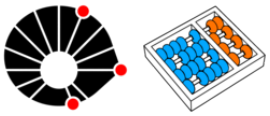


Universidade Estadual de Campinas
Instituto de Computação

MO644 - Introdução à Programação Paralela

Paralelização e análise do algoritmo de Cholesky

Alunos:

147512, Nathália Menini Cardoso dos Santos

192744, Miguel Antonio Rodriguez Santander

Professor:

Guido Araújo

26 de Junho de 2017

1 Introdução

Processos que utilizam matrizes são muito comuns em diversas áreas do conhecimento. O tempo de computação destes métodos está diretamente relacionado com a dimensão de tais estruturas matemáticas, assim, resolver os problemas de grandes matrizes significa um maior tempo de processamento.

Um problema simples, porém muito recorrente, é a resolução de um sistema linear, em que o número de operações pode se tornar relativamente grande, dependendo da aplicação em questão. Se considerarmos, por exemplo, o problema de mínimos quadrados (maiores informações em [1]), à medida em que o número de variáveis aumenta, a dimensão da matriz também aumenta, tornando o problema de estimação dos parâmetros caro computacionalmente e, desse modo, alternativas para a resolução do sistema linear devem ser empregadas.

No âmbito desse problema, métodos como a Fatoração de Cholesky são adotados. Nessa linha, realizaremos uma comparação exaustiva do método de Cholesky usando algumas ferramentas estudadas em aula (OpenMP e Pthreads). Esta comparação permitirá observar e analisar como o desempenho do algoritmo pode ser melhorado quando implementado em paralelo.

2 Fatoração de Cholesky

Sejam $A \in \mathbb{M}_n(\mathbb{R})$ uma matriz positiva-definida e um elemento $b \in \mathbb{R}^n$. Consideremos o problema de encontrar $x^* \in \mathbb{R}^m$ solução do sistema linear positivo-definido

$$Ax = b.$$

Podemos obter uma solução numérica através da Fatoração de Cholesky da matriz A, garantida pelo teorema abaixo, cuja demonstração pode ser encontrada em [2].

Teorema *Seja $A \in \mathbb{M}_n(\mathbb{R})$ uma matriz positiva definida. Então, existe uma única matriz triangular superior G, com os elementos da diagonal principal positivos, tal que $A = G^tG$.*

Desse modo, a resolução do sistema linear positivo definido poderia ser resolvido da seguinte maneira

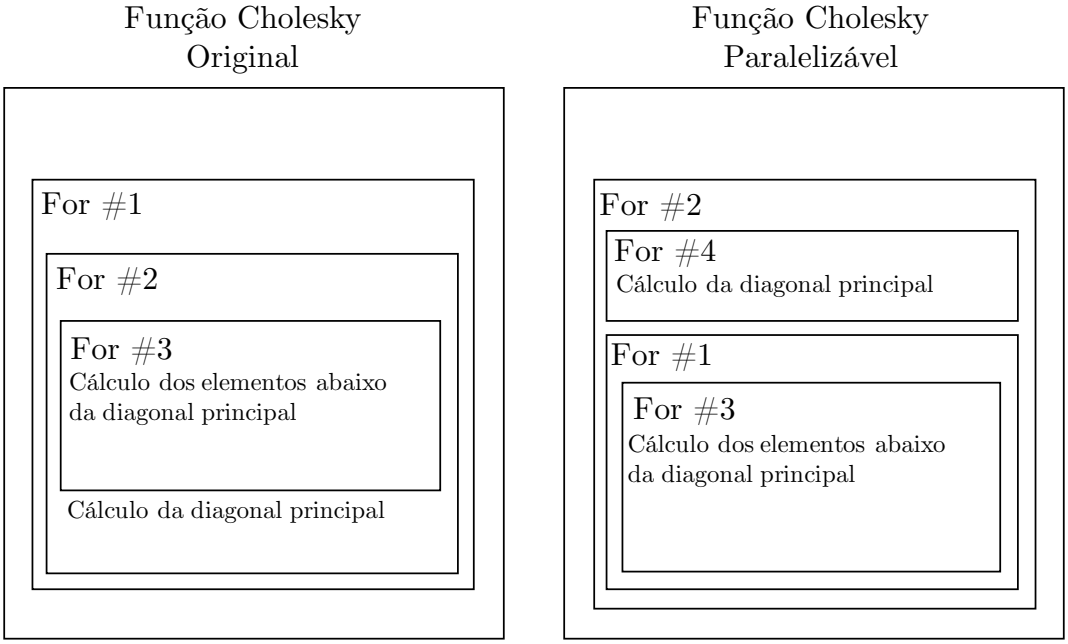
$$\begin{cases} G^t y = b \\ Gx = y \end{cases}$$

