SISTEMAS COMPLEJOS EN MÁQUINAS PARALELAS

TRABAJO PRÁCTICO 3

PROGRAMACION EN PARALELO EMPLEANDO OPENMP

Estudiante

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

Universidad de Buenos Aires
Ciudad Autónoma de Buenos Aires
Argentina
2012

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1. INTRODUCCIÓN

El presente trabajo busca aprovechar la capacidad de cómputo paralelo de una máquina con múltiples procesadores, aprovechando la Interfaz para Programación de Aplicaciones –API– OpenMP. Esta se basa en el modelo de ejecución fork-join, donde una labor de cómputo muy pesada se divide en cierto número de hilos (threads). Cada hilo ejecuta su parte de la carga por separado, para luego recolectar el cómputo de cada hilo en un solo resultado.

Se desarrollaron dos programas. Uno que corre de manera secuenciaL y otro que corre en paralelo. Ambos programas tienen como archivos de entrada una matriz y un vector, los cuales se emplean para computar una multiplicación matriz por vector y un producto punto con el mismo vector.

Se medirán los tiempos empleados por el programa secuencial en la región de cada código que ejecuta las funciones y luego estas mediciones de tiempo serán comparadas para conocer la escalabilidad del algoritmo paralelo corriendo en una máquina con varios threads.

Dado que los programas fueron diseñados para recibir un vector, además de una matriz, se desarrolló un programa para generar un arreglo de números aleatorios, con la opción de especificar su tamaño y el nombre del archivo que deseamos. Los detalles de compilación y ejecución están disponibles en cada archivo .c al igual que en este documento.

2. COMENTARIOS GENERALES

2.1 Definición del problema

Se deben reescribir las funciones Producto Interno y Multiplicación matriz por vector para una matriz banda utilizando OpenMP. También se debe realizar una medición de escalabilidad del código y analizar el Speed Up.

2.2 Selección de la matriz

La matriz seleccionada para probar el código paralelo fue descargada de la base de datos de matrices dispersas de la Universidad de Florida [3]. Se buscó una matriz con datos suficientes para que el ejercicio computacional arrojara tiempos con uno o varios órdenes de magnitud mayores que la resolución mínima que otorga la función para medir tiempos. La matriz seleccionada se puede ubicar bajo el nombre shipsec5. Su dimensión es 179.860 y el número de componentes diferentes de cero es 5´146.478. Nos limitamos a esta cantidad de datos debido a que el archivo descomprimido ocupaba más de 100 MB de espacio en disco y una matriz más grande hacía más lento su manejo.

El nombre de esta matriz se modificó a matrizA8.mtx para facilitar su escritura en la línea de comandos.

2.3 Generación del vector

El vector se generó con el programa generab.out (generab.c). A través de la línea de comandos las instrucciones son:

- Compilar: gcc -o generab.out generab.c -lm –Wall
- Ejecutar: ./generab.out nombreelegido.mtx DIM

  Ejemplo: ./generab.out b8.mtx 179860

2.4 Máquina empleada para las pruebas

En el Departamento de Computación de la UBA se cuenta con una máquina Intel con 16 cores y 32 threads, sin embargo, al momento de las pruebas se comprobó que sólo se contaba con 4 threads disponibles. Debido a esto, se decidió emplear una máquina portátil que posee hasta 8 threads disponibles. Teniendo en cuenta esto, las pruebas se llevaron a cabo variando el número de threads entre 1 y 8 con la máquina portátil (DELL XPS L502X, procesador Intel Core i7 de segunda generación con procesador 2630QM).

3. DISEÑO DEL ALGORITMO SERIAL

Como parte del trabajo práctico se debió diseñar un programa que calculara las funciones en forma secuencial. Esto con el objeto de utilizar sus resultados y compararlos numéricamente con los resultados calculados por el programa paralelo escrito empleando OpenMP.

3.1 Descripción general del programa

El algoritmo secuencial recibe una matriz banda, cuadrada y de números reales en formato del Matrix Market [1]. De igual manera, recibe un vector, y tanto éste como la matriz tienen igual dimensión. A través de línea de comandos se especifica la matriz de entrada, el vector, la dimensión y el número de componentes diferentes de cero que posee la matriz.

Los dos archivos son leídos y almacenados idénticamente en memoria. Luego, el producto interno es calculado empleando dos veces el mismo vector, mientras que la multiplicación matriz por vector se realiza leyendo línea por línea cada posición de los arreglos I (fila), J (columna) y valor, y va realizando la multiplicación leyendo una línea a la vez.

