

High Performance Computing. MPI and PETSc

Mario Storti

Centro Internacional de Métodos Numéricos
en Ingeniería - CIMEC

INTEC, (CONICET-UNL), Santa Fe, Argentina

`<mario.storti at gmail.com>`

<http://www.cimec.org.ar/mstorti>,

`(document-version "curspar-1.0.0-158-gfceed34 'clean")`

February 19, 2010

Contents

- **6.....MPI - Message Passing Interface**
 - ▷ **14.....Basic use of MPI**
 - ▷ **25.....Point to point communication**
 - ▷ **40.....Global communication**
 - * **55.....Example: Computing Pi**
 - * **72.....Example: Prime Number Theorem**
 - ▷ **122.....Load balance**
 - * **133.....The traveling salesman problem (TSP)**
 - * **155.....Computing PI by Montecarlo**
 - * **164.....Example: matrix product in parallel**
 - * **184.....Game of life**
 - * **224.....The Poisson equation**
 - ▷ **255.....Advanced MPI collective operations**
 - ▷ **300.....Definiendo tipos de datos derivados**
 - ▷ **310.....Example: The Richardson iterative method**
- **313.....PETSc**
 - ▷ **316.....PETSc Objects**
 - ▷ **320.....Using PETSc**

- ▷ **339.....PETSc elements**
- ▷ **353.....Programa de FEM basado en PETSc**
 - * **355.....Basic concepts on partitioning**
 - * **359.....FEM code**
 - * **370.....SNES: non-linear solvers**
- ▷ **375.....Assignement**
- **375.....Assignement**
- **382.....FAQ, tips and common pitfalls with MPI, PETSc and OpenMP**

About this course

- Material on this course (including these slides) can be found in <http://www.cimec.org.ar/twiki/bin/view/Cimec/CursoHPCenMC2>. (Note: in case it fails, try [this alternative direction](#).)
- All the examples in these slides are available at *clonetroop* in directory */clonetroop/storti/example*, or either as *examples.tar.gz* file in the web page above.
- You can reach me at mario.storti at gmail.com, my phone number in Barcelona is 934015696 (until Feb 25, 2010).
- There is an assignment to be done by the students. The assignment is due Monday February 22 at 12:00hs. You should send by email to me (mario.storti at gmail.com) a ZIP or TAR file with the PDF of the assignment and the sources of all the programs that you have used. Please include also in the PDF the relevant sources that you have used. To be clear, the source code of the examples that you have typed must be included in the PDF and also as separate source files in the ZIP/TAR file. (The rationale of this is that the version in the PDF is for reading, while the source files is for the case I want to try to compile and run it.)
- You can do all the work in C/C++ or Fortran, whatever you prefer. You can

also make auxiliary manipulations with Matlab/Octave or other similar soft, for instance in order to compute the bandwidth, you can write a table of n, t (message length and communication time) values to a file and then read it with Matlab and process it.

MPI - Message Passing Interface

The MPI Forum

- At the beginning of the 90's the large number of commercial and free solutions forced users to take a series of decisions compromising *portability*, and *performance*.
- In April 1992 the “*Center of Research in Parallel Computation*” organized a workshop for the definition of standards for Message Passing and distributed memory environments. As a result, an agreement was achieved on establishing a standard for Message Passing.

The MPI Forum (cont.)

- In November 1992 in the Supercomputing'92 conference a committee was appointed to define a standard of Message Passing. The objectives were:
 - ▷ To define a standard of Message Passing libraries. It would not be an official standard like ANSI, but it should tempt users and implementors of the library and related applications.
 - ▷ To work with a completely *open* philosophy. Anyone should be able to access the discussions, assisting to meetings or via e-mail discussions.
 - ▷ The standard should be defined in 1 year.
- The MPI Forum decided to follow the workflow of the HPF Forum.
- Most of the related groups participated in the Forum: Vendors like Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC, Thinking Machines. Members of preexisting libraries like PVM, p4, Zipcode, Chameleon, PARMACS, TCGMSG, Express.
- Meetings were scheduled every 6 weeks during 1 year and electronic discussion was intense (archives are available at www.mpiforum.org).
- The standard was finished in May 1994.

What is MPI?

- MPI = ***“Message Passing Interface”***
- ***“Message Passing Interface”***
- Message Passing means
 - ▷ Each process is a standalone sequential program.
 - ▷ All data is private to each process.
 - ▷ Communication is performed via library function calls.
 - ▷ The underlying language is standard: Fortran, C, (F90, C++)...
- MPI is SPMD (Single Program Multiple Data)

Reasons for parallel processing

- More computing power
- Hardware with efficient cost/performance relations from inexpensive components (COTS).
- Start from little, with growing expectations.

Needs of parallel computing

- Portability (current and future)
- Scalability (hardware and software)

References

- *Using MPI: Portable Parallel Programming with the Message Passing Interface*, W. Gropp, E. Lusk and A. Skeljum. MIT Press 1995
- *MPI: A Message-Passing Interface Standard*, June 1995 (accessible at <http://www.mpiforum.org>)
- *MPI-2: Extensions to the Message-Passing Interface* November 1996, (accessible at <http://www.mpiforum.org>)
- *MPI: the complete reference*, by Marc Snir, Bill Gropp, MIT Press (1998) (available in electronic format, *mpi-book.ps*, *mpi-book.pdf*).
- *Parallel Scientific Computing in C++ and MPI: A Seamless approach to parallel algorithms and their implementations*, by G. Karniadakis y RM Kirby, Cambridge U Press (2003) (u\$s 44.00)
- (*man pages*) available at <http://www-unix.mcs.anl.gov/mpi/www/>
- Newsgroup *comp.parallel.mpi* (accesible via <http://groups.google.com/group/comp.parallel.mpi>, pretty dead actually, but useful discussion in the past).
- Discussion lists for [MPICH](#) and [OpenMPI](#)

Other message-passing libraries

- **PVM (Parallel Virtual Machine)**
 - ▷ Historically have a GUI and process management.
 - ▷ Can add hosts dynamically.
- **P4: Shared Memory model**
- **Other commercial products currently obsolete.**
- **Research projects, many times associated to a particular architecture.**

MPICH

A portable implementation of MPI developed at Argonne National Labs (ANL) (USA)

- **MPE: A library of graphical routines, debugger interface, logging.**
- **MPIRUN, MPIEXEC: portable scripts for launching parallel processes.**
- **Implemented on top of P4.**
- **Another open-source popular implementation of MPI is Open-MPI (previously known as LAM-MPI).**

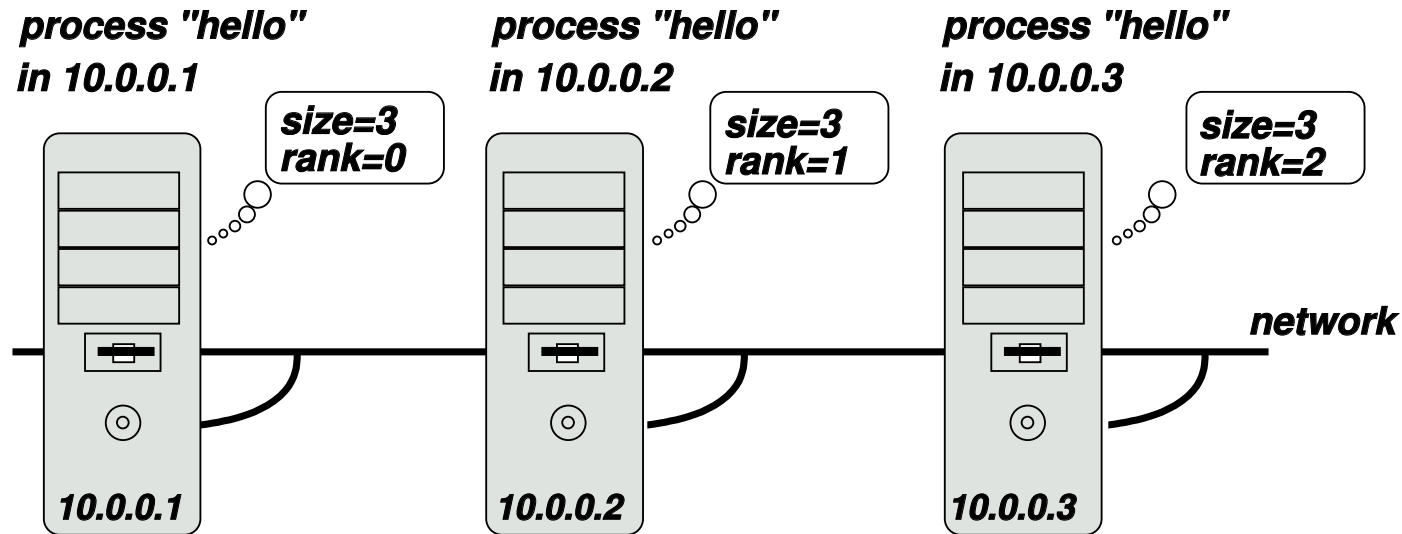
Basic use of MPI

Hello world in C

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char **argv) {
5     int ierror, rank, size;
6     MPI_Init(&argc, &argv);
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8     MPI_Comm_size(MPI_COMM_WORLD, &size);
9     printf("Hello world. I am %d out of %d.\n", rank, size);
10    MPI_Finalize();
11 }
```

- All programs start with *MPI_Init()* and end with *MPI_Finalize()*.
- *MPI_Comm_size()* returns the total number *size* of processes involved in this parallel run. *MPI_Comm_rank()* returns through *rank*, the id of the process in this parallel run ($0 \leq \text{myrank} < \text{size}$).

Hello world in C (cont.)



- At the moment of launching the program in parallel (we will see how it is done below) a copy of the program starts execution in each of the selected nodes. In the figure it runs on 3 nodes.
- Each process obtains a unique *id* (usually called *rank*, *myrank*.)
- **Oversubscription:** In general we can have more than one *process* per *processor* (but it may not be useful, though).

Hello world in C (cont.)

- If we compile and execute *hello.bin*, then when running we obtain the normal output.

```
1 [mstorti@spider example]$ ./hello.bin
2 Hello world. I am 0 out of 1.
3 [mstorti@spider example]$
```

- In order to run it on several nodes we generate a *machi.dat* file, with the processors names one per line.

```
1 [mstorti@spider example]$ cat ./machi.dat
2 node1
3 node2
4 [mstorti@spider example]$ mpirun -np 3 -machinefile \
5                               ./machi.dat hello.bin
6 Hello world. I am 0 out of 3.
7 Hello world. I am 1 out of 3.
8 Hello world. I am 2 out of 3.
9 [mstorti@spider example]$
```

The *mpirun* script, which is part of the MPICH distribution, launches a copy of *hello.bin* in the processor where *mpirun* has been called and two processes in the nodes corresponding to the first two lines of *machi.dat*.



- It is normal that each process in the parallel run **sees** the same directory at the server via **NFS**.
- Each process can open its own files for reading or writing, with the same rules that process in a UNIX environment must respect.
- Several process may open the same file for reading.
- Normally only one process (id 0) reads from **stdin**.
- All processes can write to **stdout**, but output may get scrambled (there is not a defined temporal order).

Hello world in Fortran

```
1  PROGRAM hello
2  IMPLICIT NONE
3  INCLUDE "mpif.h"
4  INTEGER ierror, rank, size
5  CALL MPI_INIT(ierror)
6  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
7  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
8  WRITE(*,*) 'Hello world. I am ',rank,' out of ',size
9  CALL MPI_FINALIZE(ierror)
10 STOP
11 END
```

Master/Slave strategy with SPMD (in C)

```
1  // ...
2  int main(int argc, char **argv) {
3      int ierror, rank, size;
4      MPI_Init(&argc, &argv);
5      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6      MPI_Comm_size(MPI_COMM_WORLD, &size);
7      // ...
8      if (rank==0) {
9          /* master code */
10     } else {
11         /* slave code */
12     }
13     // ...
14     MPI_Finalize();
15 }
```

MPI function call format

- C:

```
int ierr=MPI_Xxxxx(parameter,....); ó  
MPI_Xxxxx(parameter,....);
```

- Fortran:

```
CALL MPI_XXXX(parameter,....,ierr);
```

Error codes

- Error codes are rarely used.
- Proper usage is like this:

```
1 ierror = MPI_Xxxx(parameter, . . . .);  
2 if (ierror != MPI_SUCCESS) {  
3     /* deal with failure */  
4     abort();  
5 }
```

MPI is small - MPI is large

Moderately complex programs can be written with just 6 functions:

- ***MPI_Init*** - It's used once at initialization.
- ***MPI_Comm_size*** Identify how many processes participate in this parallel run.
- ***MPI_Comm_rank*** Identify the id of my process in the parallel run.
- ***MPI_Finalize*** Last function to be called. Ends MPI.
- ***MPI_Send*** Sends a message to another process (point to point).
- ***MPI_Recv*** Receives a message sent by other process.

MPI is small - MPI is large (cont.)

Collective communication

- *MPI_Bcast* Sends a message to all other process.
- *MPI_Reduce* Combines data in all process to only one process.

The complete MPI standard has 125 functions.

Point to point communication

Send a message

- **Template:**

`MPI_Send(address, length, type, destination, tag, communicator)`

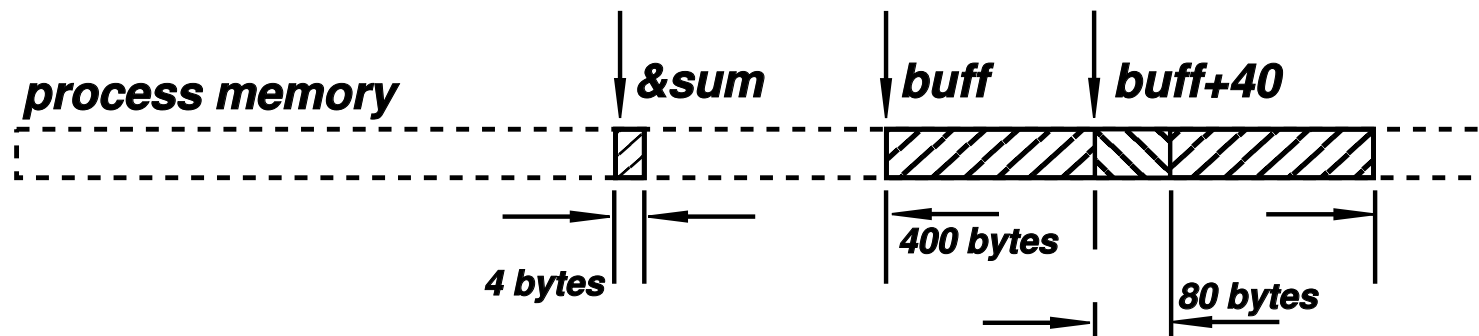
- **C:**

`ierr = MPI_Send(&sum, 1, MPI_FLOAT, 0, mtag1, MPI_COMM_WORLD);`

- **Fortran (note extra parameter):**

`call MPI_SEND(sum, 1, MPI_REAL, 0, mtag1, MPI_COMM_WORLD, ierr);`

Send a message (cont.)



```

1  int buff[100];
2  // Llenar buff. .
3  for (int j=0; j<100; j++) buff[j] = j;
4  ierr = MPI_Send(buff, 100, MPI_INT, 0, mtag1,
5                MPI_COMM_WORLD);
6
7  int sum;
8  ierr = MPI_Send(&sum, 1, MPI_INT, 0, mtag2,
9                MPI_COMM_WORLD);
10
11 ierr = MPI_Send(buff+40, 20, MPI_INT, 0, mtag2,
12                MPI_COMM_WORLD);
13
14 ierr = MPI_Send(buff+80, 40, MPI_INT, 0, mtag2,
15                MPI_COMM_WORLD); // Error! Region sent extends
16                                // beyond the end of buff

```

Receive a message

- **Template:**

MPI_Recv(address, length, type, source, tag, communicator, status)

- **C:**

ierr = MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE, mtag1, MPI_COMM_WORLD, &status);

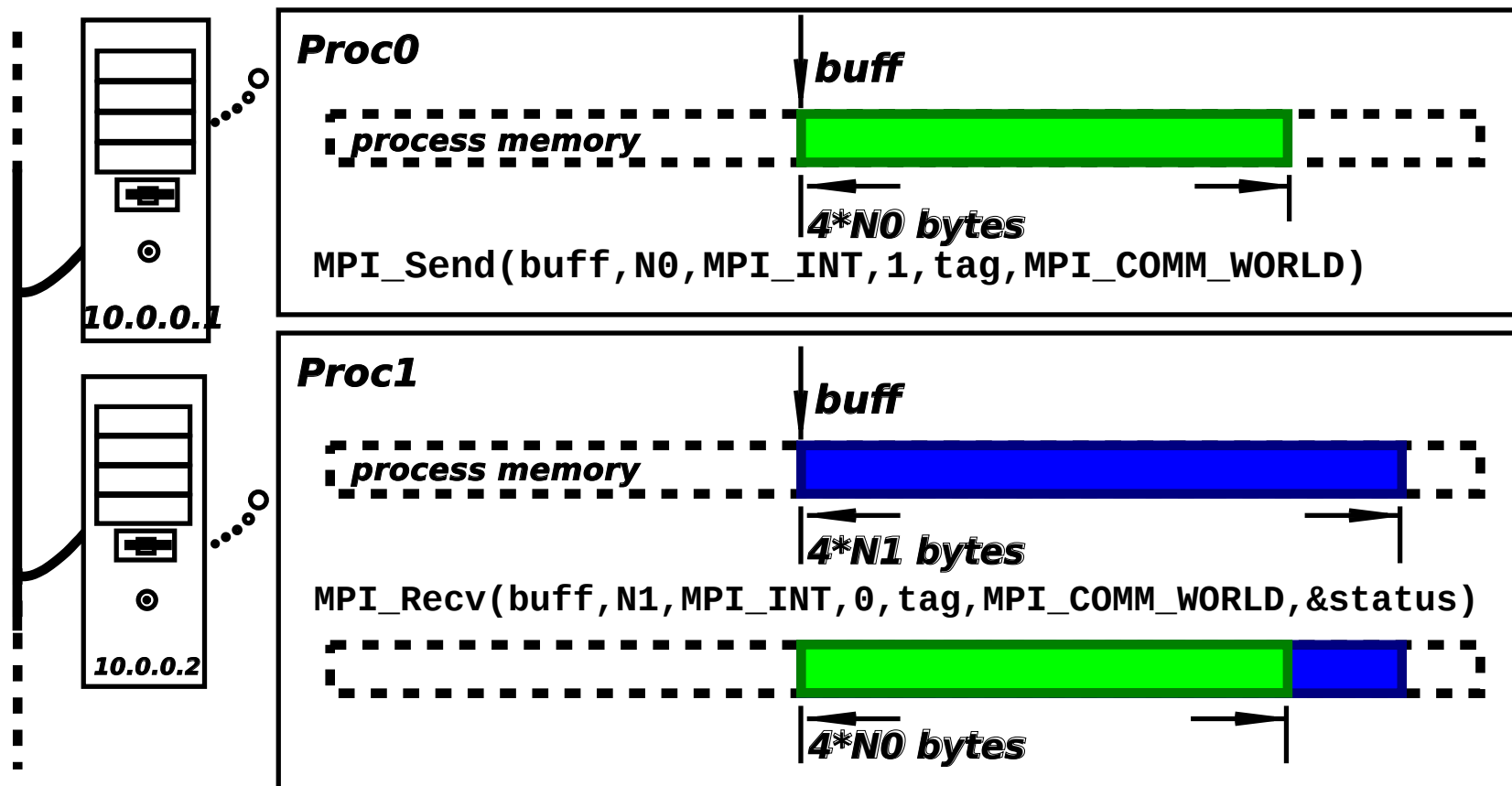
- **Fortran (note extra parameter):**

call MPI_RECV(result, 1, MPI_REAL, MPI_ANY_SOURCE, mtag1, MPI_COMM_WORLD, status, ierr)

Receive a message (cont.)

- *(address, length)* reception buffer
- *type* standard MPI type:
C: *MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR*
Fortran: *MPI_REAL, MPI_DOUBLE_PRECISION, MPI_INTEGER, MPI_CHARACTER*
- *(source, tag, communicator)*: selects message
- *status* Allows inspection of the data effectively received (e.g. length)

Receive a message (cont.)



OK if $N1 \geq N0$

send/receive parameters (cont.)

- *tag* message identifier
- *communicator* Process group, for instance *MPI_COMM_WORLD*
- *status* source, tag, and length of the received message
- Wildcards: *MPI_ANY_SOURCE*, *MPI_ANY_TAG*

Status

status (source, tag, length)

- a structure in C

```
1 MPI_Status status;  
2 . . .  
3 MPI_Recv(. . ., MPI_ANY_SOURCE, . . ., &status);  
4 source = status.MPI_SOURCE;  
5 printf("I got %f from process %d", result, source);
```

- an integer array in Fortran

```
1      integer status(MPI_STATUS_SIZE)  
2  C . . .  
3      call MPI_RECV(result, 1, MPI_REAL, MPI_ANY_SOURCE,  
4                      mtag1, MPI_COMM_WORLD, status, ierr)  
5      source = status(MPI_SOURCE)  
6      print *, 'I got ', result, ' from ', source
```


Point-to-point communication

Each *send* must be balanced by a receive in the corresponding node *recv*

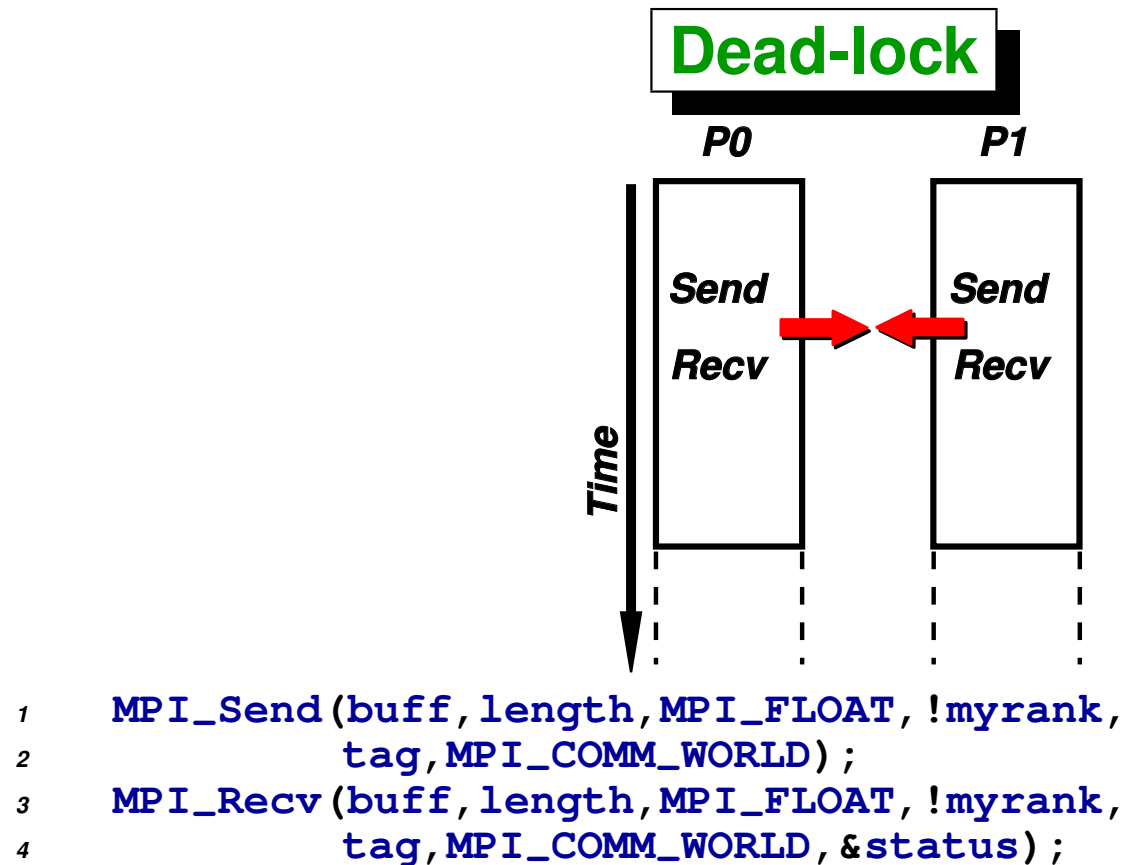
```
1 if (myid==0) {  
2     for(i=1; i<numprocs; i++)  
3         MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE,  
4                 mtag1, MPI_COMM_WORLD, &status);  
5 } else  
6     MPI_Send(&sum, 1, MPI_FLOAT, 0, mtag1, MPI_COMM_WORLD);
```

When a message is received?

When a posted *receive* matches the “*envelope*” of the message:

envelope = source/destination, tag, communicator

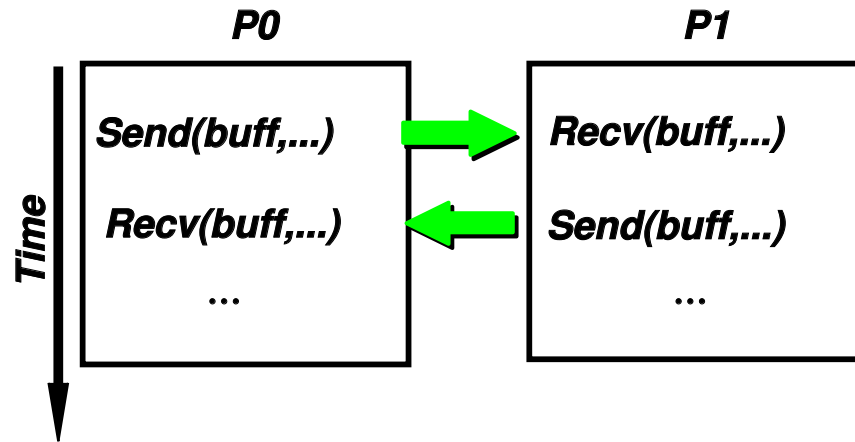
- $\text{size}(\text{receive buffer}) < \text{size}(\text{data sent}) \rightarrow \text{error}$
- $\text{size}(\text{receive buffer}) \geq \text{size}(\text{data sent}) \rightarrow \text{OK}$
- type don't match $\rightarrow \text{error}$



!myrank: Common C language to represent the *other* process. ($1 \rightarrow 0$, $0 \rightarrow 1$). Also $1 - \text{myrank}$ or $(\text{myrank} ? 0 : 1)$

MPI_Send and **MPI_Recv** are **blocking**, This means that code execution doesn't advance until the sending/reception is completed.

Correct calling order



```
1  if (!myrank) {  
2      MPI_Send(buff, length, MPI_FLOAT, !myrank,  
3              tag, MPI_COMM_WORLD);  
4      MPI_Recv(buff, length, MPI_FLOAT, !myrank,  
5              tag, MPI_COMM_WORLD, &status);  
6  } else {  
7      MPI_Recv(buff, length, MPI_FLOAT, !myrank,  
8              tag, MPI_COMM_WORLD, &status);  
9      MPI_Send(buff, length, MPI_FLOAT, !myrank,  
10             tag, MPI_COMM_WORLD);  
11 }
```

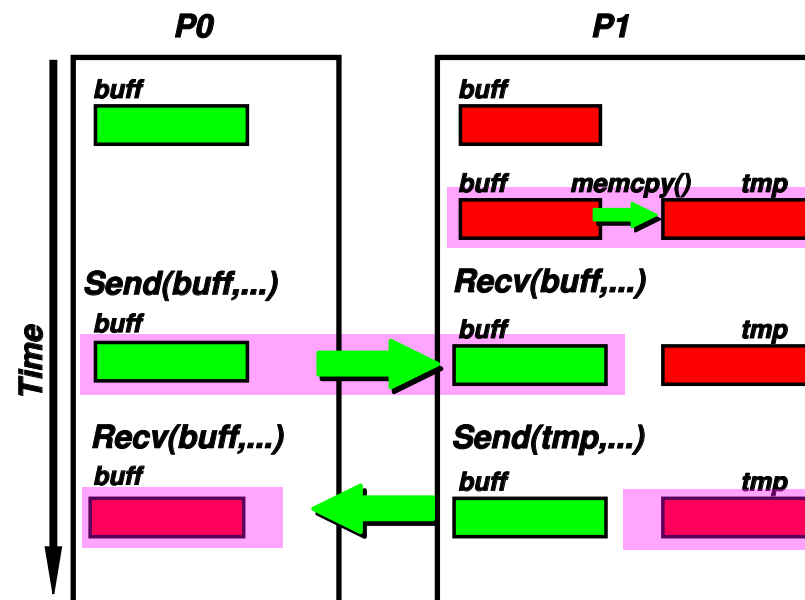
Correct calling order (cont.)

The previous code erroneously overwrites the reception buffer. We need a temporal buffer

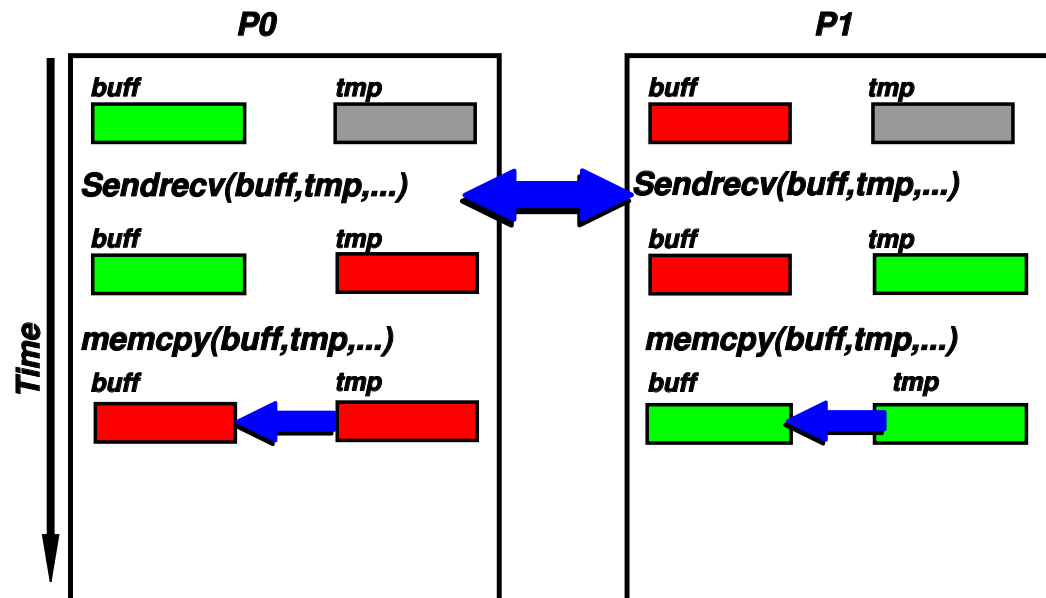
tmp:

```

1  if (!myrank) {
2      MPI_Send(buff, length, MPI_FLOAT,
3              !myrank, tag, MPI_COMM_WORLD);
4      MPI_Recv(buff, length, MPI_FLOAT,
5              !myrank, tag, MPI_COMM_WORLD,
6              &status);
7  } else {
8      float *tmp = new float[length];
9      memcpy(tmp, buff,
10             length*sizeof(float));
11     MPI_Recv(buff, length, MPI_FLOAT,
12             !myrank, tag, MPI_COMM_WORLD,
13             &status);
14     MPI_Send(tmp, length, MPI_FLOAT,
15             !myrank, tag, MPI_COMM_WORLD);
16     delete[] tmp;
17 }
    
```



Correct calling order (cont.)

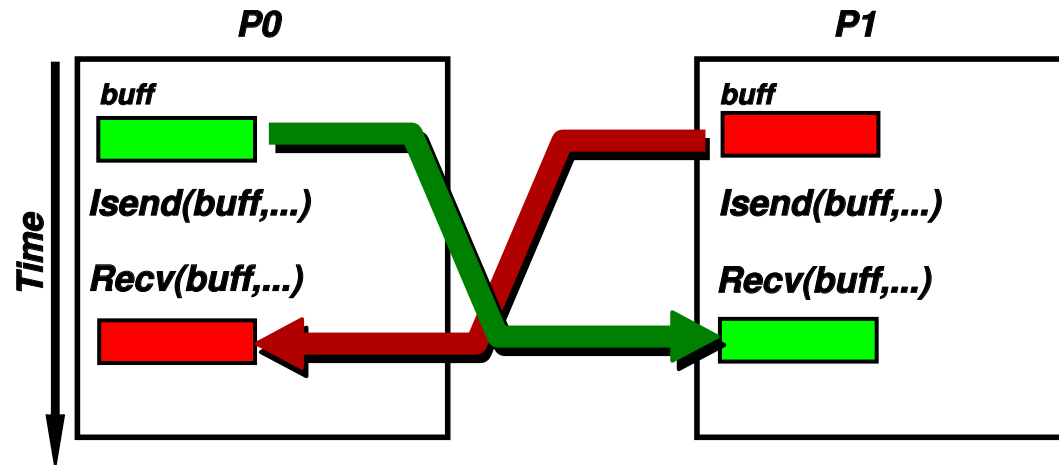


- ***MPI_Sendrecv***: sends and receives at the same time

```

1  float *tmp = new float[length];
2  int MPI_Sendrecv(buff, length, MPI_FLOAT, !myrank, stag,
3                  tmp, length, MPI_FLOAT, !myrank, rtag,
4                  MPI_COMM_WORLD, &status);
5  memcpy(tmp, buff, length*sizeof(float));
6  delete[] tmp;
    
```

Correct calling order (cont.)



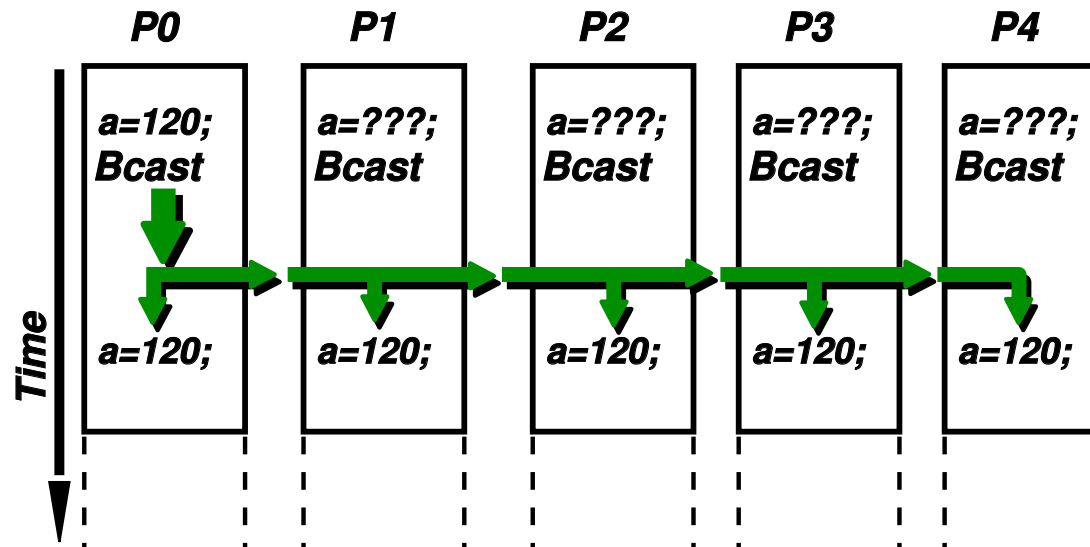
Use non-blocking send/receive

```
1 MPI_Request request;  
2 MPI_Isend(. . . ., request);  
3 MPI_Recv(. . . .);  
4 while(1) {  
5     /* do something */  
6     MPI_Test(request, flag, status);  
7     if(flag) break;  
8 }
```

- The code is the same for the two processes.
- Needs an auxiliary buffer (not shown here)

Global communication

Message broadcast



- Template:

`MPI_Bcast(address, length, type, source, comm)`

- C:

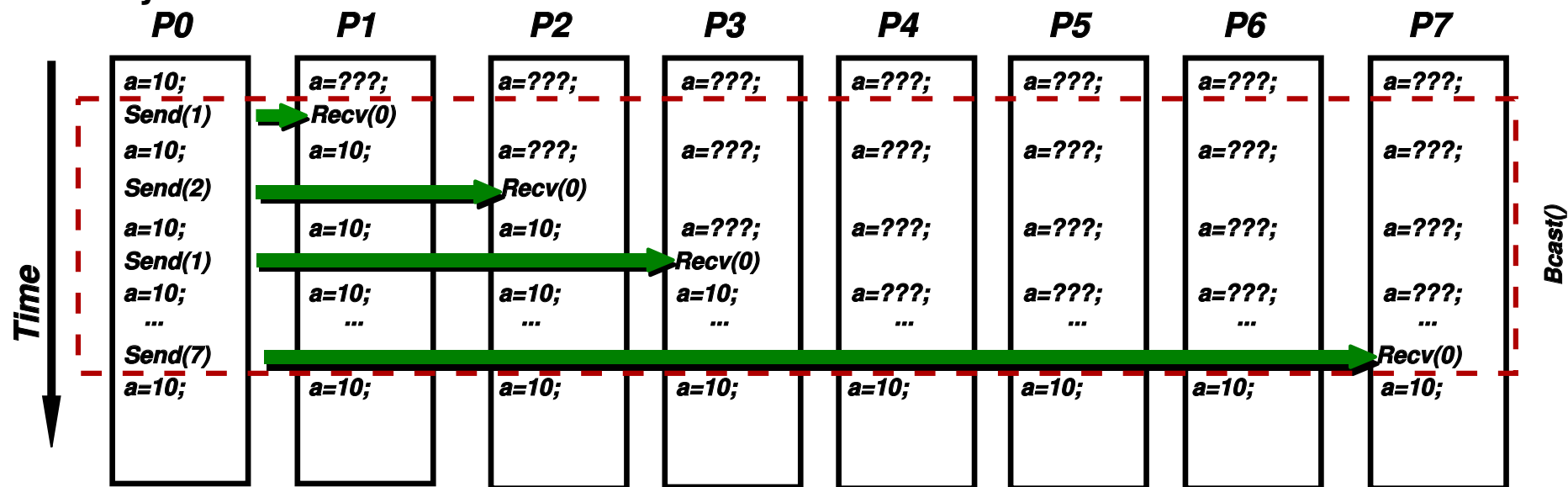
`ierr = MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);`

- Fortran:

`call MPI_Bcast(a, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)`

Message broadcast (cont.)

MPI_Bcast() is conceptually equivalent to a series of Sends/Receives, but it may be much more efficient.

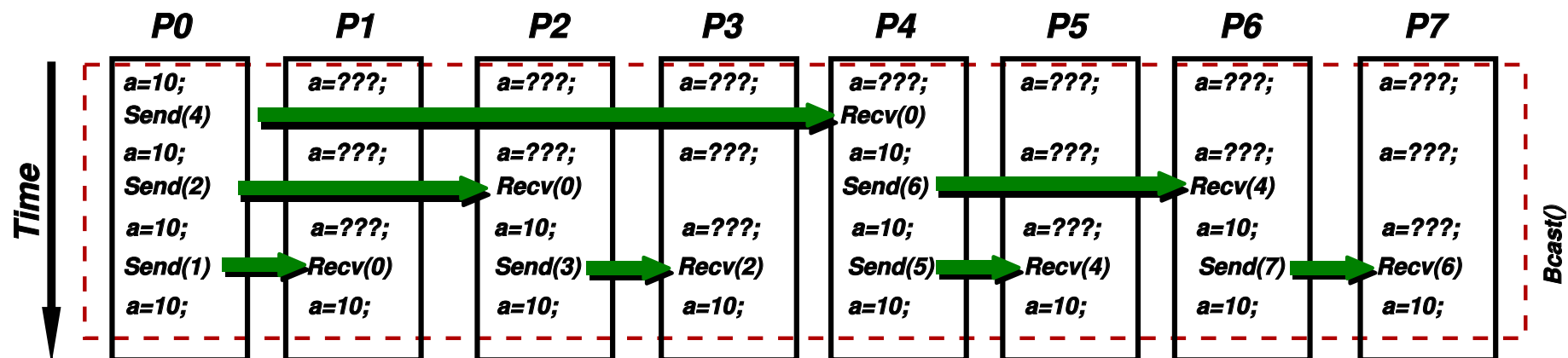


```

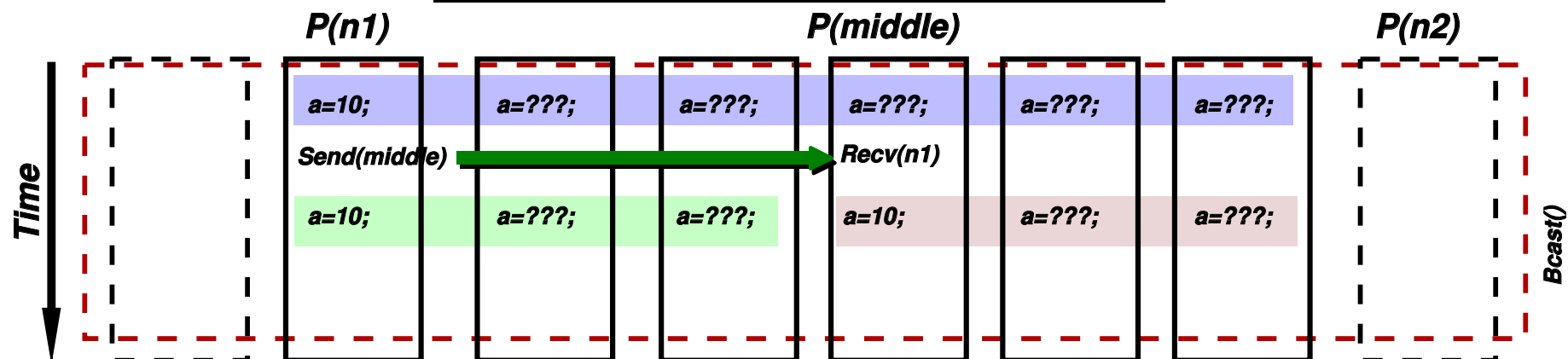
1  if (!myrank) {
2    for (int j=1; j<numprocs; j++) MPI_Send(buff, . . . , j);
3  } else {
4    MPI_Recv(buff, . . . , 0 . . . );
5  }

```

Message broadcast (cont.)



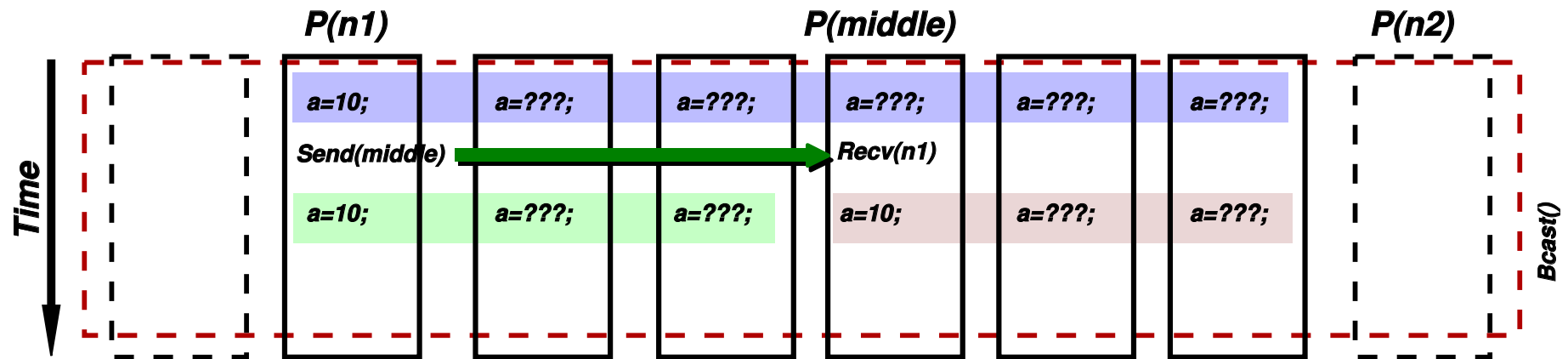
Message broadcast (cont.)



Efficient implementation of *MPI_Bcast()* with Send/Receives.

- At every moment we are in process *myrank* and we have an interval $[n1, n2)$ such that *myrank* is in $[n1, n2)$. Remember that $[n1, n2) = \{j \text{ such that } n1 \leq j < n2\}$
- Initially $n1=0$, $n2=NP$ (number of processors).
- In each step $n1$ sends to $middle = (n1+n2)/2$ and this will receive.
- In the next step we update the range to $[n1, middle)$ if $myrank < middle$ or else $[middle, n2)$.
- The process ends when $n2 - n1 == 1$

Message broadcast (cont.)



Pseudocode:

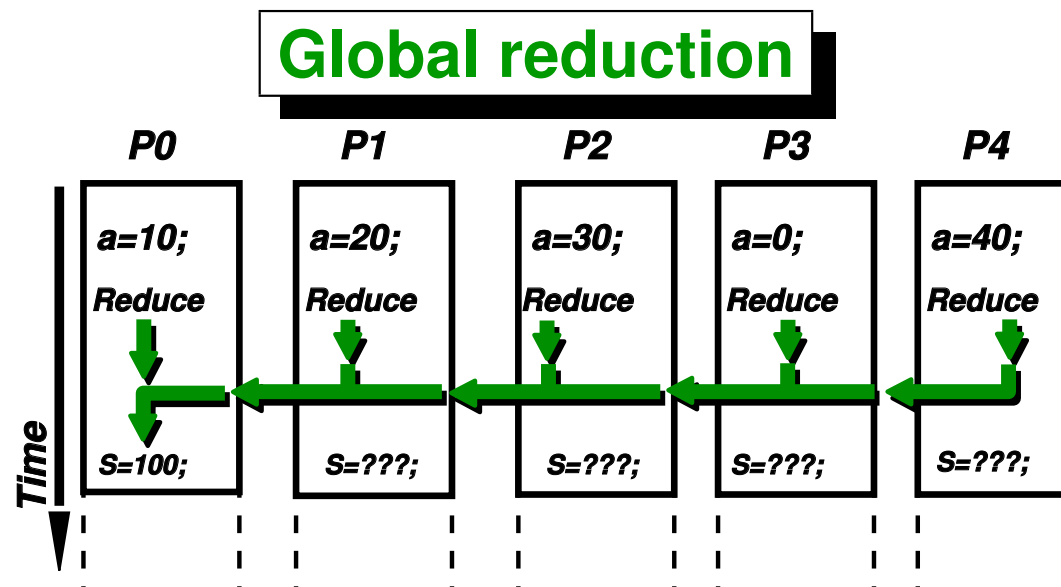
```

1  int n1=0, n2=numprocs;
2  while (1) {
3      int middle = (n1+n2)/2;
4      if (myrank==n1) MPI_Send(buff, . . . ,middle, . . . );
5      else if (myrank==middle) MPI_Recv(buff, . . . ,n1, . . . );
6      if (myrank<middle) n2 = middle;
7      else n1=middle;
8  }

```

Collective calls

These routines are *collective* (in contrast to the *point-to-point* `MPI_Send()` and `MPI_Recv()`). All processors in the communicator must call the function, and normally the collective call imposes an *implicit barrier* in the code execution.



- Template:
MPI_Reduce(s_address, r_address, length, type, operation, destination, comm)
- C:
ierr = MPI_Reduce(&s, &a, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
- Fortran:
call MPI_REDUCE(s, a, 0, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, ierr)

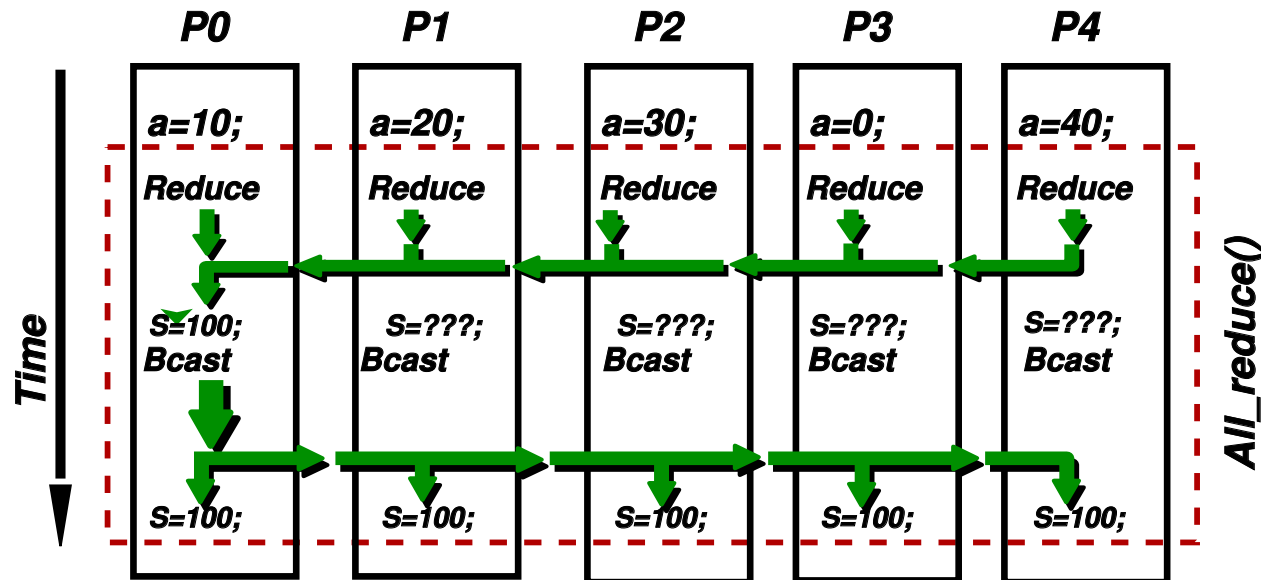
MPI associative global operations

Reduction functions apply a *binary associative operation* to a set of values. Typically,

- *MPI_SUM* sum
- *MPI_MAX* maximum
- *MPI_MIN* minimum
- *MPI_PROD* product
- *MPI_AND* boolean
- *MPI_OR* boolean

It is not specified the order in which the binary operations are done, so that it is very important that the function must be *associative*.

MPI associative global operations (cont.)



If the result of the reduction is needed in all processors, then we must use

*`MPI_Allreduce(s_address, r_address, length, type
operation, comm)`*

This is conceptually equivalent to a *`MPI_Reduce()`* followed by a *`MPI_Bcast()`*. **Warning: `MPI_Bcast()` and `MPI_Reduce()` are collective functions. All processors must call them!!**

Other MPI functions

- **Timers**
- **Gather and scatter operations**
- **MPE library. Included in MPICH but it is NOT part of the MPI standard.**
Calling grammar consistent with MPI. Usage: log-files, timing, graphs...

MPI in Unix environment

- MPICH installs normally in */usr/local/mpi* (*MPI_HOME*).
- Compiling: *> g++ -I/usr/local/mpi/include -o foo.o foo.cpp*
- Linking: *> g++ -L/usr/local/mpi/lib -lmpich -o foo foo.o*
- MPICH provides scripts *mpicc*, *mpif77* etc... that add the appropriate *-I*, *-L* and libraries.

MPI in Unix environment (cont.)

- The *mpirun* script launches the program in the specified hosts/nodes. Its options are:

- ▷ *-np <nbr-of-processors>*
- ▷ *-nolocal*
- ▷ *-machinefile machi.dat*

- Example:

```
1 [mstorti@node1]$ cat ./machi.dat
2 node2
3 node3
4 node4
5 [mstorti@node1]$ mpirun -np 4 \
6                   -machinefile ./machi.dat foo
```

Launches *foo* in *node1*, *node2*, *node3* and *node4*.

MPI in Unix environment (cont.)

- In order to NOT launch in the server use the *-nolocal* option.
- Example:

```
1 [mstorti@node1]$ cat ./machi.dat
2 node2
3 node3
4 node4
5 [mstorti@node1]$ mpirun -np 3 -nolocal \
6                   -machinefile ./machi.dat foo
```

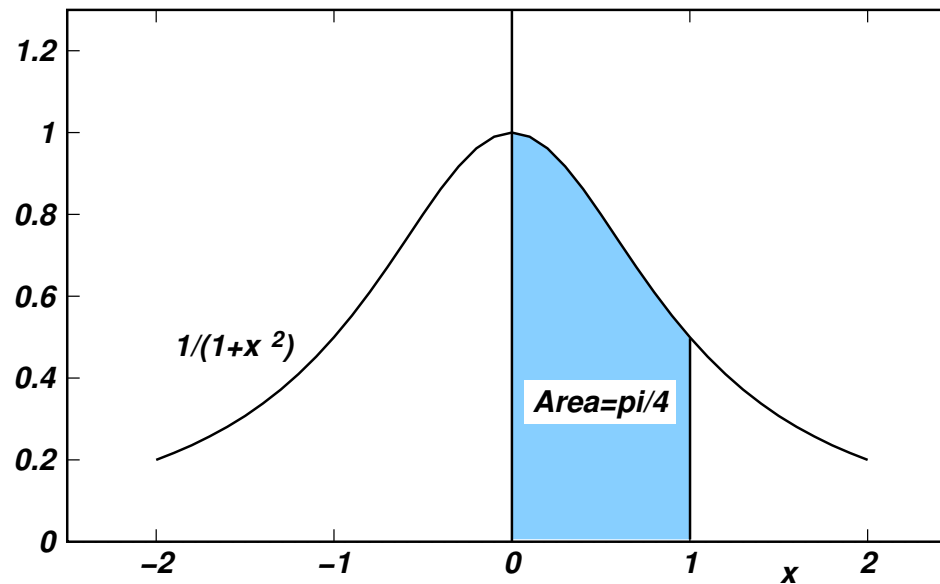
Launches *foo* in *node2*, *node3* and *node4*.

OPTIONAL Assignment Nbr. 2

Write a *mybcast* (...) function with the same signature as *MPI_Bcast* (...) using send/receive, first in sequential pattern, then in tree-like pattern, as explained above. Compare times as a number of the number of processors.

Example: Computing Pi

Example: Computing Pi by numerical integration

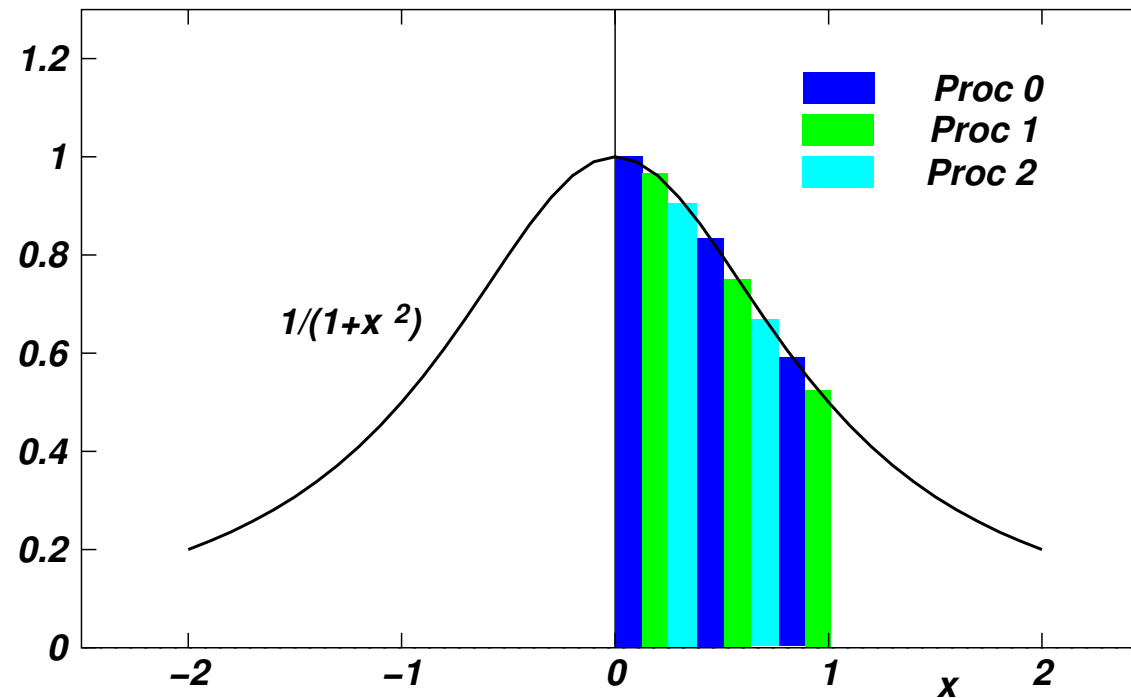


$$\text{atan}(1) = \pi/4$$

$$\frac{d \text{atan}(x)}{dx} = \frac{1}{1+x^2}$$

$$\pi/4 = \text{atan}(1) - \text{atan}(0) = \int_0^1 \frac{1}{1+x^2} dx$$

Numerical integration



Using the midpoint rule

- ***numprocs*** = number of processors
- n = Number of intervals (may be a multiple of ***numprocs*** or not)
- $h = 1/n$ = interval width

Numerical integration (cont.)

