Implementações Híbridas MPI/OpenMP/OpenACC/CUDA do Método HOPMOC na Resolução da Equação de Convecção-Difusão

Frederico Luís Cabral[1], Carla Osthoff[1], Mauricio Kischinhevsky[2], Diego N. Brandão[3], Leonardo Jasmim[4]

[1]Laboratório Nacional de Computação Científica, LNCC
Av. Getulio Vargas, 333, Petropolis-RJ

[2]Universidade Federal Fluminense, UFF
Instituto de Computação
Niterói - RJ

[3]CEFET-RJ, Colegiado de Informática
Nova Iguaçú - RJ

[4]Centro Universitário Serra dos Órgãos, UNIFESO
Teresópolis - RJ

{fcabral,osthoff}@lncc.br, kisch@ic.uff.br, dbrandao@ic.uff.br

leojasmim@gmail.com

Abstract. *A utilização da computação paralela na resolução de certos problemas descritos por equações diferenciais parciais permite um ganho significativo no tempo de computação. Este trabalho apresenta algumas implementações paralelas do método HOPMOC em ambientes de máquinas multicore e manycore. O método HOPMOC utiliza conceitos do método das características modificado associado com método Hopscotch, o que lhe fornece características ideais para abordagens em computação paralela em ambientes tanto de memória distribuída como compartilhada. O MPI é utilizado para comunicação no ambiente distribuído, enquanto OpenMP permite o paralelismo no ambiente de memória compartilhada de cada nó do cluster. OpenACC e CUDA, permitem o paralelismo no ambiente manycore disponível em placas aceleradoras gráficas. Resultados preliminares demonstram ganhos significativos de eficiência das implementações híbridas apresentadas quando comparado com uma versão sequencial do HOPMOC. As implementações que usam placas gráficas (manycore), apresentam menor tempo de execução quando comparado com OpenMP (multicore), mas por outro lado, a relação speedup por quantidade de cores é melhor no ambiente multicore, sugerindo um melhor aproveitamento das unidades de execução (cores).*

1. Introdução

Máquinas *multicore* e *manycore* são o cerne dos sistemas computacionais dos dias de hoje. Tais arquiteturas fornecem um grande poder computacional para resolução de problemas de grande porte das áreas de engenharia, física, e etc. Assim, o desenvolvimento de métodos que sejam robustos e eficientes para tais arquiteturas torna-se necessário.

O método HOPMOC foi proposto como um método para resolução de problemas de convecção-difusão com dominância convectiva para ser executado em ambientes compostos por máquinas paralelas [1]. Ele permite uma *multithreading* decomposição espacial do domínio do problema, onde cada subproblema pode ser resolvido independentemente. Além disso ele minimiza a troca de informações entre regiões vizinhas, reduzindo assim a troca de mensagens entre os nós do ambiente distribuído. Outra característica é que seu custo por instante de tempo é conhecido *a priori,* além de usar uma estratégia baseada na busca por informação ao longo da linha característica para cada instante de tempo.

Técnicas de decomposição de operadores, como as empregadas pelo HOPMOC, foram estudadas durante as últimas décadas com o propósito de reduzir o custo computacional dos métodos para solucionar problemas multidimensionais evolutivos modelados por equações diferenciais parciais [4, 2, 3]. Hoje com o avanço das tecnologias de paralelismo em ambientes de memória compartilhada e distribuída, o foco da pesquisa tem sido no desenvolvimento de estratégias que explorem ao máximo as capacidades computacionais desses ambientes, com uma atenção especial para o consumo de memória e o custo de troca de mensagens.

No contexto de computação distribuída, o padrão Message Passing Interface (MPI) tem sido utilizado por muitos algoritmos para alcançar a escalabilidade em um grande número de processadores. Todavia, ela não consegue utilizar o paralelismo do ambiente de múltiplos núcleos do processador de forma eficiente [5]. Alternativamente, OpenMP é um padrão de programação paralela que permite o paralelismo mais interno no contexto dos núcleos do processador. A associação dos modelos de programação paralelos implementados pelo OpenMP e pelo MPI permite a construção de um programa híbrido que é capaz de atingir dois níveis de paralelismo.Essa abordagem reduz os gastos com a comunicação gerado pelo MPI ao custo de introduzir gastos no OpenMP para criação de cada thread. Contudo, as abordagens híbridas dadas por meio do modelo MPI/OpenMP apresentaram um ganho significativo de eficiência conforme pode ser observado em [7, 6].

