

Computación de Alto Rendimiento. MPI y PETSc

Mario Storti

**Centro de Investigación
de Métodos Computacionales - CIMEC**

(CONICET-UNL), Santa Fe, Argentina

<[mario.storti at gmail.com](mailto:mario.storti@gmail.com)>

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

(document-version "notas.1-1199-g3fa4bdc1")

7 de mayo de 2025

Contents

- [4.....Lista de GTPs](#)
- [7.....MPI - Message Passing Interface](#)
 - ▷ [15.....Uso básico de MPI](#)
 - ▷ [26.....Comunicación punto a punto](#)
 - ▷ [51.....Comunicación global](#)
 - [68.....Ejemplo: cálculo de Pi](#)
 - [86.....Ejemplo: Prime Number Theorem](#)
 - ▷ [142.....Balance de carga](#)
 - [157.....El problema del agente viajero \(TSP\)](#)
 - [180.....Cálculo de Pi por Montecarlo](#)
 - [190.....Ejemplo: producto de matrices en paralelo](#)
 - [213.....El juego de la vida](#)
 - [255.....La ecuación de Poisson](#)
 - ▷ [289.....Operaciones colectivas avanzadas de MPI](#)
 - ▷ [343.....Definiendo tipos de datos derivados](#)
- [360.....PETSc](#)
 - ▷ [363.....Objetos PETSc](#)
 - ▷ [367.....Usando PETSc](#)

- ▷ [387.....Elementos de PETSc](#)
- ▷ [402.....Programa de FEM basado en PETSc](#)
 - [404.....Conceptos básicos de particionamiento](#)
 - [408.....Código FEM](#)
- ▷ [445.....SNES: Solvers no-lineales](#)
- ▷ [455.....TS: time stepping](#)
- [475.....OpenMP](#)
- [525.....Usando MPI y PETSc](#)

Lista de GTPs

Guías de trabajos prácticos

- GTP 1. [BW] Band-Width (Page [42](#))
- GTP 2. [BCAST] My BroadCast (Page [65](#))
- GTP 3. [POIMC] Resolver ec Poisson por Montecarlo (Page [153](#))
- GTP 4. [TSP] Traveling Salesman Problem (Page [177](#))
- GTP 5. [LIFE] Life (Page [252](#))
- GTP 6. [ORDSCAT] Ord-Scatterer (Page [354](#))
- GTP 7. [RICH] Richardson (Page [356](#))
- GTP 8. [POIPETSC] Poisson con PETSc (Page [422](#))
- GTP 9. [SNES] PETSc/SNES. Ejemplo combustión (Page [441](#))
- GTP 10. [PNT] OpenMP PNT (Page [476](#))
- GTP 11. [MCARLO] Pi MonteCarlo OpenMP (Page [477](#))
- GTP 12. [POLMAT] Polinomio de matrices con OpenMP (Page [481](#))
- GTP 13. [MEMBRANE] Membranas 3D con contacto (Page [497](#))

Sobre este curso

- La mayoría de los códigos incluidos se pueden bajar del mismo PDF picando en el clip, por ejemplo:



[Descargar: `hello.cpp`]

- Docentes: Mario Storti, Jorge D'Elía, Victorio Sonzogni
- Pag web del curso:
<http://www.cimec.org.ar/twiki/bin/view/Cimec/CursoHPCenMC> (link corto
<http://goo.gl/YnHNxM>)
- Horarios, lugar
- Temas que se van a dar: MPI, PETSc, OpenMP
- Acceso a los clusters del CIMEC
- Habrá un cierto número de GTP's individuales, con fecha de entrega.
- Evaluación: 4 Trabajos Prácticos de Laboratorio (TPLs)

MPI - Message Passing Interface

El MPI Forum

- Al comienzo de los '90 la gran cantidad de soluciones comerciales y *free* obligaba a los usuarios a tomar una serie de decisiones poniendo en compromiso *portabilidad, performance* y *prestaciones*.
- En abril de 1992 el “*Center of Research in Parallel Computation*” organizó un workshop para definición de estándares para paso de mensajes y entornos de memoria distribuida. Como producto de este encuentro se llegó al acuerdo en definir un estándar para paso de mensajes.

El MPI Forum (cont.)

- En noviembre de 1992 en la conferencia *Supercomputing'92* se formó un comité para definir un *estándar de paso de mensajes*. Los objetivos eran
 - ▷ Definir un estándar para librerías de paso de mensajes. *No sería un estándar oficial tipo ANSI*, pero debería *tentar a usuarios* e implementadores de instancias del lenguaje y aplicaciones.
 - ▷ Trabajar en forma *completamente abierta*. Cualquiera debería poder acceder a las discusiones asistiendo a meetings o via discusiones por e-mail.
 - ▷ El estándar debería estar *definido en 1 año*.
- El MPI Forum decidió seguir el formato del *HPF Forum*.
- Participaron del Forum *vendedores*: Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC, Thinking Machines. *Miembros de librerías preexistentes*: PVM, p4, Zipcode, Chameleon, PARMAKS, TCGMSG, Express.
- Hubo meetings cada 6 semanas durante 1 año e intensa discusión electrónica (los archivos están disponibles desde www.mpiforum.org).
- El estándar fue terminado en *mayo de 1994*.

¿Qué es MPI?

- 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 *lenguaje 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 (accessible at <http://www.mpiforum.org>)
- ***MPI-2: Extensions to the Message-Passing Interface*** November 1996, (accessible at <http://www.mpiforum.org>)
- ***MPI: the complete reference***, by Marc Snir, Bill Gropp, MIT Press (1998) (available in electronic format, `mpi-book.ps`, `mpi-book.pdf`).
- ***Parallel Scientific Computing in C++ and MPI: A Seamless approach to parallel algorithms and their implementations***, by G. Karniadakis y RM Kirby, Cambridge U Press (2003) (u\$s 44.00)
- Páginas ***man (man pages)*** están disponibles en
<http://www-unix.mcs.anl.gov/mpi/www/>
- Newsgroup **`comp.parallel mpi`** (accesible via
<http://groups.google.com/group/comp.parallel mpi>, not too active presently, but useful discussions in the past).
- **[Discussion lists for MPICH and OpenMPI](#)**

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)

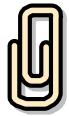


Es una implementación portable de MPI escrita en los *Argonne National Labs (ANL) (USA)*

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

Uso básico de MPI

Hello world en C

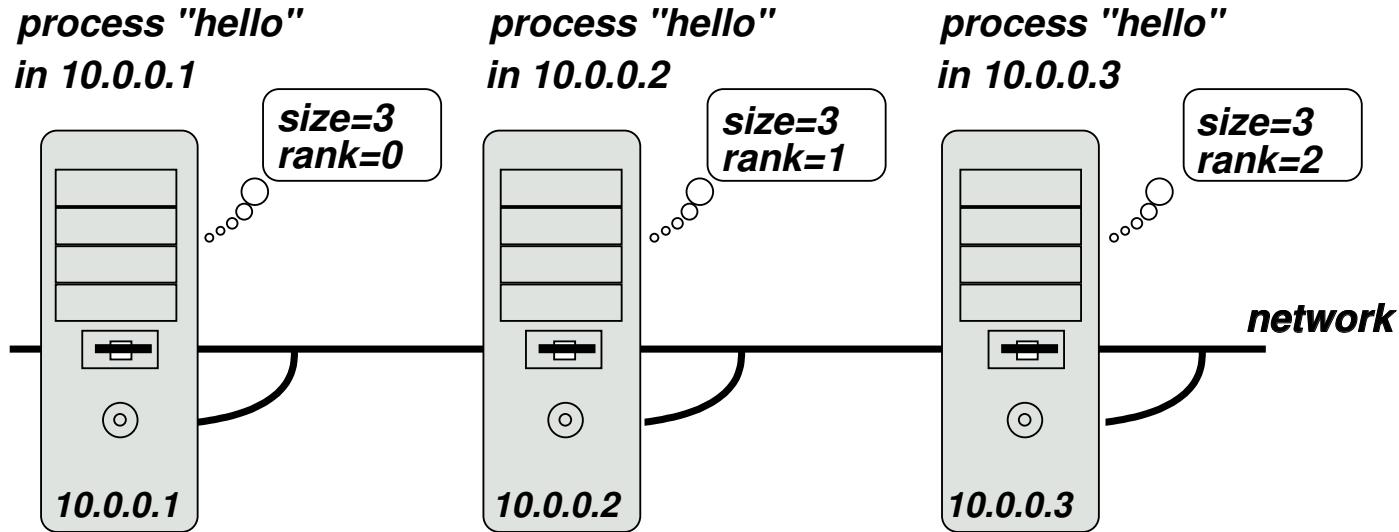


[Descargar: ./example/hello.cpp]

```
1. #include <stdio.h>
2. #include <mpi.h>
3.
4. int main(int argc, char **argv) {
5.     int 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.    return 0;
12. }
```

- Todos los programas empiezan con `MPI_Init()` y terminan con `MPI_Finalize()`.
- `MPI_Comm_rank()` retorna en `rank`, el número de *id* del 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**.
- **Oversubscription:** En general podría haber más de un “**proceso**” en cada “**procesador**” (pero en principio no representará una ganancia).

Hello world en C (cont.)

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

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

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

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

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`.



- 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)



[Descargar: ./example/mslave.cpp]

```
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:



[Descargar: ./example/usempierr.cpp]

```
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 el *id* de este 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*.

Comunicación punto a punto

Enviar un mensaje

- **Template:**

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

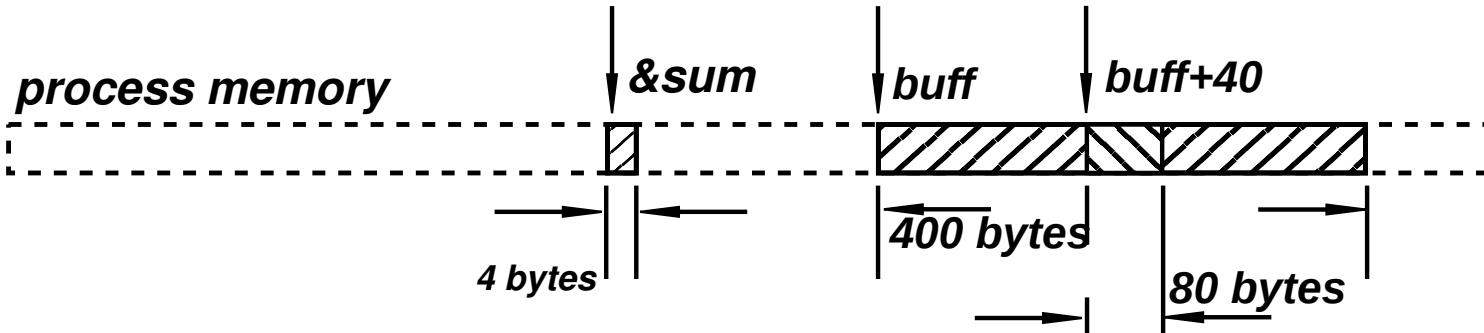
- **C:**

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

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);
```

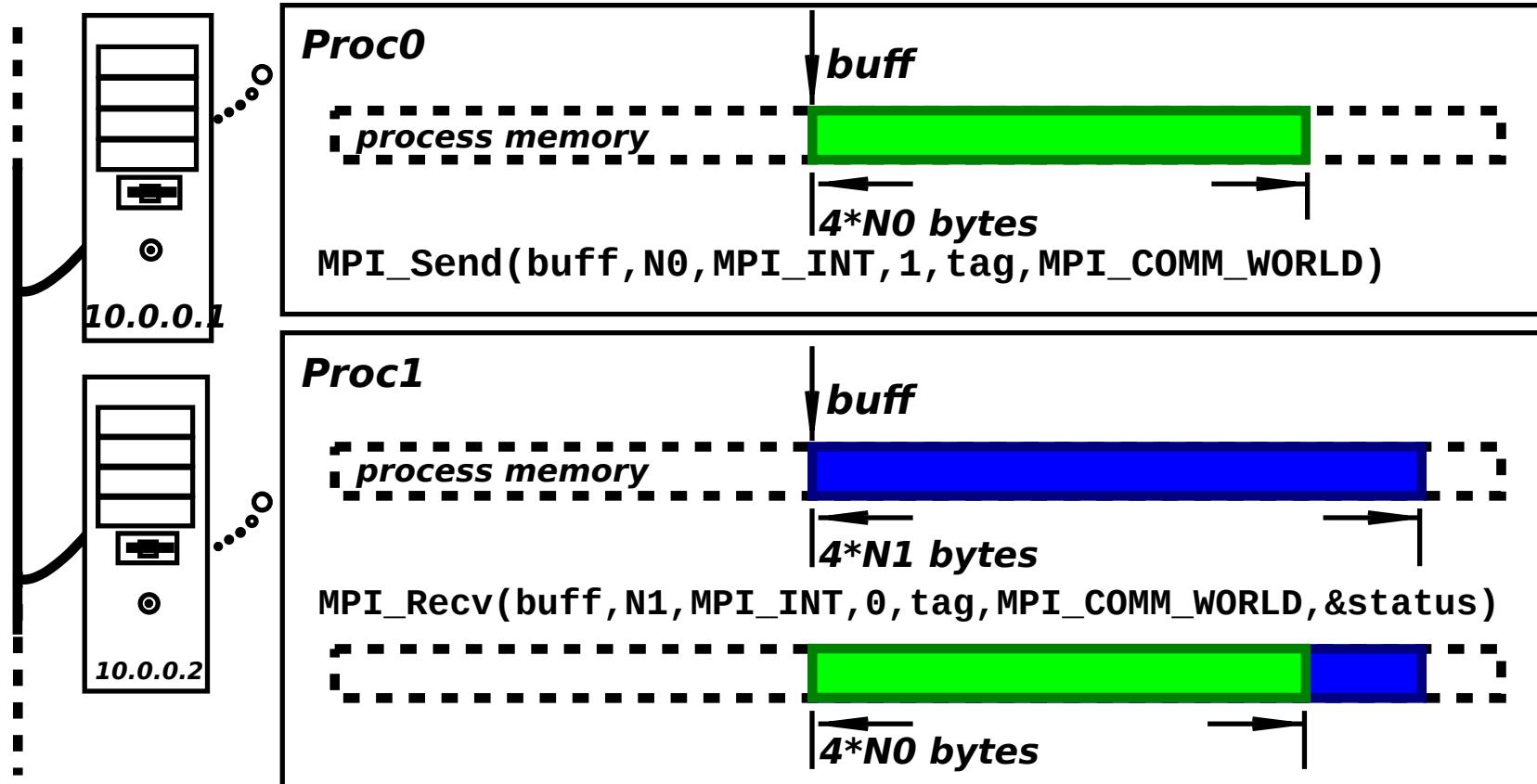
- **Fortran (notar parámetro extra):**

```
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 $N1 \geq N0$

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

- en Fortran, un arreglo de enteros

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

Comunicación punto a punto

A cada `send` debe corresponder un `recv`

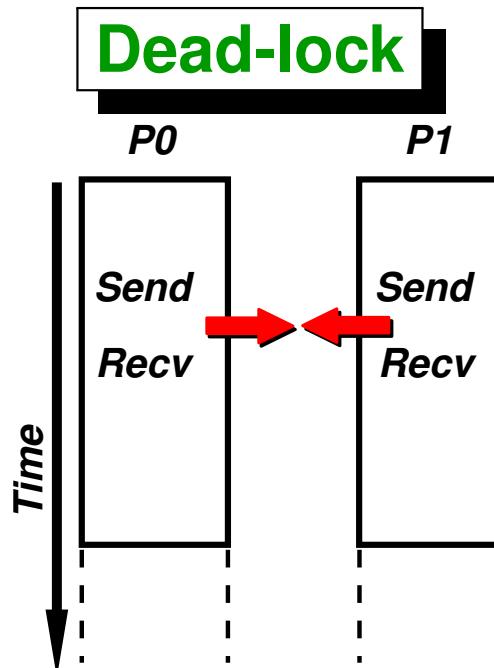
```
1. if (myid==0) {  
2.     for(i=1; i<numprocs; i++)  
3.         MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE,  
4.                     mtag1, MPI_COMM_WORLD, &status);  
5. } else  
6.     MPI_Send(&sum,1,MPI_FLOAT,0,mtag1,MPI_COMM_WORLD);
```

Cuando un mensaje es recibido?

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

envelope = *source/destination, tag, communicator*

- $\text{size(receive buffer)} < \text{size(data sent)}$ → *error*
- $\text{size(receive buffer)} \geq \text{size(data sent)}$ → *OK*
- tipos no concuerdan → *error*

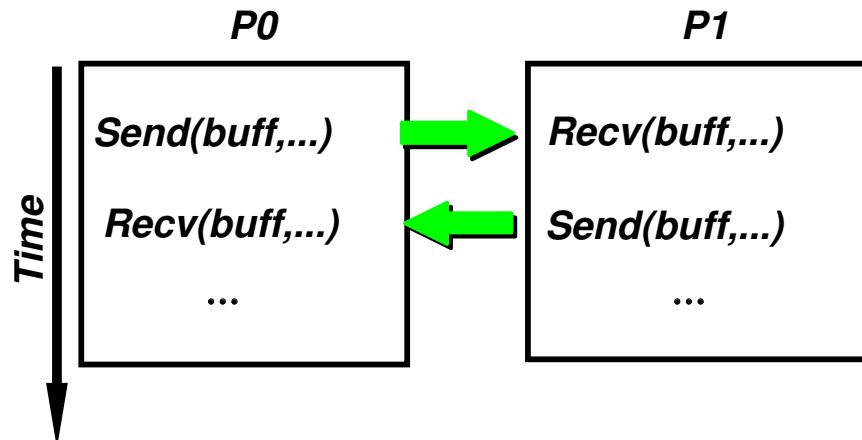


1. `MPI_Send(buff,length,MPI_FLOAT,!myrank,`
2. `tag,MPI_COMM_WORLD);`
3. `MPI_Recv(buff,length,MPI_FLOAT,!myrank,`
4. `tag,MPI_COMM_WORLD,&status);`

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

MPI_Send y **MPI_Recv** son *bloqueantes*, esto significa que la ejecución del código no pasa a la siguiente instrucción hasta que el envío/recepción se *complete*.

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.)

El código previo erróneamente **sobreescribe** el buffer de recepción. Necesitamos

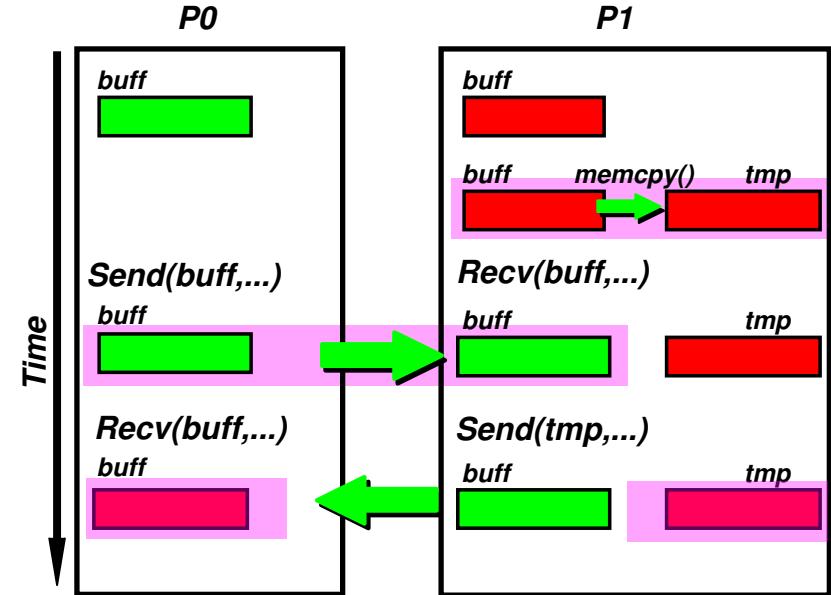


un **buffer temporal** `tmp`:

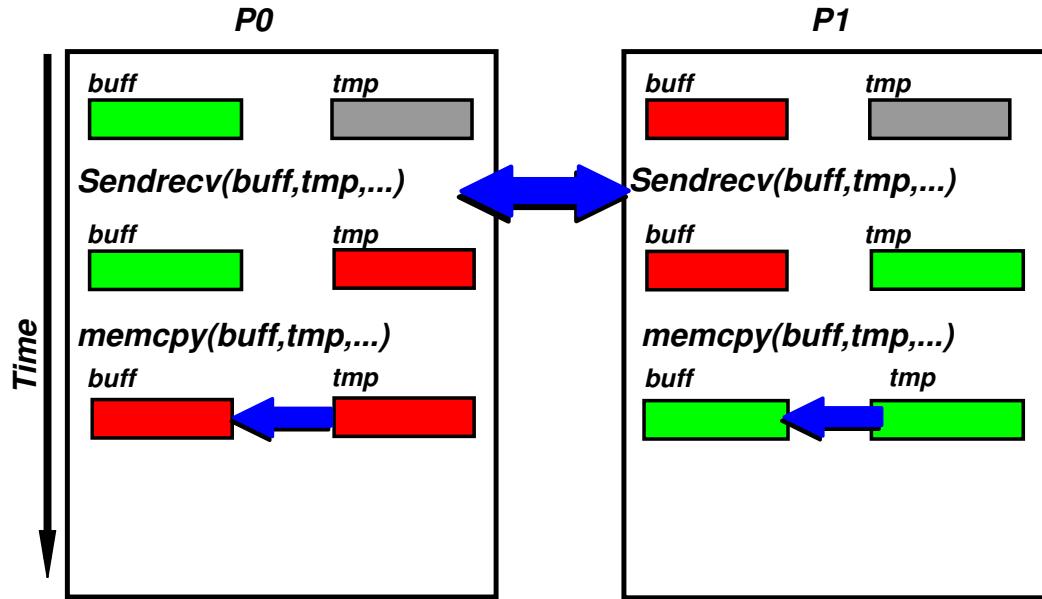
`./example/deadlock.cpp]`

```

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. }
```



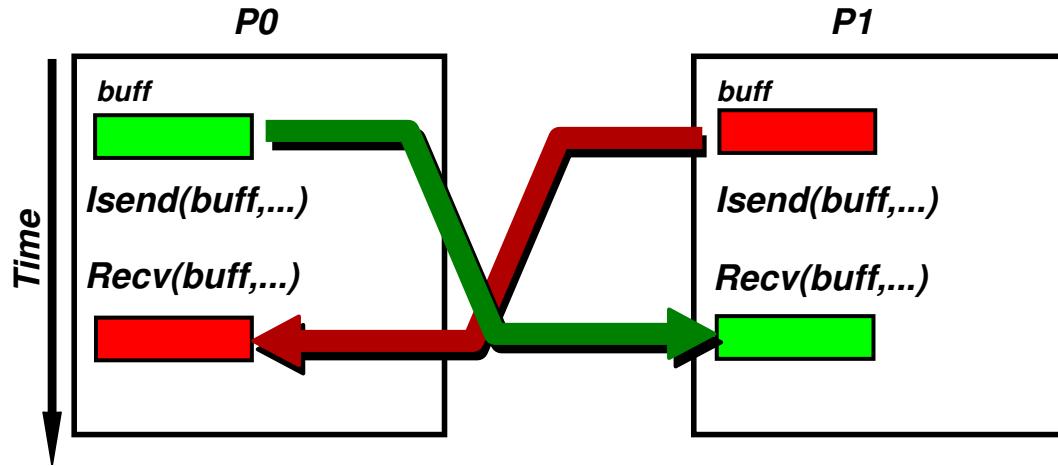
Orden de llamadas correcto (cont.)



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

```
1. float *tmp = new float[length];
2. int MPI_Sendrecv(buff,length,MPI_FLOAT,!myrank,stag,
3.                   tmp,length,MPI_FLOAT,!myrank,rtag,
4.                   MPI_COMM_WORLD,&status);
5. memcpy(buff,tmp,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.
- Necesita un *buffer auxiliar* (no mostrado aquí).

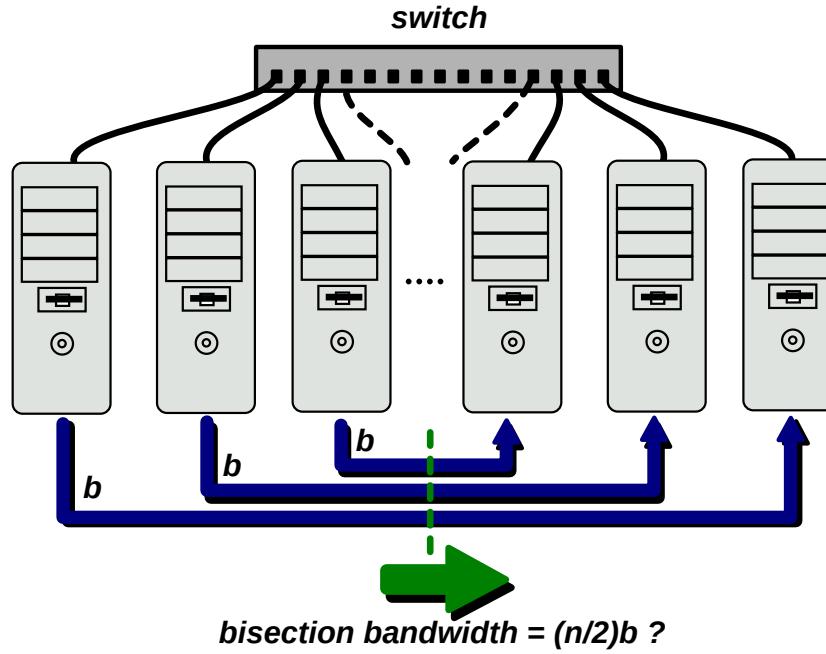
GTP 1. [BW] Band-Width

Dados dos procesadores P_0 y P_1 el tiempo que tarda en enviarse un mensaje desde P_0 hasta P_1 es una función de la *longitud del mensaje* $T_{\text{comm}} = T_{\text{comm}}(n)$, donde n es el número de bits en el mensaje. Si aproximamos esta relación por una *recta*, entonces

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

donde l es la “*latencia*” y b es el “*ancho de banda*”. Ambos dependen del hardware, software de red (capa de TCP/IP en Linux), y la librería de paso de mensajes usada (MPI).

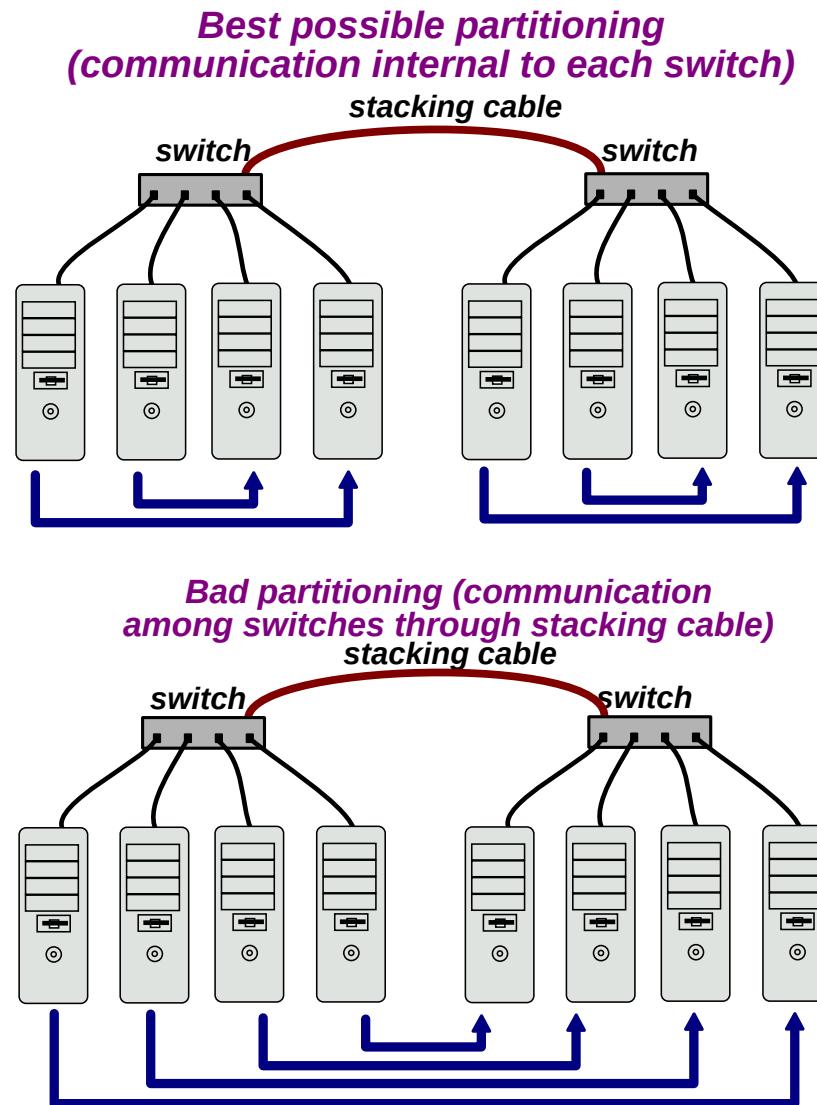
GTP 1. [BW] Band-Width (cont.)



El “**ancho de banda de disección**” de un cluster es la velocidad con la cual se transfieren datos **simultáneamente** desde $n/2$ de los procesadores a los otros $n/2$ procesadores. Asumiendo que la red es **switchada** y que todos los procesadores están conectados al mismo switch, la **velocidad de transferencia** debería ser $(n/2)b$, pero podría ser que el switch tenga una máxima tasa de transferencia interna.

GTP 1. [BW] Band-Width (cont.)

También puede pasar que algunos subconjuntos de nodos tengan un mejor ancho de banda entre sí que con otros, por ejemplo si la red está soportada por dos switches de $n/2$ ports, “**stackeados**” entre sí por un cable de velocidad de transferencia menor a $(n/2)b$. En ese caso el ancho de banda de disección depende de la partición y por lo tanto se define como el correspondiente a la **peor partición**.



GTP 1. [BW] Band-Width (cont.)

Consignas:

Ejercicio 1: Encontrar el *ancho de banda* y *latencia* de la red usada escribiendo un programa en MPI que envía paquetes de diferente tamaño y realiza una *regresión lineal* con los datos obtenidos. Obtener los parámetros para cualquier par de procesadores en el cluster. Comparar con los valores nominales de la red utilizada (por ejemplo, para *Gigabit Ethernet* $b \approx 1 \text{ Gbit/sec}$, $l = O(100 \text{ usec})$, para *Infiniband* $b \approx 40 \text{ Gbit/sec}$, b puede ser **40 Gbps (QDR)**, **56 Gbps (FDR)**, en general **2-300 Gbit/sec**, $l = O(1 \text{ usec})$).

Ejercicio 2: Tomar un número creciente de n procesadores, (10, 20, 40) dividirlos por la mitad y medir el ancho de banda para esa partición. Probar splitting par/ímpar, low-high, y otros que se les ocurra (random?). Comprobar si el ancho de banda de disección crece linealmente con n , es decir si existe algún límite interno para transferencia del switch.

1. `assert(size %2==0); // Be sure we have an even number of procs`
2. `// Splitting even/odd`
3. `if (myrank %2==0) partner = myrank+1;`
4. `else partner = myrank-1;`
- 5.

```
6. // Splitting low/high
7. int half= size/2;
8. if (myrank<half) partner = myrank+half;
9. else partner = myrank-half;
```

En cualquier caso acordarse por supuesto de evitar el deadlock, por ejemplo haciendo que el más bajo transmita primero

```
1. if (myrank<partner) {
2.   Send(....,partner);
3.   Recv(....,partner);
4. } else {
5.   Recv(....,partner);
6.   Send(....,partner);
7. }
```

o usando **Sendrecv()** o funciones no bloqueantes (e.g. **Irecv()**).

GTP 1. [BW] Band-Width (cont.)

Fórmulas para regresión lineal: Si se tienen una serie de mediciones n_j y $T_j = T(n_j)$ ($j = 1, \dots, N$, donde N es el número de mediciones) entonces se pueden calcular el ancho de banda y la latencia con las siguientes expresiones (ver Wikipedia goo.gl/xSoVze)

$$1/b = \frac{\langle Tn \rangle - \langle T \rangle \langle n \rangle}{\langle n^2 \rangle - \langle n \rangle^2},$$

$$l = \langle T \rangle - \langle n \rangle / b,$$

$$\langle n \rangle = 1/N \sum_j n_j, \quad \langle T \rangle = 1/N \sum_j T_j, \quad \langle Tn \rangle = 1/N \sum_j T_j n_j,$$

$$\langle n^2 \rangle = 1/N \sum_j n_j^2.$$

Splitting random

Para hacer el splitting random, hay que generar una apareo random entre los procs. Para esto podemos generar una permutación random `partners` de longitud `size`, y entonces los dos primeros ranks en este arreglo es el primer par, los siguientes dos el segundo par y así siguiendo. Por ejemplo, si la cantidad de procs es `np=6` entonces podemos generar una permutación random `partners=[1,0,5,4,2,3]`. Entonces 1 se va a comunicar con 0, 5 con 4, y 2 con 3.

Para ello primero hay que generar la permutación aleatoria. Esto se puede hacer fácilmente inicializando `partners` con `[0,1...,np-1]` y después aplicando la función `random_shuffle()` que mezcla aleatoriamente el vector.



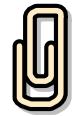
[Descargar: ./example/bwrand.cpp]

```
1. vector<int> partners(size);
2. if (!myrank) {
3.     for (int j=0; j<size; j++)
4.         partners[j] = j;
5.     random_shuffle(partners.begin(),partners.end());
6. }
```

```
7. MPI_Bcast(partners.data(),size,MPI_INT,
```

Atención: El arreglo debe ser el mismo en todos los procesadores de forma que lo mejor es generararlo en el master y bcastearlo al resto.

Una vez que todos los procs tienen el `partners` cada proc busca su id (`myrank`) en el arreglo. Si la posición donde está el `myrank` es par, entonces su par es el siguiente, si no es el anterior. Por ejemplo en el caso anterior si `myrank=5` entonces al buscarlo en `partners` lo encontramos en la posición 2 por lo tanto su partner es el siguiente 4. Para `myrank=3` en cambio, lo encuentra en la posición 5 y por lo tanto su partner es el anterior 2.



[Descargar: ./example/bwrand.cpp]

```
1. // Search my partner
2. int partner=-1,istart=0;
3. for (int j=0; j<size; j++) {
4.     if (partners[j]==myrank) {
5.         if (j%2==0) { istart=1; partner = partners[j+1]; }
6.         else { istart=0; partner = partners[j-1]; }
7.     }
8. }
```

Además, recordemos que para evitar el **dead-lock** cada procesador debe saber si tiene que transmitir primero o recibir primero. Para esto en cada par

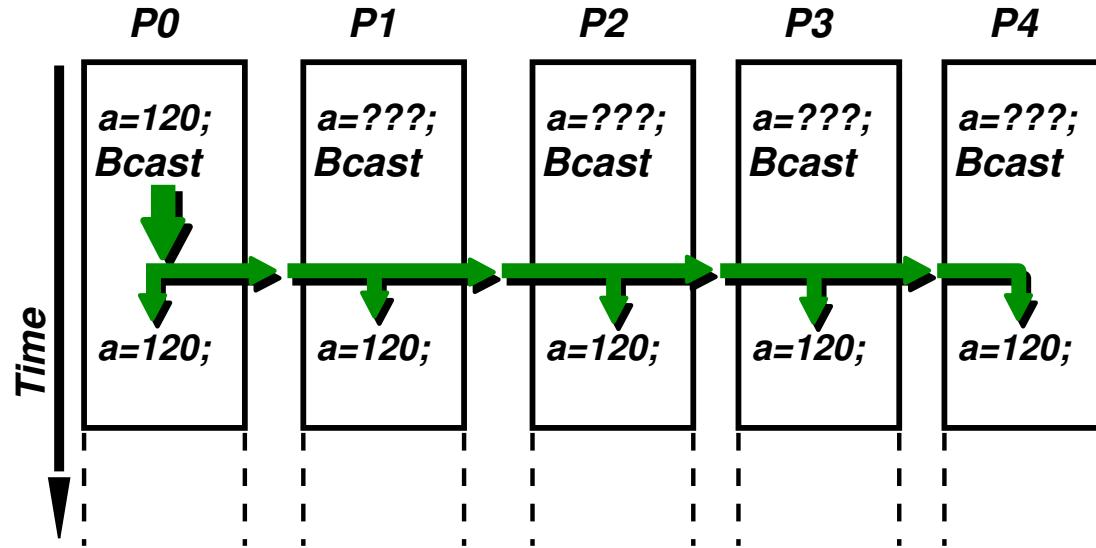
de procs debe decidirse cual de los dos comienza. Un criterio es decir que los que están en la posición par en `partners` empiezan. En el ejemplo anterior los procs 1,5, y 2 empiezan mandando y los procs 0,4,3 empiezan recibiendo.

Para esto definimos un booleano `istart` (“yo comienzo”), y después hacemos

```
1. if (istart) {  
2.   MPI_Send(...,partner,...);  
3.   MPI_Recv(...,partner,...);  
4. } else {  
5.   MPI_Recv(...,partner,...);  
6.   MPI_Send(...,partner,...);  
7. }
```

Comunicación global

Broadcast de mensajes



- **Template:**

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

- **C:**

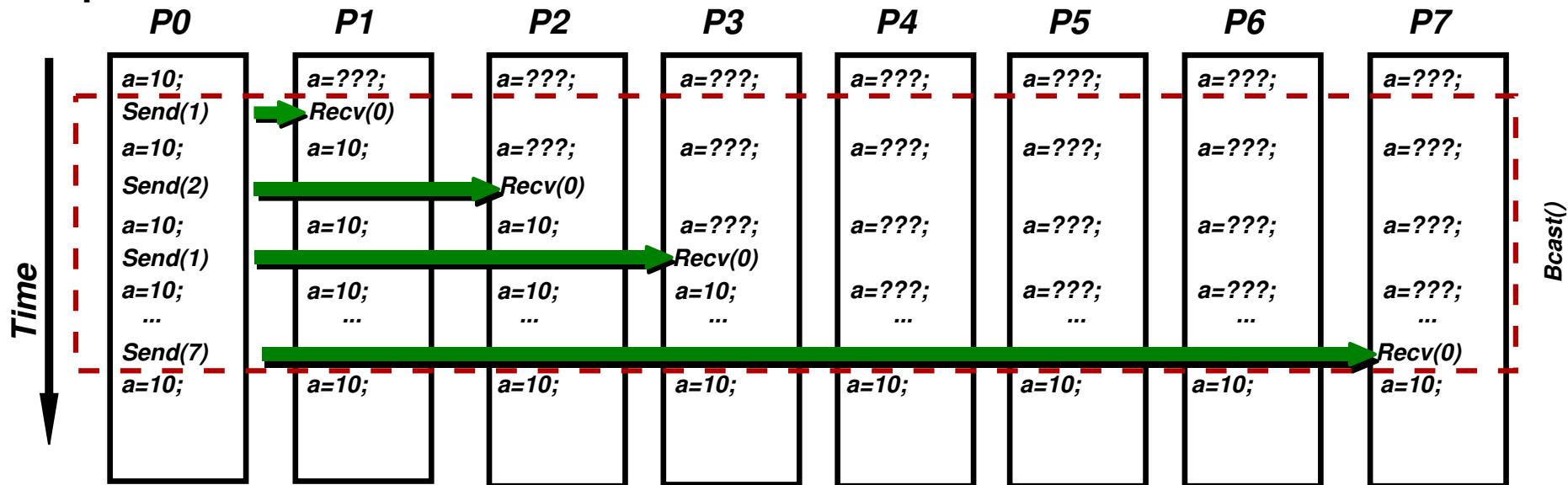
```
ierr = MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- **Fortran:**

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

Broadcast de mensajes (cont.)

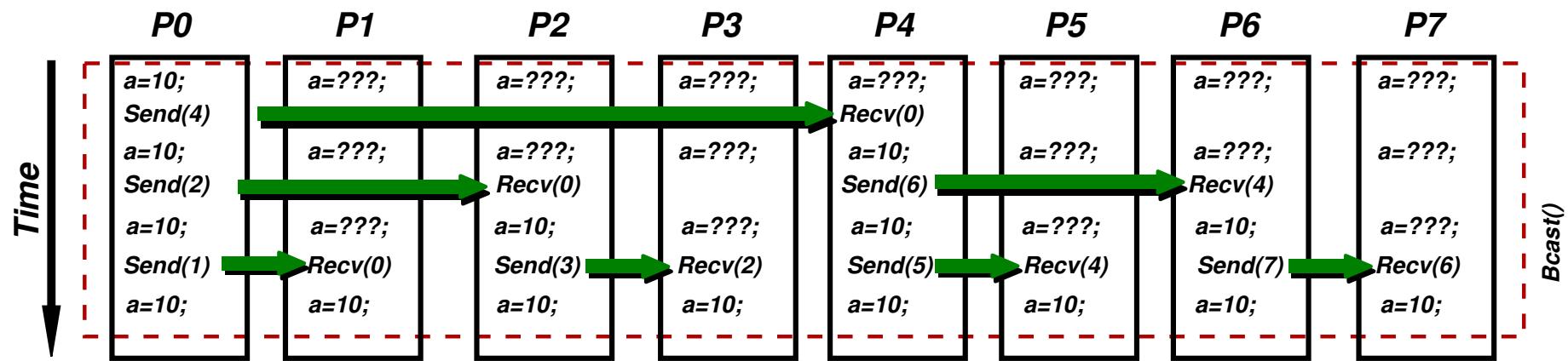
`MPI_Bcast()` es conceptualmente equivalente a una serie de Send/Receive, pero puede ser mucho más eficiente.



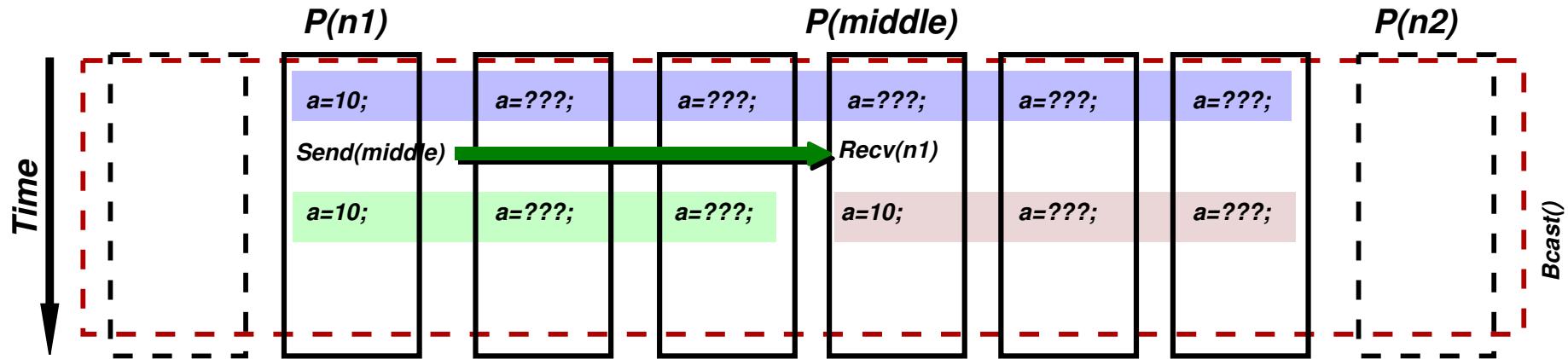
```

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

Broadcast de mensajes (cont.)



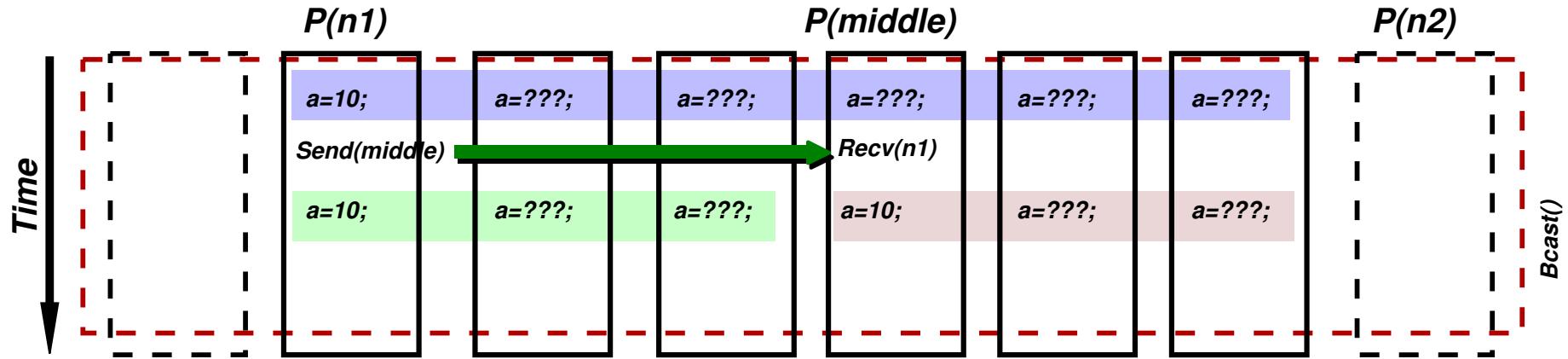
Broadcast de mensajes (cont.)



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

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

Broadcast de mensajes (cont.)



Seudocódigo:

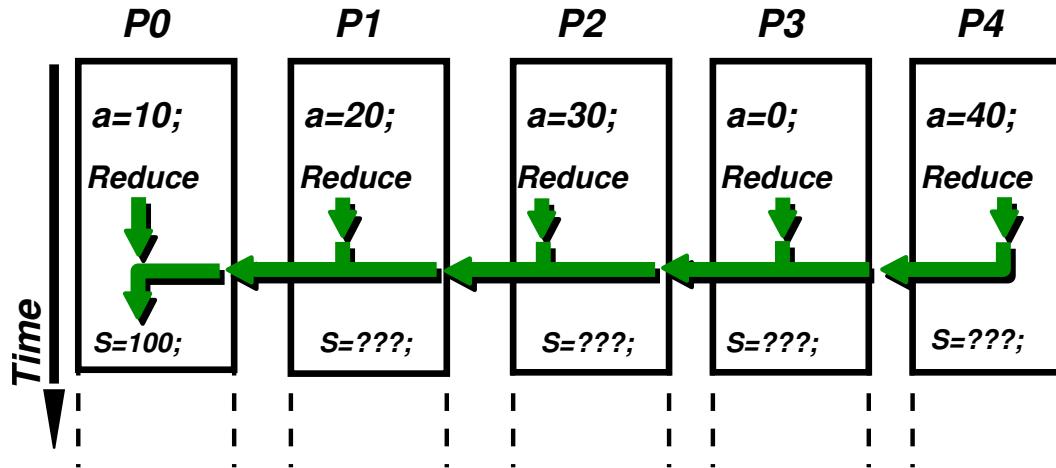
```

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

Llamadas colectivas

Estas funciones con **colectivas** (en contraste con las punto a punto, que son `MPI_Send()` y `MPI_Recv()`). Todos los procesadores en el comunicador deben llamar a la función, y normalmente la llamada colectiva impone una **barrera implícita** en la ejecución del código.

Reducción global



- **Template:**

```
MPI_Reduce(s_address, r_address, length, type, operation,  
destination, comm)
```

- **C:**

```
ierr = MPI_Reduce(&a, &s, 1, MPI_FLOAT, MPI_SUM, 0,  
MPI_COMM_WORLD);
```

- **Fortran:**

```
call MPI_REDUCE(a, s, 1, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD,  
ierr)
```

Operaciones asociativas globales de MPI

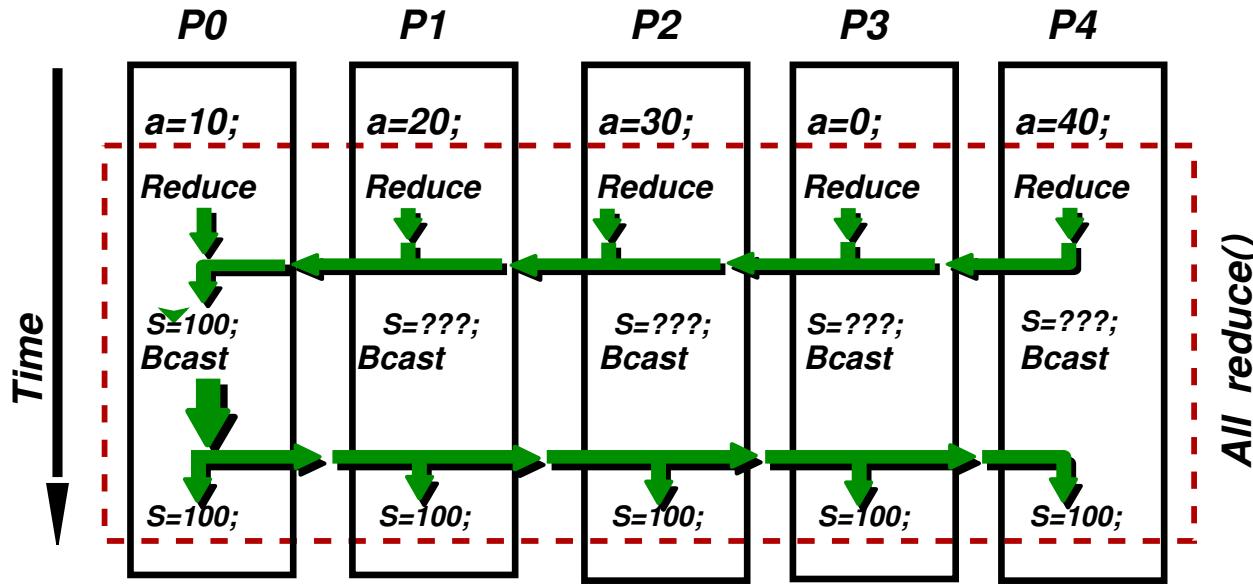
Operaciones de reducción en general aplican una “*operación binaria asociativa*” a un conjunto de valores. Tipicamente,

- `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* $f(x, y)$, commutativa o no.

Incluso esto introduce un cierto grado de indeterminación en el resultado, es decir el resultado no es *determinístico* (a orden precisión de la máquina); puede cambiar entre una corrida y otra, con los mismo datos.

Operaciones asociativas globales de MPI (cont.)



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

```
MPI_Allreduce(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**. Viene con MPICH pero *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 -lmpich -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` se encarga de lanzar el programa en los hosts. Opciones:
 - ▷ `-np <nro-de-procesos>`
 - ▷ `-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`. Una alternativa en versiones recientes de MPICH es `mpiexec` (ver pág [525](#)).

MPI en ambientes Unix (cont.)

- Para que no lance en el server actual usar **-nolocal**.
- Ejemplo:

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

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

GTP 2. [BCAST] My BroadCast

Escribir una rutina `mybcast(...)` con la misma signatura que `MPI_Bcast(...)` mediante send/receive, primero en *forma secuencial* y luego en *forma de árbol* (como fue explicado en pág. 54) . Comparar tiempos, en función del número de procesadores.

NOTAS:

- Primero conviene verificar que el bcast (tanto secuencial como en árbol) funciona bien. Para eso primero probar enviando un entero con un valor cualquiera (por.ej. `a=23`) y verificar que los otros reciben el valor correcto. Luego pueden probar con un vector más grande con los valores todos iguales (por ejemplo en 34) y verificar que después del bcast la suma en todos los procesos es `34*len` (donde `len` es la longitud del vector).
- Una vez que tienen el bcast andando correctamente medir el ancho de banda para diferentes número de procesadores y también comparando secuencial con árbol.
- Versión con `source!=0`: Es recomendable implementar primero los bcast (tanto secuencial como en árbol) primero con `source=0`. El secuencial es sencillo de implementar para `source!=0`, el bcast en árbol es más

complejo, por lo cual incluyo la siguiente ayuda. Los ranks de MPI en cierta forma son arbitrarios, si pudieramos remapearlos de tal forma que el `source` pase a ser el `0` entonces toda la lógica anterior va a funcionar enviando desde el `source`. Por ejemplo podemos hacer un *mapeo cíclico* de los ranks. La lógica sería así, para cada rank **absoluto ARANK** (o sea el rank que da MPI) tenemos un **rank relativo a la fuente RRANK** (en forma cíclica). Por ejemplo si `size=4` y `source=3` tenemos

1	ARANK	RRANK
2	0	1
3	1	2
4	2	3
5	3	0

Nos podemos escribir funciones que mapean de una a otra numeración.

1. // Convert absolute rank to relative
2. int a2r(int arank,int size,int source) { return modulo(arank-source,size); }
3. // Convert relative rank to absolute
4. int r2a(int rrank,int size,int source) { return modulo(rrank+source,size); }

OJO que hay que usar la función `modulo()` (no el operador `%`).

1. int modulo(int x,int a) {
2. int r = x%a;
3. if (r<0) r+=a;
4. }

Para números positivos dan igual pero difieren para números negativos, por ejemplo

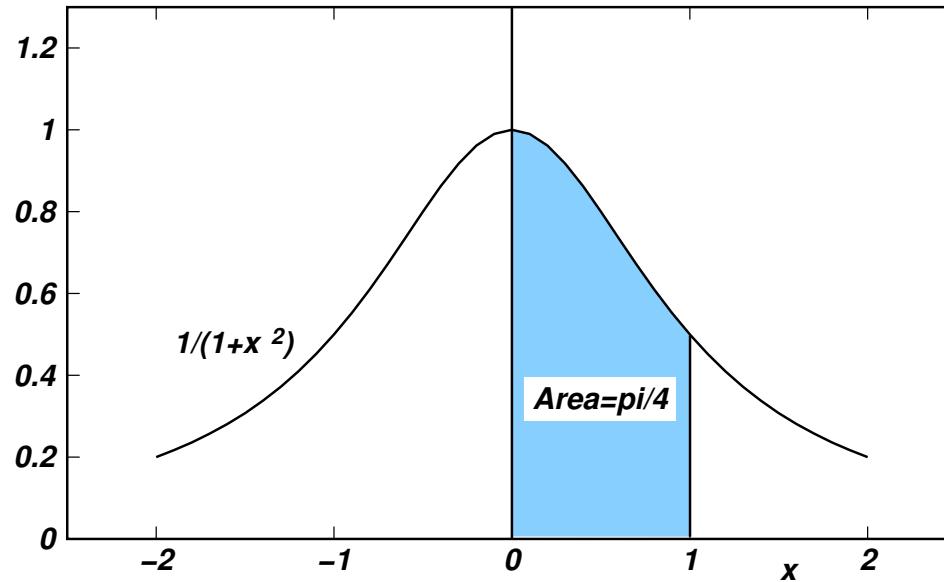
```
1 5 % 3 = 2
2 modulo(5,3) = 2
3
4 -5 % 3 = -2
5 modulo(-5 ,3) = 1
```

Entonces el seudocódigo cambiaría así. Básicamente hacemos toda la algoritmia de send/recv con los ranks relativos, nada más que al momento de transmitir hay que usar los absolutos.

```
1. // Compute my relative rank
2. int rrank = a2r(myrank);
3. int n1=0, n2=numprocs;
4. while (n2-n1>1) {
5.   int middle = (n1+n2)/2;
6.   // Use absolute ranks for the MPI calls!!
7.   if (rrank==n1) MPI_Send(buff,....,r2a(middle),....);
8.   else if (rrank==middle) MPI_Recv(buff,....,r2a(n1),....);
9.   if (rrank<middle) n2 = middle;
10.  else n1=middle;
11. }
```

Ejemplo: cálculo de Pi

Ejemplo cálculo de Pi por integración numérica

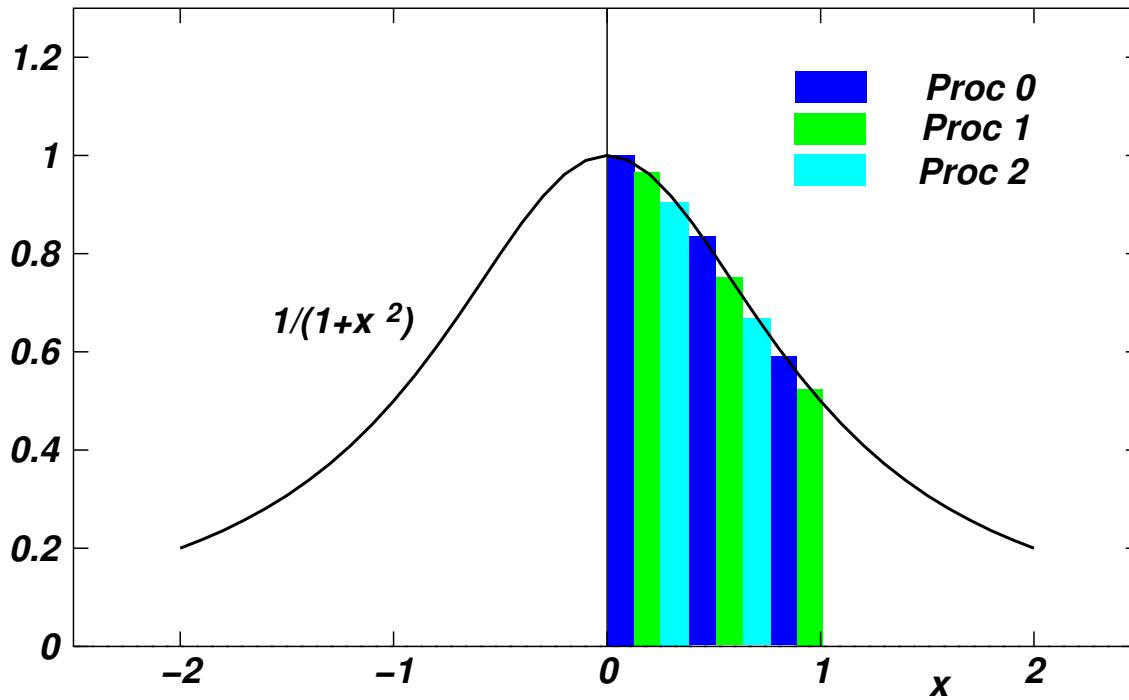


$$\text{atan}(1) = \pi/4$$

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

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

Integración numérica



Usando la *regla del punto medio*

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

Integración numérica (cont.)

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



[Descargar: ./example/pi3.cpp]

```
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 the single integer n to every
60. // other process in processor group MPI_COMM_WORLD.
61. MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
62.
63. // If the user puts in a negative number for n we leave
64. // the program by branching to MPI_FINALIZE
65. if (n<0) break;
66.
67. // Now this part of the code is running on every node
68. // and each one shares the same value of n. But all
69. // other variables are local to each individual
70. // process. So each process then calculates the each
71. // interval size.
72.
73. //*****C
74. //      Main Body : Runs on all processors
75. //*****C
76. // even step size h as a function of partitions
77. double h = 1.0/double(n);
78. double sum = 0.0;
79. for (int i=myrank+1; i<=n; i += size) {
80.     double x = h * (double(i) - 0.5);
81.     sum = sum + f(x);
82. }
83. double pi, mypi = h * sum; // this is the total area
84. // in this process,
85. // (a partial sum.)
86.
```

```
87. // Each individual sum should converge also to PI,
88. // compute the max error
89. double error, my_error = fabs(size*mypi-PI);
90. MPI_Reduce(&my_error,&error,1,MPI_DOUBLE,
91.             MPI_MAX,0,MPI_COMM_WORLD);
92.
93. // After each partition of the integral is calculated
94. // we collect all the partial sums. The MPI_SUM
95. // argument is the operation that adds all the values
96. // of mypi into pi of process 0 indicated by the 6th
97. // argument.
98. MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
99.
100. //*****
101. //      Print results from Process 0
102. //*****
103.
104. // Finally the program tests if myrank is node 0
105. // so process 0 can print the answer.
106. if (!myrank) printf("pi is aprox: %f,\n"
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 numérica (cont.)

Reporte de ps:

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

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

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

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 cálculo de 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 $W_i = W/n$ 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

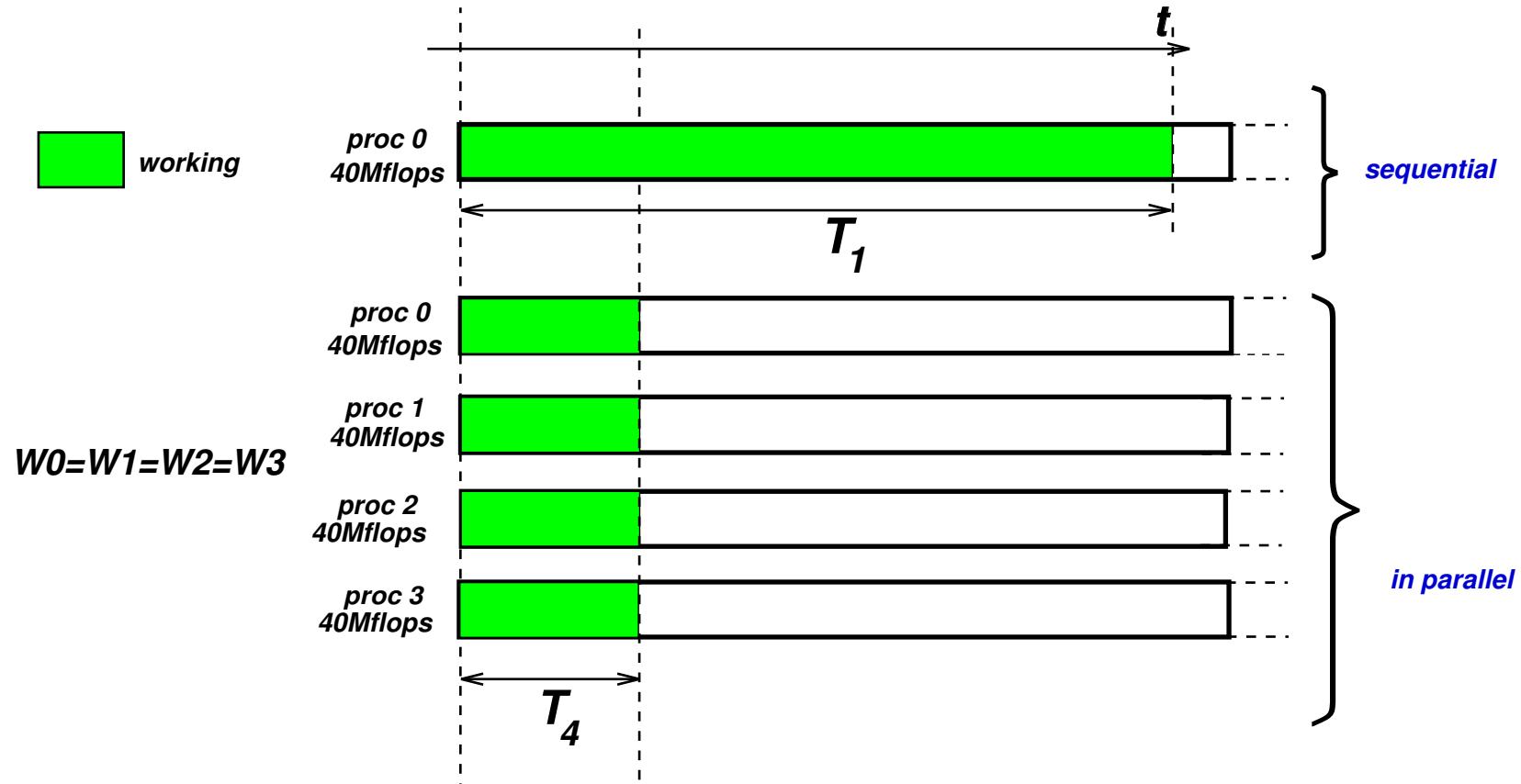
$$T_1 = \frac{W}{s}$$

De manera que

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

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

Conceptos básicos de escalabilidad (cont.)



Conceptos básicos de escalabilidad (cont.)

