Vetorização e Análise de Algoritmos Paralelos para a Migração Kirchhoff Pré-empilhamento em Tempo

Rodrigo Alves Prado da Silva[1], Maicon Melo Alves[1], Cristiana Barbosa Bentes[2] e Lúcia Maria de Assumpção Drummond[1]

[1]Universidade Federal Fluminense (UFF), Niterói, RJ, Brasil
[2]Universidade do Estado do Rio de Janeiro (UERJ), Maracanã, RJ, Brasil

{rprado, mmelo, lucia}@ic.uff.br, cris@eng.uerj.br

Resumo. *A Migração Kirchhoff Pré-empilhamento em Tempo (ou PKTM, do inglês Pre-stack Kirchhoff Time Migration) é parte central do processo de exploração de petróleo. Como o PKTM é computacionalmente intensivo, muitos trabalhos propuseram o uso de aceleradores como GPU (Graphical Processing Units) para reduzir o seu tempo de execução. Embora os processadores modernos possuam um recurso de aceleração vetorial, apenas um trabalho avaliou o uso deste recurso para acelerar o PKTM. Contudo, este trabalho avaliou apenas a versão sequencial desta aplicação. Nesse trabalho, propõe-se uma analise da vetorização de duas versões paralelas do PKTM. Para a primeira versão paralela, foi utilizado OpenMP e para a segunda, foi utilizado MPI. Em relação à vetorização, foram consideradas a vetorização automática, executada pelo compilador, e a vetorização manual, implementada pelo programador. Uma análise experimental mostrou que a vetorização automática no código com o OpenMP produziu melhores resultados do que os obtidos no código sequencial e no código com MPI. Assim, foram propostas algumas otimizações que permitiram que as versões sequenciais e com MPI obtivessem um desempenho similar ao alcançado no código com OpenMP.*

1. Introdução

O primeiro passo na exploração de petróleo e gás consiste em realizar um processamento nos dados sísmicos que foram adquiridos em uma determinada área de interesse. Esse processo fornece imagens da subsuperfície a fim de permitir a interpretação das estruturas geológicas presentes nessa área. A partir do conhecimento dessas estruturas, é possível identificar áreas onde o petróleo possa ser encontrado e extraído. Dentre as etapas executadas neste processamento sísmico, a migração sísmica é considerada como sendo o passo central de todo este processo. A migração sísmica produz uma imagem mais fiel das estruturas geológicas encontradas em subsuperfície ao colapsar as difrações hiperbólicas e mover as estruturas mergulhantes para suas reais posições.

Um dos métodos mais populares de migração sísmica é a Migração Kirchhoff Pré-empilhamento em Tempo (PKTM, do Inglês *Pre-stack Kirchhoff Time Migration*) que baseia-se no procedimento de soma de difrações. O PKTM é amplamente utilizado por sua simplicidade, eficiência, confiabilidade e flexibilidade de I/O [Xu et al. 2014]. No entanto, esta migração é computacionalmente intensiva, ou seja, mesmo em supercomputadores, sua execução pode levar semanas ou até meses para ser concluída [Shi et al. 2011].

Devido às inerentes características que permitem o paralelismo de dados do PKTM [Rizvandi et al. 2011], muitos trabalhos como [Xu et al. 2014], [Shi et al. 2011], [Panetta et al. 2012] e [Sun and Shi 2012] propuseram soluções para reduzir o tempo de execução deste algoritmo ao adotar dispositivos aceleradores como as *Graphical Processing Units* (GPU) ou *Field-programable Gate Array* (FPGA). Outros trabalhos como [Dai 2005] propuseram a divisão e distribuição do processamento do PKTM em *cluster* computacional usando o padrão MPI (*Message Passing Interface*). Já outros trabalhos como [Yang et al. 2011] propuseram uma solução híbrida que faz uso tanto do paralelismo de memória distribuída (MPI) quanto do paralelismo de memória compartilhada (OpenMP), além de utilizar também os recursos de aceleração da GPU.

Contudo, nenhum destes trabalhos considerou um importante recurso de aceleração disponível na maioria dos processadores modernos. Este recurso, chamado de unidade de processamento vetorial, permite explorar o paralelismo de dados com granularidade fina ao realizar operações simultâneas em diferentes elementos de um mesmo vetor [Lomont 2011]. A segunda geração do processador Xeon Phi da Intel, Knights Landing, por exemplo, possui duas dessas unidades de vetorização em cada um dos seus *cores*, onde cada uma dessas unidades de vetorização permite realizar operações simultâneas em um vetor de 512 bits. Em outras palavras, pode-se executar, ao mesmo tempo, 16 operações de ponto-flutuante de precisão simples ou 8 operações de ponto-flutuante de precisão dupla.

