Universidad Nacional de La Plata

Facultad de Informática e Ingeniería

Ingeniería en Computación

UNIVERSIDAD NACIONAL DE LA PLATA

Sistemas Distribuidos y Paralelos 2019

Trabajo Práctico Integrador

Integrantes:

Corrao, Santiago Vicente 1454/8

Abba, Pedro Nicolas 1257/5

Sistemas distribuidos y paralelos
Trabajo Integrador

Trabajo sobre memoria compartida

Introducción

El trabajo consiste en la resolución de dos algoritmos diferentes realizando su implementación secuencial y a su vez su paralelización. En ambos ejercicios el algoritmo secuencial debe ser lo más óptimo posible.
Por otra parte también se deben realizar cálculos de tiempos de ejecución, junto con su Speedup y Eficiencia.

A-1 Optimización de algoritmos secuenciales y modelo de memoria compartida

En la primera parte del trabajo integrador se nos pide trabajar con el uso de memoria compartida mediante las librerías de Pthread y OpenMP, para la resolución de la siguiente expresión:

$$R = minA \cdot (AL) + maxA \cdot (AA) + promA\,(UA)$$

- A es una matriz de N x N.
- L y U son matrices de NxN triangulares inferior y superior, respectivamente.
- Los escalares minA y maxA son el mínimo y el máximo valor de los elementos de la matriz A, respectivamente.
- El escalar promA es el valor promedio de los elementos de la matriz A.

Para la ejecución del código paralelo se nos pide realizarlo con matrices de tamaño igual a 512, 1024, 2048 y mediante el uso de 2 y 4 hilos.

A-2 Código Secuencial

Para la resolución de este problema decidimos dividir el procesamiento de la expresión en cuatro partes. A continuación se realizará una breve explicación de cada parte del código, incluyendo su inicialización y verificación acompañado de las imágenes correspondientes al código.

Previamente a los procesos correspondientes a la inicialización se comprueba que el usuario haya ingresado al menos un argumento, el cual corresponde a la dimensión de las matrices.
Continuando con la inicialización, en primer lugar realizamos el cálculo de la dimensión que tendrán la matrices triangulares que será utilizado para la reserva de memoria de las mismas.

```
    if (argc < 2){
        printf("\n Falta un argumento:: N dimension de la matriz");
        return 0;
    }
//Convierto la dimension de N a entero
    N=atoi(argv[1]);
//Calculo la dimension del vector para las matrices triangulares
    for (i=0;i<N;i++)
    {
        DimTriangular += N-i;
    }
```

Figura A.2.1

Una vez calculado la dimensión de las matrices triangulares se procede con la reserva de memoria tal y como la conocemos haciendo uso de la función malloc de todas las matrices a utilizar.
Por otro lado se realiza también la inicialización de todas las matrices con valor 1, esto es para una vez realizado el cálculo final poder este resultado con el conocido con anterioridad en base a estos valores de inicialización.
En el caso de las matrices triangulares decidimos no tener el cuenta los ceros tanto en su procesamiento como en el cálculo de su respectiva dimensión. Con esta decisión que tomamos tanto en su procesamiento como en su inicialización (lo cual se puede ver en la Figura A.2.2) al momento de procesar dichas matrices hacemos usos de sus índices para acceder a ellas sin tener en cuenta sus ceros (dependiendo si la matriz triangular es inferior o superior es como realizamos la comparación entre índices para su acceso).

```
//Aloca memoria para las matrices
    A  =(double*)malloc(sizeof(double)*N*N);
    U  =(double*)malloc(sizeof(double)*DimTriangular);
    L  =(double*)malloc(sizeof(double)*DimTriangular);
    AA =(double*)malloc(sizeof(double)*N*N);
    AL =(double*)malloc(sizeof(double)*N*N);
    UA =(double*)malloc(sizeof(double)*N*N);
    R  =(double*)malloc(sizeof(double)*N*N);

//Inicializar las matrices
for(i=0;i<N;i++){
    for(j=0;j<N;j++){
        A[i*N+j]=1;
        if(i>=j){
            L[j+(i*(i+1))/2]=1;
        }
        if(j>=i){
            U[i+(j*(j+1))/2]=1;
        }
        AA[i+j*N]=0;
        AL[i*N+j]=0;
        UA[i*N+j]=0;
        R[i*N+j]=0;
    }
```

Figura A.2.2

En la primera parte del procesamiento realizamos la multiplicación de la matriz A por sí misma, donde a su vez aprovechamos el recorrido que realizamos en dicha matriz para realizar el cálculo de su mínimo, máximo y promedio. Decidimos realizar la multiplicación de la matriz accediendo en ambos términos de la multiplicación por filas para disminuir los fallos de caché, ya que si optamos por acceder en alguno de los términos por columna estos fallos aumentaran.
Otra posible solución era la de realizar en un primer lugar el cálculo de la transpuesta de la matriz A y realizar la multiplicación con la propia matriz, en este caso disminuiría mucho mas los fallos de cache pero tambien tendriamos un aumento del tiempo producto de realizar el cálculo de su transpuesta.

```
//Inicio de procesamiento de A

    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            if (A[i+N*j] < MinA){ MinA = A[i+N*j]; }
            if (A[i+N*j] > MaxA){ MaxA = A[i+N*j]; }
            PromA += A[i+N*j];
            for(k=0;k<N;k++){
                AA[i*N+j]= AA[i*N+j] + A[i*N+k]*A[i*N+k];
            }
        }
    }
    PromA = PromA / (N*N);
//Fin de procesamiento A
```

