HELP US ALL STAY HEALTHY

SIX SIMPLE TIPS

Maintain 1.5 metres distance between yourself and others where possible

Cough and sneeze into your elbow or a tissue (not your hands)

Avoid shaking hands

Put used tissues in the bin

Wash hands with soap and warm water or use an alcohol-based hand sanitiser after you cough or sneeze

Do not touch your face

IF YOU ARE UNWELL AND WORRIED ABOUT COVID-19:

- Call the National Coronavirus Helpline: 1800 020 080
- Call your usual GP for advice
- Call the UWA Medical Centre for advice: 6488 2118

UWA FAQs:
uwa.edu.au/coronavirus

Report COVID-19 hazards and suspected/confirmed cases via RiskWare: **uwa.edu.au/riskware**

THE UNIVERSITY OF WESTERN AUSTRALIA

DCS:507757521

THE UNIVERSITY OF WESTERN AUSTRALIA

High-Performance Computing

Lecture 9 MPI + OpenMP, and MPI Review

CITS5507

Zeyi Wen

Computer Science and Software Engineering

School of Maths, Physics and Computing

Acknowledgement: The lecture slides are adapted from many online sources.

A Problem of Computing Pi

THE UNIVERSITY OF WESTERN AUSTRALIA

```
step = 1 / (double)numSteps;

// parallel
#pragma omp parallel firstprivate(x,sum)
{
    #pragma omp for
    for (i = 0; i < numSteps; i++)
    {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    sums[omp_get_thread_num()]=sum;//1
}
for (int j = 0; j < 4; j++)
{
    printf("%f\n",sums[j]);
    pi += sums[j];
}
```

```
// serial
for (i = 0; i < numSteps; i++)
{
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}
pi = step * sum;
```

```
Someone@DESKTOP ~/vscode/openmp_lab
$ ./paraPi
pi=3.14159265358987571659099558019
   07926797866820000000

Time: 0.006559
```

```
Someone@DESKTOP ~/vscode/openmp_lab
$ ./seqPi
pi=3.14159265358976425019932321447
   41341471672060000000

Time: 0.046145
```

Why the Pi of parallel and serial algorithms are different?

Float Point

THE UNIVERSITY OF WESTERN AUSTRALIA

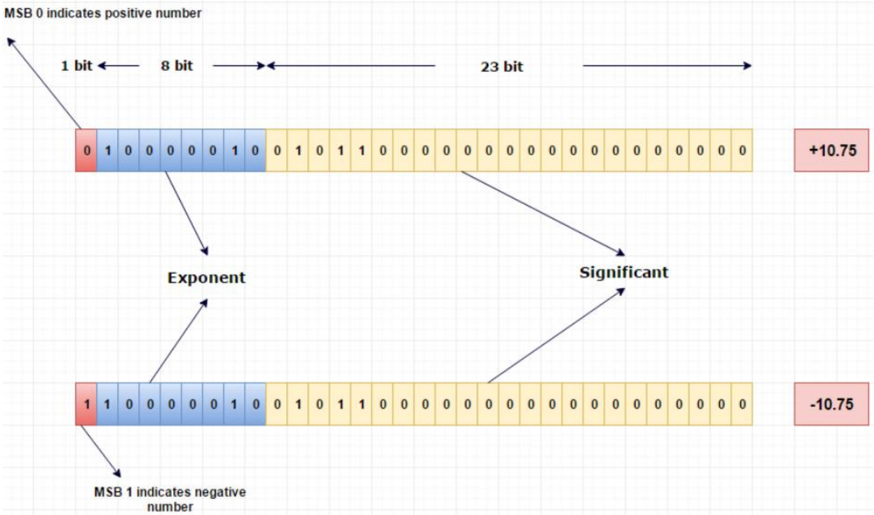
Float (32bit):

$$10000.01 = +1.000001 * 2^4$$

- Any number can be expressed in scientific notation
- “Float” are based on it to store numbers.
- It divides four bytes (32 bits) into three parts:

Sign (1 bit): 0 means positive and 1 means negative (e.g. +)
Exponent (8 bits): storing Exponent data and using shift storage (e.g. 4)
Significant (23 bits): Mantissa (e.g. 1.00001)

Float Point



Inaccurate Float Number

The reason why the two Pi values are different (i.e. the inaccuracy of floating point calculations)**:**

- Certain floating point numbers cannot be accurately represented in finite binary scientific notation, like 0.9.
- When calculating, if two floating-point numbers have different exponent, the exponent will be unified first, which may result in a loss of accuracy.

Outline

- **Hybrid Programming**
  - ✓ Introduction to MPI + Thread
  - ✓ Types of MPI Calls among Threads
  - ✓ MPI's Four Levels of Thread Safety
  - ✓ MPI_THREAD_MULTIPLE Challenge
  - ✓ Hybrid Programming with Shared Memory
  - ✓ Summary
- Review of MPI

Process and Thread (Lecture 1)

- A process can be considered as an independent execution environment in a computer system.
- There are usually many processes in a system at any time, each with its own memory space.
- Each process executes a sequence of instructions (the machine language program).
- Threads are also independent execution environments, but with a shared memory space (or address space).