Esse recurso de vetorização pode ser explorado por meio de um processo de vetorização automática provido por compiladores como GCC (*GNU Compiler Collection*) ou ICC (*Intel C/C++ Compiler*). Quando a opção de vetorização automática é ativada, o compilador procura por estruturas de código que possam ser vetorizadas, ou seja, por laços internos que não apresentem, por exemplo, controle de fluxo ou dependência de dados entre os elementos dos vetores que estão sendo processados. Portanto, a vetorização automática é transparente e não requer qualquer esforço por parte do programador [Mitra et al. 2013] [Schuchart et al. 2015] [Intel 2012]. Entretanto, por conta de decisões mais conservadoras tomadas pelo compilador, o código automaticamente vetorizado pode não extrair todo o potencial fornecido pela unidade de vetorização. Nestes casos, existe a possibilidade de realizar uma vetorização manual do código-fonte na qual o próprio programador utiliza funções de alto nível para acessar diretamente as instruções vetoriais disponíveis no processador [Hofmann et al. 2014].

Embora o trabalho descrito em [Melo Alves et al. 2016] tenha analisado o custo-benefício de adotar diferentes abordagens de vetorização para acelerar o PKTM sequencial, não há conhecimento a respeito de outro trabalho da literatura que tenha analisado o efeito da vetorização sobre códigos paralelos do PKTM. Estamos interessados em investigar como o uso de bibliotecas de paralelização, como OpenMP e MPI podem afetar a vetorização do código. Assim, neste trabalho, propõe-se avaliar o efeito da vetorização manual e automática sobre duas novas versões do PKTM, uma paralelizada com OpenMP e outra com MPI. Adicionalmente, o trabalho também apresenta algumas técnicas de otimização que foram aplicadas na versão sequencial e paralelizada com MPI com a finalidade de aumentar a quantidade de laços automaticamente vetorizados nessas versões do PKTM.

Uma análise experimental, considerando uma seção sísmica sintética e o com-

pilador ICC, revelou que o ganho de desempenho alcançado ao se combinar as duas abordagens, vetorização e paralelização, foi sempre superior ao desempenho obtido ao se aplicar isoladamente uma outra ou outra abordagem. Ao ser executado com 6 *threads*, a versão manualmente vetorizada e paralelizada com o OpenMP alcançou um speedup de 9,4 quando comparado com uma versão sequencial e escalar do PKTM. Além disso, os resultados também indicaram que, a exemplo da vetorização do código sequencial, a vetorização manual foi capaz de obter melhores resultados quando comparada com a versão automaticamente vetorizada.

2. Migração Kirchhoff com Pré-empilhamento em Tempo

O processamento de dados sísmicos consiste em produzir uma imagem de subsuperfície que reflita, o mais precisamente possível, as características físicas e a estrutura geológica de uma dada área de interesse. Essa etapa é executada a partir de dados brutos coletados durante o processo de aquisição sísmica.

Em uma aquisição de dados sísmicos, um emissor de ondas e um conjunto de receptores são posicionados em uma determinada área de interesse. A distância entre o emissor e cada um dos receptores é chamado de afastamento (ou *offset*, em Inglês). Periodicamente, o emissor propaga uma onda através da subsuperfície da Terra que, ao atingir uma camada subsuperficial de rocha, é refratada e refletida em direção à superfície. Esse processo, conhecido como *aquisição de tiro comum*, é executado várias vezes durante o processo de aquisição sísmica. Em cada tiro, o emissor e os receptores são alocados em diferentes posições da área de interesse a fim de que possam ser coletadas informações redundantes sobre um mesmo ponto da subsuperfície [Panetta et al. 2012][Yilmaz and Doherty 1987] [Xu et al. 2014].

Durante períodos discretos de tempo, os receptores coletam a energia refletida pelas camadas de rocha presentes em subsuperfície. Os dados coletados por um receptor durante um período discreto de tempo t são conhecidos como *amostra* e representam a amplitude da energia refletida por um ponto em subsuperfície. Considerando um refletor plano e paralelo, esse ponto, conhecido como *Ponto Médio Comum* (ou apenas CMP do Inglês *Commom Mid Point*), está localizado na posição central entre o emissor e o receptor. Um conjunto de amostras coletadas por um receptor durante um tiro é chamado de *traço sísmico* e representa a energia refletida por um CMP durante todo o tempo de propagação da onda. O conjunto de traços sísmicos de uma determinada área denomina-se *seção sísmica* e é geralmente gravado em um formato de arquivo padrão [Yilmaz and Doherty 1987].

