Práctica 2

Metodología de la Programación Paralela
Curso 2024/25

Paralelización con MPI


Miembros del Grupo:
Miguel Saez Martinez miguel.saezm@um.es
Juan Hernández Acosta juan.hernandeza@um.es

# Índice

**Cuestión 1.**

**a) Indicar qué pasos habría que llevar a cabo para paralelizar el algoritmo con MPI (no es necesario indicar las primitivas de comunicación). Incluir un esquema en pseudocódigo que refleje el patrón de computación y comunicaciones entre procesos en el modelo de islas, desde la lectura inicial de parámetros de entrada y su distribución, el intercambio de información entre procesos durante el desarrollo del algoritmo, la computación en paralelo, y la comunicación final de resultados.**

El modelo de islas para el algoritmo evolutivo requiere que cada proceso trabaje con una subpoblación (isla), realice computación localmente y participe en migraciones de individuos en intervalos específicos.

Pseudocódigo general

1. Inicialización

Configurar MPI:
Llamar a MPI_Init.
Obtener el rango del proceso (MPI_Comm_rank) y el número de procesos (MPI_Comm_size).

Leer y distribuir datos iniciales (en el proceso maestro):
Leer los parámetros de entrada (n, m, n_gen, tam_pob, NGM, NEM).
Generar la matriz de distancias y la población inicial.
Dividir la población inicial entre los procesos.
Enviar las subpoblaciones y la matriz de distancias a los procesos trabajadores.

2. Computación local en paralelo

Cada proceso ejecuta el algoritmo evolutivo sobre su subpoblación durante NGM generaciones.
Al finalizar cada intervalo de NGM, se realiza una migración de individuos.

3. Migración de individuos

En procesos trabajadores:
Seleccionar los NEM mejores individuos y enviarlos al proceso maestro.
En el proceso maestro:
Recibir individuos de todos los procesos.
Mezclar y ordenar la población combinada.
Seleccionar y redistribuir los mejores individuos a cada proceso.

4. Finalización

Cada proceso envía su mejor solución al maestro.
El maestro selecciona la solución global óptima.
Finalizar el entorno MPI con MPI_Finalize.

Pseudocódigo del patrón de computación y comunicación

```
// Inicialización de MPI
MPI_Init
rank ← MPI_Comm_rank
size ← MPI_Comm_size

SI rank == 0 ENTONCES
        Leer parámetros de entrada
        Generar matriz de distancias y población inicial
        Dividir población en subpoblaciones
        Enviar subpoblaciones y matriz a cada proceso
FIN SI

Recibir subpoblación y matriz

PARA generaciones = 1 HASTA n_gen HACER
        Realizar evolución local en subpoblación

        SI generaciones % NGM == 0 ENTONCES
        // Migración
        Enviar mejores NEM individuos al maestro
        SI rank == 0 ENTONCES
        Recibir individuos de todos los procesos
        Mezclar y ordenar población
        Redistribuir mejores individuos
        FIN SI
        Recibir nuevos individuos
        FIN SI
FIN PARA

Enviar mejor solución al maestro

SI rank == 0 ENTONCES
        Recibir soluciones de todos los procesos
        Seleccionar solución global óptima
        Imprimir resultados
FIN SI

MPI_Finalize
```

**b) Las primitivas como MPI_Send para comunicaciones con paso de mensajes pueden enviar arrays de datos del mismo tipo a condición de que estos estén contiguos en memoria. El código actual implementa la población como un array de punteros a Individuo con lo que los datos pueden estar en posiciones no contiguas de memoria. Como consecuencia se puede producir un error en el envío de poblaciones o subpoblaciones de individuos. Modifica el código como consideres mejor para solucionar este inconveniente, teniendo en cuenta que puedes cambiar de población de *Individuo a Individuo o reservar memoria dinámica auxiliar contigua y apuntar a ella para crear la población.**

Actualmente, la población está definida como un array de punteros a Individuo, lo que provoca problemas al utilizar primitivas MPI (requieren datos contiguos en memoria). Implementaremos la siguiente solución en el archivo cabecera:

Cambiaremos array_int a un array estático dentro de Individuo en la cabecera correspondiente:

```
typedef struct {
int array_int[MAX_SIZE];
double fitness;
} Individuo;
```

**Cuestión 2.**

**Paralelizar el algoritmo evolutivo 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. Tener en cuenta el punto (3) en Observaciones para determinar los valores de NEM y NGM.**

Se ha tomado la decisión de diseño de que NEM sea siempre igual al tamaño de la población (m) dividido por el número de procesos. De esta forma, se facilita el estudio y la configuración del código al saber siempre cuantos elementos de la población irán en cada isla.

