Sistemas Distribuidos y Paralelos

Trabajo Integrador

Sánchez, Agustín 939/2

Levy, Santiago 746/4

Consideraciones Iniciales

Todos los ejercicios fueron compilados y ejecutados en los equipos del aula de postgrado. Para el registro de los tiempos de cada programa, se los ejecutó 2 veces y luego se lo promedió.

Para ejecutarlos pusimos a disposición, junto a los archivos de código fuente de cada ejercicio, sus respectivos scripts bash que los ejecutan la cantidad de veces necesaria para obtener todos los datos solicitados, incluyendo las impresiones por consola de los tiempos que llevó completar cada ejecución. En los casos donde fuera necesario se utilizaron *machinefiles* generados en el momento con la cantidad de equipos necesarios y los slots requeridos.

Respecto a los nombres elegidos para las variables, podrá notarse que generalmente **N** será utilizado para el tamaño del problema, **T** para definir la cantidad de threads en el caso de memoria compartida y **P** para la cantidad de procesos, en el caso de memoria distribuida.

Ejercicio 1

Resolver con Pthreads y OpenMP la siguiente expresión:

$$R = minA * AL + maxA * AA + promA * UA$$

- Donde A es una matriz de NxN.
- 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.

Análisis del algoritmo secuencial

Orden:

Se hizo énfasis en lograr un algoritmo secuencial con el menor tiempo de ejecución, para así poder calcular los valores de speedup de las ejecuciones paralelas posteriores.

Para ello, la estrategia secuencial utilizada calcula primero los valores máximo, mínimo y promedio y la transpuesta de la matriz A. Luego realiza secuencialmente las distintas términos de la suma con sus multiplicaciones correspondientes, acumulando por cada uno los valores parciales de la matriz resultado. Dicho orden se puede ver en la siguiente sección de código:

```
22      timetick = dwalltime();
23
24      max_min_prom_trans();
25      mult_maxA_AA();
26      mult_minA_AL();
27      mult_promA_UA();
28
29      printf("Tiempo con N = %d >> %.4f seg.\n", N, dwalltime() - timetick);
30
```

De esta forma se logra reducir la cantidad de procesamiento necesaria para guardar los valores en sus ubicaciones finales y quitando la necesidad de utilizar las matrices intermedias.

Utilización de matriz transpuesta:

Como se comprobó a lo largo de las prácticas de la materia, a la hora de realizar una multiplicación de una matriz por sí misma, resulta más eficiente calcular primero su matriz transpuesta y luego la multiplicación, ya que se puede utilizar la matriz transpuesta como la matriz de A almacenada por columnas, lo que aumenta la localidad espacial de los valores accedidos y disminuye el tiempo de cálculo total de la multiplicación.

Matrices triangulares:

Para poder analizar el efecto de las opciones de almacenamiento de matrices triangulares se realizaron dos programas: Uno en el que las matrices U y L se almacenan con ceros y otro en el que no. A continuación se presenta una tabla con los resultados de tiempo de ejecución.

	Tiempo con N = 512	Tiempo N = 1024	Tiempo N = 2048
Con ceros	1.2684	9.9406	81.2536
Sin ceros	1.0108	7.8403	63.99525

Análisis del algoritmo paralelo con OpenMP

El código del algoritmo paralelo sigue los mismos pasos que el secuencial:

1) Primero se obtienen los valores mínimo, máximo y promedio de la matriz A, como también su transpuesta.
2) Luego se realiza la multiplicación y suma de cada término.

Para este último paso se realizaron dos programas **ejercicio1/openmp/openmp_separado.c** y **ejercicio1/openmp/openmp_junto.c** que analizan las dos opciones disponibles. Esto fue para poder elegir la forma en que se obtengan los menores tiempos de ejecución.