Após a execução das etapas de pré-processamento, a seção sísmica está pronta para ser migrada [Yilmaz and Doherty 1987]. Basicamente, a migração: (i) colapsa as difrações hiperbólicas, (ii) move as estruturas mergulhantes para suas reais posições em subsuperfície e (iii) aumenta a resolução espacial. Uma seção sísmica precisa ser migrada, porque um ponto difrator, ao receber a excitação de uma fonte de energia, produz uma onda semicircular (de acordo com a Lei de Huygens) que acaba sendo coletada pelos receptores como uma hipérbole. Assim, a imagem resultante pode apresentar estruturas geológicas que não estejam em suas posições reais ou que nem sequer existam, de fato, na área de interesse. Desse modo, a migração corrige essas distorções e gera uma imagem mais precisa a respeito das características físicas e das estruturas geológicas presentes no

subsolo [Yilmaz and Doherty 1987] [Claerbout 1985].

Mais especificamente, o PKTM recebe como entrada uma seção pré-empilhada de afastamento comum a fim de produzir uma imagem final em coordenadas de tempo. O pressuposto básico do PKTM é de que qualquer ponto na seção sísmica pode ser considerado como sendo um refletor mergulhante. Neste sentido, o método assume que cada um desses pontos estaria localizado no ápice de uma hipérbole. Sendo assim, cada ponto deve receber de volta a energia que foi dispersada durante o processo de aquisição de dados, ou seja, o ápice deve receber a contribuição de energia de todas as amostras que compõe essa hipérbole [Teixeira et al. 2013]. O comportamento desta hipérbole pode ser definido por uma equação de tempo de trânsito duplo que determina os pontos (amostras de entrada) que devem contribuir para o seu ápice (amostras de saída) [Yilmaz and Doherty 1987].

Algorithm 1 PKTM Sequencial e Escalar

```
    /* Laço de Offset */
 1: Para Todos Offsets Faça                                          ▷ Não Vetorizado
        /* Laço de Traços de Entrada */
 2:     Para Todos Traços de Entrada Faça                            ▷ Não Vetorizado
            /* Laço de Cópia de Amostras de Traço de Entrada */
 3:         Para Todos Amostras de Entrada Faça Copia_Amostra()      ▷ Vetorização Auto e Manual
            /* Laço de Filtragem */
 4:         Para Todos Frequências de Corte Faça                     ▷ Não Vetorizado
 5:             Aplica_Filtro_Anti-alias()                           ▷ Vetorização Auto e Manual
                /* Laço de Cópia de Amostras Filtradas */
 6:             Para Todos Amostras Filtradas Faça Copia_Amostra()   ▷ Vetorização Auto e Manual
 7:         Fim Para
            /* Laço de Migração */
 8:         Para Todos Amostras de Saída Faça                        ▷ Não Vetorizado
 9:             Le_Velocidade()
10:             Determina_Traços_Dentro_da_Abertura()
                /* Laço de Contribuição */
11:             Para Todos Traços de Saída na Abertura Faça          ▷ Vetorização Manual (Parcial)
12:                 Calcula_Tempo_de_Transito()
13:                 Seleciona_Amostras_para_Interpolação()
14:                 Calcula_Operador_de_Migração()
15:                 Define_Filtros()
                    /* Laço de Cópia de Amostras Selecionadas */
16:                 Para Todos Amostras Selecionadas Faça Copia_Amostra()   ▷ Vetorização Manual
17:                 Interpola_Amostras_Selecionadas()
18:                 Calcula_Fator_de_Obliquidade()
19:                 Calcula_Angulo_de_Abertura()
20:                 Calcula_Fator_de_Espalhamento_Geometrico()
21:                 Corrige_Amplitude()
22:                 Acumula_Contribuição()
23:             Fim Para
24:         Fim Para
25:     Fim Para
26: Fim Para
```

Uma versão escalar do PKTM é apresentada no Algoritmo 1. Para cada *offset*, o PKTM executa o *Laço de Traços de Entrada* (linhas 2 a 25) onde cada traço de entrada do *offset* atual é processado. Em seguida, o PKTM executa o *Laço de Filtragem* (linhas 4 a 7) a fim de obter diferentes versões filtradas do mesmo traço de entrada. Para isso, o

algoritmo aplica, de acordo com a frequência de corte atual, um filtro *anti-alias* em cada uma das amostras do traço de entrada (linha 5). Em seguida, algoritmo armazena todas as amostras filtradas em um vetor auxiliar (linha 6).

