PHYS52015 Core Ib: Introduction to High Performance Computing (HPC)

Session VII: Collective MPI

Christopher Marcotte

Michaelmas term 2023

Outline

Advanced MPI techniques
MPI + X
Beyond OpenMP & MPI

pollev.com/christophermarcotte820

Non-blocking Collective Communication

- Same idea as non-blocking peer-to-peer send/recv; now prepend call name with `I`

⇒ Overlap computation and communication by making global synchronisation calls to collective operations non-blocking (recall: *immediately return*)

⇒ Initiate using non-blocking call, finish with completion call (e.g., `MPI_Wait`)

Start a broadcast of 100 `ints` from process `root` to every process in the group, perform some computation on independent data, and then complete the outstanding broadcast operation.

```
MPI_Comm comm;
int array1[100], array2[100], root=0;
MPI_Request req;
// ...
// compute something with array1 on root...
MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
// compute something on array2...
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Almost never see used outside of libraries built on top of MPI...

Persistent Communication

In computational science we frequently use MPI in a particular pattern:

- Initialize MPI and our problem domain
- * Exchange subdomain information
- * Compute a distributed update
- ⇒ *Repeat* *
- Finalize MPI and our problem

MPI has facility for this use case, with *persistent communication*.
The peer-to-peer version uses:

```
int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Start(MPI_Request *request)
int MPI_Request_free(MPI_Request *request)
```

Persistent Communication — P2P

Cornell, *Implementing Persistent Communication*: https://cvw.cac.cornell.edu/MPIP2P/percomm2

```
// Step 1) Initialize send/recv request objects
MPI_Recv_init(buf1, cnt, tp, src, tag, com, &recv_obj);
MPI_Send_init(buf2, cnt, tp, dst, tag, com, &send_obj);
for (i=1; i<BIGNUM; i++){
    // Step 2) Use start in place of recv and send
    MPI_Start(&recv_obj);
    // Do work on buf1, buf2
    MPI_Start(&send_obj);
    // Wait for send to complete
    MPI_Wait(&send_obj, status);
    // Wait for receive to finish (no deadlock!)
    MPI_Wait(&recv_obj, status);
}
// Step 3) Clean up the requests
MPI_Request_free(&recv_obj);
MPI_Request_free(&send_obj);
```

- Usage is similar to `MPI_Send` & `MPI_Recv`

$\Rightarrow$ Set up unique send/recv once, just repeatedly start them

MPI Communicator Manipulation

- We could distinguish `MPI_Sends` & `MPI_Recvs` by `tag`
- No such ability for collective operations — always uses all processes in `COMM`

$\Rightarrow$ What if we want a collective operation which only uses a subset of processes?

$\Rightarrow$ What if we want multiple collective operations which don't interfere?

MPI Communicator Manipulation

- We could distinguish `MPI_Sends` & `MPI_Recvs` by `tag`
- No such ability for collective operations — always uses all processes in `COMM`

⇒ What if we want a collective operation which only uses a subset of processes?

⇒ What if we want multiple collective operations which don't interfere?

We must manipulate the communicator!

```
// duplicate a comm
int MPI_Comm_dup(MPI_Comm incomm, MPI_Comm *outcomm);
// release a comm
int MPI_Comm_free(MPI_Comm *comm);
// split a comm
int MPI_Comm_split(MPI_Comm incomm, int colour, int key,
                   MPI_Comm *newcomm);
```

`MPI_COMM_SELF` is an example of a pre-defined intra-communicator:
`MPI_Comm_split(MPI_COMM_WORLD, rank, rank, MPI_COMM_SELF);`.

Example – communicator manipulation

How would we use these in an example?

- First we define `incomm` as `MPI_COMM_WORLD`
- And define `newcomm`
- then split `incomm` according to `rank % 2` into `newcomm`
- And finally free `newcomm` once we are done with it.

Example – communicator manipulation

How would we use these in an example?

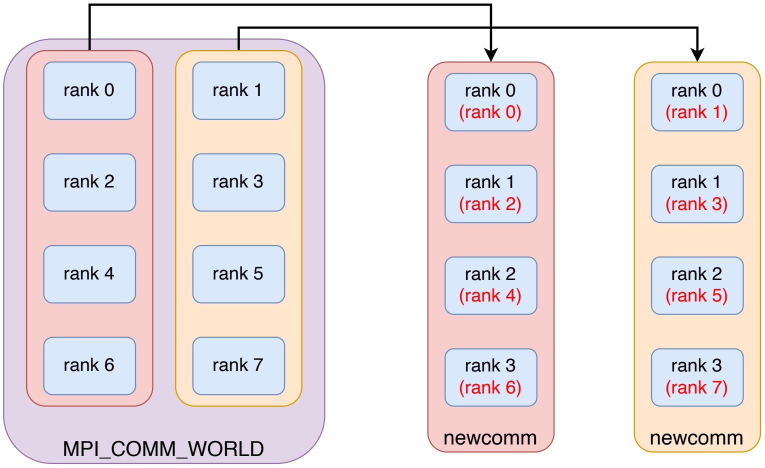
- First we define `incomm` as `MPI_COMM_WORLD`
- And define `newcomm`
- then split `incomm` according to `rank % 2` into `newcomm`
- And finally free `newcomm` once we are done with it.

