Distributed Memory Programming With MPI

MPI: *Message Passing Interface*

University of BRISTOL

Overview

- Why distributed memory? Why MPI?
- The influence of the network in distributed memory programming.
- Performance factors in shared vs. distributed programming.
- Let's get started: **`hello, world.`**
- Point-to-point communication:
  - **`MPI_send`** & **`MPI_Recv.`**
- Examples:
  - A Runtime Pitfall.
  - New Considerations for Code Robustness.

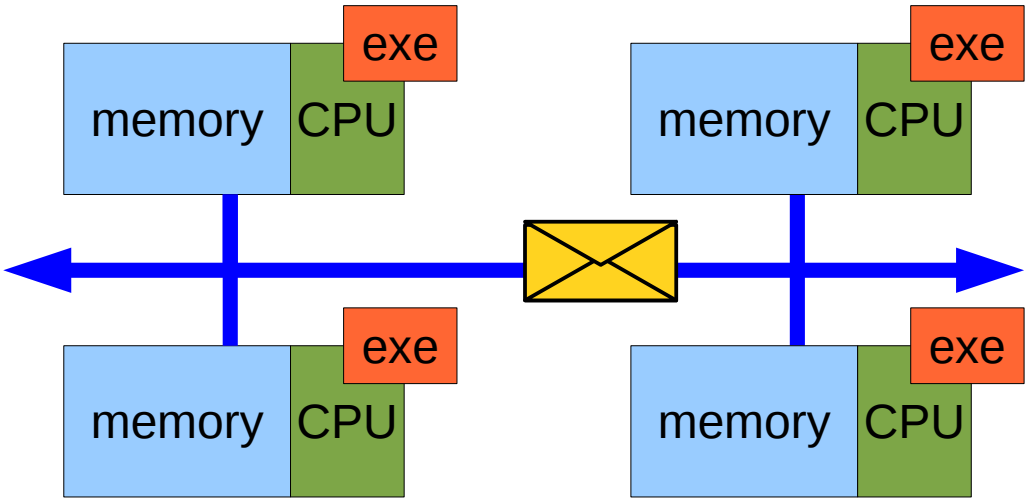
Why Distributed Memory Programming?

Commodity servers & $$$s...

Why MPI?

- A well-designed library that is intended to be portable.
- Available for C and Fortran (and elsewhere).
- Library consists of simple functions & pre-processor macros.
- Published, open standard(s) (MPI-1,-2,-3 & -4).
- Good implementations, including:
  - MPICH
  - OpenMPI

Message Passing & Networking



- **Bandwidth** limits how fast we can move large amounts of data over the network.
- ***However**, the smallest packets are still not instantaneous*.
- **Latency** places a lower limit on the transmission time for any message.

Bandwidth vs. Latency: Consider a Modem..

- Bandwidth is, say, 56kb/s. What if we need more?
  - Compression.
  - Buy another modem & double up.
- Latency is 100ms. What if we need it to be faster?
  - More modems won't help...

What's the Latency of the Internet?

- London to San Francisco is ~8600km, as the crow flies.
- Speed of light in a fibre optic cable is ~2×10^8 m/s.
- (Propagation speed in copper is about the same.)
- $8.6 \times 10^6 / 2 \times 10^8 = 0.043$s = 43ms
- There and back, so 86ms theoretical minimum time for our 17,200km journey..

```
PING new-ecology.stanford.edu (171.64.173.139) 56(84) bytes of data.
64 bytes from new-ecology.Stanford.EDU (171.64.173.139): icmp_seq=1 ttl=46 time=197 ms
64 bytes from new-ecology.Stanford.EDU (171.64.173.139): icmp_seq=2 ttl=46 time=198 ms
64 bytes from new-ecology.Stanford.EDU (171.64.173.139): icmp_seq=3 ttl=46 time=195 ms
64 bytes from new-ecology.Stanford.EDU (171.64.173.139): icmp_seq=4 ttl=46 time=193 ms
64 bytes from new-ecology.Stanford.EDU (171.64.173.139): icmp_seq=5 ttl=46 time=198 ms
64 bytes from new-ecology.Stanford.EDU (171.64.173.139): icmp_seq=6 ttl=46 time=193 ms
```

- **Only a factor of 2 out!**

The extra time is due to...

