Cálculo científico en computadoras paralelas. MPI y PETSc

Mario Storti

Centro Internacional de Métodos Numéricos
en Ingeniería - CIMEC
INTEC, (CONICET-UNL), Santa Fe, Argentina
<mstorti@intec.unl.edu.ar>
http://www.cimec.com.ar

(document-version "curso-0.1.10")

August 2, 2004

Modelos de computación paralela

- **Modelo conceptual de qué tipo de operaciones se disponen para programar.**
- **No incluye la sintaxis específica de una libería o lenguaje.**
- **Tampoco depende del hardware específico.**
- **Casi todos los modelos que discutiremos se pueden implementar en cualquier máquina paralela. Sin embargo, la eficiencia de la implementación puede variar. (Ejemplo: *"memoria compartida"* no es el modelo más apto para un cluster de PC's).**

Modelos de computación paralela (cont.)

- **Paralelismo de datos: Procesadores vectoriales, aplica una operación dada a un gran conjunto de datos.**
- **SIMD: *"Single Instruction Multiple Data"*,**

Qué es MPI?

- ***"Message Passing Interface"***
- **Paso de mensajes**
  - ▷ **Cada proceso es un programa secuencial por separado**
  - ▷ **Todos los datos son privados**
  - ▷ **La comunicación se hace via llamadas a funciones**
  - ▷ **Se usa un lenguje secuencial estándar: Fortran, C, (F90, C++)...**
- **MPI es SPMD (Single Program Multiple Data)**

Razones para procesar en paralelo

- **Más potencia de cálculo**
- **Máquinas con relación de costo eficientes a partir de componentes económicos (COTS)**
- **Empezar con poco, con expectativa de crecimiento**

Necesidades del cálculo en paralelo

- **Portabilidad (actual y futura)**
- **Escalabilidad (hardware y software)**

Referencias

- ***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 (MPI-1)**
- ***MPI-2: Extensions to the Message-Passing Interface* November 1996.**

Otras librerías de paso de mensajes

- **PVM (Parallel Virtual Machine)**
  - ▷ **Tiene una interface interactiva de manejo de procesos**
  - ▷ **Puede agregar o borrar hosts en forma dinámica**
- **P4: modelo de memoria compartida**
- **productos comerciales obsoletos, proyectos de investigación (a veces específicos para una dada arquitectura)**

MPICH

Es una implementación portable de MPI escrita en los Argon National Labs (ANL) (USA)

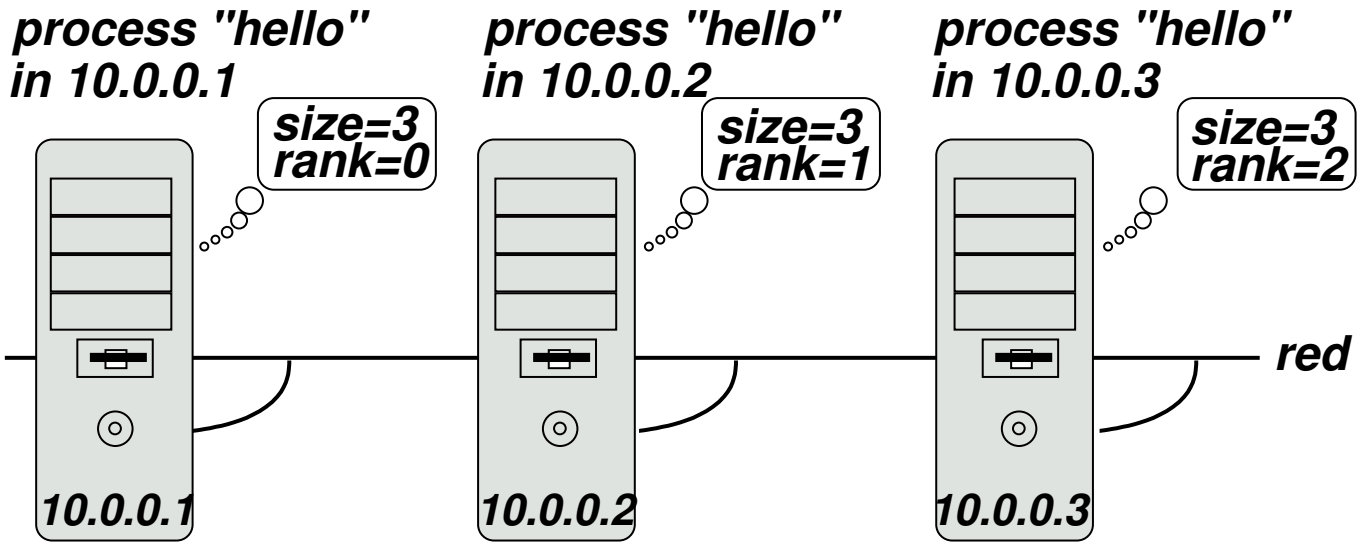
- **MPE: Una librería de rutinas gráficas, interface para debugger, producción de logfiles**
- **MPIRUN: script portable para lanzar procesos paralelos**
- **implementada en P4**
- **Otra implementación: LAM MPI**

Hello world en C

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

- Todos los programas empiezan con `MPI_Init()` y terminan con `MPI_Finalize()`.
- `MPI_Comm_rank()` retorna en `rank`, el número de proceso que está corriendo. `MPI_Comm_size()` la cantidad total de procesadores.

Hello world en C (cont.)



- **Al momento de correr el programa (ya veremos como) una copia del programa empieza a ejecutarse en cada uno de los nodos seleccionados. En la figura corre en `size=3` nodos.**
- **Cada proceso obtiene un número de `rank` individual, entre 0 y `size-1`.**
- **En general podría haber más de un *"proceso"* en cada *"procesador"*.**

Hello world en C (cont.)

- Si compilamos y generamos un ejecutable `hello.bin`, entonces al correrlo obtenemos la salida estándar.

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

- Para correrlo en varias máquinas generamos un archivo `machi.dat` con nombres de procesadores uno por línea.

```
[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]$$
```

El script `mpirun`, que es parte de la distribución de MPICH, lanza una copia de `hello.bin` en el procesador desde donde se llamó a `mpirun` y dos procesos en las primeras dos líneas de `machi.dat`.

I/O

- **Es normal que cada uno de los procesos *"vea"* el mismo directorio via NFS.**
- **Cada uno de los procesos puede abrir sus propios archivos para lectura o escritura, con las mismas reglas que deben respetar varios procesos en un sistema UNIX.**
- **Varios procesos pueden abrir el mismo archivo para lectura.**
- **Normalmente sólo el proceso 0 lee de `stdin`.**
- **Todos los procesos pueden escribir en `stdout`, pero la salida puede salir mezclada (no hay un orden temporal definido).**

Hello world en Fortran

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

Estrategia master/slave con SPMD (en C)

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

Formato de llamadas MPI

- **C:**
  ```
  int ierr=MPI_Xxxxx(parameter,....); ó
  MPI_Xxxxx(parameter,....);
  ```
- **Fortran:**
  ```
  CALL MPI_XXXX(parameter,....,ierr);
  ```

Códigos de error

- Los códigos de error son raramente usados
- Uso de los códigos de error:

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

MPI is small - MPI is large!!

Se pueden escribir programas medianamente complejos con sólo 6 funciones

- **`MPI_Init` - Se usa una sola vez al principio para inicializar**
- **`MPI_Comm_size` Identificar cuantos procesos están disponibles**
- **`MPI_Comm_rank` Identifica mi número de proceso dentro del total**
- **`MPI_Finalize` Ultima función de MPI a llamar, termina MPI**
- **`MPI_Send` Envía un mensaje a un solo proceso (point to point)**
- **`MPI_Recv` Recibe un mensaje enviado por otro proceso**

MPI is small - MPI is large!! (cont.)

Comunicaciones colectivas

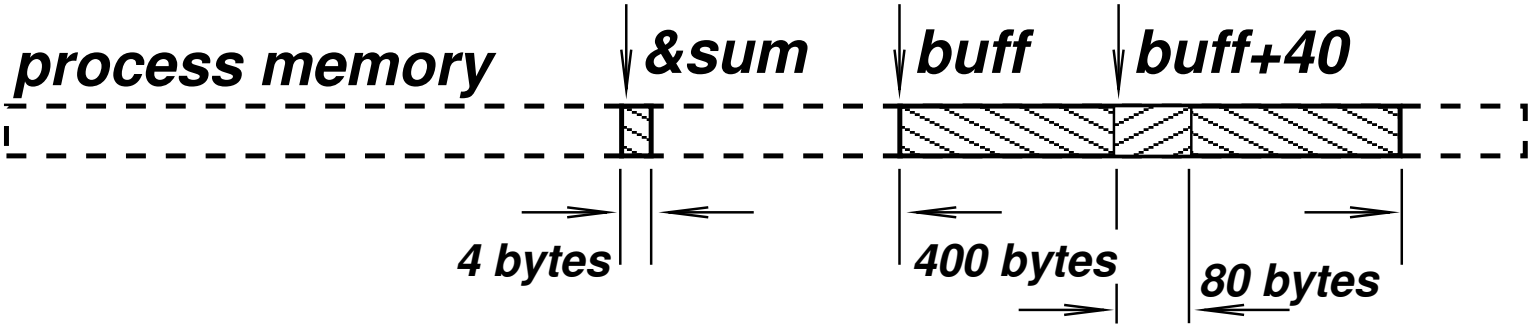
- **`MPI_Bcast` Envía un mensaje a todos los procesos**
- **`MPI_Reduce` Combina datos de todos los procesos en un solo proceso**

El estándar completo de MPI tiene 125 funciones.

Enviar un mensaje

- **Template:**
  ```
  MPI_Send(address, length, type, destination, tag,
  communicator)
  ```
- **C:**
  ```
  ierr = MPI_Send(&sum, 1, MPI_FLOAT, 0, mtag1,
  MPI_COMM_WORLD);
  ```
- **Fortran (notar parámetro extra):**
  ```
  call MPI_SEND(sum, 1, MPI_REAL, 0, mtag1,
  MPI_COMM_WORLD,ierr);
  ```

Enviar un mensaje (cont.)




```
1  int buff[100];
2  // Llenar buff..
3  for (int j=0; j<100; j++) buff[j] = j;
4  ierr = MPI_Send(buff, 100, MPI_INT, 0, mtag1,
5         MPI_COMM_WORLD);
6
7  int sum;
8  ierr = MPI_Send(&sum, 1, MPI_INT, 0, mtag2,
9         MPI_COMM_WORLD);
10
11 ierr = MPI_Send(buff+40, 20, MPI_INT, 0, mtag2,
12        MPI_COMM_WORLD);
13
14 ierr = MPI_Send(buff+80, 40, MPI_INT, 0, mtag2,
15        MPI_COMM_WORLD); // Error! se extiende mas alla
16                        // del final de buff
```

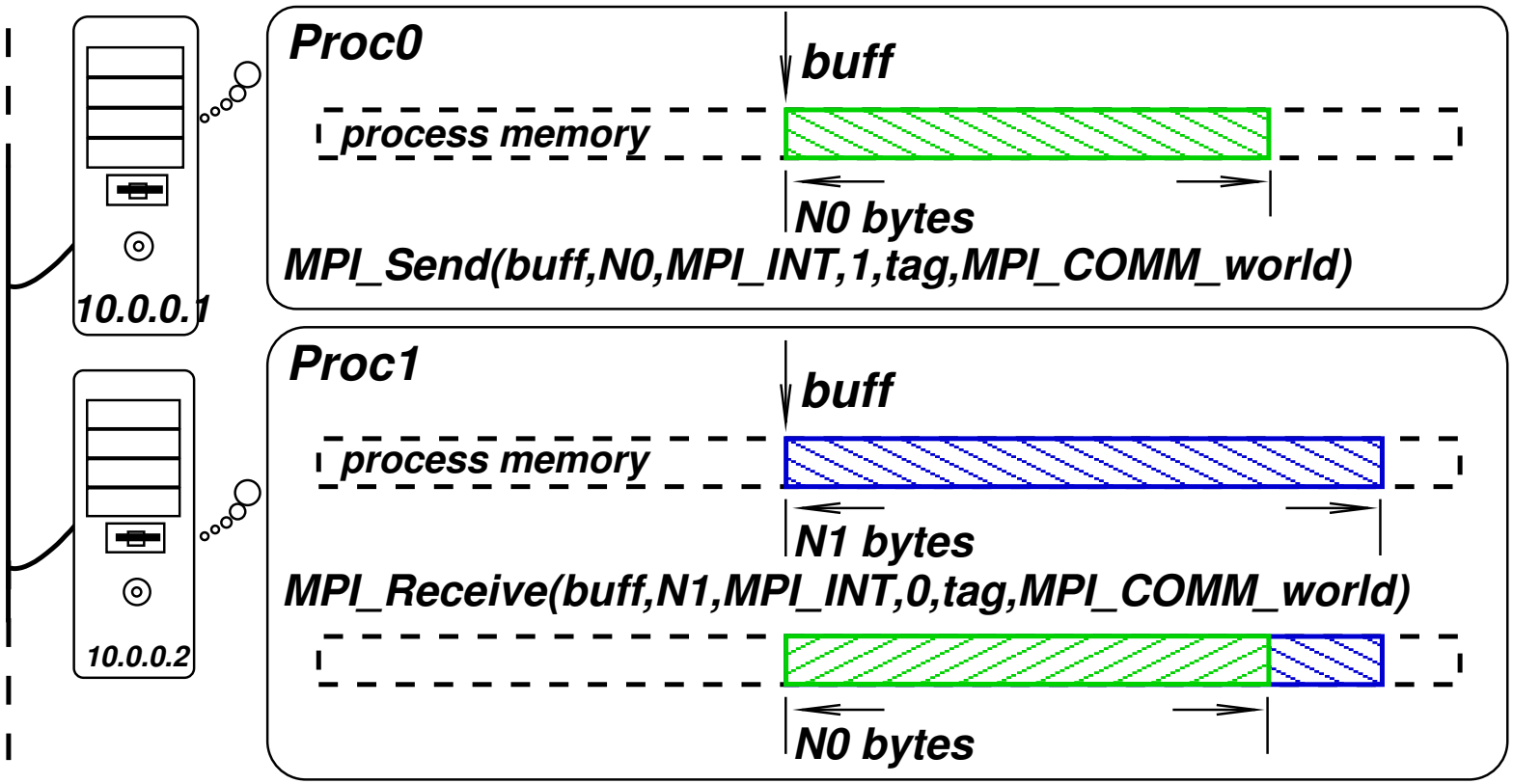
Recibir un mensaje

- **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 (notar parámetro extra):**
  ```
  call MPI_RECV(result, 1, MPI_REAL, MPI_ANY_SOURCE,
  mtag1, MPI_COMM_WORLD, status, ierr)
  ```

Recibir un mensaje (cont.)

- `(address, length)` buffer de recepción
- `type` tipo estándar de MPI:
  C: `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_INT`, `MPI_CHAR`
  Fortran: `MPI_REAL`, `MPI_DOUBLE_PRECISION`, `MPI_INTEGER`, `MPI_CHARACTER`
- `(source,tag,communicator)`: selecciona el mensaje
- `status` permite ver los datos del mensaje efectivamente recibido (p.ej. longitud)

Recibir un mensaje (cont.)



OK si `N1>=N0`

Parámetros de las send/receive (cont.)

- **`tag` identificador del mensaje**
- **`communicator` grupo de procesos, por ejemplo `MPI_COMM_WORLD`**
- **`status` fuente, tag y longitud del mensaje recibido**
- **Comodines (wildcards): `MPI_ANY_SOURCE`, `MPI_ANY_TAG`**

Status

`status (source, tag, length)`

- en C, una estructura

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

- en Fortran, un arreglo de enteros

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

Comunicación punto a punto

A cada `send` debe corresponder un `recv`

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

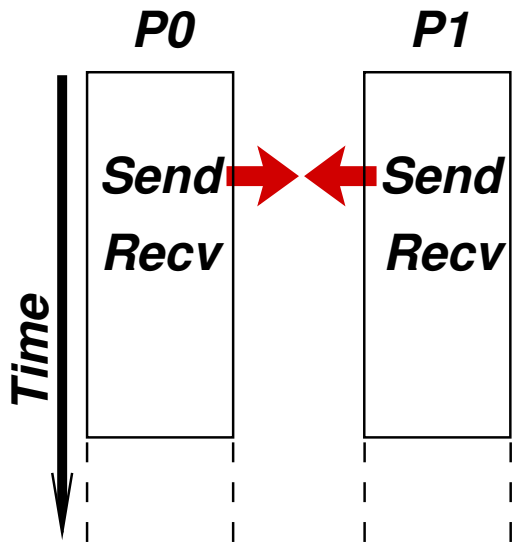
Cuando un mensaje es recibido?

Cuando encuentra un receive que concuerda en cuanto a la *"envoltura"* (*"envelope"*) del mensaje

`envelope = source/destination, tag, communicator`

- **size(receive buffer) $<$ size(data sent) $\longrightarrow$ error**
- **size(receive buffer) $>$ size(data sent) $\longrightarrow$ OK**
- **tipos no concuerdan $\longrightarrow$ error**

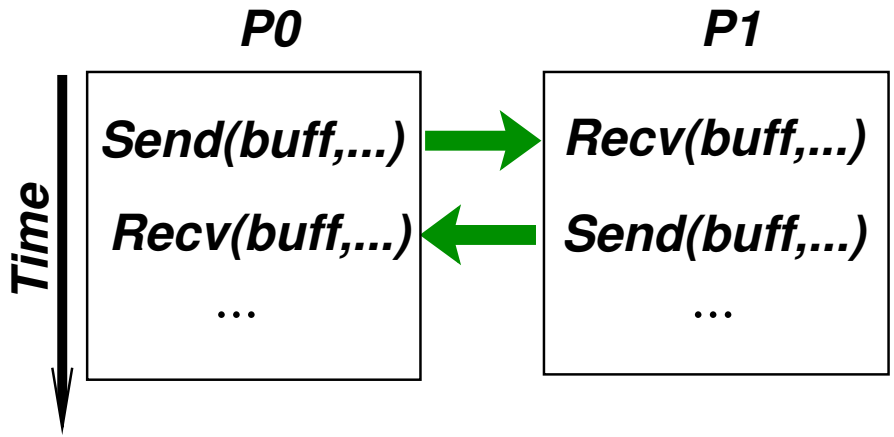
Dead-lock




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

`!myrank`: Lenguaje común para representar al otro proceso ($1 \to 0, 0 \to 1$).
También `1-myrank` o `(myrank? 0 : 1)`