A linguagem CUDA, *Compute Unified Device Architecture*, desenvolvida pela NVIDIA para a arquitetura *manycore* de suas placas gráficas, permite que os recursos matemáticos de cada uma das suas centenas a milhares de unidades de execução, sejam usados por programadores para aplicações diversas e não apenas gráficas. Tem-se assim, um ambiente de execução onde um grande número de *threads* simultâneas trabalham em paralelo para a resolução do problema. O OpenACC é uma *Application Program Interface* (API) que fornece diretivas de compilação, chamadas de função e variáveis de ambiente, que permitem a programação para placas gráficas aceleradoras sem que seja necessário conhecer grande parte dos detalhes de sua arquitetura complexa. Embora o OpenACC seja de fácil utilização, perde-se o controle do *fine tunning* que pode ser feito para um desempenho ótimo, como em CUDA.

Este trabalho apresenta três abordagem híbridas, que são adequadas ao processamento em *cluster*s computacionais compostos por vários nós de processamento que se comunicam através de *links* de rede. Para a comunicação entre os nós o MPI é usado, de modo que em cada nó um processo seja instanciado e posto em execução. Cada processo MPI ainda é dividido em *threads* que se comunicam através de memória compartilhada.

Para esta execução *multithreading*, três abordagens foram aplicadas: (1) OpenMP; (2) OpenACC ; (3) CUDA. Tem-se assim, as abordagens híbridas MPI/OpenMP, MPI/OpenACC e MPI/CUDA que serão comparadas para o HOPMOC aplicado na resolução da equação de convecção-difusão.

Uma versão sequencial deste método foi apresentada para o mesmo problema em [9]. Já sua análise teórica com as condições de estabilidade foi apresentada em [10]. Uma primeira abordagem em paralelo utilizando o MPI foi apresentada em [11]; todavia não foi utilizado um ambiente de mútiplos núcleos. Além disso a técnica de decomposição de domínito utilizada aqui reduz o custo de comunicação quando comparada com do trabalho citado.

Este trabalho é organizado da seguinte maneira: a seção 2 apresenta a discretização da equação de convecção-difusão por meio do método HOPMOC e discute algumas características do método; a seção 3, descreve a implementação paralela por meio dos modelos híbridos; o ambiente computacional e as simulações realizadas são apresentados na seção 4; por fim, as conclusões e desdobramentos do trabalho são apresentados na seção 5.

2. Descrição do Método HOPMOC

Considere a equação de convecção-difusão unidimensional:

$$u_t + v u_x = d u_{xx} \tag{1}$$

com condições iniciais e de contorno adequadas, onde a velocidade v é constante e positiva, d é uma constante positiva de difusividade, $0 \leq x \leq 1$ e $0 \leq t \leq T$.

Considere um esquema convencional para discretização em diferenças finitas deste problema, onde $\Delta t = t_{n+2} - t_n$, $\delta t = \frac{\Delta t}{2} = t_{n+1} - t_n$, $\overline{u}_i^{n+1} = u(\overline{x}_i^{n+1})$, sendo este o valor da variável no instante de tempo anterior do semi-passo obtido no pé da linha característica, essa linha originada por x_i^{n+2} e $\Delta x = x_{i+1} - x_i = \frac{1}{\kappa+1}$, onde κ é ímpar. A mesma linha característica permite obter $\overline{\overline{u}}_i^n$ no semi-passo de tempo anterior.

Seja u_i^{n+2}, onde n é ímpar, uma aproximação numérica para u em (x_i, t_{n+2}), e usando um operador diferencial L, $L_h\ u_i^n = d\frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2}$, os semi-passos consecutivos do HOPMOC podem ser descritos por:

$$\begin{aligned}
\overline{u}_i^{n+1} &= \overline{\overline{u}}_i^n + \delta t \left[\theta_i^n L_h \overline{\overline{u}}_i^n + \theta_i^{n+1} L_h \overline{u}_i^{n+1}\right], \\
u_i^{n+2} &= \overline{u}_i^{n+1} + \delta t \left[\theta_i^n L_h \overline{u}_i^{n+1} + \theta_i^{n+1} L_h u_i^{n+2}\right], \\
&\text{para} \quad \theta_i^n = \begin{cases} 1, & \text{se} \quad n+i \quad \text{é} \quad \text{ímpar}, \\ 0, & \text{se} \quad n+i \quad \text{é} \quad \text{par}, \end{cases}
\end{aligned} \tag{2}$$

e o valor $\overline{\overline{u}}_i^n$ é obtido por meio de um processo de interpolação. Os valores $\overline{x}_i^{n+1}$ e $\overline{\overline{x}}_i^{n+1}$ são obtidos por $\overline{x}_i^{n+1} = x_i - v\ \delta t$ e $\overline{\overline{x}}_i^n = x_i - 2v\ \delta t$.

