Práctica 2

Implementación distribuida de un algoritmo paralelo de datos

Jose Luis Gallego Peña

1. Introducción

En esta práctica se aborda la implementación de un algoritmo de multiplicación matriz-vector, primero de manera secuencial para ilustrar la forma básica del algoritmo, y luego de forma paralela y distribuida usando OpenMPI con dos versiones distintas en la asignación de datos a procesos. Se aportan todos los ficheros necesarios para la compilación, ejecución y recolección de datos y gráficas de los tres algoritmos que se presentan.

Los algoritmos se encuentran en los ficheros **matrizvector_1d.cc** y **matrizvector_2d.cc**, que reciben como parámetro de ejecución el tamaño del problema N. En ambos el proceso 0 genera y reparte los datos de entrada, además de cálcular de forma secuencial el algoritmo, para luego volver a ejecutarlo pero mediante el paralelismo de MPI. Se genera un vector llamado **comprobacion** en el cual se almacena el resultado secuencial para compararlo con el paralelo y comprobar que es correcto, mostrando al final el número de errores que se han cometido.

Los códigos incluyen una variable booleana llamada **modoGraficas** que por defecto está a 0 (false), y por tanto al ejecutar de forma normal los programas se muestra por pantalla información sobre los datos que se muestran. Cuando la variable está a 1 (true) por haberlo indicado como segundo parámetro opcional al programa, sólo se obtienen los datos numéricos en el formato adecuado para poder representarlos. Este es el modo que usa el script **ejecuciones.sh** que es el que realiza todo el proceso de compilación, ejecución, recolección de datos y generación de gráficas. Los tiempos se han medido usando **MPI_Wtime**.

Para su ejecución, recolección de resultados y representación gráfica se usa el script **ejecuciones.sh** que borra anteriores resultados, compila el programa y lo ejecuta varias veces para distintos tamaños de problema N (300, 600, 900, 1200 y 1400). Por último llama al script **grafica.gp**, que coge los datos de los archivos de resultados y los representa en una gráfica de tiempos y otra de ganancias para cada uno de los dos algoritmos usando la utilidad gnuplot, exportadas a un archivo PNG. Todos los archivos de datos y las gráficas se encuentran en la **carpeta resultados**.

Esta práctica está programada y ejecutada en Arch Linux x86_64 con procesador Intel Core i5-8250U @ 1.60GHz de 8 núcleos. Se usa OpenMPI en su versión 4.0.5.

2. Descomposición unidimensional (por bloques de filas)

El programa, al recibir el tamaño del problema N, comprueba si este es un múltiplo del número de procesos P y además es mayor que este, ya que esta es una de las condiciones del algoritmo. Si no lo es, se transforma n a un valor válido que sea múltiplo de P para poder continuar con la ejecución.

En este algoritmo, cada proceso puede calcular más de un elemento del vector resultado y, por tanto almacena localmente un bloque de varias filas para multiplicar en un vector llamado **miBloque** cuyo tamaño es el número de elementos de la matriz con el que opera, N^2/P. El resultado parcial que calcula ahora es un vector llamado **subFinal**, cuyo tamaño es el número de filas que va a calcular cada proceso (N/P filas, que consiste en N^2/P elementos de la matriz).

El proceso 0 genera aleatoriamente los datos para el vector y la matriz. Reparte N/P filas de la matriz a cada proceso de forma equitativa, y luego comparte una copia exacta de todo el vector a todos los procesos. Se coloca una barrera para que todos los procesos comiencen a la vez, y entonces ejecutan su parte del algoritmo, en la cual multiplican más de un resultado (cada proceso calcula N/P resultados del vector y final). Por tanto, la principal diferencia con el algoritmo base de ejemplo es que ahora cada proceso puede calcular más de un elemento del vector resultado.

