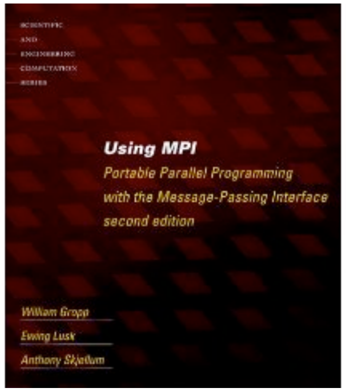
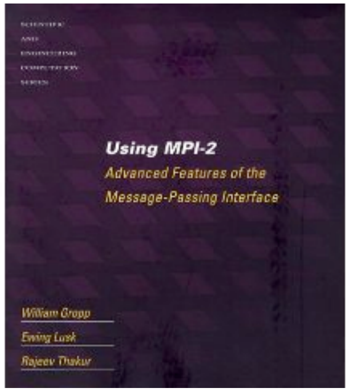
Alfio Lazzaro: Introduction to "Message-Passing Interface"

- What you will heard in this lecture:
  - Parallel implementations for High Performance Computing (HPC)
  - Basic elements of Message-Passing Interface (MPI)
  - Basic MPI functions: point-to-point and collective communications
  - Examples
- What you will NOT heard in this lecture:
  - A complete list of all MPI functions
  - Advanced use of MPI
- **Note: this is NOT an alternative to a book on MPI!**

  Get your hands dirty is the best way to understand MPI!

References


Scientific and Engineering Computation Series
Using MPI
Portable Parallel Programming with the Message-Passing Interface
second edition
William Gropp
Ewing Lusk
Anthony Skjellum


Scientific and Engineering Computation Series
Using MPI-2
Advanced Features of the Message-Passing Interface
William Gropp
Ewing Lusk
Rajeev Thakur

- Books:
  - “Using MPI”, Gropp, Lusk and Skjellum, http://www.amazon.com/Using-MPI-Programming-Engineering-Computation/dp/0262571323
  - “Using MPI-2”, Gropp, Lusk and Thakur, http://www.amazon.com/Using-MPI-2-Scientific-Engineering-Computation/dp/0262571331
- Online tutorials:
  - http://www.llnl.gov/computing/tutorials/mpi/
  - http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiintro/index.htm
  - http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html

INTRODUCTION

Parallel computing on clusters

- Current steady trend about high performance architectures is to build large clusters of **symmetric multiprocessing** (SMP) nodes with distributed memory
  - Several nodes connected with high-speed networks (Gigabit Ethernet, InfiniBand, Myrinet,…)
  - Each node has several CPUs (multi-cores/multi-sockets), with large shared memory
- Hybrid of distributed and shared memory programming is possible, but still not well exploited
  - Usually only distributed memory paradigm is used on clusters, even for workers of the same node

Key Factors

- Parallelism on clusters achieved with exchange of messages between the computational nodes (workers), using network system
    - Synchronization of the messages
    - Low overhead in the communications
    - Fast network connections, using particular topologies
- Keep in mind that latency in the network communications is O(10) microseconds for 1 KByte message (for reference: main memory latency is O(0.1) microseconds, disk latency O(10) microseconds)
- Require development of particular algorithms that keep low the number of communications and that are optimized for the hardware

Top500 (http://www.top500.org)

- Ranking of the 500 most powerful known computer systems in the world
- First position (June 2009):
  - **IBM Roadrunner (@ LANL, USA):** 12,960 IBM PowerXCell 8i (9 cores) and 6,480 AMD Opteron dual-core processors
    - 122,400 computing cores
    - 3,240 nodes, interconnected via InfiniBand (16 Gbit/s)
    - 1,105 petaflops (first system to reach petaflops scale)
    - 444.94 megaflops per Watt (2.35 MW total)
- Note:
  - LINPACK benchmark (linear algebra) to set the performance
  - For reference: i7 @ 3.2 GHz has about 51 gigaflops

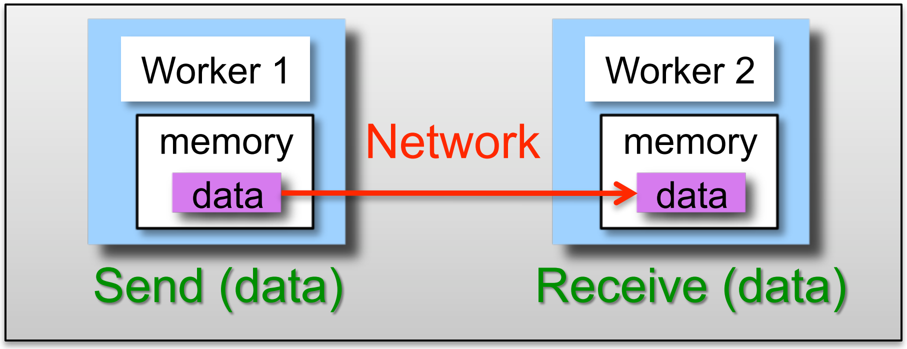
Parallel paradigms for clusters

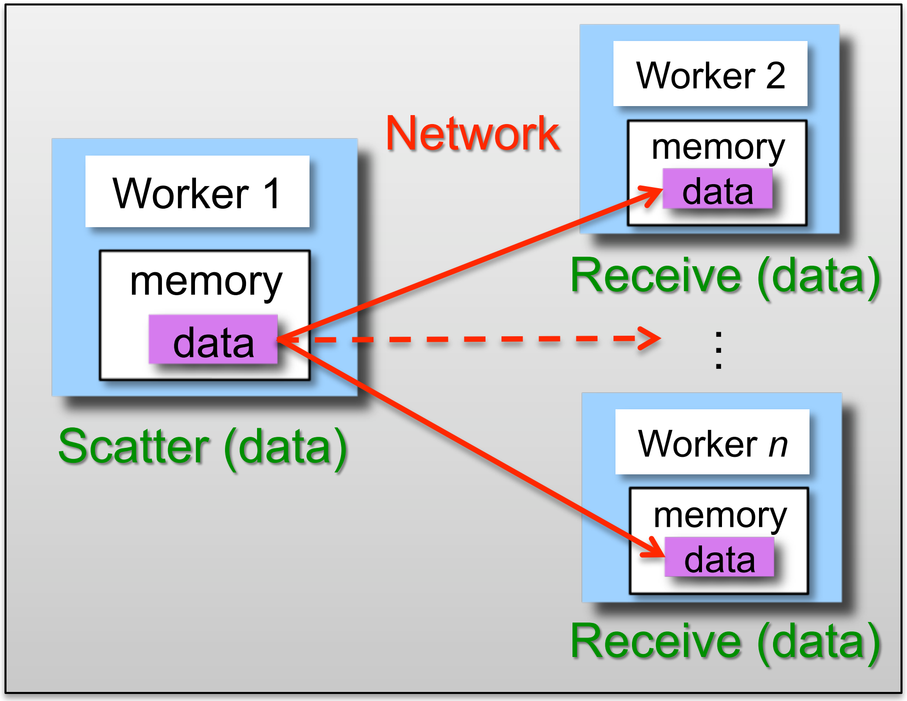
- Parallel computing types:
  - **SPMD**: Same program, different data
  - **MIMD**: Different programs, different data
  - Essentially they are the same because any MIMD can be made SPMD
- Communications for data exchange between workers:
  - **Cooperative**: all parties agree to transfer data
  - **One sided**: one worker performs transfer of data

