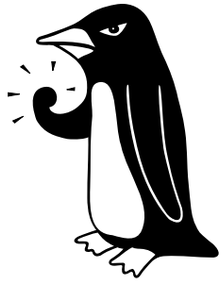
Recap of Parallelism & MPI

Chris Brady
Heather Ratcliffe



Warwick RSE

13/12/2017

Parallel programming

- Break a problem up into separate elements that different processors can work on
- Question is about layout of data
  - Single chunk of memory (either real or virtual)
    - OpenMP
    - OpenSHMEM
    - UPC
    - Others
  - Distributed memory
    - MPI

Advantages of MPI

- Works on any CPU based system
- Scales well to the largest machines
- Low overheads
- Forces developer to consider data locality
    - Improves performance

Disadvantages of MPI

- Forces developer to consider data locality
  - Can be hard to use if data locality is not possible
- Can be overkill for working on a single node
- Final program cannot in general be built without an MPI compiler
  - Unless you explicitly write serial equivalents of MPI routines

MPI In Fortran

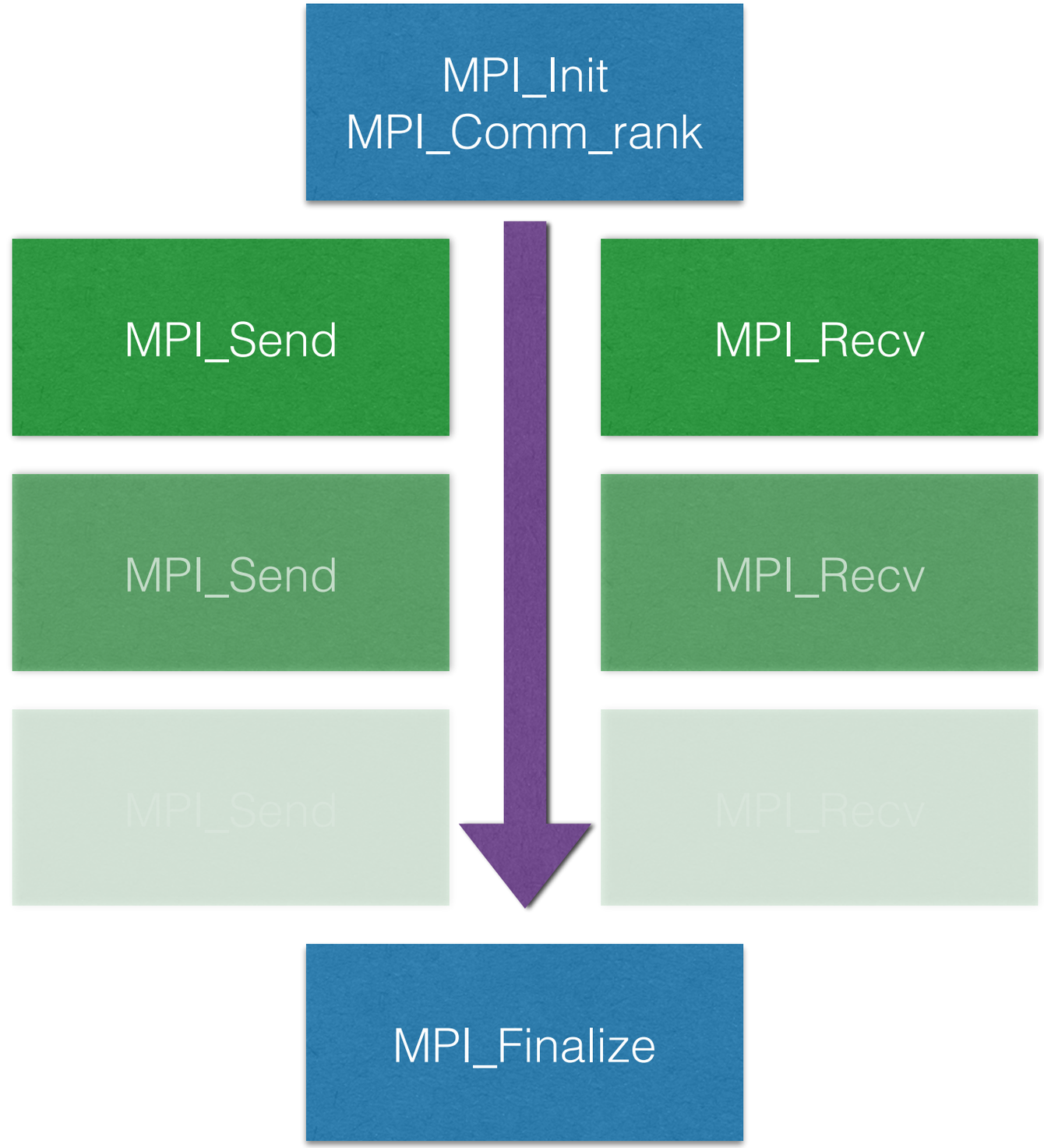
- Mostly going to define things in C
- This is because C uses types for MPI objects
- Fortran doesn't. There's two simple rules
  - Almost every parameter that's a MPI_* type in C is just INTEGER in Fortran. We'll say when it isn't
  - Every MPI function in Fortran takes an extra INTEGER parameter. This is the same as the return code in C. There is no return code.
    - Except MPI_Wtime and MPI_Wtick.