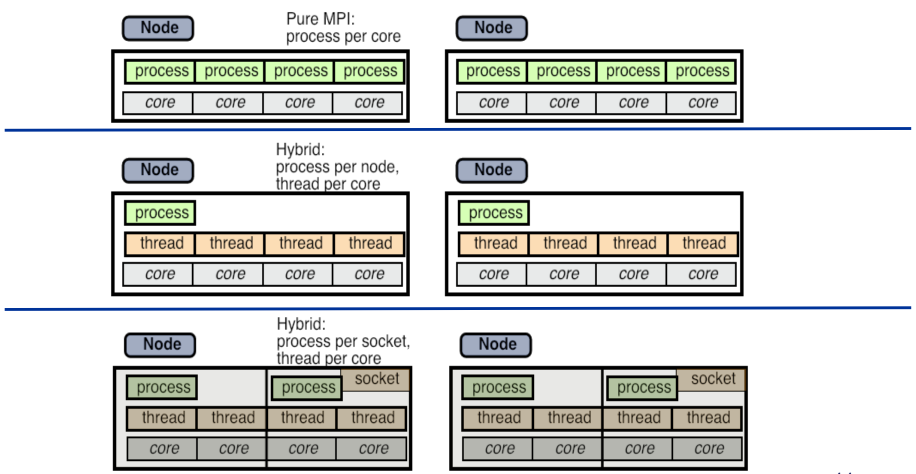
MPI and Threads

- MPI describes parallelism between processes (with **separate** address spaces)
- Thread parallelism provides a **shared-memory** model within a process
- OpenMP and Pthreads (**POSIX Threads**) are common models
  - ✓ OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - ✓ Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the developer.

Programming for Multicore

- A few options exist for programming multicore clusters
- All MPI
  - ✓ MPI between processes both within a node and across nodes
- MPI + OpenMP
  - ✓ Use OpenMP within a node and MPI across nodes
- MPI + Pthreads
  - ✓ Use Pthreads within a node and MPI across nodes

Hybrid Programming with MPI+Threads

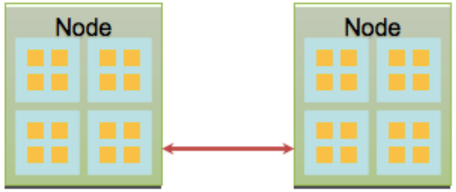


Hybrid Programming with MPI+Threads

- In MPI-only programming, each MPI process has a **“single thread”**
- In MPI+threads hybrid programming, there can be **multiple threads** executing simultaneously
  - ✓ All threads share all MPI objects (communicators, requests)
  - ✓ The MPI implementation may need to take **precautions** to make sure the state of the MPI implementation is consistent

Types of MPI Calls among Threads

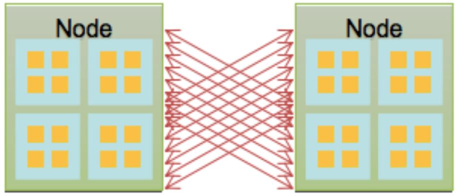
- **Single-threaded messaging**
  - ✓ Call MPI from a serial region
  - ✓ Call MPI from a single thread within a parallel region



Rank to rank

Types of MPI Calls among Threads

- **Multi-threaded messaging**
  - ✓ Call MPI from multiple threads within a parallel region
  - ✓ Requires an implementation of MPI that is thread-safe



rank-thread ID to rank-thread ID

Outline

- Hybrid Programming
  - ✓ Introduction to MPI + Thread
  - ✓ Types of MPI Calls among Threads
  - ✓ **MPI's Four Levels of Thread Safety**
  - ✓ MPI_THREAD_MULTIPLE Challenge
  - ✓ Hybrid Programming with Shared Memory
  - ✓ Summary
- Review of MPI

MPI's Four Levels of Thread Safety

- MPI defines **four** levels of thread safety—these are commitments the application makes to the MPI
  - ✓ **MPI_THREAD_SINGLE**: only one thread exists in the application
  - ✓ **MPI_THREAD_FUNNELED**: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init_thread)
  - ✓ **MPI_THREAD_SERIALIZED**: multithreaded, but only one thread at a time makes MPI calls
  - ✓ **MPI_THREAD_MULTIPLE**: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid **races**)

MPI_THREAD_SINGLE

- There are no threads in the system
  - ✓ There are **no** OpenMP parallel regions

