# Sistemas Distribuidos y Paralelismo

*Informe trabajo final*

Alumno: Perez, Federico Sebastián

## *Objetivos del Práctico*

A través de esta actividad se pretende que el alumno comprenda la problemática asociada al diseño e implementación de buenos sistemas paralelos, debiendo integrar los conceptos vistos en la materia, justificando las decisiones de diseño e implementación y los resultados obtenidos.

## *Especificación del problema*

*Simulación de la permutación de la opinión política de una población.*

Las casillas de una cuadrícula rectangular en un momento dado se encuentran coloreadas en uno de cuatro colores posibles (Azul, Rojo, Verde y Blanco). Cada color distinto al blanco (sin opinión política formada) refleja la opinión política de una persona residente en una casilla. Figura 1.

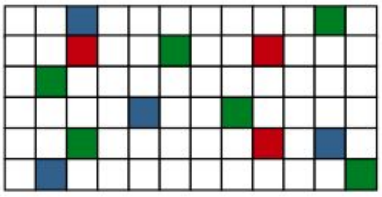


Figura 1: Grilla poblacional
Cada celda representa una persona y su opinión política

El objetivo del presente práctico es reproducir el cambio a lo largo del tiempo de la opinión política de una población dada. Para ello, un reloj virtual de cómputo conduce la evolución del modelo planteado cambiando el estado de cada una de las celdas que componen la grilla en cada paso de tiempo. Es decir, que a cada señal de reloj, cada una de las celdas de la cuadrícula (en paralelo) mantendrá o cambiará su estado a uno nuevo, dependiendo de su propia opinión y de las opiniones de sus ocho vecinos adyacentes (vecindario de moore) de acuerdo a las siguientes reglas:

- Una celda (x) en el paso de tiempo T que denota la presencia de una posición política mantendrá o cambiará su estado en el próximo paso de tiempo T+1 de acuerdo con una distribución de probabilidades que describe la cantidad personas simpatizantes de cada facción política en el vecindario de la celda. A modo de ejemplo podríamos representar esta distribución de probabilidades de la siguiente manera. Figura 2.


Figura 2: Estado del vecindario de una celda (X)

De esta forma podemos establecer que existe una probabilidad de 2/9 de que la celda central cambie su opinión política a azul debido a que existen dos personas en su vecindario con esta opinión. Además, del mismo ejemplo podemos deducir que existe una probabilidad de 3/9 de que la celda central mantenga su opinión política actual, debido a que existen 3 celdas (2 vecinos y la celda central) con dicha opinión.

De acuerdo con lo descrito, es posible entonces confeccionar un tabla de frecuencias que represente la distribución de probabilidades que se debe tener en cuenta al momento de actualizar el estado de la celda central. De esta manera, si lanzamos un número aleatorio uniforme entre 0 y 1 podremos determinar el próximo estado de la celda para el tiempo de simulación T+1 de acuerdo a la frecuencia acumulada de probabilidades. Tabla 1.

| Opinión Política | Frecuencia de opinión | Frecuencia Acumulada |
|---|---|---|
| Azul | 2/9 | 2/9 |
| Rojo | 3/9 | 5/9 |
| Verde | 1/9 | 6/9 |
| Blanco | 3/9 | 1 |

Tabla 1: Distribución de probabilidades de la figura 2.

- Una celda central en cuyo vecindario existan la misma cantidad de celdas de con diferentes facciones políticas quedará en el próximo paso de tiempo sin opinión política formada.

## *Consigna*

- Realizar un programa secuencial en C el cual sea capaz de simular la evolución de la opinión política de una población a lo largo del tiempo.

- Implementar un algoritmo paralelo híbrido utilizando pasaje de mensajes (MPI) y Threads (OpenMP).

- Realizar un análisis del desempeño de la aplicación, para ello considere el Speed Up y la Eficiencia como medidas de performance. Determine aproximadamente el máximo tamaño de grilla que es posible utilizar de acuerdo a las capacidades de la maquina paralela y realice al menos 4 pruebas diferentes disminuyendo progresivamente la cantidad de elementos en la grilla (cantidad de celdas).

- Para cada prueba de cada implementación realice un mínimo de 30 corridas y obtenga su promedio para efectuar el análisis correspondiente.

- Realice un informe sobre la implementación y el análisis comparativo del desempeño del algoritmo desarrollado.

## *Resolución: Consideraciones generales y convenciones usadas.*

- Se realizaron 3 versiones del programa, todas utilizan el mismo algoritmo y varían respecto al paralelismo con el que se quiso trabajar en cada una de ellas, a saber: código secuencial, código con paralelismo a nivel threads 'openMP' y código con paralelismo Híbrido, usando threads y pasaje de mensajes entre procesos.

- Para todos los algoritmos (Programas en c) se trabajó con matrices cuadradas de tamaño nxn, donde n es múltiplo de 2.

- En las mediciones de tiempos se consideró únicamente el tiempo que lleva el procesamiento 'neto' de la simulación, dejando de lado los aspectos de inicialización y tratamiento entrada/salida del programa.

- Para las simulaciones, se consideraron los tamaños de matrices nxn para n: 1000, 2500 y 5000.

- Para el código secuencial y el código openMP se ejecutaron 5 corridas de cada variación de tamaño, utilizando particularmente en openMP (2, 4 y 8 threads). Mientras que en el caso del código MPI se realizó una corrida con n=1000 para poder comparar con secuencial y openMP.

- Para el código MPI se utilizaron 4 máquinas (4 procesos) concurrentes.

- Cada una de las simulaciones itera por 4320 pasos de tiempo, este número no es aleatorio sino que quiere representar cómo varía la ideología política de cada uno de los sujetos durante 6 meses, observando intervalos de una hora.

- Los programas cuentan con un 'flag' que permite la visualización de la matriz y los cambios en la misma paso a paso. El mismo fue muy útil a la hora de pulir el algoritmo en sí.

## *Resolución: Algoritmo secuencial*

El funcionamiento del algoritmo a grandes es el siguiente:

- Crea e inicializa una matriz de manera completamente aleatoria, asignándole a cada elemento de la matriz una representación numérica de una de las ideologías políticas posibles, a saber: WHITE:0, RED:1, GREEN:2, BLUE:3.

- Entra en la iteración principal, fijada en 4320 pasos para este trabajo según lo visto en las consideraciones generales y en cada uno de los pasos calcula la variación de opiniones políticas en la matriz:

    - Según la especificación del problema, este cálculo se realiza iterando por cada uno de los elementos de la matriz y calculando el posible cambio de ideología política para cada uno de ellos teniendo en cuenta sus vecinos según la tabla de distribución de probabilidades.

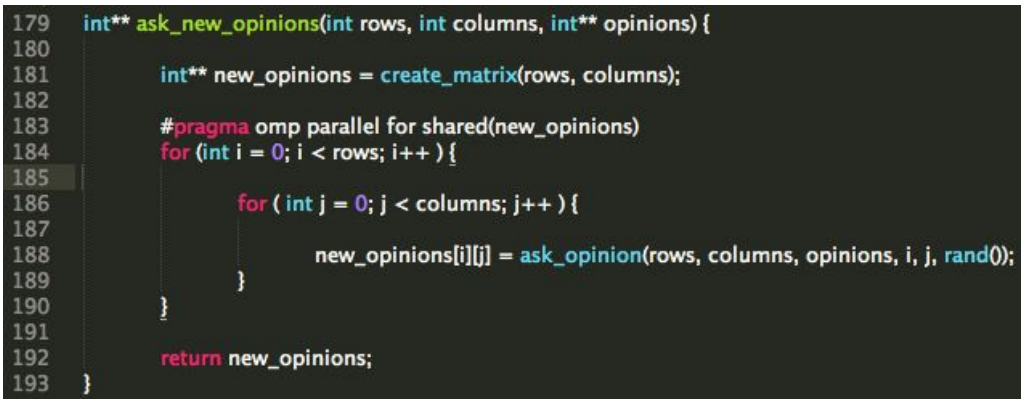
    - En el cálculo del cambio de ideología para un sujeto se tienen en cuenta los empates entre los vecinos, y en el caso de ocurrir se infiere una opinión política neutral (WHITE) con un 25% de probabilidades.

- Por último muestra la cantidad de sujetos con cada una de las opiniones políticas posibles, resultado de la simulación, así como el tiempo insumido en la misma (medido en milisegundos).

## *Resolución: Algoritmo openMP*

En esta variante del algoritmo se incorporó paralelismo a nivel de datos de entrada implementado a través de threads particularmente en el procesamiento de los cambios de opinión política de cada sujeto. Se paralelizó el procesamiento de la matriz a nivel filas, donde cada thread involucrado en la simulación es el encargado de calcular los posibles cambios en todos los elementos de una fila de la matriz.

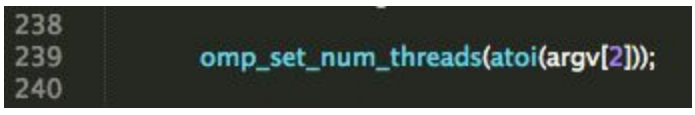
Para la paralelización se utilizaron las directivas `parallel` y `for` combinadas en el bucle externo de la iteración por la matriz:



```
179  int** ask_new_opinions(int rows, int columns, int** opinions) {
180
181          int** new_opinions = create_matrix(rows, columns);
182
183          #pragma omp parallel for shared(new_opinions)
184          for (int i = 0; i < rows; i++ ) {
185
186                  for ( int j = 0; j < columns; j++ ) {
187
188                          new_opinions[i][j] = ask_opinion(rows, columns, opinions, i, j, rand());
189                  }
190          }
191
192          return new_opinions;
193  }
```

Se realizaron distintas pruebas de la paralelización en el bucle interno de la iteración de la matriz (paralelismo a nivel de columnas) y los resultados acusaron una peor performance, es por eso que se decidió paralelizar a nivel columnas.

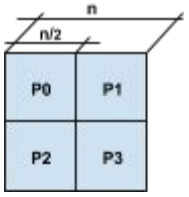
Vale destacar que se parametrizó la cantidad de threads con las que se desea correr el programa, de esta manera fue posible realizar una comparación del mismo programa, utilizando distinta cantidad de threads.



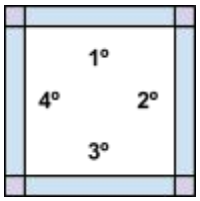
```
238
239          omp_set_num_threads(atoi(argv[2]));
240
```

## *Resolución: Algoritmo MPI*

Esta versión del algoritmo tiene algunas particularidades extra, ya que se incorpora el paralelismo a nivel de procesos con pasaje de mensajes, se utilizaron 4 procesos por lo que el problema original se divide en 4 asignándole a cada proceso su porción de matriz:




De esta manera cada proceso necesita calcular el cambio de ideologías solo en su porción de la matriz original, ahora bien, si prestamos atención a los extremos de cada matriz y recordamos que el cálculo de cambio de ideología sobre un sujeto se realizaba en función de la ideología de los vecinos, es facil notar que ahora será necesario que los procesos se comuniquen entre ellos para, en cada iteración, pasarse los "halos" y las esquinas. Con fines de implementación, se le asignó un orden a los halos laterales, superior e inferior como sigue:




Llamamos 'myhalo1' y 'myhalo2' a los halos calculados por cada proceso en cada tiempo T, los cuales deben ser enviados a los procesos vecinos, por otro lado llamamos 'halo1' y 'halo2' a los halos que cada proceso recibe de sus vecinos y por último llamamos 'mycorner' y 'corner' a la esquina que cada proceso debe enviar y recibir respectivamente. De todo lo anterior, tenemos:

| | **envía myhalo1 a...** | **envía myhalo2 a...** | **recive halo1 de...** | **recive halo2 de...** | **envia mycorner a...** | **recive corner de...** |
|---|---|---|---|---|---|---|
| **P0** | P1 | to P2 | from P1 | from P2 | to 3 | from 3 |
| **P1** | P3 | to P0 | from P3 | from P0 | to 2 | from 2 |
| **P2** | P0 | to P3 | from P0 | from P3 | to 1 | from 1 |
| **P3** | to P1 | to P2 | from P1 | from P2 | to 0 | from 0 |

## *Resolución: Ejecución*

Para ejecutar los programas es necesario tener instalado openMP y MPI, si se corre en un ambiente linux, distribución Ubuntu, puede hacerse los siguiente:

```
apt-get install gcc-4.9
apt-get install libopenmpi-dev openmpi-bin libhdf5-openmpi-dev
apt-get install mpich2 libmpich2-dev libhdf5-mpich2-dev
```

Luego, compilar los fuentes: seq.c, openmp.c y mpi.c:

```
gcc-4.9 -std=c99 -O3 -funroll-loops -ffast-math -o seq seq.c
gcc-4.9 -fopenmp -std=c99 -O4 -funroll-loops -ffast-math -o openmp openmp.c
mpicc -fopenmp -std=c99 -O4 -funroll-loops -ffast-math -o mpi mpi.c
```

*Notar que se aplicaron flags para hacer uso de optimizaciones del compilador (gcc) que mejoran los tiempos de ejecución. El resultado de la aplicación de cada uno de ellos puede verse aquí: https://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Optimize-Options.html*

Y por último para correr:

```
./seq n
./openmp n threads_number
mpiexec -np 4 ./mpi n threads_number
```

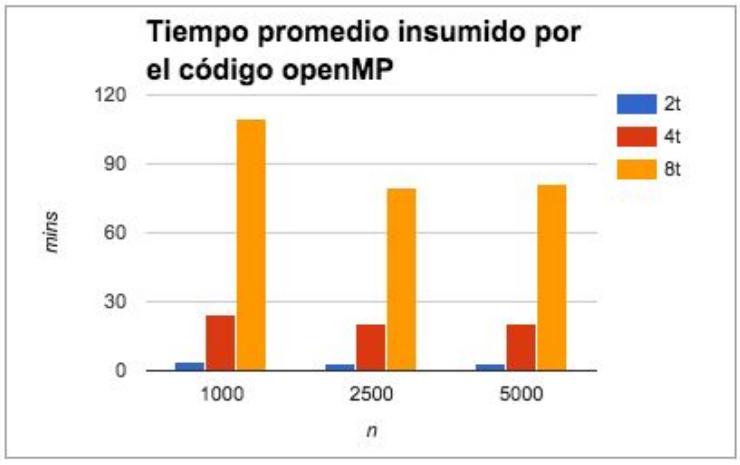
Para correr un lote de 5:

```
for i in {1..10};do ./seq n; done;
for i in {1..10};do ./openmp n threads_number; done;
for i in {1..10};do ./mpi n threads_number; done;
```

*Resolución: Ejecución Secuencial vs openMP*

| Versión | Código secuencial | | |
|---|---|---|---|
| Threads | | | |
| Matriz nxn | n=1000 | n=2500 | n=5000 |
| Corridas (millis) | 428366 | 2690767 | 10866387 |
| | 423786 | 2727378 | 10738702 |
| | 431696 | 2692570 | 10972156 |
| | 427312 | 2705879 | 10812551 |
| | 430671 | 2637241 | 10942139 |
| Promedio (mins) | 7.1394 | 44.8461 | 181.1065 |


| Versión | Código openmp | | | Código openmp | | | Código openmp | | |
|---|---|---|---|---|---|---|---|---|---|
| Threads | 2 | | | 4 | | | 8 | | |
| Matriz nxn | n=1000 | n=2500 | n=5000 | n=1000 | n=2500 | n=5000 | n=1000 | n=2500 | n=5000 |
| Corridas (millis) | 241412 | 1472644 | 6573212 | 193000 | 1218615 | 4792263 | 197110 | 1231966 | 4875550 |
| | 269091 | 1467214 | 7181587 | 193597 | 1130809 | 4854090 | 199772 | 1230653 | 4852591 |
| | 232895 | 1471521 | 5870566 | 178261 | 1242439 | 4383132 | 196526 | 1229555 | 4908904 |
| | 231125 | 1474986 | 7070431 | 201078 | 1248984 | 4965914 | 196475 | 1228423 | 4823501 |
| | 232537 | 1476853 | 6170262 | 199065 | 1252228 | 4965914 | 195665 | 1239231 | 4917202 |
| Promedio (mins) | 4.0235 | 24.5441 | 109.5535 | 3.2167 | 20.3103 | 79.871 | 3.2852 | 20.5328 | 81.2592 |



*Resolución: Ejecución Secuencial vs MPI*

| Versión | Código secuencial |
|---|---|
| Threads | |
| Matriz nxn | n=1000 |
| Corridas (millis) | 396320 |
| | 396346 |
| | 398626 |
| | 395645 |
| | 397130 |
| Promedio (mins) | 6.6136 |


**Corrida MPI: 15.4686 mins**

## *Resolución: Cálculo de SpeedUp y Eficiencia*

A continuación se muestra el cálculo de SpeedUp y Eficiencia para P = 4 que se logró con cada uno de las versiones de openMP y la de MPI. Para dicho cálculo se utilizan las siguientes fórmulas:

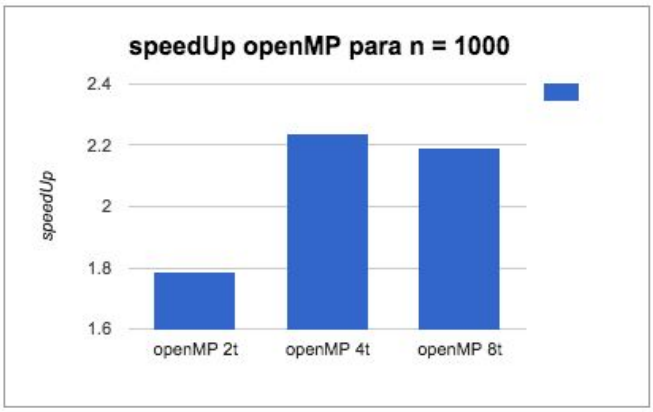
- SpeedUp(A,P) = T-bestSeq(A) / T-par(A,P)
- E(A,P) = SpeedUp(A,P) / P

Sabemos que en teoría el SpeedUp puede ser como máximo P y que la eficiencia como máximo 1. Por lo que nuestros valores deberán encontrarse dentro de esos límites.

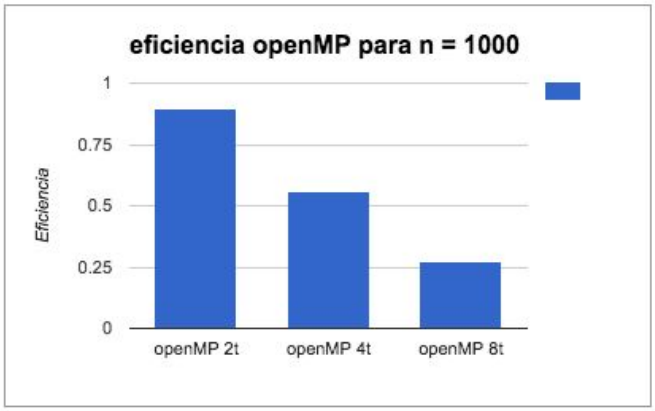
Para las pruebas secuencial y openMP se utilizó una máquina Apple con 16 GB de RAM y un procesador intel i5 de 4 cores. Mientras que para las pruebas MPI se utilizaron 4 máquinas con 8 GB de RAM y un procesador AMD 6800k.

**<u>Análisis del código openMP respecto al secuencial para n=1000</u>**

| speedUp openMP para n = 1000 | |
|---|---|
| openMP 2t | 1.788212682 |
| openMP 4t | 2.236766839 |
| openMP 8t | 2.19012734 |




| eficiencia openMP para n = 1000 | |
|---|---|
| openMP 2t | 0.894106341 |
| openMP 4t | 0.5591917098 |
| openMP 8t | 0.2737659175 |



Podemos observar el siguiente comportamiento: Conforme la cantidad de procesadores va creciendo la eficiencia cae debido a que el speedup se mantiene medianamente estable. El mejor speedUp se obtiene con 4 threads mientras que la mejor eficiencia se obtiene con 2 threads.

**Análisis del código MPI respecto al secuencial para n=1000**

T-bestSeq(A) = Max (396320, 396346, 398626, 395645, 397130) = 398626

T-par(A,P) = 928116

SpeedUp(A,P) = T-bestSeq(A) / T-par(A,P) = 398626 / 928116 = **0.4295**

E(A,P) = SpeedUp(A,P) / P = 0.4295 / 4 = **0.1073**

Por otro lado la versión de MPI se ve penalizada por el alto grado de pasaje de mensajes. La versión MPI tarda más que la sequencial. Por falta de tiempo no corrimos los algoritmos para n mayores.

## *Conclusiones*

Este trabajo me resultó muy interesante, distinto a la mayoría de los trabajos prácticos que me han tocado hacer en la Universidad, ver los conceptos vistos en la materia aplicados a un caso práctico que puede se punto de partida para una investigación al respecto, me permitió abrir la cabeza en el sentido de simulaciones de propagación y su implementación.

A nivel técnico, tuve que aplicar una gran variedad de conceptos desde buscar optimizaciones del compilador de C para mejorar la performance del programa hasta trabajar con claves ssh para distribuir el trabajo en la red. Vale destacar que conocimos 2 librerías para paralelismo.