Figura A.2.3

Una vez resuelto el cálculo de la matriz A por sí misma, procedemos con la resolución de la multiplicación de A por la matriz triangular inferior, haciendo provecho también de que poseemos el valor mínimo de A y al momento de guardar el resultado de la multiplicación realizamos su respectivo producto con el mínimo de A como indica la expresión.
Para el acceso de la matrices, en el caso del resultado realizamos un acceso por filas mientras que para el producto se realiza un acceso cruzado, en el cual se accede por columnas a la matriz A, mientras que a la triangular la accedemos por filas. Además de optimizar la multiplicación realizando el acceso cruzado, también tenemos en cuenta lo anteriormente dicho de la matrices triangulares con respecto a los ceros, debido a que estos no los guardamos en el último “for” se puede ver claramente como ignoramos las posiciones donde se estarían los respectivos ceros de estas matrices.

```
//Inicio de Procesamiento minA*(AL)
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            for(k=i;k<N;k++){
                AL[i*N+j]= AL[i*N+j] + A[i*N+k]*L[k+(i*(i+1))/2];
            }
            AL[i*N+j] *= MinA;
        }
    }
    //Fin de procesamiento minA*(AL)
```

Figura A.2.4

De la misma manera se procede para realizar la multiplicación de la matriz A por la matriz triangular superior (accediendo de la misma forma que la anterior multiplicación) y haciendo uso del promedio de A, el cual fue calculado con anterioridad al igual que su mínimo y máximo.

```
//Inicio de Procesamiento PromA*(UA)
    for(k=0;k<N;k++){
        for(j=0;j<N;j++){
            for(i=0;i<=k;i++){
                UA[j*N+i]= UA[j*N+i] + A[k*N+i]*U[i+(k*(k+1))/2];
            }
            UA[j*N+i] *= PromA;
        }
    }
//Fin de procesamiento PromA*(UA)
```

Figura A.2.5

En el último paso lo que resta son dos operaciones, en un principio realizar el producto del resultado de la multiplicación de la matriz A por sí misma con el máximo de dicha matriz.
Dicho cálculo se puede realizar en conjunto con la suma de los demás resultados previamente calculados para así obtener el resultado final.

```
//Inicio de Procesamiento R
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            R[i*N+j] = AL[i*N+j] + (AA[i*N+j] * MaxA) + UA[i*N+j];
        }
    }
////////////////////////////
/////FIN PROCESAR////////////
////////////////////////////
```

Figura A.2.6

Como se dijo previamente la inicialización de todas las matrices en 1 tiene como objetivo poder realizar una comprobación del resultado final sabiendo el resultado que se espera, esto se puede apreciar en la Figura A.2.7 donde se muestra el algoritmo que se encarga de esto.

```
int check = 1;
 //Verifica el resultado
  for(i=0;i<N;i++){
   for(j=0;j<N;j++){
    check=check&&(R[i*N+j]==3*N-i-j);
   }
  }
```

Figura A.2.7

En nuestro algoritmo además de permitir la verificación del resultado mostrando si esta es correcta o incorrecta, también se puede mediante el cambio de valor de una variable denominada “PRINT” mostrar todas las matrices utilizadas en el cálculo, incluyendo la matriz resultante.

```
if (PRINT){
//PRINT
 printf("\r\nImprimiendo Matriz A:\r\n");
imprimirMatriz(A);
 printf("\r\nImprimiendo Matriz L:\r\n");
imprimirMatrizTI(L);
 printf("\r\nImprimiendo Matriz U:\r\n");
imprimirMatrizTS(U);
 printf("\r\nImprimiendo Matriz AL:\r\n");
imprimirMatriz(AL);
 printf("\r\nImprimiendo Matriz AA:\r\n");
imprimirMatriz(AA);
 printf("\r\nImprimiendo Matriz UA:\r\n");
imprimirMatriz(UA);
 printf("\r\nImprimiendo Matriz R:\r\n");

imprimirMatriz(R);
}
```

Figura A.2.8

A-3 Código paralelizado con Pthread

El siguiente paso una vez realizado el algoritmo secuencial optimizado es paralelizarlo mediante la utilización de la librería pthread, para esto decidimos dividir el procesamiento de las cuatro partes contadas anteriormente entre los hilos que se crearán para su resolución.

Para la utilización de la librería pthread se debe incluir la siguiente línea al inicio del código:

#include <pthread.h>

Para la compilación del código utilizando pthread se debe usar la siguiente línea:

gcc -pthread -o Salida Codigo.c

Librería Pthread-Tipos utilizados

- Threads/Hilos: pthread_t
- Mutex: pthread_mutex_t
- Barrera: pthread_barrier_t

Librería Pthread- Funciones utilizadas para los hilos

Función	Descripción	Parámetros (en orden)
pthread_create	Crea un hilo	1: salida, puntero a id del hilo 2: entrada, para definir atributos del hilo, null para default / const pthread_at 3: entrada, función a correr por el hilo 4: entrada, argumento de la función del hilo. La función debe retornar un * void, el cual es interpretado como el estatus de término por pthread_join
pthread_join	Espera por el término de un hilo	1: Hilo por el cual se espera 2: Variable de retorno que devuelve THREAD_CANCELED si el hilo fue cancelado
pthread_exit	Termina el hilo sin terminar el proceso	Null

Librería Pthread - Funciones utilizadas para los mutex

Función	Descripción	Parámetros (en orden)
pthread_mutex_init	Permite dar las condiciones iniciales a un mutex.	1: Mutex a utilizar 2: Null
pthread_mutex_destroy	Destruye la variable (de tipo pthread_mutex_t) usada para manejo de exclusión mutua.	1: Mutex a destruir
pthread_mutex_lock	Permite solicitar acceso al mutex, el hilo se bloquea hasta su obtención.	1: Mutex a solicitar acceso
pthread_mutex_unlock	Permite liberar un mutex.	1: Mutex a liberar

En el caso de la inicialización del mutex nosotros utilizamos una variación a esto donde al momento de definir el mutex se le asigna una constante propia de la librería que equivale a inicializarla, esta constante es: PTHREAD_MUTEX_INITIALIZER.

