Mejora del rendimiento de una aplicación mediante la determinación automática de las opciones óptimas de compilación

Luis Alberto Vivas Tejuelo[1], Jorge Calvo Zaragoza[1], Felipe Restrepo-Calle[1]
Sergio Cuenca-Asensi[1], Andrés Ortiz García[2], Antonio Martínez-Álvarez[1]

[1]Dep. Tecnología Informática y Computación. Universidad de Alicante

[2]Departamento de Ingeniería de Comunicaciones. Universidad de Málaga

{lavt, jcz4}@alu.ua.es, {frestrepo, sergio, amartinez}@dtic.ua.es, aortiz@ic.uma.es

Resumen

Los compiladores modernos presentan un gran número de opciones de compilación, que pueden alterar las características de los programas compilados. La selección de las opciones más ventajosas para mejorar un cierto objetivo (p.e. tiempo de ejecución, tamaño de código, uso de memoria, etc) requiere un conocimiento profundo de la arquitectura subyacente y del funcionamiento del compilador. En este trabajo se propone un algoritmo para la generación automática de los parámetros óptimos de compilación basado en algoritmos genéticos y el Modelo de Islas. También se presenta una implementación paralela basada en PVM (Parallel Virtual Machine). El algoritmo es independiente del compilador, fácilmente escalable sobre un sistema multicomputador y adaptable sobre distintos objetivos (tiempo de ejecución, memoria usada, ...). Como caso de estudio se ha utilizado GCC (GNU Compiler Collection), obteniéndose ganancias de tiempo en la ejecución del código de hasta el 33 % respecto de la mejor opción de compilación predefinida.

1. Introducción

Los compiladores actuales más usados, como GCC [1], Clang [?] e ICC [3], ofrecen una amplia gama de opciones de compilación. Algunas, además, conllevan modificadores, lo que incrementa las posibilidades y complica el proceso de selección. Ciertas combinaciones de opciones pueden llegar a degradar el rendimiento en algunos casos, o incluso afectar los resultados del programa.

Con miles de opciones a considerar, estimar por fuerza bruta cuál es la combinación óptima para compilar un código concreto es inviable en términos de tiempo. Por ello, recurrimos a una solución basada en un algoritmo genético implementado en C++ que proporciona una solución que tiende hacia la óptima en un tiempo razonable.

El resto del artículo está organizado de la siguiente manera. La sección 2 alude a diversos trabajos relacionados con el que aquí se presenta. La sección 3 describe la implementación del algoritmo genético y la codificación empleada. La sección 4 presenta la forma de paralelizar el problema mediante el Modelo de Islas. La sección 5 presenta los resultados experimentales obtenidos en pruebas de escalabilidad y de bondad de la solución. Por último, la sección 6 contiene la conclusión extraída y las líneas de desarrollo futuro.

2. Trabajos relacionados

Entre los esfuerzos previos para buscar las opciones óptimas de compilación siguiendo distintos objetivos, cabe destacar ACOVEA (Analysis of Compiler Options via Evolutionary Algorithm) [4], de Scott Robert Ladd. ACOVEA es un *framework* que determina las opciones a activar para programas en C/C++ mediante algoritmos genéticos, y es además extensible a otros lenguajes y compiladores.

Del mismo modo, la herramienta ESTO de IBM (Expert System for Tuning Optimizations) [5] automatiza la selección de opciones de compilación adecuadas gracias a un algoritmo genético.

Pinkers et al. [6] proponen un procedimiento iterativo que activa y desactiva opciones de compilación, basado en vectores ortogonales, que se emplean para el análisis estadístico de la información de ejecución.

PathOpt y PathOpt2 [7] son herramientas libres y gratuitas de la suite comercial de compilación EKOPath de PathScale que emplean ficheros XML en los que el usuario puede especificar las opciones a probar y las restricciones. La mejor combinación se obtiene iterativamente a partir de búsquedas exhaustivas, aleatorias o ramificadas.

En este contexto, el principal aporte de nuestro sistema es la paralelización de un algoritmo genético al estilo de ACOVEA, lo que permite aprovechar el potencial que ofrecen los sistemas multiprocesador y/o multicomputador.

