Introducción a la Programación Paralela (primera parte)

José Luis Gordillo
Leobardo Itehua
Julio César Clemente
Coordinación de Supercómputo
DGTIC - UNAM

Introducción

Introducción

- Uno de los propósitos principales de escribir un programa paralelo suele ser conseguir un modelo computacional que permita estudiar algún fenómeno de forma más rápida

- Se necesitan programas paralelos:
  - Eficientes
  - Escalables
  - Portables
  - Flexibles

Introducción

- Seminarios/talleres típicos de IPP cubren conceptos necesarios para hacer programas eficientes
- Si el problema es trivialmente paralelizable, los conceptos se aplican de forma trivial
  - Pero la mayoría de las aplicaciones interesantes no son triviales
  - Seminarios/talleres típicos de IPP cubren aplicaciones triviales y no triviales de estos conceptos

Introducción

- Modelos de programación/arquitectura de computadoras paralelas
  - ¿Cómo son y cómo se programan las computadoras paralelas? (1)
- Reparto de carga
  - ¿Cómo se utilizan simultáneamente los procesadores de una computadora paralela?(2)
- Comunicaciones
  - ¿Cómo se definen, estructuran y miden las comunicaciones entre procesos? (3)

Introducción

- Balance de carga
  - ¿Qué dificultades se pueden encontrar al repartir la carga entre procesadores y qué efectos tiene una carga mal repartida? (4)
- Validación del programa
  - ¿Cuál es la fuente de las diferencias de resultados entre la versión secuencial y la paralela? (5)
- Evaluación de rendimiento/eficiencia
  - Speedup, eficiencia, speedup superlineal, speedup escalado, granularidad, paralelismo, concurrencia (6)

Introducción

- Sin embargo, los participantes sin experiencia en programación paralela se quedan en (1) o en (2)
  - Los casos complejos son difíciles de asimilar, ya que no se ha *experimentado* con casos triviales
- Nuestra estrategia es dividir el trabajo en dos
  - Que el estudiante aprenda a hacer programas paralelos (versión completa de este seminario)
  - Que el estudiante aprenda conceptos y herramientas para hacer buenos programas paralelos (versión completa de IPP segunda parte)

Conceptos básicos

¿Por qué hacer programas paralelos?

- A mediados de 1990's, los fabricantes de supercomputadoras determinaron que era más fácil y barato construir máquinas con muchos procesadores en vez de construir procesadores más potentes

- A mediados de 2000's, los fabricantes de procesadores determinaron que es más eficiente usar procesadores con muchos "cores" en vez de construir procesadores más rápidos
  - Prácticamente ya no se fabrican procesadores de un solo "core"

¿Por qué hacer programas paralelos?

- Las supercomputadoras han pasado en estos 15 años de unos cuantos miles de procesadores a más de 100,000 “cores”
- En servidores existen procesadores con 12 “cores” y se espera que este número se siga incrementando año con año
- ¿Qué se puede hacer con tantos “cores” ?
  - Correr muchos programas distintos simultáneamente
    - La cantidad de trabajo por unidad de tiempo aumenta
      - High Throughput Computing
    - Correr un solo programa usando todos los cores disponibles
      - High Performance Computing

¿Por qué hacer programas paralelos?

- Nota: ¿es lo mismo “core” que “procesador”?
  - En procesadores que se usan en equipos de escritorio y en servidores (Opteron, Athlon, Xeon, PowerPC), son equivalentes desde el punto de vista de programación
    - Los cores son procesadores completos incrustados en un mismo “chip”
    - Las diferencias tienen que ver con la forma en que se comunican cores de un mismo chip vs cores en distintos chips (Segundo seminario)
  - En el caso de otro tipo de dispositivos, como GPUs o procesadores Cell, son distintos
    - Se programan de forma distinta

¿Qué son los programas paralelos?

- Crear un programa paralelo significa hacer un programa que utiliza de forma coordinada más de un elemento de procesamiento
  - Procesadores, cores, etc.
- Todo programa realiza una cierta cantidad de “trabajo”
  - Una secuencia de instrucciones/operaciones en uno o más conjuntos de datos
- Un programa paralelo reparte ese trabajo entre los varios elementos de procesamiento que se pretende utilizar
  - El hecho de repartir el trabajo se conoce como “partición” o “descomposición”

Reparto de carga

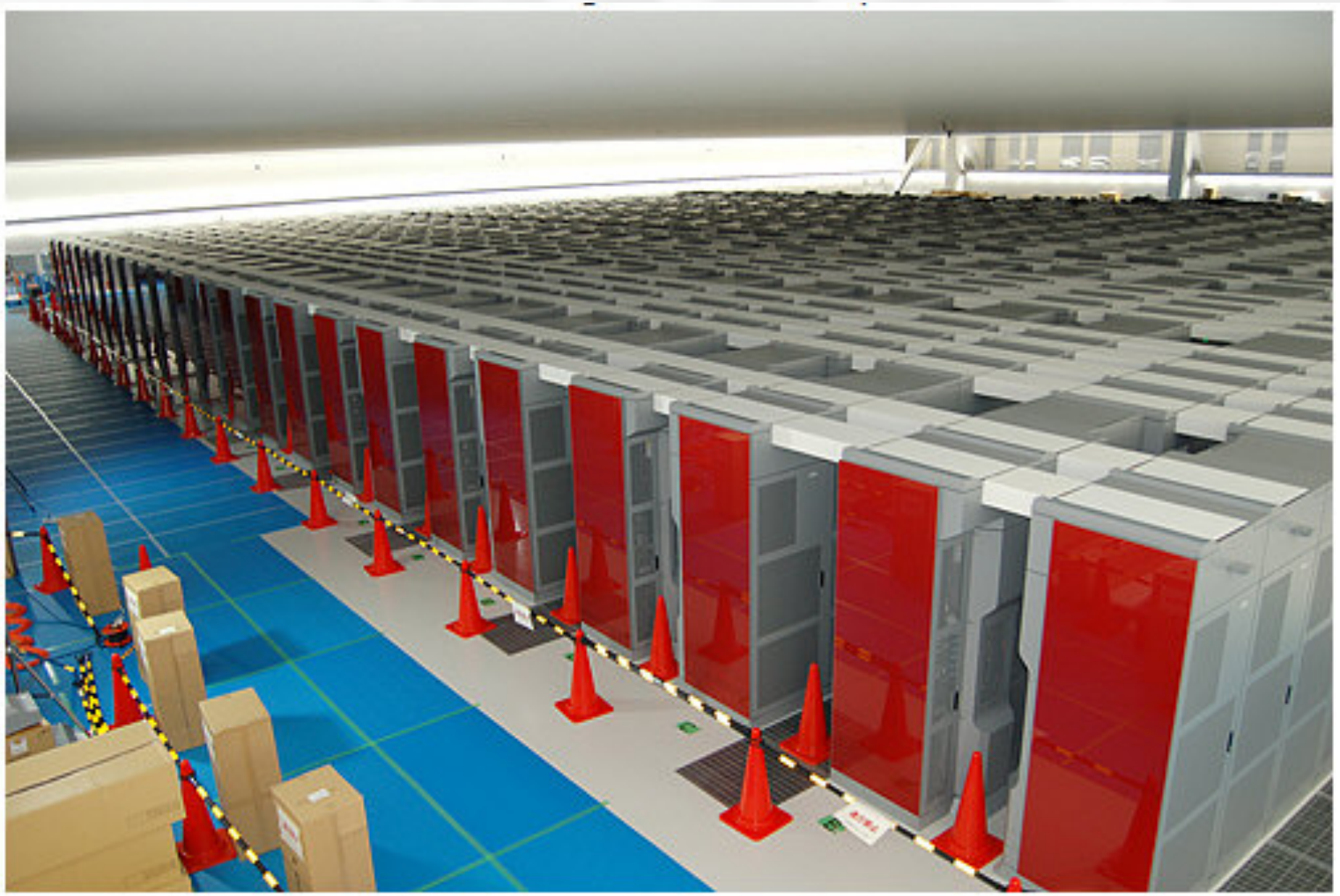
- Si se reparten las intrucciones/operaciones, se conoce como “partición funcional”
  - Poco usada, ya que en la mayoría de los casos las funciones que pueden realizarse simultáneamente son pocas
- Si se reparten los datos, se conoce como “partición de datos”
  - Es la más utilizada, ya que los conjuntos de datos suelen ser bastante grandes, y las operaciones que se realizan sobre ellos se pueden hacer de forma simultánea
    - Átomos en simulaciones de dinámica molecular
- Los mecanismos para repartir datos dependen de la *arquitectura* y del *modelo de programación paralela*

Arquitecturas de computadoras paralelas

Computadoras paralelas

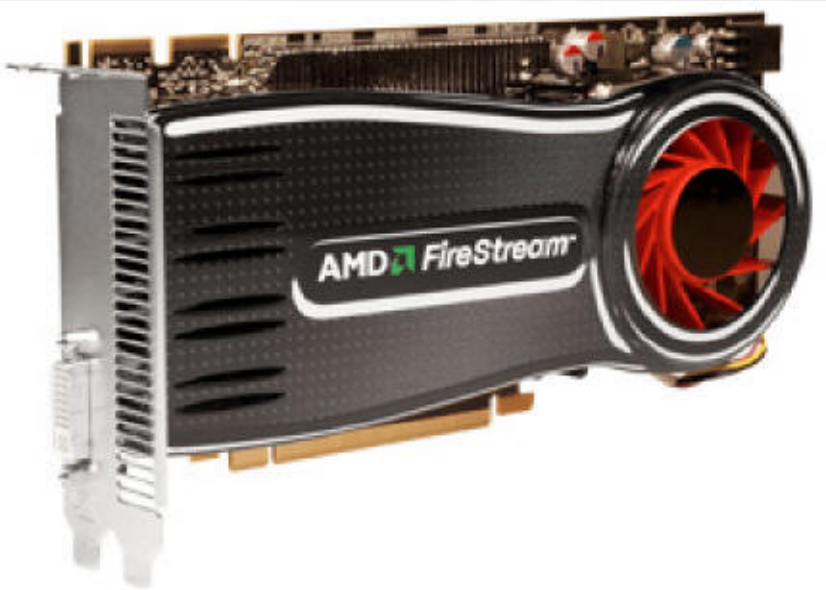
Una computadora paralela está compuesta de procesadores, memoria, sistemas de entrada/salida y sistemas de interconexión.

Fujitsu K



GPGPU

FireStream 9370



Tesla M2090

Dispositivos con varios núcleo

Computadoras paralelas

El procesador carga y ejecuta instrucciones de un programa, esta ejecución implica cálculos lógicos y aritméticos, accesos a memoria y mantener el orden de ejecución del programa.

El sistema de memoria almacena el estado actual de los cálculos y mantiene la consistencia de los datos cuando varios procesadores hacen uso de la memoria concurrentemente.

Computadoras paralelas

Estos nodos requieren de redes de interconexión que les permitan colaborar en la solución de un único problema de gran magnitud.

Las redes de interconexión también conectan los nodos con dispositivos de entrada/salida.

El sistema de Entrada/Salida da soporte a sistemas periféricos como son unidades de cinta, discos y redes externas.

Computadoras paralelas

El desempeño de todos estos sistemas determinan el rendimiento general de la computadora paralela.

Computadoras paralelas

Es importante el estudio de los elementos que componen una computadora paralela para entender lo que sucede con el flujo de los datos e instrucciones entre la memoria y las unidades de procesamiento.

De otra manera se tendrá dificultad en entender las técnicas de optimización y la información que proporcionan las herramientas para mejorar su desempeño.

Taxonomía de arquitecturas

- Existen diferentes tipos de máquinas paralelas
- Existen varias formas de clasificarlas
  - Taxonomía de Flynn
  - Por mecanismos de comunicación

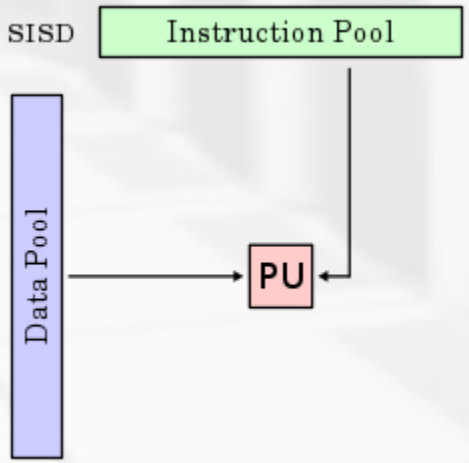
Nota: varios sistemas son una mezcla de estos enfoques y no corresponden exactamente a las clasificaciones.

Taxonomía de Flynn (Instrucción/Datos)

La clasificación de Flynn (1966) está basada en el número de flujos de instrucciones y de datos que pueden ser procesados simultáneamente.

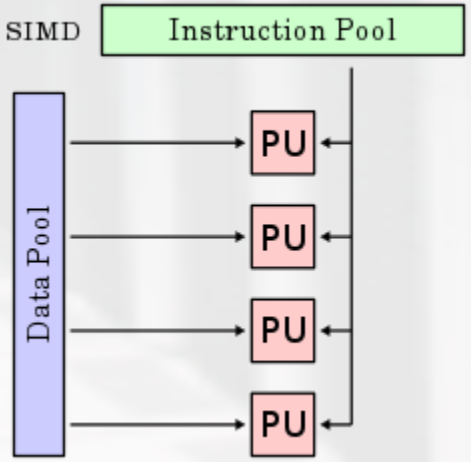
SISD
(una instrucción, un dato)

- La definición clásica de un solo procesador.
- Ejemplos de arquitecturas SISD son las máquinas con uni–procesador o monoprocesador tradicionales como algunos PC o los antiguos mainframe.



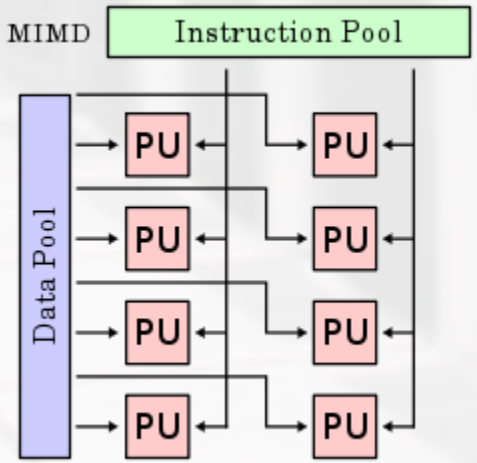
SIMD
(una instrucción, múltiples datos)

- Como ejemplo de este caso podemos citar al procesador vectorial.



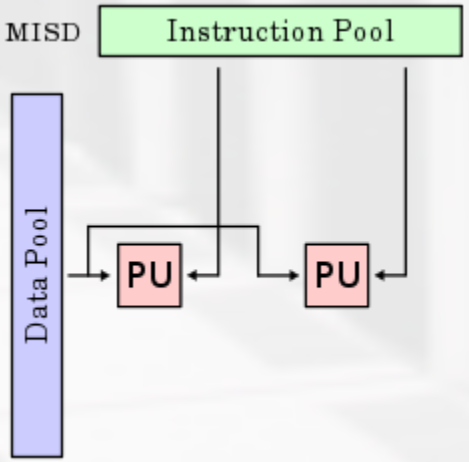
MIMD
(múltiples instrucciones, múltiples datos)

- Cubre el rango de sistemas multiprocesadores.
- Los sistemas distribuidos suelen clasificarse como arquitecturas MIMD; bien sea explotando un único espacio compartido de memoria o uno distribuido.



MISD
(múltiples instrucciones, un dato)

- Poco común debido al hecho de que la efectividad de los múltiples flujos de instrucciones suele precisar de múltiples flujos de datos.