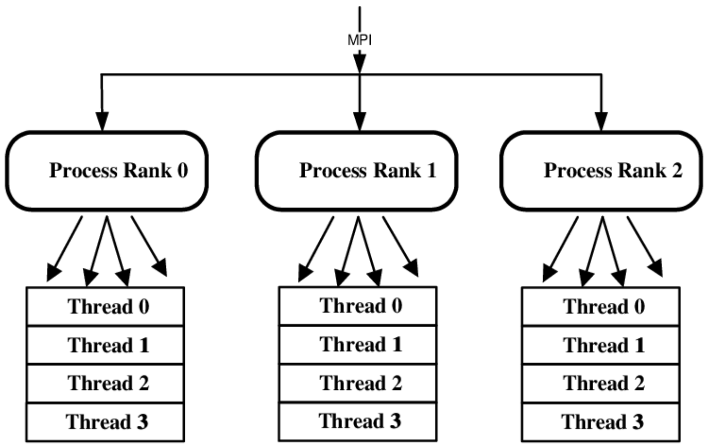
```
int main(int argc, char ** argv)
{
  int buf[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (i = 0; i < 100; i++){
    compute(buf[i]);
  }
/* Do MPI stuff */
  MPI_Finalize();
  return 0;
}
```


MPI_THREAD_FUNNELED

- All MPI calls are made by the master thread
  - ✓ Outside the OpenMP parallel regions
  - ✓ In OpenMP master regions

```
int main(int argc, char ** argv)
{
  int buf[100], provided;
  MPI_Init_thread(&argc, &argv,MPI_THREAD_FUNNELED, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel for
  for (i = 0; i < 100; i++){
    compute(buf[i]);
  }
/* Do MPI stuff */
  MPI_Finalize();
  return 0;
}
```


MPI_THREAD_FUNNELED



MPI_THREAD_FUNNELED (Example)

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

int main(int argc, char* argv[])
{
  int provided, rank, sum = 0;
  MPI_Init_thread(&argc, &argv,MPI_THREAD_FUNNELED, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < 101; i++){
     sum += i;
  }

  MPI_Bcast(&sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("[%d]: After Bcast, sum is %d\n", rank, sum);
  MPI_Finalize();
}
```

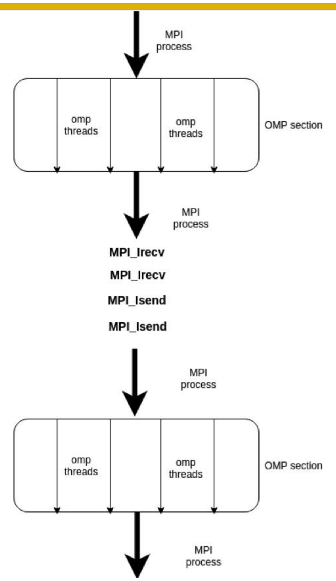
Note: mpicc -fopenmp Hello.c -o Hello

MPI_THREAD_SERIALIZED

- Only one thread can make MPI calls at a time
  - ✓ Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
  int buf[100], provided;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
     compute(buf[i]);
#pragma omp critical
/* Do MPI stuff */
   }
   MPI_Finalize();
   return 0;
}
```


MPI_THREAD_SERIALIZED (Continued)



MPI_THREAD_SERIALIZED (Example)

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
# define ProN 2
# define ThrN 2
int main(int argc, char *argv[])
{
  int provided, rank, sum = 0;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Status status;
  int thread_num[2];
  omp_set_num_threads(ThrN);
#pragma omp parallel
{
#pragma omp for reduction(+: sum)
   for (int i = 0; i < 101; i++)
   {
        sum += i;
   }
```


MPI_THREAD_SERIALIZED (Example Cont'd)

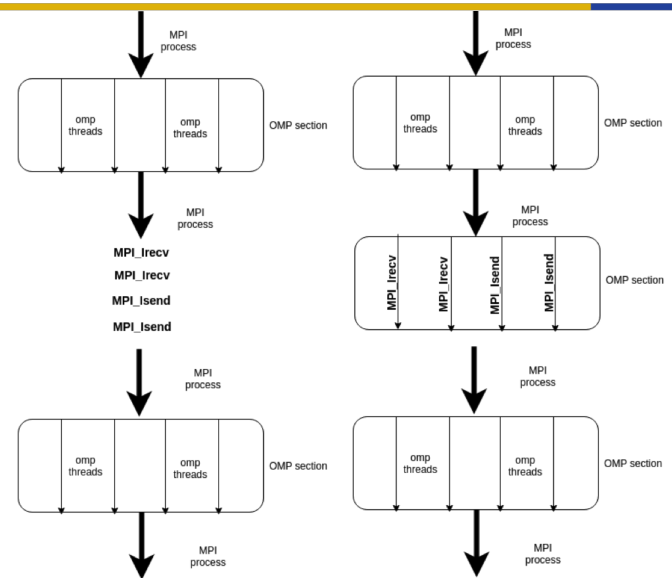
```
#pragma omp critical
  {
     if (rank == 0)
     {
        for (int i = 1; i < ProN; i++)
        {
          MPI_Recv(thread_num, 2, MPI_INT, i, omp_get_thread_num(),
MPI_COMM_WORLD, &status);
          printf("Message from node %d, thread %d\n", thread_num[0],
thread_num[1]);
        }
      }
      if (rank != 0)
      {
         thread_num[1] = omp_get_thread_num();
         thread_num[0] = rank;
         MPI_Send(&thread_num, 2, MPI_INT, 0, omp_get_thread_num(),
MPI_COMM_WORLD);
      }
   }
}
   MPI_Bcast(&sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
   printf("[%d]: After Bcast, sum is %d\n", rank, sum);
   MPI_Finalize();
}
```

MPI_THREAD_MULTIPLE

- Any thread can make MPI calls any time (restrictions apply)

