Paralelismo de dados usando *PThreads* e *OpenMP* para *Covariance* do PolyBench/C 3.2

Alessandra H. Iriguti[1]

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

alehairi@gmail.com

ra88741@uem.br

Abstract. *One of the techniques to take advantage of multiprocessors is data parallelism. It generates the race hazard, because it is shared memory and, in order to solve it, synchronization is used in critical section. In the end, it is expected to obtain good speedup, that is the sequential time divided by the parallel time for a determined number of threads. The algorithm to be parallelized is the PolyBench/C 3.2's covariance using PThreads and OpenMP interfaces. It was run eleven times for each combination of parameters (algorithm type and number of threads) on a computer with eight Intel Xeon @2.00 GHz physical processors. A good speedup was not achieved. The hardware analysis with PAPI indicates that the main reason for the low speedup was synchronization and inefficient implementation of the parallel codes.*

Resumo. *Uma das técnicas de aproveitar multiprocessadores é o paralelismo de dados. Isso gera o problema de condição de corrida, pois a memória é compartilhada, e, para solucioná-la, utiliza-se sincronização na região crítica. No fim, espera-se obter um bom fator de aceleração que representa a fração do tempo sequencial sobre o paralelo para determinada quantidade de threads. O algoritmo a ser paralelizado é o covariance do PolyBench/C 3.2 e foram utilizadas as interfaces PThreads e OpenMP. Foram feitas onze execuções para cada combinação de parâmetros (tipo do algoritmo e número de threads) em uma máquina com oito processadores físicos Intel Xeon @2.00 GHz. Não foi alcançado um bom speedup. A análise do hardware com o PAPI indicam que o fator impactante ao baixo speedup foram a sincronização e uma implementação ineficiente dos códigos paralelos.*

1. Fundamentação

O crescimento da necessidade de resolver problemas cada vez maiores exigiram evoluções nas arquiteturas dos computadores. Por exemplo, mais de um processador (ou núcleo de processador) nas máquinas. Apenas isso, porém, não era suficiente, uma vez que, neste caso (de computadores multiprocessadores), não era interessante que uma CPU ficasse ociosa, enquanto outra estava sendo totalmente utilizada. Em outras palavras, a evolução dos *hardwares*, como processadores *multicore*, a fim de melhorar o poder computacional, exige o surgimento de novas estratégias de programação que façam um uso proveitoso e eficaz dos multiprocessadores. Uma dessas técnicas é o paralelismo de dados.

1.1. Paralelismo de dados

O paralelismo de dados é uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser computadas em paralelo. Em seguida, essas partes menores são recombinadas a um único conjunto. Cada tarefa executa uma mesma série de cálculos sobre diferentes dados. Dessa forma, é possível usar melhor do potencial dos processadores.

No problema de soma de dois vetores, por exemplo. Um algoritmo sem paralelização irá computar as instruções dentro de todos os *for* de maneira sequencial. Existem diversas maneiras de se paralelizar.

A abordagem adotada é a SMP (*Symetric MultiProcessors*). Nela, o acesso à memória é compartilhada, em que os processadores acessam a mesma memória. Em problemas mais complexos, onde ocorrem leituras e escritas em um mesmo local de memória ou onde a computação de um dado depende do resultado de outro, deve haver um controle sobre os dados para que os processadores produzam o resultado correto.

1.2. Condições de corrida e sincronização

A condição de corrida é uma situação que ocorre quando há regiões críticas. Uma região crítica é uma porção de código que realiza a leitura e escrita na mesma posição de memória e em que o tempo em que ocorre a leitura ou escrita das *threads* (fluxos de execução de instruções que podem ser escalonados pelo sistema operacional) podem influenciar para uma computação equivocada.

