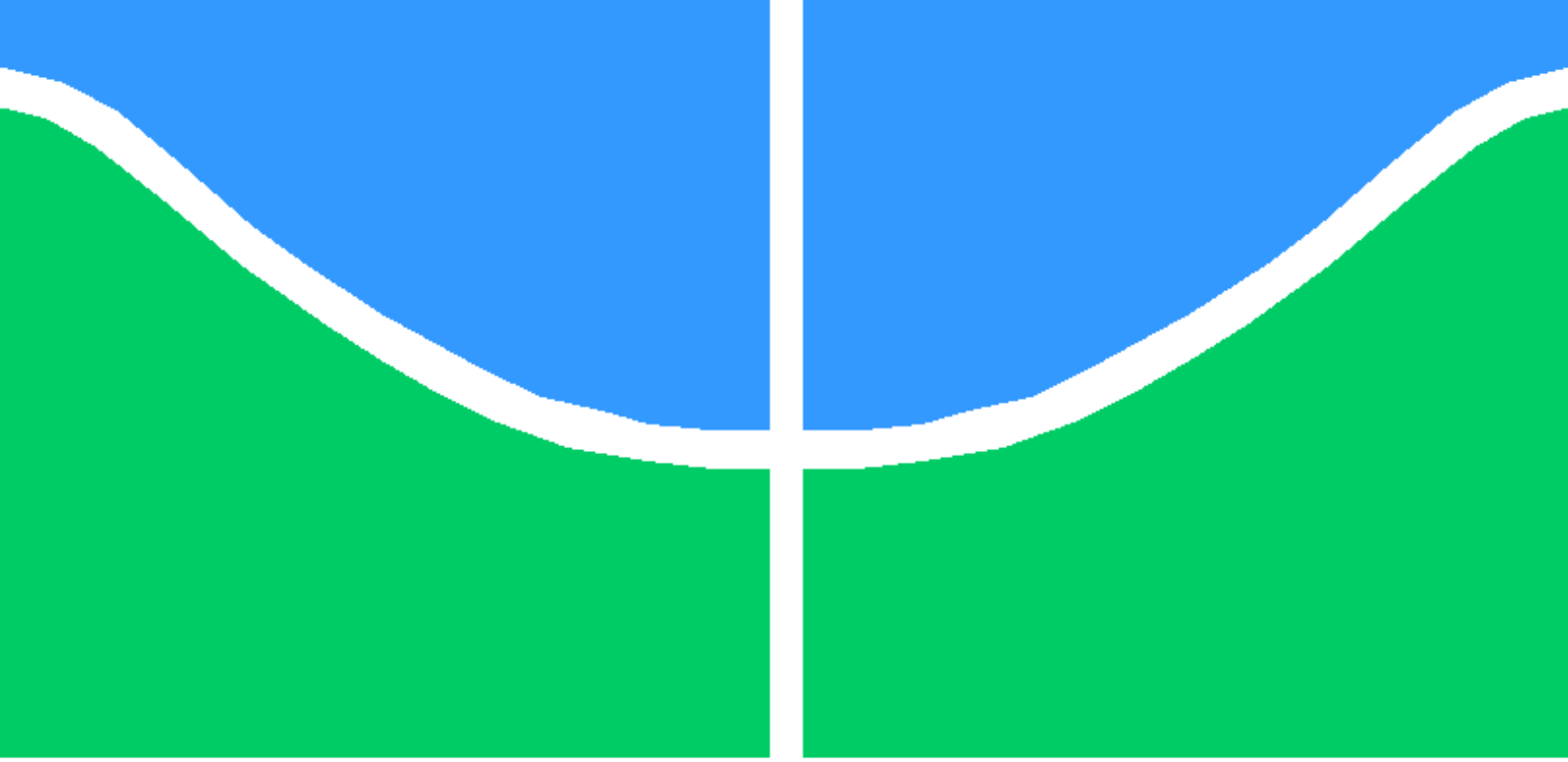


PROGRAMAÇÃO PARALELA PARA RESOLUÇÃO DA EQUAÇÃO DE POISSON COM O MÉTODO DE DIFERENÇAS FINITAS

CARLOS ADIR ELY MURUSSI LEITE

PROJETO DE GRADUAÇÃO EM ENGENHARIA MECÂNICA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

PROGRAMAÇÃO PARALELA PARA RESOLUÇÃO DA EQUAÇÃO DE POISSON COM O MÉTODO DE DIFERENÇAS FINITAS

CARLOS ADIR ELY MURUSSI LEITE

Orientador: PROF. DR. EDER LIMA DE ALBUQUERQUE, ENM/UNB

PROJETO DE GRADUAÇÃO EM ENGENHARIA MECÂNICA

PUBLICAÇÃO ENM.PG - XXX/AAAA
BRASÍLIA-DF, 25 DE SETEMBRO DE 2022.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

PROGRAMAÇÃO PARALELA PARA RESOLUÇÃO DA EQUAÇÃO DE POISSON COM O MÉTODO DE DIFERENÇAS FINITAS

CARLOS ADIR ELY MURUSSI LEITE

PROJETO DE GRADUAÇÃO SUBMETIDO AO DEPARTAMENTO DE ENGENHARIA MECÂNICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO MECÂNICO.

APROVADA POR:

Prof. Dr. Eder Lima de Albuquerque, ENM/UnB
Orientador

Prof. Dr. Taygoara Felamingo De Oliveira, ENM/UnB
Examinador interno

Prof. Dr. Marcela Rodrigues Machado, ENM/UnB
Examinador interno

BRASÍLIA, 25 DE SETEMBRO DE 2022.

FICHA CATALOGRÁFICA
CARLOS ADIR ELY MURUSSI LEITE
PROGRAMAÇÃO PARALELA PARA RESOLUÇÃO DA EQUAÇÃO DE POISSON COM O MÉTODO DE DIFERENÇAS FINITAS
2022xv, 147p., 201x297 mm
(ENM/FT/UnB, Engenheiro Mecânico, Engenharia Mecânica, 2022)
Projeto de Graduação - Universidade de Brasília
Faculdade de Tecnologia - Departamento de Engenharia Mecânica

REFERÊNCIA BIBLIOGRÁFICA

CARLOS ADIR ELY MURUSSI LEITE (2022) PROGRAMAÇÃO PARALELA PARA RESOLUÇÃO DA EQUAÇÃO DE POISSON COM O MÉTODO DE DIFERENÇAS FINITAS. Projeto de Graduação em Engenharia Mecânica, Publicação xxx/AAAA, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 147p.

CESSÃO DE DIREITOS

AUTOR: CARLOS ADIR ELY MURUSSI LEITE
TÍTULO: PROGRAMAÇÃO PARALELA PARA RESOLUÇÃO DA EQUAÇÃO DE POISSON COM O MÉTODO DE DIFERENÇAS FINITAS.
GRAU: Engenheiro Mecânico ANO: 2022

_____

CARLOS ADIR ELY MURUSSI LEITE
carlos.adir.leite@gmail.com

Abstract

Currently parallel programming is one of the alternatives to increase computational speed given the expected end of Moore's Law. High level libraries, like the PETSc used in this work, give tools which decrease the need for experienced parallel programmers to develop efficient algorithms. In this work we use PETSc along with C to implement a 2D and 3D numerical solution using Finite Difference Method for Poisson's equation in a structured mesh. Google Cloud was used to run the code up to 8 processors and meshs up to 125 million points. Then we analyse the results using the parallel metrics like Amdahl's Law or Karp-Flatt metric. The results for 2D code showed that its efficiency is near one. For the 3D case, the communication took about 20% of total time limiting the speed up. In both cases the superlinear phenomenon was observed for specific values of mesh's size.

Keywords: Parallel Programming, PETSc, Finite Difference Method, Poisson's Equation

Resumo

A programação paralela hoje é uma das alternativas usadas para aumentar a velocidade de cálculo em face do previsível fim da lei de Moore. Bibliotecas de alto nível, como o PETSc utilizado por este trabalho, fornece ferramentas que diminuem a necessidade de programadores paralelos experientes para desenvolver algoritmos eficientes. Neste trabalho, usamos o PETSc com a linguagem C para implementar a solução numérica 2D e 3D da Equação de Poisson usando o Método das Diferenças Finitas em uma malha estruturada. Foram efetuados cálculos utilizando máquinas do Google Cloud com até 8 processadores, com malhas de até 125 milhões de pontos e analisamos o código utilizando métricas como Lei de Amdahl e de Karp-Flatt. Os resultados 2D obtidos indicaram uma eficiência próxima de 1. Já para o 3D, a comunicação entre os processadores consumiu cerca de 20% do tempo total limitando a aceleração. Foi observado em ambos casos o fenômeno de aceleração superlinear para valores específicos de tamanho da malha.

Palavras chave: Programação Paralela, PETSc, Método de Diferenças Finitas, Equação de Poisson

LISTA DE FIGURAS

LISTA DE CÓDIGOS FONTE

LISTA DE TERMOS E SIGLAS

API	Application Programming Interface
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
EDP	Equação Diferencial Parcial
GPU	Graphics processing unit
HD	Hard Disck
KSP	Krylov subspace iterative method
MDF	Método das Diferenças Finitas
MEC	Método dos Elementos de Contorno
MEF	Método dos Elementos Finitas
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
PETSc	Portable, Extensible Toolkit for Scientific Computation
RAM	Random Acess Memory
SSD	Solid-State Drive
SSH	Secure Socket Shell
vCPU	Virtual Central Processing Unit

SUMÁRIO

Capítulo 1

Introdução

1.1 A evolução dos computadores

Computadores digitais começaram a ser desenvolvidos por volta de 1970 e rapidamente se tornaram essenciais para a ciência. O uso de computadores em aplicações científicas trouxe um novo ramo chamado computação científica que começou a desenvolver e validar modelos numéricos, bem como realizar simulações. O avanço da computação científica ficou intimamente ligada ao desenvolvimento e manufatura de processadores: com processadores mais potentes era possível aumentar o volume de cálculo por segundo. Gordon Moore fez uma previsão, revisada em 1975, de que a quantidade de transistores dobraria a cada 2 anos mantendo o custo constante[Wikipedia contributors 2022a]. Tal previsão foi conhecida como lei de Moore e tem se mantido até então.

Contudo, os artifícios utilizados para aumentar o poder de processamento até a última década atualmente não são mais viáveis. Hoje os transistores são feitos com dezenas de átomos e têm tamanhos de nanômetros [Wikipedia contributors 2022d] o que torna bem difícil ainda mais sua redução do tamanho. Além disso, o aumento na frequência implica em uma maior geração de calor, que é proporcional ao cubo da frequência, e a diminuição dos transistores implica em uma maior fuga de corrente [MakeTechEasier Alexander Fox 2018] o que leva em um maior consumo de energia e maior geração de calor.

Para manter a evolução da computação científica, procurou-se outras maneiras de aumentar o poder de processamento que não dependessem da arquitetura de computadores digitais. Alternativas como computadores quânticos ou o retorno de computadores analógicos ainda não trazem resultados significativos e seguem outro ramo de estudo [Washington University in St. Louis 2021].

Desta forma, a computação paralela aparece como uma solução adequada para continuar tal crescimento. O princípio parte do dividir para conquistar: Quebrar uma grande tarefa em pedaços menores, resolvê-las simultaneamente e reduzir o tempo total. Esta solução busca adaptar os processadores atuais para realizarem tarefas simultaneamente. Atualmente exis-

tem alguns compiladores [Intel 2018], linguagens funcionais [Wikipedia contributors 2022c] ou bibliotecas que usam paralelismo implícito e tentam paralelizar um código serial utilizando conexões da sintaxe do código, mas tais formas possuem uma série de restrições como paralelizar apenas *loops* não interconectados. Estas restrições levam ao uso do paralelismo explicito e requer desenvolvedores hábeis de programação paralela para o desenvolvimento dos algoritmos eficientes. Os principais problemas que ocorrem no paralelismo explicito referem-se à sincronização dos processadores, comunicação e divisão de tarefas entre processadores [Wikipedia contributors 2017], sendo praticamente necessário rescrever completamente um código serial em paralelo, visto que estes problemas citados não ocorrem na programação sequencial.

Com o desenvolvimento da programação paralela, a comunidade criou interfaces de programação de aplicações (APIs), tais como o OpenMP e o MPI, para facilitar o manuseio do fluxo de informações entre processadores [Quinn 2004] além de aumentar a portabilidade. Logo depois surgiram mais ferramentas, como o PETSc que utilizaremos neste trabalho, que utilizam estas APIs e que facilitaram ainda mais a escrita de códigos paralelos.

O PETSc (*Portable, Extensible Toolkit for Scientific Computation*) pode ser entendido como uma biblioteca *opensource* feita exclusivamente para aplicação científica modeladas por EDPs, que auxilia a separação da implementação do código da modelagem científica. Ele contém diversas estruturas de dados, sub-rotinas, *solvers* (lineares ou não) e álgebra linear paralela que servem como blocos para construir códigos de larga escala em computadores paralelos (ou seriais) [PETSc website 2022]. Os códigos do PETSc são escritos em Fortran, C, C++ e suportam MPI, CUDA e outras bibliotecas paralelas. Diversas bibliotecas *open-source*, como o FEniCSx, Firedrake, OpenFOAM, utilizam o PETSc como base e ainda é possível acoplar outros módulos de otimização como o TAO (*Toolkit for Advanced Optimization*).

1.2 O Método de Diferenças Finitas e a Equação de Poisson

As Equações Diferenciais Parciais (EDPs) modelam matematicamente uma infinidade de problemas. Em particular, a equação de Poisson (1.1) modela diversos fenômenos físicos, tais como a distribuição de temperatura em um objeto, o potencial elétrico ou gravitacional no espaço, ou o campo de pressão de um escoamento incompressível e o empenamento de seção transversal de barras sob torção. Esta EDP é linear elíptica e sua solução é bem-comportada e estável [Wikipedia contributors 2022b]. Para atender os objetivos deste trabalho, esta é a EDP que iremos tratar aqui:

$$\nabla^2 u + g = 0 \qquad \text{em } \Omega \tag{1.1}$$

e por condições de contorno, onde u é uma variável potencial (temperatura, deslocamento

transversal, carga elétrica, etc.) e g é uma função. Tanto u quanto g são definidos sobre um domínio Ω de contorno Γ.

Em domínios de geometria complexa, a solução analítica de (1.1) não é simples de se obter e soluções numéricas se tornaram atrativas. Existem diversos métodos numéricos para resolução de EDPs tais como MDF (Método de Diferenças Finitas), MEF (Método dos Elementos Finitos) ou MEC (Método dos Elementos de Contorno). Embora os conceitos de programação paralela possam ser utilizados para todos estes métodos, abordaremos apenas o método de diferenças finitas.