A análise de convergência do HOPMOC foi apresentada em [10], nele são descritas as condições de consistência e estabilidade no caso da discretização do problema da equação de convecção-difusão. O teorema de Lax garante que se o problema de valor inicial é bem posto no sentido de Hadamard, as condições de estabilidade e consistência

implicam na convergência do método numérico [12]. Assim, o método HOPMOC é incondicionalmente estável para Eq. 1. A Figura 1 ilustra esquematicamente o método HOPMOC para um problema unidimensional.

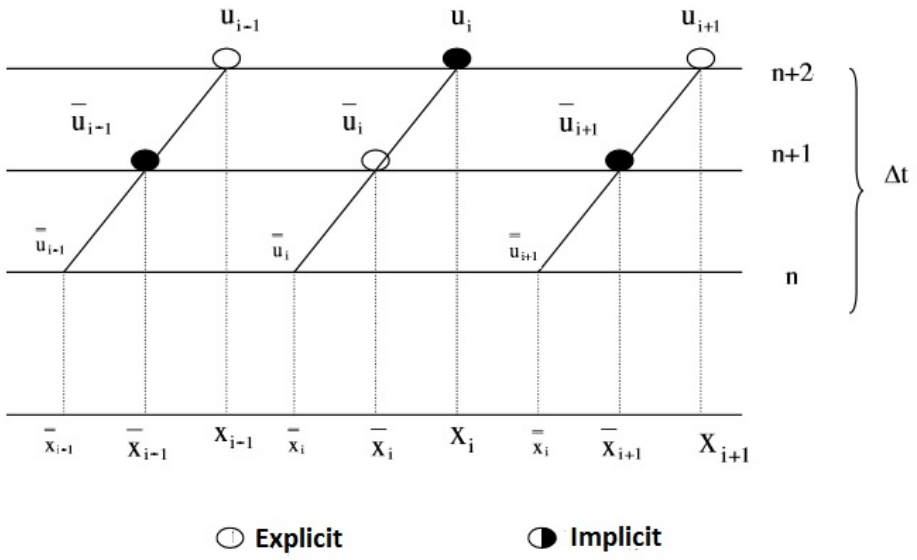

Figura 1. Método HOPMOC.

No caso bidimensional, os dois passos do método grupo HOPMOC com quatro pontos são descritos pela Eq. 3.

$$
\begin{aligned}
\overline{u}_{i,j}^{n+1} &= \overline{\overline{u}}_{i,j}^{n} + \delta t \left[\theta_{i,j}^{n} L_h \overline{\overline{u}}_{i,j}^{n} + \theta_{i,j}^{n+1} L_h \overline{u}_{i,j}^{n+1}\right], \\
u_{i,j}^{n+2} &= \overline{u}_{i,j}^{n+1} + \delta t \left[\theta_{i,j}^{n} L_h \overline{u}_{i,j}^{n+1} + \theta_{i,j}^{n+1} L_h u_{i,j}^{n+2}\right], \\
\text{para} \quad \theta_{i,j}^{n} &= \begin{cases} 1, & \text{se} \quad [i+1]/2 + [j+1]/2 + n \quad \text{é} \quad \text{ímpar}, \\ 0, & \text{se} \quad [i+1]/2 + [j+1]/2 + n \quad \text{é} \quad \text{par} \end{cases}
\end{aligned} \tag{3}
$$

O HOPMOC não requer que um sistema linear seja resolvido; assim, sua paralelização é simples de ser realizada, pois o desacoplamento espacial permite que as variáveis sejam subdivididas em processadores diferentes, sem que dependam de variáveis armazenadas em outros processadores. Uma outra vantagem a ser observada refere-se ao custo computacional do HOPMOC que é $O(N)$ por instante de tempo.

3. Computação Paralela

A biblioteca mais utilizada para comunicação entre computadores em um ambiente distribuído é a *Message Passing Interface* (MPI). Todavia, esta biblioteca não consegue obter o máximo de performance em ambientes de memória compartilhada, caso do paralelismo em processadores de múltiplos núcleos. Nestes ambientes são utilizados os modelos *multicore* ou *manycore*, os quais permitem a paralelização entre os núcleos de um processador

ou entre as diversas unidades funcionais de placas aceleradoras. Com as políticas de compartilhamento de dados, o multi*thread*ing elimina o *overhead* de comunicação. Um laço do programa pode ser facilmente paralelizado por meio de uma chamada de uma subrotina da biblioteca de *threads* do OpenMP ou OpenACC, simplesmente inserindo algumas diretivas de compilação de alto nível. A linguagem CUDA por sua vez é de baixo nível pois o comportamento das *threads* é determinado explicitamente pelo programador dentro das chamadas funções de *kernel*.