```
1 // Inicialization (rank,size) ...
2 while (1) {
3     // Master (rank==0) read number of intervals 'n' ...
4     // Broadcast 'n' to computing nodes ...
5     if (n==0) break;
6     // Compute 'mypi' (local contribution to 'pi') ...
7     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
8               MPI_SUM, 0, MPI_COMM_WORLD);
9     // Master reports error between computed pi and exact
10 }
11 MPI_Finalize();
```

```

1 // $Id: pi3.cpp,v 1.1 2004/07/22 17:44:50 mstorti Exp $
2
3 //*****
4 // pi3.cpp - compute pi by integrating  $f(x) = 4/(1 + x^2)$ 
5 //
6 // Each node:
7 // 1) receives the number of rectangles used
8 // in the approximation.
9 // 2) calculates the areas of it's rectangles.
10 // 3) Synchronizes for a global summation.
11 // Node 0 prints the result.
12 //
13 // Variables:
14 //
15 // pi      the calculated result
16 // n      number of points of integration.
17 // x      midpoint of each rectangle's interval
18 // f      function to integrate
19 // sum,pi  area of rectangles
20 // tmp     temporary scratch space for global summation
21 // i      do loop index
22 //*****
23
24 #include <mpi.h>
25 #include <stdio>
26 #include <cmath>
27
28 // The function to integrate
29 double f(double x) { return 4./(1.+x*x); }
30
31 int main(int argc, char **argv) {
32
33 // Initialize MPI environment

```

```
34 MPI_Init (&argc, &argv) ;
35
36 // Get the process number and assign it to the variable myrank
37 int myrank;
38 MPI_Comm_rank (MPI_COMM_WORLD, &myrank) ;
39
40 // Determine how many processes the program will run on and
41 // assign that number to size
42 int size;
43 MPI_Comm_size (MPI_COMM_WORLD, &size) ;
44
45 // The exact value
46 double PI=4*atan(1.0);
47
48 // Enter an infinite loop. Will exit when user enters n=0
49 while (1) {
50     int n;
51     // Test to see if this is the program running on process 0,
52     // and run this section of the code for input.
53     if (!myrank) {
54         printf("Enter the number of intervals: (0 quits) > ");
55         scanf ("%d", &n);
56     }
57
58     // The argument 0 in the 4th place indicates that
59     // process 0 will send then single integer n to every
60     // other process in processor group MPI_COMM_WORLD.
61     MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
62
63     // If the user puts in a negative number for n we leave
64     // the program by branching to MPI_FINALIZE
65     if (n<0) break;
66
```

```

67 // Now this part of the code is running on every node
68 // and each one shares the same value of n. But all
69 // other variables are local to each individual
70 // process. So each process then calculates the each
71 // interval size.
72
73 //*****C
74 //      Main Body : Runs on all processors
75 //*****C
76 // even step size h as a function of partions
77 double h = 1.0/double(n);
78 double sum = 0.0;
79 for (int i=myrank+1; i<=n; i += size) {
80     double x = h * (double(i) - 0.5);
81     sum = sum + f(x);
82 }
83 double pi, mypi = h * sum; // this is the total area
84                             // in this process,
85                             // (a partial sum.)
86
87 // Each individual sum should converge also to PI,
88 // compute the max error
89 double error, my_error = fabs(size*mypi-PI);
90 MPI_Reduce(&my_error,&error,1,MPI_DOUBLE,
91           MPI_MAX,0,MPI_COMM_WORLD);
92
93 // After each partition of the integral is calculated
94 // we collect all the partial sums. The MPI_SUM
95 // argument is the operation that adds all the values
96 // of mypi into pi of process 0 indicated by the 6th
97 // argument.
98 MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

```

```
99
100  //*****
101  //          Print results from Process 0
102  //*****
103
104  // Finally the program tests if myrank is node 0
105  // so process 0 can print the answer.
106  if (!myrank) printf("pi is aprox: %f, "
107                    "(error %f,max err over procs %f)\n",
108                    pi,fabs(pi-PI),my_error);
109  // Run the program again.
110  //
111  }
112  // Branch for the end of program. MPI_FINALIZE will close
113  // all the processes in the active group.
114
115  MPI_Finalize();
116 }
```

Numerical integration (cont.)

ps report:

```
1 [mstorti@spider example]$ ps axfw
2  PID COMMAND
3  701 xterm -e bash
4  707  \_ bash
5  907  \_ emacs
6  908  |    \_ /usr/libexec/emacs/21.2/i686-pc-linux-gnu/ema...
7  985  |    \_ /bin/bash
8 1732  |    |    \_ ps axfw
9 1037  |    \_ /bin/bash -i
10 1058  |    \_ xpdf slides.pdf
11 1059  |    \_ xfig -library_dir /home/mstorti/CONFIG/xf...
12 1641  |    \_ /bin/sh /usr/local/mpi/bin/mpirun -np 2 pi3.bin
13 1718  |    \_ /.../pi3.bin -p4pg /home/mstorti/
14 1719  |    \_ /.../pi3.bin -p4pg /home/msto....
15 1720  |    \_ /usr/bin/rsh localhost.localdomain ...
16 1537  \_ top
17 [mstorti@spider example]$
```

Basic scalability concepts

Let T_1 be the computing time for only one processor (assume that all processors are equal), and let T_n be the computing time in n processors. The **gain factor** or **speed-up** due to the use of parallel computing is

$$S_n = \frac{T_1}{T_n}$$

In the best case the time is reduced by a factor n , i.e. $T_n > T_1/n$ so that

$$S_n = \frac{T_1}{T_n} < \frac{T_1}{T_1/n} = n = S_n^* = \text{max. theoretical speed-up}$$

The efficiency η is the relation between the obtained speed-up S_n and the theoretical one, so that

$$\eta = \frac{S_n}{S_n^*} < 1$$

Basic scalability concepts (cont.)

Suppose we have a certain amount of elemental tasks W to accomplish. In the example of computing π it would be the number of rectangles to add up. If the computing requirements are equal for all tasks and if we distribute the W tasks equally to all processors, then each processor will have assigned $W_i = W/n$ tasks, and will finish in time $T_i = W_i/s$ where s is the **“processing speed”** of the processors (assumed to be the same for all processors).

Basic scalability concepts (cont.)

If there is no communication, or it is negligible, then

$$T_n = \max_i T_i = \frac{W_i}{s} = \frac{1}{n} \frac{W}{s}$$

while with only one processor, the run will take

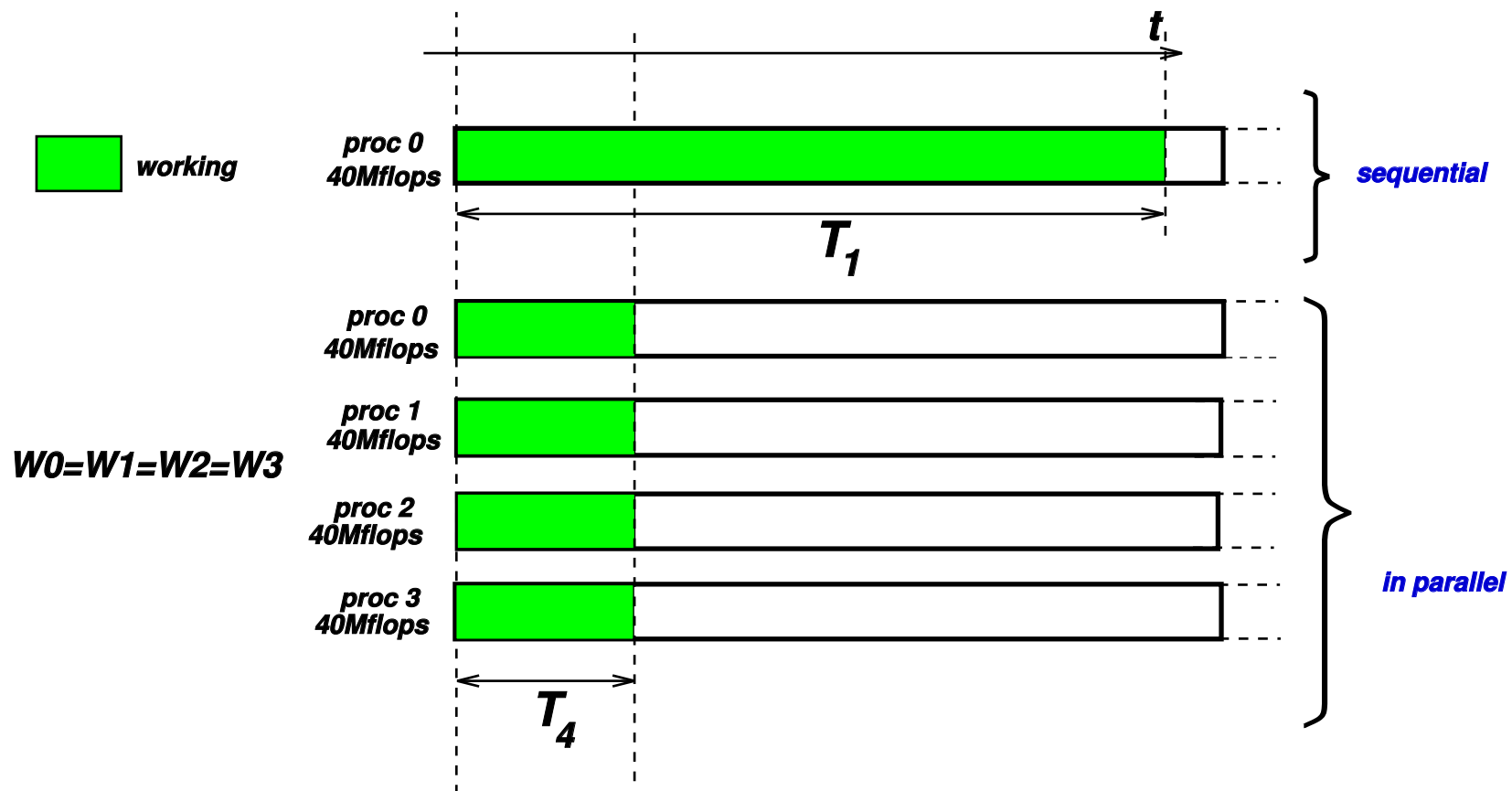
$$T_1 = \frac{W}{s}$$

So that

$$S_n = \frac{T_1}{T_n} = \frac{W/s}{W/sn} = n = S_n^*.$$

and the speed-up is equal to the theoretical one (efficiency $\eta = 1$).

Basic scalability concepts (cont.)



Basic scalability concepts (cont.)

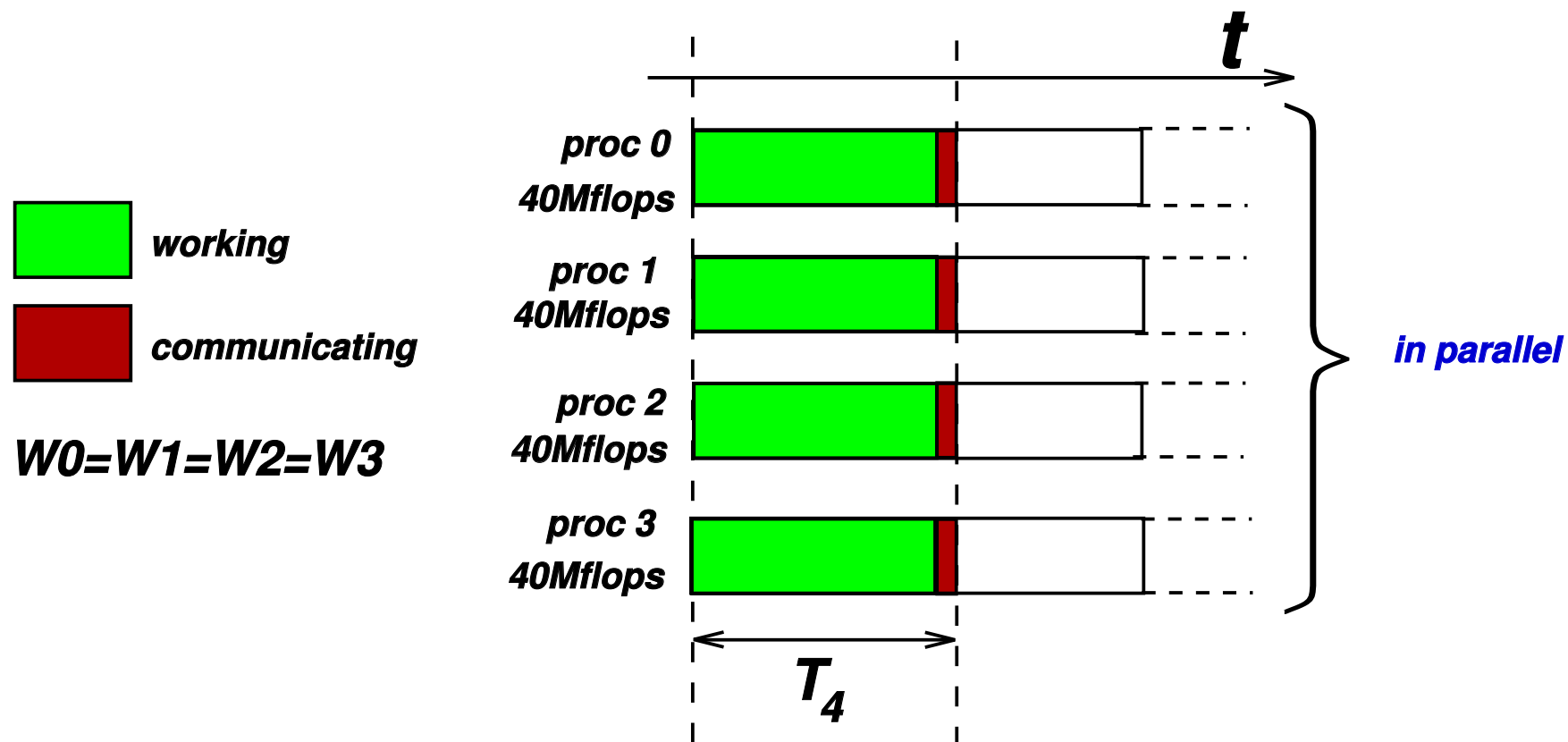
If the communication time *is not* negligible

$$T_n = \frac{W}{sn} + T_{\text{comm}}$$

$$S_n = \frac{T_1}{T_n} = \frac{W/s}{W/(sn) + T_{\text{comm}}}$$

$$\eta = \frac{W/s}{(W/s) + n T_{\text{comm}}} = \frac{\text{(total comp. time)}}{\text{(total comp. time)} + \text{(total comm. time)}}$$

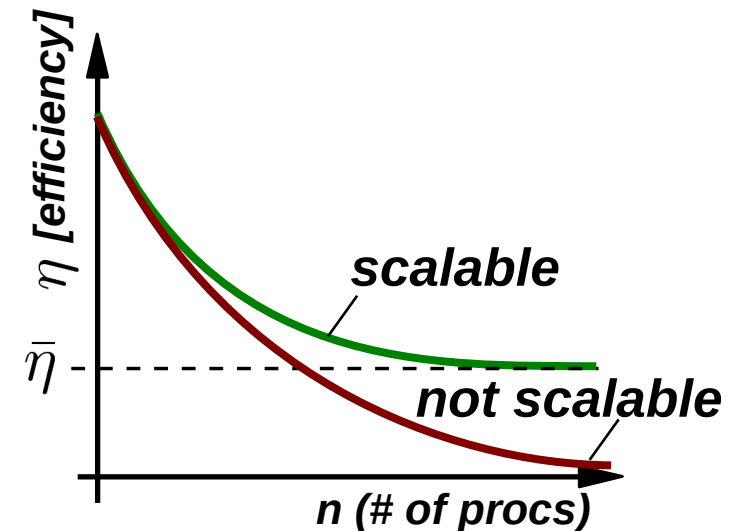
Basic scalability concepts (cont.)



Basic scalability concepts (cont.)

We say that a parallel implementation is **scalable** if we can keep the efficiency η above a certain threshold value $\bar{\eta}$ as we increase the number of processors.

$$\eta \geq \bar{\eta} > 0, \quad \text{para } n \rightarrow \infty$$



If we think at the example of computing π , then the total computing time is kept constant, but the total communication time grows with n because, even if we have to send only a double, the communication is at least a latency times the number of processors, so that posed in this way **this parallel implementation is not scalable**.

In general this will happen always. If we have a certain fixed amount of work W , and if we increase the number of processors, then the communication times will increase and no parallel implementation will be scalable.

Basic scalability concepts (cont.)

But we **can** keep efficiency bounded by below if we increase the size of the problem as the same time that we increase the number of processors. If W is the work to be done (e.g. the number of intervals in the π computation example), and if we let $W = n\bar{W}$, i.e. **we keep the number of intervals per processor constant**, then the total computing time also increases $\propto n$ and efficiency is kept bounded from below.

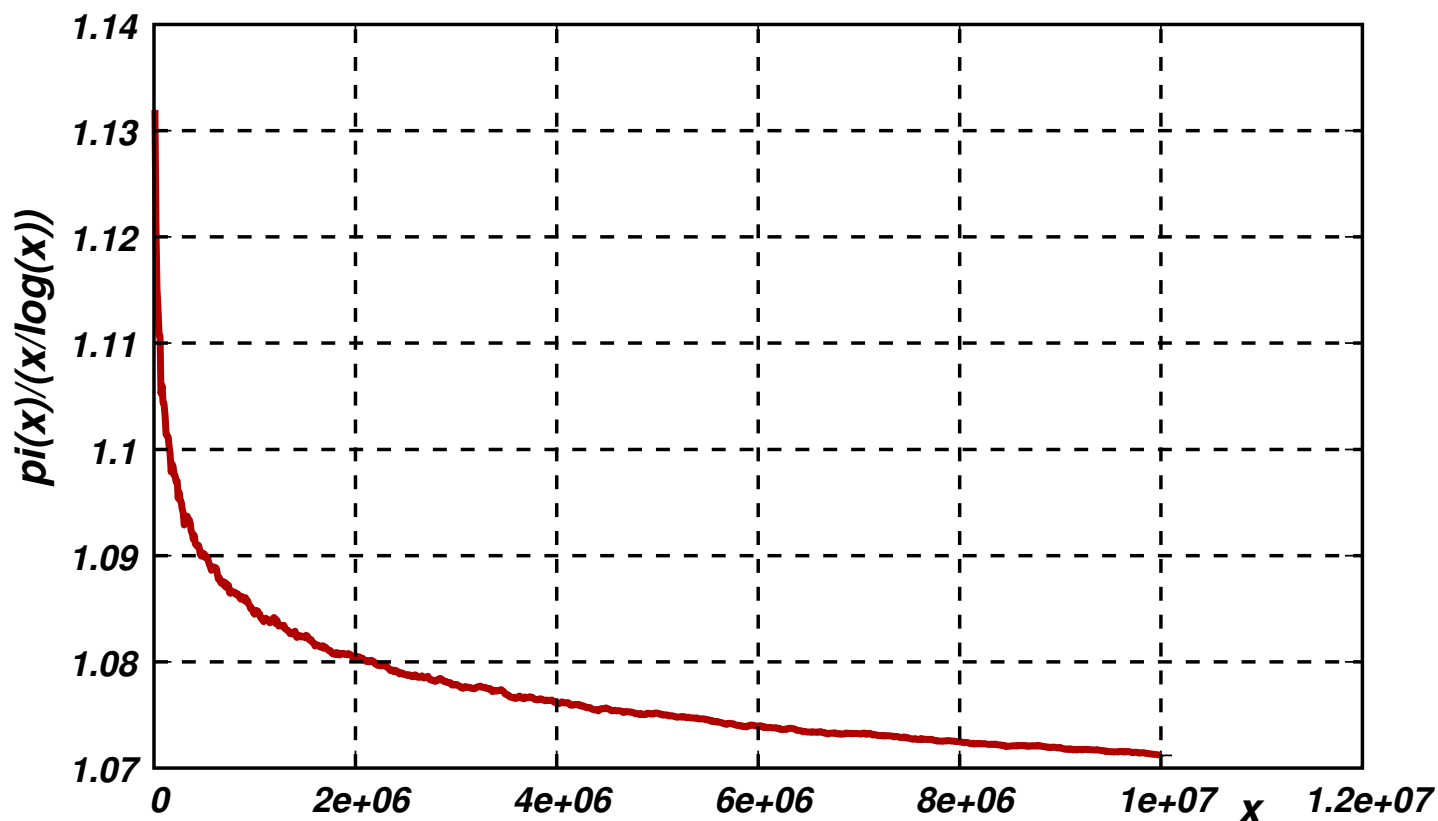
We say that such implementation is scalable in the **thermodynamic limit**, i.e. if we can keep efficiency bounded from below for $n \rightarrow \infty$ and $W \sim n \rightarrow \infty$. Basically this means that we can solve larger problems in a larger number of nodes in the same time.

Example: Prime Number Theorem

Prime number theorem

The PNT says that the number $\pi(x)$ of prime numbers smaller than x is, asymptotically

$$\pi(x) \approx \frac{x}{\log x}$$



Prime number theorem (cont.)

The most simple form to verify if a number n is prime it to divide it by all integers from 2 to $\text{floor}(\sqrt{n})$. If we can't find a divisor then the number is prime

```

1  int is_prime(int n) {
2      if (n<2) return 0;
3      int m = int(sqrt(n));
4      for (int j=2; j<=m; j++)
5          if (n % j==0) return 0;
6      return 1;
7  }

```

So that ***is_prime(n)*** is (in the worst case) $O(\sqrt{n})$. As the number of digits of n is $n_d = \text{ceil}(\log_{10} n)$, to test a number n for primality (with this algorithm) is $O(10^{n_d/2})$ i.e. is ***non polinomial in the number of digits n_d*** .

Computing $\pi(n)$ is then $\sim \sum_{n'=2}^n \sqrt{n'} \sim n^{1.5}$ (also non polinomial BTW). Its interesting as an exercise of parallel computing, since the cost of each individual computation (for each number n') is very variable and in average it grows with n .

PNT: Sequential version

```
1 // $Id: primes1.cpp, v 1.2 2005/04/29 02:35:28 mstorti Exp $
2 #include <stdio>
3 #include <cmath>
4
5 int is_prime(int n) {
6     if (n<2) return 0;
7     int m = int(sqrt(n));
8     for (int j=2; j<=m; j++)
9         if (n % j ==0) return 0;
10    return 1;
11 }
12
13 // Sequential version
14
15 int main(int argc, char **argv) {
16
17     int n=2, primes=0, chunk=10000;
18     while(1) {
19         if (is_prime(n++)) primes++;
20         if (!(n % chunk)) printf("%d primes<%d\n",primes,n);
21     }
22 }
```

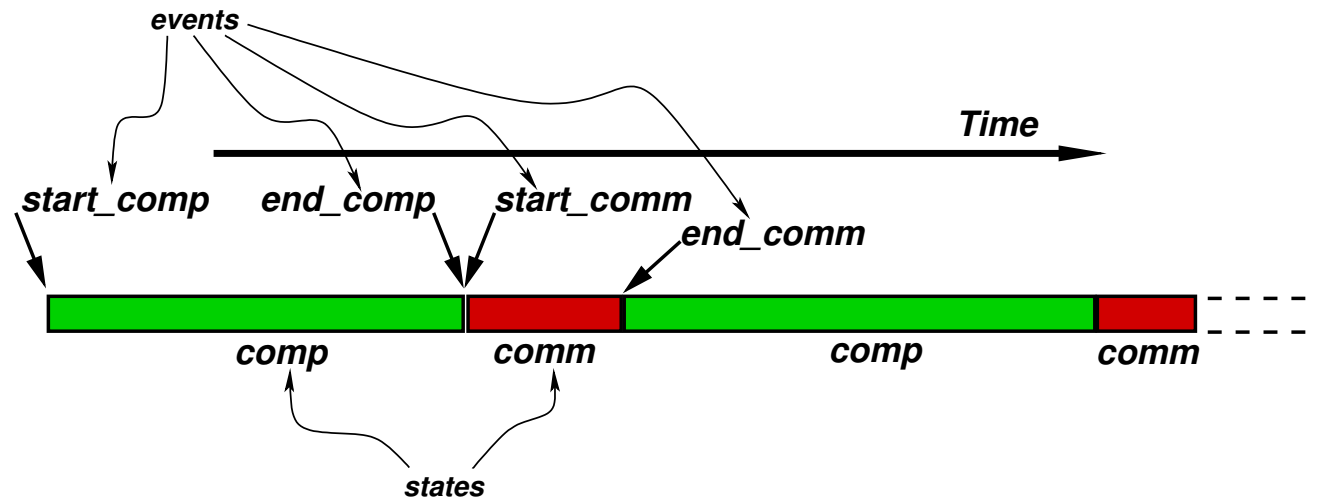
PNT: Parallel version

- Fixed length chunks.
- Each node process a chunk and final reduction with *MPI_Reduce()*.

```
1 // $Id: primes2.cpp, v 1.1 2004/07/23 01:33:27 mstorti Exp $
2 #include <mpi.h>
3 #include <stdio>
4 #include <cmath>
5
6 int is_prime(int n) {
7     int m = int(sqrt(n));
8     for (int j=2; j<=m; j++)
9         if (!(n % j)) return 0;
10    return 1;
11 }
12
13 int main(int argc, char **argv) {
14     MPI_Init(&argc, &argv);
15
16     int myrank, size;
17     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
18     MPI_Comm_size(MPI_COMM_WORLD, &size);
19
20     int n2, primesh=0, primes, chunk=100000,
21         n1 = myrank*chunk;
22     while(1) {
23         n2 = n1 + chunk;
24         for (int n=n1; n<n2; n++) {
```

```
25     if (is_prime(n)) primesh++;
26 }
27 MPI_Reduce(&primesh, &primes, 1, MPI_INT,
28           MPI_SUM, 0, MPI_COMM_WORLD);
29 n1 += size*chunk;
30 if (!myrank) printf("pi(%d) = %d\n", n1, primes);
31 }
32
33 MPI_Finalize();
34 }
```

MPE Logging



- User can define **states**. Typically we want to know how much time the program spends in **computation** (*comp*) and how much in **communication** (*comm*).
- States are delimited by **events** of very short duration (we say **atomic**). Typically we define two events for each state.
 - ▷ state *comp* = $\{t \mid \text{start_comp} < t < \text{end_comp}\}$
 - ▷ state *comm* = $\{t \mid \text{start_comm} < t < \text{end_comm}\}$

MPE Logging (cont.)

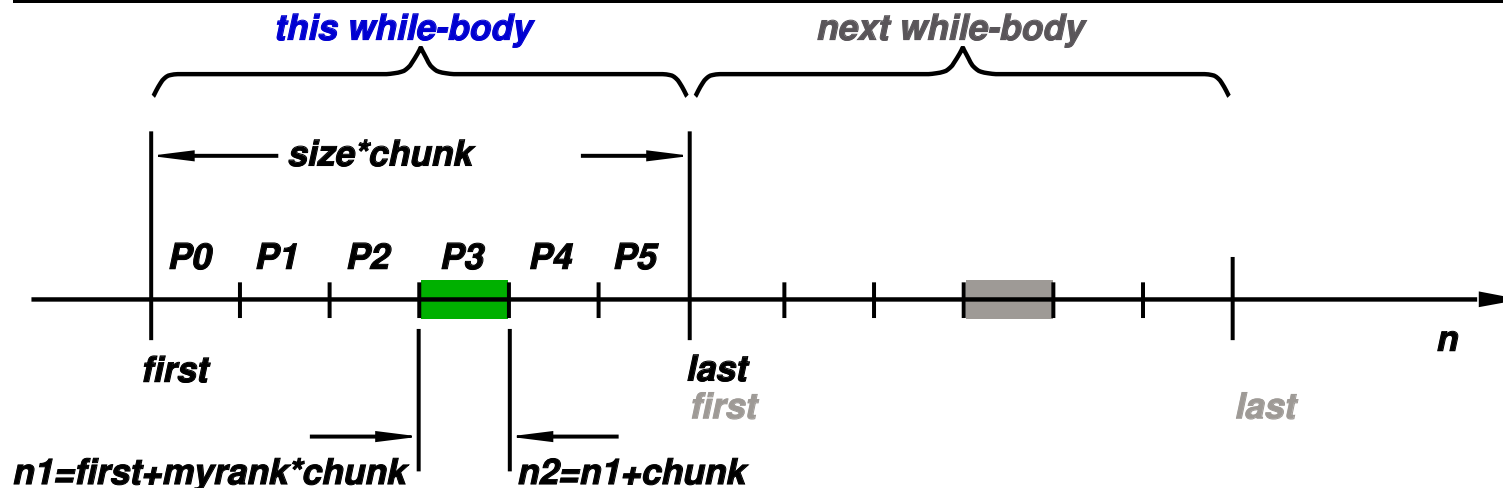
```

1  #include <mpe.h>
2  #include ...
3
4  int main(int argc, char **argv) {
5      MPI_Init(&argc, &argv);
6      MPE_Init_log();
7      int start_comp = MPE_Log_get_event_number();
8      int end_comp = MPE_Log_get_event_number();
9      int start_comm = MPE_Log_get_event_number();
10     int end_comm = MPE_Log_get_event_number();
11
12     MPE_Describe_state(start_comp, end_comp, "comp", "green:gray");
13     MPE_Describe_state(start_comm, end_comm, "comm", "red:white");
14
15     while(...) {
16         MPE_Log_event(start_comp, 0, "start-comp");
17         // compute...
18         MPE_Log_event(end_comp, 0, "end-comp");
19
20         MPE_Log_event(start_comm, 0, "start-comm");
21         // communicate...
22         MPE_Log_event(end_comm, 0, "end-comm");
23     }
24
25     MPE_Finish_log("primes");
26     MPI_Finalize();
27 }

```

- In general it is difficult to separate *communication* from *synchronization*.

PNT: Parallel dynamic version with MPE logging



```

1 // Counts primes en [0,N)
2 first = 0;
3 while (1) {
4     // Each processor checks a subrange in [first,last)
5     last = first + size*chunk;
6     n1 = first + myrank*chunk;
7     n2 = n1 + chunk;
8     if (n2>last) n2=last;
9     primesh = /* # of primes in [n1,n2) ... */;
10    // Allreduce 'primesh' to 'primes' ...
11    first += size*chunk;
12    if (last>N) break;
13 }
14 // finalize ...

```

PNT: Parallel dynamic version with MPE logging (cont.)

```
1 // $Id: primes3.cpp, v 1.4 2004/07/23 22:51:31 mstorti Exp $
2 #include <mpi.h>
3 #include <mpe.h>
4 #include <stdio>
5 #include <cmath>
6
7 int is_prime(int n) {
8     if (n<2) return 0;
9     int m = int(sqrt(n));
10    for (int j=2; j<=m; j++)
11        if (!(n % j)) return 0;
12    return 1;
13 }
14
15 int main(int argc, char **argv) {
16     MPI_Init(&argc, &argv);
17     MPE_Init_log();
18
19     int myrank, size;
20     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
21     MPI_Comm_size(MPI_COMM_WORLD, &size);
22     int start_comp = MPE_Log_get_event_number();
23     int end_comp = MPE_Log_get_event_number();
24     int start_bcast = MPE_Log_get_event_number();
25     int end_bcast = MPE_Log_get_event_number();
26
27     int n2, primesh=0, primes, chunk=200000,
```

```

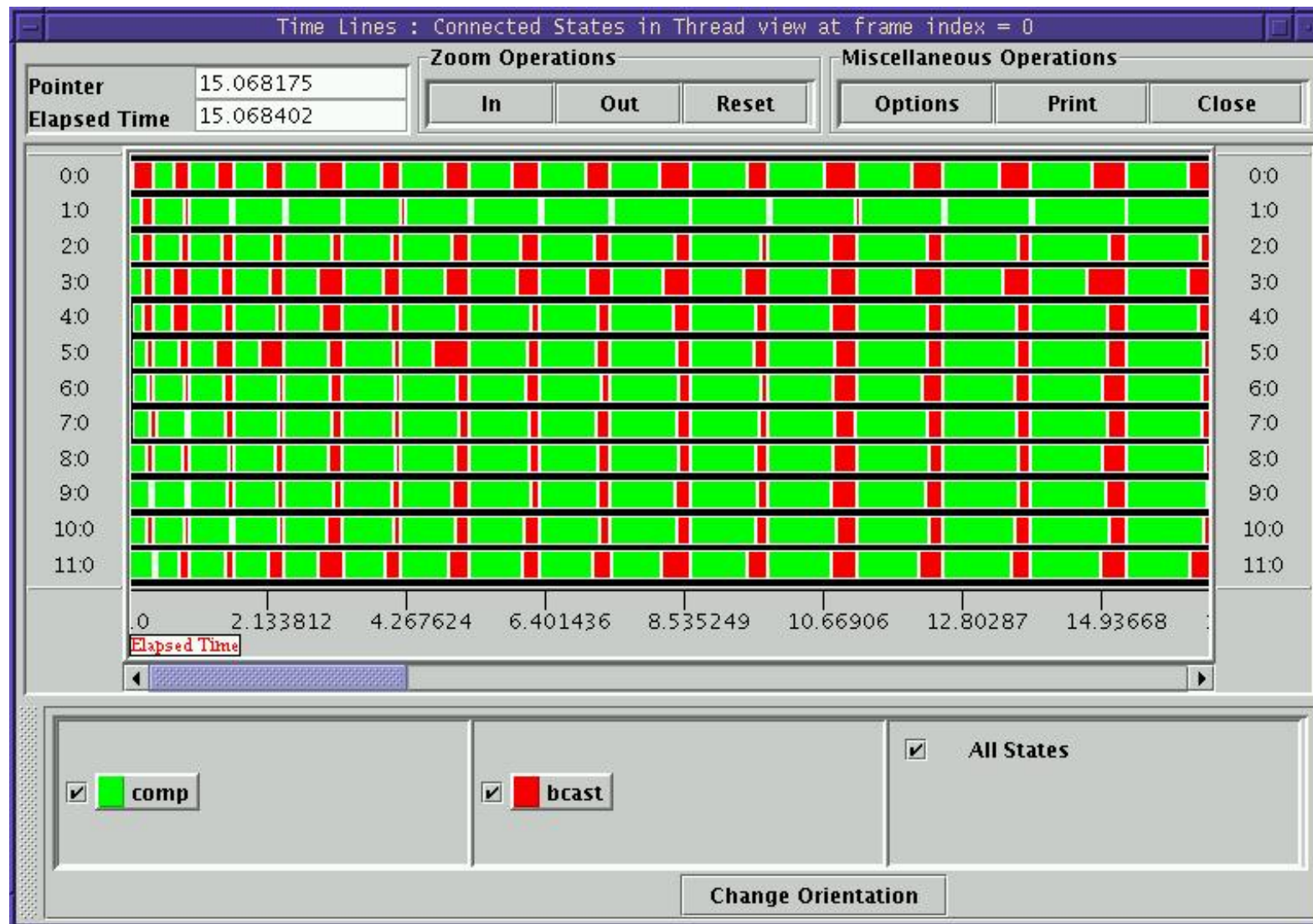
28     n1, first=0, last;
29
30     if (!myrank) {
31         MPE_Describe_state(start_comp, end_comp, "comp", "green:gray");
32         MPE_Describe_state(start_bcast, end_bcast, "bcast", "red:white");
33     }
34
35     while(1) {
36         MPE_Log_event(start_comp, 0, "start-comp");
37         last = first + size*chunk;
38         n1 = first + myrank*chunk;
39         n2 = n1 + chunk;
40         for (int n=n1; n<n2; n++) {
41             if (is_prime(n)) primesh++;
42         }
43         MPE_Log_event(end_comp, 0, "end-comp");
44         MPE_Log_event(start_bcast, 0, "start-bcast");
45         MPI_Allreduce(&primesh, &primes, 1, MPI_INT,
46                     MPI_SUM, MPI_COMM_WORLD);
47         first += size*chunk;
48         if (!myrank) printf("pi(%d) = %d\n", last, primes);
49         MPE_Log_event(end_bcast, 0, "end-bcast");
50         if (last>=10000000) break;
51     }
52
53     MPE_Finish_log("primes");
54     MPI_Finalize();
55 }

```

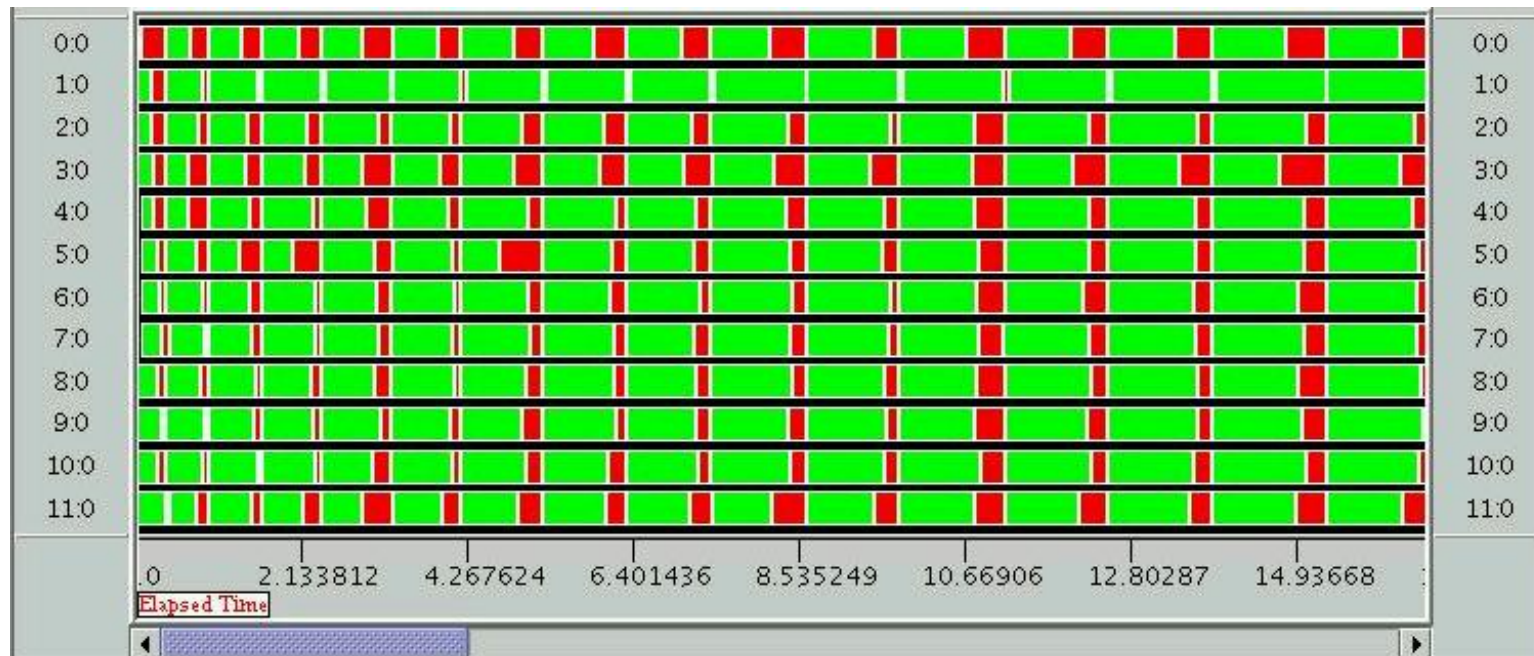
Using Jumpshot

- “*Jumpshot*” is a free utility that comes with MPICH and allows to inspect MPE logs in a graphical way.
- Configure MPI (version 1.2.5.2) with *--with-mpe*, and install some Java version, may be *j2sdk-1.4.2_04-fcs* from www.sun.org.
- After running the program a file *primes.clog* is created.
- Convert to *SLOG* format with *>> clog2slog primes.clog*
- Run *>> jumpshot primes.slog &*

Using Jumpshot (cont.)



PNT: $np = 12$ (blocking)



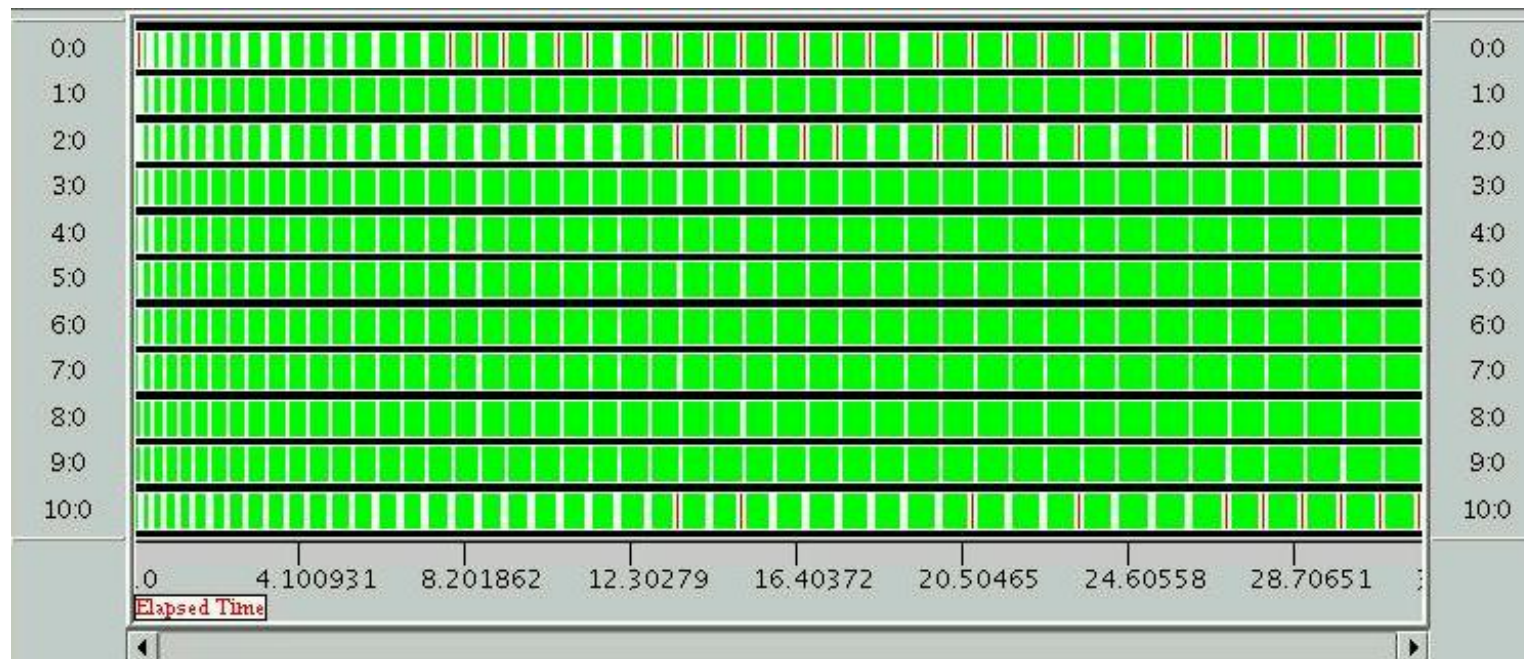
- Nodes are running other programs (simulates variable node speed)
- *comm* is in fact *communication+synchronization*.
- *MPI_Allreduce()* acts like a synchronization barrier (all processors wait until *all* of them reach the collective call).
- Processor 1 is very slow and disbalances the parallel run (the other processors must wait) resulting in a loss of efficiency of 20 to 30%.

PNT: np = 12 (blocking) (cont.)



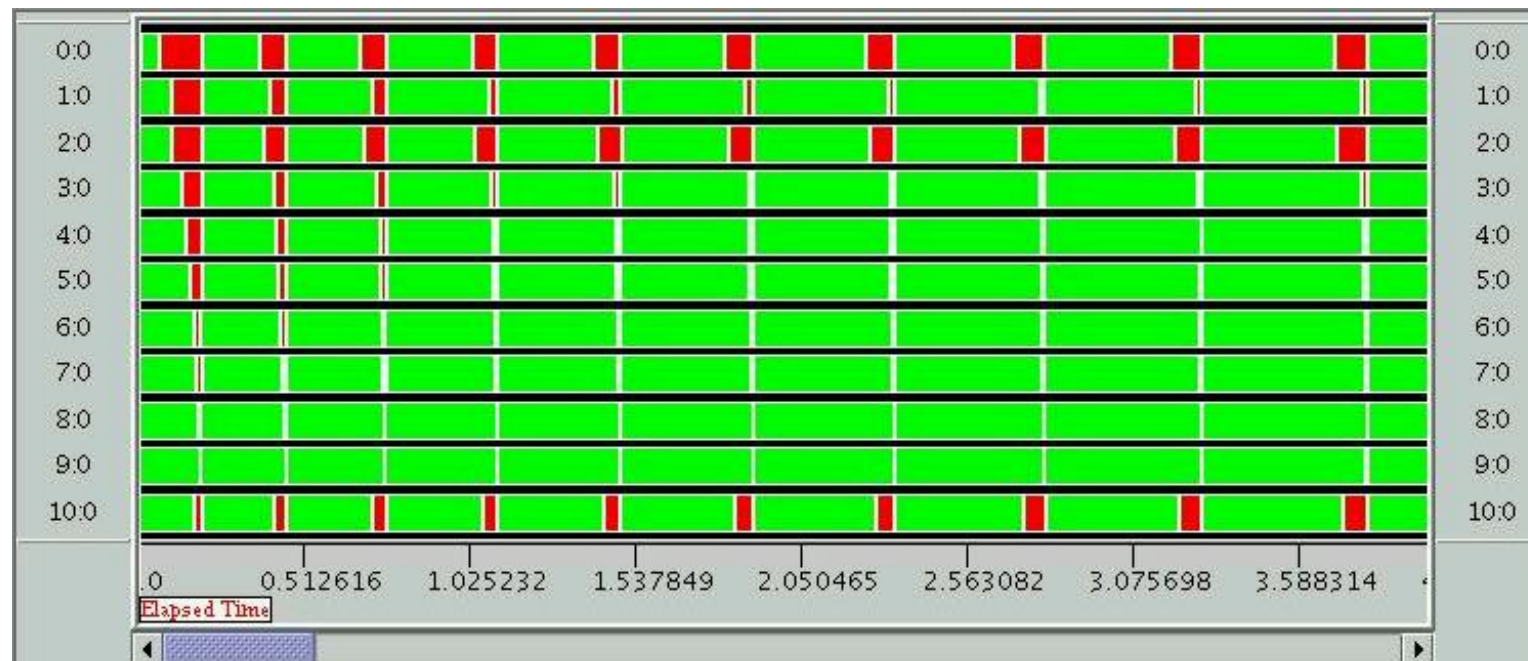
- Proc 1 is running other programs (simulates a slow processor). Other processors are dedicated.
- Disbalance is even worst.

PNT: $np = 12$ (blocking) (cont.)



- Without node12 (proc1 in the previous example). Note that good balance is achieved.

PNT: np = 12 (blocking) (cont.)



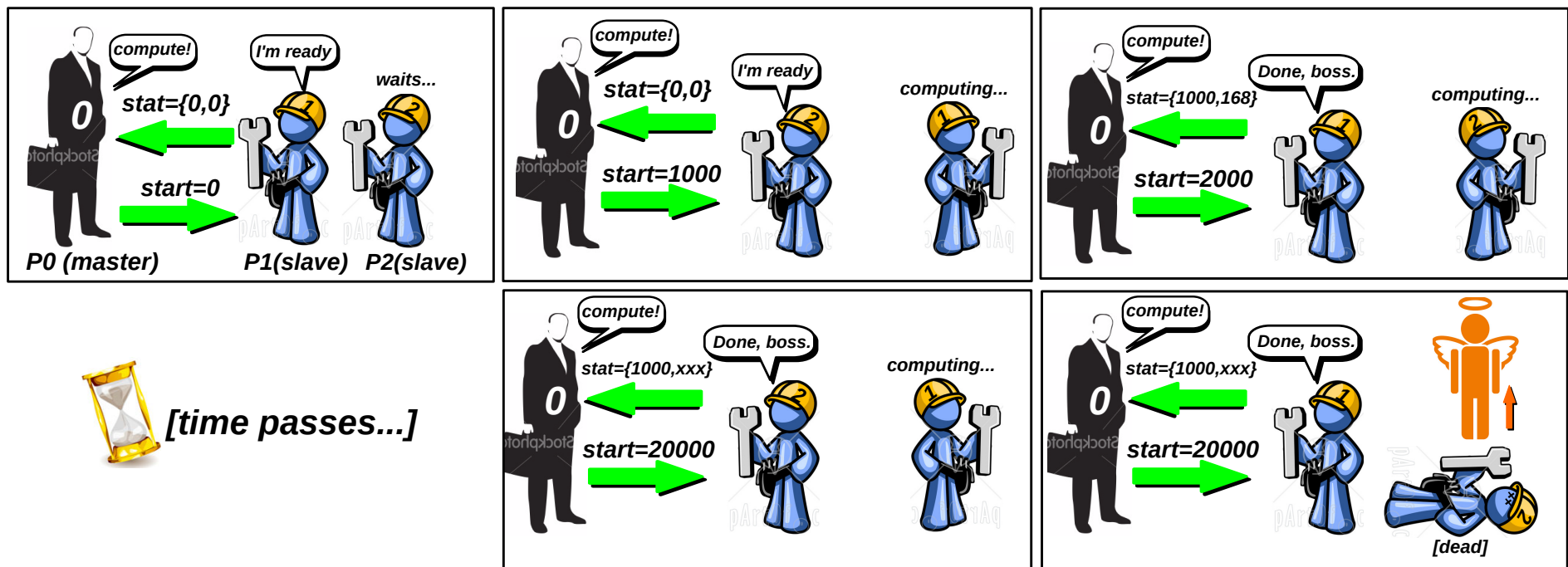
- Without node12 (proc1 in the previous example). Note that good balance is achieved. (detail)

PNT: dynamic parallel version.

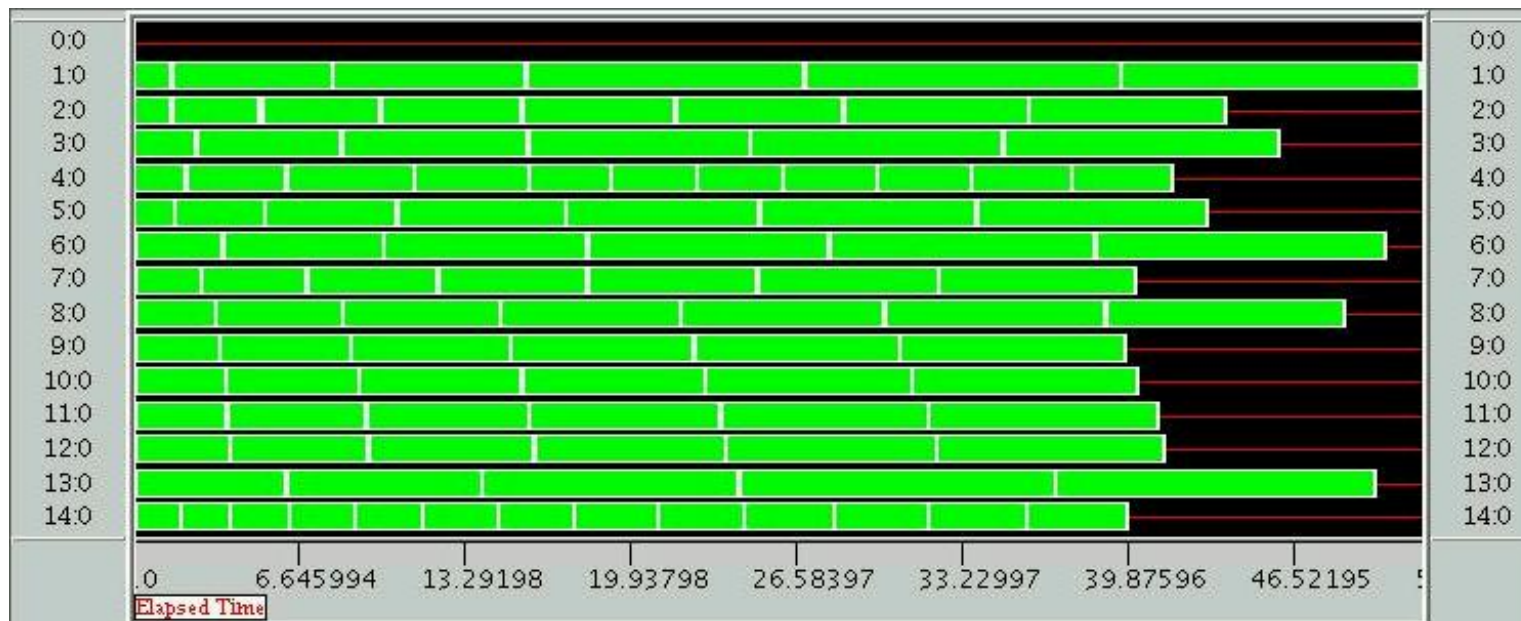
- Proc 0 acts like a server.
- Slaves send to the master the work already done and the server returns new work to be done.
- Work done is an integer array *stat*[2]: number of integers checked (*stat*[0]) and number of primes found (*stat*[1]).
- Work to be done is a single integer *start*. The slave knows that he must check for primality *chunk* integers starting at *start*.
- Initially all slaves send a message *stat*[]={0, 0} to the server signaling that they are ready to work.
- Automatically, when *last*>*N* the nodes realize that the run is over and stop.
- The master keeps a counter *down* of how much slaves have received the *stopping signal* (*first*>*N*). When this counter reaches *size-1* the server also stops.

PNT: dynamic parallel version. (cont.)

Compute $\pi(N = 20000)$, **chunk**=1000

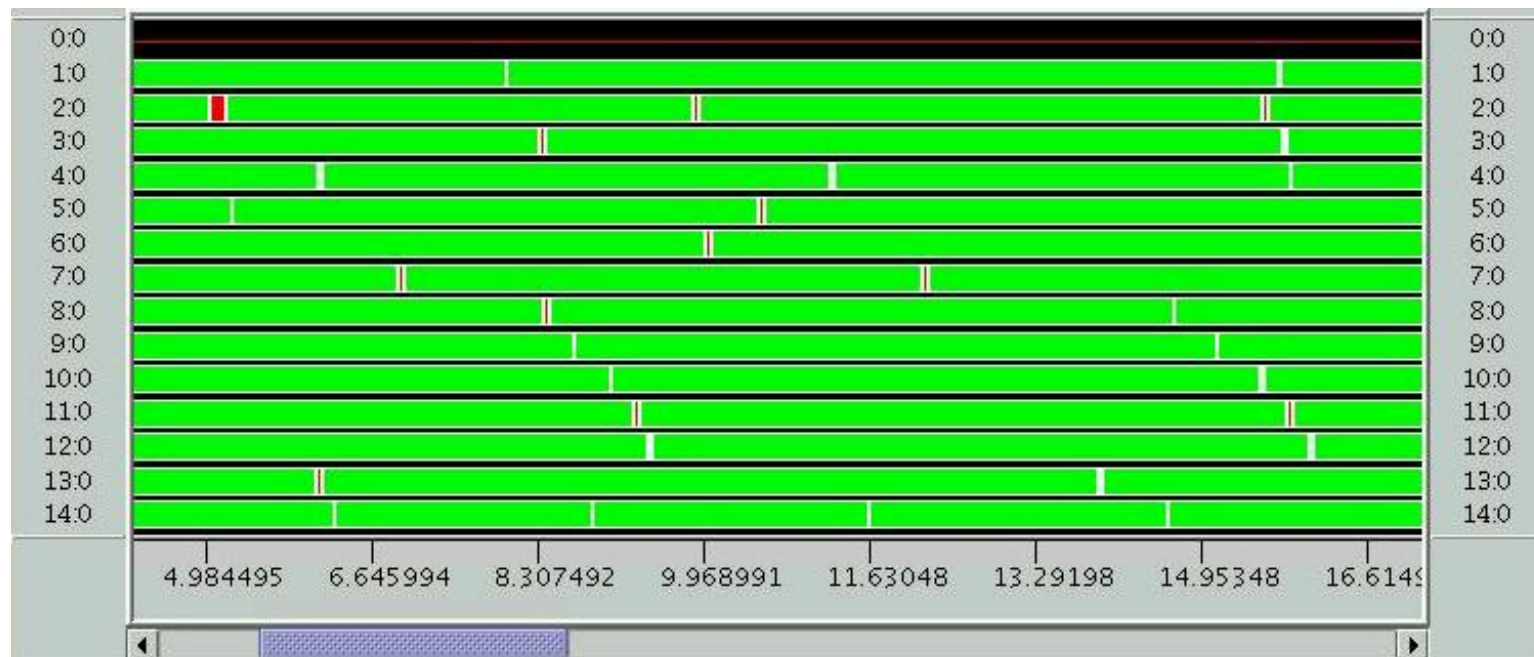


PNT: dynamic parallel version. (cont.)



- Very good dynamic balance.
- Inefficiency caused by high chunk size at the end of the run (it becomes negligible as we increase the size of the problem n , though).
- Note that certain processors (4 and 14) process more chunks (in a ratio 3:1 approx.) than the slower ones (1 and 13).
- Possible improvement: keep statistics and send smaller chunks to the slower processors. (Should be chunk size \propto processor speed approx.).

PNT: dynamic parallel version. (cont.)



PNT: Parallel dynamic version. Pseudocode

```
1  if (!myrank) {
2      int start=0, checked=0, down=0;
3      while(1) {
4          Recv(&stat, . . . , MPI_ANY_SOURCE, . . . , &status);
5          checked += stat[0];
6          primes += stat[1];
7          MPI_Send(&start, . . . , status.MPI_SOURCE, . . . );
8          if (start<N) start += chunk;
9          else down++;
10         if (down==size-1) break;
11     }
12 } else {
13     stat[0]=0; stat[1]=0;
14     MPI_Send(stat, . . . , 0, . . . );
15     while(1) {
16         int start;
17         MPI_Recv(&start, . . . , 0, . . . );
18         if (start>=N) break;
19         int last = start + chunk;
20         if (last>N) last=N;
21         stat[0] = last-start ;
22         stat[1] = 0;
23         for (int n=start; n<last; n++)
24             if (is_prime(n)) stat[1]++;
25         MPI_Send(stat, . . . , 0, . . . );
26     }
27 }
```

PNT: Parallel dynamic version. Code

