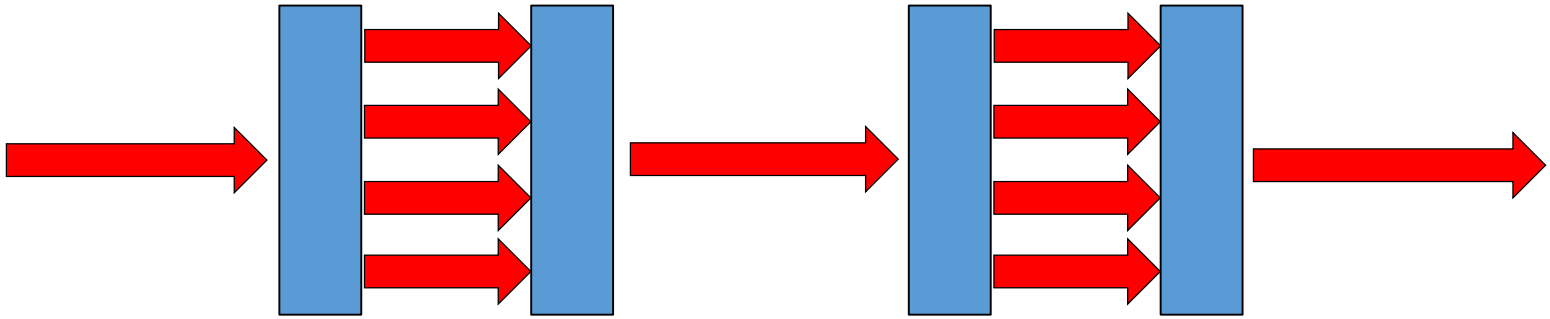
Introduction to Parallel Programming with MPI and OpenMP



Charles Augustine
October 29, 2018

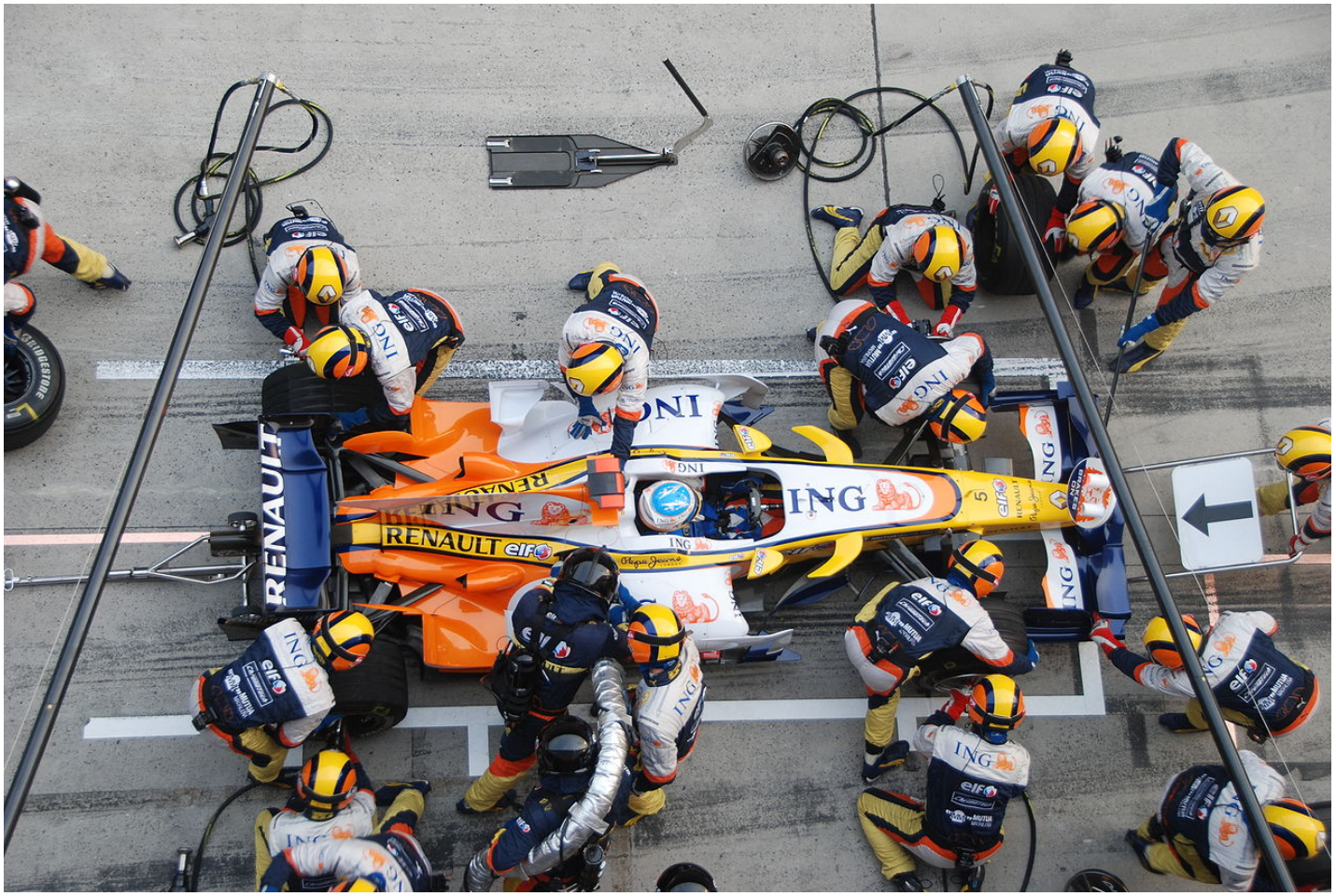
OFFICE of
INFORMATION TECHNOLOGY

PICSciE

Goals of Workshop

- Have basic understanding of
  - Parallel programming
  - MPI
  - OpenMP
- Run a few examples of C/C++ code on Princeton HPC systems.
- Be aware of some of the common problems and pitfalls
- Be knowledgeable enough to learn more (advanced topics) on your own

Parallel Programming Analogy



Source: Wikapedia.org

Disadvantages/Issues

- No free lunch - can't just "turn on" parallel
- Parallel programming requires work
  - Code modification – always
  - Algorithm modification – often
  - New sneaky bugs – you bet
- Speedup limited by many factors

Realistic Expectations

- Ex. – Your program takes 20 days to run
- 95% can be parallelized
- 5% cannot (serial)
- What is the fastest this code can run?
  - As many CPU's as you want!

1 day!

Amdahl's Law

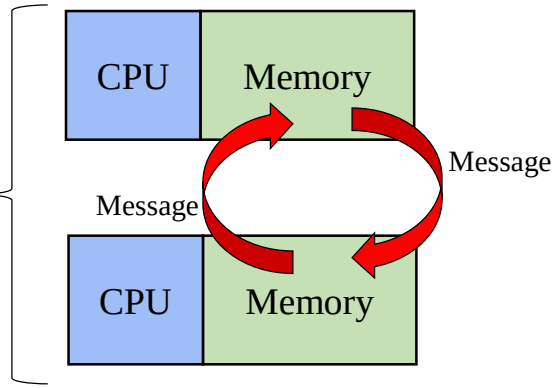
Computer Architecture

- As you consider parallel programming understanding the underlying architecture is important
- Performance is affected by hardware configuration
  - Memory or CPU architecture
  - Numbers of cores/processor
  - Network speed and architecture

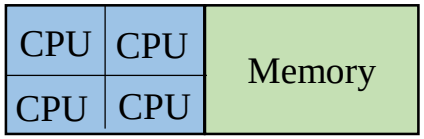
MPI and OpenMP

- MPI – Designed for distributed memory
  - Multiple systems
  - Send/receive messages
- OpenMP – Designed for shared memory
  - Single system with multiple cores
  - One thread/core sharing memory
- C, C++, and Fortran
- There are other options
  - Interpreted languages with multithreading
    - Python, R, matlab (have OpenMP & MPI underneath)
  - CUDA, OpenACC (GPUs)
  - Pthreads, Intel Cilk Plus (multithreading)
  - OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)

MPI



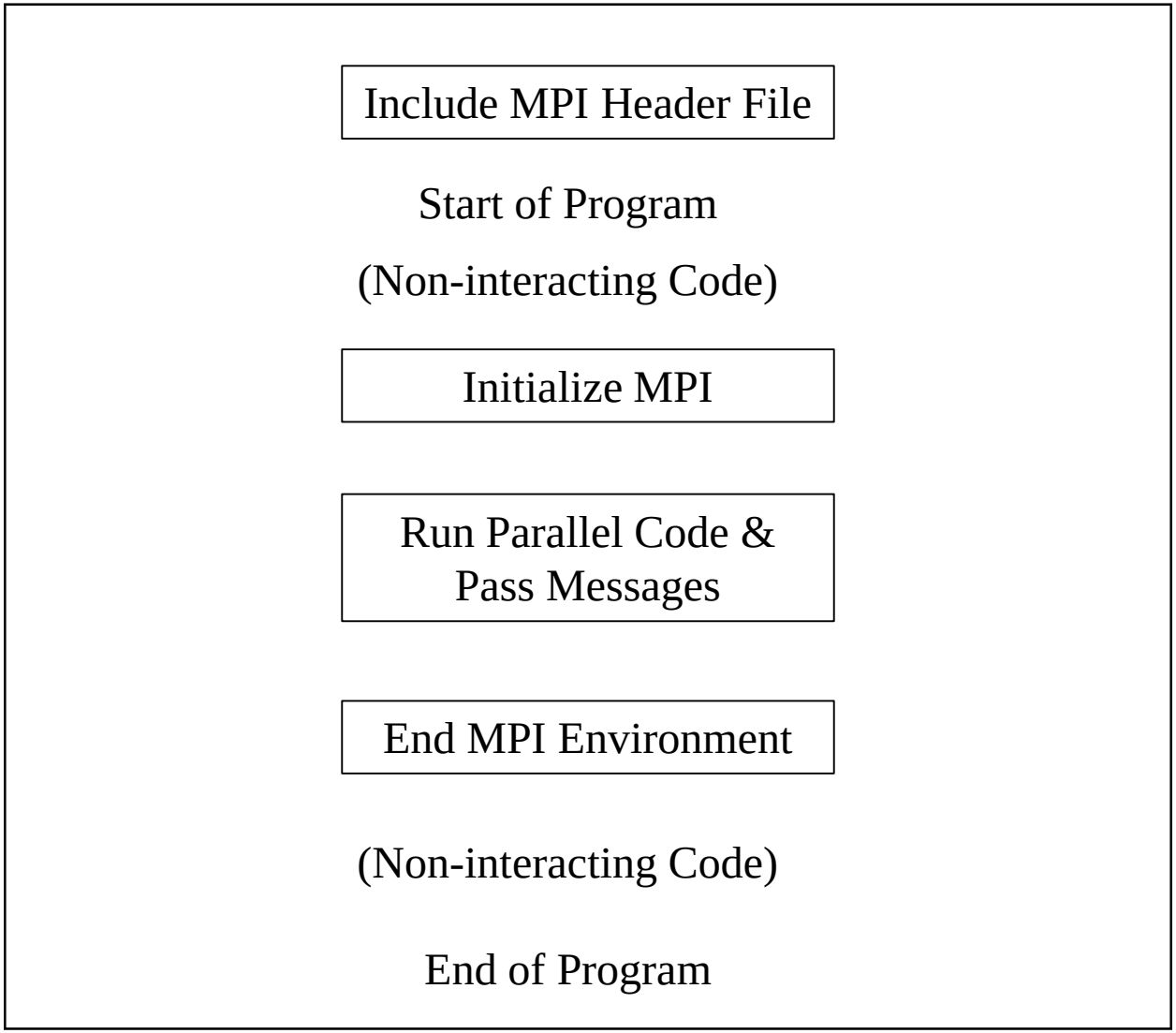
OpenMP



MPI

- Message Passing Interface
- Standard
  - MPI-1 – Covered here
  - MPI-2 – Added features
  - MPI-3 – Even more cutting edge
- Distributed Memory
  - But can work on shared
- Multiple implementations exist
  - Open MPI
  - MPICH
  - Many commercial (Intel, HP, etc..)
  - Difference should only be in the compilation not development
- C,C++, and Fortran

