TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Tutorial on MPI programming, Full Course

Victor Eijkhout eijkhout@tacc.utexas.edu

TACC training, 2017

TACC

Justification

The MPI library is the main tool for parallel programming on a large scale. This course introduces the main concepts through lecturing and exercises.

The SPMD model

Overview

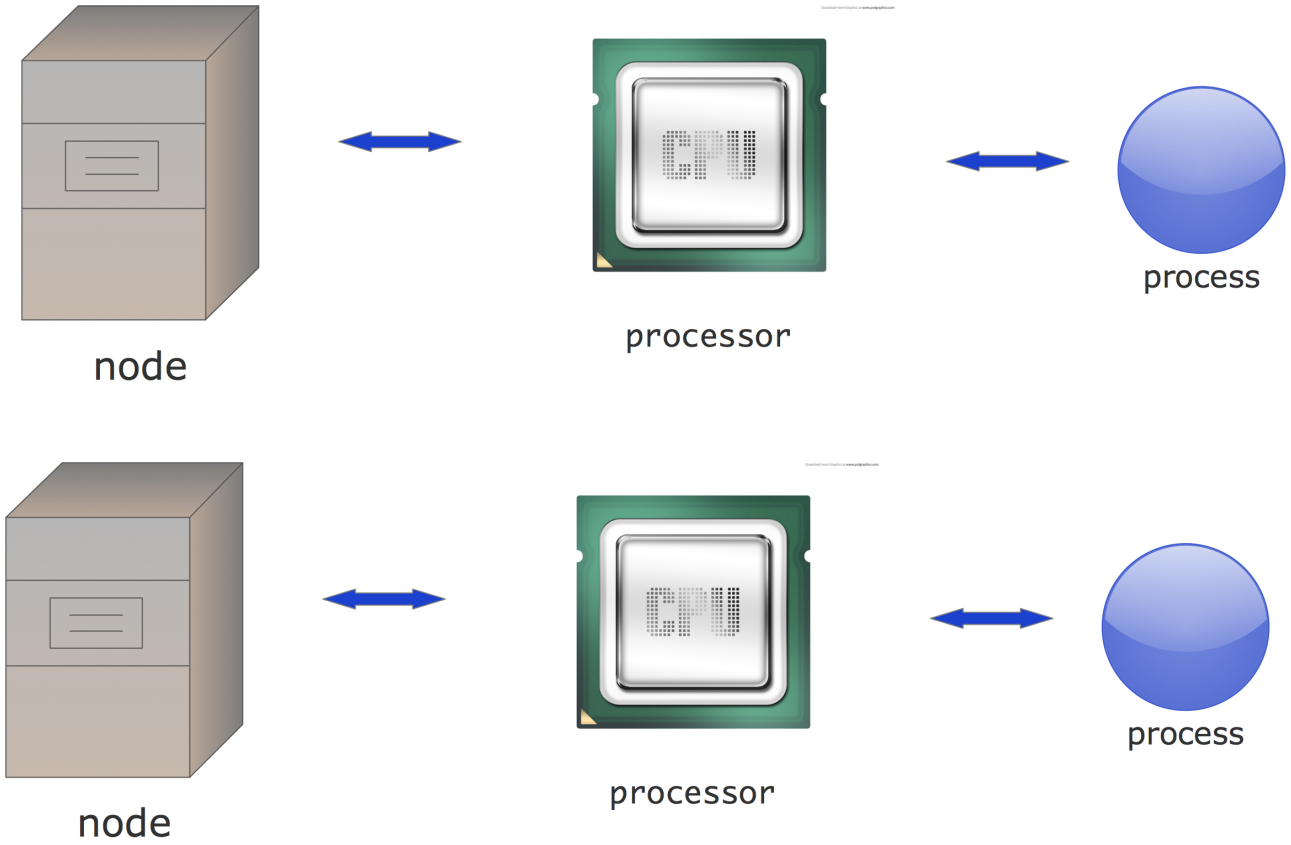
In this section you will learn how to think about parallelism in MPI.

