# PRÁCTICA 2 – MEMORIA MPP

Alberto Gálvez Gálvez | alberto.galvezg@um.es | 49448576W

Antonio Martínez Fernández | antonio.martinezf@um.es | 49973734R

Mario Rodríguez Béjar | mario.rodriguezb2@um.es | 49339550l

# Contenido

# 1. ESQUEMA DE PARALELIZACIÓN

A continuación, se muestra un diagrama que explica el esquema algorítmico empleado para la paralelización del algoritmo genético. Por simplicidad se ha ilustrado con 4 procesos, pero la misma lógica es aplicable a cualquier número de procesos.

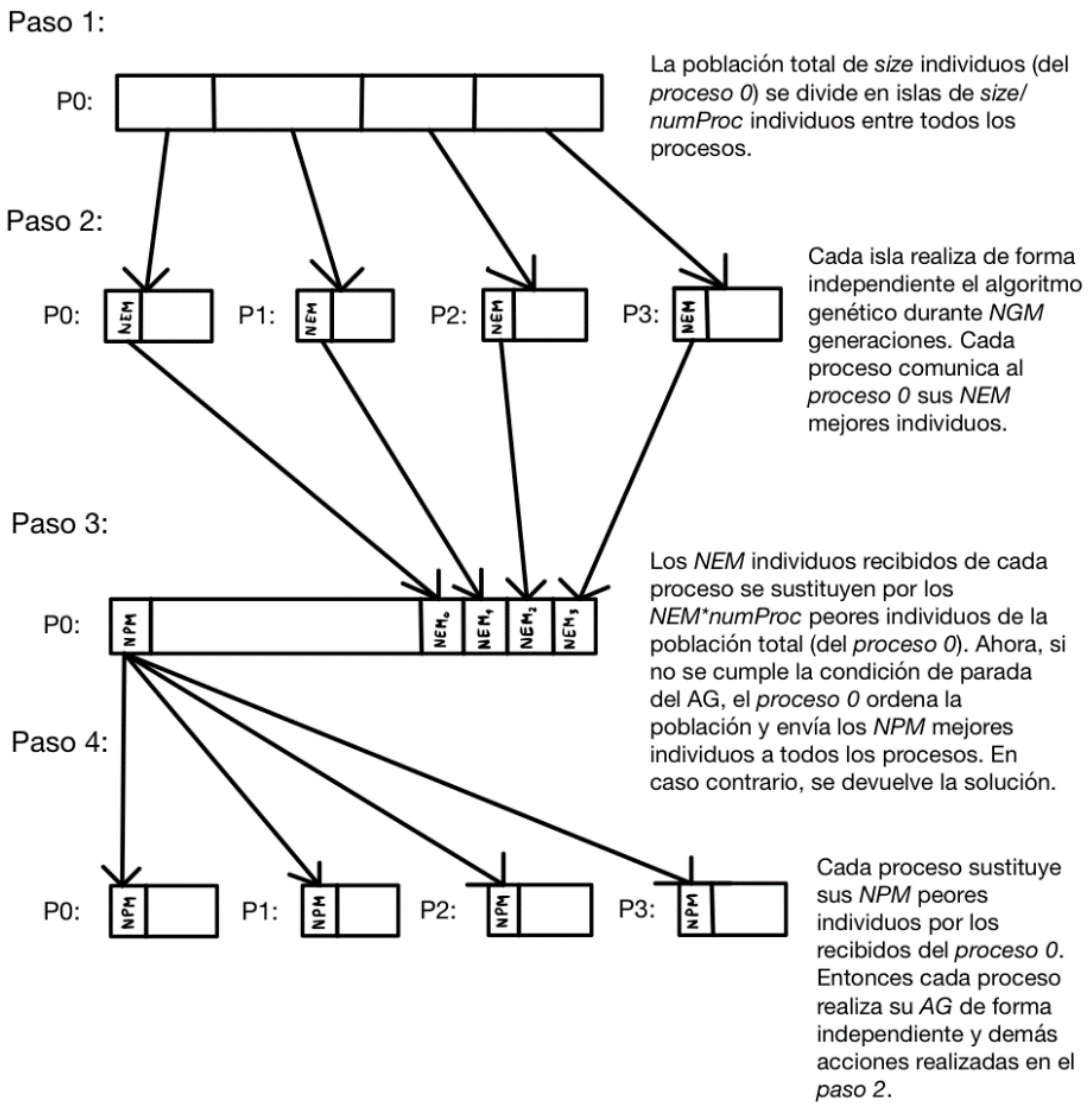


*Ilustración 1. Esquema algorítmico de la paralelización del algoritmo genético*

- ***AG***: algoritmo genético.
- ***size***: número de individuos de la población total.
- ***numProc***: número de procesos.
- ***NGM***: número de generaciones durante las cuales cada isla ejecuta el AG de forma independiente.
- ***NEM***: número de individuos que comunica al *proceso 0* cada proceso.
- ***NPM***: número de individuos que comunica el *proceso 0* a todos los procesos.