```
int main(int argc, char ** argv)
{
  int buf[100], provided;
  MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
    compute(buf[i]);
/* Do MPI stuff */
  }
  MPI_Finalize();
  return 0;
}
```


MPI_THREAD_MULTIPLE (Continued)



MPI_THREAD_MULTIPLE (Aside)

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

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam = 0, np = 1;

  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

//omp_set_num_threads(4);

#pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out
of %d on %s\n",iam, np, rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```


MPI_THREAD_MULTIPLE (Aside)

- **Compile**
  - ✓ mpicc -fopenmp Hello.c -o Hello
- **Output**

```
Hello from thread 3 out of 4 from process 0 out of 2 on greydeAir.modem
Hello from thread 0 out of 4 from process 0 out of 2 on greydeAir.modem
Hello from thread 2 out of 4 from process 0 out of 2 on greydeAir.modem
Hello from thread 1 out of 4 from process 0 out of 2 on greydeAir.modem
Hello from thread 0 out of 4 from process 1 out of 2 on greydeAir.modem
Hello from thread 1 out of 4 from process 1 out of 2 on greydeAir.modem
Hello from thread 2 out of 4 from process 1 out of 2 on greydeAir.modem
Hello from thread 3 out of 4 from process 1 out of 2 on greydeAir.modem
```

Threads and MPI

- An MPI implementation is not required to support levels higher than **MPI_THREAD_SINGLE**; that is, an implementation is not required to be thread safe
- A fully thread-safe MPI implementation will support **MPI_THREAD_MULTIPLE**
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only **MPI_THREAD_SINGLE** is supported
- A threaded MPI program that does not **call MPI_Init_thread** is an incorrect program (common user error)

Specification of MPI_THREAD_MULTIPLE

- Ordering: When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - ✓ Ordering is maintained within each thread
  - ✓ User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
  - ✓ **E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator**
  - ✓ It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
  - ✓ **E.g., accessing an info object from one thread and freeing it from another thread**

Specification of MPI_THREAD_MULTIPLE

- Ordering: When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - ✓ Ordering is maintained within each thread
  - ✓ User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
  - ✓ …
- Blocking: Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

Ordering in MPI_THREAD_MULTIPLE

- **Incorrect Example with Collectives**

	Process 0	Process 1
Thread 1	**MPI_Bcast(comm)**	**MPI_Bcast(comm)**
Thread 2	**MPI_Barrier(comm)**	**MPI_Barrier(comm)**

Note: explanation of this example in the next slide.

Ordering in MPI_THREAD_MULTIPLE

- P0 and P1 can have different orderings of **Bcast** and **Barrier**
- Here the user must use some kind of synchronisation to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is **not valid** in MPI

	Process 0	Process 1
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

Ordering in MPI_THREAD_MULTIPLE

- **Incorrect Example with Object Management**

	Process 0	Process 1
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Comm_free(comm)	MPI_Comm_free(comm)

Note: explanation of this example in the next slide.

Ordering in MPI_THREAD_MULTIPLE

- The user has to make sure that one thread is not using an object while another thread is freeing it
  - ✓ This is essentially an ordering issue; the object might get freed before it is used

	Process 0	Process 1
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Comm_free(comm)	MPI_Comm_free(comm)

Blocking Calls in MPI_THREAD_MULTIPLE

- **Correct Example**

	Process 0	Process 1
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

Blocking Calls in MPI_THREAD_MULTIPLE

- An MPI implementation must ensure that this example never **deadlocks** for any ordering of thread execution
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

	Process 0	Process 1
Thread 1	**MPI_Recv(src=1)**	**MPI_Recv(src=0)**
Thread 2	**MPI_Send(dst=1)**	**MPI_Send(dst=0)**

Ordering in MPI_THREAD_MULTIPLE

- **Incorrect Example with Random Memory Access**

