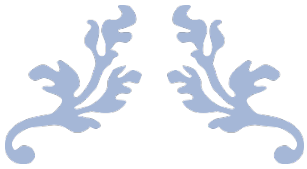


PROYECTO HPC: MPI

PSD 2019

Margarita Geleta – Oriol Narváez Serrano

MPI : reduce_mpi **Versión 1**

Vamos a analizar el programa **`reduce_mpi`** para ver que está haciendo y cómo está implementado. Iremos paso por paso:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MASTER 0
int *data; // puntero a donde se encuentran los datos.
int total_elems; // numero total de elementos.
int type_size;  // bytes tipo de datos de los elementos.
```

Aquí se han declarado variables globales: `*data` el puntero que apunta donde se encuentran los datos de la matriz, `total_elems` el número total de elementos en la matriz y `type_size` el tamaño en bytes del tipo de datos de los elementos de la matriz. También, hemos definido que el rango del proceso raíz (*Master*) es 0.

A continuación, entramos en `main`, donde, primero de todo se declaran las variables que contendrán el tamaño del comunicador (`my_size`), el rango del proceso que está ejecutando (`my_rank`) y el tamaño de trozo de la matriz que se cogerá cada proceso (`chunk`):

```
int main (int argc, char *argv[]){
        MPI_Status status;    // para guardar info sobre operaciones de recepcion.
        int my_rank, my_size; // tamaño del comunicador y rango del proceso que lo llama.
        int rc = -1;
        int chunk;
        int i;
```

Declaramos en `type_size` el tamaño de un entero. Actualmente, en sistemas de *64-bit* corresponde a 8 Bytes de memoria. Luego, comprobamos que hayamos pasado el fichero de la matriz como argumento a la aplicación. Si es que no, el proceso termina.

```
        type_size = sizeof(int); // tamaño de un entero (8 Bytes en sist 64-bit sys).
        if (argc != 2){ // si no le pasamos la matriz, nos dice como usarlo.
                printf("usage: %s file_name\n",argv[0]); // ./programa fichero
                exit(1);
        }
```

Pero como se espera, si pasamos el fichero, continuamos la ejecución llegando a la sección donde inicializamos el entorno MPI (observe al principio `#include "mpi.h"`). `MPI_COMM_WORLD` es el comunicador que incluye todos los procesos del programa. Especificamos los rangos y el tamaño del comunicador.

```
        MPI_Init(&argc,&argv); // inicializamos el entorno MPI.
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
       MPI_Comm_size(MPI_COMM_WORLD,&my_size);
```

Para el proceso raíz, hay las siguientes instrucciones; con la función `file_size` calcula el número de elementos totales de la matriz y lo guarda en la variable `total_elems`. En `data` asignamos el tamaño de la matriz en bytes (`total_elems*sizeof(int)`), que también se podría hacer como `total_elems*type_size`.

```
        if (my_rank == MASTER) // (rango 0)
        {
                printf("Using %s as input\n",argv[1]); // dice que utiliza el fichero como input.
                total_elems = file_size(argv[1],sizeof(int)); // # de elems matriz.
                int elems_process;
                if (total_elems < 0){ // no puedes tener un numero negativo de elementos
                        printf("Invalid number of elements\n"); // ups
                        MPI_Abort(MPI_COMM_WORLD, rc); // apagamos el entorno MPI.
                }

                data = (int *)malloc(total_elems*sizeof(int));
```

Nos aseguramos de que tengamos el buffer `data`. Suponiendo que todo ha ido bien, comprobamos que hayamos podido leer toda la matriz. La función nos permite leer un número especificado de elementos a partir de una posición dada. Le pasamos como parámetro la posición inicial 0, y número de elementos `total_elems` (= todos). Hacemos una conversión de tipo de `data` con `(void *)` porque la función en si puede trabajar con un buffer de cualquier tipo, no necesariamente enteros y el parámetro de entrada está definido con `void`.

```
if (data == NULL){
        printf("Error in malloc\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
}

rc = read_from_pos(argv[1], 0, total_elems, type_size, (void *)data);
if (rc<0){ // comprobamos que hayamos podido leer toda la matriz.
        printf("Error reading file\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
}
```

Ha llegado la hora para enviar los trozos de la matriz entre los procesos esclavos. Primero de todo, se calcula el tamaño de cada trozo (`chunk`). Es tan fácil como calcular el cociente entre el número total de elementos y el número de los procesos (`my_size`).