Librería Pthread - Funciones utilizadas para las barreras

Función	Descripción	Parámetros (en orden)
pthread_barrier_init	Permite dar las condiciones iniciales a una barrera.	1: Barrera a utilizar 2: Atributos de la barrera, que se pueden establecer en NULL para usar atributos predeterminados 3:Número de hilos que deben esperar llamada pthread_barrier_wait en esta barrera antes de que los hilos puedan continuar.
pthread_barrier_wait	Permite detener la ejecución de los hilos hasta que todos alcancen la barrera.	1:Barrera a utilizar
pthread_barrier_destroy	Destruye una barrera	1: Barrera a destruir

Para dividir el procesamiento de cada uno de los for entre los hilos correspondientes decidimos realizar repartir las filas de las matrices involucradas en la operación entre los hilos creados.
A continuación se detallaran las partes principales del algoritmo realizado con pthread omitiendo las partes ya explicadas anteriormente.

Como parámetros de entrada para la ejecución del algoritmo exigimos dos, siendo estos el tamaño de la matriz N*N y el número de hilos a crear, estos dos argumentos se comprueban que no sean inválidos y además se realiza una segunda comprobación donde se exige que el número de hilos sea múltiplo del tamaño de las matrices. (para así poder dividir el trabajo en partes iguales) y por otro lado que el número de hilos no supere al tamaño de las matrices.

```
N=atoi(argv[1]);
NUM_THREADS = atoi(argv[2]);
if (N < NUM_THREADS && N % NUM_THREADS == 0)
{
    printf("\n N debe ser Mayor o igual a %d \n",NUM_THREADS);
    return 0;
}
```

Figura A.3.1

La creación e inicialización de las matrices sigue siendo la misma que la explicada en el código secuencial, la única diferencia es la creación e inicialización correspondiente a los hilos mediante el

uso de la librería pthread. Sumado a la creación correspondiente de los hilos, también haremos uso de mutex y barreras que nos provee esta misma librería para el control de la ejecución de los hilos y el acceso a variables compartidas.
Con respecto a la creación de hilos realizamos una suposición donde no consideramos al master como un hilo más, este mismo únicamente se encargará de las inicializaciones necesarias, la creación de los hilos,barrera y mutex a utilizar y de su finalización.
Dichos threads e ids de estos mismos se encuentran guardados en un arreglo, el cual utilizamos para la creación y la espera por la finalización de cada uno de ellos.

```
    pthread_t threads[NUM_THREADS];
    int rc;
    int t;
    long threads_ids[NUM_THREADS];

//Inicio de procesamiento de A
    pthread_barrier_init(&barrera, NULL, NUM_THREADS);
    for(t=0; t<NUM_THREADS; t++){
        threads_ids[t]=t;
        rc = pthread_create(&threads[t], NULL, ProcesarParalelo, (void *)threads_ids[t]);
        if (rc){
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(-1);
        }
    }
    for(t=0; t<NUM_THREADS; t++){
        pthread_join(threads[t],NULL);
    }
    pthread_mutex_destroy(&PromA_mutex);
    pthread_barrier_destroy(&barrera);
```

Figura A.3.2

A todos los threads creados le pasamos la misma función a ejecutar denominada ProcesarParalelo, y como parámetro le pasamos sus respectivos ids los cuales serán utilizados para la división de la porción de matriz a trabajar.
En dicha función la cual ejecutan cada uno de los hilos es donde se realiza la utilización de la barrera La barrera es utilizada antes de finalizar la última etapa donde se realiza la suma de todas las matrices generadas en las anteriores etapas, dando así al resultado final, dado esto no se puede realizar este cálculo hasta que se tenga todos los resultados anteriores.

```
void *ProcesarParalelo(void * threadid)
{
    long tid;
    tid = (long)threadid;
    ProcesarA(tid);
    ProcesarAL(tid);
    ProcesarUA(tid);
    pthread_barrier_wait(&barrera);
    ProcesarR(tid);
    pthread_exit(NULL);
}
```

Figura A.3.3

Para la división de trabajo entre los hilos haremos uso del id correspondiente a cada hilo y se realizará dividiendo la matriz por filas como se dijo anteriormente, eso lo haremos parametrizando el segundo for de cada una las multiplicación donde el rango de este quedará comprendido entre:

$$[\,Tid \times (\frac{N}{NUM\_THREADS})\,;(Tid+1) \times (\frac{N}{NUM\_THREADS})\,]$$

Tid: id del thread

N: Tamaño de la matriz

Num_Threads: Número de threads creados

Dicho esto el procedimiento para la resolución del algoritmo sigue siendo el mismo que el secuencial, a excepción del cálculo del mínimo, máximo y el promedio. Dado que ahora cada proceso ejecuta una parte de la matriz, cada uno deberá realizar el cálculo del mínimo, máximo y la sumatoria de los valores al momento de realizar la multiplicación de la porción de A por sí misma que le fue otorgada. Cuando dicha multiplicación finalice cada proceso, además de calcular su promedio local con la sumatoria calculada, deberá entrar a una sección crítica mediante el uso de los mutex anteriormente mencionados para actualizar si fuera necesario el mínimo y el máximo global, y por último sumar al promedio global su respectivo promedio local (Figura A.3.4).
Otra diferencia con respecto al secuencial es que al momento de realizar los procedimientos respectivos de la matriz A con la triangular inferior y superior no realizamos su multiplicación con el mínimo y su promedio hasta que se realice la suma completa de todos los cálculos en ProcesarR.