```
int main(int argc, char ** argv)
{
/* Initialize MPI and RMA window */
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
     target = rand();
     MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, 0, win);
     MPI_Put(..., win);
     MPI_Win_unlock(target, win);
  }
/* Free MPI and RMA window */
  return 0;
}
```

Different threads can lock the same process causing multiple locks to the same target before the first lock is unlocked. This becomes a serial execution.

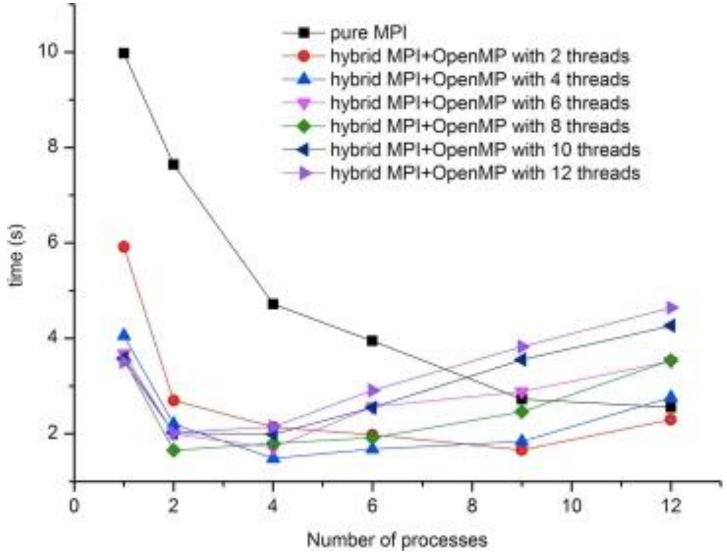
The Current Situation

- All MPI implementations support **MPI_THREAD_SINGLE.**
- Support **MPI_THREAD_FUNNELED**
  - ✓ Does require **thread-safe malloc**
  - ✓ Probably OK in OpenMP programs
- Many (but not all) implementations support **MPI_THREAD_MULTIPLE**
  - ✓ Hard to implement efficiently (lock granularity issue)
- "Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need **MPI_THREAD_FUNNELED**
  - ✓ So don't need "thread-safe" MPI for many hybrid programs

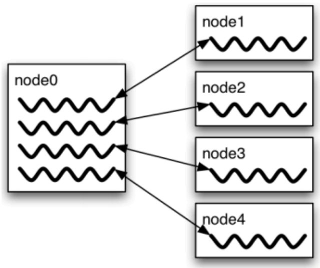
Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- To measure the performance impact, the figure in the next slide shows results on communication performance when using multiple threads versus multiple processes
  - ✓ For results, see Thakur/Gropp paper: "Test Suite for Evaluating Performance of Multithreaded MPI Communication,"(Parallel Computing, 2009)

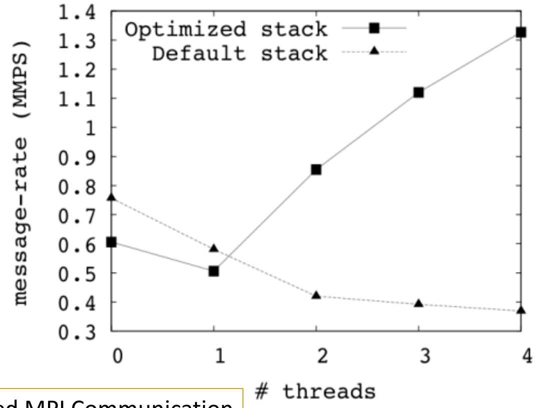
Performance with MPI_THREAD_MULTIPLE



Message Rate Results



Message Rate Benchmark



"Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems" EuroMPI 2010

Outline

- Hybrid Programming
  - ✓ Introduction to MPI + Thread
  - ✓ Types of MPI Calls among Threads
  - ✓ MPI's Four Levels of Thread Safety
  - ✓ **MPI_THREAD_MULTIPLE Challenging**
  - ✓ Hybrid Programming with Shared Memory
  - ✓ Summary
- Review of MPI

Why is it hard to optimise MPI_THREAD_MULTIPLE?

- MPI internally maintains several resources
- Because of MPI semantics, it is required that all threads have access to some of the data structures
  - ✓ **E.g. thread 1 can post an Irecv, and thread 2 can wait for its completion—thus the request queue has to be shared between both threads.**
  - ✓ Since multiple threads are accessing this shared queue, it needs to be **locked**—adds a lot of **overhead**

These are some issues in developing MPI implementations (e.g. OpenMPI).

Hybrid Programming: Correctness Requirements

- Hybrid programming with **MPI+threads** does not do much to reduce the complexity of thread programming
  - ✓ Your application still has to be a correct multi-threaded application
  - ✓ On top of that, you also need to make sure you are correctly following **MPI semantics**
- Many commercial debuggers offer support for debugging hybrid MPI+threads applications (mostly for **MPI+Pthreads** and **MPI+OpenMP**)

An Example Encountered

- The MPICH group received a bug report about a very simple multithreaded MPI program that hangs
- Run with 2 processes
- Each process has 2 threads
- Both threads communicate with threads on the other process as shown in the **next slide**
- Several hours spent trying to debug MPICH before discovering that the bug is actually in the user's program

Note: MPICH is an implementation of MPI; OpenMPI is another implementation.

An Example Encountered (Continued)

- 2 processes, 2 threads, each thread executes this code

