Week 2
Parallel Computing Models

Race Condition or Data Dependence

- A ***race condition*** exists when the result of an execution depends on the ***timing*** of two or more events.
- A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness. (More on data dependences in a subsequent lecture.)
- ***Synchronization*** is used to sequence control among threads or to sequence accesses to data in parallel code.

Simple Example (p. 4 of text)

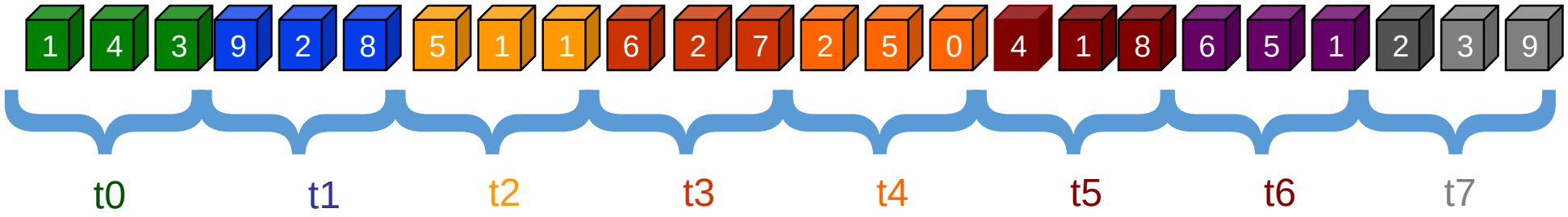
- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

- Parallel formulation?

Version 1: Computation Partitioning

- Suppose each core computes a partial sum of n/t consecutive elements (t is the number of threads or processors)
- Example: n = 24 and t = 8, threads are numbered from 0 to 7




```
int block_length_per_thread = n/t;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
    x = Compute_next_value(…);
    sum += x;
}
```

What Happened?

- Dependence on sum across iterations/threads
  - But reordering ok since operations on sum are associative
- Load/increment/store must be done *atomically* to preserve sequential meaning
- Definitions:
  - Atomicity: a set of operations is atomic if either they all execute or none executes. Thus, there is no way to see the results of a partial execution.
  - Mutual exclusion: at most one thread can execute the code at any time

Version 2: Add Locks

- Insert mutual exclusion (mutex) so that only one thread at a time is loading/incrementing/storing count atomically

```
int block_length_per_thread = n/t;
mutex m;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
    my_x = Compute_next_value(…);
    mutex_lock(m);
    sum += my_x;
    mutex_unlock(m);
}
```

Correct now. Done?

Version 3: Increase Granularity

- Version 3:
  - Lock only to update final sum from private copy

```
int block_length_per_thread = n/t;
mutex m;
int my_sum;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     my_sum += my_x;
}
mutex_lock(m);
sum += my_sum;
mutex_unlock(m);
```

Version 4: Eliminate lock

- Version 4 (bottom of page 4 in textbook):
  - "Master" processor accumulates result

```
int block_length_per_thread = n/t;
mutex m;
shared my_sum[t];
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     my_sum[id] += my_x;
}
if (id == 0) { // master thread
   sum = my_sum[0];
   for (i=1; i<t; i++) sum += my_sum[i];
}
```

Correct? Why not?

More Synchronization: Barriers

- Incorrect if master thread begins accumulating final result before other threads are done
- How can we force the master to wait until the threads are ready?
- Definition:
  - A ***barrier*** is used to block threads from proceeding beyond a program point until all of the participating threads has reached the barrier.
  - Implementation of barriers?