MPI Program - Basics



MPI Program Basics

Include MPI Header File

Start of Program

(Non-interacting Code)

Initialize MPI

Run Parallel Code &
Pass Messages

End MPI Environment

(Non-interacting Code)

End of Program

```
#include <mpi.h>

int main (int argc, char *argv[])
{

MPI_Init(&argc, &argv);

.
.     // Run parallel code
.

MPI_Finalize(); // End MPI Envir

return 0;
}
```

Basic Environment

```
MPI_Init(&argc, &argv)
```

- Initializes MPI environment
- Must be called in every MPI program
- Must be first MPI call
- Can be used to pass command line arguments to all

```
MPI_Finalize()
```

- Terminates MPI environment
- Last MPI function call

Communicators & Rank

- MPI uses objects called communicators
  - Defines which processes can talk
  - Communicators have a size
- MPI_COMM_WORLD
  - Predefined as ALL of the MPI Processes
  - $Size = N_{procs}$
- Rank
  - Integer process identifier
  - $0 \leq Rank < Size$

Basic Environment Cont.

```
MPI_Comm_rank(comm, &rank)
```

- Returns the rank of the calling MPI process
- Within the communicator, comm
  - MPI_COMM_WORLD is set during Init(…)
  - Other communicators can be created if needed

```
MPI_Comm_size(comm, &size)
```

- Returns the total number of processes
- Within the communicator, comm

```
int my_rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Hello World for MPI

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


int main (int argc, char *argv[]) {

  int rank, size;

  MPI_Init (&argc, &argv);  //initialize MPI library

  MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my process id

  //do something
  printf ("Hello World from rank %d\n", rank);
  if (rank == 0) printf("MPI World size = %d processes\n", size);

  MPI_Finalize(); //MPI cleanup

  return 0;
}
```

Hello World Output

- 4 processes

```
Hello World from rank 3
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 2
Hello World from rank 1
```

- Code ran on each process independently
- MPI Processes have *private* variables
- Processes can be on completely different machines

How to Compile @ Princeton

- Intel (icc) and GNU (gcc) compilers
  - Which to use?
  - gcc free and available everywhere
  - Often icc is faster
  - This workshop uses icc.
- MPI compiler wrapper scripts are used
  - Loaded through module command
  - Different script for each language (C, C++, Fortan)

Compile & Run Code

```
[user@adroit4]$ module load openmpi/intel-17.0 intel/17.0

[user@adroit4]$ mpicc hello_world_mpi.c -o hello_world_mpi
[user@adroit4]$ mpirun –np 1 ./hello_world_mpi
Hello World from rank 0
MPI World size = 1 processes
```

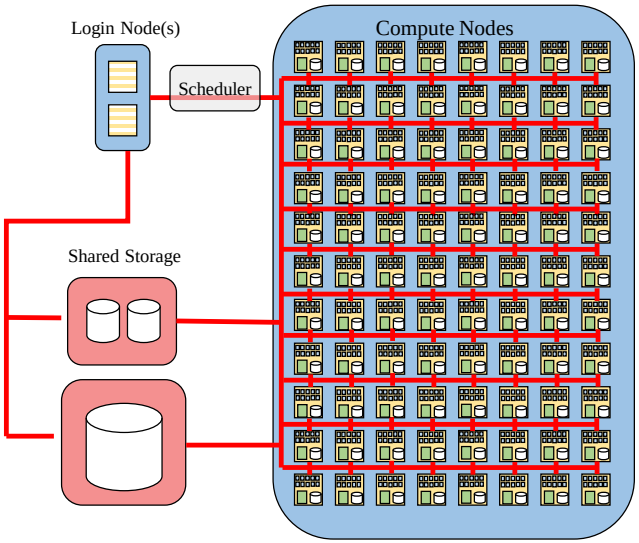
Only needed once in a session.

Language	Script Name
C	mpicc
C++	mpic++, mpiCC, mpicxx
Fortran	mpif77, mpif90

Use the --showme flag to see details of wrapper

Testing on head node

- For head/login node testing
- NOT for long running or big tests
  - Small (<8 procs) and short (<2 min)



Start an mpi job

With this number of processes

Run this executable

```
[user@adroit4]$ mpirun -np 4 ./hello_world_mpi
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 1
Hello World from rank 2
Hello World from rank 3
```

Submitting to the Scheduler

- Run on a compute node – essentially a different computer(s)
- Scheduler: SLURM
  - Tell SLURM what resources you need and for how long
  - Then tell it what to do
  - srun = run an MPI job on a SLURM cluster
    - It will call mpirun –np <n> but with better performance

```
#!/bin/bash
#SBATCH --ntasks 4          #4 mpi tasks
#SBATCH -t 00:05:00         #Time in HH:MM:SS

#set up environment
module load openmpi/intel-17.0 intel/17.0

#Launch job with srun not mpirun/mpiexec!
srun ./hello_world.out
```

Make sure environment is the same as what you compiled with!

Submitting to the scheduler

- sbatch command
  - Sends submit script to scheduler
  - Job will run when resources are available
  - Many options (and defaults): see man page (`man sbatch`)

```
[user@adroit4]$ ls
hello_world_mpi.c  hello_world_mpi  submit.slurm
[user@adroit4]$ sbatch submit.slurm
Submitted batch job 62916
[user@adroit4]$ ls
hello_world_mpi.c  hello_world_mpi  slurm-62916.out
submit.slurm
[user@adroit4]$ cat slurm-62916.out
Hello World from rank 1
Hello World from rank 2
Hello World from rank 3
Hello World from rank 0
MPI World size = 4 processes
```

Lab 1: Run Hello World Program

- Workshop materials are here

http://tigress-web.princeton.edu/~augustin/bootcamp_2018.tgz

- ssh to YourNetId@adroit.princeton.edu

```
[user@adroit4]$ wget http://tigress-web/~augustin/bootcamp_2018.tgz
[user@adroit4]$ tar –xvf bootcamp_2018.tgz
```

- Run on head node

```
[user@adroit4]$cd bootcamp
[user@adroit4 bootcamp]$ module load openmpi/intel-17.0  intel/17.0
[user@adroit4 bootcamp]$ mpicc hello_world_mpi.c –o hello_world_mpi
[user@adroit4 bootcamp]$ mpirun –np 6 hello_world_mpic
```

- Submit a job to the scheduler – look at output

```
[user@adroit4 bootcamp]$ sbatch hello_world_mpi.slurm
[user@adroit4 bootcamp]$ cat slurm-xxxxx.out
```

Some Useful SLURM Commands

Command	Purpose/Function
sbatch <filename>	Submit the job in <filename> to slurm
scancel <slurm jobid>	Cancel running or queued job
squeue –u <username>	Show username's jobs in the queue
salloc <resources req'd>	Launch an *interactive* job on a compute node(s)

Point-to-Point Communication

```
MPI_Send(&buf, count, datatype, dest, tag, comm)
```

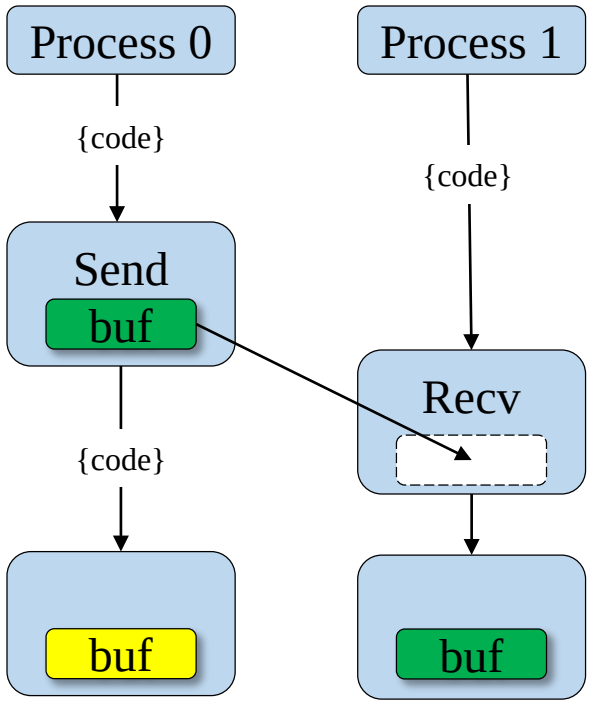
- Send a message
- Returns only after buffer is free for reuse (Blocking)

```
MPI_Recv(&buf, count, datatype, source, tag, comm, &status)
```

- Receive a message
- Returns only when the data is available
  - Blocking

```
MPI_SendRecv(...)
```

- Two way communication
- Blocking

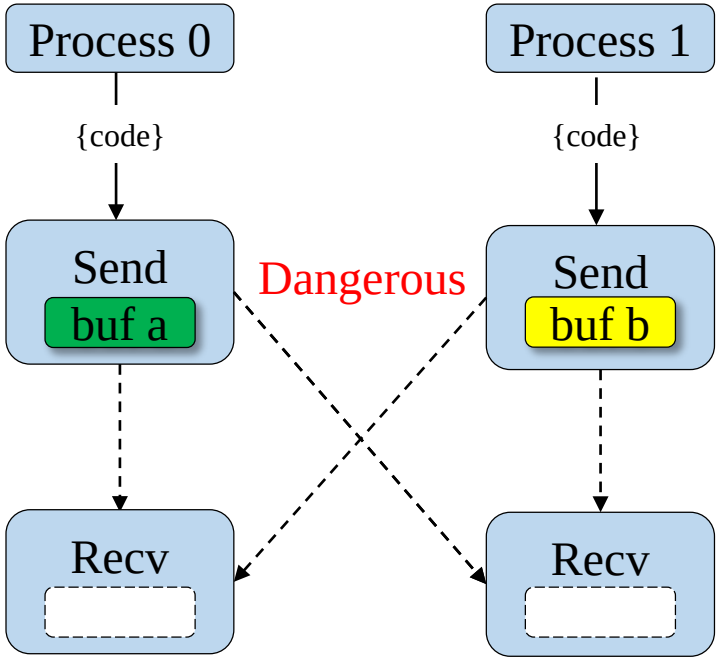


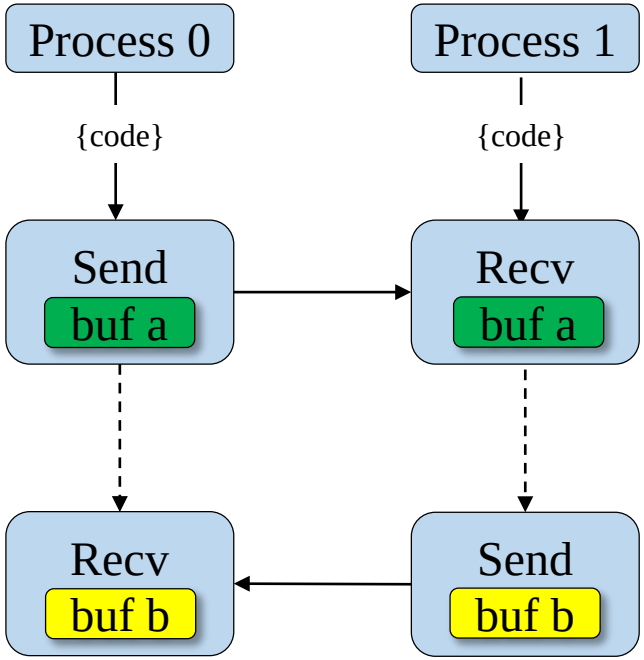
Point-to-Point Communication

- Blocking
  - Only returns after completed
    - Receive: data has arrived and ready to use
    - Send: safe to reuse sent buffer
  - Be aware of deadlocks
  - Tip: Use when possible
- Non-Blocking
  - Returns immediately
    - Unsafe to modify buffers until operation is known to be complete
  - Allows computation and communication to overlap
  - Tip: Use only when needed

