Evaluación de compiladores usados para desarrollo de algoritmos en el cómputo de autómatas celulares

Abdiel Emilio Cáceres González *
Universidad Juárez Autónoma de Tabasco, DACB
Carr. Cunduacán-Jalpa Km 1, Cunduacán Tabasco, México
A.P. 24 C.P. 86690. Tel.(+52)914 336-0928
Rubén Armando González García †
Universidad Juárez Autónoma de Tabasco, DACB
Carr. Cunduacán-Jalpa Km 1, Cunduacán Tabasco, México
A.P. 24 C.P. 86690. Tel.(+52)914 336-0928

El software que se utiliza para el cálculo de las evoluciones de los autómatas celulares, requiere de una gran cantidad de cálculos, razón por la que se han utilizado alternativas como *clusters* o *grid computing*. Sin embargo, ante la falta de recursos computacionales, se debe optimizar los algoritmos para reducir tiempo de ejecución; aún mas, es necesario elegir un compilador que garantice el mejor aprovechamiento de los recursos del sistema. En este trabajo se proporciona un método para elegir un compilador para crear programas útiles en el estudio de los autómatas celulares. Se muestra los algoritmos que son comparados utilizando varios compiladores de uso común y al final se describe las conclusiones de este estudio.

Software that are used for cellular automaton evolutions, require a great amount of calculations, for this reason clusters or grid-computing has been used as alternative. Nevertheless, because of the lack of computational resources, it is due to optimize algorithms to reduce runtime; even more, it is necessary to choose a compiler who guarantees the best advantage of the resources of the system. In this work a method is provided to choose a compiler to create useful programs in the study of the cellular automata. It shown algorithms that are compared using several compilers of common use and in the end it describes the conclusions of this study.

Palabras clave: Tiempo de ejecución, Medida de la eficiencia, Autómatas celulares.
Keywords: Execution time , Efficiency measure, Cellular automaton.

1. Introducción

Medir la eficiencia en la ejecución de algoritmos, es una práctica necesaria para estimar los costos computacionales en el procesamiento de datos. En particular, nos interesa saber el tiempo tardará la computadora en procesar algoritmos de autómatas celulares (AC).

Los AC son sistemas dinámicos discretos que se componen de un conjunto ordenado de elementos llamados células; ese conjunto se conoce como espacio celular. Cada célula puede tener solamente uno de un conjunto finito de estados discretos. Las células de todo el espacio celular actualizan su estado de manera síncrona. El tiempo avanza progresivamente de manera discreta también.

A la célula que está siendo procesada para calcular su nuevo estado en el siguiente

*abdiel.caceres@dacb.ujat.mx
†ruben.gonzalez@basicas.ujat.mx

paso de tiempo se le llama célula en transición. Cada célula del espacio celular tiene asociado un subconjunto de células llamado la vecindad de la célula. En principio, las vecindades son de la misma forma y tamaño para todo el espacio celular. El nuevo estado de la célula en transición es determinado por su estado actual y por el estado que tienen las células que pertenecen a su vecindad. Para un estudio más profundo del tema de autómatas celulares, véanse las referencias [3, 4].

Los autómatas celulares tienen diversas aplicaciones como en las citadas en [5].

Dependiendo de su configuración específica, el sistema dinámico descrito anteriormente puede requerir una gran cantidad de tiempo de procesamiento en las simulaciones por computadora.

El número de operaciones que una computadora secuencial debe hacer para calcular el estado del espacio celular en todo el tiempo de evolución se obtiene por la expresión $((2r+1)^d+2)nt$, donde r es el radio de vecindad, d es la dimensión, n el tamaño del espacio y t el número de evoluciones.

Debido a que no siempre es posible tener diferentes configuraciones de hardware ideales para diferentes configuraciones de autómatas celulares, es necesario hacer estudios de eficiencia sobre los algoritmos. Ademas hay que considerar que el desempeño real de un algoritmo en particular, cambia en dependencia del compilador usado.

En este trabajo se ha hecho una selección de lenguajes de programación y de diferentes compiladores en los cuales se ha implementado los mismos algoritmos y se ha hecho pruebas para determinar la mejor opción para desarrollar un laboratorio de autómatas celulares.

2. Descripción del problema

La productividad es una medida importante al evaluar el desempeño de un sistema multitarea, al atender una carga heterogénea de procesos por lotes. La latencia, en cambio, es ampliamente utilizada para medir la velocidad con que se ejecuta una aplicación en particular bajo condiciones específicas[6]. Por tal motivo, se utilizará la latencia como medida de desempeño.