Orden de llamadas correcto

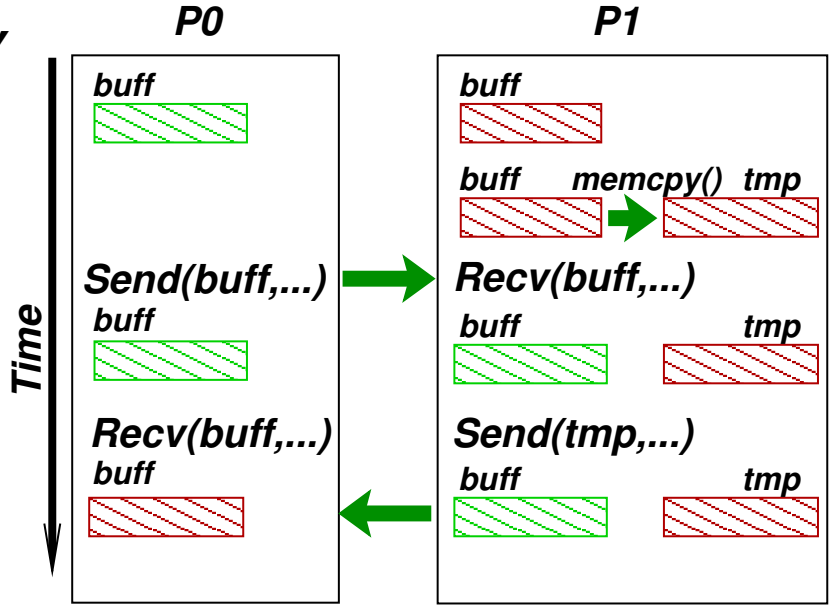



```
1  if (!myrank) {
2     MPI_Send(buff,length,MPI_FLOAT,!myrank,
3                    tag,MPI_COMM_WORLD);
4     MPI_Recv(buff,length,MPI_FLOAT,!myrank,
5                    tag,MPI_COMM_WORLD,&status);
6  } else {
7     MPI_Recv(buff,length,MPI_FLOAT,!myrank,
8                    tag,MPI_COMM_WORLD,&status);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

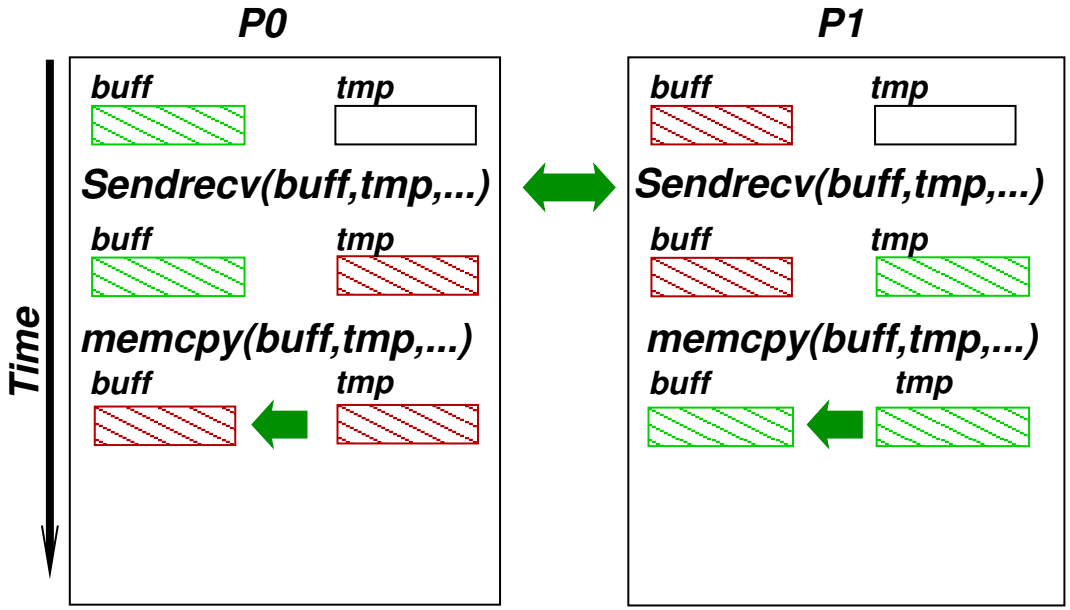
Orden de llamadas correcto (cont.)

Sobre-escribe el buffer de recepción. Necesitamos un buffer temporal `tmp`:

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



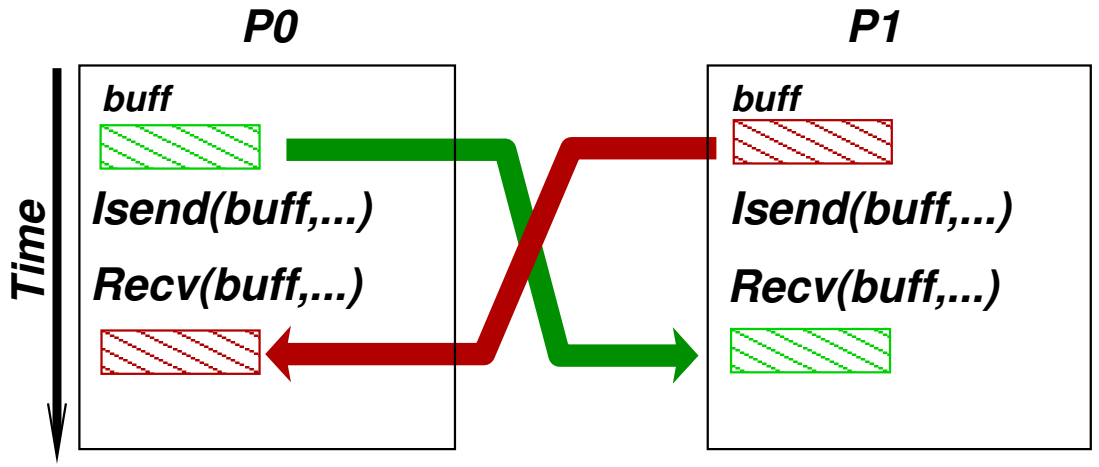
Orden de llamadas correcto (cont.)



- **`MPI_Sendrecv`: recibe y envia a la vez**

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

Orden de llamadas correcto (cont.)

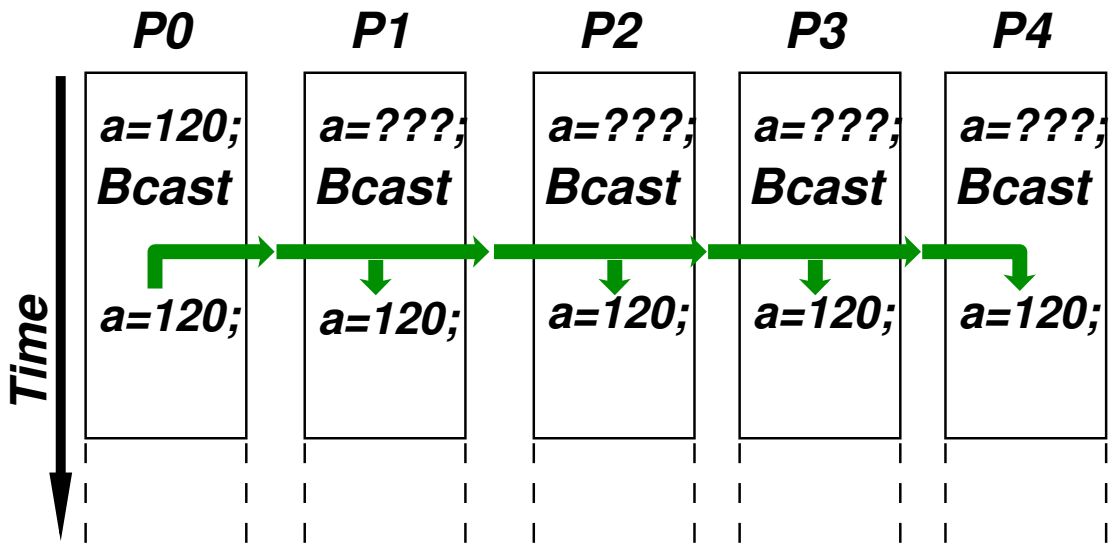


Usar send/receive no bloqueantes

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

- **El código es el mismo para los dos procesos.**
- **No hace falta un buffer auxiliar.**

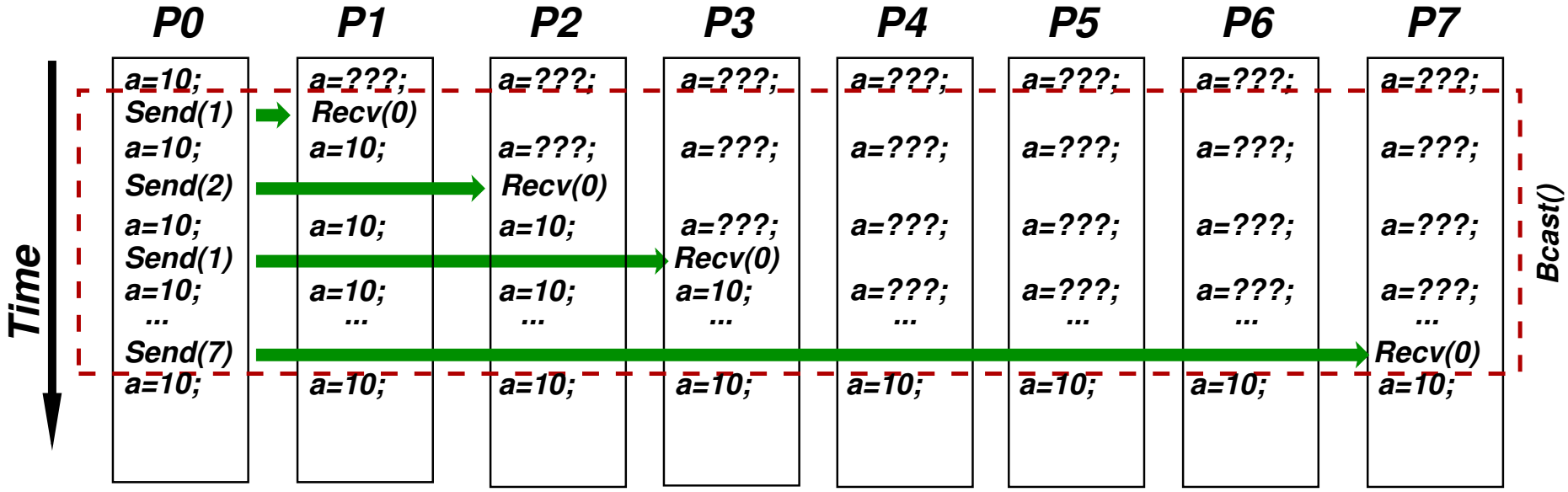
Broadcast de mensajes



- **Template:**
  `MPI_Bcast(address, length, type, source, comm)`
- **C:**
  `ierr = MPI_Bcast(&a, 1, MPI_INT, 0, comm);`
- **Fortran:**
  `call MPI_Bcast(a, 1, MPI_INTEGER, 0, comm,ierr)`

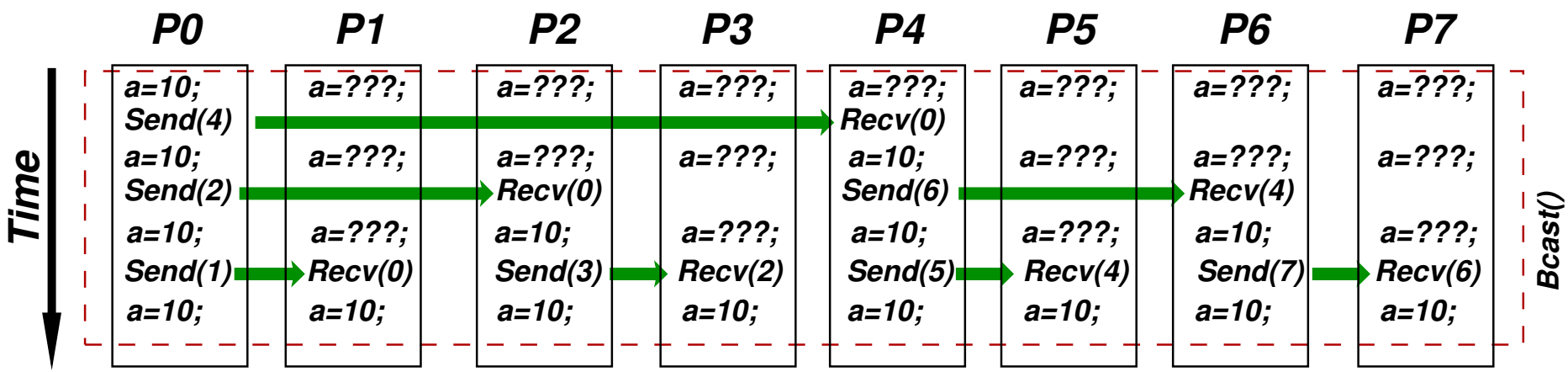
Broadcast de mensajes (cont.)

`MPI_Broadcast()` es conceptualmene equivalente a una serie de Send/Receive, pero puede ser mucho más eficiente.

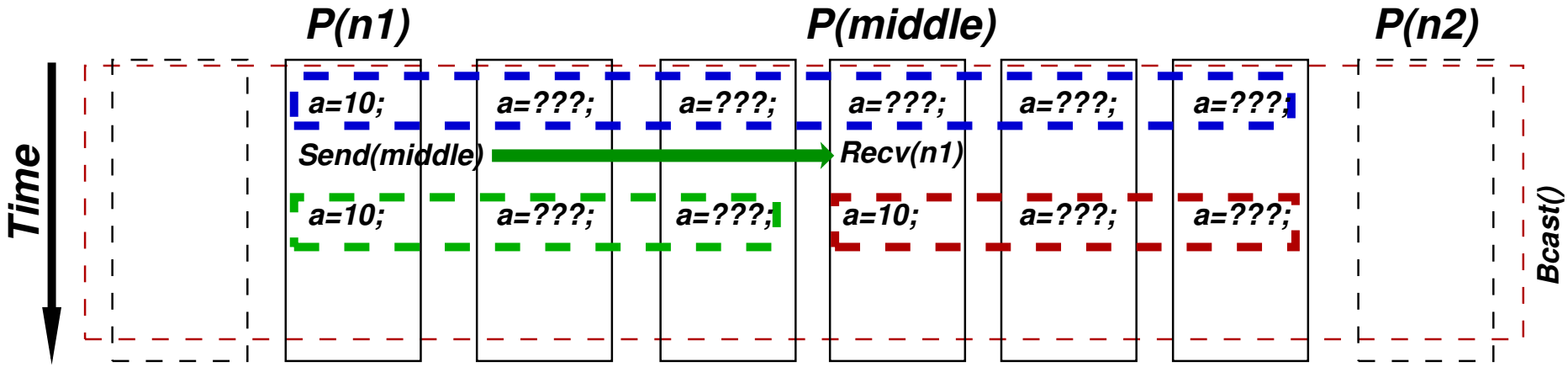



```
1 if (!myrank) {
2   for (int j=1; j<numprocs; j++) MPI_Send(buff,...,j);
3 } else {
4   MPI_Receive(buff,...,0...);
5 }
```

Broadcast de mensajes (cont.)



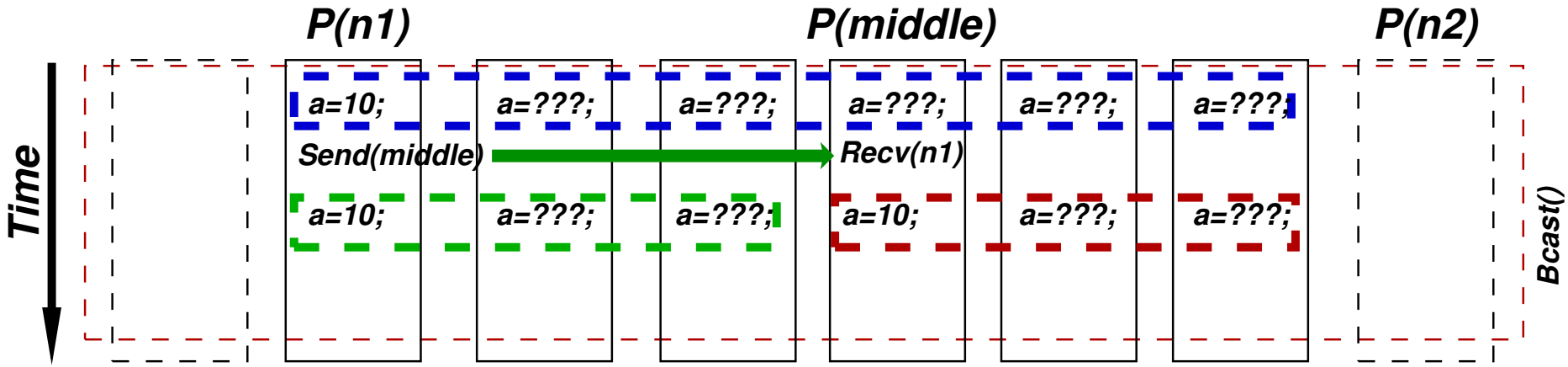
Broadcast de mensajes (cont.)



Implementación eficiente de `MPI_Bcast()` con Send/Receives.

- En todo momento estamos en el procesador `myrank` y tenemos definido un intervalo `myrank`$\in$`[n1,n2)`. Recordemos que `[n1,n2)`=$\{j$ tal que $\mathtt{n1} \leq j < \mathtt{n2}\}$
- Inicialmente `n1=0`, `n2=NP` (número de procesadores).
- En cada paso `n1` le va a enviar a `middle=(n1n2)/2+` y este va a recibir.
- Pasamos a la siguiente etapa manteniendo el rango `[n1,middle)` si `myrank<middle` y si no mantenemos `[middle,n2)`.
- El proceso se detiene cuando `n2-n1==1`

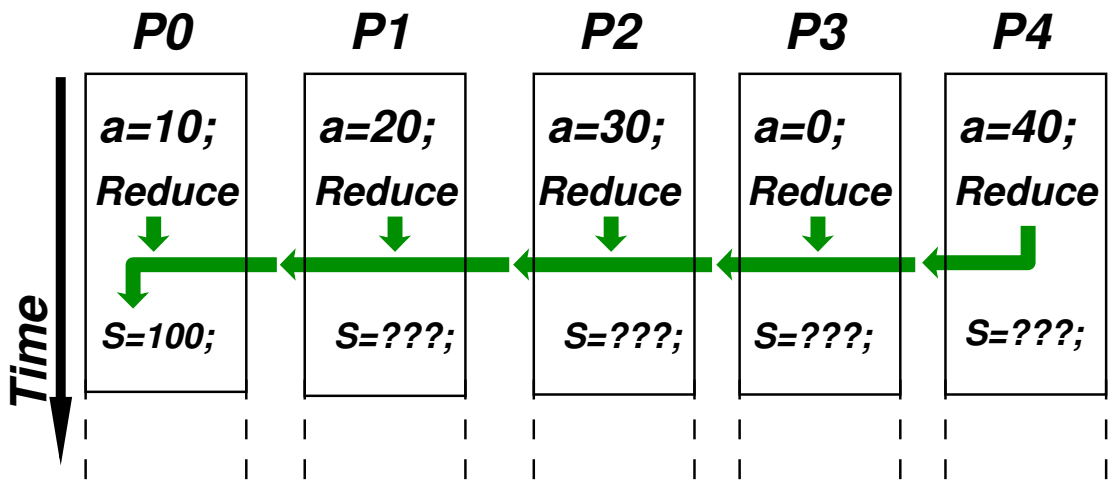
Broadcast de mensajes (cont.)



Seudocódigo:

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

Reducción global



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

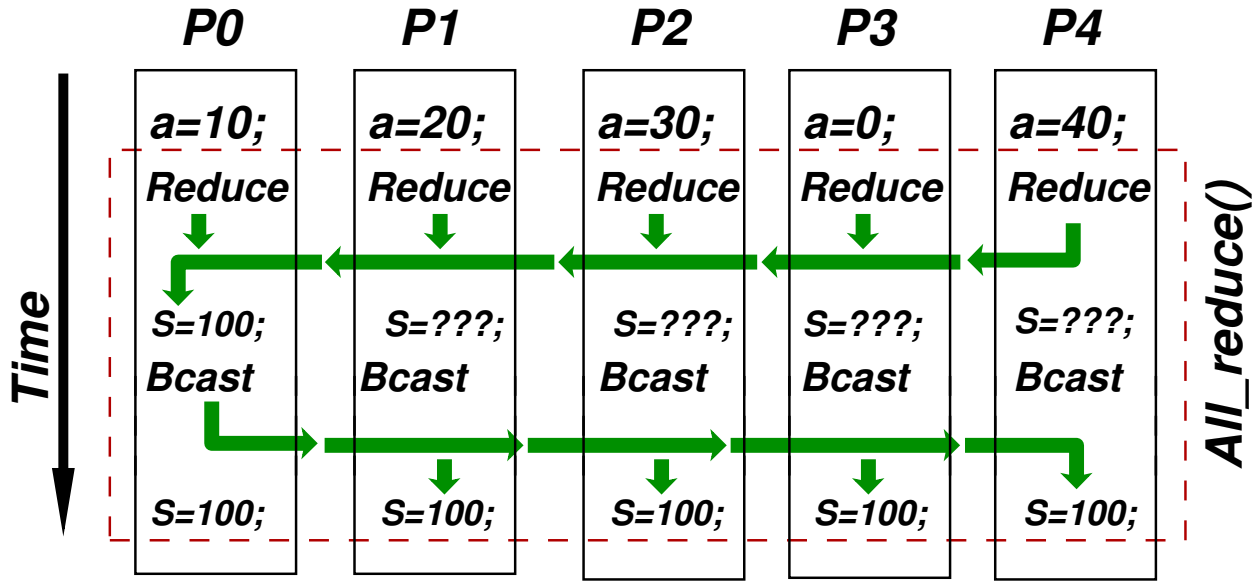
Operaciones asociativas globales de MPI

Operaciones de reducción en general aplican una *"operación binaria asociativa"* a un conjunto de valores. Tipicamente,

- **`MPI_SUM` suma**
- **`MPI_MAX` máximo**
- **`MPI_MIN` mínimo**
- **`MPI_PROD` producto**
- **`MPI_AND` boolean**
- **`MPI_OR` boolean**

No está especificado el orden en que se hacen las reducciones de manera que *debe* ser una *función asociativa* $f(x,y)$, conmutativa o no.

Operaciones asociativas globales de MPI (cont.)



Si el resultado es necesario en todos los procesadores, entonces se debe usar

```
MPI_All_reduce(s_address,r_address, length, type
     operation, comm)
```

Esto es conceptualmente equivalente a un `MPI_Reduce()` seguido de un `MPI_Bcast()`. Atención: `MPI_Bcast()` y `MPI_Reduce()` son llamadas *"colectivas". Todos los procesadores tienen que llamarlas!!*

Otras funciones de MPI

- **Timers**
- **Operaciones de gather y scatter**
- **Librería MPE. No es parte del estándar MPI. Consistente en llamado con MPI. Uso: log-files, gráficos...**

MPI en ambientes Unix

- **MPICH se instala comunmente en `/usr/local/mpi` (`MPI_HOME`).**
- **Para compilar:**
  ```
  > g++ -I/usr/local/mpi/include -o foo.o foo.cpp
  ```
- **Para linkeditar:**
  ```
  > g++ -L/usr/local/mpi/lib -lmpi -o foo foo.o
  ```
- **MPICH provee scripts `mpicc`, `mpif77` etc... que agregan los `-I`, `-L` y librerías adecuados.**

MPI en ambientes Unix (cont.)

- El script `mpirun` sen encarga de lanzar el programa en los hosts.
  Opciones:
  ```
  -np <nro-de-procesadores>
  -nolocal
  -machinefile machi.dat
  ```
- Ejemplo:

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

Lanza `foo` en `node1`, `node2`, `node3` y `node4`.

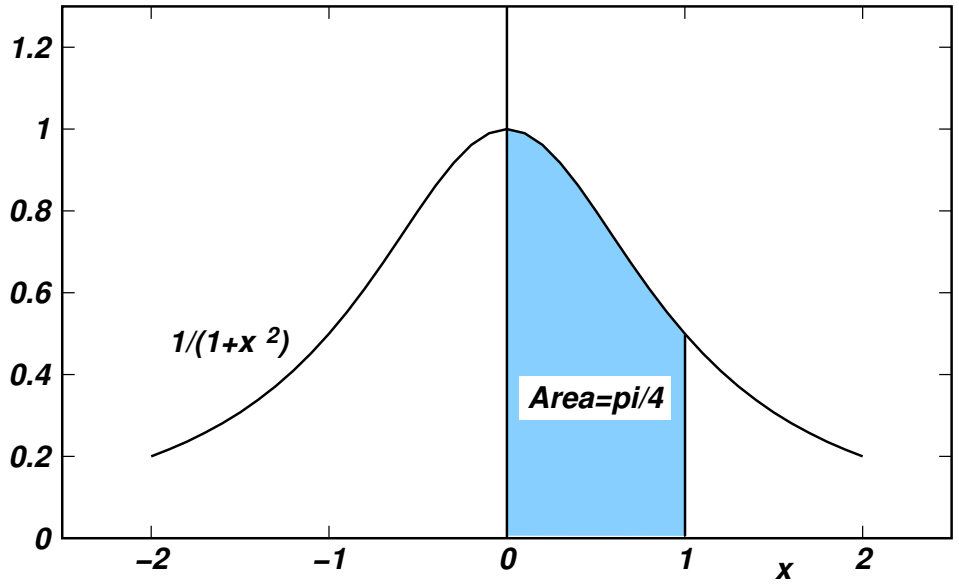
MPI en ambientes Unix (cont.)

- Para que no lanze en el host actual usar `-nolocal`. Ejemplo:
- Ejemplo:

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

Lanza `foo` en `node2`, `node3` y `node4`.

Ejemplo cálculo de Pi por integración simple

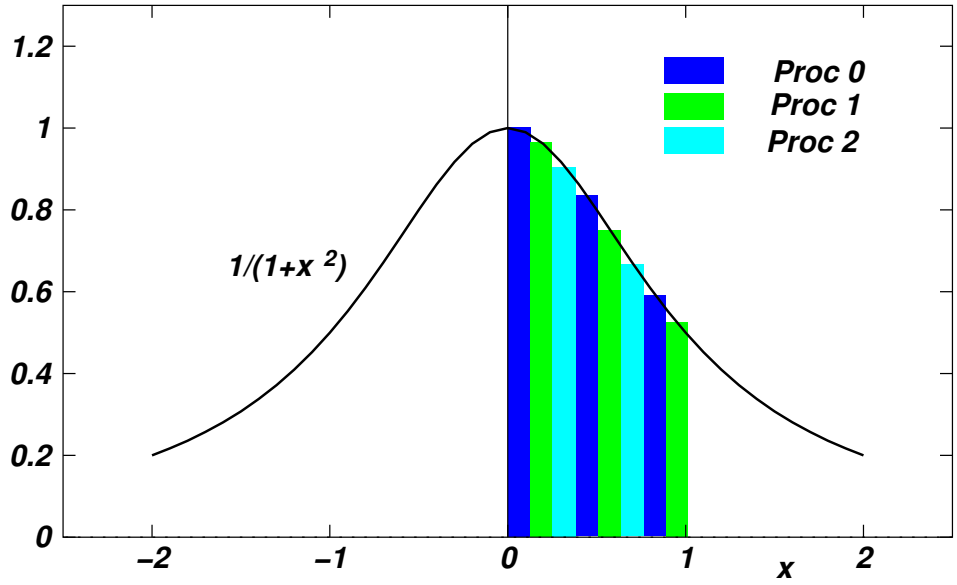


$$\operatorname{atan}(1) = \pi/4$$

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

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

Integración por suma de áreas



- **`numprocs` = número de procesadores**
- n **= número de intervalos (puede ser múltiplo de `numprocs` o no)**
- $h = 1/n$ **= ancho de cada rectángulo**

Integración por suma de áreas (cont.)