Commands learned:

- `MPI_Init`, `MPI_Finalize`,
- `MPI_Get_processor_name`, `MPI_Comm_size`, `MPI_Comm_rank`

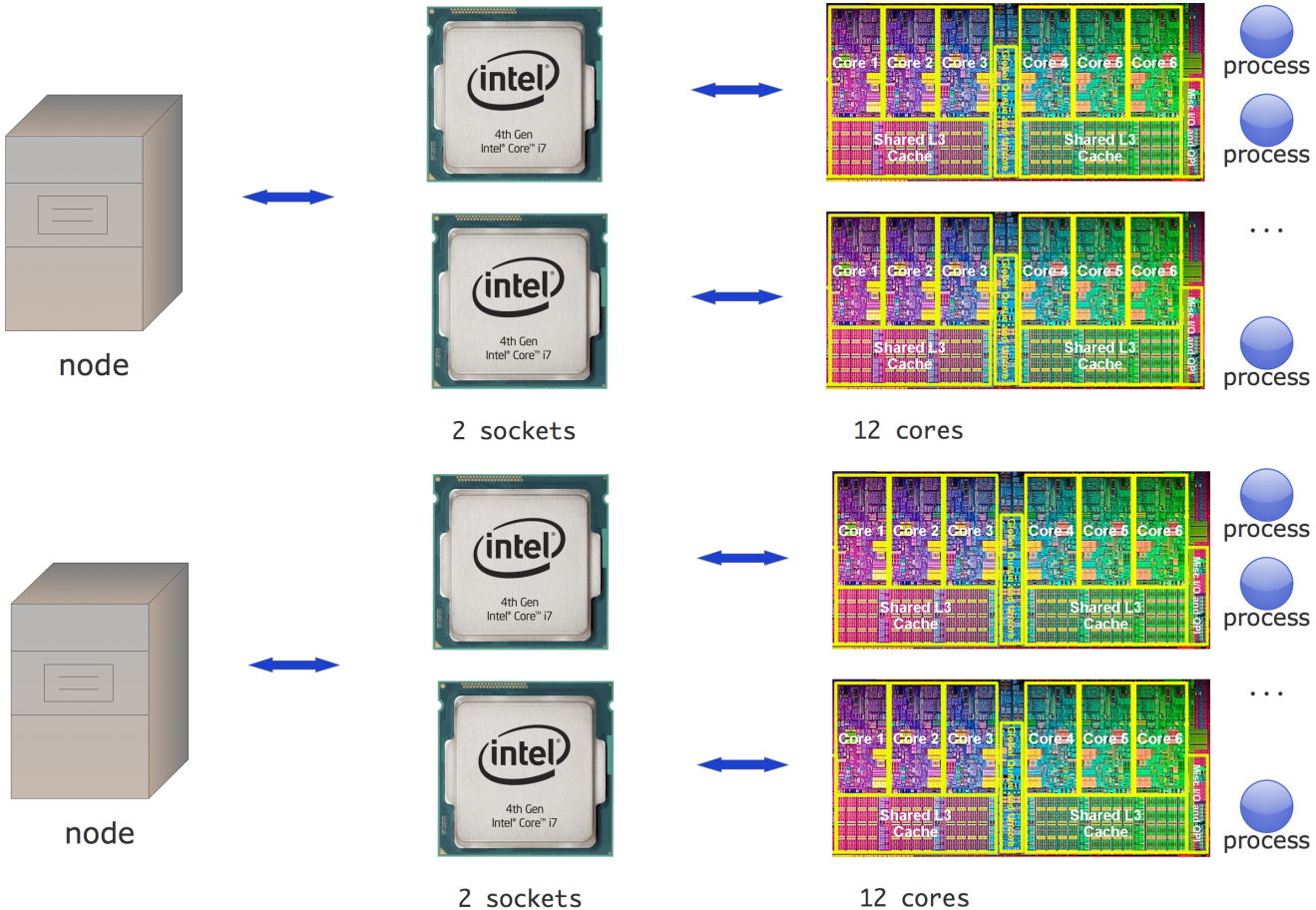
Table of Contents

Computers when MPI was designed



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

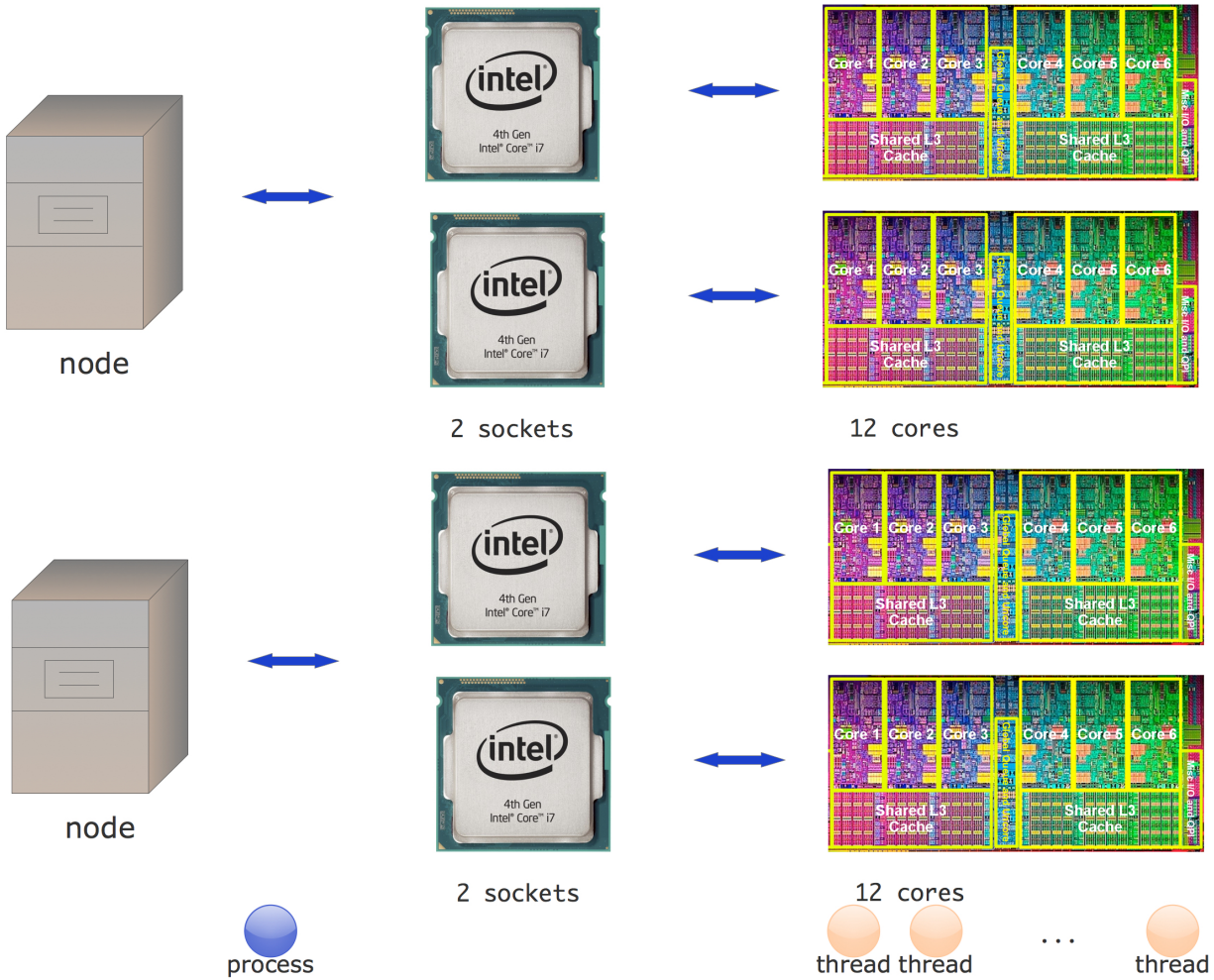
Pure MPI



A node has multiple sockets, each with multiple cores.

Pure MPI puts a process on each core: pretend shared memory doesn't exist.