que pode apresentar resultados mais eficientes do que quando comparado com a maneira tradicional (escalonamento, por exemplo). No nosso trabalho, temos o objetivo de paralelizar o

algoritmo que realiza a fatoração de Cholesky da matriz A, e não a resolução dos sistema linear positivo definido como um todo.

3 Algoritmo

Na Figura 1, ilustramos brevemente a estrutura do algoritmo que realiza a Fatoração de Cholesky (retirado de [3]). Em uma rigorosa análise feita no código referente a Figura 1a, percebemos que o algoritmo serial calcula iterativamente as linhas de G^t (`for #1`), de modo que a linha subsequente depende do resultado obtido na iteração anterior, gerando um *loop* DOACROSS. Além disso, no segundo laço (`for #2`), que é responsável por percorrer as colunas, temos que a iteração seguinte depende do cálculo da anterior, gerando assim um *loop carried dependence*. Entretando, se trocarmos a ordem dos laços `#1` e `#2` e anteciparmos o cálculo da diagonal principal, como evidenciado na Figura 1b, não teríamos mais dependencia no *loop* interno, agora `#1`, porém, o laço externo (`#2`) ainda continua não paralelizável. Desse modo, daqui em diante, utilizaremos a versão modificada do algoritmo da Fatoração de Cholesky a fim de realizarmos uma comparação do algoritmo serial e paralelizado.

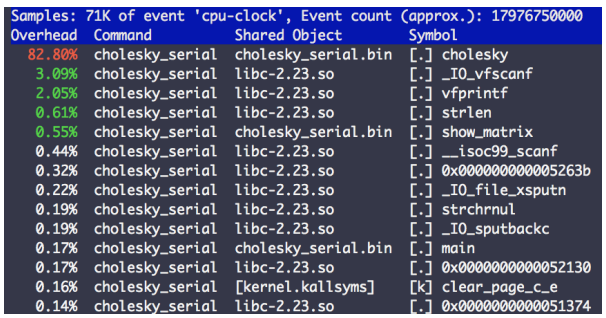


(a) Original (b) Modificado

Figura 1: Ilustração do algoritmo serial e sua versão modificada.

4 *Profile* antes da paralelização

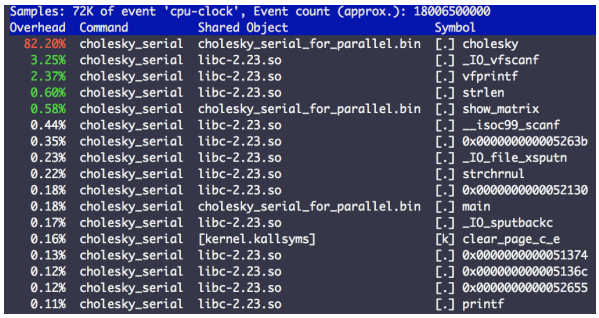
Na Figura 2, temos o *profile* para o algoritmo serial original e modificado, realizado utilizando a ferramenta perf. Pode-se notar que cerca de 82% do tempo de processamento do programa é devido a função cholesky, que é a encarregada por realizar a decomposição da matriz. Esta função será paralelizada utilizando as ferramentas OpenMP e Pthreads, de modo que as parcelas da função cholesky que serão efetivamente paralelizadas podem ser vistas na Seção 5.



