

Programação Híbrida: MPI e OpenMP

Neumar Silva Ribeiro, Luiz Gustavo Leão Fernandes

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681 – prédio 32 - PPGCC

neumar.ribeiro@acad.pucrs.br, luiz.fernandes@pucrs.br

Resumo. Este artigo tem por objetivo analisar e adquirir experiências de outros trabalhos relativos à programação híbrida, que misture os paradigmas de programação de troca de mensagens e memória compartilhada. MPI e OpenMP, são os dois padrões utilizados na maioria das situações para explorar paralelismo em clusters de máquinas multiprocessadas, ou ainda, em clusters de máquinas com hierarquia de memória, e.g. máquinas NUMA (Non-Uniform Memory Access).

1. Introdução

Usualmente, utiliza-se o paradigma de troca de mensagens quando se está programando uma arquitetura do tipo *cluster* ou até mesmo uma máquina MPP (*Massive Parallel Processors*). Por outro lado, quando se deseja programar uma máquina multiprocessada, como os computadores SMPs (*Symmetric Multiprocessor*), utiliza-se um paradigma de memória compartilhada. No entanto, o desenvolvimento de novas tecnologias possibilitou a criação de *clusters* com nós multiprocessados.

Agora, com os nós de computação destes *clusters* compostos de mais de um processador ou *core*, e compartilhando a memória, é natural questionar a possibilidade de mudar os projetos de desenvolvimento de *software*. Antes estes projetos exploravam apenas o paradigma de troca de mensagens, de modo que agora também explorarem o paradigma de memória compartilhada. Códigos de troca de mensagens puros são, obviamente, compatíveis com *clusters* de nós multiprocessados. O ponto é que a comunicação por troca de mensagens dentro de um nó SMP, ou seja, na presença de uma memória compartilhada, não é necessariamente a solução mais eficiente. No amplo espectro de soluções possíveis para o desenvolvimento de código híbrido para memória compartilhada e memória distribuída, a utilização da dupla MPI [8] e OpenMP [7] está emergindo como um padrão de fato [2]. A maioria dos códigos híbridos MPI e OpenMP são baseados em um modelo de estrutura hierárquica, que torna possível a exploração de grãos grandes e médios de paralelismo no nível de MPI, e grão fino no paralelismo no nível do OpenMP. O objetivo é claramente tirar vantagens das melhores características de ambos os paradigmas de programação.

Apesar de mesclar dois diferentes modelos de paralelismo, o modelo híbrido com MPI e OpenMP pode ser compreendido e programado sem necessidade de uma longa formação específica. Afinal, ambos MPI e OpenMP, são dois padrões bem estabelecidos e possuem sólida documentação. No entanto, deve ser buscada a utilização de um modelo híbrido sempre levando em consideração a arquitetura do *hardware* que será utilizado.

Os nós desses *clusters* podem ainda ser máquinas NUMA (*Non-Uniform Memory Access*). Estas máquinas com acesso não uniforme à memória possibilitam que o desenvolvedor especifique a localidade de alocação da memória de modo a posicioná-la o mais próximo possível do núcleo onde estará sendo executado o processo.

O objetivo desse artigo é adquirir experiências de trabalhos já realizados sobre programação híbrida, como por exemplo, [5] e [10], para que possa ser entendido de forma mais clara e objetiva todas as vantagens e desvantagens desse modelo de desenvolvimento. Ainda que o objetivo final seja o desenvolvimento de programas híbridos para *clusters* de máquinas NUMA, serão também consideradas soluções que se utilizam de *clusters* de máquinas SMPs. Será feito um estudo de caso do trabalho *A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization* [5] que aborda um estudo sobre implementação paralela híbrida do algoritmo FDTD (*Finite Difference Time Domain*), que é uma técnica de modelagem computacional para resolver problemas eletrodinâmicos. Por fim, será projetada a utilização dos conhecimentos adquiridos com esse estudo, para a extensão da solução NUMA-ICTM [1] para uma versão híbrida.

O restante deste artigo está organizado da seguinte maneira a partir dessa introdução: a segunda seção apresenta a definição de programação híbrida. Na seção 3 serão destacados dois modelos de programação híbrida. Já na seção 4 serão descritas as vantagens e desvantagens da programação híbrida. O estudo de caso será apresentado na seção 5 e na seção 6 será projetada a utilização do conhecimento adquirido. Por fim, será feito uma conclusão.

2. Programação Híbrida: MPI e OpenMP

Conforme mencionado, *clusters* de nós multiprocessados de memória compartilhada estão se tornando cada vez mais populares na comunidade de computação de alto desempenho. Assim, o foco de programação está se deslocando lenta e firmemente da computação de nós que possuem arquiteturas de memória distribuída, para arquiteturas com compartilhamento de memória. Estas arquiteturas incluem uma grande variedade de sistemas, desde arquiteturas em que a interligação do nó fornece um único espaço de endereçamento de memória, até sistemas com capacidade de acesso à memória remota direta (RDMA – *Remote Direct Memory Access*).