3.2 Código desarrollado

A continuación se presenta el código secuencial desarrollado:

```
//PROGRAMA      x * x = ppunto    [escalar]
//              A * x = b         [vector]

//argv[0]: nombre del programa.
//argv[1]: nombre del archivo de la matriz A -> matrizA1.mtx
//argv[2]: nombre del archivo del vector x --> b1.mtx
//argv[3]: dimensiÃ³n DIM

//COMPILAR: gcc -o tp3Serial2.out tp3Serial2.c -lm -Wall
//EJECUTAR: ./tp3Serial2.out matrizA.mtx vectorb.mtx DIM NZA

//ENTRADA matrizA.mtx vectorb.mtx DIM NZA
//Crear carpeta salida_s

//PRECAUCION matrizA.mtx y vectorb.mtx deben poseer igual DIM

//MIDE LOS TIEMPOS CON MPI

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<mpi.h>

void leeA(char *filename, int *I, int *J, double *valor,
  int NZA);
void leeb(char *filename, double *b, int DIM);
double prodpunto(double *a, double *b, int DIM);
void multmxv2(int *I, int *J, double *valor, double *vector,
  double *b, int DIM, int NZA);
```

```
void imprimetxt2(char *filename, double *vector, int mfilas);
void imprimetxt(char *filename, double numero);
void imptiempostxt(char *filename, double *tiempo);

int main(int argc, char **argv) {

  int DIM, NZA, *I, *J, i;
  double *x, *valor, ppunto=0.0, *b;
  double tinicial[20], tfinal[20], tiempo[20];

  sscanf(argv[3], "%d", &DIM);
  sscanf(argv[4], "%d", &NZA);

    MPI_Init(&argc, &argv);

        tinicial[0] = MPI_Wtime();

  //Asignación de memoria
  I = calloc(NZA, sizeof(int));
  J = calloc(NZA, sizeof(int));
  valor = calloc(NZA, sizeof(double));
  x = calloc(DIM, sizeof(double));
  b = calloc(DIM, sizeof(double));

        tinicial[1] = MPI_Wtime();
  leeA(argv[1], I, J, valor, NZA);
        tfinal[1] = MPI_Wtime();

        tinicial[2] = MPI_Wtime();
  leeb(argv[2], x, DIM);
        tfinal[2] = MPI_Wtime();

        tinicial[5] = MPI_Wtime();
  ppunto = prodpunto(x, x, DIM);
        tfinal[5] = MPI_Wtime();

        tinicial[6] = MPI_Wtime();
  multmxv2(I, J, valor, x, b, DIM, NZA);
        tfinal[6] = MPI_Wtime();

  printf("ppunto = %f\n", ppunto);

  imprimetxt2("./salida_s/b_s.dat", b, DIM);
  imprimetxt("./salida_s/ppunto_s.dat", ppunto);

  free(I);
  free(J);
  free(valor);
  free(x);
  free(b);

        tfinal[0] = MPI_Wtime();

  for (i=0; i<20; i++)
    tiempo[i] = tfinal[i] - tinicial[i];
```

```
  imptiempostxt("./salida_s/tiempos_s.txt", tiempo);
  printf("Tiempo A*b = %f\n", tiempo[6]);
  printf("Tiempo ppunto = %f\n", tiempo[5]);


    MPI_Finalize();

  return 0;
}



void leeA(char *filename, int *I, int *J, double *valor, int
NZA) {

  int k;
  FILE *punteroA;
  punteroA = fopen(filename, "r");

  for (k=0; k<NZA; k++) {
    fscanf(punteroA, "%d %d %lf\n", &I[k], &J[k], &valor[k]);
  }

  fclose(punteroA);
}

void leeb(char *filename, double *b, int DIM) {

  int i;

  FILE *punterob;
  punterob = fopen(filename, "r");

  for (i=0; i<DIM; i++)
    fscanf(punterob, "%lf\n", &b[i]);

  fclose(punterob);
}

double prodpunto(double *a, double *b, int DIM) {

  int i;
  double c=0.0;

  for (i=0; i<DIM; i++)
    c += a[i] * b[i];

  return c;
}

void multmxv2(int *I, int *J, double *valor, double *vector,
  double *b, int DIM, int NZA) {

  int i;
```

```
  for (i=0; i<DIM; i++)
    b[i] = 0.0;


  for (i=0; i<NZA; i++)
    b[I[i]-1] += valor[i] * vector[J[i]-1];
}

void imprimetxt2(char *filename, double *vector, int mfilas) {

  int i;

  FILE *file;
  file = fopen(filename, "w");

  for (i=0; i<mfilas; i++)
    fprintf(file, "%d %.15e\n", i+1, vector[i]);

  fclose(file);
}

void imprimetxt(char *filename, double numero) {

  FILE *file;
  file = fopen(filename, "w");
  fprintf(file, "%.15e\n", numero);
  fclose(file);
}

void imptiempostxt(char *filename, double *tiempo) {
  FILE *file;
  file = fopen(filename, "w");

  fprintf(file, "t0 Tiempo total = %f s\n", tiempo[0]);
  fprintf(file, "t1 Tiempo leeA = %f s\n", tiempo[1]);
  fprintf(file, "t5 Tiempo ppunto = %f s\n", tiempo[5]);
  fprintf(file, "t6 Tiempo multmxv2 = %f s\n", tiempo[6]);

  fclose(file);
}
```

