Introduction to Parallel Programming

Stéphane Zuckerman

Laboratoire ETIS
Université Paris-Seine, Université de Cergy-Pontoise, ENSEA, CNRS
F95000, Cergy, France

October 19, 2018

Outline

Resources

Resources I

MPI: Standards and specifications

- Graham 2009; Dongarra et al. 1995; Forum 1994
- Useful books:
  - *Using MPI* [Gropp, Lusk, and Skjellum 1999; Gropp, Lusk, and Skjellum 2014] (latest: 3rd edition),
  - *Using MPI-2* [Gropp, Lusk, and Thakur 1999]
  - *Using Advanced MPI* [Gropp, Hoefler, et al. 2014]
- http://mpi-forum.org
- http://www.mcs.anl.gov/mpi

Resources II

Available implementations

- Mpich-2: https://www.mpich.org/
- Open-MPI: https://www.open-mpi.org/
- Most supercomputer vendors provide their own implementation, often derived from either Mpich or Open-MPI
- Personal preference: Open-MPI (more modular; more recent–benefits from design issues found in Mpich)

A short history of parallel programming and why we need it

In the beginning. . . I

Why parallel computing?

- Parallel programming appears very early in the life of computing.
- For every generation of high-end processor, some computations hog all of the available resources.
- Solution: duplicate computing resources.

In the beginning. . . II

Two (non-exclusive) approaches

- Design a parallel processor/computer architecture, *i.e.*, duplicate functional units, provide vector units, . . . :
  - Cray I vector supercomputer
  - Connection Machine
- Design a computer made of multiple computing nodes (also called compute nodes):
  - Custom clusters: SGI Altix, IBM BlueGene, . . .
  - Commodity supercomputers: Beowulf clusters

In the beginning. . . III

Toward standardization of parallel computing

Until the early/mid 1990s, each supercomputer vendor provides their own parallel computing tools.

- Each new parallel computer requires to learn a new or updated environment to get good performance
- Terrible for portability and productivity

The time was ripe for a standardization effort for all types of parallelism.

In the beginning. . . IV

Toward standardization of parallel models of computation

- Distributed memory models:
  - PVM (1991)
  - MPI standard (1992)
- Shared memory models:
  - POSIX.1c / IEEE Std 1003.1c-1995 — *a.k.a.* PTHREAD library
  - OpenMP standard (1997)

The obvious benefits are portability and productivity, although *performance portability* is not guaranteed (only correctness).

Why should we care about these parallel models? I

Hardware context

- End of Dennard's scaling
- Moore's law now used to add more computing units on a single chip (instead of going to higher frequencies)
- Programming chip multiprocessors (CMP) is not just for scientific/high-performance computing anymore
    - Embedded chips require programming models and execution models to efficiently exploit all of the hardware

Why should we care about these parallel models? II

Software context – embedded systems

A lot of embedded systems are mainstream and *general purpose* nowadays

- *e.g.*, Raspberry PI, Beagle, *etc.*
- They feature CMPs such as ARM multicore chips
- Even on more specialized platforms, MPI, OpenMP, or OpenCL implementations exist:
  - Xilinx: "High-Level Synthesis" (HLS) in FPGAs with OpenCL
  - Adapteva's Parallella board:
    - Zynq-7000 = dual core Cortex A9+FPGA SoC
    - Epiphany co-processor (16 cores with scratchpads)
  - There are OpenMP and MPI implementations for both ARM and Epiphany boards.
  - Bottom line: "general purpose" parallelism is made available to all parallel platforms nowadays

Why should we care about these parallel models? III

Advantages of traditional programming and execution models

- Because these models are standardized, they are made available in mainstream programming tools
  - OpenMP and MPI both have free/open source implementations available to all
  - Same with PGAS languages
  - Same with GPU-oriented languages
  - Performance goes from "acceptable" to "pretty good"
    - . . . but proprietary implementations tend to be faster because they have better/exclusive knowledge of underlying system software and/or hardware
    - . . . but not always!