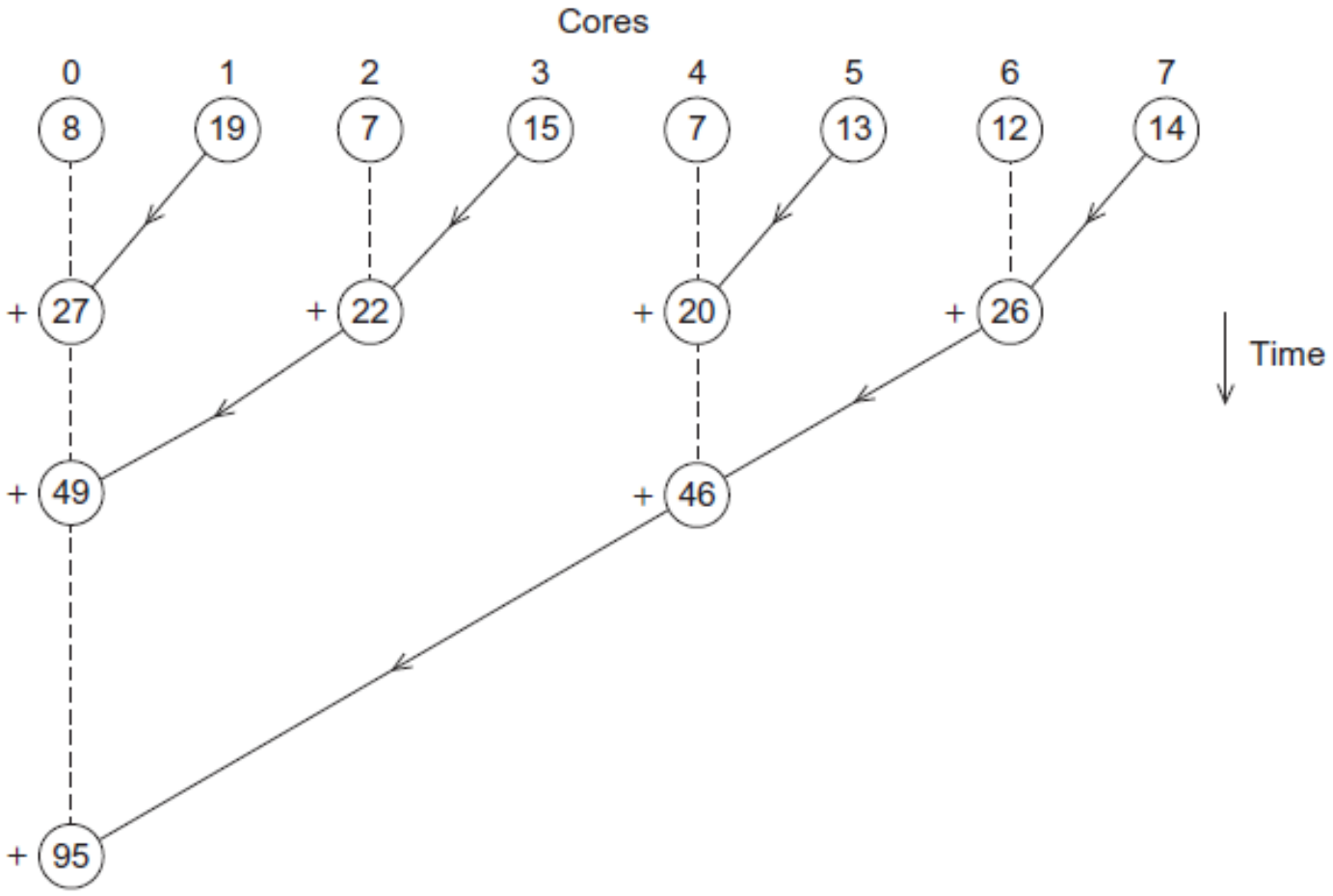
Version 5: Eliminate lock, but add barrier

- Version 5 (bottom of page 4 in textbook):
  - "Master" processor accumulates result

```
int block_length_per_thread = n/t;
mutex m;
shared my_sum[t];
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
     my_x = Compute_next_value(…);
     my_sum[t] += x;
}
Synchronize_cores(); // barrier for all participating threads
if (id == 0) { // master thread
   sum = my_sum[0];
   for (i=1; i<t; i++) sum += my_sum[t];
}
```

Now it's correct!

