Escalabilidad del Modelo de Potts en entornos distribuidos con OpenMP y MPI

Javier Nicolás Uranga

Directores: Nicolás Wolovick, Javier Blanco

Presentado como Trabajo Final Integrador de la carrera de posgrado
Especialización en Servicios y Sistemas Distribuidos

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

Marzo de 2012

Clasificación según ACM (*Association for Computing Machinery*):

C.1.2 Multiple Data Stream Architectures (Multiprocessors)
  Parallel processors
C.2.4 Distributed Systems
  Distributed applications
D.1.3 Concurrent Programming
  Distributed programming
  Parallel programming
D.2.3 Coding Tools and Techniques
  Structured programming
D.2.8 Metrics
  Performance measures
D.3.2 Language Classifications
  Concurrent, distributed, and parallel languages
E.1 Data Structures
  Distributed data structures
F.1.2 Modes of Computation
  Parallelism and concurrency
  Probabilistic computation
G.3 Probability and Statistics
  Markov processes
  Probabilistic algorithms (including Monte Carlo)
  Random number generation
I.6.8 Types of Simulation
  Monte Carlo
J.2 Physical Sciences and Engineering
  Physics

Clasificacion según PACS (The Physics and Astronomy Classification Scheme®):

64.60.De Statistical mechanics of model systems (Ising model, Potts model, field-theory models, Monte Carlo techniques, etc.)

Resumen de catálogo:

El objetivo principal del trabajo es obtener una implementación correcta en los paradigmas memoria compartida y pasaje de mensajes sobre multicore y clusters para la simulación del modelo de Potts de q-estados bidimensional en la transición de fase ferromagnética-paramagnética variando la temperatura y buscando la convergencia hacia el orden del sistema en un estado de mínima energía, utilizando OpenMP y MPI y buscando escalabilidad perfecta en el número de cores y en el número de nodos.

Palabras claves:

Mecánica Estadística, Modelo de Potts, OpenMP, MPI, Montecarlo, Metropolis, Computación Híbrida, Computación Paralela, Clusters.

Página en blanco dejada intencionalmente.

Indice General

Resumen....6
Abstract....7
Introducción....8
  Hardware y software de base utilizado....8
  Ilustraciones y notación utilizada....9
  Estados de la simulación....10
Capítulo 1: El Modelo Potts 11
  El concepto de spin en el contexto de la simulación de Potts....11
  Aplicaciones del modelo de Potts....12
  El método de Montecarlo con dinámica de Metropolis 14
  Protocolo de simulación....14
  El lattice....14
Capítulo 2: El código secuencial....16
  Estructura del código secuencial....16
  Generador de números aleatorios del código secuencial....17
  Profiling del código secuencial....18
  Parámetros de medición....18
  Estructura general de las mediciones automáticas en el cluster....19
  Compilación y salida típica del código secuencial: patrón de medida....20
Capítulo 3: Paralelización en OpenMP....21
  Estructura del código OpenMP....21
  Ejecución de la versión OpenMP....26
  Resultados de performance en OpenMP....27
Capítulo 4: Paralelización en MPI....28
  Topologías n×m y 1×n....28
  La técnica ghost-cells....29
  El lattice en MPI....31
  Cálculo de Nodos Vecinos....32
  Nodo Root y nodos Workers....32
  Estructura del código MPI....33
  Generador de números aleatorios distribuido....36
  Fase de Equilibrio....37
  Compilación y ejecución MPI....37
  Resultados de performance en MPI....38
Capítulo 5: Paralelización Híbrida: OpenMP+MPI....39
  Compilación y ejecución híbrida....39
  Resultados de performance y problemas detectados en la versión HYBR....40
  Prueba de una alternativa: utilización del tablero-simple....41
  Comparación en eficiencia de la versión OMP: SMPL vs. WB....42
  Comparación en eficiencia de la versión MPI: SMPL vs. WB....42
  Comparación en eficiencia de la versión HYBR: SMPL vs. WB....43
  Comparación en eficiencia de la versión MPI-SMPL vs HYBR-SMPL....43
Conclusiones....44
Trabajo futuro....45
Agradecimientos....46
Referencias....47

Resumen

El Objetivo del trabajo es estudiar los modelos más populares sobre paralelización de algoritmos de cómputo científico en multicore y clusters, resolviendo un problema de grilla y buscando escalabilidad perfecta en el número de cores y el número de nodos.

Se tomará como base un problema Mecánica Estadística, ya modelado por el grupo GTMC de FaMAF [3], originalmente codificado en C [7] y posteriormente llevado a CUDA [8]. Se trata de un algoritmo Monte Carlo que utiliza la dinámica de Metropolis para simular el modelo de Potts.

La simulación estudia la transición de fase ferromagnética-paramagnética variando la temperatura y buscando la convergencia hacia el orden del sistema en un estado de mínima energía, utilizando una grilla de espines de dimensión L×L, para q-estados en el modelo de Potts.

Las tecnologías de paralelización utilizadas fueron: OpenMP [1] y MPI [2]. Para tal fin se crearon seis versiones de código paralelo: dos de MPI-puro, dos OpenMP-puro y dos versiones Híbridas que combinan las características de OpenMP y MPI. En adelante serán referidas por los prefijos MPI, OMP e HYBR respectivamente.

El objetivo principal es obtener una implementación correcta en los dos paradigmas (memoria compartida y pasaje de mensajes) y solucionar los problemas de concurrencia que puedan surgir (condiciones de carrera, deadlocks). El objetivo secundario es estudiar los problemas de escalabilidad y proponer las soluciones correspondientes buscando alcanzar la escalabilidad ideal.

Abstract

The aim of this work is to study the most popular models for scientific computing algorithms in multicore clusters.

The starting point is a problem Statistical Mechanics, already modeled by GTMC group at FaMAF [3], originally coded in C [7] and later moved to CUDA [8]. It is a Monte Carlo algorithm which uses the Metropolis dynamic to simulate the Potts model.

The simulation studies the ferromagnetic-paramagnetic phase transition, varying temperature and seeking the convergence towards an ordered state of the system, which is reached at the minimum energy state, using a grid of size $L \times L$ spins for q-states in the Potts model.

The parallelization technologies used are: OpenMP [1] and MPI [2]. Six versions of parallel code are generated: two pure OpenMP, two pure MPI and two Hybrids which combines the features of OpenMP and MPI.

The main objective is to obtain a correct implementation in the two paradigms (shared memory and message passing) and solve concurrency problems that could arise (race conditions, deadlocks, etc). The secondary objective is to study scalability issues that arise when we look for solutions to achieve ideal scalability.

Keywords:
Statistical Mechanics, The Potts Model, OpenMP, MPI, Montecarlo Simulation, Metropolis, Hybrid Programming, Parallel Computing, Clusters.

Introducción

Este trabajo apunta a encontrar la mejor forma de paralelización posible buscando la escalabilidad perfecta sin innovar en los aspecos algorítmicos que corresponden a la física del problema (Modelo de Potts) sino más bien reestructurando la computación de los mismos haciéndolos distribuidos. Así entonces lo que se modificará es el “como lo hace” y no “que hace” el código secuencial original. El enfoque será siempre distribuido, ya sea internamente en un host distribuyendo el trabajo sobre los cores disponibles o en un cluster distribuyendo a un nivel más alto, entre los nodos disponibles.

Una vez logradas las implementaciones paralelas, serán comparadas entre sí y contra la versión secuencial utilizando los conceptos relacionados de *speedup* (S_p) y eficiencia (E_p), definidos por las siguientes fórmulas:

$$S_p = \frac{T_1}{T_p} \quad E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

Donde p indica el número de cores, T1 es el tiempo de ejecución del código secuencial (1 core) y Tp es el tiempo de ejecución del código paralelo con p cores.

Hardware y software de base utilizado

En el presente trabajo se utilizó un cluster compuesto por 6 nodos con el sistema operativo Linux Rocks v5.1. La descipción del sistema es el siguiente:

Nodos de computación:
- 2 Procesadores Intel core 2 quad 5420,
- 16 GB de memoria Ram,
- 2 Ethernets de 1GB,
- 1 Placa infiniband QDR,
- 2 Discos Rígidos de 250GbB.

Red:
- 1 Switch Infiniband Mellanox MTS3600.

Compilador:
- gcc versión 4.1.2 20071124 (Red Hat 4.1.2-42).

Bibliotecas:
- OpenMPI versión 1.2.8,
- OpenMP versión 2.5 (incluida en gcc).

Ilustraciones y notación utilizada

Con el fin de lograr una mayor claridad en las explicaciones de los procesos paralelos y distribuidos, en el presente trabajo se optó por una solución de compromiso y se tomaron de UML [6] ciertos diagramas y artefactos no siempre utilizados de la forma en que indica el estándar. Al no tratárse de una aplicación orientada a objetos la utilización de estos diagramas en un contexto estructurado puede bien considerarse como un abuso de notación.

En los diagramas se está modelando principalmente interacciones entre funciones y entre nodos a través de la red, utilizando los siguientes esterotipos: <<Function> y <<Node>>. También se han definido estereotipos para modelar entidades de almacenamiento en memoria como por ejemplo: <<Matrix>>.

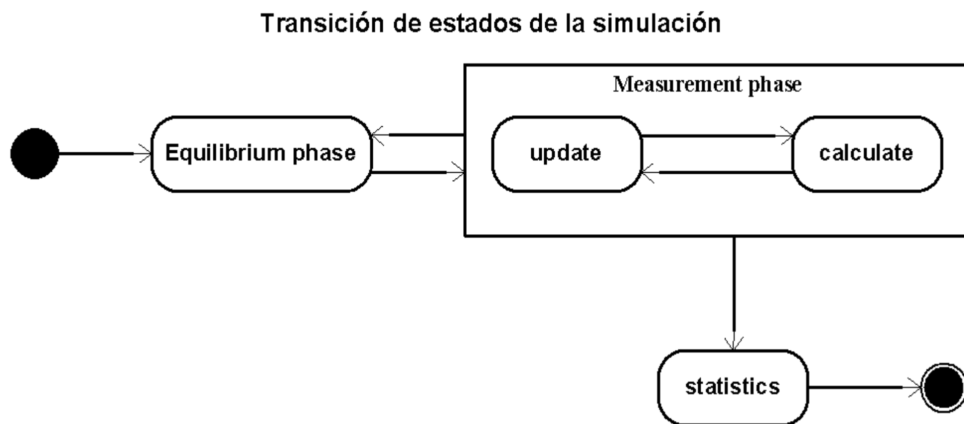
Bien sabemos que una "función" no es equivalente al concepto de "clase" de la POO, ha sido sin embargo muy conveniente la utilización de funciones en vez de clases en los diagramas de interacción de UML para mostrar la dinámica temporal de los procesos fundamentales de la simulación.

En los diagramas de interacción y esquemas se han usado dos tipos de flechas para representar la comunicación entre dos entidades de software:

1. La flecha de trazo continuo y punta cerrada representa una llamada síncrona y la flecha de trazo discontinuo con punta abierta modela una respuesta luego de una invocación previa.
2. En los esquemas de OMP (no diagramas UML) se utilizaron también flechas de trazo continuo y discontinuo para representar las acciones de escritura y de lectura de los threads sobre variables compartidas.
3. La flecha continua de punta abierta en los diagramas de interacción significan llamadas asíncronas: por ejemplo en el diagrama de interacción de *ghostCommunication* se ve este uso.
4. La flecha discontinua en los esquemas que describen topologías significa transmisión de datos en esa dirección.

Estados de la simulación

La simulación presenta conceptualmente cuatro estados muy definidos que no dependen de la implementación (SEC, OMP, MPI, HYBR). Los estados son “la fase de equilibrio” que lleva el sistema hacia el equilibrio antes de entrar al estado compuesto llamado “fase de medición”. La fase de medición está formada por el estado de “update” o proceso de actualización del *lattice* y el estado “calculate” que se refiere al proceso de cálculo de la energía y magnetización sobre el *lattice* actualizado y por último el estado “statistics” que es un estado sin retorno, en el sentido que luego del mismo la simulación sólo tiene un punto de salida.



Transición de estados para todas las implementaciones paralelas. Al centro se destaca el estado compuesto llamado: “fase de medición”.

Capítulo 1: El Modelo Potts

El modelo de Potts [5] de q-estados en Mecánica Estadística es una generalización del modelo de Ising a más de dos valores posibles para la variable de spin s, permitiéndole adoptar q estados diferentes: s = 1, 2, … , q.

El estudio original analiza el caso bidimensional en una red cuadrada de $N = L \times L$ sitios donde cada sitio i tiene asociado un spin s_i. Luego se denomina configuración del sistema a un arreglo dado $\{s_i\}$ de las variables de spins. Basta con cambiar el valor de una sola variable de spin para obtener una nueva configuración del sistema.

Las interacciones se restringen a spins adyacentes s_i , s_j que interactúan con una energía dada por: $-J_p\ \delta(s_i, s_j)$. Se estudia la versión ferromagnética donde $J_p>0$ (favorece configuraciones de spins alineados). La energía total asociada al sistema de spins se define mediante el siguiente Hamiltoniano a campo nulo (sin presencia de campos magnéticos externos):

$$\mathcal{H}_P = -J_P \sum_{<i,j>} \delta(s_i, s_j)$$

donde J_p se toma igual a la constante de Boltzmann e igual a 1 y δ_{ij}, es la delta de Kronecker: definida como $\delta_{ij}=1$ si, $s_i=s_j$ y $\delta_{ij}=0$ en caso contrario. Para una configuración dada del sistema, la suma se extiende sobre todos los pares de primeros vecinos en la red (vencinos: norte, sur, oeste, este, del sitio actual).

Se define la energía interna por spin como :

$$e = \langle H \rangle / N$$

donde <H> denota promedio sobre las configuraciones, y el parámetro de orden termodinámico (magnetización) como :

$$m = \frac{(qN_{max}/N - 1)}{q - 1}$$

donde $N_{max} = \max(N_1, N_2, \ldots, N_q)$, siendo N_i el número de spins con estado i.

La variable m toma el valor 1 cuando el sistema se ordena completamente ($N_{max}= N$), es decir, cuando alcanza uno de sus q estados fundamentales. Por otro lado, en un sistema completamente desordenado todos los estados de spin son poblados por igual y de manera aleatoria, por lo tanto $N_{max} = N/q$ y resulta $m = 0$.

El concepto de spin en el contexto de la simulación de Potts

Cada spin en el modelo representa la dirección en la que apunta el momento magnético de un átomo o también a la medida de la magnetización efectiva de un conjunto de muchos átomos en una pequeña región de un material.

Los núcleos atómicos pueden tener también un momento magnético no nulo, pero por lo general, el magnetismo que repercute a escala macroscópica es el generado por los orbitales electrónicos de los átomos (generando un momento magnético neto en alguna dirección).

El modelo de Ising (modelo padre del modelo de Potts) surge entonces para modelar una

transición de fase ferromagnética-paramagnética (magnetización neta no nula - magnetización neta nula) al aumentar la temperatura del sistema. Este modelo a cobrado en la actualidad un mayor interés en la comunidad científica y se ha comenzado a utilizar para simular procesos físicos de naturaleza muy diversa, desde crecimiento de tumores hasta comportamientos sociales, pasando por fracturas de cerámicas y comportamiento de mercados financieros. Así entonces lo que se llama "spin" en estos modelos, en la práctica puede terminar significando diversas cosas, un individuo, una célula, etc., según el problema que se esté estudiando.

En el modelo de Potts la generalización del término es todavía más exagerada, porque al permitir "q estados" posibles equivalentes, ni siquiera se está hablando de una dirección de la magnetización en el espacio, el "spin" de Potts es simplemente una variable que puede adoptar "q colores" posibles. Sus aplicaciones también son varias, como en el caso de Ising. Al variar "q" podemos pasar de tener una "transición de fase de segundo orden" a tener una "transición de primer orden", con un modelo muy simple, y eso permite estudiar varios aspectos teóricos bien generales de la Mecánica Estadística. Por ejemplo: las múltiples transiciones solido/líquido/gas son transiciones de primer orden o discontinuas debido a que involucran un cambio discontinuo en la densidad. La transición ferromagnética en metales como el hierro, en donde la magnetización crece continuamente desde cero a medida que bajamos la temperatura, es una transición de segundo orden o contínua.

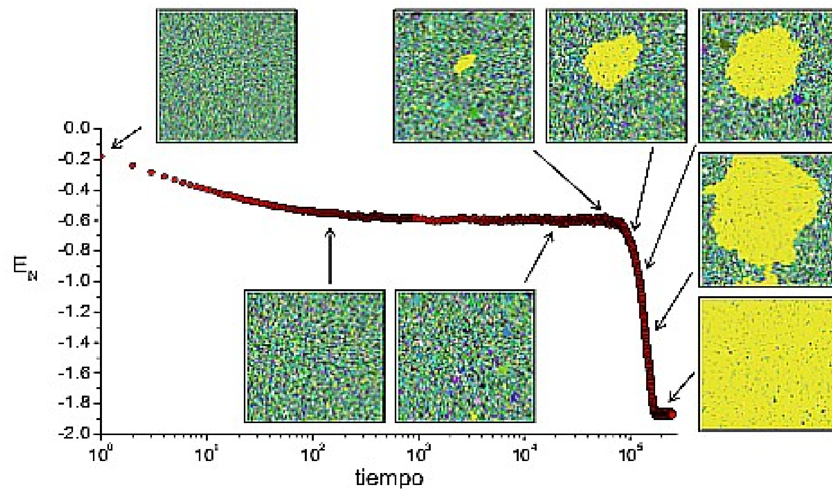
Aplicaciones del modelo de Potts

El modelo de Potts [5] es de gran interés desde el punto de vista teórico, en particular, por la posibilidad de variar con el parámetro "q" la naturaleza de su transición de fase, y por su simpleza y aptitud para el estudio de fenómenos de no-equilibrio. También es utilizado para el modelaje de aplicaciones prácticas, tales como, espumas y burbujas de jabón, segregación genética, comportamiento celular, migración tumoral, análisis de imágenes, redes neuronales o incluso en el comportamiento social y demográfico. Gracias al conocimiento de algunos resultados exactos y vasta bibliografía, constituye también una plataforma de prueba para nuevos métodos numéricos y aproximaciones en el estudio de la teoría de transiciones de fase y fenómenos críticos.

Una imporntante aplicación del modelo de Potts en Física de la Materia Condensada es para el estudio de la *nucleación* y el *coarsening* [5] que son dos procesos dinámicos mediante los cuales el sistema puede llegar al orden por debajo de su temperatura crítica.

Estudio del fenómeno de *nucleación*:

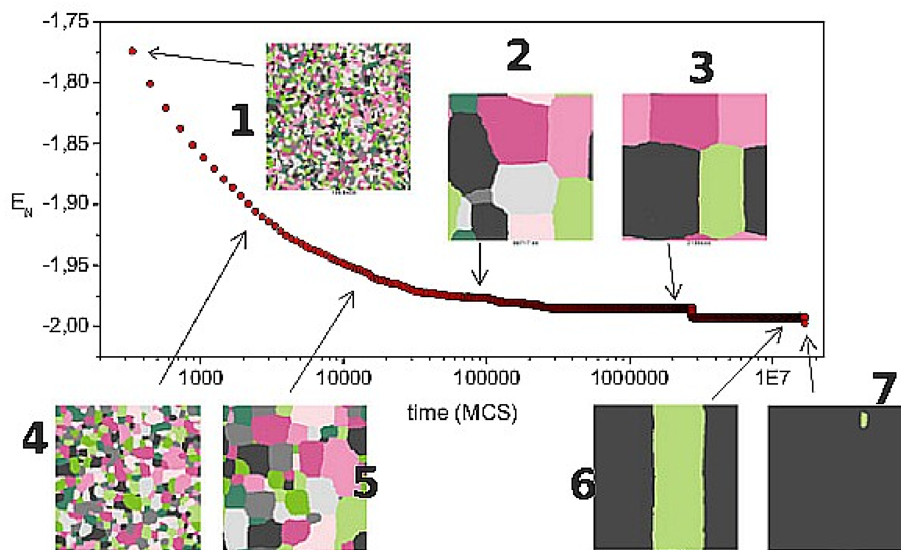
Partiendo de una temperatura (T) alta y luego de un quench (enfriamiento brusco) el sistema permance durante una ventana de tiempo en un estado metaestable en forma de plateau. Se puede observar luego la formación de un gran dominio central (podría ser más de uno pero siempre son unos pocos) que va creciendo rápidamente hasta cubrir la totalidad del sistema en la situación de equilibrio. La nucleación ocurre solamente para temperaturas por debajo pero muy cercanas a la temperatura crítica (Tc) del material: $T<\sim T_c$. La siguiente figura ilustra el proceso, donde los colores codifican los distintos valores de Q:1....9.



Nucleación. Energía por spin como función del tiempo. Después de un quench a T<~Tc. Q=24, L=200. Imagen tomada de [5] (Ref.: Fig 7.5, pag. 101. Con autorización del autor)

Estudio del *coarsening* y estados metaestables:

Luego de un quench, se hace decrecer la temperatura por debajo, pero no muy próxima a la temperatura crítica Tc del material: T<<Tc, así empieza el proceso de coarsening creciendo simultáneamente varios dominios desde tamaños muy pequeños (4 y 5). Luego dependiendo de cierta probabilidad que a su vez varía con la temperatura del quench, pueden aparecer ciertos estados metaestables donde el sistema queda bloqueado durante algún tiempo (en este punto el coarsening ya ha finalizado). En la siguiente figura se muestran cuatro instantáneas del sistema donde se observan estados metaestables a bajas energías, en forma de panal (2 y 3, estados de Lifshitz) y en forma de bandas (6). Y finalmente en este ejemplo se puede observar como el sistema relaja al estado estable (7) mediante un mecanismo de activación térmica. Los colores codifican los distintos valores de Q:1....9.



Coarsening y estados metaestables. Energía por spin como función del tiempo. Después de un quench T<<Tc. Q=9, L=300, T=0,3. Imagen tomada de [5] (Ref.: Fig 8.11, pag. 142. Con autorización del autor).

El método de Montecarlo con dinámica de Metropolis

El esquema general del algoritmo es el siguiente [5]:

(1) Se elige una celda (i,j) del *lattice* (de dimensión L×L) y se toma su *spin_old* y el estado de sus primeros vecinos: *vecindad_old*,
(2) Se calcula el *spin_new* en el rango [1,q] basado en el *spin_old* y en el sorteo de un número aleatorio,
(2) Se calcula la variación de energía: δH = H(spin_old, vecindad_old) – H(spin_new, vecindad_old) basado en el estado de las celdas vecinas: *{(i+1, j), (i-1, j), (i, j+1), (i, j-1)}*. Notar que tanto para el *spin_old* como para el nuevo spin calculado, se utilizan los mismo valores de vecindad: *vecindad_old.*
(4) Si $\delta H \leq 0$ se acepta *spin_new* como nuevo valor de estado para la celda (i,j),
(5) Si $\delta H > 0$ se sortea un número aleatorio r en (0, 1] y sólo si $r < \exp(-\delta H/k_B T)$ se aceptará *spin_new* como nuevo valor de estado para la celda (i,j), donde k_B es la constante de Boltzmann y T la temperatura.
(6) Se repite el proceso desde (2) y continua para todo i: 0 … L/2-1, j: 0 ... L-1.

Es importante resaltar que en el código *secuencial original* se define un paso de Monte Carlo al ciclo que consiste en repetir N=L×L veces el proceso recién descripto, y se adopta el paso de Monte Carlo (MCS) como la unidad de tiempo para la simulación.

Protocolo de simulación

Se comienza desde un estado inicial ordenado ($s_i = 1\ \forall i$), se fija la temperatura a $T = T_{min}$ y se corre durante t_{tran} MCS para alcanzar el equilibrio, luego se corre durante t_{max} MCS tomando una medida cada δt pasos con las que luego se realizan promedios. La última configuración del sistema se adopta como el estado inicial para la próxima temperatura $T = T_{min}+\Delta T$ y se realiza entonces el mismo proceso de equilibrado y medición. Este proceso se repite hasta que se alcanza cierta temperatura máxima T_{max}. Luego se repite el ciclo completo para muchas muestras para promediar sobre diferentes realizaciones del ruido térmico.

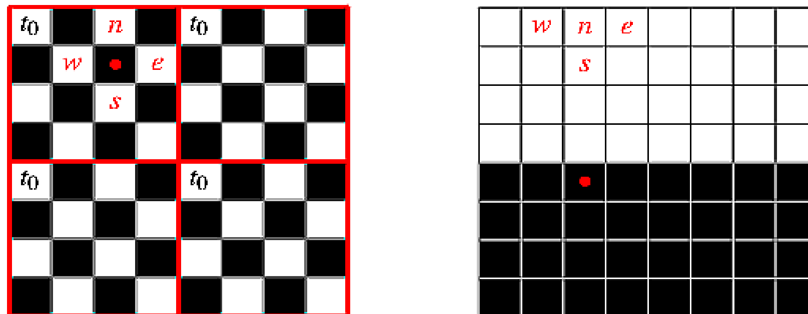
El lattice

El código *secuencial original* considera al *lattice* como un tablero de ajedrez, donde cada celda negra interactúa con sus cuatro vecinos blancos más cercanos: {norte, sur, este y oeste} y viceversa para cada celda blanca (esta vecindad también es conocida como vecindad de von Neumann). Luego el tablero de ajedrez es compactado y separado en dos mitades, las submatrices de blancas y negras de dimensiones L/2×L mediante la siguiente regla de mapeo de celdas, conocida como "*White-Black Gauss-Seidel*" [3]:

*"(i,j) del tablero de ajedrez se mapea a (((i+j) mod 2*L+i)/2, j) en el tablero compactado"*

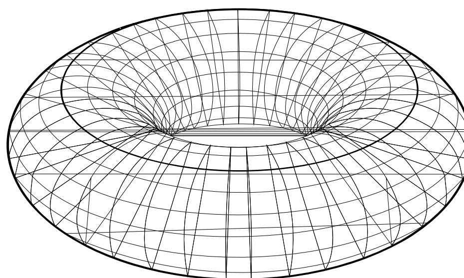
La idea detrás del código *secuencial original* era servir de base para ser luego paralelizado hacia CUDA [8]. Este tipo de compactación que empaqueta a todos los casilleros blancos y a todos los negros en ubicaciones contiguas de memoria se trasladó casi sin cambios a CUDA donde mostró ser muy beneficiosa desde el punto de vista de la performance en entornos GPGPU, mejorando la

localidad espacial y permitiendo lecturas anchas de 3 bytes consecutivos. Este trabajo toma como base aquella versión secuencial y también ha mantenido la técnica de compactación aplicándola en este caso a entornos cluster y multicore. Se muestra a continuación una ilustración de dicho mapeo: a la derecha se ve el tablero de ajedrez y a la izquierda el tablero luego del mapeo. También se muestra la ubicación inicial y final de una celda ejemplo “•” y de las celdas vecinas “n, e, s, w”.



Izquierda: tablero de ajedrez que representa al lattice, derecha: tablero mapeado. Imagen tomada de [5] (Ref.: Fig C.1, pag. 188. Con autorización del autor).

También hay que tener en cuenta que el sistema tiene condiciones de contorno periódicas, esto significa que el *lattice* anterior está unido en sus bordes norte-sur y en sus bordes este-oeste formando un toroide. Estas condiciones de contorno implican el uso de aritmética *mod* L y *mod* L/2 en el incremento de las variables i,j que controlan la posición de las celdas para no salirse de la superficie toroidal durante la actualización del estado en cada paso de Montecarlo.



Superficie toroidal sobre la que ocurre la simulación.

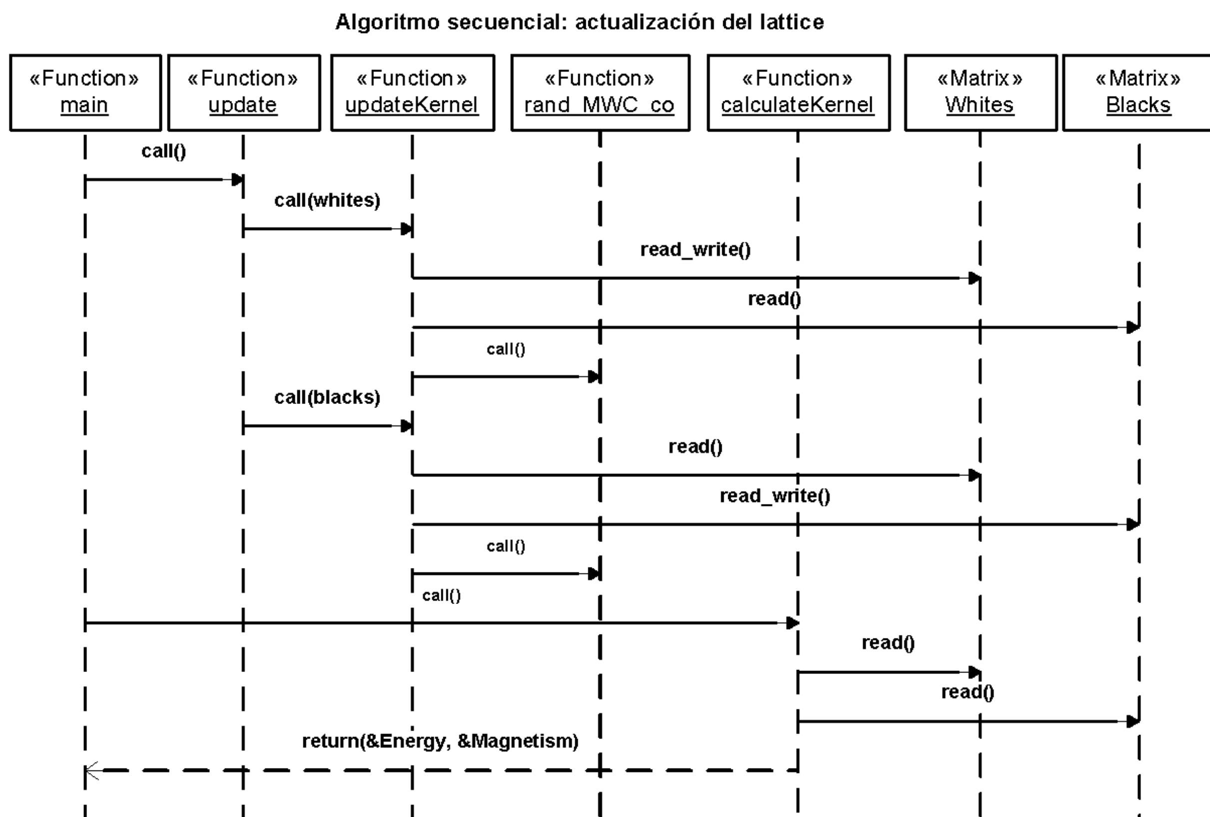
Por último se quiere hacer notar que estamos ante la definición de una dinámica de actualización probabilística, porque como se describió anteriormente, una celda puede cambiar su estado dependiendo no sólo del estado de sus vecinos sino también del sorteo de un par de números aleatorios.

Capítulo 2: El código secuencial

En este capítulo se describe a grandes rasgos el funcionamiento del algoritmo secuencial, sus parámetros y salida típica. También se investigan las funciones candidatas para ser paralelizadas más adelante en OMP y MPI.

Estructura del código secuencial

Utilizando un abuso de notación tomada del estándar UML [6] y aplicado a un caso estructurado se podría representar la parte central la simulación del algoritmo de Potts mediante el siguiente diagrama de interacción:



Proceso de actualización del lattice mediante Montecarlo-Metropolis y cálculo final de la energía y magnetización.

Para avanzar un paso de tiempo, esto es un paso de Monte Carlo, implica la actualización completa de las submatrices de blancas y negras. Esto se lleva a cabo mediante la función *update:*

```
update(const float temp, byte white[L/2][L], byte black[L/2][L])
```

la cual toma como argumentos la temperatura y las submatrices correspondientes a las celdas blancas y negras del tablero de ajedrez. Dicha función invoca a su vez a *updateKernel:*

```
updateKernel(const float temp, const unsigned int color, byte write[L/2][L], byte
read[L/2][L])
```

cuyos argumentos son la temperatura, el color de las celdas que va actualizar, la submatriz correspondientes a las celdas del color que va a actualizar y la submatriz correspondiente a las celdas del otro color.

El cálculo de las cantidades energía y magnetización se realiza luego mediante *calculateKernel:*

```
calculateKernel(byte white[L/2][L], byte black[L/2][L], unsigned int* M_max)
```

que acepta como parámetros ambas submatrices de celdas blancas y negras y una variable por referencia que utilizará para retornar el valor del magnetismo calculado. La función retorna de forma directa el valor de la energía.

Generador de números aleatorios del código secuencial

La simulación del modelo de Potts requiere una gran cantidad de números aleatorios. Cuando se actualiza el spin de cada celda se necesita un número entero aleatorio en $\{0, \ldots, q-1\}$ y posiblemente un segundo número aleatorio en el rango $[0, 1)$ para decidir sobre la aceptación o no del cambio.

En el código *secuencial original* se utilizó el generador Multiply-With-Carry (MWC) [5]. El estado del generador es de 64 bits y obtener un nuevo número pseudo-aleatorio consiste en computar $x_{n+1} = (x_n \times a + c_n) \bmod b$, donde a es el multiplicador, b la base, y c_n es el acarreo de la operación de módulo previa.

Para obtener secuencias independientes de números aleatorios el MWC usa diferentes multiplicadores, que tienen que ser “buenos” en el siguiente sentido: $(a \times b-1)$ tiene que ser un “primo seguro”, donde p es un primo seguro si tanto p como $(p-1)/2$ son primos. MWC utiliza un archivo llamado *safeprimes_base32.txt* con 150000 buenos multiplicadores.

Por el resto del trabajo y para todas las versiones paralelas (OMP, MPI e HYBR) se a mantenido el mismo generador aleatorio MWC, adaptándolo para el entorno paralelo.

Profiling del código secuencial

Mediante una comprobación utilizando la herramienta *gprof* se investigó cuales son las funciones más críticas en cuanto al tiempo que pasa la versión secuencial ejecutándolas. Para realizar esta comprobración fue necesario compilar la versión secuencial quitando todas las opciones de optimización:

```
$ make -f Makefile.shortrun
rm -f potts3-cpu *.o
gcc -c potts3-cpu.c -std=c99 -Wall -Wextra -ffast-math -march=core2 -funroll-loops
-DQ=9  -DL=1024  -DSAMPLES=1  -DTEMP_MIN=0.71f  -DTEMP_MAX=0.72f  -DDELTA_TEMP=0.005f
-DTRAN=20 -DTMAX=80 -DDELTA_T=5 -pg
gcc -o potts3-cpu  potts3-cpu.o -lm -pg

$ gprof potts3-cpu
```

Resultados:

```
Each sample counts as 0.01 seconds.
 %     cumul self  self          total
time   secs   secs  calls        s/call s/call    name
57.66  15.13  15.13  600          0.03   0.04     updateKernel
40.13  25.66  10.53  629145600    0.00   0.00     rand_MWC_co
2.13   26.22  0.56   48           0.01   0.01     calculateKernel
0.08   26.24  0.02   1            0.02   0.02     init_RNG
0.00   26.24  0.00   300          0.00   0.09     update
0.00   26.24  0.00   2            0.00   0.00     timeval_subtract
0.00   26.24  0.00   1            0.00   0.02     ConfigureRandomNumbers
0.00   26.24  0.00   1            0.00   26.22    cycle
0.00   26.24  0.00   1            0.00   26.22    sample
(...)
```

De las tres primeras funciones posicionadas en la lista, solo la primera con más del 57% del tiempo y la tercera con el 2,13% son paralelizables. Así entonces nuestras candidatas para el trabajo subsiguiente con OpenMP y MPI serán updateKernel() y calculateKernel().

Parámetros de medición

Los parámetros simulados para todas las versiones: SEC, OMP, MPI e HYBR, fueron siempre los mismos para mantener una coherencia y para que las comparaciones entre los distintos paradigmas cobren sentido. Los parámetros usados fueron:

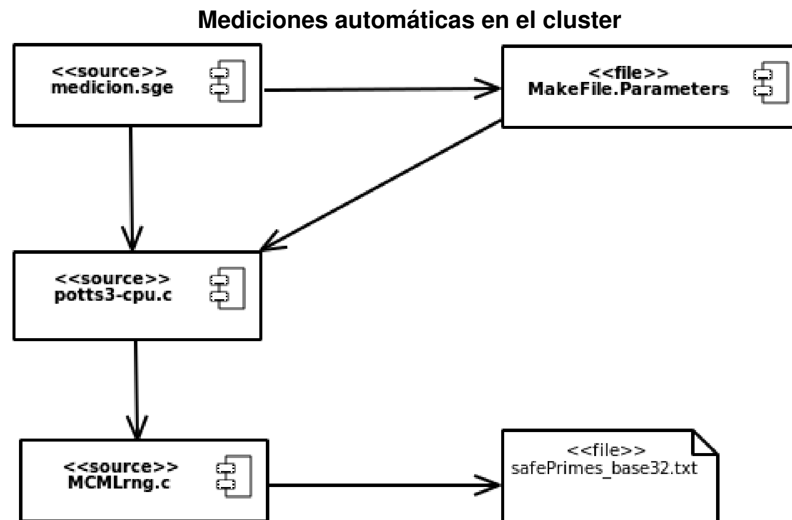
```
Q:9,
SAMPLES:1,
Temperatura mínima: 0.71,
Temperatura máxima: 0.72,
Paso de Temperatura: 0.005,
Tiempo de equilibrio: 20,
Tiempo máximo de medición: 80,
Paso de tiempo: 50
```

La cantidad de NPOINTS de la simulación fue igual a 3 (cada NPOINT corresponde a una línea en la salida final de la simulación). Las barras de desviación estándar en general no han sido dibujadas por ser demasiado pequeñas para las escalas de los gráficos (del orden de 10^{-2}). El conjunto de valores de L más conveniente para las comparaciones cruzadas entre modelos fue: `{516, 1032, 2064, 4128, 8256, 16512}`. La cantidad de muestras para cada L fue igual a 20. Es importante no confundir el número de muestras para la medición del tiempo del software con el parámetro SAMPLES que se usa internamente en la simulación, no están relacionados bajo ningún concepto.

Recordar siempre que cuando hablamos de tiempos nos estamos refiriendo a pasos de Montecarlo: 1 MCS=unidad de tiempo.

Estructura general de las mediciones automáticas en el cluster

Todas las mediciones realizadas para este trabajo, incluso para las del código secuencial fueron realizadas en un cluster Linux bajo un sistema de colas SGE (Sun Grid Engine) [9] y siguiendo siempre el siguiente esquema:



Esquema de dependencias entre los componentes del proceso de medición en el cluster.

Un script llamado *medición.sge* se envía al sistema de colas SGE para ser ejecutado. El script utiliza durante su proceso iterativo dos archivos de entrada: *MakeFile.parameters* y los fuentes en C: *potts-cpu.c*. A su vez *potts-cpu.c* incluye el módulo *MCMLrng.c* (módulo del generador de números aleatorios), y éste último utiliza el archivo de números aleatorios *safeprimes_base32.txt*. El script irá ejecutando distintos casos de simulación modificando ciertas etiquetas dentro del *MakeFile.parameters* correspondientes a L y a la dimensión de la topología.

MakeFile.parameters es utilizado por el script para generar el binario *potts-cpu* que finalmente se ejecuta midiendo el tiempo con el comando *time* de Shell.

La ejecución de *potts-cpu* se realiza 20 veces y al finalizar el script calcula el promedio y desviación estándar para el caso en cuestión.

La definición del caso varía según la ejecución se refiera a SEC, OMP; MPI o HYBR:

- para SEC, un caso se define mediante:`{L};`
- para HYBR, un caso queda definido por la tripla: `{L; Topología nxm; Nro. de Threads};`
- para OMP, un caso se define por la tupla: `{L;  Nro. de Threads};`
- para MPI, un caso se define por la tupla: `{L; Topología nxm}.`

Compilación y salida típica del código secuencial: patrón de medida

Esta salida es importante porque será el patrón base para la verificación de la corrección para todas las versiones paralelas subsiguientes.

```
$ gcc -O3 -std=c99 -Wall -Wextra -ffast-math -march=core2 -funroll-loops -lm
potts3-cpu-sec.c -o potts3-cpu-sec -DQ=9 -DL=16512 -DSAMPLES=1 -DTEMP_MIN=0.71f
-DTEMP_MAX=0.72f -DDELTA_TEMP=0.005f -DTRAN=20 -DTMAX=80 -DDELTA_T=5

$ ./potts3-cpu-sec

# Q: 9
# L: 16512
# Number of Samples: 1
# Minimum Temperature: 0.710000
# Maximum Temperature: 0.720000
# Temperature Step: 0.004999999888
# Equilibration Time: 20
# Measurement Time: 80
# Data Acquiring Step: 5
# Number of Points: 3
# Configure RNG Time (sec): 0.000692
# Total Simulation Time (sec): 2448.103235
# Temp          E           E^2            E^4            M              M^2            M^4
0.710000  -1.7857480943  3.1895005724   10.1807191870   0.9240646169   0.8540150653   0.7297542138
0.715000  -1.7305852682  2.9949609162   8.9702174160    0.8974586531   0.8054429165   0.6487734056
0.720000  -1.6979258053  2.8829858335   8.3119975262    0.8802842550   0.7749118436   0.6005239734
```

En el resto del trabajo diremos que una versión paralela dada "*verifica correción*" sólo si presenta una salida "*similar*" a la del código *secuencial original* para un conjunto de parámetros dados (Q, L, temperatura mínima, temperatura máxima, etc.). Para este trabajo "*similar*" significará aceptar variaciones únicamente a partir del segundo dígito decimal (inclusive).

Capítulo 3: Paralelización en OpenMP

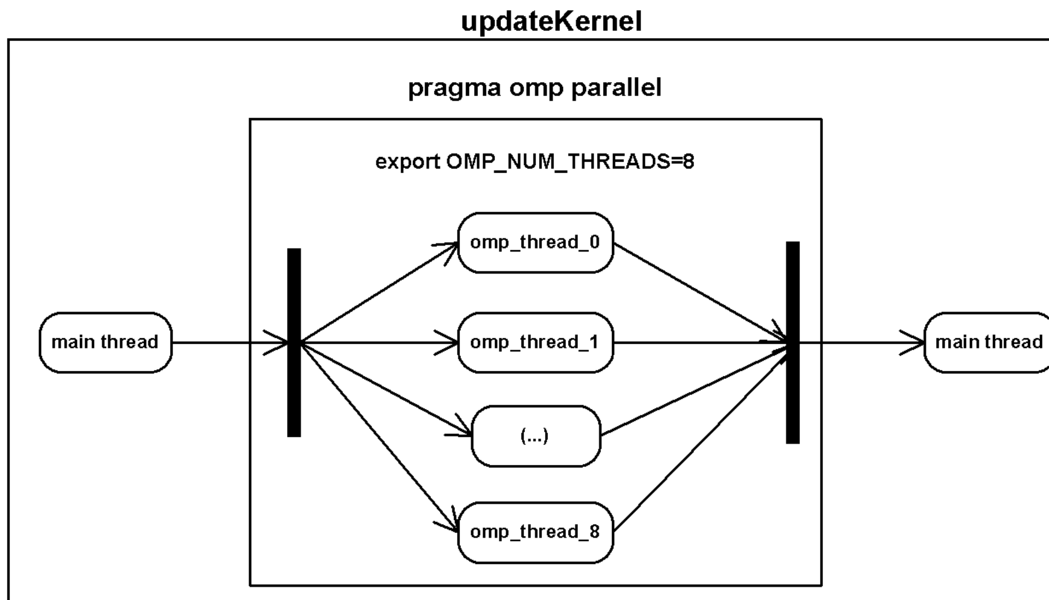
Este capítulo trata sobre los cambios que se llevaron a cabo en la versión secuencial para convertirse en una versión paralela bajo el paradigma de memoria compartida utilizando OpenMP [1]. Los cambios han sido mínimos pero causaron un altísimo impacto en el rendimiento de la aplicación.

Estructura del código OpenMP

En la siguiente sección se analizarán los cambios más significativos de la versión OMP. El código se paraleliza cuando la ejecución alcanza la línea del *pragma omp parallel,* donde se distinguen dos importantes opciones: *shared*, donde se declaran las variable compartidas entre los threads y *private*, donde se declaran las variables de uso privado. Es muy importante, para el buen funcionamiento y el *scaling* de este tipo de programas la asignación correcta de las variables en las listas *shared* y *private,* la cual se considera como una fuente muy propensa para cometer errores. A continuación se expone un resumen de la función *updateKernel* responsable de la evolución del *lattice*, donde se resalta la sección paralela:

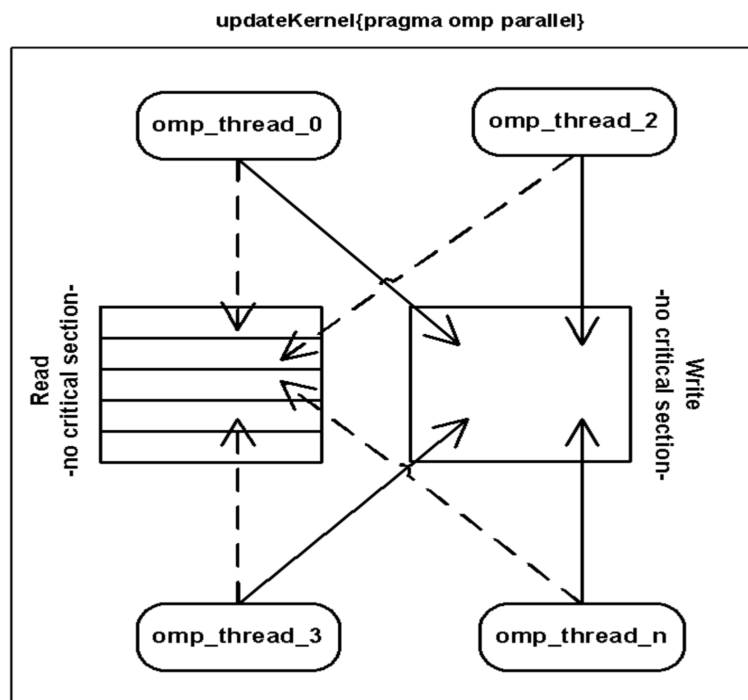
```
static void updateKernel(const float temp, const unsigned int color, byte
                         write[L/2][L],  byte read[L/2][L]) {
        (…)

        #pragma omp parallel shared(read,write,table_expf_temp)
        {
        unsigned int tid = omp_get_thread_num();
        unsigned long long x_l = x[tid];
        unsigned int a_l = a[tid];

        #pragma omp for schedule(static, CHUNKSIZE)
        for (unsigned int i=0; i<L/2; i++) {
             (...)
             for (unsigned int j=0; j<L; j++) {
                // operaciones de lectura y escritura sobre write[i][j];
                // operaciones de lectura sobre read[i][j];
                // por iteración, al menos una llamada a rand_MWC_co(&x_l, &a_l)
              }
        }
        x[tid] = x_l;
        a[tid] = a_l;
        }
}
```

Este código paraleliza el primer ciclo *for* abriéndose (método fork-join) en tantos threads como hayan sido definido en la variable de entorno exportada previamente: *export OMP_NUM_THREADS=8.*



Fork-Join. La región paralela OpenMP se bifurca en n-threads cuando alcanza la intrucción pragma.

Cada thread toma una porción de las filas de la submatriz y ejecuta un barrido para todas las columas j: 0 … L-1 leyendo a cada paso desde write[i][j] y read[i][j], y si fuera el caso, de acuerdo al sorteo de un número aleatorio (dinámica de Metropolis) el thread actualizará sobre write[i][j].



updateKernel sólo lee de blancas y lee-escribe en negras. El acceso a las submatrices no representa riesgos de race-condition.

La opción *schedule(static, CHUNKSIZE)* con *CHUNKSIZE=1* se utiliza para forzar un comportamiento tipo Round Robin en la asignación de filas de la matriz al conjunto de threads (la matriz puede ser la submatriz de blancas o negras según sea el caso). Con esta definición se previene que queden threads ociosos.

Es importante hacer notar la gran diferencia en performance lograda al cambiar como locales a las variables *x* y *a* que guardan el estado del RNG para cada thread. A continuación se muestra la transición entre los códigos desde la versión con variables globales hacia la versión final con variables locales.

Versión OMP con variables globales *x* y *a*:

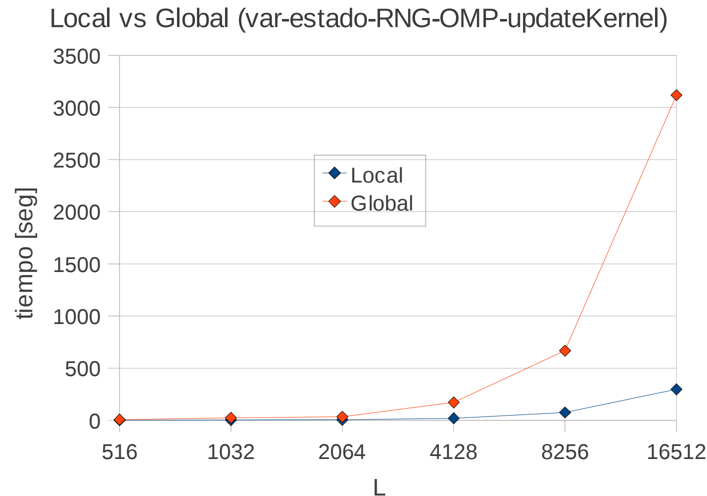
```
static unsigned long long x[MAX_NUM_THREADS];
static unsigned int a[MAX_NUM_THREADS];

updateKernel(...){
	...
	#pragma omp parallel shared(read,write,table_expf_temp,x,a)
	{
		unsigned int tid = omp_get_thread_num();

		#pragma omp for schedule(static, CHUNKSIZE)
		for (unsigned int i=0; i<L/2; i++) {
			...
			for (unsigned int j=0; j<L; j++) {
				...
				spin_new = (spin_old + (byte)(1 +
				           rand_MWC_co(&x[tid], &a[tid])*(Q-1)))%Q;
				...
				float p = rand_MWC_co(&x[tid], &a[tid]);
				...
			}
		}
	}
}
```

Versión OMP con variables locales *x_l* y *a_l* (versión final):

```
static unsigned long long x[MAX_NUM_THREADS];
static unsigned int a[MAX_NUM_THREADS];

updateKernel(...){
	...
	#pragma omp parallel shared(read,write,table_expf_temp)
	{
		unsigned int tid = omp_get_thread_num();
		unsigned long long x_l = x[tid];
		unsigned int a_l = a[tid];
		#pragma omp for schedule(static, CHUNKSIZE)
		for (unsigned int i=0; i<L/2; i++) {
			...
			for (unsigned int j=0; j<L; j++) {
				...
				spin_new = (spin_old + (byte)(1 +
				          rand_MWC_co(&x_l, &a_l)*(Q-1)))%Q;
				...
				float p = rand_MWC_co(&x_l, &a_l);
				...
			}
		}
		x[tid] = x_l;
		a[tid] = a_l;
	}
}
```

La siguiente tabla y gráfico muestran los resultados comparativos medidos en segundos para las dos versiones de OMP en el rango de *L:{516, 1032, 2064, 4128, 8256, 16512} :*

L	516	1032	2064	4128	8256	16512
Local	0,41	1,7835	5,037	19,1475	74,663	296,479
Global	5,9	22,82	31,95	170,52	666,41	3118,46



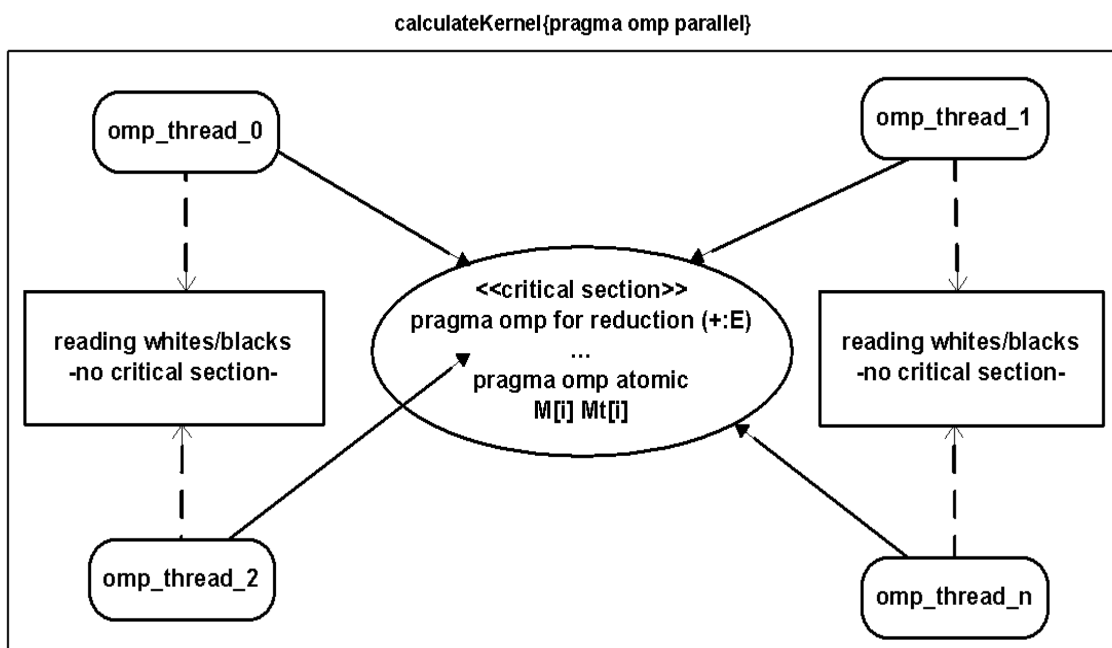
Los cambios hacia la versión final OMP también afectaron a la función calculateKernel, responsable de los cálculos finales de energía y magnetización, se muestra un resumen donde también se resalta la sección paralela:

```
static double calculateKernel( byte white[L/2][L], byte black[L/2][L],
                                unsigned int* M_max) {
        (...)
        unsigned int E=0;
        unsigned int M[Q]={0};
        unsigned int Mt[Q]={0};

        #pragma omp parallel shared(white,black,E,M)  firstprivate(Mt)
        private(i, j, spin, spin_neigh_n, spin_neigh_e, spin_neigh_s, spin_neigh_w)
        {
                #pragma omp for reduction(+:E) schedule(static, CHUNKSIZE)
                for ( i=0; i<L/2; i++) {
                    for ( j=0; j<L; j++) {
                        (...)
                                // se lee de white[i][j] y de black[i][j]
                        E += (spin==spin_neigh_n)+(spin==spin_neigh_e)
                                +(spin==spin_neigh_w) +(spin==spin_neigh_s);

                        Mt[spin] += 1;
                        (...)
                        // se lee de  black[i][j]
                        (…)
                        Mt[spin] += 1;
                    }
                }
                for (i=0; i<Q; i++) {
                        #pragma omp atomic
                        M[i] += Mt[i];
                 }
        }
        (…)
        // la funcion retorna E y el máximo de los M[i]
}
```

Aquí la novedad es la operación de “*reduction (+:)*” sobre la variable *E*. Esto significa que al finalizar el *pragma omp for* se realizará una suma final sobre todas las sumas parciales de las *E* locales de los threads.

A pesar de que E no fue incluida en la lista de *private*, la operación de *reduction* crea copias locales de E para cada thread. La acumulación de E no necesita ser sincronizada porque cada thread solamente lee, y lo hace en porciones distintas del *lattice*. Igualmente cabe aclarar que en el esquema que viene a continuacíon se incluye “*pragma omp for reduction(+:E)*” dentro del círculo que representa a la sección crítica solamente para dar idea de que la misma operación de *reduction* tomará los recaudos necesarios para condensar la suma final de los “*E locales*” en el “*E final total*” que entrega la sección paralela.

M no se pudo incluir en la operación de *reduction* debido a una restricción (es en realidad una limitación) de OpenMP 2.5 [4] que no permite arrays, punteros ni tipos por referencia en esta clase de operación. También se descartó el uso de la cláusula *omp critical* en los acumuladores de M_i (ver la versión secuencial de *updateKernel*) debido a la pobre performace demostrada. En vez de eso, se implementó un “*reduction manual*”, donde cada thread usa una copia local de M_i llamada Mt_i que se usa dentro de los dos primeros ciclos *for* y al finalizar se utiliza Mt_i para acumular las sumas parciales generadas por los threads sobre una M_i compartida (declarada en *shared*). Para esto se requirió realizar una sincronización de threads utilizando *omp atomic.*



calculateKernel lee-escribe en blancas y negras. El acceso a las submatrices no representa riesgos de race-condition. Los casos de race-condition sobre E son salvados internamente por la operación de reducción y los casos sobre M por la instrucción atomic.

Se recomienda no usar *omp critical* mientras sea posible, pues tiene realmente muy baja performance. Este es un ejemplo de la salida que obtendríamos si en lugar de usar Mt_i en el "*reduction manual*" hubiésemos optado por *omp critical* para cada uno de los acumuladores M_i:

```
#pragma omp critical
M[spin] += 1;
```

Resultados:

```
L=8192
secuencial
$ time taskset -p 0x00000001 ./potts3-cpu-SEC
16m23.96s, 16m39.66,16m23.81s
paralelo con 2 threads
$ time OMP_NUM_THREADS=2 ./potts3-cpu-OMP-critical
18m57.11s 18m46.92s, 18m53.29
```

Luego de tres corridas vemos claramente que la performance no es buena. De hecho, esta solución no escala en absoluto, la versión secuencial supera por 2 segundos a la versión paralela. También se probó la opción de usar *omp atomic* para cada acumulador M_i (ver updateKernel secuencial):

```
#pragma omp atomic
M[spin] += 1;
```

Esta opción funcionó, pero la performance no fue tan buena como la solución definitiva presentada.

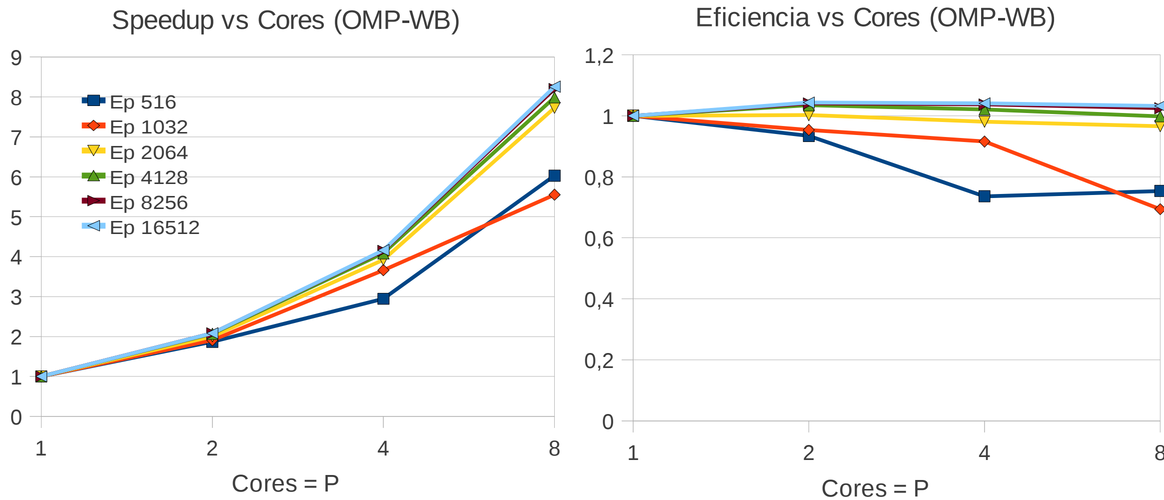
Ejecución de la versión OpenMP

```
$ gcc -O3 -std=c99 -Wall -Wextra -ffast-math -march=core2 -funroll-loops -lm -fopenmp potts3-
cpu-OMP.c -o potts3-cpu-OMP -DQ=9 -DL=16512 -DSAMPLES=1 -DTEMP_MIN=0.71f -DTEMP_MAX=0.72f
-DDELTA_TEMP=0.005f -DTRAN=20 -DTMAX=80 -DDELTA_T=5

$ export OMP_NUM_THREADS=8
$ ./potts3-cpu-OMP

# Q: 9
# L: 16512
# Number of Samples: 1
# Minimum Temperature: 0.710000
# Maximum Temperature: 0.720000
# Temperature Step (DELTA_TEMP): 0.004999999888
# Transient Equilibration Time: 20
# Transient Time in each Cycle: 80
# Time between Data Acquiring: 5
# Number of Points: 3
# Configure RNG Time: 0.000811
# Total Simulation Time: 296.394220
# Temp          E              E^2             E^4            M              M^2             M^4
0.710000   -1.7857008808     3.1893323912  10.1796522299     0.9240455826     0.8539799619     0.7296945237
0.715000   -1.7306162535     2.9950672280  8.9708430321      0.8974825772     0.8054855907     0.6488412871
0.720000   -1.6979984336     2.8832321433  8.3134140307      0.8803240233     0.7749818814     0.6006325932
```

Comparando esta salida con el patrón de salida secuencial se afirma que la versión OMP verifica corrección para todo L.

Resultados de performance en OpenMP

Se presentan a continuación los resultados de las mediciones de *speedup* y eficiencia para la versión de OMP utilizando la técnica del tablero compactado *Whites-Blacks* y 1 nodo conteniendo 8 cores, tomando datos para 1 (caso secuencial), 2, 4 y 8 cores. El conjunto de valores de L fue: {516, 1032, 2064, 4128, 8256, 16512}.



Para L grande se puede observar un comportamiento ideal para el *speedup* que se obtiene cuando S_p=P. Este comportamiento también es conocido como “*Linear Speedup*”. Para L grande la eficiencia es muy buena, alcanzando el 100%, mientras que para L pequeño la mejor eficiencia se verifica para valores bajos de P.

Capítulo 4: Paralelización en MPI

Este capítulo trata sobre los cambios que se realizaron a la versión secuencial para convertirla en una versión paralela bajo el paradigma de pasaje de mensajes utilizando MPI [2]. En este caso y al contrario de lo que ocurrió para OpenMP, fue necesario una gran cantidad de cambios al código secuencial para llegar a obtener la versión final paralela. El tiempo de desarrollo y depuración fueron muy grandes también comparados a lo que demandó la versión en OpenMP.

Topologías n×m y 1×n

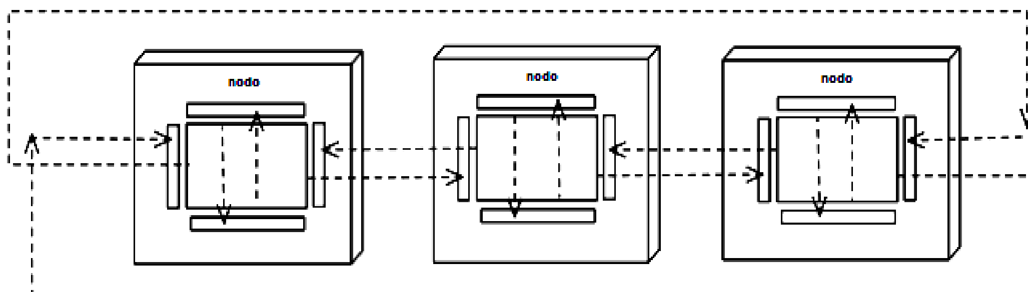
En el contexto de este trabajo llamamos topología al arreglo geométrico y lógico en el cual consideramos insertos a los nodos y la forma en que éstos pueden comunicarse.

Las versiones MPI e HYBR han sido preparadas para trabajar con ambos tipos de topologías, sin embargo cabe destacar que no se observaron diferencias sustanciales de performance entre: 2×2 vs. 1×4 ni entre 2×3 vs. 1×6, tanto para MPI como para el caso HYBR.

Para poder hacer mediciones con cantidades de nodos impares: 3 y 5 nodos, se decidió adoptar topologías de la forma 1×n para todas las pruebas realizadas.

Tanto para la versión MPI como para la HYBR y para las topologías 1×2, 1×3, 1×4, 1×6, se utilizó L: {516, 1032, 2064, 4128, 8256 y 16512}, y para la topología específica 1×5: se utilizó L:{520, 1040, 2070, 4130, 8260, 16520}. La consideración de los dos conjutos de valores para L, tiene su justificación en lo siguiente: *L debe ser divisible por el el número de nodos* entre los cuales se va a distribuir el *lattice*. En general el *scaling* de las aplicaciones paralelas pasando de 4 a 5 nodos a sido satisfactoria, y la diferencia entre los valores de L no ha representado ningún problema (por ejemplo, nótese la diferencia entre L=16520 para número de nodos igual a 5 y L=16512 para un número igual a 4).

Debido a la adopción de topologías de una sola fila aparecen en la función *syncRecv()*, los siguientes condicionales: `if (neighbors[0] != myrank_mpi){...}else{...}`. Cuando los nodos se ordenan en fila (ver esquema) sólo tienen vecinos al oeste-este, nunca al norte-sur. Entonces tenemos que cada nodo es vecino de sí mismo al norte-sur. Este control se hace en el condicional. Así entonces en caso de que el vecino a considerar sea "sí mismo", se ahorrará la transferencia *ghost* sobre la red usando MPI, pero los datos seguirán el mismo flujo "como si se hubiesen transmitidos". Dado que esta elección no representa impacto negativo en la performance de la simulación, fue realizado de esta manera por razones de claridad del código.



Topología toroidal del tipo 1xn (fila de nodos). El esquema muestra la dirección de transmisión de las filas y columnas ghost para esta topología.

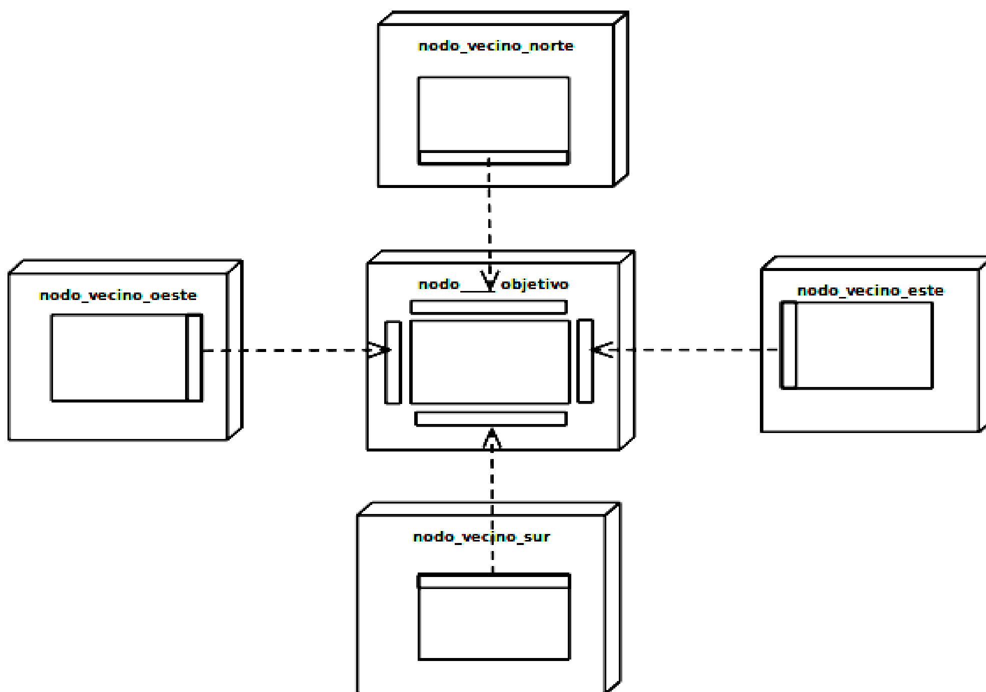
Otra característica para resaltar en la función *syncRecv()* y dentro de cada función *MPI_recv()*, son líneas de código del estilo:

```
MPI_Recv(grn, Lmpiy, MPI_BYTE, neighbors[0], MSG_DATA_S, MPI_COMM_WORLD, &statusn)
```

Notar que en el buffer receptor del norte: *grn*, se recibe la información de su vecino del sur, como lo indica el *tag*: *MSG_DATA_S* y así sucesivamente: en el buffer del este: *gre* se recibe la información del vecino del oeste: *MSG_DATA_O*. Algo tan simple como poner mal estos tags, lleva a un deadlock en estas funciones de MPI.

La técnica ghost-cells

Para trabajar con los bordes se utiliza una técnica llamada "*ghost cells*". El *lattice* local de cada nodo se amplía en dos filas: borde superior e inferior, y en dos columnas: borde izquierdo y derecho. Las celdas de estas filas y columnas externas se denominan celdas *ghost* y son de sólo lectura.



Topología toroidal del tipo n×m (arreglo cuadrado de nodos). El esquema muestra la dirección de transmisión de las filas y columnas ghost cuando se focaliza sobre un nodo específico. En las direcciones: noroeste, noreste, suroeste y sureste también hay nodos (no se muestran) que no se comunican con el nodo central por no pertener a la vecindad de von Neumann del nodo.

Las celdas *ghosts* contienen la información de vecindad necesaria para el cálculo del estado de las celdas del borde del *lattice* local del nodo. En cada llamada a la función *ghostProcessing* se actualiza el reborde *ghost* del *lattice* dejándolo listo para una nueva ejecución de *updateKernel* o de *calculateKernel*. En otras palabras, antes de la ejecución de cualquier función que tenga por propósito:

1. cambiar el estado del *lattice*,
2. o que su ejecución venga justo después de que los nodos cambiaron sus porciones locales del *lattice* sumado a que sea necesario leer el estado de las celdas del borde del *lattice* local,

se necesitará obligadamente hacer una llamada a *ghostProcessing* para refrescar la vecindad de las celdas del borde y evitar hacer cálculos con datos desactualizados que de otra forma llevarían a errores acumulables muy difíciles de detectar.

Un ejemplo del funcionamiento de esta técnica *ghost* junto con la técnica de compactación *Whites-Blacks* se da a continuación utilizando valores ficticios no relacionados con la simulación sólo como ilustración. Se representa lo que podría ser la sección local en un nodo de un mini-*lattice* global. La matriz local en el nodo, se abraza con un par de filas y un par de columnas extras que provienen desde los nodos vecinos. Estas nuevas celdas que se denominan “*ghost*” serán de sólo lectura y servirán para poder calcular el estado de las celdas del borde de la matriz local. Recordemos que para calcular el próximo estado de un celda, es necesario leer el estado actual de la misma celda y de las celdas vecinas, considerando una vecindad de von Neumann (primeros vecinos).

```
Matriz Original

M[0][0]= 0,    M[0][1]= -34,  M[0][2]=+ 35,  M[0][3]= -36,  M[0][4]=+ 37,  M[0][5]= 0,
M[1][0]= -43,  M[1][1]=+ 44,  M[1][2]= -45,  M[1][3]=+ 46,  M[1][4]= -47,  M[1][5]=+ 48,
M[2][0]=+ 53,  M[2][1]= -54,  M[2][2]=+ 55,  M[2][3]= -56,  M[2][4]=+ 57,  M[2][5]= -58,
M[3][0]= -63,  M[3][1]=+ 64,  M[3][2]= -65,  M[3][3]=+ 66,  M[3][4]= -67,  M[3][5]=+ 68,
M[4][0]=+ 73,  M[4][1]= -74,  M[4][2]=+ 75,  M[4][3]= -76,  M[4][4]=+ 77,  M[4][5]= -78,
M[5][0]= 0,    M[5][1]=+ 84,  M[5][2]= -85,  M[5][3]=+ 86,  M[5][4]= -87,  M[5][5]= 0,

Matriz transformada:

N[0][0]= 0,    N[0][1]=+ 44,  N[0][2]=+ 35,  N[0][3]=+ 46,  N[0][4]=+ 37,  N[0][5]=+ 48,
N[1][0]=+ 53,  N[1][1]=+ 64,  N[1][2]=+ 55,  N[1][3]=+ 66,  N[1][4]=+ 57,  N[1][5]=+ 68,
N[2][0]=+ 73,  N[2][1]=+ 84,  N[2][2]=+ 75,  N[2][3]=+ 86,  N[2][4]=+ 77,  N[2][5]= 0,
N[3][0]= -43,  N[3][1]= -34,  N[3][2]= -45,  N[3][3]= -36,  N[3][4]= -47,  N[3][5]= 0,
N[4][0]= -63,  N[4][1]= -54,  N[4][2]= -65,  N[4][3]= -56,  N[4][4]= -67,  N[4][5]= -58,
N[5][0]= 0,    N[5][1]= -74,  N[5][2]= -85,  N[5][3]= -76,  N[5][4]= -87,  N[5][5]= -78,
```

Matriz original y matriz resultante luego de aplicar el algoritmo de compactación.

En gris, se representa el reborde de celdas *ghost*. Se puede observar como luego de la compactación resulta una “matriz transformada” en la cual tanto las celdas normales como las *ghost* se han desplazado en *i* pero han mantenido su *j* inalterable (donde i se refiere a las filas y j a las columnas de la matriz). Una vez hecho este proceso, la matriz está lista para ser dividida en las dos submatrices de blancas y negras con celdas *ghost*:

```
Blancas:

W[0][0]=+ 0,   W[0][1]=+ 44,  W[0][2]=+ 35,  W[0][3]=+ 46,  W[0][4]=+ 37,  W[0][5]=+ 48,
W[1][0]=+ 53,  W[1][1]=+ 64,  W[1][2]=+ 55,  W[1][3]=+ 66,  W[1][4]=+ 57,  W[1][5]=+ 68,
W[2][0]=+ 73,  W[2][1]=+ 84,  W[2][2]=+ 75,  W[2][3]=+ 86,  W[2][4]=+ 77,  W[2][5]=+ 0,

Negras:

B[0][0]= -43,  B[0][1]= -34,  B[0][2]= -45,  B[0][3]= -36,  B[0][4]= -47,  B[0][5]= 0,
B[1][0]= -63,  B[1][1]= -54,  B[1][2]= -65,  B[1][3]= -56,  B[1][4]= -67,  B[1][5]= -58,
B[2][0]= 0,    B[2][1]= -74,  B[2][2]= -85,  B[2][3]= -76,  B[2][4]= -87,  B[2][5]= -78,
```

Submatrices blancas y negras, resultantes de aplicar el algoritmo de compactación.

Durante la actualización de las submatrices hay que tener mucho cuidado en no modificar los valores de las celdas *ghost*, teniendo presente siempre que deben ser consideradas de sólo lectura.

El lattice en MPI

En esta versión el *lattice* se divide en tantas porciones como nodos haya. La división del *lattice* obedece a cierta topología predefinida (arreglo de nodos), donde el tamaño lineal del *lattice* L debe ser divisible por el número de nodos. Al elegir una topología es necesario calcular las vecindades entre nodos, puesto que cada nodo deberá intercambiar los bordes de su *lattice* local (sin compactar) con cada uno de sus vecinos, para poder calcular el estado de sus propias celdas del borde.

El proceso es complicado, porque si bien el nodo procesa las celdas según el algoritmo de compactación que las matiene separadas físicamente en dos submatrices (blancas y negras), al momento de transmitir los bordes debemos considerar el *lattice* sin compactar como si fuera un tablero de ajedrez en una sola pieza, extraer los bordes, transmitirlos con MPI y en el destino se volverlos a mapear en la forma compacta de blancas y negras.

En la versión MPI cada celda del *lattice* es modelada mediante un *struct* que tiene tres campos, uno llamado ".*data*" es el valor del spin, otro llamado ".*isGhost*" que define si la celda es o no es *ghost* y se carga en la funcion *defineGhostCells().* El tercer parámetro llamado "*ii*" juega un papel fundamental en la funcion *ghostLoading()*, llevando el registro de donde provenía la celda antes del compactamiento en blancas y negras. Sólo es importante la posición en i porque j no varía al pasar del tablero de ajedrez al modelo compactado, debido a esto no se incluyó ningún campo "*jj*" en el *struct*.

```
typedef struct{
      byte          data;
      byte          isGhost;
      unsigned int  ii;
} cell;
```

Luego el *lattice* queda definido asi:

```
#define Lmpix ((L/NODESX)+2),
#define Lmpiy ((L/NODESY)+2),

cell whites[Lmpix/2][Lmpiy],
cell blacks[Lmpix/2][Lmpiy],
```

Donde los "+2" son debido a la existencia del reborde de celdas *ghost* (existen 2 filas y 2 columnas *ghost*).

Cálculo de Nodos Vecinos

El cálculo de los vecinos a nivel de nodos, se realiza usando también la técnica de celdas *ghost* pero quedando claro que este artificio no tiene nada que ver con el reborde *ghost* del *lattice* local de cada nodo. El proceso se apoya en una matriz llamada *TOPO*, que ha sido ampliada mediante 2 filas y 2 columnas *ghost*:

```
int TOPO[NODESX_AMPL][NODESY_AMPL];
```

donde:

```
#define NODESX_AMPL  (NODESX+2) ,
#define NODESY_AMPL  (NODESY+2),
```

El cálculo se ejecuta mediante las funciones: *initNodeMesh()*, *buildToroNodeMesh()* y *calculateNodeNeighbors()*. Al finalizar, cada nodo tendrá cargado y listo para usar un vector llamado *neighbors[]* con la información de sus vecinos.

Nodo Root y nodos Workers

Se define nodo *Root* de la red al nodo cuyo *rank* (identificador de nodo dado por la plataforma MPI) sea igual a cero, y a todos los demás como nodos *Workers*. Además de las responsabilidades únicas que un nodo *Root* debe cumplir, también deberá realizar las funciones comunes a todos los demás nodos *Worker*.

Responsabilidades del nodo *Root*:
Al comienzo de la simulación se encarga de distribuir los vectores X y A para el funcionamiento del generador de números aleatorios (RNG) de cada nodo. Luego durante la simulación condensa los resultados parciales de los cálculos efectuados en los nodos. Al finalizar, realiza el cálculo global de la energía-magnetización y presenta los resultados al usuario.

Responsabilidades del nodo *Worker*:
Se encarga solamente de la evolución de su *lattice* local intercambiando los bordes con sus nodos vecinos y actualizando los valores de spins para las submatrices de *Whites-Blacks* y del cálculo parcial de energía-magnetización.

Así entonces las secciones marcadas con la condición `"if (myrank_mpi==ROOT){}"` sólo serán ejecutadas por el nodo *Root*.

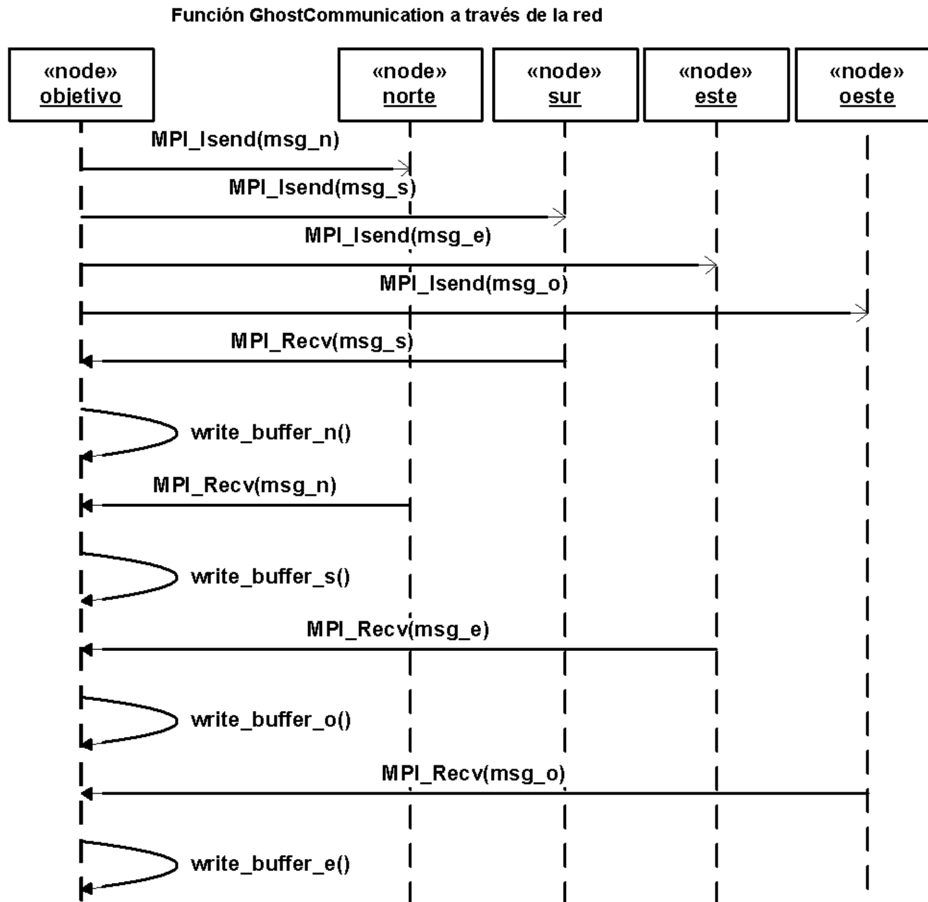
Estructura del código MPI

La transmisión de los bordes se hace en *ghostProcessing y* consta de tres llamadas: La primera es a *ghostLoading*, que toma de las submatrices *Whites-Blacks* aquellas celdas que estaban en el borde del tablero original sin compactar (por esto es importante el campo *ii* del *struct*) y las carga en los vectores de envío. La segunda es una llamada a *ghostComunication* que usa las funciones MPI de envío y recepción. Por último tiene una llamada a *ghostMapping* que toma los buffers de recepción y los remapea a las posiciones correspondientes dentro de las submatrices *Whites-Blacks*.

Dinámica del procesamiento ghost en un nodo

Proceso de intercambio de los bordes del lattice local de cada nodo con sus vecinos.

Se sabe que comunicaciones síncronas de envío y recepción en MPI conducen a deadlocks. Así entonces para evitar estos comportamientos indeseados se optó por implementar la estrategia del envío asíncrono y recepción síncrona mediante las funciones *asyncSend, syncRecv* y *syncWaitForAsyncSend* la cuales utilizan llamadas a MPI_send y *MPI_Recv*, como se muestra en el siguiente diagrama de interacción:

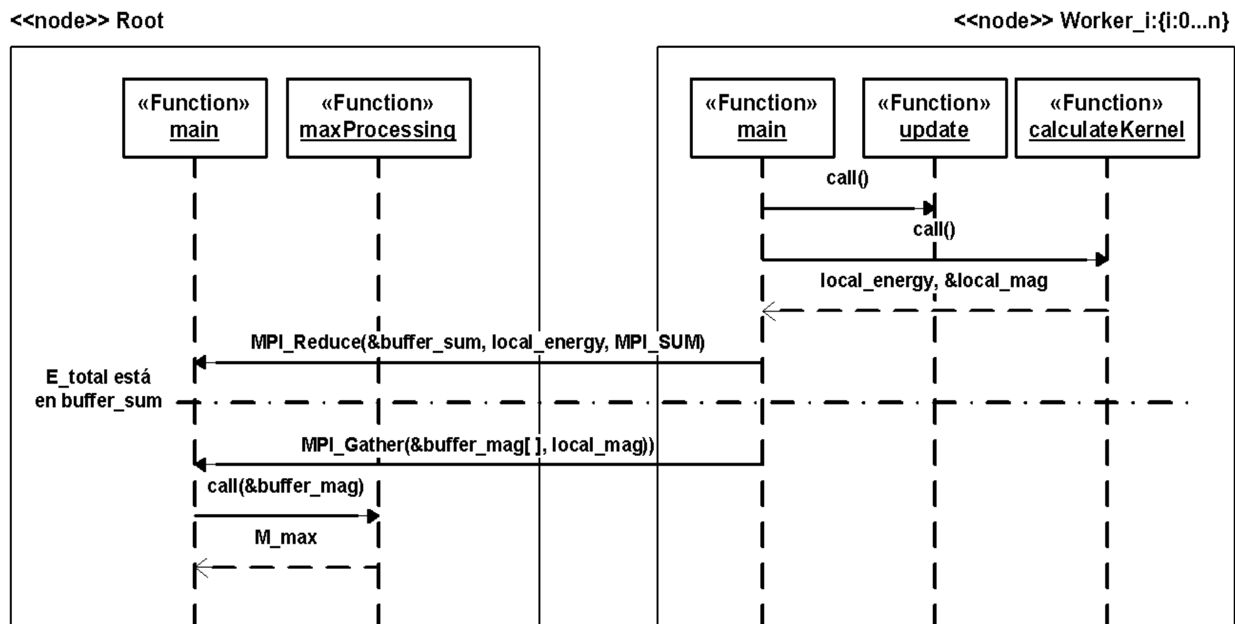


ghostCommunication utiliza técnicas de comunicación síncronas y asíncronas para evitar deadlocks.

Una de las nuevas características del *updateKernel* para la versión de MPI es la condición "`if (write[i][j].isGhost==NOGHOST){}`" que evita procesar las celdas *ghost* que son de sólo lectura y la eliminación del cálculo toroidal *%L* y *%L/2*. Cada nodo no es un toroide en sí mismo, sino que forma parte de un toroide global que ha sido distribuido en muchos nodos por la red de acuerdo a una topología, por lo tanto este tipo de cálculo ya no es necesario. En consecuencia, al llegar a su borde cada nodo no debe continuar sobre sí mismo en el extremo opuesto como un toro, sino leer simplemente la información de las celdas *ghost* que representan los bordes de los nodos vecinos.

Para mantener los bordes del *lattice* local continuamente actualizados, son muy importates las llamadas a *ghostProcessing* justo antes de cada llamada a u*pdateKernel* y *calculateKernel*. Esto asegura la actualización global del toroide durante toda la simulación.

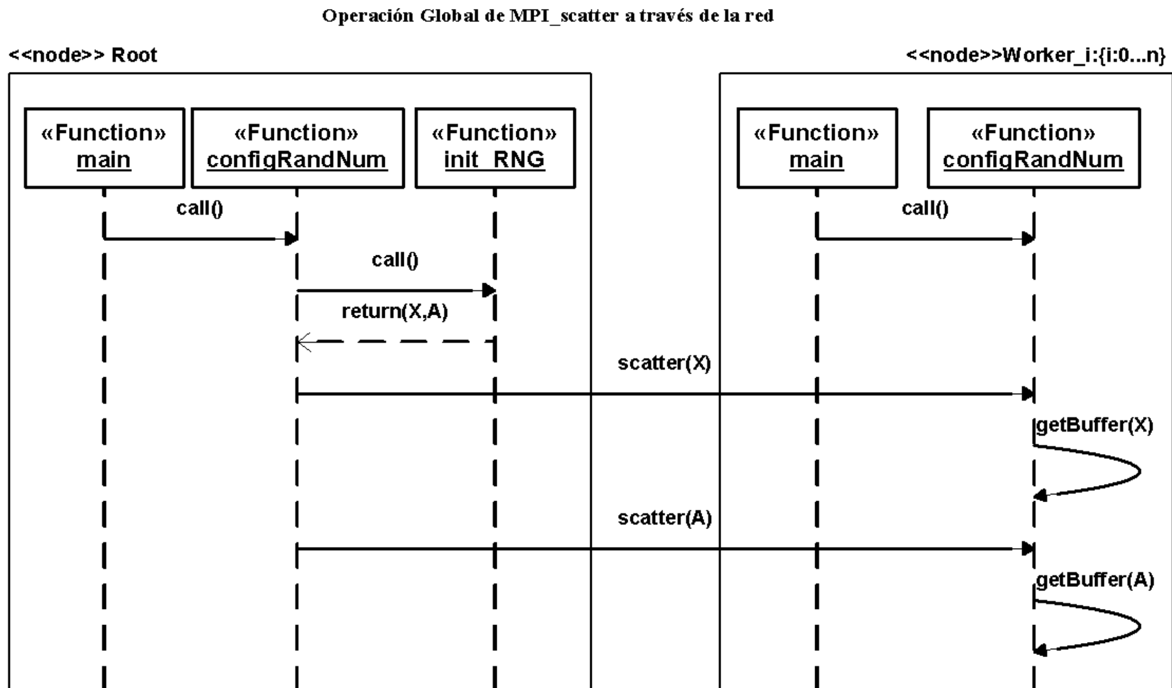
El cálculo de la energía global consta de una simple llamada a *MPI_Reduce* la cual condensa en el nodo *Root* la suma final de las energías parciales calculadas en los nodos. El procesamiento del magnetismo es más complejo: el circuito empieza en *calculateKernel* que ahora escribe en un vector global *Ma[Q]* y ya no devuelve el valor de *M_max* como valor por referencia como se hacía antes (en SEC y OMP), sigue en *Cycle* a nivel de nodo *Worker* con el procesamiento de *MPI_Gather* en cada nodo y luego retoma *Cycle* pero a nivel de *Root*, donde se hace el cálculo final llamando a la función *maxProcessing*. A continuación se brinda un diagrama de interacción que ilustra el proceso:



Proceso del cálculo global de la energía y del magnetismo: La función cycle está modelada en la línea de tiempo de main.

Generador de números aleatorios distribuido

En el contexto de MPI, la distribución de las series aleatorias iniciales *X[]* (semillas) y *A[]* (multiplicadores, ver Capítulo 2) para el funcionamiento de los RNG locales a los nodos, fue un problema a resolver. La función que implementa la solución se llama c*onfigureRandomNumbers* donde se usa ampliamente la técnica de dispersión de vectores usando *MPI_Scatter*. A continuación se muestra un diagrama de interacción detallando el proceso:



Distribución global de las series aleatorias X,A para iniciar el RNG distribuido.

En main, se puede observar que configureRandomNumbers no está en un bloque *if* exclusivo del *Root*. Esto es porque dentro de la función se realiza una operación global de dispersión (*scatter*) que necesita ser procesada por todos los nodos. Pero también notar que dentro de *configureRandomNumbers* solo el *Root* ha de generar *nrows×ncols* pares *(X,A)*, donde *nrows=NODE_THREADS* y *ncols=nprocs_mpi*, es decir: la cantidad de *threads* por nodo y la cantidad de nodos respectivamente. Cada nodo recibe su parte en los vectores *x* y *a* (en minúsculas) y los accederán según el número de thread así: *x[thread_id]* y *a[thread_id]*.

El *thread_id* se obtiene de distintas formas según sea la versión:
Para MPI: `tid=(i*Lmpix+j)%NODE_THREADS`, donde *NODE_THREADS* se define en *main* y es igual a 8. Si bien, no tenemos *threads* en la versión MPI pura, se mantuvo esta denominación por razones de compatibilidad con el código secuencial y con la versión para CUDA. La idea principal es que todas las versiones: SEC, CUDA por el lado de las originales y OMP, MPI e HYBR por el lado de las nuevas generadas por este trabajo se parezcan lo más posible. Siempre se quiere paralizar causando el menor "daño" posible a las versiones originales, entendiéndose "daño" en este contexto como un proceso de cambio. Para las versiones OMP e HYBR la obtención del *thread_id* es más natural y se obtiene fácilmente con una llamada a la función *omp_get_thread_num,* donde para estos casos: *NODE_THREADS* se define mediante dos pasos:

1. en el Shell de Linux se declara: *$ export OMP_NUM_THREADS=8,*
2. en la simulación se toma ese valor mediante la siguiente expresión: `NODE_THREADS=omp_get_num_threads()`

Fase de Equilibrio

La *fase de equilibrio* se realiza en la función *cycle()* (invocada únicamente por *main*) también de forma distribuida invocando *TRAN* veces (*Transient Equilibration Time*, fijado en 20 MCS) a la función *update(temp, whites, blacks).* Cada llamada a la función *update* implica dos llamadas a *ghostProcessing*, cada una antes de la actualización de las submatrices de blancas y negras.

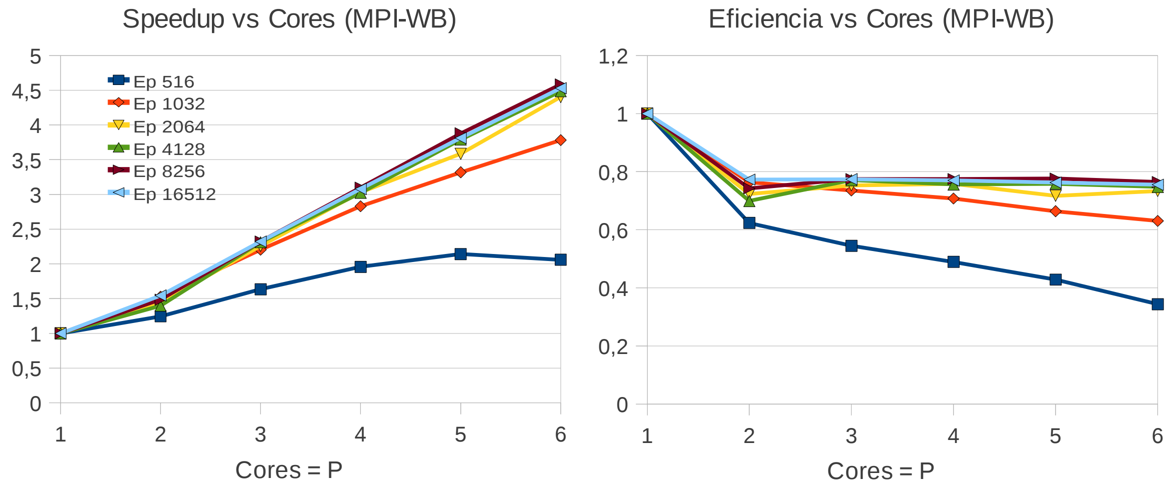
Compilación y ejecución MPI

```
$ cat mymachinefilecluster
compute-0-0
compute-0-1
compute-0-2
compute-0-4
compute-0-5
compute-0-6

$ mpicc -O3 -std=c99 -Wall -Wextra -ffast-math -march=core2 -funroll-loops -lm potts3-cpu-MPI.c -o potts3-cpu-MPI -DQ=9 -DNODESX=1 -DNODESY=6 -DL=16512 -DSAMPLES=1 -DTEMP_MIN=0.71f -DTEMP_MAX=0.72f -DDELTA_TEMP=0.005f -DTRAN=20 -DTMAX=80 -DDELTA_T=5

$ mpirun -np 6 -machinefile mymachinefilecluster  ./potts3-cpu-MPI

# Q: 9
# L: 16512
# Number of Samples: 1
# Minimum Temperature: 0.710000
# Maximum Temperature: 0.720000
# Temperature Step (DELTA_TEMP): 0.004999999888
# Transient Equilibration Time: 20
# Transient Time in each Cycle: 80
# Time between Data Acquiring: 5
# Number of Points: 3
# Configure RNG Time: 0.003007
# Total Simulation Time: 538.217339

# Temp          E                E^2          E^4           M              M^2            M^4
0.710000  -1.7856241681    3.1890620778  10.1779741781   0.9240135827   0.8539216283   0.7295976163
0.715000  -1.7305222894    2.9947426033  8.9689057002    0.8974224300   0.8053777913   0.6486681448
0.720000  -1.6980458034    2.8833929477  8.3143405737    0.8803456179   0.7750197647   0.6006908865
```

Comparando esta salida con el patrón de salida secuencial se afirma que la versión MPI verifica corrección para todo L.

Resultados de performance en MPI

A continuación se presentan los resultados de las mediciones de *speedup* y eficiencia para la versión de MPI utilizando la técnica del tablero compactado *Whites-Blacks* en 6 nodos, tomando datos para 1 nodo (caso secuencial), 2, 3, 4, 5 y 6 nodos (cada nodo usando 1 sólo core).



Se observa una eficiencia bastante buena, llegando a valores cercanos al 80% para L grande.

Capítulo 5: Paralelización Híbrida: OpenMP+MPI

En este capítulo se presentan los resultados de la versión híbrida, la cual combina las características de la versiones MPI y OMP. Llegando de esta forma a aprovechar todos los cores disponibles en el cluster: un total de 48 cores distribuidos en 6 nodos.

Respecto al código híbrido hay que decir que no presenta modificaciones altamente significativos respecto de sus versiones padres tomadas individualmente. La versión híbrida se obtuvo tomando la versión MPI como base y modificando las funciones *updateKernel* y *calculateKernel* paralelizándolas usando OpenMP.

Compilación y ejecución híbrida

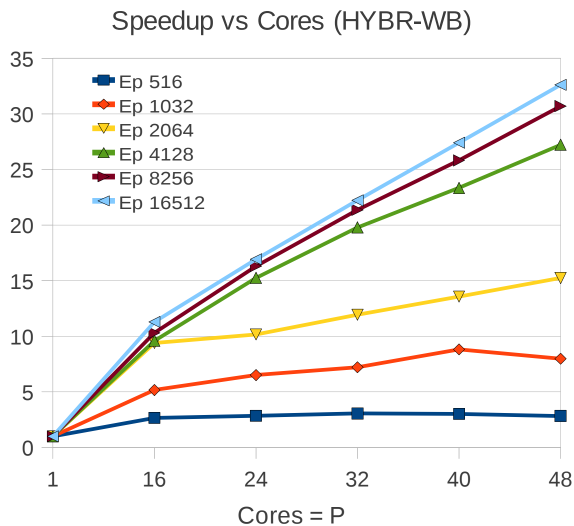
Para las topologías 1×2, 1×3, 1×4, 1x6: variando L: {516, 1032, 2064, 4128, 8256 y 16512}, y para la topología 1×5: se uso L:{520, 1040, 2070, 4130, 8260, 16520}

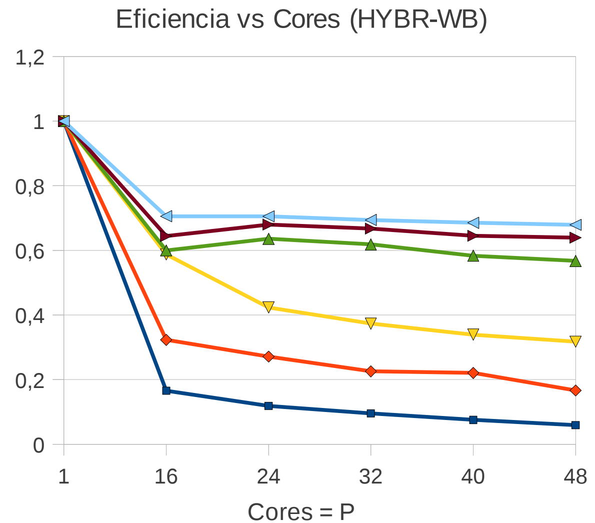
```
$ cat mymachinefilecluster

compute-0-0
compute-0-1
compute-0-2
compute-0-4
compute-0-5
compute-0-6

$ mpicc -O3 -std=c99 -Wall -Wextra -ffast-math -march=core2 -funroll-loops -lm -fopenmp potts3-cpu-HYBR.c -o potts3-cpu-HYBR -DQ=9 -DNODESX=1 -DNODESY=6 -DL=1032 -DSAMPLES=1 -DTEMP_MIN=0.71f -DTEMP_MAX=0.72f -DDELTA_TEMP=0.005f -DTRAN=20 -DTMAX=80 -DDELTA_T=5

$ export OMP_NUM_THREADS=8

$ mpirun -np 6 -machinefile mymachinefilecluster -x OMP_NUM_THREADS ./potts3-cpu-HYBR

# Q: 9
# L: 16512
# Number of Samples: 1
# Minimum Temperature: 0.710000
# Maximum Temperature: 0.720000
# Temperature Step (DELTA_TEMP): 0.004999999888
# Transient Equilibration Time: 20
# Transient Time in each Cycle: 80
# Time between Data Acquiring: 5
# Number of Points: 3
# Configure RNG Time: 0.003118
# Total Simulation Time: 74.816760
# Temp        E              E^2            E^4             M              M^2            M^4
0.710000  -1.7857020882   3.1893366612   10.1796784753   0.9240461979   0.8539812035   0.7296969766
0.715000  -1.7305743065   2.9949237484   8.9700040532    0.8974448439   0.8054183262   0.6487344247
0.720000  -1.6978697797   2.8827950193   8.3108907969    0.8802269829   0.7748109871   0.6003675797
```

Comparando esta salida con el patrón de salida secuencial se afirma que la versión HYBR verifica corrección para todo L.

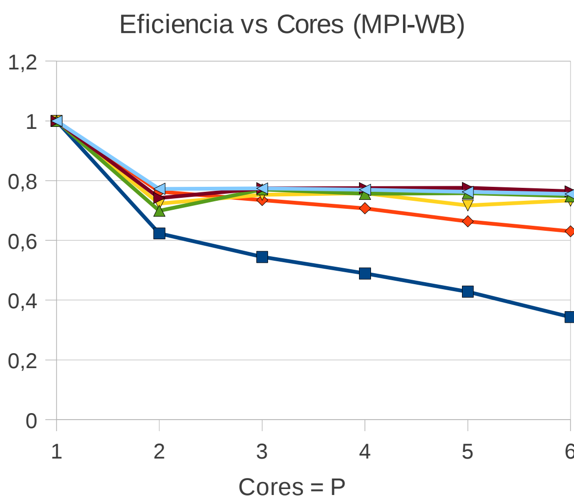
Resultados de performance y problemas detectados en la versión HYBR

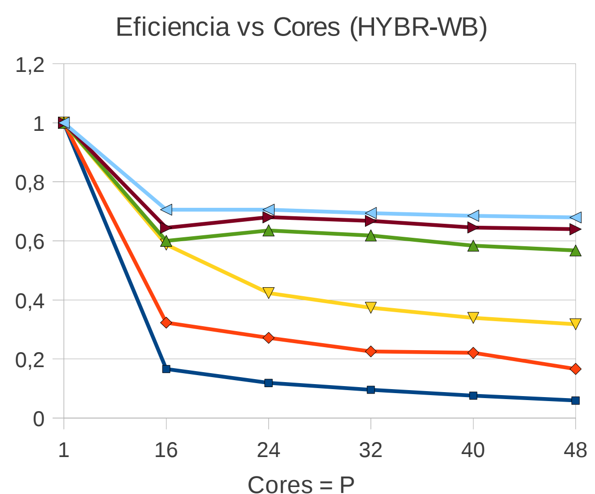
Se presentan a continuación los resultados de las mediciones de *speedup* y eficiencia para la versión de HYBR utilizando la técnica del tablero compactado *Whites-Blacks* en 6 nodos de 8 cores cada uno, tomando datos para 1 (caso secuencial), 2, 3, 4, 5 y 6 nodos (cada nodo usando 8 cores).





Observaciones: La eficiencia de la versión HYBR es buena llegando al 70% para L grande. Sin embargo, en la comparación con versiones paralelas anteriores se deja ver un problema naciente. La versión híbrida presenta una eficiencia inferior a la versión MPI usando la misma técnica de compactación *Whites-Blacks*. Este resultado es poco satisfactorio desde el momento en que la versión HYBR utiliza 42 cores, o sea 36 cores más que la versión MPI. A continuación se muestran gráficos que contrastan la diferencia mencionada y en la siguiente sección se busca una alternativa para este problema.





Prueba de una alternativa: utilización del tablero-simple

En esta sección se presentan los resultados obtenidos luego de un cambio radical en la concepción del *lattice*. Se deja atrás la técnica de compactación de *Whites-Blacks*, pasando a un esquema de “tablero de ajedrez simple”. En este caso no hay separación de submatrices blancas y negras, cada porción del tablero global que procesan lo nodos es a su vez un “tablero de ajedrez simple”

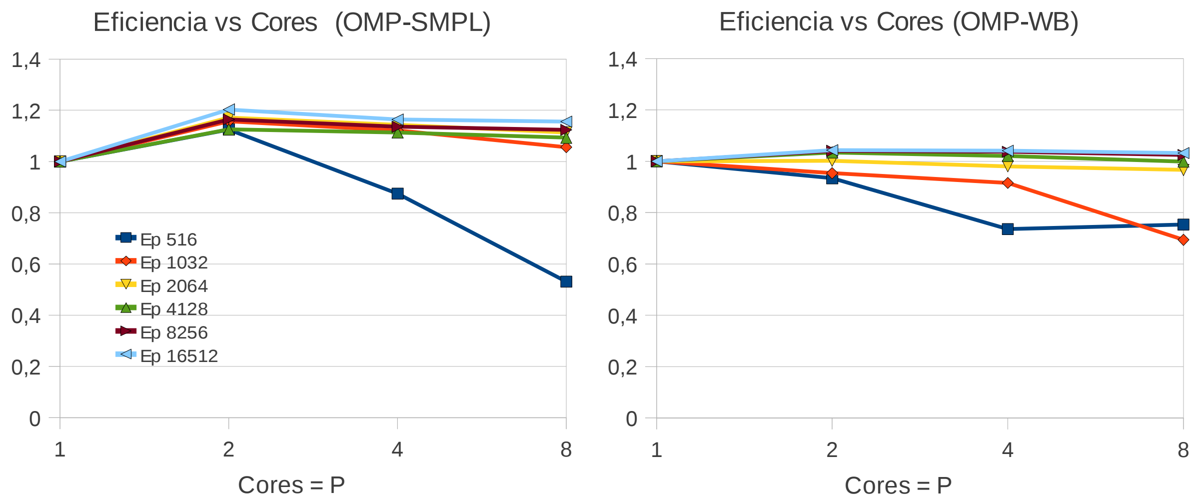
En esta ocasión hubo que generar una nueva versión secuencial que implemente el tablero simple llamada SEC-SMPL. La performance de esta versión (y no la de la versión original SEC-WB) se utilizó para el cálculo del *speedup* y eficiencia para todas las versiones paralelas SMPL. La versión SEC-SMPL evidenció una mejora poco significativa respecto a la versión *secuencial original* y sólo para L grande (ver tabla: Secuenciales vs L). Por ejemplo para L=16512 la mejora de SEC-SMPL fue de 27 seg. en un total de 2447 seg. para la versión SEC-WB.

L	516	1032	2064	4128	8256	16512
SEC-SMPL	2,38	9,684	38,772	149,25	603,342	2420,71
SEC-WB	2,4715	9,8985	38,917	152,927	611,991	2447,64

Para las versiones paralelas si bien se sigue utilizando la técnica *ghost cells* para intercambiar los bordes entre los nodos vecinos, se cuenta ahora con la ventaja de que el proceso de actualización del *lattice* se hace con una sola llamada a la función *updateKernel* por paso de Montecarlo, esto a su vez implica un ahorro del 50% de los mensajes viajando por la red, debido a que se invoca una sola vez por MCS a la funcion *ghostProcessing*.

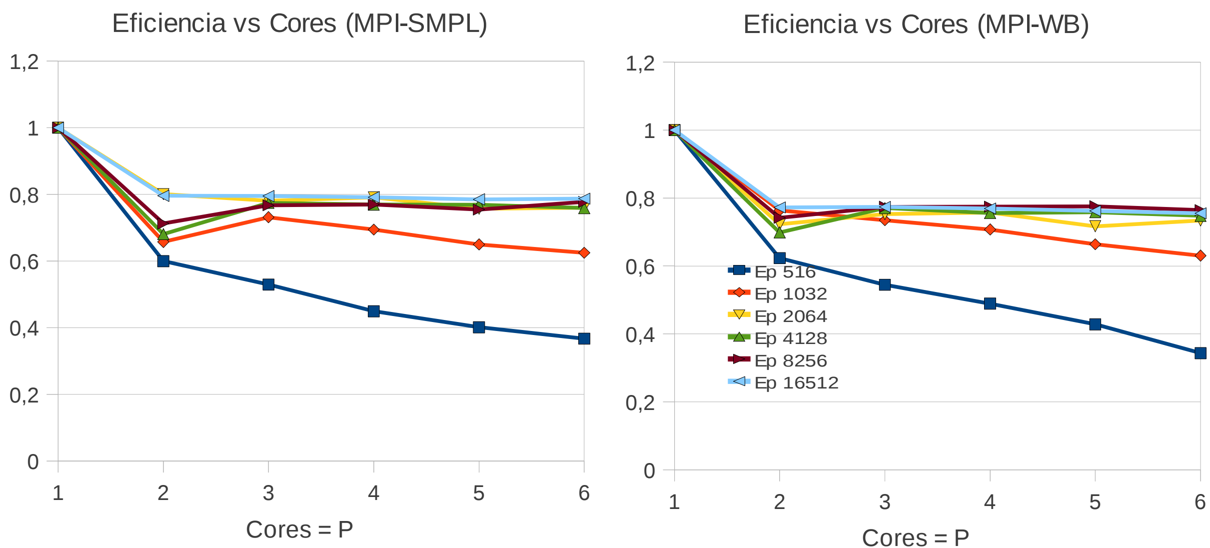
Observación importante: SEC-SMPL y todas las versiones paralelas derivadas: OMP-SMPL, MPI-SMPL, HYB-SMPL verificaron corrección para todo L. Los ejemplos de compilación y salidas de los programas se ha obviado en pos de facilitar la lectura del presente trabajo y por ser “similares” (“similar” definido en el contexto del trabajo final, ver Capítulo 2) a las salidas de sus contrapartes que usan la técnica *Whites-Blacks*.

Comparación en eficiencia de la versión OMP: SMPL vs. WB



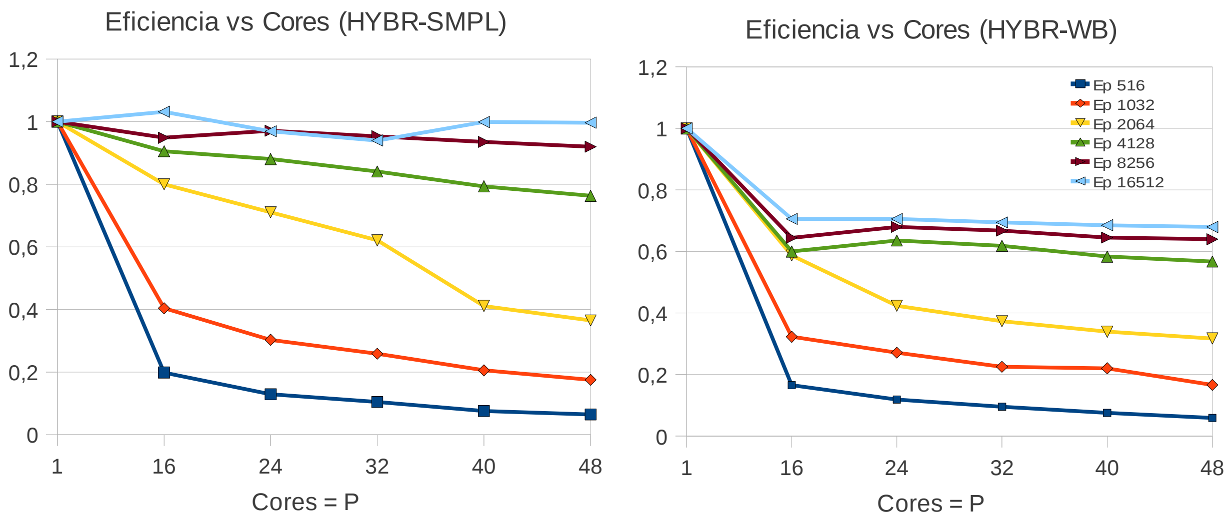
Observaciones: La versión OMP-SMPL mejora mucho comparado con la técnica de *Whites-Blacks*, mostrando un comportamiento mejor que lo ideal, conocido como "*Super Linear Speedup*", que traducido en eficiencia significa que la gráfica para la mayoría de los L ha superado la cota del 100%.

Comparación en eficiencia de la versión MPI: SMPL vs. WB



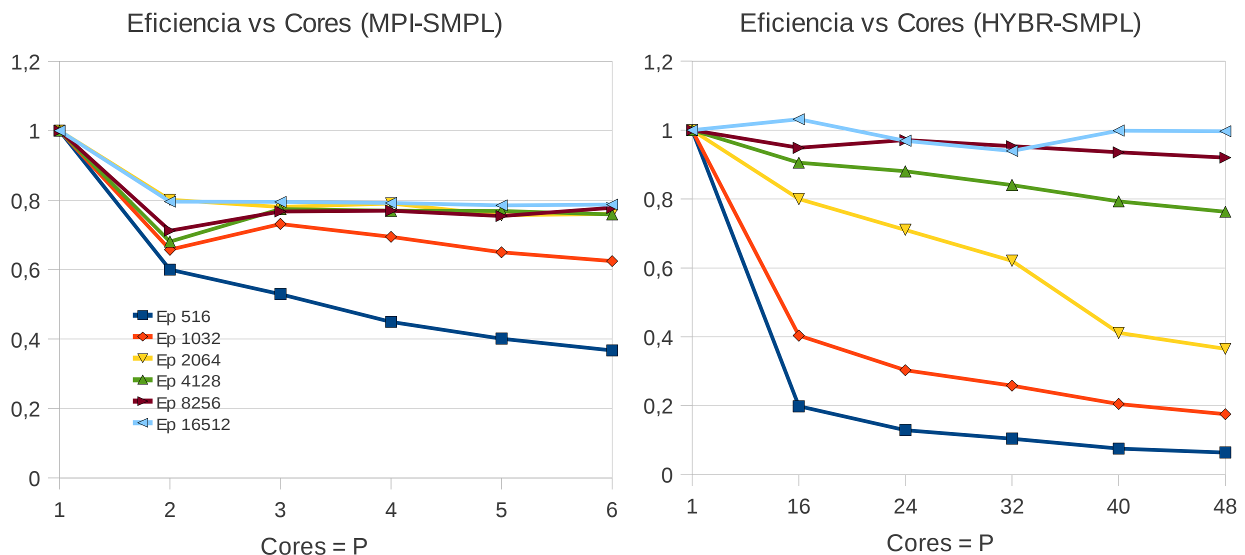
Observaciones: Para el caso de MPI, no se ha evidenciado un cambio importante en el rendimiento. Sólo se aprecia una leve mejora para L grandes en la versión del tablero simple.

Comparación en eficiencia de la versión HYBR: SMPL vs. WB



Observaciones: La versión HYBR ha mejorado notablemente para L grande pasando de eficiencias en el rango 60-70% para la técnica *Whites-Blacks* al rango 90-100% para la implementación del tablero simple, con una supuesta tendencia a seguir mejorando la eficiencia para L superiores a 16512.

Comparación en eficiencia de la versión MPI-SMPL vs HYBR-SMPL



Observaciones: En la implementación del tablero simple la versión HYBR-SMPL para L>=4128, muestra una mejor eficiencia que la versión MPI-SMPL. Esto era el resultado que se estaba buscando. Para L>16512, a partir del gráfico de MPI se podría suponer que existe una tendencia a quedar acotado en una eficiencia del 80%, mientras que por el contrario para el gráfico de HYBR se podría suponer una tendencia a mejorar aún más la eficiencia.

Conclusiones

Para la versión HYBR los resultados fueron satisfactorios obteniéndose eficiencias entre 80 y 100% para L grande después del cambio en la implementación del *lattice* desde la técnica *Whites-Blacks* hacia la del *Tablero-Simple*.

La penalización por utilizar un tablero compactado y dividido en dos submatrices se evidenció verificándose un bajo rendimiento de la aplicación híbrida que utilizaba 48 cores frente a la aplicación MPI que solamente usaba 6. Si bien esta técnica tiene grandes ventajas en entornos GPGPU, ha quedado demostrada sus debilidades en entornos Cluster al punto que no recomiendo su utilización.

Es importante notar que, luego de la corrección hacia el *Tablero-Simple* la eficiencia de la versión HYBR supera a la versión MPI, para L grande.

Para el caso de la versión OMP, los resultados siempre superaron las expectativas mostrando un comportamiento ideal y alcanzando el comportamiento de super-escalabidad (*Super-Linear-Speedup*) para la versión con *Tablero-Simple*.

El caso de MPI también fue satisfactorio, la eficiencia se mantuvo invariable para L grande en el rango de 60-80%, tanto para las versiones *Whites-Blacks* como para la que usaba el *Tablero-Simple*.

Trabajo futuro

Varias opciones se han planteado como posibles trabajos futuros que pueden usar como base lo que se ha estudiado en el trabajo actual:

Investigar en profundidad las razones por las cuales, bajo la técnica *Whites-Blacks de Gauss-Seidel,* se evidenció una mejor eficiencia de la versión MPI frente a la HYBR.

Hacer una implementación de OMP+CUDA para aprovechar todo el poder de cómputo paralelo en un nodo con placas gráficas NVidia.

Extender la simulación con implementaciones donde el *lattice* sea de 3 dimensiones: generando 4 tipo de versiones: MPI, OMP, HYBR, OMP+CUDA.

Implementar en paralelo otros sistemas de spin, como ser: Ising, Edward-Anderson, modelo Gaussiano.

Agregar interacciones a "segundos vecinos".

Agradecimientos

Un agradecimiento muy especial para: Nicolás Wolovick, Javier Blanco y Ezequiel Ferrero por todo el apoyo brindado para la realización de este trabajo final.

Referencias

[1] http://openmp.org/, The OpenMP® API specification for parallel programming.

[2] http://www.open-mpi.org/, An Open source High Performance Message Passing Library.

[3] Ezequiel Ferrero, Juan Pablo De Francesco, Nicolás Wolovick, Sergio Cannas, q-state Potts model metastability study using optimized GPU-based Monte Carlo algorithms, Computer Physics Communications Journal. In Press http://dx.doi.org/10.1016/j.cpc.2012.02.026 (2011) 6, 9-10.

[4] http://www.openmp.org/mp-documents/spec25.pdf, OpenMP Application Program Interface, Spec 2.5 (2005).

[5] Ezequiel Ferrero, "Dinámica de Relajación del Modelo de Potts de q estados bidimensional: una contribución a la descripción de propiedades de no-equilibrio en transiciones de fase de primer orden", Tesis Doctoral (Dir.: Dr. Sergio A. Cannas), Universidad Nacional de Córdoba (2011) 38-44, 101, 142, 185-188.

[6] http://www.uml.org/, UML®-The Unified Modeling Language™.

[7] B. W. Kernigham, D. M. Ritchie, The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[8] http://developer.nvidia.com/, NVidia CUDA C Programming Guide, version 3.2 (2010)

[9] http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html, "Oracle Grid Engine" anteriormente conocido como “Sun Grid Engine (SGE)”, Oracle Corporation (2010)

Página en blanco dejada intencionalmente.

Página en blanco dejada intencionalmente.