SPMD Model

Victor Eijkhout

2026 PCSE

Overview

In this section you will learn how to think about parallelism in MPI.

Commands learned:

- *MPI_Init*, *MPI_Finalize*,
- *MPI_Comm_size*, *MPI_Comm_rank*
- *MPI_Get_processor_name*,

Practicalities

Compiling

MPI compilers are usually called *mpicc*, *mpif90*, *mpicxx*.

These are not separate compilers, but scripts around the regular C/Fortran compiler. You can use all the usual flags.

```
$ mpicc -show
icx -I/intel/include/stuff -L/intel/lib/stuff -Wwarnings # et cetera
```

(Python of course no compilation needed)

(For CMake see slide **??**.)

Running (at TACC)

In your batch job indicate node and core count

```
1 #SBATCH -N 4
2 #SBATCH -n 200
3 ibrun yourprog
```

No specification on `ibrun` needed!

Running (in general)

On non-TACC machines:

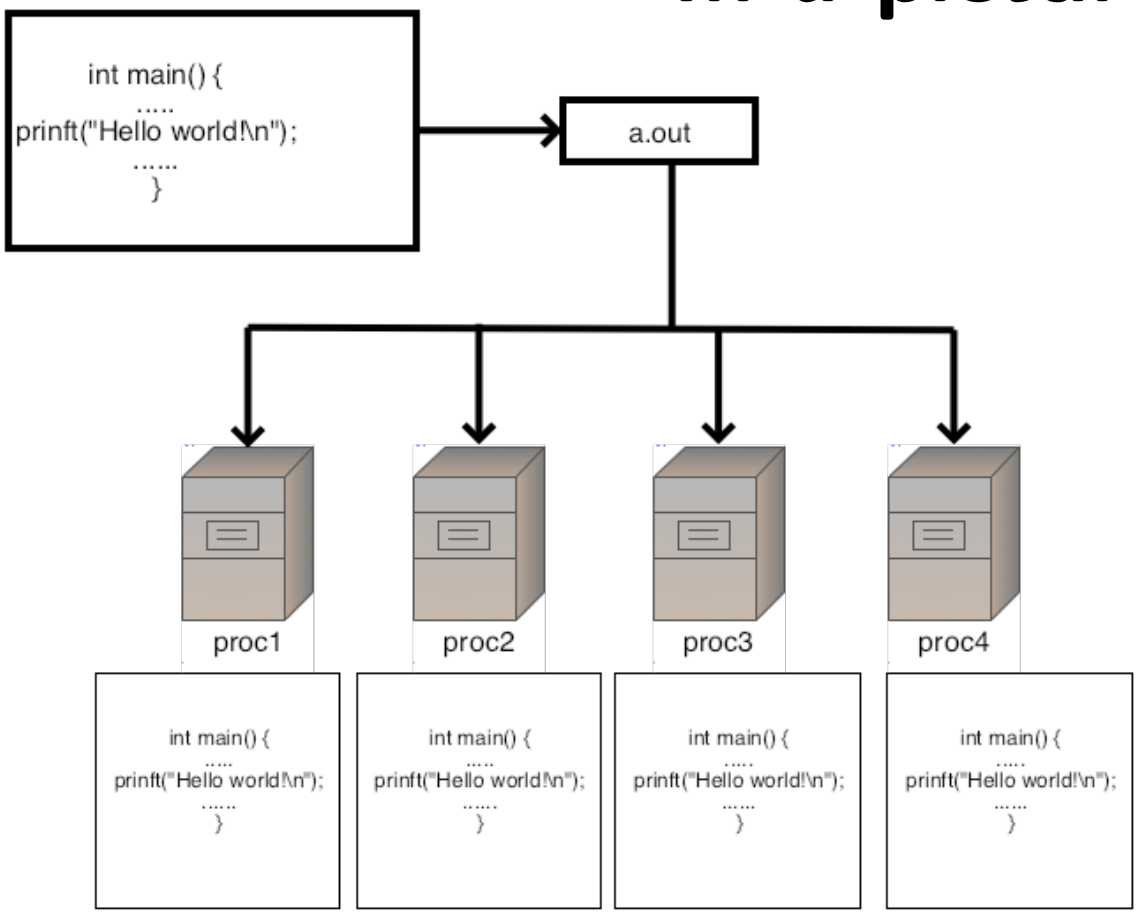
```
mpiexec -n 4 hostfile ... yourprogram arguments
mpirun -np 4 hostfile ... yourprogram arguments
```

Exercise 1 `hello`

Write a 'hello world' program, without any MPI in it, and run it in parallel with *`mpiexec`* or your local equivalent. Explain the output.