Após o processo de filtragem do traço de entrada, o algoritmo executa o *Laço de Migração* (linhas 8 a 24). Para cada amostra de saída, o PKTM lê a velocidade do CMP corrente (linha 9), além de determinar os traços que compõem a abertura (linha 10). A abertura define os traços de saída que receberão as contribuições de um dado traço de entrada. Em seguida, o PKTM executa o *Laço de Contribuição* (linhas 11 a 23). No primeiro passo desse laço, o tempo de trânsito duplo é calculado (linha 12) para identificar qual amostra de entrada deverá contribuir para a amostra de saída atual. Entretanto, como o tempo de trânsito calculado anteriormente se encontra em um domínio contínuo, o PKTM seleciona um conjunto de amostras do traço de entrada (linha 13) a fim de transpor, posteriormente, o valor calculado para um domínio discreto por meio de um processo de interpolação. Depois disso, o retardo horizontal do operador de migração é calculado (linha 14) e o seu valor é usado como parâmetro de entrada para determinar os filtros apropriados para o processo de *anti-alias* (linha 15).

Em seguida, o algoritmo calcula a amplitude da energia através de uma interpolação entre os filtros e as amostras selecionadas anteriormente (linha 17). Esse processo de interpolação visa prover um valor aproximado de amplitude de acordo com os filtros escolhidos e o conjunto de amostras previamente determinado. Então, o PKTM calcula: (i) o fator de obliquidade, (ii) o ângulo de abertura, e (iii) o fator de espalhamento geométrico (linhas 18, 19, e 20, respectivamente) a fim de realizar a correção da amplitude no passo seguinte (linha 21). Por fim, essa amplitude corrigida é acumulada na amostra de saída atual (linha 22). Uma descrição mais detalhada do PKTM pode ser encontrada em [Yilmaz and Doherty 1987].

Conforme indicado no Algoritmo 1, o *Laço de Cópia de Amostras de Traço de Entrada* (linha 3), o *Laço de Cópia de Amostras Filtradas* (linha 6) e todos os laços dentro do procedimento *Aplica_Filtro_Anti-alias* (linha 5) foram automática e manualmente vetorizados. Já o *Laço de Cópia de Amostras Selecionadas* (linha 16) e o *Laço de Contribuição* (linha 11) foram vetorizados apenas manualmente. Mais precisamente, o *Laço de Contribuição* foi parcialmente vetorizado, já que algumas etapas executadas neste laço não puderam ser vetorizadas por conta de restrições como controle de fluxo e dependência de dados. Assim, esse laço foi dividido em dois novos laços, um para executar as tarefas vetorizadas e outro para executar as tarefas escalares. Maiores detalhes sobre a vetorização manual e automática do PKTM sequencial estão descritos em [Melo Alves et al. 2016].

3. Versões Paralelas do PKTM

Com o intuito de comparar o efeito da vetorização sobre versões paralelas do PKTM, foram desenvolvidas duas novas versões paralelizadas do PKTM. A primeira utilizando OpenMP e a segunda utilizando MPI, como descrito a seguir.

3.1. PKTM Paralelo com OpenMP

Uma versão paralela *multithreading* do PKTM, implementada com OpenMP, é apresentada no Algoritmo 2. Foi verificado que o *Laço de Traços de Entrada* desse algoritmo

é responsável pela maior parte do tempo de processamento desta migração sísmica. De forma a reduzir esse tempo de execução, foi utilizada a diretiva de compilação *omp parallel for* para que o trabalho pudesse ser distribuído entre um dado número de *threads* (linha 2). Cada uma dessas *threads* processa uma mesma quantidade de traços de entrada, ou seja, todas as *threads* recebem uma carga de trabalho similar. A execução dos passos internos do *Laço de Traços de Entrada* segue conforme definido pelo Algoritmo 1.

Algorithm 2 PKTM Paralelo com OpenMP

```
   /* Laço de Offset */
1: Para Todos Offsets Faça
       /* Laço de Traços de Entrada */
       #pragma omp parallel for
2:     Para Todos Traços de Entrada Faça
           /* Laço de Filtragem */ [1]
3:     Fim Para
4: Fim Para
```

[1] O corpo do Laço de Filtragem foi omitido para simplificar o entendimento.

