Sobreposição de computação e escrita do método Fletcher com MPI

Rodrigo C. Machado, Arthur F. Lorenzon, Philippe O. A. Navaux

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{rcmachado,aflorenzon,navaux}@inf.ufrgs.br

Abstract. *The oil and gas industry needs to drill sites to locate new reservoirs of oil and gas. To make more informed decisions on where to drill, the industry utilizes seismic surveys, which involve sending and capturing seismic waves in the ocean. The Fletcher method is an algorithm that uses the collected data to run simulations. In this process, images of the ocean subsoil are produced and must be stored for later analysis. Writing these images to a storage unit accounts for a significant portion of the application's total time. To optimize the Fletcher method, we split and overlapped the computation and writing parts of the application using MPI. With our contribution, we achieved our best results with a speedup of 1.17x for executions on HD and 1.20x for SSD.*

Resumo. *Na busca por novas fontes de petróleo e gás, a indústria precisa realizar perfurações no solo. Visando mitigar os custos econômicos e riscos ambientais, a indústria emprega pesquisas sísmicas, que envolvem enviar e captar ondas sísmicas no oceano. O método Fletcher é um algoritmo que utiliza os dados coletados para realizar simulações de propagação de ondas sísmicas para gerar imagens do fundo do oceano. O armazenamento dessas imagens representa uma parte significativa do tempo total de execução. Para otimizar o desempenho da aplicação, foi utilizada MPI para sobrepor os fluxos de computação e escrita. Com nossas contribuições, atingimos nossos melhores resultados com speedup de 1,17x em HD e 1,20x em SSD.*

1. Introdução

A indústria de petróleo e gás desempenha um importante papel na economia global e brasileira. Empresas do setor buscam novas reservas frequentemente localizadas em áreas ambientalmente sensíveis, como oceanos. A descoberta dessas reservas envolve escavações dispendiosas e potencialmente prejudiciais ao meio ambiente, podendo causar contaminação de áreas protegidas e oceanos. Desse modo, para minimizar o impacto ambiental e reduzir os custos de exploração, realiza-se o mapeamento de áreas de interesse através de estudos sísmicos [ANP 2023]. Esses estudos utilizam equipamentos especializados para enviar ondas sísmicas do oceano ao subsolo, e as ondas refletidas são analisadas para criar imagens do fundo do mar e identificar possíveis reservas. O método *Fletcher* é um dos algoritmos utilizados por essas aplicações.

Estes algoritmos empregados em aplicações de mapeamento sísmico, principalmente o método *Fletcher*, são projetados para arquiteturas heterogêneas (e.g., CPU + GPU) e requerem a escrita periódica de dados no disco. Esses dados consistem nas ondas

sísmicas propagadas, cujos tamanhos variam conforme a complexidade das simulações realizadas e do tamanho da grade. Assim, ondas maiores resultam em arquivos mais extensos, o que aumenta o tempo necessário para a escrita no disco. Esse fator pode se tornar um gargalo significativo no desempenho, afetando negativamente o tempo de execução global da aplicação, especialmente em simulações de larga escala. A implementação do método Fletcher utilizada neste trabalho realiza a computação das propagações de onda em GPU, deixando a CPU ociosa.

Portanto, a otimização da escrita de dados no disco é essencial para melhorar o desempenho de aplicações sísmicas. Assim, estratégias como a sobreposição de computação com operações de entrada e saída (*I/O – input/output*) têm sido propostas como métodos alternativos para acelerar o processo de escrita, conforme discutido na Seção 4. Essas abordagens visam reduzir o impacto das operações de I/O no tempo total de execução, permitindo que a computação continue enquanto os dados são simultaneamente gravados, melhorando assim a eficiência e a rapidez das simulações sísmicas. Assim, MPI (*message-passing interface*) pode ser utilizado para criar processos independentes que podem operar em paralelo, separando as tarefas de computação das operações de comunicação.

Considerando o discutido anteriormente, este trabalho propõe quatro estratégias de sobreposição de computação com operações de entrada e saída utilizando *MPI*. Duas delas utilizam 2 processos, um encarregado de computar o método *Fletcher* e o outro de realizar a escrita em disco. O que diferencia as duas estratégias é como os processos se comunicam. Em uma delas a comunicação é bloqueante e a outra é não-bloqueante. A terceira estratégia tem um processo que realiza a computação e pode criar processos dinamicamente para realizarem a escrita no disco. Esta versão utiliza a funcionalidade *MPI I/O* que permite que diferentes processos tenham acesso a um mesmo arquivo paralelamente. A quarta é similar a terceira, mas nela os processos escritores criam um novo arquivo para cada estado de *grid* a ser salvo em disco. A escrita nessa versão não utiliza *MPI I/O*.

