Parallel Computing

Distributed memory model

MPI

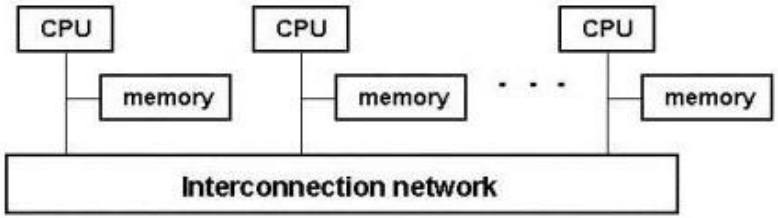
Leopold Grinberg

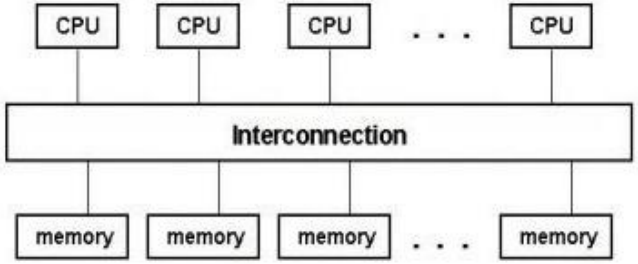
T. J. Watson IBM Research Center, USA

Why do we need **to compute in parallel**

- large problem size - <u>memory constraints</u>
- computation on a single processor takes too long – <u>time constrain</u>
- combination of memory and time constrain

(classical) classifications of computer architecture*





distributed memory

shared memory

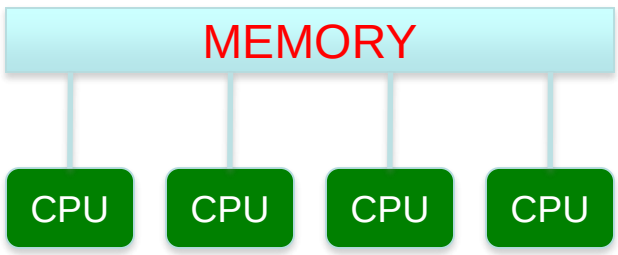
SMP

*appropriate for Multiple Instruction Multiple Data (MIMD) architecture

Shared Memory Computer Architecture

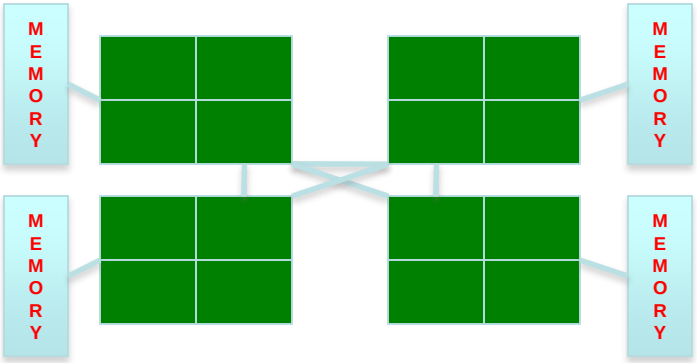
All processors can access all memory (global address space)

Uniform Memory Access (UMA)



Typically are Symmetric Multiprocessor (SMP) machines with identical processors
Equal access and access times to memory
Sometimes called CC-UMA - Cache Coherent UMA.
(Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.)

Non-Uniform Memory Access (NUMA)



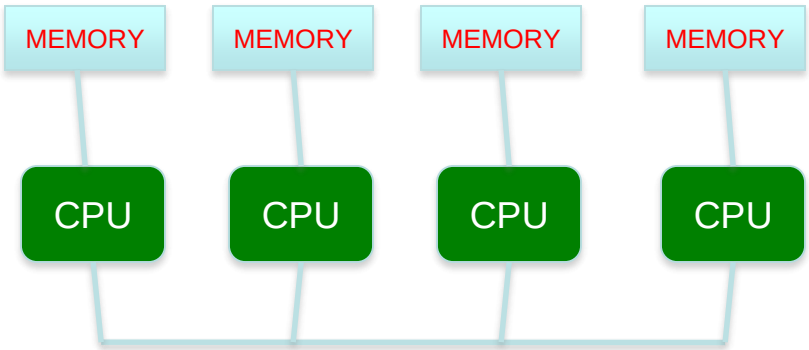
Typically made by physically linking several SMPs.
SMP *i* can directly access memory of SMP *j*.
Non-equal speed in memory access.
May also be CC-NUMA

Shared Memory Computer Architecture: the good and the bad

1. User-friendly programming perspective to memory.
2. Relatively fast access to data stored in shared memory.
3. More memory / task

1. Lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
2. Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.
3. Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory Computers



Data exchange between attached to each CPU memory requires inter-CPU *communication.*

Memory is scalable with number of processors.
Memory is local – no concept of global address space and cache-coherency.
Programmer is responsible for designing communication across CPUs

Parallel Computers of 2010...

Most of the computers have hybrid memory architecture.

Several middleware have been developed to allow using the hybrid computers as “distributed memory computers” and also distributed memory computers as “shared memory” machines.

For example it is possible to run MPI-based applications on SGI computers where memory can be shared among thousands of cores.
UPC allows use of “global” arrays, in fact it simply hides the communication.

GPUs introduce another dimension. These compute cards can be attached to CPUs such that the CPUs can “outsource” its tasks to GPUs.

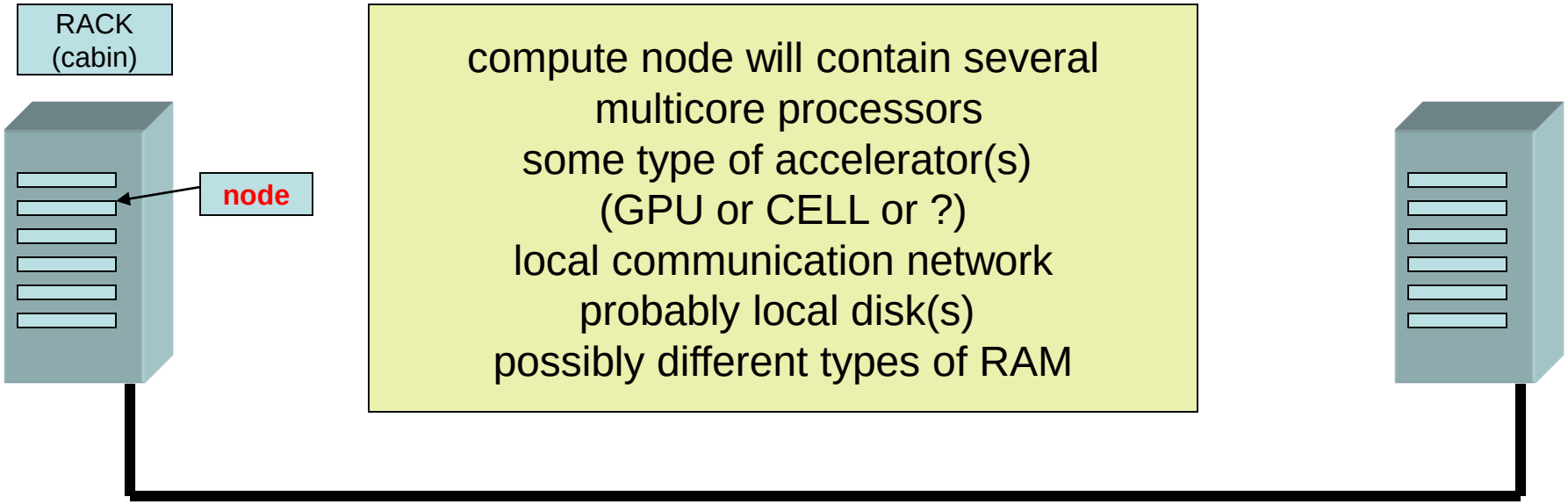
Main (classical) approaches to parallel programming

1. **message-passing model (MPI)**
   This is the most commonly used model for parallel programming on distributed-memory architectures.
2. **directives-based data-parallel model (OpenMP)**
   The message-passing model is the most commonly used model for parallel programming on distributed-memory architectures
3. **Hybrid approach MPI+OpenMP**

New hybrid approaches include use of CUDA (or openCL) distributing work to CPUs and GPUs (or CELL processors)

Trend

The core unit in a new (future) computer architecture will be a “compute node”



compute node will contain several
multicore processors
some type of accelerator(s)
(GPU or CELL or ?)
local communication network
probably local disk(s)
possibly different types of RAM

Parallel Program with MPI

Parallel programs consist of **multiple instances of a serial program** that communicate by library calls (MPI). These calls may be roughly divided into the following four classes:

1. Calls used to initialize, manage, and finally terminate communications.
These calls are used for starting communications, identifying the number of processes being used, creating subgroups of processors, and identifying which process is running a particular instance of a program.

*2. Calls used to communicate between **pairs** of processes.*
These calls, called point-to-point communications operations, consist of different types of send-and-receive operations.

*3. Calls that perform communications operations among **groups** of processes.*
These calls are the collective operations that provide synchronization, certain types of well-defined communications
operations among groups of processes, and calls that perform communication/calculation operations.

4. Calls used to create arbitrary data types.
These provide flexibility in dealing with complicated data structures.

Parallel Program Design

Your **main goal** when writing a parallel program is to get

better performance than you would get from a serial version.

You need to consider several issues when designing a parallel code:

- **problem decomposition**

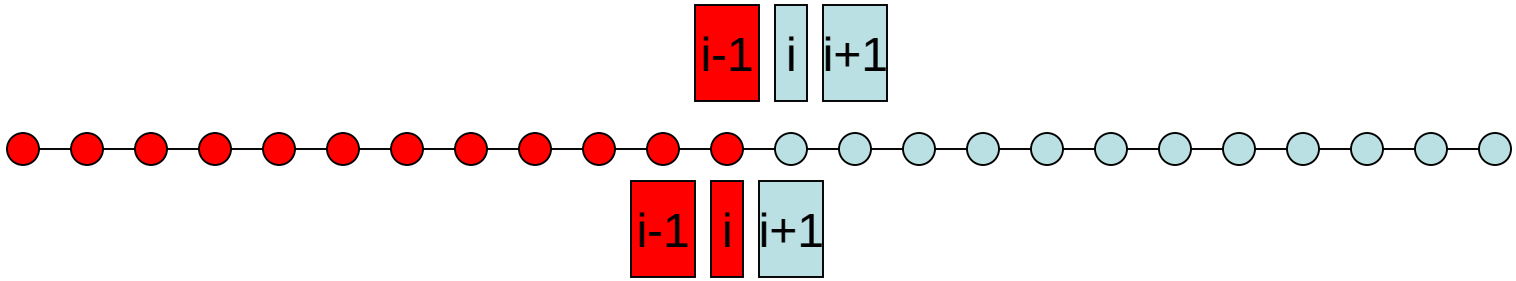
  algebraic or geometric decomposition; functional (task) decomposition;

- **load balancing** (minimizing process idle time)
- **concurrent computation and communication**
- **concurrent communications**
- **hierarchical structure of** the modern high-performance **computers**

Example of a Data Parallel Problem

$$\frac{\partial^2 u}{\partial x^2} = f(x)$$

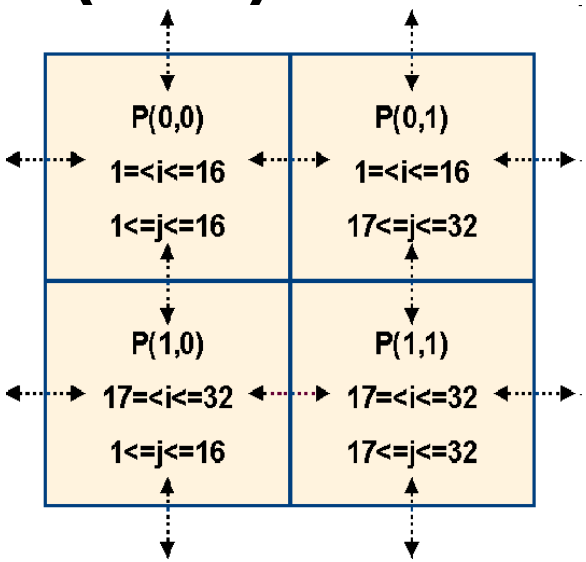
$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} = f(x_i)$$



$$u_i = 0.5(u_{i-1} + u_{i+1}) - 0.5\Delta x^2 f_i \qquad u_i = 0.5(u_{i-1} + u_{i+1}) - 0.5\Delta x^2 f_i$$

Example of a Data Parallel Problem (2D)

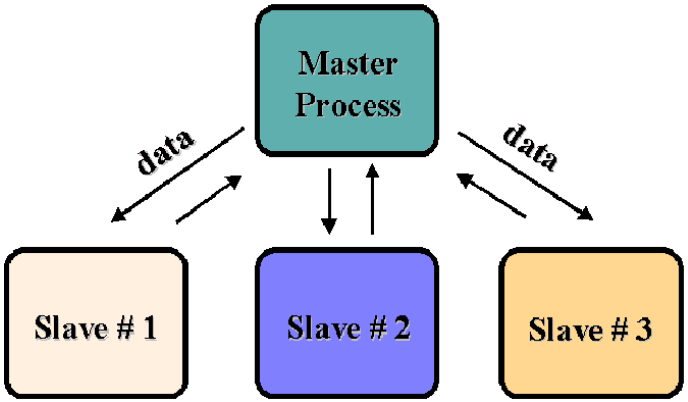
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

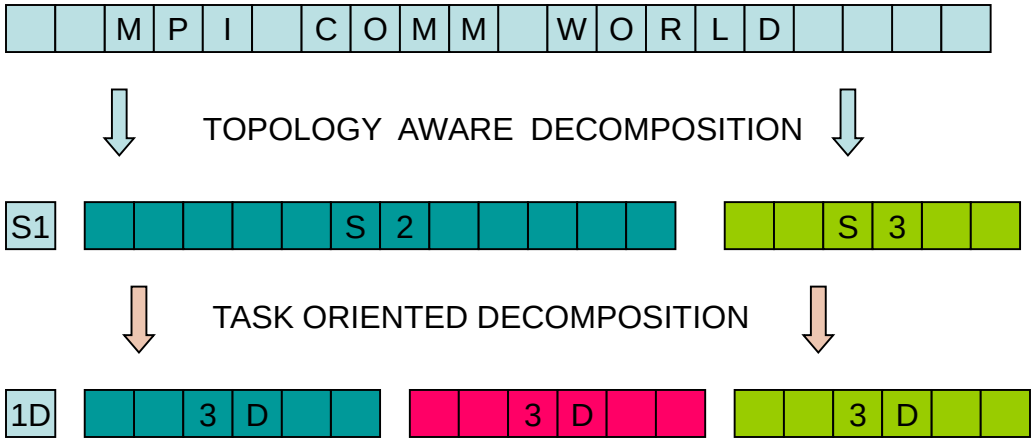


$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta x^2}$$

$$\boxed{u_i\left(1 + \frac{\Delta x^2}{\Delta y^2}\right) = 0.5\left(u_{i-1,j} + u_{i+1,j} + \frac{\Delta x^2}{\Delta y^2}(u_{i,j-1} + u_{i,j+1})\right) - 0.5\Delta x^2 f_{i,j}}$$

Functional Parallelism





Parallel Program Design: execution time

1. Computation time
2. Idle time
3. Communication time
4. IO time
5. Computer boot-up time

minimize each of this components!!!

