# Metodología de la Programación Paralela

Silvia Perez Ruiz
E-mail: silvia.perezr@um.es, DNI: 49215974E
Raúl Hernández Martínez
E-mail: raul.hernandezm@um.es, DNI: 24423648V
Eduardo Gallego Nicolás
E-mail: eduardo.gallegon@um.es, DNI: 48844607J

## Práctica 2. Paralelización con MPI

Grado en Ingeniería Informática
Metodología de la Programación Paralela
Profesor: Jose Matias Cutillas Lozano (josematias.cutillas@um.es)

UNIVERSIDAD DE MURCIA

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

# Índice

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

# 1 Cuestión 1.

**Enunciado: Indicar qué pasos habría que llevar a cabo para paralelizar el algoritmo con MPI (no es necesario indicar las primitivas de comunicación), justificando adecuadamente cómo se produce el mecanismo de paso de mensajes entre los procesos.**

Lo primero de todo, para poder usar `MPI` en nuestro código, deberemos incluir la librería en el `main.c` e incluiríamos esta linea `#include <mpi.h>`.

Lo siguiente que deberemos hacer en el `main.c` es inicializar y finalizar el entorno de ejecución de `MPI`. Para poder llevar bien la toma de tiempos de forma adecuada, es necesario poner barreras para asegurar así la sincronización de los procesos, para ellos pondremos la instrucción `MPI_Barrier` antes de calcular el tiempo.

Otro aspecto importante que tendremos que cambiar será pasarle a la función `aplicar_ga()` los parámetros `rank` y `size` ya que aunque se declaren en el `main.c` tenemos que usarlos en esa función para diferenciar el proceso 0 de los demás procesos y también saber con cuantos procesos se esta ejecutando el programa.

Además, tenemos que convertir el `array` que hay dentro del `struct Individuo` a una longitud estática, para no tener problemas hemos puesto un tamaño de array igual al de `N`.

Por último, el cambio de mensajes que se produce entre los procesos. Lo primero que hacemos es definir con constatntes nuestro valor de NGM y NEM.

Primero, cada proceso(isla), realiza el calculo local del fitness de su población y lleva a cabo la evolución de sus individuos en su isla. Esto incluye la selección, cruzamiento y mutación.

En cada NGM generaciones se ejecutará el proceso de migración. Cada proceso, selecciona sus mejores NEM individuos de su población de la isla y los envía al proceso 0 (esto se hace con la función `MPI_Send`. El proceso 0 recibe los NEM mejores individuos de todos los demás procesos utilizando `MPI_Recv`, combina las listas de los mejores individuos para formar una lista global con los mejores.

Una vez se ha ordenado la nueva población, el proceso 0 envía los nuevos individuos a los procesos mediante `MPI_Send`. Cada proceso(isla) recibe los nuevos individuos desde el proceso 0 mediando `MPI_Recv`. Estos nuevos individuos se incorporan a la población local de cada isla.

Con todo esto, cada proceso continua con la próxima generación utilizando la población actualizada que incluye a los nuevos individuos.

Para finalizar, al terminar de hacerlo durante `n_gen` hay que volver a enviar al proceso 0 la población de la isla final para que el proceso 0 pueda ordenar la población obteniendo así la población final.

# 2 Cuestión 2.

**Enunciado: Paralelizar el algoritmo genético utilizando primitivas de comunicación síncrona. Hacer uso de las constantes MPI_ANY_SOURCE o MPI_STATUS_IGNORE, si se considera conveniente, para evitar que el proceso 0 quede ocioso durante la recepción de mensajes del resto de procesos. Analizar las prestaciones que se obtienen al variar el número de procesos.**

En este ejercicio, hemos decidido poner la primitiva `MPI_STATUS_IGNORE`, ya que con la primera nos ahorramos estar declarando todo el rato el status del proceso. Esto lo ponemos en todos los `MPI_Recv` y se quedaría así:

```
MPI_Recv(isla_poblacion, isla_poblacion_size, individuo_type, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
```

Listing 1: MPI$_{S}TATUS_{I}GNORE$.

