Otimização para Ambientes Intel® de um Método Numérico para o Escoamento Bifásico de Fluidos em Meios Porosos Através da Eliminação de Barreiras OpenMP

Weber Ribeiro[1], **Thiago Teixeira**[1] , **Frederico L. Cabral**[1], **Márcio R. Borges** [1], **Carla Osthoff**[1]

[1]Laboratorio Nacional de Computacao Cientifica (LNCC)
Av. Getúlio Vargas, 333. Quitandinha – 25651-075 – Petrópolis – RJ – Brasil

webergdr@gmail.com

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

Abstract. *This paper presents optimizations of a numerical method for biphasic fluid flow in porous media, aimed at parallel execution in Intel® environments. Intel® Parallel Studios XE suite tools have been used to study possible implementations. The EWS-SYNC implementation consists of replacing OpenMP barriers with an explicit thread synchronization mechanism, MPI is implemented for communication between multiple distributed processors and making code usable in a cluster environment. The results for increasing the number of processes in the new MPI code were compared with increasing the number of threads in the EWS-SYNC code. The EWS-SYNC implementation achieved 27x Speedup compared to serial execution using Intel® Xeon Phi (KNL) @ 1.40GHz hardware with 68 physical cores 4 threads/core on one machine containing Intel Xeon E5-2698 v3 @ 2.30GHz CPU with 32 physical cores in [Teixeira et al. 2018]. Comparing the EWS-SYNC code speedup to the Intel Xeon® architecture serial code CPU E5-2698 v3 @ 2.30GHz 16 physical cores the Speedup was 10x and in this same architecture the still in the early phase MPI code implementation compared to EWS-SYNC achieved 23x Speedup.*

Resumo. *Este artigo apresenta otimizações de um método numérico para o escoamento bifásico de fluidos em meios porosos, voltado à execução paralela em ambientes Intel®. As ferramentas do suíte Intel® Parallel Studios XE, foram utilizadas no estudo de possíveis implementações. A implementação EWS-SYNC consiste em substituir as barreiras do OpenMP por um mecanismo explícito de sincronismo entre threads, o MPI é implementado para comunicação entre diversos processadores distribuídos e tornar o código utilizável em ambiente Cluster. Foram comparados os resultados para o aumento de número de processos no novo código MPI com o aumento do número de threads no código EWS-SYNC. A implementação EWS-SYNC obteve Speedup de 27x, comparado-se a execução serial, utilizando-se o hardware Intel® Xeon Phi (KNL) @ 1.40GHz com 68 cores físicos 4 threads/core em uma máquina que contém Intel Xeon CPU E5-2698 v3 @ 2.30GHz com 32 cores físicos em [Teixeira et al. 2018]. Comparando-se o Speedup do código EWS-SYNC em relação ao código serial em arquitetura Intel Xeon® CPU E5-2698 v3 @ 2.30GHz 16 cores físicos o Speedup foi de 10x e nesta mesma arquitetura o ainda em fase inicial de implementação código MPI em relação ao EWS-SYNC obteve Speedup de 23x.*