- The wireless network in my house
- User-space to kernel
- System bus (e.g. PCI) to network card
- Card interrupts to kernel
- Switch latencies (store & forward, queues...)
- Subnet Gateways
- Cable repeaters etc.

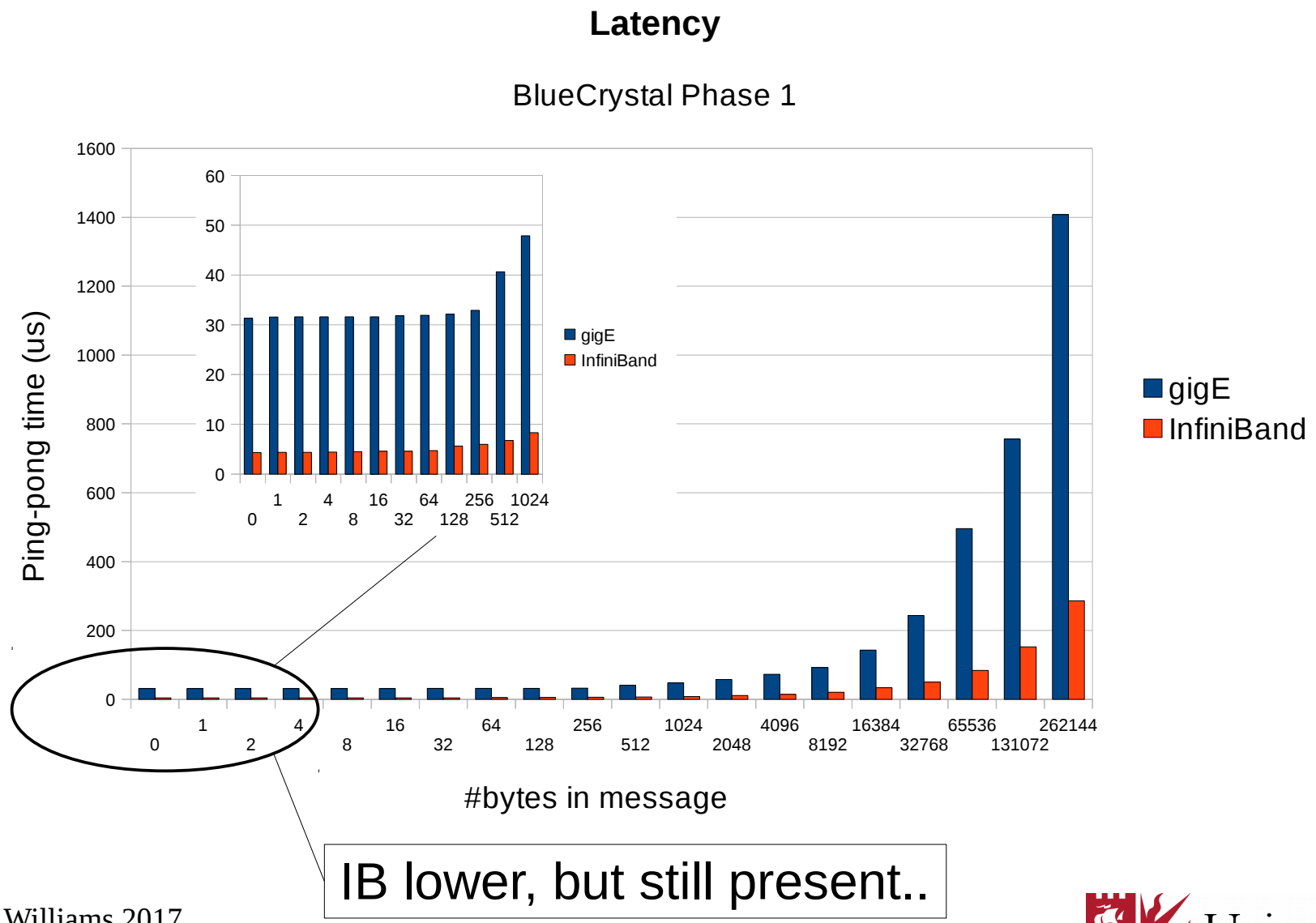
What About our Cluster?

