Tema 6.1. Algoritmos paralelos Diseño.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Índice

Índice

Objetivos

- Establecer consideraciones **generales** sobre el diseño de **algoritmos paralelos**.
- Demostrar la **complejidad** del diseño de este tipo **algoritmos** así como la necesidad de un enfoque **creativo** en dicho proceso.
- Establecer una **distinción** clara entre aspectos **independientes** de la máquina y **dependientes** de la misma.
- Estudiar un **estándar** de programación **paralela** como es **MPI**.

Índice

Modelo Tarea-Canal

- El modelo **tarea/canal** está compuesto por "tareas" y "canales".
- Se establece una equivalencia entre tareas como **programa**, una **memoria local** y una serie de **puertas** de entradas/salidas.
- Los canales son una **cola de mensajes** que establece una **conexión** entre la puerta de **salida** de una tarea y la puerta de **entrada** de otra.

Tareas Bloqueadas

- Cuando el **mensaje** no ha llegado, la tarea que **recibe** se encuentra **bloqueada**.
- Las tareas que **sólo envían** nunca se bloquean aún cuando mensajes enviados por ellas **previamente** aún no han sido **recibidos** por sus **destinatarios**.
- Se **considera** por tanto la **recepción** como una actividad **síncrona** y el envío una actividad **asíncrona**.

La metodología de diseño de Foster I

El diseño involucra cuatro etapas ¿Secuenciales?:

- **Particionar**.
- **Comunicar**.
- **Aglutinar** (Agrupar)
- **Asignar**.

La metodología de diseño de Foster II

- **Particionar**: Tanto el **código** como los **datos** se descomponen en **tareas**. No se tienen en **cuenta** aspectos de la **máquina** sobre la que se va a ejecutar (número de procesadores). La atención se centra en la **búsqueda de oportunidades** de paralelismo.
- **Comunicar**.
- **Aglutinar** (Agrupar)
- **Asignar**.

La metodología de diseño de Foster III

- **Particionar**.
- **Comunicar**: Determinar la **comunicación** necesaria para lograr la **coordinación** entre tareas. Hay que ir más allá y definir algoritmos de **comunicación** y estructuras que los **soporten**.
- **Aglutinar** (Agrupar)
- **Asignar**.

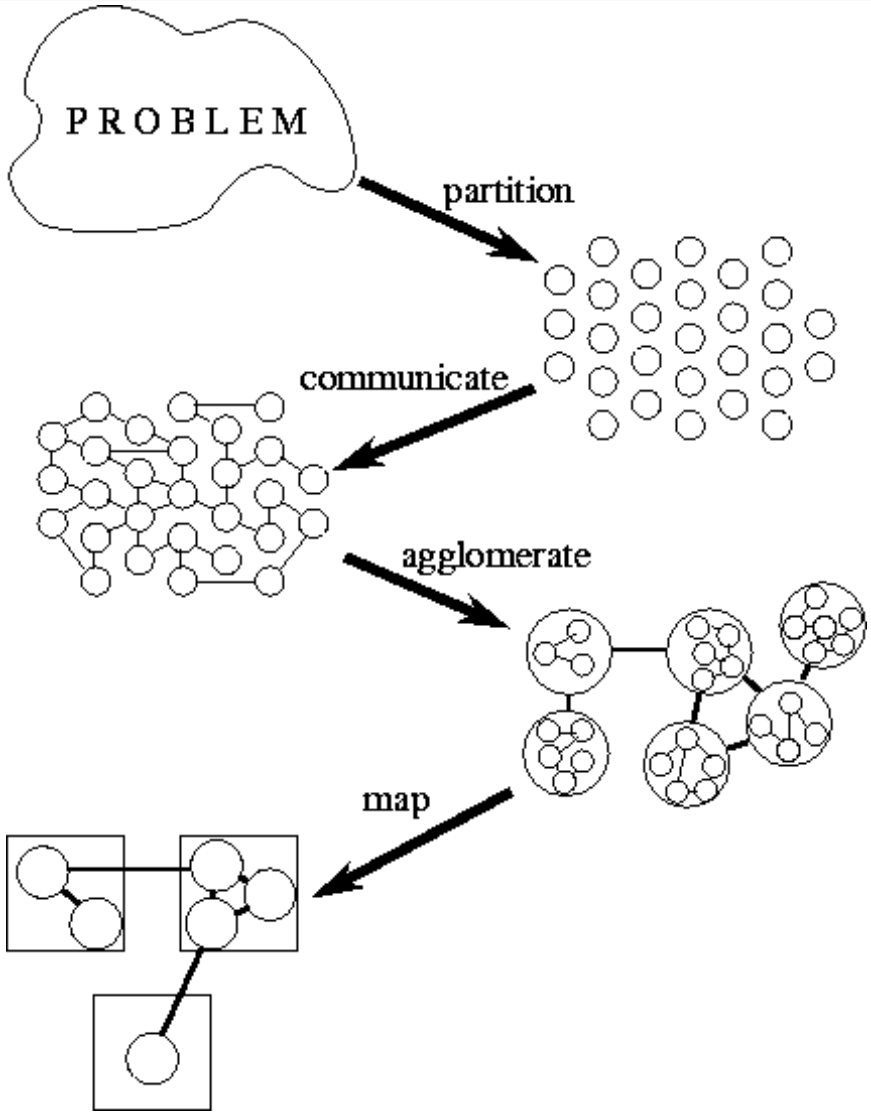
La metodología de diseño de Foster IV

- **Particionar**.
- **Comunicar**.
- **Aglutinar**: Del **estudio** de las dos tareas anteriores se **evalúa** en función de la **eficiencia** y los **costos** de implementación. Pueden llegarse a **agrupar** tareas **pequeñas** en otras más **grandes**.
- **Asignar**.

La metodología de diseño de Foster V

- **Particionar**.
- **Comunicar**.
- **Aglutinar** (Agrupar)
- **Asignar**: **tareas** a cada **procesador** con el objetivo de **maximizar** el **uso** de los mismos y **minimizar** el **coste** de la comunicación. Dos tipos de asignación: **estática** o en tiempo de ejecución (algoritmos de **balanceo** de carga).

La metodología de diseño de Foster VI



Particionado del problema

- Se puede decir que en esta etapa se quiere dar la máxima **oportunidad** al **paralelismo**.
- Se habla por tanto de **subdividir** el problema lo más finamente posible (**granularidad** fina).
- Es una **etapa activa** en la que en cada momento se puede **decidir** si ciertas tareas consideradas como paralelas se pueden **englobar** en una sola.
- Hay que tener en cuenta la **división** tanto de los **cómputos** como de los **datos**. Se habla de descomposición funcional y descomposición de dominio.

Particionado correcto del problema

- Número de tareas **superior** al número de **procesadores** disponibles.
- **Evitar** cómputos y almacenamientos **redundantes**. Problemas de **extensión** cuando los problemas sean más **grandes**.
- Si las tareas tienen un **tamaño equivalente** facilitan el **balanceo** de carga entre **procesadores**.
- **Proporcionalidad** entre el número de **tareas** y el **tamaño** del problema (escalabilidad). Se podrán resolver problemas más grandes cuando se tengan más procesadores.

Fases en la Comunicación

- **Primera**: Definir **canales** entre **tareas** que requieren **datos** y tareas que **poseen** dichos datos.
- **Segunda**: Concretar la información a **recibir** y **enviar** en dichos canales.

Memoria Compartida vs. Memoria Distribuida

- Mem. **Distribuida**: interacción **limitada** a paso de **mensajes** entre tareas.
- Mem. **Compartida**: utilización de **mecanismos** de **sincronización** para evitar **conflictos** en el acceso a memoria.

Comunicaciones correctas

- Las **operaciones** de comunicación deben estar **balanceadas** entre tareas. Si esto no es así hay más riesgo de **cuellos de botella**.
- La **comunicación** entre tareas debe ser **pequeña**, es decir, debe **afectar** a un número de tareas **vecinas** pequeño.
- Las **comunicaciones** entre tareas deben ser **concurrentes**.

Agrupación I

- Hasta ahora las **decisiones** se han tomado sin tener en cuenta la **máquina** sobre la que se ejecutará el **algoritmo**.
- Es en esta etapa cuando se **baja** de **nivel** y se estudia la **utilidad** de la **agrupación** de tareas teniendo en cuenta la **plataforma** seleccionada.
- Básicamente cuando la **partición** es muy **alta**, las necesidades de **comunicación** pueden ser **significativas** de tal forma que el algoritmo sea ineficiente.

Agrupación II

- El **agrupamiento** disminuye las comunicaciones **incrementando** la **localidad**.
- También hay que **valorar** la **transmisión** de más **información** por mensaje **reduciendo** el número de los mismos. También la **replicación** de cálculos y **datos** para reducir la **comunicación**.
- Cuando se **agrupa** se pueden **reutilizar** procesos **secuenciales**.

Agrupación III

- Otro factor a tener en cuenta es el **coste** de **creación** de procesos y cambios de **contexto** al asignar varias tareas a un mismo **procesador**.
- En caso de tener diferentes tareas ejecutándose en entornos con **memoria distribuida** antes de plantear necesidades de comunicación se debe intentar obtener una "**granularidad gruesa**".
- Si por contra se usa **memoria compartida**, se considera aceptable una "**granularidad media**". Esto es debido a que el **coste** de la **comunicación** es **menor** (siempre y cuando el número de tareas se mantenga por debajo de cierto umbral).

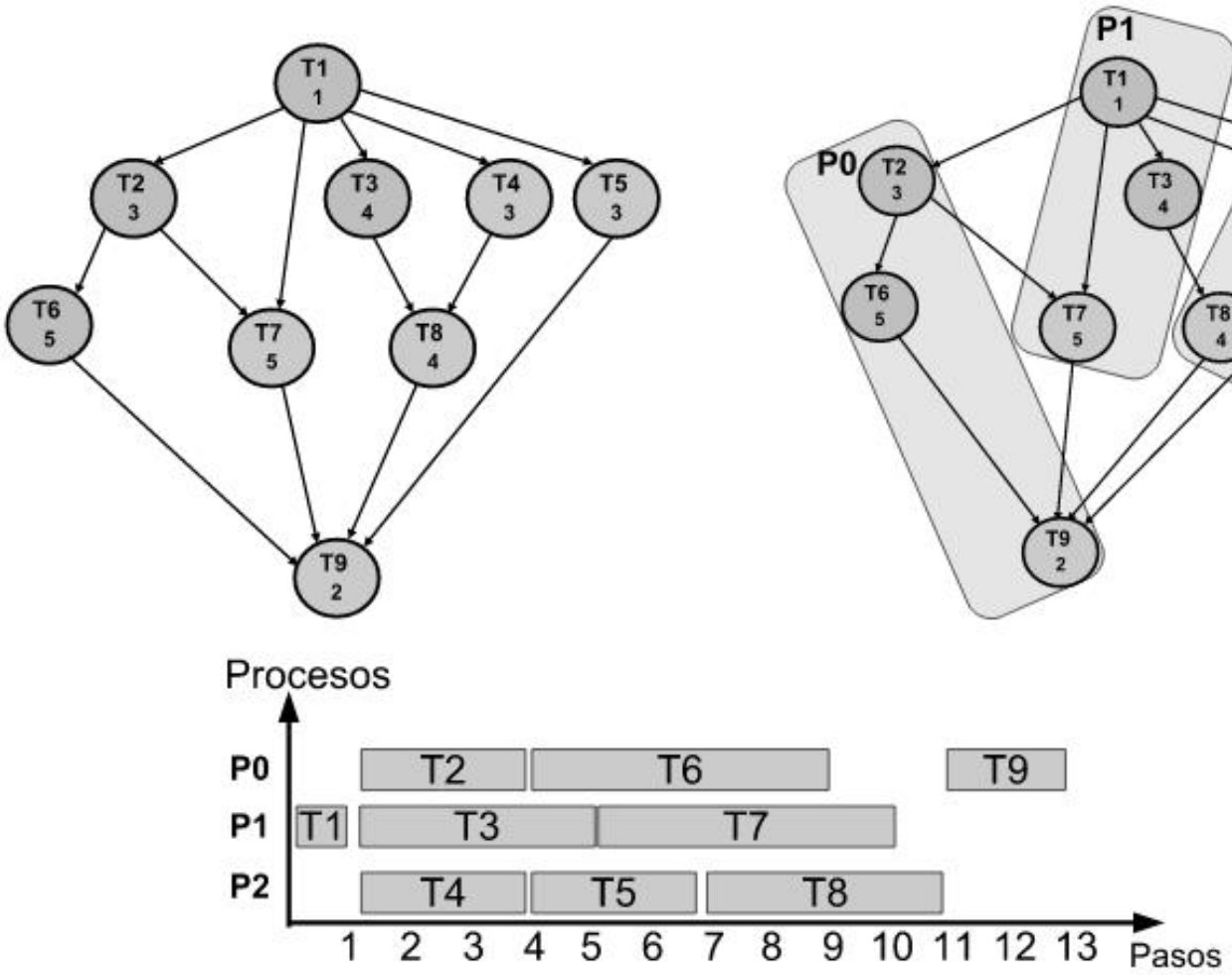
Calidad de la agrupación I

- **Comprobar** de facto que la **agrupación reduce** el **coste** asociado a la comunicación.
- Si se han **replicado** cálculos y datos se debe medir de manera fiable que los **beneficios** son **superiores** que los costos.
- Verificar que exista **similitud** a nivel de costo y comunicación de las **tareas** resultantes.