MPI In Fortran

- There is a new MPI Fortran interface
  - USE mpi_f08
  - Uses Fortran 2008 constructs
  - Much closer to the C interface
  - Probably best to not use yet, but keep an eye on it

MPI Concepts

- MPI_Init to start MPI
- MPI_Comm_rank to find unique processor rank
- MPI_Send/MPI_Recv pairs
- More Send/Recv
- MPI_Finalize



MPI Concepts

- Basic sending command is MPI_Send

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

- `buf` - a buffer containing the data

- `count` - Number of elements to send

- `datatype` - Type of elements to send

- `dest` - Where to send it to (called a **rank** in MPI)

- `tag` - uniquely identifies this message (only needs to be unique if there can be multiple inflight messages)

- comm - list of processors to send to (called a communicator in MPI)

MPI_Types

- MPI_INT - C Integer
- MPI_INTEGER - Fortran Integer
- MPI_FLOAT - C single precision
- MPI_REAL - Fortran single precision
- MPI_DOUBLE - C double precision
- MPI_DOUBLE_PRECISION - Fortran double precision
- MPI_BYTE - Single byte value
- Many, many others

MPI Concepts

- Basic receiving command is MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

- `buff` - buffer to hold the received data
- `count` - Number of elements to receive
- `datatype` - Type of elements to receive
- `source` - Where it should come from (MPI_ANY_SOURCE acceptable if you don't care)
- `tag` - uniquely identifies this message (only needs be unique if there can be multiple inflight messages) (MPI_ANY_TAG acceptable if you don't care)
- `comm` - list of processors to receive from
- `status` - object containing information about the message
    - INTEGER, DIMENSION(MPI_STATUS_SIZE) in Fortran

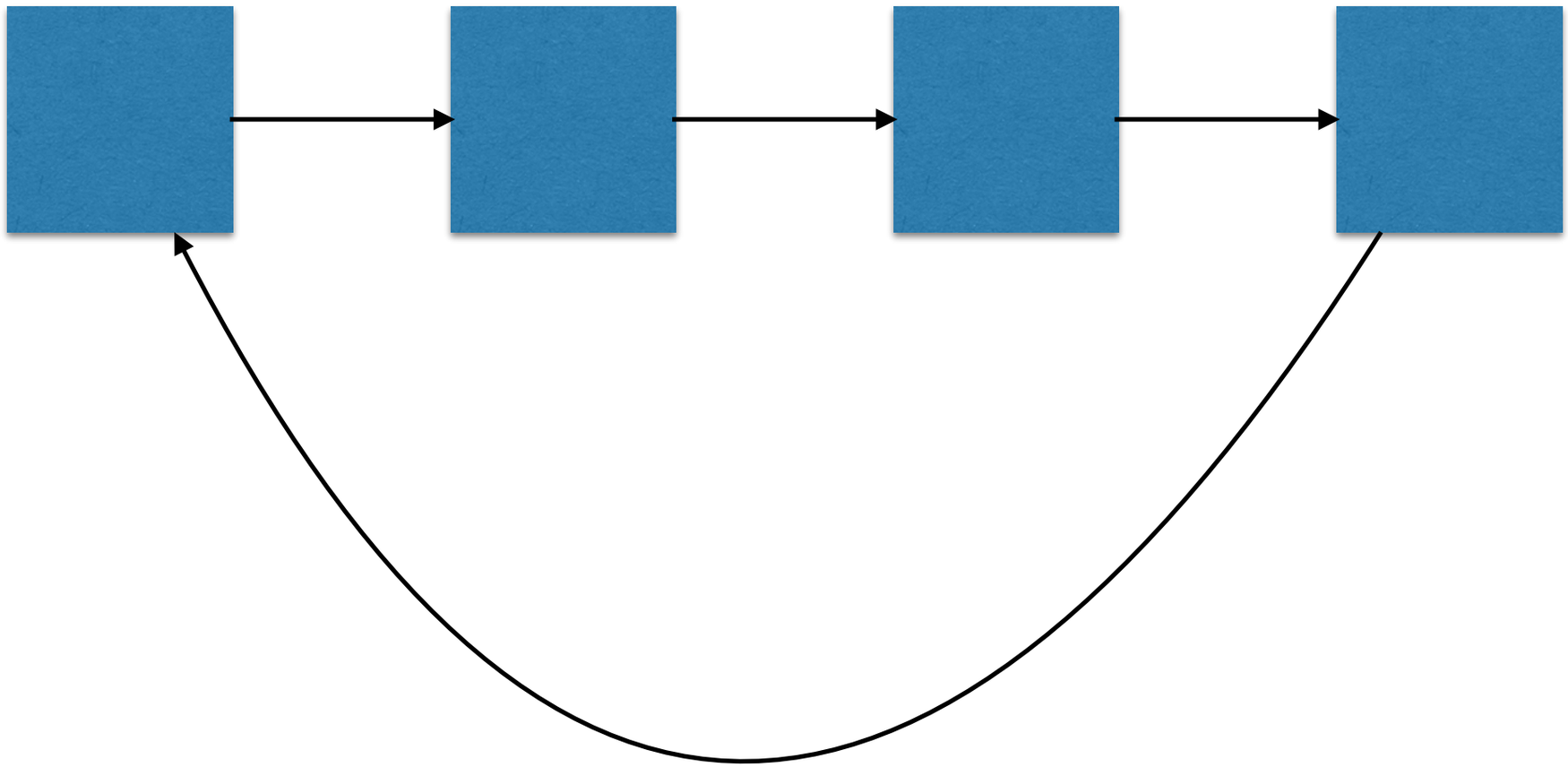
Send variants

- There are four blocking send variants
- MPI_Send - Returns once it is safe to reuse the send buffer. Message may or may not have been received
- MPI_Ssend - Does not return until the message has been received
- MPI_Bsend - Copies data to be sent into buffer for sending and returns immediately
- MPI_Rsend - Ready mode send. Only valid if matching receive has already been posted. Limited use.