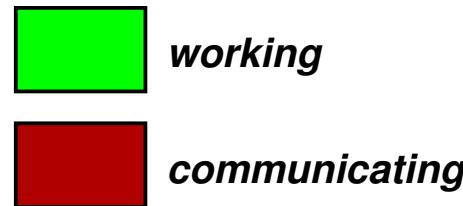
Si el tiempo de comunicación **no es** despreciable

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

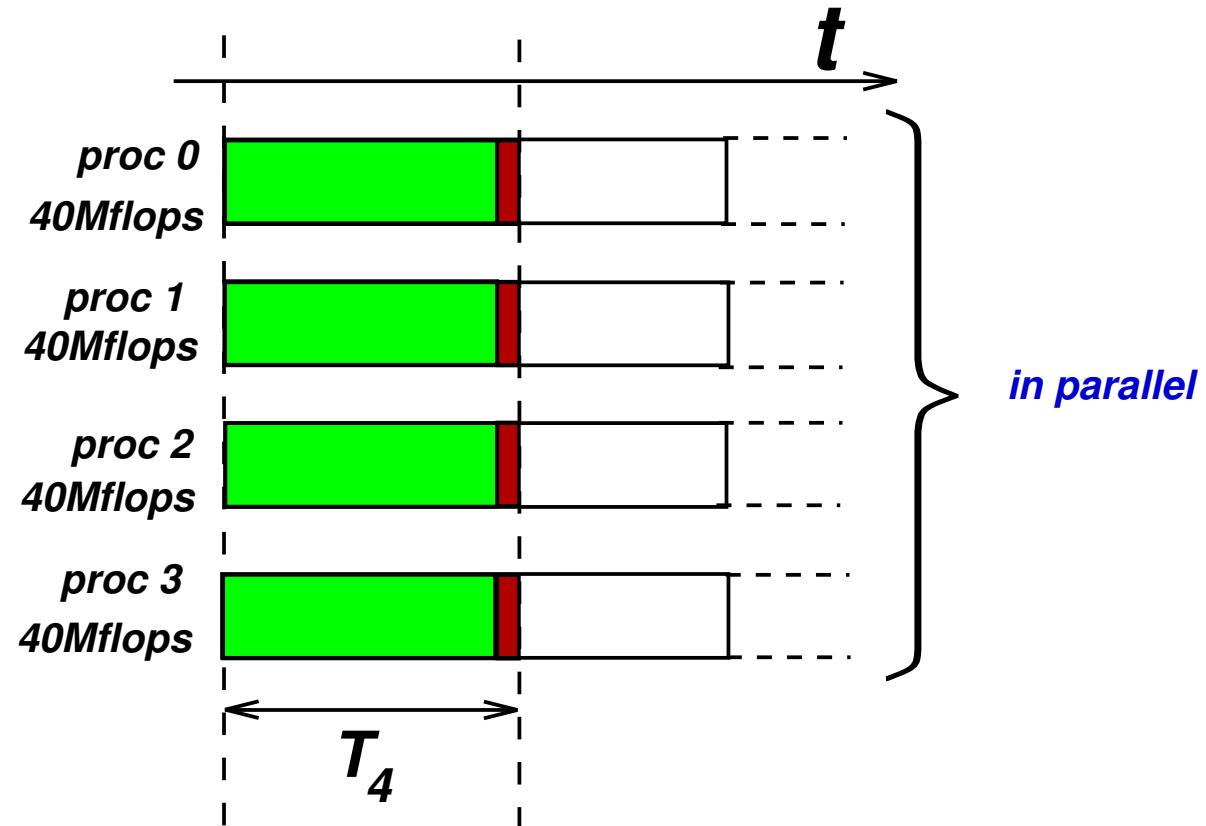
$$S_n = \frac{T_1}{T_n} = \frac{W/s}{W/(sn) + T_{\text{comm}}}$$

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

Conceptos básicos de escalabilidad (cont.)



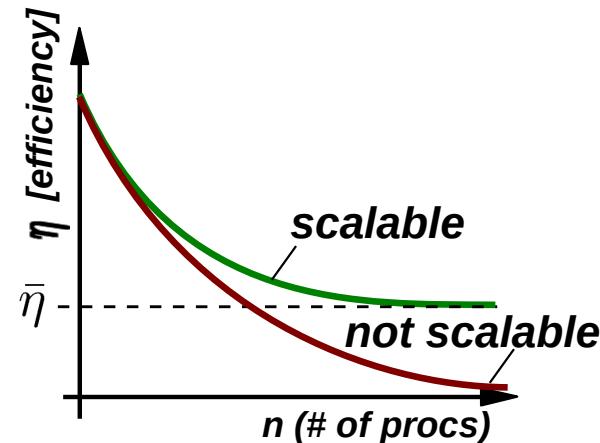
$W_0=W_1=W_2=W_3$



Escalabilidad fuerte

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

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



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 scalable**.

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 scalable**. Esta definición corresponde al criterio de **ESCALABILIDAD FUERTE**.

Escalabilidad débil

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.

$$\eta = \frac{n\bar{W}/s}{(n\bar{W}/s) + nT_{\text{comm}}} = \frac{\bar{W}/s}{(\bar{W}/s) + T_{\text{comm}}} = \text{independiente de } n$$

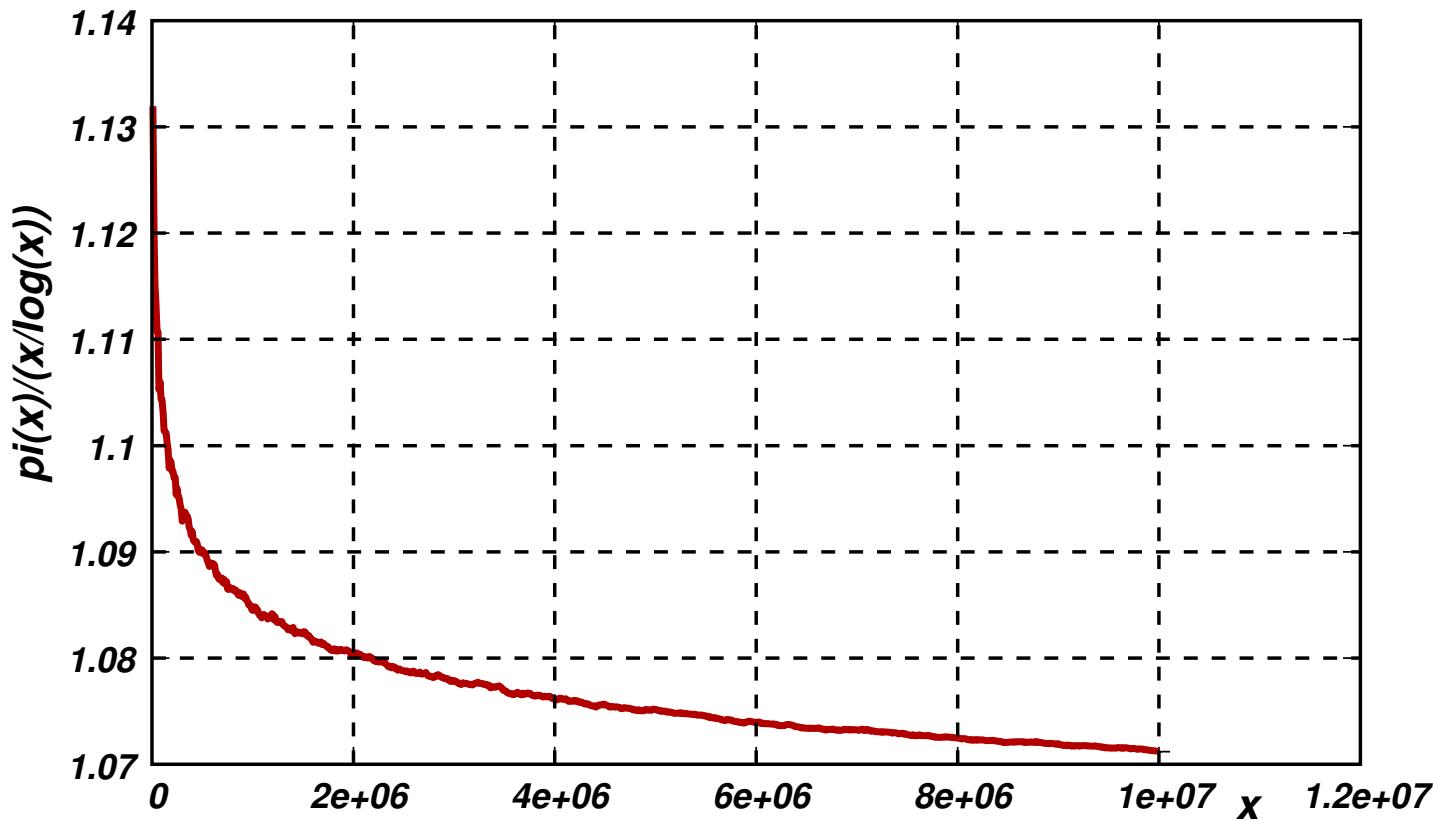
Decimos que la implementación es escalable en el “**límite termodinámico**” si podemos mantener un cierto nivel de eficiencia para $n \rightarrow \infty$ y $W \sim n \rightarrow \infty$. Básicamente esto quiere decir que podemos **resolver problemas cada vez más grandes en el mismo intervalo** de tiempo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

Ejemplo: Prime Number Theorem

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 $\text{floor}(\sqrt{n})$. Si no encontramos ningún **divisor**, entonces es primo

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

De manera que **is_prime(n)** es (en el peor caso) $O(\sqrt{n})$. Como el número de dígitos de n es $n_d = \text{ceil}(\log_{10} n)$, 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 $\sim \sum_{n'=2}^n \sqrt{n'} \sim n^{1.5}$ (por supuesto también no-polinomial). 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

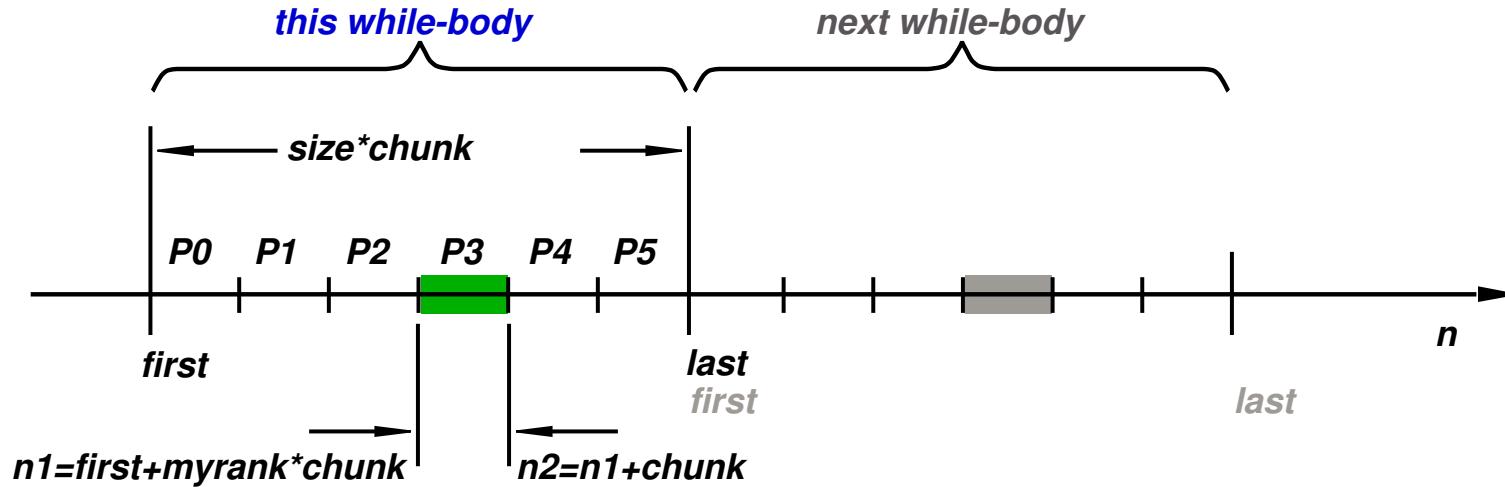


[Descargar: ./example/primes1.cpp]

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

PNT: Versión paralela

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



PNT: Versión paralela. Seudocódigo

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

PNT: Versión paralela. Código

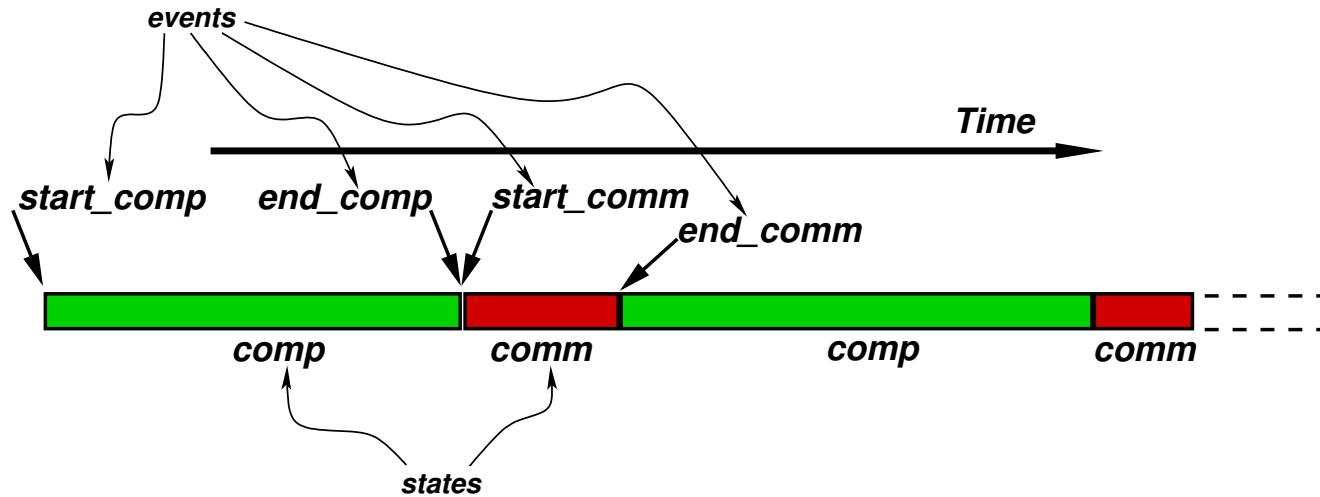


[Descargar: ./example/primes2.cpp]

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

MPE Logging



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

MPE Logging (cont.)



[Descargar: ./example/primessc.cpp]

```
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**.

MPE Logging (cont.)



[Descargar: ./example/primes3.cpp]

```
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.     if (n<2) return 0;  
9.     int m = int(sqrt(n));  
10.    for (int j=2; j<=m; j++)  
11.        if (!(n% j)) return 0;  
12.    return 1;  
13. }  
14.  
15. int main(int argc, char **argv) {  
16.     MPI_Init(&argc,&argv);  
17.     MPE_Init_log();  
18.  
19.     int myrank, size;  
20.     MPI_Comm_rank(MPI_COMM_WORLD,&myrank);  
21.     MPI_Comm_size(MPI_COMM_WORLD,&size);
```

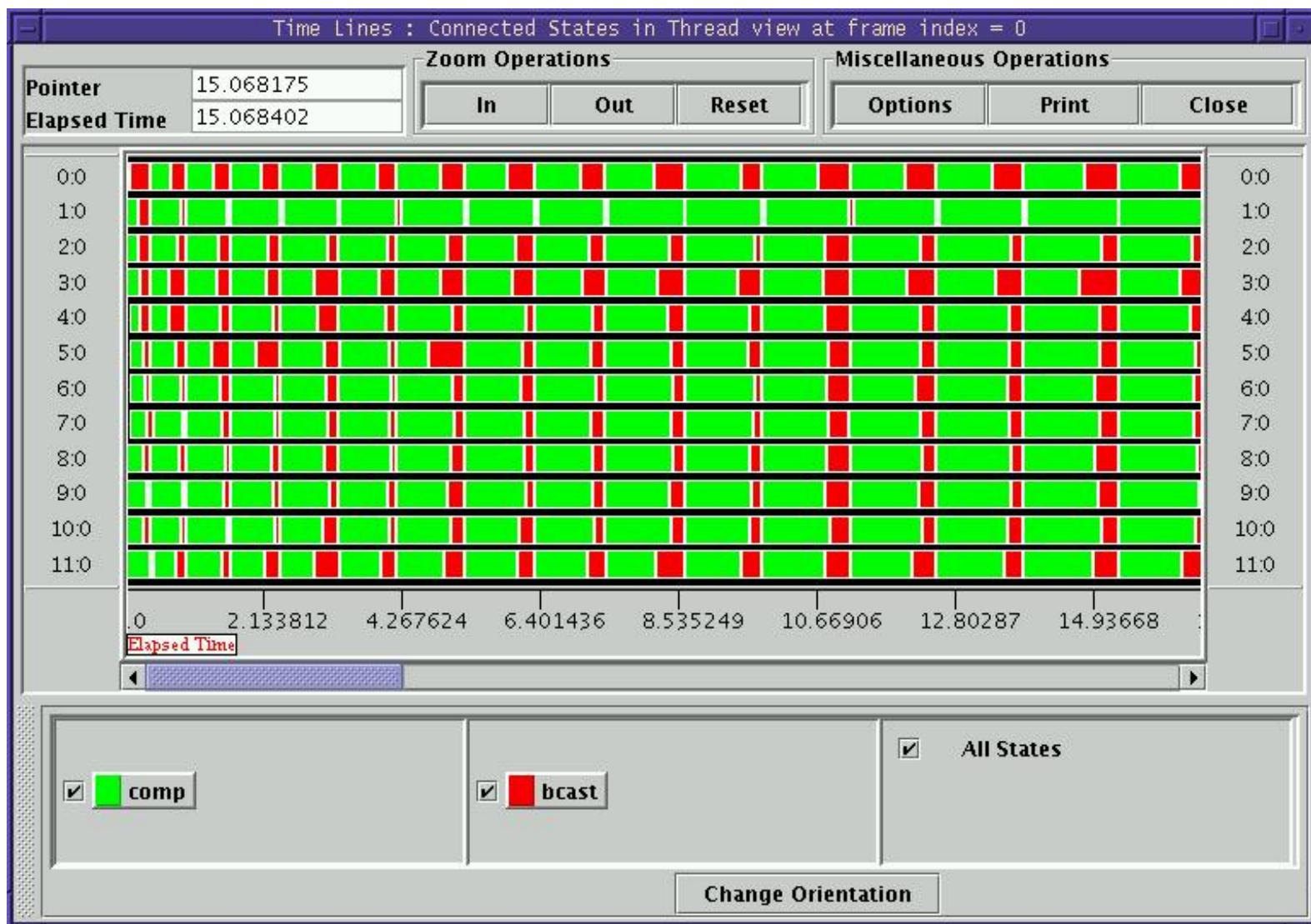
```
22. int start_comp = MPE_Log_get_event_number();
23. int end_comp = MPE_Log_get_event_number();
24. int start_bcast = MPE_Log_get_event_number();
25. int end_bcast = MPE_Log_get_event_number();
26.
27. int n2, primesh=0, primes, chunk=200000,
28.     n1, first=0, last;
29.
30. if (!myrank) {
31.     MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
32.     MPE_Describe_state(start_bcast,end_bcast,"bcast","red:white");
33. }
34.
35. while(1) {
36.     MPE_Log_event(start_comp,0,"start-comp");
37.     last = first + size*chunk;
38.     n1 = first + myrank*chunk;
39.     n2 = n1 + chunk;
40.     for (int n=n1; n<n2; n++) {
41.         if (is_prime(n)) primesh++;
42.     }
43.     MPE_Log_event(end_comp,0,"end-comp");
44.     MPE_Log_event(start_bcast,0,"start-bcast");
45.     MPI_Allreduce(&primesh,&primes,1,MPI_INT,
46.                   MPI_SUM,MPI_COMM_WORLD);
47.     first += size*chunk;
48.     if (!myrank) printf("pi(%d)=%d\n",last,primes);
```

```
49.     MPE_Log_event(end_bcast,0,"end-bcast");
50.     if (last>=10000000) break;
51. }
52.
53. MPE_Finish_log("primes");
54. MPI_Finalize();
55. }
```

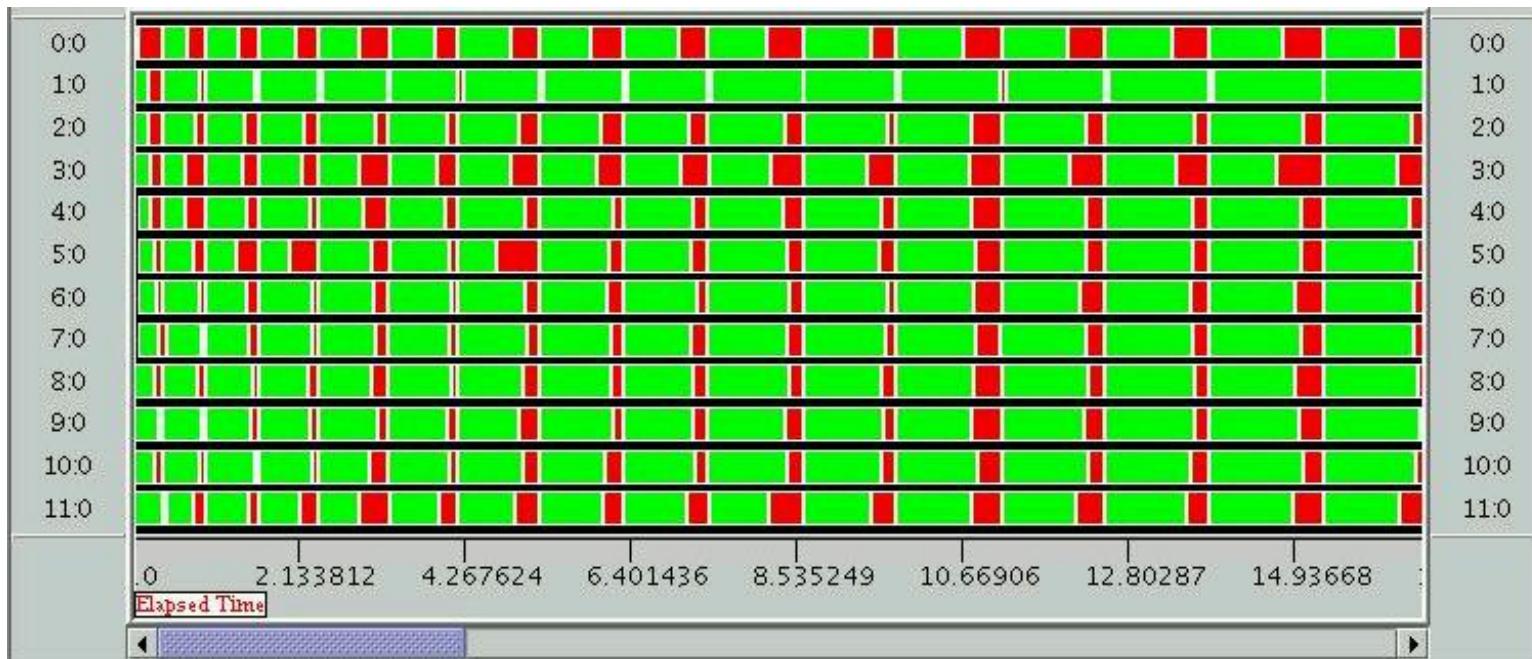
Uso de Jumpshot

- “**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.)

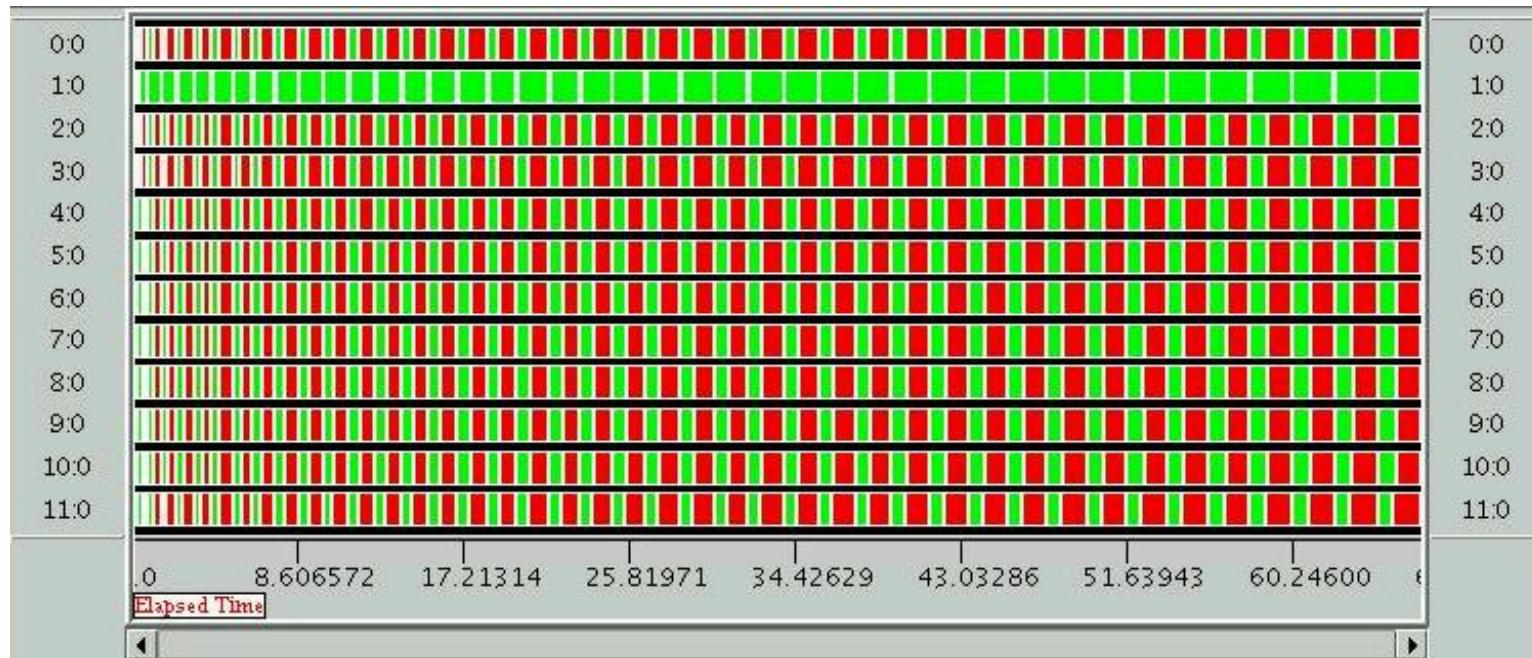


PNT: $np = 12$ (blocking)



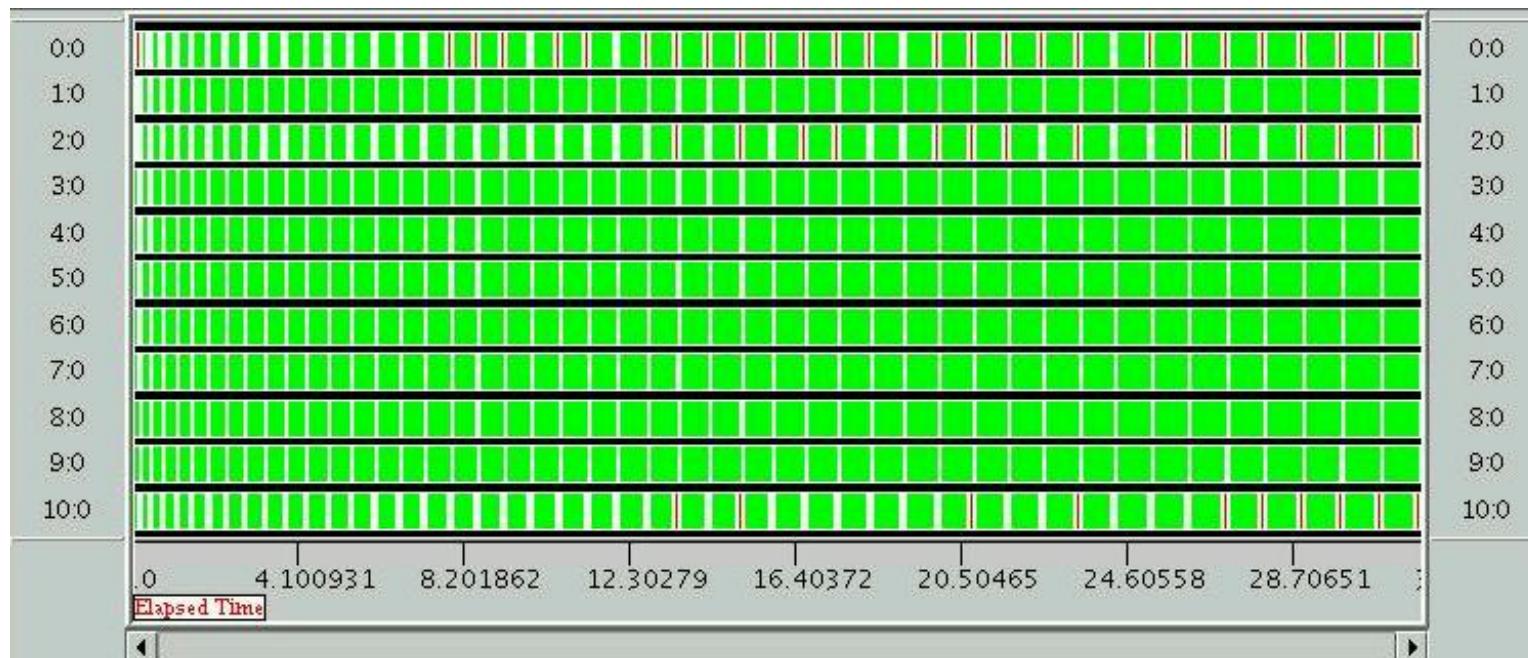
- Nodos están corriendo otros programas (con carga variable).
- `comm` es en realidad **comunicación+sincronización**.
- `MPI_Allreduce()` actúa como una barrera de sincronización (todos esperan hasta que **todos** lleguen a la barrera).
- Proc 1 es lento y desbalancea (los otros lo tienen que esperar)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

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



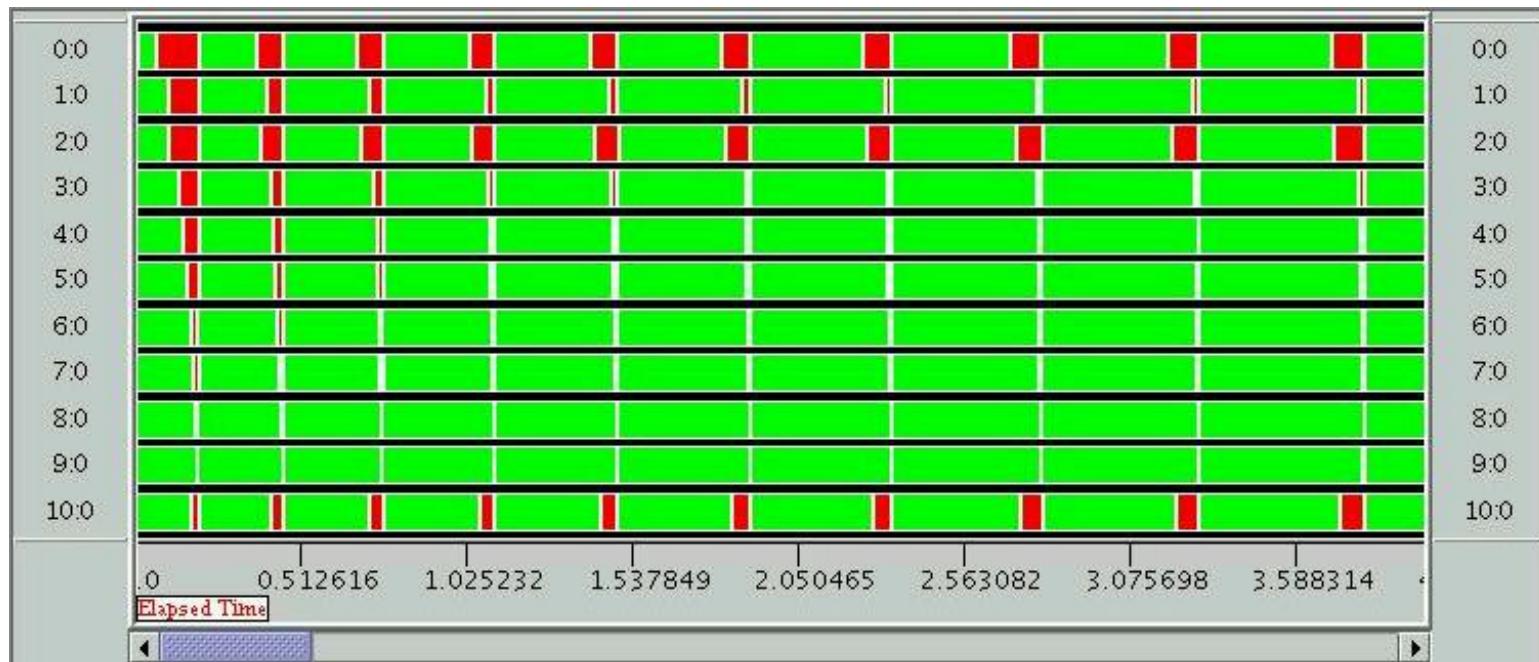
- Proc 1 está corriendo otros programas. Otros procs sin carga.
- Desbalance aún mayor.

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



- Sin el node12 (proc 1 en el ejemplo anterior). (buen balance)

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



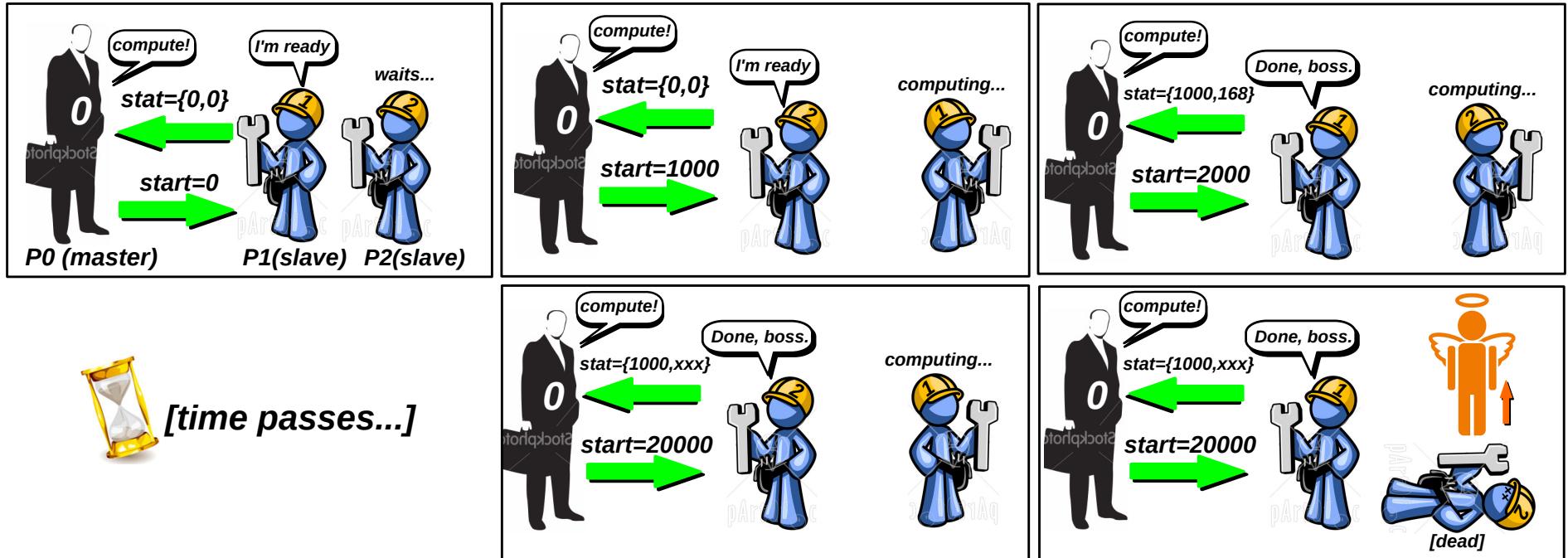
- Sin el procesador (proc 1 en el ejemplo anterior). (buen balance, detalle)

PNT: versión paralela dinámica.

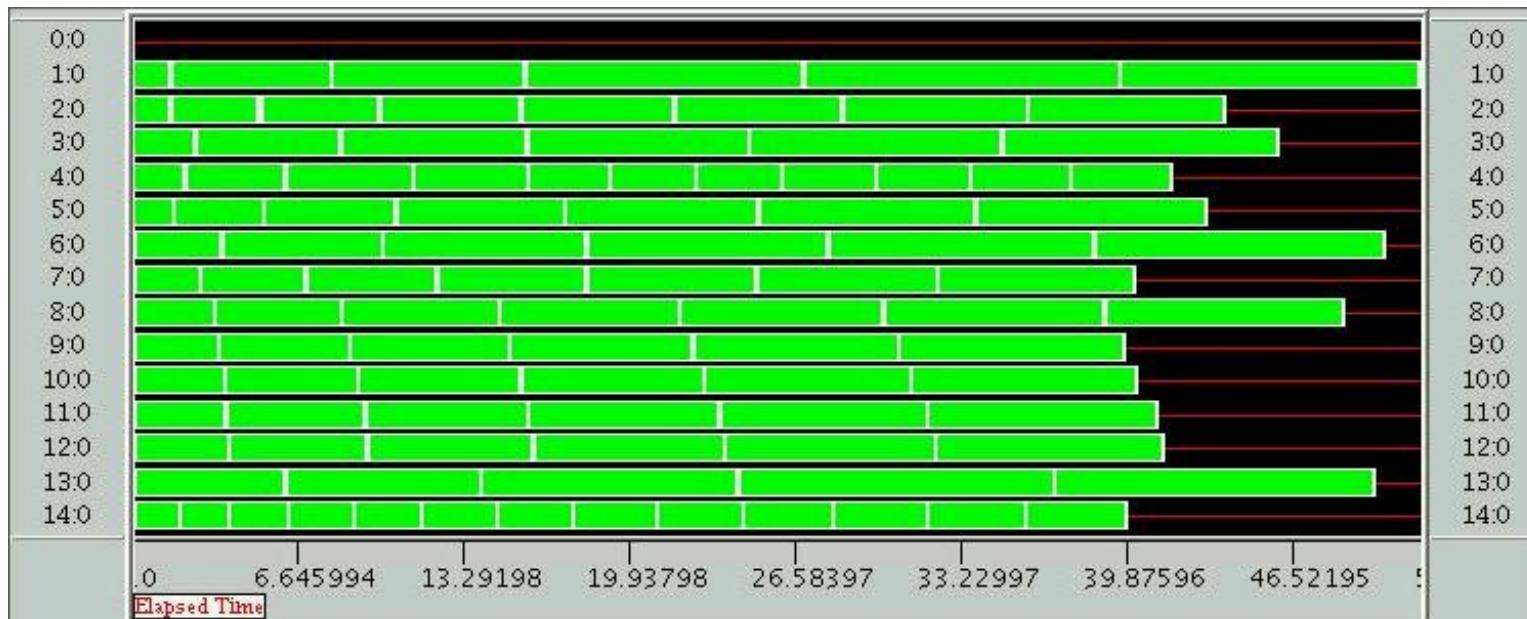
- Proc 0 actúa como servidor.
- Esclavos mandan al master el “*trabajo realizado*” y éste les devuelve el nuevo trabajo a realizar.
- Trabajo realizado es `stat[2]`: número de enteros chequeados (`stat[0]`) y primos encontrados (`stat[1]`).
- Trabajo a realizar es un solo entero: `start`. El esclavo 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 `start>N`. Cuando este contador llega a `size-1` el servidor también para.

PNT: versión paralela dinámica. (cont.)

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

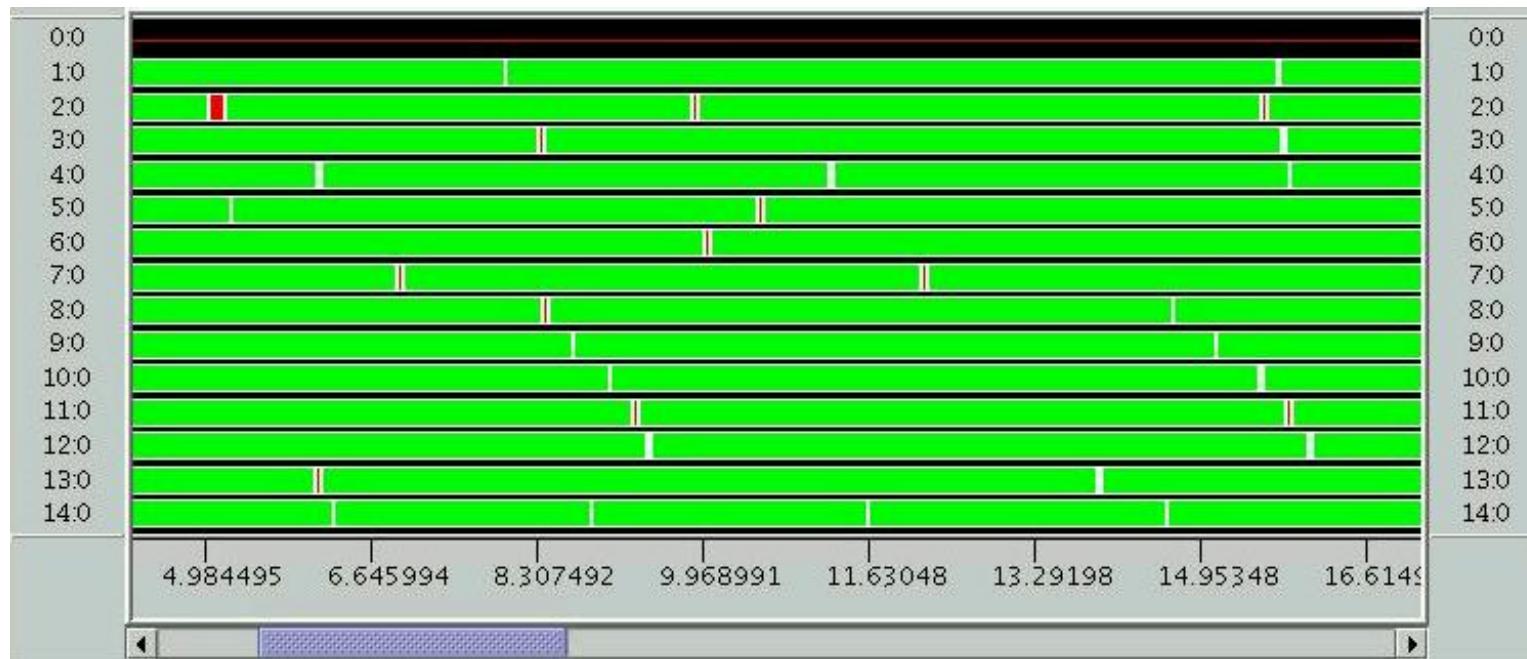


PNT: versión paralela dinámica. (cont.)

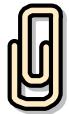


- Muy buen *balance dinámico*.
- Ineficiencia por *alta granularidad* al final (desaparece a medida que hacemos crecer el problema.)
- Notar ciertos procs (4 y 14) procesan chunks a una relación 3:1 con los *más lentos* (1 y 13).
- Llevar estadística y mandar *chunks pequeños* a los *más lentos*. (tamaño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

PNT: versión paralela dinámica. (cont.)



PNT: Versión paralela dinámica. Seudocódigo.



[Descargar: ./example/primes4sc.cpp]

```
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.



[Descargar: ./example/primes4.cpp]

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

```
22. MPI_Comm_size(MPI_COMM_WORLD,&size);
23.
24. assert(size>1);
25. int start_comp = MPE_Log_get_event_number();
26. int end_comp = MPE_Log_get_event_number();
27. int start_comm = MPE_Log_get_event_number();
28. int end_comm = MPE_Log_get_event_number();
29.
30. int chunk=200000, N=20000000;
31. MPI_Status status;
32. int stat[2]; // checked,primes
33.
34. #define COMPUTE 0
35. #define STOP 1
36.
37. if (!myrank) {
38.     MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
39.     MPE_Describe_state(start_comm,end_comm,"comm","red:white");
40.     int first=0, checked=0, down=0, primes=0;
41.     while (1) {
42.         MPI_Recv(stat,2,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
43.                  MPI_COMM_WORLD,&status);
44.         int source = status.MPI_SOURCE;
45.         if (stat[0]) {
46.             checked += stat[0];
47.             primes += stat[1];
48.             printf("recvd %d primes from %d, checked %d, cum primes %d\n",
49.                   stat[1],source,checked,primes);
```

```
50.    }
51.    printf("sending [ %d, %d) to %d\n",first,first+chunk,source);
52.    MPI_Send(&first,1,MPI_INT,source,0,MPI_COMM_WORLD);
53.    if (first<N) first += chunk;
54.    else printf("shutting down %d, so far %d\n",source,++down);
55.    if (down==size-1) break;
56.  }
57. } else {
58. int start;
59. stat[0]=0; stat[1]=0;
60. MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
61. while(1) {
62.   MPE_Log_event(start_comm,0,"start-comm");
63.   MPI_Recv(&start,1,MPI_INT,0,MPI_ANY_TAG,
64.             MPI_COMM_WORLD,&status);
65.   MPE_Log_event(end_comm,0,"end-comm");
66.   if (start>=N) break;
67.   MPE_Log_event(start_comp,0,"start-comp");
68.   int last = start + chunk;
69.   if (last>N) last=N;
70.   stat[0] = last-start ;
71.   stat[1] = 0;
72.   if (start<2) start=2;
73.   for (int n=start; n<last; n++) if (is_prime(n)) stat[1]++;
74.   MPE_Log_event(end_comp,0,"end-comp");
75.   MPE_Log_event(start_comm,0,"start-comm");
76.   MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
```

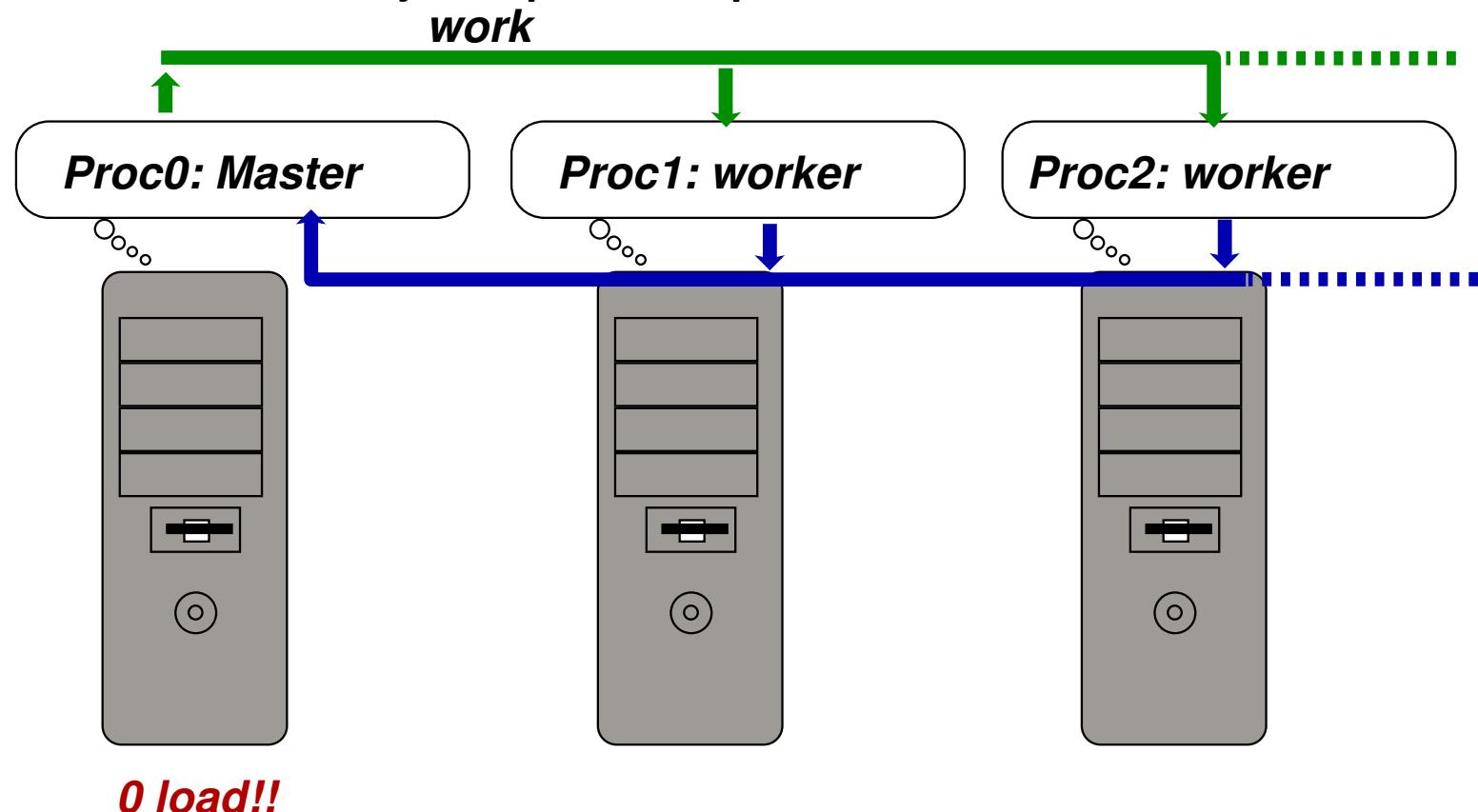
```
77.      MPE_Log_event(end_comm,0,"end-comm");
78.  }
79. }
80. MPE_Finish_log("primes");
81. MPI_Finalize();
82. }
```

PNT: 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 el código* 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.

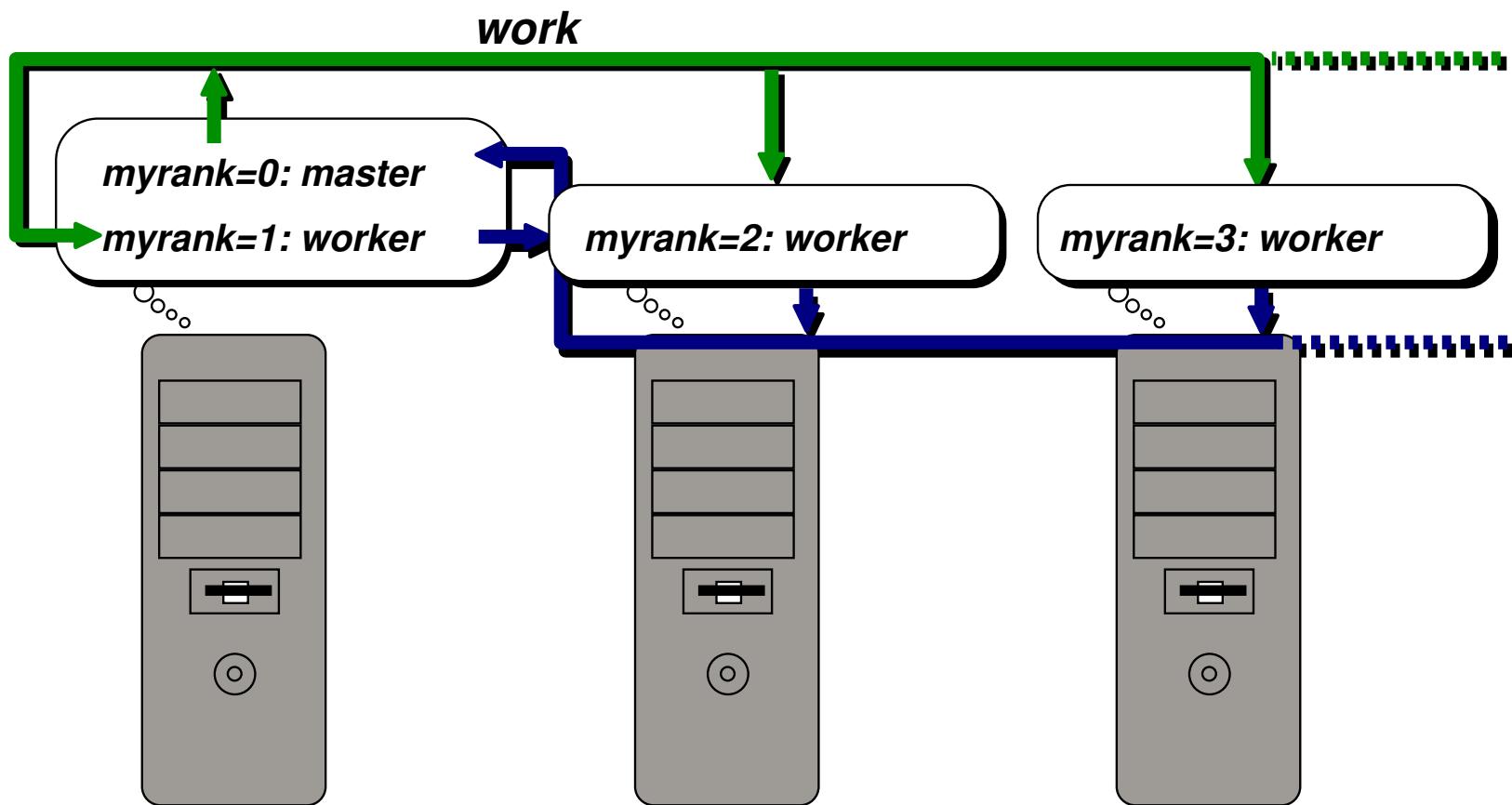
PNT: Versión paralela dinámica. (cont.)

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



PNT: Versión paralela dinámica. (cont.)

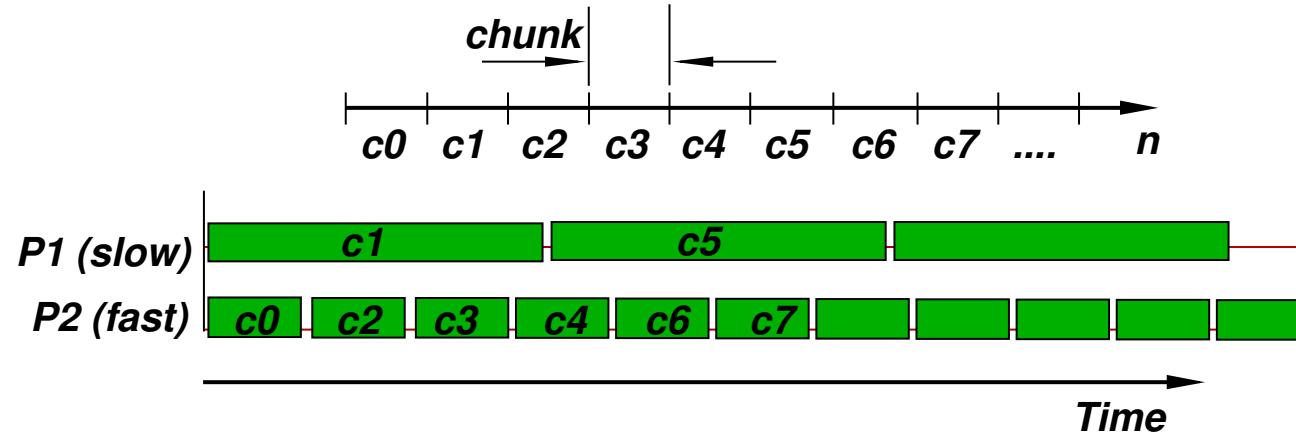
*En el procesador 0 hay dos procesos: myrank=0 (master) y myrank=1 (worker). Puede haber un cierto **retardo** en la respuesta del master. **No se pierde ningún procesador.***



PNT: Versión paralela dinámica. (cont.)

- **Tiempos calculando $\pi(5 \times 10^7) = 3001136$.**
- **16 nodos (rápidos P4HT, 3.0GHZ, DDR-RAM, 400MHz, dual channel), (lentos P4HT, 2.8GHZ, DDR-RAM, 400MHz)).**
- **Secuencial ($np = 1$) en nodo 10 (rápido): 285 sec.**
- **Secuencial ($np = 1$) en nodo 24 (lento): 338 sec.**
- **En $np = 12$ (desbalanceado, bloqueante) 69 sec.**
- **Excluyendo el nodo cargado (balanceado, bloqueante, $np = 11$): 33sec.**
- **En ($np = 14$) (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 $c1$. Similarmente $c6$ y $c7$ son reportados después de cargar $c5$...

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



[Descargar: ./example/primes5sc.cpp]

```
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.)



[Descargar: [./example/p5report.cpp](#)]

```
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.)



[Descargar: ./example/primes5.cpp]

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

```
23. int checked,primes,start;
24. bool operator<(const chunk_info& c) const {
25.     return start<c.start;
26. }
27. };
28.
29. int main(int argc, char **argv) {
30.     MPI_Init(&argc,&argv);
31.     MPE_Init_log();
32.
33.     int myrank, size;
34.     MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
35.     MPI_Comm_size(MPI_COMM_WORLD,&size);
36.
37.     assert(size>1);
38.     int start_comp = MPE_Log_get_event_number();
39.     int end_comp = MPE_Log_get_event_number();
40.     int start_comm = MPE_Log_get_event_number();
41.     int end_comm = MPE_Log_get_event_number();
42.
43.     int chunk = 20000, N = 200000;
44.     char *cvalue = NULL;
45.     int index;
46.     int c;
47.     opterr = 0;
48.
49.     while ((c = getopt (argc, argv, "N:c:")) != -1)
50.         switch (c) {
```

```
51.    case 'c':
52.        sscanf(optarg, "%d", &chunk); break;
53.    case 'N':
54.        sscanf(optarg, "%d", &N); break;
55.    case '?':
56.        if (isprint (optopt))
57.            fprintf (stderr, "Unknown option '-%c'.\n", optopt);
58.        else
59.            fprintf (stderr,
60.                     "Unknown option character '\\x%x'.\n",
61.                     optopt);
62.        return 1;
63.    default:
64.        abort ();
65.    };
66.
67.    if (!myrank)
68.        printf ("chunk %d, N%d\n", chunk, N);
69.
70.    MPI_Status status;
71.    int stat[2]; // checked_primes
72.    set<chunk_info> received;
73.    vector<int> start_sent(size,-1);
74.
75. #define COMPUTE 0
76. #define STOP 1
77.
78.    if (!myrank) {
79.        MPE_Describe_state(start_comp,end_comp,"comp","green:gray");
```

```
80.    MPE_Describe_state(start_comm,end_comm,"comm","red:white");
81.    int first=0, checked=0,
82.        down=0, primes=0, first_recv = 0;
83.    while (1) {
84.        MPI_Recv(&stat,2,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
85.                  MPI_COMM_WORLD,&status);
86.        int source = status.MPI_SOURCE;
87.        if (stat[0]) {
88.            assert(start_sent[source]>=0);
89.            chunk_info cs;
90.            cs.checked = stat[0];
91.            cs.primes = stat[1];
92.            cs.start = start_sent[source];
93.            received.insert(cs);
94.            printf("recvd %d primes from %d\n",
95.                   stat[1],source,checked,primes);
96.        }
97.        while (!received.empty()) {
98.            set<chunk_info>::iterator q = received.begin();
99.            if (q->start != first_recv) break;
100.            primes += q->primes;
101.            received.erase(q);
102.            first_recv += chunk;
103.        }
104.        printf("pi( %d ) = %d, (queued %d)\n",
105.               first_recv+chunk,primes,received.size());
```

```
106.    MPI_Send(&first,1,MPI_INT,source,0,MPI_COMM_WORLD);
107.    start_sent[source] = first;
108.    if (first<N) first += chunk;
109.    else {
110.        down++;
111.        printf("shutting down %d, so far %d\n",source,down);
112.    }
113.    if (down==size-1) break;
114. }
115. set<chunk_info>::iterator q = received.begin();
116. while (q!=received.end()) {
117.     primes += q->primes;
118.     printf("pi( %d ) =%d\n",q->start+chunk,primes);
119.     q++;
120. }
121. } else {
122.     int start;
123.     stat[0]=0; stat[1]=0;
124.     MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
125.     while(1) {
126.         MPE_Log_event(start_comm,0,"start-comm");
127.         MPI_Recv(&start,1,MPI_INT,0,MPI_ANY_TAG,
128.                  MPI_COMM_WORLD,&status);
129.         MPE_Log_event(end_comm,0,"end-comm");
130.         if (start>=N) break;
131.         MPE_Log_event(start_comp,0,"start-comp");
132.         int last = start + chunk;
```

```
133.     if (last>N) last=N;
134.     stat[0] = last-start ;
135.     stat[1] = 0;
136.     if (start<2) start=2;
137.     for (int n=start; n<last; n++) if (is_prime(n)) stat[1]++;
138.     MPE_Log_event(end_comp,0,"end-comp");
139.     MPE_Log_event(start_comm,0,"start-comm");
140.     MPI_Send(stat,2,MPI_INT,0,0,MPI_COMM_WORLD);
141.     MPE_Log_event(end_comm,0,"end-comm");
142.   }
143. }
144. MPE_Finish_log("primes");
145. MPI_Finalize();
146. }
```

Leer opciones y datos

- *Leer opciones de un archivo via NFS*



[Descargar: ./example/readnfs.cpp]

```
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*



[Descargar: ./example/bcastarg.cpp]

