Implementação de overthread em uma aplicação de método numérico HOPMOC através de um algoritmo híbrido MPI/OpenMP

Gabriel Costa[1], **Thiago Teixeira**[1], **Frederico L. Cabral**[1], **Carla Osthoff**[1]

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

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

Abstract. *This article presents a study of an overthread strategy applied to the Hopmoc numerical method, which consists of allocating a number of processes and threads higher than the number of available physical cores. The results obtained are presented and analyzed through different metrics. By the overthread strategy, we showed that it was possible to increase the performance of the application in a parallel environment beyond the maximum physical cores capacity of the machine, and that this gain is possible due to the elimination of native OpenMP synchronization barriers.*

Resumo. *Este artigo apresenta um estudo de uma estratégia de overthread aplicada ao método numérico Hopmoc, que consiste em alocar um número de processos e threads superior à quantidade de núcleos físicos disponíveis. Os resultados obtidos são apresentados e analisados por diferentes métricas. Por meio da abordagem de overthread mostramos que foi possível aumentar o desempenho da aplicação em ambiente paralelo para além da capacidade máxima de cores físicos da máquina, e que esse ganho é possível devido a eliminação das barreiras de sincronização nativas do OpenMP.*

1. Introdução

Atualmente um considerável número de problemas de diversas áreas do conhecimento podem ser representados e descritos através de equações diferenciais parciais. Esses problemas contemplam fenômenos da engenharia e da produção científica, presentes em setores como o aeroespacial, petrolífero, 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.

Considerando a equação diferencial parcial de convecção-difusão da seguinte forma $u_t + vu_x = du_{xx}$, onde v e d são constantes positivas de velocidade e difusão, respectivamente, o método numérico Hopmoc, destinado à solução de equações diferenciais parciais de problemas de convecção-difusão realiza primeiramente o cálculo do processo de convecção, para cada semipasso no tempo, eliminando o segundo termo do lado esquerdo da equação original, que se reduz a $u_\tau = du_{xx}$, onde τ é chamada linha característica.

Em seguida é feito o cálculo do processo de difusão, dividindo-se a malha discretizada em dois subconjuntos que serão calculados alternadamente para cada semipasso no tempo, um com uma abordagem explícita e outro com uma abordagem implícita, o que irá dispensar a necessidade da utilização de um solver para sistemas de equações.

Métodos dessa natureza são contemplados com a possibilidade de otimização via paralelismo e outras estratégias capazes de aumentar o desempenho da aplicação, permitindo a produção de resultados em mais curto espaço de tempo e com maiores magnitudes nos dados de entrada (o que contribui para uma solução mais acurada e próxima da realidade). O **Intel Parallel Studio XE** é um conjunto de ferramentas da Intel para análise de código que investiga instruções em tempo de execução, levanta dados, e com isso traça um perfil acerca da execução em questão. Essa ferramenta facilita o processo de desenvolvimento de código, uma vez que é capaz de indicar limitadores de desempenho que mitigam o sucesso dos resultados, e com isso consegue direcionar os desenvolvedores na produção de novas versões da aplicação, que são capazes de aplicar um paralelismo mais eficiente, menos limitado, com melhor uso dos recursos de hardware disponíveis.

O método Hopmoc conta com diversas versões e variantes desenvolvidas em anos de pesquisa e desenvolvimento. Este trabalho apresenta um estudo de uma estratégia aplicada em uma dessas versões, o Hopmoc **MPI/OMP-EWS** bidimensional, que faz uso tanto de múltiplos processos através do padrão **MPI**, quanto de múltiplas *threads* a partir da biblioteca **OpenMP** para alcançar um paralelismo mais eficiente. A estratégia é caracterizada por um aumento de desempenho quando há uma alocação de recursos paralelos (processos e *threads*) que excedem o total fisicamente disponível no hardware. Através da estratégia de *overthread* mostramos que foi possível aumentar o desempenho da aplicação em ambiente paralelo para além da capacidade máxima de *cores* físicos da máquina, e que esse ganho foi possível devido a eliminação das barreiras de sincronização nativas do **OpenMP**.

O restante do trabalho conta com a seguinte disposição: A Seção 2 apresenta trabalhos e contribuições anteriores do método Hopmoc em publicações internacionais e também no **WSCAD-WIC**, além de mencionar brevemente outros trabalhos que se relacionam com o tema. A Seção 3 explica com mais detalhes a aplicação híbrida do Hopmoc utilizada neste estudo, e também faz uma breve distinção das duas variações do Hopmoc **MPI/OMP** utilizadas nos testes. A Seção 4 é responsável por apresentar os resultados obtidos e discuti-los. Por fim a Seção 5 se limita a apresentar as conclusões acerca do estudo e também as perspectivas futuras.

2. Trabalhos relacionados

Cabral et al. [Cabral et al. 2017] mostraram o desempenho do método Hopmoc 1D em um ambiente computacional de memória compartilhada Intel® MIC Xeon® Phi utilizando o modelo de programação OpenMP. No artigo, foi comparada uma versão *naïve* 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*. Posteriormente contatou-se que tal estratégia não poderia ser estendida para maiores dimensões, então os mesmos autores [Cabral et al. 2018a] propuseram uma abordagem

do método Hopmoc 2D utilizando a técnica *explicit work sharing* e diretivas **OpenMP** mais eficientes. Os autores chamaram essa abordagem de **OMP-EWS**.

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 diversos 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 (*naïve*, **OMP-EWS** e baseado em **MPI**). Nesse trabalho, foram avaliadas métricas como *spin time* e tempo de CPU.

Para verificar se estratégia **OMP-EWS** apresentaria as mesmas vantagens em outros algoritmos similares, Cabral et al. [Cabral et al. 2019] compararam três métodos numéricos bidimensionais em diferentes processadores Intel®. Nesse trabalho, foram utilizadas versões híbridas **MPI** e **OMP-EWS** dos métodos Hopmoc, Diferenças Finitas Totalmente Explícita para as equações do Calor e de Laplace bidimensionais.

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.

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.

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 **Adaptive MPI (AMPI)**.

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(a matriz de de entrada que representa todos os pontos a serem calculados) 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.

Foram desenvolvidas duas variações da abordagem híbrida do método Hopmoc. As duas variações se diferenciam apenas na forma como o paralelismo por *threads* é aplicado. A primeira, apresentada na subseção 3.1, é destinada a explicar a implementação *naïve* do Hopmoc **MPI/OMP**. Na subseção 3.2, é mostrada a versão **EWS** do método Hopmoc **MPI/OMP**.

3.1. 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.

3.2. Implementação 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* será designado um segmento de malha no qual ela 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 2, mostra-se um esquema de pseudocódigo da paralelização utilizada na implementação por meio da estratégia **MPI/OMP-EWS**. Antes de cada laço de repetição interno *for*, a *thread* sinaliza as suas vizinhas um bloqueio de execução (veja a linha 3 no Algoritmo 2). 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 2).

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

4. Resultados e análise

Os experimentos apresentados neste artigo foram realizados em um computador com quatro processadores **Intel® Xeon® Platinum 8270 CPU @ 2.70GHz**, cada um composto de 26 núcleos físicos conectados por um canal de comunicação **Intel UPI**, com *hyperthreading* habilitado e com memória total de 1510 GB. Cada núcleo executa 2 *threads*, resultando em um total de 104 núcleos e 208 *threads*. O sistema operacional nessa máquina é o **Red Hat Enterprise Linux Server 7.6 (Maipo)**.

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 neste artigo tiveram acesso exclusivo ao *hardware* utilizado: não houve outras tarefas em execução ao mesmo tempo. Ou seja, os resultados não sofreram interferência de outros processos, apenas do sistema operacional. Foram realizadas uma execução serial com um processo e uma *thread*, em seguida com

quatro processos e uma *thread* por processo, e depois aumentando sempre uma *thread* por processo até alcançar o limite de 52. Alcançado o limite o número de processos passou a ser aumentado em 4 a cada execução, finalizando com 72 processos e 3744 *threads*.

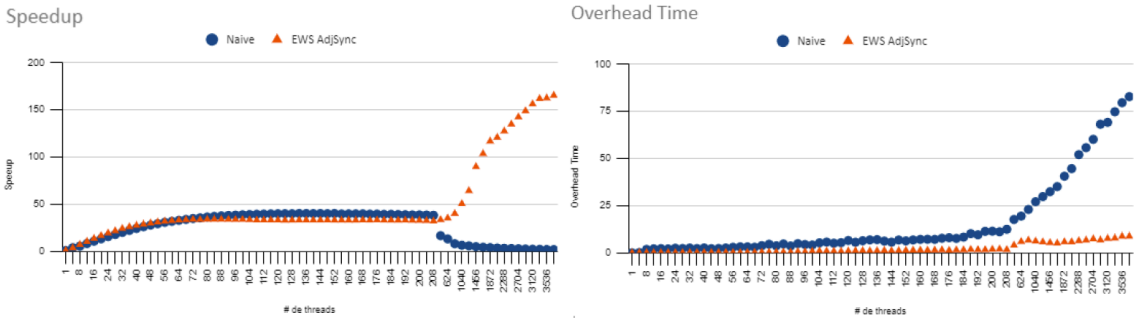

Figura 1. Gráfico apresentando o *speedup* e *overhead* time da execução.

Os gráficos apresentados na Figura 1 mostram à esquerda o *speedup* da aplicação a medida em que aumenta o numero de *threads* e à direita o tempo gasto em *overhead* (alocação e escalonamento de *threads*), a medida em que aumenta o número de *threads*. Como pode ser visto no gráfico à esquerda, ao comparar a execução do código *naïve* com o código da estratégia **EWS AdjSync**, as curvas de *speedup* apresentam perfil similar até 208 *threads*, quando, após essa marca, o *speedup* do código *naïve* sofre uma queda à medida que o da estratégia **EWS AdjSync** tem um ganho expressivo.

O gráfico da esquerda da figura 1 apresenta o *speedup* da aplicação a medida em que aumentamos o numero de *threads* em uma maquina que tem um total de 104 núcleos físicos. Podemos observar que a partir de 624 *threads* o *speedup* aumenta de forma significativa e que a partir de 1872 *threads*, a aplicação passa a ter *speedup* superior aos 104 núcleos físicos, que pode ser explicado pela diminuição do *overhead*, conforme apresentado no gráfico da direita. O tempo de *overhead* foi coletado com a ferramenta **Intel VTune** que faz parte do suíte de ferramentas **Intel Parallel Studio**. Essa métrica apresenta o tempo gasto com algumas funções importantes durante a execução do código, como a alocação de *threads* e o escalonamento delas. Como pode ser visto no gráfico, o tempo de *overhead* aumenta em ambas as execuções conforme aumenta o número de *threads*, mas no caso da execução *naïve*, após 208 *threads*, o *overhead* aumenta expressivamente quando comparado ao aumento da execução **EWS AdjSync**.

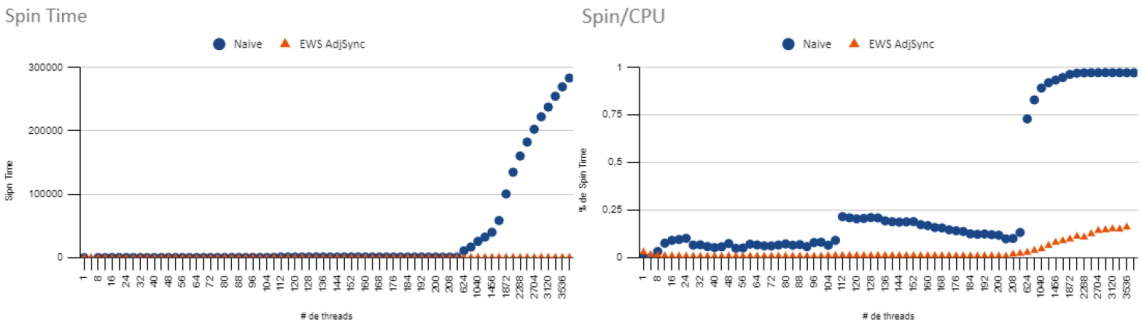

Figura 2. Gráfico apresentando o *spin time* e o *spin time* comparado ao tempo de CPU da execução.

Os gráficos da Figura 2 apresentam à esquerda o *spin time* a medida em que aumenta o numero de *threads* e à direita o *spin time* dividido pelo tempo de CPU a medida em que aumenta o numero de *threads*. O *spin time*, coletado pela ferramenta **Intel VTune**, é uma métrica que representa o tempo gasto em barreiras de sincronização entre as *threads*. Ao final de cada iteração do *loop* principal as *threads* bloqueiam temporariamente suas execuções até que todas estejam no mesmo ponto, para só então prosseguir para próxima iteração. Essa barreira é eliminada na abordagem **EWS AdjSync**, através do mecanismo de sincronização adjacente apresentado na subseção 3.2. Pode ser visto um comportamento compatível com os gráficos da Figura 1, onde os valores de *spin time* das duas versões ficam próximos até 208 *threads*, porém, a partir deste número, o *spin time* do código *naïve* aumenta expressivamente enquanto o da estratégia **EWS AdjSync** permanece quase desprezível.

O gráfico de *spin time/CPU time* da Figura 2 apresenta a razão entre o *spin time* e o tempo total gasto na execução do código (CPU time). Nele é possível observar que desde os primeiros testes a porcentagem de *spin time* em relação ao tempo de CPU da versão *naïve* é superior ao da versão **EWS AdjSync**. Entretanto, a partir de 208 *threads* esse valor sobe abruptamente e continua a manter tendência de crescimento até quase atingir a totalidade do tempo de CPU na versão *naïve*. Apesar de depois de 208 *threads* a curva da versão **EWS AdjSync** também apresentar tendência de subida, ela sempre se mantém muito abaixo da curva *naïve*, alcançando um máximo de cerca de 20% do tempo de CPU.

A diminuição do *spin time* permite o ganho de desempenho na versão **EWS AdjSync** mesmo quando muitas *threads* são alocadas, e, consequentemente a não diminuição do *spin time* limita o desempenho da versão *naïve* quando muitas *threads* são utilizadas. Isso se deve ao fato da abordagem *naïve* possuir barreiras de sincronização e portanto ser muito mais sensível aos efeitos negativos do aumento do número de *threads*. O fato da curva de *speedup* da versão *naïve* começar a cair no mesmo ponto que a de *spin time* começa a subir corrobora essa tese. O gráfico da razão entre *spin time* e tempo de CPU também reforça essa tese ao mostrar que a fração do tempo total gasta com sincronização na abordagem *naïve* sobe no mesmo momento que o *speedup* cai, uma vez que parte considerável do processamento é gasto somente em barreiras(*spin time*).

Outro ponto de grande valor para a análise dos resultados se encontra no gráfico de tempo de *overhead*, que mostra que a *overthread* na versão **EWS** não teve impacto significativo no tempo gasto nessas rotinas de alocação e escalonamento de *threads*, uma vez que esse tempo representa uma fração muito baixa do tempo total, ao contrário da versão *naïve*.

5. Conclusões e Trabalhos Futuros

Após essas análises, é possível concluir que a estratégia de *overthread* teve um grande êxito ao obter um ganho de desempenho e que ele só foi possível devido à redução do tempo gasto na alocação e escalonamento das *threads*(tempo de *overhead*) através da estratégia de divisão explícita de trabalho (**EWS**), em uso conjunto da estratégia de sincronismo de *threads* adjacentes (**AdjSync**), para a redução drástica do *spin time*.

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 *overthread* 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 entradas maiores, de dimensão 3D e com outros método numéricos e em outras arquiteturas. Este trabalho foi desenvolvido em um nó computacional com 4 *sockets* **Cascade Lake**, pretendemos também dar continuidade a este estudo nos nós computacionais do supercomputador **SDumont** que possuem 2 *sockets* **Cascade Lake** por nó.

Referências

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.