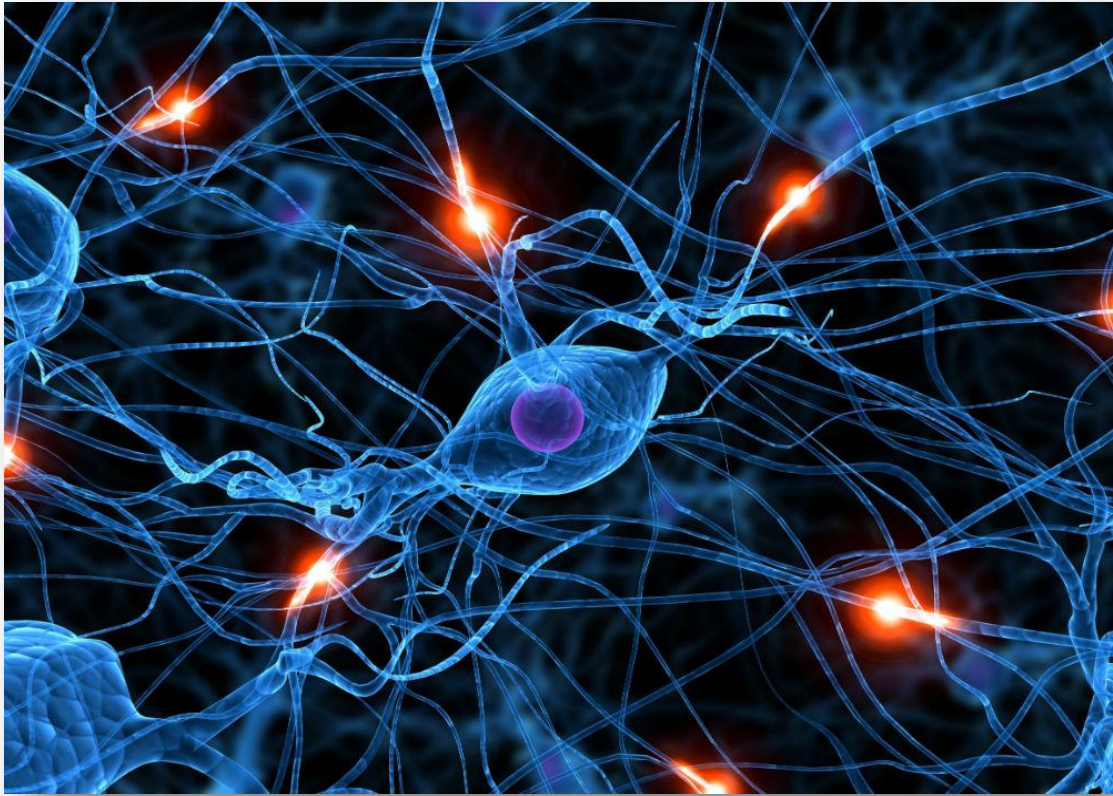




Multiprocesadores Curso 2017-18



Práctica 04.- Uso de PThreads para paralelización. Introducción al entorno OpenMP

Daniel García Mármol

Objetivos :

- Poner a prueba lo aprendido en prácticas anteriores.
- Crear 3 versiones: secuencial, paralela y paralela con mutex.
- Evaluar el rendimiento de cada versión.
- Introducción al entorno OpenMP.

Parte 1: Uso de Pthreads para paralelización

En este apartado vamos a realizar un programa que calcule en paralelo la Distancia de Jaccard dados dos conjuntos de enteros representados en vectores, y que pueden tener distinta longitud. Generar los datos a partir del esquema que se le adjunta. El programa recibirá por consola la longitud de los dos vectores y el número de hilos a desplegar cuando se ejecute. Finalmente, considerando que los usuarios finales no están acostumbrados a trabajar con programas en consola, es recomendable que se filtren mínimamente los parámetros recibidos para, por ejemplo, evitar longitudes negativas o un número mayor de hilos que de elementos por vector.

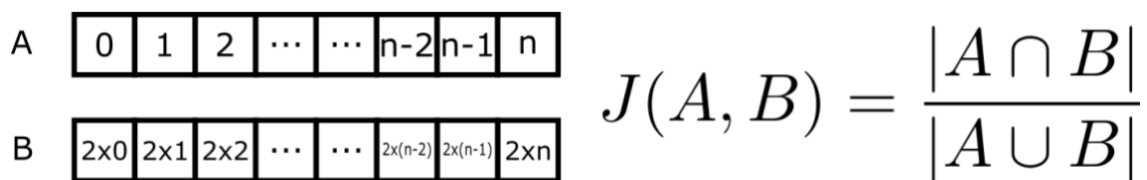


Figura 1. Esquema del cual nos basamos para rellenar nuestros vectores de enteros

A continuación, se expondrán las 3 versiones que hemos realizado para esta práctica con su correspondiente explicación paso a paso. Decir antes, que la explicación paralela y la de paralela usando mutex es muy similar, pues cambian solo cosas puntuales.

