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 "texstuff-1.0.36-28-g080dfb4")`

July 18, 2011

Contents

CIMEC-INTEC

- **CIMEC (Centro Internacional de Métodos Computacionales en Ingeniería/*International Center for Computational Methods in Engineering* is a resarch laboratory that is part of the INTEC Institute. (15 researchers approx., 25 fellowship).**
- **INTEC (Instituto de Desarrollo Tecnológico para la Industria Química/*Institute for Technology in the Chemical Industry*) is an Institute with approx. 150 researchers in Chemical Engineering, Computational Mechanics, and Physics.**
- **CIMEC-INTEC is dependent of CONICET and UNL.**
- **CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas/*National Council for Scientific and Technical Research*) is the main institution in Argentina for Scientific Research.**
- **UNL (Universidad Nacional del Litoral/*Littoral National University*) is the public argentine university located at the city of Santa Fe.**

Institutions participating at IRSES Workshop

- **IPPT-PAN: Instytut Podstawowych Problemów Techniki (IPPT), Polskiej Akademii Nauk (PAN).**
- **TU Braunschweig: Technische Universität Carolo-Wilhelmina zu Braunschweig.**
- **TU Graz: Technische Universität Graz.**
- **PUC-Rio: Pontifícia Universidade Católica do Rio de Janeiro**
- **USP: Universidade de São Paulo**
- **USACH: Universidad de Santiago de Chile.**
- **INTEMA: Instituto de Investigaciones en Ciencia y Tecnología de Materiales.**

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 stantards for Message Passing and distributed memory environments. As a result, an agreement was achieved on estabilishing a standard for Message Passing.

The MPI Forum (cont.)

- **In November 1992 in the *Supercomputing'92* conference a committe was appointed to define a *standard of Message Passing*. The objectives where:**
  - ▷ **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, asisting 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 where 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. Skeljumm. 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`*).
- ***Parallell 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*).**

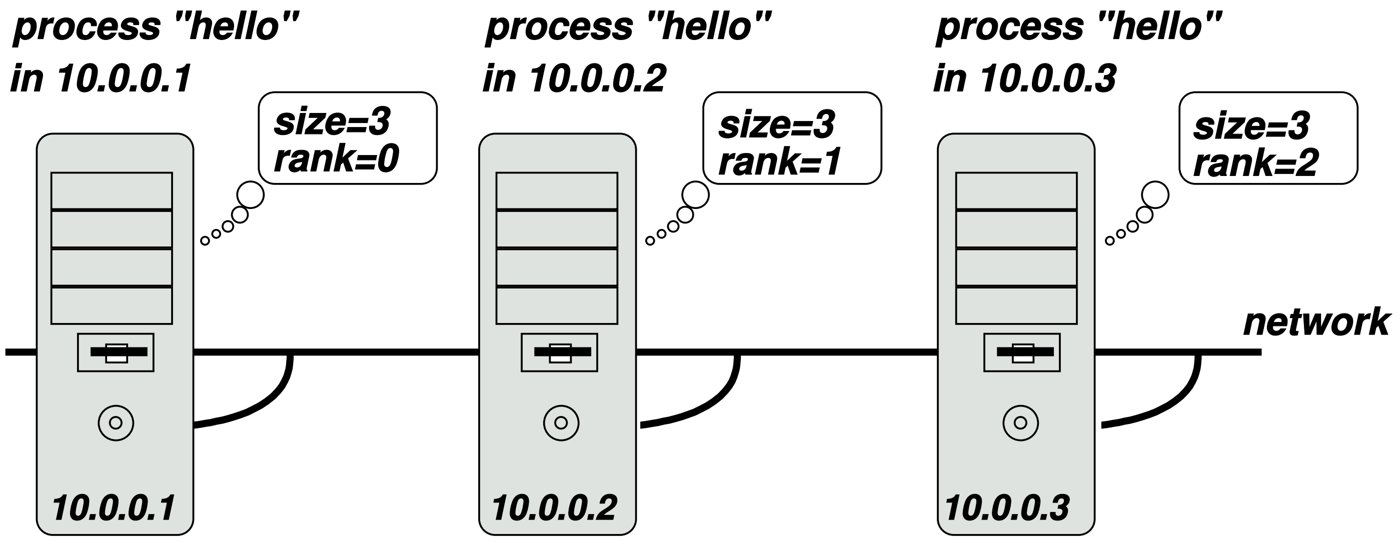
Basic use of MPI

Hello world in C

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

int main(int argc, char **argv) {
  int ierror, rank, size;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  printf("Hello world. I am %d out of %d.\n",rank,size);
  MPI_Finalize();
}
```

- **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<=myrank<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.

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

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

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

I/O

- 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

```
      PROGRAM hello
      IMPLICIT NONE
      INCLUDE "mpif.h"
      INTEGER ierror, rank, size
      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
      WRITE(*,*) 'Hello world. I am ',rank,' out of ',size
      CALL MPI_FINALIZE(ierror)
      STOP
      END
```

Master/Slave strategy with SPMD (in C)

```c
// ...
int main(int argc, char **argv) {
   int ierror, rank, size;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   // ...
   if (rank==0) {
      /* master code */
   } else {
      /* slave code */
   }
   // ...
   MPI_Finalize();
}
```

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:**

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

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.

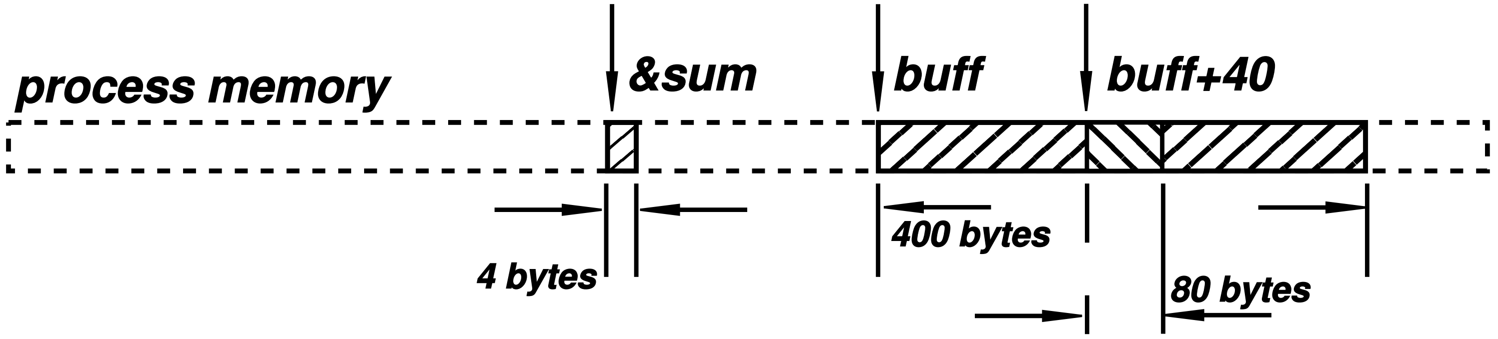
The complete MPI standard has *125 functions*.

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.)



```
int buff[100];
// Fill buff. .
for (int j=0; j<100; j++) buff[j] = j;
ierr = MPI_Send(buff, 100, MPI_INT, 0, mtag1,
      MPI_COMM_WORLD);

int sum;
ierr = MPI_Send(&sum, 1, MPI_INT, 0, mtag2,
      MPI_COMM_WORLD);

ierr = MPI_Send(buff+40, 20, MPI_INT, 0, mtag2,
      MPI_COMM_WORLD);

ierr = MPI_Send(buff+80, 40, MPI_INT, 0, mtag2,
      MPI_COMM_WORLD); // Error! Region sent extends
                      // beyond the end of buff
```

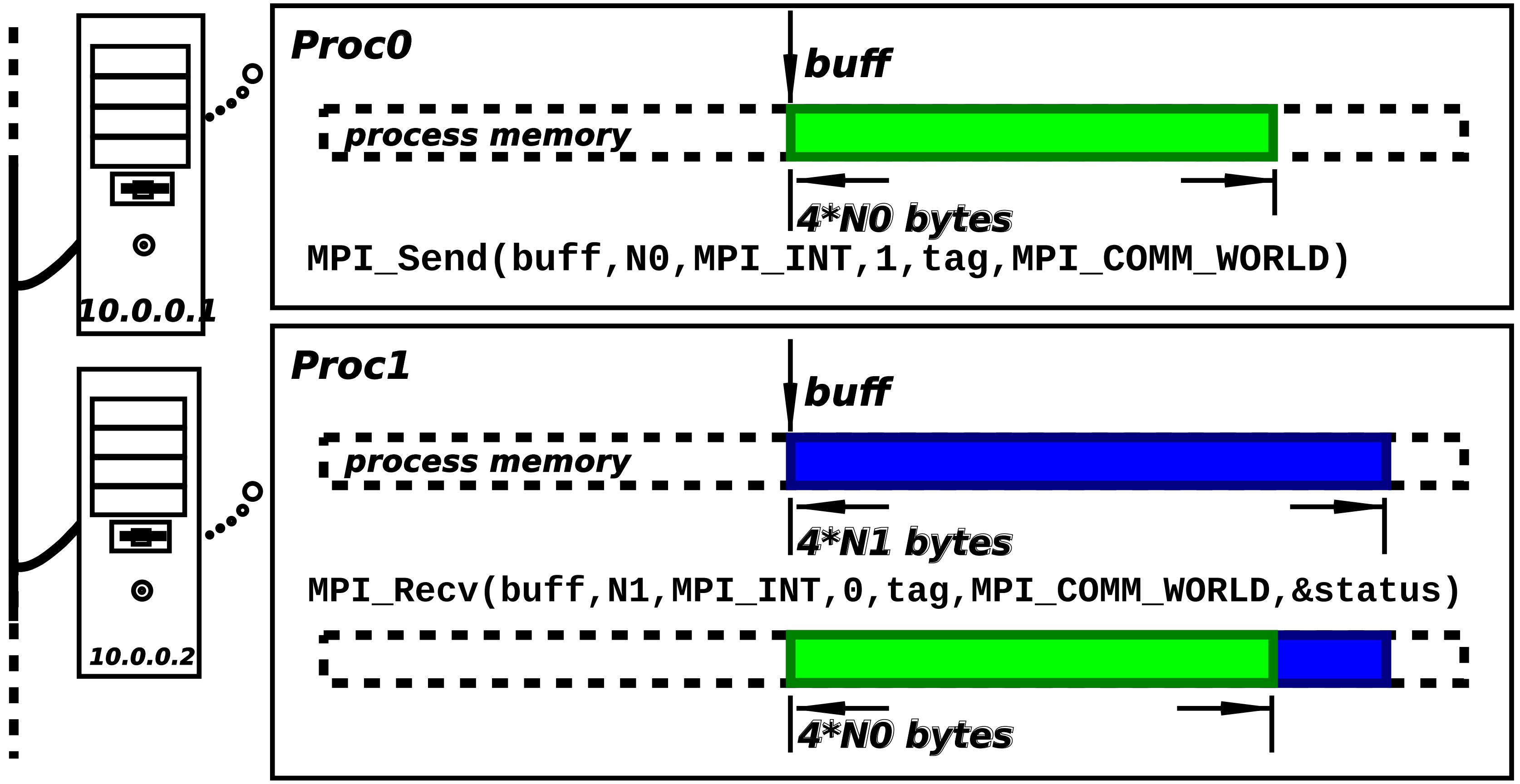
Receive a message

- **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.)



```
MPI_Send(buff,N0,MPI_INT,1,tag,MPI_COMM_WORLD)
```

```
MPI_Recv(buff,N1,MPI_INT,0,tag,MPI_COMM_WORLD,&status)
```

OK if `N1>=N0`

send/receive parameters (cont.)

- `tag` message indentifier
- `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

```c
MPI_Status status;
. . .
MPI_Recv(. . .,MPI_ANY_SOURCE, . . .,&status);
source = status.MPI_SOURCE;
printf("I got %f from process %d\n", result, source);
```

- an integer array in Fortran

```fortran
      integer status(MPI_STATUS_SIZE)
c . . .
      call MPI_RECV(result, 1, MPI_REAL, MPI_ANY_SOURCE,
                           mtag1, MPI_COMM_WORLD, status, ierr)
      source = status(MPI_SOURCE)
      print *, 'I got ', result, ' from ', source
```

Point-to-point communication

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

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

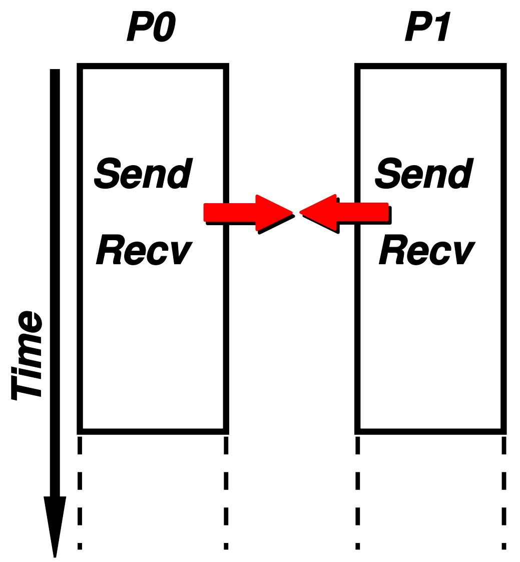
When a message is received?

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

`envelope = source/destination, tag, communicator`

- **size(receive buffer) $<$ size(data sent) $\longrightarrow$ *error***
- **size(receive buffer) $\geq$ size(data sent) $\longrightarrow$ *OK***
- **types don’t match $\longrightarrow$ *error***

Dead-lock

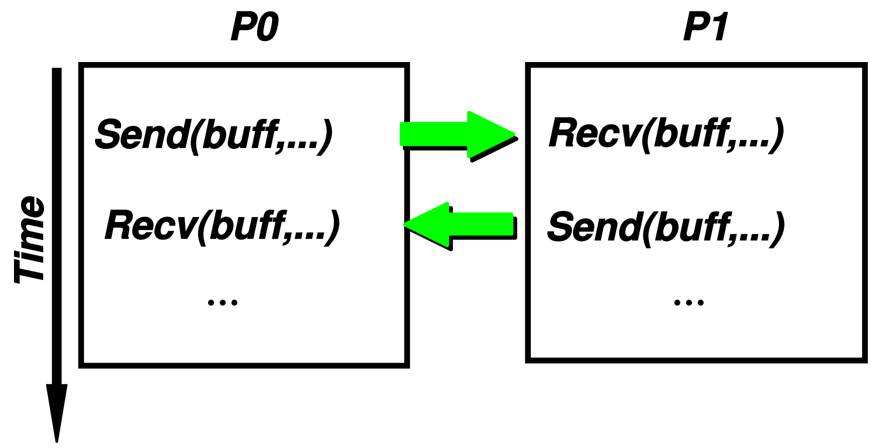


```
MPI_Send(buff,length,MPI_FLOAT,!myrank,
         tag,MPI_COMM_WORLD);
MPI_Recv(buff,length,MPI_FLOAT,!myrank,
         tag,MPI_COMM_WORLD,&status);
```

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

`MPI_Send` and `MPI_Recv` are *blocking*, This means that code execution doesn advance until the sending/reception is *completed*.

Correct calling order

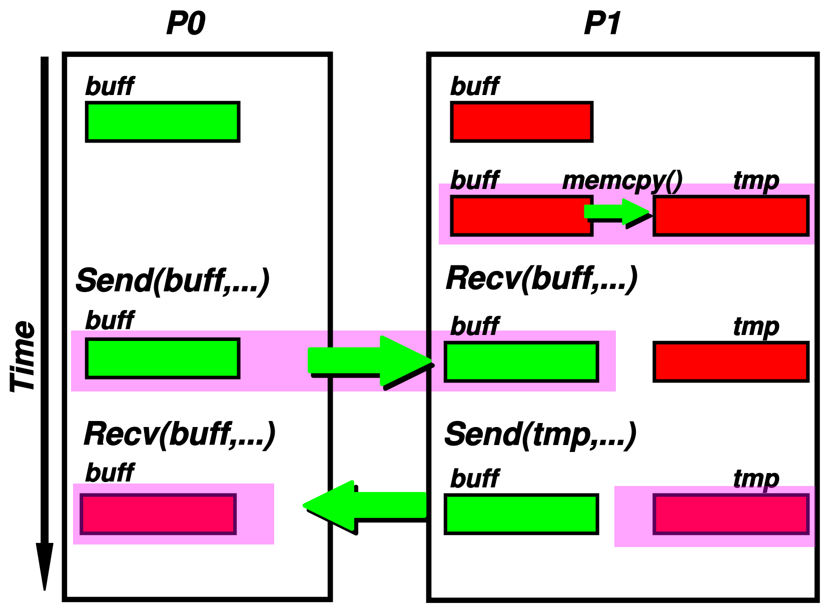


```
if (!myrank) {
  MPI_Send(buff,length,MPI_FLOAT,!myrank,
           tag,MPI_COMM_WORLD);
  MPI_Recv(buff,length,MPI_FLOAT,!myrank,
           tag,MPI_COMM_WORLD,&status);
} else {
  MPI_Recv(buff,length,MPI_FLOAT,!myrank,
           tag,MPI_COMM_WORLD,&status);
  MPI_Send(buff,length,MPI_FLOAT,!myrank,
           tag,MPI_COMM_WORLD);
}
```

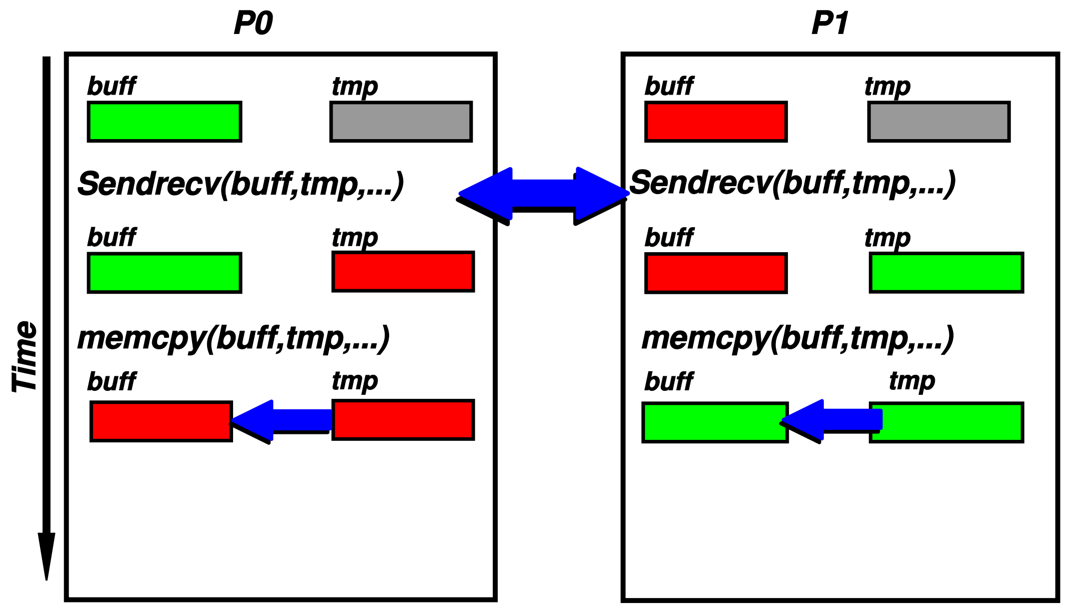
Correct calling order (cont.)

The previous code erroneously *overwrites* the reception buffer. We need a *temporal buffer* `tmp`:

```
if (!myrank) {
  MPI_Send(buff,length,MPI_FLOAT,
    !myrank,tag,MPI_COMM_WORLD);
  MPI_Recv(buff,length,MPI_FLOAT,
    !myrank,tag,MPI_COMM_WORLD,
    &status);
} else {
  float *tmp =new float[length];
  memcpy(tmp,buff,
        length*sizeof(float));
  MPI_Recv(buff,length,MPI_FLOAT,
   !myrank,tag,MPI_COMM_WORLD,
   &status);
  MPI_Send(tmp,length,MPI_FLOAT,
   !myrank,tag,MPI_COMM_WORLD);
  delete[] tmp;
}
```



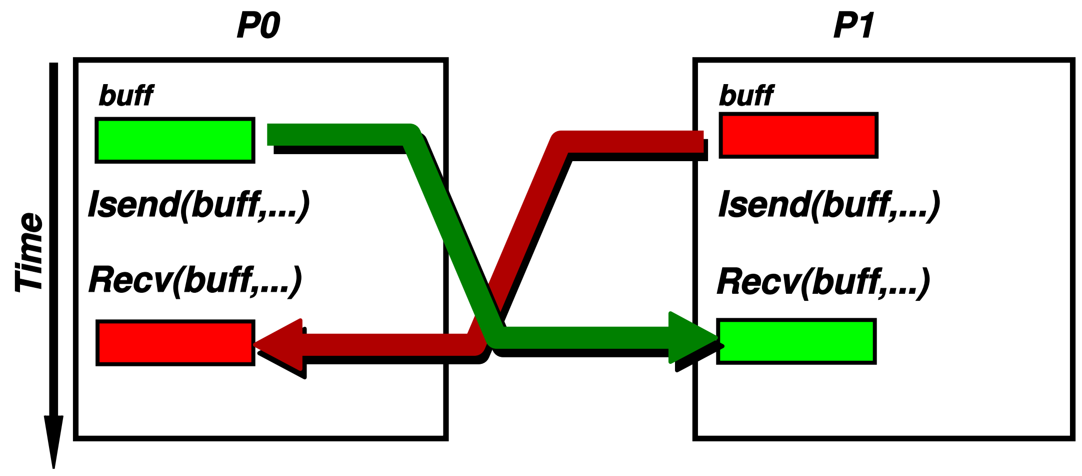
Correct calling order (cont.)



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

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

Correct calling order (cont.)



Use *non-blocking* send/receive

```
MPI_Request request;
MPI_Isend(....,request);
MPI_Recv(....);
while(1) {
  /* do something */
  MPI_Test(request,flag,status);
  if(flag) break;
}
```

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

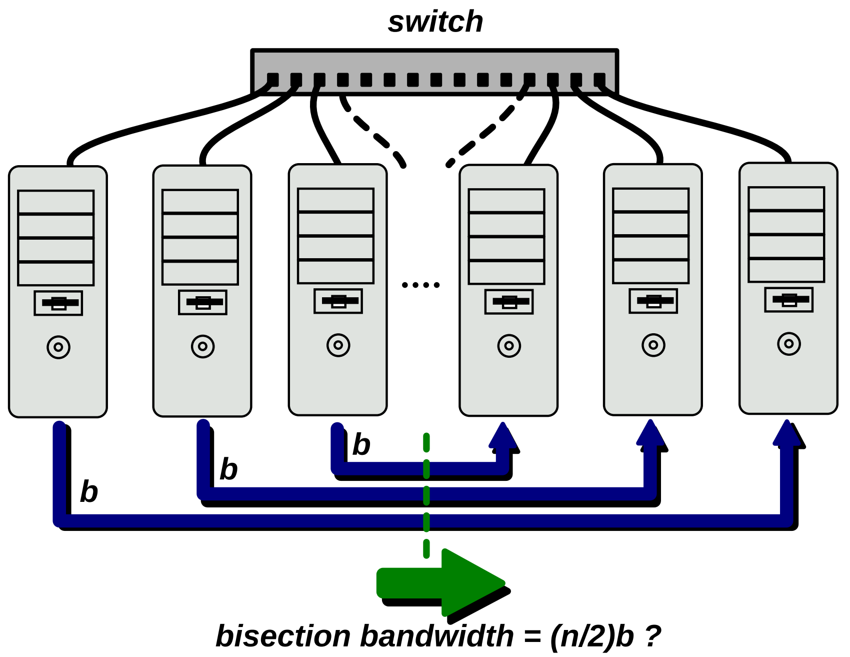
OPTIONAL Assignement Nbr. 1

Given two processors $P0$ and $P1$ the time needed to send a message from $P0$ to $P1$ is a function of the *message length* $T_{\rm comm} = T_{\rm comm}(n)$, where n is the number of bytes in the message. If we approximate this relation by a *linear relation*, then

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

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).

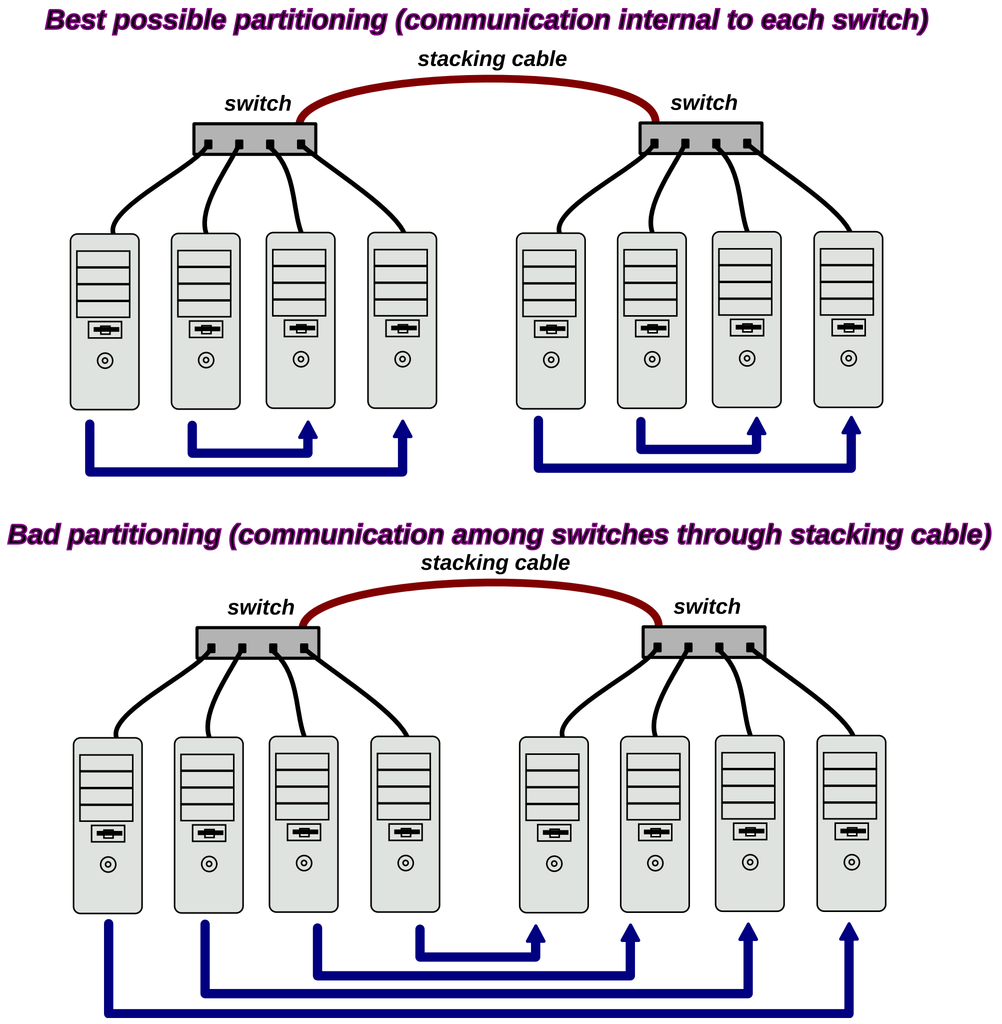
OPTIONAL Assignement Nbr. 1 (cont.)



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 swítch has a maximum internal transfer rate.

OPTIONAL Assignement Nbr. 1 (cont.)

Also, it may happen that some subset of nodes have a better bandwidth among them that with others, for instance if the network is connected by two switches of $n/2$ ports, *stacked* by a cable of transfer rate lower than $(n/2)b$. In that case the bisection bandwidth is defined as the *worst transfer rate* among all possible segmentations.


Best possible partitioning (communication internal to each switch)
stacking cable
switch
switch
Bad partitioning (communication among switches through stacking cable)
stacking cable
switch
switch

OPTIONAL Assignement Nbr. 1 (cont.)

To do:

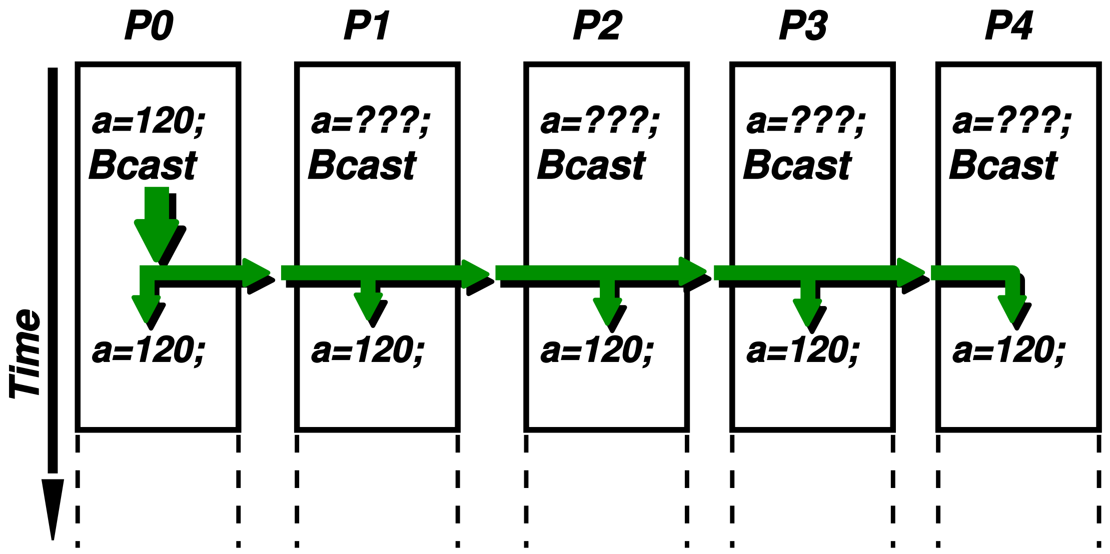
Part 1: **Find the *bandwidth* and *latency* of the network writing an MPI program that sends packets of different size and performs a *linear regression* with the resulting data. Obtain the parameters for each pair of processors in the cluster. Compare with the nominal values of the network (e.g. for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

Part 2: **Take an increasing number of** n **processors and divide them arbitrarily in two sets and take the transfer rate between both sets for that partition. Plot the trasfer rate against** n **and check if it grows linearly with** n**, or if there is an internal limit in the bandwidth for that switch.**

Part 3: **For a given** n**, try different partitions and detect if some of them have a greater transfer rate than others. May be a good point to start detecting this anomalies is to analyze the transfer rate matrix obtained in Part 1.**

Global communication

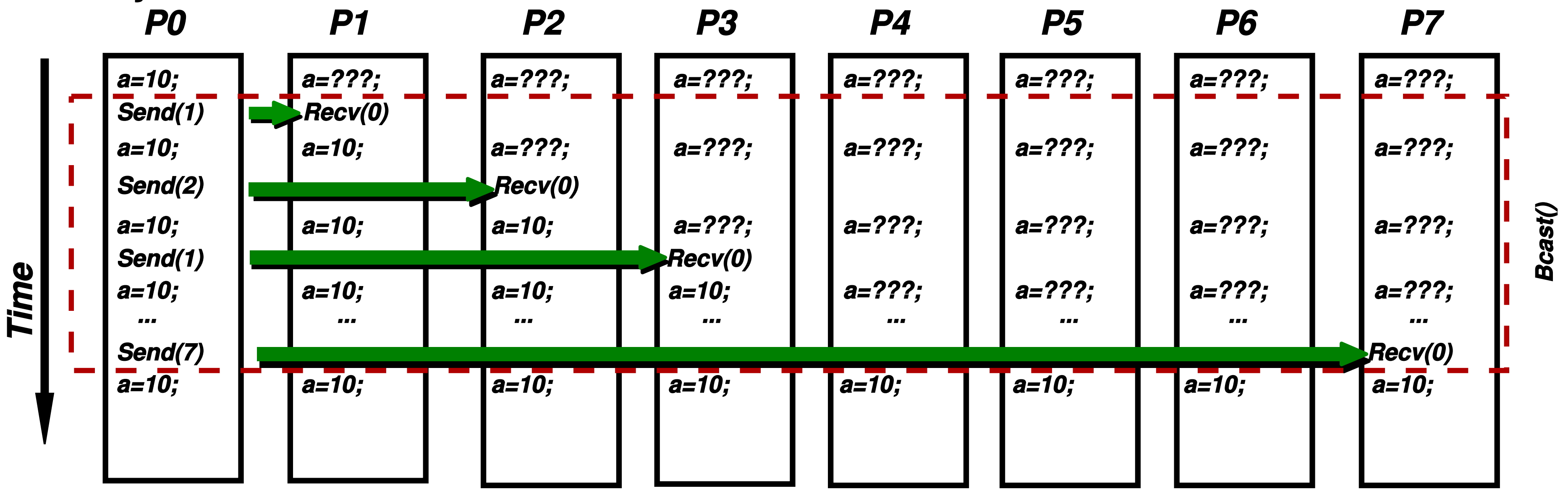
Message broadcast



- **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)`

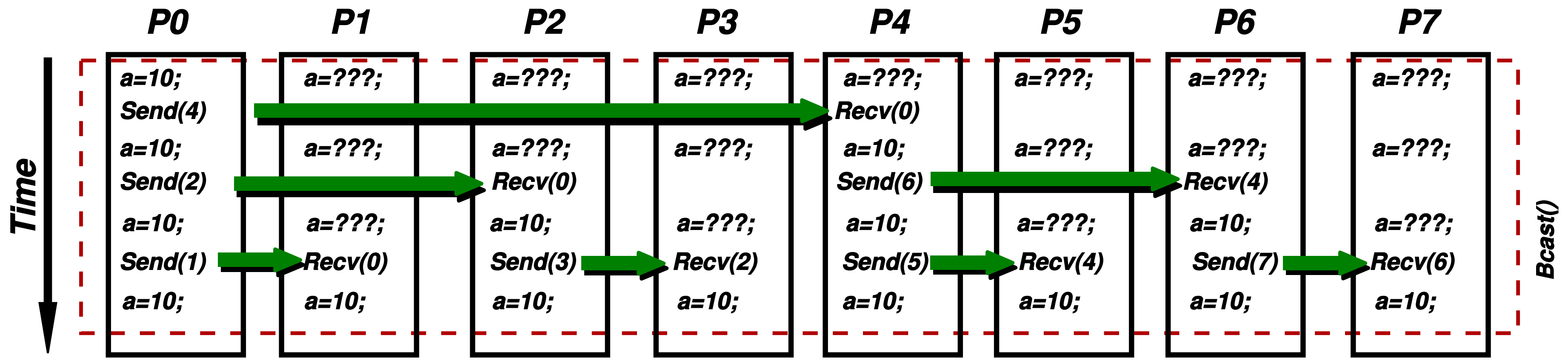
Message broadcast (cont.)

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

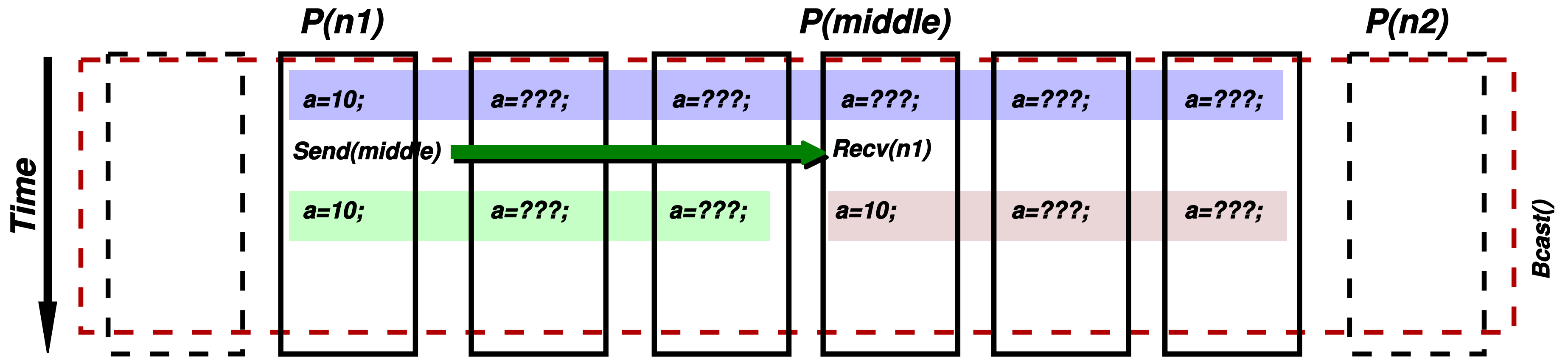


```
if (!myrank) {
  for (int j=1; j<numprocs; j++) MPI_Send(buff,...,j);
} else {
  MPI_Recv(buff,...,0...);
}
```

Message broadcast (cont.)



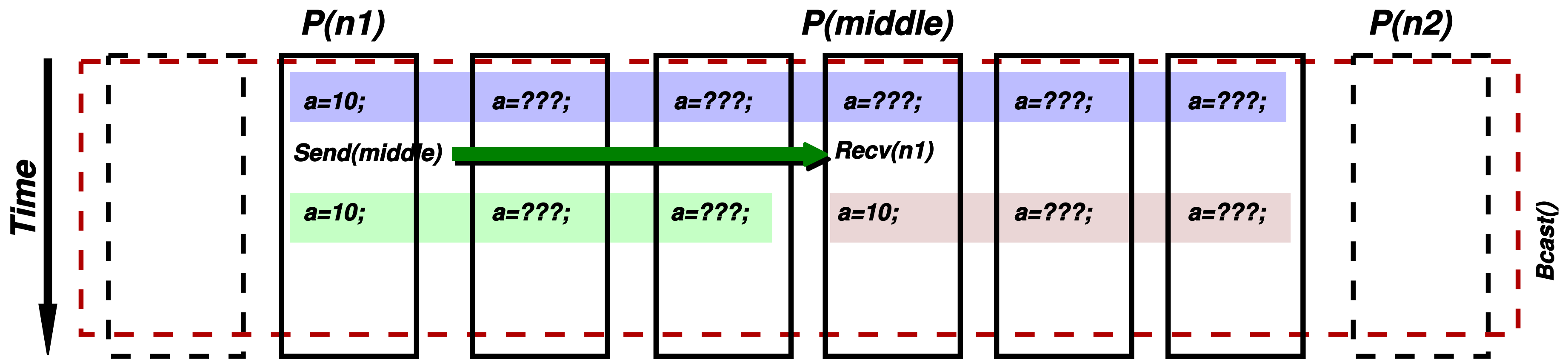
Message broadcast (cont.)



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 such that n1 <= 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.)



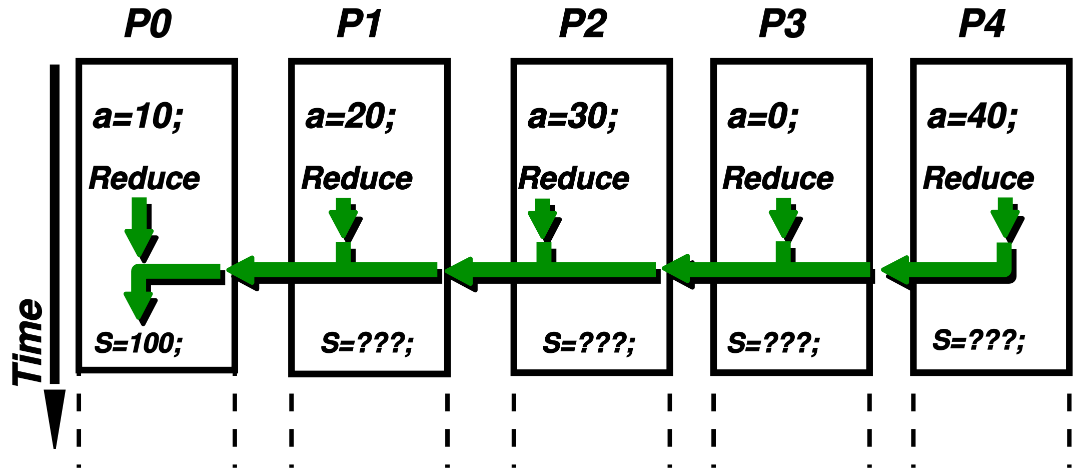
Pseudocode:

```
int n1=0, n2=numprocs;
while (1) {
  int middle = (n1+n2)/2;
  if (myrank==n1) MPI_Send(buff,...,middle,...);
  else if (myrank==middle) MPI_Recv(buff,...,n1,...);
  if (myrank<middle) n2 = middle;
  else n1=middle;
}
```

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.

Global reduction



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

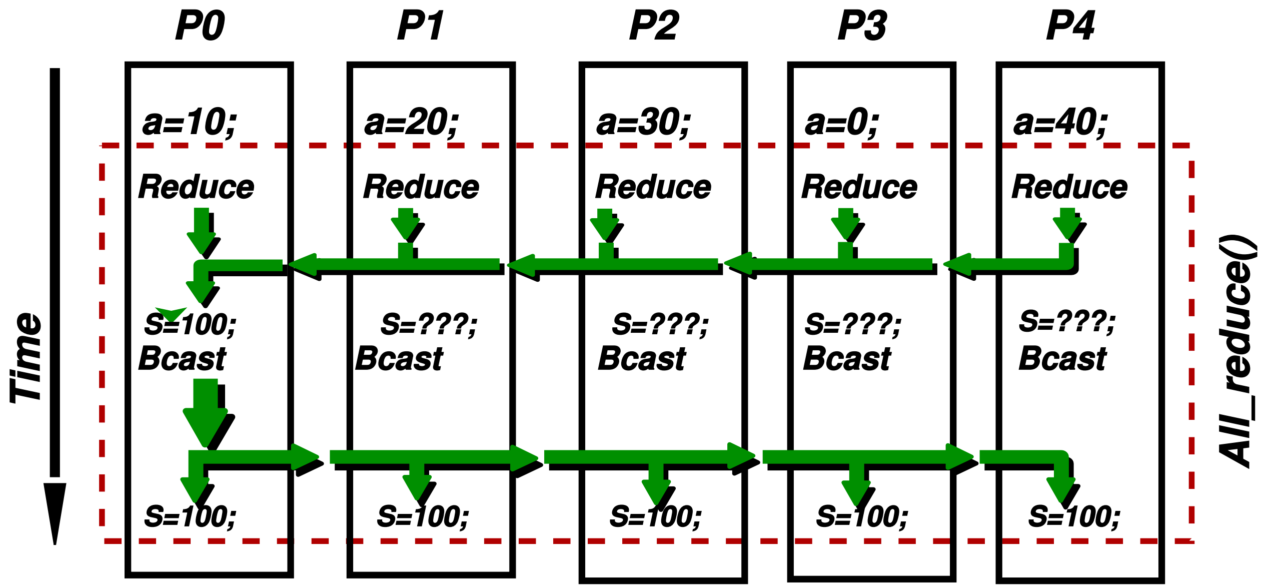
MPI associative global operations

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 appropiate `-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:

```
[mstorti@node1]$ cat ./machi.dat
node2
node3
node4
[mstorti@node1]$ mpirun -np 4 \
               -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:**

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

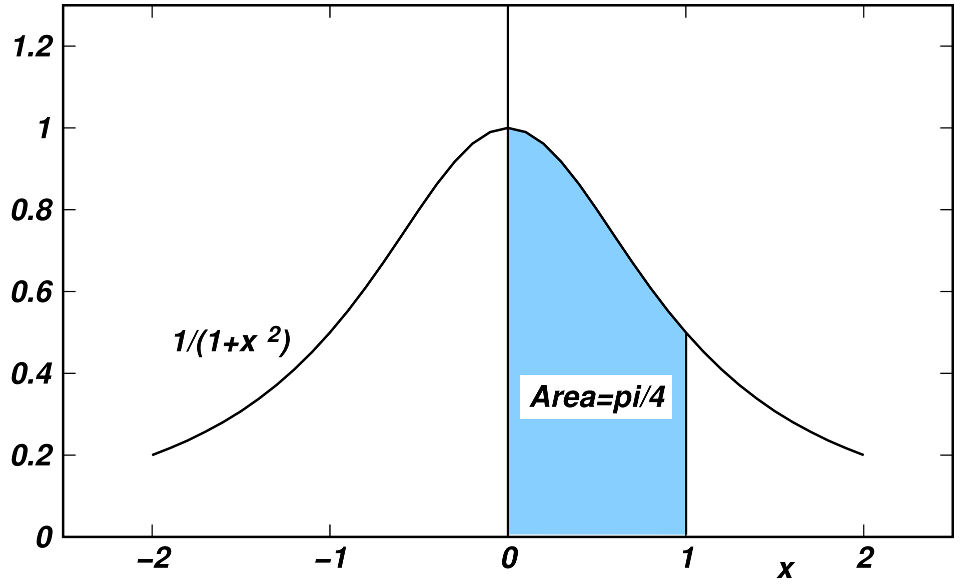
Launches *foo* in *node2*, *node3* and *node4*.

OPTIONAL Assignement 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

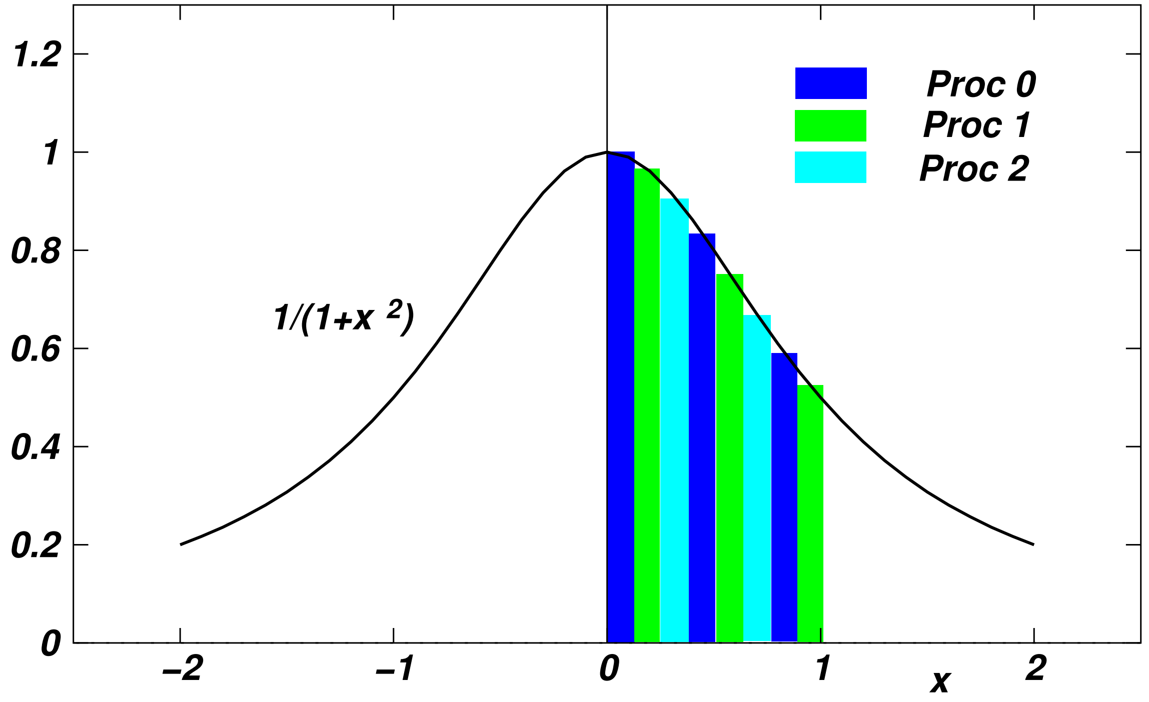


$$\mathrm{atan}(1) = \pi/4$$

$$\frac{\mathrm{d}\,\mathrm{atan}(x)}{\mathrm{d}x} = \frac{1}{1+x^2}$$

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

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.)