1. Introdução

Dentro de diversas áreas da engenharia e ciências aplicadas, existe um grande interesse no desenvolvimento de modelos matemáticos e métodos computacionais para a simulação de escoamentos em meios porosos. Na engenharia de Petróleo, a otimização dos processos de recuperação de hidrocarbonetos está intimamente relacionada com a simulação precisa destes processos. Para isto, diversos fatores devem ser adequados e considerados nos modelos físico-matemático e numéricos, como por exemplo a troca de massa e momento linear das fases que escoam, suas relações de capilaridade e mobilidade, a estabilidade dos poços de produção e injeção, dentre inúmeros outros [Correa 2013]. O código, escrito em FORTRAN, e desenvolvido para o modelo computacional, é dividido em 14 módulos que contém uma variedade de funcionalidades matemáticas e físicas. O mesmo recebeu otimizações de paralelização de diretivas OpenMP. Com o uso das ferramentas do suite da INTEL Parallel Studio, como o VTune Amplifier, Thread Advisor e o Parallel Advisor, foi possível identificar os hotspots do código, possibilitando assim uma análise mais apurada do código dentro do módulo com o maior gasto computacional chamado transporte.F90 onde se encontra o problema numérico. Foi identificado via VTune que grande parte do tempo de execução da rotina era gasto em operações de sincronização entre threads. Foi aplicada uma estratégia para mudar o padrão de escalonamento das threads geradas pela API do compilador através do modelo de programação de memória compartilhada padrão OpenMP onde foi possível reduzir o spintime (sincronismo entre threads) de forma considerável e reduzindo assim o CPI rate que é o índice de latência presente na execução do código, gerando também um ganho de desempenho de 27x, levando em consideração o tempo de execução do código com afinidade de threads Balanced, comparado ao código naive, executado de forma serial como pode ser visto em [Teixeira et al. 2018]. Apesar de realizar cálculos de dados 3D, o código foi criado inicialmente para realizar cálculos em 2D, levando assim a uma programação fora das boas práticas para a paralelização e causando um impacto negativo nos esforços de otimização. Em inúmeras áreas do código podemos ver condicionais aninhadas, tanto em outras condicionais, quanto em loops durante a execução do código, como também podemos ver inúmeras escritas em disco, também em loops, durante a execução. Este motivo nos leva ao caminho de reestruturação total do código, voltado a programação seguindo as boas práticas da paralelização e focando a escalabilidade, desta forma, atingindo ganhos de desempenho ainda maiores comparados aos que puderam ser obtidos apenas utilizando estratégias EWS-SYNC, no código original.

O método presente no código possui um estêncil local, em que cada célula é calculada de forma individual necessitando apenas, de informações de suas células vizinhas. Esse tipo de estêncil favorece a paralelização do código devida sua baixa troca de informações e sua facilidade na organização dos dados em memória. Levando em consideração estudo realizado referente ao método, uma nova estratégia inclinada ao uso de MPI em conjunto com OpenMP foi escolhida para dar prosseguimento na otimização do código, todavia, implicações surgiram quando uma estratégia adotada para a divisão do domínio em processos de memória não compartilhada nos revelou que o código não estava apto a receber otimizações em MPI sem uma reestruturação no módulo de construção do domínio físico da malha, desta maneira o código está sendo totalmente reestruturado para a implementação de estratégia MPI, reestruturação ainda em fase inicial mas que já apresenta resultados promissores.

O modelo computacional deste estudo apresenta uma metodologia numérica, proposta em [Correa 2013], para a simulação do escoamento bifásico (água e óleo) em um reservatório rígido altamente heterogêneo. Este problema é modelado por um sistema de equações diferenciais parciais, basicamente composto por um subsistema elíptico para a determinação do campo de velocidades e uma equação hiperbólica não linear para o transporte das fases que escoam (equação da saturação). Do ponto de vista numérico, o modelo propõe a aplicação de um método de elementos finitos localmente conservativo para a velocidade da mistura e um método de volumes finitos não-oscilatório de alta ordem, baseado em esquemas centrais, para a equação hiperbólica não-linear que governa a saturação das fases.

Este artigo apresenta a análise das oportunidades de otimizaçãoo de código do método numérico em questão para as plataformas Intel® Xeon™ com a biblioteca MPI sem a implementação EWS-SYNC incluída pois esta será incluída em próxima etapa do trabalho, será seguida a metodologia de otimização guiada por perfilagem (*profile guided optimization*), com o uso das ferramentas do suíte Intel ® Parallel Studio ™, como o VTune Amplifier™, Thread Advisor™ e Parallel Advisor™.

2. O Modelo Matemático

Seja $\Omega \subset \mathbb{R}^3$ um domínio conexo, aberto e limitado e *I* um tempo de intervalo. Consideramos a lei de conservação escalar da seguinte forma:

$$\phi \frac{\partial s}{\partial t} + \nabla \cdot \mathbf{f} = 0 \text{ in } \Omega \times I. \tag{1}$$

Onde $\phi : \Omega \Rightarrow (0, \phi^{max}]$ é o coeficiente de armazenamento, $s : \Omega \times I \rightarrow Im\{s\} = [s^{min}, s^{max}]$ é a função escalar, e a função vetorial $\mathbf{f} : Im\{s\} \rightarrow \mathbb{R}^3$ é o fluxo da quantidade conservada s. Particularmente, para os testes considerados neste trabalho

$$\mathbf{f} = s\mathbf{v} = \left\{ \begin{array}{c} s \\ 0 \\ 0 \end{array} \right\}. \tag{2}$$

O método numérico utilizado para aproximar a solução da equação (1) é descrito, em detalhes, em [Correa 2013].

