TACC Technical Report TR-15-02

Teaching MPI from Mental Models

Victor Eijkhout

June 29, 2016

Abstract

The Message Passing Interface (MPI) is a *de facto* standard for programming large scale parallelism, with up to millions of individual processes. Its dominant paradigm of Single Program Multiple Data (SPMD) programming is different from threaded and multicore parallelism, to an extent that students have a hard time switching models. In contrast to threaded programming, which allows for a view of the execution where someone, a 'master thread', is in charge, and there is a central repository of data, SPMD programming uses a symmetric model where all processes are active all the time, and none is priviliged in any sense, and where data is distributed.

This model is counterintuitive to the novice parallel programmer, so care needs to be taken how to instill the proper 'mental model'.

We identify problems with the currently common way of teaching MPI, and propose a better way. First of all, to instill the symmetric mindset we teach MPI by letting the students program a number of exercises that are symmetric in nature. This also means that we teach starting from realistic scenarios, rather than writing code just to exercise a newly-learned routine.

This motivation implies that we reverse the commonly used order of presenting MPI routines, starting with collectives, and later introducing point-to-point routines only as support for certain symmetric operations, avoiding the process-to-process model.

1 Introduction

The Message Passing Interface (MPI) library [6, 5] is the *de facto* tool for large scale parallelism as it is used in engineering sciences. Its main model for parallelism is described as Single Program Multiple Data (SPMD): multiple instances of a single program run on the processing elements, each operating on its own data. The synchronization between the MPI processes is done through explicit send and receive calls.

The main motivation for MPI is the fact that it can be scaled to more or less arbitrary scales, currently up to a million cores [1]. Contrast this with threaded programming, which is limited more or less by the core count on a single node, currently about 70.

While MPI programs can solve many or all of the same problems that can be solved in a multicore context, the programming approach is different, and requires an adjustment in the programmer's 'mental model' [3, 7] of the parallel execution. This paper addresses the question of how to teach MPI to best effect this shift in mindset.

1.1 The traditional view of parallelism

The problem with mastering the MPI library is that beginning programmers take a while to overcome a certain mental model for parallelism. In this model, which we can call 'sequential semantics', there is only a single strand of execution[1], where simple operations (such as between scalars) are done traditionally in a sequential manner, but certain operations (typically on arrays) are magically done in parallel. Interestingly, work by Ben-David Kolikant [2] shows that students with no prior knowledge of concurrency, when invited consider about parallel activities, will still think in terms of centralized solutions.

This mental model corresponds closely to the way algorithms are described, and it is actually correct to an extent in the context of threaded libraries such as OpenMP, where there is indeed initialy a single thread of execution, which in some places spawns a team of threads to execute certain sections of code in parallel.

However, in MPI this model is factually incorrect, since there are always multiple processes active. The centralized model can still be maintained to an extent, since the scalar operations that would be executed by a single thread become replicated operations in the MPI processes. The distinction between sequential execution and replicated execution escapes many students at first.

1. We carefully avoid the word 'thread' which carries many connotations in the context of parallel programming.

1.2 Misconceptions in programming MPI

However, this sequential semantics mental model invites the student to adopt certain programming techniques, such as the master-worker approach to parallel programming. While this is often the right approach with thread-based coding, where we indeed have a master thread and spawned threads, it is usually incorrect for MPI.

The strands of execution in an MPI run are all long-living processes (as opposed to dynamically spawned threads), and are *symmetric* in their capabilities and execution. Lack of recognition of this symmetry also induces students to solve problems by having a form of 'central data store' on one process, rather than adopting a symmetric, distributed, storage model.

For instance, we have seen a student solve a data transposition problem by collecting all data on process 0, and subsequently distributing it again in transposed form. While this may be reasonable in shared memory with OpenMP, with MPI it is unrealistic in that no process is likely to have enough storage for the full problem. Also, this introduces a sequential bottleneck in the execution.

In conclusion, we posit that beginning MPI programmers may suffer from a mental model that makes them insufficiently realize the symmetry of MPI processes, and thereby arrive at inefficient and nonscalable solutions.

We now consider the way MPI is usually taught, and offer an alternative that is less likely to lead to an incorrect mental model.

2 Teaching MPI, the traditional way

The MPI library is typically taught as follows. After an introduction about parallelism (covering speedup and such), and shared versus distributed memory parallelism, students learn about the initialization and finalization routines, and the `MPI_Comm_size` and `MPI_Comm_rank` calls for querying the number of processes and the rank of the current process.