O Método de Diferenças Finitas (MDF) consiste em discretizar o domínio Ω em diversos pontos e reescrever a EDP em uma equação algébrica para cada ponto, tomando a estimativa das derivadas como diferenças dos valores nos pontos. Resolvendo o sistema de equações algébricas se obtém uma solução aproximada da EDP. A escolha deste método se deve à maior facilidade implementação de código e mais fácil compreensão do método.

1.3 Motivação

Visto que a velocidade de cálculos por segundo dos computadores não irá mais aumentar, ao menos enquanto não houver uma revolução tecnológica, a única solução viável para continuar melhorando a performance da computação de alto desempenho é o aumento do número de processos rodando em paralelo. Entretanto, de nada vale aumentar o número de processos se os códigos não forem implementados seguindo os paradigmas da programação paralela. Considerando que o ensino de programação paralela em cursos de matemática, física e engenharia ainda é bastante incipiente, a principal motivação deste trabalho de graduação é produzir um documento com diretrizes básicas que informe ao leitor uma forma de transformar seus códigos sequenciais em códigos paralelos capazes de tirar proveito tanto dos computadores pessoais, disponíveis para a maior parte da população quanto de cluster que, cada vez mais, encontram-se disponíveis em diversas universidades e laboratórios e também para locação em ambientes de nuvens.

1.4 Objetivos

Partindo então da necessidade de programar em paralelo visando um aumento de velocidade de cálculo e o aumento do tamanho dos problemas que podem ser resolvidos, este trabalho tem como objetivo a implementação de um código paralelo para a resolução de uma EDP, particularmente da equação de Poisson, usando a bibliotca PETSc. Foram considerados problemas bi e tri dimensionais. Por simplicidade, o método numérico escolhido foi o método das diferenças finitas. Como objetivo secundário, mas não menos importante, implementar uma rotina de testes para a análise de métricas para medir a aceleração e a eficiência

do código e estudar a variação destas métricas com o número de processadores e com o número de graus de liberdade.

1.5 Organização

O trabalho está organizado como segue. O capítulo 2 apresenta o básico sobre a programação paralela, sobre dispositivos (*hardwares*) utilizados e as métricas utilizadas para avaliar um código em paralelo. O capítulo 3 apresenta o MDF aplicado na equação de Poisson e a formulação matemática das matrizes e vetores que foram implementadas em código. O capítulo 4 apresenta a implementação do código MDF usando a linguagem C com o PETSc e explicita as estruturas de dados utilizadas pelo PETSc como na montagem da matriz e resolução do sistema linear. O capítulo 5 apresenta os resultados obtidos pelos códigos bi e tridimensionais feitos, para diferentes valores de malha e número de processadores e análises são feitas a partir destes resultados. Por fim, o capítulo 6 apresenta as conclusões e propostas de trabalhos futuros.

Capítulo 2

Computação paralela

2.1 Principal conceito de programação paralela

A computação paralela surgiu com o objetivo de reduzir o tempo de cálculo utilizando mais de um processador. O principal conceito é **dividir para conquistar**: divide-se o problema em diversas partes e cada um dos processadores resolve simultaneamente uma destas partes.

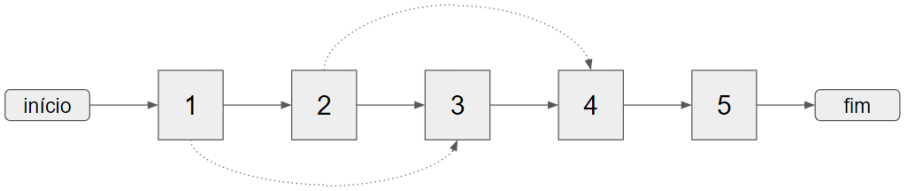


Figura 2.1: Exemplo de um processo sequencial de 5 tarefas

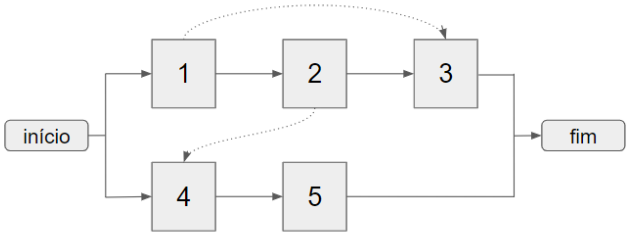


Figura 2.2: Exemplo de processo de cinco etapas divididos para dois processadores

Como exemplo, na Figura (2.1), um processo de cinco tarefas feito por apenas um processador, mas que pode ser dividido em dois processos como mostra a Figura (2.2). Idealmente, com p processadores, espera-se reduzir o tempo em p vezes. Contudo, quando há desbalanceamento de tarefas, processadores ficam ociosos: como exemplo, na Figura (2.2), o segundo processador fica ocioso durante a tarefa 3.

Outro problema que aparece é devido à dependência de informação entre as tarefas. As linhas pontilhadas na Figura (2.1) representam estas dependências: A tarefa 3 só inicia se o 1 tiver terminado, assim como o 4 se inicia somente após o 2. Desta forma consegue-se reestruturar a Figura (2.2) em uma melhor maneira como mostrado na Figura (2.3)

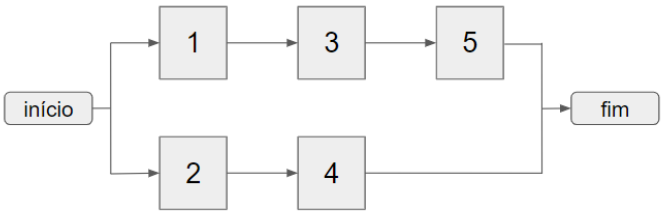


Figura 2.3: Reorganização das tarefas levando em conta as dependências mostradas (pontilhadas) na Figura (2.1)

Assim, os principais problemas que ocorrem na programação paralela referem-se à

- Sincronização dos processadores: Por exemplo, normalizar um vetor requer calcular o quadrado de cada componente (paralelizável), calcular a raiz da soma dos quadrados (serial) e dividir todo o vetor por essa raiz (paralelizável). No momento de se calcular a raiz, é necessário que todos os processos tenham terminado, e aquele que terminou mais rapidamente fica ocioso.
- Comunicação: No exemplo de normalizar um vetor, após um processo calcular a raiz, é necessário levar essa informação a todos os outros processos para que dividam sua componente por essa raiz.
- Divisão de tarefas: Como mostrado no exemplo da Figura (2.3), depende da estrutura do algoritmo.

Desenvolver um código em paralelo a partir de um sequencial envolve uma reestruturação do código, que depende do objetivo desejado e de qual região se deseja paralelizar. Às vezes é interessante paralelizar apenas a região que consome mais tempo, sendo insignificante o efeito de paralelizar partes que são naturalmente rápidas sequencialmente.

2.2 Dispositivos físicos para programação paralela

O exemplo mais comum de dispositivo (*hardware*) que realiza tarefas paralelamente é a GPU, que possui milhares de núcleos de processamento que trabalham em MHz (10^6 operações por segundo) cada. A CPU por sua vez, tem poucos ou apenas um núcleo que trabalham em GHz (10^9 operações por segundo). Como mostra na Figura (2.4), pode-se assimilar a GPU à uma equipe com várias pessoas enquanto a CPU é apenas uma pessoa que trabalha de forma bem rápida.

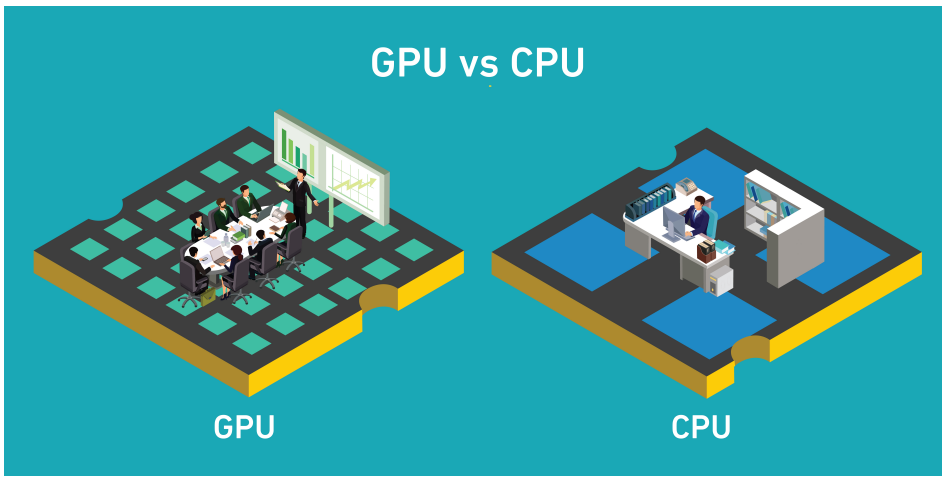


Figura 2.4: Esquema de representação de uma GPU e uma CPU. Enquanto a GPU tem várias unidades de processamento (representado por pessoas), a CPU tem algumas ou apenas uma unidade

Fonte: Ivey Business Review[Sankalp Hariharan; Remmy Martin Kilonzo 2018]

Outros dispositivos, que são mais adequados para computação paralela escalável, são os *clusters*: aglomerados de unidades de processamento [Pedro Cesar Tebaldi 2015]. Como exemplo, na Figura (2.5) tem-se um *cluster* feito com 8 unidades de *Raspberry Pi 3*. Cada unidade dessas possui 4 núcleos a 1.2 GHz, totalizando 32 núcleos de 1.2 GHz.

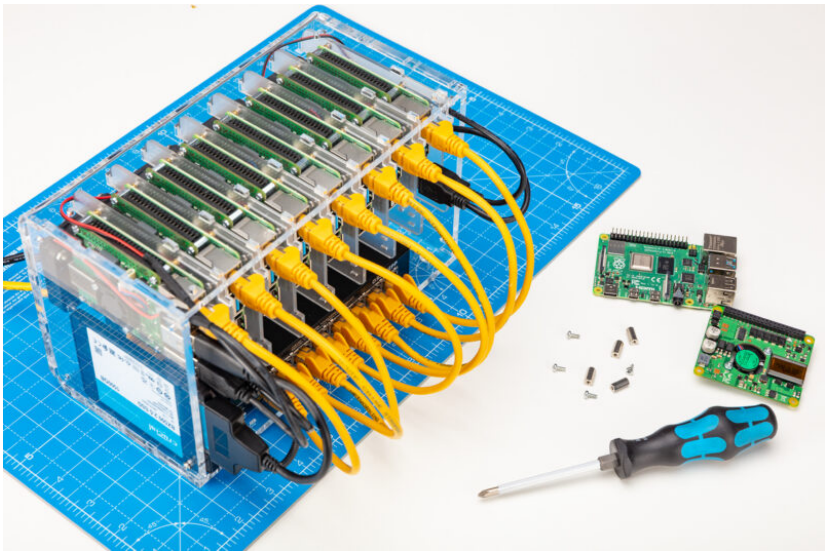


Figura 2.5: Exemplo de *cluster* feito com 8 unidades *Raspberry Pi 3*

Fonte: Raspberry Pi Site [Raspberry Pi 2022]

2.3 Tipos de memória

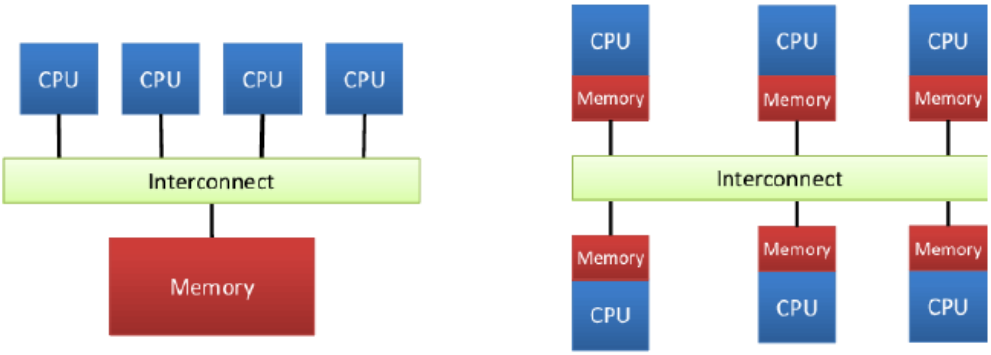
Na programação sequencial, com o avanço dos compiladores e das bibliotecas vetorizadas, os programadores não precisaram mais se preocupar sobre o que acontece a nível de memória para obter um código eficiente, pois o gerenciamento desta é feito internamente.

Contudo, quando se trata de programação paralela, os programadores ainda precisam se preocupar com a memória no momento de montar o algoritmo, principalmente para a escolha da biblioteca de comunicação que será utilizada.

As duas configurações mais comuns de hardware são de **memória compartilhada** (Figura 2.6a) e **memória distribuidade** (Figura 2.6b).

memória compartilhada: A mais comum em computadores pessoais. Os processadores compartilham a mesma memória e conseguem acessar informações de outros processos sem restrições.

memória distribuída: A mais comum em *clusters*. Há diversos blocos CPU + memória, em que cada CPU necessita enviar e receber mensagens com outros CPUs para acessar informações que não possuem.



(a) (b)

Figura 2.6: Esquema representando dois tipos diferentes de memória. (a) Memória compartilhada e (b) Memória distribuida.

Para a maioria das aplicações, a memória distribuída é preferível pois não há limite de endereçamento (o que ocorre na memória compartilhada) e consegue-se adicionar mais blocos de acordo com a necessidade de processamento ou memória. Além disso, um algoritmo que considera a memória distribuída e troca informações por mensagens tende a ser mais versátil (depende menos do *hardware*) e escalável (permite rodar problemas maiores toda vez que o número de processadores crescer).

É possível ter combinação de diversos blocos de memória compartilhada, mas geralmente a memória compartilhada é dividida artificialmente e se torna necessária a troca de informações por mensagens. O exemplo prático disto é o *cluster* feito com *Raspberry Pi 3* mostrado na Figura (2.5) em que no total tem-se 32 núcleos, mas são divididos em 8 blocos de memórias compartilhadas. Embora possa ser mais eficiente trocar informações entre uma unidade (como em um *Raspberry Pi 3*), o código precisaria se adaptar e se tornar menos versátil.

Em todo caso, como é objetivo fazer um código escalável, e como o PETSc não suporta o OpenMP, mesmo que a memória utilizada seja compartilhada, os algoritmos utilizados a consideram como distribuída.

2.4 APIs usadas para programação paralela

Quando os primeiros CPUs paralelos foram desenvolvidos, houve um esforço da comunidade de estabelecer padrões para aumentar a portabilidade de um código em diferentes dispositivos. Desta forma, foram criados APIs (*Application Program Interface*) como o OpenMP (*Open Multi-Processing*) e o MPI (*Message Passing Interface*).