Samples: 71K of event 'cpu-clock', Event count (approx.): 17976750000

Overhead	Command	Shared Object	Symbol
82.80%	cholesky_serial	cholesky_serial.bin	[.] cholesky
3.09%	cholesky_serial	libc-2.23.so	[.] _IO_vfscanf
2.05%	cholesky_serial	libc-2.23.so	[.] vfprintf
0.61%	cholesky_serial	libc-2.23.so	[.] strlen
0.55%	cholesky_serial	cholesky_serial.bin	[.] show_matrix
0.44%	cholesky_serial	libc-2.23.so	[.] __isoc99_scanf
0.32%	cholesky_serial	libc-2.23.so	[.] 0x000000000005263b
0.22%	cholesky_serial	libc-2.23.so	[.] _IO_file_xsputn
0.19%	cholesky_serial	libc-2.23.so	[.] strchrnul
0.19%	cholesky_serial	libc-2.23.so	[.] _IO_sputbackc
0.17%	cholesky_serial	cholesky_serial.bin	[.] main
0.17%	cholesky_serial	libc-2.23.so	[.] 0x0000000000052130
0.16%	cholesky_serial	[kernel.kallsyms]	[k] clear_page_c_e
0.14%	cholesky_serial	libc-2.23.so	[.] 0x0000000000051374

(a) Versão serial original



Samples: 72K of event 'cpu-clock', Event count (approx.): 18006500000

Overhead	Command	Shared Object	Symbol
82.20%	cholesky_serial	cholesky_serial_for_parallel.bin	[.] cholesky
3.25%	cholesky_serial	libc-2.23.so	[.] _IO_vfscanf
2.37%	cholesky_serial	libc-2.23.so	[.] vfprintf
0.60%	cholesky_serial	libc-2.23.so	[.] strlen
0.58%	cholesky_serial	cholesky_serial_for_parallel.bin	[.] show_matrix
0.44%	cholesky_serial	libc-2.23.so	[.] __isoc99_scanf
0.35%	cholesky_serial	libc-2.23.so	[.] 0x000000000005263b
0.23%	cholesky_serial	libc-2.23.so	[.] _IO_file_xsputn
0.22%	cholesky_serial	libc-2.23.so	[.] strchrnul
0.18%	cholesky_serial	libc-2.23.so	[.] 0x0000000000052130
0.18%	cholesky_serial	cholesky_serial_for_parallel.bin	[.] main
0.17%	cholesky_serial	libc-2.23.so	[.] _IO_sputbackc
0.16%	cholesky_serial	[kernel.kallsyms]	[k] clear_page_c_e
0.13%	cholesky_serial	libc-2.23.so	[.] 0x0000000000051374
0.12%	cholesky_serial	libc-2.23.so	[.] 0x000000000005136c
0.12%	cholesky_serial	libc-2.23.so	[.] 0x0000000000052655
0.11%	cholesky_serial	libc-2.23.so	[.] printf

(b) Versão serial modificado

Figura 2: Perf do algoritmo serial original (à esquerda) e modificado (à direita).

5 Descrição da paralelização

Na Figura 3 ilustramos as partes que foram paralelizadas do nosso algoritmo, levando em consideração o que foi discutido nas Seções 3 e 4. Mais especificamente, para cada iteracão do *loop* `#2`, paralelizamos o cálculo do elemento da diagonal principal e, com isso, computamos os elementos abaixo da diagonal, também de forma paralelizada. Tentamos realizar a paralelização de duas maneiras: (1) sem reutilizar as threads e (2) reutilizando as threads. Para o primeiro caso, foram implementadas as versões que consideram OpenMP e Pthreads, de modo que a cada iteração do laço `#2` criávamos novas threads, de forma que o tempo da criação das threads podem acarretar em uma piora na eficiência do algoritmo. Para o segundo caso, implementamos o algoritmo considerando apenas a ferramenta Pthreads, de modo que criamos uma *pool* de threads que seriam reutilizadas a cada iteração, como uma tentativa de diminuir ainda mais o tempo de execução, já que as threads não seriam mais criadas em cada iteração do laço. Além disso, também realizamos o benchmark considerando as *flags* de compilação O0 e O2, com o objetivo de verificar como se comportava a otimização com os algoritmos implementados.