- Now we're talking about distances O(10m).
- Propagation speed in cat5 cable is again ~$2x10^8$ m/s.
- $10/2x10^8$ = 0.00000005ms (= 50ps!)
- Let's say switch and back, so 0.0000001ms theoretical minimum transmission time..

```
PING node096.beowulf.cluster (10.141.0.96) 56(84) bytes of data.
64 bytes from node096.beowulf.cluster (10.141.0.96): icmp_seq=0 ttl=64 time=0.099 ms
64 bytes from node096.beowulf.cluster (10.141.0.96): icmp_seq=1 ttl=64 time=0.087 ms
64 bytes from node096.beowulf.cluster (10.141.0.96): icmp_seq=2 ttl=64 time=0.088 ms
64 bytes from node096.beowulf.cluster (10.141.0.96): icmp_seq=3 ttl=64 time=0.088 ms
64 bytes from node096.beowulf.cluster (10.141.0.96): icmp_seq=4 ttl=64 time=0.087 ms
64 bytes from node096.beowulf.cluster (10.141.0.96): icmp_seq=5 ttl=64 time=0.087 ms
```

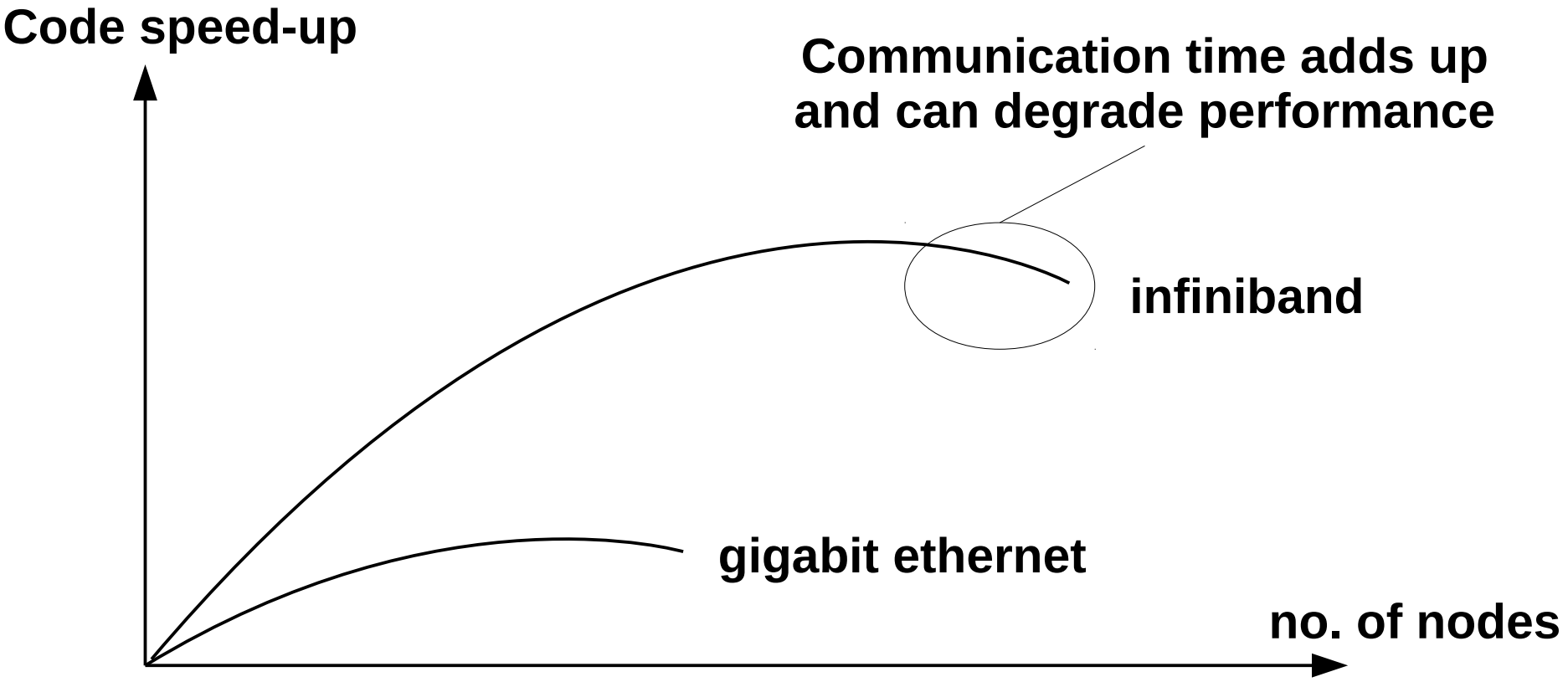
- **Faster, but around 5 orders of magnitude out!**
- **Latency is much more important in our cluster**