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

```
// $Id: pi3.cpp,v 1.1 2004/07/22 17:44:50 mstorti Exp $

//***********************************************************
//   pi3.cpp - compute pi by integrating f(x) = 4/(1 + x**2)
//
//   Each node:
//    1) receives the number of rectangles used
//              in the approximation.
//    2) calculates the areas of it's rectangles.
//    3) Synchronizes for a global summation.
//   Node 0 prints the result.
//
//  Variables:
//
//    pi      the calculated result
//    n       number of points of integration.
//    x       midpoint of each rectangle's interval
//    f       function to integrate
//    sum,pi  area of rectangles
//    tmp     temporary scratch space for global summation
//    i       do loop index
//**********************************************************

#include <mpi.h>
#include <cstdio>
#include <cmath>

// The function to integrate
double f(double x) { return 4./(1.+x*x); }

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

  // Initialize MPI environment
```

```
MPI_Init(&argc,&argv);

// Get the process number and assign it to the variable myrank
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

// Determine how many processes the program will run on and
// assign that number to size
int size;
MPI_Comm_size(MPI_COMM_WORLD,&size);

// The exact value
double PI=4*atan(1.0);

// Enter an infinite loop. Will exit when user enters n=0
while (1) {
  int n;
  // Test to see if this is the program running on process 0,
  // and run this section of the code for input.
  if (!myrank) {
    printf("Enter the number of intervals: (0 quits) > ");
    scanf("%d",&n);
  }

  // The argument 0 in the 4th place indicates that
  // process 0 will send then single integer n to every
  // other process in processor group MPI_COMM_WORLD.
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

  // If the user puts in a negative number for n we leave
  // the program by branching to MPI_FINALIZE
  if (n<0) break;

```

```
// Now this part of the code is running on every node
// and each one shares the same value of n. But all
// other variables are local to each individual
// process. So each process then calculates the each
// interval size.

//*****************************************************c
//       Main Body : Runs on all processors
//*****************************************************c
// even step size h as a function of partions
double h = 1.0/double(n);
double sum = 0.0;
for (int i=myrank+1; i<=n; i += size) {
  double x = h * (double(i) - 0.5);
  sum = sum + f(x);
}
double pi, mypi = h * sum; // this is the total area
                          // in this process,
                          // (a partial sum.)

// Each individual sum should converge also to PI,
// compute the max error
double error, my_error = fabs(size*mypi-PI);
MPI_Reduce(&my_error,&error,1,MPI_DOUBLE,
         MPI_MAX,0,MPI_COMM_WORLD);

// After each partition of the integral is calculated
// we collect all the partial sums. The MPI_SUM
// argument is the operation that adds all the values
// of mypi into pi of process 0 indicated by the 6th
// argument.
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

```

    //*****************************************************
    //           Print results from Process 0
    //*****************************************************

    // Finally the program tests if myrank is node 0
    // so process 0 can print the answer.
    if (!myrank) printf("pi is aprox: %f, "
                       "(error %f,max err over procs %f)\n",
                       pi,fabs(pi-PI),my_error);
    // Run the program again.
    //
  }
  // Branch for the end of program. MPI_FINALIZE will close
  // all the processes in the active group.

  MPI_Finalize();
}
```

Numerical integration (cont.)

ps **report:**

```
[mstorti@spider example]$ ps axfw
 PID COMMAND
 701 xterm -e bash
 707  \_ bash
 907      \_ emacs
 908      |   \_ /usr/libexec/emacs/21.2/i686-pc-linux-gnu/ema...
 985      |   \_ /bin/bash
1732      |   |   \_ ps axfw
1037      |   \_ /bin/bash -i
1058      |       \_ xpdf slides.pdf
1059      |       \_ xfig -library_dir /home/mstorti/CONFIG/xf...
1641      |       \_ /bin/sh /usr/local/mpi/bin/mpirun -np 2 pi3.bin
1718      |           \_ /.../pi3.bin -p4pg /home/mstorti/
1719      |               \_ /.../pi3.bin -p4pg /home/msto....
1720      |               \_ /usr/bin/rsh localhost.localdomain ...
1537      \_ top
[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 $T_i = W_i/s$ shere 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

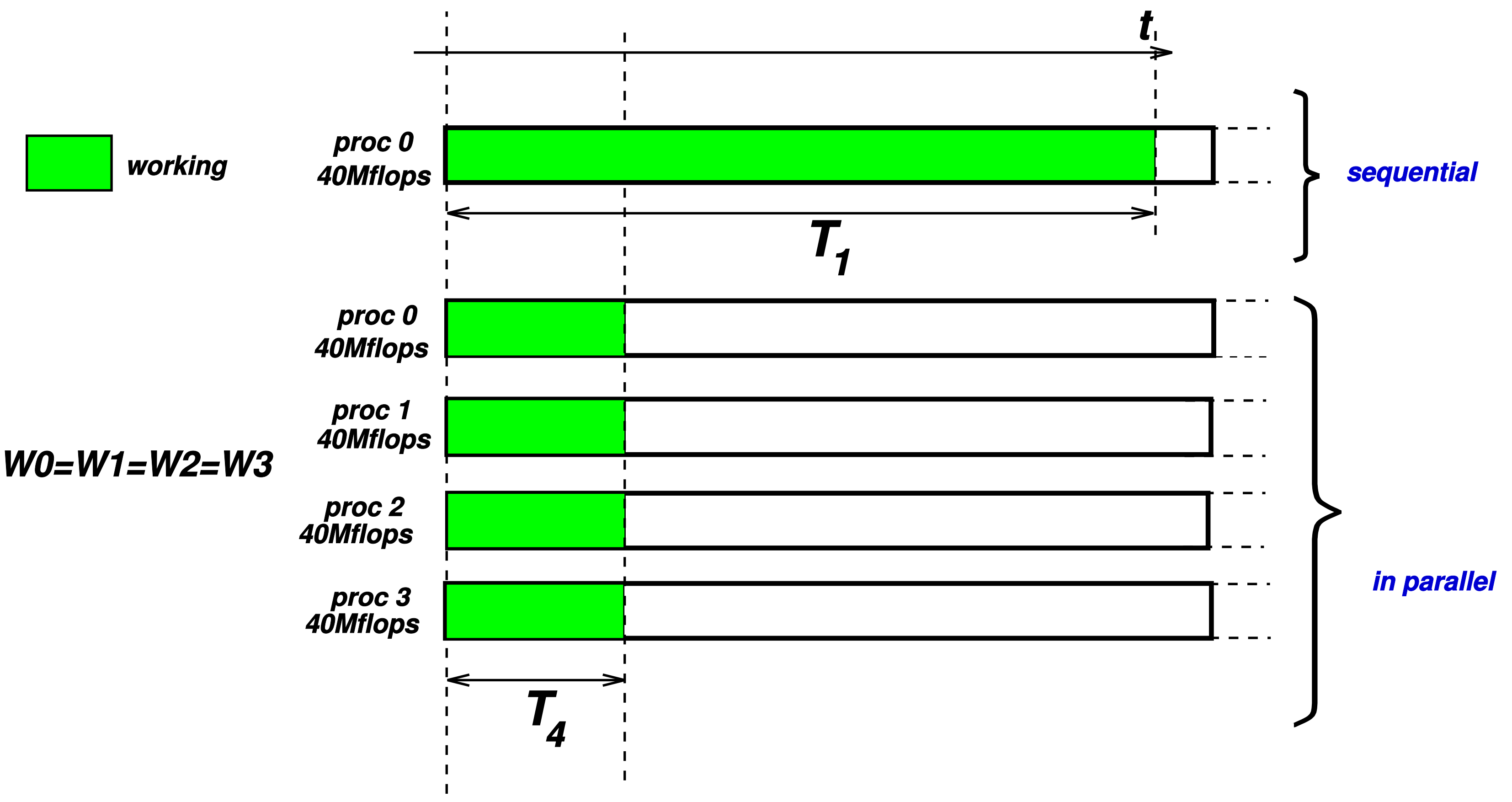
$$T_1 = \frac{W}{s}$$

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.)

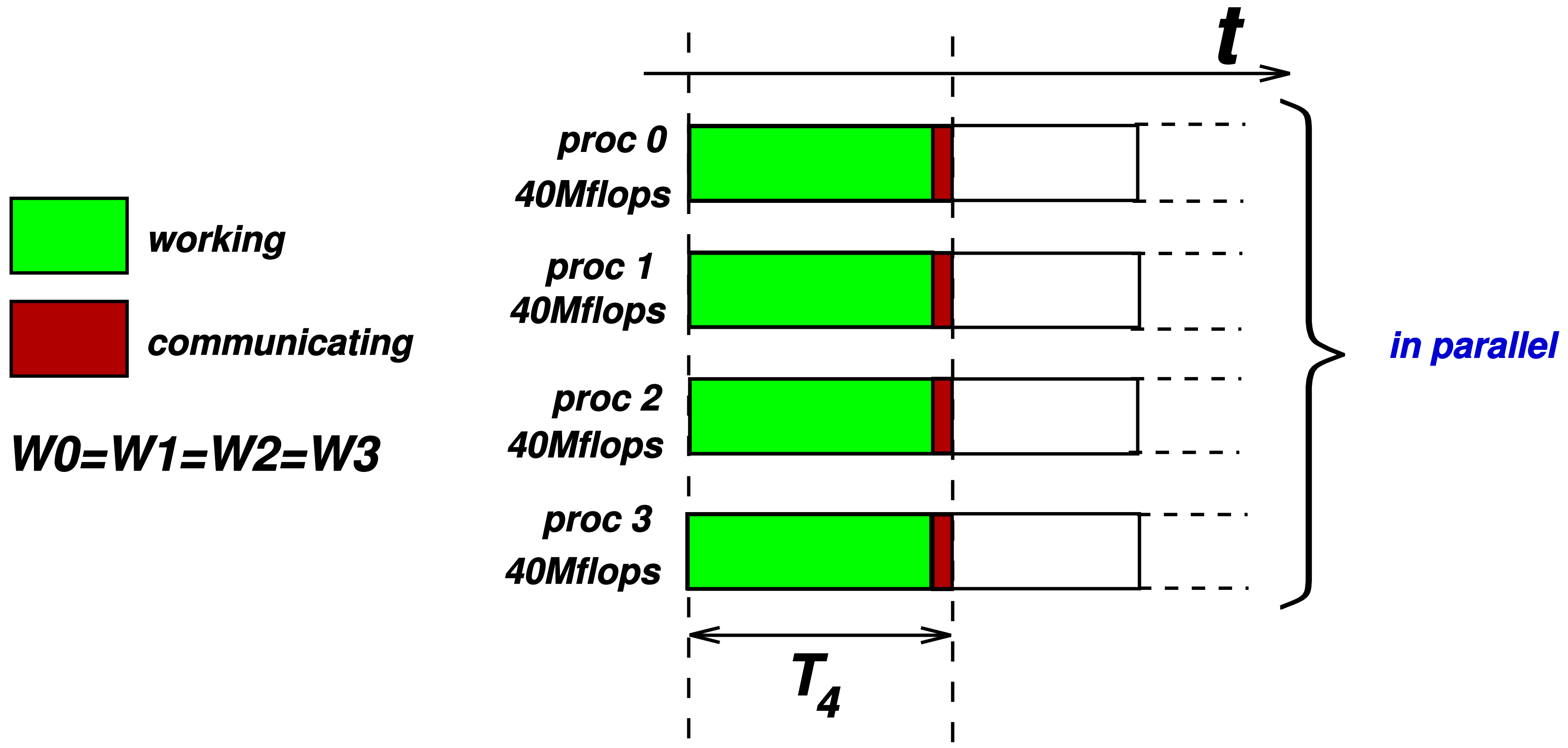
If the communication time *is not* negligible

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

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

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

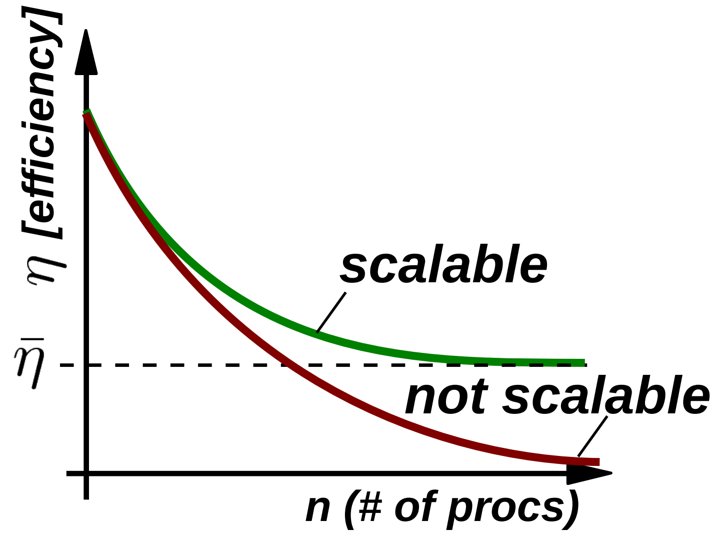
Basic scalability concepts (cont.)



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.

$$\eta \geq \bar\eta > 0, \quad \textbf{para} \quad n \to \infty$$



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.

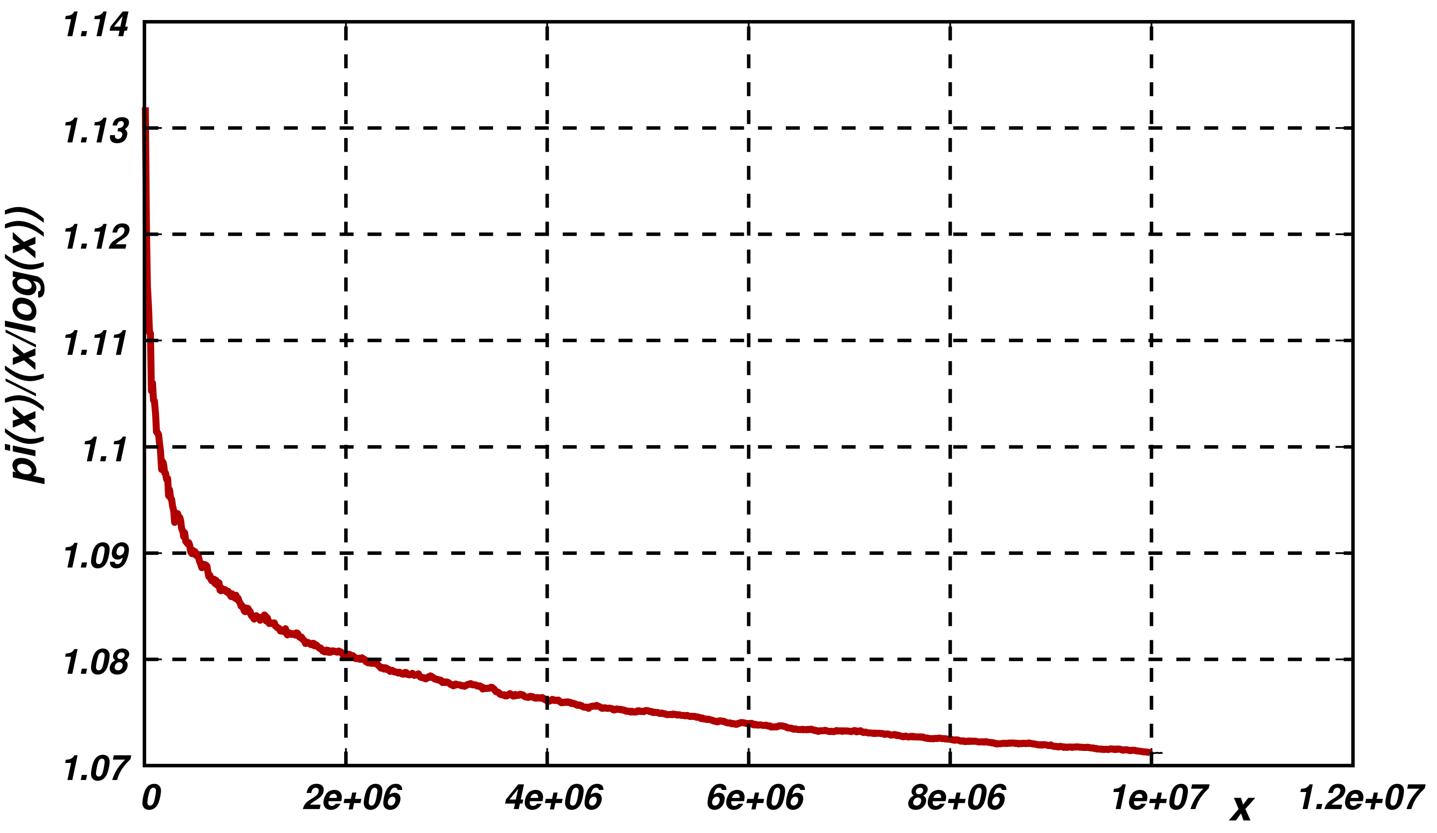
We say that such implementation is scalable in the *thermodynamic limit*, i.e. if we can keep efficiency bounded from below for $n \to \infty$ and $W \sim n \to \infty$. Basically this means that we can *solve larger problems* in a larger number of nodes *in the same time.*

Example: Prime Number Theorem

Prime number theorem

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

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



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 $\mathrm{floor}(\sqrt{n})$. If we can't find a *divisor* then the number is prime

```
int is_prime(int n) {
  if (n<2) return 0;
  int m = int(sqrt(n));
  for (int j=2; j<=m; j++)
    if (n % j==0)) return 0;
  return 1;
}
```

So that `is_prime(n)` is (in the worst case) $O(\sqrt{n})$. As the number of digits of n is $n_d = \mathrm{ceil}(\log_{10} n)$, 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

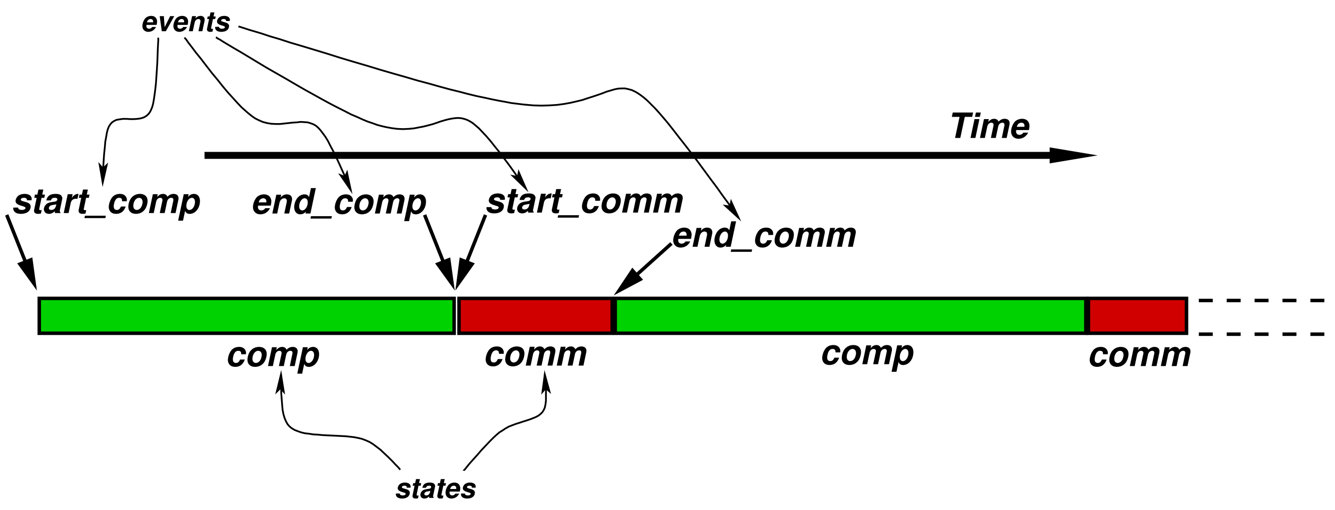
```
//$Id: primes1.cpp,v 1.2 2005/04/29 02:35:28 mstorti Exp $
#include <cstdio>
#include <cmath>

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

// Sequential version

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

  int n=2, primes=0,chunk=10000;
  while(1) {
    if (is_prime(n++)) primes++;
    if (!(n % chunk)) printf("%d primes<%d\n",primes,n);
  }
}
```

PNT: Parallel version

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

```
//$Id: primes2.cpp,v 1.1 2004/07/23 01:33:27 mstorti Exp $
#include <mpi.h>
#include <cstdio>
#include <cmath>

int is_prime(int n) {
  int m = int(sqrt(n));
  for (int j=2; j<=m; j++)
    if (!(n % j)) return 0;
  return 1;
}

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

  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int n2, primesh=0, primes, chunk=100000,
    n1 = myrank*chunk;
  while(1) {
    n2 = n1 + chunk;
    for (int n=n1; n<n2; n++) {
      if (is_prime(n)) primesh++;
    }
    MPI_Reduce(&primesh,&primes,1,MPI_INT,
               MPI_SUM,0,MPI_COMM_WORLD);
    n1 += size*chunk;
    if (!myrank) printf("pi(%d) = %d\n",n1,primes);
  }

  MPI_Finalize();
}
```

MPE Logging



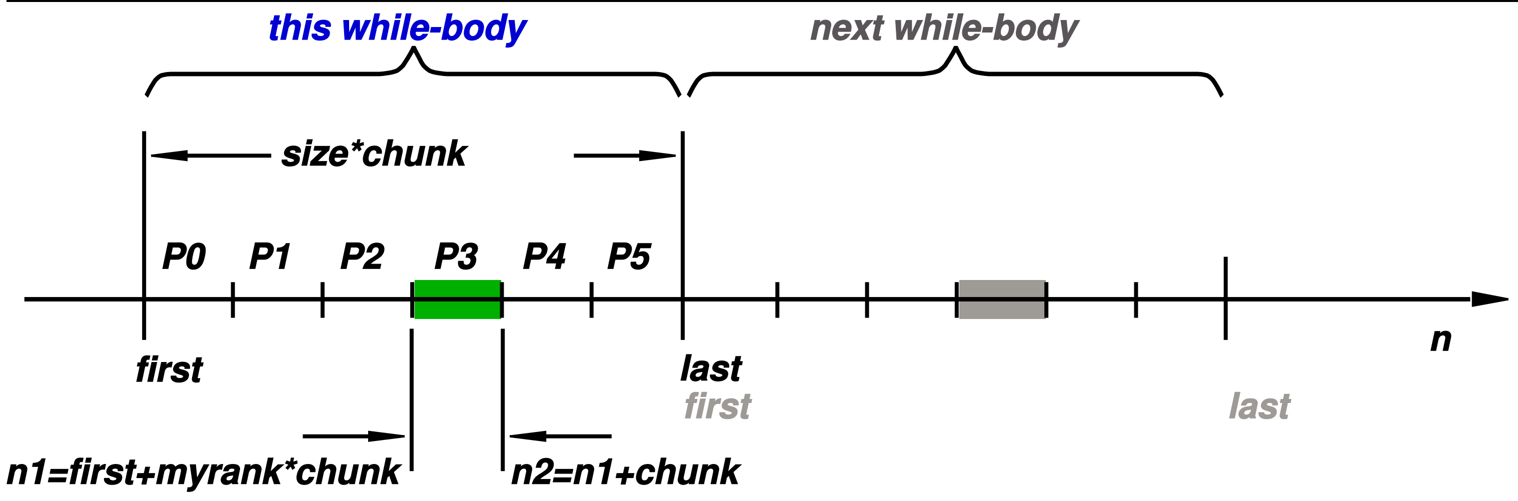
- User can define ***states***. Typically we want to know how much time the program spends in ***computation*** (`comp`) and how much in ***communication*** (`comm`).
- States are delimited by ***events*** of very short duration (we say ***atomic***). Typically we define two events for each state.
  - ▷ state `comp` = $\{t \;/\; \texttt{start\_comp} < t < \texttt{end\_comp}\}$
  - ▷ state `comm` = $\{t \;/\; \texttt{start\_comm} < t < \texttt{end\_comm}\}$

MPE Logging (cont.)

```
#include <mpe.h>
#include . . .

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  MPE_Init_log();
  int start_comp = MPE_Log_get_event_number();
  int end_comp = MPE_Log_get_event_number();
  int start_comm = MPE_Log_get_event_number();
  int end_comm = MPE_Log_get_event_number();

  MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
  MPE_Describe_state(start_comm,end_comm,"comm","red:white");

  while(. . .) {
    MPE_Log_event(start_comp,0,"start-comp");
    // compute. . .
    MPE_Log_event(end_comp,0,"end-comp");

    MPE_Log_event(start_comm,0,"start-comm");
    // communicate. . .
    MPE_Log_event(end_comm,0,"end-comm");
  }

  MPE_Finish_log("primes");
  MPI_Finalize();
}
```

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

PNT: Parallel dynamic version with MPE logging



```
// Counts primes en [0,N)
first = 0;
while (1) {
  // Each processor checks a subrange in [first,last)
  last = first + size*chunk;
  n1 = first + myrank*chunk;
  n2 = n1 + chunk;
  if (n2>last) n2=last;
  primesh = /* # of primes in [n1,n2) . . . */;
  // Allreduce ‘primesh’ to ‘primes’ . . .
  first += size*chunk;
  if (last>N) break;
}
// finalize . . .
```

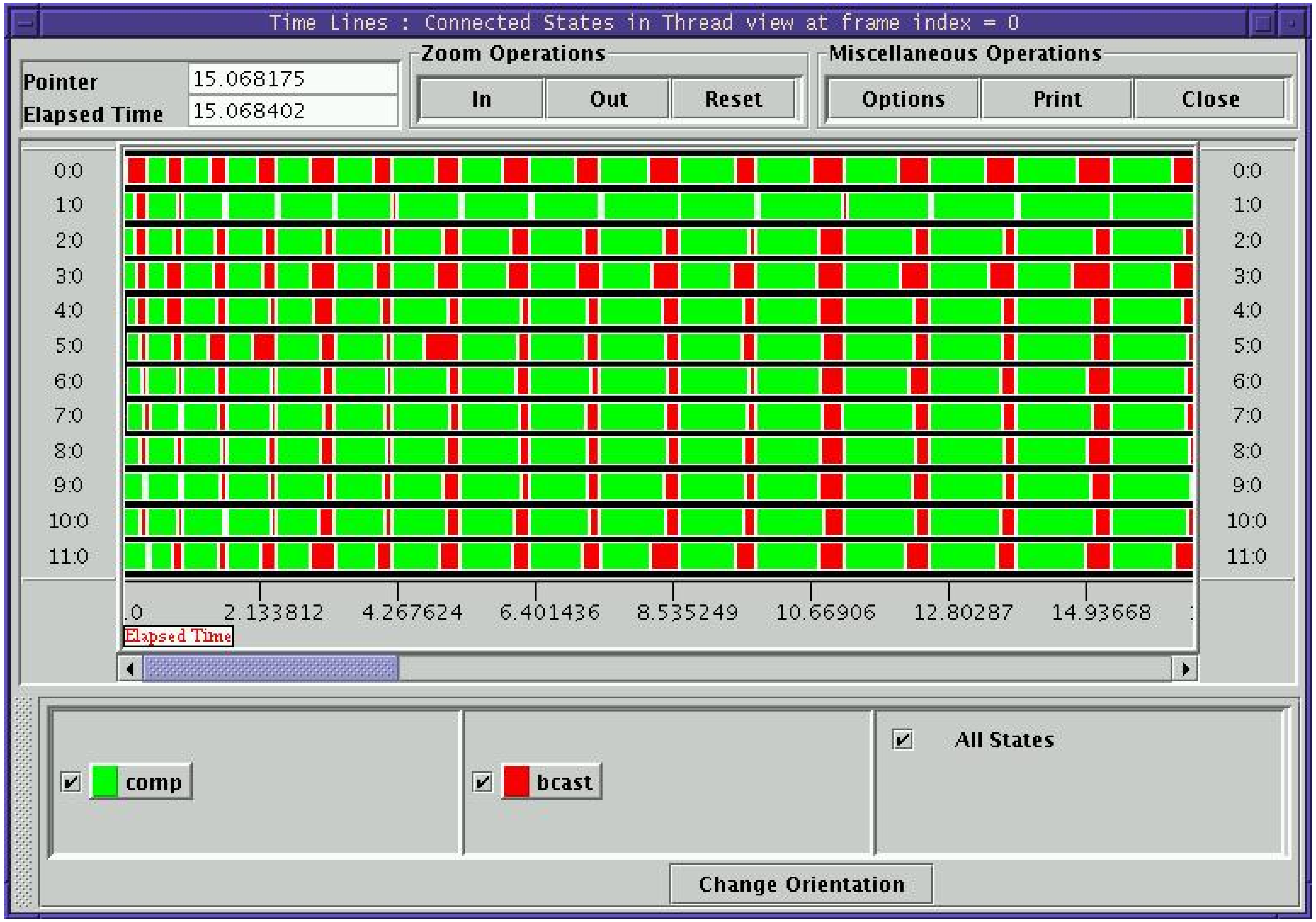
PNT: Parallel dynamic version with MPE logging (cont.)

```
//$Id: primes3.cpp,v 1.4 2004/07/23 22:51:31 mstorti Exp $
#include <mpi.h>
#include <mpe.h>
#include <cstdio>
#include <cmath>

int is_prime(int n) {
  if (n<2) return 0;
  int m = int(sqrt(n));
  for (int j=2; j<=m; j++)
    if (!(n % j)) return 0;
  return 1;
}

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  MPE_Init_log();

  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  int start_comp = MPE_Log_get_event_number();
  int end_comp = MPE_Log_get_event_number();
  int start_bcast = MPE_Log_get_event_number();
  int end_bcast = MPE_Log_get_event_number();

  int n2, primesh=0, primes, chunk=200000,
    n1, first=0, last;

  if (!myrank) {
    MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
    MPE_Describe_state(start_bcast,end_bcast,"bcast","red:white");
  }

  while(1) {
    MPE_Log_event(start_comp,0,"start-comp");
    last = first + size*chunk;
    n1 = first + myrank*chunk;
    n2 = n1 + chunk;
    for (int n=n1; n<n2; n++) {
      if (is_prime(n)) primesh++;
    }
    MPE_Log_event(end_comp,0,"end-comp");
    MPE_Log_event(start_bcast,0,"start-bcast");
    MPI_Allreduce(&primesh,&primes,1,MPI_INT,
                  MPI_SUM,MPI_COMM_WORLD);
    first += size*chunk;
    if (!myrank) printf("pi(%d) = %d\n",last,primes);
    MPE_Log_event(end_bcast,0,"end-bcast");
    if (last>=10000000) break;
  }

  MPE_Finish_log("primes");
  MPI_Finalize();
}
```

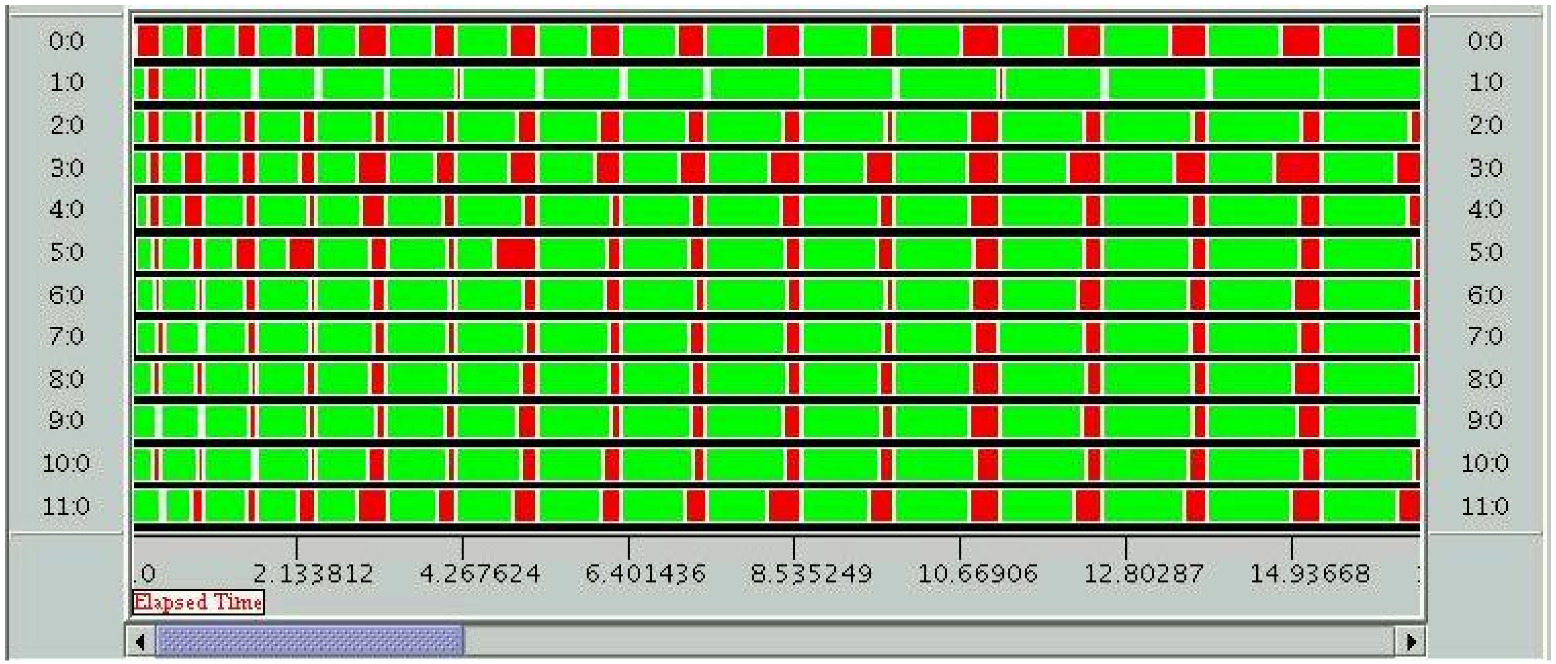
Using Jumpshot

- *"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, nay 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.)

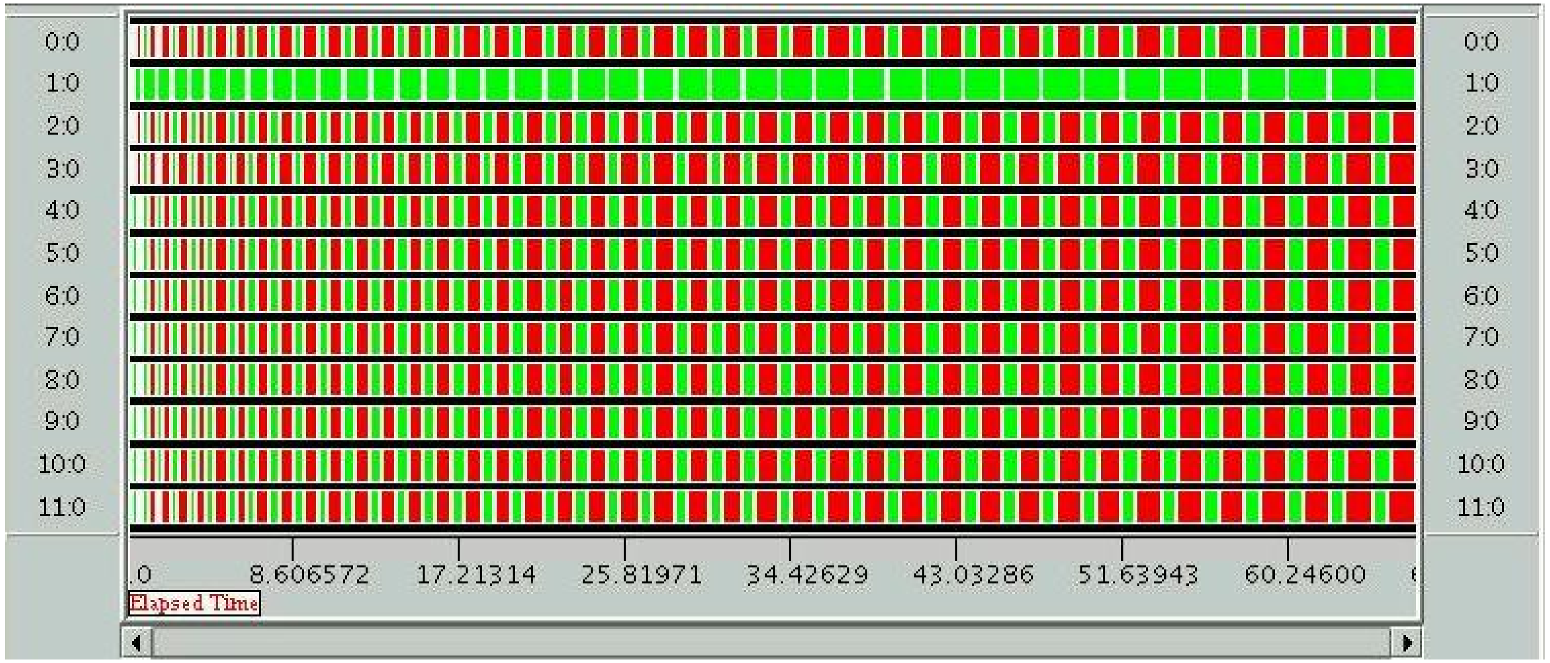


PNT: $\mathrm{np} = 12$ (blocking)



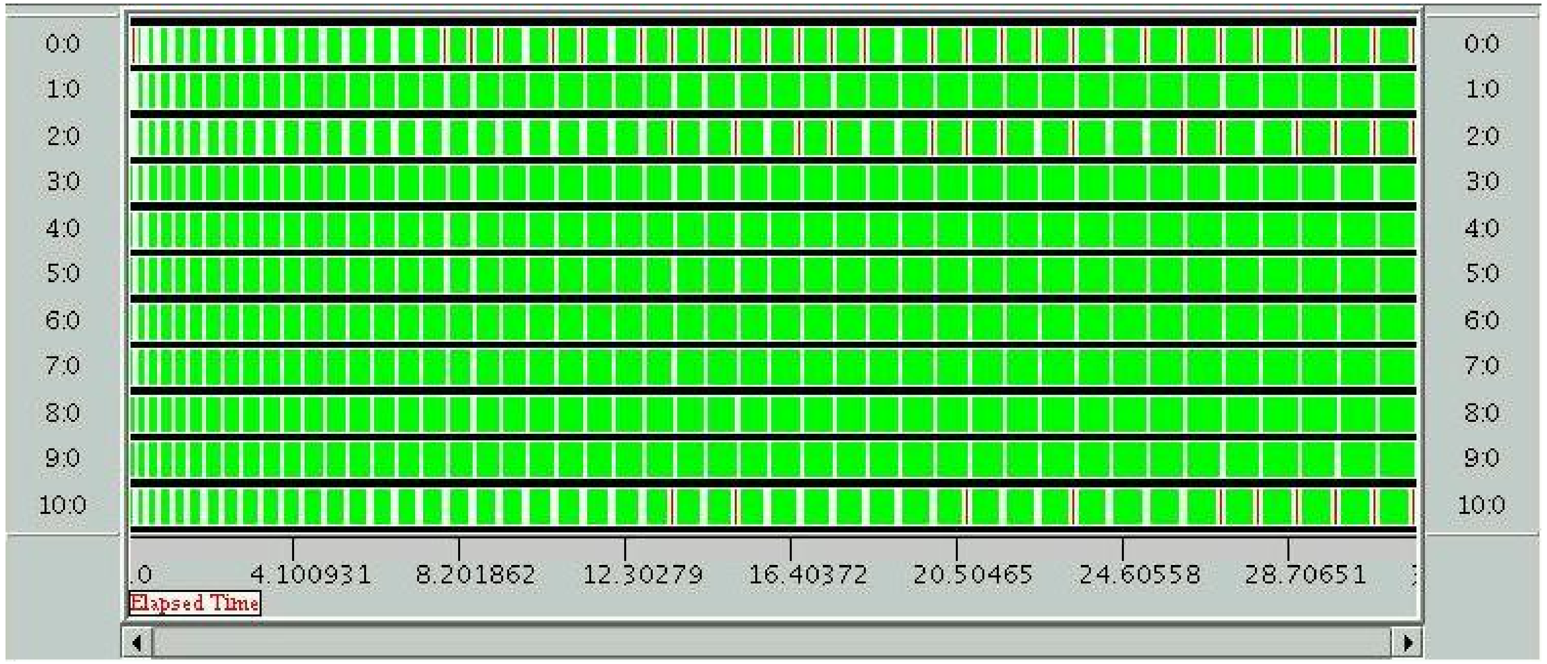
- Nodes are running other programs (simulates variable node speed)
- `comm` is in fact *communication+synchronization*.
- `MPI_Allreduce()` acts like a synchronization barrier (all processors wait until *all* of them reach the collective call).
- Processor 1 is very slow and disbalances the parallel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

PNT: $\mathrm{np} = 12$ (blocking) (cont.)



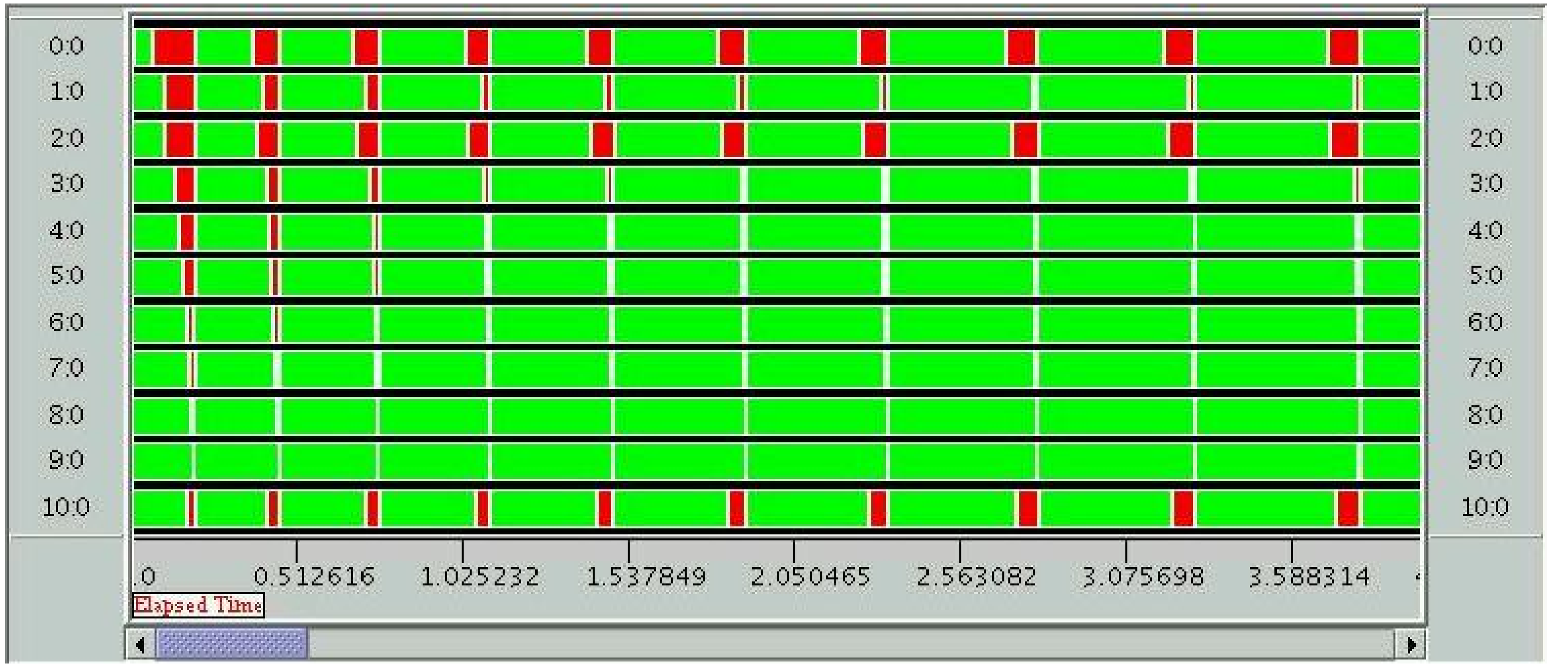
- **Proc 1 is running other programs (simulates a slow processor). Other processors are dedicated.**
- **Disbalance is even worst.**

PNT: $\mathrm{np} = 12$ (blocking) (cont.)



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

PNT: $\mathrm{np} = 12$ (blocking) (cont.)



