Otimização do Método HOPMOC 1D com auxílio das ferramentas Intel Parallel Studio

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

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

{gcosta,fcabral,osthoff}@lncc.br

Abstract. *This paper presents a comparative study among different parallelization techniques used to performance improvement of the HOPMOC numerical method for advection-diffusion hyberbolic partial different equations resolution. The objective is to evaluate the gains of the two developed strategies by the original code version, with the aim of reduce the time spent in synchronization barriers, and then compare them among themselves. Moreover this paper brings a new study if compared to the others HOPMOC publications: the analysis concerning the relation of Spin Time and CPU Time to prove the developed strategies efficiency.*

Resumo. *Esse trabalho apresenta um estudo comparativo entre diferentes técnicas de paralelização utilizadas para aumentar o desempenho do método numérico HOPMOC para resolução de equações diferenciais parciais hiperbólicas de problemas de convecção-difusão. O objetivo é avaliar os ganhos de duas estratégias desenvolvidas à partir da versão original do código, com o ituito de diminuir os tempos gastos em barreiras de sincronização, e compará-las entre si. Além disso o trabalho traz um novo estudo em relação às outras publicações envolvendo o HOPMOC: a análise da relação entre Spin Time e CPU Time para comprovar a eficiência das estratégias desenvolvidas.*

1. Introdução

Equações diferenciais parciais são uma ferramenta indispensável para a modelagem de diversos problemas de alta relevância em nosso contexto de produção contemporâneo, e em diversas áreas científicas, como por exemplo física, química, engenharia, e até aplicações de mercado. Alguns episódios capazes de ilustrar essa relevância, são, por exemplo, a dispersão de um poluente em mar aberto ou outro meio como o ar, ou a previsão do estado futuro de algum fluxo geofísico relevante para áreas como a meteorologia ou a oceanografia.

Considerando a equação diferencial parcial de convecção-difusão da seguinte forma:

$$u_t + vu_x = du_{xx} \ , \tag{1}$$

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

$$u_t = du_{xx} \ . \tag{2}$$

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

Métodos numéricos dessa natureza contam com a possibilidade de otimização computacional através do paralelismo, permitindo a divisão da carga de trabalho entre diversas threads e unidades físicas. Nessa busca por um paralelismo ótimo, ganham destaque ferramentas que analisam o comportamento das instruções em tempo de execução e levantem informações relevantes acerca desse perfil de execução, permitindo identificar limitadores e alterações que possibilitem o ganho de desempenho. O Intel® Parallel Studio XE é um conjunto de ferramentas da Intel com essas características, que facilita a análise e desenvolvimento de código para a arquitetura Haswell/Broadwell, utlizada no desenvolvimento desse trabalho. À partir das informações levantadas pelas ferramentas do Parallel Studio com base em execuções da versão mais simples do HOPMOC, apelidada de Naive, identificaram-se sérias limitações no ganho de desempenho dessa versão, e fatores que poderiam estar relacionados com essas limitações. Partindo desses indicativos foram desenvolvidas duas estratégias com o objetivo de contornar esses limitadores e aumentar o desempenho do método: o HOPMOC EWS e o HOPMOC MPI.

Esse trabalho apresenta um estudo que relaciona os tempos gastos em barreiras de sincronização (spin time) e a soma total dos tempos de todas as threads (CPU time) como uma das métricas para verificar e explicar as limitações da versão original mais simples e os ganhos e a superioridade das versões desenvolvidas, um estudo que ainda não havia sido feito em outros trabalhos envolvendo o HOPMOC. O desenvolvimento de uma versão unidimensional que baseia majoritariamente seu paralelismo em processos MPI também é inédito entre os trabalhos publicados anteriormente sobre o HOPMOC, uma vez que o paralelismo baseado majoritariamente em threads e OpenMP estavam presentes em todas as estratégias anteriores.

Os testes apresentados nesse artigo foram feitos em um computador com dois processadores Intel® Xeon ® CPU E5 - 2698 v3 @2.30GHz, totalizando 32 núcleos, uma memória total de 115097784 kB e um SO CentOS Linux 7 (Core) . Todas as simulações foram realizadas com uma malha de $N = 10^5$ pontos, ou seja, $\Delta x = 10^{-5}$ e $T = 10^6$ iterações.

