# Superalocação de threads em implementação com MPI e OpenMP do método Hopmoc bidimensional

**Frederico L. Cabral[1], Gabriel Costa[1], Carla Osthoff[1], Thiago Teixeira[1], Sanderson L. Gonzaga de Oliveira[2]**

[1]Laboratório Nacional de Computação Científica (LNCC)
Av. Getúlio Vargas, 333. Quitandinha – Petrópolis-RJ

{fcabral, gcosta, osthoff, tteixeira}@lncc.br

[2]Departamento de Ciência da Computação – Universidade Federal de Lavras (UFLA)
Caixa Postal 3037 – 37.200-000 – Lavras – MG – Brasil

sanderson@ufla.br

***Abstract.*** *This article shows experiments with the Hopmoc method using a strategy of over-allocating threads. This paper evaluates the results of implementing the strategy under different metrics. Using the scheme, we show that the reduction of implicit barriers in OpenMP allowed us to allocate a large number of threads in a multicore system. As a result, the use of the strategy improved the performance of the parallel numerical method.*

***Resumo.*** *Neste artigo, utilizamos uma estratégia de superalocação de threads com experimentos com o método numérico Hopmoc. São apresentados os resultados da implementação da estratégia sob diferentes métricas. Por meio desta estratégia, mostramos que a redução das barreiras inerentes ao OpenMP podem possibilitar uma maior alocação de threads em uma máquina composta de processadores multicores, de forma a melhorar o desempenho do método numérico paralelo.*

## 1. Introdução

Problemas de diversas áreas são modelados por equações diferenciais parciais. Exemplos incluem aplicações científicas e da engenharia, presentes em áreas como a aeroespacial, petrolífera, ambiental, engenharia química e biomédica, geociência e outras. Alguns exemplos mais específicos seriam o uso de equações diferenciais parciais para descrever problemas de fluxos característicos de áreas como geologia e oceanografia, ou a propagação de algum poluente em meios diversos como água e o ar.

Considere a equação diferencial parcial de advecção-difusão da seguinte forma $u_t + vu_x = du_{xx}$, em que $v$ e $d$ são constantes positivas de velocidade e difusão, respectivamente. O método Hopmoc é utilizado na solução de equações de advecção-difusão com convecção dominante. Primeiramente, o método calcula o processo de convecção, em cada semipasso de tempo, eliminando o segundo termo do lado esquerdo da equação original, que se reduz a $u_t = du_{xx}$. Em seguida, o método calcula o processo de difusão, ao dividir a malha computacional em dois subconjuntos que

serão calculados alternadamente em cada semipasso de tempo, um com uma abordagem explícita e outro com uma abordagem implícita. Apesar de ser um método implícito, essa abordagem elimina a necessidade da utilização de um resolutor de sistemas de equações lineares.

Métodos com essa natureza possibilitam o paralelismo e outras estratégias capazes de aumentar o desempenho da aplicação, permitindo a produção de resultados mais rapidamente e escalas espaciais cada vez menores. Isso contribui para um estudo dos resultados com mais acurácia e mais próximo da realidade.

Em publicação recente [Cabral et al. 2019], o método Hopmoc bidimensional foi aplicado com uma versão híbrida OpenMP/MPI em um sistema multicore. Foram utilizados múltiplos processos com o padrão MPI e múltiplas threads com a biblioteca OpenMP. Nomeamos essa estratégia como MPI/OMP-EWS.

Neste artigo, mostramos que a estratégia MPI/OMP-EWS, ao eliminar as barreiras implícitas do padrão OpenMP e reduzir o spin time, possibilita uma implementação eficiente de um grande número de threads por núcleos físicos. Assim, neste presente trabalho, foi aplicada uma estratégia de se utilizar uma superalocação de threads no método Hopmoc bidimensional implementado pela versão híbrida OpenMP/MPI, ou seja, com a estratégia MPI/OMP-EWS. Especificamente, analisamos o desempenho de uma estratégia de superalocação de recursos paralelos aplicada em uma versão com modelo de paralelização híbrido MPI e OpenMP em um ambiente computacional multicore.

O restante deste texto está organizado da seguinte forma. Na Seção 2, são apresentados os trabalhos relacionados. Na Seção 3, são apresentados detalhes sobre o método Hopmoc bidimensional implementado com OpenMP/MPI. Na Seção 4, é mostrada uma breve distinção das duas variações do método Hopmoc utilizadas nos testes realizados para esse trabalho. Na Seção 5, os resultados são apresentados. Por fim, na Seção 6, são apresentadas as conclusões e também as perspectivas futuras de investigação.

