Análise da paralelização do benchmark syrk utilizando Pthreads e OpenMP

Arthur Manuel Bandeira[1]

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

ra67226@uem.br

Abstract. *This paper describes the parallelization analysis of the syrk benchmark using two different approaches, Pthreads and OpenMP. Based in metrics obtained using an execution monitoring tool, results and comparatives between the two approaches was gathered and will be discussed over the paper.*

Resumo. *Este artigo descreve a análise da paralelização do benchmark syrk utilizando duas abordagens, Pthreads e OpenMP. Baseando-se em métricas obtidas com o uso de uma ferramenta de monitoramento das execuções, foram obtidos resultados e comparativos entre as duas abordagens que serão discorridos no desenvolvimento do trabalho.*

1. Introdução

Neste trabalho, serão analisadas as implementações paralelas do algoritmo Syrk, fazendo uso de duas abordagens diferentes, uma utilizando a biblioteca Pthreads e outra com a biblioteca OpenMP.

As métricas utilizadas para análise foram obtidas utilizando a ferramenta Perf e são: *speedup*, eficiência, número de instruções por ciclo de clock, erros na cache, mudanças de contexto e migrações.

Este documento se encontra dividido em seções. Na seção 2 encontra-se uma breve fundamentação sobre o problema e as ferramentas utilizadas para realização do trabalho, na seção 3 explica-se o que foi desenvolvido no presente trabalho, estratégias de paralelização e sincronização e, instruções de execução do código. Na seção 4 os resultados obtidos são apresentados, na seção 5 apresenta-se a conclusão e por fim, encontram-se as referências.

2. Fundamentação Teórica

2.1. Syrk

A atualização simétrica de *rank-k* (SYRK) é um caso especial de multiplicação entre matrizes. Esta rotina calcula a parte superior (ou inferior) triangular do resultado do produto de matrizes utilizando a equação

$$C = \alpha A A^T + \beta C$$

onde α e β são escalares, C é uma matriz simétrica n por n, A é uma matriz n por k e A^T é a transposta de A [Badia et al. 2012].

2.2. PolyBench

PolyBench é uma coleção de benchmarks . Tem como objetivo uniformizar a execução e o monitoramento de *kernels*, costumeiramente utilizados em publicações passadas e atuais.

2.3. Perf

Perf é um subsistema baseado no *kernel* Linux que proporciona um *framework* para análise de performance. Abrange recursos tanto a nível de *hardware* quanto a nível de *software* [per 2017].

2.4. OpenMP

OpenMP é uma interface de programação (API), portável, baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores. É composto por três componentes básicos: diretivas de compilação, biblioteca de execução e variáveis de ambiente.

2.5. Pthreads

POSIX Threads ou usualmente, Pthreads, é uma interface de manipulação de *threads* padronizada em 1995 pelo IEEE. É definida como um conjunto de tipos e chamadas de função da linguagem C.

3. Desenvolvimento

A paralelização foi realizada dividindo o problema

$$C = \alpha A A^T + \beta C \tag{1}$$

em duas partes.

Primeiro foi calculado βC e o resultado obtido armazenado nas próprias posições da matriz C:

```
C[i][j] = C[i][j] * beta
```
(2)

onde `i` e `j` são as dimensões da matriz C e `beta` é β.

Em seguida é realizada a segunda parte da equação, onde os valores obtidos em (2) são incrementados da multiplicação entre α, a matriz A e sua transposta A^T:

```
C[i][j] = C[i][j] * alpha * A[i][k] * A[j][k]
```
(3)

onde `i` e `j` são as dimensões da matriz C, A tem dimensões i por k e `alpha` é α.

3.1. OpenMP

Na implementação utilizando OpenMP, *threads* são criadas para o cálculo de (2) e ao fim desta computação as *threads* são destruídas. Em seguida, são criadas *threads* para a execução de (3) e ao fim desta as *threads* são novamente destruídas.

A implementação com OpenMP não altera o código sequencial com exceção das chamadas de suas diretivas, que na execução sequencial são tratadas como comentários. Nesta implementação, foram utilizadas as diretivas `#pragma omp parallel` para delimitação da porção de código a ser paralelizada e `#pragma omp for` para a execução das equações (2) e (3).