3.2. PKTM Paralelo com MPI

No caso do MPI, foi feita uma implementação considerando uma granularidade mais grossa do que a utilizada pela versão paralelizada com o OpenMP. Assim, optou-se pela distribuição do trabalho realizado pelo laço mais externo, ou seja, pelo *Laço dos Traços de Entrada*. Dessa forma, a quantidade de mensagens trocadas entre os processos é menor do que seria necessário para paralelizar a execução de cada um dos *offsets* da seção sísmica. Como a aplicação possui operações de entrada (leitura dos *offsets*) e saída (escrita dos traços migrados), optou-se por concentrar todas essas operações em um processo mestre, enquanto os demais processos, chamados trabalhadores, executam efetivamente o processo de migração dos traços de entrada.

Os Algoritmos 3 e 4 apresentam os pseudocódigos implementados em MPI dos processos mestre e trabalhador, respectivamente. No Algoritmo 4, o processo trabalhador envia uma mensagem de solicitação de *offset* ao processo mestre (linha 2) e, ao receber a resposta (linha 3), realiza o processamento do respectivo *offset* (linha 6). Este procedimento é repetido até que o processo mestre termine a leitura de todos os *offsets* da seção sísmica. Quando isto ocorre, o processo mestre responde ao processo trabalhador com uma mensagem de término (linha 7 do Algoritmo 3).

Repare que essa implementação favorece o balanceamento de carga entre os processos trabalhadores, já que um processo trabalhador, ao encerrar seu processamento, pode requisitar mais um *offset* ao processo mestre sem a necessidade de esperar que os demais processos trabalhadores terminem o seu processamento.

3.3. Análise das Vetorizações dos Códigos do PKTM Paralelo

Com o intuito de comparar o efeito da vetorização sobre as versões paralelas do PKTM, foram considerados os mesmos tipos de vetorização realizados anteriormente no código sequencial: (i) a versão automaticamente vetorizada pelo compilador e (ii) a versão manualmente vetorizada pelo programador. Portanto, essas duas abordagens de vetorização

Algorithm 3 PKTM Paralelo com MPI - Processo Mestre

Enquanto $\exists\ Fim\_processo[i] == FALSE,\ para\ i \in \{1 \ldots n\}$ **Faça**
    Recebe Mensagem *Pedido* de processo i
    Lê *Offset*
    Se $not\ Fim\ do\ Arquivo\ de$ *Offsets* **Então**
        Envia Mensagem *Offset* para processo i
    Senão
        Envia Mensagem *Fim* para processo i
        $Fim\_processo[i] = TRUE$
    Fim Se
Fim Enquanto

Algorithm 4 PKTM Paralelo com MPI - Processo Trabalhador

Enquanto $not\ Terminou$ **Faça**
    Envia Mensagem *Pedido* Para Processo Mestre
    Recebe Mensagem *msg* do Processo Mestre
    Se *msg* $==$ {*Fim*} **Então**
        $Terminou = TRUE$
    Senão
        / Laço de Offset */* [1]
    Fim Se
Fim Enquanto

[1] O corpo do Laço de Offset foi omitido para simplificar o entendimento.

foram aplicadas sobre as duas versões paralelas do PKTM apresentadas nas Seções 3.1 e 3.2.

Surpreendentemente, ao utilizar o OpenMP para efetuar a paralelização do código do PKTM, o compilador foi capaz de aumentar o número de laços vetorizados. Com isso, além de vetorizar automaticamente o *Laço de Cópia de Amostras de Traço de Entrada*, o *Laço de Cópia de Amostras Filtradas* e todos os laços dentro do procedimento chamado *Aplica_Filtro_Anti-alias*, o compilador foi capaz vetorizar automaticamente também o *Laço de Cópia de Amostras Selecionadas*. Este laço não foi vetorizado automaticamente porque o compilador acusou dependência de dados. O fato é que o OpenMP cria novas variáveis para o contexto de cada *thread* dentro do laço. Por conta desta criação de novas variáveis, o compilador provavelmente descarta a possibilidade de dois ponteiros apontarem para a mesma área de memória e consegue vetorizar o laço.

Quanto à vetorização manual, não houve diferença, uma vez que a implementação manual da vetorização é a mesma empregada na versão sequencial.

Ao utilizar o MPI para efetuar a paralelização do código do PKTM, o compilador teve a capacidade de vetorização reduzida uma vez que foi capaz de vetorizar somente o *Laço de Cópia de Amostras de Traço de Entrada* e o *Laço de Cópia de Amostras Filtradas*. O compilador acusou o impedimento à vetorização por dependência de dados nos laços que estão dentro do procedimento *Aplica_Filtro_Antialias()*. Quanto à vetorização manual, não houve mudanças, uma vez que a implementação manual da vetorização é a mesma empregada na versão sequencial.