Taxonomía de Flynn

Esta clasificación es útil por su simplicidad y amplio uso, sin embargo, existen severas críticas a la clasificación, particularmente porque contiene modelos inexistentes (MISD) y fue bastante amplio para la clasificación de sistemas multiprocesadores.

Mecanismos de comunicación

La categoría de multiprocesadores MIMD está dividida en dos subtipos:

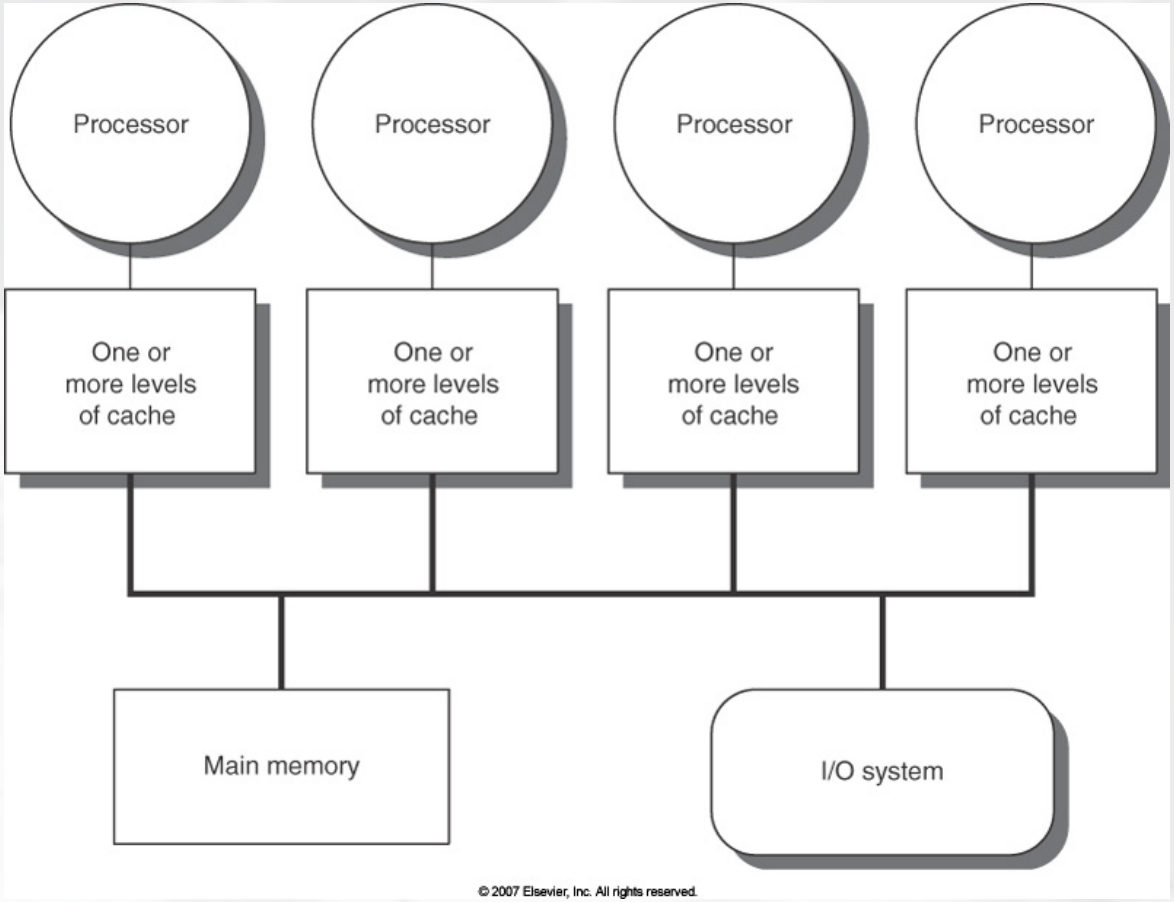
- Memoria compartida
- Memoria distribuida

Memoria compartida

Un multiprocesador MIMD de memoria compartida tiene *sólo una memoria que puede ser accedida por todos los procesadores*.

Un problema importante en este tipo de máquinas es la escalabilidad, ésta es la facilidad para incrementar el número de procesadores y otros elementos de HW significativamente con un correspondiente incremento en el rendimiento.

Memoria compartida



Memoria compartida

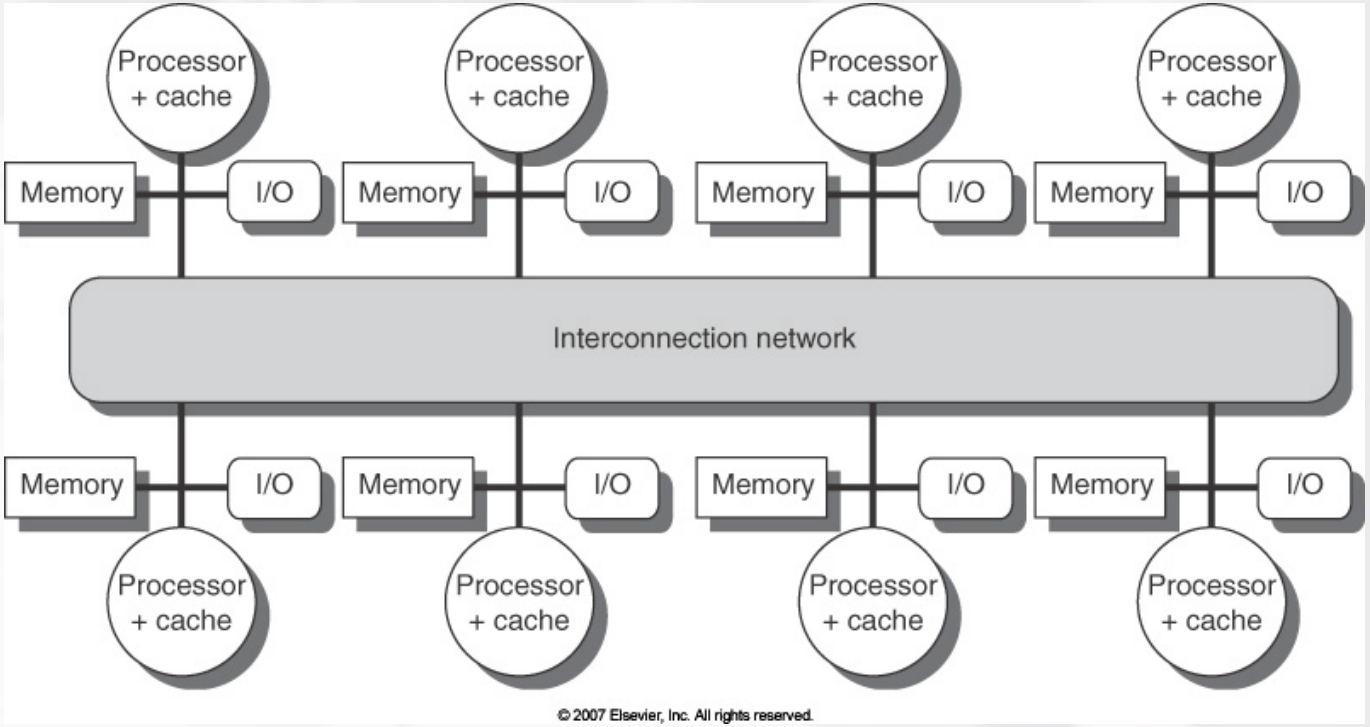
Conforme se incrementa el número de procesadores de igual manera aumenta el tráfico en el bus que conecta a los procesadores con la memoria compartida.

Incrementando el ancho de banda del bus se soluciona parcialmente el problema, de cualquier manera, muchos programas requieren el uso de variables comunes para la mayoría de los procesadores.

Memoria distribuida

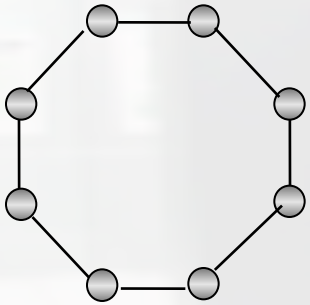
Cada uno de los procesadores individuales de un multiprocesador de memoria distribuida tiene asociado directamente a él una unidad de memoria; esto es, *cada procesador tiene su propia memoria local*. Un procesador junto con su memoria es llamado nodo y está conectado a los otros nodos mediante algún tipo de red.

Memoria distribuida

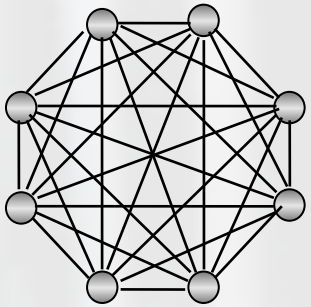


Memoria distribuida

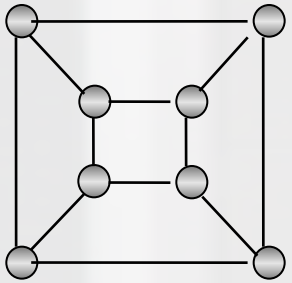
Estas redes pueden ser de algunos de los siguientes tipos:



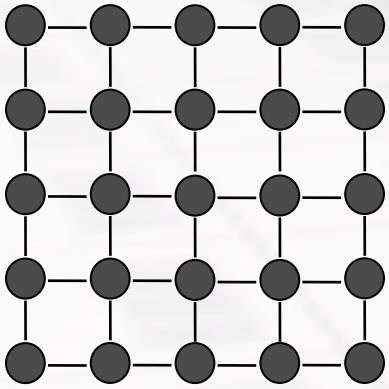
Anillos



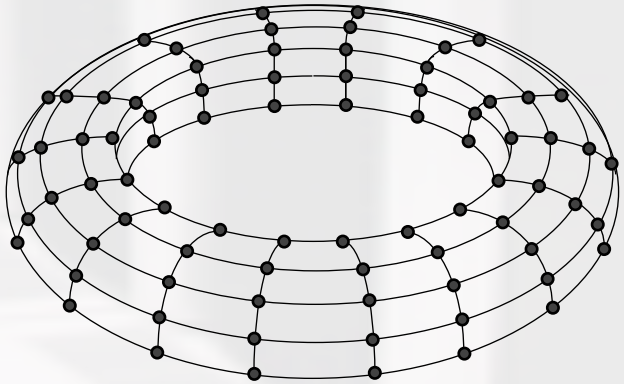
Totalmente conectadas



Hipercubo



Mallas



Toros

Memoria distribuida

Cada uno de los nodos de un multiprocesador envía mensajes a los otros nodos y recibe mensajes de ellos para distribuir y obtener los datos. Es por ello que este tipo de máquinas son también conocidas como computadoras de envío de mensajes.

Memoria distribuida

Para la transmisión de los datos entre los diferentes nodos de un multiprocesador, se deben tener en cuenta características como:

- Ancho de banda
- Tamaño de los mensajes

Debido a que son características importantes que afectan el rendimiento de las aplicaciones que se ejecutan en este tipo de máquinas.

El balance entre la capacidad de cómputo del CPU y el tiempo invertido en las comunicaciones, es de vital importancia.

Modelos de programación paralela

¿Qué es un modelo de programación paralela?

- Una vista de datos y ejecuciones.
- Donde la arquitectura y la aplicación convergen.
- Se puede visualizar como un contrato.
  - Todos conocen las reglas.
  - Las consideraciones de rendimiento importan.
- Beneficio:
  - Aplicación: el modelo permite abstraerse del hardware.
- Inconveniente:
  - El rendimiento depende del nivel de acoplamiento entre el hardware y el modelo.

Modelos de programación paralela

- Memoria compartida
- Memoria distribuida
- Híbrido compartida/distribuida
- Partitioned Global Address Space (PGAS)
- Paralelismo de datos (GPUs)

Memoria compartida

- Varios procesadores usan un mismo espacio de memoria
  - Computadoras con múltiples procesadores
  - Computadoras con multicores.
- Existe un direccionamiento de memoria común para todos los procesadores.
- La comunicación entre procesos es muy rápida.
  - Es implícita
- No escala a un número muy grande de procesadores.

Memoria compartida

- Declaraciones simples.
  - Se lee memoria “remota” vía una expresión.
  - Se escribe en memoria “remota” a través de asignación.
- La manipulación de datos podría requerir sincronización.
  - Condiciones de carrera es uno de los errores más comunes y difíciles de detectar
- Inconveniente:
  - No permite la explotación de localidad
- Ejemplos: OpenMP, pThreads

Memoria distribuida

- Los procesadores disponen de sus propios recursos.
  - Ejemplo: cluster de computadoras.
- Un sistema de interconexión permite “acceder a la memoria de otros procesadores”.
- Cada procesador tiene copias locales de los datos.
- Datos comunes se deben replicar para cada procesador.
- Se usa paso de mensajes.

Memoria distribuida

- Los programadores controlan los datos y la distribución de trabajo.
- La comunicación es explícita.
- Inconveniente:
  - Sobrecarga significativa sobre comunicaciones para transacciones pequeñas.
- Ejemplo: Biblioteca de paso de mensajes MPI.

Memoria distribuida

- La comunicación se hace explícitamente mediante mensajes.
  - Permite un mayor control y cuantificación de la comunicación.
- La programación resulta más compleja de realizar.

Híbrido compartida/distribuida

- Hardware donde los procesadores disponen de memoria compartida y distribuida.
  - Ejemplo: cluster de nodos con multicores.
- Para obtener un mejor rendimiento, es necesario utilizar ambos modelos
- Memoria compartida para programar cada nodo.
- Memoria distribuida para comunicar entre los nodos.

PGAS

- Al igual que el modelo anterior, se dispone de hardware híbrido.
- Pero se visualiza como un paradigma de memoria compartida.
- La memoria M_i tiene afinidad al hilo H_i.
- Ayuda a explotar la localidad.
- Permite hacer declaraciones simples.
- Se implementa mediante lenguajes de programación o bibliotecas:
  - Ejemplo: UPC

OpenMP: Introducción

- Se puede usar el modelo de paso de mensajes; pero:
- Existe una solución más conveniente, basada en el paradigma de memoria compartida: OpenMP
- Permite programar sobre computadoras con memoria compartida.
- El compilador es quien paraleliza el código.
- Debe ser asistido por directivas dadas por el programador.

OpenMP

- OpenMP permite escribir aplicaciones multi-hilos (multi-threads).
  - Evita programarlos de manera directa.
- La paralelización se especifica mediante:
  - Un conjunto de directivas.
  - Algunas funciones de biblioteca.
  - Unas cuantas variables de entorno.
- Se programa de manera muy semejante tanto en C/C++ como en Fortran.

OpenMP

- Modelo de programación denominado "Paralelismo Fork-Join”.
- Un hilo maestro genera un conjunto de hilos al entrar en una zona paralela.
- Al salir, los hilos hijos terminan, y continúa la parte secuencial.
- Generalmente el paralelismo se añade incrementalmente a un programa secuencial.

Consideraciones al programar con OpenMP

- Se utiliza generalmente en la paralelización de ciclos.
  - Mediante la localización de los ciclos más lentos.
- Si es posible, la tarea se reparte entre varios hilos.
  - No debe existir dependencia de cálculo entre iteraciones.
- Se deben evitar las condiciones de carrera.
  - Se presenta al compartir datos de forma no intencionada.
- Para evitarlo, se puede usar sincronización de hilos.
  - La sincronización implica serialización, por lo tanto es costosa.

MPI - Introducción

- Fue creado en 1993 como estándar abierto por fabricantes y usuarios de sistemas paralelos.
- Cada fabricante implementa MPI para su sistema.
- También existen implementaciones de código fuente abierta
  - Lo que permitió el desarrollo de sistemas paralelos de bajo costo.

Características de MPI

- Interfaz genérica que permite una implementación optimizada en cualquier sistema paralelo.
- Es una biblioteca que incluye interfaces para Fortran, C y C++
- Define varias formas de comunicación,
  - Lo que permite programar naturalmente cualquier algoritmo paralelo.