```
1  // Inicializacion (rank,size) ...
2  while (1) {
3    // Master (rank==0) lee numero de intervalos ‘n’ ...
4    // Broadcast de ‘n’ a los nodos ...
5    if (n==0) break;
6    // Calcula ‘mypi’, contribucion local a ‘pi’ ...
7    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,
8               MPI_SUM,0,MPI_COMM_WORLD);
9    // Master reporta error entre pi calculado
10   //       y exacto...
11 }
12 MPI_Finalize();
```

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

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

```
67      //  Now this part of the code is running on every node
68      //  and each one shares the same value of n.  But all
69      //  other variables are local to each individual
70      //  process. So each process then calculates the each
71      //  interval size.
72
73      //*********************************************c
74      //         Main Body : Runs on all processors
75      //*********************************************c
76      // even step size h as a function of partions
77      double h = 1.0/double(n);
78      double sum  = 0.0;
79      for (int i=myrank+1; i<=n; i += size) {
80        double x = h * (double(i) - 0.5);
81        sum = sum + f(x);
82      }
83      double pi, mypi = h * sum; // this is the total area
84                                 // in this process,
85                                 // (a partial sum.)
86
87      // Each individual sum should converge also to PI,
88      // compute the max error
89      double error, my_error = fabs(size*mypi-PI);
90      MPI_Reduce(&my_error,&error,1,MPI_DOUBLE,
91                 MPI_SUM,0,MPI_COMM_WORLD);
92
93      //  After each partition of the integral is calculated
94      //  we collect all the partial sums.  The MPI_SUM
95      //  argument is the operation that adds all the values
96      //  of mypi into pi of process 0 indicated by the 6th
97      //  argument.
98      MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
99
```

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

Integración por suma de áreas (cont.)

Reporte de ps:

```
[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/emacsserver
  985       |   \_ /bin/bash
 1732       |   |   \_ ps axfw
 1037       |   \_ /bin/bash -i
 1058       |       \_ xpdf slides.pdf
 1059       |       \_ xfig -library_dir /home/mstorti/CONFIG/xfig-lib-spider
 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]>
```

Conceptos básicos de escalabilidad.

Sea T_1 el tiempo de cálculo en un solo procesador (asumimos que todos los procesadores son iguales),y sea T_n el tiempo en n procesadores. El *"factor de ganancia"* o *"speed-up"* por el uso de cálculo paralelo es

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

En el mejor de los casos el tiempo se reduce en un factor n es decir $T_n > T_1/n$ de manera que

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

La eficiencia η es la relación entre el speedup real obtenido S_n y el teórico, por lo tanto

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

Conceptos básicos de escalabilidad. (cont.)

Supongamos que tenemos una cierta cantidad de tareas elementales W para realizar. En el ejemplo del cálculod e π sería el número de rectángulos a sumar. Si los requerimientos de cálculo son exactamente iguales para todas las tareas y si distribuimos las W tareas en forma exactamente igual en los n procesadores $W_i = W/n_i$ entonces todos los procesadores van a terminar en un tiempo $T_i = W_i/s$ donde s es la *"velocidad de procesamiento"* de los procesadores (que asumimos que son todos iguales).

Conceptos básicos de escalabilidad. (cont.)

Si no hay comunicación, o esta es despreciable, entonces

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

mientras que en un sólo procesador tardaría

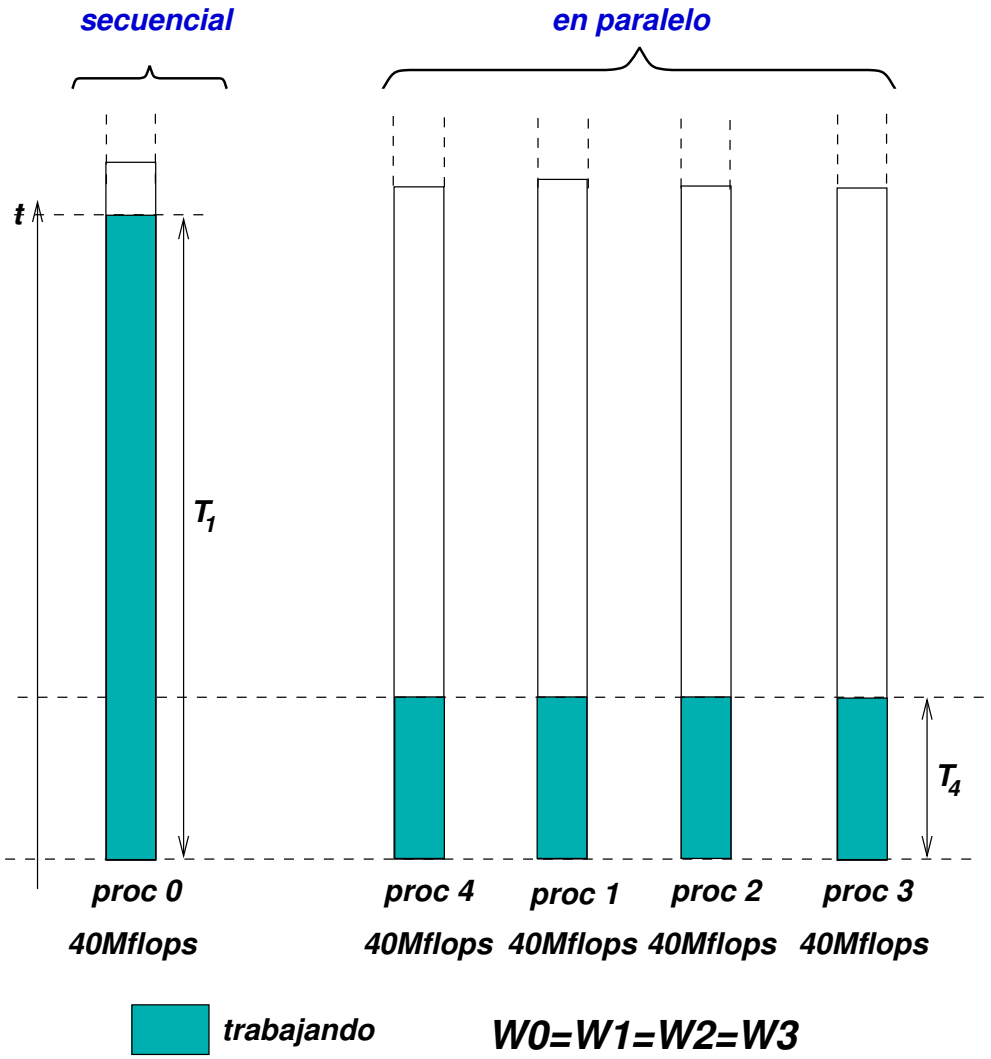
$$T_1 = \frac{W}{s}$$

De manera que

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

Con lo cual el speedu-up es igual al teórico (eficiencia $\eta = 1$).

Conceptos básicos de escalabilidad. (cont.)



Conceptos básicos de escalabilidad. (cont.)

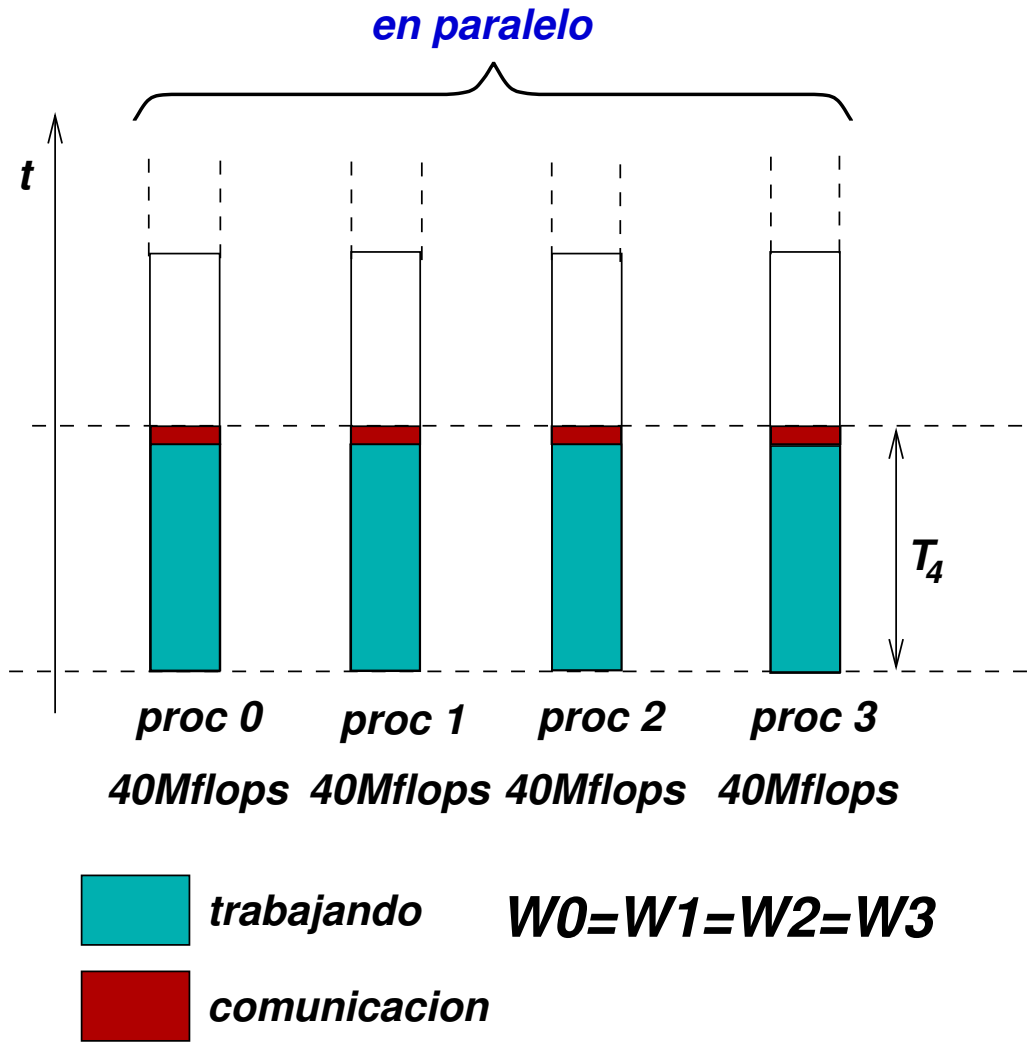
Si el tiempo de comunicación no es despreciable

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

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

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

Conceptos básicos de escalabilidad. (cont.)



Conceptos básicos de escalabilidad. (cont.)

Decimos que una implementación paralela es *"escalable"* si podemos mantener la eficiencia η por encima de un cierto valor umbral $\bar{\eta}$ al hacer crecer el número de procesadores.

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

Si pensamos en el ejemplo de cálculo de π el tiempo total de cálculo se mantiene constante, pero el tiempo total de comunicación crece con n ya que como sólo hay que mandar un doble, el tiempo de comunicación es básicamente la latencia por el número de procesadores, de manera que así planteado *la implementación no es escalable*.

En general esto ocurre siempre, para una dada cantidad fija de trabajo a realizar W, si aumentamos el número de procesadores entonces los tiempos de comunicación se van incrementando y no existe ninguna implementación escalable.

Conceptos básicos de escalabilidad. (cont.)

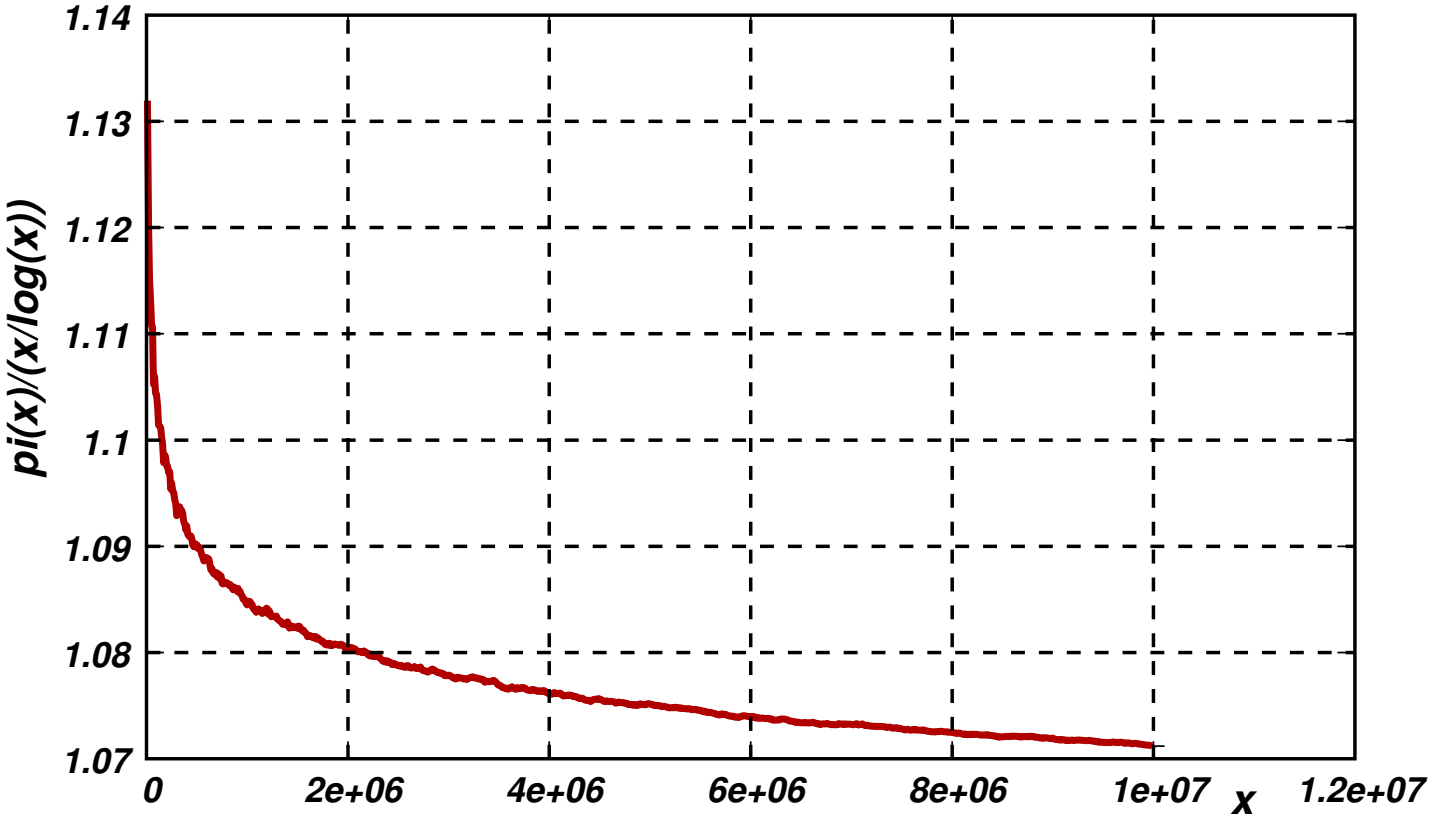
Pero si podemos mantener un cierto nivel de eficiencia prefijado si incrementamos el tamaño global del problema. Si W es el trabajo a realizar (en el ejemplo del cálculo de π la cantidad de intervalos), y hacemos $W = n\bar{W}$, es decir mantenemos el número de intervalos por procesador constante, entonces el tiempo total de cálculo también crece con n y la eficiencia se mantiene acotada.

Decimos que la implementación es escalable en el *"límite termodinámico"* si podemos mantener un cierto nivel de eficiencia para $n \to \infty$ y $W \sim n \to \infty$. Básicamente esto quiere decir que podemos resolver problemas cada vez más grandes en el mismo intervalo de tiempo aumentando el número de procesadores.

Prime number theorem

El PNT dice que la cantidad $\pi(x)$ de números primos menores que x es, asintóticamente

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



Prime number theorem (cont.)

La forma más básica de verificar si un número n es primo es dividirlo por todos los números desde 2 hasta $\mathrm{floor}(\sqrt{n})$. Si no encontramos ningún divisor, entonces es primo

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

De manera que `is_prime(n)` es (en el peor caso) $O(\sqrt{n})$. Como el número de dígitos de n es $n_d = \mathrm{ceil}(\log 10n)$, determinar si n es primo es $O(10^{n_d/2})$ es decir que es *"no polinomial en el número de dígitos n_d"*.

Calcular $\pi(n)$ es entonces $\sim \sum_{n'=2}^{n} \sqrt{n'} \sim n^{1.5}$. Es interesante como ejercicio de cálculo en paralelo, ya que el costo de cada cálculo individual es muy variable y en promedio va creciendo con n.

PNT: Versión secuencial

```
1  //$Id: primes1.cpp,v 1.1 2004/07/23 03:50:49 mstorti Exp $
2  #include <mpi.h>
3  #include <cstdio>
4  #include <cmath>
5
6  int is_prime(int n) {
7    int m = int(sqrt(n));
8    for (int j=2; j<=m; j++)
9      if (n % j ==0) return 0;
10   return 1;
11 }
12
13 // Sequential version
14
15 int main(int argc, char **argv) {
16   MPI_Init(&argc,&argv);
17
18   int  myrank, size;
19   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
20   MPI_Comm_size(MPI_COMM_WORLD,&size);
21
22   int n=2, primes=0,chunk=10000;
23   while(1) {
24     if (is_prime(n++)) primes++;
25     if (!(n % chunk)) printf("%d primes<%d\n",primes,n);
26   }
27
28   MPI_Finalize();
29 }
```

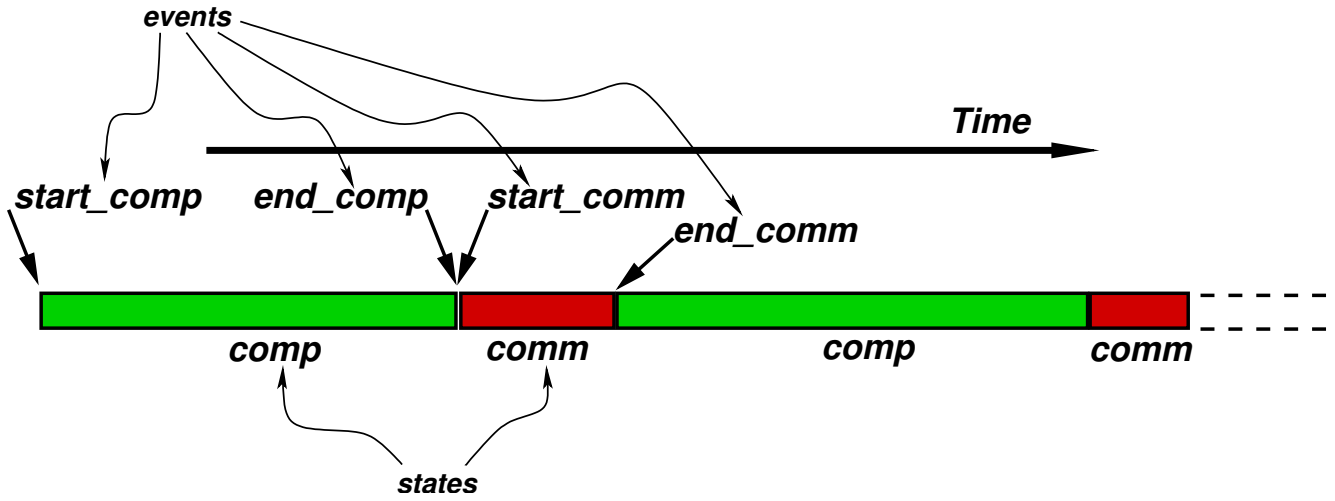
PNT: Versión paralela

- **Chunks de longitud fija.**
- **Cada proc. procesa un chunk y acumulan usando `MPI_Reduce()`.**

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

```
25      for (int n=n1; n<n2; n++) {
26        if (is_prime(n)) primesh++;
27      }
28      MPI_Reduce(&primesh,&primes,1,MPI_INT,
29                 MPI_SUM,0,MPI_COMM_WORLD);
30      n1 += size*chunk;
31      if (!myrank) printf("pi(%d) = %d\n",n1,primes);
32    }
33
34    MPI_Finalize();
35  }
```

MPE Logging



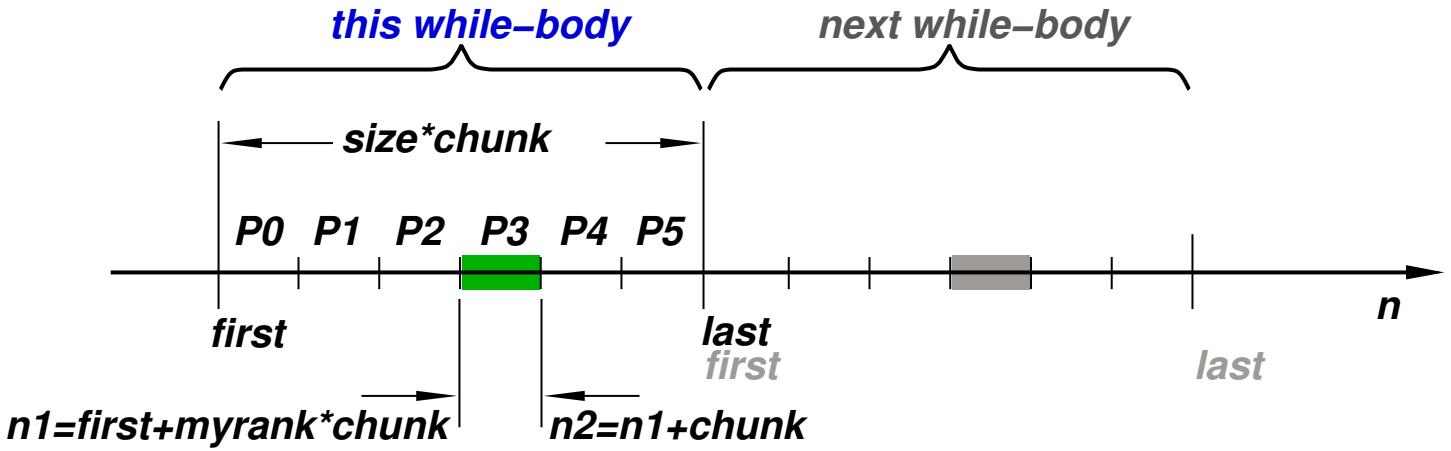
- **Se pueden definir *"estados"*. Típicamente queremos saber cuánto tiempo estamos haciendo *cálculo* (`comp`) y cuanto *comunicación* (`comm`).**
- **Los estados están delimitados por *"eventos"* de muy corta duración (*"atḿicos"*). Típicamente definimos dos eventos para cada estado**
  - ▷ **estado `comp` =** $\{t \,/\, \texttt{start_comp} < t < \texttt{end_comp}\}$
  - ▷ **estado `comm` =** $\{t \,/\, \texttt{start_comm} < t < \texttt{end_comm}\}$

MPE Logging (cont.)

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

- **Es difícil separar *comunicación* de *sincronización*.**

PNT: Versión paralela con MPE logging




```
1   // Counts primes en [0,N)
2   first = 0;
3   while (1) {
4     // Entre todos los procs. verifican [first,last)
5     last = first + size*chunk;
6     n1 = first + myrank*chunk;
7     n2 = n1 + chunk;
8     if (n2>last) n2=last;
9     primesh = /* # de primos en [n1,n2) ... */;
10    // Allreduce 'primesh' to 'primes' ...
11    first += size*chunk;
12    if (last>N) break;
13  }
14    // finalize ...
```

PNT: Versión paralela con MPE logging (cont.)

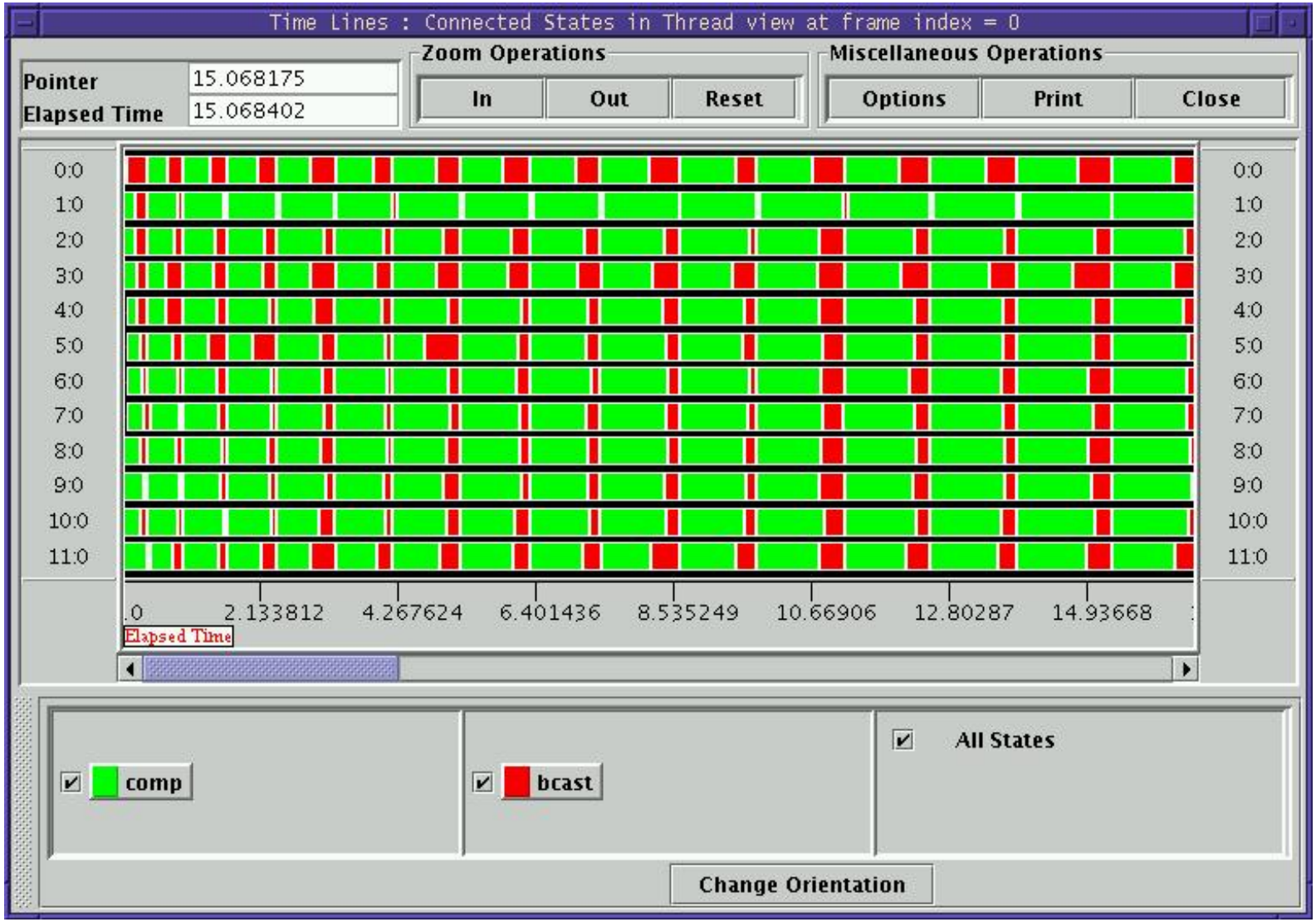
```
1  //$Id: primes3.cpp,v 1.4 2004/07/23 22:51:31 mstorti Exp $
2  #include <mpi.h>
3  #include <mpe.h>
4  #include <cstdio>
5  #include <cmath>
6
7  int is_prime(int n) {
8    int m = int(sqrt(n));
9    for (int j=2; j<=m; j++)
10     if (!(n % j)) return 0;
11   return 1;
12 }
13
14 int main(int argc, char **argv) {
15   MPI_Init(&argc,&argv);
16   MPE_Init_log();
17
18   int  myrank, size;
19   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
20   MPI_Comm_size(MPI_COMM_WORLD,&size);
21   int start_comp = MPE_Log_get_event_number();
22   int end_comp = MPE_Log_get_event_number();
23   int start_bcast = MPE_Log_get_event_number();
24   int end_bcast = MPE_Log_get_event_number();
25
26   int n2, primesh=0, primes, chunk=200000,
27     n1, first=0, last;
28
```

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

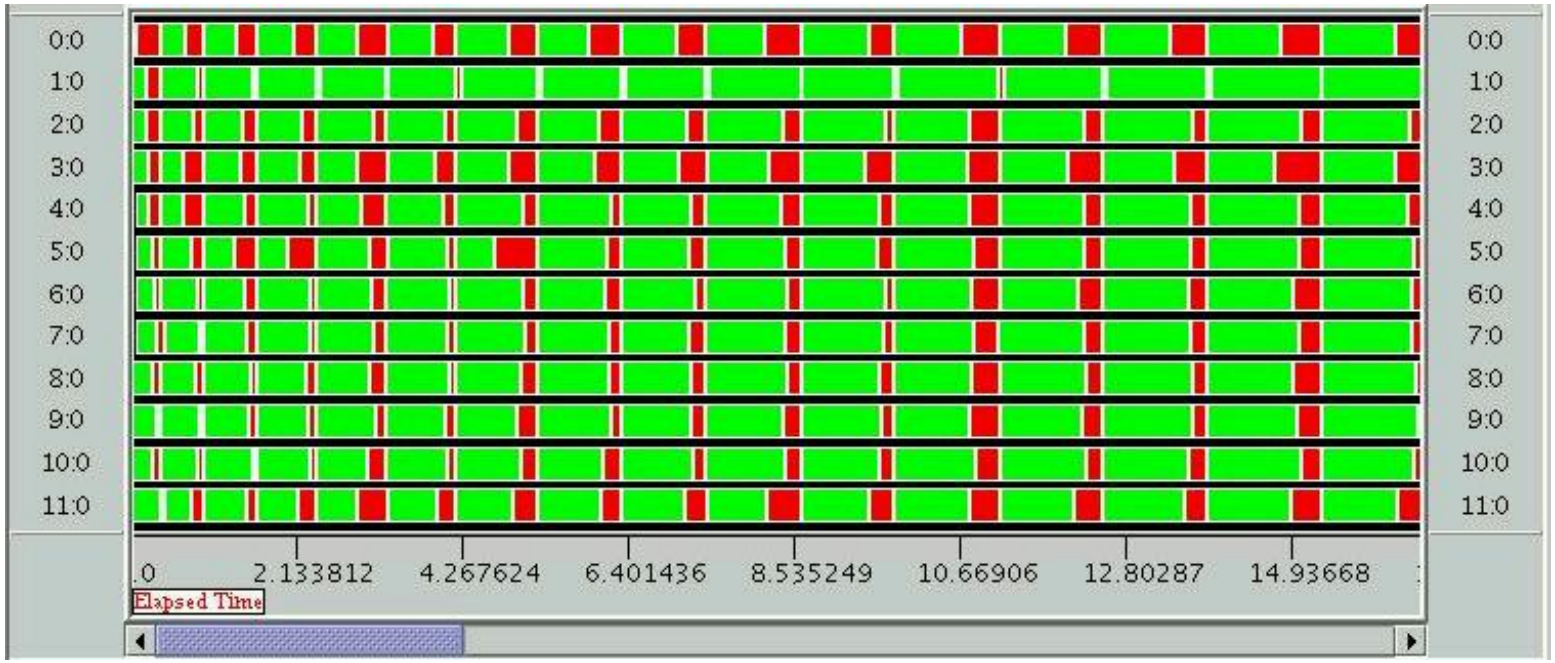
Uso de jumpshot

- **"*Jumpshot*" es un utilitario free que viene con MPICH y permite ver los logs de MPE en forma gráfica.**
- **Configurar MPI (versión 1.2.5.2) con `--with-mpe`, e instalar alguna versión de Java, puede ser `j2sdk-1.4.2_04-fcs` de `www.sun.org`.**
- **Al correr el programa crea un archivo `primes.clog`**
- **Convertir a formato "*SLOG*" con `> clog2slog primes.clog`**
- **Correr `> jumpshot primes.slog &`**

Uso de jumpshot (cont.)

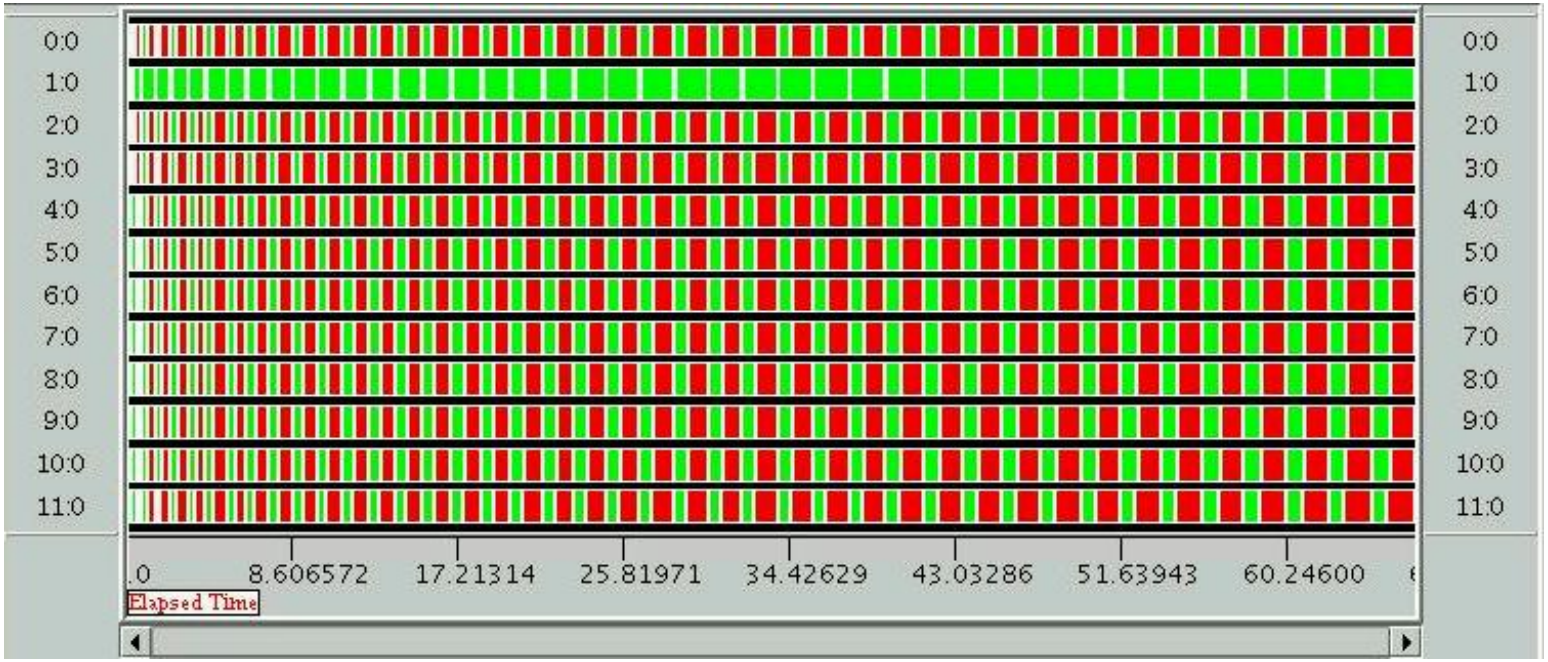


PNT: $\texttt{np} = 12$ (blocking)



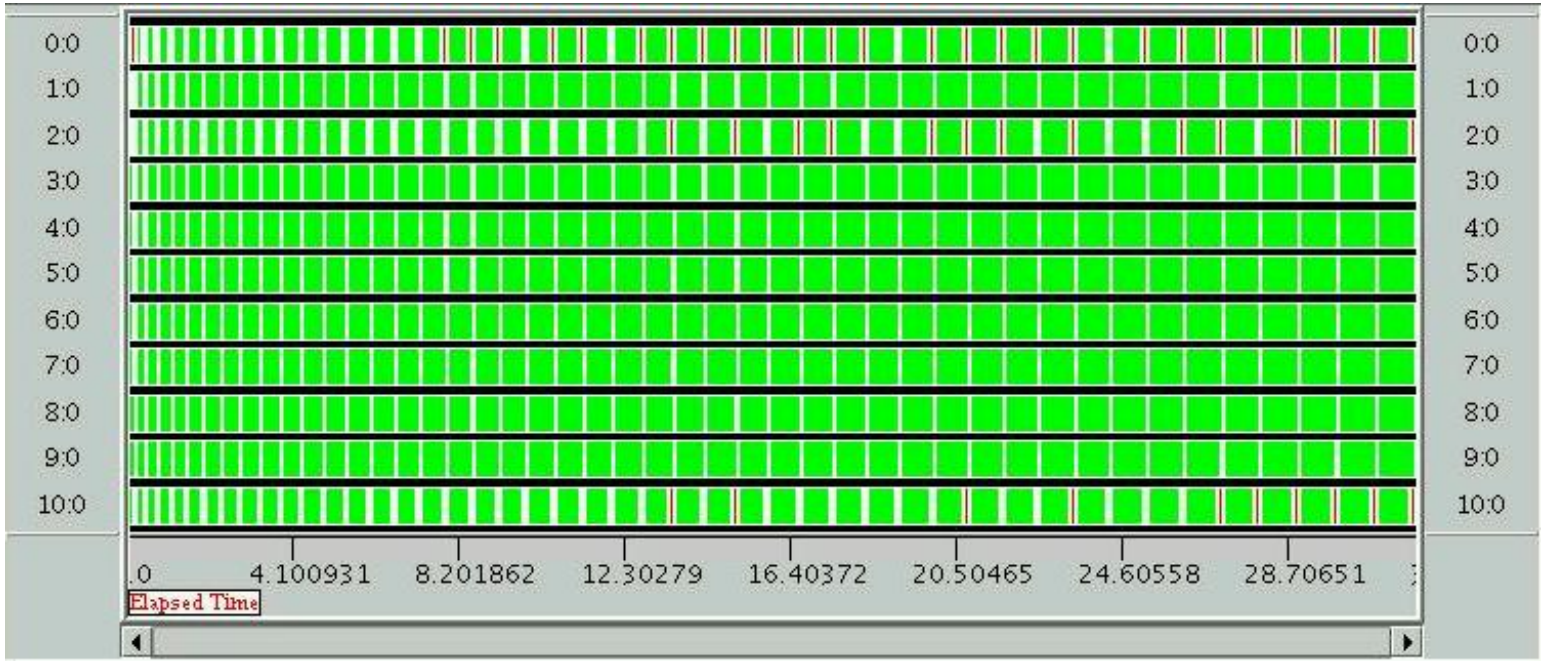
- **Nodos están corriendo otros programas (con carga variable).**
- **`comm` es en realidad *comunicación+sincronización*.**
- **`MPI_Allreduce()` actúa como una barrera de sincroniación (todos esperan hasta que *todos* lleguen a la barrera).**
- **Proc 1 es lento y desbalancea (los otros lo tienen que esperar) resultando en una pérdida de tiempo de 20 a 30%**

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



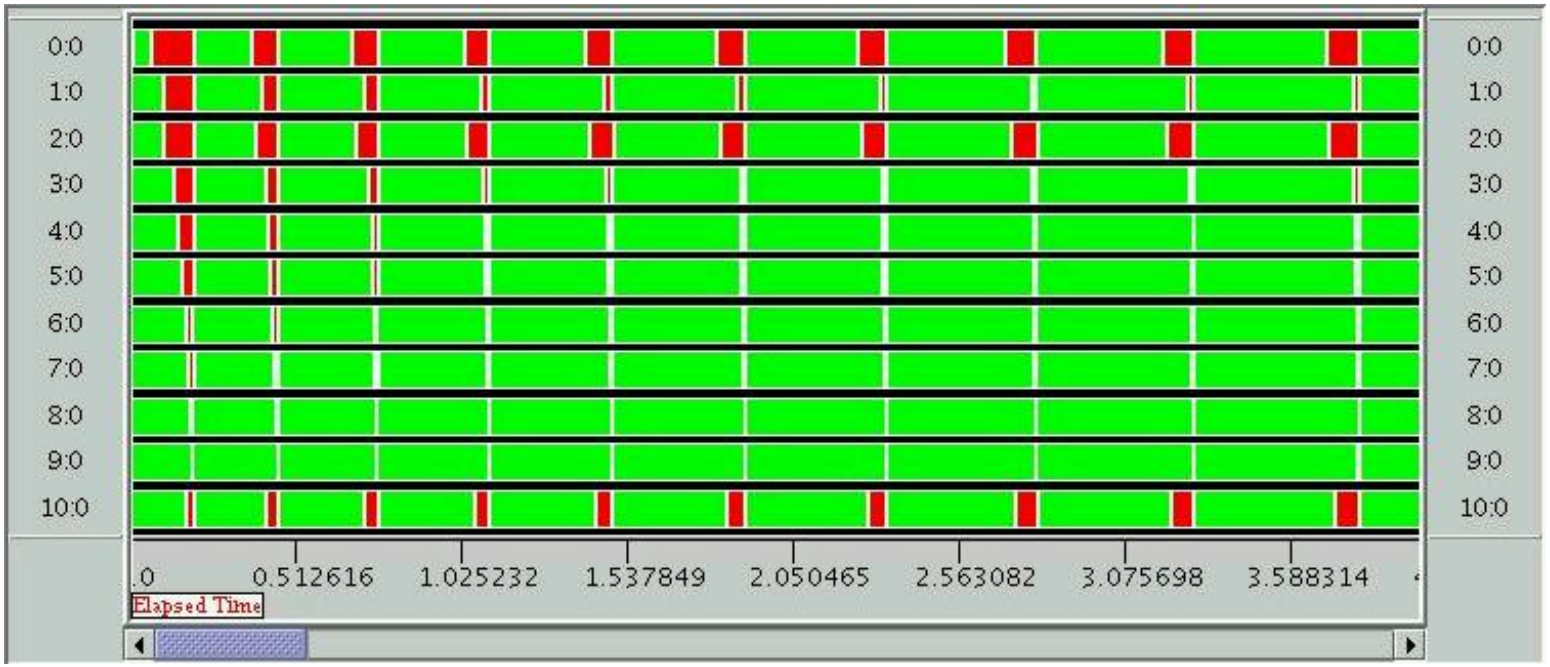
- **Proc 1 está corriendo otros programas. Otros procs sin carga.**
- **Desbalance aún mayor.**

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



- **Sin el procesador 12. (buen balance)**

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

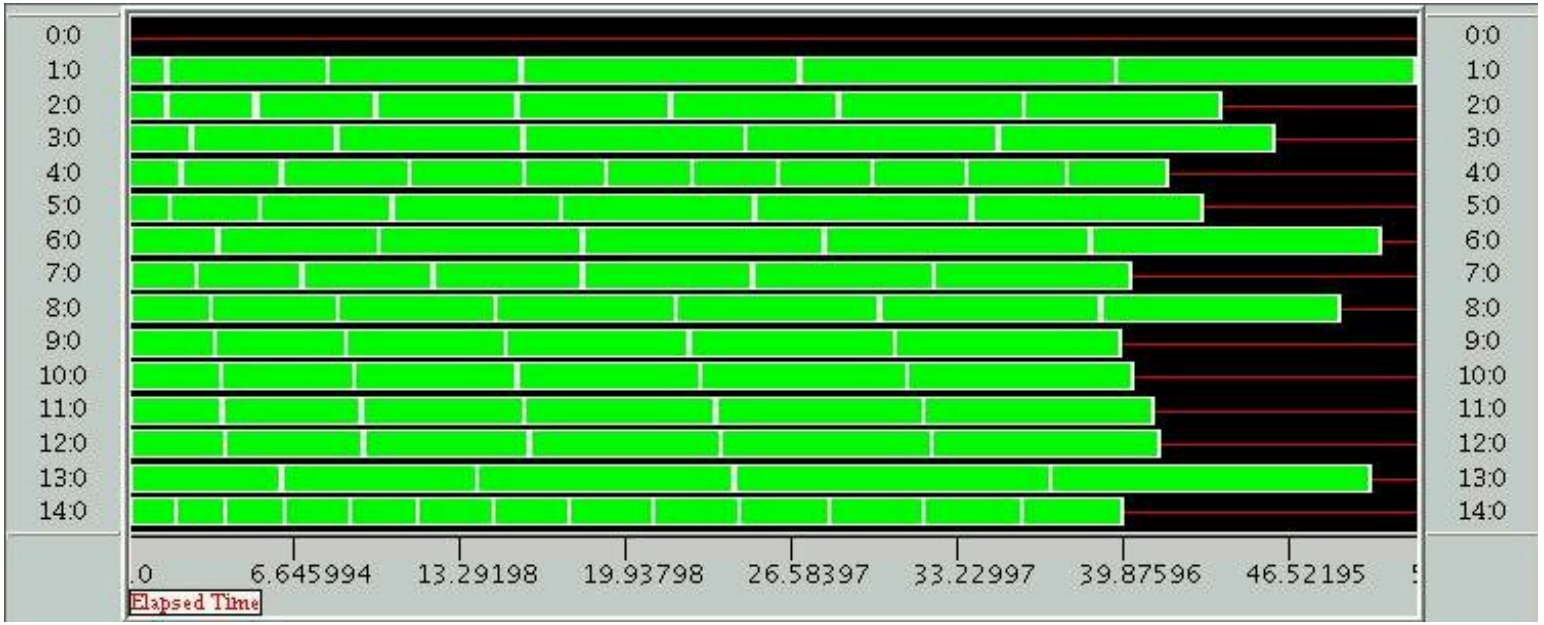


- **Sin el procesador 12. (buen balance, detalle)**

PNT: versión paralela dinámica.

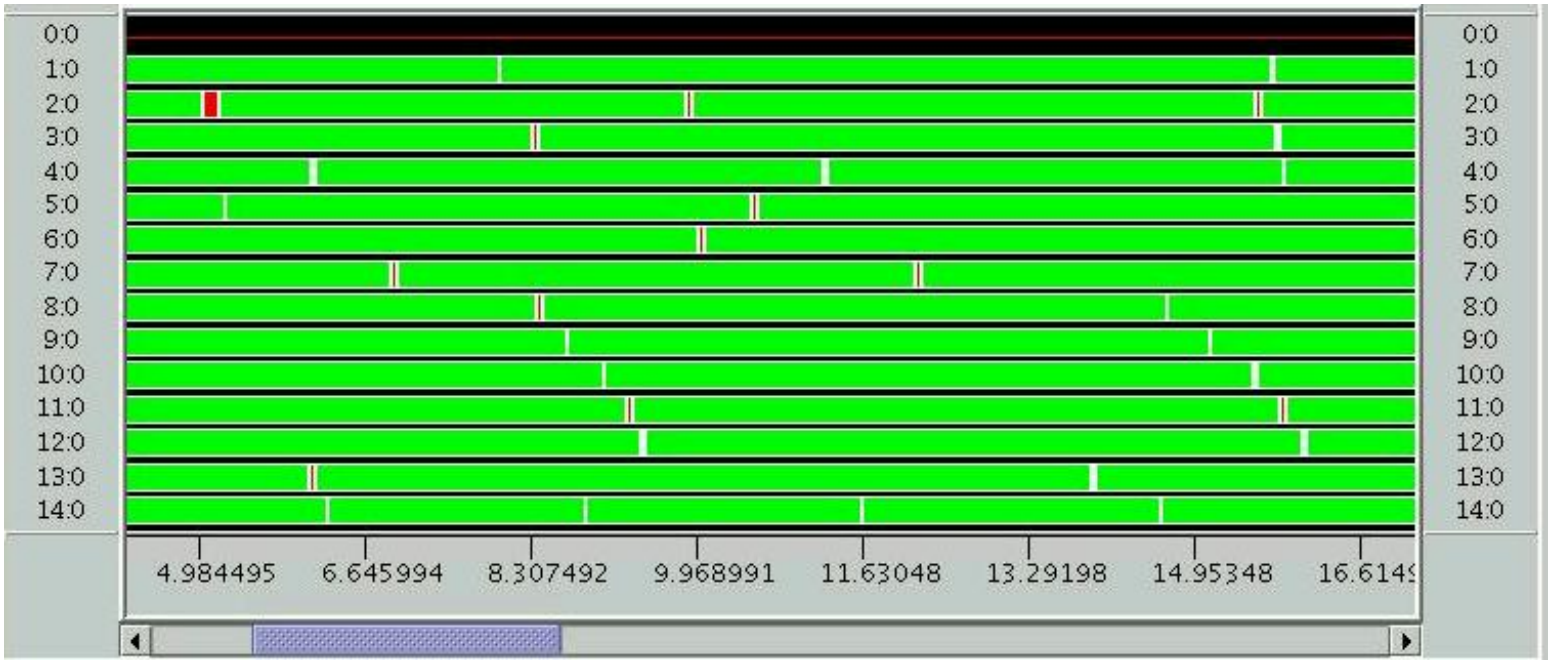
- **Proc 0 actúa como servidor.**
- **Esclavos mandan al master el *"trabajo realizado"* y éste les devuelve el nuevo trabajo a realizar.**
- **Trabajo realizado es `stat[2]`: número de enteros chequeados (`stat[0]`) y primos encontrados (`stat[1]`).**
- **Trabajo a realizar es un solo entero: `first`. El esclavo sabe que debe verificar por primalidad `chunk` enteros a partir de `start`.**
- **Inicialmente los esclavos mandan un mensaje `stat[]={0,0}` para darle a conocer al master que están listos para trabajar.**
- **Automáticamente, cuando `last>N` los nodos se dan cuenta de que no deben continuar más y se detienen.**
- **El master a su vez guarda un contador `down` de a cuántos esclavos ya le mandó el mensaje de parada `first>N`. Cuando este contador llega a `size-1` el servidor también para.**

PNT: versión paralela dinámica. (cont.)



- **Muy buen balance dinámico.**
- **Ineficiencia por alta granularidad al final (desaparece a medida que hacemos crecer el problema.)**
- **Notar ciertos procs (4 y 14) procesan chunks a ua relación 3:1 con los más lentos (1 y 13).**
- **Llevar estadística y mandar chunks pequeños a los más lentos. (tamaño del chunk $\propto$ velocidad del proc.).**

PNT: versión paralela dinámica. (cont.)



PNT: versión paralela dinámica. Seudocódigo.

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

PNT: versión paralela dinámica. Código.