```c
int rank;
MPI_Comm incomm = MPI_COMM_WORLD;
MPI_Comm newcomm;
MPI_Comm_rank(incomm, &rank)
MPI_Comm_split(incomm, rank % 2, rank, &newcomm);
// Do stuff with newcomm
MPI_Comm_free(&newcomm); // Release once we are done
```

Creating new communicators does not create new processes, it links existing ones!
Communicators are cheap to manipulate and can simplify your communication strategy.

`MPI_Comm_split` **Diagram**

```
MPI_Comm_split(MPI_COMM_WORLD, rank % 2, rank, &newcomm)
```



Virtual Process Topologies

- The communication pattern of an MPI program is, in general, a graph

$\Rightarrow$ Processes $\Leftrightarrow$ nodes , Communication $\Leftrightarrow$ edges

- Most applications use regular graph communications (rings, grids, tori, etc.)

$\Rightarrow$ Simplify set up of regular graph structures for convenience.

This can simplify finding the adjacencies for your processes according to the physical dimensions of your problem.

Virtual Process Topologies

- ▶ The communication pattern of an MPI program is, in general, a graph
- ⇒ Processes ⇔ nodes , Communication ⇔ edges
- ▶ Most applications use regular graph communications (rings, grids, tori, etc.)
- ⇒ Simplify set up of regular graph structures for convenience.

This can simplify finding the adjacencies for your processes according to the physical dimensions of your problem.

```
// ...
MPI_Dims_create(size, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
        &new_communicator);
// shift tells us our left and right neighbours
MPI_Cart_shift(new_communicator, 0, 1, &neighbours_ranks[LEFT],
        &neighbours_ranks[RIGHT]);
// shift tells us our up and down neighbours
MPI_Cart_shift(new_communicator, 1, 1, &neighbours_ranks[DOWN],
        &neighbours_ranks[UP]);
```

See HPC rookie: `https://rookiehpc.github.io/mpi/docs/mpi_cart_shift/index.html`

One-Sided Communication

- This is a new (v3.1) feature, where instead of message-passing, we specify all communication parameters from a single process through remote memory access (RMA).

One-Sided Communication

- This is a new (v3.1) feature, where instead of message-passing, we specify all communication parameters from a single process through remote memory access (RMA).
- Message-passing communication achieves two effects: **communication** of data and **synchronization**; The RMA design separates these two functions. Then can manage `A = B[index_map]` or permutation-type computations across processes.

One-Sided Communication

- This is a new (v3.1) feature, where instead of message-passing, we specify all communication parameters from a single process through remote memory access (RMA).
- Message-passing communication achieves two effects: **communication** of data and **synchronization**; The RMA design separates these two functions. Then can manage `A = B[index_map]` or permutation-type computations across processes.
- RMA functions by each process specifying a “window” in its memory that is made accessible to accesses by a remote process.

One-Sided Communication

- This is a new (v3.1) feature, where instead of message-passing, we specify all communication parameters from a single process through remote memory access (RMA).
- Message-passing communication achieves two effects: **communication** of data and **synchronization**; The RMA design separates these two functions. Then can manage `A = B[index_map]` or permutation-type computations across processes.
- RMA functions by each process specifying a "window" in its memory that is made accessible to accesses by a remote process.

```
while (!converged(A)){
    update(A);
    MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
    for(i=0; i < toneighbors; i++){
            MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                    todisp[i], 1, totype[i], win);
    }
    MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}
```

Note: as a new feature, usage may not be reliably optimised!

MPI-IO

- When using MPI, you want to write your simulation data to disk

⇒ Manually, for `nprocs` processes, this is `nprocs` files that you then need to coordinate and read to get everything back (e.g., for visualisation)

- MPI-IO coordinates writing data to disk over MPI communication infrastructure to create a single file
- The genuinely new element is `amode` (access mode):
  read-only, read-write, write-only, create, exclusive.

