Green Code

Eficiencia energética en el código fuente para computación de alto rendimiento

Fundación COMPUTAEX
info@{computaex.es, cenits.es}
CénitS - Centro Extremeño de iNvestigación, Innovación Tecnológica y Supercomputación
Cáceres, Extremadura, España

***Abstract*—El consumo energético es un reto clave en HPC (High-Performance Computing, computación de alto rendimiento). Actualmente las optimizaciones para el ahorro energético referentes al software suelen implementarse en distintos niveles, siendo patente la falta de propuestas para que el usuario final escriba directamente código eficiente energéticamente. La presente propuesta pretende contribuir a la optimización de códigos fuente que permitan obtener rendimientos óptimos y eficiencias máximas en sus ejecuciones, demostrando la importancia de ciertas estrategias en la creación del código.**

I. Introducción

En los últimos años ha crecido la necesidad de replantear el impacto medioambiental producido por la tecnología, convirtiendo el ahorro energético en un asunto crítico, donde el derroche se produce tanto en componentes hardware como software. Las decisiones tomadas durante la fase de diseño del software tienen un impacto significativo sobre el consumo de energía de un sistema computacional. Por ello, el paradigma del Green Computing[1] aborda el ahorro energético mediante la aplicación de diferentes técnicas orientadas al software y al hardware.

Las optimizaciones para el ahorro energético referidas al hardware se logran a través del diseño de circuitos mediante la implementación de diversas técnicas. Respecto al software, las optimizaciones para el ahorro energético suelen implementarse en el sistema operativo, con técnicas de planificación que analizan los procesos activos en términos energéticos, y mediante la utilización de *Green Compilers* y análisis de códigos en tiempo de compilación. Del mismo modo, también es posible ahorrar energía durante las fases del ciclo de vida del software.

La supercomputación tiene planteados varios retos claves para alcanzar prestaciones *exascale*[2], identificando cuatro de ellos donde es necesario desarrollar tecnologías disruptivas a este fin: el consumo energético, la memoria, la concurrencia y la resiliencia. La presente propuesta pretende ser una contribución en el ámbito del consumo de energía.

II. Alcance y objetivos

Green-Code persigue un desarrollo de código fuente eficiente, partiendo de la propia eficiencia energética de los dispositivos tecnológicos sobre los que se ejecuta dicho código. Las investigaciones previas desarrolladas por la Fundación COMPUTAEX (Computación y Tecnologías Avanzadas de Extremadura) centraron el estudio de la eficiencia energética en tres campos: sistemas, redes y explotación de recursos medioambientales.

La principal diferencia de un código optimizado es su velocidad de ejecución. En un supercomputador esta velocidad es especialmente importante, ya que algunas ejecuciones pueden durar semanas o inclusos meses hasta obtener el resultado final. Al tratarse de periodos tan amplios, el tiempo de ejecución puede variar significativamente dependiendo de la eficiencia del código. Un código eficiente supondrá entonces no solo un menor tiempo de ejecución, sino un consumo energético mucho menor.

Se propone como objetivo principal la optimización, dentro del desarrollo de código para computadores de altas prestaciones, de códigos fuente de ámbito general, con el fin de obtener rendimientos óptimos y eficiencias máximas, teniendo en cuenta la correcta implementación de sus funcionalidades.

III. Eficiencia energética

Es necesario analizar, investigar y evaluar las posibilidades de mejora energética de los equipos en los diferentes niveles en los que la implementación del código pueda aportar eficiencia: usuario final (es patente la falta de aproximaciones y metodologías para el desarrollo de sistemas orientados a la computación de alto rendimiento considerando la eficiencia energética); sistema operativo y gestores de colas de los clústeres de cómputo; herramientas y librerías estándares de desarrollo; y comunicaciones.

La preocupación del sector TIC por el consumo energético y sus consecuencias medioambientales es patente. Actualmente, el gasto de los centros de datos y los equipos de comunicaciones está en torno al 30 % de la energía consumida en TI. Además, se estima que este gasto sea el que más se desarrolle en los próximos años, siendo cercano al 24 % con respecto a otros consumos en TI (ver Fig. 1).