O modelo de programação híbrido MPI e OpenMP pode ser explorado com sucesso em cada um destes sistemas. De fato, os princípios de códigos paralelos híbridos são adequados para sistemas que vão desde um nó SMP até grandes *clusters* com nós SMPs, ou mesmo para máquinas NUMA, e ainda para *clusters* de máquinas NUMA. No entanto, a otimização do código híbrido é bastante diferente em cada classe de sistema. No restante desta seção, vamos considerar as características específicas do MPI e do OpenMP, descrevendo os modelos que podem ser adotados para a sua utilização em conjunto. Por fim, será feito um breve levantamento dos prós e contras da programação híbrida.

2.1. MPI

MPI (*Message-Passing Interface*) [8] é uma especificação de uma biblioteca de interface de troca de mensagens. MPI representa um paradigma de programação paralela

no qual os dados são movidos de um espaço de endereçamento de um processo para o espaço de endereçamento de outro processo, através de operações de troca de mensagens.

As principais vantagens do estabelecimento de um padrão de troca de mensagens são a portabilidade e a facilidade de utilização. Em um ambiente de comunicação de memória distribuída ter a possibilidade de utilizar rotinas em mais alto nível ou abstrair a necessidade de conhecimento e controle das rotinas de passagem de mensagens, *e.g. sockets*, é um benefício bastante razoável. E como a comunicação por troca de mensagens é padronizada por um fórum de especialistas, é correto assumir que será provido pelo MPI eficiência, escalabilidade e portabilidade.

MPI possibilita a implementação de programação paralela em memória distribuída em qualquer ambiente. Além de ter como alvo de sua utilização as plataformas de *hardware* de memórias distribuídas, MPI também pode ser utilizado em plataformas de memória compartilhada como as arquiteturas SMPs e NUMA. E ainda, torna-se mais aplicável em plataformas híbridas, como *cluster* de máquinas NUMA. Todo o paralelismo em MPI é explícito, ou seja, de responsabilidade do programador, e implementado com a utilização dos construtores da biblioteca no código.

2.2. OpenMP

OpenMP [7] é uma API (*Application Program Interface*) para programação em C/C++, Fortran entre outras linguagens, que oferece suporte para programação paralela em computadores que possuem uma arquitetura de memória compartilhada. O modelo de programação adotado pelo OpenMP é bastante portável e escalável, podendo ser utilizado numa gama de plataformas que variam desde um computador pessoal até supercomputadores. OpenMP é baseado em diretivas de compilação, rotinas de bibliotecas e variáveis de ambiente.

O OpenMP provê um padrão suportado por quase todas as plataformas ou arquiteturas de memória compartilhada. Um detalhe interessante e importante é que isso é conseguido utilizando-se um conjunto simples de diretivas de programação. Em alguns casos, o paralelismo é implementado usando 3 ou 4 diretivas. É, portanto, correto afirmar que o OpenMP possibilita a paralelização de um programa sequencial de forma amigável.

Como o OpenMP é baseado no paradigma de programação de memória compartilhada, o paralelismo consiste então de múltiplas *threads*. Assim, pode-se dizer que OpenMP é um modelo de programação paralelo explícito, oferecendo controle total ao programador.

2.3. Programação Híbrida

A principal motivação para utilizar programação híbrida MPI e OpenMP é tirar partido das melhores características de ambos os modelos de programação, mesclando a paralelização explícita de grandes tarefas com o MPI com a paralelização de tarefas simples com o OpenMP. Em alto nível, o programa está hierarquicamente estruturado como uma série de tarefas MPI, cujo código sequencial está enriquecido com diretivas OpenMP para adicionar *multithreading* e aproveitar as características da presença de memória compartilhada e multiprocessadores dos nós. As Figuras 1, 2 e 3 mostram

respectivamente: (i) uma maneira de explorar o hardware disponível de um *cluster* SMP de 2-CPUs para uma decomposição somente com MPI, (ii) para uma decomposição somente com OpenMP, (iii) para uma decomposição hierárquica híbrida com MPI e OpenMP. É importante ressaltar que a decomposição hierárquica não é a única maneira de misturar MPI e OpenMP.