```
chunk = total_elems/my_size; // calculamos el tamano del mensaje.
```

Para cada proceso esclavo (por eso empezamos por `i = 1`), en `elems_process` guardamos el número de elementos que hay en un trozo (`chunk`) que se le enviará:

```
for(i = 1; i < my_size; i++){
        if (i == (my_size-1)){
                elems_process = total_elems-((my_size-1)*chunk); // =el chunk que
                // queda para enviar, el último.
        }else{
                elems_process = chunk;
        }
```

Y el proceso raíz le envía el número de elementos que hay en el trozo, de tipo *entero* a través del comunicador `MPI_COMM_WORLD`. Si surge algún error, abortamos el entorno MPI.

```
        if (MPI_Send(&elems_process, 1, MPI_INT, i, 0,
              MPI_COMM_WORLD)!=MPI_SUCCESS){
                 printf("Error sending chunk\n");
                 MPI_Abort(MPI_COMM_WORLD, rc);
        }
```

Ponemos el puntero a la posición `chunk*proceso` correspondiente en el buffer `data` y cogemos a continuación tantos elementos como hay en un trozo por proceso y enviamos al proceso esclavo el trozo de la matriz que le corresponde:

```
        if (MPI_Send(data+(chunk*i), elems_process, MPI_INT, i, 0,
              MPI_COMM_WORLD)!=MPI_SUCCESS){
                 printf("Error sending data\n");
                 MPI_Abort(MPI_COMM_WORLD, rc);
        }
}
```

Para el (los) proceso(s) esclavo(s), se recibe el número de elementos de su trozo de la matriz y propiamente el trozo de la matriz que le corresponde de tipo entero del proceso raíz con *tag* 0 a través del comunicador `MPI_COMM_WORLD`. Si surge algún error, se aborta.

```
}else{
        if (MPI_Recv(&chunk, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE)!=MPI_SUCCESS){
                printf("Error receiving chunk size\n");
                MPI_Abort(MPI_COMM_WORLD, rc);
        }
        // asignamos el tamaño en bytes del chunk.
        data = (int *)malloc(chunk*sizeof(int));
        // guardamos en el buffer data el chunk enviado por root.
        if (MPI_Recv(data, chunk, MPI_INT, 0, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE)!=MPI_SUCCESS){
```

```
                        printf("Error receiving data\n");
                        MPI_Abort(MPI_COMM_WORLD, rc);
                }
        }
```

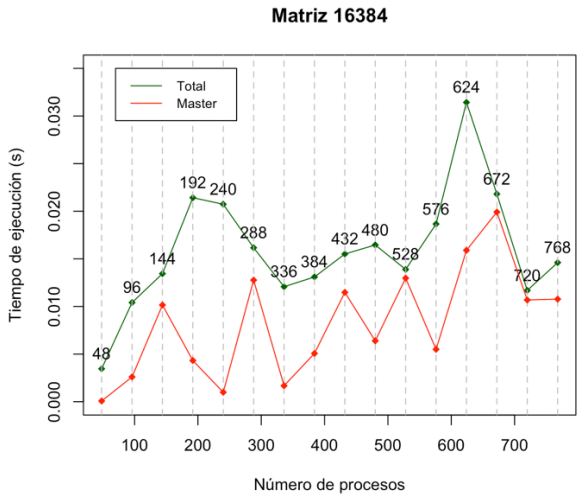
En este punto de la ejecución, tenemos los datos. A continuación, los esclavos enviaran cuantos elementos han recibido del proceso raíz a su emisor, y este sumará los elementos que le habrán enviado los esclavos y lo guardará en la variable `reduction`. Si `reduction` coincide con el número inicial de elementos de la matriz, significará que no se ha perdido nada durante la comunicación.

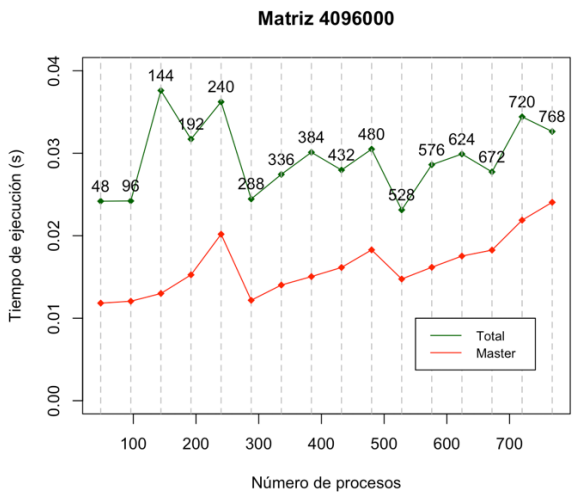
```
        int start = 0;
        int end = chunk;
        int reduction = 0;
        for (i = start; i < end; i++){
                reduction += data[i]; // en reduction guardamos los datos que tiene cada proceso
                // de su parte (de su propio buffer data, del tamaño de chunk).
        }
        printf("Reduction for rank %d is %d\n",my_rank,reduction);
        /* Send results */
        if (my_rank == MASTER){
                int local_red;
                for (i = 1; i < my_size; i++){
                        // root recibe de cada esclavo lo que ha recibido.
                        if (MPI_Recv(&local_red, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                              MPI_STATUS_IGNORE)!=MPI_SUCCESS){
                                 printf("Error receiving local_red from rank %d\n",i);
                                 MPI_Abort(MPI_COMM_WORLD, rc);
                        }
                        printf("Reduction received from rank %d is %d\n", i, local_red);
                        reduction += local_red;
                }
                printf("Total reduction %d\n",reduction); // num de elems enviado por esclavos.