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.

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 neighbours.

Table of Contents

Compiling and running

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.

Run your program with something like

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

At TACC:

```
ibrun yourprog
```

the number of processes is determined by SLURM.

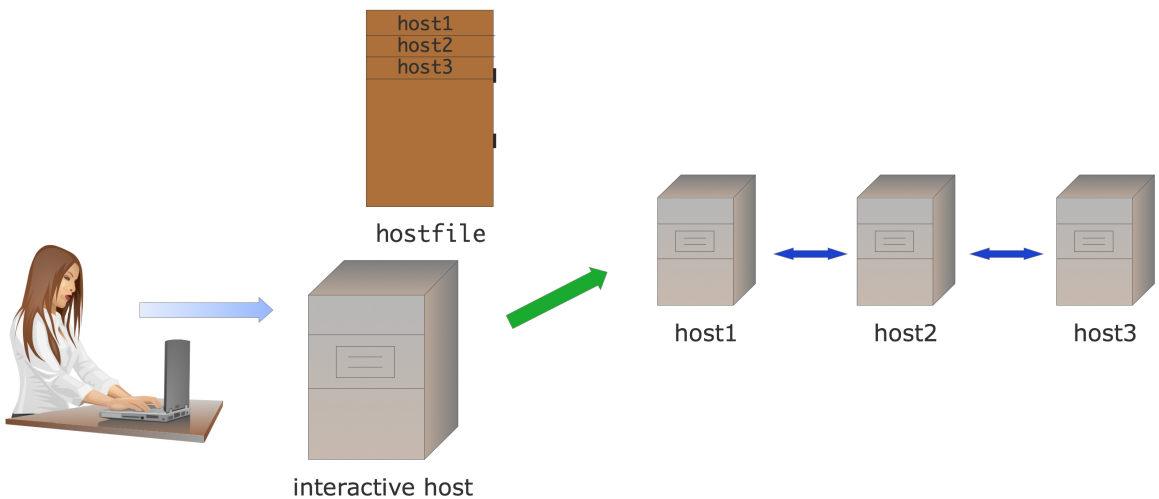
Do I need a supercomputer?

