Informe primera entrega Sistemas Distribuidos y Paralelos

Programación con Memoria compartida

Alumnos:

- Santiago Valdovinos 702/1
- Antonio Arcuri 538/8

Ejercicio 1

Programa Pthreads

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <pthread.h>

#define COMPARAR_SECUENCIAL

// Tipo de los elementos en los vectores
// Compilar con -D_INT_ para vectores de tipo entero
// Compilar con -D_DOUBLE_ para vectores de tipo double
// Predeterminado vectores de tipo float

#ifdef _INT_
typedef int basetype;      // Tipo para elementos: int
#define tipo_dato   "ints"
#elif _DOUBLE_
typedef double basetype;  // Tipo para elementos: double
#define tipo_dato    "doubles"
#else
typedef float basetype;    // Tipo para elementos: float     PREDETERMINADO
#define tipo_dato    "floats"
#endif

// Tipo de dato que se pasa de parámetro a los threads
typedef struct param
{
    int id;
} param;


// -- Variables globales --

// Matrices
basetype *A;
basetype *B;
basetype *C;     // Matriz resultado
```

```
int cant_filas_restantes;   // Cantidad de filas que restan ordenar
int cant_filas_x_thread;    // Cantidad de filas que procesa cada thread en cada
iteración (va disminuyendo)
int pos_max_global;

int N;
int CANT_THREADS;

double speedup;
double eficiencia;
double tiempo_paral;
double tiempo_sec;

basetype factor_global;

pthread_barrier_t barrera;      // Barrerra

// -- Definición de funciones --

void *funcion_threads(void *arg);
void multiplicacion(param *parametro);
void imprimir_matriz (basetype * matriz,int N);
double dwalltime();
double tiempo_copia_total=0;

#ifdef COMPARAR_SECUENCIAL
void multiplicacion_secuencial(basetype *A,basetype *B,basetype *C_secuencial,int
N);

void verificar_resultado(basetype *C,basetype *C_secuencial,int N);

basetype *C_secuencial; // Matriz resultado
#endif

int main(int argc,char *argv[]){
    int i;
    int j;

    if (argc != 3) {
        printf("Error en la cantidad de parámetros. Parametro 1 -> long matriz
Parametro 2 -> n° threads");
```

```
        printf("Ingresar la dimension de la matriz!\n");
        return 0;
    }


    N = atoi(argv[1]);  // Dimensión de la matriz: N*N
    CANT_THREADS = atoi(argv[2]);

    printf("Dimensión de la matriz: %d*%d \n",N,N);


    // Reserva de memoria para las matrices
    A=(basetype*)malloc(sizeof(basetype)*N*N);          // Reserva memoria para A
    B=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para B
    C=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para C

    #ifdef COMPARAR_SECUENCIAL
    C_secuencial=(basetype*)malloc(N*N*sizeof(basetype));           // Reserva
memoria para C
    #endif


    // -- Inicialización ALEATORIA de las matrices --

    // Inicializa matrices A y B
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            A[i*N+j]=rand()%5;      // Inicializa matriz A con random
            //A[i*N+j]=1;   // Inicializa matriz A con unos
            B[i*N+j]=rand()%5;  // Inicializa matriz B con random
            //B[i*N+j]=1;   // Inicializa matriz B con unos
        }
    }


    param parametros[CANT_THREADS]; // Arreglo de param (struct que contiene los
datos para pasar a los threads)

    // -- Inicialización de threads --
    pthread_t threads[CANT_THREADS];    // Arreglo de threads

    if (pthread_barrier_init(&barrera,NULL,CANT_THREADS)!=0) {
```

```
            printf("Error creacion de barrera\n");
        return 0;
       }

    // Inicialización de parámetros
    for (i=0;i<CANT_THREADS;i++){
        parametros[i].id=i;
    }
    double tiempo_inicial=dwalltime();
    // Creacion de los threads
    for (i=0;i<CANT_THREADS;i++){
           if (pthread_create(
&threads[i],NULL,funcion_threads,(void*)&parametros[i])!=0) {
               printf("Error creacion de thread\n");
            return 0;
           }
    }

    // Join de los threads
    for(i = 0; i < CANT_THREADS; i++) {
           pthread_join(threads[i], NULL);

       }
            tiempo_paral = dwalltime()-tiempo_inicial;
       printf("\nTiempo Total (pthreads) : %f\n\n",dwalltime()-tiempo_inicial);

 #ifdef COMPARAR_SECUENCIAL
    tiempo_inicial=dwalltime();
    multiplicacion_secuencial(A,B,C_secuencial,N);  // C_secuencial = A*B
    tiempo_sec = dwalltime()-tiempo_inicial;
    printf("-- Fin de multiplicacion (secuencial) -->> \t Tiempo: %f \n",
tiempo_sec);

    speedup = tiempo_sec / tiempo_paral;
    printf("-- Speedup conseguido: %f \n", speedup);
    eficiencia = speedup / CANT_THREADS;
    printf("-- Eficiencia: %f \n", eficiencia);

    free(C_secuencial);
    #endif
```

```
    // Libera memoria
    free(A);
   free(B);
   free(C);

    return(0);