# 2. PARALELIZACIÓN CON COMUNIACIÓN SÍNCRONA

Para llevar a cabo la paralelización síncrona, debemos usar *MPI_Ssend* y *MPI_Recv*. Usaremos un esquema en el cual el hilo padre guarda el *chunk* que le toca procesar y mediante un bucle con la función *MPI_Ssend* enviamos a los hijos la dirección de inicio de los individuos que tiene que procesar. A su vez los hijos esperan con *MPI_Recv*.

```
MPI_Status status;
if(idProceso==0){
    // El padre coge el primer bloque de chunksize y envia a los hijos las demás partes
    for(int i=0;i<chunkSize;i++){
        islaPoblacion[i]=poblacion[i];
    }
    // Enviamos a cada hijo su sección de la población
    for(int i=1;i<numProcesos;i++){
        MPI_Ssend(&poblacion[i*chunkSize], chunkSize, typeIndividuo, i, 0, MPI_COMM_WORLD);
    }
} else{
    // Recibimos la seccion de la poblacion
    MPI_Recv(islaPoblacion, chunkSize, typeIndividuo, 0, 0, MPI_COMM_WORLD, &status);
}
```

*Ilustración 2. Repartición chunsize entre los hijos*

Posteriormente ordenamos los individuos con *qsort* lo que nos permite tener para cada chunk los mejores en cuanto a fitness. A continuación seleccionamos *NEM* individuos de cada *chunk* y los almacenamos en *poblacionNEM*. Los hijos envían al padre los *NEM* individuos que son mejores.

```
// Obtenemos los mejores individuos de la poblacion local para enviarle los NEM al master
for(int i=0;i<NEM;i++){
    islaPoblacionNEM[i]=islaPoblacion[i];
}

if(idProceso==0){
    // Guardamos la poblacionNEM del padre en el array donde almacenaremos
    // los NEM individuos de cada proceso
    for(int i=0;i<NEM;i++){
        poblacionNEM[i]=islaPoblacionNEM[i];
    }

    // Recibimos los NEM individuos de cada proceso y los almacenamos juntos
    for(int i=1;i<numProcesos;i++){
        MPI_Recv(islaPoblacionNEM, NEM,typeIndividuo,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
        for(int j=0;j<NEM;j++){
            poblacionNEM[NEM*i+j]=islaPoblacionNEM[j];
        }
    }
}
else{   // si eres hijo envias al padre
    MPI_Ssend(islaPoblacionNEM, NEM, typeIndividuo, 0, 0, MPI_COMM_WORLD);
}
```

*Ilustración 3. Envío de los mejores individuos al padre*

Acto seguido, sustituímos en la poblacion original los *NPM* peores individuos por los *NEM* mejores de cada isla y para finalizar se reparte esta población entre los hijos de manera que cada uno obtiene un trozo de tamaño chunksize y el proceso puede volver a repetirse hasta que se alcance el número de generaciones.

```
if (idProceso == 0){
    // Sustituimos los NEM*numprocesos peores individuos de poblacion por los mejores NEM individuos de cada proceso
    int index_poblacion_nem = 0;
    for (int i = tam_poblacion - 1; i > tam_poblacion - 1 - (numProcesos * NEM); i--)
    {
        poblacion[i] = poblacionNEM[index_poblacion_nem];
        index_poblacion_nem++;
    }

    // Ordenar individuos según la función de bondad (menor "fitness" --> más aptos)
    qsort(poblacion, tam_poblacion, sizeof(Individuo), comp_fitness);

    for(int i=0;i<NPM;i++){
        poblacionNPM[i]=poblacion[i];
    }
}

// Compartimos los datos de la imagen leida a los hijos
if(idProceso==0){  // enviamos poblacionNPM a todos los hijos
    for(int i=1;i<numProcesos;i++){
        MPI_Ssend(poblacionNPM, NPM, typeIndividuo, i, 0, MPI_COMM_WORLD);
    }
} else{ // recibimos poblacionNPM
    MPI_Recv(poblacionNPM, NPM, typeIndividuo,0, 0, MPI_COMM_WORLD,&status);
}
```

*Ilustración 4. Actualizar población NPM en los hijos*

## 2.1. PARALELIZACIÓN DE LA FUNCIÓN SUAVIZAR

Para la paralelización de *suavizar* también se hizo uso de la comunicación síncrona con *MPI_Ssend* y *MPI_Recv*. En este caso la imagen inicial con la que trabajan los hilos ya la tenía cada uno de los procesos, ya que se pasaba por parámetro a la función por lo que nos ahorramos un paso de comunicación.

Inicialmente, se debía de crear una copia de la imagen para no alterar los datos mientras se va suavizando la matriz, para que este cálculo no fuera realizado por todos los hilos se ha optado por que el proceso master sea quien lo realice y lo comparta con los demás procesos.