También hemos decidido usar la primitiva `MPI_ANY_SOURCE`, en todos los `MPI_Recv` del proceso 0 ya que es este proceso el que recibe de los demás procesos los mejores individuos de cada isla. Por ejemplo se quedaría así:

```
MPI_Recv(isla_poblacion + (isla_poblacion_size*i) + resto, NEM, individuo_type,
    MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Listing 2: MPI$_{A}NY_{S}OURCE$.

Para poder realizar este ejercicio, hemos seguido la estructura del proceso de comunicación explicada en el ejercicio 1.

Primero hemos creado la población y la hemos inicializado en el proceso 0 con el tamaño de la población total y cuando no es el proceso 0 la hemos inicializado con el tamaño de cada isla que se ha sacado previamente. Cuando el proceso 0 ya hemos copiado toda la población y la hemos ordenado tenemos que enviar a cada proceso (menos al 0) cada parte a cada uno de los procesos existentes. En cambio en el `else` (que son los demás procesos) se tiene que poner `MPI_Recv` para reciban su parte correspondiente de población. Esto lo podemos ver en esta parte del código:

```
if(rank == 0){

    // crea poblacion inicial (array de individuos)
    isla_poblacion = (Individuo *)malloc(tam_pob * sizeof(Individuo));
    assert(isla_poblacion);

    // crea cada individuo (array de enteros aleatorios)
    for (i = 0; i < tam_pob; i++)
    {
      //sub_poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
      int *array = crear_individuo(n);
      for (int j = 0; j < n; j++) {
        isla_poblacion[i].array_int[j] = array[j];
      }
      free(array);

      // calcula el fitness del individuo
      fitness(d, &isla_poblacion[i], n);
    }

    // ordena individuos segun la funcion de bondad (menor "fitness" --> mas aptos)
    qsort(isla_poblacion, tam_pob, sizeof(Individuo), comp_fitness);

```

```
    // Le enviamos a las islas la parte del array correspondiente a partir de la isla 0
    for(i=1; i < size; i++){
      MPI_Send(isla_poblacion+(isla_poblacion_size*i)+resto, isla_poblacion_size,
    individuo_type, i, 0, MPI_COMM_WORLD);
    }
        //Se suma aqui para no pasar m s memoria a los dem s procesos.
    isla_poblacion_size = isla_poblacion_size + resto;

  }
  else{
    isla_poblacion = (Individuo *)malloc(isla_poblacion_size * sizeof(Individuo));
    assert(isla_poblacion);
    // Recibimos la sub-poblacion
    MPI_Recv(isla_poblacion, isla_poblacion_size, individuo_type, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
  }
```

Listing 3: Creación y reparto de la poblacion inicial.

Una vez que cada isla ya tiene su población, tienen que cruzarla, mutarla, sacar el fitness y ordenarla, asi hasta NGM veces. Cuando lo haya hecho NGM veces, cada proceso menos el 0 tendrá que enviarle al 0 sus mejores NEM individuos, el proceso 0 los recibirá y ordenará la nueva población. Una vez ordenada volverá a enviar la población a cada proceso y cada proceso la recibirá y volverá a cruzar, mutar y calcular el fitness, así hasta que se haya hecho n_gen veces. El código de esta parte se quedaría así ( ponemos solo la nueva parte de la migración):

```
if (NGM == contador_NGM){
  if (rank == 0){
    isla_poblacion_size = isla_poblacion_size - resto;

    for(i=1 ; i < size ; i++){
      // Obtenemos los mejores individuos de cada poblacion de cada isla
      MPI_Recv(isla_poblacion + (isla_poblacion_size*i) + resto, NEM, individuo_type,
    MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    qsort(isla_poblacion, tam_pob, sizeof(Individuo), comp_fitness);

    for(i=1 ; i < size ; i++){
      // Se envian a las islas los mejores individuos de todos los que hay (de las demas
    islas)
      MPI_Send(isla_poblacion, NEM, individuo_type, i, 0, MPI_COMM_WORLD);
    }
    isla_poblacion_size = isla_poblacion_size + resto;
  }
  else{
    // Se envian a la isla 0 los mejores individuos
    MPI_Send(isla_poblacion, NEM, individuo_type, 0, 0, MPI_COMM_WORLD);
    // Se reciben los mejores individuos que hay
    MPI_Recv(isla_poblacion, NEM, individuo_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
  contador_NGM = 0;
}
else{
    contador_NGM = contador_NGM + 1;
  }
```

Listing 4: Migración.

Una vez que salimos del for, hay que hacer la recogida de la última población. Para ello decimos que si es el proceso 0 entonces recibe los mejores NEM individuos de los demás procesos y ordena el array, teniendo así la población final. En cambio si no es el proceso 0, se envían los mejores NEM individuos de la población de la isla al proceso 0. Aqui tenemos el código de esta parte:

```
if (rank == 0){
```

```
  isla_poblacion_size = isla_poblacion_size - resto;

  for(i=1 ; i < size ; i++){
    // Obtenemos los mejores individuos de cada poblacion de cada isla
    MPI_Recv(isla_poblacion + (isla_poblacion_size*i) + resto, NEM, individuo_type,
    MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  isla_poblacion_size = isla_poblacion_size + resto;

  qsort(isla_poblacion, tam_pob, sizeof(Individuo), comp_fitness);

}
else{
  // Se envian a la isla 0 los mejores individuos
  MPI_Send(isla_poblacion, NEM, individuo_type, 0, 0, MPI_COMM_WORLD);
}
```

Listing 5: Población final.

Aquí los resultados obtenidos con este ejercicio:

| N | N_GEN | TAM_POB | M_RATE | N_THREADS | T_EJEC | SPEEDUP | EFICIENCIA |
|---|---|---|---|---|---|---|---|
| 2500 | 50 | 800 | 0.15 | 1 | 7,59 seg | 1 | 1 |
| 2500 | 50 | 800 | 0.15 | 2 | 7,17 seg | 1,058 | 0,529 |
| 2500 | 50 | 800 | 0.15 | 3 | 7,23 seg | 1,049 | 0,349 |
| 2500 | 50 | 800 | 0.15 | 4 | 7,21 seg | 1,052 | 0,263 |
| 2500 | 50 | 800 | 0.15 | 5 | 7,22 seg | 1,051 | 0,210 |
| 2500 | 50 | 800 | 0.15 | 6 | 7,38 seg | 1,028 | 0,171 |

# 3 Cuestión 3.

**Enunciado: Modificar la cuestión anterior para que el envío y la recepción de individuos se realice mediante el uso de las funciones MPI_Pack y MPI_Unpack para empaquetamiento de datos, explicando, en este caso, si se podría prescindir del tipo de dato derivado Individuo. Presentar también resultados de los experimentos realizados.**

La gran ventaja que supone usar `MPI_Pack` y `MPI_Unpack` es que reduce el número de llamadas de MPI entre procesos. Otra cosa a tener en cuenta, es que se requiere un almacenamiento adicional para una una copia de datos y además, hay que tener en cuenta el tiempo de computación ya que el código empaquetado ejecuta una llamada de función para cada elemento empaquetado mientras que de la otra forma ejecuta solo un número fijo de llamadas.

En el código utilizamos la función `MPI_Pack` para empaquetar datos de tipo Individuo antes de enviarlos y la función `MPI_Unpack` para desempaquetar los datos en el extremo receptor.

Al emplear un buffer de tamaño estático esto simplifica el proceso de envío y recepción de datos, evitando la necesidad de utilizar el tipo de dato 'Individuo' a través de instrucciones de MPI.

Este es el código de cuando hacer el `MPI_Pack`:

```
int pack_size;
MPI_Pack_size(isla_poblacion+(isla_poblacion_size*i)+resto, individuo_type, MPI_COMM_WORLD,
    &pack_size);
char *packed_data = (char *)malloc(pack_size * size);
int position = 0;
for (int i = 0; i < isla_poblacion_size; ++i) {
    MPI_Pack(&isla_poblacion[i], 1, individuo_type, packed_data, pack_size * size, &position
    , MPI_COMM_WORLD);
}

// Enviar datos empaquetados a otras islas
for (int i = 1; i < size; ++i) {
    MPI_Send(packed_data, position, MPI_PACKED, i, 0, MPI_COMM_WORLD);
}

free(packed_data);
```

Listing 6: Empaquetado.

Este es el código de cuando hacer el `MPI_Unpack`:

```
int pack_size;
MPI_Pack_size(isla_poblacion_size, individuo_type, MPI_COMM_WORLD, &pack_size);
char *packed_data = (char *)malloc(pack_size);
MPI_Recv(packed_data, pack_size, MPI_PACKED, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Desempaquetar datos en isla_poblacion
int position = 0;
for (int i = 0; i < isla_poblacion_size; ++i) {
    MPI_Unpack(packed_data, pack_size, &position, &isla_poblacion[i], 1, individuo_type,
    MPI_COMM_WORLD);
}

free(packed_data);
```

Listing 7: Desempaquetado.

Aquí están los resultados:

| N | N_GEN | TAM_POB | M_RATE | N_THREADS | T_EJEC | SPEEDUP | EFICIENCIA |
|---|---|---|---|---|---|---|---|
| 2500 | 50 | 800 | 0.15 | 1 | 7,34 seg | 1 | 1 |
| 2500 | 50 | 800 | 0.15 | 2 | 7,24 seg | 1,013 | 0,506 |
| 2500 | 50 | 800 | 0.15 | 3 | 7,28 seg | 1,008 | 0,336 |
| 2500 | 50 | 800 | 0.15 | 4 | 7,28 seg | 1,008 | 0,252 |
| 2500 | 50 | 800 | 0.15 | 5 | 7,43 seg | 0,987 | 0,197 |
| 2500 | 50 | 800 | 0.15 | 6 | 7,32 seg | 1,002 | 0,167 |

# 4 Cuestión 4.

**Enunciado: Paralelizar el algoritmo genético utilizando primitivas de comunicación asíncrona. Analizar las prestaciones obtenidas al variar el número de procesos**

Con el uso de las primitivas `MPI_Isend`, `MPI_Irecv` y `MPI_Wait`, se puede lograr la comunicación asíncrona entre los procesos, lo que permitirá la ejecución simultánea de diferentes partes del algoritmo genético.

Hemos reemplazado los envíos y recepciones regulares (`MPI_Send` y `MPI_Recv`) por sus versiones asíncronas (`MPI_Isend` e `MPI_Irecv`).

Hay que tener en cuenta que como `MPI_Irecv` es no bloqueante necesitamos hacer un `MPI_Wait` para esperar a la `request`. Esto se puede ver reflejado en el código ya que después de cada `MPI_Irecv` hemos puesto un `MPI_Wait`. Además, hay que tener en cuenta que en el último `MPI_Isend` se tiene que hacer un `MPI_Wait` ya que no podemos asegurar que le llegue al proceso 0 antes de libere la memoria. Por eso hay que poner el `MPI_Wait` en el último `MPI_Isend`.

Aquí tenemos unos ejemplos de cómo lo usamos:

```
// Le enviamos a las islas la parte del array correspondiente a partir de la isla 0
for(i=1; i < size; i++){
  MPI_Isend(isla_poblacion+(isla_poblacion_size*i)+resto, isla_poblacion_size,
     individuo_type, i, 0, MPI_COMM_WORLD, &send_request);
}
```

Listing 8: Envío con Isend.

```
// Recibimos la sub-poblacion
MPI_Irecv(isla_poblacion, isla_poblacion_size, individuo_type, 0, 0, MPI_COMM_WORLD, &
    recv_request);
MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
```

Listing 9: Recibimos con Irecv.

```
// Se envian a la isla 0 los mejores individuos
MPI_Isend(isla_poblacion, NEM, individuo_type, 0, 0, MPI_COMM_WORLD, &send_request);
MPI_Wait(&send_request, MPI_STATUS_IGNORE);
```

Listing 10: Último Isend.

Aquí están los resultados:

| N | N_GEN | TAM_POB | M_RATE | N_THREADS | T_EJEC | SPEEDUP | EFICIENCIA |
|---|---|---|---|---|---|---|---|
| 2500 | 50 | 800 | 0.15 | 1 | 7,3 seg | 1 | 1 |
| 2500 | 50 | 800 | 0.15 | 2 | 7,21 seg | 1,012 | 0,506 |
| 2500 | 50 | 800 | 0.15 | 3 | 7,22 seg | 1,011 | 0,337 |
| 2500 | 50 | 800 | 0.15 | 4 | 7,22 seg | 1,011 | 0,252 |
| 2500 | 50 | 800 | 0.15 | 5 | 7,25 seg | 1,006 | 0,201 |
| 2500 | 50 | 800 | 0.15 | 6 | 7,24 seg | 1,008 | 0,168 |

# 5 Cuestión 5.

**Enunciado: Paralelizar el algoritmo genético utilizando primitivas de comunicación colectiva. Analizar las prestaciones obtenidas al variar el número de procesos.**

Paralelizar un algoritmo genético utilizando primitivas de comunicación colectiva implica utilizar operaciones que involucran a todos los procesos en una comunicación conjunta. Estas operaciones permiten una comunicación eficiente y coordinada entre todos los procesos, lo que puede ser beneficioso en términos de rendimiento y escalabilidad en entornos de ejecución distribuida.

Para implementar la paralelización del algoritmo genético, necesitamos dividir la población global entre los procesos utilizando la función `MPI_Scatter`. Después, los resultados de cada proceso se reúnen mediante `MPI_Gather`, y para la comunicación eficiente entre los procesos, se utiliza `MPI_Bcast` para compartir la información.

**`MPI_Gather`**: Se utiliza para recopilar datos de todos los procesos y enviarlos al proceso raíz. Utilizamos `MPI_Gather` para reunir los resultados de las diferentes islas y recuperar los mejores individuos en el proceso 0.

**`MPI_Bcast`**: Esta primitiva se utiliza para enviar un mensaje desde el proceso raíz a todos los demás procesos en el mismo grupo. Utilizamos `MPI_Bcast` para difundir los mejores individuos de la población a todos los procesos después de que se han reunido en el proceso raíz mediante el `MPI_Gather`.

