Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 2999-3017 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

PROCESSAMENTO PARALELO COM OPENMP EM UM SIMULADOR DINÂMICO DE LINHAS DE ANCORAGEM E *RISERS, PARTE II*

Heleno P. Bezerra Neto, Joseanderson A. C. Costa, Fábio M. G. Ferreira and **Eduardo S. S. Silveira**

Laboratório de Computação Científica e Visualização (LCCV), Centro de Tecnologia (CTEC), Universidade Federal de Alagoas (UFAL), Campus A. C. Simões, Av. Lourival de Melo Mota S/N, Tabuleiro do Martins 57072-970, Maceió – AL – Brasil, helenopontes@lccv.ufal.br, http://www.lccv.ufal.br

Palavras-chave: Processamento Paralelo, Memória compartilhada, OpenMP.

Resumo. *O uso da Computação de Alto Desempenho vem sendo empregado fortemente ao longo dos últimos anos para auxiliar e possibilitar a solução de problemas complexos em diversas áreas do conhecimento. Entre as técnicas de alto desempenho mais utilizadas, destaca-se o processamento paralelo, que consiste na divisão de tarefas entre centrais de processamento. Dentro desse contexto, este trabalho apresenta um estudo sobre uma das técnicas de processamento paralelo em ambientes de memória compartilhada. Para isso, será utilizado o padrão OpenMP, que é uma ferramenta de programação paralela baseada em diretivas de compilação. O trabalho apresenta diferentes estratégias de paralelização aplicadas ao código e suas respectivas implicações nos resultados. Além das curvas de speedup, é feita uma comparação entre os resultados da versão serial e paralela. O uso do padrão OpenMP permitiu reduzir o tempo das simulações garantindo uma pequena margem de erro. Além disso, o uso desse padrão de paralelização permitiu a criação de uma versão paralela portável que exigiu pequenas mudanças na versão serial e foi capaz de acelerar consideravelmente simulações com elevado tempo de processamento em desktops comuns.*

1 INTRODUÇÃO

O uso de sistemas computacionais para simulações de problemas de engenharia tem avançado e crescido acentuadamente nos últimos anos, tendo a indústria desempenhado um papel fundamental nesses avanços. Como exemplo, pode-se destacar a parceria entre a PETROBRAS e algumas universidades do país através de projetos de pesquisa e redes temáticas. A Universidade Federal de Alagoas (UFAL), através do Laboratório de Computação Científica e Visualização (LCCV), é uma das universidades parceiras da PETROBRAS. Nos últimos anos a UFAL vem contribuindo com o desenvolvimento de *softwares* de engenharia voltados para a indústria do petróleo, a exemplo do simulador numérico de linhas de ancoragem e *risers:* DOOLINES (*Dynamics Of Offshore Lines*), desenvolvido por Ferreira (2009).

DOOLINES é um *framework* para análise dinâmica de *risers* e linhas de ancoragem construído com base no sistema PREADYN (Silveira, 2001). Esse *framework* é escrito utilizando a Programação Orientada a Objetos (POO) e é caracterizado como uma biblioteca e desde sua versão inicial é utilizado em conjunto com outros sistemas computacionais, como DYNASIM (Coelho *et al.*, 2001) e TPN (Nishimoto *et al*., 2004).

A etapa inicial para simulação de linhas ou *risers* utilizando o DOOLINES consiste na modelagem numérica do problema. Todo o cenário *offshore* e domínio do problema precisam ser convertidos em um modelo numérico. No caso específico, as linhas de ancoragem e *risers* precisam ser discretizadas em elementos finitos, pois esse simulador utiliza o Método dos Elementos Finitos para discretização das equações diferenciais. Em algumas situações de projeto, essa malha de elementos finitos precisa ser muito discretizada para se obter uma resposta satisfatória. Nesses casos, o tempo de processamento e/ou o consumo de memória podem ser muito elevado, o que pode inviabilizar algumas análises em *desktops* comuns.

Para contornar problemas como esse é comum o uso de técnicas de Computação de Alto Desempenho (CAD), também denominadas na literatura como *High Performance Computing* (HPC). Essas técnicas visam otimizar o uso dos recursos computacionais individuais e/ou viabilizar o uso em conjunto desses recursos visando reduzir o tempo de processamento e/ou uso excessivo de outros recursos computacionais, como memória e rede. Dentre as diversas técnicas de HPC existentes, destacam-se as estratégias de otimização e o processamento paralelo, sendo este último mais utilizado em problemas que envolvem elevado consumo de tempo. A estratégia baseada no processamento paralelo consiste em dividir o fluxo do programa entre múltiplas centrais de processamento, que podem estar conectadas através de ambientes de memória compartilhada, distribuída e ambientes híbridos.

No trabalho desenvolvido por Costa *et al*. (2008) foi proposta uma estratégia de paralelização utilizando o padrão OpenMP em ambientes de memória compartilhada. Naquela ocasião, foi descrito o potencial dessa ferramenta e ilustrado os ganhos que foram obtidos. Contudo, não foram discutidos problemas associados a erros numéricos e apenas uma estratégia de paralelização simplificada foi utilizada.

Este trabalho representa uma continuação dos estudos realizados por Costa *et al*. (2008) no simulador DOOLINES. Além da implementação de novas estratégias de paralelização, mais gerais e que podem ser aplicadas em outros problemas, também são discutidos aspectos associados a erros numéricos envolvidos em simulações paralelas iterativas. Uma versão inicial do simulador DOOLINES com o padrão MPI também é avaliado em ambientes de memória compartilhada e considerações sobre o impacto do tamanho da memória *cache* e simulações com SO executando a 32 e 64 bits também são discutidas.

As simulações foram realizadas em computadores com processadores de arquitetura *multicore* e os resultados são descritos através de gráficos de tempo de processamento, *speedup* e gráficos de erros numéricos.

2 O *FRAMEWORK* DOOLINES

O DOOLINES é um *framework* orientado a objetos que possibilita a análise dinâmica de *risers* e linhas de ancoragem. Entre as diversas classes que o compõe se destacam as classes `Model` e a `Intalg`. Elas são responsáveis pela concepção do modelo e pelos algoritmos de integração temporal, respectivamente.

A classe `Model`, refere-se ao modelo computacional e é utilizada para fazer a modelagem do problema a ser tratado. A partir dela, pode-se definir a malha de elementos finitos das estruturas de linha de ancoragem e *riser*, bem como as propriedades físicas e geométricas dos elementos.

Os algoritmos de integração estão presentes na classe `Intalg`. Atualmente, existem apenas algoritmos que utilizam o método explícito de integração no tempo, como Método Explícito Generalizado – α, Método de *Chung-Lee*.

3 COMPUTAÇÃO DE ALTO DESEMPENHO

Em algumas situações de projeto é necessária a construção de modelos computacionais mais discretizados (maior número de graus de liberdade). A simulação desses modelos exige um maior esforço computacional em relação ao tempo de processamento ou uso excessivo de memória. Para minimizar esses problemas é comum o uso de técnicas de alto desempenho cujos principais objetivos são: reduzir o tempo de processamento, minimizar o consumo de energia e memória, reduzir os problemas na transferência de dados por rede, entre outras funções cujo objetivo é sempre otimizar o desempenho do *software* e ou *hardware* envolvido. No desenvolvimento de um *software* (simulador) para análise de problemas de engenharia, algumas dessas técnicas podem ser aplicadas diretamente pelo programador, podendo elas ser classificadas como estratégias de otimização do código fonte e estratégias baseadas em processamento paralelo, sendo esta última abordada em detalhes neste trabalho.

Quando se trabalha com códigos muito extensos, que utilizam muitas funções ou métodos (contexto de programação orientada a objetos), a dificuldade inicial é encontrar os trechos de código com maior consumo de tempo ou memória. Essas regiões são denominadas regiões críticas e devem ser investigadas detalhadamente, pois são mais favoráveis a paralelização. A identificação dessas regiões é fundamental no processo de otimização do programa. Caso contrário, é possível que o programador destine boa parte do tempo em regiões que não apresentam elevado consumo do tempo e dessa forma, as estratégias utilizadas para otimização não terão resultados significativos.