MPI 101

- The simplest possible MPI program is one that simply passes a value to it's "neighbour"
  - Define neighbour as processor with rank 1 higher than your rank
  - Wrap back round so that the last processor sends to the first
- Useful model of real problem (mentioned later)

MPI 101



Send Receive Example

```
PROGRAM wave

  USE mpi
  IMPLICIT NONE

  INTEGER, PARAMETER :: tag = 100

  INTEGER :: rank, recv_rank
  INTEGER :: nproc
  INTEGER :: left, right
  INTEGER :: ierr

  CALL MPI_Init(ierr)

  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  !Set up periodic domain
  left = rank - 1
  IF (left < 0) left = nproc - 1
  right = rank + 1
  IF (right > nproc - 1) right = 0

  IF (rank == 0) THEN
    CALL MPI_Ssend(rank, 1, MPI_INTEGER, right, tag, MPI_COMM_WORLD, ierr)
    CALL MPI_Recv(recv_rank, 1, MPI_INTEGER, left, tag, MPI_COMM_WORLD, &
        MPI_STATUS_IGNORE, ierr)
  ELSE
    CALL MPI_Recv(recv_rank, 1, MPI_INTEGER, left, tag, MPI_COMM_WORLD, &
        MPI_STATUS_IGNORE, ierr)
    CALL MPI_Ssend(rank, 1, MPI_INTEGER, right, tag, MPI_COMM_WORLD, ierr)
  END IF

  PRINT *,"Rank ", rank, " got message from rank ", left, " of ", recv_rank

  CALL MPI_Finalize(ierr)

END PROGRAM wave
```

Send Receive Example

```c
#include <stdio.h>
#include <mpi.h>

#define TAG 100

int main(int argc, char ** argv)
{

  int rank, recv_rank, nproc, left, right;

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

  //Set up periodic domain
  left = rank - 1;
  if (left < 0) left = nproc - 1;
  right = rank + 1;
  if (right > nproc - 1) right = 0;

  if (rank == 0) {
    MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
    MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
  }

  printf("Rank %3d got message from rank %3d of %3d\n", rank, left, recv_rank);

  MPI_Finalize();

}
```

Result

```
Rank   1 got message from rank   0 of   0
Rank   2 got message from rank   1 of   1
Rank   3 got message from rank   2 of   2
Rank   4 got message from rank   3 of   3
Rank   5 got message from rank   4 of   4
Rank   6 got message from rank   5 of   5
Rank   7 got message from rank   6 of   6
Rank   8 got message from rank   7 of   7
Rank   9 got message from rank   8 of   8
Rank  10 got message from rank   9 of   9
Rank  11 got message from rank  10 of  10
Rank  12 got message from rank  11 of  11
Rank  13 got message from rank  12 of  12
Rank  14 got message from rank  13 of  13
Rank  15 got message from rank  14 of  14
Rank   0 got message from rank  15 of  15
```