2. Trabalhos Relacionados

O método numérico HOPMOC possibilita uma grande redução de operações de troca de mensagens em um ambiente de sistema de memórias distribuídas e tem sido estudado para diversos ambientes de computação paralela. O trabalho [Cabral et al. 2017] apresentou um estudo sobre o desempenho do método em um ambiente computacional de memória compartilhada Intel MIC Xeon PHI utilizando o modelo de programação OpenMP. O trabalho [Cabral et al. 2018a] propôs um novo algoritmo do método utilizando implementações mais eficientes de diretivas OpenMP e apresentou um estudo

do desempenho para um ambiente de memória compartilhada Intel Xeon PHI. O trabalho [Cabral et al. 2018b] estudou o desempenho da metodologia apresentada no trabalho anterior para uma versão TVD-HOPMOC e para diversas arquiteturas de memória compartilhada Intel Xeon. Este trabalho compara o desempenho do método HOPMOC em um ambiente de memória compartilhada Xeon para uma versão em MPI em relação a versão OpenMP apresentada no trabalho [Cabral et al. 2018a] chamada de HOPMOC EWS.

Trabalhos com objetivos parecidos já foram desenvolvidos para outras aplicações, como [Diener et al. 2017] em um estudo para aperfeiçoar o acesso à memória em aplicações híbridas MPI/OpenMP e compará-lo com as versões padrão e puramente MPI, e [Bassi et al. 2016] que apresenta um modelo de paralelização híbrido MPI/OpenMP para o método Galerkin Descontínuo. Os dois trabalhos desenvolvem a otimização MPI/OpenMP também em arquitetura com memória compartilhada.

3. A implementação NAIVE baseada em OpenMP

A versão mais inicial do HOPMOC, apelidada de NAIVE, consiste de um núcleo composto por um loop while responsável por realizar as operações necessárias na malha a cada instante de tempo, a saber: o computo do processo de convecção realizado pelo Método das Características e o cálculo das frações de malha explícita e implícita, dividias sempre em dois semi passos dentro de cada iteração do loop principal.

O paralelismo nessa versão é baseado em OpenMP, e consiste no uso da diretiva **parallel for** para dividir a carga de trabalho de cada loop for entre as threads disponíveis, como mostra o pseudocódigo do algoritmo 1.

```
1  begin
2      #pragma omp parallel for;
3      for (i ← 1; i ≤ n − 1; i ← i + 1) do
           // computo do passo MOC
4      end

5      #pragma omp parallel for;
6      for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // computo do primeiro semipasso explícito
7      end

8      #pragma omp parallel for;
9      for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // computo do primeiro semipasso implícito
10     end

11     #pragma omp parallel for;
12     for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // computo do segundo semipasso explícito
13     end

14     #pragma omp parallel for;
15     for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // computo do segundo semipasso implícito
16     end
17 end
```

Algorithm 1: Pseudocódigo do loop principal da versão Naive, constituído de outros cinco loops internos, responsáveis pelo cálculo da convecção e da difusão a cada iteração. [Cabral et al. 2018b]

Os resultados coletados pelo Intel ® VTune Amplifier para os testes realizados com a versão Naive apresentados no lado esquerdo figura 1 mostram a evolução do tempo total gasto para executar o código (Elapsed Time), da soma do tempo de execução de todas as threads envolvidas (CPU time), e do tempo gasto em barreiras de sincronização (Spin time), onde as threads que já terminaram sua iteração do loop principal devem aguardar as demais para continuarem a execução. O gráfico do lado direito da figura 1 apresenta o speedup conforme se aumenta o numero de threads.

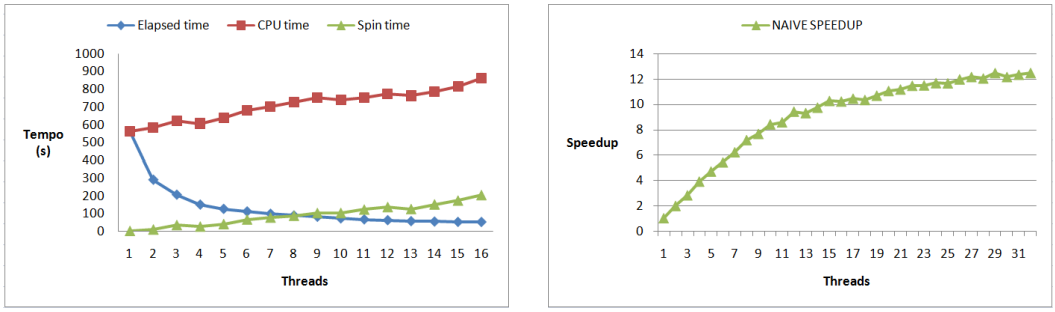

Figure 1. Elapsed time, CPU time e Spin time na versão Naive (esquerda) e Speedup na versão Naive(direita)