3.2. Pthreads

Na abordagem utilizando Pthreads, a equação (1) foi paralelizada criando e destruindo *threads* apenas uma vez. Entre a computação das equações (2) e (3), foi utilizada a estratégia de sincronização barreira.

A barreira é um mecanismo de sincronização que possibilita que várias *threads* que cooperam entre si sejam forçadas a esperar em um ponto específico até que todas tenham terminado seu processamento, antes que cada *thread* possa continuar.

Com isso, antes da execução de (3), a estratégia barreira garante que a execução de (2) esteja completa.

Ainda, devido ao esquema de passagem de parâmetros requerido pela biblioteca, foram criadas as matrizes A_copy e C_copy, cópias de A e C, respectivamente. Toda a computação foi realizada fazendo uso destas cópias.

Os dados a serem processados por cada *thread* foram divididos utilizando a seguinte formulação:

$$\texttt{D = dim/num\_threads} \tag{4}$$

onde D é a porção que será executada por cada thread, dim é o tamanho de uma das dimensões da matriz C e $num\_threads$ é o número de *threads* na execução corrente.

4. Resultados

Os resultados apresentados nessa seção foram obtidos ao executar as implementações sequencial, paralela utilizando Pthreads e paralela utilizando OpenMP.

Os testes foram executados em uma máquina com processador Intel® Core™ i7-2670QM CPU @ 2.20GHz $\times$ 8, 5,7 GB de memória, arquitetura x86_64 e sistema operacional Ubuntu 14.04 LTS. A máquina possui 4 núcleos físicos e 4 virtuais e tecnologia Hyper-Threading Intel®.

Para obtenção dos valores apresentados nas figuras abaixo, foram realizadas onze execuções para cada quantidade de *threads*, sendo que este número variou de 2^0 a 2^5, sendo que, para cada quantidade de *threads* foram descartadas a primeira execução bem como a mais lenta e a mais rápida e realizada uma média simples entre as oito execuções restantes.

O valor de *speedup*, S, é obtido através da divisão do tempo de execução do programa sequencial, T_S, pelo tempo de execução do programa paralelo, T_P:

$$S = \frac{T_S}{T_P} \tag{5}$$

.

O valor de eficiência, E, é obtido através da divisão do tempo de execução, T_P, pelo número de *threads*, P:

$$E = \frac{T}{P} \tag{6}$$

Nas figuras 1 e 2, apresenta-se os gráficos de speedup para os valores médios obtidos com as execuções sequencial, paralela com OpenMP e paralela com Pthreads do

Syrk. Na figura 2, observa-se que a queda mais brusca no *speedup* ocorre quando passa-se a utilizar 16 *threads*.

Apesar de sutil, se pode perceber um desempenho melhor com o uso de Pthreads em comparação ao uso de OpenMP.

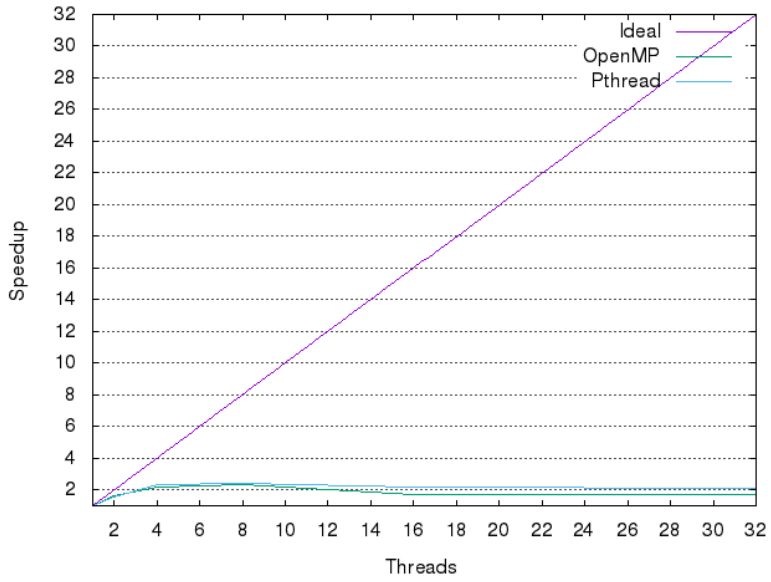

Figura 1. Gráfico de *Speedup* das execuções do Syrk.