3.1 Processamento Paralelo

O processamento paralelo é recomendado em regiões que demandam elevado esforço computacional (tempo, memória ou processamento) e consiste em dividir o fluxo do programa entre as centrais de processamento disponíveis, podendo ser núcleos (*cores*) ou processadores. Essa técnica pode ser aplicada em ambientes de memória compartilhada, distribuída e ambientes que combinam essas arquiteturas, denominados híbridos.

Em ambientes de memória distribuída, o fluxo do programa é dividido entre vários processadores que possuem recursos computacionais individuais. Nesse caso, o trabalho computacional é dividido para cada computador na forma de processos e a comunicação entre os computadores é feita através de alguma biblioteca de comunicação, como PVM

(*Processing Virtual Machine*, Pvm, 2009) e MPI (*Message Passing Interface*. Mpi, 2009). Estratégias paralelas baseadas na criação de múltiplos processos são, na maioria das vezes, utilizadas em *clusters* de computadores (ambientes formados por um conjunto de máquinas interligadas em rede e que possuem recursos computacionais individuais).

Em ambientes de memória compartilhada, o fluxo do programa é dividido entre vários núcleos de processamento/processadores que compartilham uma única memória. Nesse modelo, o trabalho computacional é dividido em *threads* (linhas de execução de um processo que compartilham a mesma memória). Como alternativa para essa estratégia de paralelização destacam-se os padrões POSIX *Threads* e o OpenMP (Costa *et al.*, 2008).

O processamento paralelo em ambientes de memória compartilhada pode ser baseado em *threading* explícito e diretivas de compilação. Em *threading* explícito o programador é o responsável por criar, definir a quantidade e destruir as *threads*, sendo a ferramenta de programação paralela mais utilizada para isso o POSIX *Threads*. No uso de diretivas de compilação, o programador é responsável por indicar onde as *threads* serão criadas e de que forma elas devem ser executadas, mas o compilador é o responsável por fazer todo o gerenciamento. A ferramenta mais comum para essa abordagem e a utilizada neste trabalho é o padrão OpenMP.

Em ambientes híbridos de programação o fluxo do programa pode ser dividido em duas camadas, a primeira relativa ao processamento distribuído e a segunda ao processamento compartilhado em cada processo.

4 O OPENMP E AS MÉTRICAS DE DESEMPENHO

O padrão OpenMP é desenvolvido e mantido pelo grupo OpenMP *Architecture Review Board* (ARB), formado pelos maiores fabricantes de *software* e *hardware* do mundo, tais como SUN *Microsystems*, SGI, IBM, Intel, dentre outros, que, no final de 1997, reuniram esforços para criar um padrão de programação paralela para arquiteturas de memória compartilhada (Costa *et al.*, 2008).

O OpenMP consiste em uma API (*Application Programming Interface*) e um conjunto de diretivas que permite a criação de programas paralelos com compartilhamento de memória através da implementação automática e otimizada de um conjunto de *threads*. O programa inicialmente é executado por uma *thread* e, ao encontrar uma região paralela, é executado por várias *threads* (número definido pelo usuário), voltando a ser executado por uma única *thread* ao término dessa região. Uma nova divisão de *threads* ocorre quando uma nova região paralela é identificada pelo compilador. Esse modelo de programação é conhecido como *fork-join* e é ilustrado na Figura 1.

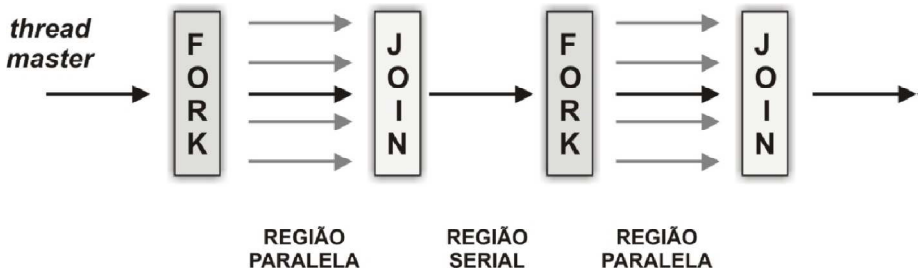


Figura 1: Modelo de programação OpenMP.

Como comentado anteriormente, o OpenMP é baseado em diretivas de compilação. Isso significa que são utilizadas palavras-chave (representadas por `#pragmas`) que são

interpretadas pelo compilador no momento da compilação. Essas palavras são instruções ao compilador e, caso este não tenha suporte a esse nível de paralelização, essas palavras serão ignoradas, fazendo com que o código seja serial.

4.1 Diretivas de compilação

Uma diretiva de compilação consiste em uma linha de código com significado "especial" para o compilador. Na linguagem C/C++ as diretivas do OpenMP são identificadas pelo `#pragma omp` e na linguagem Fortran as diretivas são identificadas pela sentinela `!$omp`.

Essas diretivas são utilizadas antes do trecho de código que se pretende paralelizar. Dessa forma, caso o compilador não interprete essa diretiva, a paralelização não será efetuada e o código será compilado serialmente. A Tabela 1 descreve algumas diretivas do OpenMP e suas respectivas funções.

Diretiva	Função
`#pragma omp parallel`	Cria uma região paralela no código.
`#pragma omp parallel for`	Estratégia de paralelização de domínio, utilizada em *loops*
`#pragma omp sections`	Estratégia de paralelização funcional

Tabela 1: Diretivas de OpenMP

Dependendo da estratégia de paralelização do código, podem ser aplicadas diferentes diretivas. Cada diretiva tem uma função específica, mas elas podem ser utilizadas em conjunto para criar novas estratégias de paralelização.

A diretiva `#pragma omp parallel for`, por exemplo, só é permitida na paralelização da estrutura de repetição `for`. É importante ressaltar que é de responsabilidade do programador o uso correto dessas diretivas. Em algumas situações, o compilador identifica erros de paralelização e não prossegue com a etapa de compilação e, em outras situações, esses erros não são observados.

Por exemplo, ao se criar *threads* dentro de uma região paralela, estas podem ser privadas ou públicas (compartilhadas) dentro do processo. A diferença é que sendo privada, cada *thread* executa com uma cópia dessa variável e assim não interfere nas outras cópias. Caso contrário, quando essa variável é pública, todas as *threads* desse processo acessam o mesmo endereço de memória.

É comum erros de programação paralela associados a variáveis que são compartilhadas. Quando duas ou mais threads tentam acessar e escrever na mesma variável, uma dessas atualizações pode ser perdida, uma vez que elas são executadas simultaneamente. Situações como essa são denominadas condições de corrida (*race conditions*).

4.2 Métricas de desempenho

Quando se trabalha com aplicações em paralelo é importante mensurar o ganho obtido com a implementação paralela do código. Existem algumas métricas que são utilizadas para quantificar esse ganho, entre elas o *speed-up* e a eficiência, sendo este último usado na maioria dos casos.

O *speed-up* é definido como a razão entre o tempo gasto utilizando o código serial e o tempo utilizando o código paralelo com vários processadores (Eq. (1)). Por exemplo, se o *speed-up* de um programa paralelo é 1.5, isso indica que a versão paralela do programa

executa 1.5 vezes mais rápido que a versão serial, ou seja, houve um ganho de 50% em relação a versão serial.

$$S_n = \frac{T_S}{T_P} \tag{1}$$

onde $\boldsymbol{T_S}$ é o tempo serial e $\boldsymbol{T_n}$ é o tempo paralelo com $\boldsymbol{n}$ processadores.

A eficiência é uma indicação da utilização efetiva dos processadores. Ela é definida pela razão entre o *speedup* obtido e o número de processadores (Eq. (2)). Quanto mais próximo de uma unidade estiver seu valor melhor será a eficiência do código paralelo. Contudo, essa métrica não deve ser utilizada isoladamente, pois pode resultar em interpretações erradas, conforme é exemplificado a seguir.

$$e = \frac{S_n}{n} \tag{2}$$

Supondo um código que apresente um custo computacional de 100 segundos. Admitindo que apenas 60% desse código é paralelizável e que os 40% restantes é serial, isso indica que a parcela paralelizável corresponde a 60 segundos e a parcela serial a 40 segundos. Supondo que aplicar as estratégias de paralelização na parcela paralelizável, esta foi reduzida à 36 segundos e, consequentemente, o tempo total de simulação foi de 76 segundos.