MPI

- Se usa MPI cuando se trabaja sobre una computadora de memoria distribuida.
  - Lo que imposibilita usar OpenMP.
- Como MPI utiliza un esquema SPMD, inicialmente todo se ejecuta en paralelo.
  - Para regiones particulares del algoritmo, es necesario que el programador lo especifique.
  - Se especifica mediante variables y funciones.

MPI

- Al igual que con OpenMP, MPI se programa de manera muy similar tanto en C/C++ como en Fortran.
- En un programa MPI, la ejecución es mediante procesos.
  - El número de procesos se especifica al ejecutar el programa.
- La comunicación entre procesos es a través de mensajes.
  - Los mensajes deben ser especificados por el programador.

MPI

- Existen diferentes sentencias:
  - Comunicación, cálculo y sincronización.
- Existen dos tipos de mensajes:
  - Colectivas
  - Punto a punto
- Los mensajes pueden ser bloqueantes o no bloqueantes.

MPI - consideraciones

- Sólo debe existir una única inicialización y una única terminación.
- Tener cuidado en la correspondencia entre mensajes, principalmente cuando son punto a punto.
- Tener cuidado al usar mensajes bloqueantes.
  - Pueden generar deadlocks (bloqueos mutuos).
- El binario debe ser accesible en todos los nodos.
  - En un sistema de archivos distribuido esto se obvia.

PGAS

- Máquinas de memoria distribuida utilizan envío de mensajes (MPI, PVM)
- Máquinas con varios procesadores y memoria compartida utilizan hebras/hilos (OpenMP, pThreads)
- Máquinas con memoria distribuida y nodos con varios procesadores pueden utilizar esquemas híbridos.
  - Usar OpenMP dentro de los nodos y MPI para intercambio de información entre éstos.
- Se propone un nuevo modelo de programación.
  - Partitioned Global Address Space (PGAS)

PGAS

- Hilos concurrentes con espacio compartido particionado.
- Un elemento puede referenciar datos en otras particiones.
- Arreglos globales se pueden fragmentar en múltiples particiones.
- Algunos lenguajes que implementan el modelo:
  - UPC, X10, Chapel, CAF, Titanium

PGAS

- Pros
  - Ayuda a la explotación de cercanía (localidad)
  - Instrucciones simples como en el modelo de memoria compartida
- Contras
  - Compartir toda la memoria puede llevar a errores sutiles y condiciones de carrera.

PGAS con paralelismo dinámico

- Concurrencia explícita.
  - SPMD es un caso particular.
- Actividades asíncronas se pueden comenzar y detener en una partición dada.
- Actividades asíncronas se pueden usar para mensajes activos.

PGAS con paralelismo dinámico

- Se implementa a través de:
  - Biblotecas: Para C, Fortran y Java (que cohabitan con MPI).
    - Referencias remotas
    - Estructuras de datos globales
    - Operadores globales o colectivos
    - Operaciones atómicas
  - Lenguajes:
    - CAF asíncrono, UPC asíncrono, X10, Chapel
  - Usar bibliotecas reduce costo de adopción.
  - Usar lenguajes aumenta productividad.

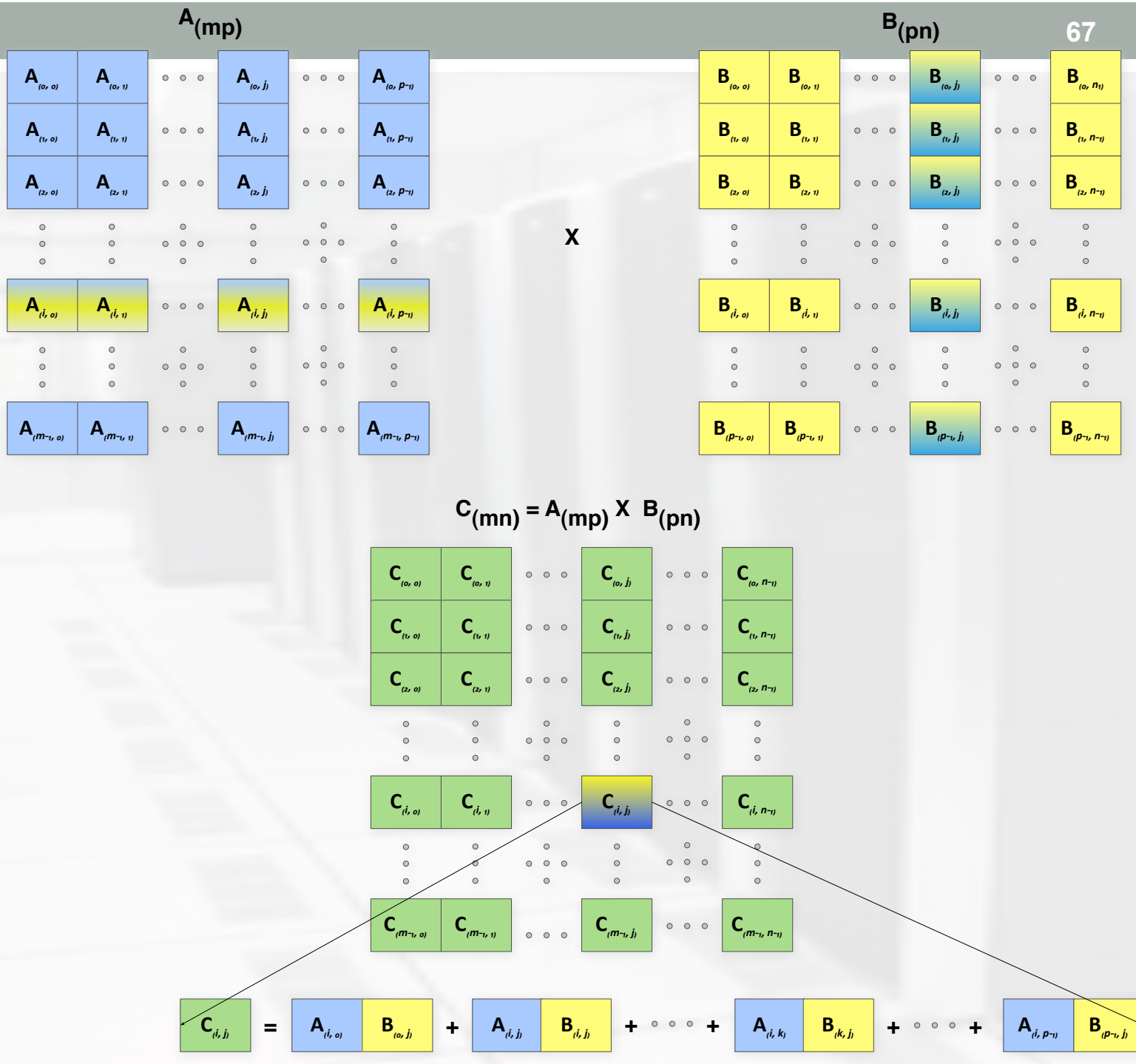
PGAS en resumen

- Modelo de programación paralela que combina:
  - Las características del rendimiento y la localidad de datos (particionado) de memoria distribuida.
  - La fácil programación y la referencia simplificada de un modelo de memoria compartida (global address space).
  - Para ello PGAS proporciona:
    - Un estilo de programación de vista local
    - Un espacio de direcciones global
    - Introducción de comunicación para resolver referencias remotas
    - Comunicación en un sólo sentido para mejorar rendimiento entre procesos
    - Soporte para tipo de datos distribuidos

Ejemplo 1: Multiplicación de matrices

Multiplicación de matrices

- Se desea realizar la multiplicación de dos matrices rectangulares A y B.
- Dadas dos matrices A, de dimensión mxp, y B, de dimensión pxn, la matriz C, de dimensión mxn, se calcula de la siguiente forma:


A(mp)
B(pn)
X
C(mn) = A(mp) X B(pn)
C(i, j) = A(i, 0) B(0, j) + A(i, j) B(i, j) + ... + A(i, k) B(k, j) + ... + A(i, p-1) B(p-1, j)

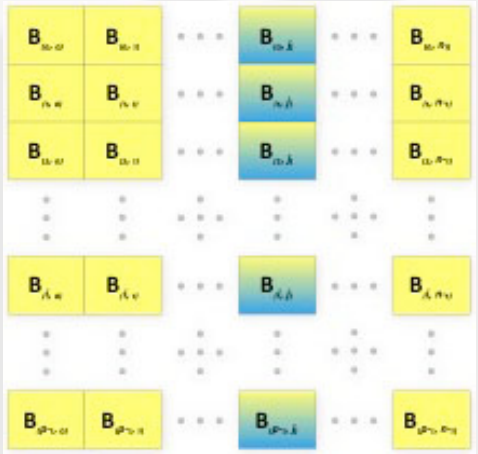
Programa serial

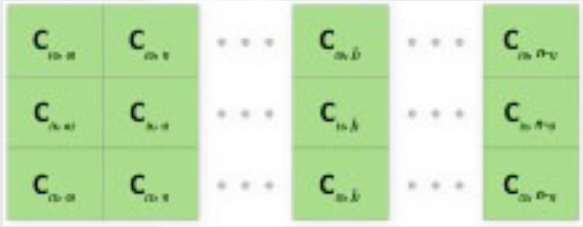
```
37    float Mat_A[num_ren_a][num_col_a];
38    float Mat_B[num_ren_b][num_col_b];
39    float Mat_C[num_ren_a][num_col_b];
40
41    int num_ren_c, num_col_c;
42
43    FILE *ptrCf;
44    int i, j, k;
45    char string[100];
46
47    struct timeval t_ini, t_fin;
48    double secs;
```

Programa serial

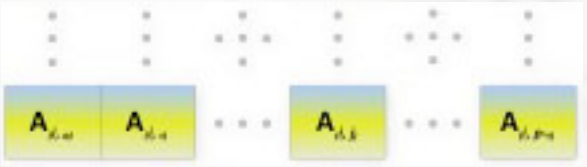
```
84    num_ren_c = num_ren_a;
85    num_col_c = num_col_b;
86
87    // inicio del conteo del tiempo
88    gettimeofday(&t_ini, NULL);
89    for (i = 0; i < num_ren_c; i++) {
90        for (j = 0; j < num_col_c; j++) {
91            Mat_C[i][j] = 0;
92            for (k = 0; k < num_col_a; k++) {
93                Mat_C[i][j] = Mat_C[i][j] + Mat_A[i][k] * Mat_B[k][j];
94            }
95        }
96    }
97    gettimeofday(&t_fin, NULL);
98    secs = timeval_diff(&t_fin, &t_ini);
99    printf("tiempo de ejecucion = %.16g milisegundos\n", secs * 1000.0);
```

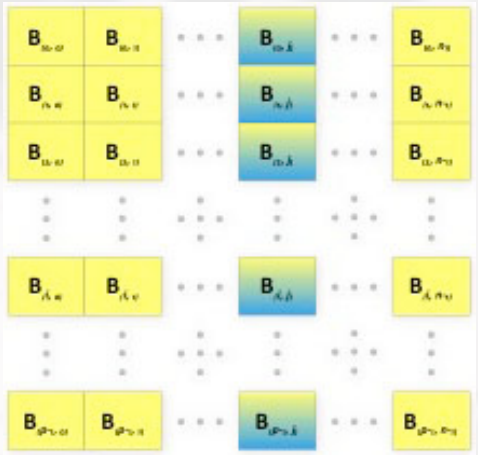
OpenMP

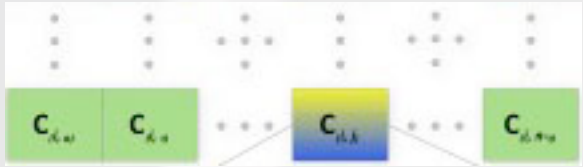




Thread de
$A_{[0:m/np][0:p]}$



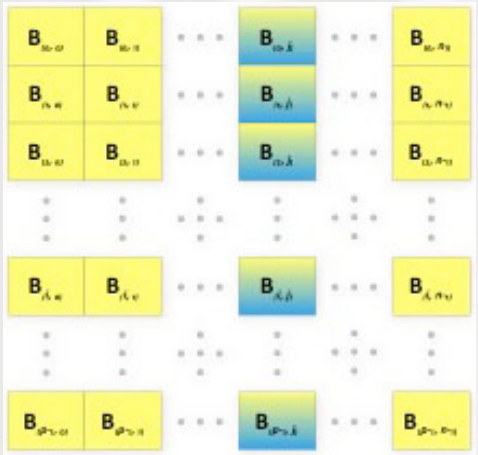


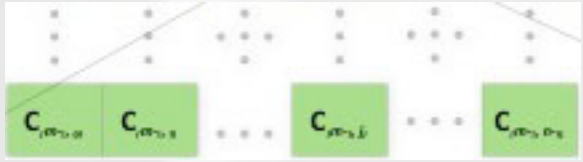


Thread de
$A_{[th*(m/np):i][0:p]}$

OpenMP

Thread de
$A_{[th*(m/np):m][0,p]}$

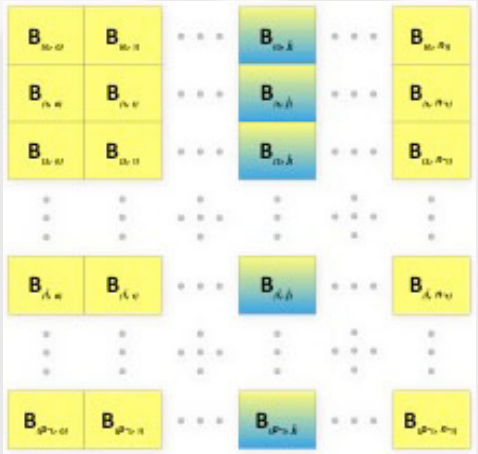


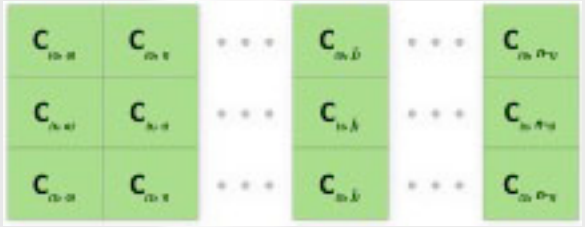


Programa OpenMP

```
86    // inicio del conteo del tiempo
87    gettimeofday(&t_ini, NULL);
88 #pragma omp parallel for default(none) shared(Mat_A, Mat_B, Mat_C,num_ren_c,num_col_c,num_col_a) private(j,k)
89    for (i = 0; i < num_ren_c; i++) {
90       for (j = 0; j < num_col_c; j++) {
91          Mat_C[i][j] = 0;
92          for (k = 0; k < num_col_a; k++) {
93             Mat_C[i][j] = Mat_C[i][j] + Mat_A[i][k] * Mat_B[k][j];
94          }
95       }
96    }
97    gettimeofday(&t_fin, NULL);
98    secs = timeval_diff(&t_fin, &t_ini);
99    printf("tiempo de ejecucion = %.16g milisegundos\n", secs * 1000.0);
```

