Comparação do uso de OpenMP e Pthreads em uma Paralelização de Multiplicação de Matrizes*

Gabriella Lopes Andrade[1], **Márcia Cristina Cera**[1]

[1]Ciência da Computação – Universidade Federal do Pampa (UNIPAMPA)
Campus Alegrete – Av. Tiarajú, 810, Bairro Ibirapuitã, CEP: 97546-550 – Alegrete – RS

gabie.lop.s@gmail.com, marciacera@unipampa.edu.br

Resumo. *Este trabalho envolve a paralelização de uma Multiplicação de Matrizes com duas APIs para a programação multithreading: PThreads e OpenMP. O objetivo deste trabalho é comparar o desempenho dessas duas APIs variando-se o número de threads usadas e a dimensão das matrizes. Nossos resultados mostram que a maior faixa de ganho de desempenho foi obtida com OpenMP devido a seus mecanismos dinâmicos de escalonamento da carga de trabalho.*

1. Introdução

Existem muitas aplicações que necessitam de um grande poder de processamento para serem executadas em tempo hábil, como por exemplo, a previsão do tempo, simulações físicas, etc.; as quais demorariam vários dias ou até meses se fossem executadas sequencialmente. Para resolver esse tipo de problema é utilizada a programação paralela, a qual consiste em solucionar um problema dividindo-o em partes, de maneira que essas partes possam ser executadas em paralelo. Assim, a programação paralela busca reduzir o tempo de execução da versão paralela em relação a versão sequencial [Schepke and Lima 2015].

Nesse trabalho utilizaremos duas interfaces de programação de aplicações (API - *Application Programming Interface*) para a programação paralela em ambiente *multicore*: POSIX *threads* (Pthreads) [Butenhof 1997] e *Open Multi-Processing* (OpenMP) [Chapman et al. 2008]. Foram desenvolvidas versões paralelas de um algoritmo que realiza a multiplicação de matrizes quadráticas com essas duas APIs. Nosso objetivo é comparar o desempenho obtido por Pthreads e OpenMP na paralelização desta aplicação.

2. Programação *Multithreading*

2.1. Pthreads

A biblioteca Pthreads fornece primitivas para a manipulação de *threads* nas linguagens de programação C e C++ para arquiteturas de memória compartilhada [Butenhof 1997]. Através delas, o programador cria, finaliza, sincroniza e distribui a carga de trabalho entre as *threads* que compõem o programa paralelo.

A criação de *threads* se dá pela primitiva `pthread_create()`, a qual tem como parâmetro a função que a nova *thread* deve executar. Logo, o programador define a carga de trabalho de cada *thread* nesta função, sendo que podem haver *threads* executando funções distintas. O término da execução de *threads* se dá pelo comando `return` ao final da função que está sendo executada ou através da primitiva `pthread_exit()`.

*Este trabalho recebeu recursos do Edital de Apoio a Grupos de Pesquisa - UNIPAMPA/2015.

Em Pthreads, uma forma de realizar a sincronização de *thread* é com o uso da primitiva `pthread_join()`, onde a *thread* principal mantêm-se bloqueada até que todas as demais terminem de computar seus dados e os disponibilizem. Outra forma é com o uso de variáveis do tipo *mutex* (`pthread_mutex_t`). O *mutex* é um construtor de sincronização que permite que uma *thread* tenha acesso exclusivo a uma área de dados. Assim, é possível garantir que uma sessão crítica do código pode ser executada sem que outra *thread* manipule a mesma área de dados, interferindo no resultado.

2.2. OpenMP

A biblioteca OpenMP fornece um modelo escalável e portável para o criação de programas com múltiplas *threads* para memória compartilhada, disponível nas linguagens de programação C, C++ e Fortran [Chapman et al. 2008]. Ela fornece diretivas de compilação que ao serem inseridas no código sequencial informam ao compilador quais blocos de código devem ser executados por *threads* em paralelo. A diretiva `#pragma omp parallel` delimita o trecho do código a ser executado em paralelo.