}

// ------------------------
// -- FUNCIONES PTHREADS //
// ------------------------


void *funcion_threads(void *arg) {
    param* parametro = (param*)arg;
    double tiempo_inicial2;
    //printf("Mi ID es: %d \n",(*parametro).id);
    if ((*parametro).id==0){
        tiempo_inicial2=dwalltime();
    }
    multiplicacion(parametro);
    pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen
    if ((*parametro).id==0){
        printf("-- Fin de multiplicacion (pthread) -->> \t Tiempo: %f
\n",dwalltime()-tiempo_inicial2);
        //printf("Matriz A:\n");
        //imprimir_matriz(A, N);
        //printf("Matriz B:\n");
        //imprimir_matriz(B, N);
        //printf("Matriz C (C=A*B):\n");
        //imprimir_matriz(C, N);
    }
    pthread_exit(NULL);
}

void multiplicacion(param* parametro){
    //printf("Comienzo etapa 1\n");
    int id = (*parametro).id;
    basetype total;
    int i,j,k;
```

```
    // 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;

    //printf("ID: %d \t fila_inicial=%d \t fila_final=%d
\n",id,fila_inicial,fila_final);

    // Multiplica A*B=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
                }
                C[i*N+j] = total;
            }
    }

}


// ---------------------------
// -- FUNCIONES AUXILIARES //
// ---------------------------

void imprimir_matriz (basetype * matriz,int N){
    int i;
    int j;
    for (i=0;i<N;i++){
        for (j = 0 ; j < N ; j++){
            printf ("%.1f\t",matriz [ i * N + j ]);
        }
        printf("\n");
    }
}

double dwalltime(){
       double sec;
       struct timeval tv;
```

```
        gettimeofday(&tv,NULL);
        sec = tv.tv_sec + tv.tv_usec/1000000.0;
        return sec;
}

// ----------------------------
// -- FUNCIONES SECUENCIALES //
// ----------------------------

#ifdef COMPARAR_SECUENCIAL
void multiplicacion_secuencial(basetype *A,basetype *B,basetype *C,int N){
    //printf("Comienzo etapa 1\n");

    basetype total;
    int i,j,k;

    // 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
                }
                C[i*N+j] = total;        // C=total
            }
       }

}

void verificar_resultado(basetype *C,basetype *C_secuencial,int N){
    int i,j;
    int check = 1;
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            check=check&&(C[i*N+j]==C_secuencial[i*N+j]);
        }
    }