Load Balancing

Load balancing divides the required work equally among all of the available processes.

This ensures that one or more **processes do not remain idle** while the other processes are actively working on their assigned sub-problems so that valuable computational resources are not wasted.

Load balancing **can be easy** when the same operations are being performed by all the processes on different pieces of data.

Most of the time load balancing is far from being trivial.
When there are large variations in processing time, you may need to adopt an alternative strategy for solving the problem.

MPI

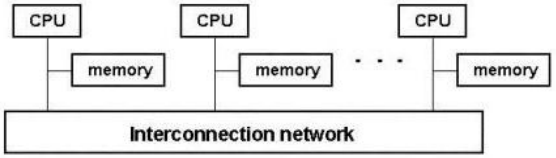
The Message Passing Interface (MPI) is a **standard library**.

MPI is **not a programming model** !!!

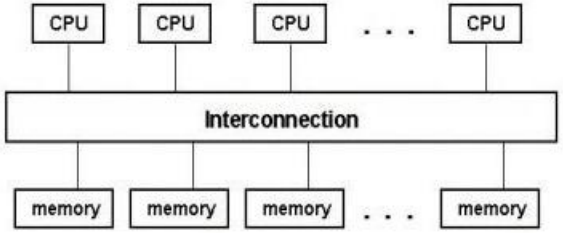
Some MPI libraries are free and some are not (commercial)

MPI

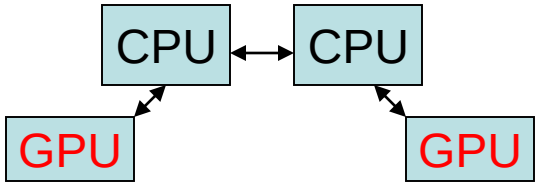
MPI allows for the coordination of a program running as multiple processes in a **distributed memory** environment.



MPI also can be used in a **shared memory** system.



MPI also can be used in a **heterogeneous system**.



The **standardization** of the MPI library makes it very powerful and enables source code **portability** since MPI programs should compile and run as-is on any platform.

MPI also allows efficient implementations **across a range of architectures**.

MPI-1 MPI-2 MPI-3 ….

- MPI was developed over two years of discussions led by the MPI Forum, a group of approximately sixty people representing about forty organizations. The **MPI-1** standard was defined in 1994, and it consists of the following:

  - It specifies the names, calling sequences, and results of subroutines and functions to be called from Fortran 77 and C, respectively. All implementations of MPI must conform to these rules, thus ensuring portability. MPI programs should compile and run on any platform that supports the MPI standard.

  -The detailed implementation of the library is left to individual vendors, who are thus free to produce optimized versions for their machines.

- Implementations of the MPI-1 standard are available for a wide variety of platforms.

- An **MPI-2** standard has also been defined. It provides for additional features, including tools for parallel I/O, C++ and Fortran 90 bindings, and one-sided communication.

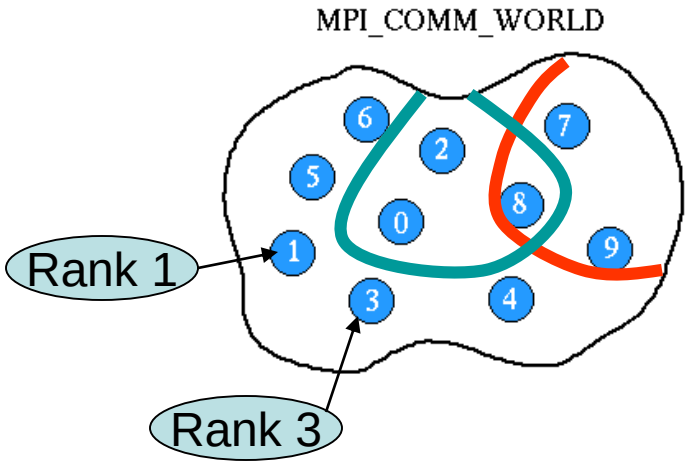
- **MPI-3.0** – year 2014

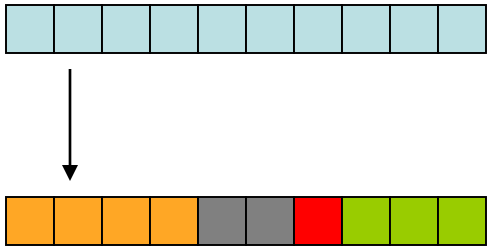
Type of MPI routines

- *Point-to-point communication*
- *Collective communication*
- Process groups
- Process topologies
- Environment management and inquiry

A communicator

A *communicator* is an MPI object that defines a group of processes that are permitted to communicate with one another. Every MPI message must specify a communicator via a “name” that is included as an explicit parameter within the argument list of the MPI call.





MPI Naming Conventions

- All names have `MPI_` prefix.
- In FORTRAN:
  - All subroutine names upper case, last argument is return code

```
call MPI_XXXX(arg1,arg2,…,ierr)
call MPI_XXXX_XXXX(arg1,arg2,…,ierr)
```

  - A few functions without return code

If `ierr == MPI_SUCCESS`, Everything is ok; otherwise, something is wrong.

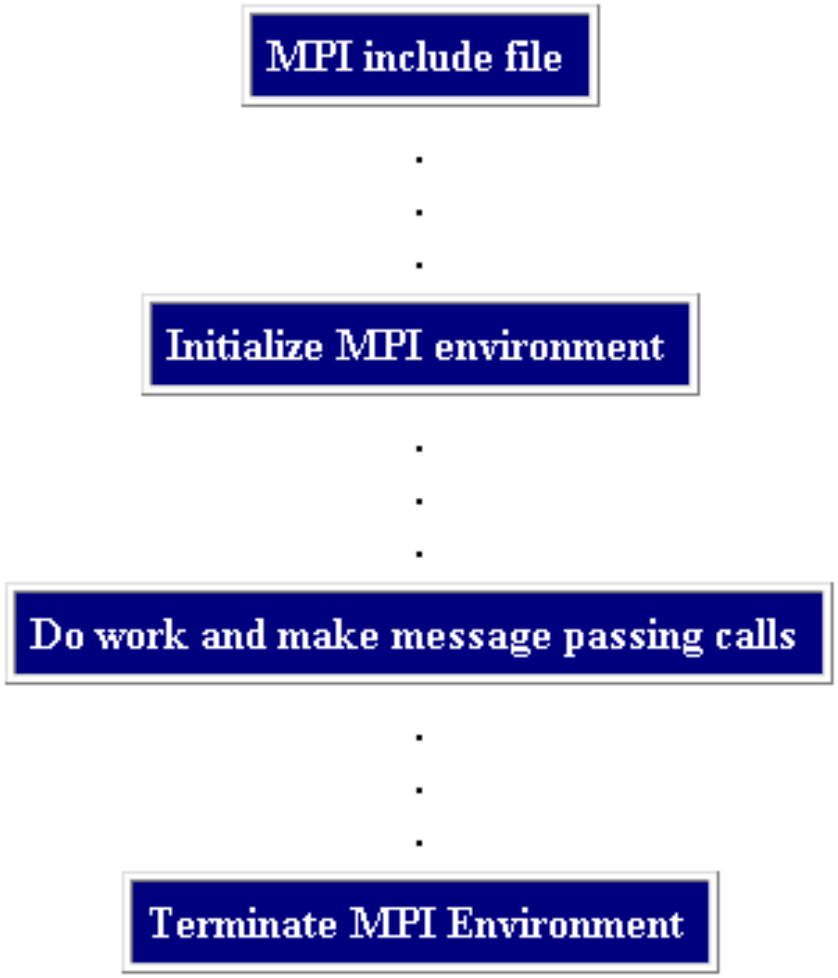
- In C++(C): mixed uppercase/lowercase

```
ierr = MPI_Xxxx(arg1,arg2,…);
ierr = MPI_Xxxx_xxx(arg1,arg2,…);
```

- MPI constants all uppercase

`MPI_COMM_WORLD, MPI_SUCCESS, MPI_DOUBLE, MPI_SUM, …`

General MPI Program Structure

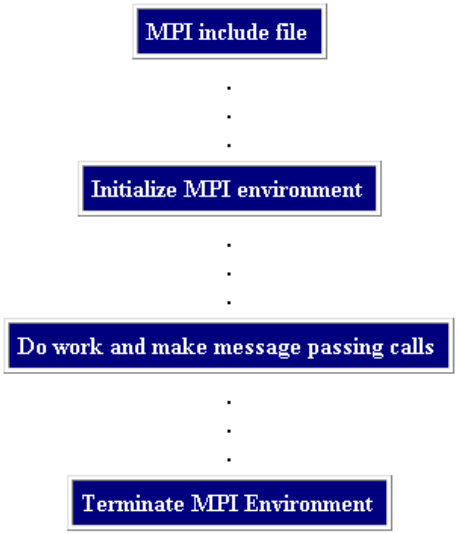


The minimal MPI subset. MPI program structure

1. MPI_Init()
2. MPI_Finalize()
3. MPI_Comm_size()
4. MPI_Comm_rank()
5. MPI_Send()
6. MPI_Recv()

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

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

MPI_Init (&argc, &argv);
/* starts MPI */

MPI_Comm_rank (MPI_COMM_WORLD, &rank);
/* get current process id */
MPI_Comm_size (MPI_COMM_WORLD, &size);
/* get number of processes */


printf( "Hello world from process %d of %d\n",
          rank, size );
MPI_Finalize();
return 0;
}
```

MPI Header Files



- In C/C++:

```
#include <mpi.h>, before including <stdio.h>
```

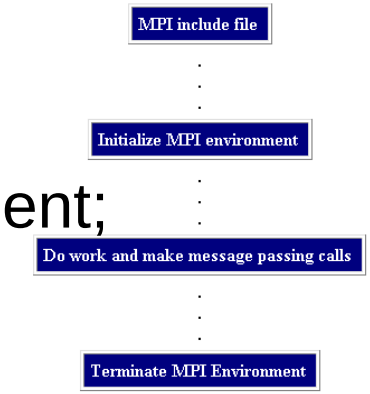
- In FORTRAN:

```
include 'mpif.h'
```

or (in FORTRAN90 and later)

```
use MPI
```

Initialization



- Initialization: `MPI_Init()` initializes MPI environment;
  - Must be called before any other MPI routine
  - Can be called only once; subsequent calls are erroneous.
- Use `MPI_Initialized(int *flag)` to check if MPI_init has been called already.

```
int MPI_Init(int *argc, char ***argv)


int main(int argc, char ** argv)
{
  MPI_Init(&argc, &argv);
  int flag;
  MPI_Initialized(&flag);
  if(flag != 0) … // MPI_Init called
  … …
  MPI_Finalize();
  return 0;
}
```

Termination

- `MPI_Finalize()` cleans up MPI environment
  - Must be called before exits.
  - No other MPI routine can be called after this call, even `MPI_INIT()`
  - Exception: `MPI_Initialized()` (and `MPI_Get_version()`,`MPI_Finalized()`).
- Abnormal termination: `MPI_Abort()`
  - terminates (all) MPI processes.