Version 6 (homework): Multiple cores forming a global sum



Parallel Programming Models

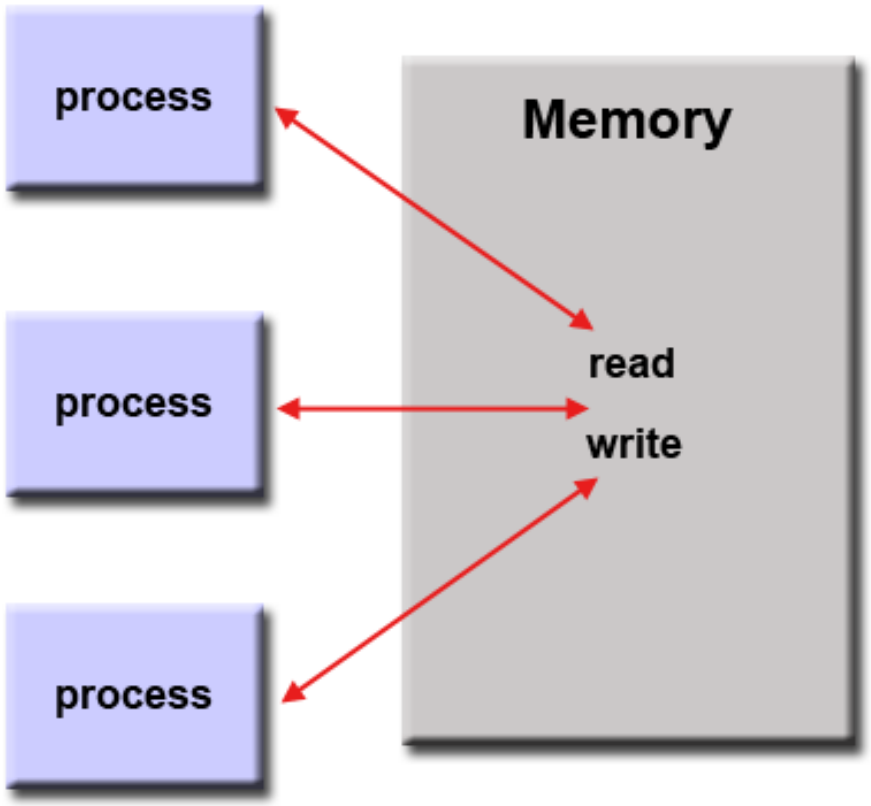
Parallel Programming Models

- Programming models are abstractions
  - Can be used on various architectures

- Shared Memory Model
  - Thread-based Model
  - Process-based Model
  - Global Address Space
- Message Passing Model
- Data Parallel Model