Calidad de la agrupación II

- **Verificar** que el número de tareas es **extensible** en función del **tamaño** del problema.
- Puede que el **agrupamiento** haya **reducido** las oportunidades de ejecución **concurrente** y en ese caso habría que verificar que hay un **nivel mínimo** de concurrencia y sino es así considerar diseños **alternativos**.

Objetivos


T1 1
T2 3
T3 4
T4 3
T5 3
T6 5
T7 5
T8 4
T9 2
P0
P1
P2
Procesos
P0
P1
P2
T2
T6
T9
T1
T3
T7
T4
T5
T8
1 2 3 4 5 6 7 8 9 10 11 12 13
Pasos

Tipos

Existen dos tipos o maneras de realizar la asignación:

- **Estática**: la tarea se asigna desde su **inicio** a su **fin** al mismo **procesador**.
- **Dinámica**: la tarea puede **migrar** en tiempo de **ejecución**. Obviamente con un **costo adicional**.

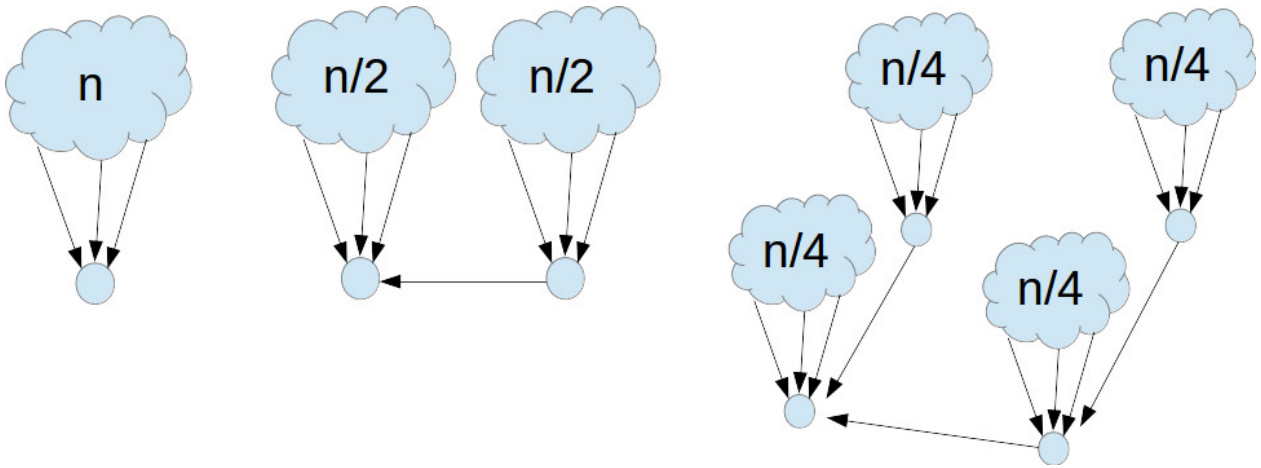
Asignación correcta

- ¿Se han **considerado** la asignación de un **procesador** por tarea y de múltiples **tareas** por procesador?
- ¿Se han **evaluado** la asignación **estática** y **dinámica**?
- Si se ha **adoptado** una asignación **dinámica**, ¿Su gestión no es un **cuello de botella**?

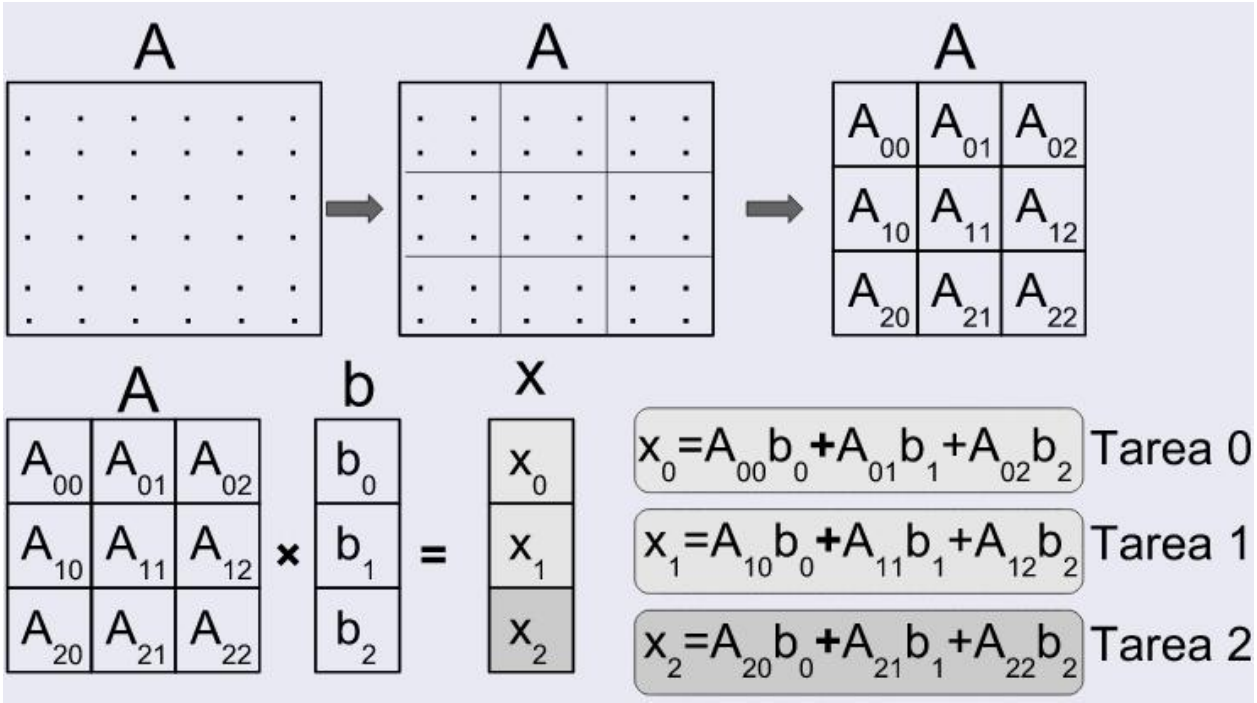
Índice

Reducción

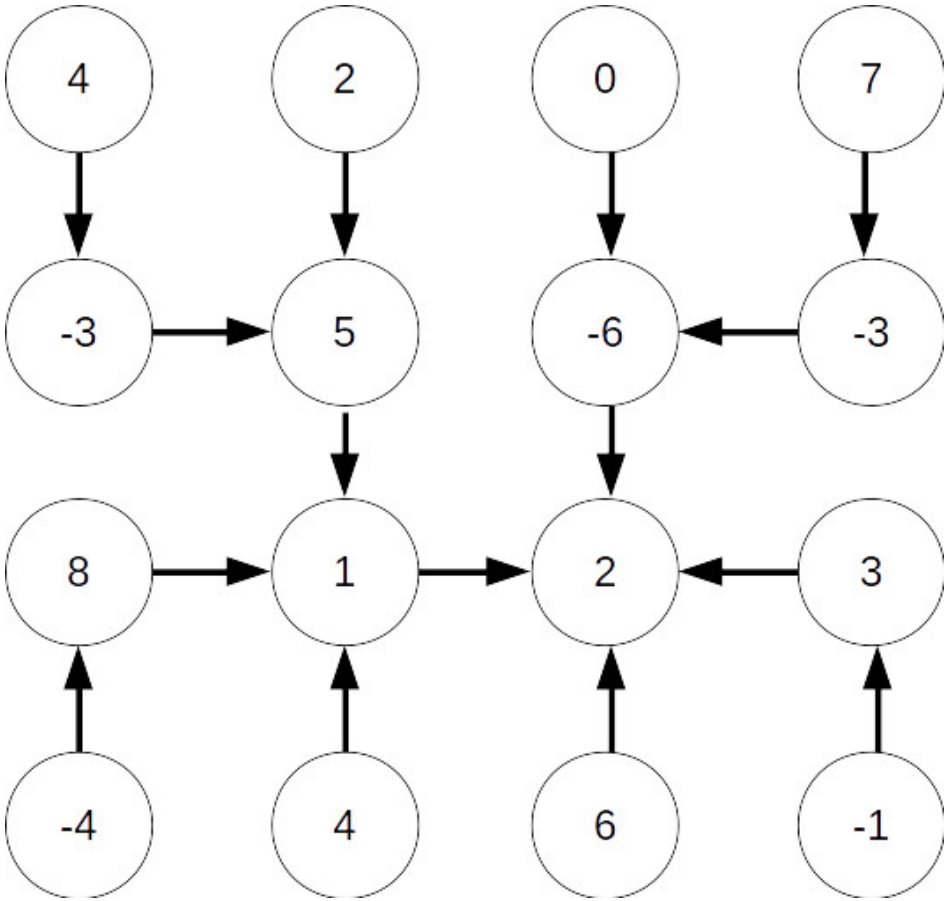
- Multiplicación n números.