```
1  //$Id: primes4.cpp,v 1.4 2004/07/25 13:34:07 mstorti Exp $
2  #include <mpi.h>
3  #include <mpe.h>
4  #include <cstdio>
5  #include <cmath>
6  #include <cassert>
7
8  int is_prime(int n) {
9    int m = int(sqrt(n));
10   for (int j=2; j<=m; j++)
11     if (!(n % j)) return 0;
12   return 1;
13 }
14
15 int main(int argc, char **argv) {
16   MPI_Init(&argc,&argv);
17   MPE_Init_log();
18
19   int  myrank, size;
20   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
21   MPI_Comm_size(MPI_COMM_WORLD,&size);
22
23   assert(size>1);
24   int start_comp = MPE_Log_get_event_number();
25   int end_comp = MPE_Log_get_event_number();
26   int start_comm = MPE_Log_get_event_number();
27   int end_comm = MPE_Log_get_event_number();
28
```

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

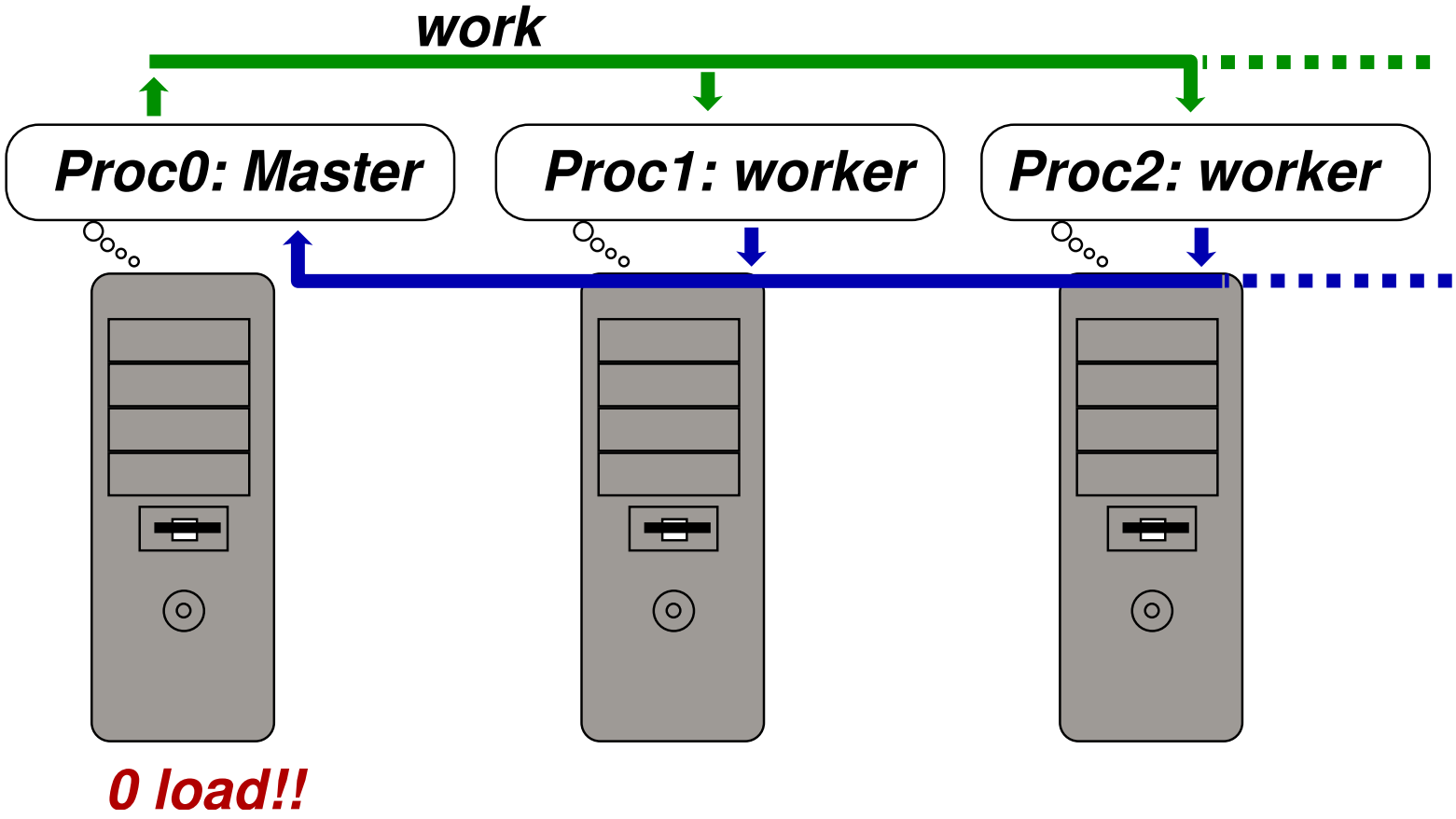
```
62          MPI_Recv(&start,1,MPI_INT,0,MPI_ANY_TAG,
63                   MPI_COMM_WORLD,&status);
64          MPE_Log_event(end_comm,0,"end-comm");
65          if (start>=N) break;
66          MPE_Log_event(start_comp,0,"start-comp");
67          int last = start + chunk;
68          if (last>N)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

PNT: versión paralela dinámica.

- **Esta estrategia se llama *"master/slave"* o *"compute-on-demand"*. Los esclavos están a la espera de recibir tarea, una vez que reciben tarea la hacen y la devuelven, esperando más tarea.**
- **Se puede implementar de varias formas**
  - ▷ **Un proceso en cada procesador. El proceso master responde inmediatamente a la demanda de trabajo. Se pierde un procesador.**
  - ▷ **En el procesador 0 hay dos procesos: `myrank=0` (master) y `myrank=1` (worker). Puede haber un cierto delay en la respuesta del master. No se pierde ningún procesador.**
  - ▷ **Modificar tal que el proceso master (`myrank=0`) procesa mientras espera que le pidan trabajo.**
- **Cual de las dos últimas es mejor depende de la granularidad del trabajo individual a realizar el master y del time-slice del sistema operativo.**

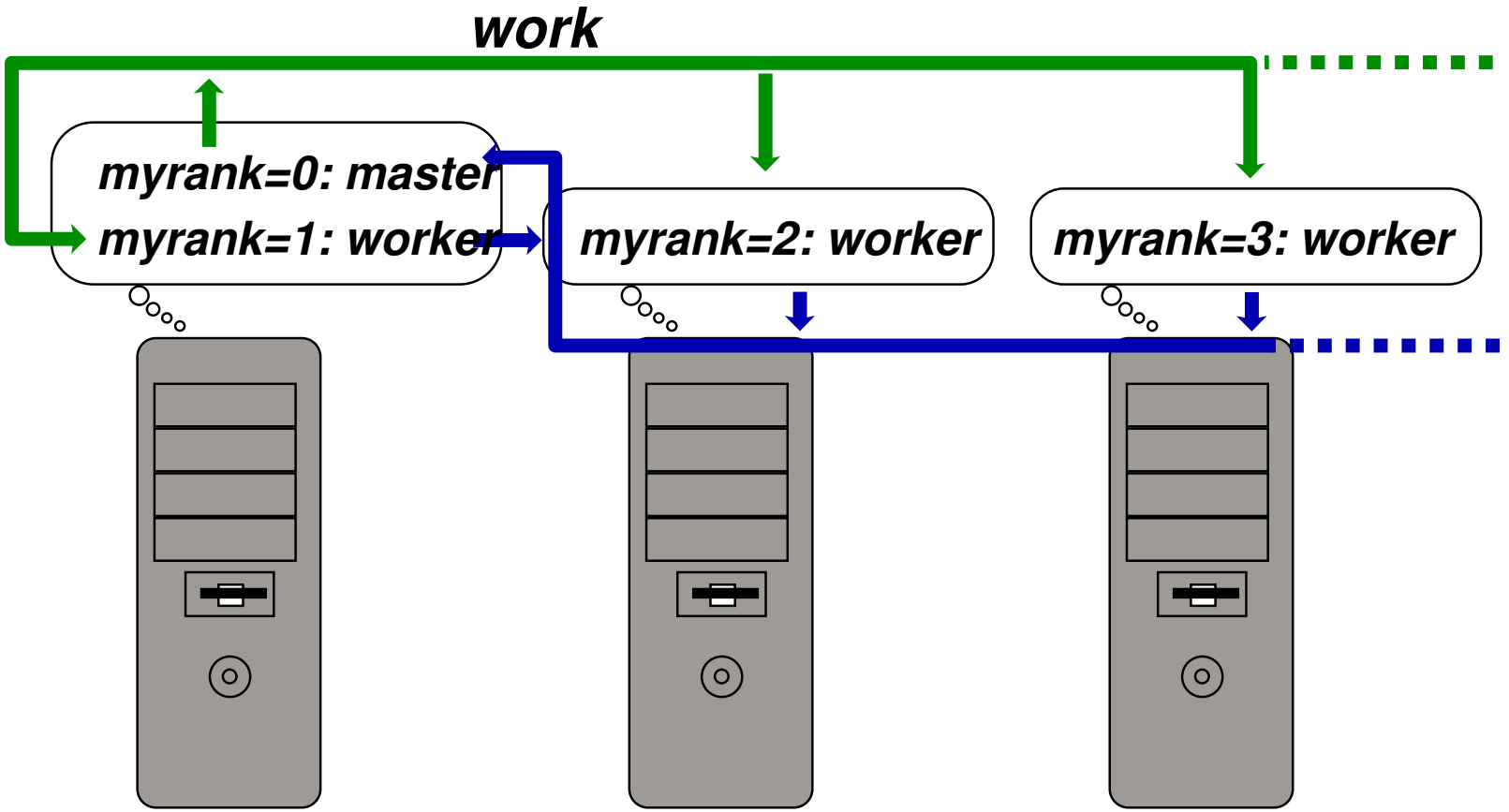
PNT: versión paralela dinámica. (cont.)

Un proceso en cada procesador. El proceso master responde inmediatamente a la demanda de trabajo. Se pierde un procesador.



PNT: versión paralela dinámica. (cont.)

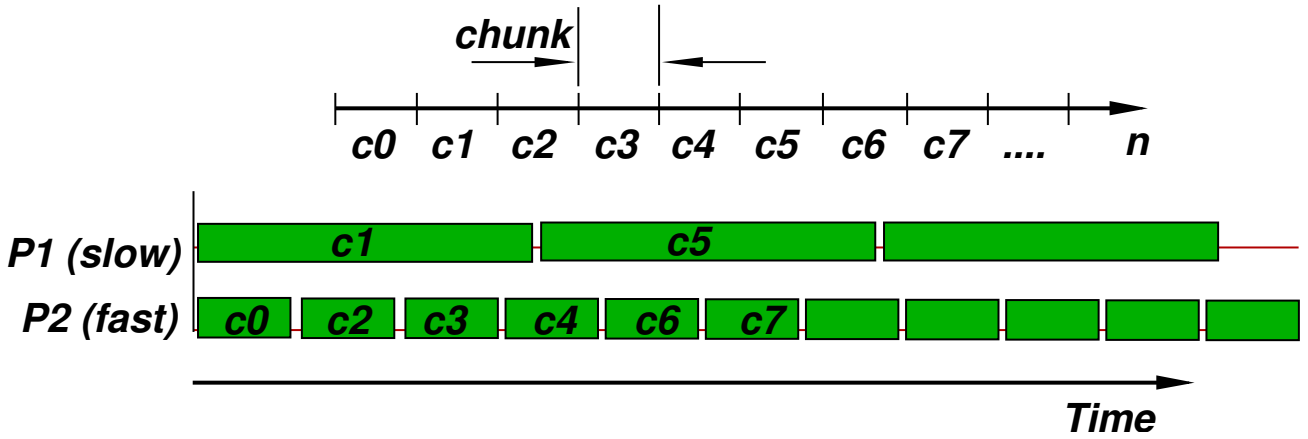
En el procesador 0 hay dos procesos: `myrank=0` (master) y `myrank=1` (worker). Puede haber un cierto delay en la respuesta del master. No se pierde ningún procesador.



PNT: versión paralela dinámica. (cont.)

- **Tiempos calculando $\pi(5\times10^7) = 3001136$.**
- **16 nodos ((rápidos P4HT, 3.0GHZ, DDR-RAM, 400MHz, dual channel), (lentos P4HT, 2.8GHZ, DDR-RAM, 400MHz)).**
- **Secuencial ($\mathrm{np} = 1$) en nodo 10 (rápido): 285 sec.**
- **Secuencial ($\mathrm{np} = 1$) en nodo 24 (rápido): 338 sec.**
- **En $\mathrm{np} = 12$ (desbalanceado, bloqueante) 69 sec.**
- **Excluyendo el nodo cargado (balanceado, bloqueante, $\mathrm{np} = 11$): 33sec.**
- **En ($\mathrm{np} = 14$) (balanceo dinámico, cargado con otros procesos, carga media=1) 59 sec.**

Imprimiendo los valores de PI(n) en tiempo real



- **Sobre todo si hay grandes desbalances en la velocidad de procesadores, puede ser que un procesador esté devolviendo un *chunk* que mientras que un chunk anterior todavía no fue calculado, de manera que no podemos directamente acumular `primes` para reportar $\pi(n)$.**
- **En la figura, los chunks $c2$, $c3$, $c4$ son reportados recién después de cargar $c1$, $c6$ y $c7$ después de cargar $c5$...**

Imprimiendo los valores de PI(n) en tiempo real (cont.)

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

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

Imprimiendo los valores de PI(n) en tiempo real (cont.)

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

Imprimiendo los valores de PI(n) en tiempo real (cont.)

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

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

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

```
95          }
96          while (!received.empty()) {
97            set<chunk_info>::iterator q = received.begin();
98            if (q->start != first_recv) break;
99            primes += q->primes;
100           received.erase(q);
101           first_recv += chunk;
102         }
103         printf("pi(%d) = %d,(queued %d)\n",
104                first_recv+chunk,primes,received.size());
105         MPI_Send(&first,1,MPI_INT,source,0,MPI_COMM_WORLD);
106         start_sent[source] = first;
107         if (first<N) first += chunk;
108         else {
109           down++;
110           printf("shutting down %d, so far %d\n",source,down);
111         }
112         if (down==size-1) break;
113       }
114       set<chunk_info>::iterator q = received.begin();
115       while (q!=received.end()) {
116         primes += q->primes;
117         printf("pi(%d) = %d\n",q->start+chunk,primes);
118         q++;
119       }
120     } else {
121       int start;
122       stat[0]=0; stat[1]=0;
123       MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
124       while(1) {
125         MPE_Log_event(start_comm,0,"start-comm");
126         MPI_Recv(&start,1,MPI_INT,0,MPI_ANY_TAG,
127                  MPI_COMM_WORLD,&status);
```

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

Leer opciones y datos

- **Leer opciones de un archivo via NFS**

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

(OK para un volumen de datos no muy grande)

- **Leer por consola (de stdin) y broadcast a los nodos**

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

(OK para opciones. Necesita Bcast).

Leer opciones y datos (cont.)

- **Entrar opciones via la línea de comando (Unix(POSIX)/getopt)**

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

- **Para grandes volúmenes de datos: leer en master y hacer Bcast.**

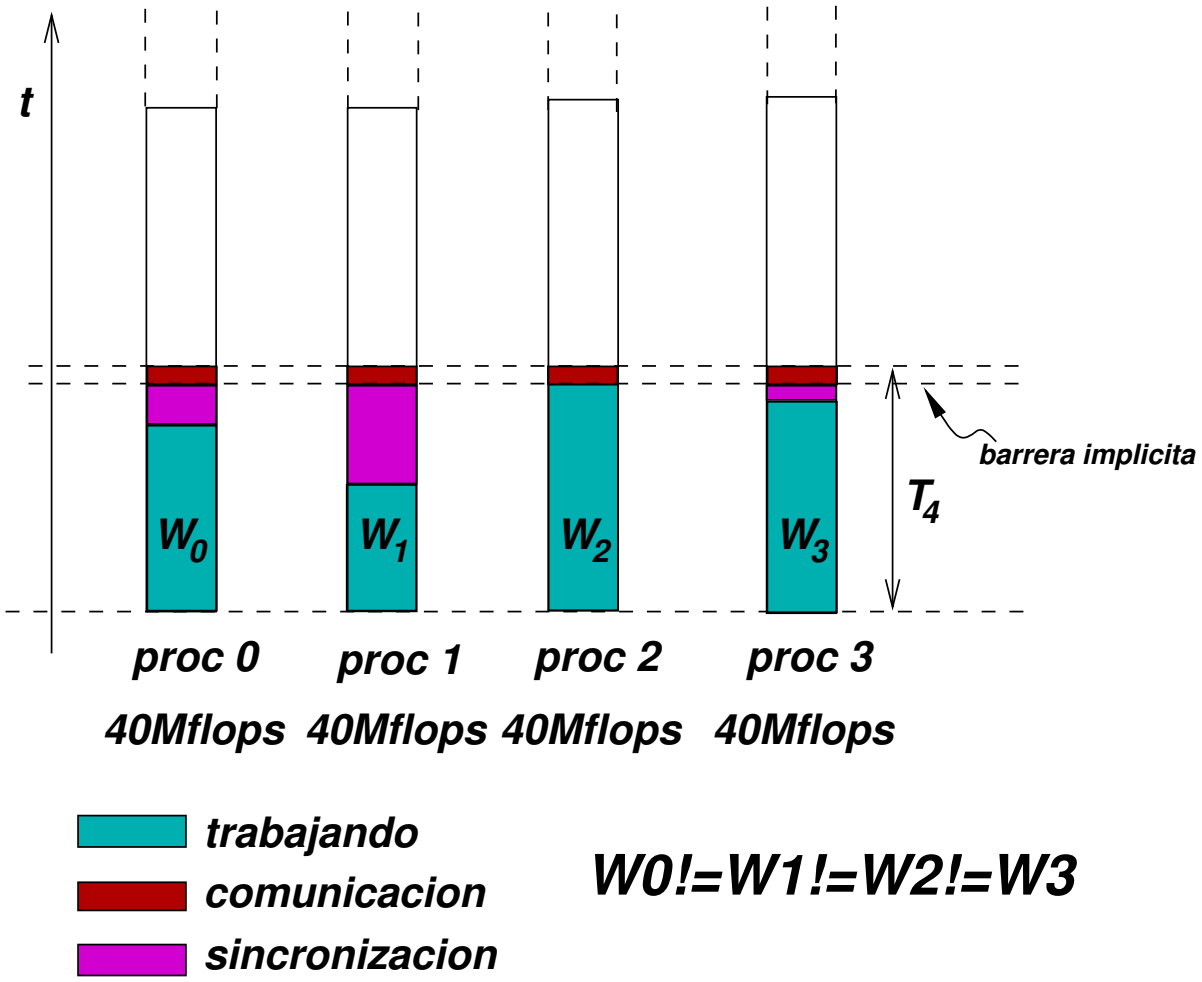
Escalabilidad

El problema de escalabilidad es peor aún si no es fácil de distribuir la carga, ya que en ese caso además del tiempo extra de comunicación tememos un tiempo extra de *"sincronización"*. En el caso del PNT determinar si un dado número j es primo o no varía mucho con j, para los pares es casi inmediato determinar que no son primos y en general el costo es proporcional al tamaño del primer divisor. Además, en general el costo crece con j, en promedio es $O(\sqrt{j})$. Si bien inicialmente podemos enviar una cierta cantidad m de enteros a cada procesador la carga real, es decir la cantidad de trabajo a realizar no es conocida a priori. Ahora bien, en el caso del algoritmo con división estática del trabajo tendremos que cada procesador terminará en un cierto tiempo variable $T_{\text{comp},i} = W_i/s$ la cantidad de trabajo que le fue esperada y se quedará esperando hasta que aquel que recibió más trabajo termine

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

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

Escalabilidad (cont.)