En la primer opción, se paraleliza cada término de la suma de forma que nos queda:

1. Paralelizar **minA . (AL)** y almacenarlo en **R**.
2. Paralelizar **maxA . (AA)** y acumularlo en **R.**
3. Paralelizar **promA . (UA)** y acumularlo en **R,** obteniendo así el valor final.

En la segunda opción, de realizar todo junto, se paraleliza una única vez el cálculo de **R = minA . (AL) + maxA . (AA) + promA . (UA).** Por lo que se paraleliza por secciones de la matriz resultado.

A continuación se presenta una tabla con los resultados obtenidos de los dos algoritmos:

	Threads	Tiempo con N = 1024	Tiempo con N = 2048
Algoritmo con términos separados	2	4.2098	34.5831
	4	2.1922	17.7105
Algoritmo con términos juntos	2	4.0622	32.3552
	4	2.1743	17.0446

Si bien los valores son muy similares, puede observarse que realizar la suma al final termina tardando menos, por lo que se eligió el algoritmo que calcula las multiplicaciones y sumas todo junto. Por lo tanto se procede a explicar la solución elegida.

Primero se lleva a cabo la primera operación en paralelo: El recorrido de la matriz **A** de NxN, obteniendo el valor mínimo, el máximo y una acumulación del total mediante la directiva de OpenMP **reduction**, la que permite obtener los valores finales en base a los cálculos localmente por cada parte paralelizada. También se realiza el cálculo de la matriz transpuesta

```
#pragma omp parallel for private(i,j) reduction(min:minA) reduction(max:maxA) reduction(+:promA)
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
    {
        promA += A[i * N + j];
        if (A[i * N + j] > maxA)
            maxA = A[i * N + j];
        if (A[i * N + j] < minA)
            minA = A[i * N + j];
        At[i * N + j] = A[i + j * N];
    }
```

Una vez finalizado el recorrido paralelo, se realiza secuencialmente la división de la suma por la cantidad de valores, obteniendo así el promedio de los valores de la matriz.

Luego, se define otra operación paralela para las multiplicaciones de **minA.AL**, **promA.UA** y **maxA.AA**. Notar que se incluye la sentencia de planificación **schedule(dynamic, 1)**, la cual permite asignar dinámicamente trabajo a cada proceso para alcanzar un balance de trabajo más equitativo. En caso de no incluirlo, es probable que el hecho de multiplicar por matrices triangulares conlleve un desbalance en el procesamiento que requiere cada fila.

```
#pragma omp parallel for private(i,j,k,accAL, accAA, accUA) schedule(dynamic, 1)
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
    {
        accAL = 0;
        accAA = 0;
        accUA = 0;
        for (k = 0; k < N; k++){
            if (k >= j)
                accAL += A[i * N + k] * Lcol[k + j * N - j * (j + 1) / 2];
            if (k >= i)
                accUA += Ufil[i * N + k - i * (i + 1) / 2] * At[k + j * N];
            accAA += A[i * N + k] * At[k + j * N];
        }
        R[i * N + j] = accAL * minA + accAA * maxA + accUA * promA;
    }
}
```

Se utilizan acumuladores para realizar la multiplicación por el valor máximo, mínimo y promedio, una única vez.

Análisis del algoritmo paralelo con Pthreads

La estrategia utilizada para la resolución mediante el uso de Pthreads difiere levemente de la utilizada para OpenMP. En este caso, la distribución de trabajo en los hilos se realiza de forma estática, es decir, se asigna una determinada cantidad de posiciones a calcular para cada Pthread. El trabajo de cada hilo será una porción de la matriz resultado **R.**

```
for (int t = 0; t < T; t++)
{
    args_array[t].id = t;
    args_array[t].start = N / T * t;
    args_array[t].end = N / T * t + N / T - 1;
    pthread_create(&threads[t], NULL, &t_function, (void *)&args_array[t]);
}
for (int i = 0; i < T; i++)
    pthread_join(threads[i], NULL);
```

Otra diferencia significativa respecto a la solución utilizada para OpenMP radica en el hecho de que los valores mínimo, máximo y promedio se calcularon reduciendo los valores de forma paralela. Cada Pthread inicialmente calcula los valores de mínimo, máximo y

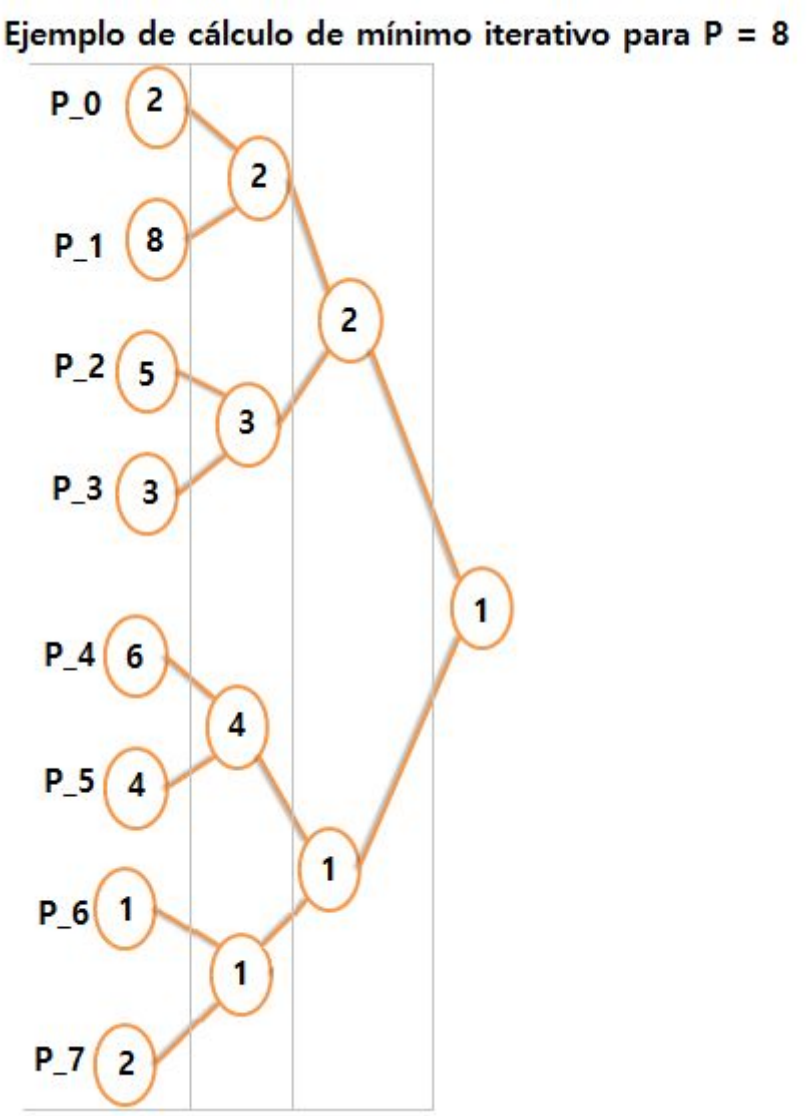
acumulado de los valores que le tocaron. A medida que los distintos pares de Pthreads terminan de procesar de esa manera sus respectivos valores, aquellos que sean de identificador par en la iteración actual se encargan de comparar los valores con los procesados por el pthread siguiente, el cual habrá indicado previamente que finalizó su procesamiento a través del semáforo correspondiente (se usan T - 1 semáforos para evitar problemas de concurrencia). El código que se encarga de esta parte es el siguiente:

```
for (i = args.start; i <= args.end; i++)
    for (j = 0; j < N; j++)
    {
        promA[args.id] += A[i * N + j];
        if (A[i * N + j] > maxA[args.id])
            maxA[args.id] = A[i * N + j];
        if (A[i * N + j] < minA[args.id])
            minA[args.id] = A[i * N + j];
        At[i * N + j] = A[i + j * N];
    }
promA[args.id] /= (N * N);

int aux, offset = 0;
for (int i = 2; i <= T; i *= 2)
{
    if (args.id % i == 0)
    {
        aux = args.id + i / 2;
        sem_wait(&semaphores[args.id / i] + offset); //espero a tener los dos valores
        if (minA[args.id] > minA[aux])
            minA[args.id] = minA[aux];
        if (maxA[args.id] < maxA[aux])
            maxA[args.id] = maxA[aux];
        promA[args.id] += promA[aux];
        offset += T / i;
    }
    else
    {
        sem_post(&semaphores[(args.id - (i / 2)) / i] + offset); //aviso que tengo mi valor disponible
        break;
    }
}

pthread_barrier_wait(&barrier);
```

Para visualizar más fácilmente la forma en la que se calcularon los valores se adjunta una imagen ejemplificando la forma en que se reducen los valores mínimos encontrados por 8 Pthreads. Notar que la cantidad de iteraciones hasta obtener el valor mínimo total será de $log_2(\#Threads)$. Esta forma de obtener los valores mínimo, máximo y suma para el promedio maximiza el paralelismo de esta parte del programa, causando mejoras apreciables en los tiempos de ejecución.



A medida que van finalizando, los hilos se quedan esperando en una barrera hasta que se termine de calcular valores finales necesarios para continuar con las multiplicaciones, que se obtienen de la misma forma que en OpenMP, con la diferencia de que se debe especificar qué filas le corresponden a cada Pthread. A continuación se muestra la parte del código encargada de realizar las multplicaciones de los valores correspondientes y de guardar sus resultados de forma acumulada en la matriz resultado **R,** nuevamente con el fin de evitar utilizar matrices intermedias que requieren espacio y procesamiento adicional:

```
//Mult
for (i = args.start; i <= args.end; i++)
    for (j = 0; j < N; j++)
    {
        accAL = 0;
        accAA = 0;
        accUA = 0;
        for (k = 0; k < N; k++)
        {
            if (k >= j)
                accAL += A[i * N + k] * Lcol[k + j * N - j * (j + 1) / 2];
            if (k >= i)
                accUA += Ufil[i * N + k - i * (i + 1) / 2] * At[k + j * N];
            accAA += A[i * N + k] * At[k + j * N];
        }
        R[i * N + j] = accAL * minA[0] + accAA * maxA[0] + accUA * promA[0];
    }

pthread_exit(NULL);
```

Una vez finalizada la tarea de cada Pthread (a lo cual se aguarda mediante el Pthread_join de cada Pthread en main) se muestran los resultados y se libera el espacio alocado para las estructuras utilizadas.

Resultados finales

En la siguiente tabla se muestra las ejecuciones de cada programa, para los valores solicitados de:

- N = 512, 1024 y 2048
- T = 2 y 4.

Secuencial						
N	512		1.024		2.048	
Tiempo	1,0108		7,8403		63,9953	
OpenMP						
N	512		1.024		2.048	
Threads	2	4	2	4	2	4
Tiempo	0,5096	0,2641	4,0807	2,1524	32,3291	17,0779
SpeedUp	1,9835	3,8273	1,9213	3,6426	1,9795	3,7473
Eficiencia	0,9918	0,9568	0,9607	0,9106	0,9897	0,9368
Pthreads						
N	512		1.024		2.048	
Threads	2	4	2	4	2	4
Tiempo	0,6084	0,3262	4,7834	2,6216	38,2787	21,3087
SpeedUp	1,6614	3,0987	1,6391	2,9907	1,6718	3,0032
Eficiencia	0,8307	0,7747	0,8195	0,7477	0,8359	0,7508

Ejercicio 2

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.

Análisis del algoritmo secuencial

En el archivo con la solución secuencial se encuentran las dos estrategias de soluciones empleadas, la recursiva y la no-recursiva. La recursiva fue planteada inicialmente por ser la más intuitiva, pero ya que la iterativa es más eficiente en el uso de recursos se explorará esta opción debido a que fue la elegida para la paralelización.

El programa comienza obteniendo de los argumentos el tamaño del tablero (NxN) y alocando el espacio necesario en memoria para guardar las posiciones de cada reina. Esta es la forma más óptima de modelar el problema: Se alcanza usando un arreglo de enteros de tamaño N donde cada valor es la fila en la que se encuentra la reina de la columna correspondiente; por ejemplo, un 3 en la posición 0 indica que la reina de la primer columna se encuentra en la cuarta fila (recordar que la notación de los arreglos empieza con 0 para denominar la primer posición).

```
int main(int argc, char *argv[])
{
    double timetick;
    N = atoi(argv[1]);
    queens = malloc(sizeof(int) * N);

    timetick = dwalltime();

    switch (MODE)
    {
    case ITERATIVE:
        get_queens();
        break;

    case RECURSIVE:
        get_queens_recursive(0);
        break;
    }

    printf("N = %d\t| Soluciones = %d\t| Tiempo = %.4f\n", N, total_solutions, dwalltime() - timetick);
    if (valores_correctos[N] != total_solutions)
        printf("\nX\nX\nSolucion incorrecta.\nX\nX\n");

    free(queens);
    return EXIT_SUCCESS;
}
```

El método **get_queens()** recorre las opciones, simulando un recorrido de árbol, chequeando qué posiciones de reina no entran en conflicto con las ya asignadas en el camino de solución actualmente explorado.

Todo posible tablero en el que la reinas actuales coinciden, horizontalmente, verticalmente o en sus diagonales son descartadas. Si durante el recorrido se logra llegar a la última columna N -1, se puede decir que se encontró una solución válida.

Si el camino de recorrido actual llega un punto en el que no contiene ninguna solución válida, realiza un **backtracing** para volver a un estado anterior y poder continuar ejecutando desde una columna anterior.

A partir de este funcionamiento en cascada se desprende que al avanzar de fila en la columna actual puede pasar que:

- La próxima fila a analizar sea N, en cuyo caso se habrá alcanzado el análisis total de la columna. Esto puede pasar en dos casos:
    - La columna actual es la primera del tablero. Eso significa que se revisaron todas las filas de todas las columnas para todas las ubicaciones del tablero. En este caso se sale del método y se finaliza el análisis.
    - La columna actual no es la primera del tablero. En este caso se debe regresar a la columna anterior y avanzar de fila sobre ella. Debe verificarse que al hacer esto no esté llegándose al caso base. Si no lo es, se sigue analizando las posiciones de reinas posibles para la columna actual.
- La próxima fila a analizar no sea N, entonces se realiza el chequeo para garantizar que se sigue sobre un camino candidato a solución válida.

Notar que los casos más relevantes resultan aquellos en donde la posición a analizar alcanza un borde del tablero. En los demás casos simplemente se verifica que la ubicación a analizar no se cruce con ninguna reina ya guardada.

Para cuando el método **get_queens()** completa su ejecución, la variable global **total_solutions** contendrá la cantidad de soluciones distintas encontradas. Finalmente se notifica si la cantidad de soluciones encontradas coincide con las que se sabe que existen para el tamaño de tablero proporcionado y se finaliza liberando el espacio de memoria utilizado.

Análisis algoritmo paralelo con MPI

"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."

Para la resolución paralela se planteó una estrategia Master Worker donde el master además de distribuir los trabajos se ocupa de realizar trabajo durante los intervalos en que no tenga solicitudes de trabajo de los workers, reduciendo la cantidad de tiempo que se podría encontrar ocioso. En principio notar como en main se utilizan métodos distintos para el proceso master y los procesos worker:

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &P);

    N = atoi(argv[1]);
    queens = malloc(sizeof(int) * N);

    if (rank == MASTER_RANK)
        master();
    else
        slave();
    free(queens);
    MPI_Finalize();
}
```

El método master comienza por calcular la cantidad de trabajo a darle a cada proceso, en función de cuántos procesos hay y cuantas soluciones iniciales existen.

```c
void calculate_workload_depth()
{
    // Calcular la cantidad de trabajo en funcion a la cantidad de columnas.
    // En otras palabras, profundidad del arbol a pasar.
    int workload;
    if (N >= P * 1.5) // Para que no se den casos por ejemplo que N = 10 y tengo P=9, lo que haria que uno solo ejecute 2 veces y el resto nada.
    {
        depth_col = 1;
        workload = N;
    }
    else if ((N - 1) * (N - 2) > P) // Formula real para todo N > 4
    {
        depth_col = 2;
        workload = (N - 1) * (N - 2);
    }
    else
    {
        depth_col = 2;
        workload = (N - 1) * (N - 2);
        int aux;
        // Realizar un analisis para las siguientes columnas, de forma que pueda ser escalable.
        while (workload < P && depth_col <= N)
        {
            aux = 0;
            depth_col++;
            get_queens(0, queens, &aux, (depth_col - 1));
            // printf("C=%d y work=%d\n", depth_col, aux);
            if (aux <= workload)
            {
                // No tiene sentido seguir iterando, porque la anterior tenia mas trabajo.
                depth_col--;
                break;
            }
            workload = aux;
        }
    }
    // printf("Quedo en C=%d y work=%d\n", depth_col, workload);
}
```

La relación entre P y N determinará cuántas columnas se analizarán para las soluciones iniciales a distribuir.

Una vez determinado cuánto trabajo dar a cada proceso, master priorizará entregar trabajo a cada slave (verificado en Iprobe) y mientras no tenga esclavos esperando trabajo, realizará trabajo él mismo. Cuando ya no queda trabajo por entregar, indica al esclavo que termine su ejecución.

```
void master()
{
    unsigned int total_solutions = 0;

    double timetick = dwalltime();

    if (P == 1) // Solo se encuentra el master ejecutando.
    {
        get_queens(0, queens, &total_solutions, N - 1);
        printf("N = %d\tP = %d\t# = %u\tTiempo = %.4f\n", N, P ,total_solutions, dwalltime() - timetick);
        if (valores_correctos[N] != total_solutions)
            printf("\nX\nX\nSolucion incorrecta.\nX\nX\n");
        return;
    }

    calculate_workload_depth();

    // Inicializo el vector que contendra la parcion de trabajo.
    int *q_next_work = malloc(sizeof(int) * depth_col);
    q_next_work[0] = 0;

    // Flag que indica si existen solicitudes de trabajo sin leer.
    int unread_msg;

    // Bucle que itera siempre y cuando exista trabajo.
    while (get_next_work(q_next_work, depth_col - 1))
    {
        // Comprobar si es que existen solicitudes pendientes.
        MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &unread_msg, &status);
        if (!unread_msg)
        {
            // Como no hay slaves pidiendo trabajo, el master trabajo para no estar ocioso.
            // Copiar la parte inicial del trabajo para no alterar el historial de trabajo.
            memcpy(queens, q_next_work, sizeof(int) * depth_col);
            get_queens_master(depth_col, queens, &local_solutions, q_next_work, N - 1);
        }
        else
        {
            // Recibir solicitud y guardar en status el rank del slave correspondiente.
            MPI_Recv(0, 0, MPI_INT, MPI_ANY_SOURCE, WORK_TAG, MPI_COMM_WORLD, &status);
            // Enviar el trabajo.
            MPI_Send(q_next_work, depth_col, MPI_INT, status.MPI_SOURCE, WORK_TAG, MPI_COMM_WORLD);
        }
    }

    // Al terminar todo el trabajo, se le informa a todos los slaves que concluyan con FINISH_TAG.
    for (int r = 1; r < P; r++)
        MPI_Send(0, 0, MPI_INT, r, FINISH_TAG, MPI_COMM_WORLD);

    // Obtener el valor final de soluciones como suma de los valores locales de cada proceso.
    MPI_Reduce(&local_solutions, &total_solutions, 1, MPI_UNSIGNED, MPI_SUM, MASTER_RANK, MPI_COMM_WORLD);

    printf("N = %d\tP = %d\t# = %u\tTiempo = %.4f\n", N, P ,total_solutions, dwalltime() - timetick);
    if (valores_correctos[N] != total_solutions)
        printf("\nX\nX\nSolucion incorrecta.\nX\nX\n");

    free(q_next_work);
```

El esclavo únicamente ejecuta el trabajo analizando soluciones a partir de las posiciones que se le pasan, y una vez terminadas de analizar, solicita nuevas hasta que ya no haya trabajo pendiente.

```c
/**
 * Funcion que indica el funcionamiento del proceso slave.
 */
void slave()
{
    // Bucle infinito que concluye cuando recibo el mensaje correspondiente, con FINISH_TAG
    while (1)
    {
        // Enviar solicitud de trabajo al master
        MPI_Send(0, 0, MPI_INT, MASTER_RANK, WORK_TAG, MPI_COMM_WORLD);
        // Recibir respuesta
        MPI_Recv(queens, N, MPI_INT, MASTER_RANK, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        // Corroborar si es que hay que finalizar
        if (status.MPI_TAG == FINISH_TAG)
            break;
        // Obtener la cantidad de reinas envidas.
        MPI_Get_count(&status, MPI_INT, &depth_col);
        // Realizo el trabajo en base a la porcion inicial recibida.
        get_queens(depth_col, queens, &local_solutions, N - 1);
    }
    // Enviar el valor final
    MPI_Reduce(&local_solutions, &local_solutions, 1, MPI_UNSIGNED, MPI_SUM, MASTER_RANK, MPI_COMM_WORLD);
}
```

Cuando el master determina que no hay más trabajo, reduce las soluciones locales de todos los esclavos, sumándose, y obteniendo así el valor final.

Además corrobora que el valor obtenido sea el correcto según un **look-up table** precargado con las soluciones correctas para todo N.

Resultados

En la siguiente tabla se listan los resultados de las ejecuciones:

N	Tiempo secuencial	Tiempo #P = 4	SpeedUp	Eficiencia	Tiempo #P = 8	SpeedUp	Eficiencia
5	0,00001	0,0110	0,0009	0,0002	0,0115	0,0009	0,0001
6	0,0001	0,0110	0,0091	0,0023	0,0145	0,0069	0,0009
7	0,0002	0,0109	0,0183	0,0046	0,0115	0,0174	0,0022
8	0,0009	0,0110	0,0818	0,0205	0,0115	0,0783	0,0098
9	0,0042	0,0106	0,3962	0,0991	0,0186	0,2258	0,0282
10	0,0180	0,0208	0,8654	0,2163	0,0163	1,1043	0,1380
11	0,0749	0,0367	2,0409	0,5102	0,0214	3,5000	0,4375
12	0,3522	0,1463	2,4074	0,6018	0,0780	4,5154	0,5644
13	2,0292	0,7509	2,7024	0,6756	0,3573	5,6793	0,7099
14	13,4724	4,7806	2,8181	0,7045	2,1591	6,2398	0,7800
15	93,5781	31,7220	2,9499	0,7375	16,9807	5,5109	0,6889

Conclusiones

Se puede observar que existe un overhead de comunicaciones importante para los primeros valores de N haciendo que el speedup sea menor a 1, indicando ineficiencia en este rango de valores para la solución paralela en memoria distribuida. A medida que N aumenta se percibe una mejora de Speedup al distribuir mejor el trabajo.