1. In the directories `exercises-mpi-xxx` do `make hello` to compile;
2. do `ibrun hello` to execute.

In a picture


int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}

The MPI worldview: SPMD

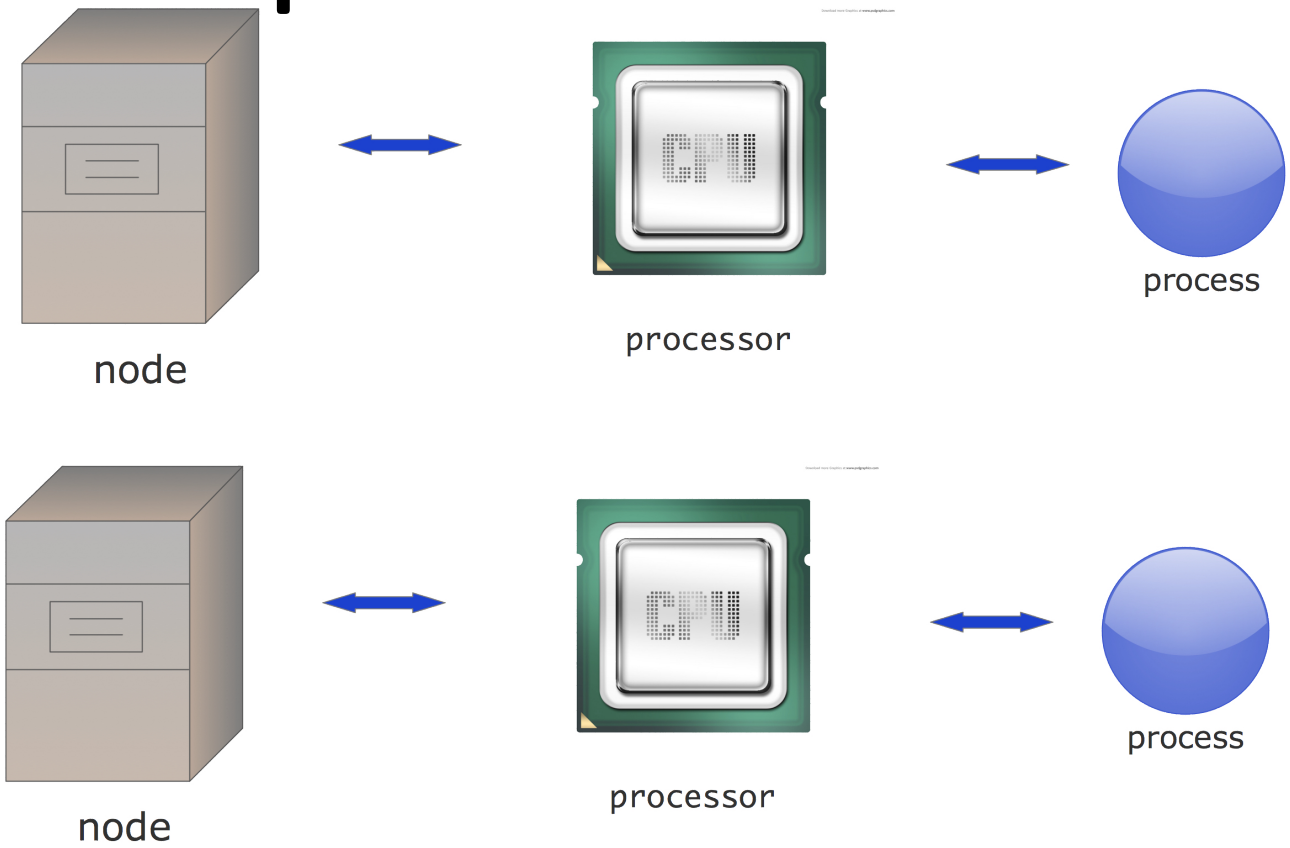
SPMD

The basic model of MPI is
'Single Program Multiple Data':
each process is an instance of the same program.