- 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. . .

Cluster setup

Typical cluster:

- Login nodes, where you ssh into; usually shared with 100 (or so) other people. You don't run your parallel program there!
- Compute nodes: where your job is run. They are often exclusive to you: no other users getting in the way of your program.



Hostfile: the description of where your job runs. Usually generated by a *job scheduler*.

How to make exercises

- Directory: `exercises-mpi-c` or `cxx` or `f` or `p`
- If a slide has a `(exercisename)` over it, there will be a template program `exercisename.c` (or `F90` or `py`).
- Type `make exercisename` to compile it
- Python: no compilation needed. Run:
  `ibrun python yourprogram`
- Add an exercise of your own to the makefile: add the name to the `EXERCISES`

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.

(On TACC machines such as stampede, use `ibrun`, no processor count.)

In a picture

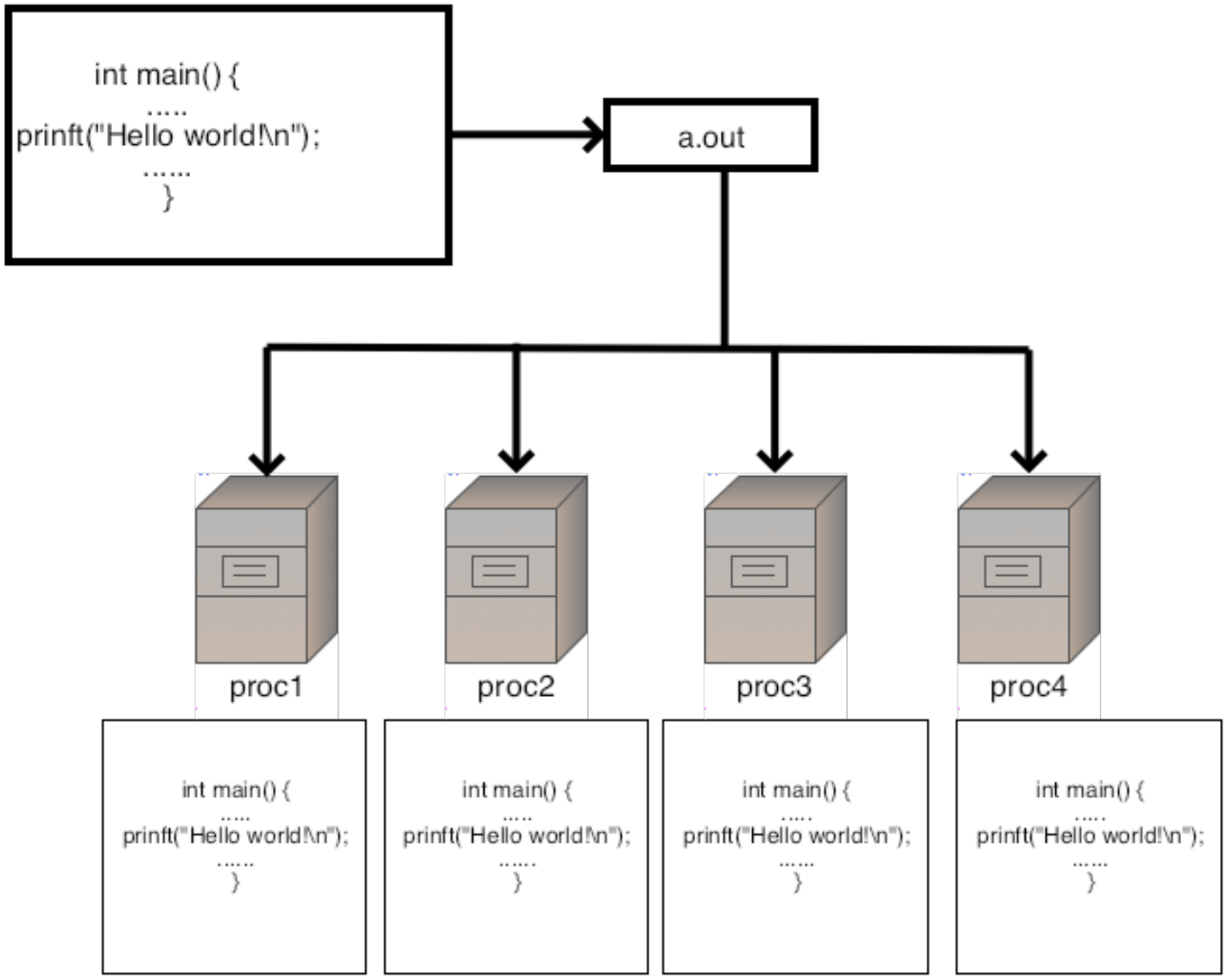

int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}

Table of Contents

MPI definitions

You need an include file:

```
#include "mpi.h" // for C
#include "mpif.h" ! for Fortran
```

- There are no real C++ bindings.
- There are true Fortran bindings, but only 2008 standard, and not widely supported yet.

MPI Init / Finalize

Then put these calls around your code:

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

and for Fortran:

```
call MPI_Init(ierr)
! your code
call MPI_Finalize(ierr)
```

About error codes

MPI routines return an integer error code

- In C: function result. Can be ignored.
- In Fortran: as parameter.
- In Python: throwing exception.

There's actually not a lot you can do with an error code: very hard to recover from errors in parallel.

Python bindings

```
module load python
```

```
from mpi4py import MPI
```

Run:

```
ibrun python-mpi yourprogram.py
```

No initialization needed.

Exercise 2

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.

Exercise (optional) 3

Now use the command `MPI_Get_processor_name` in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run a program that uses two different nodes.

(The character buffer needs to be allocated by you, it is not created by MPI, with size at least `MPI_MAX_PROCESSOR_NAME`.) 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!

