

Práctica 3:

El Problema del Viajante de Comercio

Programación Distribuida y Paralela

Juan Pedro Bolívar Puente

27 de enero de 2011

Resumen

El siguiente documento es la memoria de la tercera práctica de la asignatura Programación Distribuida y Paralela, impartida en 4º de Ingeniería Informática en la Universidad de Granada.

En ella se desarrolla una implementación paralela del problema del viajante de comercio basada en la técnica de ramificación y poda y con una estructura distribuida en anillo. Finalmente se toman medidas experimentales para comprobar la mejora en rendimiento.

1. Diseño e implementación

En esta práctica, de forma similar a la práctica anterior, hemos realizado una implementación genérica del algoritmo, utilizando las técnicas de programación con plantillas. Por tanto, la discusión que ya hicimos en la práctica anterior sobre las ventajas de esta técnica son aplicables también aquí.

En concreto, hemos abstraído la estructura general de un algoritmo de ramificación y poda (ver fichero `pdp/bnb.*` y `pdp/dist.bnb.*`) de su instanciación para un problema concreto (ver `pdp/tsp.*` para la instanciación al problema del viajante de comercio)¹. De esta forma, existe un “concepto”² abstracto de *nodo*, que el tipo `pdp::tsp::node<CostType>` modela pa-

¹En realidad una implementación general del algoritmo de ramificación y poda debería también permitir parametrizar la estrategia de ramificación LIFO o FIFO entre otros. Por no complicar demasiado la solución y alejarnos de lo fundamental de esta práctica, usamos sólo ramificación FIFO y otras posibles generalizaciones como la estrategia de poda no son parametrizables. No obstante, esto no sería complicado de realizar sobre la base actual dada la, en nuestra opinión, concisión y claridad del código.

²Un “concepto” en *programación genérica* es una noción similar a la de “interfaz” de la *programación dirigida a objetos*, sólo que más general porque se basa en todas las propiedades sintácticas de una entidad genérica y no sólo la signatura de los métodos de una clase particular.

ra el problema del viajante de comercio para cualquier tipo de coste, y que podríamos implementar para otros problemas resolubles por ramificación y poda como el MAX-SAT. Aunque no hemos redactado en ninguna parte la especificación formal de este concepto de “nodo” es fácil elicitarlo de un estudio del código fuente proporcionado. Por dar una descripción general, un nodo debe especificar mediante *traits*[1] las propiedades de su tipo de coste, una función para obtener el coste de un nodo y otra función para obtener los nodos hijos de un nodo. La versión paralela del algoritmo de ramificación y poda aplica algunas restricciones más sobre el concepto de nodo para la serialización en su distribución paralela.

Las ventajas de esta implementación deben ser evidentes: con una implementación concisa del concepto de “nodo” centrada en el dominio del problema concreto a resolver obtenemos automáticamente una versión secuencial y otra paralelizable sin tener que enfangarnos en el farragoso terreno de implementar la ramificación y poda paralela.

Otra ventaja colateral es que el código de manejo de grafos que ya había sido implementado en la práctica anterior ha podido ser reutilizado para esta práctica con mínimas modificaciones para leer el nuevo formato de entrada.³ Incluso, hemos podido reutilizar la función que abstraía la implementación genérica del *main* de un algoritmo sobre grafos paralelizable que ya realizamos para la práctica anterior: generalizar parece ser una buena inversión de futuro :-)

1.1. Sobre la eficiencia de la solución genérica

Para poder generalizar nuestra solución hemos optado por no utilizar la biblioteca `libbbb.*` proporcionada por el profesor ya que esta está demasiado acoplada, usa variables globales y tiene un estilo más cercano a C que a C++. Sin embargo, hemos tratado de que nuestra implementación sea lo más similar posible, de tal forma que el código de `pdp/tsp.*` es prácticamente una traducción literal de gran parte del código de `libbbb.*` a C++ “moderno”. Entre los cambios fundamentales, a parte de la generalización sobre el tipo de coste y el uso de la clase `pdp::graph<CostType>` desarrollada en la práctica anterior, también usamos las clases de la STL como `std::vector<T>` para

³También, para usar elegantemente la implementación del grafo anterior hemos modificado los ficheros de entrada proporcionados por el profesor para que incluyan en la primera línea el tamaño del problema; ya que el recibirlo como parámetro complica la lógica del programa y promueve soluciones ad-hoc como la de usar variables globales de la solución de ejemplo proporcionada que no nos gustan.

hacer el código seguro frente a excepciones y simplificar la gestión de memoria mediante RAII. Por lo demás, el código es algorítmicamente idéntico.