OpenMP: Utilizado para memória compartilhada. A implementação tende a ser mais simples. Ele consiste em dividir o problema em partes como mostra a Figura (2.7), e há livre acesso de informação entre processos.

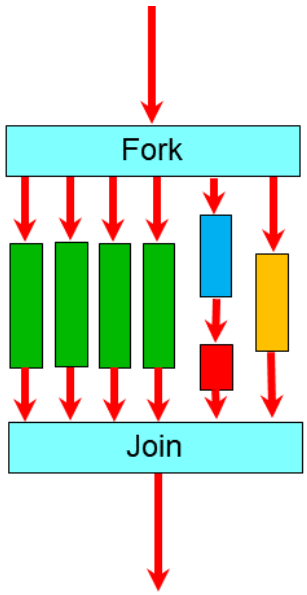


Figura 2.7: Representação de divisão de tarefas feitas pelo OpenMP.

MPI: Utilizado para memória distribuída. É realizada a transferência de mensagem entre os processos. Bibliotecas são OpenMPI (novo) e MPICH (mais versátil)

Embora tais padrões tenham sido estabelecidos e APIs tenham sido criadas, alguns dispositivos utilizam uma linguagem própria. Como exemplo tem-se as GPUs da NVidia que utilizam uma API chamada CUDA (*Compute Unified Device Architecture*) para otimizar o uso do próprio *hardware*.

Além das APIs, ferramentas e bibliotecas foram criadas utilizando tais APIs e protocolos de comunicação. O exemplo utilizado por este trabalho é o PETSc que utiliza o MPI como base para realizar a comunicação.

2.5 Métricas para análise de um código em paralelo

É esperado que com a computação paralela se tenha, ou uma redução do tempo de cálculo para um mesmo problema (tamanho do problema constante), ou um aumento do tamanho do problema resolvível no mesmo tempo (tempo constante). Para quantificar isto, definimos o tempo de cálculo $T(n,\ p)$ como função do tamanho do problema n e a quantidade de processadores p dado por (2.1).

$$T(n,p) = \sigma(n) + \frac{\varphi(n)}{p} + \kappa(n,\ p) \tag{2.1}$$

$$T_{serial} = T(n,\ 1) = \sigma(n) + \varphi(n) \tag{2.2}$$

O tempo de cálculo $T(n,\ p)$ pode ser decomposto em uma parte serial σ, uma parte paralela φ e uma componente κ que representa um fator adicional de tempo, como o tempo de comunicação entre processadores.

Como o tempo de cálculo varia de máquina para máquina, é de interesse saber apenas a razão entre os tempos. Define-se a aceleração (*speed up*) ψ e eficiência ε pelas Equações (2.3) e (2.4) para quantificar tais razões.

$$\psi(n,\ p) = \frac{T(n,1)}{T(n,p)} \tag{2.3}$$

$$\varepsilon(n,\ p) = \frac{1}{p} \cdot \psi(n,\ p) \tag{2.4}$$

É de interesse que o tempo paralelo seja o menor possível, ou seja, a aceleração ψ e a eficiência ε sejam as maiores possíveis. Pela característica construtiva, tem-se que

$$\varphi(n,\ p) \leq p \qquad\qquad \varepsilon \leq 1$$

Assim, criaram leis para avaliar e descrever um código paralelo, que exporemos aqui. Suas deduções partem do livro *Parallel Programming in C with MPI and OpenMP* [Quinn 2004].

2.5.1 Lei de Amdahl

A primeira lei tratada é a Lei de Amdahl que estima a aceleração máxima ψ a partir do tempo serial $T(n,1)$. Como $\kappa \geq 0$, por consequência de (2.3) se tem a equação (2.5)

$$\psi \leq \frac{\sigma + \varphi}{\sigma + \frac{1}{p}\varphi} \tag{2.5}$$

Sendo f a porção sequencial de um código serial (Equação 2.6), medida através do código com apenas um processador, obtemos a Lei de Ahmdal dada pela Equação (2.7).

$$f = \frac{\sigma(n)}{T(n,\ 1)} = \frac{\sigma}{\sigma + \varphi} \tag{2.6}$$

$$\psi \leq \frac{1}{f + \frac{1}{p}(1 - f)} \tag{2.7}$$

A Figura (2.8) mostra as curvas de nível da Equação (2.7) para alguns valores de porção sequencial f.

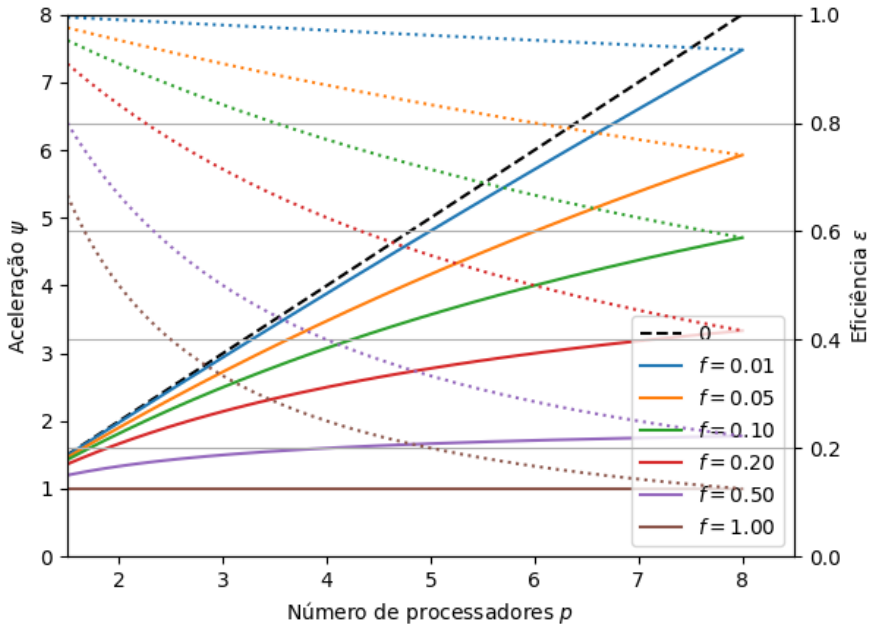


Figura 2.8: Curvas de nível teóricas de aceleração ψ (sólido) e eficiência ε (pontilhado) para diferentes valores de porção sequencial f e tamanho do problema n constante.

2.5.2 Lei de Gustafson-Barsis

A lei de Gustafson-Barsis parte do pressuposto de que não conhecemos $T(n,\ 1)$, pois frequentemente é muito alto, mas se interessa em obter uma estimativa da aceleração ψ a partir de $T(n,\ p)$. Assim, sendo s a porção sequencial medida de um código paralelo com p processadores (Equação 2.8), obtemos a Lei de Gustafson-Barsis dada pela Equação (2.9)

$$s = \frac{\sigma(n)}{T(n,\ p)} = \frac{\sigma}{\sigma + \frac{1}{p}\varphi} \tag{2.8}$$

$$\psi \leq p + (1 - p)s \tag{2.9}$$

2.5.3 Métrica de Karp-Flatt

Ao contrário da Lei de Amdahl (Equação 2.7) e da Gustafson-Barsis (Equação 2.9), que ignoram o termo κ, a métrica de Karp-Flatt tenta estimá-lo experimentalmente. Temos então a fração serial experimentalmente determinada e mostrada na Equação (2.10)

$$e = \frac{\sigma(n) + \kappa(n,\ p)}{T(n,\ 1)} = \frac{\sigma(n) + \kappa(n,\ p)}{\sigma(n) + \varphi(n)} \tag{2.10}$$

Fazendo manipulações algébricas de (2.10) obtém-se a Equação equivalente (2.11)

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.11}$$

Como $1 \leq \psi \leq p$ teremos $0 \leq e \leq 1$. No caso ideal de $\psi = p$ tem-se $e = 0$. Se e for constante, então obtemos que a porção de comunicação κ também é constante, não necessariamente zero.

2.5.4 Lei da isoeficiência

A lei de isoeficiência parte da análise de complexidade de um algoritmo e determina se é possível aumentar o tamanho n do problema, aumentando também a quantidade de processadores p sem comprometer a eficiência ε [Quinn 2004], como mostra a Equação (2.12).

$$\varepsilon(n_1,\ p_1) = \varepsilon(n_2,\ p_2) \tag{2.12}$$

Os principais problemas que ocorrem quando se aumenta o tamanho n do problema é a falta de memória para resolvê-lo. Frequentemente, a memória total M_d disponível é linearmente dependente da quantidade de processadores p. Nesta métrica, define-se uma função de isoeficiência $E(p)$ que depende da memória M_r requisitada pelo algoritmo para resolver o problema.

$$E(p) = \frac{M_r}{p}$$

O ideal é que o algoritmo tenha E constante e sempre menor que M_d/p, pois neste caso

é possível aumentar n indefinidamente sem comprometer ε. Caso não seja, depois que o limite de memória é atingido, não é possível mais manter ε constante. A Figura (2.9) mostra exemplos de funções E que, enquanto estiverem abaixo do limite de memória M_d/p pode-se aumentar n sem comprometer ε. Algoritmos cujas funções crescem mais lentamente em relação a p (tal como $c \cdot \log p$) tendem a ser mais escaláveis que algoritmos cujas funções crescem mais rapidamente.

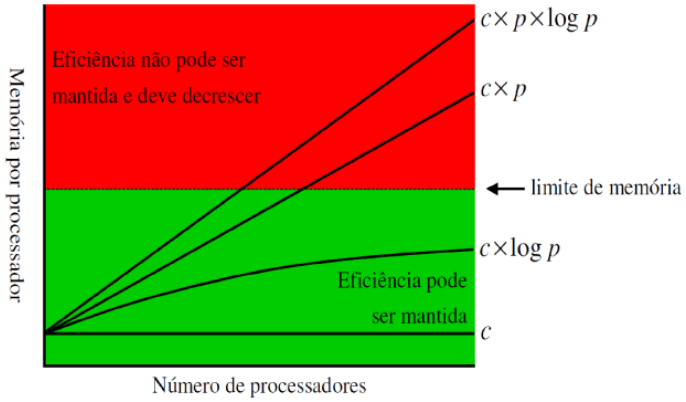


Figura 2.9: Representação de funções de isoeficiência. Para funções constantes a eficiência pode ser mantida. Ao atingir o limite de memória a eficiência cai.

Capítulo 3

Método de Diferenças Finitas

As equações diferenciais parciais (EDPs) são equações escritas na forma diferencial com mais de uma variável. Muitos problemas de engenharia são modelados utilizando EDPs. Uma boa parte destes problemas não são resolvíveis analiticamente e, desta forma, métodos numéricos foram desenvolvidos para obter soluções aproximadas. Um destes métodos, o qual utilizaremos, é o Método de de Diferenças Finitas (MDF) o qual transforma as equações diferenciais em equações algébricas fazendo estimativas das derivadas como diferenças entre os valores da função nos nós.

3.1 Estimativas das derivadas unidimensionais

Para o caso uni-dimensional, tal transformação é feita através da estimativa da derivada dada pela equação (3.1). A Figura (3.1) mostra uma linha de inclinação $f'(x_0)$ e outra próxima que é obtida pela diferença dos valores da função f ao redor de x_0. É possível reescrever a Equação (3.1) usando uma formulação matricial como mostra na Equação (3.2). A matriz que multiplica os valores da função nos pontos é chamada de *stencil*, que neste caso é um *stencil* de 3 pontos.

$$u^{(1)}(x) \approx \frac{u(x+h_x) - u(x-h_x)}{2h_x} \tag{3.1}$$

$$u^{(1)}(x_0) \approx \begin{bmatrix} \frac{-1}{2h_x} & 0 & \frac{1}{2h_x} \end{bmatrix} \cdot \begin{bmatrix} u(x_0 - h_x) \\ u(x_0) \\ u(x_0 + h_x) \end{bmatrix} \tag{3.2}$$

Do mesmo modo, pode-se obter uma estimativa da segunda derivada como mostra a Equação (3.3) e na formulação matricial (3.4) em que se tem o *stencil* de três pontos.

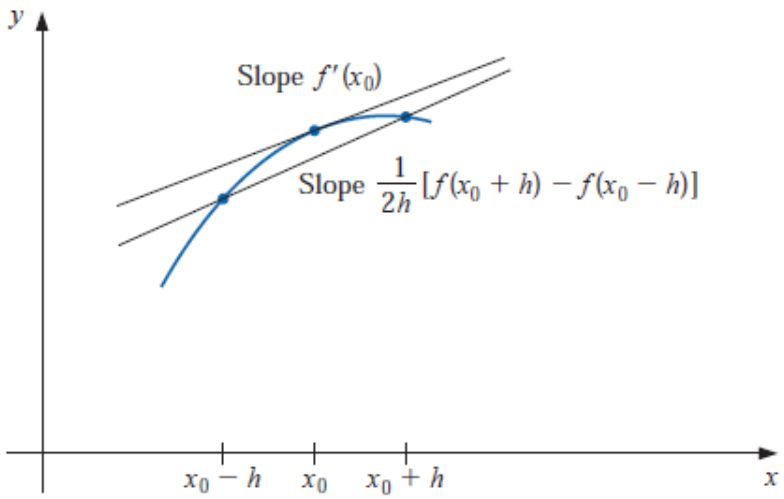


Figura 3.1: Representação geométrica da estimativa da derivada unidimensional de uma função f qualquer

Fonte: Burden and Faires, 1989 [Burden and Faires 1989]

$$u^{(2)}(x_0) \approx \frac{u(x_0+h_x) - 2u(x_0) + u(x_0-h_x)}{h_x^2} \tag{3.3}$$

$$u^{(2)}(x_0) = \begin{bmatrix} \frac{1}{h_x^2} & \frac{-2}{h_x^2} & \frac{1}{h_x^2} \end{bmatrix} \cdot \begin{bmatrix} u(x_0-h_x) \\ u(x_0) \\ u(x_0+h_x) \end{bmatrix} \tag{3.4}$$

3.2 Formulação MDF unidimensional para Poisson

A Equação Poisson unidimensional é dada pela Equação (3.5)

$$\frac{d^2u}{dx^2} + g = 0 \quad \forall\Omega \tag{3.5}$$

Sendo $\Omega = [0,\ 1]$ e fazendo a discretização uniforme com n pontos, conforme mostra a Figura (3.2), teremos:

$$x_0 = 0 \qquad x_{n-1} = 1 \qquad h_x = \frac{1}{n-1}$$
$$x_i = i \cdot h_x \qquad u_i \approx u(x_i)$$

Assim, a equação (3.5) se torna (3.6) para todos os nós em vermelho (Fig. 3.2), totalizando $n-2$ equações.