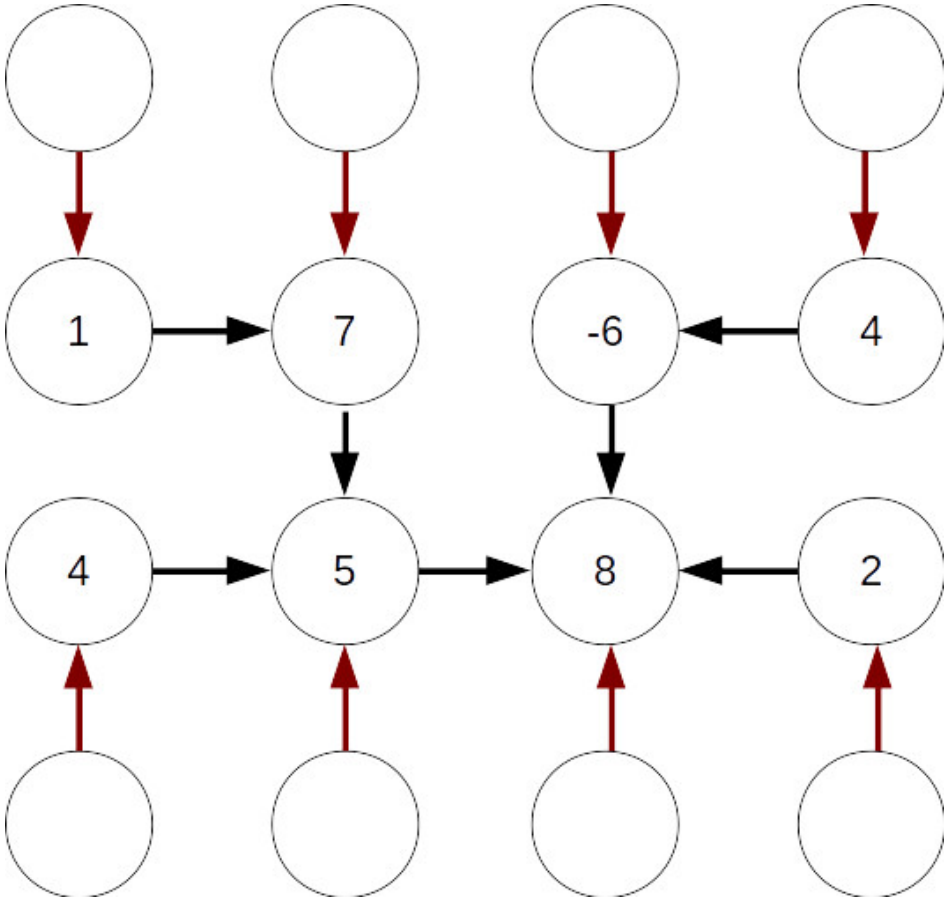
Multiplicación matriz-vector



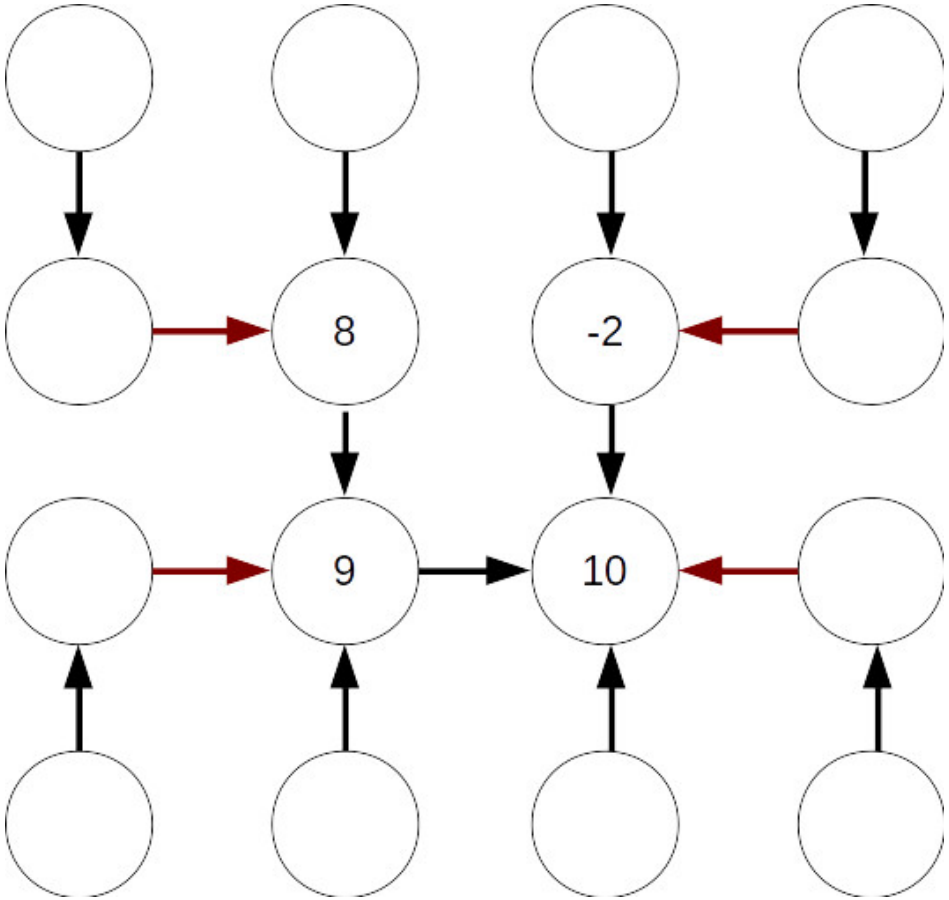
Suma de números I



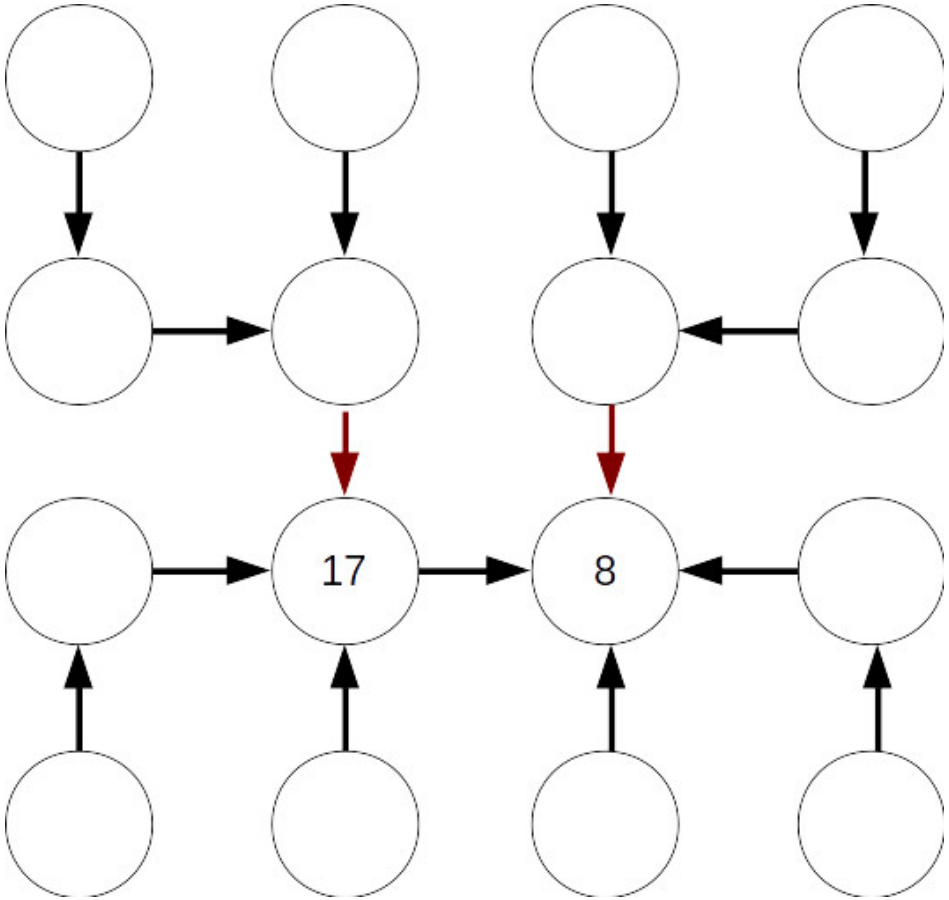
Suma de números II



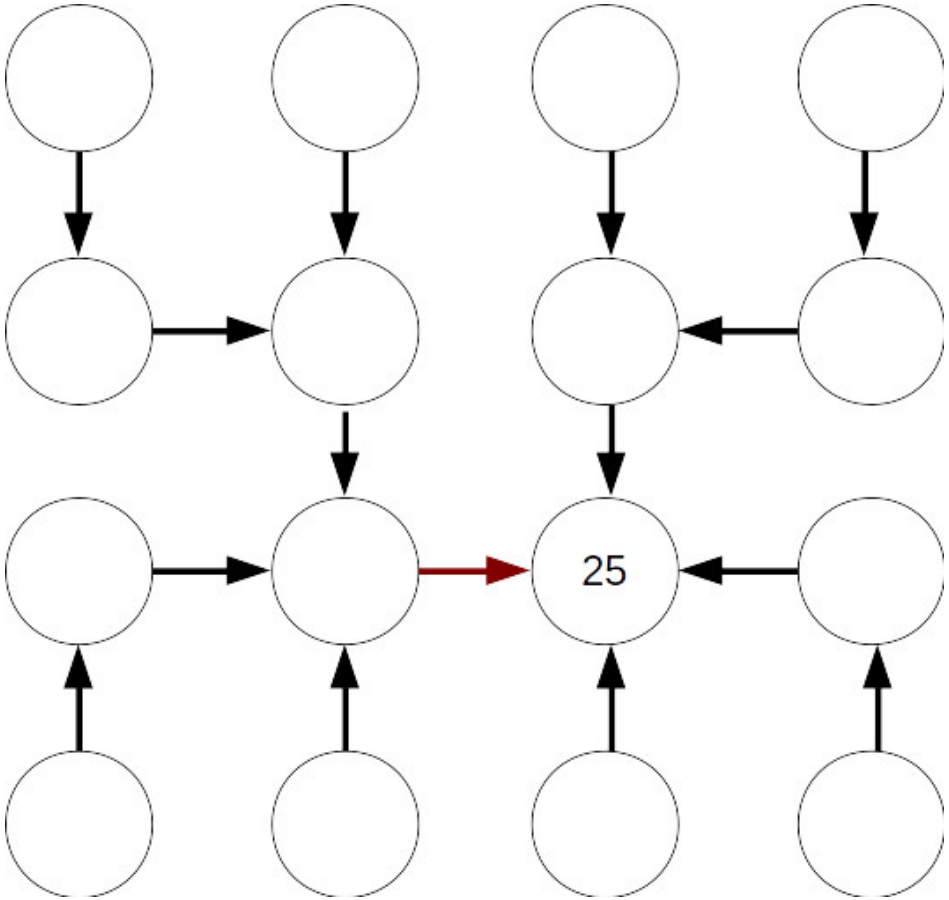
Suma de números III



Suma de números IV

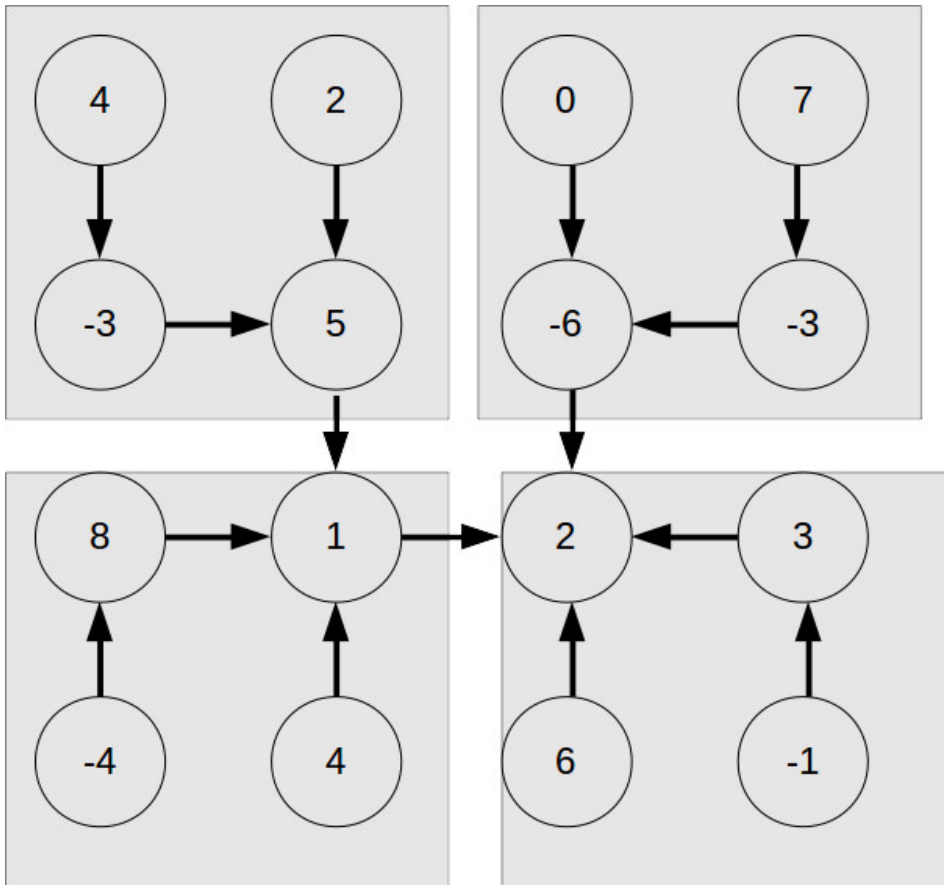


Suma de números V



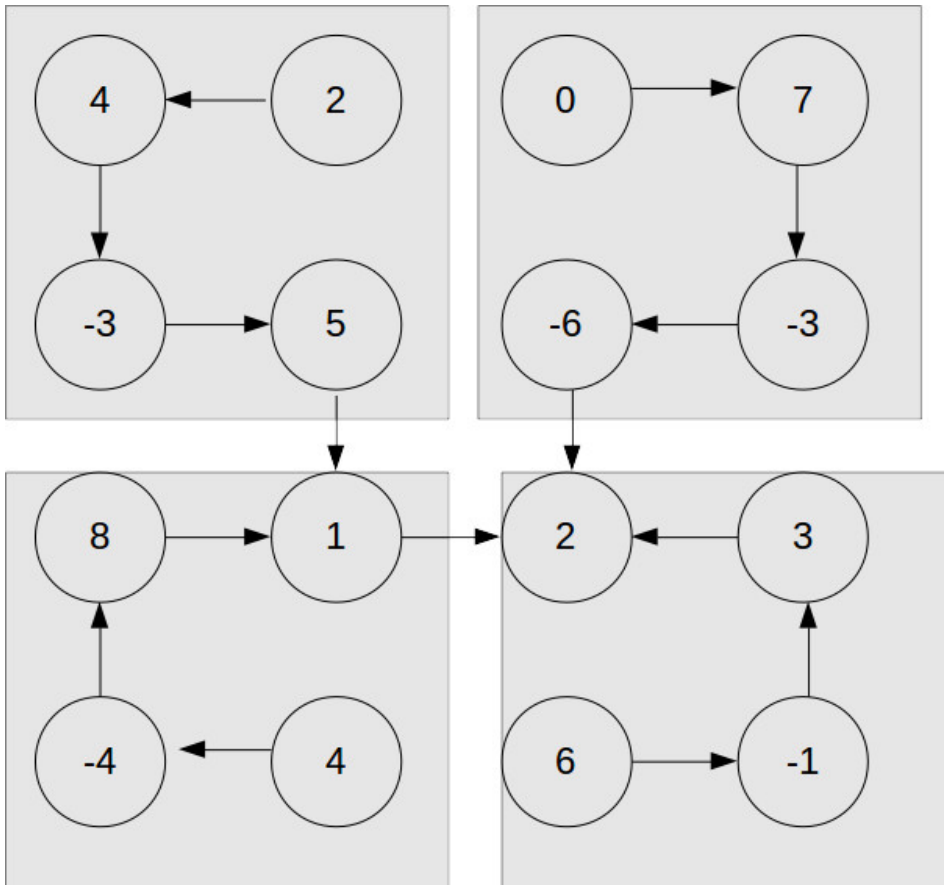
Suma de números VI

Agrupación:



Suma de números VII

Asignación:



Índice

Objetivos

- Aprender **aspectos** de la construcción de **algoritmos genéticos**.
- Ver una **aplicación práctica** de MPI en problemas de **optimización**.
- Estudiar diferentes métodos de **paralelización** de AGs.