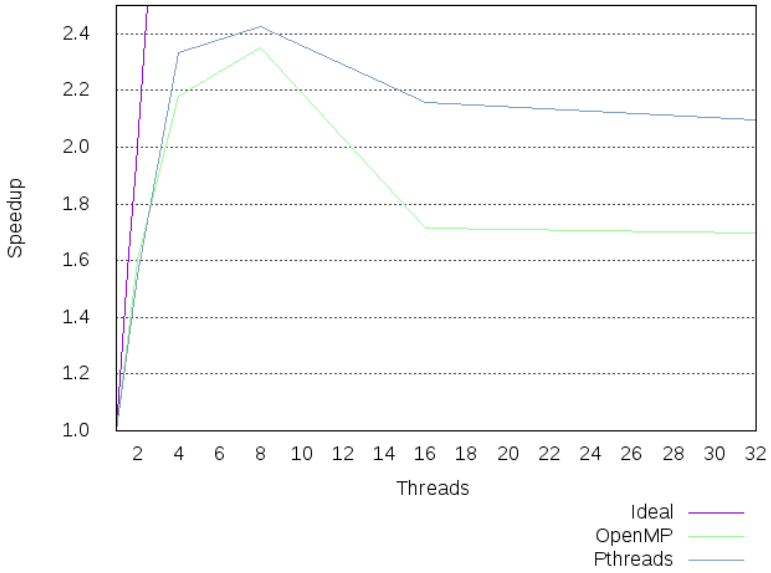

Figura 2. Gráfico de *Speedup* das execuções do Syrk – visão aproximada.

Através da figura 3, verifica-se que ocorre a diminuição na eficiência com o aumento do número de *threads*. Do mesmo modo, percebe-se uma acentuação na queda para quantidades menores de *threads*.

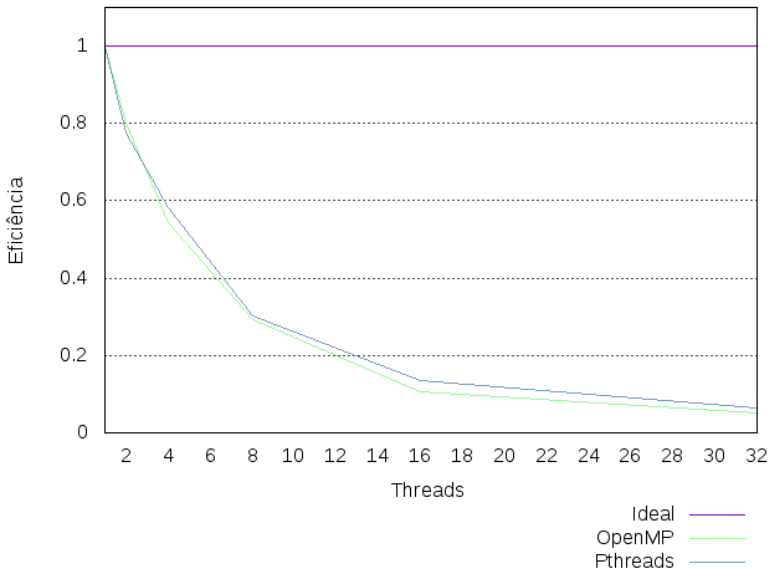

Figura 3. Gráfico de eficência

Na figura 4, observa-se que para 2 *threads* o número de instruções por ciclo de clock mantém-se bem próximo ao sequencial, para 4 *threads* o valor apresenta queda mas continua bom, sendo que há uma vantagem na implementação com Pthreads.

Já a partir de 8 *threads*, percebe-se um queda significativa no valor analisado enquanto que, aumentando o número de *threads*, as quedas continuam ocorrendo mas de maneira menos significativa.