Escalabilidad (cont.)

$$\eta = \frac{S_n}{S_n^*} = \frac{T_1}{nT_n} = \frac{W/s}{n(\max_i T_{\text{comp},i} + T_{\text{comm}})}$$

$$= \frac{W/s}{\sum_i (T_{\text{comp},i} + T_{\text{sync},i} + T_{\text{comm}})}$$

$$= \frac{W/s}{\sum_i ((W_i/s) + T_{\text{sync},i} + T_{\text{comm}})}$$

$$= \frac{W/s}{(W/s) + \sum_i (T_{\text{sync},i} + T_{\text{comm}})}$$

$$= \frac{\textbf{(total comp. time)}}{\textbf{(total comp. time)} + \textbf{(total comm.+sync. time)}}$$

Rendimiento en clusters heterogéneos

- **Si un procesador es más rápido y se asigna la misma cantidad de tarea a todos los procesadores independientemente de su velocidad de procesamiento, cada vez que se envía una tarea a todos los procesadores, los más rapidos deben esperar a que el más lento termine, de manera que la velocidad de procesamiento es a lo sumo n veces la del procesador más lento. Esto produce un deterioro en la performance del sistema para clusters heterogéneos. El concepto de speedup mencionado anteriormente debe ser extendido a grupos heterogéneos de procesadores.**

Rendimiento en clusters heterogéneos (cont.)

- **Supongamos que el trabajo W (un cierto número de operaciones) es dividido en n partes iguales $W_i = W/n$**
- **Cada procesador tarda un tiempo $t_i = W_i/s_i = W/ns_i$ donde s_i es la velocidad del procesador i (por ejemplo en Mflops). El hecho de que los t_i sean distintos entre si indica una pérdida de eficiencia.**
- **El tiempo total transcurrido es el mayor de los t_i, que corresponde al menor s_i:**

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

Rendimiento en clusters heterogéneos (cont.)

- **El tiempo T_1 para hacer el speedup podemos tomarlo como el obtenido en el más rápido, es decir**

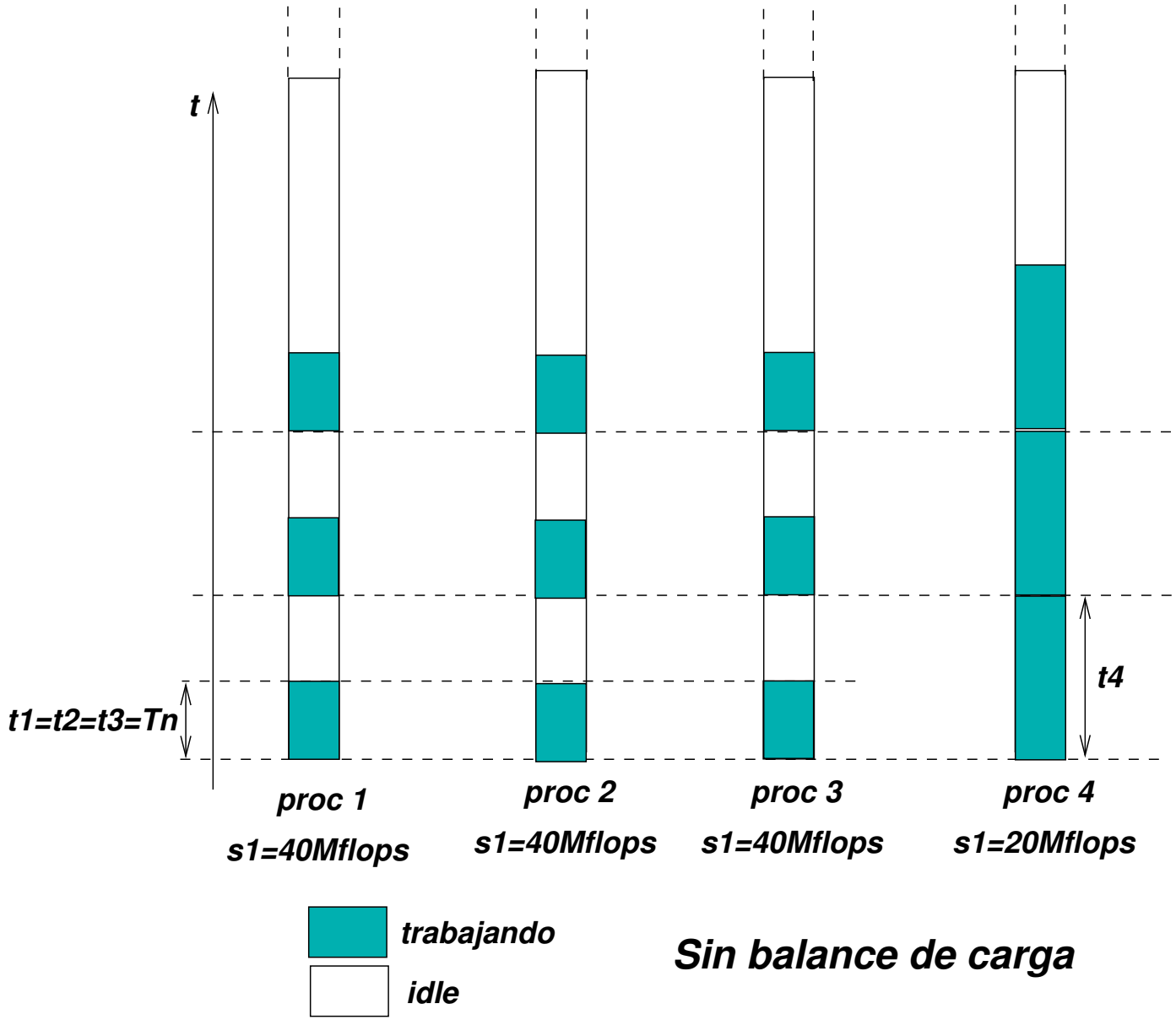
$$T_1 = \min t_i = \frac{W}{\max_i s_i}$$

- **El speedup resulta ser entonces**

$$S_n = \frac{T_1}{T_n} = \frac{W}{\max_i s_i} \Big/ \frac{W}{n \min_i s_i} = n \frac{\min_i s_i}{\max_i s_i} \qquad (1)$$

Por ejemplo, si tenemos un cluster con 12 procesadores con velocidades relativas 1x8 y 0.6x4, entonces el *"factor de desbalance"* $\min_i s_i / \max_i s_i$ **es de** 0.6**. El speedup teórico es de** $12 \times 0.6 = 7.2$**, con lo cual el rendimiento total *es menor que si se toman los 8 procesadores más rápidos solamente*. (Esto sin tener en cuenta el deterioro en el speedup por los tiempos de comunicación.)**

Rendimiento en clusters heterogéneos (cont.)


t
t1=t2=t3=Tn
t4
proc 1
s1=40Mflops
proc 2
s1=40Mflops
proc 3
s1=40Mflops
proc 4
s1=20Mflops
trabajando
idle
Sin balance de carga

Balance de carga

- **Si distribuimos la carga en forma proporcional a la velocidad de procesamiento**

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

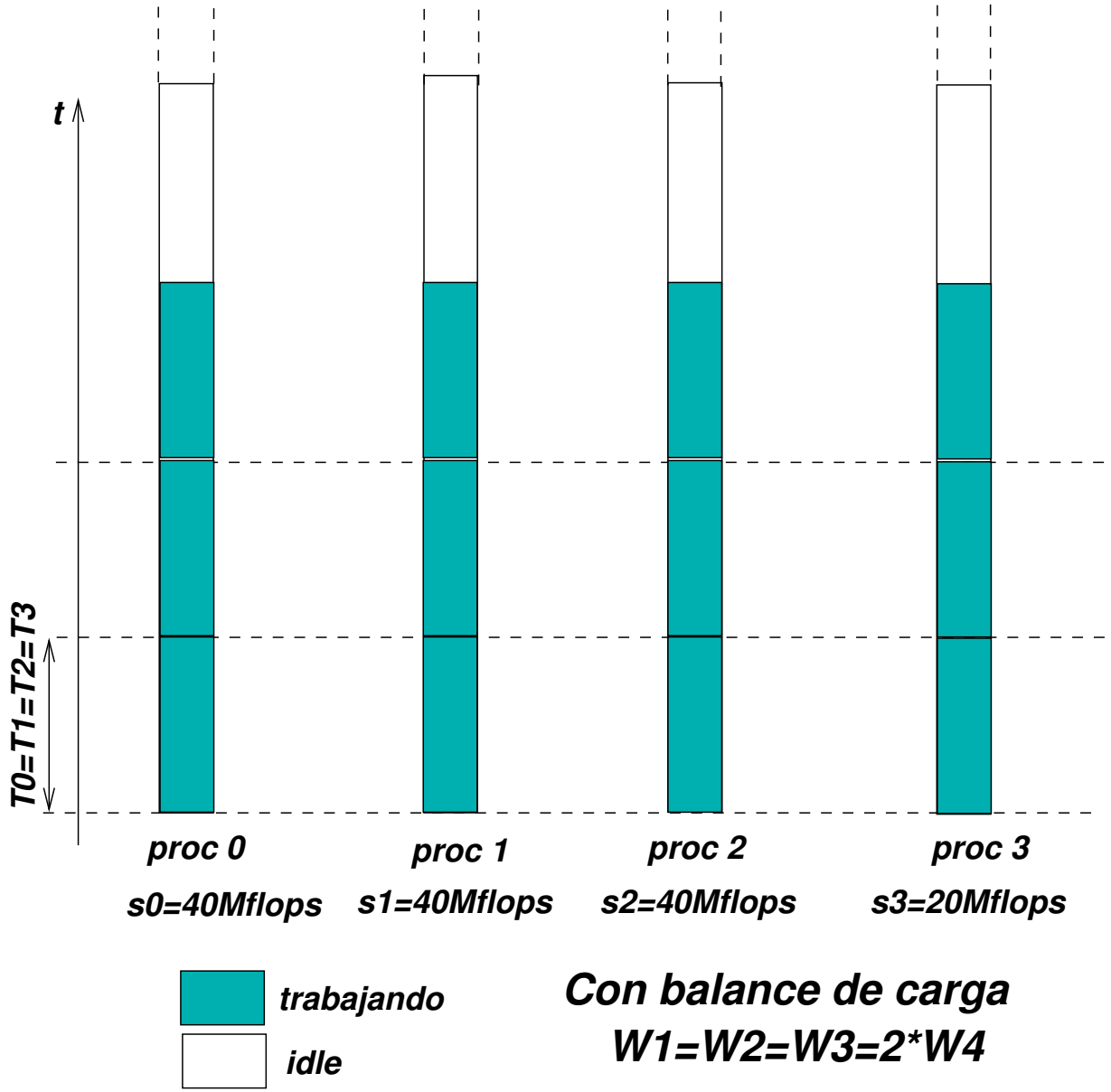
- **El tiempo necesario en cada procesador es**

$$t_i = \frac{W_i}{s_i} = \frac{W}{\sum_j s_j}$$ **(independiente de i!!!!!!)**

- **El speedup ahora es**

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

Este es el máximo speedup teórico alcanzable en clusters heterogéneos.


t
T0=T1=T2=T3
proc 0
s0=40Mflops
proc 1
s1=40Mflops
proc 2
s2=40Mflops
proc 3
s3=20Mflops
trabajando
idle
Con balance de carga
W1=W2=W3=2*W4

Paralelismo trivial

- **El ejemplo del PNT (`primes.cpp`) introdujo una serie de conceptos, llamadas punto a punto y colectivas.**
- ***"Compute-on-demand"* puede ser también implementado con programas secuenciales. Supongamos que queremos calcular una tabla de valores $f(x_i), i = 0, ..., m-1$. Se supone que tenemos un programa `computef -x <x-val>` que imprime por stdout el valor de $f(x)$.**

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

Paralelismo trivial (cont.)

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

```
29      sprintf(line,"proc%d.output",myrank);
30      FILE *fid = fopen(line,"r");
31      fscanf(fid,"%lf",&val);
32      system(line)
33      MPI_Send(val,...,0,...);
34    }
35  }
```

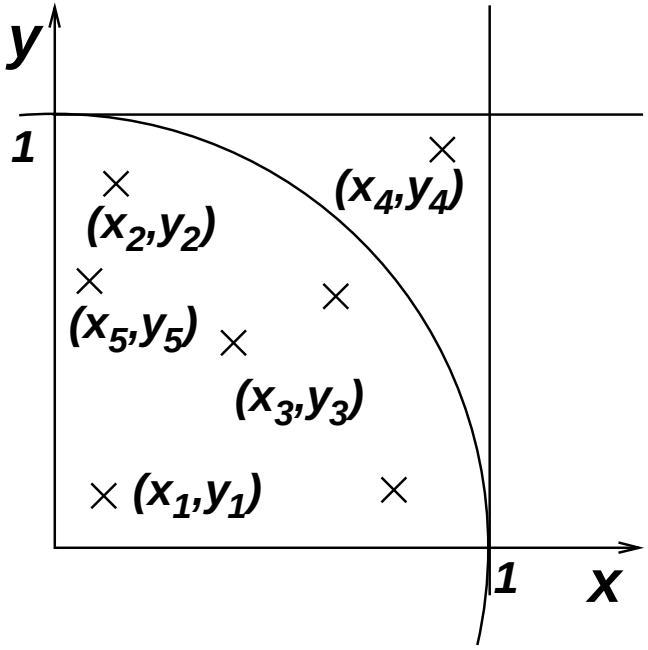
Guía de Trab. Prácticos Nro 1

Comunicación punto a punto y balance dinámico de carga.

1. **El TSP (*"Traveling Salesman Problem"*, o *"Problema del Agente Viajero"*) consiste en determinar el camino de longitud mínima que recorre todas las vértices de un grafo de n vértices, pasando una sola vez por cada vértice. Una tabla `d[n][n]` contiene las distancias entre los vértices, es decir `d[i][j]>0` es la distancia entre el vértice `i` y el `j`. Asumir que la tabla de distancias es simétrica (`d[i][j]=d[j][i]`) y `d[i][i]=0`.**
   (a) **Escribir un programa secuencial que encuentra el camino de longitud mínima por búsqueda exhaustiva intentando todos los posibles caminos, es decir todas las posibles permutaciones de n elementos. Notar que las permutaciones de n elementos $\mathrm{comb}(a0...a_{n-1})$ se pueden generar como**
   $(a_0, \mathrm{comb}(a_1, ..., a_n))$,$(a_1, \mathrm{comb}(a_0, a_2, a_3, ...)$,
   $(a_2, \mathrm{comb}(a_0, a_1, a_3, ...))$,$(a_{n_1}, \mathrm{comb}(a_0, a_1, ..., a_{n-2}))$
   (b) **Escribir una versión paralela con distribución fija del trabajo entre los procesadores (balanceo estático). Simular desbalance enviando varios procesos a un mismo procesador.**

(c) Escribir una versión paralela con distibución dinámica de la carga (*"compute-ond-demand"*).

Cálculo de Pi por Montecarlo



Generar puntos random (x_j, y_j). La probabilidad de que estén adentro del círculo es $\pi/4$.

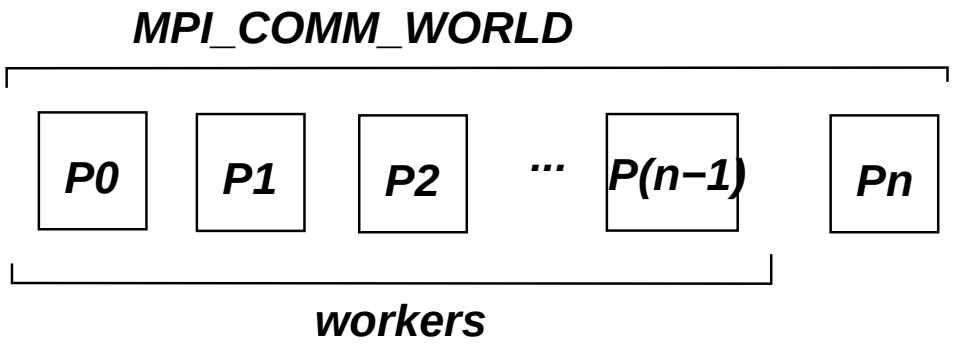
$$\pi = 4\frac{\text{(\#adentro)}}{\text{(\#total)}}$$

Los métodos de Montecarlo exhiben una tasa de convergencia lenta ($O(\sqrt{N})$).

Generación de números random en paralelo

- **Si todos los nodos generan números random sin *"randomizar"* la semilla, (Usualmente con `srand(time(NULL))`), entonces la secuencia generada será la misma en todos los procesadores y en realidad no se gana nada con procesar en paralelo.**
- **Si se randomiza el generador de random en cada nodo, entonces habría que garantizar que la ejecución se produce a tiempos diferentes. (Normalmente es así por pequeños retardos que se producen en la ejecució). Sin embargo esto no garantiza que no halla correlación entre los diferentes generadores, bajando la convergencia.**
- **Surge la necesidad de tener un *"servidor de random"*, es decir uno de los procesadores sólo se dedica a producir secuencias de números random.**

Servidor de random



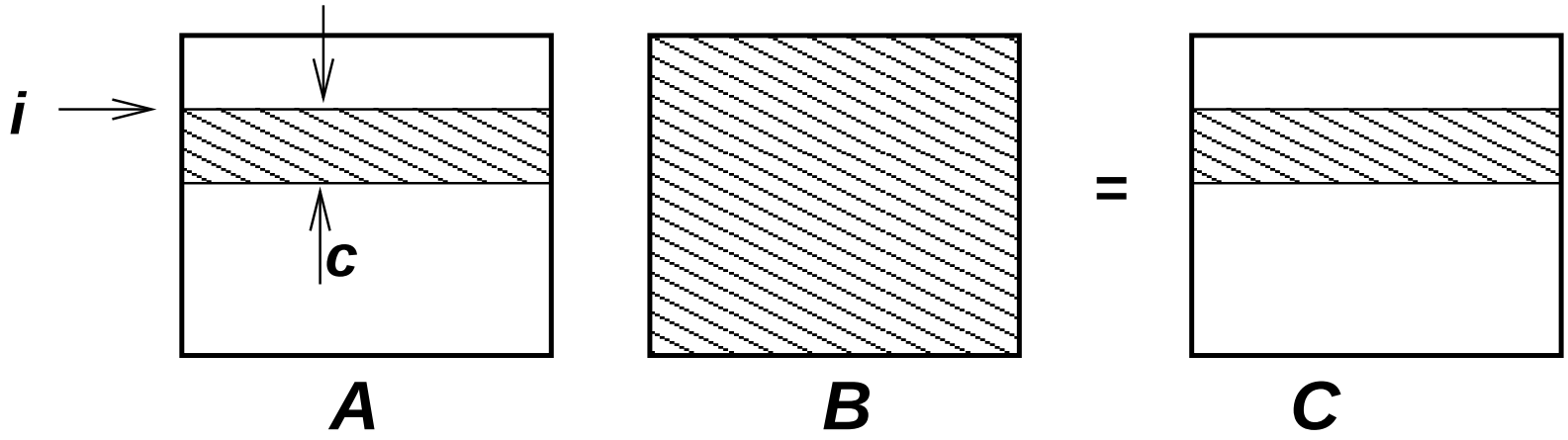
```
1  #include <math.h>
2  #include <limits.h>
3  #include <mpi.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  /* no. of random nos. to generate at one time */
7  #define CHUNKSIZE  1000
8
9  /* message tags */
10 #define REQUEST 1
11 #define REPLY   2
12
13 main(int argc, char **argv) {
14   int iter;
15   int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
16   double x, y, Pi, error, epsilon;
17   int numprocs, myid, server, totalin, totalout, workerid;
18   int rands[CHUNKSIZE], request;
19
20   MPI_Comm world, workers;
21   MPI_Group world_group, worker_group;
22   MPI_Status stat;
23
24   MPI_Init(&argc,&argv);
25   world = MPI_COMM_WORLD;
26   MPI_Comm_size(world, &numprocs);
27   MPI_Comm_rank(world, &myid);
28   server = numprocs-1;
29
30   if (numprocs==1)
31     printf("Error. At least 2 nodes are needed");
32
33   /* process 0 reads epsilon from args and broadcasts it to
```

```
34      everyone */
35    if (myid == 0) {
36      if (argc<2) {
37        epsilon = 1e-2;
38      } else {
39        sscanf ( argv[1], "%lf", &epsilon );
40      }
41    }
42    MPI_Bcast ( &epsilon, 1, MPI_DOUBLE, 0,MPI_COMM_WORLD );
43
44    /* define the workers communicator by using groups and
45       excluding the server from the group of the whole world */
46    MPI_Comm_group ( world, &world_group );
47    ranks[0] = server;
48    MPI_Group_excl ( world_group, 1, ranks, &worker_group );
49    MPI_Comm_create ( world, worker_group, &workers);
50    MPI_Group_free ( &worker_group);
51
52    /* the random number server code - receives a non-zero
53       request, generates random numbers into the array rands,
54       and passes them back to the process who sent the
55       request.  */
56    if ( myid == server ) {
57      do {
58        MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
59                 REQUEST, world, &stat);
60        if ( request ) {
61          for (i=0; i<CHUNKSIZE; i++) rands[i] = random();
62          MPI_Send(rands, CHUNKSIZE, MPI_INT,
63                   stat.MPI_SOURCE, REPLY, world);
64        }
65      } while ( request > 0 );
66      /* the code for the worker processes - each one sends a
```

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

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

Multiplicación de matrices



Todos los nodos tienen todo B, van recibiendo parte de A (un chunk de filas $A(i:i+n-1,:)$, hacen el producto $A(i:i+n-1,:)*B$ y lo devuelven.

