Comparação Cilk++ e OpenMP por Algoritmo de Ordenação

Arthur Francisco Lorenzon, Fábio Diniz Rossi[1]

[1]Instituto Federal Farroupilha - Campus Alegrete
RS 377 - Km 27 - Caixa Postal 118 - Alegrete - RS - Brasil

alorenzon@gmail.com, fdrossi@al.iffarroupilha.edu.br

1. Introdução

Arquiteturas paralelas de memória compartilhada se tornaram plataformas comuns entre os computadores pessoais, devido às limitações físicas que previnem o aumento de frequência de processamento e com o aumento da preocupação do consumo de energia dos computadores. Estas arquiteturas estão organizadas como multiprocessadores simétricos (SMP) em um *chip* (*multicore*), oferecendo um ambiente de paralelismo para as aplicações. Porém, o *software* não tem acompanhado o crescimento do *hardware*, pois sem uma grande mudança nas metodologias de desenvolvimento de sistemas, a adição de *cores* não é percebida pelo usuário.

Para tanto, os atuais desenvolvedores de sistemas devem se utilizar de modelos de desenvolvimento que proporcionem um melhor aproveitamento destas, que estão se tornando um paradigma dominante nas arquiteturas de computadores.

A programação paralela vem crescendo cada vez mais, com o advindo das arquiteturas paralelas. Programação paralela é a capacidade de dividirmos uma carga de trabalho entre vários processadores dinamicamente e de forma eficiente, proporcionando um ganho de desempenho na execução de uma aplicação paralela se comparada com sua versão sequencial (*speedup*).

Este trabalho tem o objetivo de avaliar e comparar o desempenho entre o Cilk++ e o OpenMP, que proporcionam paralelismo através de diretivas ao compilador, distribuíndo entre os *cores* a tarefa a ser realizada. Neste modelo, é importante que o programador conheça o seu programa, inserindo diretivas para gerar e controlar sua execução paralela (paralelismo explícito). Portanto, nosso artigo está organizado da seguinte forma: uma seção sobre cada um dos modelos citados, as avaliações de desempenho que realizamos e nossas conclusões.

2. Cilk++

Cilk++ foi lançado em 2007, projeto desenvolvido pelo MIT (*Massachusetts Institute of Technology* desde 1994 com o nome de Cilk. Baseada em GNU C, proporciona paralelismo da tarefa através de algumas diretivas de compilação específicas, que possibilitam a um código-fonte sequencial com pequenas modificações ter ganho de desempenho em arquiteturas multiprocessadas. O Cilk++ suporta C e C++ e é compatível com os compiladores GCC do Linux e do Microsoft C++, com suas versões *Academic* e *Open Source*. Seu código-fonte original ainda está disponível no MIT, licenciada por LGPL (*GNU Lesser General Public License*) [Leiserson 2009].

Quando usa Cilk++, o programador é responsável por explicitar o paralelismo, indentificando os elementos que podem ser executados paralelamente com segurança. A programação utilizando Cilk++ é simples, sendo necessária apenas a adição de três palavras chaves (diretivas) para expressar o paralelismo e a sincronização: cilk_for (permite a um laço executar de forma paralela), cilk_spawn (permite ao programador identificar as funções C (ou métodos) que devem executar em paralelo) e cilk_sync (serve para sincronizar os retornos das *threads* como uma 'barreira local'. Quando todas as *threads* retornam, a execução recomeça no ponto imediatamente após a indicação do cilk_sync).

3. OpenMP

O OpenMP (*Open Multi-Processing*) é uma API (*Application Programming Interface*) multi-plataforma para processamento paralelo baseado em memória compartilhada para as linguagens C/C++ e Fortran, que consiste em um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente [Tian et al. 2005].

O OpenMP tem quatro variáveis de ambiente: OMP_NUM_THREADS (números de *threads default*), OMP_DYNAMIC (ajuste dinâmico do número de *threads* para regiões paralelas), OMP_NESTED (paralelismo aninhado), OMP_SCHEDULE (controla o escalonamento de um laço).

As diretivas de compilação para paralelismo explícito seguem a seguinte sintaxe:

```
#pragma omp diretiva [cláusula]
região paralela
```

Nessa sintaxe, podemos substituir a diretiva por: atomic (especifica o local de memória que será atualizado atomicamente), barrier (sincroniza segmentos), critical (especifica uma região crítica), flush (identifica o ponto de sincronização, com todas as *threads* tendo visão consistente da memória), for (especifica uma região paralela de um laço), master (especifica que somente o mestre vai executar uma seção), ordered (especifica que um laço paralelo deve ser executado como sequencial), parallel (define uma região paralela), sections (identifica uma região a ser dividida entre as *threads*), single (permite a execução do código em uma *thread* simples), threadprivate (define uma variável privada para uma *thread*).

A maior parte das variáveis no código do OpenMP são visíveis a todas as *threads* por padrão. Porém algumas variáveis individuais são necessárias para evitar condições de corrida e existe a necessidade de passar valores entre a parte sequencial e a região paralela, para que os dados do ambiente sejam introduzidos como as cláusulas atribuidas ao compartilhamento de dados.

Essa API foi especificada por um grupo dos grandes fabricantes de *hardware/software* com o intuito de ser portável e escalável, com uma interface de utilização simples e que pudesse ser utilizado tanto para aplicações de grande porte, quanto para aplicações *desktop*. O OpenMP usa um modelo *fork/join*, onde existe um fluxo de execução principal (*master thread*) e quando necessário, novas *threads* são disparadas para dividir o trabalho (fases paralelas). Por fim, ao fim de uma seção paralela, é feito um *join*.

