SISTEMAS COMPLEJOS EN MÁQUINAS PARALELAS

TRABAJO PRÁCTICO 2

MESSAGE PASSING INTERFACE

Estudiante

FREDY ANDRÉS MERCADO NAVARRO
DNI: 94.872.342
Maestría en Simulación Numérica y Control
Docentes: Guillermo Marshall, Alejandro Soba, Francisco Eijo.
Cuatrimestre: I-2012
30 de Agosto

Universidad de Buenos Aires
Ciudad Autónoma de Buenos Aires
Argentina
2012

INDICE DE CONTENIDOS

INDICE DE TABLAS

INDICE DE FIGURAS

1. EJERCICIO 1

Escriba una función MPI que calcule la operación $A\ x\ b$, para una matriz Banda, siendo A la matriz y b un vector de igual dimensión.

1.1 Descripción general del programa

El archivo se nombra fAxb.c. El programa permite computar en paralelo con un número de procesos que puede no ser divisible en forma exacta entre el número de filas de la matriz A. Éste identifica cuántas filas adicionar y guarda este dato bajo la variable add. El algoritmo fue diseñado de tal forma que a cada proceso se le distribuirán filas de la matriz A en el mismo formato .mtx, sólo que ordenadas para que cada proceso tenga solo aquellos datos que pertenecen a filas consecutivas.

Para llevar a cabo lo anterior se emplea la función MPI_Scatterv para distribuir grupos de vectores que no tienen el mismo tamaño entre todos los procesos, ya que el número de Non-zeros por fila cambia a medida que se recorre la matriz.

Si se posee un vector b debe modificarse RHS == 0 por RHS == 1 y cambiar directamente “b.txt” en la función leeb por el nombre del archivo. Si no se posee vector b, entonces se asigna el número 10 a todo el vector b automáticamente.

1.2 Descripción de las funciones

- void AtoMtx

Pasa la matriz A que está guardada como arreglo a una matriz con el mismo formato .mtx. Esto se hace con el objeto de optimizar la multiplicación agrupando los datos por grupos con la misma fila en común. Este ordenamiento es utilizado más tarde en el programa.

- void imprimetxt2

Imprime un vector cualquiera con datos tipo double.

- void multmxv_local

Multiplicación matriz por vector local. Todos los procesos tienen el vector, pero solo (DIM + add)/P filas de la matriz. DIM es la dimensión, add es el número de filas adicionadas para dividir igual número de filas entre todos los procesos y P es el número de procesos.

- int cuentaNZ

Cuenta el número de Non-zeros que tiene la matriz luego de ser descomprimida. Las matrices simétricas a veces solo poseen un lado de la simetría, por lo tanto es necesario desdoblar la simetría y recontar el número de Non-zeros resultante.

- void contadorNZPyDESP

Cuenta el número de Non-zeros que le corresponden a cada proceso y los desplazamientos, que son la suma acumulada de esos Non-zeros previamente contados.

- void leeA

Lee la matriz A del formato Matrix Market .mtx y la asigna al arreglo A.

- void leeb

Lee vector b que está almacenado completo en el archivo .mtx en columna y lo guarda en el arreglo b.

1.3 Compilación y ejecución

El programa se estuvo compilando con la siguiente línea de comandos bajo Ubuntu:

- mpicc –o fAxb.out fAxb.c –lm –Wall

Las pruebas se estuvieron ejecutando así:

- mpirun –np NP fAxb.out matrizA.mtx DIM

ej: mpirun –np 8 fAxb.out matrizA3.mtx 1224

2. EJERCICIO 2

Escriba una función con MPI que resuelva mediante el método del Gradiente Conjugado el sistema A x $x = b$. Utilice las funciones escritas en el TP1 y en el ejercicio anterior en el caso que sea posible.

2.1 Descripción general del programa

Para el desarrollo del programa paralelo fue desarrollado un programa serial, el cual está incluido en la carpeta digital del trabajo práctico.

Durante las pruebas iniciales de ambos algoritmos se notó que algunas matrices podían ser resueltas mientras que otras no. Por sugerencia de los docentes se implementó entonces el algoritmo del Gradiente Conjugado mejorado con un precondicionador diagonal de Jacobi.