```
1 // $Id: primes4.cpp, v 1.5 2004/10/03 14:35:43 mstorti Exp $
2 #include <mpi.h>
3 #include <mpe.h>
4 #include <stdio.h>
5 #include <cmath>
6 #include <cassert>
7
8 int is_prime(int n) {
9     if (n<2) return 0;
10    int m = int(sqrt(n));
11    for (int j=2; j<=m; j++)
12        if (!(n % j)) return 0;
13    return 1;
14 }
15
16 int main(int argc, char **argv) {
17     MPI_Init(&argc, &argv);
18     MPE_Init_log();
19
20     int myrank, size;
21     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22     MPI_Comm_size(MPI_COMM_WORLD, &size);
23
24     assert(size>1);
25     int start_comp = MPE_Log_get_event_number();
26     int end_comp = MPE_Log_get_event_number();
27     int start_comm = MPE_Log_get_event_number();
```



```

28  int end_comm = MPE_Log_get_event_number();
29
30  int chunk=200000, N=20000000;
31  MPI_Status status;
32  int stat[2]; // checked,primes
33
34  #define COMPUTE 0
35  #define STOP 1
36
37  if (!myrank) {
38      MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
39      MPE_Describe_state(start_comm,end_comm,"comm","red:white");
40      int first=0, checked=0, down=0, primes=0;
41      while (1) {
42          MPI_Recv(&stat,2,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
43                  MPI_COMM_WORLD,&status);
44          int source = status.MPI_SOURCE;
45          if (stat[0]) {
46              checked += stat[0];
47              primes += stat[1];
48              printf("recvd %d primes from %d, checked %d, cum primes %d\n",
49                     stat[1],source,checked,primes);
50          }
51          printf("sending [%d,%d) to %d\n",first,first+chunk,source);
52          MPI_Send(&first,1,MPI_INT,source,0,MPI_COMM_WORLD);
53          if (first<N) first += chunk;
54          else printf("shutting down %d, so far %d\n",source,++down);
55          if (down==size-1) break;
56      }
57  } else {
58      int start;

```



```

59     stat[0]=0; stat[1]=0;
60     MPI_Send(stat, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
61     while(1) {
62         MPE_Log_event(start_comm, 0, "start-comm");
63         MPI_Recv(&start, 1, MPI_INT, 0, MPI_ANY_TAG,
64                 MPI_COMM_WORLD, &status);
65         MPE_Log_event(end_comm, 0, "end-comm");
66         if (start>=N) break;
67         MPE_Log_event(start_comp, 0, "start-comp");
68         int last = start + chunk;
69         if (last>N) last=N;
70         stat[0] = last-start;
71         stat[1] = 0;
72         if (start<2) start=2;
73         for (int n=start; n<last; n++) if (is_prime(n)) stat[1]++;
74         MPE_Log_event(end_comp, 0, "end-comp");
75         MPE_Log_event(start_comm, 0, "start-comm");
76         MPI_Send(stat, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
77         MPE_Log_event(end_comm, 0, "end-comm");
78     }
79 }
80 MPE_Finish_log("primes");
81 MPI_Finalize();
82 }

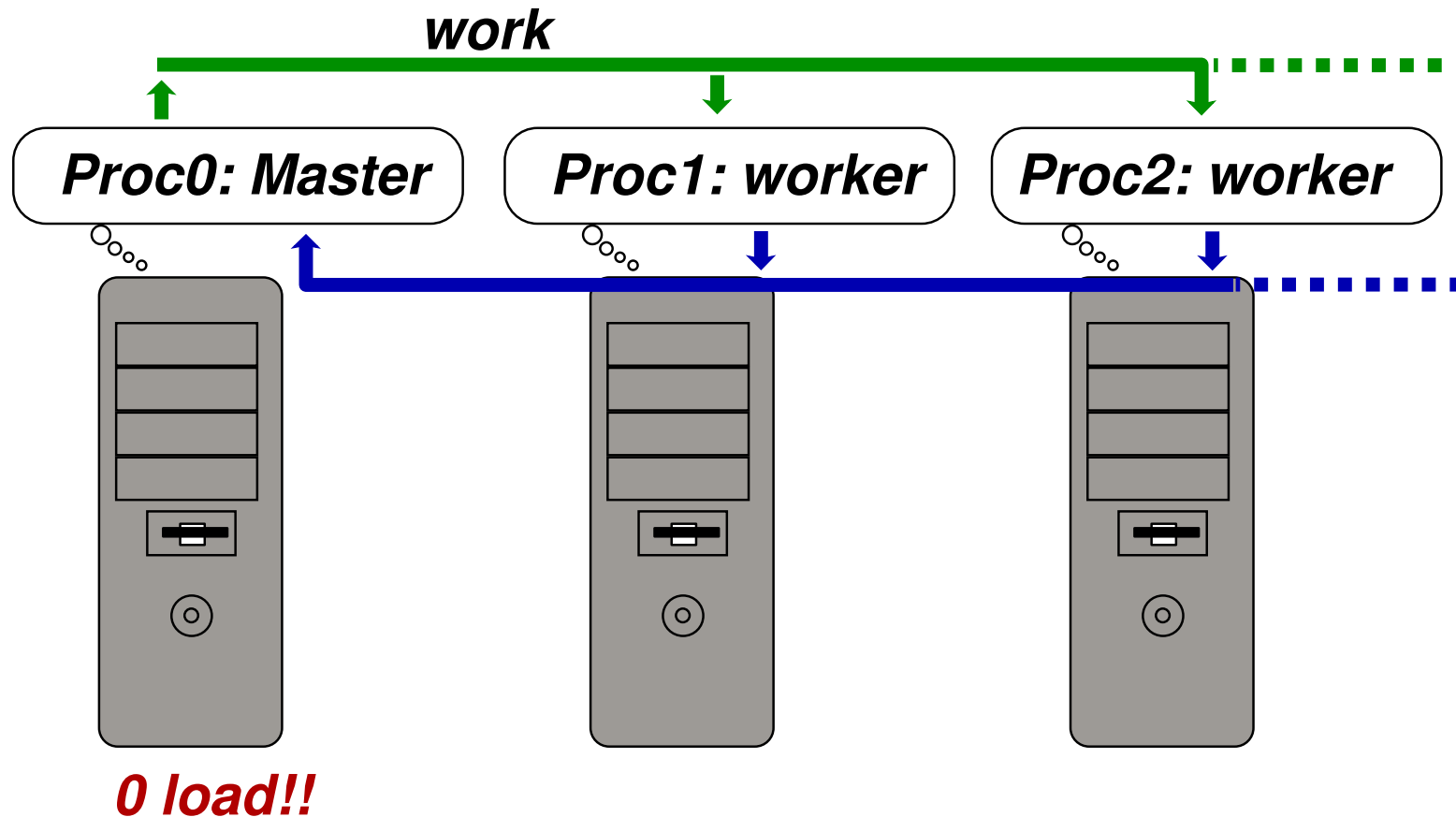
```

PNT: Parallel dynamic version

- This kind of strategy is called *master/slave* or *compute-on-demand*. Slaves wait work to be sent and once the work is completed they return the result, waiting for more work.
- Can be implemented in several ways
 - ▷ One process per processor. The master process answers *immediately* to the work demand by the slaves. *One processor is lost*.
 - ▷ Launch two process in processor 0: *myrank=0* (master) and *myrank=1* (worker). This can cause a delay in the answer of the master. *All processors do work*.
 - ▷ Modify the code in the master process (*myrank=0*) so that it does some work while waiting for work demand from the slaves (check OS time slice).
- Which of the previous alternatives is better depends on the size of the individual work to be done by the master, and of the time-slice of the operating system.

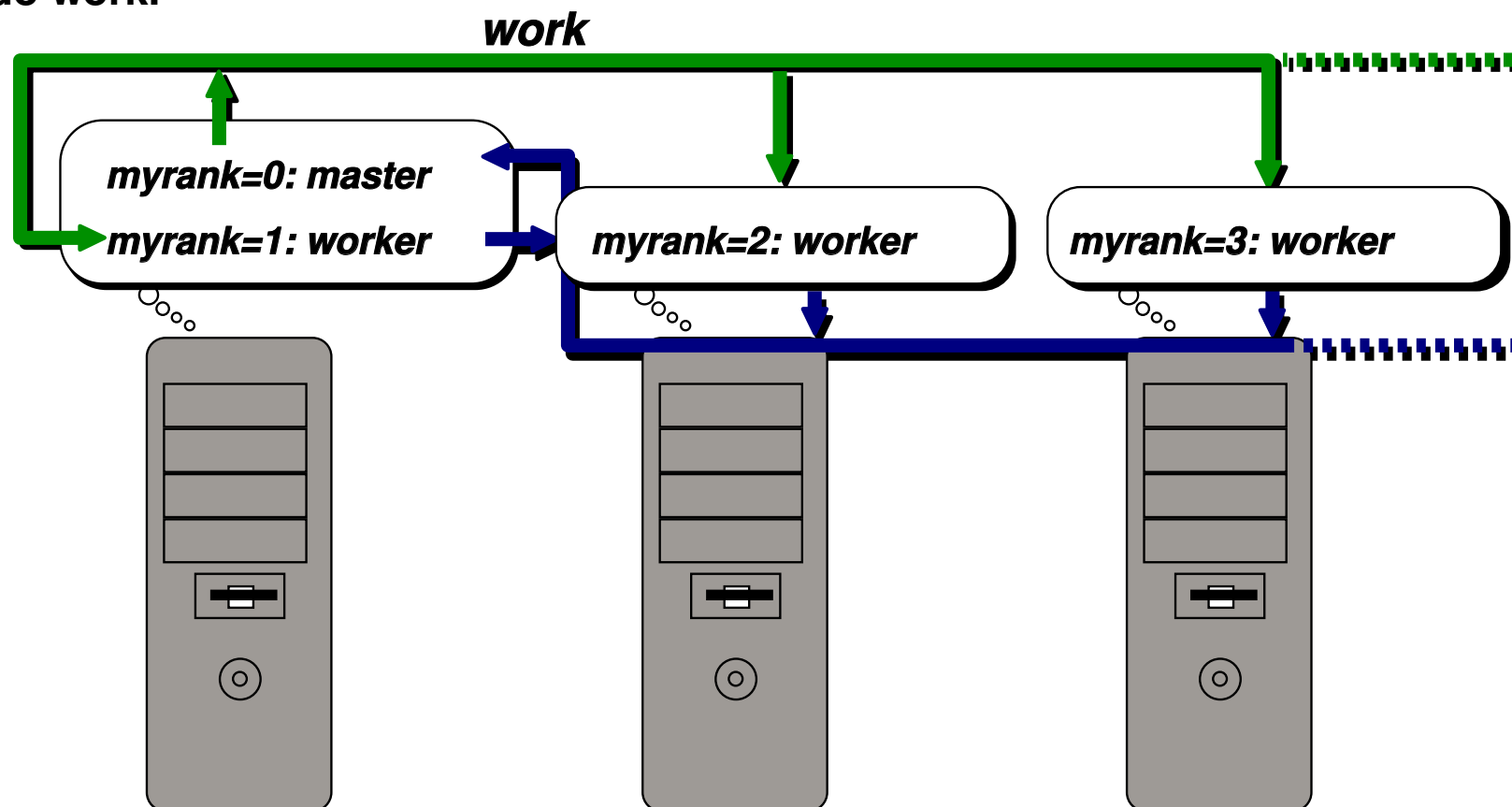
PNT: Parallel dynamic version (cont.)

One process per processor. The master process answers immediately to the work demand by the slaves. One processor is lost.



PNT: Parallel dynamic version (cont.)

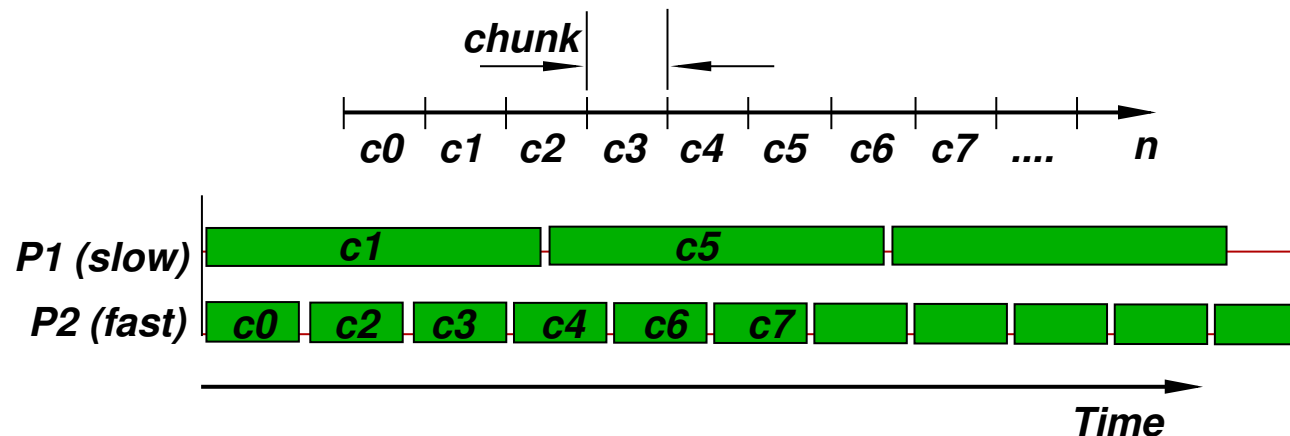
Launch two process in processor 0: *myrank=0* (master) and *myrank=1* (worker). This can cause a delay in the answer of the master. All processors do work.



PNT: Parallel dynamic version (cont.)

- Report times computing $\pi(5 \times 10^7) = 3001136$.
- 16 nodos ((fast P4HT, 3.0GHZ, DDR-RAM, 400MHz, dual channel), (slow P4HT, 2.8GHZ, DDR-RAM, 400MHz)).
- Sequential ($np = 1$) in node 10 (fast): 285 sec.
- Sequential ($np = 1$) in node 24 (slow): 338 sec.
- With $np = 12$ (disbalanced, blocking) 69 sec.
- Excluding the loaded node (balanced, blocking, $np = 11$): 33sec.
- With ($np = 14$) (dynamic balance, loaded with other processes, average load=1) 59 sec.

Printing values of $\Pi(n)$ in real time



- If there is great disbalance in processor speed, it may happen that a processor is returning the number of primes for a chunk, while a previous chunk was not yet computed, so that we can't simply accumulate *primes* in order to report $\pi(n)$.
- In the figure, chunks c_2, c_3, c_4 are reported *after* loading c_1 . Similarly c_6 and c_7 are reported after c_5 ...

Printing values of $PI(n)$ in real time (cont.)

```

1 struct chunk_info {
2     int checked, primes, start;
3 };
4 set<chunk_info> received;
5 vector<int> proc_start[size];
6
7 if (!myrank) {
8     int start=0, checked=0, down=0, primes=0;
9     while(1) {
10         Recv(&stat, . . . , MPI_ANY_SOURCE, . . . , &status);
11         int source = status.MPI_SOURCE;
12         checked += stat[0];
13         primes += stat[1];
14         // put (checked, primes, proc_start[source]) en 'received' . . .
15         // report last pi(n) computed . . .
16         MPI_Send(&start, . . . , source, . . . );
17         proc_start[source]=start;
18         if (start<N) start += chunk;
19         else down++;
20         if (down==size-1) break;
21     }
22 } else {
23     stat[0]=0; stat[1]=0;
24     MPI_Send(stat, . . . , 0, . . . );
25     while(1) {
26         int start;

```

```
27     MPI_Recv(&start, . . . , 0, . . . );
28     if (start>=N) break;
29     int last = start + chunk;
30     if (last>N) last=N;
31     stat[0] = last-start ;
32     stat[1] = 0;
33     for (int n=start; n<last; n++)
34         if (is_prime(n)) stat[1]++;
35     MPI_Send(stat, . . . , 0, . . . );
36 }
37 }
```


Printing values of $PI(n)$ in real time (cont.)

```
1 // report last pi(n) computed
2 int pi=0, last_reported = 0;
3 while (!received.empty()) {
4     // Si el primero de 'received' es 'last_reported'
5     // entonces sacarlo de 'received' y reportarlo
6     set<chunk_info>::iterator q = received.begin();
7     if (q->start != last_reported) break;
8     pi += q->primes;
9     last_reported += q->checked;
10    received.erase(q);
11    printf("pi(%d) = %d (encolados %d)\n",
12           last_reported, pi, received.size());
13 }
```

Printing values of $PI(n)$ in real time (cont.)

```
1 // $Id: primes5.cpp, v 1.4 2004/07/25 15:21:26 mstorti Exp $
2 #include <mpi.h>
3 #include <mpe.h>
4 #include <cstdio>
5 #include <cmath>
6 #include <cassert>
7 #include <vector>
8 #include <set>
9 #include <unistd.h>
10 #include <ctype.h>
11
12 using namespace std;
13
14 int is_prime(int n) {
15     if (n<2) return 0;
16     int m = int(sqrt(double(n)));
17     for (int j=2; j<=m; j++)
18         if (!(n % j)) return 0;
19     return 1;
20 }
21
22 struct chunk_info {
23     int checked, primes, start;
24     bool operator<(const chunk_info& c) const {
25         return start<c.start;
26     }
27 };
28
29 int main(int argc, char **argv) {
```

```
30 MPI_Init (&argc, &argv);
31 MPE_Init_log();
32
33 int myrank, size;
34 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
35 MPI_Comm_size(MPI_COMM_WORLD, &size);
36
37 assert(size>1);
38 int start_comp = MPE_Log_get_event_number();
39 int end_comp = MPE_Log_get_event_number();
40 int start_comm = MPE_Log_get_event_number();
41 int end_comm = MPE_Log_get_event_number();
42
43 int chunk = 20000, N = 200000;
44 char *cvalue = NULL;
45 int index;
46 int c;
47 opterr = 0;
48
49 while ((c = getopt (argc, argv, "N:c:")) != -1)
50     switch (c) {
51     case 'c':
52         sscanf(optarg, "%d", &chunk); break;
53     case 'N':
54         sscanf(optarg, "%d", &N); break;
55     case '?':
56         if (isprint (optopt))
57             fprintf (stderr, "Unknown option `-%c'.\n", optopt);
58         else
59             fprintf (stderr,
60                     "Unknown option character `\\x%x'.\n",
61                     optopt);
```

```

62     return 1;
63     default:
64         abort ();
65     };
66
67     if (!myrank)
68         printf ("chunk %d, N%d\n", chunk, N);
69
70     MPI_Status status;
71     int stat[2]; // checked, primes
72     set<chunk_info> received;
73     vector<int> start_sent(size, -1);
74
75     #define COMPUTE 0
76     #define STOP 1
77
78     if (!myrank) {
79         MPE_Describe_state(start_comp, end_comp, "comp", "green:gray");
80         MPE_Describe_state(start_comm, end_comm, "comm", "red:white");
81         int first=0, checked=0,
82             down=0, primes=0, first_recv = 0;
83         while (1) {
84             MPI_Recv(&stat, 2, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
85                     MPI_COMM_WORLD, &status);
86             int source = status.MPI_SOURCE;
87             if (stat[0]) {
88                 assert(start_sent[source] >= 0);
89                 chunk_info cs;
90                 cs.checked = stat[0];
91                 cs.primes = stat[1];
92                 cs.start = start_sent[source];

```

```

93         received.insert(cs);
94         printf("recvd %d primes from %d\n",
95             stat[1], source, checked, primes);
96     }
97     while (!received.empty()) {
98         set<chunk_info>::iterator q = received.begin();
99         if (q->start != first_recv) break;
100         primes += q->primes;
101         received.erase(q);
102         first_recv += chunk;
103     }
104     printf("pi(%d) = %d, (queued %d)\n",
105         first_recv+chunk, primes, received.size());
106     MPI_Send(&first, 1, MPI_INT, source, 0, MPI_COMM_WORLD);
107     start_sent[source] = first;
108     if (first<N) first += chunk;
109     else {
110         down++;
111         printf("shutting down %d, so far %d\n", source, down);
112     }
113     if (down==size-1) break;
114 }
115 set<chunk_info>::iterator q = received.begin();
116 while (q!=received.end()) {
117     primes += q->primes;
118     printf("pi(%d) = %d\n", q->start+chunk, primes);
119     q++;
120 }
121 } else {
122     int start;
123     stat[0]=0; stat[1]=0;

```

```

124 MPI_Send(stat, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
125 while(1) {
126     MPE_Log_event(start_comm, 0, "start-comm");
127     MPI_Recv(&start, 1, MPI_INT, 0, MPI_ANY_TAG,
128             MPI_COMM_WORLD, &status);
129     MPE_Log_event(end_comm, 0, "end-comm");
130     if (start >= N) break;
131     MPE_Log_event(start_comp, 0, "start-comp");
132     int last = start + chunk;
133     if (last > N) last = N;
134     stat[0] = last - start;
135     stat[1] = 0;
136     if (start < 2) start = 2;
137     for (int n = start; n < last; n++) if (is_prime(n)) stat[1]++;
138     MPE_Log_event(end_comp, 0, "end-comp");
139     MPE_Log_event(start_comm, 0, "start-comm");
140     MPI_Send(stat, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
141     MPE_Log_event(end_comm, 0, "end-comm");
142 }
143 }
144 MPE_Finish_log("primes");
145 MPI_Finalize();
146 }

```

Reading data and options

- *Read options from a common file via NFS*

```
1 FILE *fid = fopen("options.dat", "r");
2 int N; double mass;
3 fscanf(fid, "%d", &N);
4 fscanf(fid, "%lf", &mass);
5 fclose(fid);
```

(OK for a small data volume)

- *Read from console (stdin) and broadcast to the other nodes*

```
1 int N;
2 if (!myrank) {
3     printf("enter N> ");
4     scanf("%d", &N);
5 }
6 MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
7 // read and Bcast 'mass'
```

(OK for options. Needs Bcast).

Reading data and options (cont.)

- *Enter options via line command (Unix(POSIX)/getopt)*

```
1 #include <unistd.h>
2 #include <ctype.h>
3
4 int main(int argc, char **argv) {
5     MPI_Init(&argc, &argv);
6     // ...
7     int c;
8     opterr = 0;
9     while ((c = getopt (argc, argv, "N:c:")) != -1)
10         switch (c) {
11             case 'c':
12                 sscanf(optarg, "%d", &chunk); break;
13             case 'N':
14                 sscanf(optarg, "%d", &N); break;
15         };
16     // ...
17 }
```

- *For large data volumes: read in master and Bcast*

Scalability

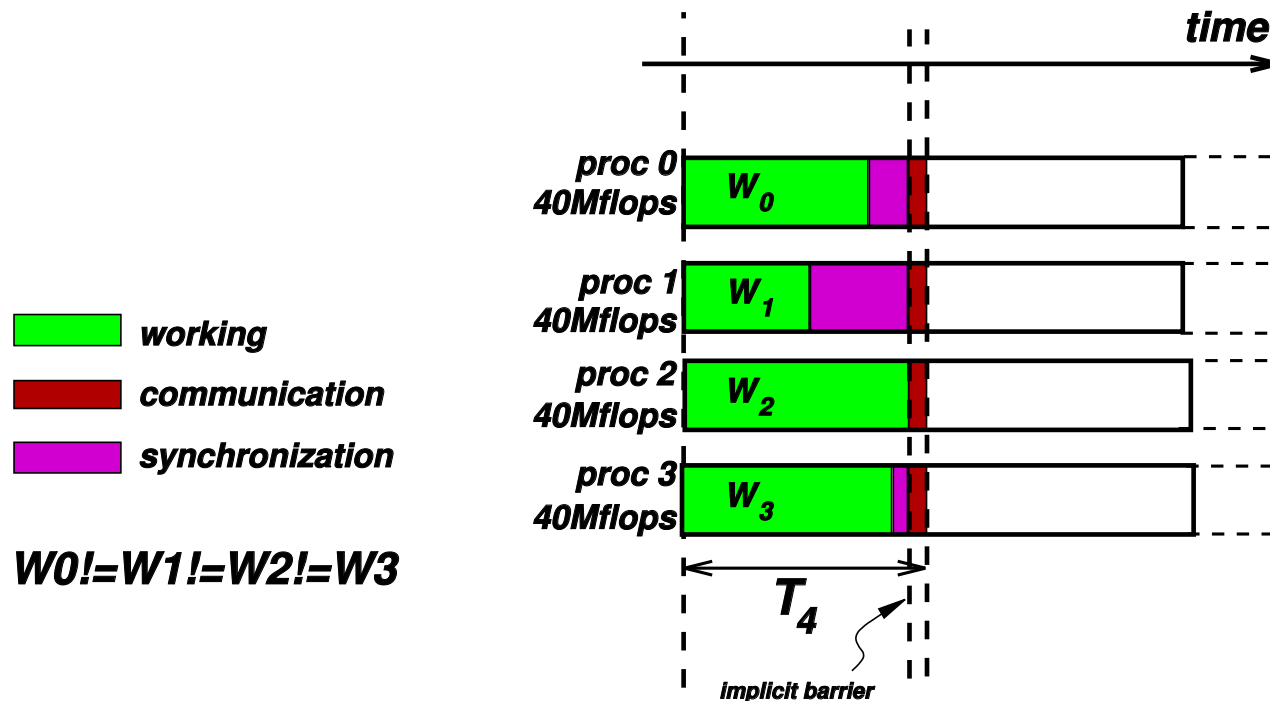
The problem of scalability is worst if it is not easy to distribute the load, since in addition to **communication** we have an extra time expended in **synchronization**. In the PNT example, the cost of the primality test for a number j greatly varies with j , for even numbers is immediate, and the cost is basically proporcional to the lowest divisor. In addition, the cost grows with j , in average it is $O(\sqrt{j})$. Even if initially we can send a certain amount of m integers to each processor, the real load, i.e. the quantity of work is not known a priori. In the static partitioning algorithm processor i will end in a certain time $T_{\text{comp},i} = W_i/s$ and will wait until the last processor ends

Scalability (cont.)

$$T_n = \left(\max_i T_{\text{comp},i} \right) + T_{\text{comm}} = T_{\text{comp,max}} + T_{\text{comm}}$$

$$T_{\text{sync},i} = T_{\text{comp,max}} - T_{\text{comp},i}$$

$$T_n = T_{\text{comp},i} + T_{\text{sync},i} + T_{\text{comm}}$$



Scalability (cont.)

$$\begin{aligned}
 \eta &= \frac{S_n}{S_n^*} = \frac{T_1}{nT_n} = \frac{W/s}{n(\max_i T_{\text{comp},i} + T_{\text{comm}})} \\
 &= \frac{W/s}{\sum_i (T_{\text{comp},i} + T_{\text{sync},i} + T_{\text{comm}})} \\
 &= \frac{W/s}{\sum_i ((W_i/s) + T_{\text{sync},i} + T_{\text{comm}})} \\
 &= \frac{W/s}{(W/s) + \sum_i (T_{\text{sync},i} + T_{\text{comm}})} \\
 &= \frac{\text{(total comp. time)}}{\text{(total comp. time)} + \text{(total comm.+sync. time)}}
 \end{aligned}$$

PNT: Detailed scalability analysis

- We use the relation

$$\eta = \frac{(\text{tot.comp.time})}{(\text{tot.comp.time})+(\text{tot.comm.time})+(\text{tot.sync.time})}$$

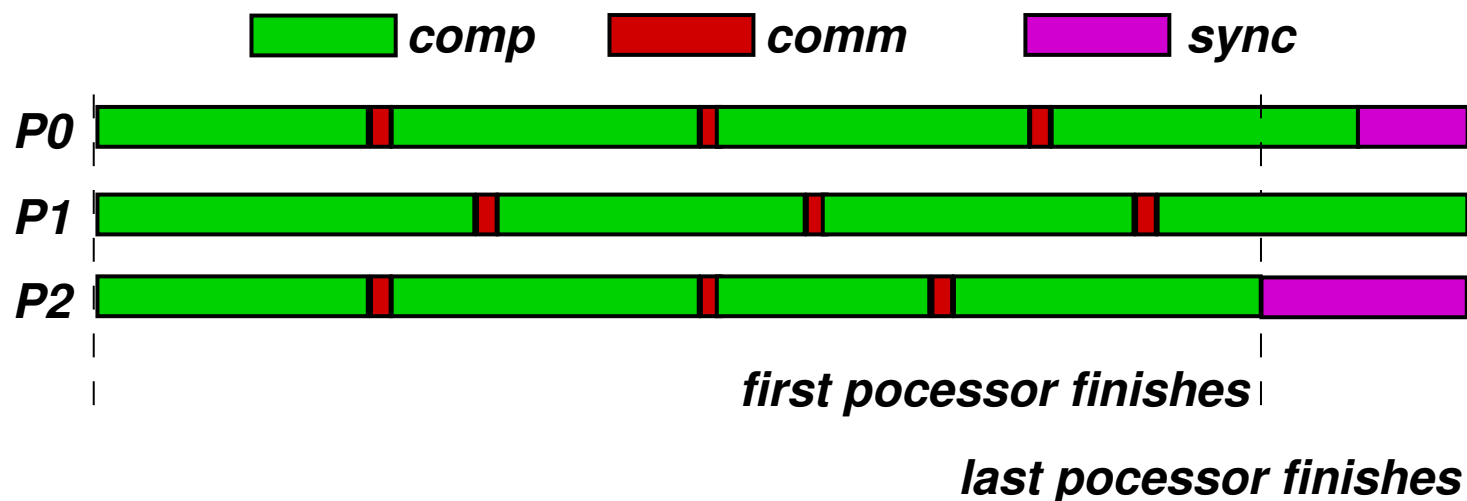
- Time to test integer j for primality: $O(j^{0.5})$
- $T_{\text{comp}}(N) = \sum_{j=1}^N c j^{0.5} \approx c \int_0^N j^{0.5} dj = 2cN^{1.5}$
- Communication time in sending and receiving an integer per chunk

$$T_{\text{comm}}(N) = (\text{nbr. of chunks})2l = \frac{N}{N_c} 2l$$

where l is latency and N_c is the chunk length.

PNT: Detailed scalability analysis (cont.)

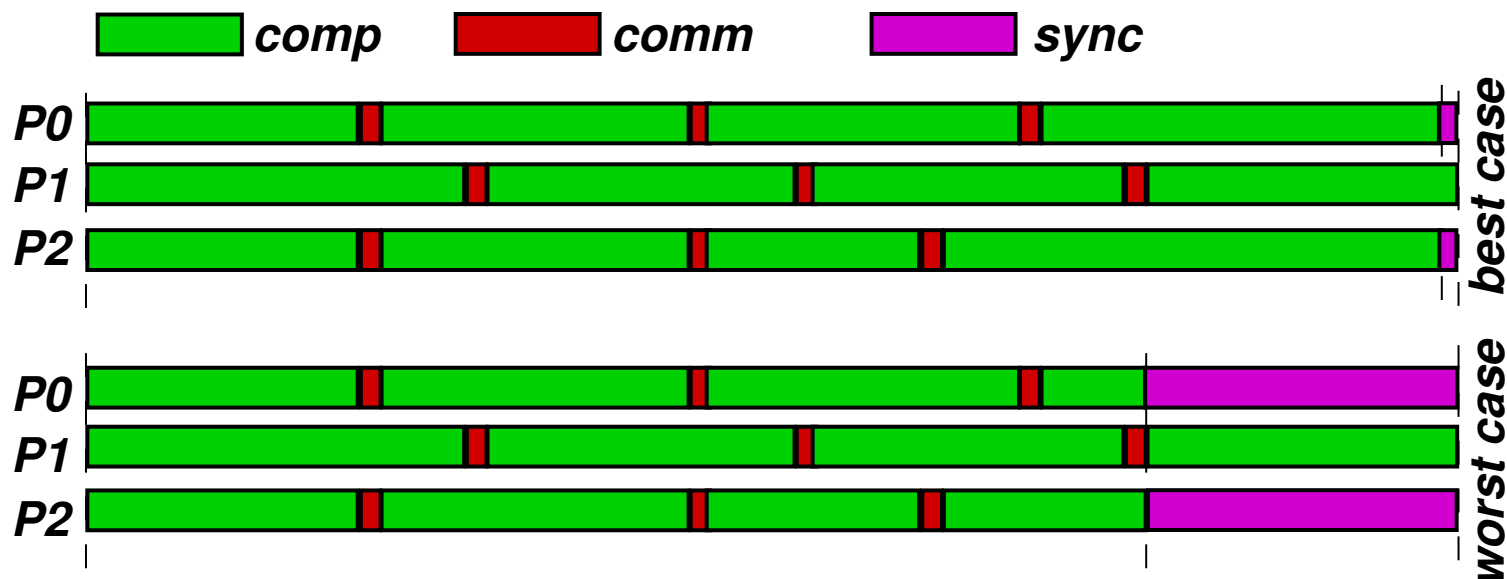
- Synchronization time is almost random.



PNT: Detailed scalability analysis (cont.)

- It may happen that all processes end at the same time, in which case $T_{\text{sync}} = 0$. The worst case is when all processes end almost at the same time, remaining only one process still computing.

$$T_{\text{sync}} = (n - 1)(\text{processing time for a chunk})$$



PNT: Detailed scalability analysis (cont.)

- In average

$$T_{\text{sync}} = (n/2)(\text{processing time for a chunk})$$

- In our case

$$(\text{processing time for a chunk}) \leq N_c c N^{0.5}$$

$$T_{\text{sync}} = (n/2)c N_c N^{0.5}$$

PNT: Detailed scalability analysis (cont.)

$$\begin{aligned}\eta &= \frac{2cN^{0.5}}{2cN^{0.5} + (N/N_c)2l + (n/2)N_cN^{0.5}} \\ &= \left(1 + (N^{0.5}l/c)/N_c + (nN^{0.5}/N^{1.5})N_c\right)^{-1} \\ &= (1 + A/N_c + BN_c)^{-1}\end{aligned}$$

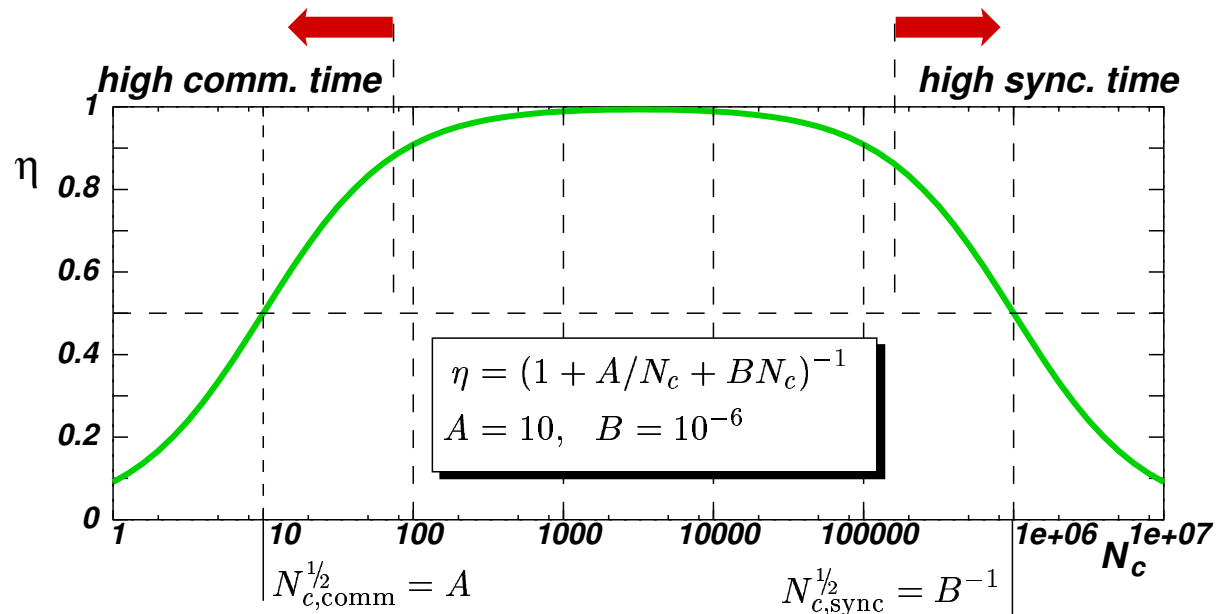
$$N_{c,\text{opt}} = \sqrt{A/B}$$

$$\eta_{\text{opt}} = (1 + 2\sqrt{B/A})^{-1}$$

PNT: Detailed scalability analysis (cont.)

$$\eta = (1 + A/N_c + BN_c)^{-1}$$

- For $N_c \gg A$ communication is negligible.
- For $N_c \ll B^{-1}$ synchronization is negligible.
- If $A \ll B^{-1}$ then we have a **window** $[A, B^{-1}]$ where efficiency is kept $\eta > 0.5$.



Load balance

Performance in heterogeneous clusters

- If processor speed is not the same for all processors than the others and the same amount of work is assigned to all processors, then the fastest ones must wait to the slowest one, so that the computing speed is at most n times the speed of the slowest. So, under these conditions there is a loss in performance for heterogeneous clusters. The concept of speedup must be extended to heterogeneous groups of processors.

Performance in heterogeneous clusters (cont.)

- Assume that the work W (a given number of operations to be performed independently one of the other) is divided in n equal parts $W_i = W/n$.
- Each processor spends $t_i = W_i/s_i = W/n s_i$ seconds, where s_i is the speed of processor i (for instance in Mflops). The fact that the t_i are not equal now, shows that there is a loss in efficiency.
- The total elapsed time corresponds to the largest t_i , which belongs to the smaller (slowest) s_i :

$$T_n = \max_i t_i = \frac{W}{n \min_i s_i}$$

Performance in heterogeneous clusters (cont.)

- For the time T_1 (computing time for one processor) we can take that one corresponding to the fastest ones

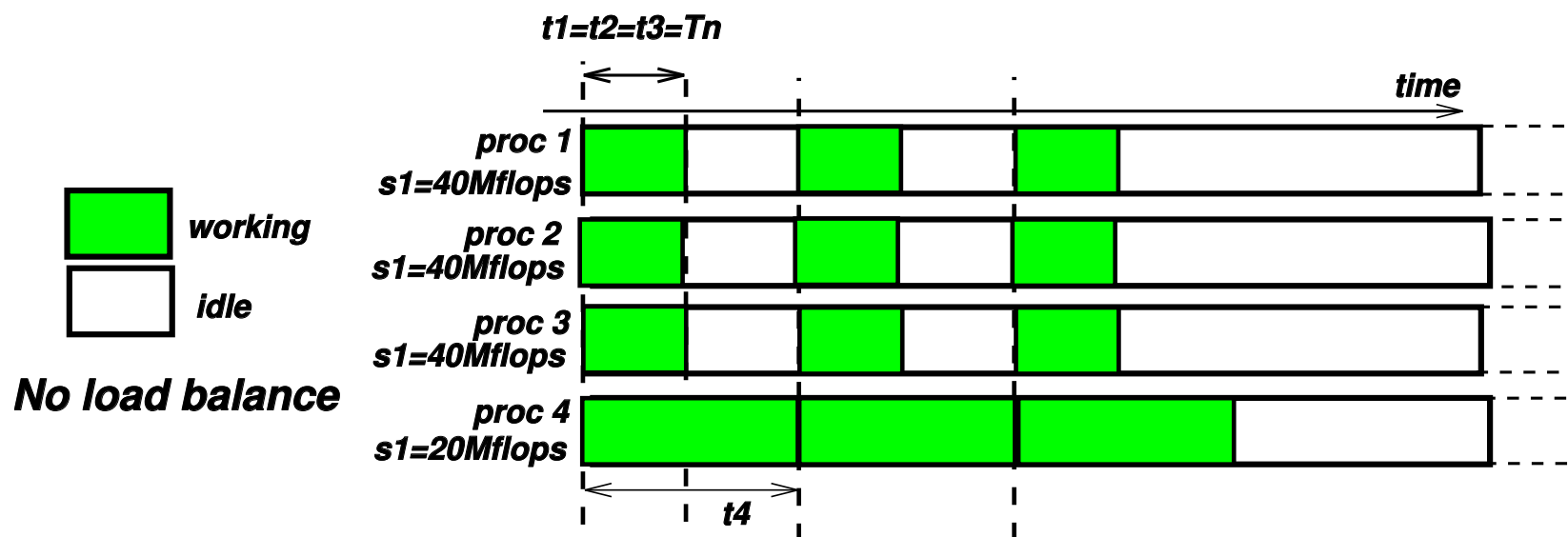
$$T_1 = \min t_i = \frac{W}{\max_i s_i}$$

- The speedup results then in

$$S_n = \frac{T_1}{T_n} = \frac{W}{\max_i s_i} / \frac{W}{n \min_i s_i} = n \frac{\min_i s_i}{\max_i s_i}$$

For instance, if we have a cluster with 12 processors with relative speeds: 8 nodes @ 4 Gflops and 4 nodes @ 2.4 Gflops, so the **disbalance factor** $\min_i s_i / \max_i s_i$ is 0.6. The theoretical speedup is then $12 \times 0.6 = 7.2$, so that the total throughput **is lower than the case where we take the 8 fastest processors only**. (In addition we are not taking into account the speedup reduction do to communication times.)

Performance in heterogeneous clusters (cont.)



Load balance

- If we distribute the work proportional to the computing speed of the nodes

$$W_i = W \frac{s_i}{\sum_j s_j}, \quad \sum_j W_j = W$$

- The time spent in each processor is

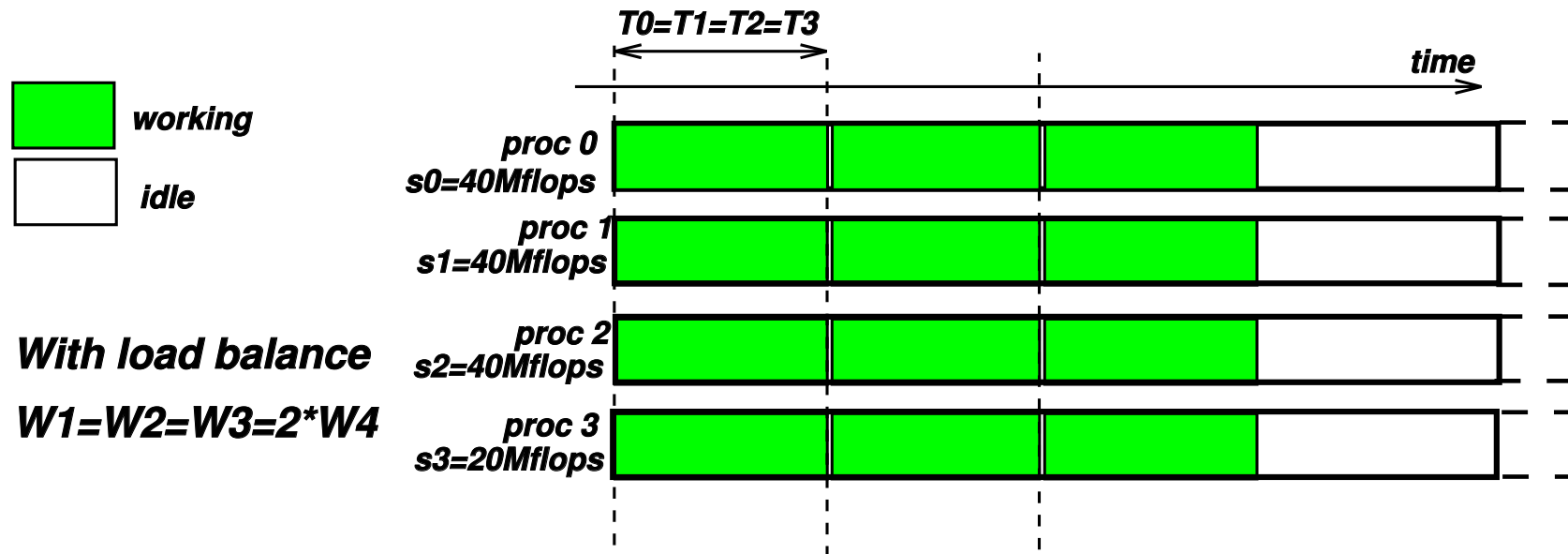
$$t_i = \frac{W_i}{s_i} = \frac{W}{\sum_j s_j} \quad (\text{independent of } i!!)$$

- The speedup is now

$$S_n = \frac{T_1}{T_n} = (W / \max_j s_j) / \left(\frac{W}{\sum_j s_j} \right) = \frac{\sum_j s_j}{\max_j s_j}$$

This is the maximum speedup attainable in heterogeneous clusters.

Load balance (cont.)



Load balance (cont.)

Coming back to the example of the cluster with 12 processors with relative speeds: 8 nodes @ 4 Gflops and 4 nodes @ 2.4 Gflops, with load balance we expect a theoretical speed up of

$$S_n^* = \frac{8 \times 4\text{Gflops} + 4 \times 2.4\text{Gflops}}{4\text{Gflops}} = 10.4 \quad (1)$$

Another way to see this is that, since the relative speed of the slower nodes is $2.4/4 = 0.6$ w.r.t. to the faster nodes, each of the slower nodes is equivalent to 0.6 of a fast node, and then the slower nodes bring at most $4 \times 0.6 = 2.4$ fast nodes. Then we have at most a speedup of $8 + 2.4 = 10.4$.

Trivial parallelism

- The PNT example (*primes.cpp*) introduced concepts like point-to-point communication and collective functions.
- *Compute-on-demand* may be also implemented for sequential programs. Suppose that we want to compute a series of values $f(x_i), i = 0, \dots, m - 1$, for which we have a sequential program *comptef -x <x-val>* that prints on standard output the value of $f(x)$.

```
1 [mstorti@spider curso]> comptef -x 0.3
2 0.34833467364
3 [mstorti@spider curso]>
```

Trivial parallelism (cont.)

```

1  // Get parameters 'm' and 'xmax'
2  if (!myrank) {
3      vector<int> proc_j(size, -1);
4      vector<double> table(m);
5      double x=0., val;
6      for (int j=0; j<m; j++) {
7          Recv(&val, . . . , MPI_ANY_SOURCE, . . . , &status);
8          int source = status.MPI_SOURCE;
9          if (proc_j[source]>=0)
10             table[proc_j[source]] = val;
11             MPI_Send(&j, . . . , source, . . . );
12             proc_start[source]=j;
13         }
14         for (int j=0; j<size-1; j++) {
15             Recv(&val, . . . , MPI_ANY_SOURCE, . . . , &status);
16             MPI_Send(&m, . . . , source, . . . );
17         }
18     } else {
19         double
20         val = 0.0,
21         deltax = xmax/double(m);
22         char line[100], line2[100];
23         MPI_Send(val, . . . , 0, . . . );
24         while(1) {
25             int j;
26             MPI_Recv(&j, . . . , 0, . . . );

```

```
27     if (j>=m) break;
28     sprintf(line, "compute -x %f > proc%d.output", j*deltax, myrank);
29     sprintf(line2, "proc%d.output", myrank);
30     FILE *fid = fopen(line2, "r");
31     fscanf(fid, "%lf", &val);
32     system(line)
33     MPI_Send(val, . . . , 0, . . . );
34 }
35 }
```

The traveling salesman problem (TSP)

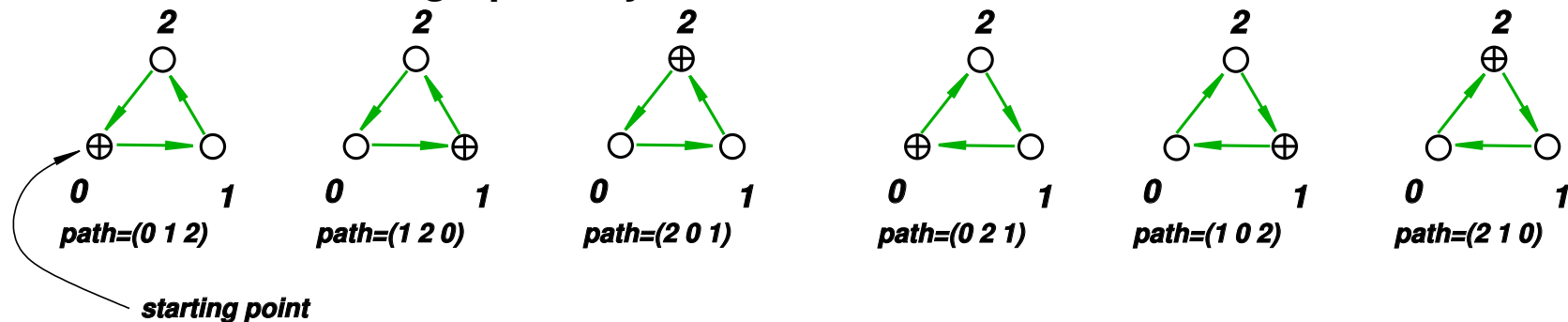
The traveling salesman problem (TSP)

The TSP (*Traveling Salesman Problem*) consists in finding the shortest path that runs all vertices from a **graph** of n vertices, passing only once through each city (i.e. vertex) and coming back to the starting point. A table $d[n][n]$ contains the distances among vertices, i.e. $d[i][j] > 0$ is the distance between vertex i and j . We assume that the graph (and so the distance table) is **symmetric** Asumimos que la tabla de distancias es simétrica ($d[i][j] = d[j][i]$) y $d[i][i] = 0$.

(Learn more? <http://www.wikipedia.org>).

The traveling salesman problem (TSP) (cont.)

Consider for instance the case of $N_v = 3$ vertices, numbered from 0 to 2. All possible paths can be represented as the *permutations* of N_v objects. So, there are $3! = 6$ distinct paths, as shown in the figure. At the bottom of each path we show the corresponding permutation. As we can see, in fact the three paths on the right are equivalent, since they differ only in the starting point (marked with a cross), and the same holds for the three on the left. In addition, the three at the right are equivalent to those on the left, since the only difference is the *sense* (clockwise or counterclockwise) which, of course, is immaterial since the graph is symmetric.



The traveling salesman problem (TSP) (cont.)

We can generate all the paths with the following algorithm. We take for instance all possible paths with two vertices, which are $(0, 1)$ and $(1, 0)$. Then the paths with three vertices can be obtained by inserting the new vertex 2 in each of the 3 possible positions in each of the paths for two vertices. So that, for each 2-vertex path we have three new 3-vertex paths. So,

$$N_{\text{path}}(3) = 3 \cdot N_{\text{path}}(2) = 3 \cdot 2 = 6$$

In general we have,

$$N_{\text{path}}(N_v) = N_v \cdot N_{\text{path}}(N_v - 1) = \dots = N_v!$$

The traveling salesman problem (TSP) (cont.)

In general, for each path we can obtain N_v equivalent paths by changing the starting point, so that the number of different paths is

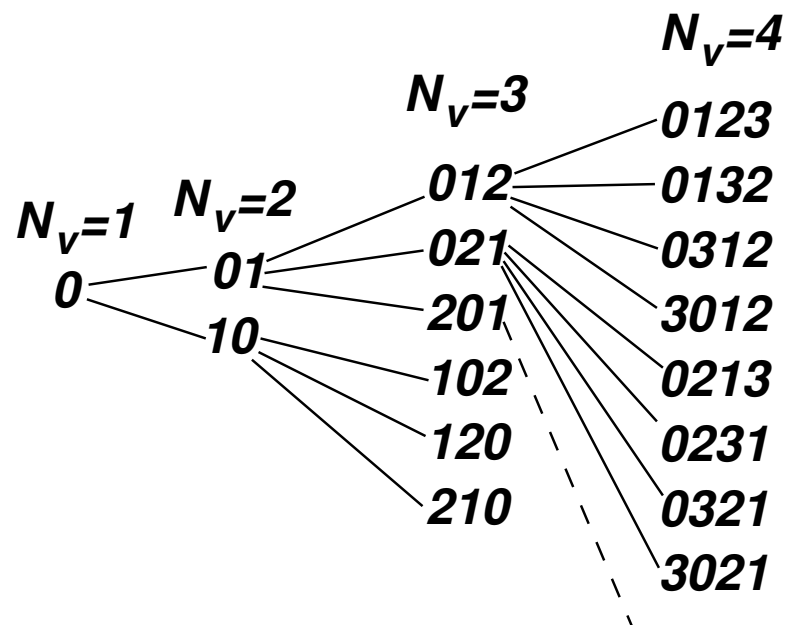
$$N_{\text{path}}(N_v) = \frac{N_v!}{N_v} = (N_v - 1)!$$

In addition, if we take into account that for each path we can obtain another equivalent one by reverting the sense of the path. So, the number of different paths is further reduced to

$$N_{\text{path}}(N_v) = \frac{(N_v - 1)!}{2}$$

The traveling salesman problem (TSP) (cont.)

The recursive process mentioned previously generates the following paths



The traveling salesman problem (TSP) (cont.)

Looking at the sequences for a given N_v we deduce an algorithm for computing the following path from the previous one. We store the path in an integer array *int *path*, and the function *next_path* advances it returning 0 if the last path was reached and 0 otherwise.

```
1 int next_path(int *path, int N) {
2     for (int j=N-1; j>=0; j--) {
3         // Is 'j' in first position?
4         if (path[0]==j) {
5             // move 'j' to the j-th position ...
6         } else {
7             // exchange 'j' with its predecessor ...
8             return 1;
9         }
10    }
11    return 0;
12 }
```

The traveling salesman problem (TSP) (cont.)

$$p_0 = (0, 1, 2)$$

$$p_1 = (0, 2, 1)$$

$$p_2 = (2, 0, 1)$$

$$p_3 = (1, 0, 2)$$

$$p_4 = (1, 2, 0)$$

$$p_5 = (2, 1, 0)$$

- For instance, for $N_v = 3$ we generate the paths in the figure.
- When applying the algorithm to path p_0 we see that we simply exchange vertex 2 with his predecessor.
- For path p_2 we can advance again 2, since it is already at the first position. We then move it to the end, leaving $(0, 1, 2)$ and we try to advance 1, leaving $(1, 0, 2)$.
- When applied to p_5 we see that in fact we cant advance any vertex, so this means that the last path has been reached and the function returns 0.

The traveling salesman problem (TSP) (cont.)

Complete code for *next_path()*

```
1 int next_path(int *path, int N) {
2     for (int j=N-1; j>=0; j--) {
3         if (path[0]==j) {
4             for (int k=0; k<j; k++) path[k] = path[k+1];
5             path[j] = j;
6         } else {
7             int k;
8             for (k=0; k<N-1; k++)
9                 if (path[k]==j) break;
10            path[k] = path[k-1];
11            path[k-1] = j;
12            return 1;
13        }
14    }
15    return 0;
16 }
```

The traveling salesman problem (TSP) (cont.)

So, in order to visit all possible paths we do something like this

```
1 int path[Nv];
2 for (int j=0; j<Nv; j++) path[j] = j;
3
4 while(1) {
5     // do something with path....
6     if (!next_path(path, Nv)) break;
7 }
```

For instance, if we define the function

*double dist(int *path, int Nv, double *d);* that computes the distance for a given path, we can find the minimum distance with the following code

```
1 int path[Nv];
2 for (int j=0; j<Nv; j++) path[j] = j;
3
4 double dmin = DBL_MAX;
5 while(1) {
6     double D = dist(path, Nv, d);
7     if (D<dmin) dmin = D;
8     if (!next_path(path, Nv)) break;
9 }
```

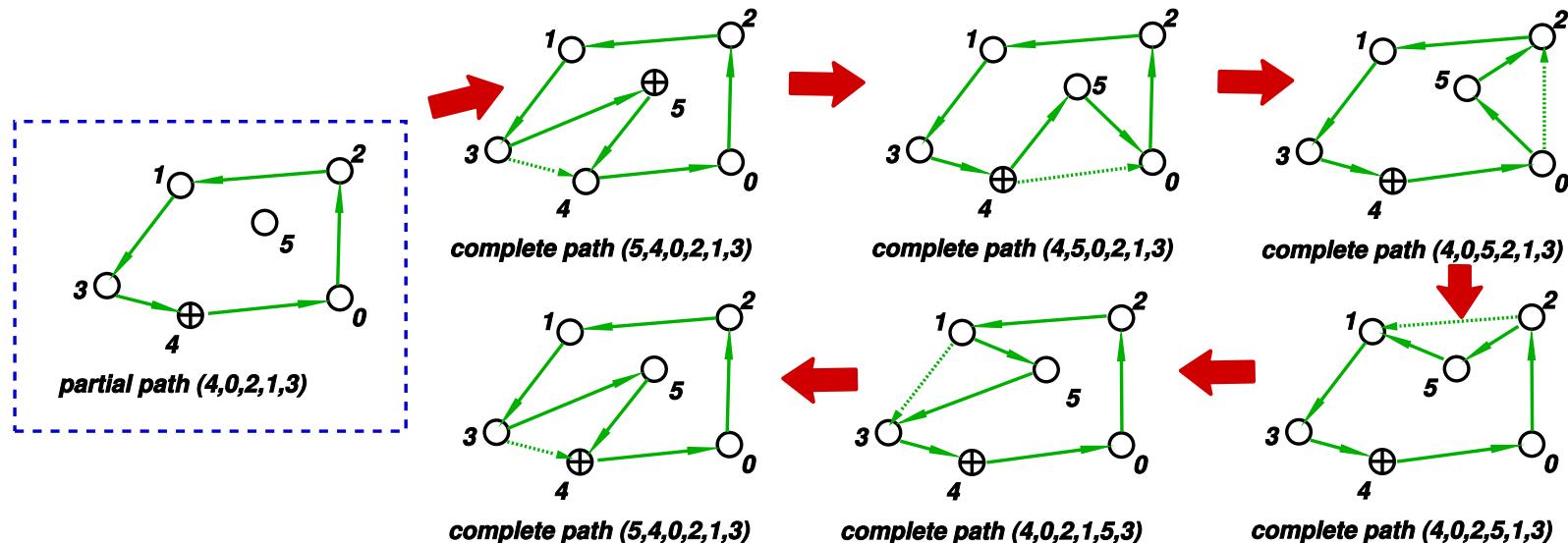
The traveling salesman problem (TSP) (cont.)

- **Possible dynamic implementation in parallel:** Generate **chunks** of N_c paths and sent to slaves for processing. For instance, if $N_v = 4$ then there are 24 possible path. If we choose $N_c = 5$ then the master generates chunks and send them to the slaves,
 - ▷ $\text{chunk0} = \{(0,1,2,3), (0,1,3,2), (0,3,1,2), (3,0,1,2), (0,2,1,3)\}$
 - ▷ $\text{chunk1} = \{(0,2,3,1), (0,3,2,1), (3,0,2,1), (2,0,1,3), (2,0,3,1)\}$
 - ▷ $\text{chunk2} = \{(2,3,0,1), (3,2,0,1), \dots\}$
 - ▷ ...**Do not scale well, since we have to send to the slaves $4N_cN$ bytes, so $T_{\text{comm}} = 4N_cN/b = O(N_c)$, and $T_{\text{comp}} = O(N_c)$.**

The traveling salesman problem (TSP) (cont.)

A **partial path** of length N_p is one possible paths for the first N_p vertices. The complete paths of length $N_p + 1$ can be obtained from the partial path by inserting a vertex in some position in the partial path.

In the figure we can see the paths of 6 vertices derived from a partial path of 5 vertices.



The traveling salesman problem (TSP) (cont.)

- **Posible parallel implementation (improved):** Send **chunks** that correspond to all the complete paths derived from a given **partial path** of length N_p . For instance, if $N_v = 4$ then we can send the partial path $(0, 1, 2)$, to the slave, and then the slave will deduce all derived complete paths, namely

- ▷ partial path $(0,1,2)$, chunk = $\{(0,1,2,3),(0,1,3,2),(0,3,1,2),(3,0,1,2)\}$
- ▷ partial path $(0,2,1)$, chunk = $\{(0,2,1,3),(0,2,3,1),(0,3,2,1),(3,0,2,1)\}$
- ▷ partial path $(2,0,1)$, chunk = $\{(2,0,1,3),(2,0,3,1),(2,3,0,1),(3,2,0,1)\}$
- ▷ ...

Each slave is in charge of generating all derived **complete paths** from this partial path. The partial path behaves like a **chunk**, but the amount of communication is greatly reduced $O(1)$, whereas $T_{\text{comp}} = O(N_c)$, where N_c is now the total number of paths derived from the partial path.

The traveling salesman problem (TSP) (cont.)

If we take partial paths of N_p vertices then we have $N_p!$ partial paths. As there are $N_v!$ complete paths, we deduce that there are $N_c = N_v! / N_p!$ complete paths per each partial path. So, N_p **small** means **large** chunks (large synchronization time) sincronización) and for N_p close to N_v we have **small** chunks (large communication time).

For instance, if $N_v = 10$ ($10! = 3,628,800$ complete path) and $N_p = 8$ we will have $N_c = 10! / 8! = 90$ partial paths, whereas if we take $N_p = 3$ then the size of the chunk will be $N_c = 10! / 3! = 604,800$.

The traveling salesman problem (TSP) (cont.)

The algorithm to generate complete paths derived from a partial path is identical to the presented one. We have only to put the partial path at the beginning of `path[]` and then complete to the right with the remaining vertices, moving to the left all vertices from $N - 1$ until N_p .