Deadlock

- Blocking calls can result in deadlock
  - One process is waiting for a message that will never arrive
  - Only option is to abort the interrupt/kill the code (ctrl-c)
  - Might not always deadlock - depends on size of system buffer





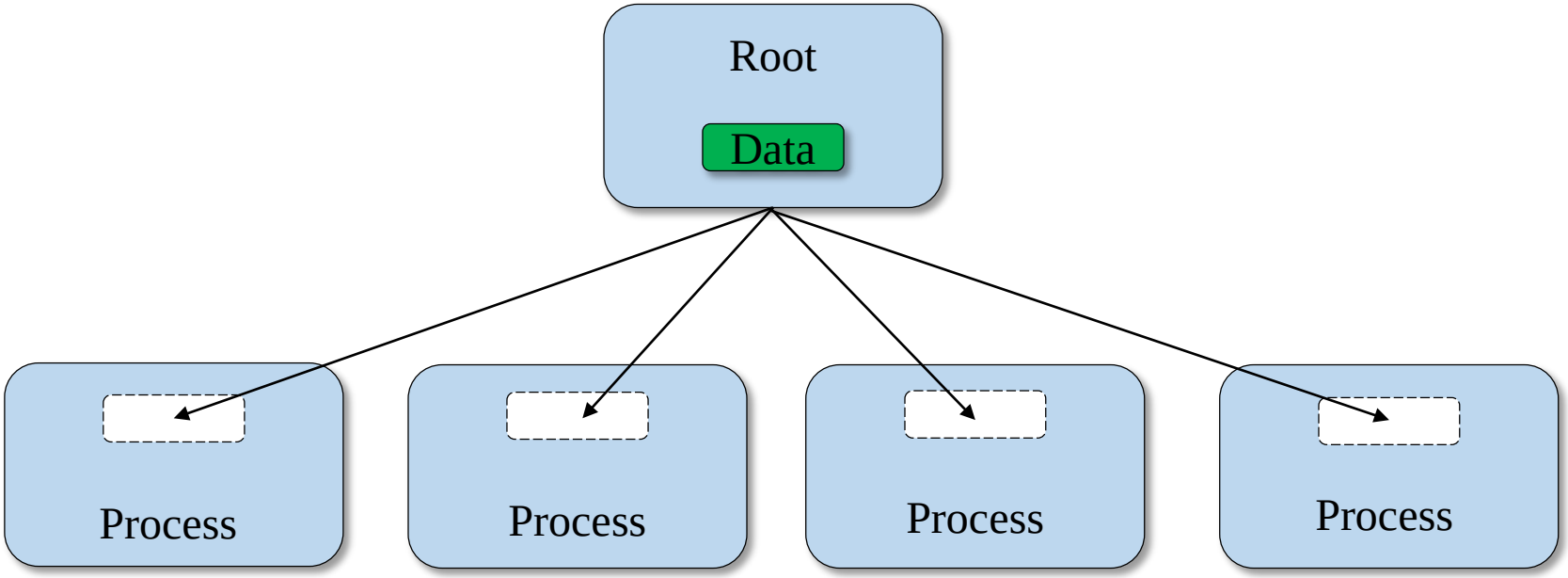
Collective Communication

- Communication between 2 or more processes
  - 1-to-many, many-to-1, many-many
- All processes call the same function with same arguments
- Data sizes must match
- Routines are blocking (MPI-1)

Collective Communication (Bcast)

```
MPI_Bcast(&buffer, count, datatype, root, comm)
```

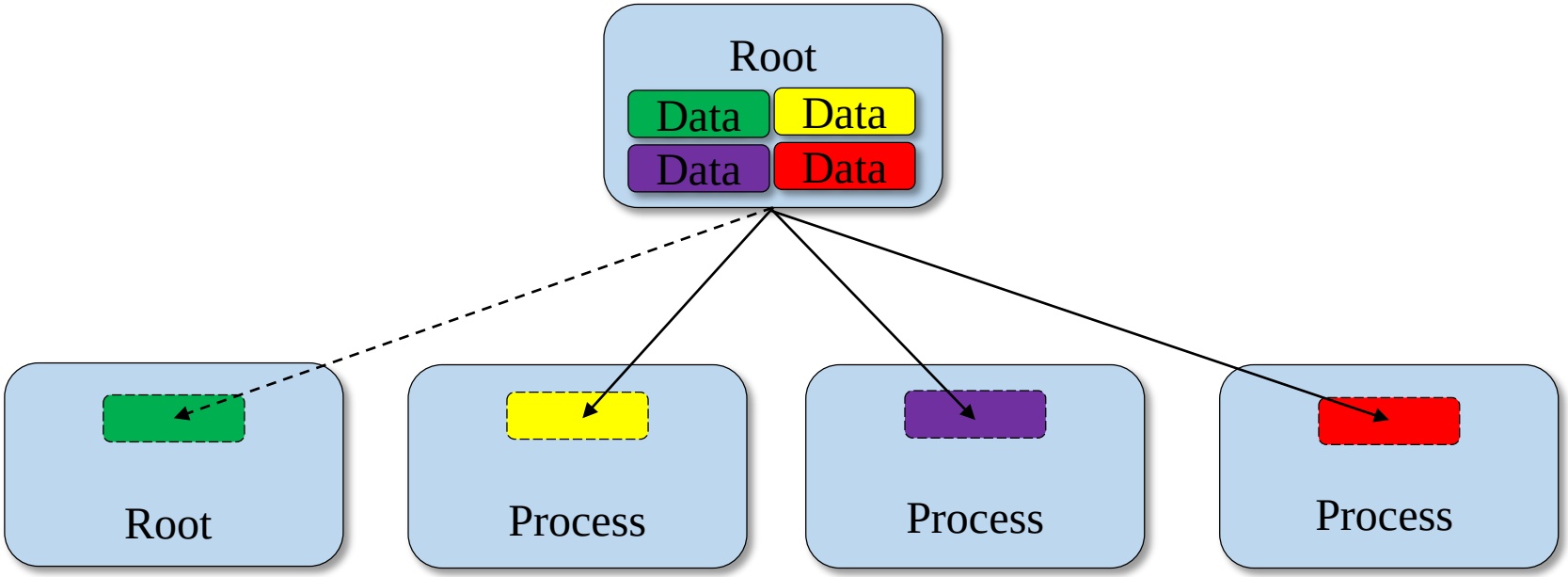
- Broadcasts a message from the root process to all other processes
- Useful when reading in input parameters from file



Collective Communication (Scatter)

```
MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf,
            recvcnt, recvtype, root, comm)
```

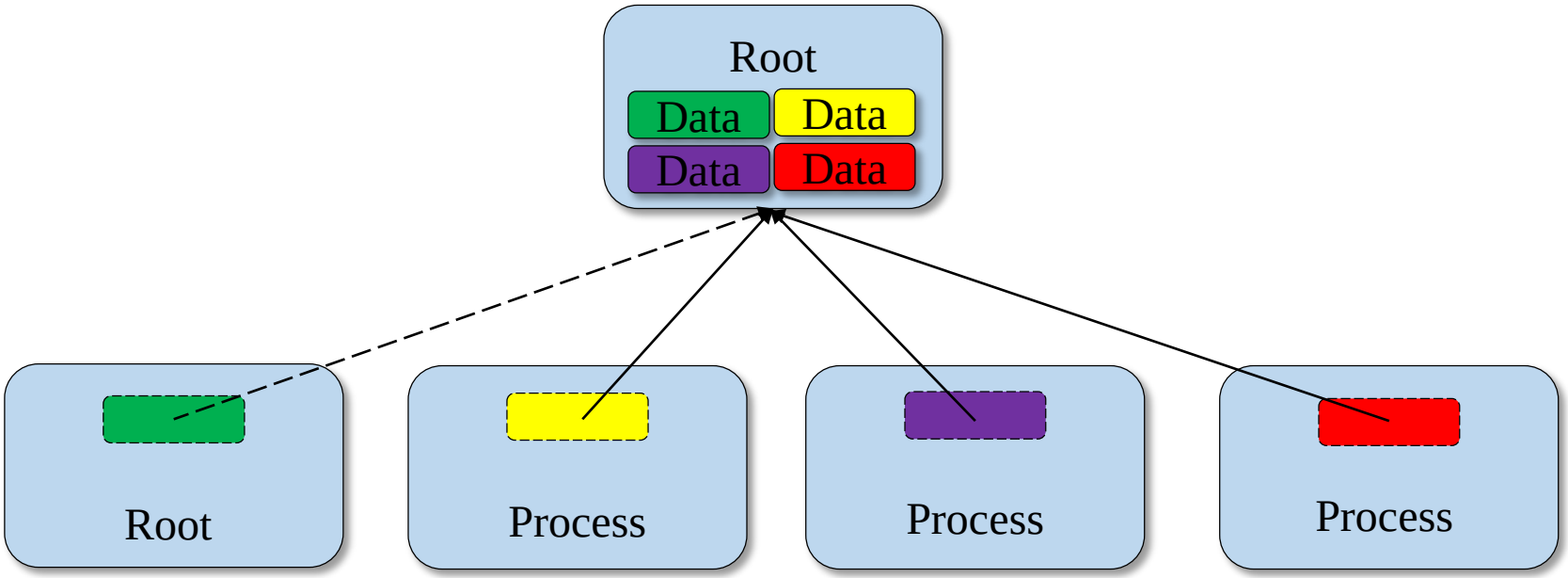
- Sends individual messages from the root process to all other processes



Collective Communication (Gather)

```
MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf,
           recvcnt, recvtype, root, comm)
```

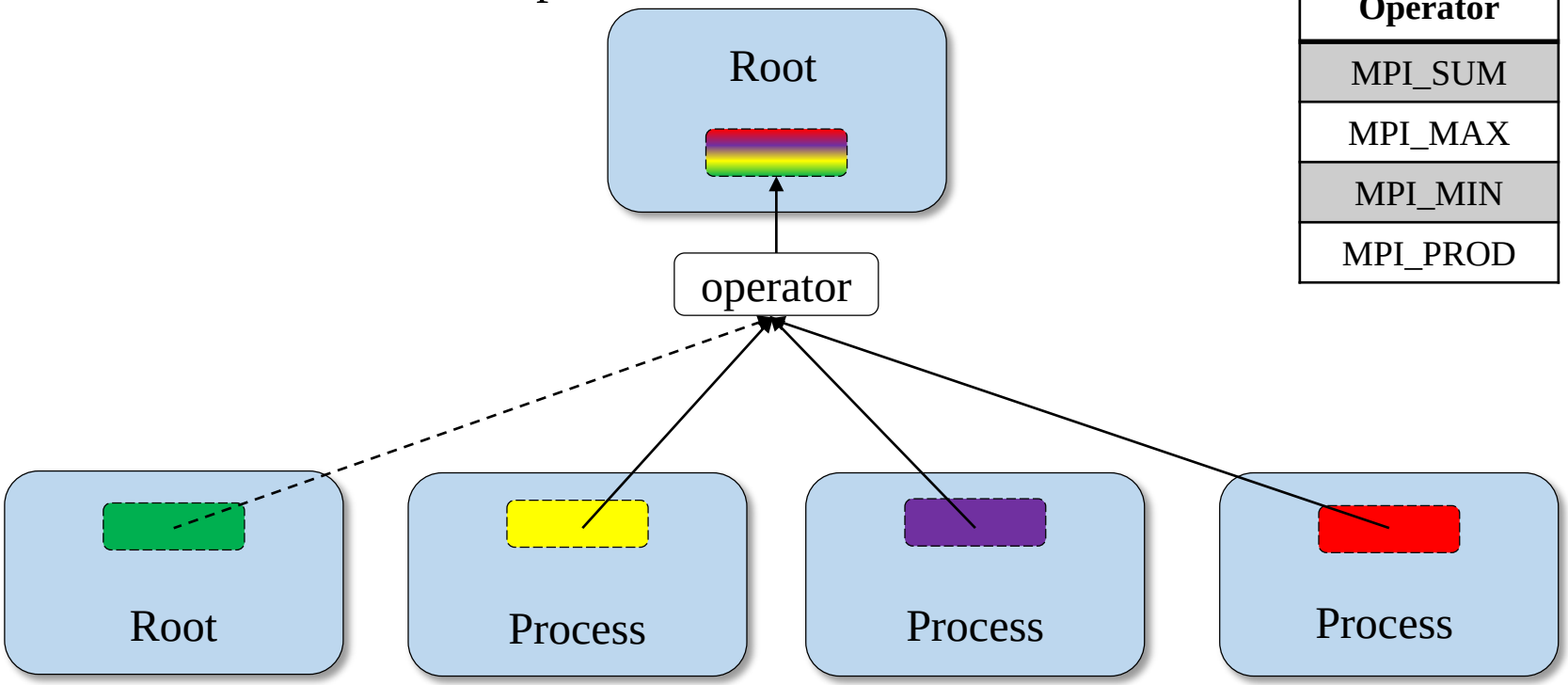
- Opposite of Scatter



Collective Communication (Reduce)

