Análise de Paralelismo
Cholesky

Paulo Alexandre Piornedo Panucci[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)

ra88380@uem.br

Abstract. *A sequential code can be written to be parallel executed in a set of threads, with the purpose of taking advantage of the core quantity in a machine, making the program execution faster. It is expected that making a program parallel, it has a significantly better execution time than the sequential program, although there are cases in which the parallelized code has a similar or worst performance compared to the sequential code. In this work will be analyzed the parallel version of Cholesky's algorithm to matrix decomposition, showing that this parallel version does not reach a good performance concerning its sequential version.*

Resumo. *Um código sequencial pode ser escrito para ser executado paralelamente em um conjunto de threads, com finalidade de tirar proveito da quantidade de núcleos presentes em uma máquina e assim tornar a execução do programa mais rápida. Espera-se que ao paralelizar um determinado programa, ele possua um tempo significantemente melhor que o código sequencial, mas também existem casos em que um código paralelizado tem um desempenho próximo, ou pior, que o sequencial. Neste trabalho será analisada a paralelização do algoritmo Cholesky para decomposição de matrizes, mostrando que este algoritmo não alcança um bom desempenho paralelizando-o com referência no código sequencial.*

1. Introdução

Como observado por [Aubanel 2016], antes da revolução dos processadores *multicore*, o programador dependia das melhoras de performance apresentadas por processadores conforme o lançamento de novas gerações. Com a estagnação do aumento da taxa de *clock*, poder utilizar múltiplos *cores* para conseguir rodar um programa tende a tornar a execução mais eficiente dependendo como este problema é modelado.

Hoje conta-se com diretivas de compilação e APIs para paralelizar um código. Um exemplo é o *OpenMP* para a linguagem C/C++. Apesar de existirem alternativas de deixar o trabalho de paralelização para o compilador, aprender a escrever códigos paralelos distribuídos tem suas vantagens. [Aubanel 2016] ainda diz que os compiladores tentam otimizar o código que é dado a eles, porém eles não reescrevem esse código para que ele realmente seja paralelo. Desta forma pode-se dizer que existe uma diferença entre paralelizar um código em C com o OpenMP e escrever um código paralelo utilizando a biblioteca Pthread do C.

2. Fundamentação Teórica

Este trabalho abordará a análise de paralelismo do algoritmo da decomposição de Cholesky. Desta forma, é importante se conhecer o objetivo desta decomposição, assim como alguns conceitos básicos de paralelismo para se entender esta análise.

2.1. Decomposição de Cholesky

A decomposição de Cholesky consiste em: dada uma matriz A simétrica e positiva definida, existe uma matriz triangular inferior L tal que multiplicada por sua transposta o resultado seja A. Logo, esta decomposição pode ser definida pela seguinte fórmula:

$$A = \mathrm{L} \times \mathrm{L}^{\mathrm{T}}.$$

Uma matriz simétrica consiste em uma matriz no qual sua transposta é igual a ela mesma. Ou seja:

$$A = \mathrm{A}^{\mathrm{T}}.$$

Uma matriz é definida positiva caso todos os seus auto-valores são maiores que zero.

2.2. Paralelismo

Serão abordados agora alguns conceitos básicos importantes para se entender o paralelismo a nível de thread e sua análise.

2.2.1. *Speedup*

Ao se escrever um código paralelo a partir de um sequencial, espera-se que seu código paralelo tenha um desempenho de tempo que se aproxima de N vezes melhor que o tempo do código sequencial, sendo este N o número de threads que serão executadas em paralelo, e este valor é dado pelo *speedup*.

Quando dividimos o tempo de execução do algoritmo sequencial pelo tempo de execução do algoritmo paralelo, tem-se então o seu *speedup*:

$$S_p = \frac{T_s}{T_p}.$$

É importante distinguir o tempo decorrido da execução do código, do tempo total levado pela soma dos tempos de execução de cada *thread*. Para fim de análise de speedup precisa-se do *elapsed time*, que é o tempo decorrido do começo ao fim do programa (ou de uma função paralela específica).

Quando o *speedup* de um determinado programa paralelo possui um valor maior que o número de *threads* que executa este código, tem-se então um *speedup* dito superescalar.

