Hybrid Programming MPI and OpenMP

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

MPI, OpenMP and Pthreads

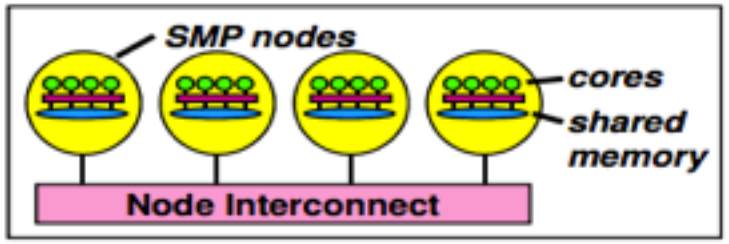
- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler,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

(Balaji, Gropp, Hoefler, Thakur, 2018)

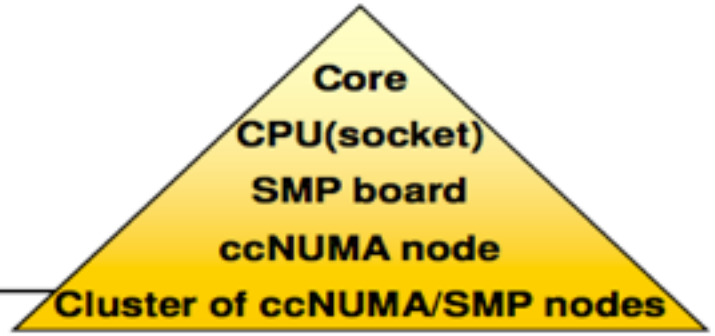
How to program on modern systems?

- Today's clusters often comprise multiple CPUs per node sharing memory, and the nodes themselves are connected by a network

- Efficient programming of clusters of shared memory (SMP) nodes



- Hierarchical system layout



- Hybrid programming seems natural
  - **MPI between the nodes**
  - **Shared memory programming inside of each SMP node**
    - OpenMP
    - new – MPI-3 shared memory programming
    - Accelerator support in new OpenMP 4.0 and OpenACC

(Balaji, Gropp, Hoefler, Thakur, 2018)

Hybridization and its benefits

- The use of inherently different models of programming in a complimentary manner, in order to achieve some benefit not possible otherwise;
- A way to use different models of parallelization in a way that takes advantage of the good points of each
- May help if
  - Introducing MPI into OpenMP applications can help scale across multiple SMP nodes
  - Introducing OpenMP into MPI applications can help make more efficient use of the shared memory on SMP nodes, thus mitigating the need for explicit intra-node communication

Basic Hybrid Stub

```
#include <omp.h>
#include "mpi.h”
#define _NUM_THREADS 4
int main (int argc, char *argv[]) {
 int p,my_rank,c;
omp_set_num_threads(_NUM_THREADS);
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
#pragma omp parallel reduction(+:c)
 {
      c = omp_get_num_threads();
}
MPI_Finalize();
```

Compiling and running

- Compilation:

  mpicc -openmp test.c –o test

- Running

  export OMP_NUM_THREADS=8

  mpirun -np 4 ./test

(Balaji, Gropp, Hoefler, Thakur, 2018)

MPI + OpenMP

- MPI defines an alternative to MPI_Init: **MPI_Init_thread**(reqd, provided)
  - *Application indicates what level of thread support it needs (reqd)*
  - *MPI returns the level of thread support it provides (provided)*
- MPI defines four levels of thread safety:

1. MPI_THREAD_SINGLE: There is no OpenMP multithreading in the program.

2. MPI_THREAD_FUNNELED: All of the MPI calls are made by the master thread, i.e. all MPI calls are: *Outside OpenMP parallel regions, or Inside OpenMP master regions, or Guarded by call to MPI_Is_thread_main*

3. MPI_THREAD_SERIALIZED

#pragma omp critical

{ …MPI calls allowed here… }

4. MPI_THREAD_MULTIPLE: Any thread may make an MPI call at any time

(Balaji, Gropp, Hoefler, Thakur, 2018)

Specification of MPI_THREAD_MULTIPLE

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- 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
- 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

(Balaji, Gropp, Hoefler, Thakur, 2018)

Threads and MPI

- The MPI implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, it is not required to be thread safe
- A fully thread-safe 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 we see)*

(Balaji, Gropp, Hoefler, Thakur, 2018)

An Incorrect Program: What is wrong here?

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

- Here the user must use some kind of synchronization 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 allowed in MPI

(Balaji, Gropp, Hoefler, Thakur, 2018)

A Correct Example: why is this right?

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

- The MPI implementation must ensure that the above 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.