Luego, debíamos de asignarle a cada uno de los procesos una imagen de la matriz, para ello hicimos uso de la variable *idProceso*, la cual indica el identificador del proceso, para asignar en que fila comienza a trabajar cada proceso. El número de filas a trabajar por cada proceso se obtiene dividiendo el número de filas entre los procesos existentes.

Finalmente, una vez cada uno ha modificado su sección, se deben juntar todas las partes en el proceso padre, el cual hará un *MPI_Recv* para obtenerlas de cada hijo. Un apunte importante es que hay que tratar el caso en el que sobre la última parte de la matriz en caso de que el número de filas de la imagen no sea divisible entre el número de procesos. Para ello hemos decidido que sea el master el que se encargue de esto ya que si no sería necesario un *MPI_SEND* y un *MPI_RECV* que enviaran una cantidad de datos distinta.

```
void suavizar(int ancho, int alto, RGB *imagen, int idProceso, int numProcesos, MPI_Datatype typeRGB)
{
    // imagen auxiliar
    RGB *imagen_auxiliar = (RGB *)malloc(alto * ancho * sizeof(RGB));

    // filas asignadas a cada proceso
    int processRows = floor(alto / numProcesos);

    // buffer en el que almacenaremos las secciones del array
    // el tamaño sera igual a las filas asignadas a cada proceso
    RGB *bufferRGB = (RGB *)malloc(processRows * ancho * sizeof(RGB));

    //el proceso master crea la copia y la pasa a los hijos
    if (idProceso == 0)
    {

        for (int i = 0; i < alto * ancho; i++)
        {
            imagen_auxiliar[i].r = imagen[i].r;
            imagen_auxiliar[i].g = imagen[i].g;
            imagen_auxiliar[i].b = imagen[i].b;
        }
        MPI_Bcast(imagen_auxiliar, ancho * alto, typeRGB, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Bcast(imagen_auxiliar, ancho * alto, typeRGB, 0, MPI_COMM_WORLD);
    }

    MPI_Status status;
```

*Ilustración 5. Función suavizar paralelizada parte 1*

```
MPI_Status status;

//Calculamos la ultima fila que le toca al proceso actual en funcion de su identificador
int ultimaFIla = (idProceso * processRows) + processRows;

//Cada uno suaviza su seccion concreta
for (int i = idProceso * processRows; i < ultimaFIla; i++)
{
    for (int j = 0; j < ancho; j++)
    {
        suavizar_pixel(i, j, ancho, alto, imagen_auxiliar, imagen);
    }
}

if (idProceso == 0)
{
    // Si no se han suavizado todas las filas las suavizara el padre
    int ultimaFilaSuavizada = ((numProcesos - 1) * processRows) + processRows;
    if (ultimaFilaSuavizada != alto)
    {
        for (int i = ultimaFilaSuavizada; i < alto; i++)
        {
            for (int j = 0; j < ancho; j++)
            {
                suavizar_pixel(i, j, ancho, alto, imagen_auxiliar, imagen);
            }
        }
    }

    // Se Reciben las distintas secciones de los hijos y se transladan a la imagen del master
    for (int i = 1; i < numProcesos; i++)
    {

        MPI_Recv(bufferRGB, processRows * ancho, typeRGB, i, 0, MPI_COMM_WORLD, &status);

        for (int j = 0; j < processRows * ancho; j++)
        {
            imagen[i * processRows * ancho + j] = bufferRGB[j];
        }
    }
}
else
{
    //enviamos la seccion que hayamos suavizado
    MPI_Ssend(&imagen[idProceso * processRows * ancho], processRows * ancho, typeRGB, 0, 0, MPI_COMM_WORLD);
}
```

*Ilustración 6. Función suavizar paralelizada parte 2*

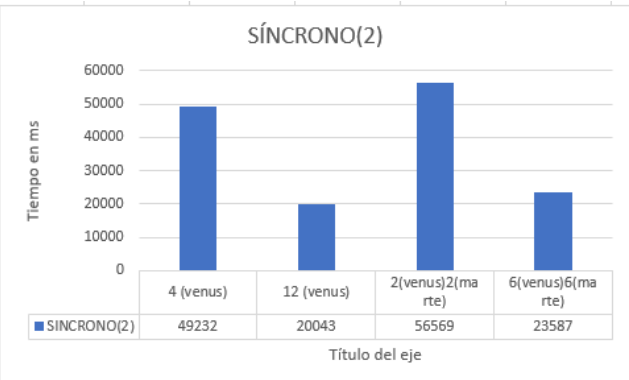


*Ilustración 7. Tiempos paralelización con comunicación síncrona con distintas configuraciones de ejecución*

# 3. PARALELIZACIÓN CON PACK Y UNPACK

Otra de las técnicas de comunicación que nos ha tocado implementar es la de ***Pack & Unpack***, ésta se basa en el **empaquetamiento** de estructuras o tipos de datos para después enviarlos y que en el cliente se **desempaqueten**.

