Sistemas distribuidos y paralelos

Primer Entrega

Russell Brandon (810/4)

18/05/12

1. Aspectos Generales

Para la realización del trabajo se tuvieron consideraciones generales a la hora de implementar el codigo en C. Dichas consideraciones son las siguientes:

- **Utilización de macros:** reemplazadas funciones llamadas recurrentemente como **setValor()** y **getValor()** evitando así la penalización que ocurre al llamar funciones y produciendo un impacto significante en el rendimiento del programa.
- **Dimension y cantidad de threads por párametros:** Para la ejecución de los programas se deben establecer la cantidad de núcleos a utilizar y el tamaño N*N de la matriz.
- **Matrices alocadas dinámicamente:** Para evitar la limitación del tamaño del stack.
- **Multiplicación de matrices optimizada para caché de CPU:** Dado que la multiplicación de matrices se efectua recorriendo filas en la matriz derecha y columnas en la matriz izquierda, para aprovechar el principio de localidad espacial implementado por los procesadores a la hora de guardar datos en la cache, de esta forma reduciendose así los fallos de caché.
- **Globalización de variables frecuentemente utilizadas:** Para evitar desperdicio de RAM y establecimiento de las mismas multiples veces se hicieron globales ciertas variables necesarias para muchas funciones. Por ejemplo: N y T.

1.1. Hardware Utilizado

Para las mediciones de los tiempos de ejecución se utilizó la siguiente configuración de hardware:

- **CPU:** Intel core i5 6600, 3.30Ghz 6MB cache L3
- **Hyper Threading:** Desactivado
- **Turbo Frequency:** Desactivado
- **RAM:** 16 GB 2133Mhz DDR4 Dual Channel

Para una mejor aproximación de los tiempos de ejecución se corrieron cuatro veces los distintos programas con los mismos parámetros así luego promediandose.

2. Ejercicio 1

2.1. Enunciado

Resolver con Pthreads y OpenMP la siguiente expresión: R = AA Donde A es una matriz de NxN. Analizar el producto AA y utilizar la estrategia que proporcione el mejor tiempo de ejecucion. Evaluar N=512, 1024 y 2048.

2.2. Secuencial

Se compila el código de OpenMP pero sin sus librerías haciendo así de esta forma que la clausula *# pragma parallel* no tenga efecto alguno.

Tabla 1: Tiempos secuenciales ejercicio uno

N	Tiempo
512	0.375048
1024	3.046207
2048	24.410414

El código implementado se detallará a continuación en la siguiente sección.

2.2.1. Openmp

Utilizando la API OpenMP se implementaron dos funciones claves para la solución del ejercicio mismo, estas son las siguientes:

1. **void filasAColumnas(double *A, double *B):** Pasa una matriz A ordenada en filas a una matriz B ordenada en columnas.
2. **void mulMatrices(double *A, double *B, double *C):** Multiplica A*B y almacena el resultado en C. Siendo A ordenada por filas, B por columnas y el resultado C por filas.

Para la paralelización de dichas funciones se escribe previo al loop *for* la sentencia **# pragma omp parallel for**.

A continuación se muestran los macros utilizados junto con la implementación de las funciones mencionadas recientemente:

```
// Acceso y asignacion por filas
#define obtenerValorMatrizFila(M, F, C, N) (M[(F)*(N)+(C)])
#define asignarValorMatrizFila(M, F, C, N, VALOR) (M[(F)*(N)+(C)
    ↪ ] = (VALOR))
// Acceso y asignacion por columnas
#define obtenerValorMatrizColumna(M, F, C, N) (M[(F)+(N)*(C)])
```

```
#define asignarValorMatrizColumna(M, F, C, N, VALOR) (M[(F)+(N)
    ↪ *(C)] = (VALOR))
//Funcion para multiplicar matrices
void mulMatrices(double *A, double *B, double *C)
{
      double sum;
      int i, j, k;
      #pragma omp parallel for private(sum, j, k)
      for(i = 0; i < N; i++)
      {
            for(j = 0; j < N; j++)
            {
                  sum = 0.0;
                  for(k = 0; k < N; k++)
                  {
                        sum += obtenerValorMatrizFila(A, i, k, N) *
                           ↪ obtenerValorMatrizColumna(B, k, j, N);
                  }
                  asignarValorMatrizFila(C, i, j, N, sum);
            }
      }
}
// Funcion para pasar de filas a columnas
void filasAColumnas(double *A, double *B)
{
      #pragma omp parallel for
      for(int i = 0; i < N; i++)
      {
            for(int j = 0; j < N; j++)
            {
                  asignarValorMatrizColumna(B, i, j, N,
                     ↪ obtenerValorMatrizFila(A, i, j, N));
            }
      }
}
```