A vetorização manual seguiu os mesmos passos descritos em [Melo Alves et al. 2016], enquanto que a vetorização automática foi realizada pelo ICC. Com a vetorização manual, foi possível vetorizar, em ambas as versões paralelas, os mesmos laços que foram vetorizados manualmente na versão sequencial. Todas as versões manualmente vetorizadas (Sequencial, MPI e OpenMP) apresentam a mesma quantidade de laços vetorizados.

3.3.1. Otimizações Para Vetorização Automática dos Códigos Sequencial e com MPI

Para fazer com que as versões sequencial e MPI vetorizassem automaticamente os mesmos laços que a versão OpenMP, foram utilizadas duas técnicas de otimização nessas versões.

A primeira delas visou resolver o problema de dependência de dados relacionado ao uso de ponteiros. Por ser conservador em suas decisões de vetorização, o compilador, ao analisar expressões que envolvam ponteiros, pode erroneamente supor que os endereços de memória apontados por esse tipo de variável possam se sobrepor, o que implicaria, em muitos casos, em dependência de dados e na não vetorização do laço de execução.

Para contornar esse problema, algumas variáveis do tipo ponteiro, localizadas dentro do *Laço de Offset* das versões sequencial e MPI, foram declaradas com a palavra-chave *restrict*. Essa palavra-chave, que foi originalmente introduzida na especificação ISO C99, foi usada para informar ao compilador que uma determinada variável do tipo ponteiro seria a única a apontar para um determinado endereço de memória. Desta forma, o compilador foi capaz de vetorizar automaticamente nas versões sequencial e com MPI os mesmos laços que foram vetorizados na versão OpenMP, já que não havia mais o risco de haver dependência de dados nestes laços de execução.

Além do uso da palavra-chave *restrict*, foi necessário aplicar mais uma técnica de otimização na versão MPI. Nesta versão, o compilador não foi capaz de vetorizar automaticamente laços presentes no processo de filtragem dos traços de entrada (procedimento *Aplica_Filtro_Anti-alias()*). Para vetorizar automaticamente esses laços, foi preciso impedir que o compilador efetuasse o *inlining* do procedimento de filtragem para dentro do código da migração. Dessa forma, as chamadas de procedimentos foram efetuadas para a execução desse procedimento, ao invés da realização da cópia dos mesmos. Em consequência disso, foi acrescentado um *overhead* da chamada de procedimento de filtragem na execução da migração do *offset*. Mas esse *overhead* foi pequeno se comparado a não vetorização automática da versão MPI do PKTM.

4. Resultados Experimentais

Os testes foram realizados em uma máquina com processador Intel i7-5930K 3.50 GHz, 32 GB de memória RAM e sistema operacional Ubuntu 14.04. Este processador possui seis núcleos de processamento e é equipado com o conjunto de instruções vetoriais *AVX2* (*Advanced Vector Extensions 2*). Esse conjunto de instruções permite realizar operações simultâneas em um vetor de 256 bits. O compilador utilizado foi o ICC versão 17.0.2 e a versão de API OpenMP utilizada foi a 4.5. Para o MPI foi utilizada a versão 1.6.5 do OpenMPI.

Todas as versões do PKTM foram compiladas com a opção *O3* do ICC a qual permite que o compilador efetue, além de diversas otimizações, a vetorização automática do código-fonte. Para as versões escalar e manualmente vetorizadas, a opção *no-vec* foi utilizada para que o processo de vetorização automática não fosse executado pelo compilador.

Todas as versões do PKTM implementadas e analisadas neste trabalho foram baseadas no *Seismic Unix* versão 44R4, que é um pacote de ferramentas *Open Source* amplamente utilizado pela indústria e academia para realizar processamento sísmico. Como dado de entrada, foi utilizada uma seção sísmica sintética com as seguintes características: 63 *offsets*, 128 traços por *offset*, 512 amostras por traço, intervalo para cada *offset* de 20 metros e intervalo de obtenção das amostras de 0,004 segundos. Ao todo, essa seção sísmica é composta por 8064 traços de entrada. Os arquivos fontes e o conjunto de dados de entrada utilizados podem ser obtidos no endereço `https://github.com/rodrigo-prado/kirchhoff`.

Para os testes com OpenMP foram utilizadas 2, 4 e 6 *threads* e para os testes usando o MPI foram utilizados 1 processo mestre e 2, 4 e 6 processos trabalhadores.