A continuación se mostrará cada una de las funciones que ejecutan los threads en la función ProcesarParalelo(Figura A.3.3).

```
void ProcesarA(long tid)
{
   double lMinA,lMaxA,lPromA;
   int i,j,k;
   lMinA = A[0];
   lMaxA = A[0];
   for(i=0;i<N;i++){
       for(j=tid*(N/NUM_THREADS);j<(tid+1)*(N/NUM_THREADS);j++){
           if (A[i+N*j] < lMinA){ lMinA = A[i+N*j]; }
           if (A[i+N*j] > lMaxA){ lMaxA = A[i+N*j]; }
           lPromA += A[i+N*j];
           for(k=0;k<N;k++){
               AA[i*N+j]= AA[i*N+j] + A[i*N+k]*A[i*N+k];
           }
       }
   }
   lPromA /= (N*N);
   pthread_mutex_lock(&PromA_mutex);
   /**/if (lMinA < MinA){ MinA = lMinA; }
   /**/if (lMaxA > MaxA){ MaxA = lMaxA; }
   /**/PromA += lPromA;
   pthread_mutex_unlock(&PromA_mutex);
}
```

Figura A.3.3

```
////////////////////////////////////////////////////////////////////////////////
void ProcesarAL(long tid)
{
   int i,j,k;
   for(i=0;i<N;i++){
       for(j=tid*(N/NUM_THREADS);j<(tid+1)*(N/NUM_THREADS);j++){
           for(k=i;k<N;k++){
               AL[i*N+j]= AL[i*N+j] + A[i*N+k]*L[k+(i*(i+1))/2];
           }
           //AL[i*N+j] *= MinA;
       }
   }
}
////////////////////////////////////////////////////////////////////////////////
```

Figura A.3.4

```
////////////////////////////////////////////////////////////////////////////////
void ProcesarUA(long tid)
{
   int i,j,k;
   for(k=0;k<N;k++){
       for(j=tid*(N/NUM_THREADS);j<(tid+1)*(N/NUM_THREADS);j++){
           for(i=0;i<=k;i++){
               UA[j*N+i]= UA[j*N+i] + A[k*N+i]*U[i+(k*(k+1))/2];
           }
           //UA[j*N+i] *= PromA;
       }
   }
}

////////////////////////////////////////////////////////////////////////////////
```

Figura A.3.5

```
////////////////////////////////////////////////////


void ProcesarR(long tid)
{
   int i,j;
   for(i=0;i<N;i++){
       for(j=tid*(N/NUM_THREADS);j<(tid+1)*(N/NUM_THREADS);j++){
           R[i*N+j] = (AL[i*N+j]*MinA) + (AA[i*N+j] * MaxA) + (UA[i*N+j]*PromA);
       }
   }
}

////////////////////////////////////////////////////
```

Figura A.3.6

A-4 Código paralelizado con OpenMP

Como parte del trabajo también se nos pidió la paralelización del código secuencial esta vez usando la librería OpenMp.

Para la utilización de dicha librería se debe agregar a la cabecera del código la siguiente directiva:

#include<omp.h>

Para la compilación de un código realizado mediante la utilización de la librería OpenMP se debe usar el siguiente formato:

gcc -fopenmp -o Salida Codigo.c

A diferencia de la librería Pthread en OpenMP no es necesario la creación de hilos con anterioridad, si no que se hace uso de diferentes directivas para la división de tareas, donde cada una realiza su ejecución en paralelo y luego se esperan en un join para continuar con su ejecución secuencial.

En nuestro código solo uso de la siguiente directiva proveída por la librería:

omp parallel for

Dicha directiva se encarga de distribuir las interacciones de un loop entre los hilos disponibles. Teniendo en cuenta esto lo que hicimos fue la aplicación de esta directa en cada uno de los loop distribuyendo las interacciones correspondientes a las filas, a excepción de la última etapa encargada de sumar los resultados obtenidos en los anteriores procesos donde decidimos paralelizar el primer loop.

Además de proveernos paralelización, OpenMP nos permite realizar la gestión de datos entre los diferentes hilos. Esto fue necesario para declarar en cada uno de los loops internos como privado sus índices, además de utilizar una segunda cláusula denominada reduction para indicar que se calculará el mínimo, máximo y promedio entre los diferentes procesos evitando así el uso de mutex para secciones críticas.

Al igual que en el secuencial las inicializaciones siguen siendo las mismas, y como argumentos de entrada al programa se nos pide tanto el tamaño de la matriz como el número de hilos que se quiere, teniendo la misma comprobación que en pthread.

A continuación se adjuntan las imágenes de cada una de las etapas realizadas en OpenMP:

```
for(i=0;i<N;i++){
    #pragma omp parallel for private (k) reduction(+:PromA) reduction(min:MinA) reduction(max:MaxA)
    for(j=0;j<N;j++){
        if (A[i+N*j] < MinA){ MinA = A[i+N*j]; }
        if (A[i+N*j] > MaxA){ MaxA = A[i+N*j]; }
        PromA += A[i+N*j];
        for(k=0;k<N;k++){
            //En el caso de A*A se realiza el acceso simultaneo que es mejor que cruzado.
            AA[i*N+j]= AA[i*N+j] + A[i*N+k]*A[i*N+k];
        }
    }
}
PromA = PromA / (N*N);
```

Figura A.4.1

```
//Inicio de Procesamiento minA*(AL)

    for(i=0;i<N;i++){
        #pragma omp parallel for private (k)
        for(j=0;j<N;j++){
            for(k=i;k<N;k++){
                //En el caso de A*A se realiza el acceso simultaneo que es mejor que cruzado.
                AL[i*N+j]= AL[i*N+j] + A[i*N+k]*L[k+(i*(i+1))/2];
            }
    //      AL[i*N+j] *= MinA;
        }
    }
    //Fin de procesamiento minA*(AL)
```

Figura A.4.2

```
//Inicio de Procesamiento PromA*(UA)


    for(k=0;k<N;k++){
        #pragma omp parallel for private (i)
        for(j=0;j<N;j++){
            for(i=0;i<=k;i++){
                //En el caso de A*A se realiza el acceso simultaneo que es mejor que cruzado.
                UA[j*N+i]= UA[j*N+i] + A[k*N+i]*U[i+(k*(k+1))/2];
            }
    //      UA[j*N+i] *= PromA;
        }
    }
//Fin de procesamiento PromA*(UA)
```

Figura A.4.3

```
//Inicio de Procesamiento R

    #pragma omp parallel for private (j)
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            R[i*N+j] = AL[i*N+j] + (AA[i*N+j] * MaxA) + UA[i*N+j];
        }
    }
```

Figura A.4.4

Análisis de la Ejecución:

Para la ejecución del código paralelo se nos pide realizarlo con matrices de tamaño igual a 512, 1024, 2048 y mediante el uso de 2 y 4 hilos.