Luego de realizar la operación:

$$R = AA$$

E imprimir el tiempo que demoró en realizarse, se verifica que el resultado sea el correcto.

Tabla 2: Ejercicio 1: Tiempos, Speedup y eficiencia de Openmp

N	Threads	Tiempo	Speedup	Eficiencia
512	2	0.187717	1.99794371	0.99897186
1024	2	1.5273	1.99450468	0.99725234
2048	2	12.204757	2.00007374	1.00003687
512	4	0.097357	3.85229619	0.96307405
1024	4	0.771325	3.94931708	0.98732927
2048	4	6.129219	3.98263041	0.9956576

2.3. Pthreads

Para el caso de las librerías POSIX Threads la principal diferencia en la implementación es que no se debe de usar la clausula *# pragma omp parallel for* sino que se han de calcular los limites de las iteraciones y asignarles dichos límites a los threads, es decir en otras palabras, redistribuir la carga/componentes de la matriz a ser procesados por cada thread. También a diferencia de la API Openmp, se implementó una función llamada ***ejercicioUno(void *args)** la cual es la que será ejecutada por cada thread individualmente, ésta misma contiene el código a ejecutar para realizar la operación requerida por el ejercicio.

Los límites de cada thread son calculados previamente a asignarles su función y estos mismos son pasados cómo parametro a la función **void ejercicioUno(void args)**. Agregando así dos nuevos argumentos en las funciones:

- *mulMatrices(double *A, double *B, double *C, int start, int end)*
- *filasAColumnas(double *A, double *B, int start, int end)*

Para la sincronización al momento de cambiar de función a otra (de **void filasAColumnas** a **void mulMatrices**) se implementó una barrera de tipo **pthread_baerrir_t** threadBarrier.

Tanto la creación cómo el cerrado de los threads se realiza en el main mediante las funciones **pthread_create()** y **pthread_join()**.

A continuación se detalla el tipo de dato de los argumentos que recibe cada threads y la función que ejecutan:

```
typedef struct threads_args
{
    int start;
    int end;
}

void *ejercicioUno(void *args)
{
```

```
    threads_args *arg = (threads_args*) args;

    // Copia A en B pero ordenado por columnas
    filasAColumnas(A, B, arg->start, arg->end);

    pthread_barrier_wait(&barrier);

    // Realiza la multiplicacion
    mulMatrices(A, B, C, arg->start, arg->end);

}
```

Tabla 3: Ejercicio 1: Tiempos, Speedup y eficiencia de Pthreads

N	Threads	Tiempo	Speedup	Eficiencia
512	2	0.189519	1.9789467	0.98947335
1024	2	1.530631	1.99016419	0.99508209
2048	2	12.244246	1.99362329	0.99681165
512	4	0.10018	3.74374127	0.93593532
1024	4	0.771032	3.95081786	0.98770447
2048	4	6.163424	3.96052811	0.99013203

3. Ejercicio 2

3.1. Enunciado

Realizar un algoritmo Pthreads y otro OpenMP que resuelva la expresión:

$$M = \overline{u.l}AAC + \overline{b}LBE + \overline{b}DUF$$

Donde A, B, C, D, E y F son matrices de NxN. L y U son matrices triangulares de N*N inferior y superior, respectivamente. $\overline{b}$ es el promedio de los valores de los elementos de la matriz B y $\overline{u.l}$ es el producto de los promedios de los valores de los elementos de las matrices U y L, respectivamente. Evaluar N=512, 1024 y 2048.

