FRAMEWORK DATAFLOW PARA COMPUTAÇÃO PARALELA DE ALTO DESEMPENHO APLICADO A SISTEMAS LINEARES ESPARSOS

Leonardo da Silva Gasparini

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Nelson Maculan Filho
Luiz Mariano Paes de Carvalho Filho

Rio de Janeiro
Abril de 2021

FRAMEWORK DATAFLOW PARA COMPUTAÇÃO PARALELA DE ALTO DESEMPENHO APLICADO A SISTEMAS LINEARES ESPARSOS

Leonardo da Silva Gasparini

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Nelson Maculan Filho
Luiz Mariano Paes de Carvalho Filho

Aprovada por: Prof. Nelson Maculan Filho
Prof. Luiz Mariano Paes de Carvalho Filho
Prof. Felipe Maia Galvão França
Prof. Jairo Panetta
Dr. José Roberto Pereira Rodrigues

RIO DE JANEIRO, RJ – BRASIL
ABRIL DE 2021

Gasparini, Leonardo da Silva

Framework Dataflow para Computação Paralela de Alto Desempenho aplicado a Sistemas Lineares Esparsos/Leonardo da Silva Gasparini. – Rio de Janeiro: UFRJ/COPPE, 2021.

XIX, 183 p.: il.; 29, 7cm.

Orientadores: Nelson Maculan Filho

Luiz Mariano Paes de Carvalho Filho

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2021.

Referências Bibliográficas: p. 148 – 152.

1. Dataflow. 2. Paralelismo. 3. Sistemas Lineares Esparsos. 4. HPC. 5. MPI. I. Maculan Filho, Nelson *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Primeiramente, agradeço à minha família, que sempre me apoiou nos estudos, e, em especial, minha esposa Jane, pelo apoio em um complicado momento de pandemia global (COVID-19).

Agradeço ao meu orientador Nelson Maculan, que me ajudou e inspirou desde a graduação. E ao meu orientador Luiz Mariano, pelos preciosos ensinamentos em álgebra linear, pela paciência e por ter proporcionado meu primeiro contato com o paradigma *dataflow*.

Agradeço especialmente ao José Roberto Rodrigues, grande idealizador e patrocinador do SolverBR, e que sempre incentivou e apoiou meu trabalhado.

Agradeço ao Prof. Jairo Panetta, pelos ensinamentos em computação paralela, e ao Prof. Felipe França, pelos ensinamentos em *dataflow*.

Agradeço a todos os colegas que atuam no desenvolvimento do SolverBR, por todas as dicas e contribuições. Ao Mateus Figueiredo, pelo apoio com o GeomecBR e com o Overleaf. E também ao Felipe Portella, pelas dicas e apoio técnico.

Por fim, embora eu não seja muito religioso, agradeço à Deus, pois nos momentos de aperto recorri muito a Ele.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

FRAMEWORK DATAFLOW PARA COMPUTAÇÃO PARALELA DE ALTO DESEMPENHO APLICADO A SISTEMAS LINEARES ESPARSOS

Leonardo da Silva Gasparini

Abril/2021

Orientadores: Nelson Maculan Filho
Luiz Mariano Paes de Carvalho Filho

Programa: Engenharia de Sistemas e Computação

As arquiteturas modernas de HPC vêm crescendo em escala e complexidade, tornando o desenvolvimento de algoritmos eficientes cada vez mais desafiador nessas arquiteturas. O paradigma *dataflow*, modelo de programação em que a computação é descrita como um grafo direcionado que representa o fluxo de dados (arestas) entre as tarefas a serem executadas (vértices), tem grande potencial para simplificar o desenvolvimento de aplicações paralelas. Esse modelo já foi amplamente estudado em arquiteturas de memória compartilhada, e, mais recentemente, vem suscitando estudos em arquiteturas distribuídas e heterogêneas. No entanto, ainda existem poucos trabalhos em HPC para álgebra linear esparsa envolvendo esse paradigma. Este trabalho apresenta um *framework dataflow* distribuído aplicado a núcleos de álgebra linear esparsa. O objetivo do *framework* é conciliar ganhos de produtividade na implementação de algoritmos paralelos com boa performance. Foram realizados testes extensivos, com até 2000 processos, comparando o desempenho de núcleos selecionados (incluindo um produto triplo de matrizes esparsas de larga escala) com o PETSc.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DATAFLOW FRAMEWORK FOR HIGH PERFORMANCE PARALLEL COMPUTING APPLIED TO SPARSE LINEAR SYSTEMS

Leonardo da Silva Gasparini

April/2021

Advisors: Nelson Maculan Filho
Luiz Mariano Paes de Carvalho Filho

Department: Systems Engineering and Computer Science

The modern HPC architectures are growing in scale and complexity, making the development of efficient algorithms increasingly challenging in these architectures. The dataflow paradigm, a programming model in which computing is described as a directed graph that represents the data flow (edges) between the tasks to be executed (vertices), has great potential to simplify the development of parallel applications. This model has already been extensively studied in shared memory architectures, and, more recently, it has given rise to studies in distributed and heterogeneous architectures. However, there are still few studies on HPC for sparse linear algebra involving this paradigm. This work presents a distributed dataflow framework applied to sparse linear algebra kernels. The objective of the framework is to reconcile productivity gains in the implementation of parallel algorithms with good performance. Extensive tests were carried out, with up to 2000 processes, comparing the performance of selected kernels (including a triple product of sparse matrices of large scale) with PETSc.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Importantes sistemas científicos de suporte a decisão da Petrobras, e da indústria do petróleo em geral, criam grande demanda por Computação de Alto Desempenho, ou *High Performence Computing* (HPC), para resolução eficiente de sistemas lineares e não lineares de larga escala. Ao mesmo tempo, as arquiteturas HPC vêm crescendo em escala e complexidade, tornando ainda mais desafiador o desenvolvimento de aplicações paralelas eficientes. O *Framework Dataflow*, tema desse estudo, faz parte dos esforços para facilitar o desenvolvimento de algoritmos paralelos, buscando conciliar maior produtividade com bom desempenho.

A principal contribuição desse trabalho é um *framework* de programação paralela baseado no modelo *dataflow* para o SolverBR, solver linear esparso de alta performance da Petrobras. Suas características mais relevantes são:

- Implementa um modelo *dataflow* para arquiteturas distribuídas, encapsulando mensagens MPI.
- Faz uso de programação orientada a objetos (POO) em C++ para a definição de tarefas e dados (usando o modelo de decomposição de domínios).
- Utiliza um grafo fisicamente distribuído e provê mecanismos para facilitar a definição de algoritmos como fluxo de dados nesse contexto.

O Capítulo 2 começa destacando a importância de HPC em sistemas científicos da área de reservatório. Em seguida, ele descreve o SolverBR e discute alguns conceitos do modelo *dataflow*. O Capítulo 3 descreve em detalhes o *framework* implementado. O Capítulo 4 traz extensivos testes de performance para núcleos de álgebra linear esparsa. O Capítulo 5 analisa os resultados, propõe futuras melhorias e apresenta a conclusão final. O Apêndice A lista os métodos abstratos das principais classes do *framework*. Por fim, o Apêndice B traz resultados de testes realizados no supercomputador Santos Dumont, que apresentaram certas inconsistências e ainda precisam ser validados.

Capítulo 2

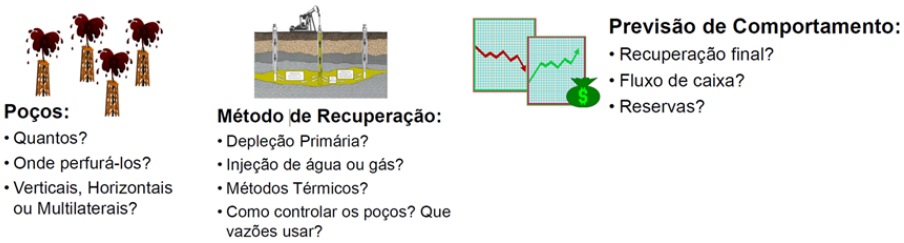
Motivação e Conceitos

2.1 Aplicações Científicas e HPC

Os Simuladores de Reservatório e os Simuladores de Geomecânica estão entre as aplicações científicas mais importantes das empresas de petróleo. Um dos maiores desafios para a produção comercial de óleo e gás consiste em conseguir simular com precisão o comportamento de um reservatório de petróleo para ajudar a traçar uma boa estratégia de produção, com poços injetores e produtores instalados em posições que maximizem a drenagem do óleo. A função de um Simulador de Reservatório é, justamente, prover estimativas futuras de um campo, dadas as condições iniciais [4, 5]. A Figura 2.1 ilustra algumas questões importantes para se delinear um bom plano de produção.

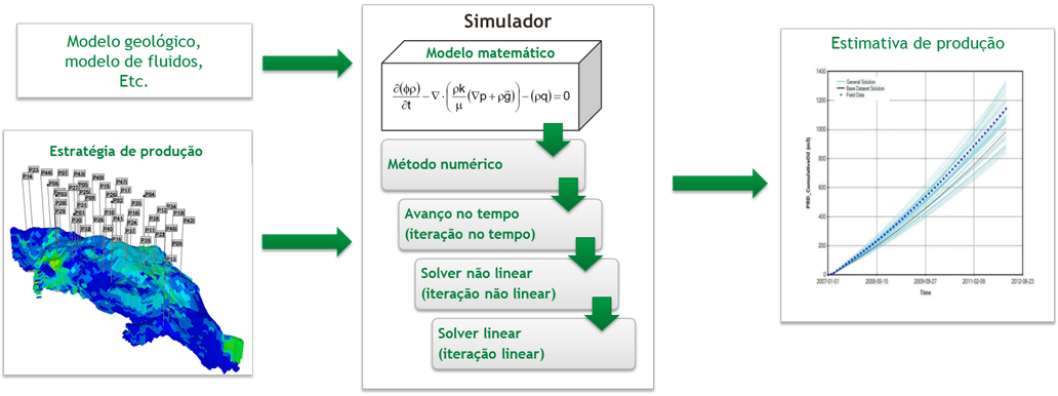
Um Simulador de Geomecânica fornece previsões de deformações e tensões em solos e rochas de um campo, ajudando, por exemplo, a estabelecer limites seguros de pressão nos poços injetores [6, 7]. Os estudos geomecânicos são complementares aos estudos de reservatórios.

Figura 2.1: Questões para um bom plano de produção



Em linhas gerais, esses simuladores empregam complexos modelos matemáticos, baseados em Equações Diferenciais Parciais (EDPs), e métodos numéricos de discretização, tais como: Elementos Finitos (FEM), Volumes Finitos (FVM) e Diferenças Finitas(FDM) [8]. A Figura 2.2 traz um esquema, em alto nível, de um Simulador de Reservatório típico.

Figura 2.2: Esquema geral de um simulador de reservatório típico



Uma das maiores dificuldades nos estudos de reservatório e de geomecânica é o tempo de simulação dos respectivos modelos. A qualidade das previsões depende de vários fatores, entre eles: o nível de detalhamento dos modelos matemáticos, a granularidade das discretizações dos métodos numéricos [5, 9], o nível de confiabilidade dos dados de entrada (que podem demandar múltiplas rodadas do simulador para análise de incertezas) [10–12], etc. A necessidade de se obter previsões cada vez mais confiáveis pressiona todos esses fatores. Como consequência, os simuladores precisam resolver sistemas (lineares e não lineares) cada vez maiores e mais complexos no menor tempo possível.

Por essas e outras razões, as empresas de petróleo, em geral, vêm investindo pesadamente em *High Performance Computing* (HPC) nos últimos anos. Para ilustrar isso, todos os 4 supercomputadores brasileiros na lista do Top 500 (DONGARRA *et al.* [13]) na presente data (Março de 2021) tiveram investimento da Petrobras. Além disso, a área de reservatórios da empresa tem um *cluster* próprio, o Guaricema, com 406 nós computacionais (mais detalhes na Seção 4.1.4).

Os investimentos em HPC não se resumem ao aumento da capacidade de processamento. Algumas iniciativas recentes de software HPC incluem: avaliação de novos simuladores comerciais, melhoria de simuladores internos (como o GeomecBR, FIGUEIREDO *et al.* [14]), e desenvolvimento de um solver linear de larga escala (o SolverBR, que será apresentado na próxima seção). O ***Framework Dataflow***, desenvolvido do âmbito do SolverBR, faz parte dos investimentos em software para computação de alto desempenho.

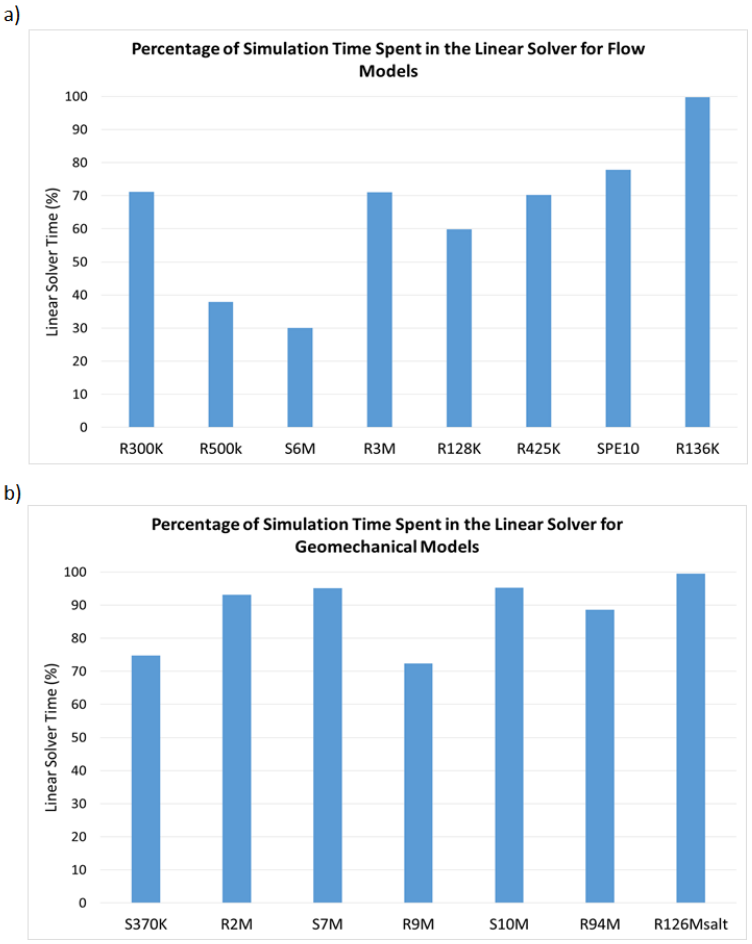
2.2 SolverBR

2.2.1 Introdução

O SolverBR (GASPARINI *et al.* [1] e [15]) é um solver linear paralelo híbrido para matrizes esparsas de larga escala. Ele foi projetado para ser acoplado a simuladores de interesse da Petrobras, com foco inicial em simuladores de reservatório e de geomecânica. Seu desenvolvimento é fruto de um esforço conjunto do CENPES (Centro de Pesquisas & Desenvolvimento Leopoldo Américo Miguez de Mello) com diversas instituições acadêmicas (UERJ, UFRJ, ITA, UFC, UFJF, Fiocruz e SENAI/CIMATEC).

A principal motivação do SolverBR é que, tipicamente, a resolução do sistema linear ocupa a maior parte do tempo de uma simulação [5, 9, 16, 17]. A Figura 2.3 traz alguns exemplos do percentual do solver linear no tempo total de diversas simulações de reservatório e de geomecânica (itens (a) e (b) da figura, respectivamente). Portanto, a performance da resolução de sistemas lineares tem grande impacto no tempo total desses simuladores.

Figura 2.3: Exemplos de percentual do solver linear no tempo total de diversas simulações (adaptado de GASPARINI *et al.* [1])



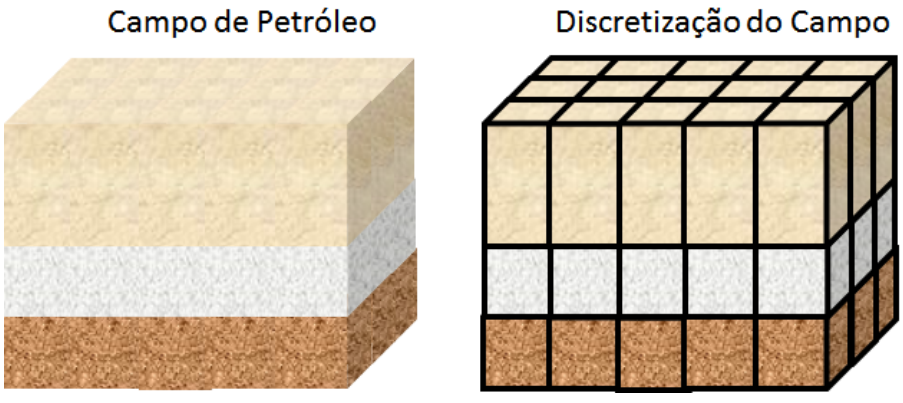
O *framework dataflow* desenvolvido, tema desse trabalho, faz parte do SolverBR, e foi projetado para facilitar a implementação de algoritmos paralelos de alta performance em arquiteturas distribuídas (MPI e, futuramente MPI+GPUs), com foco evidente em sistemas lineares esparsos. O GeomecBR, Simulador de Geomecânica da Petrobras, encontra-se acoplado (em ambiente de produção) ao SolverBR usando esse tipo de arquitetura. Por isso, todo o trabalho é baseado no GeomecBR. A próxima seção traz uma visão geral dos sistemas lineares desse simulador.

2.2.2 Matrizes do GeomecBR

As matrizes dos sistemas lineares do GeomecBR foram usadas como base em todos os algoritmos avaliados nesse trabalho. Por isso, essa seção é dedicada a detalhar sua estrutura e suas características, incluindo aspectos relacionados a paralelismo.

De um modo geral, um simulador de geomecânica precisa resolver EPDs (Equações Diferenciais Parciais) que modelam deformações e tensões em solos e rochas de um campo de petróleo e seu entorno. Para isso, eles empregam algum método de discretização. No caso do GeomecBR, utiliza-se o Método do Elementos Finitos (FEM) para linearizar as EDPs. Tal técnica, de maneira simplista, consiste em discretizar espacialmente o volume do campo a ser simulado em um *grid* 3D de elementos hexaédricos (pequenos volumes com 6 faces e 8 vértices), resultando em um stencil 27 pontos [18]. Por simplificação, vamos representar os elementos como paralelepípedos ou cubos. A Figura 2.4 ilustra um campo de petróleo à esquerda (as diferentes cores representam camadas de rochas distintas) e, à direita, um *grid* resultante de uma discretização espacial.

Figura 2.4: Exemplo simples de discretização espacial de um campo de petróleo

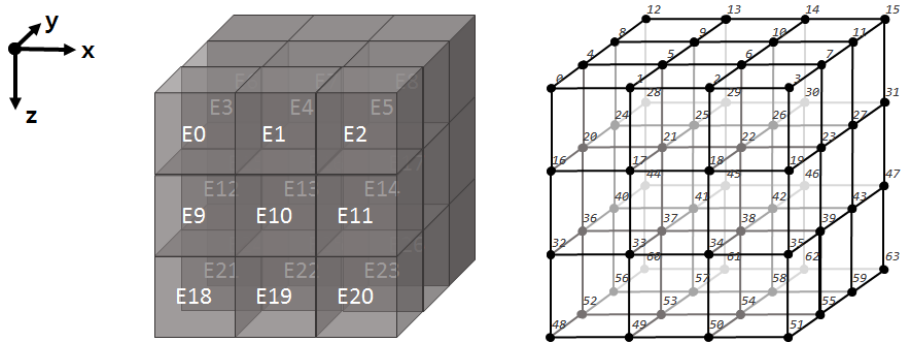


Um *grid* traz informações de **elementos** e de **nós**. Os elementos representam os volumes (paralelepípedos nos exemplos) e os nós são os vértices. A Figura 2.5 ilustra, à esquerda, os elementos e, à direita, os nós do *grid*. Ao se adotar um sistema de coordenadas, tanto os elementos quanto os nós precisam ser numerados. No exemplo

da figura, os elementos e nós foram numerados percorrendo-os nas direções $\boldsymbol{x}$, $\boldsymbol{y}$ e $\boldsymbol{z}$ (nessa ordem), conforme o sistema de coordenadas ilustrado.

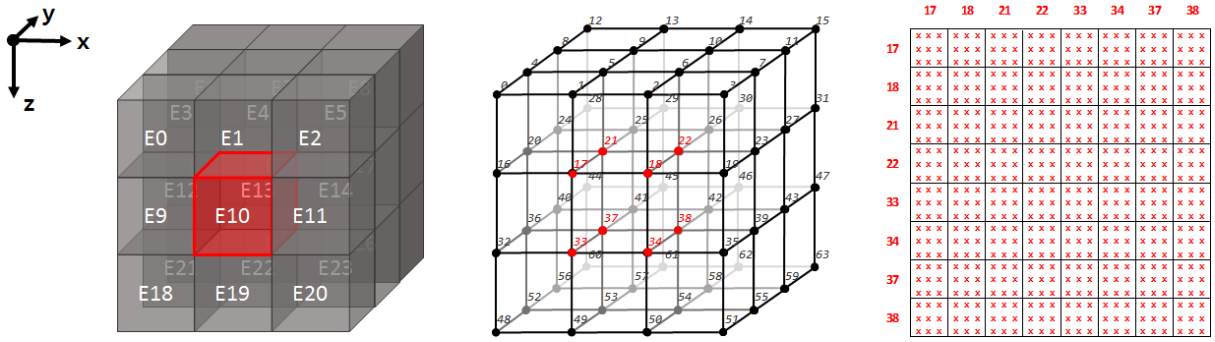
O número total de elementos é $NE = NE_x \times NE_y \times NE_z$ (número de elementos ao longo dos eixos $\boldsymbol{x}$, $\boldsymbol{y}$ e $\boldsymbol{z}$, respectivamente). Da mesma forma, o número total de nós é $NN = NN_x \times NN_y \times NN_z$. Note que $NN_* = NE_* + 1$ (para todos os eixos). No exemplo da figura 2.5, $NE = 3 \times 3 \times 3 = 27$ e $NN = 4 \times 4 \times 4 = 64$.

Figura 2.5: Representação dos elementos e dos nós de um *grid*



O Método dos Elementos Finitos usados no GeomecBR produz um sistema linear cujas equações e incógnitas se referem à malha de nós do *grid*. Portanto, a dimensão da matriz resultante será $NN \times NN$ (64×64, no exemplo), em termos de blocos. Isso porque as variáveis em cada nó possuem 3 graus de liberdade (*DOF - Degrees of Freedom*) resultando em pequenos tensores 3×3 **densos** para cada entrada da matriz [6], de modo que a dimensão total, em valores escalares, é $(NN \times DOF) \times (NN \times DOF) = (64 \times 3) \times (64 \times 3) = 192 \times 192$. Note que a dimensão total é grande mesmo para um *grid* bem pequeno. No entanto, a matriz final será **esparsa** (a maioria das suas entradas iguais a zero), como discutido a seguir.

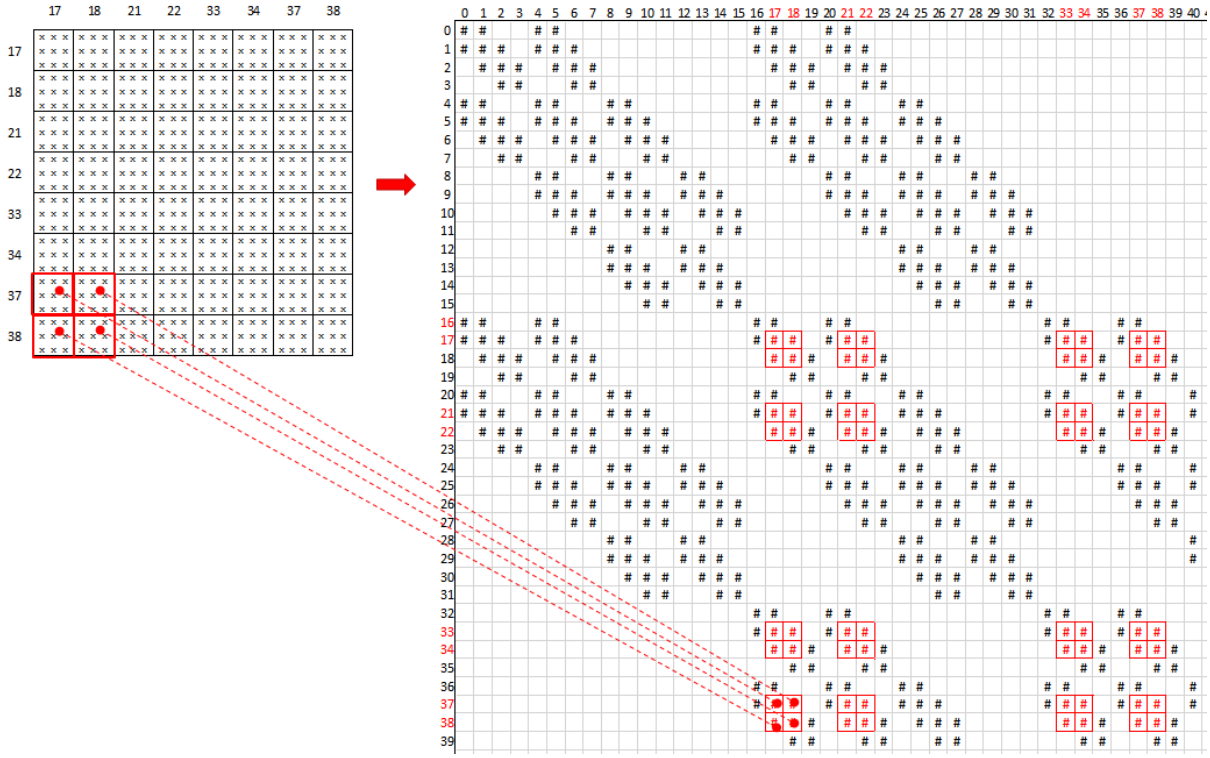
Figura 2.6: Exemplo de matriz de rigidez de um elemento



DE FIGUEIREDO [19] discute aspectos da modelagem de fenômenos geomecânicos. Colocando em termos simples (apenas para a compreensão da estrutura do sistema linear global), para cada elemento é gerada uma **matriz de rigidez local**, matriz densa de dimensão 8×8 (todos os elementos possuem exatamente 8 nós)

de blocos 3×3 densos. A matriz global é construída calculando-se a matriz de rigidez de cada elemento e acumulando os valores de cada entrada da matriz local nas respectivas posições da matriz global. A Figura 2.6 destaca o elemento 10 (à esquerda), os nós associados ao elemento (ao centro) e uma representação da matriz de rigidez local (à direita), onde cada quadrado pequeno representa um bloco 3×3.

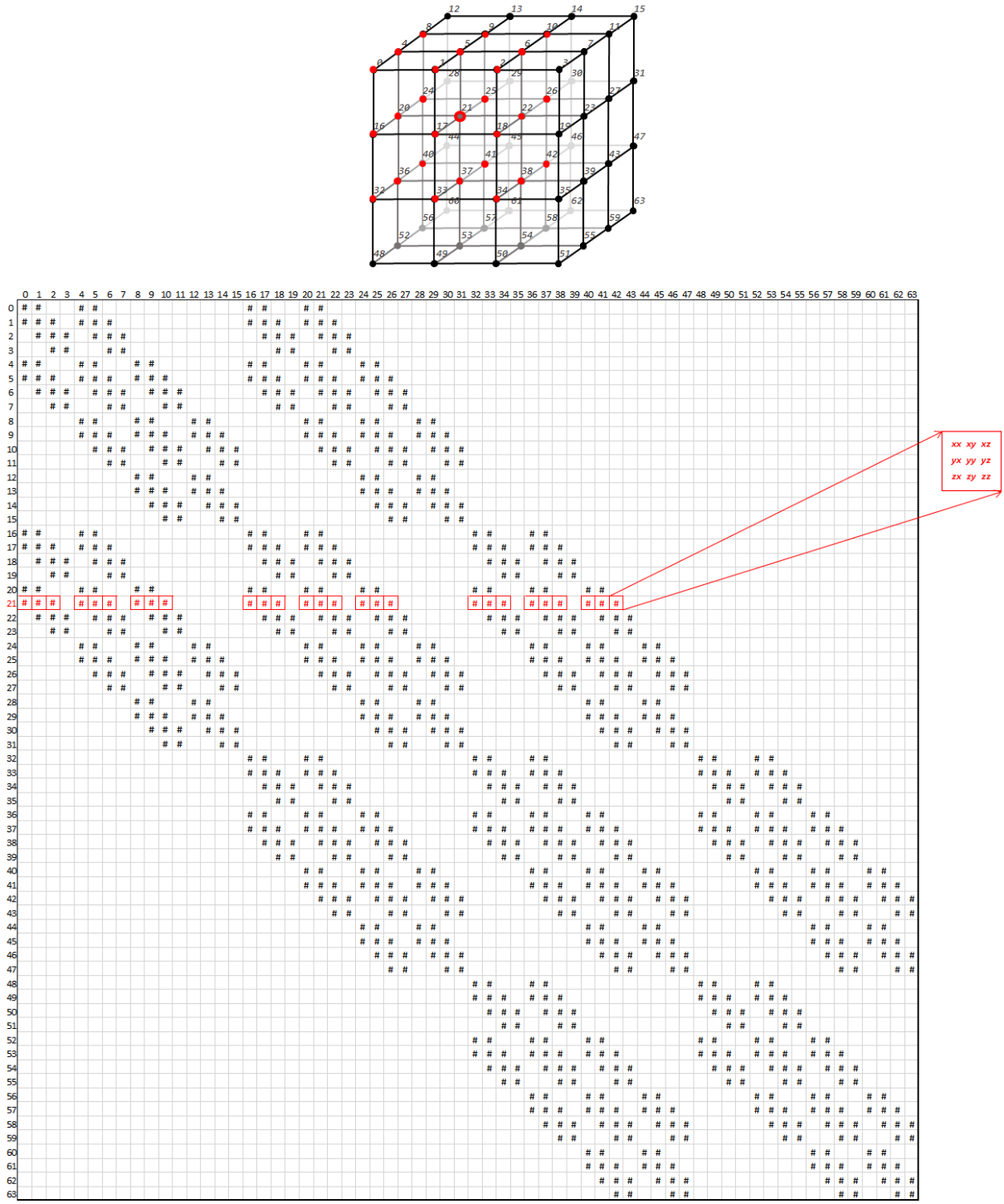
Figura 2.7: Ilustração de matriz de rigidez sendo adicionada à matriz principal



A Figura 2.7 ilustra a acumulação da matriz de rigidez do elemento 10 à matriz global. Ao final, a matriz resultante terá uma esparsidade tal como ilustrado na Figura 2.8. Uma característica importante dessa matriz é o ***stencil* de 27 pontos** [18]. No topo da Figura 2.8, o nó 21 foi destacado juntamente com outros 26 nós relacionados, o que implica que na linha 21 da matriz global, haverá $1 + 26 = 27$ blocos não nulos, como mostra a parte inferior da figura. Existem outras possibilidades de *stencil* (como o de 7 pontos, por exemplo), mas esse trabalho foca no de 27 pontos, que é usado GeomecBR.

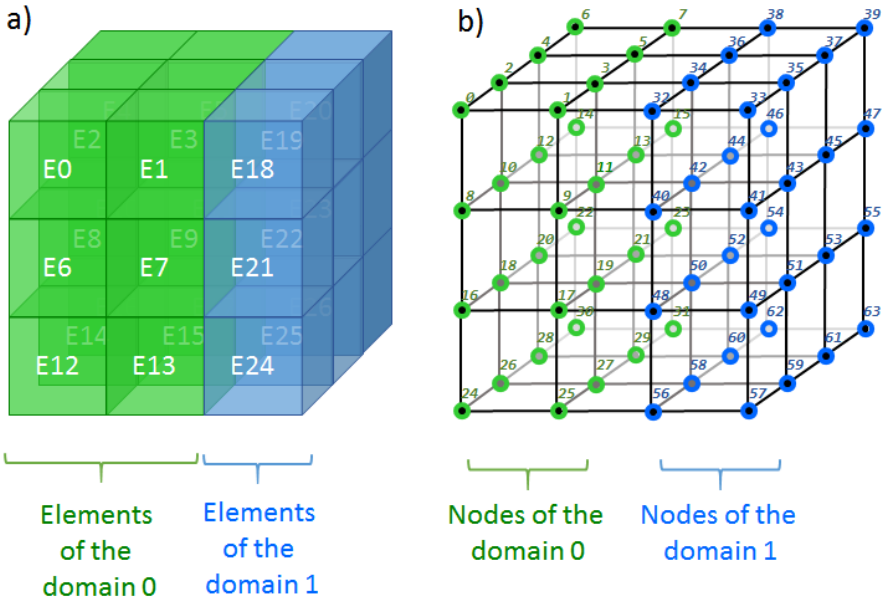
Para execução em paralelo, o *grid* precisa ser decomposto em domínios. No esquema adotado no GeomecBR, o *grid* de elementos é dividido ao longo dos eixos coordenados formando um *subgrid* para cada domínio. A Figura 2.9 ilustra o mesmo *grid* dos exemplos anteriores particionado em 2 domínios ao longo do eixo coordenado $\boldsymbol{x}$. Nesse caso, tanto os elementos quanto os nós são numerados por domínio. Ou seja, primeiro se numera os elementos/nós do *subgrid* do domínio 0, em seguida os do domínio 1, e assim por diante. A Figura 2.10 mostra a matriz final partici-

Figura 2.8: Padrão de esparsidade de uma matriz do GeomecBR



onada. No topo da figura, o mesmo nó 21 do exemplo anterior, agora renumerado para 11 devido ao particionamento, é destacado junto com os nós associados. Note que alguns desses nós associados pertencem a outro domínio. A matriz resultante está ilustrada na parte inferior da figura. A partição da matriz reflete a do *grid* de nós. A submatriz **diagonal** contém as equações de nós internos ao *subgrid* do domínio correspondente. As submatrizes fora da diagonal contêm equações de nós que são compartilhados por elementos pertencentes a domínios distintos. Por exemplo, o nó 32 é compartilhado entre os elementos 1 (pertencente ao domínio 0 do *grid*) e 18 (pertencente ao domínio 1 do *grid*), como mostra a Figura 2.9. Por isso, essas submatrizes são ditas de **conexão**.

Figura 2.9: Particionamento simples do *grid*



Uma característica importante das submatrizes de conexão é que, ao contrário das diagonais, elas normalmente possuem muitas linhas e colunas inteiramente nulas. No exemplo da Figura 2.10, metade das linhas e colunas são vazias. Por isso, as matrizes de conexão normalmente têm uma carga de dados muito menor do que as diagonais, e isso ainda pode variar com a fronteira entre os domínios do *grid*. A depender do particionamento, o próprio ***grid* de domínios** pode ser complexo. A Figura 2.11 mostra, no item (a), um *grid* particionado em domínios nos 3 eixos coordenados formando um *grid* de *subgrids* (domínios). Vamos assumir, por simplificação, que todos os domínios possuem a mesma dimensão ($NE = NE_x \times NE_y \times NE_z$). No item (b), os domínios 1 e 2 compartilham uma face, logo, o número de blocos não nulos na submatriz de conexão correspondente será proporcional ao número de nós associados a essa face. No item (c), os domínios 1 e 11 compartilham apenas uma aresta, logo, o número de blocos não nulos na submatriz de conexão correspondente

Figura 2.10: Padrão de esparsidade com particionamento simples

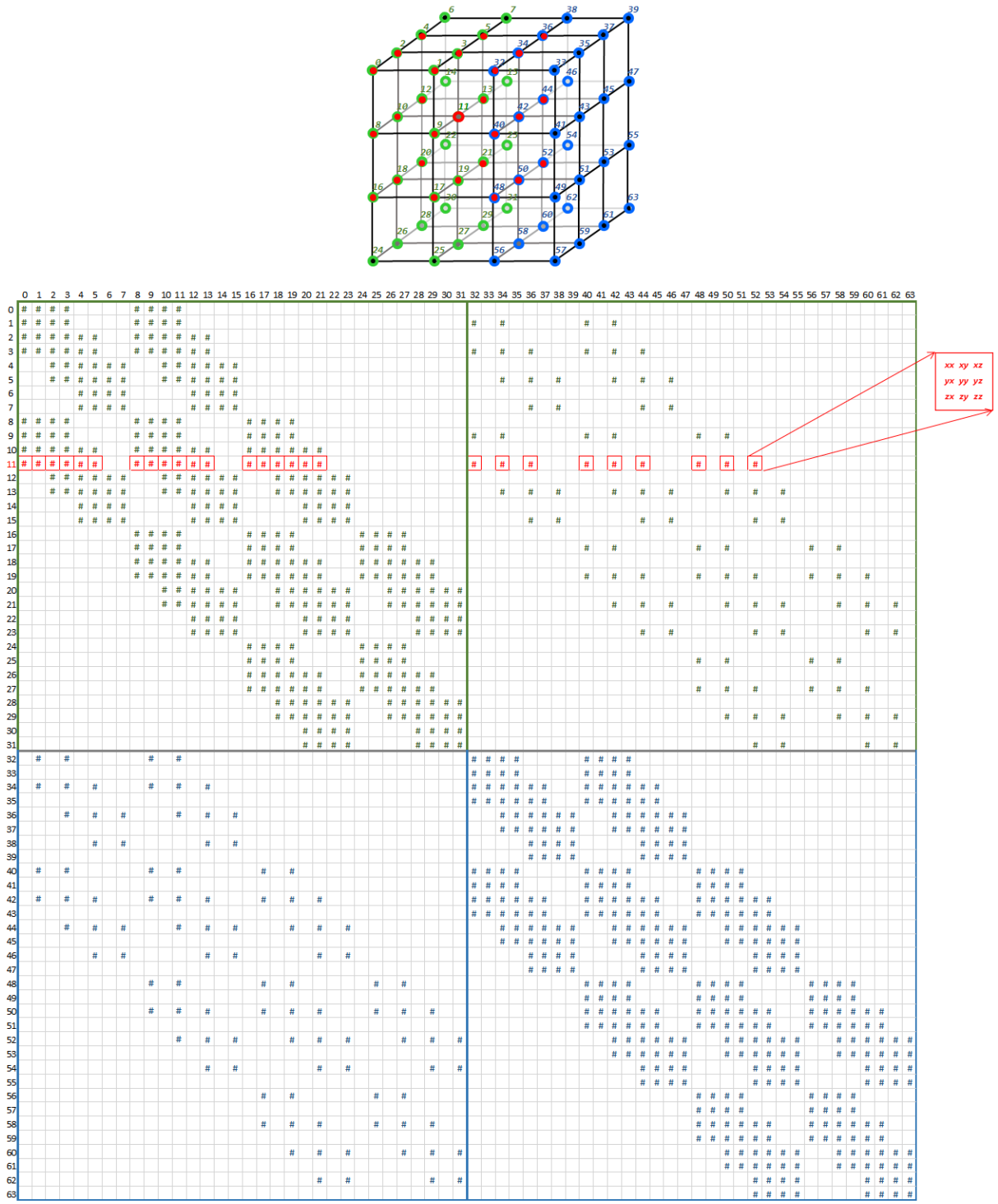
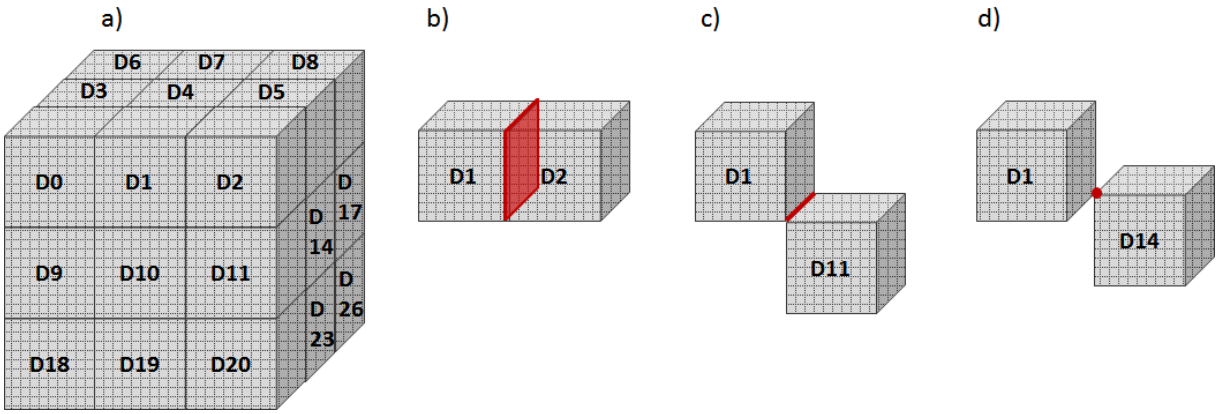


Figura 2.11: Ilustração de um grid particionado em 3 eixos coordenados, formando um *grid* de *subgrids* (domínios)



será proporcional ao número de nós associados essa aresta. Por fim, no item (d), os domínios 1 e 14 compartilham apenas um vértice, logo, o número de blocos não nulos na submatriz de conexão correspondente será proporcional ao número de nós associados esse único vértice. Resumidamente, a submatriz de conexão do item (d) é muito mais esparsa do que a do item (c), que por sua vez é muito mais esparsa do que a o item (b), que por sua vez é muito mais esparsa do que a submatriz diagonal de qualquer um dos domínios. Ou seja, a depender do esquema de particionamento, as submatrizes de conexão podem ter uma **carga de dados altamente heterogênea**.

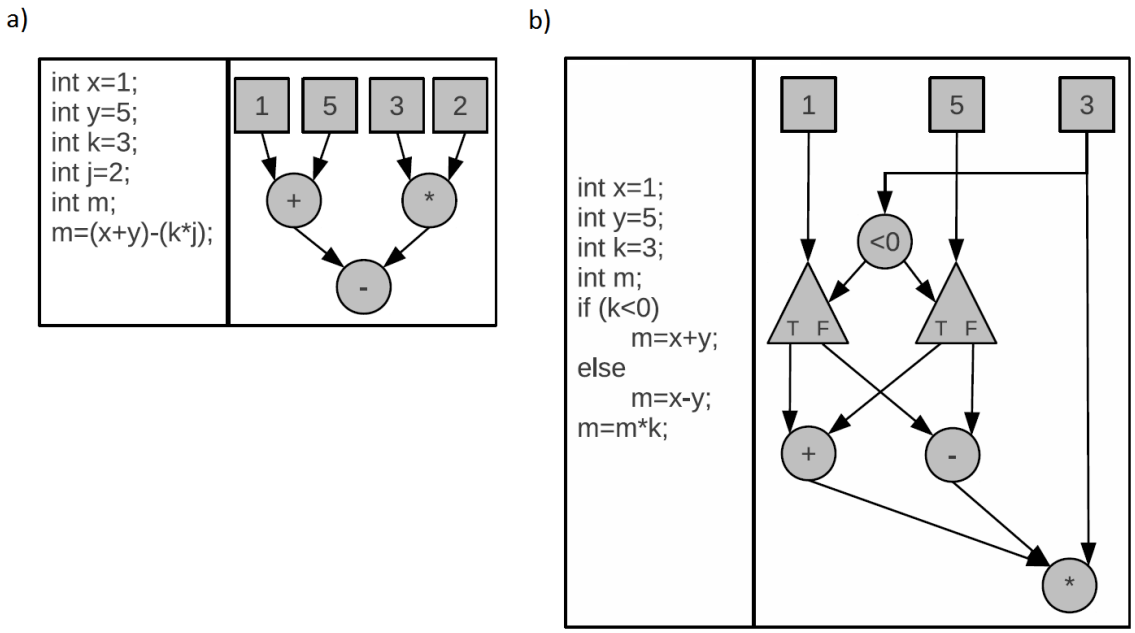
2.3 *Dataflow*

2.3.1 Modelo Dataflow

No modelo *dataflow*, a computação é descrita como um grafo direcionado que representa o fluxo de dados (arestas ou arcos) entre as operações a serem realizadas (vértices ou nós). Esse conceito começou a ganhar corpo no final da década de 60 (na área de arquitetura de computadores, como em RODRIGUEZ [20] e SUTHERLAND [21]), e, em meados da década de 70, surgiram as primeiras máquinas *data-flow* (DENNIS [22]), que disputavam espaço com as máquinas *control-flow* tradicionais (arquiteturas de Von Neumann).

A principal vantagem das máquinas *data-flow* era a facilidade em realizar processamento paralelo. As arestas que incidem em um nó do grafo são os operandos da tarefa associada. Basta que todos os operandos estejam disponíveis para que ela possa ser executada. Por outro lado, a principal dificuldade com essas arquiteturas era a complexidade das instruções de controle de fluxo (desvios, laços, etc.), muito mais naturais para as arquiteturas baseadas no modelo de Von Neumann.

Figura 2.12: Exemplos de instruções *dataflow* (adaptado de MARZULO [2])



A Figura 2.12, adaptada de MARZULO [2], mostra dois exemplos de códigos representados em grafos *dataflow*. O item (a) traz, à esquerda, um pequeno trecho de programa somente com operações aritméticas, e à direita o grafo dataflow correspondente (as operações são representadas por círculos). O item (b), traz um trecho de programa com instruções de controle (*if*/*else*) à esquerda, e o respectivo grafo à direita (as instruções de controle são representadas por triângulos).

Apesar do apelo à programação paralela, essas máquinas não tiveram o mesmo sucesso das *control-flow* (por diversas razões que extrapolam este trabalho), e, em termos de hardware, acabaram restritas a alguns nichos de HPC, como aplicações em máquinas FPGA (Field-Programmable Gate Array), por exemplo. Porém, o modelo *dataflow* como **paradigma de programação** passou a constituir um importante ramo da área de computação paralela.

No contexto geral de paralelismo por tarefas, o algoritmo é descrito como um grafo de tarefas (tipicamente um DAG - *Directed Acyclic Graph*), onde o vértice representa a tarefa e as arestas representam as dependências entre elas. Quando o grafo de dependências é construído a partir do fluxo de dados do algoritmo, o grafo de dependências é chamado de Grafo Dataflow. A vantagem dessa abordagem em processamento paralelo é a naturalidade com que a concorrência entre as tarefas é expressada. Todo o paralelismo pode ser gerenciado a partir do grafo, sem necessidade de barreiras (comuns em algoritmos paralelos com abordagem *control-flow*). Grafos Dataflow podem conter tarefas de controle de fluxo e também reentrâncias (MARZULO [2] e DE ARAÚJO [23]), mas esses esquemas sofisticados não estão no escopo desse trabalho.

2.3.2 Trabalhos Relacionados

GOLDSTEIN *et al.* [24] utiliza duas ferramentas *dataflow*, TALM [25] e Sucuri [26], em núcleos de álgebra linear no contexto de matrizes densas. Para matrizes esparsas, foco desse trabalho, ainda existem poucos trabalhos publicados. THOMAN *et al.* [3] analisou os *task based frameworks* mais usados em HPC, e propôs uma taxonomia que aborda diversos aspectos da API e do *runtime* de cada ferramenta. A Figura 2.13 traz a classificação de todas as ferramentas analisadas segundo os critérios do estudo. Olhando as duas primeiras colunas da tabela, a maioria dos *frameworks* que suporta memória distribuída utiliza modelo de comunicação *Global Address Space* (GAS). As únicas da lista que utilizam comunicação por mensagem (tal como o SolverBR) são o PaRSEC (BOSILCA *et al.* [27] e BOSILCA *et al.* [28]) e o StarPU (AUGONNET *et al.* [29]). Existem diversos trabalhos publicados sobre ambas as ferramentas no contexto de álgebra linear densa. No contexto de matrizes esparsas, em LACOSTE *et al.* [30], tanto o PaRSEC quanto o StarPU são utilizados em um solver direto (em CPU e em GPU), e, mais recentemente, em MO e LI [31], o PaRSEC é usado para resolver um sistema linear esparso iterativamente (com Gradiente Conjugado).

Figura 2.13: Comparação de características de APIs para paralelismo de tarefas (extraído de THOMAN *et al.* [3])

	Architectural			Task System				Management				Eng.	
	Communication Model	Distributed Memory	Heterogeneity	Graph Structure	Task Partitioning	Result Handling	Task Cancellation	Worker Management	Resilience Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
C++ STL	smem	×	×	dag	×	i/e	×	i	×	i/e	e	9	Library
TBB	smem	×	×	tree	×	i	✓	i	×	i	i	8	
HPX	gas	i	e	dag	✓	e	✓	i/e	×	i/e	e	6	
Legion	gas	i	e	tree	✓	e	×	i	×	i/e	e	4	
PaRSEC	msg	e	e	dag	×	e	✓	i	✓	i/e	i	4	
OpenMP	smem	×	i	dag	×	i	✓	e	×	i	i/e	9	Extension
Charm++	gas	i	e	dag	✓	i/e	×	i	✓	i/e	e	6	
OmpSs	smem	×	i	dag	×	i	×	i	✓	i	i/e	5	
AllScale	gas	i	i	dag	✓	i/e	×	i	✓	i	i/e	3	
StarPU	msg	e	e	dag	✓	i	×	i	×	i/e	e	5	
Cilk Plus	smem	×	×	tree	×	i	×	i	×	i	e	8	Lang.
Chapel	gas	i	i	dag	✓	i	×	i	×	i/e	e	5	
X10	gas	i	i	dag	✓	i	×	i	✓	i/e	e	5	

Segundo AGULLO *et al.* [32], o *Sequential Task Flow* (STF) é a estratégia mais comum para a paralelização de algoritmos *dataflow*. Ela consiste em executar o algoritmo simbolicamente (apenas criando as tarefas) de modo **sequencial** submetendo as tarefas encontradas ao *runtime*, que aplica regras de dependências de dados para gerar o DAG e gerenciar a execução. A maioria das ferramentas *dataflow* imple-

menta a estratégia STF, entre elas o OpenMP [33], o OmpSS [34], StarPU [29] e o PaRSEC. A simplicidade dessa abordagem simplifica o desenvolvimento de algoritmos paralelos, mas a escalabilidade tende a ser limitada pelo tamanho do DAG. A estratégia *Parameterized Task Graph* (PTG) [35] utiliza uma representação parametrizada das tarefas de modo a permitir a criação do DAG dinamicamente por partes (sem a necessidade de criar as tarefas sequencialmente), o que é particularmente útil para algoritmos distribuídos.

Além da estratégia STF, o PaRSEC também implementa o PTG por meio de uma linguagem específica, chamada *Job Data Flow* (JDF), que mapeia, entre outras coisas, todas as possíveis entradas e saídas de cada tarefa de forma parametrizada.

2.3.3 Contribuições

O *framework dataflow* foi implementado no âmbito do SolverBR (Seção 2.2) para arquiteturas distribuídas (com MPI). Para fins de escalabilidade, ele implementa um grafo fisicamente distribuído, em que cada processo cria apenas sua parte do DAG, na linha da estratégia PTG, porém com uma parametrização mais simples via Generators (Seção 3.2.4). Os Generators permitem a construção do grafo distribuído de forma quase tão simples quanto seria usando-se a estratégia STF, ao custo de comunicação entre os processos durante a criação dos DAGs locais.

Nessa versão inicial do *framework*, os DAGs locais são estáticos, ou seja, precisam ser construídos em uma fase prévia de *setup*. Isso traz limitações aos algoritmos que podem ser utilizados, mas, por outro lado, permite um gerenciamento amplo das tarefas locais de cada processo, visto que a ferramenta conhece todas as tarefas locais antes da execução. Além disso, o tempo da fase de *setup* é relativamente baixo (incluindo as comunicações), e pode ser diluído em múltiplas execuções do algoritmo.

O *framework* faz parte de uma biblioteca C++ (do SolverBR) e utiliza apenas programação orientada a objetos (POO) e *lambda expressions* C++ para especificar tarefas e fluxos de dados, sem necessidade de DLSs (Domain Specific Languages) ou extensões de linguagens. A ferramenta provê algumas classes abstratas para dados e tarefas que precisam ser realizadas pelo usuário para definição do algoritmo como *dataflow*. Isso torna o *framework* restrito a C++, mas dá grande flexibilidade ao usuário na definição dos algoritmos, pelo uso dos recursos de POO da linguagem.

Em álgebra linear esparsa, os domínios podem ter estruturas de dados bastante complexas. Pensando nisso, as tarefas do *framework* podem envolver múltiplos estágios assíncronos (Seção 3.3.1), o que facilita a movimentação desses dados (sobretudo quando há alguma dependência interna entre diferentes componentes da estrutura do dado).

Abaixo, segue a classificação da API do *framework dataflow* na taxonomia proposta por THOMAN *et al.* [3] (Figura 2.13).

- Architectural
  - Communication Model: message (troca de mensagens, via MPI)
  - Distributed Memory: explicit (troca de mensagens implica em distribuição explícita)
  - Heterogeneity: explicit (o artigo considera que todas as ferramentas com suporte a memória distribuída também suportam heterogeneidade)
- Task System
  - Graph Structure: dag (porém estático)
  - Task Partitioning: no (não há suporte para subtarefas)
  - Result Handling: implicit (o *framework* controla o ciclo de vida de todas as tarefas; não há suporte a *futures*)
  - Task Cancellation: no (será implementado num futuro próximo)
- Management
  - Worker Management: implicit (criação dos processos é implícita, são os processos MPI)
  - Resilience Management: no (não há suporte para *checkpoints* nem *restarts*)
  - Work Mapping: explicit (o mapeamento das unidades de processamento aos ranks MPI é explícita na ferramenta)
  - Synchronization: implicit (o *framework* controla as sincronizações implicitamente)
- Eng.
  - Technological Readiness: n/a (ferramenta nova)
  - Implementation Type: library

Capítulo 3

Framework Dataflow

Esse capítulo descreve a versão inicial do *Framework Dataflow* desenvolvido nesse trabalho.

3.1 Visão Geral do *Framework Dataflow*

3.1.1 Introdução

O *framework* foi desenvolvido em C++, no âmbito do SolverBR (2.2), e implementa um *dataflow* estático para algoritmos distribuídos, encapsulando mensagens MPI. Nessa versão inicial, ele emprega um *runtime* simples (*non threaded*), e todo o gerenciamento é feito entre execuções de tarefas do usuário em cada processo.

O grafo *dataflow* precisa ser construído em uma fase de *setup* para posterior execução. Embora seja um fator limitante, algoritmos complexos de álgebra linear **esparsa** costumam ter uma etapa simbólica e outra numérica. Nesses casos, o *setup* do grafo pode ser feito na etapa simbólica e a execução pode ocorrer seguidas vezes na etapa numérica.

Uma característica importante do *framework* é o grafo distribuído (de fato). O SolverBR replica poucos metadados entre os processos MPI. Por exemplo, na sua matriz distribuída, um *rank* não possui nenhuma informação sobre os domínios de *ranks* não acoplados na matriz (detalhes em [15]). Para usar um esquema tipo STF (Seção 2.3.2), seria preciso alterar a distribuição de metadados do solver, replicando em todos os processos a esparsidade completa de domínios da matriz. Isso motivou a tentativa de uma abordagem distribuída, mesmo em um contexto de grafo estático (construído na fase de *setup*).

3.1.2 Exemplo simples

Uso do *framework dataflow*

Em alto nível, para se executar um algoritmo paralelo, o *framework* recebe como entrada um conjunto de dados particionados em domínios e um conjunto de tarefas. Os dados são distribuídos entre os *ranks* MPI. Cada domínio só pertence a um *rank* (o seu *owner rank*), e é nele que os dados estão fisicamente alocados. Tomemos como exemplo uma versão simplificada do algoritmo de multiplicação entre matriz esparsa e vetor. A Figura 3.1 mostra um exemplo dessa operação com 3 *ranks* MPI. O item (a) da figura mostra como os domínios da matriz e do vetor estão distribuídos. Estão destacados em amarelo os domínios que pertencem a cada um dos processos MPI. O item (b) da figura ilustra as operações que precisam ser executadas em cada *rank* e os domínios envolvidos. Assumimos, por simplificação, que o objeto y é inicializado com zero em um pré-processamento. Note que algumas operações necessitam de domínios remotos como entrada, como o domínio x_1 no processo 0, os domínios x_0 e x_2 no processo 1 e o domínio x_1 no processo 2.

Figura 3.1: Exemplo simplificado de multiplicação matriz vetor com 3 *ranks* MPI

Descrição do algoritmo como fluxo de dados entre tarefas

Diferentemente de muitas ferramentas *dataflow*, que fazem uso de DSLs (*Domain Specific Languages*) ou de outros mecanismos externos (como extensões de linguagens), o *framework* proposto funciona inteiramente em ambiente C++ (da versão 11 em diante), e seu uso é todo baseado em orientação a objetos. A principal classe é o CoarseDataflow, que permite definir e executar um algoritmo paralelo baseado no fluxo de dados estabelecido entre as tarefas. Essa classe encapsula um *dataflow* estático, cujo o grafo é integralmente montado no construtor com auxílio de uma *lambda*

expression que tem como parâmetro uma referência para CoarseDataflow::Graph (classe que possui os métodos para descrição do algoritmo como fluxo de dados). O Código 3.1 traz um exemplo para a multiplicação matriz vetor. A primeira parte da definição é o registro dos objetos e *closures* que irão participar da computação, o que deve ser feito de forma coletiva por todos os *ranks*. Os objetos são dados distribuídos decompostos em domínios, e as *closures* são classes que representam tarefas entre domínios. Ambas serão discutidas na próxima subseção. Nas linhas 4, 5 e 6 do Código 3.1, são registrados os objetos A (matriz), x e y (vetores). O método *object(DataObject*)* retorna um *handler* (que é apenas um número) que passa a representar o dado no grafo. Da mesma forma, na linha 9, o método *closure(Closure*)* registra uma *closure* retornando um *handler* numérico.

A segunda parte é a definição das tarefas a serem executadas no ***rank* local**, o que é feito chamando-se uma sequência de métodos *task(taskHandler,ids)* da classe CoarseDataflow::Graph que alteram dados do *rank* local. Esse método retorna um objeto que representa uma operação entre domínios, e é dotado de funções que especificam as entradas e saídas da operação. As linhas 17 e 18 mostram o uso da função *in(objHandler,ids,inState)* e na linha 19 a função *out(objHandler,ids,outStatate)*, o que significa que a tarefa yAccAx(i,j) tem como entradas os domínios $A(i,j)^0$ e $x(j)^0$ e como saída o domínio $y(i)^s$ (os índices sobrescritos representam estados, e serão explicados a mais adiante). Essa classe faz uso de um padrão de programação chamado *Expression Builder*, que permite encadear várias chamadas em sequência, sem usar o ponto e vírgula entre elas, e sem precisar declarar a variável de retorno de *task* (FOWLER [36]).

Código 3.1: Exemplo de definição de multiplicação matriz vetor no *framework dataflow*

```
1  CoarseDataflow dflowMatVec( procEnv, "Simple MatVec", [&]( CoarseDataflow::Graph& gph )
2     { // <-- Corpo do lambda expression
3        // Registra os objetos
4        auto A = gph.object( &objA );
5        auto x = gph.object( &objX );
6        auto y = gph.object( &objY );
7
8        // Registra a Closure. Representa y(i) += A(i,j)*x(j)
9        auto yAccAx = gph.closure( &tskMatVec );
10
11       // Define o algoritmo
12       int s = 1;
13       const int i = rank;
14       for ( int j : matA.domainColumns[ i ] )
15       {
16          gph.task( yAccAx, { i, j } )
17             .in(  A, { i, j }, 0 )
18             .in(  x, { j }, 0 )
19             .out( y, { i }, s );
20          s++;
21       }
22    } // <-- Final do lambda expression
23 ); // <-- Final da chamada do construtor de dflowMatVec
24
25 // Executa o algoritmo distribuido
26 dflowMatVec.execute();
```

O construtor de *dflowMatVec* processa o *lambda function* (JÄRVI e FREEMAN [37]) e gera um Grafo de Execução que representa as tarefas e suas dependências, respeitando o fluxo de dados. Ao final do construtor, o dataflow está pronto. A linha 25 do código exemplifica a execução da multiplicação matriz-vetor. Nesse ponto, o *framework* inicia, assincronamente, a execução das tarefas disponíveis (sem dependências). À medida que as tarefas vão sendo concluídas, novas tarefas se tornam disponíveis. A Figura 3.2 mostra os grafos de execução locais do *framework* em cada *rank* MPI gerados a partir das definições das tarefas no construtor do objeto *dflowMatVec* do código anterior. As tarefas com fundo branco são tarefas que não dependem de nenhuma outra, e já estão disponíveis para serem iniciadas logo no início da execução. As tarefas com fundo escuro vão se tornando disponíveis à medida em que todas as tarefas das quais elas dependem são concluídas.

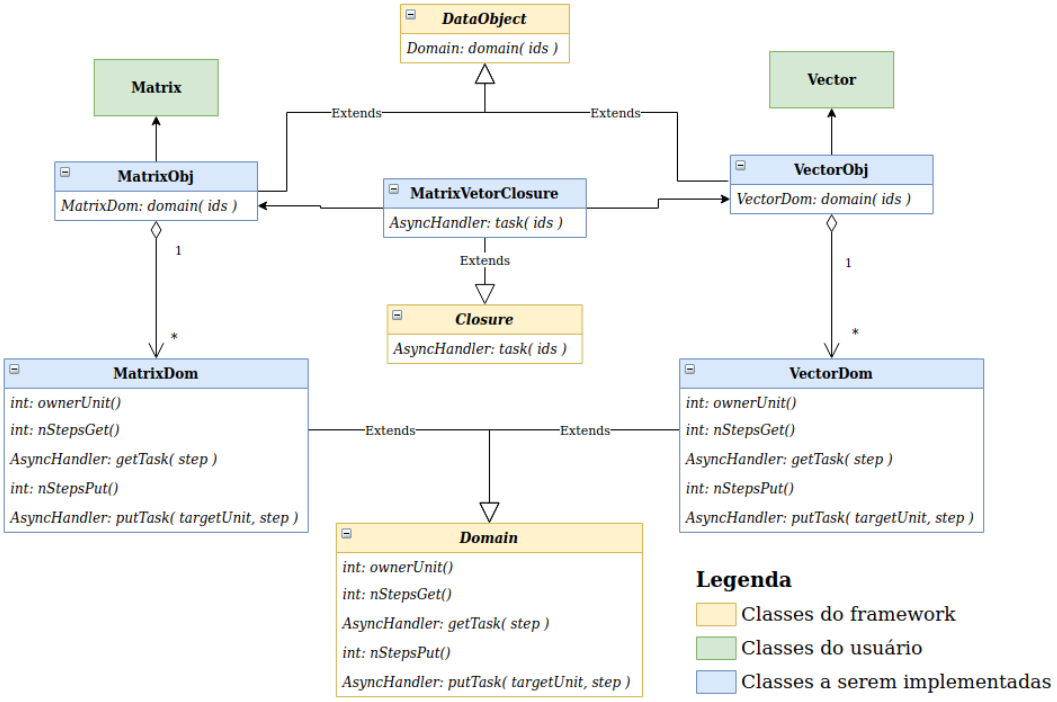
Figura 3.2: Grafo de execução local de cada *rank* MPI gerado pelo framework para o código 3.1

Definição dos objetos, domínios e *closures*

Seguindo com o exemplo, o *framework* provê as classes abstratas DataObject, Domain e Closure, que precisam ser **realizadas** (provendo implementações concretas para os métodos abstratos) pelo usuário para seus dados distribuídos, para os domínios de dados distribuídos e para as tarefas a serem executadas sobre os domínios, respectivamente.

O diagrama da Figura 3.3 traz um exemplo das classes envolvidas na multiplicação matriz-vetor. As classes em azul precisam ser implementadas pelo usuário. A MatrixObj e a VectorObj representam possíveis realizações de DataObject para matriz e para vetor, respectivamente. As classes MatrixDom e VectorDom representam possíveis realizações de domínios de matriz e de vetor. E a classe MatrixVectorClosure exemplifica uma realização da classe Closure que efetua uma multiplicação entre domínios de matrizes e de vetores. No Código 3.1, a variável objA é uma instância de MatrixObj, enquanto as variáveis objX e objY são instâncias de VectorObj. Da mesma forma, a variável tskMatVec é uma instância de MatrixVectorClosure.

Figura 3.3: Diagrama de classe para o exemplo simplificado de multiplicação matriz vetor



O principal método de DataObject que precisa ser implementado na classe do usuário é o *domain(ids)*, que retorna um domínio a partir de uma n-upla de índices (onde n depende do dado e é configurável no construtor da classe). Os principais métodos abstratos da classe Domain são o *getTask(step)* e o

putTask(targetUnit, step), que representam tarefas assíncronas de envio e de recebimento dos dados do domínio. E o principal método da classe Closure a ser implementado, é *task(ids)*, que efetua uma operação entre domínios. A n-upla de índices recebida pela tarefa normalmente contém a união dos índices dos domínios operados. A Seção 3.3 descreve o funcionamento das tarefas.