After that, the typical sequence is

1. two-sided communication, with first blocking and later non-blocking variants;
2. collectives; and
3. any number of advanced topics such as derived data types, one-sided communication, subcommunicators, MPI I/O et cetera, in no particular order.

This sequence is defensible from a point of the underlying implementation: the two-sided communication calls are a close map to hardware behaviour, and communications are both conceptually equivalent to, and can be implemented as, a sequence of

point-to-point communication calls. However, this is not a sufficient justification for teaching this sequence of topics.

2.1 Criticism

We offer three points of criticism againt this traditional approach to teaching MPI.

First of all, there is no real reason for teaching collectives after two-sided routines. They are not harder, nor require the latter as prerequisite. In fact, their interface is simpler for a beginner, requiring one line for a collective, as opposed to at least two for a send/receive pair, probably surrounded by conditionals testing the process rank. More importantly, they reinforce the symmetric process view, certainly in the case of the `MPI_All...` routines.

Our second point of criticism is regarding the blocking and non-blocking two-sided communication routines. The blocking routines are typically taught first, with a discussion of how blocking behaviour can lead to load unbalance and therefore inefficiency. The non-blocking routines are then motivated from a point of latency hiding and solving the problems inherent in blocking. In our view such performance considerations should be secondary. Non-blocking routines should instead be taught as the natural solution to a conceptual problem, to be explained below.

Also, teaching the non-blocking routines as a somehow more desirable form of the former insufficiently teaches the students the essential point that each non-blocking call needs its own buffer. We regularly see students treat the send or receive buffers in non-blocking calls identically to those in blocking routines. Typically, they will also fail to save all the request objects, only issuing a single `MPI_Wait` call on the last request. This is a correctness bug that is very hard to find, and at large scale it induces a memory leak since many requests objects just are lost.

Thirdly, starting with point-to-point routines stems from a 'Communicating Sequential Processes' [4] (CSP) view of a program: each process stands on its own, and any global behaviour is an emergent property of the run. This may make sense for the teacher who know how concepts are realized 'under the hood', but it does not lead to additional insight with the students. We believe that a more fruitful approach to MPI programming starts from the global behaviour, and then derives the MPI process in a top-down manner.

We will now outline our proposed order for teaching the MPI concepts.

3 Teaching MPI, our proposal

As alternative to the above sequence of introducing MPI concepts, we propose a sequence that focuses on practical scenarios, and that actively reinforces the mental model of SPMD execution.

3.1 Process symmetry

Paradoxically, the first way to get students to appreciate the notion of process symmetry in MPI is to run a non-MPI program. Thus, students are asked to write a 'hello world' program, and execute this with `mpiexec`, as if it were an MPI program. Every process executes the print statement identically, bearing out the total symmetry between the processes.

Next, students are asked to insert the initialize and finalize statements, with three different 'hello world' statements before, between, and after them. This will prevent any notion of the code between initialization and finalization being considered as an OpenMP style 'parallel region'.

A simple test to show that while processes are symmetric they are not identical is offered by the exercise of using the `MPI_Get_processor_name` function, which will have different output for some or all of the processes, depending on how the hostfile was arranged.

3.2 Functional parallelism

The `MPI_Comm_rank` function is introduced as a way of distinguishing between the MPI processes. Students are asked to write a program where only one rank prints the output of `MPI_Comm_size`.

Having different execution without necessarily different data is a case of 'functional parallelism'. At this point there are few examples that we can assign. For instance, in order to code the evaluation of an integral by Riemann sums ($\pi/4 = \int_0^1 \sqrt{1-x^2}dx$ is a popular one) would need a final sum collective, which has not been taught at this point.

A possible example would be primality testing, where each process tries to find a factor of some large integer N by traversing a subrange of $[2, \sqrt{N}]$, and printing a message if a factor is found. Boolean satisfyability problems form another example, where again a search space is partitioned without involving any data space; a process finding a satisfying input can simply print this fact.

3.3 Introducing collectives

At this point we can introduce collectives, for instance to find the maximum of a random value that is computed locally on each process. This requires teaching the code for random number generation and, importantly, setting a process-dependent random number seed. Generating random 2D or 3D coordinates and finding the center of mass is an examples that requires a send and receive buffer of length greater than 1, and illustrates that reductions are then done pointwise.

These examples evince both process symmetry and a first form of local data. However, a thorough treatment of distributed parallel data will come in the discussion of point-to-point routines.