Datos de las ejecuciones:

Para obtener los datos presentados a continuación se realizaron 3 ejecuciones en las mismas condiciones del mismo algoritmo y se tomó el promedio de dichas mediciones.
Los valores están expresados en Segundos

Promedio de Tiempos de Ejecución					
Tamaño de Matriz	Secuencial	Pthread-2 hilos	Pthread-4 hilos	OpenMP - 2 hilos	OpenMP - 4 hilos
512	1,5573304	0,8180906	0,4335496	0,9108556	0,4870184
1024	12,2705516	6,4123114	3,3210476	7,0231694	3,695023
2048	97,4465176	50,7947908	26,3296664	55,7525892	29,2795708

Como se puede apreciar en la anterior tabla, los tiempos de ejecución mejoran un gran porcentaje en comparación al secuencial comparándolo contra Pthread, teniendo aproximadamente la mitad del tiempo con el uso de 4 hilos. En el caso de OpenMP no se tiene una gran mejora como en Pthread cuando se utilizan 2 hilos aunque si mejora al momento de utilizar 4 hilos pero aun asi no superan los tiempos dados con Pthread con 4 hilos.

Cálculo del SpeedUp:

El cálculo del SpeedUp de forma general se hace mediante la fórmula $S = \frac{Tiempo\ Secuencial}{Tiempo\ Paralelo}$.

Promedio de Tiempos de Ejecución					
Tamaño de Matriz	Secuencial	Pthread-2 hilos	Pthread-4 hilos	OpenMP - 2 hilos	OpenMP - 4 hilos
512	1	1,903616054	3,592046677	1,709744552	3,19768288
1024	1	1,913592593	3,694783417	1,747153016	3,320832265
2048	1	1,918435258	3,701016037	1,747838423	3,328140234

En la tabla anterior podemos apreciar los SpeedUp correspondiente a cada uno de los métodos e hilos utilizados, donde en primera instancia podemos ver que el mejor speedUp lo tenemos con un tamaño de 1024, mientras que al aumentar dicho tamaño el SpeedUp no mejoró. En el caso de OpenMP su SpeedUp no es tan bueno como se espera al compararlo con Pthread.

Cálculo de la Eficiencia:

El cálculo del SpeedUp de forma general se hace mediante la fórmula $S = \frac{Tiempo\ Secuencial}{Cantidad\ Cores\ x\ 100}$.

Promedio de Tiempos de Ejecución					
Tamaño de Matriz	Secuencial	Pthread-2 hilos	Pthread-4 hilos	OpenMP - 2 hilos	OpenMP - 4 hilos
512	100	95,18080271	89,80116693	85,48722761	79,942072
1024	100	95,67962966	92,36958543	87,35765081	83,02080664
2048	100	95,92176291	92,52540093	87,39192116	83,20350584

Conclusiones:

Con respecto a la eficiencia no podemos llegar a concluir que el programa es fuertemente escalable debido a que al aumentar el número de hilos tanto en OpenMp como en Pthread manteniendo el tamaño la eficiencia no produce un aumento. A pesar de que aumentando tanto el tamaño como el número de hilos a la vez la eficiencia se mantiene moderadamente constante, no podemos decir que el programa es débilmente escalable debido a que para un dado número de hilos al aumentar el tamaño no recupera su eficiencia perdida.

Aun así consideramos que el programa que realizamos es considerablemente bueno debido a que estamos obteniendo para la mayoría de los casos una eficiencia arriba del 80% lo que se puede considerar un buen resultado.

También podemos concluir que la eficiencia resulta mejor al utilizar la librería de Pthread con respecto a la librería de OpenMP.

Trabajo sobre memoria distribuida.

Problema a planteado

Resolver con MPI el problema de N-Reinas por demanda (modelo Master-Worker).

El juego de las N-Reinas consiste en ubicar sobre un tablero de ajedrez N reinas sin que estas se amenacen entre ellas. Una reina amenaza a aquellas reinas que se encuentren en su misma fila, columna o diagonal. La solución al problema de las N-Reinas consiste en encontrar todas las posibles soluciones para un tablero de tamaño NxN. En una estrategia Master-Worker, el proceso Master realiza cierto cálculo inicial y luego entrega una cantidad de trabajo específica a los workers cuando estos lo requieren. Cada proceso Worker realiza cómputo y entrega al proceso Master los resultados. Dependiendo de la aplicación, el proceso Mater podría o no trabajar asumiendo transitoriamente el rol de Worker.

Evaluar para N entre 5 y 15 utilizando 2 máquinas:

a.4 procesos (2 en cada máquina)

b.8 procesos (4 en cada máquina)

Se debe obtener la cantidad total de soluciones encontradas NO los tableros con las soluciones.

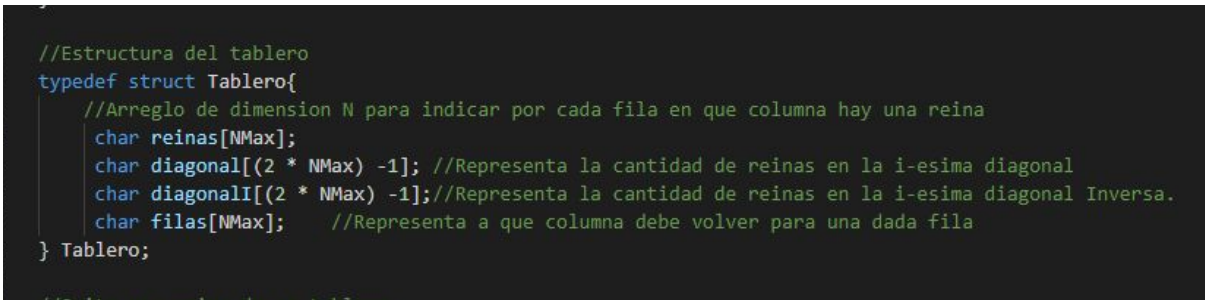
Solución Secuencial

La primera solución que se encuentra al problema de las N reinas teniendo en cuenta la complejidad de la implementación es la solución recursiva, esta tiene como primera desventaja un intensivo uso de memoria ram ya que debe apilar cada uno de los llamados a las funciones, y por otra parte se desaconsejó desde la cátedra el uso de algoritmos recursivos ya que traen problemas al momento de paralelizar el algoritmo. Por lo que decidimos hacer la implementación del algoritmo iterativo, éste con respecto al recursivo tiene la desventaja de tener que mantener a "mano" el estado anterior, aunque dicha desventaja no es de mayor relevancia.