Colocando em termos simples, na realização de *Domain*, o usuário usa rotinas MPI não-bloqueantes para enviar os dados do domínio a um *rank* remoto identificado no parâmetro *targetUnit* do método *putTask*. No método *getTask*, ele usa rotinas MPI não-bloqueantes para receber dados do *owner rank* do domínio. E no método *task* da realização da Closure, a operação entre os domínios é de fato efetuada. Se um dos operandos for remoto, a rotina *task* só será acionada pelo *framework* quando o dado estiver disponível (após conclusão de *getTask*). Importante: o framework não esconde as rotinas MPI. Em geral, implementar as comunicações ponto a ponto é a parte fácil da implementação de algoritmos paralelos por troca de mensagens. O difícil é gerenciar as comunicações com bom desempenho, e é isso que a ferramenta se propõe a fazer. Além disso, usando diretamente as rotinas MPI, o usuário pode fazer uso de *features* avançadas da API, como *custom datatypes*, por exemplo.

3.2 *Setup* do *Framework Dataflow*

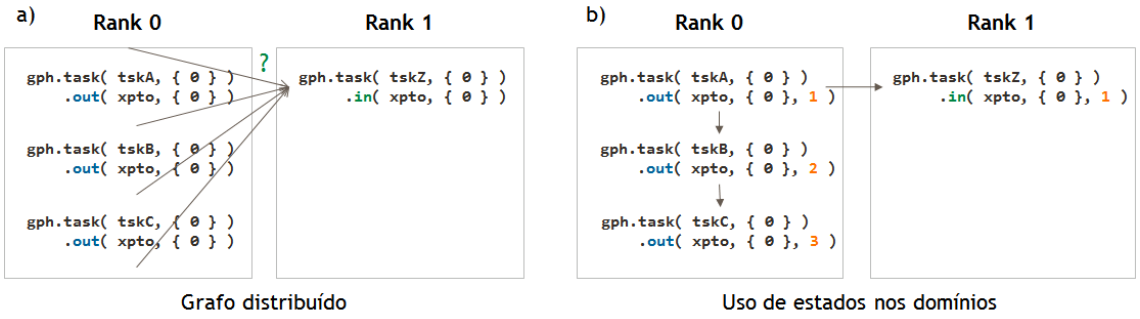
Essa seção descreve a fase de *setup* do *dataflow*. Ela detalha o que acontece no construtor de uma instância de CoarseDataflow, que recebe um *lambda expression C++* (JÄRVI e FREEMAN [37]) onde um algoritmo é definido em termos de tarefas e seus respectivos domínios de entrada e de saída. Ao final dessa fase, o *framework* possui, em cada *rank* MPI, um Grafo de Execução com as tarefas locais àquele *rank*, incluindo tarefas de computação e tarefas especiais de comunicação.

3.2.1 Definindo o fluxo de dados em ambiente distribuído

Na maioria das ferramentas *dataflow* que usam o esquema de *input* e *output*, a sequência de definição das tarefas ajuda a estabelecer as dependências. Por exemplo, se várias tarefas escrevem num mesmo domínio, a ordem em que elas aparecem gera uma cadeia de dependências (cada uma depende da anterior). Esse mecanismo é funcional num esquema STF (Seção 2.3.2), mas no contexto de grafo distribuído, não há como estabelecer ordem de definição entre tarefas de *ranks* MPI distintos. O item (a) da Figura 3.4 ilustra o problema. No *rank 0*, três tarefas escrevem no domínio *xpto(0)*. No *rank 1*, o *xpto(0)* é usado como *input* em *tskZ*. Note que não é possível saber de qual "versão"desse domínio a tarefa *tskZ* depende. Poderia ser a versão inicial (antes de qualquer alteração), poderia ser a versão gerada por *tskA*,

ou por *tskB* ou por *tskC*.

Figura 3.4: Exemplificando a dificuldade em se estabelecer dependências com base na ordem de definição de tarefas em contexto distribuído



No contexto de grafo distribuído do *framework dataflow* implementado, onde cada *rank* MPI define suas tarefas independente dos demais, faz-se necessário uma informação adicional para estabelecer corretamente as dependências. Por isso, além do *handler* do objeto e dos índices que identificam o domínio, os métodos de entrada e de saída das tarefas também requerem um valor inteiro que representa os diferentes estados do domínio durante a computação. O estado 0 é sempre o estado inicial, e sempre que uma tarefa altera um dado, o valor do novo estado deve ser maior que o anterior. Quando uma determinada tarefa tem como *input* um determinado domínio em um estado maior que 0, então, no *owner rank* dele precisa existir uma tarefa com um *output* para o mesmo domínio com o mesmo estado. Dessa forma, o *framework* é capaz de fazer toda a amarração entre as operações. Obviamente, duas ou mais tarefas não podem escrever num mesmo domínio com o mesmo estado, exceto quando usado a função *reduce*, que será discutida mais adiante. O item (b) da Figura 3.4 mostra o uso de estados ou versões nos *inputs* e *outputs* para remover as ambiguidades e permitir a correta inferência da cadeia de dependências. Observe também, na linha 19 do código anterior, que todas as tarefas *yAccAx(i,j)* escrevem no domínio *y(i)*, porém cada uma o faz com um estado diferente (o *s* é incrementado após a definição de cada tarefa, na linha 20).

O *framework* provê diversas formas de especificar as entradas e saídas de uma tarefa. A função *in(objHandler,ids,inState)* serve para definir uma entrada, e recebe o *handler* do objeto, os índices do domínio e o estado necessário. A função *out(objHandler,ids,outState)* define um domínio de saída, recendo o *handler* do objeto, os índices do domínio e o novo estado. A função *inout(objHandler,ids,inState,outState)* informa que a tarefa recebe um domínio em um estado *inState* e o altera para o estado *outState*. Por fim, a função *reduce(objHandler,ids,inState,outState)* pode ser usada por várias tarefas com o mesmo domínio e com os mesmos estados iniciais e finais. Nesse caso, o estado só será atualizado de fato para o estado final quando todas as tarefas forem con-

cluídas durante a execução. O *reduce* não induz nenhuma ordem entre as tarefas da redução (aquela que estiver disponível primeiro, é executada primeiro). O que essa função garante é que o estado do domínio é *inState* antes da execução da primeira tarefa, e que ao término da última tarefa o estado passará a ser *outState*.

Qualquer domínio de qualquer processo MPI pode figurar como *input* de uma tarefa (a ferramenta se encarregará de fazer o link entre os *ranks*), mas apenas domínios locais podem ser usados como *output*. Ou seja, o *owner rank* controla integralmente a escrita em seus domínios. Permitir que um processo altere diretamente um domínio remoto pode ser interessante em algumas situações, e essa funcionalidade deverá ser incluída no *framework* futuramente, mas pela complexidade que isso envolve, acabou ficando fora desse trabalho.

3.2.2 Criação do Grafo de Dados

Uma vez definidas as tarefas e seus respectivos domínios de entrada e de saída, o *framework* cria um grafo distribuído intermediário que representa o fluxo de dados entre as operações em cada *rank* MPI. Esse grafo é chamado de Grafo de Dados (mescla dados e tarefas), que depois será usado para deduzir o Grafo de Execução (só com tarefas). O grafo de dados é quase uma tradução direta dos métodos *task* da classe CoarseDataflow::Graph, exceto por algumas tarefas especiais inseridas pelo *framework*. Uma delas é a tarefa do tipo *GET*, responsável por receber um domínio em um estado específico do seu *owner rank*. Outra é a tarefa do tipo *PUT*, responsável por enviar um domínio local em um estado específico para outro *rank*. A Figura 3.5 mostra o grafo de dados gerado para a multiplicação matriz vetor da seção anterior. As tarefas *GET* e *PUT* correspondentes foram destacadas com a mesma cor em seus respectivos *ranks*.

Para gerar o grafo de dados, as tarefas do tipo *GET* podem ser deduzidas diretamente dos *inputs* das tarefas locais. Basta mapear os *inputs* com domínios remotos. Mas em cada *rank MPI*, o *framework* nada sabe sobre quais dados devem ser enviados a outros processos. Por isso, para criar as tarefas do tipo *PUT*, é necessário que os *ranks* troquem informações de quais dados precisam um do outro. Por exemplo, o processo 0 precisa sinalizar ao processo 1 que necessita do seu domínio x_1^0. A partir daí, o *framework* no processo 1 sabe que precisa criar uma tarefa *PUT* x_1^0 para o processo 0. O Algoritmo 1 descreve os passos usados para realizar a troca de informações.

Troca de informações de domínios para dedução de tarefas *PUT*

O objetivo do Algoritmo 1 é preencher os dados *outRemoteRanks* e *toProvideDomInfos*. O primeiro contém os *out ranks*, *ranks* para quem o processo corrente precisa

Figura 3.5: Grafo de Dados do exemplo de multiplicação matriz vetor. Os sobrescritos são os estados

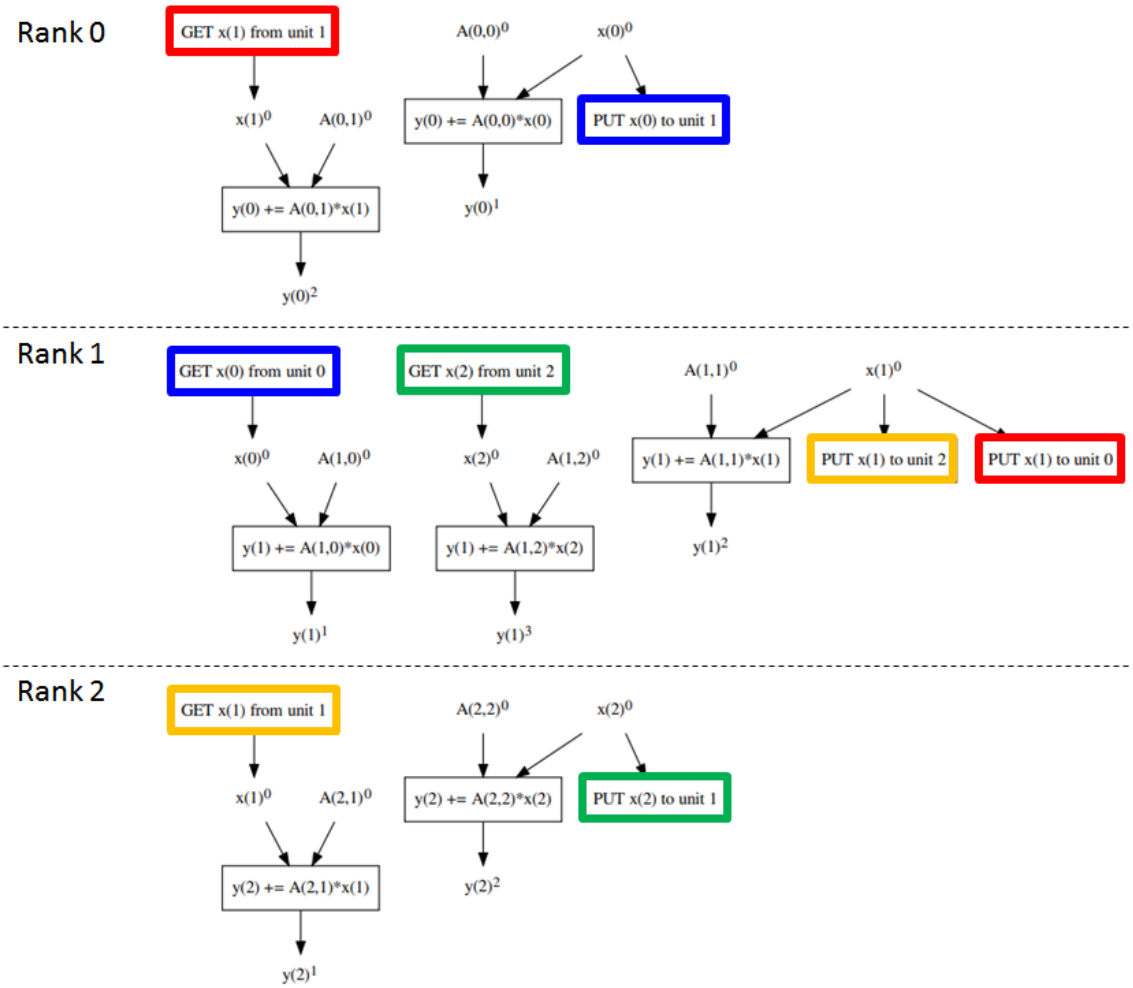
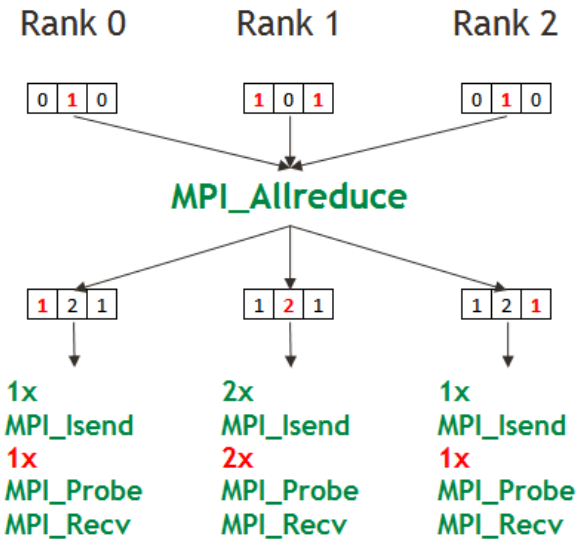


Figura 3.6: Esquema de troca de informações sobre domínios remotos requeridos entre os *ranks MPI*



enviar dados na etapa corrente de construção do GD. O segundo contém os dados a serem enviados (vetor de inteiros com os IDs dos objetos, IDs dos domínios e estados requeridos) para cada *out rank*. Para chegar nesses dados, o *framework* começa percorrendo os *in ranks* (*ranks* de quem o processo corrente precisa receber dados) preenchendo o *neededDomInfos* correspondente (com os dados de domínio a receber). Ambas as informações são obtidas diretamente dos *inputs* das tarefas locais. No processo, ele adiciona o valor 1 na posição de cada *in rank* do vetor temporário *countOutRemoteRanks* (linhas 2 a 5). Após a operação coletiva de redução na linha 7, *countOutRemoteRanks[k]* irá conter o **número de *out ranks*** do processo *k*. Em seguida, ele envia as mensagens com os dados requeridos dos *in ranks* usando rotinas MPI não bloqueantes (linhas 8 a 10). Finalmente, nesse ponto, é possível deduzir os *out ranks* invocando-se MPI_Probe *nOutRemoteRanks* vezes em sequência (variável preenchida na linha 11). Usando o atributo MPI_ANY_SOURCE, essa rotina captura mensagens vindas de qualquer *rank*. Em seguida, usa-se o *status* do *probe* para descobrir o *rank* emissor e também o tamanho da mensagem capturada. O *rank* emissor é adicionado ao vetor *outRemoteRanks*, e a mensagem em si é adicionada ao vetor *toProvideDomInfos*. Ao final do laço 12-21, ambos os dados estarão completos, e o *framework* estará apto a gerar as tarefas PUT necessárias. A Figura 3.6 ilustra o algoritmo. A depender do número de *in ranks* e *out ranks* de cada processo, essa operação pode ser significativa, pois cada processo envia $\#InRanks$ e recebe $\#OutRanks$ mensagens (todas do tipo ponto a ponto). Além disso, o próprio tamanho de cada mensagem depende do número de domínios comunicados entre elas.

3.2.3 Criação do Grafo de Execução

O Grafo de Dados é um passo intermediário da fase de *setup* do *dataflow*, e o objetivo dele é auxiliar a criação do Grafo de Execução, que é onde as tarefas e suas dependências são, de fato, definidas. As dependências entre as tarefas no Grafo de Execução são deduzidas a partir do Grafo de Dados com base nas informações de leituras e escritas nos domínios em cada estado. Existem três possíveis tipos de dependência de dados: RAW (*Read-After-Write*) ou dependência verdadeira, WAW (*Write-After-Write*) ou dependência de saída e WAR (*Write-After-Read*) ou Anti-dependência. A Figura 3.7 ilustra como cada tipo de dependência pode se manifestar no Grafo de Dados. Quando uma tarefa *taskA* escreve em um domínio D^i e outra tarefa *taskB* lê o domínio D^i, então há uma dependência verdadeira a primeira tarefa precisa ser executada antes da segunda, caracterizando uma dependência RAW. Quando uma tarefa *taskA* escreve em um domínio D^i e uma tarefa *taskB* escreve D^j, sendo o estado j maior que o estado i, então a tarefa que usa o menor valor de estado

Algoritmo 1: Troca de informações entre *ranks* para dedução de tarefas tipo *PUT*

```
   Inputs: inRemoteRanks
   Outputs: outRemoteRanks, toProvideDomInfos
 1 countOutRemoteRanks ← AllocateIntArray( nRanks )            ▷ Zero initialized
 2 for inRank in inRemoteRanks do
 3 |   countOutRemoteRanks[inRank] = 1
 4 |   neededDomInfos[inRank] ← ArrayWithNeededDomainInfosOf(inRank)
 5 end
 6 MPI_Allreduce( countOutRemoteRanks.data(), countOutRemoteRanks.size(), ... )
 7 nOutRemoteRanks = countOutRemoteRanks[thisRank]
 8 for inRank in inRemoteRanks do
 9 |   MPI_Isend( neededDomInfos[inRank].data(), neededDomInfos[inRank].size(), ... )
10 end
11 outRemoteRanks ← AllocateIntArray( nOutRemoteRanks )        ▷ Zero initialized
12 for (count = 0; count < nOutRemoteRanks; count ++) do
13 |   MPI_Status status
14 |   MPI_Probe( MPI_ANY_SOURCE, ..., &status )
15 |   outRank = status.MPI_SOURCE
16 |   outRemoteRanks[count] = outRank
17 |   messageSize = 0
18 |   MPI_Get_count( &status, ..., &messageSize )
19 |   toProvideDomInfos[count].resize(messageSize)
20 |   MPI_Recv( toProvideDomInfos[count].data(), ..., outRank, ... )
21 end
```

deve ser executada antes, caracterizando uma dependência WAW. Por fim, se uma tarefa *taskA* lê o domínio D^i e uma tarefa *taskB* escreve D^j, sendo o estado j maior que o estado i, então a tarefa *taskA* precisa ser executada primeiro, caracterizando uma dependência WAR. Esse último tipo de dependência não é tão intuitivo quanto as outras, mas suponha que *taskA* seja um PUT enviando o dado D^i de modo assíncrono. Se a tarefa *taskB* começar a alterar o dado antes da primeira terminar, o dado que está sendo enviado pode ser corrompido. Atualmente, o *framework* extrai as dependências verdadeiras (RAW) e de saída (WAW) do grafo de dados. O tratamento de anti-dependências está em desenvolvimento (esse tipo de dependência não afetou os algoritmos estudados).

O Grafo de Execução mostrado na Figura 3.2 foi gerado a partir do Grafo de Dados da Figura 3.5 aplicando-se essas regras de dependência. Por exemplo, no *rank* 0, a tarefa $y_0 \mathrel{+}= A_{0,0} * x_0$ escreve no domínio y_0 com estado 1, e a tarefa $y_0 \mathrel{+}= A_{0,1} * x_1$ escreve no mesmo domínio com o estado 2. Aplicando-se a regra WAW, a tarefa de maior estado passa a depender da de menor estado no Grafo de Execução, como se pode observar na primeira figura.

Na definição das tarefas, as funções *out* costumam gerar dependências do tipo WAW (de saída), enquanto as funções *in* e *inout* costumam gerar dependências RAW (verdadeiras). A função *reduce* é um caso à parte. O *framework* cria uma tarefa fictícia com uma dependência especial para todas as tarefas que invocam *reduce*

Figura 3.7: Exemplos dos tipos de dependência de dados

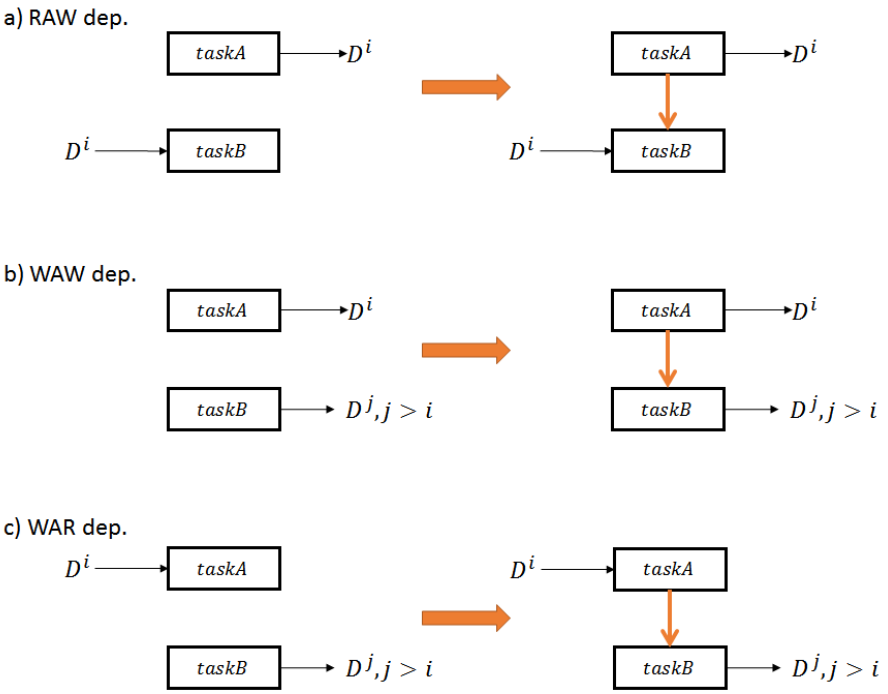
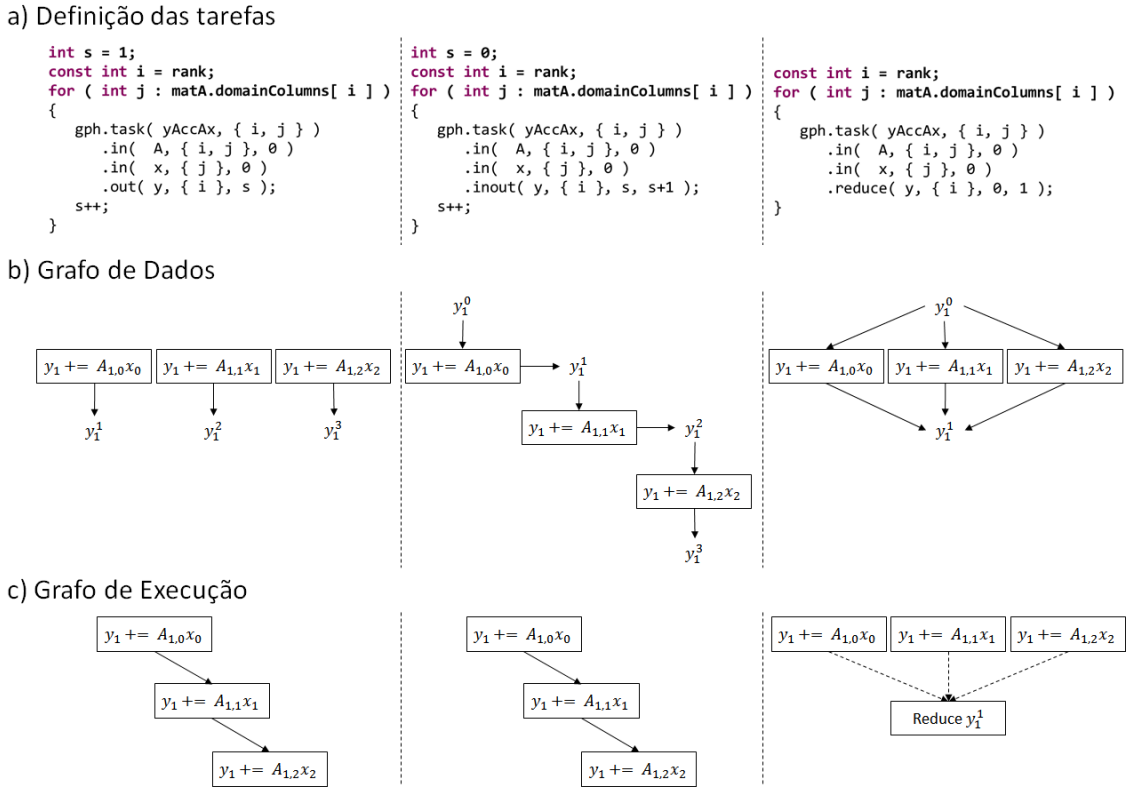


Figura 3.8: Exemplos das funções *in*, *out*, *inout* e *reduce* para definição das entradas e saídas de uma tarefa e seus efeitos nos grafos de dados e de execução



com o mesmo domínio/estado. Ela não faz nenhum tipo de computação, apenas marca o momento em que o estado foi alterado (quando todas as tarefas do *reduce* terminaram), para fins de outras dependências. A Figura 3.8, item (a), exemplifica o mesmo trecho de código (linhas 12 a 21 do Código 3.1) usando as funções *out*, *inout* e *reduce*. Os itens (b) e (c) da mesma figura mostram os efeitos de cada função no Grafo de Dados e no Grafo de Execução, respectivamente, do *rank 1* no exemplo de multiplicação matriz vetor.

Armazenamento do Grafo de Execução

O Grafo de Execução local de cada *rank* é representado internamente como um padrão de esparsidade (de matriz esparsa), que mapeia uma tarefa i às tarefas j que dependem de i. Ele é armazenado usando-se o formato *Compressed Sparse Row* ou CSR (SAAD [38]), muito utilizado por propiciar bom nível de compactação e boa performance para operações de busca.

3.2.4 *Generators*

O *Generator* é uma funcionalidade proposta para facilitar a definição de algoritmos distribuídos como fluxo de dados. As primeiras versões do *framework* forçavam o usuário a definir não só as tarefas que geravam dados para o próprio *rank*, mas também as tarefas que precisavam gerar dados para prover outros processos. Isso complicava um pouco o uso da ferramenta, comprometendo o requisito de simplicidade. As ferramentas mais avançadas que geram grafos distribuídos usam a estratégia de execução PTG (Seção 2.3.2). Nesse esquema, é necessário parametrizar todas as possíveis entradas e saídas de cada tarefa, o que pode ser bastante trabalhoso. Por outro lado, o *generator* simplesmente parametriza qual tarefa (ou conjunto de tarefas, quando se usa a construção *reduce*) gera um dado desejado, o que torna o uso do *framework* quase tão simples quanto as ferramentas com estratégia STF (Seção 2.3.2). Tal simplicidade traz um custo de comunicação, pois, como veremos, a construção do Grafo de Dados se torna iterativa, e os processos precisam trocar informações para inferir suas tarefas tipo *PUT* em cada iteração. No entanto, o tempo de comunicação só afeta o tempo da fase de *setup*, que foi baixo na grande maioria dos casos testados (Capítulo 4).

Para exemplificar seu uso, vamos usar a operação de multiplicação de vetor por uma matriz esparsa transposta logicamente: $y = A^T * x$. Essa operação é um tanto mais complexa que a multiplicação matriz-vetor comum. A Figura 3.9 ilustra as tarefas envolvidas na operação. O item (a) mostra como os dados estão distribuídos. Como a matriz A é transposta logicamente, então $A^T_{i,j}$ na verdade se refere a $A_{j,i}$ logicamente transposto. Portanto, como o *owning* dos domínios de A é por linha,

o de A^T é por coluna. O item (b) mostra as operações que deveriam ser feitas nos domínios seguindo mesmo raciocínio da multiplicação comum. No entanto, o *owning* por coluna de A^T faz com que algumas tarefas tenham como entrada um domínio de matriz remoto. Embora seja uma abordagem válida, a transferência de submatrizes teria um efeito muito negativo na performance. Por isso, o item (c) traz uma abordagem mais otimizada. É criado um novo objeto p particionado em domínios $p_{i,j}$ que armazena o vetor resultante da multiplicação parcial $A^T_{i,j} * x_j$ quando i for remoto. Assim, se transmite já o resultado da multiplicação dos domínios envolvidos (que é um vetor), ao invés de transmitir os dois domínios (uma matriz e um vetor). O *owning* de $p_{i,j}$ é igual ao de $A^T_{i,j}$.

Figura 3.9: Exemplo de multiplicação de vetor por matriz transposta

O Código 3.2 traz uma possível implementação da estratégia descrita no item (c) da figura anterior. A parte de definição do algoritmo, o primeiro laço, entre as linhas 19 e 31, contém as tarefas que atualizam os domínios y locais. Essas são as tarefas mais intuitivas, pois o usuário só precisa pensar em quais dados ele precisa para atualizar o y local de cada *rank*. O segundo laço, entre as linhas 35 e 42, contém tarefas para gerar multiplicações parciais p para serem consumidos por outros *ranks*. Nesse caso, o raciocínio é inverso. O usuário precisa pensar em quais dados locais precisam ser computados (estado maior que zero) em benefício de outros *processos*, o que pode não ser tão intuitivo. Note que, o primeiro laço percorre os domínios de

trasnpA por linha, e o segundo percorre por coluna.

O *generator* foi introduzido para facilitar a implementação de algoritmos complexos, permitindo que o usuário se concentre na escrita dos domínios locais de cada *rank MPI*. Ele contém uma receita para gerar domínios locais de um determinado objeto em um determinado estado. Assim, sempre que um *rank* qualquer sinalizar que precisa de um domínio/estado e esse dado não estiver previamente definido no seu *owner rank*, então o *framework* aciona o *generator* correspondente com a receita para criar o dado solicitado, que pode envolver a criação de várias outras tarefas, inclusive com o acionamento de outros *generators*.

Usando essa *feature*, a construção do Grafo de Dados se torna iterativa. Na primeira etapa, o grafo contém as tarefas incluídas diretamente pelo usuário. Na segunda etapa, contém também as tarefas incluídas por *generators* na primeira etapa. Nas etapas seguintes, se houver, contém as tarefas incluídas pelos *generators* da etapa anterior. Portanto, se por um lado, essa *feature* simplifica implementações de algoritmos *dataflow*, por outro ela pode onerar o desempenho da fase de *setup*, especialmente se houve muitos *generators* aninhados, pois em cada etapa da construção do Grafo de Dados, os *ranks* precisam trocar informações entre si.

Código 3.2: Exemplo de definição de multiplicação matriz transposta vetor no *framework dataflow*

```
1  CoarseDataflow dfTMatVec( penv.get(), "TMatVec", [&]( CoarseDataflow::Graph& gph )
2  {
3     // Registra os objetos
4     auto T = gph.object( &objAt ); // T = A logicamente transposto
5     auto x = gph.object( &objX );
6     auto y = gph.object( &objY );
7     auto p = gph.object( &objP );
8
9     // Registra as Closures
10    auto yAccAx = gph.closure( &tskTMatVec );  // Representa y(i) += T(i,j)*x(j), i == j
11    auto yAccP = gph.closure( &tskAccP );      // Representa y(i) += p(i,j)
12    auto pEqAx = gph.closure( &tskPartialTx ); // Representa p(i,j) = T(i,j)*x(j), i != j
13
14    const int i = rank;
15
16    // Define a tarefa diagonal 'y(i) += T(i,i)*x(i)'
17    gph.task( yAccAx, { i, i } )
18       .in(  T, { i, i }, 0 )
19       .in(  x, { i }, 0 )
20       .out( y, { i }, 1 );
21
22    // Define as tarefas 'y(i) += p(i,j)'
23    int s = 2;
24    for ( int j : transpA.domainColumns[ i ] )
25    {
26       if ( i != j )
27          gph.task( yAccP, { i, j } )
28             .in(  p, { i, j }, 1 )
29             .out( y, { i }, s );
30       s++;
31    }
32
33    // Define as tarefas 'p(i,j) = T(i,j)*x(j)'
34    const int j = rank;
35    for ( int i : transpA.domainRows[ j ] )
36       if ( i != j )
37       {
38          gph.task( pEqAx, { i, j } )
39             .in(  T, { i, j }, 0 )
40             .in(  x, { j }, 0 )
41             .out( p, { i, j }, 1 );
42       }
43 } );
44
45 // Executa o dataflow
46 dfTMatVec.execute();
```

O Código 3.3 traz a multiplicação de vetor por matriz transposta usando *generator* para os domínios locais de p. Note que, comparado ao código anterior, o segundo laço deu lugar a uma receita que simplesmente sabe gerar um domínio local $p_{i,j}$, sem se preocupar em identificar quais são os $p_{i,j}$ que precisam ser gerados somente para prover processos remotos. Isso é feito automaticamente ao final da troca de informações entre os *ranks* descrita na Seção 3.2.2. Em algoritmos complexos, a identificação dos dados que precisam ser gerados somente para uso externo, além de não intuitivo, pode acontecer de alguma das informações necessárias para a dedução desses dados não estar replicada em todos os *ranks*, forçando o usuário a comunicar esses dados previamente. O *generator* aproveita a comunicação interna que o *framework* promove entre os *ranks* para identificar os dados que precisam ser gerados.

Código 3.3: Exemplo de definição de multiplicação matriz transposta vetor no *framework dataflow* usando *generator*

```
1  CoarseDataflow dfTMatVec( penv.get(), "TMatVec", [&]( CoarseDataflow::Graph& gph )
2  {
3     // Registra os objetos
4     auto T = gph.object( &objAt ); // T = A logicamente transposto
5     auto x = gph.object( &objX );
6     auto y = gph.object( &objY );
7     auto p = gph.object( &objP );
8
9     // Registra as Closures
10    auto yAccAx = gph.closure( &tskTMatVec );  // Representa y(i) += T(i,j)*x(j), i == j
11    auto yAccP = gph.closure( &tskAccP );      // Representa y(i) += p(i,j)
12    auto pEqAx = gph.closure( &tskPartialTx ); // Representa p(i,j) = T(i,j)*x(j), i != j
13
14    const int i = rank;
15
16    // Define a tarefa diagonal 'y(i) += T(i,i)*x(i)'
17    gph.task( yAccAx, { i, i } )
18       .in(  T, { i, i }, 0 )
19       .in(  x, { i }, 0 )
20       .out( y, { i }, 1 );
21
22    // Define as tarefas 'y(i) += p(i,j)'
23    int s = 2;
24    for ( int j : transpA.domainColumns[ i ] )
25    {
26       if ( i != j )
27          gph.task( yAccP, { i, j } )
28             .in(  p, { i, j }, 1 )
29             .out( y, { i }, s );
30       s++;
31    }
32
33    // Gerador para 'p(i,j) = T(i,j)*x(j), com i != j'
34    gph.generator( p, 0, 1, [=]( const ids_t& ids ) mutable
35    {
36       int const i = ids[ 0 ];
37       int const j = ids[ 1 ];
38       gph.task( pEqAx, { i, j } )
39          .in(  T, { i, j }, 0 )
40          .in(  x, { j }, 0 )
41          .out( p, { i, j }, 1 );
42    } );
43 } );
44
45 // Executa o dataflow
46 dfTMatVec.execute();
```

O método *generator* da classe CoarseDataflow::Graph recebe como parâmetro o *handler* do objeto, o estado inicial, o estado final e um *lambda function* com a receita de um domínio local do objeto parametrizado pelos índices do domínio. Como esse *lambda* é executado à *posteriori*, não imediatamente, é preciso ter especial cuidado com a captura de variáveis. O recomendado é capturar *handlers* por cópia e classes complexas, eventualmente usadas, por referência.

O *generator* funciona quase como uma chamada de função remota. Tomando como exemplo o algoritmo da Figura 3.9 item (c), quando o *rank 0* sinaliza ao *rank 1* que precisa do domínio $p_{0,1}$ no estado 1, o *framework* no *rank 1* aciona o respectivo *generator* que cria a tarefa $p_{0,1} = A_{0,1}^T * x_1$, que fornece o dado solicitado. O mesmo raciocínio se estende aos demais $p_{i,j}$ de cada processo.

3.2.5 Subgrafos

A noção de subgrafos está relacionada a uma política de execução implementada, que pode ser especialmente útil quando se faz uso de *generators* aninhados nas definições das tarefas. A ideia básica é privilegiar tarefas tipo *PUT*, incluindo toda a cadeia de operações até elas. Mais detalhes serão discutidos na Seção 3.3.3.

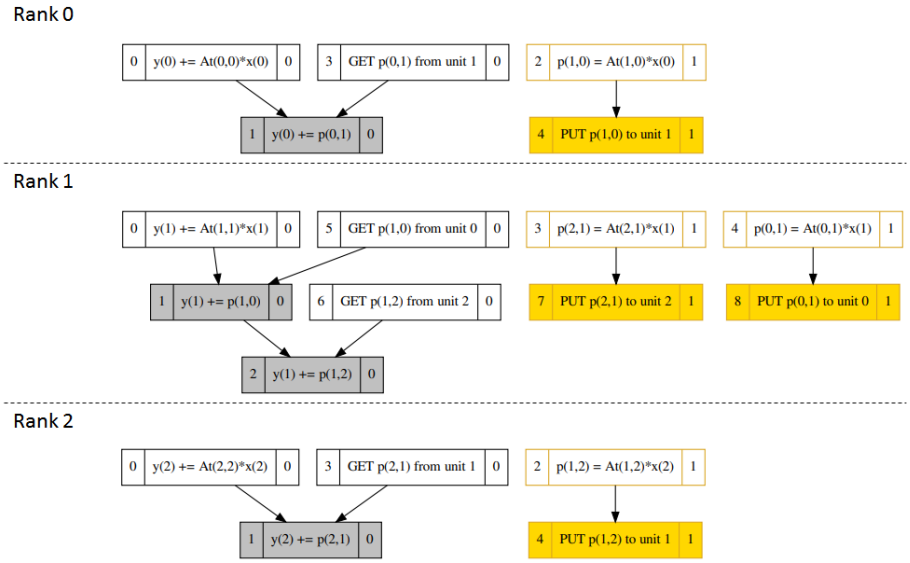
A solução desenvolvida para implementar essa política foi particionar o Grafo de Execução local em subgrafos agrupados por conjuntos de tarefas tipo *PUT*. Como visto na Seção 3.2.4, usando *generators*, o grafo de dados é construído de modo incremental. Todos os *PUTs* incluídos em uma mesma etapa da construção incremental são atribuídos a um subgrafo identificado por um número sequencial iniciando-se em 1 (inicialmente, o subgrafo 0 irá conter todas as demais tarefas). Depois que o Grafo de Dados (GD) é concluído, o Grafo de Execução (GE) é gerado normalmente para posterior particionamento. Internamente, o *framework* terá um conjunto de *NSG* subgrafos, que inclui o subgrafo 0 e outros *NSG-1* subgrafos, cada um associado a um grupo de *PUTs*. Em seguida, no Grafo de Execução é feita uma busca no sentido oposto ao fluxo de dados partindo de cada conjunto de *PUTs*. Todas as tarefas que forem alcançadas por um conjunto de *PUTs* serão atribuídas ao mesmo SG delas. As tarefas que foram alcançadas por mais de um conjunto de *PUTs* serão associadas ao SG de maior índice dentre os conjuntos. A Figura 3.10 mostra o Grafo de Execução dividido em subgrafos dos exemplos 3.2 e 3.3 (são iguais, nesse caso). Note só há dois os subgrafos: o 0 (coloridos em cinza) e o 1 (colorido em amarelo). Tipicamente, se em *NE* etapas de construção do GD um conjunto não vazio de *PUTs* foi incluído, então *NSG* será igual a *NE+1*. O *NE* do exemplo é 1 (todos os *PUTs* são inseridos na primeira etapa), logo $NSG = NE + 1 = 2$.

3.3 Execução de algoritmo *Dataflow*

A fase de *setup* do *dataflow* é concluída no construtor de uma instância de CoarseDataflow. Nesse ponto, o objeto terá uma lista de tarefas e um Grafo de Execução local, deduzido do fluxo de dados informado pelo usuário durante o *setup*.

Antes de explicar como o grafo é usado para executar o algoritmo, precisamos detalhar o conceito de tarefas no *framework dataflow*. Primeiramente, todas as tarefas são presumidamente **assíncronas**, e podem ter mais de um estágio (ou *step*) de execução. Elas são materializadas ao se implementar certos métodos abstratos da classe Closure ou da classe Domain, a depender da natureza da operação. Tarefas de natureza computacional são definidas nas realizações de Closure, e tarefas de comunicação são definidas nas realizações de Domain.

Figura 3.10: Grafo de Execução, com subgrafos, do exemplo de multiplicação de vetor por matriz transposta



3.3.1 Tarefas de comunicação

As tarefas de comunicação são tarefas especiais que sabem como enviar e receber dados de um domínio. O emissor, é sempre o *owner rank* do dado, e o receptor é um *rank* remoto qualquer. Para materializar uma tarefa do tipo *GET* para um domínio, é preciso implementar os métodos *nStepsGet()* e *getTask()* mostrados no Código 3.4. O primeiro define o número de estágios da tarefa, e o segundo retorna uma instância de AsyncHandler para o estágio atual.

Código 3.4: Métodos abstratos da classe Domain que definem tarefas de comunicação ponto a ponto

```
1  // Definicao de tarefas GET
2  virtual int nStepsGet() const = 0;
3  virtual boost::shared_ptr<AsyncHandler> getTask( int const step, int arcId ) = 0;
4  // Definicao de tarefas PUT
5  virtual int nStepsPut() const = 0;
6  virtual boost::shared_ptr<AsyncHandler> putTask( const ProcessingUnit& targetUnit,
7                                                   int const step, int arcId ) = 0;
```

O AsyncHandler é uma classe genérica cujo principal método é o *test()*, que retorna o status do *handler*: *EXECUTING* ou *COMPLETED* (futuramente também *CANCELED* e *ABORTED*). O *framework* provê a especialização HighLevelOmpMpiAsyncHandler que manipula mensagens MPI não bloqueantes (ponto a ponto). Essa especialização armazena *requests* do MPI, e seu método *test()* chama MPI_Testall para todos *requests*, retornando o status adequado.

A possibilidade de múltiplos estágios por tarefa dá flexibilidade para a comunicação de dados complexos. No mundo esparso, um *storage* de matriz pode ter uma estrutura altamente intrincada, envolvendo diversos *arrays* de variados tipos e tamanhos. Um simples *storage* CSR (SAAD [38]) escalar no SolverBR, por exem-

plo, é composto de um *array IA* de inteiros 64 bits, um *array JA* de inteiros 32 bits e um *array VA* de valores ponto flutuante de dupla precisão. O tamanho do primeiro *array* é igual ao número de linhas da matriz acrescido de uma unidade. E o tamanho dos outros dois é igual ao número de entradas não nulas na matriz. Apesar de ser um dos formatos mais simples para matrizes esparsas, receber um CSR em uma única mensagem MPI não é algo trivial. O *rank* emissor poderia até criar um *datatype* com MPI_Type_create_struct e enviar tudo com um único *send*, mas o *rank* receptor não tem informações a priori para inferir toda a estrutura do dado a ser recebido.

Código 3.5: Exemplo de tarefa *GET* com múltiplos estágios para receber uma matriz CSR simples do SolverBR

```
1
2  int ExampleCSRDomain::nStepsGet() const
3  {
4      return 3;
5  }
6
7  boost::shared_ptr<AsyncHandler> ExampleCSRDomain::getTask( int const step, int arcId )
8  {
9      if ( step == 0 )
10     {
11         // Aloca um buffer para receber o IA. O numero de linhas eh conhecido
12         SharedArray<int64> IA( nRows() + 1 );
13         // Inicia send nao bloqueante com o request do handler
14         auto asyncHandler = boost::make_shared<HighLevelOmpMpiAsyncHandler>();
15         asyncHandler->setNRequests( 1 );
16         MPI_Irecv( IA.begin(), IA.size(), ..., & asyncHandler->request( 0 ) );
17         // Retorna o handler
18         return asyncHandler;
19     }
20     else if ( step == 1 )
21     {
22         // Recupera o numero de entradas nao nulas na ultima posicao de IA
23         const int64 nnz = IA[ nRows() ];
24         // Aloca um buffers para JA e VA
25         SharedArray<int> JA( nnz );
26         SharedArray<double> VA( nnz );
27         // Cria 2 requets e chama um Isend para JA e outro para VA
28         auto asyncHandler = boost::make_shared<HighLevelOmpMpiAsyncHandler>();
29         asyncHandler->setNRequests( 2 );
30         MPI_Irecv( JA.begin(), JA.size(), ..., & asyncHandler->request( 0 ) );
31         MPI_Irecv( VA.begin(), VA.size(), ..., & asyncHandler->request( 1 ) );
32         // Retorna o handler
33         return asyncHandler;
34     }
35     else
36     {
37         // Instancia um storage com os arrays recebidos
38         m_storage = StorageFactory::CreateCSR( IA, JA, VA );
39         // Retorna um handler vazio
40         return boost::make_shared<AsyncHandler>();
41     }
42 }
```

O Código 3.5 exemplifica como um *storage* CSR poderia ser recebido em um domínio remoto usando-se múltiplos *steps* em uma classe fictícia ExampleCSRDomain. A implementação de *nStepsGet()* estabelece 3 estágios. Isso significa que, durante a execução da tarefa *GET* para um domínio ExampleCSRDomain, o método *getTask()* será invocado 3 vezes, uma vez para cada *step*. No primeiro estágio do exemplo, como o número de linhas da matriz é conhecido, aloca-se um *buffer* com o tamanho necessário para receber o IA (linha 12). A seguir, cria-

se uma instância de HighLevelOmpMpiAsyncHandler com 1 *request*, que é usado usado na rotina MPI_Irecv (linhas 14 a 16), e o *asyncHandler* do *step* é retornado. Quando o status do *handler* é avaliado como COMPLETED, o método *ExampleCSRDomain::getTask()* é novamente acionado com o parâmetro *step* incrementado. Então, no segundo estágio, o IA estará preenchido e dele se pode extrair o número de entradas não nulas (ou nnz). Com essa informação, é possível alocar os *buffers* para JA e VA (linhas 23 a 26). Os *receives* dos dois *arrays* podem ser chamados simultaneamente, então instancia-se um HighLevelOmpMpiAsyncHandler com 2 *requests*, um para cada dado (linhas 28 a 31). O *asyncHandler* desse *step* é então retornado. Esse *handler* só terminará quando ambos os *requests* terminarem. Por fim, no último *step* não há qualquer tipo de comunicação. Esse estágio foi usado no exemplo como uma espécie de pós processamento, em que todos os *arrays* recebidos são consolidados em uma classe de *storage* e atribuídos a uma variável membro de ExampleCSRDomain. Nesse caso é retornado um AsyncHandler vazio, cujo status é sempre COMPLETED (atuando como uma tarefa síncrona).

Código 3.6: Exemplo de tarefa *PUT* para envio de uma matriz CSR simples do SolverBR

```
1
2  int ExampleCSRDomain::nStepsPut() const
3  {
4      return 1;
5  }
6
7  boost::shared_ptr<AsyncHandler> ExampleCSRDomain::putTask( const ProcessingUnit& targetUnit,
8                                                             int const step, int arcId )
9  {
10     auto cons& IA = m_storage->IA();
11     auto cons& JA = m_storage->JA();
12     auto cons& VA = m_storage->VA();
13     // Cria um handler com 3 requests, um para cada dado a ser enviado
14     auto asyncHandler = boost::make_shared<HighLevelOmpMpiAsyncHandler>();
15     asyncHandler->setNRequests( 3 );
16     MPI_Isend( IA.begin(), IA.size(), ..., & asyncHandler->request( 0 ) );
17     MPI_Isend( JA.begin(), JA.size(), ..., & asyncHandler->request( 1 ) );
18     MPI_Isend( VA.begin(), VA.size(), ..., & asyncHandler->request( 2 ) );
19     // Retorna o handler
20     return asyncHandler;
21 }
```

O Código 3.6 mostra uma possível implementação para as tarefas *PUT* da mesma classe fictícia ExampleCSRDomain. Todos os dados podem ser enviados em um único *step*. O parâmetro *targetUnit* encapsula o rank MPI remoto que receberá os dados (o emissor é sempre o *owner rank* do domínio). Cria-se então um HighLevelOmpMpiAsyncHandler com 3 *requests* para serem usados em cada um dos *sends* não bloqueantes.

Classe AsyncTask

Uma tarefa, com seus múltiplos estágios, é encapsulada em uma classe interna do *framework* chamada AsyncTask. O Algoritmo 2 descreve o funcionamento do método *test()* de AsyncTask (o método que executa/verifica o status da tarefa). Enquanto

o AsyncHandler controla o status de um *step* da execução, o AsyncTask controla o status da tarefa em si. Basicamente, quando o primeiro estágio da tarefa é iniciado, seu status é EXECUTING. Quando um *handler* é concluído, o *step* é incrementado e o próximo *handler* é obtido executando-se novamente a função geradora com o novo *step* (linhas 8 a 10 e 2 a 5 do algoritmo). Ao final, quando o último *step* é concluído, o status da tarefa passa a ser COMPLETED (linha 13).

Algoritmo 2: The test() method of AsyncTask

```
1  if nStesp > 0 then
       NEXT_STEP:
2      if prevStep < step then
3          prevStep = step
               ▷ The Closure's task is executed whenever the step changes
4          asyncHandler = closure.task( taskIDs, step )
5      end
       ▷ Check the handler's status.  For communication tasks, it typically
        calls the MPI_Testall routine for the MPI requests in the handler
6      taskState = asyncHandler.test()
7      if taskState = COMPLETED then
8          step++
9          if step < nSteps then
               ▷ Try execute the next step's handler
10             go to NEXT_STEP
11         else
               ▷ Task COMPLETED. Restore the initial values of the variables
12             ResetTask()
13         end
14     end
15 else
16     taskState = COMPLETED
17 end
18 return taskState
```

3.3.2 Tarefas de computação

As tarefas de computação são definidas nas realizações da classe Closure. Normalmente elas representam uma ação ou cálculo do algoritmo sendo implementado. O Código 3.7 mostra um exemplo de tarefa de computação que realiza uma multiplicação entre um domínio de matriz e um domínio de vetor. Normalmente essas tarefas só têm um estágio e são síncronas, na prática, retornando um AsyncHandler vazio. Mas é claro que, a depender da necessidade do usuário, uma tarefa de computação pode ter múltiplos estágios e ser assíncrona.

Código 3.7: Exemplo de tarefa de computação de executa uma multiplicação entre um domínio de matriz e um domínio de vetor

```
1
2  int ExampleMatVecClosure::nSteps() const
3  {
4      return 1;
5  }
6
7  boost::shared_ptr<AsyncHandler> MatVecClosure::task( const ids_t& ids, int const step )
8  {
9      // Extrai os indices na n-upla
10     const int i = ids[ 0 ];
11     const int j = ids[ 1 ];
12
13     // Acessa os respectivos dominios dos objetos distribuidos
14     MatDom* Aij = m_A->domain( { i, j } );
15     VecDom* xj = m_x->domain( { j } );
16     VecDom* yi = m_y->domain( { i } );
17
18     // Executa a multiplicacao
19     multiply( Aij, xj, yi );
20
21     // Retorna um handler vazio
22     return boost::make_shared<AsyncHandler>();
23 }
```

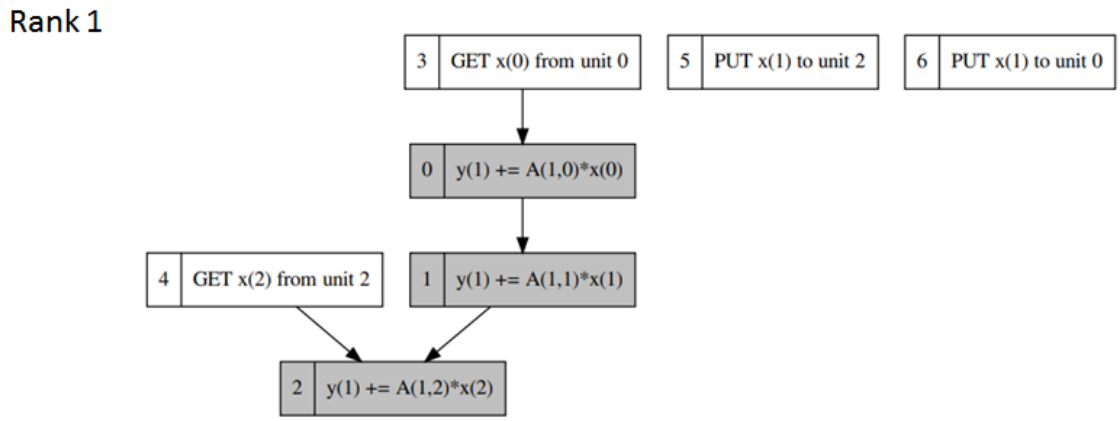
Essas tarefas são materializadas ao se implementar os métodos *nSteps()* e *task()* da classe Closure. O primeiro retorna o número de *steps* da tarefa (normalmente 1), e o segundo efetua a operação recebendo como parâmetros uma n-upla de índices e o step atual. O retorno é um AsyncHandler (normalmente vazio). Os índices recebidos, devem respeitar a cardinalidade definida pelo usuário no construtor da classe. Tipicamente, a realização da Closure possui acesso aos objetos (realizações de DataObject) dos domínios que serão operados no método *task()*, e a cardinalidade dos índices devem ser tais que permitam acessar os devidos domínios. Por exemplo, no Código 3.7, linhas 14 a 16, a classe MatVecClosure possui como membros os objetos A, x e y e utiliza um par de índices *{i,j}* para acessar os domínios $A_{i,j}$, x_j e y_i, necessários para a realização da multiplicação $y_i = A_{i,j} * x_j$ (linha 19).

3.3.3 Execução das tarefas

Essa seção detalha como o *framework* gerencia a execução das tarefas. Ao final do *setup*, que ocorre durante o construtor de CoarseDataflow, o *framework* produz o Grafo de Execução (GE), conforme explicado na Seção 3.2.3. Desse grafo se deriva o *srcAsyncCount*, um *array* interno com o número de dependências de entrada para cada tarefa (cada tarefa possui um ID sequencial interno). Assim, *srcAsyncCount[t]* informa o número de dependências que chegam na tarefa indexada por *t*. Desse dado se deriva o *initialAsyncList*, *array* com os IDs das tarefas sem dependências de entrada. Vamos tomar como exemplo o GE do *rank 1* gerado a partir do Código 3.1.

No exemplo da Figura 3.11, as caixas representam tarefas. Elas são divididas em dois campos: o campo à esquerda traz o ID sequencial da tarefa, e

Figura 3.11: Grafo de Execução do *rank 1* para o exemplo de multiplicação matriz vetor, Código 3.1



Algoritmo 3: Fill the *srcAsyncCount* array

```
1 for t=0; to nTasks-1 do
2 |   srcAsyncCount[t] ← número de dependências de entrada na tarefa t
3 end
```

Algoritmo 4: Fill the *initialAsyncList* array

```
  // Type order:  PUT, GET and others
1 for t in TaskIDsOrderedByTypeAndID() do
2 |   if srcAsyncCount[t]=0 then
3 |   |   initialAsyncList[t].pushBack(t)
4 |   end
5 end
```

o campo à direita traz o nome (configurado no construtor da classe *Closure*). O *srcAsyncCount* é preenchido segundo o Algoritmo 3. Por exemplo, a tarefa $y(1) \mathrel{+}= A(1,2) * x(2)$, cujo ID sequencial é 2, possui 2 dependências de entrada, logo $srcAsyncCount[2] \leftarrow 2$. Aplicando a regra em todas as tarefas do exemplo, temos $srcAsyncCount = \{\ 1, 1, 2, 0, 0, 0, 0\ \}$. Para preencher o *initialAsyncList* (ver Algoritmo 4), o *framework* procura as tarefas com zero entradas em *srcAsyncCount* e insere em *initialAsyncList*, ordenando primeiramente por tipo (PUT, GET e outras) e depois por ID sequencial (entre as tarefas de um mesmo tipo, a ordem de criação é preservada). **O objetivo é iniciar as tarefas de comunicação o quanto antes para favorecer a sobreposição de comunicação e computação**. Assim, para o exemplo, $initialAsyncList = \{\ 5, 6, 3, 4\ \}$ (tarefas com 0 dependências de entrada, primeiro as tipo *PUT*, depois as *GET* e depois as comuns). **Tudo isso ocorre ao final da fase de *setup***. Com essas informações, o *dataflow* está pronto para ser executado.

Full Graph Policy

Uma execução é iniciada ao se invocar o método *execute()* de uma instância da classe CoarseDataflow. O Algoritmo 5 detalha o esquema de execução usando a política padrão (chamada *Full Graph*). Duas estruturas de dados locais ao método *execute()* guiam o processamento: o vetor *asyncCount* é inicializado como uma cópia de *srcAsyncCount*, e a lista encadeada *execList* é preenchida com os ID de tarefas em *initialAsyncList* (linhas 1 e 2 do algoritmo). Os dados originais são assim preservados para futuras execuções, e os dados locais ao *execute()* podem ser alterados e serão destruídos ao final do método. Na linha 7, o *tasks* é um vetor que mapeia o ID de uma tarefa à sua instância de AsyncTask (classe que representa a tarefa assíncrona, discutida na Seção 3.3.1).

A Figura 3.12 traz um esquema do processo de execução do exemplo de multiplicação matriz-vetor no *rank 1*, com base no grafo de execução da Figura 3.11. Nos itens (a) e (b) da figura, as tarefas 5 e 6 foram iniciadas, mas não concluídas. No item (c), suponhamos que a tarefa 3 tenha sido iniciada e concluída, então a tarefa 0 se torna disponível e entra na lista de execução no lugar da tarefa 3, como ilustrado nos itens (d) e (e) da mesma figura. No item (g), por ser uma tarefa de computação, a tarefa 0 é concluída, tornando a tarefa 1 disponível. O *framework* percorre a lista de execução seguidas vezes até que todas as tarefas sejam executadas e a lista fique vazia, como ilustra o item (h). Como as tarefas de comunicação são assíncronas e usam apenas rotinas não bloqueantes, um processo fica em espera ocupada (*busy wait*) quando houver somente tarefas de comunicação em sua lista de execução (isso pode ser prejudicial do ponto de vista de consumo de energia).

Figura 3.12: Execução do exemplo de multiplicação matriz-vetor no *rank 1*

Algoritmo 5: Dataflow Execution (with **Full Graph** policy)

```
1  asyncCount ← srcAsyncCount
2  execList ← initialAsyncList
3  while not execList.empty() do
4      for (t = execList.begin(); t < execList.end(); ) do
5          status = tasks[t].test()
6          if status=COMPLETED then
7              t = execList.erase(t)
8              for dep in DependentTasksOf(t) do
9                  asyncCount[dep]−−
10                 if asyncCount[dep]=0 then
11                     execList.insert(t,dep)
12                 end
13             end
14         else
15             t ++
16         end
17     end
18 end
```

Algoritmo 6: Dataflow Execution (with **Subgraphs** policy)

```
1  asyncCount ← srcAsyncCount
2  for sg in subgraphsOrder do
3  |  subgraphExecList[sg] ← initialAsyncList[sg]
4  end
5  for sg in subgraphsOrder do
6  |  execList ← subgraphExecList[sg]
7  |  execState = execList.empty() ? COMPLETED : EXECUTING
8  |  while execState != EXECUTING do
9  |  |  for (t = 0; t < nTasks; ) do
10 |  |  |  execState = ExecSubgraph(execList,subgraphExecList,t,sg)
11 |  |  end
12 |  |  if sg < subgraphsOrder.last() then
13 |  |  |  nPutTasks = 0
14 |  |  |  for (t = execList.begin(); t < execList.end(); t ++) do
15 |  |  |  |  if taskType[t]=PUT & DependentTasksOf(t).size()=0 then
16 |  |  |  |  |  nPutTasks++
17 |  |  |  |  end
18 |  |  |  end
19 |  |  |  if nPutTasks = execList.size() then
20 |  |  |  |  break
21 |  |  |  end
22 |  |  end
23 |  end
24 end
25 Function ExecSubgraph(execList,subgraphExecList,t,sg):
26 |  subgraphExecState = EXECUTING
27 |  status = tasks[t].test()
28 |  if status=COMPLETED then
29 |  |  t = execList.erase(t)
30 |  |  for dep in DependentTasksOf(t) do
31 |  |  |  asyncCount[dep]−−
32 |  |  |  if asyncCount[dep]=0 then
33 |  |  |  |  if sg=subgraphOf[dep] then
34 |  |  |  |  |  execList.insert(t,dep)
35 |  |  |  |  else
36 |  |  |  |  |  subgraphExecList[sg].pushBack(dep);
37 |  |  |  |  end
38 |  |  |  end
39 |  |  end
40 |  |  if execList.empty() then
41 |  |  |  subgraphExecState = COMPLETED
42 |  |  end
43 |  else
44 |  |  t ++
45 |  end
46 |  return subgraphExecState
47 end
```

Subgraphs Policy

Usando a política de subgrafos, a execução ocorre de modo similar, porém, cada subgrafo é executado por vez do maior para o menor índice, mas privilegiando subgrafos sem tarefas do tipo *GET*. O Algoritmo 6 detalha o esquema de execução usando a política *Subgraphs*. Os subgrafos são processados segundo a ordem estabelecida (linha 5). Como no algoritmo anterior, as variáveis *asyncCount* e *subgraphExecList*, locais ao método, são inicializadas com os valores de *srcAsyncCount* e *initialAsyncList*, respectivamente, com a diferença que, nessa política, existe um *initialAsyncList* para cada subgrafo (linhas 1 a 4). Uma lista inicialmente vazia, *execList*, é preenchida com *subgraphExecList* no início de cada iteração (linha 6). A função ExecSubgraph, invocada na linha 7, processa uma tarefa de maneira similar ao algoritmo anterior, mas quando uma tarefa dependente se torna *ready* (seu *asyncCount* vai a zero), é necessário verificar a qual subgrafo ela pertence e inseri-lo a lista de execução inicial correta (linhas 30 a 34). Todos os parâmetros dessa função são passados por referência, e ela retorna o status COMPLETED quando *execList* fica vazia (o que significa que os subgrafos processados até o momento foram concluídos).

Após a execução dessa função, o *framework* percorre a *execList* verificando os tipos de cada tarefa pelo seu ID. Se o subgrafo corrente não for o último e se todas as tarefas restantes na lista de execução forem do tipo *PUT* sem dependentes, ele força a saída do laço *while* (linha 8), iniciando o processamento do próximo subgrafo. As tarefas anteriores permanecem em *execList* e continuarão sendo executadas, mas juntamente com as tarefas disponíveis no subgrafo seguinte, para favorecer a sobreposição de comunicação e computação (não faz sentido ficar aguardando o término de tarefas *PUT* se houver tarefas de computação a serem executadas em outros subgrafos).

3.3.4 Débitos Técnicos do *Framework Dataflow*

Essa seção discute algumas das deficiências atuais da ferramenta que são consideradas débitos técnicos. Elas não incluem todas as possíveis melhorias, mas apenas aquelas que comprometem a completude do *framework* na arquitetura atual. Evoluções mais amplas serão discutidas na Seção 5.2.