It is an interesting question whether we should dispense with 'rooted' collectives such as `MPI_Reduce` at first, and start with `MPI_Allreduce`. The latter is more symmetric in nature, and has a buffer treatment that is easier to explain; it certainly reinforces the symmetric mindset. To this author, there are few reasons to prefer

```
MPI_Reduce( indata,outdata, /* root= */ 0 );
if (myprocno==0)
  printf("The result is .... ",outdata);
```

over

```
MPI_Allreduce( indata,outdata );
if (myprocno==0)
  printf("The result is .... ",outdata);
```

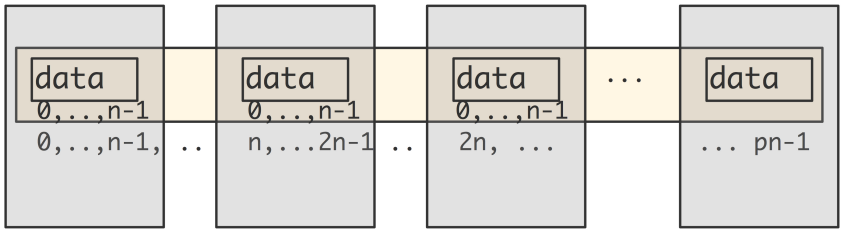
Certainly, in most applications the 'allreduce' is the more common mechanism, for instance to compute a replicated scalar quantity such as an inner product. The rooted reduction is typically only used for final results. Therefore we advocate introducing both rooted and non-rooted collectives, but letting the students initially do exercises with the non-rooted variants.

This has the added advantage of not bothering the students initially with the asymmetric treatment of the receive buffer between the root and all other processes.

3.4 Distributed data

As motivation for the following discussion of point-to-point routines, we now introduce the notion of distributed data. In its simplest form, a parallel program operates on a linear array the dimensions of which exceed the memory of any single process.

```
int n;
double data[n];
```



The lecturer stresses that the global structure of the distributed array is only 'in the programmer's mind': each MPI process sees an array with indexing starting at zero. The following snippet of code is given for the students to use in subsequent exercises:

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
   int iglobal = myfirst+ilocal;
   array[ilocal] = f(iglobal);
}
```

At this point, the students can code a second variant of the primality testing exercise above, but with an array allocated to store the integer range. Since collectives are now known, it becomes possible to have a single summary statement from one rank, rather than a partial result statement from each.

The inner product of two distributed vectors is a second illustration of working with distributed data. In this case, the reduction for collecting the global result is slightly more useful than the collective in the previous examples. For this example no translation from local to global numbering is needed.

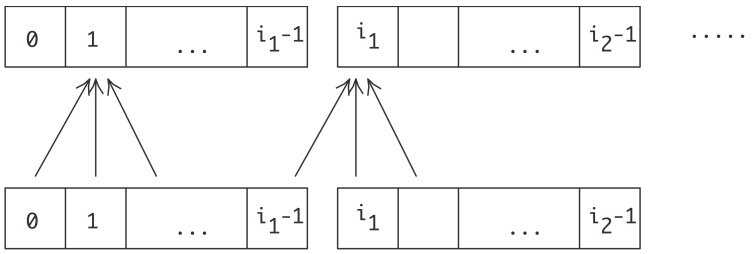
3.5 Point-to-point motivated from operations on distributed data

We now state the importance of local averaging operations such as

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \ldots, N-1$$

applied to an array. Students that know about Partial Differential Equations (PDEs) will recognize the heat equation; for others a graphics 'blur' operation can be used as illustration.

Under the 'owner computes' regime, where the process that stores location y_i performs the full calculation of that quantity, we see the need for communication in order to compute the first and last element of the local part of y:



We then state that this data transfer is realized in MPI by two-sided send/receive pairs.

3.5.1 Detour: ping-pong

At this point we briefly abandon the process symmetry, and consider the ping-pong operation between two processes A and B. We ask students to consider what the ping-pong code looks like for A and, for B. Since we are working with SPMD code, we arrive at a program where the A code and B code are two branches of a conditional.

We ask the students to implement this, and do timing with `MPI_Wtime`. The concepts of latency and bandwidth can be introduced, as the students test the ping-pong code on messages of increasing size. The concept of halfbandwidth can be introduced by letting half of all processors execute a ping-pong with a partner process in the other half.

3.5.2 Another detour: deadlock and serialization

The concept of 'blocking' is now introduced, and we discuss how this can lead to deadlock. For completeness, the 'eager limit' can be discussed, and how code that semantically should deadlock may still work in practice.

The following exercise is done in the classroom:

> Each student holds a piece of paper in the right hand – keep your left hand behind your back – and executes the following program:
> 1. If you are not the rightmost student, turn to the right and give the paper to your right neighbour.
> 2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbour.

This introduces students to some subtleties in the concept of parallel correctness: a program may give the right result, but not with the proper parallel efficiency. Asking a class to solve this conundrum will usually lead to at least one student suggesting splitting processes in odd and even subsets.

3.5.3 Back to data exchange

The foregoing detours into the behaviour of two-sided send and receive calls were necessary, but they introduced asymmetric behaviour in the processes. We return to the averaging operation given above, and with it to a code that treats all processes symmetrically. In particular, we argue that, except for the first and last one processor, each process exchanges information with its left and right neighbour.

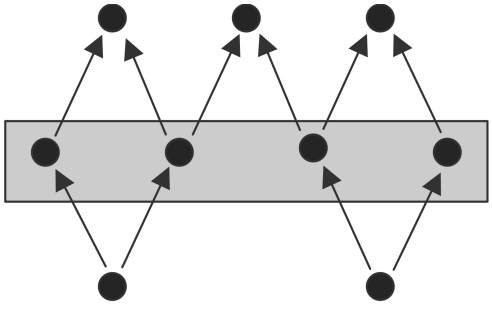
This could be implemented with blocking sends and receive calls, but students recognize how this could be somewhere between tedious and error-prone. Instead, to prevent deadlock and serialization as described above, we now offer the `MPI_Sendrecv` routine. Students are asked to implement the classroom exercise above with the sendrecv routine. Ideally, they use timing or tracing to gather evidence that no serialization is happening.

As a non-trivial example (in fact, this takes enough programming that one might assign it as an exam question, rather than an exercise during a workshop) students can now implement a swap-sort algorithm using `MPI_Sendrecv` as the main tool. For simplicity they can use a single array element per process; if each process has a subarray one has to make sure their solution has the right parallel complexity. It is easy to make errors here and implement a correct algorithm that, however, performs too slowly.

Note that students have at this point not done any serious exercises with the blocking communication calls, other than the ping-pong. No such exercises will in fact be done.

3.6 Non-blocking sends

Non-blocking sends are now introduced as the solution to a specific problem: the above schemes required paired-up processes, or careful orchestration of send and receive sequences. In the case of irregular communications



this is no longer possible or feasible. Life would be easy if we could declare ‘this data needs to be sent’ or ‘these messages are expected’, and then wait for these messages

collectively. Given this motivation, it is immediately clear that multiple send or receive buffers are needed, and that requests need to be collected.

Implementing the three-point averaging with non-blocking calls is at this point an excellent exercise.

3.7 Taking it from here

At this point various advanced topics can be discussed. For instance, Cartesian topologies can be introduced, extending the linear averaging operation to a higher dimensional one. Subcommunicators can be introduced to apply collectives to rows and columns of a matrix. The recursive matrix transposition algorithm is also an excellent application of subcommunicators.

However, didactically these topics do not require the careful attention that the introduction of the basic concepts needs, so we will not go into further detail here.

4 Summary

In this paper we have introduced a non-standard sequence for presenting the basic mechanisms in MPI. Rather than starting with sends and receives and building up from there, we start with mechanisms that emphasize the inherent symmetry between processes in the SPMD programming model. This symmetry requires a substantial shift in mindset of the programmer, and therefore we target it explicitly.

Comparing our presentation as outlined above to the standard presentation, we recognize the downplaying of the blocking send and receive calls. While students learn these, and in fact learn them before other send and receive mechanisms, they will recognize the dangers and difficulties in using them, and will have the combined sendrecv call as well as non-blocking routines as standard tools in their arsenal.

References

[1] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[2] Y. Ben-David Kolikant. Gardeners and cinema tickets: High schools' preconceptions of concurrency. *Computer Science Education*, 11:221–245, 2001.

[3] Saeed Dehnadi, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. In *Psychology of Programming Interest Group PPIG 2009*, pages 1–13. University of Limerick, Ireland, 2009.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN-10: 0131532715, ISBN-13: 978-0131532717.

[5] MPI forum: MPI documents. `http://www.mpi-forum.org/docs/docs.html`.

[6] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, Volume 1, The MPI-1 Core*. MIT Press, second edition edition, 1998.

[7] P.C. Wason and P.N. Johnson-Laird. *Thinking and Reasoning*. Harmondsworth: Penguin, 1968.