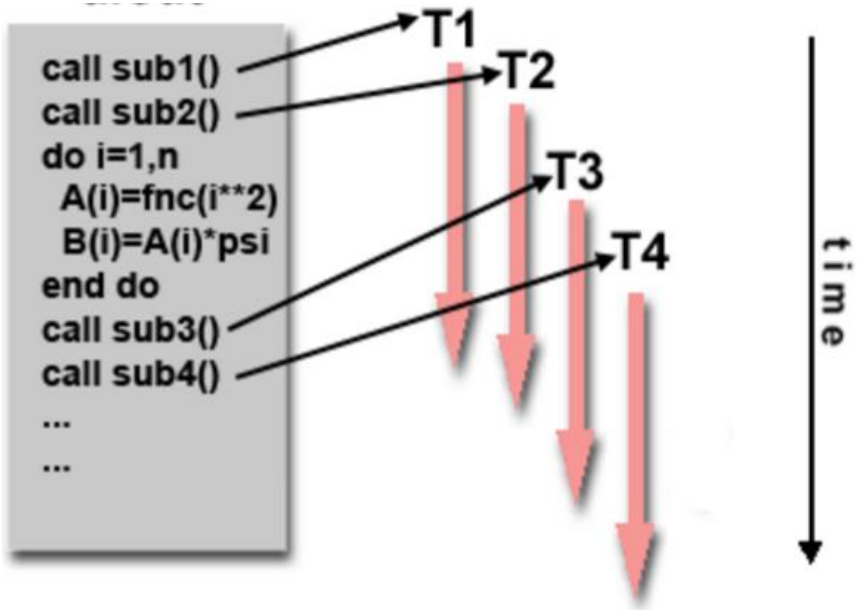
Shared Memory Model

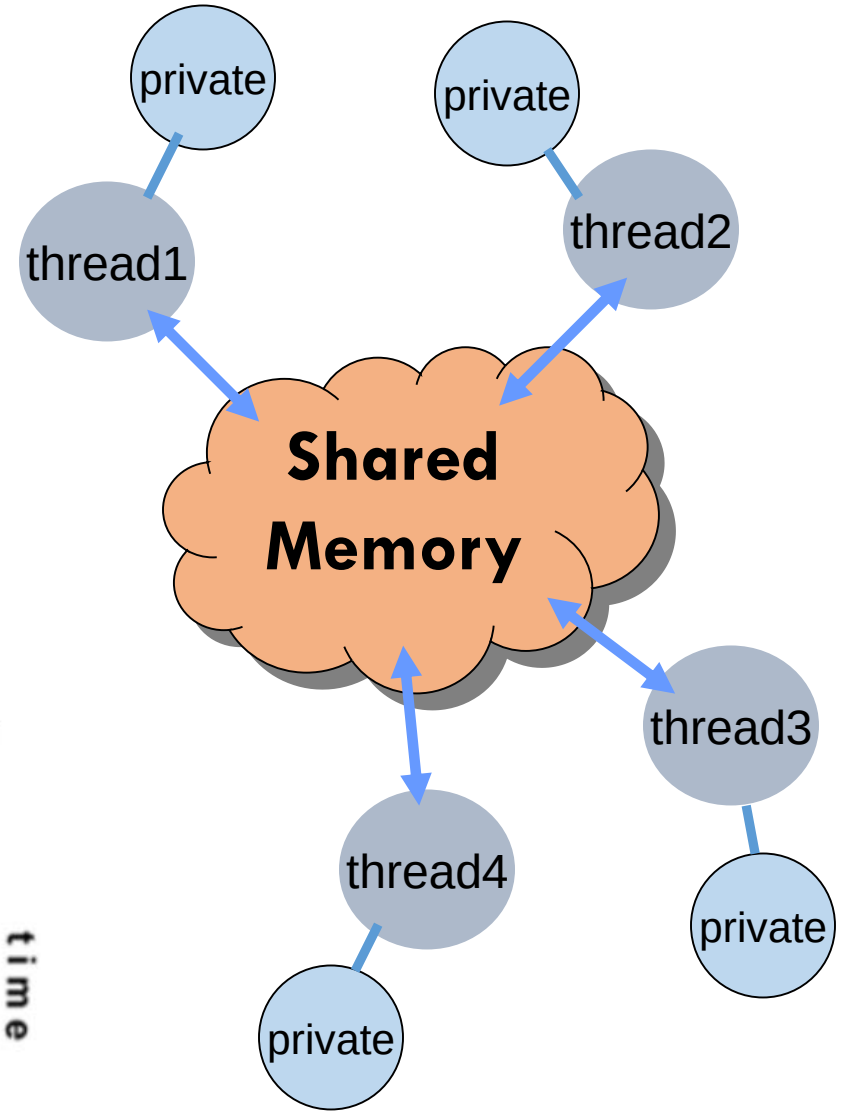
- A shared address space between tasks
- Asynchronous read/write
- Separate mechanisms for synchronization
  - Locks, semaphores, flags
- No explicit “communication” between processes
  - Nobody owns the data
- Can be used over SMP, and NUMA systems
- Emulated over distributed memory systems (e.g., Numascale, ScaleMP)



Thread-based Model

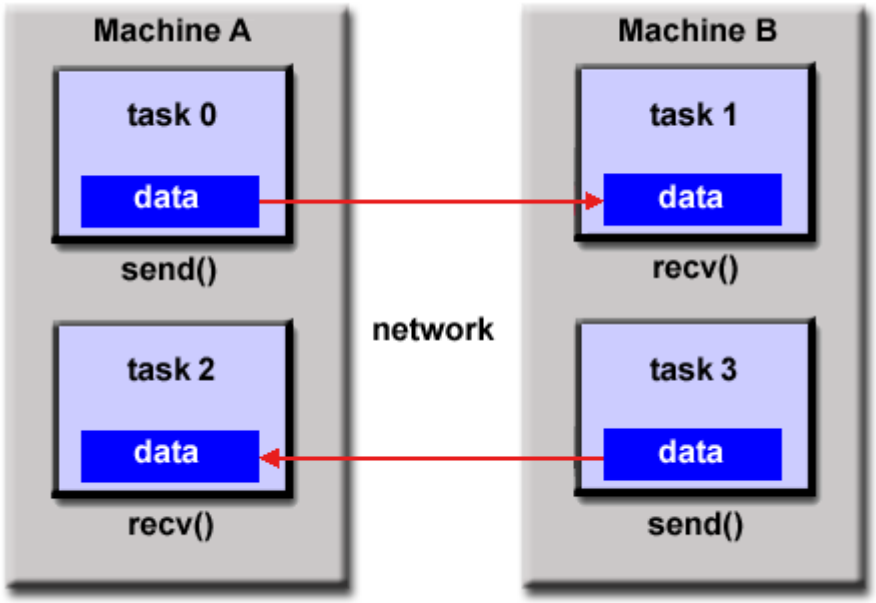
- Single heavy-weight process is divided into multiple threads
  - All share the original address space
- Subroutine/library and compiler directives
- POSIX Threads (PThreads)
- OpenMP





Message Passing Model

- Multiple processes with separate memory spaces
- May reside on separate node across an "interconnection network"
- Data communication is through messages
  - Sent from one process to another in the group
- Synchronization is usually implicit
  - Using communication-assisted synchronization (e.g., barrier)
  - As part of communication (e.g., Collectives, sendrecive)
- Message Passing Interface (MPI)



Message Passing Interface (MPI)

What is MPI?