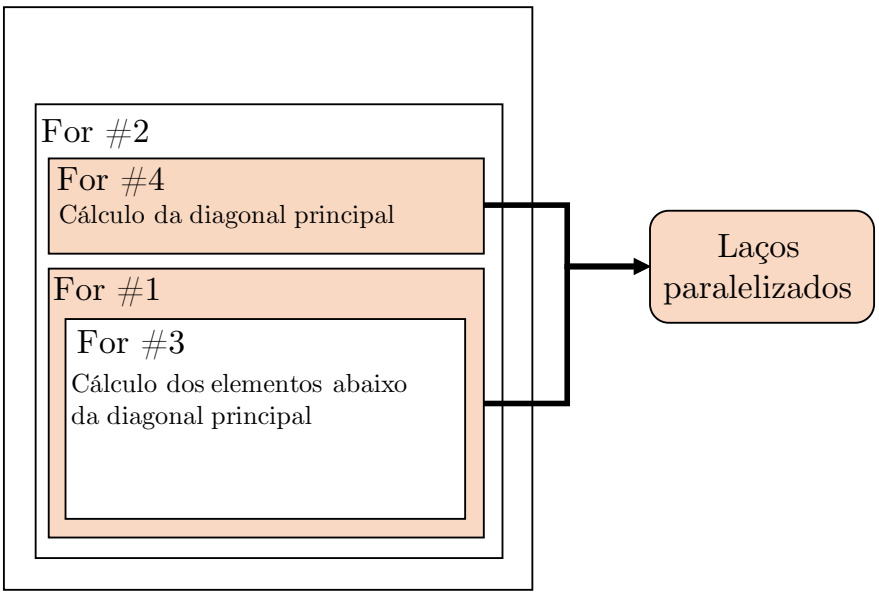


Figura 3: Estrutura da função da Fatoração de Cholesky, com destaque para as áreas paralelizadas.

Mais precisamente, para as versões que não usam a reciclagem das threads, a implementação consiste em chamar a função paralelizada em cada iteração do laço `#2`, de modo que as threads sempre sejam novamente criadas. A fim de ilustramos o desempenho dos algoritmos, denotaremos por OPENMP e PTHREAD, as abordagens OpenMP e Pthreads, respectivamente. A segunda versão, que utiliza a reciclagem e a ferramenta Pthreads, é uma implementação em que uma fila é criada no começo da função, e ela vai recebendo o trabalho a ser realizado a medida em que as iterações do laço `#2` vão avançando. Chamaremos essa implementação de PTHREAD-POOL.

6 Resultados

Para que fosse possível realizar uma comparação exaustiva das paralelizações do algoritmo de Cholesky propostas na Seção 5, criamos um benchmark com matrizes de tamanhos distintos (100, 500, 1000, 2000, 3000, 5000, 6000 e 7000) considerando os algoritmos propostos e utilizando distintos número de threads (1, 2, 4, 8, 16, 24 e 32). Para efetuar a comparação de forma que o processamento da máquina não fosse refletido nos resultados do *speedup*, decidimos executar cada algoritmo 10 vezes, de tal forma que o cálculo do *speedup* levasse em consideração a média dos tempos que as execuções demoravam. Para realizar todos os experimentos, criamos uma máquina na plataforma Azure, a qual conta com 32 núcleos, 441 GB de RAM e Ubuntu 16.04.2 LTS.

Na Figura 4 apresentamos o gráfico da variação do *speedup* com o número de threads,

considerando todas as versões paralelas que foram implementadas e *flag* O0. Pode-se observar que, conforme aumentamos a dimensão da matriz e o número de threads, mais *speedup* conseguimos, evidenciando assim a escalabilidade da implementação. Entretanto, é perceptível que a eficiência quando consideramos a versão OPENMP, sempre é superior quando comparada com as versões PTHREAD e PTHREAD-POOL. Isso pode ser explicado pelo fato de que o OpenMP é uma ferramenta que está muito otimizada para a paralelização de ciclos, de modo que qualquer implementação que utilize Pthreads deve passar por várias melhoras iterativas para poder alcançar ou superar os resultados obtidos via OpenMP. Por outro lado, ao realizar uma comparação utilizando as implementações em Pthreads, a versão PTHREAD-POOL possue um desempenho muito abaixo da PTHREAD, devido ao tempo que cada thread perde na sincronização através dos MUTEX. Esse fato será melhor discutido na Seção 7.