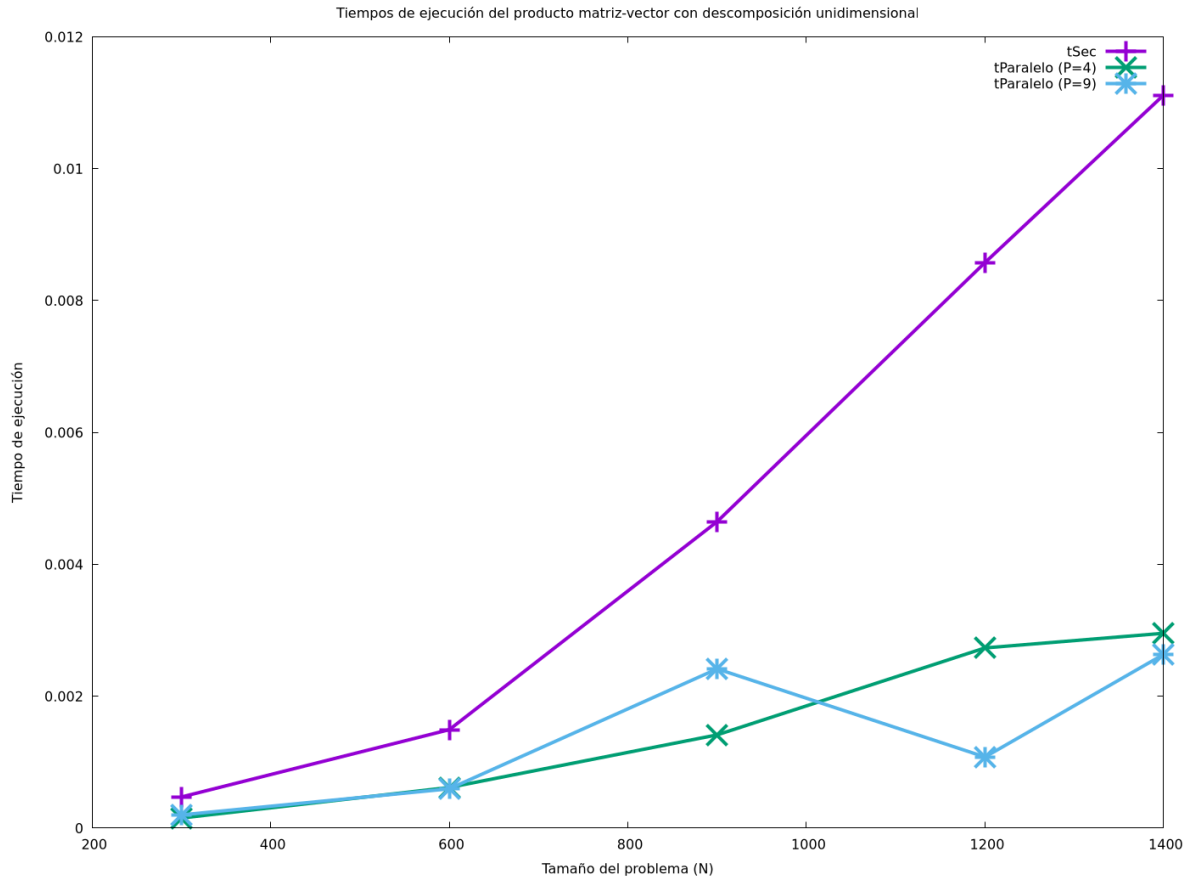
Finalmente el proceso 0 reune todos los vectores locales de cada proceso en un vector con el resultado final, con el que muestra si ha habido errores y los tiempos de ejecución empleados.