Este artigo é organizado da seguinte maneira, na Seção 2 são discutidos as fundamentações teóricas importantes a este trabalho, explicando o funcionamento do método Fletcher, dando um panorama do que é *MPI*. Na Seção 3 são apresentados alguns trabalhos relacionados. Na Seção 4 são apresentadas as implementações das quatro estrategias de sobreposição do método Fletcher. Na Seção 5 são descritos os experimentos realizados, especificando a máquina e parâmetros utilizadas. Em seguida os resultados obtidos são apresentados e discutidos. Por fim, a Seção 6 apresenta a conclusão sobre o trabalho e trabalhos futuros.

2. Fundamentação Teórica

2.1. Método Fletcher

O método *Fletcher* é um algoritmo que resolve equações diferenciais parciais (EDP) [Fletcher et al. 2009]. Essas EDP modelam a propagação de ondas sísmicas em função do tempo. O local onde a simulação ocorre é representado por uma estrutura de dados de 3 dimensões de pontos flutuantes, chamada de *grid*. Cada ponto desta *grid* representa características de um ponto do mundo real. Na simulação, o tempo é uma unidade lógica.

Algoritmo 1 Pseudocódigo Fletcher da versão Original

```
1: function FLETCHER(x, y, z, passo, tempoTotal)
2:     grid ← inicializaGrid(x, y, z)
3:     gravaEmDisco(grid)
4:     escritasEmDisco ← 1
5:     enviaParaGPU(grid)                                  ▷ grid CPU → GPU
6:     for it ← 1, it ≤ ceil(tempoTotal/passo), it++ do
7:         propagacaoOndaGPU(grid)
8:         if it * passo ≥ limiteParaEscrita then
9:             copiaParaMemoriaPrincipal(grid)             ▷ grid GPU → CPU
10:            gravaEmDisco(grid)
11:            escritasEmDisco++
12:            limiteParaEscrita ← limiteInicial * escritasEmDisco
13:        end if
14:    end for
15: end function
```

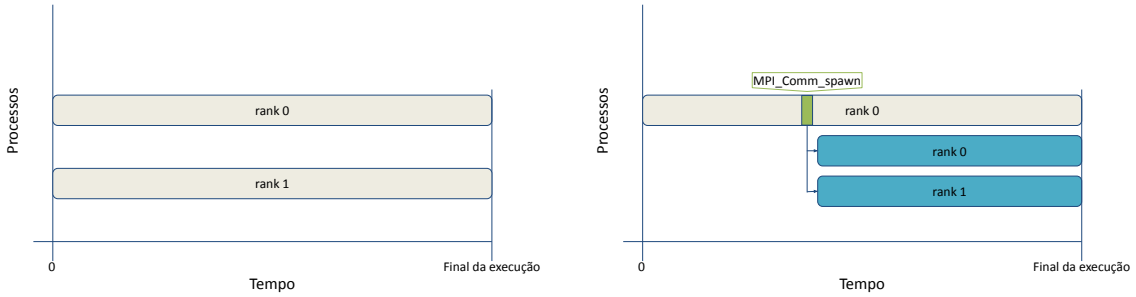
Para cada ponto da *grid* são calculadas as EDPs em um instante de tempo. Após resolver as equações para todos os pontos da *grid*, o tempo avança um passo de tempo. A computação continua até que o tempo total da simulação tenha transcorrido.

O Algoritmo 1 representa o funcionamento do método *Fletcher*. Na inicialização, o algoritmo recebe como parâmetros as dimensões da *grid 3D* (*x, y* e *z*), o tempo que passa a cada computação (*passo*) e o intervalo de tempo onde a simulação é realizada (*tempoTotal*). A *grid* é inicializada (*inicializaGrid*) e o seu estado inicial é armazenada (*gravaEmDisco*). A grid é enviada para GPU usando *enviaParaGPU*. O algoritmo entra no *loop for*, onde para cada instante de tempo da simulação será calculada a propagação da onda (*propagacaoOnda*) através da resolução das EDPs na *GPU*. Quando o tempo de simulação transcorrido atinge um valor limite (*limiteParaEscrita*, a grid que está na GPU é copiada de volta para memória principal usando *copiaParaMemoriaPrincipal*. Então o estado atual da *grid* é salvo em disco e o limiteParaEscrita é atualizado multiplicando o valor de *limiteInicial* (0,01) pela quantidade de escritas realizadas.

2.2. Message-Passing Interface

MPI é um padrão que define uma maneira padronizada de realizar troca de mensagens entre processos, o que facilita a escalabilidade de aplicações através da divisão de cargas de trabalhos entre diferentes processos, localmente, em um mesmo host, ou através da rede de forma distribuída. O padrão MPI, além de definir a comunicação, estabelece maneiras de criar processos e realizar acesso paralelo a arquivos. Todas as funcionalidades disponibilizadas pelo MPI são transparentes ao usuário, sendo a implementação do MPI a responsável por garantir o seu funcionamento correto.