2.2.2. Condição de Corrida

Ao se escrever um código paralelo, quando *threads* simultâneas podem competir por algum recurso diz-se que existe condição de corrida neste algoritmo. Pode-se definir como recursos acesso em uma determinada região da memória, e até mesmo o próprio hardware em que o algoritmo está sendo executado. Como um exemplo de disputa de

recursos, pode se dar o exemplo que uma determinada *thread* queira escrever ao mesmo tempo que outra em uma região da memória, ou também a primeira quer ler um resultado que está ao mesmo tempo sendo escrita pela segunda. Como exemplo de disputa de recurso de *hardware*, uma quantidade x de *threads* que estão prontas para ser executadas, porém a máquina contém $\frac{x}{2}$ núcleos de processamento.

Para se resolver problemas de condição de corrida, principalmente de software, deve se usar recursos como *locks* e barreiras.

Se existe alguma região crítica em que somente uma *thread* pode estar executando essa região, pode-se utilizar *lock* para não deixar que outras *threads* acessem essa área e *unlock* para liberar para as outras.

Ao se procurar por sincronia em um algoritmo, opta-se pelo uso de barreiras. Quando uma barreira é colocada no algoritmo, isso implica que todas as *threads* devem chegar na mesma região para assim poder dar continuidade na execução.

Ter condições de corrida no algoritmo que deseja paralelizar implica em diminuir a possibilidade de ter um *speedup* próximo a quantidade de *threads* que estão executando. Isto se dá pelo fato de impedir que as *threads* dependam delas mesmas para executar o trabalho que a ela foi dado.

2.2.3. Balanceamento de Carga

Independente de como é feita a implementação de um código paralelo, é interessante que *threads* tenham um balanceamento execução, ou seja, façam quantidades de trabalho parecidas, para assim alcançar uma melhor performance.

Em Pthread consegue-se enxergar nitidamente o balanceamento ou a falta dele, pelo fato de isso ter que ser explicitado pelo programador.

2.3. Ferramentas

Serão apresentadas algumas ferramentas utilizadas para o desenvolvimento do algoritmo paralelo e sua análise.

2.3.1. Polybench

O Polybench é uma coleção de *benchmarks* contendo partes de controle estático, com o propósito de uniformizar a execução e o monitoramento de kernels.

Desta coleção de *benchmarks* que se teve como base de analise o algoritmo de Cholesky.

2.3.2. OpenMP

A ferramenta *OpenMP* é uma *API* que dá suporte a programação paralela em memória compartilhada para C/C++ e Fortran.

Esta ferramenta funciona declarando a diretiva ”#pragma omp parallel”seguida de alguns argumentos nos locais em que deseja paralelizar o código. Fica à cargo do compilador decidir como esse código será então paralelizado.

2.3.3. Pthread

Pthread é uma biblioteca para a linguagem C em sistemas *UNIX* que como especificado por [Barney] padroniza a programação em threads nesse ambiente.

Ao se tratar de uma biblioteca de criação de threads, fica a cargo do programador em ditar a maneira como a paralelização funcionará. Desta forma deve-se criar threads e passar uma função como argumento para ser executada. Deve-se também fazer a junção dessas threads e se precisar, matá-las.

2.3.4. PAPI

PAPI se refere á *Performance API project* que segundo [Laboratory] especifica uma *API* padrão para acessar contadores de performance de hardware disponíveis na maioria dos processadores modernos.

Os processadores em que essa *API* é suportada possui um número x de contadores que podem ser usados para coletar dados. O que pode ocorrer é a quantidade de dados diferentes que se quer coletar seja maior que o número de contadores, ou até mesmo alguns dados precisam de mais de um contador, fazendo com que na soma total de dados e contadores necessitados passe a quantidade de contadores disponíveis. Quando isto ocorre a ferramenta faz a multiplexação de dados.

2.4. VTune™

O VTune™ é um software desenvolvido pela Intel® para realização de análise de performance em processadores da marca.

3. Proposta

Este trabalho tem como proposta desenvolver um algoritmo paralelo da decomposição de matrizes utilizando o método de Cholesky com base num código sequencial presente no Polybench, mostrar que este programa presente no conjunto de *benchmarks* precisa ter uma alteração em sua estrutura sequencial para se tornar paralelo. Através desta alteração, fazer a análise de *speedup* do código paralelo usando Pthread e *OpenMP* em relação ao código sequencial reescrito e ao código original do Polybench, e assim mostrar um baixo desempenho da versão paralela, explicando suas condições de corrida.

Para isto, será apresentado o código do algoritmo de Cholesky do Polybench e o código desenvolvido para a proposta de paralelização deste trabalho.