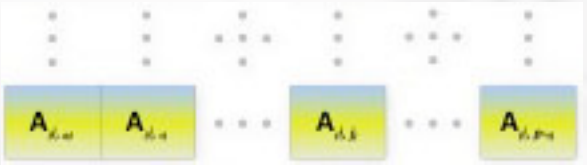
MPI

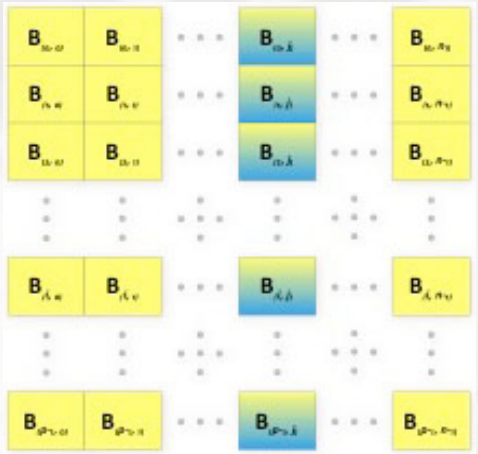


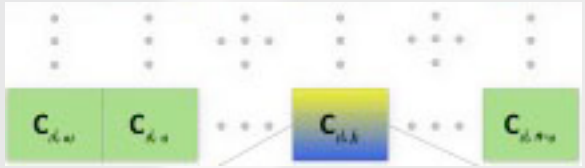


Proceso 1

$A_{[0:m/np][0:p]}$



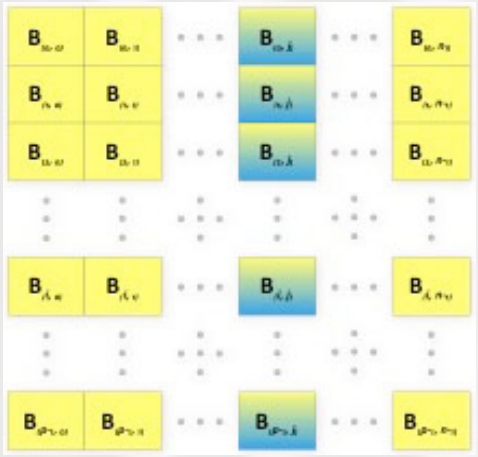


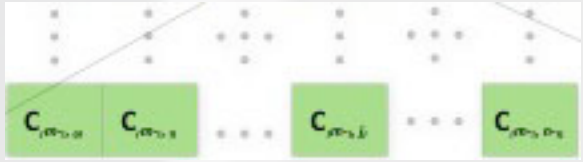


Procesos i:np

$A_{[th*(m/np):i][0:p]}$

MPI

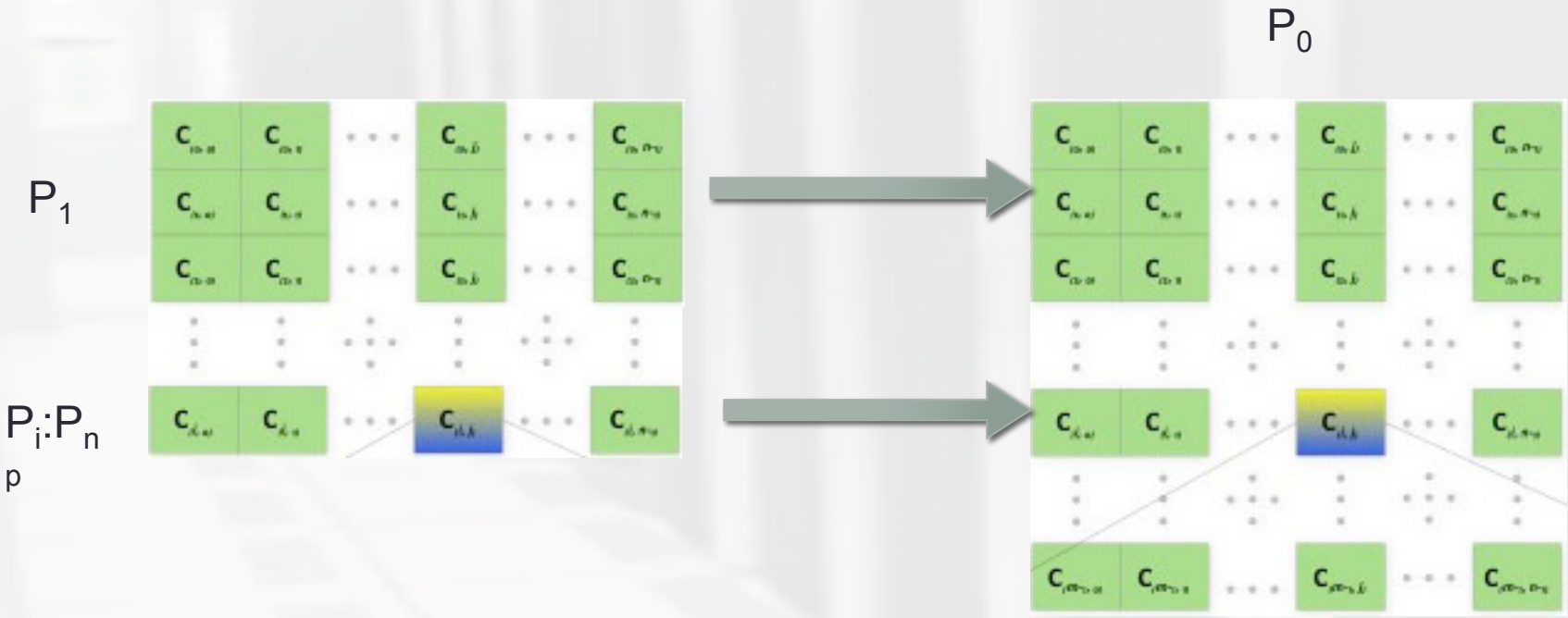




Proceso 0

$A_{[th*(m/np):m][0,p]}$

MPI



Programa MPI (master)

```
38    float Mat_A[num_ren_a][num_col_a];
39    float Mat_B[num_ren_b][num_col_b];
40    float Mat_C[num_ren_a][num_col_b];
41
42    int num_ren_c, num_col_c;
43    int rank, nprocs;
44
45    int offset, averow, extra;
46    MPI_Status status;
47    int i, j, k, rows, dest;
48
49    FILE *ptrCf;
50    char string[100];
51
52    struct timeval t_ini, t_fin;
53    double secs;
54
55    MPI_Init(&argc, &argv);
56
57    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
58    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
59
60    if ( rank == 0 ) {
61
62       if ( (ptrCf = fopen("matriz.txt", "r")) == NULL ) {
63          printf("No pudo abrirse el archivo matriz.txt \n");
64          printf("Archivo %s linea %d: \n", __FILE__, __LINE__);
65          exit(EXIT_FAILURE);
66       }
```

Programa MPI (master)

```
 94      fclose(ptrCf);
 95
 96      num_ren_c = num_ren_a;
 97      num_col_c = num_col_b;
 98
 99      averow = num_ren_a / (nprocs);
100      extra = num_ren_a % (nprocs);
101      offset = 0;
102
103      // inicio del conteo del tiempo
104      gettimeofday(&t_ini, NULL);
105      MPI_Bcast(&num_ren_a, 1, MPI_INT, 0, MPI_COMM_WORLD);
106      MPI_Bcast(&num_ren_b, 1, MPI_INT, 0, MPI_COMM_WORLD);
107      MPI_Bcast(&num_col_a, 1, MPI_INT, 0, MPI_COMM_WORLD);
108      MPI_Bcast(&num_col_b, 1, MPI_INT, 0, MPI_COMM_WORLD);
109
110      for (i = 0; i < num_ren_b; i++)
111         MPI_Bcast(&Mat_B[i][0], num_col_b, MPI_FLOAT, 0, MPI_COMM_WORLD);
112
113      for (dest = 1; dest < nprocs; dest++) {
114
115         for (i = offset; i < (offset + averow); i++)
116            MPI_Send(&Mat_A[i][0], num_col_a, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
117
118         offset = offset + averow;
119      }
120
121      for (i = offset; i < (offset + averow + extra); i++)
122         for (j = 0; j < num_col_c; j++) {
123            Mat_C[i][j] = 0.0;
124            for (k = 0; k < num_col_a; k++)
125               Mat_C[i][j] = Mat_C[i][j] + Mat_A[i][k] * Mat_B[k][j];
126         }
127
128      offset = 0;
129      for (i = 1; i < nprocs; i++) {
130
131         for (j = offset; j < (offset + averow); j++)
132            MPI_Recv(&Mat_C[j][0], num_col_c, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &status);
133
134         offset = offset + averow;
135      }
136      gettimeofday(&t_fin, NULL);
137      secs = timeval_diff(&t_fin, &t_ini);
```

Programa MPI (slave)

```
154   else {
155      MPI_Bcast(&num_ren_a, 1, MPI_INT, 0, MPI_COMM_WORLD);
156      MPI_Bcast(&num_ren_b, 1, MPI_INT, 0, MPI_COMM_WORLD);
157      MPI_Bcast(&num_col_a, 1, MPI_INT, 0, MPI_COMM_WORLD);
158      MPI_Bcast(&num_col_b, 1, MPI_INT, 0, MPI_COMM_WORLD);
159
160      num_ren_c = num_ren_a;
161      num_col_c = num_col_b;
162      averow = num_ren_a / (nprocs);
163      extra = num_ren_a % (nprocs);
164
165      for (i = 0; i < num_ren_b; i++)
166         MPI_Bcast(&Mat_B[i][0], num_col_b, MPI_FLOAT, 0, MPI_COMM_WORLD);
167
168      offset = (rank - 1) * averow;
169      for (i = offset; i < (offset + averow); i++)
170         MPI_Recv(&Mat_A[i][0], num_col_a, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
171
172      for (i = offset; i < (offset + averow); i++)
173         for (j = 0; j < num_col_c; j++) {
174            Mat_C[i][j] = 0.0;
175            for (k = 0; k < num_col_a; k++)
176               Mat_C[i][j] = Mat_C[i][j] + Mat_A[i][k] * Mat_B[k][j];
177         }
178
179      for (j = offset; j < (offset + averow); j++)
180         MPI_Send(&Mat_C[j][0], num_col_c, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
181
182   }
```

Ejemplo 2: Diferencias finitas en paralelo

La ecuación de calor en una dimensión y la forma de resolverlo mediante el método de las diferencias finitas en paralelo.

La ecuación de calor en 1D

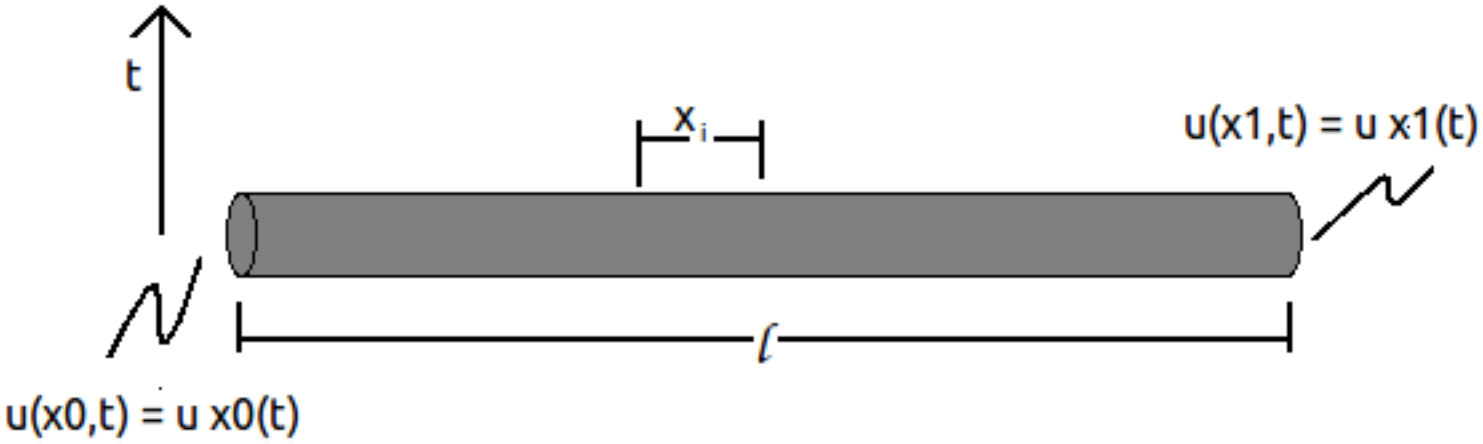
- Describe cómo se distribuye la temperatura en un cuerpo sólido en función del tiempo y el espacio.
- Representa la típica ecuación en derivadas parciales.
- La ecuación es:

$$\frac{\partial u}{\partial t} - \alpha^2 \frac{\partial^2 u}{\partial x^2} = f(x,t)$$

La ecuación de calor en 1D

- En el intervalo espacial [A, B], con condiciones de frontera:
  - u(x0,t) = u x0(t)
  - u(x1,t) = u x1(t)
- Y en el intervalo de tiempo [t0, t1], con condición inicial
  - u(x,t0) = u0(x)
- Se puede resolver numéricamente empleando el método de diferencias finitas

La ecuación de calor en 1D

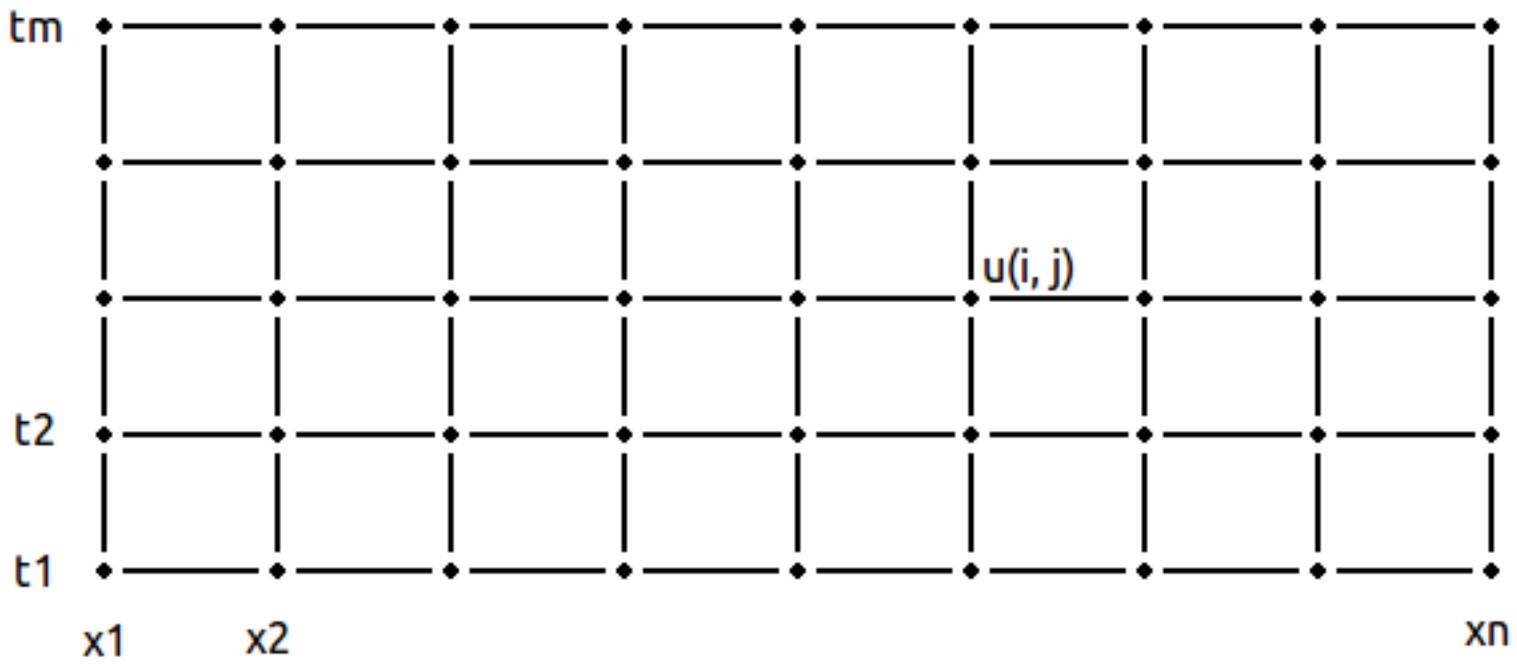


$$u(x,t0) = u0(x)$$

Diferencias finitas y la ec. de calor

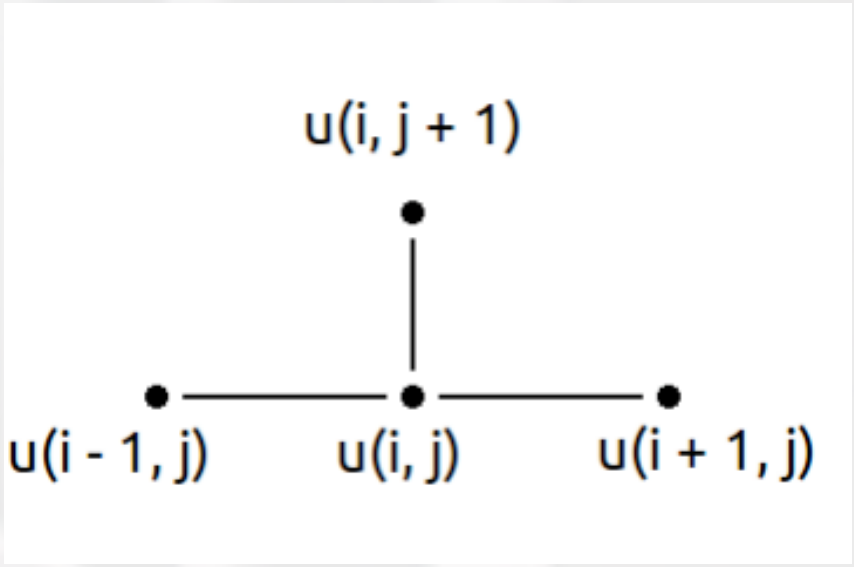
- Se divide el dominio del espacio en una malla con puntos igualmente espaciados desde x_1 a x_N.
- También se divide el tiempo de igual forma desde t_1 a t_M.
- Se denota como u(i,j) la solución aproximada en el punto x(i) en el tiempo t(j).