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h_x^2} + g(x_i) = 0 \qquad \forall i = 1,\ \cdots,\ n-2 \tag{3.6}$$

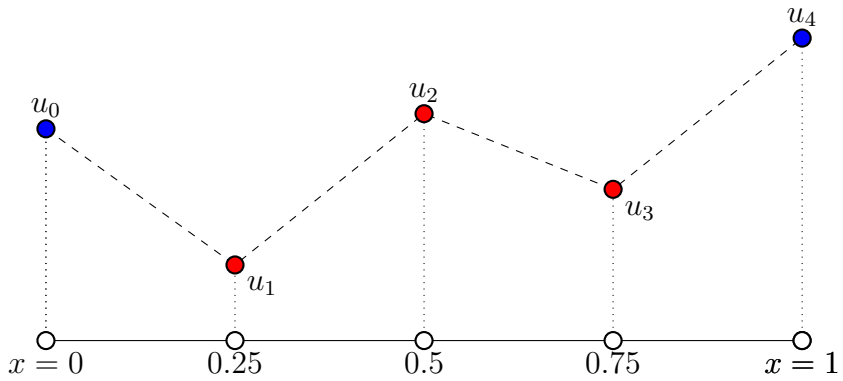


Figura 3.2: Exemplo de discretização 1D da malha com $n = 5$ pontos. Os pontos em azul representam aqueles conhecidos (devido à condição de contorno) e os pontos em vermelho aqueles desconhecidos onde se aplica a Equação (3.6)

Então montando o sistema linear a partir da equação (3.6), se obtém o sistema (3.7):

$$\frac{1}{h_x^2}\begin{bmatrix} -1 & 2 & -1 & & & & & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 2 & -1 & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & -1 & 2 & -1 & \\ & & & & & -1 & 2 & -1 \end{bmatrix}_{(n-2)\times n} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-3} \\ u_{n-2} \\ u_{n-1} \end{bmatrix}_{(n)} = \begin{bmatrix} g(x_1) \\ g(x_2) \\ g(x_3) \\ \vdots \\ g(x_{n-3}) \\ g(x_{n-2}) \end{bmatrix}_{(n-2)} \tag{3.7}$$

Multiplicando (3.7) por h_x^2 e adicionando as condições de contorno de $u_0 = 0$ e $u_{n-1} = 0$ se obtém (3.8):

$$\begin{bmatrix} 1 & & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & & 1 & \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} 0 \\ h_x^2 g(x_1) \\ \vdots \\ h_x^2 g(x_{n-2}) \\ 0 \end{bmatrix} \tag{3.8}$$

Resolvendo então o sistema (3.8), se encontra os valores de u_i para todos os nós, encontrando a solução aproximada desejada. Para o caso unidimensional com 5 pontos mostrados

na Figura (3.2), o sistema (3.8) é dado por (3.9)

$$\begin{bmatrix} 1 & & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & & 1 & \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.25^2 g(0.25) \\ 0.25^2 g(0.5) \\ 0.25^2 g(0.75) \\ 0 \end{bmatrix} \tag{3.9}$$

3.3 Formulação MDF bi e tri-dimensional para Poisson em malha estruturada

Para um problema bi e tridimensional, temos a mesma ideia que para o problema unidimensional, mas utilizamos as derivadas parciais. Como exemplo, as derivadas parciais em 2D são dadas por:

$$\frac{\partial u}{\partial x} \approx \frac{u(x+h_x,\ y) - u(x-h_x,\ y)}{2h_x} = \begin{bmatrix} \frac{-1}{2h_x} & 0 & \frac{1}{2h_x} \end{bmatrix} \begin{bmatrix} u_{j,i-1} \\ u_{j,i} \\ u_{j,i+1} \end{bmatrix}$$

$$\frac{\partial u}{\partial y} \approx \frac{u(x,\ y+h_y) - u(x,\ y-h_y)}{2h_y} = \begin{bmatrix} \frac{-1}{2h_y} & 0 & \frac{1}{2h_y} \end{bmatrix} \begin{bmatrix} u_{j-1,i} \\ u_{j,i} \\ u_{j+1,i} \end{bmatrix}$$

De mesmo modo, teremos as segundas derivadas dadas por:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+h_x,\ y) - 2u(x,\ y) + u(x-h_x,\ y)}{h_x^2} = \begin{bmatrix} \frac{1}{h_x^2} & \frac{-2}{h_x^2} & \frac{1}{h_x^2} \end{bmatrix} \begin{bmatrix} u_{j,i-1} \\ u_{k,j,i} \\ u_{j,i+1} \end{bmatrix}$$

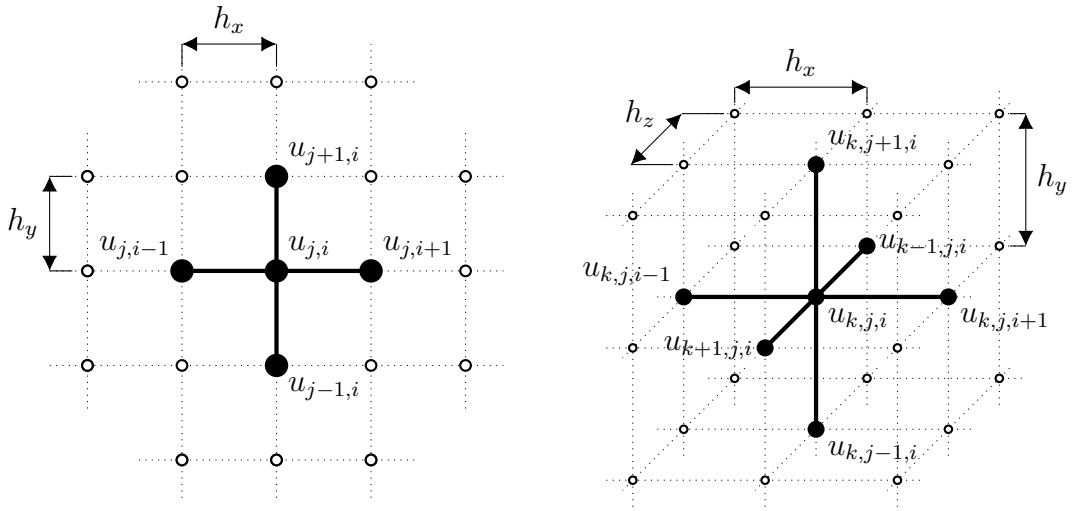
$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u(x,\ y+h_y) - 2u(x,\ y) + u(x,\ y-h_y)}{h_y^2} = \begin{bmatrix} \frac{1}{h_y^2} & \frac{-2}{h_y^2} & \frac{1}{h_y^2} \end{bmatrix} \begin{bmatrix} u_{j-1,i} \\ u_{j,i} \\ u_{j+1,i} \end{bmatrix}$$

$$\nabla^2 u \approx \begin{bmatrix} \frac{1}{h_x^2} & \frac{1}{h_y^2} & -2\left(\frac{1}{h_x^2} + \frac{1}{h_y^2}\right) & \frac{1}{h_y^2} & \frac{1}{h_x^2} \end{bmatrix} \begin{bmatrix} u_{j,i-1} \\ u_{j-1,i} \\ u_{j,i} \\ u_{j+1,i} \\ u_{j,i+1} \end{bmatrix} \tag{3.10}$$

Vemos, na equação (3.10), que o laplaciano de u no espaço bidimensional utiliza o *stencil* de 5 pontos, cuja representação geométrica é mostrada na Figura (3.3a). Para o caso tridi-

mensional, temos a adição da variável z que dá a equação (3.11) em que utiliza o *stencil* de 7 pontos representado na Figura (3.3b).

$$\nabla^2 u \approx \begin{bmatrix} \frac{1}{h_x^2} & \frac{1}{h_y^2} & \frac{1}{h_z^2} & -2\left(\frac{1}{h_x^2}+\frac{1}{h_y^2}+\frac{1}{h_z^2}\right) & \frac{1}{h_z^2} & \frac{1}{h_y^2} & \frac{1}{h_x^2} \end{bmatrix} \begin{bmatrix} u_{k,j,i-1} \\ u_{k,j-1,i} \\ u_{k-1,j,i} \\ u_{k,j,i} \\ u_{k+1,j,i} \\ u_{k,j+1,i} \\ u_{k,j,i+1} \end{bmatrix} \tag{3.11}$$



(a) (b)

Figura 3.3: Representação geométrica dos *stencils* para $\nabla^2 u$. (a) bidimensional *stencil* de 5 pontos e (b) tridimensional *stencil* de 7 pontos

Como a Equação de Poisson (3.12) possui o laplaciano $\nabla^2 u$, utiliza-se as aproximações mostradas nas Equações (3.10) e (3.11) na formulação do MDF bi e tri-dimensional (Equações 3.13 e 3.14). Os valores de $g_{j,i}$ e $g_{k,j,i}$ representam respectivamente $g(x_i, y_j)$ e $g(x_i, y_j, z_k)$.

$$\nabla^2 u + g = 0 \qquad \text{em } \Omega \tag{3.12}$$

$$\begin{bmatrix} \frac{-1}{h_x^2} & \frac{-1}{h_y^2} & 2\left(\frac{1}{h_x^2}+\frac{1}{h_y^2}\right) & \frac{-1}{h_y^2} & \frac{-1}{h_x^2} \end{bmatrix} \begin{bmatrix} u_{j,i-1} \\ u_{j-1,i} \\ u_{j,i} \\ u_{j+1,i} \\ u_{j,i+1} \end{bmatrix} = g_{j,i} \tag{3.13}$$

$$\begin{bmatrix} \frac{-1}{h_x^2} & \frac{-1}{h_y^2} & \frac{-1}{h_z^2} & 2\left(\frac{1}{h_x^2}+\frac{1}{h_y^2}+\frac{1}{h_z^2}\right) & \frac{-1}{h_z^2} & \frac{-1}{h_y^2} & \frac{-1}{h_x^2} \end{bmatrix} \begin{bmatrix} u_{k,j,i-1} \\ u_{k,j-1,i} \\ u_{k-1,j,i} \\ u_{k,j,i} \\ u_{k+1,j,i} \\ u_{k,j+1,i} \\ u_{k,j,i+1} \end{bmatrix} = g_{k,j,i} \qquad (3.14)$$

Sendo $\Omega = [0,\ 1] \times [0,\ 1]$ a malha estruturada bidimensional, temos o exemplo mostrado na Figura (3.4) em que se tem uma malha 5×5 com $h_x = h_y$. Então matrix 25×25 é preenchida como mostra a Figura (3.5a). Para matrizes maiores, segue-se o mesmo padrão, e como exemplo tem-se a matriz preenchida de uma malha 9×9 mostrada em (3.5b)

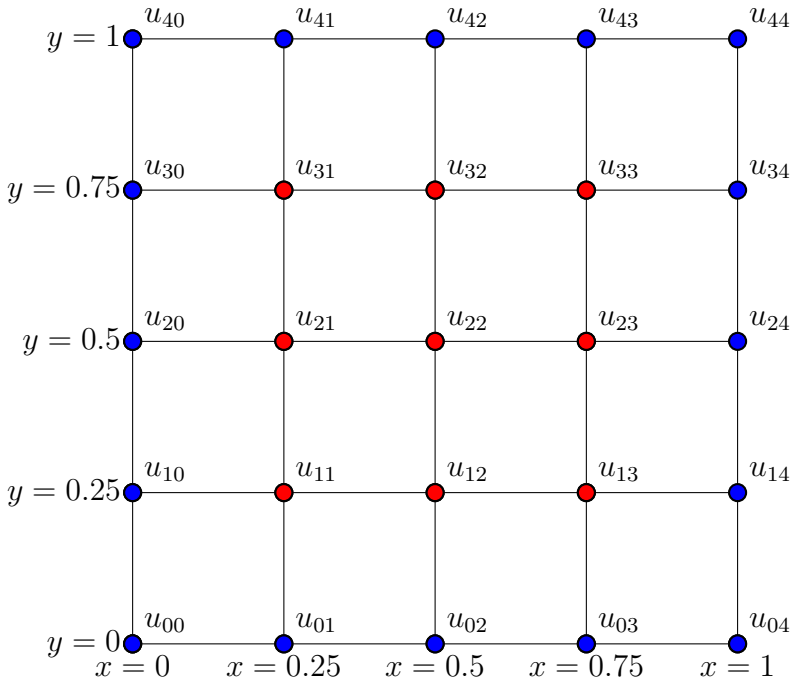
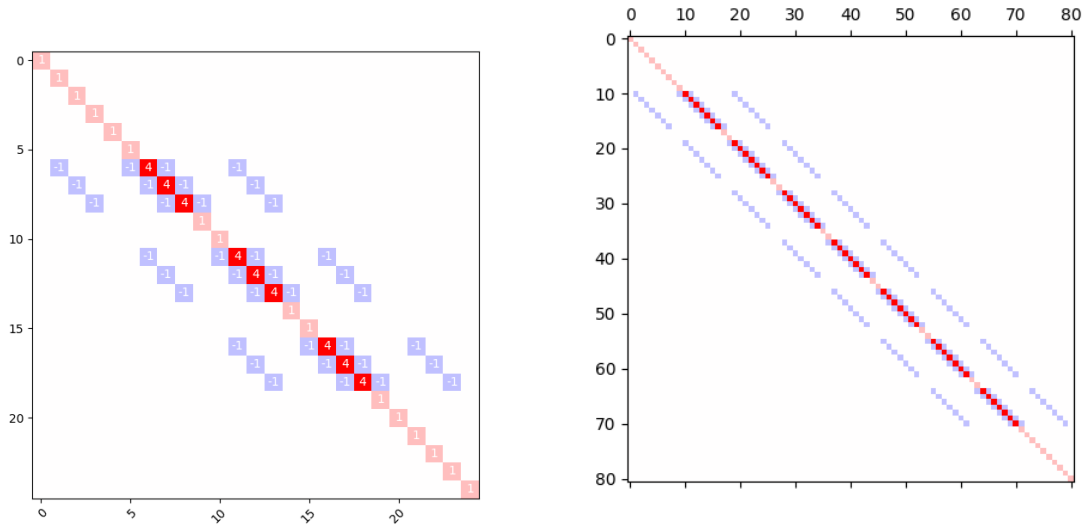


Figura 3.4: Discretização usando uma malha 5×5 no domínio $\Omega = [0,\ 1] \times [0,\ 1]$. Os pontos em azul representam os pontos conhecidos (devido às condições de contorno) e os pontos em vermelho aqueles que



(a) (b)

Figura 3.5: Matrix preenchida para o caso bidimensional para (a) $n = 5$ e (b) $n = 9$

3.4 Reformulação da matriz para métodos numéricos

Como resolveremos sistemas lineares numericamente, é interessante reescrever as matrizes de uma maneira matematicamente equivalente mas que permita reduzir o erro computacional. Os motivos são dados por:

- O número de condicionamento κ da matriz muda, influenciando na convergência e estabilidade do método. Exemplo, tem-se $\kappa(A) = 5000/3$ enquanto $\kappa(C) = 3$

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2000 & 1000 \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

- Como os métodos iterativos verificam convergência através do resíduo, certos elementos do vetor $\mathbf{u}$ possuem mais influência no cálculo de $\mathbf{r}$, tornando o erro de arredondamento bastante influente nos outros valores de $\mathbf{u}$, que têm menor influência $\mathbf{r}$.
- A divisão de números *float*s tem maior custo computacional. É preferível fazer a multiplicação de *float*s que realizar a divisão.