O gráfico da figura 1 comprova o speedup limitado da versão Naive, que segue uma tendência de estabilização por volta da décima quinta thread, subutilizando a capacidade total da máquina. Essa limitação se dá pelo aumento gradual do tempo gasto em barreiras de sincronização entre as threads criadas pelo OpenMP, uma vez que uma thread só pode seguir sua execução quando todas as outras, assim como ela, chegarem ao fim da iteração atual. Observa-se também na figura 1, uma relação direta entre o aumento gradual do spin time e o aumento gradual do tempo total de CPU (que em uma execução ideal com paralelismo totalmente eficiente se manteria constante), o que reafirma o impacto negativo e limitante do spin time em um código que basea seu paralelismo puramente nas diretivas do OpenMP.

Outra análise importante é a relativa ao histograma de tempo de execução por número de threads apresentado na figura 2. A figura consiste em 6 resultados coletados que mostram a quantidade de tempo que aquela execução passou utilizando um determinado número de threads. Cada execução conta com um número máximo de treads disponíveis, no caso da figura 2: 1, 2, 4, 8, 12 e 16. Nele é possível observar uma tendência de subutilização do número total de threads disponíveis, conforme esse número de threads disponíveis cresce. No último histograma da figura, que representa a execução com 16 threads, é possível ver que ainda sim foi gasto muito tempo executando o código com menos de 16 threads, o que leva a um uso médio da CPU sempre abaixo do máximo desejado (de pouco mais de 11 threads para quando havia 16 threads disponíveis, por exemplo). Essa subutilização de recursos é mais um dos impactos negativos dos tempos gastos em barreiras de sincronização (spin time), uma vez que as threads que terminam sua iteração mais rapidamente ficam ociosas até que as threads mais lentas as alcançem.

4. A Implementação EWS

Tendo em vista os fatores limitantes da abordagem Naive apresentados na seção anterior, foi desenvolvida uma nova implementação do HOPMOC a fim de contornar esses fatores limitantes e entregar um maior desempenho. Desse esforço foi criada a estratégia EWS-AdSynch (Explicit Work Sharing - Adjacent Synch).

A abordagem EWS consiste em uma divisão manual da carga de trabalho entre as threads disponíveis, junto com uma sincronização de threads realizada diretamente apenas com as threads adjacentes. Dessa forma, antes de iniciar a execução do loop principal a malha é dividida explicitamente de acordo com a quantidade de threads disponíveis

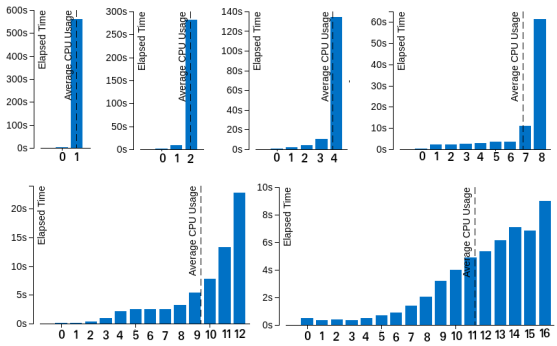

Figure 2. Histogramas do tempo de execução decorrido com cada numero de threads para a versão NAIVE

para aquela execução, de sorte que cada segmento de malha será designado a uma thread específica até o fim da execução. Além disso, a sincronização de uma thread específica não é mais realizada diretamente com todas as outras threads através de uma diretiva OpenMP como na versão Naive, mas somente entre as threads adjacentes através de uma espera ocupada ao final de cada loop for que constitui o loop principal. Para isso a versão EWS conta com um vetor booleano de status, onde cada entrada corresponde ao status de determinada thread: pronta para prosseguir ou não.

```
1  begin
2      tid ← omp_get_thread_num();
3      nt ← omp_get_num_threads();
4      size ← (n−2)/nt;
5      remainder ← (n − 2)%nt;
6      localStart ← tid · size + 1;
7      localEnd ← localStart · size − 1;
8      if (tid = nt − 1) then localEnd ← localEnd + remainder;
9      nid ← tid + 1;
10     boolean lock[242];
11     lock[nid] ← false;
12     #pragma omp master;
13     {
14       lock[0] ← false;
15       lock[nt+1] ← false;
16     }
17     #pragma omp flush (lock) // atualiza o vetor lock
18     [...]
       // mecanismo de lock:  informa as threads adjacentes que
       // essa thread esta realizando uma tarefa na memória compartilhada
19     lock[nid] ← true;
20     for (i ← localStart; i ≤ localEnd; i ← i + 2) do
         |   // realiza algum trabalho
21     end
       // libera a memória compartilhada para as threads adjacentes
22     lock[nid] ← false;
       // verifica se a memória compartilhada esta travada
       // e aguarda até que ela seja liberada
23     while (lock[nid + 1] ∨ lock[nid − 1]) do ;
24     [...]
25 end
```