Utilizando a Eq. (1), tem-se um *speedup* de 1.3157 e uma eficiência de 0.6578 (65.78%). Analisando esses valores poderíamos concluir que a paralelização não foi eficiente. Porém, em uma análise mais cuidadosa, pode-se concluir que a paralelização foi de fato eficiente. Isso pode ser explicado utilizando a lei de Amdahl, descrita na Eq. (3).

$$S_n^* = \frac{100}{T_S(\%) + T_P(\%)/n} \tag{3}$$

onde $\boldsymbol{T_S}(\%)$ e $\boldsymbol{T_P}(\%)$ representam o percentual da região serial e paralelizável.

A lei de Amdahl permite calcular o máximo *speedup*, denominado *speedup* teórico, que poderia ser obtido considerando a existência de uma parcela do código que é estritamente serial e outra que pode ser paralelizável. Utilizando os resultados do exemplo anterior, tem-se um *speedup* teórico de 1.4285, o que ilustra que a paralelização obtida pode ser considerada eficiente. A eficiência se torna uma medida mais confiável quando a maior parte do código é paralelizável.

5 PERFILAGEM DE CÓDIGO

A identificação dos trechos críticos do programa deve ser a etapa inicial no processo de paralelização. Conhecido esses trechos do código e possuindo um bom entendimento do fluxo do programa, o programador pode identificar mais facilmente oportunidades de paralelização.

Para obter o detalhamento de tempo de execução de um programa, é comum o uso de ferramentas de *profile*. Com o uso dessas ferramentas é possível obter a distribuição de tempo em cada função (método) e também quantificar o número de vezes que elas são chamadas.

O conhecimento dessas regiões críticas é fundamental, caso contrário o programador pode destinar boa parte do tempo em locais desnecessários ou com pouco consumo de tempo e memória. Se por exemplo, forem adotadas estratégias para otimizar um trecho de código que consome 10% do tempo de processamento, os ganhos com a otimização não serão muito significativos. Diferentemente ocorrerá em trechos de código com elevado consumo de tempo, pois qualquer redução causará grande impacto.

Em geral, a maior parte do tempo de processamento se encontra em um pequeno percentual do código e é nesse trecho que o programador deve aplicar as estratégias. Existem atualmente várias ferramentas de *profile*, como exemplo pode-se citar: *Vtune* (Intel, 2009), *CodeAnalyst* (Amd, 2009), *MPIP* (Mpip, 2009), *sysprof* (Sysprof, 2009), *oprofile* (Oprofile, 2009), *Valgrind* (Valgrind, 2009), *Gprof* (Gprof, 2009).

Para obter a distribuição de tempo entre os métodos do *framework* DOOLINES, foi utilizada a ferramenta de *profile* Gprof. Essa ferramenta está integrada ao compilador GCC da GNU e foi desenvolvida por Jay Fenlason. Ela permite a análise da *performance* do algoritmo exibindo os resultados na forma de grafo. Com ela foi possível obter informações da quantidade de métodos existentes no código, número de vezes que a função foi utilizada e o percentual do tempo gasto em cada método.

No DOOLINES, as forças externas atuantes ao longo da linha de ancoragem ou *riser* são provenientes do peso próprio, do efeito da corrente marítima, da reação do solo, entre outras forças. Aplicando o Gprof no DOOLINES, foi possível identificar que o trecho de código referente ao cálculo do efeito da corrente marítima sobre a malha de elementos finitos é o que consome maior de tempo de processamento, sendo responsável por aproximadamente 42% do tempo total de uma simulação convencional. Esse método está inserido em um método superior que calcula as forças externas e internas que atuam na malha, o qual representa aproximadamente 95% do tempo total de processamento. Com isso, os esforços para paralelização do código foram concentrados nesse trecho e foi possível identificar que cerca de 80% do tempo total é paralelizável.

6 ESTRATÉGIAS UTILIZADAS

Existem dois modelos clássicos de paralelismo, um deles denominado paralelismo de dados e o outro paralelismo funcional. O paralelismo de dados consiste na divisão de um conjunto de dados que operam com as mesmas instruções. No paralelismo funcional, mantêm-se o conjunto de dados e instruções diferentes é que são realizadas em paralelo. No contexto do DOOLINES, o paralelismo funcional consiste em calcular as diferentes forças de forma simultânea e o paralelismo de dados consiste em calcular essas forças de forma seqüencial, mas os dados são divididos entre as centrais de processamento na execução de cada função. Neste trabalho, é utilizado apenas o modelo de paralelismo de dados. No cálculo de cada força atuante na linha, a malha de elementos finitos é dividida entre o número de *threads* disponíveis ou determinada pelo usuário.

No caso do paralelismo de dados, é necessário uma atenção especial, pois a divisão sem um critério seguro pode ocasionar em informações que não são atualizadas, gerando condições de corrida (*race conditions*) e consequentemente erros no programa. Para entender como esses erros podem acontecer, é necessário entender a sequência de atualizações que são feitas em cada um dos métodos.

No DOOLINES, as forças externas e internas são representadas por vetores unidimensionais com dimensão múltipla do número de nós da malha. Na montagem desses vetores é feito um *loop* em cada elemento da malha e para cada elemento são calculadas as forças nodais equivalentes associadas a cada tipo de força. Essas forças nodais são atualizadas no vetor de forças global que contém as forças associadas a todos os nós da malha.

Observando a Figura 2, pode-se observar que a posição do vetor de forças correspondente a um nó adjacente a dois elementos, receberá contribuição desses elementos vizinhos. Isso indica que essa posição do vetor de forças será definida pela soma das forças provenientes de cada um desses elementos.

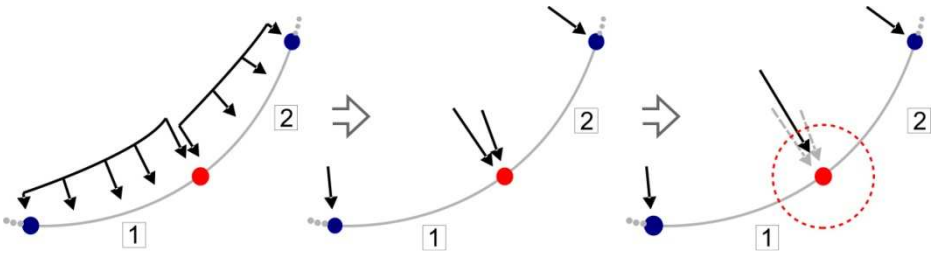


Figura 2: Montagem do vetor de forças nodal e global (Costa *et al.*, 2008).

Quando se executa o DOOLINES de forma serial, cada elemento é percorrido sequencialmente e o vetor de forças externas é montado na mesma sequência. Contudo, quando se utiliza o OpenMP para paralelização do código, cada *thread* criada será responsável por um conjunto de dados e nesse caso, existe a possibilidade de duas *threads* estarem executando elementos vizinhos e tentarem atualizar a mesma posição do vetor de forças associado ao nó em comum. Se isso ocorrer, alguma das atualizações pode ser perdida e o programa poderá ter o resultado alterado. Independente da estratégia elaborada para paralelização dessa região deve-se assegurar que condições de corrida não vão ocorrer.

Para avaliar diferentes formas de paralelização, foram implementadas no DOOLINES quatro estratégias de paralelização, dentre as quais uma delas foi à mesma utilizada no trabalho de Costa *et al*. (2008). Para avaliação dos ganhos obtidos com a implementação paralela foi simulado um modelo de linha inicialmente em catenária e sujeita a influência da corrente marítima. A linha modelada possuía 1750m de comprimento e foi discretizada com 1500 elementos de comprimento constante, conforme ilustra a Figura 3.