3.2. Secuencial

Al igual que el ejercicio 1, se compila el código de Openmp pero sin las librerías OpenMP.

Tabla 4: Ejercicio 2: Tiempos secuenciales

N	Tiempo
512	1.912922
1024	15.497197
2048	124.156582

3.3. Openmp

Se reutilizan las dos funciones mencionadas del ejercicio uno y además se agregan las siguientes:

1. **escalarPorMatriz(double *A, double *C, double escalar, int length):** Multiplica una matriz A por un escalar y la almacena en una matriz C.
2. **void sumarMatrices(double *A, double *B, double *C, int length):** Suma dos matrices que estén ordenadas de la misma forma ya que internamete realiza la suma vectorialmente.
3. **double sumarMatriz(double *A, int length):** Suma todos los elementos de la matriz A y devuelve el resultado en formato double.
4. **void triangularInferiorPorCuadrada(double *L, double *A, double *C):** Multiplica una matriz triangular inferior L ordenada por filas por una matriz A ordenada por columnas y deja el resultado en una matriz C.
5. **void triangularSuperiorPorCuadrada(double *U, double *B, double *C):** Multiplica una matriz triangular superior U ordenada por filas por una matriz A ordenada por columnas y deja el resultado en una matriz C.

Para lograr distribuir correctamente la carga en las matrices triangulares (ya que si se divide la carga por columnas o filas dada la naturaleza de estas matrices quedaría desbalanceada la carga) se utiliza una carga dinamica con la clausula **schedule dynamic** como se muestra en las dos siguientes funciones:

```
void triangularSuperiorPorCuadrada(double *U, double *B, double
    ↪ *C)
{
    double sum;
    int i, j, k;
5   #pragma omp parallel for private(sum, j, k) schedule(
        ↪ dynamic, 64)
    for(i = 0; i < N; i++)
    {
        for(j = 0; j < N; j++)
        {
```

```
                sum = 0.0;
                for(k = i; k < N ; k++)
                {
                    sum += obtenerValorMatrizTriaSupFila(U, i, k
                        ↪ , N) * obtenerValorMatrizColumna(B, k,
                        ↪ j, N);
                }
                asignarValorMatrizFila(C, i, j, N, sum);
            }
        }
}

void triangularInferiorPorCuadrada(double *L, double *A, double
    ↪ *C)
{
    double sum;
    int i, j, k;
    #pragma omp parallel for private(sum, j, k) schedule(
        ↪ dynamic, 64)
    for(i = 0; i < N; i++)
    {
        for(j = 0; j < N; j++)
        {
            sum = 0.0;
            for(k = 0; k < i + 1; k++)
            {
                sum += obtenerValorMatrizTriaInfFila(L, i, k
                    ↪ ) * obtenerValorMatrizColumna(A, k, j,
                    ↪ N);
            }
            asignarValorMatrizFila(C, i, j, N, sum);
        }
    }
}
```

La secuencia en la que se resolvió la ecuación es la siguiente:

Primero se calcula el primer termino:

$$AC$$

$$AAC$$

$$\overline{u.l}AAC$$

Luego el segundo y tercer termino se realiza se la siguiente manera:

$$BE$$

$$LBE$$

$$UF$$

$$DUF$$

$$\overline{b}(LBE + DUF) = M$$

Y por ultimo:

$$M = M + \overline{u.l}AAC$$

Tabla 5: Ejercicio 2: Tiempos, Speedup y eficiencia de Openmp

N	Threads	Tiempo	Speedup	Eficiencia
512	2	0.970493	1.97108274	0.98554137
1024	2	7.822962	1.9809884	0.9904942
2048	2	62.263788	1.99404158	0.99702079
512	4	0.505946	3.78088175	0.94522044
1024	4	3.972467	3.90115185	0.97528796
2048	4	31.458613	3.94666421	0.98666605

3.4. Pthreads

Nuevamente a diferencia de Openmp se utiliza el recurso de las barreras para sincronizar a cada thread luego de realizar cada función de cálculo. Y así tambien se incorporan nuevos argumentos los cuales son pasados a los threads (limites de las matrices cuadradasy limites de las matrices triangulares tratadas como vectores) como se muestra a continuación :

```
typedef struct threads_args
{
    int start;
    int end;
    int vectStart;
    int vectEnd;
    int  triaStart;
    int  triaEnd;
}threads_args;
```