A Tabela 1 apresenta (i) o número de *threads* e processos trabalhadores das versões do PKTM, (ii) o tempo médio de cinco execuções, (iii) o *speedup* da vetorização e (iv) o *speedup* geral, que é o combinado da paralelização com a vetorização. O *speedup* da vetorização compara, para cada versão sequencial ou paralela, o tempo de execução das versões vetorizadas em relação a versão escalar. O *speedup* geral compara cada versão com a versão sequencial e escalar do PKTM. Como a variação do tempo de execução das cinco rodadas foi insignificante em todos os casos, optou-se por não apresentar intervalos de confiança para a média do tempo de execução e para os valores de *speedup*.

Analisando a Tabela 1, inicialmente, pode-se observar que as otimizações propostas conseguiram melhorar o tempo de execução da versão automática do código sequencial. É possível observar também que a versão OpenMP Manual com 6 *threads* obteve o melhor desempenho, com *speedup* geral igual a 9,4, seguida da versão MPI Manual que alcançou *speedup* geral de 9,1. Esse resultado mostra que a vetorização manual foi mais eficiente do que a vetorização automática por conseguir vetorizar uma quantidade maior de laços de execução do PKTM. Além disso, as versões com MPI apresentaram tempos de execução maiores do que as versões com OpenMP, o que pode ser explicado pelo tempo gasto com as operações de trocas de mensagens do MPI.

Em relação à vetorização automática, observa-se que diferentes técnicas de paralelismo (OpenMP e MPI) resultaram em diferentes níveis de vetorização do código. O compilador conseguiu vetorizar mais laços do algoritmo com OpenMP do que com MPI. A dificuldade de vetorizar o código com MPI ainda foi maior do que na versão sequencial. Embora o ganho de desempenho tenha se mantido constante entre as diferentes técnicas de vetorização e graus de paralelismo, a dificuldade de vetorização, por parte do compilador, foi maior quando se utilizou o padrão MPI.

Considerando o impacto da vetorização automática otimizada no MPI, verifica-se que, quando otimizado, o código com MPI foi capaz de alcançar *speedups* de vetorização iguais ao do código com OpenMP, isto é, 1,3 nas execuções com 2, 4 e 6 *threads*, no caso do OpenMP, e nas execuções com 2, 4 e 6 processos trabalhadores, no caso do MPI. É

Threads ou Proc. Trab.	Versão do PKTM	Tempo (em segundos)	Speedup da Vetorização	Speedup Geral
1	Sequencial Base	92,5	1,0	1,0
	Sequencial Auto	74,0	1,3	1,3
	Sequencial Auto Otimizado	69,7	1,3	1,3
	Sequencial Manual	43,6	2,1	2,1
2	OpenMP Base	46,5	1,0	2,0
	OpenMP Auto	35,1	1,3	2,6
	OpenMP Manual	23,5	2,0	3,9
	MPI Base	48,5	1,0	1,9
	MPI Auto	48,6	1,0	1,9
	MPI Auto Otimizado	37,1	1,3	2,5
	MPI Manual	23,6	2,1	3,9
4	OpenMP Base	24,2	1,0	3,8
	OpenMP Auto	18,2	1,3	5,1
	OpenMP Manual	12,7	1,9	7,3
	MPI Base	24,9	1,0	3,7
	MPI Auto	24,9	1,0	3,7
	MPI Auto Otimizado	19,2	1,3	4,8
	MPI Manual	12,8	2,0	7,3
6	OpenMP Base	16,6	1,0	5,6
	OpenMP Auto	12,9	1,3	7,1
	OpenMP Manual	9,8	1,7	9,4
	MPI Base	19,7	1,0	4,7
	MPI Auto	19,8	1,0	4,7
	MPI Auto Otimizado	15,6	1,3	5,9
	MPI Manual	10,1	1,9	9,1

Tabela 1. Tempo, *Speedup* da Vetorização e *Speedup* Geral obtidos.

possível observar também que, nestes casos, não houve alteração no ganho de desempenho alcançado pela vetorização diante da variação do número de *threads* ou processos trabalhadores.

Considerando o impacto da vetorização manual nas versões paralelas, nota-se uma leve redução do ganho de desempenho a partir do aumento da quantidade de *threads*, no caso do OpenMP e da quantidade de processos trabalhadores, no caso do MPI.

Finalmente, vale notar que o ganho de desempenho obtido ao se combinar a paralelização com a vetorização foi sempre maior do que o desempenho obtido com uso de apenas uma das duas abordagens. Portanto, pode-se concluir que, para o caso do PKTM, o uso em conjunto dessas duas técnicas diminuem significativamente o tempo total de execução dessa aplicação.

