Informe
Primera Entrega

Sistemas Distribuidos y Paralelos

Juan Matías Mascioto 00292/4
Andrés José Soerensen Lopéz 00302/7

Enunciado:

Se quiere resolver el siguiente problema en tres etapas:

a. Dadas tres matrices A, B y D no nulas de NxN, donde D es una matriz diagonal. Se quiere obtener la matriz C mediante la multiplicación:

$$C=A.B.D$$

b. La matriz C debe ser multiplicada por el siguiente factor.

$$\frac{(maxA-minA)^2}{avgA} \cdot \frac{(maxB-minB)^2}{avgB}$$

Dónde:

- **maxA: es el máximo valor de la matriz A.**
- **minA: es el mínimo valor de la matriz A.**
- **avgA: es el valor promedio de los elementos de la matriz A.**
- **maxB: es el máximo valor de la matriz B.**
- **minB: es el mínimo valor de la matriz B.**
- **avgB: es el valor promedio de los elementos de la matriz B.**

c. A la matriz resultante de la etapa anterior se le aplican una serie de ordenaciones de las columnas de la siguiente forma:

I. **Se toma la columna 0 de C y se ordena en forma descendente.**
II. **Si durante la ordenación se mueve un elemento de la columna 0 a otra fila se deben mover todos los elementos de la fila con él siempre y cuando estén en columnas con índices mayores.**
III. **i El mismo proceso se debe aplicar a las siguientes columnas.**

Generalidades:

Tanto el algoritmo secuencial, como en pthread y omp, se dividió en tres etapas principales,cada una correspondiente a un inciso(a,b,c) de la entrega.

En el caso del algoritmo secuencial y el de omp, las llamadas a cada una de las etapas se hacen desde el programa principal. En el caso de pthread, cada thread hace una llamada propia a cada etapa.

A - Etapa 1:

Algoritmo secuencial: consta de tres *for* anidados que realizan la multiplicación de las matrices

```
// Multiplica A*B*D=C
        for(i=0;i<N;i++){
                        for(j=0;j<N;j++){
                                total=0;
                                for(k=0;k<N;k++){
                                        total+=A[i*N+k]*B[k*N+j];       // total=A*B
                                }
                                // D tiene una única fila llena por c/ columna
                                C[i*N+j] = total * D[j*N+j];            // C=total*D
                        }
        }
```

Como puede verse, aprovechamos que la matriz D es diagonal (un único valor de cada columna es distinto a cero) para evitar recorrer todas las posiciones de una columna en cada iteración, y accedemos directamente a la posición donde se encuentra el valor no nulo.

Algoritmo omp: se hace uso de la sentencia *#pragma omp parallel for* , con el objetivo de paralelizar la ejecución de la sentencia *for*. De forma tal que la multiplicación de las matrices se realiza entre varios threads.

```
#pragma omp parallel for private(i,j,k,total)
        for(i=0;i<N;i++){
                for(j=0;j<N;j++){
                        total=0;
                        for(k=0;k<N;k++){
                                total+=A[i*N+k]*B[k*N+j];       // total=A*B
                        }
```

```
                    // D tiene una única fila llena por c/ columna
                    C[i*N+j] = total * D[j*N+j];          // C=total*D
            }
    }
```

Algoritmo pthread: cada thread calcula el resultado para grupo de filas de la matriz C.Es decir, si la matriz C en de longitud 8 y se ejecutan 4 thread, cada thread calcula el resultado para dos filas de la matriz C.

```
// Filas que multiplica el thread
int cant_filas = N/CANT_THREADS;      // Cant de filas que multiplica cada thread
int fila_inicial = id*cant_filas;
int fila_final = fila_inicial + cant_filas -1;

// Multiplica A*B*D=C
for(i=fila_inicial;i<=fila_final;i++){          // Recorre solo algunas filas
                    for(j=0;j<N;j++){      // Recorre todas las columnas
                                total=0;
                                for(k=0;k<N;k++){
                                        total+=A[i*N+k]*B[k*N+j];       // total=A*B
                                }
                                // D tiene una única fila llena por c/ columna
                                C[i*N+j] = total * D[j*N+j];          // C=total*D
                    }
   }
```