3. Análise do Perfil do Código

Para que se possa compreender o perfil do código, é necessário antes conhecer sua estrutura básica, com cada um de seus módulos principais, suas funções e a relação chamado-chamador entre eles, bem como a complexidade computacional de cada um. O Algoritmo 1 mostra um trecho de pseudocódigo do módulo principal, onde se encontra o maior esforço computacional.

O trecho executa o método *Kurganov-Tadmor* DxD. Para isso, é necessário que seja definido o número de *Courant* (cr), as velocidades locais, que são resolvidas pela função **MaxDer**, e o Δt calculado dinâmicamente de forma a respeitar a restrição de CFL.

```
1  begin
2      cr ← 0.125;
3      #pragma omp parallel;
4      {
5        call maxDer();
6      }
7      Δt ← nDias / numPassosTransp;
8      call KTDD_RT();
9  end
```

Algoritmo 1: Um trecho do código do módulo transporte [Teixeira et al. 2018]

```
1  begin
2      #pragma omp parallel;
3      {
4      call dividirTrabalho (iniTrabalho, fimtrabalho);
5      for (passo ← 1; passo ≤ numPassosTransp; passo ← passo + +) do
6          call derivaU (iniTrabalho, fimTrabalho) ;
7          #pragma omp barrier;
8          call fkt (iniTrabalho, fimTrabalho);
9          #pragma omp single;
10             tTransporte = tTransporte + Δt
11     end
12     call derivaU (iniTrabalho, fimTrabalho) ;
13     }
14 end
```

Algoritmo 2: Um trecho do código da função KTDD_RT que executa o método Kurganov-Tadmor DxD [Teixeira et al. 2018]

Conforme demonstrado pela estratégia EWS-SYNC onde para um grande número de threads, a barreira do OpenMP que faz com que todas as threads sejam sincronizadas juntas, o desempenho tende a diminuir mesmo com a divisão explicita de trabalho. Isso se deve ao fato de que basta que uma thread demore um pouco mais a atingir a barreira para que todas as outras fiquem paralisadas, aguardando-a. Em algoritmos onde a etapa seguinte não pode prosseguir até toda uma dada etapa tenha terminado,não há outra saída mas não é o caso do método EWS-SYNC que apresenta uma dependência limitada entre as threads, a estratégia EWS-SYNC consiste na remoção das barreiras implícitas do OpenMP por sistema explícito de sincronização das threads, basicamente a sincronização explicita permite que determinada thread aguarde execução de suas adjacentes apenas.

[Teixeira et al. 2018]demostra o ganho de desempenho ao se aplicar a estratégia EWS-SYNC. Devido a tais resultados conclui-se que a princípio a estratégia EWS-SYNC deve ser mantida na futura implementação do código em sua condição de utilização em memória híbrida.

4. Otimização

Os experimentos conduzidos com a implementação do código EWS-SYNC e do código MPI, foram realizados em uma máquina que contém Intel Xeon CPU E5-2698 v3 @ 2.30GHz com 16 cores físicos .

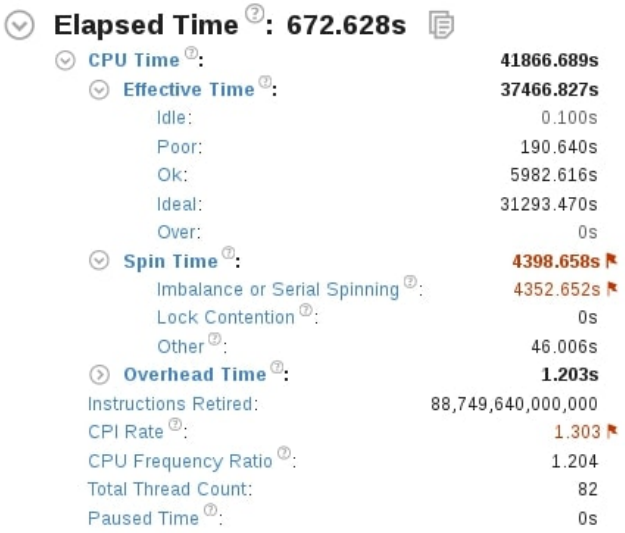

Figura 1. Análise *Advanced Hotspots* via VTune Amplifier™ com execução do código *naive* já com diretivas OpenMP. [Teixeira et al. 2018]