El objetivo de este enfoque es la realización del algoritmo como se realizaba en el pasado **cuando los tipos derivados de MPI no existían**, sin embargo, para no añadir mucha complejidad al algoritmo se hará uso del tipo derivado *RGB*.

```
int position;
if (idProceso == 0) {
    // El padre coge el primer bloque de chunksize y envia a los hijos las demás partes
    for (int i = 0; i < chunkSize; i++)
    {
        islaPoblacion[i] = poblacion[i];
    }

    for (int proceso = 1; proceso < numProcesos; proceso++)
    {
        // Ponemos el position a 0 para sobreescribir la antigua informacion
        position = 0;

        // Transformamos los individuos en dos arrays para no usar tipos derivados
        for (int i = 0; i < chunkSize; i++)
        {
            for (int j = 0; j < ancho * alto; j++)
            {
                islaPoblacionRGB[i * (ancho * alto) + j] = poblacion[i+(proceso*chunkSize)].imagen[j];
            }
            islaPoblacionFitness[i] = poblacion[i+(proceso*chunkSize)].fitness;
        }

        // Enviamos los arrays generamos anteriormente
        MPI_Pack(islaPoblacionRGB, chunkSize*alto*ancho, typeRGB, bufferIndividuos, tam_buffer, &position, MPI_COMM_WORLD);
        MPI_Pack(islaPoblacionFitness, chunkSize, MPI_DOUBLE, bufferIndividuos, tam_buffer, &position, MPI_COMM_WORLD);
        // Enviamos el buffer con los arrays
        MPI_Send(bufferIndividuos, tam_buffer, MPI_PACKED, proceso, 0, MPI_COMM_WORLD);
    }
}
else {
```

*Ilustración 7. Envío de chunksize a los hijos*

El primer paso para seguir es el envío de los fragmentos de tamaño *chunkSize* de la población a los distintos procesos. Para ello tenemos primero que **descomponer** el tipo Individuo en los **tipos** que lo conforman, la imagen y el fitness, y agrupar estos datos en dos **arrays**, uno para las imágenes y otro para el fitness, para facilitar el manejo de **Pack** y **Unpack**.

Luego, empaquetaremos ambos en el *bufferIndividuos*, que tiene el tamaño justo para que en él se puedan almacenar *chunkSize* individuos y fitness. Con esto solo quedará enviarlo a través del *Send* a cada uno de los procesos.

Para que solo necesitemos un buffer del tamaño indicado anteriormente, en el bucle que va empaquetando los componentes de los individuos, deberemos de sobrescribir en cada iteración lo almacenado anteriormente, esto lo conseguimos estableciendo el índice "*position*" a 0 al principio de la iteración.

```
else {
    // Ponemos la posicion a 0
    position = 0;
    // Recibimos el buffer con los datos
    MPI_Recv(bufferIndividuos, tam_buffer, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
    // Desempaquetamos en el orden en el que hicimos PACK los dos arrays
    MPI_Unpack(bufferIndividuos, tam_buffer, &position, islaPoblacionRGB, chunkSize*alto*ancho, typeRGB, MPI_COMM_WORLD);
    MPI_Unpack(bufferIndividuos, tam_buffer, &position, islaPoblacionFitness, chunkSize, MPI_DOUBLE, MPI_COMM_WORLD);

    // Reconvertimos de los dos arrays recibidos al tipo individuo
    for (int i = 0; i < chunkSize; i++)
    {
        Individuo ind;
        for (int j = 0; j < ancho * alto; j++)
        {
            ind.imagen[j] = islaPoblacionRGB[j];
        }
        ind.fitness = islaPoblacionFitness[i];
        islaPoblacion[i] = ind;
    }
}
```

*Ilustración 8. Recepción de la población inicial de los procesos del proceso master*

Por otro lado, los distintos procesos recibirán ese buffer, del que tendrán que desempaquetar los arrays contenidos en el orden en el que fueron almacenados. También, será necesario que una vez desempaquetados se vuelvan a convertir en individuos, como podemos ver que se hace en **la *Ilustración 9***.

```
if (idProceso == 0)
{
    for (int i = 0; i < NEM; i++)
    {
        poblacionNEM[i] = islaPoblacionNEM[i];
    }
    for (int i = 1; i < numProcesos; i++)
    {
        //ponemos el position a 0 para sobreescribir la antigua informacion
        position = 0;
        //Recibimos el array con los datos
        MPI_Recv(bufferIndividuos, tam_buffer, MPI_PACKED, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        //Desempaquetamos del buffer
        MPI_Unpack(bufferIndividuos, tam_buffer, &position, poblacionRGBNEM, NEM*ancho*alto, typeRGB, MPI_COMM_WORLD);
        MPI_Unpack(bufferIndividuos, tam_buffer, &position, poblacionFitnessNEM, NEM, MPI_DOUBLE, MPI_COMM_WORLD);

        //reconvertimos de los arrays a Individuos
        for (int j = 0; j < NEM; j++)
        {
            Individuo ind;
            for (int k = 0; k < ancho * alto; k++)
            {
                ind.imagen[k] = poblacionRGBNEM[k+(j*ancho*alto)];
            }
            ind.fitness = poblacionFitnessNEM[j];
            poblacionNEM[NEM * i + j] = ind;
        }
    }
}
else
```

*Ilustración 9. Recepción NEM mejores individuos*

A continuación, tras la realización del algoritmo genético por parte de los distintos procesos, estos le enviarán al proceso padre sus mejores NEM individuos, como podemos ver en la *Ilustración 11*. Esto se hará de forma idéntica a la usada en la repartición inicial de la población usando el mismo buffer, pero usando un array auxiliar nuevo con tamaño NEM para facilitar la comprensión.