Por exemplo, um algoritmo para contar quantos números pares aparecem em uma matriz. Possivelmente, haverá uma variável *counter* para representar a quantidade de números pares. Em algum momento do código, depois de verificado e confirmado um número par, deverá haver um incremento na variável *counter*. Primeiramente, ocorre a leitura do valor contido em *counter* para então ser somado o valor 1 e, por fim, escrever o novo valor em *counter*. Se há duas *threads* computando essa tarefa, pode ocorrer condição de corrida. Em uma iteração em que as duas *threads* encontram pares, a *thread* T1 lê o valor de *counter*, suponha que seja 3. Em seguida, a *thread* T1 incrementa o valor de *counter* para 4 e, no mesmo momento, a *thread* T2 lê *counter*, porém, como T1 ainda não escreveu, T2 leu 3. A T1 escreverá 4 em *counter* e, depois, T2 também escreverá 4. No fim dessa iteração, *counter* deveria valer 5, todavia, pode não ocorrer, como no caso descrito acima.

Há diversas soluções para esse problema. Uma solução é a sincronização. A sincronização é o gerenciamento adequado de várias linhas de execução ou processos concorrentes que acessam a mesma região crítica. Por exemplo, o uso de *locks*. Um *lock* delimita as regiões críticas e faz com que apenas uma *thread* execute a região crítica por vez. Isso é conhecido como exclusão mútua (ou pelo acrônimo *mutex* para *mutual exclusion*). Isto é, uma técnica usada em programação concorrente para evitar que dois processos ou *threads* tenham acesso simultaneamente a uma seção crítica. Outro exemplo é o uso de barreiras. Uma barreira impede que uma *thread* continue a sua execução a partir da barreira enquanto as outras *threads* não alcançaram a barreira. A partir do momento que todas as *threads* chegaram na barreira, a execução a partir dela é liberado.

1.3. Fator de aceleração

Umas das métricas de desempenho em sistemas paralelos é o fator de aceleração, ou em inglês *speedup*. O fator de aceleração relaciona é a relação do tempo do algoritmo sequencial sobre o tempo do algoritmo paralelo, para um mesmo problema e de mesmo tamanho. Formalmente, o fator de aceleração S é dado por:

$$S(p) = \frac{t_s}{t_p}$$

Em que t_s é o tempo da execução do algoritmo sequencial e, t_p, o tempo da execução do algoritmo paralelo para p indica o número de processadores utilizados.

Diversos fatores podem limitar o aumento do *speedup*, tais como períodos de ociosidade e comunicação entre processadores. Além disso, nem todo código é paralelizável.

Considerando que o tempo de comunicação entre os processadores seja desprezível, na computação ideal, quanto maior o número de processadores (ou *threads*) utilizadas, maior seria o *speedup*. Ou seja, seria uma função linear, uma bissetriz do primeiro quadrante. Por exemplo, suponha o $t_s = 100$, na situação utópica, teríamos $S(2) = 2$, $S(4) = 4$, $S(8) = 8$, $S(16) = 16$ e assim sucessivamente. Todavia, não é o que acontece, já que existe um gasto de tempo (um *overhead*) para a criação, destruição das *threads*, além do *overhead* que ocorre quando há sincronização.

1.4. Interfaces de programação paralela

Para a paralelização, inúmeros fabricantes de sistemas computacionais desenvolveram suas próprias bibliotecas. Estas eram focadas em características específicas de suas arquiteturas. Dessa forma, os códigos paralelizados não eram portáteis. A fim de solucionar isso, surgiram interfaces de programação (*APIs*) paralelas que fossem eficientes e portáteis para qualquer arquitetura disponível. Duas delas são *POSIX Threads* (ou simplesmente *PThread*) e *Open Multi-Processing Application Program Interface* (*OpenMP*).

1.4.1. *PThreads*

PThreads é uma interface de manipulação de *threads* padronizada em 1995 pelo *IEEE POSIX* 1003.1c *standard*. São definidos como um conjunto de tipos e chamadas de procedimento de programação na linguagem C e estão presentes na biblioteca *pthread.h*. Uma vantagem em utilizar *PThreads* é que o custo de gerenciar *threads* é menor que gerenciar processos.