Múltiples trabajos de investigación han mostrado mejoras en la aplicación de estrategias de eficiencia energética a ciertos algoritmos o a la asignación de recursos hardware a distintos procesos [1–3]. Otros trabajos basan su ahorro en mejorar los accesos a recursos de entrada/salida compartidos en el clúster de computación [4–6]. Cabe destacar también las mejoras que se pueden obtener al equilibrar las comunicaciones de red y elegir las mejores interconexiones [7, 8].

Por todo lo anterior, es preciso evaluar, analizar e investigar las posibles mejoras en la eficiencia energética de los equipos

[1]Green Computing (Green IT o Tecnologías Verdes) es el estudio y la práctica de tecnologías informáticas ecológicamente sostenibles, incluyendo el diseño, fabricación, uso y eliminación de software y hardware con eficiencia y eficacia con un mínimo o ningún impacto en el medio ambiente.

[2]La computación de Exascale es la relativa a aquellos sistemas computacionales capaces de llegar al menos a un exaFLOPS: 10^{18} FLOPS u operaciones de coma flotante por segundo.

TI mediante la mejora en los diferentes niveles en los que la programación del código pueda aportar eficiencia. Es decir, se busca añadir mecanismos para que los desarrolladores e investigadores puedan realizar códigos energéticamente eficientes sin empeorar las capacidades de cálculo que los centros de HPC pueden ofrecer.

IV. Eficiencia energética en el código

A. Estrategias eficientes en los compiladores

Los compiladores eficientes analizan los programas software en tiempo de ejecución y remodelan el código fuente mediante la aplicación de distintas técnicas durante la transformación de código. A continuación se muestran algunas de las técnicas que pueden ser aplicadas local, global o interproceduralmente [9, 10]:

- Omisión de operaciones de caché durante repeticiones innecesarias.
- Utilización de operandos de registro: las escrituras y lecturas en memoria tienen más coste que el uso de operandos de registro, los cuales presentan menos abstracción que los accesos a memoria y, por tanto, consumen menos energía.
- Agrupación de instrucciones: algunas arquitecturas permiten a un procesador ejecutar parejas o conjuntos de instrucciones en un único ciclo. Esto reduce el tiempo de ejecución del programa, con el consiguiente ahorro energético.
- Reordenación de instrucciones y direccionamiento de memoria.
- Utilización de bases de datos sobre costes energéticos de cada transacción/instrucción.
- Optimización de bucles.
- Gestión de energía dinámica: estableciendo la corriente del hardware en tiempo real, reduciendo la energía residual sin disminuir su rendimiento.
- Hibernación de recursos desocupados.
- Realización de tareas en la nube.
- Gestión de la recursividad: algunos compiladores realizan la conversión de ciertas recursividades en iteraciones, consiguiendo notables resultados.

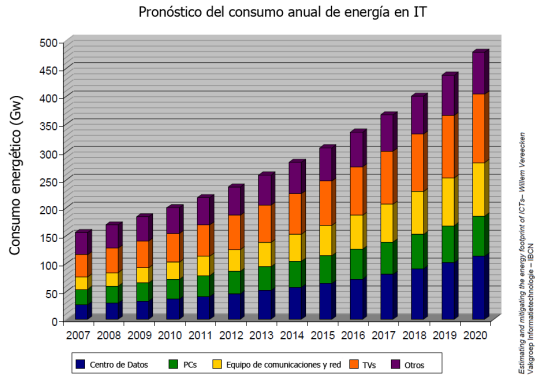

Fig. 1: Consumo anual de energía en IT estimado hasta 2020.

B. Estrategias eficientes en el ciclo de vida software

Se puede ahorrar energía durante el ciclo de vida software en fases como el análisis, el diseño o la propia implementación. A continuación se muestran distintas estrategias que se pueden seguir durante la implementación software [9]:

- Utilización de compiladores y entornos de desarrollo orientados al ahorro energético.
- Utilización de Computación GRID y Cloud: actualmente existen aplicaciones y hardware disponibles como servicios.
- Recursividad versus Iteración: el funcionamiento de la recursividad, basado en el uso de pilas, conduce a un mayor consumo de energía que si se utilizan iteraciones, evitando aquellas recursividades que sean innecesarias.
- Reducción del tiempo de ejecución: en general, cualquier estrategia que pueda reducir el tiempo de ejecución de un algoritmo puede ser útil para reducir el consumo energético [11]. Los algoritmos que tengan una complejidad lineal serán más eficaces energéticamente que aquellos con ecuaciones de complejidad exponencial. Por lo tanto, las técnicas que reducen la complejidad de los programas, deben ser utilizadas durante las fases de diseño e implementación.
- Uso de estructuras de datos eficientes.