Algoritmos Genéticos I

- Similar a la búsqueda **local** pero generando **sucesores** a partir de pares de **estados**.
- Los **estados** son como **individuos** de la **población**.
- La **población** inicial se genera de manera **aleatoria**.
- Cada **individuo** se representa como una **cadena** (**cromosoma**).

Algoritmos Genéticos II

- Cada individuo obtiene una **valoración** a través de una función objetivo (**fitness function**).
- La **probabilidad** de ser escogido para **reproducirse** es directamente proporcional a ese valor **fitness**.
- Dos padres generan **descendencia** mediante **cruce** y posteriormente hay una pequeña **probabilidad** de que ocurra una **mutación** en los nuevos individuos: **cambian** de valor bits de la cadena.

Cruce I

- Dado dos individuos:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

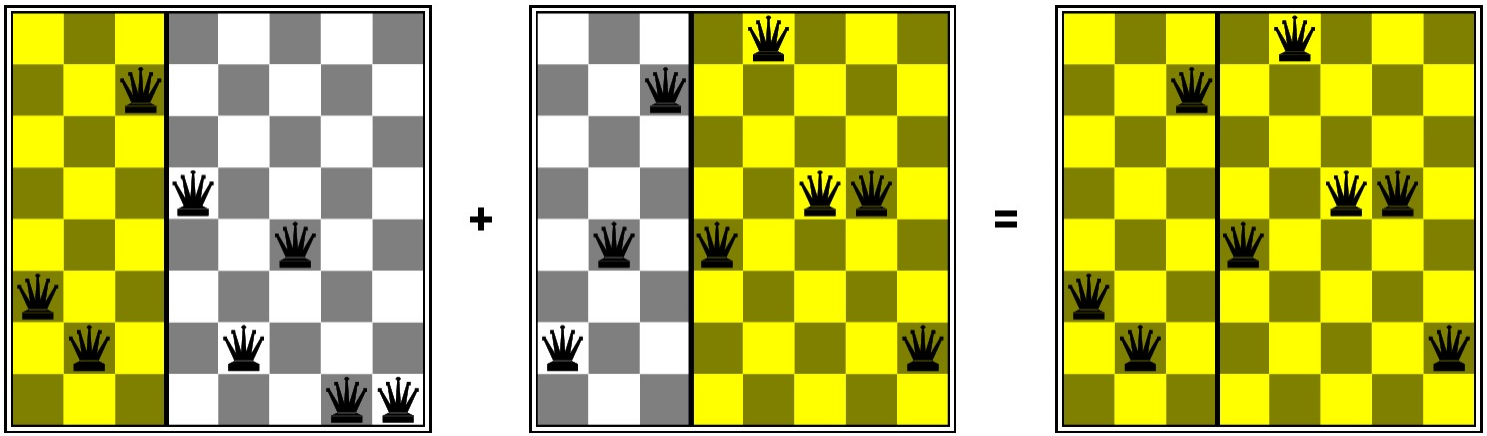
- Selecciona i aleatoriamente.
- Genera dos descendientes:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

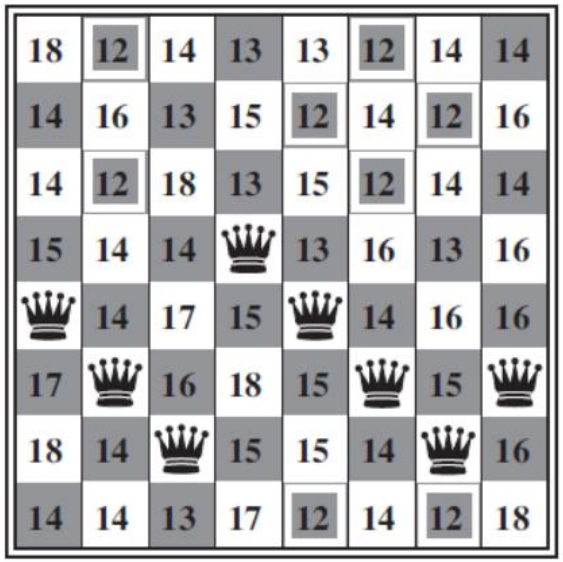
Cruce II

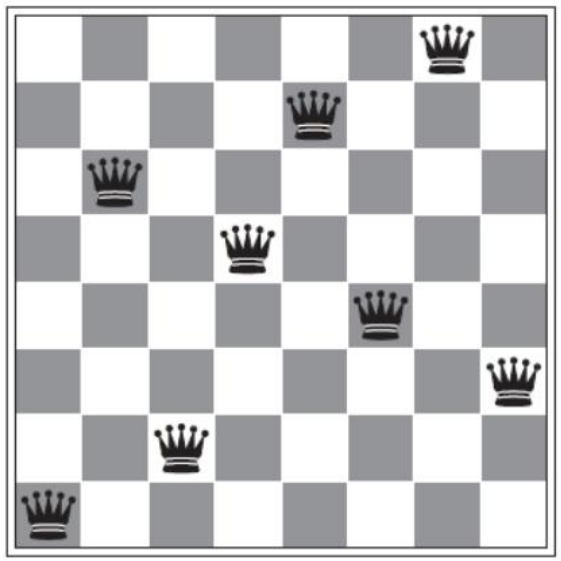
- El **cruce** es realmetne válido *si las subcadenas son elementos con significado propio*
- AGs requieren de estados **codificados** como **cadenas**.



Problema de las n-reinas I

Estados en el problema de las 8-reinas: se asume que cada **reina** está en una única **columna** y se representa un **estado** mediante una **lista** con **dígitos** del 1 al 8.





Problema de las n-reinas II

- Hay 28 **pares** de posibles ataques y queremos representar este **problema** como de **maximización**.
- Por tanto, se propone un **función fitness** de **28-h**, siendo h el **número de ataques** en un estado concreto.
- Las posibles **soluciones** tienen un fitness de **28**.
- Por ejemplo, el **fitness** del **estado** representado en la transparencia anterior es **27** (las reinas de las **columnas** 4 y 7 se **atacan** entre ellas)

Funcionamiento del Algoritmo I

function GENETIC-ALGORITHM(*population*,FITNESS-FN) **returns** an individual

inputs: *population*, a set of individuals
FITNESS-FN, a function that measures the fitness of an individual

repeat
new-population ← empty set
for *i*=1 **to** SIZE(*population*) **do**
x ← RANDOM-SELECTION(*population*,FITNESS-FN)
y ← RANDOM-SELECTION(*population*,FITNESS-FN)
child ← REPRODUCE(*x*, *y*)
if (small random probability) **then** *child* ← MUTATE(*child*)
add *child* to *new-population*
until some individual is fit enough or enough time has elapsed
return the best individual in *population*, according to FITNESS-FN

Funcionamiento del Algoritmo II

```
function REPRODUCE(x, y) returns an individual
    inputs: x, y, parent individuals
    n ← LENGTH(x); c ← random number from 1 to n
    return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c+1, n))
```

Funcionamiento del Algoritmo III

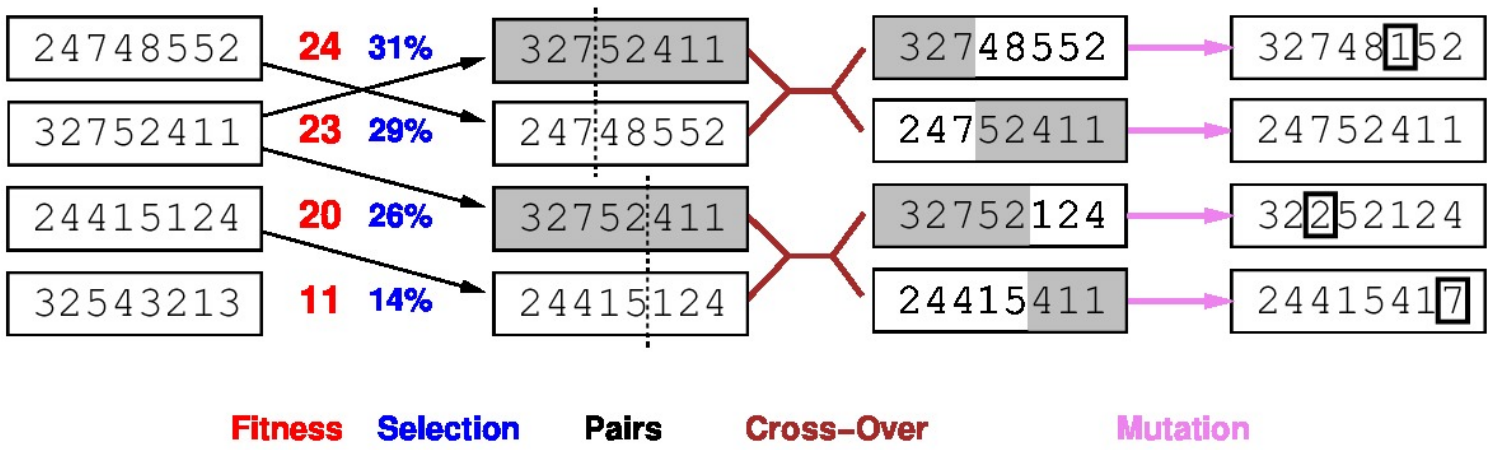


Figura 1: Ejemplo 8-reinas

Funcionamiento del Algoritmo IV

- **Combinación** de **pares** de individuos para crear la **descendencia**.
- Por cada **generación**:
  - **De manera aleatoria** elegir parejas de individuos donde los **fittest** individuos tienen una mayor probabilidad de ser escogidos.
  - Para cada **par**, ejecutar el cruce: obtener dos **descendientes** con partes diferentes de sus padres.
  - Mutar **algunos** valores.
- **Parar** cuando se encuentra una **solución**.