B - Etapa 2:

Algoritmo secuencial: utilizando dos sentencias *for* anidadas, se recorre las matrices A y B, y se calculan los valores pedidos. Una vez hecho esto, se calcula un *factor* según la ecuación establecida en el enunciado, y acto seguido , se multiplica a la matriz C por dicho *factor*.

```
// Calcula valores para A y B
        for(i=0;i<N;i++){
                for( j=0;j<N;j++){
                        // Cálculos para A
                        actA = A [ i * N + j];
                        if(actA<minA){
                                minA=actA;                      // Actualiza mínimo
                        }
                        else{
```

```
                    if(actA>maxA){
                        maxA=actA;          // Actualiza máximo
                    }

                }
                totalA+=actA;       // Incrementa total

                // Cálculos para B
                actB = B [ i * N + j];
                if(actB<minB){
                    minB=actB;              // Actualiza mínimo
                }
                else{
                    if(actB>maxB){
                        maxB=actB;          // Actualiza máximo
                    }

                }
                totalB+=actB;       // Incrementa total
            }
        }


        // Promedios
        avgA=totalA/(N*N);
        avgB=totalB/(N*N);

        float factor=(((maxA-minA)*(maxA-minA))/avgA) * (((maxB-minB)*(maxB-minB))/avgB);       // Calcula factor
        // C*factor
        for(i=0;i<N;i++){
            for(j=0;j<N;j++){
                C[i*N+j]=C[i*N+j]*factor;
            }
        }
```

Algoritmo omp: al igual que el algoritmo secuencial se utilizan dos sentencias *for* anidadas, se recorre las matrices A y B, y se calculan los valores pedidos pero, con la diferencia que se hace uso de la sentencia *#pragma omp parallel for* , con el objetivo de paralelizar la ejecución de la sentencia *for*. De manera que , cada thread calcula un *minA* parcial, *maxA* parcial, etc. En consecuencia, es necesario comparar, los valores obtenidos por cada thread y calcular un resultado global. Esto se realiza de manera secuencial. Luego, en similitud con el algoritmo secuencial, se obtiene un *factor* según la ecuación establecida en el enunciado.
La multiplicación entre dicho *factor* y la matriz C se paraleliza, nuevamente valiéndose de la sentencia *#pragma omp parallel for* .

```
#pragma omp parallel for  firstprivate(maxA,minA,maxB,minB,totalA,totalB) private(actA,actB,i,j)
        for(i=0;i<N;i++){       // Calcula valores para A y B
            int id = omp_get_thread_num();
            for( j=0;j<N;j++){
                // Cálculos para A
                actA = A [ i * N + j];
                if(actA<minA){
                    parciales[id].minA=actA;                    // Actualiza mínimo
```

```
                        minA=actA;
                    }
                    if(actA>maxA){
                        parciales[id].maxA=actA;        // Actualiza máximo
                        maxA=actA;
                    }

                    parciales[id].totalA+=actA;      // Incrementa total

                    // Cálculos para B
                    actB = B [ i * N + j];
                    if(actB<minB){
                        parciales[id].minB=actB;                    // Actualiza mínimo
                        minB=actB;
                    }
                    if(actB>maxB){
                        parciales[id].maxB=actB;        // Actualiza máximo
                        maxB=actB;
                    }
                    parciales[id].totalB+=actB;      // Incrementa total
                }
            }


            basetype maxA_total = -1, minA_total = 99999999;
            basetype maxB_total = -1, minB_total = 99999999;
            basetype totalA_total = 0;
            basetype totalB_total = 0;
            basetype avgA, avgB;
            basetype factor;
            int m;

            // Calcula los valores totales
            for (m=0;m<CANT_THREADS;m++){
                if (parciales[m].maxA>maxA_total){
                    maxA_total =        parciales[m].maxA;
                }
                if (parciales[m].maxB>maxB_total){
                    maxB_total =        parciales[m].maxB;
                }
                if (parciales[m].minA<minA_total){
                    minA_total =        parciales[m].minA;
                }
                if (parciales[m].minB<minB_total){
                    minB_total =        parciales[m].minB;
                }
                totalA_total+=parciales[m].totalA;
                totalB_total+=parciales[m].totalB;


            }

            // Promedios
            avgA=totalA_total/(N*N);
            avgB=totalB_total/(N*N);

            factor=(((maxA_total-minA_total)*(maxA_total-minA_total))/avgA)*
(((maxB_total-minB_total)*(maxB_total-minB_total))/avgB);    // Calcula factor


            // -- C*factor --
            #pragma omp parallel for private (i,j)
```

```
for(i=0;i<=N;i++){
        for(j=0;j<N;j++){
                C[i*N+j]=C[i*N+j]*factor;
        }
}
```