3.1. Cholesky do Polybench

```
1 static void cholesky_polybench(/*PARAMETROS*/){
2   int i, j, k;
3   #pragma scop
4   for (i = 0; i < size; i++) {
5      //j<i
6      for (j = 0; j < i; j++) {
7         for (k = 0; k < j; k++) {
8            A[i][j] -= A[i][k] * A[j][k];
9         }
```

```
10          A[i][j] /= A[j][j];
11        }
12        // i==j case
13        for (k = 0; k < i; k++) {
14          A[i][i] -= A[i][k] * A[i][k];
15        }
16        A[i][i] = SQRT_FUN(A[i][i]);
17      }
18      #pragma endscop
19    }
```

3.2. Cholesky Proposto

```
1  static void cholesky_proposto(){
2      int i, j, k;
3      for(k = 0; k < size; k++){
4          A[k][k] = sqrtf(A[k][k]);
5          for(j = (k + 1); j < size; j++){
6              A[k][j] /=  A[k][k];
7              A[j][k] = A[k][j];
8          }
9          for(i = (k + 1); i < size; i++){
10             for(j = i; j < size; j++){
11                 A[i][j] -=  A[k][i] *  A[k][j];
12                 A[j][i] = A[i][j];
13             }
14         }
15     }
16     for(i = 0; i < size; i++){
17         for(j = i + 1; j < size; j++){
18             A[i][j] = 0.0;
19         }
20     }
21 }
```

O cholesky escrito pela equipe do Polybench faz o procedimento sobre a mesma matriz de entrada, transformando-á em uma matriz triangular inferior. No algoritmo proposto se manteve o mesmo princípio. As operações de divisão e raiz quadrada presentes no algoritmo não podem ser executadas por *threads* simultâneas, sendo um caso de condição de corrida. Neste caso, somente uma *thread* pode executar essas operações, e as outras devem esperar a conclusão desta etapa.

Os dois *for loops* entrelaçados seguintes contém uma operação de subtração. Esta etapa é conhecida como etapa de ”eliminação”. Esta etapa pode ser paralelizada por não apresentar nenhuma condição de corrida. No caso do OpenMP se coloca a diretiva vista na subseção 2.3.2 seguida de alguns parâmetros acima do *for loop* mais externo. No caso do Pthread se define um inicio e um fim nesse laço mais externo, balanceando esse início e fim proporcionalmente entre as *threads* para de alguma forma tentar realizar um balanceamento de carga. Finalizando esse laços entrelaçados, todas as *threads* devem começar juntas a etapa de de zerar o triângulo superior da matriz.

Todos os casos citados de sincronização de *threads* usam barreiras.

4. Ambiente Experimental

Os algoritmos analisados são desenvolvidos na Linguagem C. São utilizados o OpenMP e Pthread para fazer a versão paralela do código. O PAPI é utilizado como ferramenta de análise.

Foram analisados então 4 códigos: o código sequencial do Polybench, um código sequencial desenvolvido para se adaptar ao código paralelo, um código paralelo Pthread.

Para a análise de *speedup*, cada código foi executado 11 vezes, sendo a primeira descartada para fim de alimentar a memória *cache*. Nas dez execuções seguintes, salva-se os tempos de execução a biblioteca *time.h* do C, eliminando o melhor e o pior tempo destas, e assim fazendo a média dos tempos que sobraram.

Os algoritmos foram rodados e as análises para speedup coletadas no servidor Pitomba localizado no Departamento de Informática da Universidade Estadual de Maringá. Este servidor conta com 60 núcleos de processamento, sendo 30 deles físico e 128 *GBs* de memória RAM.

Com a ferramenta PAPI foram coletados os seguintes dados, seguindo o mesmo padrão de coleta da análise de *speedup*, porém coletados somente para 8 *threads*:

- Quantidade de *cache miss* em L1, L2, e L3;
- Quantidade de ciclos de *clock*;
- Quantidade de instruções completadas.

Esta análise com as informações do PAPI foram coletadas em uma máquina Intel® i7 8 núcleos, com 8 *GBs* de RAM, disponibilizando 07 contadores para o uso da ferramenta. Esta coleta foi realizada separadamente da coleta de *speedup* pelo fato de o servidor Pitomba não conter a ferramenta PAPI nem uma outra ferramenta de análise de performance, e também não existir a possibilidade de instalar alguma ferramenta para este fim.