Symmetry: There is no 'master process', all processes are equal, start and end at the same time.

Communication calls do not see the cluster structure:
data sending/receiving is the same for all neighbors.

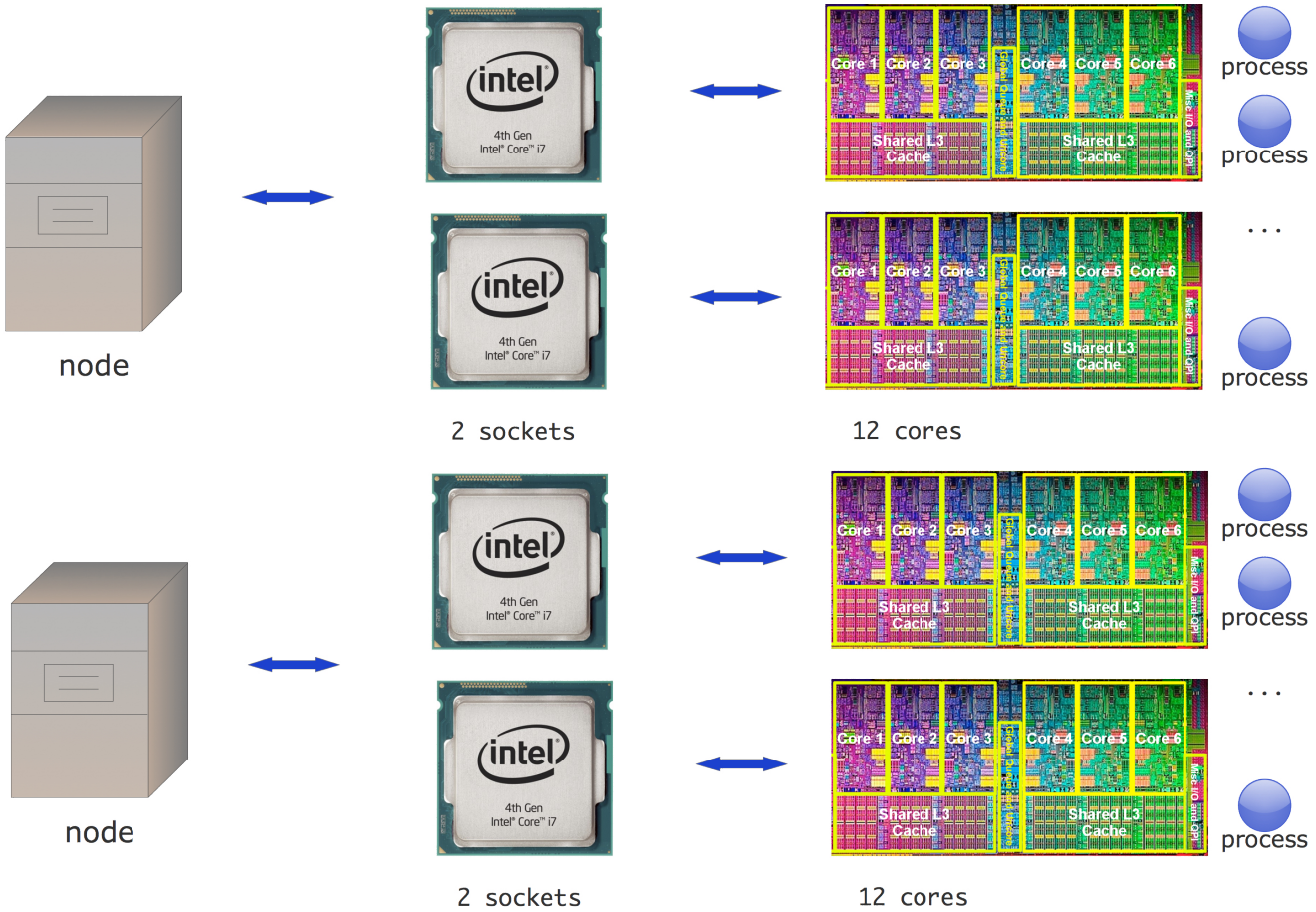
Computers when MPI was designed



One processor and one process per node;
all communication goes through the network.