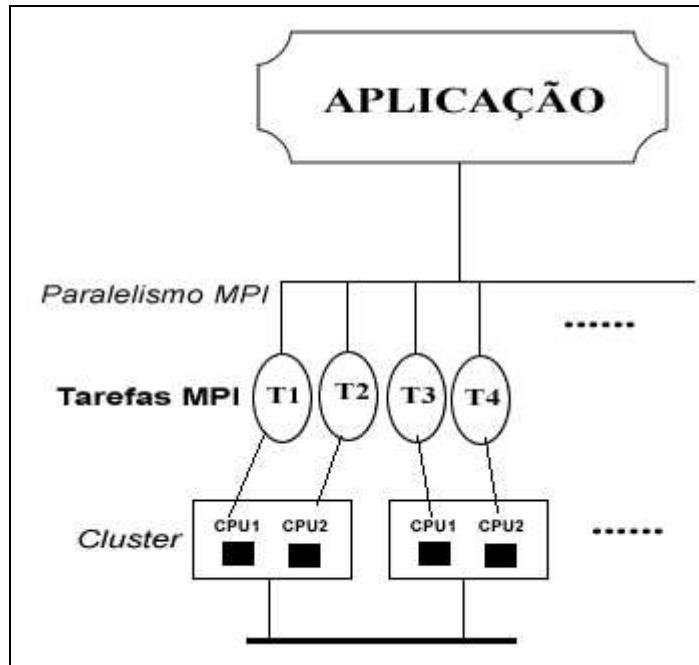


Figura 1 – Decomposição apenas com MPI

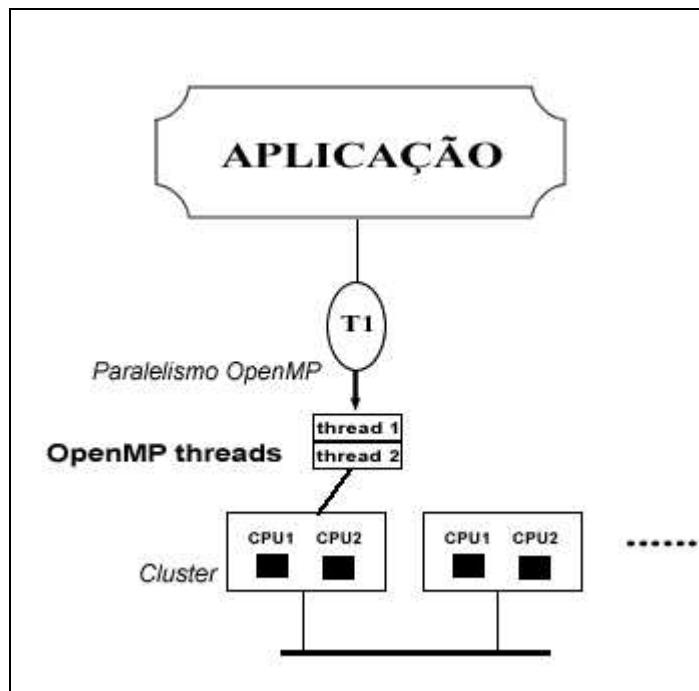


Figura 2 – Decomposição apenas com OpenMP

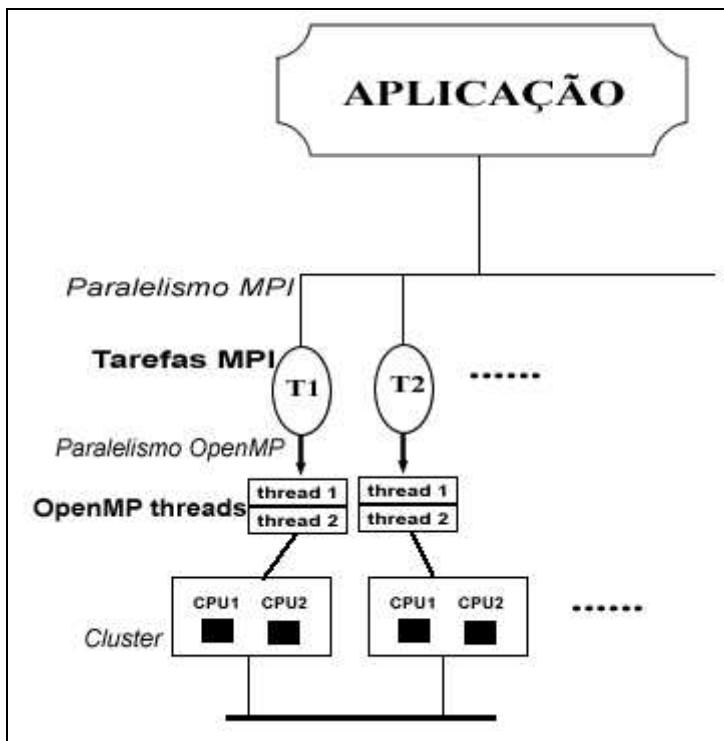


Figura 3 – Decomposição Híbrida com MPI e OpenMP

De fato, um exame mais aprofundado da interação dos modelos de programação é necessário. Afinal, o uso de OpenMP acrescenta *multithreading* aos tradicionais processos MPI , o que de certa maneira, deixa o desenvolvimento dos programas mais complexos.

3. Modelos de Programação Paralela em Plataformas Híbridas

Nessa seção serão apresentados alguns modelos de programação possíveis para plataformas híbridas, tentando deixar mais evidente a importância do casamento correto entre o modelo de programação escolhido para cada plataforma.

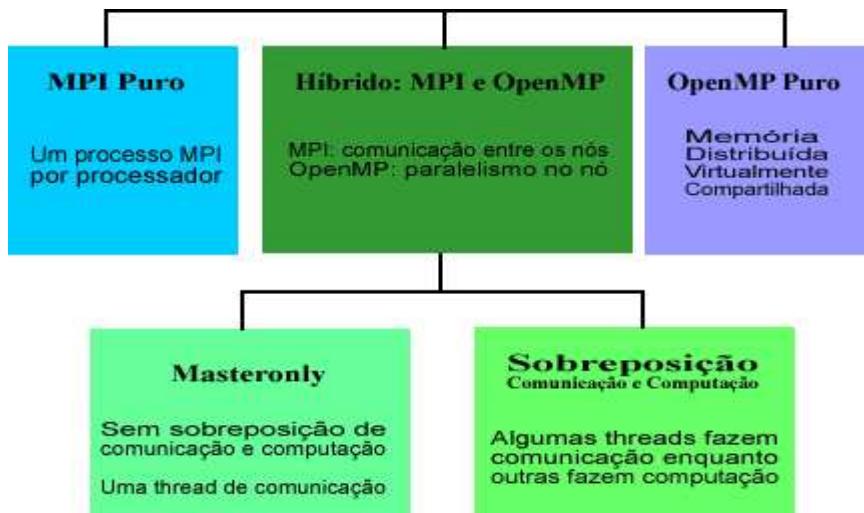


Figura 4 – Modelos de Programação para Plataformas Híbridas, adaptado de [2]