Estos Individuos descompuestos en partes serán recibidos por el padre y transformados a Individuos como podemos ver en *la Ilustración 10*, para poder reordenarlos por su fitness nuevamente.

```
else
{ // si eres hijo envias al padre

    //Transformamos los individuos en dos arrays para no usar tipos derivados
    for (int i = 0; i < NEM; i++)
    {
        for (int j = 0; j < ancho * alto; j++)
        {
            poblacionRGBNEM[i * (ancho * alto) + j] = islaPoblacion[i].imagen[j];
        }
        poblacionFitnessNEM[i] = islaPoblacion[i].fitness;
    }


    position = 0;
    //enviamos los arrays generamos anteriormente tras empaquetarlos
    MPI_Pack(poblacionRGBNEM, NEM*ancho*alto, typeRGB, bufferIndividuos,tam_buffer, &position, MPI_COMM_WORLD);
    MPI_Pack(poblacionFitnessNEM, NEM, MPI_DOUBLE, bufferIndividuos, tam_buffer, &position, MPI_COMM_WORLD);
    MPI_Send(bufferIndividuos, tam_buffer, MPI_PACKED, 0, 0, MPI_COMM_WORLD);
}
```

*Ilustración 10. Envío de los NEM mejores individuos*

Una vez obtenidos los NEM mejores individuos de cada proceso, estos individuos serán sustituidos por los peores de la población de nuestro proceso master. La población con estos nuevos individuos es reordenada en base a su fitness para así poder realizar el último paso del algoritmo, el envío de los NPM mejores individuos a todos los procesos.

Para el envío y recepción de estos individuos será necesario la descomposición y descomposición de los individuos, como hemos visto anteriormente, y podemos volver a observar en las *ilustraciones 12 y 13*.

```
// Compartimos los datos de la imagen leida a los hijos
if (idProceso == 0)
{
    // Sustituir los NEM*numprocesos peores individuos de poblacion por los mejores NEM individuos de cada proceso
    int index_poblacion_nem = 0;
    for (int i = tam_poblacion - 1; i > tam_poblacion - 1 - (numProcesos * NEM); i--)
    {
        poblacion[i] = poblacionNEM[index_poblacion_nem];
        index_poblacion_nem++;
    }

    // Ordenar individuos según la función de bondad (menor "fitness" --> más aptos)
    qsort(poblacion, tam_poblacion, sizeof(Individuo), comp_fitness);

    for (int i = 0; i < NPM; i++)
    {
        poblacionNPM[i] = poblacion[i];
    }

    // Enviamos poblacionNPM a todos los hijos
    // y convertimos de Individuo a dos arrays para no usar derivados
    for (int i = 0; i < NPM; i++)
    {
        for (int j = 0; j < ancho * alto; j++)
        {
            poblacionRGBNPM[i * (ancho * alto) + j] = poblacionNPM[i].imagen[j];
        }
        poblacionFitnessNPM[i] = poblacionNPM[i].fitness;
    }

    //Empaquetamos los dos arrays y hacemos broadcast ya que todon deben recibir el mismo dato
    position = 0;
    MPI_Pack(poblacionRGBNPM, NPM*ancho*alto, typeRGB, bufferIndividuos, tam_buffer, &position, MPI_COMM_WORLD);
    MPI_Pack(poblacionFitnessNPM, NPM, MPI_DOUBLE, bufferIndividuos, tam_buffer, &position, MPI_COMM_WORLD);
    MPI_Bcast(bufferIndividuos, tam_buffer, MPI_PACKED, 0, MPI_COMM_WORLD);
}
else
{
```