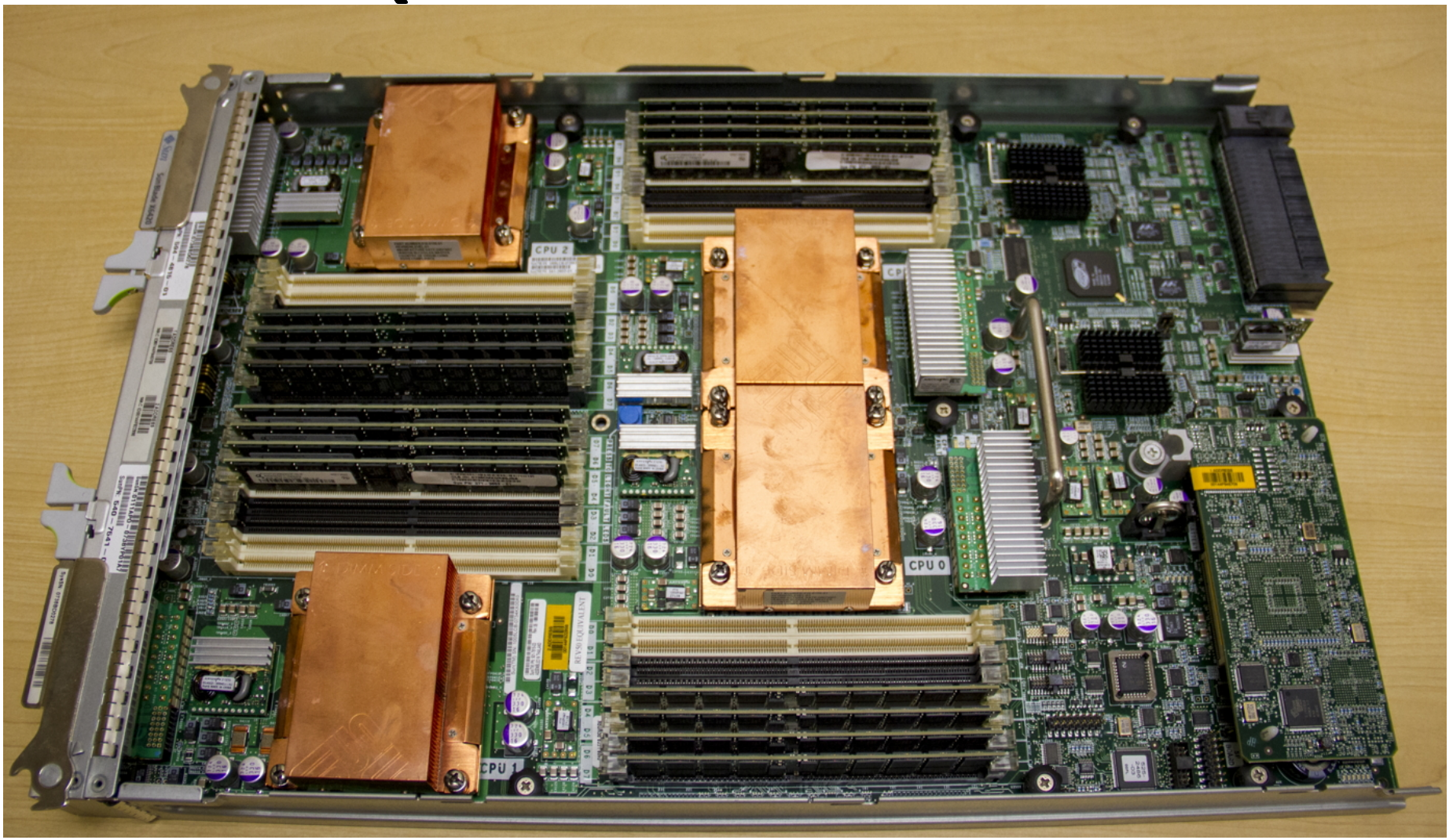
$\Rightarrow$ process model, no data sharing.