Las diferencias más notables entre el código secuencial y la versión paralela con comunicación síncrona radican, sobre todo, en la forma en que se distribuye la población entre procesos y cómo se sincronizan esos procesos.

En el código secuencial, solo existe una población global “poblacion[]” gestionada por un único programa, y todas las fases —generación de individuos, cruce, mutación, ordenación— se realizan en un único flujo de ejecución. En la versión que emplea MPI, el flujo está dividido en procesos. Al principio, solo el maestro (rank 0) contiene la población completa. Este maestro la inicializa con:

```
        // Inicializar población completa en el maestro
        for (int i = 0; i < tam_pob; i++)
        {
            inicializar_individuo(&poblacion[i], n, m);
            fitness(d, &poblacion[i], n, m);
        }
```

Luego, los procesos trabajadores (rank ≠ 0) reciben sus trozos de la población mediante llamadas como MPI_Scatterv, que reparten los individuos:

```
// Distribuir subpoblaciones a todos los procesos
        MPI_Scatterv(poblacion, sendcounts, displs, individuo_type,
                     sub_poblacion, sendcounts[rank], individuo_type, 0, MPI_COMM_WORLD);
```

Estos procesos ejecutan las fases de evolución local (cruce, mutación, re-ordenación por *fitness*) únicamente sobre su porción, mediante los siguientes bucles, durante n_gen generaciones:

```
  // Cruce
            for (int i = 0; i < sendcounts[rank] / 2 - 1; i += 2)
```

```
        {
            cruzar(&sub_poblacion[i], &sub_poblacion[i + 1],
                   &sub_poblacion[sendcounts[rank] / 2 + i],
                   &sub_poblacion[sendcounts[rank] / 2 + i + 1], n, m);
        }

        // Mutación
        for (int i = 0; i < sendcounts[rank]; i++)
        {
            mutar(&sub_poblacion[i], n, m);
            fitness(d, &sub_poblacion[i], n, m);
        }
```

y luego reenvían la subpoblación evolucionada al maestro con una llamada a MPI_Gatherv. El maestro vuelve a reunir todos los individuos en el array “poblacion” y puede reordenarlos globalmente con:

```
    // Recolectar las subpoblaciones en el maestro
    MPI_Gatherv(sub_poblacion, sendcounts[rank], individuo_type,
                poblacion, sendcounts, displs, individuo_type, 0, MPI_COMM_WORLD);

    // Proceso maestro realiza migración
    if (rank == 0)
    {
        // Ordenar población global por fitness
        qsort(poblacion, tam_pob, sizeof(Individuo), comp_fitness);

        // Migración: seleccionar los mejores y redistribuir
        // Aquí simplemente volvemos a distribuir en este caso
    }
```

De este modo, el papel central de la versión paralela consiste en **distribuir** la población, procesarla por partes en cada proceso y **volverla a recolectar** para sincronizar los resultados.

También hay una diferencia en la forma de crear la estructura de datos de cada individuo para MPI. Se define un tipo derivado con MPI_Type_create_struct, indicando que cada “Individuo” consta de un array de enteros (los genes) y un valor en coma flotante (el fitness). Después de ejecutar el código paralelizado con distintos valores para NP y NGM, según vemos en los resultados, los valores más adecuados para estas variables varían dependiendo del tamaño de la entrada. Sin embargo, de forma general, podemos determinar que los valores que usaremos a partir de ahora para el resto de ejercicios son:

NP: 2,4,8
NGM: 1,5

Una vez establecidos los valores de NEM, NGM y NP que usaremos para futuras cuestiones, procedemos a analizar los resultados que se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p2ej2_paracomusinc** o

puede acceder directamente usando este enlace:
Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

Como se puede observar, los beneficios de paralelizar nuestro algoritmo genético con paso de mensajes y memoria compartida solo se empiezan a apreciar de forma clara (eficiencias superiores a 1) en aquellos casos en los que el tamaño de entrada es grande, se utilizan únicamente dos procesos y la cantidad de generaciones locales por proceso es pequeña.Esto suele responder a varios motivos.