A solução utilizada é multiplicar cada equação pela área (no caso 2D) ou volume (no caso 3D) correspondente, resultando respectivamente nos *stencils* (3.15) e (3.16)

$$\begin{bmatrix} \dfrac{-h_y}{h_x} & \dfrac{-h_x}{h_y} & 2\left(\dfrac{h_y}{h_x} + \dfrac{h_x}{h_y}\right) & \dfrac{-h_x}{h_y} & \dfrac{-h_y}{h_x} \end{bmatrix} \tag{3.15}$$

$$\begin{bmatrix} \frac{-h_y h_z}{h_x} & \frac{-h_x h_z}{h_y} & \frac{-h_x h_y}{h_z} & 2\left(\frac{h_y h_z}{h_x} + \frac{h_x h_z}{h_y} + \frac{h_x h_y}{h_z}\right) & \frac{-h_x h_y}{h_z} & \frac{-h_x h_z}{h_y} & \frac{-h_y h_z}{h_x} \end{bmatrix} \tag{3.16}$$

Capítulo 4

Implementação de código MDF de Poisson em linguagem C usando a biblioteca PETSc

4.1 PETSc

O PETSc (*Portable, Extensible Toolkit for Scientific Computation*) é um conjunto de estruturas de dados e subrotinas feitas exclusivamente para computação científica. Ele é um *software* de código aberto (*opensource*) que pode ser implementado em Fortran, C, C++ e pode-se utilizar python para algumas aplicações. Ele contém uma série de pré-condicionadores, métodos KSP (como GMRES utilizado), integradores temporais, entre diversos outros conforme mostra a Figura (4.1).

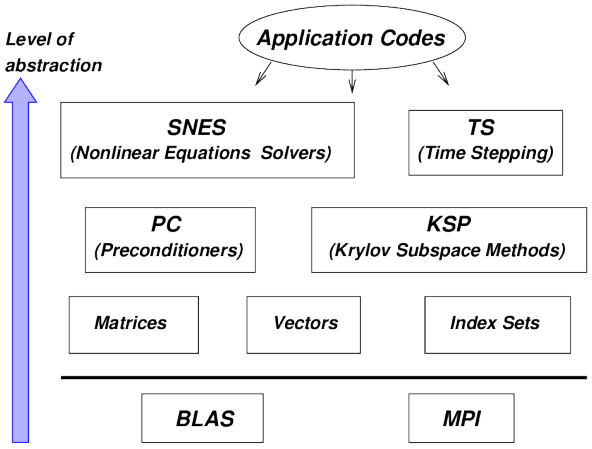


Figura 4.1: Estrutura hierárquica do PETSc com os principais componentes.
Fonte: Trabalho acadêmico do XVIII Congresso sobre métodos numéricos
[Castro et al. 2009]

4.1.1 Instalação, configuração e uso do PETSc

Na página do PETSc há um guia de instalação da ferramenta no computador. Contudo, mesmo seguindo os passos indicados, houveram problemas:

- Não há MPI único: É possível ter mais de um MPI instalado na mesma máquina, de forma que é necessário instalar e utilizar o MPI dentro do PETSc para rodar os códigos. É um adicional necessário a ser instalado e configurado. Para verificar se o MPI utilizado funciona, é necessário rodar um programa que imprime o `rank` de cada processador: Caso imprima diferentes valores (de 0 a $p-1$), está correto, mas se imprimir apenas 0, é necessário corrigir o MPI.
- Modo Debug: Por padrão a arquitetura que o PETSc utiliza é `arch-linux-c-debug`. Com esta tal opção ativada, os códigos paralelos podem ser mais lentos que os códigos seriais, o que foi observado no início do trabalho. A solução para isso foi mudar a arquitetura do PETSc para `arch-linux-c-opt`, pois ele compila a biblioteca sem certas verificações úteis apenas para implementação inicial do código.

Para fazer a instalação e configuração automática, foi criado um `makefile`. Ele faz o *download* do PETSc e o configura, baixando e instalando o MPI.

```
./configure -download-mpich -with-debugging=no -download-f2cblaslapack=1
```

Como o PETSc é uma biblioteca do ponto de vista da linguagem C, o procedimento de execução é: Compilar adicionando a *flag* do PETSc e executar usando o MPI.

- Para compilar usando a biblioteca do PETSc:

  ```
  -$CLINKER -o poisson2D_ksp poisson2D_ksp.o $PETSC_LIB
  ```

- Para executar usando uma malha com 500 elementos em x e 600 elementos em y, com apenas um processador:

  ```
  ./poisson2D_ksp -da_grid_x 500 -da_grid_y 600 -log_view
  ```

- Para executar usando o MPI com 7 processadores:

  ```
  \$ MPIEXEC -prepend-rank -np 7 ./mycode
  ```

- Enviar o *output* para o arquivo específico, no caso de um `.txt`:

  ```
  command | cat » poisson2D_ksp-p7-mesh500x600.txt
  ```

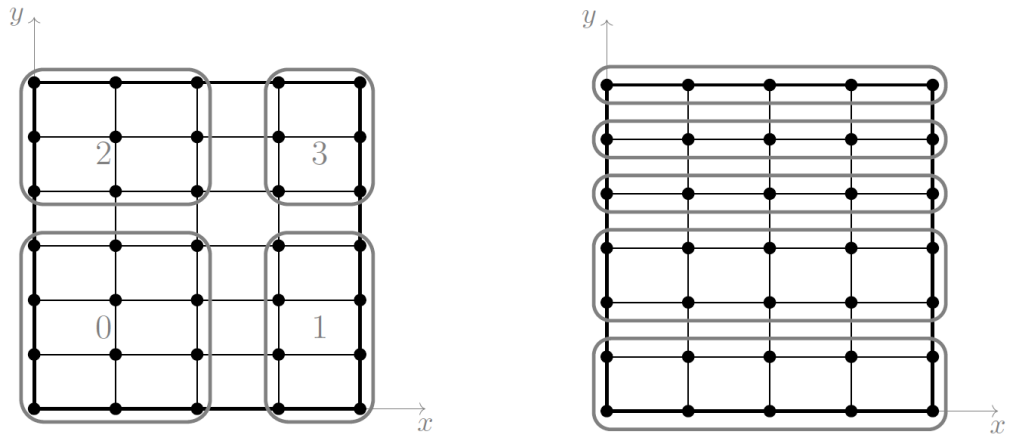
Então usou-se um `makefile` utilizando a linguagem *shellscript* para fazer a automatização das tarefas. Todos os códigos utilizados estão no repositório do GitHub `https://www.github.com/carlos-adir/ProjetoDeGraduacao`

4.1.2 Estruturas de dados utilizada pelo PETSc

As informações aqui contidas foram retiradas da página oficial do PETSc e do livro *PETSc for Partial Differential Equations* [Bueler 2020].

O PETSc oferece diversas ferramentas, como tipos de dados próprios (que permite controlar a precisão) ou estruturas mais complexas. As principais utilizadas por este trabalho são dadas por:

- Matrizes `Mat`. Por padrão elas são esparsas. Desta forma, se armazena apenas os números diferentes de zero e os índices referentes às posições que estes números estão.
- Vetores `Vec`. Do mesmo modo que para as matrizes.
- Malha estruturada `DM`: O PETSc já faz a divisão da malha de maneira automática para cada processo (`rank`), como mostra a Figura (4.2). Para reduzir o tempo total ocioso, é ideal que cada `rank` tenhas a mesma quantidade de nós.



(a) (b)

Figura 4.2: Esquema de divisão automática da malha estruturada feita pelo PETSc. No exemplo temos uma malha 5×5 dividida em (a) 4 grupos e (b) 5 grupos. Sendo um grupo por processo.

Outro fator importante na divisão da malha é referente aos nós fantasmas. Como o MDF utiliza *stencils*, os elementos da borda de um `rank` necessitam de informações dos nós de outro `rank`. O PETSc realiza cópias dos nós vizinhos (em cinza na Figura 4.3) para reduzir a dependência de comunicação. Vemos que o contorno gerado com 4 divisões (Figura 4.2a) é menor que aquele gerado com 5 (Fig. 4.2b). Para quantidades primas de processadores, não é possível realizar uma boa divisão, o que faz com que processos ocupem fileiras inteiras da malha e gerando uma grande quantidade de nós fantasmas.

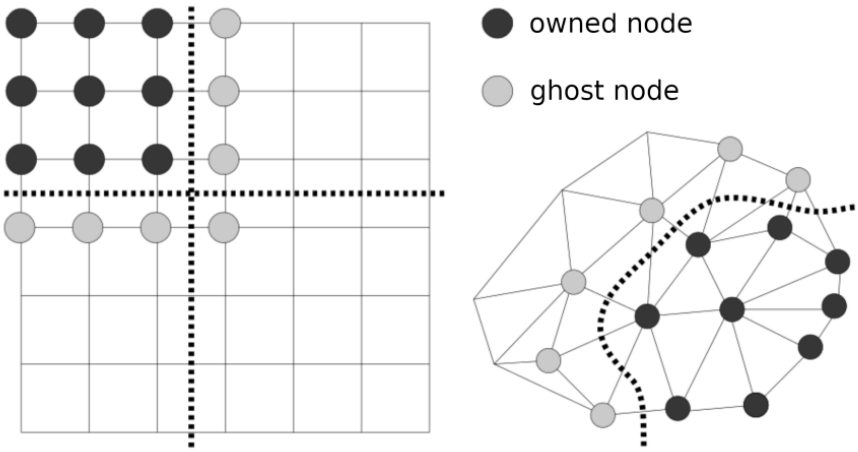


Figura 4.3: Duas formas de decomposição paralela em dois tipos diferentes de malha: estruturada à esquerda e não estruturada à direita. Há a distinção entre os nós do processo (escuros) e os nós duplicados de um outro processo (nós fantasmas)

- Krylov Spaces Methods `KSP` : Para resolução de sistemas lineares $\mathbf{Au} = \mathbf{b}$, o PETSc utiliza métodos iterativos com pré-condicionadores.

  Os pré-condicionadores melhoram o método iterativo tornando a convergência mais rápida. Idealmente seria preferível ter $\mathbf{A}^{-1}$ para resolver o sistema em apenas uma iteração, mas seu cálculo tem alto custo computacional, o que descarta esta possibilidade. Um dos pré-condicionadores utiliza uma matriz $\mathbf{M}$ de fácil inversão (P. ex. `jacobi` diagonal de $\mathbf{A}$) tal que $\mathbf{M}^{-1} \approx \mathbf{A}^{-1}$, e transformam o sistema linear em um sistema numericamente mais estável. Então o PETSc na verdade resolve o sistema dado por

  $$\mathbf{M}^{-1}\mathbf{Au} = \mathbf{M}^{-1}\mathbf{b}$$

  No caso de não utilização de pré-condicionadores, assume-se $\mathbf{M} = \mathbf{I}$.

  Então o PETSc resolve o sistema fazendo iterações usando o espaço de Krylov com a matriz (pré-condicionada ou não). Um polinômio $p_n(\mathbf{A})$ é utilizado pra aproximar $\mathbf{A}^{-1}$ e servir como matriz de iteração reduzindo o resíduo $\mathbf{r}$.

  $$p_n(\mathbf{A}) = c_0\mathbf{I} + c_1\mathbf{A} + c_2\mathbf{A}^2 + \cdots + c_{n-1}\mathbf{A}^{n-1} \approx \mathbf{A}^{-1}$$

  $$\mathbf{u}_{k+1} = \mathbf{u}_k + p(\mathbf{A}) \cdot \underbrace{(\mathbf{b} - \mathbf{Au}_k)}_{\mathbf{r}_k}$$

Embora haja diversos pré-condicionadores e métodos de Krylov, para este trabalho usamos a opção padrão do PETSc que utiliza o método dos mínimos resíduos generalizados (GMRES) e não utiliza pré-condicionador.

4.2 Estratégia de implementação com PETSC

4.2.1 Montagem da matriz

Como dito anteriormente, o PETSc divide o domínio em um conjunto de pontos e os delega para cada processo (`rank`), além de fazer a indexação. O que se obtém é como indicado na Figura (4.4) em que se tem uma região quadricular.