Versión secuencial

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

// Función de intersección, recorre el array A y para cada
// posición, recorre el array B, si ambas son iguales incrementa
// sumIntersection. Para acabar, devuelve sumIntersection, que es
// la cantidad de numeros repetidos en ambos arrays

int f_intersection(int* vA, int lA, int* vB, int lB){
    int sumIntersection=0;
    int i, j;

    for(i=0; i<lA; i++){
        for(j=0; j<lB; j++){
            if(vA[i]==vB[j]){
                sumIntersection++;
            }
        }
    }

    return sumIntersection;
}
```

```

// Función para simplificar el proceso de obtener el instante de tiempo
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        // Handle error
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}

// Función un union, es muy simple, sumamos las longitudes de los
// vectores y restamos lo que nos pasa por parámetro, que es el resultado
// de f_intersection ejecutada anteriormente,
// pues si sumamos todos los numeros y eliminamos los repetidos, tenemos 1
// de cada cual. [0,1,2,3,4] [0,2,4,6,8] --> 5 + 5 - 3 = 7
int f_union(int lA, int lB, int intersection){
    int sumUnion=0;
    int i, j;

    sumUnion = lA + lB;

    return (sumUnion - intersection);
}

int main(int argc, char *argv[]){
    system("clear");
    int i, lengthA, lengthB;
    int *vectorA;
    int *vectorB;

    double start, end;

    /* Limpiamos terminal */
    /* Inicializamos variables para longitud */
    /* Inicializamos puntero vectorA */
    /* Inicializamos puntero vectorB */

    /* Inicializamos variables para */
    /* el calculo del tiempo */

    printf("Insert the length of vector A: ");
    scanf("%d", &lengthA);
    printf("\nInsert the length of vector B: ");
    scanf("%d", &lengthB);

    // El problema nos dice que debemos trabajar para testear el programa
    // con valores de entrada de 0..n, pero claro, si insertamos 20, nuestro
    // array no va a llegar hasta 0..n-1, debemos de sumar +1 a los valores
    // introducidos por pantalla.

    lengthA+=1;
    lengthB+=1;

    start = get_wall_time();
    /* get initial time */
    /* in seconds */

    vectorA = malloc(sizeof(int)*lengthA); // Reservamos memoria dinámicamente
    for(i=0; i<lengthA; i++){
        vectorA[i] = i; // y rellenamos el puntero con valores
        // de 0..lengthA
    }

    vectorB = malloc(sizeof(int)*lengthB); // Reservamos memoria dinámicamente
    for(i=0; i<lengthB; i++){
        vectorB[i] = 2*i; // y rellenamos el puntero con valores
        // de 0x2..lengthBx2
    }

    // Guardo en intersection y en unions, el resultado de f_intersection y
    // f_union respectivamente para luego hacerles un casting a double y dividirlos
    // para imprimir su resultado con formato

    int intersection = f_intersection(vectorA, lengthA, vectorB, lengthB);
    int unions = f_union(lengthA,lengthB, intersection);
    printf("\nJaccard's distance is %.2f\n", (double)intersection/(double)unions);

    // Liberamos memoria que previamente hemos reservado dinámicamente

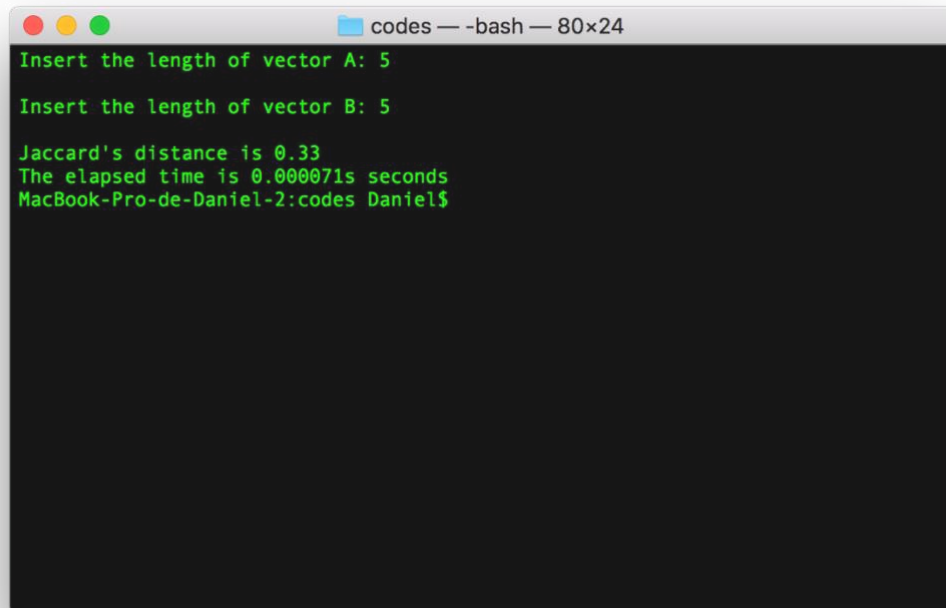
    free(vectorB);
    free(vectorA);

    end = get_wall_time();
    /* get end time */
    /* in seconds */

    printf("The elapsed time is %.6fs seconds\n", // Imprimimos por pantalla el tiempo
    end-start); // en segundos de lo que ha tardado
    // en ejecutar secuencialmente desde
    // que se insertan los datos por pantalla
    // hasta que se muestra el valor de Jaccard
}

```

En la práctica se menciona que se haga una prueba con vectores de tamaño 5. A continuación, se realiza una prueba de ejecución del código previo:

A terminal window titled 'codes — -bash — 80x24' with a dark background and green text. The output shows the program's execution for two vectors of size 5, resulting in a Jaccard distance of 0.33 and an elapsed time of 0.000071s.