*Ilustración 11. Actualización población NEM en el padre*

```
else
{
    // Recibimos poblacionNPM
    MPI_Bcast(bufferIndividuos, tam_buffer, MPI_PACKED, 0, MPI_COMM_WORLD);
    position = 0;
    // Desempaquetamos
    MPI_Unpack(bufferIndividuos, tam_buffer, &position, poblacionRGBNPM, NPM*ancho*alto, typeRGB, MPI_COMM_WORLD);
    MPI_Unpack(bufferIndividuos, tam_buffer, &position, poblacionFitnessNPM, NPM, MPI_DOUBLE, MPI_COMM_WORLD);

    // Convertimos de Arrays a Individuo
    for (int j = 0; j < NPM; j++)
    {
        Individuo ind;
        for (int k = 0; k < ancho * alto; k++)
        {
            ind.imagen[k] = poblacionRGBNPM[k+(j*(ancho*alto))];
        }
        ind.fitness = poblacionFitnessNPM[j];
        poblacionNPM[j] = ind;
    }
}

qsort(islaPoblacion, chunkSize, sizeof(Individuo), comp_fitness);

int index_poblacion_npm = 0;
for (int i = chunkSize - 1; i > chunkSize - 1 - NPM; i--)
{
    islaPoblacion[i] = poblacionNPM[index_poblacion_npm];
    index_poblacion_npm++;
}
```

*Ilustración 12. Conversión de la población NPM recibida del proceso master.*

Para finalizar, nos encontramos con los datos obtenidos de la experimentación del algoritmo implementado. De los resultados obtenidos encontramos unos resultados muy mejorados,

habiéndose reducido el tiempo de ejecución de forma considerable y donde se ve claramente la gran mejora obtenida con el aumento de los procesos usados.

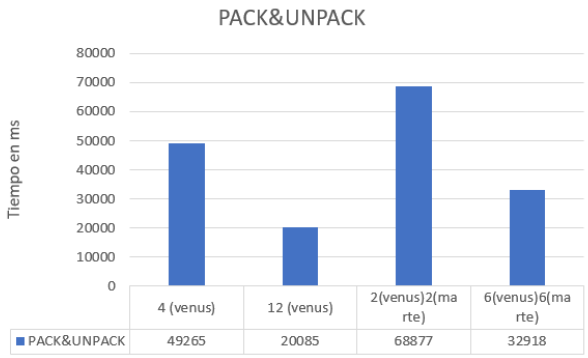


*Ilustración 13. Tiempos paralelización con comunicación síncrona usando Pack & Unpack sin tipo Individuo con distintas configuraciones de ejecución*

Como anexo de este apartado, hemos decidido mostrar los resultados obtenidos de este mismo algoritmo, pero en vez de realizando la descomposición del tipo Individuo usando un tipo derivado, lo cual facilitaba enormemente la cosa y obtenía resultados casi idénticos.

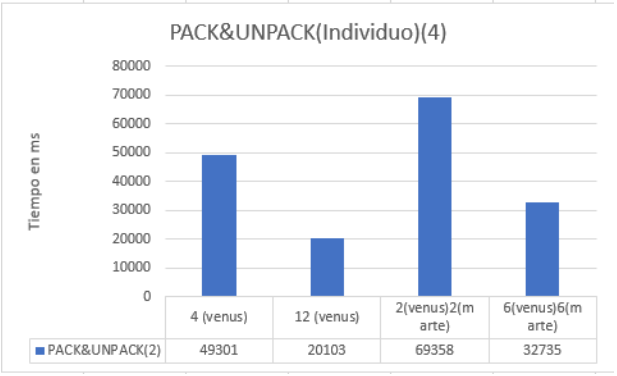


*Ilustración 14. Tiempos Pack & Unpack con tipo Individuo*

# 4. PARELELIZACIÓN CON COMUNICACIÓN ASÍNCRONA

Sobre el marco realizado en los anteriores ejercicios se quiere implementar una comunicación asíncrona entre los hilos padre e hijos lo que supone que ninguno de estos se quede esperando al receptor al momento de enviar datos.

Para este apartado necesitamos declararlas variables *MPI_Status* y *MPI_Request* para poder realizar los *Isend* e *Irecv* correctamente.

Lo primero que se realiza es la inicialización del trozo de población que el hilo padre va a tratar (*islapoblacion*) ya que no vamos a enviárselo a el mismo. Posteriormente se envían los diferentes segmentos de memoria que cada hilo hijo tendrá que procesar.

En caso de ser el hijo recibiremos en la variable anteriormente declarada *islaPoblacion* de tamaño *chunksize* y se espera a que estos datos lleguen.

```
// El padre suministra subpoblaciones (de chunkSize) a los hijos
MPI_Barrier(MPI_COMM_WORLD);

MPI_Request send_request, recv_request;
MPI_Status status;

if(idProceso==0){
    // El padre coge el primer bloque de chunksize y envia a los hijos las demás partes
    for(int i=0;i<chunkSize;i++){
        islaPoblacion[i]=poblacion[i];
    }
    for(int i=1;i<numProcesos;i++){
        MPI_Isend(&poblacion[i*chunkSize], chunkSize, typeIndividuo, i, 0, MPI_COMM_WORLD,&send_request);
    }
}
else{
    MPI_Irecv(islaPoblacion, chunkSize, typeIndividuo,0, 0, MPI_COMM_WORLD,&recv_request);
    MPI_Wait(&recv_request, &status);
}
```

*Ilustración 15. Envío de ChunkSize individuos desde el proceso master a los demás procesos*

Posteriormente cada hilo con ***qsort*** organiza según el fitness los mejores individuos (NEM individuos) y se traspasan a un array llamado *islaPoblacionNEM*.