3.3 Compilación y ejecución

El programa secuencial se compila con la siguiente línea de comandos bajo el sistema operativo Linux-Ubuntu en la Terminal:

- **Compilar**: mpicc -o tp3Serial4.out tp3Serial4.c -lm -Wall

Las pruebas se estuvieron ejecutando con la siguiente línea de comandos:

- **Correr**: mpirun -np 1 ./tp3Serial4.out matrizA11.mtx b11.mtx DIM NZA

  Ejemplo: mpirun -np 1 ./tp3Serial4.out matrizA11.mtx b11.mtx 179860 5146478

Es necesario clarificar que MPI es empleado sólo para la medición de tiempos. Notar que sólo se emplea un proceso.

3.4 Prueba del algoritmo secuencial

Al ejecutar el programa como lo muestra el ejemplo en el numeral anterior, los resultados obtenidos fueron:

- Valor del producto interno: 598´950.610.
- Tiempo transcurrido durante cálculo de producto matriz por vector: 0.037436 seg.
- Tiempo transcurrido durante cálculo de producto interno: 0.000604 seg.

4. DISEÑO DEL ALGORITMO PARALELO CON OPENMP

4.1 Descripción general del programa

El programa recibe, de igual manera que el secuencial, una matriz y un vector. El algoritmo que calcula el producto punto y la multiplicación matriz por vector son muy similares a los del código secuencial, con la diferencia de haber empleado las instrucciones de OpenMP para dividir la carga entre el número de threads especificado. Para la función producto interno sólo se antepusieron las instrucciones #pragma omp parallel for reduction(+:sum), mientras que el cálculo de A x b se diseñó haciéndolo depender del rank asignado internamente a cada thread. De este modo, cada thread calcula línea por línea de I, J y valor locales un aporte para el arreglo b (resultado), que corresponde a un arreglo compartido.

Cada thread va aportando una suma a la componente de b que le corresponda, de acuerdo al valor que posea en ese momento el vector I (fila). Para realizar esto es necesario calcular previamente en qué triada (fila) de la matriz en formato .mtx inicia el cálculo de cada thread y en qué triada termina.