```
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.         };
16.     //...
17. }
```

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

Escalabilidad

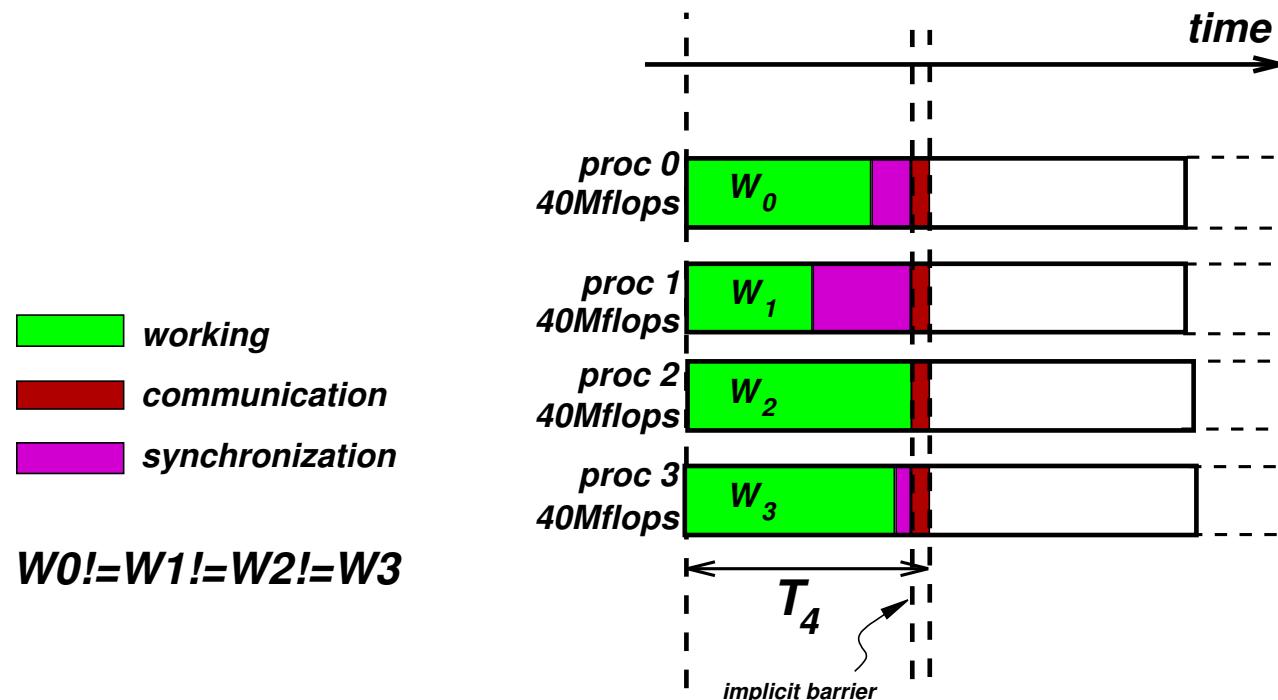
El problema de **escalabilidad** es peor aún si no es fácil de distribuir la carga, ya que en ese caso además del tiempo extra de comunicación tememos un tiempo extra de “**sincronización**”. En el caso del PNT determinar si un dado número j es primo o no varía mucho con j , para los pares es casi inmediato 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 $T_{\text{comp},i} = W_i/s$ (done W_i es la cantidad de trabajo que le fue enviada) y **se quedará esperando** hasta que aquel que recibió más trabajo termine.

Escalabilidad (cont.)

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

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

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



Escalabilidad (cont.)

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

PNT: Análisis detallado de escalabilidad

- Usamos la relación

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

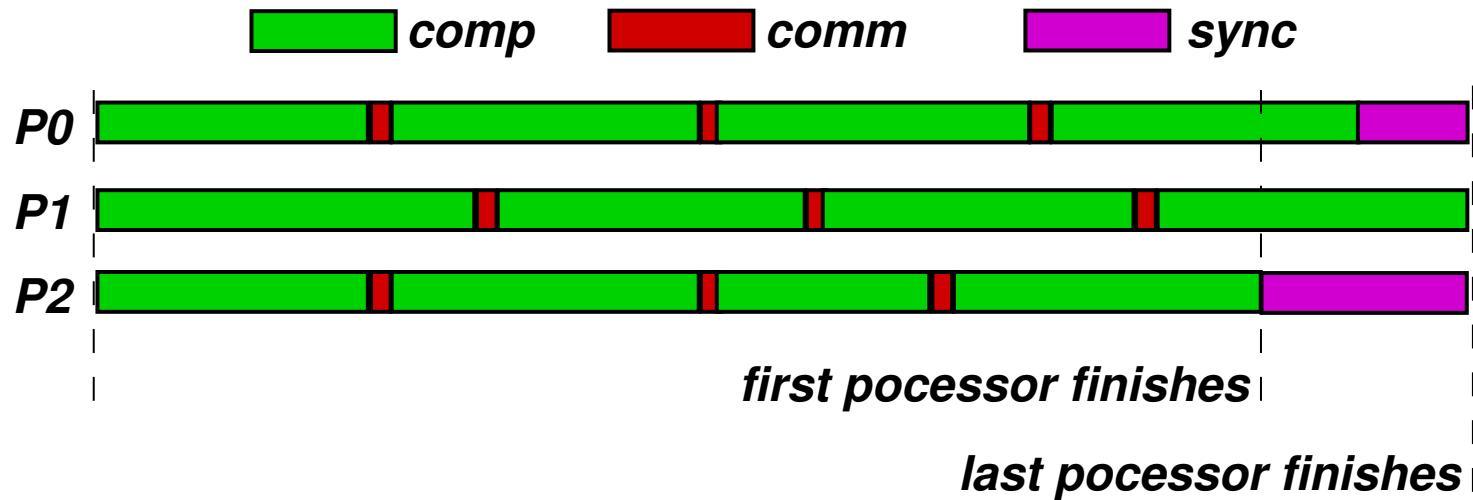
- Tiempo de cálculo para verificar si el entero j es primo: $O(j^{0.5})$
- $T_{\text{comp}}(N) = \sum_{j=1}^N c j^{0.5} \approx c \int_0^N j^{0.5} dj = 2cN^{1.5}$
- ***El tiempo de comunicación*** es mandar y recibir un entero por cada chunk, es decir

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

donde l es la latencia. N_c es la ***longitud del chunk*** que procesa cada procesador.

PNT: Análisis detallado de escalabilidad (cont.)

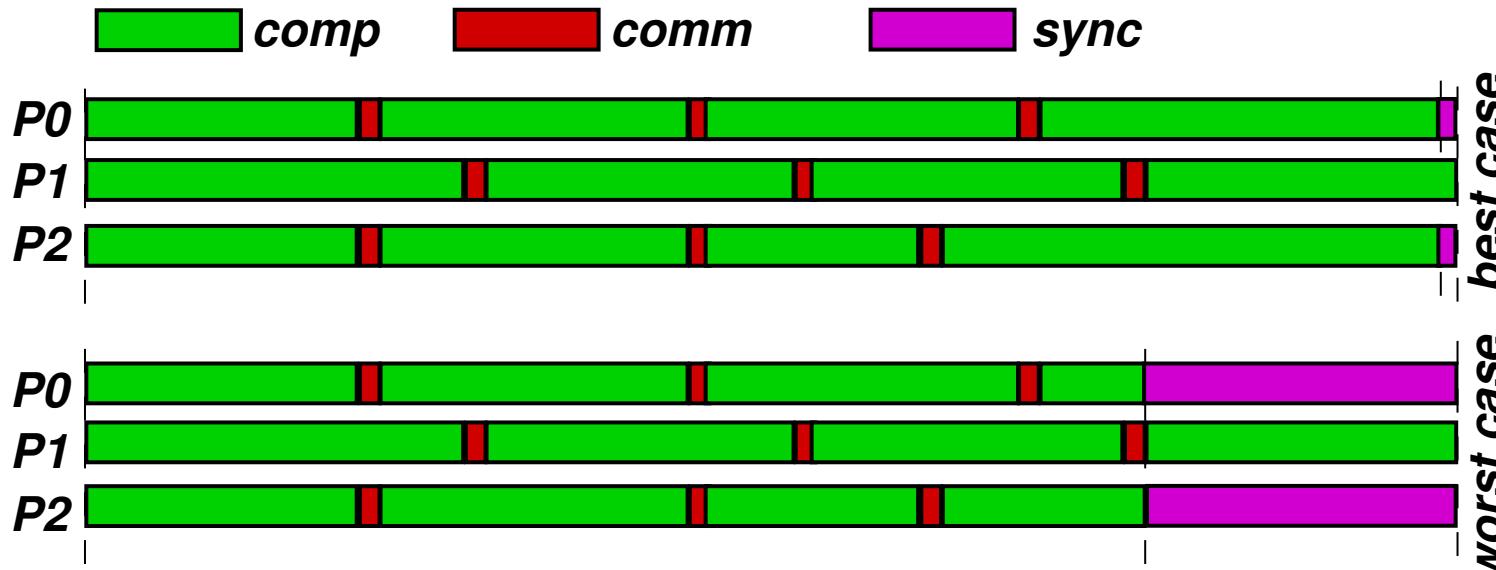
- El *tiempo de sincronización* es muy variable, casi aleatorio.



PNT: Análisis detallado de escalabilidad (cont.)

- Puede pasar que todos terminen al mismo tiempo, en cuyo caso $T_{sync} = 0$. El **peor caso** es si todos terminan de procesar casi al mismo tiempo, quedando **sólo uno de los procesadores** que falta procesar un chunk.

$$T_{sync} = (n - 1)(\text{tiempo de proc de un chunk})$$



PNT: Análisis detallado de escalabilidad (cont.)

- En *promedio*

$$T_{\text{sync}} = (n/2)(\text{tiempo de proc de un chunk})$$

- En nuestro caso

$$(\text{tiempo de proc de un chunk}) \leq N_c c N^{0.5}$$

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

PNT: Análisis detallado de escalabilidad (cont.)

$$\begin{aligned}\eta &= \frac{2cN^{1.5}}{2cN^{1.5} + (N/N_c)2l + (n/2)N_cN^{0.5}} \\ &= \left(1 + (N^{-0.5}l/c)/N_c + (n/4cN)N_c\right)^{-1} \\ &= \left(1 + A/N_c + BN_c\right)^{-1}, \quad A = (N^{-0.5}l/c), \quad B = (n/4cN)\end{aligned}$$

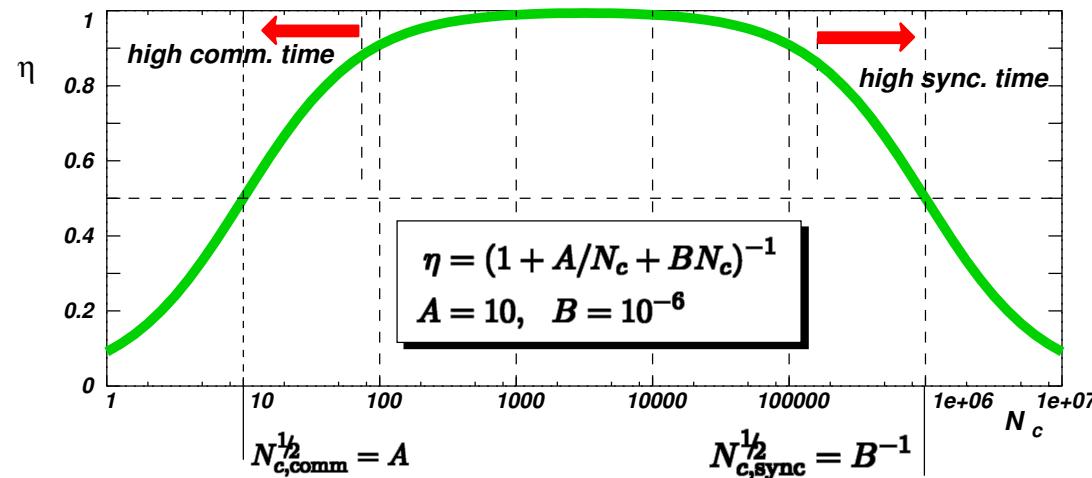
$$N_{c,\text{opt}} = \sqrt{A/B},$$

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

PNT: Análisis detallado de escalabilidad (cont.)

$$\eta = (1 + A/N_c + BN_c)^{-1}, \quad A = (N^{-0.5}l/c), \quad B = (n/4cN)$$

- Para $N_c \gg A$ la **comunicación es despreciable**.
- Para $N_c \ll B^{-1}$ la **sincronización es despreciable**.
- Si $A \ll B^{-1}$ entonces hay una “**ventana**” $[A, B^{-1}]$ donde la eficiencia se mantiene $\eta > 0.5$.
- A medida que crece N la ventana se agranda, i.e. (i.e. $A \propto N^{-0.5}$ baja y $B^{-1} \propto N$ crece).



Balance de carga

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 rápidos 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 de procesamiento** 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

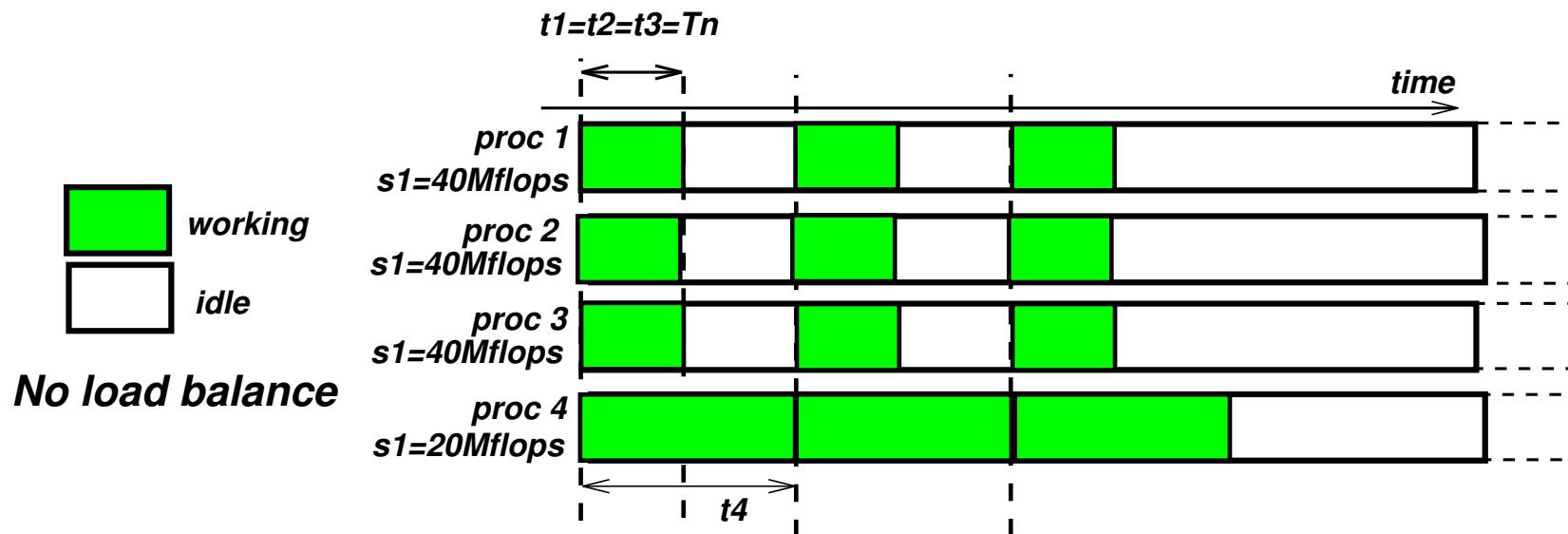
$$T_1 = \min t_i = \frac{W}{\max_i s_i}$$

- El speedup resulta ser entonces

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

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.)



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

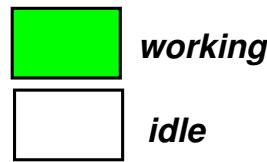
$$t_i = \frac{W_i}{s_i} = \frac{W}{\sum_j s_j} \quad (\text{independiente de } i!!)$$

- El *speedup* ahora es

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

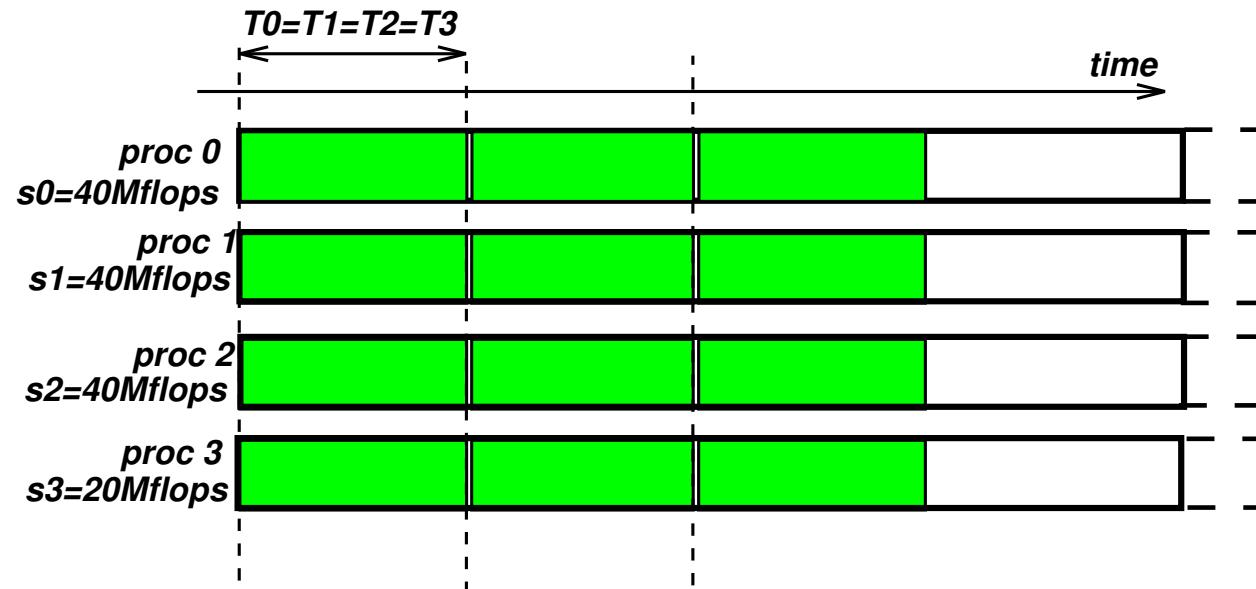
Este es el *máximo speedup teórico alcanzable en clusters heterogéneos*.

Balance de carga (cont.)



With load balance

$$W_1=W_2=W_3=2 \cdot W_4$$



Balance de carga (cont.)

Volviendo al ejemplo del cluster con 12 procs con *velocidades de procesamiento*: 8 nodos @ 4 Gflops y 4 nodos @ 2.4 Gflops, *con balance de carga* esperamos un speed-up teórico de

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

Otra forma de ver esto: Ya que la *velocidad relativa* de los nodos lento es $2.4/4 = 0.6$ con respecto a los nodos más rápidos, entonces los mas lentos pueden “*rendir*” a lo sumo como $4 \times 0.6 = 2.4$ nodos rápidos. Entonces a lo sumo podemos tener un speed-up de $8 + 2.4 = 10.4$.

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 $f(x_i), i = 0, \dots, m - 1$. Se supone que tenemos un programa

`computeF -x <x-val>` que imprime por stdout el valor de $f(x)$.



[Descargar: [./example/trivpex.cpp](#)]

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

Paralelismo trivial (cont.)



[Descargar: ./example/trivparsc.cpp]

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

```
21.     deltax = xmax/double(m);
22.     char line[100], line2[100];
23.     MPI_Send(val,...,0,...);
24.     while(1) {
25.         int j;
26.         MPI_Recv(&j,...,0,...);
27.         if (j>=m) break;
28.         sprintf(line,"computef -x %f > proc %d.output",j*deltax,myrank);
29.         sprintf(line2,"proc %d.output",myrank);
30.         FILE *fid = fopen(line2,"r");
31.         fscanf(fid,"%lf",&val);
32.         system(line)
33.         MPI_Send(val,...,0,...);
34.     }
35. }
```

GTP 3. [POIMC] Resolver ec Poisson por Montecarlo

Podemos resolver la ecuación de Poisson

$$\begin{aligned} -\Delta\phi &= 1, \quad \text{en } \Omega = [0, 1] \times [0, 1], \\ \phi &= 0, \quad \text{en } \partial\Omega, \end{aligned} \tag{2}$$

usando una técnica Montecarlo. Dividimos el dominio Ω en $N \times N$ celdas. Pongamos una ficha en la celda j, k , tiramos un número entero aleatorio con igual probabilidad en el rango $[0, 4)$ y dependiendo de ese número movemos la ficha una posición en alguna de las direcciones S, E, N, W , es decir a las celdas $j \pm 1, k \pm 1$. Si la ficha se sale del tablero, es decir $j, k < 0$ or $j, k \geq N$ la volvemos a reinyectar en una posición aleatoria en cualquier casillero del tablero.

En estas condiciones, podemos ver fácilmente que la probabilidad P_{jk} de que la ficha esté en la celda j, k satisface

$$P_{jk} = 0.25(P_{j+1,k} + P_{j-1,k} + P_{j,k+1} + P_{j,k-1}) + c, \tag{3}$$

ya que para llegar a la celda jk la ficha debe venir de algunas de las vecinas y la probabilidad de que caiga en la jk es 0.25 en cada caso. La constante c

indica la probabilidad de que la ficha sea reinyectada. Reordenando la ecuación queda

$$\frac{P_{j+1,k} + P_{j-1,k} + P_{j,k+1} + P_{j,k-1} - 4P_{jk}}{h^2} = -c' = -\frac{c}{h^2}, \quad (4)$$

Donde vemos que el miembro izquierdo es la aproximación estándar centrada de segundo orden para el operador de Laplace. Por lo tanto

$$P \rightarrow c'\phi, \quad \text{for } N \rightarrow \infty. \quad (5)$$

La constante queda indeterminada por ahora.

El algoritmo funcionaría según el siguiente seudo-código

```

1. P = zeros(N,N);
2. int count=0, j=rand()%N, k=rand()%N;
3. while (1) {
4.   int r = rand()%4;
5.   if (r==0) j++;
6.   else if (r==1) j--;
7.   else if (r==2) k++;
8.   else k--;
9.   if (j<0 || k<0 || j>=N || k>=N) {
10.     j=rand()%N, k=rand()%N;
11.   }
12.   P(j,k)++;

```

```

13.   count++;
14. }
```

Cuando **count** $\rightarrow \infty$ tenemos que $P_{jk}/\text{count} \rightarrow h^2\phi(\mathbf{x}_{jk})$, donde

$$\mathbf{x}_{jk} = ((j + 1/2)h, (k + 1/2)h) \quad (6)$$

es el centro de la celda j, k .

Implementación MPI:

- Cada proc debe inicializar su generador de randoms en forma independiente: **srand(12345678+myrank)**.
- Cada proc calcula un cierto número de movidas **chunk=100000** y calcula (acumulando) sus **P(j,k)**.
- Después de hacer una reducción de los **P(j,k)** al master se reporta el máximo dividido la media

$$z = \frac{\max \phi}{\bar{\phi}} = \frac{\max P_{jk}}{\bar{P}_{jk}} \approx 2.096254 \quad (7)$$

- Detenerse cuando la variación del valor de z no cambia en un cierto valor (e.g. 1×10^{-3}) en dos chunks sucesivos.
- Conviene almacenar los **Pjk** y **count** en forma de **doubles**.

Consigna:

- Escribir el programa como se indicó, de tal forma que va realizando chunks de movidas, y acumula las estadísticas en el master.
- El master va chequeando la convergencia del z_j , el resultado en la iteración j del lazo sobre chunks. Detener la iteración cuando $|z_j - z_{j-1}| < \text{tol}$ (eg 1×10^{-3}).
- Para una dada malla de $N \times N$ desmostrar como converge el valor de z al valor final (no tiene porqué ser igual al teórico).
- Haciendo crecer el N y para cada N tomando suficientes puntos demostrar que el z converge al valor teórico cuando se incrementa suficientemente N .
- Discutir el speedup y eficiencia de la paralelización.

El problema del agente viajero (TSP)

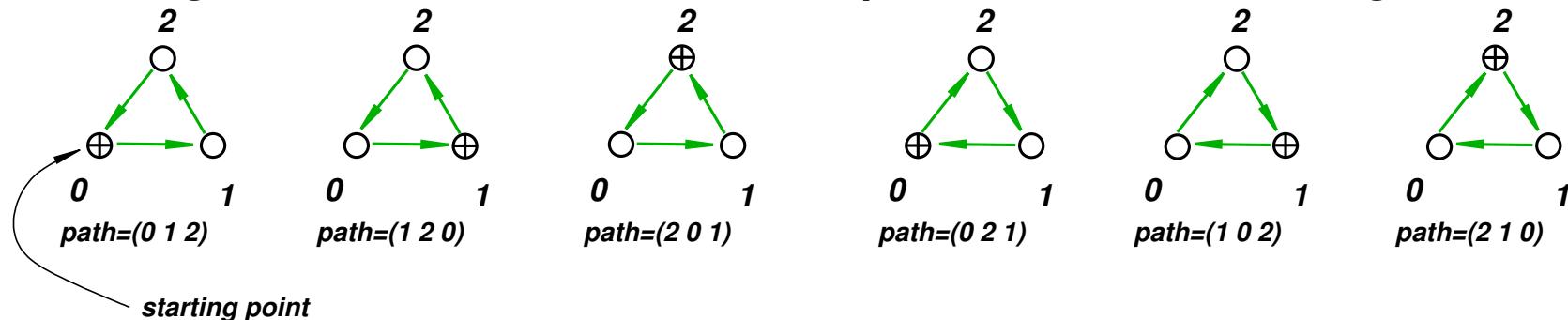
El problema del agente viajero (TSP)

El TSP (“*Traveling Salesman Problem*”, o “*Problema del Agente Viajero*”) consiste en determinar el camino (“*path*”) de **longitud mínima** que recorre todas las vértices de un **grafo** de n vértices, pasando **una sola vez por cada vértice** y volviendo al punto de partida. 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 . Asumimos que la tabla de distancias es **simétrica** ($d[i][j] = d[j][i]$) y $d[i][i] = 0$.

(Learn more TSP@WikiP: <http://goo.gl/gxSvHl>).

El problema del agente viajero (TSP) (cont.)

Consideremos por ejemplo el caso de $N_v = 3$ vértices (numeradas de 0 a 2). Todos los **recorridos posibles** se pueden representar con las permutaciones de N_v objetos. Entonces hay 6 caminos distintos mostrados en la figura. Al pie de cada recorrido se encuentra la permutación correspondiente. Como vemos, en realidad los tres caminos de la derecha son **equivalentes** ya que difieren en el punto de partida (marcado con una cruz), y lo mismo vale para los de la izquierda. A su vez, los de la derecha son equivalentes a los de la izquierda ya que sólo difieren en el **sentido** en que se recorre las vértices. Como el grafo es no orientado se deduce que los recorridos serán iguales.



El problema del agente viajero (TSP) (cont.)

Podemos **generar todos los recorridos posibles** de la siguiente manera. Tomemos por ejemplo los recorridos posibles de dos vértices, que son $(0, 1)$ y $(1, 0)$. Los recorridos para grafos de 3 vértices pueden obtenerse a partir de los de 2 *insertando el vértice 2 en cada una de las 3 posiciones posibles. De manera que por cada recorrido de dos vértices tenemos* 3 recorridos de 3 vértices, es decir que el número de recorridos N_{path} para 3 vértices es

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

En general tendremos,

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

El problema del agente viajero (TSP) (cont.)

En realidad, por cada camino se pueden generar N_v caminos equivalentes **cambiando el punto de partida**, de manera que el número de caminos diferentes es

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

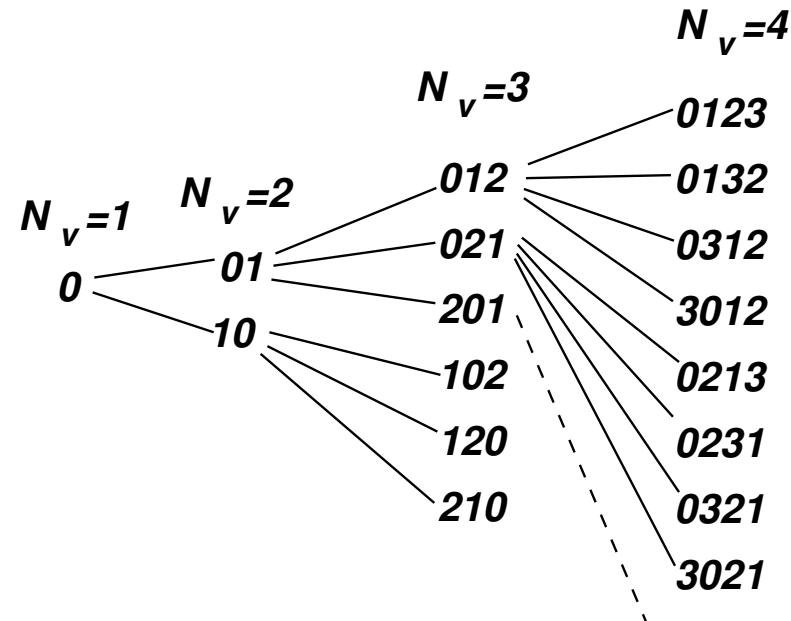
Además, si tenemos en cuenta que por cada camino podemos generar otro equivalente **invirtiendo el sentido de recorrido** tenemos que el número de caminos diferentes es

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

Por ejemplo para $N_v = 3$ tenemos $N_{\text{path}} = 1$ (**como ya vimos en la pág. 159**)

El problema del agente viajero (TSP) (cont.)

El *procedimiento recursivo* mencionado previamente para recorrer todos los caminos genera las secuencia siguientes:



El problema del agente viajero (TSP) (cont.)

Observando las secuencias para un N_v dado deducimos un algoritmo para **calcular el siguiente camino** a partir del previo. El algoritmo **avanza** un camino en la forma de un arreglo de enteros `int *path`, y retorna 1 si efectivamente pudo avanzar el camino y 0 si es **el último camino posible**.

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

El problema del agente viajero (TSP) (cont.)

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

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

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

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

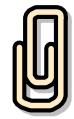
$$p_4 = (1, 2, 0)$$

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

- Por ej., para $N_v = 3$ se generan los caminos en el orden que figura al lado.
- Al aplicar el algoritmo a p_0 vemos que simplemente podemos *intercambiar el 2 con su predecessor*.
- Para p_2 no podemos avanzar el 2, ya que está en la primera posición. Lo pasamos al final, quedando $(0, 1, 2)$ y tratamos de avanzar el 1, quedando $(1, 0, 2)$.
- Al aplicarlo a p_5 vemos que *no podemos avanzar ningun vértice*, de manera que el algoritmo *retorna 0*.

El problema del agente viajero (TSP) (cont.)

Código completo para `next_path()`



[Descargar: [./example/tsp1c.cpp](#)]

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

El problema del agente viajero (TSP) (cont.)

Para visitar todos los caminos posibles

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

Por ejemplo, si definimos una función

`double dist(int *path,int Nv,double *d);` que *calcula la distancia total para un camino dado*, entonces podemos calcular la *distancia mínima* con el siguiente código

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

El problema del agente viajero (TSP) (cont.)

- **Possible implementación dinámica en paralelo:** Generar “chunks” de N_c caminos y **mandarle a los esclavos para que procesen**. Por ejemplo, si $N_v = 4$ entonces hay 24 caminos posibles. Si hacemos $N_c = 5$ entonces el master va generando chunks y enviando a los esclavos,

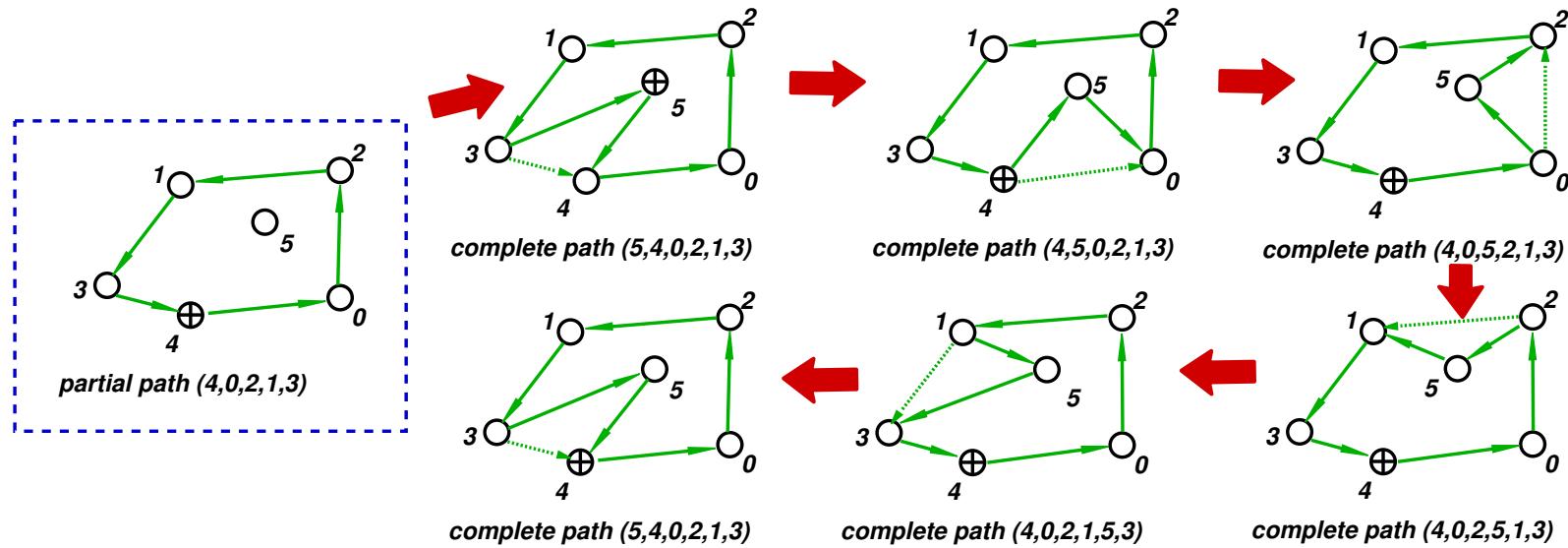
- ▷ chunk0 = {(0,1,2,3), (0,1,3,2), (0,3,1,2), (3,0,1,2), (0,2,1,3)}
- ▷ chunk1 = {(0,2,3,1), (0,3,2,1), (3,0,2,1), (2,0,1,3), (2,0,3,1)}
- ▷ chunk2 = {(2,3,0,1), (3,2,0,1), ...}
- ▷ ...

No escalea bien, ya que hay que mandarle a los esclavos una cantidad de información $T_{\text{comm}} = N_c N / b = O(N_c)$, y $T_{\text{comp}} = O(N_c)$.

El problema del agente viajero (TSP) (cont.)

Un *camino parcial* de longitud N_p es uno de los posibles caminos para los primeros N_p vértices. Los caminos completos de longitud $N_p + 1$ se pueden obtener de los de longitud N_p insertando el vértice N_p en alguna posición dentro del camino.

Caminos totales de 6 vértices generados a partir de un camino parcial de 5 vértices.



El problema del agente viajero (TSP) (cont.)

- **Possible implementación dinámica en paralelo (mejorada):** Mandar chunks que corresponden a todos los caminos totales derivados de un cierto “camino parcial” de longitud N_p . Por ejemplo, si $N_v = 4$ entonces **mandamos el camino parcial** $(0, 1, 2)$, el esclavo deducirá que debe evaluar todos los caminos totales derivados a saber
 - ▷ camino parcial $(0,1,2)$, chunk = $\{(0,1,2,3), (0,1,3,2), (0,3,1,2), (3,0,1,2)\}$
 - ▷ camino parcial $(0,2,1)$, chunk = $\{(0,2,1,3), (0,2,3,1), (0,3,2,1), (3,0,2,1)\}$
 - ▷ camino parcial $(2,0,1)$, chunk = $\{(2,0,1,3), (2,0,3,1), (2,3,0,1), (3,2,0,1)\}$
 - ▷ ...

Cada esclavo se encarga de generar todos los “caminos totales” derivados de ese camino parcial. El camino parcial actúa como un “chunk” nada más que la comunicación es $T_{\text{comm}} = O(1)$, y $T_{\text{comp}} = O(N_c)$, donde N_c es ahora el número de caminos totales derivados del camino parcial.

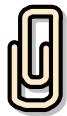
El problema del agente viajero (TSP) (cont.)

Si tomamos caminos parciales de N_p vértices entonces tenemos $N_p!$ caminos parciales. Como hay $N_v!$ caminos totales, se deduce que hay $N_c = N_v! / N_p!$ caminos totales por cada camino parcial. Por lo tanto para N_p pequeño tendremos **chunks grandes (muchísima sincronización)** y para N_p cercano a N_v tendremos **chunks pequeños (muchísima comunicación)**.

Por ejemplo si $N_v = 10$ ($10! = 3,628,800$ caminos totales) y $N_p = 8$ tendremos chunks de $N_c = 10! / 8! = 90$ caminos totales, mientras que si tomamos $N_p = 3$ entonces el tamaño del chunk es $N_c = 10! / 3! = 604,800$.

El problema del agente viajero (TSP) (cont.)

El algoritmo para generar los caminos totales derivados de un camino parcial es idéntico al usado para generar los caminos totales. Sólo basta con que en `path[]` esté el *camino parcial al principio* y después completado hacia la derecha con los vértices faltantes y aplicar el algoritmo, tratando de mover todos los vértices hacia la izquierda desde $N - 1$ hasta N_p .



[Descargar: ./example/tsp1.cpp]

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

```
14.    }
15.    return 0;
16. }
```

El problema del agente viajero (TSP) (cont.)

Para generar los caminos parciales basta con llamar a `next_path(...)` con $N_v = N_p$, ya que en ese caso la función sólo *mueve las primeras N_p posiciones*. El algoritmo siguiente recorre todos los caminos

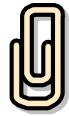


[Descargar: [./example/tsp1d.cpp](#)]

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

El problema del agente viajero (TSP) (cont.)

Por ejemplo el siguiente código *imprime todos los caminos parciales* y los caminos totales derivados.

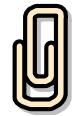


[Descargar: ./example/tsp1d.cpp]

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

El problema del agente viajero (TSP) (cont.)

Seudocódigo (master code):



[Descargar: ./example/tspsc.cpp]

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

El problema del agente viajero (TSP) (cont.)

Seudocódigo (slave code):



[Descargar: ./example/tspsc.cpp]

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

GTP 4. [TSP] Traveling Salesman Problem

Comunicación punto a punto y balance dinámico de carga.

1. Escribir un *programa secuencial* que encuentra el camino de longitud mínima para el “*Problema del Agente Viajero*” (TSP) por *búsqueda exhaustiva* intentando todos los posibles caminos.
2. Implementar las siguientes optimizaciones:
 - Podemos asumir que los caminos empiezan de una ciudad en particular (*invariancia ante rotación de las ciudades*). *Hint:* Avanzar solamente las primeras $N_v - 1$ ciudades. De esta forma la última ciudad queda siempre en la misma posición.
 - Si un camino parcial tiene *longitud mayor que el mínimo actual*, entonces no hace falta evaluar los caminos derivados de ese camino parcial. (Asumir que el grafo es “*convexo*”, es decir que $d(i, j) \leq d(i, k) + d(k, j)$, $\forall k$. Esto ocurre naturalmente cuando el grafo proviene de tomar las distancias entre puntos en alguna norma en \mathbb{R}^n).
 - Dado *un camino y su inverso*, sólo es necesario chequear uno de ellos, ya que el otro tiene la misma longitud. *Hint:* Ver más abajo.

3. Escribir una versión paralela con **distribución fija** del trabajo entre los procesadores (**balanceo estático**). Simular **desbalance** enviando varios procesos a un mismo procesador (**oversubscription**). Notar que el “trabajo” a ser enviado a los procesadores puede ser enviado en forma de un camino parcial.
4. Escribir una versión **paralela con distribución dinámica** de la carga (“compute-on-demand”).
5. Hacer un estudio de la **escalabilidad** del problema. ¿Cuál es el valor más apropiado de la longitud del camino parcial a ser enviado a los esclavos?

Paridad de los caminos

Dado un camino y su *inverso*, sólo es necesario chequear uno de ellos, ya que *el otro tiene la misma longitud*. Para esto es necesario definir una función *paridad* $\text{prty}(p)$, donde p es un camino y tal que si p' es el inverso de p , entonces $\text{prty}(p') = -\text{prty}(p)$. Una posibilidad es buscar una ciudad en particular, por ejemplo la 0 y ver si la ciudad siguiente en el camino es mayor (retorna +1) o menor (retorna -1) que la anterior a 0. Por ejemplo para el camino (503241) retorna -1 (porque $3 < 5$) y para el inverso que es (514230) retornará +1 (ya que $5 > 3$). Ahora bien, esto requiere buscar una ciudad dada, con lo cual *el costo puede ser comparable a calcular la longitud del camino*. Pero combinándolo con la optimización previa de dejar una ciudad fija, sabemos que la ciudad en la posición $N_v - 1$ es siempre la ciudad $N_v - 1$, de manera que podemos comparar la siguiente (es decir la posición 0) con la anterior (la posición $N_v - 2$).

1. `prty = (path[0]<path[Nv-2] ? 1 : -1);`

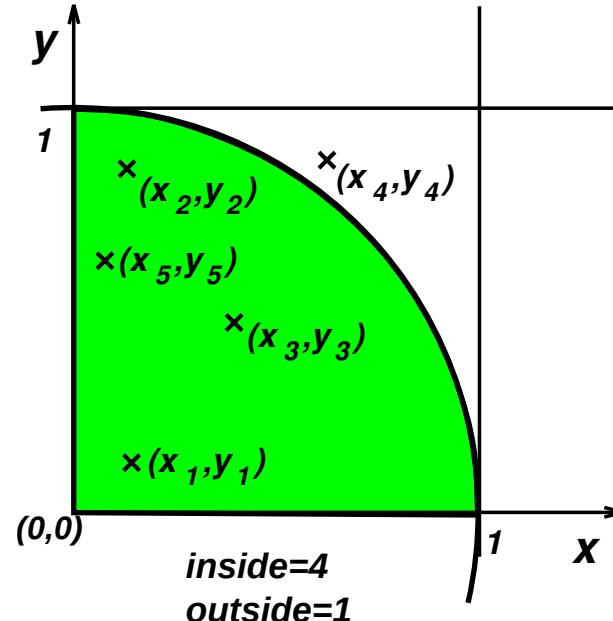
Cálculo de Pi por Montecarlo

Cálculo de Pi por Montecarlo

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

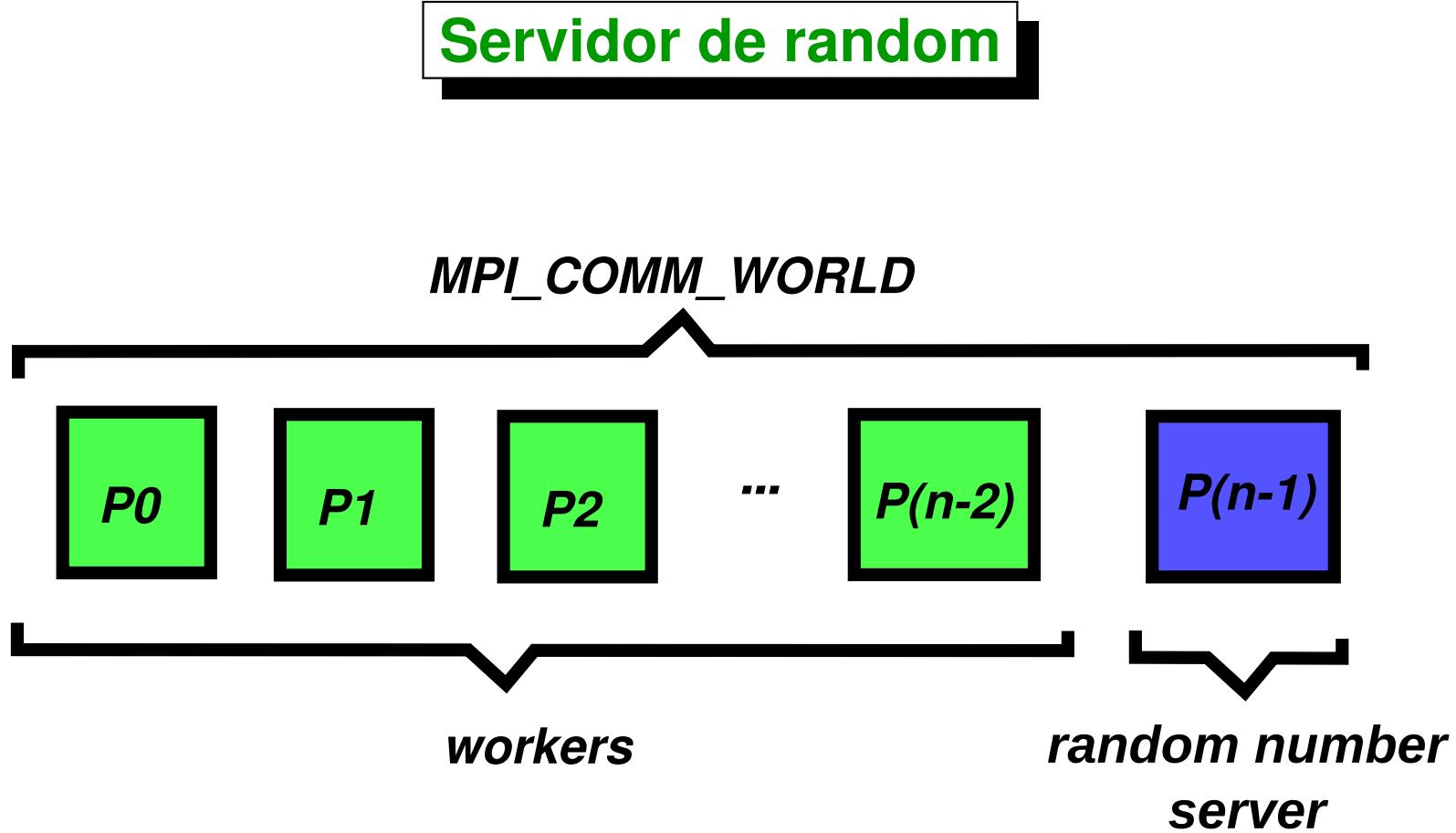
$$\pi = 4 \frac{(\# \text{adentro})}{(\# \text{total})}$$

En general los métodos de **Montecarlo** son **fácilmente paralelizables**, son **no determinísticos**, y 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 ejecución). 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.



```
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. }
```

Crear un nuevo comunicator

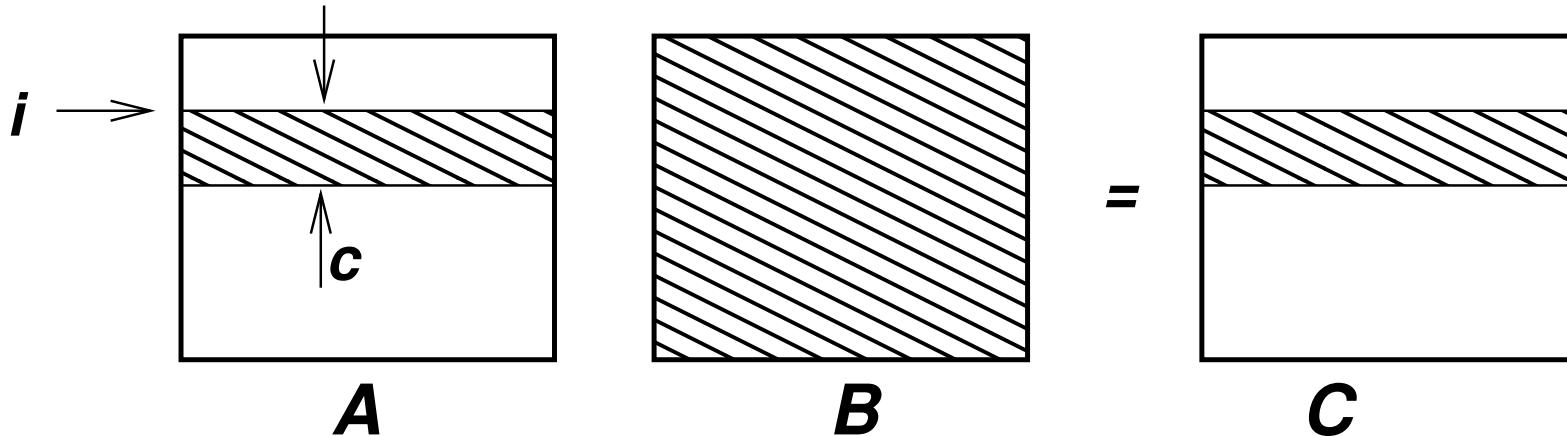


[Descargar: ./example/newcom.cpp]

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

Ejemplo: producto de matrices en paralelo

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

- Permite construir una matriz de dos índices a partir de un contenedor propio o de uno exterior.



[Descargar: ./example/mat2.cpp]

```
1. // Internal storage is released in destructor
2. Mat A(100,100);
3. // usar 'A' ...
4.
5. // Internal storage must be released by the user
6. double *b = new double[10000];
7. Mat B(100,100,b);
8. // use 'B' ...
9. delete[] b;
```

Librería simple de matrices (cont.)

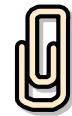
- Sobrecarga el operador `()` de manera que se puede usar tanto en el

miembro derecho como en el izquierdo.  [Descargar:]

`./example/mat1.cpp]`

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

- Como es usual en C los arreglos multidimensionales son almacenados por fila. En la clase `Mat` se puede acceder al área de almacenamiento (ya sea interno o externo), como un arreglo de C estándar. También se puede acceder a las filas individuales o conjuntos de filas contiguas.



[Descargar: `./example/matex.cpp`]

1. `Mat A(100,100);`
2. `double *a, *a10;`
3. `// Pointer to the internal storage area`
4. `a = &A(0,0);`
5. `// Pointer to the internal storage area,`

```
6. // starting at row 10 (0 base)
7. a10 = &A(10,0);
```

Librería simple de matrices (cont.)

```
1. #include <math.h>
2. #include <limits.h>
3. #include <mpi.h>
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <sys/time.h>
7. #include <cassert>
8.
9. /// Simple matrix class
10. class Mat {
11. private:
12.     /// The store
13.     double *a;
14.     /// row and column dimensions, flag if the store
15.     /// is external or not
16.     int m,n,a_is_external;
17. public:
18.     /// Constructor from row and column dimensions
19.     /// and (eventually) the external store
20.     Mat(int m_,int n_,double *a_=NULL);
21.     /// Destructor
22.     ~Mat();
23.     /// returns a reference to an element
24.     double & operator()(int i,int j);
25.     /// returns a const reference to an element
```

```
26. const double & operator()(int i,int j) const;
27. /// product of matrices
28. void matmul(const Mat &a, const Mat &b);
29. /// prints the matrix
30. void print() const;
31. };
32.
```

Balance dinámico

Abstracción: hay un serie de tareas con datos x_1, \dots, x_N y resultados r_1, \dots, r_N y P procesadores. Suponemos N suficientemente grande, y $1 \leq \text{chunksize} \ll N$ 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.

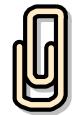


[Descargar: ./example/slave.cpp]

- ```
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

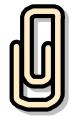
Mantiene un vector de estados para cada slave, puede ser: 0 = No empezó todavía, 1= Está trabajando, 2 = terminó.



[Descargar: ./example/server.cpp]

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

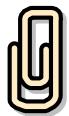
## Slave code, use local processor



[Descargar: ./example/serverul.cpp]

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

## Complete code



[Descargar: ./example/matmult2.cpp]

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

```
23. return ((double)(rand()))/((double)(RAND_MAX));
24. }
25.
26. /** Computes an 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. int main(int argc,char **argv) {
34.
35. /// Initializes MPI
36. MPI_Init(&argc,&argv);
37. /// Initializes random
38. srand(time (0));
39.
40. /// Size of problem, size of chunks (in rows)
41. int N,chunksize=100;
42. /// root procesoor, number of processor, my rank
43. int root=0, numprocs, rank;
44. /// time related quantities
45. struct timeval start, end;
46.
47. /// Status for retrieving MPI info
48. MPI_Status stat;
49. /// For non-blocking communications
50. MPI_Request request;
51. /// number of processors and my rank
```

```
52. MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
53. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
54.
55. // cursor to the next row to send
56. int row_start=0;
57. // Read input data from options
58. int c,print_mat=0,random_mat=0,use_local_processor=0,
59. slow_down=1;
60. while ((c = getopt (argc, argv, "N:c:prhls:")) != -1) {
61. switch (c) {
62. case 'h':
63. if (rank==0) {
64. printf(" usage: $ mpirun [OPTIONS] matmult2\n\n"
65. "MPI options: -np <number-of-processors>\n"
66. " -machinefile <machine-file>\n\n"
67. "Other options: -N <size-of-matrix>\n"
68. " -c <chunk-size> "
69. "# sets number of rows sent to processors\n"
70. " -p "
71. "# print input and result matrices to output\n"
72. " -r "
73. "# randomize input matrices\n");
74. }
75. exit(0);
76. case 'N':
77. sscanf(optarg, "%d", &N);
78. break;
79. case 'c':
```

```
80. sscanf(optarg, "%d", &chunkszie);
81. break;
82. case 'p':
83. print_mat=1;
84. break;
85. case 'r':
86. random_mat=1;
87. break;
88. case 'l':
89. use_local_processor=1;
90. break;
91. case 's':
92. sscanf(optarg, "%d", &slow_down);
93. if (slow_down<1) {
94. if (rank==0) printf("slow_down factor (-s option)"
95. " must be >=1. Reset to 1!\n");
96. abort();
97. }
98. break;
99. case '?':
100. if (isprint (optopt))
101. fprintf (stderr, "Unknown option '-%c'.\n", optopt);
102. else
103. fprintf (stderr,
104. "Unknown option character '\\x%x'.\n",
105. optopt);
106. return 1;
107. default:
```

```
108. abort ();
109. }
110. }
111.
112. #if 0
113. if (rank == root) {
114. printf("enter N, chunksize > ");
115. scanf("%d",&N);
116. scanf("%d",&chunksize);
117. printf("\n");
118. }
119. #endif
120.
121. // Register time for statistics
122. gettimeofday (&start,NULL);
123.
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. int ntimes=1;
136. for (int ktime=0; ktime<ntimes; ktime++) {
137. if (rank == root) {
```

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

```
167. // initialize proc_status, statistics
168. for (int proc=1; proc<numprocs; proc++) {
169. proc_status[proc] = 0;
170. statistics [proc] = 0;
171. }
172.
173. // send b to all processor
174. MPI_Bcast (&b(0,0), N*N, MPI_DOUBLE, 0,MPI_COMM_WORLD);
175.
176. while (1) {
177.
178. // non blocking receive
179. if (numprocs>1) MPI_Irecv(&bufc(0,0),chunksize*N,
180. MPI_DOUBLE,MPI_ANY_SOURCE,
181. MPI_ANY_TAG,
182. MPI_COMM_WORLD,&request);
183.
184. // While waiting for results from slaves server works ...
185. int receive_OK=0;
186. while (use_local_processor && row_start<N) {
187. // Test if some message is waiting.
188. if (numprocs>1) MPI_Test(&request,&receive_OK,&stat);
189. if(receive_OK) break;
190. // Otherwise... work
191. // Local masks
192. Mat aa(1,N,&a(row_start,0)),cc(1,N,&c(row_start,0));
193. // make product
194. for (int jj=0; jj<slow_down; jj++)
```

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

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

```
251. }
252. if (done) break;
253. }
254.
255. #if 0
256. if (slow_down>1) printf("slow_down=%d, scaled Mflops %f\n",
257. slow_down,rate*slow_down);
258. for (int procc=0; procc<numprocs; procc++) {
259. printf("%d lines processed by%d\n",
260. statistics[procc],procc);
261. }
262. #endif
263.
264. if (print_mat) {
265. printf("a: \n");
266. a.print();
267. printf("b: \n");
268. b.print();
269. printf("c: \n");
270. c.print();
271. }
272.
273. /// free memory
274. free(statistics);
275. free(proc_status);
276.
277. } else {
278. //:-<*>-:<*>-:<*>-:<*>-:<*>
```

```
280. //:-<*>:- SLAVE PART *->:-<*>
281. //:-<*>:-<*>:-<*>:-<*>:-<*>
282.
283. /// get 'b'
284. MPI_Bcast(&b(0,0), N*N, MPI_DOUBLE, 0,MPI_COMM_WORLD);
285. /// Send message zero length 'I'm ready'
286. MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);
287.
288. while (1) {
289. /// Receive task
290. MPI_Recv(&bufa(0,0),chunksize*N,MPI_DOUBLE,0,MPI_ANY_TAG,
291. MPI_COMM_WORLD,&stat);
292. int rcvlen;
293. MPI_Get_count(&stat,MPI_DOUBLE,&rcvlen);
294. /// zero length message means: no more rows to process
295. if (rcvlen==0) break;
296.
297. /// compute number of rows received
298. int nrows_sent = rcvlen/N;
299. /// index of first row sent
300. int row_start = stat.MPI_TAG;
301. /// local masks
302. Mat bufaa(nrows_sent,N,&bufa(0,0)),
303. bufcc(nrows_sent,N,&bufc(0,0));
304. /// compute product
305. for (int jj=0; jj<slow_down; jj++)
306. bufcc.matmul(bufaa,b);
307. /// send result back
```

```
308. MPI_Send(&bufcc(0,0),nrows_sent*N,MPI_DOUBLE,
309. 0,row_start,MPI_COMM_WORLD);
310.
311. }
312.
313. // Work finished. Send acknowledge (zero length message)
314. MPI_Send(&bufc(0,0),0,MPI_DOUBLE,root,0,MPI_COMM_WORLD);
315.
316. }
317. }
318. // time statistics
319. gettimeofday (&end,NULL);
320. double dtime = etime(start,end);
321. // print statistics
322. double dN = double(N), rate;
323. rate = 2*dN*dN*dN/dtime/1e9/ntimes;
324. if (rank==0)
325. printf("elaps: %f secs, rate: %f Gflops on %d procesors\n",
326. dtime,rate,numprocs);
327.
328. // Finalize MPI
329. MPI_Finalize();
330. }
```

# El juego de la vida

## El juego de la vida

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

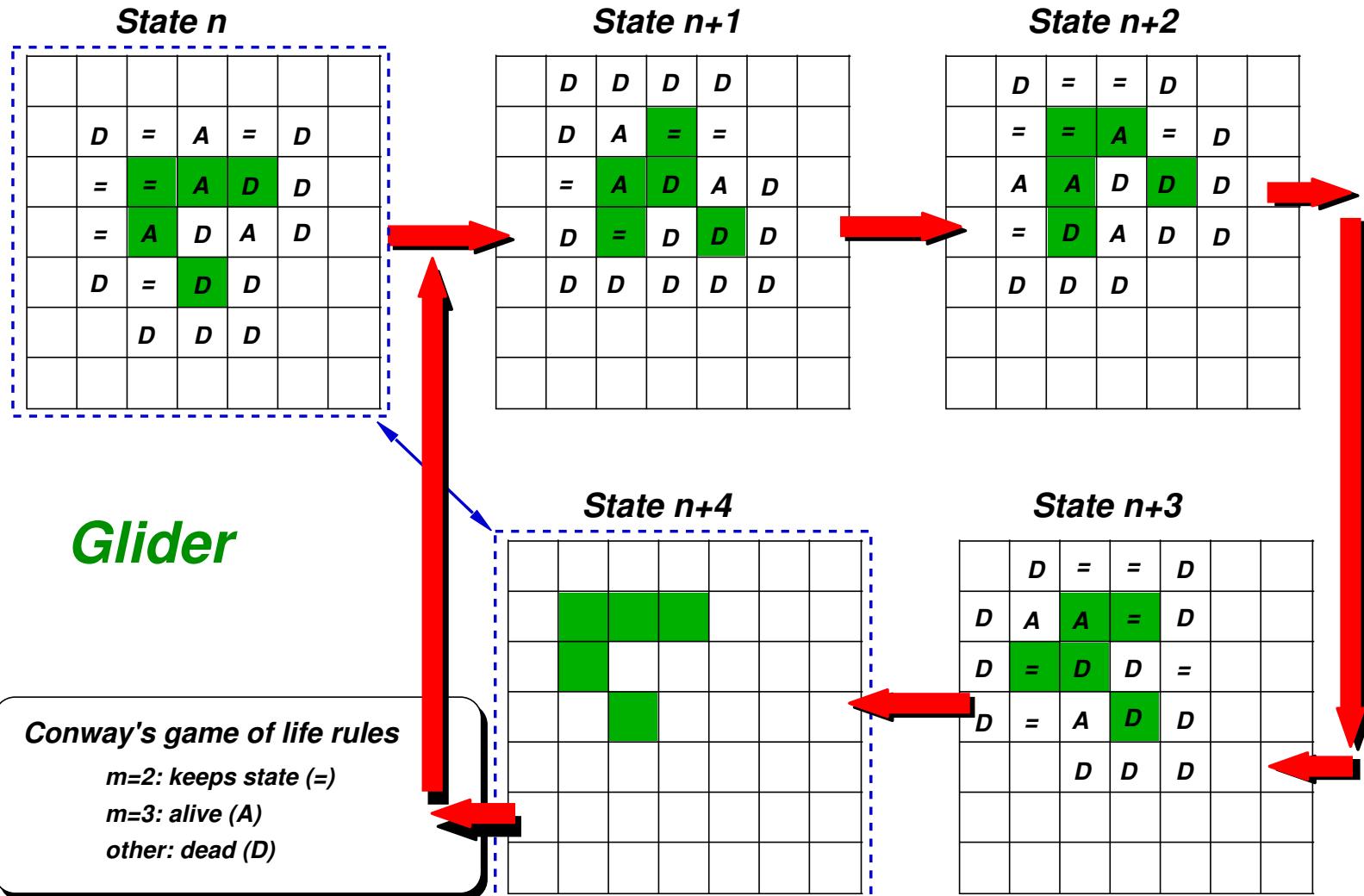
A board of  $N \times N$  cells  $c_{ij}$  for  $0 \leq i, j < N$ , that can be either in “**dead**” or “**alive**” state ( $c_{ij}^n = 0, 1$ ), is advanced from “**generation**”  $n$  to the  $n + 1$  by the following simple rules. If  $m$  is the number of neighbor cells alive in stage  $n$ , then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

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

|  |     |     |     |     |     |  |
|--|-----|-----|-----|-----|-----|--|
|  |     |     |     |     |     |  |
|  | $D$ | =   | $A$ | =   | $D$ |  |
|  | =   | =   | $A$ | $D$ | $D$ |  |
|  | =   | $A$ | $D$ | $A$ | $D$ |  |
|  | $D$ | =   | $D$ | $D$ |     |  |
|  |     | $D$ | $D$ | $D$ |     |  |
|  |     |     |     |     |     |  |



## El juego de la vida (cont.)



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

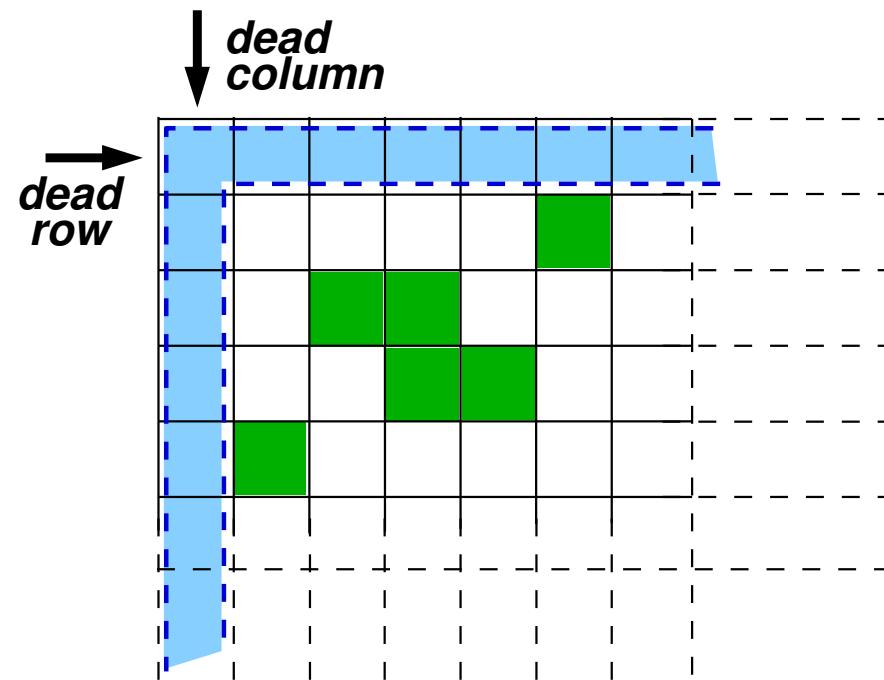
$$a_{ij}^{n+1} = \begin{cases} 1 & ; \text{ if } a_{ij}^n = 3 \\ c_{ij}^n & ; \text{ if } a_{ij}^n = 2 \\ 0 & ; \text{ otherwise.} \end{cases}$$

In some sense these rules tend to **mimic the behavior of life forms** in the sense that they die if the neighbor 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.)

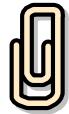
A ***layer of dead cells*** is assumed at each of the boundaries, for the evaluation of the right hand side,

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



## El juego de la vida (cont.)

### Una clase Board



[Descargar: [./example/board.cpp](#)]

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

## El juego de la vida (cont.)

- *Estado de las celdas* es representado por `char` (enteros de 8 bits). Representaciones de enteros de mayor tamaño (e.g. `int`) son **más ineficientes** porque representa más información a transmitir. Usar `vector<bool>` puede ser **más eficiente en cuanto a comunicación** pero tendría una sobrecarga por las operaciones de shifts de bits para obtener un bit dado.
- Cada fila es entonces un *arreglo* de `char` de la forma `char row[N+2]`. Los dos `char` adicionales son para las *columnas de contorno*.
- El tablero total es un *arreglo de filas*, es decir `char** board` (miembro privado dentro de la case `Board`). De manera que `board[j]` es la fila `j` y `board[j][k]` es el estado de la celda `j, k`.
- Las filas se pueden *intercambiar por punteros*, por ejemplo para intercambiar las filas `j` y `k`



[Descargar: `./example/board2.cpp`]

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

## El juego de la vida (cont.)

Las filas se *reparten* en función de pesos de los procesadores `vector<double> weights`. Primero se normaliza `weights[]` de manera que su suma sea 1. A cada procesador se le asignan `int(N*weights[myrank]+carry)` filas. `carry` es un `double` que va acumulando el *defecto por redondeo*.



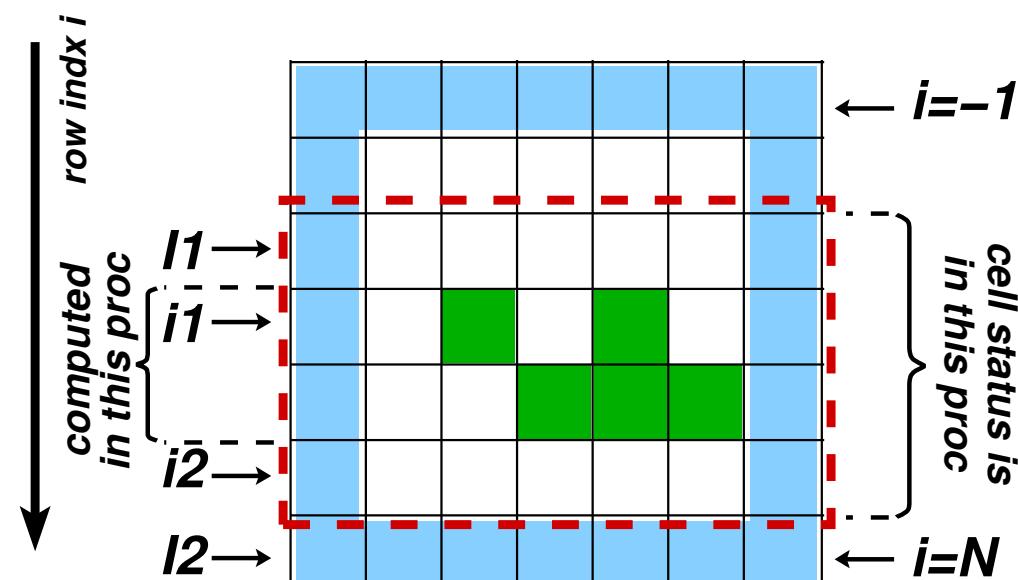
[Descargar: `./example/life.cpp`]

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

```
15. j += rows_proc[p];
16. }
17. assert(j==N);
18. assert(fabs(carry) < tol);
19.
20. I1=i1-1;
21. I2=i2+1;
```

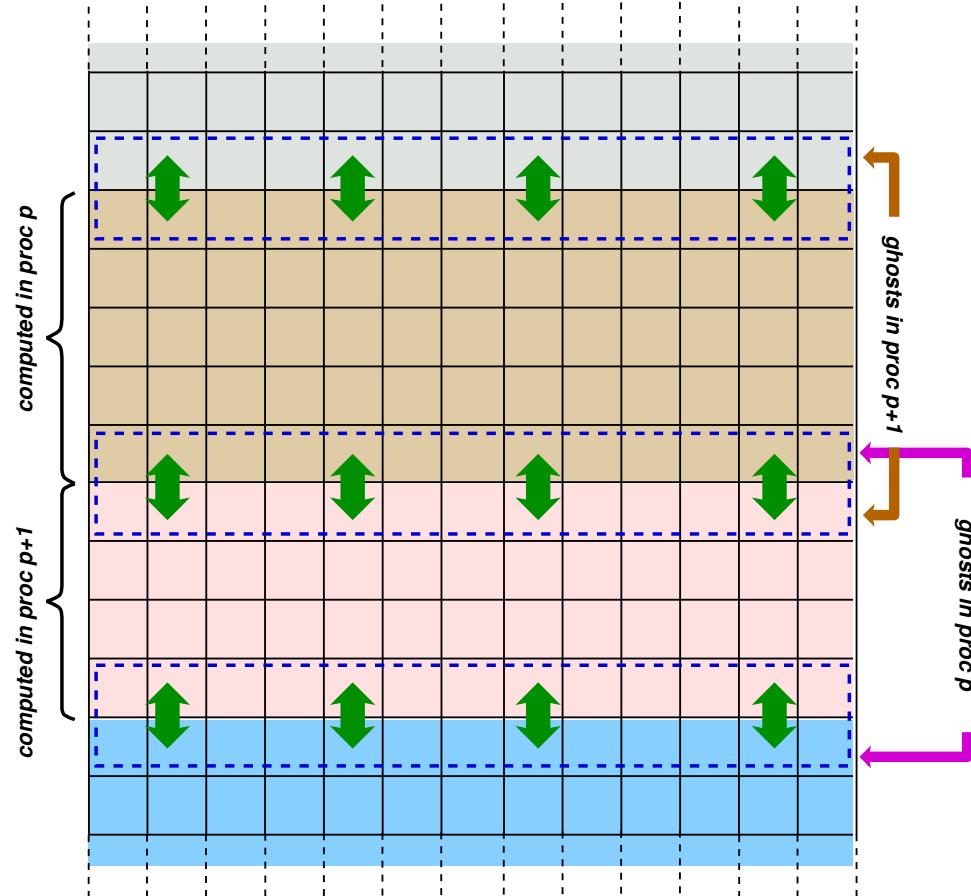
## El juego de la vida (cont.)