High Performance Networking



University of BRISTOL

Role of Latency in Parallel Scaling



- BCp1's gigabit ethernet end-to-end latency (i.e. zero bytes) is ~30μs
- BCp1's Infiniband end-to-end latency is ~4μs
- This relationship will likely change over time (ethernet catching up?)

Compare & Contrast Performance Factors

- **Shared Memory:**
  - Memory hierarchy
  - Cache coherence
  - Vectorised instructions (wide registers)
- **Distributed Memory:**
  - Memory hierarchy
  - Network latency & bandwidth
  - Vectorised instructions (wide registers)
  - *and* Cache coherence

    (if using a hybrid programming model)

Example1: **hello, world**

```
#include "mpi.h"

int main(int argc, char* argv[])
{
  ..
  MPI_Init( &argc, &argv );

  MPI_Initialized(&flag);
  if ( flag != TRUE ) {
    MPI_Abort(MPI_COMM_WORLD,EXIT_FAILURE);
  }

  MPI_Get_processor_name(hostname,&strlen);
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  printf("Hello, world; from host %s: process %d of %d\n", \
         hostname, rank, size);

  MPI_Finalize();
}
```

Example1: **hello, world**

Makefile:

```
hello_world_c: hello_world.c
    mpicc -o $@ $^
```

mpi_submit:

```
#PBS -l nodes=1:ppn=4,walltime=00:05:00

#! Create a machine file for MPI
cat $PBS_NODEFILE > machine.file.$PBS_JOBID

numprocs=`wc $PBS_NODEFILE | awk '{ print $1 }'`

#! Run the parallel MPI executable (nodes*ppn)
mpirun -np $numprocs \
       -machinefile machine.file.$PBS_JOBID \
       $application $options
```

A Practical Tip:

.bashrc

```
module add openmpi/gcc/64/1.6.5
module add openmpi/intel/64/1.6.5
module add mvapich2/open64/64/1.6
module add \
tools/gnu_builds/tau-2.23.1-openmpi
module add \
tools/intel_builds/tau-2.23.1-openmp
```