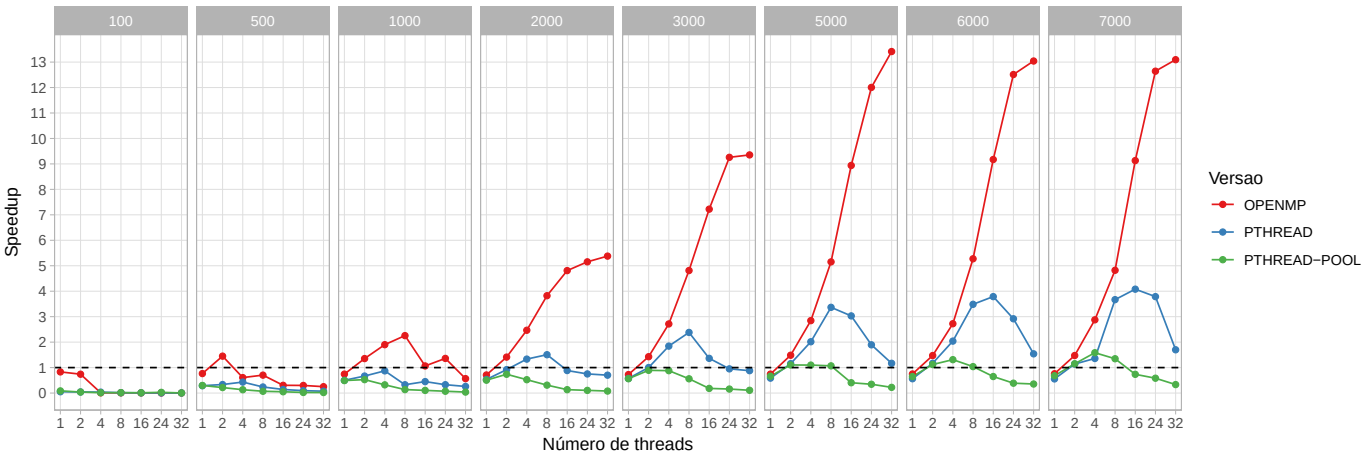


Figura 4: Variação do *speedup* com o número de threads, considerando a *flag* O0.

Já na Figura 5 mostramos a variação do *speedup* com o número de threads considerando a *flag* O2. É possível perceber que conseguimos obter valores um pouco maiores de *speedup*, do que quando consideramos a flag O0. Entretanto, percebe-se também que quando consideramos o número de threads até 24, obtemos uma redução no tempo de processamento, enquanto que quando consideramos o número de threads sendo 32, há uma queda no valor do *speedup* em alguns casos como, por exemplo, quando temos uma matriz com dimensão 5000. Novamente, observa-se melhores resultados na versão que considera OpenMP, em comparação com as versões que utilizam Pthreads. Alem disso, percebe-se também que dentre as versões que utilizam Pthreads, o algoritmo PTHREAD-POOL possui um desempenho inferior ao algoritmo PTHREAD.

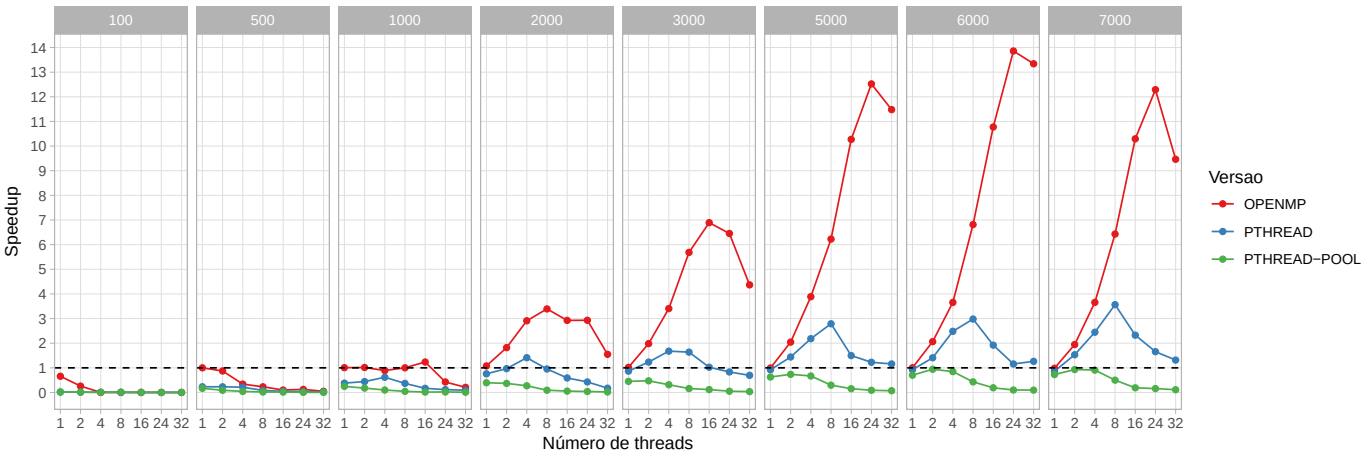


Figura 5: Variação do *speedup* com o número de threads, considerando a *flag* O2.