```
C:
int MPI_Get_processor_name(char *name, int *resultlen)
name : buffer char[MPI_MAX_PROCESSOR_NAME]

Fortran:
MPI_Get_processor_name(name, resultlen, ierror)
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
INTEGER, INTENT(OUT) :: resultlen
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Get_processor_name()
```

How to read routine prototypes: ***??****.*

About routine prototypes: C

Prototype:

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

Use:

```
MPI_Comm comm = MPI_COMM_WORLD;
int nprocs;
int errorcode;
errorcode = MPI_Comm_size( comm,&nprocs );
```

(but forget about that error code most of the time)

About routine prototypes: Fortran

Prototype

```
MPI_Comm_size(comm, size, ierror)
INTEGER, INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Use:

```
integer :: comm = MPI_COMM_WORLD
integer :: size
CALL MPI_Comm_size( comm, size, ierr )
```

- Final parameter always error parameter. Do not forget!
- Most `MPI_...` types are `INTEGER`.

About routine prototypes: Python

Prototype:

```
# object method
MPI.Comm.Send(self, buf, int dest, int tag=0)
# class method
MPI.Request.Waitall(type cls, requests, statuses=None)
```

Use:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
comm.Send(sendbuf,dest=other)
MPI.Request.Waitall(requests)
```

Process identification

Every process has a number (with respect to a communicator)

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

For now, the communicator will be `MPI_COMM_WORLD`.

Note: mapping of ranks to actual processes and cores is not predictable!

```
Semantics:
MPI_COMM_SIZE(comm, size)
IN comm: communicator (handle)
OUT size: number of processes in the group of comm (integer)

C:
int MPI_Comm_size(MPI_Comm comm, int *size)

Fortran:
MPI_Comm_size(comm, size, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Comm.Get_size(self)
```

How to read routine prototypes: **??**.

```
Semantics:
MPI_COMM_RANK(comm, rank)
IN comm: communicator (handle)
OUT rank: rank of the calling process in group of comm (integer)

C:
int MPI_Comm_rank(MPI_Comm comm, int *rank)

Fortran:
MPI_Comm_rank(comm, rank, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Comm.Get_rank(self)
```

How to read routine prototypes: **??**.