A la hora de calcular el promedio la función sumaMatriz pasa a ser una función de tipo *void* en la cual cada thread va actualizando una variable llamada *sumaPromedio* a travez de un mutex de tipo **pthread_mutext_t** el cual les garantiza exxclusión mutua a la hora de modificar su valor. La función es mostrada a continuación:

```
void sumarMatriz(double *M, int start, int end, pthread_mutex_t
    ↪ *mutex)
{
    double total = 0.0;

    for (int i = start; i <= end; i++)
    {
        total += M[i];
    }

     pthread_mutex_lock(mutex);
    sumaPromedio += total;
    pthread_mutex_unlock(mutex);
}
```

También como sucede con el ejercicio anterior, dado que hay que calcular los límites de los threads para que luego pasen como paremetros sus límites a las funciones, se deben agregar los argumentos (*int start, int end*) a todas las funciones mencionadas previamente. Resultando así la función que ejecutará cada thread:

```
void *ejercicioDos(void *args){

    threads_args *arg = (threads_args*) args;

    // Calcular promedios
    sumarMatriz(U,arg->triaStart,arg->triaEnd,&sumMutex);

    if(pthread_barrier_wait(&threadBarrier) ==
       ↪ PTHREAD_BARRIER_SERIAL_THREAD)
    {
        uAvg = sumaPromedio / (N*N);
        sumaPromedio = 0;
    }

    pthread_barrier_wait(&threadBarrier);

    sumarMatriz(L,arg->triaStart,arg->triaEnd,&sumMutex);
```

```
    if(pthread_barrier_wait(&threadBarrier) ==
       ↪ PTHREAD_BARRIER_SERIAL_THREAD)
    {
        lAvg = sumaPromedio / (N*N);
        sumaPromedio = 0;
    }

    pthread_barrier_wait(&threadBarrier);

    ulAvg = uAvg * lAvg;

    sumarMatriz(B,arg->vectStart,arg->vectEnd,&sumMutex);

    if(pthread_barrier_wait(&threadBarrier) ==
       ↪ PTHREAD_BARRIER_SERIAL_THREAD)
    {
        bAvg = sumaPromedio / (N*N);
        sumaPromedio = 0;
    }
    // Fin de calcular promedios
    pthread_barrier_wait(&threadBarrier);
    // Paso matriz C a columnas para hacer A*C
    filasAColumnas(C, tC, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    // Multiplicar AtC y guardar en TAC
    mulMatrices(A, tC, TAC, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);
    // Paso a columna nuevamente
    filasAColumnas(TAC, tTAC, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    // Multiplicar A por tTAC y guardar en TAAC
    mulMatrices(A, tTAC, TAAC, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    // Multiplicar ulAvg por TAAC y almacenar en ulTAAC
```

```
    escalarPorMatriz(TAAC, ulTAAC, ulAvg, arg->vectStart, arg->
        ↪ vectEnd);
    // ulTAAC ahora contiene el primer termino ordenado por
        ↪ filas
    pthread_barrier_wait(&threadBarrier);
    // Ordeno a E por columnas
    filasAColumnas(E, tE, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    // Multiplicar B por tE y almacenar en TBE
    mulMatrices(B, tE, TBE, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    filasAColumnas(TBE, tTBE, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    // Multiplicar L por tTBE (BE) y almacenar en TLBE
    triangularInferiorPorCuadrada(L, tTBE, TLBE, arg->start,
        ↪ arg->end);
    /// TLBE ahora contiene el segundo termino sin el escalar
        ↪ multiplicado
    pthread_barrier_wait(&threadBarrier);

    // Preparo F para ser multiplicada pasandola a columnas
    filasAColumnas(F,tF, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);

    // Multiplicar U por tF y almacenar en TUF
    triangularSuperiorPorCuadrada(U, tF, TUF, arg->start, arg->
        ↪ end);

    pthread_barrier_wait(&threadBarrier);

    filasAColumnas(TUF,tTUF, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);
```

```
    // Multiplicar D por tTUF y almacenar en TDUF (ordenado por
       ↪ filas)
    mulMatrices(D, tTUF, TDUF, arg->start, arg->end);

    pthread_barrier_wait(&threadBarrier);
    // Dado que TLBE y TDUF estan ordenadas por filas se puede
       ↪ sumar como un vector
    sumarMatrices(TLBE,TDUF,TLBEDUF, arg->vectStart, arg->
       ↪ vectEnd);

    pthread_barrier_wait(&threadBarrier);
    // Multiplico el escalar (promedio de B, bAvg) a la matriz
       ↪ resultante de la suma (TLBEDUF)
    escalarPorMatriz(TLBEDUF, M, bAvg, arg->vectStart, arg->
       ↪ vectEnd);
    // M ahora contiene el segundo y ultimo termino
    pthread_barrier_wait(&threadBarrier);

    // Sumar el primer termino haciendo asi que M tenga el
       ↪ resultado final
    sumarMatrices(M, ulTAAC, M, arg->vectStart, arg->vectEnd);
    //M ahora contiene el resutlado

}
```