- En cada procesador el *rango* de filas a procesar es  $[i_1, i_2)$
- El procesador contiene la información de las celdas en el un rango  $[I_1, I_2)$  que contiene *una fila más* en cada dirección, es decir.  $I_1=i_1-1$ ,  $I_2=i_2+1$ .
- Las filas **-1** y **N** son filas *de contorno* que contienen celdas muertas.

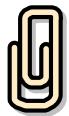


## El juego de la vida (cont.)

- `Board::advance()` calcula el nuevo estado de las celdas a partir de la *generación anterior*.
- Primero que todo hay que hacer el *scatter* de las filas *fantasma*.



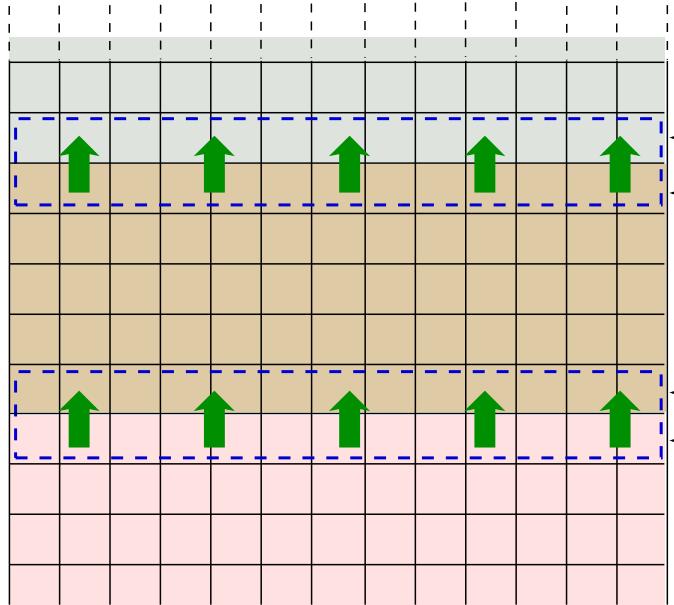
## El juego de la vida (cont.)



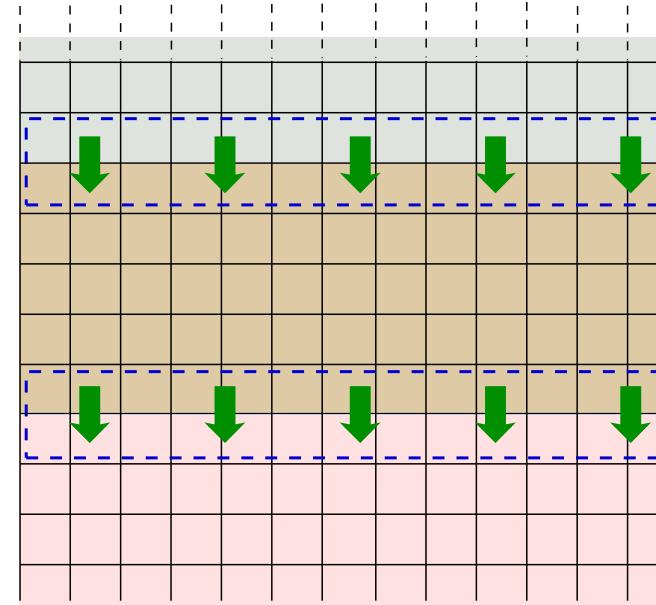
[Descargar: ./example/lifescatsc.cpp]

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

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



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

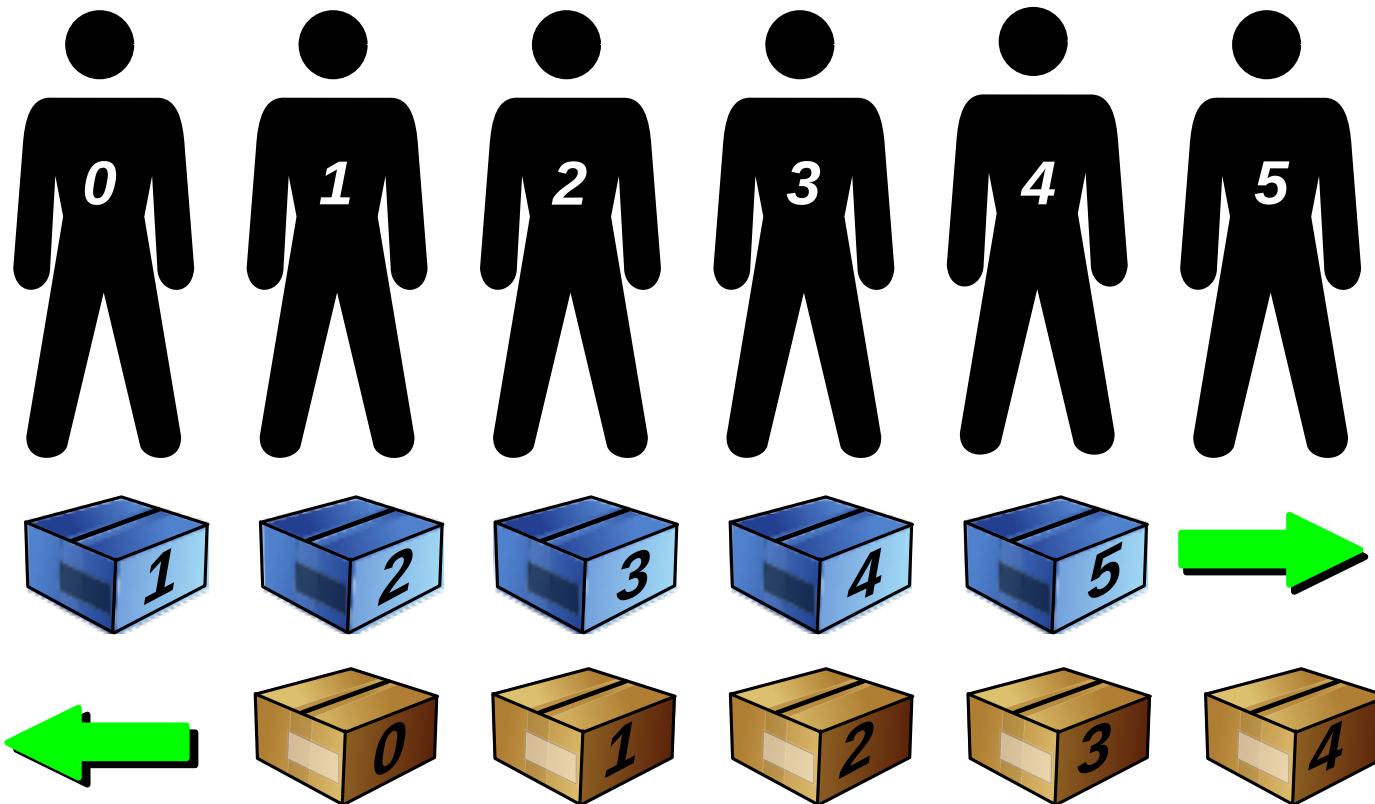


## El juego de la vida (cont.)

Este tipo de comunicación no es bloqueante pero es *muy lenta*. En la primera etapa, cuando el procesador  $p$  manda su fila  $i2 - 1$  al  $p + 1$  y después recibe la  $I1$  de  $p - 1$  se produce un retraso ya que los send y receive se “encadenan”. Notemos que el receive del proc  $p = 0$  solo se puede completar *después* de que se complete el send. De hecho, si se usaran condiciones de contorno *periódicas*, este *patrón de comunicación sería bloqueante*.

## El juego de la vida (cont.)

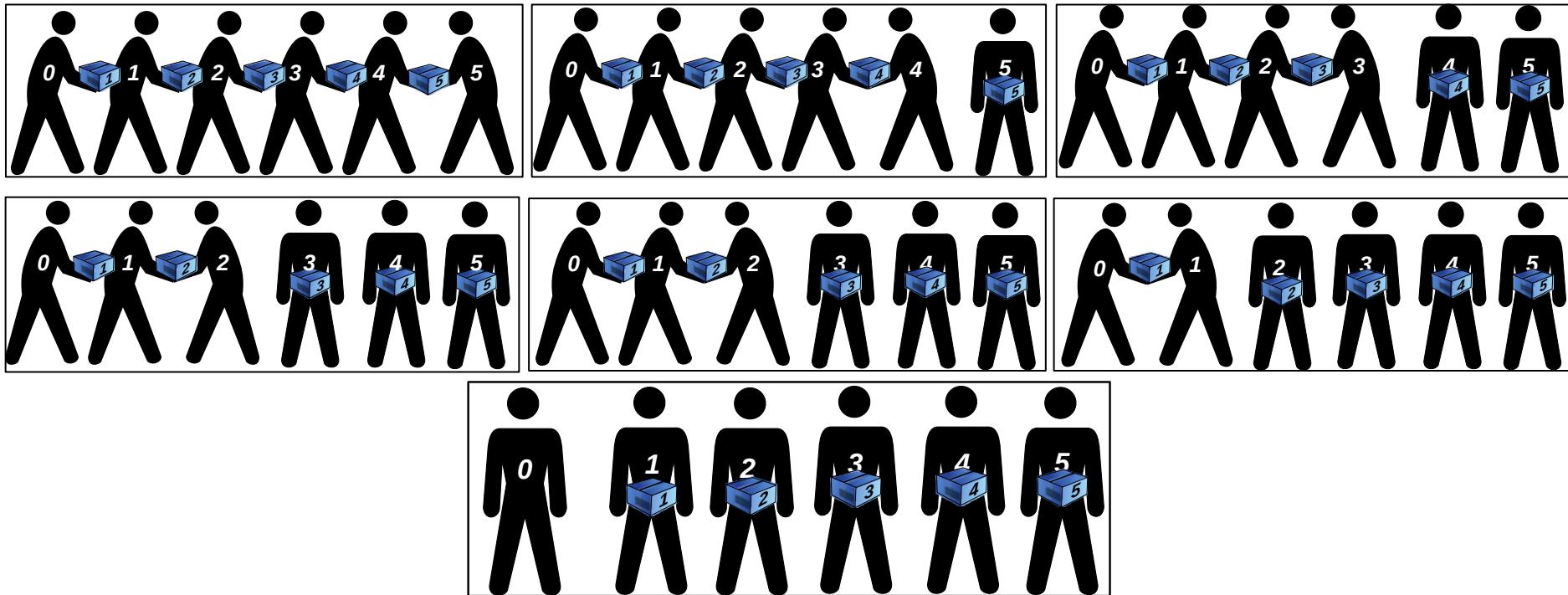
Los procesadores 0-5 deben pasar las cajas azules a la derecha y las cajas marrones a la izquierda. (El número en la caja es el proceso de destino).



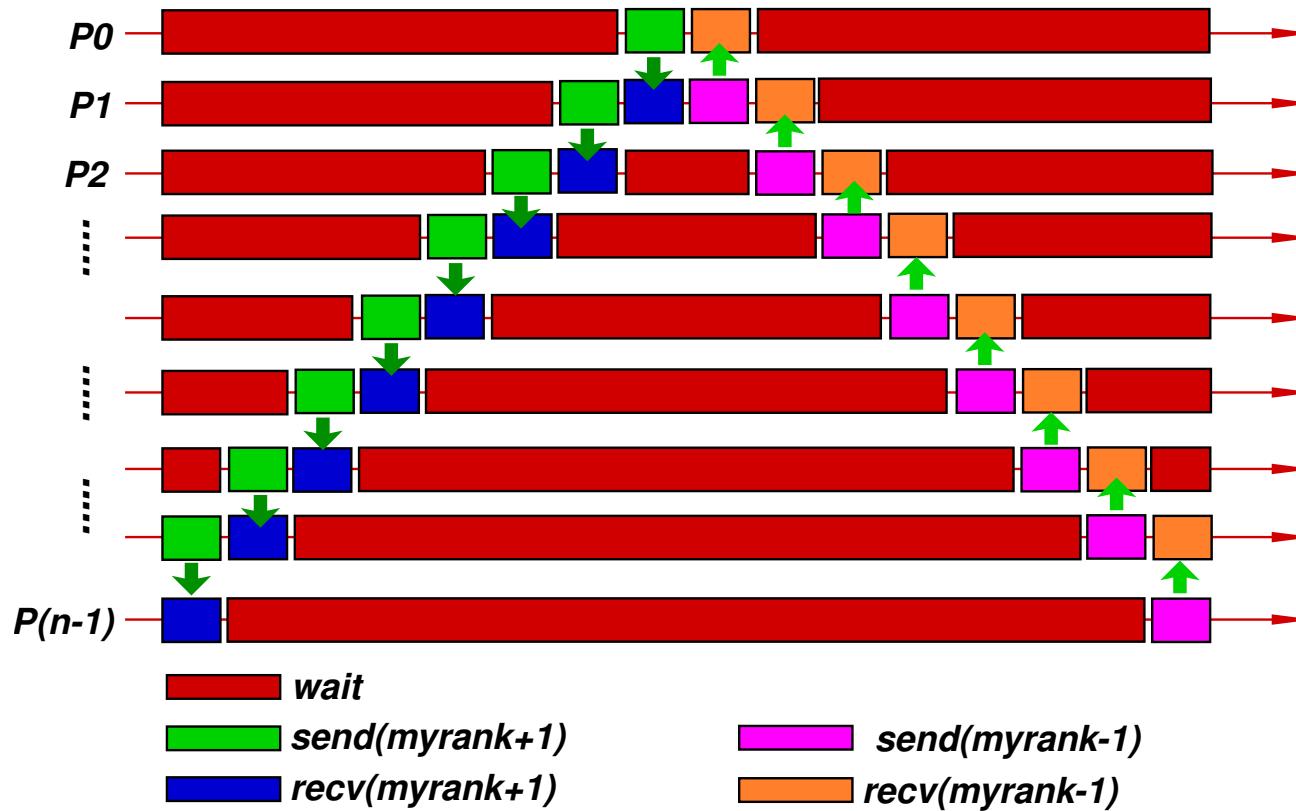
## El juego de la vida (cont.)

La comunicación hacia la derecha (cajas azules):

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

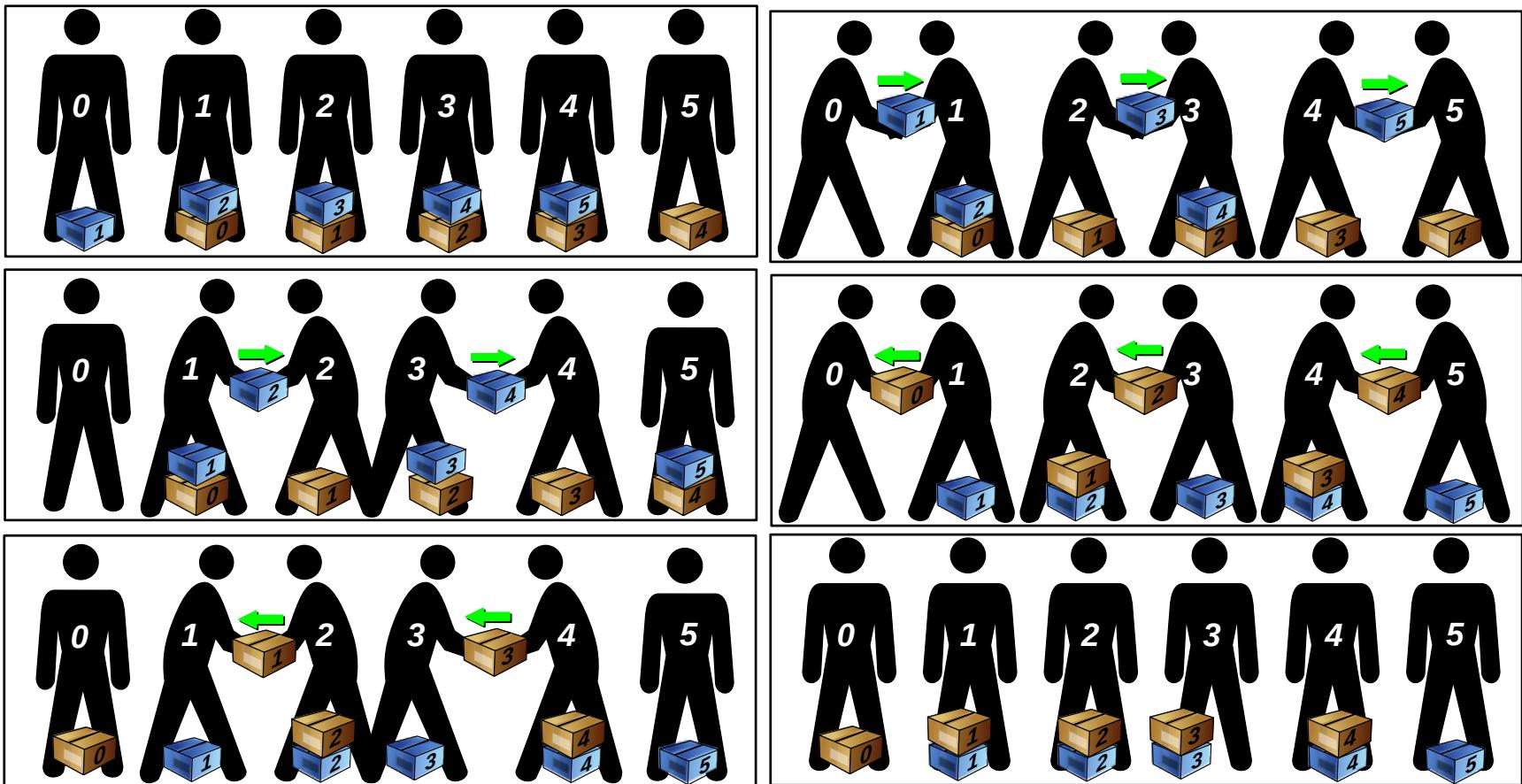


## El juego de la vida (cont.)

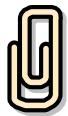


## El juego de la vida (cont.)

Haciendo un *splitting par/impar* se logra hacer toda la comunicación en **4 etapas**.



## **El juego de la vida (cont.)**

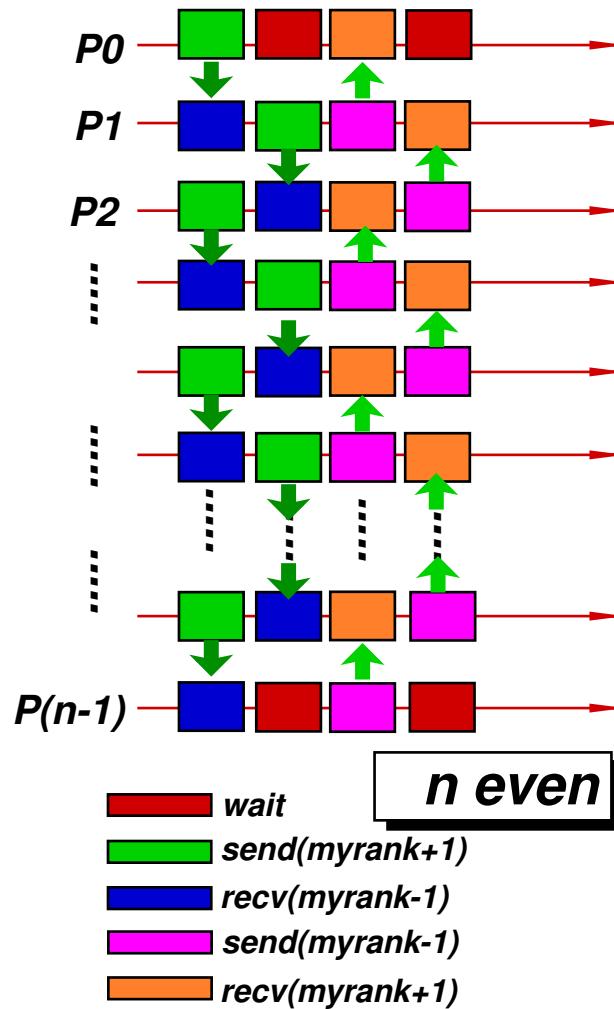


[Descargar:

[./example/lifescatsc2.cpp](#)]

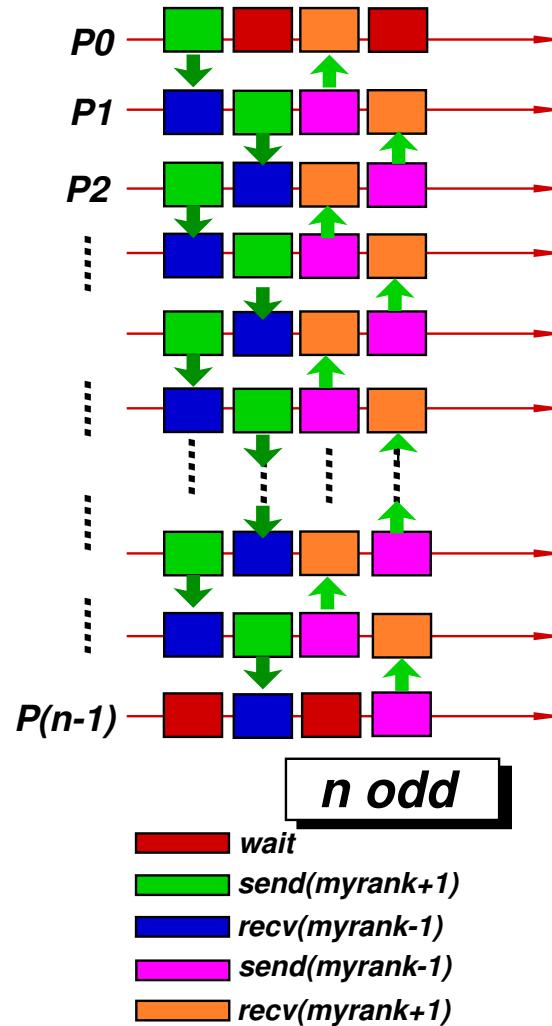
```

1. if (myrank % 2 == 0) {
2. if (myrank<size-1) {
3. SEND(i2-1,myrank+1);
4. RECEIVE(I2-1,myrank-1);
5. }
6. } else {
7. if (myrank>0) {
8. RECEIVE(I1,myrank-1);
9. SEND(i1,myrank+1);
10. }
11. }
12.
13. if (myrank % 2 == 0) {
14. if (myrank>0) {
15. SEND(i1,myrank-1);
16. RECEIVE(I1,myrank+1);
17. }
18. } else {
19. if (myrank<size-1) {
20. RECEIVE(I2-1,myrank+1);
21. SEND(i2-1,myrank-1);
22. }
23. }
```



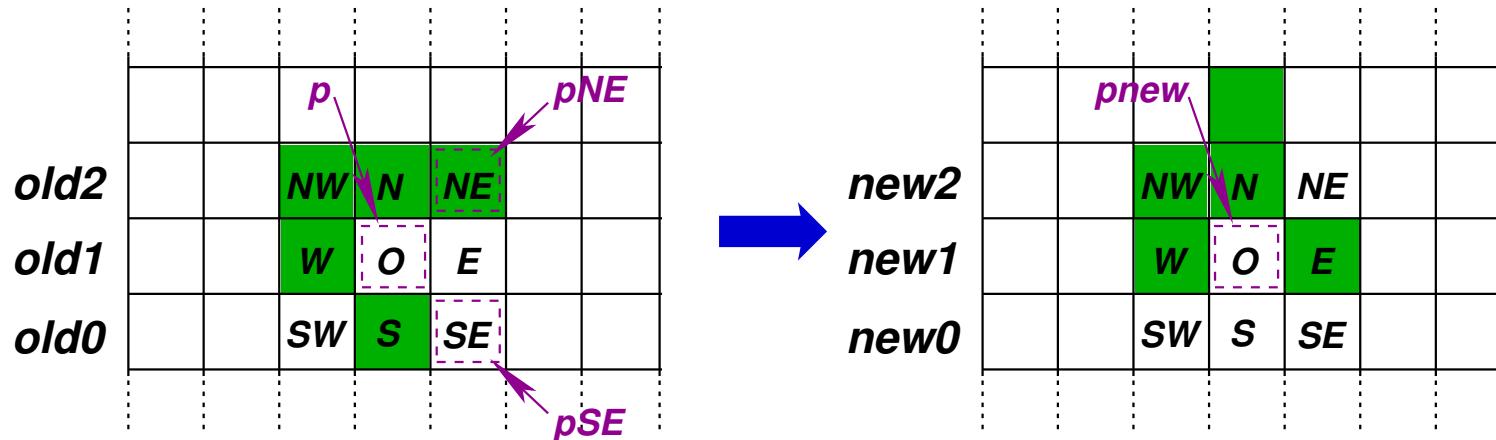
## El juego de la vida (cont.)

Esta forma de comunicación insume  $T = 4\tau$ , donde  $\tau$  es el **tiempo necesario para comunicar una fila de celdas**. Mientras que la comunicación **encadenada** previa consume  $T = 2(N_p - 1)\tau$  donde  $N_p$  es el número de procesadores. Puede aplicarse también si el número de procesadores es **ímpar**.

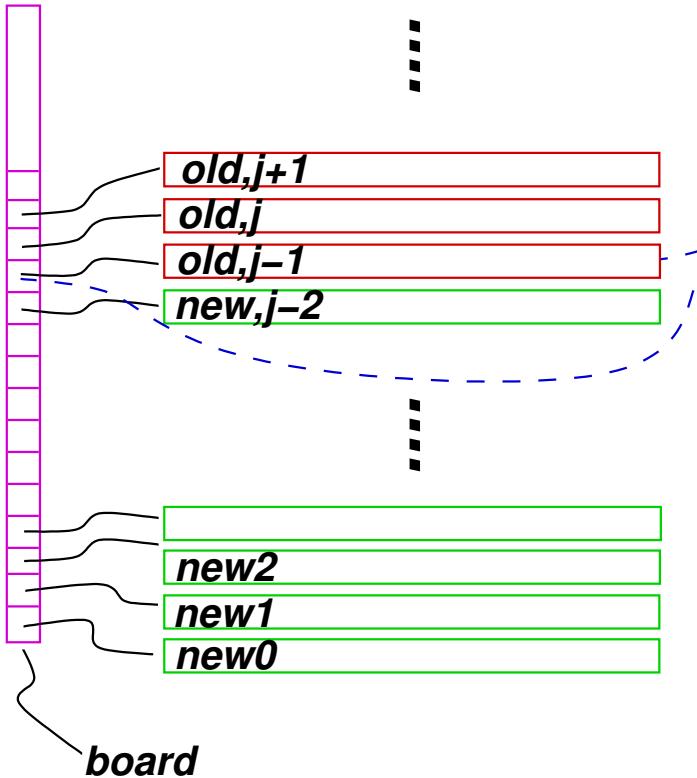


## El juego de la vida (cont.)

- Vamos calculando las celdas **por fila**, de izquierda a derecha ( $W \rightarrow E$ ) y de abajo hacia arriba ( $S \rightarrow N$ ).
- Cuando vamos a calcular la fila  $j$  necesitamos las filas  $j - 1, j$  y  $j + 1$  de la **generación anterior**, de manera que no podemos ir **sobreescritiendo** las celdas con el nuevo estado.
- Podemos mantener **dos copias** del tablero (**new** y **old**) y después hacer un **swap** (a nivel de **punteros**) para evitar la copia.



## El juego de la vida (cont.)



En realidad sólo necesitamos guardar  
*new,j*  
*new,j-1*, las **dos últimas filas calculadas** y  
vamos haciendo swap de punteros:

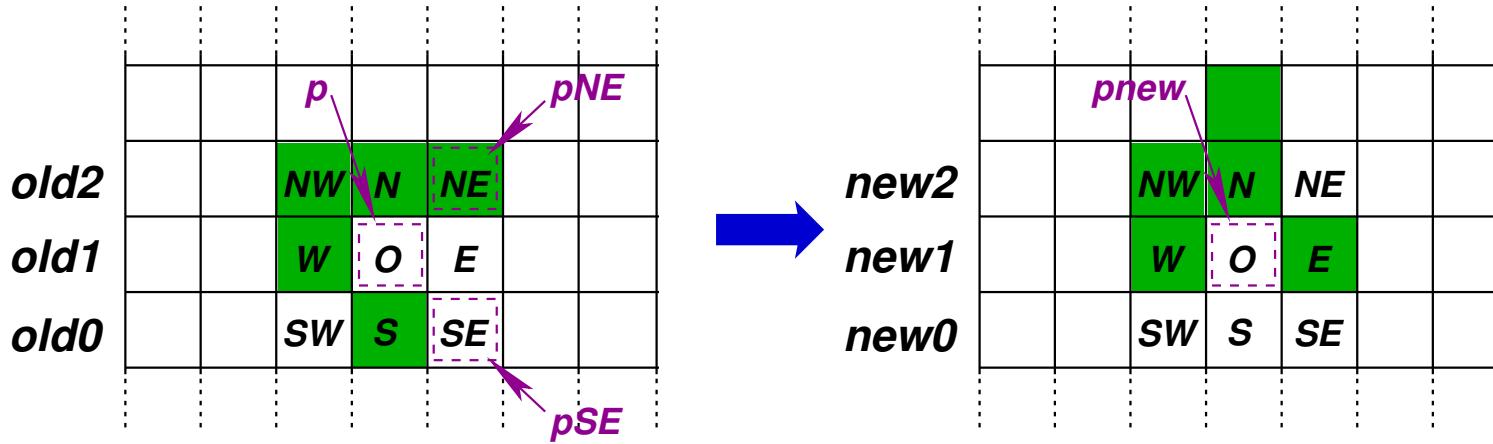


[Descargar:]

[./example/lifeswapsc.cpp](#)

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

## El juego de la vida (cont.)



- El **kernel** consiste en la función

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

que calcula el estado de la nueva fila **new1** a partir de las filas **old0**, **old1** y **old2**. La implementación que se describe a continuación ha sido optimizada lo más posible y básicamente mantiene punteros **pSE** y **pNE** a las celdas **SE** y **NE** y punteros **p** y **pnew** a las celdas **O** en sus **estados anterior y actualizado** respectivamente.

## El juego de la vida (cont.)

- Las variables enteras `ali_W`, `ali_C` y `ali_E` mantienen el *conteo del número de celdas vivas* en las columnas correspondientes, es decir

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

de manera que el número de *celdas vivas* es

1. `alive = ali_W + ali_C + ali_E - 0;`

Cada vez que se avanza una celda hacia la derecha lo único que hay que hacer es desplazar los valores de los contadores `ali_*` hacia la izquierda y recalcular `ali_E`.

Este kernel llega a 115 Mcell/sec en un Pentium 4, 2.4 GHz con memoria DDR de 400 MHz.

## Análisis de escalabilidad

Si tenemos un sistema homogéneo de  $n$  procesadores con *velocidad de cálculo*  $s$  (en cells/sec) y un tablero de  $N \times N$  celdas, entonces el *tiempo de cómputo* en cada uno de los procesadores es

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

mientras que el *tiempo de comunicación* será de

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

de manera que tenemos

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

## Análisis de escalabilidad (cont.)

Como el tiempo para 1 procesador es  $T_1 = N^2/s$ , el **speedup** será

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

y la **eficiencia** será

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

y vemos entonces que, al menos teóricamente el esquema **es débilmente escalable**, es decir podemos mantener una **eficiencia acotada** aumentando el número de procesadores mientras aumentemos simultáneamente el tamaño del problema de manera que  $N \propto n$ .

Además vemos que, para un número fijo de procesadores  $n$ , la eficiencia puede hacerse tan cercana a uno como se quiera aumentando el tamaño del tablero  $N$ .

## Análisis de escalabilidad (cont.)

Sin embargo esto no es **completamente escalable**, ya que  $N \propto n$  significa:

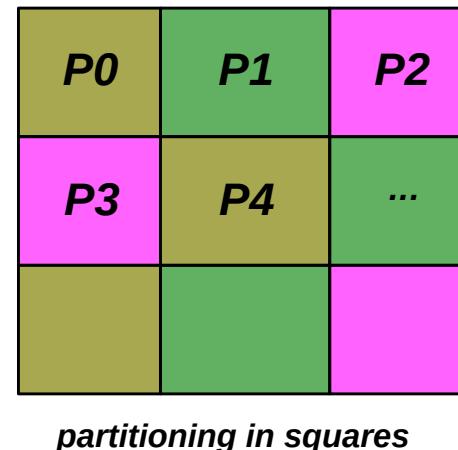
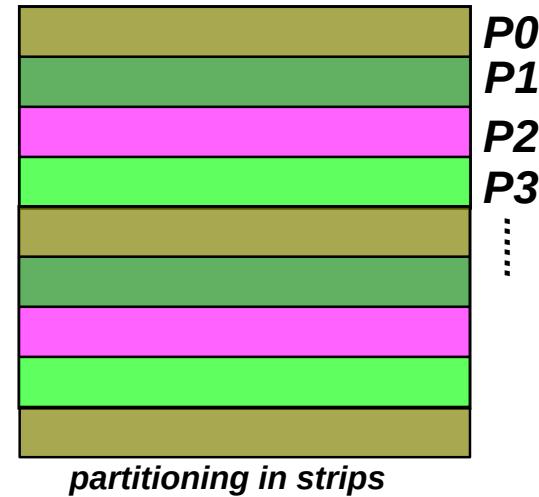
$W \propto N^2 \propto n^2$ , donde  $W$  es el trabajo a realizar. La solución es **particionar el dominio en cuadrados**, no en bandas.

Con este particionamiento

$T_{\text{comm}} = 8N/bn^{0.5}$ . (Asumimos  $n$  es un cuadrado perfecto) y la eficiencia es

$$\begin{aligned}\eta &= \frac{N^2/sn}{N^2/sn + 8N/bn^{0.5}} \\ &= \frac{1}{1 + 8n^{0.5}s/bN}\end{aligned}$$

Y entonces es suficiente con mantener  $N \propto n^{0.5}$  para mantener la eficiencia acotada, esto es  $W \propto N^2 \propto n$ . (Pero requiere que  $n$  sea un cuadrado perfecto, o implementar un particionamiento más complejo).



## Life con balance estático de carga

Hasta como fue presentado aquí, la implementación de `life` hace **balance estético de carga**. Ya hemos explicado que un inconveniente de esto es que no es claro determinar a priori cuál es la **velocidad** de los procesadores. Sin embargo en este caso podemos usar al mismo programa `life` para determinar las velocidades. Para ello debemos tomar una **estadística** de cuantas celdas calcula cada procesador y en que tiempo lo hace corriendo el programa en forma **secuencial**. Una vez calculadas las velocidades  $s_i$  esto puede pasarse como un dato al programa para que, al **distribuir las filas de celdas** lo haga en número **proporcional a la velocidad** de cada procesador.

## Life con balance seudo-dinámico de carga

El problema con el *balance estático* descrito anteriormente es que no funciona bien si la performance de los procesadores varía en el tiempo. Por ejemplo si estamos en un entorno multiusuario en el cual otros procesos pueden ser lanzados en los procesadores en forma *desbalanceada*. Una forma simple de extender el esquema de balance estático a uno dinámico es realizar una *redistribución* de la carga cada *cierto número dado de generaciones*  $m$ . La redistribución se hace con la estadística de cuantas celdas procesó cada procesador en las  $m$  generaciones precedentes. Debe prestarse atención a no incluir en este conteo de tiempo el tiempo de *comunicación y sincronización*. El  $m$  debe ser elegido cuidadosamente ya que si  $m$  es muy pequeño entonces el tiempo de redistribución será inaceptable. Por el contrario si  $n$  es muy alto entonces las variaciones de carga en tiempos menores a los que tarda el sistema en hacer  $m$  generaciones no será absorbido por el sistema de redistribución.

## Life con balance dinámico de carga

Una posibilidad de hacer **balance dinámico de carga** es usar la estrategia de **compute-on-demand**, es decir que cada esclavo pide trabajo y le es enviado un cierto “**chunk**” de filas  $N_c$ . El esclavo **calcula las celdas actualizadas** y las devuelve al master.  $N_c$  debe ser bastante menor que el número total de filas  $N$ , de lo contrario ya que se pierde un tiempo  $O(N_c/s)$  al final de cada generación por la **sincronización** en la **barrera final**. Notar que en realidad debemos enviar al esclavo **dos filas adicionales** para que pueda realizar los cálculos, es decir enviar  $N_c + 2$  filas y recibir  $N_c$  filas. En este caso el tiempo de cálculo será

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

mientras que el de comunicación será

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

## Life con balance dinámico de carga (cont.)

Siguiendo el procedimiento usado para el *PNT con balance dinámico*:

$$T_{\text{sync}} = (n/2) \times (\text{tiempo-de-procesar-un-chunk}) = (n/2) NN_c/s$$

La *eficiencia* será entonces

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

## Life con balance dinámico de carga (cont.)

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

$A/C = 2s/b(1 + s/b) = N_{c,\text{comm}}^{1/2}$  es el  $N^{1/2}$  para comunicación:

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

Si consideramos la **velocidad de procesamiento**  $s = 115$  Mcell/sec reportada previamente para el programa **secuencial** y consideramos que cada celda necesita un byte, entonces para un hardware de red tipo **Fast Ethernet con TCP/IP** tenemos un ancho de banda de  $b = 90$  Mbit/sec = 11 Mcell/sec. El cociente es entonces  $s/b \approx 10$ . Para  $N_c \gg 10$  el término  $A/N_c$  será despreciable.

## Life con balance dinámico de carga (cont.)

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

Por otra parte el término  $BN_c$  será despreciable para

$$N_c \ll C/B = N_{c,\text{sync}}^{1/2} = (1 + s/b)2N/n.$$

Podemos lograr que el ancho de la *ventana*  $[N_{c,\text{comm}}^{1/2}, N_{c,\text{sync}}^{1/2}][A/C, C/B]$  sea tan grande como queramos, aumentando suficientemente  $N$ . Sin embargo, *la eficiencia máxima está acotada* por

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

Esto es debido a que tanto el cálculo como la comunicación son *asintóticamente independientes* de  $N_c$ . Mientras que, por ejemplo, en la implementación del PNT no es así, ya que el tiempo de procesamiento es  $O(N_c)$ , mientras que el de comunicación es *independiente* del tamaño del chunk.

## Life con balance dinámico de carga (cont.)

El algoritmo para *life* descripto hasta el momento tiene el inconveniente de que el *tiempo de cálculo es del mismo orden que el de cómputo*

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

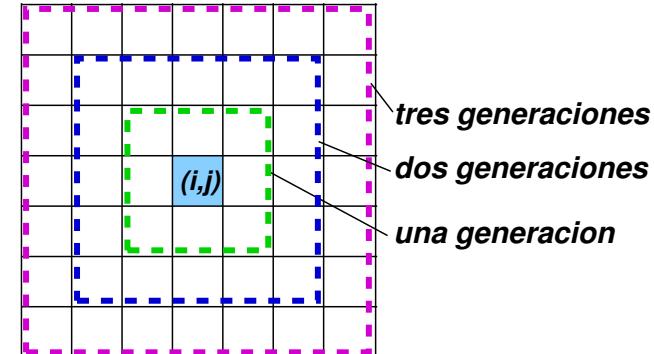
Por lo tanto, no hay ningún parámetro que nos permita *incrementar la eficiencia* más allá del *valor límite*  $(1 + s/b)^{-1}$ . Este valor límite está dado por el *hardware*. Puede ser que para Fast Ethernet (100 Mbit/sec) sea malo, mientras que para Gigabit Ethernet (1000 Mbit/sec) sea bueno. O al revés, para un dado hardware de red, puede ser que la eficiencia sea buena para un procesador lento y mala para uno rápido.

Cuando el ancho de banda  $b$  es tan lento como para que la *eficiencia máxima*  $(1 + s/b)^{-1}$  sea pobre (digamos por debajo de 0.8), entonces decimos que el procesador “*se queda con hambre de datos*” (“*data starving*”).

## Mejorando la escalabilidad de Life

Una forma de mejorar la **escalabilidad** es buscar una forma de poder hacer **más cálculo** en el procesador **sin necesidad de transmitir más datos**. Esto lo podemos hacer, por ejemplo, incrementando el número de generaciones que se evalúan en forma independiente en cada procesador.

Notemos que si queremos evaluar el estado de la celda  $(i, j)$  en la generación  $n + 1$  entonces hacen falta los estados de las celdas  $c_{i'j'}$  con  $|i' - i|, |j' - j| \leq 1$ , en la generación  $n$ , esto es la zona de dependencia de los datos de la celda  $(i, j)$  es un cuadrado de  $3 \times 3$  celdas para pasar de una generación a la otra. Para pasar a la generación siguiente  $n + 2$  necesitamos 2 capas, y así siguiendo. En general para pasar de la generación  $n$  a la  $n + k$  necesitamos el estado de  $k$  capas de celdas.



## Mejorando la escalabilidad de Life (cont.)

Es decir que si agregamos  $k$  **capas de celdas “fantasma” (“ghost”)**, entonces podemos evaluar  $k$  generaciones sin necesidad de comunicación. Por supuesto cada  $k$  generaciones hay que comunicar  $k$  capas de celdas. Si pensamos en una **descomposición unidimensional**, entonces

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

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

de manera que ahora

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

**Si queremos que  $T_{\text{comm}}/T_{\text{comp}} < 0.1$  entonces basta con tomar  $k > 10s/b$  y  $N_c \gg 4k$ .**

## Mejorando la escalabilidad de Life (cont.)

Por otra parte el **tiempo de sincronización** es

$$T_{\text{sync}} = (n/2) \times (\text{tiempo-de-procesar-un-chunk}) = (n/2) k N N_c / s$$

de manera que

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

lo cual puede hacerse ***tan pequeño como se quiera*** (digamos menor que 0.1) haciendo  $N_c \ll 0.2N/n$ . Por supuesto, en la práctica, si la combinación de hardware es demasiado lenta en comunicación con respecto a procesamiento como para que  $k$  deba ser demasiado grande, entonces los tableros pueden llegar a ser demasiado grandes para que la implementación sea ***prácticamente posible***.

## Mejorando la escalabilidad de Life (cont.)

El **tiempo de sincronización** se puede reducir usando **chunks cuadrados**.

Efectivamente si dividimos el tablero de  $N \times N$  en parches de  $N_c \times N_c$  filas por columnas, entonces tenemos

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

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

de manera que

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

es igual que antes.

## Mejorando la escalabilidad de Life (cont.)

Pero el tiempo de sincronización es

$$T_{\text{sync}} = (n/2) \times (\text{tiempo-de-procesar-un-chunk}) = (n/2) k N_c^2 / s$$

de manera que

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

lo cual ahora puede hacerse menor que 0.1 haciendo

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

Esto permite obtener un rango para  $N_c$  aceptable ***sin necesidad de usar tableros tan grandes.***

## GTP 5. [LIFE] Life

Escribir una versión de Life con balance dinámico de carga según una estrategia *compute-on-demand*.

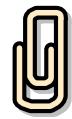
**Opcional[1]:** Utilizar procesamiento por varias generaciones al mismo tiempo (ver sección “Mejorando la escalabilidad de Life”, página [247](#)).

**Opcional[2]:** Comparar tiempos de comunicación con las siguientes variantes

- Enviar los tableros como arreglos de bytes
- Enviar los tableros como arreglos de bits
- Enviar los tableros en formato sparse.

## GTP 5. [LIFE] Life (cont.)

**Nota:** Si usan `vector<bool>` despues no van a poder extraer el arreglo de `char` para enviar. Conviene usar esta clase add hoc:



[Descargar: [./example/trybitvec.cpp](#)]

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

## GTP 5. [LIFE] Life (cont.)

Para crear el vector con

1. `bit_vector_t v(N);`

**N** es el número de bits. Con las rutinas `get()` y `set()` manipulan los valores.

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

Después finalmente para enviarlo tienen:

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

O sea que lo pueden usar directamente tanto para enviar o recibir con MPI y el tipo `MPI_CHAR`. Por ejemplo

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

# La ecuación de Poisson

## La ecuación de Poisson

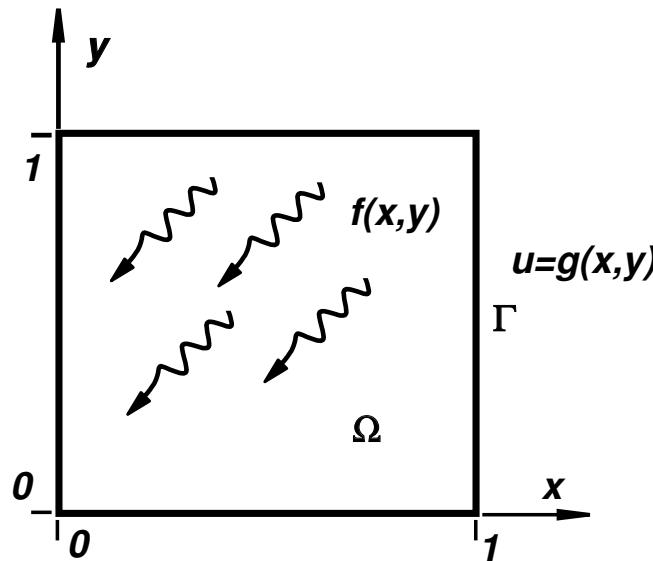
- Ejemplo de un problema de mecánica computacional con un esquema en diferencias finitas.
- Introducción del concepto de topología virtual.
- Se discute en detalle una serie de variaciones de comunicaciones send/receive para evitar “**deadlock**” y usar estrategias de comunicación óptimas.
- En general puede pensarse como la implementación en paralelo de un producto matriz vector.
- Si bien PETSc (a verse más adelante) provee herramientas que resuelven la mayoría de los problemas planteados aquí es interesante por los conceptos que se introducen y para entender el funcionamiento interno de algunos componentes de PETSc.
- Este ejemplo es ampliamente discutido en “**Using MPI**” (capítulo 4, “**Intermediate MPI**”). Código Fortran disponible en MPICH en **\$MPI\_HOME/examples/test/topol**.

## La ecuación de Poisson (cont.)

Es una PDE simple que a su vez constituye el *kernel* de muchos otros algoritmos (NS con fractional step, precondicionadores, etc...)

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

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



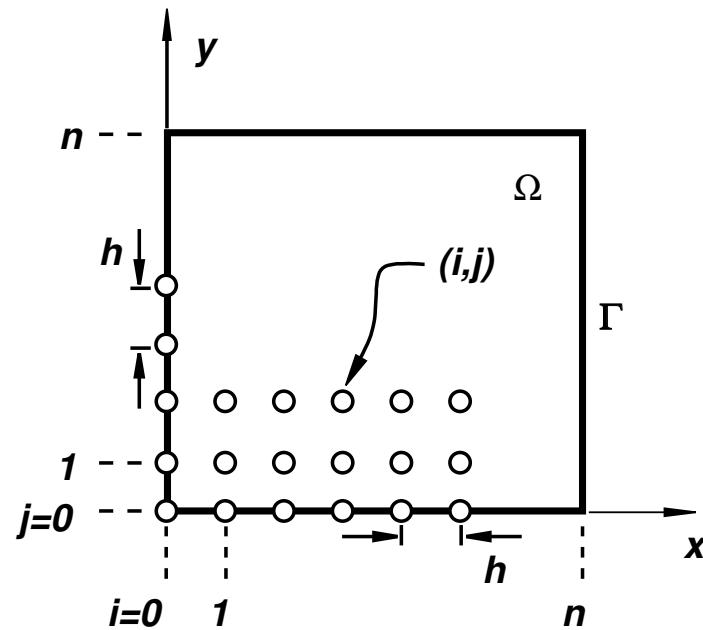
## La ecuación de Poisson (cont.)

Definimos una grilla computacional de  $n \times n$  segmentos de longitud  $h = 1/n$ .

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

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

$$x_{ij} = (x_i, y_j)$$

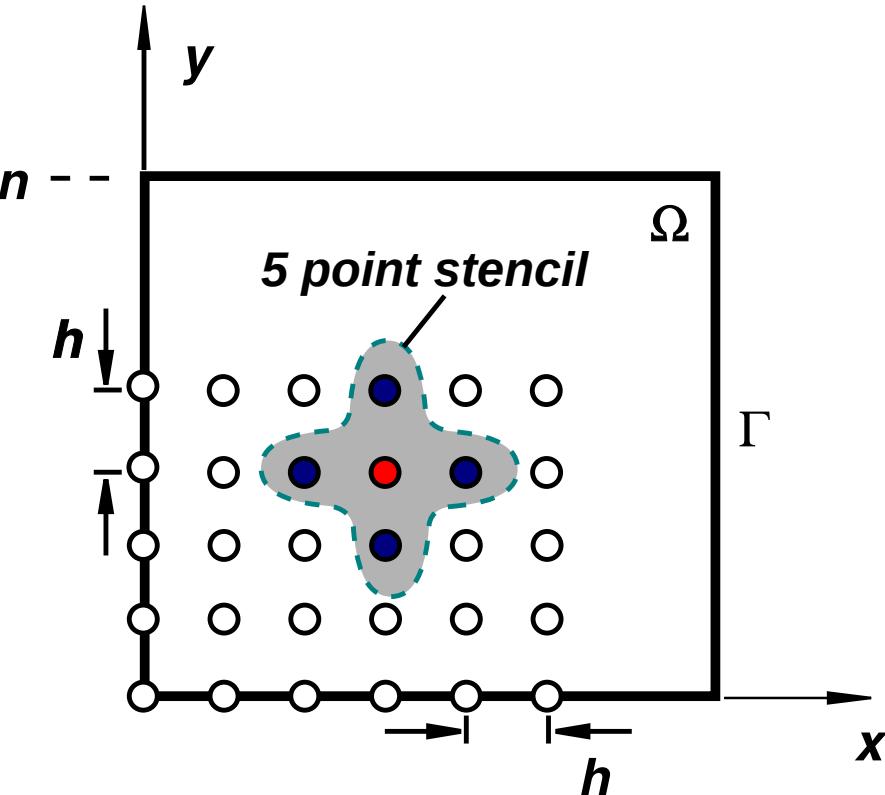


## La ecuación de Poisson (cont.)

Buscaremos valores  
aproximados para  $u$  en los  
nodos

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

Usaremos la aproximación con  
el “*stencil de 5 puntos*” que se  
obtiene al discretizar por  
diferencias finitas centradas  
de segundo orden al operador  
de Laplace



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

## La ecuación de Poisson (cont.)

Reemplazando en la ec. de Poisson

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

**Esto es un sistema lineal de  $N = (n - 1)^2$  ecuaciones (= número de puntos interiores) con  $N$  incógnitas, que son los valores en los  $N$  puntos interiores. Algunas ecuaciones refieren a valores del contorno, pero estos son conocidos, a partir de las condiciones de contorno Dirichlet.**

$$Au = f$$

## La ecuación de Poisson (cont.)

Un **método iterativo** consiste en generar una secuencia  $u^0, u^1, \dots, u^k \rightarrow u$ . Para ello llevamos a la ecuación a una **forma de punto fijo**

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

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

$$Du = f - A'u$$

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

Entonces podemos iterar haciendo

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

## La ecuación de Poisson (cont.)

Quedaría entonces,

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

que se denomina **iteración de Jacobi**. Puede demostrarse que es convergente siempre que  $A$  sea **diagonal dominante**, es decir que

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

En este caso  $A$  está justo en el límite ya que

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

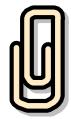
Pero puede demostrarse que con condiciones de contorno Dirichlet converge.

## La ecuación de Poisson (cont.)

Existen variantes del algoritmo (con sobrerelajación, Gauss-Seidel, por ej.) sin embargo las funciones que vamos a desarrollar básicamente implementan en paralelo lo que es un producto matriz vector genérico y por lo tanto pueden modificarse finalmente para implementar una variedad de otros esquemas iterativos como Gauss-Seidel, GC, GMRES.

## La ecuación de Poisson (cont.)

Una función en C++ que realiza estas operaciones puede escribirse así



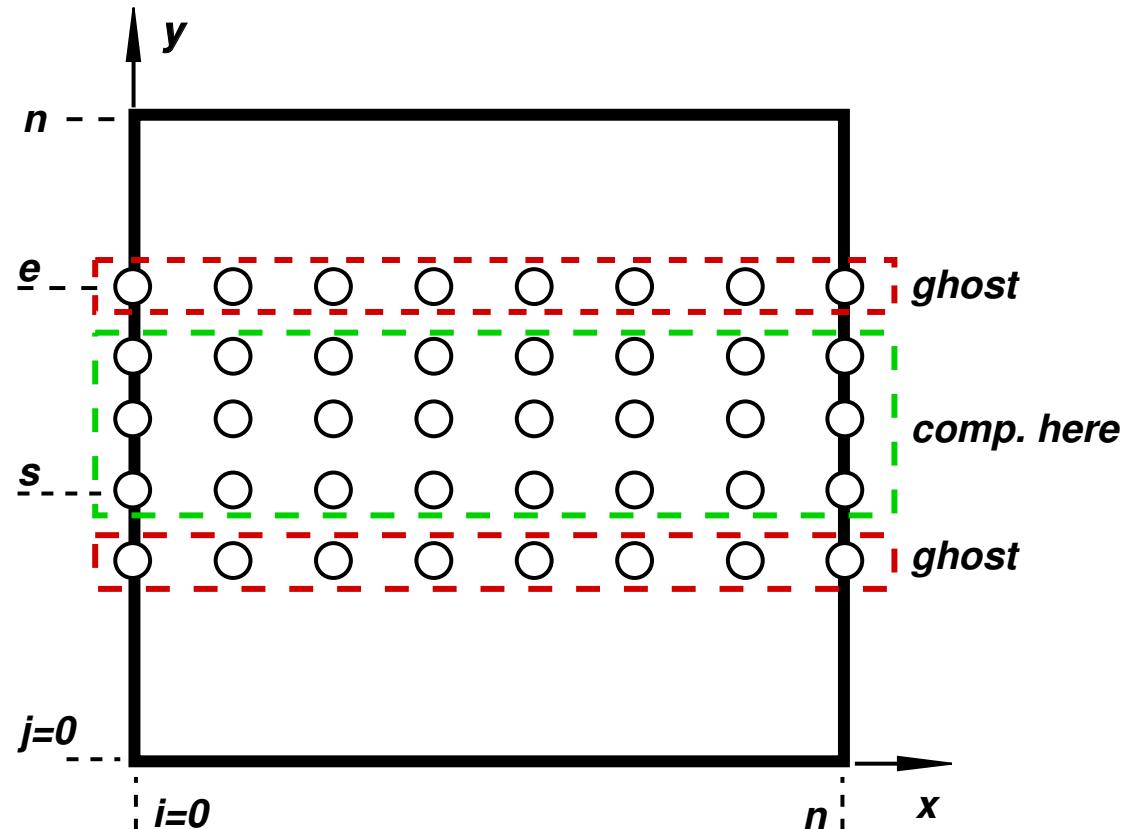
[Descargar: ./example/poissonsc.cpp]

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

```
21. for (int j=1; j<n; j++) {
22. for (int i=1; i<n; i++) {
23. UNEW(i,j) = 0.25*(U(i-1,j)+U(i+1,j)+U(i,j-1)
24. +U(i,j+1)-h*h*F(i,j));
25. }
26. }
```

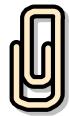
## La ecuación de Poisson (cont.)

La forma más sencilla de descomponer el problema para su procesamiento en paralelo es dividir en franjas horizontales (como en Life), de manera que en cada procesador sólo se procesan las filas en el rango  $[s, e]$ . El análisis de dependencia de los datos (como en Life) indica que también necesitamos las dos filas adyacentes (filas *ghost*).



## La ecuación de Poisson (cont.)

El código es modificado para que sólo calcule en la banda asignada.



[Descargar: ./example/poisson2sc.cpp]

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

## Topologías virtuales

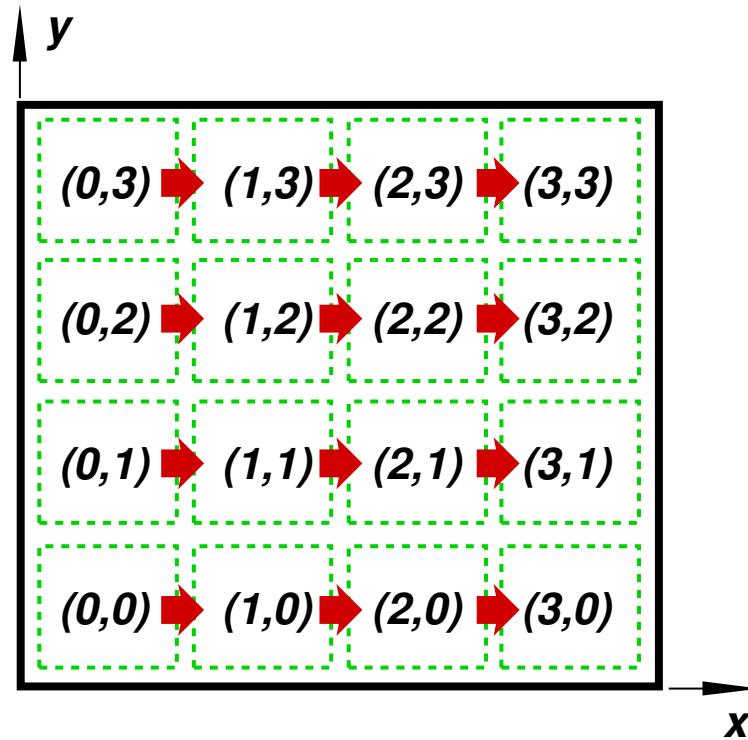
- **Particionar:** debemos definir como asignar partes (el rango  $[s, e)$ ) a cada proceso.
- La forma en que el hardware está conectado entre sí se llama *topología de la red* (anillo, estrella, toroidal ...)
- La forma óptima de particionar y asignar a los diferentes procesos puede depender del hardware subyacente. Si la topología del hardware es tipo anillo, entonces conviene asignar franjas consecutivas en el orden en que avanza el anillo.
- En muchos programas cada proceso comunica con un número reducido de procesos vecinos. Esta es la *topología de la aplicación*.

## Topologías virtuales (cont.)

- Para que la implementación paralela sea eficiente debemos hacer que la *topología de la aplicación* se adapte a la *topología del hardware*.
- Para el problema de Poisson parecería ser que el mejor orden es asignar procesos con rangos crecientes desde el fondo hasta el tope del dominio computacional, pero esto puede depender del hardware. MPI permite al vendedor de hardware desarrollar rutinas de topología especializadas a través de la implementación de las funciones de MPI de topología.
- Al elegir una “*topología de la aplicación*” o “*topología virtual*”, el usuario está diciendo como va a ser preponderantemente la comunicación. Sin embargo, como siempre, todos los procesos podrán comunicarse entre sí.

## Topologías virtuales (cont.)

La topología virtual más simple (y usada corrientemente en aplicaciones numéricas) es la “*cartesiana*”. En la forma en que lo describimos hasta ahora, usaríamos una topología cartesiana 1D, pero en realidad el problema llama a una topología 2D. También hay topologías cartesianas 3D, y de dimensión arbitraria.



## Topologías virtuales (cont.)

En la topología cartesiana 2D a cada proceso se le asigna una tupla de dos números ( $I, J$ ). MPI provee una serie de funciones para definir, examinar y manipular estas topologías.

La rutina `MPI_Cart_create(...)` permite definir una topología cartesiana 2D, su signatura es

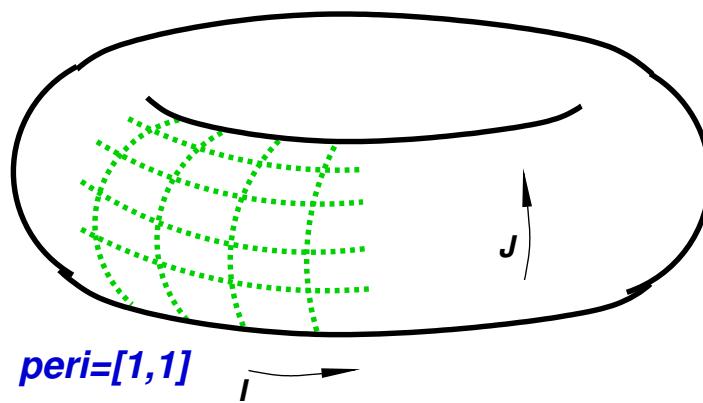
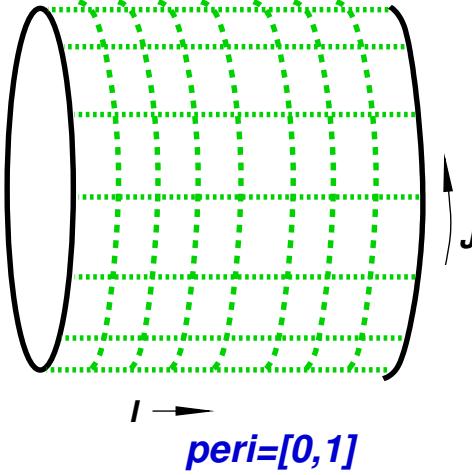
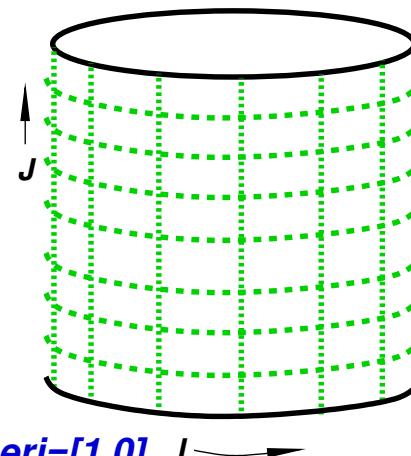
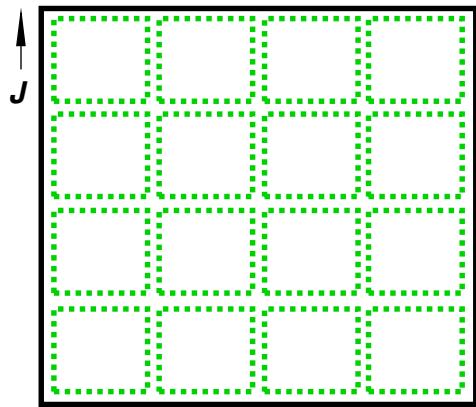
1. `MPI_Cart_create(MPI_Comm comm_old, int ndims,`
2.     `int *dims, int *periods, int reorder, MPI_Comm *new_comm);`

`dims` es un arreglo que contiene el número de filas/columnas en cada dirección, y `periods` un arreglo de *flags* que indica si una dirección dada es periódica o no.

En este caso lo llamaríamos así

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

## Topologías virtuales (cont.)



## Topologías virtuales (cont.)

- **reorder**: El argumento `bool reorder` activado quiere decir que MPI puede reordenar los procesos de tal forma de optimizar la relación entre la topología virtual y la de hardware.
- `MPI_Cart_get()` permite recuperar las dimensiones, periodicidad y coordenadas (dentro de la topología) del proceso.

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

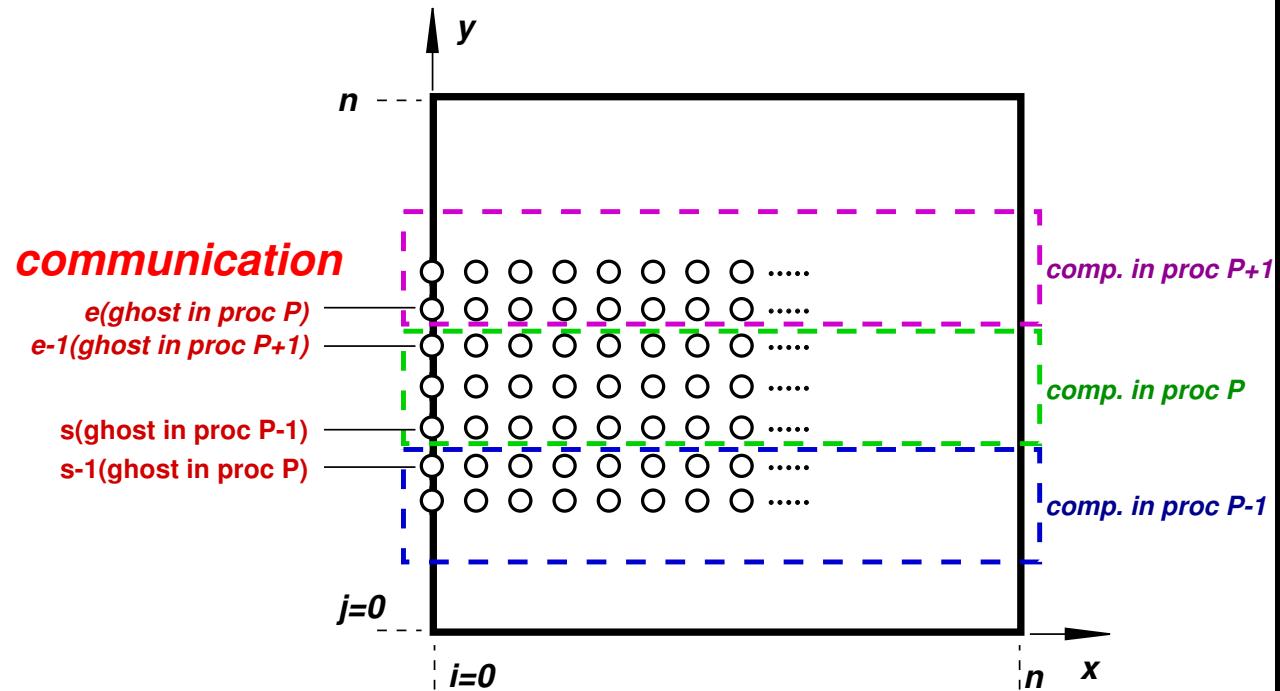
- Se pueden conseguir directamente sólo la tupla de coordenadas del proceso en la topología con

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

## Topologías virtuales (cont.)

Antes de hacer una operación de cálculo del nuevo estado  $u^k \rightarrow u^{k+1}$ , tenemos que actualizar los ***ghost values***. Por ejemplo, con una topología 1D:

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



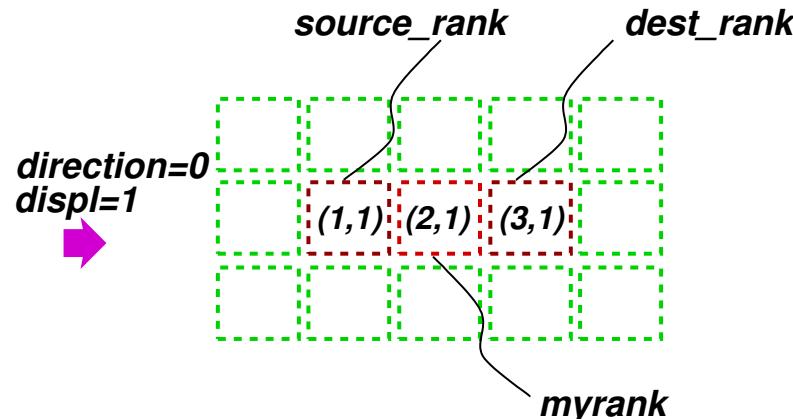
## Topologías virtuales (cont.)