Data exchange

- Cooperative:
  - Each send/receive MUST have a corresponding receive/send
    - **Point-to-point**: message passing between **two**, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation
    - **Collective**: involve **all** MPI tasks: reduction, broadcast, scatter/gather, all to all.

Point-to-Point



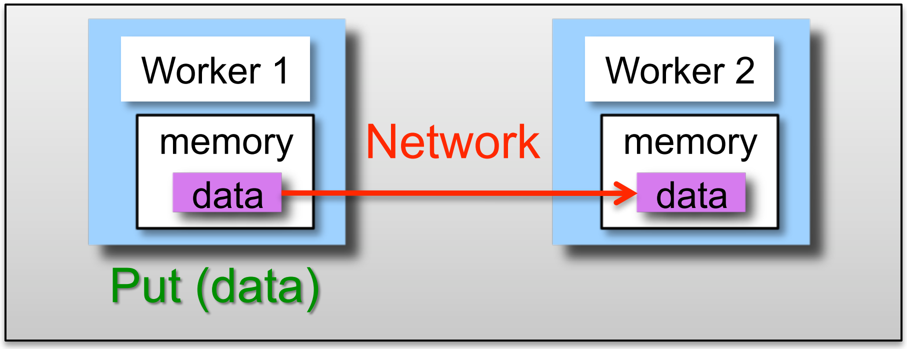


Collective

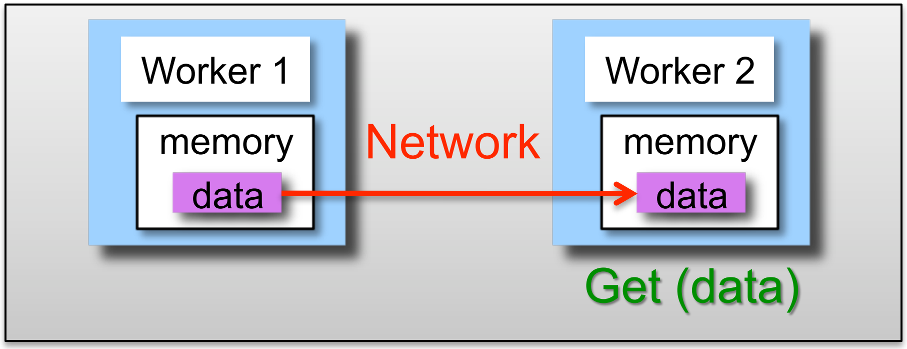
Data exchange

- One-sided:
  - Direct access to the memory of another worker
  - Include shared memory operations (put/get) and remote accumulate operations.

One-sided: Put



One-sided: Get



MESSAGE-PASSING INTERFACE (MPI)

What is MPI (http://www.mpi-forum.org)

- MPI is not a "complete" standard, but
  - It is a specification for APIs that allow many workers to communicate (distributed memory system)
    - It guarantees the portability for almost every distributed memory architecture
  - It provides a language-independent communication protocol
    - Bindings for Fortran, C, C++, Java (and correlated languages)
  - Both cooperative (point-to-point and collective) and one-sided communications are supported
  - Several implementations, depending on the hardware (mainly developed by cluster vendors)
    - It guarantees the best performance on a specific hardware

MPI Implementations

- Different implementations:
  - MPICH: http://www.mcs.anl.gov/research/projects/mpich2
  - Open MPI: http://www.open-mpi.org
  - custom MPI implementation for specific clusters (Cray, IBM,…) and networks
  - commercial implementations from HP, Intel, Microsoft…
- Each implementation decides the low-level treating of the data, depending of the hardware, in order to have the best possible performances (see backup slides for some examples)
  - Transparent to the user
  - Different performance (and results) depending on the implementation: be aware of your MPI implementation!

MPI-1 & MPI-2 Specifications

- Two versions of MPI currently used:
  - MPI-1 (version 1.3)
    - First draft in 1994
    - Cooperative data exchange and static runtime environment
    - About 128 functions
  - MPI-2 (version 2.2)
    - includes new features such as parallel I/O, dynamic runtime environment and one-sided data exchange
    - over 500 functions
- NOTE: MPI-2 is an “extension” of the MPI-1 functionality, although some functions have been deprecated
  - Both versions are used
  - MPI-1.3 programs still work under MPI implementations compliant with the MPI-2 standard

Caveats of this lecture

- We will focus on MPI-1 functions
  - The majority of problems can be solved using cooperative data exchange
  - No need to know all functions
    - Basically only about 20 functions are used in usual problems
- We will not take care of shared memory on the single node
- We will consider only MPI functions that are implicitly synchronized in the data communication
- We will use the C++ bindings of the functions
  - Fortran and C syntaxes are more or less similar