Funcionamiento del Algoritmo V

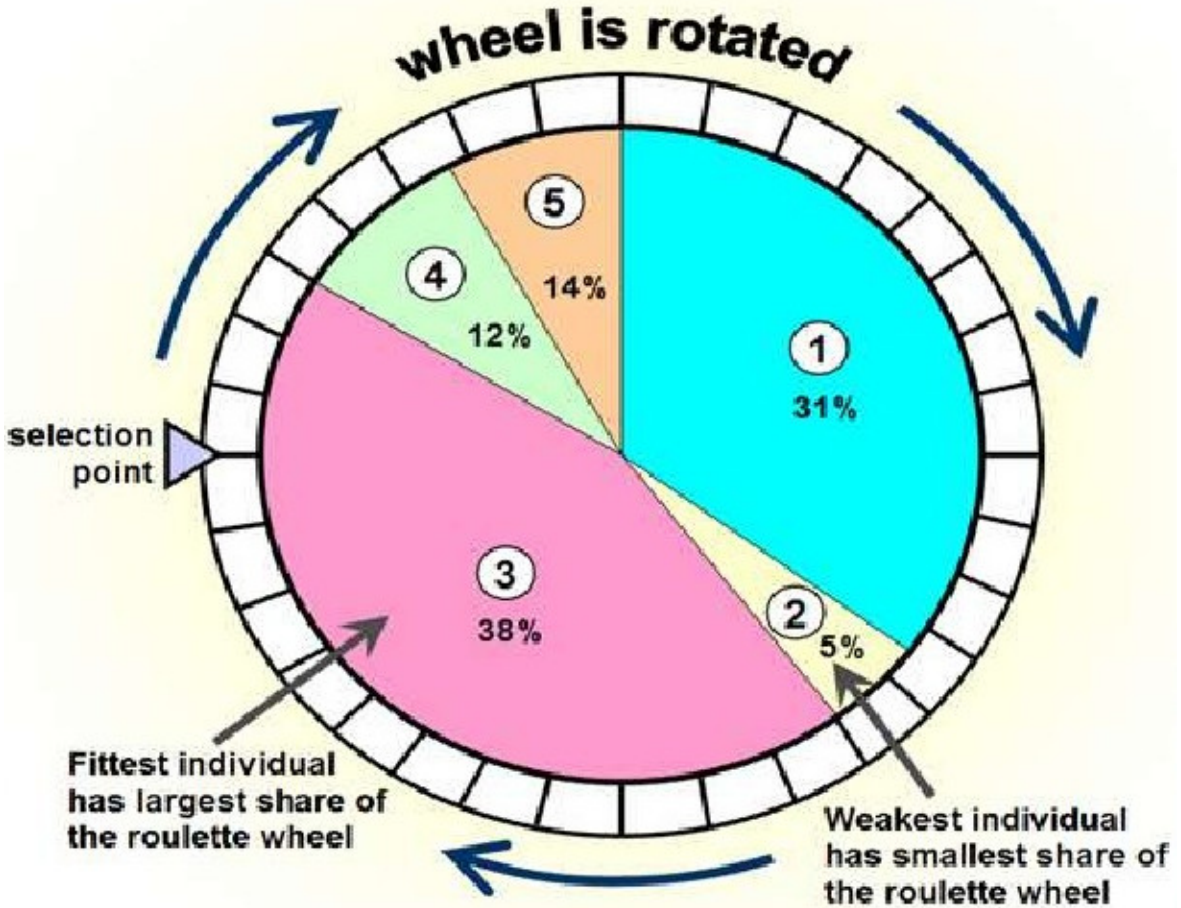


Figura 2: Selección mediante el método de la ruleta

Tipos de procesos en MPI

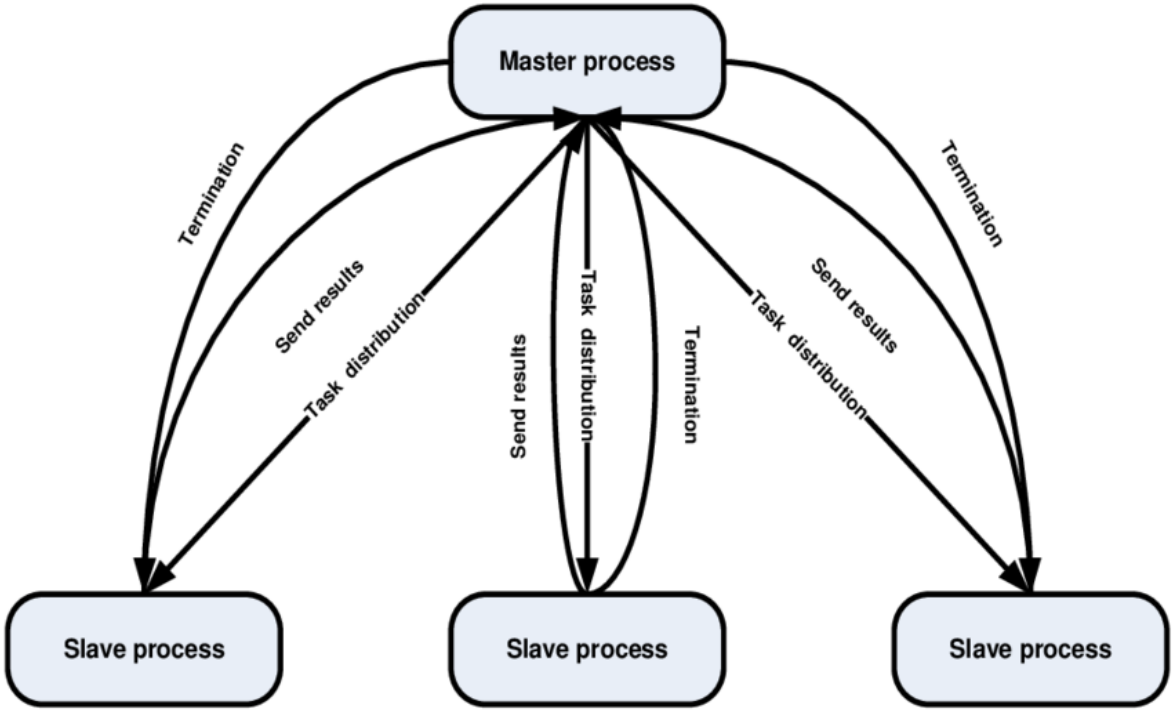


Figura 3: Flujo de información Master/Slave

El proceso Master (manager) alberga toda la población I

- **Distribuir** los genes con la solicitud de **evaluación** de la función de **aptitud**.
- Los **slaves** (y el **manager**) hacen **evaluaciones** de las funciones de aptitud.
- **Guardar** el resultado de la **evaluación** en un **array** de soluciones.
- Realizar un **ordenamiento** paralelo con los **resultados** devueltos al manager.
- **Descartar** genes.
- **Manager** ejecuta el **cruce** con la mitad **superior**.

El proceso Master (manager) alberga toda la población II

- Buen **equilibrio** de carga si la función de **fitness** es costosa de **calcular**.
- Podría **utilizar** varios **procesadores** para una sola **evaluación** de la función de aptitud.
- Fácil de **implementar**
- **Grandes** cantidades de **comunicación**

Enfoque distribuido

- Cada procesador **evalúa** su sub-población.
- Cada procesador **ejecuta** su propio proceso de **reproducción**.
- Esto da juego a gran **cantidad** de posibles **variaciones**.

Enfoque distribuido. Variante 1 I

- Permite que los **procesadores** trabajen de forma **independiente** durante X **generaciones**.
- Implementa una especie de **paralelismo global** de todos los valores de fitness.
- La **mitad superior** de todos los genes de cada **procesador** se redistribuye.

Enfoque distribuido. Variante 1 II

- Permitir **evoluciones** separadas.
- Permite **simular** el efecto de diferentes tasas de **mutación**.
- Es una **simulación** del algoritmo **secuencial**.
- Requiere **comunicación** personalizada de **todos con todos**, lo cual es difícil de configurar.
- Al enviar a la mitad de la población requiere **menos comunicación**.
- Cada **procesador** recibe el **mismo** número de genes
- Cada **procesador** envía un número **diferente** a varios procesadores
- Buen **equilibrio** de carga.

Enfoque distribuido. Variante 2

- Permite que los **procesadores** trabajen de forma **independiente** durante X generaciones.
- **Subpoblación** de **intercambio** izquierda-derecha y/o arriba-abajo.
- Permite la **migración** de soluciones a través de la **topología**.
- **Fácil** de implementar.
- Permite **observar** el efecto de diferentes **tasas** de **mutación**.

Enfoque distribuido. Variante 3 I

- Asigna un **factor** de **agresividad** a cada procesador.
- Permitir que los **procesadores** trabajen de forma **independiente** durante X generaciones
- Los **procesadores** más agresivos **obligan** a una parte de su **población** a entrar en otros procesadores.

Enfoque distribuido. Variante 3 II

- Requiere **comunicaciones** personalizadas de **todos** con **todos**.
- Se necesita **configurar** el **número** de genes que van a **transmitirse**.
- El **coste** de la **comunicación** depende del número de genes.

Enfoque distribuido. Variante 4

- Variación 1 + Variación 2 + Variación 3
- El **porcentaje** de **influencia** de cada variación ha de ser **controlado** en el inicio.
- La **dificultad** reside en **controlar** el **efecto** sobre el resultado final de las **combinaciones** de estas variaciones.

Ordenamiento paralelo

- Se utiliza para **ordenar** los **genes** según su valor de la función de **aptitud**.
- Los **resultados** acabarán almacenados en el **Manager**.
- Cada **procesador** ordena su subconjunto de **genes**.
- **Posteriormente** se utiliza una rutina de **recolección** paralela para **fusionar** las **listas** ordenadas.

Bibliografía

- **An Introduction to Parallel Programming**. Peter Pacheco. Ed. Morgan Kaufmann. February 17, 2011
- **Metodología de diseño de algoritmos paralelos**. José Miguel Mantas Ruiz.
- **Parallel Genetic Algorithms**, Timothy H. Kaiser,

Tema 6.1. Algoritmos paralelos Diseño.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Tema 6.2. Algoritmos paralelos Introducción a MPI.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Índice

Índice

Objetivos

- Estudiar un **estándar** de programación **paralela** como es **MPI**.
- Comprender como procesos **independientes** interactuan mediante el **intercambio de información**.
- Utilizar el **paso de mensajes** como herramienta que permite un intercambio **cooperativo** de información.
- Estudiar un estándar en el que la información debe ser **enviada** y **recibida** de manera **explícita**.

¿Qué es MPI? I

Es una **librería** para el **paso de mensajes** en la que:

- Se **define** un **modelo** para el paso de mensajes.
- **No** se define una **especificación** para un **compilador** concreto.
- **Tampoco** es un producto para una **tecnología específica**.

¿Qué es MPI? II

Dentro de sus características encontramos:

- Para arquitecturas **paralelas**, **clusters** y redes **heterogéneas**.
- Diseñado para permitir el **desarrollo** de **librerías** software paralelas.
- Diseñado para proporcionar **acceso** a hardware **paralelo** avanzado para usuarios finalistas, **desarrolladores** de librerías o de **herramientas**.

Alternativas a MPI

- **OpenCL**: C para CPUs y GPUs.
- **OpenAcc** (open) y CUDA (NVDIA) específico para GPUs.
- **OpenMP**: CPUs y conversión de código C, C++ o Fortran para uso de threads.
- **pthreads, C11, C++11**: específico para CPU.

Terminología asociada

- **Distribuido**: Los **recursos** se encuentran **repartidos** en varias unidades de cómputo, nodos, que pueden estar en el **mismo** o en computadores **distintos**.
- **POD** (plain-old data): Tipos de **datos** como en C pero **sin** uso de **punteros**, objetos, herencia, ...
- **Mensaje**: Un **POD** o una matriz de PODs.
- **SIMD**: Single-instrucion, Multiple Data (**streams**).
- **UE**: Unidad de Ejecución (por ejemplo, un **proceso MPI**).

Modelo de programación paralela en MPI

- La memoria es **distribuida** y **no compartida**.
- Las **Unidades de Ejecución** son **procesos**, aunque existen versiones que permite ejecutar cada proceso MPI como hebras, nosotros **siempre** vamos a trabajar con **procesos**.
- El **envío** de mensajes tiene un **coste**: Si se necesitan mandar mensajes de **10MB** a **10 nodos**, el proceso que envía necesita transmitir en total **100MB** de información.

Objetivos importantes I

Una vez se tengan una serie de **conocimientos básicos** en programación paralela y MPI, se **deberían** siempre **adoptar** a priori los siguientes **objetivos**:

- Todas las **estructuras** de **datos** a crear serán **PODs** o fácilmente convertibles a PODs.
- Las **estructuras** "contenedores" de estructuras de datos **preferidas** por el **programador** deberían ser: std::string, std::array o std::vector.

Objetivos importantes II

- El **diseño** previo de una aplicación es mucho más **relevante** que en **programación secuencial**. Antes de escribir **código** hay que tener muy claro **qué** se quiere conseguir y **cómo** se va a conseguir.

Funcionalidades de MPI

- Aunque el **número** de funciones es **grande**, unas **125**, sólo con **seis** se pueden **escribir** gran cantidad de programas paralelos.
- Por tanto **no** puede considerarse que sea **complejo**.
- Más bien se tienen las **dos opciones** de desarrollo **básico** con pocas funciones y gran cantidad de recursos para desarrollos **complejos**.

