#pragma omp barrier #pragma omp critical
#pragma omp for #pragma omp parallel for
reduction(op:list)
schedule(dynamic [,chunk]) schedule (static [,chunk])
private(list), firstprivate(list), shared(list)
nowait
#pragma omp single
#pragma omp task #pragma omp taskwait

OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives.

Compiler directives
`#pragma omp construct [clause [clause]…]`
Example

`#pragma omp parallel private(x)`
`{`
 	
`}`

Function prototypes and types:
`#include <omp.h>`

- Most OpenMP constructs apply to a structured block.
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.

Simple Example

```
#include <stdio.h>
#include <omp.h>

int main(){
    #pragma omp parallel
    printf("Hello world from thread %d\n", omp_get_thread_num());

    return 0;
}
```

Compile:

```
gcc -fopenmp helloworld.c
```

Simple Example

```
carson@r3d3:[~]: export OMP_NUM_THREADS=4

carson@r3d3:[~]: ./helloworld
Hello world from thread 1
Hello world from thread 0
Hello world from thread 2
Hello world from thread 3

carson@r3d3:[~]: ./helloworld
Hello world from thread 0
Hello world from thread 2
Hello world from thread 3
Hello world from thread 1
```

Setting the Number of Threads

Environment Variables:

```
setenv OMP_NUM_THREADS 2 (cshell)
export OMP_NUM_THREADS=2 (bash shell)
```

Library call:

```
omp_set_num_threads(2)
```

Parallel Construct

```
#include <omp.h>

int main(){

  int var1, var2, var3;
  ...serial Code

  #pragma omp parallel private(var1, var2) shared (var3)
  {
       ...parallel section
  }

  ...resume serial code
}
```

Parallel Directives

- When a thread reaches a PARALLEL directive, it becomes the master and has thread number 0.
- All threads execute the same code in the parallel region (or use work-sharing constructs to distribute the work)
- There is an implied barrier at the end of a parallel section. Only the master thread continues past this point.
- If a thread terminates within a parallel region, all threads will terminates, and the result is undefined.
- Cannot branch into or out of a parallel region.

barrier - all threads wait for each other; no thread proceeds until all threads have reached that point

Parallel Directives

- If program compiled serially, OpenMP pragmas and comments ignored, stub library for omp library routines
- easy path to parallelization
- One source for both sequential and parallel helps maintenance

Work-Sharing Constructs

- work-sharing construct divides work among member threads. Must be done dynamically within a parallel region.
- No new threads launched. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.

3 types of work-sharing construct:

- for loop: share iterates of for loop ("data parallelism") iterates must be independent
- sections: work broken into discrete section, each executed by a thread ("functional parallelism")
- single: section of code executed by one thread only

FOR directive schedule example

```
#include <stdio.h>
#include <omp.h>

#define N 20

int main(){
        int sum = 0;
        int a[N], i;
        #pragma omp parallel for
        for (i = 0; i < N; i++){
                a[i]=i;
                printf("Iterate i=%d by thread %d\n", i,
                        omp_get_thread_num());
        }
        return 0;
}
```

FOR directive schedule example

```
iterate i=  9 by thread   3
iterate i= 10 by thread   3
iterate i= 11 by thread   3
iterate i=  6 by thread   2
iterate i=  7 by thread   2
iterate i=  8 by thread   2
iterate i=  0 by thread   0
iterate i=  1 by thread   0
iterate i=  2 by thread   0
iterate i= 12 by thread   4
iterate i= 13 by thread   4
iterate i= 14 by thread   4
iterate i=  3 by thread   1
iterate i=  4 by thread   1
iterate i= 15 by thread   5
iterate i= 16 by thread   5
iterate i= 17 by thread   5
iterate i= 18 by thread   6
iterate i= 19 by thread   6
iterate i=  5 by thread   1
```

- for loop with 20 iterations and 8 threads:
- 6 threads get 3 iterations, 1 thread gets 2, 1 gets none

OMP Directives

All directives:

```
#pragma omp directive        [clause ...]
                             if (scalar_expression)
                             private (list)
                             shared (list)
                             default (shared | none)
                             firstprivate (list)
                             reduction (operator: list)
                             copyin (list)
                             num_threads (integer-expression)
```

Directives are:

- Case sensitive
- Only one directive-name per statement
- Directives apply to at most one succeeding statement, which must be a structured block.
- Can be continued on succeeding lines with backslash ("\")

Default(none) example

```
#include <stdio.h>
#include <omp.h>

#define N 20

int main(){
        int sum = 0;
        int a[N], i;
        #pragma omp parallel for default(none) private(i)
        for (i = 0; i < N; i++){
                a[i]=i;
                printf("Iterate i=%d by thread %d\n", i,
                        omp_get_thread_num());
        }
        return 0;
}
```

Firstprivate example

```
#include <stdio.h>
#include <omp.h>
int main (void)
{

    int i = 10;
    #pragma omp parallel private(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }
    printf("i = %d\n", i);
    return 0;
}
```

thread 0: i = 0
thread 3: i = 32717
thread 1: i = 32717
thread 2: i = 1
i = 10

(another run of the same program)

thread 2: i = 1
thread 1: i = 1
thread 0: i = 0
thread 3: i = 32657
i = 10

Firstprivate example

```
#include <stdio.h>
#include <omp.h>
int main (void)
{

    int i = 10;
    #pragma omp parallel firstprivate(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }
    printf("i = %d\n", i);
    return 0;
}
```

```
thread 2: i = 10
thread 0: i = 10
thread 3: i = 10
thread 1: i = 10
i = 10
```

FOR directive

```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                private (list)
                firstprivate(list)
                lastprivate(list)
                shared (list)
                reduction (operator: list)
                nowait
```

SCHEDULE: describes how to divide the loop iterates

- **static** = divided into pieces of size chunk, and statically assigned to threads. Default is approx. equal sized chunks (at most 1 per thread)
- **dynamic** = divided into pieces of size chunk and dynamically scheduled as requested. Default chunk size 1.
- **guided** = size of chunk decreases over time. (Init. size proportional to the number of unassigned iterations divided by number of threads decreasing to chunk size)
- **runtime** = schedule decision deferred to runtime, set by environment variable OMP SCHEDULE.

FOR example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region

   #pragma omp for nowait
   for (i=0;i<n;i++)
     b[i] = += a[i];

   #pragma omp for nowait
   for (i=0;i<n;i++)
     x[i] = 1./y[i];

} // end parallel region  (implied barrier)
```

- Spawning tasks is expensive: reuse if possible.
- nowait clause: minimize synchronization.

SECTIONS directive

```
#pragma omp sections [clause ...]
                              private (list)
                              firstprivate(list)
                              lastprivate(list)
                              reduction (operator: list)
                              nowait
 {
   #pragma omp section
      structured block
   #pragma omp section
      structured block
 }
```

- implied barrier at the end of a SECTIONS directive, unless a NOWAIT clause used
- for different numbers of threads and SECTIONS some threads get none or more than one
- cannot count on which thread executes which section
- no branching in or out of sections

Sections example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region
   #pragma omp sections nowait
   {
      #pragma omp section
      for (i=0;i<n;i++)
        b[i] = += a[i];

      #pragma omp section
      for (i=0;i<n;i++)
        x[i] = 1./y[i];

   } // end sections
} // end parallel region
```

SINGLE directive

```
#pragma omp single [clause ...]
                           private (list)
                           firstprivate(list)
                           nowait

  structured block
```

- SINGLE directive says only one thread in the team executes the enclosed code
- useful for code that isn't thread-safe (e.g. I/O)
- rest of threads wait at the end of enclosed code block (unless NOWAIT clause specified)
- no branching in or out of SINGLE block

private example

- What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
      x = i*dx
      y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

private example

- What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
      x = i*dx
      y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

By default, x is shared variable (i is private).

Could have: Thread 0 set x for some i.

Thread 1 sets x for different i.

Thread 0 uses x but it is now incorrect.

private example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
      x = i*dx
      y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about i,dx,y?

private example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
      x = i*dx
      y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about i,dx,y?

By default dx,n,y shared.

dx,n used but not changed.

y changed, but independently for each i

firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
      x = i*dx
      y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
      x = i*dx
      y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

- Specifying dx private creates a new private variable for each thread, but it is not initialized.
- firstprivate clause creates private variables and initializes to the value from the master thread before the loop.
- lastprivate copies last value computed by a thread (for i=n) to the maser thread copy to continue execution.