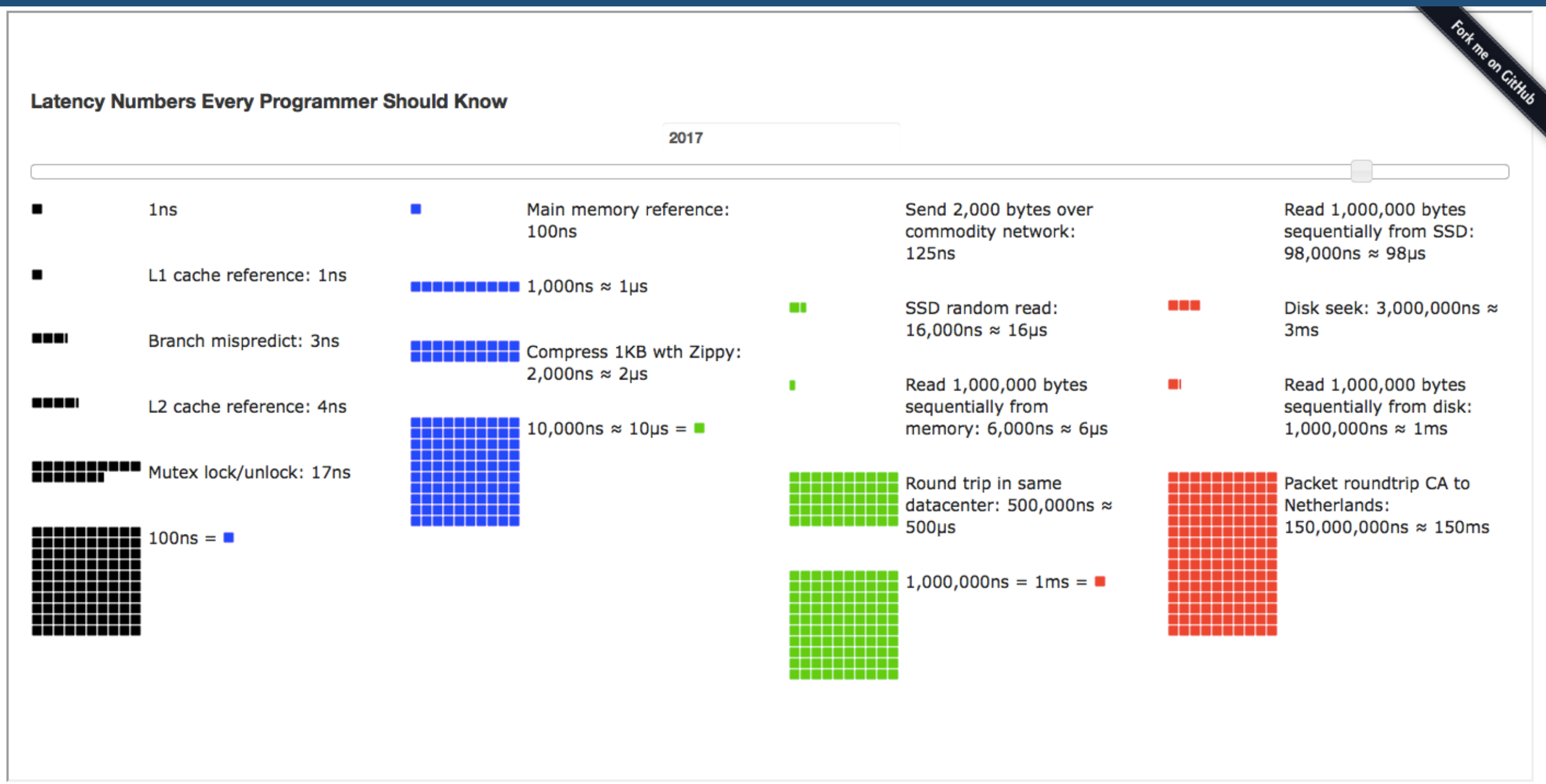
Tiny new bit

- MPI_STATUS_IGNORE
- Constant that can be used wherever an MPI_Status object would be.
- Cannot be used in place of an array of status objects
  - MPI_STATUSES_IGNORE

Performance

- What about if we do this passing loop many times?
- 100,000 repetitions = 100,000 messages per processor
- Takes about 1.3 seconds to complete on 16 cores of a shared memory machine
- Communication time of about 1 microsecond
- Latency more important than bandwidth

Theoretical value

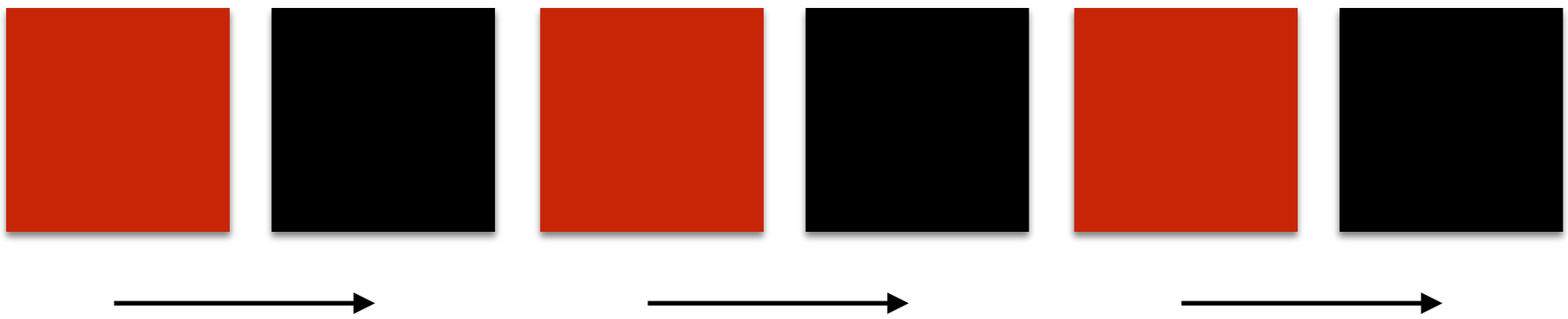


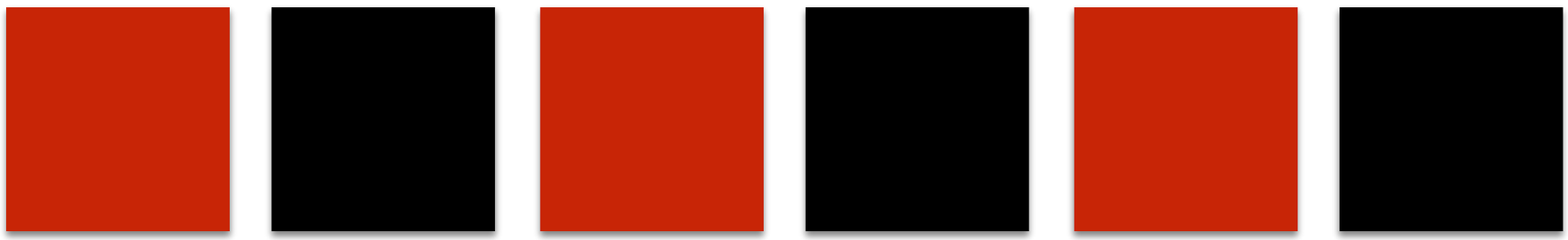
- Should be about 100ns

What's wrong?