    if(check){
        printf("Resultado correcto\n");
    }
    else{
```

```
        printf("Resultado erroneo \n");
    }
}
#endif
```

Programa OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <pthread.h>

#define COMPARAR_SECUENCIAL

// Tipo de los elementos en los vectores
// Compilar con -D_INT_ para vectores de tipo entero
// Compilar con -D_DOUBLE_ para vectores de tipo double
// Predeterminado vectores de tipo float

#ifdef _INT_
typedef int basetype;      // Tipo para elementos: int
#define tipo_dato   "ints"
#elif _DOUBLE_
typedef double basetype;  // Tipo para elementos: double
#define tipo_dato    "doubles"
#else
typedef float basetype;    // Tipo para elementos: float     PREDETERMINADO
#define tipo_dato    "floats"
#endif

// Tipo de dato que se pasa de parámetro a los threads
typedef struct param
{
    int id;
} param;


// -- Variables globales --

// Matrices
basetype *A;
```

```
basetype *B;
basetype *C;    // Matriz resultado
basetype *D;
basetype *E;
basetype *F;
basetype *L;
basetype *U;
basetype *sumaParcialB;

basetype promB;
basetype promL;
basetype promU;
basetype prodLU;



int cant_filas_restantes;   // Cantidad de filas que restan ordenar
int cant_filas_x_thread;    // Cantidad de filas que procesa cada thread en cada
iteración (va disminuyendo)
int pos_max_global;

int N;
int NT;
int CANT_THREADS;

double speedup;
double eficiencia;
double tiempo_paral;
double tiempo_sec;

basetype factor_global;

pthread_barrier_t barrera;      // Barrerra

// -- Definición de funciones --

void *funcion_threads(void *arg);
void multiplicacion(param *parametro);
void promedioB(param* parametro);
void prodPromLU(param* parametro);
void imprimir_matriz (basetype * matriz,int N);
double dwalltime();
```

```
//-- caca
double tiempo_copia_total=0;

#ifdef COMPARAR_SECUENCIAL
void multiplicacion_secuencial(basetype *A,basetype *B,basetype *C_secuencial,int
N);

void verificar_resultado(basetype *C,basetype *C_secuencial,int N);

basetype *C_secuencial; // Matriz resultado
#endif


int main(int argc,char *argv[])
{
    int i;
    int j;

    if (argc != 3) {
        printf("Error en la cantidad de parámetros. Parametro 1 -> long matriz
Parametro 2 -> n° threads");
        printf("Ingresar la dimension de la matriz!\n");
        return 0;
    }


    N = atoi(argv[1]);  // Dimensión de la matriz: N*N
    NT = (N*(N+1))>>1;
    CANT_THREADS = atoi(argv[2]);


    printf("Dimensión de la matriz: %d*%d \n",N,N);

    //Vector usado para guardar las sumas parciales de la matriz B para luego hacer
el promedio
    sumaParcialB = (basetype*)malloc(sizeof(basetype)*CANT_THREADS);
    basetype promB;

    // Reserva de memoria para las matrices
```

```
    A=(basetype*)malloc(sizeof(basetype)*N*N);          // Reserva memoria para A
    B=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para B
    C=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para C
    D=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para D
    E=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para E
    F=(basetype*)malloc(N*N*sizeof(basetype));          // Reserva memoria para F

    // -- caso especial de matrices triangulares
    L=(basetype*)malloc(NT*sizeof(basetype));           // Reserva memoria para L
    U=(basetype*)malloc(NT*sizeof(basetype));           // Reserva memoria para U

    #ifdef COMPARAR_SECUENCIAL
    C_secuencial=(basetype*)malloc(N*N*sizeof(basetype));           // Reserva
memoria para C_SECUENCIAL
    #endif


    // -- Inicialización ALEATORIA de las matrices --

    // Inicializa matrices A y B
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            A[i*N+j]=rand()%5;      // Inicializa matriz A con random
            B[i*N+j]=rand()%5;  // Inicializa matriz B con random
            C[i*N+j]=rand()%5;  // Inicializa matriz B con random
            D[i*N+j]=rand()%5;  // Inicializa matriz B con random
            E[i*N+j]=rand()%5;  // Inicializa matriz B con random
            F[i*N+j]=rand()%5;  // Inicializa matriz B con random
        }
    }

    // -- Inicializacion de matrices triangulares superior por columnas e inferior
por filas

    for (int i = 0; i < NT; ++i)
    {
        for (int j = 0; j < NT; ++j)
        {
            U[i+j*(j+1)/2]=rand()%5;
            L[i+N*j - i*(i+1)/2]=rand()%5;
        }
    }
```

```
    param parametros[CANT_THREADS]; // Arreglo de param (struct que contiene los
datos para pasar a los threads)

    // -- Inicialización de threads --
    pthread_t threads[CANT_THREADS];    // Arreglo de threads

    if (pthread_barrier_init(&barrera,NULL,CANT_THREADS)!=0) {
        printf("Error creacion de barrera\n");
        return 0;
    }

    // Inicialización de parámetros
    for (i=0;i<CANT_THREADS;i++){
        parametros[i].id=i;
    }
    double tiempo_inicial=dwalltime();
    // Creacion de los threads
    for (i=0;i<CANT_THREADS;i++){
        if (pthread_create(
&threads[i],NULL,funcion_threads,(void*)&parametros[i])!=0) {
            printf("Error creacion de thread\n");
            return 0;
        }
    }

    // Join de los threads
    for(i = 0; i < CANT_THREADS; i++)
    {
        pthread_join(threads[i], NULL);

    }

    tiempo_paral = dwalltime()-tiempo_inicial;
    printf("\nTiempo Total (pthreads) : %f\n\n",dwalltime()-tiempo_inicial);

    #ifdef COMPARAR_SECUENCIAL
        tiempo_inicial=dwalltime();
        multiplicacion_secuencial(A,B,C_secuencial,N);  // C_secuencial = A*B
```

```
        tiempo_sec = dwalltime()-tiempo_inicial;
        printf("-- Fin de multiplicacion (secuencial) -->> \t Tiempo: %f \n",
tiempo_sec);

        speedup = tiempo_sec / tiempo_paral;
        printf("-- Speedup conseguido: %f \n", speedup);
        eficiencia = speedup / CANT_THREADS;
        printf("-- Eficiencia: %f \n", eficiencia);

    free(C_secuencial);
    #endif


    // Libera memoria
    free(A);
    free(B);
    free(C);
    free(D);
    free(E);
    free(F);
    free(L);
    free(U);

    return(0);

}

// ------------------------
// -- FUNCIONES PTHREADS //
// ------------------------


void *funcion_threads(void *arg) {
    param* parametro = (param*)arg;
    double tiempo_inicial2;
    //printf("Mi ID es: %d \n",(*parametro).id);
    if ((*parametro).id==0){
        tiempo_inicial2=dwalltime();
    }
    multiplicacion(parametro);
    pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen
    if ((*parametro).id==0){
```

```
        printf("-- Fin de multiplicacion (pthread) -->> \t Tiempo: %f
\n",dwalltime()-tiempo_inicial2);
        //printf("Matriz A:\n");
    }
    pthread_exit(NULL);
}

void multiplicacion(param* parametro)
{
    //printf("Comienzo etapa 1\n");
    int id = (*parametro).id;
    basetype total;
    int i,j,k;


    // 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;

    //printf("ID: %d \t fila_inicial=%d \t fila_final=%d
\n",id,fila_inicial,fila_final);

    // Multiplica A*B=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
            }
            C[i*N+j] = total;
        }
    }
}



void prodPromLU(param* parametro)
{
```

```
    int id = (*parametro).id;
    basetype total;
    int i,j,k;


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

    total=0;

    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
            }
            C[i*N+j] = total;
        }
    }
}


void promedioB(param* parametro)
{
    int id = (*parametro).id;
    basetype total;
    int i,j,k;


    // 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;

    //printf("ID: %d \t fila_inicial=%d \t fila_final=%d
\n",id,fila_inicial,fila_final);
    total=0;
```

```
    // Multiplica A*B=C
    for(i=fila_inicial;i<=fila_final;i++)
    {                                       // Recorre solo algunas filas
        for(j=0;j<N;j++)                    // Recorre todas las columnas
        {
            total+=B[i*N + j];
        }
    }
    sumaParcialB[id]=total;
   pthread_barrier_wait(&barrera); //Espera a que todas terminen
   if(id == 0)                     //si es el hilo principal
   {
       for (int i = 0; i < CANT_THREADS; ++i)
       {
           promB+=sumaParcialB[i];
       }
       promB /=CANT_THREADS;
   }
}

// --------------------------
// -- FUNCIONES AUXILIARES //
// --------------------------

void imprimir_matriz (basetype * matriz,int N){
    int i;
    int j;
    for (i=0;i<N;i++){
        for (j = 0 ; j < N ; j++){
            printf ("%.1f\t",matriz [ i * N + j ]);
        }
        printf("\n");
    }
}

double dwalltime(){
    double sec;
    struct timeval tv;

    gettimeofday(&tv,NULL);
    sec = tv.tv_sec + tv.tv_usec/1000000.0;
    return sec;
```

```
}

// ----------------------------
// -- FUNCIONES SECUENCIALES //
// ----------------------------

#ifdef COMPARAR_SECUENCIAL
void multiplicacion_secuencial(basetype *A,basetype *B,basetype *C,int N){
    //printf("Comienzo etapa 1\n");

    basetype total;
    int i,j,k;

    // 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
                }
                C[i*N+j] = total;        // C=total
            }
        }

    }

    void verificar_resultado(basetype *C,basetype *C_secuencial,int N){
        int i,j;
        int check = 1;
        for(i=0;i<N;i++){
            for(j=0;j<N;j++){
                check=check&&(C[i*N+j]==C_secuencial[i*N+j]);
            }
        }

        if(check){
            printf("Resultado correcto\n");
        }
        else{
            printf("Resultado erroneo \n");
        }
    }
```

```
#endif
```