Em MPI, processos podem ser criados de maneira estática ou dinâmica. A criação estática de processos é o método padrão de criar processos e acontece durante a inicialização da aplicação. Nessa abordagem, é informado ao *runtime MPI* a quantidade de processos que serão criados e o binário que será utilizado por esses processos. Todos os processos são criados simultaneamente e fazem parte de um mesmo grupo



(a) Criação estática de processos **(b) Criação dinâmica de processos**

Figura 1. Formas de criação de processos.

de comunicação. Cada processo recebe um identificador denominado de *rank*, que variam de 0 até o número de processos menos um. A Figura 1a ilustra o ciclo de vida de processos estáticos. Já a criação dinâmica de processos, introduzida com MPI-2 [Forum 2003], ocorre em tempo de execução, quando um processo MPI invoca a função MPI_Comm_spawn, especificando o número de processos e o binário que será utilizado. Esses processos são criados durante a execução da aplicação e pertencem a um novo grupo de comunicação, com *ranks* que começam em 0. A Figura 1b mostra o ciclo de vida de processos criados dinamicamente, onde os processos em azul são criados em tempo de execução. Todos os processos criados, estática e dinamicamente, permanecem ativos até o término da aplicação. O encerramento da aplicação acontece quando todos os processos chamam MPI_Finalize.

A troca de mensagens em MPI pode ser feita através de funções bloqueantes e não-bloqueantes. As funções bloqueantes são aquelas em que a execução do programa é suspensa até que a operação seja concluída. Já as funções não-bloqueantes tem sua operação executada em segundo plano, permitindo que o processo continue sua execução imediatamente após chamar a função.

O acesso a arquivos pode ser realizado por meio de funções MPI para esse fim. Esse conjunto de funções é chamado de MPI I/O [Corbett et al. 1996]. Essa funcionalidade permite que processos pertencentes a um mesmo grupo de comunicação acessem um mesmo arquivo paralelamente de forma organizada. As operações de leitura e escrita são coordenadas pelo runtime MPI que gerencia a maneira como serão executadas, garantindo consistência e otimizando a utilização de recursos. Algumas das funções *MPI I/O* são chamadas de funções coletivas. O seu funcionamento se diferencia por ser necessário que todos os processos de um grupo de comunicação a chamem. Ela atua também como uma barreira, ou seja, os processos que a chamam ficam bloqueados até que esta seja chamada por todos os membros do grupo. Os recursos de *MPI I/O* foram introduzidos com *MPI-2*.

3. Trabalhos relacionados

A latência de E/S representa um gargalo crítico em Computação de Alto Desempenho (HPC), impactando significativamente aplicações que lidam com grandes volumes de dados. Para mitigar esses efeitos, diversas abordagens têm sido exploradas. Por exemplo, [Chowdhury et al. 2023] propõe uma técnica de otimização de E/S onde pequenas operações de escrita são combinadas em escritas contíguas maiores, a fim de aumen-

tar a eficiência em sistemas de arquivos paralelos. Já, [Tang et al. 2022] introduz um *framework* de escrita assíncrona que utiliza threads em segundo plano para gerenciar as operações de E/S. As operações são organizadas utilizando um grafo acíclico para garantir a consistência das escritas. [Tsujita et al. 2014] propões uma implementação *multithread* de E/S coletivas em duas fases, onde na primeira fase os dados são reorganizados a fim de otimizar o acesso ao disco, e em um segundo momento, fase dois, esses dados são efetivamente escritos em disco. A implementação foi feita sobre *ROMIO* utilizando *pthreads*.

Além disso, [Patrick et al. 2008] investigou o impacto de sobrepor comunicação e computação, comunicação e escrita, escrita e computação e todas, simultaneamente, para duas aplicações: uma de multiplicação de matrizes e outra de leitura coletiva. Ambas distribuídas utilizando *MPI*. Verificando que o tamanho do *buffer* e o número de processos tem relação direta com o desempenho dessas aplicações. Em [Tipu et al. 2022] é explorada a utilização de rede neural artificial (ANN) para predizer a banda de E/S das operações de uma aplicação. Com base nessa rede neural são realizados ajustes nos parâmetros para otimizar a escrita em disco.

A abordagem que tivemos nesse trabalho se foca em sobrepor a escrita em disco com a computação do método *Fletcher*, utilizando processos *MPI* dedicados a escrita em disco, visando esconder a latência de E/S. O método *Fletcher* tem como característica a escrita de grandes arquivos em intervalos de tempo definidos.