A Figura 4 mostra uma taxonomia de modelos de programação paralela para plataformas híbridas. Esse autor adicionou a nomenclatura OpenMP Puro porque tecnologias como *Intel Cluster OpenMP* [9] permitem o uso de OpenMP para a paralelização em *clusters* com memória distribuída, porém a análise desta ferramenta está fora do escopo deste trabalho.

Esta visão ignora os detalhes sobre como exatamente as *threads* e os processos de um programa híbrido serão mapeadas de forma hierárquica no *hardware*. Os problemas de incompatibilidade que podem ocorrer serão discutidos adiante na seção 4.1.

3.1. MPI Puro

Do ponto de vista do programador, o MPI puro ignora o fato do nó possuir vários núcleos compartilhando a memória. Esse modelo pode ser empregado de imediato sobre os *clusters* sem alterações no código. Além disso, não é necessária para a biblioteca MPI que as demais camadas de *software* suportem aplicações *multithread*, o que simplifica a implementação.

Por outro lado, um modelo de programação MPI puro implicitamente assume que a troca de mensagens é o paradigma a ser usado para todos os níveis de paralelismo disponível na aplicação. Além disso, toda comunicação entre os processos no mesmo nó vai ocorrer através das camadas de *software* do MPI, o que aumenta a sobrecarga da rede. Mesmo sendo esperado que a biblioteca seja capaz de usar "atalhos" através da memória compartilhada, para a comunicação dentro do nó. Essas otimizações geralmente estão fora da influência do programador.

3.2. Híbrido *Masteronly*

O modelo híbrido *masteronly* utiliza um processo MPI por nó e *multithread* OpenMP sobre os *cores* do nó, sem chamadas MPI dentro de regiões paralelas. Assim, apenas a *thread* principal faz chamadas MPI, sendo responsável então, pela comunicação. As demais *threads* apenas realizam computação.

Uma amostra simples da organização do programa seria o pseudocódigo do modelo híbrido *masteronly* apresentado a seguir:

```
for (i = 1 ... N)
(
    #pragma omp paralelo
    (
        /* Código numérico */
    )
    /* Na thread mestre */
    MPI_Send (envia dados em massa para as áreas de dados em outros nós)
    MPI_Recv (dados dos vizinhos)
)
```

Pseudocódigo: Modelo híbrido *masteronly*

Como não há transmissão de mensagens dentro do nó, as otimizações do MPI não são necessárias. Naturalmente, as partes do OpenMP devem ser otimizadas para a arquitetura do nó pelo programador, por exemplo, empregando cuidados com a

localidade dos dados na memória em nós NUMA, ou usando mecanismos de afinidade de *threads* [1].

Existem, no entanto, alguns problemas ligados com o modo *masteronly*:

- Todas as outras *threads* estarão ociosas durante a fase de comunicação da *thread* principal, o que poderá levar a um forte impacto de sobrecarga de comunicação;
- A capacidade de comunicação entre os nós pode ficar subutilizada, em função de apenas uma *thread* fazer chamadas MPI;

3.3. Híbrido com Sobreposição de Comunicação e Processamento

Uma maneira de evitar subutilização das *threads* de computação durante a comunicação MPI é separar uma ou mais *threads* para fazer a comunicação em paralelo. Um exemplo de estrutura do programa seria o pseudocódigo do modelo híbrido com sobreposição apresentado a seguir:

```
if (my_thread_ID <...)  
    /* threads de comunicação */  
    /* dados para comunicação */  
    MPI_Send (envia dados)  
    MPI_Recv (recebe dados)  
) else (  
    /* threads de computação */  
    /* Executar o código que não depende de dados que estão sendo trocados */  
)  
    /* todas as threads: */  
    /* Executar o código que precisa dos dados */
```

Pseudocódigo: Sobreposição de comunicação e computação

Uma possível razão para usar mais de uma *thread* de comunicação pode surgir se uma única *thread* não conseguir saturar a rede de comunicação entre os nós. Há, contudo, uma relação custo/benefício a ser gerenciada, porque quanto mais *threads* utilizadas para troca de mensagens, serão menos *threads* disponíveis para processamento do problema.

3.4. OpenMP Puro em Clusters

Algumas pesquisas têm investido na implementação de *software* para compartilhamento virtual de memória distribuída o que permite programação similar à utilizada para memória compartilhada em sistemas de memória distribuída. Desde 2006 a Intel oferece o *Cluster OpenMP* [9] que é um sistema que permite o uso de programas feitos com OpenMP em um *cluster*. O mecanismo de *software* utilizado pelo *Cluster OpenMP* é conhecido como memória compartilhada distribuída, DSM (*Distributed Shared Memory*) ou DVSM (*Distributed Virtual Shared Memory*).