```
int MPI_Finalize(void)
MPI_FINALIZE(IERR)
  integer IERR

int MPI_Abort(MPI_Comm comm, int errorcode)
MPI_ABORT(COMM,ERRORCODE,IERR)
  integer COMM, ERRORCODE, IERR
```

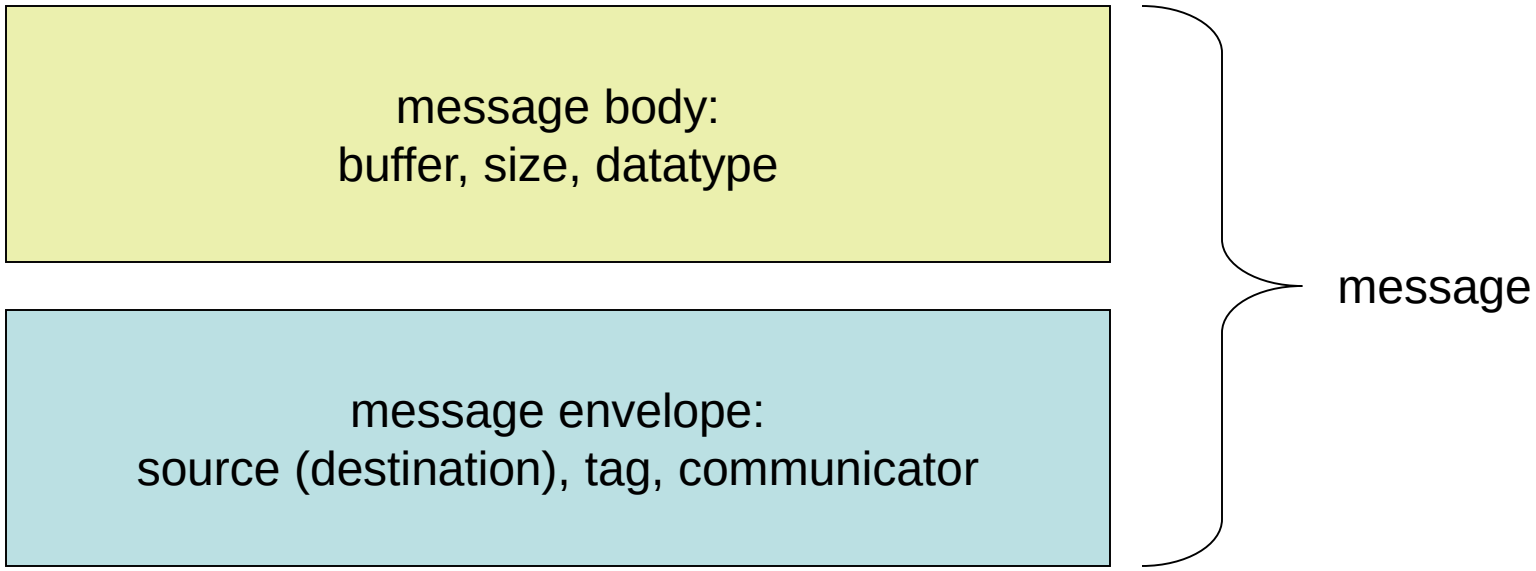
MPI Communications

- **Point-to-point communications**
  - Involves a sender and a receiver
  - Only the two processors participate in communication

- **Collective communications**
  - All processors within a communicator participate in communication (by calling same routine, may pass different arguments);
  - Barrier, reduction operations, gather, scatter…

Point – to – point communication

1. *rank i* <u>sends</u> data, *rank j* <u>receives</u> data
2. *rank i* and *rank j* <u>exchange</u> data



MPI Datatypes

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	*signed int*
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	*double*
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)
MPI_PACKED	(none)

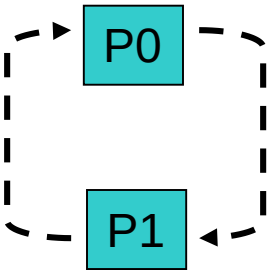
Blocking point-to-point communication


```
int MPI_Send(
    void *buf,     /* initial address of send buffer  */
    int count,     /* number of elements in send buffer (nonnegative integer) */
    MPI_Datatype datatype, /* datatype of each send buffer element */
    int dest,      /* rank of destination (integer)   */
    int tag,       /* message tag (integer)           */
    MPI_Comm comm  /* communicator                    */
);
```

```
int MPI_Recv(
    void *buf,     /* initial address of receive buffer  */
    int count,     /* number of elements in receive buffer (nonnegative integer) */
    MPI_Datatype datatype, /* datatype of each receive buffer element */
    int dest,      /* rank of source (integer)        */
    int tag,       /* message tag (integer)           */
    MPI_Comm comm,     /* communicator                */
    MPI_Status   *status    /* status object          */
);
```

Deadlock

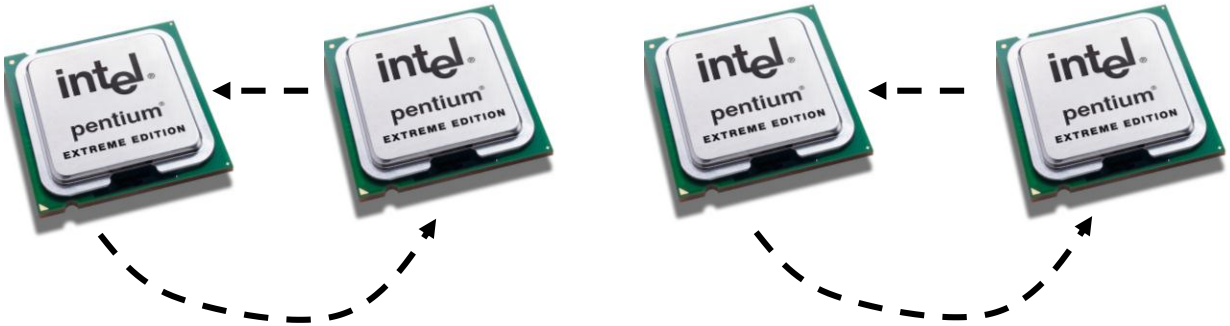
```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
If(rank==0)
{
  MPI_Recv(buf1,count,MPI_DOUBLE,1,tag,comm);
  MPI_Send(buf2,count,MPI_DOUBLE,1,tag,comm);
}
else if (rank==1)
{
  MPI_Recv(buf1,count,MPI_DOUBLE,0,tag,comm);
  MPI_Send(buf2,count,MPI_DOUBLE,0,tag,comm);
}
```




```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
If(rank==0)
{
  MPI_Recv(buf1,count,MPI_DOUBLE,1,tag,comm);
  MPI_Send(buf2,count,MPI_DOUBLE,1,tag,comm);
}
else if (rank==1)
{
  MPI_Send(buf2,count,MPI_DOUBLE,0,tag,comm);
  MPI_Recv(buf1,count,MPI_DOUBLE,0,tag,comm);
}
```

May survive
on some computers

Blocking point-to-point communication




```
int MPI_Sendrecv(
    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
    );
```

Example

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

int main(int argc, char **argv)
{
  int my_rank, ncpus;
  int left_neighbor, right_neighbor;
  int data_received;
  int send_tag = 101, recv_tag=101;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &ncpus);

  left_neighbor = (my_rank-1 + ncpus)%ncpus;
  right_neighbor = (my_rank+1)%ncpus;

  MPI_Sendrecv(&my_rank, 1, MPI_INT, left_neighbor, send_tag,
               &data_received, 1, MPI_INT, right_neighbor, recv_tag,
              MPI_COMM_WORLD, &status);

  printf("P%d received from right neighbor: P%d\n",
         my_rank, data_received);

  // clean up
  MPI_Finalize();
  return 0;
}
```

Output:
P3 received from right neighbor: P0
P2 received from right neighbor: P3
P0 received from right neighbor: P1
P1 received from right neighbor: P2

Non-blocking point-to-point communication

```
int MPI_Isend(
    void *buf,        /* initial address of send buffer                    */
    int     count,    /* number of elements in send buffer (nonnegative integer) */
    MPI_Datatype datatype, /* datatype of each send buffer element          */
    int     dest,     /* rank of destination (integer)                     */
    int     tag,      /* message tag (integer)                             */
    MPI_Comm    comm,       /*  communicator                               */
    MPI_Request  *request   /*  communication request   */
);
```

```
int MPI_Irecv(
    void *buf,        /* initial address of receive buffer  */
    int     count,    /* number of elements in receive buffer (nonnegative integer) */
    MPI_Datatype datatype, /* datatype of each receive buffer element          */
    int     dest,     /* rank of source (integer)                          */
    int     tag,      /* message tag (integer)                             */
    MPI_Comm      comm,      /* communicator                               */
    MPI_Request  *request /*  communication request */
);
```

MPI_Wait(all,any,some)

will be covered later

Communication Modes and Completion Criteria

There are <u>four communication modes</u> available for <u>sends</u>:

- Standard (MPI_SEND)
- Synchronous (MPI_SSEND)
- Buffered (MPI_BSEND)
- Ready (MPI_RSEND)

There is only *one mode* available *for receive (MPI_RECV)*

Communication Modes and Completion Criteria: Standard Mode

Standard mode send is MPI's general-purpose send mode.

When MPI executes a standard mode send, one of two things happens:
1. The message is copied into an MPI internal buffer and is transferred asynchronously to the destination process
2. The source and destination processes synchronize on the message.

The MPI implementation is free to choose (on a case-by-case basis) between buffering and synchronizing, depending on message size, resource availability, and so on. If the message is copied into an MPI internal buffer, then the send operation is formally completed as soon as the copy is done. If the two processes synchronize, then the send operation is formally completed only when the receiving process has posted a matching receive and actually begun to receive the message.

MPI_SEND does not return until the send operation it invoked has completed. *Completion can mean the message was copied into an MPI internal buffer, or it can mean the sending and receiving processes synchronized on the message.*
MPI_ISEND initiates a send operation and then **returns immediately**, without waiting for the send operation to complete. Completion has the same meaning as before: either the message was copied into an MPI internal buffer or the sending and receiving processes synchronized on the message.

Variables passed to MPI_ISEND cannot be used (should not even be read) until the send operation invoked by the call has completed.

One of the advantages of standard mode send is that the choice between buffering and synchronizing is left to MPI on a case-by-case basis.

Communication Modes and Completion Criteria: Synchronous Mode

Synchronous mode send requires MPI to synchronize the sending and receiving processes.

When a synchronous mode send operation is completed, the sending process may assume the destination process has **begun** receiving the message.
The destination process need not be done receiving the message, but it must have begun receiving the message.

The nonblocking call has the same advantages the nonblocking standard mode send has: the sending process can avoid blocking on a potentially lengthy operation.

Communication Modes and Completion Criteria: Ready Mode

Ready mode send **requires that a matching receive has** already **been posted** at the destination process before ready mode send is called.
If a matching receive has not been posted at the destination, the result is undefined. It is developers responsibility to make sure the requirement is met.

In some cases, knowledge of the state of the destination process is available without doing extra work. Communication overhead may be reduced because shorter protocols can be used internally by MPI when it is known that a receive has already been posted.

Communication Modes and Completion Criteria: Buffered Mode

Buffered mode send requires MPI to use buffering.
The downside is that developer is responsible for managing the buffer. If at any point, insufficient buffer is available to complete a call, the results are undefined.

The functions *MPI_BUFFER_ATTACH* and *MPI_BUFFER_DETACH* allow a program to make buffer available to MPI.

Collective Communications

Collective communication involves the **sending and receiving of data among processes.**

The collective routines are built using point-to-point communication routines.

Any collective communications can be substituted by MPI send and receive routines.

the "blackbox" (collective) routines hide a lot of the messy details and often implement the most efficient algorithm known for that operation (for that architecture,….).

Collective Communications

Collective communication routines transmit data among all processes in a **group**.

Collective communication calls do **no**t use the **tag mechanism** of send/receive for associating calls.

Rather, they are associated by order of program execution and because of this developer *must* ensure that all processors execute a given collective communication call.

Barrier Synchronization

int MPI_Barrier (comm)

The ***MPI_BARRIER*** routine blocks the calling process until all group processes have called the function. When MPI_BARRIER returns, all processes are synchronized at the barrier.

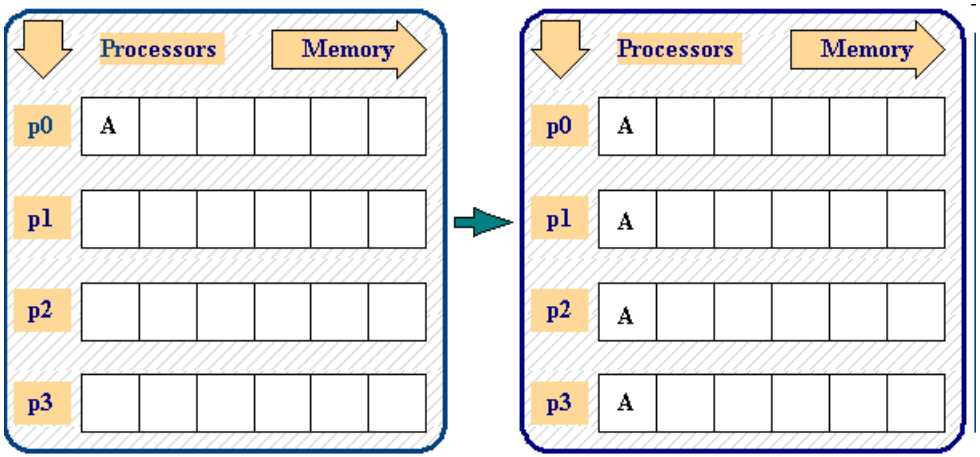
MPI_BARRIER can incur a substantial overhead on some machines.
In general, you should only insert barriers when they are truly needed.

MPI_Barries is often used for debugging
and performance evaluations

How would <u>you</u> substitute MPI_Barrier with point to point communication?

Broadcast Operation

The *MPI_BCAST* routine enables you to copy data from the memory of the root processor to the **same** memory locations for other processors in the communicator.




```
int MPI_Broadcast (
   void          *sendbuf,
   int            sendcnt,
   MPI_Datatype   sendtype,
   int            root,
   MPI_Comm       comm
);
```

How would <u>you</u> substitute MPI_Broadcast with point-to-point communication?