Nuestra solución:

El primer desafío con el que nos encontramos al momento de encarar la solución fue la representación del tablero, al cual lo terminamos representando con un vector de N posiciones la cual representa a cada columna y el valor que toma cada posición del arreglo indica en qué fila está la reina, es posible representarlo así dado a que por la naturaleza del problema no puede haber más de una reina en la misma columna, y ni tampoco más de una reina en la misma fila, por lo que todos los valores del arreglo deberían ser distintos. También teniendo en cuenta el manejo de índices que usa el lenguaje de programación definimos que la primera fila / columna va a estar representada con un 0 y la n-ésima fila / columna va a estar representada por el valor N-1 y cuando una posición en el tablero esté libre se usa el valor -1. Además utilizamos 3 arreglos extras, con el fin de facilitar la implementación y mejorar los tiempos de ejecución. Se utilizó un par de vectores para contar la cantidad de reinas en cada dirección de las diagonales estos tienen dimensión 2N - 1 y por último un arreglo* de dimensión N que nos brinda el soporte para mantener el estado anterior de cada iteración en el que cada posición representa una fila y el valor que toma dicha posición es la última columna procesada en dicha fila.

Nota: * Este último arreglo en la solución secuencial se encuentra declarado fuera de la estructura tablero.



```
//Estructura del tablero
typedef struct Tablero{
    //Arreglo de dimension N para indicar por cada fila en que columna hay una reina
    char reinas[NMax];
    char diagonal[(2 * NMax) -1]; //Representa la cantidad de reinas en la i-esima diagonal
    char diagonalI[(2 * NMax) -1];//Representa la cantidad de reinas en la i-esima diagonal Inversa.
    char filas[NMax];    //Representa a que columna debe volver para una dada fila
} Tablero;
```

Buscando las soluciones:

Para la búsqueda de soluciones nuestro algoritmo realiza fuerza bruta sobre el tablero, probando todas las distintas posibles posiciones donde puede ubicar una reina y verificando en cada paso si esa disposición del tablero es válida, si lo es continúa buscando colocar una reina en la fila siguiente, en caso contrario quita la reina colocada y continua por la siguiente columna. Si la fila en la que pudo colocar una reina es la última del tablero, es porque encontró una solución. Para una explicación más en detalle del algoritmo utilizaremos pseudocódigo.

Pseudocódigo

Crear tablero vacío
Mientras no desborde por las columnas
    Mientras no desborde por las filas
        Almacenar el estado actual en filas
        Si no hay reina
            Si puedo ubicar reina
                Ubicar
                Si es la última fila
                    Acumular solución
                    Quitar reina.
. Volver a la fila anterior a la columna indicada
                Si no es la última fila
                    Avanzar una fila
                    Ir a la primera columna
            Si no puedo ubicar reina
                Si es la última columna
                    Volver a la fila anterior a la columna indicada
                Si no es la última columna
                    Avanzar una columna
        Si hay reina
            Si es la última columna

Quitar reina
Retroceder una fila
Si desbordó el tablero
Fin de ejecución.
Si no desbordó
Volver a la columna indicada
Si no es la última columna
Quitar reina
Avanzar una columna

Solución Paralelo:

La solución de forma paralela en cuanto a la mecánica utilizada para la realizar la búsqueda de soluciones es la misma que la utilizada en el caso secuencial, con la particularidad de que ahora el trabajo lo van a realizar varios procesos, siguiendo con la arquitectura solicitada en el planteo del problema generamos un proceso Master que su función es generar tableros que sean válidos pero que no estén completos (los llamaremos semillas) y a partir del pedido por parte de un Worker enviarle uno de estos tableros incompletos para que busque las soluciones. Además deberá recolectar las respuestas de los Workers e indicarles cuando no tiene más trabajo para enviarles.

Generación de las semillas:

El proceso master genera las semillas utilizando la misma forma de funcionamiento que la solución secuencial, con la salvedad de que ahora la cantidad de filas necesarias completas para encontrar una "solución" es dos, por lo que se modificó el algoritmo planteado anteriormente para que cumpla con estos requisitos, además una vez encontrada la solución es necesario hacer una copia del estado del tablero antes de seguir ejecutando.

Si solo se utilizan las dos primeras filas como semillas se puede obtener la cantidad de semillas necesarias mediante la ecuación $S = \frac{N!}{N*(N-3)!}$ lo que se puede simplificar en $S = (N-1)(N-2)$

Búsqueda de Soluciones:

Cada Worker recibe solo el vector de reinas de la estructura completa que usamos con los fines de hacer la comunicación más rápida y sencilla, por lo que cada worker deberá completar la estructura antes de comenzar con la búsqueda de soluciones. Una vez que tenga el tablero completo el Worker empezará a buscar la soluciones de la misma forma que se venía realizando anteriormente, con la diferencia que ahora su trabajo finaliza cuando le toca quitar alguna de las reinas que ya le habían sido fijadas, para ello nuevamente se modificó el algoritmo secuencial para que cumpla con los requisitos.