Al igual que la función de multiplicación matriz por vector del EJERCICIO 1, el programa para la resolución del sistema lineal de ecuaciones comienza determinando las filas adicionales necesarias para dividir las filas de la matriz A en partes iguales, ya que el algoritmo admite un número de procesos que no sea divisible por la dimensión.

Al igual que en el algoritmo paralelo del EJERCICIO 1, la idea del programa es aprovechar el formato .mtx, ordenarlo por filas consecutivas y distribuirlo por arreglos a cada proceso, donde los tamaños pueden diferir entre un proceso y otro, lo cual fue solucionado (de nuevo) implementando la función MPI_Scatterv.

El sistema a solucionar es A x $x = b$. Si se posee vector b se debe modificar el programa igual que fue ilustrado para el EJERCICIO 1. De lo contrario, el algoritmo genera el vector $x[i] = 10$ y lo multiplica por A. De allí sale b. Posteriormente se desecha $x[i] = 10$ y se reemplaza por $x[i] = b[i]$, para comenzar así con los cálculos y las iteraciones, teniendo como solución inicial de partida al vector b.

2.2 Algoritmo del Gradiente Conjugado mejorado con precondicionador diagonal de Jacobi

El algoritmo resultante de fusionar el Gradiente Conjugado con el precondicionador de Jacobi es el presentado a continuación [1]. Es necesario en cualquier caso aclarar que el método del Gradiente Conjugado requiere de matrices reales, simétricas y definidas positivas:

$$x_0, coordenada\ inicial$$

$$g_0 = Ax_0 - b, gradiente\ inicial$$

$$q_0 = (diag\ A)^{-1} g_0$$

$$p_0 = -q_0, dirección\ inicial\ de\ descenso$$

$\epsilon, tolerancia$

$k = 0$

$mientras\ g_k \neq 0, es\ decir\ k \leq rank(A) \wedge \left|\left|g_k\right|\right| > \epsilon$

$w = Ap_k$

$\alpha_k = \frac{g_k^T q_k}{p_k^T w}$

$x_{k+1} = x_k + \alpha_k p_k$

$g_{k+1} = g_k + \alpha_k w$

$q_{k+1} = (diag\ A)^{-1} g$

$\beta_k = \frac{g_{k+1}^T q_{k+1}}{g_k^T q_k}$

$p_{k+1} = -q_{k+1} + \beta_{k+1} p_k$

$k = k + 1$

2.3 Descripción de las funciones

Fueron necesarias varias funciones para facilitar la lectura del código. A continuación se describen en forma muy resumida:

- void AtoMtx

Misma función empleada en el EJERCICIO 1: fAxb.c

- void imprimetxtNZA

Imprime los tres vectores leídos a partir del formato .mtx. Fue empleada solo para rastrear errores en el programa durante su diseño.

- void imprimetxt2

Imprime un vector cualquier con tipo de datos double.

- void imprimetxt3

Imprime vectores leídos del formato .mtx pero solo aquellos que corresponden a un solo proceso. Fue empleada solo para rastrear errores en el programa durante su diseño.

- void imprimetxt4

Imprime un vector con tipo de datos int.

- void multmxv_local

Función de multiplicación de matriz local por vector completo.

- int cuentaNZ

Misma función empleada en el EJERCICIO 1: fAxb.c

- void contadorNZPyDESP

Misma función empleada en el EJERCICIO 1: fAxb.c

- void leeA

Misma función empleada en el EJERCICIO 1: fAxb.c

- void leeb

Misma función empleada en el EJERCICIO 1: fAxb.c

- void restav_local

Resta de un vector local datos que corresponden al vector b completo. Dado que el vector b se requiere para la multiplicación, ya se halla en todos los procesos y se aprovecha así para este cómputo.

- void inversaM

Llena un vector con los datos de la inversa de la diagonal de la matriz A. Esta función se podría suprimir fusionándola con la función leeA, de tal forma que solo se requiera un barrido del archivo matrizA.mtx para generar la inversa y los datos de A.