As subrotinas presentes em *PThreads*, podem ser sintetizadas em quatro grupos:

1. Gerenciamento de *Threads*: funções de criação, de destruição, de junção de *threads*, entre outras;
2. Exclusão mútua (*mutex*): rotinas que lidam com sincronização. Criação, inicialização e destruição de *mutexes*, *locking* e *unlocking*, entre outros;
3. Variáveis condicionais: rotinas que abordam comunicações entre *threads* que compartilham um *mutex*, com base nas condições especificadas pelo programador. Criar, destruir, aguardar e assinalar de acordo com valores das variáveis especificadas; e

4. Sincronização: rotinas que gerenciam leitura e escrita com base em *locks* e barreiras.

1.4.2. *OpenMP*

OpenMP é uma API com de paralelismo de programas em linguagens C, C++ e Fortran. Consiste em uma coleção de diretivas de compilação, variáveis de ambiente e de rotinas pertencentes à biblioteca *omp.h*. Esse conjunto é usado pelo programador de maneira explícita o trecho do código a ser paralelizado.

Toda diretiva é iniciada com *#pragma omp* e é aplicada em até uma declaração sucessora que deve ser um bloco estruturado. A seguir serão apresentadas apenas as construções relevantes para o propósito deste artigo.

A primeira construção a ser apresentada é a *parallel* que inicia uma execução paralela. Para determinar o número de *threads* a computarem o código delimitado pela primeira diretiva, basta usar com a variável *num_threads(p)*, em que p é a quantidade de *threads*. Ou seja, para iniciar uma execução paralela com p *threads*, utiliza-se a diretiva *#pragma omp parallel num_threads(p)*.

Outra construção é para a forma canônica de *loops*. Para ela, utiliza-se a diretiva *for* que deve ser seguida de apenas uma instrução, que no caso é um *for*. Esta pode ser utilizada juntamente à construção *simd*. Ela indica que múltiplas iterações do laço de repetição podem ser executadas de maneira concorrente usando instruções SIMD (*Single Instructions Multiple Datas*). Se há mais de um *for* associado à construção *simd*, então as iterações de todos esses *for* são transformadas em apenas uma iteração maior, onde serão executadas instruções SIMD. A construção final ficaria *#pragma omp for simd*.

Vale ressaltar que o *#pragma omp for simd* implementa implicitamente uma barreira. Isso significa que o segundo *for* só é iniciado quando todas as *threads* terminarem o primeiro *for*, e assim sucessivamente.

2. Proposta

O objetivo é verificar o *speedup*, tanto utilizando *PThreads* quanto utilizando *OpenMP*. Para isso, foram implementados para o *benchmark Covariance* duas versões paralelas, uma para cada interface.

2.1. *Covariance*

O programa *Covariance* foi retirado da coleção de *benchmarks* chamada *Polyhedral Benchmark suite*, ou, simplesmente, PolyBench. A versão utilizada é o PolyBench/C 3.2. *Covariance* calcula a covariância, uma medida estatística que mostra o quão duas variáveis estão linearmente relacionadas. Como entrada, recebe uma matriz *data* de dimensões $N \times M$, que representa os N pontos de dados com M atributos. Após a computação, retorna uma matriz *cov* simétrica $M \times M$, em que o i, j-ésimo elemento é a covariância entre i e j.