3. Algoritmo genético

La elección de utilizar un algoritmo genético en lugar de otros algoritmos como *Simulated Annealing* surge de las correspondencia intuitiva y directa que existe entre los individuos y los genes con las cadenas y opciones de compilación, respectivamente, como veremos a lo largo de esta sección.

3.1. Conceptos básicos

Los algoritmos genéticos son un tipo de algoritmo de búsqueda basado en conceptos de la evolución biológica como la herencia, la mutación, el cruce y la selección [8] [9]. Son algoritmos estocásticos, ésto es, se basan en el azar y en funciones de probabilidad.

La idea fundamental es la generación aleatoria de un conjunto inicial compuesto de "individuos", llamado "población". Dichos individuos contienen una información asociada, o "genoma", que codifica una posible solución del problema. Dicha población va evolucionando, de forma que el destino de los genes depende de la aptitud del individuo. En cada iteración, se evalúa la aptitud de cada organismo de la población, es decir, la calidad de la solución, y se seleccionan individuos, se altera su información genética (mutación) y se recombinan para evolucionar la población. El proceso se repite hasta que se alcanza un criterio de parada (Fig. 1).

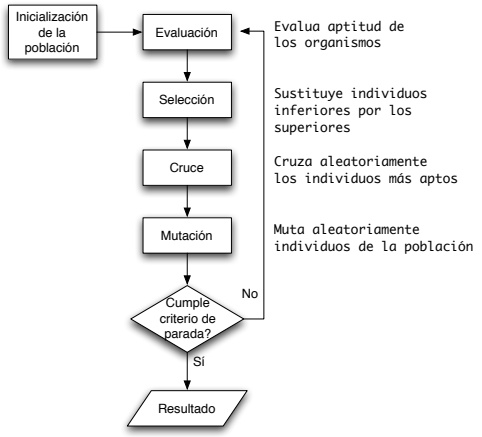

Figura 1: Etapas del algoritmo genético

3.2. Algoritmo genético para determinar opciones de compilación

En nuestro caso, se codifica cada opción de optimización como un gen. La calidad de la solución conformada por los genes de un individuo se evalua en la función de *fitness* en base al tiempo de ejecución del programa compilado. Las ejecuciones fallidas no se tienen en cuenta. Cada gen puede tener probabilidades de mutación y cruce distintas al resto.

Los operadores genéticos básicos implementados son la selección natural, el cruce y la mutación. En todos casos, la selección es de tipo directa. La selección natural se realiza sustituyendo la mitad inferior de la población por una copia de los individuos superiores; el tamaño de la población permanece constante. En cuánto al cruce, se seleccionan los organismos a pares: uno de la mitad superior y otro de la

inferior. El cruce implementado es de tipo uniforme: en función de la probabilidad de cruce de cada opción, se modifica el genoma del segundo con los genes del padre o la madre. Por último, respecto a la mutación, se deja sin alterar el primer cuarto de la población, el que conforma la "élite", y se modifica el resto en función de las probabilidades de mutación de sus genes.

Al mantener estable el tamaño de la población, los criterios de parada se limitan a los siguientes:

- Se alcanza el límite de generaciones: se ha alcanzado el límite de iteraciones definido por el usuario y el programa termina, devolviendo el mejor conjunto de opciones de compilación.
- Se alcanza el límite de generaciones sin mejora. Se almacena el mejor resultado hasta el momento y se compara tras cada iteración para ver si se ha superado. Si es el caso, se reinicia a 0 el contador de generaciones sin mejora. En caso negativo, se incrementa el contador y se comprueba si se ha alcanzado el límite, en cuyo caso, la ejecución termina y se devuelve el mejor resultado obtenido hasta ese momento.