```
// Inicializaci n de la poblaci n usando Scatter
MPI_Scatter(isla_poblacion, isla_poblacion_size, individuo_type, rcv_buff,
    isla_poblacion_size, individuo_type, 0, MPI_COMM_WORLD);
```

Listing 11: Uso del Scatter.

```
if (NGM == contador_NGM){
    isla_poblacion_size = isla_poblacion_size - resto;
    MPI_Gather(rcv_buff, isla_poblacion_size, individuo_type, isla_poblacion,
    isla_poblacion_size, individuo_type, 0, MPI_COMM_WORLD);

  if (rank == 0){
    qsort(isla_poblacion, tam_pob, sizeof(Individuo), comp_fitness);
  }

     MPI_Bcast(isla_poblacion, tam_pob, individuo_type, 0, MPI_COMM_WORLD);
  isla_poblacion_size = isla_poblacion_size + resto;
  contador_NGM = 0;
}
else{
  contador_NGM = contador_NGM + 1;
}
```

Listing 12: Envio de las mejores Gen.

Aquí están los resultados:

| N | N_GEN | TAM_POB | M_RATE | N_THREADS | T_EJEC | SPEEDUP | EFICIENCIA |
|---|---|---|---|---|---|---|---|
| 2500 | 50 | 800 | 0.15 | 1 | 9,69 seg | | |
| 2500 | 50 | 800 | 0.15 | 2 | 9,61 seg | 1,008 | 0,504 |
| 2500 | 50 | 800 | 0.15 | 3 | 9,38 seg | 1,033 | 0,344 |
| 2500 | 50 | 800 | 0.15 | 4 | 9,33 seg | 1,038 | 0,259 |

| N | N_GEN | TAM_POB | M_RATE | N_THREADS | T_EJEC | SPEEDUP | EFICIENCIA |
|---|---|---|---|---|---|---|---|
| 2500 | 50 | 800 | 0.15 | 5 | 9,50 seg | 1,02 | 0,204 |
| 2500 | 50 | 800 | 0.15 | 6 | 9,32 seg | 1,039 | 0,173 |

# 6 Cuestión 6.

**Enunciado: Realizar el informe final incluyendo las conclusiones generales y valoración personal sobre el trabajo realizado.**

Este proyecto nos ha servido para saber como paralelizar de diferentes maneras con MPI. La idea de dividir el código en islas para poder trabajar con cada proceso con una división de manera eficiente y efectiva no nos ha parecido tan fácil como pensábamos. Igualmente, en cuanto a los algoritmos genéticos hemos visto cómo la estructura natural de estos algoritmos se presta bien para la paralelización. La capacidad de dividir una población en subgrupos y permitir que evolucionen de forma independiente antes de compartir y combinar los resultados, refleja cómo ocurren los procesos de selección natural en la naturaleza.

Algo negativo, utilizar el proyecto de la práctica 0 sin ser corregido nos ha dado la preocupación de que podríamos arrastrar errores que afectaran nuestro trabajo, algo que nos pasó igual respecto a la práctica anterior. Además, al ejecutar los códigos y jugar con el número de procesos, a veces el sistema no respondía como nosotros esperábamos y desconocíamos del problema.

En resumen, este proyecto nos ha enseñado más allá de las lecciones teóricas, ya que nos ha demostrado con las diferentes maneras de programación paralela, que es un arte que implica más que simplemente dividir el trabajo. Además, hemos comprendido que los algoritmos genéticos se benefician enormemente de la paralelización.