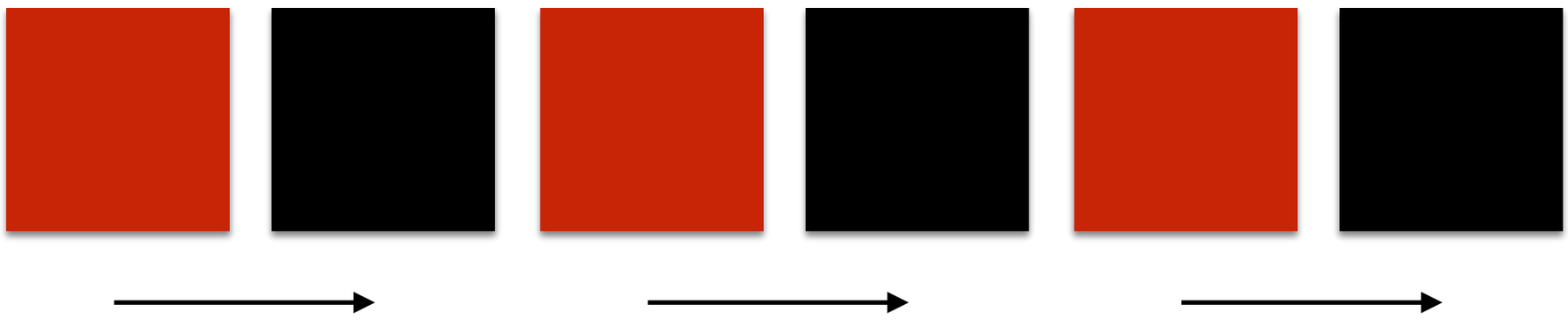
- Most of the processors are sitting idle
- Processor 1 has to wait for 0
- Processor 2 has to wait for 1 which has to wait …
- etc. etc.
- Not all processors can send at once
  - Deadlock
  - Not true, but let's pretend for the moment

