MPI and threading

CS 594 Spring 2006

Graham E Fagg

Threading

- Multi-threading can improve performance
  - Better CPU utilization
  - IO latency hiding
  - Simplified logic (letting threads block)
- Most useful on SMPs
  - Each thread can have its own CPU
- Overloading CPU's *can* be ok
  - Depends on application (e.g., latency hiding)
  - Even on uniprocessors

Threads and MPI

- Extend the threaded model to multi-level parallelism
  - Threads within an MPI process
  - Possibly spanning multiple processors
  - Allowing threads to block in communication

➔ Overlap communication and computation

Application Level Threading

- Freedom to use blocking MPI functions
  - Allow threads to block in MPI_SEND / MPI_RECV
  - Simplify application logic
- Separate communication and computation

Implementation Threading

- Asynchronous communication progress
  - Allow communication "in the background"
  - Even while no application threads in MPI
- Can help single-threaded user applications
  - Non-blocking communications can progress independent of application

Asynchronous Communication

		MPI implementation	
		One thread	Multiple threads
App	One thread	X	☺
	Multiple threads	X ☺	☺

What About "One Big Lock"?

- Put a mutex around MPI calls
  - Only allow one application thread in MPI at any given time
  - This allows a mutli-threaded application to utilize MPI
- Problem: can easily lead to deadlock
  - If multiple threads try to use MPI
  - Example
    - Thread 1 calls MPI_RECV
    - Thread 2 later calls matching MPI_SEND

Why Not Use Non-Blocking?

- Why not use MPI_ISEND? (and friends)
  - This has worked for years
  - MPI implementations already support it
  - Allows at least some degree of overlap
- Threads can allow simplicity of logic
  - Do not have to poll for MPI completion
  - Concurrency within application
  - Let threads block in MPI_SEND / MPI_RECV

Doesn't MPI Do This Already?

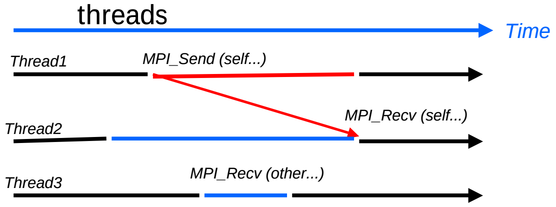
- MPI_SEND: Does it progress after return?
  - Example: in TCP, MPI typically calls write(2)
  - OS buffers and sends "in the background"
  - Do not get "true" progress (e.g., rendezvous)
- If the MPI implementation can use threads:
  - True asynchronous progress
  - Progress pending communications while application is outside of MPI

Threads and MPI

- MPI does not define if a MPI process is a thread or an OS process
  - Threads are not addressable
  - MPI_SEND(...thread_id...) is not possible
- MPI-2 Specification
  - Does not mandate thread support
  - Does define what a "Thread Compliant MPI" should do
  - Specifies 4 levels of thread support

Thread Compliant MPI

- All MPI library calls are thread safe
- Blocking calls block the calling thread only and allow progress on other threads

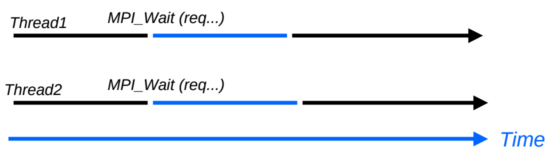


MPI Threading Rules

- MPI_INIT and MPI_FINALIZE should only be called once
  - Should only be called by a single thread
  - Both should be called by the same thread
  - Known as the **main thread**

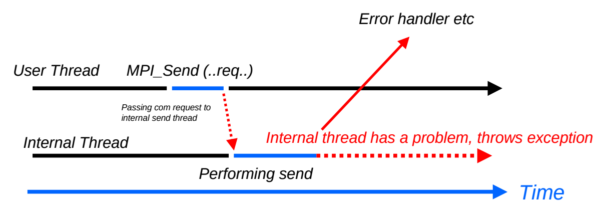
Threads and Requests

- Multiple threads should not attempt to complete the same request
- Erroneous example:



Threads and Exceptions

- Exception handlers can arise in a different thread context than the one making the MPI call

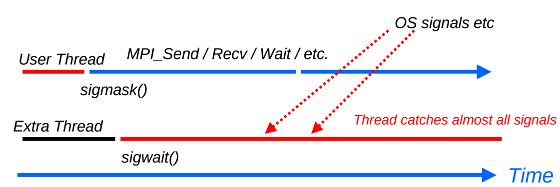


More Thread Rules

- Undefined behavior of MPI call when:
  - If a thread executes an MPI call that is cancelled by another thread
  - If a thread executes an MPI call and catches a signal
- How to deal with signals?

Avoiding Signal Problems

- Create extra thread that waits in **`sigwait()`**
- MPI threads mask signals



Threads and MPI

- Normally initialize MPI process with MPI_INIT
- Threaded MPI programs use
  - MPI_INIT_THREAD(argv, argv, **requested**, **provided**)
  - Tells MPI application threading requirements
  - Implementation informs application of what it can provide
- If implementation cannot support a requested thread level, it returns the highest level it can provide
  - This is not an error!

Threads and MPI

- Available levels of thread support
  - MPI_THREAD_SINGLE
  - MPI_THREAD_FUNNELED
  - MPI_THREAD_SERIALIZED
  - MPI_THREAD_MULTIPLE

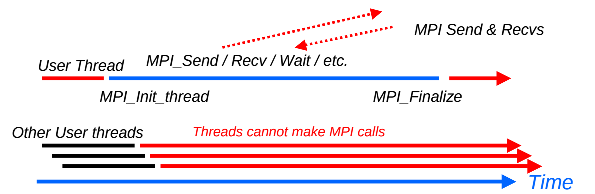
MPI_THREAD_SINGLE

- Application is NOT allowed to use threads
  - This allows an MPI implementation to avoid potentially expensive locking *
- Might cause problems / errors if the application actually does use threads

* Specification is unclear on if the Implementation can use threads

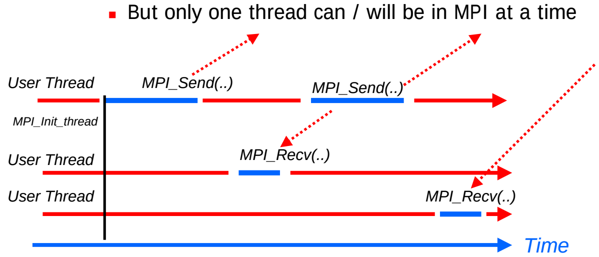
MPI_THREAD_FUNNELED

- The user application can be multi-threaded but only the main thread calls MPI functions



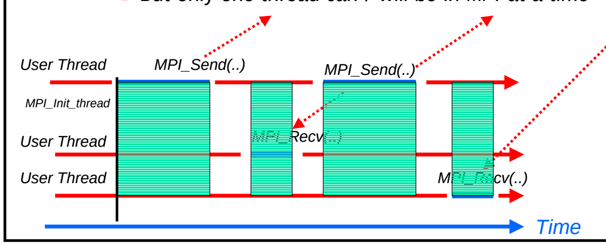
MPI_THREAD_SERIALIZED

- Users application is multi-threaded any thread can make MPI calls
  - But only one thread can / will be in MPI at a time



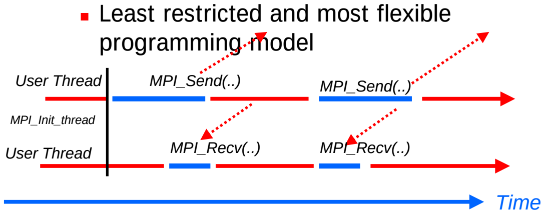
MPI_THREAD_SERIALIZED

- Application can be multi-threaded any thread can make MPI calls
  - But only one thread can / will be in MPI at a time