Índice

Instalación I

- Necesidad de tener instalado un **compilador de C/C++**, aunque en ciertos sistemas el gestor de paquetes resuelve esta dependencia.

Los **paquetes necesarios** (distribuciones basadas en Debian)

- **openmpi-bin**: Programa de ejecución de códigos paralelos (mpirun).
- **openssh-client, openssh-server**: Programa de comunicación (rutinas de control y presentación) entre procesos.

Instalación II

- **libopenmpi-dbg**: Generador de información de depuración para MPI
- **libopenmpi-dev**: Necesario para el desarrollo de programas basados en MPI (mpicc...)

Comando rápido (depende de versión de sistema operativo):

sudo apt-get install openmpi-bin openmpi-common openssh-client openssh-server libopenmpi1.6 libopenmpi1.6-dbg libopenmpi-dev

Índice

- Para programas **sencillos** se pueden usar comandos **especiales** del compilador, pero para **proyectos** de cierta envergadura lo mejor es trabajar con **makefiles**.

Compilación

```
mpicc -o first first.c
```

Compilación

```
mpirun -np 2 first
```

Índice

Mi primer programa I

Llamadas MPI

MPI_INIT: requerido para comenzar a usar MPI.
MPI_FINALIZE: terminar de usar MPI y "apagarlo".

Mi primer programa II

Listado 1: First.c

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main(argc,argv)
5  int argc;
6  char **argv;
7
8  {
9
10 MPI_Init(&argc,&argv);
11 printf("Hello world \n"); //Rutina local para cada proceso
12 MPI_Finalize();
13 return 0;
14 }
```

¿Cuántos procesos hay y quién soy yo? I

Las **primeras preguntas** que un programa paralelo podría hacerse son:

- **¿Cuántos procesos** hay?. Se responde con **MPI_Comm_size**.
- ¿**Quién soy** yo?. Se responde con **MPI_Comm_rank**. Hay que aclarar que **rank** es un número **entre cero y el número de procesos paralelos menos uno**.

¿Cuántos procesos hay y quién soy yo? II

Listado 2: RankSize.c

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main (argc, argv)
5  int argc;
6  char **argv;
7  {
8
9  int rank,size;
10 MPI_Init(&argc,&argv);
11 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
12 MPI_Comm_size(MPI_COMM_WORLD,&size);
13 printf("Hello World! I'm %d of %d\n",rank,size);
14 MPI_Finalize();
15 return 0;
16 }
```

¿Cuántos procesos hay y quién soy yo? III

Listado 3: RankSizeIf.c

```
1  #include  <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char **argv) {
5     int rank;
6     int size;
7
8     MPI_Init(&argc, &argv);
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    MPI_Comm_size(MPI_COMM_WORLD, &size);
11
12
13    printf("Hello World! I'm %d of %d\n",rank,size);
14    if (rank == 0) printf("That is all for now!\n");
15    MPI_Finalize();
16    exit(0);
17
18 }
```

Ejercicios

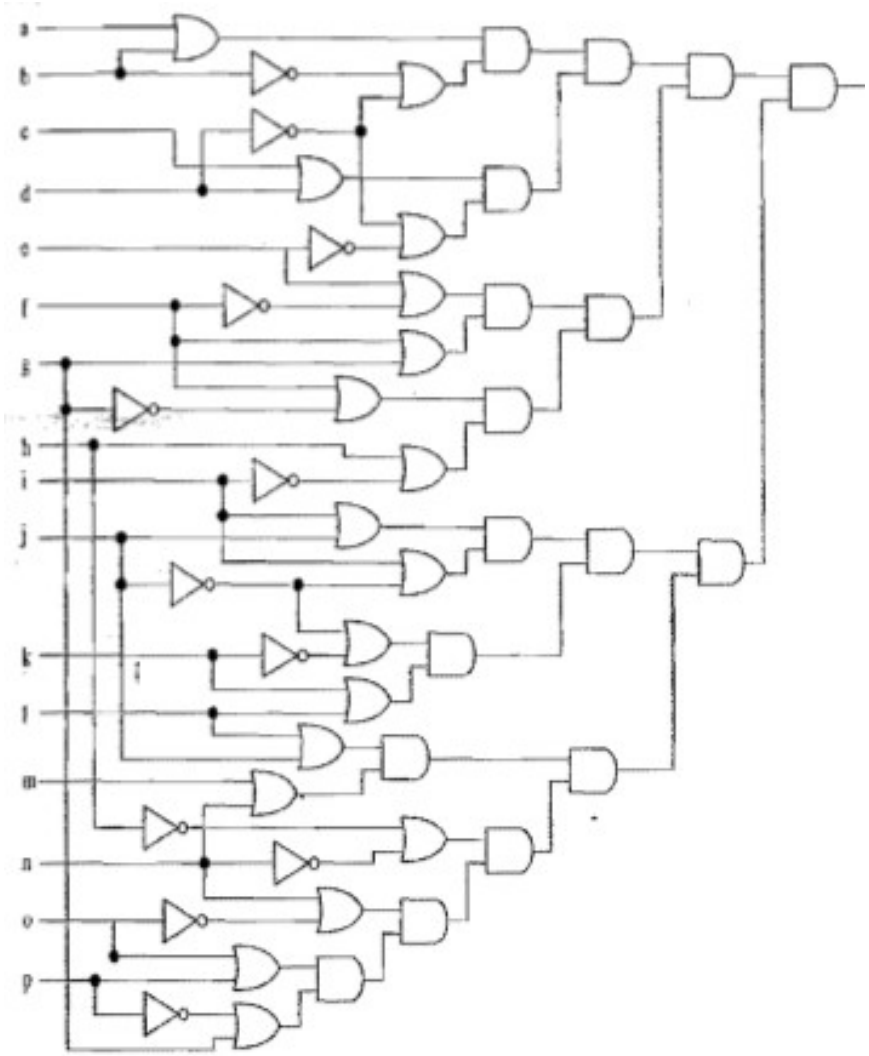
1. Implementa un programa que **finalice** si el **número** de procesos no es **superior** a un número **constante** determinado por el alumno.
2. Implementa un programa en el que los procesos **pares** e **impares** impriman su id de proceso de manera **diferente** entre ellos.

Satisfactibilidad de un circuito lógico I

CSAT

El **objetivo** de este problema es encontrar una **combinación** de valores de entrada (bits) que hagan **verdadera** la salida.

Satisfactibilidad de un circuito lógico II



Satisfactibilidad de un circuito lógico III

- Un enfoque **secuencial** de la solución **probaría** con todas las posibles **combinaciones** de entrada.
- Sería un método de **búsqueda exhaustiva**.
- Mediante **funciones lógicas** se emula el funcionamiento del **cirtuito lógico**.

Satisfactibilidad de un circuito lógico IV

Listado 4: check_circuit

```
1  void check_circuit(int id, int z) {
2    int v[16];
3    int i;
4    for (i=0;i<16;i++) v[i]=EXTRACT_BIT(z,i);
5
6    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
7      && (!v[3] || !v[4]) && (v[4] || !v[5])
8    && (v[5] || !v[6]) && (v[5] || v[6])
9    && (v[6] || ! v[15]) && (v[7] || !v[8])
10   && (!v[7] || !v[13]) && (v[8] || v[9])
11   && (v[8] || !v[9]) && (!v[9] || !v[10])
12   && (v[9] || v[11]) && (v[10] || v[11])
13   && (v[12] || v[13]) && (v[13] || !v[14])
14   && (v[14] || v[15]))
15    .......
16   }
17 }
```

Satisfactibilidad de un circuito lógico V

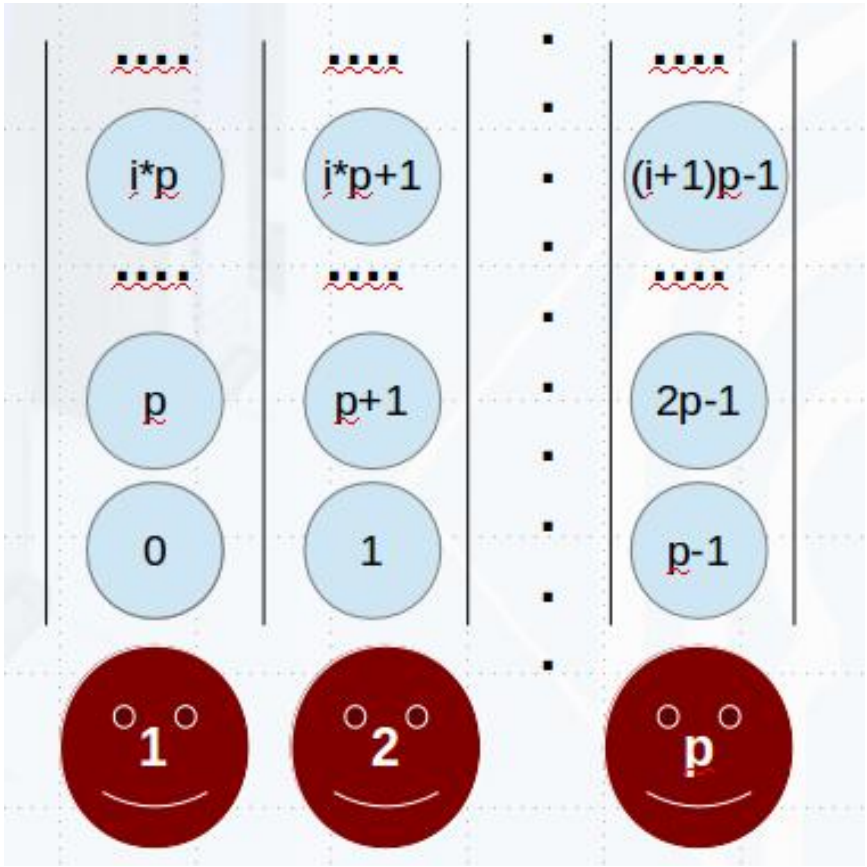
- Indica que hace el **siguiente código** siendo n e i enteros

```
1 $define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)$
```

Satisfactibilidad de un circuito lógico VI

- **Particionamiento**: Cada test para una combinación de entrada es una tarea.
- **Comunicación**: No es necesaria porque no se requiere interacción entre tareas.
- **Agregación** y **Asignación**: Distribuir tareas equitativamente entre los procesadores disponibles.

Satisfactibilidad de un circuito lógico VII



Satisfactibilidad de un circuito lógico VIII

Listado 5: main

```
1  int main(int argc, char *argv[]){
2          int i;
3          int id; /* Identificador de tarea */
4          int p;  /* Numero de tareas */
5
6          MPI_Init(&argc,&argv);
7          MPI_Comm_rank(MPI_COMM_WORLD,&id);
8          MPI_Comm_size(MPI_COMM_WORLD, &p);
9
10         for(i=id; i<65536; i+=p)
11                 check_circuit(id,i);
12
13         printf("Tarea %d terminada\n",id);
14         fflush(stdout);
15
16         MPI_Finalize();
17 }
```

Marcas de Tiempo

Listado 6: Benchmarks Code

```
1
2  double elapsed_time;
3  ...
4  MPI_Init (&argc, &argv);
5  MPI_Barrier (MPI_COMM_WORLD); //Eliminar tiempo inicio procesos
6  elapsed_time = - MPI_Wtime();
7  ...
8  MPI_Reduce (...);
9  elapsed_time += MPI_Wtime();
```

Ejercicios

1. Introduce la medición de **estadísticas** temporales para que se ejecuten sólo **asociadas** al proceso cuyo **id es cero**. ¿Se podría **medir** cuanto tarda la **suma** de todos los **procesos**?

Tema 6.2. Algoritmos paralelos Introducción a MPI.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Tema 6.3. Comunicación Básica.

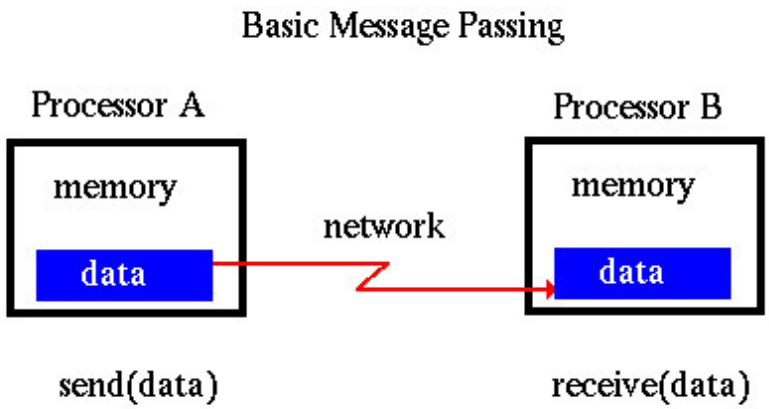
Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Índice

Índice

Introducción I



Las **preguntas** que cabe hacerse son:

- ¿A **quién** se envía la **información**?
- **¿Qué** se envía?
- **¿Cómo** el **receptor** es capaz de **determinar** las dos **preguntas** anteriores?