```
Insert the length of vector A: 5
Insert the length of vector B: 5
Jaccard's distance is 0.33
The elapsed time is 0.000071s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

Versión paralela basada en PThreads sin Mutex

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

// Estructura para pasar al hilo

struct parametros {
    int inicio; // Definimos una estructura con los parámetros
    int fin; // necesarios que necesitaremos pasarle a nuestro
    int distancia; // hilo. Esto incluye 3 variables tipo int, que
    int *vectorPrim; // indicaran, principio, fin, y un puntero de tipo
    int *vectorSec; // double, que será el array mas pequeño. Y un
    int resultadoParcial; // puntero de tipo double, que será el array más
    // grande. Para finalizar, se incluye variable double
    // que usaremos para guardar el resultado en cada hilo
};

// Función para simplificar el proceso de obtener el instante de tiempo
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time, NULL)){
        // Handle error
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}

// Función de intersección, recorre el array A, que previamente
// se ha dividido para cada hilo y se asignan un inicio y final
// del array, recorre el array B, si ambas son iguales incrementa
// sumIntersection. Para acabar, devuelve sumIntersection, que es
// la cantidad de numeros repetidos en ambos arrays.
```



```

int f_intersection(int* vA, int inicio, int fin, int* vB, int lB){
    int sumIntersection=0;
    int i, j;

    for(i=inicio; i<fin; i++){
        for(j=0; j<lB; j++){
            if(vA[i]==vB[j]){
                sumIntersection++;
            }
        }
    }

    return sumIntersection;
}

// Función un union, es muy simple, sumamos las longitudes de los
// vectores y restamos el valor que se nos pasa por parámetro intersection,
// pues si sumamos todos los numeros y eliminamos los repetidos, tenemos 1
// de cada cual. [0,1,2,3,4] [0,2,4,6,8] --> 5 + 5 - 3 = 7
//                | | | | |         | | --> 7

int f_union(int lA, int lB, int intersection){
    int sumUnion=0;
    int i, j;

    sumUnion = lA + lB;

    return (sumUnion - intersection);
}

void *do_work( void *arg );

int main(int argc, char *argv[]){
    system("clear");          /* Limpiamos terminal */
    int i, lengthA, lengthB,  /* Inicializamos variables varias
nHilos,                      /* de tipo int
resultadoIntersection,
paquete, excedente;
    int *vectorA;             /* Inicializamos puntero vectorA */
    int *vectorB;             /* Inicializamos puntero vectorB */
    double start, end;        /* Inicializamos variables para */
                                /* el calculo del tiempo */

    printf("Insert the length of vector A: "); // Pedimos por pantalla los
    scanf("%d", &lengthA); // valores de longitud de
    printf("\nInsert the length of vector B: "); // los vectores A y B;
    scanf("%d", &lengthB); // y el número de hilos
    printf("\nInsert the number of threads: "); // con los que se trabajarán
    scanf("%d", &nHilos);

    // El problema nos dice que debemos trabajar para testear el programa
    // con valores de entrada de 0..n, pero claro, si insertamos 20, nuestro
    // array no va a llegar hasta 0..n-1, debemos de sumar +1 a los valores
    // introducidos por pantalla.

    lengthA+=1;
    lengthB+=1;

    start = get_wall_time(); // get initial time */
                                /* in seconds */

    vectorA = malloc(sizeof(int)*lengthA); // Reservamos memoria dinámicamente
    for(i=0; i<lengthA; i++){ // y rellenamos el puntero con valores
        vectorA[i] = i; // de 0..lengthA
    }

    vectorB = malloc(sizeof(int)*lengthB); // Reservamos memoria dinámicamente
    for(i=0; i<lengthB; i++){ // y rellenamos el puntero con valores
        vectorB[i] = 2*i; // de 0x2..lengthBx2
    }

    // Creamos un hilo y reservamos con tantos hilos como nos haya pasado por
    // parámetro

    pthread_t *thread;
    thread = (pthread_t*)malloc(sizeof(pthread_t)*nHilos);

    // Paquete guarda la división entre el número de celdas que tiene nuestro array
    // entre el número de hilos, para saber cuantas celdas le corresponde a cada uno

```

```

paquete = lengthA / nHilos;

// Excedente recoge el resto para balancear la carga
excedente = lengthA % nHilos;

// Inicializamos las variables fin e inicio, que usaremos para informar donde empieza
// y acaba el trabajo de cada hilo en el vector
int fin = 0;
int inicio = 0;

// Creamos un array de parámetros, de tamaño cuantos hilos tengamos pues estos van
// a ser los datos con los que trabaje cada uno, que se les pasará cuando se inicialicen
struct parametros parametros_array[nHilos];

// Bucle for que rellena los parámetros con inicio, fin del hilo y crea el hilo,
// esto se repite en cada iteración con nuevos valores
for(int i = 0; i<nHilos; i++){
    parametros_array[i].vectorPrim = vectorA;
    parametros_array[i].vectorSec = vectorB;
    parametros_array[i].distancia = lengthB;
    fin = inicio + paquete;
    if(excedente > 0){
        fin++;
        excedente--;
    }
    parametros_array[i].inicio = inicio;
    parametros_array[i].fin = fin;
    pthread_create ( &thread[i], NULL, do_work, (void*)&parametros_array[i]);
    inicio = fin;
}

// Esperamos a que todos los hilos acaben y en la variable resultadoIntersection
// vamos almacenando iterativamente el valor obtenido en cada hilo
for(int i = 0; i<nHilos; i++){
    pthread_join( thread[i], NULL);
    resultadoIntersection += parametros_array[i].resultadoParcial;
}

printf("\nJaccard's distance is %.2f\n", (resultadoIntersection/f_union(lengthA, lengthB,
resultadoIntersection)));