Com este sistema, o OpenMP torna-se um modelo de programação possível para *clusters*. É, em certa medida, um modelo híbrido, sendo idêntico ao OpenMP dentro de um nó de memória compartilhada, mas empregando um protocolo sofisticado que mantém automaticamente páginas de memória "compartilhadas" coerentes entre os nós.

Com *Cluster OpenMP*, ocorrem frequentemente sincronização das páginas por todos os nós. Isso pode potencialmente tornar-se mais custoso do que utilizar MPI [2].

4. Vantagens e Desvantagens da Programação Híbrida

Pelo menos na teoria, as vantagens da programação híbrida MPI e OpenMP são interessantes. A técnica permite obter uma divisão de tarefas com MPI, fazendo o processamento nos nós usando OpenMP em máquinas de memória compartilhada. No entanto, nem sempre uma versão híbrida apresenta desempenho superior a uma versão feita com os recursos do MPI puro [6]. Contudo, pode-se adaptar programas que já estão desenvolvidos em MPI para verificar as melhorias obtidas com a utilização do OpenMP nos nós multiprocessados.

Provavelmente, a melhor maneira de entender se um projeto de código híbrido pode ser bem sucedido é fazer uma análise cuidadosa das características do problema. Após isso, é preciso mapeá-lo conforme a arquitetura disponível para resolução do problema. Situações com bom potencial para o projeto de código híbrido incluem o seguinte:

- Programas que tenham uma má escala com o aumento de tarefas (processos MPI);
- Programas MPI com balanceamento de carga mal ajustados (escolher os pontos certos onde as *threads* OpenMP podem ser implementadas);
- Problemas de paralelismo com grão-fino, mais adequado para uma utilização de OpenMP do que MPI;
- Programas com dados replicados, quando se utiliza OpenMP, é possível atribuir um único exemplar dos dados em cada nó, para ser acessado por todas as *threads* do no local;
- Programas MPI, com restrições sobre o número de tarefas;
- Programas que serão executados em máquinas com comunicação entre os nós de baixo desempenho (rede de baixa capacidade);
- Programas que serão executados em máquinas com alta latência nas interconexões.

Nos demais casos, os códigos híbridos são passíveis de ter igual ou até pior desempenho do que uma solução MPI puro. Isto deve-se ao número de ineficiências ligada à abordagem híbrida. De fato, um programa decomposto hierarquicamente tem as chamadas MPI realizadas fora de regiões paralelas. Isto significa que sempre que uma chamada MPI é necessária, terá apenas uma *thread* ativa por nó, e isso é claramente uma fonte de sobrecarga. Adicionais sobrecargas são devidas a causas mais sutis. Entre estas, deve-se mencionar possível aumento dos custos de comunicação, devido à impossibilidade de uma *thread* utilizar toda a largura de banda da interconexão do nó. Na prática, a abordagem híbrida é conveniente sempre que as vantagens se sobressaem. Mais detalhes podem ser encontrados em [6].

4.1. Problemas de Incompatibilidade

Os modelos de programação em *clusters* de nós multiprocessados têm vantagens, mas também sérias desvantagens relativas aos problemas de incompatibilidade entre o modelo híbrido de programação e a arquitetura híbrida:

- Com MPI puro, para minimizar a comunicação entre os nós, exige-se que o domínio de aplicação coincida com a topologia do *hardware*;
- MPI puro também introduz a comunicação entre os nós, que no caso de nós de SMPs pode ser omitido se for utilizado programação híbrida;
- Por outro lado, programação MPI e OpenMP não é capaz de utilizar a largura de banda completa do nó, quando se utiliza apenas uma *thread* para responder pela comunicação;
- Utilizando modelo *masteronly*, todas as *threads* que não fazem comunicação ficam paradas sem executar processamento no momento da troca de mensagens entre os nós.
- Tempo de CPU também é desperdiçado, se todos os processadores de um nó SMP comunicarem ao mesmo tempo, pois alguns processadores já são capazes de saturar a largura de banda do nó.

A sobreposição de comunicação e computação é uma maneira para tentar obter uma utilização melhor do *hardware*, mas:

- Requer um grande esforço do programador para separar no programa as *threads* responsáveis pela comunicação e as *threads* responsáveis pelos dados;
- *Threads* de comunicação e *thread* de computação devem ser balanceadas.

Mais informações detalhadas sobre problemas são apresentadas em [3].

5. Estudo de Caso

O trabalho *A Novel FDTD Application Featuring OpenMP/ MPI Hybrid Parallelization* [5] apresenta a teoria e o funcionamento da solução implementada do algoritmo FDTD (*Finite Difference Time Domain*), que é uma técnica de modelagem computacional para resolver problemas eletrodinâmicos.

Nesta seção, serão apresentados os comparativos que o autor faz entre as implementações dos algoritmos: sequencial, MPI puro, OpenMP e híbrida [5]. O Algoritmo 1 apresenta a versão sequencial que será utilizada como base comparativa para as demais implementações. Por fim, será feita uma análise destas comparações.