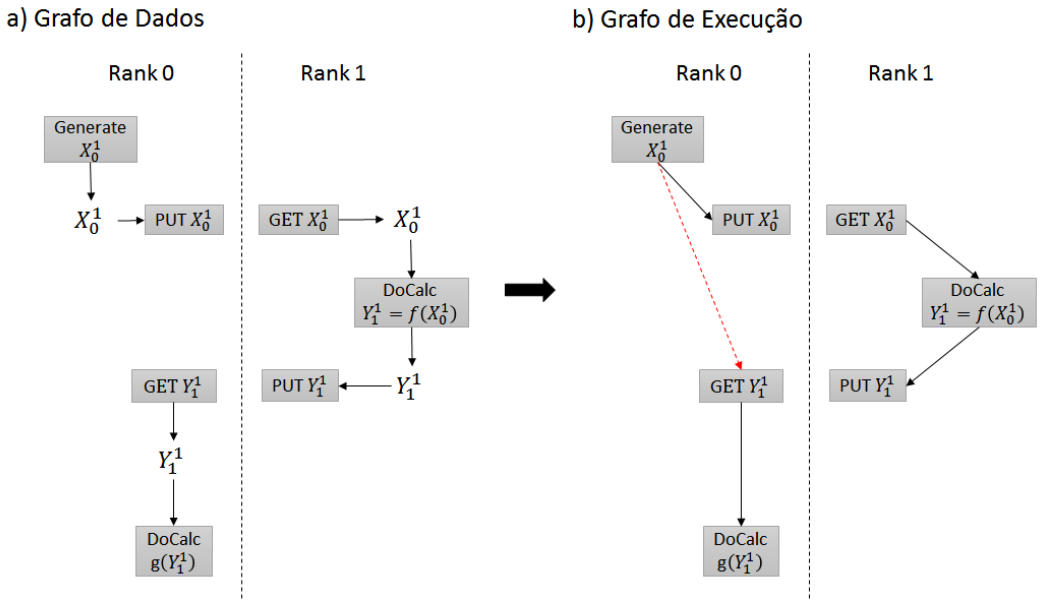
Débitos Técnicos da fase de *Setup*

Duas deficiências na construção do grafo (Seção 3.2.3) fazem com que, atualmente, as tarefas tipo *GET* e tipo *PUT* atuem sempre como **fontes** e **sumidouros** de dados, respectivamente. Isso significa que as tarefas *GET* nunca têm uma dependência de entrada, e as *PUT* nunca têm um dependência de saída. Porém, em algumas situações específicas, essas restrições causam problemas. Uma situação em que uma

tarefa PUT deveria ter uma dependência de saída, seria justamente na ocorrência de uma anti-dependência (WAR) envolvendo um PUT, conforme descrito na Seção 3.2.3. Já uma situação em que uma tarefa GET deveria ter uma dependência de entrada, está ilustrada na Figura 3.13. O item (a) da figura traz um Grafo de Dados onde o domínio X_0^1 é gerado no *rank 0* e enviado ao *rank 1*. No *rank 1* uma tarefa usa X_0^1 para gerar Y_1^1, que, por sua vez, é enviado ao *rank 0*. O item (b) da figura mostra o Grafo de Execução esperado. Conceitualmente, a tarefa *GET* Y_1^1 deveria depender da *Generate* X_0^1 (arco vermelho pontilhado na figura), pois Y_1^1 está no fluxo de X_0^1. Atualmente, o *framework* não captura essa informação no *rank 0*. A forma mais imediata de fazer isso seria com uma ordenação topológica (PRIYA [39]) global. A vantagem dessa solução é que, além de identificar corretamente as dependências de entrada de tarefas GET, elas também podem ajudar a identificar dependências cíclicas, normalmente frutos de erros nas definições das tarefas. Por outro lado, o custo de uma ordenação topológica em contexto distribuído pode ser elevado. É preciso estudar com cuidado essa e outras soluções.

Mesmo que o problema de ausência de dependência do GET ocorra, pode não ter implicação prática. Uma coisa é o domínio Y_1^1 depender de X_0^1, outra coisa é código que invoca as rotinas tipo *receive* do MPI em *GET* Y_1^1 dependerem de X_0^1. De qualquer forma, o *framework* precisa ser capaz de lidar com esses casos. Assim como também precisa implementar as dependências de dados do tipo WAR para resolver a anomalia do PUT.

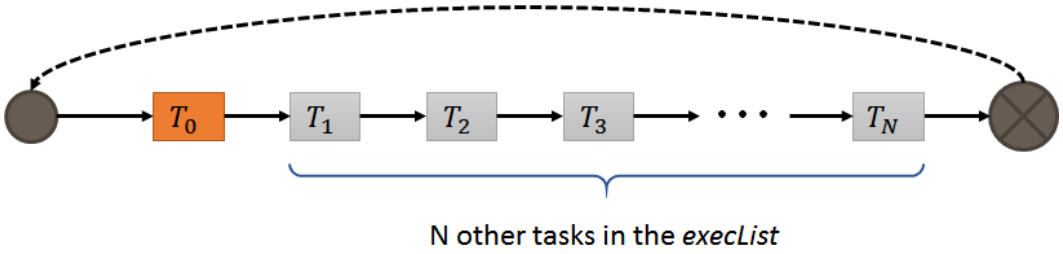
Figura 3.13: Exemplo de situação em que uma tarefa GET deveria ter uma dependência de entrada



Débitos Técnicos da fase de Execução

O principal débito tecnológico da fase de execução é o ***non-threaded*** *runtime*, único *runtime* disponível no *framework*. Atualmente, em cada *rank MPI*, todo o gerenciamento ocorre entre chamadas de tarefas do usuário. Quando uma tarefa T_0 não é concluída de imediato (situação normal para tarefas de comunicação), o *framework* prossegue processando as outras tarefas da lista de execução (Seção 3.3.3). Se houver muitas tarefas na lista, sem o auxílio de uma *thread*, o processo pode demorar a retornar à tarefa T_0. A Figura 3.14 ilustra o problema.

Figura 3.14: Ilustração da deficiência do *runtime* atual (sem *threads*)



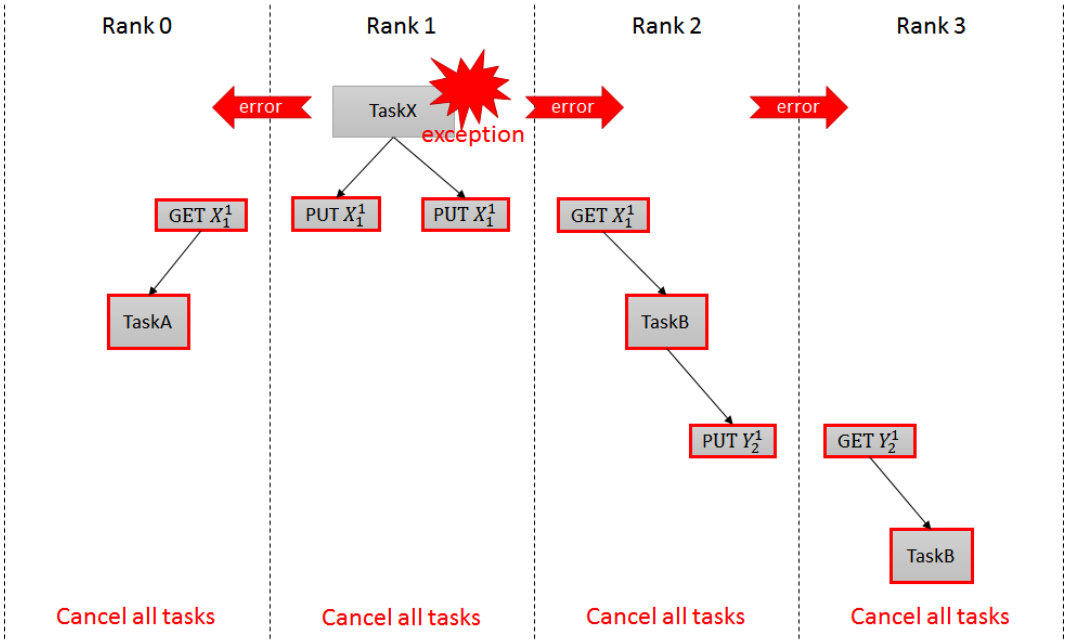
The time to process T_0 again (after the first time) is:
$tAgain_0 \geq \sum_{i=1}^{N} ExecTime(T_i)$

Suponha que o T_0 da figura seja uma tarefa *GET* semelhante a do Código 3.5, na Seção 3.3.1. Ela possui 3 *steps*, sendo que inicia um recebimento não bloqueante no *step 0* e outro no *step 1*. O último *step* é usado para consolidar os dados. Suponha que a primeira execução de T_0 tenha ocorrido no tempo *t0*, e que a mensagem do primeiro estágio seja completada pelo MPI no instante *t1*. Idealmente, o *step 1* deveria ser executado imediatamente após *t1*, mas na prática ele só será iniciado após todas as demais tarefas na lista de execução forem processadas (o $tAgain_0$ da figura). O tempo de processamento de muitas das outras tarefas pode ser pequeno (como uma simples checagem de status que retorna EXECUTING), porém outras podem executar um código com elevado custo computacional (como uma multiplicação de matrizes). Além disso, o número de tarefas na lista de execução é variável, pois quando uma tarefa T_i é concluída, outras tarefas T_js, que eventualmente tenham se tornado disponíveis, entram na lista no lugar de T_i (por isso o $\geq$ na figura). A depender do número de tarefas do algoritmo, $tAgain_0$ pode ser muito maior que *t1*, o que implica em atraso no processamento de outros *steps*. Para remover esses *overheads*, é preciso implementar um *runtime* mais sofisticado, que utilize *thread* para gerenciar ao menos as tarefas de comunicação. Estudos precisam ser feitos para definir a melhor tecnologia a ser empregada (threads C++, OpenMP, etc.).

Do ponto de visto da execução, outra *feature* ausente considerada importante para o *framework* é o suporte a cancelamento de tarefas. A Figura 3.15 ilustra

a importância do cancelamento. A tarefa *TaskX* da figura, executada no *rank 1*, lança uma exceção que impede a geração do domínio X_1^1. Porém, tanto o *rank 0* quanto o *rank 1* possuem uma tarefa *GET* X_1^1 que aguardam por um dado que jamais será enviado (devido à exceção). Hoje, essa situação provocaria um *deadlock*. Logo, o *framework* precisa implementar um mecanismo de propagação de exceção e cancelamento de tarefas. Na ocorrência de um erro, o *framework* preciso tratar a exceção e sinalizar o erro aos *ranks* dependentes recursivamente. Na mesma figura, o *rank 1* sinaliza o *rank 0* e o *rank 2*. O *rank 2*, por sua vez, sinaliza o *rank 3*. Ao receber uma sinalização de erro, o processo precisa cancelar todas as suas tarefas e o *dataflow* terminará sua execução com um status de erro. Um mecanismo semelhante ao descrito já foi implementado com sucesso em um protótipo do *framework*, o que simplifica a incorporação de tal funcionalidade.

Figura 3.15: Exemplo de cancelamento de tarefas e propagação de exceções



3.4 Aplicações em Álgebra Linear Computacional Esparsa

Alguns núcleos de Álgebra Linear esparsa foram implementados no SolverBR usando o *framework dataflow* para fins de avaliação.

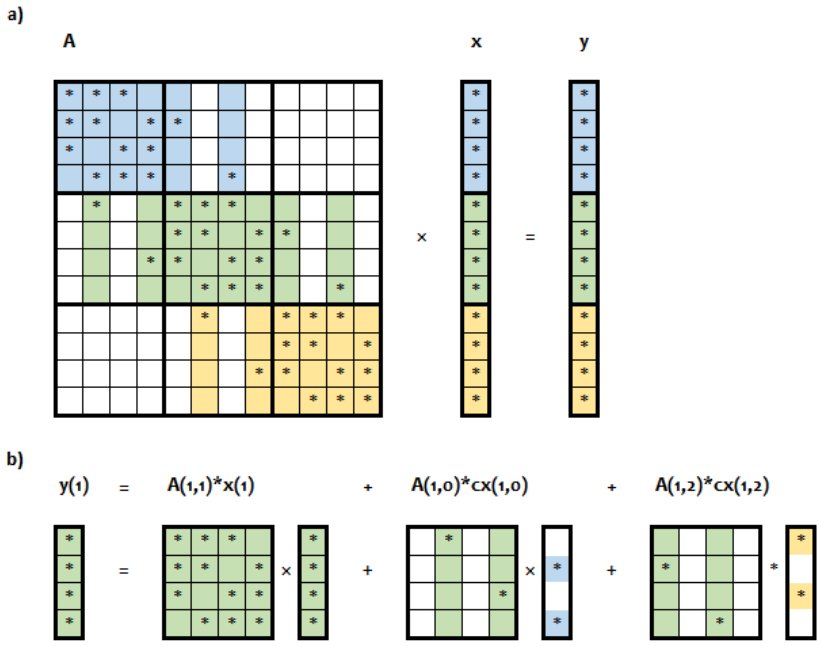
3.4.1 Multiplicação matriz-vetor: y = Ax

A multiplicação matriz-vetor, $y = A * x$, consiste em multiplicar um vetor denso distribuído x por uma matriz esparsa distribuída A, obtendo como resultado o vetor

denso distribuído y. Apesar de ser um algoritmo paralelo simples, há diversas formas de se implementar essa operação. A chave para um bom desempenho é tentar sobrepor a comunicação dos domínios de x com a tarefa que efetua a computação da submatriz diagonal $y_i = A_{i,i}x_i$. Para que essa sobreposição seja efetiva, primeiramente, os envios dos domínios de x precisam ser iniciados assincronamente antes do cômputo da diagonal. Além disso, é necessário enviar o mínimo de dados possível. É comum uma matriz de conexão $A_{i,j}$ possuir muitas colunas inteiramente nulas (Figura 2.10). Portanto, ao receber um x_j, basta que venham apenas os valores correspondentes aos índices de colunas não nulas de $A_{i,j}$. Ou seja, os domínios de x que são trocados entre os *ranks MPI*, na verdade são versões comprimidas de cada domínio com base nas colunas não nulas da submatriz que será multiplicada. A Figura 3.16 item (a) mostra uma multiplicação matriz-vetor com 3 *ranks MPI* explicitando as esparsidades de cada submatriz para ilustrar o uso de vetores comprimidos. O item (b) mostra as operações entre domínios tomando o *rank 1* como exemplo.

Duas versões de MatVec foram implementadas e serão detalhadas a seguir. O *dataflow* foi encapsulado na própria classe Matrix do SolverBR. Abaixo, segue uma descrição dos objetos e *closures* usados em ambos os algoritmos.

Figura 3.16: Exemplo de multiplicação matriz-vetor fazendo uso de vetores comprimidos em função das colunas não nulas das submatrizes de conexão



Objetos para o algoritmo *dataflow* MatVec:

- AObj: instância de InnerMatrixObject, realização de DataObject para a classe Matrix do SolverBR, que representa a matriz A. Seus domínios são indexados por dois índices $\{i,j\}$, e o seu *owner rank* é o processo i. Os domínios estendem LocalDomain (sem os métodos getTask e putTask) porque domínios de

matrizes não são comunicados nesse algoritmo.

- xObj: instância de InnerVectorObject, realização de DataObject para a classe Vector do SolverBR, que representa o vetor x. Seus domínios são indexados por um índice *{i}*, e o seu *owner rank* é o processo i. Os domínios realizam LocalDomain (também não são comunicados nesse algoritmo).
- yObj: instância de InnerVectorObject, realização de DataObject para a classe Vector do SolverBR, que representa o vetor y. Mesmas características de xObj.
- cxObj: instância de ApplyComprVectorObject, realização de DataObject que representa o vetor comprimido cx. Seus domínios são indexados por dois índices *{i,j}*, e o seu *owner rank* é o processo j. Um domínio *{i,j}* desse objeto representa o vetor x_j comprimido para as colunas não nulas de $A_{i,j}$, e realiza a classe Domain (implementando getTask e putTask para fins de comunicação).

Closures para o algoritmo *dataflow* MatVec:

- diagApplyClosure: instância de InnerDiagApply, realização de Closure para a multiplicação da submatriz diagonal $y_i = A_{i,i} * x_i$. Suas tarefas são indexadas por um índice *{i}*, usado para endereçar os domínios y_i, $A_{i,i}$ e x_i.
- connInnerOuterApplyClosure: instância de InnerOuterConnApply, realização de Closure para a multiplicação da submatriz de conexão (não diagonal) $y_i \mathrel{+}= A_{i,j} * cx_{i,j}$, acumulando o resultado em y_i. Suas tarefas são indexadas por dois índices *{i,j}*, usados para endereçar os domínios y_i, $A_{i,j}$ e $cx_{i,j}$.

Código 3.8: Algoritmo *dataflow* para multiplicação matriz-vetor (ordered)

```
1  m_dfOrderedApply( m_processingEnvironment.get(), "Dataflow Ordered Apply",
2                    [&]( CoarseDataflow::Graph& gph )
3  {
4     // Add data objects
5     auto A = gph.object( m_AObj.get() );
6     auto x = gph.object( m_xObj.get() );
7     auto y = gph.object( m_yObj.get() );
8     auto cx = gph.object( m_cxObj.get() );
9
10    // Add closures
11    auto yi_eq_Aii_xi     = gph.closure( m_diagApplyClosure.get() );
12    auto yi_acc_Aij_cxij = gph.closure( m_connInnerOuterApplyClosure.get() );
13
14    // Apply the diagonal matrix: y(i) = A(i,i)*x(i)
15    foreachDiagonalMatrix<INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aii )
16    {
17       int const ilp = Aii->iLocalPartition()->id();
18       gph.task( yi_eq_Aii_xi, { ilp } )
19          .in(    A, { ilp, ilp }, 0 )
20          .in(    x, { ilp },      0 )
21          .inout( y, { ilp },      0, 1 );
22    } );
23
24    int ys = 1;
25    // Apply the connections (off-diagonal) matrices: y(i) += A(i,j)*x(j), i != j
26    foreachConnectionMatrix<INNER,OUTER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
27    {
28       int const ilp = Aij->iLocalPartition()->id();
29       int const jlp = Aij->jLocalPartition()->id();
30       gph.task( yi_acc_Aij_cxij, { ilp, jlp } )
31          .in(    A, { ilp, jlp }, 0 )
32          .in(   cx, { ilp, jlp }, 0 ) // cx(i,j): compressed x(j) for A(i,j)*x(j)
33          .inout( y, { ilp },      ys, ys+1 );
34       ys++;
35    } );
36 } );
```

Definidos os objetos, domínios e *closures*, dois algoritmos de multiplicação matriz-vetor foram implementados. O Código 3.8 é chamado aqui de versão ***ordered*** da multiplicação. Isso porque na multiplicação das matrizes de conexão foi usada a construção *inout*, que impõe uma ordem às operações usando estados do domínio y_i. No mais, o algoritmo se comporta como descrito anteriormente. Tarefas tipo *PUT* e *GET* para vetores comprimidos serão criadas na fase de *setup*. E na hora da execução, essas tarefas aparecerão primeiro na lista, de modo que serão iniciadas antes da tarefa da diagonal. O Código 3.9 mostra o segundo algoritmo implementado, a versão ***relaxed***. Nele foi usado a construção *reduce* para a multiplicação das submatrizes de conexão, permitindo que as operações sejam realizadas conforme a chegada dos subvetores comprimidos.

A classe Matrix do SolverBR, tem funções para acessar a submatriz diagonal local de um *rank* (método foreachDiagonalMatrix<INNER>()) e também para percorrer as submatrizes ao longo da linha (método foreachConnectionMatrix<INNER,OUTER>()) ou da coluna (método foreachConnectionMatrix<OUTER,INNER>()) do *rank*. Essas rotinas são usadas para a definição das tarefas *dataflow* nas duas versões de MatVec.

Código 3.9: Algoritmo *dataflow* para multiplicação matriz-vetor (relaxed)

```
1  m_dfRelaxedApply( m_processingEnvironment.get(), "Dataflow Relaxed Apply",
2                    [&]( CoarseDataflow::Graph& gph )
3  {
4     // Add data objects
5     auto A = gph.object( m_AObj.get() );
6     auto x = gph.object( m_xObj.get() );
7     auto y = gph.object( m_yObj.get() );
8     auto cx = gph.object( m_cxObj.get() );
9
10    // Add closures
11    auto yi_eq_Aii_xi     = gph.closure( m_diagApplyClosure.get() );
12    auto yi_acc_Aij_cxij = gph.closure( m_connInnerOuterApplyClosure.get() );
13
14    // Apply the diagonal matrix: y(i) = A(i,i)*x(i)
15    foreachDiagonalMatrix<INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aii )
16    {
17       int const ilp = Aii->iLocalPartition()->id();
18       gph.task( yi_eq_Aii_xi, { ilp } )
19          .in(    A, { ilp, ilp }, 0 )
20          .in(    x, { ilp },      0 )
21          .inout( y, { ilp },      0, 1 );
22    } );
23
24    // Apply the connections (off-diagonal) matrices: y(i) += A(i,j)*x(j), i != j
25    foreachConnectionMatrix<INNER,OUTER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
26    {
27       int const ilp = Aij->iLocalPartition()->id();
28       int const jlp = Aij->jLocalPartition()->id();
29       gph.task( yi_acc_Aij_cxij, { ilp, jlp } )
30          .in(    A, { ilp, jlp }, 0 )
31          .in(   cx, { ilp, jlp }, 0 ) // cx(i,j): compressed x(j) for A(i,j)*x(j)
32          .reduce( y, { ilp },     1, 2 );
33    } );
34 } );
```

Figura 3.17: Grafo de execução da multiplicação matriz-vetor versão ***ordered*** para o *rank 1*

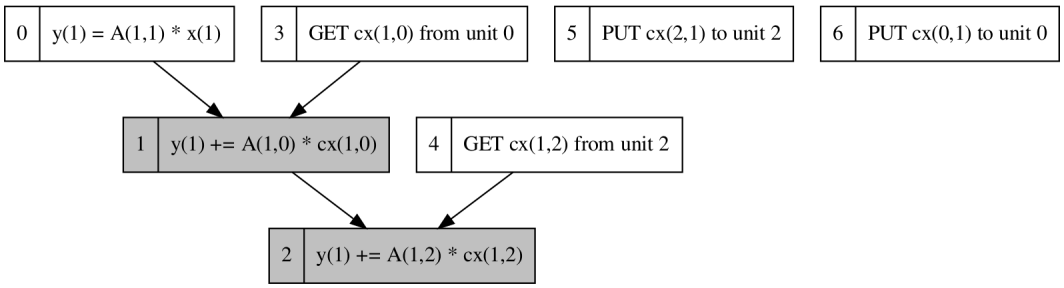
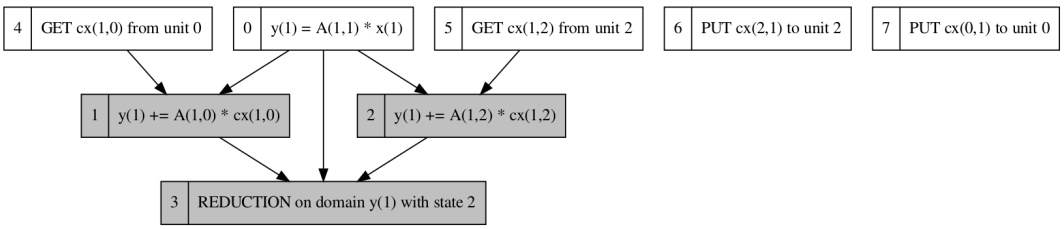


Figura 3.18: Grafo de execução da multiplicação matriz-vetor versão ***relaxed*** para o *rank 1*



As Figuras 3.17 e 3.18 trazem o Grafo de Execução das duas versões de multiplicação matriz-vetor (***ordered*** e ***relaxed***, respectivamente) no ***rank 1***, para uma execução com 3 *ranks MPI* e uma matriz com esparsidade de domínios como a do

exemplo 3.1, item (a).

3.4.2 Multiplicação matriz transposta-vetor: $\mathbf{y = A^T x}$

A multiplicação matriz transposta-vetor, $y = A^T * x$, consiste em multiplicar um vetor denso distribuído x pela transposta (logicamente) de uma matriz esparsa distribuída A, obtendo como resultado o vetor denso distribuído y. Algumas das dificuldades desse algoritmo foram discutidas na Seção 3.2.4. A exemplo da multiplicação matriz-vetor comum, a chave para um bom desempenho aqui é sobrepor a comunicação dos resultados dos produtos parciais com a multiplicação da submatriz diagonal $y_i = A^T_{i,i} * x_i$. Da mesma forma, os resultados dos produtos parciais das submatrizes de conexão $A^T_{i,j}$ que precisam ser enviados são somente os valores correspondentes aos índices das colunas não nulas de $A^T_{i,j}$ (os valores nos demais índices serão sempre zero). Por isso, foi criado o DataObject *cy*, cujos domínios são indexados por dois índices *{i,j}*, onde $cy_{i,j}$ representa o resultado de $A^T_{i,j} * x_j$ comprimido para as colunas não nulas de $A^T_{i,j}$, com i *!=* j.

Objetos para o algoritmo *dataflow* Transp. MatVec:

- AObj: mesmo objeto A do MatVec. Como aqui o A é transposto logicamente, o *owner rank* do domínio $A^T_{i,j}$ é o mesmo de $A_{j,i}$, ou seja, o *rank j*.
- xObj: mesmo objeto x do MatVec.
- yObj: mesmo objeto y do MatVec.
- cyObj: instância de ApplyTComprVectorObject, realização de DataObject que representa o vetor comprimido *cy*. Seus domínios são indexados por dois índices *{i,j}*, e o seu *owner rank* é o processo *j*. Um domínio *{i,j}* desse objeto representa o vetor resultado da operação $A^T_{i,j} * x_j$ comprimido para as colunas não nulas de $A^T_{i,j}$ (que, na verdade, são as linhas não nulas de $A_{j,i}$), e realiza a classe Domain (implementando getTask e putTask para fins de comunicação).

Closures para o algoritmo *dataflow* Transp. MatVec:

- diagApplyTClosure: instância de InnerDiagApplyT, realização de Closure para a multiplicação da submatriz diagonal $y_i = A^T_{i,i} * x_i$. Suas tarefas são indexadas por um índice *{i}*, usado para endereçar os domínios y_i, $A^T_{i,i}$ e x_i.
- connInnerOuterApplyTClosure: instância de InnerOuterConnApplyT, realização de Closure para a multiplicação da submatriz de conexão (não diagonal) $cy_{i,j} = A^T_{i,j} * x_j$. Suas tarefas são indexadas por dois índices *{i,j}*.
- connOuterInnerApplyTClosure: instância de OuterInnerConnApplyT, realização de Closure para acumulação de um $cy_{i,j}$ recebido e *rank* remoto em y_i. Suas tarefas são indexadas por dois índices *{i,j}*.

Definidos os DataObjects, Domains e Closures, foram implementadas quatro versões diferentes do Transp. MatVec para fins de teste:

- Transp. MatVec Ordered - utiliza a construção *inout* nas operações y_i += $cy_{i,j}$, forçando uma ordem específica de acumulação dos $cy_{i,j}$. ver Cófigo 3.10
- Transp. MatVec Relaxed - utiliza a construção *reduce* nas operações y_i += $cy_{i,j}$, permitindo que a acumulação dos $cy_{i,j}$ seja feita em ordem de acordo com a chegada do dado. Ver Código 3.11
- Transp. MatVec Ordered with Subgraphs (SG) - similar à versão Transp. MatVec Ordered, mas usa *generator* para os domínios $cy_{i,j}$ juntamente com a heurística de subgrafos. Ver Código 3.12. O *generator* está definido nas linhas 38-46, e o uso de subgrafos na linha 2 (parâmetro ***true***).
- Transp. MatVec Relaxed with Subgraphs (SG) - similar à versão Transp. MatVec Relaxed, mas usa *generator* para os domínios $cy_{i,j}$ juntamente com a heurística de subgrafos. Ver Código 3.13. O *generator* está definido nas linhas 36-44, e o uso de subgrafos na linha 2 (parâmetro ***true***).

Nos dois algoritmos sem subgrafos, o usuário precisa criar os domínios locais $cy^1_{i,j}$ no estado 1 explicitamente, percorrendo os domínios A^T*, j (linhas 16-24 de 3.10 e 3.11). É importante que as tarefas que criam esses domínios sejam definidas antes da tarefa que multiplica a submatriz diagonal. Dessa forma, essas tarefas entrarão primeiro na lista de execução e os respectivos PUTs serão iniciados antes do cálculo da diagonal, permitindo a sobre posição de computação com comunicação. Nos algoritmos com generators+subgrafos, a própria heurística se encarregará de executar as tarefas $cy_{i,j} = A^T_{i,j} * x_j$ antes da diagonal, pois elas estão em um subgrafo prioritário, que privilegia as tarefas ligadas às $PUTcy^1_{i,j}$. As Figuras 3.19 e 3.20 trazem o Grafo de Execução das versões Ordered e Relaxed com Subgrafos, respectivamente, para o *rank 1*, considerando uma matriz similar à do exemplo na Figura 3.9, item (a).

Código 3.10: Algoritmo Transp. MatVec Ordered

```
1  m_dfOrderedApplyT( m_processingEnvironment.get(), "Dataflow Ordered ApplyT",
2                     false, [&]( CoarseDataflow::Graph& gph )
3  {
4     // Add data objects
5     auto A = gph.object( m_AObj.get() );
6     auto x = gph.object( m_xObj.get() );
7     auto y = gph.object( m_yObj.get() );
8     auto cy = gph.object( m_cyObj.get() );
9
10    // Add closures
11    auto yi_eq_TAii_xi    = gph.closure( m_diagApplyTClosure.get() );
12    auto yj_acc_cyij      = gph.closure( m_connOuterInnerApplyTClosure.get() );
13    auto cyij_eq_TAij_xi  = gph.closure( m_connInnerOuterApplyTClosure.get() );
14
15    // Apply the cy(i,j) = At(i,j)*x(j), for i != j
16    foreachConnectionMatrix<INNER,OUTER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
17    {
18       int const ilp = Aij->iLocalPartition()->id();
19       int const jlp = Aij->jLocalPartition()->id();
20       gph.task( cyij_eq_TAij_xi, { ilp, jlp } )
21          .in(    A, { ilp, jlp }, 0 )
22          .in(    x, { ilp },      0 )
23          .out( cy, { ilp, jlp }, 1 );
24    } );
25
26    // Apply the diagonal matrix: y(i) = At(i,i)*x(i)
27    foreachDiagonalMatrix<INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aii )
28    {
29       int const ilp = Aii->iLocalPartition()->id();
30       gph.task( yi_eq_TAii_xi, { ilp } )
31          .in(    A, { ilp, ilp }, 0 )
32          .in(    x, { ilp },      0 )
33          .out(  y, { ilp },      1 );
34    } );
35
36    int ys = 1;
37    // Accumulate the compressed y's: y(i) += cy(i,j), i != j
38    foreachConnectionMatrix<OUTER,INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
39    {
40       int const ilp = Aij->iLocalPartition()->id();
41       int const jlp = Aij->jLocalPartition()->id();
42       gph.task( yj_acc_cyij, { ilp, jlp } )
43          .in(    cy, { ilp, jlp }, 1 )
44          .inout( y, { jlp },      ys, ys+1 );
45       ys++;
46    } );
47 } );
```

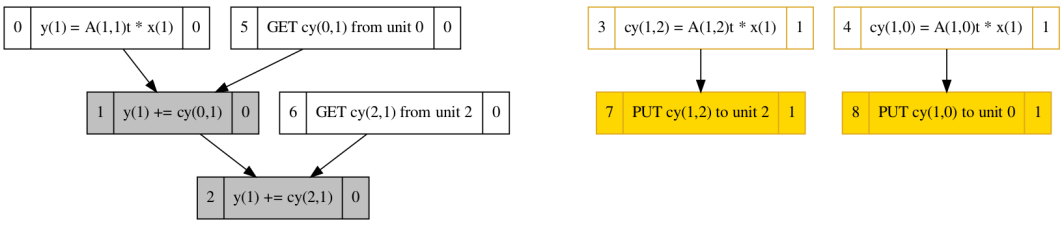
Código 3.11: Algoritmo Transp. MatVec Relaxed

```
1  m_dfRelaxedApplyT( m_processingEnvironment.get(), "Dataflow Relaxed ApplyT",
2                     false, [&]( CoarseDataflow::Graph& gph )
3  {
4     // Add data objects
5     auto A = gph.object( m_AObj.get() );
6     auto x = gph.object( m_xObj.get() );
7     auto y = gph.object( m_yObj.get() );
8     auto cy = gph.object( m_cyObj.get() );
9
10    // Add closures
11    auto yi_eq_TAii_xi    = gph.closure( m_diagApplyTClosure.get() );
12    auto yj_acc_cyij      = gph.closure( m_connOuterInnerApplyTClosure.get() );
13    auto cyij_eq_TAij_xi  = gph.closure( m_connInnerOuterApplyTClosure.get() );
14
15    // Apply the cy(i,j) = At(i,j)*x(j), for i != j
16    foreachConnectionMatrix<INNER,OUTER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
17    {
18       int const ilp = Aij->iLocalPartition()->id();
19       int const jlp = Aij->jLocalPartition()->id();
20       gph.task( cyij_eq_TAij_xi, { ilp, jlp } )
21          .in(   A, { ilp, jlp }, 0 )
22          .in(   x, { ilp },      0 )
23          .out( cy, { ilp, jlp }, 1 );
24    } );
25
26    // Apply the diagonal matrix: y(i) = At(i,i)*x(i)
27    foreachDiagonalMatrix<INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aii )
28    {
29       int const ilp = Aii->iLocalPartition()->id();
30       gph.task( yi_eq_TAii_xi, { ilp } )
31          .in(   A, { ilp, ilp }, 0 )
32          .in(   x, { ilp },      0 )
33          .out(  y, { ilp },      1 );
34    } );
35
36    // Accumulate the compressed y's: y(i) += cy(i,j), i != j
37    foreachConnectionMatrix<OUTER,INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
38    {
39       int const ilp = Aij->iLocalPartition()->id();
40       int const jlp = Aij->jLocalPartition()->id();
41       gph.task( yj_acc_cyij, { ilp, jlp } )
42          .in(   cy, { ilp, jlp }, 1 )
43          .reduce( y, { jlp },     1, 2 );
44    } );
45 } );
```

Código 3.12: Algoritmo Transp. MatVec Ordered with Subgraphs (SG)

```
1  m_dfOrderedApplyT( m_processingEnvironment.get(), "Dataflow Ordered ApplyT [SUB-GRAPH]",
2                     true, [&]( CoarseDataflow::Graph& gph )
3  {
4     // Add data objects
5     auto A = gph.object( m_AObj.get() );
6     auto x = gph.object( m_xObj.get() );
7     auto y = gph.object( m_yObj.get() );
8     auto cy = gph.object( m_cyObj.get() );
9
10    // Add closures
11    auto yi_eq_TAii_xi    = gph.closure( m_diagApplyTClosure.get() );
12    auto yj_acc_cyij      = gph.closure( m_connOuterInnerApplyTClosure.get() );
13    auto cyij_eq_TAij_xi  = gph.closure( m_connInnerOuterApplyTClosure.get() );
14
15    // Apply the diagonal matrix: y(i) = At(i,i)*x(i)
16    foreachDiagonalMatrix<INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aii )
17    {
18       int const ilp = Aii->iLocalPartition()->id();
19       gph.task( yi_eq_TAii_xi, { ilp } )
20          .in(    A, { ilp, ilp }, 0 )
21          .in(    x, { ilp },      0 )
22          .out(   y, { ilp },      1 );
23    } );
24
25    int ys = 1;
26    // Accumulate the compressed y's: y(i) += cy(i,j), i != j
27    foreachConnectionMatrix<OUTER,INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
28    {
29       int const ilp = Aij->iLocalPartition()->id();
30       int const jlp = Aij->jLocalPartition()->id();
31       gph.task( yj_acc_cyij, { ilp, jlp } )
32          .in(    cy, { ilp, jlp }, 1 )
33          .inout( y, { jlp },      ys, ys+1 );
34       ys++;
35    } );
36
37    // Generate the cy(i,j) on demand
38    gph.generator( cy, 0, 1, [=]( const ids_t& ids ) mutable
39    {
40       int const ilp = ids[ 0 ];
41       int const jlp = ids[ 1 ];
42       gph.task( cyij_eq_TAij_xi, { ilp, jlp } )
43          .in(    A, { ilp, jlp }, 0 )
44          .in(    x, { ilp },      0 )
45          .out( cy, { ilp, jlp }, 1 );
46    } );
47 } );
```

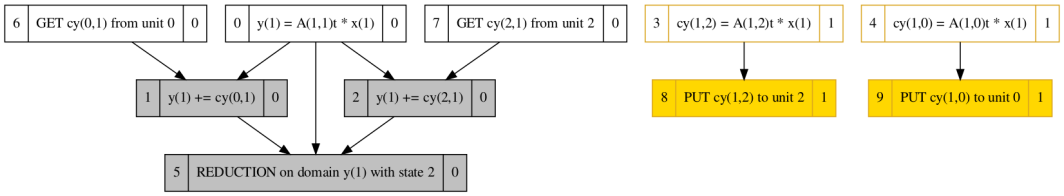
Figura 3.19: Grafo de Execução do algoritmo Transp. MatVec Ordered with Subgraphs (SG) do *rank 1*



Código 3.13: Algoritmo Transp. MatVec Relaxed with Subgraphs (SG)

```
1  m_dfRelaxedApplyT( m_processingEnvironment.get(), "Dataflow Relaxed ApplyT [SUB-GRAPH]",
2                     true, [&]( CoarseDataflow::Graph& gph )
3  {
4     // Add data objects
5     auto A = gph.object( m_AObj.get() );
6     auto x = gph.object( m_xObj.get() );
7     auto y = gph.object( m_yObj.get() );
8     auto cy = gph.object( m_cyObj.get() );
9
10    // Add closures
11    auto yi_eq_TAii_xi    = gph.closure( m_diagApplyTClosure.get() );
12    auto yj_acc_cyij      = gph.closure( m_connOuterInnerApplyTClosure.get() );
13    auto cyij_eq_TAij_xi  = gph.closure( m_connInnerOuterApplyTClosure.get() );
14
15    // Apply the diagonal matrix: y(i) = At(i,i)*x(i)
16    foreachDiagonalMatrix<INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aii )
17    {
18       int const ilp = Aii->iLocalPartition()->id();
19       gph.task( yi_eq_TAii_xi, { ilp } )
20          .in(   A, { ilp, ilp }, 0 )
21          .in(   x, { ilp },      0 )
22          .out(  y, { ilp },      1 );
23    } );
24
25    // Accumulate the compressed y's: y(i) += cy(i,j), i != j
26    foreachConnectionMatrix<OUTER,INNER>( [&]( const boost::shared_ptr<LocalMatrix>& Aij )
27    {
28       int const ilp = Aij->iLocalPartition()->id();
29       int const jlp = Aij->jLocalPartition()->id();
30       gph.task( yj_acc_cyij, { ilp, jlp } )
31          .in(   cy, { ilp, jlp }, 1 )
32          .reduce( y, { jlp },     1, 2 );
33    } );
34
35    // Generate the cy(i,j) on demand
36    gph.generator( cy, 0, 1, [=]( const ids_t& ids ) mutable
37    {
38       int const ilp = ids[ 0 ];
39       int const jlp = ids[ 1 ];
40       gph.task( cyij_eq_TAij_xi, { ilp, jlp } )
41          .in(   A, { ilp, jlp }, 0 )
42          .in(   x, { ilp },      0 )
43          .out( cy, { ilp, jlp }, 1 );
44    } );
45 } );
```

Figura 3.20: Grafo de Execução do algoritmo Transp. MatVec Relaxed with Subgraphs (SG) do *rank 1*



3.4.3 Produto triplo de matrizes esparsas: $\mathbf{C = P^T AP}$

O produto triplo de matrizes esparsas, $C = P^T * A * P$, consiste na multiplicação de três matrizes $P^T_{M,N}$, $A_{N,N}$ e $P_{N,M}$, onde $M < N$ e P^T é o P transposto logicamente. Esse núcleo é usado atualmente em um precondicionador Multiescala no GeomecBR (FIGUEIREDO *et al.* [40]). Em termos de domínios, P tem a mesma esparsidade de A. O algoritmo implementado efetua $C = P^T * (A * P)$. A Figura 3.21, item (a)

mostra um exemplo de $A * P$ com 3 *ranks MPI*. O item (b) detalha as operações realizadas para gerar os domínios de AP no *rank 1*. A Figura 3.22, item (a), ilustra a parte $P^T * AP$. O item (b) detalha as operações realizadas para gerar os domínios de C no *rank 1*.

Figura 3.21: Exemplo da parte A*P com 3 *ranks MPI*

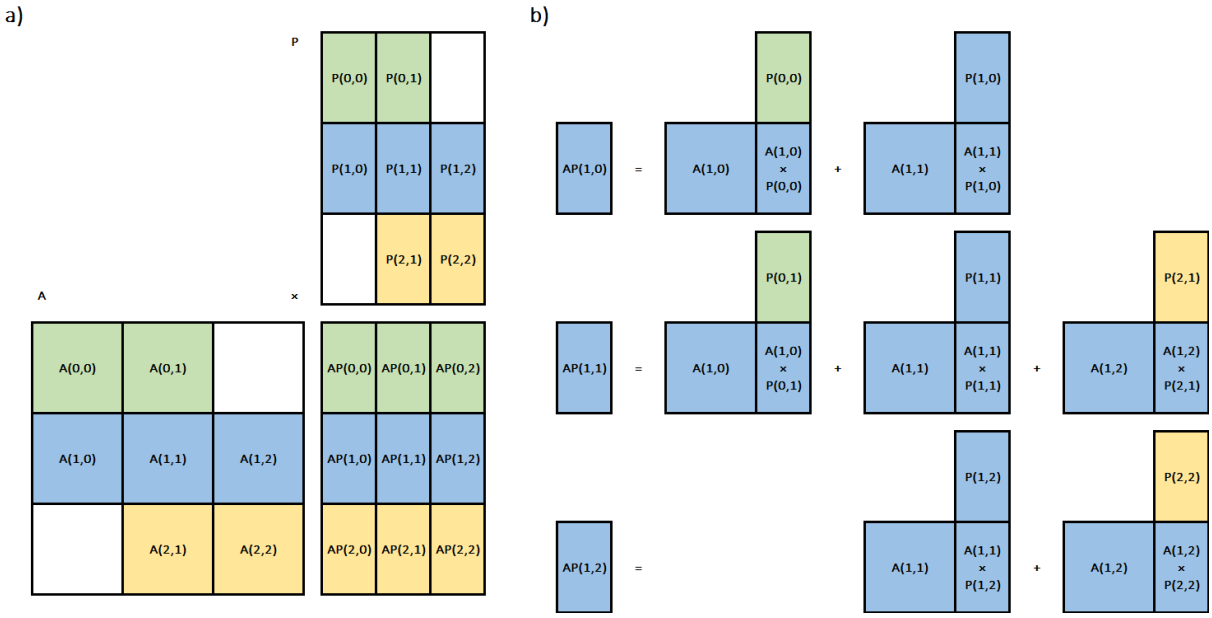
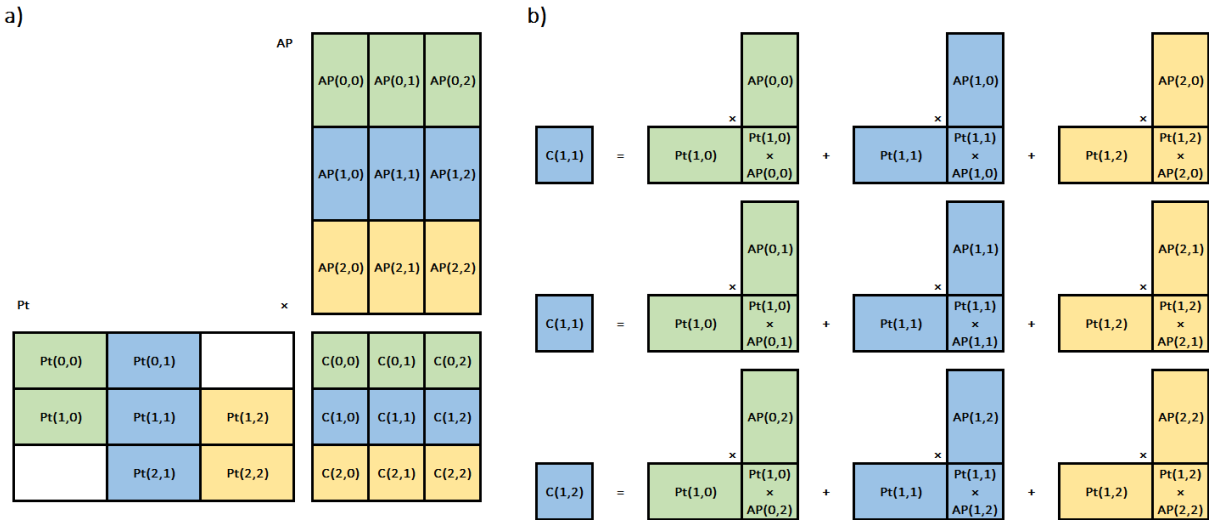


Figura 3.22: Exemplo da parte $P^T * AP$ com 3 *ranks MPI*



Durante o desenvolvimento do *framework*, foram implementados vários PtAPs diferentes, mas pela complexidade do algoritmo, será descrita apenas a última versão, que executa a operação $C = P^T * (A * P)$ (ou seja, primeiro $A * P$ e depois $P^T * AP$, tudo em um único *dataflow*).

Relizações de DataObject para o PtAP:

- A: instância de MatObj, realização de DataObject para matrizes, encapsulando a matriz distribuída A. Seus domínios são indexados por dois índices *{i,j}*, e seu *owner rank* é o processo i.
- P: instância de MatObj, realização de DataObject para matrizes, encapsulando a matriz distribuída P. Seus domínios são indexados por dois índices *{i,j}*, e seu *owner rank* é o processo i.
- Pt: instância de MatObj, realização de DataObject para matrizes, encapsulando a matriz distribuída P^T, que é a P logicamente transposta. Seus domínios são indexados por dois índices *{i,j}*, e seu *owner rank* é o processo j.
- AP: instância de MatObj, realização de DataObject para matrizes, encapsulando a matriz distribuída AP, resultante da multiplicação $AP = A * P$. Seus domínios são indexados por dois índices *{i,j}*, e seu *owner rank* é o processo i.
- PtAP: instância de MatObj, realização de DataObject para matrizes, encapsulando a matriz distribuída $PtAP$ ou C, resultante da multiplicação $P^T AP = P^T * AP$. Seus domínios são indexados por dois índices *{i,j}*, e seu *owner rank* é o processo i.
- partialAP: instância de PartialMatMatMulObj, realização de DataObject para resultados parciais de multiplicação de duas matrizes, no caso A e P. Seus domínios são indexados por três índices *{i,k,j}*, e representam os dados da multiplicação parcial $A_{i,k} * P_{k,j}$. Seu *owner rank* é o **processo i**.
- partialPtAP: instância de PartialMatMatMulObj, realização de DataObject para resultados parciais de multiplicação de duas matrizes, no caso P^T e AP. Seus domínios são indexados por três índices *{i,k,j}*, e representam os dados da multiplicação parcial $P^T_{i,k} * AP_{k,j}$. Seu *owner rank* é o **processo k**.
- compressedP: instância de ComprMatObj, realização de DataObject para a matriz P comprimida segundo as colunas não nulas de submatrizes de A. Seus domínios são indexados por três índices *{i,k,j}*, e representam $P_{k,j}$ comprimido segundo as colunas não nulas de $A_{i,k}$. Seu *owner rank* é o processo k.

Os domínios da classes MatObj, PartialMatMatMulObj e ComprMatObj implementam os métodos *getTask* e *putTask*, e são aptos para movimentação de dados entre os *ranks*. No entanto, apenas domínios de *AP*, *compressedP* e *partialPtAP* são, de fato, comunicados no algoritmo implementado.

O algoritmo do PtAP tem duas etapas: uma etapa **simbólica** (cujo objetivo é determinar o padrão de esparsidade da matriz final C e da matriz intermediária AP) e uma etapa **numérica** (que realiza todos os cálculos numéricos já conhecendo a esparsidade de todas as matrizes envolvidas). Isso acontece porque em muitas aplicações científicas os valores de uma matriz podem mudar muitas vezes sem que o padrão de esparsidade seja alterado. Nessa situação, a aplicação pode fazer a etapa simbólica, que é mais complexa e custosa, apenas uma vez. Por simplificação, e

também para evitar o *overhead* de ter dois *dataflows* distintos, as duas etapas foram descritas em um mesmo *dataflow*. No entanto, as *closures* têm comportamento distinto, a depender da etapa em vigor.

Realizações de *Closure* para o PtAP:

- genAP: instância de GenerateAP, realização da classe Closure para criação (via *generator*) do objeto *AP*. Suas tarefas são indexadas por dois índices *{i,j}*, e representam a consolidação do domínio $AP^2_{i,j}$ (o estado 2 representa o domínio consolidado). Na etapa **simbólica**, todas as multiplicações parciais $partialAP_{i,k,j}$ (que representam os resultados de $A_{i,k} * P_{k,j}$) são executados simbolicamente em um único passo gerando a esparsidade final de $A_{i,j}$. Na etapa **numérica**, a tarefa não faz nada, pois a parte numérica das multiplicações parciais já terão sido executadas pela *closure addPartialAP*.
- genPartialAP: instância de GeneratePartialAP, realização da classe Closure para criação (via *generator*) do objeto *partialAP*, que possui os operandos necessários à multiplicação. Suas tarefas são indexadas por três índices *{i,k,j}*, e representam a criação do domínio $partialAP^1_{i,k,j}$ (o estado 1 indica que $partialAP_{i,k,j}$ está pronto, ou seja, seus operandos $A_{i,k}$ e $P_{k,j}$ estão disponíveis).
- addPartialAP: instância de AddPartialMatMatMul, realização da classe Closure que trata a acumulação de um domínio de *partialAP* ao domínio *AP* correspondente. Suas tarefas são indexadas por três índices *{i,k,j}*, e representam a operação $AP_{i,j}$ += $partialAP_{i,k,j}$. Seu comportamento depende da etapa do algoritmo. Na etapa **simbólica**, o domínio $partialAP_{i,k,j}$ é apenas inserido em uma lista de $AP_{i,j}$, sem realizar qualquer cálculo. Na etapa **numérica**, a esparsidade final de $AP_{i,j}$ estará definida, então multiplicação parcial $A_{i,k} * P_{k,j}$ é realizada *inplace* (com o resultado sendo acumulado diretamente nas entradas de $AP_{i,j}$).
- genComprP: instância de GenerateCompressedP, realização da classe Closure para criação de objetos *compressedP*. Suas tarefas são indexadas por três índices *{i,k,j}*, e representam o $P_{k,j}$ comprimido segundo as colunas não nulas de $A_{i,k}$. Quando $P_{k,j}$ e $A_{i,k}$ têm o mesmo *owner rank* (ou seja, $i = k$), $compressedP_{k,j}$ apenas referencia $P_{k,j}$ (sem comprimir). Quando os *owners* são diferentes, um *storage compressed* é de fato instanciado (simbolicamente ou numericamente) para as comunicações.
- genPtAP: instância de GeneratePtAP, realização da classe Closure para criação (via *generator*) do objeto *PtAP* (ou *C*). Suas tarefas são indexadas por dois índices *{i,j}*, e representam a consolidação do domínio $P^T AP^2_{i,j}$ (o estado 2 representa o domínio consolidado). Na etapa **simbólica**, todas as multipli-

cações parciais $partialPtAP_{i,k,j}$ (que representam os resultados de $P^T_{i,k} * AP_{k,j}$) são executados simbolicamente em um único passo gerando a esparsidade final de $P^T AP_{i,j}$. Na etapa **numérica**, a tarefa não faz nada, pois a parte numérica das multiplicações parciais já terão sido executadas pela *closure addPartialPtAP*.

- genPartialPtAP: instância de GeneratePartialPtAP, realização da classe Closure para criação (via *generator*) do objeto *partialPtAP*, que possui os operandos necessários à multiplicação. Suas tarefas são indexadas por três índices *{i,k,j}*, e representam a criação do domínio $partialPtAP^1_{i,k,j}$ (o estado 1 indica que $partialPtAP_{i,k,j}$ está pronto, ou seja, seus operandos $P^T_{i,k}$ e $AP_{k,j}$ estão disponíveis). Como P^T é o P logicamente transposto, os *owner ranks* são distribuídos por colunas. Com isso, o *owner rank* de $partialPtAP_{i,k,j}$ pode ser diferente do *owner* de $P^T AP_{i,j}$ (ver Figura 3.22). Nessa situação, o cálculo (seja **simbólico** ou **numérico**) é feito e armazenado em um *storage* local para posterior comunicação (via tarefas PUT e GET).
- addPartialPtAP: instância de AddPartialMatMatMul, realização da classe Closure que trata a acumulação de um domínio de *partialPtAP* ao domínio *PtAP* correspondente. Suas tarefas são indexadas por três índices *{i,k,j}*, e representam a operação $P^T AP_{i,j}$ += $partialPtAP_{i,k,j}$. Seu comportamento depende da etapa do algoritmo. Na etapa **simbólica**, o domínio $partialPtAP_{i,k,j}$ é apenas inserido em uma lista de $AP_{i,j}$, sem realizar qualquer cálculo. Na etapa **numérica**, a esparsidade final de $P^T AP_{i,j}$ estará definida, então multiplicação parcial $P^T_{i,k} * AP_{k,j}$ é realizada *inplace* (com o resultado sendo acumulado diretamente nas entradas de $P^T AP_{i,j}$). No caso, do domínio $partialPtAP_{i,k,j}$ ser remoto, ele já irá conter o resultado parcial pronto (que foi computado do *owner rank*, na *closure* genPartialPtAP, e enviado), então as entradas do resultado são acumuladas em $P^T AP_{i,j}$.

Com base nessas realizações de Closure e nas realizações de DataObject (e respectivos Domains), o grafo dataflow para operação PtAP foi definido conforme o Código 3.14. O mesmo *dataflow* pode ser usado com ou sem a política de subgrafos, de acordo com o terceiro parâmetro, *useSG* (variável *booleana*), passado ao construtor da instância de CoarseDataflow. Quando *useSG* é *true*, usa-se subgrafos. Quando *useSG* é *false*, usa-se a política normal (ou *full graph*). Opcionalmente, há um construtor sem esse parâmetro, que assume o valor *false* internamente.

Nas linhas 13-20, a cláusula *reduce* garante que o estado de $AP_{i,j}$ só vai a 1 quando todos os produtos parciais $partialAP_{i,k,j}$ foram adicionados ao domínio $AP_{i,j}$. Na etapa **simbólica**, isso significa que eles estão inseridos em uma lista de $AP_{i,j}$. Na etapa **numérica** os produtos parciais são executados *inplace* $AP_{i,j}$ += $A_{i,k} * P_{k,j}$. Nas linhas 22-27, cada $AP_{i,j}$ é consolidado (estado 2).

Código 3.14: Algoritmo PtAP (com ou sem subgrafos)

```
1  m_PtAP( m_processingEnvironment.get(), PtAP", useSG, [&]( CoarseDataflow::Graph& gph )
2  {
3     // A * P objects and closures
4     int obj_AP = gph.object( m_AP.get() );
5     int obj_partialAP = gph.object( m_partialAP.get() );
6     int obj_compP = gph.object( m_compressedP.get() );
7     int tsk_AP = gph.closure( m_genAP.get() );
8     int tsk_partialAP = gph.closure( m_genPartialAP.get() );
9     int tsk_addPartialAP = gph.closure( m_addPartialAP.get() );
10    int tsk_compP = gph.closure( m_genComprP.get() );
11
12    // Add partials A(i,k)*P(k,j) to AP(i,j) list
13    m_AP->rowPartition()->foreachLocalPartitionId<INNER>( [&]( const int i )
14    {
15       for ( int j : m_AP->nonZeroColumns( i ) )
16          for ( int k : m_AP->intersection( i, j ) )
17             gph.task( tsk_addPartialAP, { i, k, j } )
18                .in( obj_partialAP, { i, k, j }, 1 )
19                .reduce( obj_AP, { i, j }, 0, 1 );
20    } );
21    // Consolidate the AP(i,j) with all partials A(i,k)*P(k,j) in its list
22    m_AP->rowPartition()->foreachLocalPartitionId<INNER>( [&]( const int i )
23    {
24       for ( int j : m_AP->nonZeroColumns( i ) )
25          gph.task( tsk_AP, { i, j } )
26             .inout( obj_AP, { i, j }, 1, 2 );
27    } );
28    // Generate partial A(i,k) * P(k,j) [the P may be compressed]
29    gph.generator( obj_partialAP, 0, 1, [=]( const ids_t& ids ) mutable
30    {
31       int i, k, j;
32       ids.tie( i, k, j );
33       gph.task( tsk_partialAP, { i, k, j } )
34          .in( obj_compP, { i, k, j }, 1 )
35          .out( obj_partialAP, { i, k, j }, 1 );
36    } );
37    // Generate the compressed P(k,j) for the partial A(i,k)*P(k,j)
38    gph.generator( obj_compP, 0, 1, [=]( const ids_t& ids ) mutable
39    {
40       int i, k, j;
41       ids.tie( i, k, j );
42       gph.task( tsk_compP, { i, k, j } )
43          .out( obj_compP, { i, k, j }, 1 );
44    } );
45
46    // Pt * AP objects and closures
47    int obj_PtAP = gph.object( m_PtAP.get() );
48    int obj_partialPtAP = gph.object( m_partialPtAP.get() );
49    int tsk_PtAP = gph.closure( m_genPtAP.get() );
50    int tsk_partialPtAP = gph.closure( m_genPartialPtAP.get() );
51    int tsk_addPartialPtAP = gph.closure( m_addPartialPtAP.get() );
52
53    // Add partials Pt(i,k)*AP(k,j) to PtAP(i,j) list
54    m_PtAP->rowPartition()->foreachLocalPartitionId<INNER>( [&]( const int i )
55    {
56       for ( int j : m_PtAP->nonZeroColumns( i ) )
57          for ( int k : m_PtAP->intersection( i, j ) )
58             gph.task( tsk_addPartialPtAP, { i, k, j } )
59                .in( obj_partialPtAP, { i, k, j }, 1 )
60                .reduce( obj_PtAP, { i, j }, 0, 1 );
61    } );
62    // Consolidate the PtAP(i,j) with all partials Pt(i,k)*AP(k,j) in its list
63    m_PtAP->rowPartition()->foreachLocalPartitionId<INNER>( [&]( const int i )
64    {
65       for ( int j : m_PtAP->nonZeroColumns( i ) )
66          gph.task( tsk_PtAP, { i, j } )
67             .inout( obj_PtAP, { i, j }, 1, 2 );
68    } );
69    // Generate partial Pt(i,k) * AP(k,j)
70    gph.generator( obj_partialPtAP, 0, 1, [=]( const ids_t& ids ) mutable
71    {
72       int i, k, j;
73       ids.tie( i, k, j );
74       gph.task( tsk_partialPtAP, { i, k, j } )
75          .in( obj_AP, { k, j }, 2 )
76          .out( obj_partialPtAP, { i, k, j }, 1 );
77    } );
78 } );
```

Na etapa **simbólica**, todos os $partialAP_{i,k,j}$ previamente inseridos na lista de $AP_{i,j}$ são operados simbolicamente para obter, de uma só vez, a esparsidade final de $AP_{i,j}$. Na etapa numérica, a tarefa de consolidação não executa qualquer operação. O mesmo raciocínio vale para a consolidação do $P^T AP$ (linhas 54-61 e 63-68).

Figura 3.23: Grafo de Execução do exemplo de PtAP para o *rank 1*, parte 1

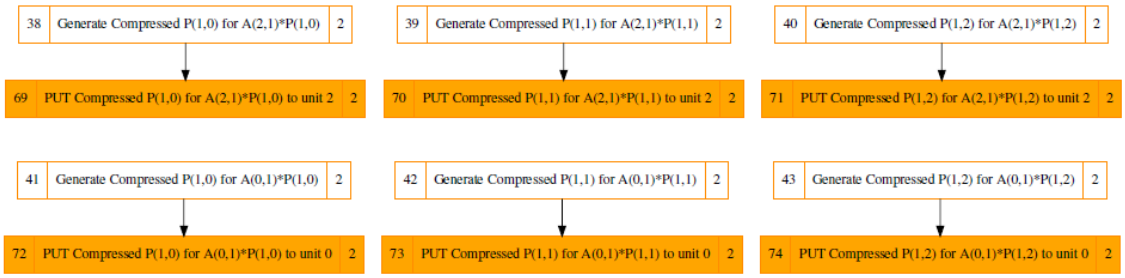


Figura 3.24: Grafo de Execução do exemplo de PtAP para o *rank 1*, parte 2

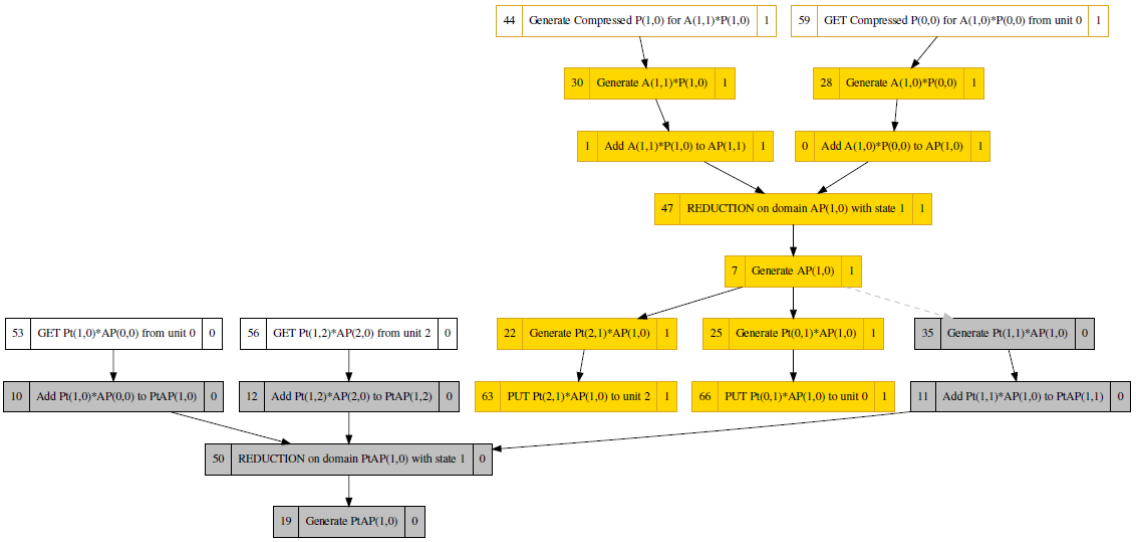


Figura 3.25: Grafo de Execução do exemplo de PtAP para o *rank 1*, parte 3

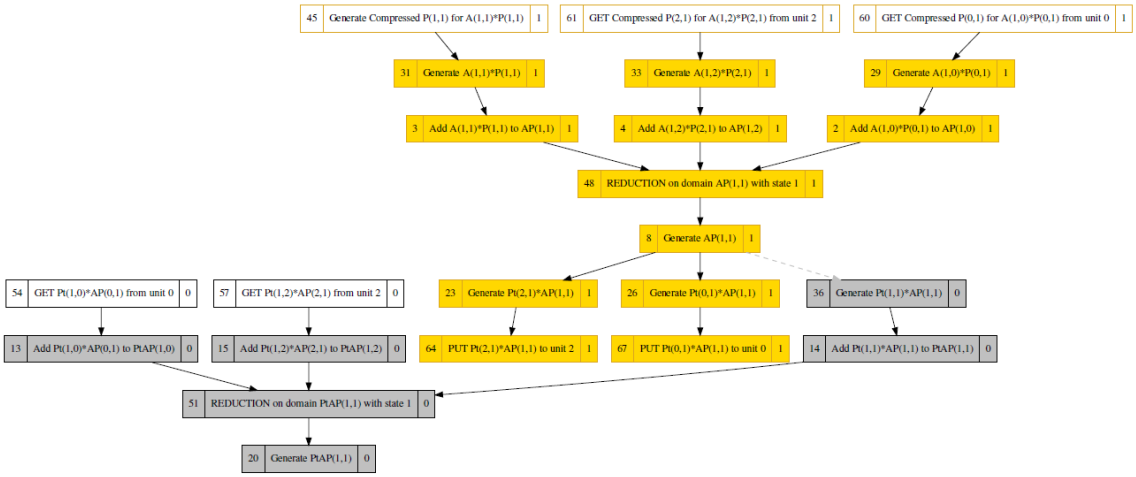
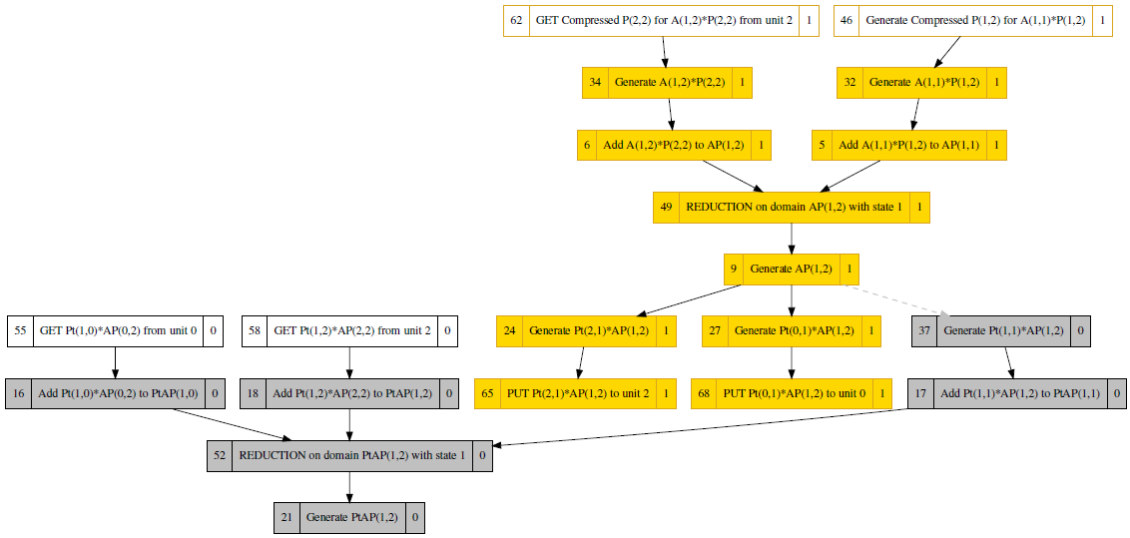


Figura 3.26: Grafo de Execução do exemplo de PtAP para o *rank 1*, parte 4



Esse núcleo é bastante intensivo em termos de operações entre domínios. Mesmo para um caso pequeno, com 3 *ranks MPI*, como o exemplo nas Figuras 3.21 e 3.22, o Grafo de Execução local de cada *rank MPI* possui dezenas de tarefas. Para exemplificar o Grafo de Execução do *rank 1*, foi necessário dividi-lo em quatro imagens para fins de legibilidade. As Figuras 3.23, 3.24, 3.25 e 3.26 mostram trechos distintos do GE do exemplo para o *rank 1*.