Algoritmo pthread: de manera similar al algoritmo omp,cada thread se encarga de obtener un *minA*,*maxA*,*minB,etc*. parcial.Esto quiere decir, que cada thread recorre un conjunto de filas de la matriz A y un conjunto de filas de la matriz B.Continuando, es necesario, comparar los valores parciales de cada thread y obtener valores globales.El thread responsable de calcular estos valores es el thread cuyo *id* es 0. A su vez, este thread , también se encarga de calcular el *factor* correspondiente a esta etapa.Para finalizar, cada thread realiza una parte de la multiplicación entre la matriz C y el *factor*.

```
// Calcula valores para A y B
        for(i=fila_inicial;i<=fila_final;i++){          // Recorre solo algunas filas
                for( j=0;j<N;j++){                      // Recorre todas las columnas
                        // Cálculos para A
                        actA = A [ i * N + j];
                        if(actA<minA){
                                minA=actA;                      // Actualiza mínimo
                        }
                        if(actA>maxA){
                                maxA=actA;          // Actualiza máximo
                        }

                        totalA+=actA;       // Incrementa total

                        // Cálculos para B
                        actB = B [ i * N + j];
                        if(actB<minB){
                                minB=actB;                      // Actualiza mínimo
                        }
                        if(actB>maxB){
                                        maxB=actB;          // Actualiza máximo
                        }


                        totalB+=actB;       // Incrementa total
                }
        }

        // Actualiza variables globales con resultados parciales
        parciales[id].maxA=maxA;
        parciales[id].maxB=maxB;
        parciales[id].minA=minA;
        parciales[id].minB=minB;
        parciales[id].totalA=totalA;
        parciales[id].totalB=totalB;
        pthread_barrier_wait(&barrera);         // Barrera

        // El thread de ID=0 calcula los valores totales
        if (id==0){
                // Resetea variables locales para calcular los totales
                maxA = -1;
                minA = 99999999;
```

```
            maxB = -1;
            minB = 99999999;
            totalA = 0;
            totalB = 0;
            // Calcula los valores totales
            for (i=0;i<CANT_THREADS;i++){
                        if (parciales[i].maxA>maxA){
                                    maxA =    parciales[i].maxA;
                        }
                        if (parciales[i].maxB>maxB){
                                    maxB =    parciales[i].maxB;
                        }
                        if (parciales[i].minA<minA){
                                    minA =    parciales[i].minA;
                        }
                        if (parciales[i].minB<minB){
                                    minB =    parciales[i].minB;
                        }
                        totalA+=parciales[i].totalA;
                        totalB+=parciales[i].totalB;


            }

            // Promedios
            avgA=totalA/(N*N);
            avgB=totalB/(N*N);

            factor=(((maxA-minA)*(maxA-minA))/avgA) * (((maxB-minB)*(maxB-minB))/avgB);// Calcula factor
            factor_global = factor;

}

pthread_barrier_wait(&barrera);           // Barrera
// C*factor
for(i=fila_inicial;i<=fila_final;i++){          // Recorre solo algunas filas
            for(j=0;j<N;j++){
                        C[i*N+j]=C[i*N+j]*factor_global;
            }
}
```

C - Etapa 3:

Algoritmo secuencial: este algoritmo consiste en recorrer la matriz C,en sentido inverso, es decir, se recorre por columna, y por cada columna , se recorren todas sus filas.