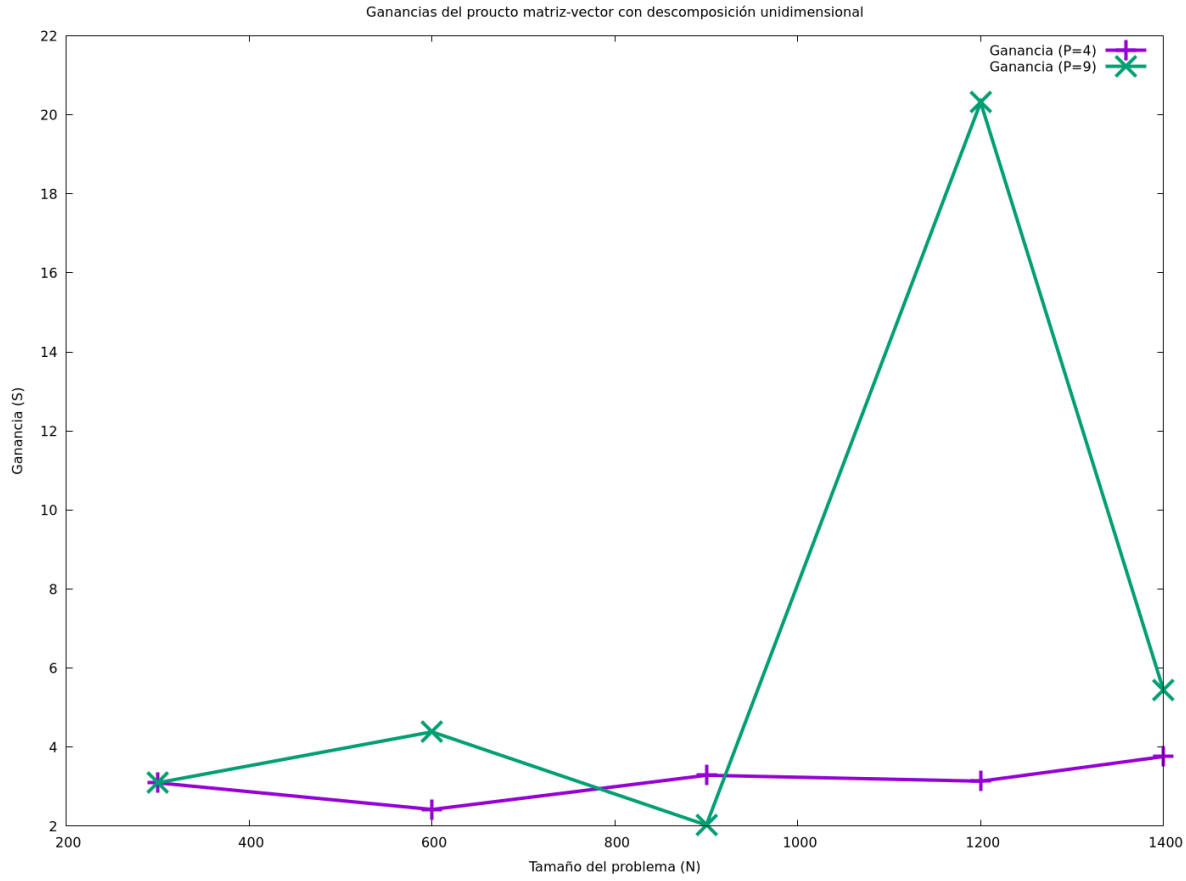
2.1. Análisis de resultados

Tamaño del problema (N)	P = 1 (sec.)	P = 4	Ganancia	P = 9	Ganancia
300	0.0004722540	0.0001525240	3.0962602607	0.0002004090	3.0980145602
600	0.0014927390	0.0006163740	2.4218072145	0.0005955520	4.3868192870
900	0.0046470860	0.0014135270	3.2875820554	0.0024144340	2.0270610006
1200	0.0085757850	0.0027325750	3.1383530187	0.0010783620	20.3132936806
1400	0.0111147470	0.0029545590	3.7618971224	0.0026337490	5.4334456321

Podemos ver que al ejecutar el algoritmo de forma paralela obtenemos mejores tiempos de ejecución, siendo el caso con 9 procesadores el mejor para tamaños de problema más grandes, aunque al principio las diferencias entre ellos sean pequeñas.

En la ganancia observamos cómo el algoritmo paralelo con 9 procesadores conlleva una gran mejora, de casi 20 veces más que el secuencial, mientras que el de 4 procesadores se queda en una mejora más o menos constante del doble que el secuencial.





3. Descomposición bidimensional (por bloques 2D)

El programa lo primero que hace es comprobar que el número de procesos con el que se ha ejecutado tiene raíz entera (4, 9, 16...) ya que es una de las condiciones que se asumen, si no es así se aborta la ejecución del programa. Al igual que el anterior, recibe el tamaño del problema N mediante parámetro de ejecución.