As cores no GE representam subgrafos, quando ativados. São sempre três subgrafos, mesmo quando executado com muitos processos, pois o número depende da quantidade de *generators* aninhados. Note que, no Código 3.14, linha 34, o *generator* **partialAP** invoca o *generator* de **compP**, e ambos podem gerar comunicação. O subgrafo 2 se concentra na geração e envio de Ps comprimidos. O subgrafo 1 se concentra na geração e envio de de produtos parciais $P^T * AP$. O subgrafo 0 contém todas as demais tarefas.

3.5 *Features* adicionais

Esta seção detalha descreve algumas *features* adicionais implementadas no *framework dataflow* que costumam ser úteis para se implementar algoritmos usando a ferramenta.

3.5.1 Visualização de grafos

Praticamente todos os grafos apresentados até aqui, sejam Grafos de Dados (GD) ou Grafos de Execução (GE), foram gerados pelo próprio *framework* com auxílio da biblioteca *boost*(KORANNE [41]) e do famoso software *graphviz*(ELLSON *et al.* [42]). Variáveis de ambiente definidas ativam a geração do arquivo *dot* (formato do *graphviz*) de cada grafo e sua conversão em *pdf*. Isso é util para verificar se os grafos de dados e de execução estão sendo gerados conforme o esperado.

As variáveis de ambiente são SBR_DF_PRINT_DG para grafos de dados e SBR_DF_PRINT_EG para grafos de execução. O valor da variável deve ser o *rank* para o qual será gerado ou -1 para gerar para todos os *ranks*. A princípio, essa funcionalidade só está ativa quando o *framework* é compilado em modo *debug*, para evitar que a *feature* seja ativada em produção.

3.5.2 Gráfico de Gantt

Outra funcionalidade útil é a geração de Gráfico de Gantt (GANTT e ADAMIECKI [43]) para as tarefas em cada *rank MPI*. O *timer* hierárquico provido pela aplicação de *benchmark* do SolverBR traz uma visão limitada do processo de execução. Em

um algoritmo complexo executado segundo o grafo *dataflow*, é importante ter uma visão, não só da duração de cada tarefa, mas também de quando cada uma se iniciou.

Figura 3.27: Exemplo de Grafico de Gantt para o *rank 0* em uma execução do PtAP para 2 *ranks MPI*

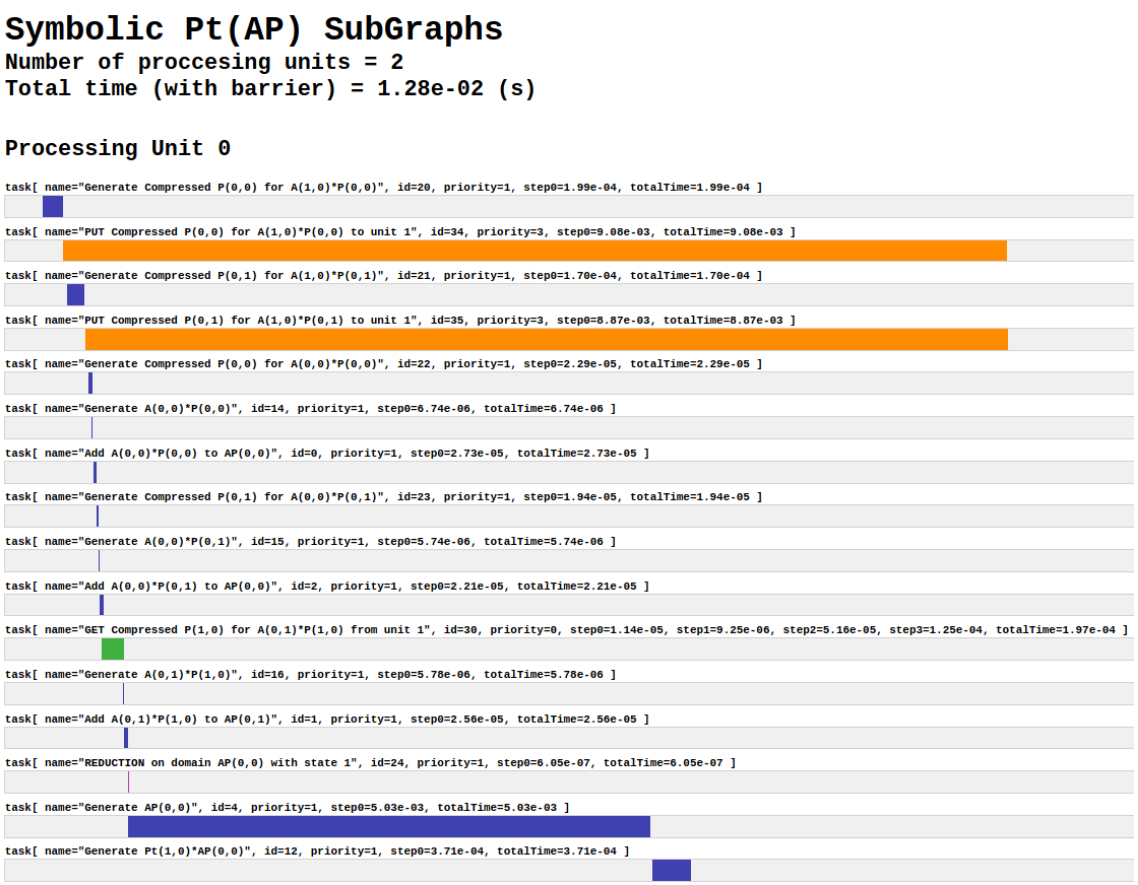


Figura 3.28: Exemplo de Grafico de Gantt global (todos os ranks) em uma execução do PtAP para 2 *ranks MPI*

O Gráfico de Gantt gerado pelo framework mostra exatamente isso para cada rank MPI, ou seja, quando cada tarefa se iniciou e quando terminou. O formato de saída é um simples arquivo *html* para cada *rank*. Além do arquivo por rank, é gerado um Grafico de Gantt global, contendo os tempos globais (de início e fim) de todos os *ranks*. A Figura 3.27 traz um exemplo de um trecho do Gráfico de Gantt para as tarefas do *rank 0* em uma execução do PtAP com2 *ranks MPI*. Tarefas com naturezas distintas recebem cores distintas. As do tipo *PUT* aparecem em laranja, as do tipo *GET* em verde e as demais em azul.

A Figura 3.28 traz um exemplo do Grafico de Gantt global, com os tempos de cada *rank MPI* (o início da barra vermelha é o início da primeira tarefa e o final é o final da última tarefa do *rank* correspondente), permitindo avaliar quais processos estão impactando mais o desempenho.

Há uma barreira MPI no início e no final da execução do *dataflow* para tornar possível a comparação de tempos entre os *ranks*. Esse recurso está desabilitado por *default*, para evitar que seja usado acidentalmente em produção. Para ser ativado, precisa ser compilado com uma definição específica (não adianta deixar ativo em *debug*, pois a medição de tempo deve ser feita sempre em modo *release*).

Capítulo 4

Experimentos Numéricos

Esse capítulo é dedicado à aplicação do *framework dataflow* em núcleos de álgebra linear esparsa relevantes à Simulação de Reservatórios e de Geomecânica, trazendo um comparativo de performance e escalabilidade, em arquiteturas distribuídas, com os mesmos núcleos da biblioteca PETSc [44]. O objetivo é ter uma avaliação extensa do *framework* frente a uma ferramenta que é referência nas operações selecionadas. Para isso, os testes realizados cobriram diversos aspectos com potencial efeito no desempenho:

- **Complexidade dos núcleos**: foram testados núcleos de baixa a alta complexidade (em termos de número de operações entre domínios), incluindo o núcleo mais complexo em uso no SolverBR: o $P^T AP$, que impõe uma elevada carga aos grafos *dataflow*.
- **Tamanhos de problema**: foram usadas matrizes e vetores de ordens pequenas, médias e grandes (no contexto das aplicações científicas da Petrobras).
- **Nível de paralelismo**: cada teste foi executado com até 2000 processos, distribuídos em 50 nós computacionais (mesmo os casos pequenos), permitindo uma ampla avaliação de escalabilidade.
- **Particionamento de dados**: em cada rodada paralela, foram exploradas diferentes estratégias de particionamento dos dados, com reflexos na quantidade, nos tamanhos e nas interconectividades entre os domínios das matrizes (com grande reflexo nos grafos *dataflow*).

4.1 Introdução

4.1.1 Núcleos selecionados

A seleção das operações foi feita com base em alguns critérios: importância do algoritmo nas aplicações de interesse e importância teórica. Além disso, a operação

precisa ser aderente ao *framework*. Como visto no Capítulo 3, a versão implementada do *dataflow* usa um grafo estático que precisa ser plenamente montado em uma fase de *setup* para posterior execução. Os núcleos mais comuns de álgebra linear (operações envolvendo matrizes e vetores) podem ser implementados na ferramenta sem maiores dificuldades. Porém, alguns algoritmos sofisticados, como métodos iterativos (cujas iterações são definidas durante a execução), não são aderentes ao *framework* atual, e serão tratadas em trabalhos futuros.

Nesse contexto, três núcleos foram selecionados:

- $\mathbf{y} = \mathbf{Ax}$ (MatVec): multiplicação matriz-vetor, onde $\mathbf{A}$ é uma matriz esparsa e $\mathbf{x}$ e $\mathbf{y}$ são vetores coluna densos. Mais detalhes na Seção 3.4.1.
- $\mathbf{y} = \mathbf{A^T x}$ (Transp. MatVec): multiplicação matriz transposta vetor, onde $\mathbf{A^T}$ é uma matriz esparsa transposta logicamente e $\mathbf{x}$ e $\mathbf{y}$ são vetores coluna densos. Mais detalhes na Seção 3.4.2.
- $\mathbf{C} = \mathbf{P^T AP}$ (PtAP): multiplicação tripla de matrizes esparsas, onde $\mathbf{A_{N \times N}}$ é uma matriz esparsa quadrada, $\mathbf{P_{N \times M}}$ é uma matriz esparsa retangular que representa uma operação de prolongamento e $\mathbf{P^T_{M \times N}}$ é matriz esparsa retangular que representa uma operação de restrição, com $\mathbf{M < N}$. Mais detalhes na Seção 3.4.3.

Os três algoritmos têm importância prática e teórica. O MatVec é uma das operações que consomem mais tempo nos métodos iterativos de Krylov (VAN DER VORST [45]). As operações Transp. MatVec e PtAP são usadas no precondicionador Multiescala no GeomecBR (FIGUEIREDO *et al.* [40]), e também podem ser usadas em métodos Multigrid (CHAN e TUMINARO [46]). Do ponto de vista teórico, o MatVec e o Transp. MatVec oferecem uma avaliação do *framework* em algoritmos paralelos de baixa e média complexidades. Já o PtAP, permite avaliar algoritmos de alta complexidade, pois o volume de operações e a própria estrutura do algoritmo tornam sua otimização em ambiente distribuído bastante desafiadora. Detalhes de cada implementação são discutidos na Seção 3.4.

4.1.2 PETSc

Sobre o PETSc

O *Portable, Extensible Toolkit for Scientific Computation - PETSc* (BALAY *et al.* [44]) é uma biblioteca para aplicações científicas desenvolvida e mantida pelo *Argonne National Laboratory*, sediado no estado de Illinois nos Estados Unidos. O PETSc reúne estruturas de dados e rotinas para resolução paralela de sistemas lineares e não-lineares de grande escala, com foco em sistemas originários de equações diferenciais parciais. Ele suporta paralelismo por MPI, por GPU e híbrido MPI+GPU. Nesse trabalho, vamos nos limitar ao paralelismo por MPI.

Para a consolidação dos resultados apresentados, foi utilizada a versão 3.14, a mais recente disponível do PETSc na data de início dos testes (janeiro/2021). Essa versão trouxe ganhos de performance significativos em relação às versões anteriores, especialmente na operação PtAP.

Por que o PETSc?

É importante destacar que o PETSc **não** é uma ferramenta *dataflow*, mas é uma ferramenta paralela com foco em escalabilidade bastante reconhecida e utilizada em HPC, tanto no meio acadêmico quanto na indústria. A intenção aqui é comparar um *framework dataflow* com uma aplicação paralela padrão bem implementada, e que seja, de fato, referência para os núcleos de álgebra linear computacionais selecionados. Embora existam diversas ferramentas *dataflow* disponíveis no mercado, elas variam bastante em arquitetura, modelo de paralelismo, granularidade de operações, grau de maturidade, etc. E nenhuma delas é tão amplamente empregada em sistemas lineares **esparsos** de grande escala em arquiteturas distribuídas quanto o PETSc.

4.1.3 SolverBR

O SolverBR vem sendo desenvolvido desde 2017, mas a versão inicial do *framework dataflow* foi implementada ao longo do ano de 2020. O SolverBR já dispunha de ferramentas internas para avaliação de desempenho, como um *timer* hierárquico para medição de tempo e uma ferramenta de *benchmark* para avaliação de desempenho e escalabilidade. Os kernels a serem executados são configurados por arquivos JSON, e geram arquivos CSV como saída. É possível executar repetidas vezes um mesmo kernel, gerando estatísticas das rodadas (média dos tempos, mediana, mínimo, máximo, etc). Isso é útil na medição de kernels leves, que são mais afetados por flutuações no desempenho da máquina.

4.1.4 Ambiente de Execução

Guaricema

O Guaricema é um supercomputador da Petrobras exclusivo para a Área de Reservatórios. Ele é composto por um *cluster* de 406 nós computacionais, cada um com 2 processadores Intel Xeon Gold 6148 de 20 *cores* cada (40 por nó) e 384 GB de memória RAM. Tudo interligado por uma rede InfiniBand EDR. Instalado no CIPD da Petrobras, no CENPES, o Guaricema aumentou em 1200% a capacidade de processamento de dados de reservatório da empresa, quando começou a operar

em capacidade máxima (no início de 2020). Esse é o principal *cluster* onde os simuladores aos quais o SolverBR está acoplado executam atualmente (em ambiente de produção). Por isso, os testes nessa máquina têm bastante interesse prático.

SDumont

O supercomputador SDumont opera no Laboratório Nacional de Computação Científica (LNCC) em Petrópolis-RJ, e está entre os supercomputadores mais rápidos da América Latina, segundo o *ranking Top500* de Novembro/2020 (DONGARRA *et al.* [13]). No Brasil, ele é superado apenas por dois supercomputadores da Petrobras: o Atlas e o Fênix. O SDumont possui um total de 36.472 núcleos de CPU, distribuídos em 1.134 nós computacionais com arquitetura multi-core, além de outros nós computacionais com arquiteturas híbridas (incluindo GPUs), todos interligados por uma rede de interconexão *Infiniband* proporcionando alto rendimento e baixa latência. A capacidade instalada de processamento é da ordem de 5,1 Petaflop/s.

A intenção inicial era executar os testes no SDumont usando até 50 nós computacionais (2400 processos, pois cada nó possui 48 *cores*). Porém, na ocasião de realização dos testes, tivemos problemas de instabilidade nas máquinas do SDumont, ocasionando erros ao em JOBs com muitos nós (mais de 10). O erro foi rapidamente sanado pela equipe técnica do *cluster*, mas as rodadas seguiram com enorme flutuação nos tempos de execução dos algoritmos, afetando de maneira inconsistente não só as implementações *dataflow*, mas também o próprio PETSc. Além disso, comparações em tempos absolutos entre rodadas de um mesmo núcleo (compilados com os mesmos compiladores e com as mesmas FLAGs) no SDumont e no Guaricema parecem não fazer sentido à primeira vista. Por exemplo, rodadas do MatVec com matriz de ordem 1 milhão e particionamento em 3 eixos usando o PETSc com 2000 processos foram cerca de 3x mais rápidas (em tempos absolutos) no Guaricema do que em rodadas com número de processos similar no SDumont. Essas questões diminuíram a confiança nos tempos de execução observados no *cluster*, e a falta de tempo hábil para o entendimento do problema nos fez tomar a decisão de usar os resultados do Guaricema, cujos tempos se mostraram bem mais constantes e coerentes. No entanto, os tempos das rodas feitas no SDumont estão disponíveis no Apêndice B (exceto o caso PtAP com matriz de ordem 100 milhões).

Compilação

Tanto o SolverBR quanto o PETSc foram compilados em modo *Release* com compilador Intel 2018 (o mais usado na Petrobras) com as mesmas *flags* de compilação. O Código 4.1 mostra as *flags* usadas para compilar o PETSc (versão 3.14), e também o SolverBR.

Código 4.1: FLAGs de compilação usadas no PETSc (e no SolverBR)

```
1  export CPP_COMP=mpiicpc
2  export C_COMP=mpiicc
3  export F_COMP=ifort
4  export CPP_FLAGS="-O3 -m64 -xHost -no-prec-div -fp-model fast=2 -no-inline-max-size -qopenmp
       -simd"
5  export FRT_FLAGS="-O3 -m64 -xHost -no-prec-div -fp-model fast=2 -no-inline-max-size -qopenmp
       -simd"
6  export LNK_FLAGS="-L$I_MPI_ROOT/intel64/lib -lmpifort -lmpi"
7  ./configure --with-cc=$C_COMP --with-cxx=$CPP_COMP --with-fc=$F_COMP --with-shared-libraries
       --with-environment-variables --with-debugging=0 COPTFLAGS="$CPP_FLAGS" CXXOPTFLAGS="
       $CPP_FLAGS" FOPTFLAGS="$FRT_FLAGS" --with-mpi-include=$I_MPI_ROOT/intel64/include --
       with-mpi-lib="-L$I_MPI_ROOT/intel64/lib -lmpifort -lmpi" --with-mpiexec=$I_MPI_ROOT/
       intel64/bin/mpirun --with-blas-lapack-dir=$MKLROOT
```

4.1.5 Organização dos testes

Matrizes

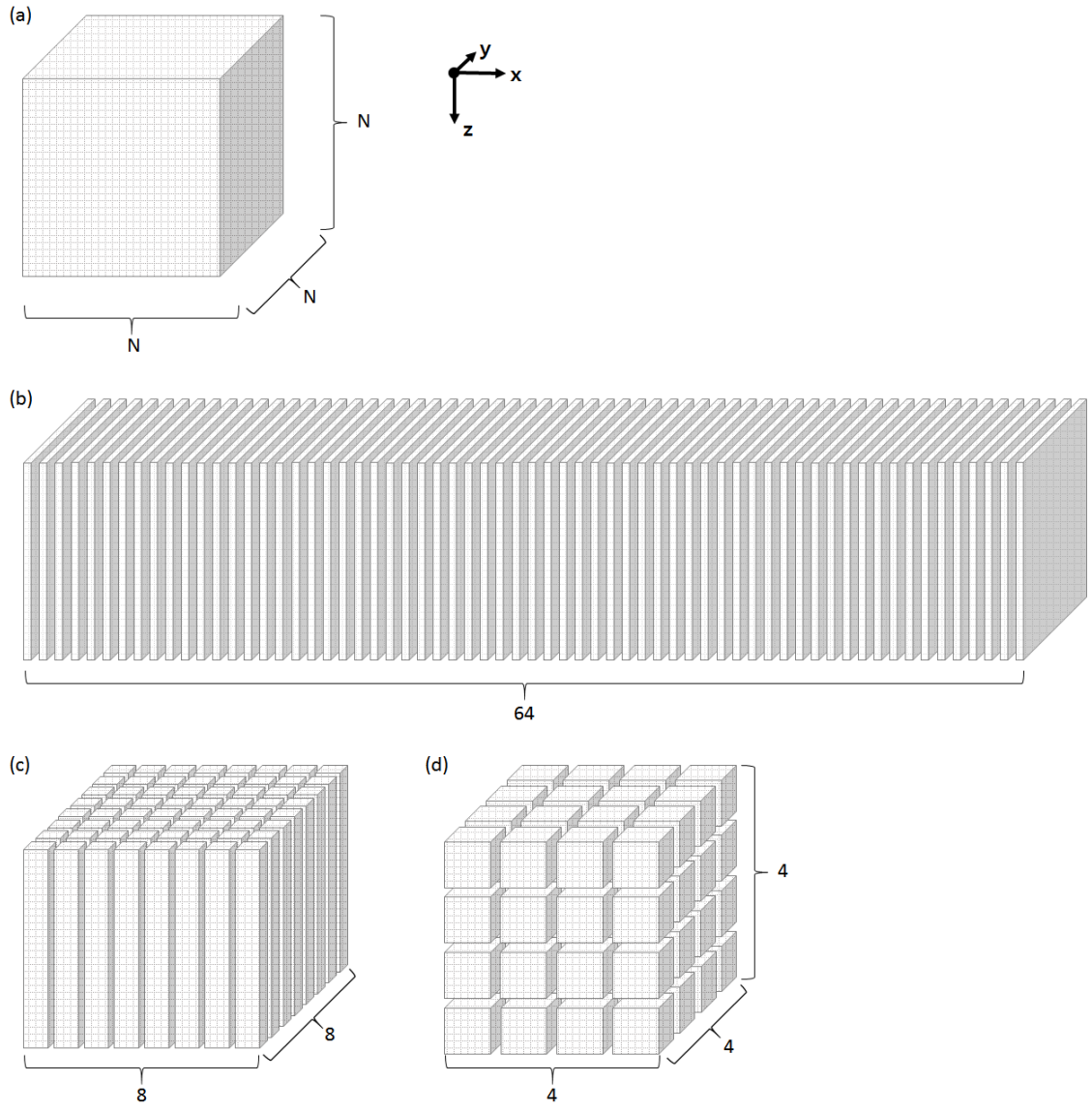
Foram usadas matrizes com estruturas idênticas às usadas no GeomecBR (descritas na Seção 2.2.2). Para se ter uma visão ampla dos efeitos da carga dedados nos núcleos de álgebra linear testados, foram geradas matrizes a partir de *grids* com diferentes ordens de grandeza: com 1 milhão, 10 milhões e 100 milhões de **elementos**. Todas as matrizes testadas são construídas a partir dos **nós dos elementos** do *grid* considerando-se um ***stencil* de 27 pontos**, em alinhamento com as matrizes usadas no GeomecBR (ver Figura 2.8). Em todos os núcleos testados, o número de graus de liberdade considerado é 3, o que implica que cada entrada da matriz é um bloco com 3x3 valores (também em alinhamento com o GeomecBR). No caso do núcleo *PtAP*, o PETSc não implementa essa operação para matrizes *blocadas*, então os testes foram feitos com a matriz *blocada* convertida em escalar mantendo-se o número total de valores, mas aumentando-se a esparsidade para endereçar cada valor escalar dentro de cada bloco.

Paralelismo

O *framework dataflow* foi concebido para domínios de grão grosso (*coarse grain*). Ele trabalha com um grafo estático distribuído que precisa ser construído em uma fase de *setup* e requer comunicação entre os processos durante sua construção. Portanto, para avaliar como a granularidade dos domínios e o tamanho do grafo podem influenciar o desempenho do *framework*, todas as operações foram testadas usando-se até 2000 processos MPI (50 nós computacionais do *cluster* Guaricema) para todos os tamanhos de matrizes, incluindo as pequenas e médias (de ordens 1 milhão e 10 milhões, respectivamente). Por exemplo, uma matriz Laplaciana de 27 pontos de ordem 1 milhão e blocos 3x3 ocupa, em formato BCSR (Blocked Compressed Sparse Row) SAAD [38], menos de 1% dos 384 GB de memória RAM de um único nó *cluster* Guaricema. Usando 50 nós, a mesma matriz ocupa menos de 0,01% da memória total, e a granularidade dos domínios fica bastante restrita.

Por essa razão, as rodadas paralelas envolvendo matrizes pequenas e médias foram divididas em **casos de interesse prático** e **casos de extrapolação**, diferenciando o nível de paralelismo usual do nível de paralelismo intensivo usado para testar os limites do *framework*. Os casos de interesse para as matrizes de ordem 1 milhão são as rodadas com múltiplos processos em um único nó, e para as matrizes de ordem 10 milhões são as rodadas com até 8 nós. Em ambos os casos, as demais rodadas, com até 50 nós, são casos de extrapolação. Para as matrizes de ordem 100 milhões, todos os casos foram considerados de interesse prático, mas as rodadas se iniciaram em 10 nós.

Figura 4.1: Ilustração das três estratégias de particionamento do grid. a) *Grid* com N^3 elementos não particionado. b) Grid particionado em 64 domínios somente no eixo x, cada um com $N/64 * N * N = N^3/64$ elementos. c) Grid particionado em 64 domínios nos eixos x e y, cada um com $N/8 * N/8 * N = N^3/64$ elementos. d) Grid particionado em 64 domínios nos eixos x, y e z, cada um com $N/4 * N/4 * N/4 = N^3/64$ elementos.

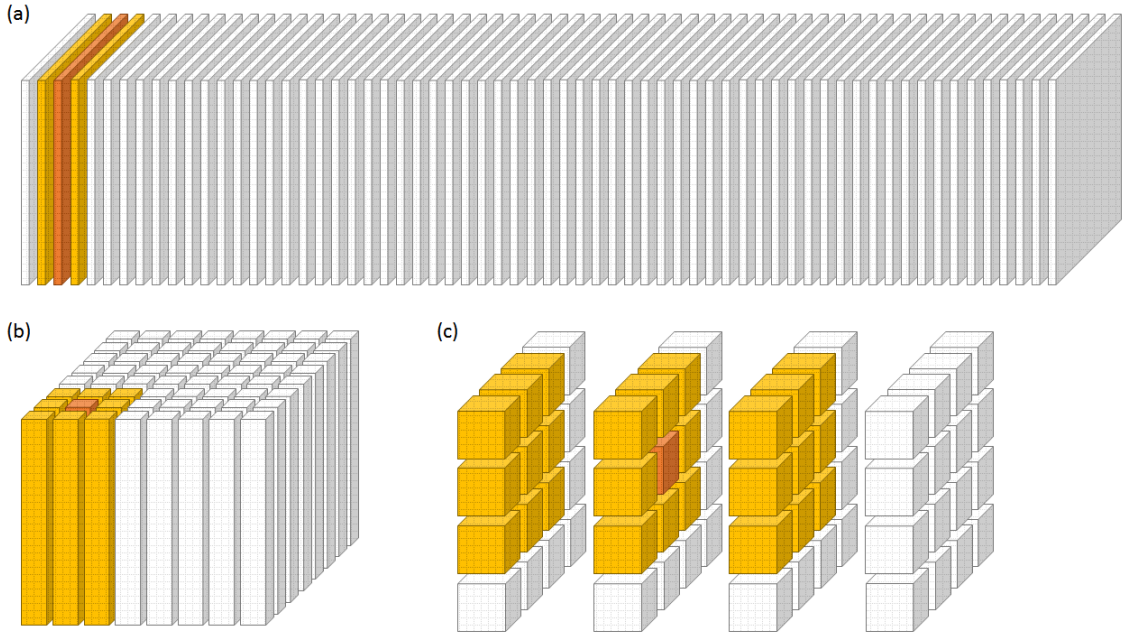


Particionamento

Além do número de processos, outro fator que pode impactar bastante os algoritmos paralelos é o esquema de particionamento. A forma de particionar afeta a conectividade entre os domínios do *grid* e, por consequência, o número de domínios da matriz resultante. Por isso, cada *grid* em cada teste foi particionado de três formas distintas: usando apenas 1 eixo coordenado (eixo x), usando 2 eixos coordenados (x e y) e usando os 3 eixos coordenados (x, y e z). Para uma divisão em N domínios, a partição em 2 eixos é a mais próxima de quadrada possível e a partição em 3 eixos é a mais próxima de cúbica possível. Por exemplo, para $N = 1200$, o particionamento em 1 eixo é $1200 \times 1 \times 1$, em 2 eixos é $40 \times 30 \times 1$ e em 3 eixos é $12 \times 10 \times 10$. Cada dimensão do *grid* será particionada conforme o número de divisores em cada eixo. Quando a divisão ao longo de um eixo não é exata, o resto da divisão é distribuído nas últimas partições ao longo do mesmo eixo. Assim, alguns domínios podem ficar com mais nós do que outros (afetando o balanceamento de carga).

A Figura 4.1 traz uma representação gráfica das três estratégias de particionamento. O item (a) da figura ilustra um *grid* cúbico com N elementos nos eixos x, y e z. Os itens (b), (c) e (d) ilustram o mesmo *grid* sendo particionado em 64 domínios usando-se 1 eixo, 2 eixos e três 3 coordenados, respectivamente.

Figura 4.2: Ilustração do padrão de conexão entre domínios nas três estratégias de particionamento. a) No particionamento em x, a maioria dos domínios se conectam a outros 2 domínios. b) No particionamento em x e y, a maioria dos domínios se conectam a outros 8 domínios. c) No particionamento em x, y e z, a maioria dos domínios se conectam a outros 26 domínios.



No exemplo, embora o número de elementos por domínio seja o mesmo, o formato do *subgrid* de cada domínio varia enormemente (com reflexos na esparsidade da submatriz diagonal correspondente), e também o padrão de conexão com outros domínios (com reflexos na quantidade e na esparsidade das submatrizes de conexão). Maiores detalhes são discutidos ao final da Seção 2.2.2.

A Figura 4.2 ilustra os padrões de conexão entre domínios do *grid* nos diferentes esquemas de particionamento, e a Figura 4.3 mostra como isso se reflete na esparsidade de domínios da matriz distribuída do SolverBR. Importante frisar que cada minúsculo quadrado representado nessa figura é, por si, só uma matriz esparsa tipicamente grande. As submatrizes diagonais (em laranja) possuem ordem próxima de $N^3/64$, e a ordem das submatrizes de conexão é proporcional ao número de nós nas interfaces entre os domínios do *grid*. Note que o particionamento do *grid* em 1 eixo, 2 eixos e 3 eixos gera matrizes com esparsidade de domínios com 3 diagonais, 9 diagonais e 27 diagonais, respectivamente. E quanto maior a quantidade de domínios na matriz, maior a complexidade dos algoritmos paralelos, especialmente o PtAP.

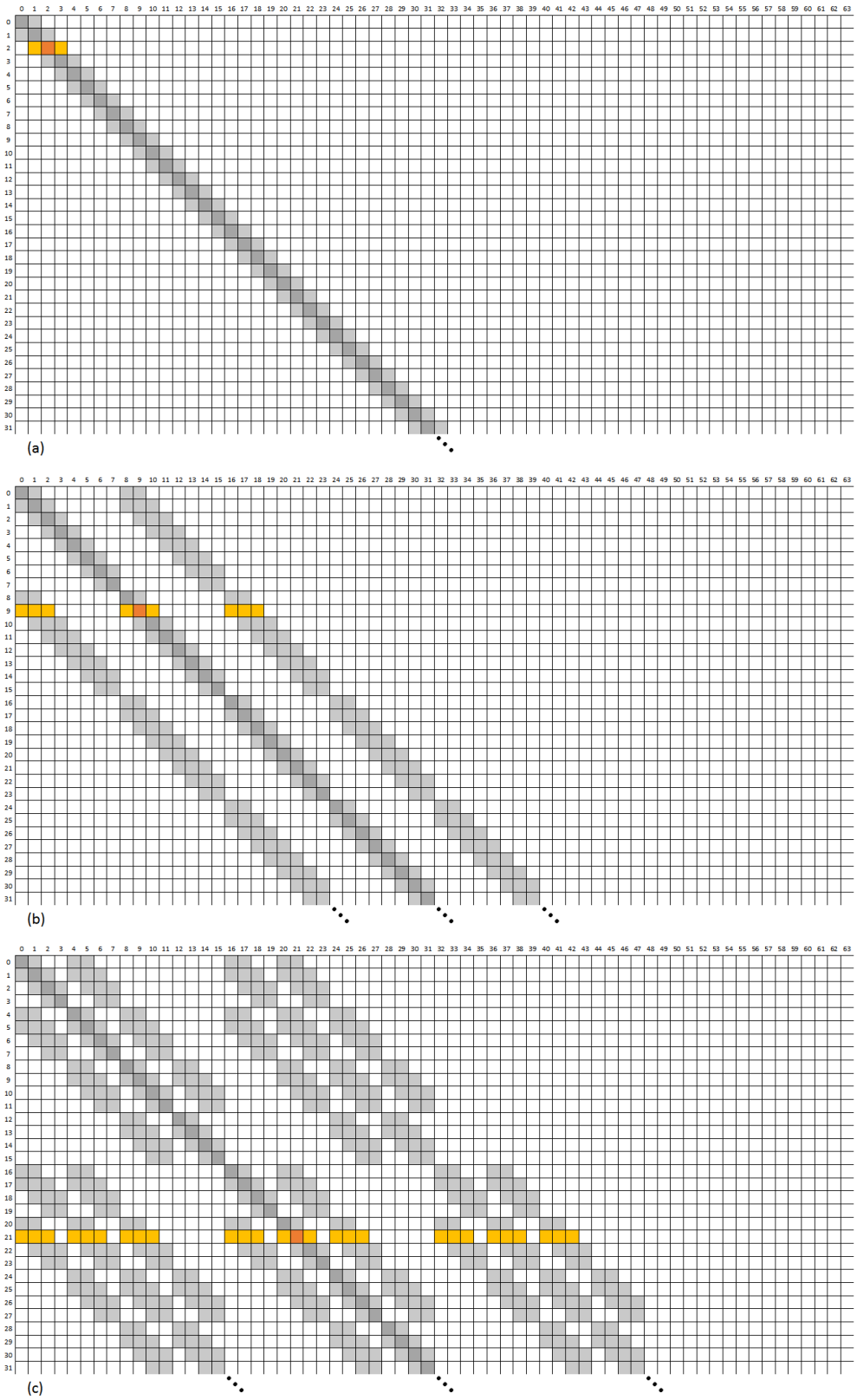
No caso PETSc, a matriz distribuída é estruturada de maneira diferente. Cada domínio do *grid* se reflete em um domínio diagonal da matriz, como no SolverBR, mas todas as equações de todas as interfaces entre os domínios ficam agrupadas em uma única matriz de conexão. Desse modo, em cada domínio-linha da matriz do PETSc, há sempre só duas submatrizes, uma diagonal (quadrada) e outra não-diagonal.

Do ponto de vista conceitual, é importante testar as três estratégias de particionamento para uma avaliação completa do *framework*. Do ponto de vista prático, nos simuladores de reservatório/geomecânica da Petrobras que usam o SolverBR, o particionamento em 2 eixos é o mais importante. Isso porque usando apenas 1 eixo, o paralelismo fica limitado ao número de elementos ao longo de uma única coordenada no *grid*; e usando 3 eixos, a convergência dos algoritmos é afetada negativamente pelo particionamento na coordenada z, devido à grande variabilidade da pressão em função da profundidade (com efeitos nos deslocamentos verticais).

4.2 Resultados - Casos de interesse prático

Nesta seção serão apresentadas as comparações de desempenho das implementações *dataflow* dos núcleos de álgebra linear esparsa selecionados com os do PETSc, usando um nível de paralelismo prático para cada tamanho de problema (tendo como referência o uso do simulador GeomecBR na Petrobras). Os casos considerados de extrapolação (com nível de paralelismo além do usual) são discutidos na Seção 4.3.

Figura 4.3: Ilustração dos domínios da matriz distribuída do SolverBR. a) Esparsidade de domínios para particionamento no eixo x apenas. b) Esparsidade de domínios para particionamento nos eixos x e y. c) Esparsidade de domínios para particionamento nos eixos x, y e z.



4.2.1 Operação y = Ax

A multiplicação matriz-vetor foi discutida na Seção 3.4.1. Foram implementadas duas versões de MatVec usando *dataflow*. Na versão *ordered*, as operações entre domínios de matriz e de vetor são efetuadas em uma ordem específica, enquanto na versão *relaxed*, as operações são efetuadas de acordo com o recebimento de dados remotos, o que pode ocorrer em qualquer ordem.

Como a flutuação de tempo dessa operação em rodadas subsequentes é significativa, cada teste foi executado **cinco** vezes, e o tempo considerado nos gráficos e tabelas é a **mediana** das cinco rodadas. Além disso, cada medição de tempo total foi feita com barreiras na ferramenta de *benchmark*, de modo a capturar deslocamentos temporais entre os *ranks MPI*.

Em todas as tabelas e gráficos desta seção, os nomes PETSc, DF-Ord e DF-Rel se referem às operações MatVec do PETSc, *dataflow ordered* e *dataflow relaxed*, respectivamente.

Matriz de ordem 1 milhão (bloco 3x3), 1 nó

O *grid* usado nesses casos possui 2000x50x10 = 1.000.000 de elementos. Tal discrepância de tamanhos entre o eixo x e o eixo y não é muito usual, mas era necessária para se obter até 2000 domínios mesmo particionando apenas no eixo x. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(50+1)x(10+1) = 1.122.561 blocos, lembrando que cada entrada da matriz é composta por blocos com 3x3 valores.

Uma matriz dessa ordem é pequena para as aplicações alvo da Petrobras. Ela ocuparia, inteira (sem particionar), menos de 3 GB de memória, ou seja, menos de 1% dos 384 GB de RAM de um único nó do *cluster*. Portanto, os casos de interesse para matrizes desse tamanho são as rodadas intra-nó (de 1 a 40 processos). Extrapolações para até 2000 processos (50 nós computacionais) são apresentados na Seção 4.3.1.

A Tabela 4.1 compara os tempos da operação MatVec entre o PETSc e as duas versões *dataflow* (a *ordered* e a *relaxed*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.4 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As Figuras 4.5 e 4.6 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Como mostra a Tabela 4.1, para esse núcleo e esse tamanho de matriz, as duas implementações *dataflow* foram competitivas com o PETSc em todos os cenários. Em geral, o DF-Ord e o DF-Rel foram ligeiramente piores com poucos processos (1 a 10) e ligeiramente melhores com muitos processos (30 e 40), indicando uma

Tabela 4.1: Tempos e Speed-up da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
1	1	0.1757	0.1940	0.1940	0.9	0.9
10	1	0.0197	0.0219	0.0220	0.9	0.9
20	1	0.0123	0.0124	0.0125	1.0	1.0
30	1	0.0161	0.0143	0.0143	1.1	1.1
40	1	0.0121	0.0108	0.0108	1.1	1.1

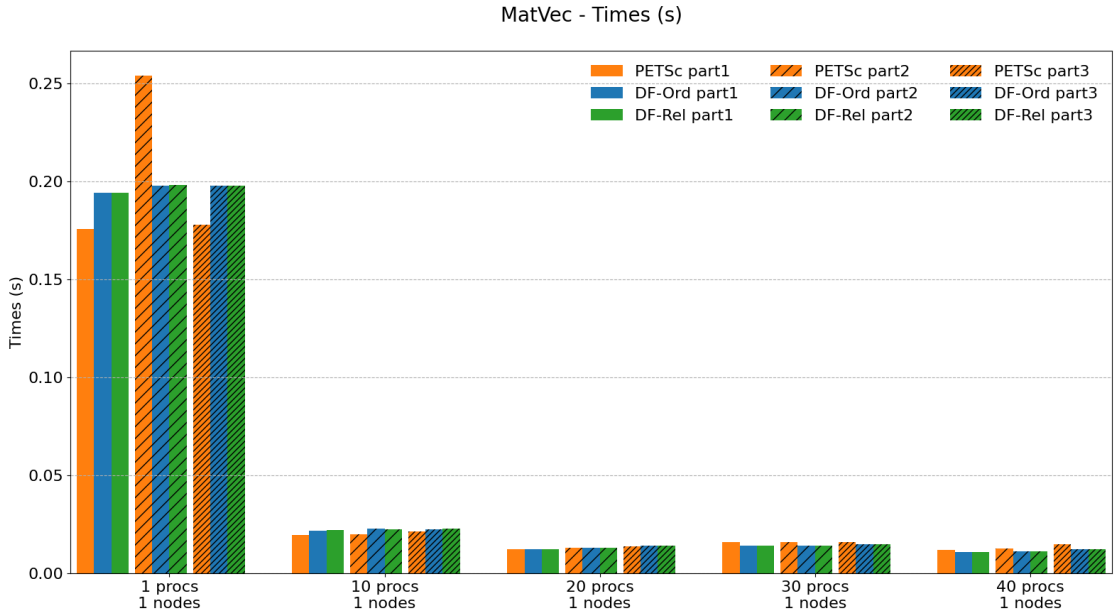
(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
1	1	0.2539	0.1978	0.1980	1.3	1.3
10	1	0.0201	0.0227	0.0226	0.9	0.9
20	1	0.0132	0.0131	0.0131	1.0	1.0
30	1	0.0161	0.0143	0.0143	1.1	1.1
40	1	0.0126	0.0111	0.0111	1.1	1.1

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
1	1	0.1779	0.1976	0.1978	0.9	0.9
10	1	0.0212	0.0227	0.0227	0.9	0.9
20	1	0.0139	0.0143	0.0143	1.0	1.0
30	1	0.0161	0.0150	0.0149	1.1	1.1
40	1	0.0149	0.0123	0.0123	1.2	1.2

Figura 4.4: Gráficos de tempos da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse



escalabilidade melhor dos núcleos *dataflow* nesse caso. A Figura 4.5 mostra um ponto de inflexão com 20 processos nos gráficos de *speed-up*, e a Figura 4.6 mostra a melhor escalabilidade de DF-Ord e DF-Rel, exceto no item (b). O *speed-up* de 1.3x na rodada com 1 processo particionando-se em 2 eixos coordenados parece ter sido problema de flutuação na execução do PETSc, pois a execução em serial não sofre influência do particionamento. Porém, esse deslize na rodada serial do PETSc acabou inflando sua escalabilidade com particionamento em 2 eixos. Entre as implementações *dataflow*, o DF-Ord e DF-Rel se equivaleram em praticamente todas as rodadas em todos os cenários.

Figura 4.5: Gráficos de speed-up da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

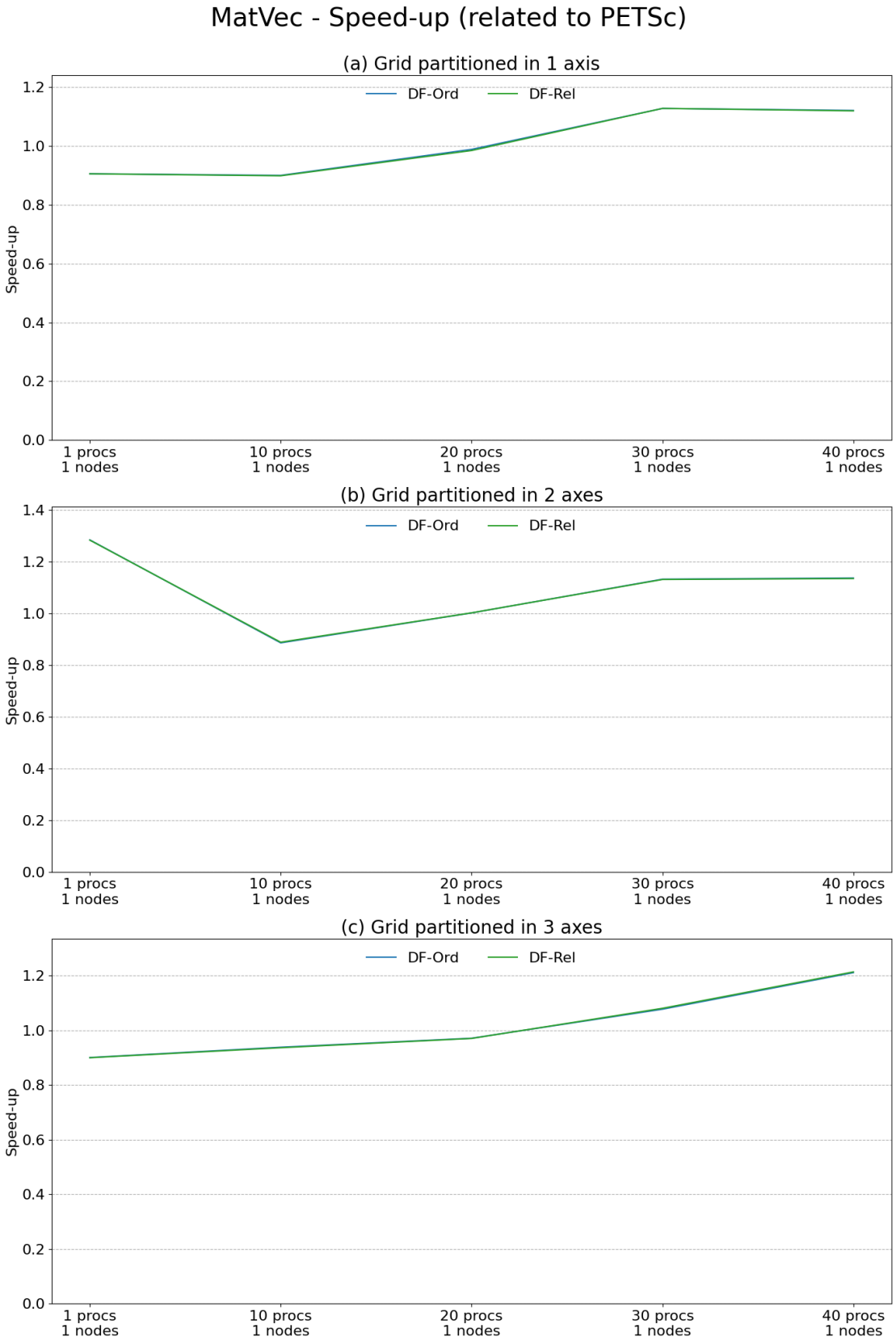
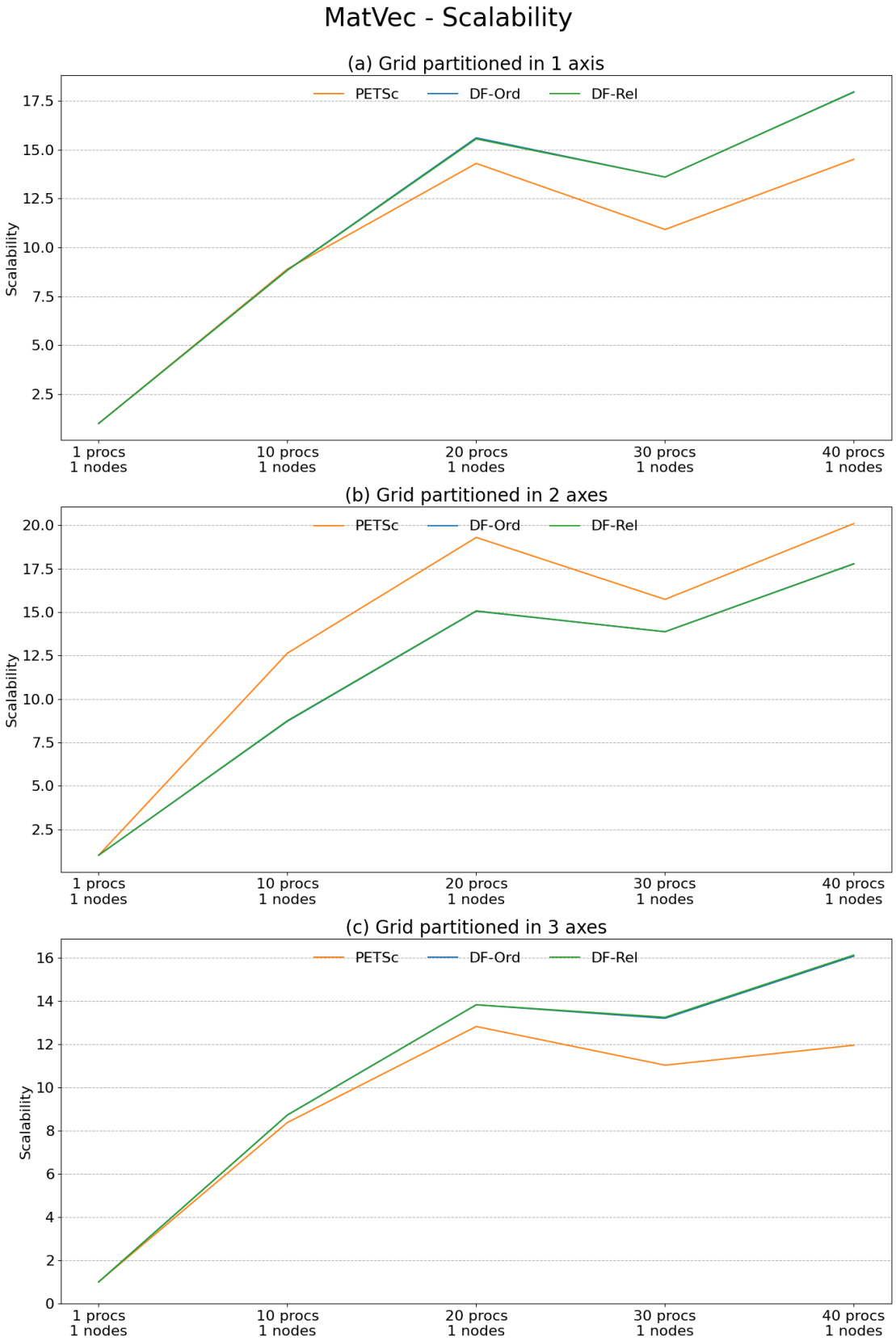


Figura 4.6: Gráficos de escalabilidade da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

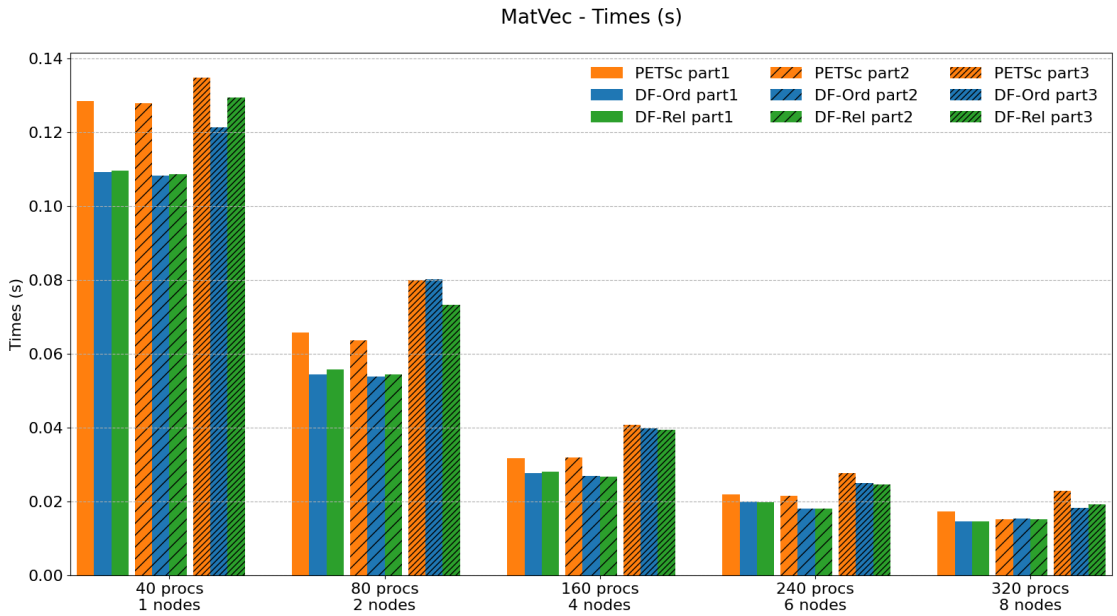


Matriz de ordem 10 milhões (bloco 3x3), até 8 nós

O *grid* usado nesses casos possui 2000x500x10 = 10.000.000 de elementos. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(500+1)x(10+1) = 11.027.511 blocos, lembrando que cada entrada da matriz é composta por blocos 3x3.

Uma matriz dessa ordem ocupa menos de 30 GB de memória, ainda longe dos 384 GB de RAM de um único nó do *cluster* Guaricema. No entanto, um simulador rodando problemas desse tamanho pode ser executado em múltiplos nós (2, 3 ou 4 nós) por questão de desempenho. Por isso, estendemos generosamente as rodadas de interesse para até 8 nós nesse caso. Extrapolações para até 50 nós (2000 processos) são apresentadas na Seção 4.3.1.

Figura 4.7: Gráficos de tempos da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse



A Tabela 4.2 compara os tempos da operação MatVec entre o PETSc e as duas versões *dataflow* (a *ordered* e a *relaxed*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.7 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.8 e 4.9 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Particionando em um único eixo, os dois núcleos *dataflow* tiveram *speed-up* de 1,1x a 1,2x em relação ao PETSc em todas as rodadas. A escalabilidade dos três núcleos (PETSc, DF-Ord e DF-Rel) foram quase equivalentes, exceto por uma ligeira vantagem do PETSc com 4 e 6 nós, mas que não foi observada com 8 nós. Com particionamento do *grid* em 2 eixos coordenados, o *speed-up* das implementações

Tabela 4.2: Tempos e Speed-up da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
40	1	0.1285	0.1092	0.1096	1.2	1.2
80	2	0.0658	0.0545	0.0557	1.2	1.2
160	4	0.0317	0.0277	0.0281	1.1	1.1
240	6	0.0220	0.0201	0.0198	1.1	1.1
320	8	0.0174	0.0147	0.0147	1.2	1.2

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
40	1	0.1279	0.1083	0.1086	1.2	1.2
80	2	0.0636	0.0538	0.0544	1.2	1.2
160	4	0.0320	0.0270	0.0268	1.2	1.2
240	6	0.0216	0.0181	0.0181	1.2	1.2
320	8	0.0153	0.0154	0.0153	1.0	1.0

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
40	1	0.1349	0.1214	0.1295	1.1	1.0
80	2	0.0801	0.0803	0.0732	1.0	1.1
160	4	0.0408	0.0398	0.0394	1.0	1.0
240	6	0.0277	0.0249	0.0246	1.1	1.1
320	8	0.0229	0.0183	0.0193	1.3	1.2

Figura 4.8: Gráficos de speed-up da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

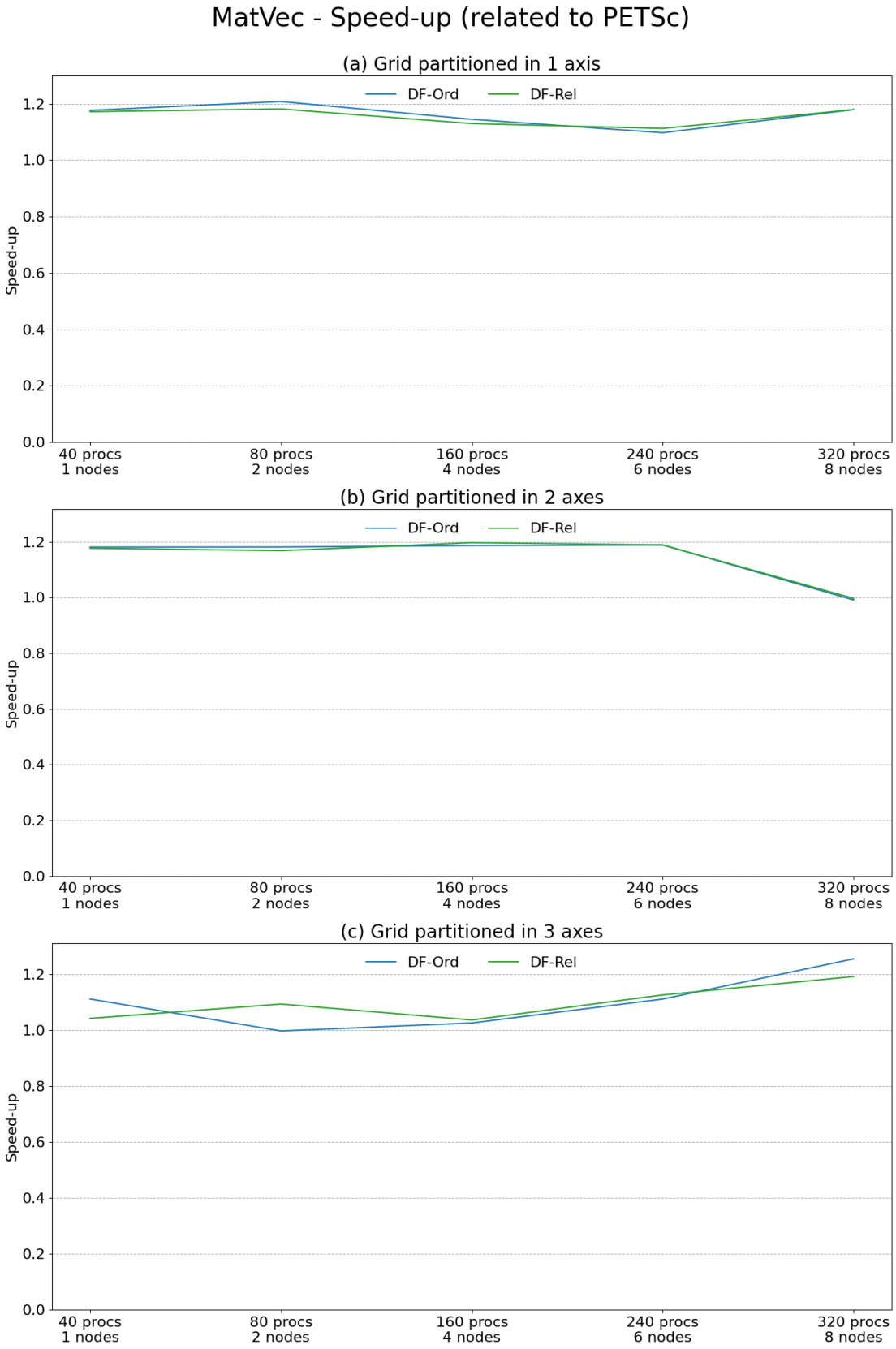
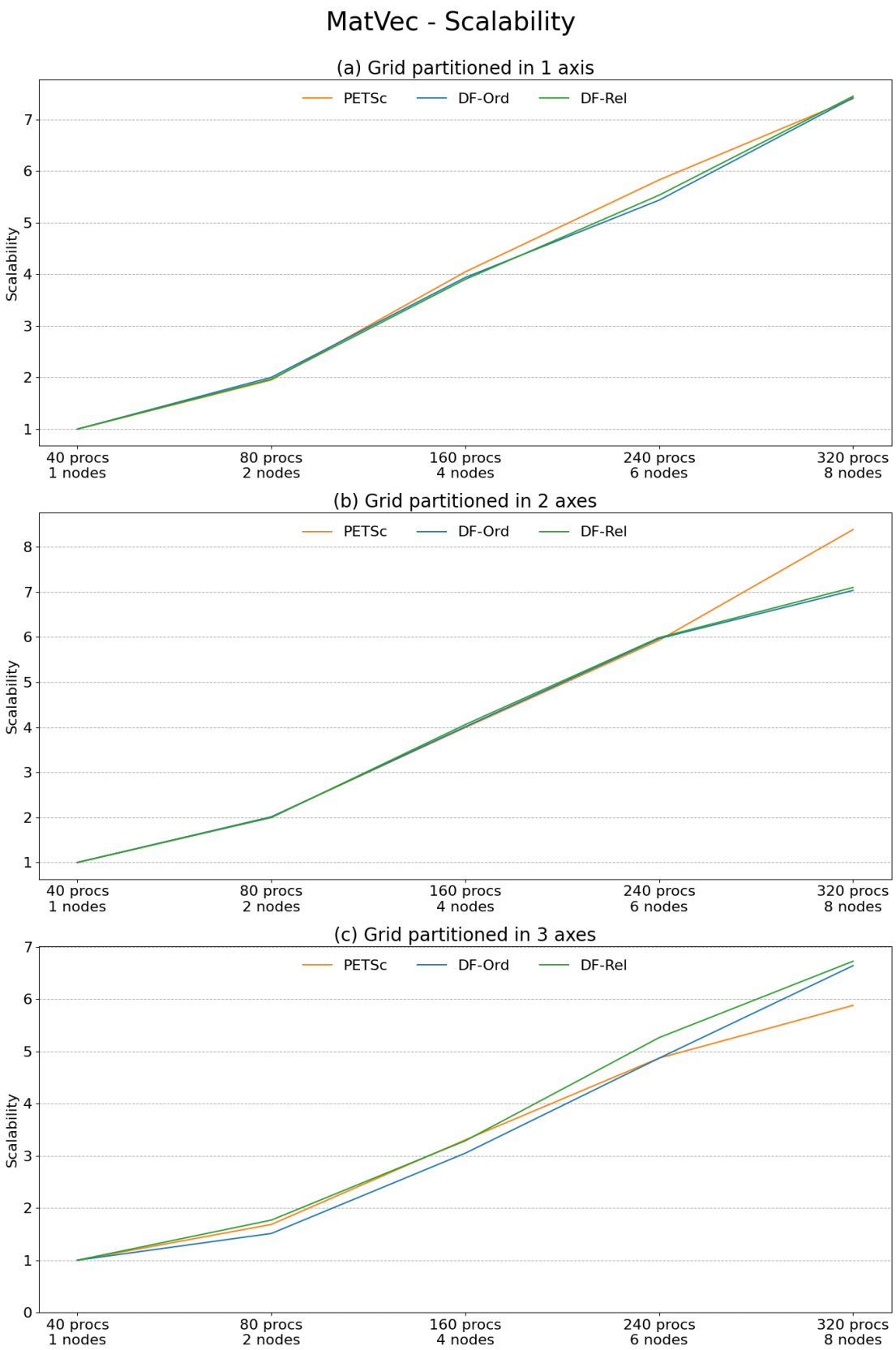


Figura 4.9: Gráficos de escalabilidade da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse



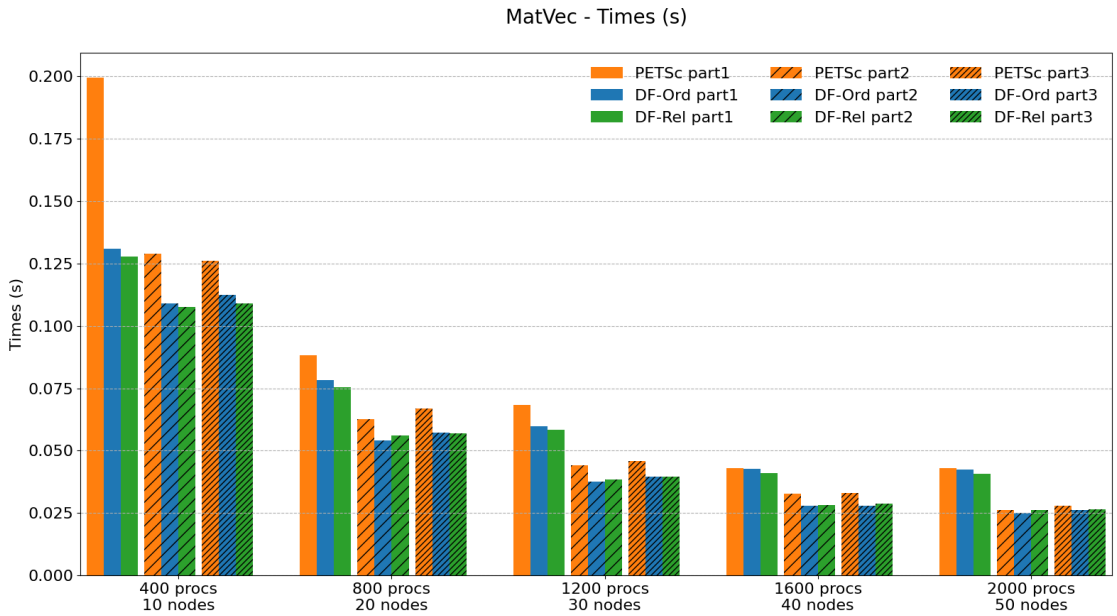
dataflow fica constante em 1.2x usando-se de 1 a 6 nós (40 a 240 processos), mas cai para 1.0x com 8 nós (320 processos). Particionando em 3 eixos, DF-Ord e DF-Rel partem com um *speed-up* de 1.1x e 1.0x em um nó e terminam com 1.3x e 1.2x em 8 nós, respectivamente. Apresentando tanto desempenho quanto escalabilidade superiores ao PETSc em todas as rodadas.

Matriz de ordem 100 milhões (bloco 3x3), até 50 nós

O *grid* usado nesses casos possui 2000x500x100 = 100.000.000 de elementos. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(500+1)x(100+1) = 101.252.601 blocos, lembrando que cada entrada da matriz é composta por blocos 3x3.