4. Avaliações de Desempenho

Existem várias maneiras de avaliar o desempenho de aplicações, mas nenhuma de implementação tão simples quanto a utilização de estruturas de dados. Uma operação bastante comum com essas estruturas de dados é a ordenação. De fato, muitas técnicas importantes usadas ao longo do projeto de algoritmos são representadas no corpo de algoritmos de ordenação que foram desenvolvidos ao longo dos anos. Desse modo, a ordenação também é um problema de interesse histórico.

Dentre tantas técnicas de ordenação, escolhemos o *bubble sort* [Vyskoč 1990], ou ordenação por flutuação (também chamado método bolha), um algoritmo de ordenação bem simples. A idéia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

No melhor caso, o algoritmo executa $\frac{n^2}{2}$ operações relevantes, onde 'n' representa o número de elementos do vetor. No pior caso, são feitas $2n^2$ operações. No caso médio, são feitas $\frac{5n^2}{2}$ operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados. Como a carga de processamento necessária para ordenar um vetor utilizando o método bolha é bastante significativo computacionalmente, o escolhemos para avaliar o desempenho das interfaces paralelas para compiladores que este artigo trata.

O mesmo código sequencial desenvolvido foi utilizado para comparação com OpenMP e Cilk++, com o mesmo laço paralelizado (código de ordenação) nos dois paradigmas.

No OpenMP, realizamos alteração do código sequencial com a utilização de #pragmas (diretivas de compilação) como vemos abaixo:

```
#pragma omp parallel for
for(int j = tam-1;; j >0;j--){
 for(int m=0;m<j;m++){
    //código de ordenação;
  }
}
```

No Cilk++ podemos ver abaixo:

```
cilk_for(int j = tam-1;; j >0;j--){
 cilk_for(int m=0;m<j;m++){
    //código de ordenação;
  }
}
```

O ambiente que escolhemos como base das avaliações é um computador com processador *Pentium 4 dual core - 3 GHz*, o que proporcionou a oportunidade de paralelismo ao nosso trabalho. Avaliamos ordenação de 3 tamanhos diferentes de vetor: 10 mil valores, 100 mil valores, 1 milhão de valores.

Para uma maior confiabilidade dos resultados, em todos os casos a ordem dos valores a ser ordenados era a mesma, o sistema operacional (*Debian Etch - Kernel 2.6.24*) estava sempre no mesmo estado, foram realizadas 100 execuções de cada tamanho de vetor, excluíndo os 10 maiores e os 10 menores valores, realizando uma média aritmética dos 80 valores restantes. Para calcular o *speedup* desenvolvemos uma versão sequencial de ordenação usando *bubble sort*, e a comparamos às versões paralelas.

Com o vetor de 10 mil valores a ser ordenados temos os seguintes resultados: Sequencial: 0.8 segundos, OpenMP: 0.2 segundos, Cilk++: 0.3 segundos. Com o vetor de 100 mil valores a ser ordenados temos os seguintes resultados: Sequencial: 70 segundos, OpenMP: 10 segundos, Cilk++: 12 segundos. Com o vetor de 1 milhão de valores a ser ordenados temos os seguintes resultados: Sequencial: 8820 segundos, OpenMP: 113 segundos, Cilk++: 126 segundos. Após as avaliações de desempenho, apresentamos nossas conclusões.

5. Conclusões

Através das avaliações realizadas, podemos analisar o *speedup* em cada situação. Com o vetor de 10 mil posições chegamos a um *speedup* de 4 com a utilização do OpenMP e 2.6 com a utilização do Cilk++, com o vetor de 100 mil posições chegamos a um *speedup* de 7 com a utilização do OpenMP e 5.8 com a utilização do Cilk++ e com um vetor de 1 milhão de posições chegamos a um *speedup* de 78 com a utilização do OpenMP e 70 com a utilização do Cilk++.

Portanto, para aplicações com a mesma característica de nossa avaliação (ordenação de vetores utilizando *bubble sort*), notamos que o OpenMP é uma opção de paralelismo um pouco mais rápida que o Cilk++, tanto em pequenas quanto em grandes vetores.

O OpenMP é bem mais versátil que a Cilk++, explicado através da gama de recursos da linguagem para que o programador possa explicitar o paralelismo. Uma recurso interessante do OpenMP é que a sincronização entre as *threads* quase sempre ocorre de maneira implícita, de maneira automática, que faz com que sua utilização se torne simples.

O Cilk++ tem a vantagem de ser mais simples, pois utiliza apenas 3 diretivas de compilação, uma para paralelizar um laço, uma para paralelizar uma função, e outra para sincronização, permitindo que códigos que não precisem de grande complexidade possam ser paralelizados facilmente.

Finalizando, a grande vantagem tanto OpenMP quanto Cilk++ é a sua transparência, pois que não é possível e nem necessário em sua implementação, ver como cada *thread* é criada e inicializada (como acontece com *Posix Threads*). Também não é visível uma função separada contendo o código que cada *thread* executa, bem como a divisão de trabalho realizado sobre um arranjo também não é visível explicitamente.

Referências

Leiserson, C. E. (2009). The cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA. ACM.

Tian, X., Hoeflinger, J. P., Haab, G., Chen, Y.-K., Girkar, M., and Shah, S. (2005). A compiler for exploiting nested parallelism in openmp programs. *Parallel Comput.*, 31(10-12):960–983.

Vyskoč, J. (1990). Making bubblesort recursive. *Inf. Process. Lett.*, 36(4):219–220.