Lecture 5 - HPC Programming Models

Gidon Rosalki
2026-06-09

Notice: If you find any mistakes, please open an issue in the github repository

1. Recap and Introduction

We have discussed RDMA, its benefits and drawbacks, and we should now be able to write software that makes use of RDMA. We are now going to jump to a very basic overview of High Performance Computing. Very basic since we are effectively compressing a complete course into a single lecture. Once we have this background, we will reconnect this to RDMA, through collective communication.

2. High Performance Computing

There are 3 important things in HPC:

1. Performance
2. Performance
3. Performance

Cloud computing also has requirements of security, where we want good isolation between our processes. However, in HPC, we will give up security for more performance. This is possible since we have more control over our computer clusters.
Performance is a quantifiable measure of rate of doing computational work. We have multiple such measures. We may measure at the level of the basic operations:

- ops - operations per second
- ips - instructions per second
- flops - floating point operations per second

We may also measure the rate at which a benchmark program is executed. This is generally a carefully crafted, and controlled piece of code used to compare systems.

- Linpack Rmax
- gups (billion updates per second)

I will note that there is a drawback to benchmarks, that sometimes vendors use the benchmark as a target, rather than real world performance, since they do not always directly correlate. Remember, in the competitive world, as soon as you have some sort of measure, it quickly becomes a target.

There are 2 main perspectives on performance:

- Peak performance, which is the maximum theoretical performance possible for a system
- Sustained performance, which is observed for a particular workload and run, and may vary across workloads and possibly even between runs

We will note that the performance stated by a vendor, such as Intel, and Nvidia, is a theoretical number. Buying a device does not guarantee that performance. Buying 1000 processors from Intel, and running a massively parallelised program across them does not guarantee 1000 times the performance, but likely rather less (though possibly more).

2.1. Two (Plus 1) Major Categories of Applications

Traditionally, HPC was mainly focused on Scientific Computing, such as in Physics and Chemistry. They mainly make use of MPI, as it is the dominant programming model.

Another use is Big data, enterprise, and commercial computing. This focuses on large amounts of data, and data analysis. Some software called Spark emerged for in memory computing.
Big data mostly disappeared into AI, since AI requires huge amounts of training data, and also when running these models, we need to store their weights. The main difference is that AI also requires huge amounts of compute, not just data. This mainly uses Message Passing Programming Model (not the MPI standard!), and is moving towards PGAS (which is for one sided communication).

2.1.1. Applications (Scientific & Engineering)

There are practically endless applications here. Between analysis of parts for aircraft, all the way to simulating the covid19 virus back in 2020, and managing to create a vaccine in 2 years, rather than 20. The main problem with this is that it is a very expensive method. To perform sufficiently accurate calculations, we need practically unlimited computation, energy, space, and so on. However, we cannot deny the benefits that it brings us (as opposed to AI, where we definitely can).

2.2. Parallel Programming Models

We have no real way of continuing to massively increase the processing power of a single core, so the only real solution to this is massive parallelisation. Programming models provide abstract machine models:

- Shared memory, many threads on the OS share memory, and may access this memory to share information. This is often does through things like pthreads, though there is also OpenMP, which provides a slightly more comfortable abstraction on top of pthreads. This is mainly used by scientists, and engineers, that do not want/need to understand the actual software like pthreads.
- The distributed memory model, and MPI (Message Passing Interface). It is very difficult if we have many different processors to have shared memory between them, since the communication speed between the different groups of memory is so limited. Additionally, it helps solve the failure mode problem, where in shared memory, if one core has power problems, the entire system fails, but in distributed computing, if one computer fails, then the system may continue
- Message Passing is relatively difficult. To resolve this, we created distributed systems, with logical shared memory. This memory assigns logical areas to each computer in the distributed system, and then create an abstraction layer that lets this appear and act like shared memory. This hides from the user the complications of passing information between the processors. This is called the Partitioned Global Address Space (PGAS).

Model	Scalability	Performance	Ease of Use	Portability
Shared Memory	Limited	High (local)	Easy	Moderate
Message Passing	High	High	Moderate	High
PGAS	High	High	Hard	Variable
Hybrid	Very High	Best case	Complex	High

Table 1:

We will note that most programs make use of Message Passing, since it allows excellent performance, and an easy to use abstraction, with only a limited increase in difficulty. It also allows us to optimise it to all manner of different systems, without our physicist having to learn all the difficult things like RDMA, and rewrite his entire application every time we create a new method of this computation. We simply change the backend of the MPI library, and he promptly has all these benefits.