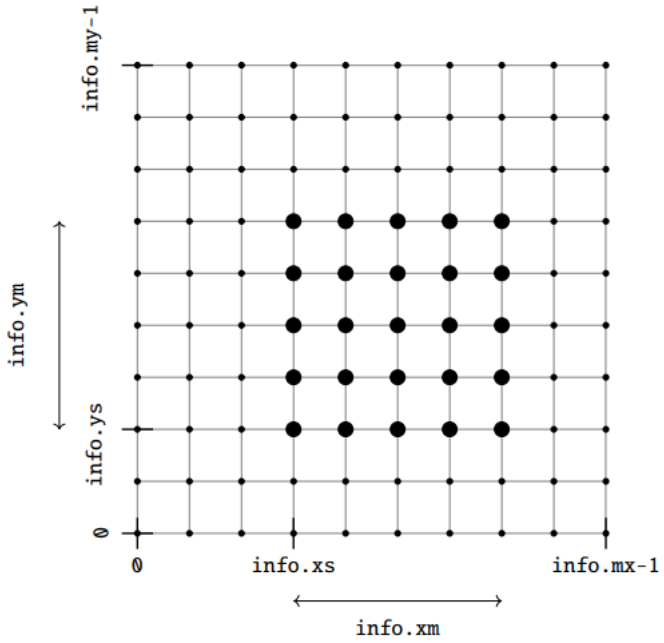


Figura 4.4: Divisão de um domínio de um processo em uma malha estruturada 2D. O processo atual fica responsável apenas por $x_m \times y_m$ nós. [Bueler 2020]

Para preencher a matriz $\mathbf{A}$, a ideia é iterar para cada ponto da malha e utilizar os *stencils* para inserir os valores nas posições da matriz. Por exemplo, temos o *stencil* de 5 pontos gravado no vetor `v[5]`

$$\{v\} = \begin{bmatrix} 2\left(\frac{h_x}{h_y} + \frac{h_y}{h_x}\right) & \frac{-h_y}{h_x} & \frac{-h_y}{h_x} & \frac{-h_x}{h_y} & \frac{-h_x}{h_y} \end{bmatrix}$$

Que queremos gravar na linha `row` da matriz

$$\mathbf{A}_{row} = \begin{bmatrix} \cdots & \square & \cdots & \square & \square & \square & \cdots & \square & \cdots \end{bmatrix}$$

Assim, no nó u_{ji} se grava as posições (da matrix) dentro do *stencil* `col`

$$\begin{bmatrix} col_i \\ col_j \end{bmatrix} = \begin{bmatrix} i & i-1 & i+1 & i & i \\ j & j & j & j-1 & j+1 \end{bmatrix}$$

Então `MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES)` in-

sere os valores de `v` na matriz global do PETSc na linha `row` e nas colunas `col`. Podemos pensar que $\mathbf{A}$ é uma matriz em $\mathbb{R}^4$ e que resolveremos o sistema

$$\mathbf{A} : \mathbf{u} = \mathbf{b} \Longleftrightarrow \sum_{k,l} A_{ijkl} \cdot u_{kl} = b_{ij}$$

de modo que preenchemos a matriz sem nos preocuparmos da indexação verdadeira do ponto, que seria algo como $index = i \cdot n_y + j$

Para a implementação, temos apenas que ter cuidado nas bordas, pois devemos colocar apenas o valor 1 e não utilizar o *stencil*. A implementação em C desta parte é dada pelo código

```
PetscErrorCode formMatrix(DM da, Mat A) {
    PetscErrorCode ierr;
    DMDALocalInfo  info;
    MatStencil     row, col[5];
    PetscReal      hx, hy, v[5];
    PetscInt       i, j, ncols;

    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
    hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
    for (j = info.ys; j < info.ys+info.ym; j++) {
        for (i = info.xs; i < info.xs+info.xm; i++) {
            row.j = j;           // row of A corresponding to (x_i,y_j)
            row.i = i;
            col[0].j = j;        // diagonal entry
            col[0].i = i;
            ncols = 1;
            if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
                v[0] = 1.0;      // on boundary: trivial equation
            } else {
                v[0] = 2*(hy/hx + hx/hy); // interior: build a row
                if (i-1 > 0) {
                    col[ncols].j = j;    col[ncols].i = i-1;
                    v[ncols++] = -hy/hx;
                }
                if (i+1 < info.mx-1) {
                    col[ncols].j = j;    col[ncols].i = i+1;
                    v[ncols++] = -hy/hx;
                }
                if (j-1 > 0) {
                    col[ncols].j = j-1;  col[ncols].i = i;
                    v[ncols++] = -hx/hy;
                }
                if (j+1 < info.my-1) {
                    col[ncols].j = j+1;  col[ncols].i = i;
                    v[ncols++] = -hx/hy;
                }
            }
            ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES); CHKERRQ(ierr);
        }
    }
    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    return 0;
```

```
44 }
```

Listagem 4.1: Função de preenchimento da matriz **A** do método de diferenças finitas 2D utilizando o PETSc

4.2.2 Montagem do vetor força

Para montar o vetor, usa-se o mesmo princípio de montar uma matriz em um código sequencial: `b[j][i] = hx*hy*g(xi,yj)`. Apenas para as condições de contorno que temos `b[j][i] = 0`.

A diferença neste caso é que se utiliza uma matriz auxiliar (nomeada por `ab`) para preencher e então inserir na matriz `b` do PETSc utilizando `DMDAVecRestoreArray`.

Assim, temos o código dado por

```
1  PetscErrorCode formRHS(DM da, Vec b) {
2      PetscErrorCode ierr;
3      PetscInt       i, j;
4      PetscReal      hx, hy, x, y, f, **ab;
5      DMDALocalInfo  info;
6
7      ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
8      hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
9      ierr = DMDAVecGetArray(da, b, &ab);CHKERRQ(ierr);
10     for (j=info.ys; j<info.ys+info.ym; j++) {
11         y = j * hy;
12         for (i=info.xs; i<info.xs+info.xm; i++) {
13             x = i * hx;
14             if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
15                 ab[j][i] = 0.0;  // on boundary: 1*u = 0
16             } else {
17                 f = 2.0 * ( (1.0 - 6.0*x*x) * y*y * (1.0 - y*y)
18                     + (1.0 - 6.0*y*y) * x*x * (1.0 - x*x) );
19                 ab[j][i] = hx * hy * f;
20             }
21         }
22     }
23     ierr = DMDAVecRestoreArray(da, b, &ab); CHKERRQ(ierr);
24     return 0;
25 }
```

Listagem 4.2: Função de preenchimento do vetor **b** do método de diferenças finitas 2D utilizando o PETSc

4.2.3 Resolução do sistema linear

Para resolver o sistema linear, apenas chamamos as funções.

```
1      // fill vectors and assemble linear system
2      ierr = formExact(da,uexact); CHKERRQ(ierr);
3      ierr = formRHS(da,b); CHKERRQ(ierr);
4      ierr = formMatrix(da,A); CHKERRQ(ierr);
5
```

```
6     // create and solve the linear system
7     ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
8     ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
9     ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
10    ierr = KSPSolve(ksp,b,u); CHKERRQ(ierr);
```

Listagem 4.3: Parte do código para resolução do sistema linear utilizando o PETSc.

Capítulo 5

Resultados para código Poisson KSP

5.1 Máquinas utilizadas e tratamento de dados

Para realizar o cálculo com diferente número de processadores e diferentes tamanhos de malha foi utilizado principalmente o *Google Cloud*. O *Google Cloud* oferece um período de teste gratuito por 3 meses, ou U$ 300,00, o que acabar primeiro. Eles permitem escolher diferentes máquinas, mas sua política de teste gratuito não permite escolher ter mais de 8 vCPUs (CPUs virtuais) simultaneamente. Desta forma, foram utilizadas três diferentes máquinas:

- *e2-highcpu-8*: 8 vCPU (4 CPU), 8 GB de memória
- *e2-standard-8*: 8 vCPU (4 CPU), 32 GB de memória
- *e2-highmem-8*: 8 vCPU (4 CPU), 64 GB de memória

Embora as máquinas representem diferentes configurações, todas se apresentam como máquinas *quad-core* (4 núcleos) utilizando o mesmo tipo de processador. Embora máquina *highcpu* pareça significar que os seus CPUs são mais rápidos em relação ao *standard* ou *highmem*, a diferença se deu apenas a nível de memória, não havendo diferença perceptível em suas velocidades. Desta forma, nos resultados não faremos distinção entre as máquinas.

Outras máquinas foram procuradas para realizar os cálculos, tal como computador pessoal ou utilizar o Google Colab. Mas ambas possuem apenas 2 processadores, o que tornaria as análises em relação ao número de processadores inconclusivas. Foi pensado em utilizar GPU da NVidia dentro do Google Colab, mas a utilização das GPUs pelo PETSc ainda está sendo desenvolvida e para tal uso seria necessário modificar o código e programar usando CUDA, adicionando complexidade ao código.

Assim, se acessou cada uma das máquinas utilizando o protocolo SSH (*Secure Socket Shell*) e a linha de comando para instalar, configurar e lançar as simulações. Para automatizar o processo de instalar e configurar, utilizou-se um `makefile` disponível dentro do

repositório GitHub criado para este trabalho[Carlos Adir Ely Murussi Leite 2022]. Para lançar as simulações e automatizar ao máximo o processo, utilizou-se um outro `makefile` usando *loop* no *shellscript* para variar a quantidade de processadores como mostrado abaixo.

```
run2Dksp: createfolder compilepoisson2D_ksp
    { \
    set -e ;\
    code="poisson2D_ksp";\
    echo "For code $${code}";\
    for nx in 50 63 64 65 100 128 129 140 150 175 200 225 256 257 300 325 350 400 450 500
        512 513 550 600 650 700 750 850 950 1000 1100 1250 1300 1400 1500 2000 3000 5000
        7500 10000; do \
        for p in {1..${pmax}}; do \
            ny=$$nx; \
            echo "    With $${p} processors and mesh $${nx} x $${ny}"; \
            filename="${FOLDER}/$${code}-p$${p}-mesh$${nx}x$${ny}.txt"; \
            if [ ! -e $${filename} ]; then \
                executeone="./$${code} -da_grid_x $${nx} -da_grid_y $${ny} -log_view"; \
                command="${MPIEXEC} -prepend-rank -np $${p} $${executeone}"; \
                { \
                    $${command} | cat >> $${filename}; \
                } 2>> $${filename}; \
            fi ; \
        done \
    done \
    }
```

Listagem 5.1: Código makefile para execução de poisson2D_ksp.c para diversos processadores e números de malha

Os resultados são automaticamente escritos em arquivos dentro de uma pasta `results` (criada automaticamente). Tome como exemplo o arquivo `poisson2D_ksp-p2-mesh500x500.txt` que foi rodado o código `poisson2D_ksp.c` com 2 processadores em uma malha 500×500. Para leitura dos arquivos foi utilizado Python com biblioteca `re` (*Regular expressions*). Para o tratamento foi utilizado o `numpy` em conjunto com o `matplotlib`.

Embora o código aceite uma malha estruturada retangular (isto é $h_x \neq h_y$), todos as simulações foram feitas com $h_x = h_y$. O comando `-log_view` do PETSc permite a coleta dos dados, cujos itens principais que utilizaremos são:

- Contador de cada operação
- Tempo de cada operação
- Flop (*Floating points operations)*
- Quantidade de mensagens
- Memória utilizada

5.2 Resultados Poisson KSP 2D

5.2.1 Resultados brutos obtidos KSP 2D

Rodando diversas vezes o código `poisson2D_ksp.c` para diferentes número de processadores p e diferentes tamanhos de malha n, obtivemos os valores do tempo t consumido, conforme mostra a Figura (5.1). Foram testados malhas de até 10000×10000 (100 milhões de pontos) pois acima destes valores a memória disponível não era suficiente para armazenar tudo e o programa encerrava por si só.

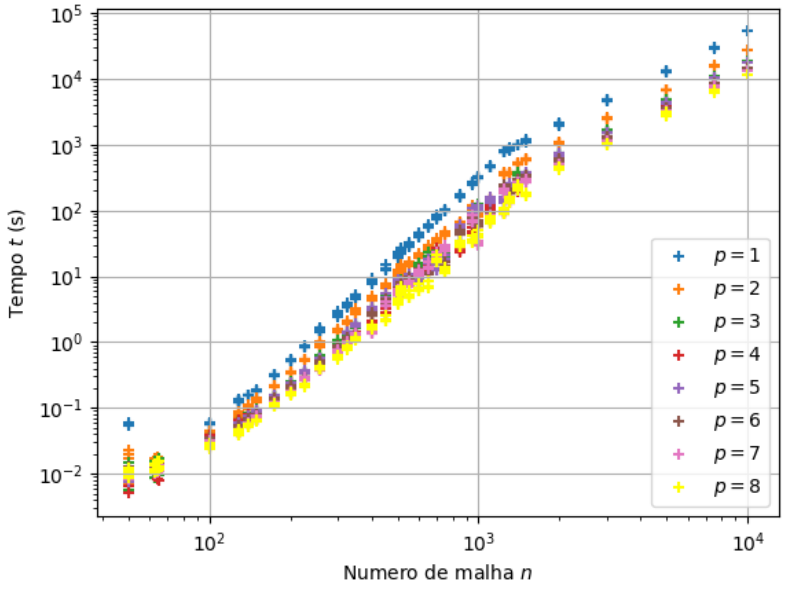


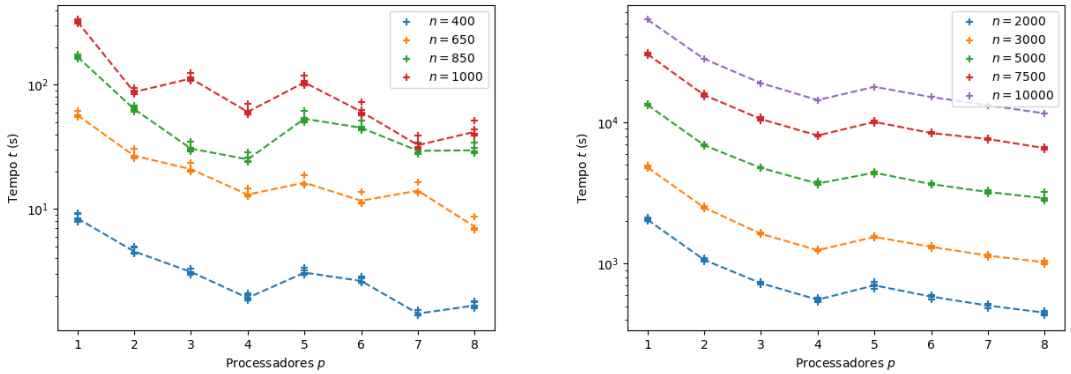
Figura 5.1: Tempo consumido em função do número de malha para o código *poisson2D_ksp.c* para diferentes números de processadores. Cada ponto representa um cálculo realizado com malha n e processador p especificado.

Desta figura é difícil de aplicar as métricas desejadas e desta forma dividimos o conjunto de dados em 2 grupos dependendo do tamanho da malha n, como mostra a Figura (5.2).

5.2.2 Aceleração e eficiência dos resultados para 2D KSP

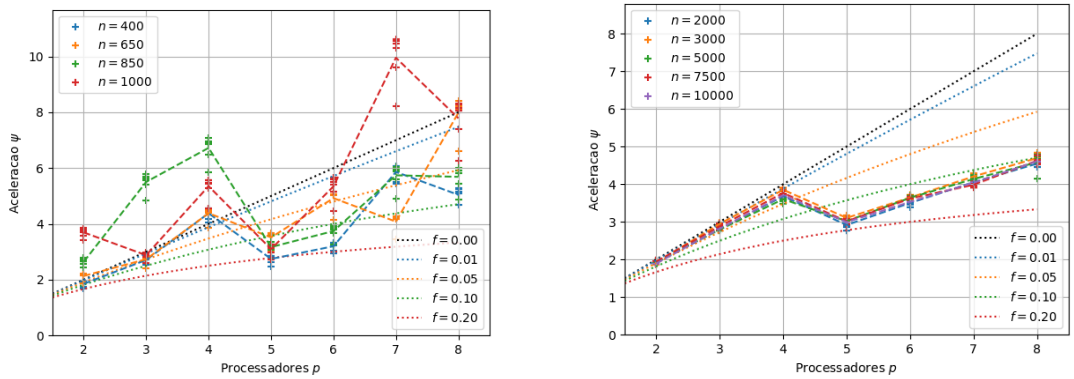
Calculando a aceleração e a eficiência utilizando as Equações (2.3) e (2.4) a partir dos dados (5.2) obtemos as Figuras (5.3) e (5.4), respectivamente. Vemos na Figura (5.3a) que a aceleração para $n = 1000$ ultrapassa a aceleração máxima $\psi = p$ para os processadores 2, 4 e 7 e para $n = 850$ para os processadores 2, 3 e 4. Este resultado é contra-intuitivo e teoricamente é impossível de acontecer segundo as leis aqui expostas. Esse fenômeno que ocorre é nomeado por **aceleração superlinear**.