Uma matriz dessa ordem ocupa menos de 300 GB de memória, valor já próximo dos 384 GB de RAM de um único nó do *cluster* Guaricema. Além disso, no contexto de um simulador, embora a matriz tenha um peso grande na memória utilizada, outras estruturas de dados (vetores, precondicionadores, dados do sistema não-linear, etc.) podem exigir mais memória. Por isso, problemas desse tamanho precisam ser executados em múltiplos nós. As rodadas desse caso usaram de 10 a 50 nós computacionais (de 400 a 2000 processos).

Figura 4.10: Gráficos de tempos da operação MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



A Tabela 4.3 compara os tempos da operação MatVec entre o PETSc e as duas versões *dataflow* (a *ordered* e a *relaxed*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.10 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de par-

Tabela 4.3: Tempos e Speed-up da operação MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
400	10	0.1996	0.1311	0.1278	1.5	1.6
800	20	0.0881	0.0782	0.0754	1.1	1.2
1200	30	0.0683	0.0599	0.0584	1.1	1.2
1600	40	0.0431	0.0428	0.0409	1.0	1.1
2000	50	0.0431	0.0425	0.0407	1.0	1.1

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
400	10	0.1290	0.1089	0.1075	1.2	1.2
800	20	0.0627	0.0540	0.0562	1.2	1.1
1200	30	0.0442	0.0376	0.0383	1.2	1.2
1600	40	0.0327	0.0279	0.0281	1.2	1.2
2000	50	0.0262	0.0251	0.0261	1.0	1.0

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
400	10	0.1261	0.1123	0.1090	1.1	1.2
800	20	0.0670	0.0572	0.0568	1.2	1.2
1200	30	0.0460	0.0395	0.0396	1.2	1.2
1600	40	0.0332	0.0281	0.0287	1.2	1.2
2000	50	0.0280	0.0262	0.0265	1.1	1.1

Figura 4.11: Gráficos de speed-up da operação MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

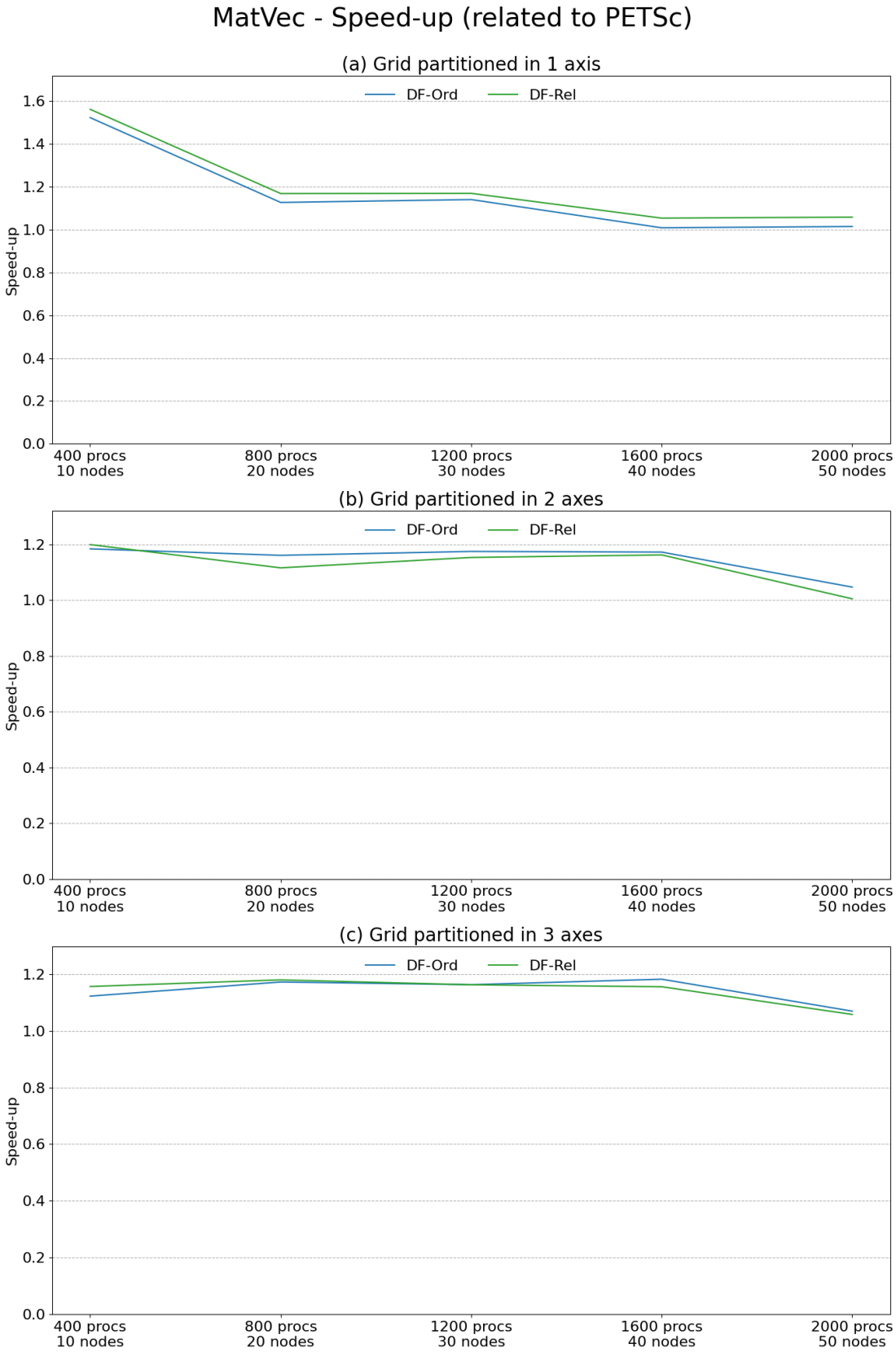
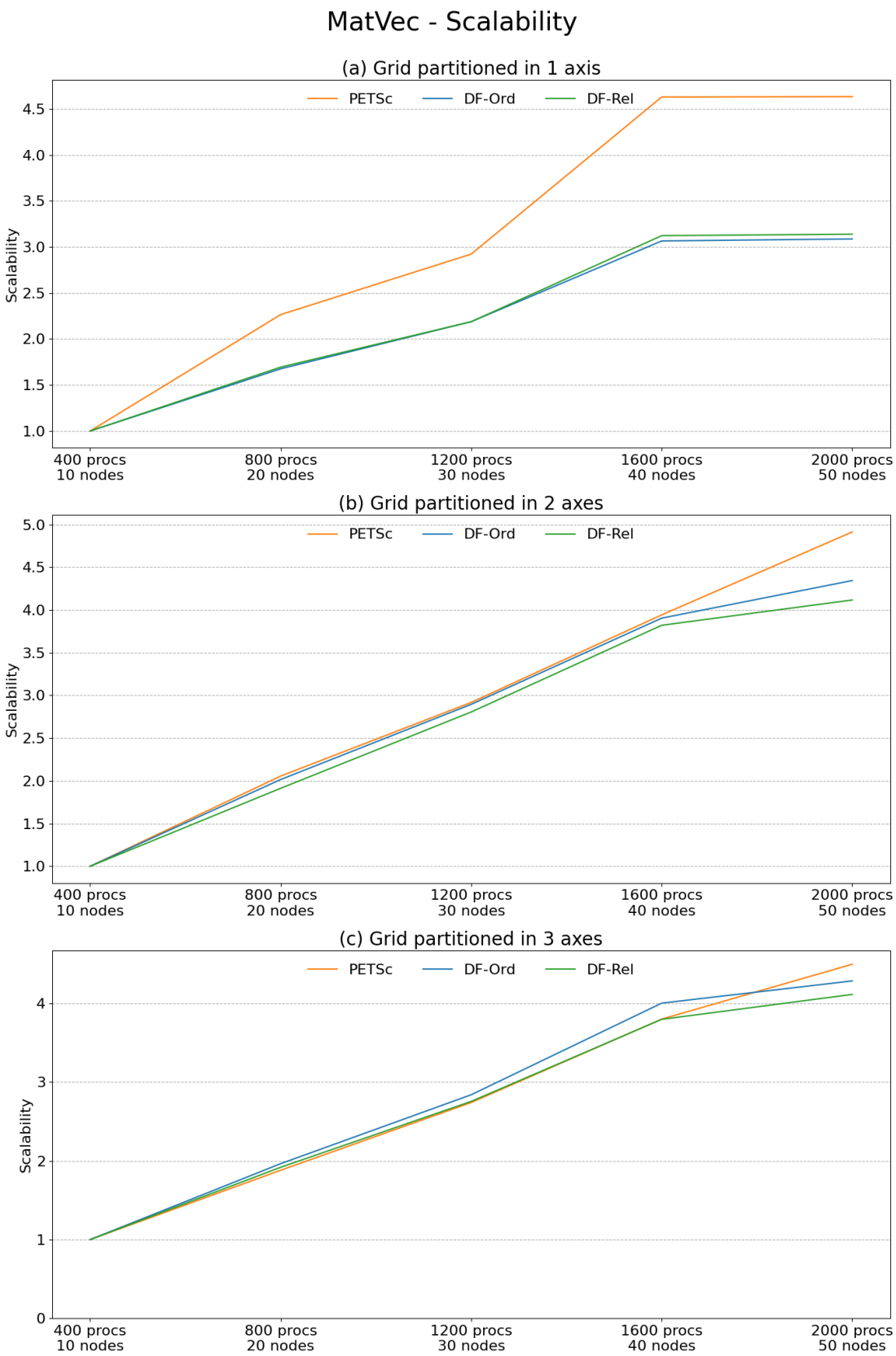


Figura 4.12: Gráficos de escalabilidade da operação MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



ticionamento. As figuras 4.11 e 4.12 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Particionando o *grid* gerador da matriz em 1 eixo coordenado, os núcleos *dataflow* DF-Ord e DF-Rel partem de um *speed-up* em relação ao PETSc de 1.5x e 1.6x com 400 processos (10 nós) e terminam com 1.0x e 1.1x com 2000 processos, respectivamente. Como mostra a figura 4.12, o PETSc teve uma escalabilidade maior, porém inflada pelo desempenho ruim na rodada base com 400 processos MPI. Em tempo absoluto, as implementações *dataflow* foram iguais ou superiores em todas as rodadas. Particionando em 2 eixos, DF-Ord e DF-Rel apresentam *speed-up* de 1.2x em relação ao PETSc em quase todas as rodadas de 10 a 40 nós. Com 50 nós, o PETSc consegue igualar o desempenho dos outros núcleos. Com particionamento em 3 eixos, a situação é parecida, mas o PETSc não consegue igualar o desempenho com 50 nós, terminando com as versões *dataflow* 1.1x mais rápidas. Nesse caso, pôde-se notar uma ligeira diferença de desempenho entre as rodadas *dataflow*, com pequena vantagem do DF-Rel sobre o DF-Ord em 6 diferentes rodadas (todas as particionadas em 1 eixo e uma rodada com particionamento em 3 eixos).

4.2.2 Operação $\mathbf{y = A^T x}$

A multiplicação entre matriz transposta e vetor foi discutida na Seção 3.4.2. Foram implementadas quatro versões de *Transposed MatVec* usando *dataflow*. Além das versões *ordered* e *relaxed*, há também as versões *ordered SG* e *relaxed SG*, que são similares às suas homônimas mas que usam *generators* e o conceito de subgrafos discutido do Capítulo 3.

Como a flutuação de tempo dessa operação em rodadas subsequentes é significativa, cada teste foi executado **cinco** vezes, e o tempo considerado nos gráficos e tabelas é a **mediana** das cinco rodadas. Além disso, cada medição de tempo total foi feita com barreiras na ferramenta de *benchmark*, de modo a capturar deslocamentos temporais entre os *ranks MPI*.

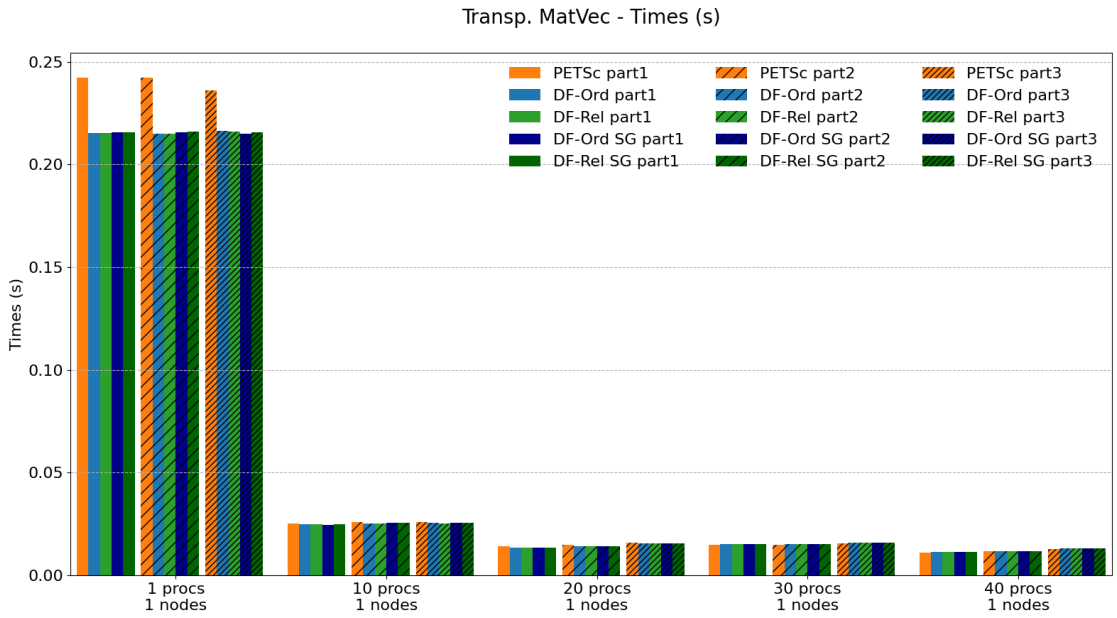
Em todas as tabelas e gráficos desta seção, os nomes PETSc, DF-Ord, DF-Rel, DF-Ord SG e DF-Rel SG se referem às operações Transp. MatVec do PETSc, *dataflow ordered*, *dataflow relaxed*, *dataflow ordered with sub-graphs* e *dataflow relaxed with sub-graphs*, respectivamente.

Matriz de ordem 1 milhão (bloco 3x3), 1 nó

O *grid* usado nesses casos possui 2000x50x10 = 1.000.000 de elementos. Tal discrepância de tamanhos entre o eixo x e o eixo y não é muito usual, mas era necessária para se obter até 2000 domínios mesmo particionando em apenas no eixo x. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem

global é (2000+1)x(50+1)x(10+1) = 1.122.561 blocos, lembrando que cada entrada da matriz é composta por blocos com 3x3 valores.

Figura 4.13: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse



Uma matriz dessa ordem é pequena para as aplicações alvo da Petrobras. Ela ocuparia, inteira (sem particionar), menos 3 GB de memória, ou seja, menos de 1% dos 384 GB de RAM de um único nó do *cluster*. Portanto, os casos de interesse para matrizes desse tamanho são as rodadas intra-nó (de 1 a 40 processos). Extrapolações para até 2000 processos (50 nós computacionais) são apresentados na Seção 4.3.2.

A Tabela 4.4 compara os tempos da operação Transp. MatVec entre o PETSc e as versões *dataflow* (*ordered*, *relaxed*, *ordered SG* e *relaxed SG*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.13 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.14 e 4.15 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Em todas as estratégias de particionamento do *grid*, os 4 núcleos *dataflow* iniciaram com um *speed-up* de 1.1x em relação ao PETSc na rodada serial e terminaram com 1.0x usando todos os 40 cores da máquina. O PETSc escalou um pouco melhor principalmente em função do desempenho 10% inferior no caso serial, mas em tempos absolutos, todas as implementações *dataflow* tiveram desempenho iguais ou superiores em todas as rodadas de todas as estratégias de particionamento. Não foi possível observar diferença significativa entre núcleos *dataflow*.

Tabela 4.4: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

Tabela 4.5: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	$\frac{\text{PETSc}}{\text{DF-Ord}}$	$\frac{\text{PETSc}}{\text{DF-Rel}}$	$\frac{\text{PETSc}}{\text{DF-Ord SG}}$	$\frac{\text{PETSc}}{\text{DF-Rel SG}}$
1	1	0.2422	0.2154	0.2155	0.2157	0.2156	1.1	1.1	1.1	1.1
10	1	0.0253	0.0248	0.0248	0.0247	0.0248	1.0	1.0	1.0	1.0
20	1	0.0142	0.0135	0.0135	0.0134	0.0134	1.1	1.1	1.1	1.1
30	1	0.0150	0.0151	0.0151	0.0151	0.0151	1.0	1.0	1.0	1.0
40	1	0.0113	0.0113	0.0114	0.0114	0.0113	1.0	1.0	1.0	1.0

Tabela 4.6: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	$\frac{\text{PETSc}}{\text{DF-Ord}}$	$\frac{\text{PETSc}}{\text{DF-Rel}}$	$\frac{\text{PETSc}}{\text{DF-Ord SG}}$	$\frac{\text{PETSc}}{\text{DF-Rel SG}}$
1	1	0.2423	0.2149	0.2149	0.2156	0.2161	1.1	1.1	1.1	1.1
10	1	0.0259	0.0254	0.0252	0.0255	0.0254	1.0	1.0	1.0	1.0
20	1	0.0149	0.0141	0.0140	0.0143	0.0142	1.1	1.1	1.0	1.1
30	1	0.0148	0.0151	0.0152	0.0151	0.0151	1.0	1.0	1.0	1.0
40	1	0.0116	0.0118	0.0118	0.0118	0.0117	1.0	1.0	1.0	1.0

Tabela 4.7: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	$\frac{\text{PETSc}}{\text{DF-Ord}}$	$\frac{\text{PETSc}}{\text{DF-Rel}}$	$\frac{\text{PETSc}}{\text{DF-Ord SG}}$	$\frac{\text{PETSc}}{\text{DF-Rel SG}}$
1	1	0.2360	0.2162	0.2162	0.2151	0.2155	1.1	1.1	1.1	1.1
10	1	0.0259	0.0255	0.0253	0.0256	0.0256	1.0	1.0	1.0	1.0
20	1	0.0161	0.0155	0.0154	0.0155	0.0156	1.0	1.0	1.0	1.0
30	1	0.0157	0.0159	0.0159	0.0159	0.0160	1.0	1.0	1.0	1.0
40	1	0.0128	0.0131	0.0131	0.0130	0.0131	1.0	1.0	1.0	1.0

Figura 4.14: Gráficos de speed-up da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

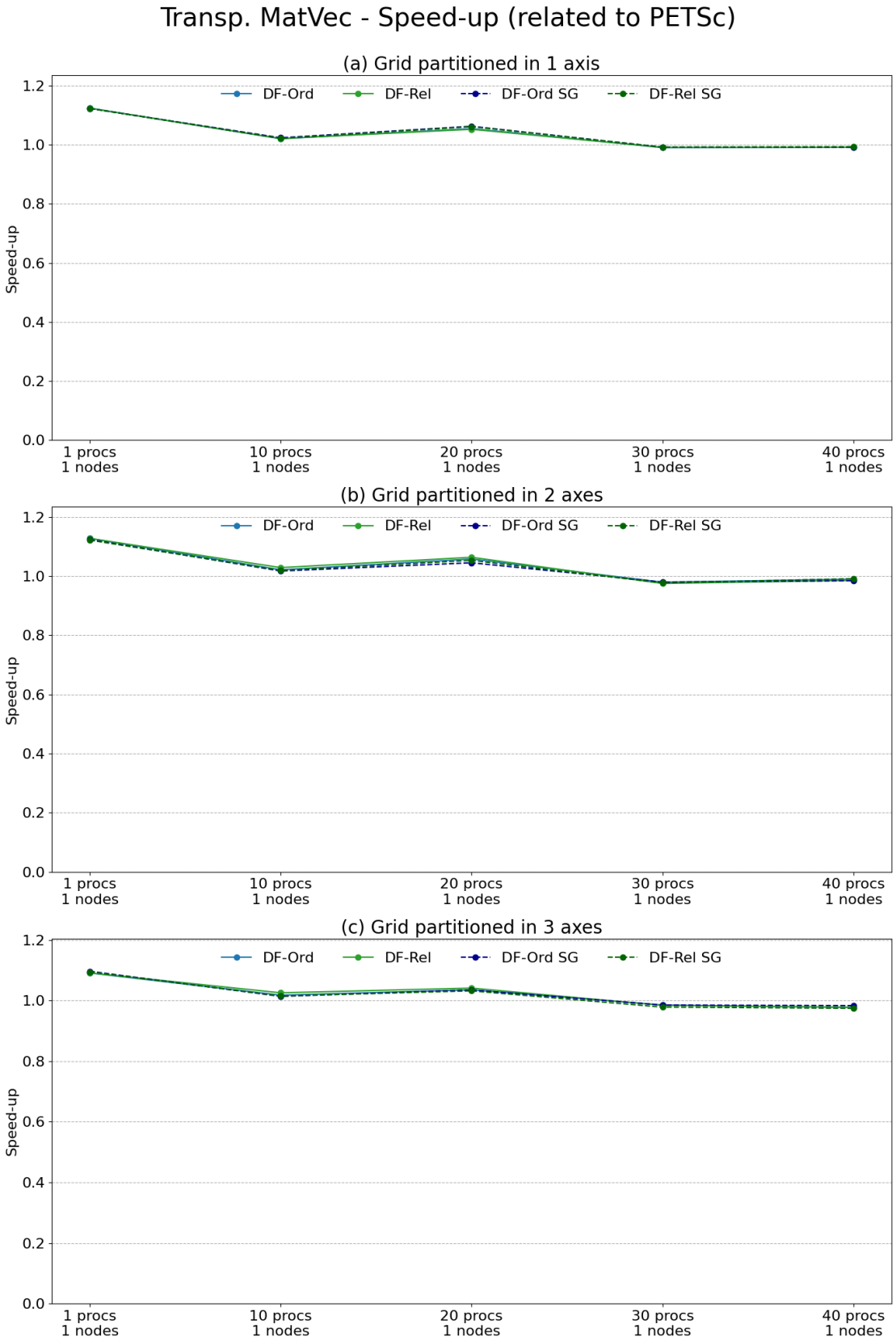
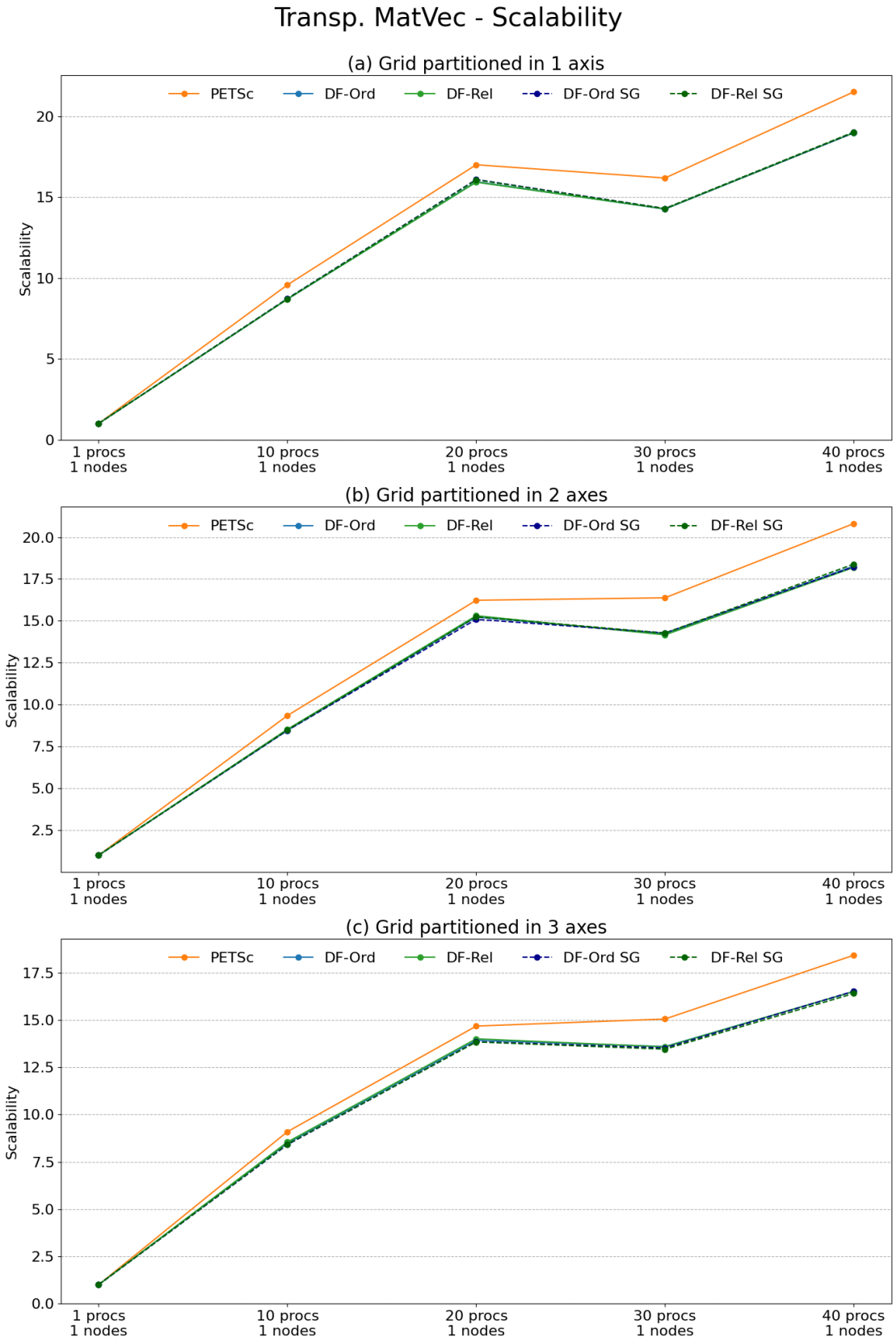


Figura 4.15: Gráficos de escalabilidade da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

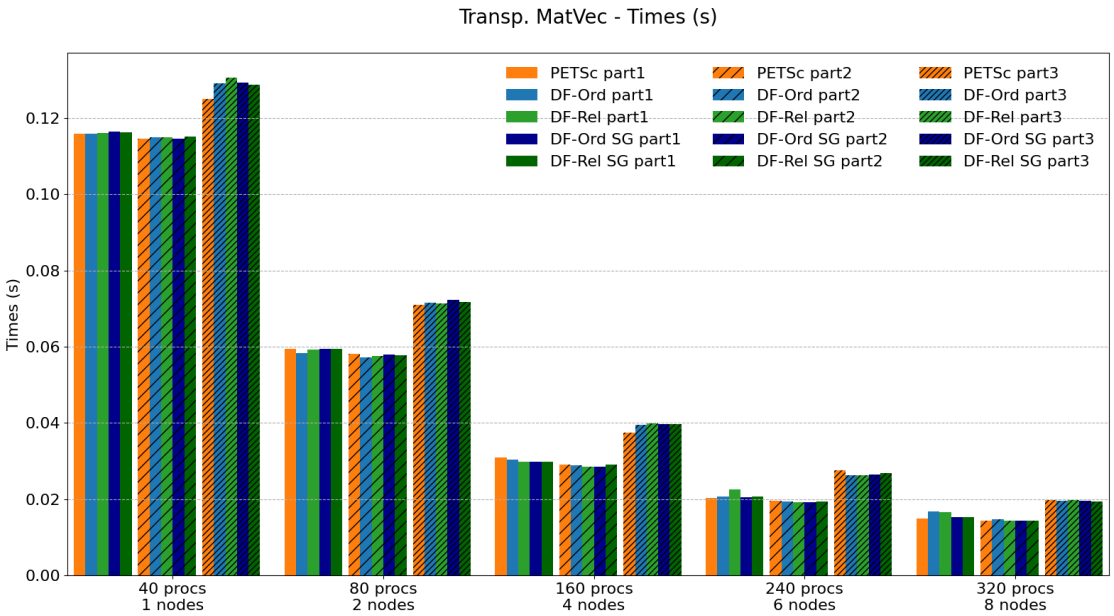


Matriz de ordem 10 milhões (bloco 3x3), até 8 nós

O *grid* usado nesses casos possui 2000x500x10 = 10.000.000 de elementos. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(500+1)x(10+1) = 11.027.511 blocos, lembrando que cada entrada da matriz é composta por blocos 3x3.

Uma matriz dessa ordem ocupa menos de 30 GB de memória, ainda longe dos 384 GB de RAM de um único nó do *cluster* Guaricema. No entanto, um simulador rodando problemas desse tamanho pode ser executado em múltiplos nós (2, 3 ou 4 nós) por questão de desempenho. Por isso, estendemos generosamente as rodadas de interesse para até 8 nós nesse caso. Extrapolações para até 50 nós (2000 processos) são apresentadas na Seção 4.3.2.

Figura 4.16: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse



A Tabela 4.8 compara os tempos da operação Transp. MatVec entre o PETSc e as versões *dataflow* (*ordered*, *relaxed*, *ordered SG* e *relaxed SG*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.16 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.17 e 4.18 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Com particionamento do *grid* gerador da matriz em 1 eixo coordenado, todos os núcleos *dataflow* começam com desempenho igual ao do PETSc usando 40 processos (1 nó cheio). Porém, as versões DF-Ord e DF-Rel terminam a rodada com 320 processos (8 nós) 0.9x. Já os núcleos correspondentes que fazem uso da heurística de subgrafos, conseguiram igualar o desempenho do PETSc em todas as rodadas até

Tabela 4.8: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

Tabela 4.9: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
40	1	0.1158	0.1160	0.1161	0.1164	0.1162	1.0	1.0	1.0	1.0
80	2	0.0595	0.0583	0.0592	0.0594	0.0594	1.0	1.0	1.0	1.0
160	4	0.0310	0.0304	0.0299	0.0299	0.0298	1.0	1.0	1.0	1.0
240	6	0.0203	0.0207	0.0226	0.0206	0.0207	1.0	0.9	1.0	1.0
320	8	0.0149	0.0168	0.0165	0.0153	0.0153	0.9	0.9	1.0	1.0

Tabela 4.10: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
40	1	0.1146	0.1149	0.1149	0.1146	0.1152	1.0	1.0	1.0	1.0
80	2	0.0582	0.0572	0.0576	0.0580	0.0577	1.0	1.0	1.0	1.0
160	4	0.0290	0.0288	0.0284	0.0284	0.0291	1.0	1.0	1.0	1.0
240	6	0.0196	0.0194	0.0192	0.0193	0.0194	1.0	1.0	1.0	1.0
320	8	0.0143	0.0147	0.0144	0.0144	0.0144	1.0	1.0	1.0	1.0

Tabela 4.11: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
40	1	0.1250	0.1291	0.1307	0.1292	0.1287	1.0	1.0	1.0	1.0
80	2	0.0710	0.0716	0.0714	0.0723	0.0717	1.0	1.0	1.0	1.0
160	4	0.0375	0.0396	0.0399	0.0397	0.0397	0.9	0.9	0.9	0.9
240	6	0.0276	0.0263	0.0263	0.0264	0.0268	1.0	1.1	1.0	1.0
320	8	0.0198	0.0197	0.0197	0.0196	0.0195	1.0	1.0	1.0	1.0

Figura 4.17: Gráficos de speed-up da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

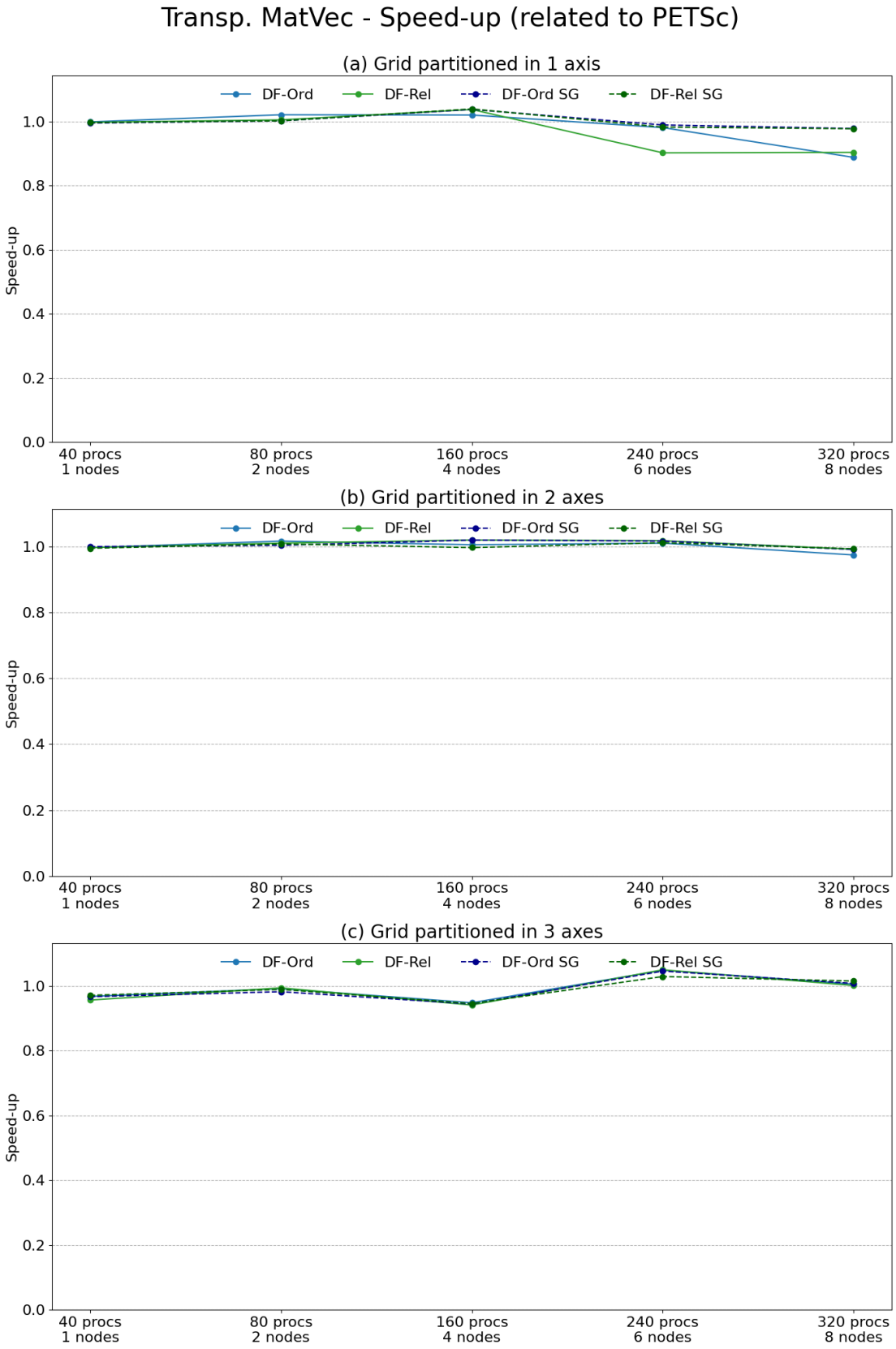
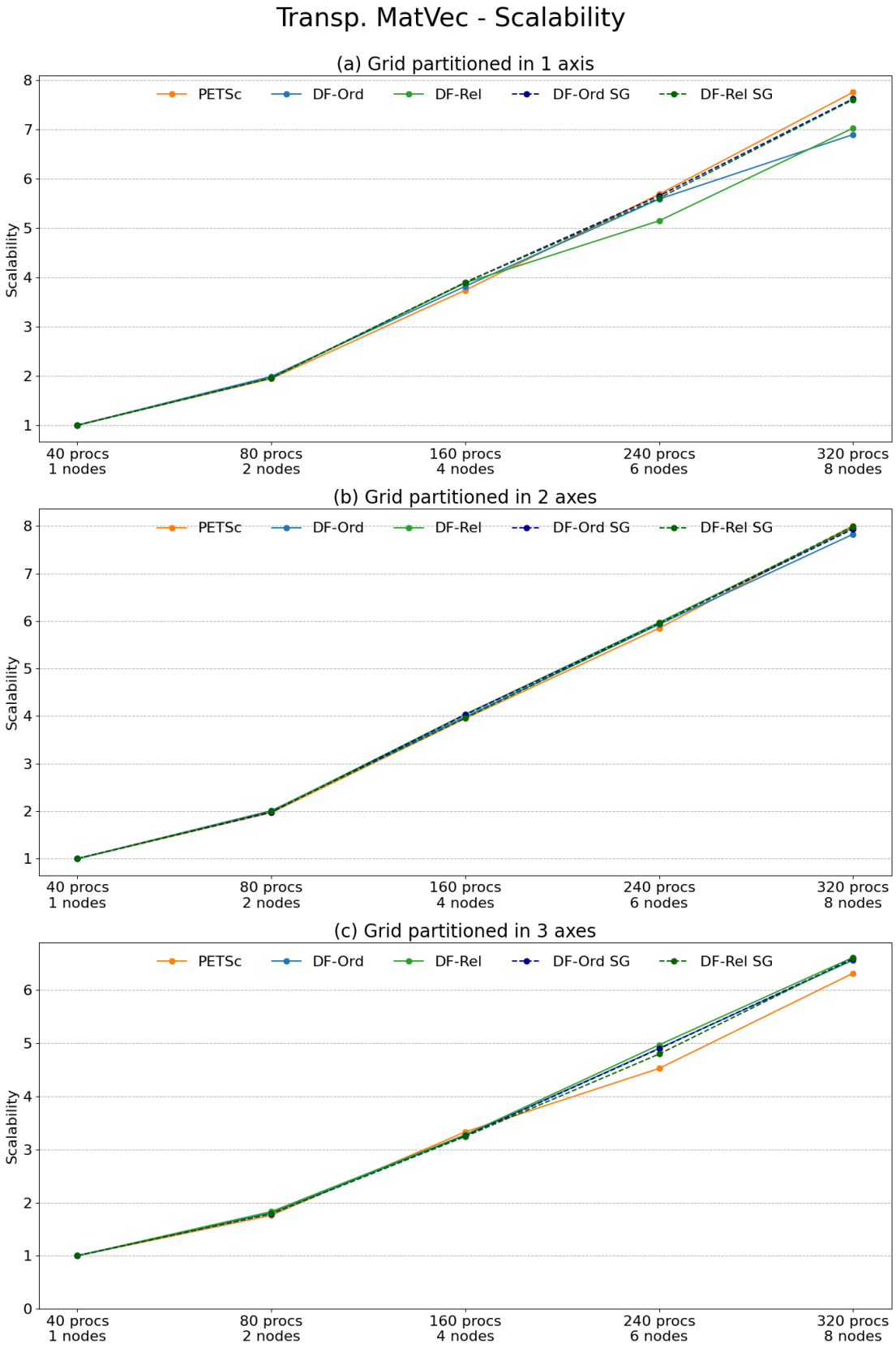


Figura 4.18: Gráficos de escalabilidade da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse



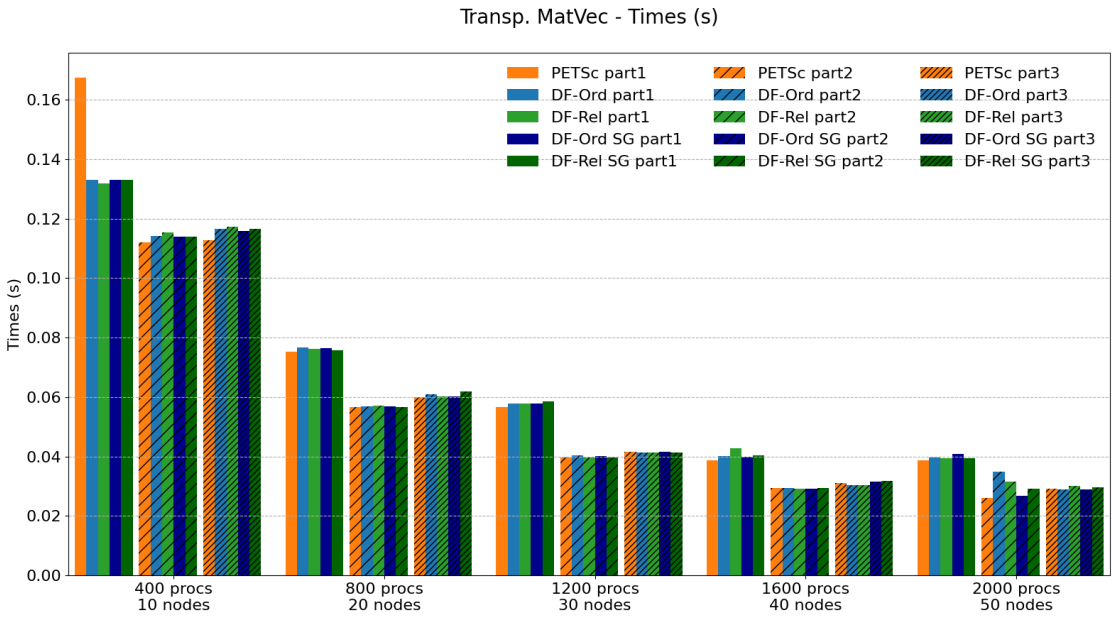
o final (ver Figura 4.17, item a). Nos demais esquemas de particionamento do *grid*, todos os núcleos *dataflow* tiveram desempenho igual ao PETSc em quase todas as rodadas, com exceção do particionamento em 3 eixos rodando com 160 processos (4 nós). Nesse caso, curiosamente todos os núcleos *dataflow* tiveram desempenho 0.9x em relação ao PETSc. Como os tempos são muito próximos em geral, a escalabilidade de todos os casos foi muito parecida (ver Figura 4.18)

Matriz de ordem 100 milhões (bloco 3x3), até 50 nós

O *grid* usado nesses casos possui 2000x500x100 = 100.000.000 de elementos. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(500+1)x(100+1) = 101.252.601 blocos, lembrando que cada entrada da matriz é composta por blocos 3x3.

Uma matriz dessa ordem ocupa menos de 300 GB de memória, valor já próximo dos 384 GB de RAM de um único nó do *cluster* Guaricema. Na prática (no contexto de um simulador) problemas desse tamanho precisam ser executados em múltiplos nós. As rodadas desse caso usaram de 10 a 50 nós computacionais (de 400 a 2000 processos). O usual seria usar até 20 nós, mas optou-se por considerar todas as rodadas, até 50 nós, como casos de interesse porque há apenas 5 rodadas no total.

Figura 4.19: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



A Tabela 4.12 compara os tempos da operação Transp. MatVec entre o PETSc e as versões *dataflow* (*ordered*, *relaxed*, *ordered SG* e *relaxed SG*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.19 exibe os tempos de cada implementação agrupados por número de processos

Tabela 4.12: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

Tabela 4.13: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	$\frac{\text{PETSc}}{\text{DF-Ord}}$	$\frac{\text{PETSc}}{\text{DF-Rel}}$	$\frac{\text{PETSc}}{\text{DF-Ord SG}}$	$\frac{\text{PETSc}}{\text{DF-Rel SG}}$
400	10	0.1675	0.1329	0.1319	0.1330	0.1330	1.3	1.3	1.3	1.3
800	20	0.0753	0.0766	0.0762	0.0763	0.0758	1.0	1.0	1.0	1.0
1200	30	0.0566	0.0577	0.0579	0.0577	0.0585	1.0	1.0	1.0	1.0
1600	40	0.0388	0.0401	0.0428	0.0399	0.0403	1.0	0.9	1.0	1.0
2000	50	0.0388	0.0396	0.0395	0.0409	0.0395	1.0	1.0	0.9	1.0

Tabela 4.14: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	$\frac{\text{PETSc}}{\text{DF-Ord}}$	$\frac{\text{PETSc}}{\text{DF-Rel}}$	$\frac{\text{PETSc}}{\text{DF-Ord SG}}$	$\frac{\text{PETSc}}{\text{DF-Rel SG}}$
400	10	0.1121	0.1141	0.1153	0.1140	0.1138	1.0	1.0	1.0	1.0
800	20	0.0566	0.0568	0.0572	0.0568	0.0566	1.0	1.0	1.0	1.0
1200	30	0.0395	0.0405	0.0396	0.0400	0.0398	1.0	1.0	1.0	1.0
1600	40	0.0294	0.0295	0.0292	0.0292	0.0294	1.0	1.0	1.0	1.0
2000	50	0.0259	0.0350	0.0315	0.0269	0.0292	0.7	0.8	1.0	0.9

Tabela 4.15: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	$\frac{\text{PETSc}}{\text{DF-Ord}}$	$\frac{\text{PETSc}}{\text{DF-Rel}}$	$\frac{\text{PETSc}}{\text{DF-Ord SG}}$	$\frac{\text{PETSc}}{\text{DF-Rel SG}}$
400	10	0.1127	0.1165	0.1172	0.1159	0.1165	1.0	1.0	1.0	1.0
800	20	0.0601	0.0609	0.0601	0.0603	0.0618	1.0	1.0	1.0	1.0
1200	30	0.0415	0.0414	0.0413	0.0415	0.0414	1.0	1.0	1.0	1.0
1600	40	0.0312	0.0305	0.0304	0.0315	0.0318	1.0	1.0	1.0	1.0
2000	50	0.0291	0.0289	0.0301	0.0289	0.0296	1.0	1.0	1.0	1.0

Figura 4.20: Gráficos de speed-up da operação Transp. MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

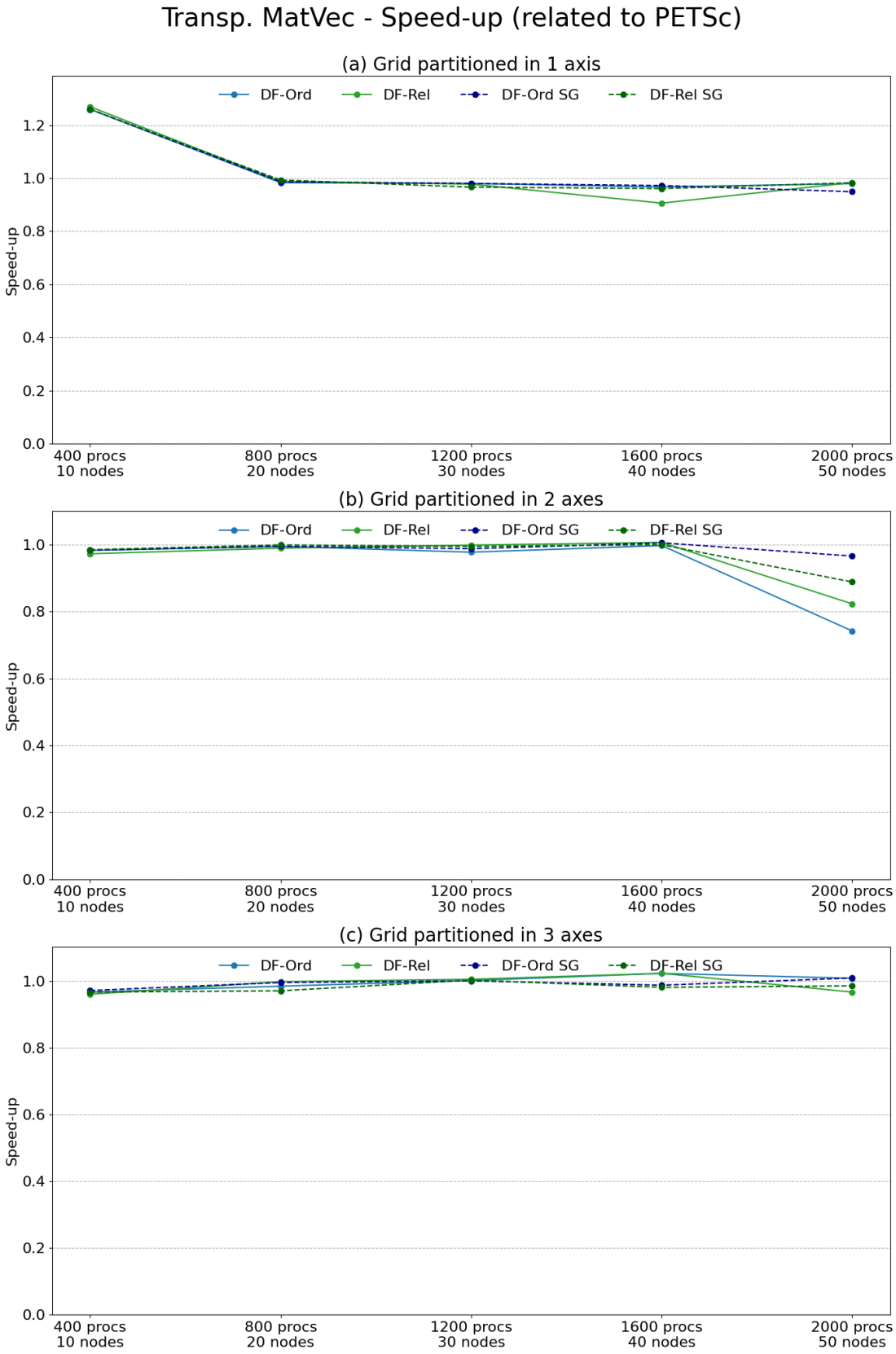
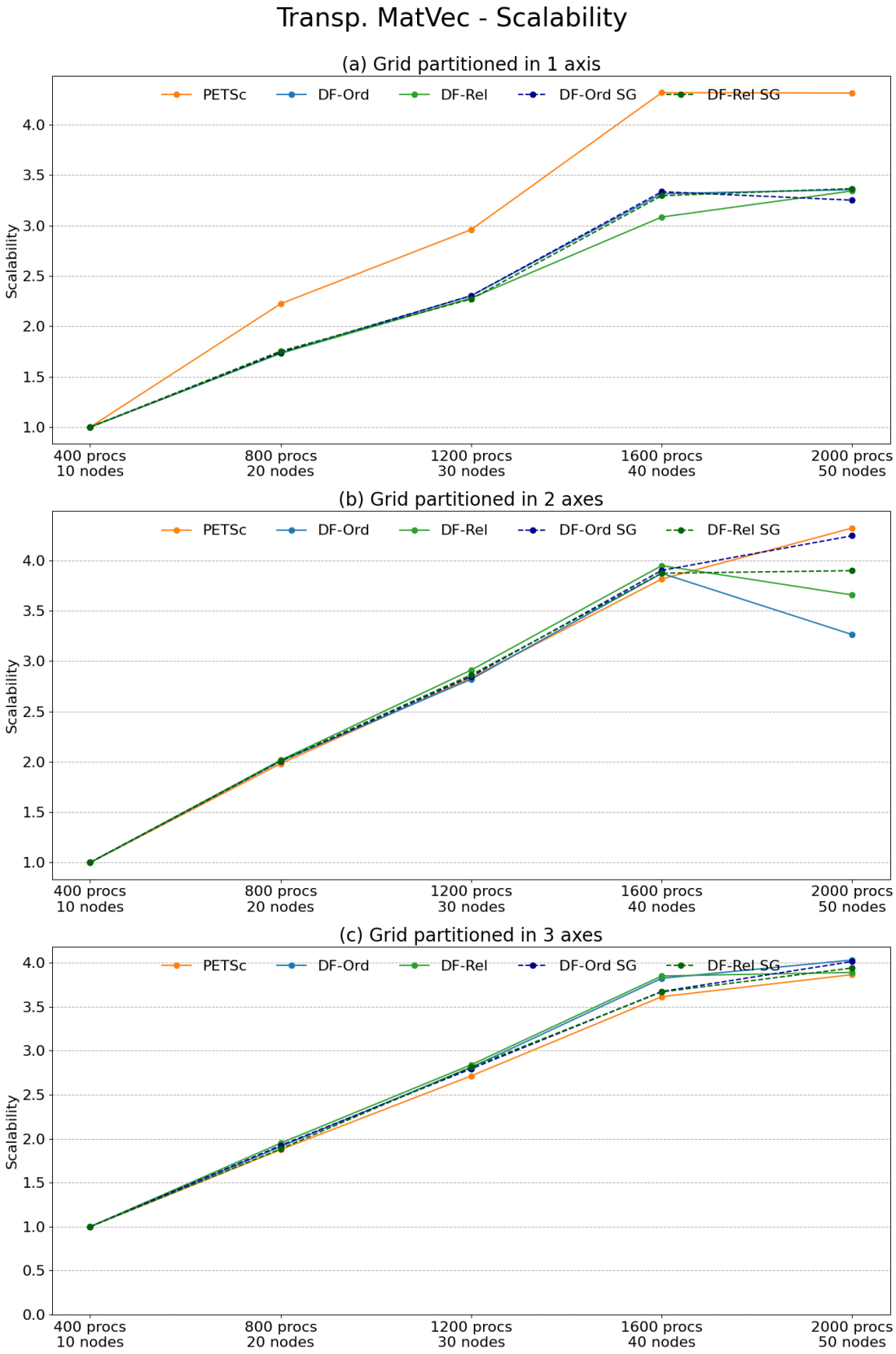


Figura 4.21: Gráficos de escalabilidade da operação Transp. MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



e por estratégia de particionamento. As figuras 4.20 e 4.21 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

A exemplo do caso anterior, particionando-se o *grid* gerador da matriz em 1 eixo coordenado, todos os núcleos *dataflow* partem de um *speed-up* considerável de 1.3x em relação ao PETSc com 400 processos (10 nós), mas terminam com *speed-up* 1.0x em 3 dos 4 núcleos *dataflow* (o DF-Ord SG terminou com 0.9x). Novamente, a melhor escalabilidade do PETSc se explica pelo desempenho inferior na rodada base com 400 processos. Particionando em 2 eixos, os núcleos *dataflow* se equivalem ao PETSc nas rodadas de 10 até 40 nós (400 a 1600 processos), mas nota-se uma deterioração do desempenho de alguns núcleos *dataflow* com 50 nós (2000 processos). Nesse caso, as implementações com subgrafos foram menos afetadas, inclusive com o DF-Ord SG igualando a performance do PETSc em todos os casos. Com particionamento em 3 eixos, todos os núcleos *dataflow* tiveram desempenho similar ao PETSc em todas as rodadas.

4.2.3 Operação $\mathbf{C = P^T AP}$

O PtAP foi discutida na Seção 3.4.3. Foram testadas duas versões de produto triplo de matrizes usando o mesmo *dataflow*: DF-FG (*Full Graph*, que executa o grafo completo) e o DF-SG (*Sub-graph*, que usa a heurística de subgrafos).

A matriz A é criada a partir do *grid* original, tal como nas seções anteriores. A matriz P é uma matriz de prolongamento, que associa cada nó do *grid* original a um nó de um *grid* grosseiro (que agrupa elementos do *grid* original em elementos grosseiros). Em todas as rodadas foram usados elementos grosseiros $4 \times 4 \times 4$, ou seja, 1 elemento grosseiro agrupa $4 \times 4 \times 4 = 64$ elementos do *grid* original. Dessa forma, a matriz P é retangular, onde o número de linhas é igual ao da matriz A e o número de colunas é igual ao número de nós do *grid* grosseiro. A matriz P^T, chamada de matriz de restrição, é a P transposta logicamente. No contexto de um simulador, a construção do P pode ser bastante complexa [40], mas para a operação em questão os valores nas entradas de P podem ser quais quer valores numéricos (os valores afetam a convergência, mas não os tempos do produto triplo $P^T AP$).

Essa operação tem uma carga computacional bem superior ao das operações MatVec e Trasnp. MatVec, por isso não apresenta tanta variabilidade de desempenho em rodadas subsequentes. Logo, cada caso foi executado uma única vez durante os testes. A Tabela 4.16 mostra o quão intensivo o núcleo PtAP pode ser a depender do particionamento. Ela traz, para cada estratégia de particionamento, o número médio de tarefas locais nos grafos das implementações *dataflow* quando executados com 2000 processos. Por exemplo, particionando o grid em 3 eixos coordenados, em cada rank MPI são executadas em média 11854 tarefas, das quais cerca de 5240

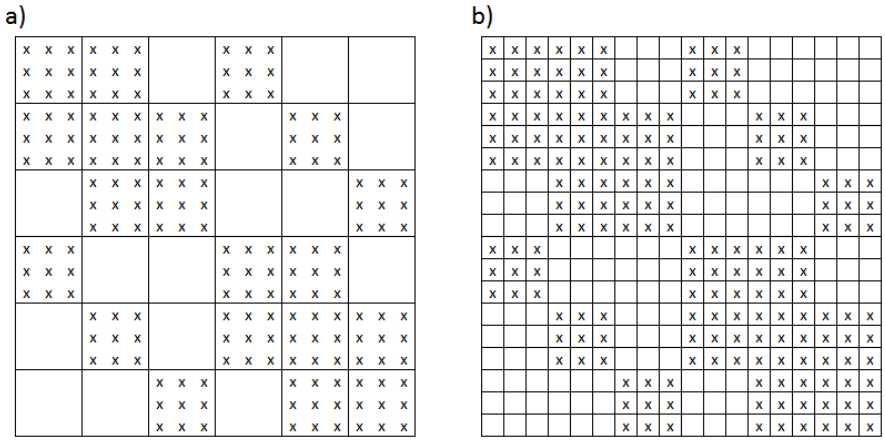
são tarefas de comunicação (GET e PUT) que podem conter uma ou mais rotinas MPI. Os tempos em todos os gráficos dessa seção incluem exatamente 1 fatoração simbólica e 1 fatoração numérica (em linha com o GeomecBR, que executa cada fatoração uma única vez em cada simulação para o precondicionador multiescala).

Em todas as tabelas e gráficos desta seção, os nomes PETSc, DF-FG e DF-SG se referem às operações PtAP do PETSc, *dataflow with full graph* e *dataflow with sub-graphs*, respectivamente. Como a quantidade de tarefas desse núcleo pode ser bem grande, os tempos de *setup* das implementações *dataflow* podem ser significativos. Por isso, nos gráficos de barras os tempos de *setup* dos núcleos DF-FG e DF-SG foram destacados em cor mais escura.

Tabela 4.16: Número médio de tarefas e dependências locais nos grafos das implementações *dataflow* do núcleo PtAP com 2000 processos por estratégia de particionamento

#Procs	#Axis Part.	AVG #Tasks per process	AVG #CommTasks per process	AVG #Deps per process	Total #Tasks	Total #CommTasks	Total #Deps
2000	1	96	28	95	192000	56000	190000
2000	2	1284	506	1173	2568000	1012000	2346000
2000	3	11854	5240	11124	23708000	10480000	22248000

Figura 4.22: Ilustração de uma matriz blocada (3x3) convertida em matriz escalar



Matriz de ordem 1 milhão (bloco 3x3), 1 nó

O *grid* usado nesses casos possui 2000x50x10 = 1.000.000 de elementos. Tal discrepância de tamanhos entre o eixo x e o eixo y não é muito usual, mas era necessária para se obter até 2000 domínios mesmo particionando em apenas um eixo. Como as entradas da matriz A são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(50+1)x(10+1) = 1.122.561 blocos de tamanho 3x3. As matrizes

Tabela 4.17: Tempos e Speed-up da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
1	1	34.7625	30.1961	29.6901	1.2	1.2
10	1	3.5310	3.6415	3.6038	1.0	1.0
20	1	1.8111	1.8066	1.8257	1.0	1.0
30	1	1.5178	1.4682	1.5431	1.0	1.0
40	1	1.1513	1.1052	1.1591	1.0	1.0

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
1	1	34.8659	30.2023	29.6792	1.2	1.2
10	1	3.7077	4.5465	4.1927	0.8	0.9
20	1	1.9995	2.5793	2.5836	0.8	0.8
30	1	1.6815	2.0508	2.0862	0.8	0.8
40	1	1.2616	1.5184	1.6406	0.8	0.8

(c) Grid partitioned in 3 axes

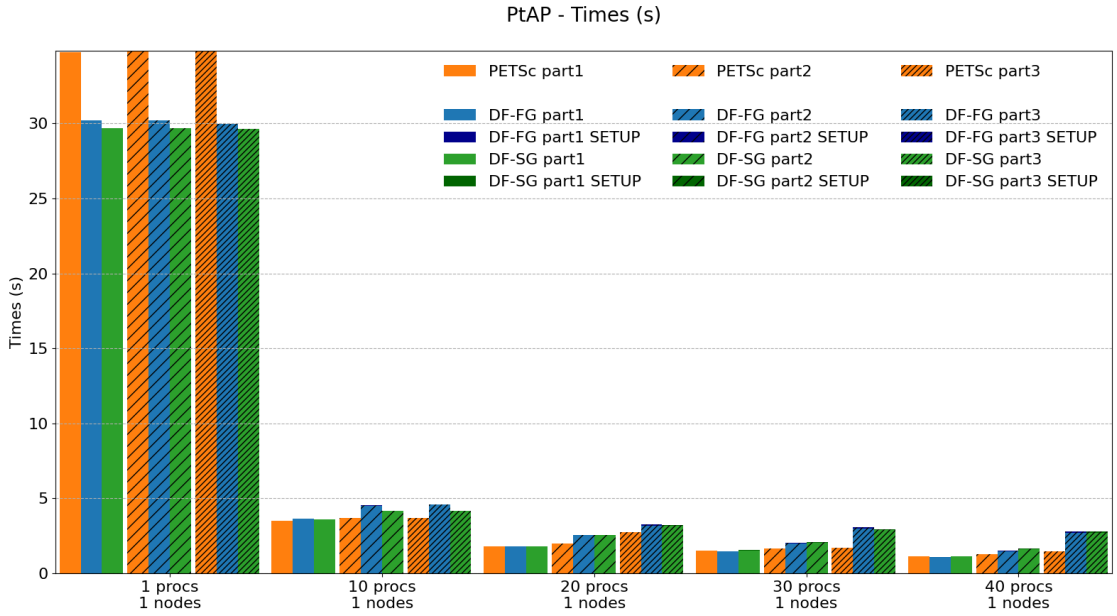
		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
1	1	34.8144	29.9917	29.6341	1.2	1.2
10	1	3.6932	4.6051	4.1902	0.8	0.9
20	1	2.7274	3.2567	3.2442	0.8	0.8
30	1	1.6974	3.0736	2.9486	0.6	0.6
40	1	1.4661	2.7895	2.7942	0.5	0.5

A e P originais foram convertidas em escalares (endereçando cada valor dentro dos blocos) porque o PETSc não suporta a operação PtAP para matrizes *blocadas*. A Figura 4.22 mostra um exemplo simples de uma matriz blocada (3x3), item (a), convertida em matriz escalar, item (b).

Uma matriz dessa ordem é pequena para as aplicações alvo da Petrobras. Ela ocuparia, inteira (sem particionar), menos 3 GB de memória, ou seja, menos de 1% dos 384 GB de RAM de um único nó do *cluster*. Portanto, os casos de interesse para matrizes desse tamanho são as rodadas intra-nó (de 1 a 40 processos). Extrapolações para até 2000 processos (50 nós computacionais) são apresentados na Seção 4.3.3

A Tabela 4.17 compara os tempos da operação PtAP entre o PETSc e as versões *dataflow* (*full graph* e *sub-graphs*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.23 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.24 e 4.25 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Figura 4.23: Gráficos de tempos da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse



Particionando em 1 eixo coordenado, as duas implementações *dataflow* iniciam com *speed-up* de 1.2x em relação ao PETSc na rodada serial e terminam com *speed-up* 1.0x na rodada final com 40 processos. Embora a escalabilidade do PETSc tenha sido melhor, os núcleos *dataflow* tiveram desempenho igual ou superior em todas as rodadas. Particionando em 2 eixos, DF-FG e DF-SG perderam desempenho já com 10 processos e terminaram com *speed-up* de 0.8x. Com particionamento em 3 eixos, a degradação de desempenho dos núcleos *dataflow* foi ainda maior, com ambos terminando com 0.5x da performance do PETSc com 40 processos.

Em todos os casos, o tempo de *setup* dos núcleos *dataflow* foram quase imperceptíveis, por se tratar de rodadas intra-nó.