Pure MPI

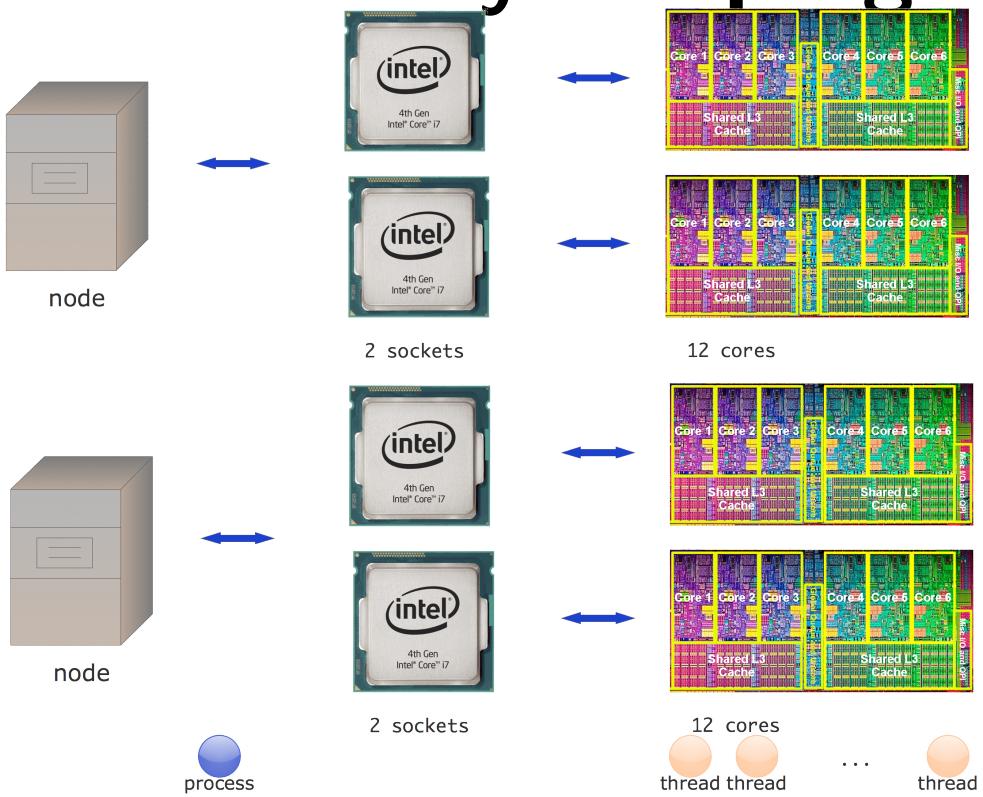


A node has multiple sockets, each with multiple cores.
Pure MPI puts a process on each core: pretend shared memory doesn't exist.

Quad socket node



Hybrid programming



Hybrid programming puts a process per node or per socket;
further parallelism comes from threading.
Not in this course. . .

Terminology

'Processor' is ambiguous: is that a chip or one independent instruction processing unit?

- Socket: the processor chip
- Processor: we don't use that word
- Core: one instruction-stream processing unit
- Process: preferred terminology in talking about MPI.

Do I need a supercomputer?

- With `mpiexec` and such, you start a bunch of processes that execute your MPI program.
- Does that mean that you need a cluster or a big multicore?
- No! You can start a large number of MPI processes, even on your laptop. The OS will use 'time slicing'.
- Of course it will not be very efficient...

Installing your own MPI

It is convenient to do MPI development on your laptop/desktop.

- Use a package manager
  Apple: brew or macports
  Linux: yum, aptget, ...
  Windows: I'll have to get back to you on that
- ... or download and compile from source `mpich.org`

We start learning MPI!

About library calls and bindings

Bindings

The standard defines interfaces to MPI from C and Fortran.
These look very similar; sometimes we will only show the C variant.

MPI can also be used from C++ and Python

MPI headers: C

You need an include file:

```
1 #include "mpi.h"
```

This defines all routines and constants.

C++ bindings, MPL

MPI-1 had C++ bindings, by MPI-2 they were deprecated, in MPI-3 they have been removed.

- Easy solution: use the C bindings unaltered.
  This is done in the `cxx` exercise directory; Ugly: very un-OO.
- There are private projects for C++ bindings.
  In particular MPL: `https://github.com/rabauke/mpl`
  (Exercises in `mpl` directory.)
  Very modern OO, Header-only
  Not a full MPI implementation: (I/O and one-sided mostly missing)

Use of MPL

In your program:

```
#include <mpl/mpl.hpp>
```

Compiling:

```
mpicxx -o prog sources.cxx -I${TACC_MPL_INC}
```

TACC: `module load mpl`

MPI Init / Finalize

These calls need to be around the MPI part of your code:

```
MPI_Init(&argc,&argv); // zeros allowed
// your code
MPI_Finalize();
```

This is not a 'parallel region'.
Only internal library initialization:
allocate buffers, discover network, ...