Através do VTune™, na mesma máquina utilizada para os testes com o PAPI, foram coletadas amostras de uma execução do código sequencial desenvolvido, uma do *OpenMP* com 8 threads e uma do Pthread também com 8 *threads*. Foi analisada então o tempo ocioso do processador.

5. Resultados

Como apresentado por [Ruschel 2016] a quantidade de dependência de dados influencia na análise de tempo de seu código paralelo, e o Cholesky apresenta implementações variadas com diferentes dependências de dados. No Cholesky proposto na subseção 3.2 vemos as condições de corrida para o paralelismo. Nas figuras 1 e 2 vemos respectivamente a análise de *speedup* comparando os códigos paralelos OpenMP e Pthread com o Cholesky sequencial do Polybench e o sequencial do proposto.

O desempenho de *speedup* não é bom nem na primeira nem na segunda ocasião, apesar de apresentar melhoras comparando a análise com o mesmo modelo de algoritmo.

Pode-se tentar explicar o mau desempenho de *speedup* analisando 2 informações

- Quantidade de *cache miss*, caso sendo esta taxa alta motivo de mais acesso a memória principal;
- Tempo ocioso do processador, analisando uma média de instruções por ciclo e o tempo de trabalho do CPU.

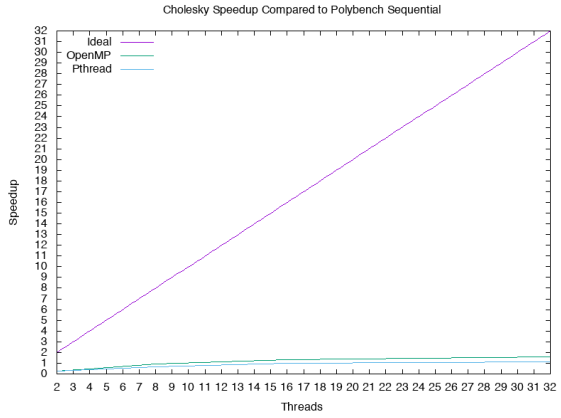

Figura 1. Análise de *speedup* a partir do código sequencial do Polybench.

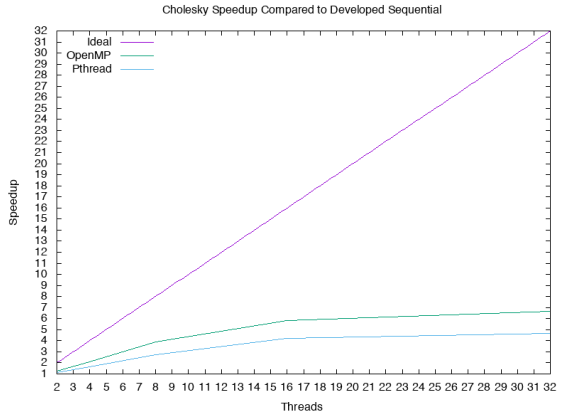

Figura 2. Análise de *speedup* a partir do código sequencial proposto.

Como pode ser visto na figura 3 a taxa de *cache miss* entre o código sequencial e o Pthread é próxima, sendo o segundo um pouco mais custoso em *misses*. Isso se dá ao fato de a parte em que as *threads* estão trabalhando juntas no código paralelo, elas trabalham em diferentes áreas da matriz, fazendo com que cada *thread* busque informações diferentes das outras, não contendo esse dado na cache, desencadeado mais *cache misses*.

Desta forma, podemos considerar a taxa de *cache miss* um fator que influencia no desempenho geral do algoritmo.

Analisando o uso do processador na figura 4 vemos um núcleo com maior parte de trabalho de processamento. Isto se dá pelo fato de ter sido recolhida uma amostragem com um *dataset* menor, e desta forma por este *core* ficar responsável pela inicialização do programa e todas as configurações das matrizes (inicialização e povoamento) ele possui um uso significantemente maior que os outros. Porém pode-se perceber que nem todos os cores apresentam uso balanceado, além de não estar sendo usados todos os 8 presentes na máquina, enquanto numa amostragem do OpenMP está sendo usado. A questão de balanceamento de carga está referida ao motivo explicado na subseção 3.2 que somente uma *thread* pode fazer a primeira etapa do algoritmo que é as operações de divisão e raiz quadrada. Enquanto isto é executado, todas as outras *threads* devem esperar para começar a trabalhar.