4. Sobrepondo Computação com Operações de Entrada e Saída

O método *Fletcher* em sua versão Original, descrito na Seção 2, originalmente executa sequencialmente a computação e escrita dos dados em disco, conforme mostrado na Figura 2. Como a computação das propagações de onda é realizada em GPU, durante a computação a CPU fica ociosa. Portanto, para melhorar o desempenho da aplicação, neste trabalho, exploramos estratégias para separar o fluxo de escrita e computação em diferentes processos MPI, aproveitando a CPU, de modo que estas operações possam ser sobrepostas, conforme destacado na Figura 2. Deste modo, partindo da versão original do método *Fletcher*, propomos quatro versões alternativas, conforme discutido a seguir: *send_recv*, *isend_recv*, *spawn_singleFile* e *spawn_spliFiles*.

4.1. Versão MPI-1 Send_Recv

A versão *send_recv* utiliza dois processos MPI durante a execução: o processo de *rank 0* é responsável por gerenciar a fase de computação da aplicação, isto é, alocar as estruturas de dados e enviar e receber os dados computados na GPU. Já o *rank 1* é responsável pela fase de entrada e saída. Para esta versão, todos os processos são criados de modo estático, ao iniciar a aplicação, através do comando `mpirun -np 2 ./fletcher args`. Deste modo, o fluxo de execução desta versão é ilustrado na Figura 3a e descrito a seguir.

Enquanto o processo de *rank 0* gerencia a fase de computação, o *rank 1* inicia a execução e fica em um laço durante toda a execução. Neste laço de repetição, o processo aguarda o recebimento da *grid* atualizada do *rank 0* à ser escrita no disco. Esta operação de comunicação ocorre de maneira bloqueante através das operações `MPI_Send` (*rank 0*) e `MPI_Recv` (*rank 1*). Uma vez que a mensagem contendo os dados a serem escritos no disco é recebida, o *rank 1* realiza a escrita dos dados no disco, em um arquivo único, adicionando dados à cada operação, enquanto o *rank 0* realiza a fase de computação.

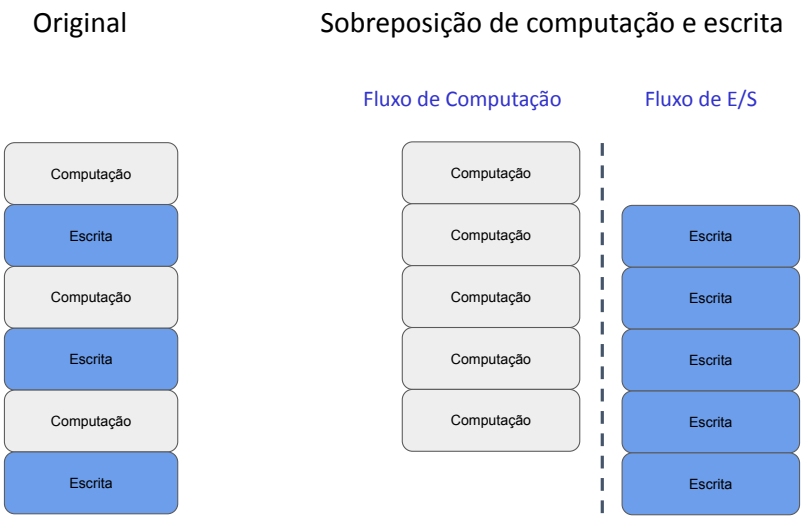


Figura 2. Fluxo original x sobreposto

(a) *send_recv*: comunicação bloqueante

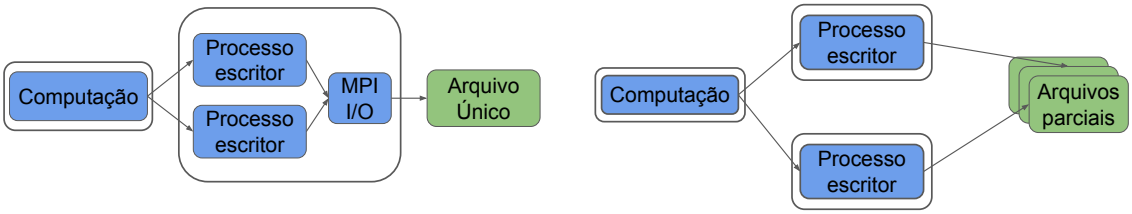
(b) *isend_recv*: comunicação não-bloqueante

Figura 3. Comparação entre comunicação bloqueante e não-bloqueante.

No momento em que a aplicação finaliza a propagação da onda, os dois processos são encerrados através da função *MPI_Finalize*. A Figura 3a ilustra a ideia geral desta versão.