3.3. Implementación

El conjunto de las opciones de compilación se codifica mediante una cadena de caracteres que se pasa por parámetro al compilador. La información de la cadena de compilación, así como las operaciones aplicables a los organismos y las constantes para el manejo de las opciones se encapsulan en una clase. A tal efecto, se han contemplado tres tipos de opciones: opciones con listas fijas de valores posibles (e.g. `mtune` en GCC), opciones parametrizadas (e.g. `finline-limit` en GCC) y *flags* (e.g. `ftree-vectorize` en GCC). Cada tipo de opción se almacena en una lista distinta. Asimismo, se incluye un atributo en el que se almacena el tiempo de la última ejecución del programa proporcionado con las opciones escogidas en el individuo, y que es el valor que señala la aptitud del individuo: a menor tiempo, mayor bondad.

Las probabilidades de mutación y cruce de cada opción se obtienen a partir de ficheros de entrada, y se emplean en funciones de probabilidad, donde se comparan con una tirada, esto es, un valor aleatorio entre 0 y 1, para determinar si se realizan las operaciones correspondientes o no.

Para converger hacia una solución aceptable con más rapidez, se admite la configuración de la población inicial parcial o totalmente. Esto permite acelerar el proceso añadiendo posibles opciones previamente contrastadas por el usuario, como por ejemplo, las opciones `-O` de GCC.

La población se almacena en un vector de individuos y las opciones de compilación se leen de un fichero de entrada y se almacenan en una estructura de datos diseñada al efecto. Para controlar el número de posibilidades de cada opción, almacenamos dicha información en un vector de enteros.

4. Paralelización

Para paralelizar el programa se ha optado por la librería PVM en su versión 3.5.2 [10]. Seguimos el enfoque del Modelo de Islas [11], ya que limita el uso del ancho de banda [12]: se dispone de varias instancias del algoritmo genético, cada una de ellas con su subpoblación de individuos aislada del resto. Cada genético genera su propia población inicial, la evalúa y la modifica mediante los operadores de cruce y mutación. Al cabo de un número determinado de generaciones, se comparten los mejores individuos de una población con otra mediante el operador de migración, siendo ambas determinadas aleatoriamente. Gracias a esto, las poblaciones tienden a verse menos afectadas por los máximos locales (convergencia hacia falsas soluciones). Este tipo de algoritmos genéticos se suele denominar "de grano grueso".

En nuestra implementación del operador migración, se traspasa el 25 % superior de la población, que sobrescribe el 25 % inferior de la población designada como receptora. El envío siempre pasa por el proceso maestro, que reenvía las cadenas al receptor (Fig. 2).

No se efectúan más operaciones al margen

Figura 2: Comunicación en las migraciones de individuos

del envío/recepción, ya que, por como se han dispuesto las operaciones, la siguiente iteración del receptor se encarga de evaluar los nuevos individuos y ordenarnos en función de su aptitud antes de realizar cualquier operación como la selección, cruce o mutación. Por lo tanto, el resultado de la ordenación determina su futuro antes de llegar a la modificación de la población.

La migración introduce una sobrecarga de comunicación, razón por la cual se ha reducido en la medida de lo posible la información a enviar. Con ese propósito, y para un manejo más sencillo de la información del individuo, se traducen las opciones elegidas en el genoma del individuo en forma de cadena de valores numéricos. Asimismo, se dispone de un método que traduce los valores de dicha cadena y obtiene la cadena de compilación. La cadena resultante se emplea en la función de prueba, o *fitness*, que compila el programa, lo ejecuta y devuelve el tiempo de ejecución, que almacenamos en el individuo correspondiente. El número de migraciones, en cualquier caso, está en manos del usuario, en base al número de etapas que éste determine.

Al final de la ejecución del programa, es decir, cuando se ha agotado el número de etapas con sus k generaciones cada una, se obtienen las mejores cadenas de compilación y sus tiempos, una por población, y se devuelve la mejor de todas. En la Fig. 3 se muestra la topología de la solución.

El proceso principal, Maestro, dispone de n genéticos, cada uno de ellos con su propia subpoblación de individuos. Cada genético dispone de m esclavos que se encargan de ejecutar el programa, recoger el tiempo y devolverlo. El tráfico resultante se ilustra en la Fig. 4.