Introducción II

Por ejemplo, un **envío bloqueante** podría tener el siguiente **formato**:

Sintaxis envío bloqueante

```
send( dest, type, address, length )
```

donde:

- **dest**: es un entero identificando el **proceso** que recibirá el **mensaje**.
- **type**: entero no negativo que el **destinatario** puede utilizar para **seleccionar** mensajes.
- **(address, length)**: permite definir un **área contigüa en memoria** que contiene el mensaje que va a ser enviado.

Introducción III

Las **ventajas** de estas **especificaciones** es que son **simples**, fáciles de comprender pero tienen una importante **desventaja**. Su **falta** de **flexibilidad** al poder enviar y recibir sólo un **conjunto de bytes contigüo**.

- Puede que la **estructura de datos** no fuera manejable por el **hardware**.
- Requiere de un **empaquetamiento** de información **dispersa**, por ejemplo, filas de una **matriz** almacenadas como columnas, **colecciones** de cierta estructura, etc.
- Pueden existir **problemas** de **comunicación** entre **máquinas** con diferentes representaciones (incluso longitudes) para el mismo tipo de dato. ¿Diferente longitud de palabra?

Índice

MPI_Send I

Es una **operación de envío bloqueante** cuyos argumentos se definen a continuación:

MPI_Send(buf,count,type,dest,tag,comm)

buf: dirección de inicio del búfer que se envía.
count: número de elementos en el búfer.
type: cada MPI_Datatype de cada elemento del búfer.
dest: id rank del nodo destinatario del mensaje.
tag: etiqueta.
comm: comunicador.

MPI_Recv I

Es una **operación de recepción bloqueante** cuyos argumentos se definen a continuación:

*MPI_Recv(buf,count,type,**src**,tag,comm,**status**)*

buf: dirección de inicio del búfer que se recibe.
count: número de elementos en el búfer.
type: cada MPI_Datatype de cada elemento del búfer.
src: id rank del nodo del que recibir el mensaje o (MPI_ANY_SOURCE)
tag: etiqueta o (MPI_ANY_TAG)
comm: comunicador.
status: objeto estado.

MPI_Datatype

- Se usan instancias de un tipo especial llamado MPI_datatype

Equivalencia con C

MPI_C_Bool: _Bool
MPI_CHAR: char
MPI_UNSIGNED_CHAR: unsigned char
MPI_SIGNED_CHAR: signed char
MPI_INT: signed int
MPI_DOUBLE: double
MPI_LONG_DOUBLE: long double.
MPI_C_DOUBLE_COMPLEX: double _Complex

Ejemplo envío recepción básico I

Listado 1: SendRecv1.c

```
1  int main(int argc, char *argv[])
2  {
3      int ierr, procid, numprocs;
4      int root = 0;
5      ierr = MPI_Init(&argc, &argv);
6      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
7      ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
8      int i;
9
10     if (procid == root) {
11         int num;
12         for (i=1; i<numprocs; i++) {
13             printf("Waiting for response from %dn", i);
14             MPI_Recv(&num, 1, MPI_INT, i, 0,
15               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
16             printf("Received: %dn", num);      }//for } //if
17
```

Ejemplo envío recepción básico II

```
18
19
20          else {
21              printf("I am proc %d of %dn", procid, numprocs);
22              MPI_Send(&procid, 1, MPI_INT, root, 0, MPI_COMM_WORLD);
23          }
24          ierr = MPI_Finalize();
25      }
```

Ejercicios propuestos

Comunicación por pares

Desarrolla un programa en el que el proceso i envíe los resultados de la operación 3.14+i al proceso i+1.

Comunicación en anillo

Desarrolla un programa que establezca una comunicación en anillo en la que en cada nodo se incremente el valor del entero comunicado en 1.

Comunicación de datos al root para suma total

Desarrolla un programa en el que cada proceso i mande el valor -i al root y este una vez reciba todos los valores los sume y muestre el resultado.

Tema 6.3. Comunicación Básica.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Tema 6.4. Comunicación Colectiva.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Índice

Índice

- En esta parte del tema se **introduce** un concepto avanzado de **MPI** como son las **comunicaciones** colectivas.
- Este tipo de comunicaciones **involucra** a todos los **procesos**.
- Estas **operaciones** son:
  1. Broadcast
  2. Reduce
  3. Gather
  4. Scatter
- Los **beneficios** del uso de estas operaciones viene **justificada** por la **simplificación** en el desarrollo y la **ganancia** en **velocidad**.

Índice

MPI_Bcast I

Es una **operación que envía** información a todos los procesos en un comunicador:

*int MPI_Bcast(void *buffer , int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

buf: dirección de inicio del búfer que se envía.
count: número de elementos en el búfer.
type: cada MPI_Datatype de cada elemento del búfer.
root: id rank del proceso root.
comm: comunicador.

MPI_Bcast II

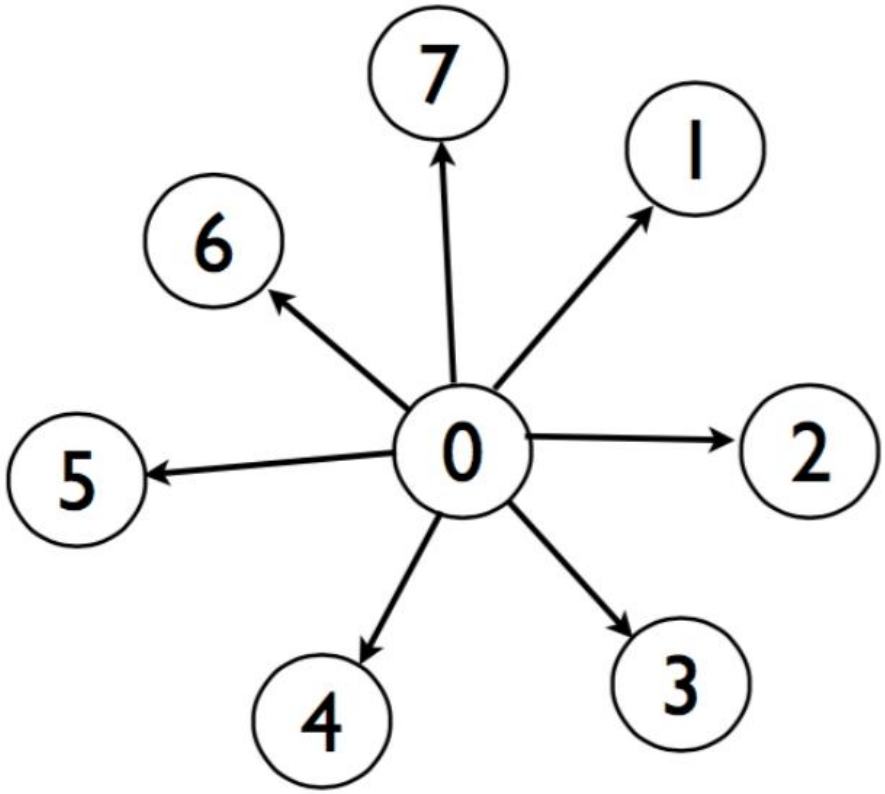
Listado 1: BCastBasico.c

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5          int rank;
6          int buf;
7          const int root=0;
8
9          MPI_Init(&argc, &argv);
10         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12         if(rank == root) {
13             buf = 777;
14         }
15
16         printf("[%d]: Before Bcast, buf is %d\n", rank, buf);
17
18         /* everyone calls bcast, data is taken from root and ends
19         MPI_Bcast(&buf, 1, MPI_INT, root, MPI_COMM_WORLD);
```

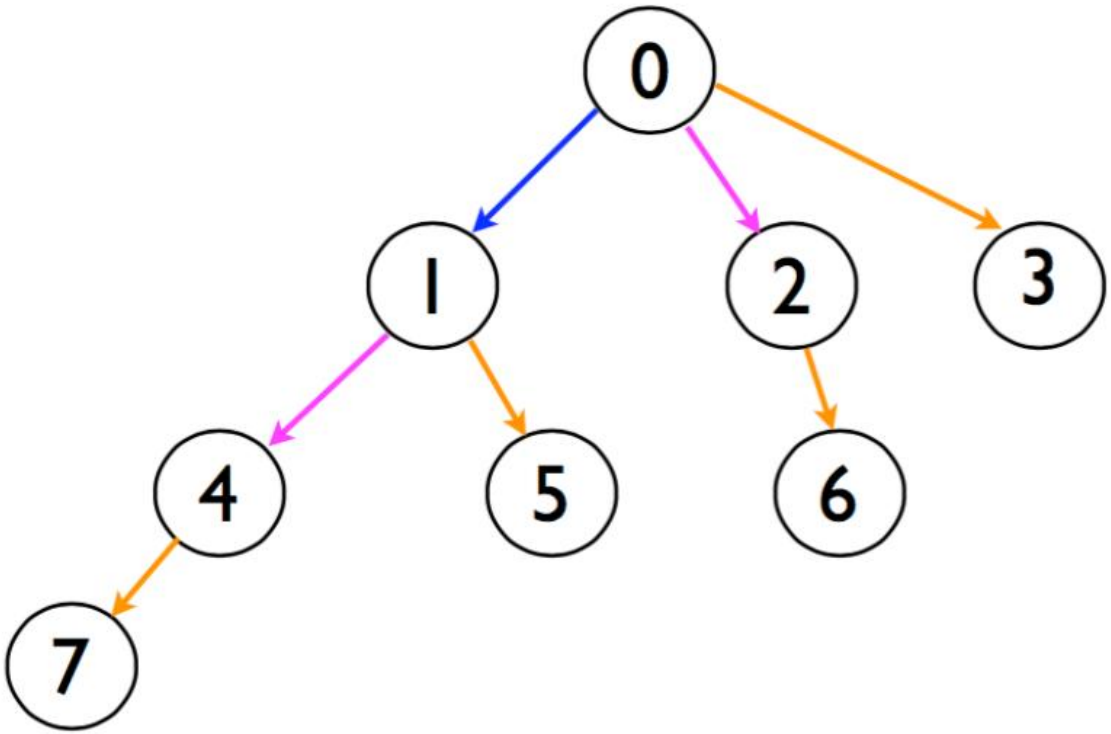
MPI_Bcast III