3.1. Versões Paralelas do Método HOPMOC

O método HOPMOC é ideal para o paralelismo em ambientes de memória tanto distribuída como compartilhada, pois a comunicação entre os subdomínios que são geometricamente contíguos é realizada apenas três vezes a cada passo no tempo. Essa característica permite que o problema seja resolvido de forma independente em cada subdomínio. Em [11] é apresentada uma versão em paralelo do método HOPMOC e do grupo HOPMOC. Ambas as versões utilizavam o MPI em um ambiente de memória distribuída.

O procedimento empregado pelo HOPMOC em cada instante de tempo pode ser dividido em dois estágios, que podem ser avaliados sequencialmente, além de serem independentes de referências às variáveis correspondentes a pontos distantes; ou seja, não há acoplamento de longo alcance. Assim, os subdomínios são desacoplados, o custo computacional escalável é esperado, isto é, o *speedup* cresce quase linearmente com o número de processadores, uma característica desejável para programas paralelos.

Os passos da abordagem paralela do HOPMOC, com balanço de carga computacional, consistem em: primeiramente, o domínio do problema deve ser dividido em subdomínios de mesmo tamanho e que tenham interface de contato, com uma faixa de replicação, para que as informações de interfaces superpostas sejam replicadas entre subdomínios vizinhos a cada passo de tempo. A união de todos os subdomínios constitui uma decomposição do domínio, com redundância monocamada nas interfaces entre os subdomínios. Para que se tenha um resultado eficiente, deve-se buscar que o número de subdomínios seja igual ao de nós processadores do *cluster*. A Figura 2 representa a subdivisão descrita para o caso de quatro subdomínios sobrepostos.

Este esquema de decomposição do domínio reduz a quantidade de mensagens trocadas, uma vez que cada processo MPI reside em um nó do *cluster*. Desta forma as várias *threads* dentro de um mesmo nó se comunicam através de memória compartilhada. Na abordagem utilizada a quantidade máxima de troca de mensagens é de duas mensagens para cada subdomínio. O esqueleto de código mostrado a seguir ilustra a implementação híbrida HOPMOC que é a mesma independente de o esquema de multi*thread*ing ser OpenMP, OpenACC ou CUDA.

Para as estratégias MPI/OpenMP e MPI/OpenACC, as *threads* são criadas e destruídas de acordo com o modelo *fork-join*, mostrado na figura (3). O trecho de código abaixo mostra uma estrutura de repetição típica com ambas as diretivas de compilação OpenMP e OpenACC. Como se pode depreender, o código fonte é o mesmo para ambas as versões, de modo que apenas no momento da compilação é feita a escolha.

```
1 #pragma acc kernels loop
2 #pragma omp parallel for
3 for (int j = 1 ; j <= N-2 ; j++) {
```

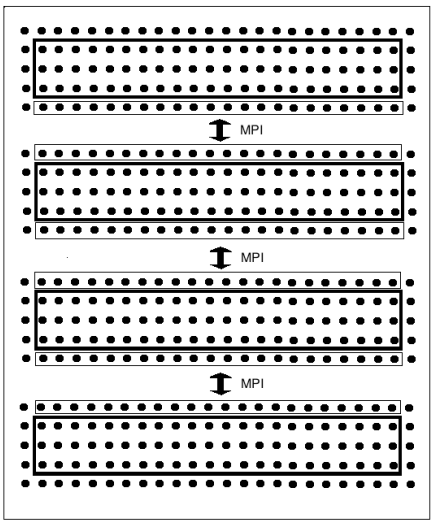

Figura 2. Divisão do domínio em quatro subdominíos sobrepostos.

Algorithm 1 Esqueleto principal de código

```
1: while (tempo < tempoFinal) do
2:     Calcula a informação no pé da linha característica
3:     Envia e Recebe via MPI a informação da vizinhança
4:     Calcula o operador explícito
5:     Envia e Recebe via MPI a informação da vizinhança
6:     Calcula o operador implícito
7:     Envia e Recebe via MPI a informação da vizinhança
```

```
4    #pragma acc loop
5    for (int i = inicioLocal ; i <= finalLocal ; i++) {
6      if ( (i+j+k) % 2 == 0 ) {
7      }
8    }
9  }
```

A implementação do método HOPMOC para CUDA buscou seguir a mesma estrutura paralela das soluções MPI/OpenMPI e MPI/OpenACC, ficando a principal diferença na criação dos *kernel*s para execução das funções específicas do método.

Primeiramente é realizada a preparação do ambiente com a criação e alocação das matrizes e variáveis necessárias na memória global da GPU. A próxima etapa é definir o dimensionamento das *threads* nos blocos e no *grid*. Esta etapa é importante pois o dimensionamento está intimamente ligado a uma melhor utilização dos *cores (occupancy)* que implica diretamente no desempenho da aplicação.