## 2. Trabalhos relacionados

Cabral et al. [Cabral et al. 2017] mostraram o desempenho do método Hopmoc em um ambiente computacional de memória compartilhada Intel® MIC Xeon® Phi utilizando o modelo de programação OpenMP. No artigo, foi comparada uma versão ingênua do método Hopmoc com uma estratégia que utilizou a técnica *explicit task chunk* e conclui-se que essa segunda alternativa leva a uma redução de *overhead time* e também de *spin time*.

Cabral et al. [Cabral et al. 2018a] propuseram uma abordagem do método Hopmoc utilizando a técnica *explicit work sharing* e diretivas OpenMP mais eficientes. Os autores chamaram essa abordagem de OMP-EWS. Os testes foram realizados em uma máquina com processador Intel® Xeon® Phi.

Cabral et al. [Cabral et al. 2018b] estudaram o desempenho da metodologia apresentada no trabalho anterior [Cabral et al. 2018a] para uma versão do método Hopmoc com *total variation diminishing* e para diversas sistemas de memória compartilhada baseados na arquitetura Intel® Xeon®. Costa et al. [Costa et al. 2019] apresentaram um estudo comparativo entre três versões unidimensionais do método Hopmoc (ingênuo, OMP-EWS e baseado em MPI). Nesse trabalho, foram avaliadas métricas como *spin time* e tempo de CPU.

Cabral et al. [Cabral et al. 2019] compararam três métodos numéricos bidimensionais em diferentes processadores Intel®. Nesse trabalho, foi utilizada uma versão híbrida do método Hopmoc bidimensional baseado em MPI e OMP-EWS.

Muitas publicações investigaram implementações híbridas de MPI e OpenMP. Jeffers et al. [Jeffers et al. 2016] avaliaram implementações paralelas de uma solução explícita por diferenças finitas da equação de Poisson. Semelhante ao nosso estudo, os autores compararam uma implementação híbrida (uma abordagem MPI/OpenMP e uma implementação baseada em MPI com threads) com uma implementação baseada em MPI pura em uma arquitetura manycore. O estudo empregou uma malha de tamanho 3.000 $\times$ 3.000 usando 1, 2, 4 e 8 nós simultaneamente, em que cada nó era um acelerador Intel® Xeon® Phi Knights Landing. Os autores concluíram que a versão híbrida produziu melhores resultados do que a implementação baseada em MPI puro.

Em outro trabalho, Bassi et al [Bassi et al. 2016] apresentaram um estudo sobre uma implementação híbrida OpenMP/MPI para uma aplicação que fez uso do método de Galerkin descontínuo. Os autores realizaram testes em diferentes máquinas, além de também variar a combinação entre número de threads e número de processos possíveis. Os autores concluíram que os arranjos possíveis entre número de threads e número de processos geram impactos significativos no desempenho final do método paralelo.

Diener et al. [Diener et al. 2017] apresentaram um estudo sobre as oportunidades de otimização do acesso à memória em uma aplicação híbrida com MPI e OpenMP. Nesse artigo, os autores buscaram otimizar a localidade de acesso à memória por meio de duas estratégias: melhora manual no código fonte da aplicação com questões relacionadas à localidade de memória, e pela utilização do ambiente Adaptative MPI (AMPI).

Koleva-Efremova [Koleva-Efremova 2019] comparou o desempenho e a escalabilidade da API MPI-2 pura com a implementação híbrida MPI-2 e OpenMP em execuções realizadas no supercomputador Avitohol. O autor usou dois testes de benchmark da Intel®: operações de ping-pong e comunicação unilateral. A implementação híbrida MPI/OpenMP proporcionou melhor consumo de memória em comparação com a versão MPI-2 pura.

## 3. Versão híbrida do método Hopmoc baseada em MPI e OpenMP

O Hopmoc é um método numérico para solução de equações de advecção-difusão com convecção dominante, que se baseia no método Hopscotch (que por sua vez apresenta um esquema geral para solução de equações diferenciais parciais parabólicas ou elípticas de segunda ordem) e no método das características modificado que permite a separação de variáveis por meio de curvas características. O método Hopmoc divide a malha computacional em dois conjuntos distintos e atualiza de forma explícita um primeiro conjunto de incógnitas no primeiro semi-passo de tempo e de forma implícita um segundo conjunto de incógnitas no segundo semi-passo de tempo. Os dois conjuntos de incógnitas são alternados a cada passo de tempo, a fim de eliminar a necessidade de se utilizar um sistema de equações lineares a cada passo no tempo. Dessa forma, o código do método Hopmoc possui um laço de repetição principal *while*, em que cada iteração representa um passo no tempo. Internos a esse laço de repetição principal, há um conjunto de laços de repetições *for* responsáveis por realizar os cálculos explícitos e implícitos alternadamente, de forma a computar os dois semi-passos de tempo de cada passo de tempo. No algoritmo