```
// open a file
int MPI_File_open(MPI_Comm comm, const char *filename,
        int amode, MPI_Info info, MPI_File *fh)
// read from a file
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf,
        int count, MPI_Datatype datatype, MPI_Status *status)
// write to a file
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
        int count, MPI_Datatype datatype, MPI_Status *status)
// close a file
int MPI_File_close(MPI_File *fh)
```

Concept of building block

- Content
  - Advanced MPI techniques
- Expected Learning Outcomes
  - The student knows of MPI communicator manipulation
  - The student knows of persistent communication
  - The student knows of non-blocking collective communication
  - The student knows of virtual process topologies & neighborhoods
  - The student knows of one-sided communication
  - The student knows of the relative benefits of MPI-IO

MPI + X

- If your problem benefits from distributed memory parallelism...
- ... then it probably also benefits from shared memory parallelism.
- Thus the drive to use both techniques to optimally solve your problem.

MPI + X

- If your problem benefits from distributed memory parallelism...
- ... then it probably also benefits from shared memory parallelism.
- Thus the drive to use both techniques to optimally solve your problem.

- This paradigm is especially relevant to GPU-accelerated computation — MPI not available on GPUs!

$\Rightarrow$ or FPGAs, other accelerators, heterogeneous systems generally...

- MPI+CUDA is a common combination because you rarely have more than one compute-ready GPU per physical node (slightly more common now...)

MPI + X

- If your problem benefits from distributed memory parallelism...
- ... then it probably also benefits from shared memory parallelism.
- Thus the drive to use both techniques to optimally solve your problem.

- This paradigm is especially relevant to GPU-accelerated computation — MPI not available on GPUs!

$\Rightarrow$ or FPGAs, other accelerators, heterogeneous systems generally...

- MPI+CUDA is a common combination because you rarely have more than one compute-ready GPU per physical node (slightly more common now...)

- "MPI+X" needs more work on the '+' part

$\Rightarrow$ The important question is *What to compute, where?*

- *What is 'X'?* is comparatively less interesting

MPI + OpenMP

- This is the paradigm most people think of for 'MPI+X', and the only one we'll consider in this class.
- The idea is simple:

⇒ use MPI to coordinate processes on different nodes

⇒ use OpenMP to parallelise the computation on each node, independently

```
#pragma omp parallel default(none) shared(??, ??, ??) private(??, ??)
{
np = omp_get_num_threads();
iam = omp_get_thread_num();
printf("thread %d of %d, process %d of %d on %s\n",
    iam, np, rank, numprocs, processor_name);
}
```

MPI + OpenMP

- This is the paradigm most people think of for ‘MPI+X’, and the only one we’ll consider in this class.
- The idea is simple:

⇒ use MPI to coordinate processes on different nodes

⇒ use OpenMP to parallelise the computation on each node, independently

```
#pragma omp parallel default(none) shared(??, ??, ??) private(??, ??)
{
np = omp_get_num_threads();
iam = omp_get_thread_num();
printf("thread %d of %d, process %d of %d on %s\n",
    iam, np, rank, numprocs, processor_name);
}
```

- Which variables should be `shared`? Which should be `private`?
- Given `OMP_NUM_THREADS=8` and `mpirun -n 4`, how many lines does this output?
- On a single node CPU with 8 cores, how many processes and threads should we use for compute-bound tasks?

MPI+CUDA & CUDA-aware MPI

- With CUDA in particular, significant work already done to improve MPI+CUDA
- With regular MPI only pointers to **host** memory can be passed to MPI.

⇒ You need to stage GPU buffers through host memory (expensive!!!)

MPI+CUDA & CUDA-aware MPI

- With CUDA in particular, significant work already done to improve MPI+CUDA
- With regular MPI only pointers to **host** memory can be passed to MPI.

⇒ You need to stage GPU buffers through host memory (expensive!!!)

Without CUDA-awareness:

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost); // expensive
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
//MPI rank n-1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice); // expensive
```

MPI+CUDA & CUDA-aware MPI

- With CUDA in particular, significant work already done to improve MPI+CUDA
- With regular MPI only pointers to **host** memory can be passed to MPI.

⇒ You need to stage GPU buffers through host memory (expensive!!!)