Mediciones

Pthreads

		N Threads = 2	N Threads = 4	Secuencial
N = 512	Tiempo	0,2669	0,1600	0,5232
	Speedup	1,9590	3,3690	
	Eficiencia	0,9790	0,8422	
N = 1024	Tiempo	7,2543	3,3640	12,0385
	Speedup	1,6594	3,5715	
	Eficiencia	0,8297	0,8928	
N = 2048	Tiempo	83,0381	35,6572	133,9773
	Speedup	1,6134	4,0180	
	Eficiencia	0,8067	1,0045	

OpenMP

		N Threads = 2	N Threads = 4	Secuencial
N = 512	Tiempo	0,3337	0,1692	0,5121
	Speedup	1,5346	3,0885	
	Eficiencia	0,7673	0,7721	
N = 1024	Tiempo	2,6457	1,3389	7,9236

	Speedup	2,9948	5,9527	
	Eficiencia	1,4974	1,4881	
N = 2048	Tiempo	21,3450	10,7081	123,7945
	Speedup	5,6276	11,5607	
	Eficiencia	2,8138	2,8901	

Ejercicio 3

Programa secuencial:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

double dwalltime();
double initial_time;
double final_time;

int main()
{
   int i, size, even;

   /* Input size of the array */
   printf("Enter size of the array: ");
   scanf("%d", &size);
   int arr[size];

   /* Fill array with random elements */
   for(i=0; i<size; i++)
   {
       int r = rand();
       arr[i] = r;
   }
```

```
    even = 0;

    initial_time = dwalltime();
    for(i=0; i<size; i++)
    {
        /* If the current element of array is even then increment even count */
        if(arr[i]%2 == 0)
        {
            even++;
        }
    }
    final_time = dwalltime() - initial_time;

    printf("Total even elements: %d\n", even);
    printf("Total time: %f\n", final_time);

    return 0;
}

double dwalltime(){
     double sec;
     struct timeval tv;

     gettimeofday(&tv,NULL);
     sec = tv.tv_sec + tv.tv_usec/1000000.0;
     return sec;
}
```

Programa paralelo OpenMP:

```
#include <omp.h>
#include <stdlib.h>
#include <sys/time.h>
#include <stdio.h>

#define CANT_THREADS 2

double dwalltime();
```

```c
double initial_time;
double final_time;

int main()
{

   int total_even_array[CANT_THREADS];
   int i, size, even, total_even;

   omp_set_num_threads(CANT_THREADS);

   /* Input size of the array */
   printf("Enter size of the array: ");
   scanf("%d", &size);
   int arr[size];

   /* Fill array with random elements */
   for(i=0; i<size; i++)
   {
       int r = rand();
       arr[i] = r;
   }

   int iam =0, np = 1, j=0;
   initial_time = dwalltime();

   #pragma omp parallel private(iam, np, j, even)
   {
       #if defined (_OPENMP)
           np = omp_get_num_threads();
           iam = omp_get_thread_num();
       #endif

       printf("Hello from thread %d out of %d\n",iam,np);

       even = 0;
       #pragma omp for schedule(static)

       for(j=0; j<size; j++)
       {
           /* If the current element of array is even then increment even count */
           if(arr[j]%2 == 0)
```

```
            {
                even++;
            }
        }

        total_even_array[iam] = even;

    }

    total_even = 0;

    for (int k=0; k < CANT_THREADS; k++ ) {
        total_even += total_even_array[k];
        printf("Position %d and value %d\n",k,total_even_array[k]);
    }

    final_time = dwalltime() - initial_time;
    printf("Total time: %f\n", final_time);
    printf("Total even : %d\n", total_even);

}

double dwalltime(){
    double sec;
    struct timeval tv;

    gettimeofday(&tv,NULL);
    sec = tv.tv_sec + tv.tv_usec/1000000.0;
    return sec;
}
```

Mediciones:

		N Threads = 2	N Threads = 4	Secuencial
N = 65.536	Tiempo	0,000884	0,001169	0,001536
	Speedup	0,575520	0,761067	

	Eficiencia	0,286110	0,190266	
N = 131.072	Tiempo	0,000945	0,001328	0,001986
	Speedup	0,475830	0,668680	
	Eficiencia	0,237915	0,167170	
N = 262.144	Tiempo	0,001464	0,001205	0,003864
	Speedup	0,378881	0,311853	
	Eficiencia	0,189440	0,077963	

Ejercicio 3 (solo parte Pthreads sin cálculos serán enviados pronto)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <pthread.h>

#define COMPARAR_SECUENCIAL

// Tipo de los elementos en los vectores
// Compilar con -D_INT_ para vectores de tipo entero
// Compilar con -D_DOUBLE_ para vectores de tipo double
// Predeterminado vectores de tipo float

#ifdef _INT_
typedef int basetype;  // Tipo para elementos: int
#define tipo_dato   "ints"
#elif _DOUBLE_
typedef double basetype;  // Tipo para elementos: double
#define tipo_dato       "doubles"
#else
typedef float basetype;   // Tipo para elementos: float         PREDETERMINADO
#define tipo_dato       "floats"
#endif

// Tipo de dato que se pasa de parámetro a los threads
typedef struct param
{
   int id;
} param;


// -- Variables globales --

// Matrices
basetype *A;
basetype *B;
basetype *C;    // Matriz resultado
basetype *D;
basetype *E;
```

```
basetype *F;
basetype *L;
basetype *U;
basetype *LT;
basetype *UT;

basetype promB;
basetype promL=0;
basetype promU=0;
basetype prodLU;

//-- matrices de resultados intermedios
basetype * sumaParcialB;
basetype * sumaL;
basetype * sumaU;
basetype * ULA;
basetype * AC;
basetype * bL;
basetype * BE;
basetype * bD;
basetype * UF;
basetype * ULLACbLBE;
basetype * bDUF;
basetype * bLBE;
basetype * ulAAC;
basetype * resultado;

//--


int cant_filas_restantes;    // Cantidad de filas que restan ordenar
int cant_filas_x_thread;    // Cantidad de filas que procesa cada thread en cada iteración (va
disminuyendo)
int pos_max_global;

int N;
int NT;
int CANT_THREADS;

double speedup;
double eficiencia;
double tiempo_paral;
```

```
double tiempo_sec;

basetype factor_global;

pthread_barrier_t barrera;      // Barrerra

// -- Definición de funciones concurrentes --

void *funcion_threads(void *arg);
void promedioB(param* parametro);
void prod_escalar                       (param * parametro, basetype * m1, basetype a,
basetype * m2);
void suma_matriz                        (param * parametro, basetype * m1, basetype * m2,
basetype * res);
void multiplicacion             (param * parametro, basetype * m1, basetype * m2, basetype
* m3, int dim);
void multiplicacionXTriangularU    (param * parametro, basetype * m1, basetype * m2,
basetype * m3, int dim);
void multiplicacionXTriangularL    (param * parametro, basetype * m1, basetype * m2,
basetype * m3, int dim);

void prodPromLU(param* parametro);
void imprimir_matriz (basetype * matriz,int N);
double dwalltime();


// -- funciones secuenciales


//-- caca
double tiempo_copia_total;

#ifdef COMPARAR_SECUENCIAL
void multiplicacion_secuencial(basetype *A,basetype *B,basetype *C_secuencial,int N);
void multiplicacionXTriangularLSECUENCIAL    (basetype * m1, basetype * m2, basetype
*m3, int dim);
void multiplicacionXTriangularUSECUENCIAL    (basetype * m1, basetype * m2, basetype
*m3, int dim);
void multiplicacionSECUENCIAL                       (basetype * m1, basetype * m2,
basetype *m3, int dim);
void promedioBSECUENCIAL                            ();
```

```
void prod_escalarSECUENCIAL                    (basetype * m1, basetype a, basetype *
m2);
void suma_matrizSECUENCIAL                     (basetype * m1, basetype * m2,
basetype * res);
void verificar_resultado(basetype *C,basetype *C_secuencial,int N);

basetype *C_secuencial;    // Matriz resultado
#endif


int main(int argc,char *argv[])
{
   int i;
   int j;

   if (argc != 3) {
         printf("Error en la cantidad de parámetros. Parametro 1 -> long matriz Parametro 2
-> n° threads");
         printf("Ingresar la dimension de la matriz!\n");
         return 0;
   }


   N = atoi(argv[1]);    // Dimensión de la matriz: N*N
   NT = (N*(N+1))/2;
   CANT_THREADS = atoi(argv[2]);


   printf("Dimensión de la matriz: %d*%d \n",N,N);

   //Vector usado para guardar las sumas parciales de la matriz B para luego hacer el
promedio
   sumaParcialB = (basetype*)malloc(sizeof(basetype)*CANT_THREADS);
   basetype promB;


   // -- Arreglos usados para calcular promedios
   sumaL = (basetype*)malloc(sizeof(basetype)*CANT_THREADS);
   sumaU = (basetype*)malloc(sizeof(basetype)*CANT_THREADS);


   // Reserva de memoria para las matrices
```

```
   A=(basetype*)malloc(sizeof(basetype)*N*N);              // Reserva memoria para A
   B=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para B
   C=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para C
   D=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para D
   E=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para E
   F=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para F

   // -- caso especial de matrices triangulares