Diferencias finitas y la ec. de calor



Diferencias finitas y la ec. de calor

- En el punto espacial x(i) y el tiempo t(j), la ecuación diferencial discretizada define una relación entre u(i-1,j), u(i,j), u(i+1,j) y el valor en el futuro u(i,j+1).
- La relación se puede dibujar simbólicamente



Código fuente - generalidades

- La función principal sólo invoca a otras funciones, no realiza cálculos.
- 3 funciones para el cálculo de la difusión de calor en la barra.
- Para calcular los valores con los que se actualiza la malla.
- Para evaluar las condiciones iniciales.
- Para evaluar las fronteras.

Código fuente - secuencial

- Función principal:

```
int main(int argc, char **argv){
printf("\n Ecuación de calor, mediante método de diferencias\
        finitas\n");
printf(" Versión en C-OpenMP\n\n");
printf(" Resuelve la ecuación del calor en 1D dependiente del\
        tiempo.\n");
actualiza(void);  // Donde se calcula todo el proceso.
return 0;
}
```

Código fuente - secuencial

- Porción de la función actualiza:
  - Evaluación de condiciones iniciales:

```
for(i = 0;i < n; i++)
      x[i] = delta_x * (double)i;

time_new = TMIN;

// Se establecen los valores de la temperatura de la barra en
// un tiempo t0
for(i = 0; i < n; i++)
    c_actual[i] = condicion_inicial(x[i], time_new);
```

Código fuente - secuencial

- Porción de la función actualiza:
  - Evaluación de los demás puntos en la malla y condiciones de frontera:

```
for(j = 0; j < NUMPASOS; j++){
  for(i = 1; i < n - 1; i++)
    c_siguiente[i] = c_actual[i] + (delta_t*ALFA/delta_x/delta_x)*
           (c_actual[i-1] - 2.0 * c_actual[i] + c_actual[i+1])+
                                  delta_t*rhs(x[i],time_new);

 time_new = time_new + delta_t;
 c_siguiente[0] = condiciones_frontera(x[0],time_new);
 c_siguiente[n-1] = condiciones_frontera(x[n-1],time_new);

 for(i = 0; i < n; i++)
   c_actual[i] = c_siguiente[i];
}
```

Implementación en paralelo con OpenMP

- Se parte del código fuente secuencial.
- La solución de todos los puntos en la malla se calcula empleando dos ciclos, uno de ellos anidado.
- Uno para desplazamiento en el espacio, y el otro para desplazamiento en el tiempo.
- El ciclo externo es dependiente de las iteraciones previas.
- Sólo se paralelizan los ciclos relacionados con el espacio.

Código fuente - OpenMP

- Sigue el mismo esquema que el secuencial.
- Sólo se paralelizan los ciclos que son independientes entre iteraciones.
- Las directivas sólo se aplican en la función actualiza.

Código fuente - OpenMP

- Porción de la función actualiza:
- Evaluación de los demás puntos en la malla y condiciones de frontera:

```
for(j = 0; j < NUMPASOS; j++){
  for(i = 1; i < n - 1; i++)
   c_siguiente[i] = c_actual[i] + (delta_t*ALFA/delta_x/delta_x)*
            (c_actual[i-1] - 2.0 * c_actual[i] + c_actual[i+1])+
             delta_t*rhs(x[i],time_new);

  time_new = time_new + delta_t;
  c_siguiente[0] = condiciones_frontera(x[0],time_new);
  c_siguiente[n-1] = condiciones_frontera(x[n-1],time_new);

  for(i = 0; i < n; i++)
    c_actual[i] = c_siguiente[i];
}
```

Código fuente - OpenMP

- El ciclo entre pasos no se paraleliza.

```
 for(j = 0; j < NUMPASOS; j++){
 // Se paraleliza el cálculo
#pragma omp parallel for shared(c_siguiente) private(i)
   for(i = 1; i < n - 1; i++)
     c_siguiente[i] = c_actual[i] +
                      (delta_t * ALFA / delta_x / delta_x) *
                      (c_actual[i-1] - 2.0 * c_actual[i] +
                       c_actual[i+1]) +
                       delta_t * rhs(x[i], time_new);
```

Código fuente - OpenMP

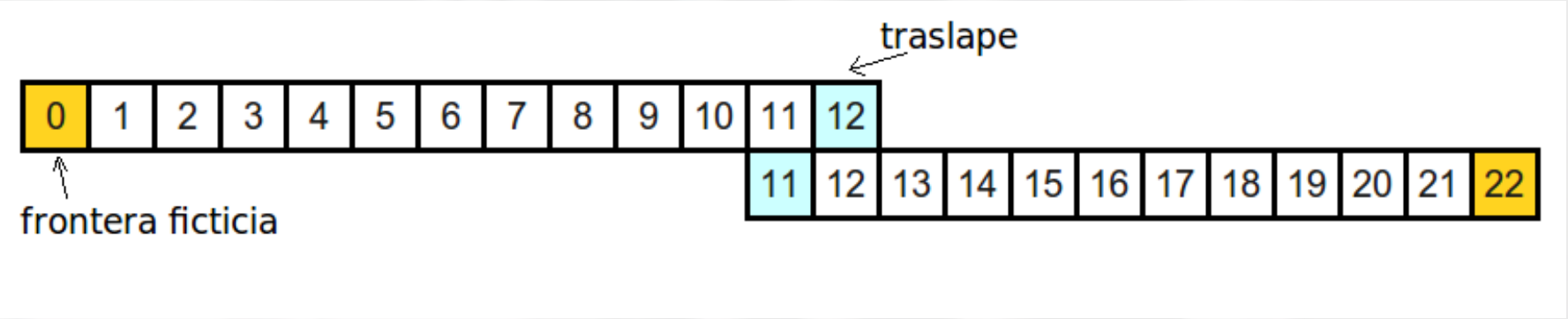
- Y se paraleliza la actualización

```
#pragma omp parallel for shared(c_actual) private (i)
      for(i = 0; i < n; i++)
         c_actual[i] = c_siguiente[i];
```

Implementación en paralelo con MPI

- Se utiliza una forma de descomposición de dominios.
- Para P procesos, se divide el intervalo [A,B] en P subintervalos iguales.
- Cada proceso configura la plantilla de ecuaciones casi de manera independiente.
- La excepción: cada proceso necesita recibir una copia de la solución de los valores de los nodos vecinos a la izquierda y la derecha.

Implementación en paralelo con MPI



Código fuente - MPI

- En la función principal se inicializa y termina el ambiente MPI

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&mi_id);
MPI_Comm_size(MPI_COMM_WORLD, &mis_proceso);
```

Código fuente - MPI

- Las demás modificaciones son en la función actualiza:
- Se considera el propio proceso y todos los procesos:

```
// El intervalo [0.0, 1.0] se divide en porciones
// equivalentes
// a mayor número de procesos, segmentos más pequeños.
 for(i = 0; i <= n + 1; i++){
   x[i] = ((double)(      id*n + i-1)* XMAX +
            (double)(p*n - id*n - i)  * XMIN) /
            (double)(p*n - 1);
 }
```

Código fuente - MPI

- Los cálculos de inicialización los realizan todos los procesos:

```
calor[0] = 0.0f;
 for(i = 1; i <= n; i++)
   calor[i] = condicion_inicial(x[i], time);
 calor[n+i] = 0.0;

 time_delta = (TFIN - TINI) / (double)(j_max - j_min);
 x_delta = (XMAX - XMIN) / (double)(p * n - 1);
```

Código fuente - MPI

- Como es memoria distribuida, es necesario compartir "fronteras".
- Se comparte hacia la izquierda:

```
...
 if(0 < id){
   etiqueta = 1;
   MPI_Send(&calor[1], 1, MPI_DOUBLE, id - 1, etiqueta, \
            MPI_COMM_WORLD);
 }
 if(id < p - 1){
   etiqueta = 1;
   MPI_Recv(&calor[n+1], 1, MPI_DOUBLE, id + 1, etiqueta,\
            MPI_COMM_WORLD, &status);
 }
```

Código fuente - MPI

- Se comparten las fronteras, ahora hacia la derecha:

```
...
    if(id < p - 1){
      etiqueta = 2;
      MPI_Send(&calor[n], 1, MPI_DOUBLE, id + 1, etiqueta, \
               MPI_COMM_WORLD);
    }
    if(0 < id){
      etiqueta = 2;
      MPI_Recv(&calor[0], 1, MPI_DOUBLE, id - 1, etiqueta, \
              MPI_COMM_WORLD, &status);
    }
```

Código fuente - MPI

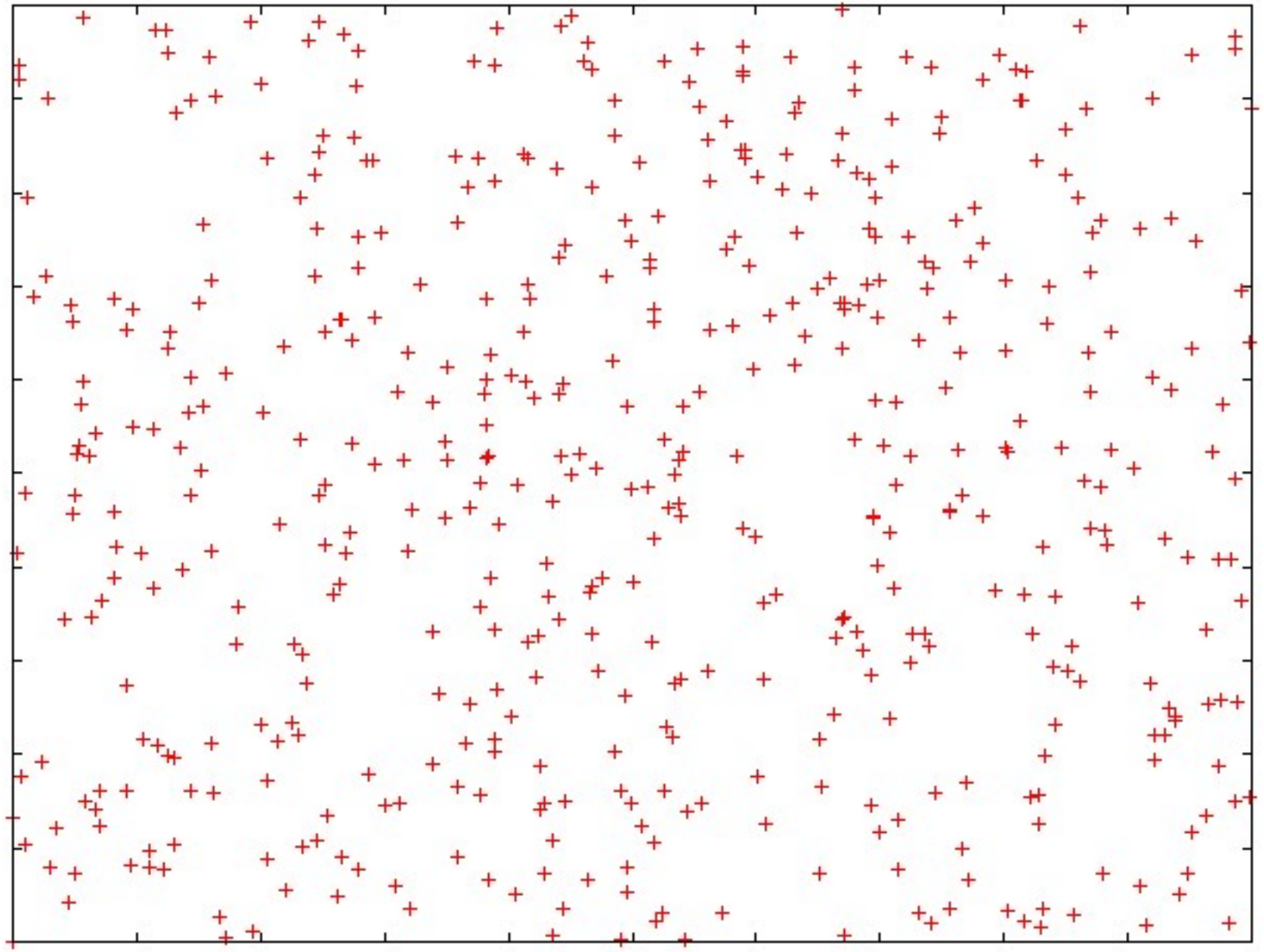
- Finalmente se actualizan los valores de frontera:

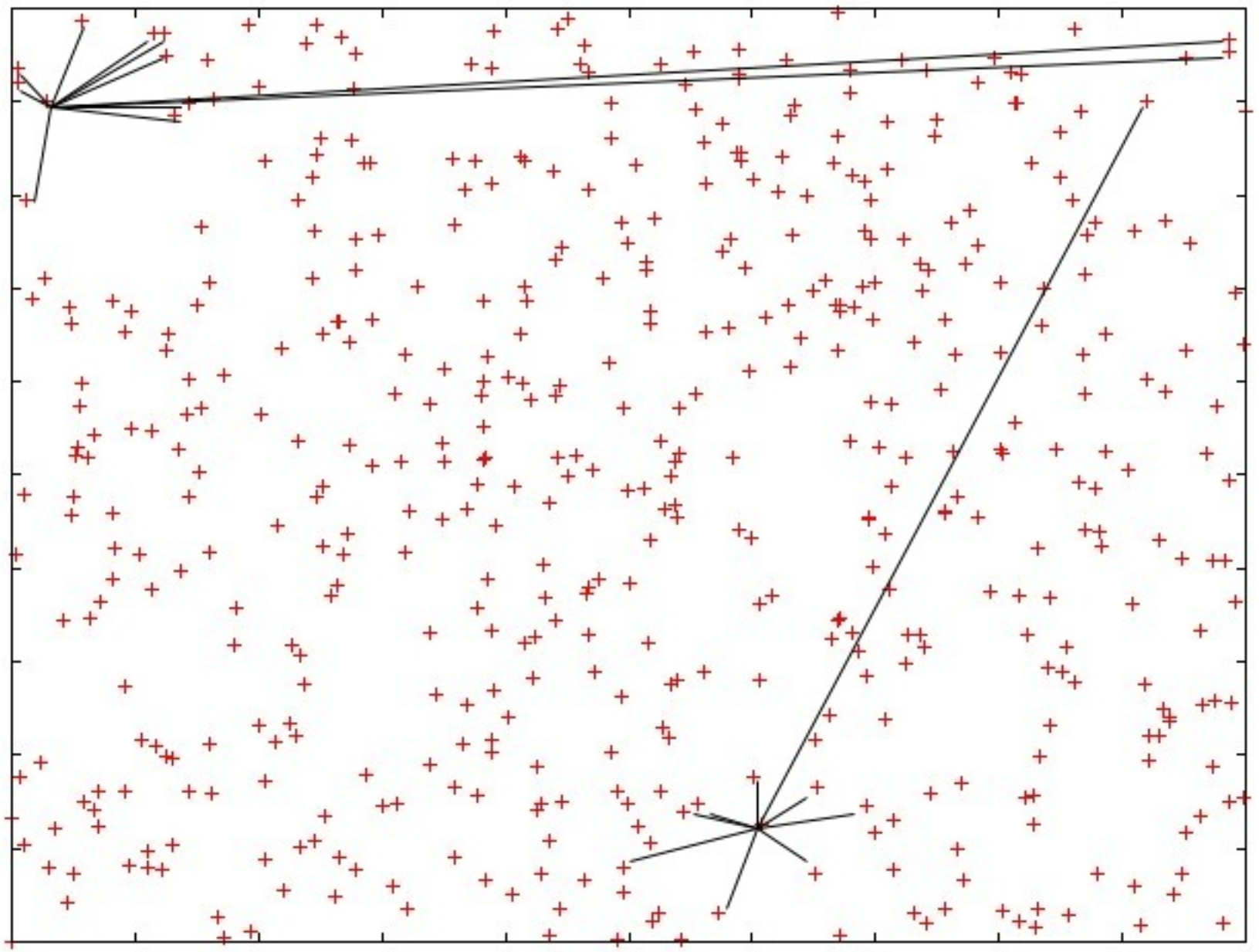
```
...
    if(0 == id)
      calor_nuevo[1] = condicion_frontera(x[1], time_new);
    if(id == p - 1)
      calor_nuevo[n] = condicion_frontera(x[n], time_new);
```

Ejemplo 3: Seguimientos de partículas

Juego de partículas

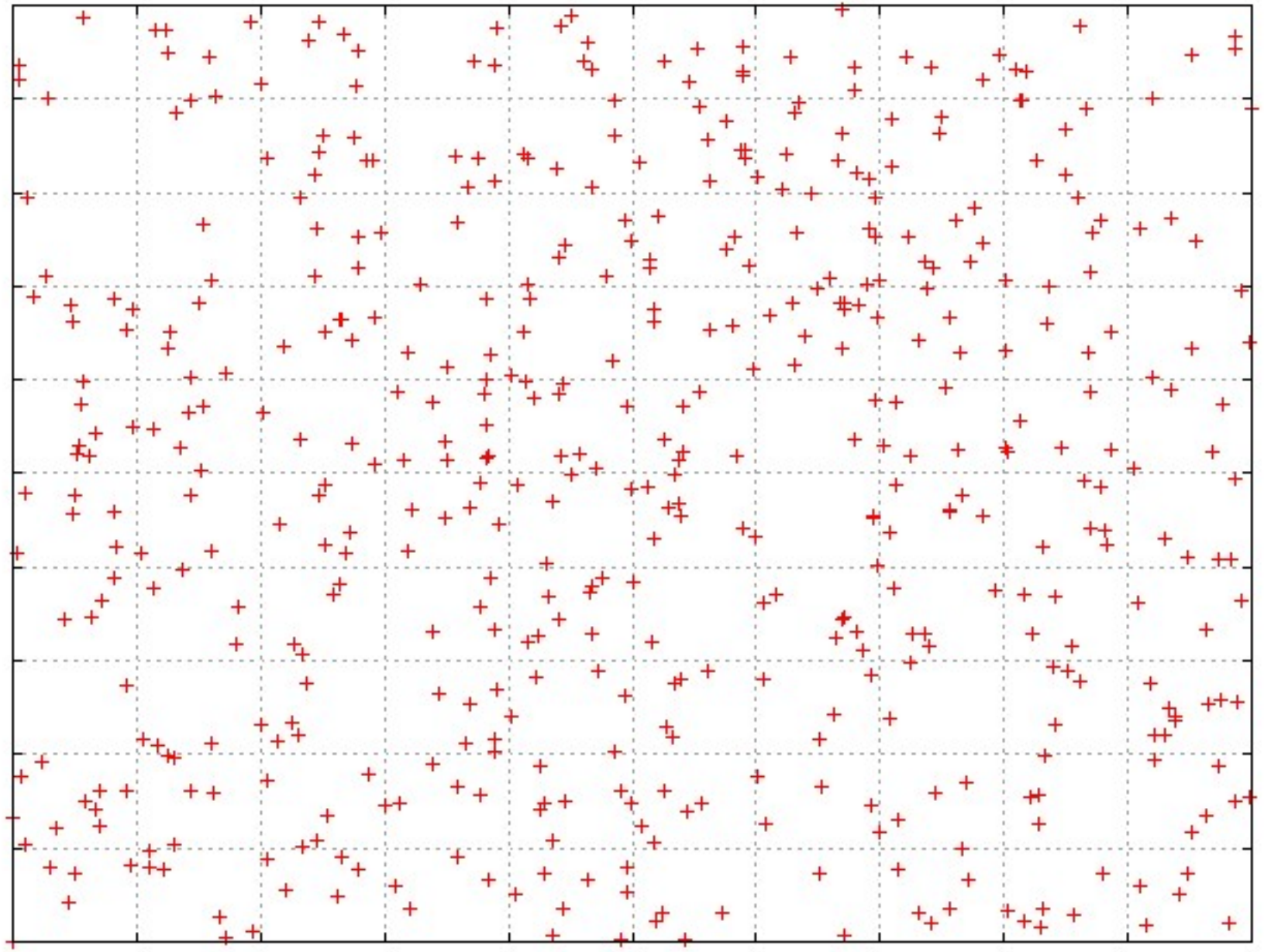
- Existe un cierto número de partículas en un espacio delimitado
- Las partículas se mueven por “fuerzas” generadas por otras partículas:
  - En este juego, $f_{ij}=1/d_{ij}$, $x_i(t+1)=x_i(t) + f_i(t)$
  - $f_i(t)$ es la suma de las fuerzas provocadas por todas las partículas sobre la partícula i

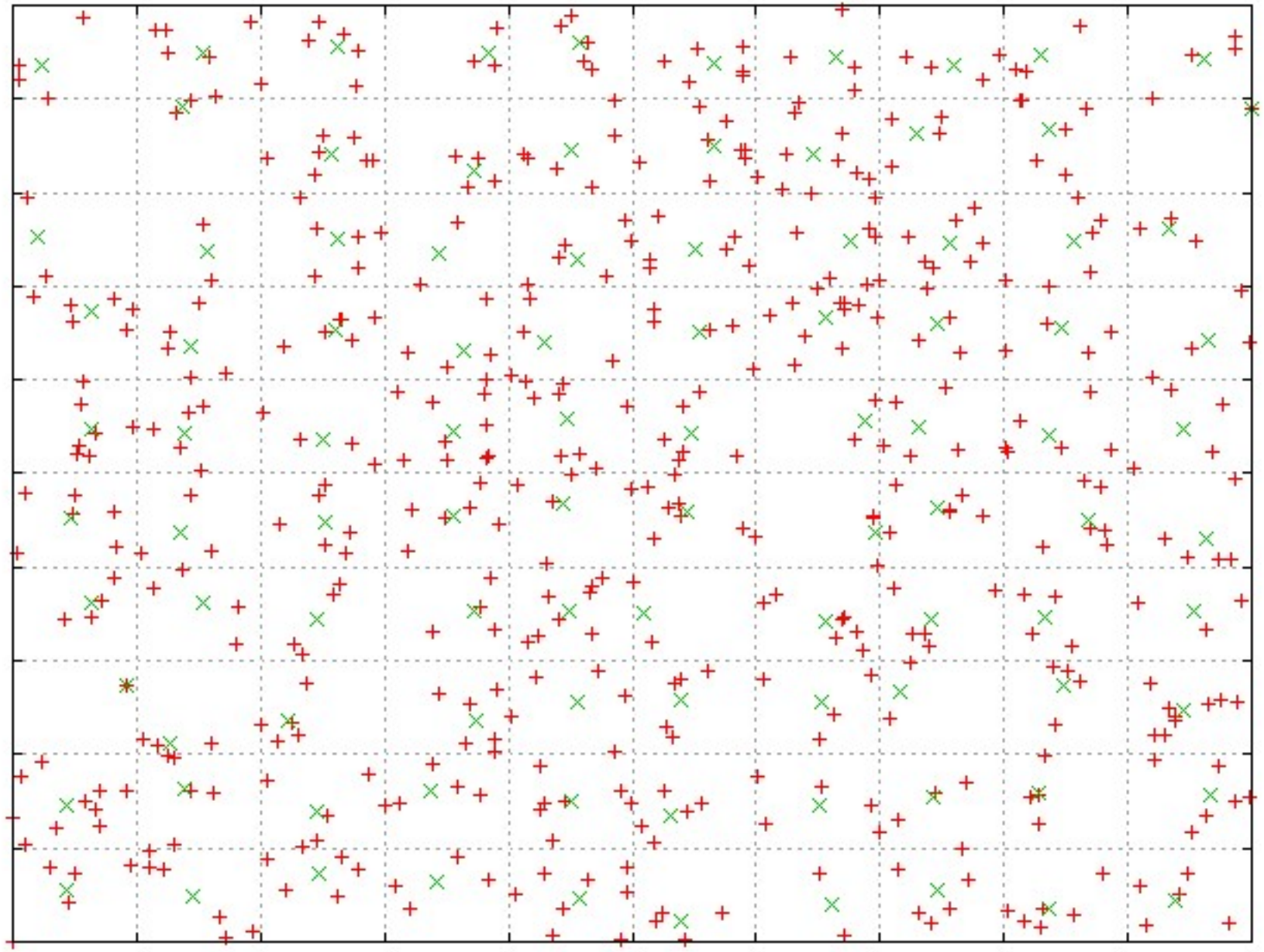


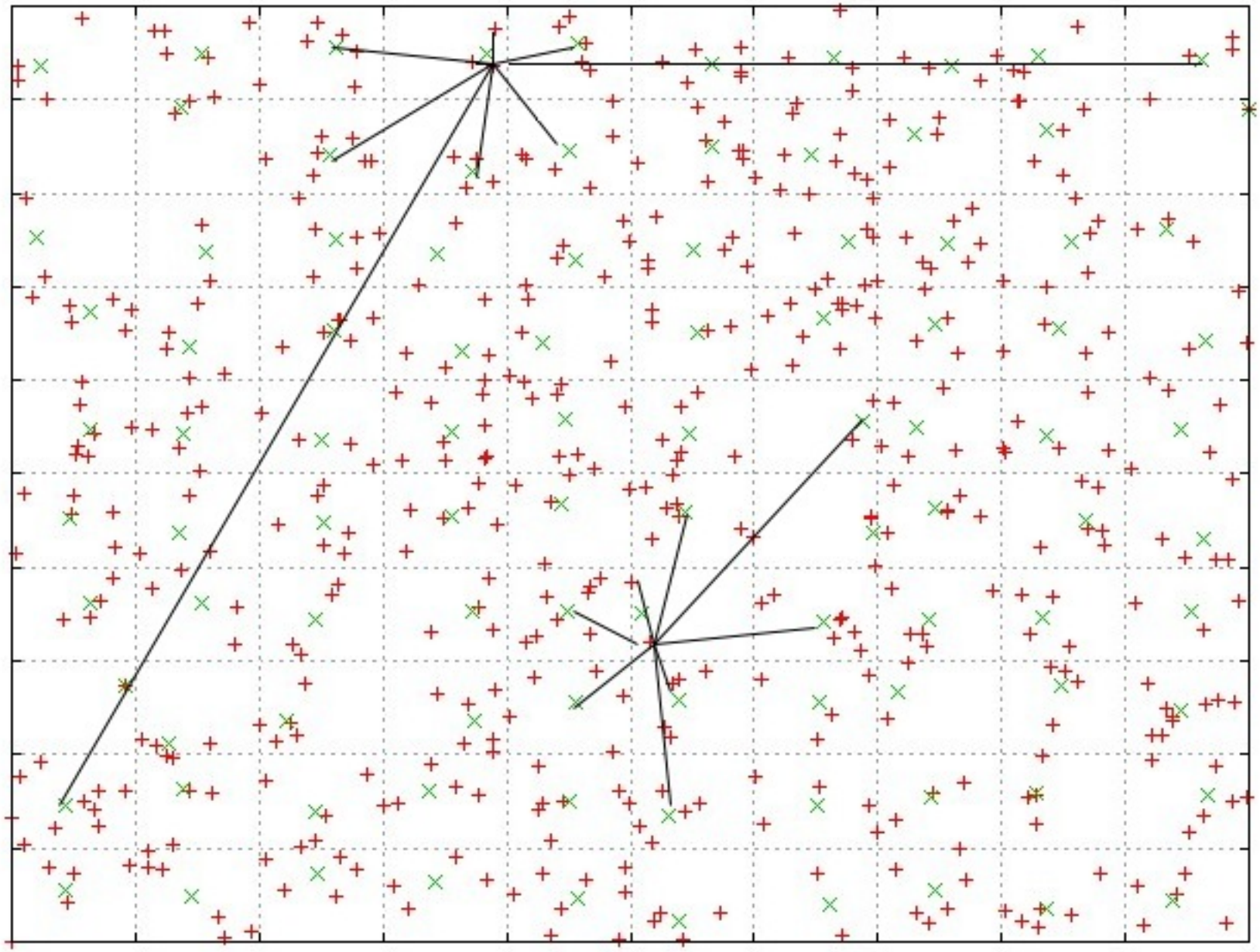


Juego de partículas

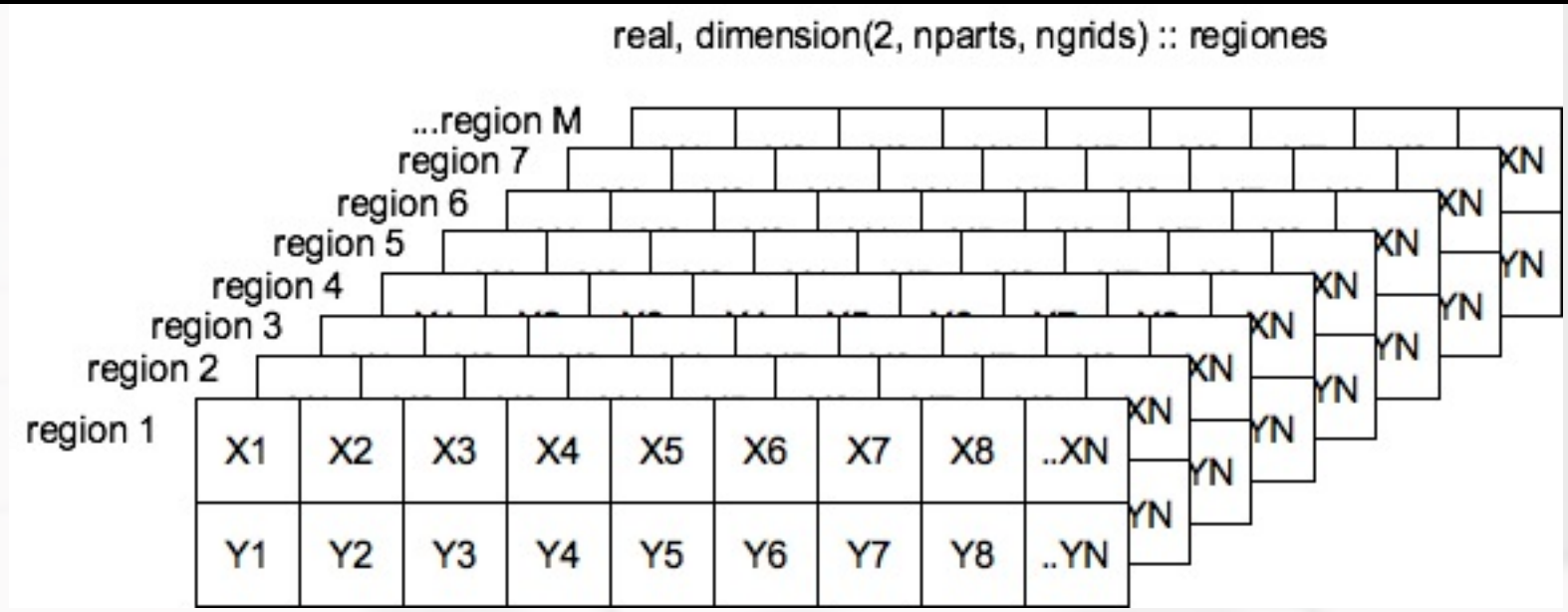
- Para reducir el número de términos en cada f_i, se sigue la siguiente estrategia:
  - Dividir el espacio en regiones
  - Cada región tiene un número determinado de partículas
  - Cada región tiene un "centro" (el centroide de las partículas de la región)
  - $f_i = f_{il} + f_{ig}$
  - f_{il} es la fuerza provocada por las partículas vecinas
  - f_{ig} es la fuerza calculada a partir de los centros de todas las demás regiones