Without CUDA-awareness:

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost); // expensive
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
//MPI rank n-1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice); // expensive
```

With CUDA-awareness:

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

Concept of building block

- Content
  - Brief look at ‘MPI + OpenMP’
  - Brief look at ‘MPI + CUDA’
- Expected Learning Outcomes
  - The student knows of MPI + OpenMP coordination
  - The student knows of CUDA-aware MPI programs
  - The student knows of pitfalls of ‘MPI + X’ paradigm
- Further reading:
  - Michael Wolfe, *Compilers and More: MPI+X*: `https://www.hpcwire.com/2014/07/16/compilers-mpix/`
  - Gropp *et al.*, *Is it time to retire the ping-pong test?*: `https://dl.acm.org/doi/10.1145/2966884.2966919`

Review: OpenMP

- Shared memory parallelism using the BSP programming model
- Progressive parallelisation approach from serial code using `pragmas`
- Primary parallel feature is the `for` loop
- Primary difficulty is the management of data access

```
#include <omp.h>
// ...
#pragma omp parallel for default(shared) private(i) reduction(+:sum)
for (i=0; i < N; i++){
        a[i] = f(b[i]); // f is very expensive!
        sum += a[i];
}
```

- Lots of other optimisation features: `simd`, `collapse(n)`, `device`, etc.
- Tasking is interesting, but unused because of intersection of scientific community users and task-level parallelism
- Out of luck if your problem doesn't fit into the memory on one machine

Review: MPI

- Distributed memory parallelism using the SPMD programming model
- Best case scenario is the bulk compute pattern is preserved from serial code
- Primary parallel feature is *everything/nothing*
  Everything is parallel, you write the communication
- Primary difficulty is the coordination of messages to avoid deadlock

```
#include <mpi.h>
// ...
MPI_Send(&buf1, n, MPI_DOUBLE, );
MPI_Recv();
```

- Huge number of communication patterns
- But MPI itself does very little for *parallel computation*, basically serial on each process
- Not quite feasible to do progressive parallelisation from serial code

Parallelism in other languages

Python:

```python
import numpy as np
A = np.random.rand(500,500)
B = np.random.rand(500,500)
C = np.random.rand(500,500)
C = A + B # calls multithreaded C library for broadcasting numpy arrays
```

Julia:

```julia
A = rand(500,500)
B = rand(500,500)
C = rand(500,500)
# C .= A .+ B does the same, single-threaded, by broadcasting
Threads.@threads for i in eachindex(A,B,C) # threaded for-loop like OpenMP
    C[i] = A[i] + B[i]
end
```

Beyond MPI & OpenMP

Disclaimer: You do not need *C* / *Fortran* & MPI / OpenMP in order to achieve high-performance computations.

Beyond MPI & OpenMP

Disclaimer: You do not need *C* / *Fortran* & MPI / OpenMP in order to achieve high-performance computations.

- *C* has MPI + OpenMP, etc.
- Julia has MPI + Threads (like OpenMP), etc.
- Python has MPI + numpy (like OpenMP template library), etc.
- Rust has channels (like MPI) + threads (like OpenMP), etc.
- Go has channels (like MPI) + go routines (like OpenMP), etc.
- Even Javascript has web workers (like OpenMP tasks)

⇒ . . . And most of these are inter-operable!

Beyond MPI & OpenMP

Disclaimer: You do not need *C* / *Fortran* & MPI / OpenMP in order to achieve high-performance computations.

- *C* has MPI + OpenMP, etc.
- Julia has MPI + Threads (like OpenMP), etc.
- Python has MPI + numpy (like OpenMP template library), etc.
- Rust has channels (like MPI) + threads (like OpenMP), etc.
- Go has channels (like MPI) + go routines (like OpenMP), etc.
- Even Javascript has web workers (like OpenMP tasks)

$\Rightarrow$ . . . And most of these are inter-operable!

In all these languages, the usual tactics are:

- Make your serial program as fast as possible first
- Write as much of your code in terms of BLAS primitives as possible
  - BLAS Level 1: vector-vector operations
  - BLAS Level 2: matrix-vector operations
  - BLAS Level 3: matrix-matrix operations

Concept of building block

- Content
  - High level review of OpenMP
  - High level review of MPI
  - Placing OpenMP & MPI in broader context of HPC
  - Other languages approaches to parallelism for HPC
- Expected Learning Outcomes
  - The student *knows* of the broader context of parallelism beyond OpenMP & MPI
  - The student knows some of the alternatives used in other languages
- Further Reading:
  - *HPC is dying and MPI is killing it* `https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it`
  - *[. . .] Fundamental Laws Run Out of Steam* `https://semiengineering.com/chip-design-shifts-as-fundamental-laws-run-out-of-steam/`
  - *Cataloging the Visible Universe through Bayesian Inference at Petascale* `https://arxiv.org/abs/1801.10277`
  - *MPI+X: Opportunities and Limitations [. . .]* `https://www.cmor-faculty.rice.edu/~mk51/presentations/SIAMPP2016_5.pdf`