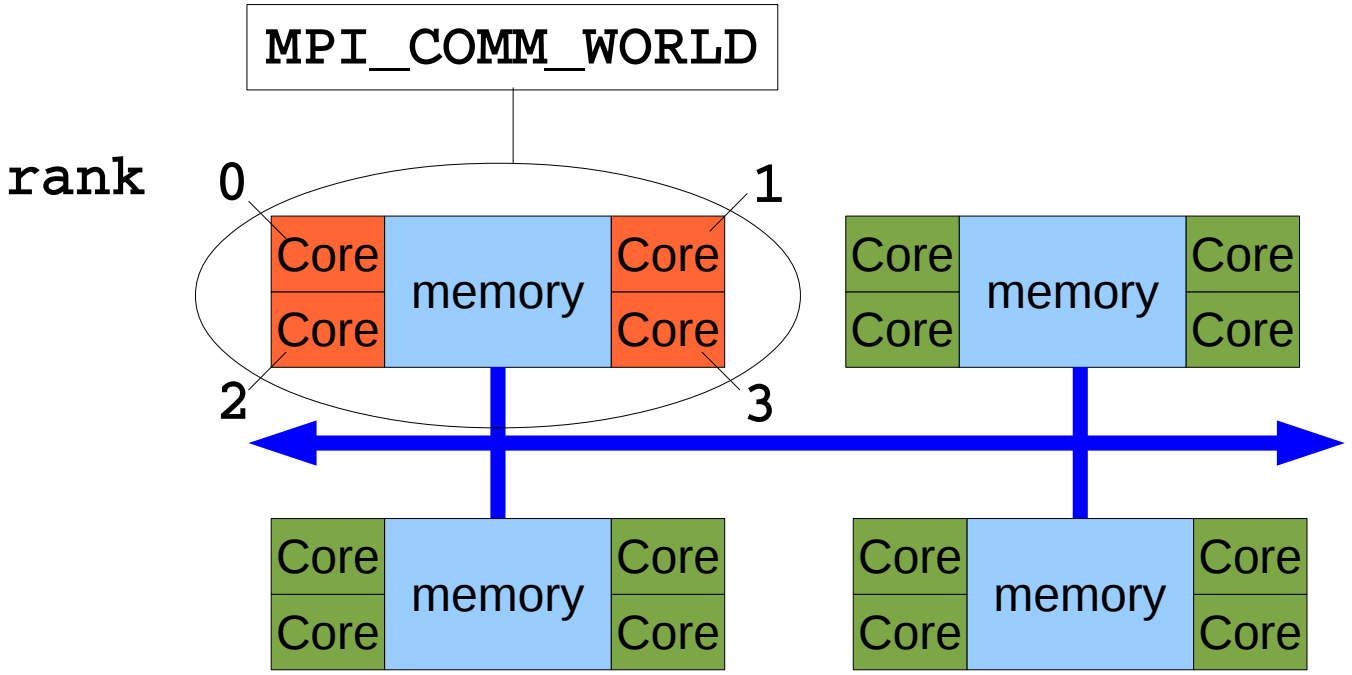
.bashrc

```
module add openmpi/gcc/64/1.6.5
module add \
tools/gnu_builds/tau-2.23.1-openmpi
```

University of BRISTOL

Establishing the Communicator

The process `mpirun` waits until all instances of `MPI_Init` have acquired knowledge of the cohort



- **Ranks: 0 (master), 1, 2, 3,..**
- **Queuing system decides how to distribute over nodes (servers)**
- **Kernel decides how to distribute over multi-core processors**