- double prodpunto_local

Calcula el producto escalar local entre dos vectores locales.

- void sumav_local

Suma dos vectores locales, pero se emplea también para restar dos vectores dentro del ciclo iterativo.

- void imptiempostxt

Imprime un archivo con los tiempos guardados con el empleo de la función MPI_Wtime() a lo largo de todo el código.

2.1 Compilación y ejecución

Los programas secuencial y paralelo se estuvieron compilando con la siguiente línea de comandos bajo Ubuntu:

- mpicc –o GCParalelo.out GCParalelo.c –lm –Wall
- mpicc –o GCSerial.out GCSerial.c –lm –Wall

Las pruebas se estuvieron ejecutando así:

- mpirun –np NP GCParalelo.out matrizA.mtx DIM MAXITER

Ej para paralelo: mpirun –np 8 GCParalelo.out matrizA3.mtx 15439 30000

Ej para serial: mpirun –np 1 GCSerial.out matrizA3.mtx 15439 30000

Para las pruebas en el CECAR se siguieron las instrucciones otorgadas por los docentes y publicadas en la página del CECAR.

2.2 Prueba del algoritmo paralelo

Para las pruebas del algoritmo se seleccionó la matrizA11.mtx, que posee dimensión 15439 y 133840 lineas de datos en el archivo .mtx. Al tener en cuenta la simetría y descomprimir resultarán 252241 datos (Non-zeros totales). La matrizA11.mtx fue hallada en la base de datos de la colección de matrices dispersas de la universidad de Florida (UF Sparse matrix collection) [3]. El nombre original de la matriz es bcsstk25.mtx.

3. EJERCICIO 3

Realizar una medición de escalabilidad del código desarrollado en el inciso 2 determinando las porciones del mismo que conllevan mayor tiempo de cómputo. Analice el speed up de todo el código y de cada una de las funciones paralelizadas por separado.

Los siguientes resultados fueron obtenidos empleando el Cluster del Centro de Cómputo de Alto Rendimiento de la Facultad de Ciencias Exactas y Naturales –CECAR-

3.1 Tiempos de procesamiento

La función MPI_Wtime() permitió la medición de tiempos a lo largo del programa. Los siguientes son los resultados:

Tabla 1. Progresión de los tiempos de procesamiento con el incremento del número de procesos (1)

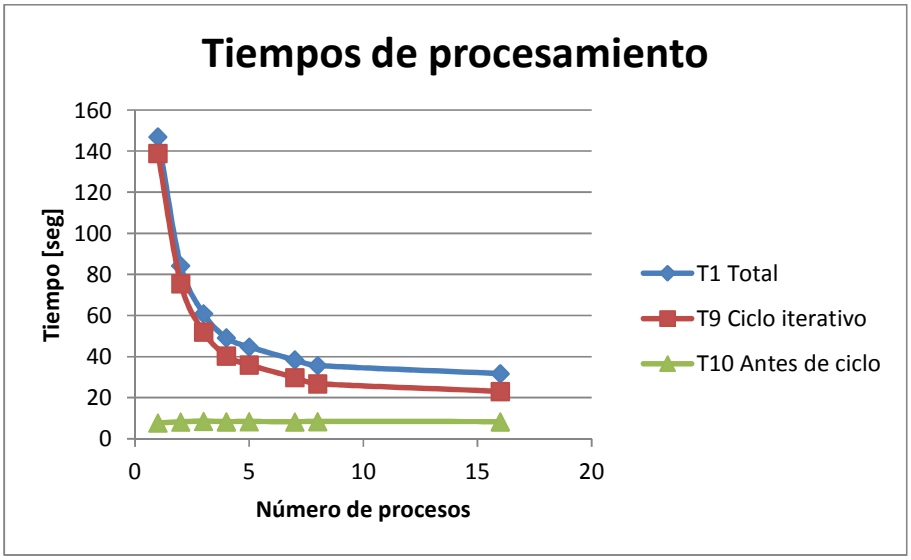
Variable	Tiempos de procesamiento (segundos)				
No. de Procesos	**1 Serial**	**1 Paralelo**	**2 P**	**3 P**	**4 P**
No. de Iteraciones	28475	28475	28914	28740	28713
T1 Total	141.629394	147.000909	84.296034	60.959946	49.218523
T2 Lee A	1.854855	1.786527	1.925702	1.906555	1.921210
T3 AtoMtx	1.467391	1.460366	1.457534	1.454957	1.458079
T4 contador	NA	0.002754	0.002757	0.002754	0.002751
T5 multmxv	0.003805	0.004015	0.001984	0.001302	0.000959
T6 restav	0.000144	0.000164	0.000072	0.000055	0.000036
T7 multmxv2	0.000160	0.000237	0.000526	0.000706	0.000897
T8 prodpunto	0.000174	0.000246	0.001254	0.001347	0.001402
T9 Ciclo iterativo	133.193687	138.896228	75.611930	52.081227	40.436339
T10 Antes de ciclo	8.068815	7.815020	8.322356	8.569227	8.375311