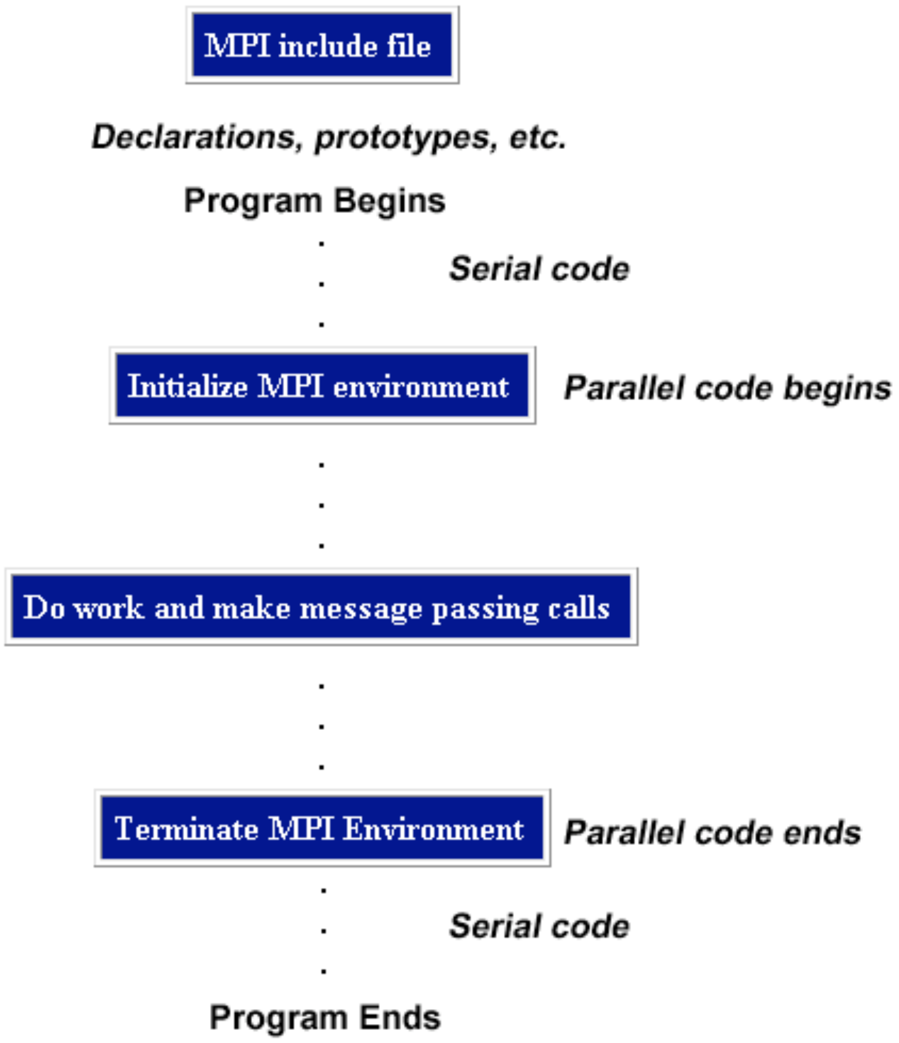
MPI PROGRAMS

MPI program structure

- Only one program is written
  - by default, every line of the code is executed by each worker
    - For example, if the code contains `float v = 0;` each worker will locally creates a variable and assigns the value
  - Specific part of the code to be executed by specific workers must be declared inside an `if` statement

```
float v;
if (workerID<3) v = 2.;
else v = 4.;
```

Here `workerID` identifies each worker


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work and make message passing calls
Terminate MPI Environment
Parallel code ends
Serial code
Program Ends

The “Hello World” example

```
#include "mpi.h"
#include <iostream>

int main(int argc, char *argv[])
{
  MPI::Init(); // MPI Initialization
  int workerID = MPI::COMM_WORLD.Get_rank();
  int nWorkers = MPI::COMM_WORLD.Get_size();

  std::cout << "Hello world! I'm the worker " << workerID
            << " of " << nWorkers << " workers." << std::endl;

  MPI::Finalize(); // MPI Finalization

  return 0;

}
```

Compile and execute

- MPI installs few wrappers for the compilation, depending on the language
  - mpic++   mpicc   mpicxx   mpif77   mpif90
  - The wrappers uses the normal compilers (GNU, Intel, PGA,…)
- They allow to use the correct MPI includes and library
  - You can specify the normal compiler parameters:

```
mpic++ -O2 helloworld.cxx -o helloworld
```

- To execute, you need the mpirun wrapper:

```
mpirun -np 10 ./helloworld
```

  - Note that the number of processors used is specified in the command line. It cannot be changed (static) during the execution (MPI-1 specification; MPI-2 allows a dynamic number)

"Hello World" output

- The stdout/stdin/stderr are in common for the workers

```
helloworld $ mpirun -np 10 ./helloworld
Hello world! I'm the worker 0 of 10 workers.
Hello world! I'm the worker 1 of 10 workers.
Hello world! I'm the worker 2 of 10 workers.
Hello world! I'm the worker 6 of 10 workers.
Hello world! I'm the worker 3 of 10 workers.
Hello world! I'm the worker 4 of 10 workers.
Hello world! I'm the worker 5 of 10 workers.
Hello world! I'm the worker 7 of 10 workers.
Hello world! I'm the worker 8 of 10 workers.
Hello world! I'm the worker 9 of 10 workers.
helloworld $
```

Init and Finalize operations

- **`void MPI::Init()`**
  - All MPI functions MUST be used after this function
  - It can be called just one time in the program
  - Create the default communicator, called **`MPI::COMM_WORLD`**
  - Assign a rank/identifier to each worker
    - The rank is an integer value, from 0 to n–1 workers
- **`void MPI::Finalize()`**
  - Close and clean up all MPI states
  - After this function, no other MPI functions (even `MPI::Init()`) can be called
  - The user MUST ensure that all pending communications involving a worker complete before the finalization

Communicators

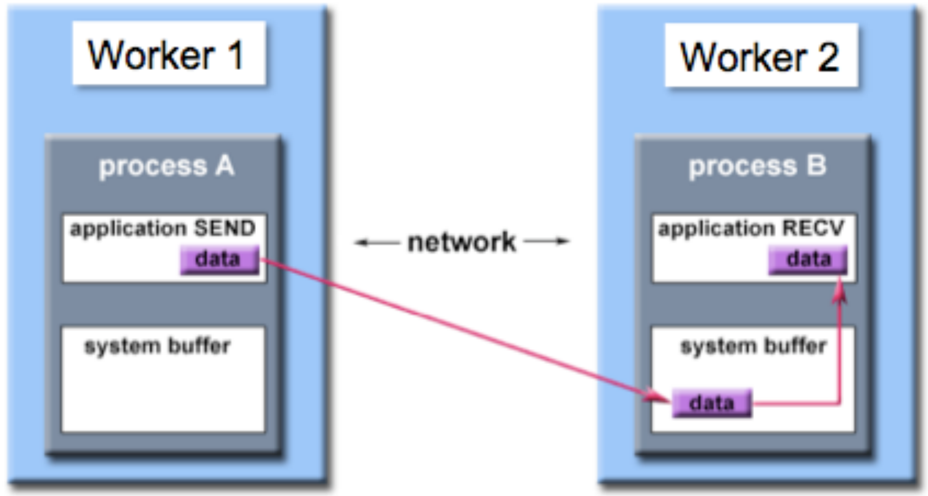
- The communicator is the basic MPI object which connects groups of workers in the MPI session
  - **`MPI::COMM_WORLD`** is the global communicator which collects all workers, declared by the **`MPI::Init()`**
  - Within each communicator each contained worker has an independent identifier and the contained workers are arranged in a topology
- In general, MPI functions must specify their communicator
  - **`int MPI::Comm::Get_rank()`**: gives the identifier of the worker
  - **`int MPI::Comm::Get_size()`**: gives the total number of workers
- Different communicators can be defined inside an MPI session, with different topologies and subset of workers
  - Useful for specific operations with regards a set of workers