Broadcast Operation: example

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

int main(int argc, char **argv)
{
  int my_rank, ncpus;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &ncpus);

  double *parameters;
  int Nparameters;
  posix_memalign((void**)&parameters,16,Nparameters*sizeof(double));

  if (my_rank == 0)
     read_parameters_from_file(filename, parameters);

  MPI_Broadcast(parameters, Nparameters, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  //do more work

  MPI_Finalize();
  return 0;
}
```

<u>Broadcast</u> Operation: example

Ax=b →LUx=b → Ly=b, Ux=y

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} = \begin{bmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ L_{3,1} & L_{3,2} & L_{3,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ & U_{2,2} & U_{2,3} \\ & & U_{3,3} \end{bmatrix}$$

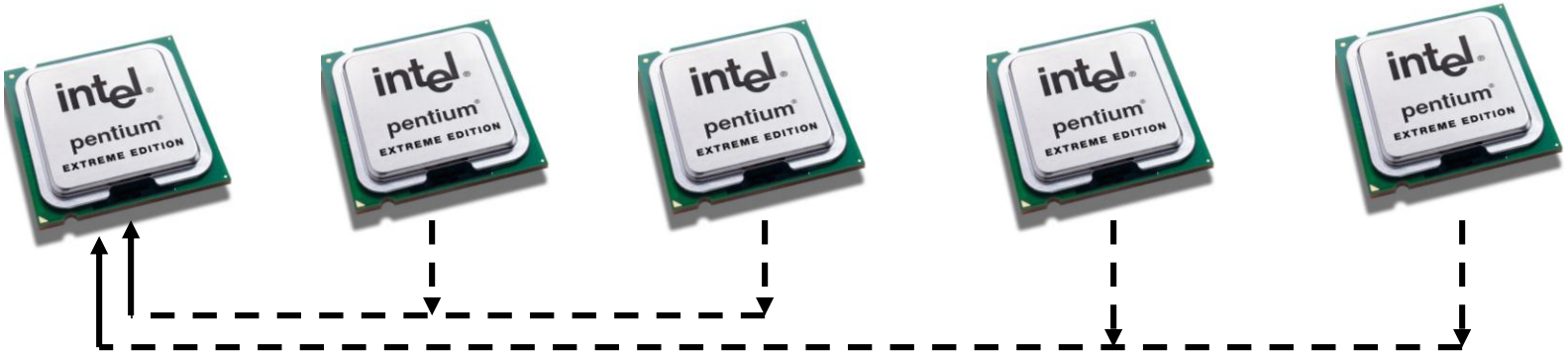
$$\begin{bmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ L_{3,1} & L_{3,2} & L_{3,2} \end{bmatrix} \begin{matrix} y_1 \\ y_2 \\ y_3 \end{matrix} = \begin{matrix} b_1 \\ b_2 \\ b_3 \end{matrix}$$

P0: solve for y_1 – and broadcast y_1
P1: solve for y_2 – and broadcast y_2

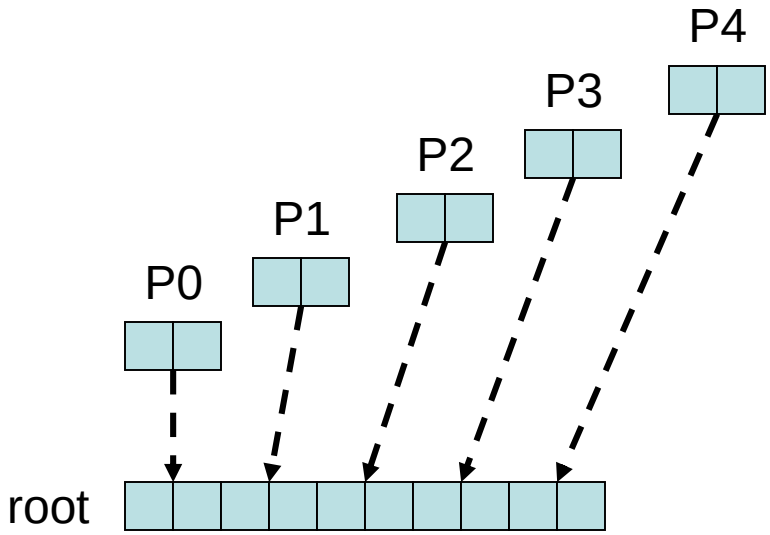
P3: solve for x_3 – and broadcast y_3
P2: solve for x_2 – and broadcast x_2

$$\begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ & U_{2,2} & U_{2,3} \\ & & U_{3,3} \end{bmatrix} \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} = \begin{matrix} y_1 \\ y_2 \\ y_3 \end{matrix}$$

Collective communication (all-to-one)

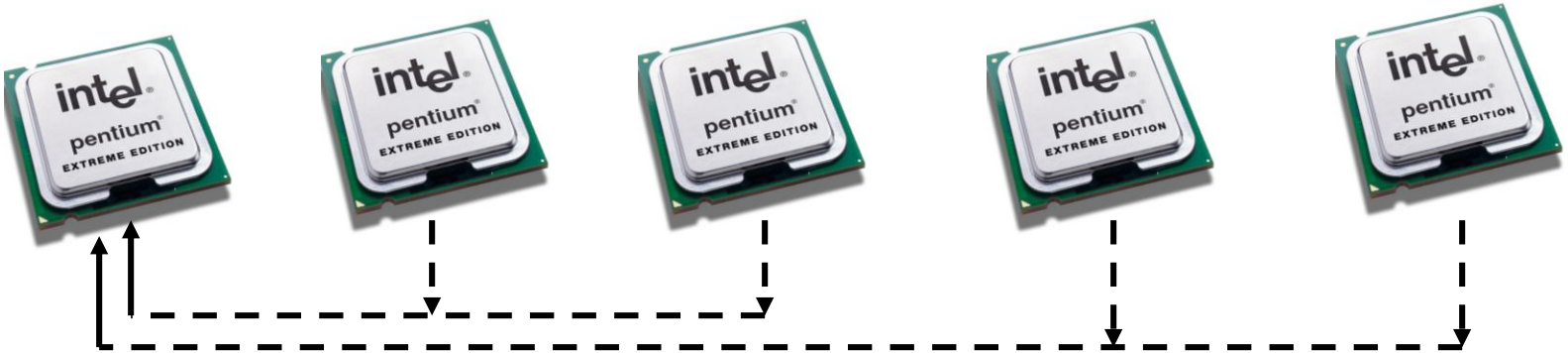


```
int MPI_Gather (
   void          *sendbuf,
   int            sendcnt,
   MPI_Datatype sendtype,
   void          *recvbuf,
   int            recvcount,
   MPI_Datatype recvtype,
   int            root,
   MPI_Comm       comm
);
```

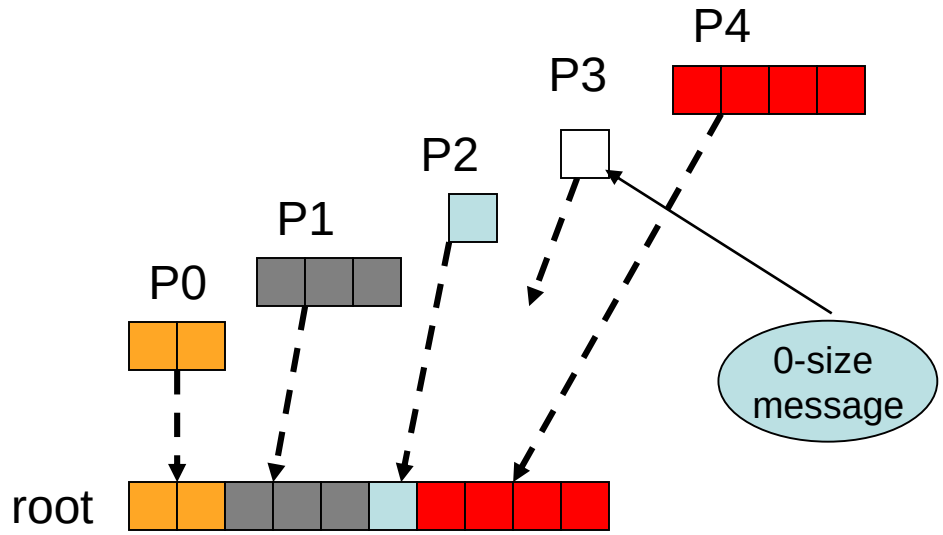


significant only at root

Collective communication (all-to-one)

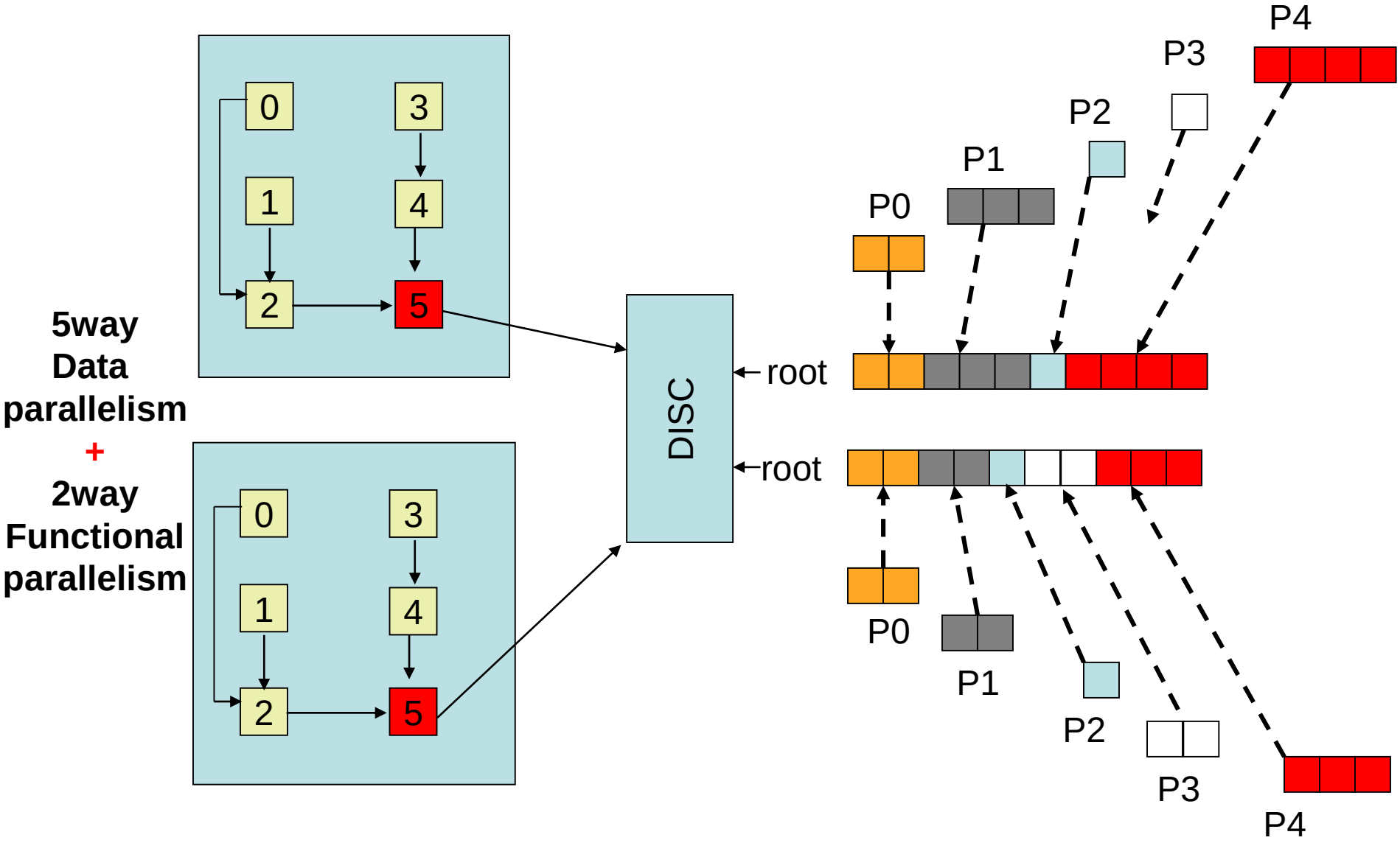


```
int MPI_Gatherv (
   void              *sendbuf,
   int                sendcnt,
   MPI_Datatype sendtype,
   void              *recvbuf,
   int               *recvcount,
   int               *displs,
   MPI_Datatype recvtype,
   int                root,
   MPI_Comm      comm
);
```



significant only at root

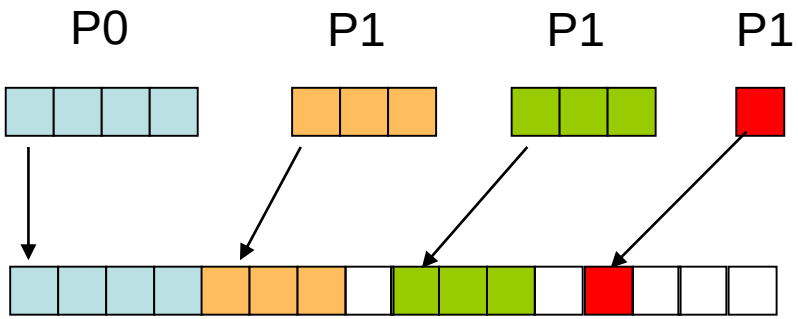
Gather(v): example



MPI_Gatherv: example