```
for (j = 0; j < 2; j++) {
   if (rank == 1) {
     for (i = 0; i < 2; i++)
       MPI_Send(NULL, 0, MPI_CHAR, 0, 0,MPI_COMM_WORLD);
     for (i = 0; i < 2; i++)
       MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
   }
   else { /* rank == 0 */
   for (i = 0; i < 2; i++)
     MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
   for (i = 0; i < 2; i++)
     MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
   }
}
```


Intended Ordering of Operations

Rank 0		*Rank 1*	
2 recvs (T1) ⟷		2 sends (T1)	
2 sends (T1)		2 recvs (T1)	
2 recvs (T1)		2 sends (T1)	
2 sends (T1) ⟷		2 recvs (T1)	
	2 recvs (T2)		2 sends (T2)
	2 sends (T2)		2 recvs (T2)
	2 recvs (T2) ⟷		2 sends (T2)
	2 sends (T2)		2 recvs (T2)

- Every send matches a receive on the other rank

Possible Ordering of Operations

Rank 0		Rank 1	
2 recvs (T1) 2 sends (T1) 1 recv (T1)	 1 recv (T2)	2 sends (T1) 1 recv (T1)	 2 sends (T2) 1 recv (T2)
1 recv (T1)	1 recv (T2)	1 recv (T1)	1 recv (T2)
2 sends (T1)	2 sends (T2) 2 recvs (T2) 2 sends (T2)	2 sends (T1) 2 recvs (T1)	2 sends (T2) 2 recvs (T2)

- Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a **RECV** call

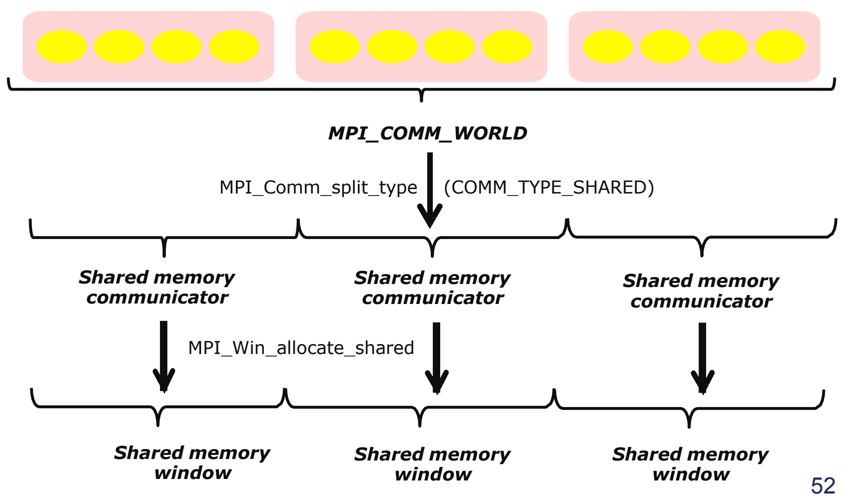
Outline

- Hybrid Programming
  - ✓ Introduction to MPI + Thread
  - ✓ Types of MPI Calls Among Threads
  - ✓ MPI's Four Levels of Thread Safety
  - ✓ MPI_THREAD_MULTIPLE Challenge
  - ✓ **Hybrid Programming with Shared Memory**
  - ✓ Summary
- Review of MPI

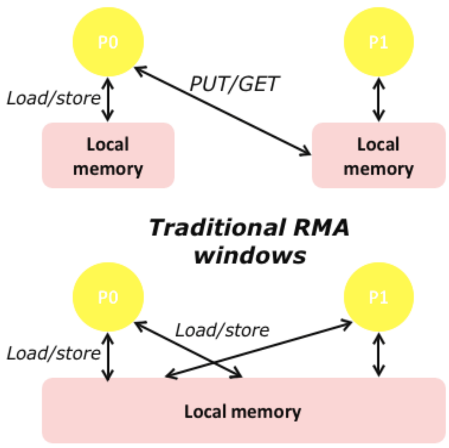
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - ✓ **MPI_Win_allocate_shared**
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronise access to shared memory regions
- Can be simpler to program than threads

Creating Shared Memory Regions in MPI



Regular RMA windows vs Shared memory windows



- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - ✓ E.g., x[100] = 10
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be useful when processes want to use threads only to get access to all of the memory on the node
  - ✓ You can create a shared memory window and put your shared data

Memory Allocation and Placement

- Shared memory allocation does not need to be uniform across processes
  - ✓ Processes can allocate a different amount of memory
- MPI standard doesn't specify where the memory to be placed (e.g. which physical memory it will be pinned to)
  - ✓ Implementations can choose their own strategies, though it is expected that an implementation will try to place shared memory allocated by a process "close to it"
- The total allocated shared memory on a communicator is contiguous by default
  - ✓ Users can pass an info hint called "noncontig" that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

Shared Arrays with Shared Memory Windows