El algoritmo crea una topología cartesiana de procesos 2d para así poder repartir los procesos acordemente. Esta topología tiene dos dimensiones, y cada dimensión tiene como tamaño raíz de P, para así ajustar los procesos a una matriz cuadrada. Se obtienen las coordenadas de cada proceso en la topología para poder usarlos posteriormente en el vector de dos elementos **coords**.

Una vez creada, se crean tres comunicadores distintos para esta topología: uno agrupando los procesos que estén en las filas (cuyo color es la coordenada fila **coords[0]** en la topología), otro agrupando los procesos que estén en las columnas (cuyo color es la coordenada columna **coords[1]** en la topología) y por último el comunicador para los procesos en la diagonal (cuyo color es aquellas coordenadas cuya coordenada de fila y columna tengan el mismo valor **coords[0] == coords[1]**, esto indica que están en la diagonal de la matriz). Se obtienen los nuevos id de proceso para cada uno de los tres comunicadores creados. Se crearon tantas divisiones de los comunicadores de fila y columna como tamaño tenga la matriz de procesos (raíz de P), y luego se dividirán en el comunicador de diagonal los procesos que estén en la diagonal y todos los demás que no están.

En este algoritmo, al igual que en el anterior, cada proceso puede calcular más de un elemento del vector resultado y, por tanto almacena localmente un bloque de varias filas para multiplicar en un vector llamado **miBloque** cuyo tamaño es el número de elementos de la matriz con el que opera, N^2/P. El resultado parcial que calcula ahora es un vector llamado **subFinal**, cuyo tamaño es el número de filas que va a calcular cada proceso (N/P filas, que consiste en N^2/P elementos de la matriz).

El proceso 0 genera aleatoriamente los datos para el vector y la matriz, sin embargo el reparto de los datos de entrada es distinto. Se busca que cada proceso tenga un sector de la matriz y una porción del vector con la que operar. Para esto, se define un tipo vector de bloque cuadrado, con tamaño N/raíz de P, y se realizan tantos empaquetados como procesos haya, para así tener los bloques de matriz uno detrás de otros y poder repartirlos con un scatter de forma correcta. **Nota**: la matriz A se copia a un vector para poder realizar de forma correcta el empaquetado, ya que con la matriz tal y como se agrupa en memoria no es posible y se repetían siempre los primeros valores.

Luego, cada proceso tiene un subvector de x llamado **xj** en el cual almacena N/raiz de P elementos del vector. El vector x se fracciona mediante un scatter sobre la diagonal, partiéndolo en N/raíz de P trozos. Ahora cada proceso de la diagonal tiene un trozo distinto, que copiará mediante un broadcast a todos los procesos de la columna a la que pertenece ese proceso de la diagonal. El proceso que está en la diagonal hace de raíz en el broadcast, siendo su id **coords[1]** ya que el id de los procesos en la diagonal coincide con su coordenada de columna.

Cada proceso guarda un vector **y_parcial** en el que almacena los resultados calculados a partir de sus trozos de matriz y vector. Estos valores se reducen por filas, para tener cada uno de los subvectores **yi** que componen el resultado, siendo el proceso raíz el que esté en la diagonal, siendo su id esta vez **coords[0]** ya que el id de los procesos en la diagonal coincide con el de su coordenada de fila. Estos resultados se reunen mediante gather en el proceso 0, formando el vector resultado final.