```

1  int next_path(int *path, int N, int Np=0) {
2      for (int j=N-1; j>=Np; j--) {
3          if (path[0]==j) {
4              for (int k=0; k<j; k++) path[k] = path[k+1];
5              path[j] = j;
6          } else {
7              int k;
8              for (k=0; k<N-1; k++)
9                  if (path[k]==j) break;
10             path[k] = path[k-1];
11             path[k-1] = j;
12             return 1;
13         }
14     }
15     return 0;
16 }
```

The traveling salesman problem (TSP) (cont.)

In order to generate the partial paths, it suffices to call `next_path(...)` with $N_v = N_p$, since in that case the function will move only the first N_p positions. The following algorithm runs over all paths

```
1  for (int j=0; j<Nv; j++) ppath[j]=j;
2  while(1) {
3      memcpy(path,ppath,Nv*sizeof(int));
4      while (1) {
5          // do something with complete path . . . .
6          if(!next_path(path,Nv,Np)) break;
7      }
8      if(!next_path(ppath,Np)) break;
9  }
10
11 }
```

The traveling salesman problem (TSP) (cont.)

For instance the following code prints all partial paths and their derived complete paths.

```
1  for (int j=0; j<Nv; j++) ppath[j]=j;
2  while(1) {
3      printf("partial path: ");
4      print_path(ppath, Np);
5      memcpy(path, ppath, Nv*sizeof(int));
6      while (1) {
7          printf("complete path: ");
8          print_path(path, Nv);
9          if(!next_path(path, Nv, Np)) break;
10     }
11     if(!next_path(ppath, Np)) break;
12 }
```

The traveling salesman problem (TSP) (cont.)

Pseudocode (master code):

```
1  if (!myrank) {
2      int done=0, down=0;
3      // Initialize partial path
4      for (int j=0; j<N; j++) path[j]=j;
5      while(1) {
6          // Receive work done by 'slave' with 'MPI_ANY_SOURCE' ...
7          int slave = status.MPI_SOURCE;
8          // Send new next partial path
9          MPI_Send(path,N,MPI_INT,slave,0,MPI_COMM_WORLD);
10         if (done) down++;
11         if (down==size-1) break;
12         if(!done && !next_path(path,Np)) {
13             done = 1;
14             path[0] = -1;
15         }
16     }
17 } else {
18     // slave code ...
19 }
```

The traveling salesman problem (TSP) (cont.)

Pseudocode (slave code):

```
1  if (!myrank) {
2    // master code ...
3  } else {
4    while (1) {
5      // Send work to master (initially garbage) ...
6      MPI_Send(. . . , 0, 0, MPI_COMM_WORLD);
7      // Receive next partial path
8      MPI_Recv(path, N, MPI_INT, 0, MPI_ANY_TAG,
9               MPI_COMM_WORLD, &status);
10     if (path[0]==-1) break;
11     while (1) {
12       // do something with path ...
13       if(!next_path(path, N, Np)) break;
14     }
15   }
16 }
```

OPTIONAL Assignment Nbr. 3

Point-to-point communication and dynamic load balance.

1. Write a sequential program that finds the path of minimal length for the **Traveling Salesman Problem** (TSP) doing an **exhaustive** sweep over all the possible paths.
2. Implement the following optimizations:
 - Make all paths start at the same vertex. (invariance under rotation of the path). *Hint:* Advance only the first $Nv - 1$ vertices. In this way the last vertex remains always in the same (last) position.
 - If a partial path has a length larger than the current minimum, then there is no need to compute the derived paths. (We assume that the graph is **convex**, i.e. $d(i, j) \leq d(i, k) + d(k, j)$, $\forall k$). This happens naturally when the graph is generated by taking points in \mathbb{R}^n).
 - Given a path and its inverse (e.g. $(0,3,2,5,4) \rightarrow (4,5,2,3,0)$), we have to check only one of them, since the other has the same length (*Hint:* See **Path Parity** below.)
3. Write a parallel version with a static distribution of work among processors (static balance). Simulate disbalance by launching several

processes to the same processor (*oversubscription*), Note that the *work* to be sent to the processors can be sent as a partial path.

4. Write a parallel version with dynamic load distribution (*compute-on-demand*).
5. Perform a theoretical analysis of the scalability of the problem. Determine experimentally which is the best partial path length to be sent to the slaves.

Path parity

Given a path and its inverse (e.g. (0,3,2,5,4) and (4,5,2,3,0)), only one of them has to be checked since their lengths are the same. In order to do this we can define a **parity function** $\text{prty}(p)$, where p is a path such that if p' is the inverse of p then $\text{prty}(p') = -\text{prty}(p)$. One possibility is to find a particular vertex, for instance 0 and check if the next vertex (in cyclic sense) is greater (parity +1) or smaller (parity -1) than the previous. For instance in the previous example $\text{prty}(03254) = -1$ since 3 is smaller than 4, while $\text{prty}(45230) = +1$ since 4 is greater than 3. Now, this requires to look for a certain vertex, so that the cost can be comparable to compute the length of the path, and it is not certain whether this represents a gain or not. But combining with the previous optimization of leaving a fixed vertex in position $N_v - 1$, we have only to check the vertices in positions 0 and $N_v - 2$.

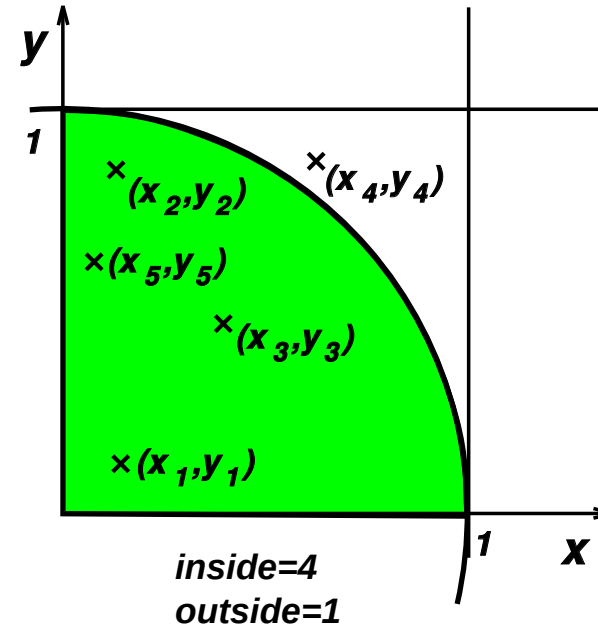
Computing PI by Montecarlo

Computing PI by Montecarlo

Generate *random points* (x_j, y_j) . The probability to fall in the unit circle is $\pi/4$.

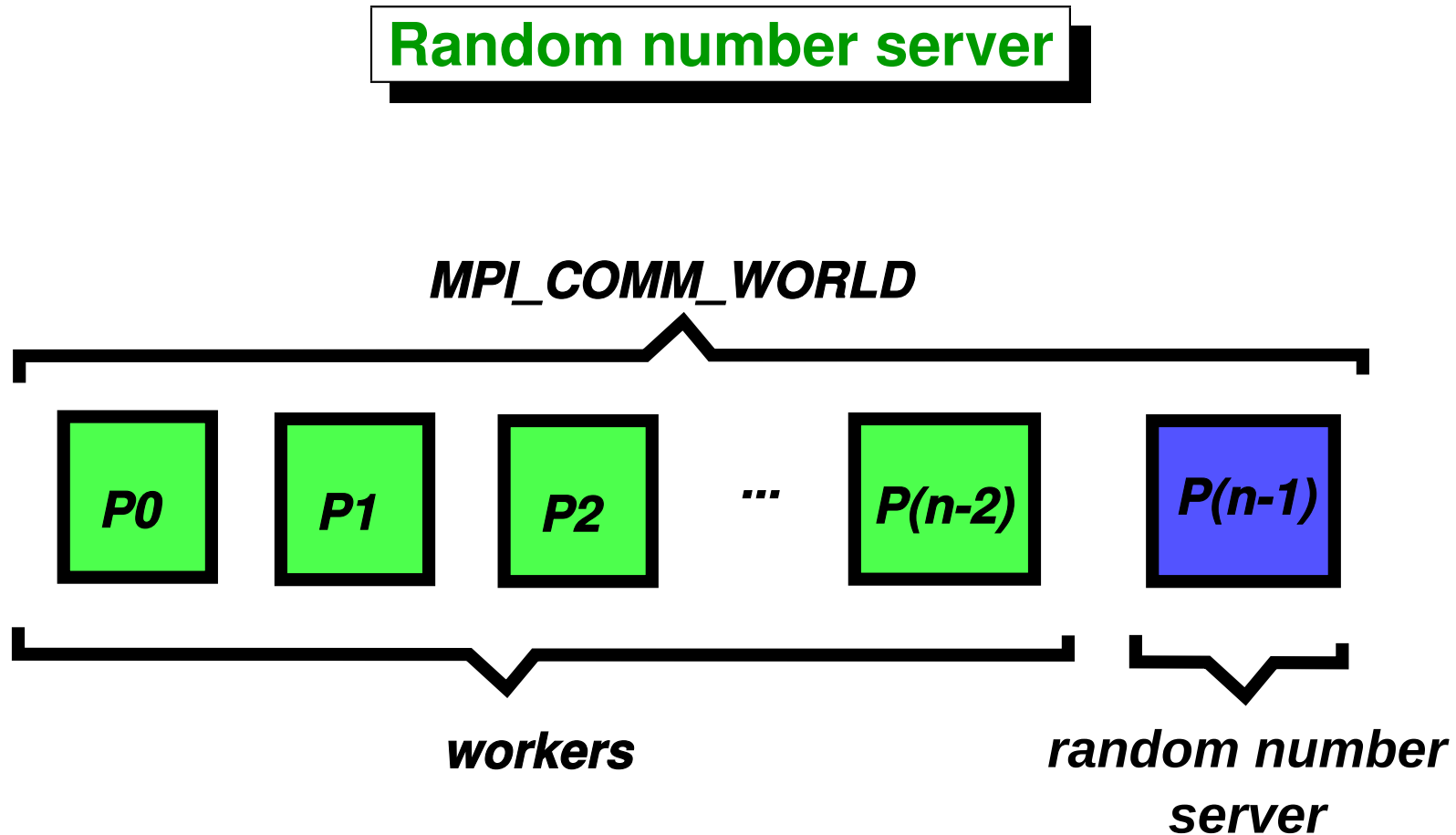
$$\pi = 4 \frac{(\text{\#inside})}{(\text{\#total})}$$

Montecarlo methods are *very easily parallelizable*, but *not deterministic* and exhibit a *slow rate of convergence* ($O(\sqrt{N})$).



Generation of random parallel points

- If all processes generate random numbers without *randomizing* the *seed*, (usually this is done with *srand(time(NULL))*), then the generated sequence will be the same in all processors and in fact there is no gain in processing in parallel.
- If each generator is randomized, then we should guarantee that the time is different in each process. This is normally so, if we use a high precision calendar function (like *gettimeofday()*) due to small delays different in each node. This is not the case for *time()* which is precise to seconds.
- The solution is to appoint some node as a *random number server*, i.e. a node that is dedicated to the generation of sequences of random numbers.



```

1  #include <math.h>
2  #include <limits.h>
3  #include <mpi.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  /* no. of random nos. to generate at one time */
7  #define CHUNKSIZE 1000
8
9  /* message tags */
10 #define REQUEST 1
11 #define REPLY 2
12
13 main(int argc, char **argv) {
14     int iter;
15     int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
16     double x, y, Pi, error, epsilon;
17     int numprocs, myid, server, totalin, totalout, workerid;
18     int rands[CHUNKSIZE], request;
19
20     MPI_Comm world, workers;
21     MPI_Group world_group, worker_group;
22     MPI_Status stat;
23
24     MPI_Init(&argc, &argv);
25     world = MPI_COMM_WORLD;
26     MPI_Comm_size(world, &numprocs);
27     MPI_Comm_rank(world, &myid);
28     server = numprocs-1;
29
30     if (numprocs==1)
31         printf("Error. At least 2 nodes are needed");
32

```

```

33  /* process 0 reads epsilon from args and broadcasts it to
34  everyone */
35  if (myid == 0) {
36      if (argc < 2) {
37          epsilon = 1e-2;
38      } else {
39          sscanf ( argv[1], "%lf", &epsilon );
40      }
41  }
42  MPI_Bcast ( &epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
43
44  /* define the workers communicator by using groups and
45  excluding the server from the group of the whole world */
46  MPI_Comm_group ( world, &world_group );
47  ranks[0] = server;
48  MPI_Group_excl ( world_group, 1, ranks, &worker_group );
49  MPI_Comm_create ( world, worker_group, &workers);
50  MPI_Group_free ( &worker_group);
51
52  /* the random number server code - receives a non-zero
53  request, generates random numbers into the array rands,
54  and passes them back to the process who sent the
55  request. */
56  if ( myid == server ) {
57      do {
58          MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
59                  REQUEST, world, &stat);
60          if ( request ) {
61              for (i=0; i<CHUNKSIZE; i++) rands[i] = random();
62              MPI_Send(rands, CHUNKSIZE, MPI_INT,
63                      stat.MPI_SOURCE, REPLY, world);

```



```

64     }
65     } while ( request > 0 );
66     /* the code for the worker processes - each one sends a
67     request for random numbers from the server, receives
68     and processes them, until done */
69 } else {
70     request = 1;
71     done = in = out = 0;
72     max = INT_MAX;
73
74     /* send first request for random numbers */
75     MPI_Send( &request, 1, MPI_INT, server, REQUEST, world );
76
77     /* all workers get a rank within the worker group */
78     MPI_Comm_rank ( workers, &workerid );
79
80     iter = 0;
81     while (!done) {
82         iter++;
83         request = 1;
84
85         /* receive the chunk of random numbers */
86         MPI_Recv(rands, CHUNKSIZE, MPI_INT, server,
87                 REPLY, world, &stat );
88         for (i=0; i<CHUNKSIZE; ) {
89             x = (((double) rands[i++])/max) * 2 - 1;
90             y = (((double) rands[i++])/max) * 2 - 1;
91             if (x*x + y*y < 1.0)
92                 in++;
93             else
94                 out++;
95         }
96         /* the value of in is sent to the variable totalin in

```

```

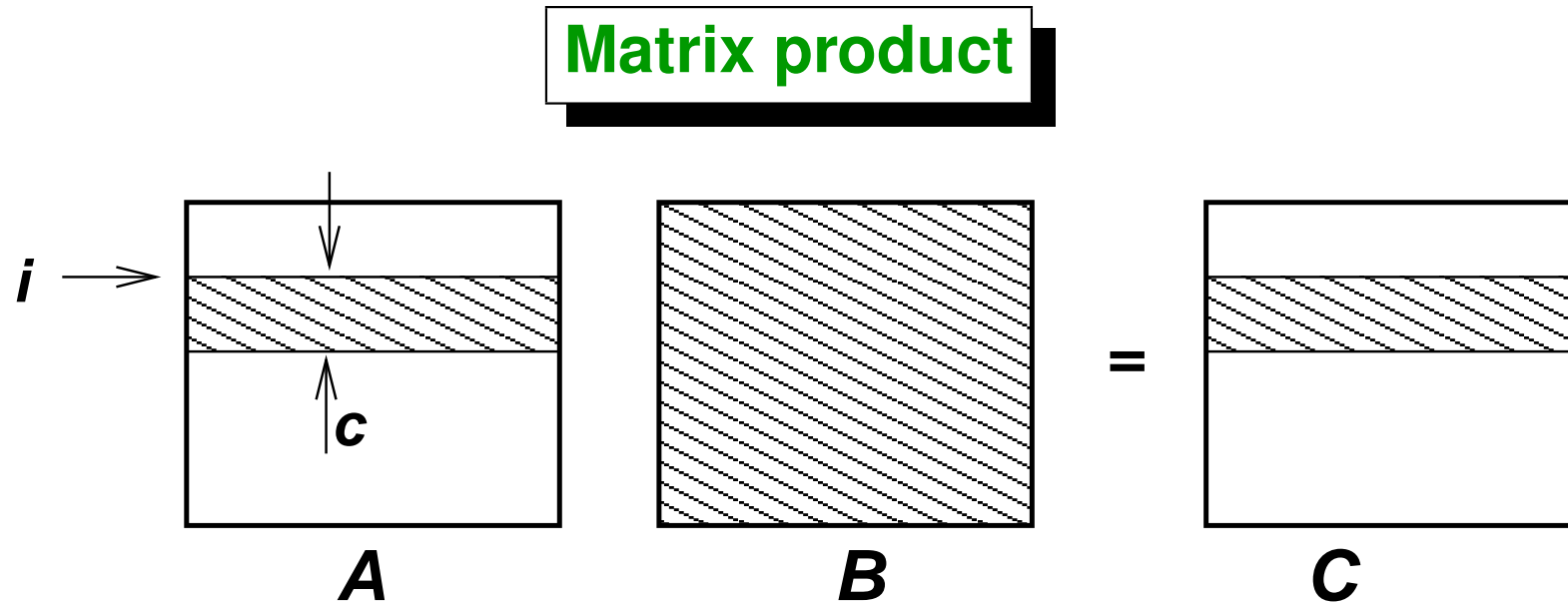
97      all processes in the workers group */
98      MPI_Allreduce(&in, &totalin, 1,
99                  MPI_INT, MPI_SUM, workers);
100     MPI_Allreduce(&out, &totalout, 1,
101                 MPI_INT, MPI_SUM, workers);
102     Pi = (4.0*totalin)/(totalin + totalout);
103     error = fabs ( Pi - M_PI);
104     done = ((error < epsilon) || ((totalin+totalout)>1000000));
105     request = (done) ? 0 : 1;
106
107     /* if done, process 0 sends a request of 0 to stop the
108     rand server, otherwise, everyone requests more
109     random numbers. */
110     if (myid == 0) {
111         printf("pi = %23.20lf\n", Pi );
112         MPI_Send( &request, 1, MPI_INT, server, REQUEST, world);
113     } else {
114         if (request)
115             MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
116     }
117 }
118 }
119 if (myid == 0)
120     printf("total %d, in %d, out %d\n",
121           totalin+totalout, totalin, totalout );
122 if (myid<server) MPI_Comm_free(&workers);
123 MPI_Finalize();
124 }

```

Creating a communicator

```
1  /* define the workers communicator by  
2    using groups and excluding the  
3    server from the group of the whole world */  
4  
5  MPI_Comm world, workers;  
6  MPI_Group world_group, worker_group;  
7  MPI_Status stat;  
8  
9  // ...  
10  
11 MPI_Comm_group(MPI_COMM_WORLD, &world_group);  
12 ranks[0] = server;  
13 MPI_Group_excl(world_group, 1, ranks, &worker_group);  
14 MPI_Comm_create(world, worker_group, &workers);  
15 MPI_Group_free(&worker_group);
```

Example: matrix product in parallel



All nodes have all **B**, and receive part of **A** (a range (chunk) of files $A(i:i+n-1, :)$). The node makes the product $A(i:i+n-1, :) * B$ and returns the result.

- Static load balance: needs to know the computing speed.
- Dynamic load balance: each node **asks** the server for a certain amount of work and once computed returns the result and asks for more work.

Simple matrix library

- Allows to construct a two index matrix (*rank=2*) from a container internal or external to the object.

```
1 // El almacenamiento interno
2 // se borra con el destructor
3 Mat A(100,100);
4 // usar 'A' ...
5
6
7 // El almacenamiento externo
8 // debe ser borrado por el usuario
9 double *b = new double[10000];
10 Mat B(100,100,b);
11 // usar 'B' ...
12 delete[] b;
```

Simple matrix library (cont.)

- Overloads operator `()` so that it can be used at the left hand side of an assignement operation.

```
1 double x, w;  
2 x = A(i, j);  
3 A(j, k) = w;
```

- As usual in C, multidimensional arrays are *stored by row*. In class *Mat* one can access the storing area (internal or external) , as a standard C array.

Also, we can access individual rows or a range of adjacent rows.

```
1 Mat A(100, 100);  
2 double *a, *a10;  
3 // Puntero al area de almacenamiento interno  
4 a = &A(0, 0);  
5 // Puntero al area de almacenamiento a partir de  
6 // la linea 10 (base 0)  
7 a10 = &A(10, 0);
```

Simple matrix library (cont.)

```
1  /// Simple matrix class
2  class Mat {
3  private:
4      /// The store
5      double *a;
6      /// row and column dimensions, flag if the store
7      /// is external or not
8      int m,n,a_is_external;
9  public:
10     /// Constructor from row and column dimensions
11     /// and (eventually) the external store
12     Mat(int m_,int n_,double *a_=NULL);
13     /// Destructor
14     ~Mat();
15     /// returns a reference to an element
16     double & operator()(int i,int j);
17     /// returns a const reference to an element
18     const double & operator()(int i,int j) const;
19     /// product of matrices
20     void matmul(const Mat &a, const Mat &b);
21     /// prints the matrix
22     void print() const;
23 };
```


Dynamic balance

Abstraction: there is a series of data x_1, \dots, x_N and results r_1, \dots, r_N and P processors. We assume that N is large enough, and $1 \leq \text{chunksize} \ll N$ so that the associated overhead with sending and receiving data is negligible.

Slave code

- Sends a message of $\text{len}=0$ that means *I'm ready*.
- Receives an amount of tasks of length *len*. If $\text{len} > 0$, process them and sends back the results.
If $\text{len}=0$, sends a $\text{len}=0$ acknowledge message, exits the loop and ends processing.

```
1 Send len=0 to server
2 while(true) {
3     Receive len, tag=k
4     if (len=0) break;
5     Process x_k, x_{k+1}, . . . , x_{k+len-1} ->
6         r_k, r_{k+1}, . . . , r_{k+len-1}
7     Send r_k, r_{k+1}, . . . , r_{k+len-1} to server
8 }
9 Send len=0 to server
```

Server code

Keeps a vector of states for each slave, may be: 0 = Not started yet, 1 = Working, 2 = Stopped.

```
1  proc_status[1..P-1] =0;
2
3  j=0;
4  while (true) {
5      Receive len,tag->k, proc
6      if (len>0) {
7          extrae r_k,...,r_{k+len-1}
8      } else {
9          proc_status[proc]++;
10     }
11     jj = min(N,j+chunksize);
12     len = jj-j;
13     Send x_j,x_{j+1}, x_{j+len-1}, tag=j to proc
14     // Verifica si todos terminaron
15     Si proc_status[proc] == 2 para proc=1..P-1 then break;
16 }
```

Slave code, use local processor

```
1 proc_status[1..P-1] =0;
2 j=0
3 while (true) {
4     Immediate Receive len,tag->k, proc
5     while(j<N) {
6         if (received) break;
7         Process x_j -> r_j
8         j++
9     }
10    Wait until received;
11    if (len==0) {
12        extrae r_k, . . . , r_{k+len-1}
13    } else {
14        proc_status[proc]++;
15    }
16    jj = min(N, j+chunksize);
17    len = jj-j;
18    Send x_j, x_{j+1}, x_{j+len-1}, tag=j to proc
19    // Verifica si todos terminaron
20    Si proc_status[proc] == 2 para proc=1..P-1 then break;
21 }
```

Complete code

```

1  #include <time.h>
2  #include <unistd.h>
3  #include <ctype.h>
4  #include "mat.h"
5
6  /* Self-scheduling algorithm.
7     Computes the product C=A*B; A,B and C matrices of NxN
8     Proc 0 is the root and sends work to the slaves
9  */
10 // $Id: matmult2.cpp,v 1.1 2004/08/28 23:05:28 mstorti Exp $
11
12 /** Computes the elapsed time between two instants captured with
13     'gettimeofday'.
14 */
15 double etime(timeval &x,timeval &y) {
16     return double(y.tv_sec)-double(x.tv_sec)
17         + (double(y.tv_usec)-double(x.tv_usec))/1e6;
18 }
19
20 /** Computes random numbers between 0 and 1
21 */
22 double drand() {
23     return ((double)(rand()))/((double)(RAND_MAX));
24 }
25
26 /** Computes an integer random number in the
27     range 'imin'-'imax'
28 */

```

```
29 int irand(int imin, int imax) {
30     return int(rint(drand()*double(imax-imin+1)-0.5))+imin;
31 }
32
33
34 int main(int argc, char **argv) {
35     /// Initializes MPI
36     MPI_Init(&argc, &argv);
37     /// Initializes random
38     srand(time(0));
39
40     /// Size of problem, size of chunks (in rows)
41     int N, chunksize;
42     /// root procesoor, number of processor, my rank
43     int root=0, numprocs, rank;
44     /// time related quantities
45     struct timeval start, end;
46
47     /// Status for retrieving MPI info
48     MPI_Status stat;
49     /// For non-blocking communications
50     MPI_Request request;
51     /// number of processors and my rank
52     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
53     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
54
55     /// cursor to the next row to send
56     int row_start=0;
57     /// Read input data from options
58     int c, print_mat=0, random_mat=0, use_local_processor=0,
59         slow_down=1;
60 }
```

```

61 while ((c = getopt (argc, argv, "N:c:prhls:")) != -1) {
62     switch (c) {
63     case 'h':
64         if (rank==0) {
65             printf(" usage: $ mpirun [OPTIONS] matmult2\n\n"
66                 "MPI options: -np <number-of-processors>\n"
67                 "                -machinefile <machine-file>\n\n"
68                 "Other options: -N <size-of-matrix>\n"
69                 "                -c <chunk-size> "
70                 "# sets number of rows sent to processors\n"
71                 "                -p "
72                 "# print input and result matrices to output\n"
73                 "                -r "
74                 "# randomize input matrices\n");
75         }
76         exit(0);
77     case 'N':
78         sscanf(optarg, "%d", &N);
79         break;
80     case 'c':
81         sscanf(optarg, "%d", &chunksize);
82         break;
83     case 'p':
84         print_mat=1;
85         break;
86     case 'r':
87         random_mat=1;
88         break;
89     case 'l':
90         use_local_processor=1;
91         break;
92     case 's':
93         sscanf(optarg, "%d", &slow_down);

```

```

94     if (slow_down<1) {
95         if (rank==0) printf("slow_down factor (-s option) "
96                             " must be >=1. Reset to 1!\n");
97         abort();
98     }
99     break;
100 case '?':
101     if (isprint (optopt))
102         fprintf (stderr, "Unknown option '-%c'.\n", optopt);
103     else
104         fprintf (stderr,
105                 "Unknown option character '\\x%x'.\n",
106                 optopt);
107     return 1;
108 default:
109     abort ();
110 }
111 }
112
113 #if 0
114 if ( rank == root ) {
115     printf("enter N, chunksize > ");
116     scanf ("%d", &N);
117     scanf ("%d", &chunksize);
118     printf ("\n");
119 }
120 #endif
121
122 /// Register time for statistics
123 gettimeofday (&start, NULL);
124 /// broadcast N and chunksize to other procs.
125 MPI_Bcast (&N, 1, MPI_INT, 0, MPI_COMM_WORLD );
126 MPI_Bcast (&chunksize, 1, MPI_INT, 0, MPI_COMM_WORLD );

```



```

127
128 /// Define matrices, bufa and bufc is used to
129 /// send matrices to other processors
130 Mat b(N,N), bufa(chunksize,N), bufc(chunksize,N);
131
132 ///-----<*>-----<*>-----<*>-----<*>-----<*>
133 ///-----<*>----- SERVER PART *>-----<*>
134 ///-----<*>-----<*>-----<*>-----<*>-----<*>
135 if ( rank == root ) {
136     Mat a(N,N), c(N,N);
137
138     /// Initialize 'a' and 'b'
139     for (int j=0; j<N; j++) {
140         for (int k=0; k<N; k++) {
141             /// random initialization
142             if (random_mat) {
143                 a(j,k) = floor(double(rand())/double(INT_MAX)*4);
144                 b(j,k) = floor(double(rand())/double(INT_MAX)*4);
145             } else {
146                 /// integer index initialization (eg 00 01 02 . . . . NN)
147                 a(j,k) = &a(j,k)-&a(0,0);
148                 b(j,k) = a(j,k)+N*N;
149             }
150         }
151     }
152 }
153
154 /// proc_status[proc] = 0 -> processor not contacted
155 ///                      = 1 -> processor is working
156 ///                      = 2 -> processor has finished
157 int *proc_status = (int *) malloc(sizeof(int)*numprocs);
158 /// statistics[proc] number of rows that have been processed
159 /// by this processor

```

```

160  int *statistics = (int *) malloc(sizeof(int)*numprocs);
161
162  /// initialize proc_status, statistics
163  for (int proc=1; proc<numprocs; proc++) {
164      proc_status[proc] = 0;
165      statistics [proc] = 0;
166  }
167
168  /// send b to all processor
169  MPI_Bcast (&b(0,0), N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD );
170
171  /// Register time for statistics
172  gettimeofday (&end, NULL);
173  double bcast = etime(start, end);
174
175  while (1) {
176
177      /// non blocking receive
178      if (numprocs>1) MPI_Irecv(&bufc(0,0), chunksize*N,
179                               MPI_DOUBLE, MPI_ANY_SOURCE,
180                               MPI_ANY_TAG,
181                               MPI_COMM_WORLD, &request);
182
183      /// While waiting for results from slaves server works ...
184      int receive_OK=0;
185      while (use_local_processor && row_start<N) {
186          /// Test if some message is waiting.
187          if (numprocs>1) MPI_Test(&request, &receive_OK, &stat);
188          if(receive_OK) break;
189          /// Otherwise... work
190          /// Local masks
191          Mat aa(1, N, &a(row_start, 0)), cc(1, N, &c(row_start, 0));

```

```
192     /// make product
193     for (int jj=0; jj<slow_down; jj++)
194         cc.matmul(aa,b);
195     /// increase cursor
196     row_start++;
197     /// register work
198     statistics[0]++;
199 }
200
201 /// If no more rows to process wait
202 // until message is received
203 if (numprocs>1) {
204     if (!receive_OK) MPI_Wait(&request,&stat);
205
206     /// length of received message
207     int rcvlen;
208     /// processor that sent the result
209     int proc = stat.MPI_SOURCE;
210     MPI_Get_count(&stat,MPI_DOUBLE,&rcvlen);
211
212     if (rcvlen!=0) {
213         /// Store result in 'c'
214         int rcv_row_start = stat.MPI_TAG;
215         int nrows_sent = rcvlen/N;
216         for (int j=rcv_row_start;
217             j<rcv_row_start+nrows_sent; j++) {
218             for (int k=0; k<N; k++) {
219                 c(j,k) = bufc(j-rcv_row_start,k);
220             }
221         }
222     } else {
```

```
223      /// Zero length messages are used
224      /// to acknowledge start work
225      /// and finished work. Increase
226      /// status of processor.
227      proc_status[proc]++;
228  }
229
230      /// Rows to be sent
231      int row_end = row_start+chunksize;
232      if (row_end>N) row_end = N;
233      int nrows_sent = row_end - row_start;
234
235      /// Increase statistics, send task to slave
236      statistics[proc] += nrows_sent;
237      MPI_Send(&a(row_start, 0), nrows_sent*N, MPI_DOUBLE,
238              proc, row_start, MPI_COMM_WORLD);
239      row_start = row_end;
240  }
241
242      /// If all processors are in state 2,
243      /// then all work is done
244      int done = 1;
245      for (int procc=1; procc<numprocs; procc++) {
246          if (proc_status[procc] != 2) {
247              done = 0;
248              break;
249          }
250      }
251      if (done) break;
252  }
253
254      /// time statistics
```

```

255     gettimeofday (&end, NULL);
256     double dttime = etime(start, end);
257     /// print statistics
258     double dN = double(N), rate;
259     rate = 2*dN*dN*dN/dttime/1e6;
260     printf("broadcast: %f secs [%5.1f %%], process: %f secs\n",
261           "rate: %f Mflops on %d procesors\n",
262           bcast, bcast/dttime*100., dttime-bcast,
263           rate, numprocs);
264     if (slow_down>1) printf("slow_down=%d, scaled Mflops %f\n",
265                             slow_down, rate*slow_down);
266     for (int procc=0; procc<numprocs; procc++) {
267         printf("%d lines processed by %d\n",
268               statistics[procc], procc);
269     }
270
271     if (print_mat) {
272         printf("a: \n");
273         a.print();
274         printf("b: \n");
275         b.print();
276         printf("c: \n");
277         c.print();
278     }
279
280     /// free memory
281     free(statistics);
282     free(proc_status);
283
284 } else {
285
286     //----:---<*>---:---<*>---:---<*>---:---<*>---:---<*>

```

```

287  //---:---<*>---:- SLAVE PART      *>---:---<*>
288  //---:---<*>---:-<*>---:-<*>---:-<*>---:-<*>---:-<*>
289
290  /// get 'b'
291  MPI_Bcast(&b(0,0), N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD );
292  /// Send message zero length 'I'm ready'
293  MPI_Send(&bufc(0,0), 0, MPI_DOUBLE, root, 0, MPI_COMM_WORLD);
294
295  while (1) {
296      /// Receive task
297      MPI_Recv(&bufa(0,0), chunksize*N, MPI_DOUBLE, 0, MPI_ANY_TAG,
298              MPI_COMM_WORLD, &stat);
299
300      int rcvlen;
301      MPI_Get_count(&stat, MPI_DOUBLE, &rcvlen);
302      /// zero length message means: no more rows to process
303      if (rcvlen==0) break;
304
305      /// compute number of rows received
306      int nrows_sent = rcvlen/N;
307      /// index of first row sent
308      int row_start = stat.MPI_TAG;
309      /// local masks
310      Mat bufaa(nrows_sent, N, &bufa(0,0)),
311             bufcc(nrows_sent, N, &bufc(0,0));
312      /// compute product
313      for (int jj=0; jj<slow_down; jj++)
314          bufcc.matmul(bufaa, b);
315      /// send result back
316      MPI_Send(&bufcc(0,0), nrows_sent*N, MPI_DOUBLE,
317              0, row_start, MPI_COMM_WORLD);

```

```
318     }
319
320     /// Work finished. Send acknowledge (zero length message)
321     MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);
322
323 }
324
325 // Finallize MPI
326 MPI_Finalize();
327 }
```

Game of life

Game of life

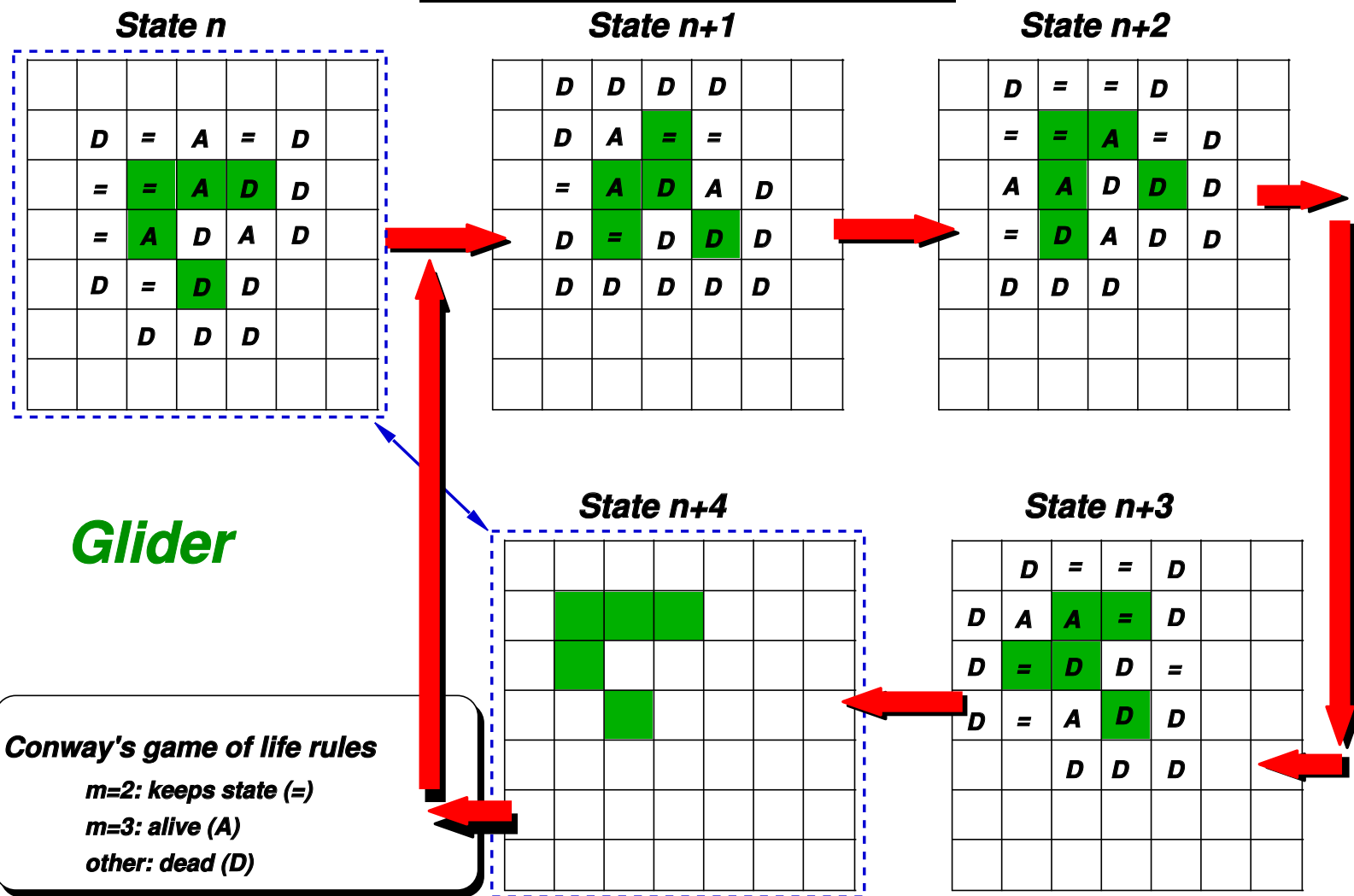
Conway's Game of Life is an example of **cellular automata** with simple evolution rules that leads to complex behavior (**self-organization** and **emergence**).

A board of $N \times N$ cells c_{ij} for $0 \leq i, j < N$, that can be either in **"dead"** or **"alive"** state ($c_{ij}^n = 0, 1$), is advanced from **"generation"** n to the $n + 1$ by the following simple rules. If m is the number of neighbor cells alive in stage n , then in stage $n + 1$ the cell is alive or dead according to the following rules

- If $m = 2$ cell keeps state.
- If $m = 3$ cell becomes alive.
- In any other case cell becomes dead (**"overcrowding"** or **"solitude"**).

	D	=	A	=	D	
	=	=	A	D	D	
	=	A	D	A	D	
	D	=	D	D		
		D	D	D		

Game of life (cont.)



Game of life (cont.)

Formally the rules are as follows. Let

$$a_{ij}^n = \sum_{\mu, \nu = -1, 0, 1} c_{i+\mu, j+\nu}^n - c_{ij}^n$$

be the number of cells neighbor to ij that are alive in generation n . Then the state of cell ij at generation $n + 1$ is

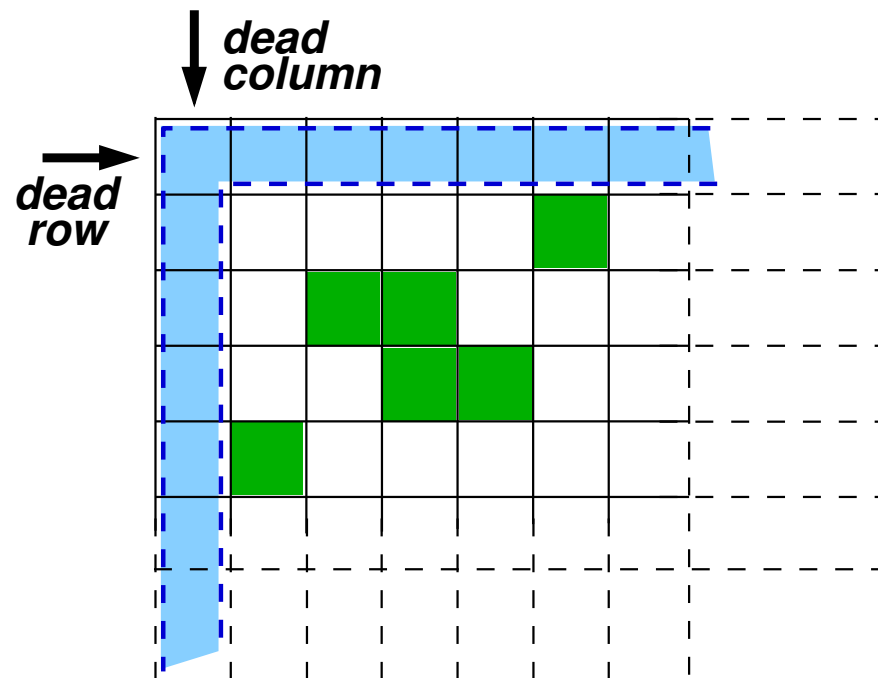
$$a_{ij}^{n+1} = \begin{cases} 1 & ; \text{ if } a_{ij}^n = 3 \\ c_{ij}^n & ; \text{ if } a_{ij}^n = 2 \\ 0 & ; \text{ otherwise.} \end{cases}$$

In some sense these rules tend to mimic the behavior of life forms in the sense that they die if neighbor the population is too large (“**overcrowding**”) or too few (“**solitude**”), and retain their state or become alive in the intermediate cases.

Game of life (cont.)

A layer of dead cells is assumed at each of the boundaries, for the evaluation of the right hand side,

$$c_{ij}^n \equiv 0, \quad \text{if } i, j = -1 \text{ or } N$$



Game of life (cont.)

A class *Board*

```
1 Board board;
2 // Initializes the board. A vector of weights is passed.
3 void board.init(N, weights);
4 // Sets the the state of a cell.
5 void board.setv(i, j, state);
6 // rescatter the board according to new weights
7 void board.re_scatter(new_weights);
8 // Advances the board one generation.
9 void board.advance();
10 // Prints the board
11 void print();
12 // Print statistics about estiamted processing rates
13 void board.statistics();
14 // Reset statistics
15 void board.reset_statistics();
```

Game of life (cont.)

- Cell state is represented as a *char* (8 bits integers). Representations with integers of larger size (e.g. *int*) are more inefficient since we have more data to transfer. Using *vector<bool>* may be more efficient regarding communication but it would have an overload of bit shifting operations to access the relevant bit.
- Each row is stored in an array of *char* such as *char row[N+2]*. The two additional chars are needed for the boundary columns.
- The total board is an array of rows, i.e. *char** board* (private member inside class *Board*). So that *board[j]* is row *j* and *board[j][k]* is the state of cell *j, k*.
- Rows may be exchanged, by swapping pointers for instance to exchange *j* and *k*

```
1 char *tmp = board[j];  
2 board[j] = board[k];  
3 board[k] = tmp;
```

Game of life (cont.)

Rows are distributed among processors according to `vector<double> weights`. First we normalize `weights[]` so that the sum is 1. To each processor we assign `int (N*weights[myrank]+carry)` rows. `carry` is a `double` that accumulates the rounding defect.

```

1  rows_proc.resize(size);
2  double sum_w = 0., carry=0., a;
3  for (int p=0; p<size; p++) sum_w += w[p];
4  int j = 0;
5  double tol = 1e-8;
6  for (int p=0; p<size; p++) {
7      w[p] /= sum_w;
8      a = w[p]*N + carry + tol;
9      rows_proc[p] = int(floor(a));
10     carry = a - rows_proc[p] - tol;
11     if (p==myrank) {
12         i1 = j;
13         i2 = i1 + rows_proc[p];
14     }
15     j += rows_proc[p];
16 }
17 assert (j==N);
18 assert (fabs(carry) < tol);
19

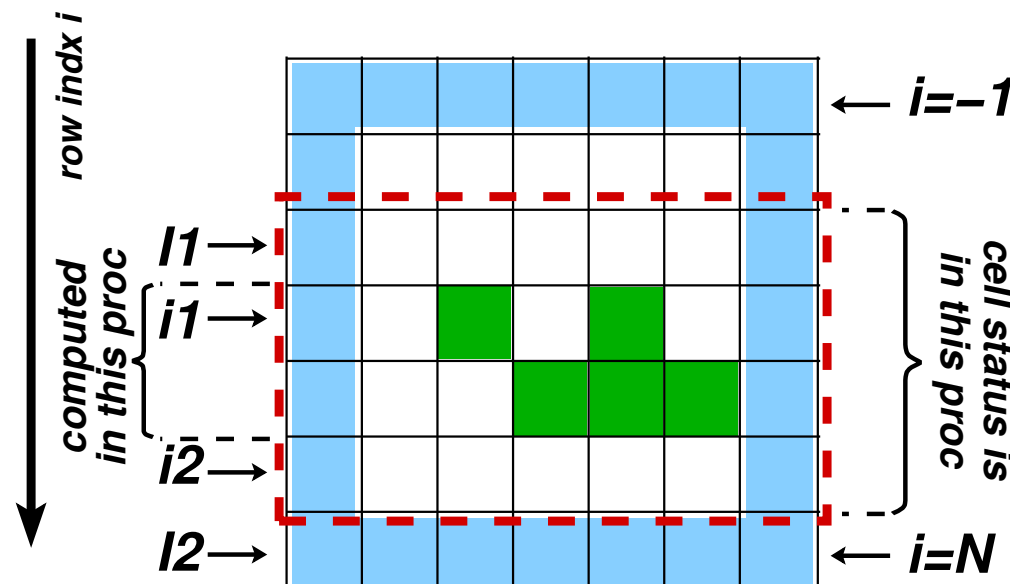
```

```
20  I1=i1-1;  
21  I2=i2+1;
```


Game of life (cont.)

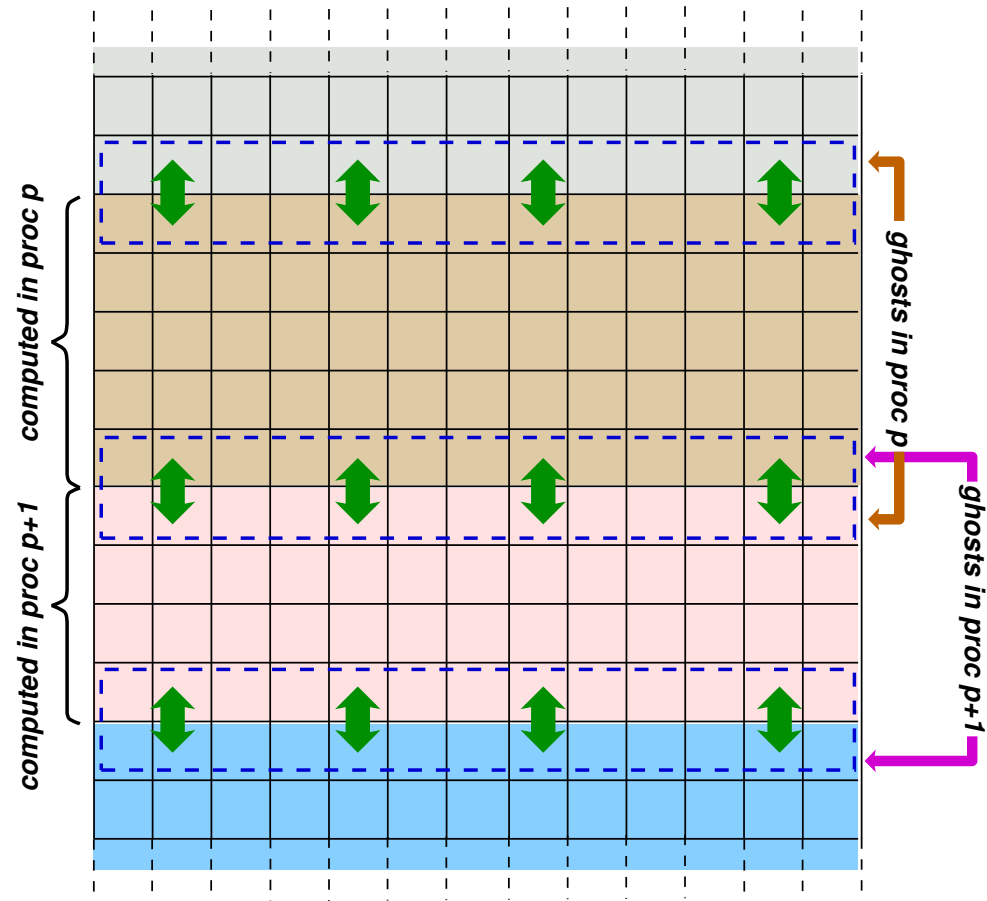
- The range of rows to process in each processor is $[i1, i2)$
- Each processor keeps cell states for rows in a broader range $[I1, I2)$ that contains one more row in each direction, i.e. $I1=i1-1$, $I2=i2+1$.

- Rows -1 and N are **boundary** rows that contain dead cells.



Game of life (cont.)

- *Board::advance()* computes the new cell states from the previous generation.
- First we must perform the *scatter* of *ghost* rows.

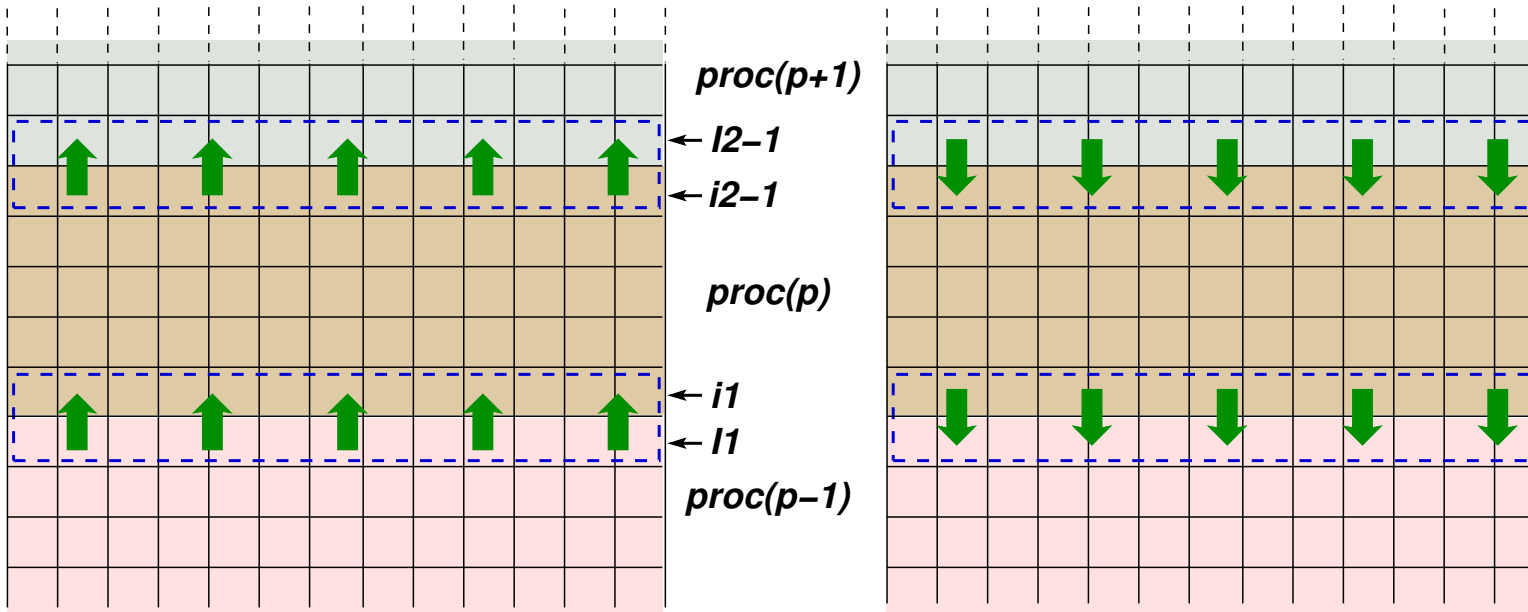


Game of life (cont.)