C. Eficiencia de código en C y C++

1) Funciones virtuales: El uso de polimorfismo y funciones virtuales facilita las fases de prueba, depuración o mantenimiento del programa. Sin embargo, el uso de funciones virtuales afecta negativamente al rendimiento, no siendo tan eficiente como podría parecer. La configuración más eficiente para el lenguaje es que las funciones virtuales no sean opcionales y que éste no sea virtual.

2) Optimización de los valores de retorno: Es habitual que los métodos que devuelven un objeto se vean obligados a crear previamente un objeto de devolución. Sin embargo, utilizando la Optimización del Valor de Retorno (*Return Value Optimization*), el compilador detecta que no tiene más razón para crear el objeto que el propio hecho de tener que devolverlo, por lo que aprovecha la ventaja que supone construir el objeto directamente en la localización del valor de retorno externo a la función. Esta acción es, por tanto, significativamente más eficiente.

3) Excepciones: Las excepciones permiten afrontar errores inesperados en la ejecución de un programa, pero tienen un coste alto en términos de eficiencia. Se estima que el lanzamiento de una excepción es tres veces más costoso que un retorno normal. La manipulación de excepciones está diseñada para el procesamiento de errores, por lo que es aconsejable realizar un uso adecuado de ellas, si se desea obtener un rendimiento óptimo del código.

4) stdio vs iostream: Se recomienda utilizar la familia *cstdio* de funciones en lugar de la familia *iostream* cuando la velocidad de salida es un parámetro fundamental a tener en cuenta o, en el caso que nos ocupa, para mejorar la eficiencia energética.

5) Operadores de prefijo: Aquellos objetos que ofrecen el concepto de incremento y decremento, proveen a menudo de operadores ++ y - -. Existen dos tipos de incremento, de prefijo (++x) y de sufijo (x++). En el caso del prefijo, el valor se incrementa y el nuevo valor es devuelto. En el caso del sufijo, el valor se incrementa, pero es devuelto el valor antiguo . Implementar correctamente la opción de sufijo requiere guardar el valor original del objeto, es decir, crear un objeto temporal. La recomendación es clara: evitar operadores de sufijo.

6) Funciones inline: Las funciones *inline* proporcionan una mayor rapidez debido a que ahorran memoria al no hacer uso de la pila, con el consecuente ahorro de una llamada a la función y la devolución del valor de retorno. El problema es el aumento de espacio que eso conlleva. Por lo tanto, se debe usar *inline* en funciones que sean livianas y no aumenten demasiado el tamaño del ejecutable.

D. Optimización de código C y C++

A continuación se muestran otras técnicas que, además de incrementar la eficiencia del código, mejoran su legibilidad y facilitan su mantenimiento posterior [12].

1) Paso de argumentos por referencia: Pasar un parámetro por valor requiere que el objeto entero sea copiado, mientras que con el paso por referencia no se invoca al constructor de copia, aunque esto último implica penalizaciones cuando el objeto es usado dentro de la función. Incluso para objetos relativamente pequeños, la utilización del paso por valor puede suponer una penalización importante. Las diferencias entre una y otra técnica llegan a ser importantes según los parámetros utilizados.

2) Posponer la declaración de una variable: Declarar una variable puede resultar una operación costosa cuando el objeto posee un constructor o un destructor no trivial. Dado que, a diferencia de su predecesor, en C++ es posible declarar las variables directamente cuando se necesitan, es recomendable declarar las variables en el mínimo ámbito necesario y sólo inmediatamente antes de ser utilizadas, para poder alcanzar una eficiencia máxima .

A continuación se muestra un ejemplo donde la condición necesaria sólo se cumple en un 50% de las ocasiones en que se ejecuta el código:

```
int x; //Declaración fuera del "if"
  (Se crea la variable siempre).
if (condicion)
  x = y;

// Declaración dentro del "if" (La variable se declara un
  50% menos de veces)
if (condicion)
    int x = y;
```

Por norma general, declarar los objetos dentro del ámbito del if siempre produce mejoras en la velocidad.