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
           mpi_operation, root, comm)
```

- Applies reduction operation on data from all processes
- Puts result on root process



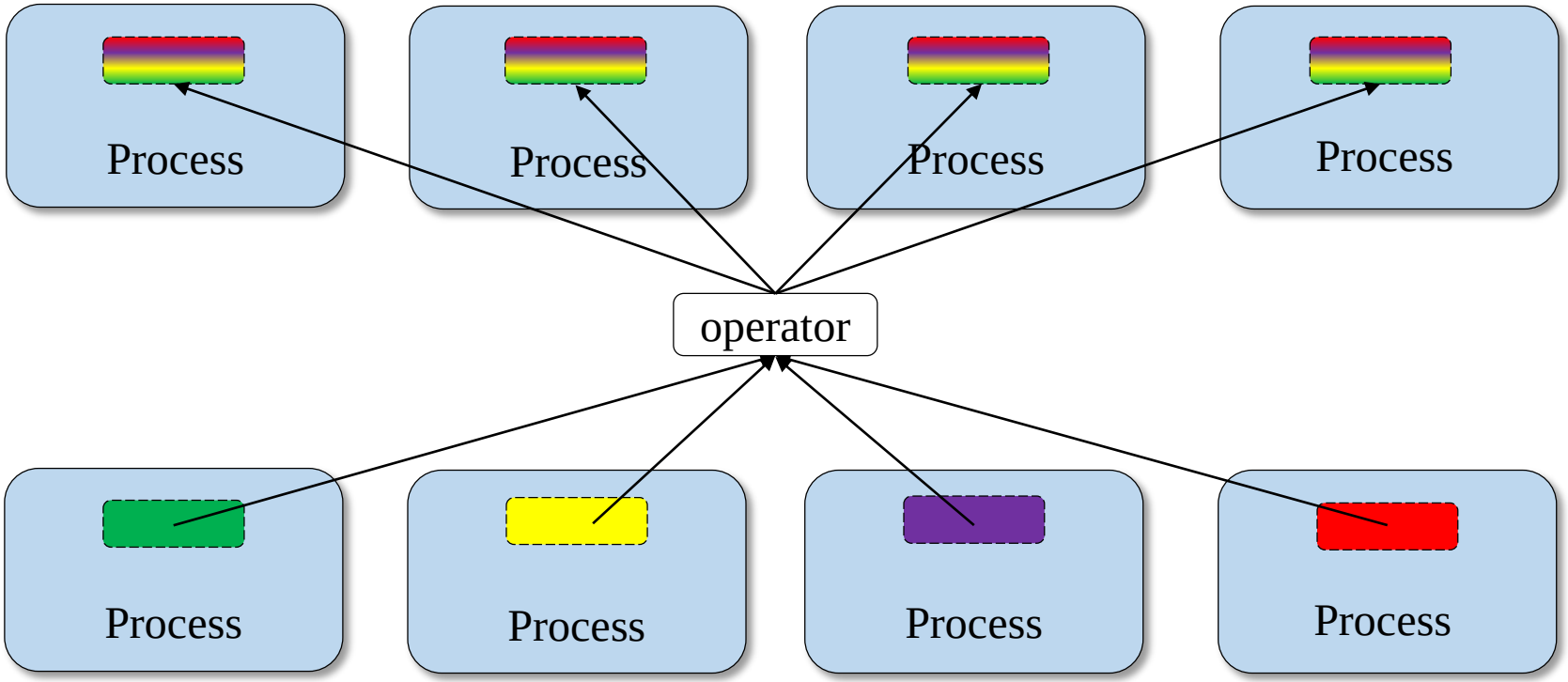
Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

Collective Communication (Allreduce)

```
MPI_Allreduce(&sendbuf, &recvbuf, count,
              datatype, mpi_operation, comm)
```

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

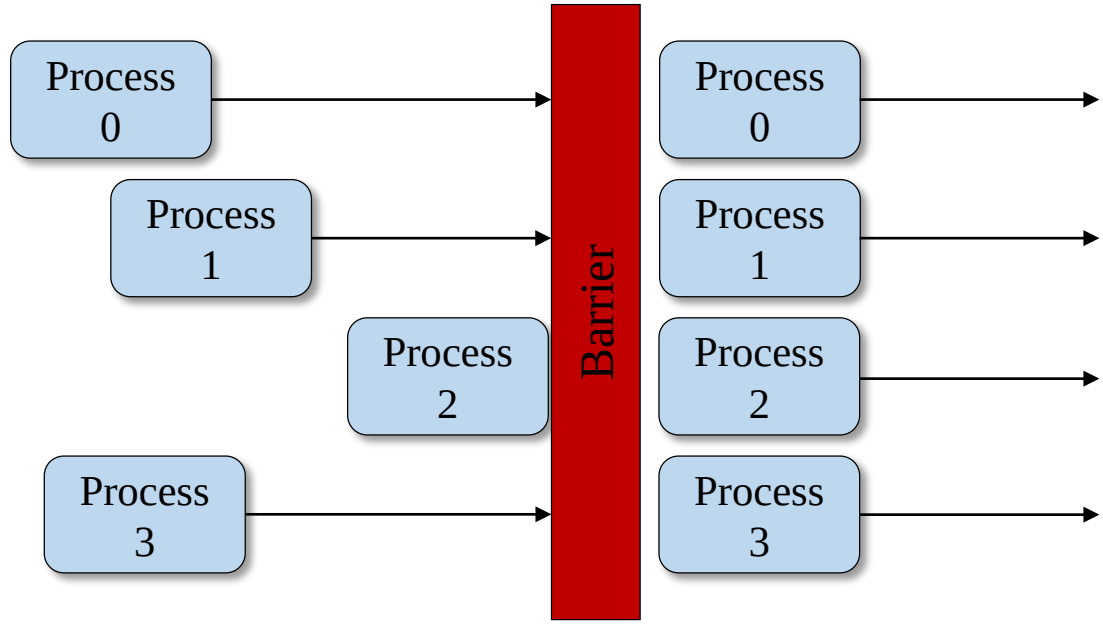
- Applies reduction operation on data from all processes
- Stores results on all processes



Collective Communication (Barrier)

```
MPI_Barrier(comm)
```

- Process synchronization (blocking)
  - All processes forced to wait for each other
- Use only where necessary
  - Will reduce parallelism



Useful MPI Routines

Routine	Purpose/Function
MPI_Init	Initialize MPI
MPI_Finalize	Clean up MPI
MPI_Comm_size	Get size of MPI communicator
MPI_Comm_Rank	Get rank of MPI Communicator
MPI_Reduce	Min, Max, Sum, etc
MPI_Bcast	Send message to everyone
MPI_Allreduce	Reduce, but store result everywhere
MPI_Barrier	Synchronize all tasks by blocking
MPI_Send	Send a message (blocking)
MPI_Recv	Receive a message (blocking)
MPI_Isend	Send a message (non-blocking)
MPI_Irecv	Receive a message (non-blocking)
MPI_Wait	Blocks until message is completed

(Some) MPI Data Types

MPI	C Data Type
MPI_INT	Singed int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_LONG	Signed long int

A note about MPI Errors

- Examples have not done any error handling
- Default: MPI_ERRORS_ARE_FATAL
- This can be changed to MPI_ERRORS_RETURN
  - Not recommended
  - Program must handle ALL errors correctly
- Does have a purpose in fault tolerance
- Long running jobs should always checkpoint in case of errors.

Example

- Situation 1: 5 nodes, 20 cores per node = 100 processes
  - 4 weeks of total run time broken down into 14, 48-hour runs
  - 100 x 14 x 48 = 672,000 core-hours

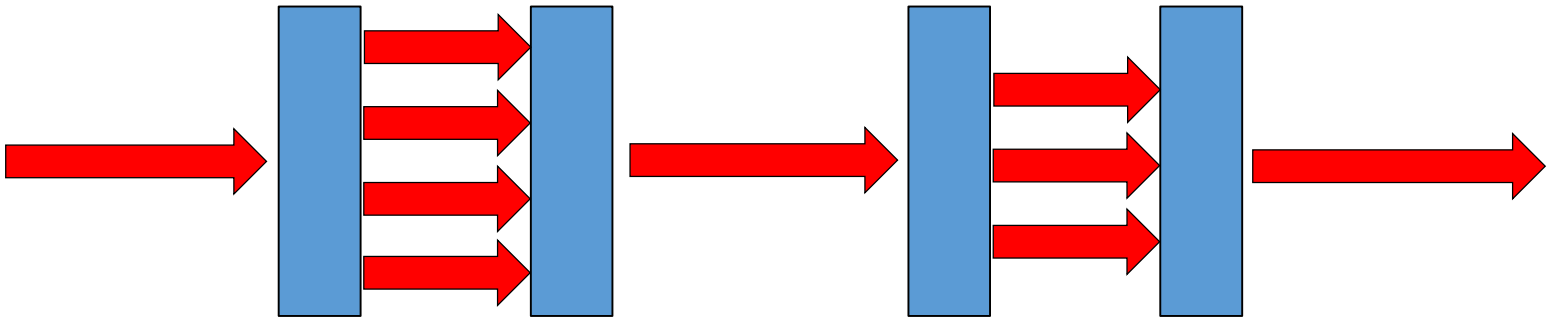
- Situation 2: 3,000 nodes, 20 cores per node = 60,000 processes
  - One 12 hour job
  - 60,000 x 12 = 720,000 core-hours

Hardware Errors

- Unfortunately, hardware fails: nodes die, switches fail
  - In case of a hardware or software error, the program aborts
  - If you aren't checkpointing ALL time for current job is wasted
    - Situation 1: one 4,800 core-hours job lost
    - Situation 2: all 720,000 core-hours lost
  - If you are checkpointing all computation from last checkpoint is lost
    - Situation 1: 1.7 core-hours per minute since last checkpoint
    - Situation 2: 1000 core-hours per minute since last checkpoint

Section 2: OpenMP

(and more…)



OpenMP

- What is it?
  - Open Multi-Processing
  - Completely independent from MPI
  - Multi-*threaded* parallelism
- Standard since 1997
  - Defined and endorsed by the major players
- Fortran, C, C++
- Requires compiler to support OpenMP
  - Nearly all do
- For shared memory machines
  - Limited by available memory
  - Not GPUs

Preprocessor Directives

- Preprocessor directives tell the compiler what to do
- Always start with #
- You've already seen one:

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

- OpenMP directives tell the compiler to add machine code for parallel execution of the following block