Matematicamente, covariância é definida como a média (*mean*) dos produtos dos desvios para i e j. Portanto temos:

$$cov\,(i,j) = \frac{\sum_{k=0}^{N-1} (data\,(k,i) - mean\,(i))\,(data\,(k,j) - mean\,(j))}{N-1}$$

Onde:

$$mean\,(x) = \frac{\sum_{k=0}^{N-1} data\,(k,x)}{N}$$

O código sequencial do PolyBench/C 3.2 para o cálculo da covariância segue abaixo:

```
static void kernel_covariance(int m, int n, DATA_TYPE float_n) {

        int i, j, j1, j2;

        /* Determine mean of column vectors of input data matrix */
        for (j = 0; j < _PB_M; j++) {
                mean[j] = 0.0;
                for (i = 0; i < _PB_N; i++)
                        mean[j] += data[i][j];
                mean[j] /= float_n;
        }

        /* Center the column vectors. */
        for (i = 0; i < _PB_N; i++)
                for (j = 0; j < _PB_M; j++)
                        data[i][j] -= mean[j];

        /* Calculate the m * m covariance matrix. */
        for (j1 = 0; j1 < _PB_M; j1++)
                for (j2 = j1; j2 < _PB_M; j2++) {
                        symmat[j1][j2] = 0.0;
                        for (i = 0; i < _PB_N; i++)
                                symmat[j1][j2] += data[i][j1] * data[i][j2];
                        symmat[j2][j1] = symmat[j1][j2];
                }

}
```

Em que a matriz *data* representa o $data$ presente na fórmula matemática, assim como o vetor *mean*. A matriz *symmat* representa o $cov\,(i,j)$ da equação. Como pode ser observado, há três *for* maiores, externos. Destes três, dois são duplos (dois *for* aninhados) e um triplo.

Paralelizando o código com *PThreads*, obtem-se o seguinte código abaixo:

```
void *covariance_pthread(void *argumentos) {
        int colunai, colunaf;
        int i, j, j1, j2;
        int n,m;

        struct args *a;
        a = (struct args *) argumentos;
        colunai = (int) a->colunai;
        colunaf = (int) a->colunaf;
        n = (int) a->n;
        m = (int) a->m;

        for (j = colunai; j < colunaf; j++) {
                mean[j] = 0.0;
                for (i = 0; i < _PB_N; i++) {
                        mean[j] += data[i][j];
                }
                mean[j] /= float_n;
        }
```

```
        for (i = 0; i < _PB_N; i++) {
                for (j = colunai; j < colunaf; j++) {
                        data[i][j] -= mean[j];
                }
        }

        pthread_barrier_wait (&barreira);
        for (j1 = colunai; j1 < colunaf; j1++){
                for (j2 = j1; j2 < _PB_M; j2++) {
                        symmat[j1][j2] = 0.0;
                        for (i = 0; i < _PB_N; i++) {
                                symmat[j1][j2] += data[i][j1] * data[i][j2];
                        }
                        symmat[j2][j1] = symmat[j1][j2];
                }
        }
}
```

A ideia utilizada para essa paralelização, foi dividir a mesma quantidade de colunas da matriz *data* para cada *thread*. Consequentemente, cada *thread* ficaria responsável pela computação da mesma quantidade de linhas da matriz *symmat*. Foi necessário o uso de uma barreira antes do *for* tripo. Este usa resultados do segundo *for* e, como diversas *threads* computam seus resultados, é necessário uma barreira para aguardar a finalização dessa computação para continuar a execução e calcular o resultado final.

Por fim, paralelizando com *OpenMP*, foram utilizadas as diretivas destacadas na seção 1.4.2.

```
void covariance_openmp(int m, int n, int nt, DATA_TYPE float_n) {

        #pragma omp parallel num_threads(nt)
        {
                #pragma omp for simd
                for (int j = 0; j < _PB_M; j++) {
                        mean[j] = 0.0;
                        for (int i = 0; i < _PB_N; i++) {
                                mean[j] += data[i][j];
                        }
                        mean[j] /= float_n;
                }

                #pragma omp for simd
                for (int i = 0; i < _PB_N; i++)
                        for (int j = 0; j < _PB_M; j++)
                                data[i][j] -= mean[j];

                #pragma omp for simd
                for (int j1 = 0; j1 < _PB_M; j1++)
                        for (int j2 = j1; j2 < _PB_M; j2++) {
                                symmat[j1][j2] = 0.0;
                                for (int i = 0; i < _PB_N; i++) {
                                        symmat[j1][j2] += data[i][j1] * data[i][j2];
                                }
                                symmat[j2][j1] = symmat[j1][j2];
                        }
        }
}
```