```
if (my_rank == 0){
    posix_memalign((void**)& rcvcnt, 16,comm_size*sizeof(int));
    posix_memalign((void**)& displs ,16,comm_size*sizeof(int));

    rcvcnt[0] = message_size_per_rank[0];
    displs[0] = 0;
    for (i = 1; i < comm_size; ++i){
        rcvcnt[i] = message_size_per_rank[i];
        displs[i] = displs[i-1]+rcvcnt[i-1];
    }
    MPI_Gatherv(sendbuf_local, message_size_local, MPI_DOUBLE,
                recv_buffer, rcvcnt, displs, MPI_DOUBLE,
                0, communicator);
    free(rcvcnt); free(displs);
 }
 else
    MPI_Gatherv(sendbuf_local, message_size_local, MPI_DOUBLE,
            NULL, NULL, NULL, MPI_DOUBLE,
            0, communicator);
```

Gather(v)



recvcnt = [4 3 3 1]

displs = [0 4 8 12]

displs[i] = i***stride**

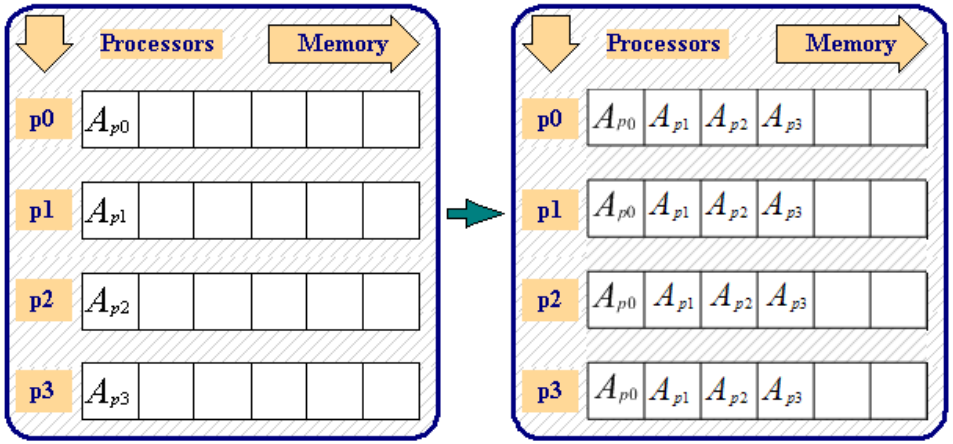
How would <u>you</u> substitute MPI_Gather(v) with point-to-point communication?

Collective communication (all-to-all)

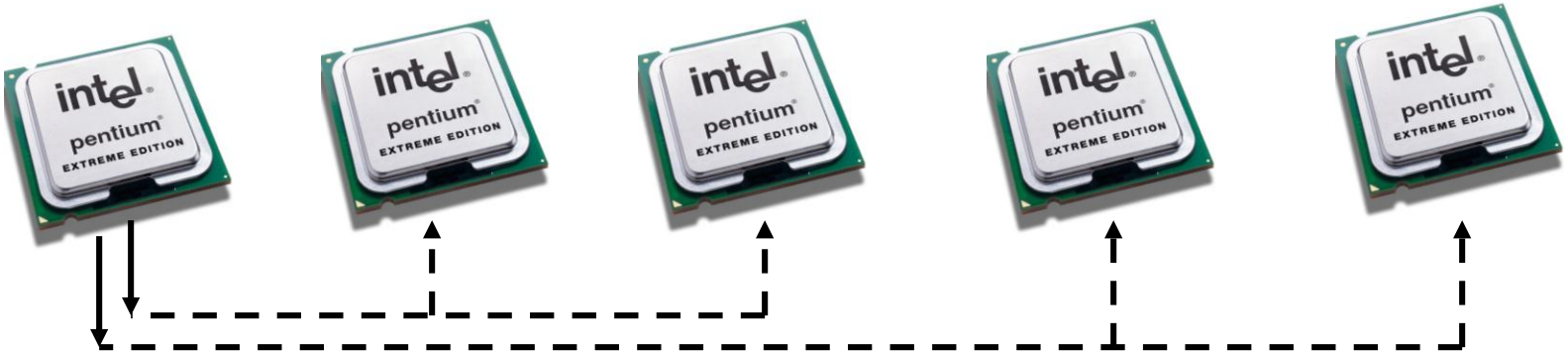
MPI_Allgather(v)

Gathers data **from all** tasks and distribute it **to all**

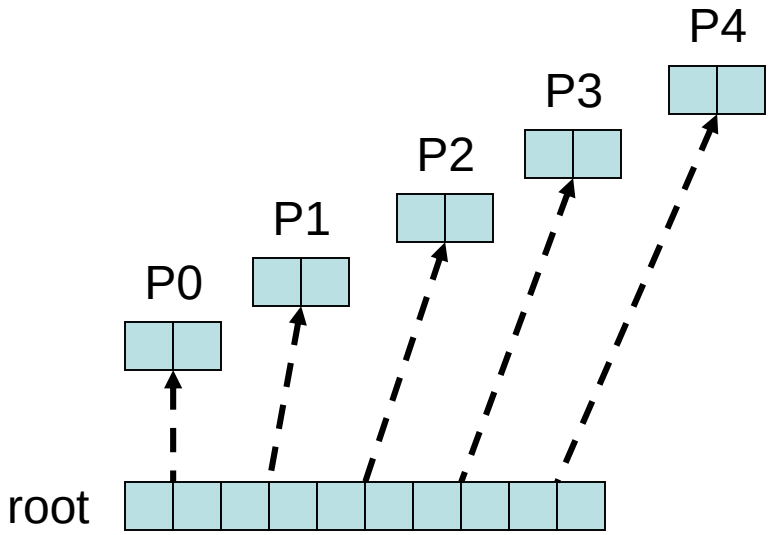
```
int MPI_Allgatherv (
   void              *sendbuf,
   int                sendcnt,
   MPI_Datatype sendtype,
   void              *recvbuf,
   int               *recvcount,
   int               *displs,
   MPI_Datatype recvtype,
   MPI_Comm     comm
);
```



Collective communication (one-to-all)

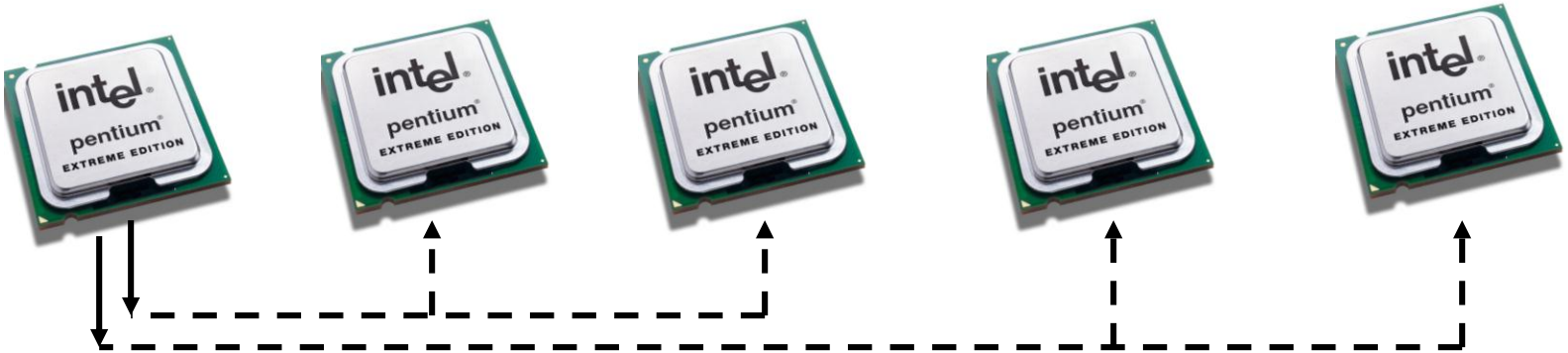


```
int MPI_Scatter (
   void          *sendbuf,
   int           sendcnt,
   MPI_Datatype  sendtype,
   void          *recvbuf,
   int           recvcount,
   MPI_Datatype  recvtype,
   int           root,
   MPI_Comm      comm
);
```

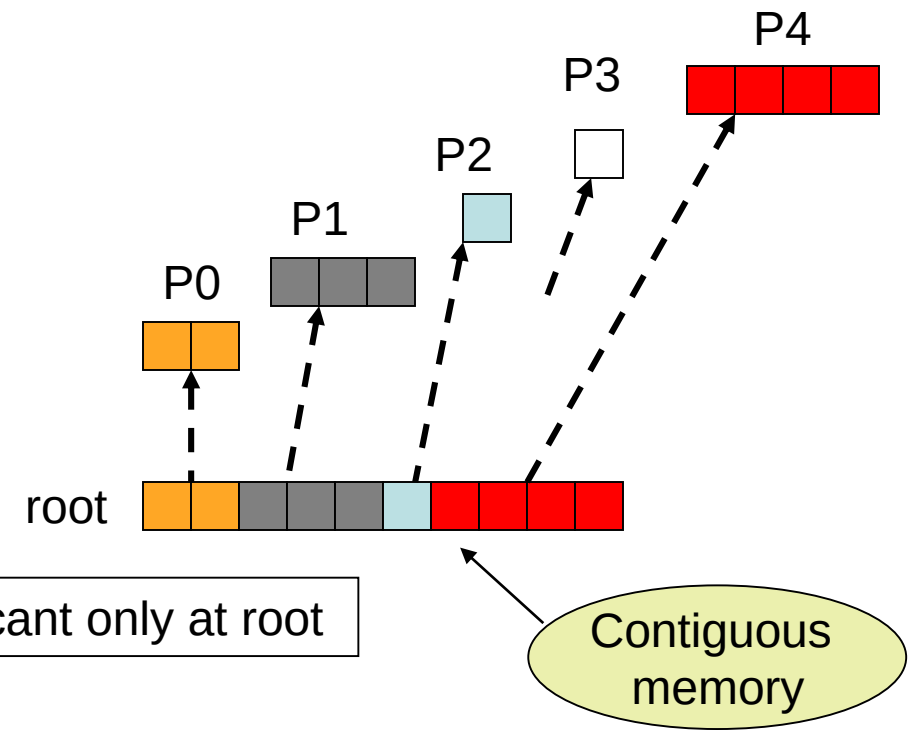


significant only at root

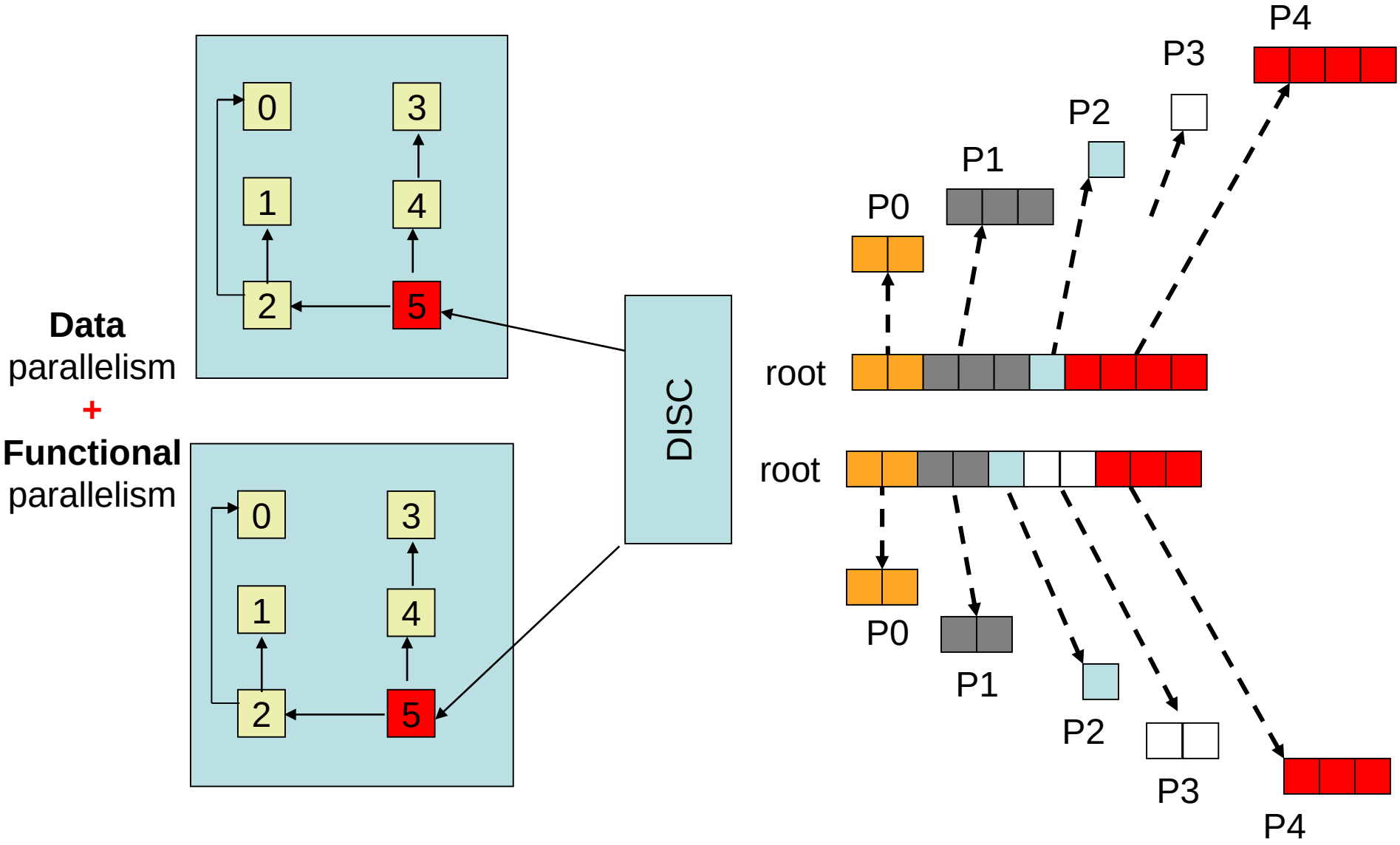
Collective communication (one-to-all)



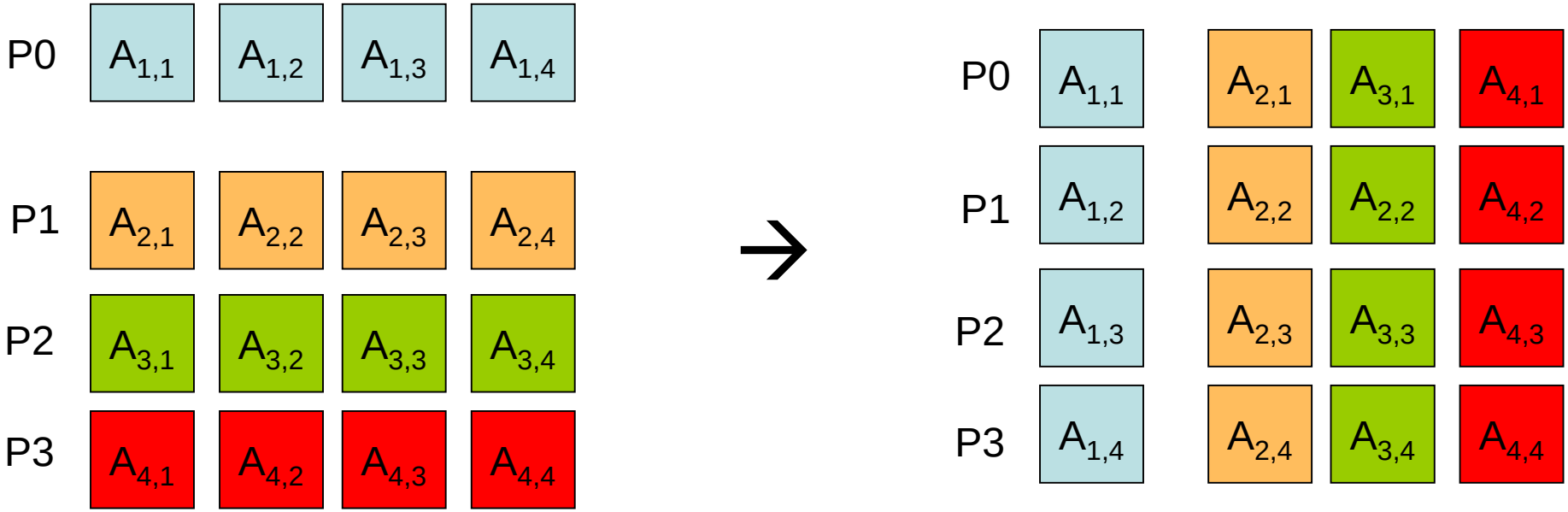
```
int MPI_Scatterv (
   void              *sendbuf,
   int               *sendcnt,
   int               *displs,
   MPI_Datatype sendtype,
   void              *recvbuf,
   int                recvcount,
   MPI_Datatype recvtype,
   int                root,
   MPI_Comm      comm
);
```



Scatter(v) operation: example

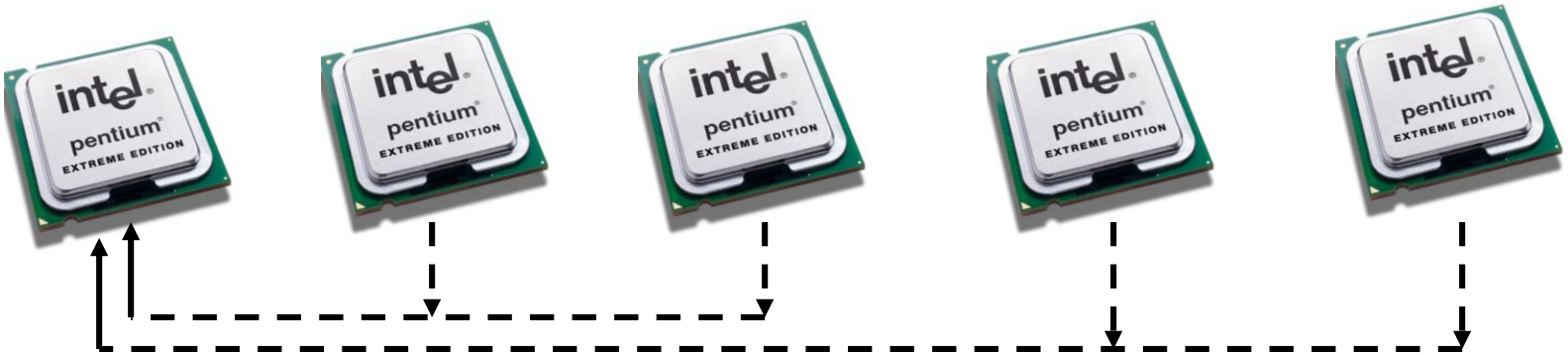


Scatter(v) operation: example A→A’