En el código, incluso, hemos primado la elegancia sobre la micro-optimización, como muestra el hecho de que a veces usamos `pdp::tsp::node<>` que es un tipo de dato complejo, como valor de retorno en lugar de usar un parámetro de salida. Aún así, el compilador de C++ probablemente realiza RVO (*Return Value Optimization* ⁴). El problema de los costes de las copias innecesarias se soluciona en el próximo estándar de C++, apodado C++0x, con las “*r-value references*”[2] que son objeto zombies a los que se le puede robar el contenido sin copiarlos como por ejemplo en el caso del retorno de valores o paso de parámetros por copia. Por comparar los resultados —sí, somos enfermizamente frikis— hemos añadido “constructores de movimiento” que se activan cuando se compila con `--std=c++0x` en GCC. Sin embargo, las pruebas informales que hemos realizado no demuestran ninguna mejora en rendimiento al compilar con esta opción, probablemente porque el compilador ya realiza RVO de todas formas, o tal vez porque las copias redundantes se producen en puntos no críticos del código. Por desgracia, el volumen inmenso de prácticas que tenemos que hacer no nos permite agarrar un *profiler* y un *debugger* para analizar con más detalle la situación :D

En cualquier caso, lo importante aquí es que, a pesar de todas estas sutilezas, escribir código genérico y de alto nivel no es necesariamente más lento y, en muchos casos, es más eficiente. Para ello hemos comparado la eficiencia de nuestra versión con la proporcionada por el profesor. La tabla 1 muestra los resultados obtenidos.

n	Original	Genérico
10	0.00155	0.00155
20	0.0718	0.0610
40	188.86	145.429

Cuadro 1: Resultados obtenidos comparando nuestra implementación del algoritmo secuencial con la proporcionada, en segundos.

¿Si el código es prácticamente una traducción del anterior sin mejoras algorítmicas, porque existe una ventaja significativa? Probablemente, sea porque mientras que el código original realiza las copias con bucles `for`, que son difíciles de optimizar por el compilador, mientras que nuestro código usa el constructor de `std::vector` y las funciones `std::copy` y `std::fill` para copiar e inicializar los datos. Estas funciones, a pesar de ser genéricas,

⁴http://en.wikipedia.org/wiki/Return_value_optimization

usan *especialización parcial*, una técnica muy útil con las plantillas de C++ que ya describimos en nuestra discusión anterior de lo *traits*, para ofrecer implementaciones optimizadas para tipos básicos. De esta forma, las copias e inicializaciones de vectores y matrices de enteros que realizamos acaban convertidas en un `memcpy` y `memset`, lo cual el compilador a su vez suele optimizar sustituyendo la llamada por las operaciones que incluyen los procesadores modernos para el manejo de bloques de datos contiguos.

2. Medidas experimentales

Nota: Los resultados abajo descritos son incorrectos debido a un fallo de programación en el mecanismo de distribución de la cota superior. Aun así, los resultados para el caso $N = 40$ no varían mucho, pero si es significativa la diferencia —el tiempo de ejecución se reduce aproximadamente a la mitad— para $N = 20$.

A continuación vamos a estudiar los resultados de la ejecución del algoritmo paralelo bajo las condiciones que se especifican en el guión de prácticas.

Por desgracia, en esta ocasión no disponemos de dos máquinas diferentes, sino de un único ordenador con un procesador Core 2 Duo T7500 @ 2.20GHz ejecutando un Debian Sid con un kernel 2.6.36 compilado por nosotros específicamente para esta máquina con *pre-emption* optimizada para baja latencia en la calendarización de procesos. Por ello, vamos a realizar las pruebas sólo con uno y dos procesos. Por desgracia, esto no arroja demasiada luz sobre la cota de crecimiento del beneficio según aumentamos el número de procesadores, pero nos permite intuir las ventajas del algoritmo paralelo y en cualquier caso el procedimiento para obtener más datos es bastante sistemático.