Tabla 6: Ejercicio 2: Tiempos, Speedup y eficiencia de Pthreads

N	Threads	Tiempo	Speedup	Eficiencia
512	2	1.058434	1.80731345	0.90365672
1024	2	8.620706	1.79767144	0.89883572
2048	2	68.79332	1.80477671	0.90238836
512	4	0.559781	3.41726854	0.85431713
1024	4	4.520093	3.42851286	0.85712822
2048	4	38.809931	3.19909309	0.79977327

4. Ejercicio 3

4.1. Inciso

Paralelizar con OpenMP un algoritmo que cuente la cantidad de número pares en un vector de N elementos. Al finalizar, el total debe quedar en una variable llamada

pares. Evaluar con valores de N donde el algoritmo paralelo represente una mejora respecto al algoritmo secuencial.

4.2. Secuencial

Se corre el mismo código que el de Openmp pero ignorando las librerías.

Tabla 7: Ejercicio 3: Tiempos Secuenciales

N	Tiempo
200000000	0.40354
400000000	0.807082
800000000	1.613978

4.2.1. Openmp

Se utiliza nuevamente la clausula de los ejercicios previos (reduction(+:pares)) solo que ésta vez se introduce un condicional dentro del bucle.

```
int64_t pares = 0;
#pragma omp parallel for reduction(+:pares)
for (int64_t i = 0; i < N; i++)
{
    if(A[i] & 1 == 0)
    {
        pares += 1;
    }
}
```

Se utiliza la máscara en vez del operador módulo para optimizar el tiempo de ejecución del programa.

Tabla 8: Ejercicio 3: Tiempos, Speedup y eficiencia de Openmp

N	Threads	Tiempo	Speedup	Eficiencia
512	2	0.075738	1.96142667	0.98071333
1024	2	0.419602	1.9234465	0.96172325
2048	2	0.852536	1.89314938	0.94657469
512	4	0.109328	3.69109469	0.92277367
1024	4	0.209235	3.85729921	0.96432480
2048	4	0.408706	3.94899512	0.98724878

5. Conclusiones

Con los datos obtenidos por la ejecución de los códigos y presentados en las tablas se puede observar que:

- El rendimiento de ambas APIs (POSIX Threads y Openmp) es casi identico, a excepción del ejercicio 2 cuando se realiza la multiplicación con matrices triangulares, la diferencia es que en Openmp es sencillo de implementar una asignación de tarea dinámica, en cambio en Pthreads dada su complejidad se optó por realizar asignación estática de trabajo, resultando así una diferencia de rendimiento entre ambas implementaciones.
- También se puede observar que las operaciones aritmeticas de matrices son altamente paralelizables, por lo que el rendimiento aumenta considerablemente al aumentar los threads. Esto se debe a que son tareas sin necesidad de mucha comunicación entre ellas. Estas concluciones pueden observarse en la eficiencia que resulta de calcular $SpeedUp/NroThreads$ que en la mayoría de los casos es cercana a 1.