Os blocos e *grids* foram definidos de forma bidimensional a fim de melhor se ajustar às características do problema. Os blocos foram dimensionados com 16x16x1 *threads*/bloco, totalizando 256 *threads*/bloco, o que garante o melhor aproveitamento dos

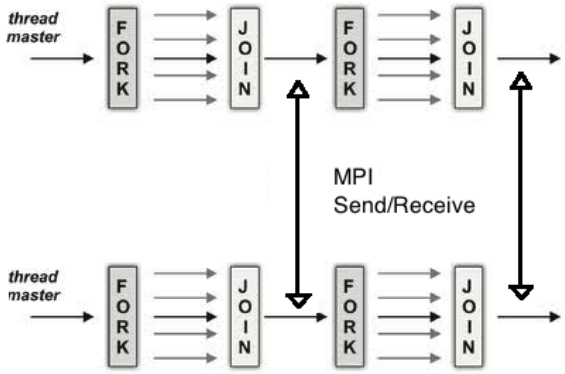

Figura 3. Ambiente Híbrido MPI/OpenMP.

registradores e da memória compartilhada. [8]. Já o número de blocos por *grid* é calculado dinamicamente, de acordo com o tamanho da matriz de entrada do problema:

```
1 int GRID_WIDTH = (int) ceil((double)(NP+1)/BLOCK_WIDTH)
```

A estrutura do laço do tempo foi mantida, entretanto os laços internos que realizam os cálculos do pé da característica, atualização da matriz e semipassos explícitos e implícitos foram transformados em *kernels*, permitindo que cada *thread* lançada opere em uma única posição específica das matrizes, de forma muito semelhante ao exemplo da figura 5.

Seja M um vetor multidimensional (matriz) alocado estaticamente, em geral, podemos acessar o valor de M na linha i coluna j utilizando a abstração M[i][j]. Entretanto, a extensão CUDA C trabalha com vetores alocados dinamicamente, sendo necessário que o programador, explicitamente, realize o trabalho do compilador de transladar a matriz para uma matriz linearizada. Utilizamos uma abordagem *row-major layout*, onde os itens são acessados linha a linha, através de um identificador único (tid) [8].

Ao executar estes *kernel*s, cada *thread* faz um cópia dos seus pontos vizinhos da memória global da GPU para sua memória privada, devido a velocidade de acesso desta ser amplamente maior que a anterior. Por exemplo, para realizar o cálculo do semipasso explícito, as *threads*, de modo intercalado, copiam os dados dos pontos vizinhos, em um formato de *stencil*, para sua memória privada. Ao final de cada *kernel*, a *thread* armazena o seu resultado de volta na matriz global. De forma análoga são calculados os pontos restantes pelo *kernel* do semipasso implícito, conforme a descrição do método citada anteriormente.

```
1 while (tempo <= tempoFinal){
2   cudaMemcpy (dev_k, &k, sizeof(long int),
       cudaMemcpyHostToDevice);
3   tempo = (double) (k*deltaT)/2;
4   calculaPeCaracteristica <<<gridDim, blockDim>>>
5   MPI_SEND_REICV
6   semipassoExplicito <<</gridDim, blockDim>>>
7   MPI_SEND_REICV
8   semipassoImplicito <<<gridDim, blockDim>>>
```