Algorithm 2: Fragmento da versão EWS: mecanismo de divisão explícita de threads e sincronização entre adjacentes. [Cabral et al. 2018b]

Testes realizados na mesma máquina e novamente com uma malha de 10^5 pontos apresentaram, com dados coletados pelo Intel ® VTune Amplifier, os resultados observados na figura 3

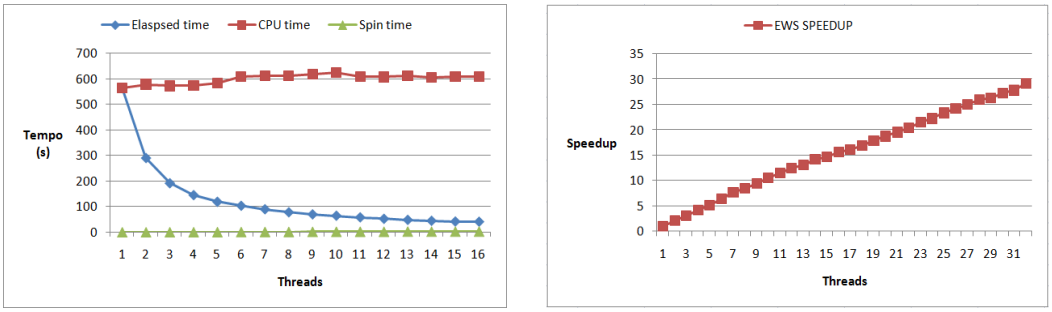

Figure 3. Elapsed time, CPU time e Spin time na versão EWS (esquerda) e Speedup na versão EWS(direita)

O segundo gráfico da figura 3 apresenta um speedup quase linear para o HOPMOC EWS, consideravelmente superior ao da versão NAIVE que apresenta uma tendência de estabilização alcançando o speedup de 12.45 para 32 threads contra 29.15 do EWS para as mesmas 32 threads. Na figura 3 torna-se notável a quase não existência de spin time apesar do aumento do número de threads(salta de um valor de 0.21 segundos com 2 threads para apenas 3.16 segundos com 16 threads), o que se traduz, também, em uma curva mais contínua de CPU time. Essa relação entre um spin time desprezível e um tempo quase constante de CPU time leva a uma observação muito relevante: o mecanismo de sincronização adjacente através de espera ocupada não gera acréscimo de tempo na execução, ao contrário da barreira de sincronização do OpenMP.

O conjunto de histogramas da figura 4 mostra o tempo decorrido utilizando um determinado número de threads dentre o máximo disponível no momento, em três execuções distintas: com um máximo de 4 threads, um máximo de 8 threads e um máximo de 16 threads. Em todas as execuções observa-se um aproveitamento ótimo dos recursos disponíveis, uma vez que quase toda a totalidade do tempo é gasta utilizando-se o número máximo de threads disponíveis.

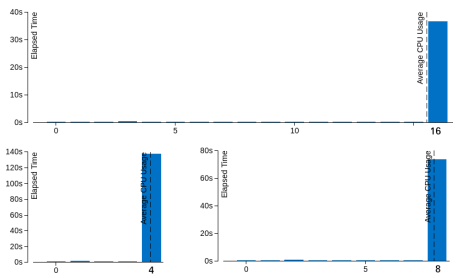

Figure 4. Histogramas para 4, 8 e 16 threads na versão EWS

Os resultados apontam para o sucesso da estratégia EWS, que através do mecanismo de divisão explícita da malha e sincronização por espera ocupada entre as threads adjacentes elimina o spin time e o tempo de alocação de threads, e permite um melhor aproveitamento das threads disponíveis, o que resulta em ganho de speedup mais poderoso.