- **Balance estático de carga: necesita conocer la velocidad de procesamiento.**
- **Balance dinámico de carga: cada nodo *"pide"* una cierta cantidad de trabajo al server y una vez terminada la devuelve.**

Librería simple de matrices

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

Balance dinámico

Abstracción: hay un serie de tareas con datos $x_1, \ldots, x_N$ y resultados $r_1, \ldots, r_N$ y P procesadores. Suponemos N suficientemente grande, y $1 <= $ chunksize $\ll N$ de tal manera que el overhead asociado con enviar y recibir las tareas sea despreciable.

Slave code

- **Manda un mensaje len=0 que significa “estoy listo”.**
- **Recibe una cantidad de tareas “len”. Si len>0, procesa y envia los resultados. Si len=0, manda un len=0 de reconocimiento, sale del lazo y termina.**

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

Server code

Mantiene un vector de estados para cada slave, puede ser: 0 = No empezó todavía, 1= Está trabajando, 2 = terminó.

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

Slave code, use local processor

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

```
1  #include <time.h>
2  #include <unistd.h>
3  #include <ctype.h>
4  #include "mat.h"
5
6  /* Self-scheduling algorithm.
7     Computes the product C=A*B; A,B and C matrices of NxN
8     Proc 0 is the root and sends work to the slaves
9   */
10 // $Id: matmult2.cpp,v 1.1 2003/04/24 01:47:16 mstorti Exp $
11
12 /** Computes the elapsed time between two instants captured with
13     'gettimeofday'.
14 */
15 double etime(timeval &x,timeval &y) {
16   return double(y.tv_sec)-double(x.tv_sec)
17     +(double(y.tv_usec)-double(x.tv_usec))/1e6;
18 }
19
20 /** Computes random numbers between 0 and 1
21 */
22 double drand() {
23   return ((double)(rand()))/((double)(RAND_MAX));
24 }
25
26 /** Computes an intger random number in the
27     range 'imin'-'imax'
28 */
29 int irand(int imin,int imax) {
30   return int(rint(drand()*double(imax-imin+1)-0.5))+imin;
31 }
32
33
34 int main(int argc,char **argv) {
```

```
35
36    /// Initializes MPI
37    MPI_Init(&argc,&argv);
38    /// Initializes random
39    srand(time (0));
40
41    /// Size of problem, size of chunks (in rows)
42    int N,chunksize;
43    /// root procesoor, number of processor, my rank
44    int root=0, numprocs, rank;
45    /// time related quantities
46    struct timeval start, end;
47
48    /// Status for retrieving MPI info
49    MPI_Status stat;
50    /// For non-blocking communications
51    MPI_Request request;
52    /// number of processors and my rank
53    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
54    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
55
56    /// cursor to the next row to send
57    int row_start=0;
58    /// Read input data from options
59    int c,print_mat=0,random_mat=0,use_local_processor=0,
60      slow_down=1;
61    while ((c = getopt (argc, argv, "N:c:prhls:")) != -1) {
62      switch (c) {
63      case 'h':
64        if (rank==0) {
65          printf(" usage: $ mpirun [OPTIONS] matmult2\n\n"
66                 "MPI options: -np <number-of-processors>\n"
67                 "              -machinefile <machine-file>\n\n"
```

```
68                  "Other options: -N <size-of-matrix>\n"
69                  "               -c <chunk-size>  "
70                  "# sets number of rows sent to processors\n"
71                  "               -p                       "
72                  "# print input and result matrices to output\n"
73                  "               -r                       "
74                  "# randomize input matrices\n");
75        }
76        exit(0);
77      case 'N':
78        sscanf(optarg,"%d",&N);
79        break;
80      case 'c':
81        sscanf(optarg,"%d",&chunksize);
82        break;
83      case 'p':
84        print_mat=1;
85        break;
86      case 'r':
87        random_mat=1;
88        break;
89      case 'l':
90        use_local_processor=1;
91        break;
92      case 's':
93        sscanf(optarg,"%d",&slow_down);
94        if (slow_down<1) {
95          if (rank==0) printf("slow_down factor (-s option)"
96                              " must be >=1. Reset to 1!\n");
97          abort();
98        }
99        break;
100     case '?':
101       if (isprint (optopt))
```

```
102         fprintf (stderr, "Unknown option ‘-%c’.\n", optopt);
103       else
104         fprintf (stderr,
105                  "Unknown option character ‘\\x%x’.\n",
106                  optopt);
107       return 1;
108     default:
109       abort ();
110     }
111   }
112
113 #if 0
114   if ( rank == root ) {
115     printf("enter N, chunksize > ");
116     scanf("%d",&N);
117     scanf("%d",&chunksize);
118     printf("\n");
119   }
120 #endif
121
122   /// Register time for statistics
123   gettimeofday (&start,NULL);
124   /// broadcast N and chunksize to other procs.
125   MPI_Bcast (&N, 1, MPI_INT, 0,MPI_COMM_WORLD );
126   MPI_Bcast (&chunksize, 1, MPI_INT, 0,MPI_COMM_WORLD );
127
128   /// Define matrices, bufa and bufc is used to
129   /// send matrices to other processors
130   Mat b(N,N),bufa(chunksize,N), bufc(chunksize,N);
131
132   //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
133   //---:---<*>---:-     SERVER PART       *>---:---<*>
134   //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
135   if ( rank == root ) {
```

```
136
137     Mat a(N,N),c(N,N);
138
139     /// Initialize 'a' and 'b'
140     for (int j=0; j<N; j++) {
141       for (int k=0; k<N; k++) {
142         // random initialization
143         if (random_mat) {
144           a(j,k) = floor(double(rand())/double(INT_MAX)*4);
145           b(j,k) = floor(double(rand())/double(INT_MAX)*4);
146         } else {
147           // integer index initialization (eg 00 01 02 .... NN)
148           a(j,k) = &a(j,k)-&a(0,0);
149           b(j,k) = a(j,k)+N*N;
150         }
151       }
152     }
153
154     /// proc_status[proc] = 0 -> processor not contacted
155     ///                   = 1 -> processor is working
156     ///                   = 2 -> processor has finished
157     int *proc_status = (int *) malloc(sizeof(int)*numprocs);
158     /// statistics[proc] number of rows that have been processed
159     /// by this processor
160     int *statistics  = (int *) malloc(sizeof(int)*numprocs);
161
162     /// initializate proc_status, statistics
163     for (int proc=1; proc<numprocs; proc++) {
164       proc_status[proc] = 0;
165       statistics [proc] = 0;
166     }
167
168     /// send b to all processor
```

```
169     MPI_Bcast (&b(0,0), N*N, MPI_DOUBLE, 0,MPI_COMM_WORLD );
170
171     /// Register time for statistics
172     gettimeofday (&end,NULL);
173     double bcast = etime(start,end);
174
175     while (1) {
176
177       /// non blocking receive
178       if (numprocs>1) MPI_Irecv(&bufc(0,0),chunksize*N,
179                                 MPI_DOUBLE,MPI_ANY_SOURCE,
180                                 MPI_ANY_TAG,
181                                 MPI_COMM_WORLD,&request);
182
183       /// While waiting for results from slaves server works ...
184       int receive_OK=0;
185       while (use_local_processor && row_start<N) {
186         /// Test if some message is waiting.
187         if (numprocs>1) MPI_Test(&request,&receive_OK,&stat);
188         if(receive_OK) break;
189         /// Otherwise... work
190         /// Local masks
191         Mat aa(1,N,&a(row_start,0)),cc(1,N,&c(row_start,0));
192         /// make product
193         for (int jj=0; jj<slow_down; jj++)
194           cc.matmul(aa,b);
195         /// increase cursor
196         row_start++;
197         /// register work
198         statistics[0]++;
199       }
200
201       /// If no more rows to process wait
```

```
202         // until message is received
203         if (numprocs>1) {
204           if (!receive_OK) MPI_Wait(&request,&stat);
205
206           /// length of received message
207           int rcvlen;
208           /// processor that sent the result
209           int proc = stat.MPI_SOURCE;
210           MPI_Get_count(&stat,MPI_DOUBLE,&rcvlen);
211
212           if (rcvlen!=0) {
213             /// Store result in 'c'
214             int rcv_row_start = stat.MPI_TAG;
215             int nrows_sent = rcvlen/N;
216             for (int j=rcv_row_start;
217                  j<rcv_row_start+nrows_sent; j++) {
218               for (int k=0; k<N; k++) {
219                 c(j,k) = bufc(j-rcv_row_start,k);
220               }
221             }
222           } else {
223             /// Zero length messages are used
224             // to acknowledge start work
225             // and finished work. Increase
226             // status of processor.
227             proc_status[proc]++;
228           }
229
230           /// Rows to be sent
231           int row_end = row_start+chunksize;
232           if (row_end>N) row_end = N;
233           int nrows_sent = row_end - row_start;
234
```

```
235           /// Increase statistics, send task to slave
236           statistics[proc] += nrows_sent;
237           MPI_Send(&a(row_start,0),nrows_sent*N,MPI_DOUBLE,
238                    proc,row_start,MPI_COMM_WORLD);
239           row_start = row_end;
240         }
241
242         /// If all processors are in state 2,
243         // then all work is done
244         int done = 1;
245         for (int procc=1; procc<numprocs; procc++) {
246           if (proc_status[procc]!=2) {
247             done = 0;
248             break;
249           }
250         }
251         if (done) break;
252       }
253
254       /// time statistics
255       gettimeofday (&end,NULL);
256       double dtime = etime(start,end);
257       /// print statistics
258       double dN = double(N), rate;
259       rate = 2*dN*dN*dN/dtime/1e6;
260       printf("broadcast: %f secs [%5.1f %%], process: %f secs\n"
261              "rate: %f Mflops on %d procesors\n",
262              bcast,bcast/dtime*100.,dtime-bcast,
263              rate,numprocs);
264       if (slow_down>1) printf("slow_down=%d, scaled Mflops %f\n",
265                               slow_down,rate*slow_down);
266       for (int procc=0; procc<numprocs; procc++) {
267         printf("%d lines processed by %d\n",
```

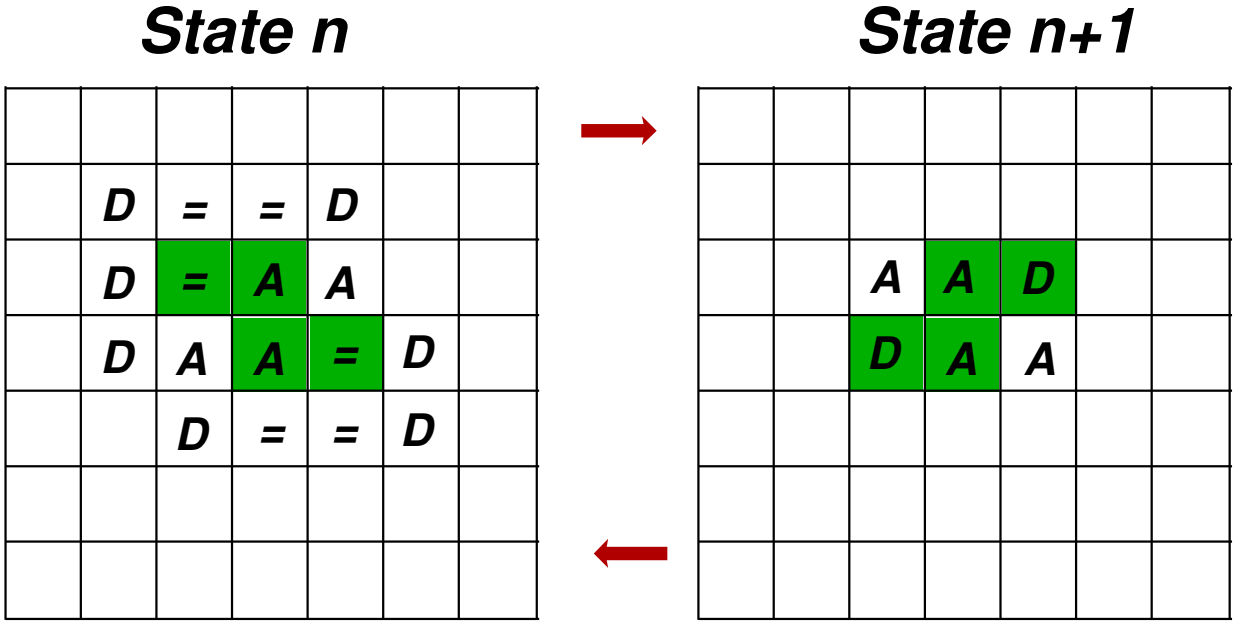
```
268             statistics[procc],procc);
269       }
270
271       if (print_mat) {
272         printf("a: \n");
273         a.print();
274         printf("b: \n");
275         b.print();
276         printf("c: \n");
277         c.print();
278       }
279
280       /// free memory
281       free(statistics);
282       free(proc_status);
283
284     } else {
285
286       //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
287       //---:---<*>---:-     SLAVE  PART       *>---:---<*>
288       //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>
289
290       /// get 'b'
291       MPI_Bcast(&b(0,0), N*N, MPI_DOUBLE, 0,MPI_COMM_WORLD );
292       /// Send message zero length 'I'm ready'
293       MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);
294
295       while (1) {
296         /// Receive task
297         MPI_Recv(&bufa(0,0),chunksize*N,MPI_DOUBLE,0,MPI_ANY_TAG,
298                  MPI_COMM_WORLD,&stat);
299         int rcvlen;
300         MPI_Get_count(&stat,MPI_DOUBLE,&rcvlen);
```

```
301         /// zero length message means: no more rows to process
302         if (rcvlen==0) break;
303
304         /// compute number of rows received
305         int nrows_sent = rcvlen/N;
306         /// index of first row sent
307         int row_start = stat.MPI_TAG;
308         /// local masks
309         Mat bufaa(nrows_sent,N,&bufa(0,0)),
310           bufcc(nrows_sent,N,&bufc(0,0));
311         /// compute product
312         for (int jj=0; jj<slow_down; jj++)
313           bufcc.matmul(bufaa,b);
314         /// send result back
315         MPI_Send(&bufcc(0,0),nrows_sent*N,MPI_DOUBLE,
316                  0,row_start,MPI_COMM_WORLD);
317
318       }
319
320       /// Work finished. Send acknowledge (zero length message)
321       MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);
322
323     }
324
325     // Finallize MPI
326     MPI_Finalize();
327   }
```

El juego de la vida

Game of Life is an example of cellular automata with simple evolution rules that leads to complex behavior. A board of $N \times N$ cells c_{ij} for $0 \leq i, j < N$, that can be either in *"dead"* of *"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 accoeding to the following rules

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



El juego de la vida (cont.)

Formally the rules are as follows. Let

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

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

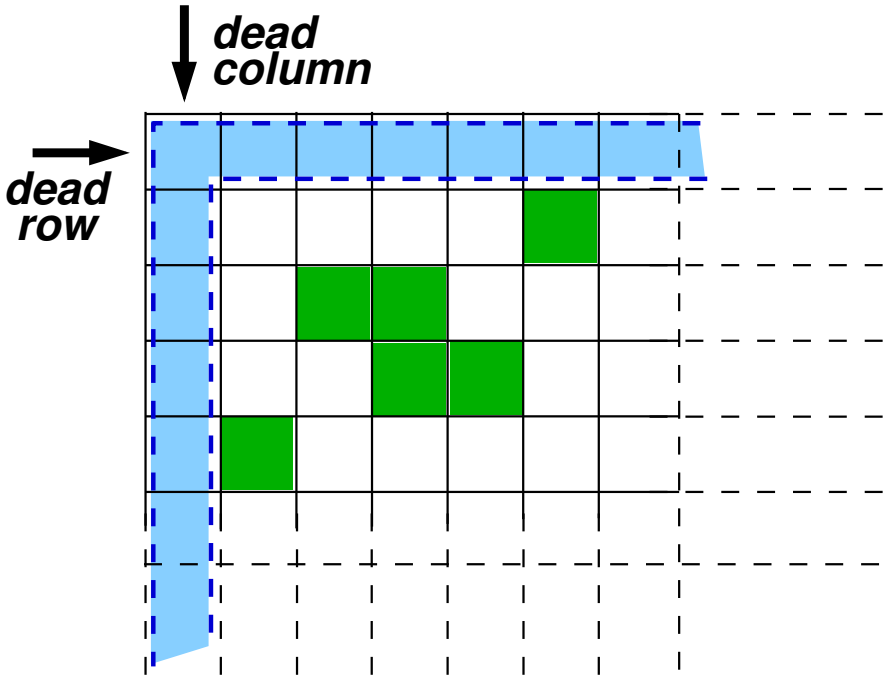
$$a_{ij}^{n+1} = \begin{cases} 1 & ; \text{ if } a_{ij}^n = 3 \\ c_{ij}^n & ; \text{ if } a_{ij}^n = 2 \\ 0 & ; \text{ otherwise.} \end{cases}$$

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

El juego de la vida (cont.)

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

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



PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) es una serie de librerías y estructuras de datos que facilitan la tarea de resolver en forma numérica ecuaciones en derivadas parciales en computadoras de alta performance (HPC). PETSc fue desarrollado en ANL (Argonne National Laboratory, IL) por un equipo de científicos entre los cuales están Satish Balay, William Gropp y otros.

`http://www.mcs.anl.gov/petsc/`

Motivación

- **PETSc usa el concepto de *"programación orientada a objetos "* (OOP) como paradigma para facilitar el desarrollo de programas de uso científico a gran escala**
- **Si bien usa un concepto OOP está escrito en C y puede ser llamado desde C/C++ y Fortran. Las rutinas están agrupadas de acuerdo a los tipos de objetos abstractos con los que operan (e.g. vectores, matrices, solvers...), en forma similar a las clases de C++.**

Objetos PETSc

- **Conjuntos de índices (*"index sets"*), permutaciones, renumeración**
- **vectores**
- **matrices (especialmente ralas)**
- **arreglos distribuidos (mallas estructuradas)**
- **precondicionadores (incluye multigrilla y solvers directos)**
- **solvers no-lineales (Newton-Raphson ...)**
- **integradores temporales nolineales**

Objetos PETSc (cont.)

- **Cada uno de estos tipos consiste de una interface abstracta y una o más implementaciones de esta interface.**
- **Esto permite fácilmente probar diferentes implementaciones de algoritmos en las diferentes fases de la resolución de PDE's como ser los diferentes tipos de métodos de Krylov, diferentes precondicionadores, etc...**
- **Esto promueve la reusabilidad y flexibilidad del código desarrollado.**

Estructura de las librerías PETSc

ver fig. en pag. 15 de <HOME>/PETSC/petsc-2.1.3/docs/manual.pdf

Usando PETSc

Agregar las siguientes variables de entorno

- **`PETSC_DIR` = directorio raíz de la instalacion de PETSc (e.g. `PETSC_DIR=/home/mstorti/PETSC/petsc-2.1.3`)**
- **`PETSC_ARCH` arquitectura de la computadora donde se está usando PETSc (e.g. `PETSC_ARCH=linux`). La utilidad `<PETSC_DIR>/bin/petscarch` permite hallar la arquitectura en tiempo de ejecución (e.g. `PETSC_ARCH='$PETSC_DIR/bin/petscarch'`)**

Usando PETSc

- **PETSc usa MPI para el paso de mensajes de manera que los programas compilados con PETSc deben ser ejecutados con `mpirun` e.g.**

```
sh> mpirun -np 8 <mpi_options> <petsc_program name> \
                 <petsc_options> <user_prog_options>
```

Por ejemplo

```
sh> mpirun -np 8 -machinefile machi.dat \
              my_fem -log_summary -N 100 -M 20
```

Escribiendo programas que usan PETSc

```
1 PetscInitialize (int *argc,char ***argv,
2                  char *file,char *help);   // C
3 call PetscInitialize (character file,
4                       integer ierr)  !Fortran
```

- Llama a `MPI_Init` (si todavía no fue llamado). Si es necesario llamar a `MPI_Init` forzosamente entonces debe ser llamado *antes*.
- Define communicators `PETSC_COMM_WORLD=MPI_COMM_WORLD` y `PETSC_COMM_SELF` (para indicar un solo procesador).

Escribiendo programas que usan PETSc (contd.)

```
1 PetscFinalize (); // C
2 call PetscFinalize (ierr)  ! Fortran
```

Llama a `MPI_Finalize` (si MPI fue inicializado por PETSc).