El hilo padre actualiza con sus datos procesados el array *poblacionNEM* donde convergerán todos los datos enviados por los hijos.

```
// Si eres el padre -> guardas tu parte en el array donde se junta todo y recibes de cada hijo
for(int i=0;i<NEM;i++){
    islaPoblacionNEM[i]=islaPoblacion[i];
}

if(idProceso==0){
    for(int i=0;i<NEM;i++){
        poblacionNEM[i]=islaPoblacionNEM[i];
    }

    for(int i=1;i<numProcesos;i++){
        MPI_Irecv(islaPoblacionNEM, NEM,typeIndividuo,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&recv_request);
        MPI_Wait(&recv_request, &status);
        for(int j=0;j<NEM;j++){
            poblacionNEM[NEM*i+j]=islaPoblacionNEM[j];
        }
    }
}
else{   // si eres hijo envias al padre
    MPI_Isend(islaPoblacionNEM, NEM, typeIndividuo, 0, 0, MPI_COMM_WORLD,&send_request);
}
```

*Ilustración 16. Sustitución de los NEM mejores individuos por los peores de la población del master*

A continuación enviamos la *poblaciónNPM* a todos los hijos para que actualicen su fragmento islaPoblación que será procesado en la siguiente iteración.

```
if (idProceso == 0){
    // Sustituir los NEM*numprocesos peores individuos de poblacion por los mejores NEM individuos de cada proceso
    int index_poblacion_nem = 0;

    for (int i = tam_poblacion - 1; i > tam_poblacion - 1 - (numProcesos * NEM); i--)
    {
        poblacion[i] = poblacionNEM[index_poblacion_nem];
        index_poblacion_nem++;
    }

    // Ordenar individuos según la función de bondad (menor "fitness" --> más aptos)
    qsort(poblacion, tam_poblacion, sizeof(Individuo), comp_fitness);

    for(int i=0;i<NPM;i++){
        poblacionNPM[i]=poblacion[i];
    }

}
```

*Ilustración 17. Obtención de los NPM mejores individuos*

Como apunte interesante nos gustaría comentar un error que tuvimos en este ejercicio, que pasaba desapercibido, pero causaba **segmentation fault**. Este error se debía a que habíamos colocado el *MPI_Barrier* después de realizar algunos "free", lo que resultaba en ejecuciones que a veces eran correctas y otras que no, ya que la liberación de algunas estructuras de datos antes de que finalizará el uso de estos por algunos procesos causaba este **segmentation fault.**