En general se puede ver como que estamos haciendo un “*shift*” de los datos hacia un arriba y hacia abajo. Esta es una operación muy común y MPI provee una rutina ([MPI\\_Cart\\_shift\(\)](#)) que calcula los rangos de los procesos para una dada operación de shift:

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

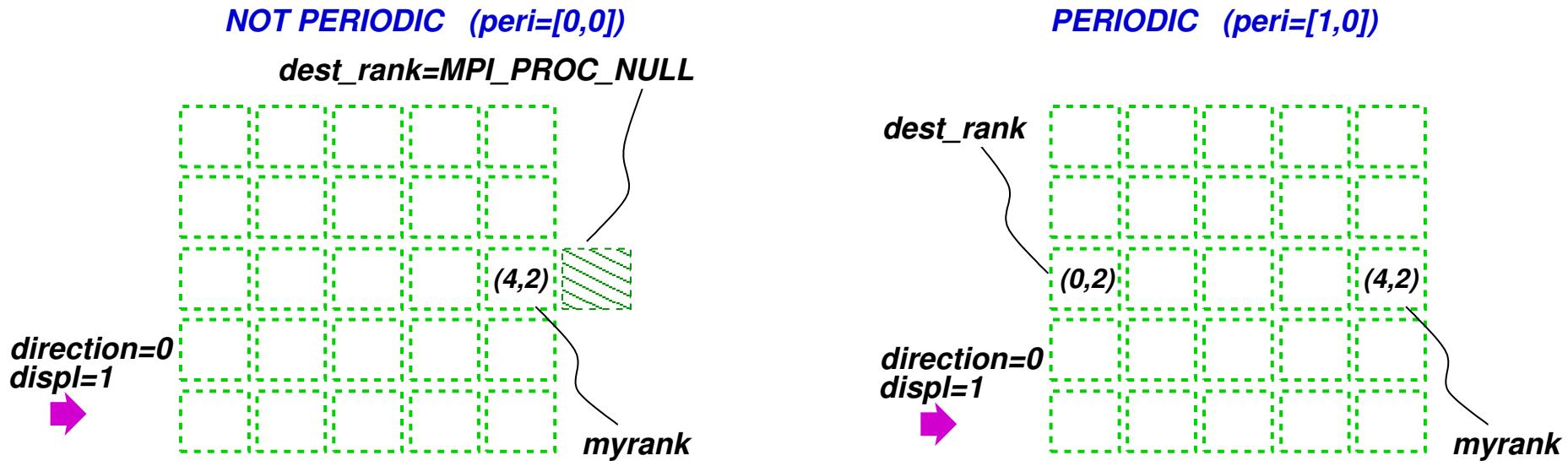
Por ejemplo, en una topología 2D, para hacer un shift en la dirección horizontal.

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



## Topologías virtuales (cont.)

Que ocurre en los bordes? Si la topología es de 5x5, entonces un shift en la dirección 0 (eje  $x$ ) con desplazamiento `displ=1` para el proceso (4, 2) da



## Topologías virtuales (cont.)

`MPI_PROC_NULL` es un *proceso nulo*. La idea es que enviar a `MPI_PROC_NULL` equivale a no hacer nada (como el `/dev/null` de Unix).

Una operación como

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

equivaldría a

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

## Topologías virtuales (cont.)

Si el número de filas a procesar  $n-1$  es múltiplo de `size`, entonces podemos calcular el rango  $[s, e)$  de las filas que se calculan en este proceso haciendo

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

Si no es múltiplo podemos hacer

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

## Topologías virtuales (cont.)

MPI provee una rutina `MPE_Decompile(...)` que hace esta operación. (Recordar que MPE es una librería que viene con MPICH pero no es parte del estándar de MPI).

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

Si el peso de los procesos no es uniforme, entonces podemos aplicar el siguiente algoritmo. Queremos distribuir `n` objetos entre `size` procesos proporcionalmente a `weights[j]` (normalizados).

A cada procesador le corresponden en principio `weights[j]*n` objetos, pero como esta cuenta puede dar no entera separamos en entero y mantisa

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

Recorremos los procesadores y vamos acumulando `carry`, cuando llega a uno, a ese procesador le asignamos un objeto más.

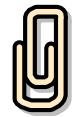
## Topologías virtuales (cont.)

Por ejemplo, queremos distribuir 1000 objetos en 10 procesadores con pesos:

```
1 in [0] weight 0.143364, wants 143.364, assign 143, carry 0.364
2 in [1] weight 0.067295, wants 67.295, assign 67, carry 0.659
3 in [2] weight 0.134623, wants 133.623, assign 134, carry 0.283
4 in [3] weight 0.136241, wants 136.241, assign 136, carry 0.523
5 in [4] weight 0.156558, wants 155.558, assign 156, carry 0.081
6 in [5] weight 0.034709, wants 33.709, assign 33, carry 0.790
7 in [6] weight 0.057200, wants 57.200, assign 57, carry 0.990
8 in [7] weight 0.131086, wants 131.086, assign 132, carry 0.076
9 in [8] weight 0.047398, wants 47.398, assign 47, carry 0.474
10 in [9] weight 0.095526, wants 94.526, assign 95, carry 0.000
11 total rows 1000
```

## Topologías virtuales (cont.)

La siguiente función reparte `n` objetos entre `size` procesadores con pesos `weights[]`.



[Descargar: [./example/decomp.cpp](#)]

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

18. }

**Esto es hacer *balance estático* de carga.**

## Ec. de Poisson. Estrateg. de comunicación

La función `exchng1(u1,u2,...)` realiza la comunicación (intercambio de valores “ghost”).



[Descargar: [./example/poisson.cpp](#)]

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

## Ec. de Poisson. Estrateg. de comunicación (cont.)

Pero la comunicación es muy lenta (ver página [225](#)).



[Descargar: [./example/poisson.cpp](#)]

```
1. void exchng2(double *a,int n,int s,int e,
2. MPI_Comm comm1d,int nbrbot,int nbrtop) {
3. MPI_Status status;
4. #define ROW(j) (&a[(j-s+1)*(n+1)])
5.
6. // Shift data to top
7. MPI_Sendrecv(ROW(e-1),n+1,MPI_DOUBLE,nbrtop,0,
8. ROW(s-1),n+1,MPI_DOUBLE,nbrbot,0,
9. comm1d,&status);
10.
11. // Shift data to bottom
12. MPI_Sendrecv(ROW(s),n+1,MPI_DOUBLE,nbrbot,0,
13. ROW(e),n+1,MPI_DOUBLE,nbrtop,0,
14. comm1d,&status);
15. }
```

## Ec. de Poisson. Estrateg. de comunicación (cont.)

Casi tenemos todos los componentes necesarios para resolver el problema.

- Agregamos una función `sweep(u1,u2,...)` que realiza la iteración de Jacobi, calculando el nuevo estado  $u^{k+1}$  (`u2`) en términos del anterior  $u^k$  (`u1`). El código para esta función se puede realizar fácilmente a partir de código presentado más arriba.
- Una función `double diff(u1,u2,...)` que calcula la norma de la diferencia entre los estados `u1` y `u2` en cada procesador.

## Ec. de Poisson. Estrateg. de comunicación (cont.)



[Descargar: ./example/poisson.cpp]

```
1. int main(int argc, char **argv) {
2. int n=10;
3.
4. MPI_Init(&argc,&argv);
5. // Get MPI info
6. int size,rank;
7. MPI_Comm_size(MPI_COMM_WORLD,&size);
8.
9. int periods = 0;
10. MPI_Comm comm1d;
11. MPI_Cart_create(MPI_COMM_WORLD,1,&size,&periods,1,&comm1d);
12. MPI_Comm_rank(MPI_COMM_WORLD,&rank);
13.
14. int direction=0, nbrbot, nbrtop;
15. MPI_Cart_shift(comm1d,direction,1,&nbrbot,&nbrtop);
16.
17. int s,e;
18. MPE_Decompose1d(n-1,size,rank,&s,&e);
19.
20. int
21. rows_here = e-s+2,
```

```
22. points_here = rows_here*(n+1);
23. double
24. *tmp,
25. *u1 = new double[points_here],
26. *u2 = new double[points_here],
27. *f = new double[points_here];
28. #define U1(i,j) u1((n+1)*(j-s+1)+i)
29. #define U2(i,j) u2((n+1)*(j-s+1)+i)
30. #define F(i,j) f((n+1)*(j-s+1)+i)
31.
32. // Assign bdry conditions to 'u1' and 'u2' ...
33. // Define 'f' ...
34. // Initialize 'u1' and 'u2'
35.
36. // Do computations
37. int iter, itmax = 1000;
38. for (iter=0; iter<itmax; iter++) {
39. // Communicate ghost values
40. exchng2(u1,n,s,e,comm1d,nbrbot,nbrtop);
41. // Compute new 'u' values from Jacobi iteration
42. sweep(u1,u2,f,n,s,e);
43.
44. // Compute difference between
45. // 'u1' (u^k) and 'u2' (u^{k+1})
46. double diff1 = diff(u1,u2,n,s,e);
47. double diffu;
48. MPI_Allreduce(&diff1,&diffu,1,MPI_DOUBLE,
49. MPI_SUM,MPI_COMM_WORLD);
```

```
50. diffu = sqrt(diffu);
51. printf("iter%d, | |u'-u| | %f\n",iter,diffu);
52.
53. // Swap pointers
54. tmp=u1; u1=u2; u2=tmp;
55.
56. if (diffu<1e-5) break;
57. }
58.
59. printf("Converged in %d iterations\n",iter);
60.
61. delete[] u1;
62. delete[] u2;
63. delete[] f;
64. }
```

# **Operaciones colectivas avanzadas de MPI**

## Operaciones colectivas avanzadas de MPI

Ya hemos visto las operaciones colectivas básicas ([MPI\\_Bcast\(\)](#) y [MPI\\_Reduce\(\)](#)). Las funciones colectivas tienen la ventaja de que permiten realizar operaciones complejas comunes en forma sencilla y eficiente.

Hay otras funciones colectivas a saber:

- [MPI\\_Scatter\(\)](#), [MPI\\_Gather\(\)](#)
- [MPI\\_Allgather\(\)](#)
- [MPI\\_Alltoall\(\)](#)

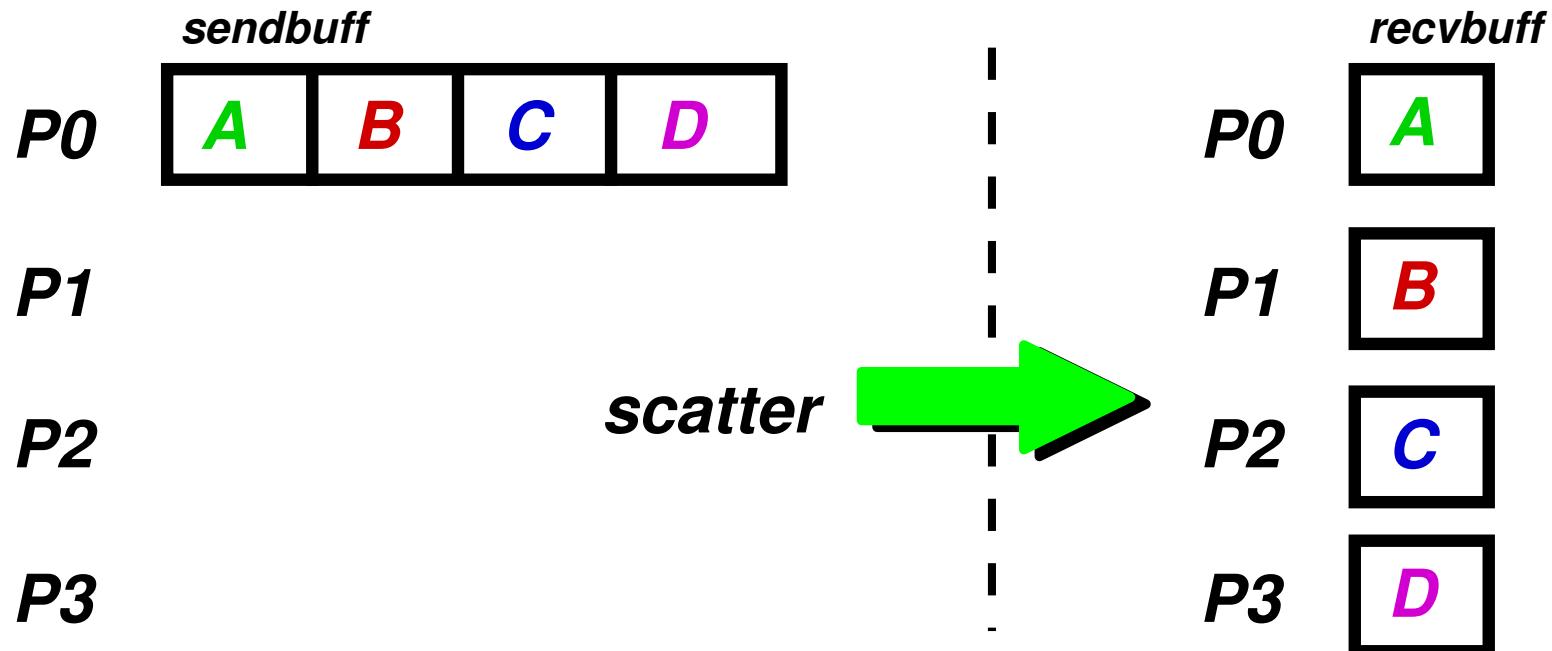
y versiones vectorizadas (de longitud variable)

- [MPI\\_Scatterv\(\)](#), [MPI\\_Gatherv\(\)](#)
- [MPI\\_Allgatherv\(\)](#)
- [MPI\\_Alltoallv\(\)](#)

## Scatter operations

Envía una cierta cantidad de datos del mismo tamaño y tipo a todos los procesos (como `MPI_Bcast()`, pero el dato a enviar depende del procesador).

1. `int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,`
2.               `void *recvbuf, int recvcnt, MPI_Datatype recvtype,`
3.               `int root,MPI_Comm comm);`



## Scatter operations (cont.)

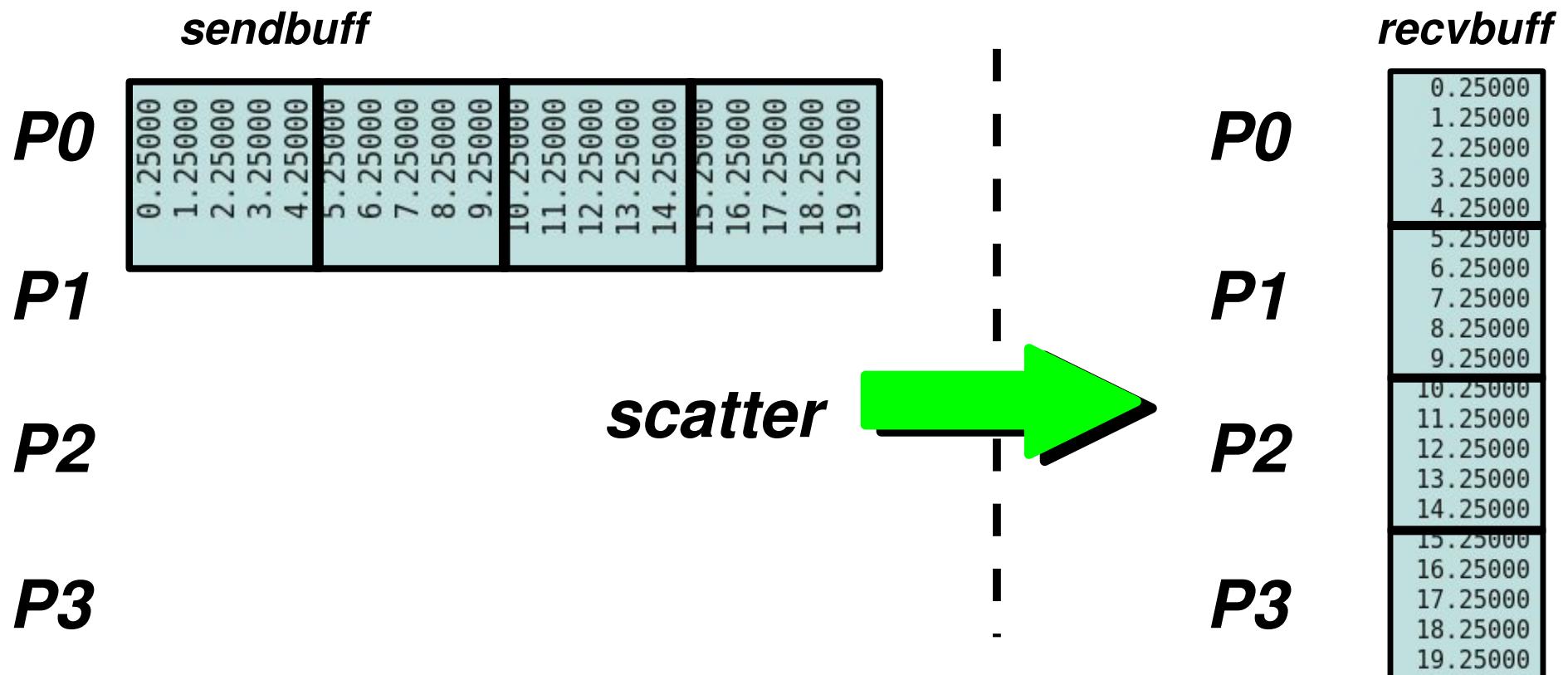


[Descargar: ./example/scatter2.cpp]

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

```
22. printf("[%d] %d ->%f\n",myrank,j,rbuff[j]);
23. MPI_Finalize();
24. if (!myrank) delete[] sbuff;
25. delete[] rbuff;
26. }
```

## Scatter operations (cont.)



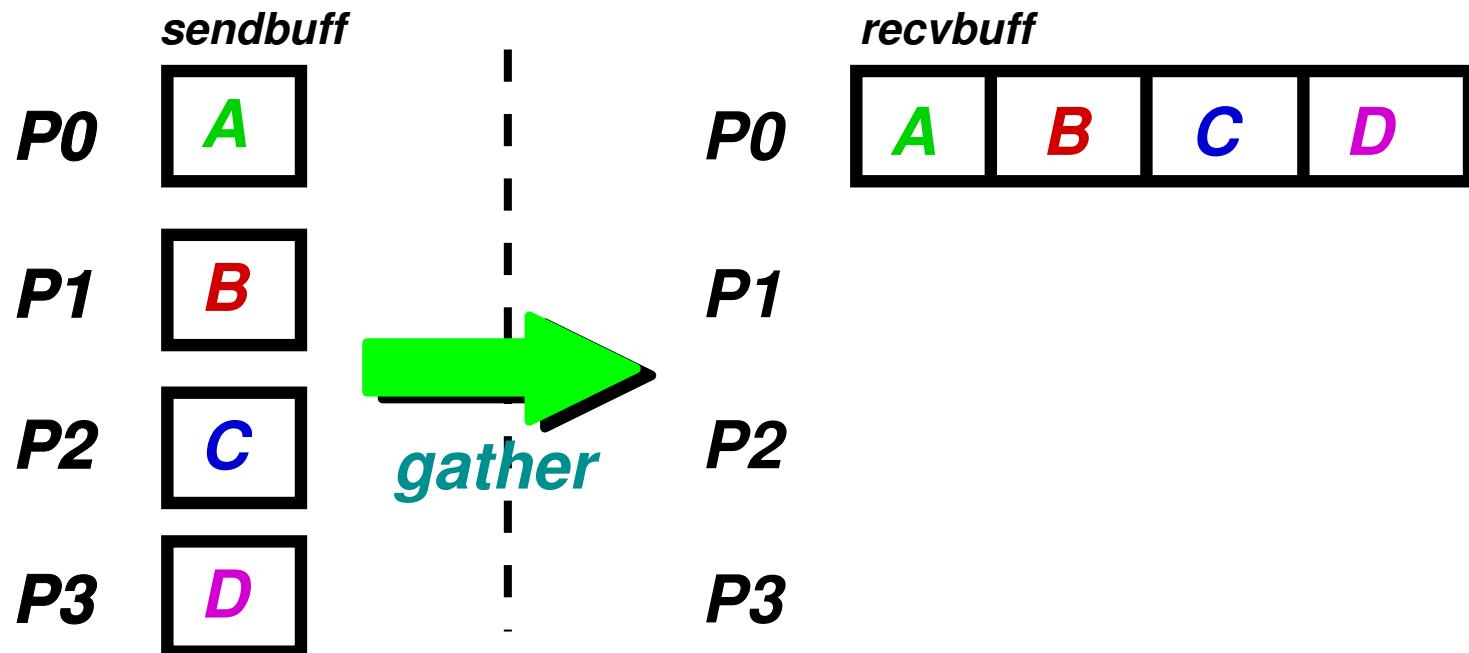
## Scatter operations (cont.)

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

## Gather operations

Es la inversa de scatter, junta (*gather*) una cierta cantidad fija de datos de cada procesador.

1. `int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,`
2.           `void *recvbuf, int recvcnt, MPI_Datatype recvtype,`
3.           `int root, MPI_Comm comm)`



## Gather operations (cont.)



[Descargar: ./example/gather.cpp]

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

```
23. for (int j=0; j<N*size; j++)
24. printf("%d ->%f\n",j,rbuff[j]);
25. MPI_Finalize();
26.
27. delete[] sbuff;
28. if (!myrank) delete[] rbuff;
29. }
```

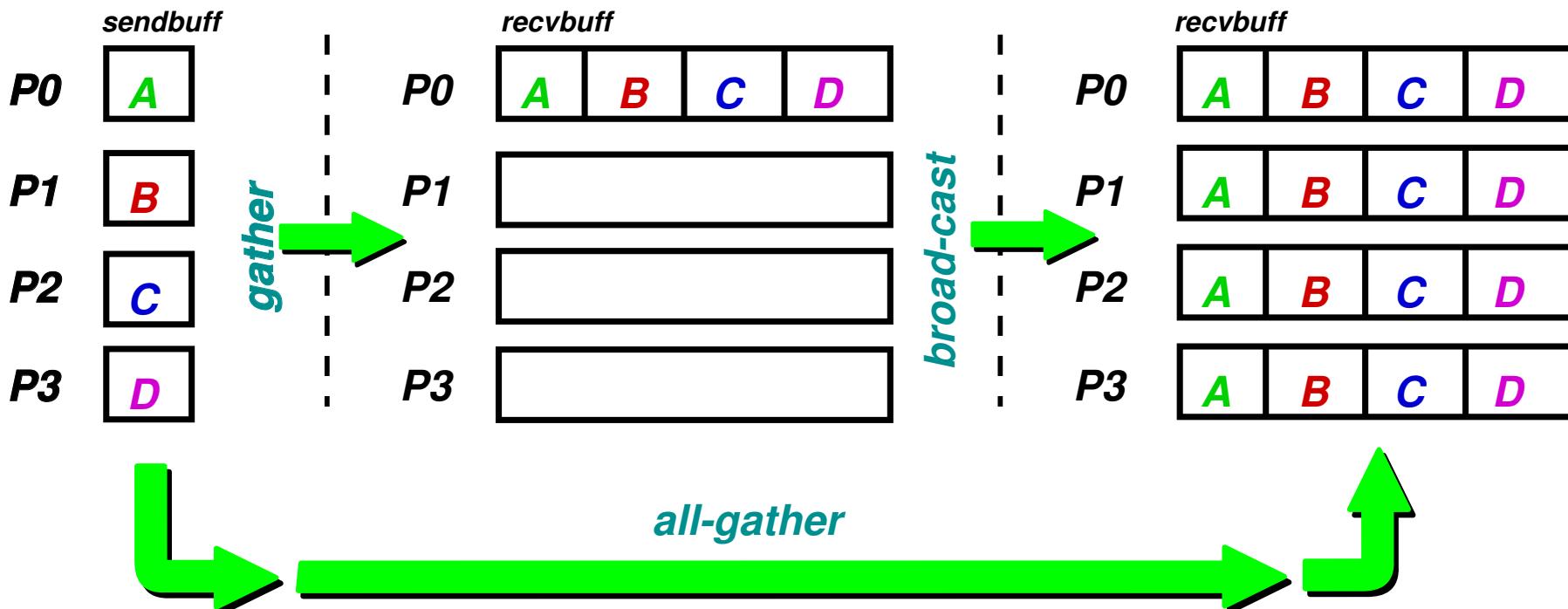
## Gather operations (cont.)

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

## All-gather operation

Equivale a hacer un *gather* seguido de un *broad-cast*.

1. `int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype,  
void *rbuf, int rcount, MPI_Datatype rtype,  
MPI_Comm comm)`



## All-gather operation (cont.)



[Descargar: ./example/allgather.cpp]

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

```
23. delete[] sbuff;
24. delete[] rbuff;
25. }
```

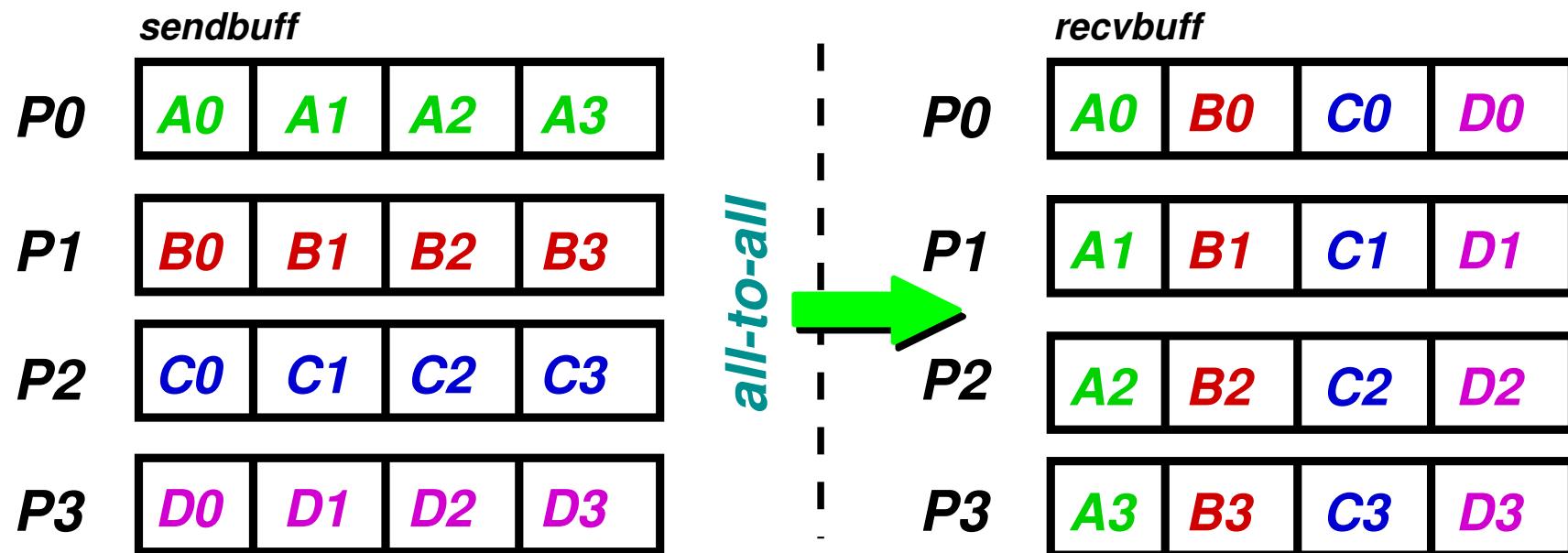
## All-gather operation (cont.)

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

## All-to-all operation

- Es equivalente a un scatter desde  $P_0$  seguido de un scatter desde  $P_1$ , etc...
- O también es equivalente a un gather a  $P_0$  seguido de un gather a  $P_1$  etc...

1. `int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,`
2. `void *recvbuf, int recvcount, MPI_Datatype recvtype,`
3. `MPI_Comm comm)`



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

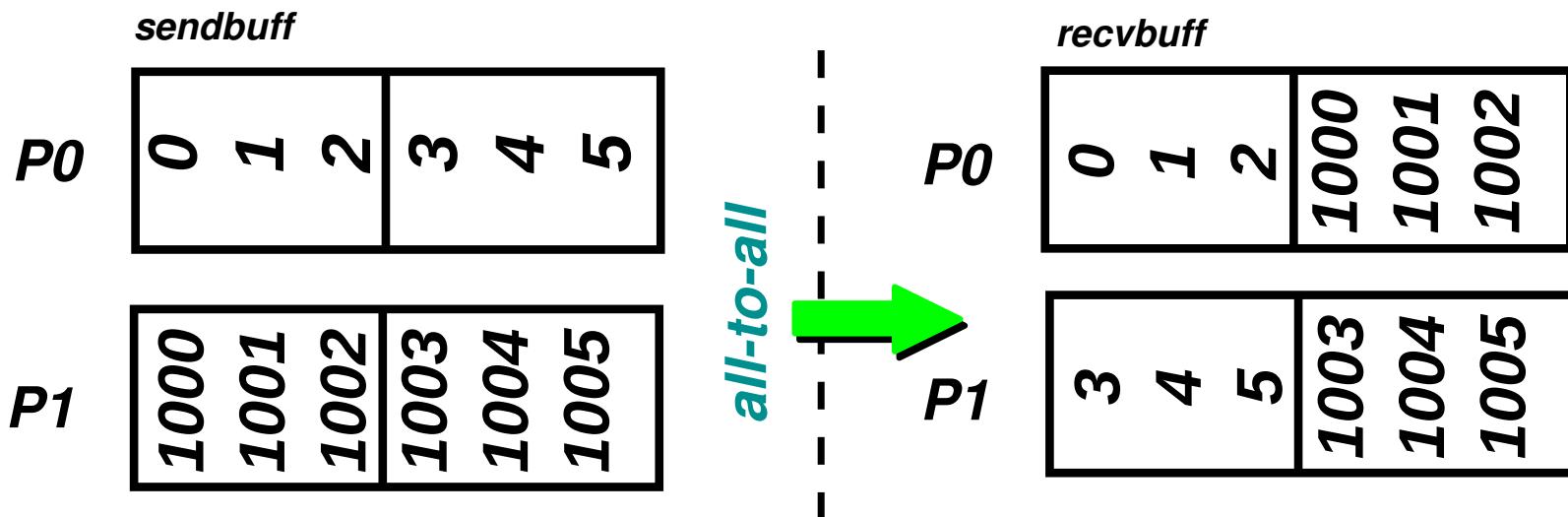


[Descargar: ./example/alltoall.cpp]

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

```
23. delete[] sbuff;
24. delete[] rbuff;
25. }
```

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



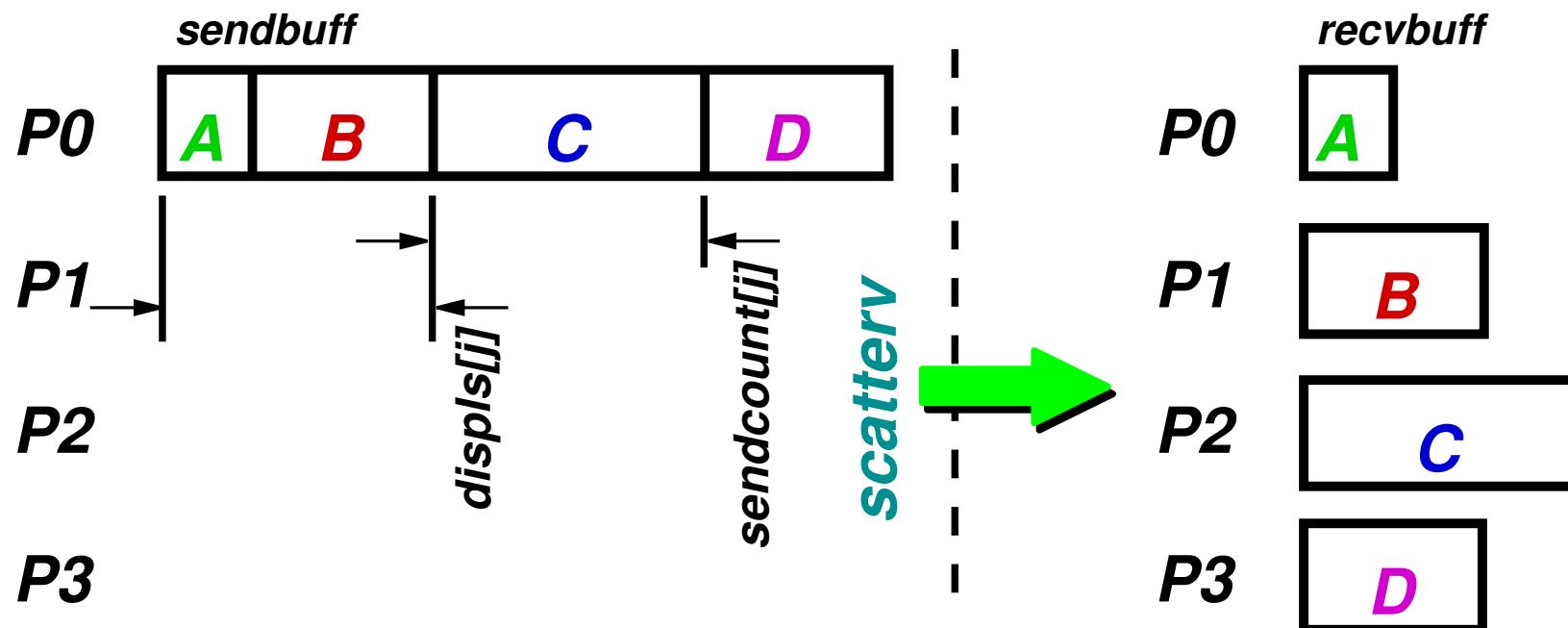
## All-to-all operation (cont.)

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

## Scatter vectorizado (longitud variable)

Conceptualmente es igual a [MPI\\_Scatter\(\)](#) pero permite que la cantidad de datos a mandar a cada procesador sea variable.

```
1. int MPI_Scatterv(void *sendbuf, int *sendcnts, int *displs,
2. MPI_Datatype sendtype, void *recvbuf, int recvnt,
3. MPI_Datatype recvtype,
4. int root, MPI_Comm comm);
```



## Scatter vectorizado (longitud variable) (cont.)



[Descargar: ./example/scatterv.cpp]

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

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

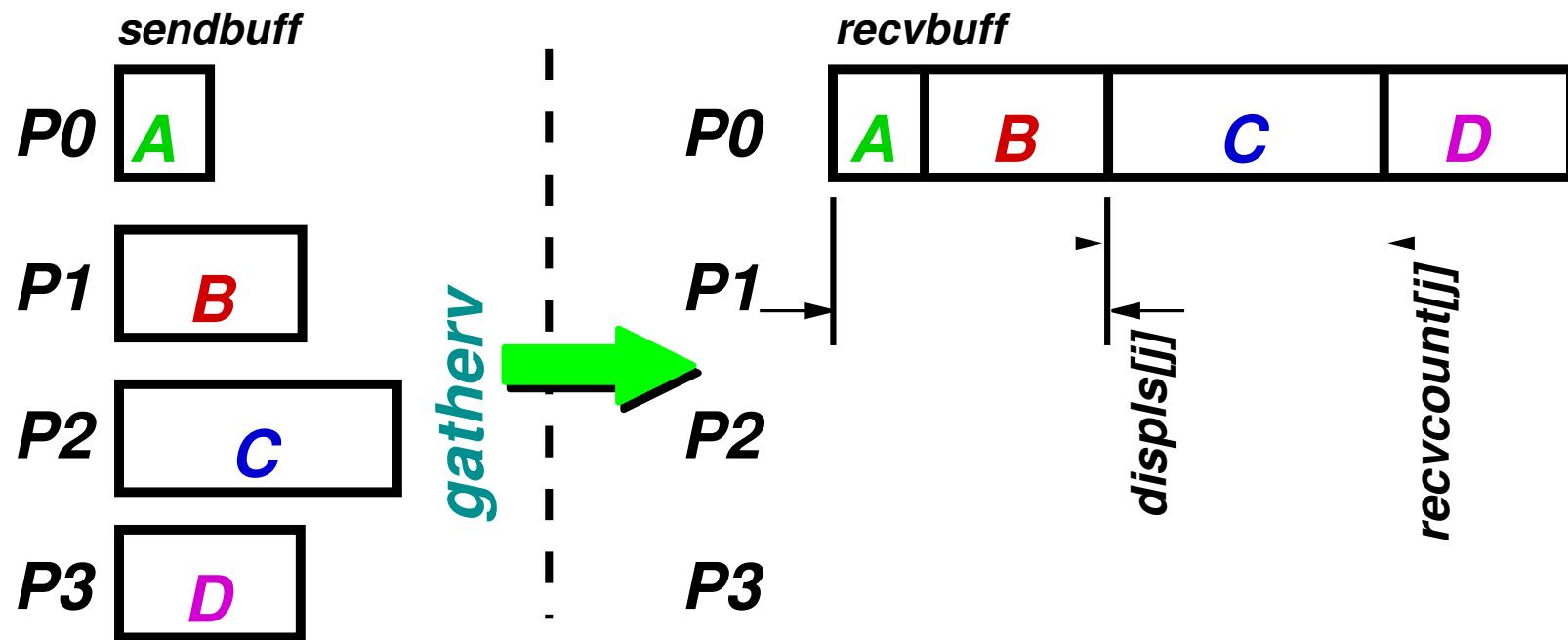
## Scatter vectorizado (longitud variable) (cont.)

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

## Gatherv operation

Es igual a *gather*, pero recibe una longitud de datos variable de cada procesador.

```
1. int MPI_Gatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
2. void *recvbuf, int *recvcnts, int *displs,
3. MPI_Datatype recvtype, int root, MPI_Comm comm);
```



## Gatherv operation (cont.)



[Descargar: ./example/gatherv.cpp]

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

```
22. rbuff,recvnts,displs,MPI_DOUBLE,
23. 0,MPI_COMM_WORLD);
```

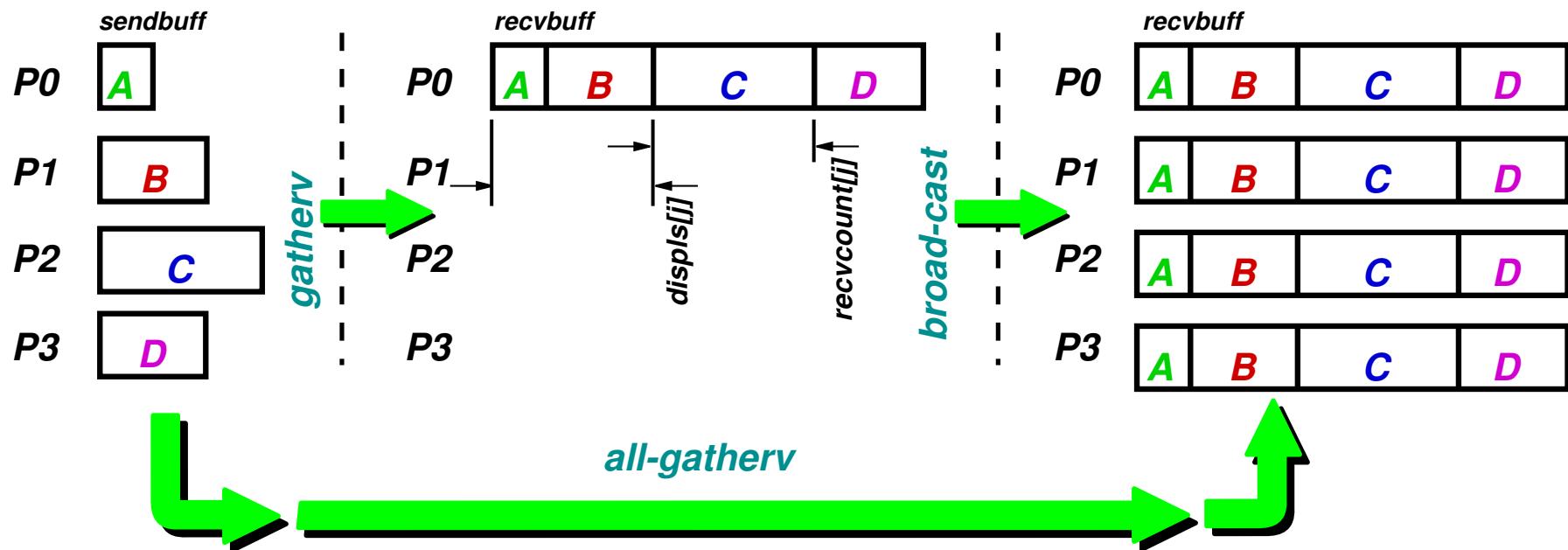
## Gatherv operation (cont.)

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

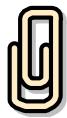
## Allgatherv operation

Es igual a *gatherv*, seguida de un *Bcast*.

1. `int MPI_Allgatherv(void *sbuf, int scount, MPI_Datatype stype,  
void *rbuf, int *rcounts, int *displs,  
MPI_Datatype rtype, MPI_Comm comm)`



## Allgatherv operation (cont.)



[Descargar: ./example/allgatherv.cpp]

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

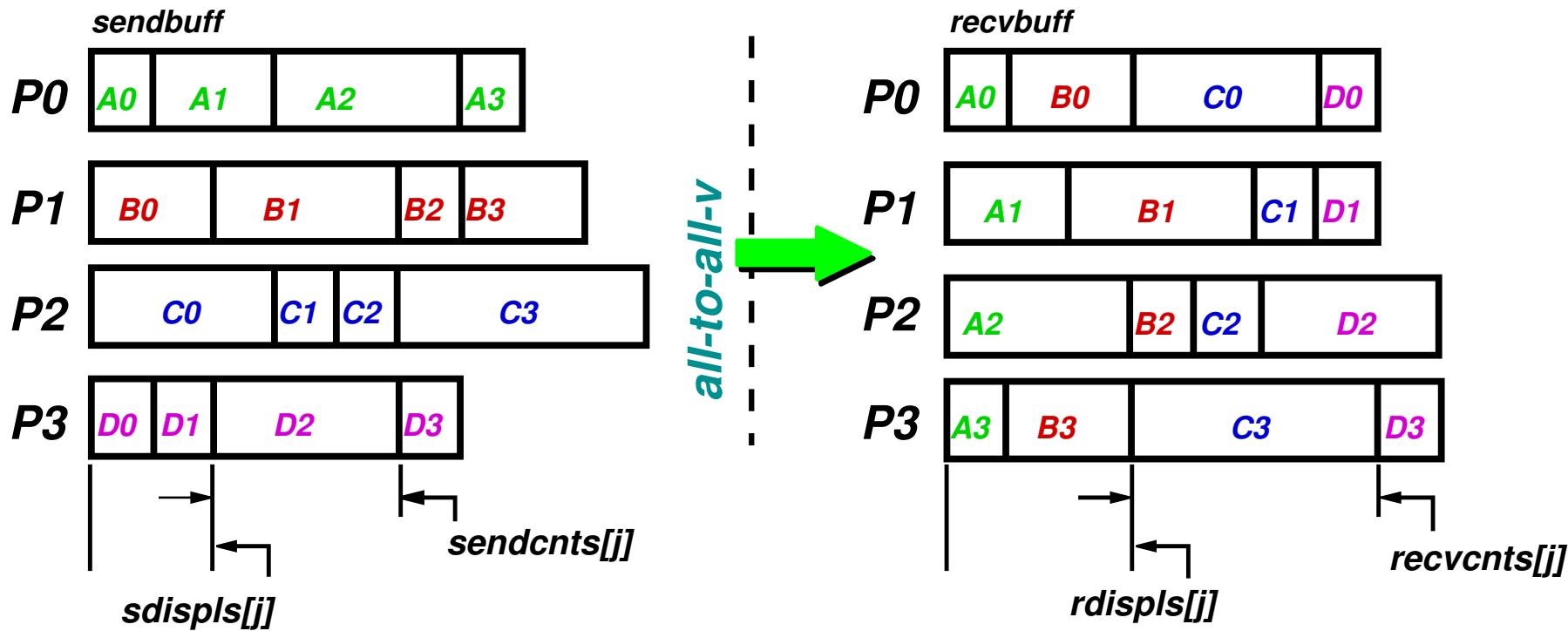
## Allgatherv operation (cont.)

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

## All-to-all-v operation

Versión vectorizada (longitud variable) de [MPI\\_Alltoall\(\)](#).

```
1. int MPI_Alltoallv(void *sbuf, int *scnts, int *sdispls,
2. MPI_Datatype stype, void *rbuf, int *rcnts,
3. int *rdispls, MPI_Datatype rtype, MPI_Comm comm);
```



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

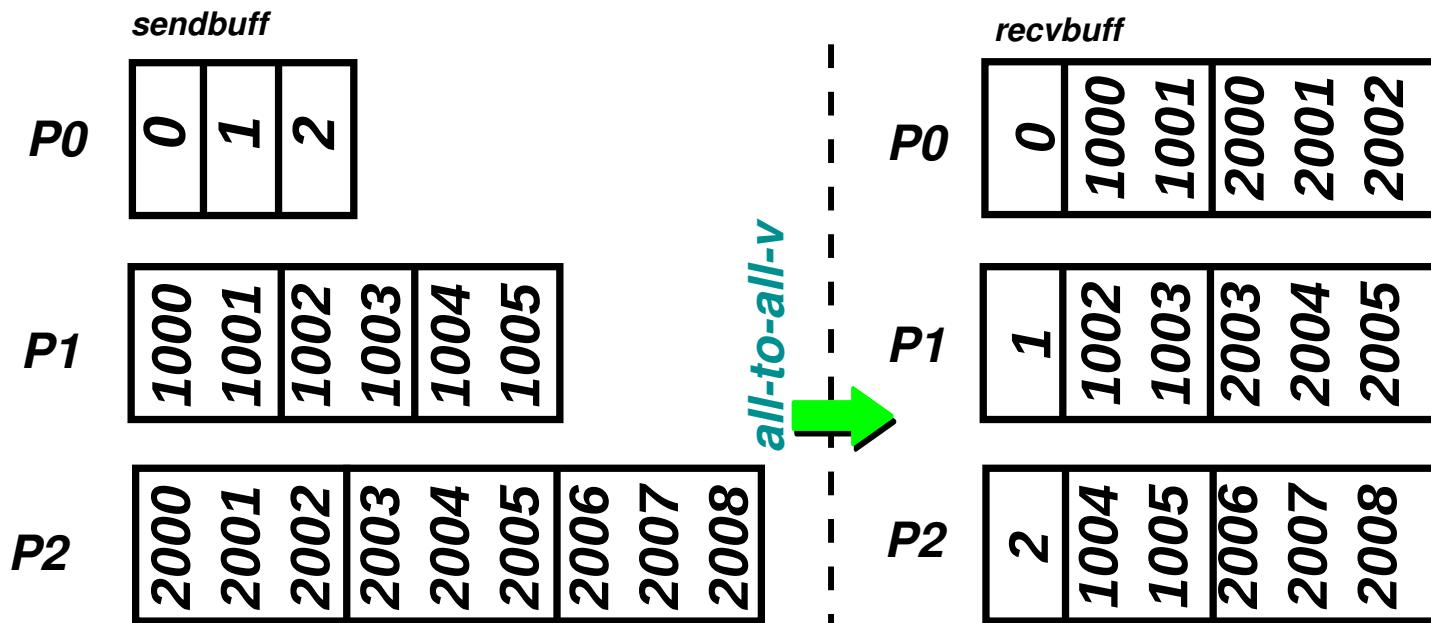


[Descargar: ./example/alltoallv.cpp]

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

```
23. MPI_Alltoallv(sbuff,sendcnts,sdispls,MPI_DOUBLE,
24. rbuff,recvcnts,rdispls,MPI_DOUBLE,
25. MPI_COMM_WORLD);
```

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

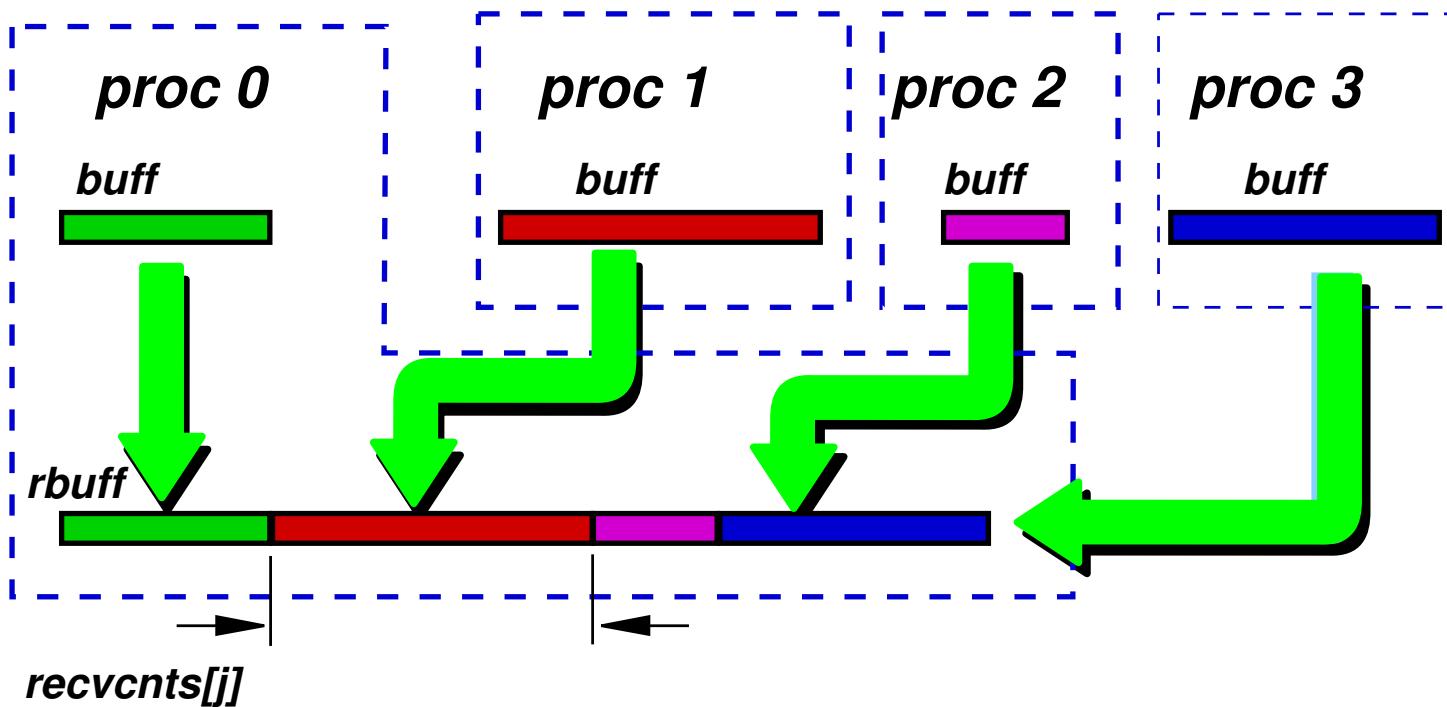


## All-to-all-v operation (cont.)

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

## La función print-par

Como ejemplo veamos una función `print_par()` que imprime los contenidos de un buffer de tamaño variable (por procesador).



## La función print-par (cont.)



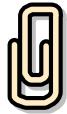
[Descargar: ./example/pparsc.cpp]

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

## La función print-par (cont.)

- Cada procesador tiene un `vector<int> buff` conteniendo elementos, escribimos una función que los imprime por pantalla todos los elementos del procesador 0, después los del 1, etc...
- El tamaño de `buff` puede ser distinto en cada procesador.
- Primero hacemos un `gather` de todos los tamaños de los vectores locales a `recvnts[]`. Después en base a `recvnts[]` calculamos los “*displacements*” `displs[]`.
- La suma de los `recvnts[]` nos da el tamaño del buffer de recepción en el procesador 0.
- Notar que esto tal vez se podría hacer más eficientemente enviando de a uno por vez los datos de los esclavos al master, ya que de esa forma no es necesario alocar un buffer con el tamaño de todos los datos en todos los procesadores, sino que sólo hace falta uno con el tamaño máximo sobre todos los procesadores.

## La función print-par (cont.)



[Descargar: ./example/resscatter.cpp]

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

```
22. displs[j] = displs[j-1] + recvnts[j-1];
23.
24. }
25. // Do the gather
26. MPI_Gatherv(&buff[0], sendcnt, MPI_INT,
27. &rbuff[0], &recvnts[0], &displs[0], MPI_INT,
28. 0, MPI_COMM_WORLD);
29. if (!myrank) {
30. // Print all buffers in master
31. if (s) printf("%s", s);
32. for (int j=0; j<size; j++) {
33. printf("in proc [%d]: ", j);
34. for (int k=0; k<recvnts[j]; k++) {
35. int ptr = displs[j]+k;
36. printf("%d ", rbuff[ptr]);
37. }
38. printf("\n");
39. }
40. }
41. }
```

## Ejemplo de uso de all-to-all-v rescatter

Como otro ejemplo consideremos el caso de que tenemos una cierta cantidad de objetos (para simplificar asumamos un arreglo de enteros). Queremos escribir una función

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

que redistribuye los elementos que están en `buff` en cada procesador de acuerdo con el criterio impuesto a la función `proc`. Esta función `proc_fun proc;` retorna el número de proceso para cada elemento del arreglo. La signatura de tales funciones está dada por el siguiente `typedef`

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

Por ejemplo, si queremos que cada procesador reciba aquellos elementos `x` tales que `x % size == myrank` entonces debemos hacer

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

## Ejemplo de uso de all-to-all-v rescatter (cont.)

Si

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

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

entonces después de **re\_scatter(buff,...,mod\_scatter,...)** debemos tener

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

## Ejemplo de uso de all-to-all-v rescatter (cont.)

Podríamos pasar parámetros globales a `mod_scatter`,

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

Entonces:

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

## Ejemplo de uso de all-to-all-v rescatter (cont.)

El argumento `void *data` permite *pasar argumentos* a la función, evitando el uso de variables globales. Por ejemplo si queremos que el envío tipo `proc = x % size` vaya rotando, es decir `proc = (x-k) % size`, debemos ser capaces de pasarle `k` a `f`, esto se hace via `void *data` de la siguiente forma.

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

## Ejemplo de uso de all-to-all-v rescatter (cont.)

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

## Ejemplo de uso de all-to-all-v rescatter (cont.)

Detalles de *Programación Funcional (FP)*:

- Algunos lenguajes facilitan la tarea de usar funciones como si fueran objetos: crearlos y destruirlos en tiempo de ejecución, pasarlo como si fueran argumentos ...
- Ciertos lenguajes dan un soporte completo para FP: Haskell, ML, Scheme, Lisp, y en menor medida Perl, Python, C/C++.
- En C++ básicamente se pueden usar definiciones tipo **typedef** como
  1. `typedef int (*proc_fun)(int x,int size,void *data);`  
Esto define un tipo especial de funciones (el tipo **proc\_fun**) que son aquellas que toman como argumento dos enteros y un **void \*** y devuelven un entero.
  - Podemos definir funciones con esta “**signatura**” (argumentos y tipo de retorno) y pasárlas como objetos a procedimientos “**de mayor orden**”.
    1. `void re_scatter(vector<int> &buff,int size,`
    2. `int myrank,proc_fun proc,void *data);`

## Ejemplo de uso de all-to-all-v rescatter (cont.)

### Ejemplos:

- Ordenar con una función de comparación:

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

Por ejemplo, ordenar por valor absoluto:

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

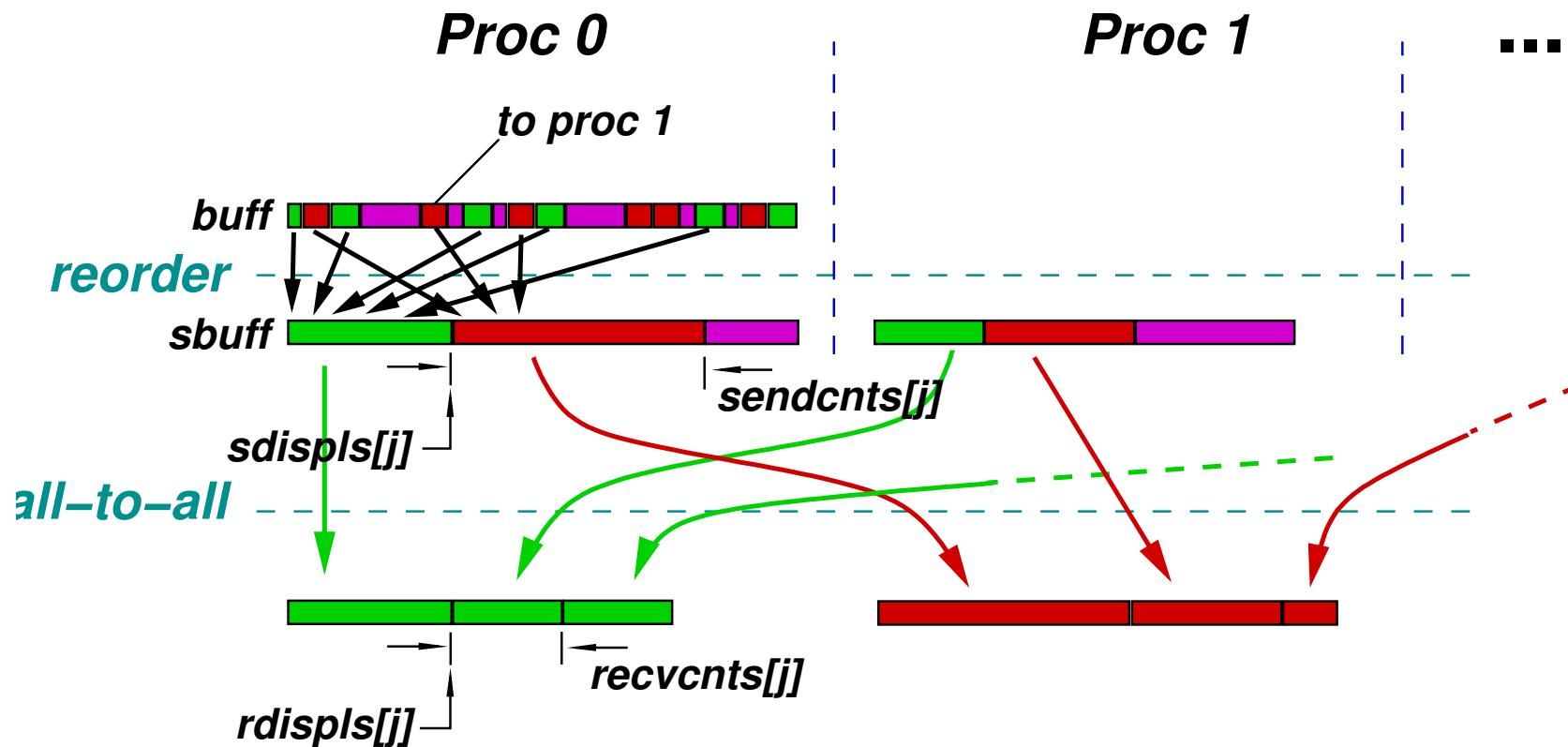
- Filtrar una lista (dejar sólo los objetos que satisfacen un predicado):

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

Por ejemplo, eliminar los elementos impares

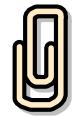
```
1. int odd(int x) {
2. return x% 2 != 0;
3. }
4. // fill array 'a' with values ...
5. filter(a,odd);
```

## Ejemplo de uso de all-to-all-v rescatter (cont.)



## Ejemplo de uso de all-to-all-v rescatter (cont.)

Seudo-código:



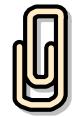
[Descargar: ./example/rescsc.cpp]

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

```
20. // Compute receive size 'rsize' and 'rdispls' from 'recvnts' ...
21. // resize 'buff' to 'rsize' ...
22.
23. MPI_Alltoallv(&sbuff[0],&sendcnts[0],&sdispls[0],MPI_INT,
24. &buff[0],&recvnts[0],&rdispls[0],MPI_INT,
25. MPI_COMM_WORLD);
26. }
```

## Ejemplo de uso de all-to-all-v rescatter (cont.)

Código completo:



[Descargar: ./example/rescatter.cpp]

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

```
20.
21. // Reorder by processor from buff to sbuff
22. for (int j=0; j<N; j++) {
23. int p = proc(buff[j],size,data);
24. int pos = sdispls[p];
25. sbuff[pos] = buff[j];
26. sdispls[p]++;
27. }
28. // Use 'Alltoall' for scattering the dimensions
29. // of the buffers to be received
30. MPI_Alltoall(&sendcnts[0],1,MPI_INT,
31. &recvcnts[0],1,MPI_INT,MPI_COMM_WORLD);
32.
33. // Compute the 'send' and 'recv' displacements.
34. rdispls[0] = 0;
35. sdispls[0] = 0;
36. for (int j=1; j<size; j++) {
37. rdispls[j] = rdispls[j-1] + recvcnts[j-1];
38. sdispls[j] = sdispls[j-1] + sendcnts[j-1];
39. }
40.
41. // Dimension the receive size
42. int rsize = 0;
43. for (int j=0; j<size; j++) rsize += recvcnts[j];
44. buff.resize(rsize);
45.
46. // Do the scatter.
47. MPI_Alltoallv(&sbuff[0],&sendcnts[0],&sdispls[0],MPI_INT,
48. &buff[0],&recvcnts[0],&rdispls[0],MPI_INT,
```

```
49. MPI_COMM_WORLD);
50. }
```

# Definiendo tipos de datos derivados

## Definiendo tipos de datos derivados

Supongamos que tenemos los coeficientes de una matriz de  $N \times N$  almacenados en un arreglo `double a[N*N]`. La fila  $j$  consiste en  $N$  dobles que ocupan las posiciones  $[j*N, (j+1)*N)$ . en el arreglo `a`.

Si queremos enviar la fila  $j$  de un procesador `source` a otro `dest` para reemplazar la fila  $k$ , entonces podemos usar `MPI_Send()` y `MPI_Recv()` ya que las filas representan valores contiguos

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

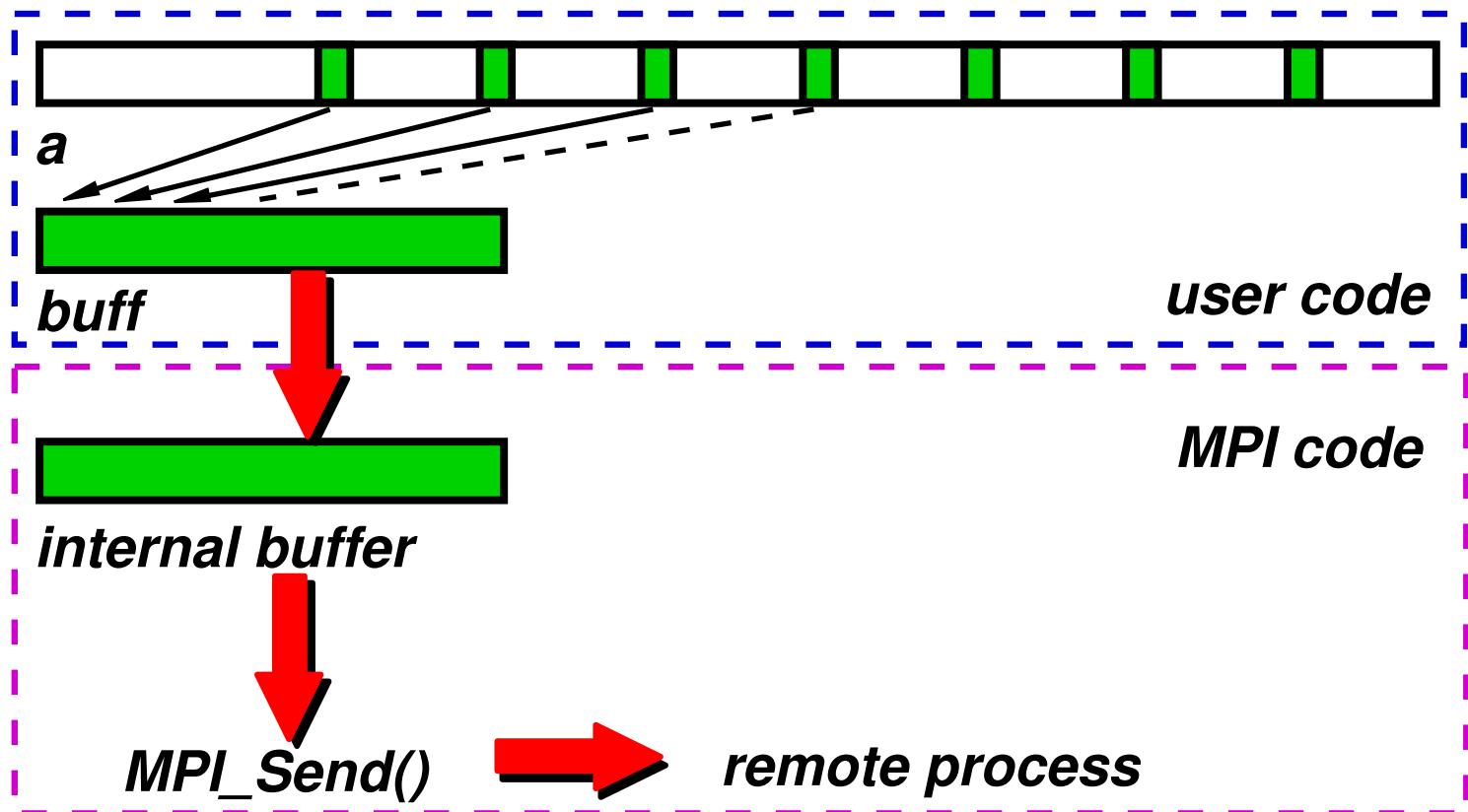
## Definiendo tipos de datos derivados (cont.)