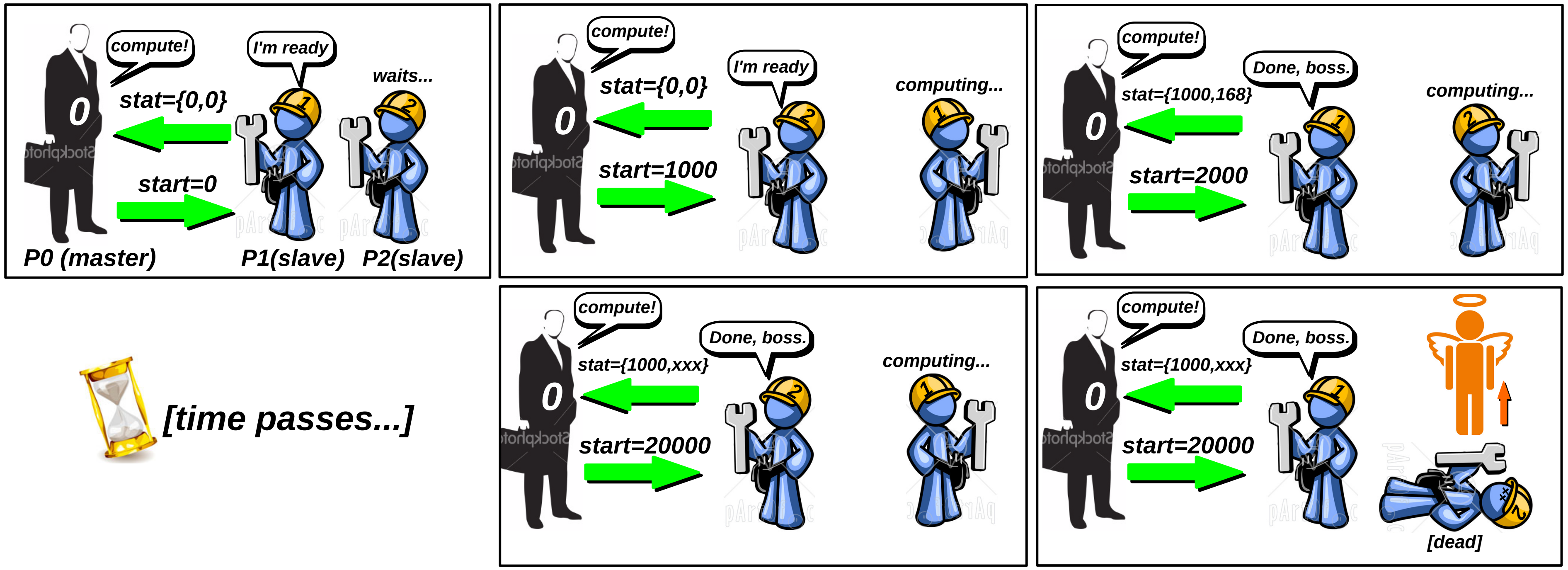
- **Without node12 (proc1 in the previous example). Note that good balance is achieved. (detail)**

PNT: dynamic parallel version.

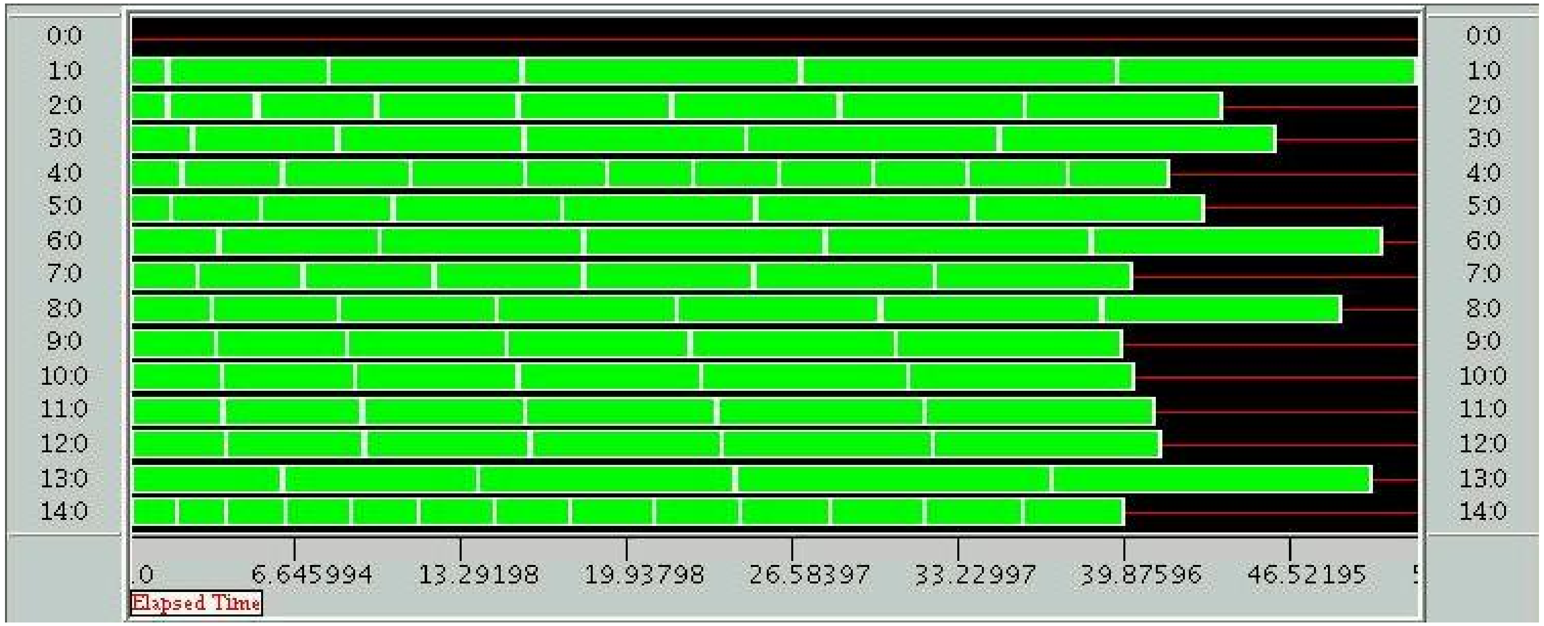
- Proc 0 acts like a server.
- Slaves send to the master the work already done and the server returns new work to be done.
- Work done is an integer array `stat[2]`: number of integers checked (`stat[0]`) and number of primes found (`stat[1]`).
- Work to be done is a single integer `start`. The slave 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

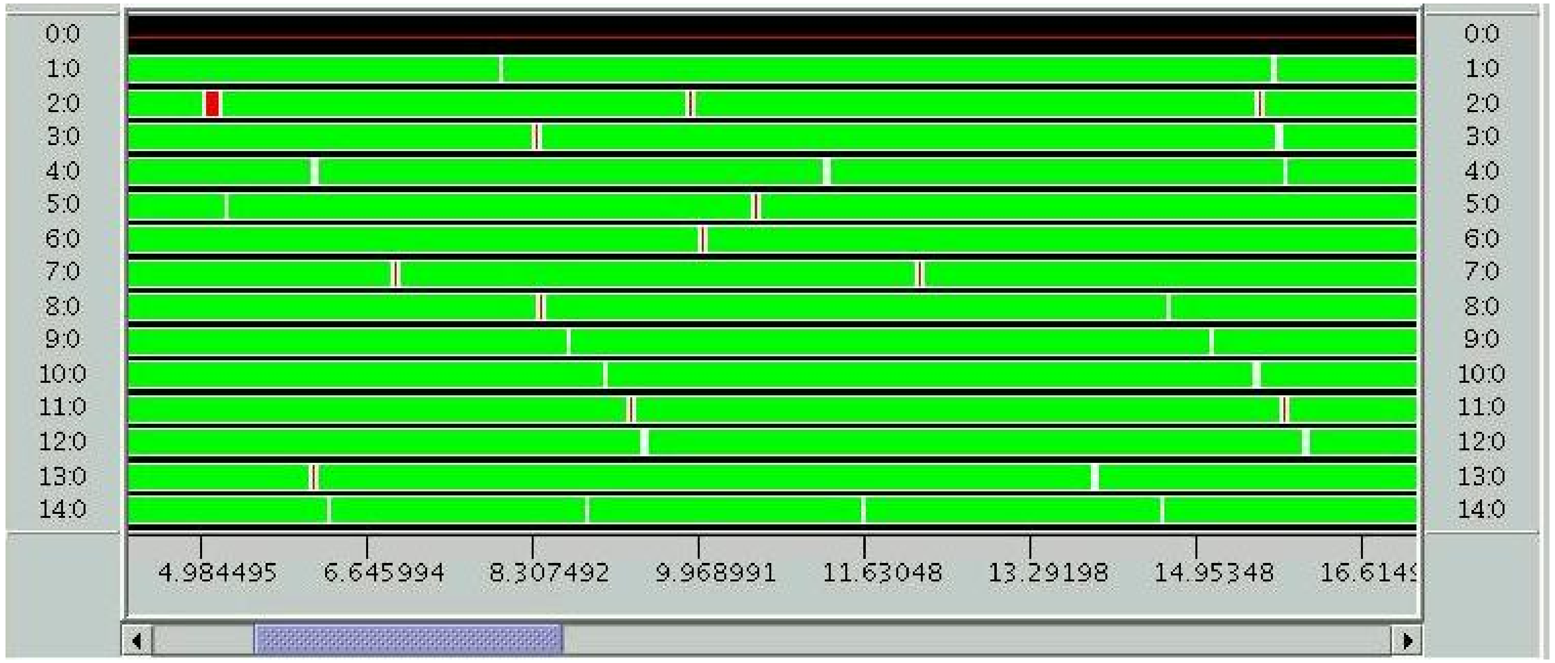


PNT: dynamic parallel version. (cont.)



- **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

```
if (!myrank) {
  int start=0, checked=0, down=0;
  while(1) {
    Recv(&stat,...,MPI_ANY_SOURCE,...,&status);
    checked += stat[0];
    primes += stat[1];
    MPI_Send(&start,...,status.MPI_SOURCE,...);
    if (start<N) start += chunk;
    else down++;
    if (down==size-1) break;
  }
} else {
  stat[0]=0; stat[1]=0;
  MPI_Send(stat,...,0,...);
  while(1) {
    int start;
    MPI_Recv(&start,...,0,...);
    if (start>=N) break;
    int last = start + chunk;
    if (last>N) last=N;
    stat[0] = last-start ;
    stat[1] = 0;
    for (int n=start; n<last; n++)
      if (is_prime(n)) stat[1]++;
    MPI_Send(stat,...,0,...);
  }
}
```

PNT: Parallel dynamic version. Code

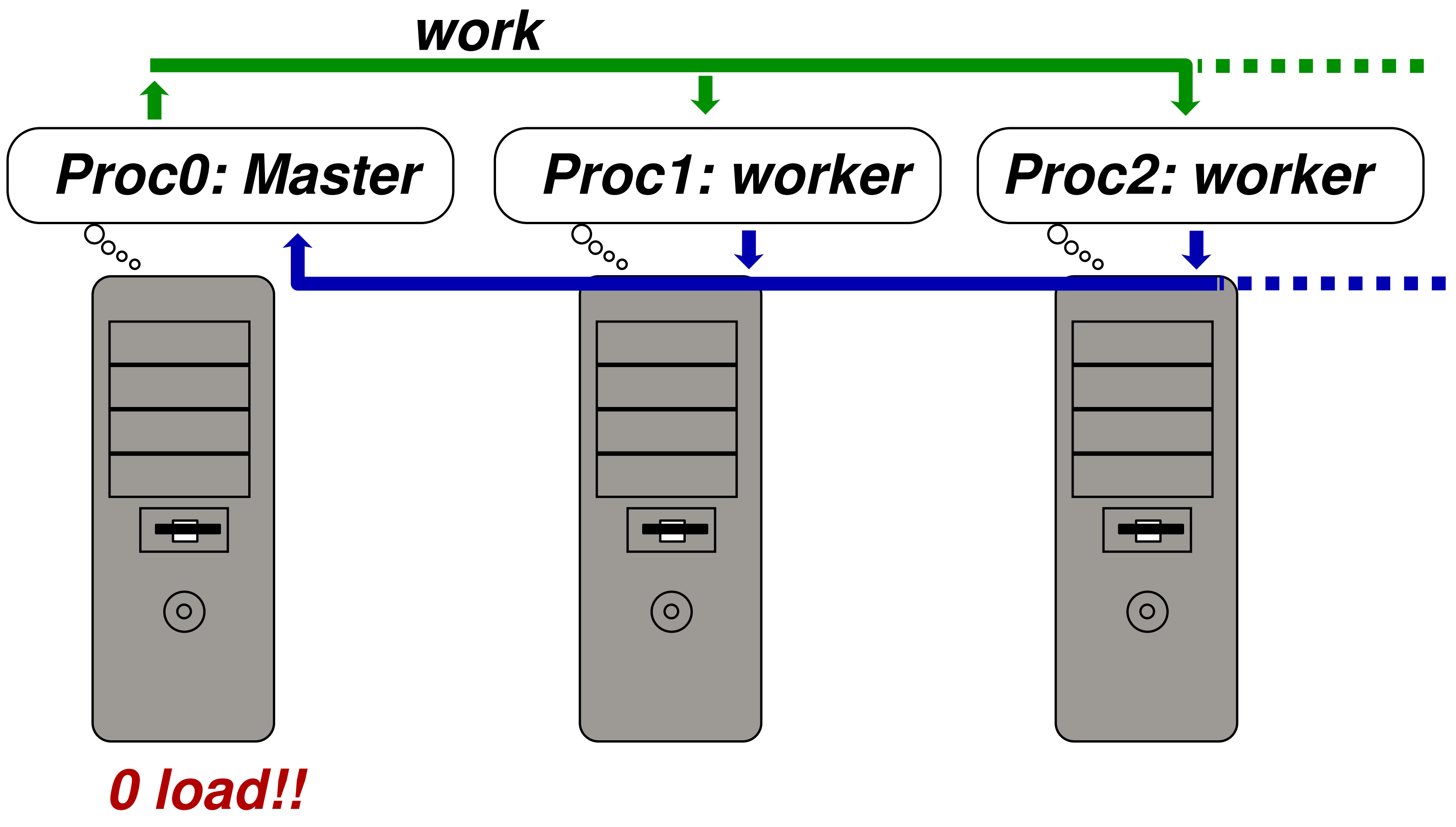
```
//$Id: primes4.cpp,v 1.5 2004/10/03 14:35:43 mstorti Exp $
#include <mpi.h>
#include <mpe.h>
#include <cstdio>
#include <cmath>
#include <cassert>

int is_prime(int n) {
  if (n<2) return 0;
  int m = int(sqrt(n));
  for (int j=2; j<=m; j++)
    if (!(n % j)) return 0;
  return 1;
}

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  MPE_Init_log();

  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  assert(size>1);
  int start_comp = MPE_Log_get_event_number();
  int end_comp = MPE_Log_get_event_number();
  int start_comm = MPE_Log_get_event_number();
  int end_comm = MPE_Log_get_event_number();

  int chunk=200000, N=20000000;
  MPI_Status status;
  int stat[2]; // checked,primes

#define COMPUTE 0
#define STOP 1

  if (!myrank) {
    MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
    MPE_Describe_state(start_comm,end_comm,"comm","red:white");
    int first=0, checked=0, down=0, primes=0;
    while (1) {
      MPI_Recv(&stat,2,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
               MPI_COMM_WORLD,&status);
      int source = status.MPI_SOURCE;
      if (stat[0]) {
        checked += stat[0];
        primes += stat[1];
        printf("recvd %d primes from %d, checked %d, cum primes %d\n",
               stat[1],source,checked,primes);
      }
      printf("sending [%d,%d) to %d\n",first,first+chunk,source);
      MPI_Send(&first,1,MPI_INT,source,0,MPI_COMM_WORLD);
      if (first<N) first += chunk;
      else printf("shutting down %d, so far %d\n",source,++down);
      if (down==size-1) break;
    }
  } else {
    int start;
    stat[0]=0; stat[1]=0;
    MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
    while(1) {
      MPE_Log_event(start_comm,0,"start-comm");
      MPI_Recv(&start,1,MPI_INT,0,MPI_ANY_TAG,
               MPI_COMM_WORLD,&status);
      MPE_Log_event(end_comm,0,"end-comm");
      if (start>=N) break;
      MPE_Log_event(start_comp,0,"start-comp");
      int last = start + chunk;
      if (last>N) last=N;
      stat[0] = last-start ;
      stat[1] = 0;
      if (start<2) start=2;
      for (int n=start; n<last; n++) if (is_prime(n)) stat[1]++;
      MPE_Log_event(end_comp,0,"end-comp");
      MPE_Log_event(start_comm,0,"start-comm");
      MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
      MPE_Log_event(end_comm,0,"end-comm");
    }
  }
  MPE_Finish_log("primes");
  MPI_Finalize();
}
```

PNT: Parallel dynamic version

- This kind of strategy is called *master/slave* or *compute-on-demand*. Slaves *wait for 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 elemental work* to be done by the master, and of the *time-slice* of the operating system.

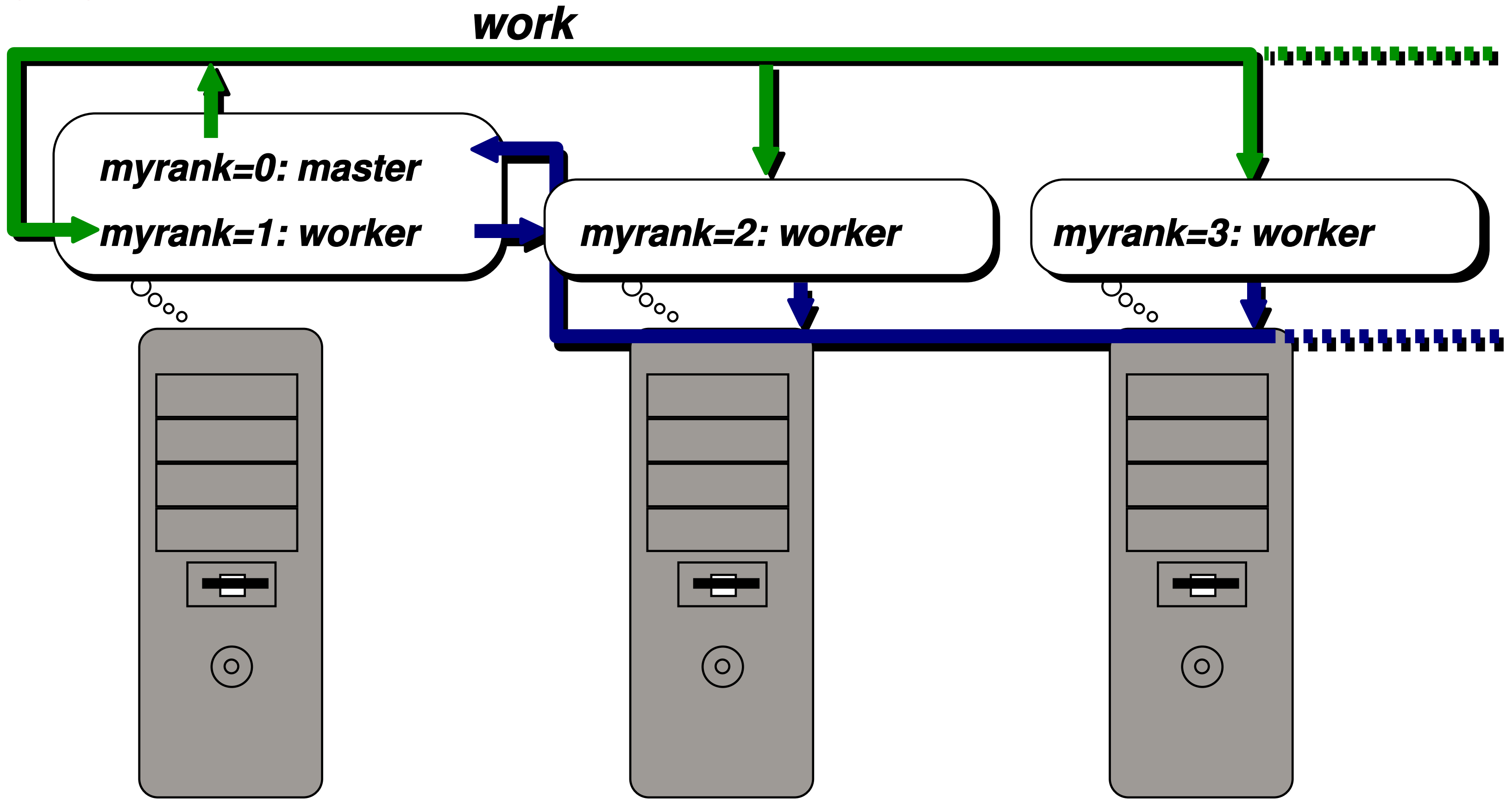
PNT: Parallel dynamic version (cont.)

***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.)

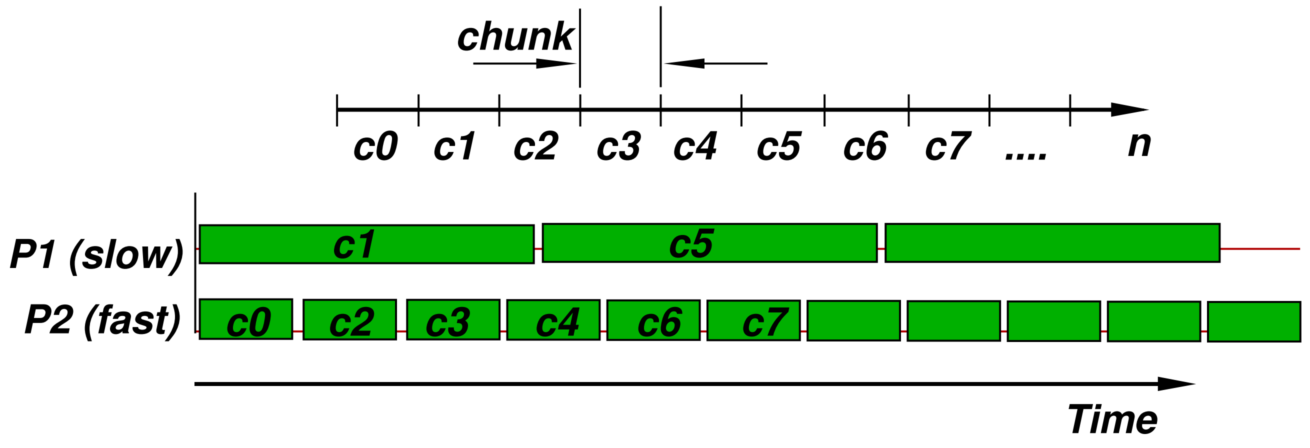
Launch two process in processor 0: `myrank=0` (master) and `myrank=1` (worker). This can cause a *delay* in the answer of the master. ***All processors do work***.



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 ($\mathtt{np} = 1$) in node 10 (fast): 285 sec.**
- **Sequential ($\mathtt{np} = 1$) in node 24 (slow): 338 sec.**
- **With $\mathtt{np} = 12$ (disbalanced, blocking) 69 sec.**
- **Excluding the loaded node (balanced, blocking, $\mathtt{np} = 11$): 33sec.**
- **With ($\mathtt{np} = 14$) (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 $c2$, $c3$, $c4$ are reported *after* loading $c1$. Similarly $c6$ and $c7$ are reported after $c5$...**

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

```
struct chunk_info {
  int checked,primes,start;
};
set<chunk_info> received;
vector<int> proc_start[size];

if (!myrank) {
  int start=0, checked=0, down=0, primes=0;
  while(1) {
    Recv(&stat,. . .,MPI_ANY_SOURCE,. . .,&status);
    int source = status.MPI_SOURCE;
    checked += stat[0];
    primes += stat[1];
    // put(checked,primes,proc_start[source]) en ‘received’ . . .
    // report last pi(n) computed . . .
    MPI_Send(&start,. . .,source,. . .);
    proc_start[source]=start;
    if (start<N) start += chunk;
    else down++;
    if (down==size-1) break;
  }
} else {
  stat[0]=0; stat[1]=0;
  MPI_Send(stat,. . .,0,. . .);
  while(1) {
    int start;
    MPI_Recv(&start,...,0,...);
    if (start>=N) break;
    int last = start + chunk;
    if (last>N) last=N;
    stat[0] = last-start ;
    stat[1] = 0;
    for (int n=start; n<last; n++)
      if (is_prime(n)) stat[1]++;
    MPI_Send(stat,...,0,...);
  }
}
```

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

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

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

```
//$Id: primes5.cpp,v 1.4 2004/07/25 15:21:26 mstorti Exp $
#include <mpi.h>
#include <mpe.h>
#include <cstdio>
#include <cmath>
#include <cassert>
#include <vector>
#include <set>
#include <unistd.h>
#include <ctype.h>

using namespace std;

int is_prime(int n) {
  if (n<2) return 0;
  int m = int(sqrt(double(n)));
  for (int j=2; j<=m; j++)
    if (!(n % j)) return 0;
  return 1;
}

struct chunk_info {
  int checked,primes,start;
  bool operator<(const chunk_info& c) const {
    return start<c.start;
  }
};

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

```
MPI_Init(&argc,&argv);
MPE_Init_log();

int myrank, size;
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

assert(size>1);
int start_comp = MPE_Log_get_event_number();
int end_comp = MPE_Log_get_event_number();
int start_comm = MPE_Log_get_event_number();
int end_comm = MPE_Log_get_event_number();

int chunk = 20000, N = 200000;
char *cvalue = NULL;
int index;
int c;
opterr = 0;

while ((c = getopt (argc, argv, "N:c:")) != -1)
  switch (c) {
  case 'c':
    sscanf(optarg,"%d",&chunk); break;
  case 'N':
    sscanf(optarg,"%d",&N); break;
  case '?':
    if (isprint (optopt))
      fprintf (stderr, "Unknown option `-%c'.\n", optopt);
    else
      fprintf (stderr,
               "Unknown option character `\\x%x'.\n",
               optopt);
      return 1;
    default:
      abort ();
    };

  if (!myrank)
    printf ("chunk %d, N%d\n",chunk,N);

  MPI_Status status;
  int stat[2]; // checked,primes
  set<chunk_info> received;
  vector<int> start_sent(size,-1);

#define COMPUTE 0
#define STOP 1

  if (!myrank) {
    MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
    MPE_Describe_state(start_comm,end_comm,"comm","red:white");
    int first=0, checked=0,
      down=0, primes=0, first_recv = 0;
    while (1) {
      MPI_Recv(&stat,2,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
               MPI_COMM_WORLD,&status);
      int source = status.MPI_SOURCE;
      if (stat[0]) {
        assert(start_sent[source]>=0);
        chunk_info cs;
        cs.checked = stat[0];
        cs.primes = stat[1];
        cs.start = start_sent[source];
```

```
          received.insert(cs);
          printf("recvd %d primes from %d\n",
                 stat[1],source,checked,primes);
        }
        while (!received.empty()) {
          set<chunk_info>::iterator q = received.begin();
          if (q->start != first_recv) break;
          primes += q->primes;
          received.erase(q);
          first_recv += chunk;
        }
        printf("pi(%d) = %d, (queued %d)\n",
               first_recv+chunk,primes,received.size());
        MPI_Send(&first,1,MPI_INT,source,0,MPI_COMM_WORLD);
        start_sent[source] = first;
        if (first<N) first += chunk;
        else {
          down++;
          printf("shutting down %d, so far %d\n",source,down);
        }
        if (down==size-1) break;
      }
      set<chunk_info>::iterator q = received.begin();
      while (q!=received.end()) {
        primes += q->primes;
        printf("pi(%d) = %d\n",q->start+chunk,primes);
        q++;
      }
    } else {
      int start;
      stat[0]=0; stat[1]=0;
      MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
      while(1) {
        MPE_Log_event(start_comm,0,"start-comm");
        MPI_Recv(&start,1,MPI_INT,0,MPI_ANY_TAG,
                 MPI_COMM_WORLD,&status);
        MPE_Log_event(end_comm,0,"end-comm");
        if (start>=N) break;
        MPE_Log_event(start_comp,0,"start-comp");
        int last = start + chunk;
        if (last>N) last=N;
        stat[0] = last-start ;
        stat[1] = 0;
        if (start<2) start=2;
        for (int n=start; n<last; n++) if (is_prime(n)) stat[1]++;
        MPE_Log_event(end_comp,0,"end-comp");
        MPE_Log_event(start_comm,0,"start-comm");
        MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
        MPE_Log_event(end_comm,0,"end-comm");
      }
    }
    MPE_Finish_log("primes");
    MPI_Finalize();
  }
```

Reading data and options

- *Read options from a common file via NFS*

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

(OK for a small data volume)

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

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

(OK for options. Needs Bcast).

Reading data and options (cont.)

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

```
#include <unistd.h>
#include <ctype.h>

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  // ...
  int c;
  opterr = 0;
  while ((c = getopt (argc, argv, "N:c:")) != -1)
    switch (c) {
    case 'c':
      sscanf(optarg,"%d",&chunk); break;
    case 'N':
      sscanf(optarg,"%d",&N); break;
    };
  //...
}
```

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

Scalability

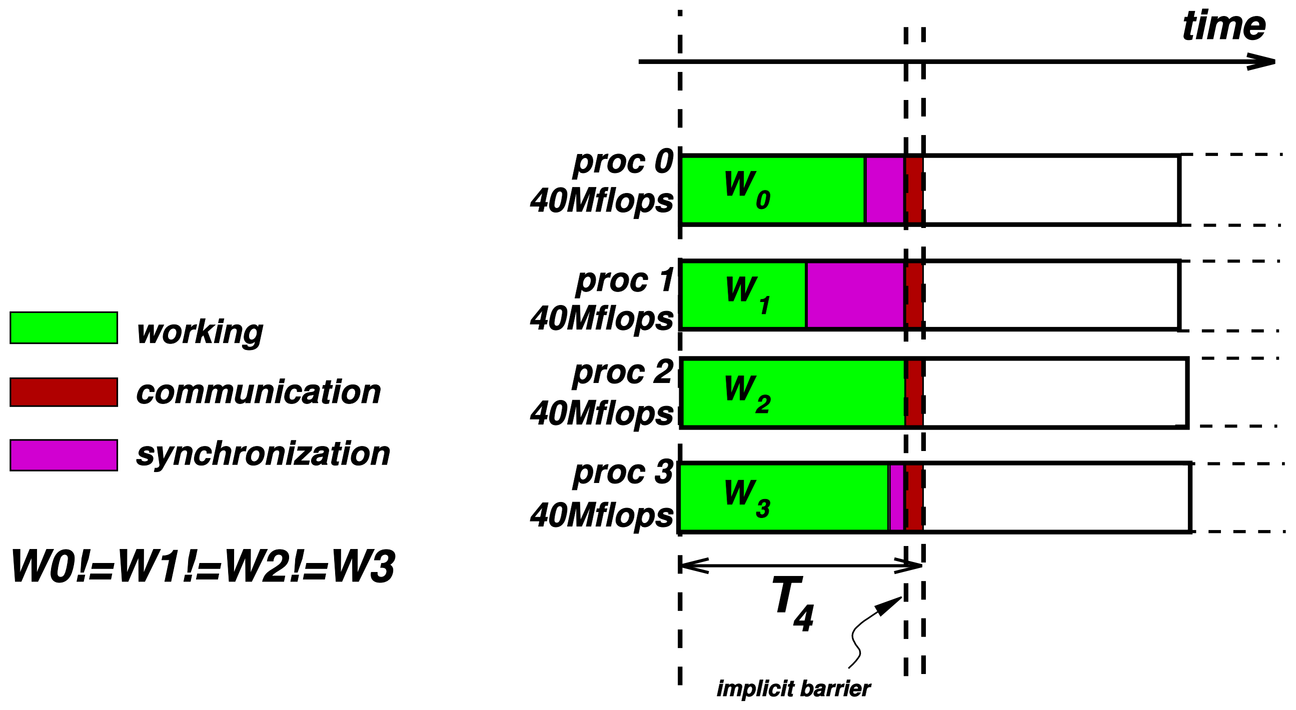
The problem of *scalability* is worst if it is not easy to distribute the load, since in addition to *communication* we have an extra time expended in *synchronization*. In the PNT example, the *cost of the primality test* for a number j greatly varies with j, *for even numbers is immediate*, and the 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 $T_{\mathrm{comp},i} = W_i/s$ 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.)

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

PNT: Detailed scalability analysis

- We use the relation

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

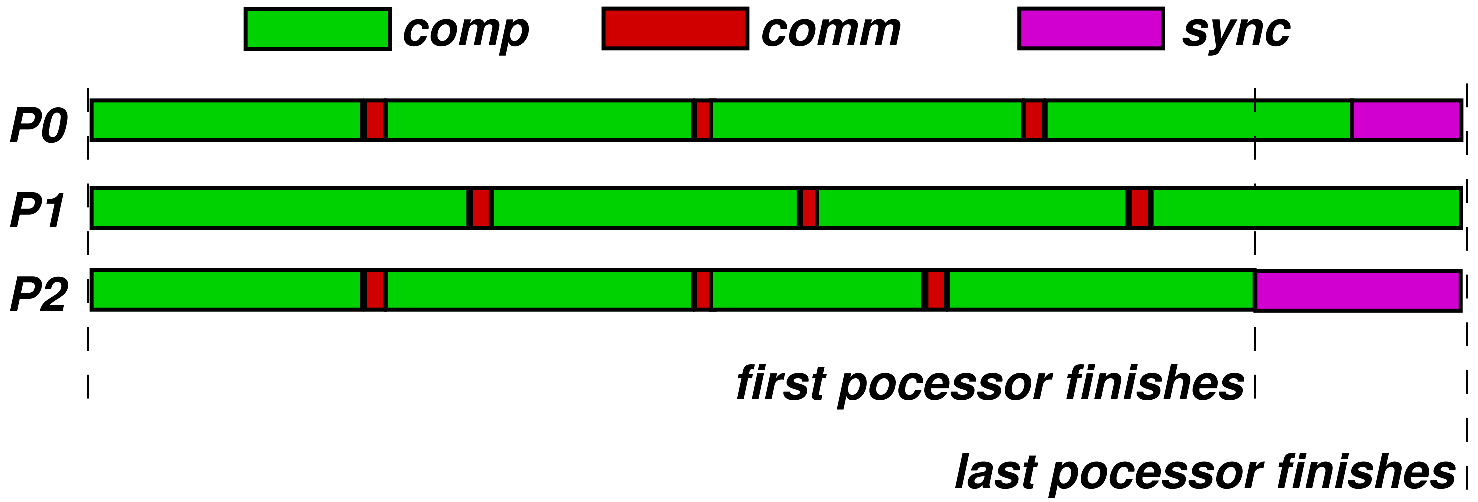
- Time to test integer j for primality: $O(j^{0.5})$
- $T_{\rm comp}(N) = \sum_{j=1}^{N} cj^{0.5} \approx c\int_0^N j^{0.5}\,{\rm d}j = 2cN^{1.5}$
- ***Communication time*** in sending and receiving an integer per chunk

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

where l is latency and N_c is the ***chunk length***.

PNT: Detailed scalability analysis (cont.)

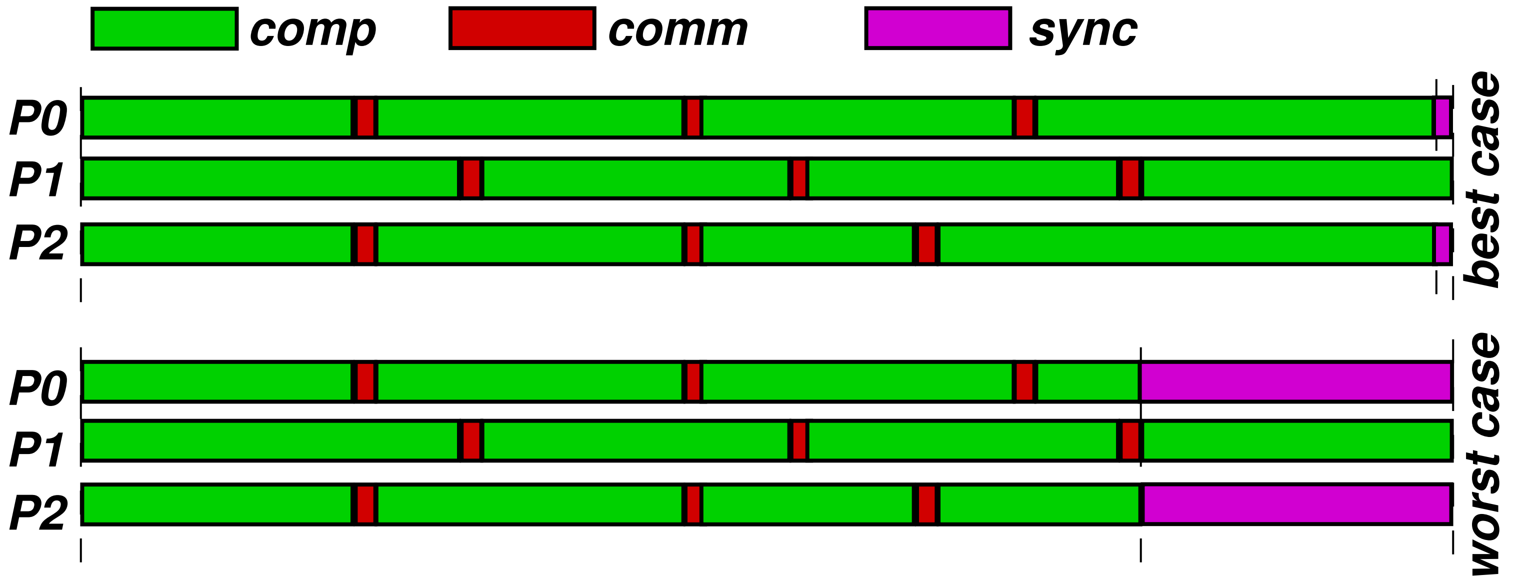
- ***Synchronization time*** **is almost random.**



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, reamining *only one process* still computing.**

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



PNT: Detailed scalability analysis (cont.)

- **In *average***

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

- **In our case**

$$\textbf{(processing time for a chunk)} \leq N_c\, cN^{0.5}$$

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

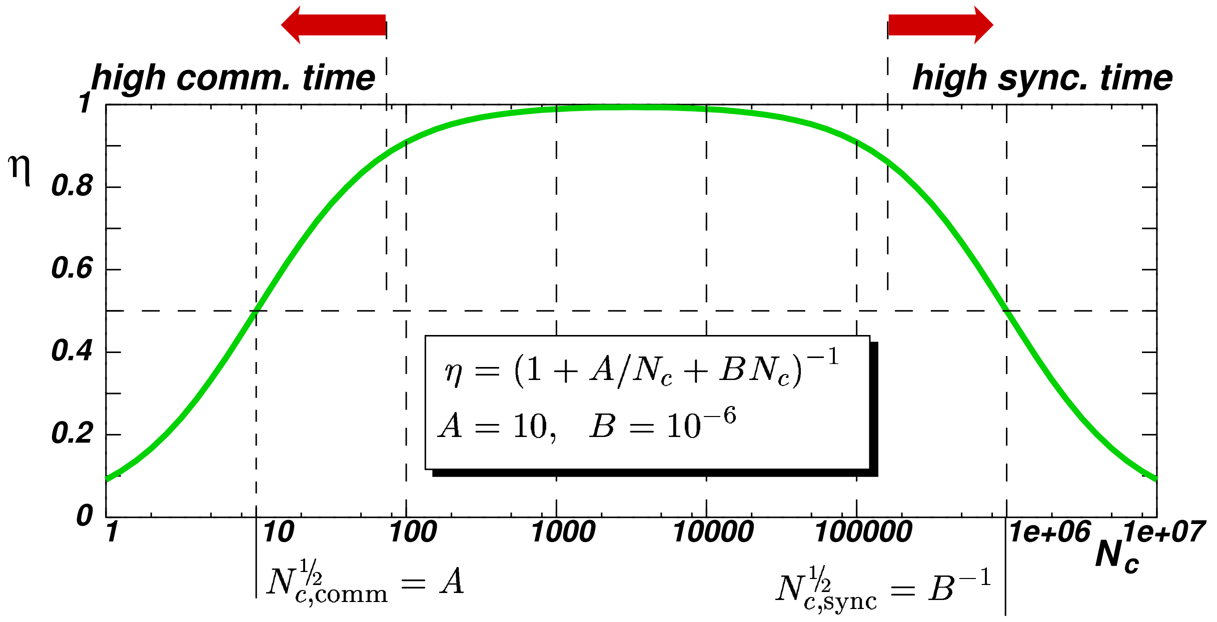
PNT: Detailed scalability analysis (cont.)

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

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 heterogéneos clusters. The concept of speedup must be extended to *heterogéneos 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 $t_i = W_i/s_i = W/ns_i$ seconds, where s_i is the *processing 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**

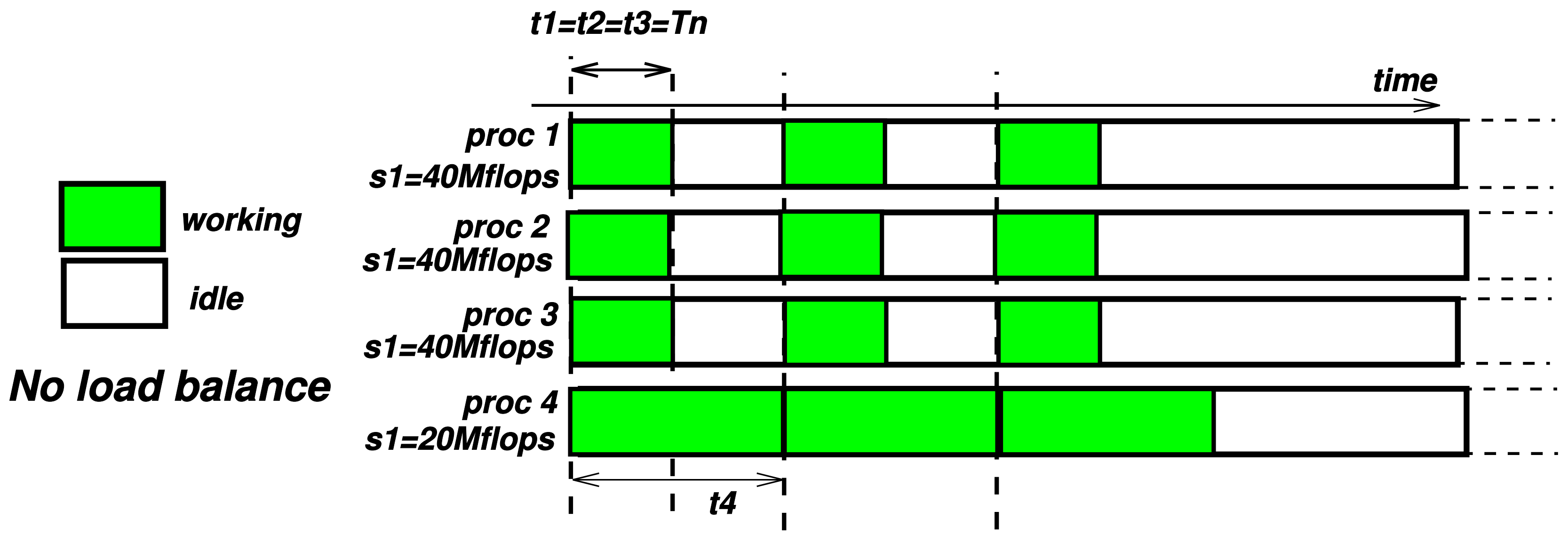
$$T_1 = \min t_i = \frac{W}{\max_i s_i}$$

- **The speedup results then in**

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

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

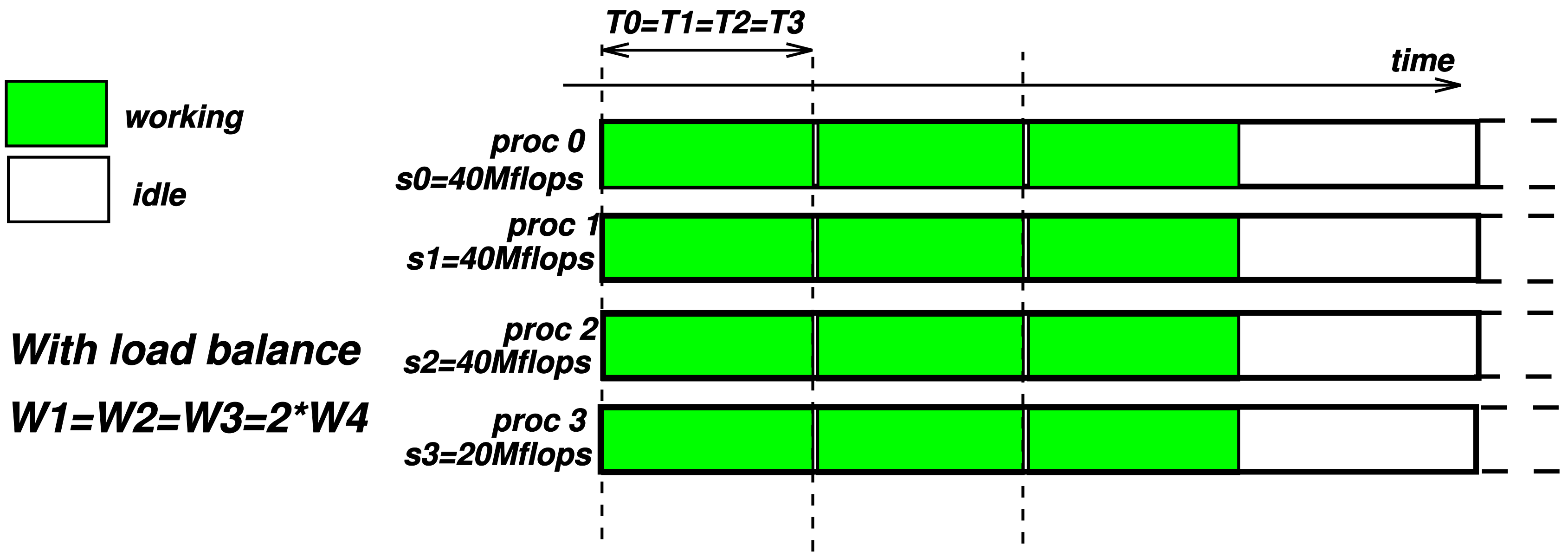
$$t_i = \frac{W_i}{s_i} = \frac{W}{\sum_j s_j} \qquad \text{(independent of } i\text{!!)}$$

- 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 heterogeneouss clusters*.

Load balance (cont.)



With load balance

W1=W2=W3=2*W4

Load balance (cont.)

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

$$S_n^* = \frac{8 \times 4\textbf{Gflops} + 4 \times 2.4\textbf{Gflops}}{4\textbf{Gflops}} = 10.4 \tag{1}$$

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 4x0.6=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 $f(x_i), i = 0, ..., m-1$, for which we have a sequential program `computef -x <x-val>` that prints on standard output the value of $f(x)$.