4.2 Código desarrollado

El código paralelo diseñado se observa a continuación:

```
//PROGRAMA        x * x = ppunto    [escalar]
//                A * x = b         [vector]

//argv[0]: nombre del programa.
//argv[1]: nombre del archivo de la matriz A -> matrizA8.mtx
//argv[2]: nombre del archivo del vector x --> b8.mtx
//argv[3]: dimensión DIM
//argv[4]: nonzeros A NZA
//argv[5]: número de threads NTH

//COMPILAR: gcc -fopenmp -o tp3Paralelo2.out tp3Paralelo2.c -lm
-Wall
//EJECUTAR: ./tp3Paralelo.out matrizA.mtx vectorb.mtx DIM NZA
NTH

//ENTRADA matrizA.mtx vectorb.mtx DIM NTH
    //Crear carpeta ./salida_p

//PRECAUCIÓN matrizA.mtx y vectorb.mtx deben poseer igual
DIMensión

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<omp.h>

void leeA(char *filename, int *I, int *J, double *valor,
    int NZA);
void leeb(char *filename, double *b, int DIM);
```

```
void multmxv2(int *I, int *J, double *valor, double *x,
    double *b, int inicio_NZA_local, int NZA_local);
double prodpunto(double *a, double *b, int DIM, int NTH);
void imprimetxt2(char *filename, double *vector, int mfilas);
void imprimetxt(char *filename, double numero);
void imptiempostxt(char *filename, double *tiempo);

int main(int argc, char **argv) {

  int DIM, NZA, NTH, *I, *J, i;
  int rank, nthreads, NZA_local, inicio_NZA_local;
  double *x, *valor, ppunto=0.0, *b;
  double tinicial[20], tfinal[20], tiempo[20];

  sscanf(argv[3], "%d", &DIM);
  sscanf(argv[4], "%d", &NZA);
  sscanf(argv[5], "%d", &NTH);

        tinicial[0] = omp_get_wtime();

  //Asignación de memoria
  I = calloc(NZA, sizeof(int));
  J = calloc(NZA, sizeof(int));
  valor = calloc(NZA, sizeof(double));
  x = calloc(DIM, sizeof(double));
  b = calloc(DIM, sizeof(double));

  printf("Tamaño de reserva de memoria = %d MB\n",
NZA*5*8/(1024*1024));

  //Función para leer A
        tinicial[1] = omp_get_wtime();
  leeA(argv[1], I, J, valor, NZA);
        tfinal[1] = omp_get_wtime();

  //Función para leer b
        tinicial[2] = omp_get_wtime();
  leeb(argv[2], x, DIM);
        tfinal[2] = omp_get_wtime();

        tinicial[7] = omp_get_wtime();
  #pragma omp parallel for shared(b) num_threads(NTH)
  for (i=0; i<DIM; i++)
    b[i] = 0.0;
        tfinal[7] = omp_get_wtime();

  printf("Threads empleados = %d de %d\n", NTH,
omp_get_max_threads());

        tinicial[8] = omp_get_wtime();
  #pragma omp parallel shared(nthreads,NZA,I,J,valor,x,b)
private(rank,NZA_local,inicio_NZA_local,i) num_threads(NTH)
  {
    //Pide rank de ese hilo
    rank = omp_get_thread_num();
```

```
    //Pide total de hilos disponibles
    nthreads = omp_get_num_threads();
    //if (rank == 0)
      //printf("num_threads = %d\n", nthreads);

    //Cada hilo calcula cuantos Non Zeros de A
    //procesará.
    NZA_local = NZA / nthreads;

    //El algoritmo de multiplicación matriz*vector
    //es dependiente del rank. Por eso se calcula
    //un número para el inicio del cálculo.
    inicio_NZA_local = rank * NZA_local;
    //printf("thread %d inicio_NZA_local = %d\n", rank,
inicio_NZA_local);

    //Cuando NZA_totales no es divisible en enteros
    //entre el número de hilos, el último hilo
    //computa su carga mas el residuo de esa división.
    if (rank == nthreads-1)
      NZA_local = NZA - inicio_NZA_local;
      //printf("thread %d NZA_local = %d\n", rank, NZA_local);

    //FUNCION PARA MULTIPLICACION matriz * vector
    //Depende del rank
        #pragma omp barrier
        tinicial[6] = omp_get_wtime();
    multmxv2(I, J, valor, x, b, inicio_NZA_local, NZA_local);
        #pragma omp barrier
        tfinal[6] = omp_get_wtime();
  }
        tfinal[8] = omp_get_wtime();

  //PRODUCTO PUNTO
  //Independiente del rank
        #pragma omp barrier
        tinicial[5] = omp_get_wtime();
  ppunto = prodpunto(x, x, DIM, NTH);
        #pragma omp barrier
        tfinal[5] = omp_get_wtime();

        tinicial[9] = omp_get_wtime();
  //Impresiones
  printf("ppunto = %f\n", ppunto);
  imprimetxt2("./salida_p/b_p.dat", b, DIM);
  imprimetxt("./salida_p/ppunto_p.dat", ppunto);
        tfinal[9] = omp_get_wtime();

        tinicial[10] = omp_get_wtime();
  //Libera memoria
  free(I);
  free(J);
  free(valor);
  free(x);
  free(b);
```

```
        tfinal[10] = omp_get_wtime();
        tfinal[0] = omp_get_wtime();

  //CÃ¡lculo de tiempos
  for (i=0; i<20; i++)
    tiempo[i] = tfinal[i] - tinicial[i];

  //Imprime archivo con tiempos
  imptiempostxt("./salida_p/tiempos_p.txt", tiempo);

  printf("Tiempo A*b = %f\n", tiempo[8]);
  printf("Tiempo ppunto = %f\n", tiempo[5]);

  return 0;
}

void leeA(char *filename, int *I, int *J, double *valor, int
NZA) {

  int k;
  FILE *punteroA;
  punteroA = fopen(filename, "r");

  for (k=0; k<NZA; k++)
    fscanf(punteroA, "%d %d %lf\n", &I[k], &J[k], &valor[k]);

  fclose(punteroA);
}

void leeb(char *filename, double *b, int DIM) {

  int i;

  FILE *punterob;
  punterob = fopen(filename, "r");

  for (i=0; i<DIM; i++)
    fscanf(punterob, "%lf\n", &b[i]);

  fclose(punterob);
}

void multmxv2(int *I, int *J, double *valor, double *x, double
*b, int inicio_NZA_local, int NZA_local) {

  int i;

  for (i=inicio_NZA_local; i<inicio_NZA_local+NZA_local; i++)
    b[I[i]-1] += valor[i] * x[J[i]-1];
}

double prodpunto(double *a, double *b, int DIM, int NTH) {

  int i;
  double sum=0.0;
```

```
  #pragma omp parallel for shared(a,b,DIM) private(i)
reduction(+:sum) num_threads(NTH)
  for (i=0; i<DIM; i++)
    sum += a[i] * b[i];

  return sum;
}

void imprimetxt2(char *filename, double *vector, int mfilas) {

  int i;

  FILE *file;
  file = fopen(filename, "w");

  for (i=0; i<mfilas; i++)
    fprintf(file, "%d %.15e\n", i+1, vector[i]);

  fclose(file);
}

void imprimetxt(char *filename, double numero) {

  FILE *file;
  file = fopen(filename, "w");
  fprintf(file, "%.15e\n", numero);
  fclose(file);
}

void imptiempostxt(char *filename, double *tiempo) {
  FILE *file;
  file = fopen(filename, "w");

  fprintf(file, "t0 Tiempo total = %f s\n", tiempo[0]);
  fprintf(file, "t1 Tiempo leeA = %f s\n", tiempo[1]);
  fprintf(file, "t2 Tiempo leeb = %f s\n", tiempo[2]);
  fprintf(file, "t5 Tiempo ppunto = %f s\n", tiempo[5]);
  fprintf(file, "t6 Tiempo multmxv2 = %f s\n", tiempo[6]);
  fprintf(file, "t7 Tiempo #pragma{} b=0.0 = %f s\n",
tiempo[7]);
  fprintf(file, "t8 Tiempo #pragma{} multmxv2 = %f s\n",
tiempo[8]);
  fprintf(file, "t9 Tiempo impresiones = %f s\n", tiempo[9]);
  fprintf(file, "t10 Tiempo libera memoria = %f s\n",
tiempo[10]);

  fclose(file);
}
```