1, mostra-se uma visão geral do pseudocódigo do método Hopmoc.

```
1  begin
2      Processo aloca sua seção de malha
3      while condição do
4          for condição do
               // Aplicação do método das características
                  modificado
5          end
6          for condição do
               // Primeiro semi-passo explícito
7          end
8          for condição do
               // Primeiro semi-passo implícito
9          end
10         for condição do
               // Segundo semi-passo explícito
11         end
12         for condição do
               // Segundo semi-passo implícito
13         end
14     end
15 end
```

**Algorithm 1:** Visão geral de pseudocódigo do método Hopmoc.

A versão híbrida do método Hopmoc é uma estratégia que faz uso tanto de threads OpenMP quanto de processos MPI para dividir a carga de trabalho e permitir uma paralelização eficiente [Cabral et al. 2019]. A malha de entrada é inicialmente dividida em seções menores, de forma igualitária. Cada uma dessas partições de malha é designada a um processo disponível, de forma permanente até o fim da execução. Esse processo garante o primeiro nível de paralelismo, fazendo uso exclusivo de processos. Em seguida, cada processo divide a sua carga de trabalho contida dentro do laço de repetição principal *while* entre as threads disponíveis. Isso se dá pela paralelização de todos os laços de repetição *for* contidos dentro do laço de repetição principal. Esse processo garante o segundo nível de paralelismo, fazendo uso exclusivo de threads, e completa a estratégia híbrida.

## 4. Duas implementações do método Hopmoc

Nesta seção, são apresentadas as duas implementações do método Hopmoc. As duas variações se diferenciam apenas na forma como o paralelismo por threads é aplicado. Assim, esta seção foi dividida em duas partes. A primeira, apresentada na subseção 4.1, é destinada a explicar a implementação *naïve* do método Hopmoc. Na subseção 4.2, é mostrada a versão híbrida do método Hopmoc, baseada em MPI e OpenMP.

### 4.1. Implementação *naïve*

O paralelismo a nível de threads desta implementação foi baseada em uma das estratégias mais simples possíveis: o uso de diretivas padrões OpenMP. Dentro do laço de repetição

principal *while*, cada laço de repetição interno *for* é imediatamente precedido por uma diretiva OpenMP **#pragma omp parallel for**. Essa diretiva é responsável por paralelizar de maneira implícita e automática o laço de repetição no qual ela é aplicada, ao criar uma região paralela exclusiva para essa estrutura de repetição. No Algoritmo 2, mostra-se um esquema de pseudocódigo da paralelização utilizada na implementação *naïve*.

```
1 begin
2 |  ...
3 |  #pragma omp parallel for
4 |  for condição do
  |  |  // realiza trabalho
5 |  end
6 |  ...
7 end
```

**Algorithm 2:** Trecho de pseudocódigo da implementação paralela *naïve*.

### 4.2. Estratégia MPI/OMP-EWS

A implementação do método Hopmoc bidimensional por meio da estratégia MPI/OMP-EWS é baseada no algoritmo implementado em uma publicação recente [Cabral et al. 2019]. Essa abordagem conta com um modelo de paralelização constituído de dois mecanismos principais: a divisão explícita de trabalho (EWS) e a sincronização entre threads adjacentes (AdjSync).

O mecanismo EWS consiste em dividir a malha, que o processo em questão é responsável de processar, de maneira explícita e igualitária entre as threads disponíveis. Isso é realizado de forma que, a cada thread, é designado um segmento de malha que deverá trabalhar até o fim da execução.

O mecanismo AdjSync é responsável pela sincronização entre as diversas threads disponíveis no processo. Com essa abordagem, a sincronização não espera que todas as threads alcancem o mesmo ponto de sincronização, como é realizado na implementação *naïve*. A sincronização é realizada somente entre threads adjacentes. Isso significa que cada thread é sincronizada somente com suas duas threads adjacentes, uma thread anterior e uma thread posterior.