Uma das hipóteses que explica tal fenômeno é feita em relação à memória: Toda CPU tem uma memória *cache* próxima e de rápido acesso. Se o problema couber completamente



(a) (b)

Figura 5.2: Valores de tempo medidos em função do número de processadores para diferentes tamanhos de malha. (a) malha mediana ($n \approx 800$) e (b) malha grande ($n \approx 6000$).



(a) (b)

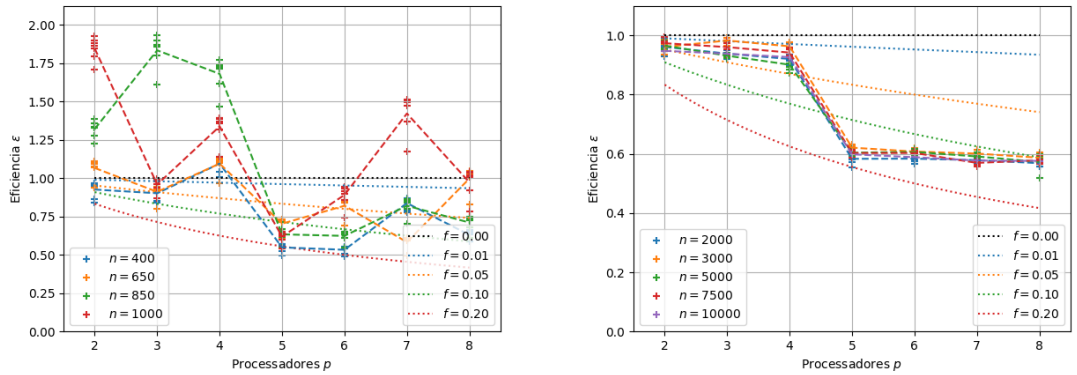
Figura 5.3: Valores medidos de aceleração (2.3) para o caso 2D KSP para diferentes malhas, (a) uma malha mediana e (b) uma malha grande. As linhas pontilhadas representam curvas teóricas de fração sequencial f constante. As linhas tracejadas representam a média dos pontos.

dentro da memória *cache* sem o uso de memória RAM ou de memória externa como HD/SSD ou *swap* (para o linux), então o tempo $T(n, p)$ fica reduzido em relação a $T(n, 1)$, pois com apenas 1 processador é necessário o acesso de outros tipos de memória. Este fenômeno não é observado para valores $n \approx 400$ pois, segundo esta hipótese, o $T(n, 1)$ seria pequeno pois o problema todo caberia dentro da memória *cache* de apenas um processador. Não pudemos confirmar tal hipótese por desconhecimento dos componentes físicos das máquinas utilizadas pelo Google Cloud.

Outro fenômeno que vemos a partir das Figuras (5.2a) e (5.2b) é que o tempo para 5 processadores é sempre maior que o tempo com 4 processadores. Isso se traduz nos gráficos da aceleração (5.3a) e (5.3b) como sempre havendo uma queda dos valores de 4 para 5 processadores.

A primeira hipótese a ser feita diz respeito aos CPUs virtuais. No caso do Google

Cloud, uma CPU é dividida em 2 vCPUs, e estas não executam tarefas simultaneamente [IBM 2022]. Para tentarem se comportar como dois CPUs independentes, eles recorrem ao *hyperthreading*, o que leva a um comportamento mais próximo, mas não igual de duas CPUs separadas, e assim o resultado com 8 vCPUs (obtido) não é mesmo com 8 CPU (desejado). O que reforça essa ideia é o gráfico do efeito do *hyperthreading* é a Figura (5.4b) em que a eficiência é praticamente constante (devido à fração sequencial próxima de zero) com a exceção de um salto (de 4 a 5 processadores) Tentou-se encontrar no Google Cloud máquinas com 8 CPUs e 8 vCPUs (isto é, 1 vCPU por CPU) para avaliar tal hipótese mas não foi possível.



(a) (b)

Figura 5.4: Valores medidos de eficiência (2.4) para o caso 2D KSP para diferentes malhas, (a) com malha mediana e (b) com malha grande. As linhas pontilhadas representam curvas teóricas de fração sequencial f constante. As linhas tracejadas representam a média dos pontos.

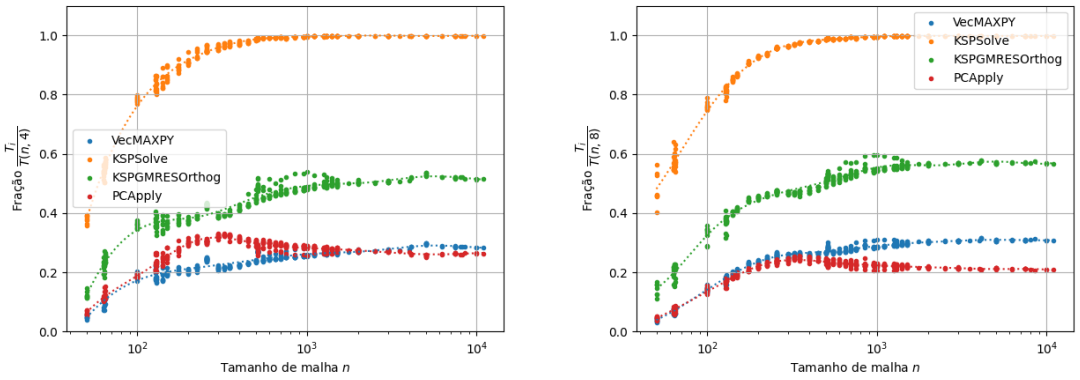
A segunda hipótese se diz em relação à forma de divisão que o PETSc faz em relação à malha quadrada. Como dito anteriormente, o PETSc faz a divisão da malha de forma automática e para p não adequados (geralmente primos), a divisão da malha gera desbalanceamentos, seja na quantidade de nós por processador, seja na quantidade de nós fantasmas criados. Para 4 processadores, se tem uma divisão intuitiva: cada processador recebe uma das 4 regiões delimitadas por $x = 0.5$ e $y = 0.5$. Já para 5 processadores, a divisão não é tão óbvia, como mostra a Figura (4.2b).

5.2.3 Utilização das métricas para 2D KSP

Lei de Amdahl e Gustafson-Barsis

O código `poisson2D_ksp.c` foi paralelizado e as partes que consumiam mais tempo, na montagem dos elementos matriz e vetor, e na resolução do sistema linear usando o espaço de Krylov (KSP), se tornaram completamente paralelas. Como o PETSc já retorna o tempo ocorrido por cada operação através do comando `-log_view`, desconsideramos pegar os tempos de cada evento (serial ou paralelizável) manualmente, através de `print` ao longo do código.

A Figura (5.5) mostra os eventos que levaram mais tempo para acontecer, considerando apenas aqueles que levaram pelo menos 20% do tempo de processamento. Para todos os processadores, os que consumiram mais tempo foi o `KSPSolve`, que resolve o sistema de matriz, e o `KSPGMRESOrthog`, que realiza a ortogonalização de vetores devido ao método GMRES padrão do KSP. É difícil ter valores para f (medido serialmente) ou de s (medido paralelamente) pois todos os eventos que consumiram tempo obtidos pelo comando `-log_view` se mantiveram proporcionais ao tempo total $T(n,p)$, indicando que a parte serial σ é muito pequena. A Figura (5.4b) corrobora com esse resultado, pois a eficiência para grandes valores de n com $p \leq 4$ ficaram logo abaixo da curva $f = 0.01$, indicando que a fração sequencial é próxima de 1%.



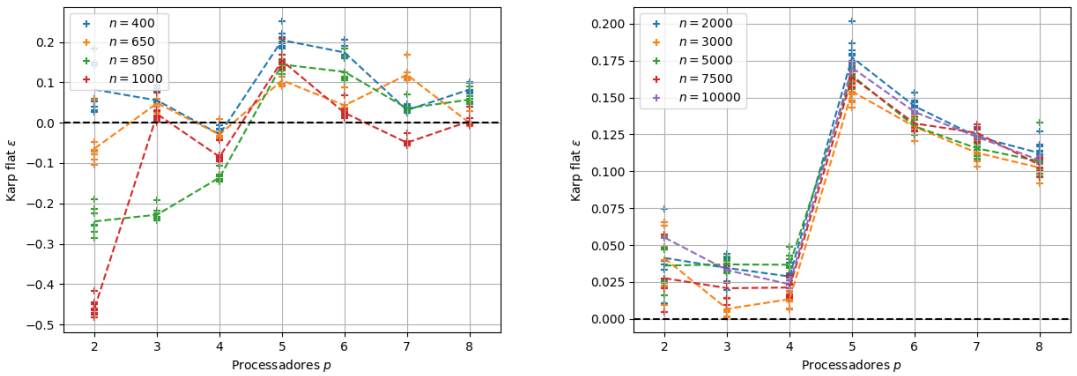
(a) (b)

Figura 5.5: Fração de tempo T_i de cada evento e o tempo $T(n,p)$ com apenas (a) 1 processador, (b) 4 processadores, (c) 5 processadores e (d) 8 processadores. Foram levados em consideração apenas eventos que tiveram tempo $T_i > 0.2T(n,p)$

Métrica de Karp-Flatt

Agora, calculando a fração serial experimentalmente determinada utilizando a Equação (2.11 obtemos a Figura (5.6). Primeiro ponto que vemos é que para n médio (Figura 5.6a) obtemos valores e menores que zero, devido ao fenômeno de aceleração superlinear falado anteriormente.

Já para n grande (Figura 5.6b), vemos que há uma grande subida de 4 a 5 processadores e isso se teve principalmente à hipótese dos CPUs virtuais como analisado na queda da eficiência na Figura (5.4b). Vemos que para $2 \leq p \leq 4$ o valor de e é praticamente constante, indicando que a parcela de comunicação κ não é relevante.



(a) (b)

Figura 5.6: Fração sequencial experimental e (Equação 2.11) de Karp-Flatt para o caso 2D KSP para diferentes malhas (a) com malha mediana ($n \approx 800$) e (b) com malha grande ($n \approx 6000$).

Lei da isoeficiência

Para a lei de isoeficiência, fizemos como esperado: calculamos ε para um conjunto de $(p,\ n)$. Disto, poderíamos fazer uma regressão e encontrar as funções $n(p)$ de eficiência ε constante.

Contudo, para certos valores de n, a eficiência ε obtida foi maior que 1 e não é esperado que tal eficiência seja mantida conforme aumentemos o número de processadores e de malha. Além disso, com fração serial praticamente nula já se tem uma eficiência praticamente constante, independente de p. Para avaliação deste quesito, seria importante efetuar cálculos com valores maiores de p.

Outro fator importante é que para os nossos cálculos, não era possível adicionar mais memória em conjunto com os processadores: A memória utilizada com 2 processadores era a mesma com 8. Além disso, para o nosso algoritmo, assim que se atingia o limite de memória, o processo não completava, nos impedindo de analisar a queda de eficiência à memória máxima.

Para o nosso caso, também não é possível obter um valor teórico sobre a função de isoeficiência pois esta função é calculada a partir da quantidade de cálculos necessário para se completar as tarefas. É bem conhecida a quantidade de operações que se faz para preencher os elementos do vetor e da matriz. Contudo, como a resolução do sistema linear é iterativa e não se conhece o número total de iterações, encontrar a função de isoeficiência é difícil. Poderíamos estimar a partir do raio espectral ρ da matriz de iteração, mas como o polinômio

$p_n(\mathbf{A})$ pode mudar ao longo da resolução do sistema linear, o que bem difícil de obter o número máximo de iterações o qual se tem certeza que o método converge.

5.3 Resultados Poisson KSP 3D

5.3.1 Resultados brutos obtidos KSP 3D

Do mesmo modo que fizemos para o problema bidimensional, o programa de poisson tridimensional `poisson3D_ksp.c` foi executado para diferentes processadores p e diferentes números de malha n, obtendo o valor do tempo t consumido mostrado pela Figura (5.7). Foram testados apenas malhas até $500 \times 500 \times 500$ (125 milhões de nós) pois acima destes valores a memória disponível não era suficiente para armazenar o problema e o programa se encerrava.

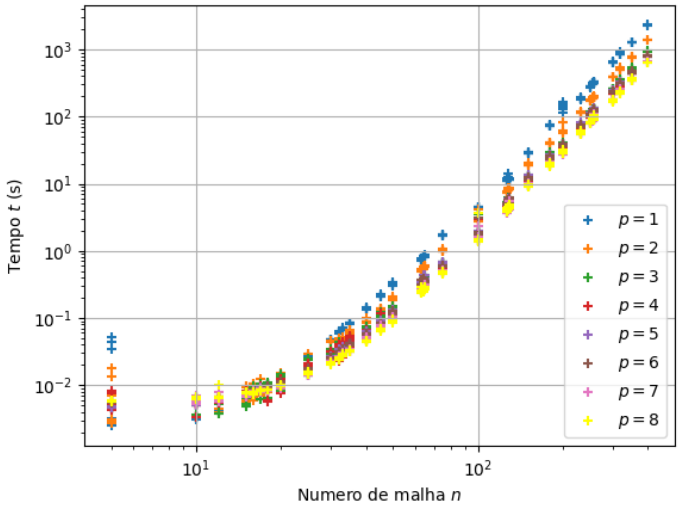
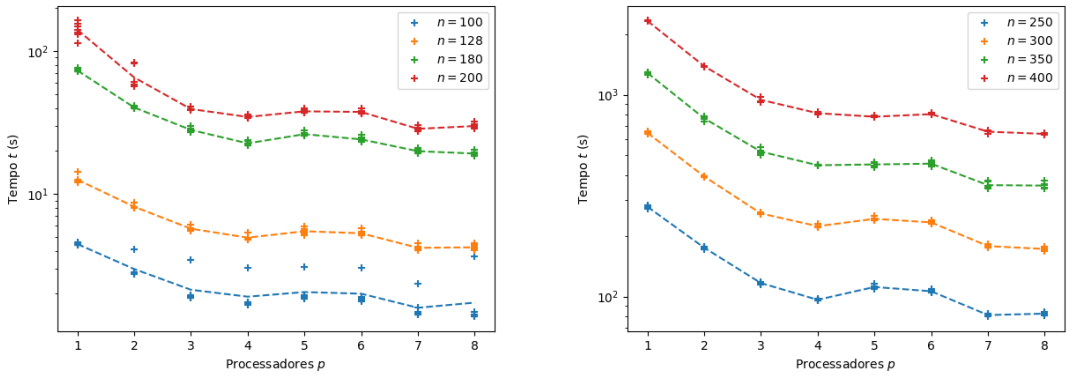


Figura 5.7: Tempo consumido em função do número de malha para o código *poisson3D_ksp.c* para diferentes números de processadores. Cada ponto representa um cálculo realizado com malha n e processador p especificado.