A diretiva *#pragma omp parallel num_threads(*nt*)* cria nt *threads* para executarem paralelamente o código delimitado por essa diretiva. Essas *threads* são só destruídas no fim, depois do término na execução do *for* triplo. Outra diretiva utilizada foi o *#pragma omp for simd*, para possibilitar a execução em paralelo do *for* que segue imediatamente tal diretiva.

3. Ambiente de execução

A máquina utilizada para as execuções dos algoritmos possui 8 processadores físicos e Ubuntu 15.10 (GNU/Linux 4.2.0-22-generic) como sistema operacional. O processador é Intel(R) Xeon(R) CPU E5504 @2.00GHz, com arquitetura x86, 64-bits, Little Endian. Possui três níveis de cache, em que a L1 possui $32K$, a L2 possui $256K$ e a L3 $4096K$ e possui sete contadores de *hardware*.

3.1. Estratégias de execuções

Para a coleta de resultados, foram adotadas algumas estratégias. Primeiramente, foi avaliado se o aumento do tamanho do problema influenciava no fator de aceleração, já que os tempos entre o sequencial e o paralelo apresentaria vantagem quando o tempo sequencial era consideravelmente alto (aproximadamente cinco minutos). Porém, após vários testes, conclui-se que o tamanho da entrada não influenciava no cálculo do *speedup*. Dessa forma, foi selecionado o tamanho de 2048.

Para verificar a distância da forma ideal linear, foi calculado *speedup* para 2, 4, 8, 16 e 32 *threads*. Isso foi feito para *PThread* e *OpenMP*. Além disso, para maior precisão de dados, foram realizadas onze execuções. Ou seja, foram executadas onze vezes o sequencial, onze vezes usando *PThreads* para 4, 8, 16 e 32 *threads* e onzes vezes usando *OpenMP* para 4, 8, 16 e 32 *threads*. Após as execuções, três tempos foram descartados: a da primeira execução, pois esta serve apenas para "aquecer"a *cache*; a de maior tempo e a de menor tempo, para eliminar os tempos mais dispersos do conjunto. Em seguida, foi feito a média dos oito tempos restantes. Estes foram utilizados para calcular o fator de aceleração. Com esses dados, foi feito o gráfico de *speedup* (Figura 1).

Por fim, foi utilizado a ferramenta PAPI (*Performance Application Programming Interface*) na versão $5.4.1.0$ para monitorar a relação entre o *hardware* (o ambiente de execução) e o *sofware* (o programa em questão). Com ele foram capturados os dados de número total de *cache miss* nos três níveis: L1, L2 e L3; total de instruções ponto flutuante; total de ciclos e total de instruções. Foram recolhidos esses dados e feitas as médias desses valores para cada operação (sequencial, *PThread* e *OpenMP*).

4. Resultados

Na figura 1, é possível observar o fator de aceleração para o algoritmo *covariance*. A linha pontilhada de cor lilás se refere à situação ideal, quanto maior o número de *threads* maior o *speedup*. Porém o que ocorreu estão representadas nas linhas verde (*PThread*) e azul claro (*OpenMP*). A queda no crescimento dessas duas curvas já era esperado, devido a motivos já discutidos na seção 1.3.

Pode-se observar que não foi obtido um bom fator de aceleração, uma vez que as curvas da execução paralela ficaram bem distantes da curva ideal. Utilizar *PThreads* se mostrou melhor para 2, 4 e 16 *threads*, enquanto o *OpenMP* mostrou melhor tempo para 8 *threads*. Isso se deve ao fato de que as diretivas do *OpenMP* implementam implicitamente barreiras. Para 32 *threads* já não se observa diferença e pode-se dizer que a partir desse valor, ambos algoritmos apresentariam resultados próximos.