Dado que todos los procesos se mantienen hasta el final de la ejecución, se envían mensa-

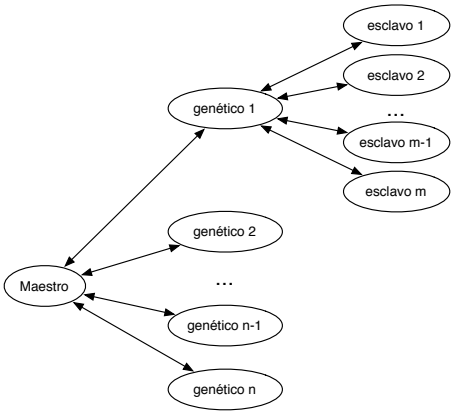

Figura 3: Árbol de procesos

jes de tipo señal para indicar el estado o el rol en la migración (*no implicado*, *envía* o *recibe*). Cada mensaje emplea una etiqueta distinta, y los posibles valores de las señales están definidos como constantes de tipo entero para su manejo.

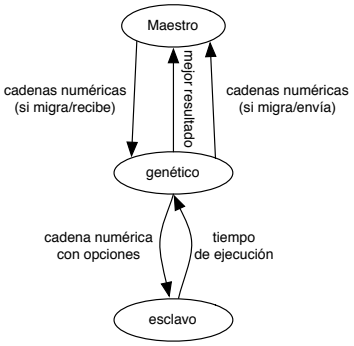

Figura 4: Comunicación entre los distintos tipos de procesos

En el caso de la implementación paralela del algoritmo genético, se ha modificado el criterio de parada del límite de generaciones sin mejora: dado que se dispone de más de una población, alcanzar el límite en una de ellas ya no finaliza el programa, pero si que finali-

za la etapa para la población en cuestión, es decir, ya no hay más iteraciones hasta que el proceso maestro inicie una nueva etapa.

4.1. Configurabilidad

El usuario puede indicar el número de etapas o periodos (tras las cuales se migran individuos), el número de generaciones por periodo (iteraciones de cada población), el límite de etapas sin mejora (tras el cual la población se mantiene pero finaliza la generación), el número de poblaciones, el tamaño de las mismas y el número de procesos esclavos de evaluación de los que dispone cada una para probar individuos.

Es conveniente equilibrar el número de etapas y las generaciones por etapa para evitar congestionar la red, especialmente si las poblaciones tienen muchos individuos.

5. Resultados experimentales

El objetivo es maximizar el rendimiento de un programa dado. Por consiguiente, la métrica empleada por los procesos de *fitness* es descendiente: cuánto más bajo es el tiempo, mejor es el rendimiento. En primer lugar se han obtenido resultados experimentales siguiendo ese criterio. A continuación se ha analizado el rendimiento del propio algoritmo genético en su versión multicomputador.

5.1. Comparación con las opciones predefinidas

Se ha empleado un código de prueba que calcula la transformada de *Wavelet*. El resultado se obtuvo tras una prueba de aproximadamente 2h 31m de duración, en una sola máquina con las siguientes características: *Intel Core 2 Duo* a 2.4 GHz y 4 GB de RAM bajo *Ubuntu* 9.10. Se ha empleado la versión 4.4.1 de GCC.

La cadena de compilación resultante fue:

```
g++ -O2 -march=core2 -mtune=nocona -mfpmath=sse
-finline-limit=680187 -falign-jumps -falign-loops
-fargument-noalias -fargument-noalias-anything
-fasynchronous-unwind-tables -fbranch-count-reg
-fbranch-target-load-optimize2 -fcaller-saves
-fcprop-registers -fdata-sections -fdelayed-branch
-fexpensive-optimizations -ffinite-math-only
-fgcse-sm -fguess-branch-probability
-fif-conversion -fif-conversion2 -finline-functions
-finline-functions-called-once -fipa-cp -fipa-pta
-fipa-pure-const -fipa-type-escape -fmerge-all-constants
-fmove-loop-invariants -fnon-call-exceptions
-foptimize-register-move -fpeel-loops -fpeephole
-fpeephole2 -frename-registers -frounding-math
-frtl-abstract-sequences -frtti -fsched2-use-traces
-fsection-anchors -fshort-enums -fstrict-aliasing
-fthread-jumps -ftoplevel-reorder -ftree-ch -ftree-dce
-ftree-dse -ftree-loop-im -ftree-loop-ivcanon
-ftree-salias -ftree-sink -ftree-store-ccp
-funit-at-a-time -funswitch-loops -funwind-tables
-fvariable-expansion-in-unroller -fweb -fwhole-program
wt.c
```