Message-passing execution models

An introduction to MPI
Execution Model

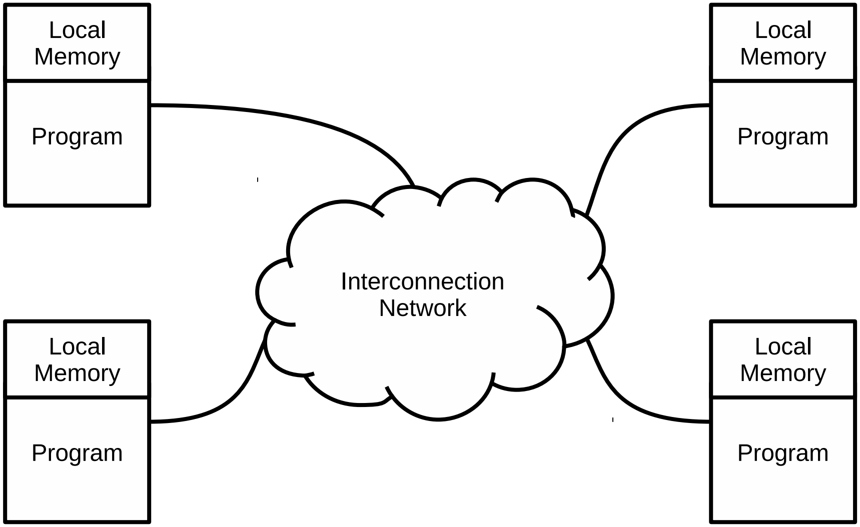
Execution Model

- Relies on the notion of distributed memory
- All data transfers between MPI *processes* are explicit
- Processes can also be synchronized with each other
- Achieved using a library API

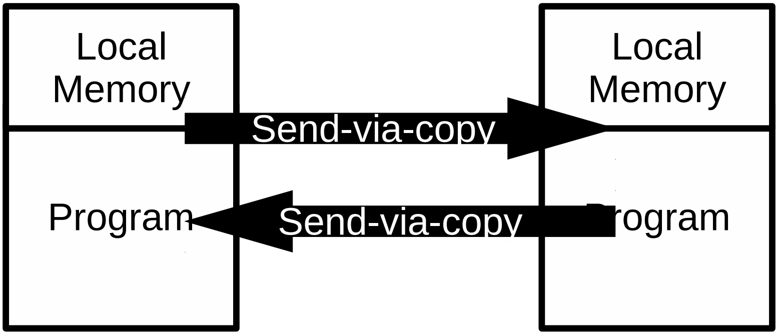
MPI process $\neq$ UNIX or Windows process.

- A process $=$ a program counter $+$ a (separate) address space
- An MPI process could be implemented as a thread [Huang, Lawlor, and Kale 2003]

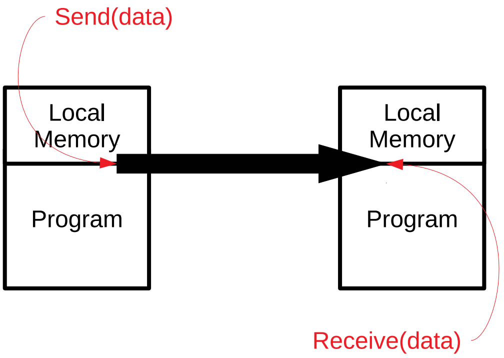
MPI execution model in a nutshell I



MPI execution model in a nutshell II



MPI execution model in a nutshell III I



MPI-1

- Basic I/O communication functions (100+)
- Blocking send and receive operations
- Nonblocking send and receive operations
- *Blocking* collective communications
  - Broadcast, scatter, gather, *etc.*
  - Important for performance
- Datatypes to describe data layout
- Process topologies (use of communicators, tags)
- C, C++, Fortran bindings
  - In practice, "unofficial" bindings exist for many languages: Python, Ruby, ...
- Error codes and classes