Por otra parte, el tiempo que consume un programa para su ejecución depende de diversos factores:

1. La cantidad de datos
2. La complejidad del algoritmo
3. La herramienta de programación empleada
4. El tipo de operaciones a realizar
5. La plataforma de software
6. Las características del hardware
7. La carga de trabajo del sistema
8. La forma de medir el tiempo de ejecución

El problema general abordado en este estudio consiste en encontrar el conjunto de herramientas más eficientes para el procesamiento de las evoluciones de los autómatas celulares. Existen, sin embargo, algunas restricciones prácticas que afectan la selección de los parámetros a considerar en la evaluación del desempeño de una aplicación.

Las restricciones consideradas fueron, **el hardware disponible**: En este caso se utilizó computadoras personales con procesador x86 de 32 bits; **la carga de trabajo**: las pruebas se hicieron sobre un sistema con una carga promedio de trabajo inferior al 1%, esto significa que el sistema estuvo dedicado únicamente a la ejecución del programa de cómputo celular; **el sistema operativo**: En este caso se ejecutó las pruebas sobre Windows XP debido a que este es el sistema operativo instalado en los equipos utilizados y se prefirió no hacerles cambios en su configuración; **el tamaño de los datos**: el número de datos fue idéntico para todas las pruebas; **el algoritmo implementado**: en todos los casos se utilizó el mismo algoritmo de cómputo; **la forma de medir el tiempo de ejecución**: en este caso se medió la latencia de los programas.

De los 8 factores que afectan el tiempo de ejecución de una aplicación, 6 de ellas no se consideraron, porque su influencia en el resultado final es despreciable; quedando únicamente la herramienta de programación empleada y el tipo de operaciones a realizar como los dos parámetros variables para la realización de las pruebas.

Sin embargo, en muchos casos, el tipo de operaciones utilizadas, mantiene dependencia con la herramienta de programación . En todo caso, para un lenguaje dado, se utilizó como referencia la implementación que se ejecutó más rápidamente. Esto permitió medir el tiempo consumido por código más rápido que se puede generar con un compilador en particular, y al mismo tiempo impone una restricción sobre el tipo de operaciones utilizadas en la implementación del algoritmo, dejando como consecuencia, únicamente un grado de libertad: **la herramienta de programación**.

Por lo tanto, el problema se redujo a encontrar el compilador que genere el código que se ejecute en menos tiempo.

3. Diseño de experimentos

El siguiente paso consistió en seleccionar, en base a sus características, los lenguajes a ser probados. Las características deseables son:

- Portabilidad del código fuente
- Soporte para cómputo distribuido
- Manejo de grandes arreglos de números enteros ($>$ 1Gb)
- Velocidad de ejecución

La portabilidad del código fuente es la característica más importante, ya que permitirá recompilar el código y adecuarlo a las características de la plataforma de cómputo disponible. El soporte para cómputo distribuido es también muy importante, ya que este trabajo es la base para la implementación de una herramienta distribuida para

computación celular. La capacidad para manejar grandes arreglos de nmero enteros, es otra característica indispensable, ya que es la implementación más eficiente y natural de los espacios celulares. Finalmente, la velocidad de ejecución es también una característica importante, y es precisamente el objeto de estudio del presente trabajo.

Los lenguajes seleccionados fueron `C/C++` y `Java`, ya que ambos cumplen con las características deseadas, sobre todo en lo referente a la portabilidad. En particular, `Java` es reconocido por su portabilidad, aunque tiene la etiqueta de ser considerado como un lenguaje lento debido a que los programas compilados en `Java` se ejecutan sobre una máquina virtual. Sin embargo, diversos autores afirman que un programa compilado en `Java`, puede ejecutarse incluso más rápido que un programa compilado en `C` [1]. Estas afirmaciones están respaldadas por diversas pruebas de desempeo [1, 3] y por el hecho de que muchas de las máquinas virtuales de `Java` (`JVM`) son en realidad compiladores *just in time* (jit) que traducen, a código nativo, los programas generados por los compiladores `Java` [3].

En la tabla 1 se muestran los compiladores elegidos para las pruebas. Estos compiladores están basados en especificaciones estándares y cumplen con al menos una de estas características:

- Disponibilidad para diversas plataformas (al menos Windows y Unix).
- Reconocida eficiencia para generar código optimizado.
- Alta popularidad como herramienta de desarrollo.