Por cada columna de la matriz C, se recorren todas sus filas, buscando el valor máximo de la columna.Como se explica en el enunciado, se debe ir reordenando la columna para que los valores queden en forma decreciente.
A continuación, se intercambian la filas entre la correspondiente al nuevo máximo encontrado y la fila correspondiente a la posición para la cual se busco ese nuevo máximo. Para este intercambio se hace uso de una fila auxiliar.

```
for(j=0;j<N;j++){      // Recorre por columna
        for(i=0;i<N;i++){      // Recorre por filas
                actual = C[i*N+j];    // En cada iteración "actual" toma el valor de un nuevo elemento de la columna
(cambia de fila)
                pos_max = i;
                valor_max = actual;

                for (k=i+1;k<N;k++){// Recorre todas las posiciones restantes de la columna para buscar el máximo
                        if (C[k*N+j]>valor_max){
                                pos_max = k;          // Actualiza posición del máximo
                                valor_max = C[k*N+j];           // Actualiza valor del máximo
                        }
                }
                // Copia filas que luego swappea
                for (k=j;k<N;k++){
                        fila_actual [k] = C [i*N+k];
                        fila_max [k] = C [pos_max*N+k];
                }
                // Swappea filas
                for (k=j;k<N;k++){
                        C [i*N+k] = fila_max[k];
                        C [pos_max*N+k]= fila_actual[k];
                }
        }
}
```

Algoritmo omp: para este algoritmo, por momentos el trabajo se distribuye entre threads, y por otros momentos, se concentra en el thread *master* .
Todos los threads se encargan de buscar el máximo en una columna. Consecuentemente, se generan valores máximos parciales . El thread *master* es el encargado de comparar esos valores y obtener un máximo global. Una vez logrado esto, todos los threads se encargan de intercambiar la fila correspondiente al nuevo máximo encontrado y la fila correspondiente a la posición para la cual se busco ese nuevo máximo
Nuevamente se hace uso de de la sentencia *#pragma omp parallel for* . Y para la sincronización de la threads se utiliza la sentencia *#pragma omp barrier .*

```
for(j=0;j<N;j++){      // Recorre por columna
        #pragma omp parallel
        {
                int i,k,m;
                int id = omp_get_thread_num();
                basetype actual;
                cant_filas_restantes = N;
                cant_filas_x_thread = N/CANT_THREADS;
                cant_columnas_restantes= N-j;
                cant_columnas_x_thread = cant_columnas_restantes/CANT_THREADS;

                int columna_inicial = j+id*cant_columnas_x_thread;
                int columna_final;
                if(id==CANT_THREADS-1){
                        columna_final = N-1;
```

```
		}
		else{
			columna_final = columna_inicial + cant_columnas_x_thread -1;
		}
		for(i=0;i<N;i++){	// Recorre por filas
			int fila_inicial = i+id*cant_filas_x_thread;
			int fila_final;
			if(id==CANT_THREADS-1){
				fila_final = N-1;
			}
			else{
				fila_final = fila_inicial + cant_filas_x_thread -1;
			}
			if (fila_inicial<=fila_final){
				actual = C[fila_inicial*N+j];	// En c/ iteración "actual" es un nuevo elemento de la col (cambia de fila)
				maximos[id].posicion = fila_inicial;
				maximos[id].valor = actual;
				for (k=fila_inicial+1;k<=fila_final;k++){	// Recorre las pos restantes de la col para buscar el máx
					if (C[k*N+j]>maximos[id].valor){
						maximos[id].posicion = k;	// Actualiza posición del máx
						maximos[id].valor = C[k*N+j];	// Actualiza valor del máx
					}
				}
				}
				else{
					maximos[id].valor = -1;		// Para que no queden valores viejos
				}
				#pragma omp barrier // Barrera (espera a que cada thread encuentre su máximo local)

				#pragma omp master // Master calcula el maximo global (valor y posición)
				{
					valor_max = -1;
					for (m=0;m<CANT_THREADS;m++){
						if (maximos[m].valor>valor_max){
							valor_max = maximos[m].valor;
							pos_max = maximos[m].posicion;
						}
					}
					cant_filas_restantes--;			// Decrementa la cantidad de filas restantes
					cant_filas_x_thread = cant_filas_restantes/CANT_THREADS;
				}

				#pragma omp barrier // Barrera (espera a que el thread 0 calcule el máximo global)

				// Copia filas que luego swappea
				// Cada thread copia un conj de columnas (colu inicial=fila_inicial y col final=fila_final)
				for (k=columna_inicial;k<=columna_final;k++){
					fila_actual [k] = C [i*N+k];
					fila_max [k] = C [pos_max*N+k];
				}

				// Swappea filas
				for (k=columna_inicial;k<=columna_final;k++){
					C [i*N+k] = fila_max[k];
					C [pos_max*N+k]= fila_actual[k];
				}

				#pragma omp barrier
				#pragma omp master
```

```
                        {
                                //printf("Matriz C - Ordenación columna %d \n",j);
                                //imprimir_matriz(C,N);
                        }
                }

        }
}
```