La tabla 2 muestra los resultados así como el número de iteraciones que realiza cada proceso, cuando existe distribución de la cota superior. La tabla 3 muestra los mismos resultados sin distribución de la cota superior. Incluimos también en estas tablas la ganancia en la ejecución paralela así como el coste por nodo calculado como el tiempo total de ejecución entre el número total de iteraciones.

n	T (P=1)	Iter P1	T/I	T (P=2)	Iter P1	Iter P2	T/I	Ganan.
10	0.0020	205	9.75	0.0013	130	113	5.34	1.53
20	0.0669	3749	17.84	0.0434	2423	2229	9.32	1.54
40	151.817	3129475	48.51	38.734	786281	779425	24.73	3.91

Cuadro 2: Resultados de las medidas experimentales del tiempo de ejecución (en segundos), el tamaño del espacio de búsqueda, y el tiempo por nodo (en microsegundos) para el algoritmo distribuido con difusión de cota superior.

n	T (P=2)	Iter P1	Iter P2	T/I	Ganancia
10	0.0013	133	133	4.88	1.53
20	0.0583	3092	2987	9.49	1.14
40	97.345	1935451	1968020	24.93	1.55

Cuadro 3: Resultados de las medidas experimentales del tiempo de ejecución (en segundos), el tamaño del espacio de búsqueda, y el tiempo por nodo (en microsegundos) para el algoritmo distribuido con difusión de cota superior.

3. Conclusiones

Observamos en estos resultados cómo el algoritmo distribuido con difusión de cota aporta una ganancia muy significativa. Intentemos explicar por qué.

Resulta bastante evidente observando los resultados que el tiempo por nodo disminuye linealmente respecto al número de procesadores. Esto es además muy lógico, si se procesan dos nodos a la vez, se tarda la mitad en procesar cada nodo. También, especulamos, parece aumentar logarítmicamente con respecto al número de nodos procesados: esto se explica en que cada vez procesar un nodo implica acotar una cola más grande cuando el tamaño del problema crece.

Lo que es sorprendente es la gran ganancia que tiene el algoritmo con difusión de cota, que es mayor de la ganancia proporcional al número de procesos esperable si simplemente dividiésemos la carga. Podemos intentar especular por qué ocurre esto. Cuando existe difusión de cota, la cota que recibimos en un proceso es, en realidad, un “oráculo” que va P nodos por delante de nosotros. Es decir, imaginándonos los procesos avanzando sincronamente para simplificar esta imagen, cuando un procesado ha terminado de procesar 1 nodo, recibe una cota que es la mejor de procesar P nodos entre todos los procesos. De esta forma, podemos cojeturar que la ganancia, obviando los costes de comunicación, es de aproximadamente P^2 , ya que, por un lado, procesamos P procesos a la vez y, por otro, dicho imprecisamente,

acotamos con una cota P veces mejor. Por desgracia, los resultados experimentales y nuestras herramientas analíticas son demasiado pocos como para confirmar con mayor formalidad esta hipótesis.

Sin embargo, esta conjetura también nos sirve para justificar por qué la mejora del algoritmo sin difusión de cota es sub-lineal con respecto al número de procesos: a pesar de que se procesan P nodos a la vez, como los procesos no han visto algunos de los nodos que había en mitad de la búsqueda no acotan tanto y se procesan más nodos que procesados secuencialmente. Me pregunto si este efecto puede verse ligeramente agravado para el caso $P = 2$, por alguna forma de acoplamiento “en fase” entre los procesos, debido a la simetría que existe en la ramificación de grado 2 y el hecho de que la función que divide el trabajo divida la cola según elementos pares e impares. De nuevo, la parquedad de nuestros datos experimentales nos hace difícil especular con más precisión sobre estos factores.

Referencias

- [1] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [2] Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki. A Brief Introduction to Rvalue References. Technical Report N2027=06-0097, ISO JTC1/SC22/WG21 – C++ working group, 2006.