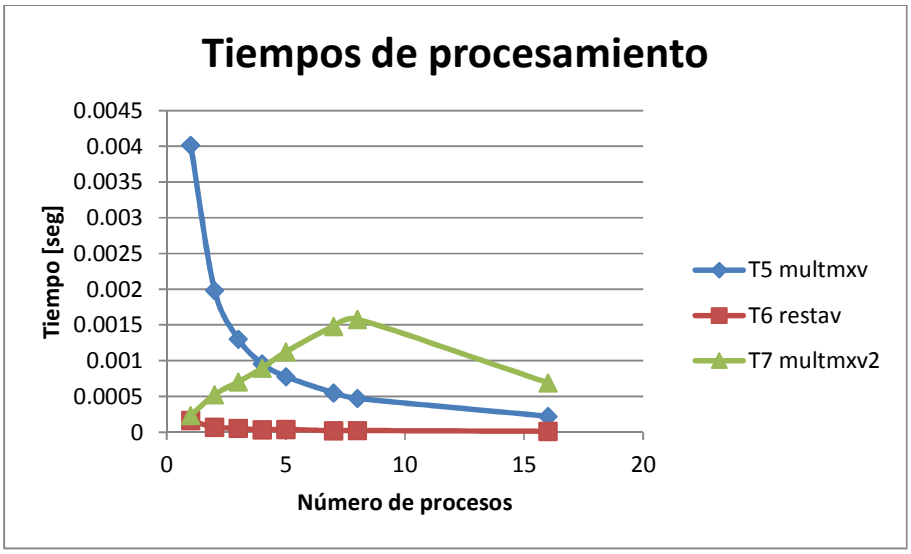
Tabla 2. Continuación de Tabla 1

Variable	Tiempos de procesamiento (segundos)			
No. de Procesos	**5 P**	**7 P**	**8 P**	**16 P**
No. de Iteraciones	28871	28851	28896	28727
T1 Total	44.783989	38.465365	35.777662	31.804420
T2 Lee A	1.925468	1.907062	1.908532	1.931320
T3 AtoMtx	1.454599	1.454410	1.454975	1.457102
T4 contador	0.002753	0.002722	0.002720	0.002731
T5 multmxv	0.000776	0.000549	0.000472	0.000221
T6 restav	0.000040	0.000022	0.000024	0.000013
T7 multmxv2	0.001125	0.001482	0.001578	0.000690
T8 prodpunto	0.007031	0.002632	0.001903	0.040264
T9 Ciclo iterativo	35.990964	29.874796	26.863464	23.114558
T10 Antes de ciclo	8.443805	8.294411	8.517911	8.370479

Resulta más fácil analizar los resultados con los siguientes gráficos:

Ilustración 1. Gráficos de tiempos de procesamiento


Tiempos de procesamiento
Tiempo [seg]
160
140
120
100
80
60
40
20
0
0
5
10
15
20
Número de procesos
T1 Total
T9 Ciclo iterativo
T10 Antes de ciclo