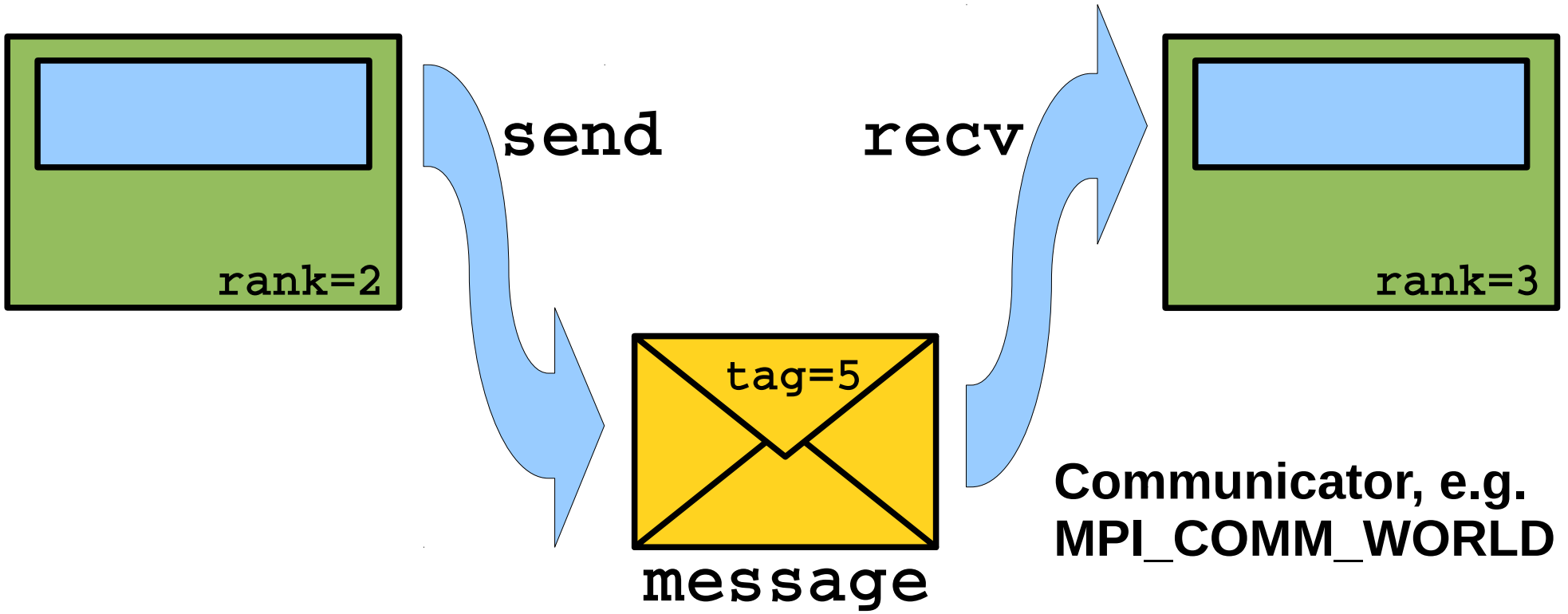
Example2: Send & Recieve

```
  if (myrank != 0) { /* if this is not the master process */
    /* create a message to send, in this case a character array */
    sprintf(message, "Come-in Danny-Boy, this is process:\t%d!", \
            myrank);
    /* send to the master process */
    dest = 0;
    MPI_Send(message,strlen(message)+1, MPI_CHAR, dest, tag, \
             MPI_COMM_WORLD);

  }
  else { /* i.e. this is the master process */
    /* loop over all the other processes */
    for (source=1; source<size; source++) {
      /* receive their messages */
      MPI_Recv(message, BUFSIZ, MPI_CHAR, source, tag, \
               MPI_COMM_WORLD, &status);
      /* and then print them */
      printf("%s\n", message);
    }
```

receiving buffer can be bigger.. but not smaller!

Point-to-Point



There is a Pattern:

```
MPI_Func(buffer,count,datatype,..,comm,status)
```

Point-to-Point: Extras

- `MPI_Recv(..,MPI_ANY_SOURCE,..)`
- `MPI_Recv(..,MPI_ANY_TAG,..)`
- `int MPI_Get_count(`

```
  MPI_Status*   status    /* in */,
  MPI_Datatype datatype   /* in */,
  int*         count_ptr  /* out */)
```

From Recv()

Not counting bytes

- Message buffer must be **contiguous** in memory
  - e.g. a 1-d array in C

Blocking Functions & Deadlock

```
if (sendfirst == 1) {
  printf("process %d of %d. Sending..\n",rank,size);
  /* first send.. */
  MPI_Send(message, BUFSIZ, MPI_CHAR, other, tag, MPI_COMM_WORLD);
  /* ..then receive */
  MPI_Recv(message, BUFSIZ, MPI_CHAR, other, tag ,MPI_COMM_WORLD, \
           &status);
  printf("process %d of %d. Received.\n",rank,size);
}
else {
  printf("process %d of %d. Receiving..\n",rank,size);
  MPI_Recv(message, BUFSIZ, MPI_CHAR, other, tag, MPI_COMM_WORLD, \
           &status);
  MPI_Send(message, BUFSIZ ,MPI_CHAR, other, tag, MPI_COMM_WORLD);
  printf("We'll never get here!\n");
}
```

- **'Blocking' `MPI_Recv` won't return until message has been received into buffer**
- **Also 'Blocking' send pattern is '*unsafe*' as it relies on a system buffer**
- **See exercise in practical session..**

Avoiding Deadlock

- Avoid growing your communication patterns in an ad hoc fashion.
- Instead, plan ahead.
- Choose an established and understandable pattern of communication. Think ***Design patterns***.
- Keep it as simple as possible.
- Start with a skeleton and gradually add the computation. See the 'debugging' tarball.