```
for (i = 0; i < comm_size; ++i)
    MPI_Scatterv(my_row,            sendcount, displs, MPI_DOUBLE,
                 recvbuf+offset[i], recvcount[i],      MPI_DOUBLE,
                 i,   /*  root */
                 communicator);
```

Reduction communication: all to one

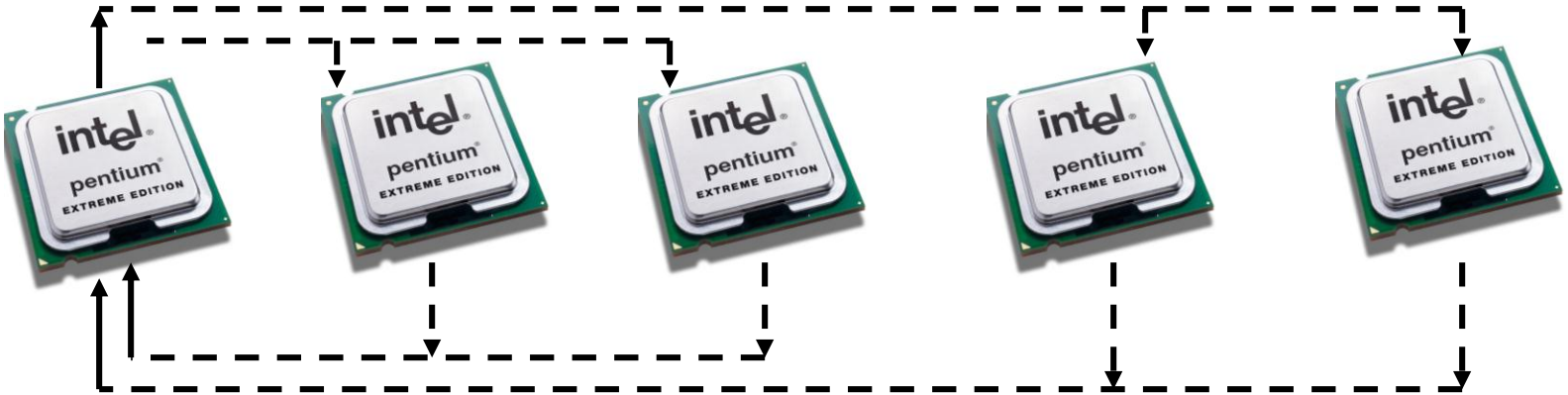



```
int MPI_Reduce (
    void          *sendbuf,
    void          *recvbuf,
    int            count,
    MPI_Datatype datatype,
    MPI_Op         op,
    int            root,
    MPI_Comm       comm
);
```

MPI function	Math Meaning
MPI_MAX maximum,	max
MPI_MIN minimum,	min
MPI_MAXLOC	maximum and location of maximum
MPI_MINLOC	minimum and location of minimum
MPI_SUM	sum
MPI_PROD	product
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

Implemented in integration, dot products, finding maxima or minima ….

Reduction communication: all to all




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

MPI function	Math Meaning
MPI_MAX maximum,	max
MPI_MIN minimum,	min
MPI_MAXLOC	maximum and location of maximum
MPI_MINLOC	minimum and location of minimum
MPI_SUM	sum
MPI_PROD	product
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

Implemented in integration, dot products, finding maxima or minima ….

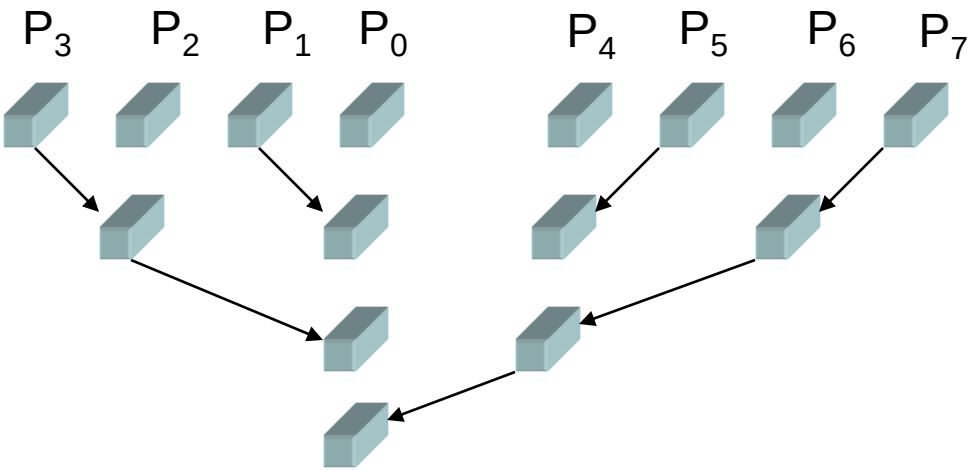
MPI_Allreduce: example

```
iter = 0;

while (error > TOL){

   u_new = parallel_solve(A,u_old);

   for (i = 0, errpr = 0; i < Nlocal; ++i){
       delta=(u_old-u_new);
       error +=  delta*delta;
   }

   MPI_Allreduce (&error,&delta,1,MPI_DOUBLE, MPI_SUM,communicator);
   error = sqrt(delta / Nglobal);

    if (iter > MAX_ITER) break;

    iter++;

    swap(u_old,u_new);

 }
```

stopping criteria must be identical on all processors!!!

MPI_Allreduce guarantees that the value of “delta” is identical in all ranks

(simple) reduction algorithm



$P_{i+1} \rightarrow P_i$

$P_{i+2} \rightarrow P_i$

$P_{i+4} \rightarrow P_i$

modern
core→ socket → node → rack
architecture requires more sophisticated algorithms

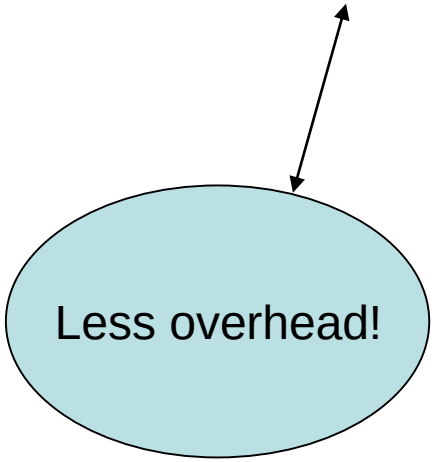
Reduction communication: example

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  int my_rank, ncpus;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &ncpus);

  double *x, *y;
  double y_max,y_max_global;
  int N;
  N = 300;
  posix_memalign((void**)&x,16,N*sizeof(double));
  posix_memalign((void**)&y,16,N*sizeof(double));

  eval_function(N,x,y);
  y_max = find_max(x,y);
  MPI_Reduce(&y_max,&y_max_global,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

  if (my_rank == 0)
      fprintf(stdout,"max(y) = %f\n",y_max_global);

  // clean up
  free(x); free(y);
  MPI_Finalize();
  return 0;
}
```

MPI_Alltoall

MPI_Alltoall allows each task to send specific data to all other tasks all **at once**!

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

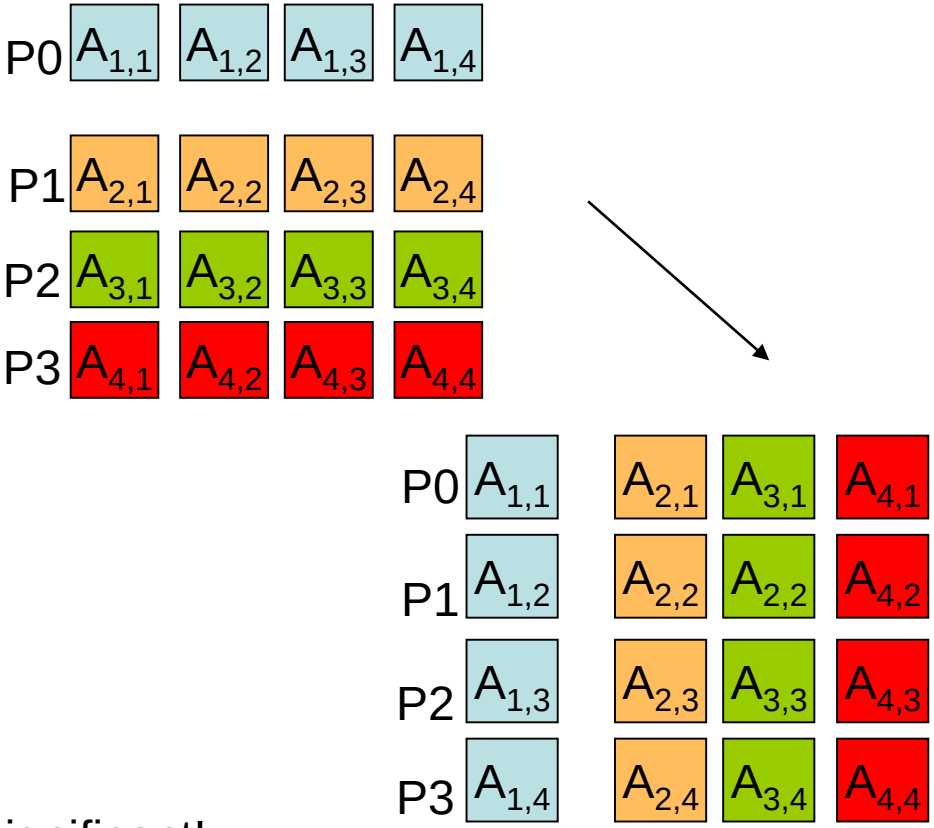


All arguments on all processes are significant!
Zero-size messages are OK.

MPI_Alltoall**v**

Sends data from all to all processes;
each process may send/recv a different amount of data **at once**!

```
int MPI_Alltoallv(
    void *sendbuf,
    int *sendcount,
    int *senddispls,
    MPI_Datatype sendtype,
    void *recvbuf,
    int *recvcount,
    int  *recvdispls,
    MPI_Datatype recvtype,
    MPI_Comm comm
);
```



All arguments on all processes are significant!