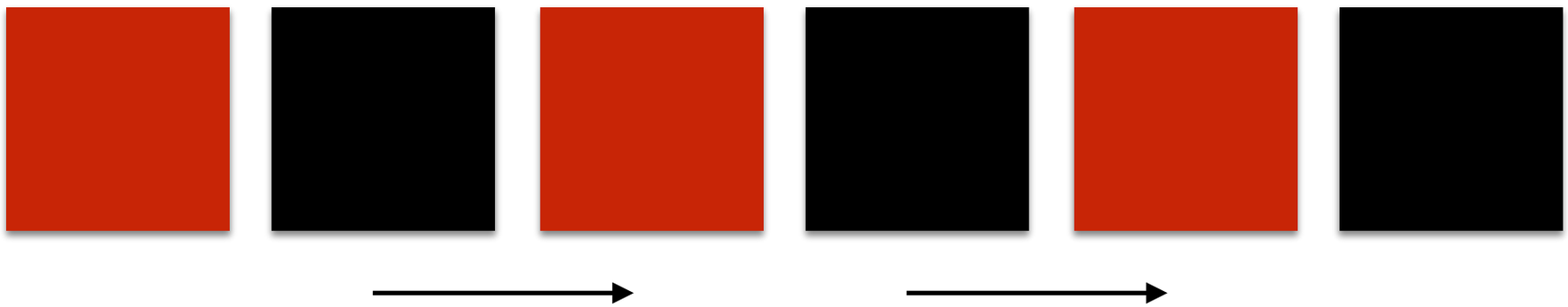
Red/Black ordering



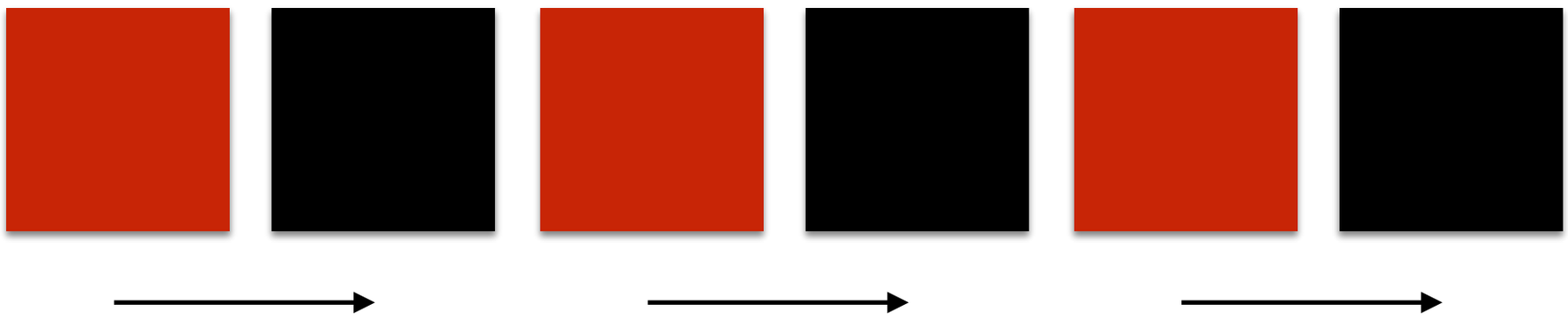


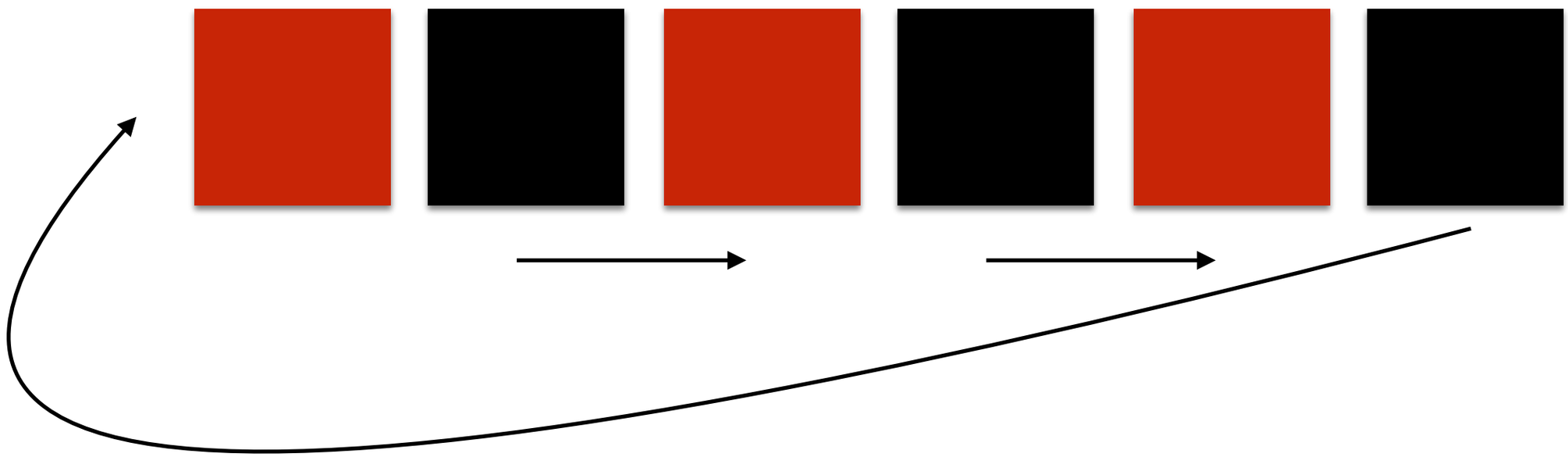
Red/Black ordering





Red/Black ordering





Change in code

```
if (rank == 0) {
  MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
  MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
} else {
  MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
}
```

```
if (rank%2 == 0) {
  MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
  MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
} else {
  MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
}
```

Better?

- Takes about 0.22 seconds on the same 16 processors
- Effective latency is now about 138ns
- Much closer to theoretical value
- I said that processors can in reality all send at once
  - Special commands
- Is that any faster?

MPI_Sendrecv

- This is a combined send and receive in a single command
- Might be new, some courses cover this too
- Doesn't deadlock so long as both the send and the receive parts complete