        }else{ // los esclavos envian al root que es lo que han recibido de él:
                if (MPI_Send(&reduction, 1, MPI_INT, 0, 0, MPI_COMM_WORLD)!=MPI_SUCCESS){
                        printf("Error sending reduction from %d to master\n", my_rank);
                        MPI_Abort(MPI_COMM_WORLD, rc);
                }
        }

        MPI_Finalize();
}
```

Para estudiar a fondo el programa, creamos dos matrices, de tamaños 16384 y 4096000, solamente con valores '1'. Para calcular el tiempo total de ejecución y el coste de enviar los datos del master a los procesos esclavos, utilizaremos la función específica de MPI `MPI_Wtime(void)`.

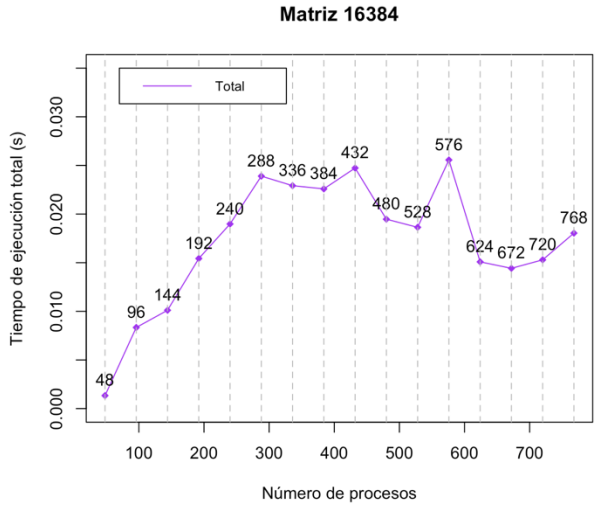
- Para el tiempo de ejecución total, hemos calculado el periodo de tiempo desde la llamada a `MPI_Init(&argc,&argv)` hasta el `printf` de `reduction`.
- Para el tiempo de envío del proceso raíz a los esclavos, hemos calculado el tiempo comprendido en el bucle `for(i = 1; i < my_size; i++)` que envía datos a los procesos esclavos.

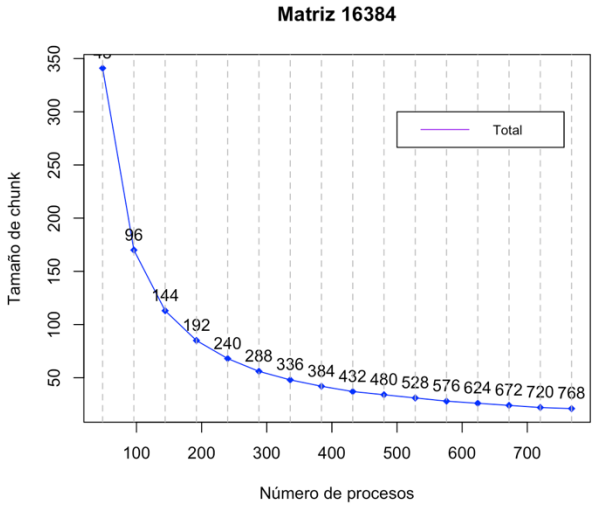
Las gráficas se pueden observar a continuación:

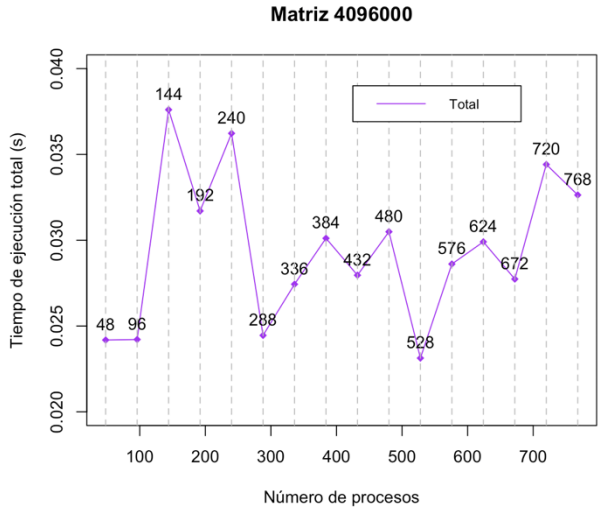


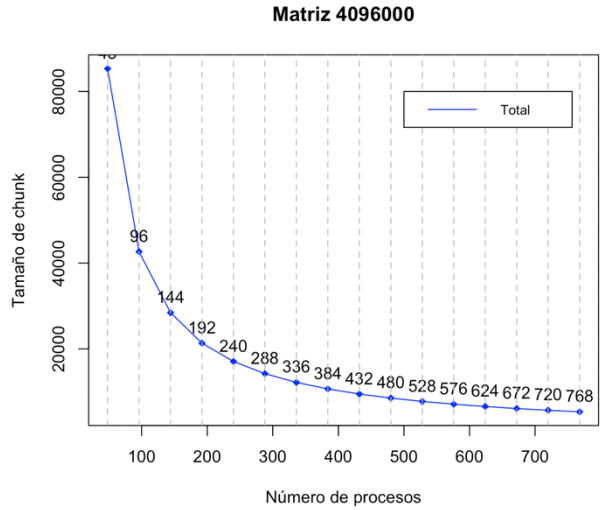


Como se puede observar en las gráficas, al paralelizar el tiempo de ejecución aumenta. Esto puede ser debido a varios factores, uno de los cuáles podría ser el *overhead* de comunicación (no estamos utilizando *comunicación asíncrona*, por ejemplo).

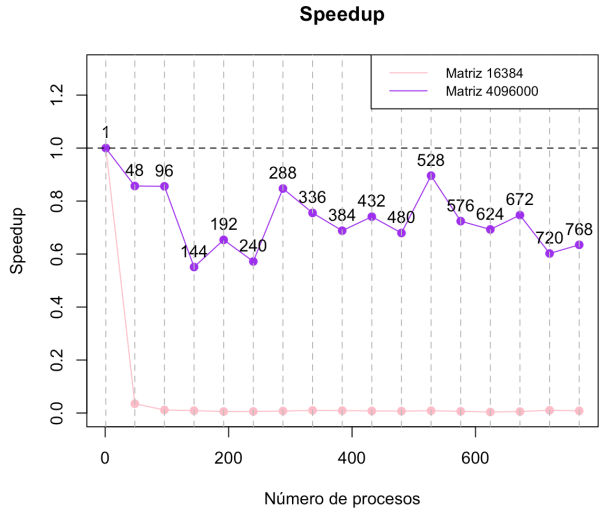


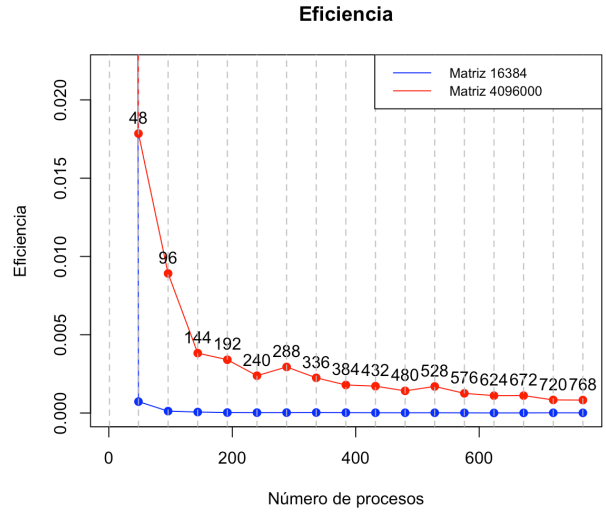






Teniendo como referencia el tiempo de ejecución total y de envío con 1 procesador, generamos gráficas con el *speedup* y la *eficiencia* de cada caso:





La eficiencia decae rápidamente y por los resultados que vemos, con tal paralelización es mejor no paralelizar y dejarlo en secuencial. A continuación, vamos a modificar la forma en que se distribuyen los trozos entre los procesos y miraremos el impacto de esta mejora.

MPI : reduce_mpi_local **Versión 2**

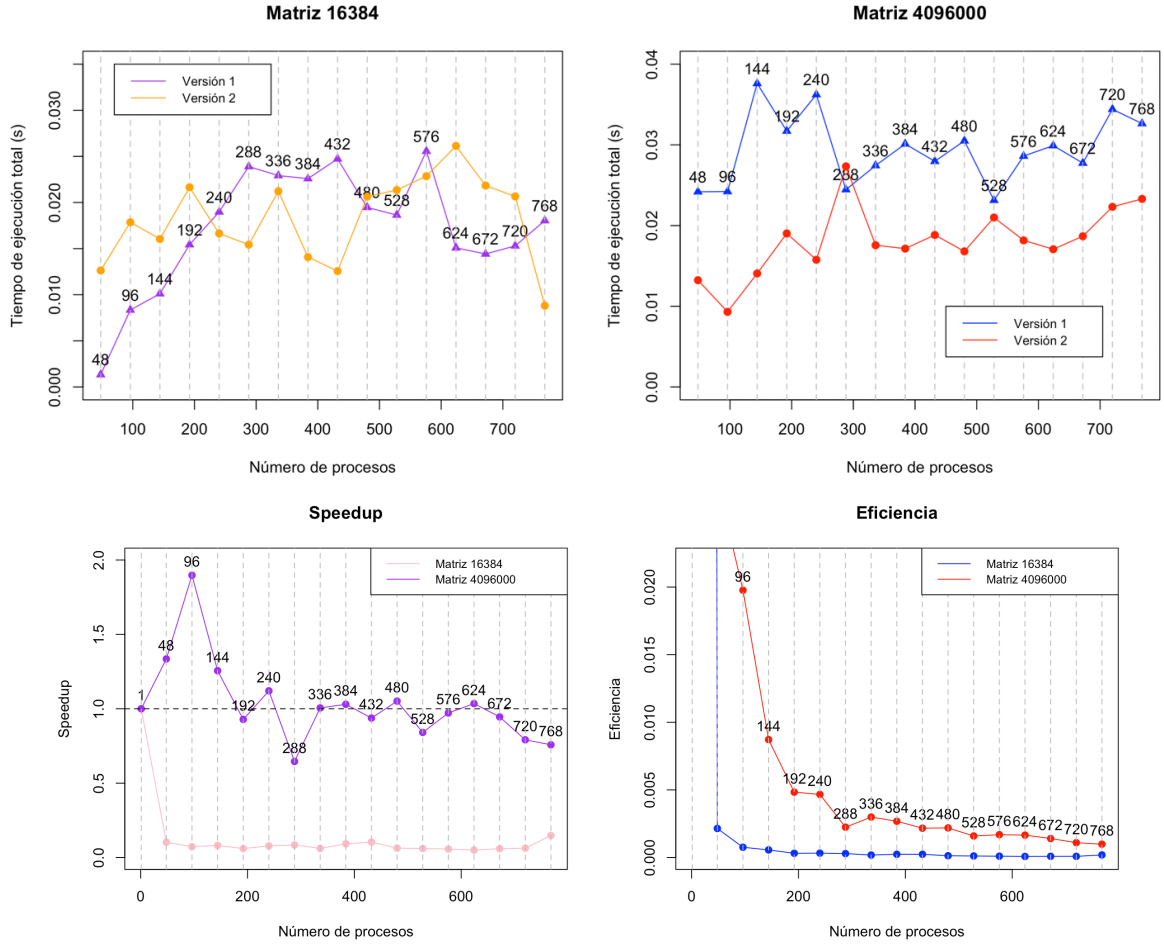
Ahora vamos a modificar el programa anterior de forma que cada proceso se calcule individualmente el trozo de fichero que le toca procesar y haga la lectura de sus propios datos sin que el *Master* tenga que enviárselo a cada uno. Observaciones:

- Cada proceso debe reservar el vector de datos que necesita, no más **(1)**.
- Cada proceso debe leer solo la parte del fichero que va a procesar **(2)**.

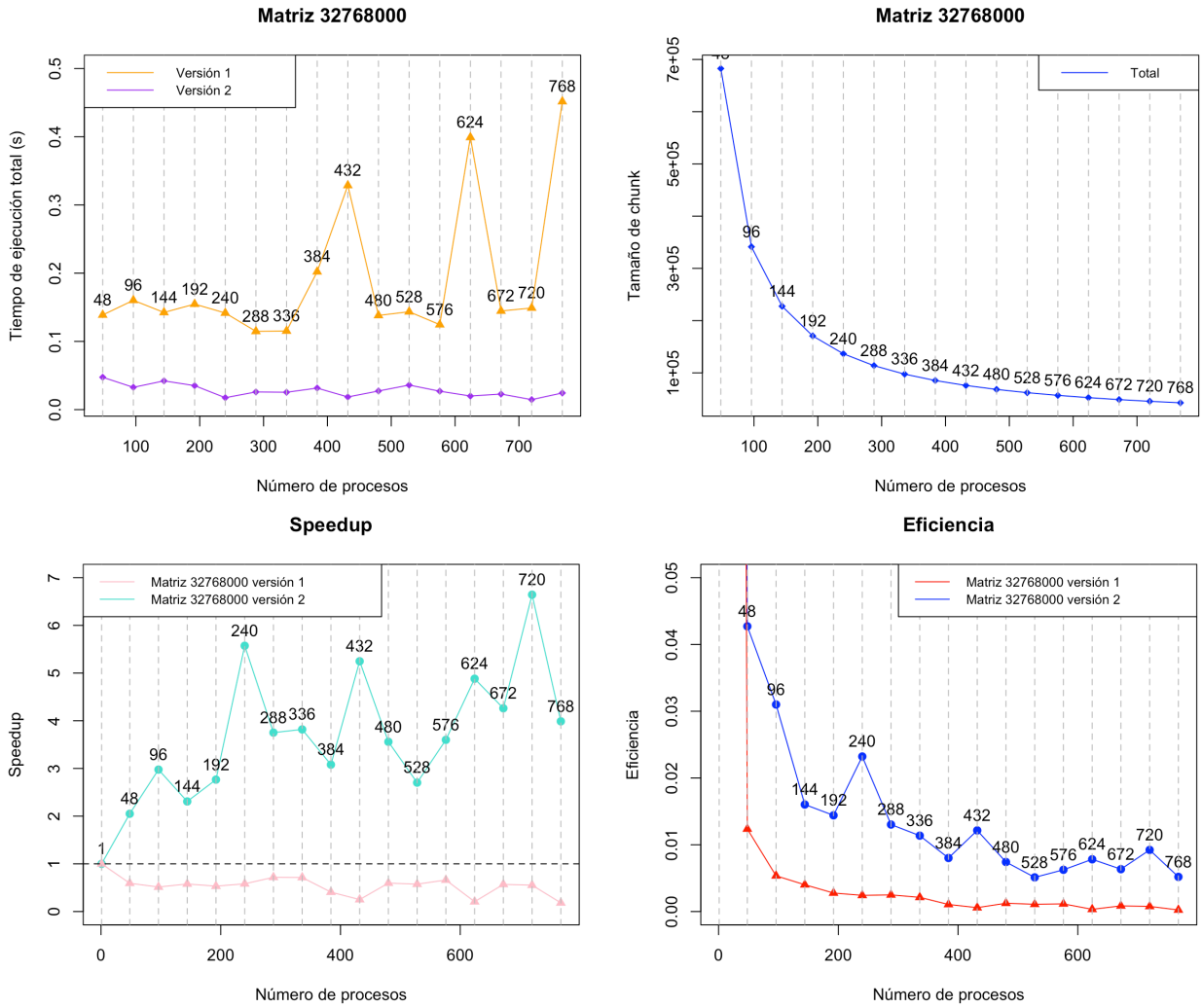
Este es el único trozo de código modificado, desde la inicialización del entorno MPI hasta la declaración de la variable `reduction`:

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
chunk = total_elems/my_size; // calculamos el tamano del trozo.
MPI_Barrier(MPI_COMM_WORLD);
int elems_left;
if(my_rank == my_size - 1)
{
        elems_left = total_elems-((my_size-1)*chunk);
        data = (int *)malloc(elems_left*sizeof(int));  (1)
}else{
        elems_left = chunk;
        data  = (int *)malloc(elems_left*sizeof(int)); (1)
}
if(data == NULL){
          printf("Error in malloc\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
}
 rc = read_from_pos(argv[1], chunk*(my_rank), elems_left, type_size, (void
                                                                  *)data); (2)
 if(rc < 0){
        printf("Error reading file\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
}
int start = 0;
int end = elems_left;
```