```
[mstorti@spider curso]> computef -x 0.3
0.34833467364
[mstorti@spider curso]>
```

Trivial parallelism (cont.)

```
// Get parameters 'm' and 'xmax'
if (!myrank) {
  vector<int> proc_j(size,-1);
  vector<double> table(m);
  double x=0., val;
  for (int j=0; j<m; j++) {
    Recv(&val,...,MPI_ANY_SOURCE,...,&status);
    int source = status.MPI_SOURCE;
    if (proc_j[source]>=0)
      table[proc_j[source]] = val;
    MPI_Send(&j,...,source,...);
    proc_start[source]=j;
  }
  for (int j=0; j<size-1; j++) {
    Recv(&val,...,MPI_ANY_SOURCE,...,&status);
    MPI_Send(&m,...,source,...);
  }
} else {
  double
    val = 0.0,
    deltax = xmax/double(m);
  char line[100], line2[100];
  MPI_Send(val,...,0,...);
  while(1) {
    int j;
    MPI_Recv(&j,...,0,...);
    if (j>=m) break;
    sprintf(line,"computef -x %f > proc%d.output",j*deltax,myrank);
    sprintf(line2,"proc%d.output",myrank);
    FILE *fid = fopen(line2,"r");
    fscanf(fid,"%lf",&val);
    system(line)
    MPI_Send(val,. . .,0,. . .);
  }
}
```

The traveling salesman problem (TSP)

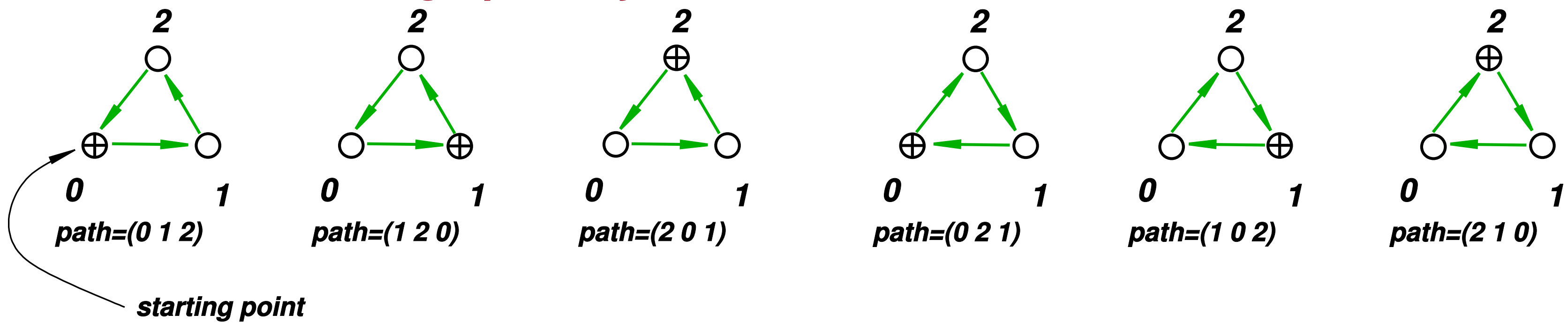
The traveling salesman problem (TSP)

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,

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

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

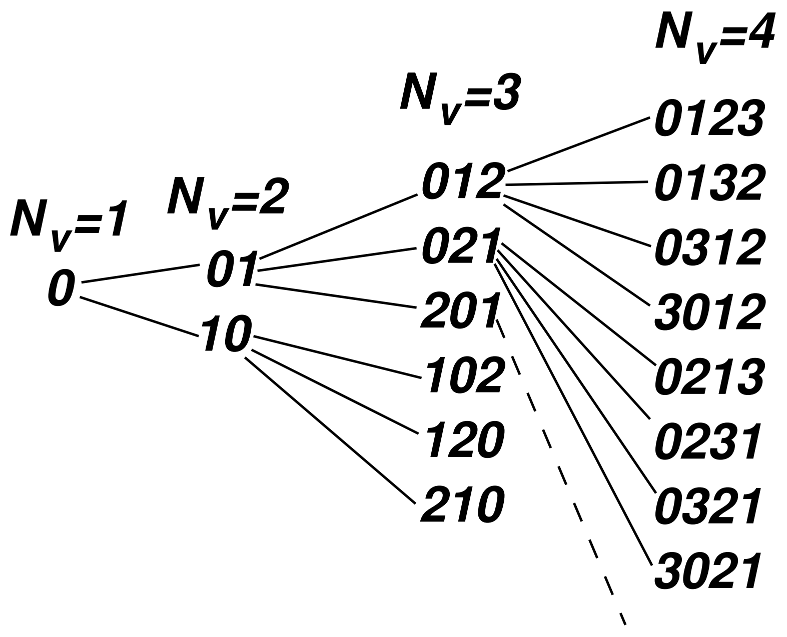
$$N_{\text{path}}(N_v) = \frac{N_v!}{N_v} = (N_v - 1)!$$

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.

```
int next_path(int *path,int N) {
  for (int j=N-1; j>=0; j--) {
    // Is 'j' in first position?
    if (path[0]==j) {
      // move 'j' to the j-th position . . .
    } else {
      // exchange 'j' with its predecesor . . .
      return 1;
    }
  }
  return 0;
}
```

The traveling salesman problem (TSP) (cont.)

$$
\begin{aligned}
p_0 &= (0,1,2)\\
p_1 &= (0,2,1)\\
p_2 &= (2,0,1)\\
p_3 &= (1,0,2)\\
p_4 &= (1,2,0)\\
p_5 &= (2,1,0)
\end{aligned}
$$

- **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 can't 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()`

```
int next_path(int *path,int N) {
  for (int j=N-1; j>=0; j--) {
    if (path[0]==j) {
      for (int k=0; k<j; k++) path[k] = path[k+1];
      path[j] = j;
    } else {
      int k;
      for (k=0; k<N-1; k++)
        if (path[k]==j) break;
      path[k] = path[k-1];
      path[k-1] = j;
      return 1;
    }
  }
  return 0;
}
```

The traveling salesman problem (TSP) (cont.)

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

```
int path[Nv];
for (int j=0; j<Nv; j++) path[j] = j;

while(1) {
  // do something with path....
  if (!next_path(path,Nv)) break;
}
```

For instance, if we define the function `double dist(int *path,int Nv,double *d);` that comput*es the distance for a given path*, we can find the *minimum distance* with the following code

```
int path[Nv];
for (int j=0; j<Nv; j++) path[j] = j;

double dmin = DBL_MAX;
while(1) {
  double D = dist(path,Nv,d);
  if (D<dmin) dmin = D;
  if (!next_path(path,Nv)) break;
}
```

The traveling salesman problem (TSP) (cont.)

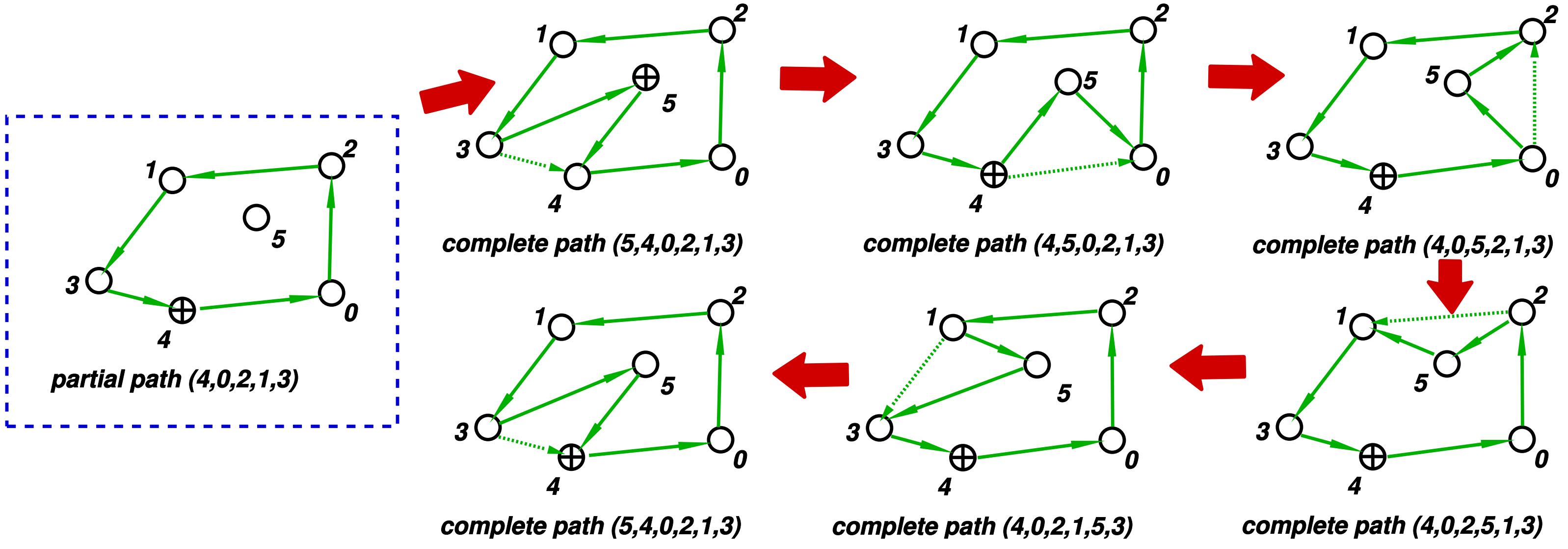
- ***Possible dynamic implementation in parallel:*** Generate ***chunks*** of N_c paths and ***sent to slaves for processing***. For instance, if $N_v = 4$ then there are 24 possible path. If we choose $N_c = 5$ then the master generates chunks and send them to the slaves,
  - ▷ chunk0 = $\{$(0,1,2,3), (0,1,3,2), (0,3,1,2), (3,0,1,2), (0,2,1,3)$\}$
  - ▷ chunk1 = $\{$(0,2,3,1), (0,3,2,1), (3,0,2,1), (2,0,1,3), (2,0,3,1)$\}$
  - ▷ chunk2 = $\{$(2,3,0,1), (3,2,0,1), ...$\}$
  - ▷ ...

  Does not scale well, since we have to send to the slaves $4N_cN$ bytes, so $T_{\rm comm} = 4N_cN/b = O(N_c)$, and $T_{\rm comp} = O(N_c)$.

The traveling salesman problem (TSP) (cont.)

A *partial path* of length N_p is one possible path 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.



partial path (4,0,2,1,3)
complete path (5,4,0,2,1,3)
complete path (4,5,0,2,1,3)
complete path (4,0,5,2,1,3)
complete path (4,0,2,5,1,3)
complete path (4,0,2,1,5,3)
complete path (5,4,0,2,1,3)
0
1
2
3
4
5

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 $T_{\rm comp} = O(N_c)$, 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 identital 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.

```
int next_path(int *path,int N,int Np=0) {
  for (int j=N-1; j>=Np; j--) {
    if (path[0]==j) {
      for (int k=0; k<j; k++) path[k] = path[k+1];
      path[j] = j;
    } else {
      int k;
      for (k=0; k<N-1; k++)
        if (path[k]==j) break;
      path[k] = path[k-1];
      path[k-1] = j;
      return 1;
    }
  }
  return 0;
}
```

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

```
for (int j=0; j<Nv; j++) ppath[j]=j;
while(1) {
  memcpy(path,ppath,Nv*sizeof(int));
  while (1) {
    // do something with complete path . . . .
    if(!next_path(path,Nv,Np)) break;
  }
  if(!next_path(ppath,Np)) break;
}

}
```

The traveling salesman problem (TSP) (cont.)

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

```
for (int j=0; j<Nv; j++) ppath[j]=j;
while(1) {
  printf("parcial path: ");
  print_path(ppath,Np);
  memcpy(path,ppath,Nv*sizeof(int));
  while (1) {
    printf("complete path: ");
    print_path(path,Nv);
    if(!next_path(path,Nv,Np)) break;
  }
  if(!next_path(ppath,Np)) break;
}
```

The traveling salesman problem (TSP) (cont.)

Pseudocode (master code):

```
if (!myrank) {
  int done=0, down=0;
  // Initialize partial path
  for (int j=0; j<N; j++) path[j]=j;
  while(1) {
    // Receive work done by ‘slave’ with ‘MPI_ANY_SOURCE’ . . .
    int slave = status.MPI_SOURCE;
    // Send new next partial path
    MPI_Send(path,N,MPI_INT,slave,0,MPI_COMM_WORLD);
    if (done) down++;
    if (down==size-1) break;
    if(!done && !next_path(path,Np)) {
      done = 1;
      path[0] = -1;
    }
  }
} else {
  // slave code . . .
}
```

The traveling salesman problem (TSP) (cont.)

Pseudocode (slave code):

```
if (!myrank) {
  // master code . . .
} else {
  while (1) {
    // Send work to master (initially garbage) . . .
    MPI_Send(. . .,0,0,MPI_COMM_WORLD);
    // Receive next partial path
    MPI_Recv(path,N,MPI_INT,0,MPI_ANY_TAG,
             MPI_COMM_WORLD,&status);
    if (path[0]==-1) break;
    while (1) {
      // do something with path . . .
      if(!next_path(path,N,Np)) break;
    }
  }
}
```

OPTIONAL Assignement 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 $N_v - 1$ 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 mimimum*, 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.

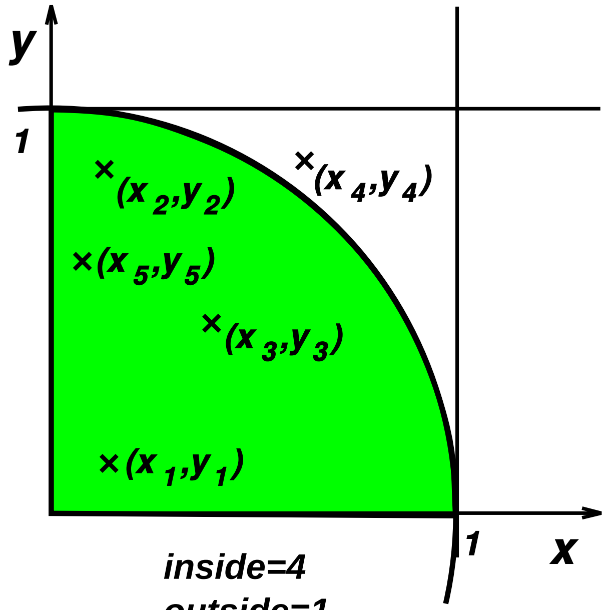
4. Write a parallel version with *dynamic load distribution* (*compute-on-demand*).
5. Perform a *theoretical analysis of the scalability* of the problem. Determine experimentally which is the best partial path length to be sent to the slaves.

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* $\mathrm{prty}(p)$, where p is a path such that if p' is the inverse of p then $\mathrm{prty}(p') = -\mathrm{prty}(p)$. 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 $\mathrm{prty}(03254) = -1$ since 3 is smaller than than 4, while $\mathrm{prty}(45230) = +1$ 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



inside=4
outside=1

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

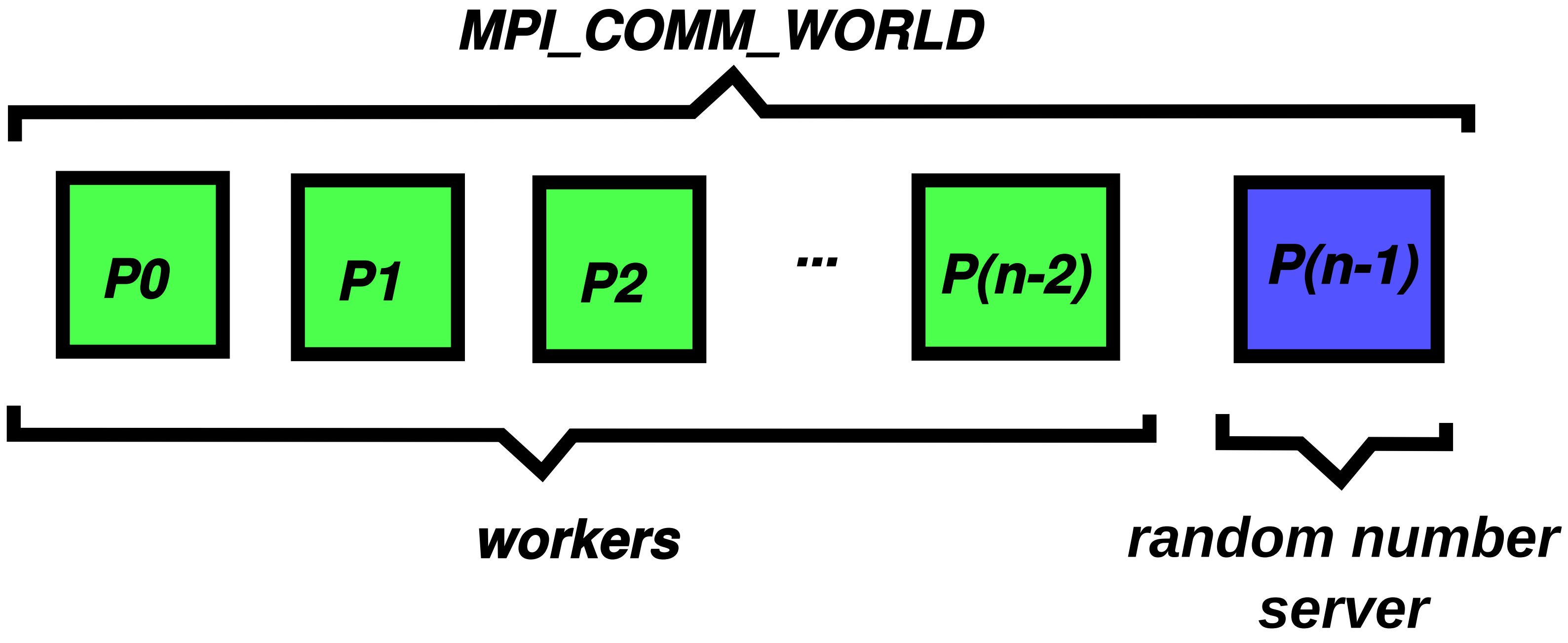
$$\pi = 4\frac{\text{(\#inside)}}{\text{(\#total)}}$$

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.

Random number server



```
#include <math.h>
#include <limits.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
/* no. of random nos. to generate at one time */
#define CHUNKSIZE 1000

/* message tags */
#define REQUEST 1
#define REPLY 2

main(int argc, char **argv) {
  int iter;
  int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
  double x, y, Pi, error, epsilon;
  int numprocs, myid, server, totalin, totalout, workerid;
  int rands[CHUNKSIZE], request;

  MPI_Comm world, workers;
  MPI_Group world_group, worker_group;
  MPI_Status stat;

  MPI_Init(&argc,&argv);
  world = MPI_COMM_WORLD;
  MPI_Comm_size(world, &numprocs);
  MPI_Comm_rank(world, &myid);
  server = numprocs-1;

  if (numprocs==1)
    printf("Error. At least 2 nodes are needed");

```

```
/* process 0 reads epsilon from args and broadcasts it to
   everyone */
if (myid == 0) {
  if (argc<2) {
    epsilon = 1e-2;
  } else {
    sscanf ( argv[1], "%lf", &epsilon );
  }
}
MPI_Bcast ( &epsilon, 1, MPI_DOUBLE, 0,MPI_COMM_WORLD );

/* define the workers communicator by using groups and
   excluding the server from the group of the whole world */
MPI_Comm_group ( world, &world_group );
ranks[0] = server;
MPI_Group_excl ( world_group, 1, ranks, &worker_group );
MPI_Comm_create ( world, worker_group, &workers);
MPI_Group_free ( &worker_group);

/* the random number server code - receives a non-zero
   request, generates random numbers into the array rands,
   and passes them back to the process who sent the
   request. */
if ( myid == server ) {
  do {
    MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
             REQUEST, world, &stat);
    if ( request ) {
      for (i=0; i<CHUNKSIZE; i++) rands[i] = random();
      MPI_Send(rands, CHUNKSIZE, MPI_INT,
               stat.MPI_SOURCE, REPLY, world);
    }
  } while ( request > 0 );
  /* the code for the worker processes - each one sends a
     request for random numbers from the server, receives
     and processes them, until done */
} else {
  request = 1;
  done = in = out = 0;
  max = INT_MAX;

  /* send first request for random numbers */
  MPI_Send( &request, 1, MPI_INT, server, REQUEST, world );

  /* all workers get a rank within the worker group */
  MPI_Comm_rank ( workers, &workerid );

  iter = 0;
  while (!done) {
    iter++;
    request = 1;

    /* receive the chunk of random numbers */
    MPI_Recv(rands, CHUNKSIZE, MPI_INT, server,
             REPLY, world, &stat );
    for (i=0; i<CHUNKSIZE; ) {
      x = (((double) rands[i++])/max) * 2 - 1;
      y = (((double) rands[i++])/max) * 2 - 1;
      if (x*x + y*y < 1.0)
        in++;
      else
        out++;
    }
    /* the value of in is sent to the variable totalin in
```

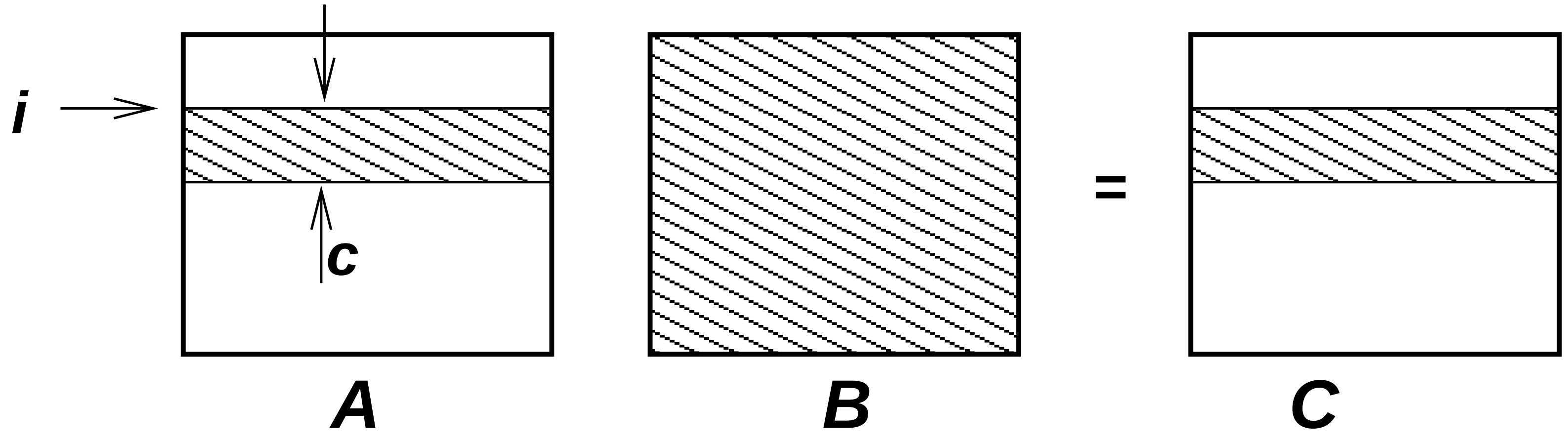
```
             all processes in the workers group */
          MPI_Allreduce(&in, &totalin, 1,
                        MPI_INT, MPI_SUM, workers);
          MPI_Allreduce(&out, &totalout, 1,
                        MPI_INT, MPI_SUM, workers);
          Pi = (4.0*totalin)/(totalin + totalout);
          error = fabs ( Pi - M_PI);
          done = ((error < epsilon) || ((totalin+totalout)>1000000));
          request = (done) ? 0 : 1;

          /* if done, process 0 sends a request of 0 to stop the
             rand server, otherwise, everyone requests more
             random numbers. */
          if (myid == 0) {
            printf("pi = %23.20lf\n", Pi );
            MPI_Send( &request, 1, MPI_INT, server, REQUEST, world);
          } else {
            if (request)
              MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
          }
        }
      }
      if (myid == 0)
        printf("total %d, in %d, out %d\n",
               totalin+totalout, totalin, totalout );
      if (myid<server) MPI_Comm_free(&workers);
      MPI_Finalize();
    }
```

Creating a communicator

```
/* define the workers communicator by
   using groups and excluding the
   server from the group of the whole world */

MPI_Comm world, workers;
MPI_Group world_group, worker_group;
MPI_Status stat;

// . . .

MPI_Comm_group(MPI_COMM_WORLD,&world_group);
ranks[0] = server;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&worker_group);
```

Example: matrix product in parallel

Matrix product



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.

```
// El almacenamiento interno
//  se borra con el destructor
Mat A(100,100);
// usar 'A' . . .


// El almacenamiento externo
//  debe ser borrado por el usuario
double *b = new double[10000];
Mat B(100,100,b);
// usar 'B' . . .
delete[] b;
```

Simple matrix library (cont.)

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

```
double x,w;
x = A(i,j);
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.

```
Mat A(100,100);
double *a, *a10;
// Puntero al area de almacenamiento interno
a = &A(0,0);
// Puntero al area de almacenamiento a partir de
//  la linea 10 (base 0)
a10 = &A(10,0);
```

Simple matrix library (cont.)

```
/// Simple matrix class
class Mat {
private:
  /// The store
  double *a;
  /// row and column dimensions, flag if the store
  /// is external or not
  int m,n,a_is_external;
public:
  /// Constructor from row and column dimensions
  /// and (eventually) the external store
  Mat(int m_,int n_,double *a_=NULL);
  /// Destructor
  ~Mat();
  /// returns a reference to an element
  double & operator()(int i,int j);
  /// returns a const reference to an element
  const double & operator()(int i,int j) const;
  /// product of matrices
  void matmul(const Mat &a, const Mat &b);
  /// prints the matrix
  void print() const;
};
```

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 $1 <= \texttt{chunksize} \ll N$ so that the associated overhead with sending and receiving data is negligible.**

Slave code

- **Sends a message of len=0 that means *I'm ready*.**
- **Receivs an amount of tasks of length *len*. If len>0, process them and sends back the results.**
  If len=0, sends a len=0 aknowledge message, exits the loop and ends processing.

```
Send len=0 to server
while(true) {
    Receive len,tag=k
    if (len=0) break;
    Process x_k,x_{k+1},...,x_{k+len-1} ->
                            r_k,r_{k+1},...,r_{k+len-1}
    Send r_k,r_{k+1},...,r_{k+len-1} to server
}
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.

```
proc_status[1..P-1] =0;

j=0;
while (true) {
      Receive len,tag->k, proc
      if (len>0) {
         extrae r_k,...,r_{k+len-1}
      } else {
        proc_status[proc]++;
      }
      jj = min(N,j+chunksize);
      len = jj-j;
      Send x_j,x_{j+1}, x_{j+len-1}, tag=j to proc
      // Verifica si todos terminaron
      Si proc_status[proc] == 2 para proc=1..P-1 then break;
}
```

Slave code, use local processor

```
proc_status[1..P-1] =0;
j=0
while (true) {
      Immediate Receive len,tag->k, proc
      while(j<N) {
         if (received) break;
         Process x_j -> r_j
         j++
       }
      Wait until received;
      if (len==0) {
         extrae r_k,...,r_{k+len-1}
      } else {
         proc_status[proc]++;
      }
      jj = min(N,j+chunksize);
      len = jj-j;
      Send x_j,x_{j+1}, x_{j+len-1}, tag=j to proc
      // Verifica si todos terminaron
      Si proc_status[proc] == 2 para proc=1..P-1 then break;
}
```

Complete code

```
#include <time.h>
#include <unistd.h>
#include <ctype.h>
#include "mat.h"

/* Self-scheduling algorithm.
   Computes the product C=A*B; A,B and C matrices of NxN
   Proc 0 is the root and sends work to the slaves
 */
// $Id: matmult2.cpp,v 1.1 2004/08/28 23:05:28 mstorti Exp $

/** Computes the elapsed time between two instants captured with
    `gettimeofday'.
*/
double etime(timeval &x,timeval &y) {
  return double(y.tv_sec)-double(x.tv_sec)
    +(double(y.tv_usec)-double(x.tv_usec))/1e6;
}

/** Computes random numbers between 0 and 1
*/
double drand() {
  return ((double)(rand()))/((double)(RAND_MAX));
}

/** Computes an intger random number in the
    range `imin'-`imax'
*/
```

```
int irand(int imin,int imax) {
  return int(rint(drand()*double(imax-imin+1)-0.5))+imin;
}


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

  /// Initializes MPI
  MPI_Init(&argc,&argv);
  /// Initializes random
  srand(time (0));

  /// Size of problem, size of chunks (in rows)
  int N,chunksize;
  /// root procesoor, number of processor, my rank
  int root=0, numprocs, rank;
  /// time related quantities
  struct timeval start, end;

  /// Status for retrieving MPI info
  MPI_Status stat;
  /// For non-blocking communications
  MPI_Request request;
  /// number of processors and my rank
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /// cursor to the next row to send
  int row_start=0;
  /// Read input data from options
  int c,print_mat=0,random_mat=0,use_local_processor=0,
    slow_down=1;
  while ((c = getopt (argc, argv, "N:c:prhls:")) != -1) {
    switch (c) {
    case 'h':
      if (rank==0) {
        printf(" usage: $ mpirun [OPTIONS] matmult2\n\n"
               "MPI options: -np <number-of-processors>\n"
               "              -machinefile <machine-file>\n\n"
               "Other options: -N <size-of-matrix>\n"
               "                -c <chunk-size> "
               "# sets number of rows sent to processors\n"
               "                -p              "
               "# print input and result matrices to output\n"
               "                -r              "
               "# randomize input matrices\n");
      }
      exit(0);
    case 'N':
      sscanf(optarg,"%d",&N);
      break;
    case 'c':
      sscanf(optarg,"%d",&chunksize);
      break;
    case 'p':
      print_mat=1;
      break;
    case 'r':
      random_mat=1;
      break;
    case 'l':
      use_local_processor=1;
      break;
    case 's':
      sscanf(optarg,"%d",&slow_down);
      if (slow_down<1) {
        if (rank==0) printf("slow_down factor (-s option)"
                            " must be >=1. Reset to 1!\n");
        abort();
      }
      break;
    case '?':
      if (isprint (optopt))
        fprintf (stderr, "Unknown option `-%c'.\n", optopt);
      else
        fprintf (stderr,
                 "Unknown option character `\\x%x'.\n",
                 optopt);
      return 1;
    default:
      abort ();
    }
  }

#if 0
  if ( rank == root ) {
    printf("enter N, chunksize > ");
    scanf("%d",&N);
    scanf("%d",&chunksize);
    printf("\n");
  }
#endif

  /// Register time for statistics
  gettimeofday (&start,NULL);
  /// broadcast N and chunksize to other procs.
  MPI_Bcast (&N, 1, MPI_INT, 0,MPI_COMM_WORLD );
  MPI_Bcast (&chunksize, 1, MPI_INT, 0,MPI_COMM_WORLD );
```

```

/// Define matrices, bufa and bufc is used to
/// send matrices to other processors
Mat b(N,N),bufa(chunksize,N), bufc(chunksize,N);

//---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
//---:---<*>---:- SERVER PART    *>---:---<*>
//---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
if ( rank == root ) {

  Mat a(N,N),c(N,N);

  /// Initialize 'a' and 'b'
  for (int j=0; j<N; j++) {
    for (int k=0; k<N; k++) {
      // random initialization
      if (random_mat) {
        a(j,k) = floor(double(rand())/double(INT_MAX)*4);
        b(j,k) = floor(double(rand())/double(INT_MAX)*4);
      } else {
        // integer index initialization (eg 00 01 02 . . . . NN)
        a(j,k) = &a(j,k)-&a(0,0);
        b(j,k) = a(j,k)+N*N;
      }
    }
  }

  /// proc_status[proc] = 0 -> processor not contacted
  ///                   = 1 -> processor is working
  ///                   = 2 -> processor has finished
  int *proc_status = (int *) malloc(sizeof(int)*numprocs);
  /// statistics[proc] number of rows that have been processed
  /// by this processor
```

```
    int *statistics = (int *) malloc(sizeof(int)*numprocs);

    /// initializate proc_status, statistics
    for (int proc=1; proc<numprocs; proc++) {
      proc_status[proc] = 0;
      statistics [proc] = 0;
    }

    /// send b to all processor
    MPI_Bcast (&b(0,0), N*N, MPI_DOUBLE, 0,MPI_COMM_WORLD );

    /// Register time for statistics
    gettimeofday (&end,NULL);
    double bcast = etime(start,end);

    while (1) {

      /// non blocking receive
      if (numprocs>1) MPI_Irecv(&bufc(0,0),chunksize*N,
                                MPI_DOUBLE,MPI_ANY_SOURCE,
                                MPI_ANY_TAG,
                                MPI_COMM_WORLD,&request);

      /// While waiting for results from slaves server works . . .
      int receive_OK=0;
      while (use_local_processor && row_start<N) {
        /// Test if some message is waiting.
        if (numprocs>1) MPI_Test(&request,&receive_OK,&stat);
        if(receive_OK) break;
        /// Otherwise. . . work
        /// Local masks
        Mat aa(1,N,&a(row_start,0)),cc(1,N,&c(row_start,0));
        /// make product
        for (int jj=0; jj<slow_down; jj++)
          cc.matmul(aa,b);
        /// increase cursor
        row_start++;
        /// register work
        statistics[0]++;
      }

      /// If no more rows to process wait
      // until message is received
      if (numprocs>1) {
        if (!receive_OK) MPI_Wait(&request,&stat);

        /// length of received message
        int rcvlen;
        /// processor that sent the result
        int proc = stat.MPI_SOURCE;
        MPI_Get_count(&stat,MPI_DOUBLE,&rcvlen);

        if (rcvlen!=0) {
          /// Store result in 'c'
          int rcv_row_start = stat.MPI_TAG;
          int nrows_sent = rcvlen/N;
          for (int j=rcv_row_start;
               j<rcv_row_start+nrows_sent; j++) {
            for (int k=0; k<N; k++) {
              c(j,k) = bufc(j-rcv_row_start,k);
            }
          }
        } else {
          /// Zero length messages are used
          // to acknowledge start work
          // and finished work. Increase
          // status of processor.
          proc_status[proc]++;
        }

        /// Rows to be sent
        int row_end = row_start+chunksize;
        if (row_end>N) row_end = N;
        int nrows_sent = row_end - row_start;

        /// Increase statistics, send task to slave
        statistics[proc] += nrows_sent;
        MPI_Send(&a(row_start,0),nrows_sent*N,MPI_DOUBLE,
                 proc,row_start,MPI_COMM_WORLD);
        row_start = row_end;
      }

      /// If all processors are in state 2,
      // then all work is done
      int done = 1;
      for (int procc=1; procc<numprocs; procc++) {
        if (proc_status[procc]!=2) {
          done = 0;
          break;
        }
      }
      if (done) break;
    }

    /// time statistics
    gettimeofday (&end,NULL);
    double dtime = etime(start,end);
    /// print statistics
    double dN = double(N), rate;
    rate = 2*dN*dN*dN/dtime/1e6;
    printf("broadcast: %f secs [%5.1f %%], process: %f secs\n"
           "rate: %f Mflops on %d procesors\n",
           bcast,bcast/dtime*100.,dtime-bcast,
           rate,numprocs);
    if (slow_down>1) printf("slow_down=%d, scaled Mflops %f\n",
                           slow_down,rate*slow_down);
    for (int procc=0; procc<numprocs; procc++) {
      printf("%d lines processed by %d\n",
             statistics[procc],procc);
    }

    if (print_mat) {
      printf("a: \n");
      a.print();
      printf("b: \n");
      b.print();
      printf("c: \n");
      c.print();
    }

    /// free memory
    free(statistics);
    free(proc_status);

  } else {

    //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
```

```
//---:---<*>---:- SLAVE PART     *>---:---<*>
//---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>

/// get 'b'
MPI_Bcast(&b(0,0), N*N, MPI_DOUBLE, 0,MPI_COMM_WORLD );
/// Send message zero length 'I'm ready'
MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);

while (1) {
  /// Receive task
  MPI_Recv(&bufa(0,0),chunksize*N,MPI_DOUBLE,0,MPI_ANY_TAG,
           MPI_COMM_WORLD,&stat);
  int rcvlen;
  MPI_Get_count(&stat,MPI_DOUBLE,&rcvlen);
  /// zero length message means: no more rows to process
  if (rcvlen==0) break;

  /// compute number of rows received
  int nrows_sent = rcvlen/N;
  /// index of first row sent
  int row_start = stat.MPI_TAG;
  /// local masks
  Mat bufaa(nrows_sent,N,&bufa(0,0)),
    bufcc(nrows_sent,N,&bufc(0,0));
  /// compute product
  for (int jj=0; jj<slow_down; jj++)
    bufcc.matmul(bufaa,b);
  /// send result back
  MPI_Send(&bufcc(0,0),nrows_sent*N,MPI_DOUBLE,
           0,row_start,MPI_COMM_WORLD);

    }

    /// Work finished. Send acknowledge (zero length message)
    MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);

  }

  // Finallize MPI
  MPI_Finalize();
}
```

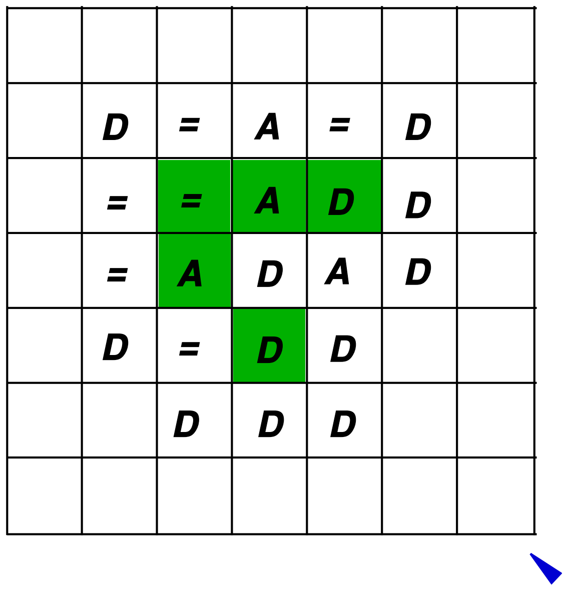
Game of life

Game of life

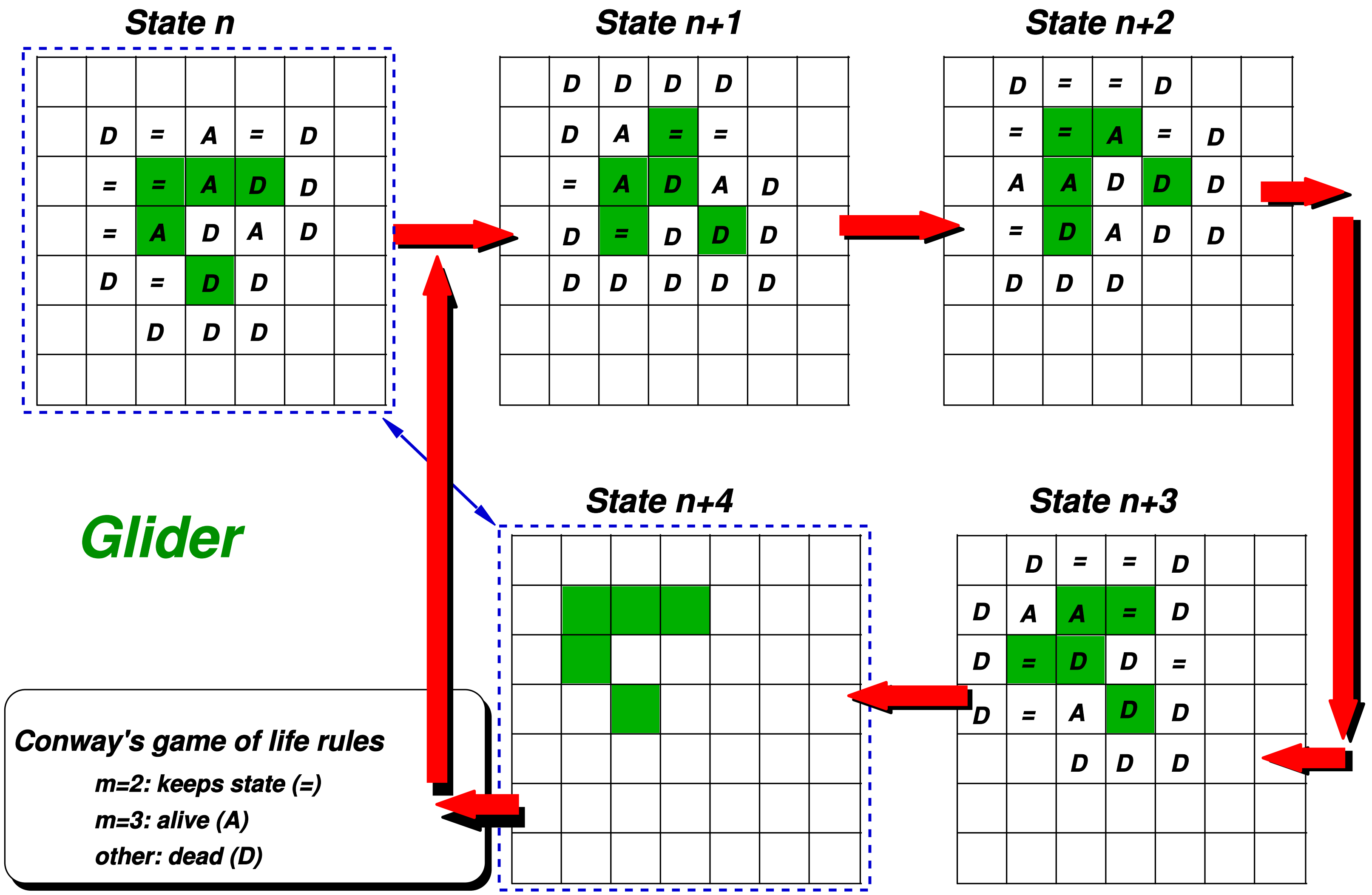
Conway's Game of Life **is an example of *cellular automata* with simple evolution rules that leads to complex behavior (*self-organization* and *emergence*).**

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"*).**



Game of life (cont.)



Conway's game of life rules

- *m=2: keeps state (=)*
- *m=3: alive (A)*
- *other: dead (D)*

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

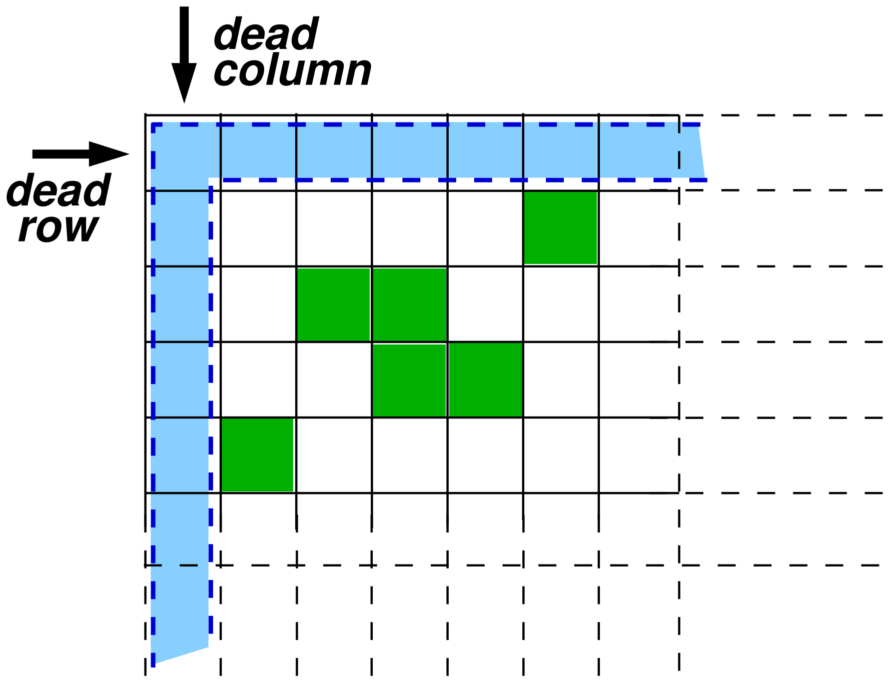
$$a_{ij}^{n+1} = \begin{cases} 1 & ; \text{ if } a_{ij}^n = 3 \\ c_{ij}^n & ; \text{ if } a_{ij}^n = 2 \\ 0 & ; \text{ otherwise.} \end{cases}$$

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,

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



Game of life (cont.)

A class *Board*