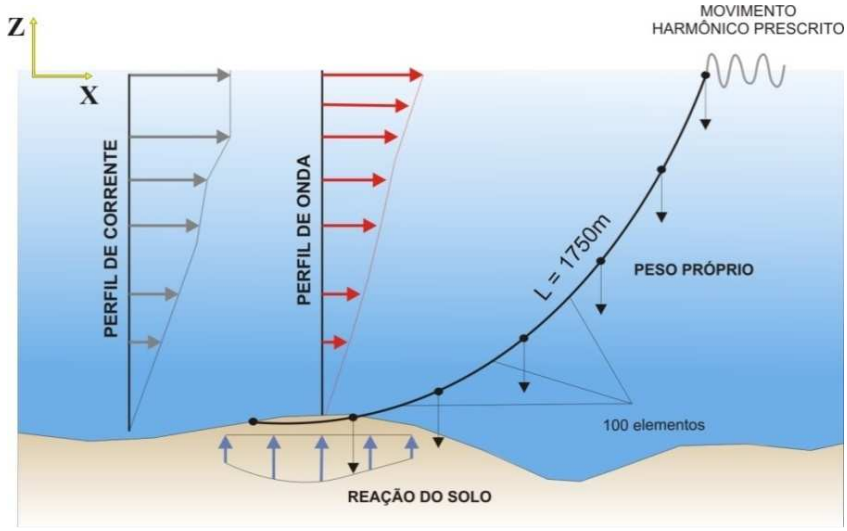


Figura 3: Exemplo do modelo simulado.

- **Estratégia 1 – Paralelização direta com uma única diretiva**

Essa estratégia foi abordada no trabalho Costa *et al*. (2008) e consistiu em aplicar diretamente, sobre os trechos de código com maior consumo de tempo, a diretiva do OpenMP `#pragma omp parallel for`. Essa estratégia pode ser considerada simplificada uma vez que, apenas com o uso da diretiva é possível paralelizar o código. Um pseudo-código do DOOLINES com a diretiva essa diretiva de paralelização é ilustrado na Figura 4.

```
//Calculo da Força devido a Corrente Marítima
#pragma omp parallel for
for (int i=1;i<NumElements;i++)
 {...}
```

Figura 4 - Exemplificação do uso de uma diretiva de OpenMP

Por *default*, se nenhum argumento é utilizado nessa diretiva, o compilador admite que a divisão das iterações deva ser feita de forma estática. Isso significa que cada *thread* receberá um bloco de elementos correspondente a `NumElements/NumThreads`. Ou seja, se a malha tiver 1000 elementos e forem utilizados duas *threads*, cada *thread* receberá 500 elementos. Contudo, o que assegura a não ocorrência de *race-conditions* é a divisão de blocos estáticos seqüenciais (quando nenhum argumento é indicado). Isso significa que, obrigatoriamente, a *thread* 1 receberá os elementos com identificação de 1 a 500 e a *thread* 2, receberá os elementos com identificação de 501 a 1000. Então, como cada *thread* inicia sua execução de forma seqüencial, enquanto a *thread* 1 estiver executando e computando as forças do elemento 1, a *thread* 2 estará executando o elemento 501. Como o único ponto de interface é se encontra entre os elementos 500 e 501 a probabilidade de acontecer alguma condição de corrida é mínima em simulações convencionais.

Maiores detalhes dessa implementação são descritos em Costa *et al*. (2008). Porém, é importante ressaltar que a simplicidade dessa estratégia está associada à dimensão do domínio do problema. Mesmo considerando o cenário *offshore* como tridimensional, o domínio do problema é unidimensional e isso reduz o número de interfaces sujeita a erros devido à condição de corrida. Mas, mesmo o mesmo problema sendo unidimensional, caso sejam utilizadas um elevado número de *threads*, seja considerada uma simulação com várias linhas se conectando ou o esquema de divisão das *threads* seja alterado, poderá ocorrer condições de corrida e assim comprometer os resultados da análise. A Figura 5 ilustra a divisão da malha para três *threads* e destaca os nós de interface entre as *threads*.

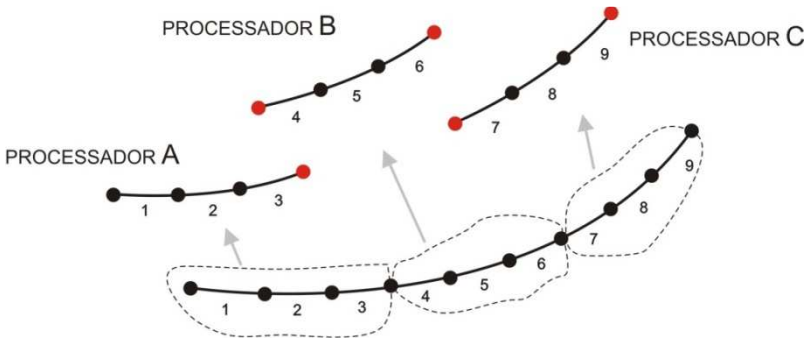


Figura 5: Divisão da malha considerando três *threads* e a estratégia de divisão direta do *loop*.

- **Estratégia 2 - Divisão dos loops nos blocos de iterações pares e ímpares**

Nessa estratégia, cada um dos *loops* que calcula as forças internas e externas do DOOLINES foi dividido em dois *loops*, o primeiro contendo apenas os elementos com identificação ímpar e o segundo com os elementos com identificação par (Figura 6).

Essa estratégia pode ser considerada menos eficiente uma vez que, requer a criação e destruição de *threads* com uma maior freqüência. Contudo, pode ser mais considerada mais segura por evitar perdas de informações por atualizações não realizadas nas interfaces entre

elementos computados por *threads* diferentes. Essa estratégia evita algumas condições de corrida, mas também não pode ser extrapolada para casos 2D e 3D, onde o número e a ocorrência de condições de corrida é maior e mais difícil de ser gerenciado.

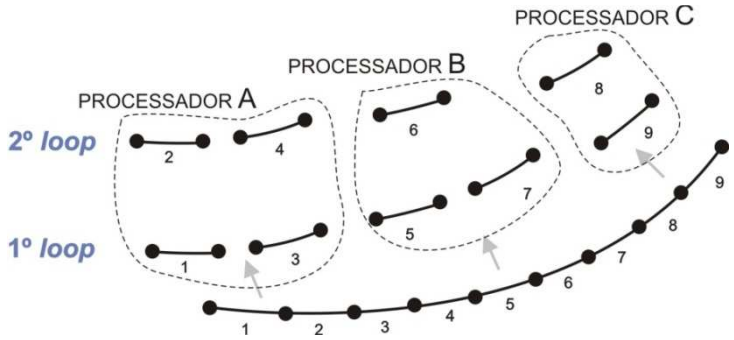


Figura 6: Divisão da malha considerando três *threads* e a estratégia de divisão do *loop* em elementos pares e ímpares.

- **Estratégia 3 - Criação de vetores auxiliares para o cálculo das forças**

A paralelização através dessa estratégia pode ser considerada genérica, podendo ser aplicada em qualquer tipo de problema e independente também da dimensão considerada para o domínio. Para exemplificar isso, pode-se citar o trabalho de Sena *et al* (2008), onde é descrito o uso dessa estratégia na análise de meios contínuos com base no Método dos Elementos Discretos.

A idéia consiste em criar *n* vetores de forças auxiliares, onde *n* corresponde exatamente ao número de *threads* criadas na região paralela. Cada vetor auxiliar deve ter dimensão igual ao vetor de forças global e deve estar associado a uma *thread*. Dessa forma, durante a execução da região paralela, cada *thread* deverá escrever em um dos vetores auxiliares, eliminando qualquer possibilidade de condição de corrida, como é ilustrado na Figura 7. Ao final do processo em paralelo, um novo trecho de código deve ser inserido para reunir as informações dos vetores auxiliares no vetor de forças global do sistema e dessa forma, continuar a execução do programa.