4.1.1. Versão MPI-1 Isend_Recv

Nesta versão, também são criados dois processos de maneira estática durante a inicialização da aplicação, do mesmo modo que a versão *MPI-1 Send_Recv*. De modo similar, as responsabilidades de cada *rank* são iguais da versão anterior, com a diferença de que o envio da mensagem por parte do *rank 0* é realizada utilizando a função de envio não-bloqueante *MPI_Isend*. Nosso objetivo com esta versão é de que com o envio sendo realizado em *background*, o tempo percebido de enviar uma mensagem pelo *rank 0* seja menor que o das versões que utilizam comunicação bloqueante. O que diminuiria ainda mais o tempo entre as computações da simulação. Na Figura 3b é ilustrado o comportamento dessa versão.



(a) ***spawn_singleFile*: criação dinâmica de processos com escrita paralela**

(b) ***spawn_spliFiles*: criação dinâmica de processos com múltiplos arquivos**

Figura 4. Comparação entre diferentes abordagens de criação dinâmica de processos.

4.2. Versão MPI-2 Spawn_SingleFile

Esta versão faz uso da criação dinâmica de processos, disponível a partir da versão *MPI-2*. Assim, a versão denominada *Spawn_SingleFile* começa a execução com apenas um único processo, responsável por gerenciar a fase de computação da aplicação. Durante a execução, este processo cria dinamicamente processos responsáveis por realizar a fase de escrita. A criação de todos os processos é feita de uma única vez. Estes processos, definidos como um argumento para a aplicação, são chamados de *escritores* na Figura 4a (exemplo para 2 processos escritores).

Deste modo, o processo com *rank 0* envia o tamanho de *grid* para todos os processos criados. E passa a enviar, através de comunicação bloqueante (*Send_Recv*), o número da escrita e a *grid* alternando entre os processos criados dinamicamente conforme a política *round-robin*. Quando todos os dados foram escritos o processo com *rank 0* envia uma mensagem de encerramento para todos os processos escritores.

Os processo escritores, por sua vez, no início de sua execução chamam a função coletiva *MPI_File_open* que lhes dá acesso ao arquivo de saída. Os processos recebem o tamanho de *grid* e entram em *loop* onde aguardam o recebimento de uma mensagem de controle que define o número de escrita atual ou se o programa deve ser encerrado. No primeiro caso, o programa aguarda o envio da grid pelo processo com *rank 0*, e ao recebê-la chama a função *MPI_File_write_at* passando como parâmetro o deslocamento em relação ao início do arquivo. Esse valor é calculado multiplicando o número de escrita menos um pelo tamanho da *grid*. No caso de a mensagem indicar o encerramento da execução do processo, este sai do *loop* e chama a função coletiva *MPI_File_close*. As funções *MPI_File** fazem parte de *MPI I/O*.

4.3. Versão MPI-2 Spawn_SpliFiles

De modo similar à versão anterior, esta variante inicia a execução com apenas um único processo, responsável por gerenciar a fase da computação. No entanto, toda vez que existe uma escrita da *grid* em disco, este processo cria um novo processo escritor (via *MPI_Comm_Spawn*), que será responsável por realizar a escrita dos dados no disco. Como maneira a limitar o número total de processos filhos que serão criados durante toda a execução da aplicação, a quantidade máxima de processos escritores é definida via parâmetro na inicialização da aplicação. Para cada processo criado, lhe é enviado o

tamanho da *grid*. Após atingida a quantidade máxima de processos criados, o processo inicial alterna o envio de duas mensagens, uma com o número da escrita e outra com a *grid*, via estratégia *round-robin* entre eles utilizando comunicação bloqueante (*MPI_Send* e *MPI_Recv*). Ao final, o processo de *rank 0* envia uma mensagem de controle que indica o encerramento da aplicação. Os processos escritores operam em *loop*, similar ao descrito para versão *spawn_singleFile*, onde aguardam uma mensagem de controle que pode indica o número de escrita ou o encerramento do programa. Os processos escritores criam um novo arquivo para cada estado da *grid* que forem escrever em disco. Assim, ao fim da execução do programa temos um arquivo parcial para cada escrita realizada.

5. Análise experimental

5.1. Metodologia

Os testes foram realizados no Parque Computacional de Alto Desempenho (PCAD) do Instituto de Informática da UFRGS. Foi utilizado o nó computacional Blaise, que possui uma GPU P100 com 3584 CUDA cores, 2 processadores Intel Xeon E5-2699, 256 GB de RAM, um HD Seagate 7E2000 de 2,5TB com velocidade de escrita de 136 MB/s e um SSD Samsung EVO de 1TB com velocidade de escrita de 520m/s. A máquina roda o sistema operacional Debian 12, o sistema de arquivos utilizado nas partições do HD e SSD é ext4 e a implementação MPI é OpenMPI v3.1 [Open MPI 2018].