Example3: SPMD PI – Numerical Integration

```
/* width of trapezoid is the same for all trapezoids, on all procs */
width = (b – a)/(double)N;  /* N trapezoids in total */

/* how many trapezoids per process? */
local_n = N/nprocs;

printf("rank %d:\tlocal_n %d\n", rank, local_n);

/* calculate local intervals to work on */
local_a = a + (rank * (local_n * width));
/* since each local interval is (local_n * width) */
local_b = local_a + (local_n * width);

printf("rank %d:\t local_a %f, local_b %f\n", rank, local_a, local_b);

/* local trapezoid calculations are bundled into a function */
local_sum = trapezoid(local_a, local_b, local_n, width);

printf("rank %d:\t local_sum %f\n", rank, local_sum);
```

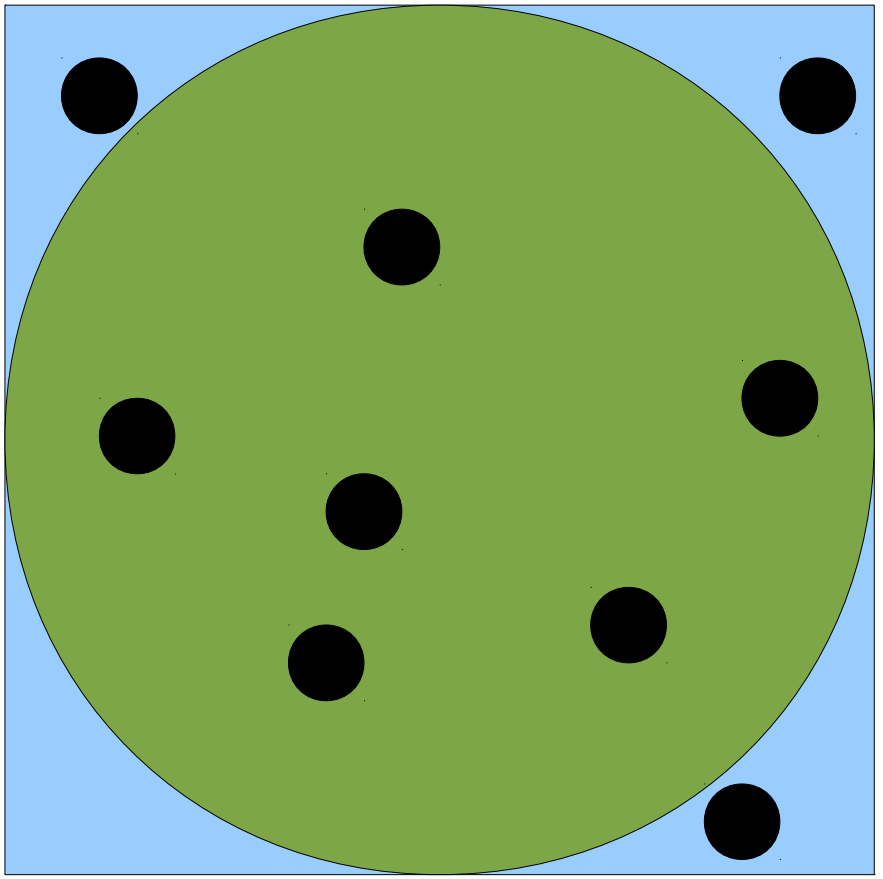
Example3: SPMD PI – Numerical Integration

```
/* message passing and totalling of local sums */
if (rank != MASTER) {
  MPI_Send(&local_sum, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
}
else { /* I am the master */
  integral = local_sum;
  for (source = 1; source < nprocs; source++) {
    MPI_Recv(&local_sum, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, \
             &status);
    integral += local_sum;
  }
}
```

<u>Exercise in Practical</u>: Code is syntactically correct, but is not guaranteed to work correctly. Ideas why?.. ..and how can we fix it?

Alternative: Dartboard Algorithm

pi:ratio of diameter to circumference of a circle



pi = 4 * A-circ / A-sq

approx. ratio of areas as ratio of dart landing counts

- **Monte Carlo**: Not as efficient in serial.. but robust to dead processors

Recap