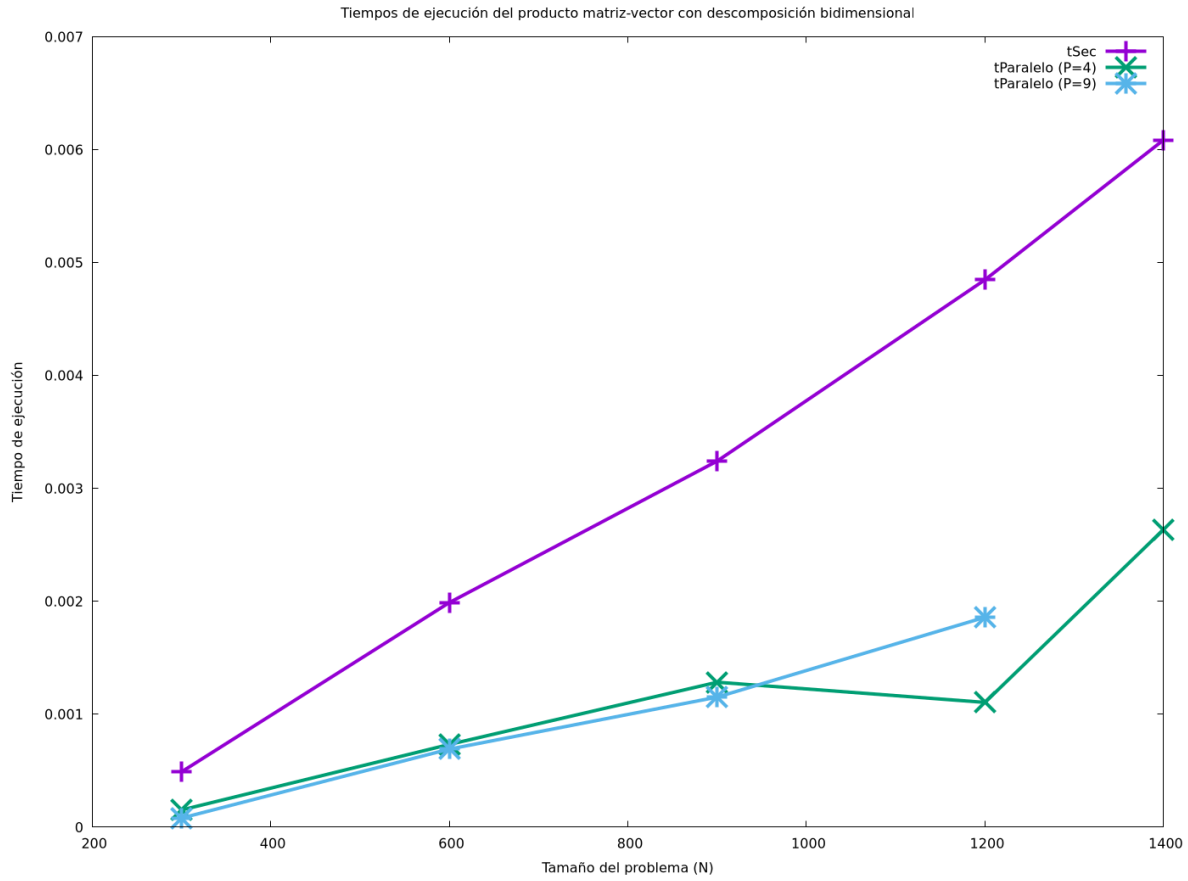
Finalmente el proceso 0 muestra si ha habido errores y los tiempos de ejecución empleados.