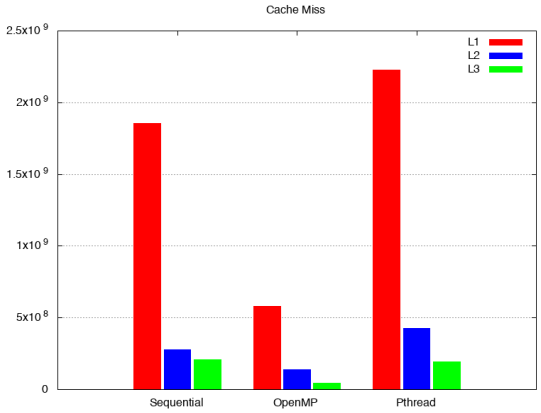

Figura 3. Análise de *cache miss* 1.

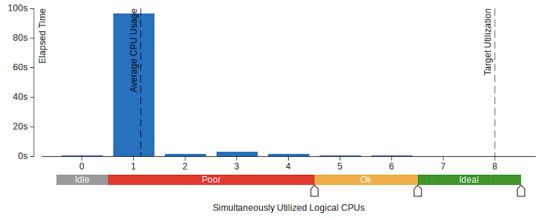

Figura 4. Uso de CPU na execução do Pthread com 8 *threads*

Esse tempo ocioso da maior parte das *threads* pode ser percebido com os dados coletados pelo PAPI que mostra que para o mesmo *dataset* utilizado na analise de CPU os códigos sequencial proposto tem uma média de 1,116 instruções por ciclo de *clock*, o OpenMP uma média de 1,26 instruções por ciclo e o Pthread 1,31. O que se espera pensando em um código paralelo a partir do sequencial, é que a relação de instruções por ciclo tenha uma relação crescente proporcional ao número de *threads* que executam no programa. Não se percebe nenhuma mudança significante entre o código sequencial e os paralelos. Isto pode indicar ociosidade de processador, não tendo um balanceamento de carga bom o suficiente para aumentar de alguma maneira a relação de instruções por ciclo. Uma diferença importante de se lembrar é que o código sequencial acaba sendo executado em um núcleo, enquanto os paralelos deveriam utilizar mais núcleos e assim distribuir o trabalho.

O que pode explicar um melhor desempenho de *speedup* do OpenMP em relação ao Pthread é o tempo ocioso que este último possui e o primeiro não no mesmo nível, devido ao fato de um *for loop* externo do OpenMP ser declarado como paralelizável. Desta forma a primeira parte, que é executada somente por uma *thread* no Pthread fazendo com que as outras fiquem ociosas esperando esta terminar a cada iteração do laço principal, é feita sequencialmente no OpenMP, diminuindo o tempo de ociosidade de outros núcleos.

6. Conclusão

Paralelizar o algoritmo sequencial que está contido na coleção de *benchmarks* Polybench exige modificações de forma a permitir lidar com suas condições de corrida. O fato de o algoritmo apresentar uma parte significante de suas operações com condições de corrida faz com que sua paralelização seguindo o modelo do código sequencial apresente

um desempenho muito abaixo do esperado em termos de *speedup*. O código paralelo em sua parte paralelizável ao trabalhar em diferentes áreas da matriz de entrada faz com que ocorra bastante *cache miss*, e também a sincronização do algoritmo devido ao grande conflito de dados gerando condições de corrida faz com que o desempenho do *speedup* não seja o tão significante como o esperado. Para uma melhora neste ponto, exige-se uma nova modelagem e abordagem deste algoritmo de decomposição de matrizes.

Apesar do desempenho não tão significante, existe uma melhora de tempo do sequencial para o paralelo. Diminuir o tempo de execução pela metade ou em um quarto pode ser significante quando se pensa em uma escala de tempo maior. Logicamente ter um algoritmo que trabalha mais rápido em cima de um dado problema é vantajoso quando não se tem outras opções. Porém é esperado poder tirar o máximo de vantagem da quantidade de núcleos que se tem disponíveis para processamento, fazendo com que um *speedup* como o apresentado na seção 5 não seja tão interessante.

Referências

Aubanel, E. (2016). *Elements of Parallel Computing*. Chapman and Hall/CRC.

Barney, B. Pthreads overview. `https://computing.llnl.gov/tutorials/pthreads/#Overview`. Accessed: 2017-05-22.

Laboratory, I. C. Papi overview. `http://icl.cs.utk.edu/papi/`. Accessed: 2017-05-22.

Ruschel, J. P. T. (2016). Parallel implementations of the cholesky decomposition on cpus and gpus. Technical report, Universidade Federal do Rio Grande do Sul, Instituto de Informática.