```c
int main(int argc, char ** argv)
{
   int buf[100];
   MPI_Init(&argc, &argv);
   MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
   MPI_Win_allocate_shared(comm, ..., &win);
   MPI_Comm_rank(comm, &rank);
   MPI_Win_lockall(win);
/* copy data to local part of shared memory */
   MPI_Win_sync(win); /* or memory flush if available */
/* use shared memory */
   MPI_Win_unlock_all(win);
   MPI_Win_free(&win);
   MPI_Finalize();
   return 0;
}
```


Summary

- MPI + X a reasonable way to handle
  - ✓ Extreme parallelism
  - ✓ SMP nodes; other hierarchical memory architectures
- Many choices for X
  - ✓ OpenMP
  - ✓ Pthreads
  - ✓ MPI (using shared memory)

SMP: Symmetric Multi-Processing

Outline

- Hybrid Programming
  - ✓ Introduction to MPI + Thread
  - ✓ Types of MPI Calls Among Threads
  - ✓ MPI's Four Levels of Thread Safety
  - ✓ MPI_THREAD_MULTIPLE Challenge
  - ✓ Hybrid Programming with Shared Memory
  - ✓ Summary
- **Review of MPI**

What is MPI (Lecture 6)

- **M**essage **P**assing **I**nterface
- All machines run the same code
- Messages are sent between them to guide computation

- MPI is a standard not a library itself
  - ➢ OpenMPI, MPICH are libraries/implementations
- MPI is portable
- MPI can work with heterogenous clusters
- MPI code can work on various configurations of machines

Message Passing

- A process is a program in execution and has its own independent address space.

- Message passing is used for communication among processes.
  - o is about data transfer
  - o requires cooperation of sender and receiver
  - o cooperation not always apparent in code

- Inter-process communication
  - o Types: synchronous and asynchronous
  - o Movement of data from one process's address space to another's

Error Handling

- By default, an error causes all processes to **abort**.

- The user can have his/her own error handling routines.

- Some custom error handlers are available for downloading from the net.

Synchronous vs. Asynchronous

- A **synchronous** communication is not complete until the message has been received.
  - blocking send/receive in MPI
- An **asynchronous** communication completes as soon as the message is on the way.
  - non-blocking send/receive in MPI

Sending and Receiving Routines (Review)

Usage:

MPI_Send(**start**, **count**, **datatype**, **dest**, **tag**, **comm**)

- Message buffer described by
  - **Start**
  - **Count**
  - **Data types**
- Target process given by
  - **Dest**
  - **Comm**
- **Tag** can be used to create different 'types' of messages

Sending and Receiving Routines (Review)

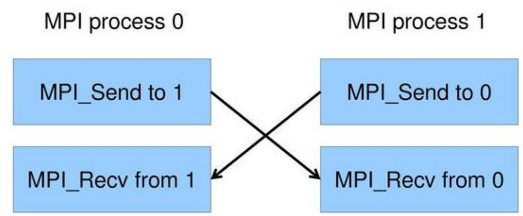
Usage:

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until a matching (source, tag) message is available
- Reads into the buffer
  - Start
  - Count
  - Datatype
- Target process specified by
  - Source
  - Comm
- Status contains more information
- Receiving fewer than count occurrences of datatype is okay, more is an error

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



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

Blocking vs Non-blocking

- Depending on the implementation you use this may cause a **deadlock**
  - ✓ If you have enough buffer space it might be okay (but don't rely on this)
- We have been using the blocking send/receive functions
  - ✓ Halt execution **until completed**
- There exist non-blocking versions of send/recv
  - ✓ **MPI_Isend** – Same arguments
  - ✓ **MPI_Irecv** – Same arguments but replace MPI_Status with MPI_Request
- Return **immediately** and continue with computation

MPI Program 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

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype)
- An MPI *datatype* is recursively defined as:
  - predefined data type (e.g. MPI_INT)
  - a contiguous array of MPI datatypes
  - a stridden 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 columnwise.

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very **different memory representations** and lengths of elementary datatypes (heterogeneous communication).
- Specifying **application-oriented** layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

MPI Datatype - Pack

```
MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)
```

- IN inbuf input buffer start (choice)
- IN incount number of input data items (non-negative integer)
- IN datatype datatype of each input data item (handle)
- OUT outbuf output buffer start (choice)
- IN outsize output buffer size, in bytes (non-negative integer)
- INOUT position current position in buffer, in bytes (integer)
- IN comm communicator for packed message (handle)

- Used by repeatedly calling MPI_PACK with changed inbuf and outbuf values

MPI Datatype - Unpack