```
Board board;
// Initializes the board. A vector of weights is passed.
void board.init(N,weights);
// Sets the the state of a cell.
void board.setv(i,j,state);
// rescatter the board according to new weights
void board.re_scatter(new_weights);
// Advances the board one generation.
void board.advance();
// Prints the board
void print();
// Print statistics about estiamted processing rates
void board.statistics();
// Reset statistics
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`

```
char *tmp = board[j];
board[j] = board[k];
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*.

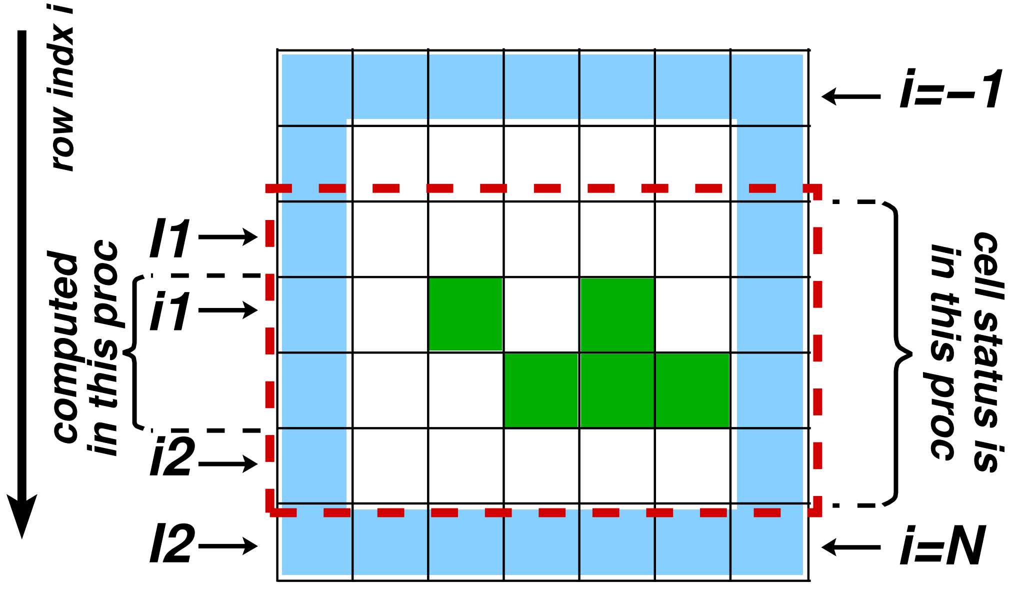
```
rows_proc.resize(size);
double sum_w = 0., carry=0., a;
for (int p=0; p<size; p++) sum_w += w[p];
int j = 0;
double tol = 1e-8;
for (int p=0; p<size; p++) {
  w[p] /= sum_w;
  a = w[p]*N + carry + tol;
  rows_proc[p] = int(floor(a));
  carry = a - rows_proc[p] - tol;
  if (p==myrank) {
    i1 = j;
    i2 = i1 + rows_proc[p];
  }
  j += rows_proc[p];
}
assert(j==N);
assert(fabs(carry) < tol);

```

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

Game of life (cont.)

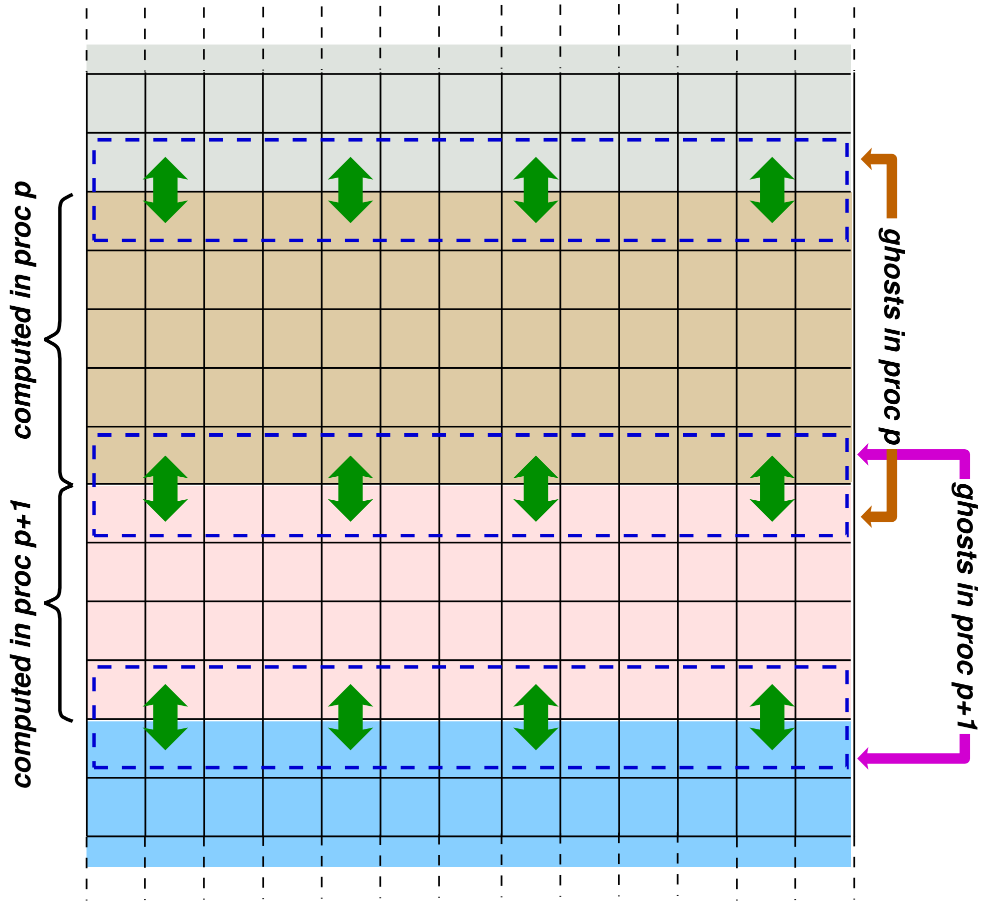
- The *range* of rows to process in each processor is `[i1,i2)`
- Each processor keeps cell states for rows in a *broader* range `[I1,I2)` that contains one more row in each direction, i.e. `I1=i1-1`, `I2=i2+1`.

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



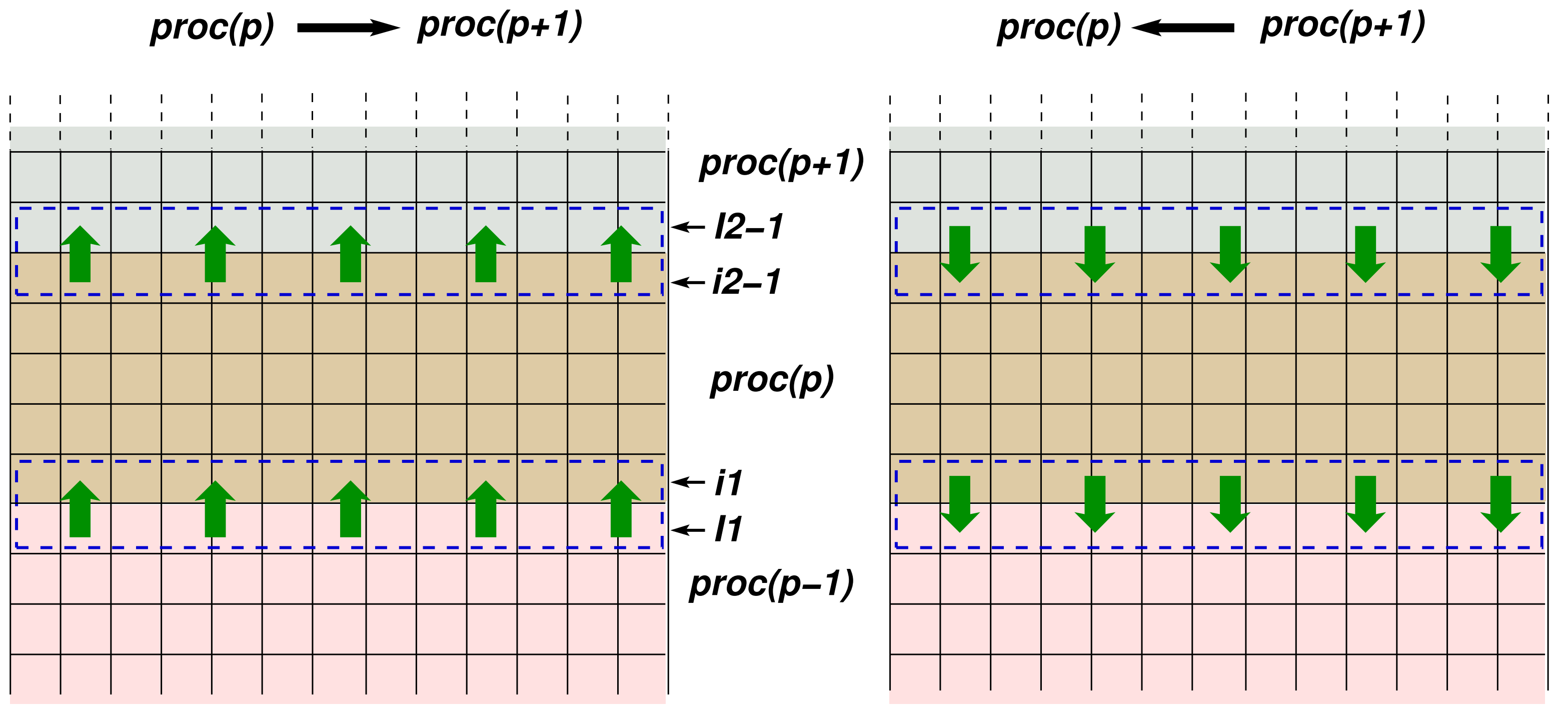
Game of life (cont.)

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



Game of life (cont.)

```
if (myrank<size-1) SEND(row(i2-1),myrank+1);
if (myrank>0) RECEIVE(row(I1),myrank-1);

if (myrank>0) SEND(row(i1),myrank-1);
if (myrank<size-1) RECEIVE(row(I2),myrank+1);
```

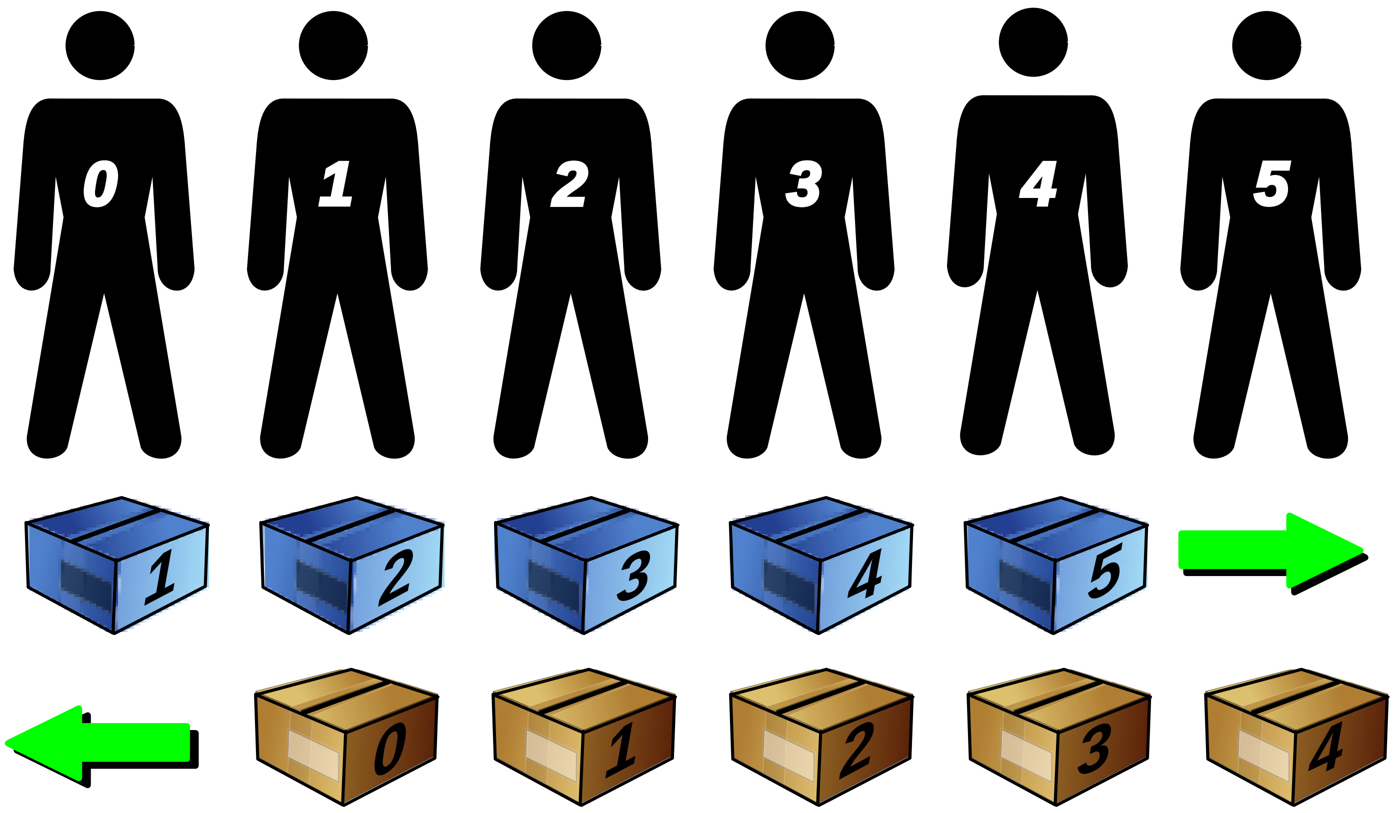


Game of life (cont.)

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. In fact, if *periodic* boundary conditios are used, *this communication pattern is blocking*.

Game of life (cont.)

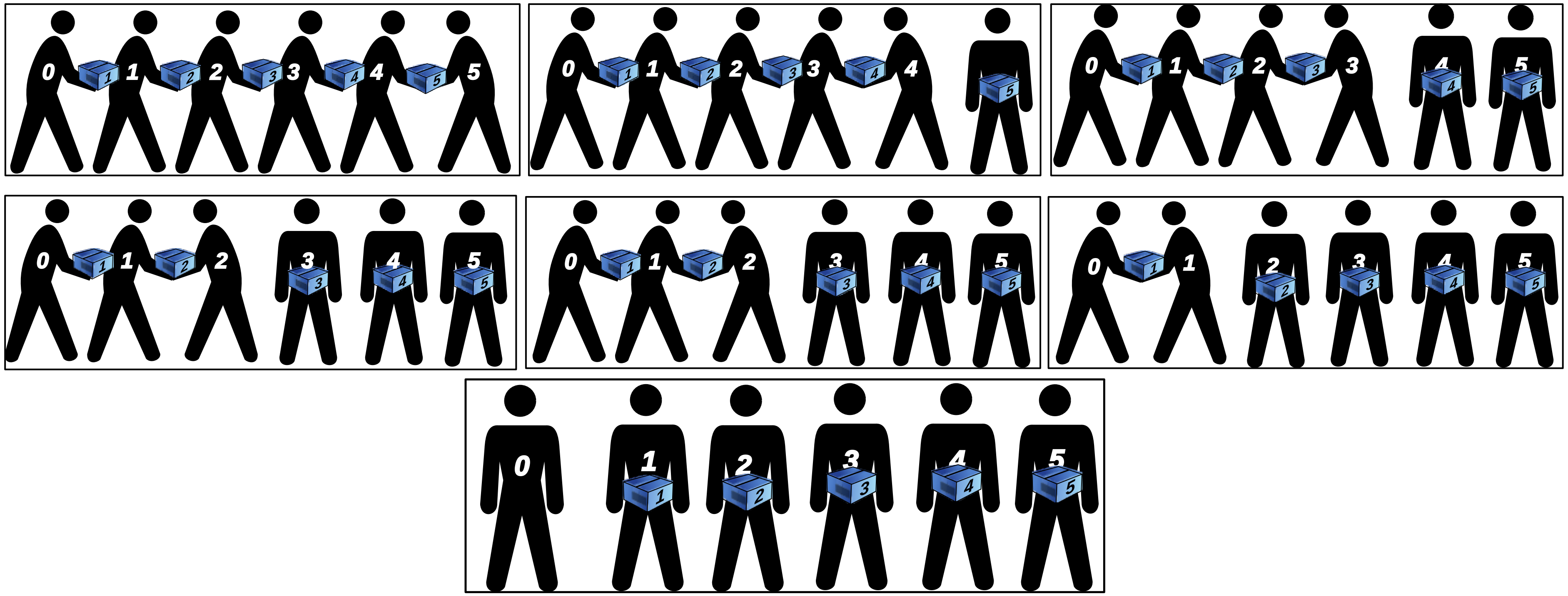
Processors 0-5 must pass the blue boxes to the right and the brown boxes to the left. (The number in the box is the destination process).



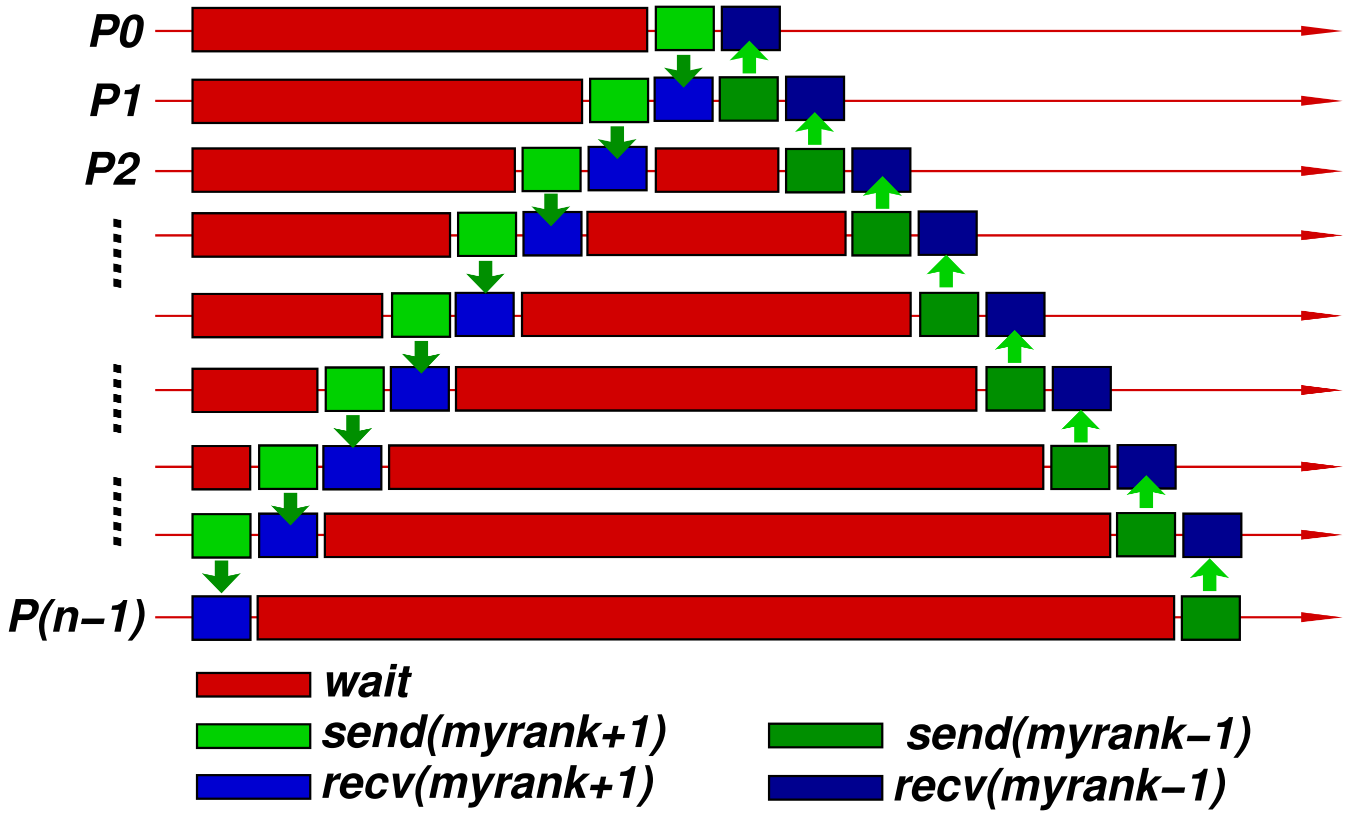
Game of life (cont.)

Communication going right (blue boxes):

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

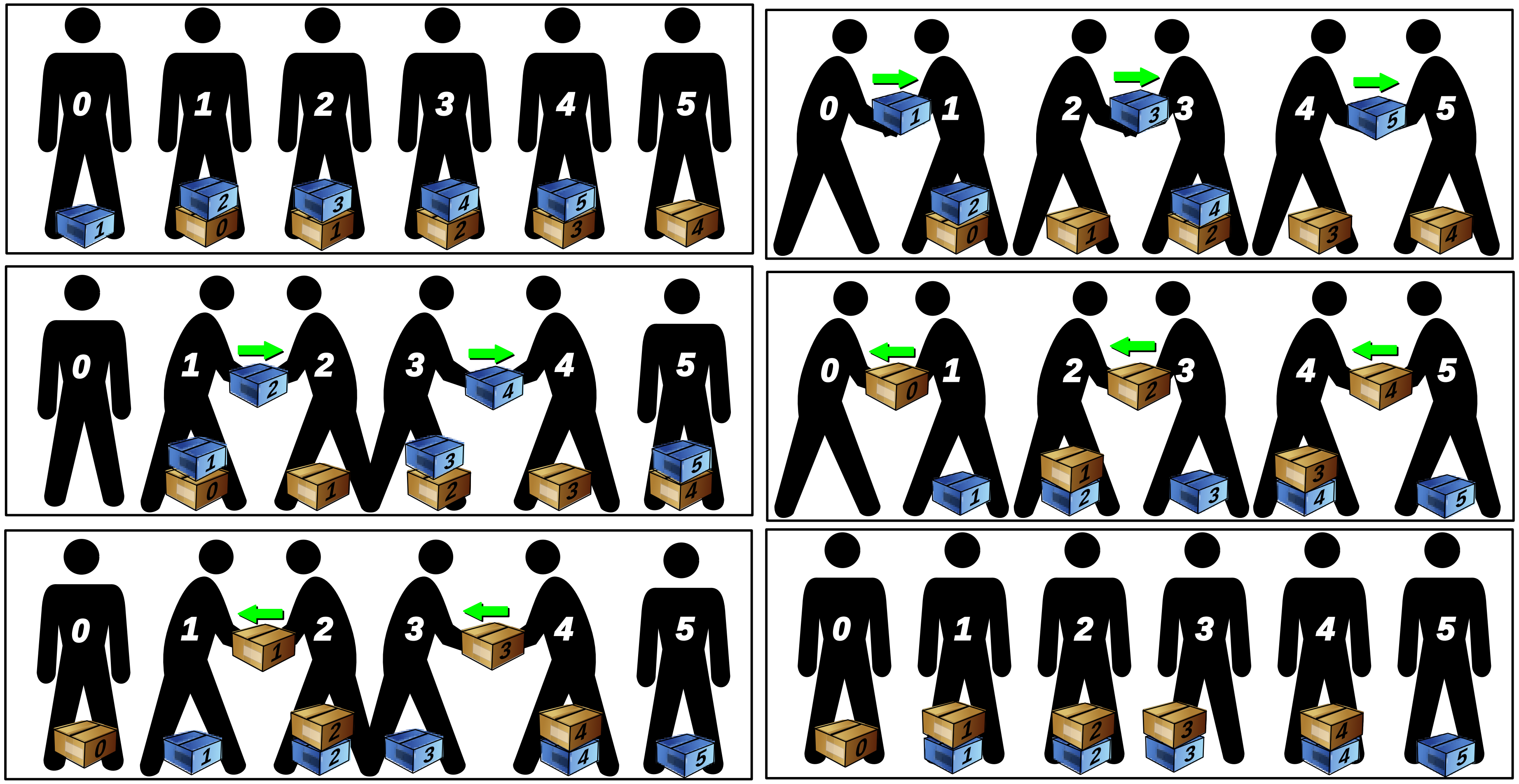


Game of life (cont.)



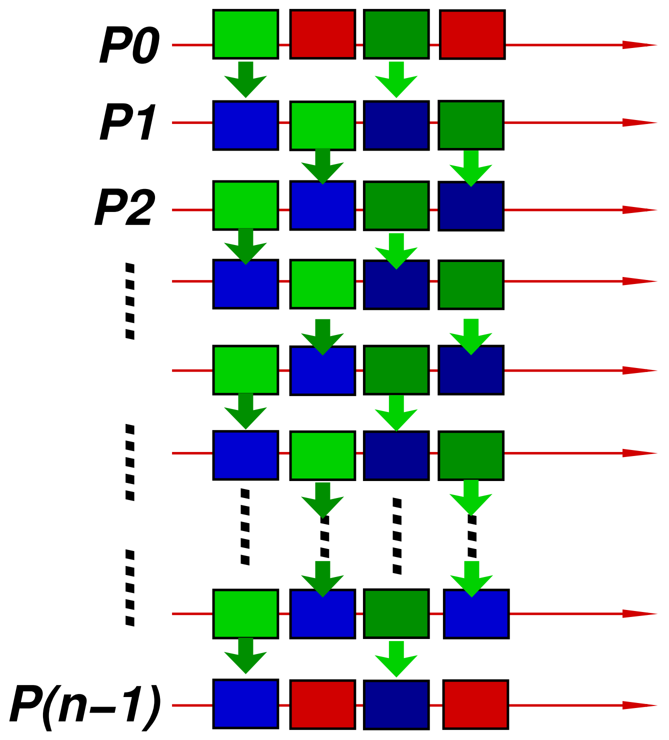
Game of life (cont.)

By *splitting* the workers in *even/odd* we can do the whole communication in *4 stages*.



Game of life (cont.)

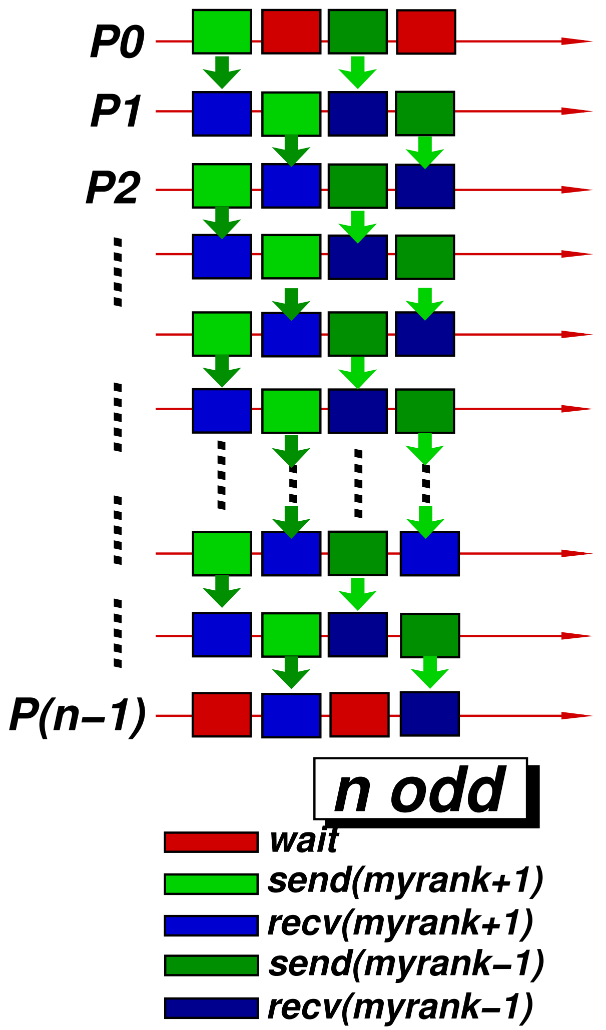
```
if (myrank % 2 == 0 ) {
  if (myrank<size-1) {
    SEND(i2-1,myrank+1);
    RECEIVE(I2-1,myrank+1);
  }
} else {
  if (myrank>0) {
    RECEIVE(I1,myrank-1);
    SEND(i1,myrank-1);
  }
}

if (myrank % 2 == 0 ) {
  if (myrank>0) {
    SEND(i1,myrank-1);
    RECEIVE(I1,myrank-1);
  }
} else {
  if (myrank<size-1) {
    RECEIVE(I2-1,myrank+1);
    SEND(i2-1,myrank+1);
  }
}
```



n even

🟥 wait
🟩 send(myrank+1)
🟦 recv(myrank+1)
🟩 send(myrank−1)
🟦 recv(myrank−1)

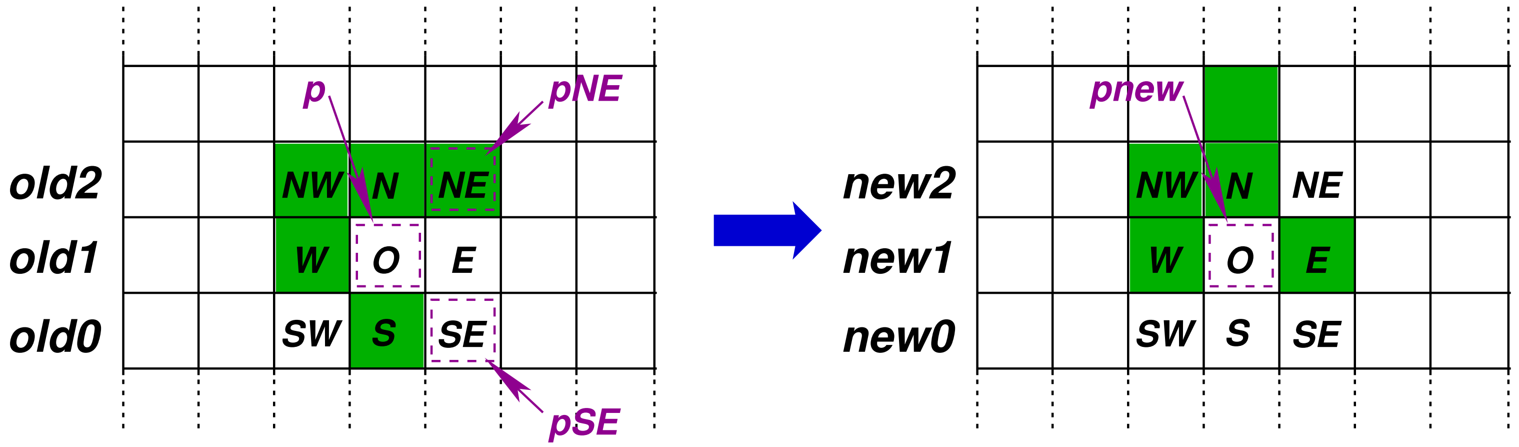
Game of life (cont.)



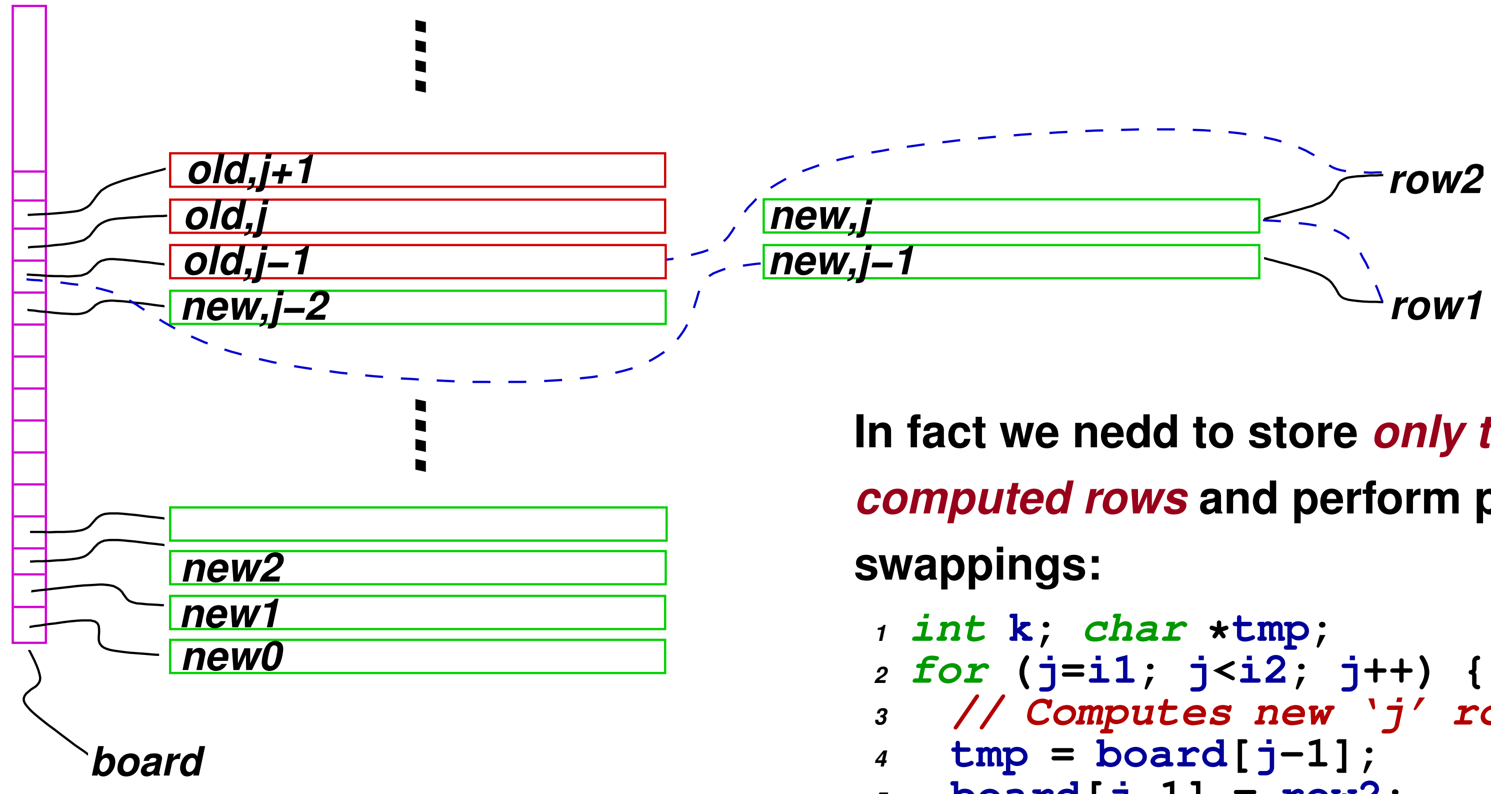
This communication pattern requires $T = 4\tau$, where τ is the *time needed for communicating a cell row*, whilst the previous *chained* version requires $T = 2(N_p - 2)\tau$ where N_p is the number of processors.
It may be applied also if the number of processors is *odd*.

Game of life (cont.)

- The cells are computed *by row*, from left to right ($W \to E$) and from top to bottom ($S \to N$).
- 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 if 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.



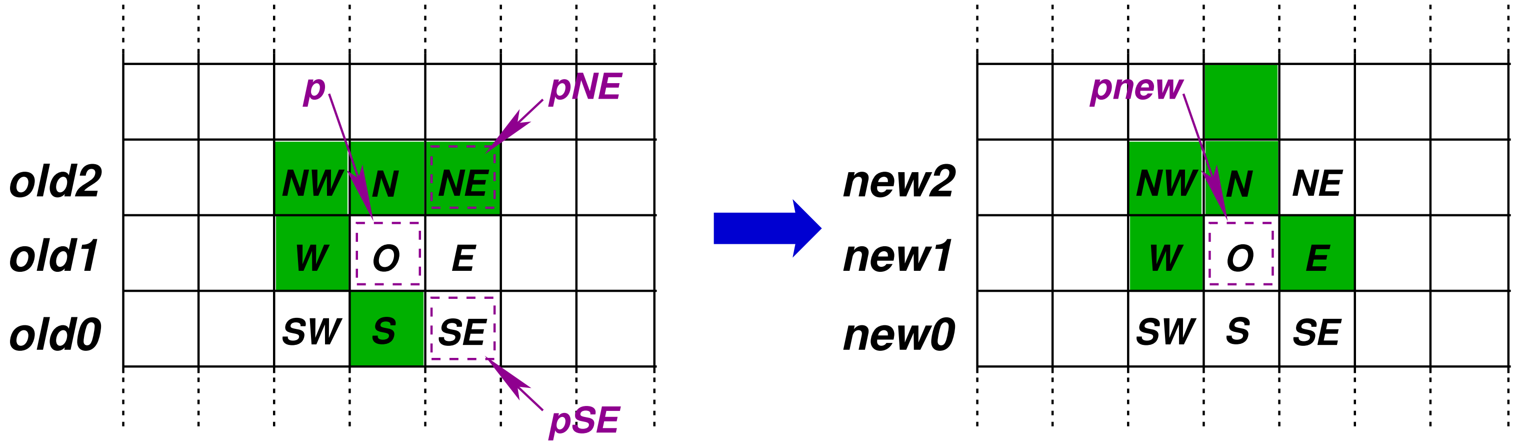
Game of life (cont.)



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

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

Game of life (cont.)



- The *kernel* consists in the function

```
void go(int N, char *old0, char *old1,
        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 alive cells* in the corresponding columns, i.e.

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

so that the number of *alive cells* is

```
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

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

whilst the *communication time* will be

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

so that we have

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

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.

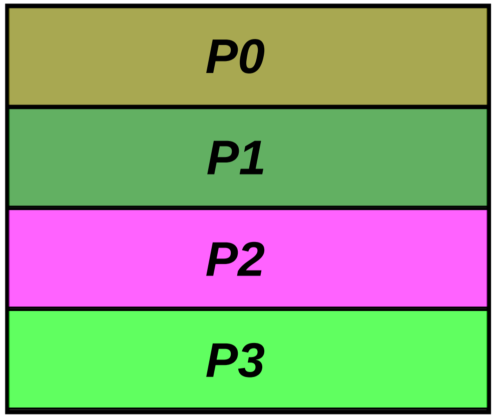
Scalability analysis (cont.)

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 $T_{\rm comm} = 8N/bn^{0.5}$. (We assume n is a perfect square integer) and the efficiency is

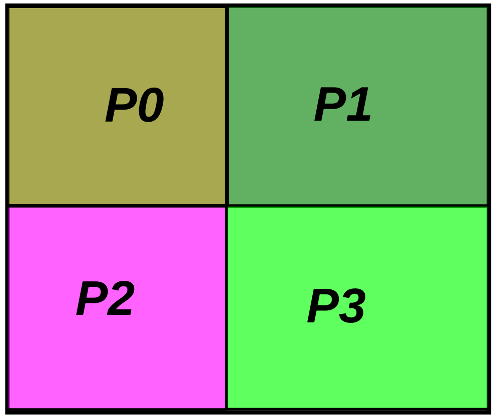
$$\eta = \frac{N^2/sn}{N^2/sn + 8N/bn^{0.5}}$$

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

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 uf 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 ed 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 rime will be

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

whereas the communication time will be

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

Life with dynamic load balance (cont.)

Much in the same line as in the previous analysis for the *PNT with dynamic load balance*:

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

***Efficiency* will be, then**

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

$$C = 1 + s/b; \quad A = 2s/b; \quad B = n/2N$$

Life with dynamic load balance (cont.)

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

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

$$T_{\mathrm{comp}} = T_{\mathrm{comm}}, \quad \textbf{para } N_c = N^{1/2}_{c,\mathrm{comm}}$$

If we consider the *computing speed* as $s = 115\ \mathrm{Mcell/sec}$ **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** $b = 90\ \mathrm{Mbit/sec} = 11\ \mathrm{Mcell/sec}$**. 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.)

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

On the other hand, the term BN_c will be negligible for
$N_c >> C/B = N_{c,\mathrm{sync}}^{1/2} = 2N/n$**.**

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*

$$\frac{T_{\rm sync}}{T_{\rm comp}} \to {\rm cte}$$

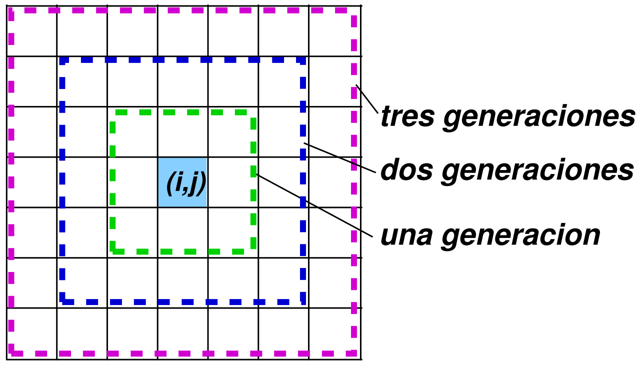
So that, there is no parameter that allows us to *increase efficiency* beyond the *limit value* $(1+s/b)^{-1}$. This limit value is dependent on *hardware*. It may be bad (low) for Fast Ethernet (100 Mbit/sec) whereas it may be good (high) for Gigabit Ethernet (1000 Mbit/sec). Or conversely, for a given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

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

so that now

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

If we want that $T_{\rm comm}/T_{\rm comp} < 0.1$ 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

$$T_{\mathrm{sync}} = (n/2) \times \textbf{(time-to-process-a-chunk)} = (n/2)\, kNN_c/s$$

so that

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

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 to 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

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

so that

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

is the same as before.

Improving Life scalability (cont.)

But the synchronization time will be

$$T_{\rm sync} = (n/2) \times \textbf{(time-to-process-a-chunk)} = (n/2)\, kN_c^2/s$$

so that

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

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

$$N_c < \sqrt{0.2/n}N$$

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

OPTIONAL Assignement 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 Assignement 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:

```
#define NBITCHAR 8
class bit_vector_t {
public:
  int nchar, N;
  unsigned char *buff;
  bit_vector_t(int N)
  : nchar(N/NBITCHAR+(N%NBITCHAR>0)) {
    buff = new unsigned char[nchar];
    for (int j=0; j<nchar; j++) buff[j] = 0;
  }
  int get(int j) {
    return (buff[j/NBITCHAR] & (1<<j%NBITCHAR))>0;
  }
  void set(int j,int val) {
    if (val) buff[j/NBITCHAR] |= (1<<j%NBITCHAR);
    else buff[j/NBITCHAR] &= ~(1<<j%NBITCHAR);
  }
};
```

OPTIONAL Assignement Nbr. 4 (cont.)

To create the vector

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

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

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

Finalle, to send the array:

```
char *v.buff : el buffer interno
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:

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

The Poisson equation

The Poisson equation

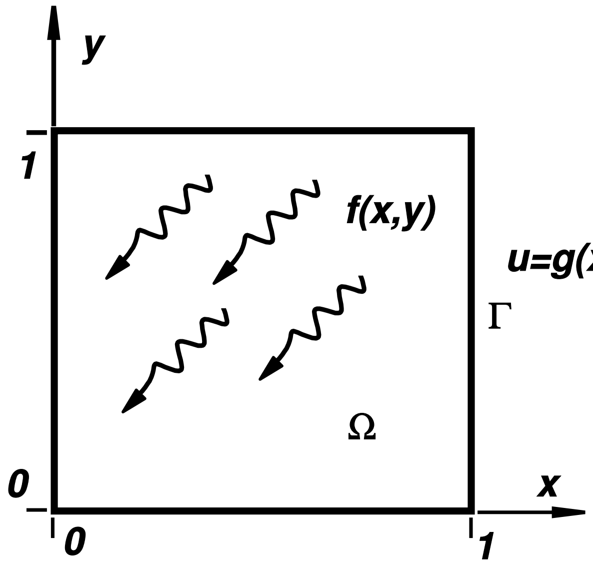
- It's an example of a Computational Mechanics problem with a finite difference scheme.
- Introduces the *virtual topology* concept.
- Discusses in detail a series of variations in the send/receive communication pattern to avoid *deadlock*.
- It's an introduction to the parallel implementation of matrix-vector 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 componenents.
- 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, precondicioners, etc...)

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

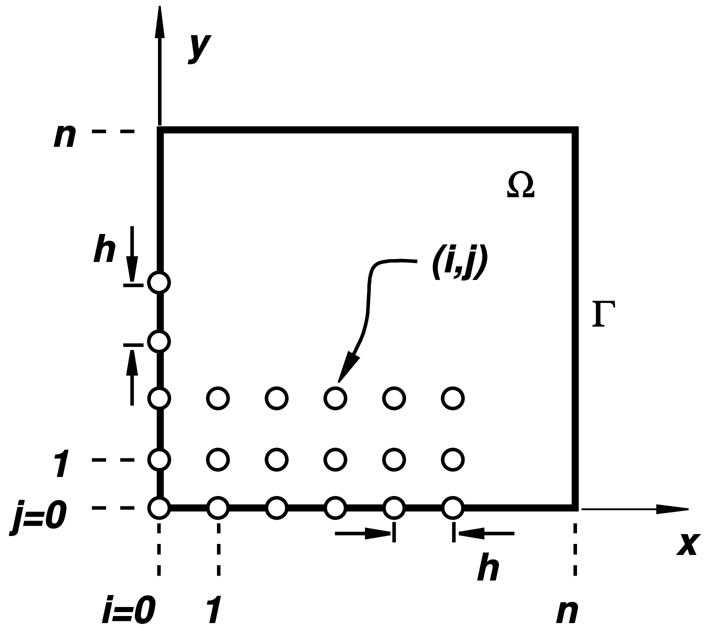
$$u(x,y) = g(x,y), \quad \textbf{in } \Gamma = \partial\Omega$$



The Poisson equation (cont.)

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

$$
\begin{aligned}
x_i &= i/n; \qquad & i &= 0, 1, \ldots, n \\
y_j &= j/n; \qquad & j &= 0, 1, \ldots, n \\
\mathbf{x}_{ij} &= (x_i, y_j)
\end{aligned}
$$

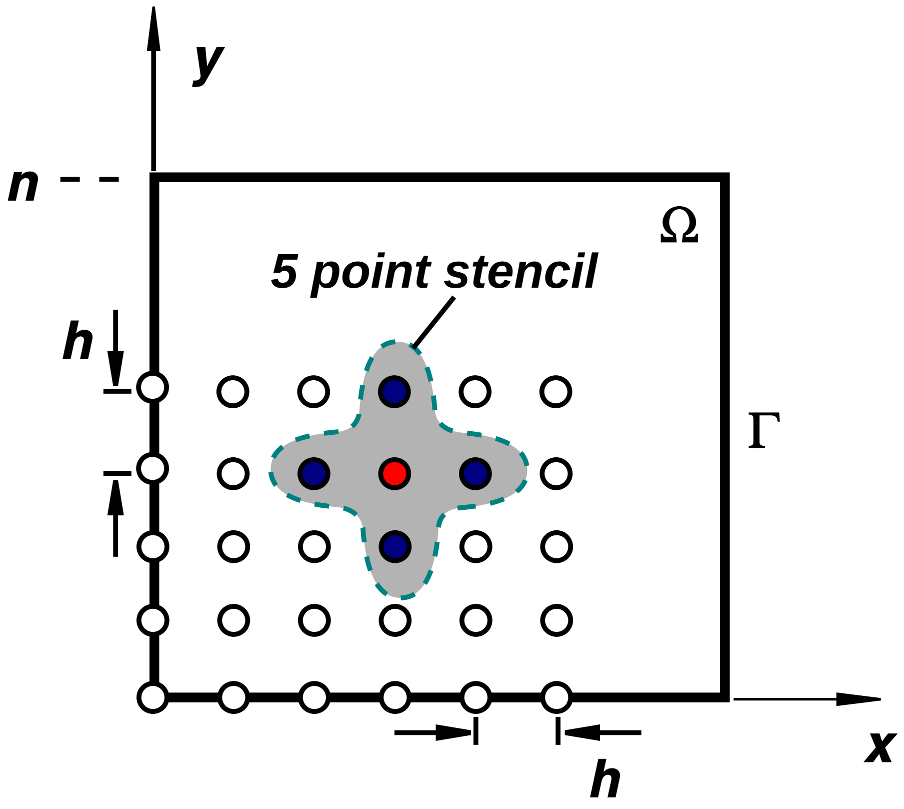


The Poisson equation (cont.)

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



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

The Poisson equation (cont.)

Reemplazando en le ec. de Poisson

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

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. Som equations include values at the boundary, but they are known, due to the Dirichlet kind of boundary condition.

$$\mathbf{Au} = \mathbf{f}$$

The Poisson equation (cont.)