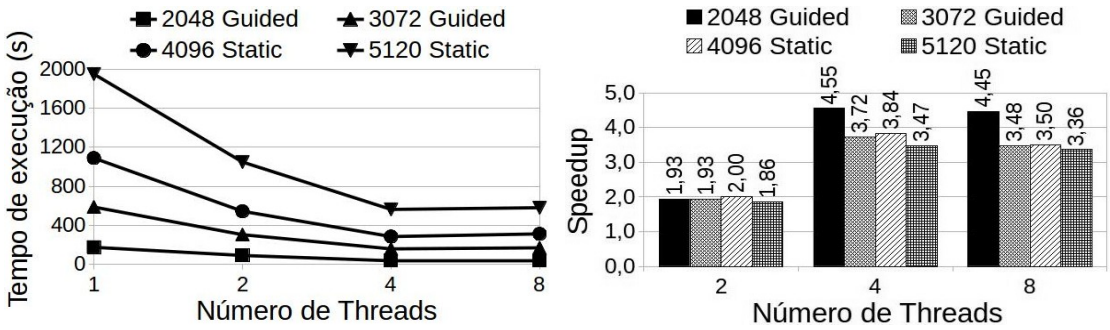
Para paralelizar laços `for` com OpenMP basta inserir a diretiva `#pragma omp parallel for`. Ao inserir essa diretiva anteriormente ao laço a ser paralelizado, as iterações do laço serão distribuídas entre as *threads*. A OpenMP fornece diferentes políticas de distribuição de iterações de laços `for` entre *threads* (cláusula `schedule (política,chunk)`). As principais são: *(i) Static:* A distribuição é igualitária entre todas as *threads*, de forma estática e em tempo de compilação; *(ii) Dynamic:* A distribuição é de forma dinâmica, ou seja um bloco de iterações é atribuído a cada *thread* que tenha terminado seu bloco anterior; *(iii) Guided:* A distribuição também é dinâmica, mas o bloco de iterações inicia grande e vai diminuindo até chegar ao tamanho do *chunk*.

3. Desenvolvimento das Versões Paralelas

Foi utilizado um código em linguagem C que realiza a multiplicação de duas matrizes quadráticas, conforme ilustrado no pseudocódigo abaixo. Nele, o laço A controla as linhas da matriz M1, o laço B controla as colunas da matriz M2 e o laço C é responsável por realizar a multiplicação de uma linha da matriz M1 por uma coluna da matriz M2 e armazenar o resultado na posição correspondente da matriz MR.

```
1. for(i=0; i<nL; i++){   // laço A
2.     for(j=0; j<nC; j++){   // laço B
3.         for(k=0; k<nC;k++){   // laço C
4.             MR[i][j] += M1[i][k]*M2[k][j];
5.         }
6.     }
7. }
```

A paralelização com Pthreads buscou dividir igualitariamente a carga de trabalho entre as *threads*. Nela, a matriz MR é compartilhada entre todas as *threads*. Cada *thread* receberá uma certa quantidade de linhas de M1 e as multiplicará por todas as colunas de M2. A implementação desta paralelização alterou a linha 1 do pseudocódigo acima para `for(i=(threadId*(nL/nT)); i<((threadId+1)*(nL/nT));i++)`. Onde, para atingir o particionamento de dados igualitário entre as *threads*, calculou-se a quantidade de linhas de M1 que cada *thread* receberá através de: `nL/nT` linhas, onde `nL` é o número total de linhas de M1 e `nT` é o número de *threads* utilizadas.



(a) Tempo de execução. (b) Speedup.

Figure 1. Tempo de execução em segundos e *Speedup* obtidos com OpenMP.

Para realizar a paralelização com OpenMP, primeiro testamos quais dos laços aninhados traria maiores ganhos ao ser paralelizado. Com uma amostra de 10 execuções, identificamos que a paralelização do laço B, mostrado no pseudocódigo acima foi 16% mais eficiente que a do laço A. Sendo assim, a diretiva `#pragma omp for` foi inserida antes do laço B seguida da primitiva `#pragma omp parallel private(i,j,k) shared(M1,M2,MR,nL,nC)` antes do laço A. Assim, serão distribuídas entre as *threads* colunas de M2 e cada *thread* terá um conjunto de colunas a serem multiplicadas pela linha de M1 correspondente.

4. Ambiente de Execução

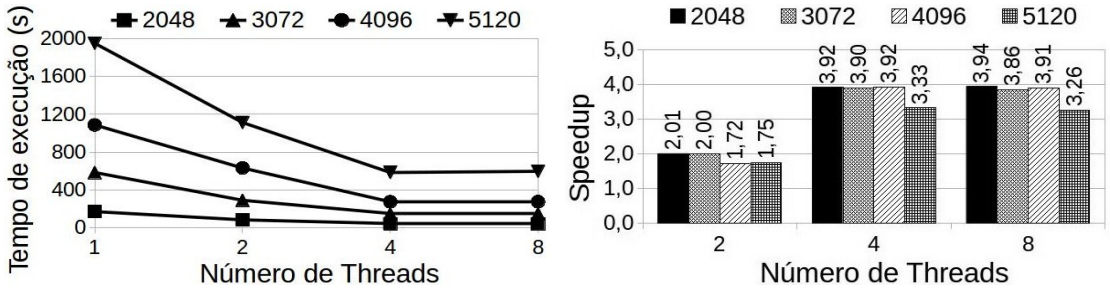
Tanto para Pthreads quanto para OpenMP, variou-se o número de *threads* entre 2, 4, e 8, além da execução sequencial. A ordem das matrizes de entrada foram variadas entre 2048, 3072, 4096 e 5120. Para o OpenMP também variou-se a política de distribuição de iterações entre as *threads* e o número de iterações atribuídas de cada vez (*chunk*). Neste trabalho usaremos o *chunk* grande (`dimensão da matriz/número de threads`) por este ter sido mais eficiente em testes anteriores [Andrade and Cera 2015]. Para obter o tempo de execução, foram calculadas as médias dos tempos de amostras de 10 execuções. Para todas as amostras o desvio padrão ficou abaixo de 2% da média.