No Algoritmo 3, mostra-se um esquema de pseudocódigo da paralelização utilizada na implementação por meio da estratégia MPI/EWS-OMP. Antes de cada laço de repetição interno *for*, a thread sinaliza as suas vizinhas um bloqueio de execução (veja a linha 3 no Algoritmo 3). Ao completar o laço de repetição, a thread remove esse bloqueio e fica em espera ocupada até que suas vizinhas também façam o mesmo (veja as linhas 6 e 7 no Algoritmo 3).

## 5. Resultados e análise

Os experimentos apresentados nesse artigo foram realizados em um computador com dois processadores Intel® Xeon® CPU E5-2698 v3 com frequência de 2,3GHz, cada um composto de 16 núcleos físicos, com *hyperthreading* desabilitado e com memória total de 115.097.784 KB. O sistema operacional nessa máquina é o CentOS Linux versão 7.3.1611 (*Core*).

```
1 begin
2 |   ...
3 |   lock
4 |   for condição do
  |   |   // realiza algum trabalho
5 |   end
6 |   unlock
7 |   Espera pelo unlock das threads vizinhas
8 |   ...
9 end
```

**Algorithm 3:** Trecho de pseudocódigo da paralelização pela estratégia MPI/OMP-EWS do método Hopmoc.

As simulações foram realizadas com uma malha bidimensional de $10^5 \times 10^5$ pontos, com variação no espaçamento da malha de $10^{-5}$. Foram realizadas 1.000 iterações em cada execução do método Hopmoc bidimensional. As execuções para as simulações que constituem os resultados apresentados nesse artigo tiveram acesso exclusivo ao hardware utilizado: não houve outras tarefas em execução ao mesmo tempo. Foram realizadas execuções com uma thread, com duas threads e em números pares até 32 threads. Após esse número de threads, foram feitas execuções de 32 em 32 threads até 2.496 threads.

Na Figura 1, são mostrados os resultados de speedup e LLC miss (Last Level *cache miss*, que representa a quantidade de *cache miss* no último nível da memória *cache*), para execuções das duas implementações avaliadas neste trabalho. Até aproximadamente 500 threads, o speedup das duas estratégias é muito próximo (≈6x) com leve superioridade da implementação com a estratégia MPI/OMP-EWS. Com execuções com mais de 500 threads, o speedup da implementação pela estratégia MPI/OMP-EWS aumenta com execuções a partir de aproximadamente 1.000 threads para mais de 25x, enquanto o speedup da implementação *naïve* continuou com aproximadamente 6x.

Apesar de terem sido feitas execuções com até 2.496 threads, foram mostradas nas figuras os resultados de execuções com um número menor de threads. Isso foi feito para que as figuras tivessem uma melhor apresentação, porque a partir do número de threads mostradas nas figuras, o comportamento dos resultados não difere.

No gráfico que mostra os resultados de LLC miss na Figura 1, são observados resultados muito próximos ($\approx 300 \cdot 10^6$) entre as duas abordagens com execuções com até 500 threads. Com execuções acima de 500 threads, a taxa de *cache miss* na memória LLC com a implementação *naïve* continua a aumentar. Por outro lado, com execuções entre 500 e 700 threads, a taxa de *cache miss* na memória LLC com o código com a estratégia MPI/OMP-EWS passa a diminuir significativamente, chegando a ficar abaixo de $10 \cdot 10^6$ em execuções com mais do que 700 threads. Com execuções com a implementação baseada na estratégia MPI/OMP-EWS com mais do que 500 threads, possivelmente, as frações de malha começam a ficar tão pequenas que conseguem entrar e se manter no mesmo nível da memória *cache*. A superalocação de threads cria seções de malha tão pequenas que são capazes de serem alocadas de forma eficiente no mesmo nível de memória *cache*, em proporção que faz com que o custo computacional de criar e escalonar

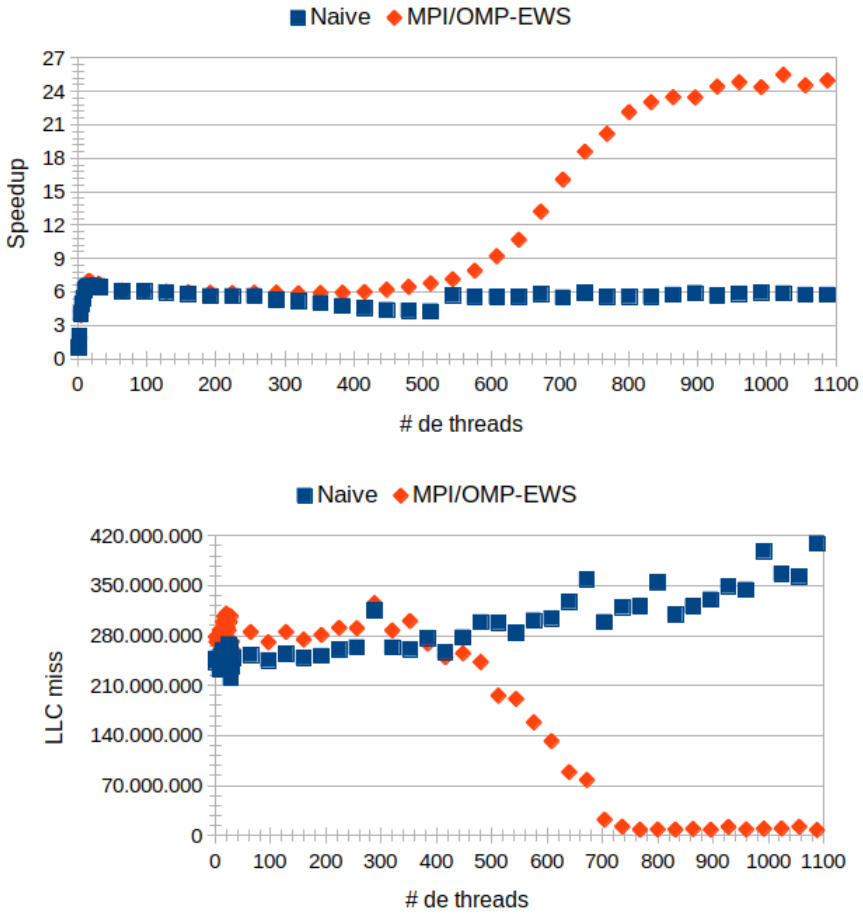


Figure 1. Speedups e LLC miss de execuções de implementações *naïve* e MPI-OMP-EWS do método Hopmoc bidimensional com até 1.100 threads.

essas threads seja muito pequeno em relação ao tempo total de execução.

Na Figura 2, são mostrados os resultados de DRAM Bound de execuções de implementações *naïve* e MPI-OMP-EWS com até 900 threads. As taxas de *cache miss* e *cache hit* nem sempre refletem as respectivas taxas de perda e ganho de desempenho de um algoritmo paralelo[1]. As métricas de ”bound” são as que realmente importam. Além de outros fatores, os resultados de DRAM Bound das execuções das implementações *naïve* e MPI-OMP-EWS são também consequência da taxa de LLC miss, mostrada na Figura 1.

Como pode ser observado nos gráficos mostrados na Figura 1, a taxa de *cache miss* na memória LLC da implementação com a estratégia MPI/OMP-EWS torna-se baixa com execuções com mais de 700 threads. Até com esse número de threads, o speedup da implementação é de ≈16x. O speedup da implementação passa a ser maior que 25x com execuções com mais de 1.000 threads. Portanto, o ganho de speedup de 16x para 25x nessa diferença de 300 threads não é decorrente de alta taxa de *cache hit* na memória LLC. Com isso, nas execuções com um número de threads de 700 a 1.000, supomos que

[1]https://software.intel.com/content/www/us/en/develop/articles/cache-miss-rates-in-intel-vtune-amplifier-xe.html

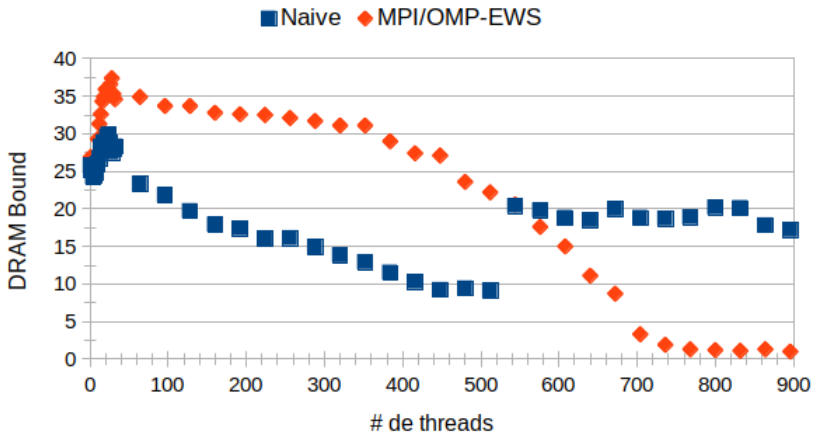


Figure 2. Resultados de DRAM Bound de execuções de implementações *naïve* e MPI-OMP-EWS do método Hopmoc bidimensional com até 900 threads.

a taxa de *cache hit* passa a ser maior na memória L2, e depois na memória L1.

O Intel® VTune™ Amplifier possui a habilidade de utilizar unidades de monitoramento de desempenho em CPUs da Intel® para contar eventos de hardware e utilizar esses eventos para indicar problemas de desempenho[1]. A forma mais comum de se realizar isso é pela análise de tipo General Exploration. As métricas do General Exploration não incluem taxas de *cache hit* e *cache miss* em memórias L1 e L2 porque a caracterização top-down no General Exploration tenta encontrar gargalos no hardware que causem problemas de desempenhos *reais*. No Intel® VTune™ Amplifier, foram abstraídas as contagens reais de *cache miss* e as substituíram por métricas L1/L3 e DRAM Bound. Extraímos as medidas de memórias *cache* L1, L2, L3 e DRAM Bound (esta última mostrada na Figura 2). Todavia, essas medidas não ajudam a explicar as taxas de *cache miss* nas memórias cache L1 e L2. Isso porque os limites não ocorrem necessariamente por conta de *cache miss* ou *cache hit*, mas sim de um conjunto de eventos possíveis.

Na Figura 3, são mostrados os resultados da razão entre Spin Time e tempo de CPU, e da razão entre Overhead Time e tempo de CPU. O Spin Time é o tempo em barreiras de sincronização, ou seja, o tempo em que a execução do código fica contida até que todas as threads estejam sincronizadas e prontas para continuar a trabalhar. O Overhead Time é o tempo em rotinas próprias do processador, que envolvem o processamento de threads, como por exemplo a criação e o escalonamento de threads. O tempo de CPU é a métrica que representa o tempo total de CPU em processamento efetivo, ou seja, é o tempo que não é nem Spin Time nem Overhead Time, mas sim o processamento em cálculos e procedimentos próprios do método.

Observa-se na Figura 3 que a razão entre Spin Time e tempo de CPU na implementação *naïve* é sempre superior à razão na implementação MPI/OMP-EWS. Também, observa-se que, com execuções com mais do que 500 threads, os perfis dos desempenhos mudam. O gráfico da razão entre Overhead Time e tempo de CPU mantém sempre um perfil de crescimento dos valores, acentuado após aproximadamente de 500

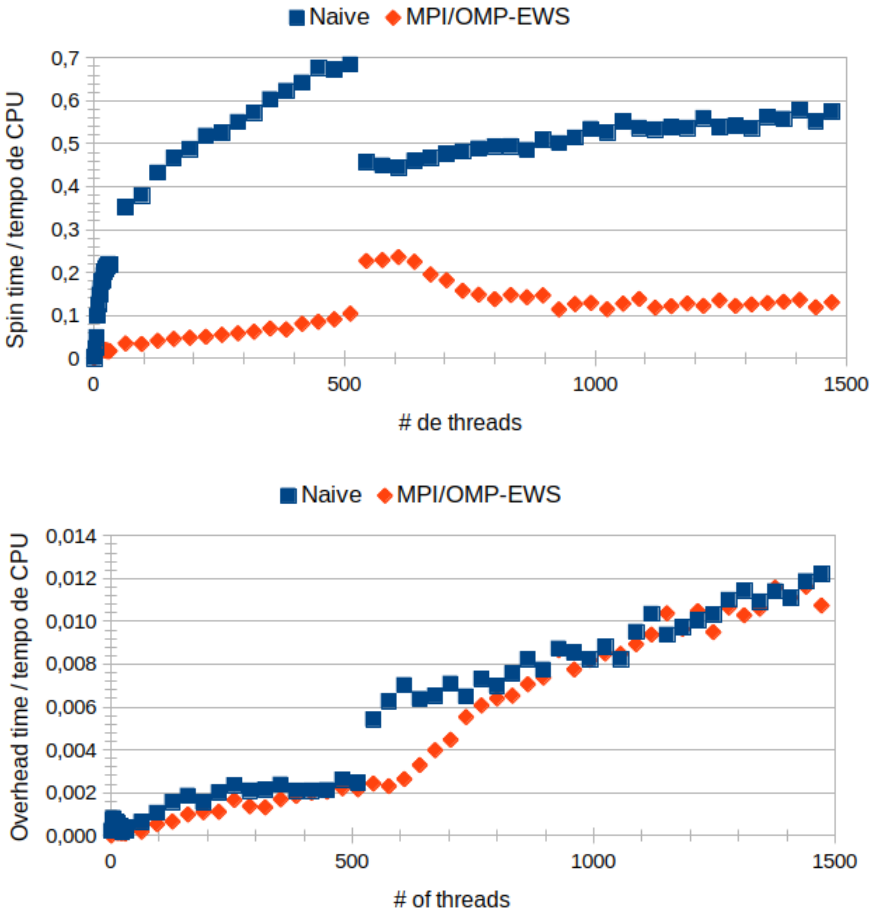


Figure 3. Razões de Spin Time por tempo de CPU e Overhead Time por tempo de CPU de execuções das implementações *naïve* e MPI-OMP-EWS do método Hopmoc bidimensional com até 1.500 threads.

threads alocadas.

Como a implementação *naïve* não faz uma segunda divisão da malha dentro de cada processo, já que a diretiva **#pragma omp parallel for** é responsável pela paralelização, o efeito de superalocação de threads não é percebido. Por outro lado, isso é perceptível na implementação com a estratégia MPI/OMP-EWS que, além de dividir a malha explicitamente entre processos, também o faz entre as threads.

Com o gráfico da razão entre Overhead Time por tempo de CPU mostrado na Figura 3, mostra-se que o custo computacional de criar, alocar e escalonar uma grande quantidade de threads não aumentou significativamente o tempo total de execução: para aproximadamente 500 a 2.000 threads, o tempo de overhead representa menos de 2% do tempo em processamento efetivo do método. Esse resultado aponta para a efetividade da superalocação de threads na estratégia MPI/OMP-EWS: a diminuição dos segmentos de malha até sua adequação ao tamanho da memória *cache* compensa o tempo com a superalocação de threads (preempção e mudança de contexto entre as threads).

Outro resultado que aponta para a efetividade da estratégia apresentada neste

trabalho é mostrado no gráfico de tempo de CPU da Figura 4. São observados dois comportamentos distintos de tempo de CPU para as duas versões do método Hopmoc avaliadas neste trabalho. Com execuções com menos de 500 threads, as duas curvas apresentam um perfil muito semelhante, com tendência de subida em conformidade com execuções com maior número de threads. Entretanto, com execuções com mais de 500 threads, as duas curvas apresentam diferenças. Na implementação *naïve*, ocorre uma queda com descontinuidade e em seguida a curva volta a subir com menor inclinação. Na implementação com a estratégia MPI/OMP-EWS, ocorre uma queda contínua e em seguida os resultados se estabilizam com execuções com mais do que 700 threads.

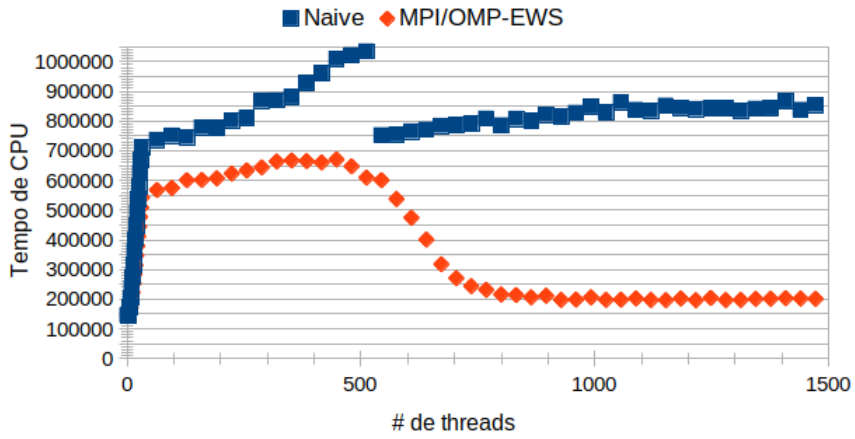


Figure 4. Tempo de CPU de execuções com as implementações *naïve* e MPI/OMP-EWS do método Hopmoc bidimensional com até 1.500 threads.

Esses resultados reafirmam e estão de acordo com aqueles apresentados nos gráficos de speedup e LLC miss mostrados na Figura 1. Ao passo que ocorre uma redução do *cache miss* no último nível de cache, há concomitantemente uma diminuição significativa do tempo de CPU para a implementação do método Hopmoc com a estratégia MPI/OMP-EWS, que leva diretamente em um aumento de speedup, como pode ser observado também na Figura 1. Por não contar com uma segunda divisão explícita no interior de cada processo como na implementação MPI/OMP-EWS, a implementação *naïve* é menos sensível à redução de *cache miss* e consequentemente de seus benefícios.