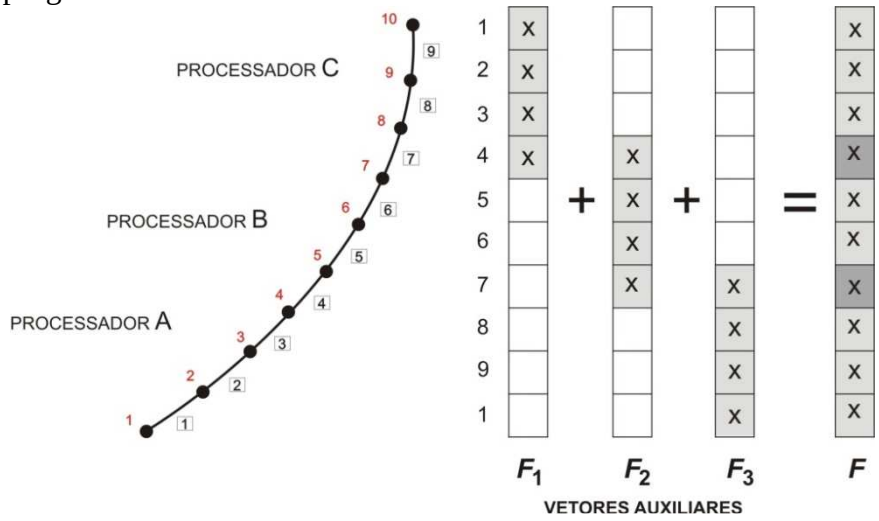


Figura 7: Divisão da malha considerando três *threads* e a estratégia de divisão de criação de vetores auxiliares.

Apesar de genérica, essa estratégia é mais complexa do ponto de vista computacional. Além de ser necessário implementar a rotina para reunir as informações, o programa terá um consumo maior de memória em função da criação dos vetores auxiliares, implicando em um gerenciamento manual dos dados. Outro aspecto é o tempo adicional que surge pela necessidade desse gerenciamento manual, isso pode comprometer os ganhos com a paralelização quando forem simulados problemas pouco discretizados ou muito simplificados.

- **Estratégia 4 - Uso do padrão MPI em ambientes de memória compartilhada**

Nessa estratégia foi utilizado o padrão MPI e avaliado o seu desempenho em ambientes de memória compartilhada. O padrão MPI é utilizado em ambientes de memória distribuída, a exemplo de simulações paralelas em *clusters*. Devido sua portabilidade e hierarquia de memória implementada, quando executado em ambientes de memória compartilhada, ele é capaz de compartilhar os dados ao invés de enviá-los como mensagens, como é feito em ambientes de memória distribuída. Esse trabalho não apresentou maiores detalhes do padrão MPI porque a estratégia implementada relativa a esse padrão teve apenas o objetivo de demonstrar sua portabilidade e os ganhos que podem ser obtidos.

A versão do DOOLINES com MPI procurou interferir o mínimo possível no código. Dessa forma, foi possível criar uma versão que pode ser utilizada de forma serial ou em paralelo com MPI ou OpenMP. A opção por essa metodologia resultou em uma versão com MPI pouco escalável, conforme será visto nos resultados. Para criar essa versão, foi implementada a classe `mpisolver`, que é responsável pela criação dos processos a serem utilizados, divisão das tarefas e troca de informações entre os processos. A paralelização com o uso do MPI foi aplicada tanto aos métodos que calculavam as forças internas e externas quanto ao método que calculam a aceleração, velocidade e deslocamento.

7 RESULTADOS

Nessa seção são apresentados os resultados obtidos com as versões implementadas (serial e paralela). Além dos gráficos de tempo de processamento, *speedup* e eficiência, também são apresentados gráficos dos erros envolvidos nessas simulações, assunto pouco abordado quando se trata de simulações em paralelo.

7.1 Erros Numéricos

A simples aplicação de estratégias de paralelização resultou em pequenos erros numéricos associados às operações de ponto flutuante. Essas perturbações são aparentemente desprezíveis e em problemas estáticos não são relevantes. Contudo, quando se trata de problemas iterativos, uma pequena perturbação pode se propagar ao longo das iterações temporais e gerar resultados distintos entre diferentes implementações. Uma estratégia para conter essa propagação é o uso de equações de equilíbrio (conservação) ou equações de prescrição ao longo da análise, evitando o acúmulo excessivo de erros.

O primeiro exemplo apresenta a coordenada z de um nó que se encontra próximo a superfície livre da água ao longo da simulação. Nesse exemplo, são consideradas como forças externas atuantes na linha, o efeito da corrente marítima, a reação do solo, o peso próprio, empuxo e uma força harmônica prescrita no topo, simulando o movimento da plataforma (Figura 3). Como pode ser observado no gráfico apresentado na Figura 8, os resultados para o deslocamento diferem em vários instantes de tempo e em todas as estratégias implementadas. Contudo, os erros observados relativo à estratégia serial são pequenos comparados a escala do problema.

Nesse exemplo, o uso da prescrição harmônica funciona como uma contenção à propagação do erro, pois independente da implementação realizada, a trajetória do nó que recebe a prescrição é definida pela função harmônica e é obrigatoriamente a mesma em todas as estratégias propostas. Como conseqüência, ela funciona como uma equação de correção em cada intervalo do tempo.

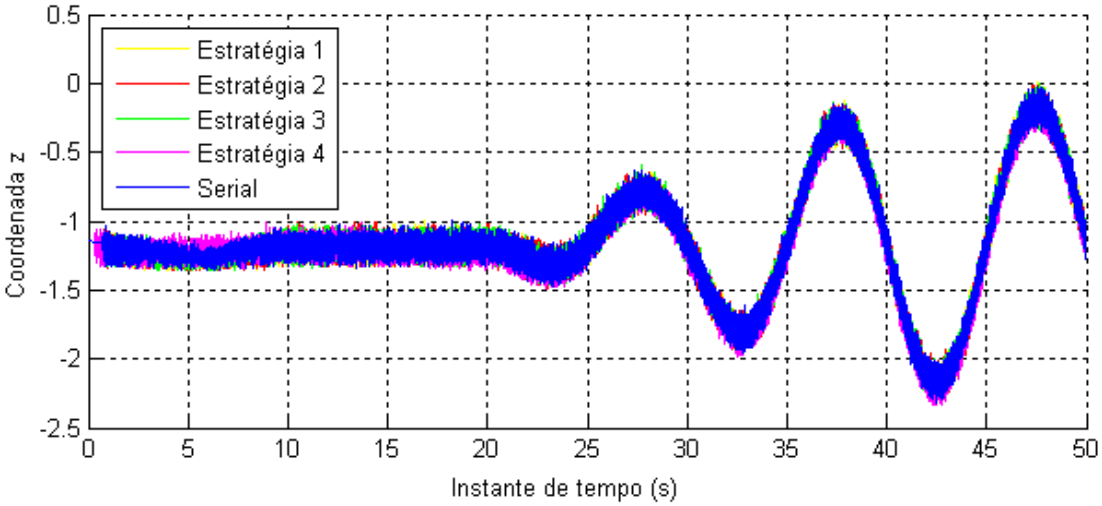


Figura 8: Coordenada *z* do nó 1498 (próximo a superfície livre da água) ao longo da simulação considerando a prescrição harmônica.

O segundo exemplo considera o mesmo modelo utilizado no exemplo anterior e desconsidera a prescrição harmônica no topo da linha. Nenhuma força é prescrita e dessa forma, a maior parte da linha estará sob componentes de forças verticais devido apenas ao peso próprio e empuxo. Os resultados apresentados se referem a coordena *z* do mesmo nó analisado do exemplo anterior. O gráfico apresentado na Figura 9 ilustra a trajetória da coordenada *z* desse nó na versão serial e paralela (estratégia 1) do DOOLINES. Nesse exemplo foram omitidos os resultados das outras estratégias por apresentarem o mesmo comportamento.

Diferentemente dos erros apresentados no exemplo anterior, para a linha em queda livre, a magnitude dos erros foram mais expressivas. A ausência da prescrição harmônica resultou em um conjunto de forças atuando na linha com características monotônicas e livres, ou seja, sempre atuam na mesma direção e não são prescritas. Como conseqüência, o problema é simulado sem nenhuma equação que possa conter o erro e sua propagação não pode ser evitada.