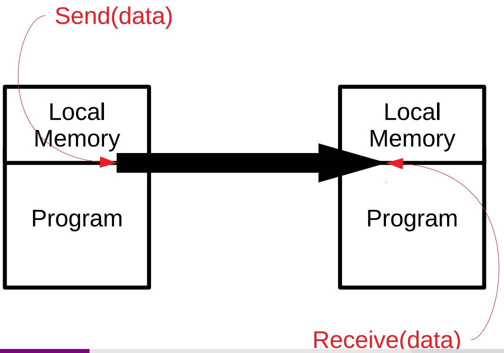
MPI-2 and beyond

- MPI-2 (2000):
  - Thread support
  - MPI-I/O: efficient (parallel) file I/O
  - R-DMA: remote direct memory access—asynchronous memory accesses, possibly on memory-mapped I/Os
- MPI-2.1 (2008) and MPI-2.2 (2009):
  - Corrections to standard, small additional features
- MPI-3 (2012):
  - Lots of new features to standard (briefly discussed at the end)

MPI Basics

Stuff needed by the MPI implementation from application

- How to compile and run MPI programs
- How to identify processes
- How to describe the data



Compiling and running MPI programs

MPI is a library

Need to use function calls, to leverage MPI features.

Compilation

- Regular compilation: use of `cc`, *e.g.*, `gcc -o test test.c`
- MPI compilation: `mpicc -o test test.c`

Execution

- Regular execution: `./test`
- MPI execution: `mpiexec -np 16 ./test`

MPI process identification

MPI groups

- Each MPI process belongs to one or more groups
- Each MPI process is given one or more colors
- Group+color = *communicator*
- All MPI processes belong to `MPI_COMM_WORLD` when the program starts

Identifying individual processes: ranks

- If a process belongs to two different communicators, its rank may be different from the point of view of each communicator.

Most basic MPI program

Hello World

```
#include <mpi.h> // required to use MPI functions
#include <stdio.h>

int main(int argc, char* argv[]) {
    int rank, size;

//  must ALWAYS be called to run an MPI program
    MPI_Init(&argc, &argv);
//  get process rank/id
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
//  get total number of processes in communicator
    MPI_Comm_size(MPI_COMM_WOLRD, &size);

    printf("I␣am␣process␣#%d/%d\n", rank, size);

//  must ALWAYS be called to run an MPI program
    MPI_Finalize();
    return 0;
}
```

Basic data transfers

MPI_Send

Syntax:

```
int MPI_Send ( const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm );
```

- `buf`: the buffer from where to read the data to send
- `count`: the number of elements to send over the network link
- `datatype`: the type of data that is being sent, *e.g.*, `MPI_CHAR`, `MPI_INT`, `MPI_DOUBLE`, *etc.*
- `dest`: which process is meant to receive the data (identified by its rank)
- `tag`: a way to discriminate between various messages sent to the same process rank
- `comm`: the communicator (or “group of tasks”) to target

Basic data transfers

MPI_Recv

Syntax: `int MPI_Recv (const void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status);`

- buf: the buffer from where to write the data to be read
- count: the number of elements to receive over the network link
  - count can be bigger than what was received in practice (the real count can be obtained using MPI_Get_count)
  - If count is smaller than what is being sent, an error occurs
- datatype: the type of data that is being received, *e.g.*, MPI_CHAR, MPI_INT, MPI_DOUBLE, *etc.*
- dest: from which process the data originates (identified by its rank)
- tag: a way to discriminate between various messages sent to the same receiving process rank
- comm: the communicator (or "group of tasks") to target
- status: contains the source of the message, the tag, how many elements were sent.

Basic data transfers I

Wildcards & status

Receive wildcards

- `MPI_ANY_SOURCE`: accepts data from any sender
- `MPI_ANY_TAG`: accepts data with any tag (as long as the receiver is a valid target)

Basic data transfers II
Wildcards & status

Status object

Objects of type `MPI_Status` have the following accessible fields (assume our object name is `status`):

- `MPI_SOURCE`: the rank of the process which sent the message (useful when using `MPI_ANY_SOURCE`)
- `MPI_TAG`: the tag used to identify the received message (useful when using `MPI_ANY_TAG`)
- `MPI_ERROR`: the error status (assuming the MPI program does not crash when an error is detected—which is the behavior by default).

To get the number of elements received, the user can query `status` using the `MPI_Get_count` function.