MPI_THREAD_MULTIPLE

- Application can be multi-threaded and any thread can make an MPI call at any time
  - Least restricted and most flexible programming model



Threads and MPI

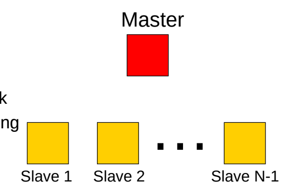
- MPI_QUERY_THREAD
  - Returns provided level of thread support
  - Useful if MPI_INIT was invoked (vs. MPI_INIT_THREAD)
  - ➔ Thread level may be set via environment variable!
- MPI_IS_THREAD_MAIN
  - Returns true if this is the thread that invoked MPI_INIT / MPI_INIT_THREAD

Threading Example

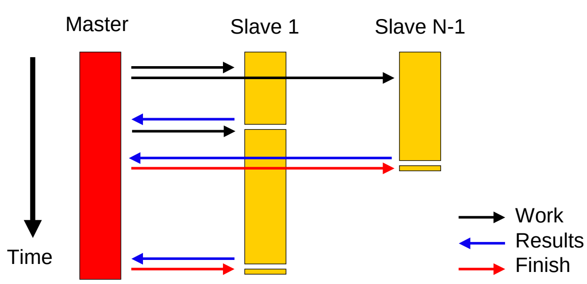
- Use a common master / slave framework
  - Master sends out work
  - Workers receive work, do work, return work
  - Loop until complete
- Show how threads can be beneficial in this scenario

Method 1: Pure Master / Slave

- Total of N processes
  - 1 Master process
  - (N-1) Slave processes
- Master
  - Send initial set of work
  - Loop receiving / sending
- Worker
  - Loop: receive, work, send



Pure Master / Slave



Application main()

```
MPI_Init(…);
MPI_Comm_rank(…, &rank);
if (rank == 0)
  do_master()
else
  do_slave()
MPI_Finalize()
```

Master Main Loop

```
for (i = 0; i < n; ++i)
    MPI_Send(work[i], …, slaves[i], …);
while (i < total_work) {
    MPI_Recv(answer, …, MPI_ANY_SOURCE, …);
    process_answer(answer);
    if (++i < total_work) {
        MPI_Send(work[i], …, slave[X], …);
    }else {
        MPI_Send(you_are_done, …,slave[X], …);
    }
}
```

Slave Main Loop

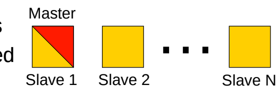
```
while (1) {
    MPI_Recv(work, …);
    if (work == you_are_done)
        break;
    answer = do_work(work);
    MPI_Send(answer, …);
}
```

Summary

- Benefits
  - Easily understood paradigm
  - Robust algorithm
- Drawbacks
  - Master process cannot do any work other than calculating the final result
  - To improve: Master needs to do work and control simultaneously

Method 2: Combined Master / Slave

- Total of N MPI processes
  - N Slave processes
  - Master is combined with Slave 1
- Not wasting a full process for the Master



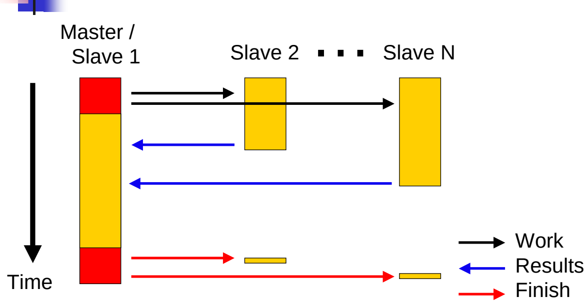
Combined Master / Slave

- Combined master and slave routines in 1st Slave
  - Send / receive work
  - Do work / calculate answers
- Use non-blocking receives to collect results
  - Use MPI_TEST calls to poll for results
- Master must track state of receives rather than simple outstanding work counter