   L=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para L
   U=(basetype*)malloc(N*N*sizeof(basetype));              // Reserva memoria para U


   LT=(basetype*)malloc(NT*NT*sizeof(basetype));                  // Reserva memoria para
L transformada para ahorrar espacio
   UT=(basetype*)malloc(NT*NT*sizeof(basetype));                  // Reserva memoria para
U transformada para ahorrar espacio


   // -- inicialización de matrices de resultados intermedios
   AC=                    (basetype*)malloc(sizeof(basetype)*N*N);
   ULA=             (basetype*)malloc(sizeof(basetype)*N*N);
   bL=                    (basetype*)malloc(sizeof(basetype)*N*N);
   BE=                    (basetype*)malloc(sizeof(basetype)*N*N);
   bD=                    (basetype*)malloc(sizeof(basetype)*N*N);
   UF=                    (basetype*)malloc(sizeof(basetype)*N*N);
   bDUF=            (basetype*)malloc(sizeof(basetype)*N*N);
   bLBE=            (basetype*)malloc(sizeof(basetype)*N*N);
   ulAAC=           (basetype*)malloc(sizeof(basetype)*N*N);
   resultado=    (basetype*)malloc(sizeof(basetype)*N*N);
   ULLACbLBE=       (basetype*)malloc(sizeof(basetype)*N*N);


   printf("Matrices creadas \n");

   #ifdef COMPARAR_SECUENCIAL
   C_secuencial=(basetype*)malloc(N*N*sizeof(basetype));                  // Reserva
memoria para C_SECUENCIAL
   #endif


   // -- Inicialización ALEATORIA de las matrices --
```

```
// Inicializa matrices A y B
for(i=0;i<N;i++){
        for(j=0;j<N;j++){
                A[i*N+j]=rand()%5;     // Inicializa matriz A con random
                B[i*N+j]=rand()%5;     // Inicializa matriz B con random
                C[i*N+j]=rand()%5;     // Inicializa matriz B con random
                D[i*N+j]=rand()%5;     // Inicializa matriz B con random
                E[i*N+j]=rand()%5;     // Inicializa matriz B con random
                F[i*N+j]=rand()%5;     // Inicializa matriz B con random
        }
}

// -- Inicializacion de matrices triangulares superior por columnas e inferior por filas

for (int i = 0; i < N; ++i)
{
        for (int j = 0; j < N; ++j)
        {
                if(i==j)
                {
                        L[i*N+j]=rand()%5;
                        U[i*N+j]=rand()%5;
                }
                else if(i>j)
                {
                        U[i*N+j]=rand()%5;
                        L[i*N+j]=0;
                }
                else
                {
                        L[i*N+j]=rand()%5;
                        U[i*N+j]=0;
                }
        }
}

// -- Transformo las matrices triangulares en arreglos para ahorrar espacio y libero el
espacio ocupado por las triangulares
int indice;
for (int i = 0; i < N; ++i)
{
        for (int j = 0; j < N; ++j)
```

```
        {
                indice = N * i + j - ((i *(i+1))/2);
                UT[indice]    = U[i*N + j];
                indice = N * j + i - ((j *(j+1))/2);
                LT[indice] = L[i*N+j];
                //UT[i*NT + j - i*(i+1)/2]    =    U[i*N+j];
                //LT[j*N + i - j*(j+1)/2]    =    L[j*N+i];
        }
   }
   printf("Matrices inicializadas \n");

   free(U);
   free(L);


   param parametros[CANT_THREADS];    // Arreglo de param (struct que contiene los datos
para pasar a los threads)

   // -- Inicialización de threads --
   pthread_t threads[CANT_THREADS];    // Arreglo de threads

   if (pthread_barrier_init(&barrera,NULL,CANT_THREADS)!=0) {
        printf("Error creacion de barrera\n");
        return 0;
   }

   // Inicialización de parámetros
   for (i=0;i<CANT_THREADS;i++){
        parametros[i].id=i;
   }
   double tiempo_inicial=dwalltime();
   // Creacion de los threads
   for (i=0;i<CANT_THREADS;i++){
        if (pthread_create( &threads[i],NULL,funcion_threads,(void*)&parametros[i])!=0) {
                printf("Error creacion de thread\n");
                return 0;
        }
   }

   // Join de los threads
   for(i = 0; i < CANT_THREADS; i++)
   {
```

```
        pthread_join(threads[i], NULL);

    }

    tiempo_paral = dwalltime()-tiempo_inicial;
    printf("\nTiempo Total (pthreads) : %f\n\n",dwalltime()-tiempo_inicial);

    #ifdef COMPARAR_SECUENCIAL
        tiempo_inicial=dwalltime();
        multiplicacion_secuencial(A,B,C_secuencial,N);    // C_secuencial = A*B
        tiempo_sec = dwalltime()-tiempo_inicial;
        printf("-- Fin de multiplicacion (secuencial) -->> \t Tiempo: %f \n", tiempo_sec);

        speedup = tiempo_sec / tiempo_paral;
        printf("-- Speedup conseguido: %f \n", speedup);
        eficiencia = speedup / CANT_THREADS;
        printf("-- Eficiencia: %f \n", eficiencia);

        free(C_secuencial);
    #endif


    // Libera memoria
    free(A);
    free(B);
    free(C);
    free(D);
    free(E);
    free(F);
    free(AC);
    free(ULA);
    free(ulAAC);
    free(BE);
    free(LT);
    free(UT);
    free(bLBE);
    free(ULLACbLBE);
    free(bDUF);
    free(UF);
    free(bD);
    return(0);
}
```

```
// ------------------------
// -- FUNCIONES PTHREADS //
// ------------------------


   void *funcion_threads(void *arg) {
          param* parametro = (param*)arg;
          double tiempo_inicial2;
//printf("Mi ID es: %d \n",(*parametro).id);