Algoritmo: Sequencial-FDTD
/* inicialização dos tempos; */
/* inicializa os campos, aplica condições iniciais; */
for t = 1 to tmax do
 for i, j, k = 1 to imax, jmax, kmax do
 Atualiza os campos elétricos usando campos magnéticos;
 Atualiza os campos magnéticos usando campos elétricos;
 Atualiza os campos das bordas, aplica condições de borda;
 end
end

Algoritmo 1 – Sequencial FDTD

5.1. FDTD e Paralelização MPI

Em um primeiro momento, é feita uma análise sobre a versão da solução implementada com a paralelização MPI para FDTD. O paradigma de programação por troca de mensagens assume que as estruturas de dados não estão compartilhadas, mas sim divididas entre os vários nós utilizados. Essa idéia de divisão de armazenamento é utilizada como uma técnica de decomposição de domínio do problema. Uma vez que as atualizações de campos em um ponto da malha dependem de utilizar valores de campos em torno dos pontos, a troca de informações é necessária nos domínio de bordas. O Algoritmo 2 implementa a decomposição do domínio utilizando MPI para a troca de mensagens, e é uma modificação do algoritmo sequencial FDTD.

O número de trocas de mensagens requeridas cresce com a área total da superfície dos domínios, o que é proporcional tanto para o tamanho total da malha quanto para o número de processadores (ou domínios distintos) utilizado no cálculo. Este é um custo indireto que é trazido pela decomposição em domínios. Por outro lado, a paralelização MPI traz benefícios não só porque permite que se aproveite o paralelismo de máquina para reduzir o tempo de execução, mas também tem os benefícios da alocação distribuída e armazenamento distribuído, o que permite a utilização desta solução para problemas maiores. No FDTD paralelo com MPI, o armazenamento total usado é dividido quase que igualmente entre todos os processos MPI. Assim, o armazenamento por processo é pequeno, e torna-se ainda menor à medida que o número de processos aumenta.

```
Algoritmo: MPI-FDTD
/* inicialização dos tempos; */
/* inicializa os campos, aplica condições iniciais; */
for t = 1 to tmax do
    for i, j, k = 1 to imax, jmax, kmax do
        Utiliza MPI para trocar informações dos campos magnéticos com vizinhos;
        Atualiza os campos elétricos usando campos magnéticos;
        Utiliza MPI para trocar informações dos campos elétricos com vizinhos;
        Atualiza os campos magnéticos usando campos elétricos;
        Atualiza os campos das bordas, aplica condições de borda;
    end
end
```

Algoritmo 2: FDTD paralelizado com MPI

5.2. FDTD e Paralelização OpenMP

Segundo o autor de [5], as atualizações necessárias para a solução do problema possuem características de modo que não ocorram dependências de dados entre as operações. Sendo assim, o problema pode ser mapeado para um ambiente *multithread*. Também não há problema de sobrecarga com relação ao domínio de distribuição, pois o armazenamento é compartilhado e acessível por todas as *threads*. O padrão de programação paralela para memória compartilhada OpenMP foi utilizado para implementar uma solução para o FDTD. O código FDTD mapeia naturalmente ao paradigma de memória compartilhada, logo usufrui dos benefícios mencionados acima. No entanto, existem algumas desvantagens em utilizar memória compartilhada para o FDTD:

- OpenMP precisa de suporte para alocação distribuída de estruturas compartilhadas, podendo causar gargalos em alguns sistemas.;
- Uma vez que o conjunto de dados é muito grande, há muitos *misses* na *cache*, afetando severamente o desempenho e a escalabilidade;
- Otimizações nos códigos OpenMP não são simples, demandando alto conhecimento do programador.

Para minimizar esses problemas o autor utilizou recursos de alocação disponíveis na máquina da SGI utilizada nos testes [5]. A implementação do OpenMP da SGI (*Silicon Graphics, Inc*) aceita parametrizações que afetam o modo como o sistema aloca os dados e como os dados são transferidos em tempo de execução para minimizar a latência de acesso aos dados na memória.

É possível configurar o sistema para utilizar algumas diferentes políticas de alocação de memória, entre elas: a política de alocação *first-touch*, que distribui os dados para armazenamento convenientemente alocado para todos os processadores em paralelo. Outra política é alocar cada página de memória em diferentes regiões da memória de forma distribuída, isto é, a utilizando a chamada política *round-robin*. Dessa forma, a latência de acesso não é otimizada, mas todos os gargalos causados pela utilização maciça de um único espaço da memória física é impedido. A terceira política verificada foi a *predetermined storage map*, que requer que seja executado de forma separada a geração de um mapa para alocação. Isso pode minimizar a latência de acesso a memória, porém adiciona mais um processo na solução como um todo. O artigo decidiu utilizar a política *round-robin*.

Para tentar amenizar a segunda desvantagem, foram utilizados alguns métodos para reduzir a exigência de armazenamento de dados (para conter o número total de *cache misses*), bem como o rearranjo dos cálculos para fazer melhor uso de características de *prefetch* do processador. Por último, algumas técnicas de engenharia de algoritmo foram utilizadas para melhorar o rearranjo das equações do FDTD para aumentar eficiência e otimizar essa versão.