4.1 Compilación y ejecución

El programa paralelo se compiló con la siguiente línea de comandos bajo el sistema operativo Linux-Ubuntu, desde la Terminal:

- Compilar: gcc -fopenmp -o tp3Paralelo2.out tp3Paralelo2.c -lm -Wall

Las pruebas se ejecutaron con la siguiente línea de comandos:

- Correr: ./tp3Paralelo2.out matrizA.mtx vectorb.mtx DIM NZA NTH, siendo DIM: dimensión, NZA: número de componentes diferentes de cero de matriz A, NTH: número de threads deseados para los cálculos paralelos.

  Ejemplo: ./tp3Paralelo2.out matrizA8.mtx b8.mtx 179860 5146478 8

4.2 Prueba del algoritmo paralelo

Como se observa en el numeral anterior, existe la posibilidad de especificar el número de threads que se desean emplear para los cálculos. El programa tp3Paralelo2.out se ejecutó 8 veces, variando el número de threads entre 1 y 8, registrando las mediciones de tiempo manualmente en una hoja de cálculo.

5. PRUEBAS Y ANÁLISIS DE RESULTADOS

5.1 Sectores de código para toma de tiempos

A continuación se ilustran los sectores de código a los cuales se realizó medición de tiempos. En las secciones 3.2 y 4.2 se pueden identificar en negrilla las líneas de código donde se tomaron los tiempos.

5.1.1 Tiempos para el programa secuencial

Los tiempos se calcularon midiendo dos valores de tiempo y calculando la diferencia entre el tiempo final y el tiempo inicial:

```
        tinicial[5] = MPI_Wtime();
ppunto = prodpunto(x, x, DIM);
        tfinal[5] = MPI_Wtime();


        tinicial[6] = MPI_Wtime();
multmxv2(I, J, valor, x, b, DIM, NZA);
        tfinal[6] = MPI_Wtime();
```

5.1.2 Tiempos para el programa paralelo

Los tiempos se calcularon de igual forma que en el programa secuencial, pero teniendo en cuenta que ese sección del código está siendo recorrido en forma paralela por múltiples threads. Se empleó la instrucción #pragma omp barrier para sincronizar todos los threads en esas líneas y garantizar que los tiempos medidos corresponden al tiempo total que les llevó a todos los threads terminar su trabajo:

```
        #pragma omp barrier
        tinicial[5] = omp_get_wtime();
ppunto = prodpunto(x, x, DIM, NTH);
        #pragma omp barrier
        tfinal[5] = omp_get_wtime();


        #pragma omp barrier
        tinicial[6] = omp_get_wtime();
multmxv2(I, J, valor, x, b, inicio_NZA_local, NZA_local);
        #pragma omp barrier
        tfinal[6] = omp_get_wtime();
```

5.2 Resultados de las mediciones de tiempo

Los tiempos medidos durante la ejecución de las funciones en el programa secuencial están consignados en la Tabla 1.

Tabla 1. Tiempos de cálculo medidos en el código secuencial desarrollado en C.