          if ((*parametro).id==0){
                  tiempo_inicial2=dwalltime();
          }

          multiplicacion(parametro,A,C,AC,N);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   prodPromLU(parametro);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   prod_escalar(parametro,A,prodLU,ULA);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   multiplicacion(parametro,ULA,AC,ulAAC,N);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   promedioB(parametro);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   multiplicacion(parametro,B,E,BE,N);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   multiplicacionXTriangularL(parametro,BE,LT,bLBE,N);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   prod_escalar(parametro,bLBE,promB,bLBE);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   prod_escalar(parametro,D,promB,bD);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen
```

```
   multiplicacionXTriangularU(parametro,F,UT,UF,N);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   multiplicacion(parametro,bD,UF,bDUF,N);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   suma_matriz(parametro,ulAAC,bLBE,ULLACbLBE);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen

   suma_matriz(parametro,ULLACbLBE,bDUF,resultado);
   pthread_barrier_wait(&barrera); //espero a que todos los hilos finalicen


   if ((*parametro).id==0){
         printf("-- Fin de operacion (pthread) -->> \t Tiempo: %f
\n",dwalltime()-tiempo_inicial2);
         //printf("Matriz A:\n");
   }
   pthread_exit(NULL);
}

/*
*
*    m2 es triangular superior
*/
void multiplicacionXTriangularU(param* parametro, basetype * m1, basetype * m2,
basetype *m3, int dim)
{
   int id = (*parametro).id;
   basetype total;
   basetype aux;
   int i,j,k;


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


   for(int i=fila_inicial;i<=fila_final;i++)
   {    // Recorre solo algunas filas
```

```
        for(int j=0;j<dim;j++)
        {   // Recorre todas las columnas

                total=0;
                for(int k=0;k<dim;k++)
                {
                        if(i>=j)
                        {
                                aux=m2[i*NT + j - i*(i+1)/2];
                        }
                        else aux = 0;
                        total+=m1[i*dim+k]*aux;
                }
                m3[i*dim+j] = total;
        }
   }
}

void multiplicacionXTriangularUSECUENCIAL(basetype * m1, basetype * m2, basetype *m3,
int dim)
{
   basetype aux;
   basetype total;
   for(int i=0;i<dim;i++)
   {   // Recorre solo algunas filas
        for(int j=0;j<dim;j++)
        {   // Recorre todas las columnas

                total=0;
                for(int k=0;k<dim;k++)
                {
                        if(i>=j)
                        {
                                aux=m2[i*NT + j - i*(i+1)/2];
                        }
                        else aux = 0;
                        total+=m1[i*dim+k]*aux;
                }
                m3[i*dim+j] = total;
        }
   }
}
```

```
void multiplicacionXTriangularLSECUENCIAL( basetype * m1, basetype * m2, basetype *m3,
int dim)
{
   basetype total;
   basetype aux;
   for(int i=0;i<dim;i++)
   {   // Recorre solo algunas filas
         for(int j=0;j<dim;j++)
         {   // Recorre todas las columnas

                  total=0;
                  for(int k=0;k<dim;k++)
                  {
                           if(i>=j)
                           {
                                    aux=m2[i*NT + j - i*(i+1)/2];
                           }
                           else aux = 0;
                           total+=m1[i*dim+k]*aux;
                  }
                  m3[i*dim+j] = total;
         }
   }
}

/*
*
*    m2 es triangular superior
*/
void multiplicacionXTriangularL(param* parametro, basetype * m1, basetype * m2,
basetype *m3, int dim)
{
   int id = (*parametro).id;
   basetype total;
   basetype aux;
   int i,j,k;


   // Filas que multiplica el thread
   int cant_filas = dim/CANT_THREADS;    // Cant de filas que multiplica cada thread
```

```
   int fila_inicial = id*cant_filas;
   int fila_final = fila_inicial + cant_filas -1;


   for(int i=fila_inicial;i<=fila_final;i++)
   {   // Recorre solo algunas filas
         for(int j=0;j<dim;j++)
         {   // Recorre todas las columnas

                  total=0;
                  for(int k=0;k<dim;k++)
                  {
                           if(i<=j)
                           {
                                    aux=m2[i+j*NT - i*(i+1)/2];
                           }
                           else aux = 0;
                           total+=m1[i*dim+k]*aux;
                  }
                  m3[i*dim+j] = total;
         }
   }
}

void multiplicacion(param* parametro, basetype * m1, basetype * m2, basetype * m3, int
dim)
{
   int id = (*parametro).id;
   basetype total;
   int i,j,k;


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

   //printf("ID: %d \t fila_inicial=%d \t fila_final=%d \n",id,fila_inicial,fila_final);

   // Multiplica A*B=C
   for(i=fila_inicial;i<=fila_final;i++)
   {   // Recorre solo algunas filas
```

```