Back to point-to-point communication

Non-blocking point-to-point communications in "for" loops

and

MPI_Wait,

MPI_Waitall,

MPI_Waitany,

MPI_Waitsome

Example of a Data Parallel Problem (2D)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

0	1	2	3
4	5	6	7
8	**9**	10	11
12	13	14	15

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta x^2}$$

$$\boxed{u_i\left(1 + \frac{\Delta x^2}{\Delta y^2}\right) = 0.5\left(u_{i-1,j} + u_{i+1,j} + \frac{\Delta x^2}{\Delta y^2}(u_{i,j-1} + u_{i,j+1})\right) - 0.5\Delta x^2 f_{i,j}}$$

```
MPI_Request *recv_request, *send_request;

recv_request = new MPI_Request [Nneighbors *2];
send_request = recv_request + Nneighbors;
```

0	1	2	3
4	5	6	7
8	**9**	10	11
12	13	14	15

1

```
for (i = 0; i < Nneighbors; ++i)
    MPI_Irecv( recvbuf[i],
               recvcount[i],
               MPI_DOUBLE,
               neighbor[i],
               tag[i],
               communicator,
               &recv_request[i]);
```

2

```
for (i = 0; i < Nneighbors; ++i)
    fill_sendbuffer (i, sendbuf[i]);
```

3

```
for (i = 0; i < Nneighbors; ++i)
    MPI_Isend(sendbuf[i],
              sendcount[i],
              MPI_DOUBLE,
              neighbor[i],
              tag[i],
              communicator,
              &send_request[i] );
```

2 and 3 can also be combined (and "*properly*" ordered)

1 MPI_Irecv(…, &recv_request[i]);
2 prepare data;
3 MPI_Isend(…, &send_request[i]);

0	1	2	3
4	5	6	7
8	**9**	10	11
12	13	14	15

```
MPI_Waitall (Nneighbors, recv_request, MPI_STATUS_IGNORE);

for (i = 0; i < Nneighbors; ++i)
   procerss_recv_buffer(i, recvbuf[i]);

MPI_Waitall(Nneighbors, send_request, MPI_STATUS_IGNORE);
```

4

OR

```
MPI_Waitall(Nneighbors*2, recv_request, MPI_STATUS_IGNORE);

for (i = 0; i < Nneighbors; ++i)
   procerss_recv_buffer(i, recvbuf[i]);
```

4

OR

```
for (i = 0; i < Nneighbors; ++i){
   MPI_Wait(&recv_request[i], MPI_STATUS_IGNORE);
   procerss_recv_buffer(i, recvbuf[i]);
}
MPI_Waitall(Nneighbors, send_request, MPI_STATUS_IGNORE);
```

4

OR

1 MPI_Irecv(…, &recv_request[i]);
2 prepare data;
3 MPI_Isend(…, &send_request[i]);

0	1	2	3
4	5	6	7
8	**9**	10	11
12	13	14	15

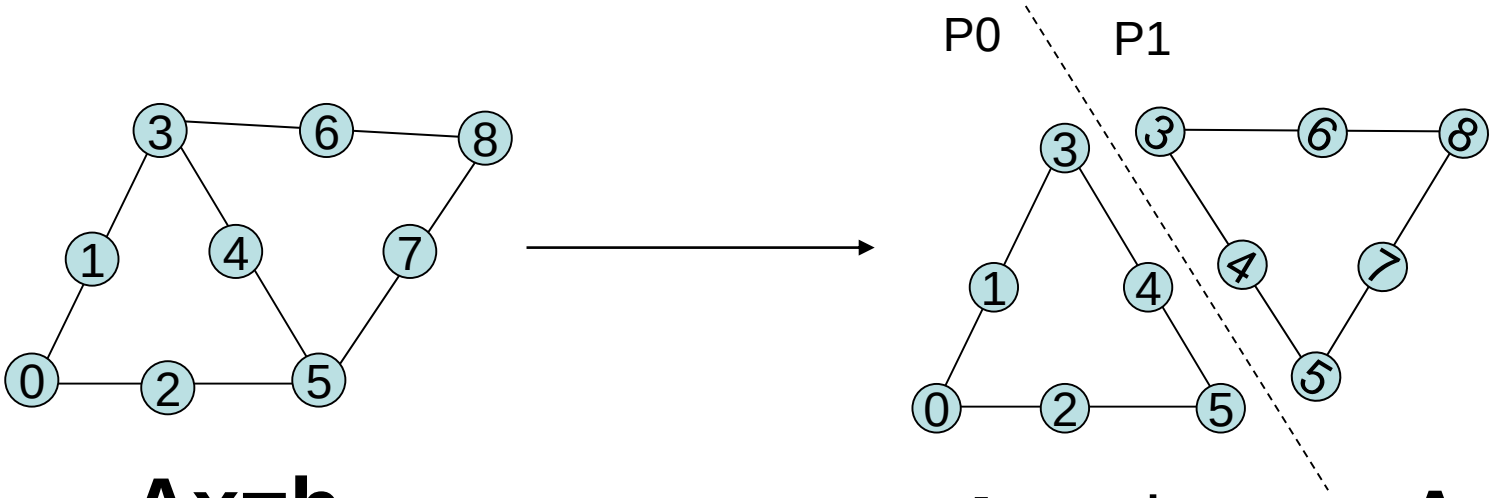
```
for (i = 0; i < Nneighbors; ++i){
   MPI_Waitany(Nneighbors, recv_request, &index, MPI_STATUS_IGNORE);
   procerss_recv_buffer(index, recvbuf[index]);
}
MPI_Waitall(Nneighbors, send_request, MPI_STATUS_IGNORE);
```

4

```
delete[] recv_request;
```

5

Element-wise matrix-vector multiplication



Ax=b

A is 9x9 operator

x is 9x1
b is 9x1

$\mathbf{A}_1 x_1 = b_1$

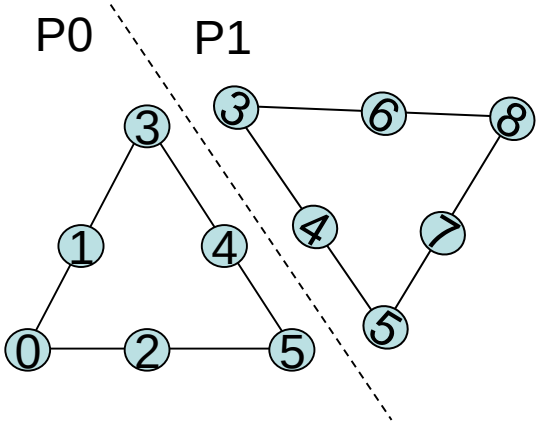
$\mathbf{A}_1$ is 6x6 operator

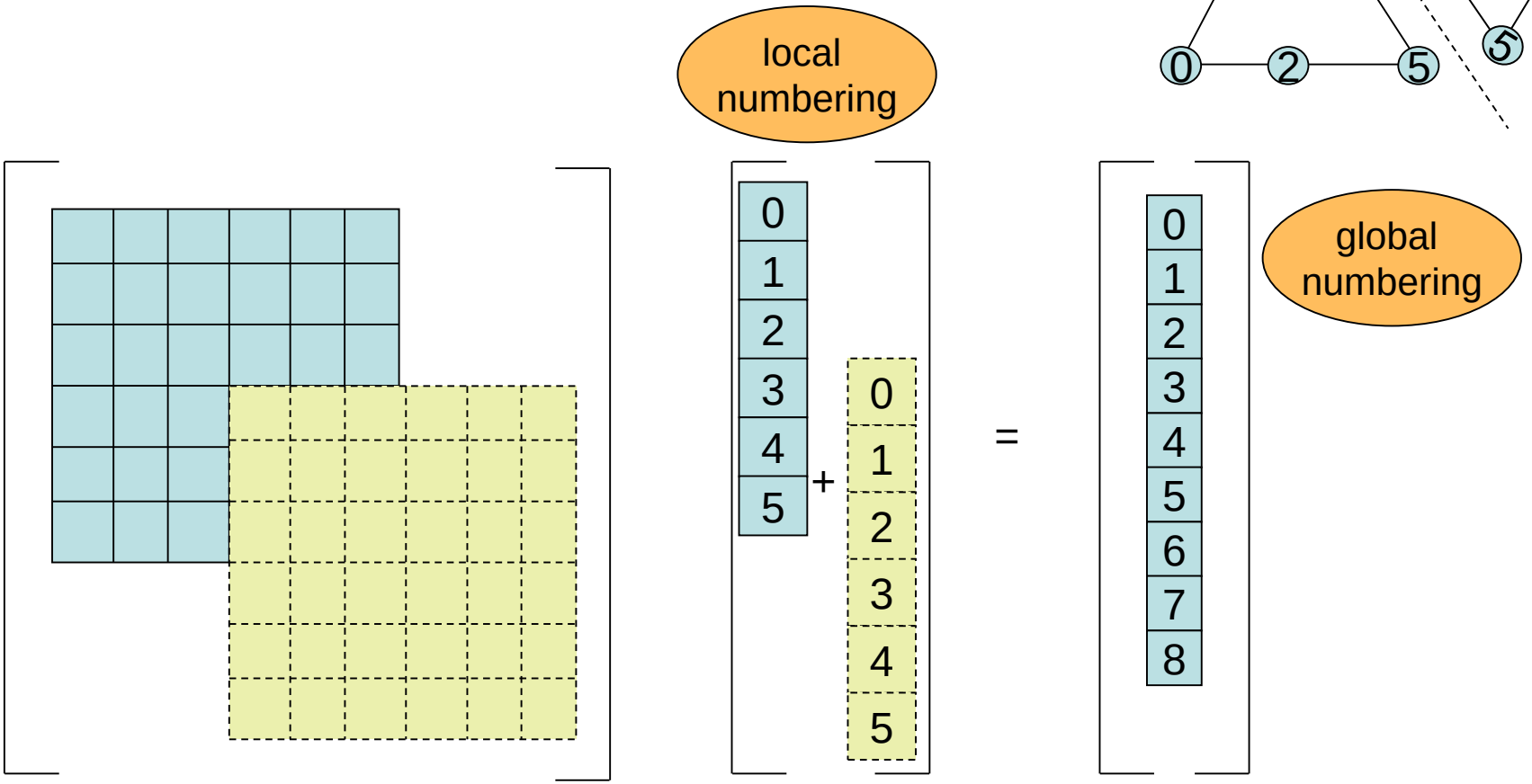
x_1 is 6x1
b_1 is 6x1

$\mathbf{A}_2 x_2 = b_2$

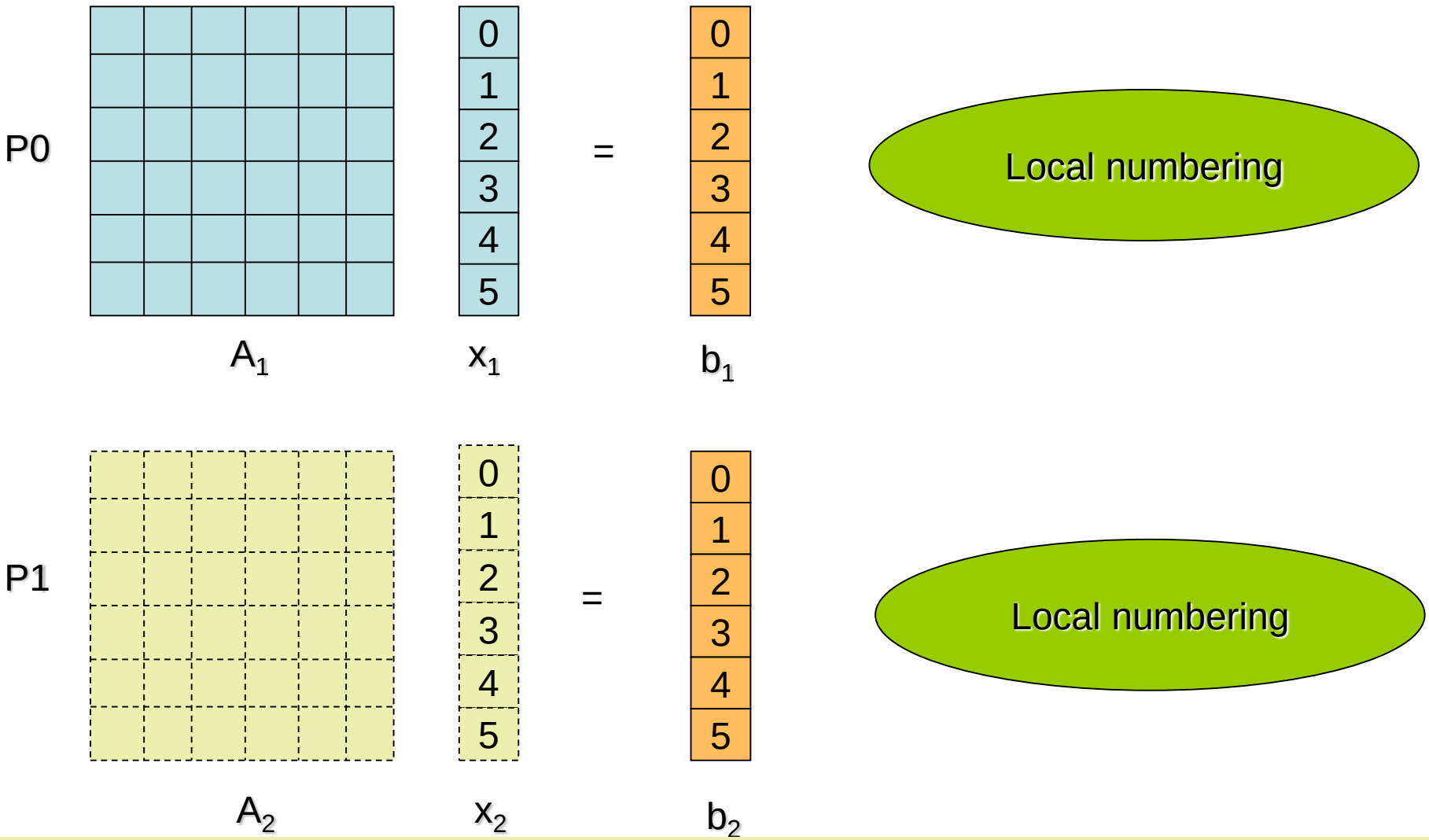
$\mathbf{A}_2$ is 6x6 operator
x_1 is 6x1
b_1 is 6x1