```
20
21          printf("[%d]: After Bcast, buf is %d\n", rank, buf);
22
23          MPI_Finalize();
24          return 0;
25  }
```

Implementación no eficiente



Implementación eficiente



Índice

MPI_Reduce I

Es una **operación que combina** información recibida de todos los procesos mediante una operación binaria:

*int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,int root, MPI_Comm comm)*

sendbuf: dirección de inicio del búfer que se envía.
recvbuf: dirección de inicio del búfer que se recibe. Sólo con significado para el root.
count: número de elementos en el búfer.
datatype: cada MPI_Datatype de cada elemento del búfer.
op: operación de combinación, predefinida o definida por el usuario.
root: id rank del proceso root.
comm: comunicador.

MPI_Reduce II

Tipos de **Operaciones**:

- MPI_SUM.
- MPI_PROD.
- MPI_MAX.
- MPI_MIN.
- MPI_BAND.

Índice

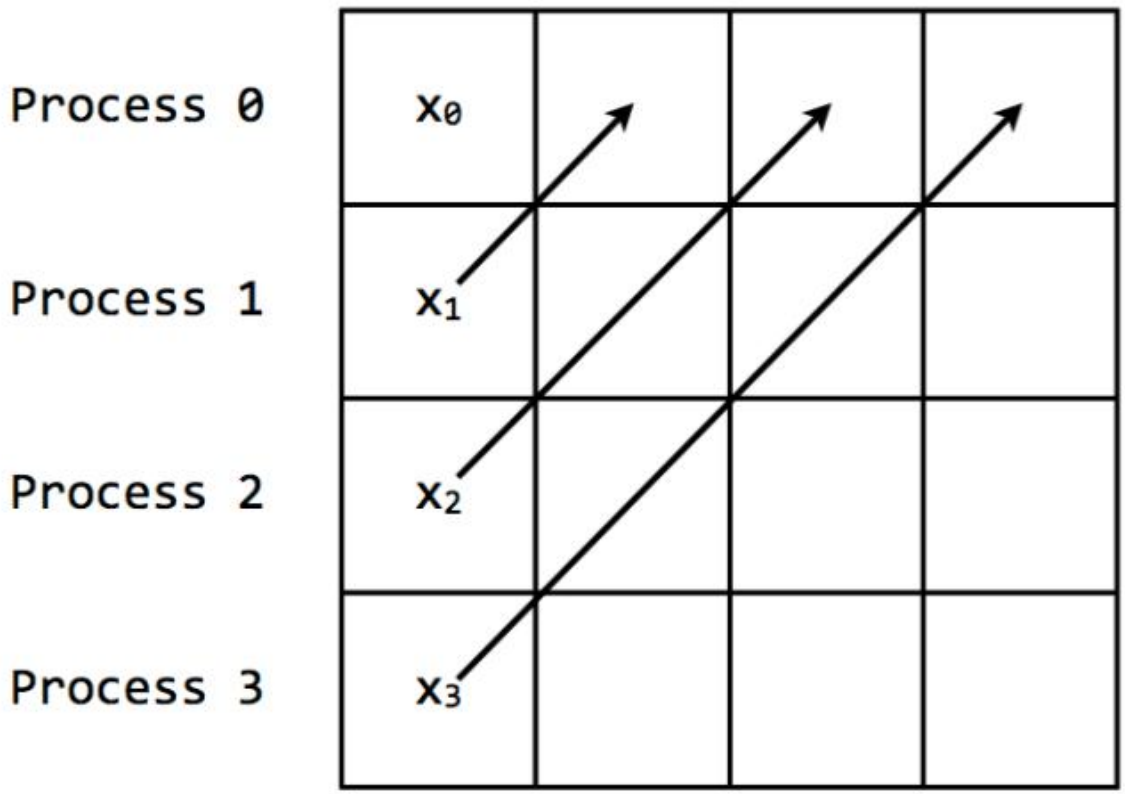
MPI_AllReduce

_MPI_AllReduce_

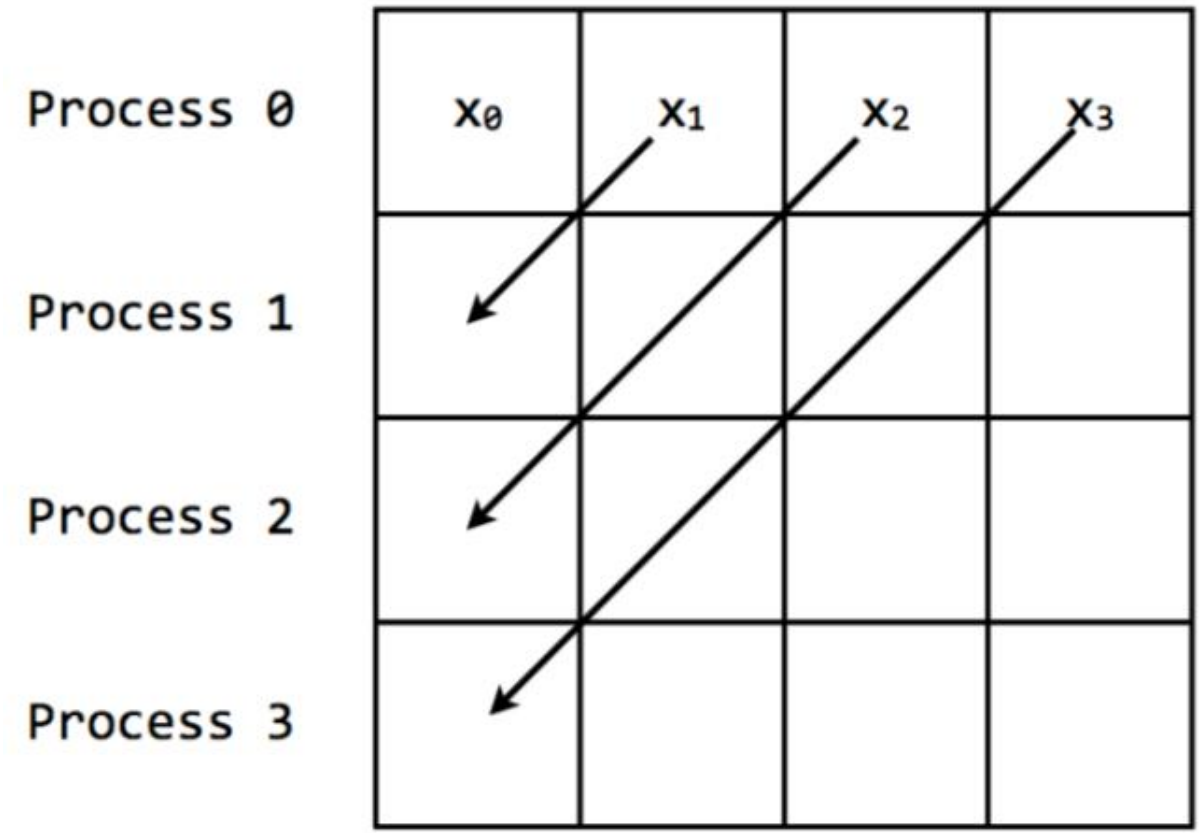
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- Similar a MPI_Reduce excepto que el **resultado** se devuelve al buffer de recepción de cada **proceso**.

Gather



Scatter



Índice

Calculando Pi I

- La idea es **implementar** un programa **sencillo** para calcular **PI**.
- El método a **implementar** evalúa la **integral** de $4/(1 + x * x)$ entre 0 y 1.
- La integral es **aproximada** mediante la suma de **n intervalos**, la aproximación a la integral en cada **intervalo** es $(1/n)*4/(1+x*x)$.
- El proceso **root** (rank 0) pregunta al usuario el número de **intervalos**, este número es **enviado** al resto de procesos.
- Cada **proceso** suma cada n-ésimo **intervalo** ($x = rank/n, rank/n+size/n, ...$).
- Finalmente, la suma **calculada** por cada proceso es **añadida** en una variable en el proceso **root**.

Calculando Pi II

Listado 2: pi.c

```
1  #include "mpi.h"
2  #include <math.h>
3
4  int main(argc,argv)
5  int argc;
6  char *argv[];
7  {
8      int done = 0, n, myid, numprocs, i;
9      double PI25DT = 3.14159265358979323846264 3;
10     double mypi, pi, h, sum, x;
11
12     MPI_Init(&argc,&argv);
13     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
14     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
15     while (!done)
16     {
17         if (myid == 0) {
18             printf("Enter the number of intervals: (0 quits) ");
19             scanf("%d",&n);
```

Calculando Pi III

```
20          }
21          MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
22          if (n == 0) break;
23
24          h   = 1.0 / (double) n;
25          sum = 0.0;
26          for (i = myid + 1; i <= n; i += numprocs) {
27              x = h * ((double)i - 0.5);
28              sum += 4.0 / (1.0 + x*x);
29          }
30          mypi = h * sum;
31
32          MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
33                     MPI_COMM_WORLD);
34
35          if (myid == 0)
36              printf("pi is approximately %.16f, Error is %.16f\n",
37                     pi, fabs(pi - PI25DT));
38      }
39      MPI_Finalize();
```

Calculando Pi IV

```
40      return 0;
41  }
```

Ejercicios Propuestos

Satisfactibilidad de Circuito Lógico

Escribe una nueva versión del problema de satisfactibilidad del circuito lógico de tal forma que devuelva el número total de soluciones.

- Para ello se **modificará** la función check_circuit para que **devuelva** 1 si se satisface alguna **combinación** de entrada y 0 en otro caso.
- Cada proceso **guardará** una variable local de las **entradas** solución al circuito.

Conclusiones

- La "cantidad" de **información** enviada debe **corresponderse** con la **recibida**.
- Las **operaciones** son **bloqueantes**.
- **Simplifican** enormemente el **diseño** de programas paralelos.
- **No** hay uso de **etiquetas** en los mensajes.

Tema 6.4. Comunicación Colectiva.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Tema 6.5. Comunicación Segura.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Abril de 2017

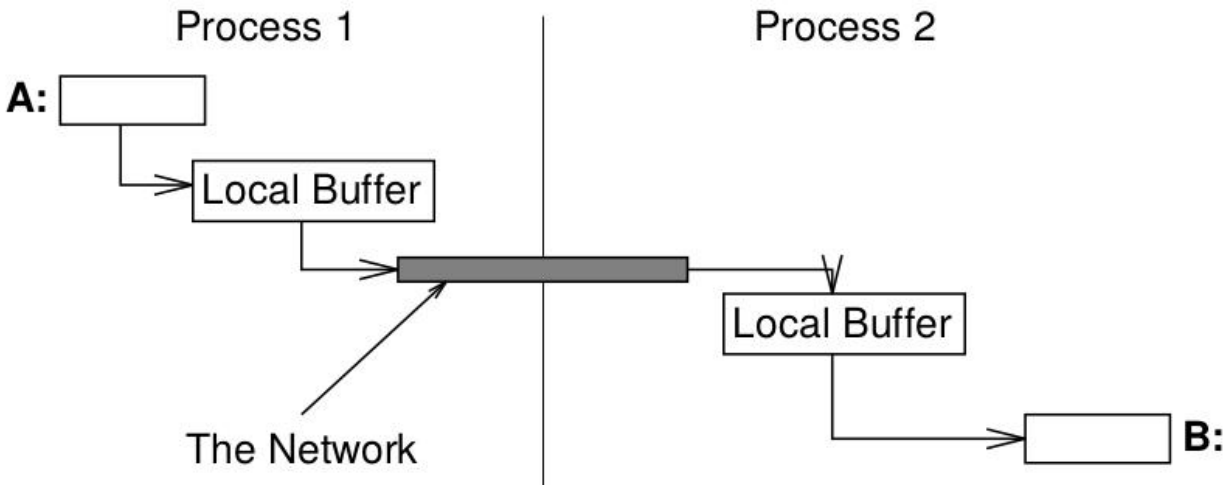
Índice

Índice

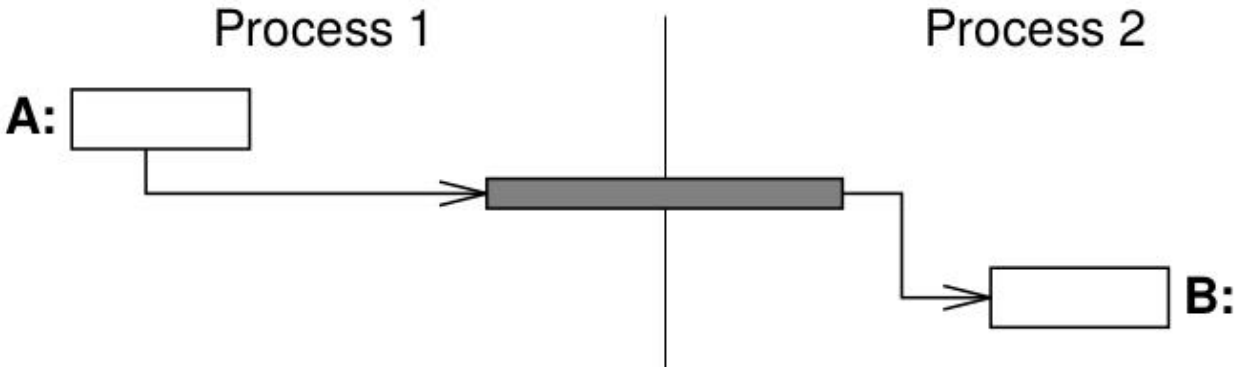
- En esta parte del tema se aclaran aspectos relacionados con la semántica bloqueante de las operaciones MPI_Send and MPI_Recv.
- La principal diferencia con otros sistemas de pasos de mensajes que el alumno puede haber estudiado durante el grado está vinculado a la justificación del bloqueo en el envío.

Índice

Posibles implementaciones de comunicación I



Posibles implementaciones de comunicación II



En este caso pueden darse dos circunstancias:

1. Que send no retorne hasta que el dato haya sido enviado.
2. Que se permite a send retornar antes de concluirse el envío.

Posibles implementaciones de comunicación III

Hasta ahora se ha utilizado comunicación bloqueante:

Tema 6.5. Comunicación Segura.

Servicios de Computación de Altas Prestaciones y Disponibilidad

Máster Universitario en Ingeniería Informática (UCLM)

Abril de 2017