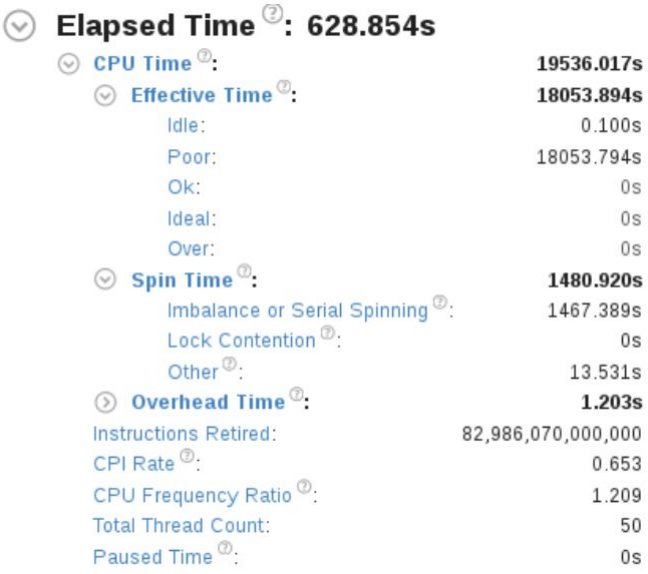

Figura 2. Análise *Advanced Hotspots* via VTune Amplifier™ com execução do código com a estratégia EWS-SYNC. [Teixeira et al. 2018]

Pudemos observar através do Intel ® VTune Amplifier™, que grande parte do tempo de execução da função é gasto em operações de sincronização, onde em múltiplos momentos as *threads*, que terminam a execução de um *loop* dentro de uma função, aguardam o término de execução de cada uma das outras *threads* para dar continuidade a execução do código. A analise realizada na Figura 1 foi a *Advanced Hotspots* e apresentou um alto índice de *Spin Time* (desbalanceamento de carga ou tempo de execução

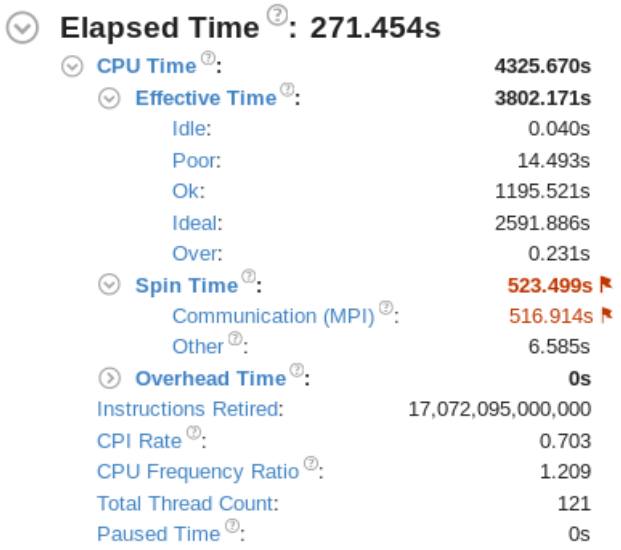

Figura 3. Análise *Advanced Hotspots* via VTune Amplifier™ com execução do código *MPI* sem a implementação EWS-SYNC.

serial) causado pelo mecanismo de sincronização do OpenMP e um alto índice de *clock ticks* por instrução *retired* (**CPI** *rate*). A Figura 2 mostra o resultado, também da análise *Advanced Hotspots*, do código com implementação EWS-SYNC onde o *spin time* fica reduzido a 1480 segundos, o que significa que é menos tempo gasto nas barreiras do OpenMP. Isto acontece pelo fato da estratégia EWS-Sync consistir em se substituir a barreira que aparece na linha 7 do Algoritmo 2 por um mecanismo de *lock* que permite que uma determinada *thread* aguarde apenas as threads adjacentes para continuar adiante, conforme proposto em [Cabral et al. 2018].