Si queremos enviar columnas entonces el problema es más complicado, ya que, como en C++ los coeficientes son guardados por fila, los elementos de una columnas estan separados de a cada N elementos. La posibilidad más obvia es crear un buffer temporal `double buff[N]` donde juntar los elementos y después mandarlos/recibirlos.

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

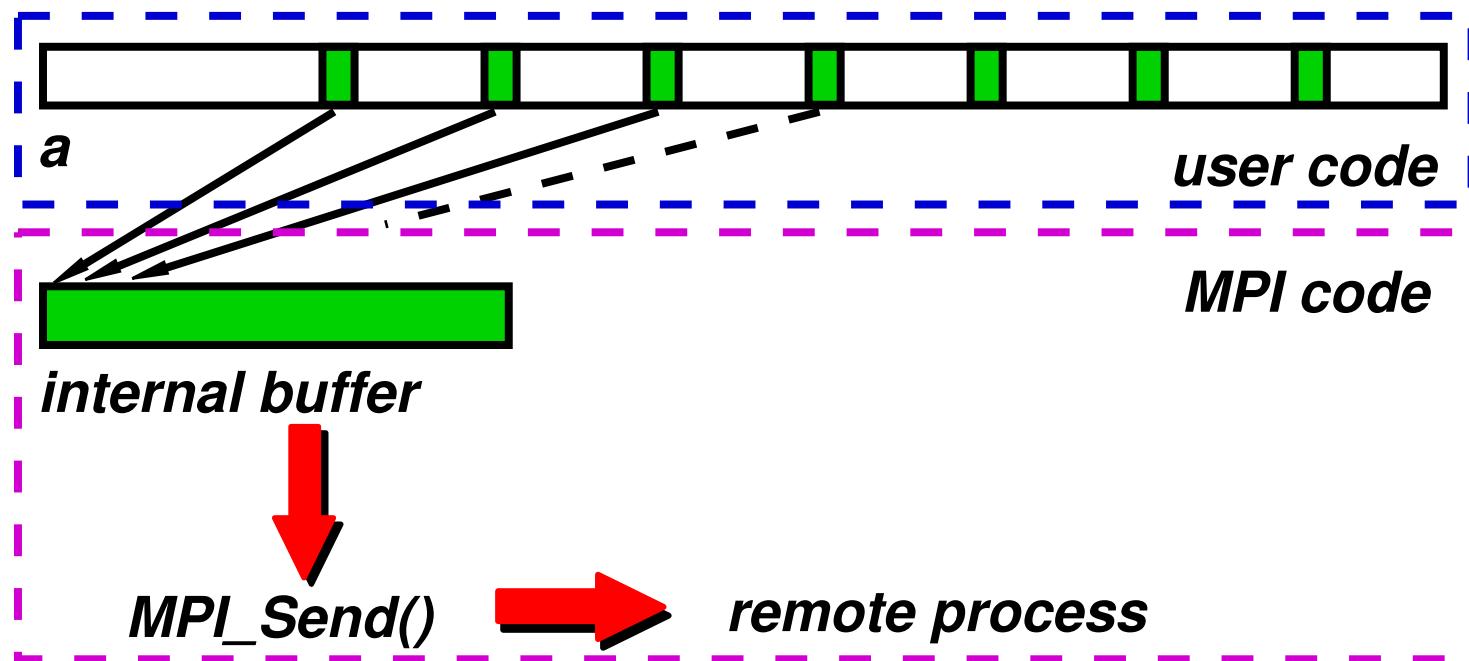
## Definiendo tipos de datos derivados (cont.)

Esto tiene el problema de que requiere un buffer adicional del tamaño de los datos a enviar, además del overhead en tiempo asociado en copiar estos datos al buffer. Además, es de notar que muy probablemente MPI vuelve a copiar el buffer **buff** en un buffer interno de MPI que es luego enviado.



## Definiendo tipos de datos derivados (cont.)

Vemos que hay una cierta duplicación de tareas y nos preguntamos si podemos evitar el usar **buff** pasándole directamente a MPI las direcciones donde están los elementos a ser enviados para que el arme el buffer interno directamente de **a**.



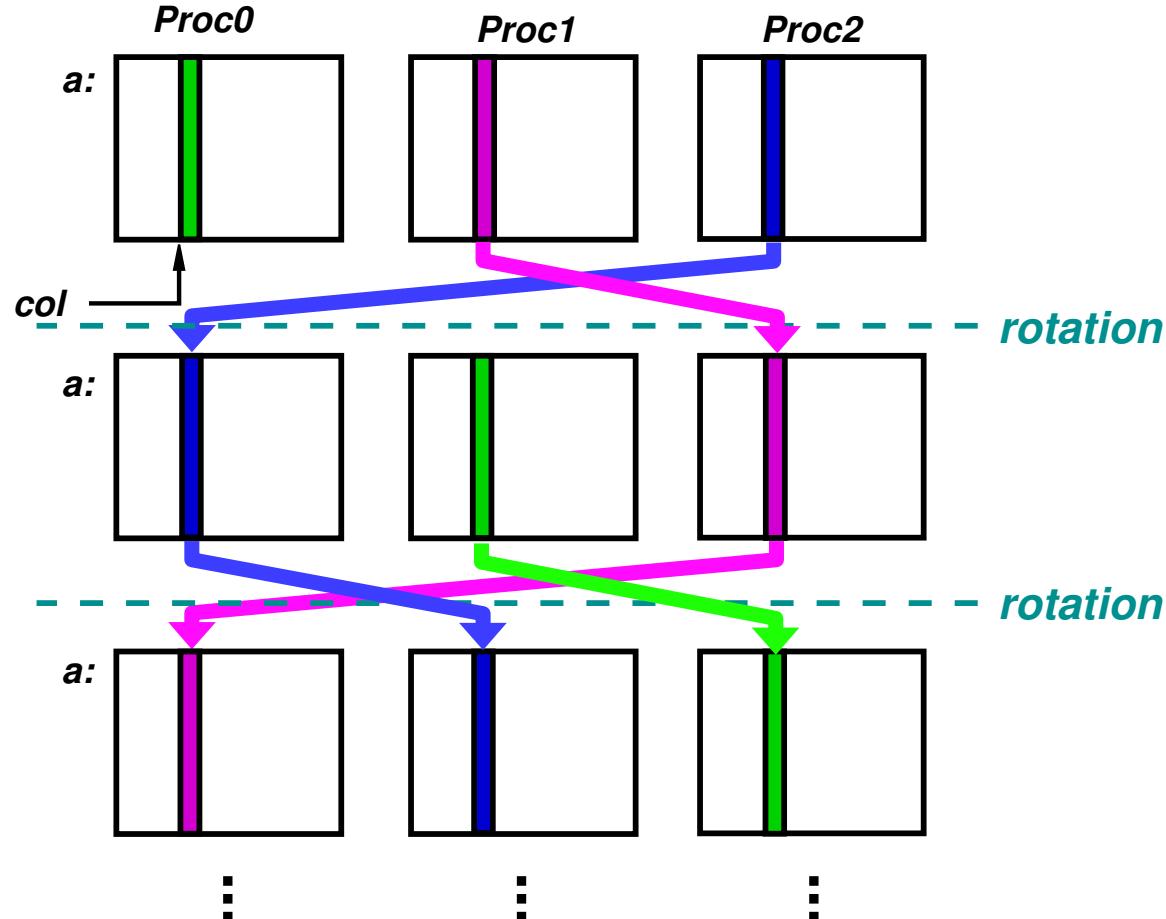
## Definiendo tipos de datos derivados (cont.)

La forma de hacerlo es definiendo un *tipo de datos derivado de MPI*.

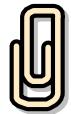
1. `MPI_Datatype stride;`
2. `MPI_Type_vector(N,1,N,MPI_DOUBLE,&stride);`
3. `MPI_Type_commit(&stride);`
- 4.
5. `// use 'stride' ...`
- 6.
7. `// Free resources reserved for type 'stride'`
8. `MPI_Type_free(&stride);`

## Ejemplo: rotar columna de una matriz

El siguiente programa rota la columna  $col=2$  de  $A$  entre los procesadores.



## Ejemplo: rotar columna de una matriz (cont.)



[Descargar: ./example/mpitypes.cpp]

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

8.
9. MPI_Datatype stride;
10. MPI_Type_vector(N,1,N,MPI_DOUBLE,&stride);
11. MPI_Type_commit(&stride);
12.
13. for (int k=0; k<10; k++) {
14. MPI_Sendrecv(&A(0,col),1,stride,dest,0,
15. &recv_val[0],N,MPI_DOUBLE,source,0,
16. MPI_COMM_WORLD,&status);
17. for (int j=0; j<N; j++)
18. A(j,col) = recv_val[j];
19. if (!myrank) {
20. printf("After rotation step%d\n",k);
```

```
21. print_mat(a,N);
22. }
23. }
24. MPI_Type_free(&stride);
25. MPI_Finalize();
```

## Ejemplo: rotar columna de una matriz (cont.)

Antes de rotar

1. Before rotation

2. Proc [0] a:

3. 4.8 3.2 [0.2] 9.9  
4. 7.9 8.8 [1.7] 7.2  
5. 2.7 4.2 [2.0] 4.1  
6. 7.1 0.8 [9.2] 9.4

Proc [1] a:

7.2 3.3 [9.9] 1.7  
8.9 2.2 [9.8] 2.3  
6.7 8.5 [5.7] 3.0  
6.0 1.9 [8.2] 7.9

Proc [2] a:

3.9 5.6 [9.5] 5.5  
5.1 3.3 [8.9] 1.2  
3.0 5.2 [6.1] 8.4  
7.0 5.5 [6.1] 1.9

En Proc 0, despues de aplicar rotaciones:

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

## GTP 6. [ORDSCAT] Ord-Scatterer

Escribir una función `ord_scat(vector<int> &sbuff, vector<int> &rbuf)` que, dados los vectores `sbuff`, redistribuye los elementos en `rbuf` de manera que todos los elementos en `sbuff` en el rango `[(rank*N)/size, ((rank+1)*N)/size]` quedan en el procesador `rank`, donde `N` es el mayor elemento de todos los `sbuff` más uno, es decir `N=max(SBUFF)+1`. `SBUFF` es la concatenación de todos los `sbuff`. Pueden asumir que los elementos de `sbuff` están ordenados en cada procesador.

Nota: OJO al hacer las cuentas de los rangos hay que usar paréntesis para que las divisiones enteras se hagan correctamente, es decir si el rango a procesar aquí es `[jstart, jend]` entonces se calculan así

1. `jstart = (N*rank)/size;`
2. `jend = (N*(rank+1))/size;`

Por ejemplo, si tenemos `NP=4` y los `sbuff`:

- En `P0`: `sbuff=[16, 17, 18, 19]`
- En `P1`: `sbuff=[2]`
- En `P2`: `sbuff=[1]`
- En `P3`: `sbuff=[0]`

Entonces tenemos  $N=20$  y los rangos que van a cada procesador son

- P0: [0, 5)
- P1: [5, 10)
- P2: [10, 15)
- P3: [15, 20)

Y entonces debe quedar después de hacer `ord_scat(sbuff, rbuff)`

- En P0: `rbuff=[0, 1, 2]`
- En P1: `rbuff=[]`
- En P2: `rbuff=[]`
- En P3: `rbuff=[16, 17, 18, 19]`

Ayuda:

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

## GTP 7. [RICH] Richardson

### Ejemplo: El método iterativo de Richardson

El método iterativo de Richardson para resolver un sistema lineal  $Ax = b$  se basa en el siguiente esquema de iteración:

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

La matriz sparse  $A$  está guardada en formato sparse, es decir tenemos vectores  $AIJ[nc]$ ,  $II[nc]$ ,  $JJ[nc]$ , de manera que para cada  $k$ , tal que  $0 \leq k < nc$ ,  $II[k]$ ,  $JJ[k]$ ,  $AIJ[k]$  son el índice de fila, de columnas y el coeficiente de  $A$ .

Por ejemplo, si  $A=[0 \ 1 \ 0; 2 \ 3 \ 0; 1 \ 0 \ 0]$ , entonces los vectores serían

$II=[0 \ 1 \ 1 \ 2]$ ,  $JJ=[1 \ 0 \ 1 \ 0]$ ,  $AIJ=[1 \ 2 \ 3 \ 1]$ .

El vector residuo  $r$  se puede calcular fácilmente haciendo un lazo sobre los elementos de la matriz

1. `for (int i=0; i<n; i++) r[i] = 0;`
2. `for (int k=0; k<nc; k++) {`
3.   `int i = II[k];`

```
4. int j = JJ[k];
5. int a = AIJ[k];
6. r[i] -= a * x[j];
7. }
8. for (int i=0; i<n; i++) r[i] += b[i];
```

El seudocódigo para el método de Richardson sería entonces

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

Proponemos la siguiente implementación en paralelo usando MPI,

- El master lee **A** y **B** de un archivo. Para eso se provee una rutina

```
1. void read_matrix(const char *file,
2. vector<int> &I, vector<int> &J, vector<double> &AIJ,
3. vector<double> &B, vector<double> &X);
```

- El master hace un broadcast del RHS a los nodos.
- Un rango de filas es asignado a cada uno de los procs. Usaremos el criterio de que cada procesador use una cantidad de filas tal que el

número de coeficientes correspondientes sea lo más balanceado posible. Es decir que el número de coeficientes sea lo más cercano posible a `ncmax/size` donde `ncmax` es el número de coeficientes no nulos, es decir la longitud de los vectores `I`, `J`, y `AIJ`.

- Por ejemplo, si `n=10` y el número de coefs no nulos en cada fila es `ncoefs=[1 1 1 3 3 1 1 1 3 3]` entonces `nc=sum(ncoefs)=18`. Si la cantidad de procs es `size=6` entonces cada proc debería recibir 3 coefs, y entonces la cantidad de filas que debe recibir cada proc es `nrows=[3 1 1 3 1 1]`.
- Este caso es simple ya que los 3 coefs se obtienen con un número entero de filas. Si `size=4` entonces a cada proc le corresponderían `z=nc/p=4.5` coefs, es decir el proc `p` debería procesar los coeficientes en el rango `[z*p, z*(p+1))`. Notar que como `z` no es entero tomamos el `ceil()` es decir al proc `p=0` le corresponde los coefs `[0, 5)` al `p=1` le corresponde los coefs `[5, 9)`.
- Ahora bien queremos que todos los coeficientes de una fila estén en el mismo procesador, de manera que calculamos las posiciones donde empiezan los coeficientes de cada fila  
`rowstart=cumsum(ncoefs)=[0 1 2 3 6 9 10 11 12 15]` y buscamos para cada procesador la fila `rp[j]` para la cual el `rowstart(rp[j])>=z*p`. Entonces cada fila procesa las filas entre `[rp[j], rp[j+1))` Por lo tanto en el caso anterior los procs procesan las filas empezando en `rp=[0, 4, 5, 9]`. Es decir

el proc 0 procesa el rango  $[0, 4]$  el proc 1 procesa  $[4, 5]$  y así siguiendo.

- Cada procesador tiene en todo momento toda la matriz y vector, pero calcula sólo una parte del residuo (aquellas filas  $i$  que están en el rango seleccionado).
- Una vez que cada uno calculó su parte, todos deben intercambiar sus contribuciones usando `MPI_Allgatherv`.
- El factor de relajación  $\omega$ . Las siguientes condiciones son suficientes para que el esquema sea convergente: 1) que  $A$  sea simétrica y definida positiva, y 2) que  $\omega < 2/\lambda_{\max}$ . Para no tener que calcular el autovalor máximo  $\lambda_{\max}$  puede usarse la cota  $\lambda_{\max} < \|A\|_1$  (norma  $L_1$  de  $A$ ).



**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 Satisf Balay, William Gropp y otros.

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

## PETSc (cont.)

- 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

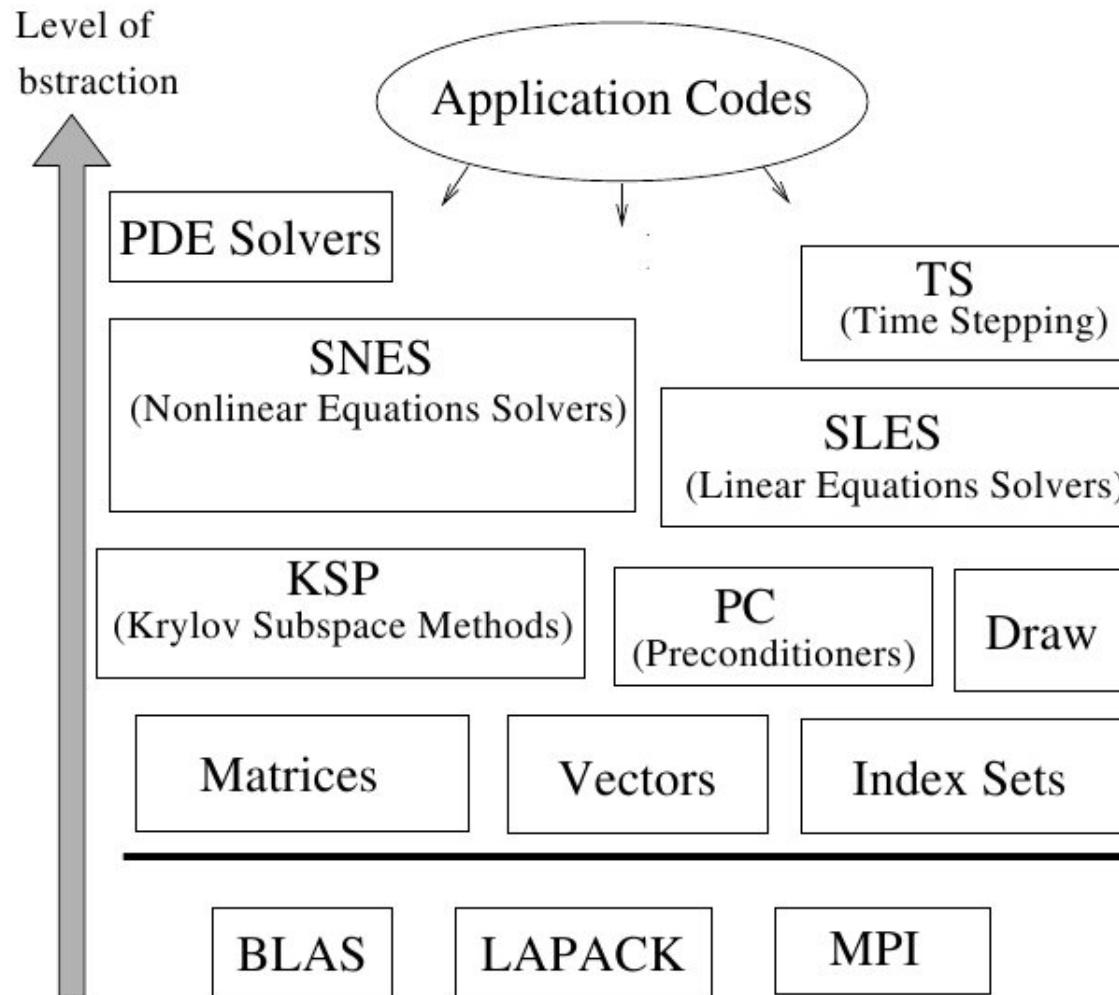
## Objetos PETSc

- 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



# Usando PETSc

## Usando PETSc

Agregar las siguientes variables de entorno

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

## Usando PETSc (cont.)

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

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

Por ejemplo

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

## Escribiendo programas que usan PETSc

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

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

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

## Escribiendo programas que usan PETSc (cont.)

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

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

## Ejemplo simple. Ec. de Laplace 1D

Resolver  $Ax = b$ , donde

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

## Ejemplo simple. Ec. de Laplace 1D (cont.)

Declarar variables (vectores y matrices) PETSc (inicialmente son *punteros*)

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

**Crear** los objetos

1. ierr = VecCreate(PETSC\_COMM\_WORLD,&x);CHKERRQ(ierr);
2. ierr = MatCreate(PETSC\_COMM\_WORLD,PETSC\_DECIDE,  
PETSC\_DECIDE,n,n,&A);

En general

1. PetscType object;
2. ierr = PetscTypeCreate(PETSC\_COMM\_WORLD,...options...,&object);
3. // use 'object'...
4. ierr = PetscTypeDestroy(x);CHKERRQ(ierr);

PetscType puede ser Vec, Mat, PC, KSP ....

## Ejemplo simple. Ec. de Laplace 1D (cont.)

Se pueden *duplicar (clonar)* objetos

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

## Ejemplo simple. Ec. de Laplace 1D (cont.)

### Setear valores de vectores y matrices

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

También se pueden *ensamblar matrices por bloques*

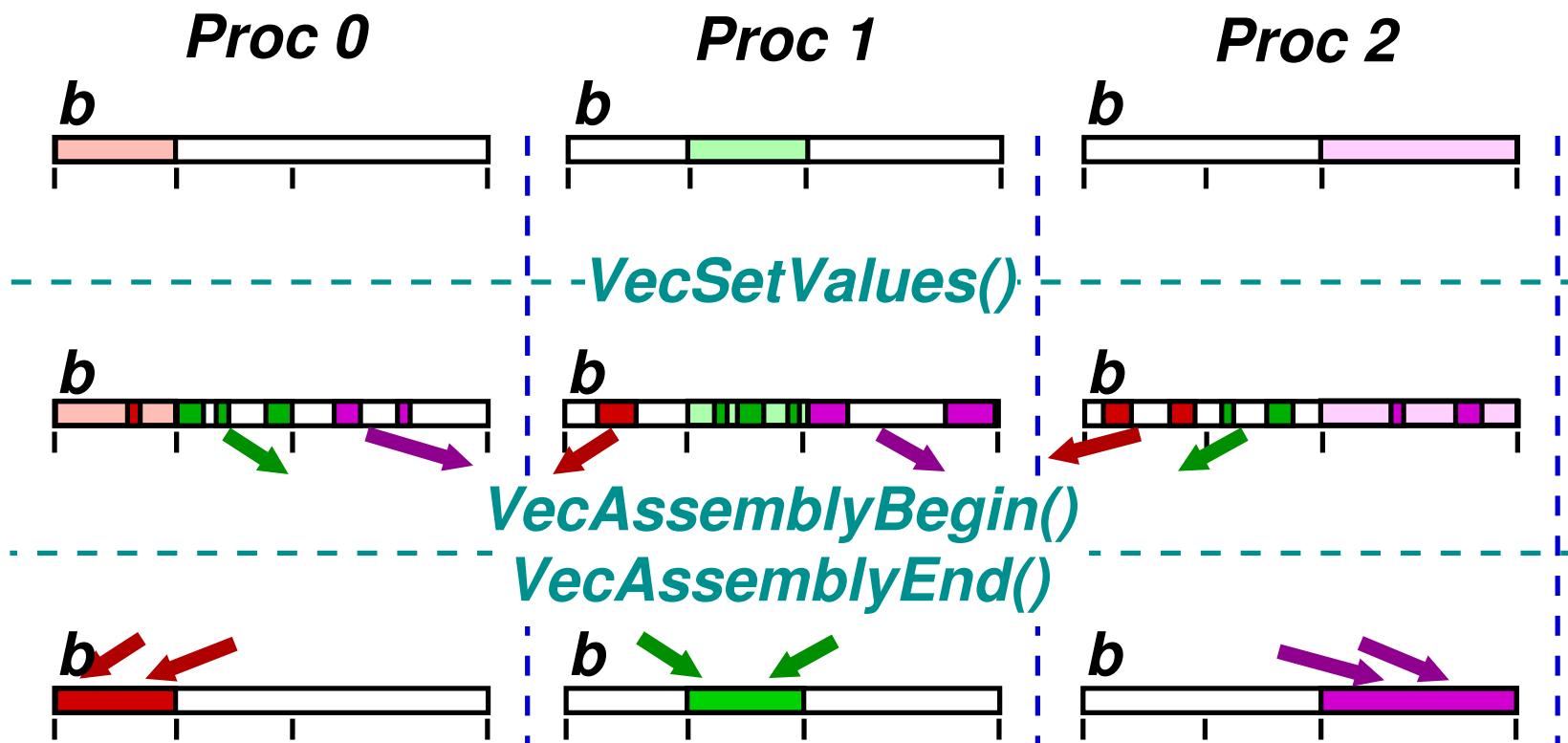
```
1. int nrow, *rows, ncols, *cols;
2. double *vals;
3. // Equivalent to Matlab: A(rows,cols) = vals;
4. MatSetValues(A,nrows,rows,ncols,cols,vals,INSERT_VALUES);
```

## Ejemplo simple. Ec. de Laplace 1D (cont.)

Despues de hacer `...SetValues(...)` hay que hacer `...Assembly(...)`

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

## Ejemplo simple. Ec. de Laplace 1D (cont.)



## Ejemplo simple. Ec. de Laplace 1D (cont.)

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

## Ejemplo simple. Ec. de Laplace 1D (cont.)

### Uso de KSP (*Krylov Space Scalable Linear Equations Solvers*)

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

## Ejemplo simple. Ec. de Laplace 1D (cont.)



[Descargar: ./example/ex1.cpp]

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

```
23. PetscInitialize(&argc,&args,(char *)0,help);
24. ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
25. if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
26. ierr = PetscOptionsGetInt(PETSC_NULL,"-n",
27. &n,PETSC_NULL);CHKERRQ(ierr);
28.
29. /* -----
30. Compute the matrix and right-hand-side vector that define
31. the linear system, Ax = b.
32. ----- */
33.
34. /*
35. Create vectors. Note that we form 1 vector from scratch and
36. then duplicate as needed.
37. */
38. ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
39. ierr = PetscObjectSetName((PetscObject)x,
40. "Solution");CHKERRQ(ierr);
41. ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
42. ierr = VecSetFromOptions(x);CHKERRQ(ierr);
43. ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
44. ierr = VecDuplicate(x,&u);CHKERRQ(ierr);
45.
46. /*
47. Create matrix. When using MatCreate(), the matrix
48. format can be specified at runtime.
49.
50. Performance tuning note: For problems of substantial
51. size, preallocation of matrix memory is crucial for
```

```
52. attaining good performance. See the matrix chapter of
53. the users manual for details.
54. */
55. ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
56. ierr = MatSetSizes(A,PETSC_DECIDE,
57. PETSC_DECIDE,n,n); CHKERRQ(ierr);
58. ierr = MatSetFromOptions(A);CHKERRQ(ierr);
59.
60. /*
61. Assemble matrix
62. */
63. value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
64. for (i=1; i<n-1; i++) {
65. col[0] = i-1; col[1] = i; col[2] = i+1;
66. ierr = MatSetValues(A,1,&i,3,col,
67. value,INSERT_VALUES);CHKERRQ(ierr);
68. }
69. i = n - 1; col[0] = n - 2; col[1] = n - 1;
70. ierr = MatSetValues(A,1,&i,2,col,
71. value,INSERT_VALUES);CHKERRQ(ierr);
72. i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
73. ierr = MatSetValues(A,1,&i,2,col,
74. value,INSERT_VALUES);CHKERRQ(ierr);
75. ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
76. ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
77.
78. /*
79. Set exact solution; then compute right-hand-side vector.
```

```
80. */
81. ierr = VecSet(u,one);CHKERRQ(ierr);
82. ierr = MatMult(A,u,b);CHKERRQ(ierr);
83.
84. /* ----- Create the linear solver and set various options -----
85. Create linear solver context
86. */
87. /*
88. Set operators. Here the matrix that defines the linear
89. system also serves as the preconditioning matrix.
90. */
91. /*
92. Set linear solver defaults for this problem (optional).
93. - By extracting the KSP and PC contexts from the KSP
94. context, we can then directly call any KSP and PC
95. routines to set various options.
96. - The following four statements are optional; all of
97. these parameters could alternatively be specified at
98. runtime via KSPSetFromOptions();
99. */
100. ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);
101. CHKERRQ(ierr);
102.
103. /*
104. Set linear solver defaults for this problem (optional).
105. - By extracting the KSP and PC contexts from the KSP
106. context, we can then directly call any KSP and PC
107. routines to set various options.
108. - The following four statements are optional; all of
109. these parameters could alternatively be specified at
 runtime via KSPSetFromOptions();
*/
```

```

110. ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,
111. PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);
112.
113. /*
114. Set runtime options, e.g.,
115. -ksp_type <type> -pc_type <type>
116. -ksp_monitor -ksp_rtol <rtol>
117. These options will override those specified above as
118. long as KSPSetFromOptions() is called after any other
119. customization routines.
120. */
121. ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);
122.
123. /*
124. Solve the linear system
125. -----
126. ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
127.
128. /*
129. View solver info; we could instead use the option
130. -ksp_view to print this info to the screen at the
131. conclusion of KSPSolve().
132. */
133. ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
134.
135. /*
136. Check solution and clean up
137. -----
138. */

```

```
139. Check the error
140. */
141. ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
142. ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
143. ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
144. ierr = PetscPrintf(PETSC_COMM_WORLD,
145. "Norm of error %.2g, Iterations %d\n",
146. norm,its);CHKERRQ(ierr);
147. /*
148. Free work space. All PETSc objects should be destroyed
149. when they are no longer needed.
150. */
151. ierr = VecDestroy(x);CHKERRQ(ierr);
152. ierr = VecDestroy(u);CHKERRQ(ierr);
153. ierr = VecDestroy(b);CHKERRQ(ierr);
154. ierr = MatDestroy(A);CHKERRQ(ierr);
155. ierr = KSPDestroy(ksp);CHKERRQ(ierr);
156.
157. /*
158.
159. Always call PetscFinalize() before exiting a program.
160. This routine
161. - finalizes the PETSc libraries as well as MPI
162. - provides summary and diagnostic information if
163. certain runtime options are chosen (e.g.,
164. -log_summary).
165. */
166. ierr = PetscFinalize();CHKERRQ(ierr);
167. return 0;
```

168. }

# Elementos de PETSc

## Headers

```
1 #include "petscsksp.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. `petscsksp.h` (Krylov space 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

1. `PetscOptionsGetInt(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,`  
`PetscScalar *values,INSERT_VALUES);`

- **m = (optional) local (puede ser PETSC\_DECIDE)**
- **M tamaño global**
- **VecsetType(), VecSetFromOptions() permiten definir el tipo de almacenamiento.**

## Matrices

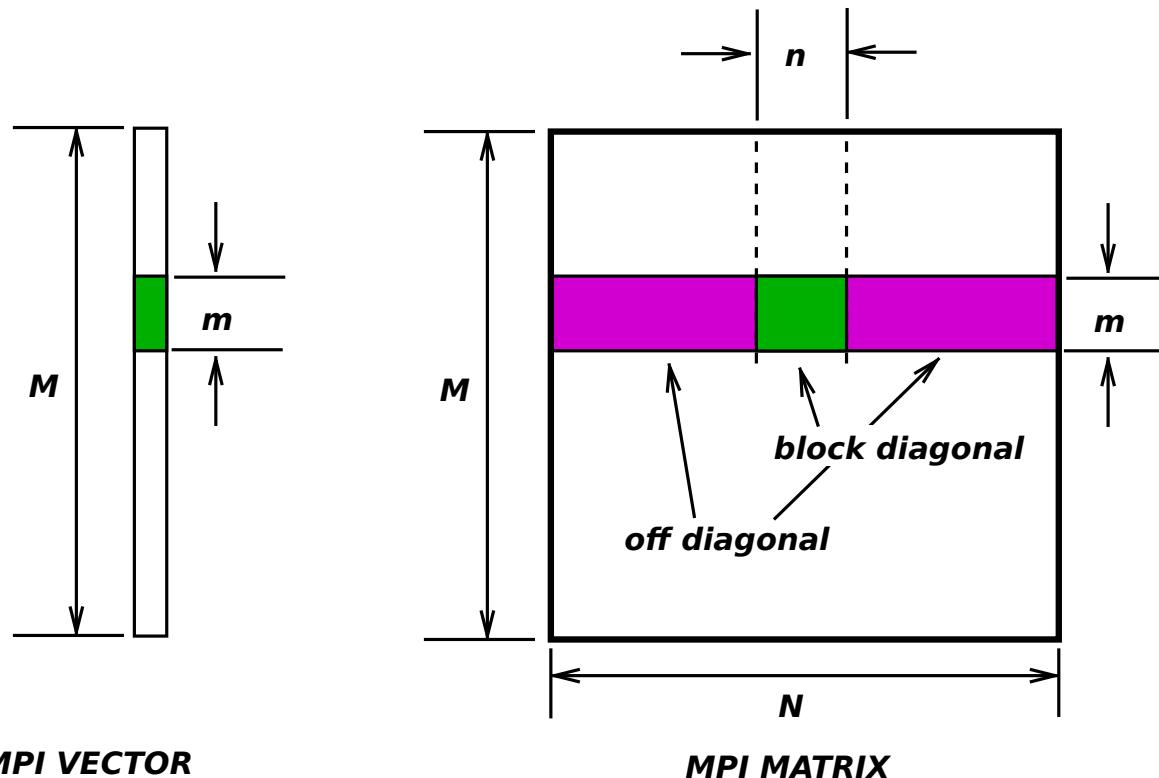
```
1. // Create a matrix
2. MatCreate(MPI_Comm comm ,int m,
3. int n,int M ,int N,Mat *A);
4.
5. // Set values in rows 'im' and columns 'in'
6. MatSetValues(Mat A,int m,int *im,int n,int *in,
7. PetscScalar *values,INSERT_VALUES);
8.
9. // Do assembly BEFORE using the matrices
10. // (values may be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

## Linear solvers

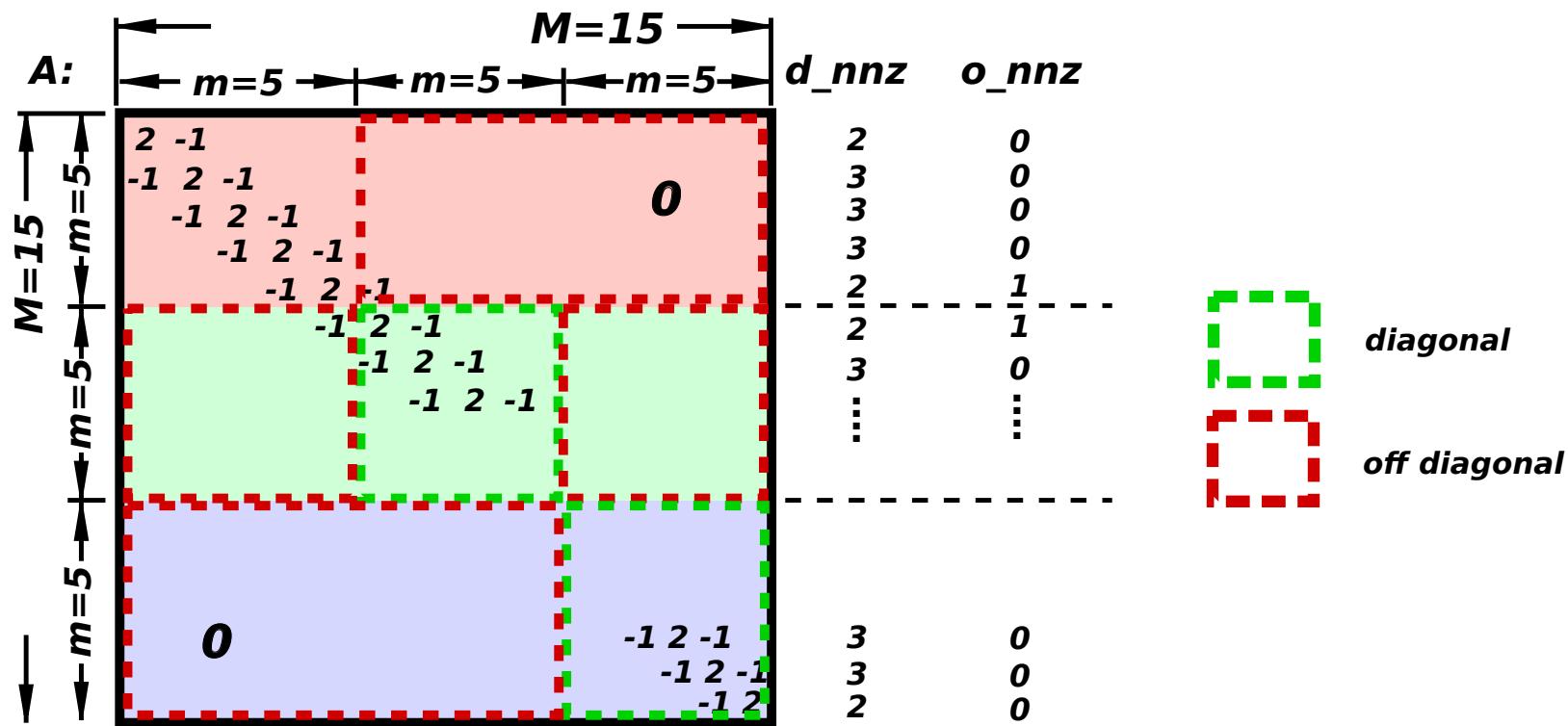
```
1. // Create the KSP
2. KSPCreate(MPI_Comm comm ,KSP *ksp);
3.
4. // Set matrix and preconditioning
5. KSPSetOperators (KSP ksp,Mat A,
6. Mat PrecA,MatStructure flag);
7.
8. // Use options from data base (solver type?)
9. // (file/env/command)
10. KSPSetFromOptions (KSP ksp);
11.
12. // Solve linear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

## Programación en paralelo

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



## Programación en paralelo (cont.)



## Programación en paralelo (cont.)



[Descargar: ./example/ex23.cpp]

```
1. static char help[] = "Solves a tridiagonal linear system.\n\n";
2. #include "petscksp.h"
3.
4. #undef __FUNCT__
5. #define __FUNCT__ "main"
6. int main(int argc,char **args)
7. {
8. Vec x, b, u; /* approx solution, RHS,
9. exact solution */
10. Mat A; /* linear system matrix */
11. KSP ksp; /* linear solver context */
12. PC pc; /* preconditioner context */
13. PetscReal norm; /* norm of solution error */
14. PetscErrorCode ierr;
15. PetscInt i,n = 10,col[3],its,rstart,rend,nlocal;
16. PetscScalar neg_one = -1.0,one = 1.0,value[3];
17.
18. PetscInitialize(&argc,&args,(char *)0,help);
19. ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
20. CHKERRQ(ierr);
21.
```

```
22. /* -----
23. Compute the matrix and right-hand-side vector that
24. define the linear system, Ax = b.
25. ----- */
26.
27. /* Create vectors. Note that we form 1 vector from
28. scratch and then duplicate as needed. For this simple
29. case let PETSc decide how many elements of the vector
30. are stored on each processor. The second argument to
31. VecSetSizes() below causes PETSc to decide. */
32. ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
33. ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
34. ierr = VecSetFromOptions(x);CHKERRQ(ierr);
35. ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
36. ierr = VecDuplicate(x,&u);CHKERRQ(ierr);
37.
38. /* Identify the starting and ending mesh points on each
39. processor for the interior part of the mesh. We let
40. PETSc decide above. */
41.
42. ierr = VecGetOwnershipRange(x,&rstart,&rend);CHKERRQ(ierr);
43. ierr = VecGetLocalSize(x,&nlocal);CHKERRQ(ierr);
44.
45. /* Create matrix. When using MatCreate(), the matrix
46. format can be specified at runtime.
47.
48. Performance tuning note: For problems of substantial
49. size, preallocation of matrix memory is crucial for
50. attaining good performance. See the matrix chapter of
51. the users manual for details.
```

```

52.
53. We pass in nlocal as the “local” size of the matrix to
54. force it to have the same parallel layout as the vector
55. created above. */
56. ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
57. ierr = MatSetSizes(A,nlocal,nlocal,n,n);CHKERRQ(ierr);
58. ierr = MatSetFromOptions(A);CHKERRQ(ierr);
59.
60. /* Assemble matrix.
61. The linear system is distributed across the processors
62. by chunks of contiguous rows, which correspond to
63. contiguous sections of the mesh on which the problem is
64. discretized. For matrix assembly, each processor
65. contributes entries for the part that it owns locally.*/
66. if (!rstart) {
67. rstart = 1;
68. i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
69. ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);
70. CHKERRQ(ierr);
71. }
72. if (rend == n) {
73. rend = n-1;
74. i = n-1; col[0] = n-2; col[1] = n-1;
75. value[0] = -1.0; value[1] = 2.0;
76. ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);
77. CHKERRQ(ierr);
78. }
79.
80. /* Set entries corresponding to the mesh interior */

```

```

81. value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
82. for (i=rstart; i<rend; i++) {
83. col[0] = i-1; col[1] = i; col[2] = i+1;
84. ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);
85. CHKERRQ(ierr);
86. }
87.
88. /* Assemble the matrix */
89. ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
90. ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
91.
92. /* Set exact solution; then compute right-hand-side vector. */
93. ierr = VecSet(u,one);CHKERRQ(ierr);
94. ierr = MatMult(A,u,b);CHKERRQ(ierr);
95.
96. /* -----
97. Create the linear solver and set various options
98. ----- */
99. ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
100.
101. /* Set operators. Here the matrix that defines the linear system
102. also serves as the preconditioning matrix. */
103. ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);
104. CHKERRQ(ierr);
105.
106. /*
107. Set linear solver defaults for this problem (optional).
108. - By extracting the KSP and PC contexts from the KSP context,
109. we can then directly call any KSP and PC routines to set

```

```

110. various options.
111. - The following four statements are optional; all of these
112. parameters could alternatively be specified at runtime via
113. KSPSetFromOptions();
114. */
115. ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
116. ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
117. ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,
118. PETSC_DEFAULT);CHKERRQ(ierr);
119. /*
120. * Set runtime options, e.g.,
121. * -ksp_type <type> -pc_type <type>
122. * -ksp_monitor -ksp_rtol <rtol>
123. These options will override those specified above as
124. long as KSPSetFromOptions() is called _after_ any other
125. customization routines.
126. */
127. ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);
128. /*
129. * -----
130. Solve the linear system
131. * -----
132. */
133. /*
134. Solve linear system
135. */
136. ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
137. /*
138. */

```

```
139. View solver info; we could instead use the option
140. -ksp_view to print this info to the screen at the
141. conclusion of KSPSolve().
142. */
143. ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
144.
145. /* ----- *
146. Check solution and clean up
147. ----- */
148. /*
149. Check the error
150. */
151. ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
152. ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
153. ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
154. ierr = PetscPrintf(PETSC_COMM_WORLD,
155. "Norm of error %A, Iterations %D\n",
156. norm,its);CHKERRQ(ierr);
157. /*
158. Free work space. All PETSc objects should be destroyed
159. when they are no longer needed.
160. */
161. ierr = VecDestroy(x);CHKERRQ(ierr);
162. ierr = VecDestroy(u);CHKERRQ(ierr);
163. ierr = VecDestroy(b);CHKERRQ(ierr);
164. ierr = MatDestroy(A);CHKERRQ(ierr);
165. ierr = KSPDestroy(ksp);CHKERRQ(ierr);
166.
167. /*
```

```
168. Always call PetscFinalize() before exiting a program.
169. This routine
170. - finalizes the PETSc libraries as well as MPI
171. - provides summary and diagnostic information if
172. certain runtime options are chosen (e.g.,
173. -log_summary).
174. */
175. ierr = PetscFinalize();CHKERRQ(ierr);
176. return 0;
177. }
```

# **Programa de FEM basado en PETSc**

## Programa de FEM basado en PETSc

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

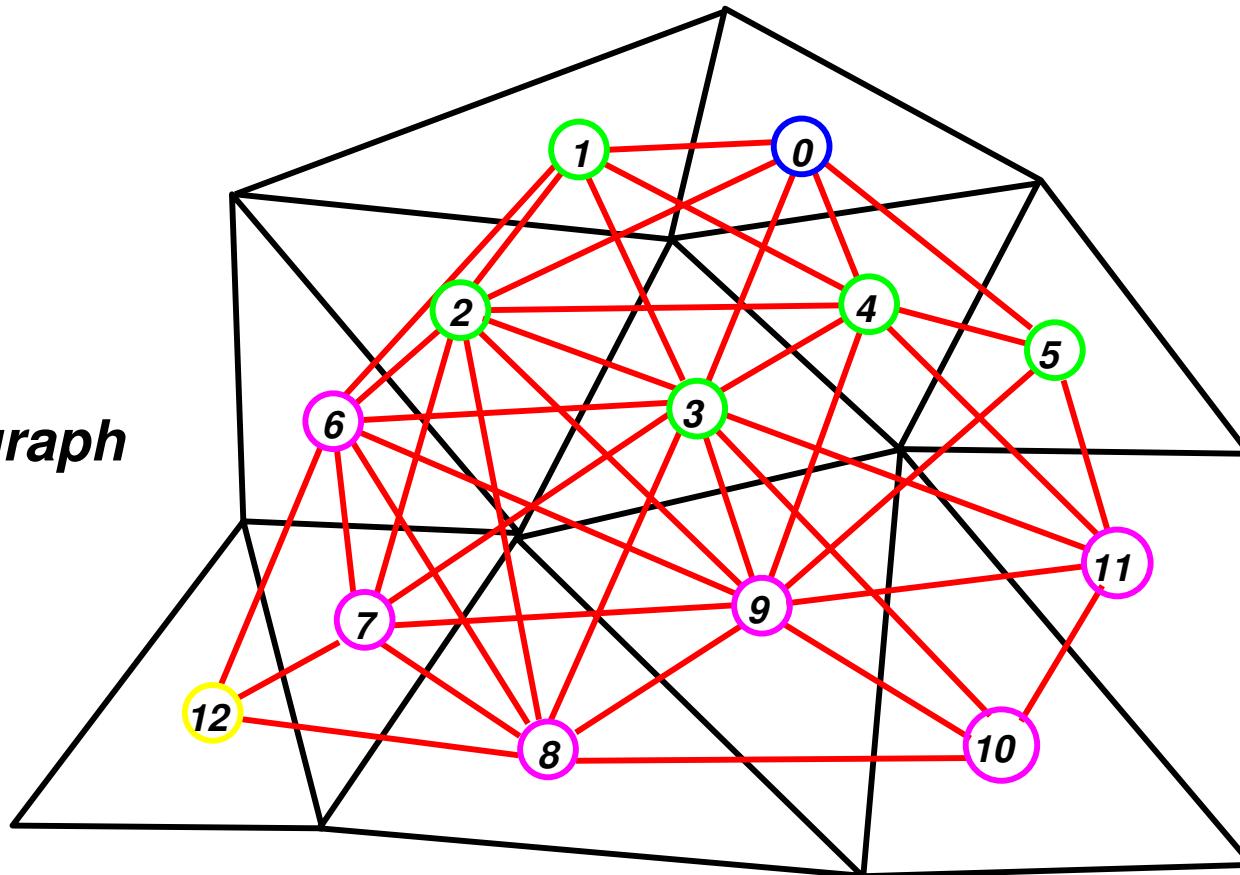
# Conceptos básicos de particionamiento

## Numerar grafo por Cuthill-Mc.Kee

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

## Numerar grafo por Cuthill-Mc.Kee (cont.)

*dual graph*



## Particionamiento de elementos

1. // nelem[proc] es el numero de elementos a poner en
2. // en el procesador 'proc'
3. int nproc;
4. int nelem[nproc];
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-1] elementos en proc. nproc-1

## Particionamiento de nodos

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

# Código FEM

## Programa de FEM basado en PETSc



[Descargar: ./example/femprog/fem.cpp]

```
1. #include <cstdio>
2. #include <cassert>
3.
4. #include <iostream>
5. #include <set>
6. #include <vector>
7. #include <deque>
8. #include <map>
9.
10. #include <petscksp.h>
11.
12. #include <newmatio.h>
13.
14. static char help[] =
15. "Basic test for self scheduling algorithm FEM with PETSc";
16.
17. using namespace std;
18.
19. //---<*>---<*>---<*>---<*>---<*>---
20. #undef __FUNC__
21. #define __FUNC__ "main"
22. int main(int argc,char **args) {
23. Vec b,u;
```

```
24. Mat A;
25. // SLES sles;
26. PC pc; /* preconditioner context */
27. KSP ksp; /* Krylov subspace method context */
28.
29. // Number of nodes per element
30. const int nel=3;
31. // Space dimension
32. const int ndim=2;
33. PetscInitialize(&argc,&args,(char *)0,help);
34.
35. // Get MPI info
36. int size,rank;
37. MPI_Comm_size(PETSC_COMM_WORLD,&size);
38. MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
39.
40. // read nodes
41. double x[ndim];
42. int read;
43. vector<double> xnod;
44. #define XNOD(j,k) xnod[j*ndim+k]
45. vector<int> icone;
46. FILE *fid = fopen("node.dat","r");
47. assert(fid);
48. while (true) {
49. for (int k=0; k<ndim; k++) {
50. read = fscanf(fid,"%lf",&x[k]);
51. if (read == EOF) break;
52. }
}
```

```
53. if (read == EOF) break;
54. xnod.push_back(x[0]);
55. xnod.push_back(x[1]);
56. }
57. fclose(fid);
58. int nnod=xnod.size()/ndim;
59. PetscPrintf(PETSC_COMM_WORLD,"Read %d nodes.\n",nnod);
60.
61. // read elements
62. int ix[nel];
63. fid = fopen("icone.dat","r");
64. assert(fid);
65. while (true) {
66. for (int k=0; k<nel; k++) {
67. read = fscanf(fid,"%d",&ix[k]);
68. if (read == EOF) break;
69. }
70. if (read == EOF) break;
71. icone.push_back(ix[0]);
72. icone.push_back(ix[1]);
73. icone.push_back(ix[2]);
74. }
75. fclose(fid);
76.
77. int nelem=icone.size()/nel;
78. PetscPrintf(PETSC_COMM_WORLD,"Read %d elements.\n",nelem);
79.
80. // read fixations stored as a map node -> value
```

```

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

```

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

```
139. if (eord[*k-1]==0) {
140. q.push_back(*k);
141. eord[*k-1]=-1;
142. }
143. }
144. }
145. }
146. q.clear();
147.
148. // Element partition. Put (approximatively)
149. // nelem/size in each processor
150. int *e_indx = new int[size+1];
151. e_indx[0]=1;
152. for (int p=0; p<size; p++)
153. e_indx[p+1] = e_indx[p]+(nelem/size) + (p<(nelem% size) ? 1 : 0);
154.
155. // Node partitioning. If a node is connected to an element 'j' then
156. // put it in the processor where element 'j' belongs.
157. // In case the elements connected to the node belong to
158. // different processors take any one of them.
159. int *id_indx = new int[size+2];
160. for (int j=0; j<nnod; j++) {
161. if (fixa.find(j+1) != fixa.end()) {
162. id[j]=-(size+1);
163. } else {
164. set<int> &e = n2e[j];
165. assert(e.size()>0);
166. int order = eord[*e.begin()-1];
```

```
167. for (int p=0; p<size; p++) {
168. if (order >= e_indx[p] && order < e_indx[p+1]) {
169. id[j] = -(p+1);
170. break;
171. }
172. }
173. }
174.
175.
176. // 'id_indx[j-1]' is the dof associated to node 'j'
177. int dof=0;
178. id_indx[0]=0;
179. if (size>1)
180. PetscPrintf(PETSC_COMM_WORLD,
181. "dof distribution among processors\n");
182. for (int p=0; p<=size; p++) {
183. for (int j=0; j<nnod; j++)
184. if (id[j]==-(p+1)) id[j]=dof++;
185. id_indx[p+1] = dof;
186. if (p<size) {
187. PetscPrintf(PETSC_COMM_WORLD,
188. "proc: %d, from %d to %d, total %d\n",
189. p,id_indx[p],id_indx[p+1],id_indx[p+1]-id_indx[p]);
190. } else {
191. PetscPrintf(PETSC_COMM_WORLD,
192. "fixed: from %d to %d, total %d\n",
193. id_indx[p],id_indx[p+1],id_indx[p+1]-id_indx[p]);
```

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

```

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

```

```

250. gN(2) = -x13(1);
251. grad_N.Column(2) = gN/(2*area);
252. // last gradient can be computed from \sum_j grad_N_j = 0
253. grad_N.Column(1) = -(grad_N.Column(2)+grad_N.Column(3));
254. // Integrand is constant over element
255. ke = grad_N.t() * grad_N * area;
256.
257. // Load matrix element on global matrix
258. int nod1,nod2,eq1,eq2;
259. for (int i1=1; i1<=nel; i1++) {
260. nod1 = icone[(e-1)*nel+i1-1];
261. if (fixa.find(nod1)!=fixa.end()) continue;
262. eq1=id[nod1-1];
263. for (int i2=1; i2<=nel; i2++) {
264. nod2 = icone[(e-1)*nel+i2-1];
265. eq2=id[nod2-1];
266. if (fixa.find(nod2)!=fixa.end()) {
267. // Fixed nodes contribute to the right hand side
268. VecSetValue(b,eq1,-ke(i1,i2)*fixa[nod2],ADD_VALUES);
269. } else {
270. // Load on the global matrix
271. MatSetValue(A,eq1,eq2,ke(i1,i2),ADD_VALUES);
272. }
273. }
274. }
275. }
276. }
277. // Finish assembly.

```

```

278. ierr = VecAssemblyBegin(b); CHKERRQ(ierr);
279. ierr = VecAssemblyEnd(b); CHKERRQ(ierr);
280.
281. ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
282. ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
283. }
284.
285. // Create SLES and set options
286. ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
287. ierr = KSPSetOperators(ksp,A,A,
288. DIFFERENT_NONZERO_PATTERN); CHKERRQ(ierr);
289. ierr = KSPGetPC(ksp,&pc); CHKERRQ(ierr);
290.
291. ierr = KSPSetType(ksp,KSPGMRES); CHKERRQ(ierr);
292. // This only works for one processor only
293. // ierr = PCSetType(pc,PCLU); CHKERRQ(ierr);
294. ierr = PCSetType(pc,PCJACOBI); CHKERRQ(ierr);
295. ierr = KSPSetTolerances(ksp,1e-6,1e-6,1e3,100); CHKERRQ(ierr);
296. int its;
297. ierr = KSPSolve(ksp,b,u); CHKERRQ(ierr);
298. ierr = KSPGetIterationNumber(ksp,&its); CHKERRQ(ierr);
299.
300. // Now we have to reorder the values in the vector of
301. // equations to the node numbering. First we gather all
302. // the values on the master using a "PETSc Scatter". A
303. // scatter reorders the values of a vector. In this case
304. // we use a special scatter already defined in PETSc that
305. // gathers all the parts of the vector in a local vector
306. // in the master.

```

```

307. //
308. // These are the resulting vector in the master and the
309. // scatter object. They are created at the same time by
310. // VecScatterCreateToZero()
311. Vec uloc;
312. VecScatter scat;
313. // Create the scatter and the target vector
314. VecScatterCreateToZero(u,&scat,&uloc);
315. // Do the scatter
316. VecScatterBegin(scat,u,uloc,INSERT_VALUES,SCATTER_FORWARD);
317. VecScatterEnd(scat,u,uloc,INSERT_VALUES,SCATTER_FORWARD);
318. // Now the ULOC vector has all the values in the master
319. // and has null length in the slaves.
320.
321. if (!rank) {
322. // Write the values to a plain text file
323. FILE *fido = fopen("result.dat","w");
324. double *ulocp;
325. ierr = VecGetArray(uloc,&ulocp); CHKERRQ(ierr);
326. for (int j=0; j<nnod; j++) {
327. // Get the equation number of the node
328. int jeq = id[j];
329. double val = NAN;
330. // If jeq<neq then it means that the nodes is free (not BC),
331. // so get the value from the vector of unknowns
332. if (jeq<neq) val= ulocp[jeq];
333. // else get the value from the map
334. else val = fixa[j+1];

```

```
335. // printf("node %d, jeq %d, x%f%f, val %f\n",
336. // j,id[j],XNOD(j,0),XNOD(j,1),val);
337. fprintf(fido,"%15.10g\n",val);
338. }
339. fclose(fido);
340. }
341.
342. // Destroy the objects
343. MatDestroy(&A);
344. VecDestroy(&b);
345. VecDestroy(&u);
346. VecScatterDestroy(&scat);
347.
348. // Cleanup. Free memory
349. delete[] e_idx;
350. delete[] id_idx;
351. PetscFinalize();
352. }
```

## GTP 8. [POIPETSC] Poisson con PETSc

- Para el “problema de Poisson” en el cuadrado unitario dividido en  $N_x \times N_y$  celdas se suministra un código `poisson2D-df.c` La práctica consiste en modificar este código para resolver la ecuación de *advección-difusión-reacción*
- Consideremos la ec. de *advección-difusión-reacción* estacionaria lineal escalar

$$\begin{aligned} -k\Delta\phi + \mathbf{a} \cdot \nabla\phi + c\phi &= Q \quad \text{en } 0 < (x, y) < 1 \\ \phi(x = 0) = \phi(y = 0) = \phi(x = 1) = \phi(y = 1) &= 0, \\ Q &= 1. \end{aligned} \tag{8}$$

donde

- ▷  $\phi$  : temperatura del fluido,
- ▷  $k$  : conductividad térmica del fluido,
- ▷  $\mathbf{a} = (a_x, a_y)$  : velocidad del fluido (puede ser positiva o negativa) y
- ▷  $c$  : constante de reacción.

Usar la discretización por diferencias numéricas para las derivadas, por

ejmpl para la dirección  $x$ :

$$\phi_j'' \approx (\phi_{j+1} - 2\phi_j + \phi_{j-1})/h^2 \quad (9)$$

y

$$\phi_j' \approx (\phi_{j+1} - \phi_{j-1})/(2h) \quad (10)$$

Lamentablemente, esta discretización falla cuando el  $\text{Pe} \gg 1$ , en el sentido de que produce fuertes oscilaciones numéricas. La solución usual es agregar una cierta cantidad de *difusión numérica*, es decir que se modifica la ecuación original a

$$-(k + k_{\text{num}})\Delta\phi + \mathbf{a} \cdot \nabla\phi + c\phi = Q \quad (11)$$

donde, en cada una de las direcciones se puede plantear

$$k_{\text{num}} = f(\text{Pe}_h)||\mathbf{a}||h/2, \quad (12)$$

siendo

$$\text{Pe}_h = \frac{\|a\|h}{2k},$$

$$f(\text{Pe}_h) = \begin{cases} \text{Pe}_h/3 & ; \text{ si } \text{Pe}_h < 3; \\ 1 & ; \text{ si } \text{Pe}_h \geq 3; \end{cases} \quad (13)$$

**Si es necesario, realizar la estabilización correspondiente.**

**Realizar las siguientes tareas en base a al código `poisson2D-df.c`:**

1. Modificar el código para introducir los términos advectivo y reactivo.
2. Implementar la solución del sistema lineal  $Ax = b$  ensamblando la matriz  $A$  y vector  $b$  y resolviendo don KSP.
3. Resolver el problema con  $N_x = N_y = 10$  y los siguientes juegos de parámetros:  $(a_x = a_y, c) = (0,0), (100,0), (10000,0), (0,1), (0,10000), (100,100), (10000,10000)$  y  $k = 10^2$ . Graficar la solución  $\phi(x, y)$  en cada uno de los casos.
4. Resolver con los métodos CG, GMRes, RICHARDSON, BICG, CGNE. Usar precondicionamiento de “Jacobi”. Reportar la performance de los solvers ( $\|r_k\|/\|r_0\|$  vs. iters). Tener en cuenta que por defecto en PETSc se reportan para algunos solvers los residuos del problema precondicionado (si es que se usó). Tener en cuenta los siguientes

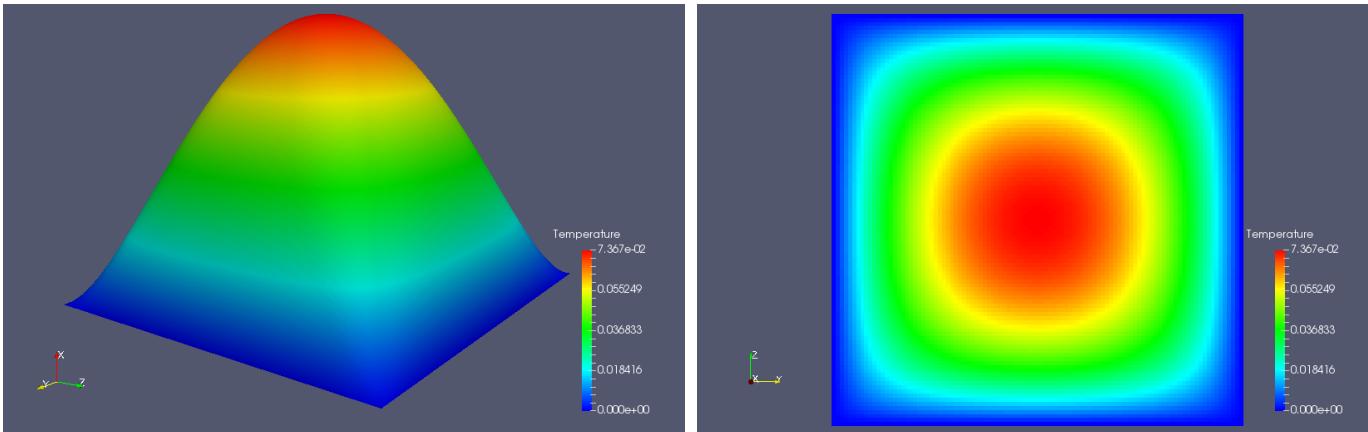
juegos de parámetros:  $(a_x = a_y, c) = (100, 10000)$  y  $k = 10^2$ . Resolver con valores de  $N_x = N_y = 100, 1000$ .

5. Para GMRES y BICG estudiar la escalabilidad en una malla cuyo tamaño crezca en función de  $n_{\text{proc}}$ . El solver es escalable si la cantidad de iteraciones se mantiene (más o menos) constante a medida que el tamaño del problema crece en la misma proporción que el número de procesadores  $n_{\text{proc}}$ . Realizar la gráfica de número de iteraciones para converger en el residuo relativo ( $\|r_k\|/\|r_0\|$ ) una dada tolerancia (e.g.  $10^{-7}$ ) en función del número de procesadores  $n_{\text{proc}}$  comparando performances para alguno de los casos advecitivo-difusivo-reactivo.

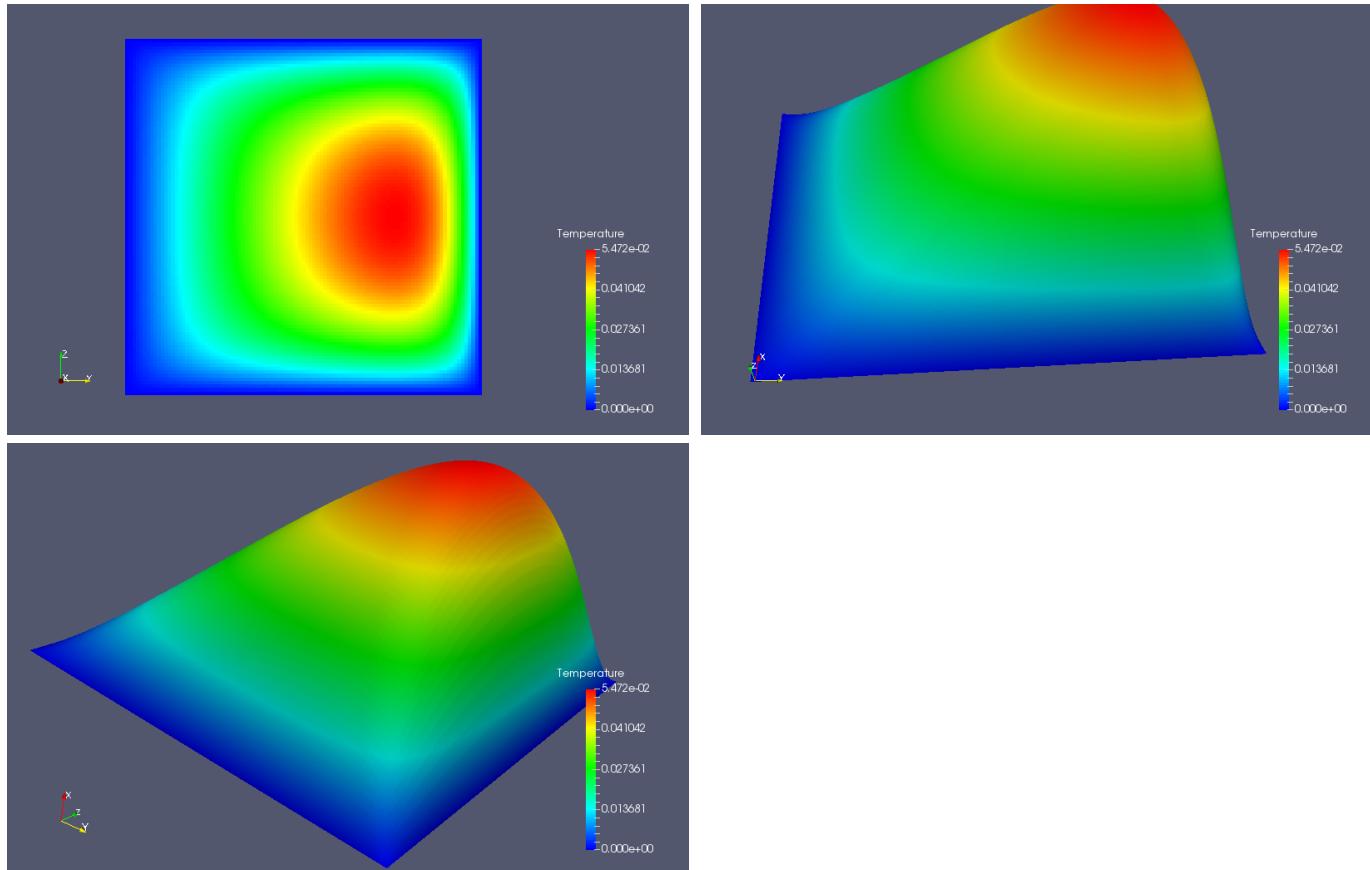
## GTP 8. [POIPETSC] Poisson con PETSc (cont.)