MPL init/finalize

No explicit init/finalize:

- init is done by the first command that needs it
- finalize in some destructor.

Exercise 2 `hello`

Add the commands *`MPI_Init`* and *`MPI_Finalize`* to your code. Put three different print statements in your code: one before the init, one between init and finalize, and one after the finalize. Again explain the output.

Ranks

Process identification

- Processes are organized in 'communicators'.
- For now only the 'world' communicator
- Each process has a unique 'rank' wrt the communicator.

```
int MPI_Comm_size( MPI_Comm comm, int *nprocs )
int MPI_Comm_rank( MPI_Comm comm, int *procno )
```

Lowest number is always zero.

This is a logical view of parallelism: mapping to physical processors/cores is invisible here.

World communicator

For now, the communicator will be *MPI_COMM_WORLD*.

C:

```
MPI_Comm comm = MPI_COMM_WORLD;
```

Fortran:

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
! legacy
Integer :: comm = MPI_COMM_WORLD
```

Python:

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
```

MPL:

```
1     const mpl::communicator &comm_world =
2         mpl::environment::comm_world();
```

MPI_Comm_size

Name	Param name	Explanation	C type	F type
MPI_Comm_size (				
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	size	number of processes in the group of comm	int*	INTEGER
	)			

MPL: MPI_Comm_size

```
1 int mpl::communicator::size ( ) const
```

MPI_Comm_rank

Name	Param name	Explanation	C type	F type
MPI_Comm_rank (				
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	rank	rank of the calling process in group of comm	int*	INTEGER
	)			

MPL: MPI_Comm_rank

```
1 int mpl::communicator::rank ( ) const
```

About routine signatures: C/C++

Signature:

```
int MPI_Comm_size(MPI_Comm comm,int *nprocs)
```

Use:

```
1 MPI_Comm comm = MPI_COMM_WORLD;
2 int nprocs;
3 int errorcode;
4 errorcode = MPI_Comm_size( comm,&nprocs );
```

But forget about that error code most of the time:

```
MPI_Comm_size( comm,&nprocs );
```

About routine signatures: MPL

Signature:

```
int mpl::communicator::size ( ) const
```

Use

```
1 const mpl::communicator &comm_world = mpl::environment::comm_world();
2 int nprocs, procno;
3 nprocs = comm_world.size();
```

Exercise 3 `commrank`

Write a program where each process prints out a message reporting its number, and how many processes there are:

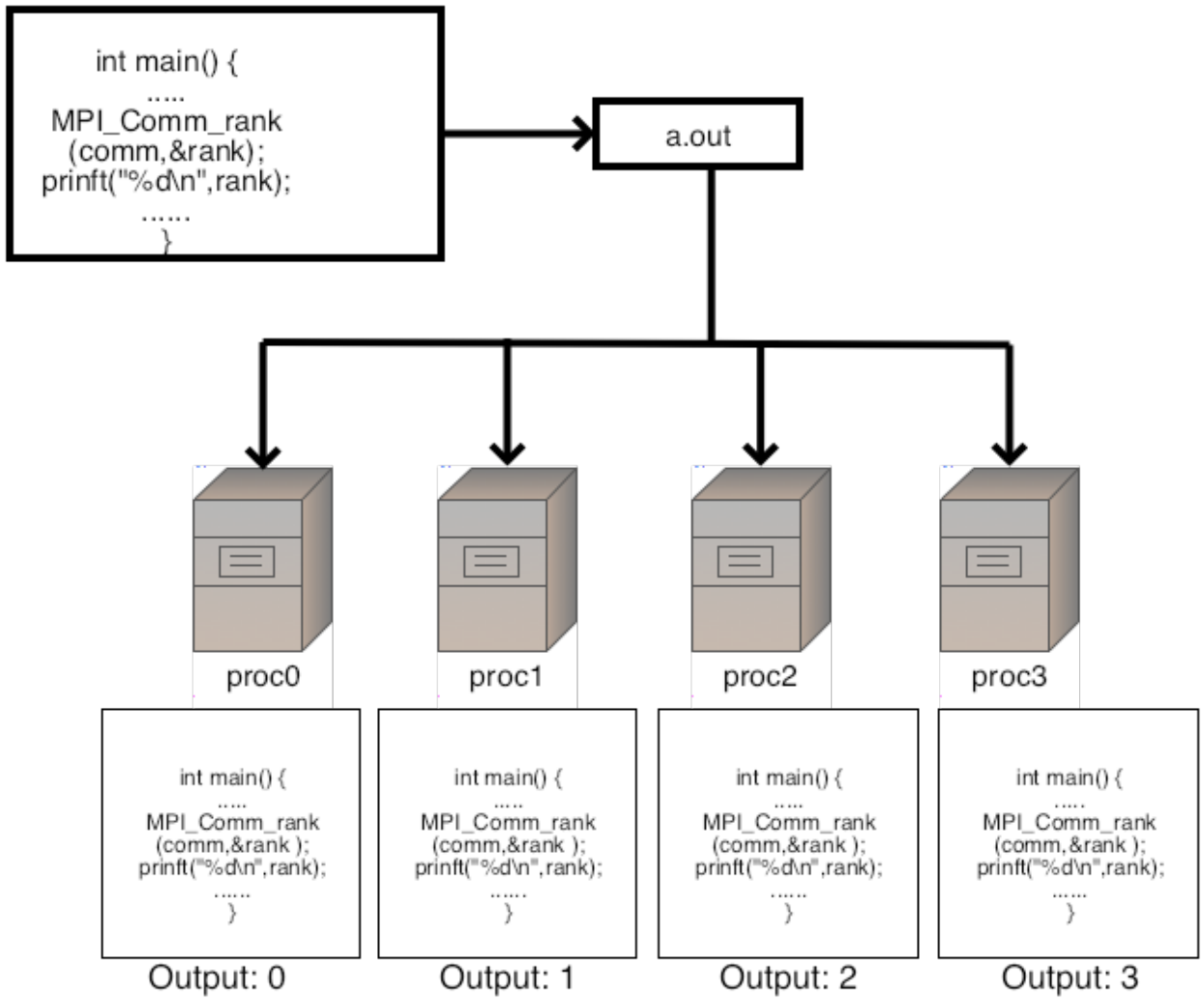
```
Hello from process 2 out of 5!
```

Write a second version of this program, where each process opens a unique file and writes to it.

Exercise 4 `commrank`

Write a program where only the process with number zero reports on how many processes there are in total.

Illustration



About errors

MPI routines invoke an error handler; (slide **??**) Default action: abort

Every routine is defined as returning integer error code

- In C: function result.

```
1 ierr = MPI_Init(0,0);
2 if (ierr!=MPI_SUCCESS) /* do something */
```

  But really: can often be ignored; is ignored in this course.

```
1 MPI_Init(0,0);
```

- In Fortran: as optional (F08 only) parameter.
- MPL: throws exceptions
- In Python: throwing exception.

There's not a lot you can do with an error code:
very hard to recover from errors in parallel.
By default code bombs with (hopefully informative) message.

Processor name

MPI_Get_processor_name

Name	Param name	Explanation	C type	F type
MPI_Get_processor_name (				
	name	A unique specifier for the actual (as opposed to virtual) node.	char*	CHARACTER
	resultlen	Length (in printable characters) of the result returned in name	int*	INTEGER
	)			

MPL: MPI_Get_processor_name

Missing MPL proto: mpi˙get˙processor˙name

Exercise 5

Use the command *MPI_Get_processor_name*. Confirm that you are able to run a program that uses two different nodes.

TACC nodes have a hostname cRRR-CNN, where RRR is the rack number, C is the chassis number in the rack, and NN is the node number within the chassis. Communication is faster inside a rack than between racks!