C/C++	Java
GCC v.3.4.2	(GNU)Sun JDK v.1.5.0
Intel C++ Compiler v.9.0	IBM JDK v.1.4.2
C++ Builder v.6.0	Rockit JDK v.1.5.0
Visual C++ 2003	

Tabla 1. Compiladores elegidos para cada lenguaje

El siguiente paso fue elegir tipo de autómata celular y el algoritmo a utilizar. Los algoritmos de cómputo celular se caracterizan por ser intensivos en: el manejo de índices y el acceso a memoria principal. Con la finalidad de aumentar la cantidad de procesamiento y la carga de trabajo adicional relacionada con el manejo de índices, se decidió realizar la pruebas con autómatas celulares de 256 estados, de dimensión $d = 3$ y con radio de vecindad $r = 1$.

Por otra parte, se eligió un espacio celular suficientemente grande para no ser alojado en la memoria caché de la computadora, y que a la vez suficientemente pequeño para ser alojado completamente en memoria principal. El espacio celular fue de 300^3 celdas. Este espacio celular es representativo de pequeños problemas de cómputo celular, y es suficiente para los fines del presente trabajo.

El algoritmo básico utilizado durante las pruebas se muestra en el apéndice A. En este algoritmo, cada acceso a un elemento del arreglo `CEL`, requiere la conversión de los índices x, y, z, a una dirección lineal. Para calcular el siguiente estado de una célula, la función de evolución debe acceder a 29 células (27 que conforman la vecindad y 1 donde se almacena el siguiente estado temporal y 1 mas para recuperarlo). Para

calcular el siguiente estado del autómata celular completo, el programa debe acceder a un total de:

$$300^3 \times 29 + 300^3 \times 2 = 837 \text{ millones de células.}$$

Si se simplifica el cálculo de las direcciones de memoria a partir de los índices x, y, z se puede reducir significativamente el tiempo de ejecución de los programas.

En el algoritmo del apéndice B, la función de evolución calcula la vecindad de una célula a partir de la posición de la misma en el arreglo lineal, sumándole o restándole el número de celdas contenidas en 1 y 2 dimensiones (`MAX` y `MAX*MAX` respectivamente).

Para la realización de las pruebas, se consideró también varias implementaciones basadas en punteros , pero se descartó debido a los pobres resultados obtenidos en las pruebas preliminares.

4. Pruebas y resultados

Para realizar las pruebas de desempeño, se implementó ambos algoritmos (apéndices A y B) en los lenguajes `C` y `Java`, y se compiló cada uno de ellos utilizando los compiladores listados en la tabla 1. Como resultado de esta compilación se generó los 14 programas estudiados.

Los resultados de las pruebas con el compilador `Borland C++` fueron descartados debido a que no fue posible implementar el espacio celular como una matríz de dimensión $d = 3$ de números enteros; en su lugar se hizo una implementación con punteros, sin embargo, en las pruebas, dicha implementación mostró un desempeño sumamente bajo.

La tabla 2 muestra los tiempos de ejecución de todas las pruebas en 4 computadoras diferentes. En esta tabla, un tiempo menor indica un mejor desempeño. Se ha omitido intencionalmente los detalles de los equipos de cómputo sobre los cuales se realizó las pruebas, ya que no es el interés de este trabajo, comparar el desempeño de estas computadoras, sino de los compiladores.

En el caso del compilador de `Java` de Sun, para cada uno de los 2 algoritmos se realizó 4 variantes de la prueba; se probó las 2 versiones más recientes del compilador y de la máquina virtual: `J2SE` (`Sun JDK 1.5`) y `J2EE` (`Sun JSDK 1.4`). A su vez, para cada versión se ejecutó la JVM en modo client y en modo server . En total se realizó 8 pruebas con los compiladores de Sun, en cada procesador.

Por razones de espacio y de legibilidad, en la tabla 2 se omitió los resultados de las ejecuciones en el procesador Turion 64, ya que estos son muy similares a los obtenidos con procesador el Athlon 64. Además en el caso de las pruebas p1_sun y p2_sun, la figura 3 solamente muestra los mejores tiempos de ejecución, sin especificar si corresponden a la versión 1.5 o 1.4, y a los modos de ejecución `client` o `server`. También se omitió los resultados de las pruebas p1_vc_intel y p2_vc_intel, ya que estos son casi idénticos los de las pruebas p1_intel y p2_intel, respectivamente.