Sobre a relação do programa com o *hardware*, o sequencial apresentou número bem superiores em todos os contadores utilizados (*cache miss* em L1, L2 e L3; instruções

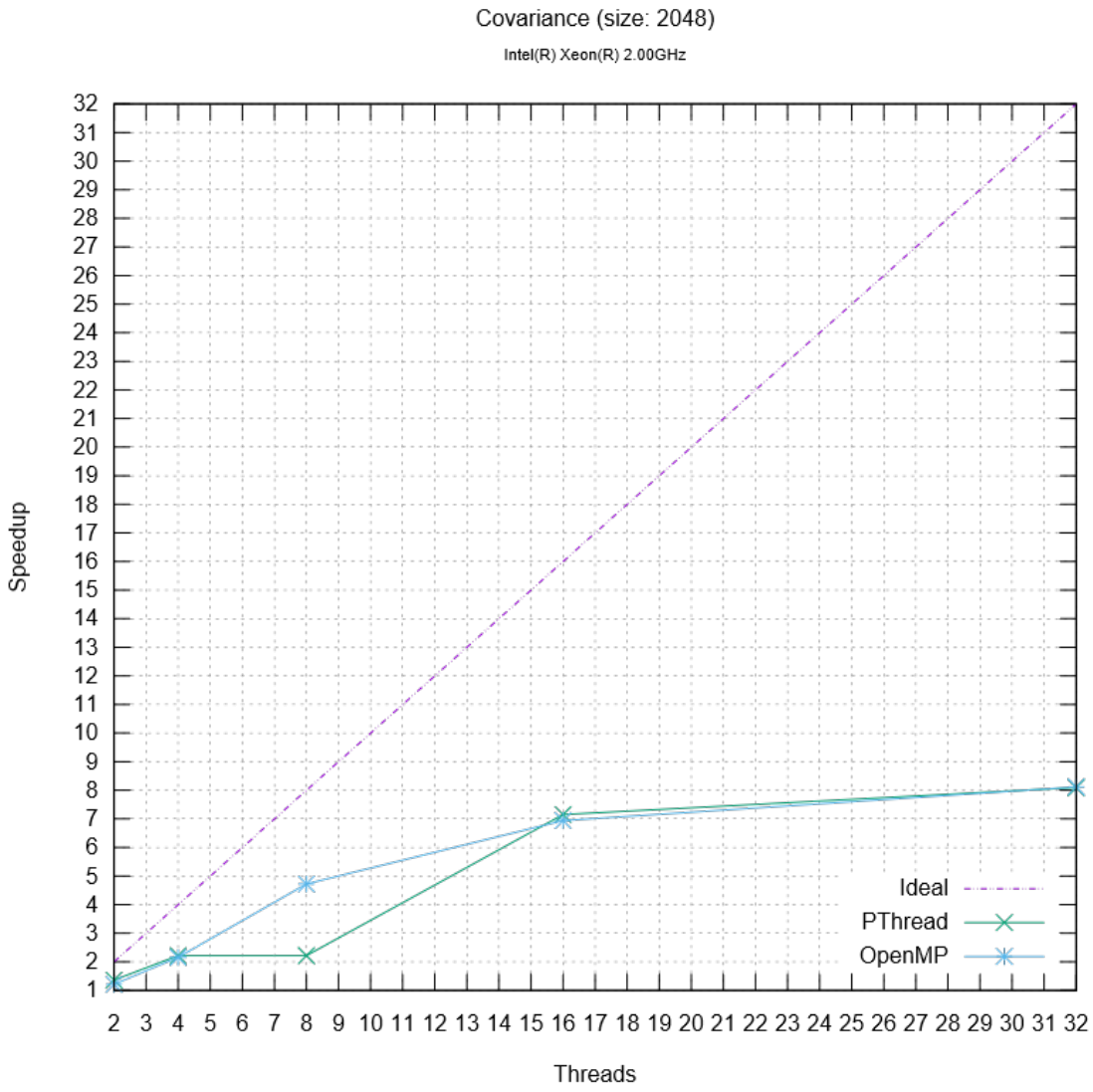

Figura 1. Speedup do *covariance*

ponto flutuante; ciclos e instruções). Para o *OpenMp*, as porcentagens dos números totais de cada contador com relação ao total do sequencial são: o número de L1 *cache miss* representa $23,12\%$; o número de L2 *cache miss* é $22,17\%$; L3 *cache miss* $21,97\%$; são $22,44\%$ de instruções PF; $23,2\%$ de ciclos e $24,36\%$ de instruções. Ou seja, os valores para *OpenMP* são por volta de 20% dos valores sequenciais. Para as porcentagens do *PThreads*, ocorrem valores próximos ao do *OpenMP*.

5. Conclusões

Como é possível observar na seção anterior (seção 4), não foi obtido um bom *speedup*. Diversos fatores poderiam ter contribuído para isso. A primeira delas seria o tamanho do problema. Foram realizados diversos testes para diversos tamanhos e, consequentemente, para diversos tempos, porém não houveram variações significativas do *speedup*. Portanto, o tamanho do problema não foi o fator que provocou o baixo *speedup*.

Um segundo fator seria excessos de *cache miss*, como acontecem na execução sequencial. Todavia, em *PThreads* e em *OpenMP*, a quantidades de *cache miss* são menores, equivalente a 20% do sequencial. Sobre os resultados obtidos utilizando o PAPI, em resumo, mesmo que *PThreads* e *OpenMP* tenham contadores menores do que o sequencial para *cache miss*, instruções de ponto flutuante, número de ciclos e de instruções, o *speedup*, consideravelmente, bom não foi atingido (não foi atingido nem um *speedup* $S(p) = 10$ para $p \geq 10$).