En el Cuadro 1 se presentan los resultados obtenidos mediante las opciones de optimización predefinidas de GCC y los obtenidos con la aplicación de la cadena de compilación mencionada. En la Fig. 5 se ilustra la mejora del tiempo de ejecución del programa frente a una compilación por defecto, con `-mtune=generic`.

Opción	Tiempo
Por defecto	3,48s
`-O1`	3,03s
`-O2`	2,94s
`-O3`	2,92s
`-Os`	3,7s
Genético	2,19

Cuadro 1: Comparación de tiempos

Como se puede apreciar, la mejora es notable respecto a las optimizaciones predefinidas. Frente a la configuración por defecto del compilador, el algoritmo obtiene un incremento del rendimiento cercano al 60 %, siendo casi del 70 % para la configuración `-Os`. En cuanto las configuraciones `-O1`, `-O2` y `-O3` el algoritmo alcanza un incremento del 33 % en el peor de los casos.

5.2. Estudio de la paralelización

Para estudiar la escalabilidad del algoritmo se ha empleado un código sencillo que realiza una serie de operaciones de *swap*, sumas y

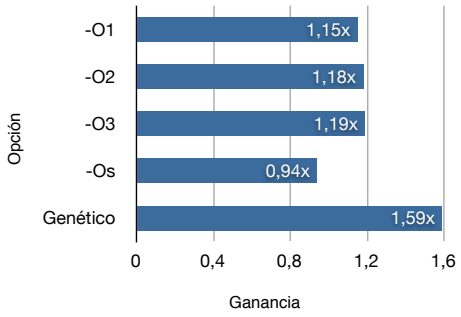

Figura 5: Mejora respecto a las opciones predefinidas

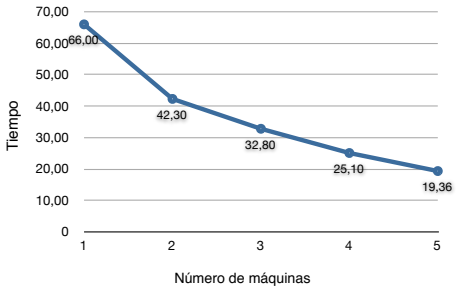

Figura 6: Progresión del tiempo respecto al número de máquinas

potencias con matrices cuadradas de tamaño definido en una constante.

En los experimentos se han empleado máquinas con idéntica configuración consistente en procesadores *Pentium Dual Core* y 2 GB de memoria RAM bajo *Ubuntu* 8.04, con la versión 4.2.4 de GCC, conectados mediante una red Fast Ethernet.

5.2.1. Prueba n°1

Para la prueba 1 se utilizó un tamaño de matrices de $500x500$. El número de períodos y generaciones por periodo fue de 4, mientras que el límite de etapas sin mejora se situó en 2. El número de poblaciones fue de 4, con 2 procesos de prueba por población, y con 8 individuos cada una.

En la Fig. 6 se muestran los tiempos obtenidos (en segundos) con diferente número de máquinas.

Se aprecia una ganancia de velocidad considerable respecto a la prueba en una sola máquina. Se obtienen ganancias de 1.56, 2.01 y 2.63 con dos, tres y cuatro máquinas respectivamente. Se puede observar que la repercursión de la adición de máquinas se mantiene en los casos contemplados.

5.2.2. Prueba n°2

En la segunda prueba se utilizó un tamaño de matrices de $1000x1000$. El número de períodos y generaciones por periodo se mantuvo en 4, mientras que el límite de etapas sin mejora se aumentó a 4. El número de poblaciones fue de 4, con 4 procesos de prueba por población, y con 16 individuos cada una.