A simple example to send and receive data

```
#include <mpi.h> // required to use MPI functions
#include <stdio.h>

int main(int argc, char* argv[]) {
  int rank, data[100];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0)
    MPI_Send(data,100,MPI_INT,1,0,MPI_COMM_WORLD);
  else
    MPI_Recv(data,100,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

MPI is simple

- `MPI_Init`
- `MPI_Comm_rank`
- `MPI_Comm_size`
- `MPI_Send`
- `MPI_Recv`
- `MPI_Finalize`

…are enough to write any application using message passing.
However, to be *productive* and ensure reasonable *performance portability*, other functions are required.

MPI pragmatics

Building and running MPI programs

Building MPI programs

- C: `mpicc`
- C++: `mpicxx`
- Fortran: `mpif77` (Fortran 77) or `mpif90` (Fortran 90)

Running MPI programs

- `mpiexec -np 16 ./test`
  - ...will run the program `test` on 16 MPI processes.
- `mpiexec -host` *`h1,h2,...`* `-np 16 ./test`
  - ...will run the program `test` on the various hosts specified on the command line in a round-robin fashion
  - In our example, host `h1` will receive MPI processes 0,2,4,6.

Note: `mpiexec` or `mpirun` can be used interchangeably (they are aliases).

Non-blocking communications

Limits to `MPI_Send` and `MPI_Recv`

`MPI_Send` and `MPI_Recv` are blocking communication calls

- The sending process must wait until the data it is sending has been received
- The receiving process must block once it has initiated the receiving operation
- Consequence: data sent or received through blocking communications is *safe to (re)use*
- However, this can severely hamper the overall performance of an application

Non-blocking variants: `MPI_Isend` and `MPI_Irecv`

- Routine returns immediately – completion has to be tested separately
- Primarily used to overlap computation and communication

Non-blocking communications I
Syntax

API

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Irecv(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

Non-blocking communications II
Syntax

Properties

- Non-blocking operations allow overlapping of computation and communication
- Completion can be tested using `MPI_Test(MPI_Request *request, int flag, MPI_Status *status)`
- Anywhere one uses `MPI_Send` or `MPI_Recv`, one can use `MPI_Isend`/`MPI_Wait` or `MPI_Irecv`/`MPI_Wait` pairs instead
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

Non-blocking communications III
Syntax

Multiple completions

- `int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)`
- `int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)`
- `int MPI_Waitsome(int count, MPI_Request *array_of_requests, int *array_of_indices, MPI_Status *array_of_status)`

There are corresponding versions of `MPI_Test` for each of those.

A simple example to use non-blocking communications

```
#include <mpi.h>
// ...
int main(int argc, char* argv[]) {
//     ...
   if (rank == 0) {
     for (i=0; i< 100; i++) {
       /* Compute each data element and send it out */
       data[i] = compute(i);
       MPI_ISend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                 &request[i]);
     }
     MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
   } else {
     for (i = 0; i < 100; i++)
       MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
   }
//   ...
}
```

Collective operations

Introduction

- Collective operations are called by all processes belonging to the same communicator
- `MPI_Bcast` distributes data from one process (the root) to all others in a communicator
- `MPI_Reduce` combines data from all processes in the communicator and returns it to one process
- In many (numerical) algorithms, send/receive pairs can be replaced by broadcast/reduce ones
  - Simpler and more efficient

Properties

- Tags are not used; only communicators matter.
- Non-blocking collective operations were added in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

Synchronization

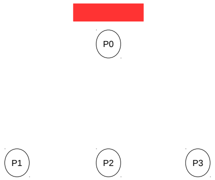
```
int MPI_Barrier(MPI_Comm *comm)
```

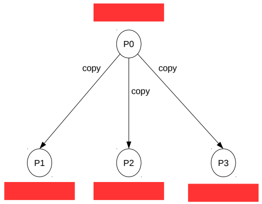
- Blocks until all processes belonging to communicator `comm` call it
- No process can get out of the barrier unless all the other processes have reached it

Collective data movements

Broadcast

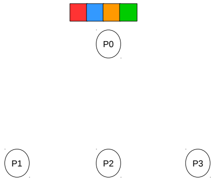
```
int MPI_Bcast (void* data, int count, MPI_Datatype datatype,
int root, MPI_Comm communicator);
```

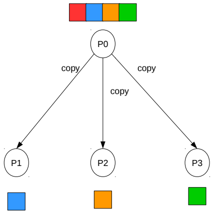




Collective data movements
Scatter

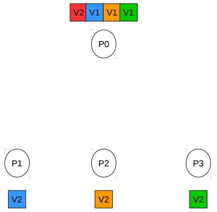
```
int MPI_Scatter (void* send_data, int send_count,
MPI_Datatype send_datatype, void* recv_data, int recv_count,
MPI_Datatype recv_datatype, int root, MPI_Comm
communicator);
```

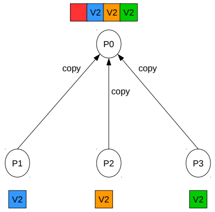




Collective data movements
Gather