Algoritmo pthread: la lógica de este algoritmo es similar al algoritmo de omp. Por momentos el trabajo se distribuye entre threads, y por otros momentos, se concentra en en un único thread, es este caso el thread cuyo *id* es 0.

Todos los threads se encargan de buscar el máximo en una columna. Consecuentemente, se generan valores máximos parciales , los cuales son colocados en una arreglo (global) de máximos. El thread cuyo *id* es 0 es el encargado de comparar esos valores y obtener un máximo global. Una vez logrado esto, todos los threads se encargan de intercambiar la fila correspondiente al nuevo máximo encontrado y la fila correspondiente a la posición para la cual se busco ese nuevo máximo.

```
int i,j,k,m;
int id = (*parametro).id;
basetype fila_actual[N];
basetype fila_max [N];
basetype valor_max = -1;
basetype actual;
cant_filas_restantes= N;                    // Cantidad de filas que restan ordenar
cant_filas_x_thread = N/CANT_THREADS;       // Cantidad de filas que procesa cada thread en cada iteración (va disminuyendo)

int cant_columnas_restantes= N;             // Cantidad de filas que restan ordenar
int cant_columnas_x_thread = N/CANT_THREADS;        // Cantidad de filas que procesa cada thread en cada iteración (va disminuyendo)

int columna_inicial, columna_final;
int fila_inicial, fila_final;

for(j=0;j<N;j++){       // Recorre por columna
        basetype actual;
        cant_filas_restantes = N;
        cant_filas_x_thread = N/CANT_THREADS;
        cant_columnas_restantes= N-j;
        cant_columnas_x_thread = cant_columnas_restantes/CANT_THREADS;
        columna_inicial = j+id*cant_columnas_x_thread;

        if(id==CANT_THREADS-1){
                columna_final = N-1;
        }
        else{
                columna_final = columna_inicial + cant_columnas_x_thread -1;
```

```
	}
	for(i=0;i<N;i++){	// Recorre por filas
		fila_inicial = i+id*cant_filas_x_thread;
		if(id==CANT_THREADS-1){
			fila_final = N-1;
		}
		else{
			fila_final = fila_inicial + cant_filas_x_thread -1;
		}
		if (fila_inicial<=fila_final){
			actual = C[fila_inicial*N+j];	// En c/ iteración "actual" es un nuevo elemento de la col (cambia
de fila)

			maximos[id].posicion = fila_inicial;
			maximos[id].valor = actual;
			//printf("Columna %d \t Actual: %f\n",j,maximos[id].valor);
			for (k=fila_inicial+1;k<=fila_final;k++){	// Recorre las pos restantes de la col para buscar el
máx
				if (C[k*N+j]>maximos[id].valor){
					maximos[id].posicion = k;	// Actualiza posición del máx
					maximos[id].valor = C[k*N+j];	// Actualiza valor del máx

				}
			}
		}
		else{
			maximos[id].valor = -1;			// Para que no queden valores viejos