Element-wise matrix-vector multiplication

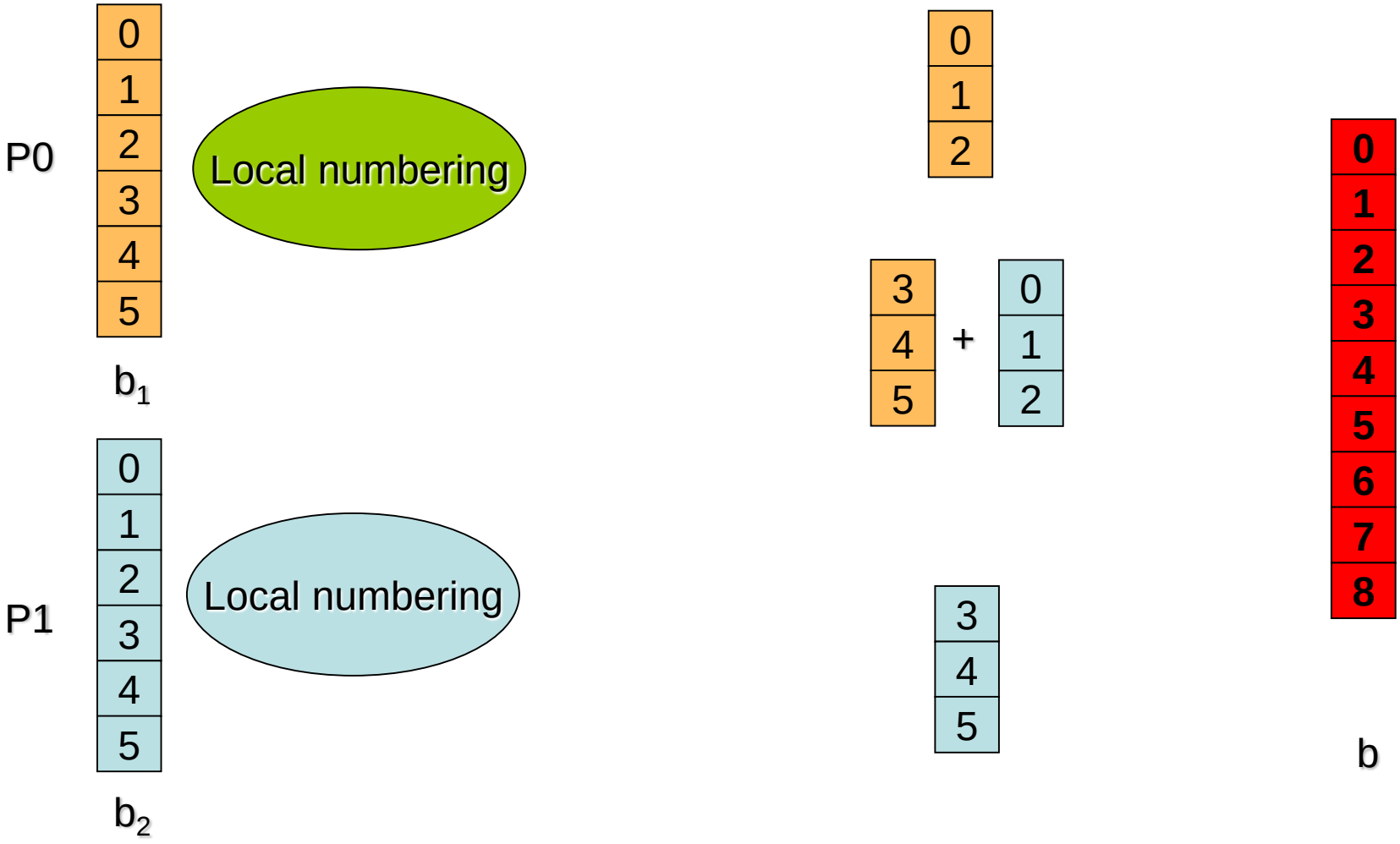




Element-wise matrix-vector multiplication



Element-wise matrix-vector multiplication



Element-wise matrix-vector multiplication

P0:

```
MPI_Irecv(recvbuf,3,MPI_DOUBLE,1,1,communicator,&recv_request);
for (i = 0; i < 3; ++0)
    sendbuf[i] = b[i+3];
MPI_Isend(sendbuf,3,MPI_DOUBLE,1,1,communicator,&send_request);
```

P1:

```
MPI_Irecv(recvbuf,3,MPI_DOUBLE,0,1,communicator,&recv_request);
for (i = 0; i < 3; ++0)
    sendbuf[i] = b[i];
MPI_Isend(sendbuf,3,MPI_DOUBLE,0,1,communicator,&send_request);
```

Element-wise matrix-vector multiplication

P0:

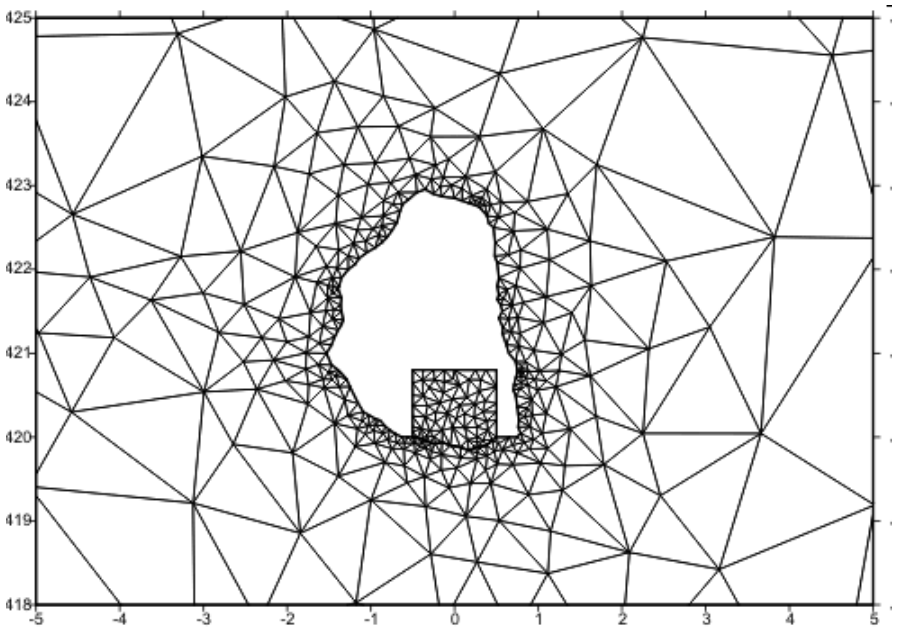
```
MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
for (i = 0; i < 3; ++0)
    b[i+3] = b[i+3]+recvbuf[i];
MPI_Wait(&send_request, MPI_STATUS_IGNORE);
```

P1:

```
MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
for (i = 0; i < 3; ++0)
    b[i] = b[i]+recvbuf[i];
MPI_Wait(&send_request, MPI_STATUS_IGNORE);
```

here we employ symmetric data exchange
(rank i exchanges data with rank j,
message sizes are identical)

Element-wise matrix-vector multiplication: general case (1)



Global IDs of degrees of freedom:

P0: [0 1 2 5 7 8 11 13 14 29 88]
P1: [2 3 5 6 10 12 14 80]
P3: [0 2 20 21 28 81]
.
.
.
P4095: [1 2 88 110 90111 90112]

Element-wise matrix-vector multiplication: general case (2)

Global IDs of degrees of freedom:

P0: [0 1 2 5 7 8 11 13 14 29 88]
P1: [2 3 5 6 10 12 14 80]
P3: [0 2 20 21 28 81]
.
.
.
P4095: [1 2 88 110 90111 90112]

1. Who are my neighbors ?

All ranks *can* be my neighbors

Only certain subset of ranks include my neighbors

This is going to be messy ….
Need a lot of memory for mapping and/or a lot of communications

I am in better shape:
will need less memory and less communications!

Element-wise matrix-vector multiplication: general case (3)

Global IDs of degrees of freedom:

P0: [0 1 2 5 7 8 11 13 14 29 88]
P1: [2 3 5 6 10 12 14 80]
P3: [0 2 20 21 28 81]
.
.
.
P4095: [1 2 88 110 90111 90112]

2. How many d.o.f each my of potential neighbors has?

All ranks *can* be my neighbors

Only certain subset of ranks include my neighbors

Use MPI_Allgather to collect the number of d.o.f

Use MPI_Isend / MPI_Irecv to collect the number of d.o.f of neighbors

Element-wise matrix-vector multiplication: general case (4)

Global IDs of degrees of freedom:

P0: [0 1 2 5 7 8 11 13 14 29 88]
P1: [2 3 5 6 10 12 14 80]
P3: [0 2 20 21 28 81]
.
.
.
P4095: [1 2 88 110 90111 90112]

2. What d.o.f each my of potential neighbors has?

All ranks *can* be my neighbors

Only certain subset of ranks include my neighbors

Use MPI_Allgatherv to collect the neighbors d.o.f

Use MPI_Alltoallv or MPI_Isend / MPI_Irecv to collect neighbors d.o.f

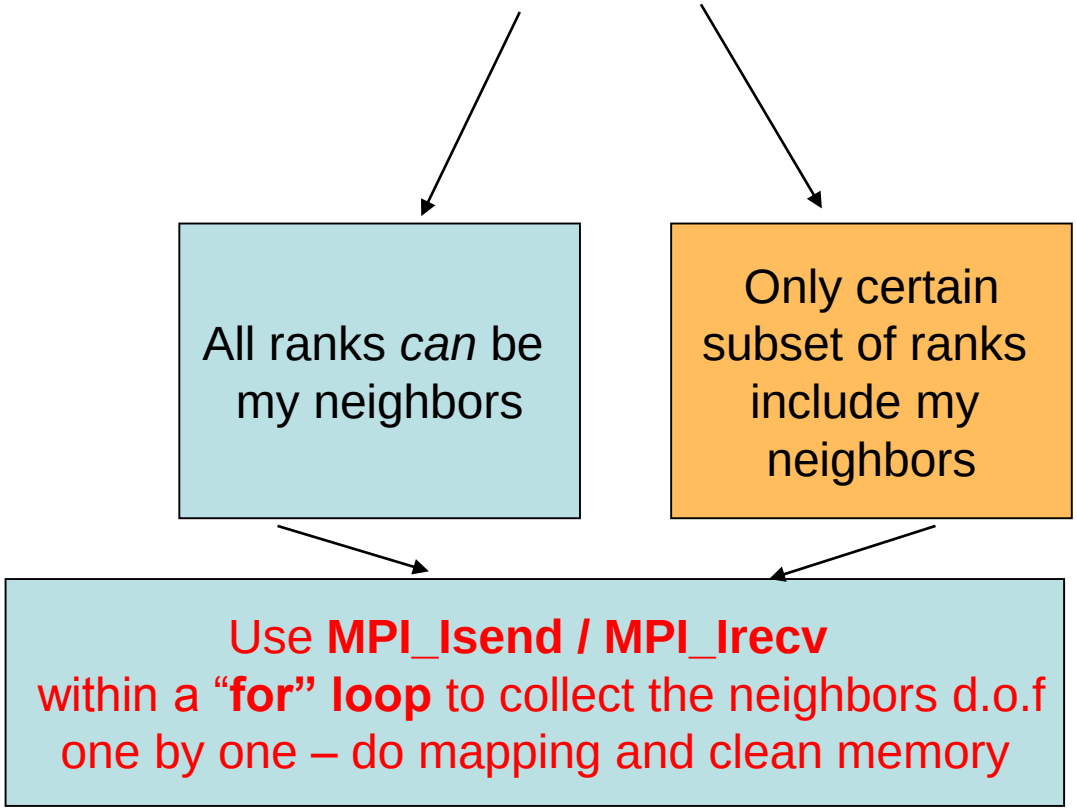
May run out of memory ….

Element-wise matrix-vector multiplication: general case (5)

Global IDs of degrees of freedom:

P0: [0 1 2 5 7 8 11 13 14 29 88]
P1: [2 3 5 6 10 12 14 80]
P3: [0 2 20 21 28 81]
.
.
.
P4095: [1 2 88 110 90111 90112]

2. What d.o.f each my of potential neighbors has?



(Alternative option is to use bit array)

Element-wise matrix-vector multiplication: general case (6)

```
for (partner = 0; partner < Npartners; ++ partner ){
   for (i = 0, ii=0; i < n; ++i){
        for (j = 0; j < partner_map_size[partner]; ++j){
          if (map[i] == partners_map[partner][j]){
              message_send_map[partner][ii] = i;
              ii++;
              break;
          } // end of if (map[i]
        }
      } // end of "for (i = 0,…"

     for (j = 0, jj = 0; j < partner_map_size[partner]; ++j){
       for (i = 0; i < n; ++i){
          if (map[i] == partners_map[partner][j]){
             message_recv_map[partner ][jj] = i;
             jj++;
             break;
          } // end of "if (map[i]…."
       }
     } // end of "for (j = 0,…"
}
```

mapping
local d.o.f ← → neighbors d.o.f

Element-wise matrix-vector multiplication: general case (7)

```
void NEKTAR_MEX::MEX_plus(double *val){
  double *dp;
  int *map;
  int i,j,partner,index;

  MEX_post_recv();

  for (partner = 0; partner < Npartners; ++partner){
    dp = send_buffer[partner];
    map = message_send_map[partner];
    for (i = 0; i < message_size[partner]; ++i)
      dp[i] = local_values[map[i]];
  }
  MEX_post_send();

  for (i = 0; i < Npartners; i++){
     MPI_Waitany(Npartners,request_recv,&index,MPI_STATUS_IGNORE);
     dp = recv_buffer[index];
     map = message_recv_map[index];
     for (j = 0; j < message_size[index]; ++j)
       local_values [map[j]] += dp[j];
  }
  MPI_Waitall(Npartners,request_send,MPI_STATUS_IGNORE);
}
```

Global summation

How to learn programming with MPI ?

Just do it!