```
int MPI_Gather (void* send_data, int send_count,
MPI_Datatype send_datatype, void* recv_data, int recv_count,
MPI_Datatype recv_datatype, int root, MPI_Comm
communicator);
```





Collective data movements
Other MPI Calls

- `int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);`
- `int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);`

Collective computations

- `int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);`
- `int MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);`

Other MPI Collective routines

- Other useful collective operations
  - `MPI_Allgatherv`
  - `MPI_Alltoallv`
  - `MPI_Gatherv`
  - `MPI_Scatterv`
  - `MPI_Reducescatter`
- "All" versions deliver results to all participating processes
- "v" versions (stands for *vector*) allow the chunks to have different sizes
  - Important when the number of processes involved in the computation is no a multiple of the number of data elements
- `MPI_Allreduce`, `MPI_Reduce`, `MPI_Reducescatter`, and `MPI_Scan` take both built-in and user-defined combiner functions.

Built-in collective computation operations

`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	Product
`MPI_SUM`	Sum
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise exclusive or
`MPI_MAXLOC`	Maximum and location
`MPI_MINLOC`	Minimum and location

Example using collective operations I

```
#include <mpi.h>
#include <math.h>
int main(int argc, char* argv[]) {
  const double g_PI25DT = 3.141592653589793238462643;
  double   mypi, pi, h, sum, x, a;
  int      n, myid, numprocs, i, ierr;

  xMPI_Init(&argc, &argv);
  xMPI_Comm_rank(MPI_COMM_WORLD, &myid);
  xMPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  for (;;) {
    if (myid == 0) {
      printf("Enter␣the␣number␣of␣intervals:␣(0␣quits)␣");
      fflush(stdout);
      scanf("%f", &n);
      if (n <= 0) break;
    }

    xMPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n < 0) { xMPI_Finalize(); exit(0); }
```

Example using collective operations II

```
    h = 1.0/n; sum = 0.0;
    for (int i = myid+1; i < n; i+=numprocs) {
      x = h * ((double)i-0.5);
      sum += (4.0 / (1.0 * x*x));
    }
    mypi = h*sum;

    xMPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

    if (myid == 0)
      printf("pi␣is␣%f.␣Error␣is␣%f\n", pi, fabs(pi-g_PI25DT));
  }
  xMPI_Finalize();
  return 0;
}
```

References

References

Forum, Message Passing (1994). *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA.

Dongarra, Jack J. et al. (1995). *An Introduction to the MPI Standard*. Tech. rep. Knoxville, TN, USA.

Gropp, William, Ewing Lusk, and Anthony Skjellum (1999). *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

Gropp, William, Ewing Lusk, and Rajeev Thakur (1999). *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press. ISBN: 0262571331.

Huang, Chao, Orion Lawlor, and Laxmikant V Kale (2003). "Adaptive mpi". In: *International workshop on languages and compilers for parallel computing*. Springer, pp. 306–322.

Graham, Richard L. (2009). "The MPI 2.2 Standard and the Emerging MPI 3 Standard". In: *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Espoo, Finland: Springer-Verlag, pp. 2–2. ISBN: 978-3-642-03769-6. DOI:

10.1007/978-3-642-03770-2_2. URL: http://dx.doi.org/10.1007/978-3-642-03770-2_2.

Gropp, William, Torsten Hoefler, et al. (2014). *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press. ISBN: 0262527634, 9780262527637.

Gropp, William, Ewing Lusk, and Anthony Skjellum (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press. ISBN: 0262527391, 9780262527392.