```
#pragma omp parallel
```

- "Run this next set of instructions in parallel"

Some OpenMP Subroutines

```
int omp_get_max_threads()
```

- Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

- Returns number of threads in current team\\

```
int omp_get_thread_num()
```

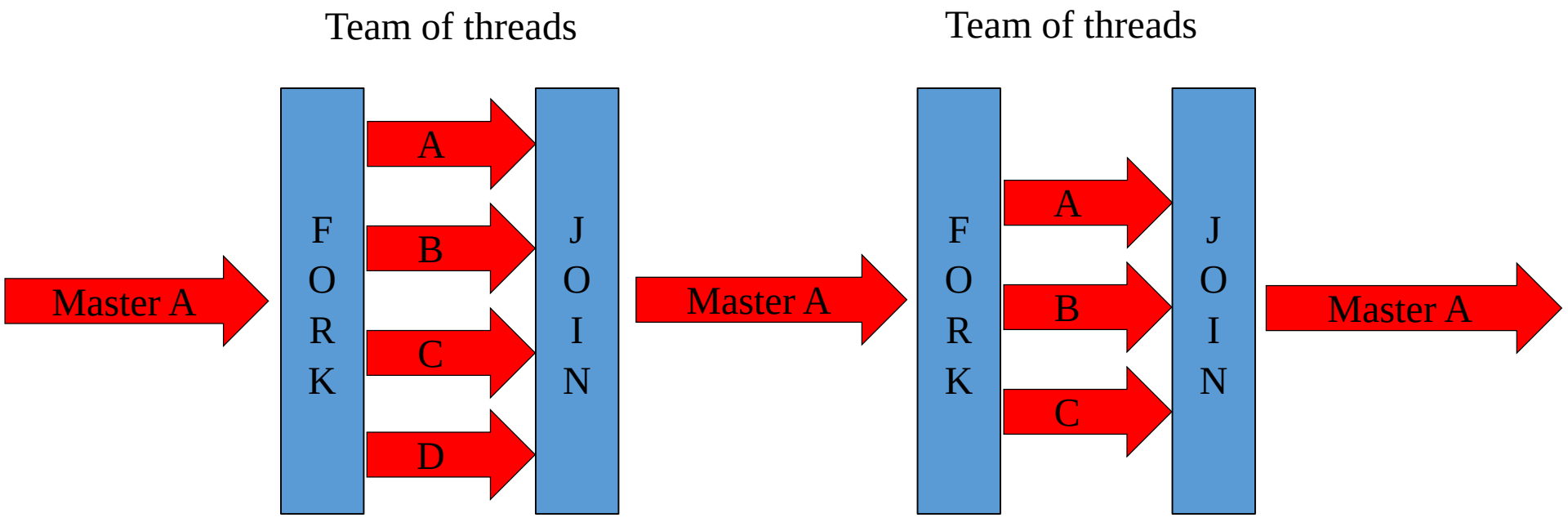
- Returns thread id of calling thread
- Between 0 and omp_get_num_threads-1

Process vs. Thread

- MPI = Process, OpenMP = Thread
- Program starts with a single process
- Processes have their own (private) memory space
- A process can create one or more threads
- Threads created by a process share its memory space
  - Read and write to same memory addresses
  - Share same process ids and file descriptors
- Each thread has a unique instruction counter and stack pointer
  - A thread can have private storage on the stack

OpenMP Fork-Join Model

- Automatically distributes work
- Fork-Join Model



OpenMP Hello World

```
#include <omp.h>  //<-- necessary header file for OpenMP API
#include <stdio.h>

int main(int argc, char *argv[]){

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

Running OpenMP Hello World

```
[user@adroit4]$ module load intel
[user@adroit4]$ icc -qopenmp hello_world_omp.c -o hello_world_omp
```

Compiler flag to enable OpenMP
(-fopenmp for gcc)
(-qopenmp-stubs for icc serial)

Environment variable defining max threads

```
[user@adroit4]$ export OMP_NUM_THREADS=4
[user@adroit4]$ ./hello_world_omp
OpenMP running with 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 2
Hello World from thread 3
```

- OMP_NUM_THREADS defines run time number of threads can be set in code as well using: omp_set_num_threads()
- OpenMP may try to use all available cpus if not set (On cluster–Always set it!)

Lab 2: OpenMP Hello World

```
[user@adroit4 bootcamp]$ module load intel
[user@adroit4 bootcamp]$ icc -qopenmp hello_world_omp.c -o hello_world_omp
```

```
[user@adroit4 bootcamp]$ export OMP_NUM_THREADS=4
[user@adroit4 bootcamp]$ ./hello_world_omp
OpenMP running with 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 2
Hello World from thread 3
```

Private Variables 1

```c
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(shared)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_1.c -o omp_private_1
[user@adroit3]$ export OMP_NUM_THREADS=1
[user@adroit3]$ ./omp_private_1
a=50 b=1049 (expected a=50 b=1049)
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_1
a=50 b=799 (expected a=50 b=1049)
```

Private Variables 2

```c
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(none) private(i) private(a) private(b)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_2.c -o omp_private_2
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_2
a=50 b=0 (expected a=50 b=1049)
```

Private Variables 3

```c
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(none) private(i) private(a) lastprivate(b)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_3 -o omp_private_3
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_3
a=50 b=4197725 (expected a=50 b=1049)
```

Private Variables 4

```
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(none) private(i) firstprivate(a) lastprivate(b)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_4.c -o omp_private_4
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_4
a=50 b=1049 (expected a=50 b=1049)
```

OpenMP Constructs

- Parallel region
  - Thread creates team, and becomes master (id 0)
  - All threads run code after
  - Barrier at end of parallel section

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

   structured_block
```

(not a complete list)

OMP Parallel Clauses 1

```
#pragma omp parallel if (scalar_expression)
```

- Only execute in parallel if true
- Otherwise serial

```
#pragma omp parallel private (list)
```

- Data local to thread
- Values are not guaranteed to be defined on exit (even if defined before)
- No storage associated with original object
  - Use `firstprivate` and/or `lastprivate` clause to override

OMP Parallel Clauses 2

```
#pragma omp parallel firstprivate (list)
```

- Variables in list are private
- Initialized with the value the variable had *before* entering the construct

```
#pragma omp parallel for lastprivate (list)
```

- Only in for loops
- Variables in list are private
- The thread that executes the *sequentially last iteration* updates the value of the variables in the list

OMP Parallel Clause 3

```
#pragma omp shared (list)
```

- Data is accessible by all threads in team
- All threads access same address space

- Improperly scoped variables are big source of OMP bugs
  - Shared when should be private
  - Race condition

```
#pragma omp default (shared | none)
```

- Tip: Safest is to use default(none) and declare by hand

Shared and Private Variables

- Take home message:
  - Be careful with the scope of your variables
  - Results must be independent of thread count
  - Test & debug thoroughly!

- Important note about compilers
  - C (before C99) does not allow variables declared in for loop syntax
    - Compiler will make loop variables private
    - Still recommend explicit

C

```
#pragma omp parallel private(i)
  for (i=0; i<N; i++) {
    b = a + i;
  }
```

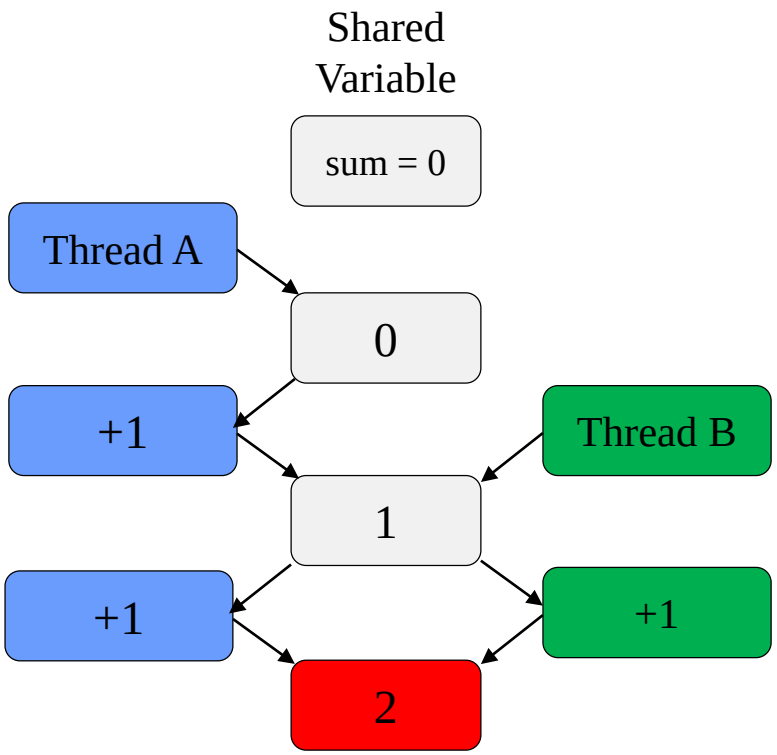
C++

```
#pragma omp parallel
  for (int i=0; i<N; i++) {
    b = a + i;
  }
```

Automatically private

Caution: Race Condition

- When multiple threads simultaneously read/write shared variable
- Multiple OMP solutions
  - Reduction
  - Atomic
  - Critical




```
#pragma omp parallel for private(i) shared(sum)
  for (i=0; i<N; i++) {
    sum += i;
  }
```

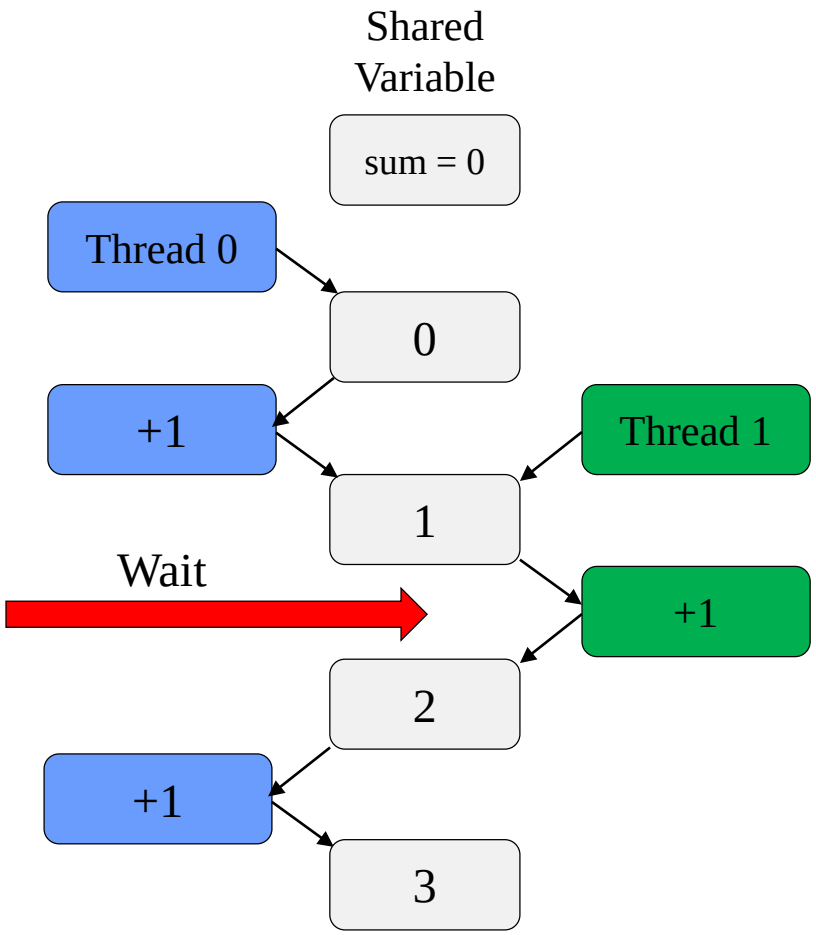
Should be 3!

Critical Section

- One solution: use critical
- Only one thread at a time can execute a critical section