```

1 if (myrank<size-1) SEND (row(i2-1),myrank+1);
2 if (myrank>0) RECEIVE (row(I1),myrank-1);
3
4 if (myrank>0) SEND (row(i1),myrank-1);
5 if (myrank<size-1) RECEIVE (row(I2),myrank+1);

```

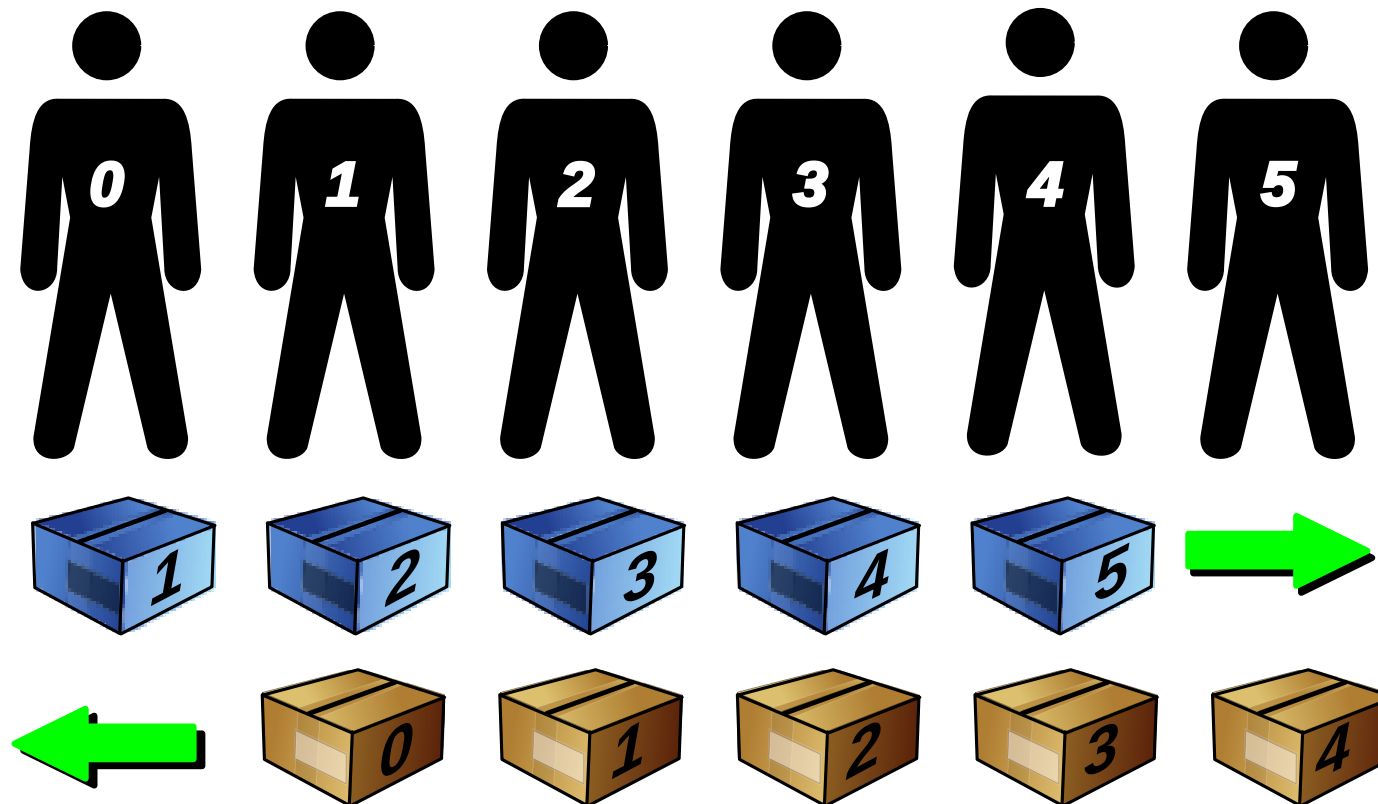
$$proc(p) \longrightarrow proc(p+1)$$
$$proc(p) \longleftarrow proc(p+1)$$


Game of life (cont.)

This kind of communication is not blocking but very slow. In the first stage, when processor p sends his row $i2 - 1$ to $p + 1$ and the receives the $I1$ row from $p - 1$, a delay is produced since the sends and receives are *chained* in sequence. Note that the receive for $p = 0$ only may be completed *after* that the send is completed.

Game of life (cont.)

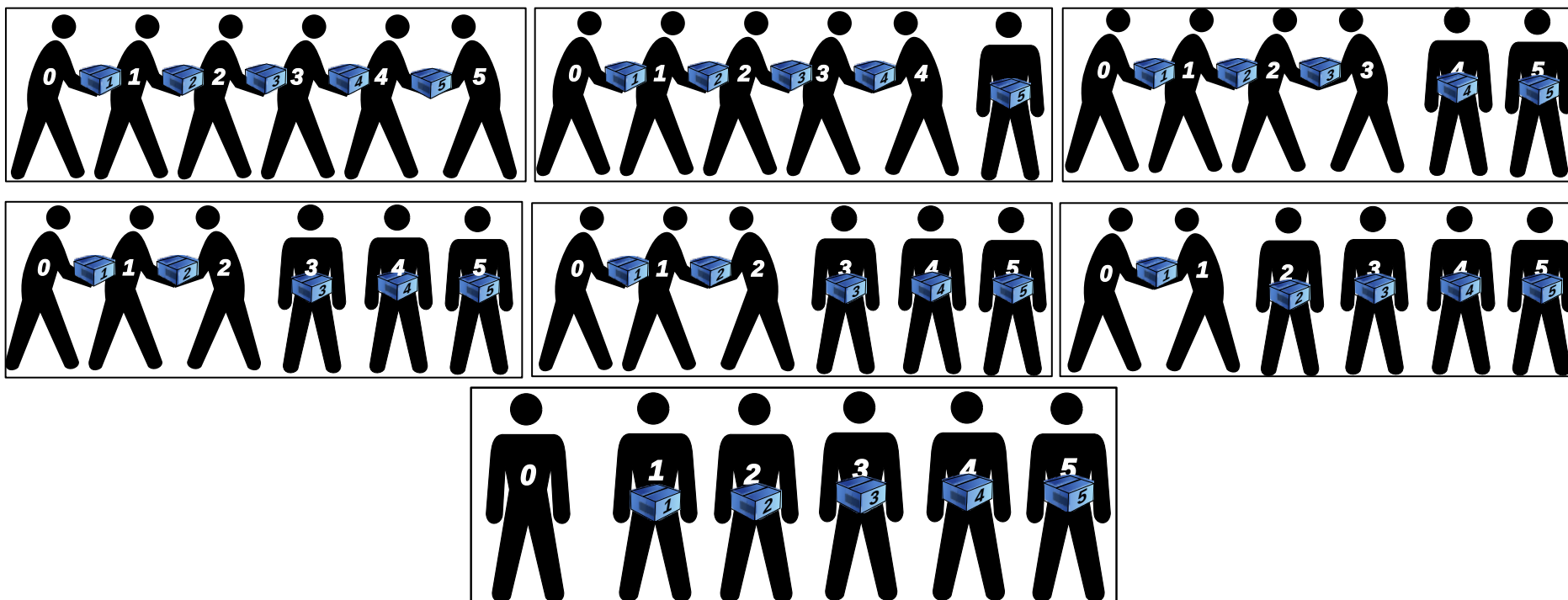
Processors 0-5 must pass the blue boxes to the right and the brown boxes to the left. (The number in the box is the destination process).



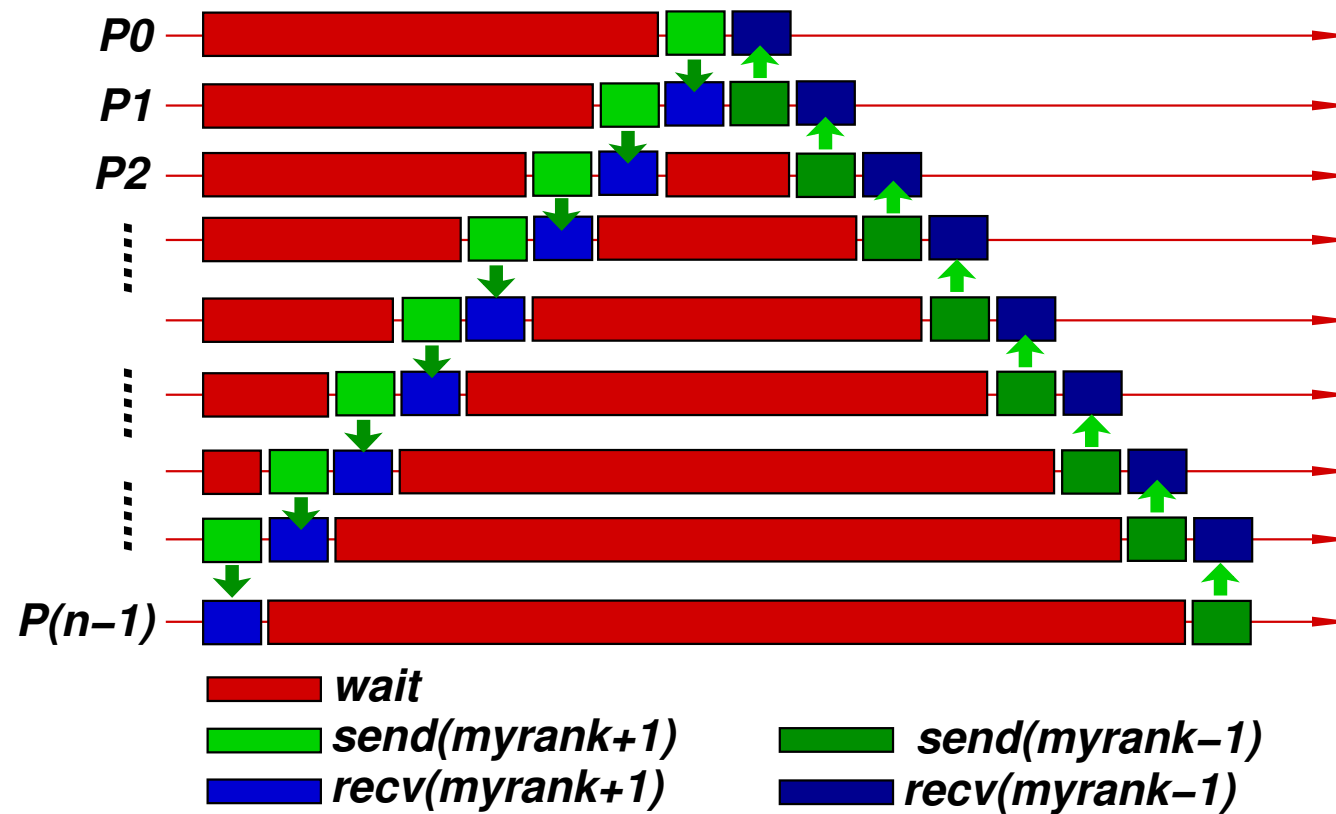
Game of life (cont.)

Communication going right (blue boxes):

```
1 if (myrank<size-1) SEND (. . . ,myrank+1) ;  
2 if (myrank>0) RECEIVE (. . . ,myrank-1) ;
```



Game of life (cont.)

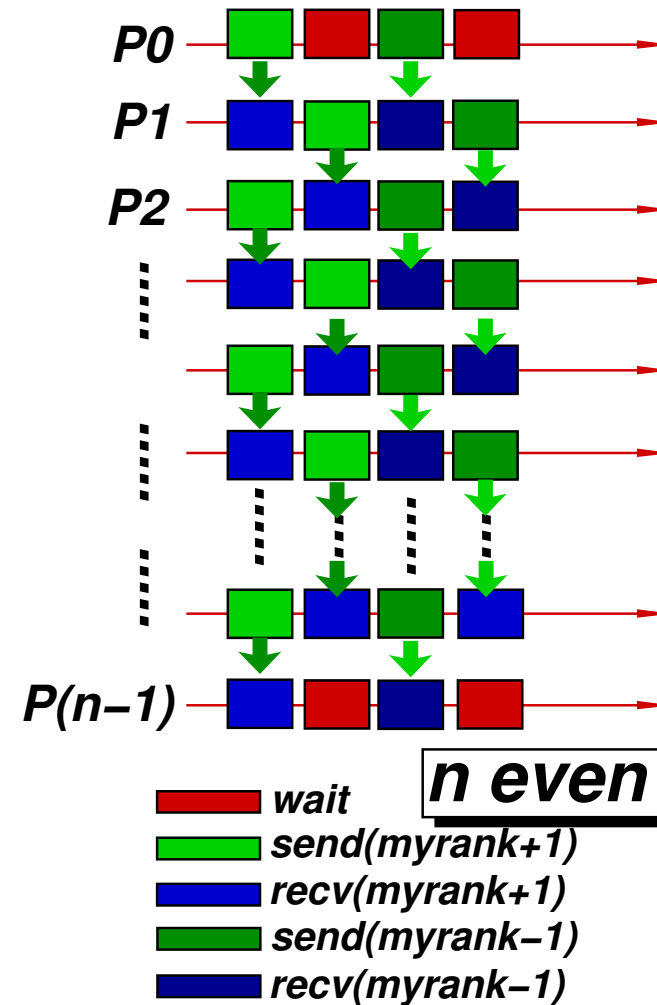


Game of life (cont.)

```

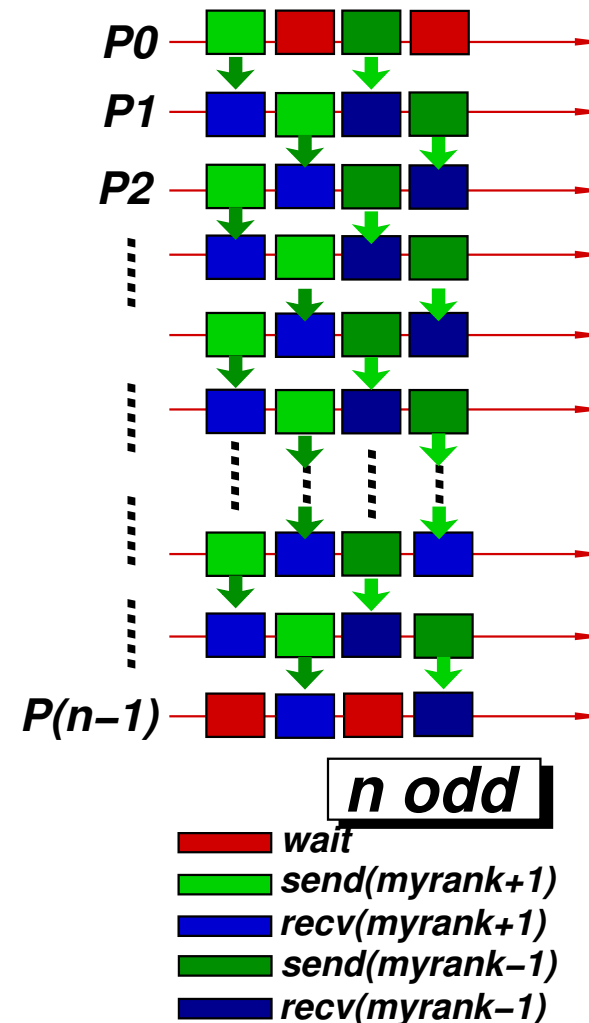
1  if (myrank % 2 == 0 ) {
2    if (myrank < size-1) {
3      SEND (i2-1, myrank+1);
4      RECEIVE (i2-1, myrank+1);
5    }
6  } else {
7    if (myrank > 0) {
8      RECEIVE (i1, myrank-1);
9      SEND (i1, myrank-1);
10   }
11 }
12
13 if (myrank % 2 == 0 ) {
14   if (myrank > 0) {
15     SEND (i1, myrank-1);
16     RECEIVE (i1, myrank-1);
17   }
18 } else {
19   if (myrank < size-1) {
20     RECEIVE (i2-1, myrank+1);
21     SEND (i2-1, myrank+1);
22   }
23 }

```



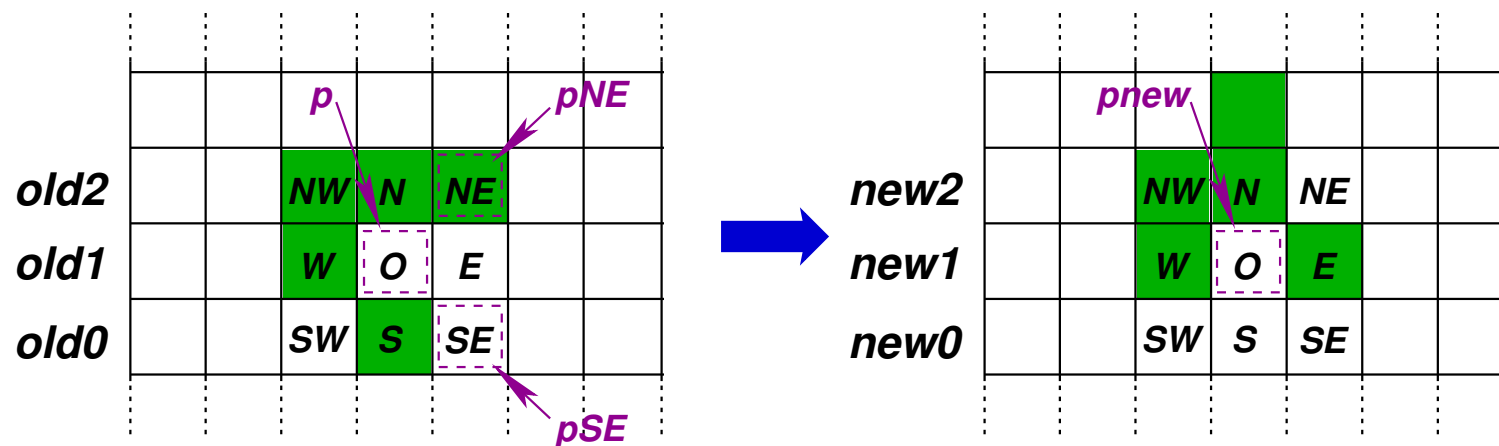
Game of life (cont.)

This communication pattern requires $T = 4\tau$, where τ is the time needed for communicating a cell row, whilst the previous *chained* version requires $T = 2(N_p - 2)\tau$ where N_p is the number of processors. It may be applied also if the number of processors is odd.

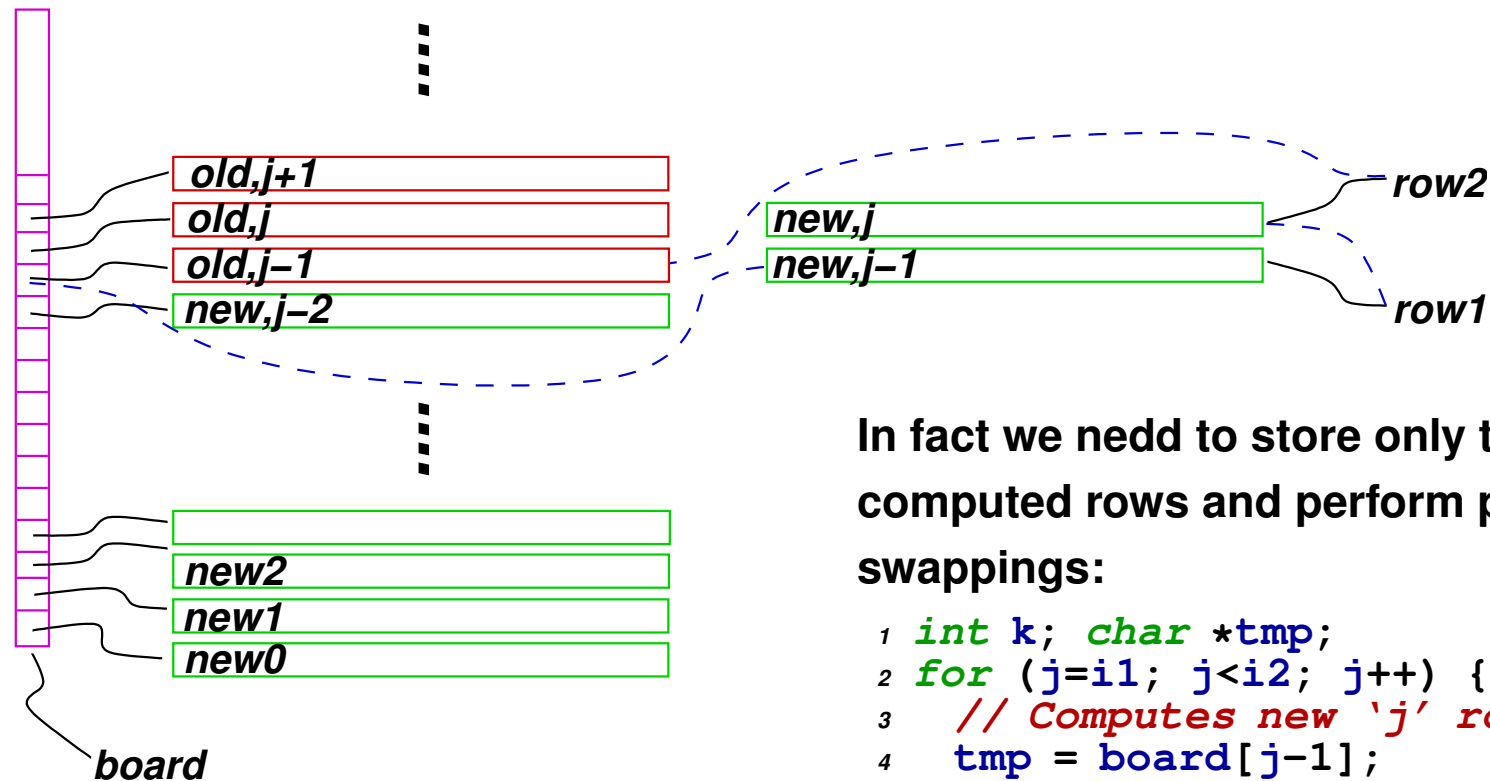


Game of life (cont.)

- The cells are computed by row, from left to right ($W \rightarrow E$) and from top to bottom ($S \rightarrow N$).
- When we compute row j we need rows $j - 1$, j and $j + 1$ at the previous generation, so that we can not overwrite the previous state at row j since it will be needed later for computing row $j + 1$.
- We keep two copies of the board (*new* and *old*) and after we make a swap of the pointers to avoid the copy.



Game of life (cont.)

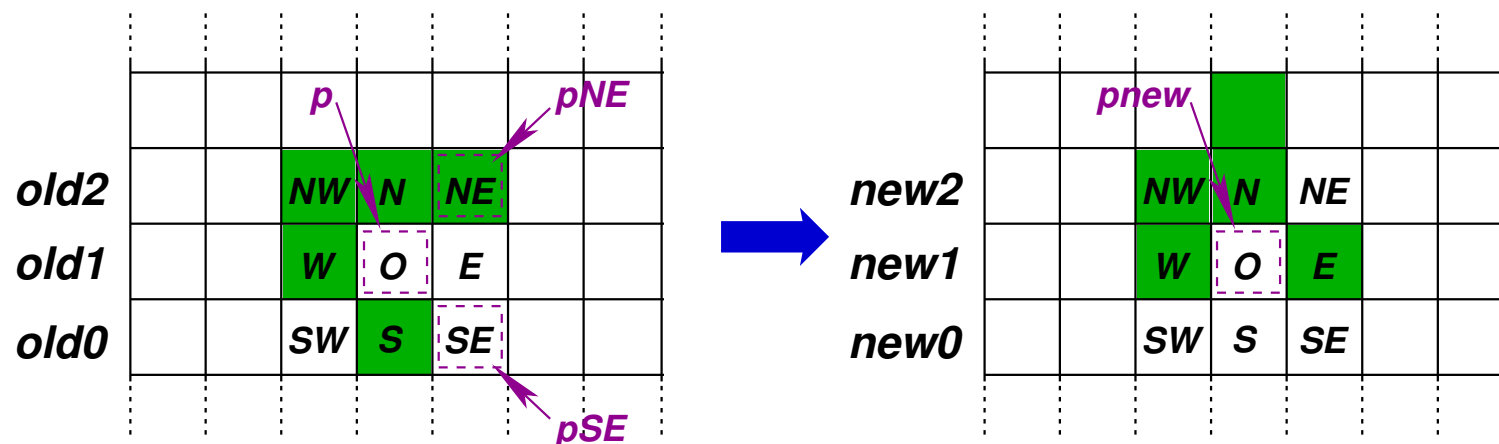


In fact we need to store only the two computed rows and perform pointer swappings:

```

1 int k; char *tmp;
2 for (j=i1; j<i2; j++) {
3     // Computes new 'j' row ...
4     tmp = board[j-1];
5     board[j-1] = row2;
6     row2 = row1;
7     row1 = tmp;
8 }
    
```

Game of life (cont.)



- The **kernel** consists in the function

```
1 void go(int N, char *old0, char *old1,
2         char *old2, char *new1);
```

that computes the new state **new1** from the rows **old0**, **old1** y **old2**. The implementation that is described below has been optimized using pointers **pSE** and **pNE** to cells **SE** and **NE** and pointers **p** and **pnew** to cells **O** in their previous and current state, respectively.

Game of life (cont.)

- Integer variables *ali_W*, *ali_C* and *ali_E* store the counting of live cells in the corresponding columns, i.e.

```
1 int ali_W = SW + W + NW;  
2 int ali_C = NN + O + S;  
3 int ali_E = SE + E + NE;
```

so that the number of live cells is

```
1 alive = ali_W + ali_C + ali_E - O;
```

Each time that we advance a cell to the right the only task to perform is to shift the counters *ali_** to the left and to recompute *ali_E*.

This kernel reaches 115 Mcell/sec in a Pentium 4, 2.4 GHz with DDR memory 400 MHz.

Scalability analysis

If we take a homogeneous system of n processors with computing speeds s (in cells/sec) and a board of $N \times N$ cells, then the computing time in each processor is

$$T_{\text{comp}} = \frac{N^2}{ns}$$

whilst the communication time will be

$$T_{\text{comm}} = \frac{4N}{b}$$

so that we have

$$T_n = T_{\text{comp}} + T_{\text{comm}} = \frac{N^2}{ns} + \frac{4N}{b}$$

Scalability analysis (cont.)

As the time for 1 processor is $T_1 = N^2/s$, the speedup will be

$$S_n = \frac{N^2/s}{N^2/ns + 4N/b}$$

and efficiency will be

$$\eta = \frac{N^2/sn}{N^2/sn + 4N/b} = \frac{1}{1 + 4ns/bN}$$

and we can see that, at least in theory, the algorithm is scalable, i.e. we can keep efficiency bounded from below by while increasing the number of processors **if** we increase at the same time the size of the problem so that $N \propto n$.

In addition we see that, for a certain fixed number of processors n , efficiency may be as close to one as we want by increasing the size of board N .

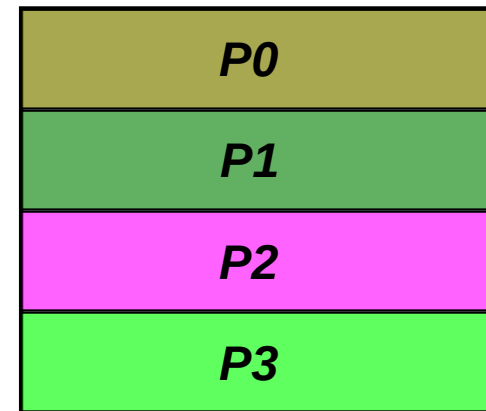
Scalability analysis (cont.)

However, this is not truly scalable, since $N \propto n$ means: $W \propto N^2 \propto n^2$, where W is the work to be done. The solution is to **partition the board in squares**, not in strips. With this partitioning $T_{\text{comm}} = 8N/bn^{0.5}$. (We assume n is a perfect square integer) and the efficiency is

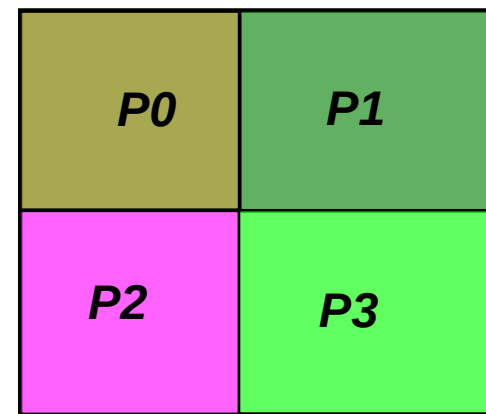
$$\eta = \frac{N^2/sn}{N^2/sn + 8N/bn^{0.5}}$$

$$\frac{1}{1 + 8n^{0.5}s/bN}$$

And so it is enough to keep $N \propto n^{0.5}$ in order to keep efficiency bounded, i.e.
 $W \propto N^2 \propto n$.



partitioning in strips



partitioning in squares

Life with static load balance

As presented so far, the implementation of *life* makes static load balance. We have already seen that an inconvenience of this is that it is not clear a priori which is the *speed* of each processor. However, in this case we can simply run then *life* program sequentially in each node to determine their speeds. For this we make statistics of how many cells computes each processor and how much time it spends, sequentially. Once computed the processor speeds s_i this can be passed to the program so that we assign to each processor a number of cells proportional to s_i .

Life with pseudo-dynamic load balance

The problem with the static balance described previously is that it doesn't perform well if the performance of the processors vary in time. For instance this may happen if we are in a multiuser environment in which other processes are launched. One way to extend this scheme is to redistribute the load every m generations in basis of a statistics of how many cells each processor has computed in the m preceding generations. We must take account of not including in this statistics the communication and synchronization times. m must be chosen carefully, since if m is too small then the redistribution will not be acceptable and in addition there will be a certain overhead associated with the redistribution. On the other hand, if m is too large, then the load variations with characteristic times smaller than the time that takes to compute m generations will be filtered out and not seen by the redistribution processes.

Life with dynamic load balance

One possibility is to perform a dynamic load balance based on the **compute-on-demand** strategy, i.e. each slave asks for work and receives a certain range (**chunk**) of rows. N_c . The slave computes the updated states and returns the result to the master. N_c must be much smaller than the total row number N , or otherwise we loose a time $O(N_c/s)$ at the end of each generation in synchronization at the final collective call. Note that in fact we have to send to the slave two additional rows in order to perform the computations, i.e. we must send $N_c + 2$ rows. The computing time will be

$$T_{\text{comp}} = N^2/s$$

whereas the communication time will be

$$\begin{aligned} T_{\text{comm}} &= (\text{number-of-chunks}) (2 + N_c)N/b \\ &= (N/N_c)((2 + N_c)N/b) = (1 + 2/N_c) N^2/b \end{aligned}$$

Life with dynamic load balance (cont.)

Much in the same line as in the [previous analysis](#) for the PNT with dynamic load balance:

$$T_{\text{sync}} = (n/2) \times (\text{time-to-process-a-chunk}) = (n/2) N N_c / s$$

Efficiency will be, then

$$\begin{aligned}\eta &= \frac{T_{\text{comp}}}{T_{\text{comp}} + T_{\text{comm}} + T_{\text{sync}}} \\ &= \left[1 + \frac{(1 + 2/N_c)N^2/b}{N^2/s} + \frac{(n/2)N N_c}{N^2/s} \right]^{-1} \\ &= [1 + (1 + 2/N_c)s/b + nN_c/2N]^{-1} \\ &= [C + A/N_c + BN_c]^{-1} \\ C &= 1 + s/b; \quad A = 2s/b; \quad B = n/2N\end{aligned}$$

Life with dynamic load balance (cont.)

$$\eta = [C + A/N_c + BN_c]^{-1}; \quad C = 1 + s/b; \quad A = 2s/b; \quad B = n/2N$$

$A/C = N_{c,\text{comm}}^{1/2}$ is the $N^{1/2}$ for communication:

$$T_{\text{comp}} = T_{\text{comm}}, \quad \text{para } N_c = N_{c,\text{comm}}^{1/2}$$

If we consider the computing speed as $s = 115$ Mcell/sec reported previously for the sequential program and we consider that each cell needs a byte, then for a network hardware like Fast Ethernet with TCP/IP we have a bandwidth of $b = 90$ Mbit/sec = 11 Mcell/sec. The quotient is then $s/b \approx 10$. For $N_c \gg 10$ the term A/N_c will be negligible.

Life with dynamic load balance (cont.)

$$\eta = [C + A/N_c + BN_c]^{-1}; \quad C = 1 + s/b; \quad A = 2s/b; \quad B = n/2N$$

On the other hand, the term BN_c will be negligible for

$$N_c \gg C/B = N_{c,\text{sync}}^{1/2} = 2N/n.$$

We can make the **window** $[A, B^{-1}]$ be as large as we want by increasing enough N . However, efficiency will be bounded by above by

$$\eta \leq (1 + s/b)^{-1}$$

This is due to the fact that both computing and communication times are asymptotically independent of N_c . Whilst, for instance, in the implementation for the PNT it was not so, since the processing time was $O(N_c)$, whereas the communication times was independent of chunk size.

Life with dynamic load balance (cont.)

The algorithm for *life* described so far has the inconvenience that the computing time is of the same size as the communication time

$$\frac{T_{\text{sync}}}{T_{\text{comp}}} \rightarrow \text{cte}$$

So that, there is no parameter that allows us to increase efficiency beyond the limit value $(1 + s/b)^{-1}$. This limit value is dependent on hardware. It may be bad (low) for Fast Ethernet (100 Mbit/sec) whereas it may be good (high) for Gigabit Ethernet (1000 Mbit/sec). Or conversely, for a given network hardware, the limit value may be good for a slow processor or bad for a fast processor.

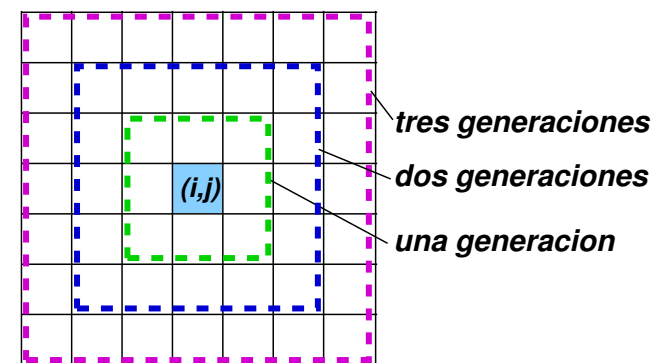
When bandwidth b is so slow that the maximum efficiency $(1 + s/b)^{-1}$ is low (say below 0.8), we can said that we experience of *data starvation*, i.e. the processor has computing power but he can use it because he is not *fed* with enough data.

Improving Life scalability

One way to improve the scalability is to look for a way to perform more computation without increasing the communication. We can do this by increasing the number of generations that are evaluated without communication between processors.

Note that if we want to compute the state of cell (i, j) in generation $n + 1$ then we need the states for cells $c_{i'j'}$ with

$|i' - i|, |j' - j| \leq 1$, in generation n , that means, the **dependency** zone for the cell at (i, j) is a square of 3×3 cells in order to advance one step. If we want to compute the state at cell (i, j) , state $n + 2$, then we need 2 additional layers, i.e. a square of 5×5 , and so on. In general to step from state n to state $n + k$ we will need k additional cell layers.



Improving Life scalability (cont.)

So, if we add k **ghost** cell layers, then we can evaluate k generations without need of communication. Of course, each k generations we have to communicate the k additional layers. If we think a 1D decomposition, then

$$T_{\text{comp}} = kN^2/s$$

$$\begin{aligned} T_{\text{comm}} &= (\text{number-of-chunks}) \times (4k + N_c)N/b \\ &= (1 + 4k/N_c)N^2/b \end{aligned}$$

so that now

$$T_{\text{comm}}/T_{\text{comp}} = (1 + 4k/N_c)s/kb$$

If we want that $T_{\text{comm}}/T_{\text{comp}} < 0.1$ then it will be enough to take $k > 10s/b$ y $N_c \gg 4k$.

Improving Life scalability (cont.)

On the other hand, the synchronization time is

$$T_{\text{sync}} = (n/2) \times (\text{time-to-process-a-chunk}) = (n/2) k N N_c / s$$

so that

$$T_{\text{sync}} / T_{\text{comp}} = \frac{n}{2} \frac{N_c}{N}$$

and this ratio can be made smaller as we want (let's say less than 0.1) by making $N_c \ll 0.2N/n$. Of course, in practice if the combination of hardware is too slow in the communication side so that k is too large, then the boards will be too big so as the implementation will not be of any practical use.

Improving Life scalability (cont.)

The synchronization time may be reduced by using square chunks. Effectively if we divide the board in $N \times N$ square (approx.) patches of $N_c \times N_c$ rows and columns, then we have

$$T_{\text{comp}} = kN^2/s$$

$$\begin{aligned} T_{\text{comm}} &= (\text{number-of-chunks}) \times (4k + N_c)N_c/b \\ &= (N/N_c)^2 (4k + N_c)N_c/b = N^2/b (1 + 4k/N_c) \end{aligned}$$

so that

$$T_{\text{comm}}/T_{\text{comp}} = (1 + 4k/N_c)s/kb$$

is the same as before.

Improving Life scalability (cont.)

But the synchronization time will be

$$T_{\text{sync}} = (n/2) \times (\text{time-to-process-a-chunk}) = (n/2) k N_c^2 / s$$

so that

$$T_{\text{sync}} / T_{\text{comp}} = \frac{n}{2} \left(\frac{N_c}{N} \right)^2$$

which can be made small than, say, 0.1 by taking

$$N_c < \sqrt{0.2/nN}$$

This allows to obtain a range for N_c acceptable without having to increase the size of the board too much.

OPTIONAL Assignment Nbr. 4

Write an implementation for Life with dynamic load balance following the strategy *compute-on-demand* and processing several stages without communication. Compare communication times with the following variants

- Send boards as byte arrays.
- Send boards as bits arrays.
- Send arrays in *sparse* format.

OPTIONAL Assigment Nbr. 4 (cont.)

Note: If you use `vector<bool>` then you will not be able to extract the `char` array to send. You can instead use this ad-hoc class:

```
1 #define NBITCHAR 8
2 class bit_vector_t {
3 public:
4     int nchar, N;
5     unsigned char *buff;
6     bit_vector_t(int N)
7     : nchar(N/NBITCHAR+(N%NBITCHAR>0)) {
8         buff = new unsigned char[nchar];
9         for (int j=0; j<nchar; j++) buff[j] = 0;
10    }
11    int get(int j) {
12        return (buff[j/NBITCHAR] & (1<<j%NBITCHAR))>0;
13    }
14    void set(int j, int val) {
15        if (val) buff[j/NBITCHAR] |= (1<<j%NBITCHAR);
16        else buff[j/NBITCHAR] &= ~(1<<j%NBITCHAR);
17    }
18 };
```

OPTIONAL Assignment Nbr. 4 (cont.)

To create the vector

```
1  bit_vector_t v(N);
```

N is the number of bits. With the routines *get()* and *set()* you can manipulate the values.

```
1  int v = v.get(j); // retorna el bit en la posición 'j'
2  v.set(j, val);    // setea el bit en la posición 'j' al valor 'val'
```

Finale, to send the array:

```
1  char *v.buff : el buffer interno
2  int v.char: el numero de chars en el vector
```

So that you can send or receive it with the *MPI_CHAR* type. For instance:

```
1  MPI_Send(v.buff, v.nchar, MPI_CHAR, . . .);
```

The Poisson equation

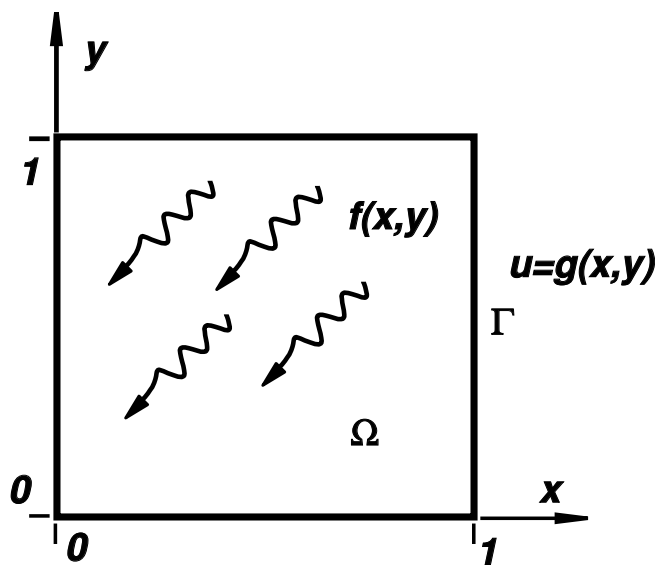
The Poisson equation

- It's an example of a Computational Mechanics problem with a finite difference scheme.
- Introduces the *virtual topology* concept.
- Discusses in detail a series of variations in the send/receive communication pattern to avoid *deadlock*.
- It's an introduction to the parallel implementation of matrix-vector products.
- Even if PETSc provides tools in order to solve some of the problems posed here it's interesting for the concepts introduced here and in order to understand some PETSc components.
- This example is taken from *Using MPI* (chapter 4, *Intermediate MPI*). Fortran code is available in the MPICH distribution *\$MPI_HOME/examples/test/topol*.

The Poisson equation (cont.)

It's a simple PDE but at the same time consists in the *kernel* for other algorithms and preconditioners (NS con fractional step, preconditioners, etc...)

$$\Delta u = f(x, y), \quad \text{in } \Omega = [0, 1] \times [0, 1]$$
$$u(x, y) = g(x, y), \quad \text{in } \Gamma = \partial\Omega$$



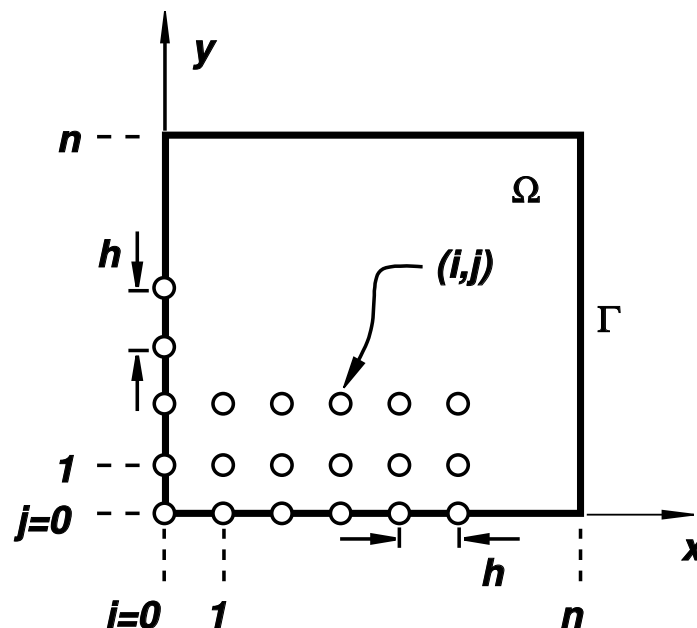
The Poisson equation (cont.)

We define a computational grid composed of $n \times n$ linear segments of step size $h = 1/n$.

$$x_i = i/n; \quad i = 0, 1, \dots, n$$

$$y_j = j/n; \quad j = 0, 1, \dots, n$$

$$\mathbf{x}_{ij} = (x_i, y_j)$$

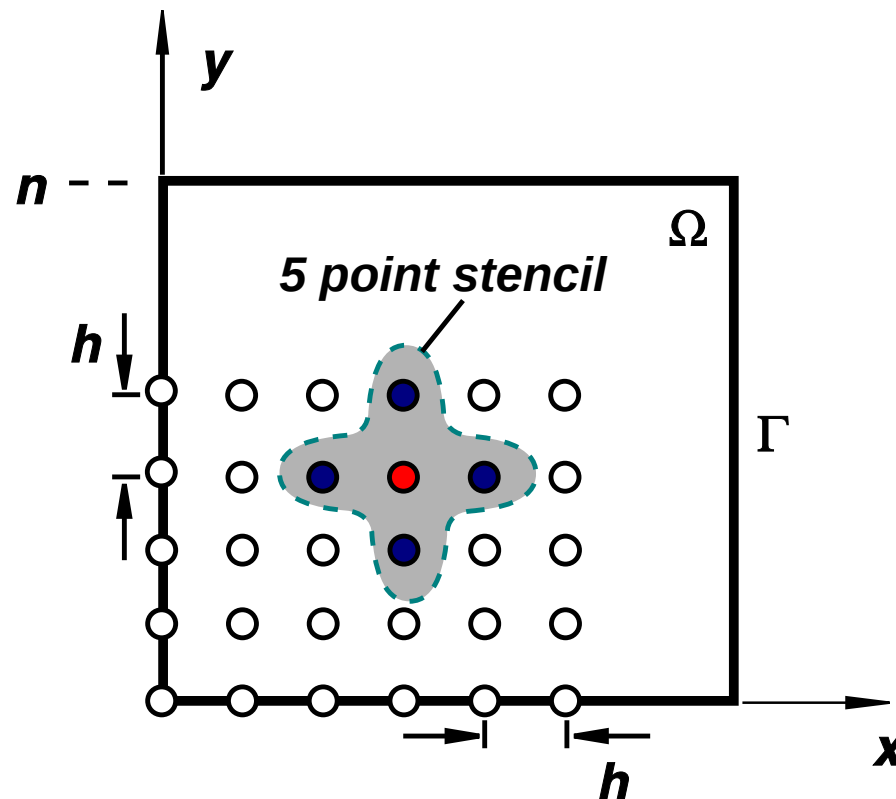


The Poisson equation (cont.)

We look for approximate values for u at the nodes

$$u_{ij} \approx u(x_i, y_j)$$

We use the approximation of the **5 point stencil** that is obtained by discretizing the Laplace operator by centered finite differences of second order



$$\begin{aligned} \Delta u &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \\ &= (1/h^2) / (u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) \end{aligned}$$

The Poisson equation (cont.)

Reemplazando en le ec. de Poisson

$$(1/h^2) / (u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) = f_{ij}$$

This is linear system of $N = (n - 1)^2$ equations (= number of interior points), with N unknowns, that are the values at the interior points. Some equations include values at the boundary, but they are known, due to the Dirichlet kind of boundary condition.

$$\mathbf{A}\mathbf{u} = \mathbf{f}$$

The Poisson equation (cont.)

An **iterative method** consists in some kind of constructive algorithm that generates a sequence of vectors u^0, u^1, \dots, u^k such that $u^k \rightarrow u$. One possibility is to put the system as a **fixed point** equation

$$A = D + A', \quad D = \text{diag}(A)$$

$$(D + A')u = f$$

$$Du = f - A'u$$

$$u = D^{-1}(f - A'u)$$

So that, we can iterate in the following way

$$u^{k+1} = D^{-1}(f - A'u^k)$$

The Poisson equation (cont.)

The iterative method is then

$$u_{ij}^{k+1} = \frac{1}{4}(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - h^2 f_{ij})$$

a.k.a. **Jacobi iteration**. It can be shown that the scheme is convergent provided that A is **diagonal dominant**, i.e. that

$$\sum_{j \neq i} |A_{ij}| < |A_{ii}|, \quad \forall i$$

In this case A is just in the limit of convergence since

$$|A_{ii}| = \sum_{j \neq i} |A_{ij}| = 4/h^2$$

But it can be shown that due to the Dirichlet boundary conditions the scheme is indeed convergent.

The Poisson equation (cont.)

There are variants of the algorithm (overrelaxed, Gauss-Seidel, for instance), however the functions that we will develop here basically implement a generic matrix-vector product in parallel, and so they can be used with minor modifications with the said algorithms.

The Poisson equation (cont.)

A C++ function that performs this operations may be written as follows:

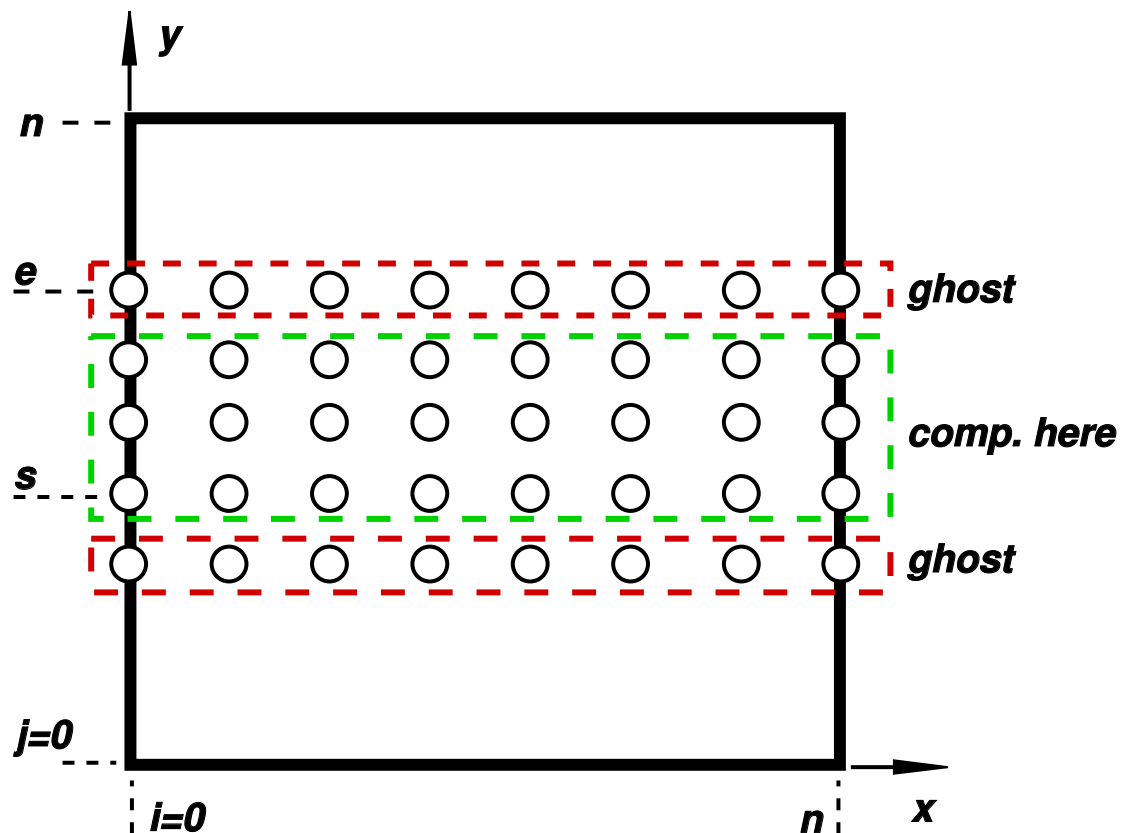
```

1  int n=100, N=(n+1)^2;
2  double h=1.0/double(n);
3  double
4      *u = new double[N],
5      *unew = new double[N];
6  #define U(i,j) u((n+1)*(j)+(i))
7  #define UNEW(i,j) unew((n+1)*(j)+(i))
8
9  // Assign b.c. values
10 for (int i=0; i<n; i++) {
11     U(i,0) = g(h*i,0.0);
12     U(i,n) = g(h*i,1.0);
13 }
14
15 for (int j=0; j<n; j++) {
16     U(0,j) = g(0.0,h*j);
17     U(n,j) = g(1.0,h*j);
18 }
19
20 // Jacobi iteration
21 for (int j=1; j<n; j++) {
22     for (int i=1; i<n; i++) {
23         UNEW(i,j) = 0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)
24                             +U(i,j+1)-h*h*F(i,j));
25     }
26 }

```

The Poisson equation (cont.)

The simplest form to decompose the problem for parallel processing is to split the domain in horizontal strips (as in the Life example), so that in each processor only the files in range $[s, e)$. The data dependency analysis (similar to Life) shows that we need two adjacent (*ghost*) rows.



The Poisson equation (cont.)

The code is modified in order to only compute rows in the assigned band.

```
1 // define s,e ...
2 int rows_here = e-s+2;
3 double
4   *u = new double[ (n+1)*rows_here],
5   *unew = new double[ (n+1)*rows_here];
6 #define U(i,j) u( (n+1)*(j-s+1)+i)
7 #define UNEW(i,j) unew( (n+1)*(j-s+1)+i)
8
9 // Assign b.c. values ...
10
11 // Jacobi iteration
12 for (int j=s; j<e; j++) {
13   for (int i=1; i<n; i++) {
14     UNEW(i,j) = 0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)
15                      +U(i,j+1)-h*h*F(i,j));
16   }
17 }
```

Virtual Topologies

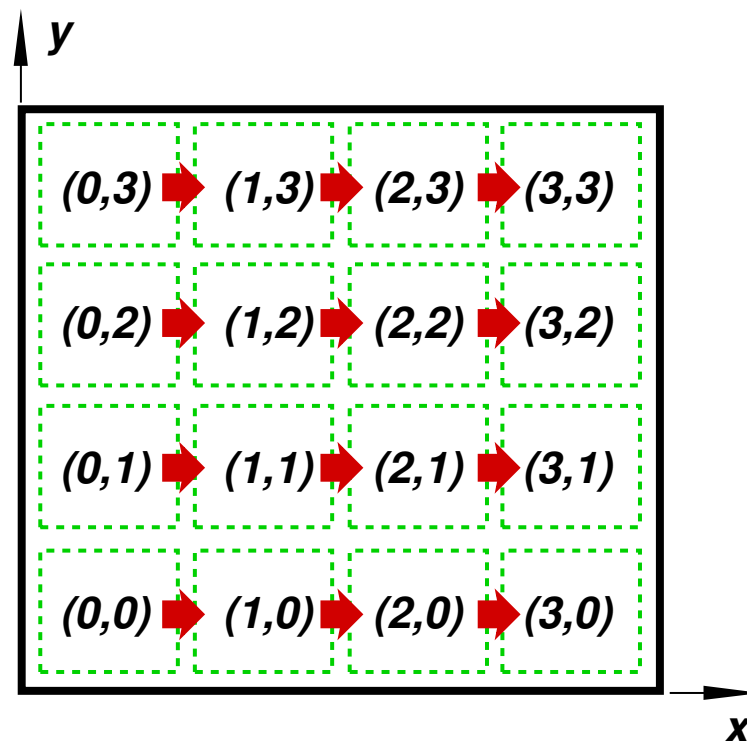
- ***Partiton***: we must assign a partition (i.e. a certain rank $[s, e)$) to each process.
- The optimal partition depends on how the network hardware is connected, i.e. the ***network topology*** (ring, star, toroidal...)
- For instance, if the underlying topology is a ***ring*** then the best partitioning could be a simple division in horizontal strips.
- In many codes, each process communicates with a reduced number of neighbors. This is called the ***application topology***.

Virtual Topologies (cont.)

- In order that the parallel implementation be efficient we want the *application topology* to match as closely as possible to the *hardware topology*.
- For the Poisson problem at hand it seems that the best order us to assign processes with increasing rank from the bottom to the top of the computational domain, but this can depend on hardware. MPI allows the hardware vendor to develop specialized topology routines.
- As the user chooses an *application topology*, he is establishing which is the main communication pattern. However, of course, all processes will be able to communicate among them.

Virtual Topologies (cont.)

The simplest virtual topology (and used frequently in numerical applications) is the ***cartesian topology***. As described so far, we could use a 1D cartesian topology, but in fact the problem calls for a 2D cartesian topology. There are also off course 3D cartesian topologies and beyond.



Virtual Topologies (cont.)

In the 2D cartesian topology each process is assigned a 2 number tuple (I, J) . MPI provides a series of functions in order to define, examine and manipulate this topologies.

The ***MPI_Cart_create(...)*** function creates a 2D cartesian topology, his signature is

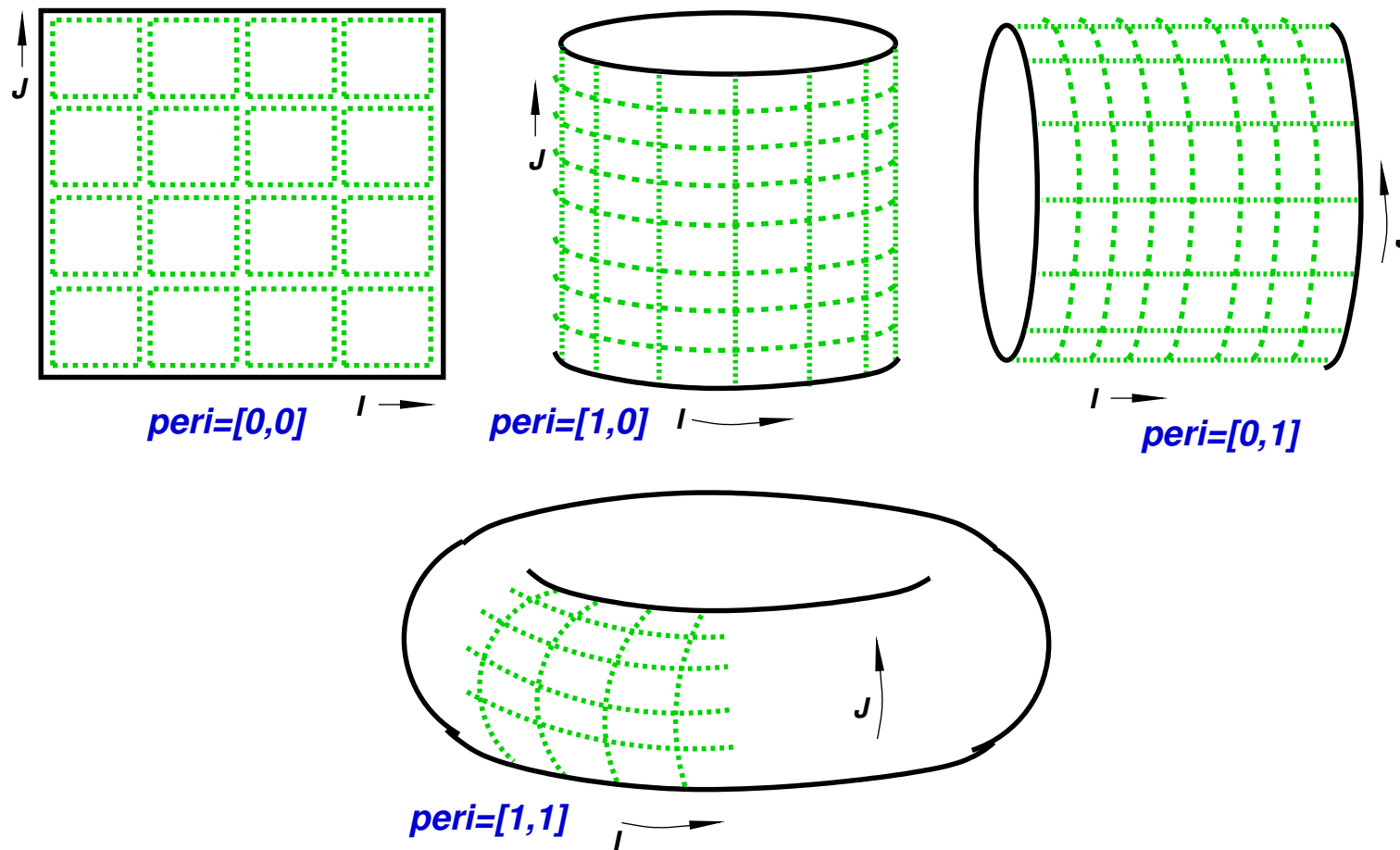
```
1 MPI_Cart_create(MPI_Comm comm_old, int ndims,  
2    int *dims, int *periods, int reorder, MPI_Comm *new_comm);
```

dims is an array of files/rows numbers in each direction and ***periods*** is an array of ***flags*** that signales whether a direction is periodic or not.

In this example the call is as follows

```
1 int dims[2]={4,4}, periods[2]={0,0};  
2 MPI_Comm comm2d;  
3 MPI_Cart_create(MPI_COMM_WORLD, 2,  
4    dims, periods, 1, comm2d);
```

Virtual Topologies (cont.)



Virtual Topologies (cont.)

- **reorder**: if set to true it means that MPI can reorder the processes so as to optimize the relation between the virtual and hardware topologies.
- **MPI_Cart_get()** allows to recover the dimensions, periodicity and coordinates (within the topology) of this process.