Exercise 4 (`commrank`)

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

Write a second version of this program, where each process opens a unique file and writes to it. *On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.*

Exercise 5 (`commrank`)

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

In a picture

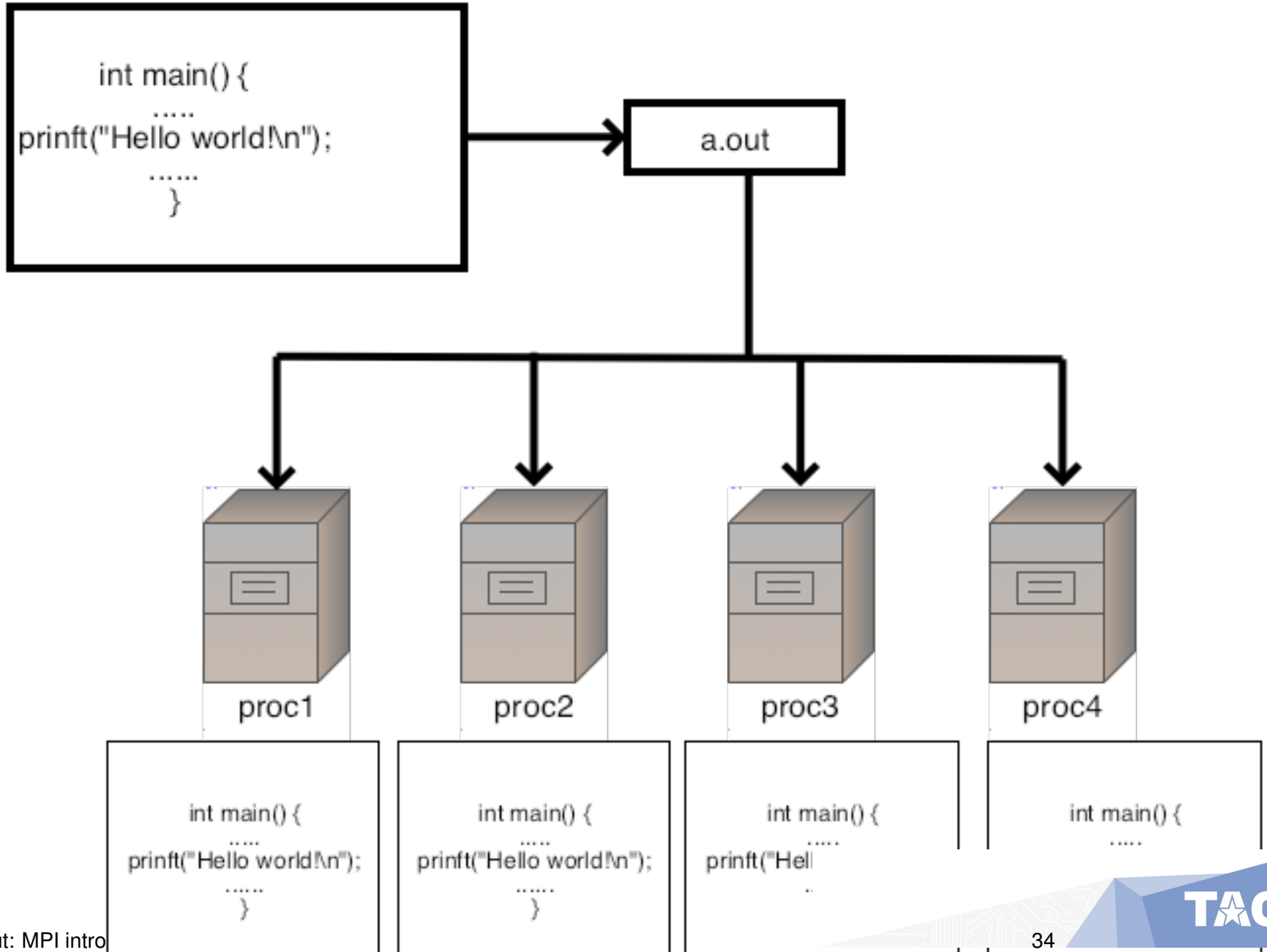

int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hell
int main() {
.....

Table of Contents

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 range 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.)

Collectives

Overview

In this section you will learn ‘collective’ operations, that combine information from all processes.