De modo geral, acreditamos que foi possível reduzir significativamente o tempo de execução quando consideramos as versões de OpenMP, sendo possível obter um *speedup* próximo de 13 em matrizes com dimensões 5000, 6000 e 7000. Entretanto, quando possuimos matrizes com dimensões menores como, por exemplo, 100, 500 e 1000, a redução no tempo computacional não existe ou não é muito perceptível, devido ao custo temporal de criar e distribuir o trabalho para as threads ser muito grande para a baixa quantidade de dados que serão calculados.

Nas implementações que utilizam Pthread foi possível obter uma melhora muito menor do que a obtida por meio das versões OpenMP, sendo possível obter um *speedup* próximo de 3 em matrizes com dimensões de 5000, 6000 e 7000. Por outro lado, na implementação utilizando a técnica da criação de uma *pool* de threads, não conseguimos nenhum *speedup* significativo, esse fato pode ser explicado pelo alto tempo que a implementação gasta nas seções críticas dos laços.

7 *Profile* depois da paralelização

Na Figura 6 podemos observar a análise do profile utilizando a ferramenta *perf*. A Figura 6a diz respeito a análise do perfilamento da paralelização realizada utilizando a versão OPENMP, de modo que pode-se observar que aproximadamente 50% do tempo é utilizado pela função implementada, e cerca de 44% do tempo é devido a *library* OpenMP.

Observando a Figura 6b, referente ao *profile* da implementação PTHREAD, pode-se ver que cerca de 50% do tempo é devido à função *rest_worker_parallel*, que é responsável por calcular todos os números abaixo da diagonal principal da matriz. Por outro lado, na Figura 6c, observa-se que cerca de 88% do tempo é referente a uma SYSCALL, a qual acreditamos que seja chamada pelos MUTEX, fato que poderia explicar o porquê a implementação não tem um

bom desempenho, devido a grande quantidade de sincronizações necessárias para essa função.

```
Samples: 207K of event 'cpu-clock', Event count (approx.): 51918250000
Overhead  Command          Shared Object         Symbol
  47.31%  cholesky_openmp  cholesky_openmp.bin   [.] cholesky._omp_fn.1
   9.78%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011ceb
   8.96%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011ce9
   6.89%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011b2b
   6.20%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011b29
   5.47%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011ced
   3.88%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011b2d
   1.77%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011ce0
   1.21%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011b20
   1.19%  cholesky_openmp  libc-2.23.so          [.] _IO_vfscanf
   1.04%  cholesky_openmp  libc-2.23.so          [.] vfprintf
   1.04%  cholesky_openmp  cholesky_openmp.bin   [.] cholesky._omp_fn.0
   0.54%  cholesky_openmp  libgomp.so.1.0.0      [.] 0x0000000000011be0
   0.22%  cholesky_openmp  libc-2.23.so          [.] strlen
```

(a) Versão OPENMP

```
Samples: 128K of event 'cpu-clock', Event count (approx.): 32080500000
Overhead  Command          Shared Object          Symbol
  50.18%  cholesky_pthrea  cholesky_pthreads.bin  [.] rest_worker_parallel
   6.44%  cholesky_pthrea  [kernel.kallsyms]      [k] __lock_text_start
   4.81%  cholesky_pthrea  [kernel.kallsyms]      [k] perf_event_alloc
   2.32%  cholesky_pthrea  [kernel.kallsyms]      [k] inherit_event.isra.96
   2.31%  cholesky_pthrea  [kernel.kallsyms]      [k] memset_erms
   2.25%  cholesky_pthrea  [kernel.kallsyms]      [k] default_send_IPI_mask_sequ
   1.94%  cholesky_pthrea  [kernel.kallsyms]      [k] __mutex_init
   1.69%  cholesky_pthrea  libc-2.23.so           [.] _IO_vfscanf
   1.59%  cholesky_pthrea  libc-2.23.so           [.] vfprintf
   1.47%  cholesky_pthrea  [kernel.kallsyms]      [k] mutex_lock
   1.38%  cholesky_pthrea  [kernel.kallsyms]      [k] mutex_unlock
   1.12%  cholesky_pthrea  [kernel.kallsyms]      [k] smp_call_function_many
   0.70%  cholesky_pthrea  [kernel.kallsyms]      [k] unmap_page_range
   0.61%  cholesky_pthrea  [kernel.kallsyms]      [k] flush_tlb_mm_range
```