Clauses

These clauses not strictly necessary but may be convenient (and may have performance penalties too).

- lastprivate private data is undefined after parallel construct. This gives it the value of last iteration (as if sequential) or sections construct (in lexical order).
- firstprivate pre-initialize private vars with value of variable with same name before parallel construct.
- default (none | shared). Then only need to list exceptions. (none is better habit).
- nowait suppress implicit barrier at end of work sharing construct. Cannot ignore at end of parallel region. (But no guarantee that if have 2 for loops where second depends on data from first that same threads execute same iterates)

More Clauses

- if (logical expr) true = execute parallel region with team of threads; false = run serially (loop too small, too much overhead)
- reduction for assoc. and commutative operators compiler helps out; reduction variable is shared by default (no need to specify).

```
#pragma omp parallel for default(none) shared(n,a) \
                                        reduction(+:sum)
for (i=0;i<n;i++)
        sum += a[i]
```

- Also other arithmetic and logical ops.
- copyprivate only with single direction. one thread reads and initializes private vars. which are copied to other threads before they leave barrier.
- threadprivate variables persist between different parallel sections (unlike private vars). (applies to global vars. must have dynamic false)

Synchronization

- Implicit barrier synchronization at end of parallel region (no explicit support for synch. subset of threads). Can invoke explicitly with

```
#pragma omp barrier
```

  All threads must see same sequence of work-sharing and barrier regions .

- critical sections: only one thread at a time in critical region

```
#pragma omp critical [(name)]
```

- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- locks: low-level run-time library routines (like mutex vars., semaphores)
- flush operation - forces the executing thread to make its values of shared data consistent with shared memory
- master (like single but not implied barrier at end)

At all these (implicit or explicit) synchronization points OpenMP ensures that threads have consistent values of shared data.

Critical Example

```
#pragma omp parallel sections
{
	#pragma omp section
	{
		task = produce_task();
		#pragma omp critical (task_queue)
		{
			insert_into_queue(task);
		}
	}
	#pragma omp section
	{
		#pragma omp critical (task_queue)
		{
			task = delete_from_queue(task);
		}
		consume_task(task);
	}
}
```

Atomic Examples

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
          ic = ic +1;
   }
```

`ic` incremented atomically

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
        ic = ic + bigfunc();
   }
```

`bigfunc` not atomic, only `ic` update

allowable atomic operations:

```
x = expr binop x          x++          x--
```

Atomic Example

```
int sum = 0;
#pragma omp parallel for shared(n,a,sum)
{
  for (i=0; i<n; i++){
  #pragma omp atomic
     sum = sum + a[i];
  }
}
```

Better to use a *reduction* clause:

```
int sum = 0;
#pragma omp parallel for shared(n,a)   \
                     reduction(+:sum)
{
  for (i=0; i<n; i++){
     sum += a[i];
  }
}
```

Locks

Locks control access to shared resources. Up to implementation to use spin locks (busy waiting) or not.

- Lock variables must be accessed only through locking routines:

```
omp_init_lock      omp_destroy_lock
omp_set_lock       omp_unset_lock     omp_test_lock
```

- In C, lock is a type `omp_lock_t` or `omp_nest_lock_t`
- initial state of lock is unlocked.
- `omp_set_lock(omp_lock_t *lock)` forces calling thread to wait until the specified lock is available. (Non-blocking version is `omp_test_lock`)

Examining and setting a lock must be *uninterruptible* operation.

Lock Example

```
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf("Thread %d - starting locked region\n", tid);

         printf("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

Deadlock

Runtime situation that occurs when a thread is waiting for a resource that will never be available. Common situation is when two (or more) actions are each waiting for the other to finish (for example, 2 threads acquire 2 locks in different order)

```
work1() { /* do some work */
        #pragma omp barrier
}
work2(){ /* do some work */
}
main(){
        #pragma omp parallel sections
        {
                #pragma omp section
                        work1();
                #pragma omp section
                        work2();
        }
}
```

Also livelock: state changes but no progress is made.

Nested Loops

Which is better (assuming m ≈ n)?