```
1 int MPI_Cart_get(MPI_Comm comm, int maxdims,  
2   int *dims, int *periods, int *coords );
```

- User can get only the process coordinate tuple within the topology:

```
1 int MPI_Cart_coords(MPI_Comm comm, int rank,  
2   int maxdims, int *coords);
```

Virtual Topologies (cont.)

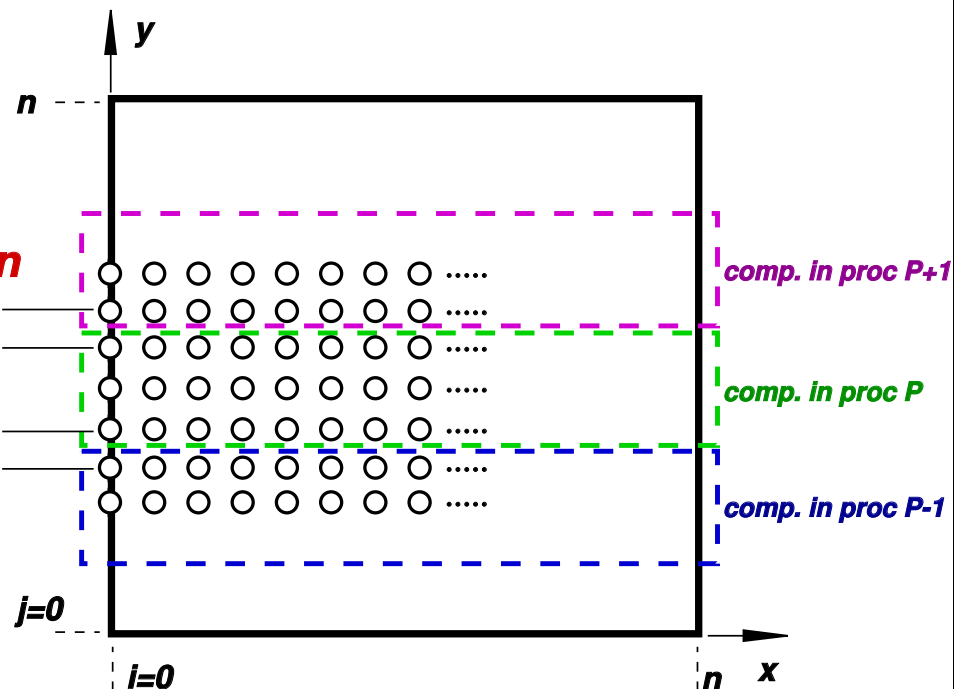
Before making a computation of the new state $u^k \rightarrow u^{k+1}$, we have to update the *ghost values*, for instance, with a 1D topology:

- Send $e-1$ to $P+1$
- Receive e from $P+1$
- Send s to $P-1$
- Receive $s-1$ from $P-1$

communication

$e(\text{ghost in proc } P)$
 $e-1(\text{ghost in proc } P+1)$

$s(\text{ghost in proc } P-1)$
 $s-1(\text{ghost in proc } P)$



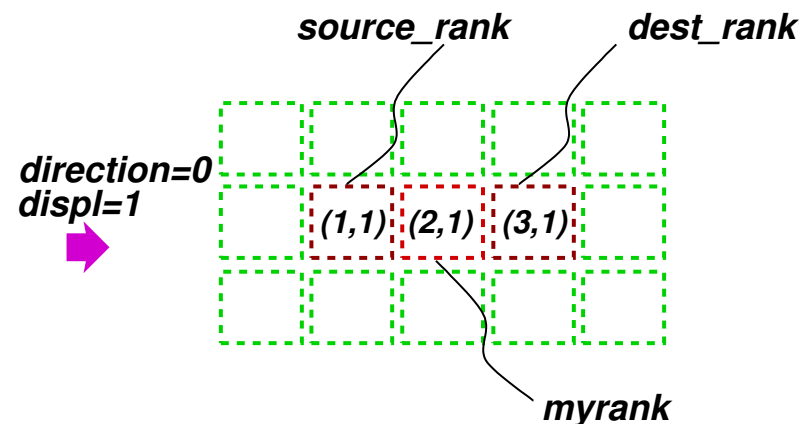
Virtual Topologies (cont.)

In general it can be seen that we are doing a **shift** of the ghost values to the processors up and below from the current one. This a very common operation and MPI provides a specific routine (***MPI_Cart_shift()***) that computes the ranks of the processes for a shift operation:

```
1 int MPI_Cart_shift(MPI_Comm comm, int direction, int displ,
2                   int *source, int *dest);
```

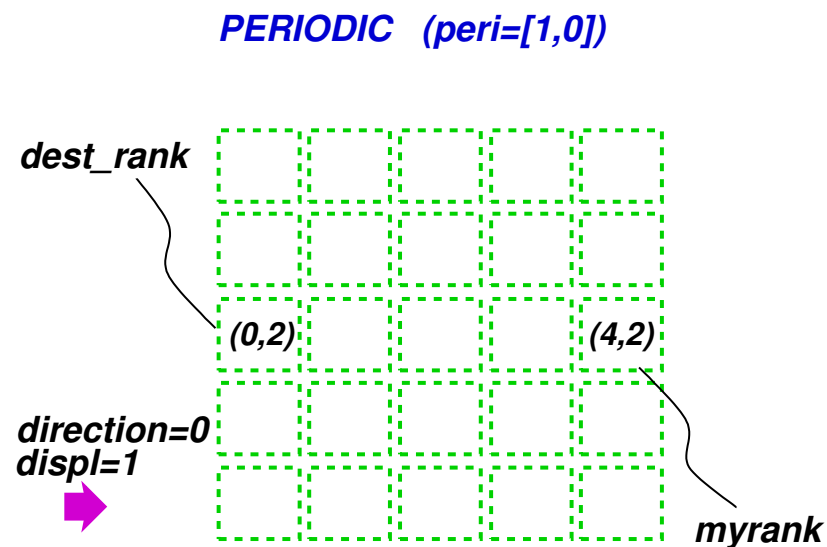
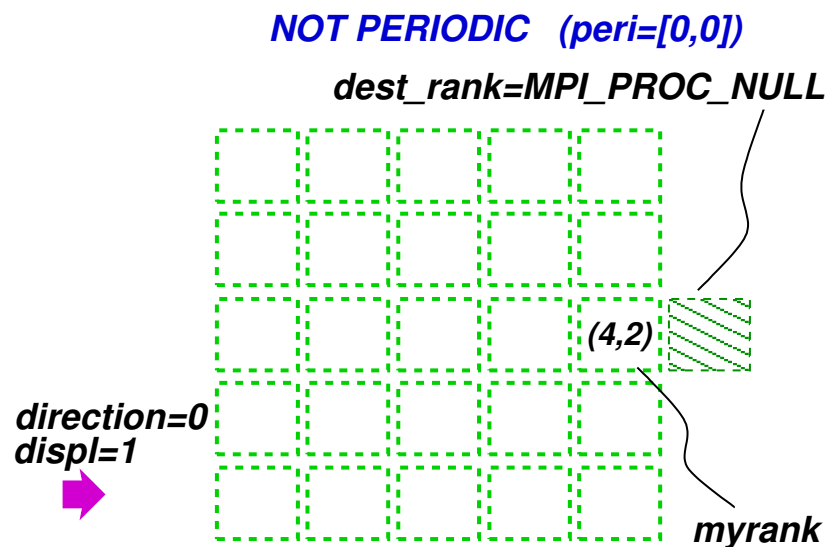
For instance, with a 2D topology, in order to do a **shift** in the horizontal direction.

```
1 int source_rank, dest_rank;
2 MPI_Cart_shift(comm2d, 0, 1, &source_rank, &dest_rank);
```



Virtual Topologies (cont.)

What happens at the boundary processors? For instance if the topology is 5x5, then a shift in direction 0 (x axis) with a displacement *displ=1* for processor (4, 2) gives



Virtual Topologies (cont.)

MPI_PROC_NULL is the ***null process***. The idea is that sending messages to ***MPI_PROC_NULL*** is equivalent to do nothing (as writing to ***/dev/null*** in Unix).

An operation like

```
1 MPI_Send( . . . , dest , . . . )
```

is equivalent to

```
1 if (dest != MPI_PROC_NULL) MPI_Send( . . . , dest , . . . ) ;
```

Virtual Topologies (cont.)

If the number of rows to process is $n-1$ is a multiple of $size$, then we can compute the row range to be computed in this processor $[s, e)$ as

```
1 s = 1 + myrank*(n-1)/size;  
2 e = s + (n-1)/size;
```

If it is not a multiple then

```
1 // All receive 'nrp' or 'nrp+1'  
2 // First 'rest' processors are assigned 'nrp+1'  
3 nrp = (n-1)/size;  
4 rest = (n-1) % size;  
5 s = 1 + myrank*nrp + (myrank < rest ? myrank : rest);  
6 e = s + nrp + (myrank < rest);
```

Virtual Topologies (cont.)

MPI provides a routine *MPE_Decomp1d(...)* that does wxactly this operation.

```
1 int MPE_Decomp1d(int n, int size, int rank, int *s, int *e);
```

If the weight is not uniform, then we can apply the following algorithm. We want to distribute *n* objects among *size* processes proportionally yo *weights[j]* (normalized).

To each processors it correspond roughly *weights[j]*n* objects, but as the numbers may not be integer, we have to take into account the carry en

$$weights[j]*n = \text{floor}(weights[j]*n) + carry;$$

We loop over the processes and accumulate *carry*, when this quantity reaches unity, we assign one more object to this process.

Virtual Topologies (cont.)

For instance, if we want to distribute 1000 objects among 10 processes with the following weights, the resulting distribution is as follows

```
1 in [0] weight 0.143364, wants 143.364373, assigned 143
2 in [1] weight 0.067295, wants 67.295034, assigned 67
3 in [2] weight 0.133623, wants 133.623150, assigned 134
4 in [3] weight 0.136241, wants 136.240810, assigned 136
5 in [4] weight 0.155558, wants 155.557799, assigned 156
6 in [5] weight 0.033709, wants 33.708929, assigned 33
7 in [6] weight 0.057200, wants 57.200313, assigned 57
8 in [7] weight 0.131086, wants 131.085889, assigned 132
9 in [8] weight 0.047398, wants 47.397738, assigned 47
10 in [9] weight 0.094526, wants 94.525966, assigned 95
11 total rows 1000
```


Virtual Topologies (cont.)

The following function distributes *n* objects among *size* processes with weights *weights[]*.

```

1 void decomp(int n, vector<double> &weights,
2             vector<int> &nrows) {
3     int size = weights.size();
4     nrows.resize(size);
5     double sum_w = 0., carry=0., a;
6     for (int p=0; p<size; p++) sum_w += weights[p];
7     int j = 0;
8     double tol = 1e-8;
9     for (int p=0; p<size; p++) {
10         double w = weights[p]/sum_w;
11         a = w*n + carry + tol;
12         nrows[p] = int(floor(a));
13         carry = a - nrows[p] - tol;
14         j += nrows[p];
15     }
16     assert(j==n);
17     assert(fabs(carry) < tol);
18 }

```

This kind of partitioning is an example of *static load balance*.

Poisson's eq. Communication strategy

Function *exchng1* (*u1*, *u2*, ...) performs the communication (exchange of *ghost* values).

```
1 void exchng1(double *a, int n, int s, int e,  
2             MPI_Comm comm1d, int nbrbot, int nbrtop) {  
3     MPI_Status status;  
4     #define ROW(j) (&a[(j-s+1)*(n+1)])  
5  
6     // Exchange top row  
7     MPI_Send(ROW(e-1), n+1, MPI_DOUBLE, nbrtop, 0, comm1d);  
8     MPI_Recv(ROW(s-1), n+1, MPI_DOUBLE, nbrbot, 0, comm1d, &status);  
9  
10    // Exchange top row  
11    MPI_Send(ROW(s), n+1, MPI_DOUBLE, nbrbot, 0, comm1d);  
12    MPI_Recv(ROW(e), n+1, MPI_DOUBLE, nbrtop, 0, comm1d, &status);  
13 }
```

Poisson's eq. Communication strategy (cont.)

We have already almost all tools in order to solve the problem.

- We add a *sweep*(*u1*, *u2*, ...) function that performs the Jacobi iteration, computing the new state u^{k+1} (*u2*) in terms of the previous state u^k (*u1*). The code for this function can be easily implemented from the one previously shown.
- A function *double diff*(*u1*, *u2*, ...) that computes the norm of the difference between the stated *u1* and *u2* in each processor.

Poisson's eq. Communication strategy (cont.)

```
1 int main() {
2     int n=10;
3
4     // Get MPI info
5     int size, rank;
6     MPI_Comm_size(MPI_COMM_WORLD, &size);
7
8     int periods = 0;
9     MPI_Comm comm1d;
10    MPI_Cart_create(MPI_COMM_WORLD, 1, &size, &periods, 1, &comm1d);
11    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
13    int direction=1, nbrbot, nbrtop;
14    MPI_Cart_shift(comm1d, direction, 1, &nbrbot, &nbrtop);
15
16    int s, e;
17    MPE_Decomp1d(n-1, size, rank, &s, &e);
18
19    int
20        rows_here = e-s+2,
21        points_here = rows_here*(n+1);
22    double
23        *tmp,
24        *u1 = new double[points_here],
25        *u2 = new double[points_here],
26        *f = new double[points_here];
27    #define U1(i, j) u1((n+1)*(j-s+1)+i)
```

```

28 #define U2(i,j) u2((n+1)*(j-s+1)+i)
29 #define F(i,j) f((n+1)*(j-s+1)+i)
30
31 // Assign bdry conditions to 'u1' and 'u2' ...
32 // Define 'f' ...
33 // Initialize 'u1' and 'u2'
34
35 // Do computations
36 int iter, itmax = 1000;
37 for (iter=0; iter<itmax; iter++) {
38     // Communicate ghost values
39     exchngl(u1,n,s,e,commld,nbrbot,nbrtop);
40     // Compute new 'u' values from Jacobi iteration
41     sweep(u1,u2,f,n,s,e);
42
43     // Compute difference between
44     // 'u1' ( $u^k$ ) and 'u2' ( $u^{k+1}$ )
45     double diff_here2 = diff(u1,u2,n,s,e);
46     double diff;
47     MPI_Allreduce(&diff_here2,&diff,1,MPI_DOUBLE,
48                 MPI_SUM,MPI_COMM_WORLD);
49     diff = sqrt(diff);
50     printf("iter %d, diff %d\n",iter,diff);
51
52     // Swap pointers
53     tmp=u1; u1=u2; u2=tmp;
54
55     if (diff<1e-5) break;
56 }
57
58 printf("Converged in %d iterations\n",iter);
59
60 delete[] u1;

```

```
61  delete[ ] u2;  
62  delete[ ] f;  
63 }
```

Advanced MPI collective operations

Advanced MPI collective operations

We have already seen the basic collective operations (*MPI_Bcast()* and *MPI_Reduce()*). Collective functions have the advantage that allow to perform complex common operations in simply and efficiently.

There are other collective operations, namely

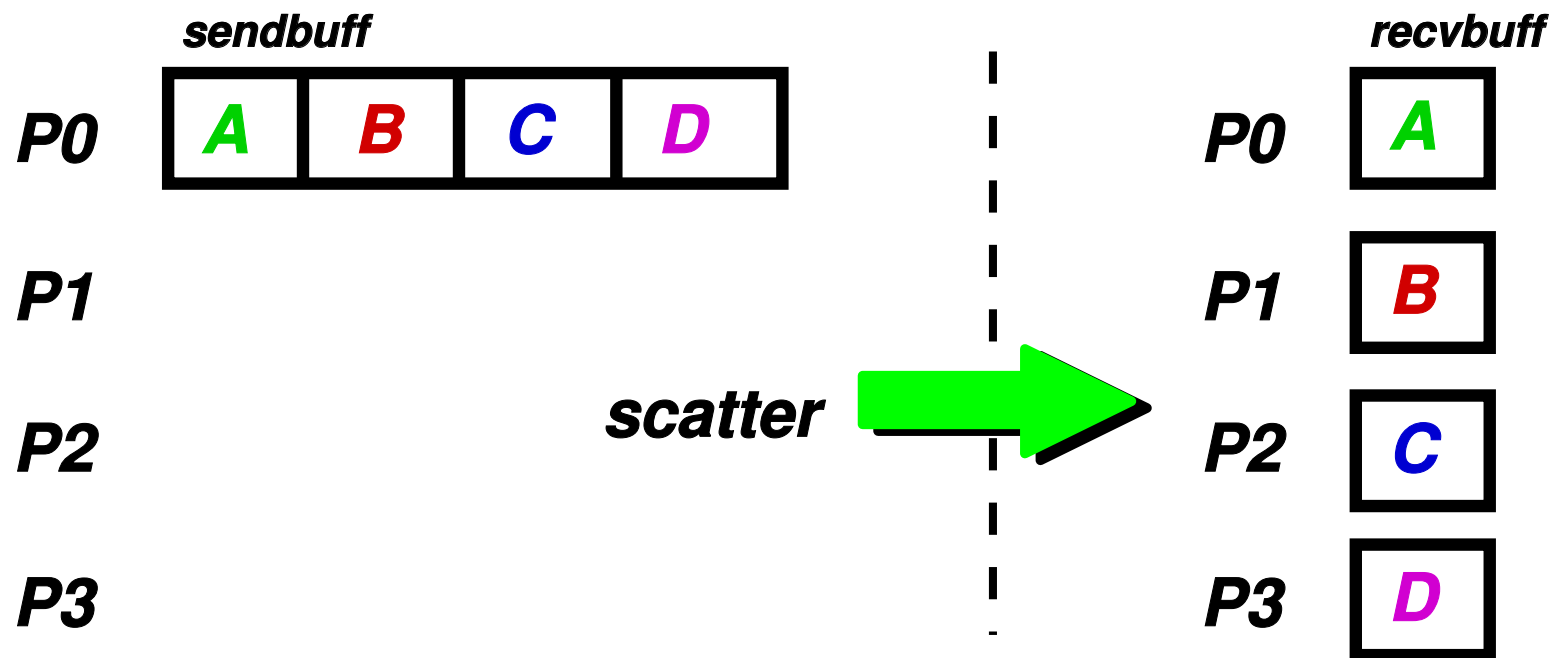
- *MPI_Scatter()*, *MPI_Gather()*
- *MPI_Allgather()*
- *MPI_Alltoall()*

And their vectorized (variable length per processor) versions

- *MPI_Scatterv()*, *MPI_Gatherv()*
- *MPI_Allgatherv()*
- *MPI_Alltoallv()*

Scatter operations

Sends a certain amount of data of the same size and type to the other processes (as in `MPI_Bcast()`, but the data to be sent is not the same to all processes).



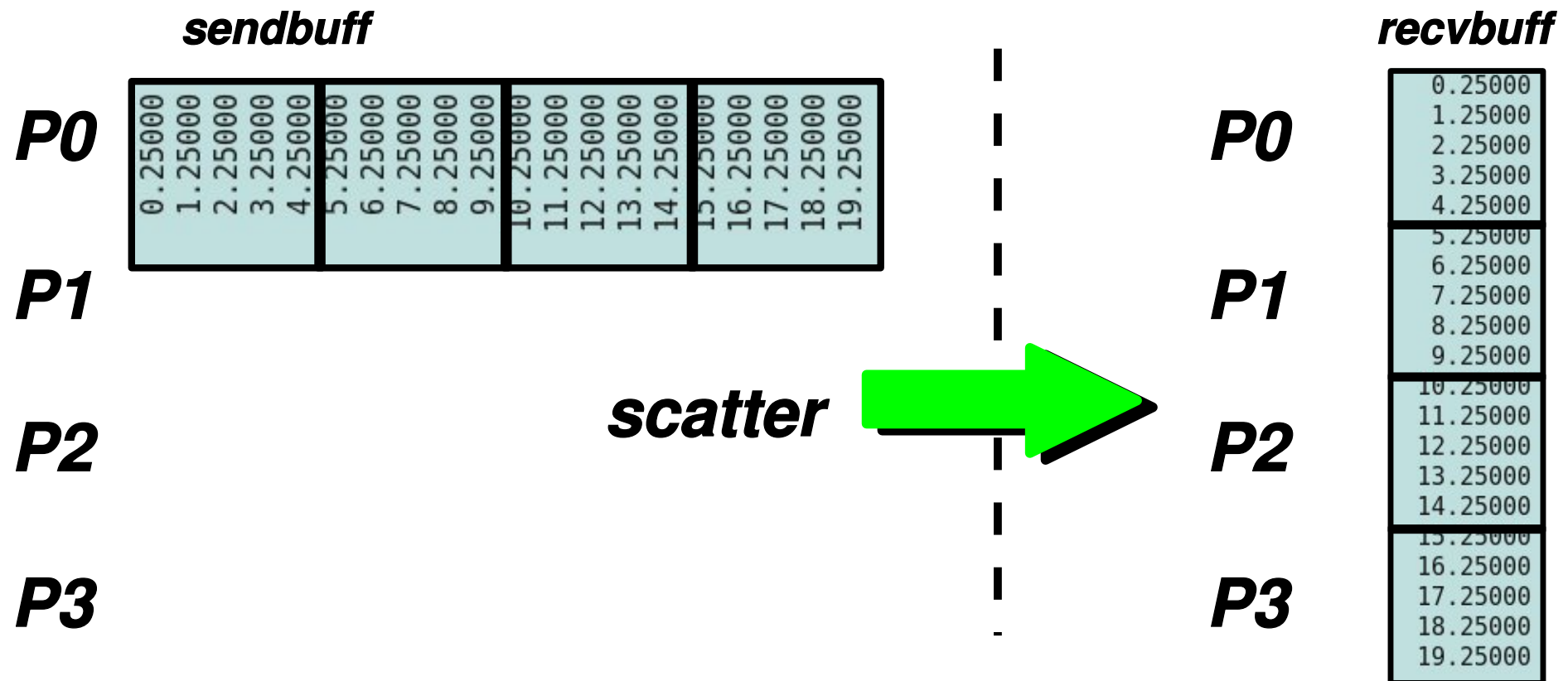
Scatter operations (cont.)

```

1  #include <mpi.h>
2  #include <stdio>
3
4  int main(int argc, char **argv) {
5      MPI_Init(&argc, &argv);
6      int myrank, size;
7      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10     int N = 5;           // Nbr of elements to send to each processor
11     double *sbuff = NULL;
12     if (!myrank) {
13         sbuff = new double[N*size]; // send buffer only in master
14         for (int j=0; j<N*size; j++) sbuff[j] = j+0.25; // fills 'sbuff'
15     }
16     double *rbuff = new double[N]; // receive buffer in all procs
17
18     MPI_Scatter(sbuff, N, MPI_DOUBLE,
19               rbuff, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
20
21     for (int j=0; j<N; j++)
22         printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
23     MPI_Finalize();
24     if (!myrank) delete[] sbuff;
25     delete[] rbuff;
26 }

```

Scatter operations (cont.)

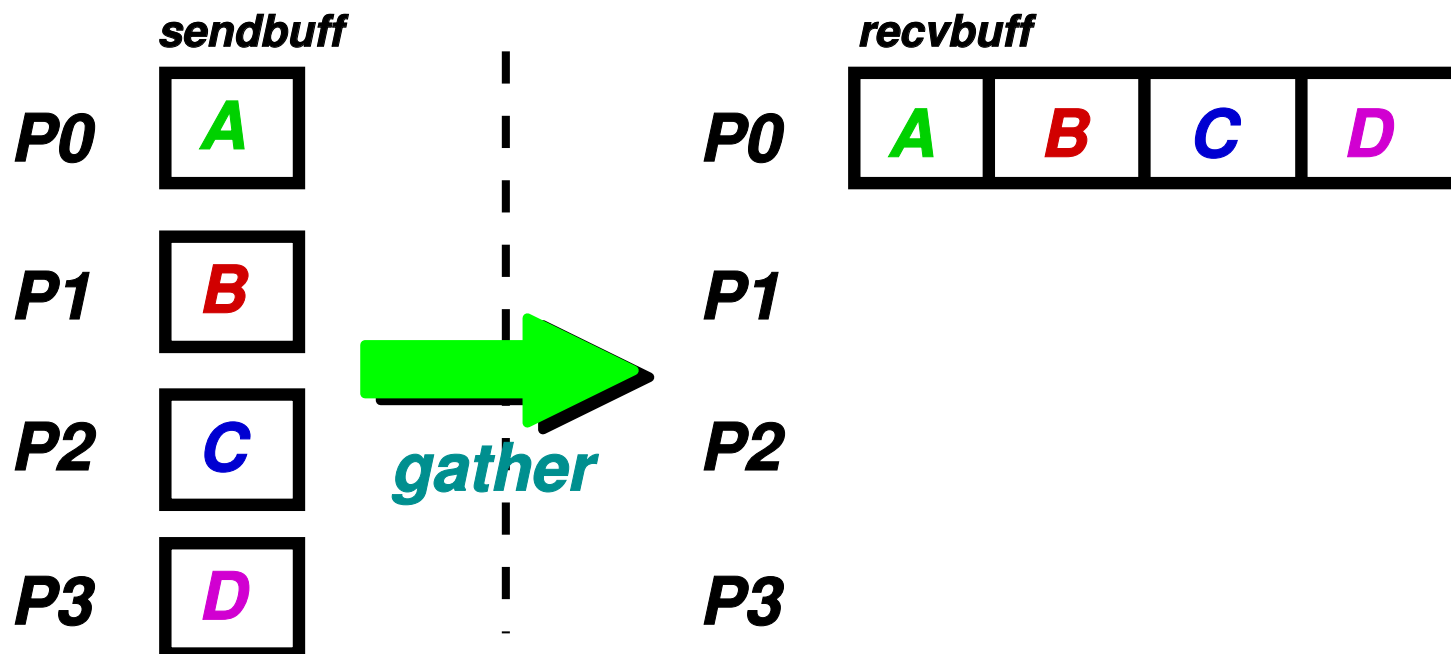


Scatter operations (cont.)

```
1 [mstorti@spider example]$ mpirun -np 4 -machinefile \
2                               machi.dat scatter2.bin
3 [0] 0 -> 0.250000
4 [0] 1 -> 1.250000
5 [0] 2 -> 2.250000
6 [0] 3 -> 3.250000
7 [0] 4 -> 4.250000
8 [2] 0 -> 10.250000
9 [2] 1 -> 11.250000
10 [2] 2 -> 12.250000
11 [2] 3 -> 13.250000
12 [2] 4 -> 14.250000
13 [3] 0 -> 15.250000
14 [3] 1 -> 16.250000
15 [3] 2 -> 17.250000
16 [3] 3 -> 18.250000
17 [3] 4 -> 19.250000
18 [1] 0 -> 5.250000
19 [1] 1 -> 6.250000
20 [1] 2 -> 7.250000
21 [1] 3 -> 8.250000
22 [1] 4 -> 9.250000
23 [mstorti@spider example]$
```

Gather operations

Is the inverse to scatter, (*gathers*) a certain length of data from each processor in a destination processor.



Gather operations (cont.)

```

1  #include <mpi.h>
2  #include <stdio>
3
4  int main(int argc, char **argv) {
5      MPI_Init(&argc, &argv);
6      int myrank, size;
7      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10     int N = 5;          // Nbr of elements to send to each processor
11     double *sbuff = new double[N]; // send buffer in all procs
12     for (int j=0; j<N; j++) sbuff[j] = myrank*1000.0+j;
13
14     double *rbuff = NULL;
15     if (!myrank) {
16         rbuff = new double[N*size]; // recv buffer only in master
17     }
18
19     MPI_Gather(sbuff, N, MPI_DOUBLE,
20               rbuff, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
21
22     if (!myrank)
23         for (int j=0; j<N*size; j++)
24             printf("%d -> %f\n", j, rbuff[j]);
25     MPI_Finalize();
26
27     delete[] sbuff;
28     if (!myrank) delete[] rbuff;
29 }

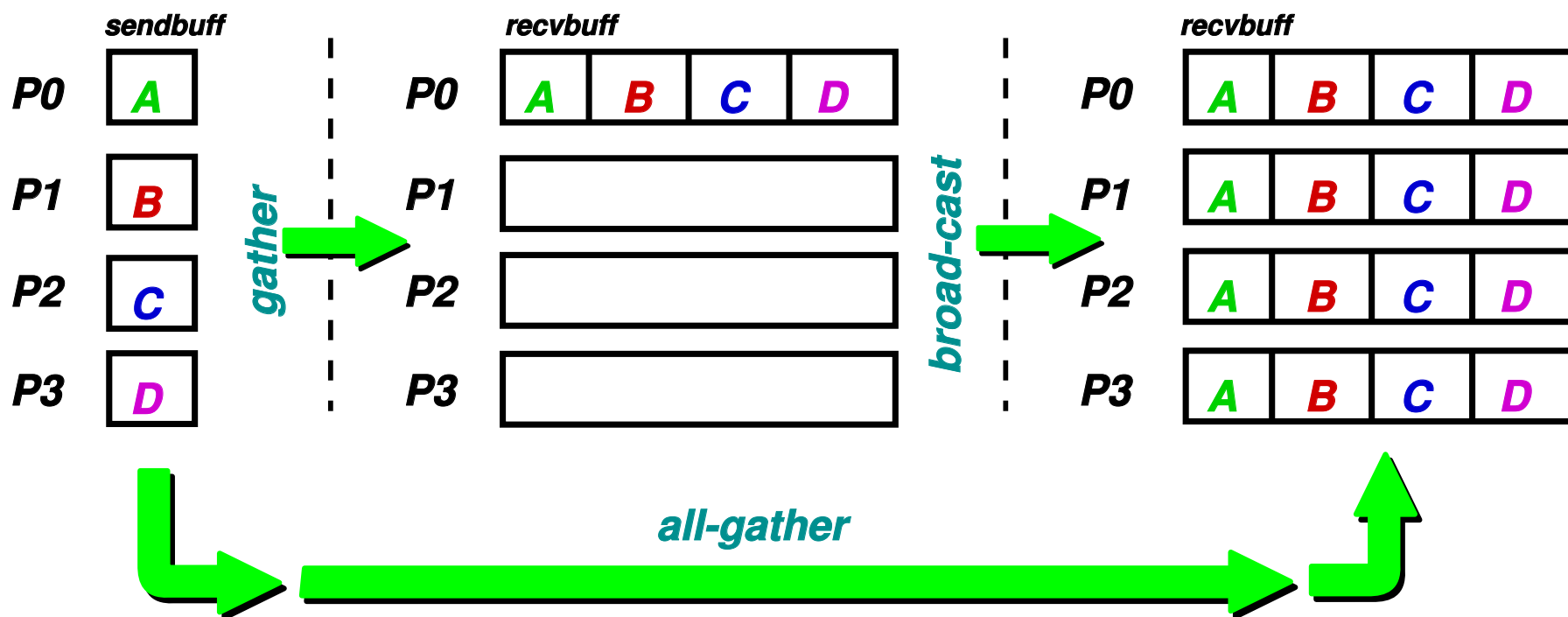
```

Gather operations (cont.)

```
1 [mstorti@spider example]$ mpirun -np 4 \  
2                               -machinefile machi.dat gather.bin  
3 0 -> 0.000000  
4 1 -> 1.000000  
5 2 -> 2.000000  
6 3 -> 3.000000  
7 4 -> 4.000000  
8 5 -> 1000.000000  
9 6 -> 1001.000000  
10 7 -> 1002.000000  
11 8 -> 1003.000000  
12 9 -> 1004.000000  
13 10 -> 2000.000000  
14 11 -> 2001.000000  
15 12 -> 2002.000000  
16 13 -> 2003.000000  
17 14 -> 2004.000000  
18 15 -> 3000.000000  
19 16 -> 3001.000000  
20 17 -> 3002.000000  
21 18 -> 3003.000000  
22 19 -> 3004.000000  
23 [mstorti@spider example]$
```

All-gather operation

It's conceptually equivalent to perform a *gather* followed by a *broadcast*.



All-gather operation (cont.)

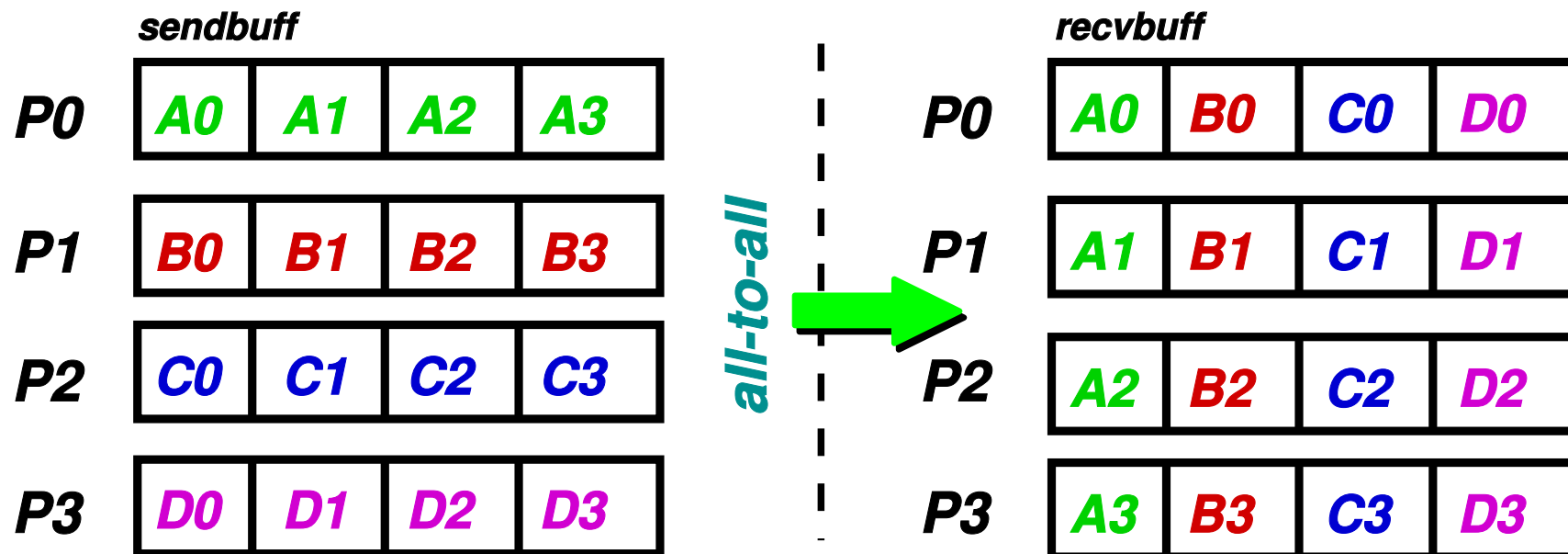
```
1 #include <mpi.h>
2 #include <stdio>
3
4 int main(int argc, char **argv) {
5     MPI_Init(&argc, &argv);
6     int myrank, size;
7     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
8     MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10    int N = 3;          // Nbr of elements to send to each processor
11    double *sbuff = new double[N]; // send buffer in all procs
12    for (int j=0; j<N; j++) sbuff[j] = myrank*1000.0+j;
13
14    double *rbuff = new double[N*size]; // receive buffer in all procs
15
16    MPI_Allgather(sbuff, N, MPI_DOUBLE,
17                 rbuff, N, MPI_DOUBLE, MPI_COMM_WORLD);
18
19    for (int j=0; j<N*size; j++)
20        printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
21    MPI_Finalize();
22
23    delete[] sbuff;
24    delete[] rbuff;
25 }
```

All-gather operation (cont.)

```
1 [mstorti@spider example]$ mpirun -np 3 \  
2 -machinefile machi.dat allgather.bin  
3 [0] 0 -> 0.000000  
4 [0] 1 -> 1.000000  
5 [0] 2 -> 2.000000  
6 [0] 3 -> 1000.000000  
7 [0] 4 -> 1001.000000  
8 [0] 5 -> 1002.000000  
9 [0] 6 -> 2000.000000  
10 [0] 7 -> 2001.000000  
11 [0] 8 -> 2002.000000  
12 [1] 0 -> 0.000000  
13 [1] 1 -> 1.000000  
14 [1] 2 -> 2.000000  
15 [1] 3 -> 1000.000000  
16 [1] 4 -> 1001.000000  
17 [1] 5 -> 1002.000000  
18 [1] 6 -> 2000.000000  
19 [1] 7 -> 2001.000000  
20 [1] 8 -> 2002.000000  
21 [2] 0 -> 0.000000  
22 [2] 1 -> 1.000000  
23 [2] 2 -> 2.000000  
24 [2] 3 -> 1000.000000  
25 [2] 4 -> 1001.000000  
26 [2] 5 -> 1002.000000  
27 [2] 6 -> 2000.000000  
28 [2] 7 -> 2001.000000  
29 [2] 8 -> 2002.000000  
30 [mstorti@spider example]$
```

All-to-all operation

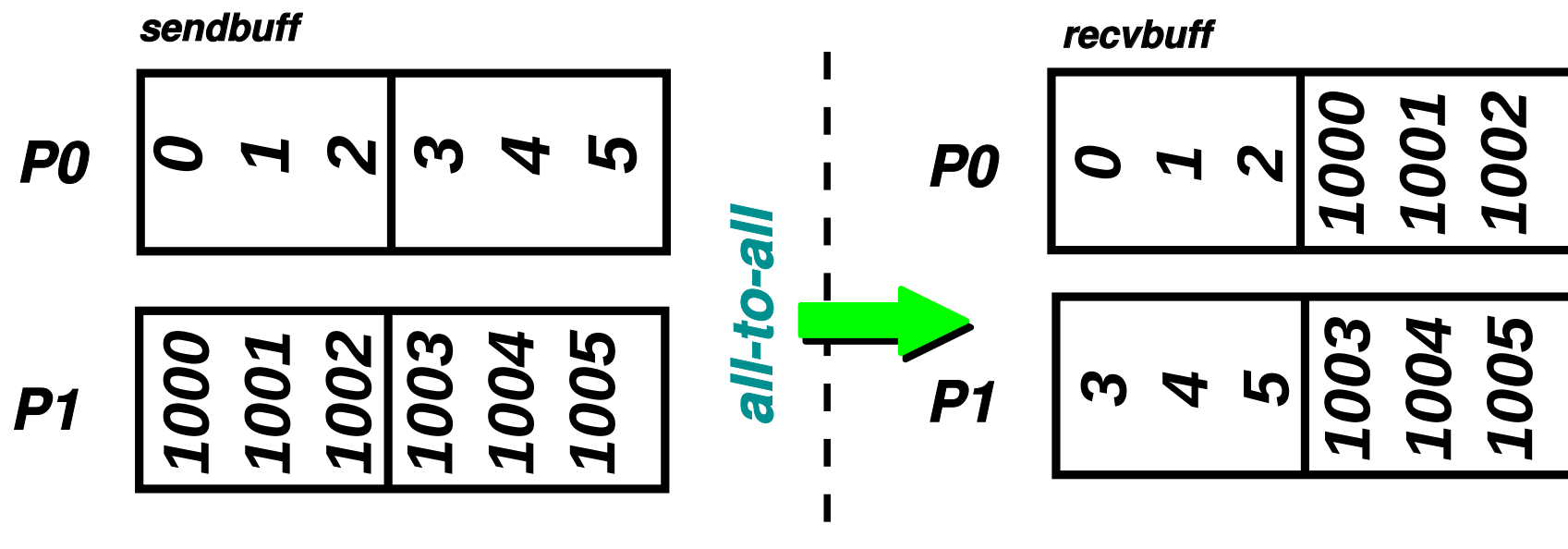
- Its conceptually equivalent to a scatter from P_0 followed by a scatter from P_1 , etc...
- Or either a gather to P_0 , followed by a gather to P_1 , and so on...



All-to-all operation (cont.)

```
1  #include <mpi.h>
2  #include <stdio>
3
4  int main(int argc, char **argv) {
5      MPI_Init(&argc, &argv);
6      int myrank, size;
7      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10     int N = 3;           // Nbr of elements to send to each processor
11     double *sbuff = new double[size*N]; // send buffer in all procs
12     for (int j=0; j<size*N; j++) sbuff[j] = myrank*1000.0+j;
13
14     double *rbuff = new double[N*size]; // receive buffer in all procs
15
16     MPI_Alltoall(sbuff, N, MPI_DOUBLE,
17                 rbuff, N, MPI_DOUBLE, MPI_COMM_WORLD);
18
19     for (int j=0; j<N*size; j++)
20         printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
21     MPI_Finalize();
22
23     delete[] sbuff;
24     delete[] rbuff;
25 }
```

All-to-all operation (cont.)

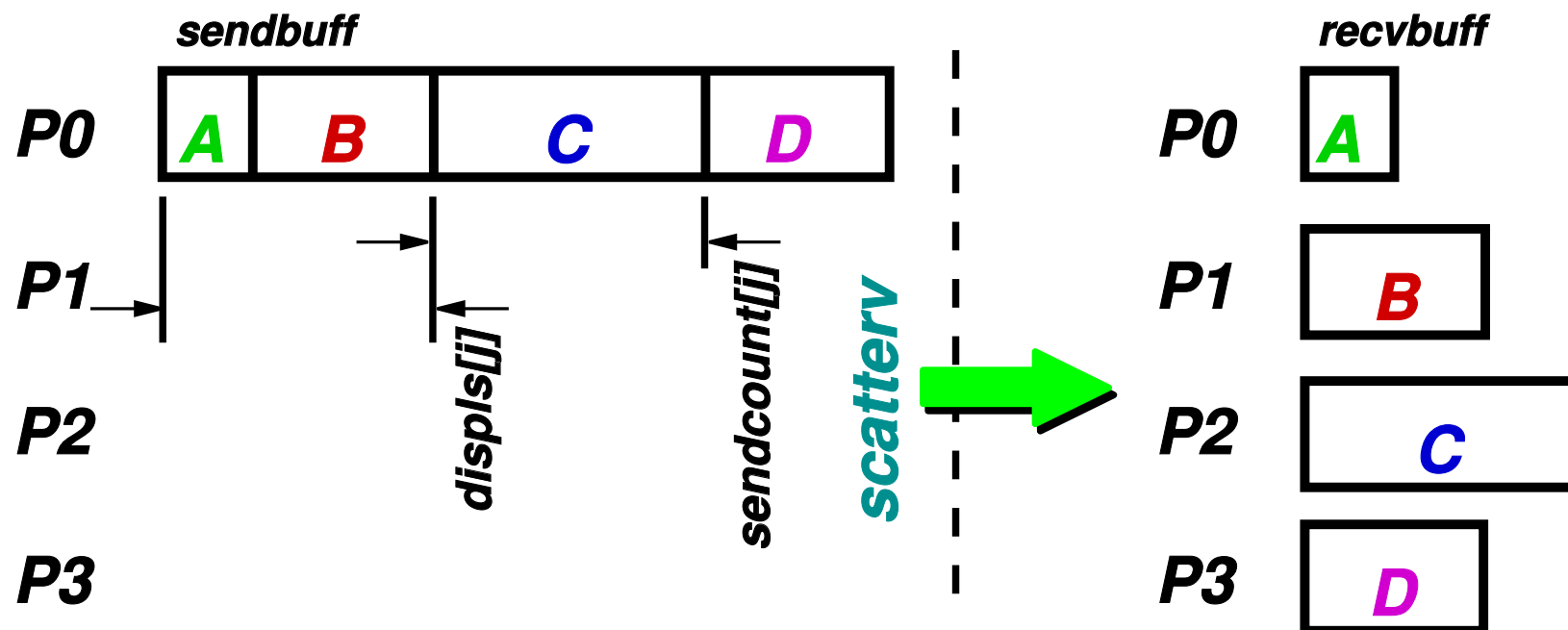


All-to-all operation (cont.)

```
1 [mstorti@spider example]$ mpirun -np 2 \  
2                               -machinefile machi.dat alltoall.bin  
3 [0] 0 -> 0.000000  
4 [0] 1 -> 1.000000  
5 [0] 2 -> 2.000000  
6 [0] 3 -> 1000.000000  
7 [0] 4 -> 1001.000000  
8 [0] 5 -> 1002.000000  
9 [1] 0 -> 3.000000  
10 [1] 1 -> 4.000000  
11 [1] 2 -> 5.000000  
12 [1] 3 -> 1003.000000  
13 [1] 4 -> 1004.000000  
14 [1] 5 -> 1005.000000  
15 [mstorti@spider example]$
```

Vector scatter (variable length)

It is conceptually equivalent to a `MPI_Scatter()` but allows that the length of data send to each processor may be different.



Vector scatter (variable length) (cont.)

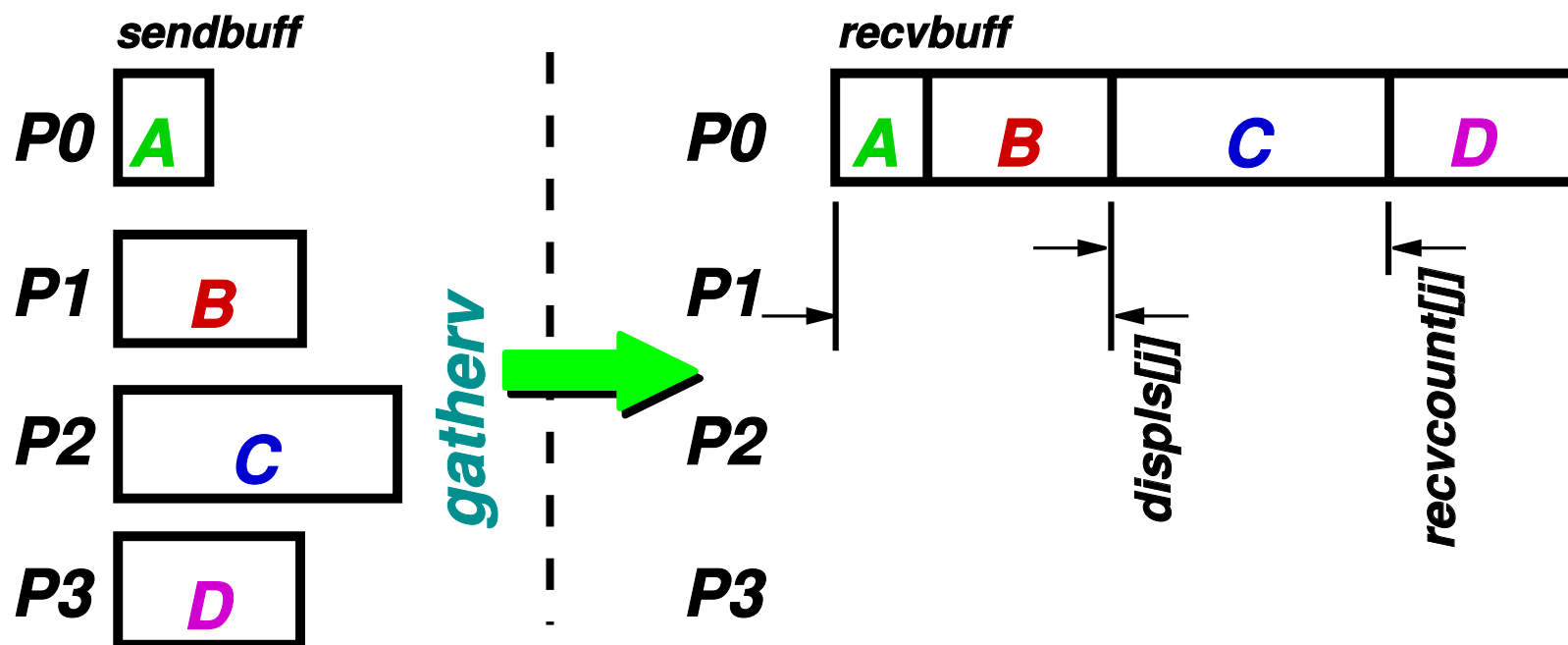
```
1  int N = size*(size+1)/2;
2  double *sbuff = NULL;
3  int *sendcnts = NULL;
4  int *displs = NULL;
5  if (!myrank) {
6      sbuff = new double[N]; // send buffer only in master
7      for (int j=0; j<N; j++) sbuff[j] = j; // fills 'sbuff'
8
9      sendcnts = new int[size];
10     displs = new int[size];
11     for (int j=0; j<size; j++) sendcnts[j] = (j+1);
12     displs[0]=0;
13     for (int j=1; j<size; j++)
14         displs[j] = displs[j-1] + sendcnts[j-1];
15 }
16
17 // receive buffer in all procs
18 double *rbuff = new double[myrank+1];
19
20 MPI_Scatterv(sbuff, sendcnts, displs, MPI_DOUBLE,
21             rbuff, myrank+1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
22
23 for (int j=0; j<myrank+1; j++)
24     printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
```


Vector scatter (variable length) (cont.)

```
1 [mstorti@spider example]$ mpirun -np 4 \  
2                               -machinefile machi.dat scatterv.bin  
3 [0] 0 -> 0.000000  
4 [3] 0 -> 6.000000  
5 [3] 1 -> 7.000000  
6 [3] 2 -> 8.000000  
7 [3] 3 -> 9.000000  
8 [1] 0 -> 1.000000  
9 [1] 1 -> 2.000000  
10 [2] 0 -> 3.000000  
11 [2] 1 -> 4.000000  
12 [2] 2 -> 5.000000  
13 [mstorti@spider example]$
```

Gatherv operation

Is the same as *gather*, but each processor receives data of different length.



Gatherv operation (cont.)

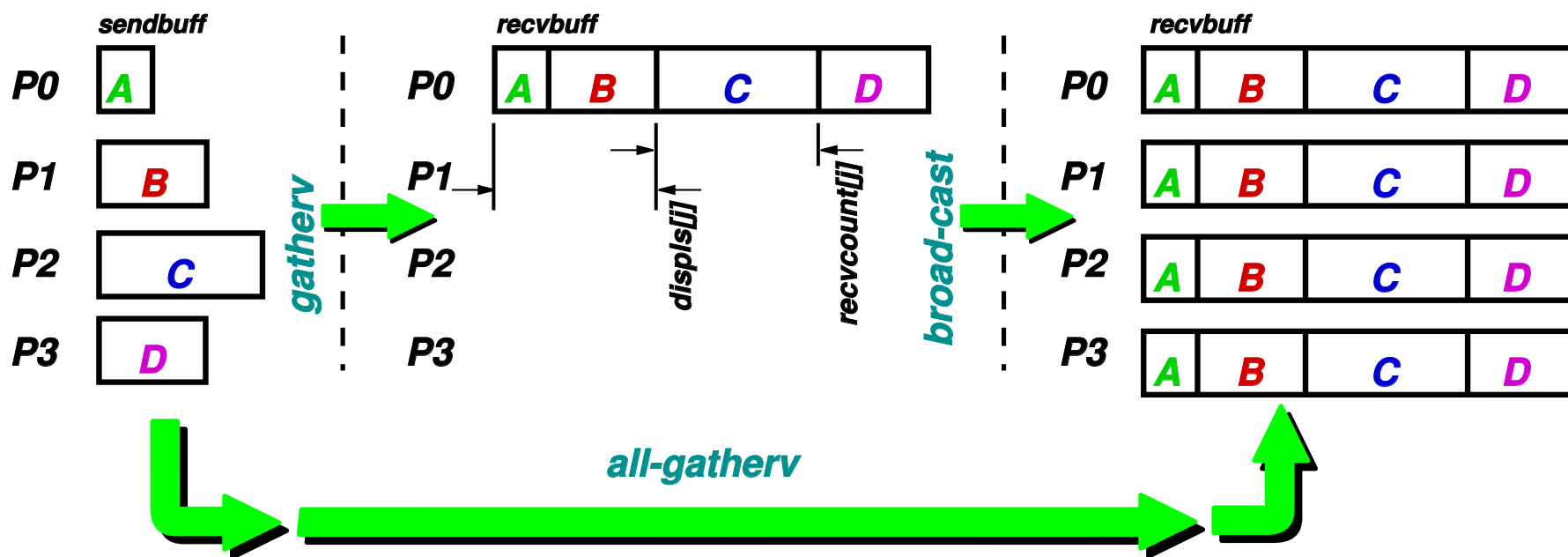
```
1  int sendcnt = myrank+1; // send buffer in all
2  double *sbuff = new double[myrank+1];
3  for (int j=0; j<sendcnt; j++)
4      sbuff[j] = myrank*1000+j;
5
6  int rsize = size*(size+1)/2;
7  int *recvcnts = NULL;
8  int *displs = NULL;
9  double *rbuff = NULL;
10 if (!myrank) {
11     // receive buffer and ptrs only in master
12     rbuff = new double[rsize]; // recv buffer only in master
13     recvcnts = new int[size];
14     displs = new int[size];
15     for (int j=0; j<size; j++) recvcnts[j] = (j+1);
16     displs[0]=0;
17     for (int j=1; j<size; j++)
18         displs[j] = displs[j-1] + recvcnts[j-1];
19 }
20
21 MPI_Gatherv(sbuff, sendcnt, MPI_DOUBLE,
22             rbuff, recvcnts, displs, MPI_DOUBLE,
23             0, MPI_COMM_WORLD);
```

Gatherv operation (cont.)

```
1 [mstorti@spider example]$ mpirun -np 4 \  
2                               -machinefile machi.dat gatherv.bin  
3 0 -> 0.000000  
4 1 -> 1000.000000  
5 2 -> 1001.000000  
6 3 -> 2000.000000  
7 4 -> 2001.000000  
8 5 -> 2002.000000  
9 6 -> 3000.000000  
10 7 -> 3001.000000  
11 8 -> 3002.000000  
12 9 -> 3003.000000  
13 [mstorti@spider example]$
```

Allgatherv operation

Is the same as *gatherv*, followed by a *broadcast*.



Allgatherv operation (cont.)

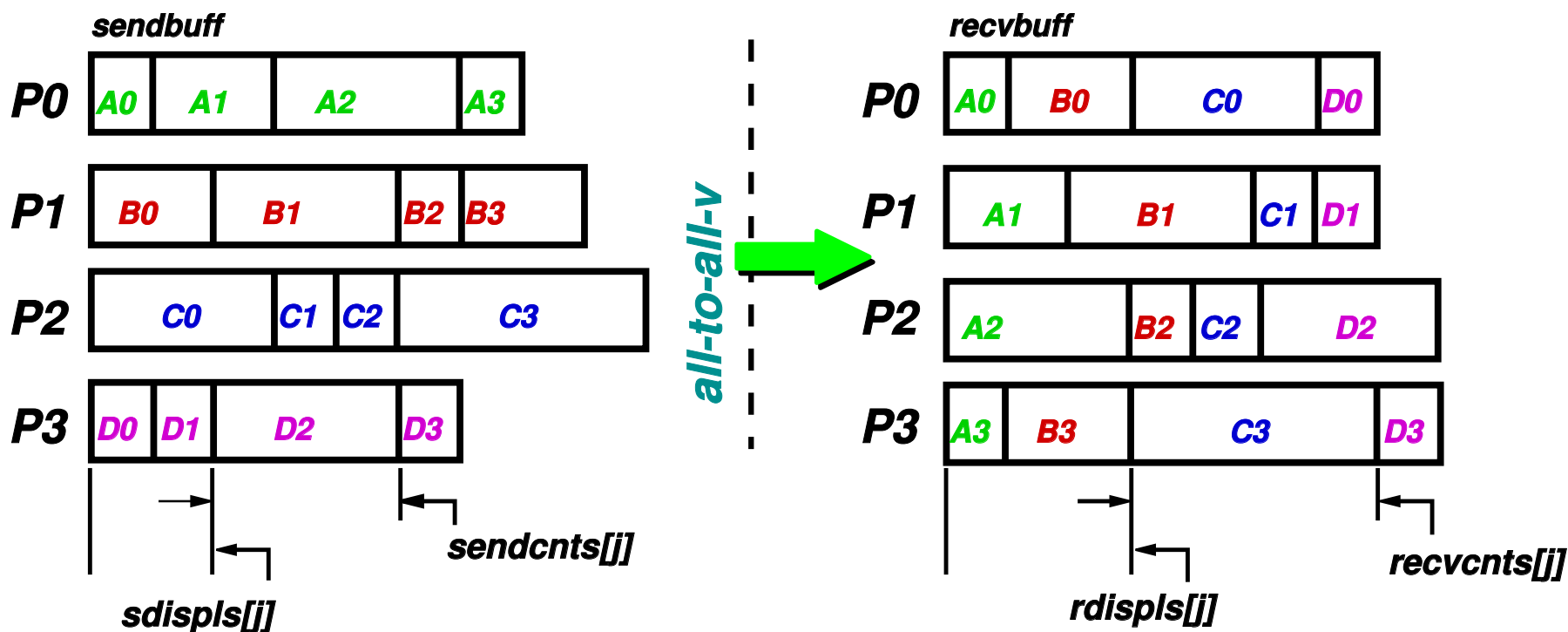
```
1  int sendcnt = myrank+1; // send buffer in all
2  double *sbuff = new double[myrank+1];
3  for (int j=0; j<sendcnt; j++)
4      sbuff[j] = myrank*1000+j;
5
6  // receive buffer and ptrs in all
7  int rsize = size*(size+1)/2;
8  double *rbuff = new double[rsize];
9  int *recvcnts = new int[size];
10 int *displs = new int[size];
11 for (int j=0; j<size; j++) recvcnts[j] = (j+1);
12 displs[0]=0;
13 for (int j=1; j<size; j++)
14     displs[j] = displs[j-1] + recvcnts[j-1];
15
16 MPI_Allgatherv(sbuff, sendcnt, MPI_DOUBLE,
17               rbuff, recvcnts, displs, MPI_DOUBLE,
18               MPI_COMM_WORLD);
```

Allgatherv operation (cont.)