5.3. FDTD Híbrido: MPI e OpenMP

Máquinas de memória compartilhada não escalam para além de um certo número de processadores em função das sobrecargas conhecidas. As máquinas de memória distribuídas estão sujeitas às latências da rede durante as trocas de mensagens. Assim, para obter uma aplicação paralela do FDTD mais eficiente foi utilizado um conceito híbrido, misturando programação que utiliza MPI e OpenMP.

Neste caso, os cálculos necessários para resolução do problema foram divididos em domínios de armazenamento. Assim cada domínio foi enviado para um nó do *cluster* utilizando MPI. Dentro do nó, os cálculos são feitos com múltiplas *threads*. Isso reduz a sobrecarga de decomposição de domínio da versão MPI, uma vez que não terá muitos domínios como na versão MPI puro.

Também é necessária a troca de informações das bordas dos domínios entre os nós para os cálculos. Estas trocas são feitas utilizando MPI em dois estágios diferentes: no primeiro estágio são trocadas informações referentes aos campos magnéticos, após é feita a computação necessária para a atualização dos campos magnéticos; no segundo

estágio são trocadas as informações referentes aos campos elétricos, e após é feita a computação necessária para atualização dos campos elétricos. Essas computações realizadas nos nós são sempre paralelizadas utilizando OpenMP. A versão híbrida do FDTD apresentada no algoritmo 3, ajuda a compreender essa decomposição híbrida.

```
Algoritmo: Híbrido-FDTD
/* inicialização dos tempos; */
/* Usando OpenMP multi-threading inicializa os campos, aplica condições iniciais; */
for t = 1 to tmax do
    for i, j, k = 1 to imax, jmax, kmax do
        Utiliza MPI para trocar informações dos campos magnéticos com vizinhos;
        Usando OpenMP multi-threading: atualiza os campos elétricos usando campos magnéticos;
        Utiliza MPI para trocar informações dos campos elétricos com vizinhos;
        Usando OpenMP multi-threading: atualiza os campos magnéticos usando campos elétricos;
        Usando OpenMP multi-threading: atualiza os campos das bordas, aplica condições de borda;
    end
end
```

Algoritmo 3 – MPI-OpenMP híbrido para FDTD

6. NUMA-ICTM utilizando Programação Híbrida

O objetivo do próximo trabalho é estender uma aplicação modelada unicamente para memória compartilhada em uma máquina com hierarquia de memória, para que essa aplicação utilize também o paradigma de troca de mensagens e que possa tirar proveito de um *cluster* de máquinas multiprocessadas NUMA.

A aplicação é o ICTM (*Interval Categorizer Tessellation Model*) [4], que é um modelo multi-camada desenvolvido para categorização de regiões geográficas utilizando informações extraídas de imagens de satélites. Porém, a categorização de grandes regiões requer um alto poder computacional. Além disso, o processo de categorização como um todo resulta em um uso intensivo de memória. Consequentemente, estas duas características principais motivam o desenvolvimento de uma solução paralela para esta aplicação com intuito de categorizar grandes regiões de forma mais rápida.

O trabalho *NUMA-ICTM: Uma versão paralela do ICTM explorando estratégias de alocação de memória para máquinas NUMA* [12], realizado no GMAP (Grupo de Modelagem de Aplicações Paralelas) da Faculdade de Informática da PUCRS, implementou uma versão paralela do ICTM utilizando OpenMP e a biblioteca MAI (*Memory Affinity Interface*) [13]. A biblioteca MAI é utilizada para facilitar a alocação da memória levando em conta a localidade dos dados, o que é crucial para se obter um bom desempenho em máquinas NUMA [11].

Com base nos bons resultados obtidos no trabalho NUMA-ICTM e nos avanços da programação híbrida e dos *clusters* de máquinas NUMA, é possível construir uma implementação do ICTM para uma modelagem híbrida em *cluster* de máquinas NUMA utilizando MPI e OpenMP, e conseguir melhorar os resultados conseguidos com o NUMA-ICTM.

A Figura 5 ilustra uma matriz com as informações do ICTM dividida em blocos, de modo que cada nó do *cluster* receba um destes blocos para realizar o processamento. Este processamento requer que os nós troquem informações das bordas dos blocos, e

isto será feito utilizando MPI. Já o processamento do bloco em cada um dos nós multiprocessados será paralelizado utilizando OpenMP.