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

$$\begin{aligned} \mathbf{A} = \mathbf{D} + \mathbf{A}', \quad \mathbf{D} &= \mathrm{diag}(\mathbf{A}) \\ (\mathbf{D}+\mathbf{A}')\mathbf{u} &= \mathbf{f} \\ \mathbf{D}\mathbf{u} &= \mathbf{f} - \mathbf{A}'\mathbf{u} \\ \mathbf{u} &= \mathbf{D}^{-1}(\mathbf{f} - \mathbf{A}'\mathbf{u}) \end{aligned}$$

So that, we can iterate in the following way

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

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.

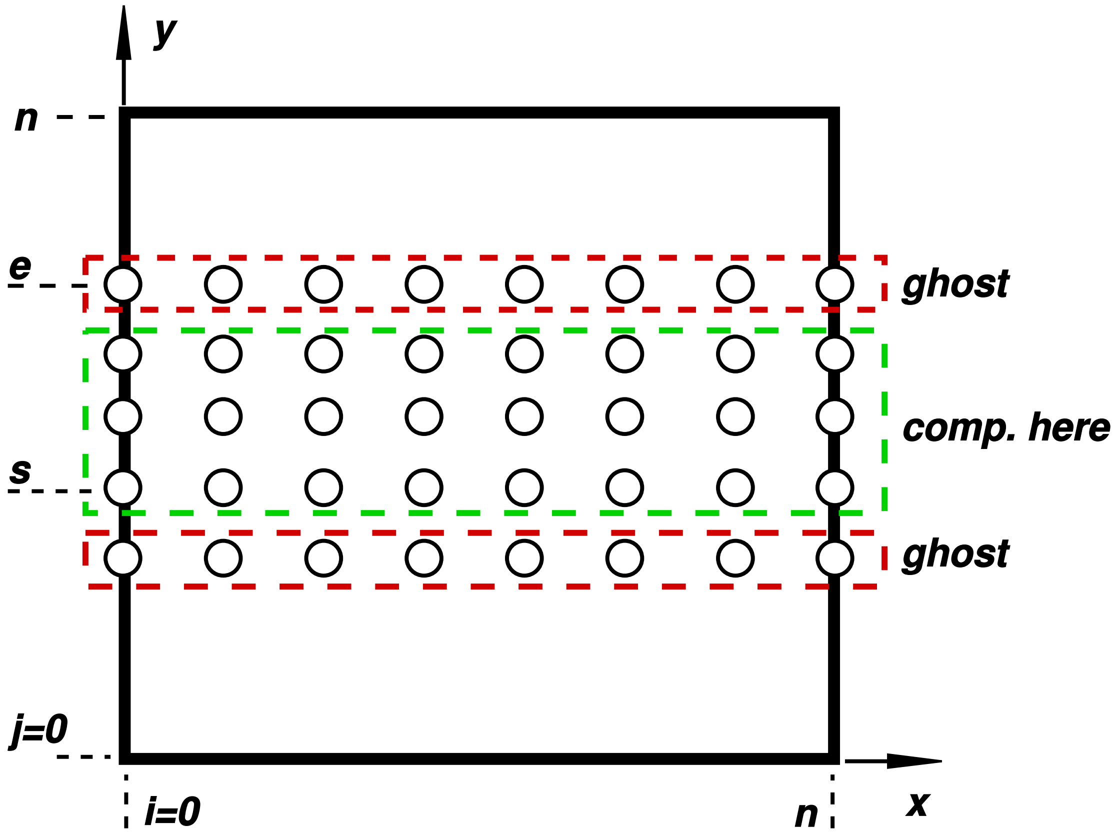
The Poisson equation (cont.)

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

```
int n=100, N=(n+1)^2;
double h=1.0/double(n);
double
  *u = new double[N],
  *unew = new double[N];
#define U(i,j) u((n+1)*(j)+(i))
#define UNEW(i,j) unew((n+1)*(j)+(i))

// Assign b.c. values
for (int i=0; i<n; i++) {
  U(i,0) = g(h*i,0.0);
  U(i,n) = g(h*i,1.0);
}

for (int j=0; j<n; j++) {
  U(0,j) = g(0.0,h*j);
  U(n,j) = g(1.0,h*j);
}

// Jacobi iteration
for (int j=1; j<n; j++) {
  for (int i=1; i<n; i++) {
    UNEW(i,j) = 0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)
                      +U(i,j+1)-h*h*F(i,j));
  }
}
```

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.

```
// define s,e ...
int rows_here = e-s+2;
double
  *u = new double[(n+1)*rows_here],
  *unew = new double[(n+1)*rows_here];
#define U(i,j) u((n+1)*(j-s+1)+i)
#define UNEW(i,j) unew((n+1)*(j-s+1)+i)

// Assign b.c. values ...

// Jacobi iteration
for (int j=s; j<e; j++) {
  for (int i=1; i<n; i++) {
    UNEW(i,j) = 0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)
                      +U(i,j+1)-h*h*F(i,j));
  }
}
```

Virtual Topologies

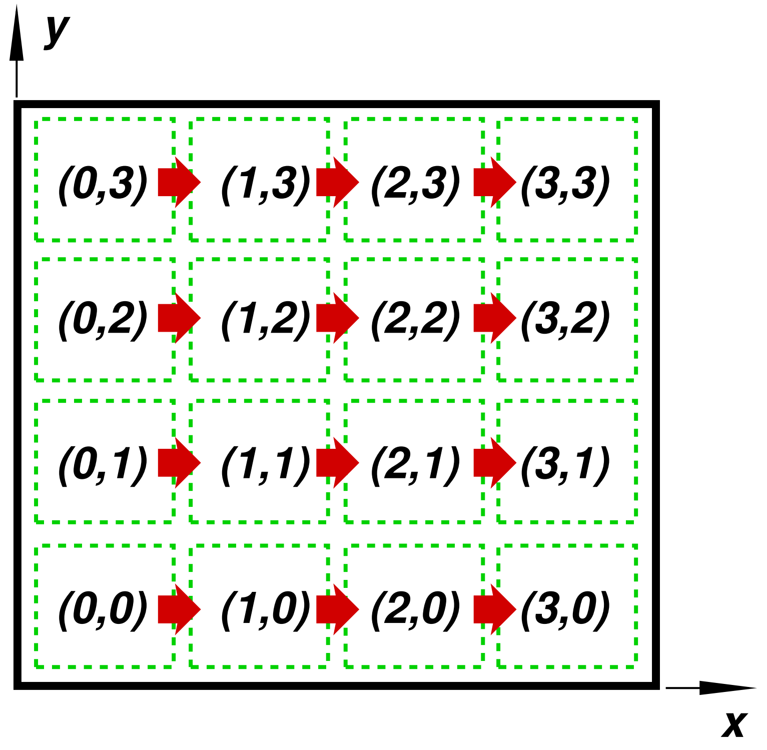
- ***Partiton:*** **we must assign a partition (i.e. a certain rank `[s,e)`) to each process.**
- **The optimal partition depends on how the network hardware is connected, i.e. the *network topology* (ring, star, toroidal...)**
- **For instance, if the underlying topology is a *ring* then the best partitioning 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 estabilishing 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

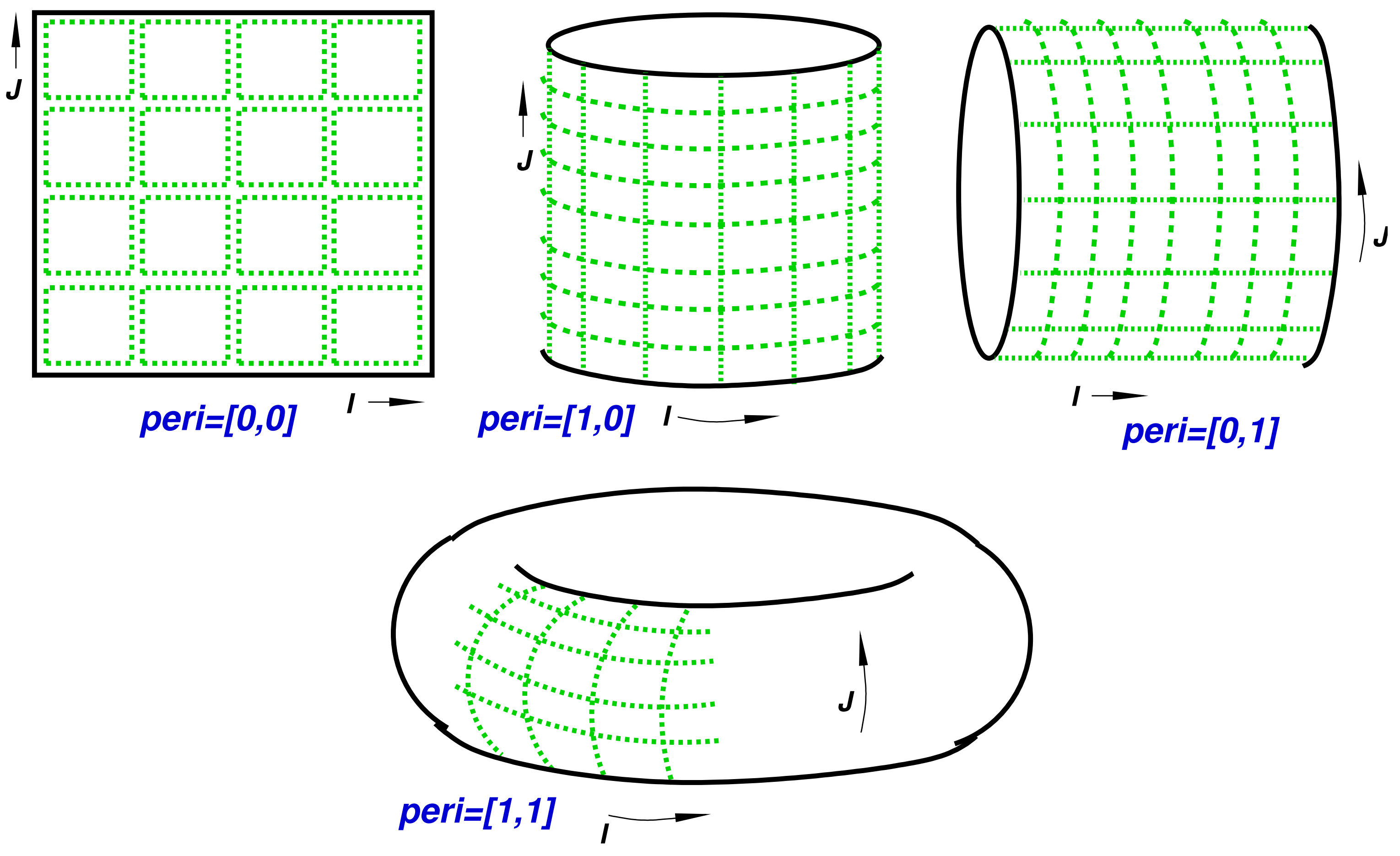
```
MPI_Cart_create(MPI_Comm comm_old, int ndims,
    int *dims, int *periods, int reorder, MPI_Comm *new_comm);
```

`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

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

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

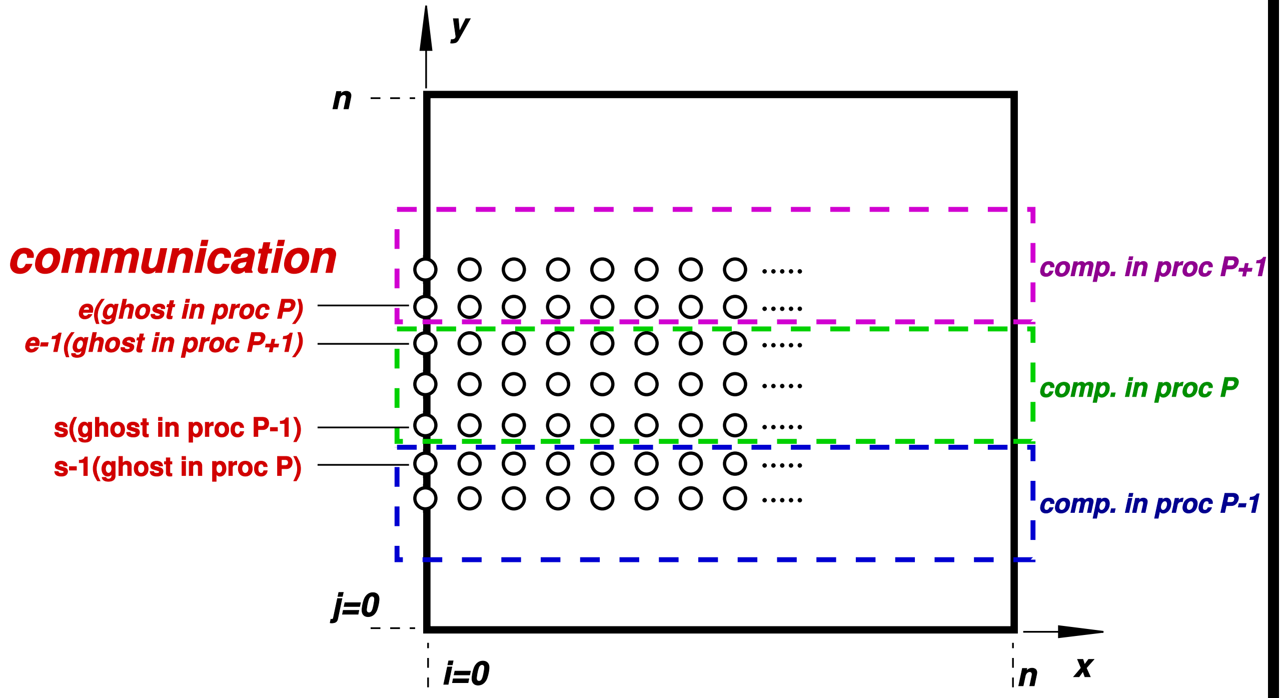
- User can get only the process coordinate tuple within the topology:

```
int MPI_Cart_coords(MPI_Comm comm,int rank,
      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`**



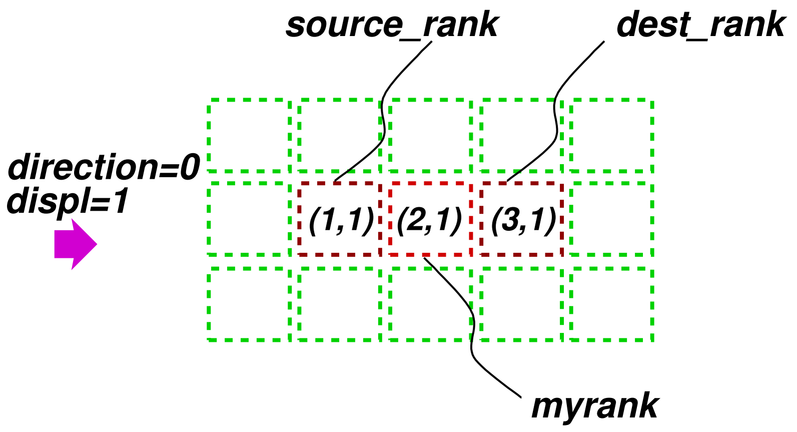
Virtual Topologies (cont.)

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:

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

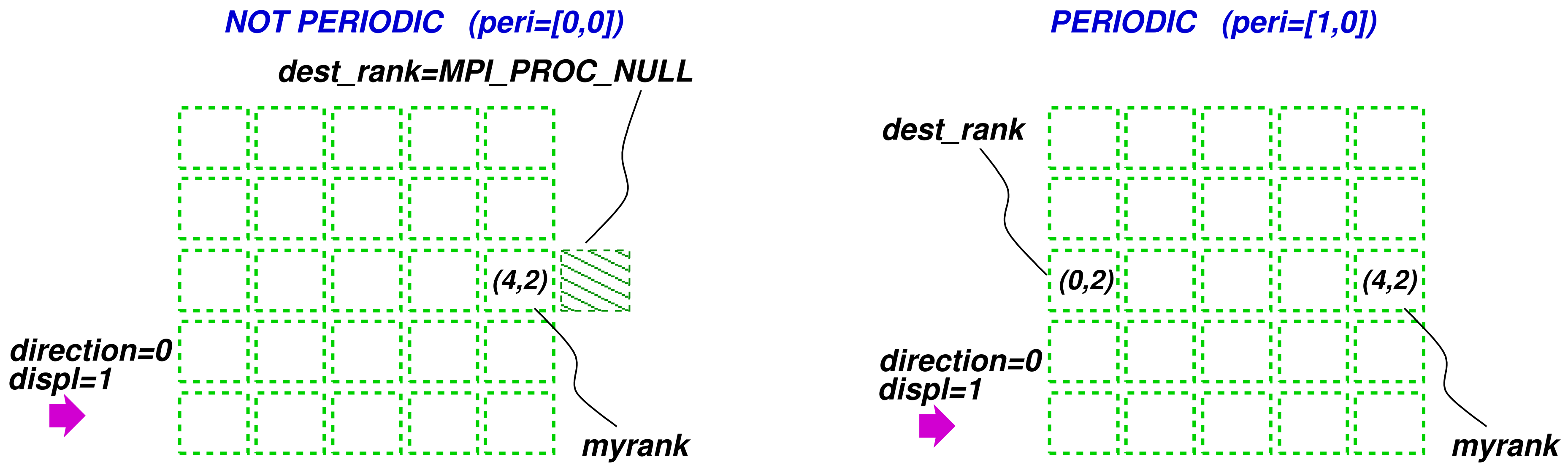
For instance, with a 2D topology, in order to do a *shift* in the horizontal direction.

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



Virtual Topologies (cont.)

What happens at the boundary processors? For instante 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

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

is equivalent to

```
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`, the we can compute the row range to be computed in this processor `[s,e)` as

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

If it is not a multiple then

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

Virtual Topologies (cont.)

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

```
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

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

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

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

Virtual Topologies (cont.)

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

```
void decomp(int n,vector<double> &weights,
            vector<int> &nrows) {
  int size = weights.size();
  nrows.resize(size);
  double sum_w = 0., carry=0., a;
  for (int p=0; p<size; p++) sum_w += weights[p];
  int j = 0;
  double tol = 1e-8;
  for (int p=0; p<size; p++) {
    double w = weights[p]/sum_w;
    a = w*n + carry + tol;
    nrows[p] = int(floor(a));
    carry = a - nrows[p] - tol;
    j += nrows[p];
  }
  assert(j==n);
  assert(fabs(carry) < tol);
}
```

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).

```
void exchng1(double *a,int n,int s,int e,
             MPI_Comm comm1d,int nbrbot,int nbrtop) {
  MPI_Status status;
#define ROW(j) (&a[(j-s+1)*(n+1)])

  // Exchange top row
  MPI_Send(ROW(e-1),n+1,MPI_DOUBLE,nbrtop,0,comm1d);
  MPI_Recv(ROW(s-1),n+1,MPI_DOUBLE,nbrbot,0,comm1d,&status);

  // Exchange top row
  MPI_Send(ROW(s),n+1,MPI_DOUBLE,nbrbot,0,comm1d);
  MPI_Recv(ROW(e),n+1,MPI_DOUBLE,nbrtop,0,comm1d,&status);
}
```

Poisson's eq. Communication strategy (cont.)

But the communication is slow (see section 200).

```
void exchng2(double *a,int n,int s,int e,
             MPI_Comm comm1d,int nbrbot,int nbrtop) {
  MPI_Status status;
#define ROW(j) (&a[(j-s+1)*(n+1)])

  // Exchange top row
  MPI_Sendrecv(ROW(e-1),n+1,MPI_DOUBLE,nbrtop,0,
               ROW(s-1),n+1,MPI_DOUBLE,nbrbot,0,
               comm1d,&status);

  // Exchange top row
  MPI_Sendrecv(ROW(s),n+1,MPI_DOUBLE,nbrbot,0,
               ROW(e),n+1,MPI_DOUBLE,nbrtop,0,
               comm1d,&status);
}
```

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.)

```
int main() {
  int n=10;

  // Get MPI info
  int size,rank;
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int periods = 0;
  MPI_Comm comm1d;
  MPI_Cart_create(MPI_COMM_WORLD,1,&size,&periods,1,&comm1d);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  int direction=1, nbrbot, nbrtop;
  MPI_Cart_shift(comm1d,direction,1,&nbrbot,&nbrtop);

  int s,e;
  MPE_Decomp1d(n-1,size,rank,&s,&e);

  int
    rows_here = e-s+2,
    points_here = rows_here*(n+1);
  double
    *tmp,
    *u1 = new double[points_here],
    *u2 = new double[points_here],
    *f = new double[points_here];
#define U1(i,j) u1((n+1)*(j-s+1)+i)
```

```
#define U2(i,j) u2((n+1)*(j-s+1)+i)
#define F(i,j) f((n+1)*(j-s+1)+i)

  // Assign bdry conditions to 'u1' and 'u2' . . .
  // Define 'f' . . .
  // Initialize 'u1' and 'u2'

  // Do computations
  int iter, itmax = 1000;
  for (iter=0; iter<itmax; iter++) {
    // Communicate ghost values
    exchng2(u1,n,s,e,comm1d,nbrbot,nbrtop);
    // Compute new 'u' values from Jacobi iteration
    sweep(u1,u2,f,n,s,e);

    // Compute difference between
    // 'u1' (u^k) and 'u2' (u^{k+1})
    double diff_here2 = diff(u1,u2,n,s,e);
    double diff;
    MPI_Allreduce(&diff_here2,&diff,1,MPI_DOUBLE,
                  MPI_SUM,MPI_COMM_WORLD);
    diff = sqrt(diff);
    printf("iter %d, diff %d\n",iter,diff);

    // Swap pointers
    tmp=u1; u1=u2; u2=tmp;

    if (diff<1e-5) break;
  }

  printf("Converged in %d iterations\n",iter);

  delete[] u1;
  delete[] u2;
  delete[] f;
}
```

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

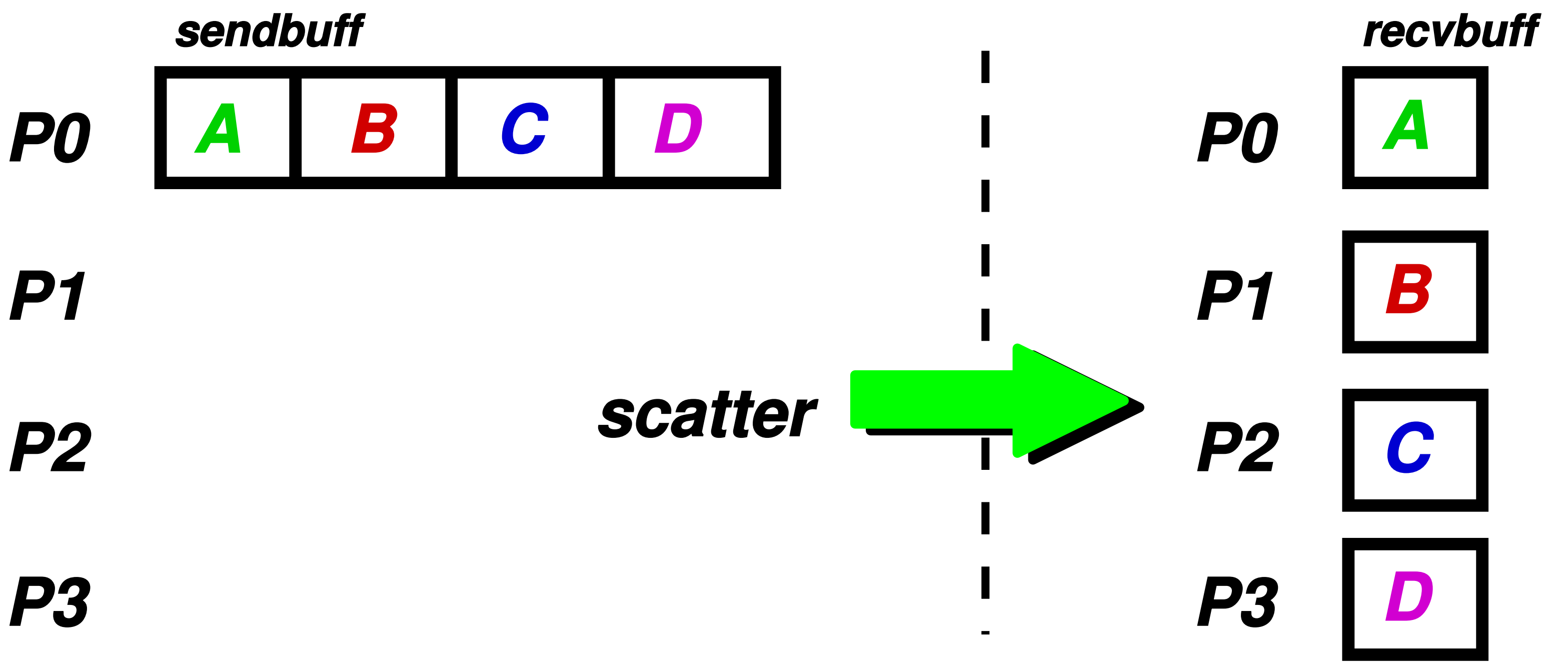
- `MPI_Scatter()`, `MPI_Gather()`
- `MPI_Allgather()`
- `MPI_Alltoall()`

And their vectorized (varible length per processor) versions

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

Scatter operations

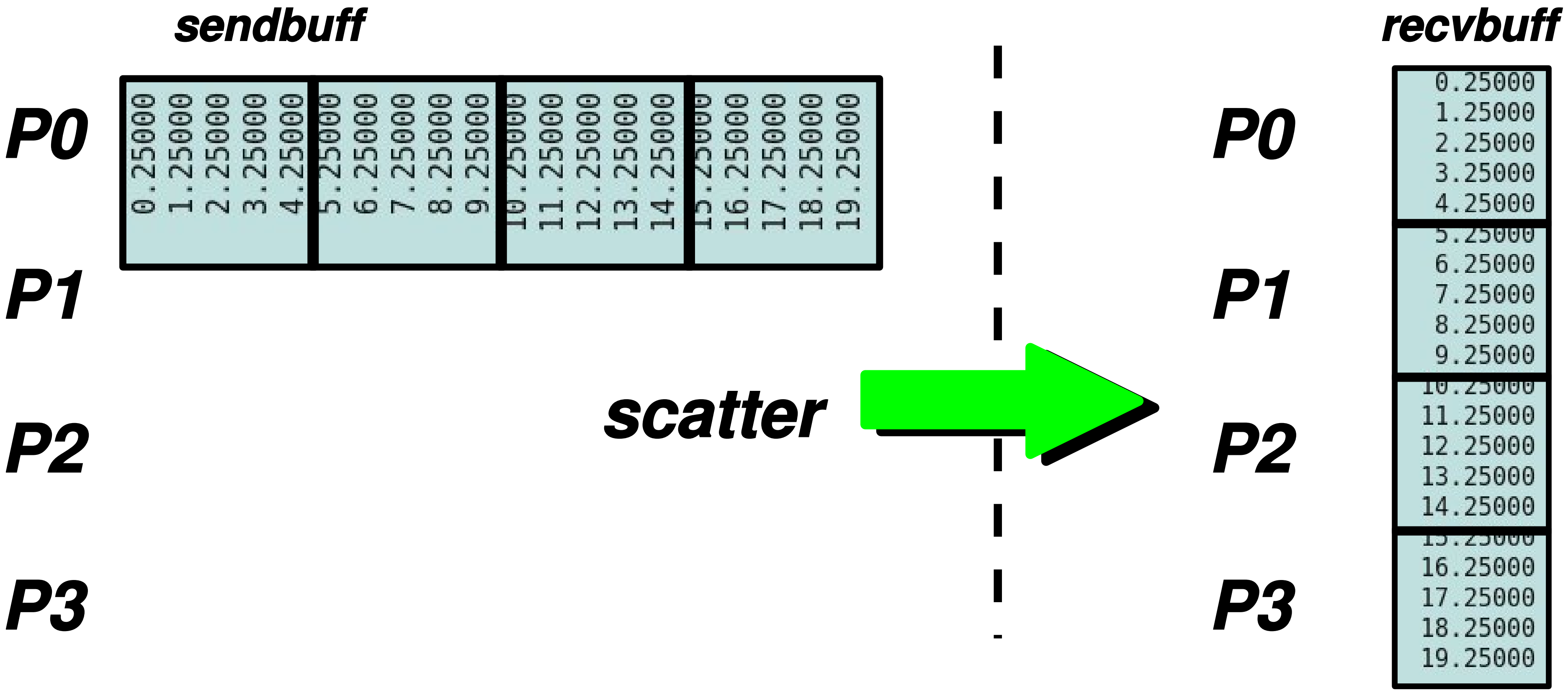
Sends a certain amount of data of the same size and type to the other processes (as in `MPI_Bcast()`, but the data to be sent is not the same to all processes).



Scatter operations (cont.)

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

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int N = 5;              // Nbr of elements to send to each processor
  double *sbuff = NULL;
  if (!myrank) {
    sbuff = new double[N*size]; // send buffer only in master
    for (int j=0; j<N*size; j++) sbuff[j] = j+0.25; // fills ‘sbuff’
  }
  double *rbuff = new double[N]; // receive buffer in all procs

  MPI_Scatter(sbuff,N,MPI_DOUBLE,
              rbuff,N,MPI_DOUBLE,0,MPI_COMM_WORLD);

  for (int j=0; j<N; j++)
    printf("[%d] %d -> %f\n",myrank,j,rbuff[j]);
  MPI_Finalize();
  if (!myrank) delete[] sbuff;
  delete[] rbuff;
}
```

Scatter operations (cont.)

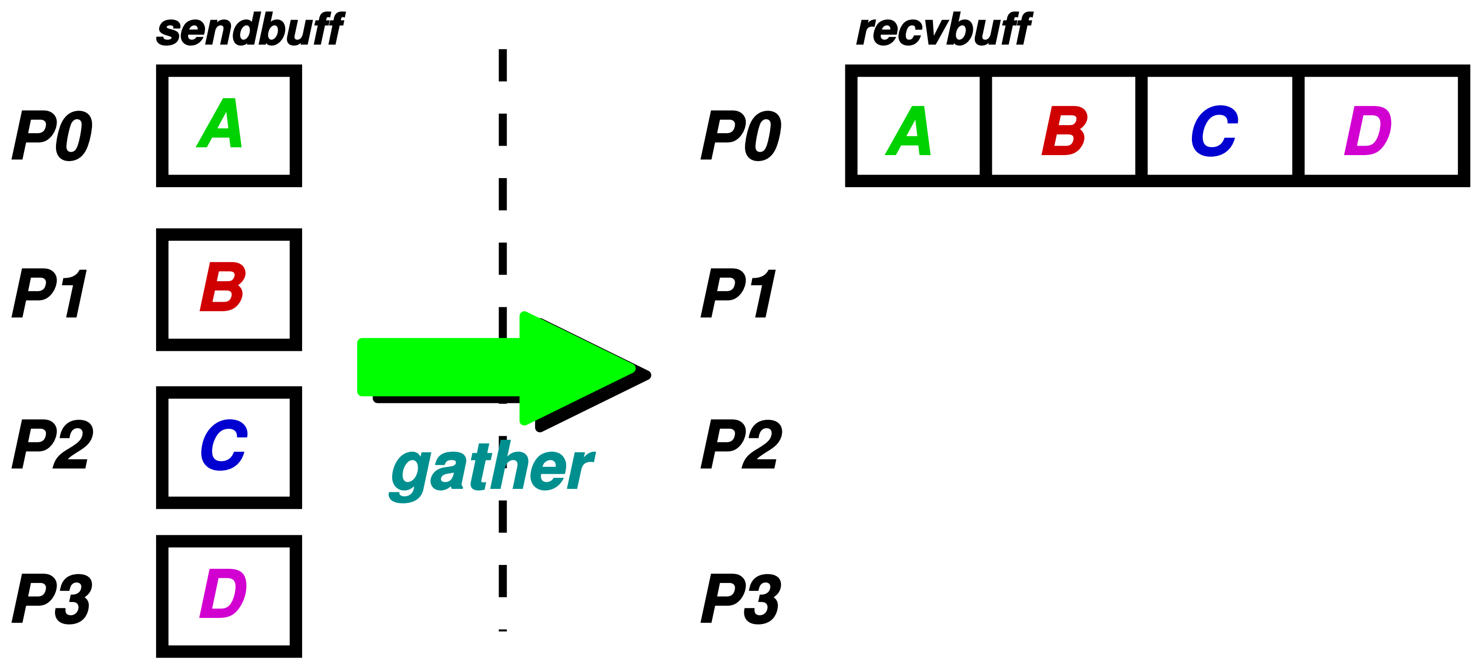


Scatter operations (cont.)

```
[mstorti@spider example]$ mpirun -np 4 -machinefile \
                                 machi.dat scatter2.bin
[0] 0 -> 0.250000
[0] 1 -> 1.250000
[0] 2 -> 2.250000
[0] 3 -> 3.250000
[0] 4 -> 4.250000
[2] 0 -> 10.250000
[2] 1 -> 11.250000
[2] 2 -> 12.250000
[2] 3 -> 13.250000
[2] 4 -> 14.250000
[3] 0 -> 15.250000
[3] 1 -> 16.250000
[3] 2 -> 17.250000
[3] 3 -> 18.250000
[3] 4 -> 19.250000
[1] 0 -> 5.250000
[1] 1 -> 6.250000
[1] 2 -> 7.250000
[1] 3 -> 8.250000
[1] 4 -> 9.250000
[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.)

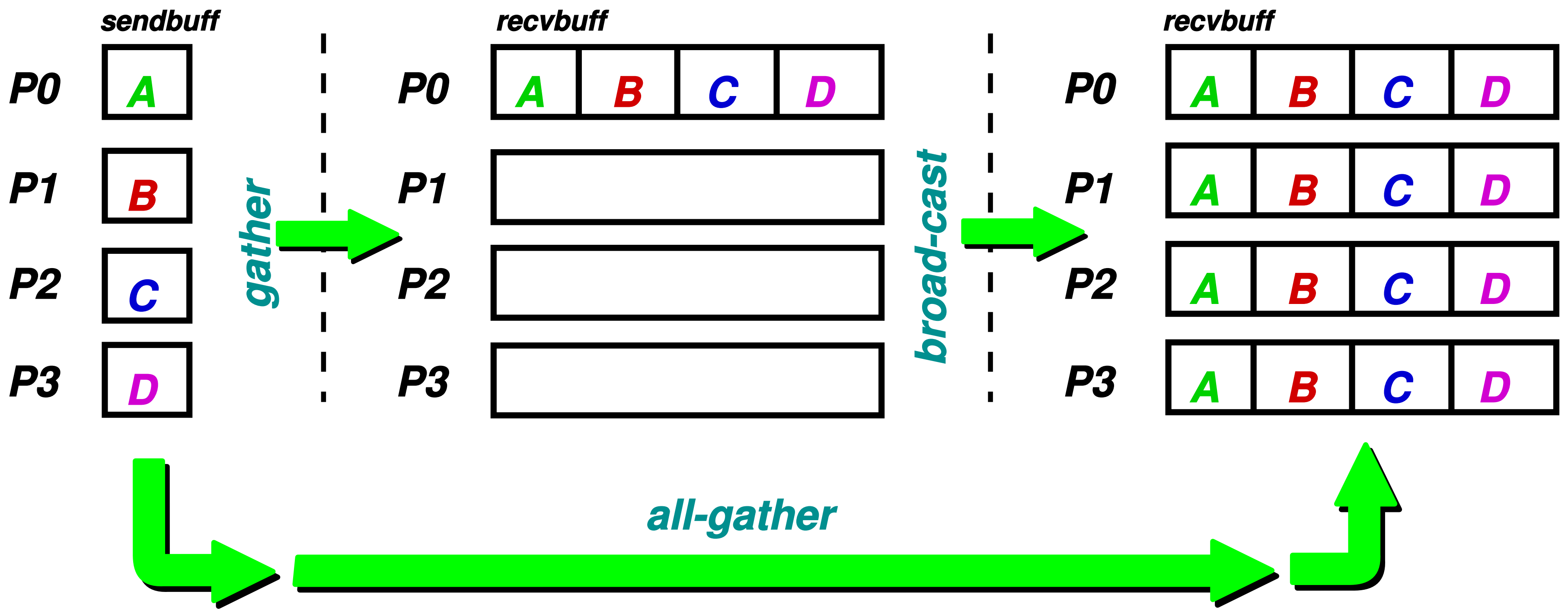
```
#include <mpi.h>
#include <cstdio>

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int N = 5;      // Nbr of elements to send to each processor
  double *sbuff = new double[N]; // send buffer in all procs
  for (int j=0; j<N; j++) sbuff[j] = myrank*1000.0+j;

  double *rbuff = NULL;
  if (!myrank) {
    rbuff = new double[N*size]; // recv buffer only in master
  }

  MPI_Gather(sbuff,N,MPI_DOUBLE,
             rbuff,N,MPI_DOUBLE,0,MPI_COMM_WORLD);

  if (!myrank)
    for (int j=0; j<N*size; j++)
      printf("%d -> %f\n",j,rbuff[j]);
  MPI_Finalize();

  delete[] sbuff;
  if (!myrank) delete[] rbuff;
}
```

Gather operations (cont.)

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

All-gather operation

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



All-gather operation (cont.)

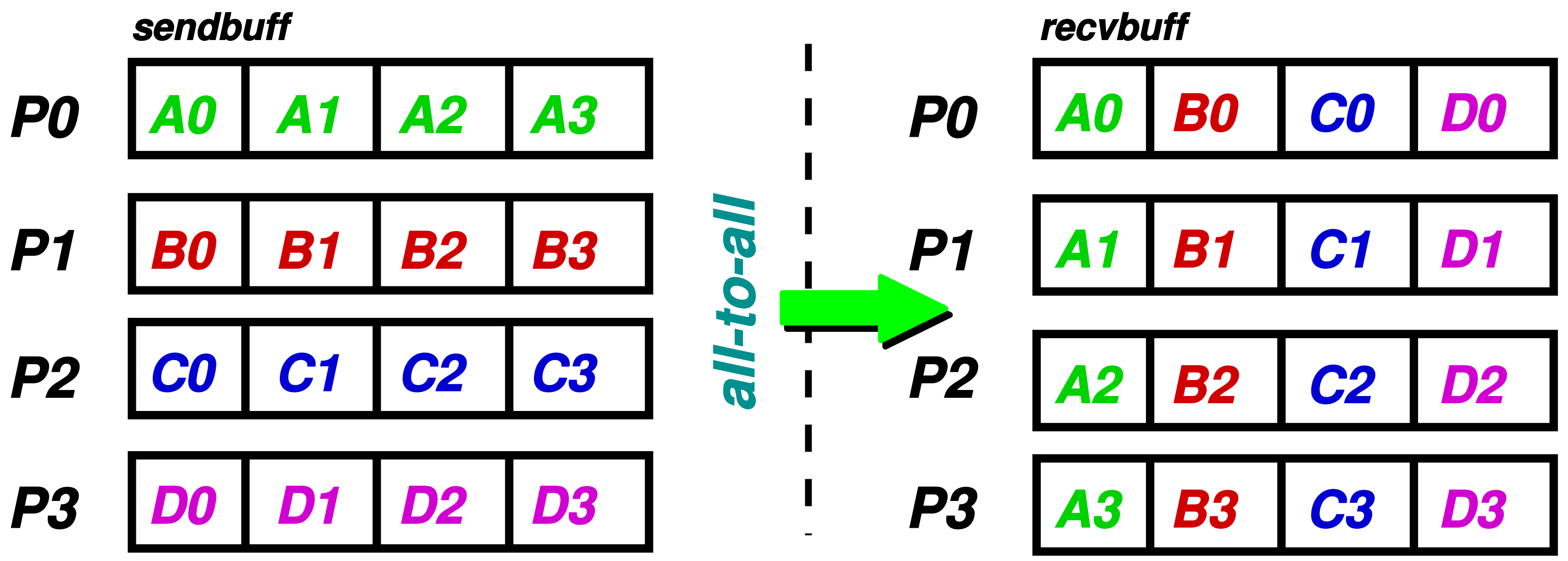
```
#include <mpi.h>
#include <cstdio>

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int N = 3;      // Nbr of elements to send to each processor
  double *sbuff = new double[N]; // send buffer in all procs
  for (int j=0; j<N; j++) sbuff[j] = myrank*1000.0+j;

  double *rbuff = new double[N*size]; // receive buffer in all procs

  MPI_Allgather(sbuff,N,MPI_DOUBLE,
                rbuff,N,MPI_DOUBLE,MPI_COMM_WORLD);

  for (int j=0; j<N*size; j++)
    printf("[%d] %d -> %f\n",myrank,j,rbuff[j]);
  MPI_Finalize();

  delete[] sbuff;
  delete[] rbuff;
}
```

All-gather operation (cont.)

```
[mstorti@spider example]$ mpirun -np 3 \
                -machinefile machi.dat allgather.bin
[0] 0 -> 0.000000
[0] 1 -> 1.000000
[0] 2 -> 2.000000
[0] 3 -> 1000.000000
[0] 4 -> 1001.000000
[0] 5 -> 1002.000000
[0] 6 -> 2000.000000
[0] 7 -> 2001.000000
[0] 8 -> 2002.000000
[1] 0 -> 0.000000
[1] 1 -> 1.000000
[1] 2 -> 2.000000
[1] 3 -> 1000.000000
[1] 4 -> 1001.000000
[1] 5 -> 1002.000000
[1] 6 -> 2000.000000
[1] 7 -> 2001.000000
[1] 8 -> 2002.000000
[2] 0 -> 0.000000
[2] 1 -> 1.000000
[2] 2 -> 2.000000
[2] 3 -> 1000.000000
[2] 4 -> 1001.000000
[2] 5 -> 1002.000000
[2] 6 -> 2000.000000
[2] 7 -> 2001.000000
[2] 8 -> 2002.000000
[mstorti@spider example]$
```

All-to-all operation

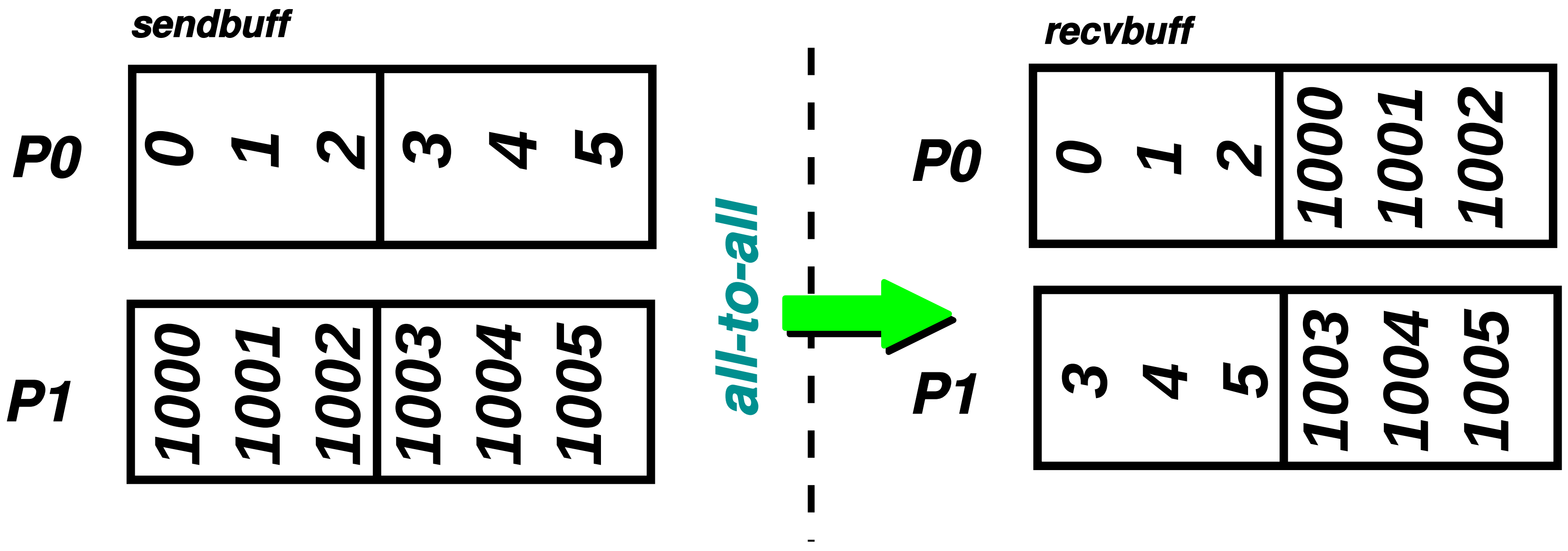
- Its conceptually equivalent to a scatter from $P0$ followed by a scatter from $P1$, etc...
- Or either a gather to $P0$, followed by a gather to $P1$, and so on...



All-to-all operation (cont.)

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

int main(int argc, char **argv) {
  MPI_Init(&argc,&argv);
  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int N = 3;      // Nbr of elements to send to each processor
  double *sbuff = new double[size*N]; // send buffer in all procs
  for (int j=0; j<size*N; j++) sbuff[j] = myrank*1000.0+j;

  double *rbuff = new double[N*size]; // receive buffer in all procs

  MPI_Alltoall(sbuff,N,MPI_DOUBLE,
               rbuff,N,MPI_DOUBLE,MPI_COMM_WORLD);

  for (int j=0; j<N*size; j++)
    printf("[%d] %d -> %f\n",myrank,j,rbuff[j]);
  MPI_Finalize();

  delete[] sbuff;
  delete[] rbuff;
}
```

All-to-all operation (cont.)