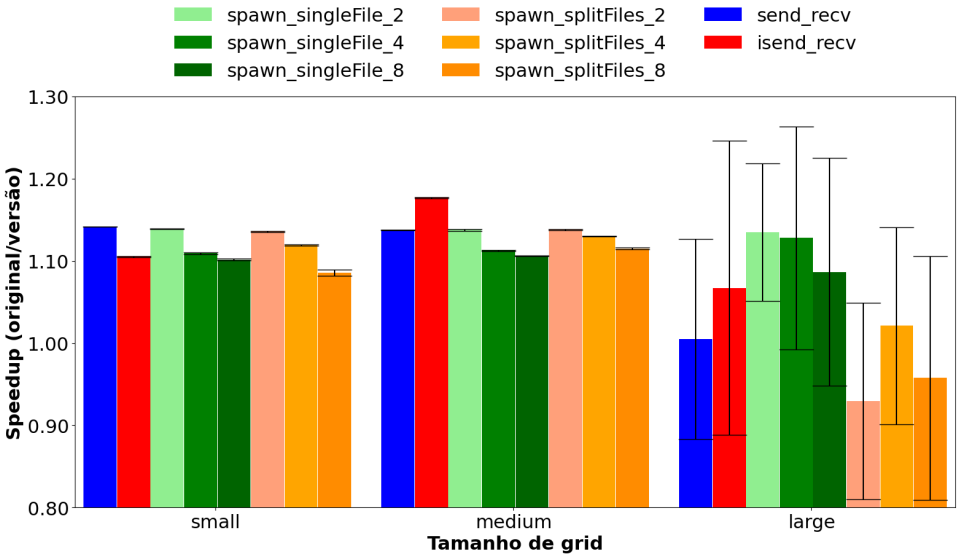
As versões *original*, *send_recv*, *isend_recv*, *spawn_singleFile* e *spawn_spliFiles* foram executadas 10 vezes para os tamanhos de *grid small* (280x280x280), *medium* (376x376x376) e *large* (472x472x472). O passo de tempo foi fixado em 0,001 e o tempo total em 1,5. Com esses valores, a propagação é computada 150 vezes para *grid*. Como o limite para escrita é 0,01, a *grid* é escrita em disco a cada 10 passos de tempo, totalizando 151 escritas, incluindo a escrita inicial. As versões *spawn_sigleFile* e *spawn_splitFiles*, recebem um parâmetro extra que define a quantidade de processos escritores. Para os testes o valor desse parâmetro foi variado entre 2, 4 e 8. Os experimentos foram realizados para *HD* e *SSD*.

Foram coletados o tempo total de execução e o tempo entre as rodadas de computação, que representam o tempo de envio das *grids* nas versões *MPI* e o tempo de escrita na versão original. Com base no tempo total foram calculados o tempo médio de execução e desvio padrão para todas as versões. Para as versões *MPI* foi calculado o *speedup* delas em relação à versão *original*, dividindo o tempo médio de execução da versão *original* pelo de cada versão.