A coleta dos dados foi realizada em um Desktop com processador AMD A8-6500B APU de 1,7 GHz de frequência. Ele possui 4 núcleos físicos, com 2 *threads* por núcleos, com 2 núcleos por *socket*. Possui 2 níveis de cache, com cache L1 de dados de 16 KB, cache L1 de instruções de 64 KB ambas privadas, cache L2 compartilhada de 2048 KB e memória RAM de 3 GB. O sistema operacional utilizado é o Ubuntu na sua versão 14.04 LTS e compilador GCC em sua versão 4.8.2-1.

5. Análise de Desempenho

As Figuras 1 e 2 apresentam os tempo de execução em segundos e os *Speedups* para OpenMP e Pthreads. Neles, variou-se a ordem das matrizes (2048, 3072, 4096 e 5120) e o número de *threads* (1 - representando o sequencial, 2, 4 e 8).

A Figura 1 apresenta apenas as políticas de distribuição de iteração que levaram aos menores tempos conforme a ordem das matrizes: 2048 - *Guided*; 3072 - *Guided*; 4096 - *Static* e 5120 - *Static*. Nota-se que conforme a dimensão da matriz aumenta, a



(a) Tempo de execução. (b) Speedup.

Figure 2. Tempo de execução em segundos e *Speedup* obtidos com Pthreads.

melhor política de distribuição muda de *Guided* para *Static*. Em outras palavras, conforme aumenta-se a quantidade de iterações a serem distribuídas e consequentemente o tamanho do *chunk*, é mais vantajoso alocá-las em tempo de compilação. Observando a Figura 1(a), percebe-se que, para todos os tamanhos de matrizes, os menores tempos de execução foram obtidos com 4 *threads*. Isso ocorre pois o processador utilizado possui 4 núcleos físicos e o uso de uma *thread* por núcleo fornece o melhor desempenho. Também identificou-se que os maiores *Speedups* foram obtidos para matrizes de ordem 2048, conforme pode ser visto na Figura 1(b). Logo, este tamanho de entrada na arquitetura alvo levou a melhor utilização da hierarquia de memória.

A Figura 2 mostra que as execuções com Pthreads tiveram um comportamento similar ao OpenMP, onde entradas de ordem 2048 obtiveram os menores tempos (Figura 2(a)), e portanto os maiores *Speedups* (Figura 2(b)). Porém, a faixa dos maiores ganhos de desempenho, que para OpenMP foi de 3,47 a 4,55 conforme a ordem da matriz, para Pthreads foi de 3,33 a 3,92. Essa diferença está vinculada ao métodoo de balanceamento de carga. Enquanto que em Pthreads utilizou-se uma distribuição igualitária, em OpenMP a dinamicidade da política de distribuição *Guided* levou a um melhor desempenho.

6. Conclusões

Este artigo buscou comparar os ganhos de desempenho obtidos na paralelização de uma multiplicação de matrizes utilizando OpenMP e Pthreads. Através dos resultados obtidos podemos concluir que a paralelização utilizando OpenMP levou a maior faixa de ganho de desempenho quando comparada as execuções com Pthreads, de 4 a 13 % maior. Isto deve-se ao melhor balanceamento de carga fornecido pela política de distribuição de iterações *Guided* do OpenMP, a qual permite o ajuste da carga de trabalho em tempo de execução.

References

Andrade, G. L. and Cera, M. C. (2015). Paralelização de uma multiplicação de matrizes utilizando openmp. In *Anais do SIEPE 2015*.

Butenhof, D. R. (1997). *Programming with Posix threads*. Addison-Wesley, Boston (Mass.), 1st edition.

Chapman, B., Jost, G., and Van DeR Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*. MIT press, volume 10 edition.

Schepke, C. and Lima, J. V. F. (2015). Programação paralela em memória compartilhada e distribuída. In *Anais da ERAD/RS 2015*, pages 45–70. SBC.