Figura 4.24: Gráficos de speed-up da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

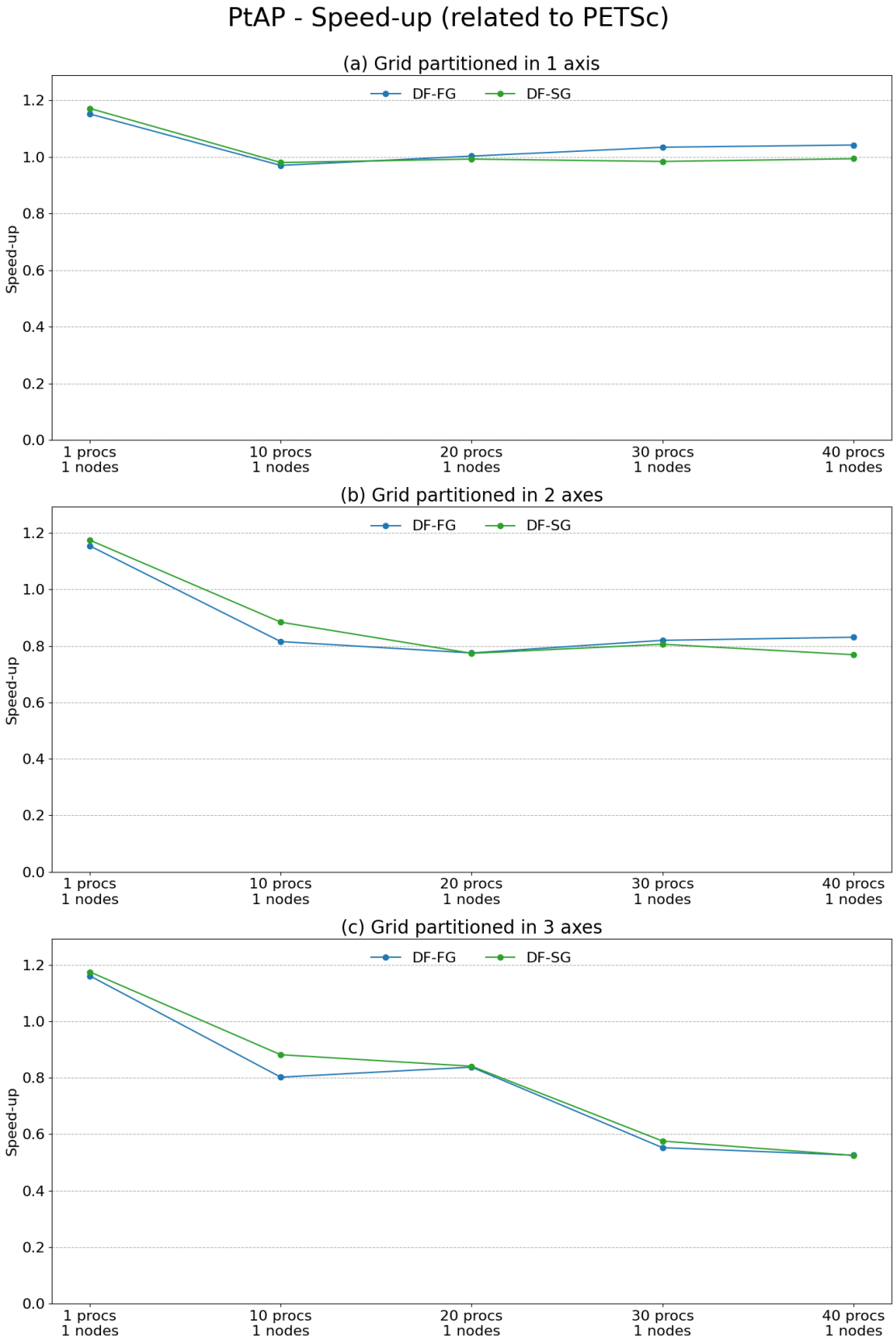
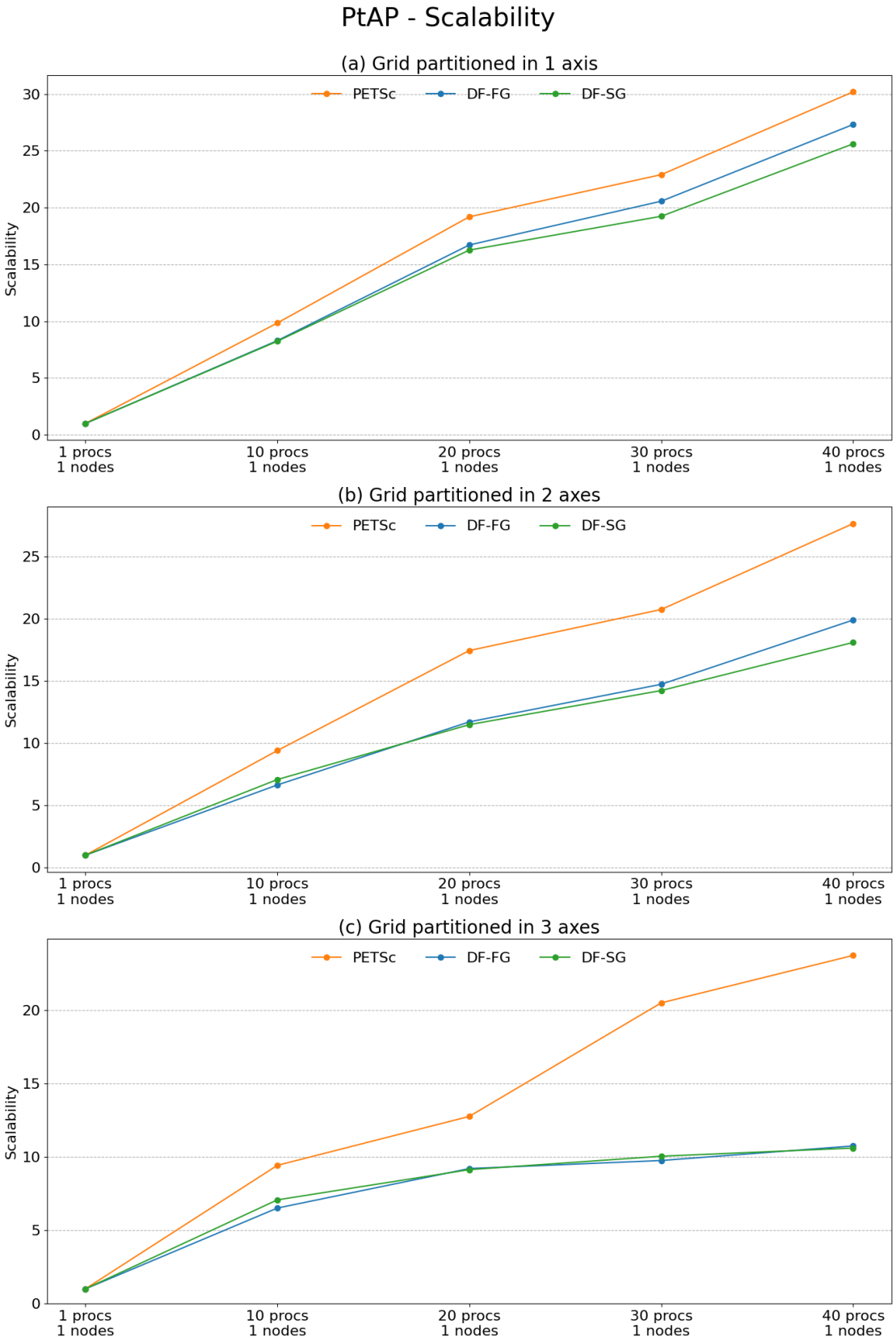


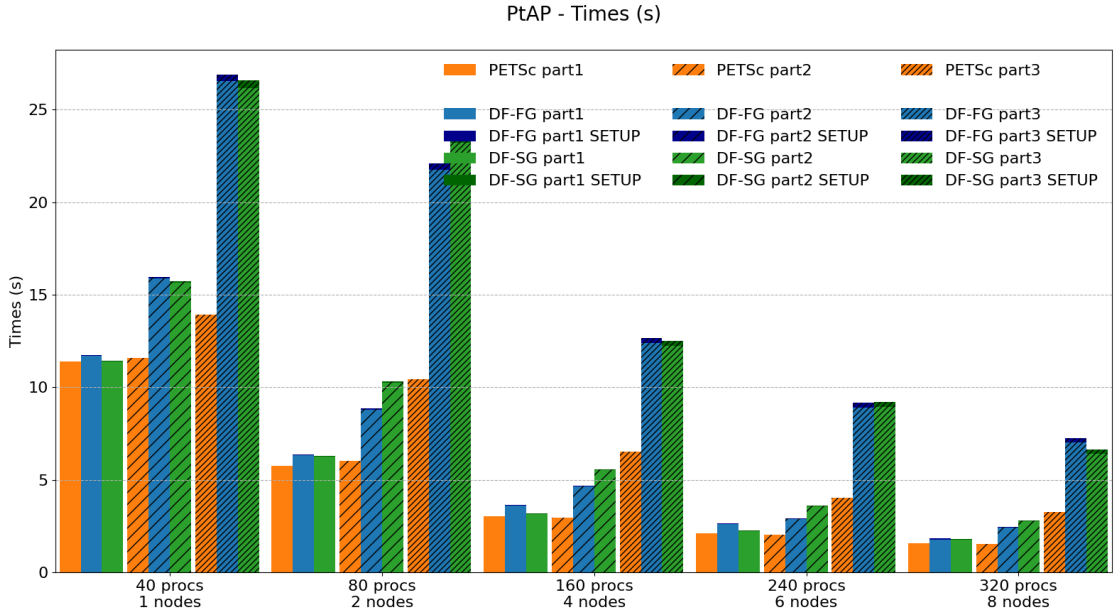
Figura 4.25: Gráficos de escalabilidade da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse



Matriz de ordem 10 milhões (bloco 3x3), até 8 nós

O *grid* usado nesses casos possui 2000x500x10 = 10.000.000 de elementos. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(500+1)x(10+1) = 11.027.511 blocos, lembrando que cada entrada da matriz é composta por blocos 3x3. As matrizes A e P originais foram convertidas em escalares (endereçando cada valor dentro dos blocos) porque o PETSc não suporta a operação PtAP para matrizes *blocadas*.

Figura 4.26: Gráficos de tempos da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse



Uma matriz dessa ordem ocupa menos de 30 GB de memória, ainda longe dos 384 GB de RAM de um único nó do *cluster* Guaricema. No entanto, um simulador rodando problemas desse tamanho pode ser executado em múltiplos nós (2, 3 ou 4 nós) por questão de desempenho. Por isso, estendemos generosamente as rodadas de interesse para até 8 nós nesse caso. Extrapolações para até 50 nós (2000 processos) são apresentadas na Seção 4.3.3.

A Tabela 4.18 compara os tempos da operação PtAP entre o PETSc e as versões *dataflow* (*full graph* e *sub-graphs*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.26 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.27 e 4.28 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Com particionamento em 1 eixo coordenado, os dois núcleos *dataflow* foram ligeiramente inferiores ao PETSc, porém competitivos em todas as rodadas de 40 a 320 processos (1 a 8 nós). Como os tempos foram próximos, a escalabilidade de

Tabela 4.18: Tempos e Speed-up da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
40	1	11.3834	11.7456	11.4407	1.0	1.0
80	2	5.7612	6.3895	6.3025	0.9	0.9
160	4	3.0312	3.6555	3.1850	0.8	1.0
240	6	2.1168	2.6344	2.2676	0.8	0.9
320	8	1.5745	1.8295	1.8250	0.9	0.9

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
40	1	11.6011	15.9448	15.7266	0.7	0.7
80	2	6.0289	8.8474	10.3083	0.7	0.6
160	4	2.9421	4.6699	5.5665	0.6	0.5
240	6	2.0528	2.9285	3.6134	0.7	0.6
320	8	1.5435	2.4433	2.7868	0.6	0.6

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
40	1	13.9379	26.9005	26.5779	0.5	0.5
80	2	10.4350	22.1016	23.5722	0.5	0.4
160	4	6.5428	12.6649	12.5162	0.5	0.5
240	6	4.0472	9.1563	9.2073	0.4	0.4
320	8	3.2648	7.2501	6.6455	0.5	0.5

Figura 4.27: Gráficos de speed-up da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

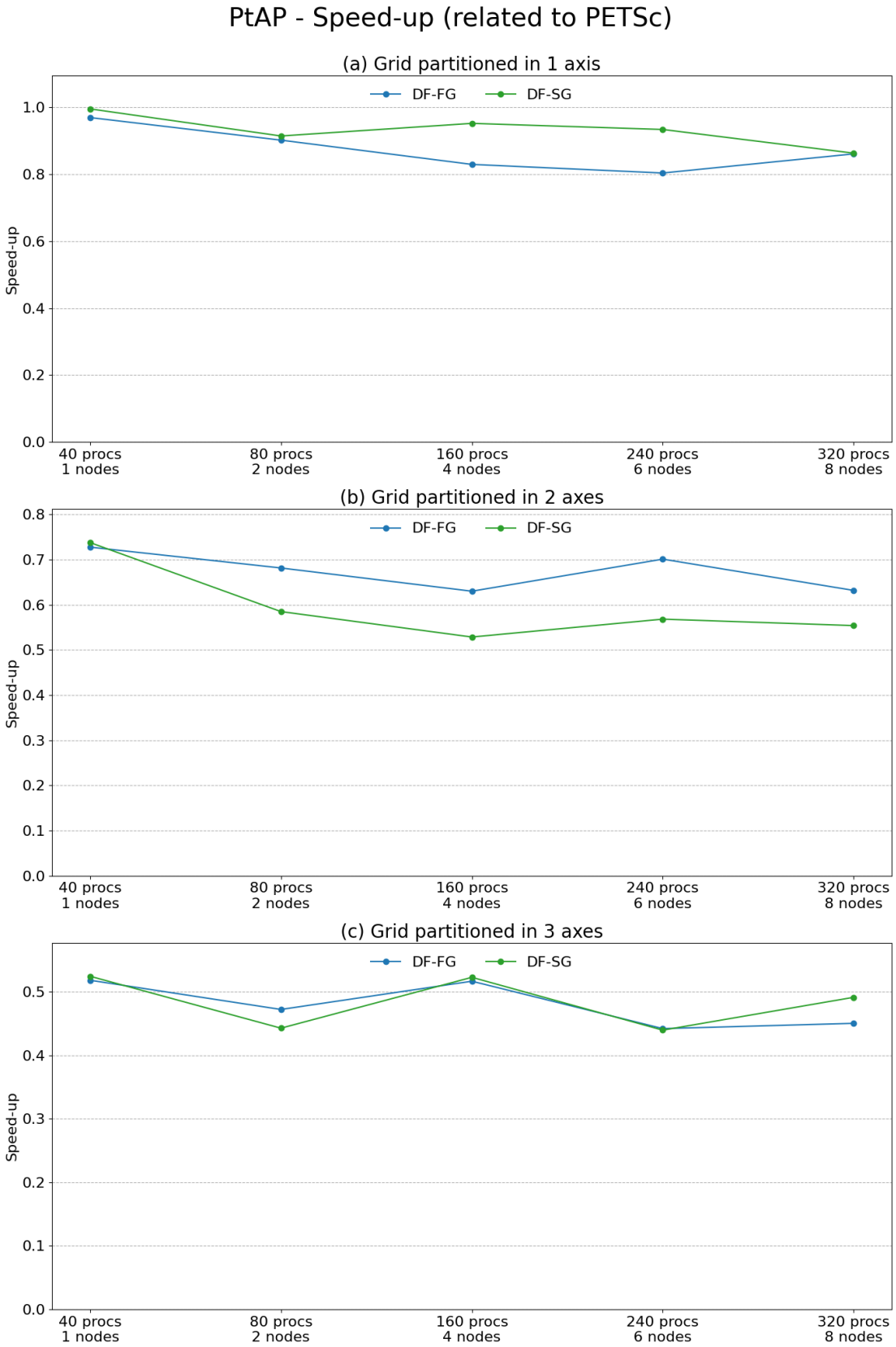
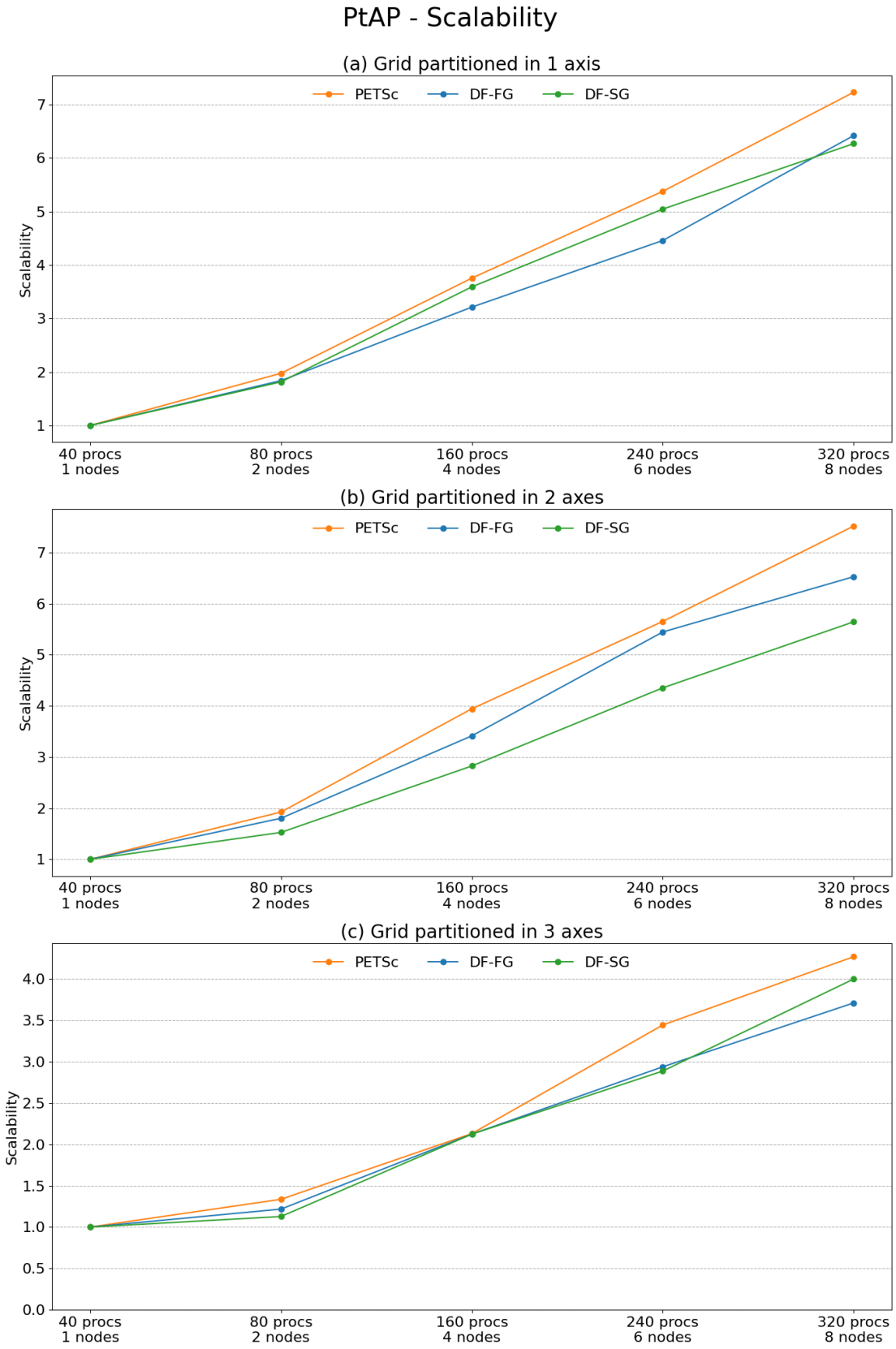


Figura 4.28: Gráficos de escalabilidade da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

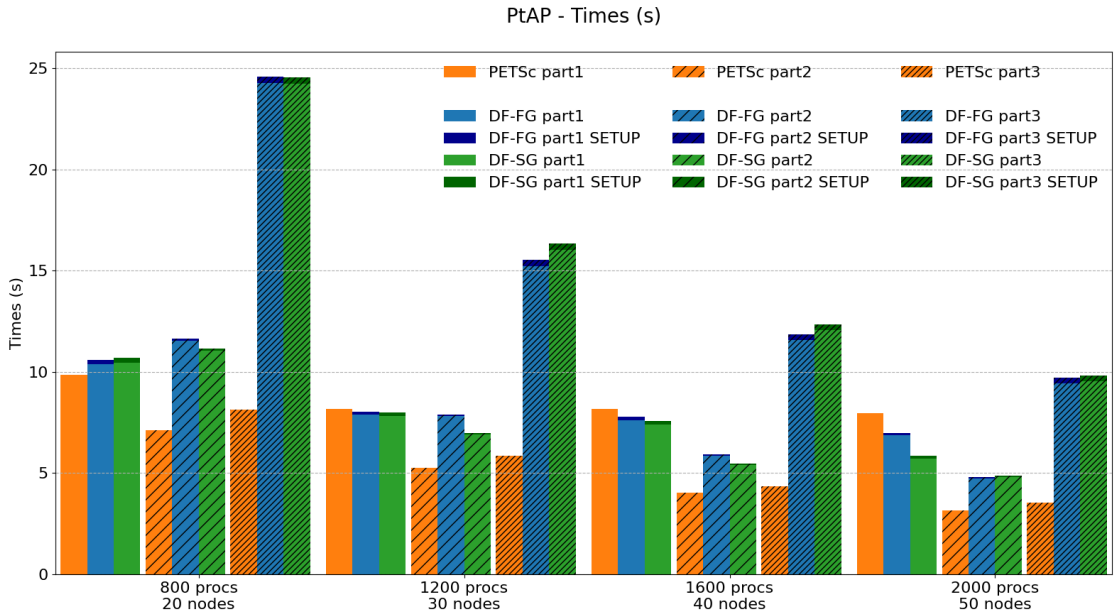


todos os núcleos foi similar. Com particionamento em 2 eixos, as versões *dataflow* iniciaram com *speed-up* 0.7x e terminaram com 0.6x em relação ao PETSc. Com particionamento em 3 eixos, o desempenho final de DF-FG e DF-SG fica em entre 0.4x e 0.5x em relação ao PETSc em todas as rodadas. Aqui, o tempo de setup começa a ficar perceptível, mas ainda é pequeno em relação ao tempo total.

Matriz de ordem 100 milhões (bloco 3x3), até 50 nós

O *grid* usado nesses casos possui 2000x500x100 = 100.000.000 de elementos. Como as entradas da matriz são em função dos nós do *grid*, não dos elementos, sua ordem global é (2000+1)x(500+1)x(100+1) = 101.252.601 blocos, lembrando que cada entrada da matriz é composta por blocos 3x3. As matrizes A e P originais foram convertidas em escalares (endereçando cada valor dentro dos blocos) porque o PETSc não suporta a operação PtAP para matrizes *blocadas*.

Figura 4.29: Gráficos de tempos da operação PtAP com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



Uma matriz dessa ordem ocupa menos de 300 GB de memória, valor já próximo dos 384 GB de RAM de um único nó do *cluster* Guaricema. Na prática (no contexto de um simulador) problemas desse tamanho precisam ser executados em múltiplos nós. As rodadas desse caso usaram de 20 a 50 nós computacionais (de 800 a 2000 processos), e todas as rodadas foram consideradas casos de interesse.

A Tabela 4.19 compara os tempos da operação PtAP entre o PETSc e as versões *dataflow* (*full graph* e *sub-graphs*) para diferentes números de processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.29 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de par-

Tabela 4.19: Tempos e Speed-up da operação PtAP com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
800	20	9.8416	10.5998	10.7054	0.9	0.9
1200	30	8.1683	8.0457	8.0077	1.0	1.0
1600	40	8.1872	7.7825	7.5693	1.1	1.1
2000	50	7.9563	6.9899	5.8449	1.1	1.4

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
800	20	7.1075	11.6364	11.1433	0.6	0.6
1200	30	5.2557	7.8871	6.9668	0.7	0.8
1600	40	4.0521	5.9305	5.4828	0.7	0.7
2000	50	3.1480	4.8037	4.8661	0.7	0.6

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
800	20	8.1482	24.5869	24.5285	0.3	0.3
1200	30	5.8567	15.5284	16.3415	0.4	0.4
1600	40	4.3399	11.8439	12.3495	0.4	0.4
2000	50	3.5516	9.7025	9.8339	0.4	0.4

Figura 4.30: Gráficos de speed-up da operação PtAP com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

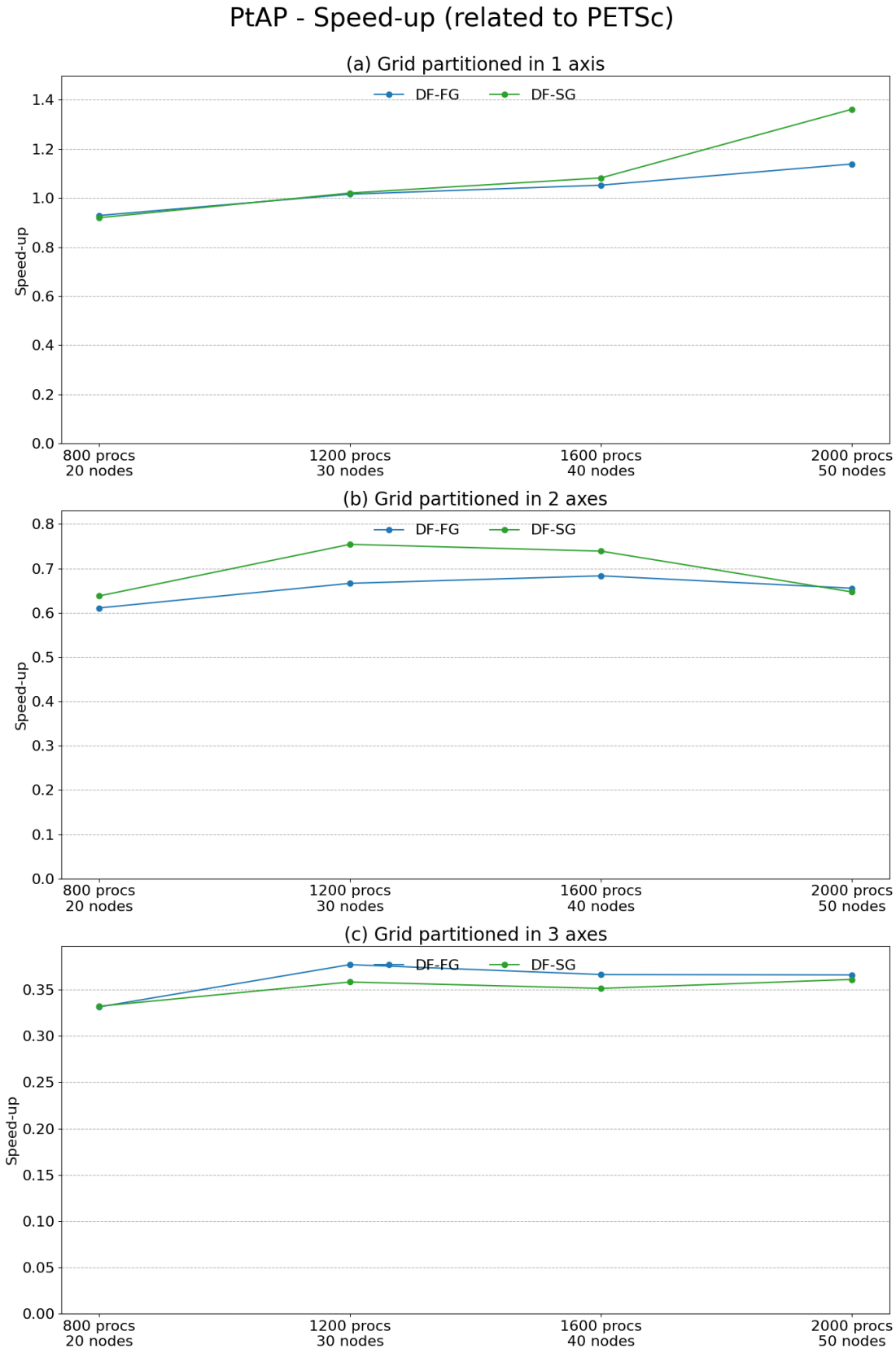
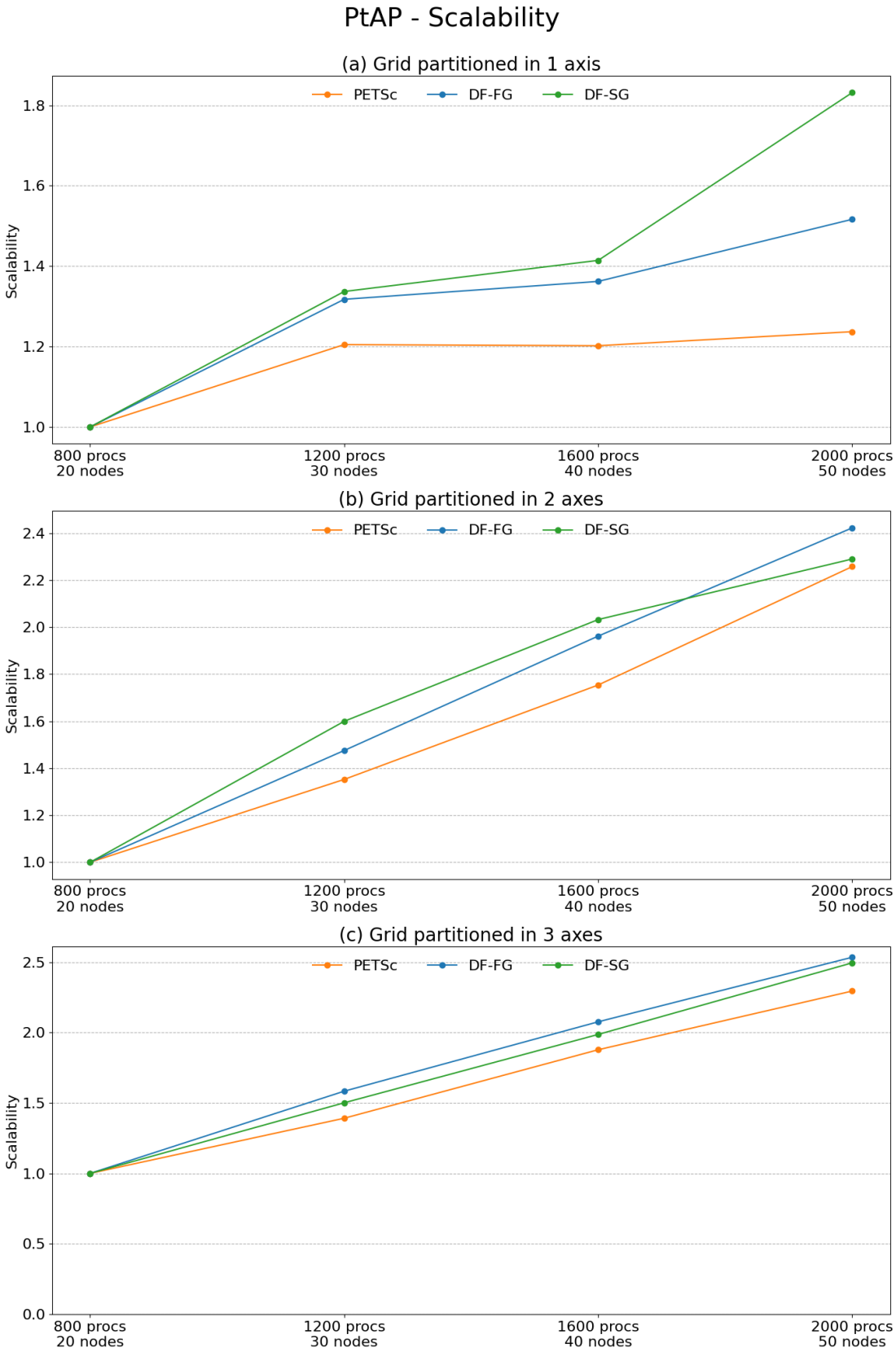


Figura 4.31: Gráficos de escalabilidade da operação PtAP com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



ticionamento. As figuras 4.30 e 4.31 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Com particionamento em 1 eixo coordenado, as versões dataflow escalam melhor, inciando com *speed-up* de 0.9x em relação ao PETSc com 800 processos e terminando com *speed-up* 1.1x e 1.4x (DF-FG e DF-SG, respectivamente). Nesse caso, a versão com subgrafos foi significativamente melhor. Particionando em 2 eixos, os *speed-ups* flutuaram entre 0.6x e 0.8x. Com particionamento em 3 eixos, o desempenho ficou entre 0.3x e 0.4x em relação ao PETSc. Em todos os casos, o tempo de *setup* dos núcleos *dataflow* ainda foi pequeno em relação aos tempos totais.

4.3 Resultados - Casos de extrapolação

As Seções 4.2.1, 4.2.2 e 4.2.3 trouxeram os resultados dos três núcleos de álgebra linear selecionados considerando um nível de paralelismo prático com base no tamanho do dado de entrada. Essa seção traz os resultados dos mesmos núcleos executados com até 2000 processos MPI (50 nós computacionais) para as matrizes de ordem 1 milhão e 10 milhões. Nessa situação, a granularidade dos domínios fica bastante restrita e os *overheads* do *framework*, como o tempo de *setup* do grafo, tornam-se bastante significativos. O objetivo aqui é avaliar o quanto o desempenho do *framework* é afetado nessas situações para uma maior clareza dos limites de aplicabilidade da ferramenta. E, também, para mapear possíveis melhorias futuras.

4.3.1 Operação y = Ax

Matriz de ordem 1 milhão (bloco 3x3), até 50 nós

A Tabela 4.20 compara os tempos da operação MatVec entre o PETSc e as duas versões *dataflow* (a *ordered* e a *relaxed*) para até 2000 processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.32 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.33 e 4.34 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Com particionamento em 1 eixo coordenado, os núcleos *dataflow* escalaram melhor, partindo de 1.1x de *speed-up* em relação ao PETSc com 80 processos até *speed-ups* de 1.8 e 1.9 (DF-Ord e DF-Rel, respectivamente). Nesse caso, as implementações *dataflow* escalaram muito melhor. Com particionamento em 2 eixos, o quadro geral foi parecido, exceto que o ganho final foi de 1.4x e 1.5x para os mesmo núcleos *dataflow*. Com particionamento em 3 eixos, os tempos foram bem próximos em quase todas as rodadas, exceto com 800 e com 1200 processo. Nesses casos o PETSc foi

Tabela 4.20: Tempos e Speed-up da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
80	2	0.0060	0.0055	0.0055	1.1	1.1
160	4	0.0030	0.0028	0.0028	1.1	1.1
240	6	0.0020	0.0019	0.0019	1.1	1.1
320	8	0.0015	0.0014	0.0014	1.1	1.1
400	10	0.0013	0.0012	0.0012	1.1	1.1
800	20	0.0012	0.0006	0.0007	1.9	1.8
1200	30	0.0005	0.0004	0.0005	1.3	1.2
1600	40	0.0004	0.0003	0.0003	1.3	1.3
2000	50	0.0005	0.0003	0.0003	1.8	1.9

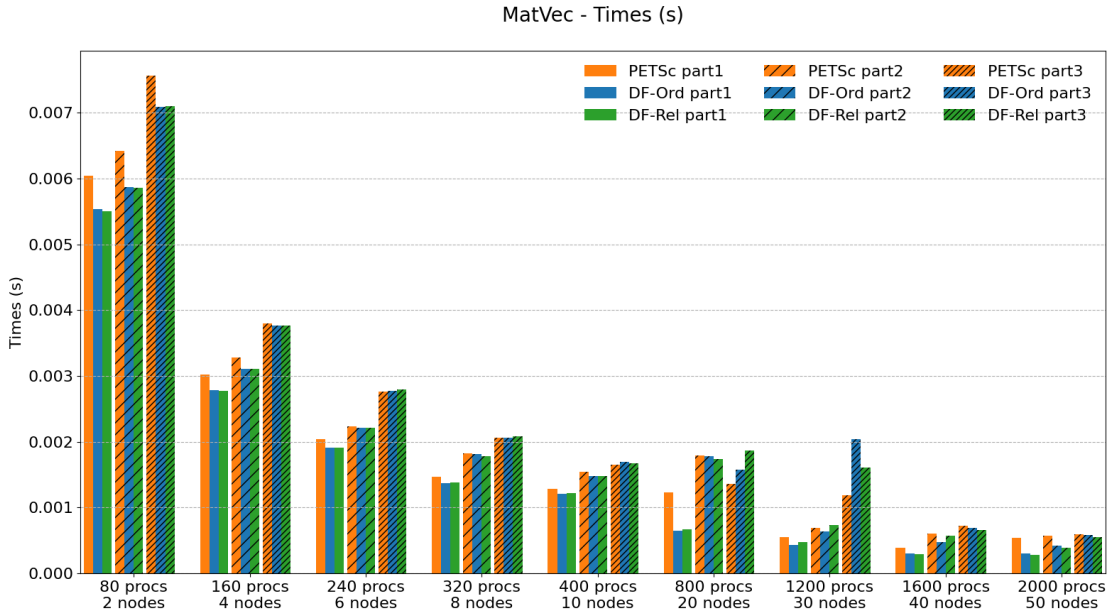
(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
80	2	0.0064	0.0059	0.0059	1.1	1.1
160	4	0.0033	0.0031	0.0031	1.1	1.1
240	6	0.0022	0.0022	0.0022	1.0	1.0
320	8	0.0018	0.0018	0.0018	1.0	1.0
400	10	0.0015	0.0015	0.0015	1.0	1.0
800	20	0.0018	0.0018	0.0017	1.0	1.0
1200	30	0.0007	0.0006	0.0007	1.1	0.9
1600	40	0.0006	0.0005	0.0006	1.3	1.1
2000	50	0.0006	0.0004	0.0004	1.4	1.5

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
80	2	0.0076	0.0071	0.0071	1.1	1.1
160	4	0.0038	0.0038	0.0038	1.0	1.0
240	6	0.0028	0.0028	0.0028	1.0	1.0
320	8	0.0021	0.0021	0.0021	1.0	1.0
400	10	0.0016	0.0017	0.0017	1.0	1.0
800	20	0.0014	0.0016	0.0019	0.9	0.7
1200	30	0.0012	0.0020	0.0016	0.6	0.7
1600	40	0.0007	0.0007	0.0007	1.0	1.1
2000	50	0.0006	0.0006	0.0006	1.0	1.1

Figura 4.32: Gráficos de tempos da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação



melhor, mas os núcleos *dataflow* foram competitivos em geral, e até ligeiramente melhores em algumas rodadas.

Figura 4.33: Gráficos de speed-up da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

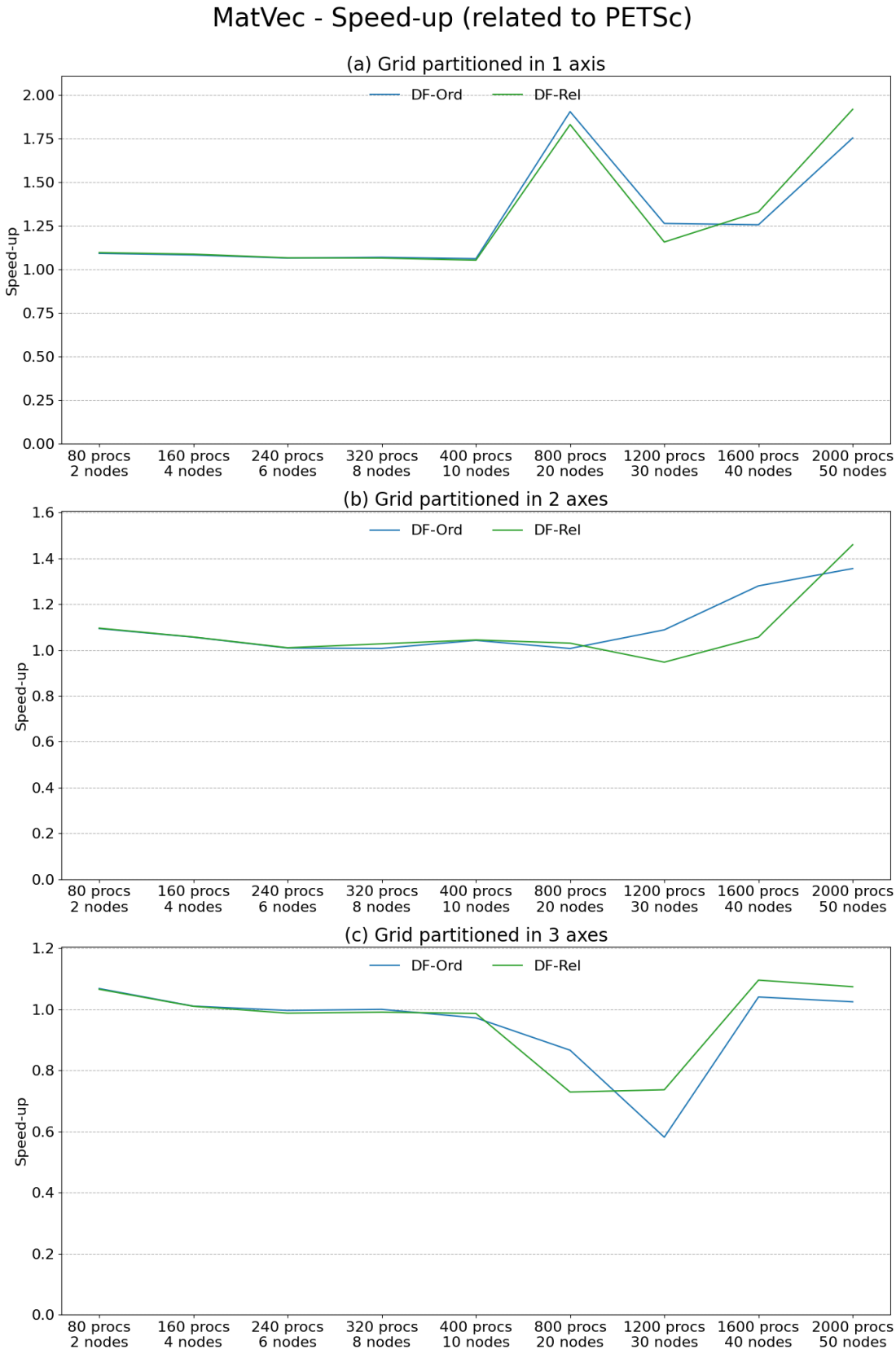
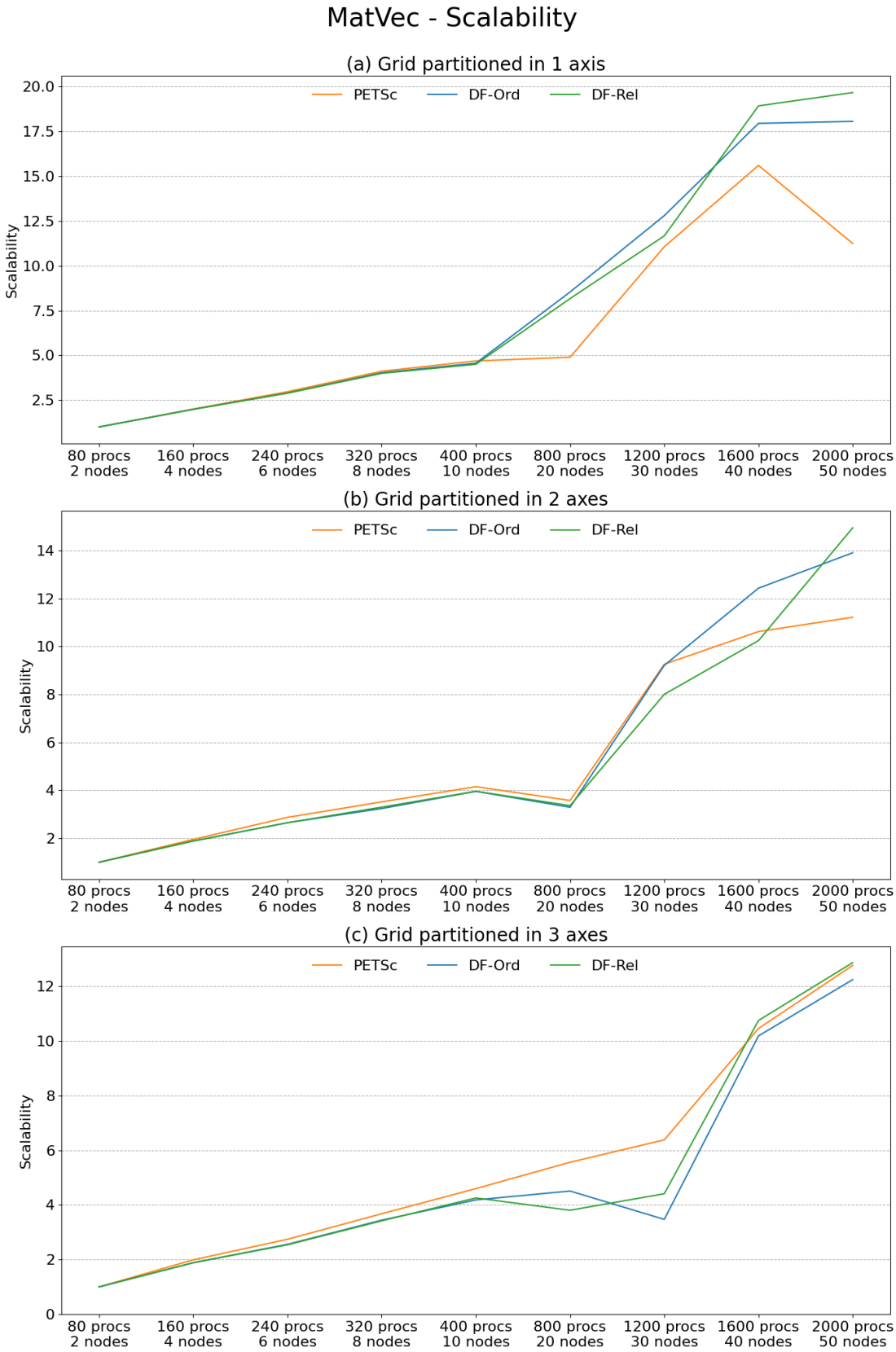


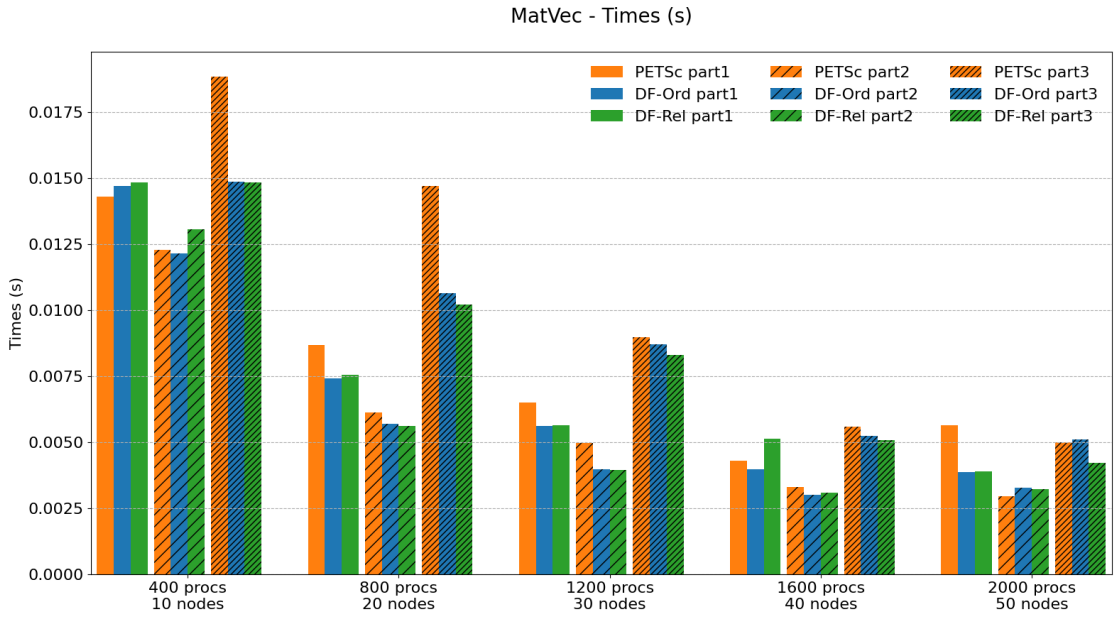
Figura 4.34: Gráficos de escalabilidade da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação



Matriz de ordem 10 milhões (bloco 3x3), até 50 nós

A Tabela 4.21 compara os tempos da operação MatVec entre o PETSc e as duas versões *dataflow* (a *ordered* e a *relaxed*) para até 2000 processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.35 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.36 e 4.37 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Figura 4.35: Gráficos de tempos da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação



Para esse tamanho de matriz, os núcleos *dataflow* foram melhores com particionamento em 1 eixo, atingindo *speed-ups* de 1.5x e 1.4x (DF-Ord e DF-Rel, respectivamente) em relação ao PETSc com 2000 processos. Com particionamento em 2 eixos, os núcleos *dataflow* foram competitivos, com flutuações nas rodadas. Com particionamento em 3 eixos, DF-Ord e DF-Rel foram iguais ou superiores em todos os casos, mas iniciam com *speed-up* 1.3x e terminaram em 1.0x e 1.2x com 2000 processes, respectivamente.

Tabela 4.21: Tempos e Speed-up da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
400	10	0.0143	0.0147	0.0148	1.0	1.0
800	20	0.0087	0.0074	0.0075	1.2	1.2
1200	30	0.0065	0.0056	0.0056	1.2	1.2
1600	40	0.0043	0.0040	0.0051	1.1	0.8
2000	50	0.0056	0.0039	0.0039	1.5	1.4

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
400	10	0.0123	0.0122	0.0131	1.0	0.9
800	20	0.0061	0.0057	0.0056	1.1	1.1
1200	30	0.0050	0.0040	0.0040	1.3	1.3
1600	40	0.0033	0.0030	0.0031	1.1	1.1
2000	50	0.0030	0.0033	0.0032	0.9	0.9

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
400	10	0.0188	0.0149	0.0148	1.3	1.3
800	20	0.0147	0.0107	0.0102	1.4	1.4
1200	30	0.0090	0.0087	0.0083	1.0	1.1
1600	40	0.0056	0.0052	0.0051	1.1	1.1
2000	50	0.0050	0.0051	0.0042	1.0	1.2

Figura 4.36: Gráficos de speed-up da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

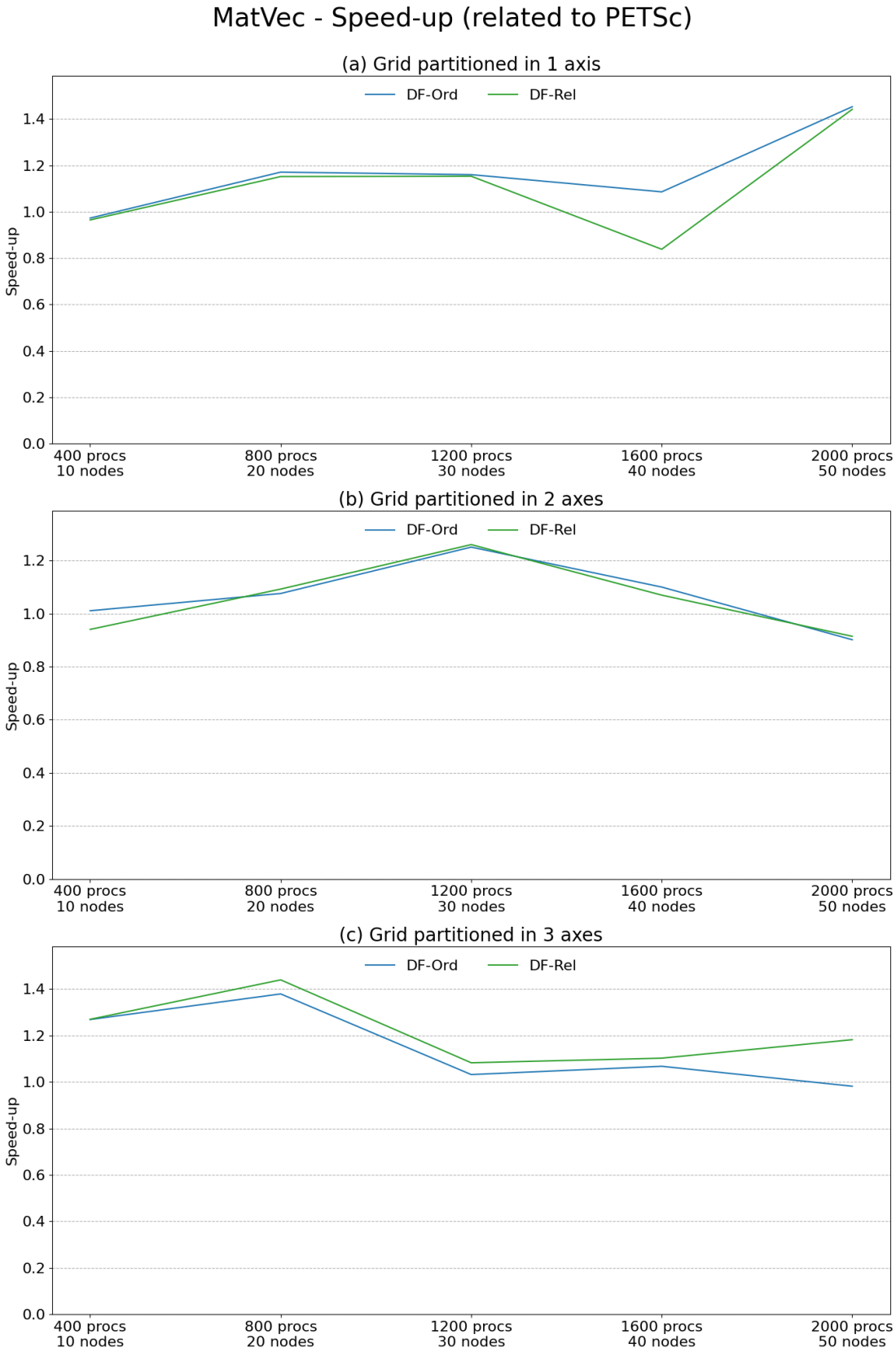
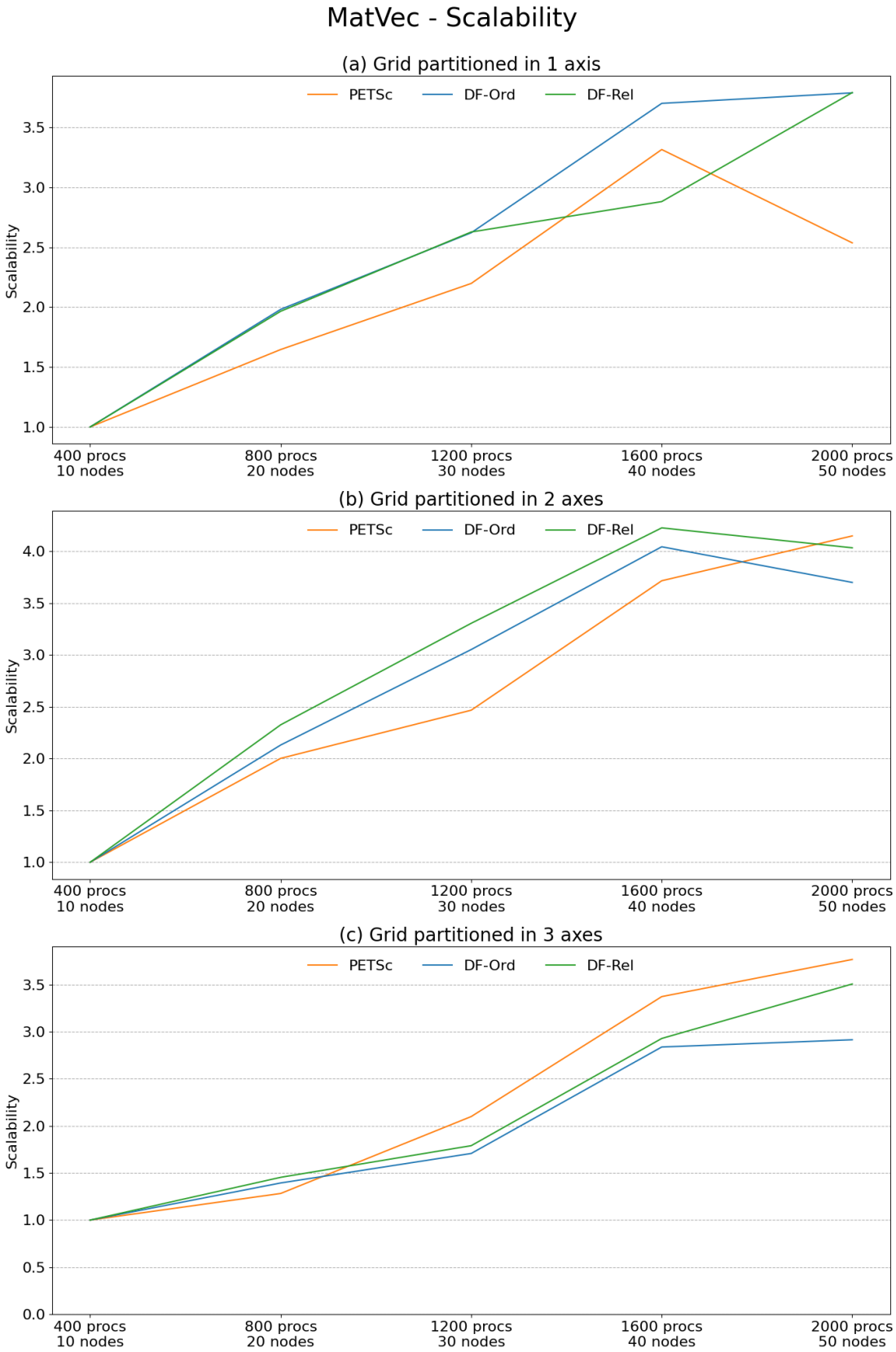


Figura 4.37: Gráficos de escalabilidade da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

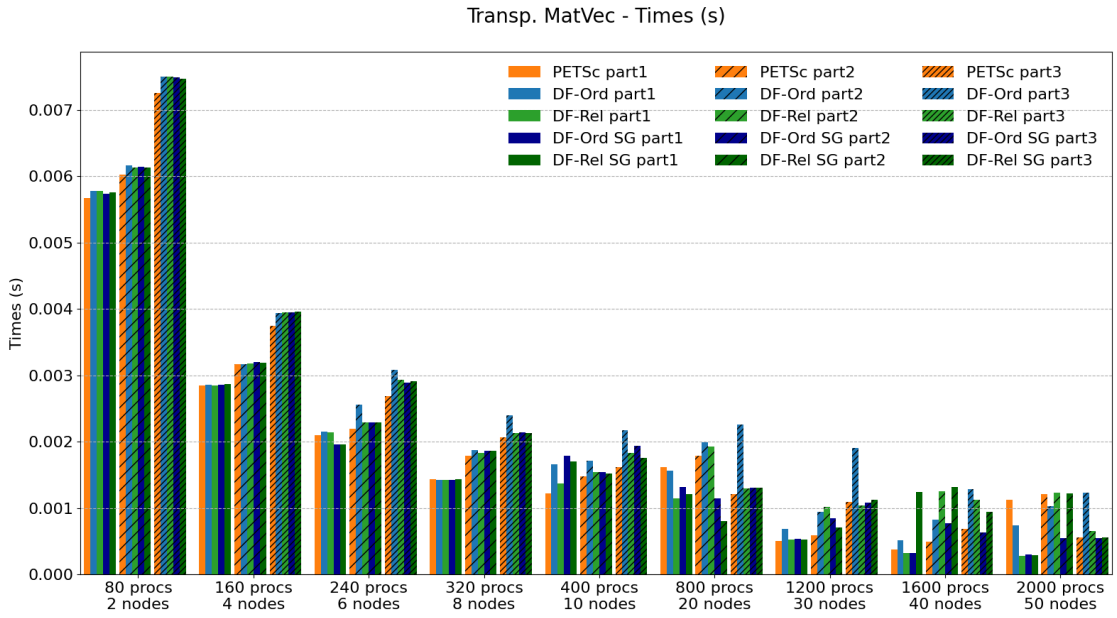


4.3.2 Operação $y = A^T x$

Matriz de ordem 1 milhão (bloco 3x3), até 50 nós

A Tabela 4.22 compara os tempos da operação Transp. MatVec entre o PETSc e as versões *dataflow* (*ordered*, *relaxed*, *ordered SG* e *relaxed SG*) para até 2000 processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.38 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.39 e 4.40 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Figura 4.38: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação



Para esse tamanho de matriz na operação Trasnp. MatVec, os resultados tiveram bastante variação entre si. Com particionamento em 1 eixo coordenado, os núcleos *dataflow* DF-Ord, DF-Rel, DF-Ord SG e DF-Rel SG atingiram *speed-ups* em relação ao PETSc de 1.5x, 4.1x, 3.8x e 4.0x, respectivamente, com 2000 processos MPI (50 nós). No entanto ocorreram variações em algumas rodadas intermediárias,e em alguns casos o PETSc foi mais rápido. Com particionamento em 2 eixos, ocorreram flutuações, mas os tempos das versões *dataflow* foram competitivos. Com particionamento em 3 eixos, foram competitivas com o PETSc, embora ligeiramente inferiores no geral. Já as versões sem subgrafos se saíram pior, principalmente a DF-Ord, que terminou 0.4x mais lento. O menor tempo global foi obtido pelas versões DF-Rel e DF-Rel SG com 2000 processos e particionamento em 1 eixo.