2.3. MPI

2.3.1. What is MPI

MPI is a standardised, adnd portable message passing system designed to function on a wide variety of parallel computing architectures. It enables processes to communicate with one another by sending and receiving messages, making it suitable for distributed memory systems. It comes with a few key features:

- Portability: Runs on various parallel computing platforms
- Performance: Designed for high efficiency and Scalability
- Language support: Provides direct bindings for C, C++, and Fortran, but functionally has bindings for every language you may want

MPI has a few basic concepts:

- Processes: Independent execution units with separate memory spaces
- Communicators: Defines groups of processes that may communicate
- Ranks: Unique identifiers for processes within a communicator

2.3.2. MPI Communication Paradigms

1. Point to Point communication
   - This has blocking operations such as MPI_Send, MPI_Recv
   - Non blocking operations such as MPI_Isend, MPI_Irecv
   - These are used for direct communication between pairs for processes
2. Collective Communication
   - Operations involving a group of processes
   - Examples:
     - Broadcast: MPI_Bcast. We can obviously have this just be one process sending all the data to all the other processes, or we can do this in $O(\log n)$ instead of $O(n)$, and have each process send to 2 children instead (there are other methods to optimise, like k-nary trees rather than binary, and more)
     - Gather/Scatter: MPI_Gather, MPI_Scatter
     - Reduction: MPI_Reduce, MPI_Allreduce. These are incredibly interesting, and we will focus on them. Reduce just means that each part computes part of the problem (like everyone finds their local minimum in an optimisation problem), and then at the end we find the result from each part (the minimum of the minima). In MPI_Reduce, the result is just kept at the root, and in Allreduce it is shared to all the nodes.
3. One-Sided Communication (introduced in MPI-2)
   - Allows a process to specify all communication parameters for both sending and receiving data
   - Examples include MPI_Put, and MPI_Get

Data parallelisation is a use of reduce. We may parallelise the computation of backpropagation (for example). We make n copies of our neural network, each copy is given an n-th of the data, and then backpropagates across that data. To find the final weights, we just need to sum these results. This is a problem with many excellent solutions.

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

int
main (int argc, char **argv)
{
    // Initialize the MPI environment
    MPI_Init ();

    // Get the number of processes
    int world_size;
    MPI_Comm_size (MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank (MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name (processor_name, &name_len);

    // Print off a hello world message
    printf ("Hello world from processor rank %d out of %d processors\n",
            world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize ();
}
```