- A *message-passing library standard*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured, 3 standard versions (currently version 3)
- Designed to provide access to parallel hardware for
  - end users
  - library writers
  - tool developers

A Minimal MPI Program (C)

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

A Minimal MPI Program (Fortran)

```
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Running MPI Programs

- The Standard does not specify how to run an MPI program.
- In general, starting an MPI program is **dependent on the implementation of MPI you are using**, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** or **`mpirun <args>`** is part of MPI-2 and MPI-3, as a recommendation, but not a requirement

```
$mpirun –host compute-0-0,compute-0-1 –n 32 ./calculate_pi 1500
```

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

Better Hello (C)

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Better Hello (Fortran)

```
program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Some Basic Concepts

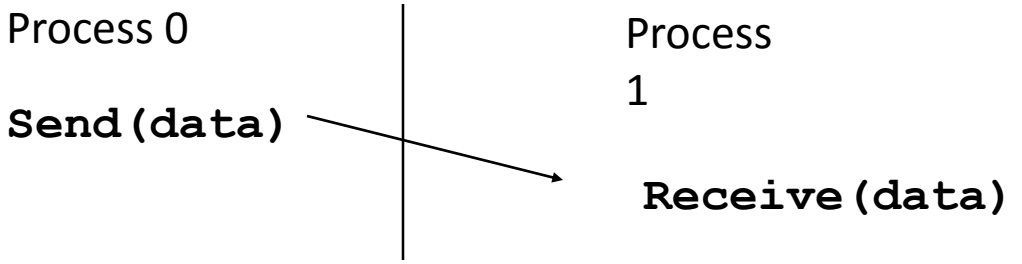
- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored column-wise.

MPI Basic Send/Receive

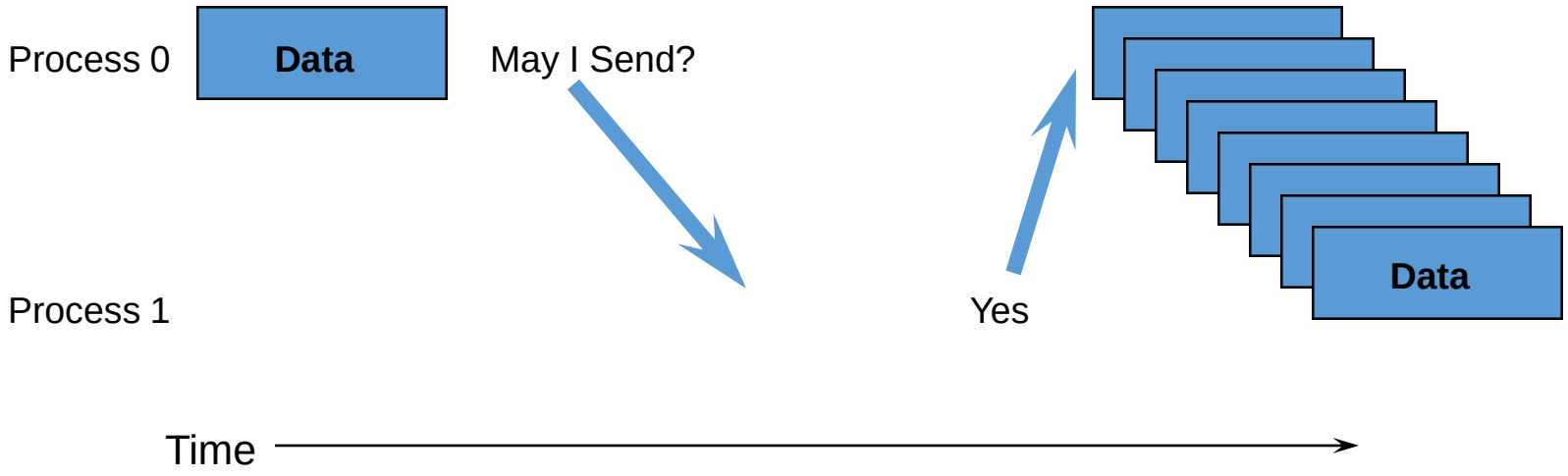
- We need to fill in the details in



- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

```
MPI_SEND (start, count, datatype, dest,
  tag, comm)
```

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

```
MPI_RECV(start, count, datatype, source, tag,
  comm, status)
```

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`status`** contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Simple Fortran Example - 1

```
      program main
      use MPI

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(10)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src  = 0
```