Portanto, pode-se afirmar que há dois possíveis fatores que, certamente, impactaram significativamente para o baixo fator de aceleração. O primeiro deles seria a necessidade de sincronização. Que possivelmente é a maior causa de *overhead* nos tempos. A barreira necessária antes de um *for* triplo pode ser o fator impactante para o baixo *speedup*, ou o próprio *for* triplo seja o fator impactante, ou ambos. E, por fim, a implementação ineficiente do código. Por exemplo, uso incorreto das diretivas do *OpenMP* e/ou a divisão ineficiente de tarefas entre as *threads* do *PThread*.

6. References

- Andrews, Gregory R. (1999) *Foundations of Multithreaded Parallel and Distributed Programming*. Editora Pearson, 1ª edição.
- Grama, Ananth; Karypis, George; Kumar, Vipin. (2003) *Introduction to Parallel Computing*. Editora Pearson, 2ª edição.
- Mendes Jr., Pedro Ribeiro. (2012) Uso de paralelismo de dados em algoritmos de processamento de imagens utilizando Haskell. Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto, 52 páginas.
- Oliveira, Rebeka G. de; Cavalcante, Sérgio V. (2010) Explorando o paralelismo de dados e de thread para atingir a eficiência energética do ponto de vista do desenvolvedor de software. In anais da XXX Congresso da SBC, páginas 1776 – 1789.
- *OpenMP 4.5 Complete Specifications*. Disponível em ¡http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf¿, acesso em 24 de maio de 2017.
- Pacheco, Peter. (2011) *An Introduction to Parallel Programming*. Editora Morgan Kaufmann, 1ª edição.
- Tutorial de *POSIX Threads Programming*. In *Lawrence Livermore National Laboratory*. Disponível em: ¡https://computing.llnl.gov/tutorials/pthreads/¿, acesso em 24 de maio de 2017.