In this lecture we will use only **`MPI::COMM_WORLD`**

MPI COMMUNICATIONS

Blocking/non-blocking communications

- Blocking functions will only "return" after the data is safely delivered (from a send to a receive)
- They require synchronization between send and receive:
  - A blocking send can be asynchronous if a **system buffer** is used to hold the data for eventual delivery to the receive
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program (must be synchronous)



Blocking/non-blocking communications

- Blocking communications are used for programs where there is a good load balance between workers
  - Speed-up based on the computation to communication ratio

- Non-blocking functions will “return” almost immediately, without any synchronization
  - can be unsafe in case of multiple communications
  - primarily used to overlap computation with communication and exploit possible performance gains
  - Not described in this lecture (see backup slides for more details)

Point-to-Point communication functions

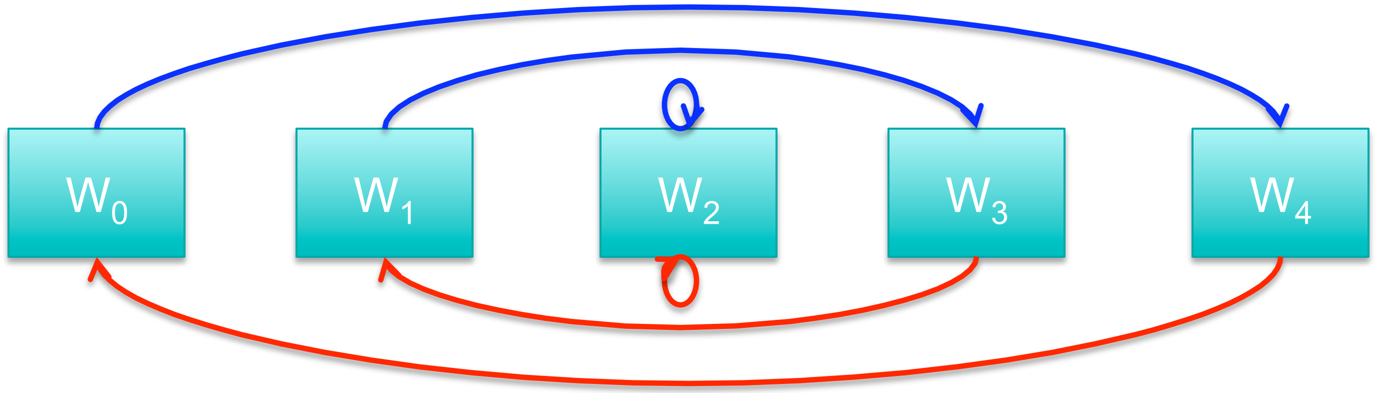
- Blocking asynchronous send/receive
  - `void MPI::COMM_WORLD.Send(buffer,count,datatype,dest,tag)`
  - `void MPI::COMM_WORLD.Recv(buffer,count,datatype,source,tag)`
- Parameters
  - `const void*` **`buffer`**: local variable in the worker used for the communication. It can be a vector (e.g. `int buffer[10]`)
  - `const Datatype&` **`datatype`**: basic element type of `buffer`
    - `MPI::CHAR`, `MPI::INT`, `MPI::FLOAT`, `MPI::DOUBLE`,…
  - `int` **`count`**: number of basic elements to move, i.e. dimension of `buffer` (e.g. for `int buffer[10]`, `count` is 10)
  - `int` **`dest`**`/`**`source`**: ID of destination/source worker for send/receive
  - `int` **`tag`**: Arbitrary non-negative integer assigned to uniquely identify a message. Send/receive operations should match message tags. For a receive operation, the wild card `MPI::ANY_TAG` can be used to receive any message regardless of its tag

Example: simple exchange of values

For ***n*** Workers:

- **Worker 0**: send to Worker *n* – 1, receive from Worker *n* – 1
- **Worker 1**: send to Worker *n* – 2, receive from Worker *n* – 2
- …
- **Worker *n* – 2**: send to Worker 1, receive from Worker 1
- **Worker *n* – 1**: send to Worker 0, receive from Worker 0