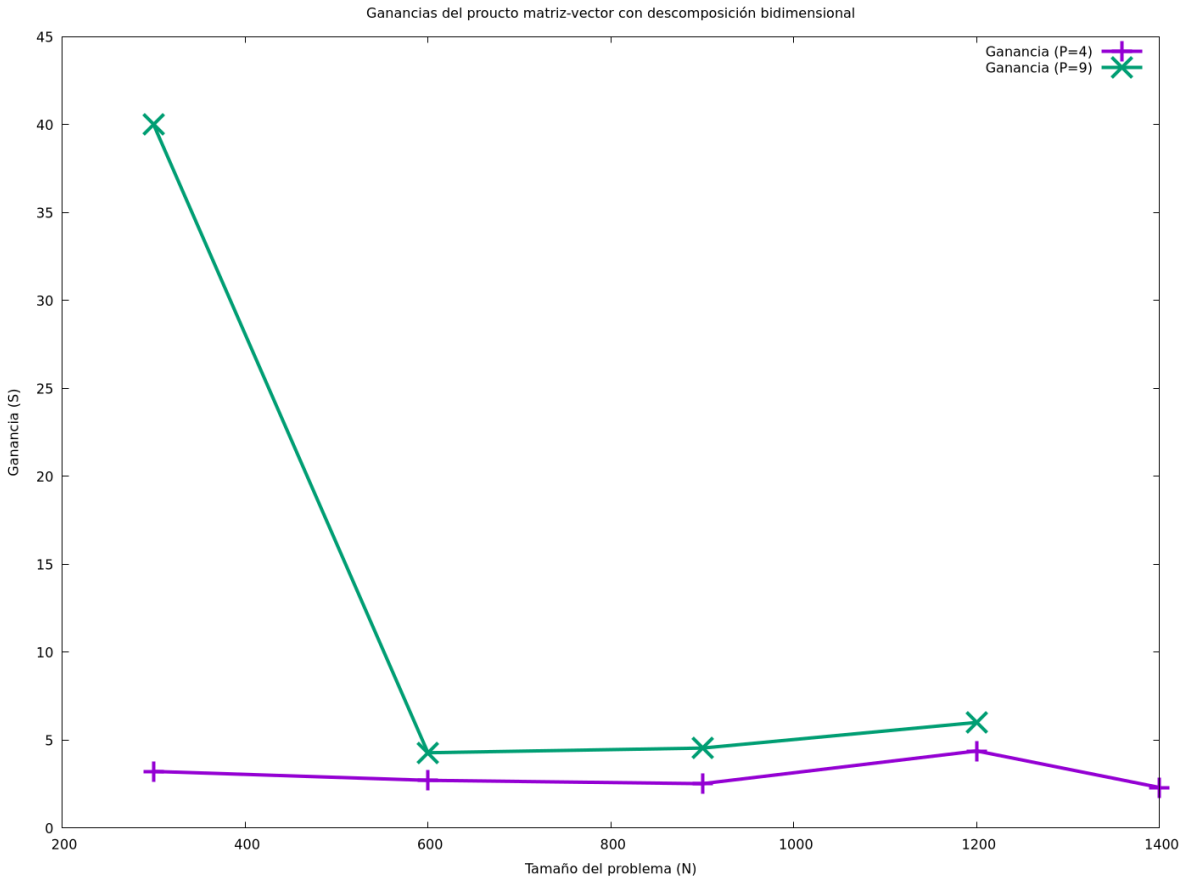
3.1. Análisis de resultados

Tamaño del problema (N)	P = 1 (sec.)	P = 4	Ganancia	P = 9	Ganancia
300	0.0004892260	0.0001518660	3.2214320519	0.0000805580	40.0093473026
600	0.0019885610	0.0007324240	2.7150407414	0.0006910240	4.2847629026
900	0.0032417570	0.0012834550	2.5258049562	0.0011516020	4.5501909514
1200	0.0048475020	0.0011048190	4.3875983306	0.0018587790	6.0097004539
1400	0.0060858810	0.0026366720	2.3081676447	X	X

En este caso, para 9 procesadores no se ha ejecutado el algoritmo puesto que el algoritmo asume que el tamaño del problema N debe ser múltiplo de la raíz cuadrada de 9, en este caso 3. 1400 no es múltiplo de 3, por lo que no se puede ejecutar el algoritmo en este caso y sólamente se ha ejecutado para 4 procesadores.

Los resultados nos dan que el algoritmo paralelo tanto para 4 como para 9 procesadores es más rápido que el secuencial, y entre ellos no hay mucha diferencia hasta que se llega a un tamaño del problema de 1000. A partir de entonces, el algoritmo de 4 procesadores acaba siendo un poco más rápido. En las ganancias podemos ver como para tamaños de problema más pequeños el algoritmo con 9 procesadores ofrece una mejora más sustancial, pero conforme aumenta N la mejora se vuelve más constante y parecida tanto en 4 como en 9 procesadores, teniendo cerca de cinco veces más que el secuencial. En general, con 9 procesadores se obtiene una mejora ligeramente mejor.




Ganancias del proucto matriz-vector con descomposición bidimensional
Ganancia (P=4)
Ganancia (P=9)
Ganancia (S)
Tamaño del problema (N)
0
5
10
15
20
25
30
35
40
45
200
400
600
800
1000
1200
1400