Master: Pseudocódigo de la implementación

Calcular la cantidad de semillas
Crear e Inicializar tantos tableros como semillas se necesiten
Generar las semillas
Mientras no se terminen las semillas
Si hay algún mensaje pendiente
Recibirlo

Si tiene tag Pedido
Enviar trabajo
Si tiene tag Resultado
Acumular solución
Enviar Trabajo
Mientras todos no hayan terminado
Si hay un mensaje pendiente
Recibirlo
Si tiene tag Resultado
Acumular solución
Enviar Fín de Trabajo

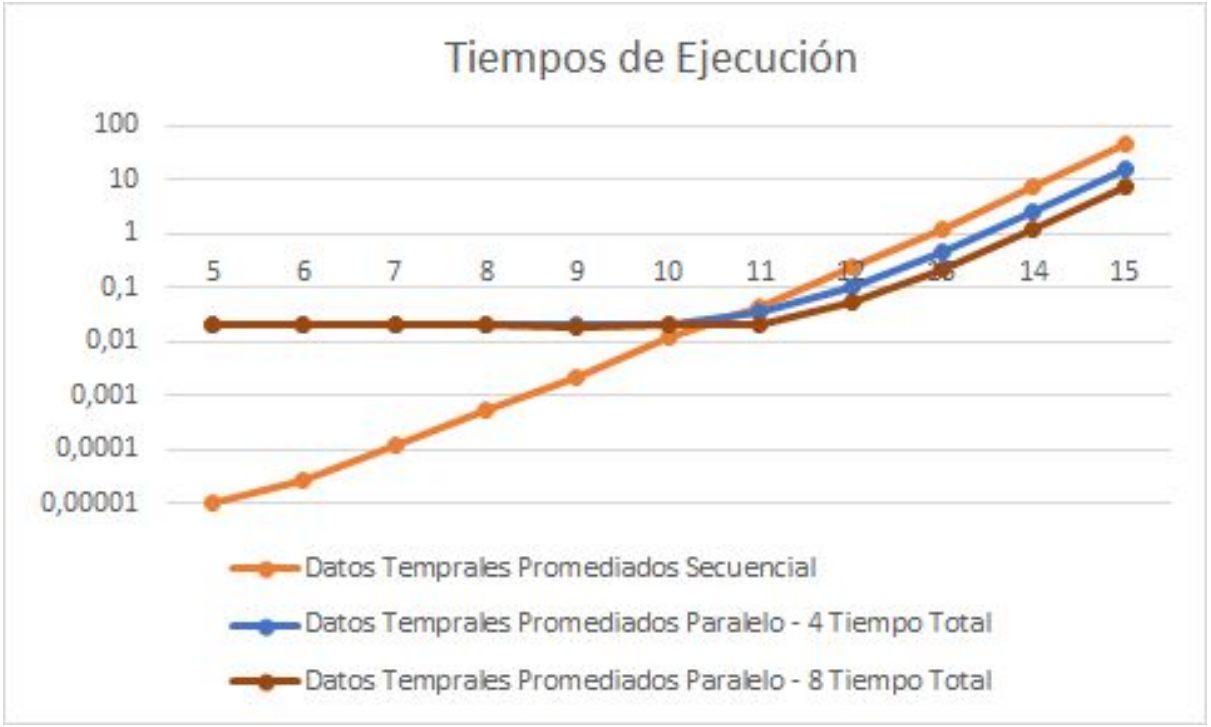
Worker: Pseudocódigo de la implementación
Enviar al master un Pedido
Mientras el master no indique que se terminaron los datos…
Recibir de forma bloqueante
Si recibe datos
Inicializar tablero
Generar Tablero completo con los datos
Buscar soluciones
Enviar al Master las soluciones
Si recibió fin de datos
Finalizar la ejecución

Análisis de la Ejecución:

Datos de las ejecuciones:

Para obtener los datos presentados a continuación se realizaron 3 ejecuciones en las mismas condiciones del mismo algoritmo y se tomó el promedio de dichas mediciones.
Los valores están expresados en Segundos.

Datos Temporales Promediados							
Reinas	Secuencial	Paralelo - 4			Paralelo - 8		
		% Secuencial	% Paralelo	Tiempo Total	% Secuencial	% Paralelo	Tiempo Total
5	0,000010	0,0200	99,9800	0,020955	0,0205	99,9795	0,021497
6	0,000028	0,0343	99,9657	0,020977	0,0294	99,9706	0,020397
7	0,000123	0,0409	99,9591	0,021049	0,0381	99,9619	0,021506
8	0,000536	0,0552	99,9448	0,021013	0,0523	99,9477	0,021781
9	0,002173	0,0716	99,9284	0,020958	0,0788	99,9212	0,019796
10	0,012007	0,1017	99,8983	0,021043	0,0832	99,9168	0,021399
11	0,045039	0,0723	99,9277	0,035131	0,1100	99,8900	0,021813
12	0,236371	0,0301	99,9699	0,105006	0,0540	99,9460	0,055228
13	1,248391	0,0091	99,9909	0,457159	0,0193	99,9807	0,210590
14	7,393726	0,0018	99,9982	2,603476	0,0037	99,9963	1,175156
15	47,905729	0,0003	99,9997	16,666814	0,0008	99,9992	7,505987