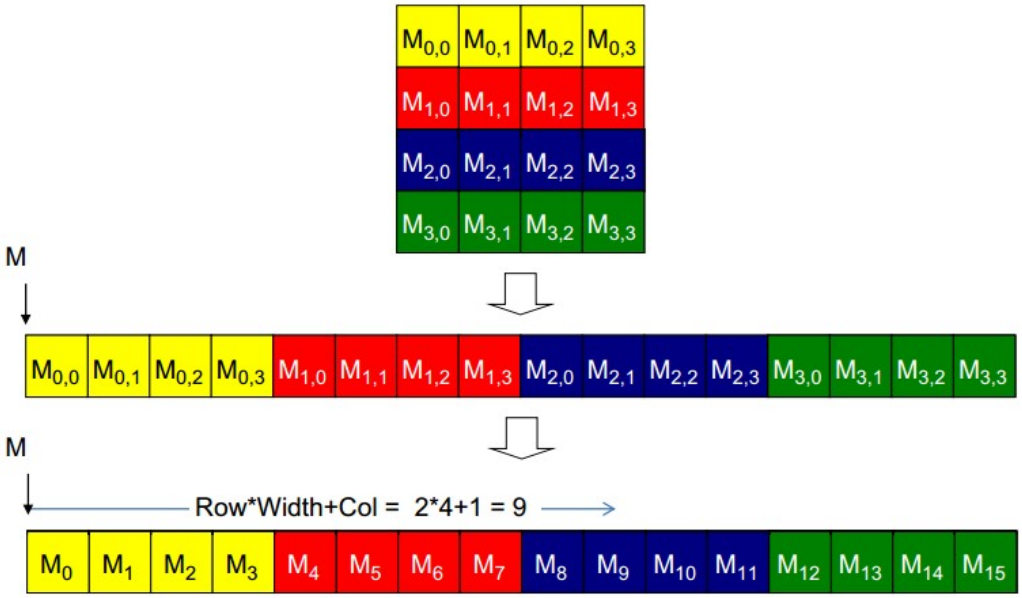

Figura 4. CUDA

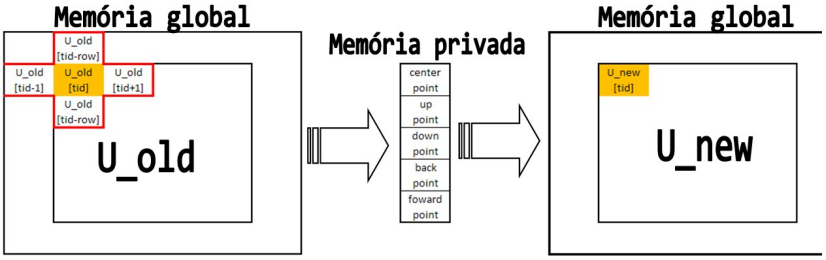

Figura 5. Malha de diferenças finitas em CUDA

```
9     MPI_SEND_REICV
10    k++;
11  }
```

Nesta implementação não foi necessária a utilização de barreiras de sincronização, pois as próprias chamadas de *kernel* já são barreiras implícitas, que foram suficientes para abordagem adotada. Para trabalhos futuros podem ser explorados aspectos da utilização da memória compartilhada entre as *threads* (*shared memory*) que é uma memória intermediária, de acesso mais rápido que a memória global, bem como a utilização de paralelismo dinâmico.

4. Simulação Numérica e Análise de Desempenho

Neste trabalho os experimentos computacionais foram realizados simulando a Eq.4 bidimensional.

$$u_t + v\{u_x + u_y\} = d\{u_{xx} + u_{yy}\} \tag{4}$$

com condição inicial

$$u(x, y, 0) = e^{\{-100*((x-xo)^2+(y-yo)^2)\}} \quad (5)$$

Os experimentos mostrados aqui foram executados com $D = 10^{-3}, xo = yo = 0.5, vx = vy = 0.1, \Delta t = 10^{-5}$ e condição de contorno de Dirichlet $u(X, t) = 0$ para X em toda fronteira e $t > 0$.

Esta equação tem papel fundamental na descrição dos modelos dos mais variados fenômenos físicos. Algumas aplicações podem ser encontradas na modelagem de transporte de poluentes em rios, nos modelos que descrevem o movimento de aerossóis na atmosfera, etc.

Ambiente Computacional e Experimentos Numéricos

Os resultados mostrados a seguir foram obtidos em um *cluster* de máquinas *multi-core* com sistema operacional Linux com um canal de comunicação *infiniband*. O *cluster* era composto por 4 nós, sendo cada nó formado por dois processadores Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz (16 *cores* por nó) com memória de 64GB DDR3 DIMMs e uma placa aceleradora NVIDIA k20.

A figura 6 apresenta a simulação do método HOPMOC para a Eq.4. As figuras demonstram o método representado o fenômeno representado pela Eq.4.

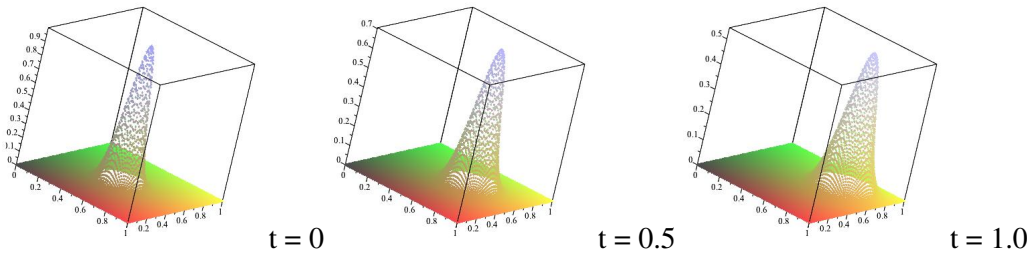

Figura 6. Simulação da equação de convecção-difusão com o método HOPMOC para diferentes instantes de tempo.