```
#pragma omp parallel for private(i)
for (j=0;j<m;j++)
      for (i=0;i<n;i++)
            a[j][i] = 0.;
```

or

```
for (j=0;j<m;j++)
      # pragma omp parallel for
      for (i=0;i<n;i++)
            a[j][i] = 0.;
```

Nested Loops

Which is better (assuming m ≈ n)?

```
#pragma omp parallel for private(i)
for (j=0;j<m;j++)
      for (i=0;i<n;i++)
            a[j][i] = 0.;
```

or

```
for (j=0;j<m;j++)
      # pragma omp parallel for
      for (i=0;i<n;i++)
            a[j][i] = 0.;
```

- First has less overhead: threads created once instead of m times.
- What about order of indices?

OpenMP Runtime

Runtime Environment

Can set runtime vars (or query from within program) to control:

- OMP_NUM_THREADS - sets number of threads to use. (omp_set_num_threads(pos. integer) at runtime)
- OMP_DYNAMIC true/false - to permit or disallow system to dynamically adjust number of threads used in future parallel regions. (omp_set_dynamic(flag) at runtime)
- OMP_NESTED to find out if parallel nesting allowed (omp_set_nested or omp_get_nested at runtime)
- OMP_SCHEDULE to set default scheduling type for parallel loops of type runtime

Also runtime calls:

```
omp_get_num_threads(),
omp_in_parallel(),
omp_get_thread_num(),
omp_get_num_procs()
```

Number of Threads

The number of threads is determined in order of precedence by:

- Evaluation of **if** clause (if evaluates to zero - false- serial execution)
- Setting the **num_threads** clause in pragma
- the **omp_set_num_threads()** library function
- the **OMP_NUM_THREADS** environment variable
- Implementation default

Threads numbers from 0 (master thread) to N-1.

Performance Issues

Performance Issues

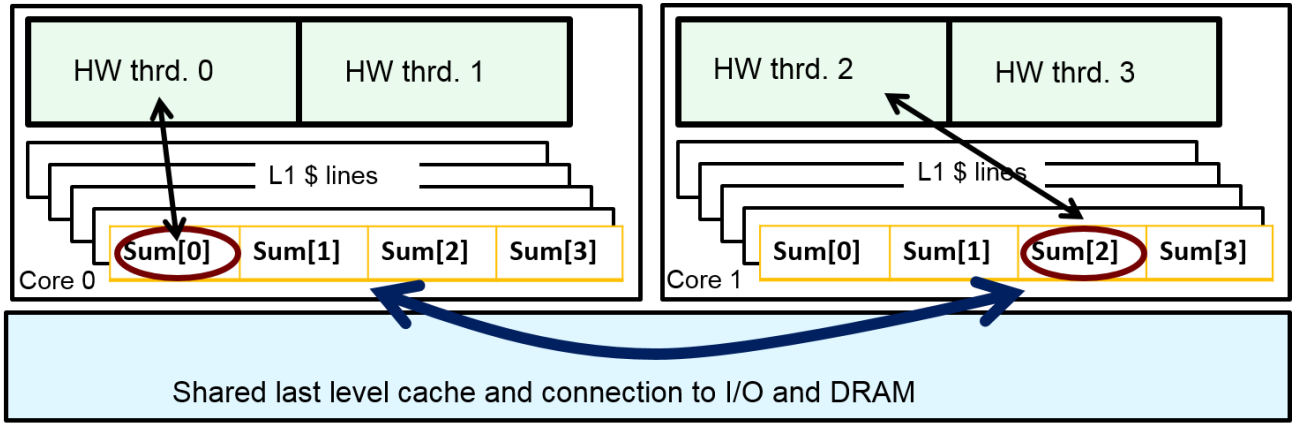
Use profiling to show where program spends most of its time.

- state of a thread: waiting for work, synchronizing, forking, joining, doing useful work.
- Time spent in parallel regions and work-sharing constructs
- time spent in user and system level routines
- hardware counter info: CPU cycles, instructions, cache misses
- time spend in communication, message length, number of messages

False Sharing

False Sharing = when two threads update different data elements in the same cache line.

- Side effect of cache line granularity.
- Can be problem on shared memory machines
- Any time cache line is modified, cache coherence mech. notifies other caches with copies that cache line has been modified elsewhere. Local cache line invalidated, even if different bytes modified. Cache line hops from one cache to the other.
- Solution: Pad arrays so elements you use are on distinct cache lines; eliminate shared arrays

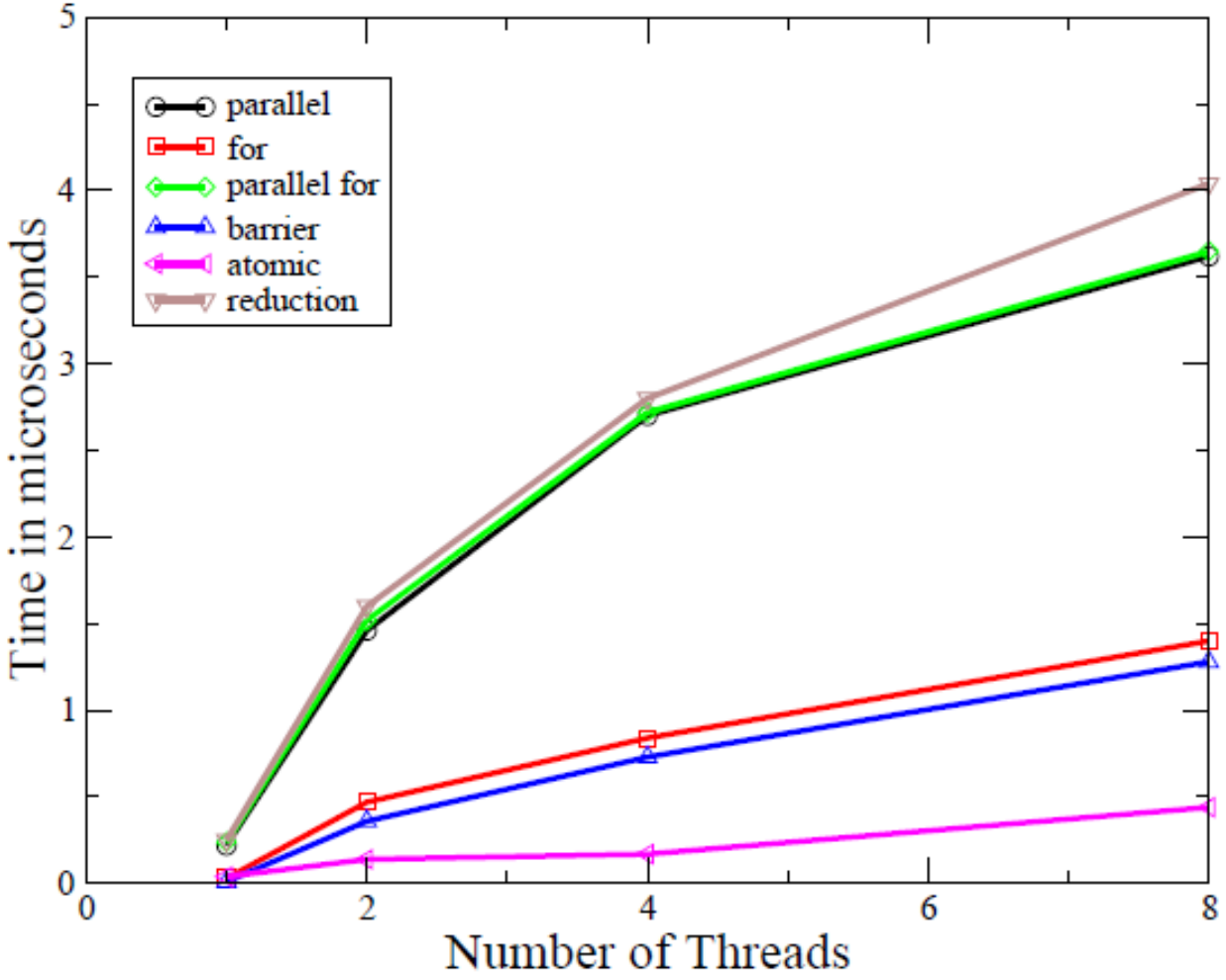


Speedup and Efficiency

- On p processors speedup $S(p) = T_1/T_p$
- Linear (ideal) speedup: on p procs code is p times faster
  - this doesn't usually happen due to overhead, contention, other bottlenecks
  - however can sometimes observe superlinear speedup due to cache effects (smaller problem fits more easily in cache)

OpenMP Overhead

- Results (selected) of running epcc micro-benchmarks on one node of cluster at Cornell (results of David Bindel)



Typical Bugs

*Examples from Using OpenMP, by Chapman, Jost and Van Der Pas

Typical Bugs

Default behavior for parallel variables is shared.