## 6. Conclusões e Trabalhos Futuros

A obtenção de altos speedups em métodos numéricos para resolução de equações diferenciais parciais em máquinas de alto desempenho não é tarefa trivial. Com a utilização da estratégia MPI/OMP-EWS no método Hopmoc bidimensional em execuções em uma máquina composta de 32 núcleos físicos, foi obtido um speedup de 7x em execução com 32 threads. Com a utilização também de superalocação de threads, foi obtido speedup de 25x, quase alcançando-se um speedup igual ao número de núcleos físicos da máquina.

Basicamente, a superalocação de threads consiste em definir um número de threads maior do que a quantidade de threads disponíveis no sistema. Apesar de que, à primeira vista, a ideia pareça impossível de produzir bons resultados, uma vez que

o tempo de overhead criado com a alocação e escalonamento (ou seja, preempção e mudança de contexto) de threads prejudicaria o desempenho, os valores obtidos nos experimentos mostrados neste trabalho apontam para bons resultados quando utilizado em conjunto com a estratégia MPI/OMP-EWS aplicada em uma implementação do método Hopmoc bidimensional.

A estratégia MPI/OMP-EWS foi projetada para reduzir as barreiras implícitas do padrão OpenMP. A utilização dessa estratégia no método Hopmoc bidimensional em conjunto com superalocação de threads revelou a possibilidade de se extrair um maior desempenho no paralelismo e de utilizar de forma eficiente os recursos físicos de um sistema multicore. Nossos trabalhos futuros incluem investigações e continuidade do estudo com outros método numéricos e em outras arquiteturas, como na microarquitetura Cascade Lake e no supercomputador SDumont.

## References


Bassi, F., Colombo, A., Crivellini, A., and Franciolini, M. (2016). Hybrid openmp/mpi parallelization of a high–order discontinuous galerkin cfd/caa solver. In *7th European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS Congress*, pages 7992–8012.

Cabral, F. L., Gonzaga de Oliveira, S. L., Osthoff, C., Costa, G. P., Brandão, D. N., and Kischinhevsky, M. (2019). An evaluation of MPI and OpenMP paradigms in finite-difference explicit methods for PDEs on shared-memory multi-and manycore systems. *Concurrency and Computation: Practice and Experience*, page e5642.

Cabral, F. L., Osthoff, C., Costa, G. P., Brandão, D., Kischinhevsky, M., and Gonzaga de Oliveira, S. L. (2017). Tuning up the TVD-HOPMOC method on Intel MIC Xeon Phi architectures with Intel Parallel Studio tools. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 19–24. IEEE.

Cabral, F. L., Osthoff, C., Costa, G. P., Gonzaga de Oliveira, S. L., Brandão, D., and Kischinhevsky, M. (2018a). An OpenMP implementation of the TVD–Hopmoc method based on a synchronization mechanism using locks between adjacent threads on Xeon Phi(TM) accelerators. In *International Conference on Computational Science*, pages 701–707. Springer.

Cabral, F. L., Osthoff, C., Souto, R. P., Costa, G. P., Gonzaga de Oliveira, S. L., Brandão, D., and Kischinhevsky, M. (2018b). Fine-tuning an OpenMP-based TVD–Hopmoc method using Intel® Parallel Studio XE Tools on Intel® Xeon® architectures. In *Latin American High Performance Computing Conference*, pages 194–209. Springer.

Costa, G., Cabral, F., and Osthoff, C. (2019). Otimização do método hopmoc 1d com auxílio das ferramentas intel parallel studio. In *Anais Estendidos do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 41–48, Porto Alegre, RS, Brasil. SBC.

Diener, M., White, S., Kale, L. V., Campbell, M., Bodony, D. J., and Freund, J. B. (2017). Improving the memory access locality of hybrid mpi applications. In *Proceedings of the 24th European MPI Users' Group Meeting*, page 11. ACM.

Jeffers, J., Reinders, J., and Sodani, A. (2016). *Intel Xeon Phi Processor High Performance Programming – Knights Landing Edition*. Morgan Kaufmann, Burlington, MA, 2 edition.

Koleva-Efremova, V. (2019). Testing performance and scalability of the pure mpi model versus hybrid mpi-2/openmp model on the heterogeneous supercomputer avitohol. volume 793 of *Studies in Computational Intelligence (SCI)*, pages 93–105, Sofia, Bulgaria. Annual Meeting of the Bulgarian Section of SIAM - BGSIAM 2017 - Advanced Computing in Industrial Mathematics, Springer.