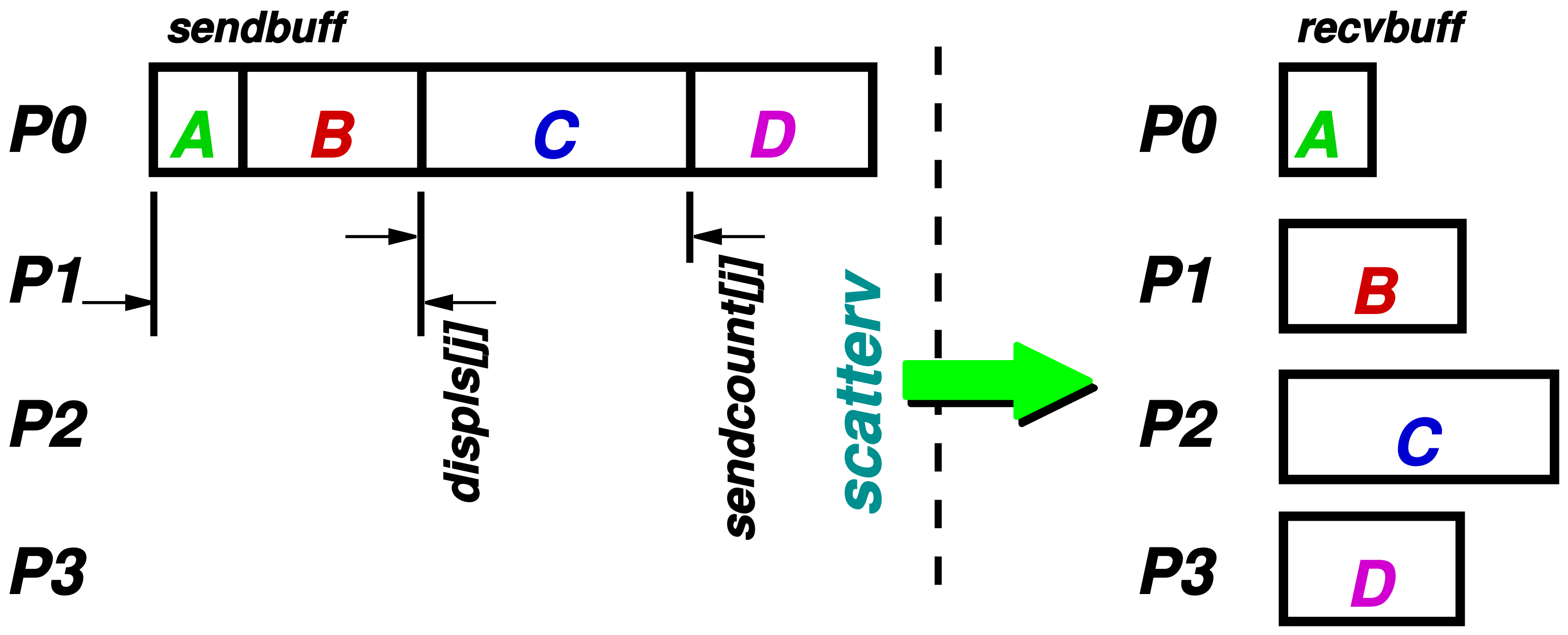


All-to-all operation (cont.)

```
[mstorti@spider example]$ mpirun -np 2 \
                               -machinefile machi.dat alltoall.bin
[0] 0 -> 0.000000
[0] 1 -> 1.000000
[0] 2 -> 2.000000
[0] 3 -> 1000.000000
[0] 4 -> 1001.000000
[0] 5 -> 1002.000000
[1] 0 -> 3.000000
[1] 1 -> 4.000000
[1] 2 -> 5.000000
[1] 3 -> 1003.000000
[1] 4 -> 1004.000000
[1] 5 -> 1005.000000
[mstorti@spider example]$
```

Vector scatter (variable length)

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



Vector scatter (variable length) (cont.)

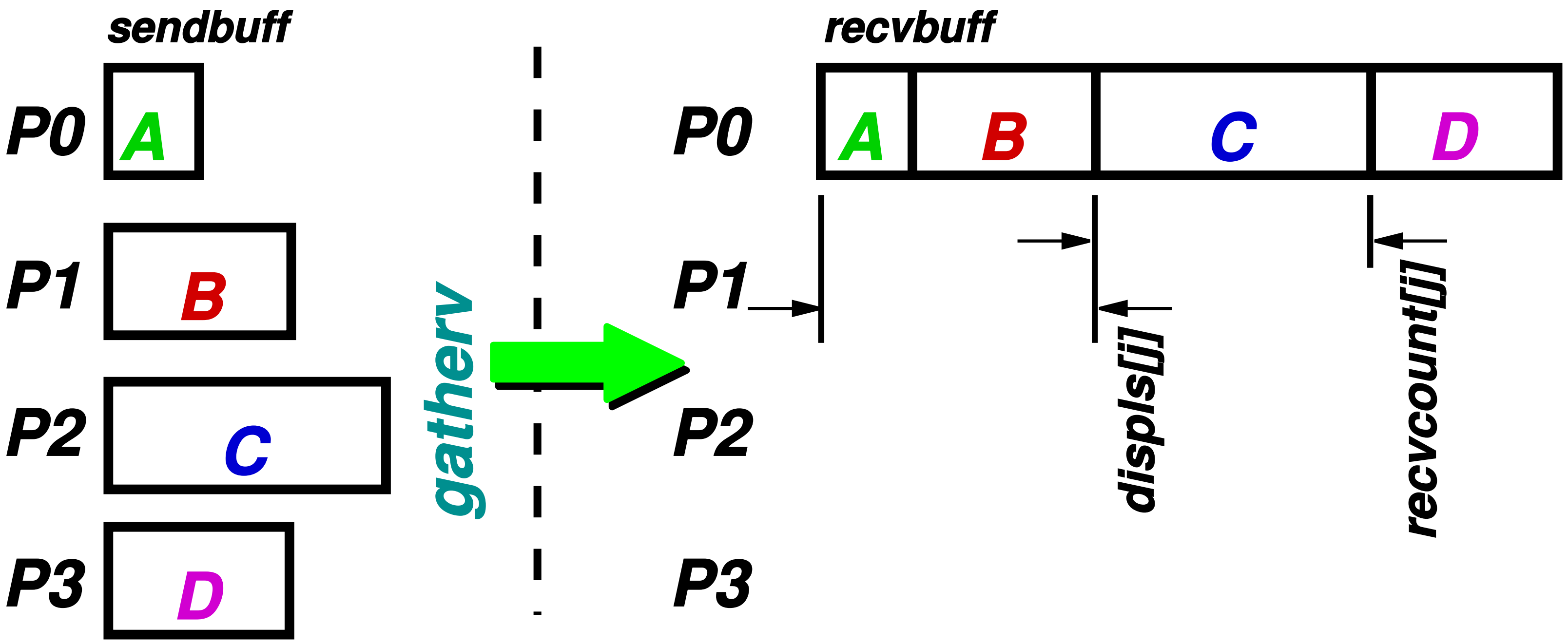
```
int N = size*(size+1)/2;
double *sbuff = NULL;
int *sendcnts = NULL;
int *displs = NULL;
if (!myrank) {
  sbuff = new double[N]; // send buffer only in master
  for (int j=0; j<N; j++) sbuff[j] = j; // fills 'sbuff'

  sendcnts = new int[size];
  displs = new int[size];
  for (int j=0; j<size; j++) sendcnts[j] = (j+1);
  displs[0]=0;
  for (int j=1; j<size; j++)
    displs[j] = displs[j-1] + sendcnts[j-1];
}

// receive buffer in all procs
double *rbuff = new double[myrank+1];

MPI_Scatterv(sbuff,sendcnts,displs,MPI_DOUBLE,
             rbuff,myrank+1,MPI_DOUBLE,0,MPI_COMM_WORLD);

for (int j=0; j<myrank+1; j++)
  printf("[%d] %d -> %f\n",myrank,j,rbuff[j]);
```

Vector scatter (variable length) (cont.)

```
[mstorti@spider example]$ mpirun -np 4 \
                    -machinefile machi.dat scatterv.bin
[0] 0 -> 0.000000
[3] 0 -> 6.000000
[3] 1 -> 7.000000
[3] 2 -> 8.000000
[3] 3 -> 9.000000
[1] 0 -> 1.000000
[1] 1 -> 2.000000
[2] 0 -> 3.000000
[2] 1 -> 4.000000
[2] 2 -> 5.000000
[mstorti@spider example]$
```

Gatherv operation

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



Gatherv operation (cont.)

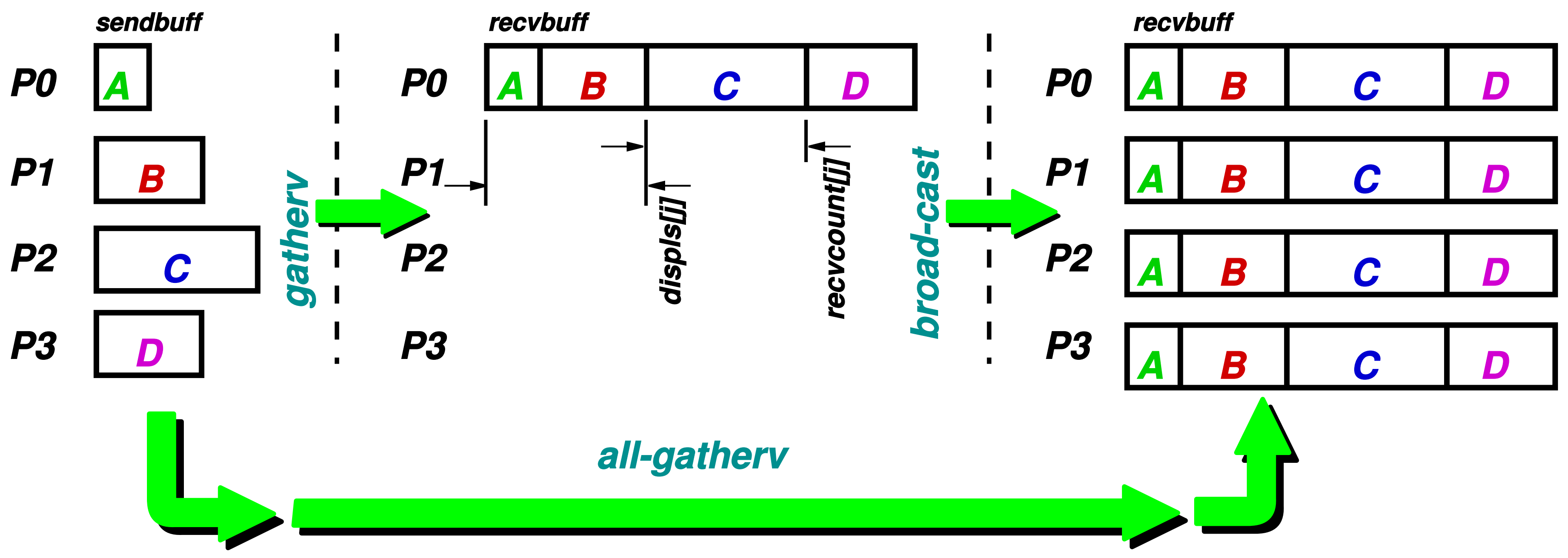
```
int sendcnt = myrank+1; // send buffer in all
double *sbuff = new double[myrank+1];
for (int j=0; j<sendcnt; j++)
  sbuff[j] = myrank*1000+j;

int rsize = size*(size+1)/2;
int *recvcnts = NULL;
int *displs = NULL;
double *rbuff = NULL;
if (!myrank) {
  // receive buffer and ptrs only in master
  rbuff = new double[rsize]; // recv buffer only in master
  recvcnts = new int[size];
  displs = new int[size];
  for (int j=0; j<size; j++) recvcnts[j] = (j+1);
  displs[0]=0;
  for (int j=1; j<size; j++)
    displs[j] = displs[j-1] + recvcnts[j-1];
}

MPI_Gatherv(sbuff,sendcnt,MPI_DOUBLE,
            rbuff,recvcnts,displs,MPI_DOUBLE,
            0,MPI_COMM_WORLD);
```

Gatherv operation (cont.)

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

Allgatherv operation

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



Allgatherv operation (cont.)

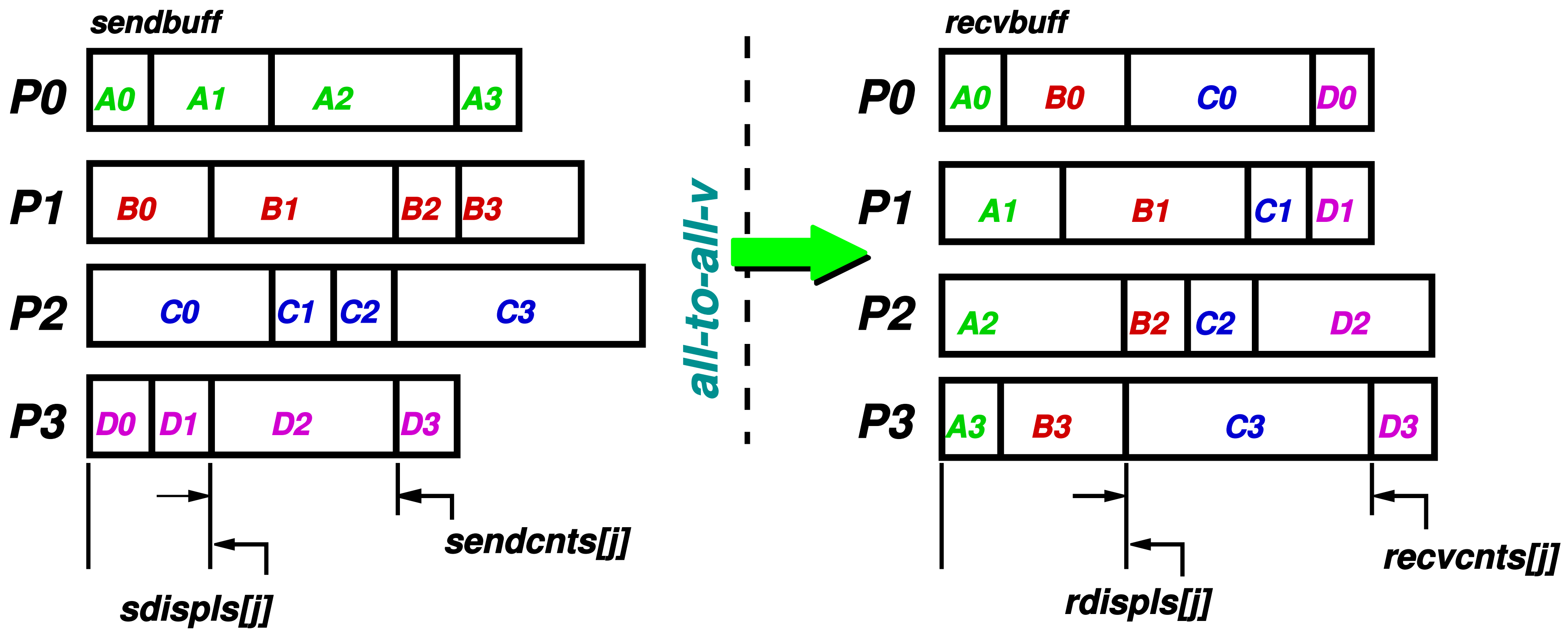
```
int sendcnt = myrank+1; // send buffer in all
double *sbuff = new double[myrank+1];
for (int j=0; j<sendcnt; j++)
  sbuff[j] = myrank*1000+j;

// receive buffer and ptrs in all
int rsize = size*(size+1)/2;
double *rbuff = new double[rsize];
int *recvcnts = new int[size];
int *displs = new int[size];
for (int j=0; j<size; j++) recvcnts[j] = (j+1);
displs[0]=0;
for (int j=1; j<size; j++)
  displs[j] = displs[j-1] + recvcnts[j-1];

MPI_Allgatherv(sbuff,sendcnt,MPI_DOUBLE,
            rbuff,recvcnts,displs,MPI_DOUBLE,
            MPI_COMM_WORLD);
```

Allgatherv operation (cont.)

```
[mstorti@spider example]$ mpirun -np 3 \
                -machinefile machi.dat allgatherv.bin
[0] 0 -> 0.000000
[0] 1 -> 1000.000000
[0] 2 -> 1001.000000
[0] 3 -> 2000.000000
[0] 4 -> 2001.000000
[0] 5 -> 2002.000000
[1] 0 -> 0.000000
[1] 1 -> 1000.000000
[1] 2 -> 1001.000000
[1] 3 -> 2000.000000
[1] 4 -> 2001.000000
[1] 5 -> 2002.000000
[2] 0 -> 0.000000
[2] 1 -> 1000.000000
[2] 2 -> 1001.000000
[2] 3 -> 2000.000000
[2] 4 -> 2001.000000
[2] 5 -> 2002.000000
[mstorti@spider example]$
```

All-to-all-v operation

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



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

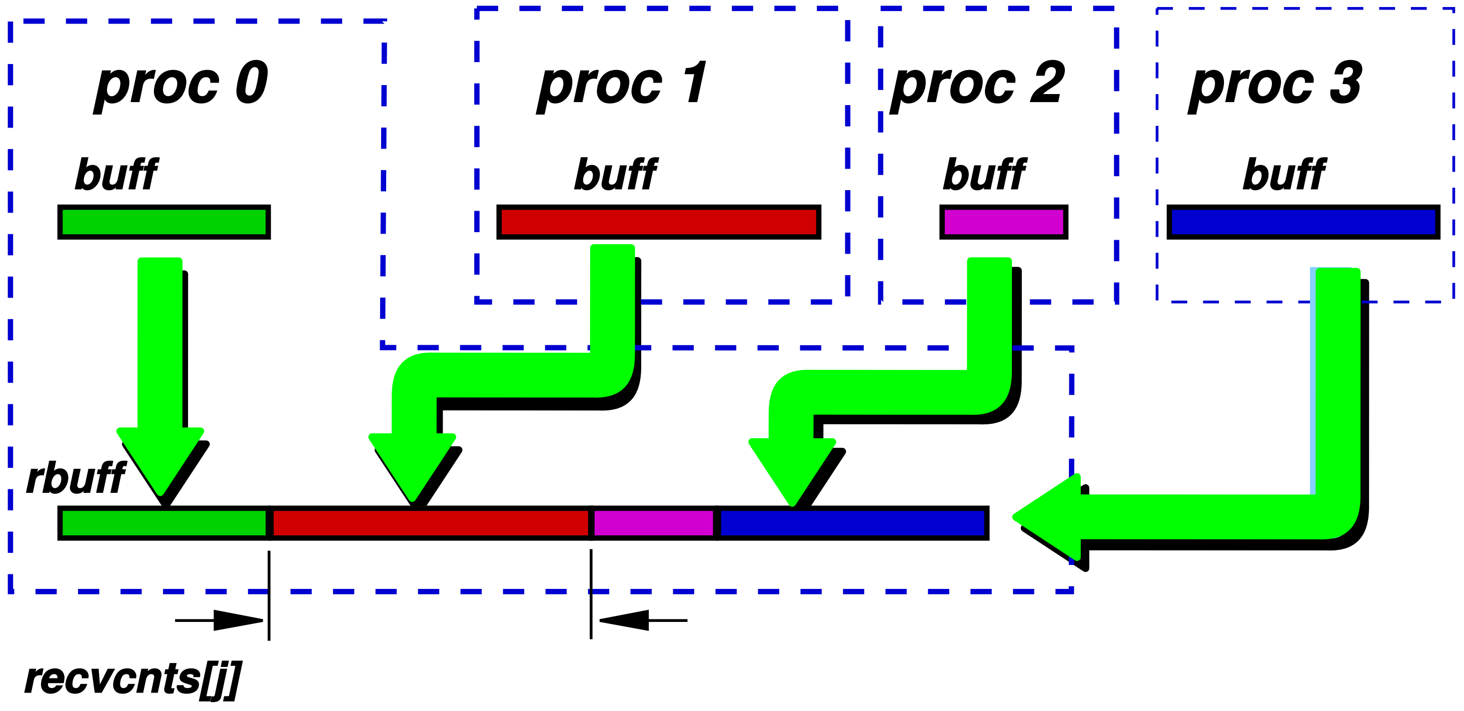
```
// vectorized send buffer in all
int ssize = (myrank+1)*size;
double *sbuff = new double[ssize];
int *sendcnts = new int[size];
int *sdispls = new int[size];
for (int j=0; j<ssize; j++)
  sbuff[j] = myrank*1000+j;
for (int j=0; j<size; j++) sendcnts[j] = (myrank+1);
sdispls[0]=0;
for (int j=1; j<size; j++)
  sdispls[j] = sdispls[j-1] + sendcnts[j-1];

// vectorized receive buffer and ptrs in all
int rsize = size*(size+1)/2;
double *rbuff = new double[rsize];
int *recvcnts = new int[size];
int *rdispls = new int[size];
for (int j=0; j<size; j++) recvcnts[j] = (j+1);
rdispls[0]=0;
for (int j=1; j<size; j++)
  rdispls[j] = rdispls[j-1] + recvcnts[j-1];

MPI_Alltoallv(sbuff,sendcnts,sdispls,MPI_DOUBLE,
              rbuff,recvcnts,rdispls,MPI_DOUBLE,
              MPI_COMM_WORLD);
```

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

```
[mstorti@spider example]$ mpirun -np 3 \
                  -machinefile machi.dat alltoallv.bin
[0] 0 -> 0.000000
[0] 1 -> 1000.000000
[0] 2 -> 1001.000000
[0] 3 -> 2000.000000
[0] 4 -> 2001.000000
[0] 5 -> 2002.000000
[1] 0 -> 1.000000
[1] 1 -> 1002.000000
[1] 2 -> 1003.000000
[1] 3 -> 2003.000000
[1] 4 -> 2004.000000
[1] 5 -> 2005.000000
[2] 0 -> 2.000000
[2] 1 -> 1004.000000
[2] 2 -> 1005.000000
[2] 3 -> 2006.000000
[2] 4 -> 2007.000000
[2] 5 -> 2008.000000
[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.)

```
void print_par(vector<int> &buff,const char *s=NULL) {

  int sendcnt = buff.size();
  vector<int> recvcnts(size);
  MPI_Gather(sendcnt,...,recvcnts,...);

  int rsize = /* sum of recvcnts[] */;
  vector<int> buff(rsize);

  vector<int> displs;
  displs = /* cum-sum of recvcnts[] */;

  MPI_Gatherv(buff,sendcnt....,
              rbuff,rcvcnts,displs,...);
  if (!myrank) {
    for (int rank=0; rank<size; rank++) {
      // print elements belonging to
      // processor 'rank' ...
    }
  }
}
```

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...
- Ths size of `buff` can be different on each processor.
- We first do a `gather` of the sizes of the local vectors to `recvcnts[]`. With this information we compute the *displacements* `displs[]`.
- The sum of the `recvcnts[]` 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 eith the size of the largest buffer in the slaves.

The print-par function (cont.)

```
void print_par(vector<int> &buff,const char *s=NULL) {
  int myrank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  // reception buffer in master, receive counts
  // and displacements
  vector<int> rbuff, recvcnts(size,0), displs(size,0);

  // Each procesor send its size
  int sendcnt = buff.size();
  MPI_Gather(&sendcnt,1,MPI_INT,
             &recvcnts[0],1,MPI_INT,0,MPI_COMM_WORLD);
  if (!myrank) {
    // Resize reception buffer and
    // compute displs[] in master
    int rsize = 0;
    for (int j=0; j<size; j++) rsize += recvcnts[j];
    rbuff.resize(rsize);
    displs[0] = 0;
    for (int j=1; j<size; j++)
      displs[j] = displs[j-1] + recvcnts[j-1];

  }
  // Do the gather
  MPI_Gatherv(&buff[0],sendcnt,MPI_INT,
              &rbuff[0],&recvcnts[0],&displs[0],MPI_INT,
                    0,MPI_COMM_WORLD);
  if (!myrank) {
    // Print all buffers in master
    if (s) printf("%s",s);
    for (int j=0; j<size; j++) {
      printf("in proc [%d]: ",j);
      for (int k=0; k<recvcnts[j]; k++) {
        int ptr = displs[j]+k;
        printf("%d ",rbuff[ptr]);
      }
      printf("\n");
    }
  }
}
```

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

```
void re_scatter(vector<int> &buff,int size,
                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`

```
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

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

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

If

- `buff=`$\{$3,4,2,5,4$\}$ in $P0$
- `buff=`$\{$3,5,2,1,3,2$\}$ in $P1$
- `buff=`$\{$7,9,10$\}$ en $P2$

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

then after `re_scatter(buff,...,mod_scatter,...)` we should have

- `buff=`$\{$3,3,3,9$\}$ in $P0$
- `buff=`$\{$4,4,1,7,10$\}$ in $P1$
- `buff=`$\{$2,5,5,2,2$\}$ in $P2$

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

We can pass global parameters to *mod_scatter*,

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

so that:

```
//-------------------------------------------------
// Initially:
// buff={3,4,2,5,4} en P0, {3,5,2,1,3,2} en P1, {7,9,10} en P2.

k = 0; re_scatter(buff,...,mod_scatter,...);
// buff={3,3,3,9} en P0, {4,4,1,7,10} en P1, {2,5,5,2,2} en P2

k = 1; re_scatter(buff,...,mod_scatter,...);
// buff={4,4,1,7,10} en P0, {2,5,5,2,2} en P1, {3,3,3,9} en P2

k = 2; re_scatter(buff,...,mod_scatter,...);
// 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

```
int rotate_elems(int elem,int size, void *data) {
  int key =*(int *)data;
  return (elem - key) % size;
}
// Fill 'buff'
. . . .
for (int k=0; k<kmax; k++)
  re_scatter(buff,size,myrank,mod_scatter,&k);
  // print elements . . . .
}
```

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

```
[mstorti@spider example]$ mpirun -np 4 \
                         -machinefile machi.dat rescatter.bin
initial:
[0]: 383 886 777 915 793 335 386 492 649 421
[1]:
[2]:
[3]:
after rescatter: -------
[0]: 492
[1]: 777 793 649 421
[2]: 886 386
[3]: 383 915 335
after rescatter: -------
[0]: 777 793 649 421
[1]: 886 386
[2]: 383 915 335
[3]: 492
after rescatter: -------
[0]: 886 386
[1]: 383 915 335
[2]: 492
[3]: 777 793 649 421
...
```

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

```
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*.

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

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

Examples:

- **Sort with a comparison function:**

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

For instance, sort by absolute value:

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

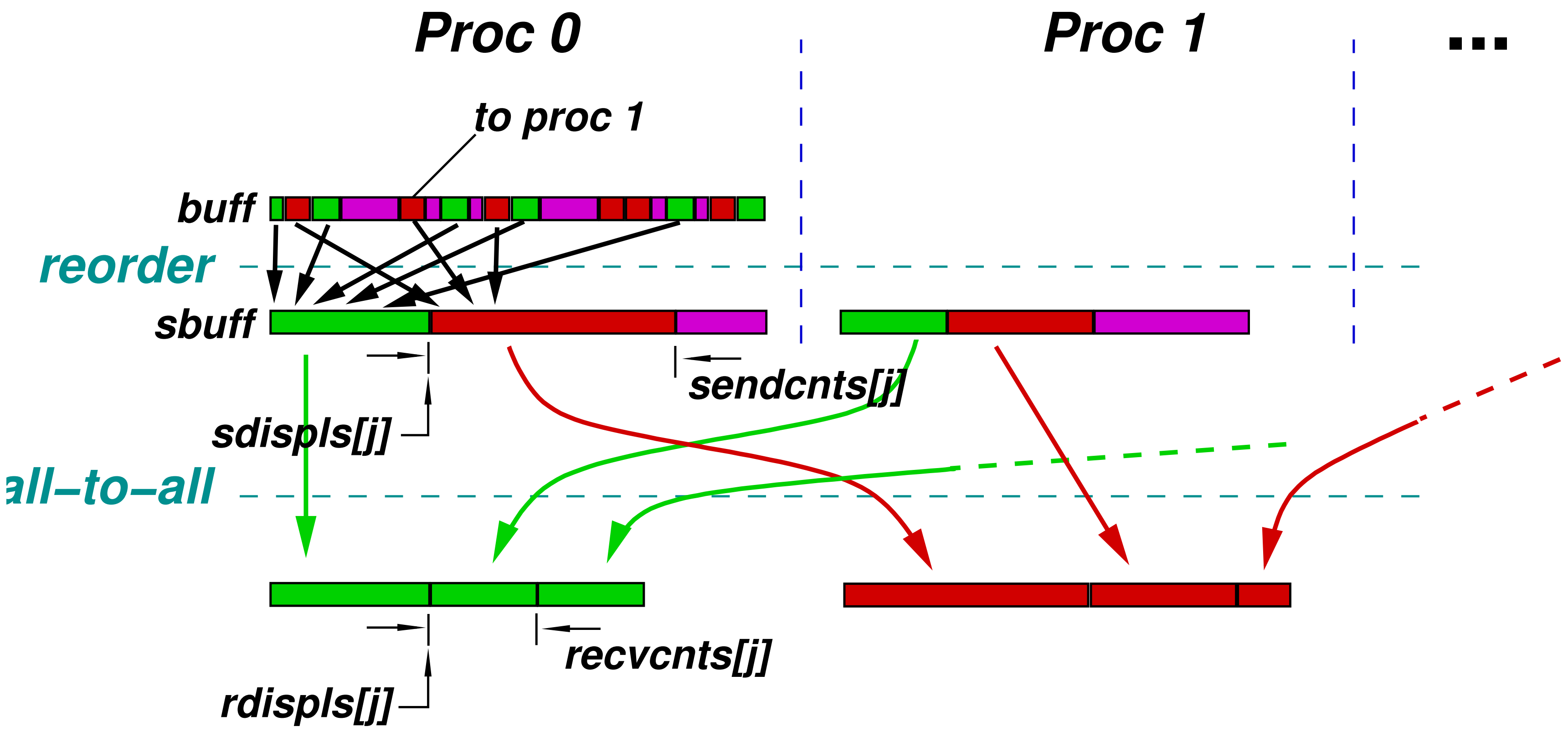
- **Filter a list (leave only objects that satisfy a given predicate):**

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

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

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

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



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

Seudo-código:

```
void re_scatter(vector<int> &buff,int size,
                int myrank,proc_fun proc,void *data) {
  int N = buff.size();
  // Check how many elements should go to
  // each processor
  for (int j=0; j<N; j++) {
    int p = proc(buff[j],size,data);
    sendcnts[p]++;
  }
  // Allocate sbuff and reorder (buff -> sbuff). . .
  // Compute all 'send' displacements
  for (int j=1; j<size; j++)
    sdispls[j] = sdispls[j-1] + sendcnts[j-1];

  // Use 'Alltoall' for scattering the dimensions
  // of the buffers to be received
  MPI_Alltoall(&sendcnts[0],1,MPI_INT,
               &recvcnts[0],1,MPI_INT,MPI_COMM_WORLD);

  // Compute receive size 'rsize' and 'rdispls' from 'recvcnts' . . .
  // resize 'buff' to 'rsize' . . .

  MPI_Alltoallv(&sbuff[0],&sendcnts[0],&sdispls[0],MPI_INT,
                &buff[0],&recvcnts[0],&rdispls[0],MPI_INT,
                MPI_COMM_WORLD);
}
```

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

Complete code:

```
void re_scatter(vector<int> &buff,int size,
                int myrank,proc_fun proc,void *data) {
  vector<int> sendcnts(size,0);
  int N = buff.size();
  // Check how many elements should go to
  // each processor
  for (int j=0; j<N; j++) {
    int p = proc(buff[j],size,data);
    sendcnts[p]++;
  }

  // Dimension buffers and ptr vectors
  vector<int> sbuff(N);
  vector<int> sdispls(size);
  vector<int> recvcnts(size);
  vector<int> rdispls(size);
  sdispls[0] = 0;
  for (int j=1; j<size; j++)
    sdispls[j] = sdispls[j-1] + sendcnts[j-1];

  // Reorder by processor from buff to sbuff
  for (int j=0; j<N; j++) {
    int p = proc(buff[j],size,data);
    int pos = sdispls[p];
    sbuff[pos] = buff[j];
    sdispls[p]++;
  }
  // Use ‘Alltoall’ for scattering the dimensions
  // of the buffers to be received
  MPI_Alltoall(&sendcnts[0],1,MPI_INT,
               &recvcnts[0],1,MPI_INT,MPI_COMM_WORLD);

  // Compute the ‘send’ and ‘recv’ displacements.
  rdispls[0] = 0;
  sdispls[0] = 0;
  for (int j=1; j<size; j++) {
    rdispls[j] = rdispls[j-1] + recvcnts[j-1];
    sdispls[j] = sdispls[j-1] + sendcnts[j-1];
  }

  // Dimension the receive size
  int rsize = 0;
  for (int j=0; j<size; j++) rsize += recvcnts[j];
  buff.resize(rsize);

  // Do the scatter.
  MPI_Alltoallv(&sbuff[0],&sendcnts[0],&sdispls[0],MPI_INT,
                &buff[0],&recvcnts[0],&rdispls[0],MPI_INT,
                MPI_COMM_WORLD);
}
```

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 reaplce row `k`, then we can use `MPI_Send()` and `MPI_Recv()` since the rows represent adjacent values

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

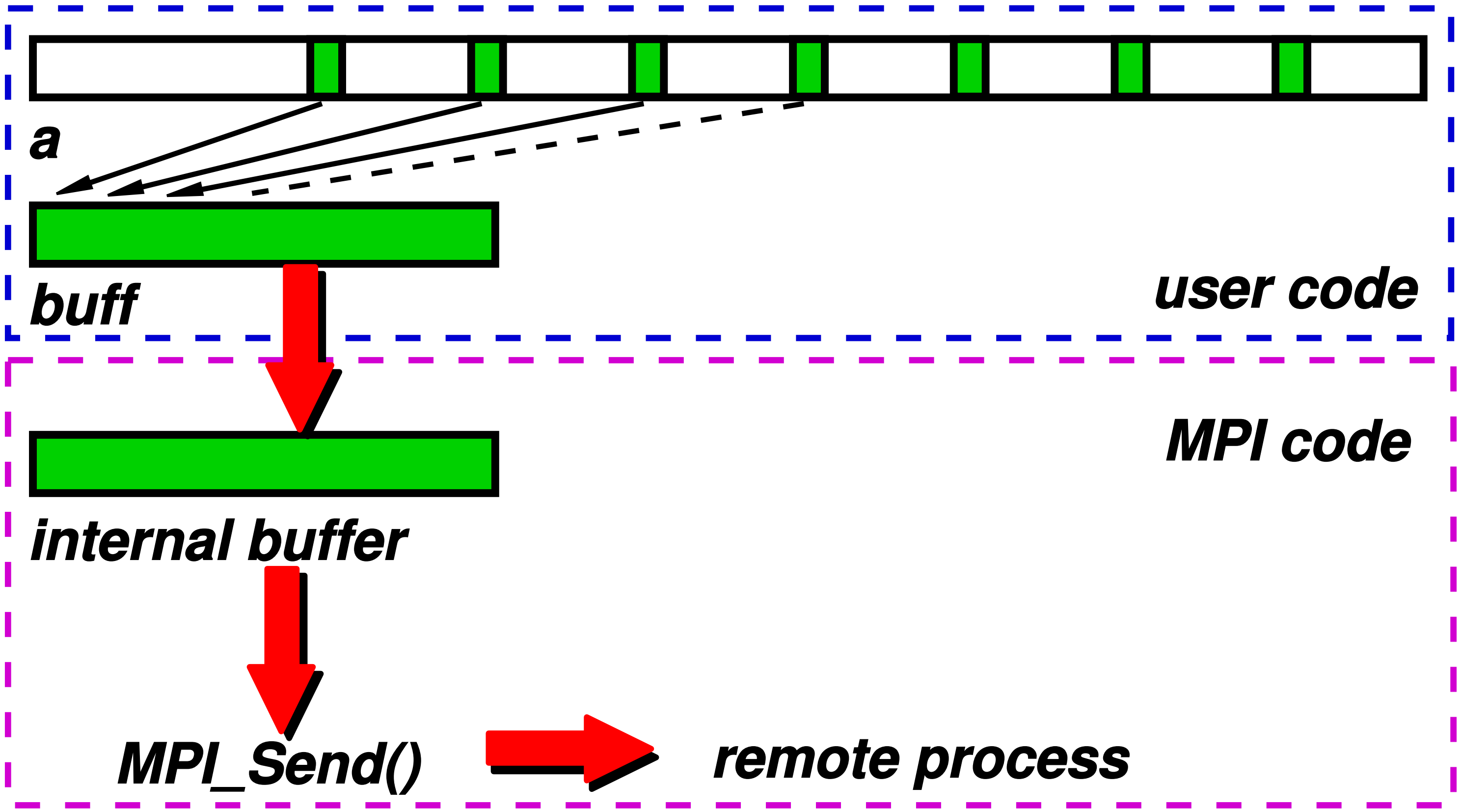
Definiendo tipos de datos derivados (cont.)

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.

```
double *buff = new double[N];
if (myrank==source) {
  // Gather column in `buff' and send
  for (int l=0; l<N; l++)
     buff[l] = a[l*N+j];
     MPI_Send(buff,N,MPI_DOUBLE,dest,0,MPI_COMM_WORLD);
} else if (myrank==dest) {
  // Receive `buff' and put data in column
  MPI_Recv(buff,N,MPI_DOUBLE,source,0,
           MPI_COMM_WORLD,&status);
  for (int l=0; l<N; l++)
     a[l*N+k] = buff[l];
}
delete[] buff;
```

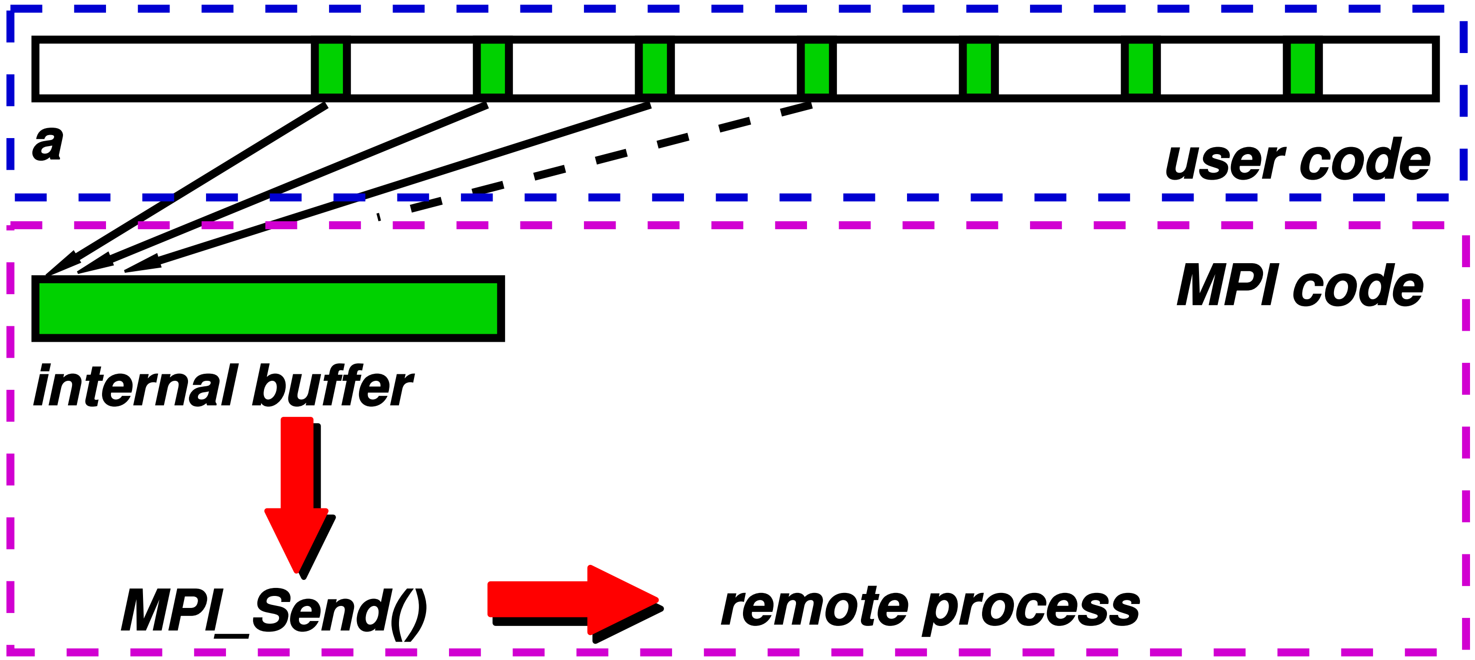
Definiendo tipos de datos derivados (cont.)

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.



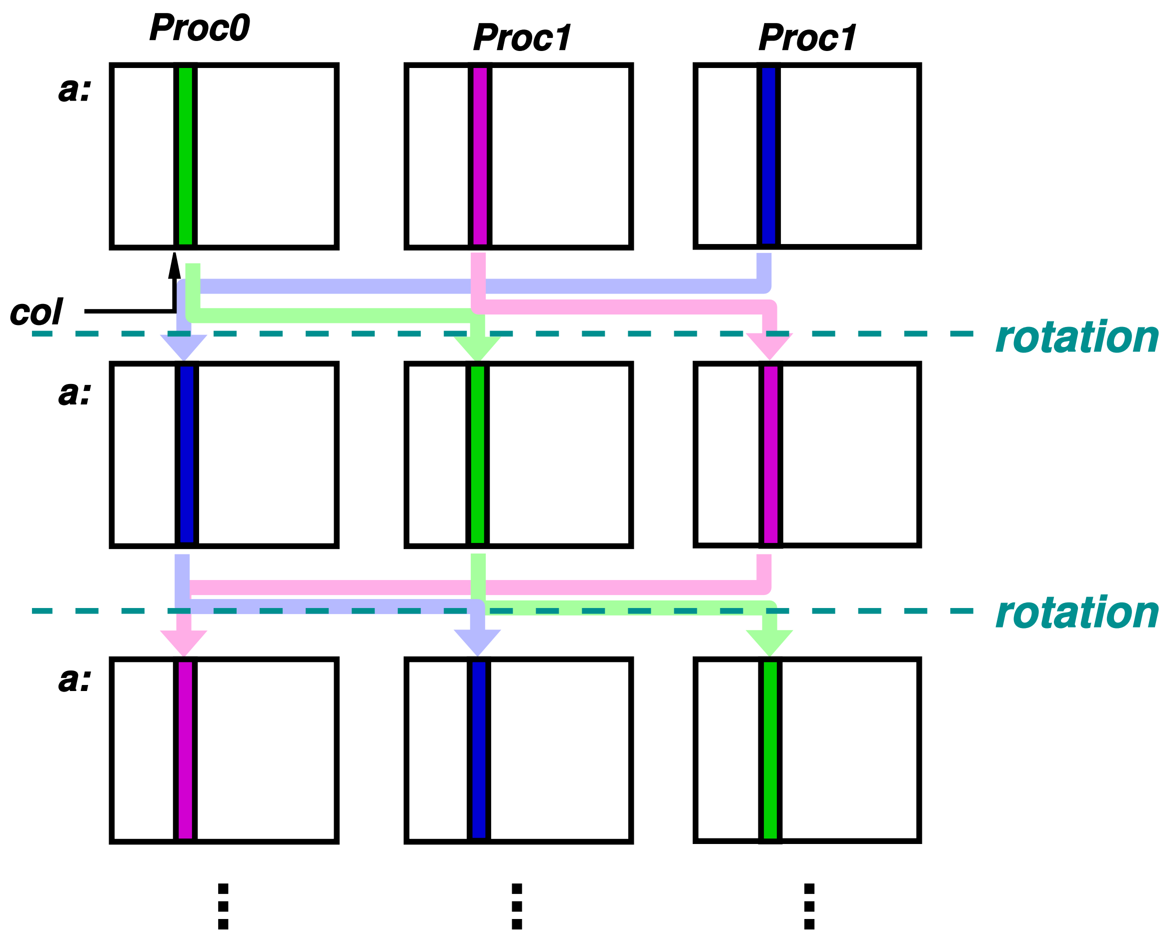
Definiendo tipos de datos derivados (cont.)

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

```
MPI_Datatype stride;
MPI_Type_vector(N,1,N,MPI_DOUBLE,&stride);
MPI_Type_commit(&stride);

// use 'stride' . . .

// Free resources reserved for type 'stride'
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.)

```
int col = 2;
MPI_Status status;
int dest = myrank+1;
if (dest==size) dest = 0;
int source = myrank-1;
if (source==-1) source = size-1;
vector<double> recv_val(N);;

MPI_Datatype stride;
MPI_Type_vector(N,1,N,MPI_DOUBLE,&stride);
MPI_Type_commit(&stride);

for (int k=0; k<10; k++) {
  MPI_Sendrecv(&A(0,col),1,stride,dest,0,
               &recv_val[0],N,MPI_DOUBLE,source,0,
               MPI_COMM_WORLD,&status);
  for (int j=0; j<N; j++)
    A(j,col) = recv_val[j];
  if (!myrank) {
    printf("After rotation step %d\n",k);
    print_mat(a,N);
  }
}
MPI_Type_free(&stride);
MPI_Finalize();
```

Ejemplo: rotar columna de una matriz (cont.)