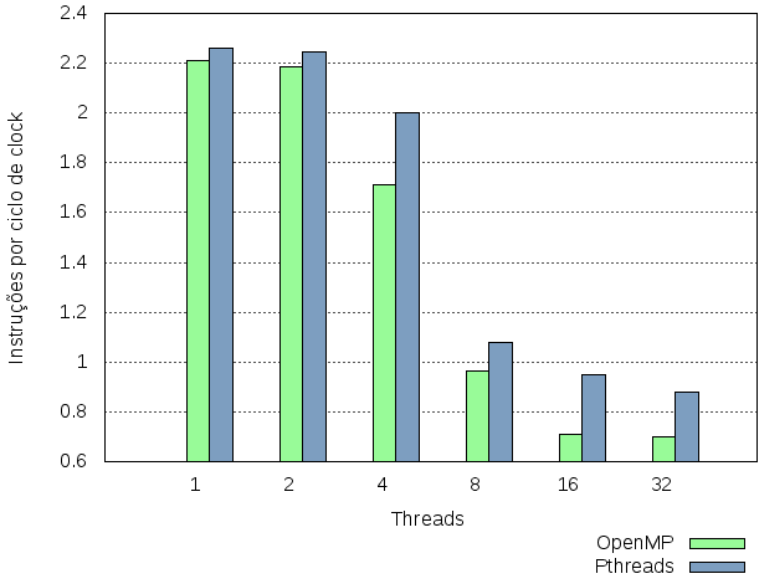

Figura 4. Intruções por ciclo de clock

A figura 5 apresenta os erros de cache e podemos observar que estes são sempre maiores quando comparados a execução sequencial no caso da implementação com

OpenMP e sempre menores no caso de Pthreads.

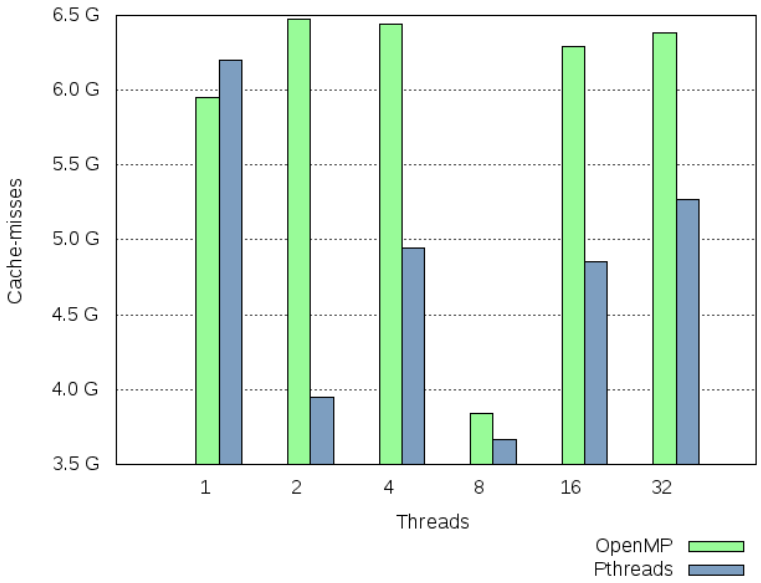

Figura 5. Erros de cache – cache-misses

Para as figuras 6 e 7 se pode obervar que como a máquina em que os testes foram executados possui 8 núcleos – 4 físicos e 4 virtuais – se percebe um aumento tanto nas mudanças de contexto quanto nas migrações de CPU a partir de 16 *threads*.

Mudanças de contexto referem-se a comutação da CPU de uma *thread* para outra. [Li et al. 2007].