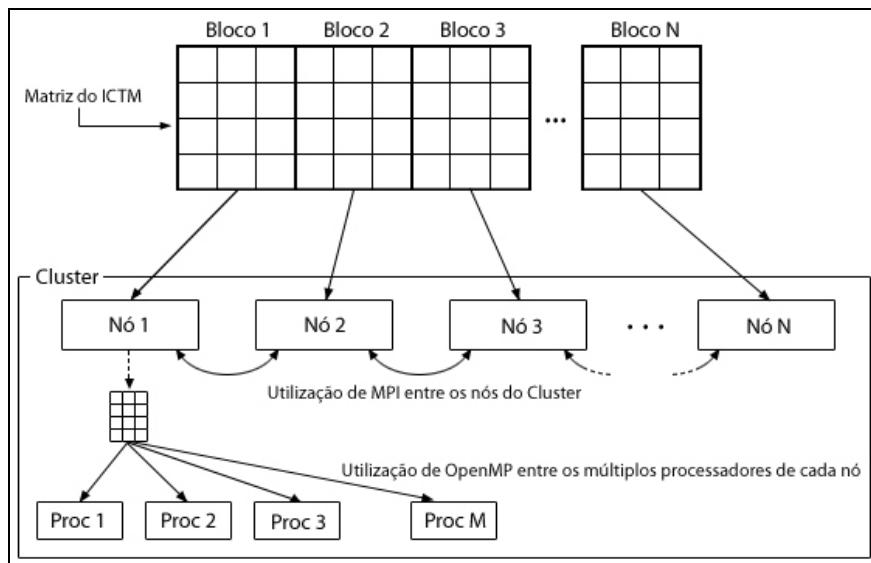


Figura 5 – Divisão dos blocos da matriz do ICTM utilizando hierarquia de memória

Aumentando o paralelismo através da biblioteca MPI, pretende-se alcançar resultados ainda melhores, explorando toda a capacidade dos novos modelos de *clusters* de multiprocessadores disponibilizados para computação de alto desempenho.

7. Conclusão

Esse artigo teve como principal objetivo conhecer trabalhos já realizados sobre programação híbrida para saber das vantagens e desvantagens desse modelo de desenvolvimento. O estudo de caso do trabalho *A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization* foi muito interessante porque aborda uma estratégia de decomposição de problema bastante semelhante a proposta de decomposição que será utilizada para a programação híbrida da NUMA-ICTM. Assim, as experiências adquiridas com esse trabalho certamente serão de suma importância para os trabalhos futuros.

Referências

- [1] Castro, M; Fernandes, L.G.; Pousa, C.; Mehaut, J.-F.; de Aguiar, M.S.; NUMA-ICTM: A parallel version of ICTM exploiting memory placement strategies for NUMA machines. Parallel & Distributed Processing. IPDPS 2009. IEEE International Symposium on 23-29 May 2009 p. 1-8.
- [2] Rabenseifner, R., Hager, G., and Jost, G. 2009. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In Proceedings of the 2009 17th Euromicro international Conference on Parallel, Distributed and Network-Based Processing - Volume 00 (February 18 - 20, 2009). PDP. IEEE Computer Society, Washington, DC, p. 427-436.

- [3] Rabenseifner, R., Hager, G., and Jost, G. 2009. Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. Proceedings of the Cray Users Group Conference 2009 (CUG 2009), Atlanta, GA, USA, May 4-7, 2009.
- [4] Aguiar, M. S. de et al. The Multi-layered Interval Categorizer Tesselation-based Model. In: GeoInfo'04: Proceedings of the 6th Brazilian Symposium on Geoinformatics. Campos do Jordão, São Paulo, Brazil: Instituto Nacional de Pesquisas Espaciais, 2004. p. 437-454.
- [5] Su, M. F., El-Kady, I., Bader, D. A., and Lin, S. 2004. A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization. In Proceedings of the 2004 international Conference on Parallel Processing (August 15 - 18, 2004). ICPP. IEEE Computer Society, Washington, DC, p. 373-379.
- [6] Aversa, R., Di Martino, B., Rak, M., Venticinque, S., and Villano, U. 2005. Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Comput.* 31, 10-12 (Oct. 2005), p. 1013-1033.
- [7] OpenMP Application Program Interface Version 3.0 Complete Specifications. May, 2008. Disponível em <http://www.openmp.org/mp-documents/specs30.pdf>. Acessado em 25 de novembro de 2009.
- [8] MPI: A Message-Passing Interface Standard Version 2.1. Disponível em: www.mpi-forum.org/docs/mpi21-report.pdf. Acessado em: 25 de novembro de 2009.
- [9] Cluster OpenMP for Intel Compilers. Disponível em <http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers/>. Acessado em: 25 de novembro de 2009.
- [10] Chorley, M. J., Walker, D. W., and Guest, M. F. 2009. Hybrid Message-Passing and Shared-Memory Programming in a Molecular Dynamics Application On Multicore Clusters. *Int. J. High Perform. Comput. Appl.* 23, 3 (Aug. 2009), p. 196-211.
- [11] Pousa, C. ; Castro, M. B. ; Méhaut, J.-F. ; Carissimi, A. ; Fernandes, L. G. . Memory Affinity for Hierarchical Shared Memory Multiprocessors. In: SBAC-PAD - International Symposium on Computer Architecture and High Performance Computing, 2009, São Paulo. Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing. Los Alamitos, CA, EUA : IEEE Computer Society, 2009. p. 59-66.
- [12] Castro, M. B., NUMA-ICTM: Uma versão paralela do ICTM explorando estratégias de alocação de memória para máquinas NUMA, Dissertação de Mestrado – Pontifícia Universidade Católica do Rio Grande do Sul, 2008.
- [13] Pousa Ribeiro, Christiane and M'ehaut, Jean-Francois, MAI: Memory Affinity Interface, Laboratoire d'Informatique de Grenoble - LIG - INRIA - Universit'e Pierre Mend`es-France - Grenoble II - Universit'e Joseph Fourier - Grenoble I - Institut Polytechnique de Grenoble. Disponível em: <http://hal.inria.fr/inria-00344189/en/> Acessado em: 25 de Novembro de 2009.