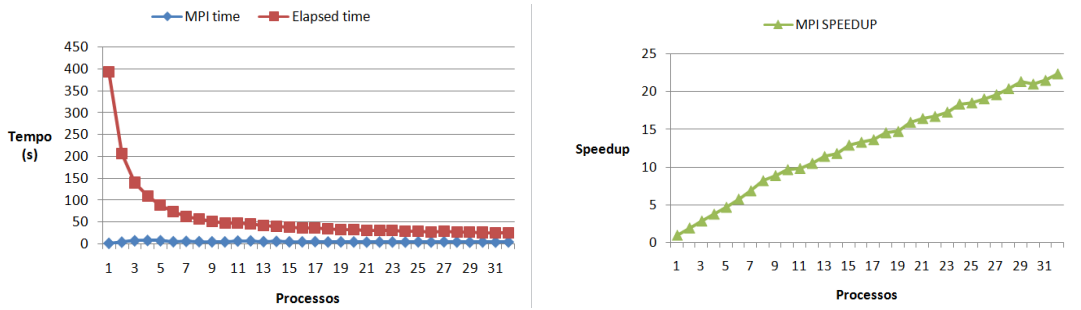

Figure 5. Elapsed time e MPI time na versão MPI (esquerda) e Speedup na versão MPI(direita)

5. A implementação MPI

Outra estratégia desenvolvida foi uma versão do HOPMOC com o paralelismo baseado em multi processos, através do padrão MPI. Essa abordagem guarda fortes semelhanças com o EWS, diferindo, em linhas gerais, apenas no fato de fazer uso majoritário de processos ao invés de threads para garantir o paralelismo, e com isso utilizar a comunicação entre processos adjacentes para garantir a sincronia: a espera ocupada ao fim de cada loop for é substituída pela troca de mensagens entre processos, através de chamadas não bloqueantes para envio e recebimento. Nessa versão a malha pricipal também é dividida em submalhas menores, cada uma designada explicitamente a um dos processos disponíveis de forma permanente até o fim da execução, antes de se iniciar o loop principal.

A figura 5 mostra os resultados dos testes realizados na mesma máquina das seções apresentadas previamente.

O lado direito figura 5 revela um speedup considerável, atingindo 22.33 para um máximo de 32 processos. Já o lado esquerdo da figura 5 exibe o tempo total de execução (Elapsed time) junto com o MPI Time, que é o tempo gasto em rotinas estritamente pertencentes ao padrão MPI, como o envio e o recebimento de mensagens entre os processos e a espera realizada por cada processo afim de se manter sincronizado com seus vizinhos. Assim, o MPI time se apresenta como um equivalente do Spin time das versões baseadas em OpenMP como a Naive e a EWS. Nota-se também a constância dos baixos valores de MPI time, que se estabilizam entre 4 e 5 segundos, assim como os baixos valores de Spin time da versão EWS, explicados pelo fato de que a sincronização nessas versões passa diretamente somente pela sincronização entre adjacentes, e não com todas as demais threads/processos.

6. Conclusão e trabalhos futuros

Analisando os resultados apresentados nas três seções anteriores, torna-se notável o impacto negativo que o tempo gasto em barreiras de sincronização tem sobre a utilização dos recursos disponíveis para se alcançar o paralelismo, gerando uma subutilização do total de threads disponíveis. Essa subutilização se traduz em um maior tempo de CPU e em uma perda de ganho de speedup. Assim, notadamente, a versão Naive, por ter os maiores spin times apresenta um desempenho inferior às outras duas estratégias. Por outro lado, o melhor desempenho foi alcançado na estratégia EWS, que reduz em grande medida o spin time através da sincronização adjacente, e com isso consegue melhores tempos de CPU e

maior ganho de speedup, já que tira maior proveito das threads disponíveis. Esse speedup acompanhado de um tempo constante de CPU (que não sobe com o aumento do número de threads, como na versão Naive) comprova a eficiência do mecanismo de sincronização adjacente por espera ocupada que não acrescenta um tempo significativo à execução. A abordagem MPI apresenta desempenho consideravelmente superior à versão Naive, pelos mesmos motivos que a EWS os apresenta. Entretanto essa última abordagem fica levemente atrás da estratégia EWS em termos de desempenho. Essa pequena diferença se dá pelo mecanismo de sincronização: a troca de mensagem entre os processos se prova mais lenta que a espera ocupada utilizada para sincronização de threads.

Trabalhos futuros incluem a extensão das estratégias aprsentadas nesse artigo para modelos matriciais bidimensionais, e a aplicação do método para a resolução de outras equações diferenciais parciais como a equação do calor e a equação de Laplace.

7. Referências

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., Osthoff, C., Costa, G. P., Brandão, D., Kischinhevsky, M., and de Oliveira, S. L. G. (2017). Tuning up 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., de Oliveira, S. L. G., 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., de Oliveira, S. L. G., 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.

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.