        for(j=0;j<dim;j++)
        {   // Recorre todas las columnas
                total=0;
                for(k=0;k<dim;k++)
                {
                        total+=m1[i*dim+k]*m2[k*dim+j];    // total=A*B
                }
                m3[i*dim+j] = total;
        }
   }
}



void prodPromLU(param* parametro)
{
   int id = (*parametro).id;
   basetype total;
   int i,j,k;


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

   sumaL[id]=0;
   sumaU[id]=0;


   for(i=fila_inicial;i<=fila_final;i++)
   {   // Recorre solo algunas filas
         for(j=0;j<NT;j++)
         {
                sumaL[id]+=UT[i*NT+j];
                sumaU[id]+=LT[i*NT+j];
         }
   }
   pthread_barrier_wait(&barrera); //Espera a que todas terminen
        if(id == 0)                                          //si es el hilo principal
        {
         for (int i = 0; i < CANT_THREADS; ++i)
```

```
        {
                promL+=sumaL[i];
                promU+=sumaU[i];
        }
        promL/=CANT_THREADS;
        promU/=CANT_THREADS;
        prodLU=promU*promL;
       }
}
basetype prodLUSEC;
basetype sumaLSEC,sumaUSEC;
void prodPromLUSECUENCIAL()
{
   sumaLSEC=0;
   sumaUSEC=0;
   for(int i=0;i<=NT;i++)
   {    // Recorre solo algunas filas
        for(int j=0;j<NT;j++)
        {
                sumaLSEC+=UT[i*NT+j];
                sumaUSEC+=LT[i*NT+j];
        }
   }

   prodLUSEC=promU*promL;

}

void promedioB(param* parametro)
{
   int id = (*parametro).id;
   basetype total;
   int i,j,k;


   // 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;

   //printf("ID: %d \t fila_inicial=%d \t fila_final=%d \n",id,fila_inicial,fila_final);
   total=0;
```

```
   // Multiplica A*B=C
   for(i=fila_inicial;i<=fila_final;i++)
   {                                                                    // Recorre solo algunas
filas
          for(j=0;j<N;j++)                                  // Recorre todas las columnas
          {
                    total+=B[i*N + j];
          }
   }
   sumaParcialB[id]=total;
          pthread_barrier_wait(&barrera); //Espera a que todas terminen
          if(id == 0)                                       //si es el hilo principal
          {
           for (int i = 0; i < CANT_THREADS; ++i)
           {
                    promB+=sumaParcialB[i];
           }
           promB /=(N*N);
          }
}

void promedioBSECUENCIAL()
{

   basetype total=0;
   // Multiplica A*B=C
   for(int i=0;i<=N;i++)
   {                                                                    // Recorre solo algunas
filas
          for(int j=0;j<N;j++)                              // Recorre todas las columnas
          {
                    total+=B[i*N + j];
          }
   }

   promB =total/(N*N);

}

// -------------------------
// -- FUNCIONES AUXILIARES //
// -------------------------
```

```
void imprimir_matriz (basetype * matriz,int N){

   for (int i=0;i<N;i++){
         for (int j = 0 ; j < N ; j++){
               printf ("%.1f\t",matriz [ i * N + j ]);
         }
         printf("\n");
   }
}

void prod_escalar (param * parametro, basetype * m1, basetype a, basetype * m2)
{
   int id = (*parametro).id;
   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;

   for (int i = fila_inicial; i < fila_final; ++i)
   {
         for (int j = 0; j < N; ++j)
         {
               m2[i*N+j]=m1[i*N+j]*a;
         }
   }
}

void prod_escalarSECUENCIAL ( basetype * m1, basetype a, basetype * m2)
{


   for (int i = 0; i < N; ++i)
   {
         for (int j = 0; j < N; ++j)
         {
               m2[i*N+j]=m1[i*N+j]*a;
         }
   }
}

void suma_matriz (param * parametro, basetype * m1, basetype * m2, basetype * res)
{
```

```
   int id = (*parametro).id;
   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;

   for (int i = fila_inicial; i < fila_final; ++i)
   {
         for (int j = 0; j < N; ++j)
         {
               res[i*N+j]   =   m1[i*N+j]   +   m2[i*N+j];
         }
   }
}
void suma_matrizSECUENCIAL ( basetype * m1, basetype * m2, basetype * res)
{
   for (int i = 0; i < N; ++i)
   {
         for (int j = 0; j < N; ++j)
         {
               res[i*N+j]   =   m1[i*N+j]   +   m2[i*N+j];
         }
   }
}


double dwalltime(){
   double sec;
   struct timeval tv;

   gettimeofday(&tv,NULL);
   sec = tv.tv_sec + tv.tv_usec/1000000.0;
   return sec;
}

// ----------------------------
// -- FUNCIONES SECUENCIALES //
// ----------------------------

#ifdef COMPARAR_SECUENCIAL
void multiplicacion_secuencial(basetype *A,basetype *B,basetype *C,int N){
   //printf("Comienzo etapa 1\n");
```

```
    basetype total;
    int i,j,k;

    // 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
            }
            C[i*N+j] = total;          // C=total
        }
    }

    void verificar_resultado(basetype *C,basetype *C_secuencial,int N){
        int i,j;
        int check = 1;
        for(i=0;i<N;i++){
            for(j=0;j<N;j++){
                check=check&&(C[i*N+j]==C_secuencial[i*N+j]);
            }
        }

        if(check){
            printf("Resultado correcto\n");
        }
        else{
            printf("Resultado erroneo \n");
        }
    }
#endif
```