Os resultados iniciais do código em desenvolvimento com implementação MPI aparecem Figura 3, onde ocorre melhoria significativa de desempenho em relação à implementação EWS-SYNC, como se trata de implementação MPI nesta análise o *spin time* trata do tempo de comunicação MPI, resultado este que ainda pode ser aperfeiçoado uma vez que o código ainda se encontra em fase inicial de implementação.

Em geral, o **CPI** é a primeira métrica a ser verificada na performance da aplicação durante o aperfeiçoamento do código. Essa métrica é determinada ao dividir o número de ciclos do processador (*clock ticks*) pelo número de instruções *retired* (que já terminaram). O valor de **CPI** de uma aplicação é o indicador de quanta latência sua execução possui. Um alto índice de **CPI** significa mais latência, geralmente, durante a execução. Isso significa que a aplicação demorou mais *clock tick* por instrução *retired*. Geralmente o código, o processador e a configuração do sistema operacional influenciam no **CPI** de uma carga de trabalho, e o valor 0,75 é uma valor razoável (máximo) para essa métrica.

4.1. Sincronização entre Threads Adjacentes: EWS-Sync e Reestruturação para MPI

Um tempo baixo de *spin* pode ser desejado ao invés do aumento de trocas de contexto de *threads*. Um tempo alto de *spin*, entretanto, pode diminuir o tempo produtivo de trabalho. Neste intuito foi criada uma estratégia para alterar o padrão de escalonamento das

threads geradas pela **API** do compilador através do modelo de programação de memória compartilhada padrão OpenMP de forma a diminuir o tempo de sincronização entre elas e o aumento no desempenho.

Os experimentos conduzidos com a implementação EWS-Sync e MPI do código foram realizados em uma máquina que contém Intel ® Xeon Intel ® Intel Xeon CPU E5-2698 v3 @ 2.30GHz com 16 cores físicos. O impacto desta estratégia também pode ser observado ao se comparar as métricas geradas pelo Intel ® VTune ™. Na Figura 1 percebe-se que o *spin time* alto, em torno de 4398 segundos, o que significa bastante tempo sendo gasto na barreira do OpenMP, enquanto na Figura 2 o *spin time* fica reduzido a 1480 segundos, o que significa que há menos tempo gasto nas barreiras do OpenMP. Isto acontece pelo fato da estratégia EWS-SYNC consistir em se substituir a barreira que aparece na linha 7 do Algoritmo 2 por um mecanismo de *lock* que permite que uma determinada *thread* aguarde apenas as threads adjacentes para continuar adiante, conforme proposto em [Cabral et al. 2018].

Percebe-se que a carga de trabalho foi melhor balanceada com a estratégia EWS-SYNC, o que é importante para o ganho de desempenho final no código. Houve uma boa redução de **CPI** *rate* e no tempo de *spin* obtidos pela estratégia EWS-Sync.

4.2. Resultados de Tempo de execução e Speedup referêntes à implementação MPI

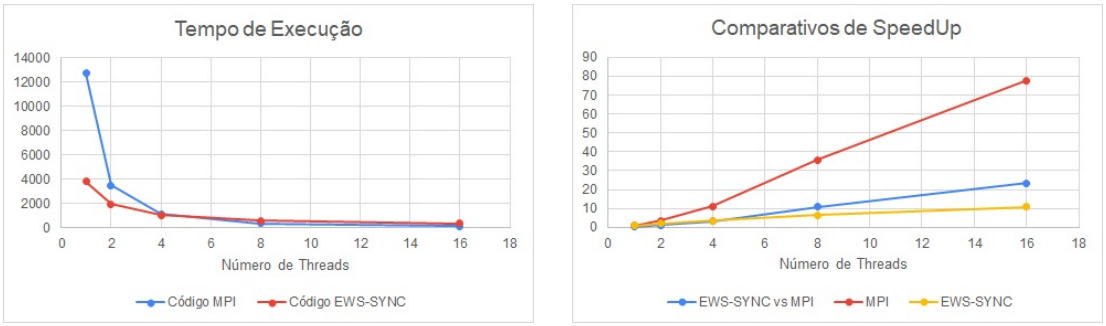

Figura 4. Tempo de execução(esquerda) e Speedup (direita)