3) Inicialización frente a asignación: Otro vestigio de C es la restricción de que las variables deben ser primero definidas y después inicializadas. En C++ esto no es necesario. De hecho, es una ventaja poder inicializar la variable en el momento de ser declarada. La inicialización de un objeto invoca directamente al constructor copia del mismo, sin más. Definir un objeto para asignarlo posteriormente implica invocar primero al constructor por defecto y luego al operador de asignación. No tiene sentido hacer esto último cuando se puede conseguir lo mismo en un único paso.

4) Listas de inicialización en los contructores: La utilización de métodos constructores y destructores puede implicar un aumento considerable en el tiempo de ejecución. Algunos constructores, especialmente aquellos con múltiples objetos o con objetos de jerarquías muy complejas, emplean mucho tiempo para su creación, que se produce en cascada de constructores, lo que ralentiza el proceso de creación. El programador debe, por tanto, ser consciente de esta "ejecución silenciosa", de forma que minimice lo máximo posible los cálculos realizados por el constructor. Emplear una lista de inicialización (en lugar de una simple asignación) es un modo sencillo de incrementar la eficiencia en el constructor.

V. Resultados experimentales

Se pretende demostrar la importancia de las estrategías eficientes en la creación de código. Para ello, sobre varios códigos fuente originales, se aplicarón diversas modificaciones, con el fin de probar su eficiencia. La presente sección muestra algunos de los resultados más significativos.

A. Cálculo del número π

Para realizar las comprobaciones, se utilizó un código fuente correspondiente a la paralelización del calculo de la aproximación del numero pi[3]. Para ello se realiza el cálculo definido como la integral 0 a 1 de $\frac{4}{1+x^2}$.

Para realizar estos ejemplos, se utilizó una máquina con 4 procesadores GenuineIntel, IA-64 32, Dual-Core Intel(R) Itanium(R) Processor 9140M.

1) Código original: A continuación se muestra un fragmento del código OpenMP que implementa la solución indicada:

```
#pragma omp parallel
{
  #pragma omp master
  {
    nthreads = omp_get_num_threads();
  }
  #pragma omp for private(x) reduction(+:sum)
    schedule(static)
  for (int i=0; i < NUM_STEPS; ++i) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);
  }
  #pragma omp master
  {
    pi = step * sum;
  }\makebox
}
```

Los resultados, calculando el número π con una aproximación integral realizada con 4 hilos en 400 millones de pasos, fueron los siguientes:

```
computed pi = 3.14159  (3.141592653590041)
time to compute = 3.85416 seconds
```

[3] π (pi) es la relación entre la longitud de una circunferencia y su diámetro, en geometría euclidiana.

2) Legibilidad versus Tiempo de Ejecución: Con el fin de demostrar que mejorar la legibilidad de un código puede acarrear importantes penalizaciones en su tiempo de ejecución, se decidió modificar el bloque paralelo de sincronización obligatoria, con reducción en la suma[4], de tal manera que el fragmento de código quedó como se muestra a continuación:

```
#pragma omp for private(x) reduction(+:sum) schedule(static)
for (int i=0; i < NUM_STEPS; ++i) {
  x = (i+0.5)*step;
  x_square = x*x;
  numerator = 4.0;
  denominator = 1 + x_square;
  fraction = numerator / denominator;
  sum = sum + fraction;
  /*sum += 4.0/(1.0+x*x);*/
}
```

Obteniéndose así el siguiente resultado:

```
time to compute = 64.4057 seconds
```

Es decir, se produjo una diferencia entre los tiempos de ejecución de 60,55154 segundos, con una penalización del 1671%.

Esto, apoya el hecho de que mejorar la legibilidad de un código llega a pejudicar seriamente el tiempo de ejecución y, consecuentemente, su eficiencia energética.

3) Sobrecarga del operador +: Con el objetivo de demostrar que el uso de la sobrecarga de operadores mejora los tiempos de ejecución, se decidió modificar el bloque paralelo de sincronización obligatoria, de tal manera que, en lugar de realizar la suma con sobrecarga $sum+ = 4.0/(1.0+x*x)$; se utilizaba $sum = sum + (4.0/(1.0 + x * x))$;. Con este único cambio, el tiempo de ejecución pasó de 3.85416 segundos a 4.99425 segundos. Es decir, se produjo un incremento del 130% en el tiempo de ejecución.