(Balaji, Gropp, Hoefler, Thakur, 2018)

Performance with MPI_THREAD_MULTIPLE

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (lock granularity issue)
- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections

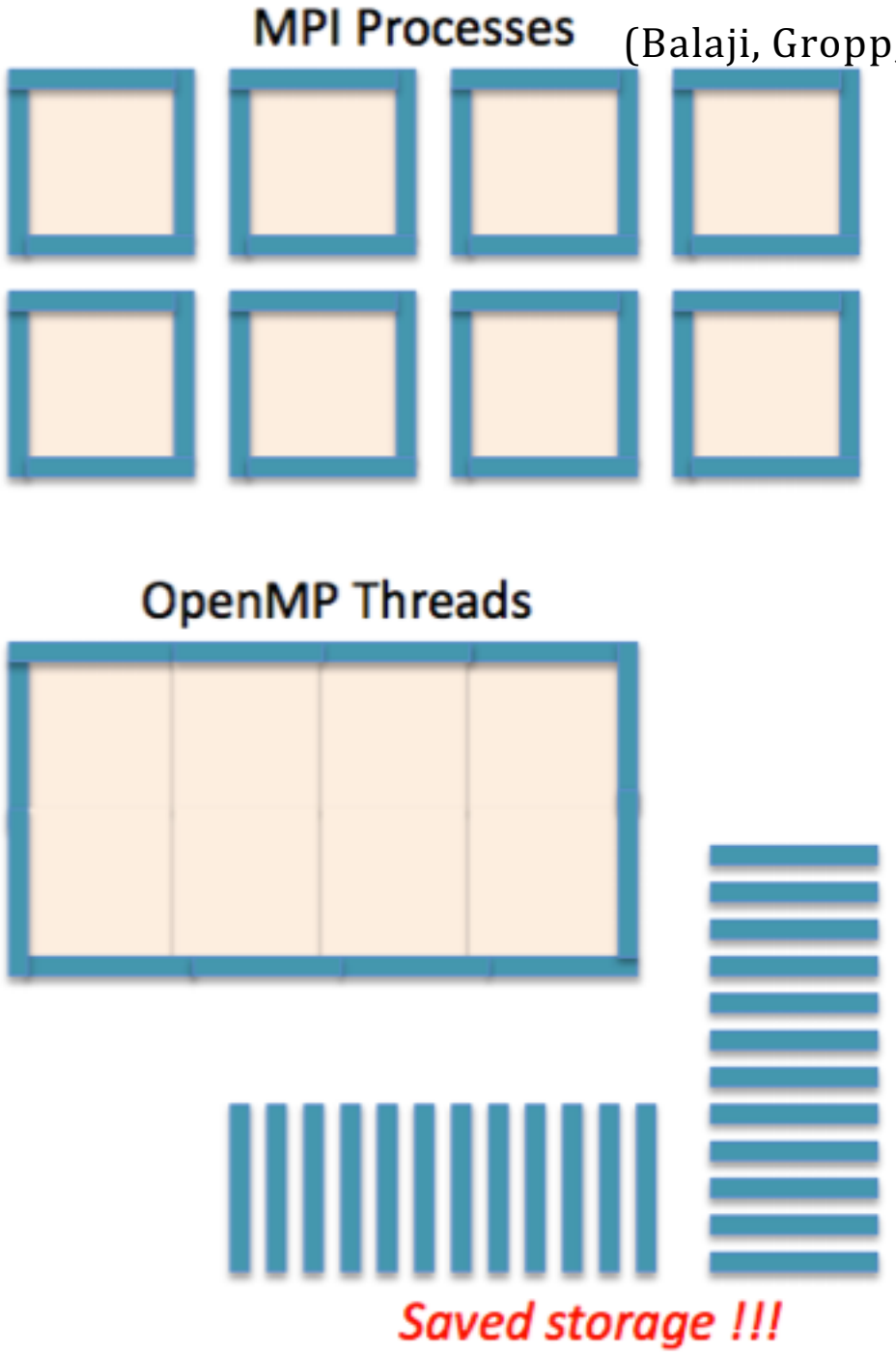
Process Memory Requirements MPI vs. MPI + OpenMP

(Balaji, Gropp, Hoefler, Thakur, 2018)

- Separate processes need separate address and memory space
- There are much more lightweight memory requirements for OpenMP threads
- Only one copy of MPI buffers etc. exists per process, and therefore only one copy exists shared between all threads launched from a process
- Using MPI/OpenMP hybrid programming reduces the memory requirement overhead from multiple processes

Halo Regions

- Halo regions are local copies of remote data that are needed for computations (remember jacobi example?)
- Using OpenMP parallelism reduces the size of halos region copies that need to be stored
- Reducing halo region sizes also reduces communication requirements



(Balaji, Gropp, Hoefler, Thakur, 2018)

(Balaji, Gropp, Hoefler, Thakur, 2018)

Overlapping Communication with Computation

- While some threads take care of communication, other threads can get the computation work done
- Earlier, we were trying to achieve this using non-blocking operations and careful overlap

```
if (my_thread_rank < ...) {
   MPI_Send/Recv....
     i.e., communicate all halo data
} else {
   Execute those parts of the application
     that do not need halo data
     (on non-communicating threads)
}

Execute those parts of the application
  that  need halo data
  (on all threads)
```

Hybrid Programming: Benefits and Pitfalls

- Natural use is when master thread does communication, and then you parallelize computational loops using OpenMP, or Pthreads
  - The problem here is that you decreased computation time significantly, but kept the communication time the same, which means overall efficiency suffers
  - Attempt to overlap computation with communication
- Complexity increases because of correctness issues and performance "gotcha"s (pitfalls.. Unexpected performance losses)
  - E.g. locking costs of shared data structures,
- But is necessary: MPI-everywhere (i.e. a rank on each hardware thread or core) leaves too much capability un-exploited
  - E.g. Avoiding copies of common (shared, read-only or read-mostly) data

Reference

Pavan Balaji, William Gropp, Torsten Hoefler, and Rajiv Thakur 2018 [Retrieved from : https://anl.app.box.com/v/balaji-tutorials-2018/