Tabela 4.22: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

Tabela 4.23: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
80	2	0.0057	0.0058	0.0058	0.0057	0.0058	1.0	1.0	1.0	1.0
160	4	0.0028	0.0029	0.0029	0.0029	0.0029	1.0	1.0	1.0	1.0
240	6	0.0021	0.0021	0.0021	0.0020	0.0020	1.0	1.0	1.1	1.1
320	8	0.0014	0.0014	0.0014	0.0014	0.0014	1.0	1.0	1.0	1.0
400	10	0.0012	0.0017	0.0014	0.0018	0.0017	0.7	0.9	0.7	0.7
800	20	0.0016	0.0016	0.0011	0.0013	0.0012	1.0	1.4	1.2	1.3
1200	30	0.0005	0.0007	0.0005	0.0005	0.0005	0.7	0.9	0.9	1.0
1600	40	0.0004	0.0005	0.0003	0.0003	0.0012	0.7	1.2	1.2	0.3
2000	50	0.0011	0.0007	0.0003	0.0003	0.0003	1.5	4.1	3.8	4.0

Tabela 4.24: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
80	2	0.0060	0.0062	0.0061	0.0061	0.0061	1.0	1.0	1.0	1.0
160	4	0.0032	0.0032	0.0032	0.0032	0.0032	1.0	1.0	1.0	1.0
240	6	0.0022	0.0026	0.0023	0.0023	0.0023	0.9	1.0	1.0	1.0
320	8	0.0018	0.0019	0.0018	0.0019	0.0019	1.0	1.0	1.0	1.0
400	10	0.0015	0.0017	0.0015	0.0015	0.0015	0.9	1.0	1.0	1.0
800	20	0.0018	0.0020	0.0019	0.0011	0.0008	0.9	0.9	1.6	2.2
1200	30	0.0006	0.0009	0.0010	0.0008	0.0007	0.6	0.6	0.7	0.8
1600	40	0.0005	0.0008	0.0012	0.0008	0.0013	0.6	0.4	0.6	0.4
2000	50	0.0012	0.0010	0.0012	0.0005	0.0012	1.2	1.0	2.2	1.0

Tabela 4.25: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
80	2	0.0073	0.0075	0.0075	0.0075	0.0075	1.0	1.0	1.0	1.0
160	4	0.0037	0.0039	0.0039	0.0039	0.0040	0.9	0.9	0.9	0.9
240	6	0.0027	0.0031	0.0029	0.0029	0.0029	0.9	0.9	0.9	0.9
320	8	0.0021	0.0024	0.0021	0.0021	0.0021	0.9	1.0	1.0	1.0
400	10	0.0016	0.0022	0.0018	0.0019	0.0018	0.7	0.9	0.8	0.9
800	20	0.0012	0.0023	0.0013	0.0013	0.0013	0.5	0.9	0.9	0.9
1200	30	0.0011	0.0019	0.0010	0.0011	0.0011	0.6	1.1	1.0	1.0
1600	40	0.0007	0.0013	0.0011	0.0006	0.0009	0.5	0.6	1.1	0.7
2000	50	0.0006	0.0012	0.0007	0.0005	0.0006	0.4	0.8	1.0	1.0

Figura 4.39: Gráficos de speed-up da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

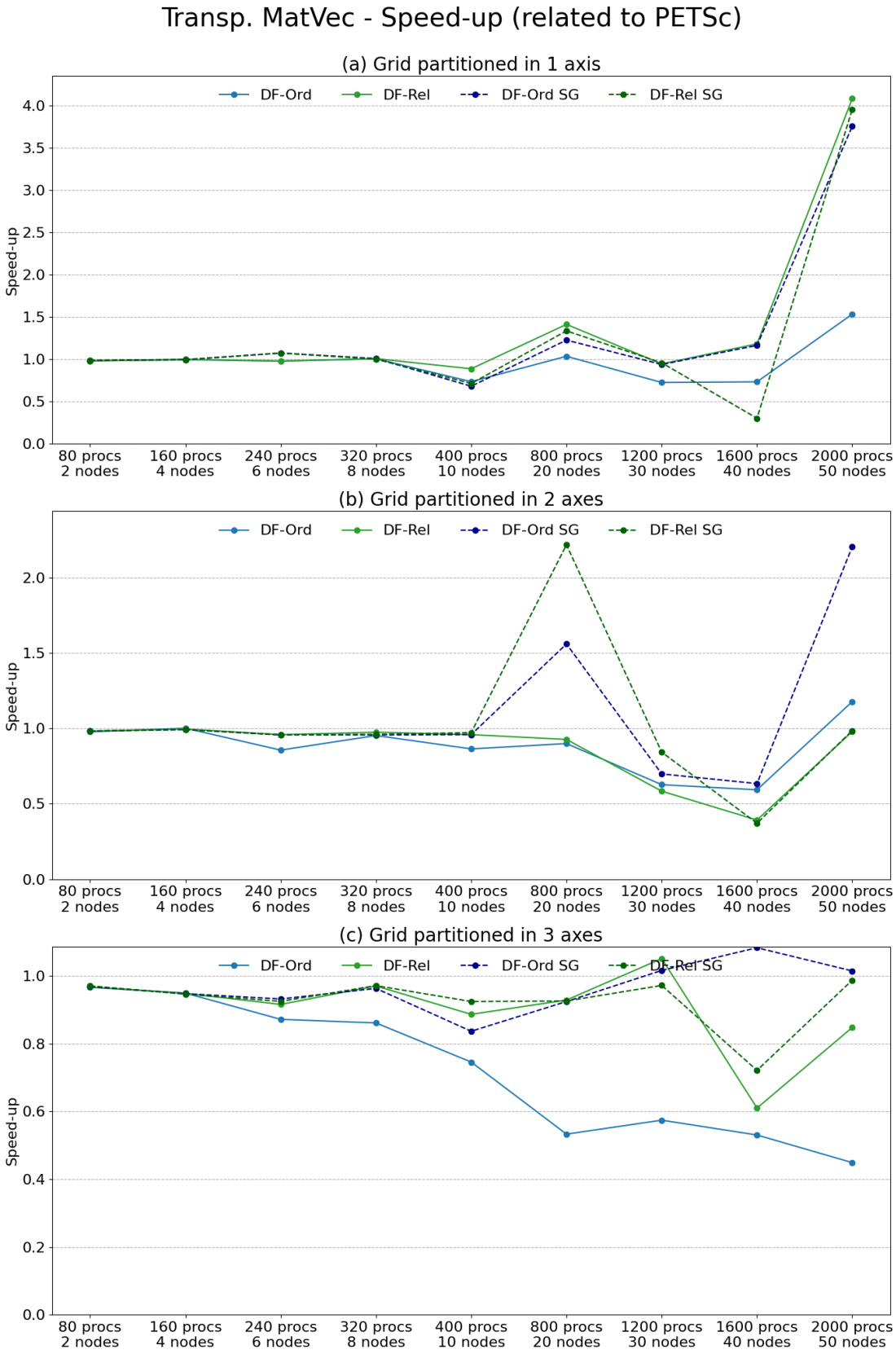
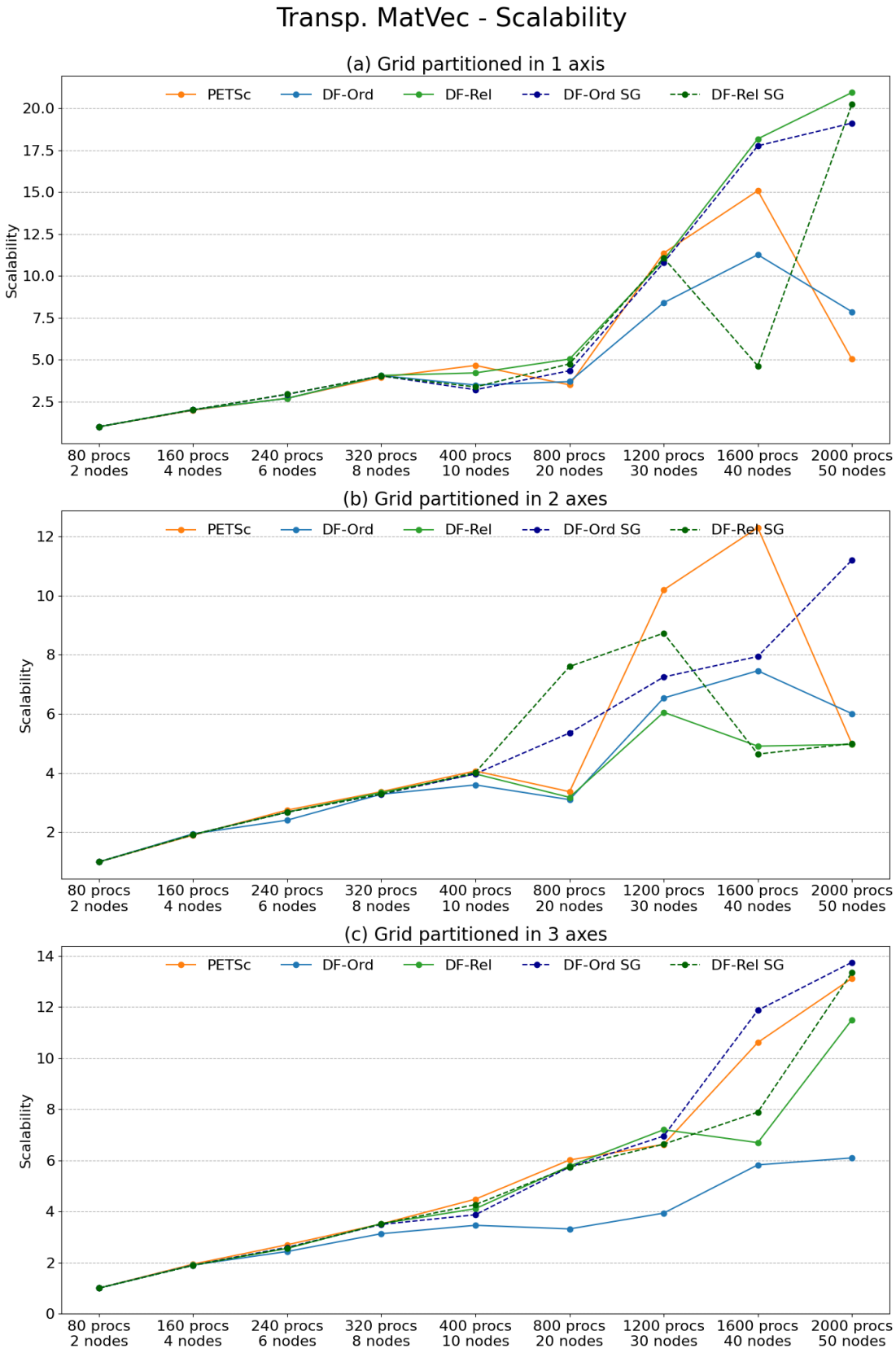


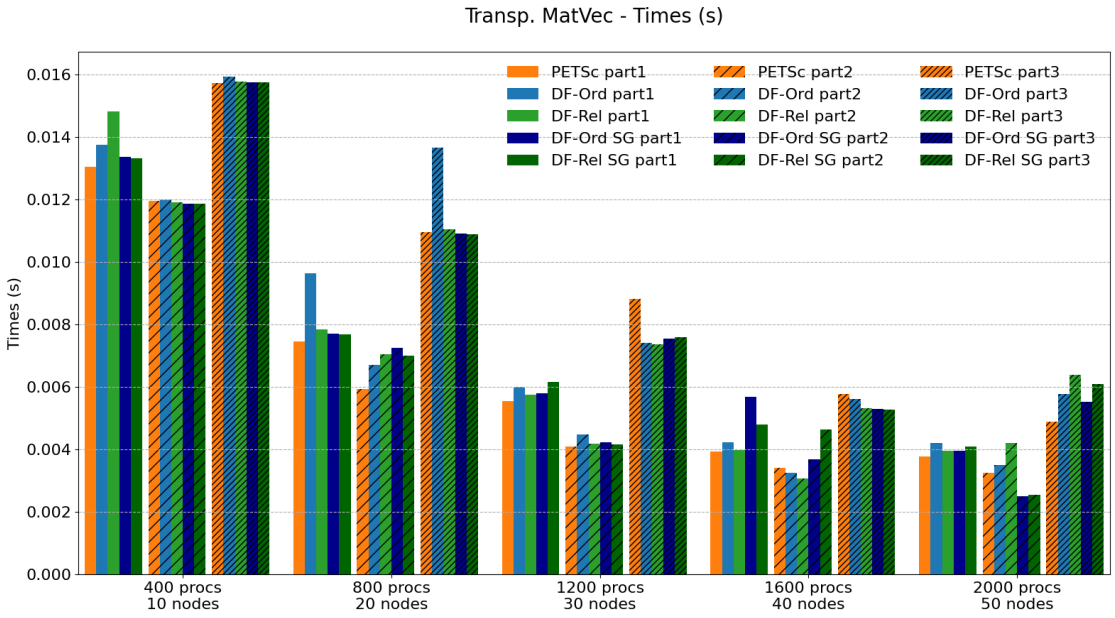
Figura 4.40: Gráficos de escalabilidade da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação



Matriz de ordem 10 milhões (bloco 3x3), até 50 nós

A Tabela 4.26 compara os tempos da operação Transp. MatVec entre o PETSc e as versões *dataflow* (*ordered*, *relaxed*, *ordered SG* e *relaxed SG*) para até 2000 processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.41 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. As figuras 4.42 e 4.43 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Figura 4.41: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação



Com esse tamanho de matriz, ocorreram flutuações em rodadas com as três estratégias de particionamento. De um modo geral, as versões *dataflow* foram competitivas com o PETSc, embora um pouco piores na maioria dos casos. O melhor tempo dentre todos os núcleos foi obtido com DF-Ord SG e DF-Rel SG rodando com 2000 processos e particionamento em 2 eixos.

Tabela 4.26: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

Tabela 4.27: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
400	10	0.0130	0.0138	0.0148	0.0134	0.0133	0.9	0.9	1.0	1.0
800	20	0.0074	0.0096	0.0078	0.0077	0.0077	0.8	1.0	1.0	1.0
1200	30	0.0056	0.0060	0.0057	0.0058	0.0062	0.9	1.0	1.0	0.9
1600	40	0.0039	0.0042	0.0040	0.0057	0.0048	0.9	1.0	0.7	0.8
2000	50	0.0038	0.0042	0.0040	0.0039	0.0041	0.9	1.0	1.0	0.9

Tabela 4.28: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
400	10	0.0120	0.0120	0.0119	0.0119	0.0119	1.0	1.0	1.0	1.0
800	20	0.0059	0.0067	0.0071	0.0073	0.0070	0.9	0.8	0.8	0.8
1200	30	0.0041	0.0045	0.0042	0.0042	0.0042	0.9	1.0	1.0	1.0
1600	40	0.0034	0.0032	0.0031	0.0037	0.0046	1.1	1.1	0.9	0.7
2000	50	0.0033	0.0035	0.0042	0.0025	0.0025	0.9	0.8	1.3	1.3

Tabela 4.29: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
400	10	0.0157	0.0159	0.0158	0.0157	0.0157	1.0	1.0	1.0	1.0
800	20	0.0110	0.0137	0.0110	0.0109	0.0109	0.8	1.0	1.0	1.0
1200	30	0.0088	0.0074	0.0074	0.0076	0.0076	1.2	1.2	1.2	1.2
1600	40	0.0058	0.0056	0.0053	0.0053	0.0053	1.0	1.1	1.1	1.1
2000	50	0.0049	0.0058	0.0064	0.0055	0.0061	0.8	0.8	0.9	0.8

Figura 4.42: Gráficos de speed-up da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

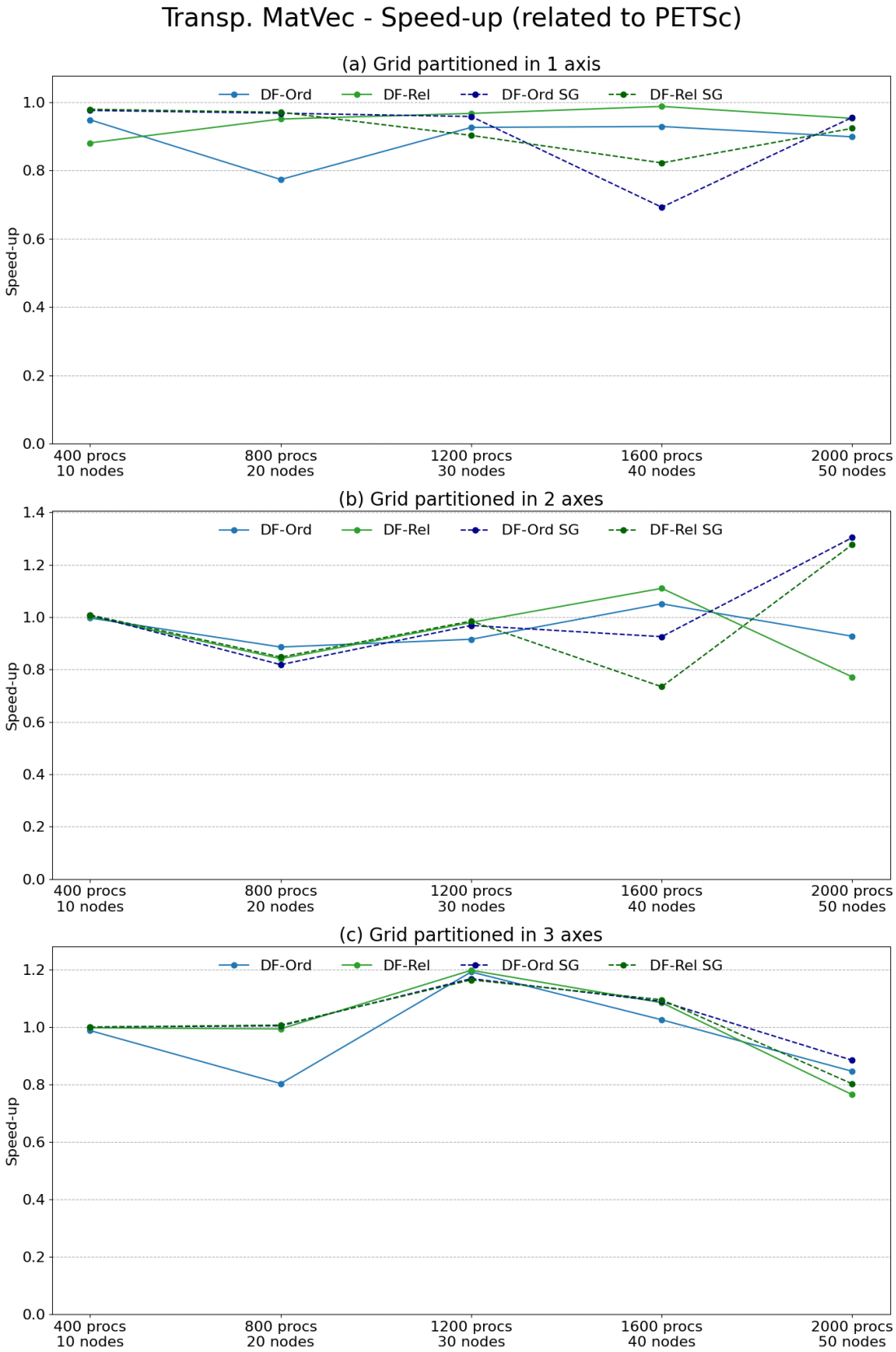
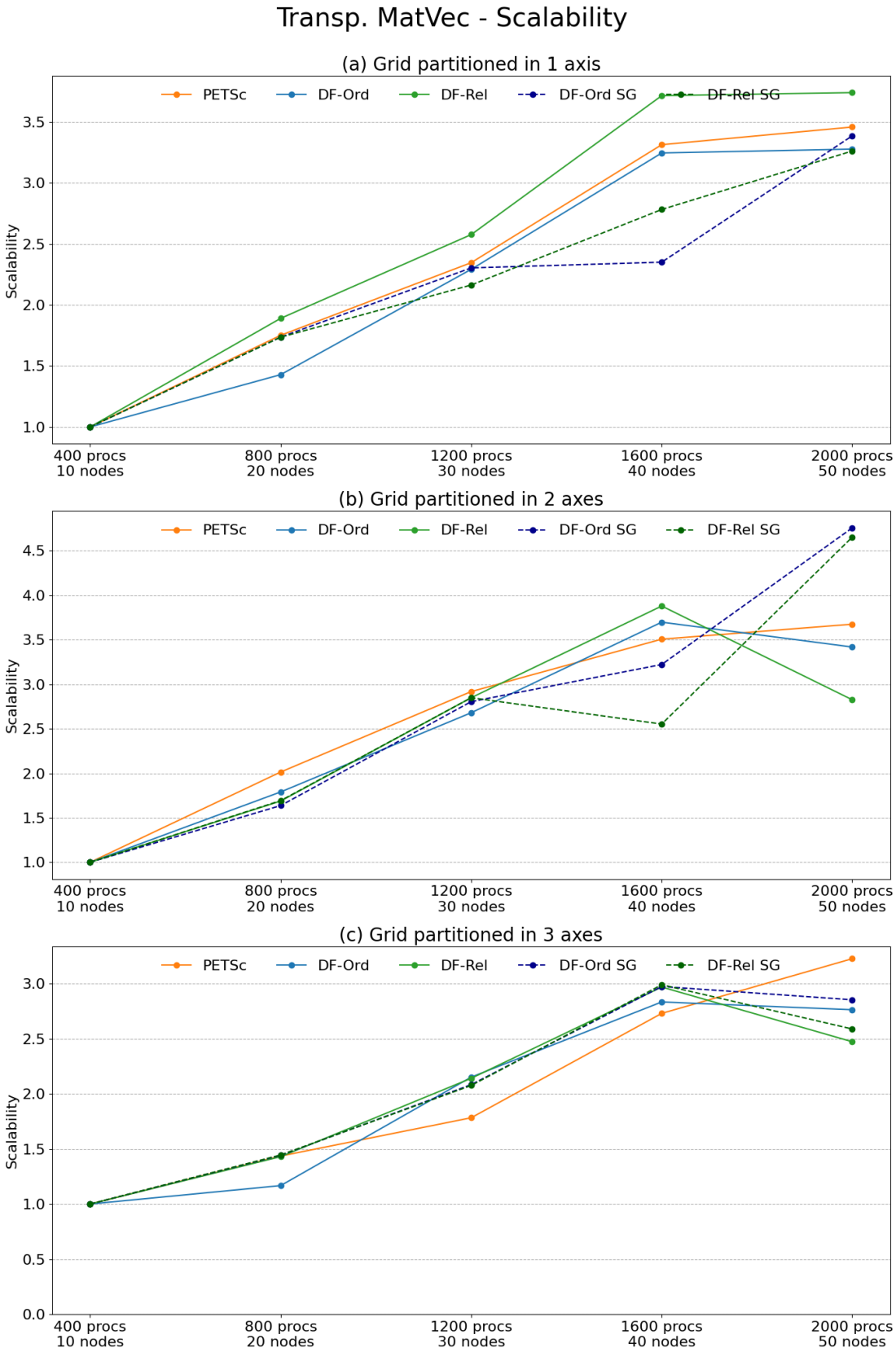


Figura 4.43: Gráficos de escalabilidade da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

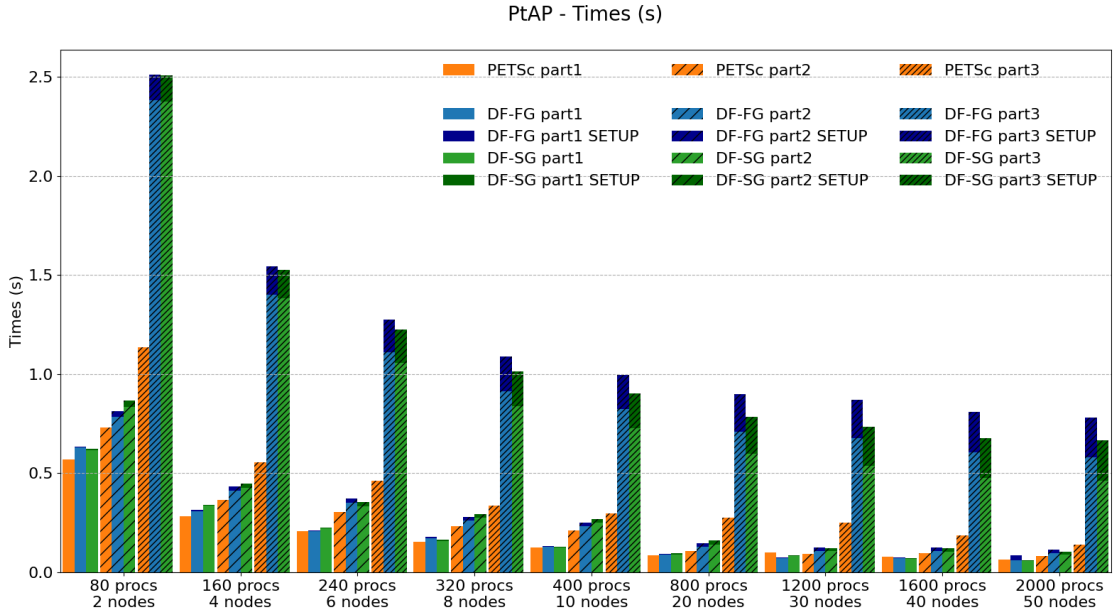


4.3.3 Operação $C = P^T AP$

Matriz de ordem 1 milhão (bloco 3x3), até 50 nós

A Tabela 4.30 compara os tempos da operação PtAP entre o PETSc e as duas versões *dataflow* (a *full graph* e a *sub-graphs*) para até 2000 processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.44 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. Os tempos de *setup* de cada núcleo *dataflow* estão destacados em cor mais escura na respectiva barra. As figuras 4.45 e 4.46 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Figura 4.44: Gráficos de tempos da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação



Com particionamento em 1 e 2 eixos, os núcleos *dataflow* apresentam boa escalabilidade. O DF-SG fica até superior ao PETSc com 2000 processos e particionamento em um eixo. Com particionamento em 2 eixos, as versões *dataflow* ficam piores, porém ainda competitivas (speed-ups entre 0.7x e 0.8x com 2000 processos). O particionamento em 3 eixos degrada bastante a performance das implementações *dataflow*. O DF-SG, por exemplo, com 2000 processos executa em 0.06, 0.10 e 0.66 segundos em cada estratégia de particionamento. Nota-se uma degradação de 2x do primeiro caso para o segundo e de 10x do primeiro para o terceiro. O hiper particionamento de uma matriz desse tamanho afeta duplamente os núcleos *dataflow*: aumenta muito o tamanho do grafo devido à natureza da operação (multiplicação tripla de matrizes) e, ao mesmo tempo, reduz a granularidade dos domínios (os domínios de matrizes são mais esparsos).

Tabela 4.30: Tempos e Speed-up da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
80	2	0.5695	0.6354	0.6217	0.9	0.9
160	4	0.2838	0.3142	0.3416	0.9	0.8
240	6	0.2078	0.2122	0.2277	1.0	0.9
320	8	0.1525	0.1786	0.1643	0.9	0.9
400	10	0.1265	0.1338	0.1289	0.9	1.0
800	20	0.0875	0.0948	0.0986	0.9	0.9
1200	30	0.1007	0.0767	0.0868	1.3	1.2
1600	40	0.0784	0.0741	0.0734	1.1	1.1
2000	50	0.0655	0.0860	0.0599	0.8	1.1

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
80	2	0.7314	0.8144	0.8656	0.9	0.8
160	4	0.3666	0.4326	0.4471	0.8	0.8
240	6	0.3062	0.3741	0.3533	0.8	0.9
320	8	0.2317	0.2798	0.2937	0.8	0.8
400	10	0.2116	0.2523	0.2694	0.8	0.8
800	20	0.1075	0.1473	0.1603	0.7	0.7
1200	30	0.0939	0.1254	0.1224	0.7	0.8
1600	40	0.0965	0.1258	0.1209	0.8	0.8
2000	50	0.0828	0.1147	0.1059	0.7	0.8

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
80	2	1.1372	2.5113	2.5061	0.5	0.5
160	4	0.5547	1.5423	1.5268	0.4	0.4
240	6	0.4624	1.2765	1.2262	0.4	0.4
320	8	0.3358	1.0899	1.0136	0.3	0.3
400	10	0.2984	0.9968	0.9033	0.3	0.3
800	20	0.2772	0.8998	0.7854	0.3	0.4
1200	30	0.2500	0.8714	0.7351	0.3	0.3
1600	40	0.1864	0.8095	0.6785	0.2	0.3
2000	50	0.1391	0.7822	0.6671	0.2	0.2

Figura 4.45: Gráficos de speed-up da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

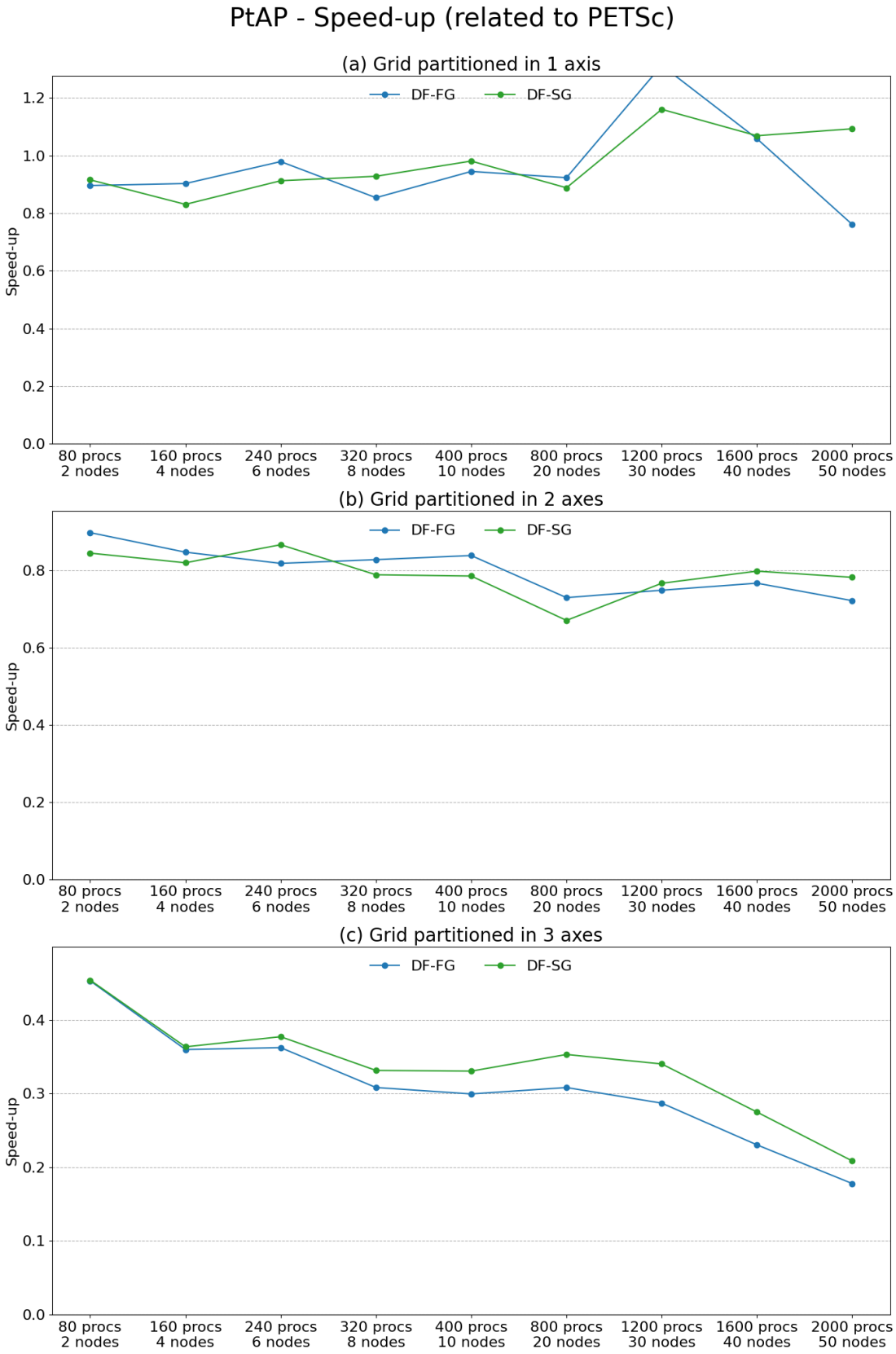
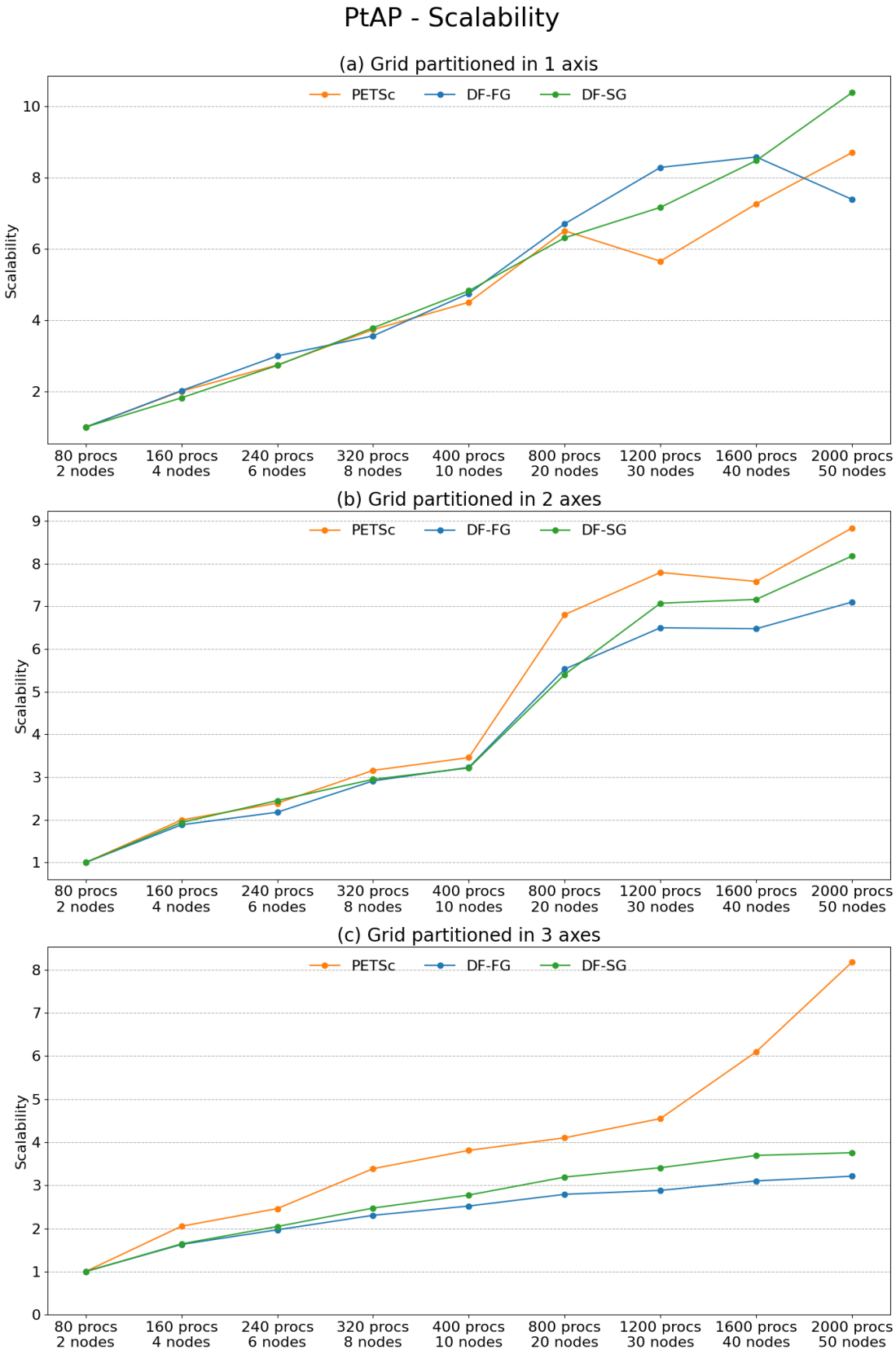


Figura 4.46: Gráficos de escalabilidade da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

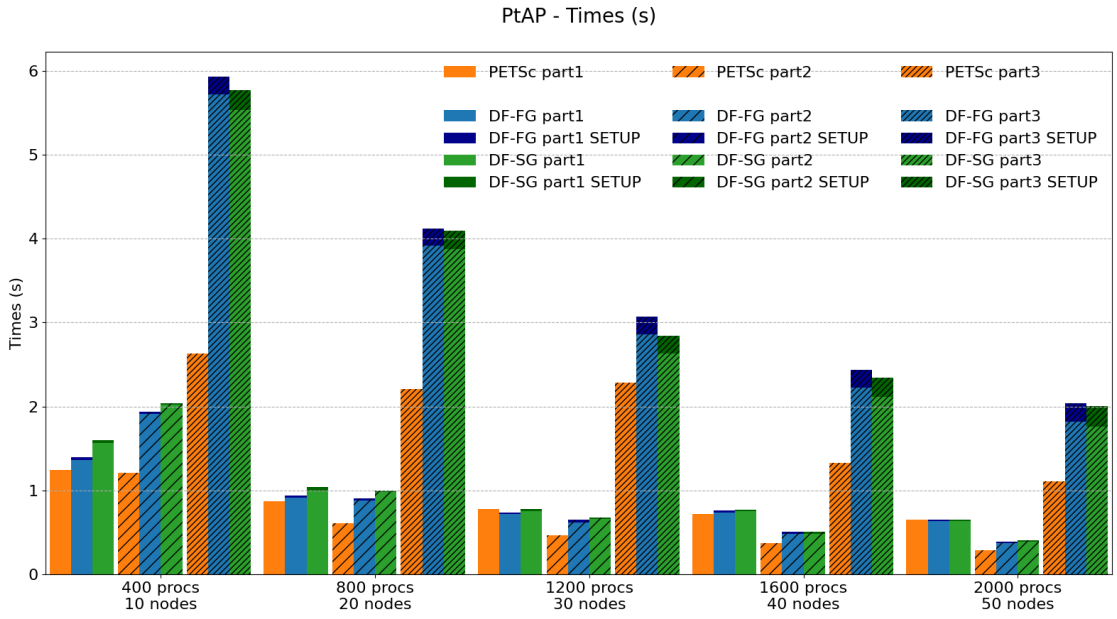


Nota-se, também, que o tempo de *setup* das implementações *dataflow* com muitos nós ficou bastante elevado, inclusive superando o tempo do PtAP inteiro do PETSc em muitos casos.

Matriz de ordem 10 milhões (bloco 3x3), até 50 nós

A Tabela 4.31 compara os tempos da operação PtAP entre o PETSc e as duas versões *dataflow* (a *full graph* e a *sub-graphs*) para até 2000 processos MPI e diferentes tipos de particionamento do *grid*. A Figura 4.47 exibe os tempos de cada implementação agrupados por número de processos e por estratégia de particionamento. Os tempos de *setup* de cada núcleo *dataflow* estão destacados em cor mais escura na respectiva barra. As figuras 4.48 e 4.49 trazem o *speed-up* em relação ao PETSc e a escalabilidade forte de cada núcleo, respectivamente.

Figura 4.47: Gráficos de tempos da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação



Com particionamento em 1 eixo, os núcleos *dataflow* começaram ligeiramente inferiores ao PETSc com 400 processos e terminaram com a mesma performance, mostrando melhor escalabilidade nesse caso. Com particionamento em 2 eixos, os núcleos *dataflow* partiram de 0.6x com 400 processos para 0.7x com 2000 processos, escalando ligeiramente melhor, mas terminado 30% mais lento. O particionamento em 3 eixos trouxe degradação de desempenho aos núcleos *dataflow*, mas em uma escala menor que no caso anterior (provavelmente devido a maior carga de trabalho da matriz). Aqui, o DF-SG se saiu um pouco melhor que a DF-FG em quase todas as rodadas. Os tempos de *setup* foram bem significativos, mas nada tão drástico quanto no caso anterior.

Tabela 4.31: Tempos e Speed-up da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
400	10	1.2479	1.3977	1.6008	0.9	0.8
800	20	0.8749	0.9409	1.0414	0.9	0.8
1200	30	0.7779	0.7397	0.7764	1.1	1.0
1600	40	0.7158	0.7596	0.7689	0.9	0.9
2000	50	0.6541	0.6522	0.6501	1.0	1.0

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
400	10	1.2104	1.9388	2.0426	0.6	0.6
800	20	0.6135	0.9037	1.0002	0.7	0.6
1200	30	0.4624	0.6484	0.6810	0.7	0.7
1600	40	0.3731	0.5110	0.5057	0.7	0.7
2000	50	0.2900	0.3913	0.4078	0.7	0.7

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
400	10	2.6272	5.9301	5.7649	0.4	0.5
800	20	2.2105	4.1218	4.0900	0.5	0.5
1200	30	2.2860	3.0710	2.8425	0.7	0.8
1600	40	1.3254	2.4345	2.3469	0.5	0.6
2000	50	1.1050	2.0359	2.0020	0.5	0.6

Figura 4.48: Gráficos de speed-up da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

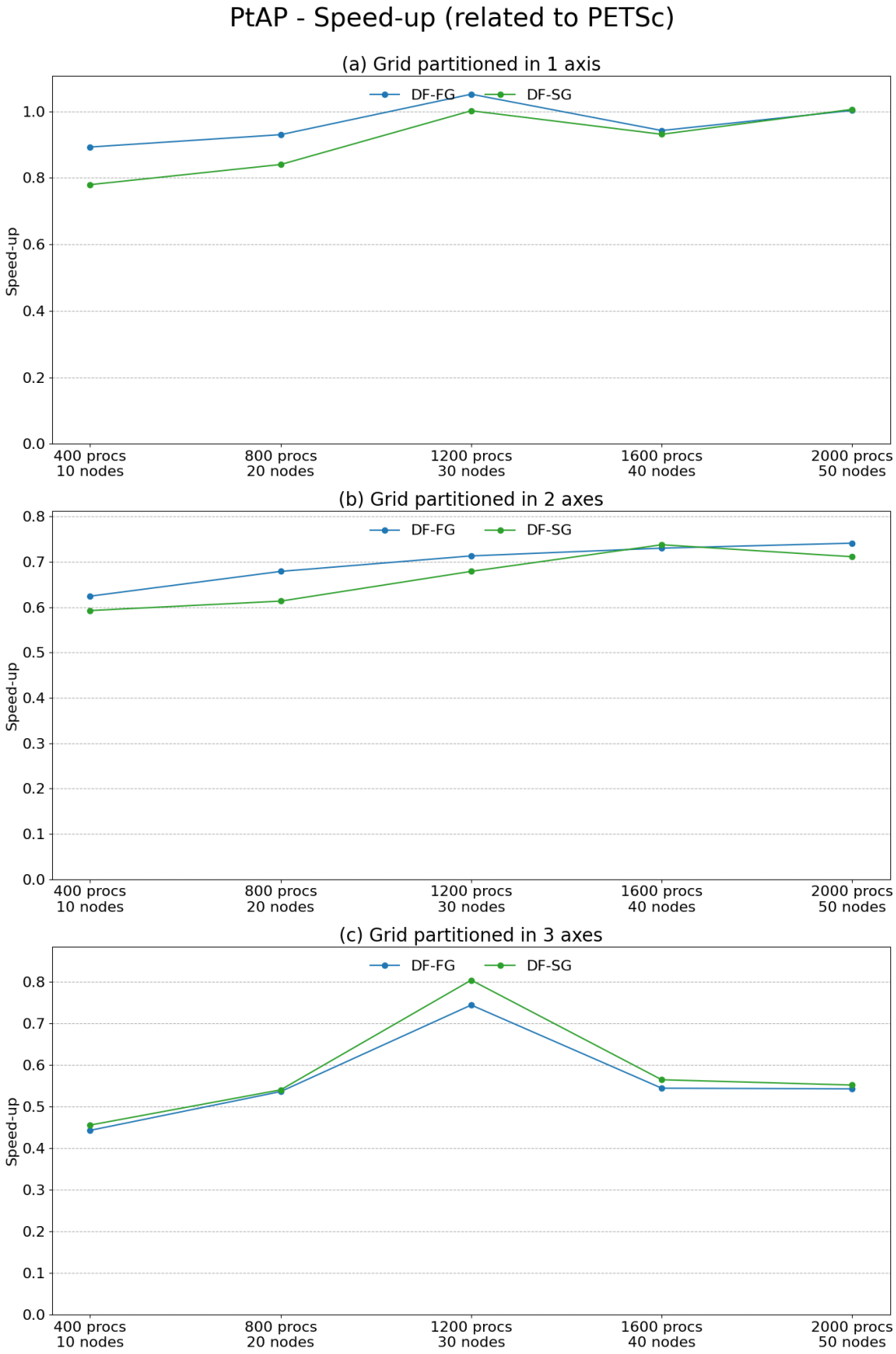
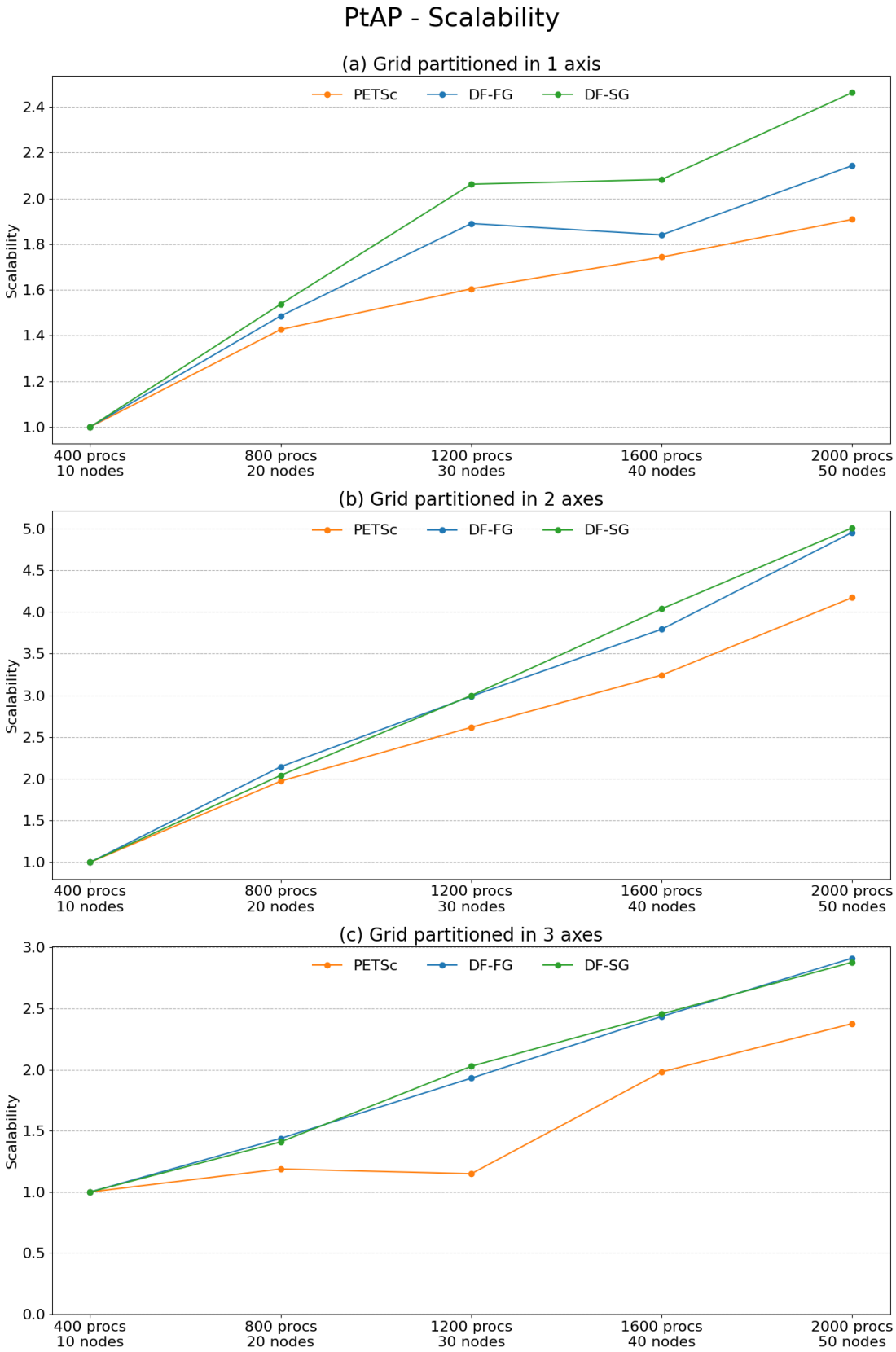


Figura 4.49: Gráficos de escalabilidade da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação



Capítulo 5

Conclusão e Trabalhos Futuros

Esse capítulo começa com uma análise dos experimentos numéricos do Capítulo 4. A seção seguinte discorre sobre os trabalhos futuros, apresentando uma lista de possíveis evoluções do *framework*. A última seção traz as considerações finais desse trabalho.

5.1 Análise dos Experimentos Numéricos

No Capítulo 4, foram feitos testes extensivos envolvendo três diferentes núcleos de álgebra linear esparsa. Os testes variaram em tamanho dos dados de entrada, nível de paralelismo e esquema de particionamento do *grid* gerador das matrizes.

5.1.1 Análise da operação y = Ax

Os resultados dos casos de interesse dessa operação estão descritos na Seção 4.2.1, e os resultados dos casos de extrapolação na Seção 4.3.1.

As Figuras 4.4 e 4.7 e 4.10 mostram que as duas versões *dataflow* para MatVec (DF-Ord e DF-Rel) foram melhores que o PETSc (na maioria das vezes por pequena diferença) em todos os tamanhos de matriz e em todas as estratégias de particionamento para os casos considerados de interesse prático.

Para as rodadas consideradas de extrapolação, onde casos com matrizes pequenas e médias foram executados com até 2000 processos MPI, as Figuras 4.32 e 4.35 mostram que as duas versões *dataflow* também tiveram desempenho superior ao do PETSc na maioria dos cenários, incluindo ganhos de até 1.9x em alguns casos (ver Tabela 4.20).

Esse resultado mostra que, ao menos para algoritmos paralelos simples como o MatVec, o *framework* é capaz de igualar ou até superar uma ferramenta HPC que é referência na área de álgebra linear computacional de larga escala, o que pode ser considerado um excelente resultado.

Contudo, algumas das limitações do *framework dataflow* atual são menos impactantes no MatVec. Por exemplo, o tamanho do Grafo de Execução em cara *rank MPI* é diretamente proporcional ao número de domínios locais da matriz (submatrizes na linha do *rank*). Ou seja, mesmo particionando o *grid* em 3 eixos coordenados, o número total de tarefas locais ainda é relativamente baixo. Outro fator é que o envio dos vetores x comprimidos (Seção 3.4.1) nas tarefas *GET* é feito em um único passo (*step*), o que reduz o possível *overhead* do *non-threaded runtime* do *framework* (limitação escrita na Seção 3.3.4).

5.1.2 Análise da operação $\mathbf{y = A^T x}$

Os resultados dos casos de interesse dessa operação estão descritos na Seção 4.2.2, e os resultados dos casos de extrapolação na Seção 4.3.2.

As Figuras 4.13 e 4.16 e 4.19 mostram que as quatro versões *dataflow* do núcleo Transp. MatVec (DF-Ord, DF-Rel, DF-Ord SG e DF-Rel SG) tiveram desempenho competitivo com o PETSc em todos os tamanhos de matriz e em todas as estratégias de particionamento para os casos considerados de interesse prático.

Para as rodadas consideradas de extrapolação, onde casos com matrizes pequenas e médias foram executados com até 2000 processos MPI, as Figuras 4.38 e 4.41 mostram algumas flutuações de tempo em desfavor das versões *dataflow* com política de execução padrão (*Full Graph*) em alguns casos, mas as implementações *dataflow* executando a política *Subgraphs* foram competitivas em geral. Nessas situações, tal política se mostrou efetiva (as políticas de execução foram discutidas na Seção 3.3.3).

Os núcleos *dataflow* de Transp. MatVec também conseguiram ser competitivos com o PETSc (principalmente executando a política de Subgrafos), o que também é um excelente resultado. No entanto, as mesmas ponderações feitas sobre o MatVec (sobre o tamanho do grafo e a simplicidade das comunicações) também valem aqui.

5.1.3 Análise da operação $\mathbf{C = P^T AP}$

Os resultados dos casos de interesse dessa operação estão descritos na Seção 4.2.3, e os resultados dos casos de extrapolação na Seção 4.3.3.

As Figuras 4.23 e 4.26 e 4.29 mostram um quadro mais complexo para as rodadas de interesse. Primeiramente, as duas versões *dataflow* (DF-FG e DF-SG, mesmo algoritmo, mas executando as políticas *Full Graph* e *Subgraphs*, respectivamente) tiveram desempenho competitivo com o PETSc usando particionamento em 1 eixo coordenado em todos os tamanhos de matriz. Com particionamento em 2 eixos (o esquema *default* do geomecBR), as versões *dataflow* perdem competitividade, ficando entre 0.6x e 0.8x mais lentos que o PETSc nas rodadas em geral.

Aumentando para 3 eixos, a situação se agrava e o desempenho comparado com o PETSc fica entre 0.4x e 0.5x nas rodadas em geral.

Para as rodadas consideradas de extrapolação, onde casos com matrizes pequenas e médias foram executados com até 2000 processos MPI, as Figuras 4.44 e 4.47 mostram que as duas versões *dataflow* tiveram um desempenho similar ao desempenho nos casos de interesse para particionamento em 1 eixo e 2 eixos nos tamanhos de matriz em geral. No entanto, usando particionamento em 3 eixos o impacto no desempenho é severo com matrizes pequenas (de ordem 1 milhão de blocos). Para matrizes de tamanho médio, os tempos ficaram entre 0.5x e 0.6x em relação ao PETSc (valores próximos aos obtidos nos casos de interesse).

Aqui cabem várias considerações. Primeiramente, sobre o bom desempenho das versões *dataflow* com particionamento em 1 eixo para todos os tamanhos de matriz (incluindo as rodadas de extrapolação), como mostra o item (a) da Figura 4.3, esse esquema forma uma esparsidade de domínios **tridiagonal**. A operação PtAP em matrizes com essa esparsidade de domínios ainda gera um número relativamente pequeno de tarefas. A Tabela 4.16 mostra que uma rodada com 2000 processos usando particionamento em 1 eixo gera, em média, 96 tarefas por *rank MPI*. Note que, aqui, todas as comunicações de submatrizes são feitas usando-se múltiplos *steps* nas tarefas tipo GET, o que, ao menos hipoteticamente, pode diminuir a eficiência do *non-thread runtime* do *frameowrk*. Mesmo assim, as versões *dataflow* conseguiram ser competitivas com o PETSc nesses casos. Porém, infelizmente, esse tipo de particionamento não é muito usado na prática por algumas razões. A principal delas é que o paralelismo é limitado pelo tamanho do *grid* ao longo de um único eixo coordenado. Por exemplo, usando esse esquema, um *grid* com 300x200x100 elementos pode ser dividido em, no máximo, 300 domínios (maior valor por eixo).

Sobre o particionamento em 2 eixos coordenados, que é o mais importante para o GeomecBR, como mostra o item (b) da Figura 4.3, a esparsidade de domínios típica desse esquema contém 9 diagonais. Aqui, a complexidade do PtAP (em termos de número de tarefas) já começa a criar dificuldades. A Tabela 4.16 mostra que usando 2000 processos com esse particionamento, o número médio de tarefas locais por *rank* MPI é 1284, ou seja, uma ordem de grandeza superior ao mesmo caso particionando-se em 1 eixo. Isso eleva o *overhead* de gerenciamento de tarefas, sobretudo em função do *non-threaded runtime* do *framework*, que, como visto na Seção 3.14, tende a perder eficiência no gerenciamento de comunicações quando há muitas tarefas na lista de execução. Outra hipótese sobre o que pode ter afetado o desempenho é quanto a heterogeneidade introduzida na granularidade dos domínios das matrizes com o particionamento em 2 eixos. Com particionamento em 1 eixo, todas as submatrizes de conexão são do tipo ilustrado no item (b) Figura 2.11, em que os domínios do *grid* compartilham uma face. O particionamento em 2 eixos

introduz, também, submatrizes de conexão como as do item (c) da mesma figura, ou seja, em que os domínios compartilham apenas uma aresta. As submatrizes do primeiro caso tendem a ter muito mais dados do que a segunda, e nenhuma política no *framework* leva isso em consideração.

Sobre o particionamento em 3 eixos coordenados, como mostra o item (c) da Figura 4.3, a esparsidade de domínios típica desse esquema contém 27 diagonais. A Tabela 4.16 mostra que usando 2000 processos, o número médio de tarefas por *rank* MPI é 11854, ou seja, uma ordem de grandeza maior que com particionamento em 2 eixos e duas ordens de grandeza maior que com particionamento em 1 eixo. Logo, os efeitos negativos do tamanho do grafo discutido no parágrafo anterior são muito mais impactantes aqui. Além disso, o particionamento em 3 eixos também introduz submatrizes de conexão como as do item (d) da Figura 2.11, ou seja, quando dois domínios compartilham apenas um vértice. Isso aumenta ainda mais a heterogeneidade da granularidade dos domínios. Uma ponderação importante é que esse esquema também não é muito usado no GeomecBR, pois divisões no eixo z costumam afetar negativamente a convergência dos métodos iterativos empregados. Além disso, os resultados dos testes mostram que mesmo o PETSc perde desempenho nesse esquema (em menor grau, claro).

Mesmo sem alterar o *framework*, é possível melhorar o desempenho do PtAP. Um dos artifícios é usar matrizes blocadas de fato (as usadas nos testes foram convertidas em escalares por limitação do PETSc, ver Figura 4.22). Isso é facilmente implementável no SolverBR (foi implementado em versões antigas do PtAP), e só não foi feito ainda porque o foco era a comparação com o PETSc para esse trabalho. A expectativa é que o uso de matriz blocada equilibre os tempos à favor do *dataflow*, pois além de trabalhar com uma esparsidade menor nas matrizes, as operações podem ser vetorizadas. Além disso, outra possibilidade, ainda sem alterar o *framework*, é implementar um algoritmo PtAP que agrupe domínios de matrizes (mesmo que apenas logicamente) de modo a reduzir os tamanhos dos grafos (essa estratégia de agrupar tarefas é comum em tecnologias *dataflow*). Quanto ao *framework*, o primeiro passo seria avaliar a implementação de um *runtime* mais sofisticado, que gerencie melhor as tarefas de comunicação (por exemplo, usando *threads*).

Vale destacar os tempos de *setup* do *dataflow*. Mesmo com a complexidade do PtAP, o tempo de *setup* foi pequeno em todas as rodadas dos casos de interesse (para todos os tamanhos de matriz e esquemas de particionamento). O que é considerado um bom resultado. Até porque, esse tempo é diluído em múltiplas execuções do *dataflow* (o *setup* só é feito na etapa simbólica do PtAP). Já para os casos de extrapolação, particionando em 1 eixo, o *setup* foi praticamente imperceptível. Com particionamento em 2 eixos, o tempo é perceptível e até significativo para matrizes pequenas, mas em geral parece aceitável. Com particionamento em 3 eixos, o

tempo de *setup* foi significativo com matrizes médias, e bastante alto com matrizes pequenas, chegando a consumir mais tempo no *setup* do que o PETSc consome na operação inteira. O tamanho da matriz pode explicar, ao menos em parte, o problema:

- Para essa matriz, o tamanho do *grid* é: 2000x50x10 = 1000000 elementos
- O particionamento em 3 eixos para 2000 domínios usados no teste é: 20x10x10
- Logo, os *subgrids* têm tamanhos: (2000/20)x(50/10)x(10/10) = 100x5x1
- Logo, as submatrizes diagonais têm ordem: (100+1)x(5+1)x(1+1) = 1212 blocos 3x3, ou 1212x3 = 3636 em valores escalares.

Note que a ordem média dos grafos locais de cada *rank* (11854, conforme a Tabela 4.16) é maior do que a ordem da submatriz diagonal. Nesse nível de granularidade, não se espera que um *framework dataflow* com grafo estático seja efetivo para essa operação. Pode ser que uma abordagem dinâmica seja mais eficiente nesse caso. No entanto, trata-se de um caso de extrapolação, ou seja, uma situação incomum de uso na prática.

5.2 Trabalhos Futuros

O *framework* aqui apresentado está em sua versão inicial, e o desenvolvimento priorizou as funcionalidades necessárias aos núcleos selecionados para esse trabalho, muitas vezes em detrimento de algumas *features* importantes do ponto de vista conceitual. Portanto, dentre as atividades futuras, a mais importante é a resolução dos débitos tecnológicos discutidos na Seção 3.3.4, que daria completude e robustez ao *framework*. No entanto, isso não encerra as evoluções possíveis. Longe disso. Na Seção 5.1.3, foram sugeridas algumas melhorias para o núcleo PtAP, em particular. E, abaixo, segue uma lista com algumas ideias que podem ser desenvolvidas no futuro:

- Suporte a aceleradores: o *framework* em si já suporta esquema híbrido MPI+OpenMP. O próximo passo é dar suporte a MPI+GPU, também um requisito importante para o SolverBR. As principais classes da ferramenta já têm pontos de extensão projetadas para essa *feature*.
- Tarefas coletivas: avaliar a viabilidade de introduzir tarefas coletivas. Pode ser útil em muitos algoritmos (se for viável).
- *Owning* de domínios mais sofisticado: atualmente um domínio pertence a um *rank* e somente a ele. Avaliar viabilidade de estratégias mais flexíveis, como movimentação de domínios, domínios replicados ou com múltiplos *owners*, hierarquia de domínios, etc. Essas questões podem facilitar a implementação do próximo item.

- Balanceamento de carga: estudar as melhores possibilidades para implementar balanceamento de carga.
- Tratar dependências redundantes do grafo: remover dependências redundantes, diminuindo as arestas do grafo.
- Escalonamento baseado em carga de dados: atualmente, o *framework* só leva em conta o tipo da tarefa e a ordem de definição na hora de ordenar as tarefas no grafo. O particionamento do *grid* em 3 eixos tende a gerar domínios de matrizes com cargas bastante heterogêneas. O escalonamento pode melhorar se a carga de dados puder ser levada em conta.
- Priorização de tarefas: permitir priorização de tarefas pelo usuário, dando a ele mais controle sobre a execução do algoritmo.
- Suporte a operações condicionais: avaliar o suporte a operações de desvio no grafo *dataflow* (como no item(b) da Figura 2.12).
- Ordenação topológica: no contexto de grafo distribuído atual, uma ordenação topológica pode ser bem custosa computacionalmente. É preciso estudar as opções.
- Grafo global: avaliar o uso de grafos globais (replicados por processo e compartilhados entre *threads*), não necessariamente em substituição ao esquema distribuído atual. Os dois modelos poderiam coexistir na ferramenta, conferindo mais flexibilidade ao *framework*.
- *Dataflow* dinâmico: essa é uma funcionalidade bastante interessante por permitir a implementação de algoritmos com reentrâncias (como métodos iterativos), e também por não depender de uma etapa de *setup*, como no caso estático. Essa *feature* introduziria uma complexidade grande ao *framework*, especialmente no contexto distribuído. É necessário estudar com cuidado as opções.

5.3 Considerações Finais

A principal contribuição desse trabalho foi a implementação de um *framework dataflow* distribuído para ser usado no SolverBR em problemas de álgebra linear esparsa. O objetivo da ferramenta é conciliar maior produtividade no desenvolvimento de algoritmos paralelos com bom desempenho. Do ponto de vista da produtividade, o *framework* parece cumprir bem seu papel, pois o conceito de *generators* permite que a descrição do algoritmo em termos de tarefas foque apenas nos processamentos de dados locais. Do ponto de vista do desempenho, os núcleos *dataflow* para MatVec e Transp. MatVec foram competitivos com o PETSc em todos os casos. Porém, os núcleos *dataflow* para PtAP só foram competitivos quando usados com particionamento simples (em 1 eixo coordenado). Nos demais casos, os tempos foram

significativamente inferiores aos do PETSc. No entanto, essa operação é uma das mais complexas em álgebra linear esparsa. Além disso, as múltiplas possibilidades de melhorias discutidas nas seções anteriores podem tornar o *framework* competitivo nessa operação futuramente.

Referências Bibliográficas

[1] GASPARINI, L., RODRIGUES, J. R., AUGUSTO, D. A., et al. "Hybrid parallel iterative sparse linear solver framework for reservoir geomechanical and flow simulation", *Journal of Computational Science*, v. 51, pp. 101330, 2021. ISSN: 1877-7503. doi: https://doi.org/10.1016/j.jocs.2021.101330. Disponível em: <`https://www.sciencedirect.com/science/article/pii/S1877750321000284`>.

[2] MARZULO, L. A. J. *Explorando Linhas de Execução Paralelas com Programação Orientada por Fluxo de Dados*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2011.

[3] THOMAN, P., DICHEV, K., HELLER, T., et al. "A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing", *J. Supercomput.*, v. 74, n. 4, pp. 1422–1434, abr. 2018. ISSN: 0920-8542. doi: 10.1007/s11227-018-2238-4. Disponível em: <`https://doi.org/10.1007/s11227-018-2238-4`>.

[4] FRANÇOIS, A., MAUBEUGE, F., TOURTE, E., et al. "High-Performance Computing: Towards a Safer, Faster and Sharper Exploration & Production". p. 22, Moscow, Russia, jan. 2014. World Petroleum Congress. WPC-21-1350.

[5] AZIZ, K. *Petroleum Reservoir Simulation*. 1 st ed. London, England, Springer, 1979.

[6] ZOBACK, M. D. *Reservoir Geomechanics*. California, USA, Cambridge, 2007.

[7] SOUSA JR., L. C. D., DOS SANTOS, L. O. S., DE SOUZA RIOS, V., et al. "Methodology for geomechanically controlled transmissibility through active natural fractures in reservoir simulation", *Journal of Petroleum Science and Engineering*, v. 147, pp. 7–14, 2016.

[8] CHUNG, T., OTHERS. *Computational fluid dynamics*. Cambridge; New York, Cambridge university press, 2002.

[9] CHEN, Z., HUAN, G., MA, Y. *Computational Methods for Multiphase Flows in Porous Media*. USA, SIAM, 2006.

[10] JANSEN, J.-D., BROUWER, R., DOUMA, S. G. "Closed Loop Reservoir Management". p. 18, The Woodlands, Texas, jan. 2009. Society of Petroleum Engineers. doi: http://dx.doi.org/10.2118/119098-MS.

[11] HANSSEN, K. G., CODAS, A., FOSS, B. "Closed-Loop Predictions in Reservoir Management Under Uncertainty", *SPE Journal*, v. 22, n. 05, pp. 1585–1595, out. 2017. ISSN: 1086-055X. doi: http://dx.doi.org/10.2118/185956-PA.

[12] LE, D. H., EMERICK, A. A., REYNOLDS, A. C. "An Adaptive Ensemble Smoother With Multiple Data Assimilation for Assisted History Matching", *SPE Journal*, v. 21, n. 06, pp. 2195–2207, dez. 2016. ISSN: 1086-055X. doi: http://dx.doi.org/10.2118/173214-PA.

[13] DONGARRA, J. J., MEUER, H. W., STROHMAIER, E., et al. "TOP500 supercomputer sites", *Supercomputer*, v. 13, pp. 89–111, 1997.

[14] FIGUEIREDO, M. O., RODRIGUES, J. R. P., FRANÇOIS, J. P., et al. "Parallelization of a geomechanics simulator using MPI". In: *Rio Oil & Gas Expo and Conference 2018, Rio de Janeiro, Brazil. In Portuguese.*, set. 2018.

[15] GASPARINI, L., RODRIGUES, J. R. P., CONOPOIMA, C., et al. "A Linear Solver Framework for Flow and Geomechanics Reservoir Simulation". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 708–717. IEEE, 2019.

[16] DOGRU, A. H., SUNAIDI, H. A., FUNG, L. S., et al. "A Parallel Reservoir Simulator for Large-Scale Reservoir Simulation", *SPE Reservoir Evaluation & Engineering*, v. 5, n. 01, pp. 11–23, 2002. ISSN: 1094-6470. doi: http://dx.doi.org/10.2118/75805-PA.

[17] DOGRU, A. H., FUNG, L. S., AL-SHAALAN, T. M., et al. "From mega-cell to giga-cell reservoir simulation". In: *2008 SPE Annual Technical Conference and Exhibition*, pp. 1–19, Denver, Colorado, USA., set. 2008. SPE. Paper SPE 116675.

[18] SMITH, I. M., GRIFFITHS, D. V. *Programming the Finite Element Method*. 2 nd ed. USA, Jonh Wiley & Sons, 1988.