Simple Fortran Example - 2

```
      if (rank .eq. 0) then
        do 10, i=1, 10
          data(i) = i
10      continue

        call MPI_SEND( data, 10, MPI_DOUBLE_PRECISION,
     +                 dest, 2001, MPI_COMM_WORLD, ierr)
      else if (rank .eq. dest) then
        tag = MPI_ANY_TAG
        source = MPI_ANY_SOURCE
        call MPI_RECV( data, 10, MPI_DOUBLE_PRECISION,
     +                 source, tag, MPI_COMM_WORLD,
     +                 status, ierr)
```

Simple Fortran Example - 3

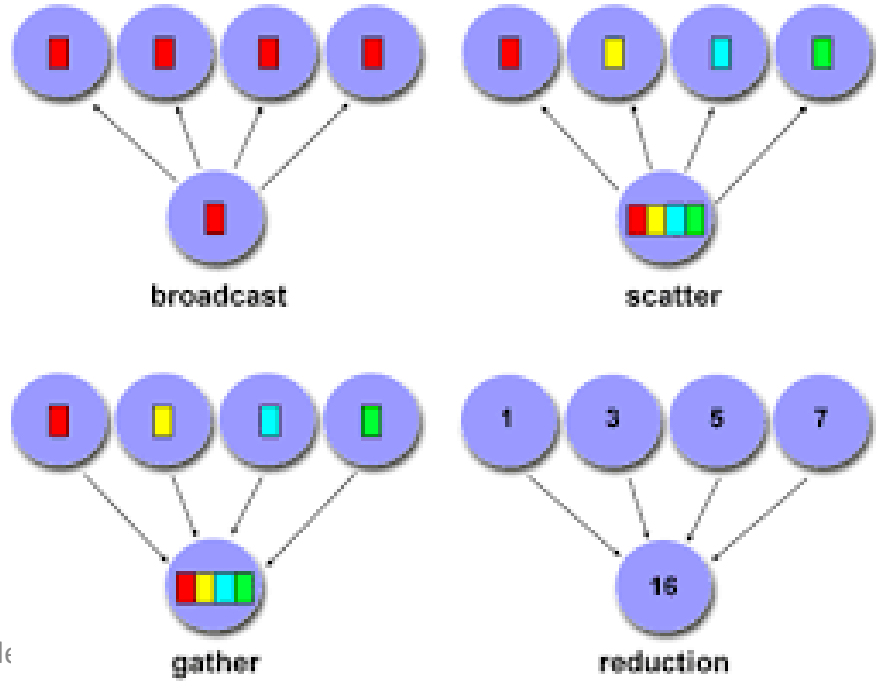
```
      call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
                          st_count, ierr )
      st_source = status( MPI_SOURCE )
      st_tag    = status( MPI_TAG )
      print *, 'status info: source = ', st_source,
     +           ' tag = ', st_tag, 'count = ', st_count
      endif

      call MPI_FINALIZE( ierr )
      end
```

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn't the only way...

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.



Example: PI (π) in C -1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)  {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
```

Example: PI (π) in C - 2

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();

  return 0;

}
```

Alternative set of 6 Functions for Simplified MPI

- **`MPI_INIT`**
- **`MPI_FINALIZE`**
- **`MPI_COMM_SIZE`**
- **`MPI_COMM_RANK`**
- **`MPI_BCAST`**
- **`MPI_REDUCE`**

- What else is needed (and why)?

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This is called “unsafe” because it depends on the availability of system buffers

Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	**Isend(0)**
Irecv(1)	**Irecv(0)**
Waitall	**Waitall**

Toward a Portable MPI Environment

- In a wide variety of environments, one can do:

```
mpicc myprog.c –o myprog
mpirun –hostfile ./machines.list -np 10 myprog
```

to build, compile, run, and analyze performance.

Extending the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/ Fortran-90; interlanguage issues
- MPI-3 features
  - Non-blocking and topological collectives

When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

When *not* to use MPI

- Regular computation matches HPF
  - But see PETSc/HPF comparison (ICASE 97-72)
- Solution (e.g., library) already exists
  - http://www.mcs.anl.gov/mpi/libraries.html
- Require Fault Tolerance
  - Sockets
- Distributed Computing
  - CORBA, DCOM, etc.

OpenMP Standard/Library

OpenMP: Some syntax details to get us started

- Used for parallel programming in a shared-memory space
- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

    #pragma omp *construct [clause [clause]…]*
  - For Fortran, the directives take one of the forms:

    C$OMP *construct [clause [clause]…]*

    !$OMP *construct [clause [clause]…]*

    *$OMP *construct [clause [clause]…]*
- Include files

  #include “omp.h”

How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Sequential Program

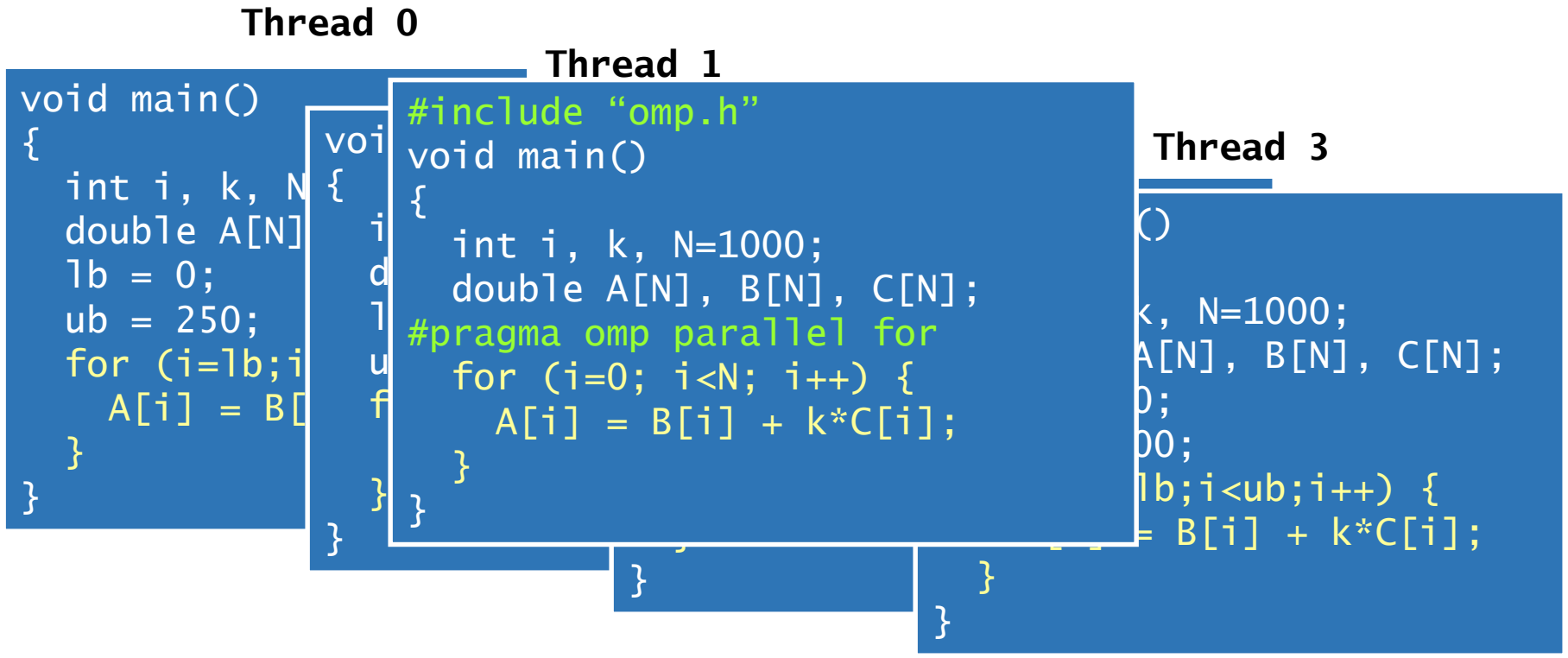
```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

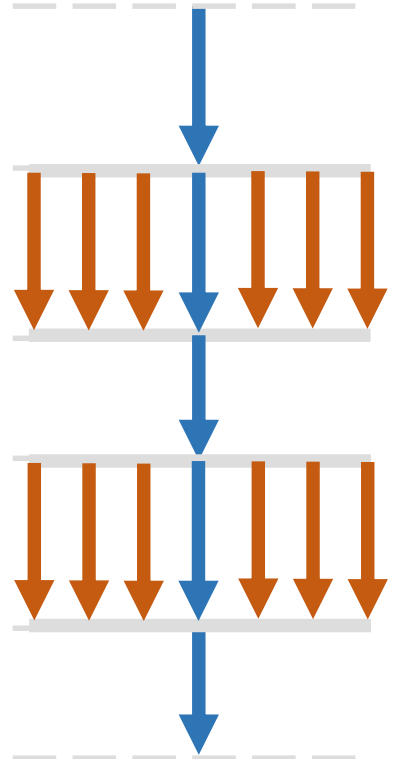
How is OpenMP typically used?

$gcc ./my_omp_loop.c -o ./my_omp_loop -fopenmp



OpenMP Fork-and-Join model

```
printf("program begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("program done\n");
```



OpenMP Constructs

- Parallel Regions
- Worksharing (for/DO, sections, …)
- Data Environment (shared, private, …)
- Synchronization (barrier, flush, …)
- Critical sections (critical)
- Runtime functions/environment variables (omp_get_num_threads(), …)

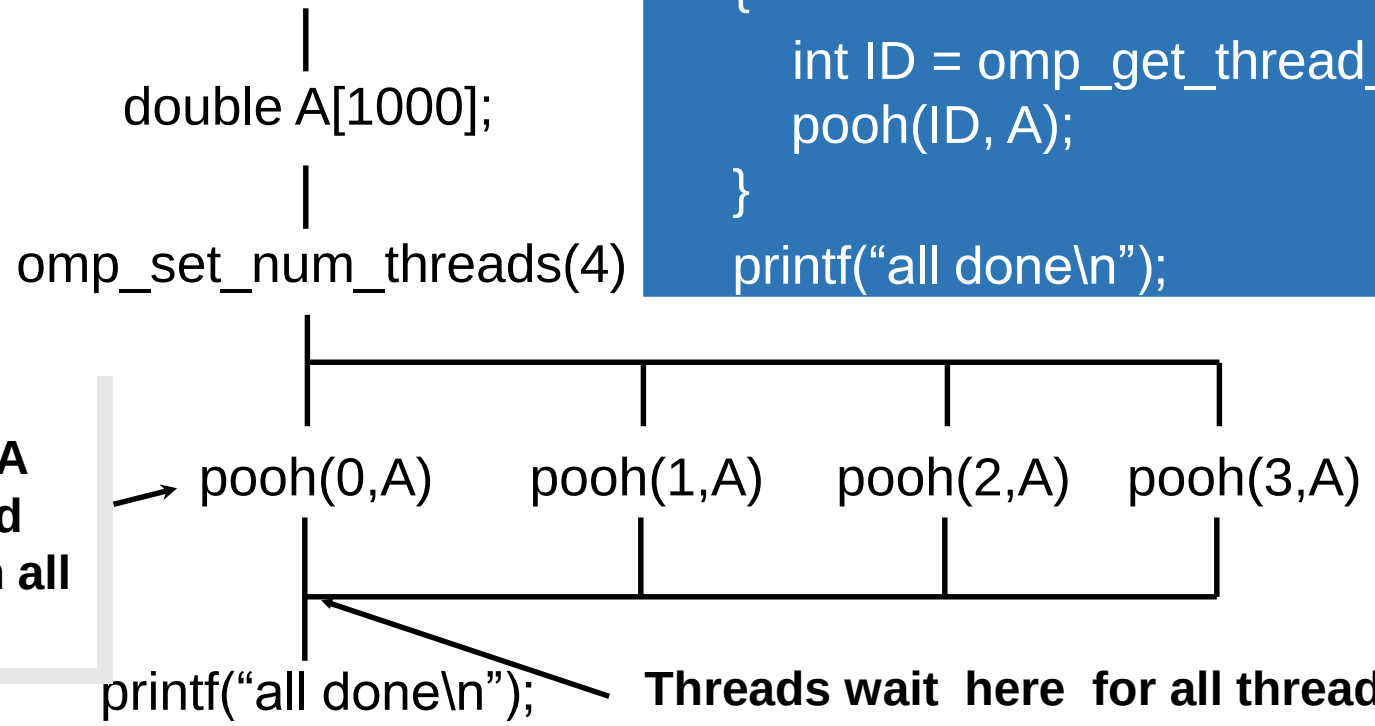
Data Environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

OpenMP Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



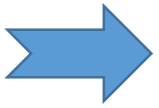
A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (I.e. a *barrier*)

The OpenMP API
Combined parallel work-share

- OpenMP shortcut: Put the “parallel” and the work-share on the same line

```
int i;
double  res[MAX];
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
}
```



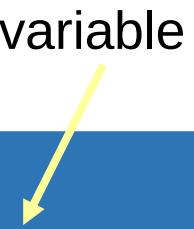
```
int i;
double  res[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
  res[i] = huge();
}
```

Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
    lsum = 0;
    #pragma omp for
    for (i=0; i<N; i++) {
        lsum = lsum + A[i];
    }
    #pragma omp critical
    { sum += lsum; }
}
```


Threads wait their turn;
only one thread at a time
executes the critical section

Reduction Clause




```
sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++)
{
   sum = sum + A[i];
}
```