// liberamos memoria reservada dinámicamente
free(thread);
free(vectorA);
free(vectorB);

end = get_wall_time();          /* get end time */
                                /* in seconds */

printf("The elapsed time is %.6fs seconds\n",          // Imprimimos por pantalla el tiempo
end-start);      // en segundos de lo que ha tardado
                                // en ejecutar secuencialmente desde
                                // que se insertan los datos por pantalla
                                // hasta que se muestra el valor de Jaccard
}

void *do_work(void *arg){
    struct parametros * p;          // Mediante un casting creamos una estructura de tipo
    p = ( struct parametros *) arg ; // parametros recibe los valores que le hemos pasado al
                                // crear el hilo(inicio, fin, vectorPrim, vectorSec,
                                // resultadoParcial)

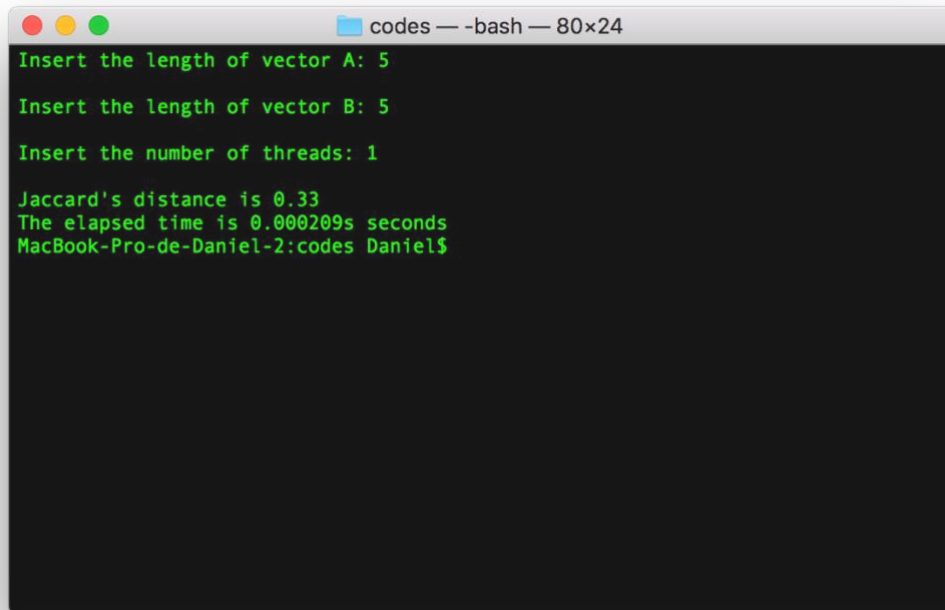
    // Guardamos en la variable resultadoParcial, el valor de f_intersection con los parámetros
    // para cada hilo, no lo hacemos iterativo (+=) pues cada resultadoParcial pertenece a un
    // hilo

    p->resultadoParcial = f_intersection(p->vectorPrim,p->inicio, p->fin, p->vectorSec,
    p->distancia);

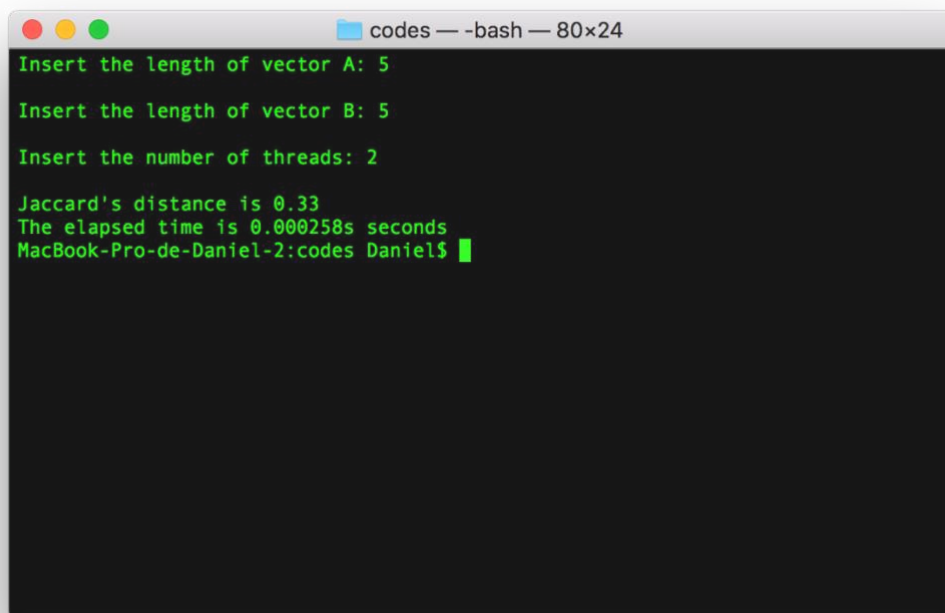
    pthread_exit((void*) p);
}

```

A continuación, se realiza una prueba de ejecución del código previo de 1 hasta 4 procesadores:



```
codes — -bash — 80x24
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 1
Jaccard's distance is 0.33
The elapsed time is 0.000209s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```



```
codes — -bash — 80x24
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 2
Jaccard's distance is 0.33
The elapsed time is 0.000258s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

```
codes — -bash — 80x24
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 3
Jaccard's distance is 0.33
The elapsed time is 0.000274s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

```
codes — -bash — 80x24
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 4
Jaccard's distance is 0.33
The elapsed time is 0.000285s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

[Versión paralela basada en PThreads con Mutex](#)


```

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

pthread_mutex_t llave;

int SumaIntersection = 0; // Variable global donde almacenaremos la suma de
                          // la intersección, usaremos mutex para evitar errores

// Estructura para pasar al hilo

struct parametros {
    int inicio; // Definimos una estructura con los parámetros
    int fin;    // necesarios que necesitaremos pasarle a nuestro
    int distancia; // hilo. Esto incluye 3 variables tipo int, que
    int *vectorPrim; // indicaran, principio, fin, y un puntero de tipo
    int *vectorSec; // double, que será el array mas pequeño. Y un
                  // puntero de tipo double, que será el array más
                  // grande.
};

// funcion para simplificar el proceso de obtener el instante de tiempo
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        // Handle error
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}

// Función de intersección, recorre el array A, que previamente
// se ha dividido para cada hilo y se asignan un inicio y final
// del array, recorre el array B, si ambas son iguales incrementa
// sumIntersection. Para acabar, devuelve sumIntersection, que es
// la cantidad de numeros repetidos en ambos arrays.
int f_intersection(int* vA, int inicio, int fin, int* vB, int lB){
    int sumIntersection=0;
    int i, j;

    for(i=inicio; i<fin; i++){
        for(j=0; j<lB; j++){
            if(vA[i]==vB[j]){
                sumIntersection++;
            }
        }
    }

    return sumIntersection;
}

// Función un union, es muy simple, sumamos las longitudes de los
// vectores y restamos el valor de la variable global,
// pues si sumamos todos los numeros y eliminamos los repetidos, tenemos 1
// de cada cual. [0,1,2,3,4] [0,2,4,6,8] --> 5 + 5 - 3 = 7
//               | | | | |   | | | | |
double f_union(int lA, int lB){
    int sumUnion=0;
    int i, j;

    sumUnion = lA + lB;

    return (sumUnion - SumaIntersection);
}

void *do_work( void *arg );

int main(int argc, char *argv[]){
    system("clear"); // Limpiamos terminal */
    int i, lengthA, lengthB, // Inicializamos variables
    nHilos, resultadoIntersection, // varias de tipo int
    paquete, excedente;
    int *vectorA; // Inicializamos puntero vectorA */
    int *vectorB; // Inicializamos puntero vectorB */
    double start, end; // Inicializamos variables para */
                      // el calculo del tiempo */

```

```

printf("Insert the length of vector A: "); // Pedimos por pantalla los
scanf("%d", &lengthA); // valores de longitud de
printf("\nInsert the length of vector B: "); // los vectores A y B;
scanf("%d", &lengthB); // y el número de hilos
printf("\nInsert the number of threads: "); // con los que se trabajarán
scanf("%d", &nHilos);

// El problema nos dice que debemos trabajar para testear el programa
// con valores de entrada de 0..n, pero claro, si insertamos 20, nuestro
// array no va a llegar hasta 0..n-1, debemos de sumar +1 a los valores
// introducidos por pantalla.

lengthA+=1;
lengthB+=1;

start = get_wall_time(); // get initial time */
// in seconds */

vectorA = malloc(sizeof(int)*lengthA); // Reservamos memoria dinámicamente
for(i=0; i<lengthA; i++){ // y rellenamos el puntero con valores
    vectorA[i] = i; // de 0..lengthA
}

vectorB = malloc(sizeof(int)*lengthB); // Reservamos memoria dinámicamente
for(i=0; i<lengthB; i++){ // y rellenamos el puntero con valores
    vectorB[i] = 2*i; // de 0x2..lengthBx2
}

// Creamos un hilo y reservamos con tantos hilos como nos haya pasado por
// parámetro

pthread_t *thread;
thread = (pthread_t*)malloc(sizeof(pthread_t)*nHilos);

// Paquete guarda la división entre el número de celdas que tiene nuestro array
// entre el número de hilos, para saber cuantas celdas le corresponde a cada uno

paquete = lengthA / nHilos;

// Excedente recoge el resto para balancear la carga

excedente = lengthA % nHilos;

// Inicializamos las variables fin e inicio, que usaremos para informar donde empieza
// y acaba el trabajo de cada hilo en el vector

int fin = 0;
int inicio = 0;

// Creamos un array de parámetros, de tamaño cuantos hilos tengamos pues estos van
// a ser los datos con los que trabaje cada uno, que se les pasará cuando se inicialicen

struct parametros parametros_array[nHilos];

// Bucle for que rellena los parámetros con inicio, fin del hilo y crea el hilo,
// esto se repite en cada iteración con nuevos valores

for(int i = 0; i<nHilos; i++){
    parametros_array[i].vectorPrim = vectorA;
    parametros_array[i].vectorSec = vectorB;
    parametros_array[i].distancia = lengthB;
    fin = inicio + paquete;
    if(excedente > 0){
        fin++;
        excedente--;
    }
    parametros_array[i].inicio = inicio;
    parametros_array[i].fin = fin;
    pthread_create ( &thread[i], NULL, do_work, (void*)&parametros_array[i]);
    inicio = fin;
}

// Esperamos a que todos los hilos acaben

for(int i = 0; i<nHilos; i++){
    pthread_join( thread[i], NULL);
}

printf("\nJaccard's distance is %.2f\n", ((double)SumaIntersection/f_union(lengthA, lengthB)));

// liberamos memoria reservada dinámicamente y eliminamos la llave

pthread_mutex_destroy(&llave);
free(thread);
free(vectorA);
free(vectorB);

end = get_wall_time();
printf("The elapsed time is %.6f seconds\n", // Imprimimos por pantalla el tiempo
end-start); // en segundos de lo que ha tardado
// en ejecutar secuencialmente desde
// que se insertan los datos por pantalla
// hasta que se muestra el valor de Jaccard

```

```

}

void *do_work(void *arg){

    int sum;

    struct parametros * p;          // Mediante un casting creamos una estructura de tipo
    p = ( struct parametros *) arg ; // parametros que recibe los valores que le hemos pasado
                                    // al crear el hilo (inicio, fin, vectorPrim, vectorSec,
                                    // resultadoParcial)

    sum = f_intersection(p->vectorPrim,p->inicio, p->fin, p->vectorSec, p->distancia);

    pthread_mutex_lock(&llave);      // Cerramos la llave para que otros
                                    // hilos no puedan acceder temporalmente

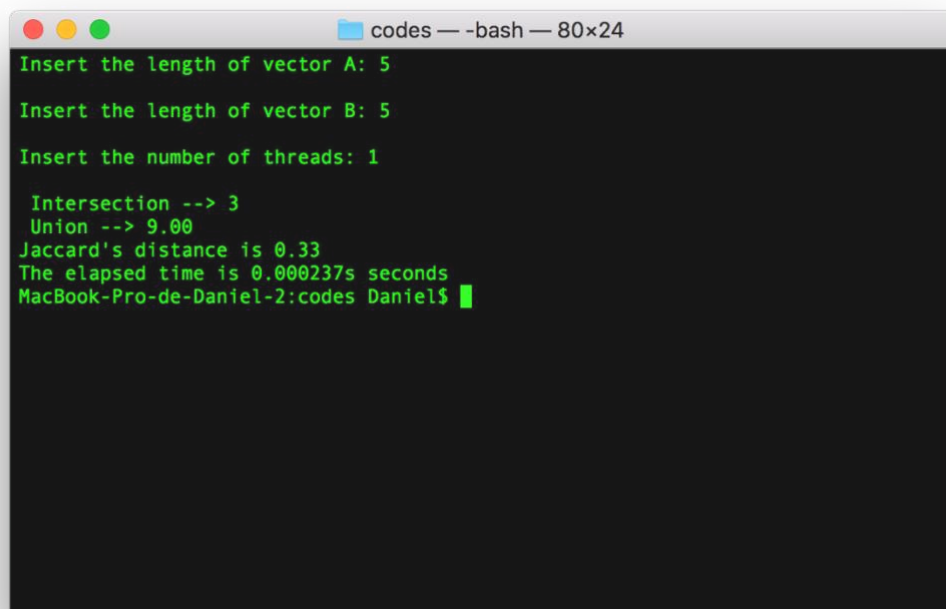
    SumaIntersection += sum;         // Guardamos la suma local en la suma global

    pthread_mutex_unlock(&llave);    // Abrimos la llave para que accedan a el

    pthread_exit((void*) p);
}

```

A continuación, se realiza una prueba de ejecución del código previo de 1 hasta 4 procesadores:



A terminal window titled 'codes — -bash — 80x24' displays the output of a program. The output shows the user inputting values for vector lengths and the number of threads, followed by calculated intersection and union values, Jaccard's distance, and elapsed time.

```

Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 1

Intersection --> 3
Union --> 9.00
Jaccard's distance is 0.33
The elapsed time is 0.000237s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$

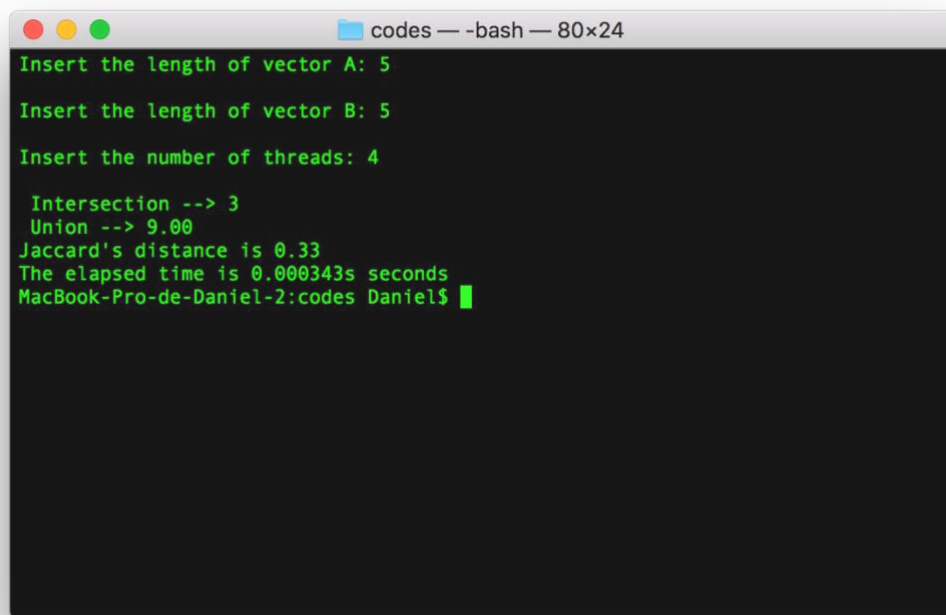
```

```
codes — -bash — 80×24
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 2

Intersection --> 3
Union --> 9.00
Jaccard's distance is 0.33
The elapsed time is 0.000283s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

```
codes — -bash — 80×24
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 3

Intersection --> 3
Union --> 9.00
Jaccard's distance is 0.33
The elapsed time is 0.000375s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

A terminal window titled 'codes — -bash — 80x24' with a dark background and green text. The output shows the user inputting values for vector lengths and threads, followed by calculated intersection and union values, Jaccard's distance, and elapsed time.

```
Insert the length of vector A: 5
Insert the length of vector B: 5
Insert the number of threads: 4

Intersection --> 3
Union --> 9.00
Jaccard's distance is 0.33
The elapsed time is 0.000343s seconds
MacBook-Pro-de-Daniel-2:codes Daniel$
```

Para concluir esta parte, decir que falta implementar en las 3 versiones la parte en la que dice que se debe hacer un filtrado previo de las entradas por consola para la eliminación de posibles fallos a la hora de su utilización. Como ya está el pdf casi acabado y acabo de darme cuenta de este fallo, comento que es lo que habría que hacer. Para filtrar que no se puedan meter valores negativos, después de las entradas por consola se pondría un:

```
if(longitudA <=0 || longitudB <=0 || nHilos > longitudA || nHilos > longitudB){ return 0};
```

que comprobaría si las longitudes de los vectores son menores o iguales que 0 y si se ha insertado más números de hilos que longitud del vector hay, si alguna de estas se cumple, terminará la ejecución del programa. Se podría meter un printf en modo de “excepción” para decir el por qué se ha parado la ejecución.

[Actividad 1](#)

Evaluar el rendimiento de cada versión registrando los tiempos de ejecución, para las tres versiones anteriores, en cada uno de estos casos (y promediados tras 5 ejecuciones). Se realizará una tabla comparando secuencial, paralelo y mutex para un mismo número de elementos que incrementará x10 en cada columna:

	40 elementos	400 elementos	4000 elementos	40000 elementos	400000 elementos
Secuencial	0.000072s	0.000263s	0.0139s	0.9632s	97.12s
Paralelo-1	0.000218s	0.000426s	0.0141s	0.9752s	96.89s
Paralelo-2	0.000259s	0.000372s	0.00790s	0.4913s	48.90s
Paralelo-3	0.000291s	0.000348s	0.00573s	0.3385s	33.51s
Paralelo-4	0.000314s	0.000351s	0.00458s	0.2567s	25.39s
Mutex-1	0.000268s	0.000460s	0.01365s	0.9718s	96.86s
Mutex-2	0.000292s	0.000398s	0.00792s	0.4932s	49.86s
Mutex-3	0.000324s	0.000400s	0.00464s	0.3393s	33.63s
Mutex-4	0.000338s	0.000397s	0.00480s	0.2592s	25.47s

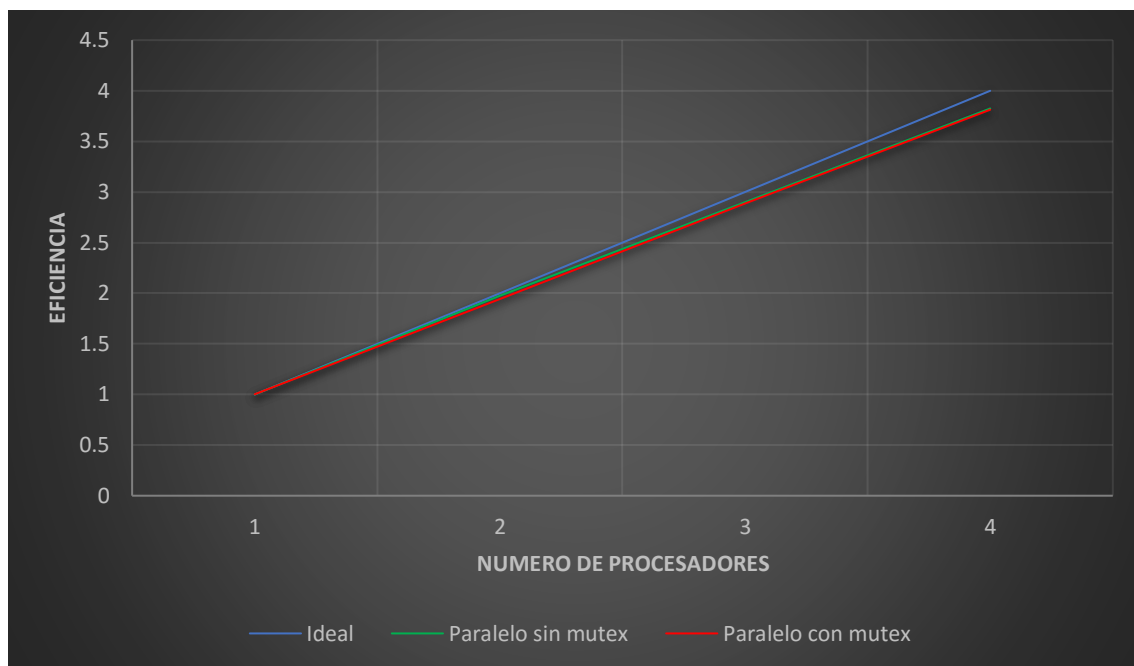
Los datos obtenidos tienen bastante sentido, pues a elementos pequeños, dividir el código y luego sumar cada parte, rellenar estructuras para pasar a los hilos que se creen, hacen que el código sea bastante peor que el secuencial, llegándolo a empeorar hasta más de 4 veces. En cambio, a medida que los elementos son un poco más grandes, ya se empiezan a notar las primeras diferencias de tiempos. Y ya si nos fijamos en la última columna de toma de datos, podemos ver que con 2 procesadores el tiempo se reduce a la mitad, con 3 procesadores se reduce a 3, y con 4 procesadores casi a 4. Ahora, si nos fijamos en los tiempos con mutex y sin mutex, pues la diferencia es prácticamente nula, son diferencias de tiempo muy pequeñas.

Actividad 2

Hacer una gráfica de aceleración ("speedup") conjunta con los datos recogidos anteriormente de las versiones paralelas con respecto a la secuencial. Además de

mostrarse la evolución de las versiones paralelas, debe incluirse en la gráfica la aceleración ideal lineal.

Para realizar la gráfica que se nos pide, debemos escoger un tamaño de la tabla de la actividad anterior. Yo he escogido la de mayor tamaño, 400000 elementos, pues los tiempos son más precisos que a menor tamaño. Una vez tengamos nuestros datos seleccionados, vamos a proceder a calcular la aceleración para 1, 2, 3 y 4 procesadores. Para ello, debemos dividir el tiempo secuencial, entre el tiempo que tarda cada versión con sus distintos números de procesadores. Cuando hayamos hecho esto, solo nos queda representar la gráfica:



Podemos ver, que conforme aumentan los procesadores, la aceleración de los tiempos en paralelo tanto con mutex, como sin él, se van despegando de la aceleración ideal y tienden hacia abajo. La aceleración en paralelo con mutex y sin mutex, están ínfimamente separadas, por lo que podríamos decir con estos datos que son iguales.

Actividad 3

Comentar los datos obtenidos. Resulta importante dar en primera instancia un tono general a los comentarios en relación a los beneficios logrados por la explotación del

paralelismo en este problema. Los comentarios deben centrarse finalmente en las diferencias que se observen en el comportamiento de las dos versiones paralelas en un tono comparativo

Las comparaciones se han ido haciendo en cada actividad, pero aquí un breve resumen. En la primera actividad, podemos ver como a niveles bajo de elementos trabajar sin paralelismo era hasta 4 veces más eficaz, pero que a medida que empezamos a aumentar el número de elementos podemos observar que, a grandes cantidades de elementos, el tiempo se reduce tanto como procesadores se hayan utilizado para ejecutar el proceso. Gracias a esto, si en nuestro caso, trabajamos con elementos muy muy grandes, y tenemos un buen número de procesadores, podemos optimizar el tiempo que se invierte en hacer estas operaciones.

Para finalizar, no noto diferencia alguna con las dos versiones paralelas, pues el speedup sale prácticamente igual y los tiempos son casi iguales. Puede ser que no esté bien implementada esta versión, pero diría que no pues según lo visto en clase es de este modo. Lo que sí es verdad, es que con mutex, es más fácil programar la versión en paralelo que sin mutex, pues sin mutex debemos crear un parámetro que en cada hilo guarde un nuevo valor, y una vez finalizados todos los hilos, crear un bucle for que recorra todas las estructuras de datos y acceder a la suma donde se ha guardado dicha información de cada hilo y hacer a una suma iterativa de esta para así poder trabajar con este dato en el main o donde se requiera. Con mutex, es tan simple como crear una variable global, y cuando se desee escribir ahí dentro de un hilo, cerrar la llave, escribir y volverla a abrir una vez ya esté modificada nuestra variable. Aparte, al ser una variable global, tenemos acceso desde cualquier sitio de nuestro programa.

Actividad 4

¿Qué es OpenMP?

Es una API (interfaz de programación de aplicaciones) que nos permite añadir concurrencia a un código mediante paralelismo con memoria compartida. Se basa en la creación de hilos de ejecución paralelos compartiendo las variables del proceso padre que los crea.

¿Para qué se utiliza?

OpenMP ha sido desarrollado específicamente para procesamiento de memoria compartida en paralelo. De hecho, se ha vuelto al estándar de paralelización en este tipo de arquitecturas.

¿Qué necesito para utilizar OpenMP?

En primer lugar, necesitaremos instalar un entorno de programación donde poder desarrollar este lenguaje. Algunos de estos entornos son: Gedit y SublimeText. Sobre estos entornos de programación podemos trabajar con varios compiladores de diferentes lenguajes. La versión de OpenMP soportada dependerá de la versión del compilador instalado. A continuación, se muestra la lista completa de los diferentes compiladores que implementan las últimas actualizaciones de la API OpenMP.

OpenMP está diseñado para los lenguajes Fortran, C y C++.

Actividad 5

Comentar la siguiente función:

```
double calcularProductoEscalarOMP(void){  
    double productoEscalar = 0.0;  
    #pragma omp parallel for reduction(+:productoEscalar) num_threads(numHilos)  
    for(int i = 0; i<longitudVector; i++){  
        productoEscalar += (vectorA[i]*vectorB[i]);  
    }  
    return productoEscalar;  
}
```

Lo único nuevo que vemos respecto a lo aprendido anteriormente es la función: `#pragma omp parallel for reduction(+:productoEscalar) num_threads(numHilos)` que hace, repartir las iteraciones del `for` entre todos los hilos que le hayamos pasado como parámetro(`numHilos`). `Reduction(+:productoEscalar)`, hace una operación privada en cada iteración y al final suma cada resultado.

¿Qué cree que ofrecen entornos como OpenMP con respecto a PThreads?

Más facilidades a la hora de realizar un programa con paralelismo, pues al fin y al cabo, trata de mejorar lo que PThreads realiza, pero mucho más fácil. En el ejemplo que nos ponen, con una simple línea delante del bucle `for`, elimina la creación de hilos, reserva dinámica de este, el bucle `for` para saber que todos han terminado, realizar estructuras de parámetros para poder pasarle a nuestro hilo, pues con OpenMP este las hereda, y bastantes mejoras. Pero claro, si aprendes directamente con OpenMP, no entiendes lo que de verdad realiza el programa, pues como ya he dicho antes, se carga de una sentencia toda la “guerra” que da PThreads (carga hilos, lanzalos, crea estructuras, ect...)