Tiempos de procesamiento
Tiempo [seg]
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
5
10
15
20
Número de procesos
T5 multmxv
T6 restav
T7 multmxv2


Tiempos de procesamiento
Tiempo [seg]
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
5
10
15
20
Número de procesos
T8 prodpunto

3.2 Análisis de Speed Up del código

El Speed up de un programa paralelo es [2]:

$$S = \frac{T_{serial}}{T_{paralelo}}$$

Los resultados ilustrados en las siguientes gráficas se obtuvieron dividiendo los tiempos de la columna 1 Serial de la tabla 1 sobre los tiempos de las columnas 1 Paralelo hasta 16 Paralelo.

Ilustración 2. Speed Up de funciones individuales y secciones del código (1)

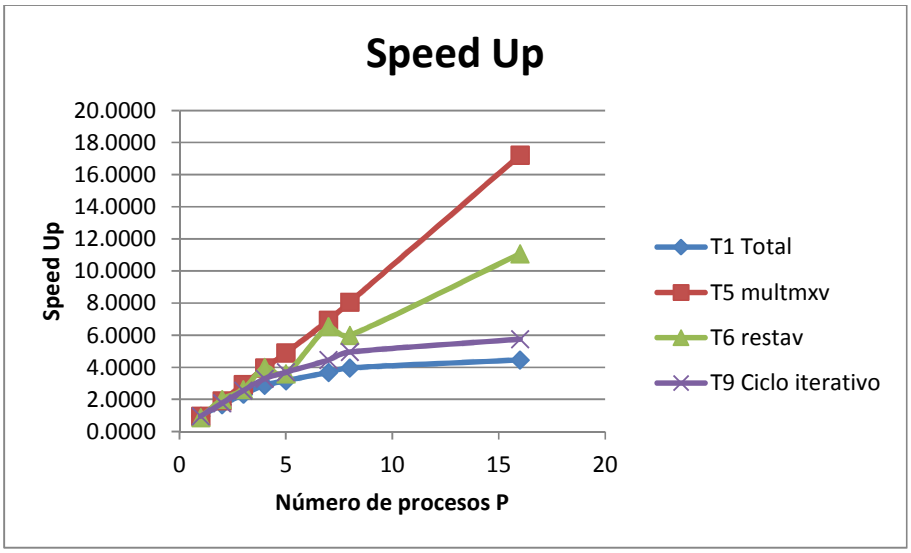
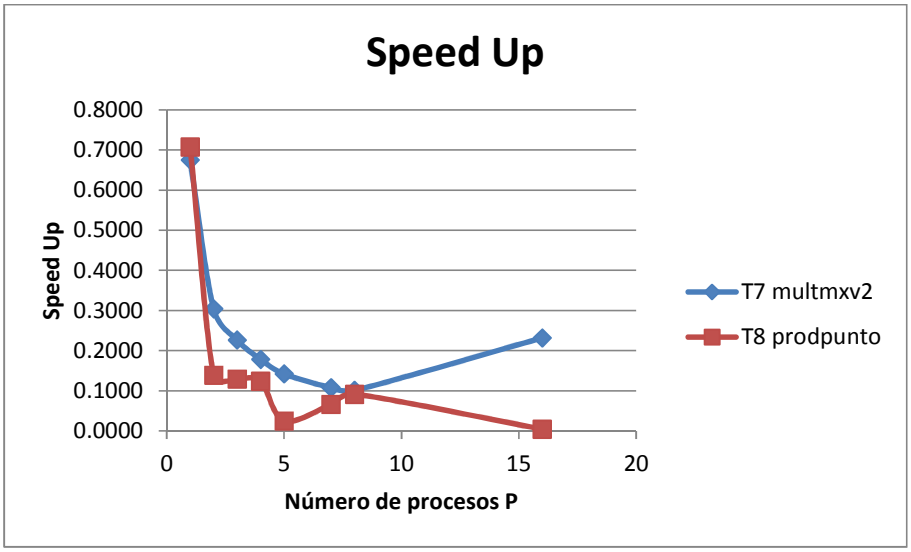


Ilustración 3. Speed Up de funciones individuales y secciones del código (2)



En los gráficos anteriores solo fueron tenidos en cuenta las funciones paralelizadas ya que, naturalmente, las funciones no paralelizadas dan Speed Up muy aproximados a 1.