(b) Versão PTHREAD

```
Samples: 14M of event 'cpu-clock', Event count (approx.): 3678264500000
Overhead  Command          Shared Object                  Symbol
  88.48%  cholesky_pthrea  [kernel.kallsyms]              [k] native_queued_spin_lock_slowpath
   2.66%  cholesky_pthrea  [kernel.kallsyms]              [k] futex_wake
   2.10%  cholesky_pthrea  [kernel.kallsyms]              [k] futex_wait_setup
   2.04%  cholesky_pthrea  [kernel.kallsyms]              [k] _raw_spin_lock
   1.50%  cholesky_pthrea  libpthread-2.23.so             [.] pthread_mutex_lock
   0.83%  cholesky_pthrea  cholesky_pthreads_prodcon.bin  [.] worker
   0.60%  cholesky_pthrea  libpthread-2.23.so             [.] pthread_mutex_unlock
   0.41%  cholesky_pthrea  [kernel.kallsyms]              [k] __lock_text_start
   0.15%  cholesky_pthrea  libpthread-2.23.so             [.] __lll_lock_wait
   0.15%  cholesky_pthrea  [kernel.kallsyms]              [k] __unqueue_futex
   0.11%  cholesky_pthrea  [kernel.kallsyms]              [k] finish_task_switch
   0.10%  cholesky_pthrea  [kernel.kallsyms]              [k] get_futex_value_locked
   0.08%  cholesky_pthrea  [kernel.kallsyms]              [k] futex_wait_queue_me
   0.07%  cholesky_pthrea  [kernel.kallsyms]              [k] entry_SYSCALL_64_after_swapgs
```

(c) Versão PTHREAD-POOL

Figura 6: Perf dos algoritmos OPENMP, PTHREAD e PTHREAD-POOL.

8 Dificuldades encontradas

A primeira dificuldade encontrada foi o fato de que o algoritmo escolhido estava implementado de uma maneira em que não era possível realizar a paralelização, de modo que foi necessário uma reorganização de laços e operações, a fim de torná-lo paralelizável. A segunda dificuldade que encontramos, foi implementar a versão do Pthreads que considerasse uma *pool* de threads, com o objetivo de reutilizá-las a cada iteração, como uma tentativa de minimizar o tempo de execução. Por fim, tivemos uma grande dificuldade de obter melhores resultados utilizando

Pthreads e, por mais que tentássemos reduzir o tempo gasto em criar as threads, não foi possível encontrar uma implementação capaz de reduzir significativamente o tempo computacional.

9 Conclusão

De modo geral, acreditamos que a paralelização do algoritmo Cholesky propiciou uma redução siginificativa no tempo de execução. Esse fato é muito mais evidente e presente na versão que considera OpenMP.

Em relação as versões que utilizam Pthreads, tentamos diversas vezes obter uma melhora nos resultados, ou seja, aumentar o *speedup* nas execuções. Apesar de grandes esforços e diversas tentativas de diferentes algoritmos, não foi possível obter uma versão que considerasse Pthreads e obtivesse melhores resultados. Na primeira versão implementada em Pthreads, não ocorre a reutilização das threads e, na segunda versão (THREAD-POOL), embora tenhamos implementado um algoritmo que seja capaz de reutilizar as threads criadas, há uma quantidade muito grande de sincronizações, o que faz com o speedup caia significativamente.

10 Referências Bibliográficas

[1] Harrell, F. *"Regression Modeling Strategies"*, Spring, 2001.

[2] Pulino, P. *"Álgebra Linear e suas Aplicações: Notas de Aula"*, 2004.

[3] `https://rosettacode.org/wiki/Cholesky_decomposition#C` [Acessado em 22/05/2017]