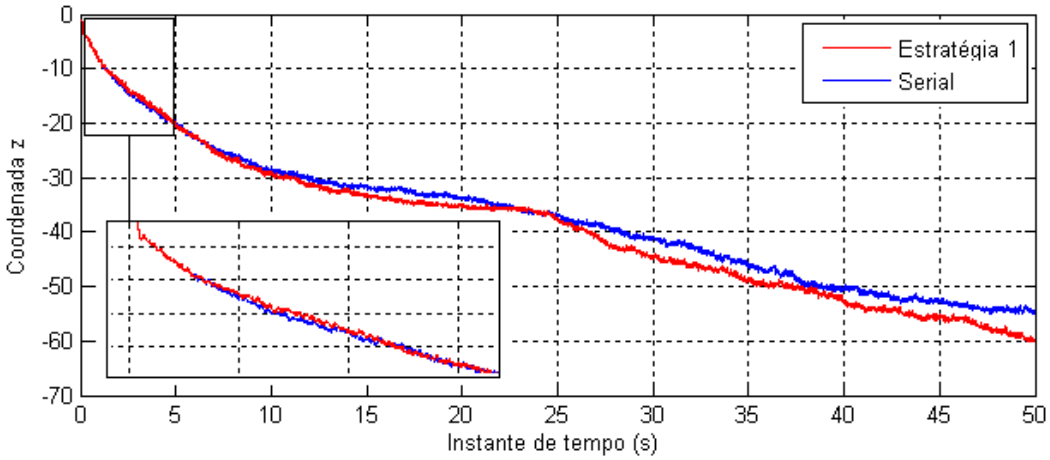


Figura 9: Coordenada *z* d nó 1498 (próximo a superfície livre da água) ao longo da simulação.

Mesmo com resultados distintos, pode-se observar que o problema não é a não convergência de alguma das estratégias, mas sim pequenos erros numéricos que ao longo das iterações passam a apresentar influência cada vez maior.

O comportamento do erro numérico pode ser melhor entendido analisando o gráfico apresentado na Figura 10, em que são plotados os erros ao longo do tempo entre a resposta paralela (considerando a estratégia 1) e a versão serial do programa. Essa figura ilustra o histórico completo do erro ao longo de toda simulação e pode-se observar que o erro em algumas instantes foi superior a 5 metros.

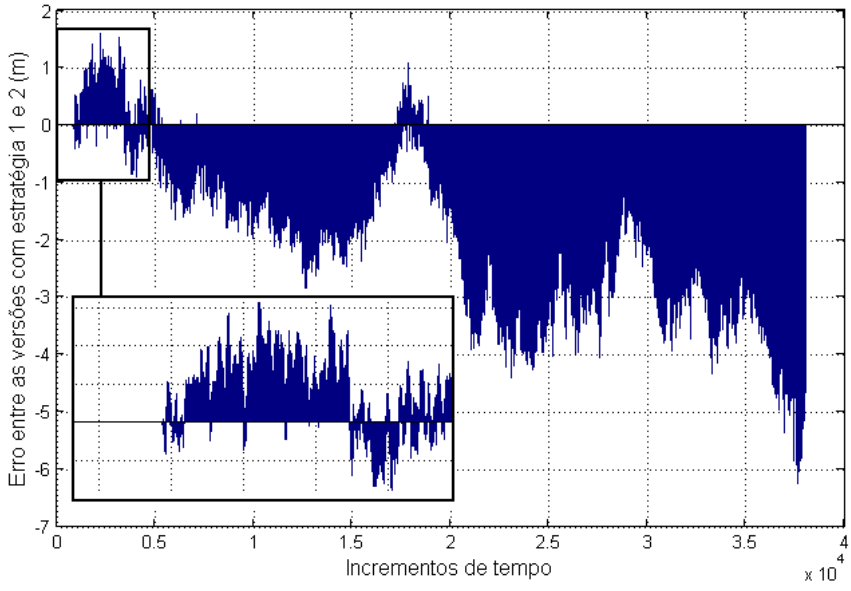


Figura 10 – Erro entre a resposta serial e paralela (estratégias 1) ao longo de toda simulação.

Para melhor entendimento do surgimento do erro e visualizar a existência do erro numérico, pode-se analisar o gráfico ilustrado na Figura 11. Nesse gráfico são plotados os erros nos 1200 incrementos de tempo iniciais. Pode-se observar pelo gráfico que até os 800 intervalos de tempo iniciais não existe erro entre a solução serial e a paralela. A ausência do

erro não necessariamente significa coincidência total entre as respostas, mas sim um erro residual pequeno, impossível de ser representado numericamente.

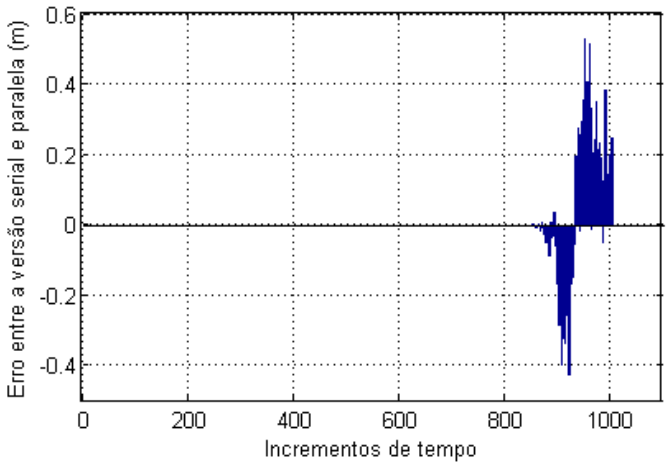


Figura 11: Erro entre a resposta serial e paralela (estratégias 1).

Essas pequenas perturbações se propagam, lentamente ou rapidamente, em função dos parâmetros e características do modelo. Elas evoluem na direção dos números significativos do modelo e quando uma dessas perturbações atinge a escala de representação numérica, acabam por inserir perturbações no modelo. Considerando o modelo altamente não-linear, do ponto de vista geométrico, uma pequena perturbação resulta na alteração da configuração de equilíbrio do modelo e o comportamento global do modelo é comprometido.

7.2 Tempo de Processamento

Nessa seção são descritos os ganhos, em termos de tempo computacional, que foram obtidos com a implementação das estratégias paralelas no DOOLINES. Os testes foram realizados em diferentes computadores visando investigar o comportamento em cada uma deles. As características dessas máquinas são descritas na Tabela 2.

Computador A	Core2Quad, 8Mb *cache* L2, 2.8GHz, 24 Gb de RAM e Sistema Operacional Linux 64 bits – Centos 5.4.
Computador B	Core2Duo, 2 Mb *cache* L2, 1.86 GHz, 1Gb de RAM, e Sistema Operacional Linux 32 bits – Ubuntu 9.04.

Tabela 2: Descrição das máquinas utilizadas.

Os primeiros resultados apresentados foram obtidos no Computador A. Como esse computador possui dois processadores Core2Quad, é possível executar as regiões paralelas com oito *threads* independentes e simultâneas. Dessa forma, foram analisados a redução do tempo computacional e a evolução do *speedup* com o aumento do número de *threads*.

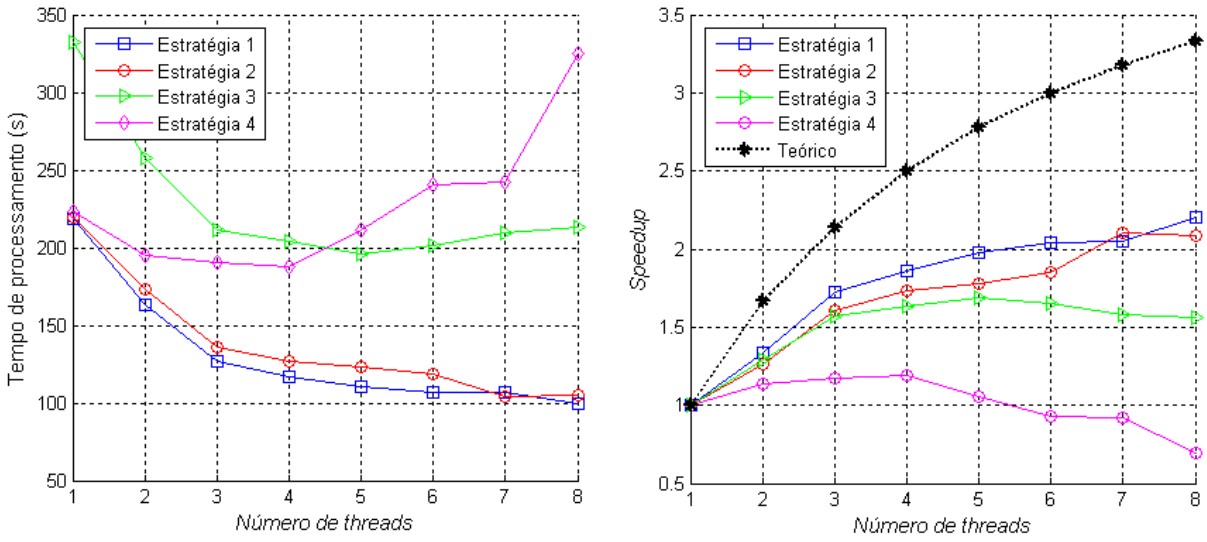
Na Figura 12-a é apresentado a variação do tempo computacional com o aumento do número de *threads*. As estratégias 1, 2 apresentaram resultados esperados, uma vez que, com o aumento do número de *threads* o tempo da região paralela deve diminuir. Contudo, os resultados obtidos para a estratégia 3 e 4, apresentaram resultados similares quando foram

utilizados até quatro e cinco *threads*, respectivamente. Quando o número de *threads* excedeu esses valores, o tempo computacional aumentou.