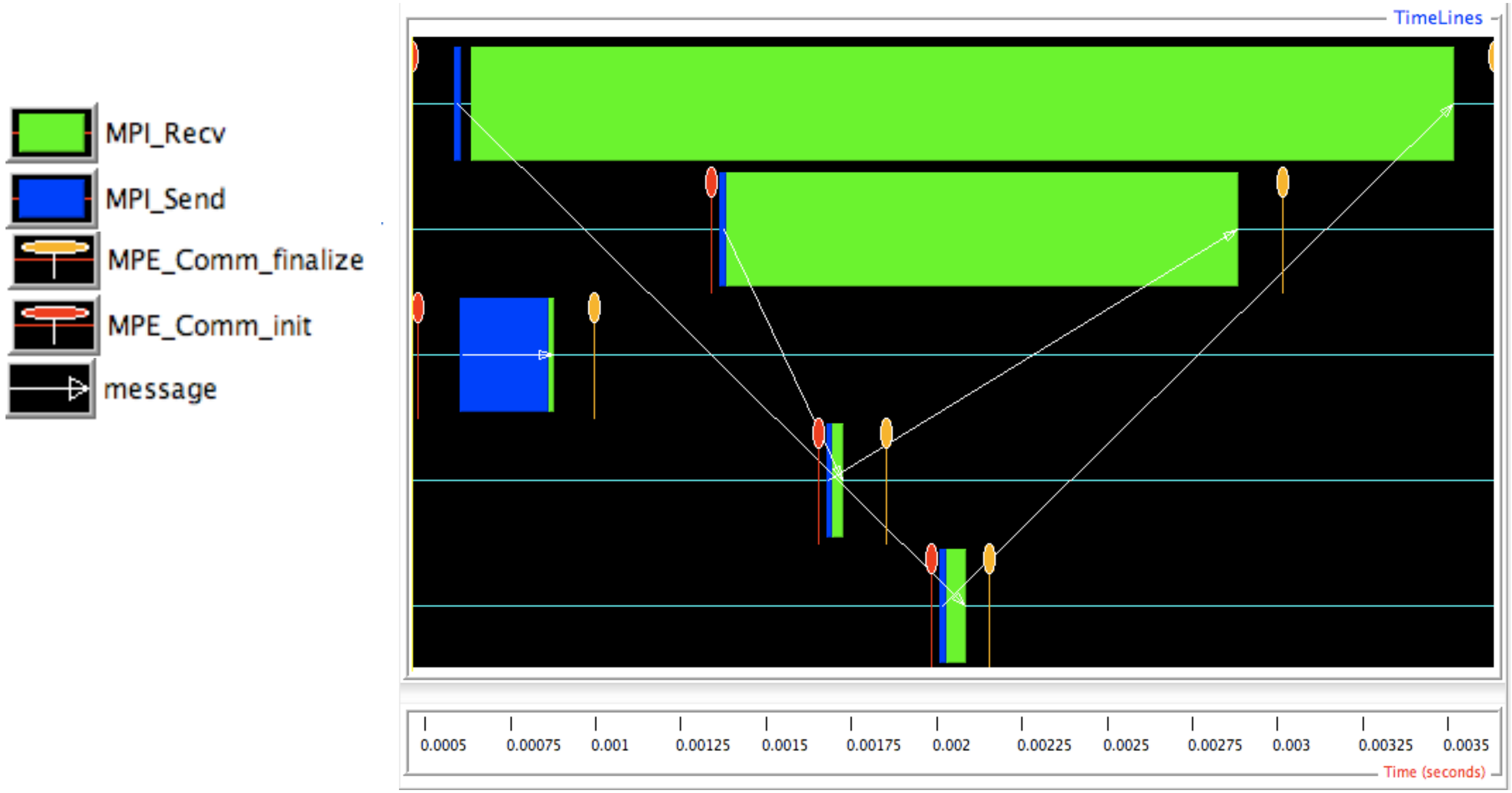
Example: 5 workers



```
#include "mpi.h"
#include <iostream>

int main(int argc, char *argv[])
{
  MPI::Init();
  int workerID = MPI::COMM_WORLD.Get_rank();
  int nWorkers = MPI::COMM_WORLD.Get_size();

  unsigned int tag(0);
  int sBuffer = workerID+1000; // value to send
  int rBuffer; // value to receive
  int destWorkerID = nWorkers-workerID-1;

  MPI::COMM_WORLD.Send(&sBuffer,1,MPI::INT,destWorkerID,tag);
  MPI::COMM_WORLD.Recv(&rBuffer,1,MPI::INT,destWorkerID,tag);

  std::cout << "I'm the worker " << workerID << "/" << nWorkers << ". "
            << "Sending " << sBuffer << " to worker " << destWorkerID << ". "
            << "Receiving " << rBuffer << " from worker "
            << destWorkerID << "." << std::endl;

  MPI::Finalize();

  return 0;
}
```

Example: simple exchange of values

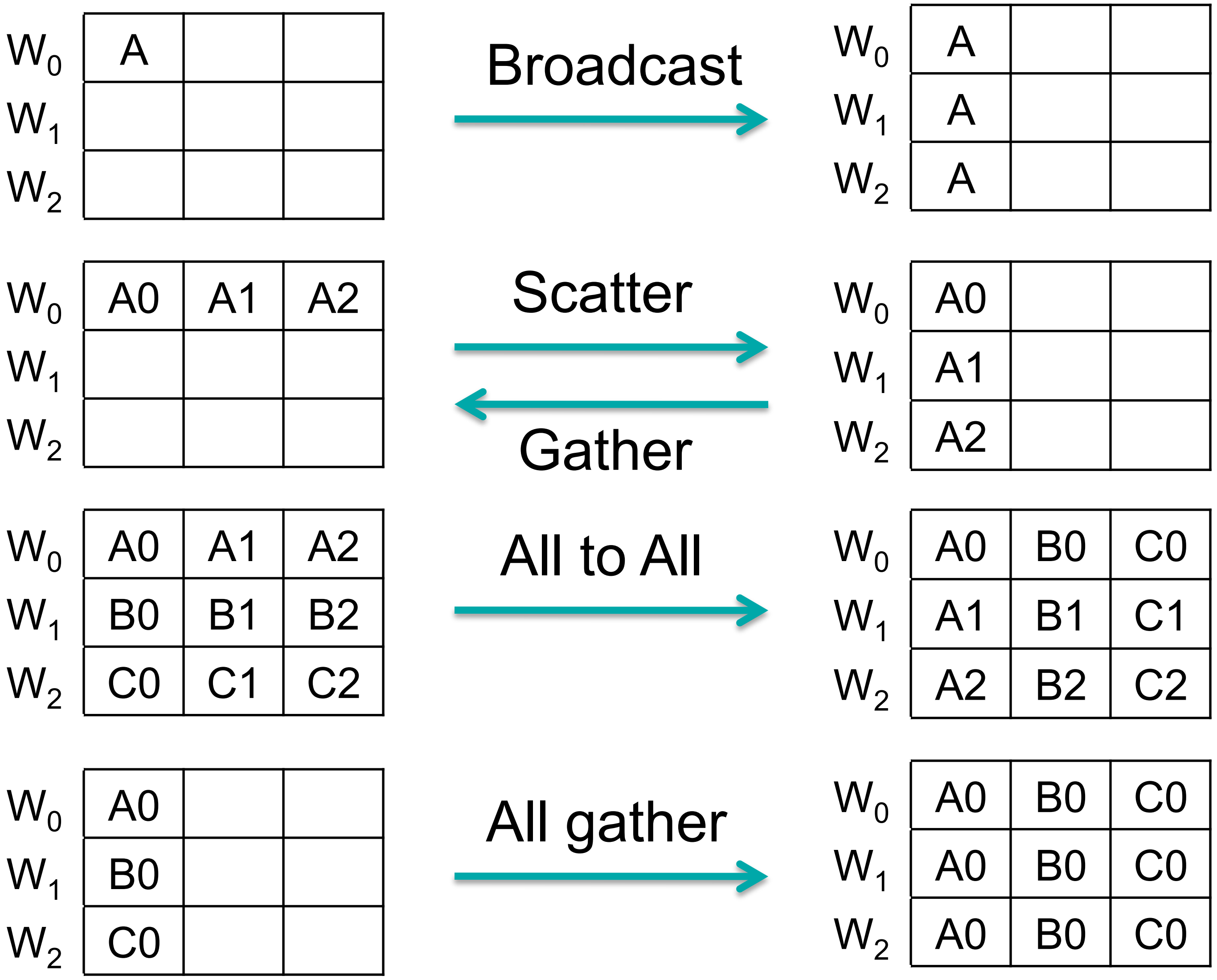
- Using the MPE library:
  http://www.mcs.anl.gov/research/projects/perfvis/