```
Output:
Hello from processor rank 0 out of 4 processors
```

```
Hello from processor rank 1 out of 4 processors
Hello from processor rank 2 out of 4 processors
Hello from processor rank 3 out of 4 processors
```

2.4. PGAS

2.4.1. What is PGAS

PGAS is a parallel programming model that provides a global memory address space, logically partitioned among all processes. Each process has affinity to a portion of the shared memory, enabling both local and remote memory access.

Key characteristics: Combines the ease of shared memory programming, with the performance of message passing. It also supports one sided communication, allowing a process to read / write memory without getting the other process involved.

Some common libraries include UPC, CAF, Chapel, C10, OpenSHMEM.

3. Part 2: Message Passing Interface (MPI)

A slight problem here is that we have the MPI model, and the MPI library. The MPI library is the implementation of the MPI model, and very standardised, but it is important to remember that they are distinct things, and not necessarily a one to one correlation.

MPI point to point communication is for direct communication between two processes, and may be blocking or non blocking. There is also collective communication, which involves all processes in a communicator. It is possible that the communicator only has 2 processes, but also possible that it has 2 million.

We also have **one sided communication** (RMA). Here, a process directly accesses the memory of another, without its active participation. There are a few key functions:

- MPI_Put: Write to remote memory
- MPI_Get: Read from remote memory
- MPI_Accumulate: Remote atomic update

There are a few additional memory model concepts:

- Exposure epoch (MPI_Win_post/start)
- Access epoch (MPI_Win_lock/unlock)

We will note that this is not massively used, since the person that did this did not really understand what he was doing. Those that want one sided communication use PGAS instead.

3.1. Key Terminology

- Rank: This is the unique identifier for each process within a communicator. It has the value in $[0, n-1]$, where n is the number of processes
- Communicator: A group of processes that may communicate with each other. Default is MPI_COMM_WORLD, which includes all processes. Custom communicators may be defined for subgroup operations
- Tag: This is an integer label attached to a message, and helps distinguish between different message types or sources. It is useful for filtering in MPI_Recv. Consider if we have a single channel, we may use tag to multiplex on this channel, and allow us to send many different sets of communication over this single channel.
- Message: A unit of communication, consists of a data buffer, data type, count, source, destination, tag, and communicator.
- Datatype: Describes the type of elements in a message (MPI_INT, MPI_FLOAT). May also define custom datatypes using MPI_Type_create_struct. Why does the type matter to MPI? Operations like Allreduce may be able to optimise depending on the type.

In these examples, each table is a table of processors, and data. The processors are the rows, and the data the columns.

A_0					

Broadcast $\Longrightarrow$

A_0					
A_0					
A_0					
A_0					
A_0					
A_0					

Table 2: Broadcast

As we can see, broadcast simply transmits 1 data vector from 1 process, to all the other processes.

A_0	A_1	A_2	A_3	A_4	A_5

Scatter $\Longrightarrow$
$\Longleftarrow$ Gather

A_0					
A_1					
A_2					
A_3					
A_4					
A_5					

Table 3: Scatter, and gather

Scatter splits the vector, and sends the first value to the first processor, the second to the second processor, and so on. Gather is in the other direction, where we gather together all the data from all the processes into a single process.

A_0					
B_1					
C_2					
D_3					
E_4					
F_5					

Allgather $\Longrightarrow$

A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0

Table 4: Allgather

Here, every process has a part of a vector, and we want to gather it, and ensure that the result arrives to everyone.

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

Alltoall $\Longrightarrow$

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Table 5: Alltoall

Alltoall is that every process has a vector, and every process does a scatter of its vector, and gathers all the vectors. As may be seen above, if we consider the matrix of processes and data, this is like doing a transposition of the matrix. It is a very intense operation, since every process is both sending the whole vector of data, *and* receiving. If you imagine each vector is 1GB, and you have 12000 process, then *each* process is receiving a gigabyte from each other process, and sending a gigabyte to each other process.

3.2. Multiple Algorithms, One Collective

The same collective operation may be implemented using different algorithms. Each algorithm trades off latency, bandwidth, message size, and scalability. MPI libraries choose implementations based on heuristics, benchmarks, or user settings. Not every implementation is best for every setup, consider if we have a datacentre in Haifa, and one in Tel Aviv. We split them up due to electricity limitations, and the connection speed between them is clearly much slower than the connection speed inside the datacentre. There are algorithms to optimise communicating across this boundary, when we need to share information in a broadcast, for example.

3.3. Theoretical Cost Model

We will use a simple cost model to evaluate the algorithm cost:

$$T(n, P) = \alpha + \beta \cdot n + \gamma \cdot n$$

Where

- α is the startup cost, represents the fixed latency overhead associated with initiating any communication operation
- β is the communication cost, and denotes the inverse bandwidth, representing the per byte transmission time through the network fabric
- γ is the computation cost, where if we need to perform some computation operation on the data, that must also be factored in
- n is the vector size
- P is the number of processes

We are missing here the **incast term**, which accounts for congestion when multiple flows converge simultaneously, and the **memory access term**, which captures the overhead of accessing data in different memory hierarchies.

3.4. MPI_Allreduce

The purpose of allreduce is to combine the reduction, and broadcast in one step. All processes contribute values and receive the reduced result. This is widely used in AI/ML for gradient aggregation, and is also used in scientific computing for aggregating statistics. Implementing this on top of RDMA, for our architecture, is going to be our final project.

There are a many different algorithms for implementing allreduce. Each has its own drawbacks, and benefits, and for a slight spoiler, we are going to implement ring.

- Ring algorithm: Bandwidth efficient, better for large messages
- Recursive doubling - good for short messages
- Rabenseifner's algorithm - hybrid of reduce scatter and allgather

To detail ring further, each process has an input vector of data, and we need at the end that each process will have the sum of all the input vectors in its output vector. For 4 processors, we will begin by splitting our input vectors into 4. Every process will take a quarter of the vector (each taking a different quarter), and send it to the next in line in a ring. We may now sum together the relevant part of the vector, and send the results to the right once more, meaning what we have now sent is the sum of 2 entries, not just one. We now repeat this until every processor now has a quarter of the final summed vector (note that we only sent the data 3 times, not 4, thank you fence post). We copy this part to the output buffer, and send to the right once more. We now do this until everyone has all the vector.

Note, this is effectively applying reduce-scatter, and then allgather. Do this division of operations when you write your implementation.