Commands learned:

- `MPI_Bcast`, `MPI_Reduce`, `MPI_Gather`, `MPI_Scatter`
- `MPI_All...` variants, `MPI_....v` variants
- `MPI_Barrier`, `MPI_Alltoall`, `MPI_Scan`

Table of Contents

Collectives

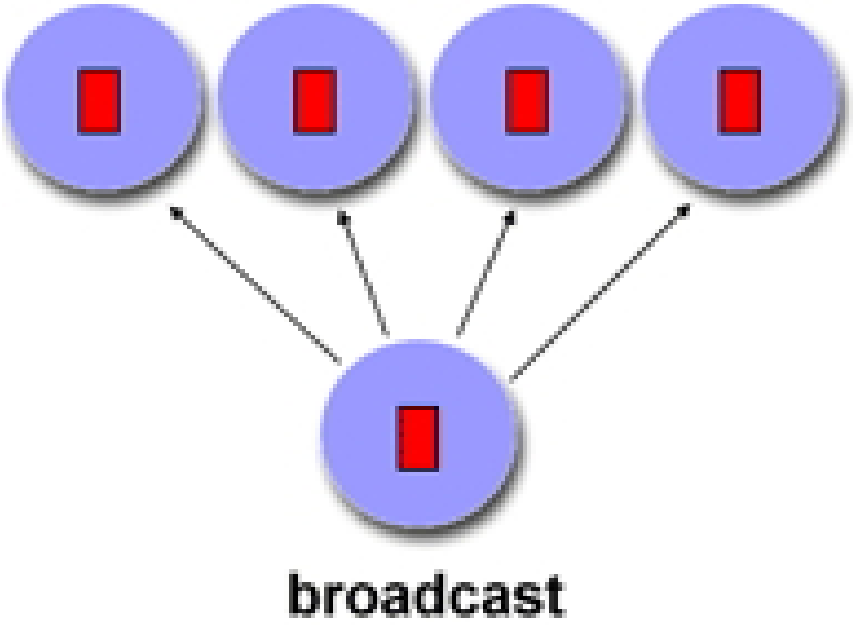
Gathering and spreading information:

- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

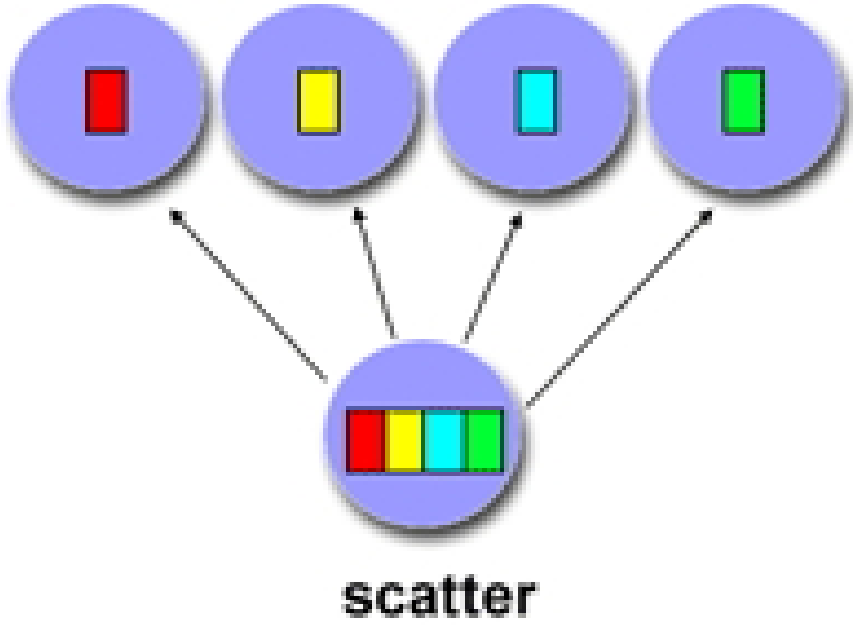
Root process: the one doing the collecting or disseminating.