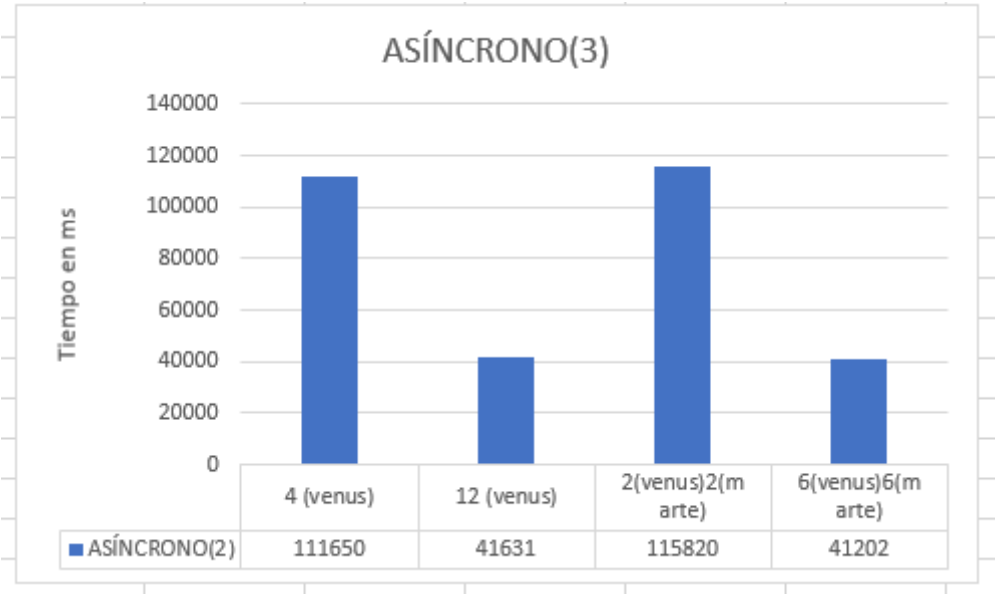


*Ilustración 18. Tiempo ejecución asíncrono*

# 5. PARALELIZACIÓN CON COMUNICACIÓN COLECTIVA

En este apartado se tiene que usar comunicación colectiva entre los diferentes hilos para lograr realizar el algoritmo genético. Primero inicializamos los diferentes arrays que se declaran para cada hilo y esperamos en una barrera a que todos acaben. Posteriormente se realiza un *MPI_Scatter* de manera que se reparte a cada hilo una población de tamaño *chunksize* que usará para aplicarle el algoritmo genético.

```
// Reservamos memoria para porcion de subpoblacion de cada proceso (OJO: tambien se crea en el padre)
islaPoblacion = (Individuo *) malloc(chunkSize*sizeof(Individuo));
assert(islaPoblacion);
islaPoblacionNEM = (Individuo *) malloc(NEM*sizeof(Individuo));
assert(islaPoblacionNEM);
poblacionNPM = (Individuo *) malloc(NPM*sizeof(Individuo));
assert(poblacionNPM);

// El padre suministra subpoblaciones (de chunkSize) a los hijos
MPI_Barrier(MPI_COMM_WORLD);
MPI_Scatter(poblacion, chunkSize, typeIndividuo, islaPoblacion, chunkSize, typeIndividuo, 0, MPI_COMM_WORLD);
```

*Ilustración 19. Envío de chunksize a los hijos*

A continuación, se realiza el *MPI_Gather* para recoger el dicho array y se ordena la población dependiendo del fitness.

```
MPI_Gather(islaPoblacionNEM, NEM, typeIndividuo, poblacionNEM, NEM, typeIndividuo, 0, MPI_COMM_WORLD);

if (idProceso == 0){
    // Sustituir los NEM*numprocesos peores individuos de poblacion por los mejores NEM individuos de cada proceso
    int index_poblacion_nem = 0;
    for (int i = tam_poblacion - 1; i > tam_poblacion - 1 - (numProcesos * NEM); i--)
    {
        poblacion[i] = poblacionNEM[index_poblacion_nem];
        index_poblacion_nem++;
    }

    // Ordenar individuos según la función de bondad (menor "fitness" --> más aptos)
    qsort(poblacion, tam_poblacion, sizeof(Individuo), comp_fitness);

    for(int i=0;i<NPM;i++){
        poblacionNPM[i]=poblacion[i];
    }
}
```

*Ilustración 20. envió al padre de los mejores individuos*

Finalmente se reparte a todos los individuos mediante un broadcast la nueva población con los peores valores sustituidos por los mejores NEM individuos seleccionados de lo que han procesado los hijos. Se reparten arrays de tamaño *chunksize* que se ordenan según el fitness y el proceso se vuelve a realizar.

```
// Compartimos los datos de la imagen leida a los hijos
MPI_Bcast(poblacionNPM,NPM,typeIndividuo,0,MPI_COMM_WORLD);

qsort(islaPoblacion,chunkSize,sizeof(Individuo),comp_fitness);

int index_poblacion_npm = 0;

for (int i = chunkSize - 1; i > chunkSize - 1 - NPM; i--)
{
    islaPoblacion[i] = poblacionNPM[index_poblacion_npm];
    index_poblacion_npm++;
}
```

*Ilustración 21. Envío poblacionNPM a los hijos*

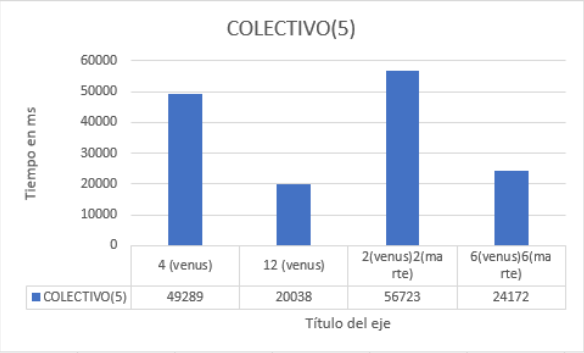


*Ilustración 22. Tiempos paralelización con comunicación colectiva con distintas configuraciones de ejecución*

# 6. REPARTO DE TAREAS ENTRE LOS MIEMBROS DEL GRUPO

- Alberto Gálvez Gálvez:
  - Desarrollo del esquema de paralelización.
  - Desarrollo del esqueleto MPI usado en el main.c y en ga.c común a todos los ejercicios.
  - Desarrollo de la paralelización con comunicación colectiva.
  - Ayuda con los ejercicios del resto de compañeros.
  - Realización de tests y escrito memoria.
- Antonio Martínez Fernández:
  - Desarrollo de la paralelización síncrona sin incluir la función suavizar.
  - Desarrollo de la paralelización asíncrona.
  - Desarrollo de la paralelización usando pack y unpack.
  - Realización de tests y escrito memoria.
- Mario Rodríguez Béjar:
  - Desarrollo de la paralelización síncrona de suavizar.
  - Desarrollo de la paralelización usando pack y unpack.
  - Realización de tests y escrito memoria.

Cabe destacar el uso de la herramienta de control de versiones *git* para un trabajo en equipo cohesionado, así como robusto para responder a los distintos errores o cambios repentinos que puedan producirse. El repositorio que hemos creado se encuentra alojado en *github.com*.