VARIABLE	TIEMPOS DE CÁLCULO [seg]
Tiempo A x b	0.037436
Tiempo P. punto	0.000604

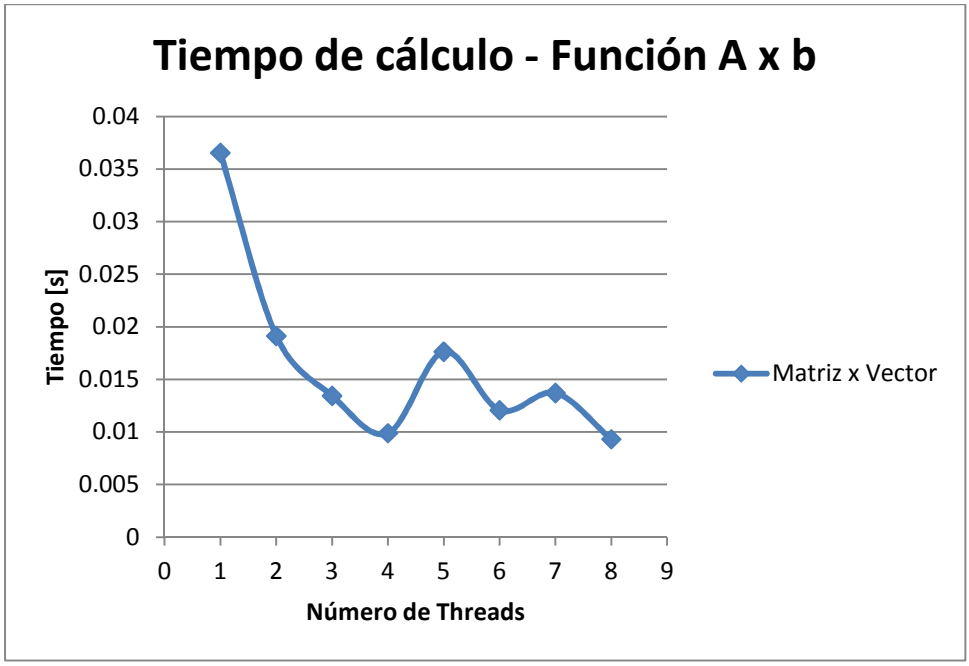
Los resultados numéricos de los tiempos medidos para evaluar el desempeño del programa paralelo están consignados en la Tabla 2.

Tabla 2. Tiempos de cálculo y Speed Up del código paralelo desarrollado con OpenMP

VARIABLE	TIEMPOS DE CÁLCULO [seg] y SPEED UP [adimensional]							
Threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Tiempo A x b	0.03654	0.019122	0.013435	0.00988	0.017628	0.012063	0.013704	0.009303
Speed Up	1.02	1.96	2.79	3.79	2.12	3.10	2.73	4.02
Tiempo P. punto	0.000600	0.000319	0.000226	0.000172	0.000257	0.000216	0.000186	0.000165
Speed Up	1.01	1.89	2.67	3.51	2.35	2.80	3.25	3.66

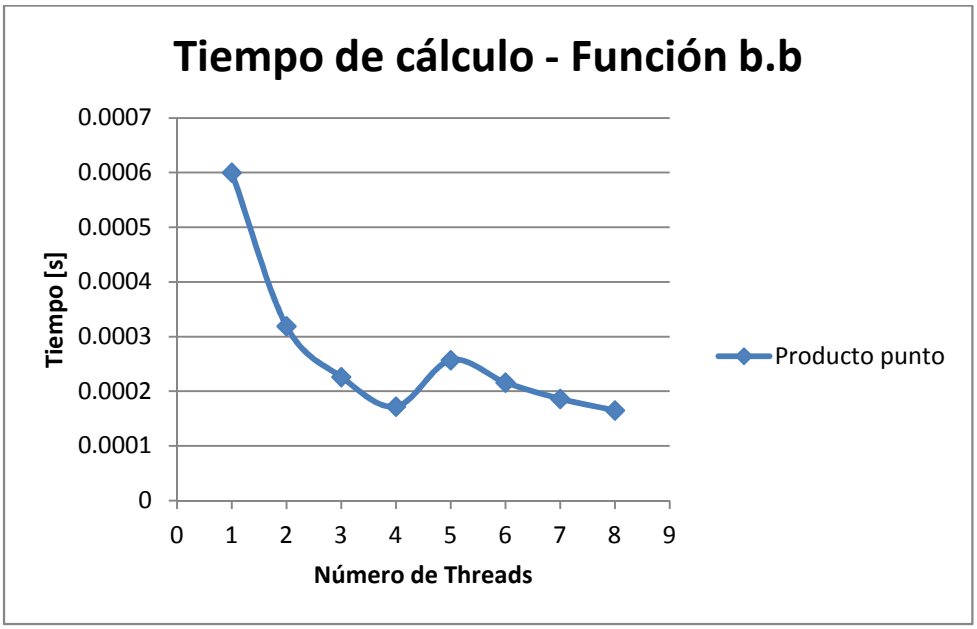
La Ilustración 1 contiene la gráfica de los tiempos que le tomó al programa paralelo realizar una sola multiplicación Matriz por Vector. La tendencia muestra una disminución de tiempos con el aumento del número de threads.

Ilustración 1. Tiempos de cálculo para la multiplicación Matriz por Vector.