La tabla 3 muestra los resultados normalizados de las pruebas, con respecto al menor tiempo de ejecución para cada procesador. Un número mayor representa un mejor desempeño. Estos resultados permiten comparar el desempeño de un compilador con respecto a los otros, así como de las implementaciones p1 y p2, pero no

	Tiempo de Ejecución en segundos (un número menor indica mejor desempeño)			
Programa	Celeron 1.7 GHz	Pentium 4 HT 3.0 GHz	Turion 64 2.0 GHz	Athlon 64 2.0 GHz
p1_gcc	20	15	9	9
p1_intel	25	16	20	20
p1_vc	30	18	18	18
p1_vc_intel	25	16	21	21
p1_sun(1.5Cli)	104	69	104	103
p1_sun(1.5Ser)	81	60	77	77
p1_sun(1.4Cli)	104	69	104	103
p1_sun(1.4Ser)	81	61	72	72
p1_ibm(1.4)	60	36	42	42
p1_rockit(1.5)	175	114	109	108
p2_gcc	21	16	11	10
p2_intel	25	16	21	20
p2_vc	30	18	22	21
p2_vc_intel	26	16	21	20
p2_sun(1.5Cli)	59	36	50	50
p2_sun(1.5Ser)	65	56	35	35
p2_sun(1.4Cli)	60	37	49	49
p2_sun(1.4Ser)	55	46	30	30
p2_ibm(1.4)	31	20	21	21
p2_rockit(1.5)	100	59	79	79

Tabla 2. Tiempo de ejecución de las pruebas de desempeño de los compiladores.

permite comparar entre si el desempeño de los diferentes procesadores.

En la tabla 3 se puede ver que para cada procesador, los mejores resultados se obtuvieron con la prueba p1_gcc, seguidos en segundo lugar por la prueba p2_gcc. En general, los compiladores de Intel y de Microsoft (Intel y VC) tuvieron un desempeño similar. Mientras que las implementaciones en `Java` tuvieron un desempeño significativamente menor a las implementaciones en `C`. Cabe destacar el notable desempeño del `IBM JDK 1.4` en la prueba p2_ibm, que obtuvo un desempeño muy similar al de los compiladores de Intel y de Microsoft, aun a pesar de que `Java` se ejecuta sobre una máquina virtual.

Los compiladores `Intel` y `Visual C` mostraron un mejor desempeño en los procesadores `Intel` (Celeron y Pentium 4 HT) que en los procesadores `AMD` (Turio 64 y Athlon 64).

En las pruebas en `Java`, se observó una mejora significativa en las pruebas p2, con respecto a las pruebas p1. Sin duda alguna, esto se debe a que las pruebas p2 implementan arreglos unidimensionales (se maneja un sólo índice), mientras que las prueba p1 implementan arreglos tridimensionales (se manejan tres índices), lo cual genera trabajo adicional en la `JVM` debido a que `Java` realiza la verificación de índices en tiempo de ejecución.

En los diversos compiladores de `C`, no hubo diferencias significativas entre las prue-

	Desempeño de los compiladores en diferentes procesadores (un número mayor indica mejor desempeño)			
Programa	Celeron 1.7 GHz	Pentium 4 HT 3.0 GHz	Turion 64 2.0 GHz	Athlon 64 2.0 GHz
p1_gcc	100	100	100	100
p1_intel	80	94	45	45
p1_vc	67	83	50	50
p1_vc_intel	80	94	43	43
p1_sun(1.5Cli)	19	22	9	9
p1_sun(1.5Ser)	25	25	12	12
p1_sun(1.4Cli)	19	22	9	9
p1_sun(1.4Ser)	25	25	13	13
p1_ibm(1.4)	33	42	21	21
p1_rockit(1.5)	11	13	8	8
p2_gcc	95	94	82	90
p2_intel	80	94	43	45
p2_vc	67	83	41	43
p2_vc_intel	77	94	43	45
p2_sun(1.5Cli)	34	42	18	18
p2_sun(1.5Ser)	31	27	26	26
p2_sun(1.4Cli)	33	41	18	18
p2_sun(1.4Ser)	36	33	30	30
p2_ibm(1.4)	65	75	43	43
p2_rockit(1.5)	20	25	11	11

Tabla 3. Resultados normalizados de las pruebas de desempeño.

bas p1 y p2, debido a que `C` no realiza verificaciones de índices en tiempo de ejecución.

5. Conclusion

Muchas veces nos preocupamos por seleccionar el equipo de cómputo más poderoso que podamos adquirir con el presupuesto disponible, sin embargo pocas veces tenemos presente que la elección de un compilador adecuado, puede ser más significativa cuando de desempeño se trata.

Las pruebas demostraron que el algoritmo del apéndice A, implementado en el compilador `gcc 3.4.2` generó código más eficiente que cualquier otro compilador, al obtener los mejores resultados, en cualquiera de los procesadores.