Primero, cuando el tamaño de la población y los parámetros del problema (n, m) son grandes, el trabajo a repartir entre los procesos o hilos resulta suficientemente significativo para que la sobrecarga de comunicación y sincronización quede amortizada. Por el contrario, en problemas muy pequeños o con demasiados procesos, esa sobrecarga a menudo supera la ganancia potencial.

Segundo, si se usan solo dos procesos en entornos de paso de mensajes, la coordinación es más sencilla y disminuye el número de envíos y recepciones de datos. Esto ayuda a reducir el tiempo que los procesos pasan esperando o comunicándose, por lo que la eficiencia puede llegar a superar el 100%.

Tercero, mantener un número reducido de generaciones locales (por ejemplo, una sola generación) minimiza el tiempo de computación en cada proceso antes de volver a sincronizarse, de modo que no hay largos intervalos en los que la carga de un proceso podría desequilibrarse respecto a los demás. Este intercambio más frecuente de información también contribuye a que la población global se mantenga coherente y aproveche rápidamente las mejoras generadas en cada isla.

**Cuestión 3.**

**Modificar la cuestión anterior para que el envío y la recepción de datos se realice mediante el uso de las funciones MPI_Pack y MPI_Unpack para empaquetamiento de datos, explicando si se debe usar el tipo derivado Individuo. Presentar también resultados de los experimentos realizados.**

Para implementar este ejercicio haremos uso de dos funciones de apoyo para el empaquetado y el desempaquetado.

Empaquetado

```
static void pack_individuos(Individuo *poblacion, int start, int count,
                            int m, char *buffer, int buffer_size, int root)
{
    int position = 0;
    for (int i = 0; i < count; i++)
    {
        // Empaqueta array_int
        MPI_Pack(poblacion[start + i].array_int, m,
                 MPI_INT, buffer, buffer_size, &position, MPI_COMM_WORLD);
        // Empaqueta fitness
        MPI_Pack(&(poblacion[start + i].fitness), 1,
                 MPI_DOUBLE, buffer, buffer_size, &position, MPI_COMM_WORLD);
    }

    // Envía a 'root' (o a cada rank, según sea un gather o scatter).
    // Nota: El "TAG" y si se usa MPI_Send/MPI_Bcast/etc. depende del contexto.
    MPI_Send(buffer, position, MPI_PACKED, root, 0, MPI_COMM_WORLD);
}
```

Desempaquetado

```
static void unpack_individuos(Individuo *dest, int count, int m,
                              char *buffer, int buffer_size)
{
    int position = 0;
    for (int i = 0; i < count; i++)
    {
        MPI_Unpack(buffer, buffer_size, &position,
                   dest[i].array_int, m, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, buffer_size, &position,
                   &(dest[i].fitness), 1, MPI_DOUBLE, MPI_COMM_WORLD);
    }
}
```

Usos

```
double aplicar_mh(const double *d, int n, int m, int n_gen,
                  int tam_pob, int *sol, int ngm){

    // …

    // ========================================
    // Bucle de generaciones
    // ========================================
    for (int g = 0; g < n_gen; g++)
    {
        if (rank == 0){
            for (int i = 0; i < size; i++){
                if (i == 0){
                    // …
```

```
                }else{
                    int count_i = sendcounts[i];
                    int start_i = displs[i];
                    int buffer_size = count_i * size_one_ind;
                    pack_individuos(poblacion, start_i, count_i, m,
                                    buffer, buffer_size, i);
                }
            }
        }else{
            int count_local = local_count;
            int buffer_size = count_local * size_one_ind;
            MPI_Recv(buffer, buffer_size, MPI_PACKED, 0, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            unpack_individuos(sub_poblacion, count_local, m,
                              buffer, buffer_size);
        }

        // …

        if (rank == 0){
            memcpy(&poblacion[displs[0]],
                   sub_poblacion,
                   sendcounts[0] * sizeof(Individuo));
            for (int i = 1; i < size; i++){
                int count_i = sendcounts[i];
                int start_i = displs[i];
                int buffer_size = count_i * size_one_ind;

                MPI_Recv(buffer, buffer_size, MPI_PACKED, i, 0,
                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                unpack_individuos(&poblacion[start_i], count_i, m,
                                  buffer, buffer_size);
            }
            qsort(poblacion, tam_pob, sizeof(Individuo), comp_fitness);
        }else{
            int buffer_size = local_count * size_one_ind;
            pack_individuos(sub_poblacion, 0, local_count, m,
                            buffer, buffer_size, 0);
        }
    }

    // …

    return best_fitness;
}
```