La Ilustración 2 contiene la gráfica de los tiempos que le tomó al programa paralelo realizar un solo producto interno b.b, siendo b el archivo de entrada b8.mtx. La tendencia en la disminución de tiempos es muy similar a la del producto Matriz por Vector.

Ilustración 2. Tiempos de cálculo para la función Producto Interno.



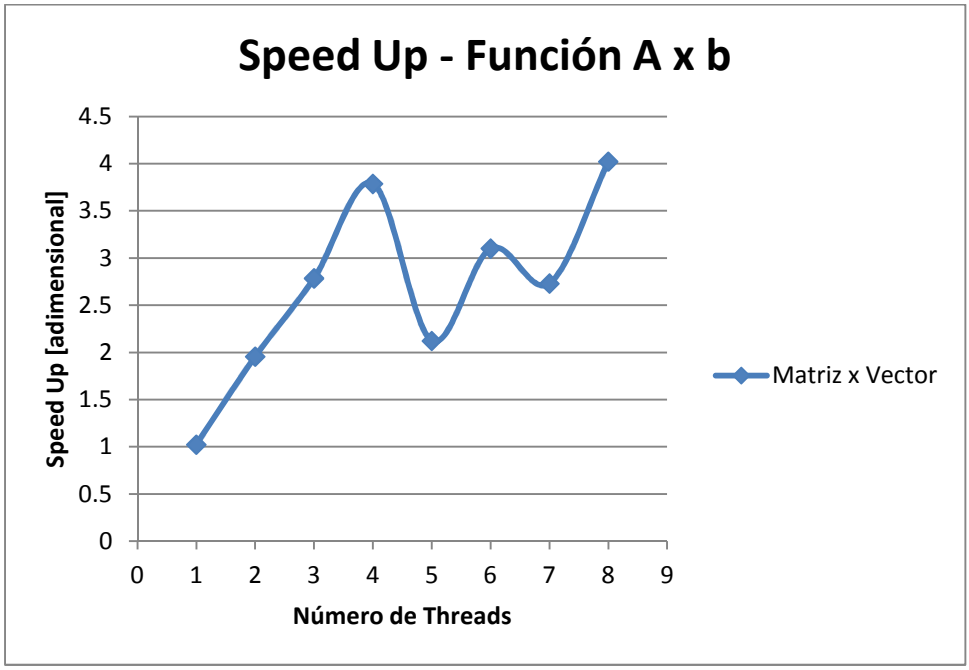
5.3 Análisis de Speed Up del código

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

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

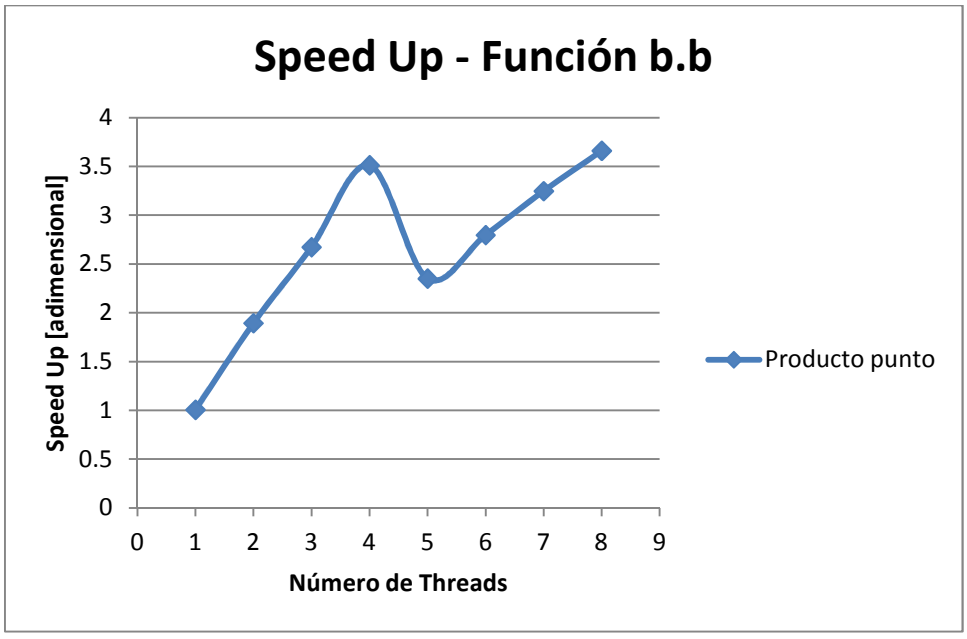
Los resultados numéricos del Speed Up calculado pueden apreciarse en la Tabla 2. Las gráficas del Speed Up alcanzado al aumentar los threads se aprecian en las ilustraciones 3 y 4. El máximo Speed Up alcanzado para la función A x b fue de 4.02, mientras que para la función Producto Interno fue de 3.66, ambos empleando los 8 threads disponibles en la máquina.