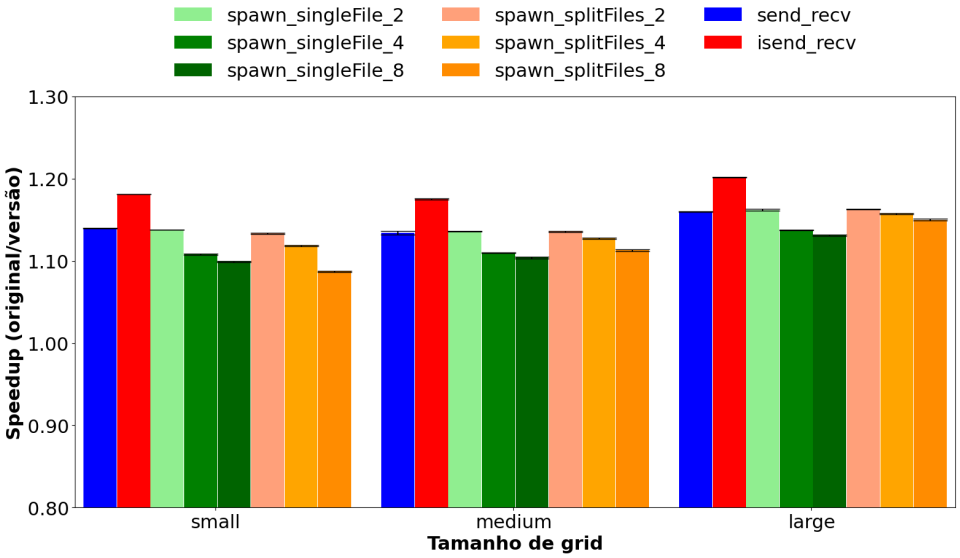
5.2. Resultados

Para melhor compreendermos a dimensão dos resultados é interessante saber o tempo médio da versão original. Para *grid small* o tempo médio foi de 83 segundos no *HD* e *SSD*, para a *grid medium* foi de 188 segundos o *HD* e *SSD*, já para a grid *large* o tempo no *HD* foi de 618 segundos e 423 segundos no *SSD*. A Imagem 5 apresenta os gráficos de *speedup* para as versões todas as versões propostas em relação à versão original. Quanto mais alta a barra, melhor o resultado. A barra de erro presente nas barras de *speedup* é referente ao desvio padrão. O Gráfico da figura 5a mostra os resultados referentes aos experimentos realizados com o uso de *HD*, e o Gráfico da figura 5b mostra os referentes as execuções em *SSD*.

Inicialmente analisaremos o desempenho das soluções para os resultados obtidos com *HD*. Para as duas maiores *grids*, quando comparamos somente a utilização de comunicação bloqueante (*send_recv*) e não-bloqueante (*isend_recv*), a segunda apresenta melhor desempenho, comprovando a intuição de que seu uso diminuiria o tempo entre computações. Para a *grid large* é possível notar que houve uma grande variação nos tempos de execução para todas as versões. Essa variação indica um esgotamento dos recursos utilizados por operações de E/S. O *HD* utilizado nos experimentos é compartilhado, com isso o espaço necessário para escrever arquivos maiores pode precisar ser fragmentado pelo disco. Além disso, não existe garantia de que para cada uma das execuções os mesmos blocos serão utilizados para escrita.



(a) Speedup para HD



(b) Speedup para SSD

Figura 5. Speedup obtido paras versões MPI do método Fletcher.

Devido à natureza mecânica do *HD*, a variação do local onde os dados serão escritos no disco pode acarretar maior ou menor variação da movimentação da cabeça de leitura e escrita, o que tem influência no tempo total de escrita. Sistemas operacionais implementam buffers para que o tempo de operações de E/S tenha menor impacto nas aplicações. Com o disco rígido sendo sobrecarregado, esses buffers tendem a ser esgotados, e com isso o tempo real de escrita passa a ser percebido pelos processos. Nessa *grid*, versão *spawn_singleFile*, para todas suas variações de número de processos, consegue manter um desempenho médio similar ao que vinha apresentando nas demais *grids*, enquanto o desempenho das demais versões decai.

Quando o disco fica sobrecarregado, as escritas passam a ser realizadas paralelamente, já que a escrita i+1 começa antes que a escrita i tenha sido concluída. Esse tipo de operação é otimizada pelo uso de funções de *MPI I/O*, presentes nesta versão.

Pela característica da versão *spawn_splitFiles* ter múltiplos processos que podem escrever em arquivos diferentes ao mesmo tempo, esperávamos que seu desempenho pudesse ser mais próximo da *spawn_singleFile*, para *grid large*, por escreverem em paralelo. Mas isso não se confirma, e ainda, essa versão apresenta os dois piores desempenhos para essa grid em suas variações com 2 e 8 processos escritores. Podemos pensar que dois processos que estão escrevendo em arquivos diferentes concorrem pelo mesmo recurso, o disco rígido. É possível que cada arquivo esteja sendo escrito em trilhas distantes entre elas. O acesso ao disco para escrita sendo alternado entre os processos aumentaria o tempo de busca da trilha, o que aumenta o tempo médio de escrita.

Todas as versões propostas tem bom desempenho para as execuções realizadas em *SSD*. As versões que utilizam comunicação bloqueante, tem desempenho similar. Aqui o desempenho médio da versão *isend_recv* é melhor do que os das demais versões para todos os tamanhos de *grid*. Quando a unidade de armazenamento opera abaixo do seu limite, o tempo de E/S percebido pelas soluções é similar. Nesse caso, o ganho de desempenho se deve ao fato de que a comunicação não-bloqueante utiliza menos tempo no envio de mensagens.

A versão *spawn_sigleFile*, que utiliza *MPI I/O*, mesmo nos piores casos apresentados, mantém o seu desempenho similar à versão mais simples que usa o mesmo tipo de comunicação. Isso indica que seu uso não adiciona *overhead* no tempo de execução. Porém, quando a escrita passa a ser feita paralelamente, o uso *MPI I/O* tem melhor desempenho que as demais versões. Nas versões que utilizam múltiplos processos, percebemos que o desempenho diminui levemente a medida que a quantidade de processos aumenta. Essa perda de desempenho se deve ao *overhead* que o *runtime MPI* adiciona no gerenciamento de processos. Em nenhum dos casos, a versão *spawn_splitFiles* apresenta melhor desempenho que as demais. Caso tivéssemos múltiplos discos para escrita, localmente, poderíamos determinar onde cada processo realizaria a escrita em disco. Isso permitiria uma maior escalabilidade da versão *spawn_splitFiles* e esconderia o *overhead* do gerenciamento de processos, visto que a escrita seria sobreposta entre os processos escritores.