Collective communications

- Involve communication between all processes in a specific communicator (I omit `MPI::COMM_WORLD.` before the function, i.e. `MPI::COMM_WORLD.Bcast`)
  - **`Bcast`**: takes same data from one specific node (root) and sends that message to all processes (broadcast)
  - **`Reduce`**: takes data from all processes, performs a user-chosen operation, and store the results on one individual node
  - **`Scatter`**: distributes distinct messages from a root to each processes in the group
  - **`Gather`**: Gathers distinct messages from each process in the group to a root (inverse of Scatter operation)
  - "All" operations: **`Allreduce`**, **`Alltoall`**, **`Allgather`**
- Only blocking communications with synchronization
  - Do not take message tag arguments
- Optimized, involving far less function calls

Collective communications

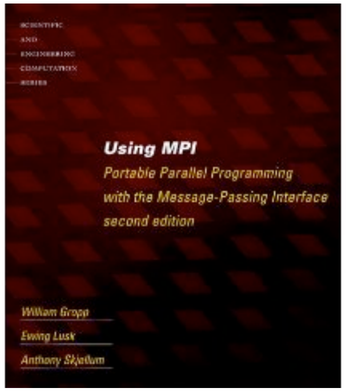


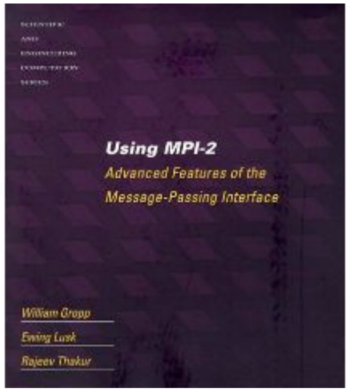
```
#include "mpi.h"
#include <iostream>

int main(int argc, char* argv[])
{
  const int DIM  = 3; // matrix and vector dimension
  const int ROOT = 0; // ROOT index (master)

  int A[DIM][DIM] = {0}, b[DIM] = {0};

  MPI::Init();
  int myID = MPI::COMM_WORLD.Get_rank();

  if (myID==ROOT)  // Fill the vector, only by the root
    for (int i = 0; i<DIM; i++)
      b[i] = DIM-i; // some calculation

  // Broadcast the vector from ROOT to all workers
  MPI::COMM_WORLD.Bcast(b,DIM,MPI::INT,ROOT);

  // Output of the vector from each worker
  // skip...
  // Do some calculations...
  int sum = 0; // local value
  for (int i = 0; i<DIM; i++) {
    b[i] *= myID+1; // change the local values of b
    sum += b[i];
  }

  // Make the reduce, results only in root
  int max(-1);
  MPI::COMM_WORLD.Reduce(&sum,&max,1,MPI::INT,
                         MPI::MAX,ROOT);

  // Insert all vectors in the matrix of each worker
  MPI::COMM_WORLD.Allgather(b,DIM,MPI::INT,
                            A,DIM,MPI::INT);

  // Output of max and the matrix from each worker
  // skip...

  MPI::Finalize();
  return 0;
}
```

Complex MPI functions

- If you want to do something complicated, take a look in the MPI references. You can find a specific MPI function which does the work for you (doing specific optimization of the code)
  - Essentially most of the functions that I didn't mention in this lecture are optimized combinations of basic functions
- Full lists at:
  - MPI-1: http://www.mpi-forum.org/docs/mpi-11-html/node182.html
  - MPI-2: http://www.mpi-forum.org/docs/mpi-20-html/node306.html

References


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES
Using MPI
Portable Parallel Programming with the Message-Passing Interface
second edition
William Gropp
Ewing Lusk
Anthony Skjellum


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES
Using MPI-2
Advanced Features of the Message-Passing Interface
William Gropp
Ewing Lusk
Rajeev Thakur

- Books:
  - "Using MPI", Gropp, Lusk and Skjellum, http://www.amazon.com/Using-MPI-Programming-Engineering-Computation/dp/0262571323
  - "Using MPI-2", Gropp, Lusk and Thakur, http://www.amazon.com/Using-MPI-2-Scientific-Engineering-Computation/dp/0262571331
- Online tutorials:
  - http://www.llnl.gov/computing/tutorials/mpi/
  - http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiintro/index.htm
  - http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html