		}

		pthread_barrier_wait(&barrera);		// Barrera (espera a que cada thread encuentre su máximo local)

		if (id==0){
			valor_max = -1;
			for (m=0;m<CANT_THREADS;m++){
				if (maximos[m].valor>valor_max){
					valor_max = maximos[m].valor;
					pos_max_global = maximos[m].posicion;
				}
			}
			cant_filas_restantes--;			// Decrementa la cantidad de filas restantes
			cant_filas_x_thread = cant_filas_restantes/CANT_THREADS;

		}

		pthread_barrier_wait(&barrera);		// Barrera (espera a que el thread 0 calcule el máximo global)

		int pos1,pos2;
		int pos_max_global_local = pos_max_global;
		basetype aux;

		// Copia filas que luego swappea
		// Cada thread copia un conj de columnas (colu inicial=fila_inicial y col final=fila_final)
		for (k=columna_inicial;k<=columna_final;k++){
			pos1=i*N+k;
			pos2=pos_max_global*N+k;
			fila_actual [k] = C [pos1];
			fila_max [k] = C [pos2];
		}


		// Swappea filas
		for (k=columna_inicial;k<=columna_final;k++){
```

```
                pos1=i*N+k;
                pos2=pos_max_global*N+k;
                C [pos1] = fila_max[k];
                C [pos2]= fila_actual[k];
        }
        pthread_barrier_wait(&barrera);          // Barrera (espera a que el thread 0 calcule el máximo global)

        }          // Fin recorrido por fila



}
```

Aclaración: al momento de comparar los tiempos medidos de los algoritmos, se notó que el tiempo de ejecución del algoritmo de pthread (principalmente la etapa 3) fue notablemente superior al tiempo medido para los otros dos algoritmos. No se logró discernir la causa por la cual esto sucede.

Mediciones de tiempo:

	Secuencial	OpenMP			PThreads		
N	Tiempo	2 THREADS	4 THREADS	8 THREADS	2 THREADS	4 THREADS	8 THREADS
512	1.465879	0.9977	0.723757	11.3837	10.8838	10.4141	28.6408
1024	24.866819	13.1589	7.768355	49.8501	47.6557	43.0674	118.925
2048	233.484564	125.0132	77.190529	317.5245	302.4351	220.4302	450.6563

Tablas de Speed Up:

OpenMP			
N	2 THREADS	4 THREADS	8 THREADS
512	1.469258294	2.025374539	0.1287699957
1024	1.889733868	3.201040503	0.498831878
2048	1.867679285	3.024782535	0.7353277117

PThreads			
N	2 THREADS	4 THREADS	8 THREADS
512	0.1346844852	0.140759067	0.05118149633
1024	2.284755233	0.5773930862	0.2090966491
2048	0.7720154307	1.059222212	0.5180989681

Eficiencia:

OpenMP			
N	**2 THREADS**	**4 THREADS**	**8 THREADS**
512	0.3673145735	0.5063436347	0.03219249892
1024	0.4724334671	0.8002601259	0.1247079695
2048	0.4669198213	0.7561956338	0.1838319279

PThreads			
N	**2 THREADS**	**4 THREADS**	**8 THREADS**
512	0.0336711213	0.03518976676	0.01279537408
1024	0.5711888081	0.1443482715	0.05227416229
2048	0.1930038577	0.264805553	0.129524742