Como referencia, una malla de 100x100 segmentos tenemos los siguientes resultados:

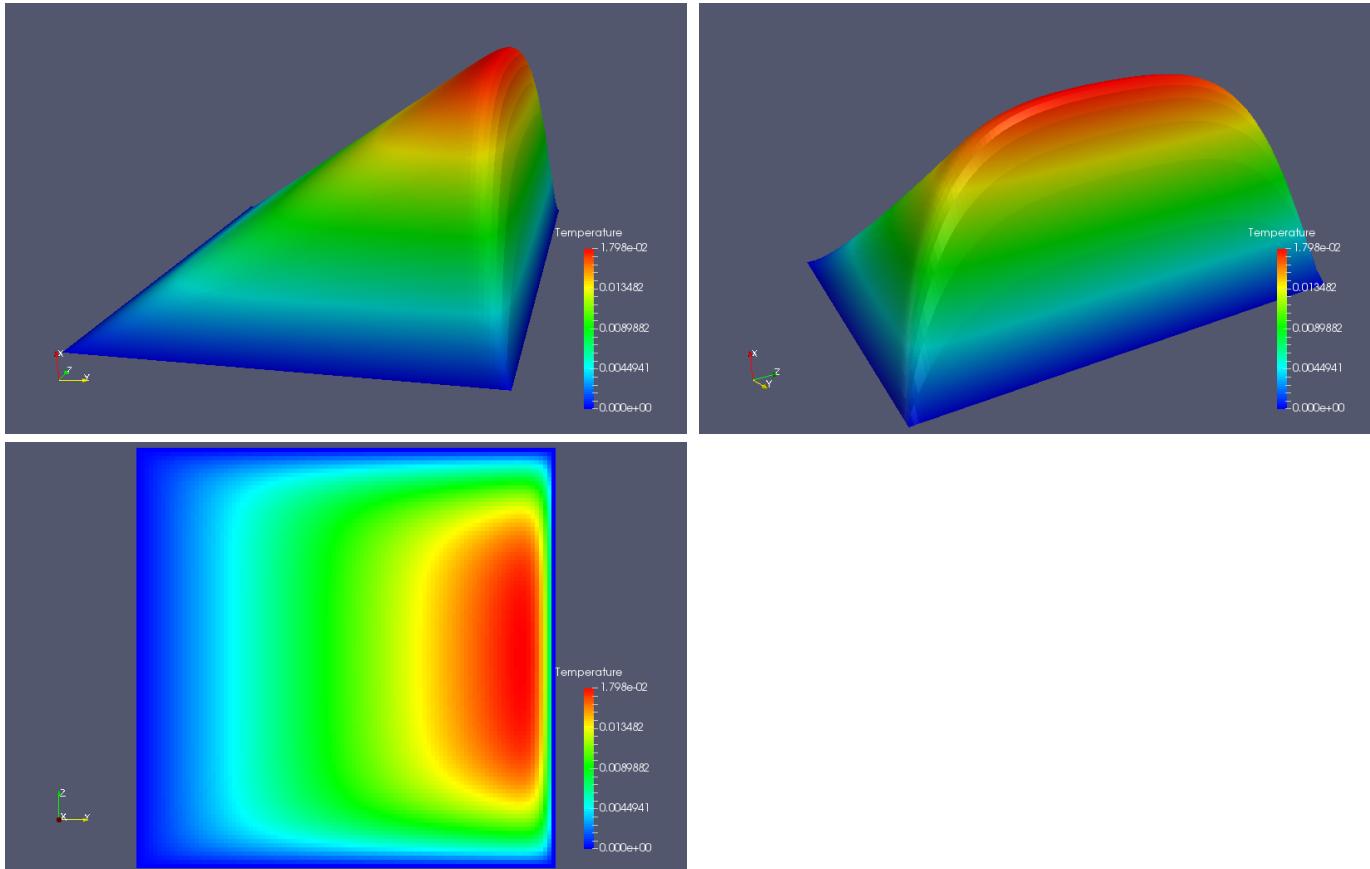
- Para  $k = 1, \mathbf{a} = 0, c = 0, \phi_{\max} = 0.073653$ , en  $\mathbf{x} = (0.5, 0.5)$ .



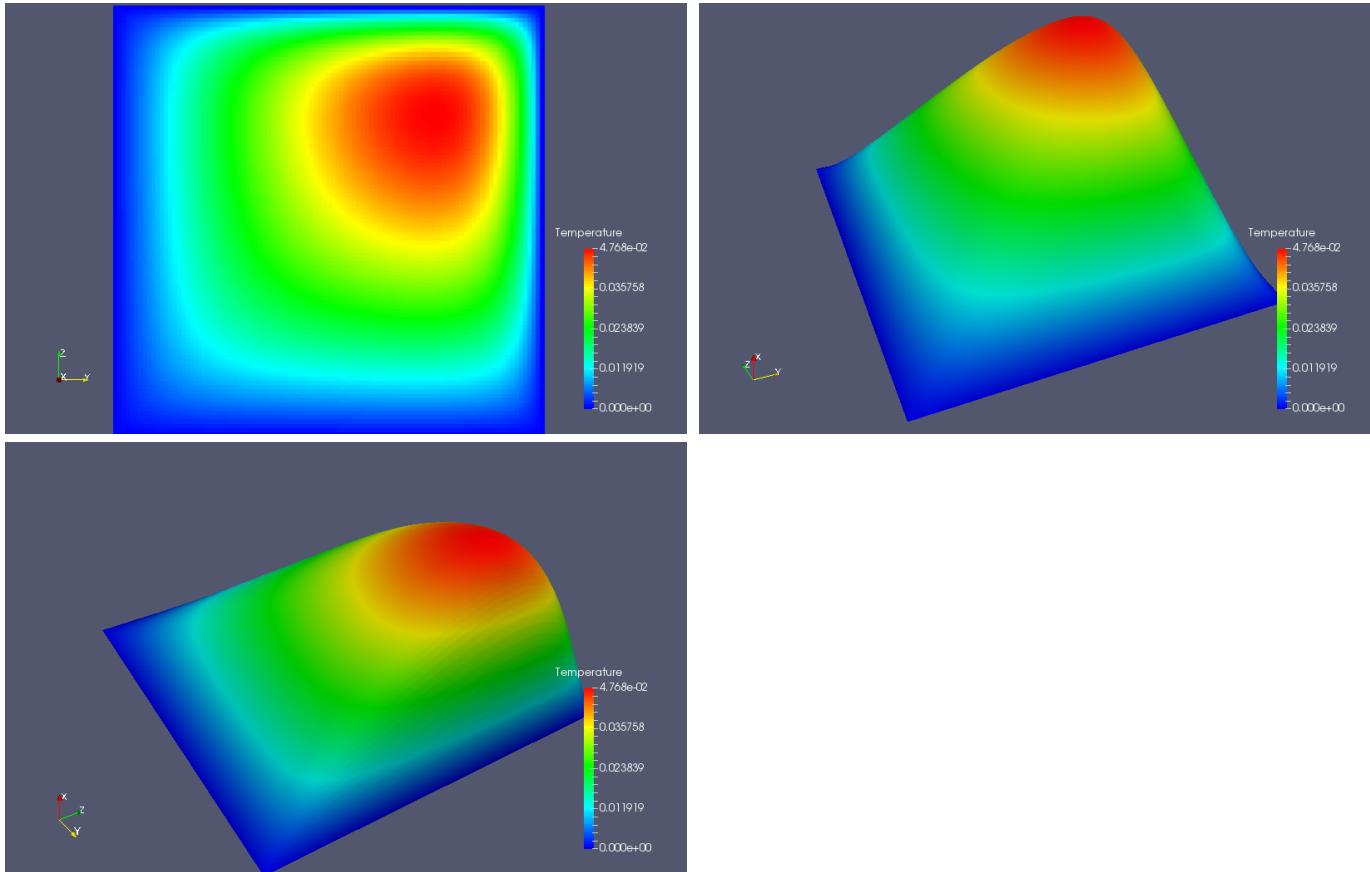
- Para  $k = 1, \mathbf{a} = (10, 0), c = 0, \phi_{\max} = 0.054723$ , en  $\mathbf{x} = (0.76, 0.5)$ .



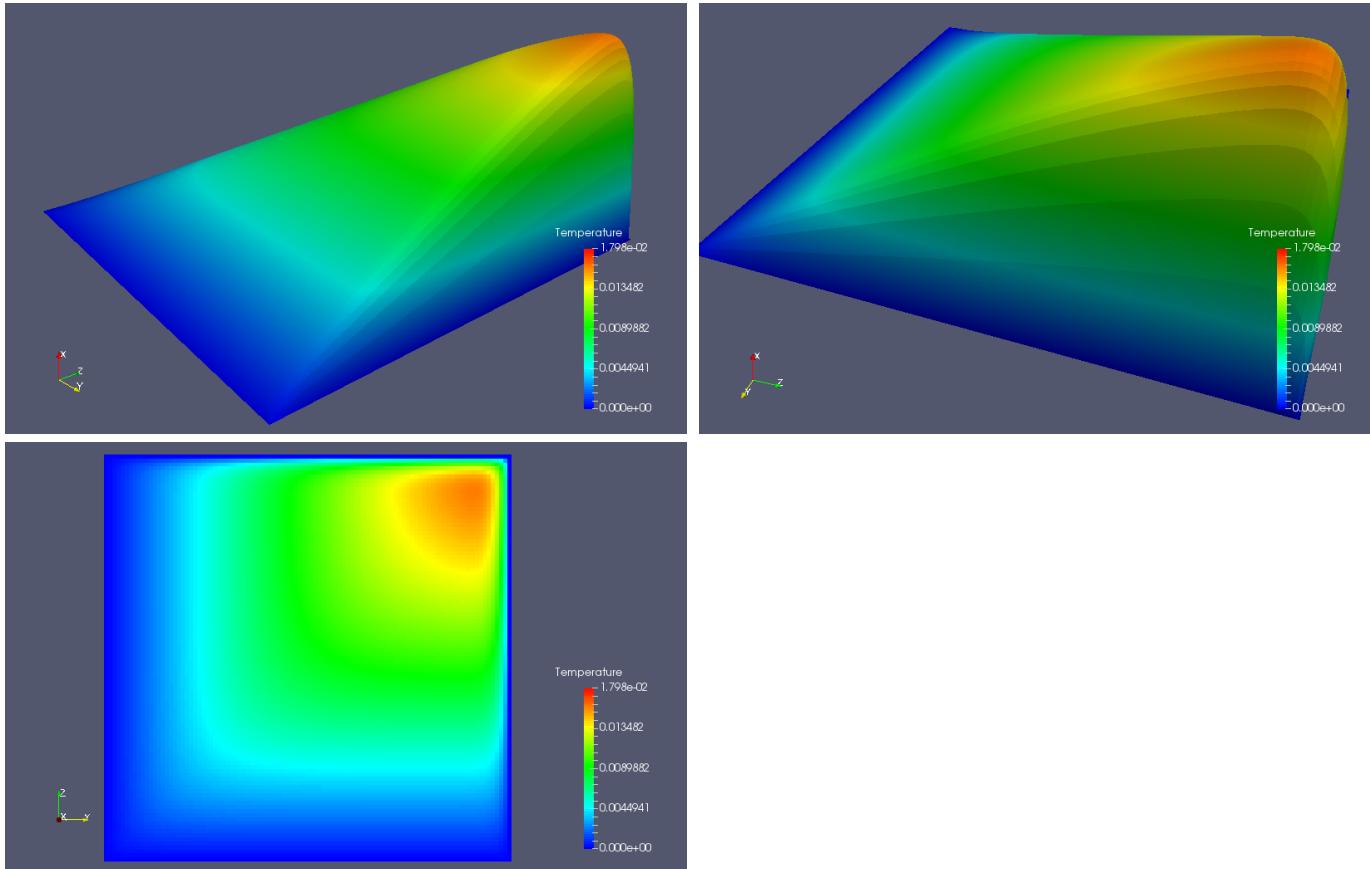
- Para  $k = 1, \mathbf{a} = (50, 0), c = 0, \phi_{\max} = 0.017976$ , en  
 $\mathbf{x} = (0.91584, 0.5)$ .



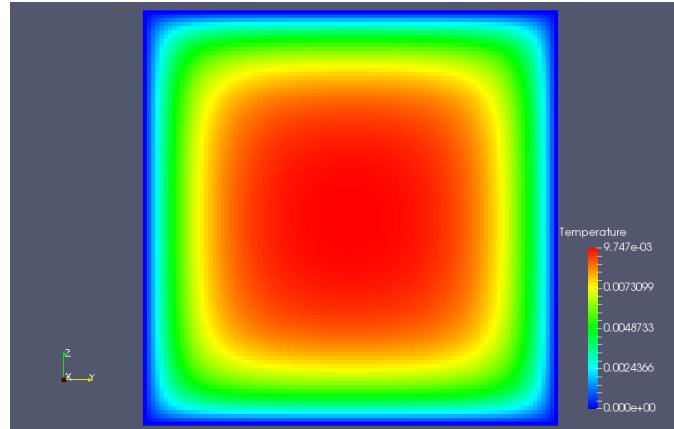
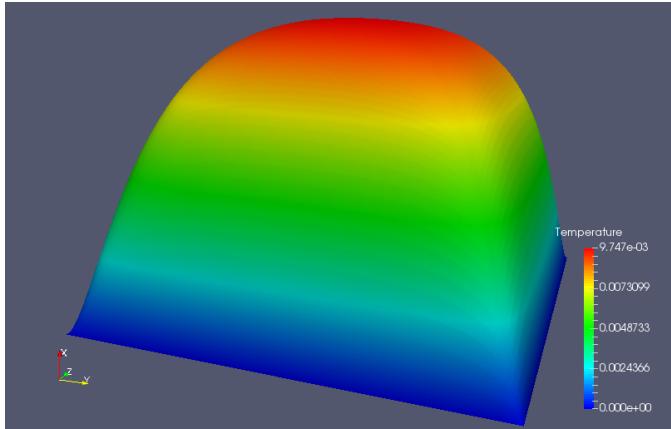
- Para  $k = 1, \mathbf{a} = (10, 10), c = 0, \phi_{\max} = 0.047678$ , en  $\mathbf{x} = (0.75, 0.75)$ .



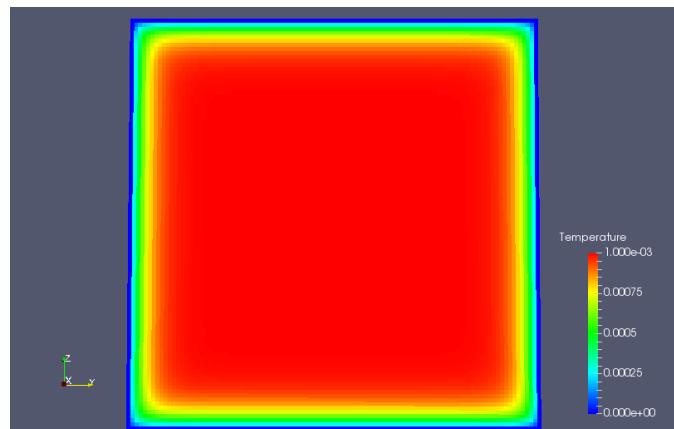
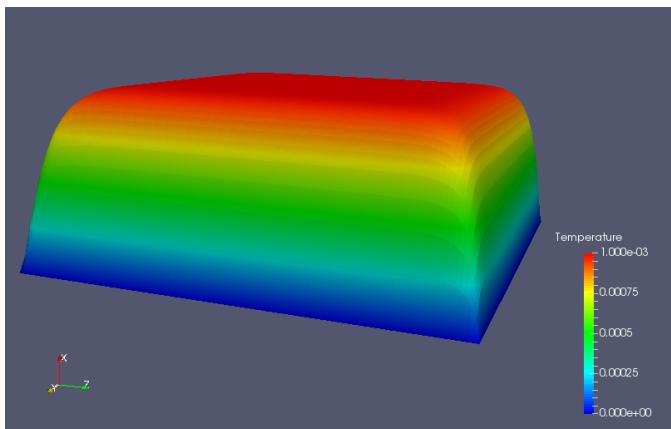
- Para  $k = 1, \mathbf{a} = (50, 50), c = 0, \phi_{\max} = 0.015764$ , en  $\mathbf{x} = (0.91, 0.91)$ .



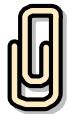
- Para  $k = 1, \mathbf{a} = (0, 0), c = 100, \phi_{\max} = 0.0097465$ , en  $\mathbf{x} = (0.5, 0.5)$ .



- Para  $k = 1$ ,  $\mathbf{a} = (0, 0)$ ,  $c = 1000$ ,  $\phi_{\max} = 0.0097465$ , en  $\mathbf{x} = (0.5, 0.5)$ .



## GTP 8. [POIPETSC] Poisson con PETSc (cont.)



[Descargar: ./example/poiss2ddf/poiss2ddf.cpp]

```
1. // Program usage: mpirun -np <procs> poisson.bin [-help] [all PETSc options]
2.
3. static char help[] = "Solves a linear system in parallel with KSP.\n\
4. Input parameters include:\n\
5. -random_exact_sol : use a random exact solution vector\n\
6. -view_exact_sol : write exact solution vector to stdout\n\
7. -m <mesh_x> : number of mesh points in x-direction\n\
8. -n <mesh_n> : number of mesh points in y-direction\n\n";
9.
10. //#include "petscksp.h"
11. #include "petsc.h"
12. #include "petscsys.h"
13.
14. #undef __FUNCT__
15. #define __FUNCT__ "main"
16. int main(int argc,char **args)
17. {
18. Vec x,b,u; // approx solution, RHS, exact solution
19. Mat A; // linear system matrix
20. KSP ksp; // linear solver context
21. PetscInt i,j,I,J,Istart,Iend,n = 100,m;
```

```
22. PetscErrorCode ierr;
23. int myrank, size;
24.
25. PetscInitialize(&argc,&args,(char *)0,help);
26. ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
27. MPI_Comm_rank(PETSC_COMM_WORLD,&myrank);
28. MPI_Comm_size(PETSC_COMM_WORLD,&size);
29.
30. double start = MPI_Wtime();
31. int n1 = n-1;
32. m=n1;
33.
34. /* -----
35. Compute the matrix and right-hand-side vector that define
36. the linear system, Ax = b.
37. -----
38. */
39. /* Create parallel matrix, specifying only its global dimensions.
40. When using MatCreate(), the matrix format can be specified at
41. runtime. Also, the parallel partitioning of the matrix is
42. determined by PETSc at runtime.
43.
44. Performance tuning note: For problems of substantial size,
45. preallocation of matrix memory is crucial for attaining good
46. performance. Since preallocation is not possible via the generic
47. matrix creation routine MatCreate(), we recommend for practical
48. problems instead to use the creation routine for a particular matrix
49. format, e.g.,
50. MatCreateMPIAIJ() - parallel AIJ (compressed sparse row)
```

```
51. MatCreateMPIBAIJ() - parallel block AIJ
52. See the matrix chapter of the users manual for details.
53. */
54. ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD,PETSC_DECIDE,
55. PETSC_DECIDE,m*n1,m*n1,
56. 5,NULL,0,NULL,&A); CHKERRQ(ierr);
57.
58. ierr = MatSetFromOptions(A);CHKERRQ(ierr);
59.
60. /*
61. Currently, all PETSc parallel matrix formats are partitioned by
62. contiguous chunks of rows across the processors. Determine which
63. rows of the matrix are locally owned.
64. */
65. ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);
66.
67. /*
68. Set matrix elements for the 2-D, five-point stencil in parallel.
69. - Each processor needs to insert only elements that it owns
70. locally (but any non-local elements will be sent to the
71. appropriate processor during matrix assembly).
72. - Always specify global rows and columns of matrix entries.
73.
74. Note: this uses the less common natural ordering that orders first
75. all the unknowns for $x = h$ then for $x = 2h$ etc; Hence you see $J = I \pm n_1$
76. instead of $J = I \pm m$ as you might expect. The more standard ordering
77. would first do all variables for $y = h$, then $y = 2h$ etc.
78. */
79. double h=1.0/(n1+1),
```

```
80. kond=1.0,
81. ax = 0.0,
82. ay = 0.0,
83. c = 0,
84. coef = kond/(h*h),
85. c0 = 4*coef+c,
86. cW = -coef-ax/(2*h),
87. cE = -coef+ax/(2*h),
88. cS = -coef-ay/(2*h),
89. cN = -coef+ay/(2*h);
90. for (I=Istart; I<Iend; I++) {
91. i = I/n1; j = I - i*n1;
92. if (i>0) {J = I - n1; ierr =
93. MatSetValues(A,1,&I,1,&J,&cS,INSERT_VALUES);CHKERRQ(ierr);}
94. if (i<m-1) {J = I + n1; ierr =
95. MatSetValues(A,1,&I,1,&J,&cN,INSERT_VALUES);CHKERRQ(ierr);}
96. if (j>0) {J = I - 1; ierr =
97. MatSetValues(A,1,&I,1,&J,&cW,INSERT_VALUES);CHKERRQ(ierr);}
98. if (j<n1-1) {J = I + 1; ierr =
99. MatSetValues(A,1,&I,1,&J,&cE,INSERT_VALUES);CHKERRQ(ierr);}
100. ierr = MatSetValues(A,1,&I,1,&I,&c0,INSERT_VALUES);CHKERRQ(ierr);
101. }
102. /*
103. * Assemble matrix, using the 2-step process:
104. * MatAssemblyBegin(), MatAssemblyEnd()
105. * Computations can be done while messages are in transition
106. * by placing code between these two statements.
107. */
```

```

108. */
109. ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
110. ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
111.
112. /*
113. Create parallel vectors.
114. - We form 1 vector from scratch and then duplicate as needed.
115. - When using VecCreate(), VecSetSizes and VecSetFromOptions()
116. in this example, we specify only the
117. vector's global dimension; the parallel partitioning is determined
118. at runtime.
119. - When solving a linear system, the vectors and matrices MUST
120. be partitioned accordingly. PETSc automatically generates
121. appropriately partitioned matrices and vectors when MatCreate()
122. and VecCreate() are used with the same communicator.
123. - The user can alternatively specify the local vector and matrix
124. dimensions when more sophisticated partitioning is needed
125. (replacing the PETSC_DECIDE argument in the VecSetSizes() statement
126. below).
127. */
128. ierr = VecCreate(PETSC_COMM_WORLD,&u);CHKERRQ(ierr);
129. ierr = VecSetSizes(u,PETSC_DECIDE,m*n1);CHKERRQ(ierr);
130. ierr = VecSetFromOptions(u);CHKERRQ(ierr);
131. ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
132. ierr = VecDuplicate(b,&x);CHKERRQ(ierr);
133.
134. for (I=Istart; I<Iend; I++) {
135. i = I/n1; j = I - i*n1;

```

```

136. double x=i*h, y=j*h;
137. #define SQ(x) (x)*(x)
138. double r = sqrt(SQ(x-0.5)+SQ(y-0.5));
139. double q = r>0.2 && r<0.25;
140. VecSetValue(b,I,q,ADD_VALUES);
141. }
142. ierr = VecAssemblyBegin(b); CHKERRQ(ierr);
143. ierr = VecAssemblyEnd(b); CHKERRQ(ierr);
144.
145. /* ----- Create the linear solver and set various options -----
146. -----
147. */
148.
149. /*
150. Create linear solver context
151. */
152. ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
153.
154. /*
155. Set operators. Here the matrix that defines the linear system
156. also serves as the preconditioning matrix.
157. */
158. ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);
159.
160. /*
161. Set linear solver defaults for this problem (optional).
162. - By extracting the KSP and PC contexts from the KSP context,
163. we can then directly call any KSP and PC routines to set
164. various options.
165. - The following two statements are optional; all of these

```

```

166. parameters could alternatively be specified at runtime via
167. KSPSetFromOptions(). All of these defaults can be
168. overridden at runtime, as indicated below.
169. */
170.
171. ierr = KSPSetTolerances(ksp,1.e-2/((m+1)*(n1+1)),1.e-50,PETSC_DEFAULT,
172. PETSC_DEFAULT);CHKERRQ(ierr);
173. /*
174. Set runtime options, e.g.,
175. -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
176. These options will override those specified above as long as
177. KSPSetFromOptions() is called _after_ any other customization
178. routines.
179. */
180. ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);
181.
182. MPI_Barrier(PETSC_COMM_WORLD);
183. double elaps0 = MPI_Wtime()-start;
184. PetscPrintf(PETSC_COMM_WORLD,"n %d, size %d, assembly elaps %f[s]\n",
185. n,size,elaps0);
186.
187. /* ----- Solve the linear system -----
188. ----- */
189. // PetscPreLoadBegin(PETSC_TRUE,“Solve System profiling”);
190. ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
191. // PetscPreLoadEnd();
192. elaps0 = MPI_Wtime()-start;

```

```

194. PetscPrintf(PETSC_COMM_WORLD,"After solve %f[s]\n",elaps0);
195.
196. #if 1
197. PetscObjectSetName((PetscObject)x,"temp");
198. PetscViewer viewer;
199. ierr = PetscViewerHDF5Open(PETSC_COMM_WORLD,"./temp.h5",
200. FILE_MODE_WRITE,&viewer); CHKERRQ(ierr);
201. ierr = VecView(x,viewer); CHKERRQ(ierr);
202. ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
203. #else
204. ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
205. #endif
206.
207. /*
208. Free work space. All PETSc objects should be destroyed when they
209. are no longer needed.
210. */
211. ierr = KSPDestroy(&ksp);CHKERRQ(ierr);
212. ierr = VecDestroy(&u);CHKERRQ(ierr);
213. ierr = VecDestroy(&x);CHKERRQ(ierr);
214. ierr = VecDestroy(&b);CHKERRQ(ierr);
215. ierr = MatDestroy(&A);CHKERRQ(ierr);
216.
217. MPI_Barrier(PETSC_COMM_WORLD);
218. double elaps = MPI_Wtime()-start;
219.
220. PetscPrintf(PETSC_COMM_WORLD,"%d, size %d, elaps %f[s]\n",n,size,elaps);
221. /*
222. Always call PetscFinalize() before exiting a program. This routine

```

```
223. - finalizes the PETSc libraries as well as MPI
224. - provides summary and diagnostic information if certain runtime
225. options are chosen (e.g., -log_summary).
226. */
227. ierr = PetscFinalize();CHKERRQ(ierr);
228.
229. return 0;
230. }
```

## GTP 9. [SNES] PETSc/SNES. Ejemplo combustión

- Resolver el siguiente problema nolineal de combustión usando el SNES de PETSc:

$$-k\Delta T + c\varphi(T) = 0, \quad \text{en } \Omega = 0 < x, y < 1,$$

donde  $\varphi(T) = T(0.5 - T)(1 - T),$  (14)

con b.c.  $T = \bar{T}, \quad \text{en } x = 0, x = 1, y = 0, y = 1.$

- Este problema presenta el inconveniente de que es fuertemente nolineal, de hecho puede tener múltiples soluciones. La física del problema está explicada en el capítulo 7, pág 24 de este apunte <http://goo.gl/20D8yT>.
- Utilizar diferencias finitas con una grilla de paso uniforme  $h = 1/N.$  Por lo tanto el vector de estado (las temperaturas) va a tener tamaño  $\text{nnod}=(N+1)^2.$
- La ecuación correspondiente al nodo  $(i,j)$  es

$$k \frac{4T_O - T_S - T_W - T_E - T_N}{h^2} + c\varphi(T_O) = 0, \quad (15)$$

donde  $O$  es el nodo  $(i, j)$ ,  $W$  es el nodo al oeste  $(i - 1, j)$  y así siguiendo. Esta ecuación es para los nodos interiores  $1 \leq i, j \leq N - 1$ . Los otros nodos son de contorno y por lo tanto su valor está fijo a 0, por lo tanto la ecuación correspondiente es  $T_{ij} = 0$ .

- Se suministra un código base `snes2.cpp` que resuelve el problema 1D en secuencial. La consigna del examen es escribir un programa que resuelva el problema 2D en paralelo.
- Se sugiere usar un particionamiento del problema por filas. Asumir que el número de filas (`N+1`) es múltiplo del número de procesadores  $P$ . Una posibilidad es que si el programa recibe como dato un entero `N` y `N+1` no es múltiplo de `P`, tome el valor inmediatamente superior tal que sí cumpla la condición `N -> N-(N%P)+P-1`.
- Cada proceso procesa  $m=(N+1)/P$  filas, es decir el rango `[m*rank, m*(rank+1))`. Calcula el residuo y Jacobiano de los nodos correspondientes, es decir los que están en ese rango.
- En el momento de calcular los residuos para el nodo  $(i, j)$  ocurre un problema si el `j` es el primero o último de ese proceso ya que necesita valores que están en otro procesador. Se suministra una función

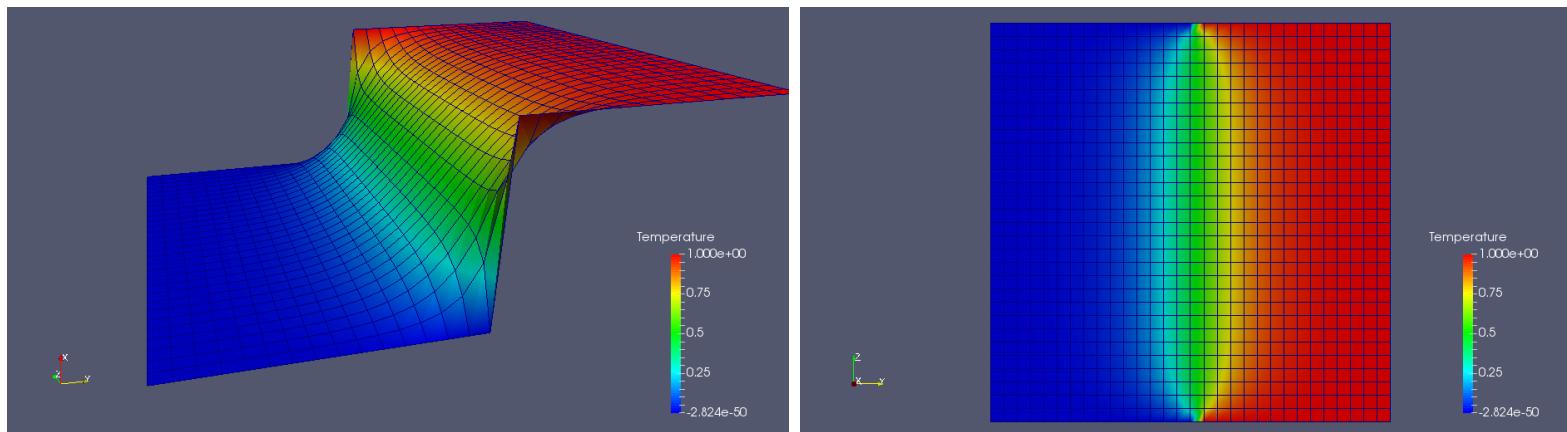
```
void vec_gather(MPI_Comm comm, Vec v, vector<double> &values);
```

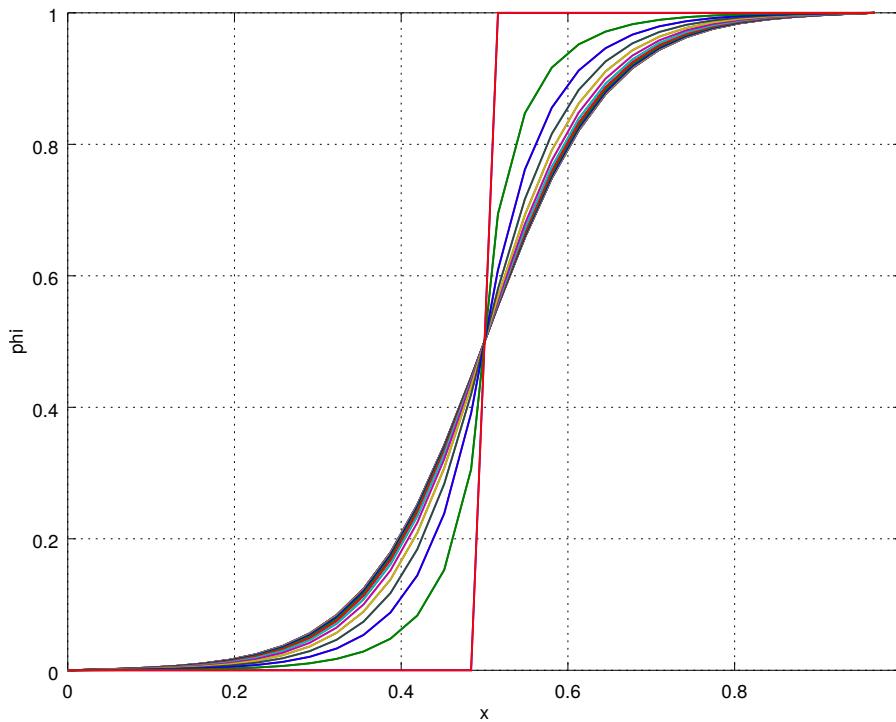
Dado un arreglo distribuido de PETSc **v** esta función se encarga de juntar todos los valores (entre todos los procesos) y juntarlos en un gran vector **values**. Este **values** está disponible en todos los procesos.

- Como control: para  $c = 1, k = 3 \times 10^{-3}$  si la condición de contorno es

$$T = \begin{cases} 0; & \text{si } x < 0.5; \\ 1; & \text{si } x > 0.5; \end{cases} \quad (16)$$

(en C++ simplemente `phi=(x>0.5)`). Los resultados son como siguen:





En particular en el eje de simetría  $y = 0.5$  tenemos

- ▷  $T = 0.8$ , para  $x = (0.6, 0.5)$
- ▷  $T = 0.2$ , para  $x = (0.4, 0.5)$

- **Condiciones de contorno:** La mejor forma de hacerlo es reemplazar la línea correspondiente por la ecuación de contorno. Para ellos hay que poner un coeficiente unitario en la diagonal, cero en todo el resto de la fila, y el valor impuesto en el miembro derecho. Para esto puede ser interesante usar la función `MatZeroRows`.

# SNES: Solvers no-lineales



[Descargar: ./hpcmc-examples/snes2.cpp]

```
1. static char help[] =
2. "Newton's method to solve a combustion-like 1D problem.\n";
3.
4. #include <vector>
5. #include <cstdlib>
6. #include "petscsnes.h"
7. #include "H5Cpp.h"
8.
9. using namespace std;
10.
11. /*
12. User-defined routines
13. */
14. extern int resfun(SNES,Vec,Vec,void*);
```

```
15. extern int jacfun(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
16.
17. struct SnesCtx {
18. int N;
19. double k,c,h,f0;
20. };
21.
22. // Given a distributed PETSc vector 'vec' gather all the
23. // ranges in all processor in a full vector of doubles
24. // 'values'. This full vector is available in all the
25. // processors.
26. // WARNING: this may be inefficient and non
27. // scalable, it is just a dirty trick to have access to all
28. // the values of the vectors in all the processor.
29. // Usage:
30. // Vec v;
31. // // ... create and fill v eith values at each processor
32. // // ... do the Assembly
33. // vector<double> values;
34. // vec_gather(MPI_COMM_WORLD,v,values);
35. // //... now you have all the elements of 'v' in 'values'
36. void vec_gather(MPI_Comm comm,Vec v,vector<double> &values) {
37. // n: global size of vector
38. // nlocal: local (PETSc) size
39. int n,nlocal;
40. // Get the global size
41. VecGetSize(v,&n);
42. // Resize the local buffer
43. values.clear();
```

```
44. values.resize(n,0.0);
45. // Get the local size
46. VecGetLocalSize(v,&nlocal);
47.
48. // Gather all the local sizes in order to compute the
49. // counts and displs for the Allgatherv
50. int size, myrank;
51. MPI_Comm_rank(comm,&myrank);
52. MPI_Comm_size(comm,&size);
53. vector<int> counts(size),displs(size);
54. MPI_Allgather(&nlocal,1,MPI_INT,
55. &counts[0],1,MPI_INT,comm);
56. displs[0]=0;
57. for (int j=1; j<size; j++)
58. displs[j] = displs[j-1] + counts[j-1];
59.
60. // Get the internal values of the PETSc vector
61. double *vp;
62. VecGetArray(v,&vp);
63. // Do the Allgatherv to the local vector
64. MPI_Allgatherv(vp,nlocal,MPI_DOUBLE,
65. &values[0],&counts[0],&displs[0],MPI_DOUBLE,comm);
66. // Restore the array
67. VecRestoreArray(v,&vp);
68. }
69.
70. //:-<*>-:<*>-:<*>-:<*>-:<*>
71. int h5petsc_vec_save(Vec x,const char *filename,const char *varname) {
```

```
72. vector<double> vx;
73. vec_gather(PETSC_COMM_WORLD,x,vx);
74. H5::H5File file(filename,H5F_ACC_TRUNC);
75. hsize_t n = vx.size();
76. H5::DataSpace dataspace(1,&n);
77. // Create the dataset.
78. string svar(varname);
79. H5::DataSet xdset =
80. file.createDataSet(svar,H5::PredType::NATIVE_DOUBLE,dataspace);
81. xdset.write(vx.data(),H5::PredType::NATIVE_DOUBLE);
82. file.close();
83. return 0;
84. }
85.
86. #undef __FUNCT__
87. #define __FUNCT__ "main"
88. int main(int argc,char **argv)
89. {
90. SNES snes; /* nonlinear solver context */
91. Vec x,r; /* solution, residual vectors */
92. Mat J; /* Jacobian matrix */
93. int ierr,its,size;
94. SnesCtx ctx;
95. int N = 100;
96.
97. PetscInitialize(&argc,&argv,(char *)0,help);
98. ierr = PetscOptionsGetInt(PETSC_NULL,"-N",&N,PETSC_NULL);CHKERRQ(ierr);
99.
```

```

100. ctx.N = N;
101. ctx.k = 0.001;
102. ctx.c = 1;
103. ctx.f0 = 0.0;
104. ctx.h = 1.0/N;
105.
106. ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
107. if (size != 1) SETERRQ(PETSC_COMM_WORLD,1,"This is a uniprocessor example only!");
108.
109. // Create nonlinear solver context
110. ierr = SNESCreate(PETSC_COMM_WORLD,&snes); CHKERRQ(ierr);
111. ierr = SNESSetType(snes,SNESLS); CHKERRQ(ierr);
112. SNESSetFromOptions(snes);
113. double abstol, rtol, stol;
114. int maxit, maxf;
115. SNESGetTolerances(snes,&abstol,&rtol,&stol,&maxit,&maxf);
116. PetscPrintf(PETSC_COMM_WORLD,"atol=%g, rtol=%g, stol=%g, maxit=%D, maxf=%D\n",
117. (double)abstol,(double)rtol,(double)stol,maxit,maxf);
118.
119. // Create matrix and vector data structures;
120. // set corresponding routines
121.
122. // Create vectors for solution and nonlinear function
123. ierr = VecCreateSeq(PETSC_COMM_SELF,N+1,&x);CHKERRQ(ierr);
124. ierr = VecDuplicate(x,&r); CHKERRQ(ierr);
125.
126. double *xx;
127. ierr = VecGetArray(x,&xx); CHKERRQ(ierr);
128. for (int j=0; j<=N; j++) {

```

```

129. xx[j]=((j %50)>25);
130. }
131. ierr = VecRestoreArray(x,&xx); CHKERRQ(ierr);
132.
133. ierr = MatCreateMPIAIJ(PETSC_COMM_SELF,PETSC_DECIDE,
134. PETSC_DECIDE,N+1,N+1,
135. 1,NULL,0,NULL,&J);CHKERRQ(ierr);
136.
137. ierr = SNESSetFunction(snes,r,resfun,&ctx); CHKERRQ(ierr);
138. ierr = SNESSetJacobian(snes,J,J,jacfun,&ctx); CHKERRQ(ierr);
139.
140. ierr = SNESolve(snes,NULL,x);CHKERRQ(ierr);
141.
142. #if 0
143. // Doesn't work
144. PetscObjectSetName((PetscObject)x,"temp");
145. PetscViewer viewer;
146. ierr = PetscViewerHDF5Open(PETSC_COMM_WORLD,"./temp.h5",
147. FILE_MODE_WRITE,&viewer); CHKERRQ(ierr);
148. ierr = VecView(x,viewer);CHKERRQ(ierr);
149. ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);
150. #elif 1
151. // Use the function defined above
152. h5petsc_vec_save(x,"temp.h5","u");
153. #else
154. // Just plain output to stdout
155. ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
156. #endif
157.
```

```
158. Vec f;
159. ierr = SNESGetFunction(snes,&f,0,0);CHKERRQ(ierr);
160. double rnorm;
161. ierr = VecNorm(r,NORM_2,&rnorm);
162.
163. SNESGetLinearSolveIterations(snes,&its);
164. ierr = PetscPrintf(PETSC_COMM_SELF,
165. "number of Newton iterations = "
166. "%d, norm res %g\n",
167. its,rnorm);CHKERRQ(ierr);
168.
169. ierr = VecDestroy(&x);CHKERRQ(ierr);
170. ierr = VecDestroy(&r);CHKERRQ(ierr);
171. ierr = SNESDestroy(&snes);CHKERRQ(ierr);
172.
173. PetscLogView(PETSC_VIEWER_STDOUT_WORLD);
174. ierr = PetscFinalize();CHKERRQ(ierr);
175. return 0;
176. }
177.
178. //:-<*>-:<*>-:<*>-:<*>-:<*>
179. #undef __FUNCT__
180. #define __FUNCT__ "resfun"
181. int resfun(SNES snes,Vec x,Vec f,void *data)
182. {
183. double *xx,*ff;
184. SnesCtx &ctx = *(SnesCtx *)data;
185. int ierr;
186. double h = ctx.h;
```

```

187.
188. ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
189. ierr = VecGetArray(f,&ff);CHKERRQ(ierr);
190.
191. ff[0] = xx[0];
192. ff[ctx.N] = xx[ctx.N];
193.
194. for (int j=1; j<ctx.N; j++) {
195. double xxx = xx[j];
196. ff[j] = ctx.f0 + ctx.c*xxx*(0.5-xxx)*(1.0-xxx);
197. ff[j] += ctx.k*(-xx[j+1]+2.0*xx[j]-xx[j-1])/(h*h);
198. }
199.
200. ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
201. ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);
202.
203. #if 0
204. ierr = PetscPrintf(PETSC_COMM_SELF,"x:\n");
205. ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
206. ierr = PetscPrintf(PETSC_COMM_SELF,"f:\n");
207. ierr = VecView(f,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
208. ierr = PetscPrintf(PETSC_COMM_SELF,"=====\\n");
209. #endif
210. return 0;
211. }
212.
213. //:-<*>-:<*>-:<*>-:<*>-:<*>
214. #undef __FUNCT__
215. #define __FUNCT__ "jacfun"

```

```

216. int jacfun(SNES snes,Vec x,Mat* jac,Mat* jac1,
217. MatStructure *flag,void *data) {
218. double *xx, A;
219. SnesCtx &ctx = *(SnesCtx *)data;
220. int ierr, j;
221.
222. ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
223. ierr = MatZeroEntries(*jac);
224.
225. j=0; A = 1;
226. ierr = MatSetValues(*jac,1,&j,1,&j,&A,
227. INSERT_VALUES); CHKERRQ(ierr);
228.
229. j=ctx.N; A = 1;
230. ierr = MatSetValues(*jac,1,&j,1,&j,&A,
231. INSERT_VALUES); CHKERRQ(ierr);
232.
233. int cols[3];
234. double coefs[3];
235. double h=ctx.h, h2=h*h;
236. coefs[0] = -ctx.k*1.0/h2;
237. coefs[1] = +ctx.k*2.0/h2;
238. coefs[2] = -ctx.k*1.0/h2;
239. for (j=1; j<ctx.N; j++) {
240. double xxx = xx[j];
241. A = ctx.c * ((0.5-xxx)*(1.0-xxx) - xxx*(1.0-xxx) - xxx*(0.5-xxx));
242. ierr = MatSetValues(*jac,1,&j,1,&j,&A,INSERT_VALUES);
243. // ff[j] += ctx.k*(-xx[j+1]+2.0*xx[j+1]-xx[j-1])/(h*h);
244. cols[0] = j-1;

```

```
245. cols[1] = j;
246. cols[2] = j+1;
247. ierr = MatSetValues(*jac,1,&j,3,cols,coefs,ADD_VALUES);
248.
249. CHKERRQ(ierr);
250. }
251. ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
252. ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
253. // ierr = MatView(*jac,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
254. // printf("en jacfun\n");
255. ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
256. return 0;
257. }
```

# TS: time stepping



[Descargar: ./example/ex3heat.cpp]

```
1. /* Program usage: ex3 [-help] [all PETSc options] */
2. static char help[] ="Solves a simple time-dependent linear PDE \
3. (the heat equation).\n\
4. Input parameters include:\n\
5. -m <points>, where <points> = number of grid points\n\
6. -time_dependent_rhs : Treat the problem as having a time-dependent \
7. right-hand side\n\
8. -debug : Activate debugging printouts\n\
9. -nox : Deactivate x-window graphics\n\n";
10. /*
11. Concepts: TS^time-dependent linear problems
12. Concepts: TS^heat equation
13. Concepts: TS^diffusion equation
14. Processors: 1
15. */
16.
17. /* -----
```

18.  
19. This program solves the one-dimensional heat equation (also called the  
20. diffusion equation),  
21.  $u_t = u_{xx}$ ,  
22. on the domain  $0 \leq x \leq 1$ , with the boundary conditions  
23.  $u(t, 0) = 0$ ,  $u(t, 1) = 0$ ,  
24. and the initial condition  
25.  $u(0, x) = \sin(6\pi x) + 3\sin(2\pi x)$ .  
26. This is a linear, second-order, parabolic equation.  
27.  
28. We discretize the right-hand side using finite differences with  
29. uniform grid spacing  $h$ :  
30.  $u_{xx} = (u_{i+1} - 2u_i + u_{i-1})/(h^2)$   
31. We then demonstrate time evolution using the various TS methods by  
32. running the program via  
33. `ex3 -ts_type <timestepping solver>`  
34.  
35. We compare the approximate solution with the exact solution, given by  
36.  $u_{exact}(x, t) = \exp(-36\pi^2 t) \sin(6\pi x) +$   
37.  $3\exp(-4\pi^2 t) \sin(2\pi x)$   
38.  
39. Notes:  
40. This code demonstrates the TS solver interface to two variants of  
41. linear problems,  $u_t = f(u, t)$ , namely  
42. - time-dependent  $f$ :  $f(u, t)$  is a function of  $t$   
43. - time-independent  $f$ :  $f(u, t)$  is simply  $f(u)$   
44.  
45. The parallel version of this code is `ts/examples/tutorials/ex4.c`  
46.

```
47. _____ */
48.
49. #include "petscts.h"
50.
51. /*
52. User-defined application context - contains data needed by the
53. application-provided call-back routines.
54. */
55. typedef struct {
56. Vec solution; /* global exact solution vector */
57. PetscInt m; /* total number of grid points */
58. PetscReal h; /* mesh width h = 1/(m-1) */
59. PetscTruth debug; /* flag (1 indicates activation
60. of debugging printouts) */
61. PetscViewer viewer1,viewer2; /* viewers for the solution and error */
62. PetscReal norm_2,norm_max; /* error norms */
63. } AppCtx;
64.
65. /*
66. User-defined routines
67. */
68. extern PetscErrorCode InitialConditions(Vec,AppCtx*);
69. extern PetscErrorCode RHSMatrixHeat(TS,PetscReal,Mat*,
70. Mat*,MatStructure*,void*);
71. extern PetscErrorCode Monitor(TS,PetscInt,PetscReal,Vec,void*);
72. extern PetscErrorCode ExactSolution(PetscReal,Vec,AppCtx*);
73. extern PetscErrorCode MyBCRoutine(TS,PetscReal,Vec,void*);
74.
75. #undef __FUNCT__
```

```
76. #define __FUNCT__ "main"
77. int main(int argc,char **argv)
78. {
79. AppCtx appctx; /* user-defined application context */
80. TS ts; /* timestepping context */
81. Mat A; /* matrix data structure */
82. Vec u; /* approximate solution vector */
83. PetscReal time_total_max = 100.0; /* default max total time */
84. PetscInt time_steps_max = 100; /* default max timesteps */
85. PetscDraw draw; /* drawing context */
86. PetscErrorCode ierr;
87. PetscInt steps,m;
88. PetscMPIInt size;
89. PetscReal dt,ftime;
90. PetscTruth flg;
91.
92. /* - - - - - Initialize program and set problem parameters
93. - - - - - */
94.
95.
96. ierr = PetscInitialize(&argc,&argv,(char*)0,help);CHKERRQ(ierr);
97. ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
98. if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
99.
100. m = 60;
101. ierr = PetscOptionsGetInt(PETSC_NULL,"-m",&m,PETSC_NULL);CHKERRQ(ierr);
102. ierr = PetscOptionsHasName(PETSC_NULL,"-debug",
103. &appctx.debug);CHKERRQ(ierr);
```

```
104. appctx.m = m;
105. appctx.h = 1.0/(m-1.0);
106. appctx.norm_2 = 0.0;
107. appctx.norm_max = 0.0;
108. ierr = PetscPrintf(PETSC_COMM_SELF,"Solving a linear TS problem "
109. "on 1 processor\n");CHKERRQ(ierr);
110.
111. /* ----- */
112. Create vector data structures
113. ----- */
114.
115. /*
116. Create vector data structures for approximate and exact solutions
117. */
118. ierr = VecCreateSeq(PETSC_COMM_SELF,m,&u);CHKERRQ(ierr);
119. ierr = VecDuplicate(u,&appctx.solution);CHKERRQ(ierr);
120.
121. /* ----- */
122. Set up displays to show graphs of the solution and error
123. ----- */
124.
125. ierr = PetscViewerDrawOpen(PETSC_COMM_SELF,0,"",80,380,400,160,
126. &appctx.viewer1);CHKERRQ(ierr);
127. ierr = PetscViewerDrawGetDraw(appctx.viewer1,0,&draw);CHKERRQ(ierr);
128. ierr = PetscDrawSetDoubleBuffer(draw);CHKERRQ(ierr);
129. ierr = PetscViewerDrawOpen(PETSC_COMM_SELF,0,"",80,0,400,160,
130. &appctx.viewer2);CHKERRQ(ierr);
131. ierr = PetscViewerDrawGetDraw(appctx.viewer2,0,&draw);CHKERRQ(ierr);
132. ierr = PetscDrawSetDoubleBuffer(draw);CHKERRQ(ierr);
```

```

133.
134. /* -----
135. Create timestepping solver context
136. -----
137.
138. ierr = TSCreate(PETSC_COMM_SELF,&ts);CHKERRQ(ierr);
139. ierr = TSSetProblemType(ts,TS_LINEAR);CHKERRQ(ierr);
140.
141. /* -----
142. Set optional user-defined monitoring routine
143. -----
144.
145. ierr = TSMonitorSet(ts,Monitor,&appctx,PETSC_NULL);CHKERRQ(ierr);
146.
147. /* -----
148.
149. Create matrix data structure; set matrix evaluation routine.
150. -----
151.
152. ierr = MatCreate(PETSC_COMM_SELF,&A);CHKERRQ(ierr);
153. ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m,m);CHKERRQ(ierr);
154. ierr = MatSetFromOptions(A);CHKERRQ(ierr);
155.
156. ierr = PetscOptionsHasName(PETSC_NULL,
157. "-time_dependent_rhs",&flg);CHKERRQ(ierr);
158. if (flg) {
159. /*
160. For linear problems with a time-dependent $f(u,t)$ in the equation
161. $u_t = f(u,t)$, the user provides the discretized right-hand-side
162. as a time-dependent matrix.

```

```

163. */
164. ierr = TSSetMatrices(ts,A,RHSMatrixHeat,PETSC_NULL,
165. PETSC_NULL,DIFFERENT_NONZERO_PATTERN,
166. &appctx);CHKERRQ(ierr);
167. } else {
168. /*
169. For linear problems with a time-independent f(u) in the equation
170. u_t = f(u), the user provides the discretized right-hand-side
171. as a matrix only once, and then sets a null matrix evaluation
172. routine.
173. */
174. MatStructure A_structure;
175. ierr = RHSMatrixHeat(ts,0.0,&A,&A,&A_structure,&appctx);CHKERRQ(ierr);
176. ierr = TSSetMatrices(ts,A,PETSC_NULL,PETSC_NULL,
177. PETSC_NULL,DIFFERENT_NONZERO_PATTERN,
178. &appctx);CHKERRQ(ierr);
179. }
180.
181. /* ----- Set solution vector and initial timestep -----
182. ----- */
183. dt = appctx.h*appctx.h/2.0;
184. ierr = TSSetInitialTimeStep(ts,0.0,dt);CHKERRQ(ierr);
185. ierr = TSSetSolution(ts,u);CHKERRQ(ierr);
186.
187. /* ----- Customize timestepping solver:
188. - Set the solution method to be the Backward Euler method.
189. ----- */
190.
191.
```

```

192. - Set timestepping duration info
193. Then set runtime options, which can override these defaults.
194. For example,
195. -ts_max_steps <maxsteps> -ts_max_time <maxtime>
196. to override the defaults set by TSSetDuration().
197. -----
198.
199. ierr = TSSetDuration(ts,time_steps_max,time_total_max);CHKERRQ(ierr);
200. ierr = TSSetFromOptions(ts);CHKERRQ(ierr);
201.
202. /* -----
203. Solve the problem
204. -----
205.
206. /*
207. Evaluate initial conditions
208. */
209. ierr = InitialConditions(u,&appctx);CHKERRQ(ierr);
210.
211. /*
212. Run the timestepping solver
213. */
214. ierr = TSStep(ts,&steps,&ftime);CHKERRQ(ierr);
215.
216. /* -----
217. View timestepping solver info
218. -----
219.
220. ierr = PetscPrintf(PETSC_COMM_SELF,
221. "avg. error (2 norm) =%G, avg. error (max norm) =%G\n",

```

```

222. appctx.norm_2/steps,appctx.norm_max/steps);CHKERRQ(ierr);
223. ierr = TSView(ts,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
224.
225. /* -----
226. Free work space. All PETSc objects should be destroyed when they
227. are no longer needed.
228. ----- */
229.
230. ierr = TSDestroy(ts);CHKERRQ(ierr);
231. ierr = MatDestroy(A);CHKERRQ(ierr);
232. ierr = VecDestroy(u);CHKERRQ(ierr);
233. ierr = PetscViewerDestroy(appctx.viewer1);CHKERRQ(ierr);
234. ierr = PetscViewerDestroy(appctx.viewer2);CHKERRQ(ierr);
235. ierr = VecDestroy(appctx.solution);CHKERRQ(ierr);
236.
237. /*
238. Always call PetscFinalize() before exiting a program. This routine
239. - finalizes the PETSc libraries as well as MPI
240. - provides summary and diagnostic information if certain runtime
241. options are chosen (e.g., -log_summary).
242. */
243. ierr = PetscFinalize();CHKERRQ(ierr);
244. return 0;
245. }
246. /* _____ */
247. #undef __FUNCT__
248. #define __FUNCT__ "InitialConditions"
249. /*
250. InitialConditions - Computes the solution at the initial time.

```

```
251.
252. Input Parameter:
253. u - uninitialized solution vector (global)
254. appctx - user-defined application context
255.
256. Output Parameter:
257. u - vector with solution at initial time (global)
258. */
259. PetscErrorCode InitialConditions(Vec u,AppCtx *appctx)
260. {
261. PetscScalar *u_localptr,h = appctx->h;
262. PetscErrorCode ierr;
263. PetscInt i;
264.
265. /*
266. Get a pointer to vector data.
267. - For default PETSc vectors, VecGetArray() returns a pointer to
268. the data array. Otherwise, the routine is implementation dependent.
269. - You MUST call VecRestoreArray() when you no longer need access to
270. the array.
271. - Note that the Fortran interface to VecGetArray() differs from the
272. C version. See the users manual for details.
273. */
274. ierr = VecGetArray(u,&u_localptr);CHKERRQ(ierr);
275.
276. /*
277. We initialize the solution array by simply writing the solution
278. directly into the array locations. Alternatively, we could use
279. VecSetValues() or VecSetValuesLocal().
280. */
```

```
281. for (i=0; i<appctx->m; i++) {
282. u_localptr[i] = PetscSinScalar(PETSC_PI*i*6.*h) +
283. 3.*PetscSinScalar(PETSC_PI*i*2.*h);
284. }
285.
286. /*
287. Restore vector
288. */
289. ierr = VecRestoreArray(u,&u_localptr);CHKERRQ(ierr);
290.
291. /*
292. Print debugging information if desired
293. */
294. if (appctx->debug) {
295. printf("initial guess vector\n");
296. ierr = VecView(u,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
297. }
298.
299. return 0;
300. }
301. /* _____ */
302. #undef __FUNCT__
303. #define __FUNCT__ "ExactSolution"
304. /*
305. ExactSolution - Computes the exact solution at a given time.
306.
307. Input Parameters:
308. t - current time
309. solution - vector in which exact solution will be computed
```

```

310. appctx - user-defined application context
311.
312. Output Parameter:
313. solution - vector with the newly computed exact solution
314. */
315. PetscErrorCode ExactSolution(PetscReal t,Vec solution,
316. AppCtx *appctx) {
317. PetscScalar *s_localptr,h = appctx->h,ex1,ex2,sc1,sc2,tc = t;
318. PetscErrorCode ierr;
319. PetscInt i;
320.
321. /*
322. Get a pointer to vector data.
323. */
324. ierr = VecGetArray(solution,&s_localptr);CHKERRQ(ierr);
325.
326. /*
327. Simply write the solution directly into the array locations.
328. Alternatively, we could use VecSetValues() or VecSetValuesLocal().
329. */
330. ex1 = PetscExpScalar(-36.*PETSC_PI*PETSC_PI*tc);
331. ex2 = PetscExpScalar(-4.*PETSC_PI*PETSC_PI*tc);
332. sc1 = PETSC_PI*6.*h; sc2 = PETSC_PI*2.*h;
333. for (i=0; i<appctx->m; i++) {
334. s_localptr[i] =
335. PetscSinScalar(sc1*(PetscReal)i)*ex1
336. + 3.*PetscSinScalar(sc2*(PetscReal)i)*ex2;
337. }
338.

```

```

339. /*
340. Restore vector
341. */
342. ierr = VecRestoreArray(solution,&s_localptr);CHKERRQ(ierr);
343. return 0;
344. }
345. /* _____ */
346. #undef __FUNCT__
347. #define __FUNCT__ "Monitor"
348. /*
349. Monitor - User-provided routine to monitor the solution computed at
350. each timestep. This example plots the solution and computes the
351. error in two different norms.
352.
353. This example also demonstrates changing the timestep via TSSetTimeStep().
354.
355. Input Parameters:
356. ts - the timestep context
357. step - the count of the current step (with 0 meaning the
358. initial condition)
359. time - the current time
360. u - the solution at this timestep
361. ctx - the user-provided context for this monitoring routine.
362. In this case we use the application context which contains
363. information about the problem size, workspace and the exact
364. solution.
365. */
366. PetscErrorCode Monitor(TS ts,PetscInt step,PetscReal time,
367. Vec u,void *ctx) {

```

```
368. /* user-defined application context */
369. AppCtx *appctx = (AppCtx*) ctx;
370. PetscErrorCode ierr;
371. PetscReal norm_2,norm_max,dt,dttol;
372. /*
373. View a graph of the current iterate
374. */
375. ierr = VecView(u,appctx->viewer2);CHKERRQ(ierr);
376.
377. /*
378. Compute the exact solution
379. */
380. ierr = ExactSolution(time,appctx->solution,appctx);CHKERRQ(ierr);
381.
382. /*
383. Print debugging information if desired
384. */
385. if (appctx->debug) {
386. printf("Computed solution vector\n");
387. ierr = VecView(u,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
388. printf("Exact solution vector\n");
389. ierr = VecView(appctx->solution,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
390. }
391.
392. /*
393. Compute the 2-norm and max-norm of the error
394. */
395. ierr = VecAXPY(appctx->solution,-1.0,u);CHKERRQ(ierr);
396. ierr = VecNorm(appctx->solution,NORM_2,&norm_2);CHKERRQ(ierr);
```

```
397. norm_2 = sqrt(appctx->h)*norm_2;
398. ierr = VecNorm(appctx->solution,NORM_MAX,&norm_max);CHKERRQ(ierr);
399.
400. ierr = TSGetTimeStep(ts,&dt);CHKERRQ(ierr);
401. ierr = PetscPrintf(PETSC_COMM_WORLD,
402. "Timestep %3D: step size =%-11g, time =%-11g,"
403. " 2-norm error =%-11g, max norm error =%-11g\n",
404. step,dt,time,norm_2,norm_max);CHKERRQ(ierr);
405. appctx->norm_2 += norm_2;
406. appctx->norm_max += norm_max;
407.
408. dttol = .0001;
409. ierr = PetscOptionsGetReal(PETSC_NULL,"-dttol",&dttol,
410. PETSC_NULL);CHKERRQ(ierr);
411. if (dt < dttol) {
412. dt *= .999;
413. ierr = TSSetTimeStep(ts,dt);CHKERRQ(ierr);
414. }
415.
416. /*
417. View a graph of the error
418. */
419. ierr = VecView(appctx->solution,appctx->viewer1);CHKERRQ(ierr);
420.
421. /*
422. Print debugging information if desired
423. */
424. if (appctx->debug) {
425. printf("Error vector\n");
```

```
426. ierr = VecView(appctx->solution,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
427. }
428.
429. return 0;
430. }
431. /* _____ */
432. #undef __FUNCT__
433. #define __FUNCT__ "RHSMatrixHeat"
434. /*
435. RHSMatrixHeat - User-provided routine to compute the right-hand-side
436. matrix for the heat equation.
437.
438. Input Parameters:
439. ts - the TS context
440. t - current time
441. global_in - global input vector
442. dummy - optional user-defined context, as set by TSetRHSJacobian()
443.
444. Output Parameters:
445. AA - Jacobian matrix
446. BB - optionally different preconditioning matrix
447. str - flag indicating matrix structure
448.
449. Notes:
450. Recall that MatSetValues() uses 0-based row and column numbers
451. in Fortran as well as in C.
452. */
453. PetscErrorCode RHSMatrixHeat(TS ts,PetscReal t,Mat *AA,Mat *BB,
454. MatStructure *str,void *ctx)
455. {
```

```

456. Mat A = *AA; /* Jacobian matrix */
457. /* user-defined application context */
458. AppCtx *appctx = (AppCtx*)ctx;
459. PetscInt mstart = 0;
460. PetscInt mend = appctx->m;
461. PetscErrorCode ierr;
462. PetscInt i, idx[3];
463. PetscScalar v[3], stwo = -2./(appctx->h*appctx->h), sone = -.5*stwo;
464.
465. /* -----
466. Compute entries for the locally owned part of the matrix
467. -----
468. */
469. /*
470. Set matrix rows corresponding to boundary data
471. */
472. mstart = 0;
473. v[0] = 1.0;
474. ierr = MatSetValues(A,1,&mstart,1,&mstart,v,INSERT_VALUES);CHKERRQ(ierr);
475. mstart++;
476.
477. mend--;
478. v[0] = 1.0;
479. ierr = MatSetValues(A,1,&mend,1,&mend,v,INSERT_VALUES);CHKERRQ(ierr);
480.
481. /*
482. Set matrix rows corresponding to interior data. We construct the
483. matrix one row at a time.
484. */
485. v[0] = sone; v[1] = stwo; v[2] = sone;

```

```

486. for (i=mstart; i<mend; i++) {
487. idx[0] = i-1; idx[1] = i; idx[2] = i+1;
488. ierr = MatSetValues(A,1,&i,3,idx,v,INSERT_VALUES);CHKERRQ(ierr);
489. }
490.
491. /* ----- *
492. Complete the matrix assembly process and set some options
493. ----- */
494. /*
495. Assemble matrix, using the 2-step process:
496. MatAssemblyBegin(), MatAssemblyEnd()
497. Computations can be done while messages are in transition
498. by placing code between these two statements.
499. */
500. ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
501. ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
502.
503. /*
504. Set flag to indicate that the Jacobian matrix retains an identical
505. nonzero structure throughout all timestepping iterations (although the
506. values of the entries change). Thus, we can save some work in setting
507. up the preconditioner (e.g., no need to redo symbolic factorization for
508. ILU/ICC preconditioners).
509. - If the nonzero structure of the matrix is different during
510. successive linear solves, then the flag DIFFERENT_NONZERO_PATTERN
511. must be used instead. If you are unsure whether the matrix
512. structure has changed or not, use the flag DIFFERENT_NONZERO_PATTERN.
513. - Caution: If you specify SAME_NONZERO_PATTERN, PETSc

```

```

514. believes your assertion and does not check the structure
515. of the matrix. If you erroneously claim that the structure
516. is the same when it actually is not, the new preconditioner
517. will not function correctly. Thus, use this optimization
518. feature with caution!
519. */
520. *str = SAME_NONZERO_PATTERN;
521.
522. /*
523. Set and option to indicate that we will never add a new nonzero location
524. to the matrix. If we do, it will generate an error.
525. */
526. ierr = MatSetOption(A,MAT_NEW_NONZERO_LOCATION_ERR,PETSC_TRUE);CHKERRQ(ierr);
527.
528. return 0;
529. }
530. /* _____ */
531. #undef __FUNCT__
532. #define __FUNCT__ "MYBCRoutine"
533. /*
534. Input Parameters:
535. ts - the TS context
536. t - current time
537. f - function
538. ctx - optional user-defined context, as set by TSetBCFunction()
539. */
540. PetscErrorCode MyBCRoutine(TS ts,PetscReal t,Vec f,void *ctx)
541. {
542. /* user-defined application context */

```

```
543. AppCtx *appctx = (AppCtx*)ctx;
544. PetscErrorCode ierr, m = appctx->m;
545. PetscScalar *fa;
546.
547. ierr = VecGetArray(f,&fa);CHKERRQ(ierr);
548. fa[0] = 0.0;
549. fa[m-1] = 0.0;
550. ierr = VecRestoreArray(f,&fa);CHKERRQ(ierr);
551. printf("t=%g\n",t);
552.
553. return 0;
554. }
```

# OpenMP

## GTP 10. [PNT] OpenMP PNT

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

## GTP 11. [MCARLO] Pi MonteCarlo OpenMP

### *Calculando PI por Montecarlo con OpenMP*

- El programa secuencial de más abajo (478) calcula  $\pi$  con la estrategia Montecarlo (ver página 180).
- Para generar los números rando debemos usar una función generadora de números seudo-aleatorios que sea reentrante, por ejemplo la `drand48_r()` de la librería GNU Libc.
- La secuencia de llamada típica para `drand48_r()` es

```
1. // This buffer stores the data for the
2. // random number generator
3. drand48_data buffer;
4. // This buffer must be initialized first
5. memset(&buffer, '\0', sizeof(struct drand48_data));
6. // Randomize the generator
7. srand48_r(time(0),&buffer);
8. ...
9. double x;
10. // Generate a random numbers x
11. drand48_r(&buffer,&x);
```

- Nota: la variable `buffer` (es decir el bufer interno del que es el estado del generador aleatorio) debe ser *privado* y también la inicialización del

mismo debe ser local en cada thread y con una semilla diferente para cada thread (usar el thread id). thread.