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

Change in code

```
if (rank%2 == 0) {
  MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
  MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
} else {
  MPI_Recv(&recv_rank, 1, MPI_INTEGER, left, TAG, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Ssend(&rank, 1, MPI_INTEGER, right, TAG, MPI_COMM_WORLD);
}
```

```
MPI_Sendrecv(&rank, 1, MPI_INTEGER, right, TAG, &recv_rank, 1, MPI_INTEGER,
    left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Better now?

- Now takes about 0.09 seconds to complete on same 16 cores
- ~100ns latency
- Yay!

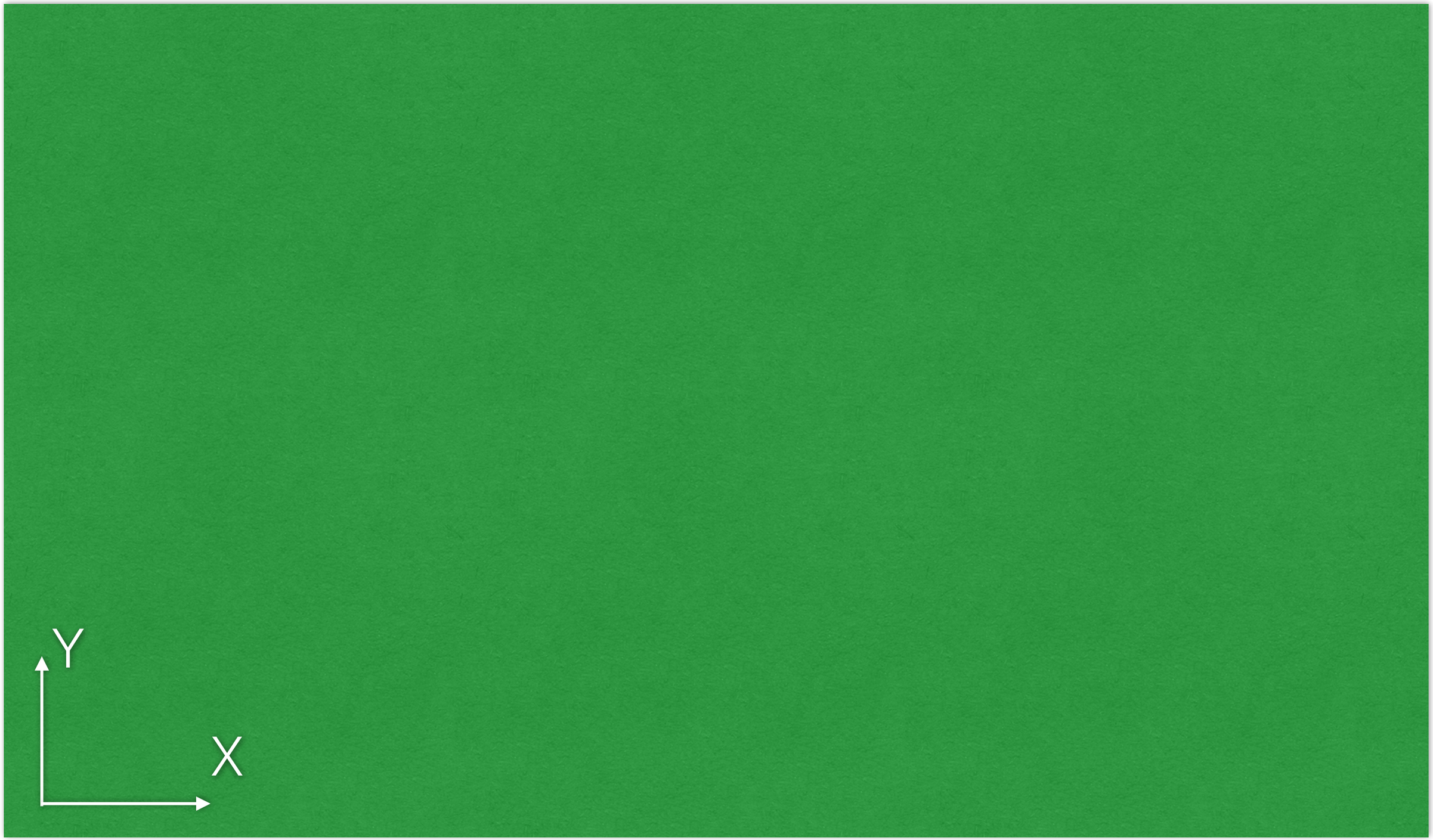
Types of parallel decomposition

Aims

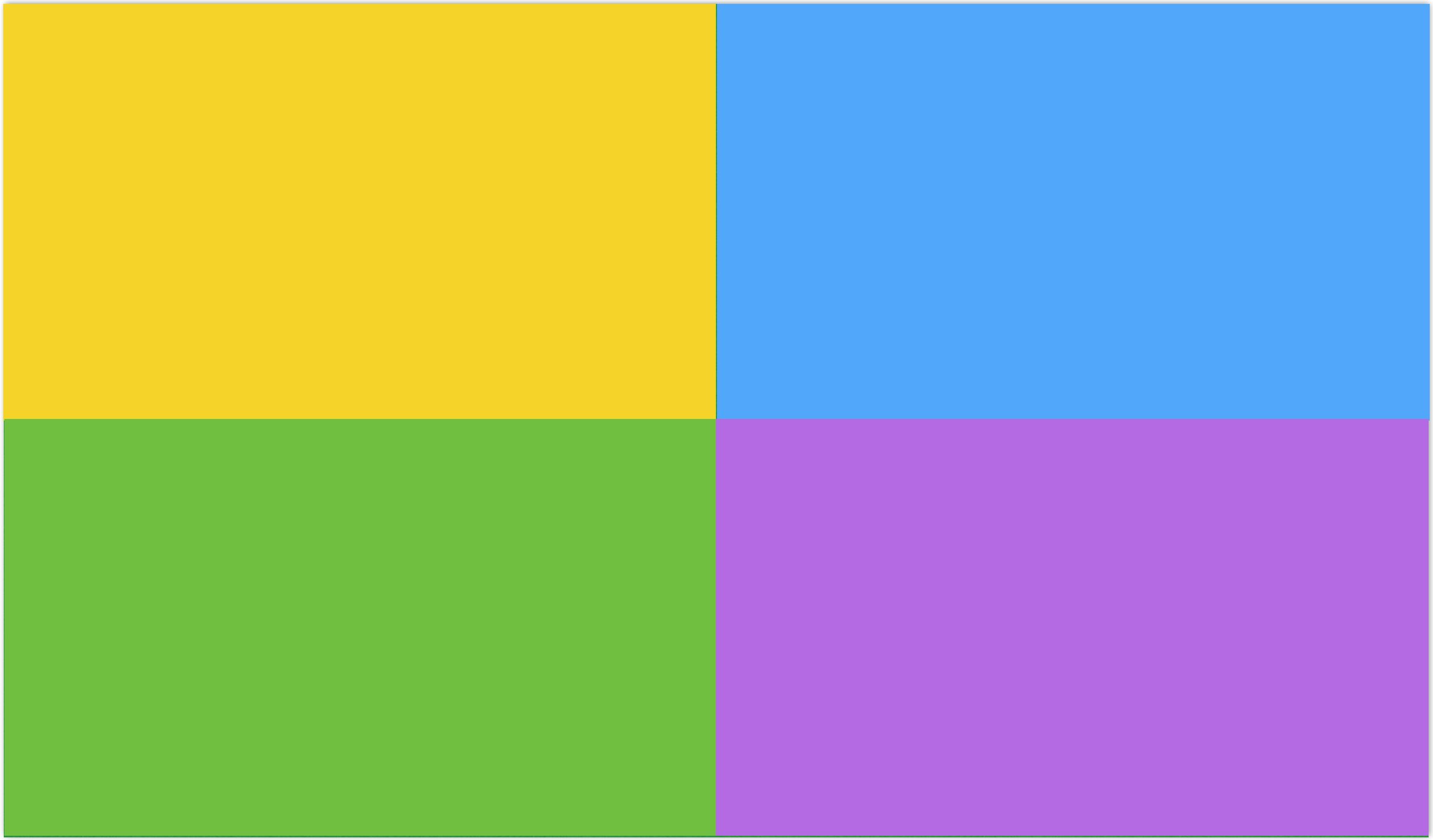
- Want to decompose your **data** across several processors

- Want to do it so that each processor *mostly* works with the data that it owns
  - Limiting case of each processor working only with it's own data.
  - Embarrassing or trivial parallelism
- Not always possible, but always optimal if possible

Domain decomposition



Domain decomposition



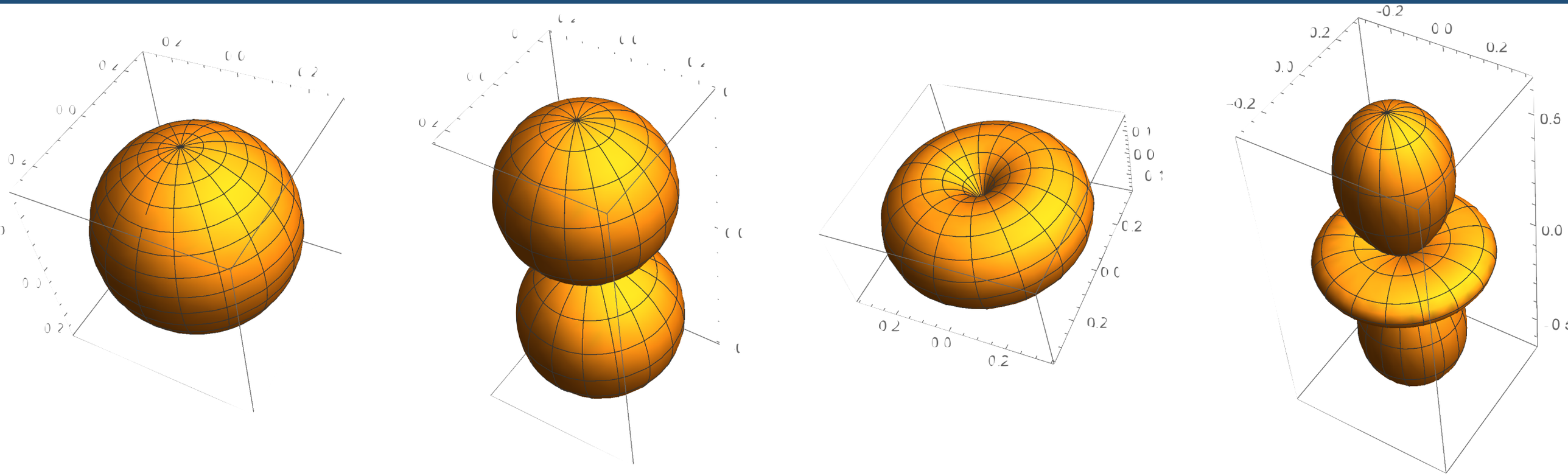
Domain decomposition

- Simplest solution
- Solving problem in spatial domain
- Split space up so that each processor has a single part of spatial domain
- If you have an operation that is local in space this will scale to world class supercomputers
  - Then can just send and receive (sendrecv generally) at edges
- Not all operations are like this

Arithmetic decomposition

- You have to decompose on something
- If you can’t just decompose in space, need something else
- If your solution exists defined over a set of orthogonal basis functions you can decompose on each basis function
- Same idea as decomposing in space (literally, x is orthogonal to y is orthogonal to z)

Arithmetic decomposition



- Basis functions are orthogonal so do not intrinsically couple together
- Can have equations that couple them
- That causes communication
- Again want as far as possible only connection between “nearby” functions

Other decompositions

- Other variations on a theme
- Sometimes you really are coupling data together in arbitrary ways
- Decomposition mostly doesn’t matter then
- Just split things up and deal with the access problems later
- MPI may not be the best solution
  - Look at PGAS systems
  - Performance will probably not be good

Reduction

Reduction

- Get data from all processes, operate on it and then give it to
  - One “root” processor (MPI_Reduce)
  - All processors (MPI_Allreduce)
- If given an array, operate on it elementwise. Does not reduce over elements

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

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

Allreduce in operation

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char ** argv)
{

  int rank, nproc, recv;

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

  MPI_Allreduce(&rank, &recv, 1, MPI_INTEGER, MPI_MAX, MPI_COMM_WORLD);

  printf("On rank %3d MPI_Allreduce gives maximum rank as %3d\n", rank, recv);

  MPI_Finalize();

}
```