Go to `examples/mpi/xxx` and do `make procname`, then `ibrun procname`

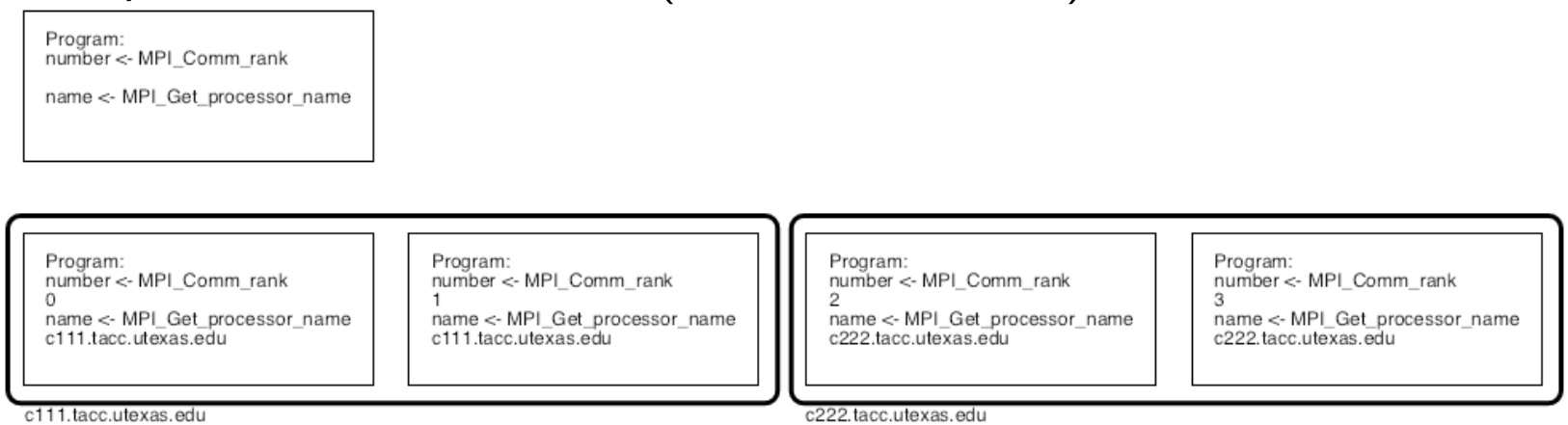
Processor name

Processes (can) run on physically distinct locations.

```
1 // procname.c
2 int name_length = MPI_MAX_PROCESSOR_NAME;
3 char proc_name[name_length];
4 MPI_Get_processor_name(proc_name,&name_length);
5 printf("Process %d/%d is running on node <<%s>>\n",
6          procid,nprocs,proc_name);
```

In a picture

Four processes on two nodes (`idev -N 2 -n 4`)



Processor name: MPL

Processes (can) run on physically distinct locations.

Code:

```
1 // procname.cxx
2 procno = comm_world.rank();
3 string procname =
4   mpl::environment::processor_name();
5 stringstream ss;
6 ss << "[" << procno << "] "
7    << " Running on: " << procname;
8 cout << ss.str() << '\n';
```

Output:

```
1 TACC:  Starting up job
      ↪6051291
2 TACC:  Starting
      ↪parallel tasks...
3 [6] Running on:
      ↪c204-031.frontera.tacc.utex
4 [7] Running on:
      ↪c204-031.frontera.tacc.utex
5 [2] Running on:
      ↪c204-029.frontera.tacc.utex
6 [4] Running on:
      ↪c204-030.frontera.tacc.utex
7 [0] Running on:
      ↪c204-028.frontera.tacc.utex
8 [3] Running on:
      ↪c204-029.frontera.tacc.utex
9 [1] Running on:
      ↪c204-028.frontera.tacc.utex
10 [5] Running on:
      ↪c204-030.frontera.tacc.utex
11 TACC:  Shutdown
```

Your first useful parallel program

Functional Parallelism

Parallelism by letting each process do a different thing.

Example: divide up a search space.

Each process knows its rank, so it can find its part of the search space.

Exercise 6 `prime`

Is the number $N = 2,000,000,111$ prime? Let each process test a disjoint set of integers, and print out any factor they find. You don't have to test all integers $< N$: any factor is at most $\sqrt{N} \approx 45,200$.

(Hint: `i%0` probably gives a runtime error.)

Can you find more than one solution?

Exercise 7

Allocate on each process an array:

```
1 int my_ints[10];
```

and fill it so that process 0 has the integers $0 \cdots 9$, process 1 has $10 \cdots 19$, et cetera.

Let each process print out its array. It may be hard to print the output in a non-messy way.