6. Conclusão e Trabalhos Futuros

As soluções propostas se mostraram eficientes para a sobreposição de computação e escrita, trazendo um aumento no desempenho do método *Fletcher*. A versão *isend_recv* teve o melhor desempenho para HD e SSD dentre todas as versões. Para grid medium,

obteve seu melhor desempenho para HD com um speedup de 1,17x, no SSD 1,20x para grid large. O melhor desempenho médio no HD entre todas as grids foi o da versão *spawn_singleFile* com dois processo escritores, com speedup de 1,14x. O melhor desempenho médio entre as grids no SSD foi na versão *isend_recv*, com um speedup médio de 1,18x.

A escolha da tecnologia de unidade de armazenamento é fundamental, sendo o fator limitador para otimização do método *Fletcher*. Para esse tipo de aplicação, onde crescimento do tamanho de *grid* aumenta rapidamente o volume de escrita, *HDs* logo se tornam insuficientes.

A utilização de *SSD* consegue manter o desempenho para grids maiores, sendo mais adequado para maiores volumes de dados [Song and Lee 2013]. Porém, em algum ponto, com o aumento do tamanho de *grid* a tendência é que um único *SSD* acabe sendo levado ao limite, tornando-se novamente o gargalo da aplicação. A utilização de chamadas não-bloqueantes se mostrou uma melhor opção. Como esperado, o tempo de envio com comunicação não-bloqueante foi mascarado, diminuindo o impacto no desempenho da aplicação. A utilização de *MPI I/O* é uma boa forma de otimizar a escrita em disco, como visto nas execuções em *HD* onde ocorreu o acúmulo de escritas paralelas, essa funcionalidade melhorou o desempenho médio da aplicação. Para as execuções em *SSD*, não foi possível avaliar a utilização da função *MPI I/O*, já que não ocorreu o acumulo de escritas em paralelo.

Para futuros trabalhos a ideia é implementar uma versão distribuída do método *Fletcher*, para avaliar a escalabilidade do método e explorar técnicas de escrita em sistemas de arquivos distribuídos, como *Lustre*. Outra frente de trabalho seria com a utilização de bibliotecas otimizadas de *I/O* que permitem compactar e descompactar os dados de forma transparente na leitura e escrita, como *HDF5* e *ADIOS*, para avaliar a relação de tempo e razão de compressão utilizando técnicas *lossless*.

7. Agradecimentos

Este trabalho foi parcialmente apoiado pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior- Brasil (CAPES)- Código de Financiamento 001; pela Petrobras sob número 2020/00182-5 e pelo edital CNPq/MCTI/FNDCT- Universal 18/2021 sob número 406182/2021-3.

Referências

ANP (2023). Como funciona o processo de exploração e produção de petróleo e gás natural no brasil. Acesso em: 22 jul. 2024.

Chowdhury, M. K. H., Tang, H., Bez, J. L., Bangalore, P. V., and Byna, S. (2023). Efficient asynchronous i/o with request merging. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 628–636.

Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snirt, M., Traversat, B., and Wong, P. (1996). *Overview of the MPI-IO Parallel I/O Interface*, pages 127–146. Springer US, Boston, MA.

Fletcher, R. P., Du, X., and Fowler, P. J. (2009). Reverse time migration in tilted transversely isotropic (TTI) media. *Geophysics*, 74(6):179–187.

Forum, T. M. (2003). *MPI-2: Extensions to the MPI Standard*. Acesso em: 11 ago. 2024.

Open MPI (2018). Open MPI: Open Source High Performance Computing. Acessado em 16/02/2024.

Patrick, C. M., Son, S., and Kandemir, M. (2008). Comparative evaluation of overlap strategies with study of i/o overlap in mpi-io. *SIGOPS Oper. Syst. Rev.*, 42(6):43–49.

Song, H.-J. and Lee, Y.-H. (2013). A study on the disk performance comparison. *International Journal of Multimedia and Ubiquitous Engineering*, 8.

Tang, H., Koziol, Q., Ravi, J., and Byna, S. (2022). Transparent asynchronous parallel i/o using background threads. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):891–902.

Tipu, A. J. S., Conbhuí, P. , and Howley, E. (2022). Seismic data io and sorting optimization in hpc through anns prediction based auto-tuning for exseisdat. *Neural Computing and Applications*, 35:5855–5888.

Tsujita, Y., Yoshinaga, K., Hori, A., Sato, M., Namiki, M., and Ishikawa, Y. (2014). Multithreaded two-phase i/o: Improving collective mpi-io performance on a lustre file system. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 232–235.