As análises referentes ao Speedup e ao tempo de execução foram realizadas com o mesmo processador arquitetura Intel Xeon® CPU E5-2698 v3 @ 2.30GHz 16 cores físicos, utilizando-se 16 **threads** e com carga de 3200000 elementos.

Na Figura 4 à esquerda são apresentados os tempos de execução alcançados quando executado o código com MPI e com implementação ESW-SYNC, os resultados do código MPI apresentam um ganho no tempo de execução próximo a 2,5x. Na Figura 4 à direita são apresentados os resultados de **Speedup** do código com implementação MPI em relação ao código com implementação EWS-SYNC, MPI em relação à Naive e ESW-SYNC em relação ao código Naive, na implementação inicial de MPI verifica-se um ganho de **Speedup** na ordem de 77x em relação ao Naive e um ganho de 23x em relação ao código EWS-SYNC observa-se também o ganho do código EWS-SYNC em relação ao código Naive com ganho aproximido de 10x em relação ao **Speedup**

5. Conclusão

Por sua vez, o código reestruturado voltado a compatibilidade com o MPI, consegue um desempenho ainda melhor comparado ao da estratégia EWS-Sync. O motivo inicial do

ganho de desempenho através da implementação MPI ocorre devido ao balanceamento de carga explicito que define de maneira direta a quantidade de carga de trabalho que cada nó realizará durante a simulação. Porém é necessário ainda realizar uma pesquisa mais aprofundada para que todos os motivos desse ganho de desempenho com a implementação MPI fiquem claros, com essa divisão explicita de domínio, pode haver relação com fácil acesso a memória.

Devido a todas essas informações recolhidas com as devidas análises em cada ponto forte e fraco no desempenho deste código, o foco do trabalho segue em uma grande reestruturação do código, tanto no módulo de construção do domínio físico da malha, quanto no módulo de transporte, com o intuito de atingir, desta forma, o maior ganho de desempenho possível utilizando estratégias em OpenMP e MPI para criar um código de alta escalabilidade e aperfeiçoado seguindo todas as boas práticas na área de paralelização e vetorização de código.

Pretende-se realizar a divisão do domínio em dois níveis, sendo o primeiro nível a divisão entre diversos processadores distribuídos, e o segundo nível dividindo-o em cada processador, tendo assim um código preparado para funcionar em um sistema computacional de memória híbrida para buscar-se o aumento ainda maior de Speedup.

Agradecimentos

Este projeto teve apoio do CNPq. Gostaríamos de agradecer ao Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP) por nos disponibilizar o uso do cluster multi-core heterogêneo para a execução dos nossos experimentos. Esses recursos foram parcialmente financiados pela Intel® via projetos intitulados Intel Parallel Computing Center, Modern Code Partner, e Intel/Unesp Center of Excellence in Machine Learning.

Referências

Cabral, F. L., Osthoff, C., Costa, G., Gonzaga de Oliveira, S., Brandão, D., and Kischinhevsky, M. (2018). An openmp implementation of the tvd hopmoc method based on a synchronization mechanism using locks between adjacent threads on xeon phi (tm) accelerators. *Lecture Notes in Computer Science. Springer International Publishing*, 3:701–707.

Correa, M. R.; Borges, M. R. (2013). A semi-discrete central scheme for scalar hyperbolic conservation laws with heterogeneous storage coefficient and its applications to porous media flow. *International Journal for Numerical Methods in Fluids*, 73(3):205–224.

Teixeira, T., Cabral, F., Osthoff, C., Borges, M. R., and Souto, R. P. (2018). Analysis of optimization opportunities for intel xeon phi and intel xeon scalable processors environments of a numerical method for the biphasic flow of fluids in porous media. pages 237–242.