Operations

- MPI_MAX : Maximum element
- MPI_MIN : Minimum element
- MPI_SUM : Sum all elements
- MPI_PROD : Multiply all elements
- MPI_LAND : Logical AND all elements
- MPI_LOR : Logical OR all elements
- MPI_BAND : Bitwise AND all elements
- MPI_BOR : Bitwise OR all elements
- MPI_MAXLOC : Value and rank of maximum element
- MPI_MINLOC : Value and rank of minimum element

Non-blocking

Non blocking

- Both send and receive have “non-blocking” variants
  - MPI_Isend
  - MPI_Irecv

- These return immediately

- If you change the data in the send buffer before the send is completed the changed version will be sent

- If you put data in the receive buffer before the receive is completed it will be overwritten by the receive

Non-blocking

```
int MPI_Isend(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm, MPI_Request
*request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- Similar syntax to Send and Recv
- Adds MPI_Request object
- Needed because you need to check the status of non-blocking sends and receives

MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Waits until the communication request “request” has completed

- Works for sends and receives

- MPI_Sendrecv is a non-blocking send + non-blocking receive + wait for both wrapped up in a single convenient package

Other "conventional" routines

Broadcast

- Send information from one rank (root) to all other ranks

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm )
```

Scatter

- Send different data from root rank to all other processors

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm)
```

- sendbuf is actually sendcount * nproc long
- Sends sendcount elements to each other processor
- recvbuf is thus recvcount long
- unless using custom types recvcount=sendcount

Gather

- Collect data from all ranks onto root rank

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm)
```

- recvbuf is actually recvcount * nproc long
- Receives sendcount elements from each other processor
- sendbuf is thus sendcount long
- unless using custom types sendcount=recvcount

Allgather

- Collect data from all ranks and gives to all ranks

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

- recvbuf is actually recvcount * nproc long
- Each rank receives sendcount elements from each other rank
- sendbuf is thus sendcount long
- unless using types sendcount=recvcount

MPI_Alltoall

- Generalised scatter and gather combined.
- Each rank sends some data to every other rank and gets data from every other rank

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

- sendbuf is sendcount*nproc long
- recvbuf is recvcount*nproc long
- Unless using custom types, sendcount = recvcount

V - variants

- MPI_Scatter, MPI_Gather, MPI_Allgather and MPI_Alltoall have v variants
  - MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv and MPI_Alltoallv
- These are more general and allow you to send or receive different amounts of data from different processors
- Have to have a mechanism to ensure that every processor knows how much memory to allocate
- Generally using them is an indication that you might be making your MPI too complex (but sometimes necessary)