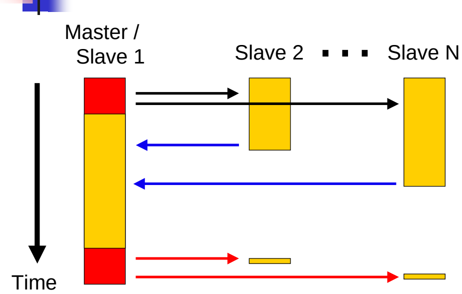
Combined Master / Slave

- New combined master algorithm
  - Send initial work set
    - Post MPI_IRECV for each item of work sent
  - Loop
    - If work available, do work locally
    - Check for completion of other slaves
    - If completion, send more work or "finish" message
  - End loop when no more work to be done and all slaves finished

Combined Master / Slave

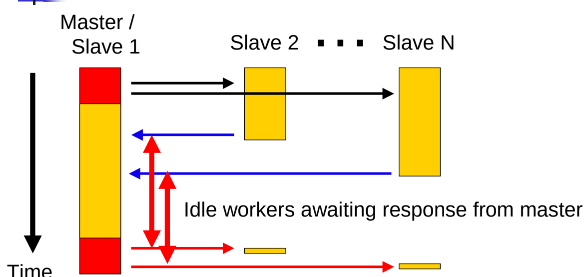


Combined Master / Slave



Overall completion time is shorter, BUT…

Combined Master / Slave



Results cannot be processed while master is working even when using IRECV / TEST

Summary

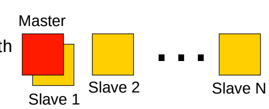
- Benefits
  - Does not waste a process for the Master
- Drawbacks
  - Complicated application code
  - Master does not asynchronously process messages while working
  - Not just simple overlapping of computation and communication
  - Stalls the work pipeline -- idle workers

Method 3: Thread Based Combined Master / Slave

- Use threads
  - Master code in one thread
  - Slave code in another thread
  - Independent progress
- Code now almost identical to Method 1
  - Simplified code / less custom code = less errors

Thread Based Combined Master / Slave

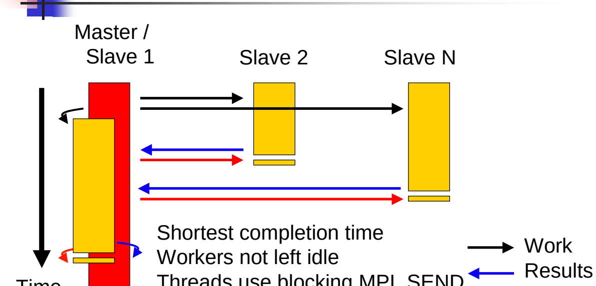
- Total of N MPI processes
  - N Slave processes
  - Master is combined with Slave 1
- Similar concept to Method 2 (one process)
- But similar code to Method 1 (simple code)



Application main()

```
MPI_Init_thread(…,
  MPI_THREAD_MULTIPLE, …);
MPI_Comm_rank(…, &rank)
if (rank == 0)
  pthread_create(…,do_master, …);
do_slave();
pthread_join(…);
MPI_Finalize();
```

Thread Based Combined Master / Slave



Shortest completion time
Workers not left idle
Threads use blocking MPI_SEND and MPI_RECV

Practical Exercise

- Goals:
  - Compile and run the provided master-slave example code
  - Alter the master example code so that it creates an additional thread in MPI_COMM_WORLD rank 0
    - The main thread performs Master functionality
    - The additional thread performs Slave work
    - The two threads can only communicate via MPI send and receive calls

Summary

- Benefits
  - Simple code -- similar to method 1
  - Overlap communication and computation
- Drawbacks
  - 1st Slave might run somewhat slower than its peers