Los progamas generados por el compilador Intel, mostraron un desempeño aceptable al ejecutarlos sobre procesadores Intel, especialmente en el Pentium 4; sin embargo al ejecutarse sobre los procesadores AMD, los resultados fueron sorprendentemente decepcionantes.

En los programas implementados en C, el algoritmo 1 fue al menos tan eficiente como el algoritmo del apéndice B, mientras que en los programas implementados en `Java`, el algoritmo mostrado en el apéndice B fue muy superior al algoritmo que se

puede apreciar en el apéndice A.

La elección del compilador más adecuado para desarrollar una aplicación puede ser una tarea más compleja de lo que parece, sobre todo cuando el desempeño de la aplicación es un factor importante.

Apéndice

A. Algoritmo 1

```
//declaracin del espacio celular
//y de un espacio celular temporal

MAX=300;
CEL [MAX][MAX][MAX];
TMP [MAX][MAX][MAX];

Inicio()
{
//<inicia cronmetro>
//Configuracin incial del espacio
//celular
  for (x=0; x<MAX; x++)
    for (y=0; y<MAX; y++)
      for (z=0; z<MAX; z++)
        CEL[x][y][z]= random(256)


//Calculamos solamente 20 evoluciones
  for(i=0; i<20; i++)
  {
    //Calculamos 1 evolucin a la vez
    for (x=0; x<MAX; x++)
      for (y=0; y<MAX; y++)
        for (z=0; z<MAX; z++)
          TMP[x][y][z]=
          FuncionDeEvolucion(CEL,x,y,z);


    //Copiar TMP a CEL
    for (x=0; x<MAX; x++)
      for (y=0; y<MAX; y++)
        for (z=0; z<MAX; z++)
          CEL[x][y][z] = TMP[x][y][z];
  }
//  <fin del cronmetro>
```

B. Algoritmo 2

```
//declaracin del espacio celular
//y de un espacio celular temporal

MAX=300;
MAXMAX=300*300;
MAXMAXMAX=300*300*300;

CEL [MAX*MAX*MAX];
TMP [MAX*MAX*MAX];

Inicio()
{
//  <inicia cronmetro>

//Configuracin incial del espacio celular
  for (i=0; x<MAXMAXMAX; x++)
    CEL[i]= random(256;

//Calculamos solamente 20 evoluciones
  for(i=0; i<20; i++)
  {  //Calculamos 1 evolucin a la vez
    for (x=0; x<MAX; x++)
      for (y=0; y<MAX; y++)
        for (z=0; z<MAX; z++)
        {
          i = x*MAXMAX + y*MAX + z;
          TMP[i]=
          FuncDeEvolucion(CEL,i,MAXMAX,MAX);
        }


    //Copiar TMP a CEL
    for (i=0; x<MAXMAXMAX; x++)
      CEL[i]= random(256);
  }

//  <fin del cronmetro>
```

Referencias

[1] Kano.net. *"The Java is Faster than C++ and C++ Sucks Unbiased Benchmark"*. Documento en línea, disponible en Internet en la dirección URL http://www.kano.net/javabench/. Visitado el 12 de mayo de 2006.

[2] Robert C. Martin. *"Java and C++: A critical comparison"*. Documento portable en formato PDF, disponible en Internet con el autor, o bien, en la dirección URL http://www.objectmentor.com/resources/articles/javacpp.pdf, creado el 9 de marzo de 1997 y revisado el 15 de mayo de 2006

[3] David J. Eck, Tutorial en linea, disponible en la dirección `http://godel.hws.edu/xJava/CA/CA.html`, Department of Mathematics and Computer Science Hobart and William Smith Colleges, Geneva, NY 14456, Contacto por correo electrónico con el autor en eck@hws.edu.

[4] Harold V. McIntosh. *Linear Cellular Autómata*. Documento disponible en formato PDF y HTML en la dirección `http://delta.cs.cinvestav.mx/~mcintosh/oldweb/pautomata.html`. Consultado en junio 2006

[5] Anonymous. "Cellular Automata Algorithms". Documento disponible en formato HTML en la dirección `http://www.hermetic.ch/pca/algorithms.htm`. Se describen 5 algoritmos para crear autómatas celulares con diferentes aplicaciones, principalmente en las ciencias naturales. Consultado en junio 2006.

[6] Ur Es. *A Performance Evaluation of Cluster-based Architectures*, artículo disponible en formato PDF en la base de datos `citeseer.ist.psu.edu/46464.html`. Consultado el 3 de septiembre de 2006