Los datos que corresponden a los gráficos anteriores están consignados en la siguiente tabla.

Tabla 3. Progreso del Speed Up con el incremento del número de procesos

Variable	**Speed Up**							
No. de Procesos	**1**	**2**	**3**	**4**	**5**	**7**	**8**	**16**
T1 Total	0.9635	1.6801	2.3233	2.8776	3.1625	3.6820	3.9586	4.4531
T2 Lee A	1.0382	0.9632	0.9729	0.9655	0.9633	0.9726	0.9719	0.9604
T3 AtoMtx	1.0048	1.0068	1.0085	1.0064	1.0088	1.0089	1.0085	1.0071
T4 contador	NA	NA	NA	NA	NA	NA	NA	NA
T5 multmxv	0.9477	1.9178	2.9224	3.9677	4.9034	6.9308	8.0614	17.2172
T6 restav	0.8780	2.0000	2.6182	4.0000	3.6000	6.5455	6.0000	11.0769
T7 multmxv2	0.6751	0.3042	0.2266	0.1784	0.1422	0.1080	0.1014	0.2319
T8 prodpunto	0.7073	0.1388	0.1292	0.1241	0.0247	0.0661	0.0914	0.0043
T9 Ciclo iterativo	0.9589	1.7615	2.5574	3.2939	3.7008	4.4584	4.9582	5.7623
T10 Antes de ciclo	1.0325	0.9695	0.9416	0.9634	0.9556	0.9728	0.9473	0.9640

4. CONCLUSIONES Y OBSERVACIONES

- Por alguna razón desconocida, para un número de procesos P=6 el sistema retorna un error. Por esta razón no se obtuvieron datos para P=6 procesos.

- El empleo del precondicionador de Jacobi resultó fundamental para lograr una rápida convergencia de las matrices de prueba. No obstante, algunas matrices convergieron a soluciones diferentes a la esperada (x[i] = 10).

- El número de iteraciones requeridas por el programa paralelo para lograr la convergencia fue diferente para diferente número de procesos, excepto si comparamos la corrida del programa serial con el paralelo con 1 proceso, en donde convergieron en igual número de iteraciones.

- El programa desarrollado no procesa las líneas de comentarios de los archivos .mtx, por lo tanto dichas líneas deben ser suprimidas antes de correr los programas.

- Según la literatura, el número de iteraciones requeridas para la convergencia es a lo sumo la misma dimensión. Dado que los cómputos no son ideales, algunas matrices convergen en una iteración que sobrepasa la dimensión.

- Como era de esperarse, el tiempo medido para el cómputo de las funciones que no se paralelizaron, como leeA, AtoMtx, contador, permanecen iguales con el incremento del número de procesos.

- El Speed Up de la función multmxv es casi ideal, según las mediciones. El Speed Up de la función es prácticamente S=P (P es el número de procesos).

- Las funciones multmxv2 y prodpunto son realmente productos punto las dos. La hipótesis de la falla del Speed Up evidenciada por la gráfica radica en que la medición de tiempo incluyó el pasaje de mensajes, lo cual probablemente es la causa de la incoherencia de los datos con lo esperado.

- El Speed Up alcanzado para el ciclo iterativo es de 5.76 para 16 procesos.

- La multiplicación matriz por vector es la función que más tiempo requiere. El haber obtenido un Speed Up casi ideal con ésta función contribuye enormemente a la escalabilidad del código.

REFERENCIAS BIBLIOGRÁFICAS

1. VARGAS-Félix, Miguel. Presentación: Gradiente Conjugado. Septiembre de 2011.
2. PACHECO, Peter S. An introduction to parallel programming. Morgan Kaufmann. Elsevier Inc. 2011.
3. Página de internet: http://www.cise.ufl.edu/research/sparse/matrices/list_by_type.html Base de datos de matrices dispersas de la Universidad de Florida.