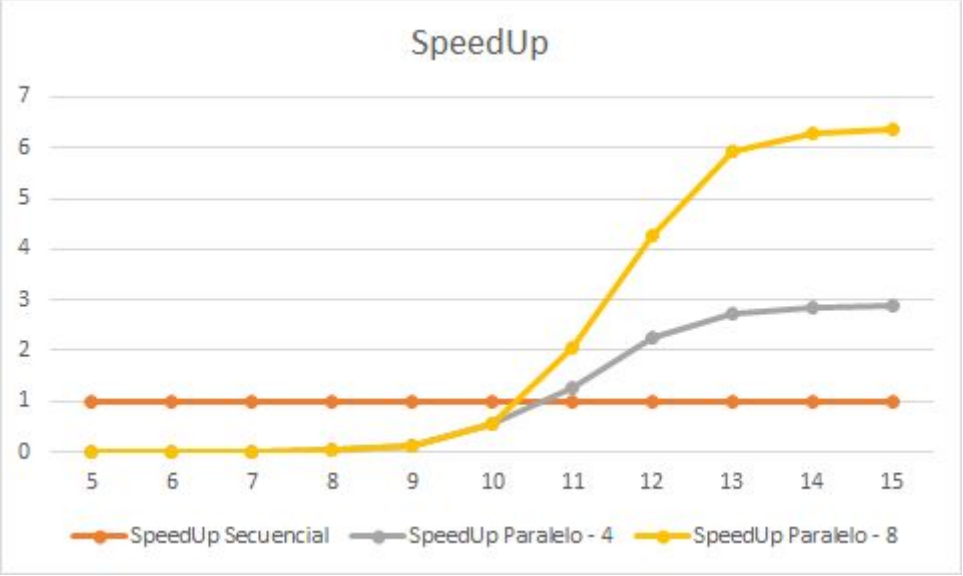


En el gráfico de los tiempo de ejecución se utilizó una escala logarítmica en base 10 en el eje vertical debido a la gran diferencia entre el tiempo de ejecución con N = 5 y con N = 15, esto es de particular importancia ya que al momento de apreciar el gráfico la curva se ve de forma parecida a una lineal, pero realmente es exponencial.

Cálculo del SpeedUp:

El cálculo del SpeedUp de forma general se hace mediante la fórmula $S = \frac{Tiempo\ Secuencial}{Tiempo\ Paralelo}$, en el caso de un cluster completamente homogéneo el tiempo de ejecución secuencial en distintas PC debería ser relativamente similar. En nuestro caso particular el cluster es muy distinto por lo que para el cálculo del SpeedUp consideramos un promedio entre los tiempos obtenidos en cada computadora. $S = \frac{Ts1 + Ts2}{2 * Tp}$

SpeedUp			
Reinas	Secuencial	Paralelo - 4	Paralelo - 8
5	1,000	0,000	0,000
6	1,000	0,001	0,001
7	1,000	0,006	0,006
8	1,000	0,025	0,025
9	1,000	0,104	0,110
10	1,000	0,571	0,561
11	1,000	1,282	2,065
12	1,000	2,251	4,280
13	1,000	2,731	5,928
14	1,000	2,840	6,292
15	1,000	2,874	6,382

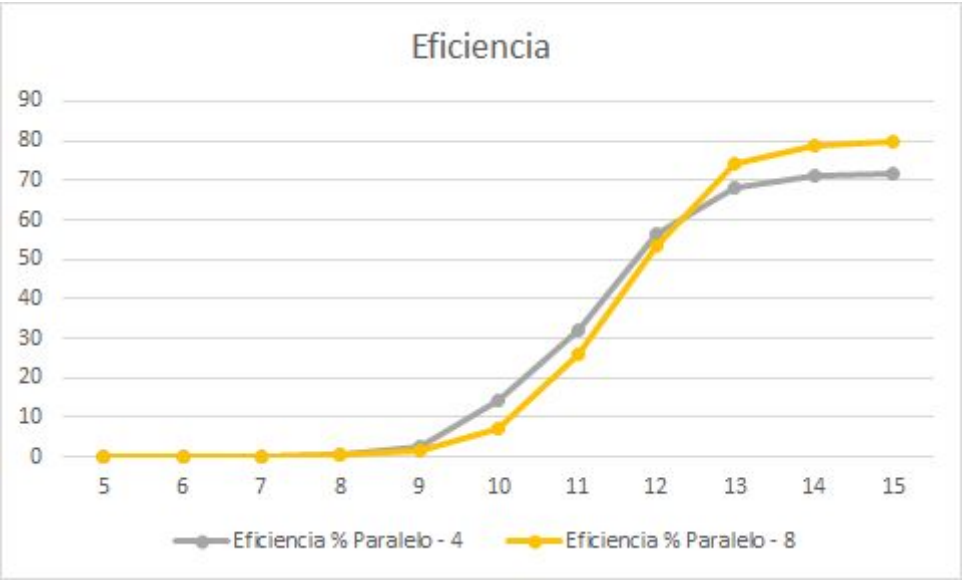


Eficiencia:

La eficiencia es una métrica de rendimiento que indica el porcentaje de tiempo en el que los procesadores están realizando trabajo útil y se puede calcular mediante: $Ef = \frac{SpeedUp}{Número\ de\ Procesadores}$

Valores expresados en porcentaje.

Eficiencia			
Reinas	% Secuencial	% Paralelo - 4	% Paralelo - 8
5	100,000	0,012	0,006
6	100,000	0,034	0,017
7	100,000	0,146	0,071
8	100,000	0,637	0,307
9	100,000	2,593	1,372
10	100,000	14,265	7,014
11	100,000	32,051	25,810
12	100,000	56,276	53,499
13	100,000	68,269	74,101
14	100,000	70,999	78,646
15	100,000	71,858	79,779



Conclusiones:

Antes de hacer un análisis sobre la escalabilidad del sistema, queremos destacar algunos datos particulares obtenidos, si observamos los tiempos de ejecución con un tamaño de problema pequeño, la solución secuencial con respecto a las soluciones paralelas siempre es mejor, y esto se lo atribuimos particularmente a que para el procesamiento paralelo con memoria distribuida existe un procesamiento que es inherente a esta arquitectura de trabajo debido a la necesidad de comunicación entre los procesos.

Ahora si remitiendonos a la definición de escalabilidad.

Definición: Un sistema es escalable si mantiene constante la eficiencia al aumentar el número de procesadores aumentando también el tamaño del problema.

Fuertemente escalable: mantiene la eficiencia constante sin incrementar el tamaño del problema.

Débilmente escalable: mantiene la eficiencia constante al incrementar el tamaño de problema al mismo tiempo que se incrementa el número de procesadores.

En cuanto a nuestro sistema en particular, es fuertemente escalable, debido que a partir de 13 reinas la eficiencia mejora aumentando el número de procesadores y manteniendo el número de procesadores.