Basic cases:

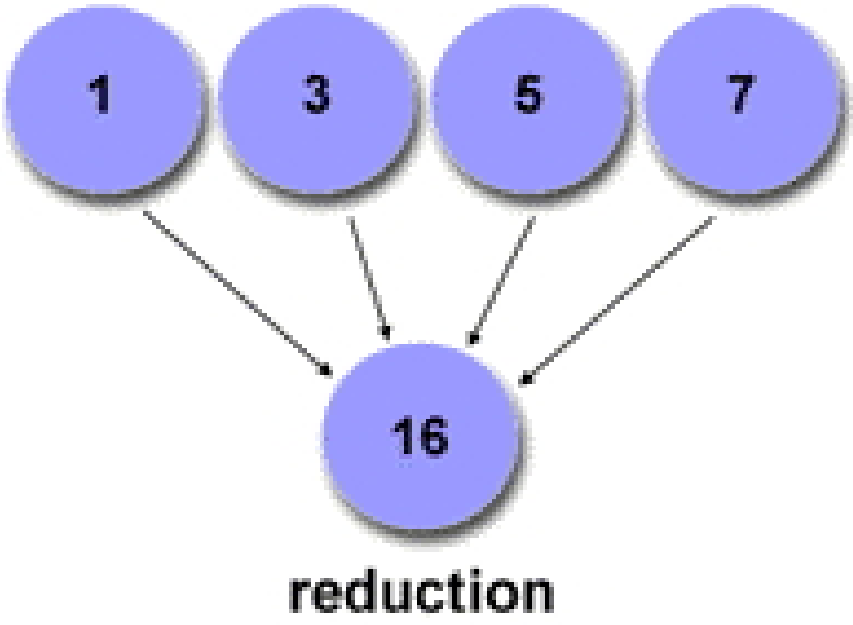
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



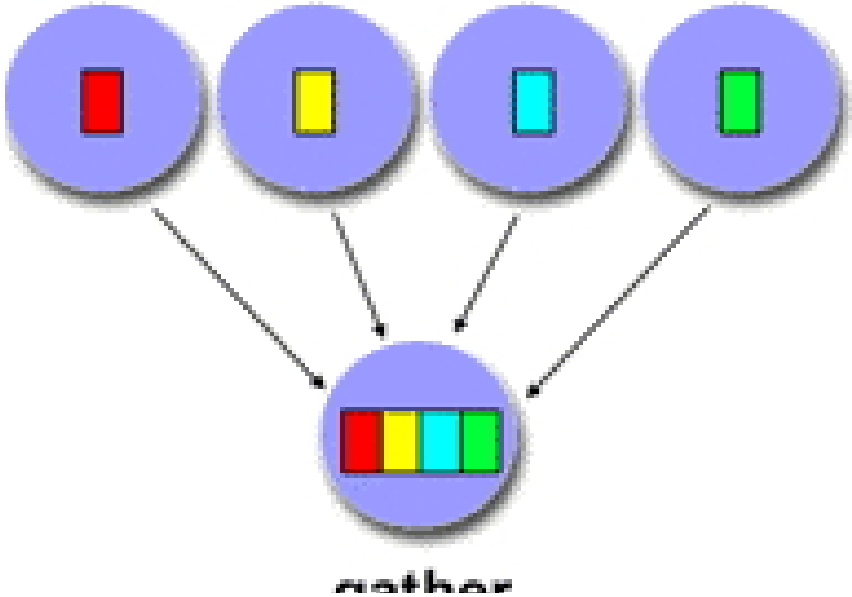
broadcast



scatter



reduction



gather

Exercise 7

How would you realize the following scenarios with MPI collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

More collectives

- Instead of a root, collect to all: `MPI_All...`
- Scatter individual data, but also individual size: `MPI_Scatterv`
- Everyone broadcasts: all-to-all
- Scan: like a reduction, but with partial results

...and more

Table of Contents

Motivation for allreduce

Standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i}^{N}(x_i - \mu)} \qquad \text{where} \qquad \mu = \frac{\sum_i^N x_i}{N}$$

and assume that every processor stores just one x_i value.

1. The calculation of the average μ is a reduction.
2. Every process needs to compute $x_i - \mu$ for its value x_i, so use allreduce operation, which does the reduction and leaves the result on all processors.
3. $\sum_i (x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Might as well use alreduce.

Allreduce

Often: everyone needs the result of a reduction

$$y \leftarrow x/\|x\|$$

- Vectors `x,y` are distributed: every process has certain elements
- The norm calculation is an all-reduce: every process gets same value
- Every process scales its part of the vector.

Reduction to single process

Regular reduce: great for printing out summary information at the end of your job.

Allreduce syntax

```
int MPI_Allreduce(
  const void* sendbuf,
  void* recvbuf, int count, MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm)
```

- All processes have send and recv buffer
- No root argument
- `count` is number of items in the buffer: 1 for scalar.
- `MPI_Datatype` is `MPI_INT`, `MPI_REAL8` et cetera.
- `MPI_Op` is `MPI_SUM`, `MPI_MAX` et cetera.