```
1 [mstorti@spider example]$ mpirun -np 3 \  
2     -machinefile machi.dat allgatherv.bin  
3 [0] 0 -> 0.000000  
4 [0] 1 -> 1000.000000  
5 [0] 2 -> 1001.000000  
6 [0] 3 -> 2000.000000  
7 [0] 4 -> 2001.000000  
8 [0] 5 -> 2002.000000  
9 [1] 0 -> 0.000000  
10 [1] 1 -> 1000.000000  
11 [1] 2 -> 1001.000000  
12 [1] 3 -> 2000.000000  
13 [1] 4 -> 2001.000000  
14 [1] 5 -> 2002.000000  
15 [2] 0 -> 0.000000  
16 [2] 1 -> 1000.000000  
17 [2] 2 -> 1001.000000  
18 [2] 3 -> 2000.000000  
19 [2] 4 -> 2001.000000  
20 [2] 5 -> 2002.000000  
21 [mstorti@spider example]$
```

All-to-all-v operation

Vectorized version (variable length data) of `MPI_Alltoall()`.



All-to-all-v operation (cont.)

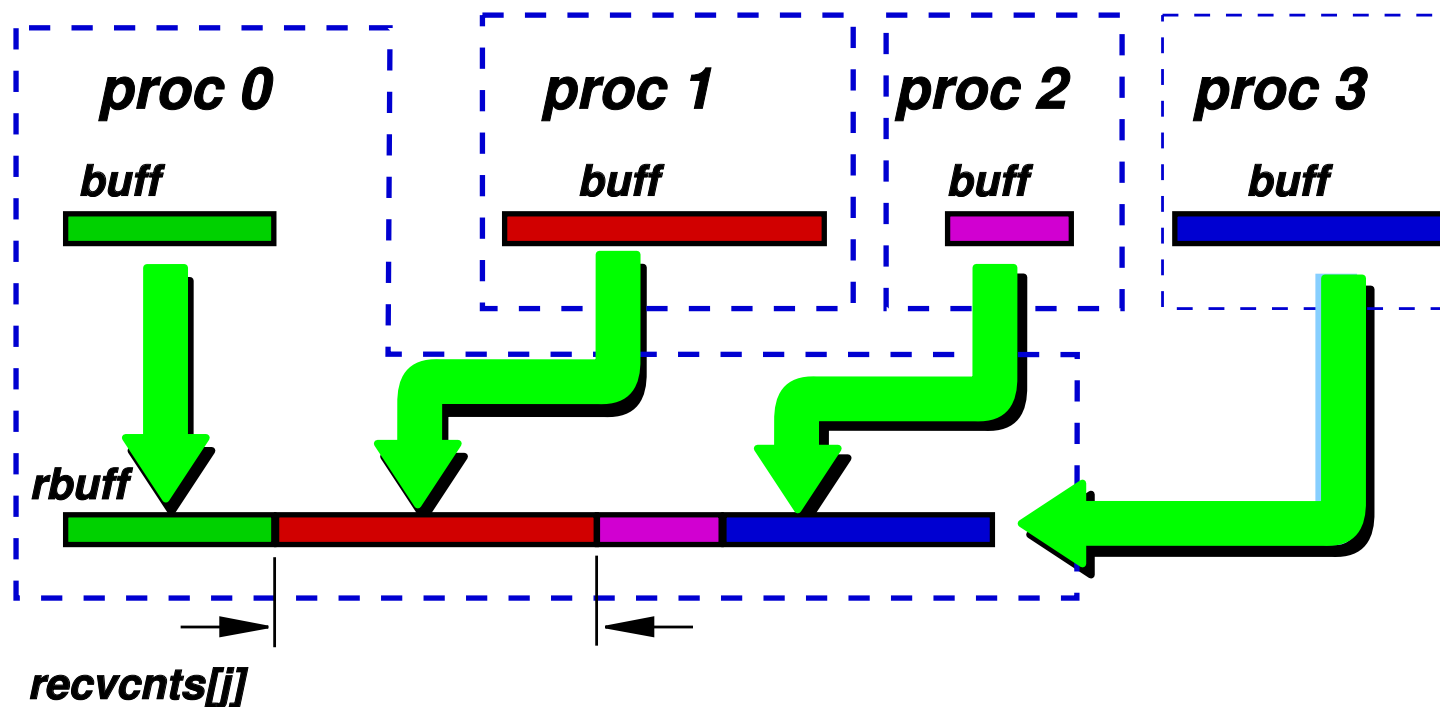
```
1  // vectorized send buffer in all
2  int ssize = (myrank+1)*size;
3  double *sbuff = new double[ssize];
4  int *sendcnts = new int[size];
5  int *sdispls = new int[size];
6  for (int j=0; j<ssize; j++)
7      sbuff[j] = myrank*1000+j;
8  for (int j=0; j<size; j++) sendcnts[j] = (myrank+1);
9  sdispls[0]=0;
10 for (int j=1; j<size; j++)
11     sdispls[j] = sdispls[j-1] + sendcnts[j-1];
12
13 // vectorized receive buffer and ptrs in all
14 int rsize = size*(size+1)/2;
15 double *rbuff = new double[rsize];
16 int *recvcnts = new int[size];
17 int *rdispls = new int[size];
18 for (int j=0; j<size; j++) recvcnts[j] = (j+1);
19 rdispls[0]=0;
20 for (int j=1; j<size; j++)
21     rdispls[j] = rdispls[j-1] + recvcnts[j-1];
22
23 MPI_Alltoallv(sbuff, sendcnts, sdispls, MPI_DOUBLE,
24              rbuff, recvcnts, rdispls, MPI_DOUBLE,
25              MPI_COMM_WORLD);
```

All-to-all-v operation (cont.)

```
1 [mstorti@spider example]$ mpirun -np 3 \  
2                               -machinefile machi.dat alltoallv.bin  
3 [0] 0 -> 0.000000  
4 [0] 1 -> 1000.000000  
5 [0] 2 -> 1001.000000  
6 [0] 3 -> 2000.000000  
7 [0] 4 -> 2001.000000  
8 [0] 5 -> 2002.000000  
9 [1] 0 -> 1.000000  
10 [1] 1 -> 1002.000000  
11 [1] 2 -> 1003.000000  
12 [1] 3 -> 2003.000000  
13 [1] 4 -> 2004.000000  
14 [1] 5 -> 2005.000000  
15 [2] 0 -> 2.000000  
16 [2] 1 -> 1004.000000  
17 [2] 2 -> 1005.000000  
18 [2] 3 -> 2006.000000  
19 [2] 4 -> 2007.000000  
20 [2] 5 -> 2008.000000  
21 [mstorti@spider example]$
```

The print-par function

As an example, let's write a function `print_par()` that prints the content of a buffer of variable length (per processor).



The print-par function (cont.)

```
1 void print_par(vector<int> &buff, const char *s=NULL) {
2
3     int sendcnt = buff.size();
4     vector<int> recvcnts(size);
5     MPI_Gather(sendcnt, . . . , recvcnts, . . . );
6
7     int rsize = /* sum of recvcnts[] */;
8     vector<int> buff(rsize);
9
10    vector<int> displs;
11    displs = /* cum-sum of recvcnts[] */;
12
13    MPI_Gatherv(buff, sendcnt . . . . ,
14               rbuff, recvcnts, displs, . . . );
15    if (!myrank) {
16        for (int rank=0; rank<size; rank++) {
17            // print elements belonging to
18            // processor 'rank' . . .
19        }
20    }
21 }
```

The print-par function (cont.)

- Each processor has a *vector<int> buff* containing elements. We write a function that prints on stdout all elements on processor 0, then on proc 1, and so on...
- The size of *buff* can be different on each processor.
- We first do a *gather* of the sizes of the local vectors to *recvnts[]*. With this information we compute the *displacements displs[]*.
- The sum of the *recvnts[]* gives us the size of the reception buffer on processor 0.
- Note that this could perhaps be done more efficiently by sending the data from the slaves to the master one by one, since in this form it is not necessary to allocate a buffer with the size of the sum of the sizes on all processors. It will only need a buffer with the size of the largest buffer in the slaves.

The print-par function (cont.)

```

1 void print_par(vector<int> &buff, const char *s=NULL) {
2     int myrank, size;
3     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
4     MPI_Comm_size(MPI_COMM_WORLD, &size);
5
6     // reception buffer in master, receive counts
7     // and displacements
8     vector<int> rbuff, recvcnts(size, 0), displs(size, 0);
9
10    // Each procesor send its size
11    int sendcnt = buff.size();
12    MPI_Gather(&sendcnt, 1, MPI_INT,
13              &recvcnts[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
14    if (!myrank) {
15        // Resize reception buffer and
16        // compute displs[] in master
17        int rsize = 0;
18        for (int j=0; j<size; j++) rsize += recvcnts[j];
19        rbuff.resize(rsize);
20        displs[0] = 0;
21        for (int j=1; j<size; j++)
22            displs[j] = displs[j-1] + recvcnts[j-1];
23    }
24    // Do the gather
25    MPI_Gatherv(&buff[0], sendcnt, MPI_INT,
26               &rbuff[0], &recvcnts[0], &displs[0], MPI_INT,
27
```

```
28         0, MPI_COMM_WORLD);
29     if (!myrank) {
30         // Print all buffers in master
31         if (s) printf("%s", s);
32         for (int j=0; j<size; j++) {
33             printf("in proc [%d]: ", j);
34             for (int k=0; k<recvnts[j]; k++) {
35                 int ptr = displs[j]+k;
36                 printf("%d ", rbuff[ptr]);
37             }
38             printf("\n");
39         }
40     }
41 }
```

Example of all-to-all-v rescatter

As another example, consider the case where we have a certain amount of objects (for simplicity we will assume an array of vectors), and we want to write a function

```
1 void re_scatter(vector<int> &buff, int size,  
2               int myrank, proc_fun proc, void *data);
```

that redistributes the elements that are in *buff* in each processor according to the criterion given by the function *f*. This function *f* returns the number of processor for each element of the array. The signature of this kind of functions is given by the following *typedef*

```
1 typedef int (*proc_fun) (int x, int size, void *data);
```

For instance, if we want that each processor receives those elements *x* that are $x \% size == myrank$ then we have to use

```
1 int mod_scatter(int x, int size, void *data) {  
2     return x % size;  
3 }
```


Example of all-to-all-v rescatter (cont.)

If

- $buff = \{3, 4, 2, 5, 4\}$ in P_0
- $buff = \{3, 5, 2, 1, 3, 2\}$ in P_1
- $buff = \{7, 9, 10\}$ en P_2

```
1 int mod_scatter(int x, int size, void *data) {  
2     return x % size;  
3 }
```

then after $re_scatter(buff, \dots, mod_scatter, \dots)$ we should have

- $buff = \{3, 3, 3, 9\}$ in P_0
- $buff = \{4, 4, 1, 7, 10\}$ in P_1
- $buff = \{2, 5, 5, 2, 2\}$ in P_2

Example of all-to-all-v rescatter (cont.)

We can pass global parameters to *mod_scatter*,

```
1 int k;
2 int mod_scatter(int x, int size, void *data) {
3     return (x-k) % size;
4 }
```

so that:

```
1 //-----
2 // Initially:
3 // buff={3,4,2,5,4} en P0, {3,5,2,1,3,2} en P1, {7,9,10} en P2.
4
5 k = 0; re_scatter(buff, . . . , mod_scatter, . . . );
6 // buff={3,3,3,9} en P0, {4,4,1,7,10} en P1, {2,5,5,2,2} en P2
7
8 k = 1; re_scatter(buff, . . . , mod_scatter, . . . );
9 // buff={4,4,1,7,10} en P0, {2,5,5,2,2} en P1, {3,3,3,9} en P2
10
11 k = 2; re_scatter(buff, . . . , mod_scatter, . . . );
12 // buff={2,5,5,2,2} en P0, {3,3,3,9} en P1, {4,4,1,7,10} en P2
```

Example of all-to-all-v rescatter (cont.)

The argument *void *data* allows to *pass arguments* to the function, avoiding the use of global variables. For instance if we want that the distribution of type *proc = x % size* rotates, i.e. *proc = (x-k) % size*, with a varying *k*, then we must be able to pass *k* to *f*, this is done via the *void *data* argument in the following way

```

1  int rotate_elems(int elem, int size, void *data) {
2      int key = *(int *)data;
3      return (elem - key) % size;
4  }
5  // Fill 'buff'
6  ...
7  for (int k=0; k<kmax; k++)
8      re_scatter(buff, size, myrank, mod_scatter, &k);
9  // print elements ....
10 }
```

Example of all-to-all-v rescatter (cont.)

```
1 [mstorti@spider example]$ mpirun -np 4 \  
2                               -machinefile machi.dat rescatter.bin  
3 initial:  
4 [0]: 383 886 777 915 793 335 386 492 649 421  
5 [1]:  
6 [2]:  
7 [3]:  
8 after rescatter: -----  
9 [0]: 492  
10 [1]: 777 793 649 421  
11 [2]: 886 386  
12 [3]: 383 915 335  
13 after rescatter: -----  
14 [0]: 777 793 649 421  
15 [1]: 886 386  
16 [2]: 383 915 335  
17 [3]: 492  
18 after rescatter: -----  
19 [0]: 886 386  
20 [1]: 383 915 335  
21 [2]: 492  
22 [3]: 777 793 649 421  
23 ...
```

Example of all-to-all-v rescatter (cont.)

This is an example of *Functional Programming (FP)* technique

- Some languages are more friendly with the functional programming style, for instance they allow to use functions as objects: they allow to create and destroy them, even in execution time, pass them as arguments, and so on....
- Certain languages give a full support to FP: Haskell, ML, Scheme, Lisp, and in lesser extent Perl, Python, C/C++.
- In *C++* define the kind of argument functions via a *typedef* as follows

```
1 typedef int (*proc_fun) (int x, int size, void *data);
```

This defines a special type of functions (the *proc_fun* type) that are those that take as arguments two integers and a generic pointer *void ** and return an integer.

- We can define functions with this *signature* (argument types and return type) and pass them as objects to *higher order procedures*.

```
1 void re_scatter(vector<int> &buff, int size,  
2               int myrank, proc_fun proc, void *data);
```

Example of all-to-all-v rescatter (cont.)

Examples:

- Sort with a comparison function:

```
1 int (*comp) (int x, int y);  
2 void sort (int *a, comp comp_f);
```

For instance, sort by absolute value:

```
1 int comp_abs (int x, int y) {  
2     return abs (x) < abs (y);  
3 }  
4 // fill array 'a' with values ...  
5 sort (a, comp_abs);
```

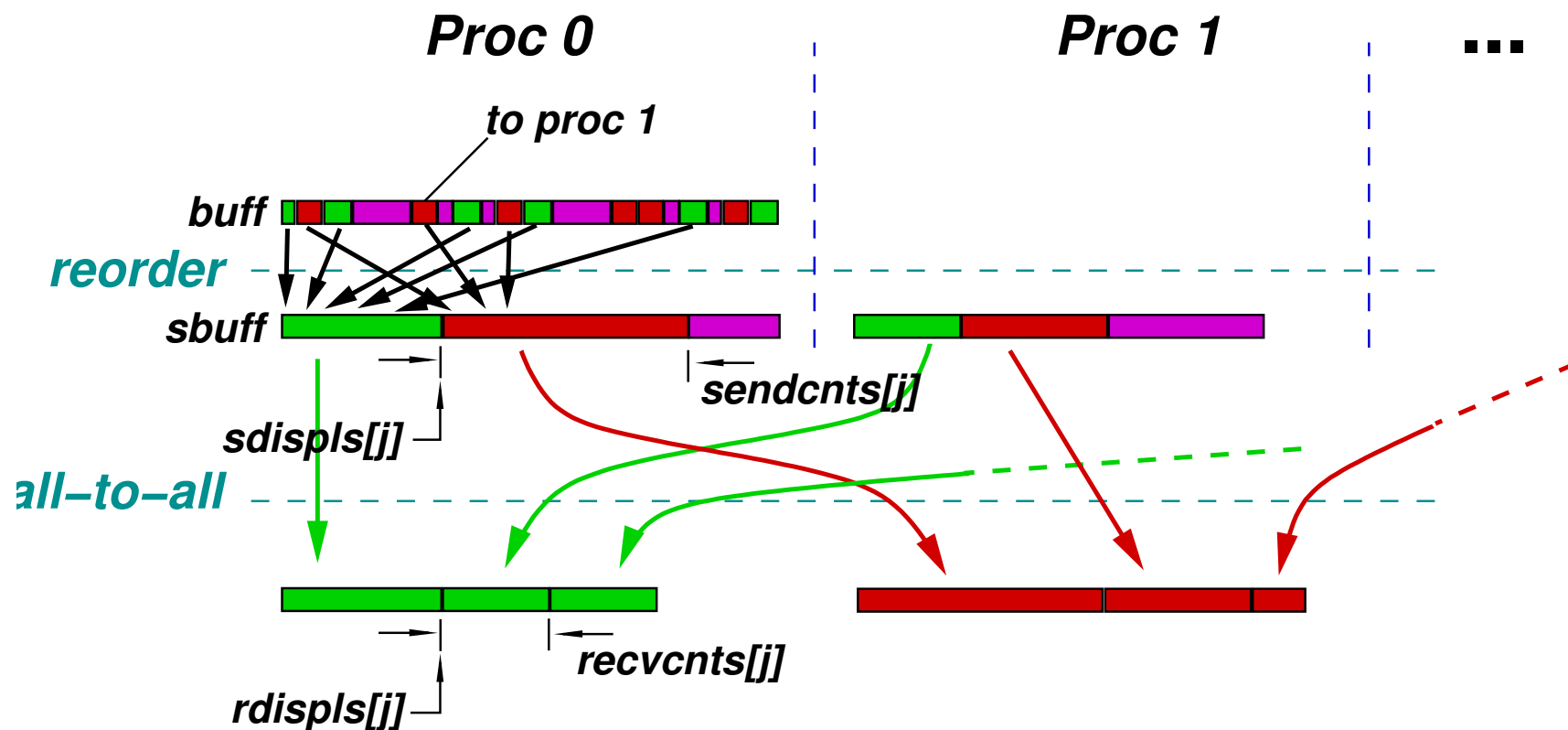
- Filter a list (leave only objects that satisfy a given predicate):

```
1 int (*pred) (int x);  
2 void filter (int *a, pred filter);
```

For instance, eliminate the even values, leave only the odd ones),

```
1 int odd (int x) {  
2     return x % 2 != 0;  
3 }  
4 // fill array 'a' with values ...  
5 filter (a, odd);
```

Example of all-to-all-v rescatter (cont.)



Example of all-to-all-v rescatter (cont.)

Seudo-código:

```

1 void re_scatter(vector<int> &buff, int size,
2               int myrank, proc_fun proc, void *data) {
3     int N = buff.size();
4     // Check how many elements should go to
5     // each processor
6     for (int j=0; j<N; j++) {
7         int p = proc(buff[j], size, data);
8         sendcnts[p]++;
9     }
10    // Allocate sbuff and reorder (buff -> sbuff)...
11    // Compute all 'send' displacements
12    for (int j=1; j<size; j++)
13        sdispls[j] = sdispls[j-1] + sendcnts[j-1];
14
15    // Use 'Alltoall' for scattering the dimensions
16    // of the buffers to be received
17    MPI_Alltoall(&sendcnts[0], 1, MPI_INT,
18               &recvcnts[0], 1, MPI_INT, MPI_COMM_WORLD);
19
20    // Compute receive size 'rsize' and 'rdispls' from 'recvcnts'...
21    // resize 'buff' to 'rsize'...
22
23    MPI_Alltoallv(&sbuff[0], &sendcnts[0], &sdispls[0], MPI_INT,
24                 &buff[0], &recvcnts[0], &rdispls[0], MPI_INT,
25                 MPI_COMM_WORLD);

```


26 }

Example of all-to-all-v rescatter (cont.)

Complete code:

```

1 void re_scatter(vector<int> &buff, int size,
2               int myrank, proc_fun proc, void *data) {
3     vector<int> sendcnts(size, 0);
4     int N = buff.size();
5     // Check how many elements should go to
6     // each processor
7     for (int j=0; j<N; j++) {
8         int p = proc(buff[j], size, data);
9         sendcnts[p]++;
10    }
11
12    // Dimension buffers and ptr vectors
13    vector<int> sbuff(N);
14    vector<int> sdispls(size);
15    vector<int> recvcnts(size);
16    vector<int> rdispls(size);
17    sdispls[0] = 0;
18    for (int j=1; j<size; j++)
19        sdispls[j] = sdispls[j-1] + sendcnts[j-1];
20
21    // Reorder by processor from buff to sbuff
22    for (int j=0; j<N; j++) {
23        int p = proc(buff[j], size, data);
24        int pos = sdispls[p];
25        sbuff[pos] = buff[j];

```

```
26     sdispls[p]++;
27 }
28 // Use 'Alltoall' for scattering the dimensions
29 // of the buffers to be received
30 MPI_Alltoall(&sendcnts[0],1,MPI_INT,
31             &recvcnts[0],1,MPI_INT,MPI_COMM_WORLD);
32
33 // Compute the 'send' and 'recv' displacements.
34 rdispls[0] = 0;
35 sdispls[0] = 0;
36 for (int j=1; j<size; j++) {
37     rdispls[j] = rdispls[j-1] + recvcnts[j-1];
38     sdispls[j] = sdispls[j-1] + sendcnts[j-1];
39 }
40
41 // Dimension the receive size
42 int rsize = 0;
43 for (int j=0; j<size; j++) rsize += recvcnts[j];
44 buff.resize(rsize);
45
46 // Do the scatter.
47 MPI_Alltoallv(&sbuff[0],&sendcnts[0],&sdispls[0],MPI_INT,
48              &buff[0],&recvcnts[0],&rdispls[0],MPI_INT,
49              MPI_COMM_WORLD);
50 }
```

Definiendo tipos de datos derivados

Definiendo tipos de datos derivados

Suppose we have a the coefficients of a matrix $N \times N$ stored in an array `double a[N*N]`. Row j consists in N doubles that are stored in positions `[j*N, (j+1)*N)` in array `a`.

If we want to send row j from processor `source` to another processor `dest` to replace row k , then we can use `MPI_Send()` and `MPI_Recv()` since the rows represent adjacent values

```
1 if (myrank==source)
2   MPI_Send(&a[j*N], N, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
3 else if (myrank==dest)
4   MPI_Recv(&a[k*N], N, MPI_DOUBLE, source, 0,
5           MPI_COMM_WORLD, &status);
```

Definiendo tipos de datos derivados (cont.)

If we can send *columns*, then the problem is harder, since, as en *C++* the coefficients are stored by row, the elements in the columns are separated by *N* elements. The obvious possibility is to create a temporary buffer *double buff[N]* where we gather the elements in adjacent positions and then we send/receive them.

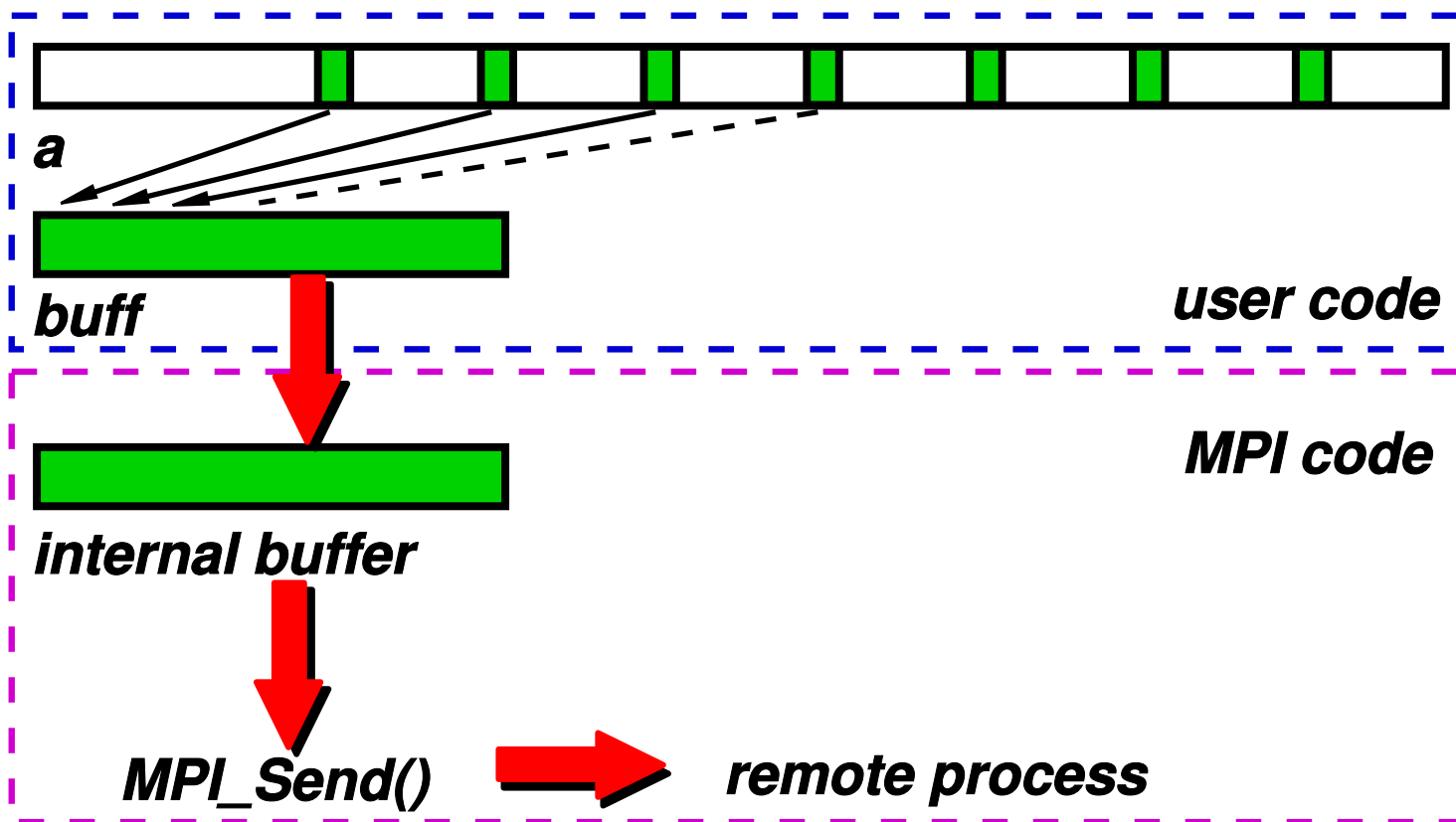
```

1 double *buff = new double[N];
2 if (myrank==source) {
3     // Gather column in 'buff' and send
4     for (int l=0; l<N; l++)
5         buff[l] = a[l*N+j];
6     MPI_Send(buff, N, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
7 } else if (myrank==dest) {
8     // Receive 'buff' and put data in column
9     MPI_Recv(buff, N, MPI_DOUBLE, source, 0,
10             MPI_COMM_WORLD, &status);
11     for (int l=0; l<N; l++)
12         a[l*N+k] = buff[l];
13 }
14 delete[] buff;

```

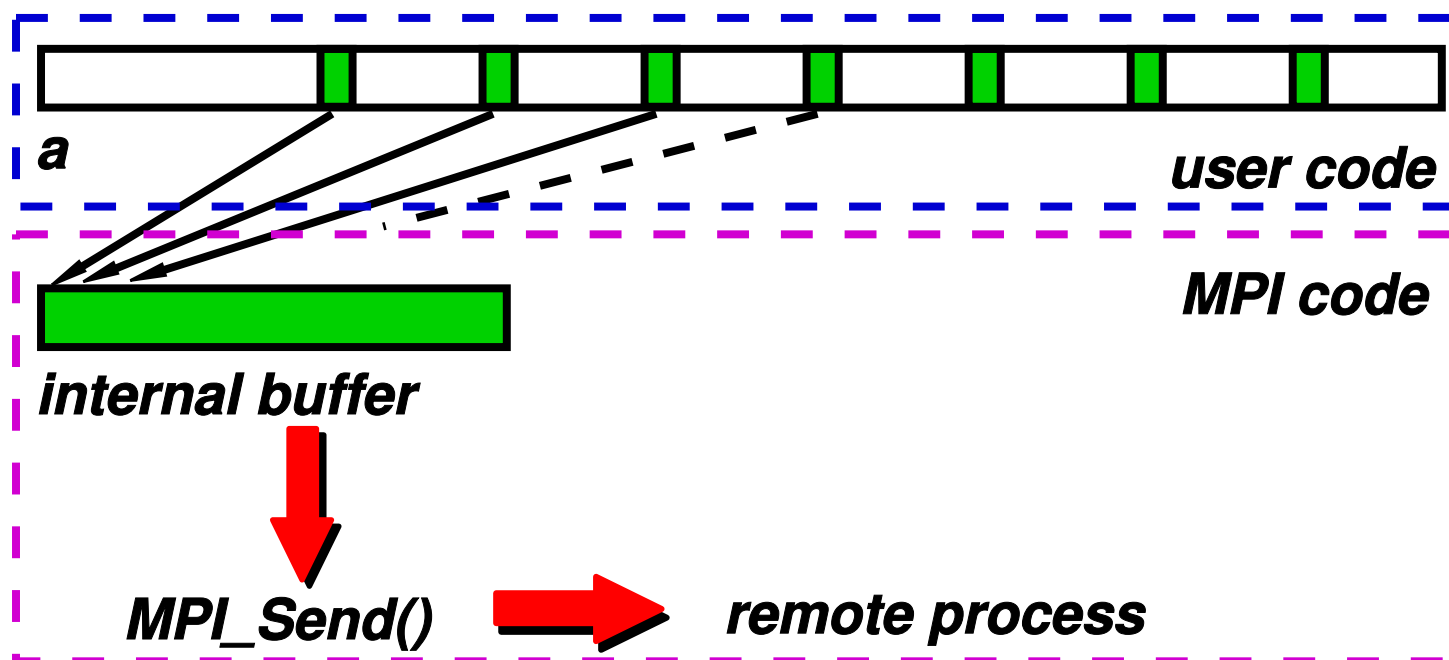
Definiendo tipos de datos derivados (cont.)

This has the problem that requires an additional buffer with the same size as the data to be sent. In addition the time overhead associated to copy these data to the buffer.



Definiendo tipos de datos derivados (cont.)

The idea is to avoid the use of the auxiliary *buff*, passing the data directly to MPI.



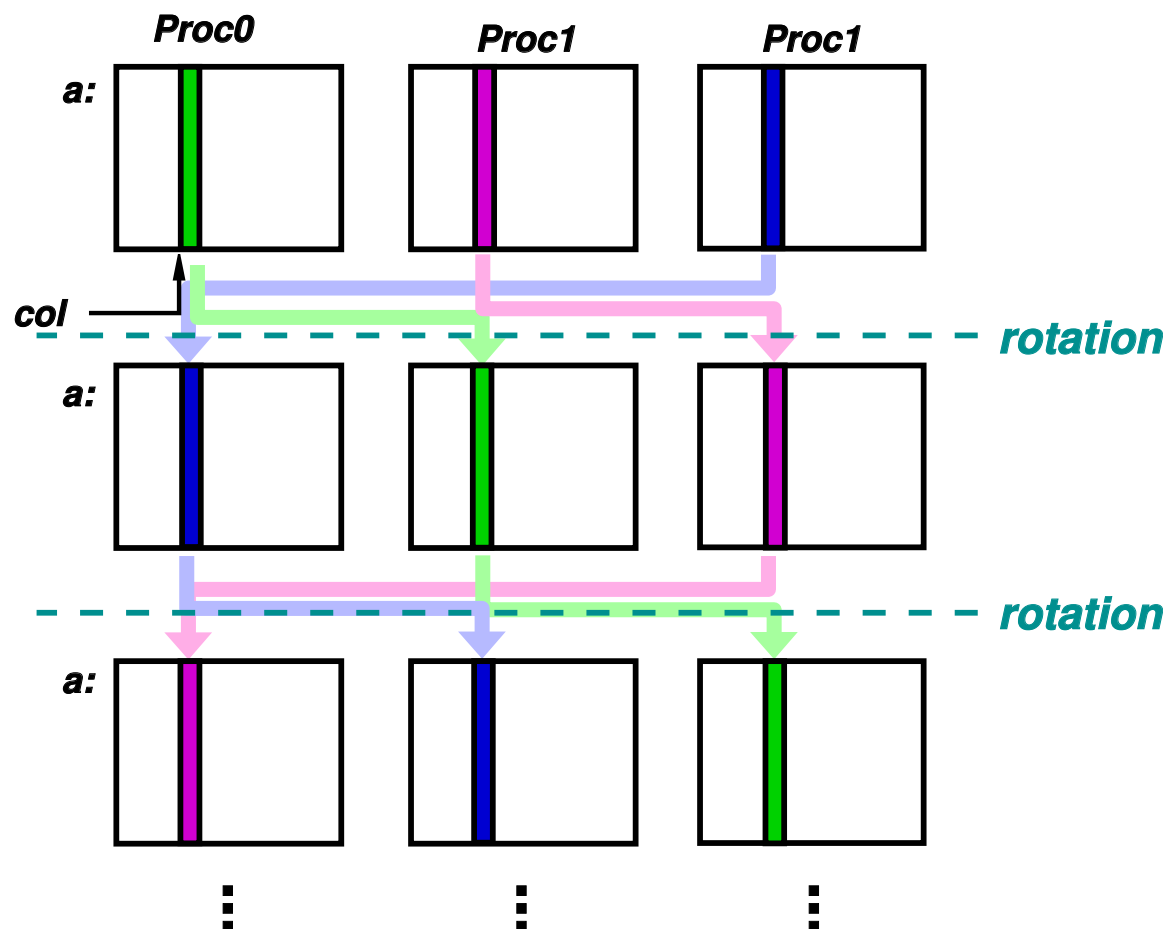
Definiendo tipos de datos derivados (cont.)

This is done by defining what is called an *MPI derived data type*.

```
1 MPI_Datatype stride;  
2 MPI_Type_vector(N, 1, N, MPI_DOUBLE, &stride);  
3 MPI_Type_commit(&stride);  
4  
5 // use 'stride' ...  
6  
7 // Free resources reserved for type 'stride'  
8 MPI_Type_free(&stride);
```

Ejemplo: rotar columna de una matriz

The following program rotates the columns $col=2$ from A between the processors.



Ejemplo: rotar columna de una matriz (cont.)

```
1  int col = 2;
2  MPI_Status status;
3  int dest = myrank+1;
4  if (dest==size) dest = 0;
5  int source = myrank-1;
6  if (source==-1) source = size-1;
7  vector<double> recv_val(N);
8
9  MPI_Datatype stride;
10 MPI_Type_vector(N,1,N,MPI_DOUBLE,&stride);
11 MPI_Type_commit(&stride);
12
13 for (int k=0; k<10; k++) {
14     MPI_Sendrecv(&A(0,col),1,stride,dest,0,
15                 &recv_val[0],N,MPI_DOUBLE,source,0,
16                 MPI_COMM_WORLD,&status);
17     for (int j=0; j<N; j++)
18         A(j,col) = recv_val[j];
19     if (!myrank) {
20         printf("After rotation step %d\n",k);
21         print_mat(a,N);
22     }
23 }
24 MPI_Type_free(&stride);
25 MPI_Finalize();
```

Ejemplo: rotar columna de una matriz (cont.)

Between the rotation

	Before rotation		Proc [1] a:		Proc [2] a:
1	Proc [0] a:				
3	4.8 3.2 [0.2] 9.9		7.2 3.3 [9.9] 1.7		3.9 5.6 [9.5] 5.5
4	7.9 8.8 [1.7] 7.2		8.9 2.2 [9.8] 2.3		5.1 3.3 [8.9] 1.2
5	2.7 4.2 [2.0] 4.1		6.7 8.5 [5.7] 3.0		3.0 5.2 [6.1] 8.4
6	7.1 0.8 [9.2] 9.4		6.0 1.9 [8.2] 7.9		7.0 5.5 [6.1] 1.9

In proc 0, after applying the rotations:

```

1 [mstorti@spider example]$ mpirun -np 3 \
2 -machinefile ./machi.dat mpitypes.bin
3 After rotation step 0
4 4.8 3.2 9.5 9.9
5 7.9 8.8 8.9 7.2
6 2.7 4.2 6.1 4.1
7 7.1 0.8 6.1 9.4
8 After rotation step 1
9 4.8 3.2 9.9 9.9
10 7.9 8.8 9.8 7.2
11 2.7 4.2 5.7 4.1
12 7.1 0.8 8.2 9.4
13 After rotation step 2
14 4.8 3.2 0.2 9.9
15 7.9 8.8 1.7 7.2
16 2.7 4.2 2.0 4.1
17 7.1 0.8 9.2 9.4
18 ...

```

OPTIONAL Assignment Nbr. 5

Write a function `ord_scatter(int *sbuf, int scount, int **rbuf, int *rcount)` that, given vectors `sbuf[scount]` (`scount` it may be different in each processor), redistributes the elements in `rbuf[rcount]` so that all the elements in `sbuf[]` in rank `[rank, rank+1) * N/size` remain in processor `rank`, where `N` is the largest element of all stored in `sbuf`. Assume that the elements in `sbuf` are stored in each processor.

- Usar operaciones de *reduce* para obtener el valor de `N`
- Usar operaciones *all-to-all* para calcular los *counts* and *displs* en cada procesador.
- Usar `MPI_Alltoallv()` para redistribuir los datos.

Example: The Richardson iterative method

Example: The Richardson iterative method

Richardson's iterative method solves a system $Ax = b$ based on the following iterative scheme

$$x^{n+1} = x^n + \omega r^n \quad r^n = b - Ax$$

The sparse matrix A is stored in sparse format, i.e. we have $AIJ[nc]$, $II[nc]$, $JJ[nc]$, so that for each k , such that $0 \leq k < nc$, $II[k]$, $JJ[k]$, $AIJ[k]$ are the row and column indices, and the coefficients of A . For instance, if $A = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 3 & 0 \\ 1 & 0 & 0 \end{bmatrix}$, then the vectors are $II = [0 \ 1 \ 1 \ 2]$, $JJ = [1 \ 0 \ 1 \ 0]$, $AIJ = [1 \ 2 \ 3 \ 1]$.

The residual vector r can be easily computed with a loop over the matrix elements

```

1  for (int i=0; i<n; i++) r[i] = 0;
2  for (int k=0; k<nc; k++) {
3      int i = II[k];
4      int j = JJ[k];
5      int a = AIJ[k];
6      r[i] -= a * x[j];
7  }
8  for (int i=0; i<n; i++) r[i] += b[i];

```

The pseudocode for the Richardson method would be then

```

1 // declare matrix A, vectors x, r ...
2 // initialize x = 0.
3 int itmax=100;
4 double norm, tol=1e-3;
5 for (int iter=0; iter< itmax; iter++) {
6     // compute r ...
7     for (int i=0; i<n; i++) x[i] += omega * r[i];
8 }
    
```

We propose the following parallel implementation using MPI

- The master node reads **A** and **B** from a file. We use the following provided routine

```

1 void read_matrix(char *file,
2                 int ncmx, int *I, int *J, double * AIJ, int &nc,
3                 int nmx, double *B, int &n) {
    
```

- The master does a broadcast of the matrix and rhs vector to the nodes.
- A range of rows is assigned to each processor.

```

1 int nlocal = n/size;
2 if (myrank< n % size) nlocal++;
    
```

- Each processor has the matrix and vector and only computes that part of the residual (those rows **i** that are stored in the corresponding range).
- Once every processor has computed his part, they exchange their contributions using **MPI_Allgatherv**.

PETSc

PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is a series of libraries and data structures for the numerical solution of system of equations coming from the discretization of PDE's in HPC computers. PETSc was developed at ANL (Argonne National Laboratory, IL) by a team composed of Satish Balay, William Gropp, and others.

<http://www.mcs.anl.gov/petsc/>

PETSc (cont.)

- PETSc uses the concept of **Object Oriented Programming** (OOP) to ease the development of large scientific programs.
- The library is written in C and can be called from C/C++ and Fortran. The routines are grouped according to the abstract object types that are involved (e.g. vectors, matrices, solvers...), in a similar way to classes in C++.

PETSc Objects

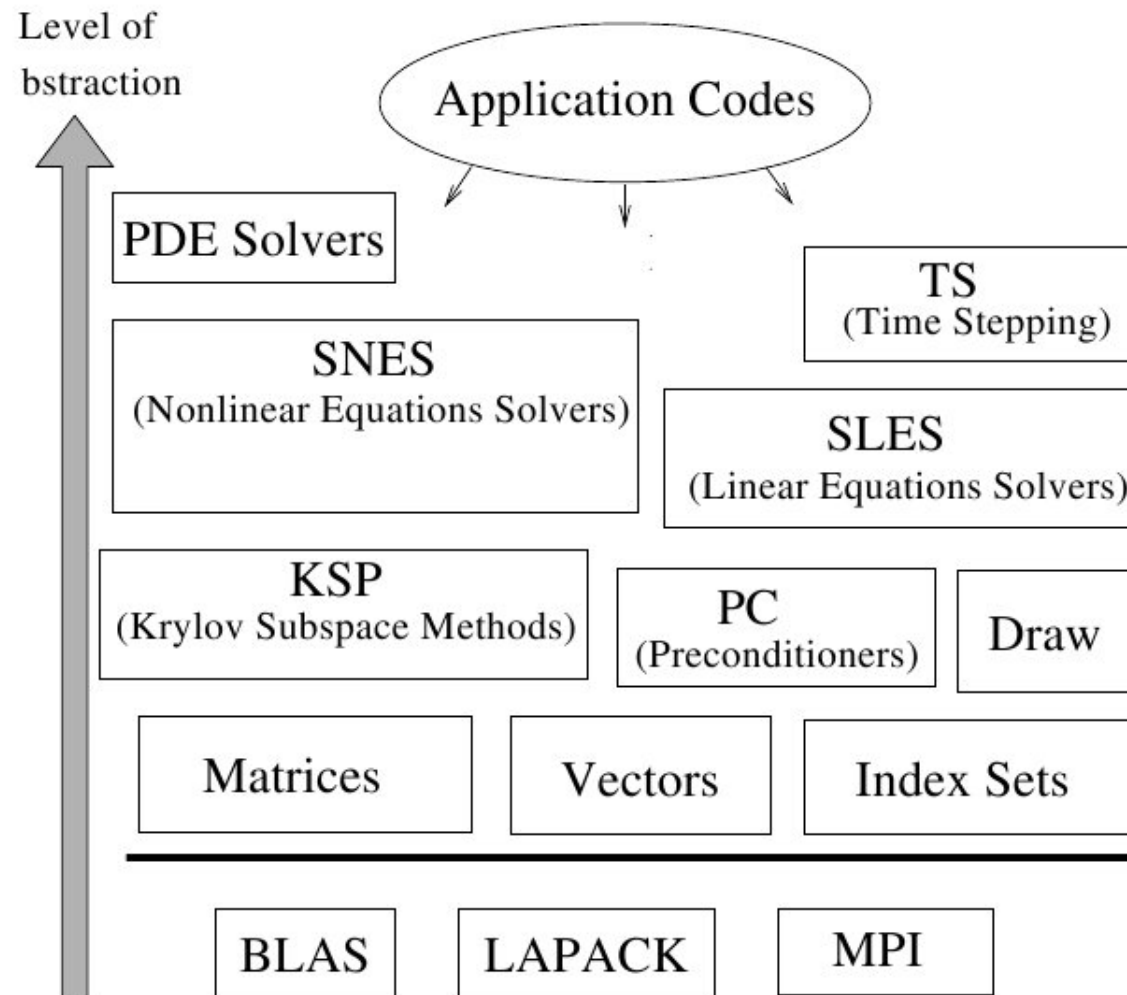
PETSc objects

- Index sets, permutations, renumbering.
- vectors
- matrices (specially sparse)
- distributed arrays (structured meshes)
- preconditioners (including multigrid, direct solvers, Schwartz decomposition).
- non-linear solvers (Newton-Raphson ...)
- non-linear temporal integration

PETSc objects (cont.)

- Each one of this types have an abstract interface and one or more implementations of this interface.
- This allows to easily test for different implementations of algorithms in the different phases of resolution of PDE's, for instance the different Krylov solvers, different preconditioners...
- This promotes reusability and flexibility of the code.

Structure of the PETSc library



Using PETSc

Using PETSc

Define the following environment variables

- ***PETSC_DIR*** = points to the root directory of the PETSc installation. (e.g. *PETSC_DIR=/home/mstorti/PETSC/petsc-2.1.3*)
- ***PETSC_ARCH*** architecture and compilation options (e.g. *PETSC_ARCH=linux*). The *<PETSC_DIR>/bin/petscarch* utility allows to determine the system architecture in execution time (e.g. *PETSC_ARCH= '\$PETSC_DIR/bin/petscarch '*)

Using PETSc (cont.)

- PETSc uses MPI for the message passing so that all programs compiled with PETSc must be run with *mpirun*.
PETSc usa MPI para el paso de mensajes de manera que los programas compilados con PETSc deben ser ejecutados con *mpirun* e.g.

```
1 [mstorti@spider example]$ mpirun <mpi_options> \  
2     <petsc_program_name> <petsc_options> \  
3     <user_prog_options>  
4 [mstorti@spider example]$
```

Por ejemplo

```
1 [mstorti@spider example]$ mpirun -np 8 \  
2     -machinefile machi.dat my_fem \  
3     -log_summary -N 100 -M 20  
4 [mstorti@spider example]$
```

Writing programs that use PETSc

```
1 // C/C++
2 PetscInitialize(int *argc, char ***argv,
3               char *file, char *help);

1 C FORTRAN
2      call PetscInitialize (character file,
3                          integer ierr) !Fortran
```

- Calls *MPI_Init* (if it was not called yet). If user needs to call *MPI_Init* *before PetscInitialize* then it must be called *before*, i.e. *MPI_Init* *then PetscInitialize*.
- Defines communicators *PETSC_COMM_WORLD=MPI_COMM_WORLD* and *PETSC_COMM_SELF* (for a one-processor only run).

Writing programs that use PETSc (cont.)

```
1 // C/C++
2 PetscFinalize();

1 C Fortran
2      call PetscFinalize(ierr)
```

Calls *MPI_Finalize* (if MPI was initialized by PETSc).

Simple example. The Laplace 1D eq.

Solve $Ax = b$, where

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & & \\ -1 & 2 & -1 & 0 & \dots & \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & & & \ddots & & \\ \dots & 0 & -1 & 2 & -1 & 0 \\ & \dots & 0 & -1 & 2 & -1 \\ & & \dots & 0 & -1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \quad x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}$$

Simple example. The Laplace 1D eq. (cont.)

Declare PETSc variables (vectors and matrices, initially *invalid pointers*).

```
1  Vec      x, b, u;      /* approx solution, RHS, exact solution */
2  Mat      A;            /* linear system matrix */
```

Create the objects

```
1  ierr = VecCreate(PETSC_COMM_WORLD, &x); CHKERRQ(ierr);
2  ierr = MatCreate(PETSC_COMM_WORLD, PETSC_DECIDE,
3                  PETSC_DECIDE, n, n, &A);
```

In general

```
1  PetscType object;
2  ierr = PetscTypeCreate(PETSC_COMM_WORLD, . . .options. . ., &object);
3  // use 'object' . . .
4  ierr = PetscTypeDestroy(x); CHKERRQ(ierr);
```

PetscType may be *Vec*, *Mat*, *PC*, *KSP*

Simple example. The Laplace 1D eq. (cont.)

Duplicate (clone) objects

```
1 Vec x,b,u;  
2 ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);  
3 ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);  
4 ierr = VecSetFromOptions(x);CHKERRQ(ierr);  
5  
6 ierr = VecDuplicate(x,&b);CHKERRQ(ierr);  
7 ierr = VecDuplicate(x,&u);CHKERRQ(ierr);
```

Simple example. The Laplace 1D eq. (cont.)

Set values on vectors and matrices

```
1 Vec b; VecCreate( . . . , b );
2 Mat A; MatCreate( . . . , A );
3
4 // Equivalent to Matlab: b(row) = val; A(row,col) = val;
5 VecSetValue(b, row, val, INSERT_VALUES);
6 MatSetValue(A, row, col, val, INSERT_VALUES);
7
8 // Equivalent to Matlab: b(row) = b(row) + val;
9 //                      A(row,col) = A(row,col) + val;
10 VecSetValue(b, row, val, ADD_VALUES);
11 MatSetValue(A, row, col, val, ADD_VALUES);
```

Also may assemble matrices *by blocks*

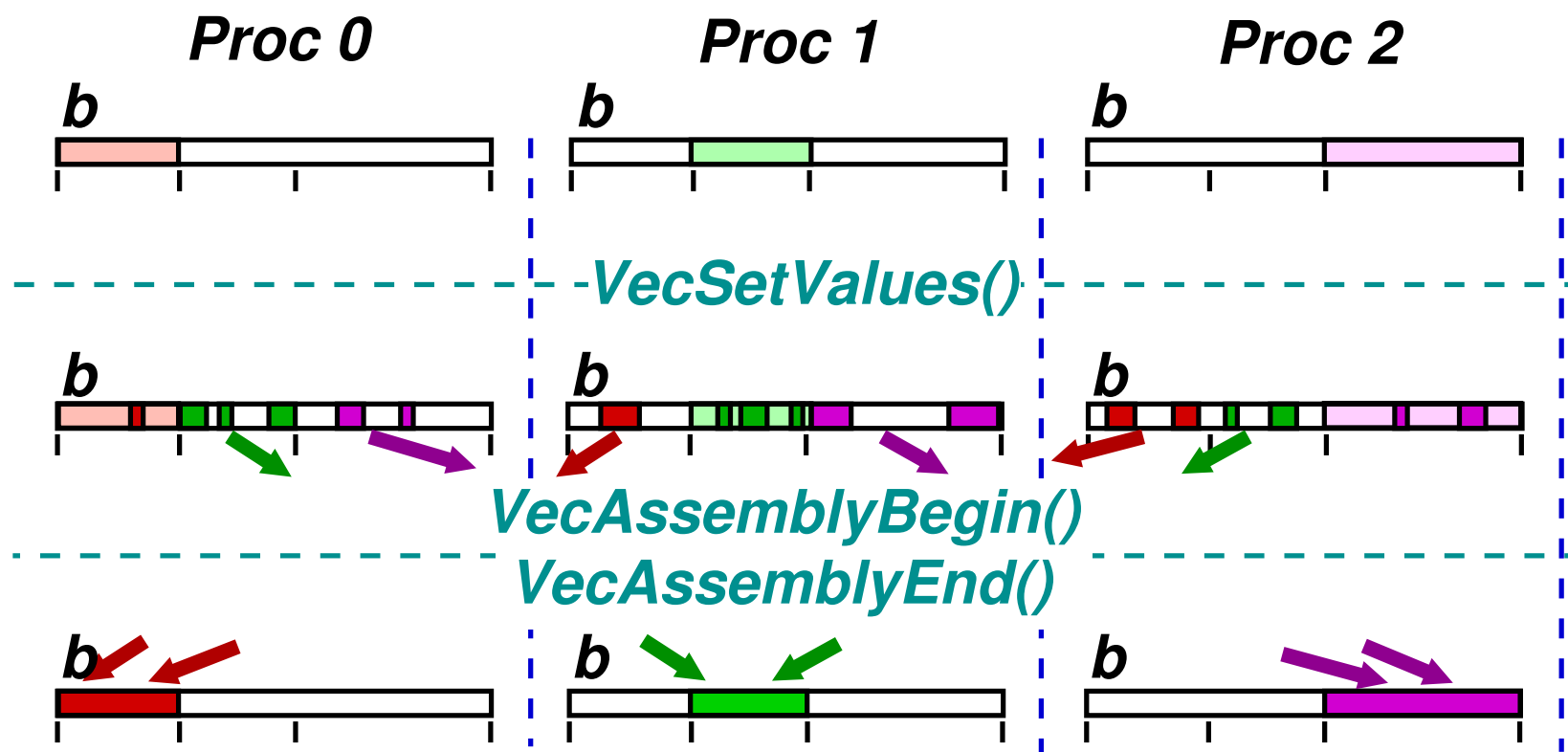
```
1 int nrow, *rows, ncols, *cols;
2 double *vals;
3 // Equivalent to Matlab: A(rows,cols) = vals;
4 MatSetValues(A, nrows, rows, ncols, cols, vals, INSERT_VALUES);
```


Simple example. The Laplace 1D eq. (cont.)

After doing *...SetValues(...)* we must call *...Assembly(...)*

```
1 Mat A; MatCreate(...,A);
2 MatSetValues(A,nrows,rows,ncols,cols,vals,INSERT_VALUES);
3
4 // Starts communication
5 MatAssemblyBegin(A,...);
6 // Can't use 'A' yet
7 // (Can overlap comp/comm.) Do computations .....
8 // Ends communication
9 MatAssemblyEnd(A,...);
10 // Can use 'A' now
```

Simple example. The Laplace 1D eq. (cont.)



Simple example. The Laplace 1D eq. (cont.)

```
1 // Assemble matrix. All processors set all values.
2 value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
3 for (i=1; i<n-1; i++) {
4     col[0] = i-1; col[1] = i; col[2] = i+1;
5     ierr = MatSetValues(A, 1, &i, 3, col, value, INSERT_VALUES);
6     CHKERRQ(ierr);
7 }
8 i = n - 1; col[0] = n - 2; col[1] = n - 1;
9 ierr = MatSetValues(A, 1, &i, 2, col, value,
10                     INSERT_VALUES); CHKERRQ(ierr);
11 i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
12 ierr = MatSetValues(A, 1, &i, 2, col, value,
13                     INSERT_VALUES); CHKERRQ(ierr);
14 ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
15 ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
```

Simple example. The Laplace 1D eq. (cont.)

Use of *KSP* (*Krylov Space Scalable Linear Equations Solvers*)

```
1 KSP ksp;  
2 KSPCreate (PETSC_COMM_WORLD, &ksp) ;  
3 ierr = KSPSetOperators (KSP, A, A,  
4                         DIFFERENT_NONZERO_PATTERN) ;  
5  
6 PC pc;  
7 KSPGetPC (KSP, &pc) ;  
8 PCSetType (pc, PCJACOBI) ;  
9 KSPSetTolerances (ksp, 1.e-7, PETSC_DEFAULT,  
10                  PETSC_DEFAULT, PETSC_DEFAULT) ;  
11  
12 KSPSolve (ksp, b, x, &its) ;  
13  
14 ierr = KSPDestroy (ksp) ; CHKERRQ (ierr) ;
```

Simple example. The Laplace 1D eq. (cont.)