Ilustración 3. Speed Up del código paralelo para la función A x b



La gráfica de Speed Up que se presenta en la Ilustración 4 es muy similar a la Ilustración 3. La causa de este patrón de comportamiento no lineal podría explicarse por diversas causas. Algunas pueden ser el manejo que le da OpenMP a las instrucciones de paralelización del ciclo for, sobre el cual no se conoce mucho, o bien depender de factores como el ancho de banda compartido por los threads o la arquitectura de memoria disponible, todos ellos, dependientes de la máquina de computo que se emplea.

Ilustración 4. Speed Up del código paralelo para la función Producto Interno



6. DIFICULTADES ENCONTRADAS

- Al momento de realizar las pruebas en la máquina Intel del Departamento de Computación se halló que sólo disponía de 4 threads para los ejercicios. Al correr el programa paralelo con 4 threads y comparar con el mismo número de threads en la máquina de escritorio, se halló que la máquina de escritorio arrojaba mejores resultados de tiempo. Su diferencia no era grande, sin embargo, teniendo 8 threads disponibles en la máquina de escritorio se optó por emplearla en lugar de la máquina Intel del DC.

- Una dificultad radica en la medición de tiempos para el programa serial desarrollado en lenguaje C. Esto se solucionó empleando MPI para la medición de tiempos. Se desconoce si existe alguna contraindicación que desvirtúe el uso de MPI sólo para medir tiempos en un programa desarrollado para correr en forma serial.

- Nunca se empleó un programa manejador de versiones de los programas y de sus modificaciones. Esta herramienta no se aprendió a manejar y permanece pendiente.

- A pesar de haber obtenido cierta escalabilidad para las funciones evaluadas, se desconoce qué tan influyente sea el empleo de un computador de escritorio en el desempeño mostrado por las gráficas de Speed Up. Se conoce que OpenMP corre eficientemente en plataformas con Memoria Compartida, pero se desconocen las características que en este sentido tenga la máquina empleada para los ejercicios (DELL XPS L502X, procesador Intel Core i7 de segunda generación 2630QM).

7. CONCLUSIONES Y OBSERVACIONES

- El Speed Up máximo logrado con el algoritmo paralelo para la multiplicación Matriz por Vector fue de 4.02. El Speed Up logrado para la misma función del TP2 del curso fue de 8.06, aproximadamente el doble que el logrado con OpenMP. Ambos ejercicios fueron desarrollados con matrices y vectores de tamaños diferentes, de modo que no son comparables, sin embargo, los datos son útiles como punto de referencia para futuros ejercicios de estudio. Se debe tener en cuenta también que el tamaño de la matriz no es la única variable que puede afectar un cálculo de Speed Up. Otras como haber empleado máquinas distintas también es fundamental en este tipo de análisis.

- El presente trabajo se focalizó en el análisis del desempeño de las funciones de multiplicación Matriz por Vector y Producto Interno. No en el desempeño general del programa paralelo.

- La escalabilidad de un código paralelo también depende de la optimización del código secuencial. Si este segundo no está bien optimizado no se puede esperar que el código paralelo escale.

- Para evaluar el desempeño general del programa paralelo se debe tener en cuenta la Ley de Amdahl para tener en cuenta aquellas porciones del código que no pueden ser paralelizadas y adicionar este dato a la ecuación para calcular el máximo Speed Up alcanzable.

- Las mediciones de tiempos y Speed Up para 5, 6 y 7 threads fueron mayores que empleando 4 threads. La causa de este comportamiento oscilatorio es desconocida y contrasta con el hecho de que los mejores Speed Up obtenidos se lograron con 8 threads.

- Una manera de mejorar el desempeño de un código paralelo es diseñarlo de tal forma que la carga de todos los threads tienda a ser la misma durante un cálculo.

- El Speed Up obtenido es una variable que depende del problema específico a resolver.

- El máximo Speed Up obtenido fue de 4.02 empleando 8 threads. Para mejorarlo se deben analizar factores como el ancho de banda que los procesadores comparten y la arquitectura de memoria disponible.

8. REFERENCIAS BIBLIOGRÁFICAS

[1] Formato de almacenamiento de matrices del Matrix Market: http://math.nist.gov/MatrixMarket/mmio-c.html. Ultima fecha de consulta: 17 de octubre de 2012.

[2] PACHECO, Peter S. An introduction to parallel programming. Morgan Kaufmann. Elsevier Inc. 2011.

[3] Página de Internet: Colección de matrices dispersas de la Universidad de Florida. UF Sparse Matrix Collection: http://www.cise.ufl.edu/research/sparse/matrices/list_by_type.html. Ultima fecha de consulta: 17 de octubre de 2012.