```
PROGRAM NBODY
implicit none
integer, parameter :: ngrids_d=10, nparts=500, stepf=90, ngrids=ngrids_d*ngrids_d
real, parameter :: max_coord=1000.0, tam_grid=max_coord/ngrids_d

real, dimension(2, nparts, ngrids) :: regiones
real, dimension(2, ngrids) :: centros
real, dimension(2, nparts) :: fuerza
integer, dimension(ngrids) :: parts_region
real, dimension(3, nparts) :: parts_cambio
integer, dimension(ngrids_d, ngrids_d) :: map_region
integer :: cparts_cambio, nparts_cambio

real :: x, y, fx, fy, distx, disty, dist

integer :: xgrid, ygrid, nvareg, cregion, cregion2, rregion, cparts, cparts2, region, step
```



```
 cregion=1
 do xgrid=1, ngrids_d
  do ygrid=1, ngrids_d
     map_region(ygrid, xgrid) = cregion
     parts_region(cregion) = 0
     cregion = cregion +1
    end do
 end do

open(unit=20, file="particules-1")
do cparts=1,nparts
    read (20, *) x, y
    xgrid = ceiling(x/tam_grid)
    ygrid = ceiling(y/tam_grid)
    region=map_region(xgrid, ygrid)
    parts_region(region)=parts_region(region) + 1
    regiones(1,parts_region(region), region)=x
    regiones(2,parts_region(region), region)=y
enddo
close(20)
```

```
! ciclo de los pasos de la simulacion

      do step=1, stepf

! Los centros
        do cregion= 1, ngrids
         if (parts_region(cregion) .gt. 0) then
           x = 0
           y = 0
           do cparts= 1, parts_region(cregion)
              x= x + regiones(1, cparts, cregion)
              y= y + regiones(2, cparts, cregion)
           end do
           centros(1, cregion) = x / parts_region(cregion)
           centros(2, cregion) = y / parts_region(cregion)
         else
           centros(1, cregion) = -1
         end if
       enddo
```

```
! en cada paso contamos cuantas particulas cambian de region
       nparts_cambio = 0
!recorremos las regiones
       do cregion= 1, ngrids

! Cada particula en cada region acumulara una fuerza en cada ciclo
         do cparts= 1, parts_region(cregion)
           fuerza(1, cparts) = 0
           fuerza(2, cparts) = 0
         end do

! FUERZAS LOCALES. CADA PARTICULA RESIENTE LA FUERZA DE CADA UNA DE SUS VECINAS. LA FUERZA ES RECIPROCA
         do cparts=1, parts_region(cregion) - 1
           do cparts2= cparts + 1, parts_region(cregion)
              distx = regiones(1, cparts, cregion) - regiones(1, cparts2, cregion)
              disty = regiones(2, cparts, cregion) - regiones(2, cparts2, cregion)
              dist = sqrt(distx*distx* + disty*disty)
              fx = distx/dist;
              fy = disty/dist;
              fuerza(1, cparts) = fuerza(1, cparts) - fx
              fuerza(2, cparts) = fuerza(2, cparts) - fy
              fuerza(1, cparts2) = fuerza(1, cparts2) + fx
              fuerza(2, cparts2) = fuerza(2, cparts2) + fy
           end do
         end do
```

```
! FUERZAS GLOBALES. CADA PARTICULA RESIENTE LA FUERZA DE CADA UNO DE LOS CENTROIDES
! (excepto el suyo y las regiones vacias)
        do cparts= 1, parts_region(cregion)
          do rregion=0, ngrids - 2
            cregion2 = mod( cregion + rregion, ngrids) + 1
            if (centros(1, cregion2) .ne. -1 ) then
               distx = regiones(1, cparts, cregion) - centros(1, cregion2)
               disty = regiones(2, cparts, cregion) - centros(2, cregion2)
               dist = sqrt(distx*distx* + disty*disty)
               fx = distx/dist;
               fy = disty/dist;
               fuerza(1, cparts) = fuerza(1, cparts) + fx
               fuerza(2, cparts) = fuerza(2, cparts) + fy
            end if
          end do
        end do
```

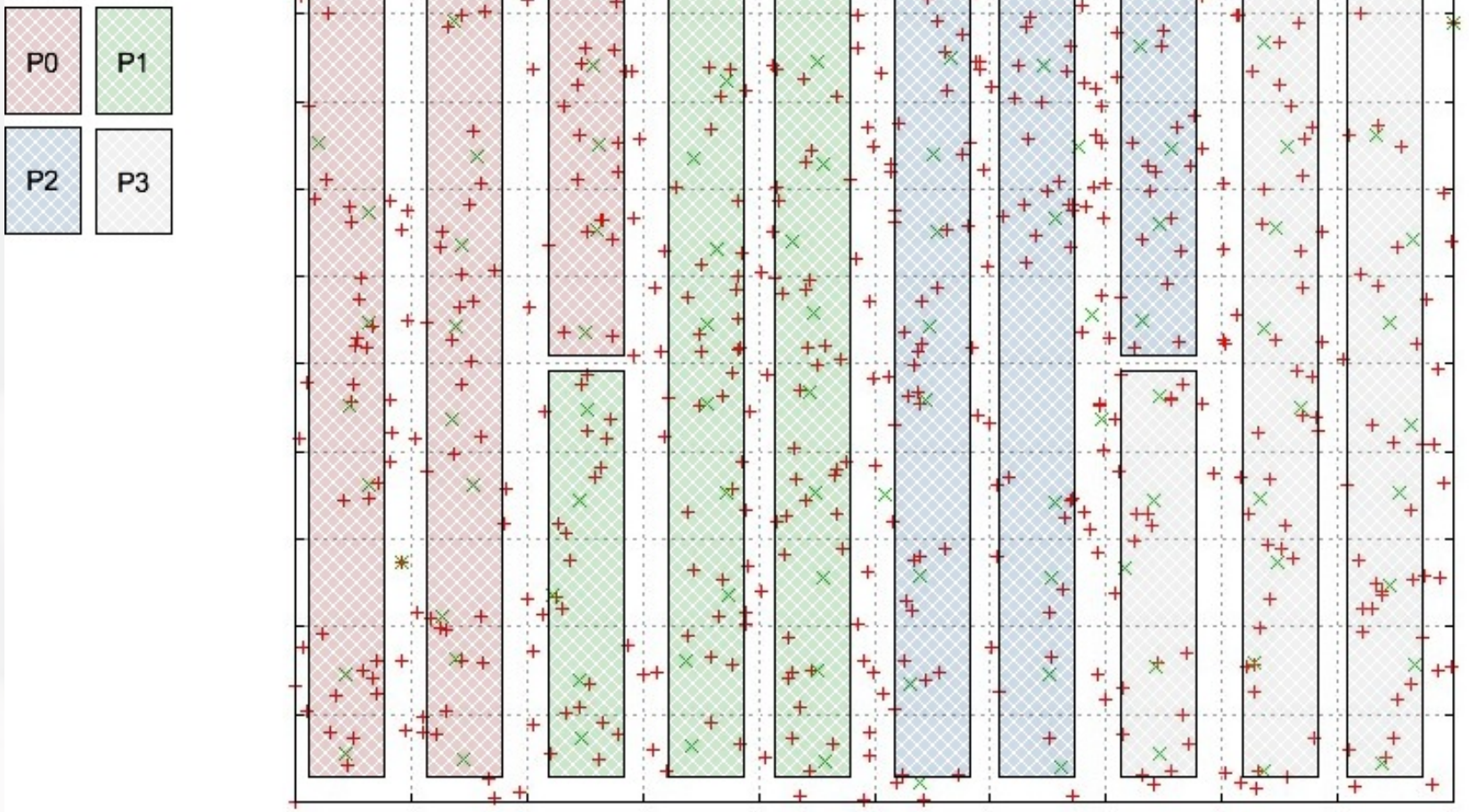
```
! CADA PARTICULA SE MUEVE DE ACUERDO A LA FUERZA TOTAL ACUMULADA.
        do cparts= parts_region(cregion), 1, -1
          regiones(1, cparts, cregion) = regiones(1, cparts, cregion) + fuerza(1, cparts)
          regiones(2, cparts, cregion) = regiones(2, cparts, cregion) + fuerza(2, cparts)
          xgrid = ceiling(regiones(1, cparts, cregion)/tam_grid)
          ygrid = ceiling(regiones(2, cparts, cregion)/tam_grid)
! SI LA PARTICULA SE SALE DE LA REGION DE ESTUDIO, DESAPARECE
          if (xgrid .gt. 0 .and. xgrid .le. ngrids_d .and. ygrid .gt. 0 .and. ygrid .le.  ngrids_d) then
            nvareg = map_region(xgrid, ygrid)
          else
             nvareg = -1
          end if
! SI LA NUEVA REGION NO CORRESPONDE A LA ACTUAL, LA PASAMOS A LA NUEVA O DESAPARECE
          if (nvareg .ne. cregion) then
            if( nvareg .gt. 0) then
               nparts_cambio = nparts_cambio + 1 ! cuenta de cuantas cambiaron
               parts_cambio(1, nparts_cambio) = regiones(1, cparts, cregion)
               parts_cambio(2, nparts_cambio) = regiones(2, cparts, cregion)
               parts_cambio(3, nparts_cambio) = nvareg
            end if
            regiones(1, cparts, cregion) = regiones(1, parts_region(cregion), cregion)
            regiones(2, cparts, cregion) = regiones(2, parts_region(cregion), cregion)
            parts_region(cregion) = parts_region(cregion) - 1
          end if
        end do
      end do ! termina do que recorre las regiones
```

```
erPoint
! CAMBIAMOS LAS PARTICULAS A SU NUEVA REGION
      do cparts_cambio=1, nparts_cambio
         nvareg= parts_cambio(3, cparts_cambio)
         parts_region(nvareg) = parts_region(nvareg) + 1
         regiones(1, parts_region(nvareg), nvareg) = parts_cambio(1, cparts_cambio)
         regiones(2, parts_region(nvareg), nvareg) = parts_cambio(2, cparts_cambio)
      end do
```

¿ Por qué esperar hasta el final para mover las partículas a su nueva región?

Reparto de carga

- El trabajo se puede dividir de dos maneras:
  - Repartiendo las partículas entre los procesadores
  - Repartiendo las regiones entre los procesadores
- Es necesario evaluar cuál genera más “paralelismo” y menos comunicaciones
  - En este caso ejemplificamos el reparto de regiones
  - Cada procesador:
    - Calcula las fuerzas locales y globales de las partículas que están en sus regiones
    - Mueve las partículas que están en sus regiones
    - Calcula los centros de sus regiones
  - ¿Cómo se lleva a cabo la reubicación de partículas?


P0
P1
P2
P3

Implementación con OpenMP

- Esta estrategia se puede implementar en OpenMP de forma sencilla
  - Simplemente hay que paralelizar los ciclos que recorren las regiones
  - Los puntos a considerar son:
    - ¿Cuáles variables deben ser privadas?
    - Evitar sobreescribir en variables compartidas

```
! Ciclo de los pasos de la simulacion
      do step=1, stepf

! Los centros
!$OMP parallel do private(x, y, cparts, cregion)
        do cregion= 1, ngrids
         if (parts_region(cregion) .gt. 0) then
           x = 0
           y = 0
           do cparts= 1, parts_region(cregion)
              x= x + regiones(1, cparts, cregion)
              y= y + regiones(2, cparts, cregion)
           end do
           centros(1, cregion) = x / parts_region(cregion)
           centros(2, cregion) = y / parts_region(cregion)
         else
           centros(1, cregion) = -1
         end if
       enddo
!$OMP end parallel do
```

Calculamos los centros en paralelo (cada hebra recibe una porción de regiones)
regiones, **parts_region y centros** son variables compartidas
Cada hebra escribe su respectiva y no-traslapada porción de regiones

```
! en cada paso contamos cuantas particulas cambian de region
      nparts_cambio = 0
!$OMP Parallel do private(cparts, cregion, fuerza, cparts2, fx, fy, nvareg, xgrid, ygrid, cregion2, disty, distx, dist, rregion)
! recorremos las regiones (en paralelo)
      do cregion= 1, ngrids
! Cada particula en cada region acumulara una fuerza en cada ciclo
        do cparts= 1, parts_region(cregion)
          fuerza(1, cparts) = 0
          fuerza(2, cparts) = 0
        end do
! FUERZAS LOCALES. CADA PARTICULA RESIENTE LA FUERZA DE CADA UNA DE SUS VECINAS. LA FUERZA ES RECIPROCA
        do cparts=1, parts_region(cregion) - 1
          do cparts2= cparts + 1, parts_region(cregion)
             distx = regiones(1, cparts, cregion) - regiones(1, cparts2, cregion)
             disty = regiones(2, cparts, cregion) - regiones(2, cparts2, cregion)
             dist = sqrt(distx*distx* + disty*disty)
             fx = distx/dist;
             fy = disty/dist;
             fuerza(1, cparts) = fuerza(1, cparts) - fx
             fuerza(2, cparts) = fuerza(2, cparts) - fy
             fuerza(1, cparts2) = fuerza(1, cparts2) + fx
             fuerza(2, cparts2) = fuerza(2, cparts2) + fy
          end do
        end do
```

- Calculamos fuerzas locales, “globales” y movimientos en paralelo
- Cada hebra se encarga de su porción de regiones
- En cada región, se calcula la acumulación de fuerzas para cada partícula
- **fuerza** es local a cada hebra.+

```
! SI LA NUEVA REGION NO CORRESPONDE A LA ACTUAL, LA PASAMOS A LA NUEVA O DESAPARECE
          if (nvareg .ne. cregion) then
            if( nvareg .gt. 0) then
            !$OMP ATOMIC
               nparts_cambio = nparts_cambio + 1
               parts_cambio(1, nparts_cambio) = regiones(1, cparts, cregion)
               parts_cambio(2, nparts_cambio) = regiones(2, cparts, cregion)
               parts_cambio(3, nparts_cambio) = nvareg
            end if
            regiones(1, cparts, cregion) = regiones(1, parts_region(cregion), cregion)
            regiones(2, cparts, cregion) = regiones(2, parts_region(cregion), cregion)
            parts_region(cregion) = parts_region(cregion) - 1
          end if
        end do
      end do ! termina do que recorre las regiones
!$OMP end parallel do
```

- **nparts_cambio** y **parts_cambio** son compartidas
- Si dos hebras obtienen el mismo valor de **nparts_cambio** escribirán en la misma “casilla” en **parts_cambio**
- Deben modificarse y asignarse por sólo una hebra a la vez