As Tabelas de 1 a 4 apresentam os resultados obtidos na simulação de um domínio contendo 1000x1000 pontos. Cada tarefa de trabalho MPI corresponde a um processo, dentro do qual existe um bloco de trabalho OpenMP, ou seja, várias threads, sendo que cada uma é executada por exatamente um core (núcleo). Assim, no início da execução são criados vários processos MPI, um para cada nó do cluster, conforme a figura 2. Estes processos duram por toda a simulação, mas para cada semi-passo do HOPMOC as threads OpenMP são criadas e destruídas, de acordo com o modelo fork-join e apenas a thread master executa os envios e recebimentos do MPI, entre os nós do cluster.

A tabela 1 apresenta a média do tempo de 3 execuções para cada simulação da execução do HOPMOC com multithreading OpenMP em função do número de cores, de 1 a 16 cores, em apenas um nó computacional multicore e a sua respectiva aceleração, apresentada na coluna speed-up[1].

A tabela 2 apresenta a média do tempo de 3 execuções para cada simulação em

função do número de cores, em um ambiente de quatro nós multicores e as suas respectivas acelerações em relação a um nó, apresentado na coluna speed-up[2]. A tabela 2 mostra o resultado da execução do código híbrido MPI/OpenMP no ambiente multicore. Percebe-se facilmente que enquantose aumenta o número de threads dentro de um único nó computacional, que o ganho de desempenho é praticamente linear.

threads	**tempo (em segundos)**	*speedup*[1]
1	10.574,7	1,0
2	5.407,88	1,95
4	2.773,87	3,8
8	1.522,32	6,9
16	674,98	15,66

Tabela 1. Resultados OpenMP

Na tabela 2 observa-se que há ganho de desempenho, porém com ganho sublinear devido ao gasto causado pelas trocas de mensagem via MPI. Isto fica evidente ao se analisar o speedup2 onde o ganho fica muito abaixo do número de nós usados do cluster. Por exemplo, o teste com quatro nós apresenta um ganho de apenas 2,6 vezes. O speedup[2] mostra o ganho com 2, 3, e 4 nós em relação á execução com apenas um nó (16 threads).

threads	**tempo (em segundos)**	*speedup*[1]	*speedup*[2]
16	674,98	15,66	1,0
32	534,71	19,77	1,2
48	349,84	30,22	1,9
64	259,24	40,79	2,6

Tabela 2. Resultados MPI/OpenMP

A tabela 3 apresenta a média do tempo de 3 execuções para cada simulação da versão hibrida MPI/OpenACC em função do número de cores, em um ambiente de quatro nós, cada um com uma placa GPU Tesla K20. A tabela apresenta as suas respectivas acelerações em relação a um nó executando com uma GPU, apresentado na coluna speed-up[3].

A tabela 4 apresenta a média do tempo de 3 execuções para cada simulação da versão hibridas MPI/Cuda em função do número de cores, em um ambiente de quatro nós, cada um com um GPU Tesla K20, e as suas respectivas acelerações em relação a um nó executando com uma única GPU, apresentado na coluna speed-up[4].

O ganho abaixo do linear também é observado nas versões hibridas MPI/OpenACC e MPI/CUDA. Embora o tempo de execução seja menor para um nó, o ganho com quatro nós é de 2,7 e 2,9, respectivamente (speedup[3] e speedup[4]), similarmente ao speedup[2].

Vale a pena ressaltar que a versão MPI/CUDA é ligeiramente mais eficiente do que a versão MPI/OpenACC, mesmo com um código não otimizado, ou seja, o mais trivial e ingênuo possível. Isto ocorre pelo fato de o OpenACC ser uma API de nível de abstração mais alto, de modo que os detalhes de criação das funções de kernel ficam escondidos do

nós (GPUs)	**tempo (em segundos)**	*Speedup*[1]	*Speedup*[3]
1	239,87	44,08	1,0
2	133,26	79,35	1,8
3	95,95	110,21	2,5
4	87,79	120,45	2,7

Tabela 3. Resultados MPI/OpenACC

programador, o que não acontece com a linguagem CUDA, pois tais funções são programadas explicitamente. Sendo assim, o potencial para otimização de código é maior em CUDA do que em OpenACC, embora este seja mais simples de se programar [8].