Backup Slides

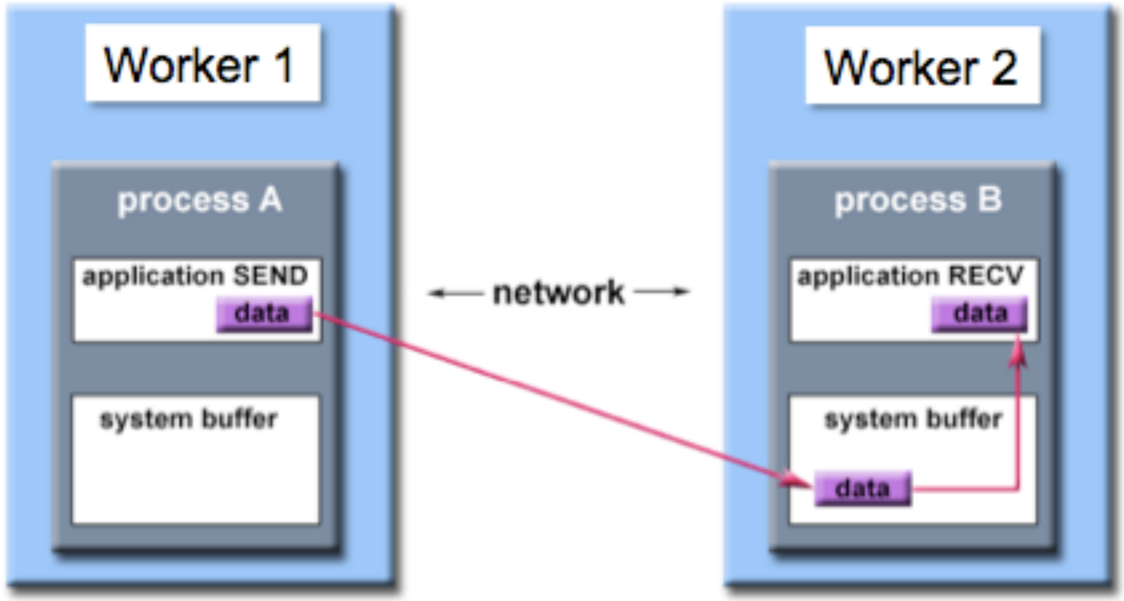
- C++/C/Fortran Syntax differences
- Examples of MPI functions implementation
- Non-blocking communications
- Libraries based on MPI
- Debugging
- Profiling

C++/C/Fortran Syntax differences

- Example: the function for the MPI initialization
  - C++: `void MPI::Init(int& argc, char**& argv)`
  - C: `int MPI_Init(int *argc, char ***argv)`
  - Fortran: `call MPI_INIT(ierror)`
- Note:
  - In case of error in each MPI function:
    - C++ throw an exception
    - C return value of the function is reserved for the error
    - Fortran requires a specific parameter for the error value
  - C++ uses the namespace `MPI::`, C and Fortran do not
  - C/C++ names are case sensitive, Fortran names are not

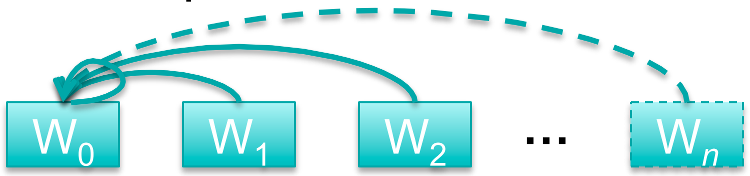
Examples of MPI functions implementation

- Data buffering:
  - Each send operation must match a receive operation, usually with some sort of synchronization
  - But what happens if the two tasks are out of synchronization?
    - Typically, a system buffer area is reserved to hold data in transit
    - **Not specified by the standard**, but from the particular MPI implementation

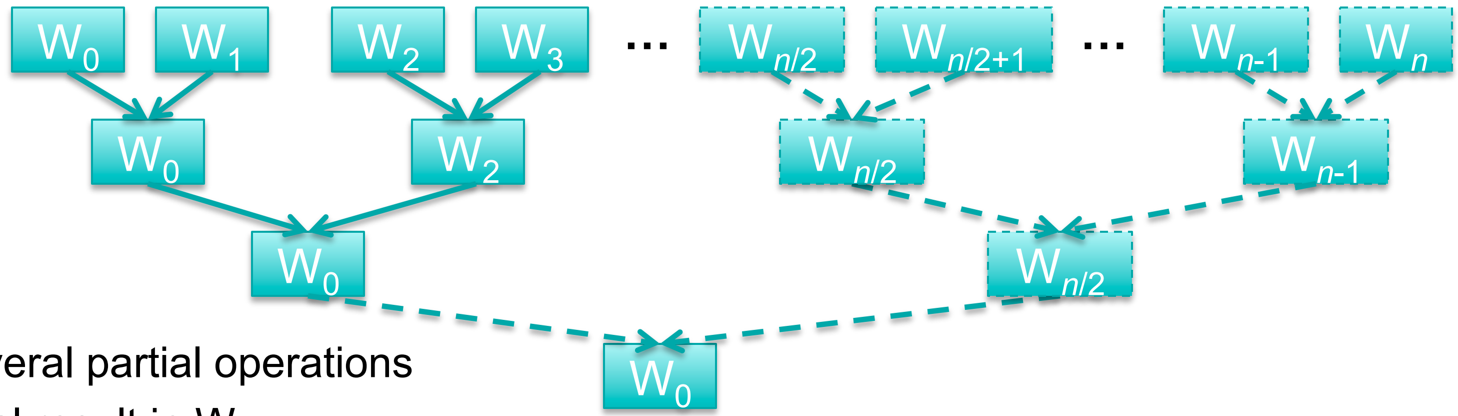


Examples of MPI functions implementation

- Collective Computation (reductions)
  - One worker of the group collects data from the other workers and performs an operation (add, multiply, etc.) on that data
  - MPI provides a particular function for that: MPI::Reduce
  - Different possible implementations, for examples:
    - One worker reduce
      - All workers send their data to W_0
      - Only W_0 does the reduction operation



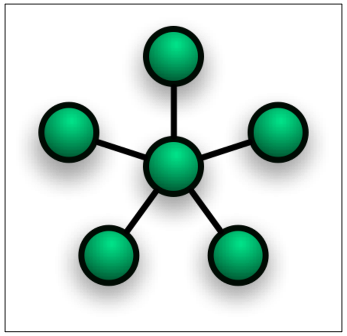
    - Tree-based reduce



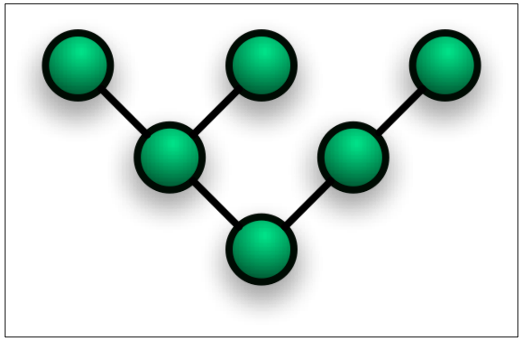
      - Several partial operations
      - Final result in W_0

Examples of MPI functions implementation

- Best performance on the reduction depends on the hardware, for example the topology of the network
  - Examples:



Star network topology:
good for one worker reduce



Tree network topology:
good tree-based reduce

- Note: Possible different results due to rounding

Bottom Line: be aware of your MPI implementation!