```
      end do ! termina do que recorre las regiones
!$OMP end parallel do

! CAMBIAMOS LAS PARTICULAS A SU NUEVA REGION
      do cparts_cambio=1, nparts_cambio
         nvareg= parts_cambio(3, cparts_cambio)
         parts_region(nvareg) = parts_region(nvareg) + 1
         regiones(1, parts_region(nvareg), nvareg) = parts_cambio(1, cparts_cambio)
         regiones(2, parts_region(nvareg), nvareg) = parts_cambio(2, cparts_cambio)
      end do
```

- El trabajo en paralelo termina al mover todas las partículas
- La relocalización se hace en forma serial

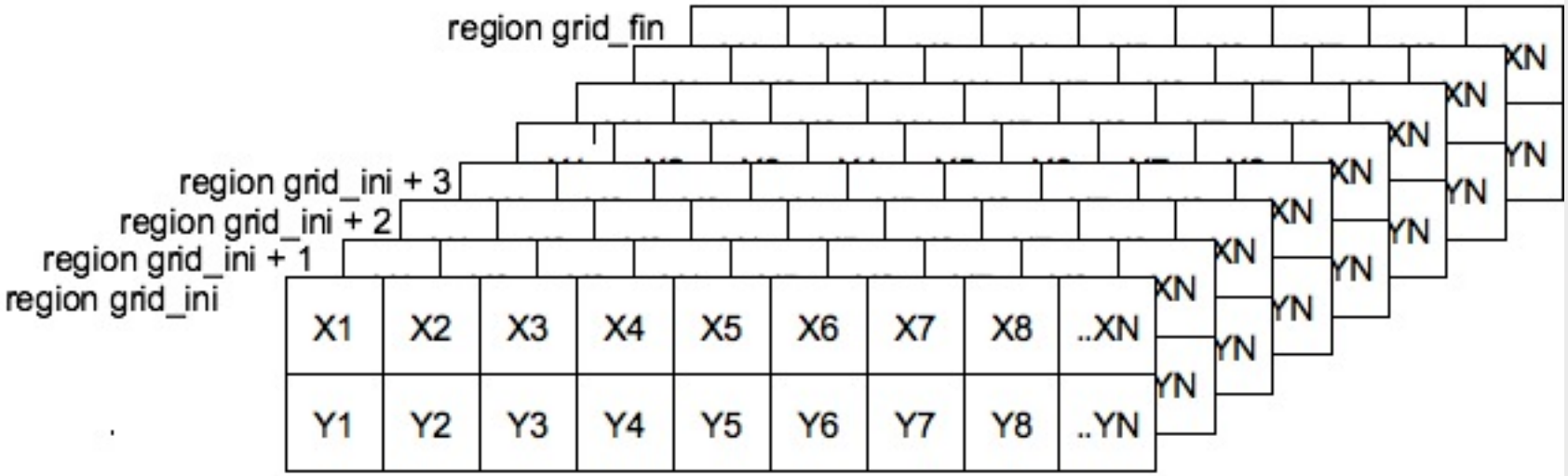
Implementación con MPI

- La implementación usando MPI requiere definir las partes de las estructuras de datos que cada proceso va a usar y a requerir
  - Cada proceso usa y actualiza una parte de la estructura de regiones
  - Cada proceso requiere una versión actualizada de la estructura de centros
  - Cada proceso requiere recibir una estructura con sus nuevas partículas –si es el caso

```
implicit none
include 'mpif.h'
integer, parameter :: ngrids_d=10, nparts=500, stepf=5, ngrids=ngrids_d*ngrids_d
integer, parameter :: MAX_PROC = 32
real, parameter :: max_coord=1000.0, tam_grid=max_coord/ngrids_d, PI=3.1416, PI_2=PI/2.0

real, allocatable, dimension(:,:,:) :: regiones
integer, allocatable, dimension(:) :: parts_region

real, dimension(2, ngrids) :: centros

real, dimension(2, nparts) :: fuerza
real, dimension(3, nparts) :: parts_cambio, parts_cambio_global

integer, dimension(ngrids_d, ngrids_d) :: map_region
integer :: cparts_cambio, nparts_cambio, nparts_cambio_global
integer, dimension (MAX_PROC) :: nparts_cambio_vector, distl

integer :: i, j

real :: x, y, fx, fy, distx, disty, dist

integer :: xgrid, ygrid, nvareg, cregion, cregion2, rregion, cparts, cparts2, region, step
integer :: rank, isize, slice, desbalance, ierror, grid_ini, grid_fin
```

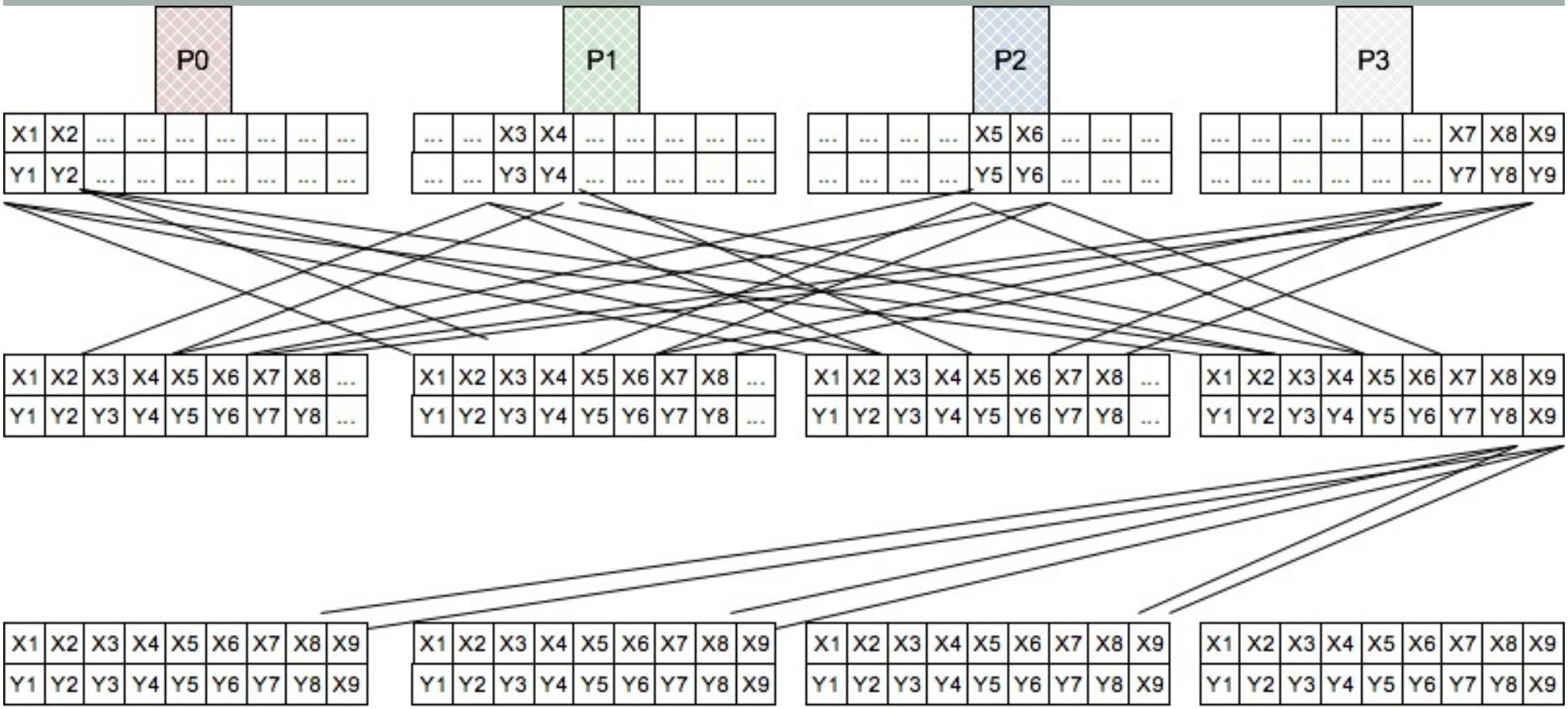
```
call mpi_init(ierror)
call mpi_comm_rank(mpi_comm_world, rank, ierror)
call mpi_comm_size(mpi_comm_world, isize, ierror)

slice= ngrids/isize
grid_ini= slice*rank + 1
grid_fin= slice*(rank + 1)
if ( rank .eq. isize -1) then
    grid_fin = ngrids
end if
desbalance = ngrids - (slice*isize)

allocate (regiones(2, nparts, grid_ini:grid_fin), parts_region(grid_ini:grid_fin))

cregion=1
do i=1, ngrids_d
 do j=1, ngrids_d
   map_region(j, i) = cregion
   cregion = cregion +1
  end do
end do
```

```
real, dimension(:, :, :) :: regiones
allocate(regiones(2,nparts,grid_ini:grid_fin))
```



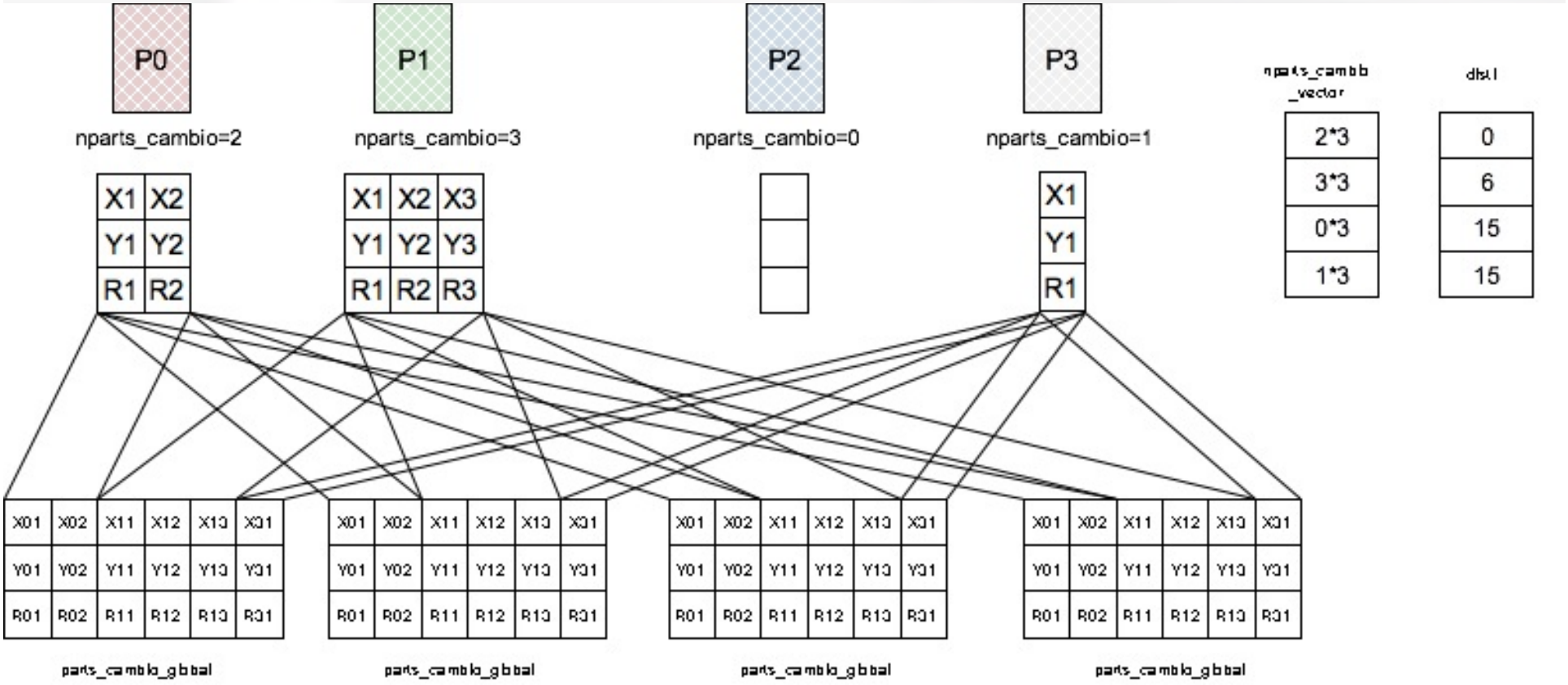
```
open(unit=20, file="particules-l")
do cparts=1,nparts
   read (20, *) x, y
   xgrid = ceiling(x/tam_grid)
   ygrid = ceiling(y/tam_grid)
   region=map_region(xgrid, ygrid)
   if (region .ge. grid_ini .and. region .le. grid_fin) then
      parts_region(region)=parts_region(region) + 1
      regiones(1,parts_region(region), region)=x
      regiones(2,parts_region(region), region)=y
   end if
enddo
close(20)
```

```
      do cregion= grid_ini, grid_fin
         if (parts_region(cregion) .gt. 0) then
           x = 0
           y = 0
           do cparts= 1, parts_region(cregion)
             x= x + regiones(1, cparts, cregion)
             y= y + regiones(2, cparts, cregion)
           end do
           centros(1, cregion) = x / parts_region(cregion)
           centros(2, cregion) = y / parts_region(cregion)
         else
          centros(1, cregion) = -1
         end if
      enddo
! sincronizamos los centros
      call mpi_allgather(MPI_IN_PLACE, slice*2, MPI_REAL, centros(1,1), slice*2, MPI_REAL, MPI_COMM_WORLD, ierror)
      if (desbalance .gt. 0) then
         call mpi_bcast(centros(1, slice*isize +1), desbalance*2, isize- 1, MPI_COMM_WORLD, ierror)
      end if
```




```
call mpi_allgather(MPI_IN_PLACE, slice*2, MPI_REAL, centros(1,1), slice*2, MPI_REAL, MPI_COMM_WORLD, ierror)
if (desbalance .gt. 0) then
   call mpi_bcast(centros(1, slice*isize +1), desbalance*2, isize- 1, MPI_COMM_WORLD, ierror)
end if
```

```
      call mpi_allgather(nparts_cambio, 1, mpi_integer,  nparts_cambio_vector, 1, mpi_integer, mpi_comm_world, ierror)
      distl(1)=0
      do i=1, isize-1
         nparts_cambio_vector(i) = nparts_cambio_vector(i) * 3
         distl(i+1)=distl(i) + nparts_cambio_vector(i)
      end do

      call mpi_allgatherv(parts_cambio, nparts_cambio, mpi_real, parts_cambio_global, nparts_cambio_vector,distl, mpi_real, mpi_comm_world, ierror)

! CAMBIAMOS LAS PARTICULAS A SU NUEVA REGION
      nparts_cambio_global= distl(isize) + nparts_cambio_vector(isize)
      do cparts_cambio=1, nparts_cambio_global
         nvareg= parts_cambio_global(3, cparts_cambio)
         if( nvareg .ge. grid_ini .and. nvareg .le. grid_fin) then
            parts_region(nvareg) = parts_region(nvareg) + 1
            regiones(1, parts_region(nvareg), nvareg) = parts_cambio_global(1, cparts_cambio)
            regiones(2, parts_region(nvareg), nvareg) = parts_cambio_global(2, cparts_cambio)
         end if
      end do
```




```
call mpi_allgatherv(parts_cambio, nparts_cambio, mpi_real, parts_cambio_global, nparts_cambio_vector,distl, mpi_real, mpi_comm_world, ierror)
```