Outra observação importante é em relação ao speedup1 por quantidade de cores. Na tabela 2, ambiente multicore, pode-se observar que com 64 threads o ganho é próximo a 40 vezes, dando uma relação de aproximadamente 0,63. Nas tabelas 3 e 4, relativas ao ambiente manycore (GPU), os resultados apresentado foram obtidos nas placas aceleradoras NVIDIA k20, que possuem 2496 cuda cores. Somando todos os cores dos 4 nós computacionais do cluster com manycores, tem-se um total de 9984 cores (CUDA CORES), para um speedup de 120,45 na versão híbrida MPI/OpenACC e de 150,31 na versão híbrida MPI/CUDA, dando uma relação de 0,012 e 0,015 respectivamente. Esta baixa relação sugere que o desempenho pode ser melhorado com técnicas de otimização em CUDA e OpenACC.

nós (GPUs)	**tempo (em segundos)**	*Speedup*[1]	*Speedup*[4]
1	204,04	51,8	1,0
2	107,39	98,47	1,9
3	75,57	139,93	2,7
4	70,35	150,31	2,9

Tabela 4. Resultados MPI/CUDA

5. Conclusões e Trabalhos Futuros

Este artigo apresentou uma resolução para o problema de convecção-difusão utilizando implementações híbridas MPI/OpenMP, MPI/OpenACC e MPI/CUDA do método HOPMOC em um *cluster* de máquinas *multicore* e placas aceleradoras NVIDIA K20. Os resultados obtidos indicam um ganho significativo de desempenho com essas abordagens, devido as características intrinsecamente paralelas do HOPMOC. Ainda faz-se nescessário, no entanto, uma análise mais profunda do comportamento das trocas de mensagem MPI e ocupação das unidades funcionais das placas aceleradoras para melhorar o *speedup*. Para tal, técnicas de otimização de código CUDA deverão ser consideradas. Novos esquemas de subdivisão do domínio serão estudados de forma a avaliar o impacto do custo de troca de mensagens no método. Além disso, os resultados apresentados em [10] demonstram que o uso da interpolação para o cálculo da informação referente ao "pé" da linha característica reduz a precisão do método. Técnicas baseadas na diminuição da variação total do fluxo (TVD-*Total Variation Diminishing*) são frequentemente utilizadas na redução do erro referente a interpolação do termo convectivo no instante de tempo precedente. Uma

associação desses esquemas com o HOPMOC deverá ser avaliada, visando a redução do erro inserido pela fase de interpolação.

Agradecimentos

Os autores agradecem à Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo apoio financeiro ao projeto e ao Sistema Nacional de Processamento de Alto Desempenho (SINAPAD) por disponibilizar o ambiente computacional para as simulações apresentadas.

Referências

[1] Kischinhevsky, M.: An operator splitting for optimal message-passing computation of parabolic equation with hyperbolic dominance. SIAM Annual Meeting, Missouri (1996)

[2] Boonkkamp, J. H. M.T.J., Verwer, J. G.: On the odd-even hopscotch scheme for the numerical integration of time-dependent partial differential equations. *Appl. Num. Math*, 3 (1987) 183-193.

[3] Hansen, J., Matthey, T., Sorevik, T.: A Parallel Split Operator Method for the Time Dependent Schrodinger Equation. *Lecture Notes in Computer Science: Recent Advances in Parallel Virtual Machine and Message Passing Interface.* (2003) 2840, 503-510.

[4] Yanenko, N. N.: "The Method of Fractional Steps."Springer-Verlag, New York (1970).

[5] Li, D., Zhou, Z., Wang, Q.: A hybrid MPI/OpenMP model based on DDM for large-scale partial differential equations. IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, 1839-1843, Liverpool (2012).

[6] Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: *High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing*, (2011) 562-575.

[7] Mininni, P.D., Rosenberg, D., Reddy, R., Poquet, A.: A hybrid MPI/OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Computing*, (2011) 316-326.

[8] Kirk, D.B., Hwu, W.W.: "Programming Massively Parallel Processors: A Hands-on Approach."Elsevier Inc. (2010).

[9] Kischinhevsky, M.: A spatially decoupled alternating direction procedure for convection-diffusion equations. Proceedings of the XXth CILAMCE-Iberian Latin American Congress on Numerical Methods in Engineearing (1999)

[10] Oliveira, S., Gonzaga, S.L., Kischinhevsky, M.: Convergence analysis of the Hopmoc method. *International Journal of Computer Mathematics*, 86, (2009) 1375-1393.

[11] Cabral, F.L.: Hopmoc methods to solve convection-diffusion equations and its parallel implementation (in Portuguese). *master* Thesis, Instituto de Computação/Universidade Federal Fluminense, Brasil (2001).

[12] Richtmyer, R.D., Morton, K.W. : "Difference Methods for Initial-Value Problems"Interscience. New York (1967)

[13] Harten, A.: On a Class of High Resolution Total-Variation-Stable Finite-Difference Schemes. *SIAM J. Numer. Anal.*, 21 (1984) 1-23.