Ejemplo simple. Ec. de Laplace 1D

Resolver $Ax = b$, donde

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

Ejemplo simple. Ec. de Laplace 1D (contd.)

```
1  /*$Id: ex1.c,v 1.1 2004/03/16 01:39:24 mstorti Exp $*/
2  /* $(PETSC_DIR)/src/sles/examples/tutorials/ex1.c */
3
4  /* Program usage:  mpirun ex1 [-help] [all PETSc options] */
5
6  static char help[] = "Solves a tridiagonal linear system with SLES.\n\n";
7
8  /*T
9     Concepts: SLESˆsolving a system of linear equations
10    Processors: 1
11 T*/
12
13 /*
14   Include "petscsles.h" so that we can use SLES solvers.  Note that this file
15   automatically includes:
16      petsc.h       - base PETSc routines   petscvec.h - vectors
17      petscsys.h    - system routines       petscmat.h - matrices
18      petscis.h     - index sets            petscksp.h - Krylov subspace meth.
19      petscviewer.h - viewers               petscpc.h  - preconditioners
20
21   Note:  The corresponding parallel example is ex23.c
22 */
23 #include "petscsles.h"
24
25 #undef __FUNCT__
```

```
26 #define __FUNCT__ "main"
27 int main(int argc,char **args)
28 {
29   Vec         x, b, u;      /* approx solution, RHS, exact solution */
30   Mat         A;            /* linear system matrix */
31   SLES        sles;         /* linear solver context */
32   PC          pc;           /* preconditioner context */
33   KSP         ksp;          /* Krylov subspace method context */
34   PetscReal   norm;         /* norm of solution error */
35   int         ierr,i,n = 10,col[3],its,size;
36   PetscScalar neg_one = -1.0,one = 1.0,value[3];
37
38   PetscInitialize(&argc,&args,(char *)0,help);
39   ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
40   if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
41   ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
42
43   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
44         Compute the matrix and right-hand-side vector that define
45         the linear system, Ax = b.
46      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
47
48   /*
49      Create vectors.  Note that we form 1 vector from scratch and
50      then duplicate as needed.
51   */
52   ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
53   ierr = PetscObjectSetName((PetscObject) x, "Solution");CHKERRQ(ierr);
54   ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
```

```
55   ierr = VecSetFromOptions(x);CHKERRQ(ierr);
56   ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
57   ierr = VecDuplicate(x,&u);CHKERRQ(ierr);
58
59   /*
60      Create matrix.  When using MatCreate(), the matrix format can
61      be specified at runtime.
62
63      Performance tuning note:  For problems of substantial size,
64      preallocation of matrix memory is crucial for attaining good
65      performance.  Since preallocation is not possible via the generic
66      matrix creation routine MatCreate(), we recommend for practical
67      problems instead to use the creation routine for a particular matrix
68      format, e.g.,
69          MatCreateSeqAIJ() - sequential AIJ (compressed sparse row)
70          MatCreateSeqBAIJ() - block AIJ
71      See the matrix chapter of the users manual for details.
72   */
73   ierr = MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,n,n,&A);
74   CHKERRQ(ierr);
75   ierr = MatSetFromOptions(A);CHKERRQ(ierr);
76
77   /*
78      Assemble matrix
79   */
80   value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
81   for (i=1; i<n-1; i++) {
82     col[0] = i-1; col[1] = i; col[2] = i+1;
83     ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
```

```
84    }
85    i = n - 1; col[0] = n - 2; col[1] = n - 1;
86    ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
87    i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
88    ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
89    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
90    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
91
92    /*
93       Set exact solution; then compute right-hand-side vector.
94    */
95    ierr = VecSet(&one,u);CHKERRQ(ierr);
96    ierr = MatMult(A,u,b);CHKERRQ(ierr);
97
98    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
99                  Create the linear solver and set various options
100      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
101   /*
102      Create linear solver context
103   */
104   ierr = SLESCreate(PETSC_COMM_WORLD,&sles);CHKERRQ(ierr);
105
106   /*
107      Set operators. Here the matrix that defines the linear system
108      also serves as the preconditioning matrix.
109   */
110   ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);
111
112   /*
```

```
113      Set linear solver defaults for this problem (optional).
114      - By extracting the KSP and PC contexts from the SLES context,
115        we can then directly call any KSP and PC routines to set
116        various options.
117      - The following four statements are optional; all of these
118        parameters could alternatively be specified at runtime via
119        SLESSetFromOptions();
120    */
121    ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
122    ierr = SLESGetPC(sles,&pc);CHKERRQ(ierr);
123    ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
124    ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);
125    CHKERRQ(ierr);
126
127    /*
128      Set runtime options, e.g.,
129          -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
130      These options will override those specified above as long as
131      SLESSetFromOptions() is called _after_ any other customization
132      routines.
133    */
134    ierr = SLESSetFromOptions(sles);CHKERRQ(ierr);
135
136    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
137                          Solve the linear system
138       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
139    /*
140       Solve linear system
141    */
```

```
142   ierr = SLESSolve(sles,b,x,&its);CHKERRQ(ierr);
143
144   /*
145      View solver info; we could instead use the option -sles_view to
146      print this info to the screen at the conclusion of SLESSolve().
147   */
148   ierr = SLESView(sles,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
149
150   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
151                       Check solution and clean up
152      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
153   /*
154      Check the error
155   */
156   ierr = VecAXPY(&neg_one,u,x);CHKERRQ(ierr);
157   ierr  = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
158   ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A, Iterations %d\n",norm,
159                      its);CHKERRQ(ierr);
160   /*
161      Free work space.  All PETSc objects should be destroyed when they
162      are no longer needed.
163   */
164   ierr = VecDestroy(x);CHKERRQ(ierr); ierr = VecDestroy(u);CHKERRQ(ierr);
165   ierr = VecDestroy(b);CHKERRQ(ierr); ierr = MatDestroy(A);CHKERRQ(ierr);
166   ierr = SLESDestroy(sles);CHKERRQ(ierr);
167
168   /*
169      Always call PetscFinalize() before exiting a program.  This routine
170        - finalizes the PETSc libraries as well as MPI
```

```
171         - provides summary and diagnostic information if certain runtime
172           options are chosen (e.g., -log_summary).
173    */
174    ierr = PetscFinalize();CHKERRQ(ierr);
175    return 0;
176  }
```

Headers

```
1 #include  "petscsles.h"
```

Ubicados en `<PETSC_DIR>/include`
Los headers para las librerías de más alto nivel incluyen los headers de las librerías de bajo nivel, e.g. `petscsles.h` (linear solvers) incluye

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

Base de datos. Opciones

Se pueden pasar opciones de usuario via

- **archivo de configuración `~/.petscrc`**
- **variable de entorno `PETSC_OPTIONS`**
- **línea de comandos, e.g.: `mpirun -np 1 ex1 -n 100`**

Las opciones se obtienen en tiempo de ejecución de

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

Vectores

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

- `m` = (optional) local (puede ser `PETSC_DECIDE`)
- `M` tamaño global
- `VecSetType()`, `VecSetFromOptions()` permiten definir el tipo de almacenamiento.

Matrices

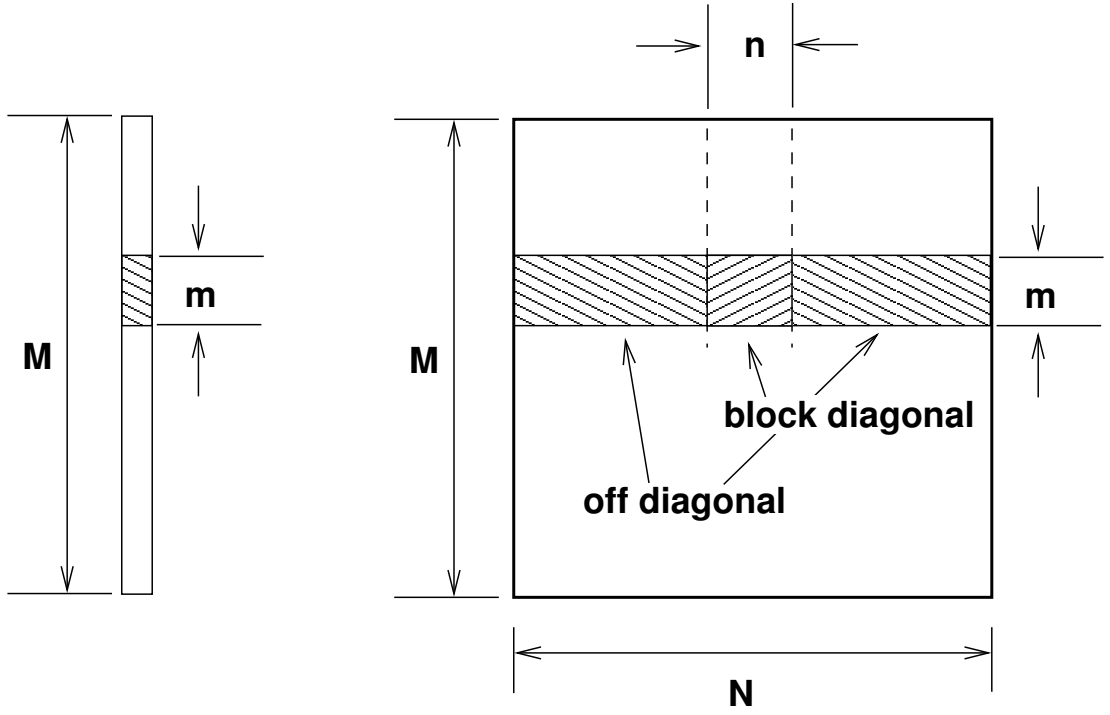
```
1  // Crea una matriz
2  MatCreate(MPI_Comm comm ,int m,
3                  int n,int M ,int N,Mat *A);
4
5  // Setea valores en filas 'im' columnas 'in'
6  MatSetValues(Mat A,int m,int *im,int n,int *in,
7                  PetscScalar *values,INSERT_VALUES);
8
9  // Hacer el ensamblaje antes de usar las matrices
10 // (los valores pueden estar en buffers)
11 MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
12 MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

Linear solvers

```
1  // Crea el SLES
2  SLESCreate(MPI_Comm comm ,SLES *sles);
3
4  // Setea matriz y precondicionamiento
5  SLESSetOperators (SLES sles,Mat A,
6       Mat PrecA,MatStructure flag);
7
8  // Usa opciones de la base de datos (tipo de solver?)
9  //                  (file/env/command)
10 SLESSetFromOptions (SLES sles);
11
12 // Resuelve el sistema lineal
13 SLESSolve (SLES sles,Vec b,Vec x,int *its);
14
15 // Destruye el SLES (libera mem. matris fact.)
16 SLESDestroy (SLES sles);
```

Programación en paralelo

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



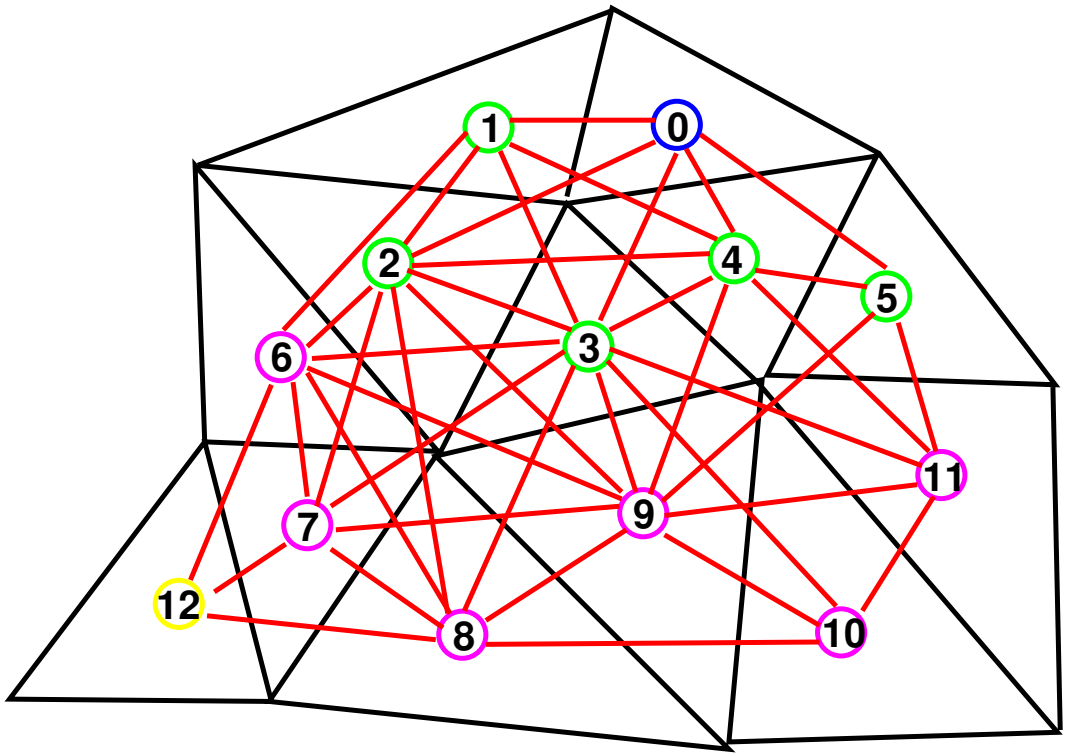
Programa de FEM basado en PETSc

```
1  Leer cordenadas xnod(nnod,ndim=2);
2  Leer conectividades icone(nelem,nel=3);
3  Leer fijaciones (map) fixa: nodo -> valor;
4  Construir (map) n2e:
5             nodo -> lista de elementos conectados;
6  Particionar elementos (grafo dual) por Cuthill-Mc.Kee;
7  Particionar nodos (inducido por particion de elementos);
8  Numerar ecuaciones;
9  Crear matrices y vectores MPI;
10 for (e=1; e<=nelem; e++) {
11   calcular be,ke = rhs y matriz del elemento e en A y b;
12   ensamblar be y ke;
13 }
14 MatAssembly de A y b;
15 Construir sles;
16 Resolver Ax = b;
```

Numerar grafo por Cuthill-Mc.Kee

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

Numerar grafo por Cuthill-Mc.Kee



grafo dual

Particionamiento de elementos

```
1 int nproc;
2 int nelem[nproc];
3 // nelem[proc] es el numero de elementos a poner en
4 // en el procesador 'proc'
5 Numerar elementos por Cuthill-McKee Indx[e];
6 Poner los primeros nelem[0] elementos en proc. 0;
7 Poner los siguientes nelem[1] elementos en proc. 1;
8 ...
9 Poner los siguientes nelem[nproc] elementos en proc. nproc;
```

Particionamiento de nodos

```
1 Dado un nodo n, asignar procesador al cual pertenece
2    cualquier elemento e conectado al nodo;
```

Programa de FEM basado en PETSc (contd.)

```
 1 #include <cstdio>
 2
 3 #include <iostream>
 4 #include <set>
 5 #include <vector>
 6 #include <deque>
 7 #include <map>
 8
 9 #include <petscsles.h>
10
11 #include <newmatio.h>
12
13 static char help[] = "Basic test for self scheduling algorithm FEM with PETSc";
14
15 //-------<*>-------<*>-------<*>-------<*>-------<*>-------
16 #undef __FUNC__
17 #define __FUNC__ "main"
18 int main(int argc,char **args) {
19   Vec b,u;
20   Mat A;
21   SLES sles;
22   PC      pc;                /* preconditioner context */
23   KSP     ksp;                /* Krylov subspace method context */
24
25   // Number of nodes per element
26   const int nel=3;
27   // Space dimension
28   const int ndim=2;
29
30   PetscInitialize(&argc,&args,(char *)0,help);
31
32   // Get MPI info
33   int size,rank;
```

```
34    MPI_Comm_size(PETSC_COMM_WORLD,&size);
35    MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
36
37    // read nodes
38    double x[ndim];
39    int read;
40    vector<double> xnod;
41    vector<int> icone;
42    FILE *fid = fopen("node.dat","r");
43    assert(fid);
44    while (true) {
45      for (int k=0; k<ndim; k++) {
46        read = fscanf(fid,"%lf",&x[k]);
47        if (read == EOF) break;
48      }
49      if (read == EOF) break;
50      xnod.push_back(x[0]);
51      xnod.push_back(x[1]);
52    }
53    fclose(fid);
54    int nnod=xnod.size()/ndim;
55    PetscPrintf(PETSC_COMM_WORLD,"Read %d nodes.\n",nnod);
56
57    // read elements
58    int ix[nel];
59    fid = fopen("icone.dat","r");
60    assert(fid);
61    while (true) {
62      for (int k=0; k<nel; k++) {
63        read = fscanf(fid,"%d",&ix[k]);
64        if (read == EOF) break;
65      }
66      if (read == EOF) break;
67      icone.push_back(ix[0]);
68      icone.push_back(ix[1]);
69      icone.push_back(ix[2]);
70    }
```

```
71    fclose(fid);
72
73    int nelem=icone.size()/nel;
74    PetscPrintf(PETSC_COMM_WORLD,"Read %d elements.\n",nelem);
75
76    // read fixations stored as a map node -> value
77    map<int,double> fixa;
78    fid = fopen("fixa.dat","r");
79    assert(fid);
80    while (true) {
81      int nod;
82      double val;
83      read = fscanf(fid,"%d %lf",&nod,&val);
84      if (read == EOF) break;
85      fixa[nod]=val;
86    }
87    fclose(fid);
88    PetscPrintf(PETSC_COMM_WORLD,"Read %d fixations.\n",fixa.size());
89
90    // Construct node to element pointer
91    // n2e[j-1] is the set of elements connected to node 'j'
92    vector< set<int> > n2e;
93    n2e.resize(nnod);
94    for (int j=0; j<nelem; j++) {
95      for (int k=0; k<nel; k++) {
96        int nodo = icone[j*nel+k];
97        n2e[nodo-1].insert(j+1);
98      }
99    }
100
101 #if 0    // Output 'n2e' array if needed
102   for (int j=0; j<nnod; j++) {
103     set<int>::iterator k;
104     PetscPrintf(PETSC_COMM_WORLD,"node %d, elements: ",j+1);
105     for (k=n2e[j].begin(); k!=n2e[j].end(); k++) {
106       PetscPrintf(PETSC_COMM_WORLD,"%d ",*k);
107     }
```

```
108     PetscPrintf(PETSC_COMM_WORLD,"\n");
109   }
110 #endif
111
112   //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
113   // Simple partitioning algorithm
114   deque<int> q;
115   vector<int> eord,id;
116   eord.resize(nelem,0);
117   id.resize(nnod,0);
118
119   // Mark element 0 as belonging to processor 0
120   q.push_back(1);
121   eord[0]=-1;
122   int order=0;
123
124
125   while (q.size()>0) {
126     // Pop an element from the queue
127     int elem=q.front();
128     q.pop_front();
129     eord[elem-1]=++order;
130     // Push all elements neighbor to this one in the queue
131     for (int nod=0; nod<nel; nod++) {
132       int node = icone[(elem-1)*nel+nod];
133       set<int> &e = n2e[node-1];
134
135       for (set<int>::iterator k=e.begin(); k!=e.end(); k++) {
136         if (eord[*k-1]==0) {
137           q.push_back(*k);
138           eord[*k-1]=-1;
139         }
140       }
141     }
142   }
143   q.clear();
144
145   // Element partition. Put (approximmately) nelem/size in each processor
```

```
146   int *e_indx = new int[size+1];
147   e_indx[0]=1;
148   for (int p=0; p<size; p++)
149     e_indx[p+1] = e_indx[p]+(nelem/size) + (p<(nelem % size) ? 1 : 0);
150
151   // Node partitioning. If a node is connected to an element 'j' then
152   // put it in the processor where element 'j' belongs.
153   // In case the elements connected to the node belong to
154   // different processors take any one of them.
155   int *id_indx = new int[size+2];
156   for (int j=0; j<nnod; j++) {
157     if (fixa.find(j+1) != fixa.end()) {
158       id[j]=-(size+1);
159     } else {
160       set<int> &e = n2e[j];
161       assert(e.size()>0);
162       int order = eord[*e.begin()-1];
163       for (int p=0; p<size; p++) {
164         if (order >= e_indx[p] && order < e_indx[p+1]) {
165           id[j] = -(p+1);
166           break;
167         }
168       }
169     }
170   }
171
172   // 'id_indx[j-1]' is the dof associated to node 'j'
173   int dof=0;
174   id_indx[0]=0;
175   if (size>1)
176     PetscPrintf(PETSC_COMM_WORLD,"dof distribution among processors\n");
177   for (int p=0; p<=size; p++) {
178     for (int j=0; j<nnod; j++)
179       if (id[j]==-(p+1)) id[j]=dof++;
180     id_indx[p+1] = dof;
181     if (p<size) {
182       PetscPrintf(PETSC_COMM_WORLD,
```

```
183                     "proc: %d, from %d to %d, total %d\n",
184                     p,id_indx[p],id_indx[p+1],id_indx[p+1]-id_indx[p]);
185       } else {
186         PetscPrintf(PETSC_COMM_WORLD,
187                     "fixed: from %d to %d, total %d\n",
188                     id_indx[p],id_indx[p+1],id_indx[p+1]-id_indx[p]);
189       }
190     }
191     n2e.clear();
192
193     int ierr;
194     // Total number of unknowns (equations)
195     int neq = id_indx[size];
196     // Number of unknowns in this processor
197     int ndof_here = id_indx[rank+1]-id_indx[rank];
198     // Creates a square MPI PETSc matrix
199     ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD,ndof_here,ndof_here,neq,neq,0,
200                            PETSC_NULL,0,PETSC_NULL,&A); CHKERRQ(ierr);
201     // Creates PETSc vectors
202     ierr = VecCreateMPI(PETSC_COMM_WORLD,ndof_here,neq,&b); CHKERRQ(ierr);
203     ierr = VecDuplicate(b,&u); CHKERRQ(ierr);
204     double scal=0.;
205     ierr =  VecSet(&scal,b);
206
207     { // Compute element matrices and load them.
208       // Each processor computes the elements belonging to him.
209
210       // x12:= vector going from first to second node,
211       // x13:= idem 1->3.
212       // x1:= coordinate of node 1
213       // gN gradient of shape function
214       ColumnVector x12(ndim),x13(ndim),x1(ndim),gN(ndim);
215       // grad_N := matrix whose columns are gradients of interpolation functions
216       // ke:= element matrix for the Laplace operator
217       Matrix grad_N(ndim,nel),ke(nel,nel);
218       // area:= element area
219       double area;
```

```
220     for (int e=1; e<=nelem; e++) {
221       int ord=eord[e-1];
222       // skip if the element doesn't belong to this processor
223       if (ord < e_indx[rank] || ord >= e_indx[rank+1]) continue;
224       // indices for vertex nodes of this element
225       int n1,n2,n3;
226       n1 = icone[(e-1)*nel];
227       n2 = icone[(e-1)*nel+1];
228       n3 = icone[(e-1)*nel+2];
229       x1 << &xnod[(n1-1)*ndim];
230       x12 << &xnod[(n2-1)*ndim];
231       x12 = x12 - x1;
232       x13 << &xnod[(n3-1)*ndim];
233       x13 = x13 - x1;
234       // compute as vector product
235       area = (x12(1)*x13(2)-x13(1)*x12(2))/2.;
236       // gradients are proportional to the edge vector rotated 90 degrees
237       gN(1) = -x12(2);
238       gN(2) = +x12(1);
239       grad_N.Column(3) = gN/(2*area);
240       gN(1) = +x13(2);
241       gN(2) = -x13(1);
242       grad_N.Column(2) = gN/(2*area);
243       // last gradient can be computed from \sum_j grad_N_j = 0
244       grad_N.Column(1) = -(grad_N.Column(2)+grad_N.Column(3));
245       // Integrand is constant over element
246       ke = grad_N.t() * grad_N * area;
247
248       // Load matrix element on global matrix
249       int nod1,nod2,eq1,eq2;
250       for (int i1=1; i1<=nel; i1++) {
251         nod1 = icone[(e-1)*nel+i1-1];
252         if (fixa.find(nod1)!=fixa.end()) continue;
253         eq1=id[nod1-1];
254         for (int i2=1; i2<=nel; i2++) {
255           nod2 = icone[(e-1)*nel+i2-1];
256           eq2=id[nod2-1];
```

```
257             if (fixa.find(nod2)!=fixa.end()) {
258               // Fixed nodes contribute to the right hand side
259               VecSetValue(b,eq1,-ke(i1,i2)*fixa[nod2],ADD_VALUES);
260             } else {
261               // Load on the global matrix
262               MatSetValue(A,eq1,eq2,ke(i1,i2),ADD_VALUES);
263             }
264           }
265         }
266
267       }
268       // Finish assembly.
269       ierr = VecAssemblyBegin(b); CHKERRQ(ierr);
270       ierr = VecAssemblyEnd(b); CHKERRQ(ierr);
271
272       ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
273       ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
274     }
275
276     // Create SLES and set options
277     ierr = SLESCreate(PETSC_COMM_WORLD,&sles); CHKERRQ(ierr);
278     ierr = SLESSetOperators(sles,A,A,
279                             DIFFERENT_NONZERO_PATTERN); CHKERRQ(ierr);
280     ierr = SLESGetKSP(sles,&ksp); CHKERRQ(ierr);
281     ierr = SLESGetPC(sles,&pc); CHKERRQ(ierr);
282
283     ierr = KSPSetType(ksp,KSPGMRES); CHKERRQ(ierr);
284     // This only works for one processor only
285     //  ierr = PCSetType(pc,PCLU); CHKERRQ(ierr);
286     ierr = PCSetType(pc,PCJACOBI); CHKERRQ(ierr);
287     ierr = KSPSetTolerances(ksp,1e-6,1e-6,1e3,100); CHKERRQ(ierr);
288     int its;
289     ierr = SLESSolve(sles,b,u,&its);
290
291     // Print vector on screen
292     ierr =  VecView(u,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
293
294     // Cleanup. Free memory
```

```
295   delete[] e_indx;
296   delete[] id_indx;
297   PetscFinalize();
298
299 }
```