5. Conclusão e Trabalhos Futuros

Neste artigo, foram avaliadas diversas abordagens para vetorização em duas versões paralelas do PKTM, uma com OpenMP e outra com MPI. Através de diversos experimentos foi possível observar que os ganhos com as vetorizações automática e manual nas versões paralelas foram similares aos obtidos na versão sequencial. Além disso, verificou-se que

a vetorização automática produziu melhores resultados na versão com OpenMP do que nas demais, sequencial e com MPI.

As otimizações propostas para se alcançar a mesma eficiência de vetorização em todos os códigos se mostraram eficientes e apontam que seu uso em outros códigos, em que se deseje utilizar a vetorização automática, seja promissor.

Para trabalhos futuros, pretende-se implementar uma versão paralela híbrida utilizando tanto o MPI quando o OpenMP. Além disso, seria interessante avaliar vetorizações automática e manual do PKTM nas arquiteturas Intel MIC (*Many Integrated Core*) que permitem processar, ao mesmo tempo, dezesseis números de ponto flutuante com precisão simples. Por fim, pretende-se avaliar os efeitos da vetorização em outras aplicações paralelas de migração como a Migração Reversa no Tempo e a Migração por Mínimos Quadrados.

Referências

Claerbout, J. F. (1985). *Fundamentals of Geophysical Data Processing*. Pennwell Books.

Dai, H. (2005). Parallel Processing of Prestack Kirchhoff Time Migration on a PC cluster. *Computers & geosciences*, 31(7):891–899.

Hofmann, J., Treibig, J., Hager, G., e Wellein, G. (2014). Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi and manycore chips. In *Proceedings of the Workshop on Programming models for SIMD/Vector processing*, páginas 57–64. ACM.

Intel (2012). A Guide to Vectorization with Intel® C++ Compilers. Relatório técnico.

Lomont, C. (2011). Introduction to Intel Advanced Vector Extensions. *Intel White Paper*.

Melo Alves, M., Cruz Pestana, R., Alves Prado da Silva, R., e Drummond, L. (2016). Accelerating Pre-stack Kirchhoff Time Migration by Manual Vectorization. *Concurrency and Computation: Practice and Experience*.

Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., e Zhou, J. (2013). Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-powered ARM and Intel Platforms. In *27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, páginas 1107–1116. IEEE.

Panetta, J., Teixeira, T., de Souza Filho, P. R., da Cunha Filho, C. A., Sotelo, D., da Motta, F. M. R., Pinheiro, S. S., Rosa, A. L. R., Monnerat, L. R., Carneiro, L. T., et al. (2012). Accelerating Time and Depth Seismic Migration by CPU and GPU Cooperation. *International Journal of Parallel Programming*, 40(3):290–312.

Rizvandi, Nikzad Babaii e Boloori, A. J., Kamyabpour, N., e Zomaya, A. Y. (2011). MapReduce Implementation of Prestack Kirchhoff Time Migration (PKTM) on Seismic Data. In *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, páginas 86–91. IEEE.

Schuchart, J., Waurich, V., Flehmig, M., Walther, M., Nagel, W. E., e Gubsch, I. (2015). Exploiting Repeated Structures and Vectorization in Modelica. In *11th International Modelica Conference*, páginas 265–272. Modelica Association Paris, France.

Shi, X., Li, C., Wang, S., e Wang, X. (2011). Computing Prestack Kirchhoff Time Migration on General Purpose GPU. *Computers & Geosciences*, 37(10):1702–1710.

Sun, P. e Shi, X. (2012). An OpenCL Approach of Prestack Kirchhoff Time Migration Algorithm on General Purpose GPU. In *13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, páginas 179–183. IEEE.

Teixeira, D., Yeh, A., e Sampath Gajawada, T. (2013). Implementation of Kirchhoff Prestack Depth Migration on GPU. *SEG Technical Program Expanded Abstracts*, 3683:3686.

Xu, R., Hugues, M., Calandra, H., Chandrasekaran, S., e Chapman, B. (2014). Accelerating Kirchhoff Migration on GPU using Directives. In *First Workshop on Accelerator Programming using Directives (WACCPD)*, páginas 37–46. IEEE.

Yang, C.-T., Huang, C.-L., e Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters. *Computer Physics Communications*, 182(1):266–269.

Yilmaz, O. e Doherty, S. M. (1987). *Seismic Data Processing*, volume 2 of *Investigations in Geophysics*. Society of Exploration.