```
MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
```

- IN inbuf Input buffer start (choice)
- IN insize size of input buffer, in bytes (non-negative integer)
- INOUT position current position in bytes (integer)
- OUT outbuf output buffer start (choice)
- IN outcount number of items to be unpacked (integer)
- IN datatype datatype of each output data item (handle)
- IN comm communicator for packed message (handle)

The exact inverse of MPI_PACK. Used by repeatedly calling unpack, extracting each subsequent element

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.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

Types of collective communication

Collective communication operations are made of the following types:

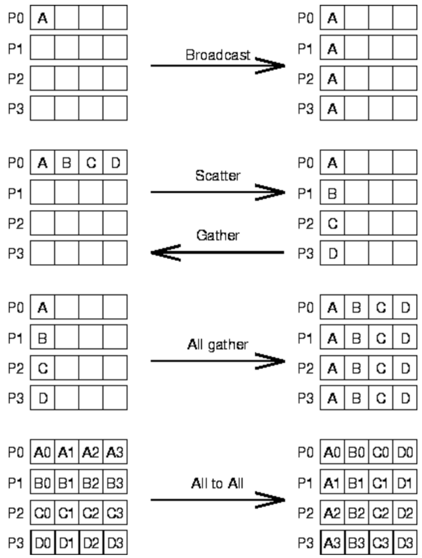
Barrier Synchronisation – Blocks until all processes have reached a synchronisation point

Data Movement (or **Global Communication**) – Broadcast, Scatters, Gather, All to All transmission of data across the communicator.

Collective Operations (or **Global Reduction**) – One process from the communicator collects data from each process and performs an operation on that data to compute a result.

Global Communication – Overview

THE UNIVERSITY OF WESTERN AUSTRALIA



What is Communicator?

THE UNIVERSITY OF WESTERN AUSTRALIA

- Many MPI users are only familiar with the communicator **MPI_COMM_WORLD**
- A communicator is a handle to a **group of processes**.
- A group is an ordered set of processes
  - Each process is associated with a **rank**
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines

Two Types of I/O in MPI

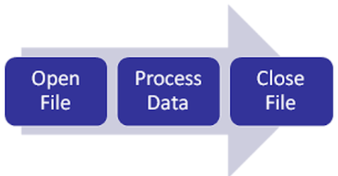
THE UNIVERSITY OF WESTERN AUSTRALIA

- MPI I/O supports **two** types of I/O
- Independent
  - ✓ Each process handles its **own** I/O
  - ✓ Supports derived datatypes (different to POSIX)
- Collective
  - ✓ I/O calls must be made by **all** processes
  - ✓ Used “shared file, all write”
  - ✓ Optimised by the MPI library

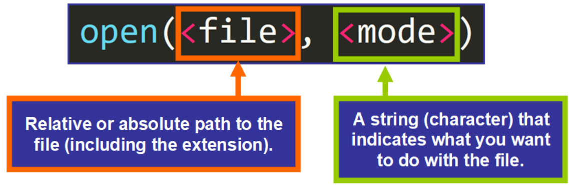
POSIX stands for Portable Operating System Interface, and is an IEEE standard designed to facilitate application portability.

Independent I/O

THE UNIVERSITY OF WESTERN AUSTRALIA

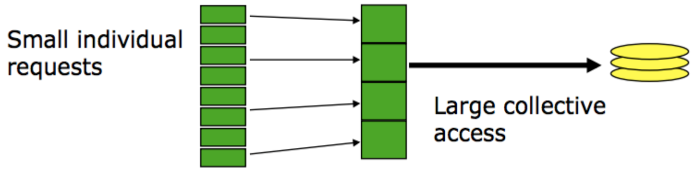


- Just like C/C++ I/O you need to
  - ✓ Open the file
  - ✓ Read/Write data from/to the file
  - ✓ Close the file
- In MPI, these steps are almost the same:
  - ✓ Open the file: **MPI_File_open**
  - ✓ Write to the file: **MPI_File_write**
  - ✓ Close the file: **MPI_File_close**



Collective I/O

- Provides **massive speedup**
- Like communication, **all processes** (in a communicator) need to call the same function
- Allows implementation to make many optimisations
- Basic idea:
  - ✓ **Building large blocks** of data, so reads/writes to the I/O system can be large
  - ✓ **Merging of requests** from different processes
  - ✓ Particularly effective with very non-contiguous but overlapping requests are interleaved



Why we need virtual topologies

- The processes needs to be mapped onto the hardware
  - ✓ Which is probably not a line of machines
  - ✓ Many different architectures
- Most numerical algorithms have some structure to their communication
- If we don't exploit the structure in an algorithm
  - ✓ Could get 'random' process assignment
  - ✓ Extra communication overhead
- Virtual topologies allow you to specify communication patterns, allowing MPI to make smarter mapping choices no matter what machine you use
- Easier to write programs

References

- Readings
  - [Hybrid Programming]
  - [False Sharing]
  - [MPI_Init() vs MPI_Init_thread()]

Copyright Notice