Dois fatores podem ter contribuído para esse comportamento. O primeiro deles se refere à estratégia utilizada na versão MPI. Uma vez concebida para ambientes de memória distribuída, sua escalabilidade pode não ser assegurada em ambientes de memória compartilhada. O segundo fator tem como base a arquitetura da máquina utilizada. O Computador A é constituído por dois processadores Core2Quad e isso indica que apesar de ser possível criar oito *threads* independentes, quatro *threads* (1,2,3,4) estão associadas aos *cores* de um processador e as outras quatro *threads* (5,6,7,8) associadas aos *cores* do outro processador. Por isso, quando são utilizados cinco *threads*, por exemplo, quatro delas estão associadas a um processador e a outra *thread* estará no outro processador, implicando em um gerenciamento mais complexo. Essa situação acontece em todas as estratégias, mas nas estratégias 3 e 4 é mais evidente.

Como resultado, o tempo de sincronização e troca de informações assume um valor maior e implica em um tempo computacional mais elevado. Isso pode ser observado não apenas com o aumento de *threads* para estratégia 4, mas também na estratégia 3, quando se utilizou 6,7,8 *threads*. O gerenciamento mais complexo e uma carga de trabalho reduzida nas estratégias 3 e 4, podem ter implicado nesse efeito inverso de aumento do tempo computacional.

Pois nesses casos, o tempo adicionado devido a estratégia mais complexa, teve um grande impacto. Contudo, assim como descrito em Costa *et al*. (2008), o *speedup* também é função da carga de trabalho. Isso significa que ao aumentar o tamanho de blocos de iterações por *threads*, existe uma tendência de aumento do *speedup* até um limite. Dessa forma é possível que, aumentando a discretização da malha, os resultados para as estratégias 3 e 4 apresentem resultados melhores.



a) b)

Figura 12: a) Tempo de processamento versus número de *threads*.
b) *Speedup* versus número de *threads*

No gráfico da Figura 12-b são apresentados os valores de *speedup* obtidos com o aumento do número de *threads*. Nesse gráfico, além das curvas de *speedup* para cada estratégia, também é plotada a curva do *speedup* teórico, calculada com base na Eq. (3). Pode-se observar que todas as curvas de *speedup* estão abaixo da curva do *speedup* teórico. Contudo,

os resultados obtidos foram satisfatórios. É importante ressaltar também que os valores de *speedup* calculados, foram feitos assumindo como referência para cada estratégia, a versão modificada dessa estratégia sendo executada com apenas uma *thread*. Caso fosse utilizado o tempo serial da versão serial inalterada, os valores relativos às estratégias 3 e 4 seriam próximas de 1, indicando pequenos ou, em alguns casos, nenhum ganho relativo a versão inalterada.

Isso nos permite concluir que, em muitas vezes, o esforço para tornar um código paralelizável pode elevar seu tempo de processamento quando apenas uma central de processamento for utilizada. Pois nesses casos, o código necessita ser alterado e a estratégia de paralelização implementada pode significar mais consumo de processamento. Esse efeito na maioria das vezes desaparece quando a simulação envolve uma elevada carga de processamento (malhas tridimensionais muito discretizadas), pois o tempo adicional devido às modificações no código apresentará um menor impacto no tempo total da simulação, que será dominado pelos tempos consumidos por cada *thread* no processamento de seus blocos.

Para a simulação paralela considerando duas *threads*, o *speedup* teórico calculado pela Eq. (3) é 1.6667 e os valores de *speedup* obtidos para as estratégias 1,2,3 e 4 foram 1.34, 1.26, 1.29 e 1.14, conforme o gráfico da Figura 12-b.

Os valores de *speedup* observados são descritos na Tabela 3 e observa-se que se considerarmos um número de *threads* sempre menor ou igual a quatro, as versões paralelas apresentarão ganhos comparada com a serial com a mesma estratégia. Acima desse valor, algumas estratégias apresentaram uma queda no *speedup* e a razão para isso é a mesma apresentada para explicar o aumento do tempo de processamento em algumas estratégias.

Estratégias	Número de *Threads*						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Teórico	**1.6667**	**2.1428**	**2.5000**	**2.7777**	**3.0000**	**3.1818**	**3.3333**
1	1.34	1.72	1.86	1.98	2.04	2.05	2.20
2	1.26	1.61	1.73	1.78	1.85	2.10	2.09
3	1.29	1.57	1.63	1.69	1.65	1.58	1.56
4	1.14	1.17	1.19	1.06	0.93	0.92	0.69

Tabela 3: *Speedups* das estratégias variando o número de *threads*.

É importante ressaltar os resultados obtidos com as estratégias 1 e 3. Na estratégia 1, utilizando duas *threads*, o *speedup* foi de 1.34, indicando que a velocidade de processamento foi 1.34 vezes mais rápida que a serial inalterada. Para o caso de três *threads*, essa a velocidade foi 1.72 vezes mais rápida que a mesma versão serial anterior. Esses resultados foram obtidos para o Computador 1, cujo sistema operacional opera a 64 bits. Esses resultados estão de acordo com os resultados obtido por Costa *et al*. (2008) para uma máquina com configuração semelhante.

No trabalho de Costa *et al*. (2008) foram identificados algumas variáveis que apresentam grande influência no processamento paralelo. Entre eles, destacaram-se o sistema operacional, o compilador e o tamanho do *cache* L2. Com o mesmo objetivo, foram realizadas algumas simulações para investigar a influência dessas variáveis em cada uma das estratégias implementadas. O gráfico de barras apresentado na Figura 13 ilustra os valores de *speedup* obtidos para as simulações com as quatro estratégias de paralelização no Computador 2. Esse computador possui Sistema Operacional Linux a 32 bits, diferentemente do Computador 1, que possui o sistema Linux a 64 bits e possui uma arquitetura mais moderna.

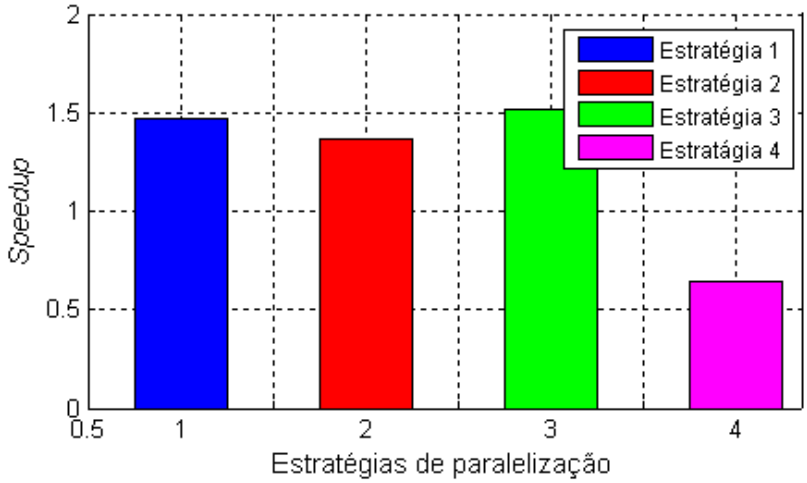


Figura 13: Erro entre a resposta serial e paralela (estratégias 1).