Between the rotation

```
Before rotation
Proc [0] a:               Proc [1] a:               Proc [2] a:
4.8 3.2 [0.2] 9.9         7.2 3.3 [9.9] 1.7         3.9 5.6 [9.5] 5.5
7.9 8.8 [1.7] 7.2         8.9 2.2 [9.8] 2.3         5.1 3.3 [8.9] 1.2
2.7 4.2 [2.0] 4.1         6.7 8.5 [5.7] 3.0         3.0 5.2 [6.1] 8.4
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:

```
[mstorti@spider example]$ mpirun -np 3 \
               -machinefile ./machi.dat mpitypes.bin
After rotation step 0
4.8 3.2 9.5 9.9
7.9 8.8 8.9 7.2
2.7 4.2 6.1 4.1
7.1 0.8 6.1 9.4
After rotation step 1
4.8 3.2 9.9 9.9
7.9 8.8 9.8 7.2
2.7 4.2 5.7 4.1
7.1 0.8 8.2 9.4
After rotation step 2
4.8 3.2 0.2 9.9
7.9 8.8 1.7 7.2
2.7 4.2 2.0 4.1
7.1 0.8 9.2 9.4
...
```

OPTIONAL Assignement Nbr. 5

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

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

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<=k<nc`, `II[k]`, `JJ[k]`, `AIJ[k]` are the row and column indices, and the coefficients of A. For instance, if `A=[0 1 0;2 3 0;1 0 0]`, 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

```
for (int i=0; i<n; i++) r[i] = 0;
for (int k=0; k<nc; k++) {
  int i = II[k];
  int j = JJ[k];
  int a = AIJ[k];
  r[i] -= a * x[j];
}
for (int i=0; i<n; i++) r[i] += b[i];
```

The pseudocode for the Richardson method would be then

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

We propose the following parallel implementation using MPI

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

```
void read_matrix(char *file,
                 int ncmax,int *I,int *J,double * AIJ,int &nc,
                 int nmax,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.

```
  int nlocal = n/size;
  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 ad 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 de 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 groups according to the abstract object types that are involved (e.g. vectores, matrices, solvers...), in a similar way to classes in C++.**

PETSc Objects

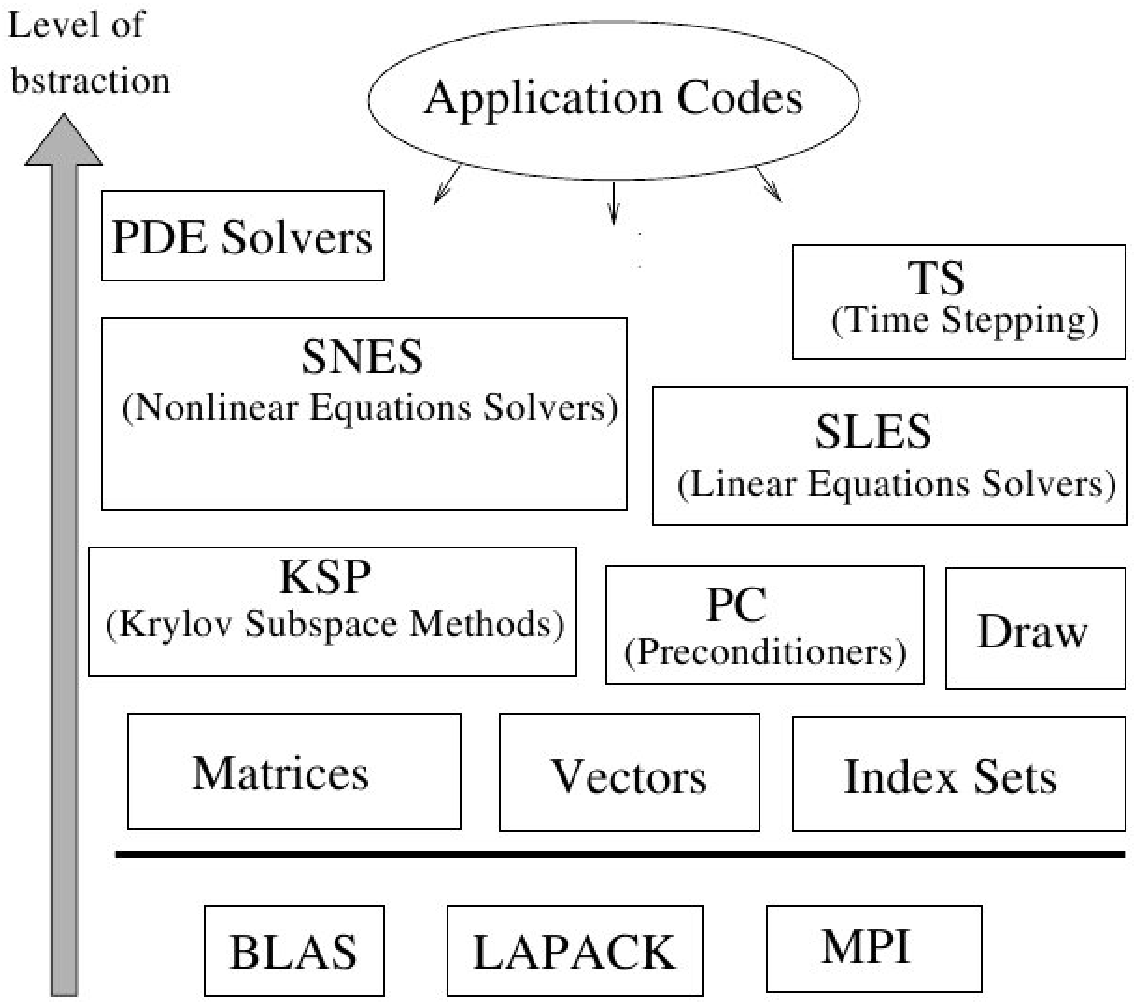
PETSc objects

- **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 compillation 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.**

```
[mstorti@spider example]$ mpirun  <mpi_options> \
              <petsc_program name> <petsc_options> \
              <user_prog_options>
[mstorti@spider example]$
```

Por ejemplo

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

Writing programs that use PETSc

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

```
C FORTRAN
      call PetscInitialize (character file,
                 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.)

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

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

Calls `MPI_Finalize` (if MPI was initializated by PETSc).

Simple example. The Laplace 1D eq.

Solve $Ax = b$, where

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

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

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

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

Create the objects

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

In general

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

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

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

Duplicate **(*clone*) objects**

```
Vec x,b,u;
ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);

ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
ierr = VecDuplicate(x,&u);CHKERRQ(ierr);
```

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

Set values on vectors and matrices

```
Vec b; VecCreate(...,b);
Mat A; MatCreate(...,A);

// Equivalent to Matlab: b(row) = val; A(row,col) = val;
VecSetValue(b, row, val,INSERT_VALUES);
MatSetValue(A, row, col, val,INSERT_VALUES);

// Equivalent to Matlab: b(row) = b(row) + val;
//                       A(row,col) = A(row,col) + val;
VecSetValue(b, row, val,ADD_VALUES);
MatSetValue(A, row, col, val,ADD_VALUES);
```

Also may assemble matrices *by blocks*

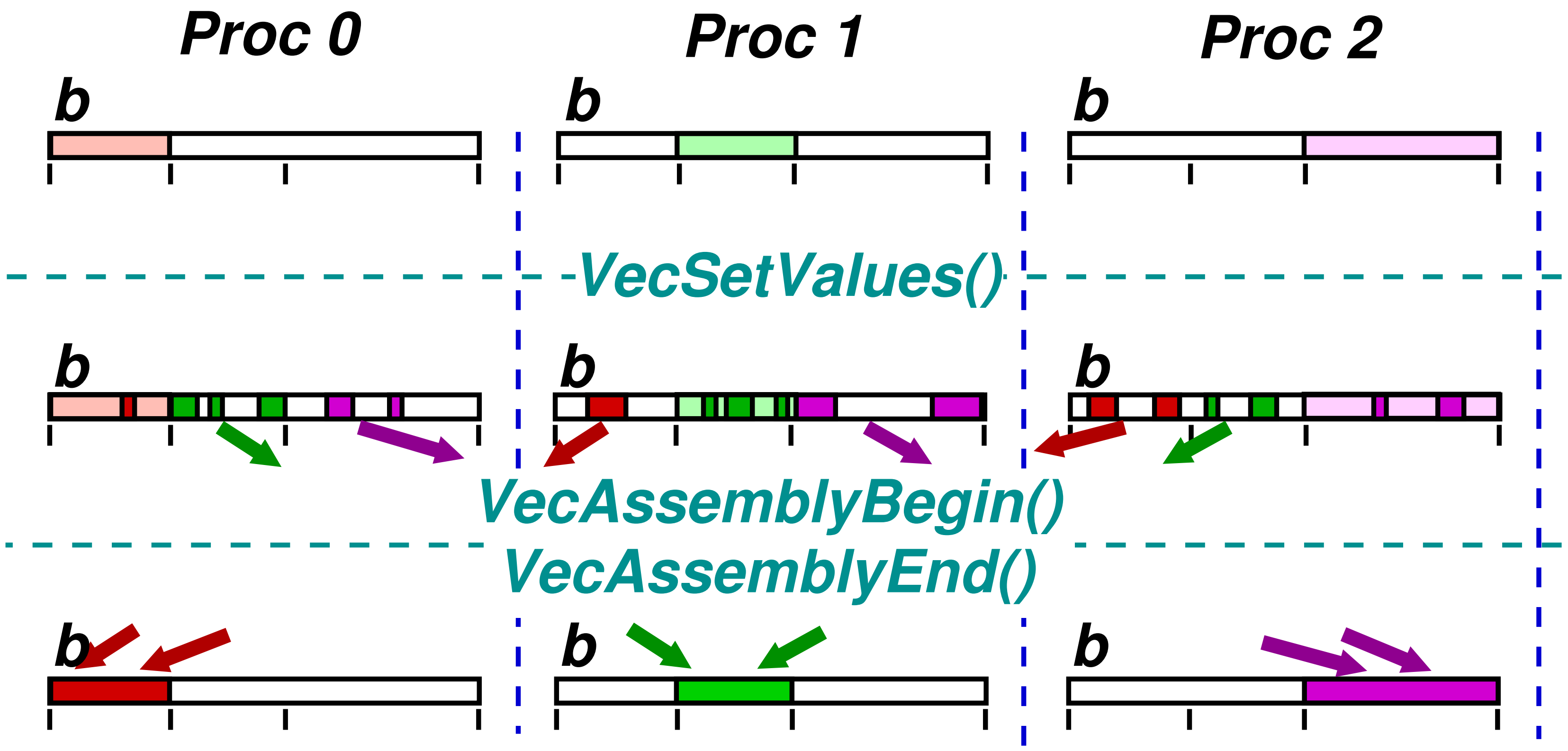
```
int nrow, *rows, ncols, *cols;
double *vals;
// Equivalent to Matlab: A(rows,cols) = vals;
MatSetValues(A,nrows,rows,ncols,cols,vals,INSERT_VALUES);
```

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

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

```
Mat A; MatCreate(...,A);
MatSetValues(A,nrows,rows,ncols,cols,vals,INSERT_VALUES);

// Starts communication
MatAssemblyBegin(A,...);
// Can't use 'A' yet
// (Can overlap comp/comm.) Do computations ....
// Ends communication
MatAssemblyEnd(A,...);
// Can use 'A' now
```

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



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

```
// Assemble matrix. All processors set all values.
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=1; i<n-1; i++) {
  col[0] = i-1; col[1] = i; col[2] = i+1;
  ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);
  CHKERRQ(ierr);
}
i = n - 1; col[0] = n - 2; col[1] = n - 1;
ierr = MatSetValues(A,1,&i,2,col,value,
                    INSERT_VALUES); CHKERRQ(ierr);
i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
ierr = MatSetValues(A,1,&i,2,col,value,
                    INSERT_VALUES); CHKERRQ(ierr);
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
```

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

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

```
KSP ksp;
KSPCreate(PETSC_COMM_WORLD,&ksp);
ierr = KSPSetOperators(KSP,A,A,
                       DIFFERENT_NONZERO_PATTERN);

PC pc;
KSPGetPC(KSP,&pc);
PCSetType(pc,PCJACOBI);
KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,
                 PETSC_DEFAULT,PETSC_DEFAULT);

KSPSolve(ksp,b,x,&its);

ierr = KSPDestroy(ksp);CHKERRQ(ierr);
```

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

```
/* Program usage: mpiexec ex1 [-help] [all PETSc options] */

static char help[] =
  "Solves a tridiagonal linear system with KSP.\n\n";

#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  Vec          x, b, u;    /* approx solution, RHS,
                                exact solution */
  Mat          A;          /* linear system matrix */
  KSP          ksp;        /* linear solver context */
  PC           pc;         /* preconditioner context */
  PetscReal    norm;       /* norm of solution error */
  PetscErrorCode ierr;
  PetscInt     i,n = 10,col[3],its;
  PetscMPIInt  size;
  PetscScalar  neg_one = -1.0,one = 1.0,value[3];

  PetscInitialize(&argc,&args,(char *)0,help);
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
  if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",
                            &n,PETSC_NULL);CHKERRQ(ierr);
```

```

/* - - - - - - - - - - - - - - - - - - - - - -
      Compute the matrix and right-hand-side vector that define
      the linear system, Ax = b.
   - - - - - - - - - - - - - - - - - - - - - - */

/*
   Create vectors. Note that we form 1 vector from scratch and
   then duplicate as needed.
*/
ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject) x,
                          "Solution");CHKERRQ(ierr);
ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

/*
   Create matrix. When using MatCreate(), the matrix
   format can be specified at runtime.

   Performance tuning note: For problems of substantial
   size, preallocation of matrix memory is crucial for
   attaining good performance. See the matrix chapter of
   the users manual for details.
*/
ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
ierr = MatSetSizes(A,PETSC_DECIDE,
                   PETSC_DECIDE,n,n); CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);

/*
```

```
     Assemble matrix
  */
  value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
  for (i=1; i<n-1; i++) {
    col[0] = i-1; col[1] = i; col[2] = i+1;
    ierr = MatSetValues(A,1,&i,3,col,
                        value,INSERT_VALUES);CHKERRQ(ierr);
  }
  i = n - 1; col[0] = n - 2; col[1] = n - 1;
  ierr = MatSetValues(A,1,&i,2,col,
                      value,INSERT_VALUES);CHKERRQ(ierr);
  i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
  ierr = MatSetValues(A,1,&i,2,col,
                      value,INSERT_VALUES);CHKERRQ(ierr);
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  /*
     Set exact solution; then compute right-hand-side vector.
  */
  ierr = VecSet(u,one);CHKERRQ(ierr);
  ierr = MatMult(A,u,b);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - -
     Create the linear solver and set various options
     - - - - - - - - - - - - - - - - - - - - - - - -*/
  /*
     Create linear solver context
  */
  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

  /*
```

```
     Set operators. Here the matrix that defines the linear
     system also serves as the preconditioning matrix.
  */
  ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);
  CHKERRQ(ierr);

  /*
     Set linear solver defaults for this problem (optional).
     - By extracting the KSP and PC contexts from the KSP
       context, we can then directly call any KSP and PC
       routines to set various options.
     - The following four statements are optional; all of
       these parameters could alternatively be specified at
       runtime via KSPSetFromOptions();
  */
  ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
  ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
  ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,
                           PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);

  /*
    Set runtime options, e.g.,
      -ksp_type <type> -pc_type <type>
            -ksp_monitor -ksp_rtol <rtol>
    These options will override those specified above as
    long as KSPSetFromOptions() is called _after_ any other
    customization routines.
  */
  ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - -
            Solve the linear system
  - - - - - - - - - - - - - - - - - - - - - - - */
ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

/*
  View solver info; we could instead use the option
  -ksp_view to print this info to the screen at the
  conclusion of KSPSolve().
*/
ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - -
                Check solution and clean up
  - - - - - - - - - - - - - - - - - - - - - - - -*/
/*
  Check the error
*/
ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,
               "Norm of error %A, Iterations %D\n",
               norm,its);CHKERRQ(ierr);
/*
  Free work space. All PETSc objects should be destroyed
  when they are no longer needed.
*/
ierr = VecDestroy(x);CHKERRQ(ierr);
ierr = VecDestroy(u);CHKERRQ(ierr);
ierr = VecDestroy(b);CHKERRQ(ierr);
ierr = MatDestroy(A);CHKERRQ(ierr);
ierr = KSPDestroy(ksp);CHKERRQ(ierr);

```

```
  /*

     Always call PetscFinalize() before exiting a program.
     This routine
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if
         certain runtime options are chosen (e.g.,
         -log_summary).
  */
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

PETSc elements

Headers

```
#include "petscsksp.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** `~/.petscrc`
- **environment variables** `PETSC_OPTIONS`
- **command line, e.g.:** `mpirun -np 1 ex1 -n 100`

Options can be obtained in execution time with

```
OptionsGetInt(PETSC_NULL,"-n",&n,&flg);
```

Vectors

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

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

Matrices

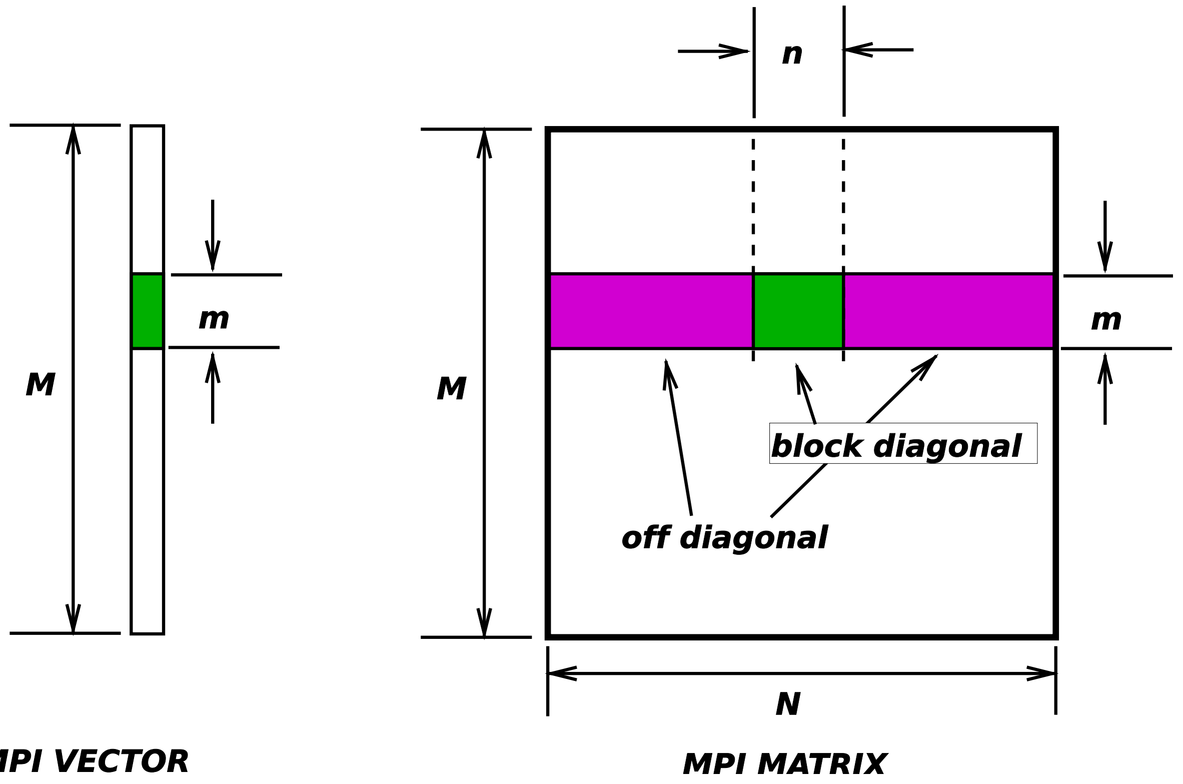
```
// Create a matrix
MatCreate(MPI_Comm comm ,int m,
               int n,int M ,int N,Mat *A);

// Set values in rows 'im' and columns 'in'
MatSetValues(Mat A,int m,int *im,int n,int *in,
               PetscScalar *values,INSERT_VALUES);

// Do assembly BEFORE using the matrices
// (values may be stored in temporary buffers yet)
MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

Linear solvers

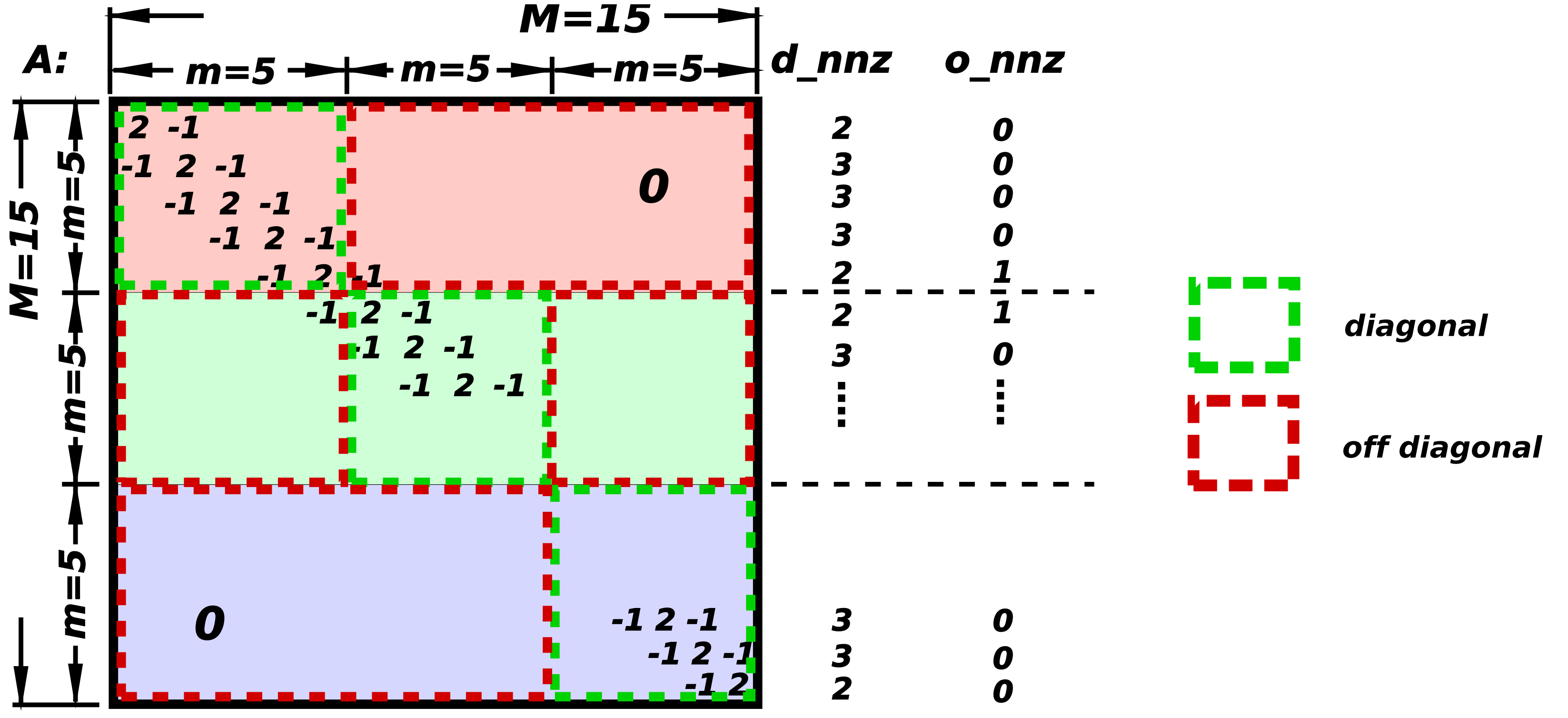
```
// Crete the KSP
KSPCreate(MPI_Comm comm ,KSP *ksp);

// Set matrix and preconditioning
KSPSetOperators (KSP ksp,Mat A,
     Mat PrecA,MatStructure flag);

// Use options from data base (solver type?)
//                (file/env/command)
KSPSetFromOptions (KSP ksp);

// Solve linear system
KSPSolve (KSP ksp,Vec b,Vec x,int *its);

// Destroy KSP (free mem. of fact. matrix)
KSPDestroy (KSP ksp);
```

Parallel programming

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



Parallel programming (cont.)



Parallel programming (cont.)

```
static char help[] = "Solves a tridiagonal linear system.\n\n";
#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  Vec            x, b, u;    /* approx solution, RHS,
                                exact solution */
  Mat            A;          /* linear system matrix */
  KSP            ksp;       /* linear solver context */
  PC             pc;         /* preconditioner context */
  PetscReal      norm;       /* norm of solution error */
  PetscErrorCode ierr;
  PetscInt       i,n = 10,col[3],its,rstart,rend,nlocal;
  PetscScalar    neg_one = -1.0,one = 1.0,value[3];

  PetscInitialize(&argc,&args,(char *)0,help);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
  CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - -
         Compute the matrix and right-hand-side vector that
     define the linear system, Ax = b.
     -- - - - - - - - - - - - - - - - - - - - - - */

  /* Create vectors. Note that we form 1 vector from
    scratch and then duplicate as needed. For this simple
```

```
     case let PETSc decide how many elements of the vector
     are stored on each processor. The second argument to
     VecSetSizes() below causes PETSc to decide.  */
  ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
  ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
  ierr = VecSetFromOptions(x);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

  /* Identify the starting and ending mesh points on each
     processor for the interior part of the mesh. We let
     PETSc decide above. */

  ierr = VecGetOwnershipRange(x,&rstart,&rend);CHKERRQ(ierr);
  ierr = VecGetLocalSize(x,&nlocal);CHKERRQ(ierr);

  /* Create matrix. When using MatCreate(), the matrix
     format can be specified at runtime.

     Performance tuning note: For problems of substantial
     size, preallocation of matrix memory is crucial for
     attaining good performance. See the matrix chapter of
     the users manual for details.

     We pass in nlocal as the ``local'' size of the matrix to
     force it to have the same parallel layout as the vector
     created above. */
  ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
  ierr = MatSetSizes(A,nlocal,nlocal,n,n);CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);

  /* Assemble matrix.
     The linear system is distributed across the processors
     by chunks of contiguous rows, which correspond to
     contiguous sections of the mesh on which the problem is
     discretized. For matrix assembly, each processor
     contributes entries for the part that it owns locally. */
  if (!rstart) {
    rstart = 1;
    i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
    ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);
    CHKERRQ(ierr);
  }
  if (rend == n) {
    rend = n-1;
    i = n-1; col[0] = n-2; col[1] = n-1;
    value[0] = -1.0; value[1] = 2.0;
    ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);
    CHKERRQ(ierr);
  }

  /* Set entries corresponding to the mesh interior */
  value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
  for (i=rstart; i<rend; i++) {
    col[0] = i-1; col[1] = i; col[2] = i+1;
    ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);
    CHKERRQ(ierr);
  }

  /* Assemble the matrix */
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  /* Set exact solution; then compute right-hand-side vector. */
  ierr = VecSet(u,one);CHKERRQ(ierr);
```

```
ierr = MatMult(A,u,b);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - -
      Create the linear solver and set various options
   - - - - - - - - - - - - - - - - - - - - - - - - - - */
ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

/* Set operators. Here the matrix that defines the linear system
   also serves as the preconditioning matrix. */
ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);
CHKERRQ(ierr);

/*
   Set linear solver defaults for this problem (optional).
   - By extracting the KSP and PC contexts from the KSP context,
     we can then directly call any KSP and PC routines to set
     various options.
   - The following four statements are optional; all of these
     parameters could alternatively be specified at runtime via
     KSPSetFromOptions();
*/
ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,
                        PETSC_DEFAULT);CHKERRQ(ierr);

/*
  Set runtime options, e.g.,
      -ksp_type <type> -pc_type <type>
      -ksp_monitor -ksp_rtol <rtol>
  These options will override those specified above as
  long as KSPSetFromOptions() is called _after_ any other
```

```
    customization routines.
  */
  ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - -
             Solve the linear system
     - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
     Solve linear system
  */
  ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

  /*
     View solver info; we could instead use the option
     -ksp_view to print this info to the screen at the
     conclusion of KSPSolve().
  */
  ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - -
              Check solution and clean up
     - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
     Check the error
  */
  ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
  ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
  ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,
                     "Norm of error %A, Iterations %D\n",
                     norm,its);CHKERRQ(ierr);
  /*
     Free work space. All PETSc objects should be destroyed
     when they are no longer needed.
  */
  ierr = VecDestroy(x);CHKERRQ(ierr);
  ierr = VecDestroy(u);CHKERRQ(ierr);
  ierr = VecDestroy(b);CHKERRQ(ierr);
  ierr = MatDestroy(A);CHKERRQ(ierr);
  ierr = KSPDestroy(ksp);CHKERRQ(ierr);

  /*
     Always call PetscFinalize() before exiting a program.
     This routine
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if
         certain runtime options are chosen (e.g.,
         -log_summary).
  */
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

FEM program based on PETSc

FEM program based on PETSc

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

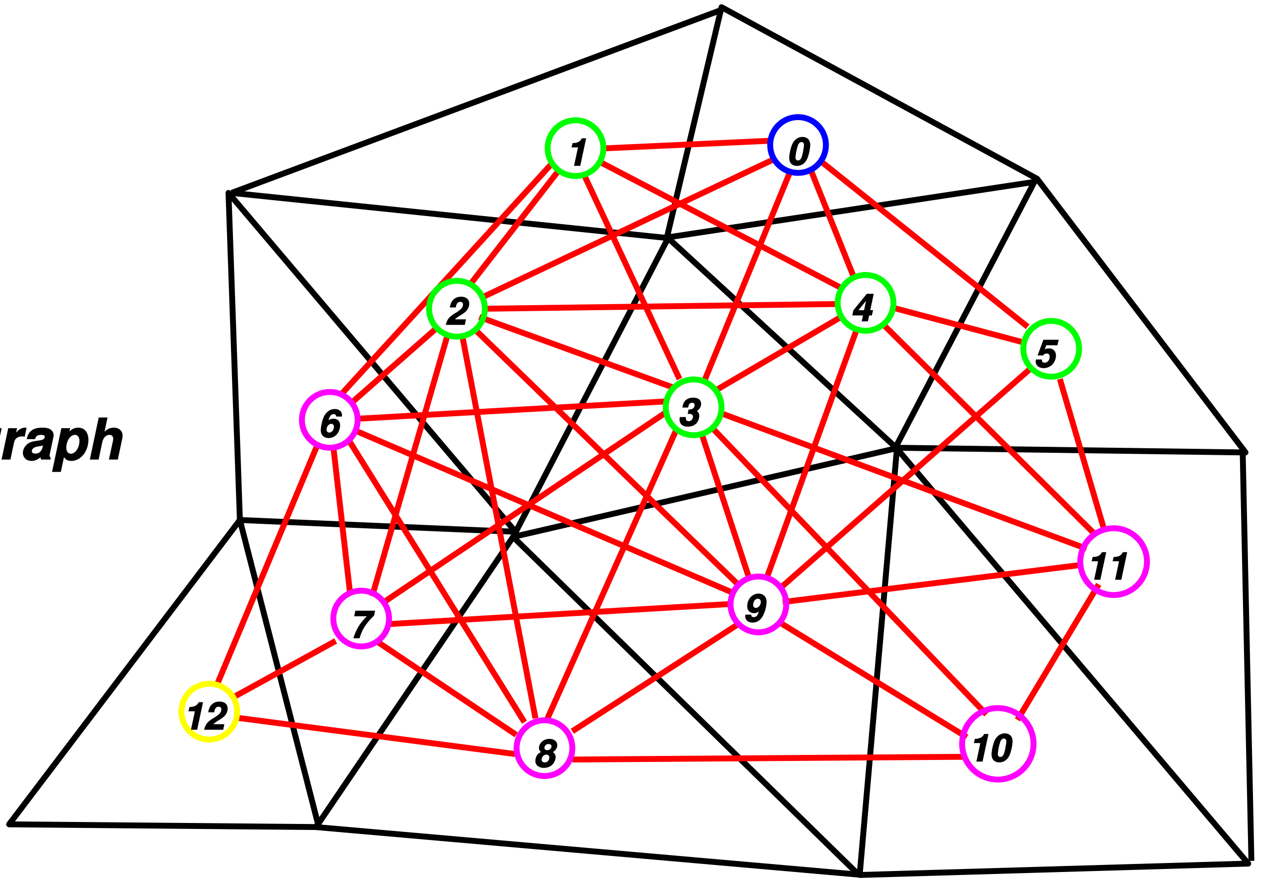
Basic concepts on partitioning

Graph numbering by Cuthill-Mc.Kee

```
// Number vertices in a graph
Graph G, Queue C;
Vertex n = any vertex in G;
last = 0;
Indx[n] = last++;
Put n in C;
Vertex c,m;
while (! C.empty()) {
  n = C.front();
  C.pop();
  for (m in neighbors(c,G)) {
    if (m not already indexed) {
      C.push(m);
      Indx[m] = last++;
    }
  }
}
```

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

dual graph



Element partitioning

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

```
#include <cstdio>

#include <iostream>
#include <set>
#include <vector>
#include <deque>
#include <map>

#include <petscsles.h>

#include <newmatio.h>

static char help[] =
"Basic test for self scheduling algorithm FEM with PETSc";

//-------<*>-------<*>-------<*>-------<*>-------<*>-------
#undef __FUNC__
#define __FUNC__ "main"
int main(int argc,char **args) {
  Vec b,u;
  Mat A;
  SLES sles;
  PC     pc;          /* preconditioner context */
  KSP    ksp;         /* Krylov subspace method context */

  // Number of nodes per element
  const int nel=3;
  // Space dimension
  const int ndim=2;
```

```

PetscInitialize(&argc,&args,(char *)0,help);

// Get MPI info
int size,rank;
MPI_Comm_size(PETSC_COMM_WORLD,&size);
MPI_Comm_rank(PETSC_COMM_WORLD,&rank);

// read nodes
double x[ndim];
int read;
vector<double> xnod;
vector<int> icone;
FILE *fid = fopen("node.dat","r");
assert(fid);
while (true) {
  for (int k=0; k<ndim; k++) {
    read = fscanf(fid,"%lf",&x[k]);
    if (read == EOF) break;
  }
  if (read == EOF) break;
  xnod.push_back(x[0]);
  xnod.push_back(x[1]);
}
fclose(fid);
int nnod=xnod.size()/ndim;
PetscPrintf(PETSC_COMM_WORLD,"Read %d nodes.\n",nnod);

// read elements
int ix[nel];
fid = fopen("icone.dat","r");
assert(fid);
while (true) {
  for (int k=0; k<nel; k++) {
    read = fscanf(fid,"%d",&ix[k]);
    if (read == EOF) break;
  }
  if (read == EOF) break;
  icone.push_back(ix[0]);
  icone.push_back(ix[1]);
  icone.push_back(ix[2]);
}
fclose(fid);

int nelem=icone.size()/nel;
PetscPrintf(PETSC_COMM_WORLD,"Read %d elements.\n",nelem);

// read fixations stored as a map node -> value
map<int,double> fixa;
fid = fopen("fixa.dat","r");
assert(fid);
while (true) {
  int nod;
  double val;
  read = fscanf(fid,"%d %lf",&nod,&val);
  if (read == EOF) break;
  fixa[nod]=val;
}
fclose(fid);
PetscPrintf(PETSC_COMM_WORLD,"Read %d fixations.\n",fixa.size());

// Construct node to element pointer
// n2e[j-1] is the set of elements connected to node 'j'
vector< set<int> > n2e;
n2e.resize(nnod);
for (int j=0; j<nelem; j++) {
  for (int k=0; k<nel; k++) {
    int nodo = icone[j*nel+k];
    n2e[nodo-1].insert(j+1);
  }
}

#if 0   // Output 'n2e' array if needed
for (int j=0; j<nnod; j++) {
  set<int>::iterator k;
  PetscPrintf(PETSC_COMM_WORLD,"node %d, elements: ",j+1);
  for (k=n2e[j].begin(); k!=n2e[j].end(); k++) {
    PetscPrintf(PETSC_COMM_WORLD,"%d ",*k);
  }
  PetscPrintf(PETSC_COMM_WORLD,"\n");
}
#endif

//---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
// Simple partitioning algorithm
deque<int> q;
vector<int> eord,id;
eord.resize(nelem,0);
id.resize(nnod,0);

// Mark element 0 as belonging to processor 0
q.push_back(1);
eord[0]=-1;
int order=0;


while (q.size()>0) {
```

```
  // Pop an element from the queue
  int elem=q.front();
  q.pop_front();
  eord[elem-1]=++order;
  // Push all elements neighbor to this one in the queue
  for (int nod=0; nod<nel; nod++) {
    int node = icone[(elem-1)*nel+nod];
    set<int> &e = n2e[node-1];

    for (set<int>::iterator k=e.begin(); k!=e.end(); k++) {
      if (eord[*k-1]==0) {
        q.push_back(*k);
        eord[*k-1]=-1;
      }
    }
  }
}
q.clear();

// Element partition. Put (approximmately)
// nelem/size in each processor
int *e_indx = new int[size+1];
e_indx[0]=1;
for (int p=0; p<size; p++)
  e_indx[p+1] = e_indx[p]+(nelem/size) + (p<(nelem % size) ? 1 : 0)

// Node partitioning. If a node is connected to an element 'j' then
// put it in the processor where element 'j' belongs.
// In case the elements connected to the node belong to
// different processors take any one of them.
int *id_indx = new int[size+2];
for (int j=0; j<nnod; j++) {
 if (fixa.find(j+1) != fixa.end()) {
   id[j]=-(size+1);
 } else {
   set<int> &e = n2e[j];
   assert(e.size()>0);
   int order = eord[*e.begin()-1];
   for (int p=0; p<size; p++) {
     if (order >= e_indx[p] && order < e_indx[p+1]) {
       id[j] = -(p+1);
       break;
     }
   }
 }
}

// ‘id_indx[j-1]’ is the dof associated to node ’j’
int dof=0;
id_indx[0]=0;
if (size>1)
 PetscPrintf(PETSC_COMM_WORLD,
           "dof distribution among processors\n");
for (int p=0; p<=size; p++) {
 for (int j=0; j<nnod; j++)
   if (id[j]==-(p+1)) id[j]=dof++;
 id_indx[p+1] = dof;
 if (p<size) {
   PetscPrintf(PETSC_COMM_WORLD,
               "proc: %d, from %d to %d, total %d\n",
               p,id_indx[p],id_indx[p+1],id_indx[p+1]-id_indx[p]);
 } else {
   PetscPrintf(PETSC_COMM_WORLD,
               "fixed: from %d to %d, total %d\n",
               id_indx[p],id_indx[p+1],id_indx[p+1]-id_indx[p]);
 }
}
n2e.clear();

int ierr;
// Total number of unknowns (equations)
int neq = id_indx[size];
// Number of unknowns in this processor
int ndof_here = id_indx[rank+1]-id_indx[rank];
// Creates a square MPI PETSc matrix
ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD,ndof_here,ndof_here,
                       neq,neq,0,
                       PETSC_NULL,0,PETSC_NULL,&A); CHKERRQ(ierr);
// Creates PETSc vectors
ierr = VecCreateMPI(PETSC_COMM_WORLD,
                    ndof_here,neq,&b); CHKERRQ(ierr);
ierr = VecDuplicate(b,&u); CHKERRQ(ierr);
double scal=0.;
ierr = VecSet(&scal,b);

{ // Compute element matrices and load them.
 // Each processor computes the elements belonging to him.

 // x12:= vector going from first to second node,
 // x13:= idem 1->3.
 // x1:= coordinate of node 1
 // gN gradient of shape function
 ColumnVector x12(ndim),x13(ndim),x1(ndim),gN(ndim);
 // grad_N := matrix whose columns are gradients of interpolation
```

```
// functions
// ke:= element matrix for the Laplace operator
Matrix grad_N(ndim,nel),ke(nel,nel);
// area:= element area
double area;
for (int e=1; e<=nelem; e++) {
  int ord=eord[e-1];
  // skip if the element doesn't belong to this processor
  if (ord < e_indx[rank] || ord >= e_indx[rank+1]) continue;
  // indices for vertex nodes of this element
  int n1,n2,n3;
  n1 = icone[(e-1)*nel];
  n2 = icone[(e-1)*nel+1];
  n3 = icone[(e-1)*nel+2];
  x1 << &xnod[(n1-1)*ndim];
  x12 << &xnod[(n2-1)*ndim];
  x12 = x12 - x1;
  x13 << &xnod[(n3-1)*ndim];
  x13 = x13 - x1;
  // compute as vector product
  area = (x12(1)*x13(2)-x13(1)*x12(2))/2.;
  // gradients are proportional to the edge
  // vector rotated 90 degrees
  gN(1) = -x12(2);
  gN(2) = +x12(1);
  grad_N.Column(3) = gN/(2*area);
  gN(1) = +x13(2);
  gN(2) = -x13(1);
  grad_N.Column(2) = gN/(2*area);
  // last gradient can be computed from \sum_j grad_N_j = 0
  grad_N.Column(1) = -(grad_N.Column(2)+grad_N.Column(3));
```

```
      // Integrand is constant over element
      ke = grad_N.t() * grad_N * area;

      // Load matrix element on global matrix
      int nod1,nod2,eq1,eq2;
      for (int i1=1; i1<=nel; i1++) {
        nod1 = icone[(e-1)*nel+i1-1];
        if (fixa.find(nod1)!=fixa.end()) continue;
        eq1=id[nod1-1];
        for (int i2=1; i2<=nel; i2++) {
          nod2 = icone[(e-1)*nel+i2-1];
          eq2=id[nod2-1];
          if (fixa.find(nod2)!=fixa.end()) {
            // Fixed nodes contribute to the right hand side
            VecSetValue(b,eq1,-ke(i1,i2)*fixa[nod2],ADD_VALUES);
          } else {
            // Load on the global matrix
            MatSetValue(A,eq1,eq2,ke(i1,i2),ADD_VALUES);
          }
        }
      }

    }
    // Finish assembly.
    ierr = VecAssemblyBegin(b); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(b); CHKERRQ(ierr);

    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
  }

  // Create SLES and set options
```

```
    ierr = SLESCreate(PETSC_COMM_WORLD,&sles); CHKERRQ(ierr);
    ierr = SLESSetOperators(sles,A,A,
                            DIFFERENT_NONZERO_PATTERN); CHKERRQ(ierr);
    ierr = SLESGetKSP(sles,&ksp); CHKERRQ(ierr);
    ierr = SLESGetPC(sles,&pc); CHKERRQ(ierr);

    ierr = KSPSetType(ksp,KSPGMRES); CHKERRQ(ierr);
    // This only works for one processor only
    //  ierr = PCSetType(pc,PCLU); CHKERRQ(ierr);
    ierr = PCSetType(pc,PCJACOBI); CHKERRQ(ierr);
    ierr = KSPSetTolerances(ksp,1e-6,1e-6,1e3,100); CHKERRQ(ierr);
    int its;
    ierr = SLESSolve(sles,b,u,&its);

    // Print vector on screen
    ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);

    // Cleanup. Free memory
    delete[] e_indx;
    delete[] id_indx;
    PetscFinalize();

}
```

SNES: non-linear solvers

```c
/*$Id curspar-1.0.0-15-gabee420 Thu Jun 14 00:46:44 2007 -0300$*/

static char help[] =
  "Newton's method to solve a combustion-like 1D problem.\n";

#include "petscsnes.h"

/*
   User-defined routines
*/
extern int resfun(SNES,Vec,Vec,void*);
extern int jacfun(SNES,Vec,Mat*,Mat*,MatStructure*,void*);

struct SnesCtx {
  int N;
  double k,c,h;
};

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
{
  SNES         snes;         /* nonlinear solver context */
  SLES         sles;         /* linear solver context */
```

```
PC          pc;          /* preconditioner context */
KSP         ksp;         /* Krylov subspace method context */
Vec         x,r;        /* solution, residual vectors */
Mat         J;           /* Jacobian matrix */
int         ierr,its,size;
PetscScalar pfive = .5,*xx;
PetscTruth flg;
SnesCtx ctx;
int N = 10;
ctx.N = N;
ctx.k = 0.1;
ctx.c = 1;
ctx.h = 1.0/N;

PetscInitialize(&argc,&argv,(char *)0,help);
ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");

// Create nonlinear solver context
ierr = SNESCreate(PETSC_COMM_WORLD,&snes); CHKERRQ(ierr);
ierr = SNESSetType(snes,SNESLS); CHKERRQ(ierr);

// Create matrix and vector data structures;
// set corresponding routines

// Create vectors for solution and nonlinear function
ierr = VecCreateSeq(PETSC_COMM_SELF,N+1,&x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&r); CHKERRQ(ierr);
double scal = 0.9;
ierr = VecSet(&scal,x); CHKERRQ(ierr);

ierr = MatCreateMPIAIJ(PETSC_COMM_SELF,PETSC_DECIDE,
                         PETSC_DECIDE,N+1,N+1,
                          1,NULL,0,NULL,&J);CHKERRQ(ierr);

  ierr = SNESSetFunction(snes,r,resfun,&ctx); CHKERRQ(ierr);
  ierr = SNESSetJacobian(snes,J,J,jacfun,&ctx); CHKERRQ(ierr);

  ierr = SNESSolve(snes,x,&its);CHKERRQ(ierr);
  Vec f;
  ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
  ierr = SNESGetFunction(snes,&f,0,0);CHKERRQ(ierr);
  double rnorm;
  ierr = VecNorm(r,NORM_2,&rnorm);

  ierr = PetscPrintf(PETSC_COMM_SELF,
                     "number of Newton iterations = "
                     "%d, norm res %g\n",
                     its,rnorm);CHKERRQ(ierr);

  ierr = VecDestroy(x);CHKERRQ(ierr);
  ierr = VecDestroy(r);CHKERRQ(ierr);
  ierr = SNESDestroy(snes);CHKERRQ(ierr);

  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}

/* ------------------------------------------------------------------- */
#undef __FUNCT__
#define __FUNCT__ "resfun"
int resfun(SNES snes,Vec x,Vec f,void *data)
{
  double *xx,*ff;
  SnesCtx &ctx = *(SnesCtx *)data;
  int ierr;
  double h = ctx.h;

  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
  ierr = VecGetArray(f,&ff);CHKERRQ(ierr);

  ff[0] = xx[0];
  ff[ctx.N] = xx[ctx.N];

  for (int j=1; j<ctx.N; j++) {
    double xxx = xx[j];
    ff[j] = xxx*(0.5-xxx)*(1.0-xxx);
    ff[j] += ctx.k*(-xx[j+1]+2.0*xx[j+1]-xx[j-1])/(h*h);
  }

  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);
  return 0;
}

/* ------------------------------------------------------------------- */
#undef __FUNCT__
#define __FUNCT__ "resfun"
int jacfun(SNES snes,Vec x,Mat* jac,Mat* jac1,
           MatStructure *flag,void *data) {
  double *xx, A;
  SnesCtx &ctx = *(SnesCtx *)data;
  int ierr, j;

  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
  ierr = MatZeroEntries(*jac);

  j=0; A = 1;
  ierr = MatSetValues(*jac,1,&j,1,&j,&A,
                      INSERT_VALUES); CHKERRQ(ierr);

  j=ctx.N; A = 1;
  ierr = MatSetValues(*jac,1,&j,1,&j,&A,
                      INSERT_VALUES); CHKERRQ(ierr);

  for (j=1; j<ctx.N; j++) {
    double xxx = xx[j];
    A = (0.5-xxx)*(1.0-xxx) - xxx*(1.0-xxx) - xxx*(0.5-xxx);
    ierr = MatSetValues(*jac,1,&j,1,&j,&A,INSERT_VALUES);
    CHKERRQ(ierr);
  }
  ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  return 0;
}
```

OpenMP

OPTIONAL Assignement Nbr. 7

- **Escribir un programa en paralelo usando OpenMP para resolver el *problema del PNT*. Probar las diferentes opciones de scheduling. Calcular los speedup y discutir. Escribir una versión tipo *dynamic,chunk* pero implementada con un *lock*. Idem, con *critical section*.**
- **Escribir un programa en paralelo usando OpenMP para resolver el *problema del TSP*. Usar una region crítica o semáforo para recorrer los caminos parciales, y que cada thread recorra los caminos totales derivados. Usar un lock para no provocar *race condition* en la actualización de la distancia mínima actual.**