```
void compute(int n){
      int i;
      double h,x,sum;
      h = 1.0/(double)/n;
      sum = 0.0;
      #pragma omp for reduction (+:sum) shared(h)
      for (i=1; i<=n; i++){
            x = h*((double)i - 0.5);
            sum += (1.0)/(1.0+x*x));
      }
      pi = h * sum;
}
```

Race condition due to forgetting to declare x as private.

Typical Bugs

Default for index variables of parallel for loops is private, but not for loops at a deeper nesting level.

```
int i,j;
#pragma omp parallel for
for (i=0;i<n;i++){
      for (j=0;j<m;j++){
            a[i][j] = compute(i,j)
      }
}
```

Loop variable j shared by default – data race. Explicitly declare private

Typical Bugs

Problems with private variables:

```
void main (){
. . .
#pragma omp parallel for private(i,a,b)
for (i=0;i<n;i++){
      b++;
      a = b+i;
}

c = a + b;
```

- Remember that value of a private copy is uninitialized on entry to parallel region (unless use firstprivate(b))
- the value of the original variable is undefined on exit from the parallel region (unless use lastprivate(a,b))

Good habit to use default(none) clause - helps debugging

Typical Bugs

nowait causes problems:

```
#pragma omp parallel
{
      #pragma omp for schedule(static) nowait
      for (i=0;i<n;i++)
            b[i] = (a[i]+a[i-1])/2.0;

      #pragma omp for schedule(static) nowait
      for (i=0;i<n;i++)
            z[i] = sqrt(b[i]);
}
```

Can't assume which thread executes which loop iterations.

Second loop might read values of b not yet written in first loop.

Typical Bugs

Illegal use of barrier:

```
#pragma omp parallel
{
      if (omp_get_thread_num()==0)
      {     ...
            #pragma omp barrier
      }
      else
      {     ...
            #pragma omp barrier
      }
}
```

barrier must be encountered by all threads in a team. The runtime behavior of this is undefined.

Typical Bugs

Missing curly braces:

```
#pragma omp parallel
{
      work1(); /* executed in parallel */
      work2(); /* executed in parallel */
}
#pragma omp parallel
      work1(); /* executed in parallel */
      work2(); /* executed sequentially */
```

Need curly brackets for parallel region more than a single statement.

Typical Bugs

How many times is the alphabet printed in each block?

```
int i;
#pragma omp parallel for
      for (i='a'; i<= 'z'; i++)
            printf ("%c",i);

int i;
#pragma omp parallel
      for (i='a'; i<='z';i++)
            printf("%c",i);
```

Typical Bugs

```
int i;
#pragma omp parallel for
      for (i='a'; i<= 'z'; i++)
            printf ("%c",i);
```

```
v   thread 3
a   thread 0
h   thread 1
o   thread 2
w   thread 3
b   thread 0
i   thread 1
p   thread 2
x   thread 3
c   thread 0
j   thread 1
q   thread 2
y   thread 3
d   thread 0
k   thread 1
r   thread 2
z   thread 3
e   thread 0
l   thread 1
s   thread 2
f   thread 0
m   thread 1
t   thread 2
g   thread 0
n   thread 1
u   thread 2
```

Typical Bugs

```
int i;
#pragma omp parallel
      for (i='a'; i<='z';i++)
            printf("%c",i);
```

```
a   thread 0
a   thread 2
a   thread 1
a   thread 3
b   thread 0
c   thread 2
d   thread 1
e   thread 3
f   thread 0
g   thread 2
h   thread 1
i   thread 3
j   thread 0
k   thread 2
l   thread 1
m   thread 3
n   thread 0
o   thread 2
p   thread 1
q   thread 3
r   thread 0
s   thread 2
t   thread 1
u   thread 3
v   thread 0
w   thread 2
x   thread 1
y   thread 3
z   thread 0
```

Resources

- http://computing.llnl.gov/tutorials/openMP/
  - very complete description of OpenMP for Fortran and C
- http://www.openmp.org
- https://www.openmp.org/about/openmp-faq/