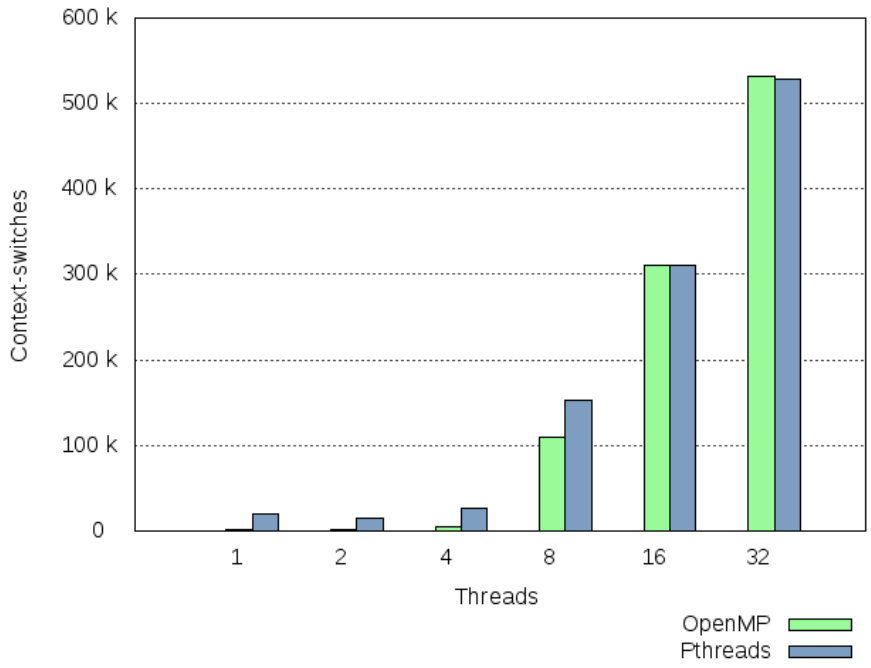

Figura 6. Mudanças de contexto – context-switches

As migrações de CPU apresentadas na figura 7 referem-se ao número de vezes em que as *threads* são movidas de um ambiente de execução para outro.

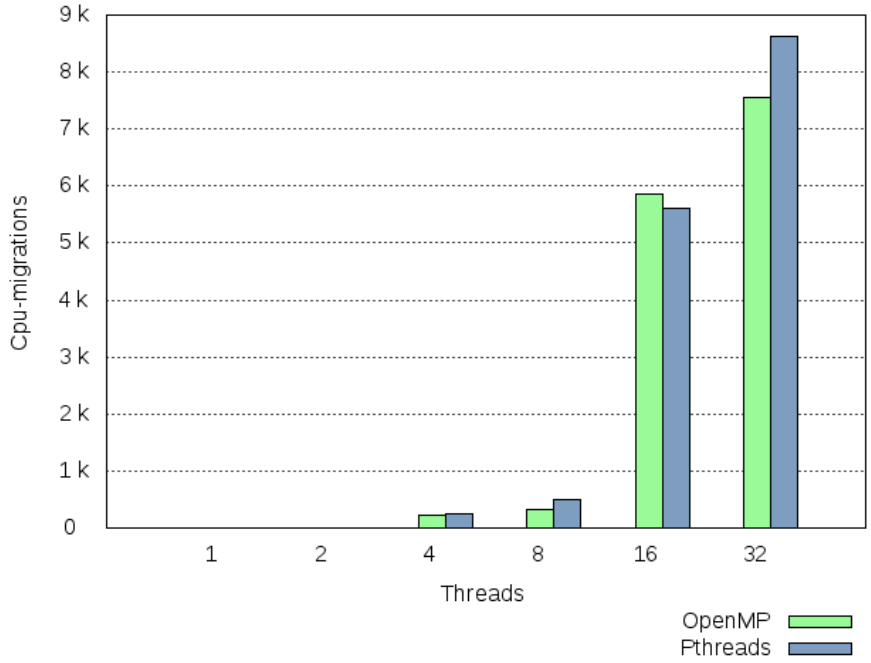

Figura 7. Migrações de CPU – cpu-migrations

5. Conclusão

Através do presente trabalho, obteve-se resultados satisfatórios utilizando-se um número de *threads* menor ou igual a oito, eis que este é o número de *threads* da máquina testada (quatro físicos e quatro virtuais). Nesse sentido, verificou-se que, com número de *threads* maior que oito, devido à criação e gerência de *threads* virtuais, o desempenho é drasticamente afetado.

Referências

(2017). *PERF(1) Linux User's Manual*, 4.4.62 edition.

Badia, R. M., Labarta, J., Marjanovic, V., Martín, A. F., Mayo, R., Quintana-Ortí, E. S., and Reyes, R. (2012). Symmetric rank-k update on clusters of multicore processors with smpss. *Advances in Parallel Computing*, 22(Applications, Tools and Techniques on the Road to Exascale Computing):657–664.

Li, C., Ding, C., and Shen, K. (2007). Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA. ACM.