Como se puede apreciar en la Fig. 7, la ganancia de velocidad es todavía superior al caso anterior y la repercusión de la adición de máquinas se mantiene. La razón de ésto está en la relación directa con la complejidad del programa proporcionado: la duración media de la ejecución del código es muy superior en este caso por el mayor tamaño de las matrices. El resultado es que la sobrecarga del algoritmo genético se desvanece y la relación del tiempo con el número de máquinas llega a ser supralineal.

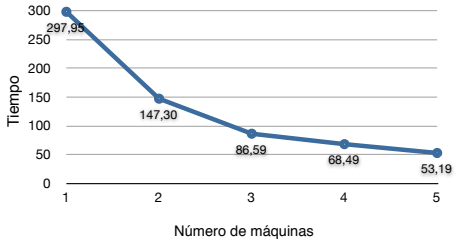

Figura 7: Progresión del tiempo respecto al número de máquinas

6. Conclusiones y trabajo futuro

Nuestra solución al problema de la selección de opciones óptimas de compilación se ha implementado mediante un algoritmo genético paralelizado con conceptos del Modelo de Islas. Considerando las limitaciones del entorno de pruebas, los resultados superan las expectativas iniciales. El algoritmo resultante obtiene cadenas de compilación cuyos resultados superan notablemente a las opciones predefinidas de GCC, con ganancias de hasta un 33 % respecto a la mejor opción de compilación predefinida. De igual modo, se ha comprobado que la versión paralela es perfectamente escalable en un entorno multicomputador.

La primera línea de continuación de este trabajo es la mejora de la eficacia y eficiencia del algoritmo mediante la localización de opciones incompatibles, evitando de esta forma repetir ejecuciones fallidas, reduciendo el dominio de búsqueda y disminuyendo la complejidad del problema. La siguiente mejora propuesta consiste en el estudio de la influencia de las opciones en los resultados y la detección de patrones a partir de esta información, que ayuden a dirigir la búsqueda y acelerar la convergencia hacia soluciones de calidad. Otra línea futura consiste en utilizar el mismo método descrito en el presente artículo para optimizar la compilación del código, orientado a mejorar las prestaciones en aplicaciones paralelas que presenten una especial dependencia con la localidad de caché.

Agradecimientos

El trabajo presentado aquí ha sido financiado por el proyecto de investigación "Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware" (GV/2009/098) (Generalitat Valenciana, España).

Referencias

[1] "GNU Compiler Collection (GCC)", http://gcc.gnu.org, 1987

[2] "CLang: a C language family frontend for LLVM", http://clang.llvm.org/

[3] "Intel C++ Compiler", http://software.intel.com/en-us/intel-compilers/

[4] Scott Robert Ladd, "ACOVEA", disponible en http://www.coyotegulch.com/products/acovea/index.html, 2006

[5] Guy Bashkansky, Yaakov Yaari, "Black Box Approach for Selecting Optimization Options Using Budget-Limited Genetic Algorithms", IBM Haifa Research Lab, 2007

[6] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, H. A. G. Wijshoff, "Statistical Selection of Compiler Options", en "Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '04)", IEEE Computer Society, 2004

[7] PathScale, "Automated Application Tuning", PathScale Inc., 2005

[8] Darrell Whitley, "A Genetic Algorithm Tutorial", Statistics and Computing Volume 4, pp. 65-85, Colorado State University, 1994. Disponible en http://www.cs.colostate.edu/~genitor/1994/tutorial.ps.gz

[9] S.N. Sivanadam, "Introduction to Genetic Algorithms", Springer, 2009

[10] "PVM: Parallel Virtual Machine", http://www.csm.ornl.gov/pvm/

[11] D. Whitley, S. Rana, and R. B. Heckendorn, " Island Model Genetic Algorithms and Linearly Separable Problems", In Evolutionary Computing, AISB Workshop pp. 109-125, Springer-Verlag, 1997

[12] Marco Antonio Castro Liera, "Algoritmos Genéticos Distribuidos", La Paz, B.C.S., 2007.