- Other details at https://computing.llnl.gov/tutorials/mpi_performance/

Non-blocking communications

- Send and receive will “return” almost immediately
  - Basically do not wait for any communication to complete
  - Communications will be completed when possible – user can not predict when that will happen
- Non-blocking communications can be unsafe in case of multiple communications
  - MPI guarantees that messages will not overtake each other (order is respected)
- Only point-to-point communications can be non-blocking
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Non-blocking communications

- Send/receive functions:
  - `void MPI::COMM_WORLD.`**`Isend`**`(`**`buffer,count,datatype,dest,tag`**`)`
  - `void MPI::COMM_WORLD.`**`Irecv`**`(`**`buffer,count,datatype,source,tag`**`)`
- Parameters (same as blocking functions)
  - `const void*` **`buffer`**: local variable in the worker used for the communication. It can be a vector (e.g. `int buffer[10]`)
  - `const Datatype&` **`datatype`**: basic element type of `buffer`
    - `MPI::CHAR`, `MPI::INT`, `MPI::FLOAT`, `MPI::DOUBLE`,…
  - `int` **`count`**: number of basic elements to move, i.e. dimension of `buffer` (e.g. for `int buffer[10]`, `count` is 10)
  - `int` **`dest`**`/`**`source`**: ID of destination/source worker for send/receive
  - `int` **`tag`**: Arbitrary non-negative integer assigned to uniquely identify a message. Send/receive operations should match message tags. For a receive operation, the wild card `MPI::ANY_TAG` can be used to receive any message regardless of its tag

Libraries based on MPI

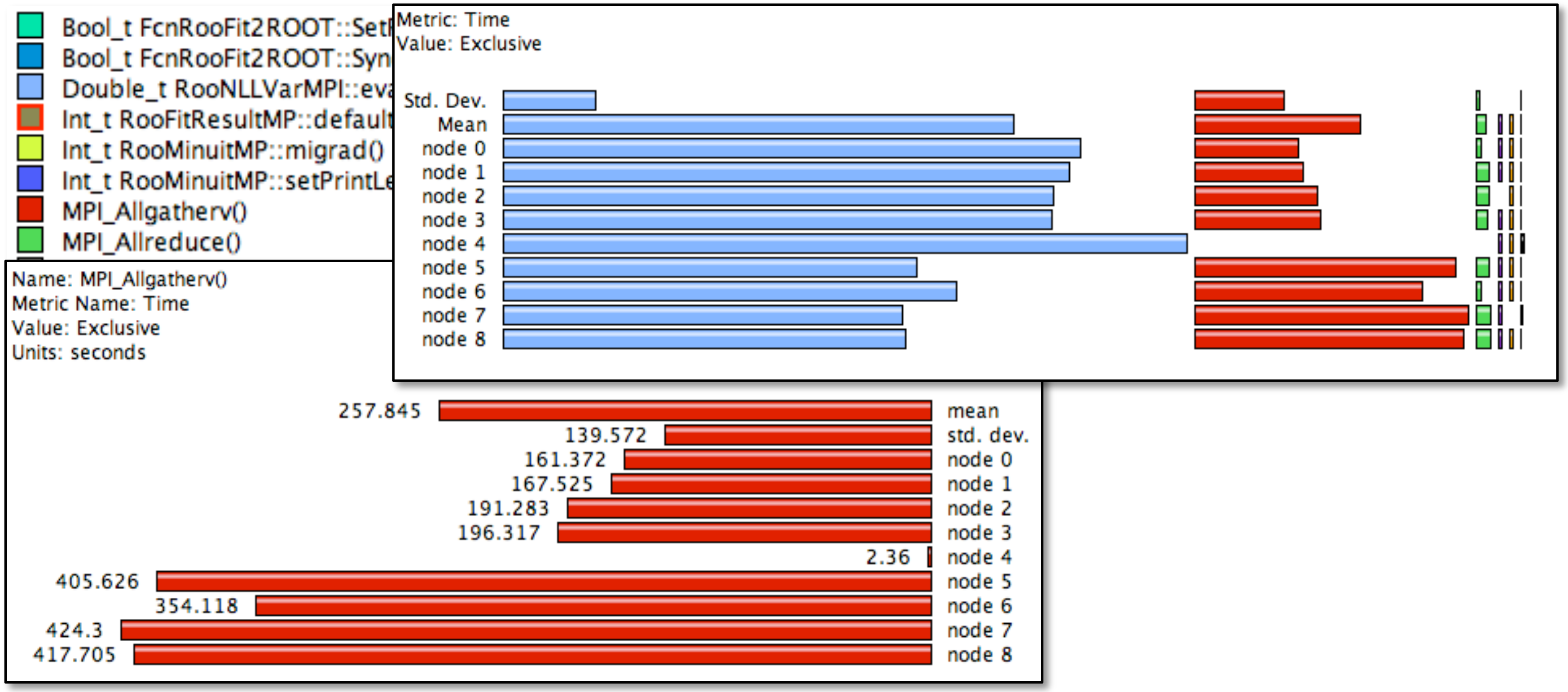
- There are different libraries, communally used in HPC, which are based on MPI
  - ScaLAPACK: http://www.netlib.org/scalapack
    - Scalable package based on **LAPACK** (Linear Algebra PACKage)
    - Routines for numerical algebra, such as solution of linear systems of equations, matrix inversion, full-rank linear least squares problems
  - SPRNG: http://sprng.cs.fsu.edu/
    - Scalable package for parallel pseudo random number generation
    - This library optimize the random generation in parallel, for example for Monte Carlo studies

Debugging

- Debugging is a pain for a sequential application, even more complicated for a parallel shared-memory application, and really a pain for distributed-memory application…
  - TotalView: http://www.totalviewtech.com
    - commercial-grade portable debugger for parallel and multithreaded programs.
    - Debugging even if you are running on multiple machines
    - Tutorial: https://computing.llnl.gov/tutorials/totalview/
  - The OpenMPI site has a great FAQ on MPI debugging
    - http://www.open-mpi.org/faq/?category=debugging

Profiling

- A good tool for profiling is TAU
  - http://www.cs.uoregon.edu/research/tau/home.php
- It provides several GUI applications to see speed-up and scalability



- Other details at https://computing.llnl.gov/tutorials/performance_tools/