4) Cout versus Printf: Para comprobar las diferencias entre $cout$ y $printf$ se introdujo un fragmento de código dentro del bloque de sincronización. Con 4 millones de iteraciones se produjo una diferencia de 2,05 segundos entre *cout* y *printf*. Con 40 millones de iteraciones, se obtuvo una diferencia de 7,75 segundos a favor de *printf*.

VI. Conclusiones

Se propuso como objetivo principal del proyecto la optimización, dentro del desarrollo de código para computadores de altas prestaciones, de códigos fuentes de ámbito general, con el fin de obtener rendimientos óptimos y eficiencias máximas, sin perder de vista la correcta implementación de sus funcionalidades.

Se considera demostrada la importancia de las estrategias eficientes en la creación de código. Concretamente, según las ejecuciones realizadas, la aplicación de algunas modificaciones sobre un mismo código puede dar lugar a idénticos resultados con tiempos de ejecución muy dispares que, a largo plazo, acarrearán, además, importantes penalizaciones en el consumo energético.

[4]La cláusula *OpenMP reduction* permite especificar una o más variables privadas de subprocesos que están sujetas a una operación de reducción al final de la región paralela.

A partir de los resultados obtenidos es posible afirmar que, mejorar la legibilidad o la futura comprensión de un código por parte de alguien ajeno a su creación, llega a perjudicar seriamente el tiempo de ejecución del mismo y, consecuentemente, su eficiencia energética. Del mismo modo, la sobrecarga de operadores o la elección de una librería apropiada, son destacadas técnicas a tener en cuenta a la hora de realizar una optimización adecuada.

Aunque se trata de un proyecto ambicioso y de complejidad notable, se pretende que en un futuro próximo sea posible generar una solución software que permita medir de un modo más preciso la productividad de la aplicación de técnicas de eficiencia energética a la generación de códigos fuente. Dicha validación sería realizada mediante la utilización de la herramienta sobre diferentes arquitecturas, por parte de diversos investigadores y usuarios de HPC, los cuales presentarían habilidades y experiencias distintas en varios campos de la ciencia.

Referencias

[1] Zhichun Zhu and Xiadong Zhang. Look-ahead architecture adaptation to reduce processor power consumption. *Micro, IEEE*, 25(4):10–19, 2005.

[2] Anne Benoit, Paul Renaud-Goud, and Yves Robert. On the performance of greedy algorithms for power consumption minimization. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 454–463. IEEE, 2011.

[3] Mauro Marinoni and Giorgio Buttazzo. Balancing energy vs. performance in processors with discrete voltage/frequency modes. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 294–304. IEEE, 2006.

[4] Rong Ge. Evaluating parallel i/o energy efficiency. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 213–220. IEEE Computer Society, 2010.

[5] Rong Ge, Xizhou Feng, and Kirk W Cameron. Improvement of power-performance efficiency for high-end computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[6] Nandini Kappiah, Vincent W Freeh, and David K Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33. IEEE Computer Society, 2005.

[7] Torsten Hoefler. Software and hardware techniques for power-efficient HPC networking. *Computing in Science and Engineering*, 12(6):30–37, 2010.

[8] Jayant Baliga, Robert Ayre, Kerry Hinton, and Rodney S Tucker. Energy consumption in wired and wireless access networks. *Communications Magazine, IEEE*, 49(6):70–77, 2011.

[9] Faiza Fakhar, Barkha Javed, Raihan ur Rasool, Owais Malik, and Khurram Zulfiqar. Software level green computing for large scale systems. *Journal of Cloud Computing*, 1(1):1–17, 2012.

[10] Ulrich Kremer. Low power/energy compiler optimizations. 2004.

[11] Kshirasagar Naik. *A survey of software based energy saving methodologies for handheld wireless communication devices*. Department of Electrical and Computer Engineering, University of Waterloo, 2010.

[12] Pete Isensee. C++ optimization strategies and techniques. In *Conference proceedings: conference, March 15-19: expo, March 16-18, San Jose, CA: the game development platform for real life*, page 419. The Conference, 1999.