Como podemos observar en los resultados aunque existe una mejora general respecto a los resultados sin empaquetamiento únicamente las ejecuciones con tamaños de n grandes obtienen speed-ups y eficiencias por encima de 1.

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p1ej3_empaquetamiento** o puede acceder directamente usando este enlace:
Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

**Cuestión 4.**

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

Para implementar este ejercicio simplemente cambiaremos las funciones de envío y recepción MPI_* por sus versiones MPI_I*

```
double aplicar_mh(const double *d, int n, int m, int n_gen, int tam_pob, int *sol, int
ngm)
{

    // …

    for (int g = 0; g < n_gen; g++){
        if (rank == 0){

            // …

            for (int i = 1; i < size; i++){
                MPI_Isend(&poblacion[displs[i]],
                          sendcounts[i],
                          individuo_type,
                          i,
                          TAG_SCATTER,
                          MPI_COMM_WORLD,
                          &reqs[idx_req++]);
            }

            // …

        }else{
            MPI_Request req;
            MPI_Irecv(sub_poblacion,
                      sendcounts[rank],
                      individuo_type,
                      0,
                      TAG_SCATTER,
                      MPI_COMM_WORLD,
                      &req);

            MPI_Wait(&req, MPI_STATUS_IGNORE);
        }

            // …

        if (rank == 0){

            // …

            for (int i = 1; i < size; i++){
                MPI_Irecv(&poblacion[displs[i]],
                          sendcounts[i],
                          individuo_type,
                          i, // origen
                          TAG_GATHER,
                          MPI_COMM_WORLD,
                          &reqs[idx_req++]);
            }

            // …
        }
        else
        {
            MPI_Request req;
            MPI_Isend(sub_poblacion,
                      sendcounts[rank],
                      individuo_type,
                      0,
```

```
                    TAG_GATHER,
                    MPI_COMM_WORLD,
                    &req);

            MPI_Wait(&req, MPI_STATUS_IGNORE);
        }

    }

                // …
}
```

Una vez más, observamos en los resultados cierta mejora respecto a los anteriores, en este caso la inclusión del empaquetamiento, ya que perfilamos las mejoras anteriores. De la misma forma que antes los tamaños de n grandes son los únicos con una eficiencia superior a 1 indicandonos que la paralelización es efectiva especialmente conforme crece la carga de trabajo

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p2ej4_paracomuasinc** o puede acceder directamente usando este enlace:
Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

**Cuestión 5.**

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

```
double aplicar_mh(const double *d, int n, int m, int n_gen,
                  int tam_pob, int *sol, int ngm){

    // …

    for (int g = 0; g < n_gen; g++)
    {

        MPI_Scatterv(
            /* sendbuf    = */ poblacion, // sólo útil en rank=0
            /* sendcounts = */ sendcounts,
            /* displs     = */ displs,
            /* sendtype   = */ individuo_type,
            /* recvbuf    = */ sub_poblacion, // buffer local
            /* recvcount  = */ local_count,
            /* recvtype   = */ individuo_type,
            /* root       = */ 0,
            /* comm       = */ MPI_COMM_WORLD);

        // …

        // 5.3) Recolectar subpoblaciones con MPI_Gatherv
        MPI_Gatherv(
            /* sendbuf    = */ sub_poblacion,
            /* sendcount  = */ local_count,
            /* sendtype   = */ individuo_type,
            /* recvbuf    = */ poblacion, // sólo válido en rank=0
            /* recvcounts = */ sendcounts,
            /* displs     = */ displs,
            /* recvtype   = */ individuo_type,
            /* root       = */ 0,
            /* comm       = */ MPI_COMM_WORLD);

        // …
    }

    // …
}
```

Al igual que en los casos anteriores la comunicación colectiva funciona decentemente en tamaños grandes y pobremente en tamaños pequeños. En este caso, sin embargo, no encontramos una mejora consistente frente a la versión con comunicación asíncrona.