```

1  /* Program usage: mpiexec ex1 [-help] [all PETSc options] */
2
3  static char help[] =
4      "Solves a tridiagonal linear system with KSP.\n\n";
5
6  #include "petscksp.h"
7
8  #undef __FUNCT__
9  #define __FUNCT__ "main"
10 int main(int argc, char **args)
11 {
12     Vec          x, b, u;      /* approx solution, RHS,
13                               exact solution */
14     Mat          A;           /* linear system matrix */
15     KSP          ksp;         /* linear solver context */
16     PC           pc;          /* preconditioner context */
17     PetscReal    norm;        /* norm of solution error */
18     PetscErrorCode ierr;
19     PetscInt     i, n = 10, col[3], its;
20     PetscMPIInt  size;
21     PetscScalar  neg_one = -1.0, one = 1.0, value[3];
22
23     PetscInitialize(&argc, &args, (char *) 0, help);
24     ierr = MPI_Comm_size(PETSC_COMM_WORLD, &size); CHKERRQ(ierr);
25     if (size != 1) SETERRQ(1, "This is a uniprocessor example only!");
26     ierr = PetscOptionsGetInt(PETSC_NULL, "-n",
27                               &n, PETSC_NULL); CHKERRQ(ierr);

```

```

28
29  /* - - - - -
30      Compute the matrix and right-hand-side vector that define
31      the linear system,  $Ax = b$ .
32  - - - - - */
33
34  /*
35      Create vectors. Note that we form 1 vector from scratch and
36      then duplicate as needed.
37  */
38  ierr = VecCreate(PETSC_COMM_WORLD, &x); CHKERRQ(ierr);
39  ierr = PetscObjectSetName((PetscObject) x,
40                          "Solution"); CHKERRQ(ierr);
41  ierr = VecSetSizes(x, PETSC_DECIDE, n); CHKERRQ(ierr);
42  ierr = VecSetFromOptions(x); CHKERRQ(ierr);
43  ierr = VecDuplicate(x, &b); CHKERRQ(ierr);
44  ierr = VecDuplicate(x, &u); CHKERRQ(ierr);
45
46  /*
47      Create matrix. When using MatCreate(), the matrix
48      format can be specified at runtime.
49
50      Performance tuning note: For problems of substantial
51      size, preallocation of matrix memory is crucial for
52      attaining good performance. See the matrix chapter of
53      the users manual for details.
54  */
55  ierr = MatCreate(PETSC_COMM_WORLD, &A); CHKERRQ(ierr);
56  ierr = MatSetSizes(A, PETSC_DECIDE,
57                    PETSC_DECIDE, n, n); CHKERRQ(ierr);
58  ierr = MatSetFromOptions(A); CHKERRQ(ierr);
59
60  /*

```

```

61  Assemble matrix
62  */
63  value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
64  for (i=1; i<n-1; i++) {
65      col[0] = i-1; col[1] = i; col[2] = i+1;
66      ierr = MatSetValues(A, 1, &i, 3, col,
67                          value, INSERT_VALUES); CHKERRQ(ierr);
68  }
69  i = n - 1; col[0] = n - 2; col[1] = n - 1;
70  ierr = MatSetValues(A, 1, &i, 2, col,
71                      value, INSERT_VALUES); CHKERRQ(ierr);
72  i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
73  ierr = MatSetValues(A, 1, &i, 2, col,
74                      value, INSERT_VALUES); CHKERRQ(ierr);
75  ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
76  ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
77
78  /*
79  Set exact solution; then compute right-hand-side vector.
80  */
81  ierr = VecSet(u, one); CHKERRQ(ierr);
82  ierr = MatMult(A, u, b); CHKERRQ(ierr);
83
84  /* - - - - -
85  Create the linear solver and set various options
86  - - - - -*/
87  /*
88  Create linear solver context
89  */
90  ierr = KSPCreate(PETSC_COMM_WORLD, &ksp); CHKERRQ(ierr);
91
92  /*

```

```

93      Set operators. Here the matrix that defines the linear
94      system also serves as the preconditioning matrix.
95      */
96      ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);
97      CHKERRQ(ierr);
98
99      /*
100     Set linear solver defaults for this problem (optional).
101     - By extracting the KSP and PC contexts from the KSP
102     context, we can then directly call any KSP and PC
103     routines to set various options.
104     - The following four statements are optional; all of
105     these parameters could alternatively be specified at
106     runtime via KSPSetFromOptions();
107     */
108     ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
109     ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
110     ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,
111                             PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);
112
113     /*
114     Set runtime options, e.g.,
115     -ksp_type <type> -pc_type <type>
116     -ksp_monitor -ksp_rtol <rtol>
117     These options will override those specified above as
118     long as KSPSetFromOptions() is called _after_ any other
119     customization routines.
120     */
121     ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);
122
123     /* - - - - -
124     Solve the linear system

```



```

125      - - - - - */
126      ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
127
128      /*
129      View solver info; we could instead use the option
130      -ksp_view to print this info to the screen at the
131      conclusion of KSPSolve().
132      */
133      ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
134
135      /* - - - - -
136      Check solution and clean up
137      - - - - - */
138      /*
139      Check the error
140      */
141      ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
142      ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
143      ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
144      ierr = PetscPrintf(PETSC_COMM_WORLD,
145                          "Norm of error %A, Iterations %D\n",
146                          norm,its);CHKERRQ(ierr);
147      /*
148      Free work space. All PETSc objects should be destroyed
149      when they are no longer needed.
150      */
151      ierr = VecDestroy(x);CHKERRQ(ierr);
152      ierr = VecDestroy(u);CHKERRQ(ierr);
153      ierr = VecDestroy(b);CHKERRQ(ierr);
154      ierr = MatDestroy(A);CHKERRQ(ierr);
155      ierr = KSPDestroy(ksp);CHKERRQ(ierr);
156

```

```
157  /*
158
159     Always call PetscFinalize() before exiting a program.
160     This routine
161     - finalizes the PETSc libraries as well as MPI
162     - provides summary and diagnostic information if
163       certain runtime options are chosen (e.g.,
164       -log_summary) .
165  */
166  ierr = PetscFinalize();CHKERRQ(ierr);
167  return 0;
168 }
```

PETSc elements

Headers

```
1 #include "petscksp.h"
```

Located at `<PETSC_DIR>/include`

Headers for the high level libraries include headers for the low level libraries, e.g. `petscksp.h` (Krylov space linear solvers) include already

- `petscmat.h` (matrices),
- `petscvec.h` (vectors), and
- `petsc.h` (base PETSc file).

Options data base

You can pass user options via

- configuration file `~/.petsrc`
- environment variables `PETSC_OPTIONS`
- command line, e.g.: `mpirun -np 1 ex1 -n 100`

Options can be obtained in execution time with

```
1 OptionsGetInt (PETSC_NULL, "-n", &n, &flag) ;
```

Vectors

```
1 VecCreate (MPI_Comm comm , Vec *x);
2 VecSetSizes (Vec x, int m, int M );
3 VecDuplicate (Vec old, Vec *new);
4 VecSet (PetscScalar *value, Vec x);
5 VecSetValues (Vec x, int n, int *indices,
6             PetscScalar *values, INSERT_VALUES);
```

- *m* = (optional) local (may be *PETSC_DECIDE*)
- *M* global size
- *VecSetType()*, *VecSetFromOptions()* allow the definition of the storage type.

Matrices

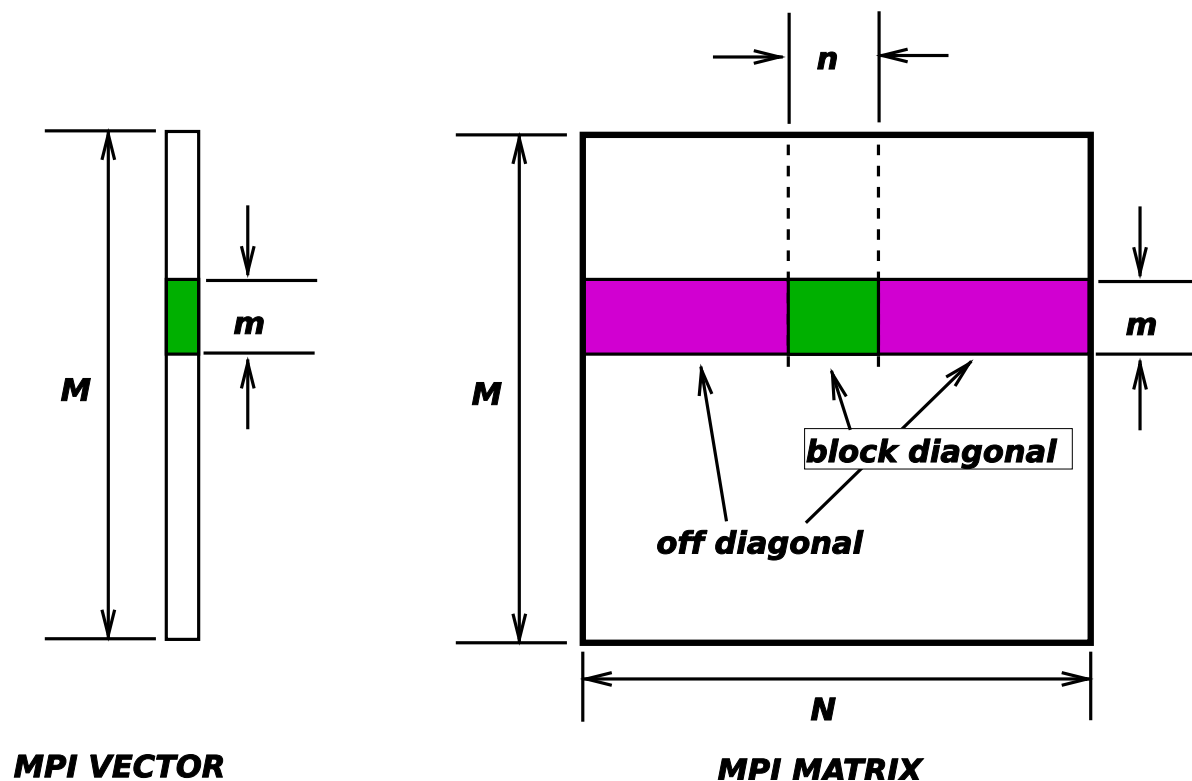
```
1 // Create a matrix
2 MatCreate(MPI_Comm comm, int m,
3           int n, int M, int N, Mat *A);
4
5 // Set values in rows 'im' and columns 'in'
6 MatSetValues(Mat A, int m, int *im, int n, int *in,
7             PetscScalar *values, INSERT_VALUES);
8
9 // Do assembly BEFORE using the matrices
10 // (values may be stored in temporary buffers yet)
11 MatAssemblyBegin(Mat A, MAT_FINAL_ASSEMBLY);
12 MatAssemblyEnd(Mat A, MAT_FINAL_ASSEMBLY);
```

Linear solvers

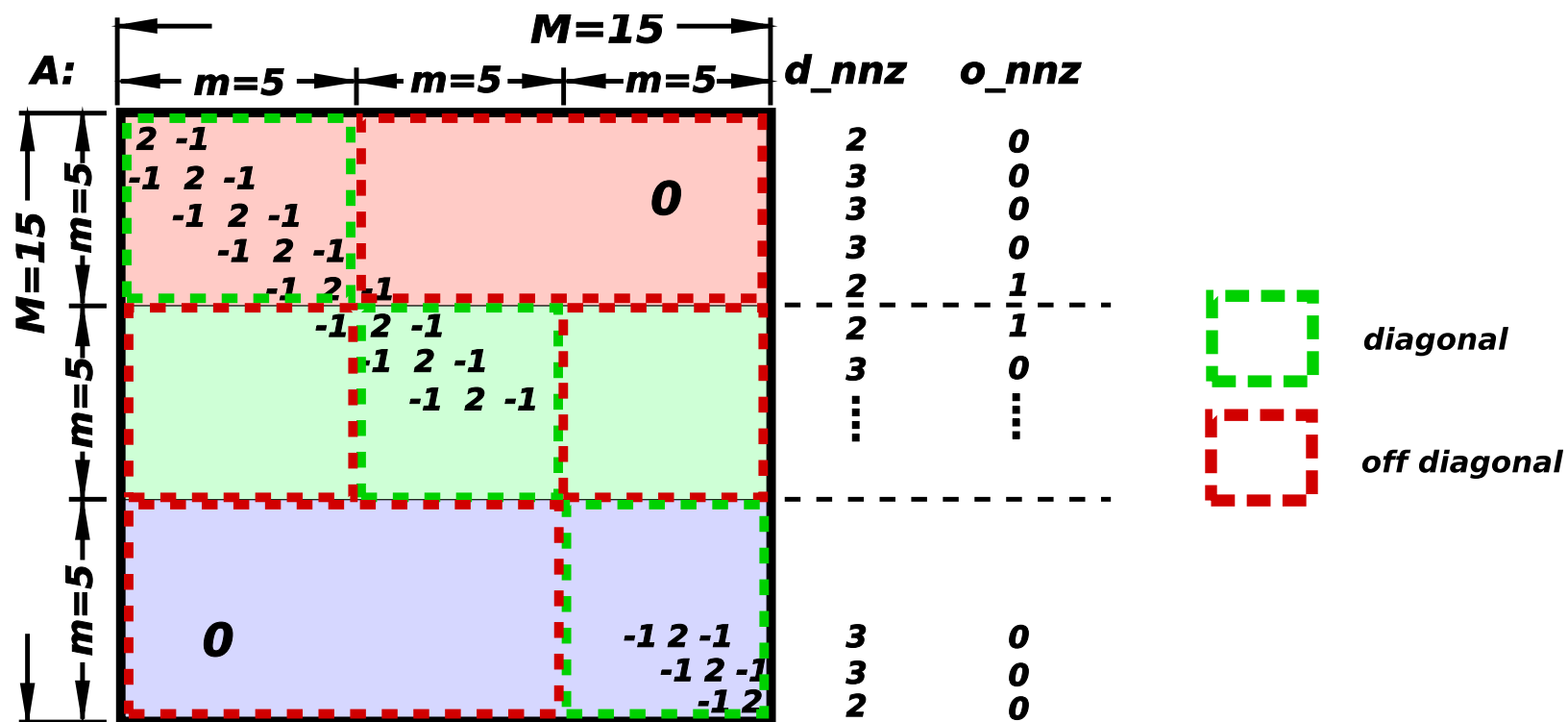
```
1 // Create the KSP
2 KSPCreate(MPI_Comm comm ,KSP *ksp);
3
4 // Set matrix and preconditioning
5 KSPSetOperators (KSP ksp,Mat A,
6     Mat PrecA,MatStructure flag);
7
8 // Use options from data base (solver type?)
9 //     (file/env/command)
10 KSPSetFromOptions (KSP ksp);
11
12 // Solve linear system
13 KSPSolve (KSP ksp,Vec b,Vec x,int *its);
14
15 // Destroy KSP (free mem. of fact. matrix)
16 KSPDestroy (KSP ksp);
```


Parallel programming

```
1 int VecCreateMPI(MPI_Comm comm, int m, int M, Vec *v);  
2 int MatCreateMPIAIJ(MPI_Comm comm, int m, int n, int M, int N,  
3   int d_nz, int *d_nnz, int o_nz, int *o_nnz, Mat *A);
```



Parallel programming (cont.)



Parallel programming (cont.)

```

1 static char help[] = "Solves a tridiagonal linear system.\n\n";
2 #include "petscksp.h"
3
4 #undef __FUNCT__
5 #define __FUNCT__ "main"
6 int main(int argc, char **args)
7 {
8     Vec          x, b, u;      /* approx solution, RHS,
9                                exact solution */
10    Mat           A;           /* linear system matrix */
11    KSP           ksp;         /* linear solver context */
12    PC            pc;          /* preconditioner context */
13    PetscReal     norm;        /* norm of solution error */
14    PetscErrorCode ierr;
15    PetscInt      i, n = 10, col[3], its, rstart, rend, nlocal;
16    PetscScalar   neg_one = -1.0, one = 1.0, value[3];
17
18    PetscInitialize(&argc, &args, (char *) 0, help);
19    ierr = PetscOptionsGetInt(PETSC_NULL, "-n", &n, PETSC_NULL);
20    CHKERRQ(ierr);
21
22    /* -----
23       Compute the matrix and right-hand-side vector that
24       define the linear system, Ax = b.
25       ----- */
26
27    /* Create vectors. Note that we form 1 vector from
28       scratch and then duplicate as needed. For this simple

```

```

29      case let PETSc decide how many elements of the vector
30      are stored on each processor. The second argument to
31      VecSetSizes() below causes PETSc to decide. */
32      ierr = VecCreate(PETSC_COMM_WORLD, &x); CHKERRQ(ierr);
33      ierr = VecSetSizes(x, PETSC_DECIDE, n); CHKERRQ(ierr);
34      ierr = VecSetFromOptions(x); CHKERRQ(ierr);
35      ierr = VecDuplicate(x, &b); CHKERRQ(ierr);
36      ierr = VecDuplicate(x, &u); CHKERRQ(ierr);
37
38      /* Identify the starting and ending mesh points on each
39      processor for the interior part of the mesh. We let
40      PETSc decide above. */
41
42      ierr = VecGetOwnershipRange(x, &rstart, &rend); CHKERRQ(ierr);
43      ierr = VecGetLocalSize(x, &nlocal); CHKERRQ(ierr);
44
45      /* Create matrix. When using MatCreate(), the matrix
46      format can be specified at runtime.
47
48      Performance tuning note: For problems of substantial
49      size, preallocation of matrix memory is crucial for
50      attaining good performance. See the matrix chapter of
51      the users manual for details.
52
53      We pass in nlocal as the ``local'' size of the matrix to
54      force it to have the same parallel layout as the vector
55      created above. */
56      ierr = MatCreate(PETSC_COMM_WORLD, &A); CHKERRQ(ierr);
57      ierr = MatSetSizes(A, nlocal, nlocal, n, n); CHKERRQ(ierr);
58      ierr = MatSetFromOptions(A); CHKERRQ(ierr);
59
60      /* Assemble matrix.
61      The linear system is distributed across the processors

```

```

62      by chunks of contiguous rows, which correspond to
63      contiguous sections of the mesh on which the problem is
64      discretized. For matrix assembly, each processor
65      contributes entries for the part that it owns locally. */
66      if (!rstart) {
67          rstart = 1;
68          i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
69          ierr = MatSetValues(A, 1, &i, 2, col, value, INSERT_VALUES);
70          CHKERRQ(ierr);
71      }
72      if (rend == n) {
73          rend = n-1;
74          i = n-1; col[0] = n-2; col[1] = n-1;
75          value[0] = -1.0; value[1] = 2.0;
76          ierr = MatSetValues(A, 1, &i, 2, col, value, INSERT_VALUES);
77          CHKERRQ(ierr);
78      }
79
80      /* Set entries corresponding to the mesh interior */
81      value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
82      for (i=rstart; i<rend; i++) {
83          col[0] = i-1; col[1] = i; col[2] = i+1;
84          ierr = MatSetValues(A, 1, &i, 3, col, value, INSERT_VALUES);
85          CHKERRQ(ierr);
86      }
87
88      /* Assemble the matrix */
89      ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
90      ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
91
92      /* Set exact solution; then compute right-hand-side vector. */
93      ierr = VecSet(u, one); CHKERRQ(ierr);

```

```

94  ierr = MatMult (A, u, b) ; CHKERRQ (ierr) ;
95
96  /* - - - - -
97      Create the linear solver and set various options
98      - - - - - */
99  ierr = KSPCreate (PETSC_COMM_WORLD, &ksp) ; CHKERRQ (ierr) ;
100
101  /* Set operators. Here the matrix that defines the linear system
102      also serves as the preconditioning matrix. */
103  ierr = KSPSetOperators (ksp, A, A, DIFFERENT_NONZERO_PATTERN) ;
104  CHKERRQ (ierr) ;
105
106  /*
107      Set linear solver defaults for this problem (optional).
108      - By extracting the KSP and PC contexts from the KSP context,
109      we can then directly call any KSP and PC routines to set
110      various options.
111      - The following four statements are optional; all of these
112      parameters could alternatively be specified at runtime via
113      KSPSetFromOptions () ;
114  */
115  ierr = KSPGetPC (ksp, &pc) ; CHKERRQ (ierr) ;
116  ierr = PCSetType (pc, PCJACOBI) ; CHKERRQ (ierr) ;
117  ierr = KSPSetTolerances (ksp, 1.e-7, PETSC_DEFAULT, PETSC_DEFAULT,
118                          PETSC_DEFAULT) ; CHKERRQ (ierr) ;
119
120  /*
121      Set runtime options, e.g.,
122      -ksp_type <type> -pc_type <type>
123      -ksp_monitor -ksp_rtol <rtol>
124
125      These options will override those specified above as
126      long as KSPSetFromOptions() is called _after_ any other

```

```

126      customization routines.
127  */
128  ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
129
130  /* -----
131      Solve the linear system
132  ----- */
133  /*
134      Solve linear system
135  */
136  ierr = KSPSolve(ksp,b,x); CHKERRQ(ierr);
137
138  /*
139      View solver info; we could instead use the option
140      -ksp_view to print this info to the screen at the
141      conclusion of KSPSolve().
142  */
143  ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
144
145  /* -----
146      Check solution and clean up
147  ----- */
148  /*
149      Check the error
150  */
151  ierr = VecAXPY(x,neg_one,u); CHKERRQ(ierr);
152  ierr = VecNorm(x,NORM_2,&norm); CHKERRQ(ierr);
153  ierr = KSPGetIterationNumber(ksp,&its); CHKERRQ(ierr);
154  ierr = PetscPrintf(PETSC_COMM_WORLD,
155                    "Norm of error %A, Iterations %D\n",
156                    norm,its); CHKERRQ(ierr);
157  /*
158      Free work space. All PETSc objects should be destroyed

```

```

159      when they are no longer needed.
160      */
161      ierr = VecDestroy(x);CHKERRQ(ierr);
162      ierr = VecDestroy(u);CHKERRQ(ierr);
163      ierr = VecDestroy(b);CHKERRQ(ierr);
164      ierr = MatDestroy(A);CHKERRQ(ierr);
165      ierr = KSPDestroy(ksp);CHKERRQ(ierr);
166
167      /*
168      Always call PetscFinalize() before exiting a program.
169      This routine
170      - finalizes the PETSc libraries as well as MPI
171      - provides summary and diagnostic information if
172        certain runtime options are chosen (e.g.,
173          -log_summary).
174      */
175      ierr = PetscFinalize();CHKERRQ(ierr);
176      return 0;
177  }

```


Programa de FEM basado en PETSc

FEM program based on PETSc

```
1 double xnod(nnod, ndim=2); // Read coords ...
2 int icone(nelem, nel=3); // Read connectivities...
3 // Read fixations (map) fixa: nodo -> valor ...
4 // Build (map) n2e:
5 //      nodo -> list of connected elems ...
6 // Partition elements (dual graph) by Cuthill-Mc.Kee ...
7 // Partition nodes (induced by element partitio) ...
8 // Number equations ...
9 // Create MPI-PETSc matrix and vectors ...
10 for (e=1; e<=nelem; e++) {
11     // compute be, ke = rhs and matrix for element e in 'A' and 'b;'
12     // assemble be and ke;
13 }
14 MatAssembly(A, ... )
15 MatAssembly(b, ... );
16 // Build KSP...
17 // Solve Ax = b;
```

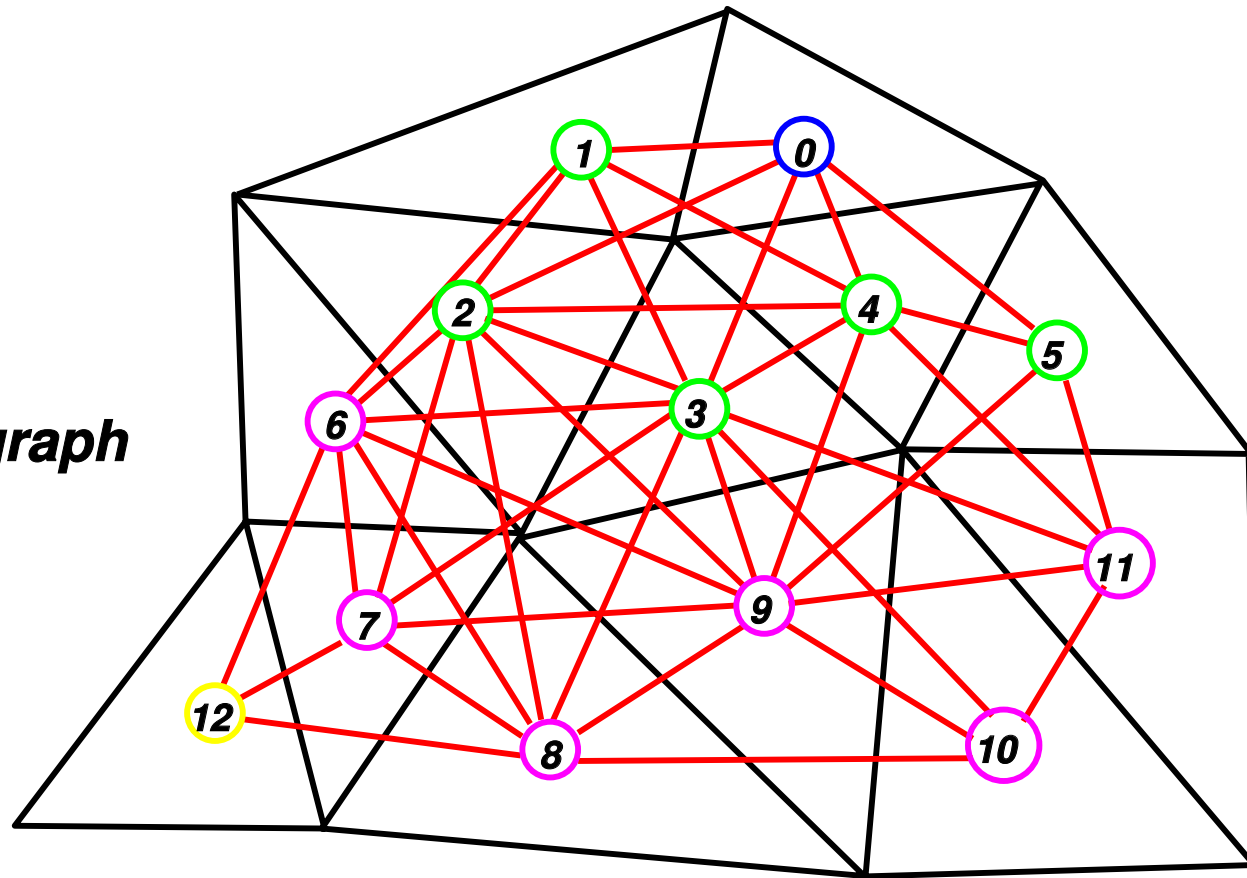
Basic concepts on partitioning

Graph numbering by Cuthill-Mc.Kee

```
1 // Number vertices in a graph
2 Graph G, Queue C;
3 Vertex n = any vertex in G;
4 last = 0;
5 Indx[n] = last++;
6 Put n in C;
7 Vertex c,m;
8 while (! C.empty()) {
9     n = C.front();
10    C.pop();
11    for (m in neighbors(c,G)) {
12        if (m not already indexed) {
13            C.put(m);
14            Indx[m] = last++;
15        }
16    }
17 }
```

Graph numbering by Cuthill-Mc.Kee (cont.)

dual graph



Element partitioning

```
1 // nelem[proc] is the number of elements to put
2 // on processor 'proc'
3 int nproc;
4 int nelem[nproc];
5 // Number elements by Cuthill-McKee Indx[e] ...
6 // Put first nelem[0] elements in proc. 0 ...
7 // Put following nelem[1] elements in proc. 1 ...
8 // ...
9 // Put following nelem[nproc-1] elements in proc. nproc-1
```

Node partitioning

Given a node *n*, assign it to any processor to which a connected element belongs.

FEM code

FEM program based on PETSc

```

1  #include <cstdio>
2
3  #include <iostream>
4  #include <set>
5  #include <vector>
6  #include <deque>
7  #include <map>
8
9  #include <petscsles.h>
10
11 #include <newmatio.h>
12
13 static char help[] =
14 "Basic test for self scheduling algorithm FEM with PETSc";
15
16 //-----<*>-----<*>-----<*>-----<*>-----<*>-----
17 #undef __FUNC__
18 #define __FUNC__ "main"
19 int main(int argc, char **args) {
20     Vec b, u;
21     Mat A;
22     SLES sles;
23     PC    pc;          /* preconditioner context */
24     KSP    ksp;        /* Krylov subspace method context */
25
26     // Number of nodes per element
27     const int nel=3;
28     // Space dimension
29     const int ndim=2;

```



```
30
31 PetscInitialize (&argc, &args, (char *) 0, help);
32
33 // Get MPI info
34 int size, rank;
35 MPI_Comm_size (PETSC_COMM_WORLD, &size);
36 MPI_Comm_rank (PETSC_COMM_WORLD, &rank);
37
38 // read nodes
39 double x[ndim];
40 int read;
41 vector<double> xnod;
42 vector<int> iconc;
43 FILE *fid = fopen("node.dat", "r");
44 assert(fid);
45 while (true) {
46     for (int k=0; k<ndim; k++) {
47         read = fscanf(fid, "%lf", &x[k]);
48         if (read == EOF) break;
49     }
50     if (read == EOF) break;
51     xnod.push_back(x[0]);
52     xnod.push_back(x[1]);
53 }
54 fclose(fid);
55 int nnod=xnod.size()/ndim;
56 PetscPrintf(PETSC_COMM_WORLD, "Read %d nodes.\n", nnod);
57
58 // read elements
59 int ix[nel];
60 fid = fopen("icone.dat", "r");
61 assert(fid);
62 while (true) {
```

```

63     for (int k=0; k<nel; k++) {
64         read = fscanf(fid, "%d", &ix[k]);
65         if (read == EOF) break;
66     }
67     if (read == EOF) break;
68     icone.push_back(ix[0]);
69     icone.push_back(ix[1]);
70     icone.push_back(ix[2]);
71 }
72 fclose(fid);
73
74 int nelem=icone.size()/nel;
75 PetscPrintf(PETSC_COMM_WORLD, "Read %d elements.\n", nelem);
76
77 // read fixations stored as a map node -> value
78 map<int, double> fixa;
79 fid = fopen("fixa.dat", "r");
80 assert(fid);
81 while (true) {
82     int nod;
83     double val;
84     read = fscanf(fid, "%d %lf", &nod, &val);
85     if (read == EOF) break;
86     fixa[nod]=val;
87 }
88 fclose(fid);
89 PetscPrintf(PETSC_COMM_WORLD, "Read %d fixations.\n", fixa.size());
90
91 // Construct node to element pointer
92 // n2e[j-1] is the set of elements connected to node 'j'
93 vector< set<int> > n2e;
94 n2e.resize(nnod);

```

```

95   for (int j=0; j<nelem; j++) {
96       for (int k=0; k<nel; k++) {
97           int nodo = icone[j*nel+k];
98           n2e[nodo-1].insert(j+1);
99       }
100  }
101
102  #if 0    // Output 'n2e' array if needed
103      for (int j=0; j<nnod; j++) {
104          set<int>::iterator k;
105          PetscPrintf(PETSC_COMM_WORLD, "node %d, elements: ", j+1);
106          for (k=n2e[j].begin(); k!=n2e[j].end(); k++) {
107              PetscPrintf(PETSC_COMM_WORLD, "%d ", *k);
108          }
109          PetscPrintf(PETSC_COMM_WORLD, "\n");
110      }
111  #endif
112
113  //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
114  // Simple partitioning algorithm
115  deque<int> q;
116  vector<int> eord, id;
117  eord.resize(nelem, 0);
118  id.resize(nnod, 0);
119
120  // Mark element 0 as belonging to processor 0
121  q.push_back(1);
122  eord[0]=-1;
123  int order=0;
124
125
126  while (q.size()>0) {

```

```

127      // Pop an element from the queue
128      int elem=q.front();
129      q.pop_front();
130      eord[elem-1]=++order;
131      // Push all elements neighbor to this one in the queue
132      for (int nod=0; nod<nel; nod++) {
133          int node = icone[(elem-1)*nel+nod];
134          set<int> &e = n2e[node-1];
135
136          for (set<int>::iterator k=e.begin(); k!=e.end(); k++) {
137              if (eord[*k-1]==0) {
138                  q.push_back(*k);
139                  eord[*k-1]=-1;
140              }
141          }
142      }
143  }
144  q.clear();
145
146      // Element partition. Put (approximately)
147      // nelem/size in each processor
148      int *e_indx = new int[size+1];
149      e_indx[0]=1;
150      for (int p=0; p<size; p++)
151          e_indx[p+1] = e_indx[p]+(nelem/size) + (p<(nelem % size) ? 1 : 0);
152
153      // Node partitioning. If a node is connected to an element 'j' then
154      // put it in the processor where element 'j' belongs.
155      // In case the elements connected to the node belong to
156      // different processors take any one of them.
157      int *id_indx = new int[size+2];
158      for (int j=0; j<nnod; j++) {

```

```

159     if (fixa.find(j+1) != fixa.end()) {
160         id[j]=-(size+1);
161     } else {
162         set<int> &e = n2e[j];
163         assert(e.size()>0);
164         int order = eord[*e.begin()-1];
165         for (int p=0; p<size; p++) {
166             if (order >= e_indx[p] && order < e_indx[p+1]) {
167                 id[j] = -(p+1);
168                 break;
169             }
170         }
171     }
172 }
173
174 // 'id_indx[j-1]' is the dof associated to node 'j'
175 int dof=0;
176 id_indx[0]=0;
177 if (size>1)
178     PetscPrintf(PETSC_COMM_WORLD,
179                 "dof distribution among processors\n");
180 for (int p=0; p<=size; p++) {
181     for (int j=0; j<nnod; j++)
182         if (id[j]==-(p+1)) id[j]=dof++;
183     id_indx[p+1] = dof;
184     if (p<size) {
185         PetscPrintf(PETSC_COMM_WORLD,
186                     "proc: %d, from %d to %d, total %d\n",
187                     p, id_indx[p], id_indx[p+1], id_indx[p+1]-id_indx[p]);
188     } else {
189         PetscPrintf(PETSC_COMM_WORLD,

```

```

190         "fixed: from %d to %d, total %d\n",
191         id_indx[p], id_indx[p+1], id_indx[p+1]-id_indx[p]);
192     }
193 }
194 n2e.clear();
195
196 int ierr;
197 // Total number of unknowns (equations)
198 int neq = id_indx[size];
199 // Number of unknowns in this processor
200 int ndof_here = id_indx[rank+1]-id_indx[rank];
201 // Creates a square MPI PETSc matrix
202 ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD, ndof_here, ndof_here,
203                        neq, neq, 0,
204                        PETSC_NULL, 0, PETSC_NULL, &A); CHKERRQ(ierr);
205 // Creates PETSc vectors
206 ierr = VecCreateMPI(PETSC_COMM_WORLD,
207                    ndof_here, neq, &b); CHKERRQ(ierr);
208 ierr = VecDuplicate(b, &u); CHKERRQ(ierr);
209 double scal=0.;
210 ierr = VecSet(&scal, b);
211
212 { // Compute element matrices and load them.
213     // Each processor computes the elements belonging to him.
214
215     // x12:= vector going from first to second node,
216     // x13:= idem 1->3.
217     // x1:= coordinate of node 1
218     // gN gradient of shape function
219     ColumnVector x12(ndim), x13(ndim), x1(ndim), gN(ndim);
220     // grad_N := matrix whose columns are gradients of interpolation

```

```

221 // functions
222 // ke:= element matrix for the Laplace operator
223 Matrix grad_N(ndim,nel), ke(nel,nel);
224 // area:= element area
225 double area;
226 for (int e=1; e<=nelem; e++) {
227     int ord=eord[e-1];
228     // skip if the element doesn't belong to this processor
229     if (ord < e_indx[rank] || ord >= e_indx[rank+1]) continue;
230     // indices for vertex nodes of this element
231     int n1,n2,n3;
232     n1 = icone[(e-1)*nel];
233     n2 = icone[(e-1)*nel+1];
234     n3 = icone[(e-1)*nel+2];
235     x1 << &xnod[(n1-1)*ndim];
236     x12 << &xnod[(n2-1)*ndim];
237     x12 = x12 - x1;
238     x13 << &xnod[(n3-1)*ndim];
239     x13 = x13 - x1;
240     // compute as vector product
241     area = (x12(1)*x13(2)-x13(1)*x12(2))/2.;
242     // gradients are proportional to the edge
243     // vector rotated 90 degrees
244     gN(1) = -x12(2);
245     gN(2) = +x12(1);
246     grad_N.Column(3) = gN/(2*area);
247     gN(1) = +x13(2);
248     gN(2) = -x13(1);
249     grad_N.Column(2) = gN/(2*area);
250     // last gradient can be computed from \sum_j grad_N_j = 0
251     grad_N.Column(1) = -(grad_N.Column(2)+grad_N.Column(3));

```

```

252      // Integrand is constant over element
253      ke = grad_N.t() * grad_N * area;
254
255      // Load matrix element on global matrix
256      int nod1,nod2,eq1,eq2;
257      for (int i1=1; i1<=nel; i1++) {
258          nod1 = icone[(e-1)*nel+i1-1];
259          if (fixa.find(nod1)!=fixa.end()) continue;
260          eq1=id[nod1-1];
261          for (int i2=1; i2<=nel; i2++) {
262              nod2 = icone[(e-1)*nel+i2-1];
263              eq2=id[nod2-1];
264              if (fixa.find(nod2)!=fixa.end()) {
265                  // Fixed nodes contribute to the right hand side
266                  VecSetValue(b,eq1,-ke(i1,i2)*fixa[nod2],ADD_VALUES);
267              } else {
268                  // Load on the global matrix
269                  MatSetValue(A,eq1,eq2,ke(i1,i2),ADD_VALUES);
270              }
271          }
272      }
273
274  }
275  // Finish assembly.
276  ierr = VecAssemblyBegin(b); CHKERRQ(ierr);
277  ierr = VecAssemblyEnd(b); CHKERRQ(ierr);
278
279  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
280  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
281  }
282
283  // Create SLES and set options

```



```

284 ierr = SLESCreate(PETSC_COMM_WORLD,&sles); CHKERRQ(ierr);
285 ierr = SLESSetOperators(sles,A,A,
286                        DIFFERENT_NONZERO_PATTERN); CHKERRQ(ierr);
287 ierr = SLESGetKSP(sles,&ksp); CHKERRQ(ierr);
288 ierr = SLESGetPC(sles,&pc); CHKERRQ(ierr);
289
290 ierr = KSPSetType(ksp,KSPGMRES); CHKERRQ(ierr);
291 // This only works for one processor only
292 // ierr = PCSetType(pc,PCLU); CHKERRQ(ierr);
293 ierr = PCSetType(pc,PCJACOBI); CHKERRQ(ierr);
294 ierr = KSPSetTolerances(ksp,1e-6,1e-6,1e3,100); CHKERRQ(ierr);
295 int its;
296 ierr = SLESSolve(sles,b,u,&its);
297
298 // Print vector on screen
299 ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
300
301 // Cleanup. Free memory
302 delete[] e_indx;
303 delete[] id_indx;
304 PetscFinalize();
305
306 }

```

SNES: non-linear solvers

```
1 /*$Id curspar-1.0.0-15-gabee420 Thu Jun 14 00:46:44 2007 -0300$*/
2
3 static char help[] =
4     "Newton's method to solve a combustion-like 1D problem.\n";
5
6 #include "petscsnes.h"
7
8 /*
9     User-defined routines
10 */
11 extern int resfun(SNES, Vec, Vec, void*);
12 extern int jacfun(SNES, Vec, Mat*, Mat*, MatStructure*, void*);
13
14 struct SnesCtx {
15     int N;
16     double k, c, h;
17 };
18
19 #undef __FUNCT__
20 #define __FUNCT__ "main"
21 int main(int argc, char **argv)
22 {
23     SNES          snes;          /* nonlinear solver context */
24     SLES          sles;          /* linear solver context */
25 }
```

```

25  PC          pc;          /* preconditioner context */
26  KSP          ksp;        /* Krylov subspace method context */
27  Vec          x,r;        /* solution, residual vectors */
28  Mat          J;          /* Jacobian matrix */
29  int          ierr, its, size;
30  PetscScalar pfive = .5, *xx;
31  PetscTruth flg;
32  SnesCtx ctx;
33  int N = 10;
34  ctx.N = N;
35  ctx.k = 0.1;
36  ctx.c = 1;
37  ctx.h = 1.0/N;
38
39  PetscInitialize(&argc, &argv, (char *) 0, help);
40  ierr = MPI_Comm_size(PETSC_COMM_WORLD, &size); CHKERRQ(ierr);
41  if (size != 1) SETERRQ(1, "This is a uniprocessor example only!");
42
43  // Create nonlinear solver context
44  ierr = SNESCreate(PETSC_COMM_WORLD, &snes); CHKERRQ(ierr);
45  ierr = SNESSetType(snes, SNESLS); CHKERRQ(ierr);
46
47  // Create matrix and vector data structures;
48  // set corresponding routines
49
50  // Create vectors for solution and nonlinear function
51  ierr = VecCreateSeq(PETSC_COMM_SELF, N+1, &x); CHKERRQ(ierr);
52  ierr = VecDuplicate(x, &r); CHKERRQ(ierr);
53  double scal = 0.9;
54  ierr = VecSet(&scal, x); CHKERRQ(ierr);
55
56  ierr = MatCreateMPIAIJ(PETSC_COMM_SELF, PETSC_DECIDE,
57                        PETSC_DECIDE, N+1, N+1,

```

```

58             1, NULL, 0, NULL, &J); CHKERRQ(ierr);
59
60     ierr = SNESSetFunction(snes, r, resfun, &ctx); CHKERRQ(ierr);
61     ierr = SNESSetJacobian(snes, J, J, jacfun, &ctx); CHKERRQ(ierr);
62
63     ierr = SNESolve(snes, x, &its); CHKERRQ(ierr);
64     Vec f;
65     ierr = VecView(x, PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
66     ierr = SNESGetFunction(snes, &f, 0, 0); CHKERRQ(ierr);
67     double rnorm;
68     ierr = VecNorm(r, NORM_2, &rnorm);
69
70     ierr = PetscPrintf(PETSC_COMM_SELF,
71                       "number of Newton iterations = "
72                       "%d, norm res %g\n",
73                       its, rnorm); CHKERRQ(ierr);
74
75     ierr = VecDestroy(x); CHKERRQ(ierr);
76     ierr = VecDestroy(r); CHKERRQ(ierr);
77     ierr = SNESDestroy(snes); CHKERRQ(ierr);
78
79     ierr = PetscFinalize(); CHKERRQ(ierr);
80     return 0;
81 }
82
83 /* ----- */
84 #undef __FUNCT__
85 #define __FUNCT__ "resfun"
86 int resfun(SNES snes, Vec x, Vec f, void *data)
87 {
88     double *xx, *ff;
89     SnesCtx &ctx = *(SnesCtx *) data;
90     int ierr;

```

```

91  double h = ctx.h;
92
93  ierr = VecGetArray(x, &xx); CHKERRQ(ierr);
94  ierr = VecGetArray(f, &ff); CHKERRQ(ierr);
95
96  ff[0] = xx[0];
97  ff[ctx.N] = xx[ctx.N];
98
99  for (int j=1; j<ctx.N; j++) {
100     double xxx = xx[j];
101     ff[j] = xxx*(0.5-xxx)*(1.0-xxx);
102     ff[j] += ctx.k*(-xx[j+1]+2.0*xx[j+1]-xx[j-1])/(h*h);
103  }
104
105  ierr = VecRestoreArray(x, &xx); CHKERRQ(ierr);
106  ierr = VecRestoreArray(f, &ff); CHKERRQ(ierr);
107  return 0;
108 }
109
110 /* ----- */
111 #undef __FUNCT__
112 #define __FUNCT__ "resfun"
113 int jacfun(SNES snes, Vec x, Mat* jac, Mat* jac1,
114            MatStructure *flag, void *data) {
115     double *xx, A;
116     SnesCtx &ctx = *(SnesCtx *)data;
117     int ierr, j;
118
119     ierr = VecGetArray(x, &xx); CHKERRQ(ierr);
120     ierr = MatZeroEntries(*jac);
121
122     j=0; A = 1;
123     ierr = MatSetValues(*jac, 1, &j, 1, &j, &A,

```

```

124             INSERT_VALUES); CHKERRQ(ierr);
125
126     j=ctx.N; A = 1;
127     ierr = MatSetValues(*jac, 1, &j, 1, &j, &A,
128             INSERT_VALUES); CHKERRQ(ierr);
129
130     for (j=1; j<ctx.N; j++) {
131         double xxx = xx[j];
132         A = (0.5-xxx)*(1.0-xxx) - xxx*(1.0-xxx) - xxx*(0.5-xxx);
133         ierr = MatSetValues(*jac, 1, &j, 1, &j, &A, INSERT_VALUES);
134         CHKERRQ(ierr);
135     }
136     ierr = MatAssemblyBegin(*jac, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
137     ierr = MatAssemblyEnd(*jac, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
138     ierr = VecRestoreArray(x, &xx); CHKERRQ(ierr);
139     return 0;
140 }

```

Assignment

REQUIRED Assignment Nbr. 1. MPI

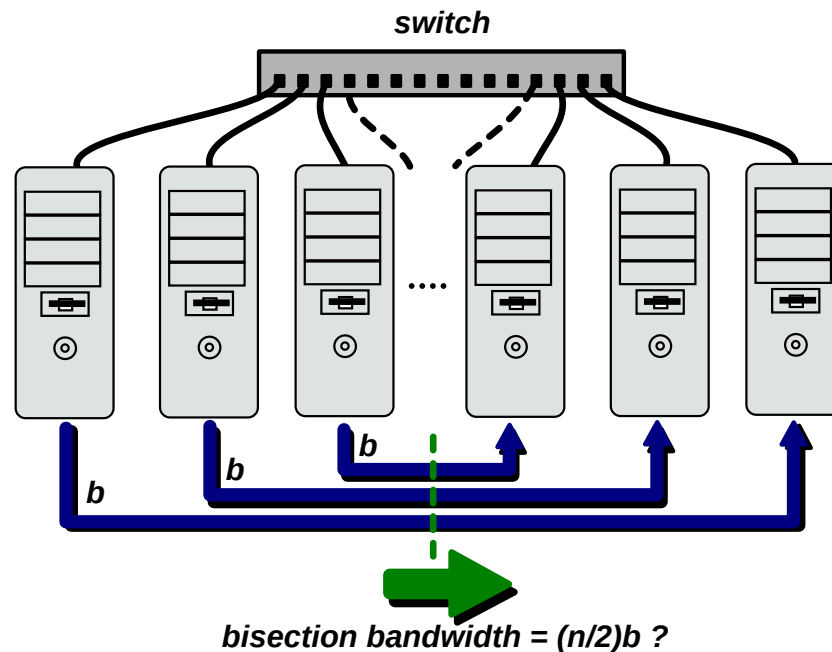
EXERCISE 1. Determine Network params

Given two processors P_0 and P_1 the time needed to send a message from P_0 to P_1 is a function of the message length $T_{\text{comm}} = T_{\text{comm}}(n)$, where n is the number of bytes in the message. If we approximate this relation by a linear relation, then

$$T_{\text{comm}}(n) = l + n/b$$

where l is the **latency** and b is **bandwidth**. Both parameters depend on the hardware and software of the network. (TCP/IP layer in Linux) and the message passing library (MPI).

REQUIRED Assignment Nbr. 1. MPI (cont.)



The **bisection bandwidth** of a cluster is the transfer speed with which data is transferred simultaneously from $n/2$ processors to other $n/2$ processors. Assuming that the network is switched and all processors are connected to the same switch, the transfer speed should be $(n/2)b$, but it may happen that the switch has a maximum internal transfer rate.

REQUIRED Assignment Nbr. 1. MPI (cont.)

To do:

1A: Find the bandwidth and latency of the network writing an MPI program that sends packets of different size and performs a linear regression with the resulting data. Obtain the parameters for each pair of processors in the cluster. Compare with the nominal values of the network (e.g. for Gigabit Ethernet $b \approx 1 \text{ Gbit/sec}$, $l = O(100\text{usec})$, for Myrinet $b \approx 2 \text{ Gbit/sec}$, $l = O(3\text{usec})$).

2B: For a given n , try the following partitions by half and determine their bisection bandwidth. (Assumes n even).

- Low/High partition: $[0, n/2)$ and $[n/2, n)$
- Even/Odd partition.

REQUIRED Assignment Nbr. 1. MPI (cont.)

EXERCISE 2. The scatter function.

Write an MPI program that fills a vector with N random integers in the server and then scatters them according to the following rule: integer k goes to processor $k \% size$ where $k \% size$ is the C notation for the remainder of the division of k by $size$, and $size$ is the number of procs in the parallel run. As a special case, when $size=2$ even numbers go to proc 0, and odd numbers go to proc 1.

REQUIRED Assignment Nbr. 1. MPI (cont.)

EXERCISE 3. Solving the 1D biharmonic eq. with PETSc

The deflection of a beam u is governed by the *biharmonic equation*

$$EIu^{IV} = f \quad (2)$$

where E, I are constants (Young Modulus and inertia moment of the cross section). A finite difference on a constant step size h 1D grid is

$$EI \frac{u_{j+2} - 4u_{j+1} + 6u_j - 4u_{j-1} + u_{j-2}}{h^4} = f_j \quad (3)$$

Solve this equation for a beam of length L , and E, I, f constant, and boundary conditions $u, u' = 0$ (clamped) at both edges. For the u' boundary condition use the following lateral, 2nd order approximations:

$$\begin{aligned} u'_0 &\approx \frac{-3u_0 + 4u_1 - u_2}{h} \\ u'_N &\approx \frac{u_{N-2} - 4u_{N-1} + 3u_N}{h} \end{aligned} \quad (4)$$

REQUIRED Assignement Nbr. 1. MPI (cont.)

EXERCISE 4. The PNT problem with OpenMP

- Try different schedulings (dynamic, statis, chunk of various sizes, guided) for the TNP problem. Compute efficiency and discuss.

FAQ, tips and common pitfalls with MPI, PETSc and OpenMP

FAQ, tips, pitfalls

- **CONFIGURATION OF MPI. THIS IS VERY IMPORTANT:** Use the *openmpi-intel-1.2.4* configuration and submit jobs ONLY to the parallel2 queue.

```
1 $ switcher mpi = openmpi-intel-1.2.4
2 $ module load openmpi-intel-1.2.4
```

- **Initialize the tag in point-to-point communication:** Look at the following code where processor 0 is sending some data to processor 1.

```
1 int mtag; // WRONG: mtag is not initialized!!
2 if (myrank==0) MPI_Send(buff, . . . , mtag, . . . .);
3 else if (myrank==1) MPI_Recv(buff, . . . , mtag, . . . . , &status);
```

The problem is that *mtag* is undefined, because the user has not initialized it. So it may have a certain random value in one processor and another *different* value in other processor. The fact is that even if you don't plan to use the tag in order to select the message *you must assure that the tag has the same value in both processors*. Otherwise the message will not be received by processor 1 and the symptom is that execution is deadlocked at that point. The solution is to assign a value to the tag, i.e. replace the first line by *int mtag=0;*.

- **Put barriers when taking time:**

Suppose you want to take times in a ping-pong, for instance

```

1 double start = MPI_Wtime();
2 if (myrank==0) {
3   MPI_Send(buff, . . . , mtag, . . . .);
4   MPI_Recv(buff, . . . , mtag, . . . .);
5 } else if (myrank==1) {
6   MPI_Recv(buff, . . . , mtag, . . . .);
7   MPI_Send(buff, . . . , mtag, . . . .);
8 }
9 double elapsed = MPI_Wtime() - start;

```

Note that the times computed in each processor (i.e. the values of *elapsed*) may be different in both processors, because the receive in processor 0 probably ends *after* the send at processor 1. Probably the best is to take the time in processor 0, because the message starts and ends there. However the best is to put barriers before and after the whole round-trip communication.

```

1 MPI_Barrier();
2 double start = MPI_Wtime();
3 if (myrank==0) {
4   MPI_Send(buff, . . . , mtag, . . . .);
5   MPI_Recv(buff, . . . , mtag, . . . .);
6 } else if (myrank==1) {
7   MPI_Recv(buff, . . . , mtag, . . . .);

```



```

8  MPI_Send(buff, . . . , mtag, . . . .) ;
9  }
10 MPI_Barrier() ;
11 double elapsed = MPI_Wtime() - start;

```

This guarantees that all the communication is done in between the times are taken. Note also that the calls to the time routine `MPI_Wtime()` are done always *after* the barrier.

- **Make statistics:** When taking time, specially if the expected times are very small, it's better to make statistics by repeating the operation many times, i.e. for instance for a ping-pong exchange

```

1  int ntimes=100;
2  MPI_Barrier() ;
3  double start = MPI_Wtime() ;
4  for (int j=0; j<ntime; j++) {
5      if (myrank==0) {
6          MPI_Send(buff, . . . , mtag, . . . ., &stat) ;
7          MPI_Recv(buff, . . . , mtag, . . . .) ;
8      } else if (myrank==1) {
9          MPI_Recv(buff, . . . , mtag, . . . .) ;
10         MPI_Send(buff, . . . , mtag, . . . ., &stat) ;
11     }
12 }
13 MPI_Barrier() ;
14 double elapsed = (MPI_Wtime() - start) / ntimes;

```

- **If read from stdin, do it in master and then broadcast to slaves:** For instance

```

1 if (!rank) {
2     printf("enter N");
3     scanf("%d", &N);
4     printf("Entered N = %d\n", N);
5 }
6 MPI_Bcast (&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
    
```

- **Correct code for two-point communication:** If you want to communicate from server *i* to server *j* (and the other processors don't participate), for instance a round-trip ping-pong, then the correct code is

```

1 if (rank==i) {
2     MPI_Send(&buff, . . . , j, . . . );
3     MPI_Recv(&buff, . . . , j, . . . );
4 }
5 if (rank==j) {
6     MPI_Recv(&buff, . . . , i, . . . );
7     MPI_Send(&buff, . . . , i, . . . );
8 }
    
```

Note the conditionals *if (rank==i) {}* and *if (rank==j) {}*. They avoid the other processors to participate in the communication.

- **Network parameters in clonetroop:** Beware that clonetroop is **Gigabit Ethernet**, not **Infiniband** (as I erroneously asserted in the class). Also, take

into account that the processors in clonetroop are *multicore* (8 cores per processor). If you set in the PBS file the value to the *ppn* parameter (*processes per node*) to a value ≥ 1 , for instance *-l nodes4:ppn2* that means that PBS will launch 8 processes on 4 nodes, with 2 process in each node, so that you will get processes 0 and 1 in the first nodes, 2 and 3 in the second, and so on. But then, if you measure the bandwidth between node 2 and all the other processes, you will find that *the bandwidth with process 3 is much larger than with the other processes*. This is because in that case the data goes from one core in the node to another core in the same node and is much larger than the communication through the network.

- ***Beware to make collective calls be called in all processors:*** For instance if in the code before make the ping-pong between some nodes *i, j*, if you put barriers like this

```

1 // WRONG: MPI_Barrier() is called
2 // only in processors i and j
3 if (rank==i) {
4     MPI_Barrier();
5     MPI_Send(&buff, . . . , j, . . . );
6     MPI_Recv(&buff, . . . , j, . . . );
7     MPI_Barrier();
8 }
9 if (rank==j) {

```

```

10 MPI_Barrier();
11 MPI_Recv(&buff, . . . , i, . . . );
12 MPI_Send(&buff, . . . , i, . . . );
13 MPI_Barrier();
14 }

```

then you are calling the *MPI_Barrier()* function *only* in processors *i, j* and not in the others. The correct call is:

```

1 // CORRECT: MPI_Barrier() is called in all processors
2 MPI_Barrier();
3 if (rank==i) {
4     MPI_Send(&buff, . . . , j, . . . );
5     MPI_Recv(&buff, . . . , j, . . . );
6 }
7 if (rank==j) {
8     MPI_Recv(&buff, . . . , i, . . . );
9     MPI_Send(&buff, . . . , i, . . . );
10 }
11 MPI_Barrier();

```

- **Running MPI jobs in clonetroop:** In the *clonetroop* cluster you have to submit your jobs with via PBS (Portable Batch System). Read the instructions in the [HOWTO](#). Some details
 - ▷ Basic use of PBS is explained in the HOWTO. A more detailed description of PBS can be found in this [tutorial](#), or the [man page](#).
 - ▷ There are several possibilities of combinations of MPI implementations

and compilers. I did it work with the *switcher* mpi option set to *openmpi-intel-1.2.4* and launching the jobs to the parallel2 queue.

- ▷ The option *1 nodes=8:ppn=2* means that you are requesting 8 nodes with 2 processes in each node, so that you end up with 16 process, i.e. *MPI_Comm_rank()* will return 16.
- ▷ When measuring bandwidths take into account that process that are in the same node will have a bandwidth much larger than that for processes not in the same node.
- ▷ If you want to test OpenMP code you can do some **VERY SIMPLE** and **SHORT** runs in the server *clonetroop*, but remember that the spirit of the cluster is to leave the server only for editing and non cpu time consuming tasks. You should make the runs with PBS, using the option *1 nodes=1:ppn=1*, and not using *mpirun*. You can select the number of threads via the *OMP_NUM_THREADS* environment variable or either via the *omp_set_num_threads()* function.
- ▷ For OpenMP remember to use the *-fopenmp* with the GNU compiler or the *-openmp* with the Intel compiler.
- ▷ For running PETSc programs copy the *Makefile*, the C++ program *ex1.cpp*, and PBS script *job.sh* from directory */clonetroop/storti/petsc-examples* and start modifying them. For

compiling the program you do *make ex23.bin*. For submitting to PBS it is the same as for any MPI program.

- ▷ Regarding PETSc, in the new versions there is no SLES structures since they have been converted directly to KSP, so that the code in the slides and in the examples have been updated. Basically the calling sequence is something like this

```
1 KSP ksp;  
2 KSPCreate (PETSC_COMM_WORLD, &ksp) ;  
3 ierr = KSPSetOperators (KSP, A, A,  
4                         DIFFERENT_NONZERO_PATTERN) ;  
5  
6 PC pc;  
7 KSPGetPC (KSP, &pc) ;  
8 PCSetType (pc, PCJACOBI) ;  
9 KSPSetTolerances (ksp, 1.e-7, PETSC_DEFAULT,  
10                  PETSC_DEFAULT, PETSC_DEFAULT) ;  
11  
12 KSPSolve (ksp, b, x, &its) ;  
13  
14 ierr = KSPDestroy (ksp) ; CHKERRQ (ierr) ;
```

Anyway, it is supposed that you *copy* the examples *ex1.cpp* or *ex23.cpp* that are in */clonetroop/storti/petsc-examples*, and these are correct.

- **Some considerations on the PETSc example:**

- ▷ You have to use the interior 5 point stencil (3) in all the points except 2

at each end.

- ▷ At those points you impose $u = 0$ and $u' = 0$ using the 2nd order approximation given.
- ▷ You can solve for $f/EI = 1$ and $L = 1$ since it is easy to scale to any other value, due to the linearity of the problem.
- ▷ For comparison the exact solution is

$$u_{\text{exact}} = \frac{1}{24}x^2(1 - x^2), \quad (5)$$

so that it is symmetric and the maximum deflection is obtained at the center $x = 0.5$ and is

$$u_{\text{exact,max}} = \frac{1}{24 \cdot 16} = 0.0026042 \quad (6)$$

- ▷ If you solve in only one processor you can use the **PCLU** preconditioner which means using a direct solver,

```
1 ierr = PCSetType(pc, PCJACOBI); CHKERRQ(ierr);
```

- ▷ If you solve in parallel, then you **can not** use the direct solver, and for instance you can use the **PCJACOBI** preconditioner,

```
1 ierr = PCSetType(pc, PCJACOBI); CHKERRQ(ierr);
```

In this case take care of using enough iterations for GMRES

```
1 ierr = KSPSetTolerances(ksp, 1.e-7, PETSC_DEFAULT,  
2                        PETSC_DEFAULT, 100); CHKERRQ(ierr);
```

Check the convergence of the iterative process

```
1 ierr = KSPMonitorSet(ksp, KSPMonitorDefault, NULL, NULL);
```

This will print the norm of the residual vector at each GMRES iteration.