Usando las matrices anteriores obtenemos los siguientes resultados:



Hay una clara mejora respecto la primera versión del código, que se nota más en la matriz más grande. Vamos a usar una matriz aún más grande para ver mejor el impacto que tiene esta modificación: utilizaremos una matriz de tamaño 4096000*x*8 = 32768000.



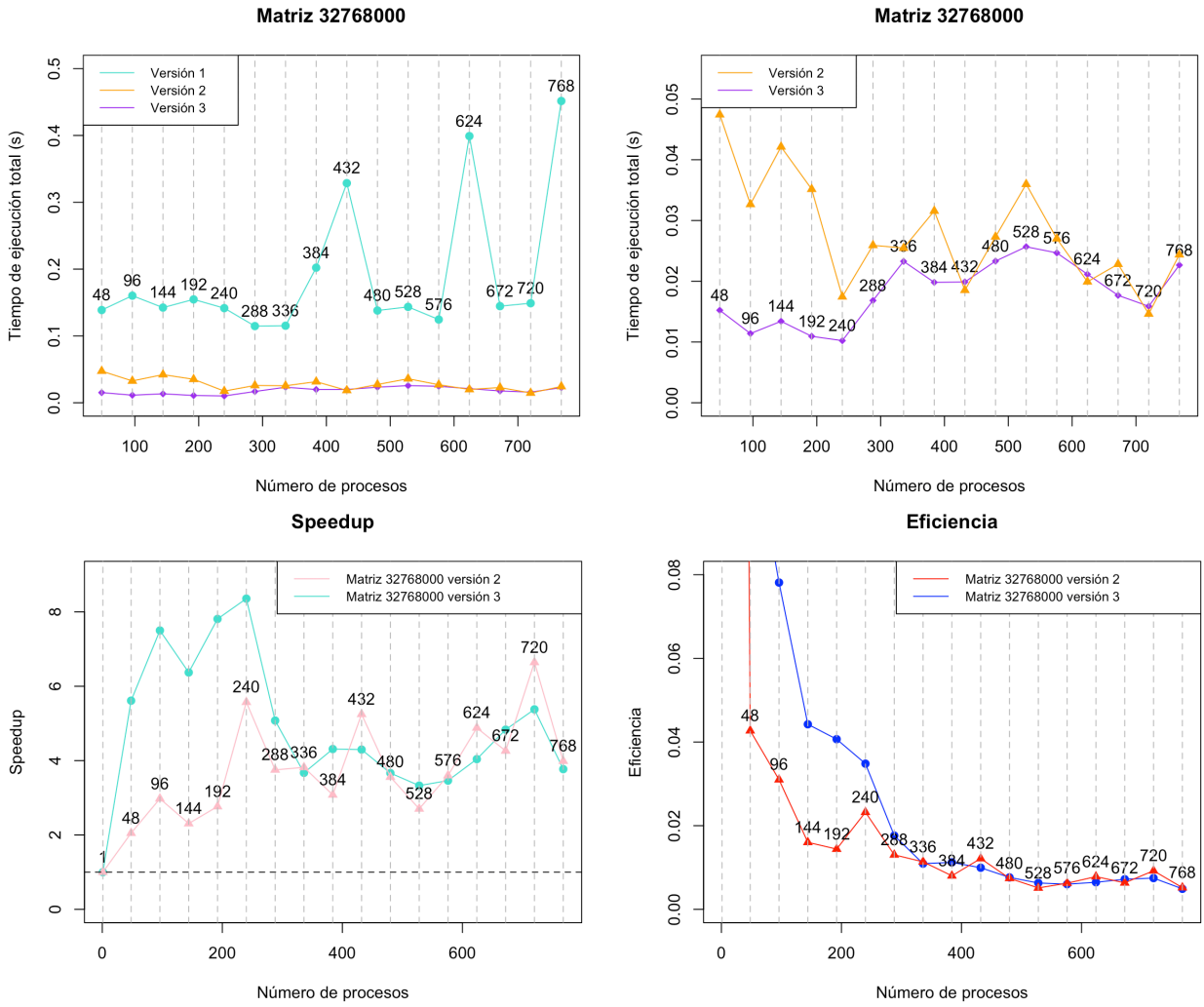
Podemos concluir que la considerable mejora se hace más visible al utilizar matrices de tamaño más grande. El *speedup* y la *eficiencia* aumentan.

MPI : reduce_mpi_local_withred **Versión 3**

Vamos a sustituir el envío final de los valores por la función `MPI_Reduce`, para reducir los valores de cada proceso esclavo en el único proceso raíz. Para sumar los valores, utilizaremos la operación `MPI_SUM`. Solo tenemos que modificar la parte final del código:

```
int start = 0;
int end = elems_left;
int local_sum = 0;
for (i = start; i < end; i++){
        local_sum += data[i];
}
//printf("Proc %d has local_sum = %d\n", my_rank, local_sum);
int global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
if(my_rank == MASTER)
{
        t2 = MPI_Wtime();
        printf("Total reduction %d\n",global_sum);
        printf("Total exec time: %1.9f\n", t2 - t1);
}
MPI_Finalize();
```

Comparamos las versiones del código con la matriz más grande:



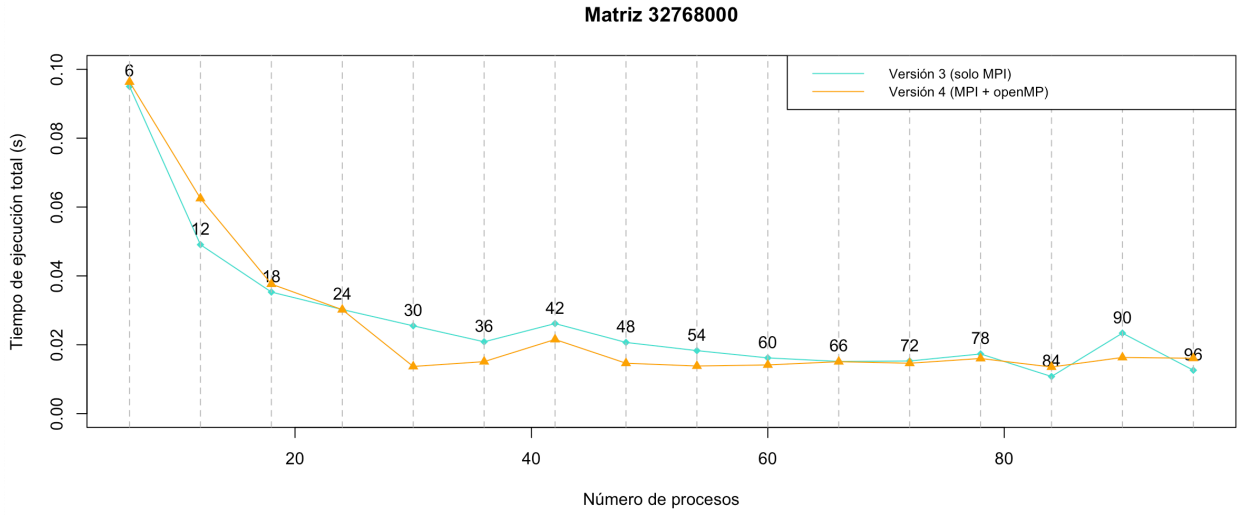
El *speedup* de la segunda versión es más grande con un número más pequeño de procesos, pero luego decae. Sin embargo, el *speedup* de la tercera versión tiene una tendencia creciente y llega un momento donde la eficiencia de la versión 3 es levemente superior a la de la versión 2.

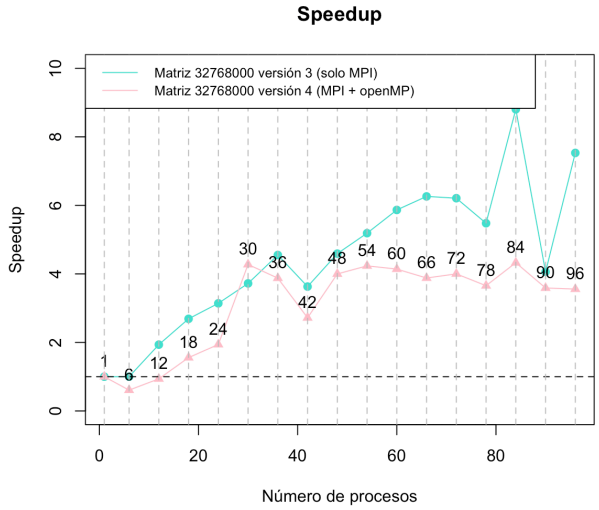
MPI : reduce_mpi_openmp **Versión 4**

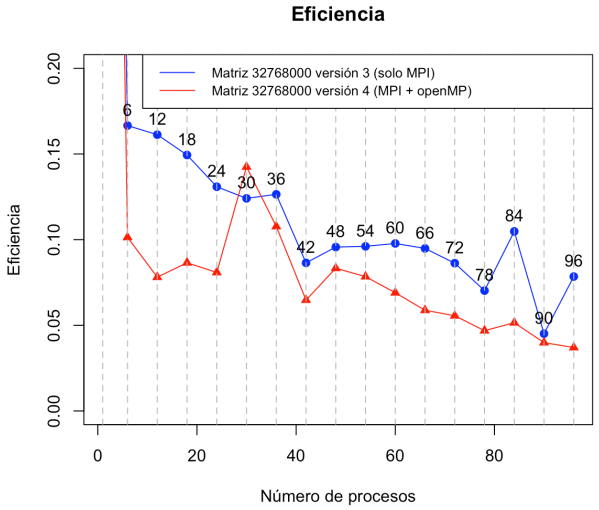
Vamos a modificar el programa anterior para añadir *OpenMP* para hacer el cálculo de la reducción en paralelo. Solamente tenemos que modificar la sección del bucle, donde se calcula la suma de los elementos del buffer de cada uno de los procesos esclavos:

```
int start = 0;
int end = elems_left;
int local_sum = 0;
#pragma omp parallel shared(data) reduction(+:local_sum)
{
        #pragma omp for
        for (i = start; i < end; i++){
              local_sum += data[i];
        }
}
int global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

En los gráficos a continuación observamos que, la versión con *OpenMP* va un poco más rápido que la versión sin. Sin embargo, el *speedup* y la *eficiencia* son mucho mejores en la versión sin *OpenMP*.







Conclusión: la mejor versión

Una vez analizados los gráficos, podemos concluir que los mejores resultados que se han obtenido han sido con la tercera versión del código, es decir, la versión `reduce_mpi_local_withred.c`.