```
#pragma omp critical
   {
     sum += i;
   }
```

- Downside?
  - SLOOOOWWW
  - Overhead & serialization

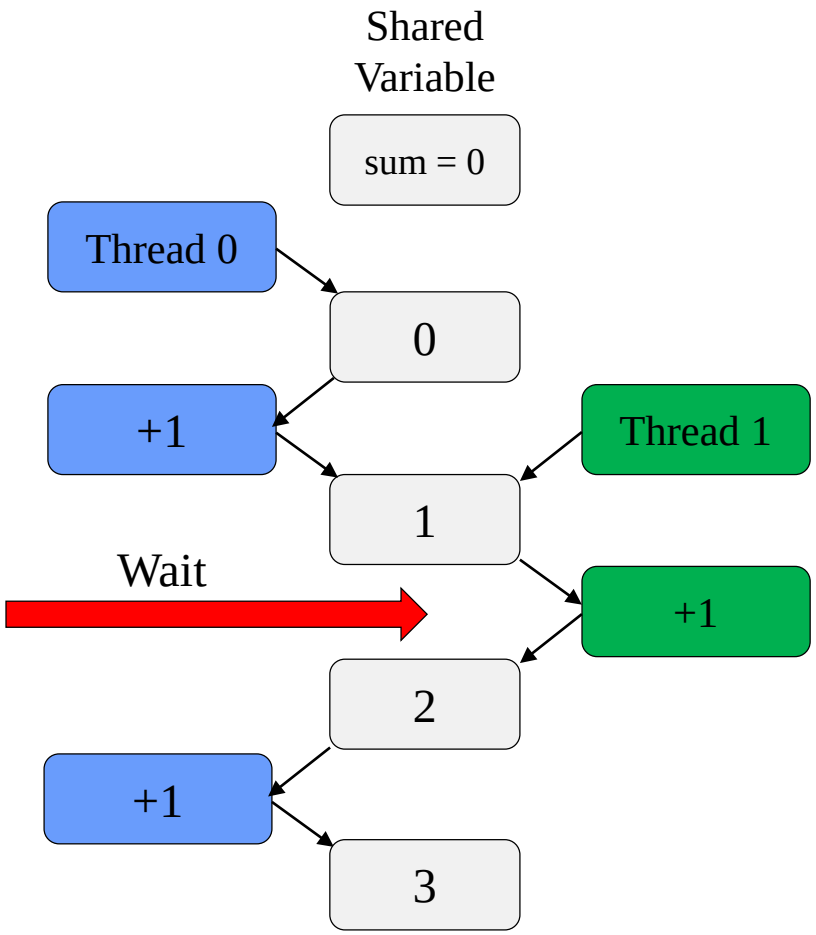


OMP Atomic

- Atomic like “mini” critical
- Only one line
  - Certain limitations

```
#pragma omp atomic
    sum += i;
```

- Hardware controlled
  - Less overhead than critical



OMP Reduction

```
#pragma omp reduction (operator:variable)
```

- Avoids race condition
- Reduction variable must be shared
- Makes variable private, then performs operator at end of loop
- Operator cannot be overloaded (c++)
  - One of: +, *, -, / (and &, ^, |, &&, ||)
  - OpenMP 3.1: added min and max for c/c++

Reduction Example

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

int main() {

  int i;
  const int N = 1000;
  int sum = 0;

#pragma omp parallel for private(i) reduction(+: sum)
  for (i=0; i<N; i++) {
    sum += i;
  }

  printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
```

```
[user@adroit3]$ gcc -fopenmp omp_race.c -o omp_race.out
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_race.out
reduction sum=499500 (expected 499500)
```

Relative Performance

- See example omp_race_time.c
  - For 4 threads:
    - Reduction is 100x faster than critical
    - Reduction is 10x faster than atomic
    - Reduction is faster than atomic with private sums (see example)
  - Note: read the disclaimer at the top.

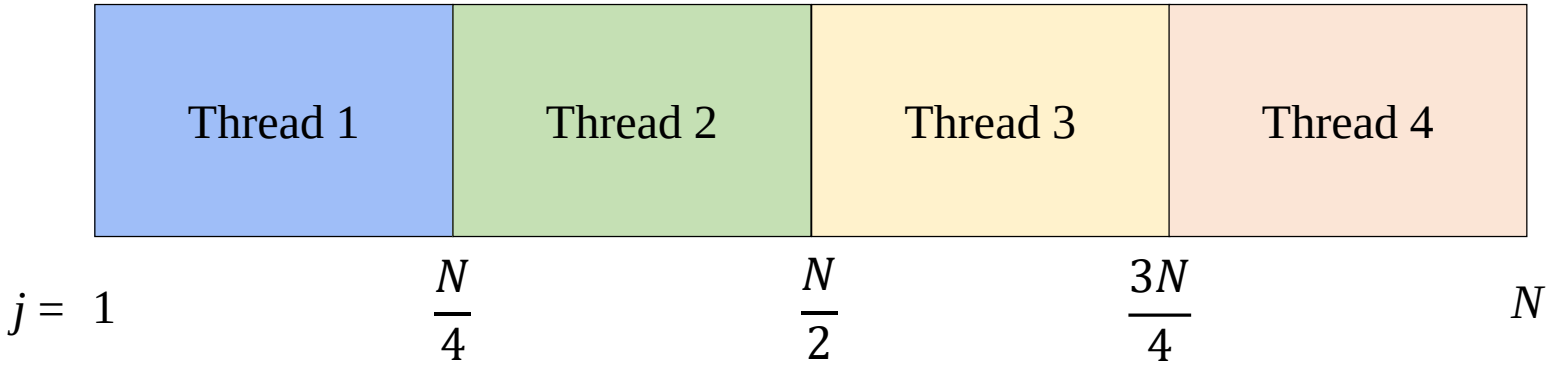
- Don’t reinvent the wheel, use Reduction!

Scheduling `omp for`

- How does a loop get split up?
  - In MPI, we have to do it manually
- If you don't tell it what to do, the compiler decides
- Usually compiler chooses "static" – chunks of N/p

```
#pragma omp parallel for default(shared) private(j)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

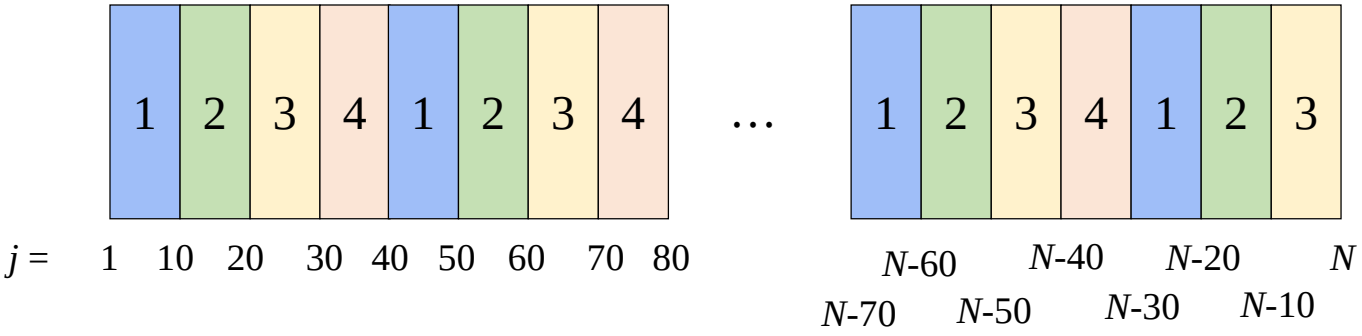
Unspecified schedule



Static Scheduling

- You can tell the compiler what size chunks to take

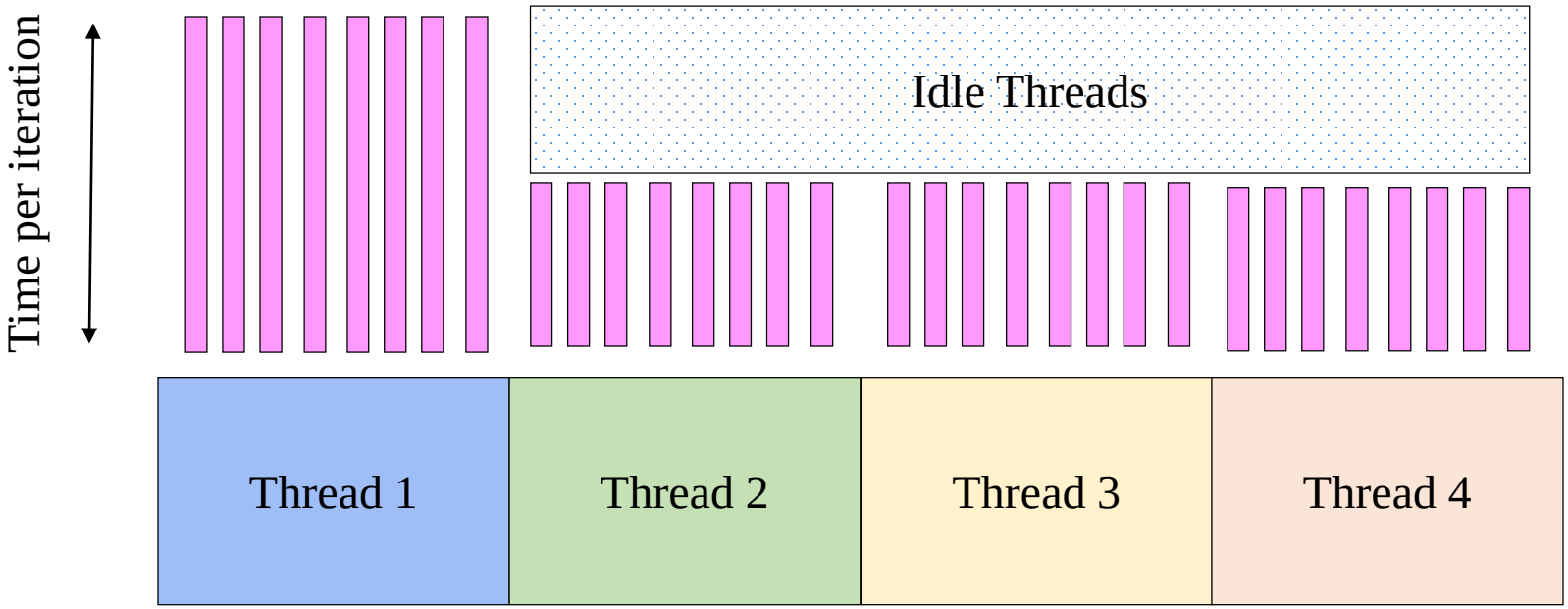
```
#pragma omp parallel for default(shared) private(j) schedule(static,10)
  for (j=0; j<N; j++) {
        ... // some work here
}
```



- Keeps assigning chunks until done
- Chunk size that isn't a multiple of the loop will result in threads with uneven numbers

Problem with Static Scheduling

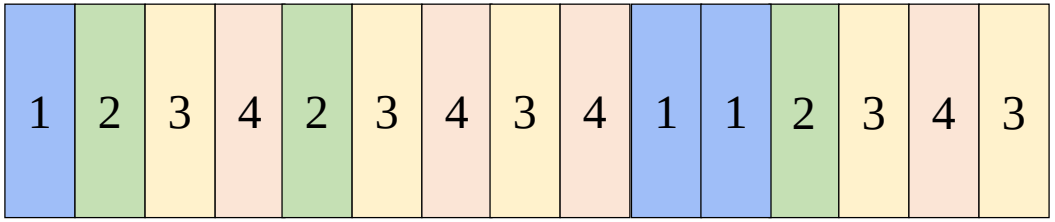
- What happens if loop iterations do not take the same amount of time?
  - Load imbalance



Dynamic Scheduling

- Chunks are assigned on the fly, as threads become available
  - When a thread finishes one chunk, it is assigned another

```
#pragma omp parallel for default(shared) private(j) schedule(dynamic,10)
  for (j=0; j<N; j++) {
        ... // some work here
}
```



- Caveat Emptor: higher overhead than static!

`omp for` Scheduling Recap

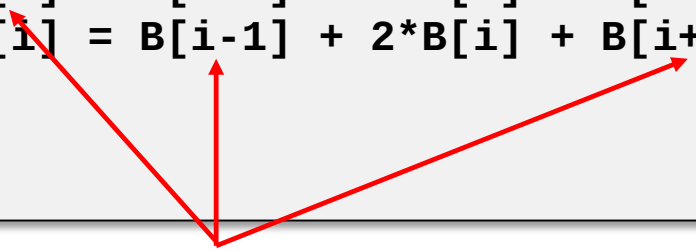
```
#pragma omp parallel for schedule(type [,size])
```

- Scheduling types
  - Static
    - Chucks of specified size assigned round-robin
  - Dynamic
    - Chunks of specified size are assigned when thread finishes previous chunk
  - Guided
    - Like dynamic, but chunks are exponentially decreasing
    - Chunk will not be smaller than specified size
  - Runtime
    - Type and chunk determined at runtime via environment variables

Where not to use OpenMP

What could go wrong here?

```
...
  const int N = 1000;
  int A[N], B[N], C[N];

... // arrays initialized etc.

#pragma omp parallel for shared(A,B,C) private(i)
  for (i=1; i<(N-1); i++) {
    B[i] = A[i-1] + 2*A[i] + A[i+1];
    C[i] = B[i-1] + 2*B[i] + B[i+1];
  }

...
```



`B[i-1]` and `B[i+1]` are not guaranteed to be available/correct

OpenMP API

- API for library calls that perform useful functions
  - We will only touch on a few
- Must include “omp.h”
- Will not compile without openmp compiler support
  - Intel has the -qopenmp-stubs option

```
#include <omp.h>   //<-- necessary header file for OpenMP API
#include <stdio.h>

int main(int argc, char *argv[]){

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

OpenMP API

```
void omp_set_num_threads(int num_threads)
```

- Sets number of threads used in next parallel section
- Overrides OMP_NUM_THREADS environment variable
- Positive integer

```
int omp_get_max_threads()
```

- Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

- Returns number of threads currently in team

```
int omp_get_thread_num()
```

- Returns thread id of calling thread
- Between 0 and omp_get_num_threads-1

```
double omp_get_wtime()
```

- Returns number of seconds since some point
- Use in pairs time=(t2-t1)

OpenMP Performance Tips

- Avoid serialization!
- Avoid using `#pragma omp parallel for` before each loop
  - Can have significant overhead
    - Thread creation and scheduling is NOT free!!
  - Try for broader parallelism
    - One `#pragma omp parallel`, multiple `#pragma omp for`
  - Always try to parallelize the outer most loop
- Use reduction whenever possible
- Minimize I/O
- Minimize `critical`
  - Use `atomic` instead of `critical` where possible

Hybrid OpenMP & MPI

- Two-level Parallelization
  - Mimics hardware layout of cluster
    - Only place this really make sense
  - MPI between nodes
  - OpenMP within shared-memory nodes
- Why?
  - Saves memory by not duplicating data
  - Minimize interconnect communication by only having 1 MPI process per node
- Careful of MPI calls within OpenMP block
  - Safest to do MPI calls outside (but not required)
- Obviously requires some thought!

Hybrid Programming

- In hybrid programming each process can have multiple threads executing simultaneously
  - All threads within a process share all MPI objects
    - Communicators, requests, etc.
- MPI defines 4 levels of thread safety
  - MPI_THREAD_SINGLE
    - One thread exists in program
  - MPI_THREAD_FUNNELED
    - Multithreaded but only the master thread can make MPI calls
    - Master is one that calls MPI_Init_thread()
  - MPI_THREAD_SERIALIZED
    - Multithreaded, but only one thread can make MPI calls at a time
  - MPI_THREAD_MULTIPLE
    - Multithreaded and any thread can make MPI calls at any time
- Use MPI_Init_thread instead of MPI_Init if more than single thread

```
MPI_Init_thread(int required, int *provided)
```

Hybrid Programming

- Safest (easiest) to use MPI_THREAD_FUNNLED
- Fits nicely with most OpenMP models
  - Expensive loops parallelized with OpenMP
  - Communication and MPI calls between loops
- Eliminates need for true “thread-safe” MPI
- Parallel scaling efficiency may be limited (Amdahl’s law) by MPI_THREAD_FUNNLED approach
- Moving to MPI_THREAD_MULTIPLE does come at a performance price (and programming challenge)

Strategies for Debugging

- Sometimes `printf` or `cout` during development can save headaches down the road
  - Tip: Flush stdout (or use unbuffered)
  - And write the MPI process rank

```
printf("Process %d has var1=%g var2=%d\n”, rank, var1, var2);
fflush(stdout);
```

```
 std::cout.setf(std::ios::unitbuf);
```

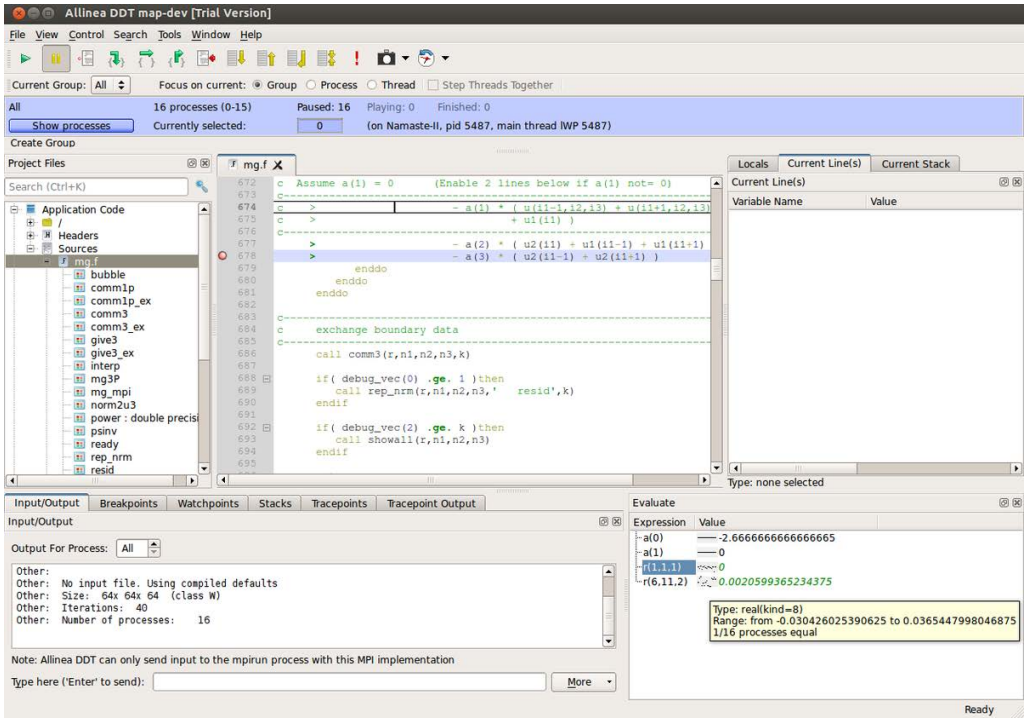
  - Stderr is already unbuffered

```
fprintf(stderr, "Process %d has var1=%g var2=%d\n”, rank, var1, var2);
```

```
cerr<<"Process "<<rank<<" has var1="<<var1<<" var2="<<var2<<endl;
```

Debugging

- DDT
  - Visual debugger
  - Licensed Product
  - Available on clusters



- http://www.princeton.edu/researchcomputing/faq/debugging-with-ddt-on-the/

Profiling

- Many HPC codes operate far below peak
- Measuring the performance of your code
  - Find the “hotspots”
    - How much time is spent in each function
    - Not always where you think it is
    - Identify regions to optimize/parallelize
  - Hardware Performance
    - Vectorization, cache misses, branch misprediction, etc.

- Quick & dirty: Put time calls around loops
- Free & basic: gprof

Timing with MPI and OpenMP APIs

- MPI

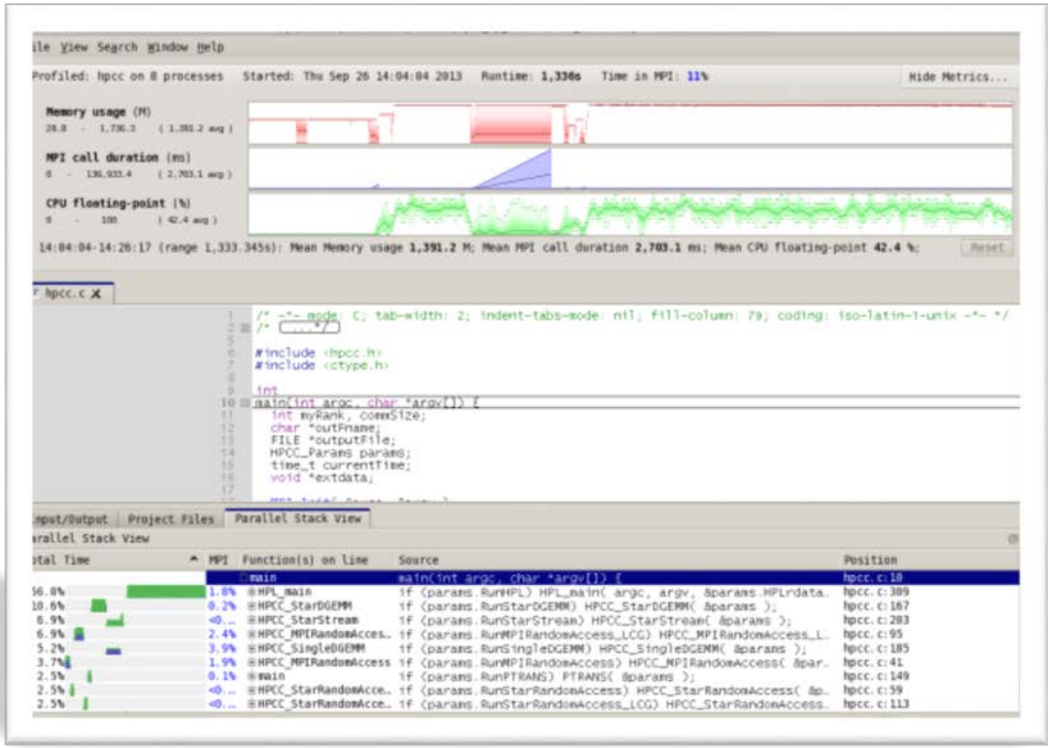
```
double t1 = MPI_Wtime();
  //do something expensive...
double t2 = MPI_Wtime();

if(my_rank == final_rank) {
printf("Total runtime = %g s\n", (t2-t1));
}
```

- OpenMP

```
double t1, t2;
t1=omp_get_wtime();
//do something expensive...
t2=omp_get_wtime();
  printf("Total Runtime = %g\n", t2-t1);
```

Allinea MAP

- Allinea MAP
  - Commercial profiler
  - C, C++, Fortran
  - Lightweight GUI

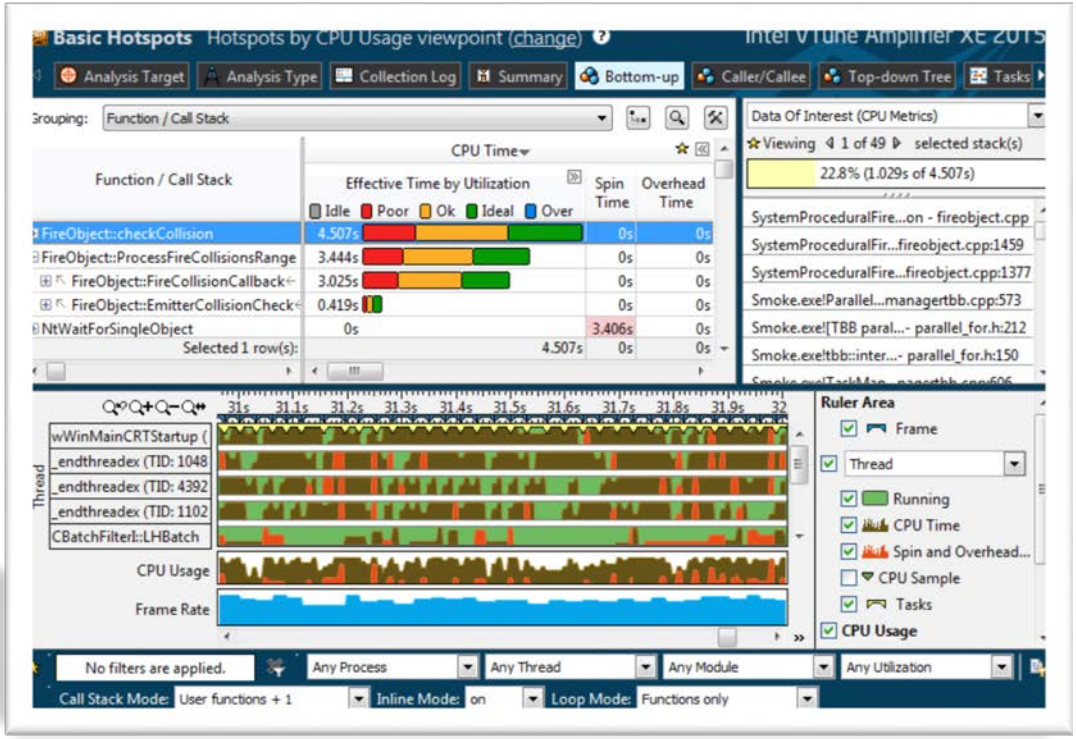


- Source code profiling
- Compute, I/O, Memory, MPI bottlenecks

http://www.princeton.edu/researchcomputing/faq/profiling-with-allinea-ma/

Intel VTune

- Intel VTune Amplifer XE
  - Commercial Profiler
  - Extraordinarily powerful
    (and complicated)
  - Nice GUI
- Shared memory only
  - Serial
  - OpenMP
  - MPI on single node

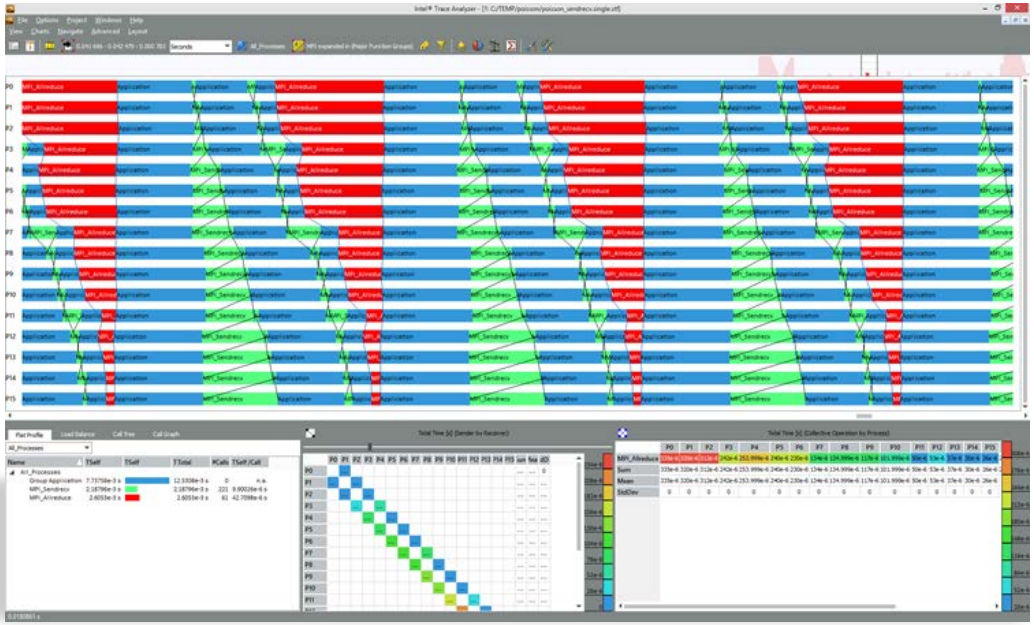


- Excellent for hardware performance and threading

http://www.princeton.edu/researchcomputing/faq/profiling-with-intel-vtun/

Intel Trace Analyzer and Collector

- Intel Trace Analyzer and Collector
  - Creates timeline for every process
- Good for MPI scaling & bottlenecks
- Can have large overhead & big files



http://www.princeton.edu/researchcomputing/faq/using-intel-trace-analyze/

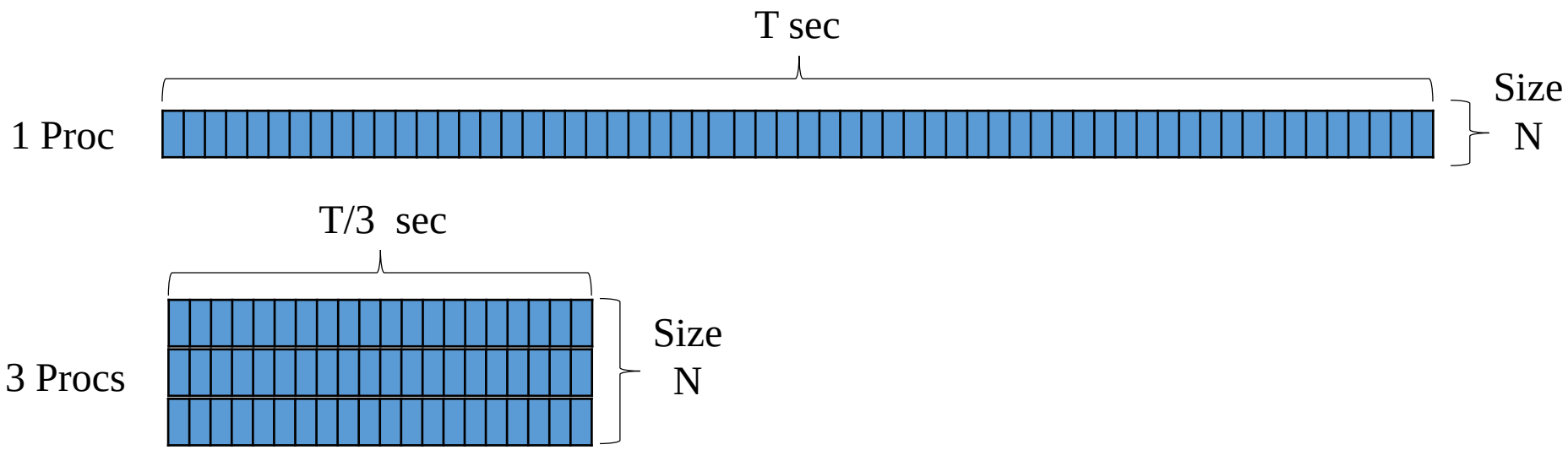
Scaling

- Measure the parallel performance of your code
- Know your code
- For time on national supercomputers (XSEDE) *proof* of scaling is required
  - CPU hours are a precious commodity
  - Prevents wasting resources
  - Not a requirement at Princeton
- Algorithm and implementation specific

- Remember Amdahl's Law

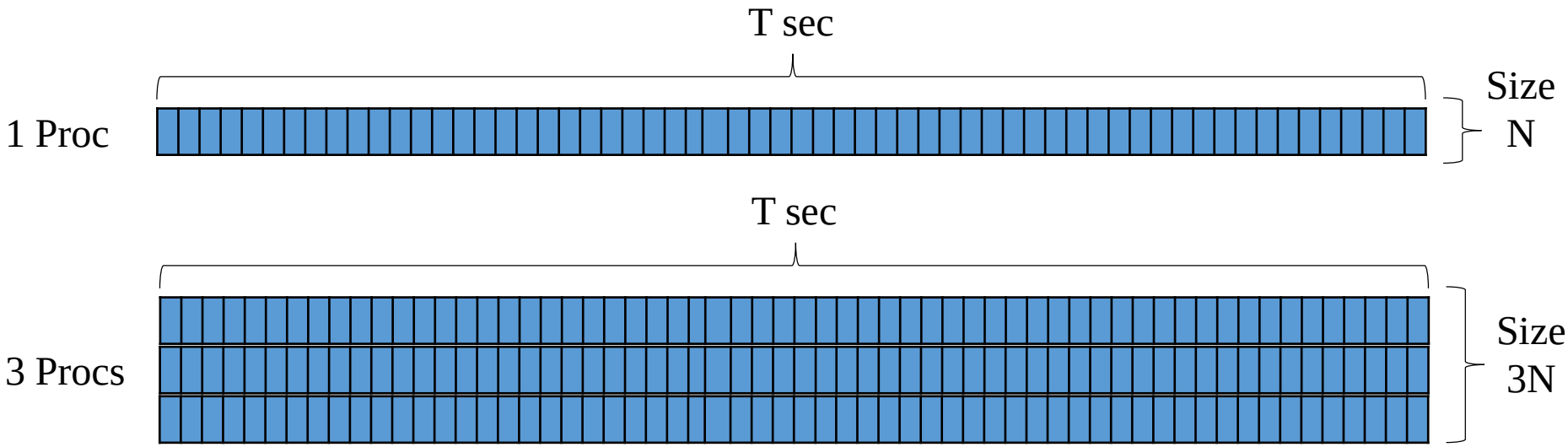
Scaling: Strong vs. Weak

- Strong Scaling
  - Fixed problem size
  - Measure how solution time decreases with more processors



Weak Scaling

- Weak Scaling
  - Fixed problem size per processor
  - Measure by solution time remaining unchanged with larger problem (more processors)



Exercise: Numerical Integration

- Calculate π numerically

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- Integrate numerically with midpoint rule

$$\int_a^b f(x)dx \approx \sum_{j=0}^{N-1} f\left(x_j + \frac{h}{2}\right)h$$

N = number of intervals
$x_j = a + j*h$
$h = (b - a)/N$

Exercise: Numerical Integration

- Serial (non-parallel) program for computing π by numerical integration is in the bootcamp directory.
- As an exercise, try to make MPI and OpenMP versions.
- See the full-day version of this workshop for more information:

```
[user@adroit4 bootcamp]$ wget http://tigress-
web/~icosden/Intro_Parallel_Computing/2018-Spring/lab_materials.tgz
[user@adroit4 bootcamp]$ tar –xvf lab_material.tgz
```

Upcoming Workshops

- Introduction to Parallel Programming with MPI and OpenMP
  - Dr. Stephane Either, PPPL
  - December ___, 2018

Possible Spring Workshops

Introduction to Debugging with the Allinea DDT Advanced Debugger

- Dr. Stephane Either, PPPL

- Introduction to Parallel Programming with MPI and OpenMP
  - Ian Cosden, Princeton Research Computing
  - 2 day workshop

Resources

- Where to learn more?
  - OpenMP
    - YouTube videos “Introduction to OpenMP” by Tim Matteson
    - http://www.openmp.org/resources
    - https://computing.llnl.gov/tutorials/openMP/
    - Online + Google (what can’t you learn?)
  - MPI
    - http://www.mpi-forum.org (location of the MPI standard)
    - http://www.llnl.gov/computing/tutorials/mpi/
    - http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
    - http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
    - http://www-unix.mcs.anl.gov/mpi/tutorial/
    - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
    - Open MPI (http://www.open-mpi.org/)
    - Books:
      - Using MPI “Portable Parallel Programming with the Message-Passing Interface” by William Gropp, Ewing Lusk, and Anthony Skjellum
      - Using MPI-2 “Advanced Features of the Message-Passing Interface”

Introduction to Parallel Programming with MPI and OpenMP

Questions?