- La consigna es paralelizar este programa con OpenMP.
- Calcular el speedup para diferente número de procesadores.
- probar diferentes tipos de scheduling (static, dynamic, guided...). Probar diferentes tamaños de chunk y determinar una buena combinación de scheduling y chunk. Discutir.



[Descargar: ./example/pimc.cpp]

```
1. #include <cassert>
2. #include <cstdio>
3. #include <cstdlib>
4. #include <ctime>
5. #include <cstring>
6. #include <cmath>
7.
8. #include <stdint.h>
9. #include <inttypes.h>
10. #include <unistd.h>
11.
12. #include <omp.h>
13.
14. using namespace std;
15.
16. int main(int argc, char **argv) {
17. uint64_t chunk=100000000, inside=0;
```

```
18. double start = omp_get_wtime();
19. drand48_data buffer; // This buffer stores the data for the
20. // random number generator
21. // This buffer must be initialized first
22. memset(&buffer, '\0', sizeof(struct drand48_data));
23. // Then we randomize the generator
24. srand48_r(time(0),&buffer);
25. int nchunk=0;
26. while (1) {
27. for (uint64_t j=0; j<chunk; j++) {
28. double x,y;
29. // Generate a pair of random numbers x,y
30. drand48_r(&buffer,&x);
31. drand48_r(&buffer,&y);
32. inside += (x*x+y*y<1-0);
33. }
34. double
35. now = omp_get_wtime(),
36. elapsed = now-start;
37. double npoints = double(chunk);
38. double rate = double(chunk)/elapsed/1e6;
39. nchunk++;
40. double mypi = 4.0*double(inside)/double(chunk)/nchunk;
41. printf("PI %f, error %g, comp %g points, "
42. "elapsed %fs, rate %f[Mpoints/s]\n",
43. mypi,fabs(mypi-M_PI),npoints,elapsed,rate);
44. start = now;
```

```
45. }
46. return 0;
47. }
```

## GTP 12. [POLMAT] Polinomio de matrices con OpenMP

### Calculando un polinomio de matrices con OpenMP

- Dado un polinomio  $p(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$  y una matriz cuadrada  $A \in \mathbb{R}^{m \times m}$ , definimos la aplicación del polinomio a la matriz  $p(A)$  como

$$p(A) = c_0I + c_1A + c_2A^2 \dots + c_nA^n$$

**CONSIGNA:** Dado un polinomio  $p(x)$  representado como un VectorXd `c(n+1)` y los coeficientes de la matriz como MatrixXd `a(m,m)` escribir una función

`void apply_poly(VectorXd &c, MatrixXd &A, MatrixXd &PA);` usando OpenMP, que devuelve por `PA` la matriz resultado.

**Hints:**

- Usar la regla de Hörner

`px = 0; for (int j=n; j>=0; j--) px = px*x+c[j];` (Nota: `px` es  $p(x)$ ). Este código es para escalares, el ejercicio consiste en implementarlo para matrices).

- **Algoritmo interlaced:** Se sugiere usar un algoritmo en el cual el thread de

rango  $j$  calcula la contribución de los términos  $c_k A^k$  para los  $k$  tales que:  $k \bmod P = j$  (es decir una distribución *interlaced*) donde  $P$  es el número de threads,

$$\begin{aligned} Y_K &= c_K A^K + c_{P+K} A^{P+K} + c_{2P+K} A^{2P+K} + \dots \\ &= A^K (c_K + c_{P+K} A' + c_{2P+K} A'^2 + \dots) \end{aligned}$$

Después por supuesto hay que hacer una reducción por la suma de los  $Y_K$  para obtener el  $p(A)$ . Las contribuciones en cada procesador se pueden reescribir como un polinomio en  $A^P$ . El siguiente en Octave calcula la contribución en el procesador **K**, (**numprocs** es el número de procesadores).

```

1. k = N-rem(N,P)+K;
2. if k>N; k -= P; endif
3.
4. ## Apply Horner's rule
5. xp = x^P;
6. while k>=0;
7. yproc = yproc*xp+coef(k+1)*Id;
8. k -= P;
9. endwhile
10.
11. ## Apply the coefficient factor x^k
12. yproc *= matpow(x,K);

```

Las primeras dos líneas calculan el índice del último término en el polinomio que se debe calcular en este pro. Por ejemplo si el polinomio tiene  $N = 105$  términos y  $P = 10$  y estoy en el thread  $K = 4$  entonces en este thread se deben calcular los términos  $4, 14, 24, \dots, 104$ . Una posibilidad sería hacer justamente eso, empezar con  $k = K$  y sumar  $P$  hasta que no se pase de  $N$ :

1. `k = K;`
2. `while k<=N; k += P; endwhile`
3. `k -= P;`

Otra posibilidad es a la inversa, empezar desde un  $k$  que esté garantizado que sea congruente con  $K$  modulo  $P$ , y  $k \geq N$  y eventualmente empezar a restar  $P$ . Puede verse que las dos líneas del código de más arriba realizan exactamente eso:

1. `k = N-rem(N,P)+K;`
2. `if k>N; k -= P; endif`

Por ejemplo en el caso de  $P = 2$  se calculan los pares en el thread 0 y los impares en el thread 1.

- Con este algoritmo cada procesador sólo debe realizar  $O(N/P)$  productos de matrices. Para calcular la matriz  $A^P$  puede usarse el siguiente algoritmo que hace el cálculo con  $O(\log_2(P))$  productos: Si  $P$

es par calculamos  $Z = A^{P/2}$  (recursivamente) y después hacemos  $A^P = ZZ$ . Si  $P$  es impar entonces  $A^P = ZZ A$ , donde  $Z = A^{(P-1)/2}$ .

La siguiente función de Octave implementa el mencionado algoritmo.

```

1. function y = matpow(x,n);
2. if n==0;
3. y = eye(size(x));
4. return;
5. elseif n==1;
6. y = x;
7. return;
8. endif
9. r = rem(n,2);
10. m = (n-r)/2;
11. y = matpow(x,m);
12. y = y*y;
13. if r; y = y*x; endif
14. endfunction

```

(Se incluye una versión en C++ en el archivo adjunto [hornere.cpp](#)).

- Para implementar en paralelo el algoritmo interlaced se sugiere la siguiente estrategia. Escribir una función [apply\\_poly\\_omp\(c,A,PA\)](#); que internamente extrae los coeficientes que deben usarse para la contribución local y llama al [apply\\_poly](#) secuencial.
  - ▷ Calcular  $A^P$  y  $A^K$  usando [matpow\(\)](#).

- ▷ Extraer los coeficientes de `coefs` que corresponden a este thread:

```

1 vector<double> clocal;
2 j=tid;
3 while (j<=N) { clocal.push_back(j); j+=P; }
```

- ▷ Convertir el `clocal` a Eigen (`VectorXd`)
- ▷ Evaluar el polinomio `clocal` en  $A^P$
- ▷ Multiplicar el resultado por  $A^K$ .
- ▷ Hacer la suma sobre todos los threads de los  $Y^K$ .
- Como verificación podemos usar como  $p(x)$  una expresión truncada del desarrollo de  $1/(1 - x)$  (ver `test3()` en `hornere.cpp`):

$$\frac{1}{1-x} \approx 1 + x + x^2 + x^3 + \dots + x^n + \dots \quad (17)$$

para calcular la inversa de una matriz  $A$ . Efectivamente, si ponemos  $X = D^{-1}(D - A)$ , donde  $D$  es la parte diagonal de  $A$ , entonces tenemos  $A^{-1} = (I - X)^{-1}D^{-1}$ . Por ejemplo si  $A$  es la matriz de 5x5 definida como

1. 2.1 -1.0 -0.0 -0.0 -1.0
2. -1.0 2.1 -1.0 -0.0 -0.0
3. -0.0 -1.0 2.1 -1.0 -0.0
4. -0.0 -0.0 -1.0 2.1 -1.0

5. -1.0 -0.0 -0.0 -1.0 2.1

debe dar

1. 2.3775 1.9964 1.8149 1.8149 1.9964
2. 1.9964 2.3775 1.9964 1.8149 1.8149
3. 1.8149 1.9964 2.3775 1.9964 1.8149
4. 1.8149 1.8149 1.9964 2.3775 1.9964
5. 1.9964 1.8149 1.8149 1.9964 2.3775

- Otra verificación: usar como  $p(x)$  una expresión truncada del desarrollo de la exponencial (ver `test2()` en `hornere.cpp`):

$$e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \dots + \frac{x^n}{n!} + \dots \quad (18)$$

y verificar que si  $A = I_{2 \times 2}$ , entonces  $e^A = e I_{2 \times 2}$ , y si

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (19)$$

entonces

$$e^A = \begin{bmatrix} 1.5431 & 1.1752 \\ 1.1752 & 1.5431 \end{bmatrix} \quad (20)$$

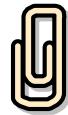
- **Algoritmo por chunks contiguos:** Otra posibilidad es que cada procesador calcule un rango del polinomio  $[K_1, K_2)$  es decir la contribución

$$\begin{aligned} Y_K &= \sum_{k=K_1}^{K_2-1} c_k A^k, \\ &= A^{K_1} \sum_{k=0}^{K_2-K_1-1} c_{K_1+k} A^k, \end{aligned}$$

Es decir que la sumatoria es un polinomio que se puede evaluar por la regla de Hörner y el factor  $A^{K_1}$  se puede evaluar en  $\mathcal{O}(\log_2(N))$  multiplicaciones. Este algoritmo puede ser un poco más simple para la programación pero necesita evaluar el factor  $A^{k-1}$  que involucra potencias mayores que el otro algoritmo. Por ejemplo, si  $N = 100$  y  $P = 4$  entonces con la versión interlaced tenemos que calcular 25 productos en cada procesador para calcular la serie y 2 productos para calcular  $A^P$ . En cambio si usamos el algoritmo por chunks contiguos tenemos que calcular también 25 productos para la suma pero hasta 7 productos para calcular  $A^{75}$  en el último procesador.

- Para dividir los términos del polinomio entre los threads se puede usar un

código como el siguiente. Básicamente si tenemos un polinomio de grado  $N$  entonces hay  $ncoef=N+1$  coeficientes, y los podemos dividir en partes casi iguales asignando  $ncoef/nthreads$  a todos. Si  $ncoefs$  no es múltiplo de  $nthreads$  van a sobrar  $rem=ncoef \% nthreads$  ( $rem$  por remainder, resto) por lo tanto podemos agregar a los primeros  $rem$  threads 1 coeficiente. Por lo tanto la cantidad de coeficientes a asignar al thread  $tid$  es  $nhere=(ncoef/nthreads)+(tid<ncoef \% nthreads)$ . Notar que la expresión  $tid< ncoef \% nthreads$  da usa el hecho de que en C/C++ una expresión de comparación retorna 1 cuando da verdadero y 0 cuando es falso.



[Descargar: [./example/plmsplit.cpp](#)]

```

1. // Order of polynomial
2. int N = 22;
3. // Number of coefs
4. int ncoef=N+1;
5. vector<double> coefs(N+1);
6. printf("N%d, coefs: ",N);
7. // Fill coefs with random values
8. for (int j=0; j<ncoef; j++) {
9. coefs[j] = rand() %100;
10. printf("c[%d]=%.1f ",j,coefs[j]);
11. }
12. printf("\n");
13. // Number of threads

```

```
14. int nthreads = 5;
15. // This array stores the range for all threads.
16. // The range for thread TID is [kstart[tid],kstart[tid+1]]
17. vector<int> kstart(nthreads+1,0);
18. // It is computed as the “cumsum” of the local sizes “nhere”
19. // The local sizes are either ncoef/nthreads or ncoef/nthreads+1
20. kstart[0] = 0;
21. int ncfloc = ncoef/nthreads;
22. int rem = ncoef%nthreads;
23. printf("ncfloc %d, remainder %d\n",ncfloc,rem);
24. for (int tid=0; tid<nthreads; tid++) {
25. // For the first rem threads we add 1
26. int nhere = ncfloc+(tid<rem);
27. // Do the cumsum
28. kstart[tid+1] = kstart[tid]+nhere;
29. // [K1,K2) is the range for thread “tid”
30. int K1=kstart[tid],K2=kstart[tid+1];
31. printf("thread=%d, K1=%d, K2=%d, ",
32. tid,K1,K2);
33. // Print the coefs for the thread
34. printf("coefs: ");
35. for (int k=K1; k<K2; k++)
36. printf("c[%d]=%.1f ",k,coefs[k]);
37. printf("\n");
```

- Se adjunta un programa en C++ `horner.cpp` (secuencial!!) que calcula el `apply_poly()`. Tiene utilidades para manipular matrices y el algoritmo

básico de Hörner. Resuelve el problema de calcular la inversa y la exponencial de una matriz.

- Para juntar (reducción por la suma) de las contribuciones parciales  $Y_K$  en el resultado final  $P(A)$  hay al menos dos posibilidades (para evitar la **race condition**):

- ▷ Usar una región crítica.
- ▷ Usando las funciones del **runtime environment** crear (antes de entrar al bloque paralelo) un vector de matrices:

`vector<MatrixXd> vmat(P,MatrixXd(n,n));`. Cada thread acumula en `vmat[tid]` y finalmente después de salir del bloque paralelo hacemos la reducción simplemente sumando sobre el vector.

Probar ambas estrategias.

- Reproducir la versión con región crítica pero usando semáforos (`omp_lock_t`).
- Probar los diferentes scheduling (static, dynamic, guided) con diferentes chunks.
- Comparar la versión con paralelización hecha por el usuario (el algoritmo explicado arriba) con la paralelización interna de Eigen. Es decir probar con bloque paralelo de `nthreads>1`, y threads de Eigen=1, y viceversa.

- Graficar el speedup obtenido en función del número de threads y discutir.
- **ATENCION: Eigen y OpenMP.** Eigen es de por sí **multithreaded** así que si uno activa OpenMP (es decir compila con `-fopenmp`) ya de por sí se obtendría un speedup probablemente bueno. Esto interfiere con el cálculo del speedup ya que incluso corriendo el algoritmo secuencial que presentamos y que está en el programa template obtendríamos un speedup.

Para hacer el GTP entonces la idea es deshabilitar el multithreading dentro de Eigen. Para ello hay que hacer

```
Eigen::setNbThreads(1); // [*]
```

Entonces resumiendo, cuando van a calcular el speedup  $S_n = T_1/T_n$

- ▷ Para tener  $T_1$ : hay que usar el algoritmo secuencial y asegurarse que no esté corriendo con Eigen multithreaded (es decir hacer la llamada de arriba [\*]), o bien usar la versión paralela pero poniendo `OMP_NUM_THREADS=1`.
- ▷ Para calcular el  $T_n$ : hay que usar el ejemplo paralelo y forzar que Eigen no haga multithreading (usar [\*]).
- **Nota:** En versiones anteriores (e.g. 3.0.6) la llamada es  
`Eigen::internal::setNbThreads(1);`



[Descargar: ./example/hornere.cpp]

```
1. #include <iostream>
2. #include <Eigen/Dense>
3.
4. using namespace std;
5. using Eigen::MatrixXd;
6. using Eigen::VectorXd;
7.
8. // Returns j mod n. Note that this C/C++ provides
9. // already the% operator. This is equivalente to 'modulo' for
10. // positive 'j' but not for negative values.
11. int modulo(int j,int n) {
12. int r = j %n;
13. if (r<0) r += n;
14. return r;
15. }
16.
17. void matpow2(MatrixXd &A,MatrixXd &Ak,int k) {
18. // Brute force approach
19. int m = A.rows();
20. assert(A.cols()==m);
21. Ak = MatrixXd::Identity(m,m);
22. for (int j=0; j<k; j++) Ak *= A;
23. }
24.
25. void matpow(MatrixXd &A,MatrixXd &Ak,int k) {
26. int m = A.rows();
```

```
27. assert(A.cols()==m);
28. if (k==0) Ak = MatrixXd::Identity(m,m);
29. else if (k==1) Ak = A;
30. else {
31. MatrixXd Z;
32. int r = k%2;
33. int k2 = k/2;
34. matpow(A,Z,k2);
35. Ak = Z*Z;
36. if (r) Ak *= A;
37. }
38. }
39.
40. void apply_poly(VectorXd &coefs,
41. MatrixXd &X,
42. MatrixXd &PX) {
43. int m = X.rows();
44. assert(X.cols()==m);
45. PX = 0*X;
46. MatrixXd Id = MatrixXd::Identity(m,m);
47. int N = coefs.size();
48. for (int k=N-1; k>=0; k--)
49. PX = PX*X + coefs(k)*Id;
50. }
51.
52. void test1() {
53. int m=10;
54. MatrixXd A(m,m),Akl,Akbf;
55. for (int j=0; j<m; j++)
```

```
56. for (int k=0; k<m; k++)
57. A(j,k) = 1+(j-k)/10.0;
58. cout << A << endl;
59. int k=5;
60. matpow(A,Akl,k);
61. cout << "A^" << k << " (Akl=log2 algo): "
62. << endl << Akl << endl;
63.
64. matpow2(A,Akbf,k);
65. cout << "A^" << k << " (Akbf=brute force algo): "
66. << endl << Akbf << endl;
67. MatrixXd error;
68. error = Akl-Akbf;
69. cout << "| |Akl-Akbf| |: " << error.norm() << endl;
70. }
71.
72. void test2() {
73. // Computes the exponential of a matrix
74. // The coefficients are 1,1,1/2,1/6,1/24,...,1/n!
75. int N=2;
76. MatrixXd X(2,2),PX;
77. X << 0, 1, 1 ,0;
78. int M=50;
79. VectorXd coefs(M);
80. coefs[0] = 1;
81. for (int j=1; j<M; j++) coefs(j) = coefs(j-1)/j;
82. apply_poly(coefs,X,PX);
83. cout << "X" << endl << X << endl << endl;
84. cout << "PX (=exp(X)): " << endl << PX << endl;
```

```
85. }
86.
87. #define MP(x) cout << #x ":" << endl << x << endl
88.
89. void test3() {
90. int m=5;
91. MatrixXd A(m,m), D, Dinv, X, PX, invA;
92. VectorXd d;
93. // Computes the inverse of A matrix
94. // using the series for 1/(1-X)
95. for (int j=0; j<m; j++) {
96. A(j,j) = 2.1;
97. A(j,modulo(j+1,m)) = -1.0;
98. A(j,modulo(j-1,m)) = -1.0;
99. }
100.
101. MP(A);
102. // Makes: X = inv(D) * (D-A)
103. X = A;
104. for (int j=0; j<m; j++) {
105. double ajj = A(j,j);
106. for (int k=0; k<m; k++) X(j,k) = -A(j,k)/ajj;
107. X(j,j) = 0.0;
108. }
109. MP(X);
110. int M=300;
111. VectorXd coefs(M);
112. for (int j=0; j<M; j++) coefs(j) = 1.0;
113. apply_poly(coefs,X,PX);
```

```
114. // Makes: inv(A) = PX*inv(D)
115. MP(PX);
116. invA = PX;
117. for (int j=0; j<m; j++)
118. for (int k=0; k<m; k++) invA(j,k) = PX(j,k)/A(k,k);
119. MP(invA);
120. }
121.
122. int main() {
123. test1();
124. test2();
125. test3();
126. return 0;
127. }
```

## GTP 13. [MEMBRANE] Membranas 3D con contacto



[Descargar: [./hpcmc-examples/nlspring.cpp](#)]

```
1. static char help[] =
2. "Newton's method to solve a 2D network of springs.\n";
3.
4. #include <cstdlib>
5. #include <iostream>
6. #include <vector>
7. #include <map>
8. #include <Eigen/Dense>
9. #include "petscsnes.h"
10.
11. using namespace std;
12. using Eigen::MatrixXd;
13. using Eigen::VectorXd;
14.
15. /*
16. User-defined routines
17. */
18. int resfun(SNES,Vec,Vec,void*);
19. int jacfun_fd(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
20. int jacfun_anly(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
21.
22. // Regularized Heaviside function based on COS, in interval [0,b]
```

```
23. double regheavis(double x,double b) {
24. double xi=x/b;
25. return xi<0.0? 0.0 : xi>1.0? 1.0 : 0.5*(1.0-cos(M_PI*xi));
26. }
27.
28. // Regularized Heaviside function based on COS, in interval [a,b]
29. double regheavis(double x,double a,double b) {
30. return regheavis(x-a,b-a);
31. }
32.
33. //--<*>--<*>--<*>--<*>--<*>
34. class spring_t {
35. public:
36. virtual double force(double len)=0;
37. };
38.
39. //--<*>--<*>--<*>--<*>--<*>
40. class lin_spring_t : public spring_t {
41. public:
42. double force(double len) {
43. // return sqrt(len);
44. return len;
45. }
46. } lin_spring;
47.
48. //--<*>--<*>--<*>--<*>--<*>
49. class force_t {
50. public:
51. virtual void force(VectorXd &x,VectorXd &force)=0;
```

```
52. };
53.
54. //--<*>--<*>--<*>--<*>--<*>
55. class grav_t : public force_t {
56. public:
57. double w;
58. void force(VectorXd &x,VectorXd &force) {
59. force(2) = -w;
60. }
61. grav_t() : w(0.0) {}
62. } grav;
63.
64. //--<*>--<*>--<*>--<*>--<*>
65. class sphere_force_t : public force_t {
66. public:
67. double R,dr,k;
68. VectorXd xc;
69. void force(VectorXd &x,VectorXd &force) {
70. VectorXd dx = x-xc;
71. // Distance to center
72. double r = dx.norm();
73. force = -(1.0-regheavis(r,0,R))*k*dx/r;
74. }
75. } sphere_force;
76.
77. double sqr(double x) { return x*x; }
78.
79. //--<*>--<*>--<*>--<*>--<*>
80. class ellipsoid_t : public force_t {
```

```
81. public:
82. double dr,k;
83. VectorXd axis,xc;
84. void force(VectorXd &x,VectorXd &force) {
85. VectorXd dx = x-xc;
86. // Distance to center
87. const int ndim=3;
88. VectorXd nor(ndim);
89. double rho=0.0;
90. for (int j=0; j<ndim; j++) {
91. nor(j) = dx(j)/sqr(axis(j));
92. rho += sqr(dx(j))/axis(j));
93. }
94. // force = -(1.0-regheavis(rho,1-rho,1+rho))*k*nor/dx.norm();
95. force = -(1.0-regheavis(rho,1-rho,1+rho))*k*nor/dx.norm();
96. }
97. } ellipsoid;
98.
99. //:-<*>-:-<*>-:-<*>-:-<*>-:-<*>
100. class field_t : public force_t {
101. public:
102. void force(VectorXd &x,VectorXd &force) {
103. // Attraction to the z=0.1 plane
104. force.fill(0.0);
105. force(2) = 1*(x(2)-0.1);
106. }
107. } field;
108.
109. //:-<*>-:-<*>-:-<*>-:-<*>-:-<*>
```

```
110. struct snes_ctx_t {
111. // N := number of segments per side
112. // N1 := N+1, number of nodes per side
113. // ndim := number of dimensions
114. // nnod := number of nodes
115. // nelem := number of elements (quads)
116. // neq := number of equations
117. // nstep := number of steps to solve
118. int N,N1,ndim,nnod,nelem,neq,nstep,nsteprlx,nstepz;
119. // h := Step size, length of the segments in the ref mesh
120. // L0 := length of side in the ref mesh
121. // DL := elongation of right side
122. // p := power in the stiffness law
123. double h,L0,DL,p;
124. // Call back function that defines relative stiffness coeff
125. double stiff(double x) { return 1.0; }
126. // Initial position of the nodes
127. vector<double> xref;
128. // (node,dof) -> fixed displs
129. map< pair<int,int>,double> bcfixed;
130. // Vector and scatter to gather all values in the global vector
131. Vec uloc;
132. VecScatter scat;
133. // Utility function to set the value of a node and dof
134. // pair to a value
135. void set_bc(int node,int dof,double val) {
136. bcfixed[pair<int,int>(node,dof)] = val;
137. }
```

```
138. // Spring
139. spring_t *springp;
140. // Force field
141. force_t *forcep;
142. // Initialize the problem
143. void init();
144. // Set the initial position vector
145. double xinit(int node,int dof);
146. // Set the boundary conditions
147. void setup_step(int step);
148. // Compute the resfun for the SNES
149. int resfun(Vec x,Vec f);
150. // Compute the jacfun for the SNES
151. int jacfun(Vec x0,Mat* jac);
152. // Ctor
153. snes_ctx_t();
154. // Dtor
155. ~snes_ctx_t();
156. };
157.
158. //:-<*>-:<*>-:<*>-:<*>-:<*>
159. snes_ctx_t::snes_ctx_t() : uloc(NULL), scat(NULL),
160. springp(NULL), forcep(NULL) {}
161.
162. //:-<*>-:<*>-:<*>-:<*>-:<*>
163. snes_ctx_t::~snes_ctx_t() {
164. if (uloc) {
165. VecDestroy(&uloc);
```

```
166. VecScatterDestroy(&scat);
167. }
168. }
169.
170. //---<*>---<*>---<*>---<*>---<*>
171. void snes_ctx_t::init() {
172.
173. ndim = 3;
174. N = 120;
175. L0 = 1;
176. p = 1;
177. nsteprlx = 1;
178. nstepz = 400;
179. nstep = nsteprlx*nstepz;
180. grav_t &g = grav;
181. g.w = -0.01;
182. // forcep = &g;
183. sphere_force_t &f = sphere_force;
184. f.R = 0.25;
185. f.xc.resize(ndim);
186. cout << "xc " << f.xc.size() << endl;
187. f.xc << 0.5*L0,0.5*L0,-1.1*f.R;
188. f.dr= 0.5*f.R;
189. f.k = 0.01;
190. forcep = &f;
191. // Ellipsoid
192. ellipsoid_t &e = ellipsoid;
193. e.axis.resize(ndim);
194. e.xc.resize(ndim);
```

```
195. double R = 0.25;
196. e.axis << R,R,R;
197. e.xc << 0.5*L0,0.5*L0,0.5*R;
198. e.dr = 0.05;
199. e.k = 10;
200.
201. // Special force field for debugging
202. // forcep = &field;
203. #if 0
204. // Explore the shape of the force field
205. VectorXd x(ndim),ff(ndim);
206. x << 0.5,0.5,0;
207. int N = 300;
208. for (int k=0; k<N; k++) {
209. x(2) = -0.5+1*double(k)/N;
210. forcep->force(x,ff);
211. printf("z%f f%f%f%f\n",x(2),ff(0),ff(1),ff(2));
212. }
213. exit(0);
214. #endif
215. N1 = N+1;
216. h = 1.0/N;
217. DL = 0.5;
218. springp = &lin_spring;
219.
220. nnod = N1*N1;
221. nelem = N*N;
222. neq = nnod*ndim;
223.
```

```
224. xref.resize(neq);
225. for (int j=0; j<N1; j++) {
226. double x=j*h;
227. for (int i=0; i<N1; i++) {
228. double y=i*h;
229. int node = i*N1+j;
230. xref[node*ndim+0] = x;
231. xref[node*ndim+1] = y;
232. xref[node*ndim+2] = 0.0;
233. }
234. }
235. }
236.
237. //--:<*>--:<*>--:<*>--:<*>--:<*>
238. // Set the boundary conditions
239. void snes_ctx_t::setup_step(int step) {
240. // Boundaries BCs
241. // x=0 side to DX=0, x=1 side to DX=0.1, (y=0,y=1 → DY=0)
242. // Nodes are numbered first by X and then by Y, so node
243. // (i*h,j*h) → is node=i*(N+1)+j
244. #if 1
245. for (int j=0; j<=N; j++) {
246. // Right side set to DX=0,DY=0
247. int node = j*N1;
248. set_bc(node,0,0.0);
249. set_bc(node,2,0.0);
250. // Left side set to DX=DL,DY=0
251. node = j*N1+N;
```

```
252. set_bc(node,0,L0);
253. set_bc(node,2,0.0);
254. // Bottom side
255. node = j;
256. set_bc(node,1,0.0);
257. set_bc(node,2,0.0);
258. // Bottom side
259. node = N*N1+j;
260. set_bc(node,1,L0);
261. set_bc(node,2,0.0);
262. }
263. sphere_force_t &s = sphere_force;
264. if (step%nsteprlx==0) {
265. // s.dr= 0.5*s.R;
266. s.dr= 0.1*s.R;
267. s.k = 10;
268. // double zc = s.xc(2)+0.005*s.R;
269. double zc = s.xc(2) + (4*s.R)/nstepz;
270. if (1 || zc<0.145) s.xc(2) = zc;
271.
272. double
273. Rorbit=0.0,
274. Dt=0.01,
275. T=1,
276. omega=2*M_PI/T,
277. time = (step/nsteprlx)*Dt;
278. s.xc(0) = 0.5*L0+Rorbit*cos(omega*time);
279. s.xc(1) = 0.5*L0+Rorbit*sin(omega*time);
```

```
280. } else {
281. s.dr *= 0.9;
282. s.k *= 2;
283. }
284. printf("setup step%d, dr%g, k%g, xc%f\n",
285. step,s.dr,s.k,s.xc(2));
286. #elif 0
287. double xi = -0.5+double(step)/nstep;
288. for (int j=0; j<=N; j++) {
289. // Right side set to DX=0,DY=0
290. int node = j*N1;
291. set_bc(node,0,0.0);
292. set_bc(node,1,xinit(node,1));
293. set_bc(node,2,0.0);
294. // Left side set to DX=DL,DY=0
295. node = j*N1+N;
296. set_bc(node,0,L0+xi*DL);
297. set_bc(node,1,xinit(node,1));
298. set_bc(node,2,0.0);
299. }
300. #else
301. VectorXd X(2),XC(2);
302. XC.fill(0.5*L0);
303. for (int node=0; node<nnod; node++) {
304. double
305. x = xinit(node,0),
306. y = xinit(node,1);
307. double tol=1e-6;
```

```
308. if (x<tol || x>L0-tol || y<tol || y>L0-tol) {
309. X << x,y;
310. X -= XC;
311. X /= X.norm();
312. X += XC;
313. set_bc(node,0,X(0));
314. set_bc(node,1,X(1));
315. set_bc(node,2,0.0);
316. }
317. }
318. #endif
319. }
320.
321. //:-<*>-:<*>-:<*>-:<*>-:<*>
322. double snes_ctx_t::xinit(int node,int dof) {
323. return xref[node*ndim+dof];
324. }
325.
326. // Given a distributed PETSc vector 'vec' gather all the
327. // ranges in all processor in a full vector of doubles
328. // 'values'. This full vector is available in all the
329. // processors.
330. // WARNING: this may be inefficient and non
331. // scalable, it is just a dirty trick to have access to all
332. // the values of the vectors in all the processor.
333. // Usage:
334. // Vec v;
335. // // ... create and fill v eith values at each processor
336. // // ... do the Assembly
```

```
337. // vector<double> values;
338. // vec_gather(MPI_COMM_WORLD,v,values);
339. // ... now you have all the elements of 'v' in 'values'
340. void vec_gather(MPI_Comm comm,Vec v,vector<double> &values) {
341. // n: global size of vector
342. // nlocal: local (PETSc) size
343. int n,nlocal;
344. // Get the global size
345. VecGetSize(v,&n);
346. // Resize the local buffer
347. values.clear();
348. values.resize(n,0.0);
349. // Get the local size
350. VecGetLocalSize(v,&nlocal);
351.
352. // Gather all the local sizes in order to compute the
353. // counts and displs for the Allgatherv
354. int size, myrank;
355. MPI_Comm_rank(comm,&myrank);
356. MPI_Comm_size(comm,&size);
357. vector<int> counts(size),displs(size);
358. MPI_Allgather(&nlocal,1,MPI_INT,
359. &counts[0],1,MPI_INT,comm);
360. displs[0]=0;
361. for (int j=1; j<size; j++)
362. displs[j] = displs[j-1] + counts[j-1];
363.
364. // Get the internal values of the PETSc vector
```

```
365. double *vp;
366. VecGetArray(v,&vp);
367. // Do the Allgatherv to the local vector
368. MPI_Allgatherv(vp,nlocal,MPI_DOUBLE,
369. &values[0],&counts[0],&displs[0],MPI_DOUBLE,comm);
370. // Restore the array
371. VecRestoreArray(v,&vp);
372. }
373.
374. #undef __FUNCT__
375. #define __FUNCT__ "main"
376. int main(int argc,char **argv) {
377. SNES snes; /* nonlinear solver context */
378. Vec x,r; /* solution, residual vectors */
379. Mat J; /* Jacobian matrix */
380. int ierr,its,size;
381. snes_ctx_t ctx;
382.
383. PetscInitialize(&argc,&argv,(char *)0,help);
384.
385. ctx.init();
386.
387. ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
388. if (size != 1) SETERRQ(PETSC_COMM_WORLD,1,
389. "This is a uniprocessor example only!");
390.
391. // Create nonlinear solver context
392. ierr = SNESCreate(PETSC_COMM_WORLD,&snes); CHKERRQ(ierr);
```

```

393. ierr = SNESSetType(snes,SNESLS); CHKERRQ(ierr);
394. SNESSetFromOptions(snes);
395. double abstol, rtol, stol;
396. int maxit, maxf;
397. SNESGetTolerances(snes,&abstol,&rtol,&stol,&maxit,&maxf);
398. SNESMonitorSet(snes,SNESMonitorDefault,NULL, NULL);
399.
400. PetscPrintf(PETSC_COMM_WORLD,
401. "atol=%g, rtol=%g, stol=%g, maxit=%D, maxf=%D\n",
402. (double)abstol,(double)rtol,(double)stol,maxit,maxf);
403.
404. // Create vectors for solution and nonlinear function
405. ierr = VecCreateSeq(PETSC_COMM_SELF,ctx.neq,&x);CHKERRQ(ierr);
406. ierr = VecDuplicate(x,&r); CHKERRQ(ierr);
407.
408. #if 0
409. // Create the scatter and the target vector
410. VecScatterCreateToZero(u,ctx.scat,ctx.uloc);
411. // Do the scatter
412. VecScatterBegin(scat,u,uloc,INSERT_VALUES,SCATTER_FORWARD);
413. VecScatterEnd(scat,u,uloc,INSERT_VALUES,SCATTER_FORWARD);
414. #endif
415.
416. double *xx;
417. int ndim = ctx.ndim;
418. ierr = VecGetArray(x,&xx); CHKERRQ(ierr);
419. for (int j=0; j<ctx.nnod; j++)
420. for (int k=0; k<ndim; k++)
421. xx[j*ndim+k] = ctx.xinit(j,k);

```

```
422. ierr = VecRestoreArray(x,&xx); CHKERRQ(ierr);
423.
424. ierr = MatCreateMPIAIJ(PETSC_COMM_SELF,PETSC_DECIDE,
425. PETSC_DECIDE,ctx.neq,ctx.neq,
426. 27,NULL,0,NULL,&J);CHKERRQ(ierr);
427.
428. ierr = SNESSetFunction(snes,r,resfun,&ctx); CHKERRQ(ierr);
429. // ierr = SNESSet Jacobian(snes,J,J,jacfun_fd,&ctx); CHKERRQ(ierr);
430. ierr = SNESSet Jacobian(snes,J,J,jacfun_anly,&ctx); CHKERRQ(ierr);
431.
432. #if 0
433. // resfun(snes,x,r,&ctx);
434. jacfun_fd(snes,x,&J,&J,NULL,&ctx);
435. // MatView(J,PETSC_VIEWER_STDOUT_SELF); CHKERRQ(ierr);
436.
437. const int *colsp;
438. const double *valsp;
439. int ncols;
440. for (int j=0; j<ctx.neq; j++) {
441. MatGetRow(J,j,&ncols,&colsp,&valsp);
442. for (int l=0; l<ncols; l++)
443. printf("%d %d %f\n",j,colsp[l],valsp[l]);
444. }
445.
446. exit(0);
447. #endif
448.
449. for (int j=0; j<ctx.nstep; j++) {
450. ctx.setup_step(j);
```

```

451. ierr = SNESSolve(snes,NULL,x);CHKERRQ(ierr);
452.
453. #if 1
454. // Use HDF5
455. PetscObjectSetName((PetscObject)x,"u");
456. PetscViewer viewer;
457. char filename[1000];
458. sprintf(filename,"./states/u %d.h5",j);
459. ierr = PetscViewerHDF5Open(PETSC_COMM_WORLD,filename,
460. FILE_MODE_WRITE,&viewer); CHKERRQ(ierr);
461. ierr = VecView(x,viewer);CHKERRQ(ierr);
462. ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);
463. #else
464. // Write to stdout
465. ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
466. #endif
467.
468. Vec f;
469. ierr = SNESGetFunction(snes,&f,0,0);CHKERRQ(ierr);
470. double rnorm;
471. ierr = VecNorm(r,NORM_2,&rnorm);
472.
473. SNESGetLinearSolveIterations(snes,&its);
474. ierr = PetscPrintf(PETSC_COMM_SELF,
475. "number of Newton iterations = "
476. "%d, norm res %g\n",
477. its,rnorm);CHKERRQ(ierr);
478. if (rnorm>1e-7) {
479. printf("aborting due to large error rnorm%g\n",rnorm);

```

```
480. exit(0);
481. }
482. }
483.
484. ierr = VecDestroy(&x);CHKERRQ(ierr);
485. ierr = VecDestroy(&r);CHKERRQ(ierr);
486. ierr = SNESDestroy(&snes);CHKERRQ(ierr);
487.
488. ierr = PetscFinalize();CHKERRQ(ierr);
489. return 0;
490. }
491.
492. //:-<*>-:<*>-:<*>-:<*>-:<*>
493. #undef __FUNCT__
494. #define __FUNCT__ "resfun"
495. int resfun(SNES snes,Vec x,Vec f,void *data) {
496. snes_ctx_t &ctx = *(snes_ctx_t *)data;
497. return ctx.resfun(x,f);
498. }
499.
500. //:-<*>-:<*>-:<*>-:<*>-:<*>
501. int snes_ctx_t::resfun(Vec x,Vec f) {
502. int ierr;
503. VecSet(f,0.0);
504. double *xx,*ff;
505. ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
506. ierr = VecGetArray(f,&ff);CHKERRQ(ierr);
507.
508. // node0,node1 are the nodes at the extreme of the spring
```

```

509. // xref{0,1} are the ref positions of the nodes
510. // x{0,1} are the current positions of the nodes
511. VectorXd xref0(ndim),xref1(ndim),force(ndim);
512. VectorXd x0(ndim),x1(ndim),dx;
513. double *xp;
514. for (int i0=0; i0<N1; i0++) {
515. for (int j0=0; j0<N1; j0++) {
516. // Node at one extreme of the spring
517. int node0=i0*N1+j0;
518. // Reference position of node0
519. xp = &xref[node0*ndim];
520. xref0 << xp[0],xp[1],xp[2];
521. // Current position of node0
522. xp = &xx[node0*ndim];
523. x0 << xp[0],xp[1],xp[2];
524. // Force field at node 0
525. forceep->force(x0,force);
526. for (int l=0; l<ndim; l++)
527. ff[node0*ndim+l] += force(l);
528. for (int l=0; l<4; l++) {
529. int i1=i0,j1=j0;
530. // node1 = EAST node
531. if (l==0) i1++;
532. // node1 = NORTH node
533. else if (l==1) j1++;
534. // node1 = NORTH-EAST node
535. else if (l==2) { i1++; j1++; }
536. // node1 = SOUTH-EAST node
537. else if (l==3) { i1++; j1--; }

```

```

538.
539. if (i1<0 || i1>=N1 || j1<0 || j1>=N1) continue;
540.
541. int node1=i1*N1+j1;
542. // if (!(node0==6 && node1==7)) continue;
543. // Reference position of node1
544. xp = &xref[node1*ndim];
545. xref1 << xp[0],xp[1],xp[2];
546. // Current position of node1
547. xp = &xx[node1*ndim];
548. x1 << xp[0],xp[1],xp[2];
549.
550. dx = x1-x0;
551. // Force of this element
552. double len = dx.norm();
553. double ffj = springp->force(len)/len;
554. // printf("node%d-%d len%f len0%f ffj%f\n",node0,node1,len,len0,ffj);
555. // Unit vector in the direction of the spring
556. for (int l=0; l<ndim; l++) {
557. double w = ffj*dx(l);
558. ff[node0*ndim+l] -= w;
559. ff[node1*ndim+l] += w;
560. }
561. }
562. }
563. }
564.
565. for (auto &q : bcfix) {
566. int node = q.first.first;

```

```

567. int dof = q.first.second;
568. double val = q.second;
569. int jeq = node*ndim+dof;
570. ff[jeq] = xx[jeq]-val;
571. }
572.
573. ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
574. ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);
575. double fnorm;
576. ierr = VecNorm(f,NORM_2,&fnorm);
577.
578. return 0;
579. }
580.
581. //--:<*>--:<*>--:<*>--:<*>--:<*>
582. // This is a wrapper, just calls the snes_ctx_t method
583. #undef __FUNCT__
584. #define __FUNCT__ "jacfun"
585. int jacfun_fd(SNES snes,Vec x0,Mat *jac,Mat*,
586. MatStructure *,void *data) {
587. int ierr;
588. // x0 is the reference x, x1 is perturbed
589. // f0,f1 are the residuals at those states
590. Vec x1,f0,f1;
591. ierr = VecDuplicate(x0,&x1); CHKERRQ(ierr);
592. ierr = VecDuplicate(x0,&f0); CHKERRQ(ierr);
593. ierr = VecDuplicate(x0,&f1); CHKERRQ(ierr);
594.
595. ierr = MatZeroEntries(*jac); CHKERRQ(ierr);

```

```
596. double *x1p,*f0p,*f1p;
597. double epsil=1e-7;
598. int neq;
599. ierr = VecGetSize(x0,&neq); CHKERRQ(ierr);
600.
601. resfun(snes,x0,f0,data);
602.
603. for (int k=0; k<neq; k++) {
604. ierr = VecCopy(x0,x1); CHKERRQ(ierr);
605. ierr = VecGetArray(x1,&x1p); CHKERRQ(ierr);
606. x1p[k] += epsil;
607. ierr = VecRestoreArray(x1,&x1p); CHKERRQ(ierr);
608. resfun(snes,x1,f1,data);
609.
610. ierr = VecGetArray(f0,&f0p); CHKERRQ(ierr);
611. ierr = VecGetArray(f1,&f1p); CHKERRQ(ierr);
612. double tol=1e-10;
613. vector<int> indx;
614. vector<double> coef;
615. // printf("J(%d,:) ",k);
616. for (int j=0; j<neq; j++) {
617. if (fabs(f1p[j]-f0p[j])>tol) {
618. indx.push_back(j);
619. double c = (f1p[j]-f0p[j])/epsil;
620. coef.push_back(c);
621. // printf("(%d, %f) ",j,c);
622. }
623. }
624. // printf("\n");
```

```

625. ierr = VecRestoreArray(f0,&f0p); CHKERRQ(ierr);
626. ierr = VecRestoreArray(f1,&f1p); CHKERRQ(ierr);
627. MatSetValues(*jac,indx.size(),indx.data(),1,&k,
628. coef.data(),INSERT_VALUES); CHKERRQ(ierr);
629. }
630. // Assembly the matrix
631. ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
632. ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
633.
634. // Destroy auxiliary vectors
635. ierr = VecDestroy(&x1); CHKERRQ(ierr);
636. ierr = VecDestroy(&f0); CHKERRQ(ierr);
637. ierr = VecDestroy(&f1); CHKERRQ(ierr);
638.
639. return 0;
640. }
641.
642. //--<*>--<*>--<*>--<*>--<*>
643. #undef __FUNCT__
644. #define __FUNCT__ "jacfun_anly"
645. int jacfun_anly(SNES snes,Vec x0,Mat* jac,Mat*,
646. MatStructure *,void *data) {
647. snes_ctx_t &ctx = *(snes_ctx_t *)data;
648. return ctx.jacfun(x0,jac);
649. }
650.
651. //--<*>--<*>--<*>--<*>--<*>
652. // Computes the Jacobian of the residual vector
653. // with respect to the state vector.

```

```

654. int snes_ctx_t::jacfun(Vec x,Mat* jac) {
655.
656. int ierr;
657. // ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
658.
659. double *xx;
660. ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
661.
662. // node0,node1 are the nodes at the extreme of the spring
663. // xref{0,1} are the ref positions of the nodes
664. // x{0,1} are the current positions of the nodes
665. VectorXd xref0(ndim),xref1(ndim);
666. VectorXd x0(ndim),x1(ndim),dx,tan;
667. MatrixXd jac1(2*ndim,2*ndim),F11(ndim,ndim);
668. MatrixXd Id = MatrixXd::Identity(ndim,ndim);
669. double *xp;
670. vector<int> indx(6);
671. VectorXd f0(ndim),fp(ndim),fm(ndim);
672. MatrixXd fjac(ndim,ndim);
673.
674. ierr = MatZeroEntries(*jac); CHKERRQ(ierr);
675. for (int i0=0; i0<N1; i0++) {
676. for (int j0=0; j0<N1; j0++) {
677. // Node at one extreme of the spring
678. int node0=i0*N1+j0;
679. // Reference position of node0
680. xp = &xref[node0*ndim];
681. xref0 << xp[0],xp[1],xp[2];
682. // Current position of node0

```

```
683. xp = &xx[node0*ndim];
684. x0 << xp[0],xp[1],xp[2];
685.
686. fjac.fill(0.0);
687. // Jacobian of the force field at node 0
688. double epsln=1e-5;
689. for (int l=0; l<ndim; l++) {
690. indx[l] = node0*ndim+l;
691. x1 = x0;
692. x1(l) += epsln;
693. forcep->force(x1,fp);
694. x1(l) -= 2*epsln;
695. forcep->force(x1,fm);
696. fjac.col(l) = (fp-fm)/(2*epsln);
697. }
698. fjac.transposeInPlace();
699. MatSetValues(*jac,3,indx.data(),3,indx.data(),
700. fjac.data(),ADD_VALUES); CHKERRQ(ierr);
701.
702. for (int l=0; l<4; l++) {
703. int i1=i0,j1=j0;
704. // node1 = EAST node
705. if (l==0) i1++;
706. // node1 = NORTH node
707. else if (l==1) j1++;
708. // node1 = NORTH-EAST node
709. else if (l==2) { i1++; j1++; }
710. // node1 = SOUTH-EAST node
711. else if (l==3) { i1++; j1--; }
```

```
712.
713. if (i1<0 || i1>=N1 || j1<0 || j1>=N1) continue;
714.
715. int node1=i1*N1+j1;
716. // Reference position of node1
717. xp = &xref[node1*ndim];
718. xref1 << xp[0],xp[1],xp[2];
719. // Current position of node1
720. xp = &xx[node1*ndim];
721. // printf("xx[node1,:]"g "g %g\n",
722. // xx[node1*ndim],xx[node1*ndim+1],xx[node1*ndim+2]);
723. x1 << xp[0],xp[1],xp[2];
724. // cout << "xref0 " << endl << xref0 << endl;
725. // cout << "xref1 " << endl << xref1 << endl;
726. // cout << "x0 " << endl << x0 << endl;
727. // cout << "x1 " << endl << x1 << endl;
728.
729. dx = x1-x0;
730. double len = dx.norm();
731. // printf("node %d-%d len%f len0%f\n",
732. // node0,node1,len,len0);
733. // Unit vector in the direction of the spring
734. tan = dx/len;
735. double Force = springp->force(len);
736. double epsln = 1e-5;
737. double Fp,Fm;
738. Fp = springp->force(len+epsln);
739. Fm = springp->force(len-epsln);
740. double Fdot = (Fp-Fm)/(2*epsln);
```

```
741. double fdot = Fdot/len-Force/(len*len);
742. F11 = fdot*len*tan*tan.transpose() + Force/len*Id;
743. // cout << "F11: " << endl << F11 << endl;
744. jac1.block(0,0,ndim,ndim) = F11;
745. jac1.block(0,ndim,ndim,ndim) = -F11;
746. jac1.block(ndim,0,ndim,ndim) = -F11;
747. jac1.block(ndim,ndim,ndim,ndim) = F11;
748. for (int j=0; j<ndim; j++) {
749. indx[j] = node0*ndim+j;
750. indx[ndim+j] = node1*ndim+j;
751. }
752. MatSetValues(*jac,6,indx.data(),6,indx.data(),
753. jac1.data(),ADD_VALUES); CHKERRQ(ierr);
754. }
755. }
756. }
757. // Assembly the matrix
758. ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
759. ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
760.
761. vector<int> bcdofs;
762. for (auto &q : bcfix) {
763. int node = q.first.first;
764. int dof = q.first.second;
765. bcdofs.push_back(node*ndim+dof);
766. }
767. MatZeroRows(*jac,bcdofs.size(),bcdofs.data(),1.0,NULL,NULL);
768.
```

```
769. // MatView(*jac,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
770.
771. ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
772. return 0;
773. }
```

# Usando MPI y PETSc

## Usando MPI y PETSc en los clusters del CIMEC

- Para entrar se debe usar un cliente de **ssh** desde Linux o la utilidad **Putty** desde Windows.
- En Linux basta con instalar el cliente de SSH, o sea normalmente el paquete **openssh-clients**, por ejemplo en Fedora:  
`# yum install openssh-clients`.
- En Windows instalar la utilidad Putty desde aquí  
<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>.
- Actualmente el cluster que esta accesible para uso para los cursos es **coyote**. El acceso es por **SSH**, sin embargo para llegar a **coyote** hay que primero entrar al servidor **aquiles** (IP público 200.9.237.240) y luego de ahí a **coyote**.
  1. [bobesp]\$ ssh -C guest04@200.9.237.240
  2. guest04@200.9.237.240's password: \*\*\*\*\*
  3. Last login: Wed Aug 12 16:57:13 2015 from 190.183.116.0
  4. \* Bienvenido al servidor del ex-cluster AQUILES \*
  5. ...
  6. [guest04@aquiles ~]\$ ssh coyote
  7. guest04@coyote's password: \*\*\*\*
  8. \* Welcome to the COYOTE 64-bit cluster server (32 active nodes) \*
  9. ....

10. [guest04@coyote ~]\$

- De manera que normalmente hay que hacer `ssh` (o Putty) a aquiles y después a Coyote. Sin embargo existe un tunel directo desde aquiles a Coyote. Para eso basta con especificar el port 9010.

1. [mstorti@galileo ~]\$ `ssh -p 9010 -C guest04@200.9.237.240`
2. `guest04@200.9.237.240's password:`
3. `Last login: Wed May 18 16:19:00 2016 from 172.16.255.146`
4. `* Welcome to the COYOTE 64-bit cluster server (32 active nodes) *`
5. . . .

La bandera `-C` indica compresión y permite acceder en forma más rápida.

- El usuario puede ser `guest0` con `X=1-5`. El password es el mismo para todos y se dará en clase o por mail.
- Para acceder desde *dentro* de la red del Predio CONICET debe usar el IP interno `172.16.254.78`, o puede ser que funcione el DNS y se puede utilizar directamente el nombre *aquiles*.
- Conviene elegir una de las 5 cuentas en forma random de manera de evitar colisiones con los archivos y corridas entre los diferentes usuarios. De todas formas conviene crearse un directorio para cada uno.

1. [guest03]\$ `mkdir bobesponja`
2. [guest03]\$ `cd bobesponja`
3. [bobesponja]\$

- *Integrantes del CIMEC* pueden usar su cuenta si la tienen o solicitar una.

## En el server URUBU

- **Acceso al server urubu:** A partir de 2020 usaremos el server **urubu** que es un AMD Ryzen Threadripper 2950X, con 16 cores. Para eso las instrucciones son equivalentes a las previas pero con puerto 9019 (en vez de 9010 para **coyote**) y IP interno **172.16.255.87**.
- El acceso a **urubu** es sólo por SSH con clave privada/pública. Para eso tienen que generar un par de claves privada/pública con la utilidad **ssh-keygen** y mandarme la pública. Deben tener instalado el paquete **ssh**, correr la utilidad **ssh-keygen** y darle enter (sobre todo NO poner **passphrase**). Eso genera dos archivos **id\_rsa** y **id\_rsa.pub** en el directorio **~/.ssh**. Ustedes deben enviarme la clave pública que es el **id\_rsa.pub** (es un archivo con una sola línea muy larga). Por ejemplo mi clave pública es,

1. [mstorti@galileo curso-mpi-petsc]\$ cat ~/.ssh/id\_rsa.pub
2. ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAQIEAociEHlc1YHm+SIgI6RIqB6kABVgJ\
3. M8K6yA4HmTUH2MtZoG2lZ1G/qLfwInYIDqAkeDAEItEYcIrjnayQWo/LagExdCj00\
4. ysBJckbbqwaRFJfIfn6w0/Z2jRRpU6f2DmU0ES1jT5udBCE7s4ogyIx4YcGqLU2c7H\
5. aaAvvbSH8= mstorti@aquiles

Pueden ver más sobre estos enlaces: [do.co/3ecexU0](https://do.co/3ecexU0), [bit.ly/3c8va1J](https://bit.ly/3c8va1J)

- Al generar la clave es acoseñable NO poner **passphrase**.

- Una vez que hayan generado la clave pública y yo la cargue en el *urubu* pueden entrar y usar la cuenta *guest* es decir
1. [user]\$ `ssh -p 9019 guest@200.9.237.240`
    - Una vez adentro de la cuenta *guest* en *urubu*, crear su directorio propio (e.g. `mkdir bobesponja`) y continuar como en los clusters.
    - Para acceder desde Windows deben utilizar Putty (explicado más arriba) y crearse la clave como se explica aquí [bit.ly/3c5JkjY](http://bit.ly/3c5JkjY).
    - En Windows la clave la deben generar con *Puttygen*. Asegúrense que sea una clave RSA-2.

## Instrucciones generales

- Para arrancar pueden copiar el `hello.cpp` que está en  
`/u/guest01/MSTORTI/hpcmc-example`. (en *urubu*:  
`/home/guest/mstorti/mpi-hello`).
- Para compilar un programa simple (e.g. que no usa PETSc u otras librerías)
  1. [bobesponja]\$ `mpicxx -o hello.bin hello.cpp`
- ***Posibles editores de texto:*** Para editar archivos pueden usar `nano` que es un editor muy simple. Otro posible es `mc` (Midnight Commander) que es un clon del Norton Commander. Permite navegar directorios, y además editar archivos con un editor simple. Hay muchos tutorials online, algunos posibles son:
  - ▷ [Midnight Commander, by William E. Shotts, J](#)
  - ▷ [Midnight Commander Guide, by Mueen Nawaz](#)
  - ▷ [Use Midnight Commander like a pro, by Igor Klimer](#)
- Otros dos editores muy comunes en el mundo Linux son Emacs y Vi.
- ***Para copiar archivos desde/hacia el server:*** desde Linux pueden copiar desde la línea de comando con la utilidad `scp` (secure copy) parte del

paquete **SSH**,

1. **## Copia local.txt desde la note a urubu**
2. **[user]\$ scp -P 9019 local.txt guest@200.9.237.240:**
3. **## Copia remote.txt desde urubu a la notebook**
4. **[user]\$ scp -P 9019 guest@200.9.237.240:remote.txt .**

En el caso de copiar al server el archivo queda en el home del usuario remoteo (o sea `/home/guest`), despues deben encargarse de moverlo a su directorio de usuario. Para copiarlo directamente a su directorio de usuario pueden hacer

1. **## Copia local.txt desde la note a urubu a la cuenta de usuario**
2. **[user]\$ scp -P 9019 local.txt guest@200.9.237.240:bobesponja/**

También pueden usar navegadores de archivos como Dolphin (KDE) o Nautilus (Gnome), en la barra de direcciones deben poner  
`fish://guest@200.9.237.240:9019` (para urubu). De esa forma pueden llevar y traer archivos de un lado hacia el otro fácilmente.

Para los usuarios de Windows pueden usar el utilitario WinSCP que se baja del mismo sitio que Putty. Para cargar en WinSCP la clave privada deben seguir estas instrucciones [bit.ly/2XvP4j4](http://bit.ly/2XvP4j4)

- **Seguridad en el cluster:** Debido a que aquiles (el nodo de acceso a coyote) tiene un IP público es muy accesible a ataques informáticos de todo tipo todo el tiempo. De manera que recomendamos extremar las

**medidas de seguridad.** En particular hay un **robot** (dispositivo informático automatizado) que monitorea los **logs** y si detecta que han querido entrar infructuosamente en forma repetida desde un IP, lo **bannea** es decir impide que se pueda conectar. Por eso si en algún momento reciben un mensaje al querer entrar de **connection refused** o similar, comuníquenlo ya que probablemente han sido **baneados**. Deben reportar su IP entrando a la siguiente página <http://www.ip-adress.com/>.

## Instrucciones específicas para PETSc

Conviene escribir un **Makefile** siguiendo estas instrucciones

- Copiar todo el directorio `/u/guest01/MSTORTI/petsc-example` en vuestro directorio (`/home/guest/mstorti/petscex` en urubu).

1. `$ cd /u/guest03/SpongeBob`
2. `$ cp -r /u/guest01/MSTORTI/petsc-example .`

- Probar a que funcione

1. `$ make ex1.bin`
2. `$ make snes3.bin`

Debería crear los ejecutables. Estos son ejemplos que dimos en la clase.

- El Makefile contiene lo siguiente:

1. `PETSC_DIR := /usr/local/petsc/3.2`
2. `PETSC_ARCH := arch-linux2-c-opt`
3. `include ${PETSC_DIR}/conf/variables`
4. `include ${PETSC_DIR}/conf/rules`
- 5.
6. `ex23.bin: ex23.cpp`
7. `mpicxx $(COPTFLAGS) $(PETSC_CC_INCLUDES) -o $@ $^ $(PETSC_LIB)`
- 8.
9. `poisson.bin: poisson.cpp`

10. `mpicxx $(COPTFLAGS) $(PETSC_CC_INCLUDES) -o $@ $^ $(PETSC_LIB)`

- Si es necesario se pueden cambiar las variables del Makefile.
- `PETSC_DIR` el directorio donde está PETSc.
- `PETSC_ARCH` la configuración de PETSc a usar. Ante la duda buscar los directorios en `PETSC_DIR`, por ejemplo en nuestro caso veremos que existe el directorio `/usr/local/petsc/3.2/arch-linux2-c-opt` por lo tanto la arquitectura a utilizar es `arch-linux2-c-opt`.
- En caso de querer compilar otro programa, digamos `myprog.cpp` copiar las dos líneas correspondientes a `ex23.cpp` o a `snes3.cpp` y cambiar `ex1` por `myprog`, es decir

1. `myprog.bin: myprog.cpp`
2. `mpicxx $(COPTFLAGS) $(PETSC_CC_INCLUDES) -o $@ $^ $(PETSC_LIB)`

- Para compilar/linkeditar el programa `$ make myprog.bin`

## Uso de SLURM

- Para correr el programa en el cluster **coyote** deben submitirlo a la cola con SLURM. Para eso deben crear un script, por ejemplo **runjob** con el siguiente contenido

```
1. #!/usr/bin/bash
2. #SBATCH -ntasks=18 -ntasks-per-node=6 -t 1
3.
4. DIRE=/u/guest04/mstorti
5. cd $DIRE
6. /usr/lib64/mpich2/bin/mpexec $DIRE/ex23.bin -n 100
```

- En este script deben configurar
  - ▷ **ntasks** es el número total de procesos a lanzar.
  - ▷ **ntasks-per-node** la cantidad de tareas por nodos a lanzar.
  - ▷ Por ejemplo, si van a utilizar nodos de tipo XEON W3690 (son los de **coyote**) entonces tienen 6 cores por nodo y por lo tanto **ntasks-per-node=6**. Si quieren usar 3 nodos, entonces deben poner además **ntasks=18**.
  - ▷ **DIRE** el directorio de trabajo, en mi caso **/u/guest04/mstorti**
  - ▷ El nombre del programa, en nuestro caso **ex23.bin**.
  - ▷ Argumentos para el programa se pueden pasar después del binario,

por ejemplo en este caso **-n 100** quiere decir que corra el ejemplo de Poisson 1D con 100 nodos.

- Para submitir el job

1. [guest04@coyote mstorti]\$ sbatch runjob
2. Submitted batch job 3159

quiere decir que el job fue submitido correctamente y fue asignado el JOBID 3159.

- Para ver el estado

1. guest04@coyote mstorti]\$ squeue -l
2. Sun May 22 11:37:32 2016
3. JOBID PART NAME USER STATE TIME T\_LIM NDS NDLIST
4. 2314 work job.sh cvenier RUNNING 19:27:29 UNLIM 1 node31
5. 3159 work runjob guest04 RUNNING 0:11 1:00 1 node32
6. 3092 work molec jnavarro RUNNING 23:27:44 UNLIM 3 node[22-24]
7. . . .

Si el job no aparece en la cola puede ser que ya haya terminado.

- Para cancelar un job \$ scancel 3159
- Límite de tiempo: si el job debe terminar rápido es conveniente ponerle un límite agregando a la línea SBATCH del script la opción **-t**, por ejemplo **-t 1** (como está arriba) indica que el tiempo máximo es 1 minuto. Si el job se pasa de ese tiempo será matado por SLURM. Si no se utiliza este opción existe el riesgo de que el job (ya sea por un deadlock o porque

**tarda demasiado por otra razón) bloquee la cuenta correspondiente.**

## Uso de MPI en computadora personal con Linux

- En la mayoría de las distros actuales hay paquetes (RPM y DEB) para MPI y PETSc. Por ejemplo en Fedora 31 tenemos

1 `mpich-3.3.2-1.fc31.x86_64`  
2 `mpich-devel-3.3.2-1.fc31.x86_64`

- Para OpenMPI los paquetes son

1 `openmpi.x86_64` : Open Message Passing Interface  
2 `openmpi-devel.x86_64` : Development files for openmpi

Notar que en ambos casos hay que instalar dos paquetes, uno con la librería (`mpich` o `openmpi`) y otro con los headers que usualmente terminan en `devel` (e.g. `mpich-devel` o `openmpi-devel`).

- En Ubuntu los paquetes como OpenMPI y MPICH (que son librerías) empiezan con `lib`, por lo tanto los paquetes son `libopenmpi`, `libopenmpi-dev` para OpenMPI y `libmpich`, `libmpich-dev` para MPICH.
- Una vez instalados los paquetes hay que tener acceso a las utilidades, fundamentalmente `mpicxx` para compilar C++, y también `mpirun` y `mpiexec` para lanzar los programas. Para esto hay que agregar el directorio donde están los binarios al `PATH`, e.g. agregar al `~/.bashrc`

```
1 PATH=$PATH:/usr/lib64/mpich/bin
```