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p2ej5_paracomucolec** o puede acceder directamente usando este enlace:
Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

**Cuestión 6.**

**Elegir la mejor configuración de parámetros y tipo de comunicaciones para paralelizar la totalidad del algoritmo. Justificar las decisiones tomadas. De este modo, se pretende tener un código paralelo óptimo en memoria distribuida que se podrá usar para comparar con el código secuencial, y paralelo en memoria compartida e híbrido.**

En esta versión completamente paralelizada, se ha optado por **MPI_Scatterv** para distribuir la población porque se pretendía manejar de forma flexible casos en los que el número total de individuos no sea múltiplo del número de procesos. Con *MPI_Scatterv*, es posible asignar a cada proceso el número exacto de individuos que le corresponde, incluso si algunos reciben uno o varios individuos de sobra cuando la división no sea perfecta. Otras funciones como *MPI_Scatter* no permiten tan fácilmente este reparto irregular, por lo que *MPI_Scatterv* resulta más eficaz en escenarios donde la población y el número de procesos no están perfectamente balanceados.

El opuesto simétrico de este reparto se realiza con **MPI_Gatherv**, que reúne los fragmentos de población en el maestro tras cada etapa de evolución local. De nuevo, se recurre a esta variante "*v*" para acomodar las diferencias en el tamaño de cada subpoblación. Así, tanto el maestro como cada proceso saben exactamente cuántos individuos hay que enviar y recibir, garantizando la reconstrucción completa de la población sin complicaciones.

Cuando se envían y reciben los datos de los individuos, se emplea el tipo derivado de MPI construido con **MPI_Type_create_struct**. Este tipo describe tanto el array de genes como la variable de *fitness* en una sola entidad, haciendo posible una sola operación de envío o recepción para transferir todos los campos del individuo. En lugar de separar el envío de genes y *fitness* en múltiples mensajes, este tipo derivado reduce la complejidad y el número de intercambios, lo que suele mejorar el rendimiento.

Además, algunas comunicaciones se han diseñado para ser no bloqueantes (asincrónicas) mediante el uso de *MPI_Isend* y *MPI_Irecv*, con el objetivo de que los procesos puedan realizar otras tareas mientras los datos se trasladan por la red. No obstante, cada vez que se requiere un punto de sincronización (por ejemplo, para garantizar que la información esté lista antes de la siguiente fase de la evolución), se incluye la espera correspondiente (como *MPI_Wait* o *MPI_Waitall*). De ese modo, se equilibra la paralelización con la necesidad de que todos los procesos dispongan de los datos coherentes en el momento preciso.

En cuanto a la comunicación colectiva —por ejemplo, *MPI_Bcast*—, se opta por ella en partes del código donde todos los procesos necesitan actualizarse simultáneamente con la misma información, o sincronizar de forma barrera. Para la parte principal, sin embargo, se recurre al reparto selectivo (con *Scatterv* y *Gatherv*) y a la migración en el maestro porque la población se maneja principalmente en fragmentos independientes, y solo cuando es preciso unificar o redistribuir resultados se recurre a la recolección y posterior difusión.

Como podemos observar, al igual que en casos anteriores, obtenemos los mejores resultados de eficiencia y speedup en aquellos casos donde el tamaño de la entrada es mayor. Esto sucede porque, cuando la población y los parámetros del problema crecen lo suficiente, el número de operaciones necesarias para la evolución del algoritmo genético

aumenta de forma que la sobrecarga asociada a la comunicación y a la sincronización entre procesos queda más que compensada. Dicho de otro modo, el coste fijo de coordinar los procesos o hilos se reparte entre una cantidad mucho mayor de cómputo, de modo que la paralelización se aprovecha mejor y la mejora de rendimiento es más significativa.

Los resultados de la ejecución de este código se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p2ej6_totalpm** o puede acceder directamente usando este enlace:
Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx