Trabajo Final de la Cátedra Computación Paralela y Distribuida

Gridificación de una aplicación de Multiplicación de Matrices utilizando GridGain

Marcelo Javier Rodríguez

mrodriguez@alumnos.exa.unicen.edu.ar

Tandil, Septiembre de 2013

Profesores:

Dr. Ing. Zunino, Alejandro.
Dr. Ing. Mateos Diaz, Cristian.

Resumen

Este documento reporta la experiencia obtenida al implementar una aplicación de multiplicación de matrices originalmente escrita en Lenguaje Java (habilitada para ejecución en grid vía Ibis Satin) , utilizando el middleware GridGain. El reporte se enfoca en el método considerado para diseñar un algoritmo paralelo distribuido que se ajusta al modelo de GridGain; finalmente, se analiza la complejidad temporal del algoritmo y resultados reales de su ejecución.

1. Diseño del algoritmo paralelo distribuido.

A la implementación Java de multiplicación de matrices proporcionada por la cátedra es necesario modificarla para obtener una versión secuencial a partir de la cual desarrollar una versión gridificada. Para lograr esto, debemos eliminar las sentencias relativas a Satin (sync, extends SatinObject, etc.), y la funcionalidad que será ejecutada explícitamente en forma distribuida. Dichas modificaciones están presentes en los archivos de código fuente entregables.

Disponiendo de un código fuente secuencial, el primer paso consiste en identificar las unidades de ejecución paralela; en algunos casos, el paralelismo puede resultar explicito; en otros, la semántica del problema en cuestión exhibe relaciones inherentes a su solución: probablemente serán dependencias verdaderas que implican un orden secuencial en las operaciones y disminuyen las oportunidades del paralelismo que podamos explotar. La multiplicación de matrices A y B de dimensión N puede escribirse como:

```
for (int i=0; i < N; i ++)
        for (int j=0; j < N; j++)
                for (int k=0; k < N; k++)
                        C [i, j] + = A [i, k] * B [k, j];
```

Llamaremos Jobs a las unidades de ejecución en un ambiente paralelo distribuido. Si analizamos el código anterior, se admiten varias y diferentes implementaciones de Jobs. Ej.: desenrollando completamente los bucles, podemos obtener paralelismo completo aunque limitado por la sincronización al momento completar las sumas. Entre otras, una segunda opción sería: cada Job calcula el bucle "for (int k….)". Ambas alternativas generan un número diferente de Jobs; también, vale destacar que el número de Jobs está relacionado con el método de división y posterior mapeo de datos a dichas unidades de ejecución.

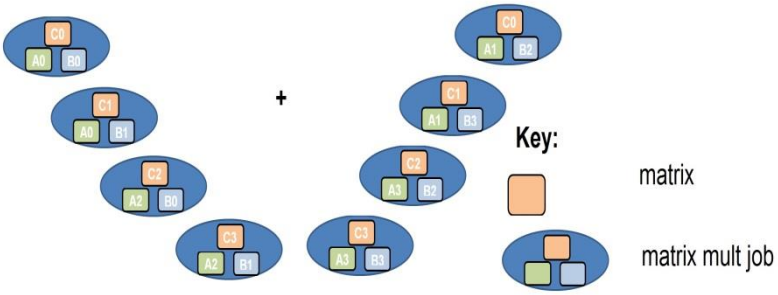
La implementación original Satin se presenta en un algoritmo Divide and Conquer. Este particiona el dataset en submatrices cuadradas y calcula los productos en el siguiente orden:

```
C01 =A0*B0;
C11 =A0*B1;
C21 =A2*B0;
C31= A2*B1;

C02= A1*B2;
C12= A1*B3;
C22= A3*B2;
C32= A3*B3;
```

En el backtracking del algoritmo, el tamaño de la matriz se duplica.

```
C0=C01+C02;
C1=C11+C12;
C2=C21+C22;
C3=C31+C32;
```



Si bien los algoritmos Divide and Conquer resultan en códigos legibles y/o elegantes, la ventaja principal es que dividen un problema es partes más pequeñas, así mismo el dataset de entrada. La mayor computación de estos algoritmos se realiza en los casos triviales, u hojas del árbol implícito que genera su ejecución; por ende, el hecho de mantener un dataset dividido facilita el mecanismo de caché acelerando dicha computación. En contraste, la segunda alternativa (Jobs que ejecutan el bucle **for int k**…..) debe acceder tanto a la columna como fila k, en el caso de matrices grandes no solo es probable que no quepan en memoria cache sino que el almacenamiento y ubicación de los vectores puede favorecer el detrimento de performance debido a fallos de página (particularmente cuando se accede a los datos por columnas, B [k, j]). Adicionalmente, la implementación Divide and Conquer propone que los factores izquierdos de la multiplicación sean utilizados 2 veces, favoreciendo aún más el mecanismo de cacheo.

Por otro lado, el número de Jobs creados en cada llamado recursivo y el tamaño de los datos mapeados a los trabajos es constante y equitativo lo que resulta beneficioso respecto a la escalabilidad y balanceo de carga de la ejecución de la aplicación.

Estas razones constituyen una guía para gridificar una versión similar a la original Satin utilizando GridGain.

2. Tiempo de ejecución teórico

Con la implementación del algoritmo paralelo distribuido es natural preguntar cuál es la función que describe su tiempo de ejecución.

Para este cálculo, primero derivaremos una expresión en notación **Big O** y luego añadiremos el overhead que se produce en un ambiente distribuido.

Cuando la computación se realiza sobre un Job; es decir, en forma secuencial, los parámetros que guían su ejecución son: depth (niveles de recursividad) y loop (dimensión de la matriz). Las submatrices se generan dinámicamente, en lugar de llegar a un caso base por reducción del dataset de entrada, "la división" estará guiada por el parámetro depth. Cuando se llega a un caso trivial, ejecutamos una multiplicación de complejidad temporal típica. Entonces definimos una función por partes:

$$T(depth,loop) \leq \begin{cases} loop^3 & \text{if } depth = 0 \\ 4T(depth-1,loop) + c_0 & \text{if } depth \geq 1 \end{cases}$$

1º call $T(depth, loop) \leq 4T(depth-1, loop)+ c_0$

2º call $T(depth, loop) \leq 4[4T(depth-1-1, loop)+c_0] + c_0 \leq 4^2T(depth-2,loop)+4c_0 + c_0$

3º call $T(depth, loop) \leq 4[4^2T(depth-3, loop)+4c_0+c_0]+c_0 \leq 4^3T(depth-3,loop)+ 4^2c_0 + 4c_0 + c_0$

I-th call, $T(depth, loop) \leq 4^iT(depth-I ,loop) + \sum^{i-1} 4^kc_0$

and by the most significant term

. ˙ . depth – i=0 ➔ depth=I ➔ T(depth, loop) ∈ $\mathbf{O(4^{depth}*loop^3)}$

Para añadir el overhead que implica un ambiente distribuido, definimos los siguientes parámetros:

- **task**: cantidad de niveles del quadtree que afecta al número de Jobs a crear.
- **depth**: niveles del quadtree útiles para ejecución secuencial por cada Job child.
- **β**: ancho de banda del canal que conecta los nodos del grid.
- **ʎ**: tiempo para realizar una suma entre dos submatrices que forman un componente producto.

Considerando que al terminar las multiplicaciones de cada Job child los resultados deben sumarse por pares para completar el producto matricial, la matriz producto final se duplicará en dimensión por cada nivel que se asciende en el árbol. Entonces la expresión para estimar el tiempo teórico sería:

$$\left(\frac{2^{task} * 2^{depth}}{ʎ} * loop^2 + \frac{2^{task} * 2^{depth}}{\beta} * loop^2 \right) + 4^{depth} * loop^3$$

3. Aspectos de implementación

GridGain presenta un modelo de ejecución preferentemente map reduce; Satin, Divide and Conquer. Para adaptar la solución en GridGain, en tiempo de ejecución, cada nodo del grid puede ejecutar un Task, Job-Task o en un caso trivial, un Job. En tiempo de ejecución se tiene como resultado una red virtual con topología en árbol, donde cada nodo hijo calcula un componente $C_{i,j}$ de la relación de multiplicación $Ci, j = C_{i,j} + A_{i,k} + B_{k,i}$. Cuando dicho componente es de un tamaño que merece descomposición, el concepto se aplica recursivamente. Específicamente, Divide and Conquer en GridGain puede implementarse con GridTaskFuture; y para permitir ejecuciones asíncronas de Job childs, es ventajoso suspender la master task (método holdcc), lo que sería un nodo padre hasta que cada Job child envíe un resultado parcial del producto. En vistas de optimizar el uso de memoria y procurar el cómputo de datasets más grandes, la master task no cachea o mantiene todos los resultados parciales hasta completarse, sino más bien realiza **reducciones** en la medida que arriban (GridTaskNoResultCache). La master task, ahora no retorna un resultado completo sino hasta que todos sus Job childs terminen satisfactoriamente (GridJobPolicy.Wait).

La disponibilidad de la aplicación también se tuvo en cuenta: ante un "crash" de la JVM o el rechazo de Jobs por parte de algún dispara la reprogramación de ejecución del Job involucrado en un nodo aleatorio del grid (GridJobPolicy.FailOver).

4. Performance de ejecución

A continuación se presenta una tabla con benchmarks obtenidos sobre diferentes parámetros de ejecución. Las marcas corresponden a la convergencia de tiempo de ejecución o en su defecto al promedio de 100 iteraciones.

Las ejecuciones se llevaron a cabo en una laptop con procesador Intel x86-64 gama i5 450m de 2.60 GHz y 4Gb de memoria RAM, ejecutando un único nodo GridGain. El sistema operativo subyacente es Linux Arch.

Tamaño de la matriz	Jobs	Tamaño de partición	Niveles de recursividad	Tiempo de cálculo en segundos	Tamaño de la matriz	Jobs	Tamaño de partición	Niveles de recursividad	Tiempo de cálculo en segundos
8192x8192	8	512x512	4	533.713	8192x8192	8	1024x1024	3	610.695
8192x8192	8	2048x2048	2	642.59	8192x8192	8	4096x4096	1	888.636
8192 x8192	8	128x128	6	321.593	8192x8192	8	256x256	5	451.049
8192x8192	64	128 x128	5	374.395	8192x8192	64	256x256	4	487.899
8192x8192	64	512 x512	3	596.395	8192x8192	64	64x64	6	293.393
8192x8192	64	32 x32	7	387.567	4096x4096	8	32x32	7	37.034
4096x4096	8	64 x64	6	39.405	4096x4096	8	128 x128	5	43.365
4096x4096	8	256 x256	4	54.219	4096x4096	8	512 x512	3	67.543
4096x4096	8	1024x1024	2	78.591	4096x4096	8	2048 x2048	1	83.661
4096x4096	64	32 x32	6	49.501	4096x4096	64	64 x64	5	48.587
4096x4096	64	128 x128	4	46.846	4096x4096	64	256x256	3	57.963
4096x4096	64	512 x512	2	69.99	4096x4096	64	1024x1024	1	82.2
2048x2048	8	32 x32	6	4.629	2048x2048	8	64 x64	5	4.502
2048x2048	8	128 x128	4	4.532	2048x2048	8	256 x256	3	5.625
2048x2048	8	512 x512	2	7.057	2048x2048	8	1024x1024	1	9.286
2048x2048	64	32 x32	5	5.11	2048x2048	64	64 x64	4	4.992
2048x2048	64	128 x128	3	5.098	2048x2048	64	256 x256	2	6.574
2048x2048	64	512 x512	1	7.672	2048x2048	64	1024 x1024	0	9.972
1024x1024	8	32 x32	5	0.6	1024x1024	8	64 x64	4	0.571
1024x1024	8	128 x128	3	0.565	1024x1024	8	256 x256	2	0.702
1024x1024	8	512 x512	1	0.889	1024x1024	64	512 x512	0	1.05
1024x1024	64	256 x256	1	0.709	1024x1024	64	128 x128	2	0.808
1024x1024	64	64 x64	3	0.777	1024x1024	64	32 x32	4	0.851
1024x1024	64	16x16	5	0.905	8192x8192	1	4096x4096	1	1619.375
8192x8192	1	2048 x2048	2	1996.578	8192x8192	1	1024x1024	3	1659.952
8192x8192	1	512 x512	4	861.088	8192x8192	1	256x256	5	799.701
8192x8192	1	128 x128	6	736.481	8192x8192	1	64 x64	6	786.846
4096x4096	1	2048 x2048	1	174.536	4096x4096	1	1024x1024	2	149.194
4096x4096	1	512 x512	3	95.89	4096x4096	1	256 x256	4	89.182
4096x4096	1	128 x128	5	68.139	4096x4096	1	64 x64	6	78.538
4096x4096	1	32 x32	7	76.305	2048x2048	1	1024 x1024	1	19.916
2048x2048	1	512 x512	2	14.901	2048x2048	1	256 x256	3	12.914
2048x2048	1	128 x128	4	8.899	2048x2048	1	64 x64	5	9.629
2048x2048)	1	32 x32	6	9.197	1024x1024	1	512 x512	5	1.834
1024x1024	1	256 x256	2	1.689	1024x1024	1	128 x128	3	1.17

Notas.

- Los tamaños de las matrices y sus particiones son potencias de 2 solo por comodidad. Los cálculos de la versión gridificada son exclusivos para valores de dimensión par. Ej.: 20x20.
- La columna Jobs con valor 1 se refiere a la versión de multiplicación de matrices secuencial.
- La tabla exhibe solo los tamaños de matrices para los cuales el tiempo de ejecución de la versión secuencial es supera los tiempos de ejecución de la versión gridificada, lo cual se da cuando las matrices son de menor tamaño (overhead de tiempo teórico de la formula anterior).

A partir de estas marcas, podemos apreciar una relación entre el número de Jobs, niveles de recursividad y tamaño de datos. En general, vemos que para un mismo tamaño de datos de entrada el tiempo de ejecución mejora hasta un límite según el nivel de recursividad. También, para un mismo tamaño de datos y niveles de recursividad, al aumentar exponencialmente el número de Jobs sobre un único nodo, (y sobrepasando el paralelismo para resolver el problema, particularmente ≥ 64) aumenta el tiempo de ejecución debido al tiempo de setup, comunicación, etcétera, que demanda su gestión.

En conclusión, tener mayor cantidad de Jobs permite la escalabilidad de la aplicación y una mayor disponibilidad ya que es más rápido migrar Jobs fallidos si estos son pequeños, pero estas ventajas y mejores tiempos de ejecución sólo se pueden experimentar cuando se dispone de más nodos; intuitivamente, un Job por nodo lograría la más alta performance independientemente de los niveles de recursividad. Por lo tanto, la ejecución en un grid de datos muy grandes tiene mejor rendimiento que una ejecución local-secuencial.

5. Conclusión.

Para el desarrollo de la aplicación fue necesario tener una idea del framework Satin, con esto podemos resumir algunas diferencias respecto de GridGain: Satin provee pocos métodos para desarrollar una aplicación grid-enabled, lo cual promueve su uso rápido; y se adapta directamente algoritmos Divide and Conquer. En GridGain pueden implementarse algoritmos Divide and Conquer, aunque es más directo implementar algoritmos map reduce; además, GridGain tiene un api aparentemente más variado y personalizable, lo cual requiere más estudio para aprovechar su uso: balanceo de cargas, colocalización de datos y computación, estructuras de datos distribuidas, etc.

El uso de estas herramientas deja una clara idea de las mejoras o el resurgimiento que las aplicaciones secuenciales pueden obtener al ejecutarse en un ambiente distribuido (aún más y nuevos servicios contemplando su integración). Por último, es fácil imaginar el desafío que representa desarrollar una herramienta de gridificación automática, o un middleware con soportes cada vez más completo, transparente, fácil y necesarios que cubran las demandas actuales como por ej.: disponer de memoria virtual compartida distribuida, sincronización de los datos-objetos distribuidos, tratamiento de errores de comunicación, heterogeneidad entre nodos, etc.

Finalmente, la enseñanza de esta experiencia deja una directriz útil, sobre los pasos a seguir para desarrollar una aplicación distribuida, y los requerimientos no funcionales que deben destacarse.

6. Apéndice

a. Posibles mejoras.

- Es sencillo incorporar lógica previa al particionamiento de datos y ejecución en el grid que determine una aproximación a la mejor configuración entre el número de Jobs, niveles de recursividad y tamaño de las particiones de datos para alcanzar un máximo rendimiento.
- Dado que la aplicación se ejecutó en una computadora multinúcleo, no fue necesario determinar una política de balanceo de cargas. Para sacar provecho de un clúster heterogéneo, se puede incorporar programáticamente un método que chequee las capacidades y el estado de carga de un nodo al momento de mapear Jobs.

b. Como ejecutar la aplicación.

La aplicación recibe tres parámetros [task rec loop]

- task: número de thread-Jobs que se crean para la multiplicación.
- rec: niveles de recursividad que cada thread-Job realizará para su computación**.**
- loop: tamaño de las particiones que componen la matriz. El tamaño final será: $\frac{2^{task}*2^{depth}}{} * loop$

Descargar GridGain desde http://www.gridgain.com/, descomprimir en el directorio deseado.
Establecer la variable de entorno **GRIDGAIN_HOME** apuntando al directorio **/distro** del directorio.
Invocar desde el shell, tipee sh runmmult.sh con los parámetros task, rec y loop. Algo así

```
sh runmmult.sh 2 2 128
```

Nota: la opción **–c** permite construir el ejecutable. Ej. sh runmmult.sh **–c** 2 2 128

c. Dificultades de desarrollo y correcciones pendientes.

Al momento de chequear el resultado final de la multiplicación, particularmente para matrices grandes, en ocasiones se devolvían componentes nulos, es decir los Jobs retornaban matrices ralas, donde el resto de las componentes eran datos coherentes. La falta de certeza se refleja en método check en la clase Leaf: **la única** modificación, (que **no cambia** de funcionalidad en el código original) es que retorna siempre true. Independientemente de cualquier error, la aplicación comienza y termina bien.