[19] DE FIGUEIREDO, M. O. *PRECONDICIONADOR MULTIESCALA APLICADOA GEOMEC^ANICA*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2019.

[20] RODRIGUEZ, J. E. "A graph models for parallel computations". 1969.

[21] SUTHERLAND, W. R. "The on-line graphical specification of computer procedures". 1966.

[22] DENNIS, J. "Data Flow Supercomputers", *Computer*, v. 13, n. 11, pp. 48–56, nov. 1980. ISSN: 1558-0814. doi: 10.1109/MC.1980.1653418.

[23] DE ARAÚJO, L. S. *OTIMIZANDO LACOS EM COMPUTAC AO POR FLUXO DE DADOS*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2016.

[24] GOLDSTEIN, B. F., FRANÇA, F. M. G., MARZULO, L. A. J., et al. "Exploiting Parallelism in Linear Algebra Kernels through Dataflow Execution". In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pp. 103–108, 2015. doi: 10.1109/SBAC-PADW.2015.21.

[25] ALVES, T. A., MARZULO, L. A., FRANÇA, F. M., et al. "Trebuchet: exploring TLP with dataflow virtualisation", *International Journal of High Performance Systems Architecture*, v. 3, n. 2-3, pp. 137–148, 2011.

[26] ALVES, T. A., GOLDSTEIN, B. F., FRANÇA, F. M., et al. "A minimalistic dataflow programming library for python". In: *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pp. 96–101. IEEE, 2014.

[27] BOSILCA, G., BOUTEILLER, A., DANALIS, A., et al. "DAGuE: A Generic Distributed DAG Engine for High Performance Computing". pp. 1151–1158, Anchorage, Alaska, USA, 2011-00 2011. IEEE.

[28] BOSILCA, G., BOUTEILLER, A., DANALIS, A., et al. "Parsec: Exploiting heterogeneity to enhance scalability", *Computing in Science & Engineering*, v. 15, n. 6, pp. 36–45, 2013.

[29] AUGONNET, C., THIBAULT, S., NAMYST, R., et al. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures", *Concurrency and Computation: Practice and Experience*, v. 23, n. 2, pp. 187–198, 2011.

[30] LACOSTE, X., FAVERGE, M., BOSILCA, G., et al. "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes". In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pp. 29–38. IEEE, 2014.

[31] MO, T., LI, R. "Iteratively solving sparse linear system based on PaRSEC task scheduling", *The International Journal of High Performance Computing Applications*, v. 34, n. 3, pp. 306–315, 2020.

[32] AGULLO, E., BOSILCA, G., BUTTARI, A., et al. "Exploiting a Parametrized Task Graph model for the parallelization of a sparse direct multifrontal solver". In: *European Conference on Parallel Processing*, pp. 175–186. Springer, 2016.

[33] DAGUM, L., MENON, R. "OpenMP: an industry standard API for shared-memory programming", *IEEE Computational Science and Engineering*, v. 5, n. 1, pp. 46–55, 1998. doi: 10.1109/99.660313.

[34] DURAN, A., AYGUADÉ, E., BADIA, R. M., et al. "Ompss: a proposal for programming heterogeneous multi-core architectures", *Parallel processing letters*, v. 21, n. 02, pp. 173–193, 2011.

[35] COSNARD, M., LOI, M. "Automatic task graph generation techniques". In: *Proceedings of the twenty-eighth annual Hawaii international conference on system sciences*, v. 2, pp. 113–122. IEEE, 1995.

[36] FOWLER, M. *Domain-Specific Languages*. Upper Saddle River, NJ, Addison-Wesley, 2010. ISBN: 978-0-321-71294-3. Disponível em: <`https://www.safaribooksonline.com/library/view/domain-specific-languages/9780132107549/`>.

[37] JÄRVI, J., FREEMAN, J. "C++ lambda expressions and closures", *Science of Computer Programming*, v. 75, n. 9, pp. 762–772, 2010. ISSN: 0167-6423. doi: https://doi.org/10.1016/j.scico.2009.04.003. Disponível em: <`https://www.sciencedirect.com/science/article/pii/S0167642309000720`>. Special Issue on Object-Oriented Programming Languages and Systems (OOPS 2008), A Special Track at the 23rd ACM Symposium on Applied Computing.

[38] SAAD, Y. "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2". 1994.

[39] PRIYA, M. M. "Topological sorting". 2012.

[40] FIGUEIREDO, M., ILIEV, CARVALHO, L. M., MORAES, R. J. "An efficient multiscale preconditioning strategy for linear elasticgeomechanics", *TBD*, To be submitted.

[41] KORANNE, S. "Boost C++ Libraries". In: *Handbook of Open Source Tools*, pp. 127–143, Boston, MA, Springer US, 2011. ISBN: 978-1-4419-7719-9. doi: 10.1007/978-1-4419-7719-9_6. Disponível em: <https://doi.org/10.1007/978-1-4419-7719-9_6>.

[42] ELLSON, J., GANSNER, E., KOUTSOFIOS, L., et al. "Graphviz— Open Source Graph Drawing Tools". In: Mutzel, P., Jünger, M., Leipert, S. (Eds.), *Graph Drawing*, pp. 483–484, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN: 978-3-540-45848-7.

[43] GANTT, H., ADAMIECKI, K. "Gantt chart". 2015.

[44] BALAY, S., ABHYANKAR, S., ADAMS, M. F., et al. *PETSc Users Manual*. Relatório Técnico ANL-95/11 - Revision 3.14, Argonne National Laboratory, 2020. Disponível em: <https://www.mcs.anl.gov/petsc>.

[45] VAN DER VORST, H. A. *Iterative Krylov methods for large linear systems*. Cambridge; New York, Cambridge University Press, 2003. ISBN: 0521818281 9780521818285. Disponível em: <https://www.worldcat.org/title/iterative-krylov-methods-for-large-linear-systems/oclc/50717963&referer=brief_results>.

[46] CHAN, T., TUMINARO, R. S. "SURVEY OF PARALLEL MULTIGRID ALGORITHMS." *American Society of Mechanical Engineers, Applied Mechanics Division, AMD*, v. 86, pp. 155–170, 1987.

Apêndice A

Principais Interfaces do *Framework Dataflow*

Aqui são listadas as interfaces das principais classes do *framework*, que precisam ser realizadas pelo usuário.

A.1 ProcessingEnvironment/ProcessingUnit - Classes de abstração do ambiente de execução

A classe ProcessingEnvironment representa o ambiente de execução paralelo. Ela agrega uma coleção de instâncias da classe ProcessingUnit, que é uma abstração para a unidade de processamento onde, de fato, cada tarefa é executada. Por hora, há apenas uma implementação de ProcessingUnit, que representa um *rank* MPI. Portanto, no contexto desse trabalho, *rank* MPI e unidade de processamento podem ser considerados sinônimos. Futuramente, a classe ProcessingUnit poderá ter outras implementações, como, por exemplo, uma representação de GPU, entre outras. A ferramenta provê uma *factory* que instancia um ProcessingEnvironment com um ProcessingUnit por *rank* MPI.

A.1.1 DataObject/Domain - Classes de dados distribuídos

A classe abstrata DataObject representa um dado distribuído particionado em domínios. A partição é multidimensional, e é definida no construtor da classe. Por exemplo, se o DataObject representa uma matriz, então pode-se usar dois índices para mapear seus domínios, enquanto se o DataObject representa um vetor, o particionamento pode ser unidimensional, naturalmente. O construtor da classe recebe uma instância de ProcessingEnvironment, a cardinalidade em termos de domínios (uma tupla n-dimensional com o número máximo de domínios por dimensão), e uma

string com o nome do objeto. Essa classe tem um método abstrato puro chamado *domain*, que recebe uma tupla n-dimensional de índices (respeitando a cardinalidade definida no construtor) e retorna um ponteiro para a classe Domain. Ver Código A.1.

Código A.1: Fragmento da classe DataObject

```
1  class DataObject
2  {
3  public:
4    DataObject( const boost::shared_ptr<ProcessingEnvironment>& penv,
5                const ids_t& cardinality, const std::string& name );
6    virtual Domain* domain( const ids_t& ids ) const = 0;
7    // ... other methods
8  };
```

A classe Domain representa um domínio (idealmente *coarse grain*) de um DataObject. Cada instância de Domain possui um *owner unit*, que é a unidade de processamento (ou *rank* MPI). O construtor dessa classe recebe um ponteiro para o DataObject pai, uma tupla n-dimensional com os índices que identificam o domínio e um ponteiro para a unidade de processamento dona do objeto. Essa classe possui quatro métodos abstratos puros (nStepsGet, getTask, nStepsPut e putTask) relacionados a movimentação de dados entre unidades de processamento distintas pelo *framework*. Sempre que um domínio puder ser usado fora da sua unidade de processamento, o usuário precisa implementar esses métodos. Se, por algum motivo, o domínio sempre for usado só localmente, existe a classe concreta LocalDomain que só permite o acesso a domínios locais (e exime o usuário de implementar tais métodos). O parâmetro arcId é um metadado que o framework passa ao método que ajuda a definir TAGs MPI não conflitantes. Futuramente será movido para uma classe de contexto.

Código A.2: Fragmento da classe Domain

```
1  class Domain
2  {
3  public:
4    Domain( DataObject* parent, const ids_t& ids, ProcessingUnit* ownerUnit );
5    virtual int nStepsGet() const = 0;
6    virtual boost::shared_ptr<AsyncHandler> getTask( int const step, int arcId ) = 0;
7    virtual int nStepsPut() const = 0;
8    virtual boost::shared_ptr<AsyncHandler> putTask( const ProcessingUnit& targetUnit,
9                                                     int const step, int arcId ) = 0;
10   // ... other methods
11 };
```

A.1.2 Closure - Classe de tarefas

A classe abstrata Closure representa tarefas parametrizadas por índices (que endereçam os operandos, domínios, no caso). O construtor da classe recebe uma instância de ProcessingEnvironment, a cardinalidade dos índices (uma tupla n-dimensional com o número máximo por dimensão), e uma *string* com o nome da closure. Essa

classe tem dois métodos abstratos puros: nSteps e task. O primeiro indica quantos steps a tarefa possui, e o segundo representa a tarefa em si. Ver Código A.3.

Código A.3: Fragmento da classe Closure

```
1  class Closure
2  {
3  public:
4     Closure( ProcessingEnvironment* penv, const ids_t& cardinality, const std::string& name );
5     virtual int nSteps( const ids_t& indexes ) = 0;
6     virtual boost::shared_ptr<AsyncHandler> task( const ids_t& ids, int const step ) = 0;
7     // ... other methods
8  };
```

Apêndice B

Experimentos numéricos no SDumont

Seguem os experimentos numéricos realizados no SDumont, e que ainda precisam ser validados (Seção 4.1.4).

B.1 Resultados - Casos de interesse prático

B.1.1 Operação y = Ax

Matriz de ordem 1 milhão (bloco 3x3), 1 nó

Figura B.1: Gráficos de tempos da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

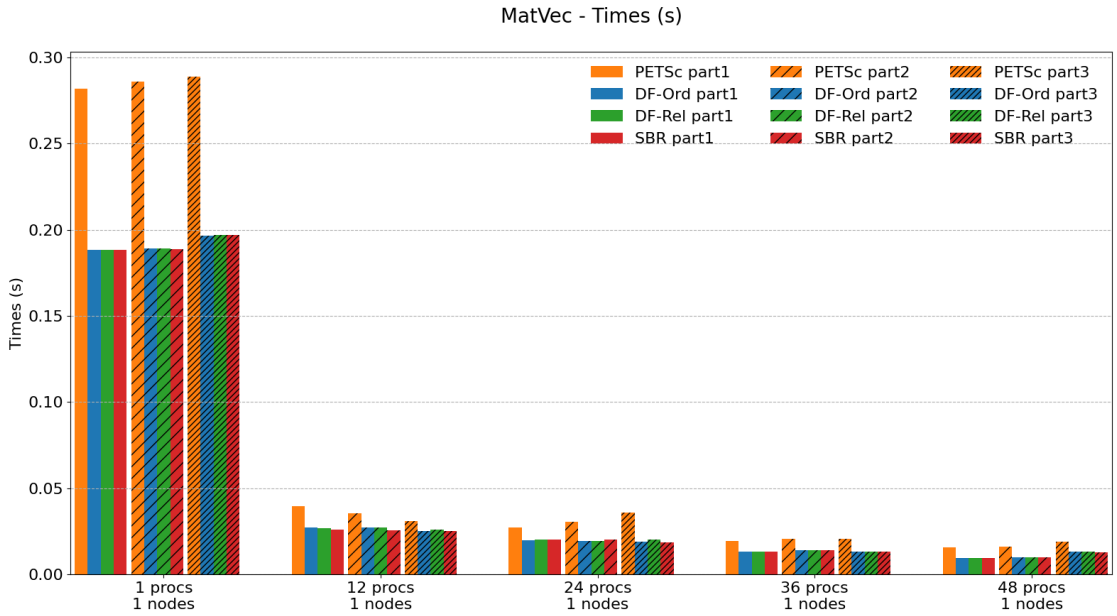


Tabela B.1: Tempos e Speed-up da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
1	1	0.2819	0.1882	0.1885	1.5	1.5
12	1	0.0397	0.0273	0.0267	1.5	1.5
24	1	0.0271	0.0198	0.0202	1.4	1.3
36	1	0.0193	0.0131	0.0133	1.5	1.5
48	1	0.0157	0.0097	0.0097	1.6	1.6

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
1	1	0.2859	0.1891	0.1893	1.5	1.5
12	1	0.0357	0.0271	0.0271	1.3	1.3
24	1	0.0306	0.0195	0.0195	1.6	1.6
36	1	0.0207	0.0140	0.0140	1.5	1.5
48	1	0.0162	0.0101	0.0101	1.6	1.6

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
1	1	0.2890	0.1968	0.1972	1.5	1.5
12	1	0.0311	0.0254	0.0258	1.2	1.2
24	1	0.0357	0.0191	0.0202	1.9	1.8
36	1	0.0208	0.0131	0.0131	1.6	1.6
48	1	0.0191	0.0132	0.0131	1.5	1.5

Matriz de ordem 10 milhões (bloco 3x3), até 8 nós

Figura B.2: Gráficos de tempos da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

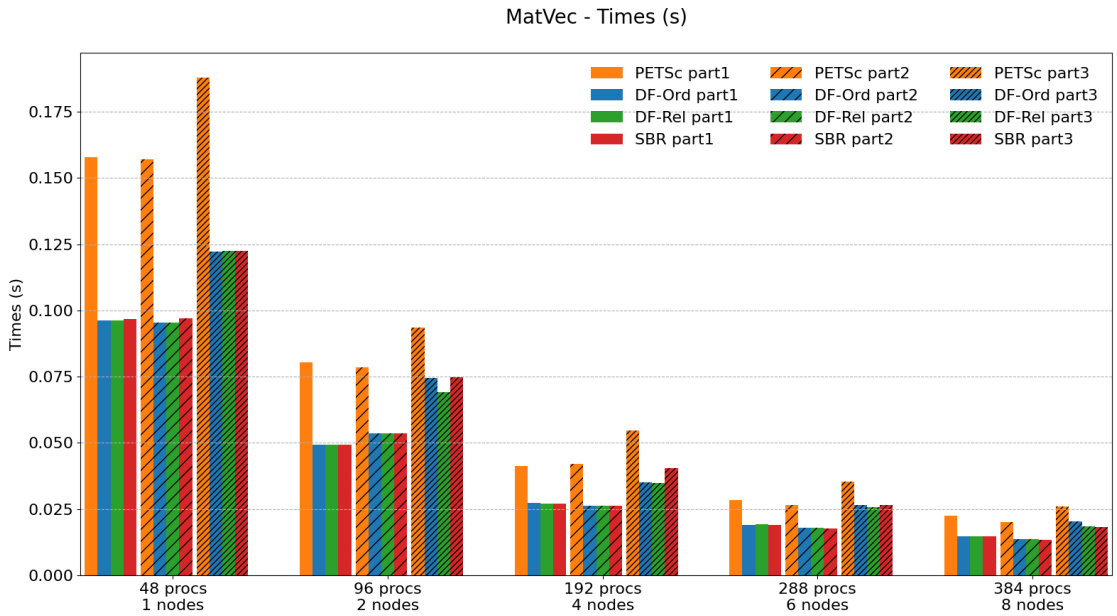


Tabela B.2: Tempos e Speed-up da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
48	1	0.1578	0.0962	0.0962	1.6	1.6
96	2	0.0804	0.0493	0.0493	1.6	1.6
192	4	0.0412	0.0273	0.0272	1.5	1.5
288	6	0.0285	0.0191	0.0194	1.5	1.5
384	8	0.0224	0.0148	0.0149	1.5	1.5

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
48	1	0.1572	0.0955	0.0954	1.6	1.6
96	2	0.0784	0.0537	0.0535	1.5	1.5
192	4	0.0421	0.0263	0.0263	1.6	1.6
288	6	0.0264	0.0179	0.0179	1.5	1.5
384	8	0.0200	0.0136	0.0137	1.5	1.5

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
48	1	0.1879	0.1223	0.1224	1.5	1.5
96	2	0.0936	0.0745	0.0690	1.3	1.4
192	4	0.0548	0.0351	0.0349	1.6	1.6
288	6	0.0353	0.0266	0.0258	1.3	1.4
384	8	0.0260	0.0205	0.0186	1.3	1.4

Matriz de ordem 100 milhões (bloco 3x3), até 50 nós

Figura B.3: Gráficos de tempos da operação MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

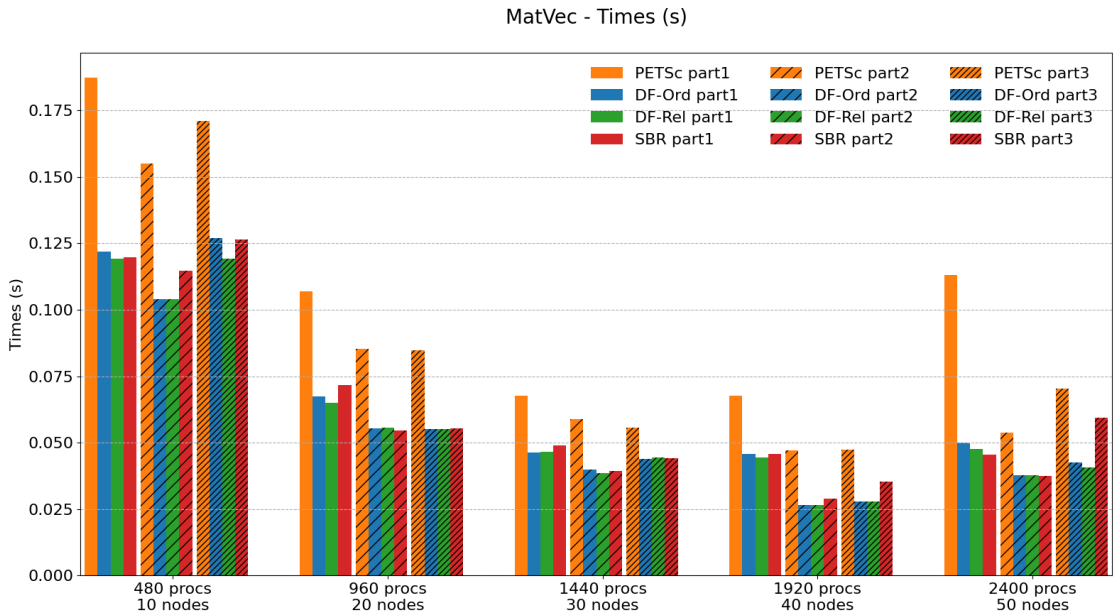


Tabela B.3: Tempos e Speed-up da operação MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
480	10	0.1875	0.1219	0.1192	1.5	1.6
960	20	0.1070	0.0674	0.0651	1.6	1.6
1440	30	0.0675	0.0462	0.0464	1.5	1.5
1920	40	0.0675	0.0457	0.0444	1.5	1.5
2400	50	0.1130	0.0501	0.0475	2.3	2.4

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
480	10	0.1551	0.1041	0.1040	1.5	1.5
960	20	0.0853	0.0554	0.0556	1.5	1.5
1440	30	0.0589	0.0398	0.0385	1.5	1.5
1920	40	0.0472	0.0266	0.0266	1.8	1.8
2400	50	0.0538	0.0378	0.0378	1.4	1.4

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
480	10	0.1711	0.1271	0.1192	1.3	1.4
960	20	0.0847	0.0551	0.0550	1.5	1.5
1440	30	0.0556	0.0440	0.0443	1.3	1.3
1920	40	0.0474	0.0278	0.0277	1.7	1.7
2400	50	0.0702	0.0426	0.0407	1.6	1.7

B.1.2 Operação $\mathbf{y = A^T x}$

Matriz de ordem 1 milhão (bloco 3x3), 1 nó

Figura B.4: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

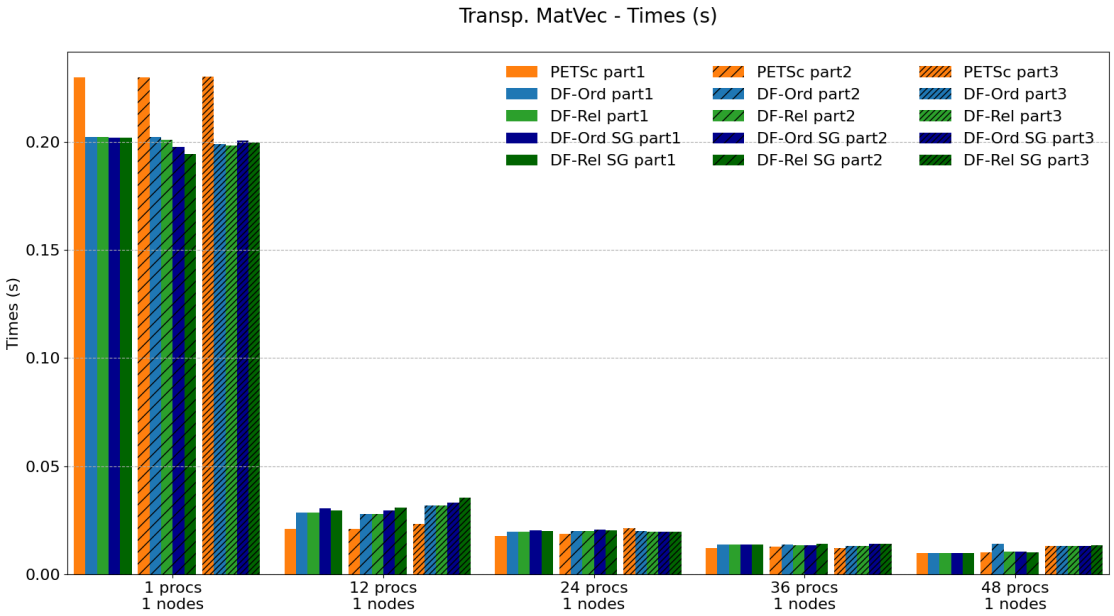


Tabela B.4: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

Tabela B.5: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
1	1	0.2296	0.2022	0.2023	0.2020	0.2018	1.1	1.1	1.1	1.1
12	1	0.0209	0.0285	0.0285	0.0306	0.0295	0.7	0.7	0.7	0.7
24	1	0.0179	0.0198	0.0199	0.0204	0.0202	0.9	0.9	0.9	0.9
36	1	0.0123	0.0139	0.0138	0.0138	0.0137	0.9	0.9	0.9	0.9
48	1	0.0099	0.0099	0.0100	0.0099	0.0099	1.0	1.0	1.0	1.0

Tabela B.6: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
1	1	0.2297	0.2023	0.2009	0.1977	0.1944	1.1	1.1	1.2	1.2
12	1	0.0211	0.0279	0.0279	0.0296	0.0310	0.8	0.8	0.7	0.7
24	1	0.0188	0.0200	0.0201	0.0206	0.0204	0.9	0.9	0.9	0.9
36	1	0.0127	0.0138	0.0136	0.0136	0.0143	0.9	0.9	0.9	0.9
48	1	0.0103	0.0140	0.0104	0.0104	0.0103	0.7	1.0	1.0	1.0

Tabela B.7: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
1	1	0.2302	0.1988	0.1984	0.2004	0.1994	1.2	1.2	1.1	1.2
12	1	0.0232	0.0319	0.0319	0.0332	0.0353	0.7	0.7	0.7	0.7
24	1	0.0215	0.0199	0.0198	0.0198	0.0197	1.1	1.1	1.1	1.1
36	1	0.0121	0.0131	0.0131	0.0141	0.0140	0.9	0.9	0.9	0.9
48	1	0.0133	0.0132	0.0132	0.0132	0.0133	1.0	1.0	1.0	1.0

Matriz de ordem 10 milhões (bloco 3x3), até 8 nós

Figura B.5: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

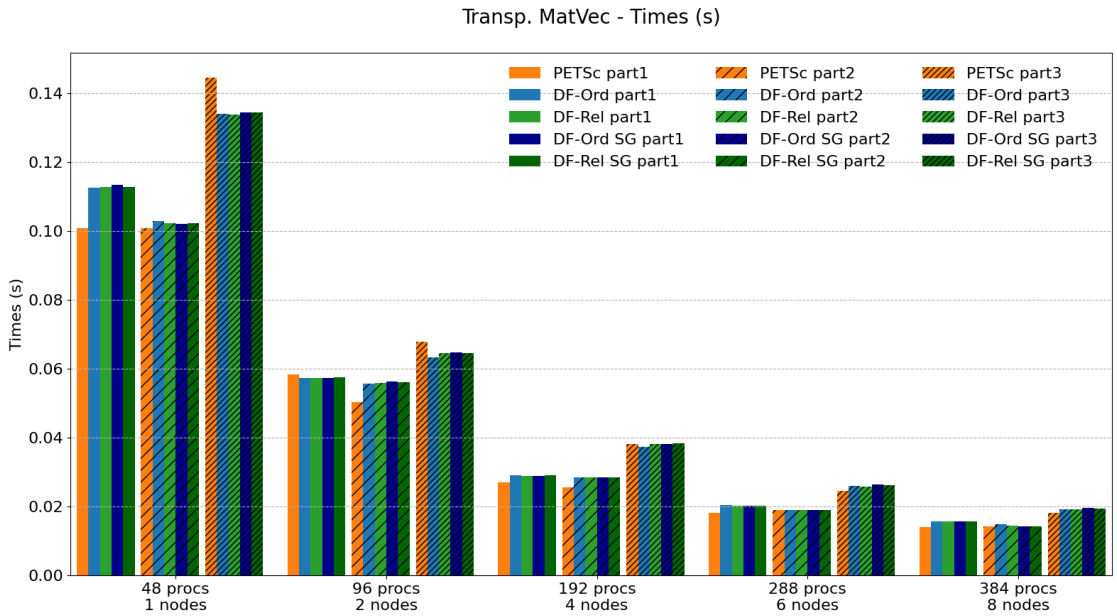


Tabela B.8: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

Tabela B.9: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
48	1	0.1009	0.1126	0.1128	0.1134	0.1129	0.9	0.9	0.9	0.9
96	2	0.0585	0.0574	0.0574	0.0573	0.0574	1.0	1.0	1.0	1.0
192	4	0.0269	0.0290	0.0289	0.0289	0.0290	0.9	0.9	0.9	0.9
288	6	0.0182	0.0204	0.0203	0.0203	0.0202	0.9	0.9	0.9	0.9
384	8	0.0141	0.0157	0.0157	0.0157	0.0157	0.9	0.9	0.9	0.9

Tabela B.10: Grid partitioned in 2 axes

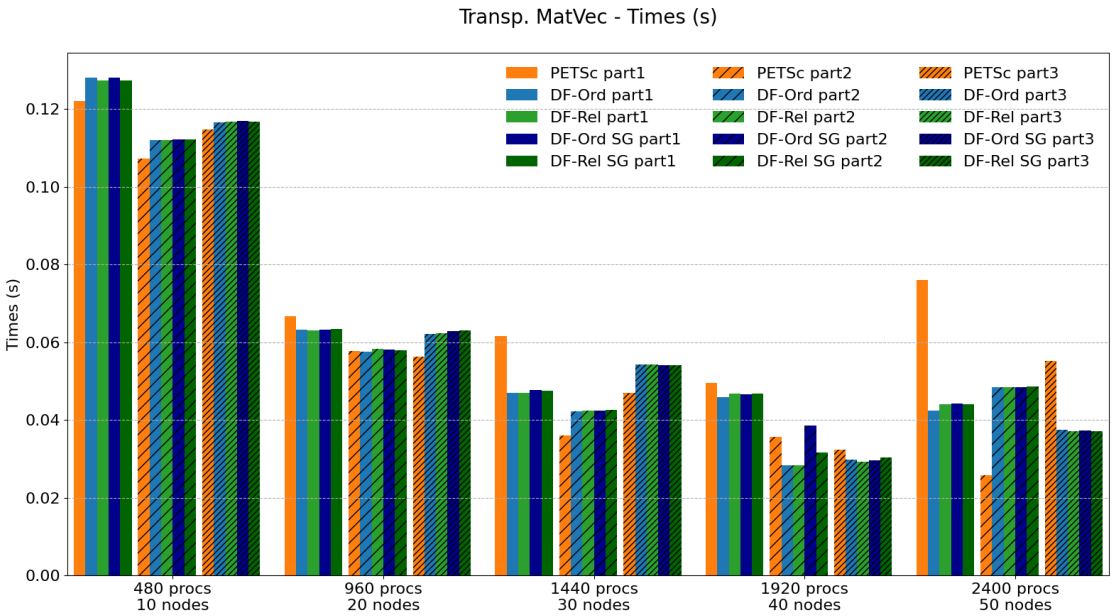
Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
48	1	0.1008	0.1028	0.1023	0.1021	0.1022	1.0	1.0	1.0	1.0
96	2	0.0503	0.0557	0.0560	0.0563	0.0560	0.9	0.9	0.9	0.9
192	4	0.0255	0.0285	0.0285	0.0286	0.0284	0.9	0.9	0.9	0.9
288	6	0.0190	0.0191	0.0190	0.0190	0.0190	1.0	1.0	1.0	1.0
384	8	0.0142	0.0148	0.0145	0.0143	0.0142	1.0	1.0	1.0	1.0

Tabela B.11: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
48	1	0.1446	0.1341	0.1338	0.1344	0.1345	1.1	1.1	1.1	1.1
96	2	0.0678	0.0634	0.0646	0.0648	0.0646	1.1	1.0	1.0	1.0
192	4	0.0382	0.0374	0.0381	0.0382	0.0383	1.0	1.0	1.0	1.0
288	6	0.0245	0.0260	0.0259	0.0264	0.0263	0.9	0.9	0.9	0.9
384	8	0.0182	0.0191	0.0191	0.0196	0.0195	1.0	1.0	0.9	0.9

Matriz de ordem 100 milhões (bloco 3x3), até 50 nós

Figura B.6: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse



B.1.3 Operação $\mathbf{C = P^T AP}$

Tabela B.12: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 100 milhões (bloco 3x3) para os casos de interesse

Tabela B.13: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
480	10	0.1220	0.1281	0.1274	0.1281	0.1274	1.0	1.0	1.0	1.0
960	20	0.0667	0.0632	0.0631	0.0633	0.0634	1.1	1.1	1.1	1.1
1440	30	0.0616	0.0470	0.0470	0.0477	0.0475	1.3	1.3	1.3	1.3
1920	40	0.0495	0.0459	0.0469	0.0466	0.0468	1.1	1.1	1.1	1.1
2400	50	0.0760	0.0425	0.0440	0.0441	0.0440	1.8	1.7	1.7	1.7

Tabela B.14: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
480	10	0.1072	0.1120	0.1120	0.1122	0.1122	1.0	1.0	1.0	1.0
960	20	0.0578	0.0575	0.0583	0.0581	0.0580	1.0	1.0	1.0	1.0
1440	30	0.0361	0.0423	0.0423	0.0423	0.0426	0.9	0.9	0.9	0.8
1920	40	0.0356	0.0284	0.0283	0.0386	0.0317	1.3	1.3	0.9	1.1
2400	50	0.0258	0.0485	0.0485	0.0485	0.0485	0.5	0.5	0.5	0.5

Tabela B.15: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
480	10	0.1147	0.1166	0.1167	0.1170	0.1168	1.0	1.0	1.0	1.0
960	20	0.0564	0.0621	0.0623	0.0628	0.0631	0.9	0.9	0.9	0.9
1440	30	0.0470	0.0543	0.0543	0.0540	0.0540	0.9	0.9	0.9	0.9
1920	40	0.0323	0.0297	0.0292	0.0296	0.0303	1.1	1.1	1.1	1.1
2400	50	0.0553	0.0375	0.0370	0.0373	0.0371	1.5	1.5	1.5	1.5

Matriz de ordem 1 milhão (bloco 3x3), 1 nó

Figura B.7: Gráficos de tempos da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

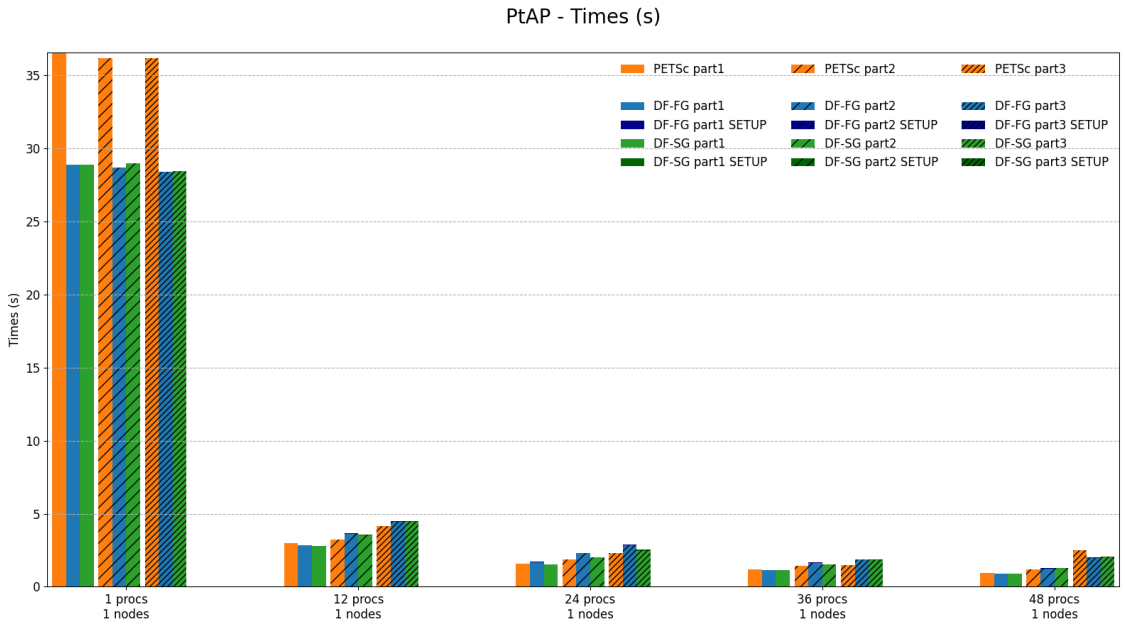


Tabela B.16: Tempos e Speed-up da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
1	1	36.5686	28.8694	28.8899	1.3	1.3
12	1	2.9935	2.8378	2.7778	1.1	1.1
24	1	1.5732	1.7049	1.5295	0.9	1.0
36	1	1.1995	1.1541	1.1461	1.0	1.0
48	1	0.9623	0.8877	0.8880	1.1	1.1

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
1	1	36.1759	28.6986	28.9654	1.3	1.2
12	1	3.2162	3.6609	3.5483	0.9	0.9
24	1	1.8604	2.3212	2.0331	0.8	0.9
36	1	1.4469	1.6679	1.5459	0.9	0.9
48	1	1.1824	1.2681	1.2901	0.9	0.9

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
1	1	36.1630	28.3761	28.4606	1.3	1.3
12	1	4.1581	4.5109	4.5106	0.9	0.9
24	1	2.2890	2.9122	2.5292	0.8	0.9
36	1	1.4967	1.8927	1.8626	0.8	0.8
48	1	2.5187	2.0321	2.0656	1.2	1.2

Matriz de ordem 10 milhões (bloco 3x3), até 8 nós

Figura B.8: Gráficos de tempos da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

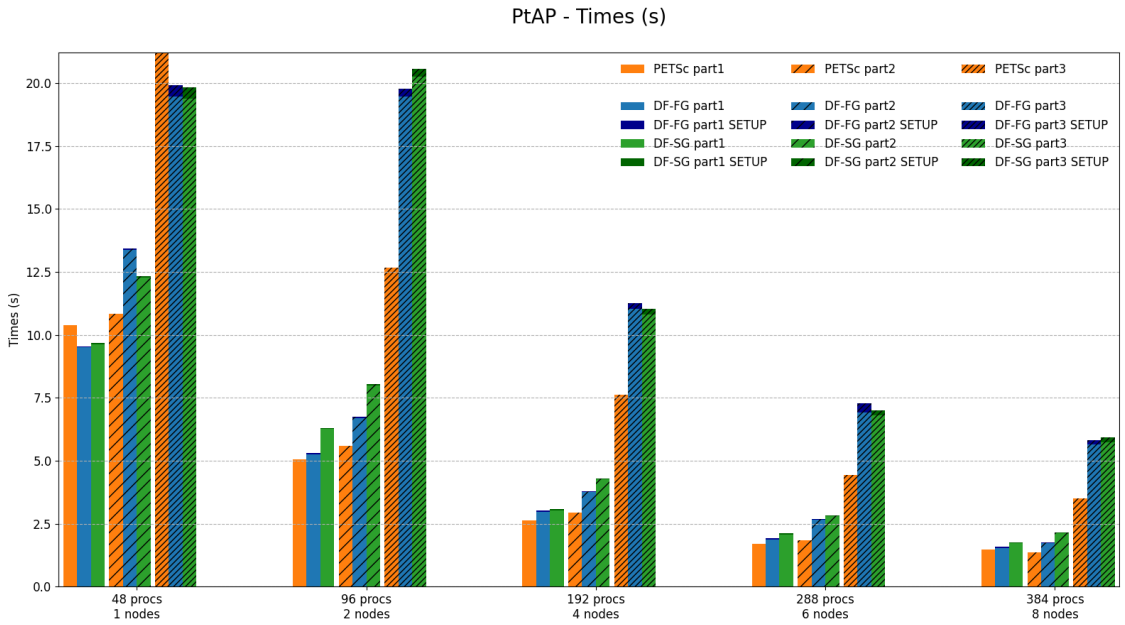


Tabela B.17: Tempos e Speed-up da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de interesse

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
48	1	10.3724	9.5442	9.6691	1.1	1.1
96	2	5.0713	5.2997	6.3031	1.0	0.8
192	4	2.6360	3.0244	3.0750	0.9	0.9
288	6	1.6922	1.9174	2.1140	0.9	0.8
384	8	1.4877	1.5946	1.7700	0.9	0.8

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
48	1	10.8247	13.4316	12.3270	0.8	0.9
96	2	5.5906	6.7371	8.0336	0.8	0.7
192	4	2.9353	3.7873	4.3002	0.8	0.7
288	6	1.8347	2.7015	2.8272	0.7	0.6
384	8	1.3617	1.7648	2.1432	0.8	0.6

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
48	1	21.2161	19.9098	19.8231	1.1	1.1
96	2	12.6787	19.7840	20.5561	0.6	0.6
192	4	7.6107	11.2669	11.0371	0.7	0.7
288	6	4.4371	7.2904	7.0004	0.6	0.6
384	8	3.5070	5.8327	5.9189	0.6	0.6

B.2 Resultados - Casos de extrapolação

B.2.1 Operação y = Ax

Matriz de ordem 1 milhão (bloco 3x3), até 50 nós

Figura B.9: Gráficos de tempos da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

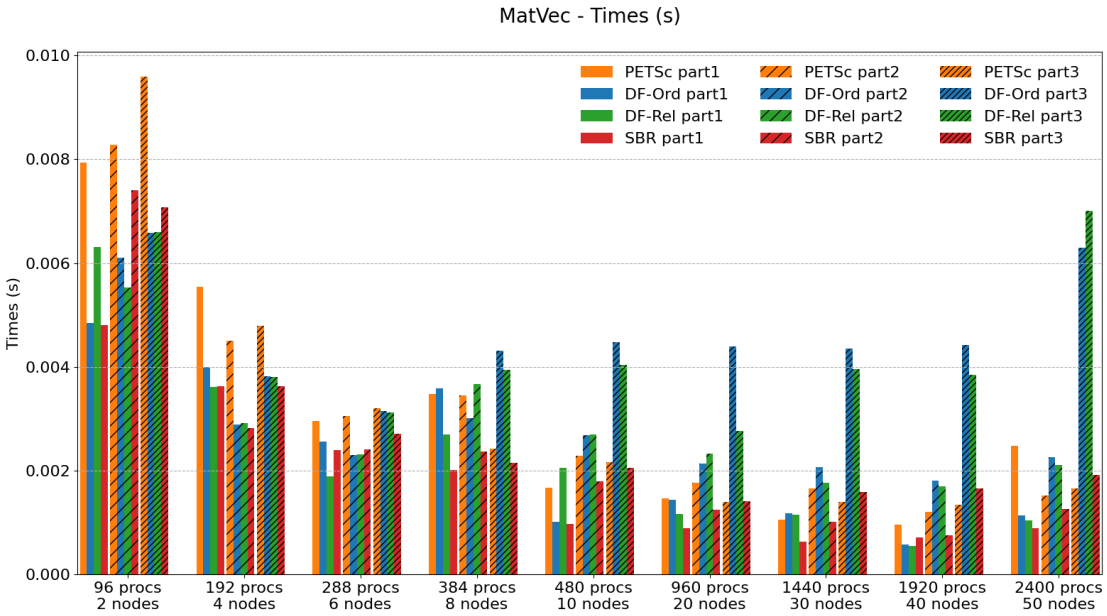


Tabela B.18: Tempos e Speed-up da operação MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
96	2	0.0079	0.0048	0.0063	1.6	1.3
192	4	0.0055	0.0040	0.0036	1.4	1.5
288	6	0.0030	0.0026	0.0019	1.2	1.6
384	8	0.0035	0.0036	0.0027	1.0	1.3
480	10	0.0017	0.0010	0.0020	1.6	0.8
960	20	0.0015	0.0014	0.0012	1.0	1.3
1440	30	0.0011	0.0012	0.0012	0.9	0.9
1920	40	0.0010	0.0006	0.0005	1.7	1.7
2400	50	0.0025	0.0011	0.0010	2.2	2.4

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
96	2	0.0083	0.0061	0.0055	1.4	1.5
192	4	0.0045	0.0029	0.0029	1.6	1.5
288	6	0.0031	0.0023	0.0023	1.3	1.3
384	8	0.0034	0.0030	0.0037	1.1	0.9
480	10	0.0023	0.0027	0.0027	0.9	0.8
960	20	0.0018	0.0021	0.0023	0.8	0.8
1440	30	0.0017	0.0021	0.0018	0.8	0.9
1920	40	0.0012	0.0018	0.0017	0.7	0.7
2400	50	0.0015	0.0023	0.0021	0.7	0.7

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
96	2	0.0096	0.0066	0.0066	1.5	1.5
192	4	0.0048	0.0038	0.0038	1.3	1.3
288	6	0.0032	0.0031	0.0031	1.0	1.0
384	8	0.0024	0.0043	0.0039	0.6	0.6
480	10	0.0022	0.0045	0.0040	0.5	0.5
960	20	0.0014	0.0044	0.0028	0.3	0.5
1440	30	0.0014	0.0044	0.0040	0.3	0.4
1920	40	0.0013	0.0044	0.0038	0.3	0.3
2400	50	0.0017	0.0063	0.0070	0.3	0.2

Matriz de ordem 10 milhões (bloco 3x3), até 50 nós

Figura B.10: Gráficos de tempos da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

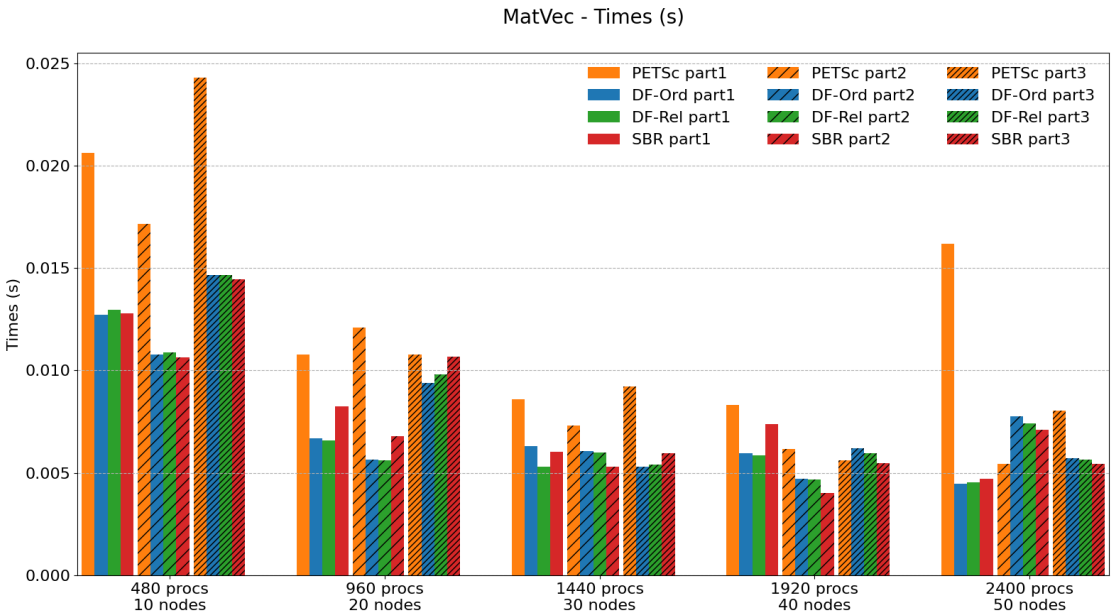


Tabela B.19: Tempos e Speed-up da operação MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
480	10	0.0206	0.0127	0.0129	1.6	1.6
960	20	0.0108	0.0067	0.0066	1.6	1.6
1440	30	0.0086	0.0063	0.0053	1.4	1.6
1920	40	0.0083	0.0060	0.0059	1.4	1.4
2400	50	0.0162	0.0045	0.0045	3.6	3.6

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
480	10	0.0171	0.0108	0.0109	1.6	1.6
960	20	0.0121	0.0057	0.0056	2.1	2.1
1440	30	0.0073	0.0061	0.0060	1.2	1.2
1920	40	0.0062	0.0047	0.0047	1.3	1.3
2400	50	0.0054	0.0078	0.0074	0.7	0.7

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-Ord	DF-Rel	PETSc / DF-Ord	PETSc / DF-Rel
480	10	0.0243	0.0146	0.0147	1.7	1.7
960	20	0.0108	0.0094	0.0098	1.1	1.1
1440	30	0.0092	0.0053	0.0054	1.7	1.7
1920	40	0.0056	0.0062	0.0060	0.9	0.9
2400	50	0.0080	0.0057	0.0057	1.4	1.4

B.2.2 Operação $\mathbf{y = A^T x}$

Matriz de ordem 1 milhão (bloco 3x3), até 50 nós

Figura B.11: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

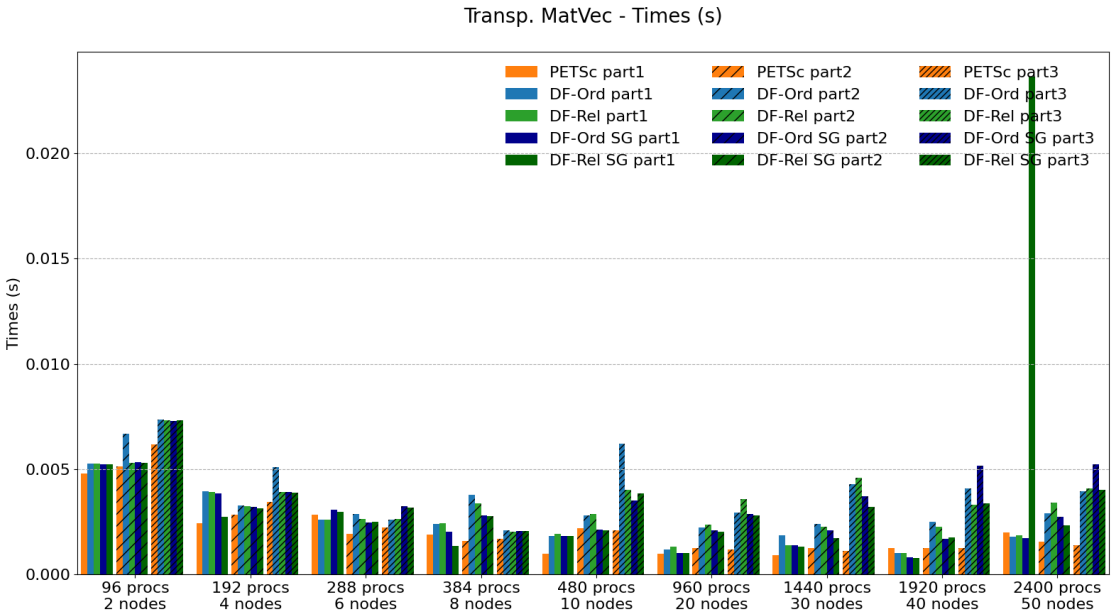


Tabela B.20: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

Tabela B.21: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
96	2	0.0048	0.0053	0.0053	0.0052	0.0052	0.9	0.9	0.9	0.9
192	4	0.0024	0.0039	0.0039	0.0039	0.0028	0.6	0.6	0.6	0.9
288	6	0.0028	0.0026	0.0026	0.0031	0.0030	1.1	1.1	0.9	0.9
384	8	0.0019	0.0024	0.0024	0.0020	0.0014	0.8	0.8	0.9	1.4
480	10	0.0010	0.0018	0.0019	0.0018	0.0018	0.5	0.5	0.5	0.5
960	20	0.0010	0.0012	0.0013	0.0010	0.0010	0.8	0.7	1.0	1.0
1440	30	0.0009	0.0019	0.0014	0.0014	0.0013	0.5	0.6	0.7	0.7
1920	40	0.0013	0.0010	0.0010	0.0008	0.0008	1.2	1.2	1.6	1.6
2400	50	0.0020	0.0018	0.0019	0.0017	0.0237	1.1	1.1	1.1	0.1

Tabela B.22: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
96	2	0.0051	0.0067	0.0053	0.0053	0.0053	0.8	1.0	1.0	1.0
192	4	0.0028	0.0033	0.0032	0.0032	0.0032	0.9	0.9	0.9	0.9
288	6	0.0019	0.0029	0.0026	0.0025	0.0025	0.7	0.7	0.8	0.8
384	8	0.0016	0.0038	0.0034	0.0028	0.0028	0.4	0.5	0.6	0.6
480	10	0.0022	0.0028	0.0029	0.0021	0.0021	0.8	0.8	1.0	1.0
960	20	0.0013	0.0022	0.0024	0.0021	0.0020	0.6	0.5	0.6	0.6
1440	30	0.0012	0.0024	0.0023	0.0021	0.0017	0.5	0.5	0.6	0.7
1920	40	0.0013	0.0025	0.0023	0.0017	0.0017	0.5	0.6	0.7	0.7
2400	50	0.0015	0.0029	0.0034	0.0027	0.0023	0.5	0.5	0.6	0.7

Tabela B.23: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
96	2	0.0062	0.0074	0.0073	0.0073	0.0073	0.8	0.8	0.8	0.8
192	4	0.0034	0.0051	0.0039	0.0039	0.0039	0.7	0.9	0.9	0.9
288	6	0.0022	0.0026	0.0026	0.0033	0.0032	0.8	0.8	0.7	0.7
384	8	0.0017	0.0021	0.0020	0.0021	0.0021	0.8	0.8	0.8	0.8
480	10	0.0021	0.0062	0.0040	0.0035	0.0038	0.3	0.5	0.6	0.5
960	20	0.0012	0.0030	0.0036	0.0029	0.0028	0.4	0.3	0.4	0.4
1440	30	0.0011	0.0043	0.0046	0.0037	0.0032	0.3	0.2	0.3	0.4
1920	40	0.0012	0.0041	0.0033	0.0052	0.0034	0.3	0.4	0.2	0.4
2400	50	0.0014	0.0040	0.0041	0.0052	0.0040	0.3	0.3	0.3	0.3

Matriz de ordem 10 milhões (bloco 3x3), até 50 nós

Figura B.12: Gráficos de tempos da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

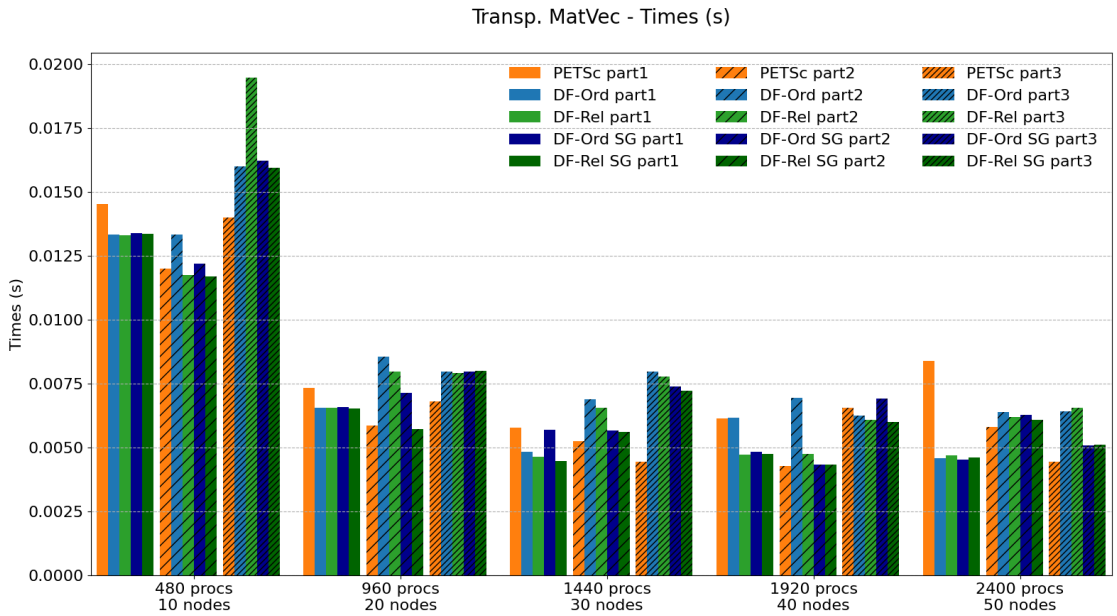


Tabela B.24: Tempos e Speed-up da operação Transp. MatVec com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

Tabela B.25: Grid partitioned in 1 axis

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
480	10	0.0145	0.0133	0.0133	0.0134	0.0134	1.1	1.1	1.1	1.1
960	20	0.0073	0.0066	0.0066	0.0066	0.0065	1.1	1.1	1.1	1.1
1440	30	0.0058	0.0048	0.0046	0.0057	0.0045	1.2	1.2	1.0	1.3
1920	40	0.0061	0.0062	0.0047	0.0048	0.0047	1.0	1.3	1.3	1.3
2400	50	0.0084	0.0046	0.0047	0.0045	0.0046	1.8	1.8	1.9	1.8

Tabela B.26: Grid partitioned in 2 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
480	10	0.0120	0.0133	0.0117	0.0122	0.0117	0.9	1.0	1.0	1.0
960	20	0.0059	0.0086	0.0080	0.0071	0.0057	0.7	0.7	0.8	1.0
1440	30	0.0052	0.0069	0.0066	0.0057	0.0056	0.8	0.8	0.9	0.9
1920	40	0.0043	0.0069	0.0047	0.0043	0.0043	0.6	0.9	1.0	1.0
2400	50	0.0058	0.0064	0.0062	0.0063	0.0061	0.9	0.9	0.9	1.0

Tabela B.27: Grid partitioned in 3 axes

Partitioning		Time (s)					Speed-up			
#procs	#nós	PETSc	DF-Ord	DF-Rel	DF-Ord SG	DF-Rel SG	PETSc / DF-Ord	PETSc / DF-Rel	PETSc / DF-Ord SG	PETSc / DF-Rel SG
480	10	0.0140	0.0160	0.0195	0.0162	0.0159	0.9	0.7	0.9	0.9
960	20	0.0068	0.0080	0.0079	0.0080	0.0080	0.9	0.9	0.9	0.9
1440	30	0.0045	0.0080	0.0078	0.0074	0.0072	0.6	0.6	0.6	0.6
1920	40	0.0066	0.0063	0.0061	0.0069	0.0060	1.0	1.1	0.9	1.1
2400	50	0.0044	0.0064	0.0066	0.0051	0.0051	0.7	0.7	0.9	0.9

B.2.3 Operação $C = P^T AP$

Matriz de ordem 1 milhão (bloco 3x3), até 50 nós

Figura B.13: Gráficos de tempos da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

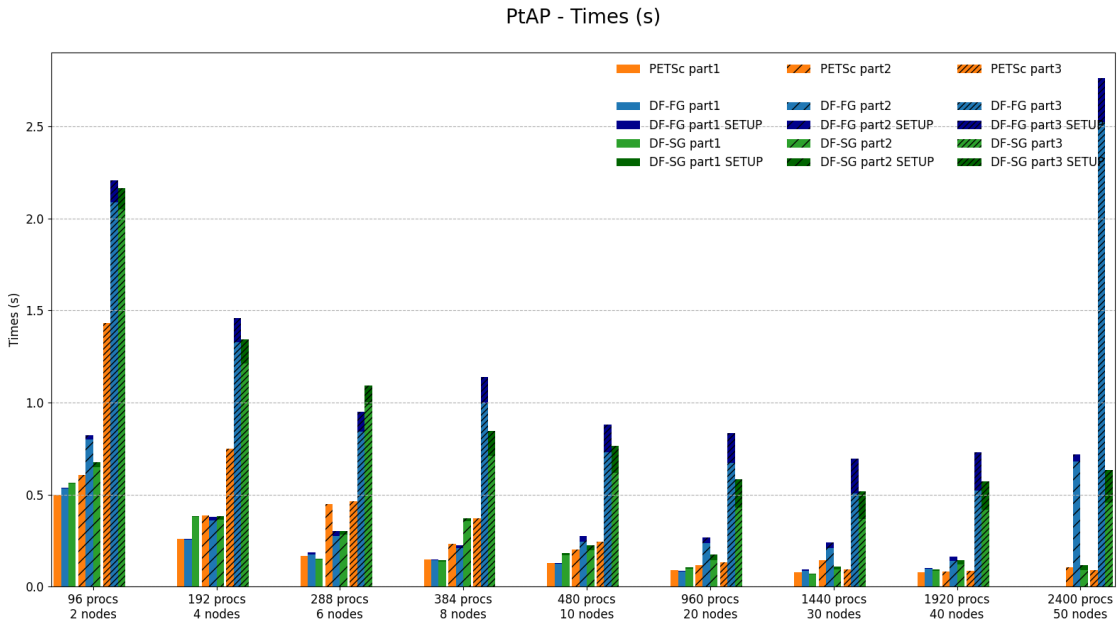


Tabela B.28: Tempos e Speed-up da operação PtAP com matriz de ordem 1 milhão (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
96	2	0.4987	0.5377	0.5641	0.9	0.9
192	4	0.2599	0.2611	0.3834	1.0	0.7
288	6	0.1690	0.1850	0.1522	0.9	1.1
384	8	0.1477	0.1497	0.1427	1.0	1.0
480	10	0.1282	0.1310	0.1822	1.0	0.7
960	20	0.0902	0.0885	0.1053	1.0	0.9
1440	30	0.0790	0.0930	0.0699	0.8	1.1
1920	40	0.0786	0.1038	0.0931	0.8	0.8

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
96	2	0.6078	0.8226	0.6753	0.7	0.9
192	4	0.3855	0.3794	0.3840	1.0	1.0
288	6	0.4498	0.3012	0.3010	1.5	1.5
384	8	0.2318	0.2267	0.3713	1.0	0.6
480	10	0.2037	0.2759	0.2245	0.7	0.9
960	20	0.1177	0.2697	0.1764	0.4	0.7
1440	30	0.1433	0.2421	0.1097	0.6	1.3
1920	40	0.0816	0.1646	0.1451	0.5	0.6
2400	50	0.1061	0.7195	0.1166	0.1	0.9

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	PETSc / DF-FG	PETSc / DF-SG
96	2	1.4305	2.2052	2.1656	0.6	0.7
192	4	0.7504	1.4602	1.3434	0.5	0.6
288	6	0.4653	0.9488	1.0934	0.5	0.4
384	8	0.3710	1.1397	0.8450	0.3	0.4
480	10	0.2434	0.8798	0.7664	0.3	0.3
960	20	0.1336	0.8358	0.5824	0.2	0.2
1440	30	0.0957	0.6968	0.5186	0.1	0.2
1920	40	0.0884	0.7289	0.5716	0.1	0.2
2400	50	0.0914	2.7631	0.6328	0.0	0.1

Matriz de ordem 10 milhões (bloco 3x3), até 50 nós

Figura B.14: Gráficos de tempos da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

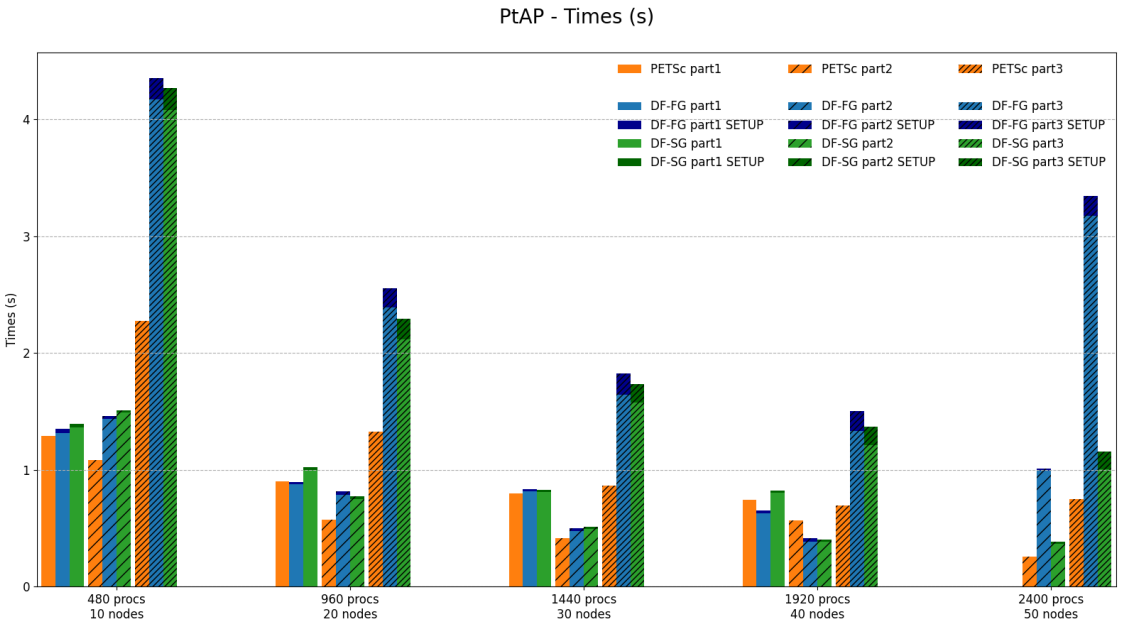


Tabela B.29: Tempos e Speed-up da operação PtAP com matriz de ordem 10 milhões (bloco 3x3) para os casos de extrapolação

(a) Grid partitioned in 1 axis

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
480	10	1.2898	1.3506	1.3978	1.0	0.9
960	20	0.9021	0.8993	1.0246	1.0	0.9
1440	30	0.7992	0.8382	0.8308	1.0	1.0
1920	40	0.7437	0.6506	0.8235	1.1	0.9

(b) Grid partitioned in 2 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
480	10	1.0869	1.4645	1.5117	0.7	0.7
960	20	0.5768	0.8186	0.7738	0.7	0.7
1440	30	0.4181	0.5011	0.5120	0.8	0.8
1920	40	0.5664	0.4183	0.4026	1.4	1.4
2400	50	0.2553	1.0109	0.3881	0.3	0.7

(c) Grid partitioned in 3 axes

		Time (s)			Speed-up	
#procs	#nós	PETSc	DF-FG	DF-SG	$\frac{\text{PETSc}}{\text{DF-FG}}$	$\frac{\text{PETSc}}{\text{DF-SG}}$
480	10	2.2733	4.3573	4.2685	0.5	0.5
960	20	1.3255	2.5577	2.2924	0.5	0.6
1440	30	0.8653	1.8291	1.7379	0.5	0.5
1920	40	0.6939	1.5037	1.3717	0.5	0.5
2400	50	0.7500	3.3459	1.1605	0.2	0.6