Dessa forma, os resultados para o tempo de processamento no Computador 2 foram todos superiores aos tempos no Computador 1. Por exemplo, a versão serial no Computador 1 demandava cerca de 218 seg. de processamento e a essa mesma versão no Computador 2 demandava cerca de 614 seg. Contudo, os ganhos obtidos, avaliados pelo *speedup*, foram superiores aos ganhos obtidos com as simulações feitas no Computador 1.

Testes realizados com computadores que apresentavam tamanhos da memória *cache* diferentes, também apresentaram tempos de processamento diferentes, onde o menor tempo foi obtido no computador com tamanho da memória *cache* maior.

8 CONCLUSÃO

Esse trabalho representa o resultado do contínuo estudo sobre técnicas computacionais de alto desempenho aplicadas à simulação de linhas de ancoragem e *risers*. Ele representa uma continuação do trabalho desenvolvido por Costa *et. al.* (2008) e apresenta como principal diferencial, as discussões acerca dos erros numéricos envolvidos em simulações paralelas e as diferentes estratégias de paralelização que podem ser implementadas.

Na seção que discute os erros numéricos envolvidos em simulações paralelas, foi analisado o mesmo modelo submetido a condições de força diferenciadas. Os resultados indicam a provável propagação de erro que pode acontecer em problemas iterativos que não possuem uma condição de contenção dessa propagação. No exemplo 2 (simulando a queda livre da linha), onde ilustrou a propagação do erro, é necessário ressaltar que este não se trata de um exemplo de aplicação real desse simulador, sendo o modelo construído apenas para fins didáticos.

Os resultados obtidos e discutidos para o tempo de processamento foram obtidos com tempos de simulação da ordem de 600 a 1000 segundos. Esses números ainda são muito pequenos comparados as simulações que são feitas nessas áreas, que podem demandar horas de processamento. Contudo, os ganhos obtidos para simulações com esses tempos serão os mesmos para simulações que apenas tenham mais incrementos de tempo. Isso ocorre porque o ganho do processamento paralelo sofre influência do tamanho dos blocos que cada *thread* irá executar e não das iterações no tempo que são feitas, pois a cada intervalo de integração os ganhos são os mesmos. Outras variáveis que também apresentam influência e que já haviam sido registrado no trabalho de Costa et al. (2008) é o tamanho da memória *cache* e como o sistema operacional é executado (32 ou 64 bits).

Admitindo uma simulação com tempo total de 30 min. (1800 seg.), os resultados mostram que é possível reduzir esse tempo para aproximadamente 22.38 min. e 16.12 min. utilizando a estratégia 1 em um computador com dois *cores*, a exemplo de um Core2Duo.

Os ganhos em termos de *speedup* da estratégia 3 também foram significativos. Contudo, esses ganhos só foram obtidos quando foi considerada como referência a versão da estratégia 3 sendo executada com apenas uma *thread*. Ou seja, em relação à versão serial inalterada, para malhas com poucos elementos, não serão observados ganhos, pois o tempo adicional necessário devido às modificações no código acaba sendo dominante. Esse fato provavelmente não irá acontecer se cada *thread* receber grandes blocos de iterações.

O estudo e implementação dessas estratégias de paralelização permitiu concluir que as estratégias 1 e 2 são aquelas que apresentam maiores ganhos, apesar de simplificadas e limitadas a alguns tipos de malhas. Estratégias mais complexas só devem ser utilizadas em problemas mais complexos (tridimensionais) e onde a carga de trabalho seja elevada o suficiente para permitir que o tempo adicional devido as alterações no código, não apresentem grande influência. Destaca-se a generalidade da estratégia 3 como sua principal vantagem, pois pode ser aplicada a qualquer tipo de problema usando a mesma metodologia. Porém, recomenda-se seu uso para problemas mais complexos ou com elevada carga de trabalho, como o problema analisado por Sena *et al*. (2008).

Em relação à estratégia 4, usando o padrão MPI, as mesmas considerações relativas a estratégia 3 podem ser feitas. Seu uso só deve ser feito quando o demanda computacional for extremamente elevada. Caso contrário, sugerem-se estratégias mais simples de paralelização. Quanto ao uso do MPI em memória compartilhada, destaca-se sua portabilidade. Pois a mesma versão voltada para *cluster* foi executada sem nenhuma alteração em máquinas de memória compartilhada.

Por fim, concluímos a importância de um estudo detalhado do problema que se deseja paralelizar e principalmente, das estratégias disponíveis na literatura. Esperamos com esse trabalho, contribuir com nossa experiência e com as discussões apresentadas ao longo desse trabalho.

Agradecimentos

Os autores agradecem ao suporte técnico e financeiro do CENPES/Petrobras através da participação nos projetos de pesquisa da Rede Galileu e ao suporte do LCCV com o local para desenvolvimento da pesquisa.

REFERÊNCIAS

Costa, J. & Sena, M., 2008. Tutorial OpenMP C/C++. Laboratório de Computação Científica e Visualização, Maceió, AL, 1 edition.

Silveira, E. S., 2001. Análise dinâmica de linhas de ancoragem com adaptação no tempo e subciclagem (Tese de Doutorado). (P. U. Janeiro, Ed.) Rio de Janeiro, RJ, Brasil.

Ferreira, F. M. G., 2005. Desenvolvimento e aplicações de um framework orientado a objetos para análise dinâmica de linhas de ancoragem e de risers. Mestrado em engenharia civil, Universidade Federal de Alagoas, Maceió, AL.

Coelho, L. C., Nishimoto, K., & Masetti, I. Q., 2001. *Dynamic simulation of anchoring systems using computer graphics*. *Omae Conference*. Rio de Janeiro, Brasil.

Nishimoto, K.; Ferreira, M. D.; Masettin, I. Q.; Silveira, E. S. S.; Menezes, I. F. M.; Russo, A. A.; Fucatu, C. H.; Tanuri, E. A. *Development of numerical offshore tank for ultra deep water oil production systems with multibodies*. In: CILAMCE'2004 (XXV Iberian Latin-

American Congress On Computational Methods In Engineering). Recife, PE, Brazil: UFPE/ABMEC, 2004.

Sena, M. C. R.; Costa, J. A. C.; Silveira, E. S. S.. Implementação de uma técnica de processamento paralelo para o método dos elementos discretos utilizando o padrão de memória compartilhada openmp em ambientes multicore. In: CILAMCE'2008 (XXIX Iberian Latin-American Congress On Computational Methods In Engineering). Maceió, AL, Brasil. 2008.

Costa, J. A. C.; Ferreira, F. M. G.; Sena, M. C. R.; Silveira, E. S. S.. Implementação da técnica de processamento paralelo com o uso do padrão openmp em um *framework* que realiza análises dinâmicas de linhas de ancoragem e *risers*. In: CILAMCE'2008 (XXIX Iberian Latin-American Congress On Computational Methods In Engineering). Maceió, AL, Brasil. 2008.

Silveira, E. S. S., Lages, E. N., & Ferreira, F. M. G., 2009. *DOOLINES: An Object-Oriented Framework for Dynamic Analysis of Offshore Lines*. Advances in Engineering Software (a ser publicado).

Pvm, 2009. Disponível em http://www.csm.ornl.gov/pvm. Acessado em Agosto de 2009.

Mpi, 2009. Disponível em http://www.mcs.anl.gov/research/projects/mpi. Acessado em Agosto de 2009.

OpenMP, 2009. Disponível em http://openmp.org. Acessado em Agosto de 2009.

MPIP, 2009. Disponível em http://www.mpip.org. Acessado em Agosto de 2009.

Sysprof, 2009. Disponível em http://live.gnome.org/Sysprof. Acessado em Agosto de 2009.

Gprof, 2009. Disponével em http://www.gnu.org. Acessado em Agosto de 2009.

Oprofile, 2009. Disponível em http://oprofile.sourceforge.net/news. Acessado em Agosto de 2009.

Valgrind, 2009. Disponível em http://valgrind.org. Acessado em Agosto de 2009.

Intel, 2009. Disponível em http://www.intel.com. Acessado em Agosto de 2009.

Amd, 2009. Disponível em http://www.amd.com. Acessado em Agosto de 2009.