(a) (b)

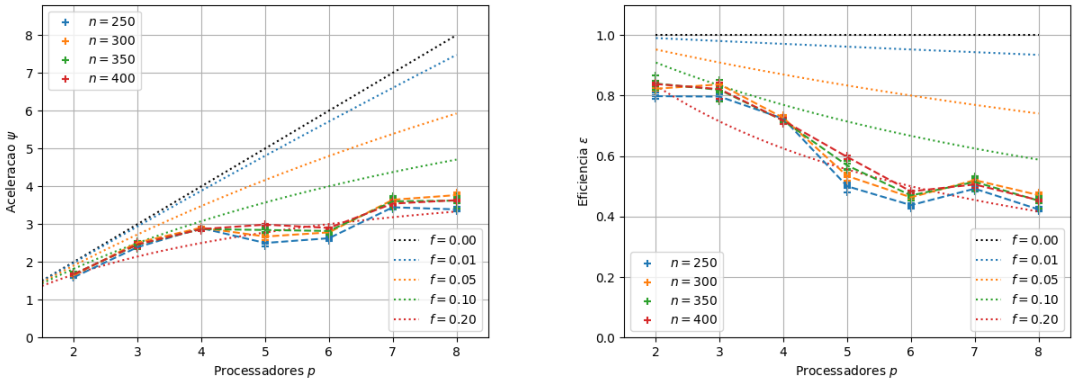
Figura 5.8: Valores de tempo medidos em função do número de processadores para diferentes tamanhos de malha n: (a) com malha mediana e (b) com malha grande

5.3.2 Aceleração e eficiência dos resultados para 3D KSP

As acelerações e eficiências foram calculados a partir dos dados da Figura (5.8) são mostradas pelas Figuras (5.9) e (5.10).

Podemos ver que para grandes valores de n a aceleração (Figura 5.9a) foi progressivamente caindo à medida que o número de processadores aumentava, o que se vê também no gráfico da eficiência (Figura 5.9b). Elas ficaram próximas uma da outra, assim como aconteceu para o caso 2D, e indica que a aceleração ψ e a eficiência ε não varia para grandes valores de n.

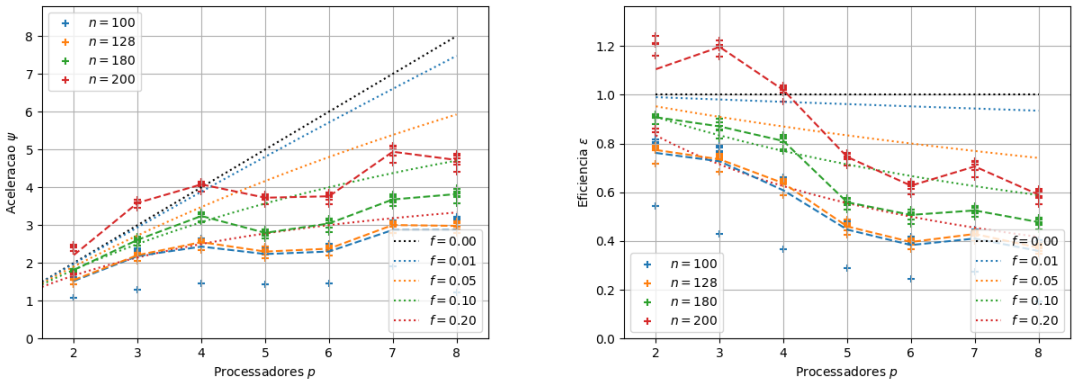
Além disso, no caso bidimensional para grandes valores de n, havia uma queda abrupta na aceleração de 4 para 5 processadores que não se vê no caso tridimensional. Este resultado indica que, ou o processo de divisão da malha era bem importante para determinar a aceleração no caso bidimensional, ou a hipótese de que os CPUs virtuais são mais lentos que um CPU independente não é aplicável neste caso.



(a) (b)

Figura 5.9: Valores medidos de (a) aceleração (Eq. 2.3) e (b) eficiência (Eq. 2.4) para o caso 3D KSP com uma malha grande($n \approx 350$). As linhas pontilhadas representam curvas teóricas de fração sequencial f (Eq. 2.6) constantes

Já para médios valores de n mostrados pela Figura (5.10), vemos que houve resultados acima da aceleração máxima (eficiência maior que um), indicando a aceleração superlinear como ocorrida também no caso bidimensional. No caso bidimensional, umas hipóteses que se levantou foi em relação ao preenchimento da memória *cache* que também pode se aplicar nesse caso.



(a) (b)

Figura 5.10: Valores medidos de (a) aceleração (Eq. 2.3) e (b) eficiência (Eq. 2.4) para o caso 3D KSP com uma malha mediana ($n \approx 150$). As linhas pontilhadas representam curvas teóricas de fração sequencial f (Eq. 2.6) constantes

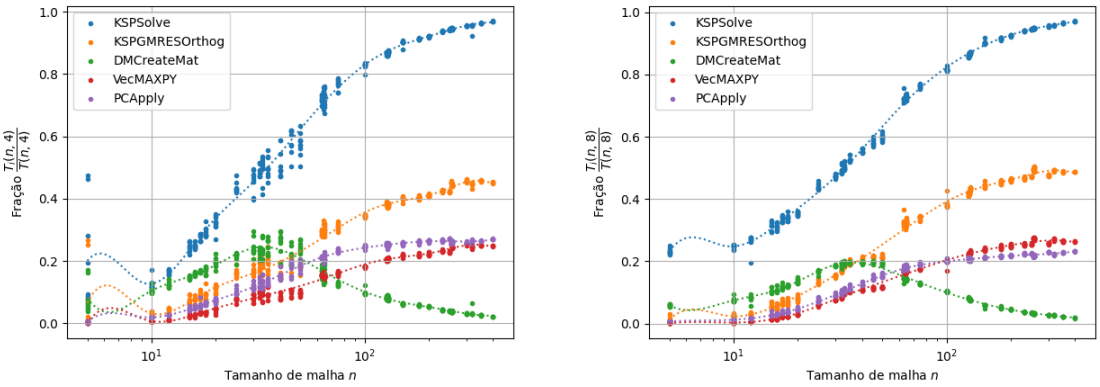
5.3.3 Utilização das métricas para 3D KSP

Lei de Amdahl e Gustafson-Barsis

Como podemos ver na Figura (5.9), para grandes valores de n a aceleração e eficiência ficam abaixo das curvas de $f = 0.10$, indicando que a fração sequencial deste código seria próxima de 10%.

Contudo, fazendo a mesma análise referente ao tempo que cada evento utilizou através o comando `-log_view`, se obtém o valor de f bem baixo ($f \approx 0.01$, e a Figura (5.11) não apresenta modificação significativa entre 4 e 8 processadores. Se f fosse próximo de 10%, então na Figura (5.11b) haveria uma curva que consumiria ao menos $s = 47\%$ do tempo. Além disso, o código é praticamente o mesmo do 2D, de modo que as partes paralelizadas são exatamente as mesmas.

Um fator que explica isso é que o fator de comunicação κ é grande para $p \geq 2$. Isso é algo lógico pois a quantidade de nós fantasmas é bem maior para o problema 3D que para o problema 2D: Enquanto no 2D se tem aproximadamente $2pn$ nós fantasmas, para o 3D se tem $2pn^2$ nós fantasmas. De todo modo, vemos que os valores da aceleração ψ e eficiência ε ficam abaixo da curva $f = 0.01$, condizente com as leis de Amdahl e Gustafson-Barsis.

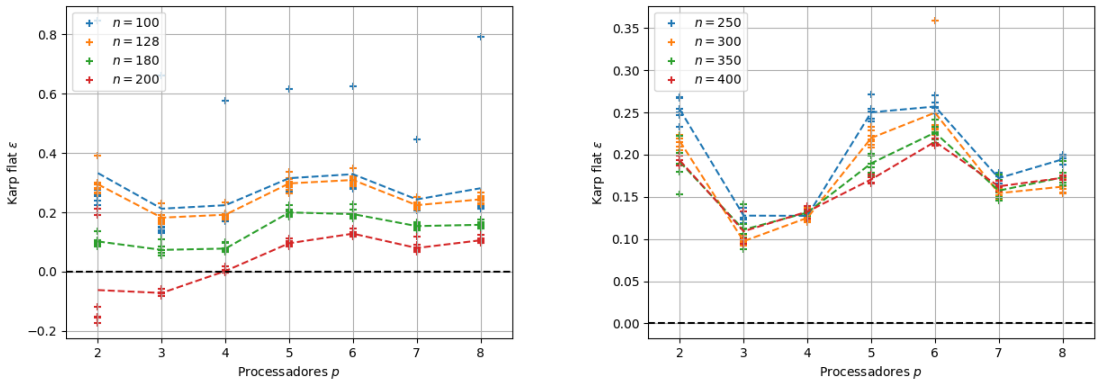


(a) (b)

Figura 5.11: Fração de tempo T_i de cada evento e o tempo $T(n, p)$ com apenas (a) 4 processadores e (b) 8 processadores. Foram levados em consideração apenas os eventos que tiveram tempo $T_i > 0.2\,T(n,\ p)$.

Métrica de Karp-Flatt

Usando a métrica de Karp-Flatt, obtemos a Figura (5.12), onde vemos que parcela de comunicação κ não é constante pois as curvas também não são constantes. Para $n = 200$ (Figura 5.12) vemos que a fração sequencial experimental e se torna negativa, devido à aceleração superlinear como comentado acima. Os demais resultados indicam que a parcela de comunicação κ se torna bem elevada, sendo responsável por aproximadamente 20% de todo o tempo de cálculo.



(a) (b)

Figura 5.12: Fração sequencial experimental e de Karp-Flatt para o caso 3D KSP para diferentes malhas: (a) malha mediana ($n \approx 150$) e (b) malha grande ($n \approx 350$)

Capítulo 6

Conclusão

A programação paralela é uma alternativa que a curto prazo deve ser considerada para aumentar a velocidade de cálculo. Contudo, os problemas que aparecem na programação paralela não são muito intuitivos. Mesmo para um problema de Poisson linear, em uma malha estruturada quadrada, os resultados obtidos não seguiam curvas padrões esperadas das métricas utilizadas.

Houve resultados inesperados, como eficiências ε maiores que 1 e que a priori deveriam ser impossíveis e que resultam no fenômeno de aceleração superlinear. A hipótese levantada foi de que o tamanho da malha era suficientemente pequeno para caber inteiramente na memória *cache* do processador, que é de muito fácil acesso e isso acelerava bastante em relação a um código serial. Este fenômeno não foi observado para grandes valores de malha.

Uma surpresa obtida é que a eficiência do código 2D sempre caia além do esperado (para porção sequencial constante) quando passava de 4 para 5 processadores. Não se sabe se isso se deve à divisão interna do *hardware* do Google Cloud que utiliza vCPU que são concorrentes com outros processos e influencia no cálculo. Outra hipótese levantada é da divisão da malha, que para números primos, no caso 5, a divisão dos nós é desbalanceada requerendo mais comunicação e mais memória. Já para o caso 3D, não se observa a queda de 4 a 5 processadores pois, na hipótese levantada, a comunicação é um fator bem mais importante em 3D que em 2D, representando cerca de 20% do tempo serial.

O uso do PETSc foi fundamental para mais fácil implementação de código, sendo não necessário fazer a sincronização manual entre os processadores e o preenchimento dos elementos vetor e matriz foram bem intuitivos. Embora tal ferramenta permita um nível maior de abstração e uma redução do requisito dos programadores, ainda é necessário que estes entendam sobre o algoritmo paralelo utilizado para ser capaz de analisar os fenômenos decorrentes do *hardware* utilizado, como apresentado o fenômeno de aceleração superlinear obtido.

6.1 Trabalhos futuros

Outras análises sobre este problema podem ser feitas modificando o *solver*: aplicando pré-condicinadores ou mudando método KSP utilizado. Outras análises podem ser feitas utilizando malhas retangulares ou analisar o erro obtido em relação ao número de iterações. Outra possibilidade é de resolver o problema de poisson não linear.

Referências Bibliográficas

[Bueler 2020] Bueler, E. (2020). *PETSc for Partial Differential Equations: Numerical Solutions in C and Python*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[Burden and Faires 1989] Burden, R. L. and Faires, J. D. (1989). *Numerical Analysis*. The Prindle, Weber and Schmidt Series in Mathematics. PWS-Kent Publishing Company, Boston, fourth edition.

[Carlos Adir Ely Murussi Leite 2022] Carlos Adir Ely Murussi Leite (2022). Repositório github - projeto de graduação. `https://github.com/carlos-adir/ProjetoDeGraduacao`. [Online; accessed 19-April-2022].

[Castro et al. 2009] Castro, G., Paz, R., Storti, M., Sonzogni, V., and Dalcin, L. (2009). Parallel implementation of a fem code by using mpi/petsc and openmp hybrid programming techniques. volume XXVIII.

[IBM 2022] IBM (2022). Núcleo do processador virtual (vpc). `https://www.ibm.com/docs/pt-br/license-metric-tool?topic=metrics-virtual-processor-core-vpc`. [Online; accessed 01-Septembre-2022].

[Intel 2018] Intel (2018). Automatic parallelization with intel compilers. `https://www.intel.com/content/www/us/en/developer/articles/technical/automatic-parallelization-with-intel-compilers.html`. [Online; Acesso em 28 de Agosto de 2022].

[MakeTechEasier Alexander Fox 2018] MakeTechEasier Alexander Fox (2018). Why cpu clock speed isn't increasing. `https://www.maketecheasier.com/why-cpu-clock-speed-isnt-increasing/`. [Online; accessed 20-April-2022].

[Pedro Cesar Tebaldi 2015] Pedro Cesar Tebaldi (2015). O que é e como funciona um cluster? `https://www.opservices.com.br/o-que-e-um-cluster/`. [Online; accessed 25-April-2022].

[PETSc website 2022] PETSc website (2022). Petsc - manual. https://petsc.org/release/docs/manual/about_this_manual/. [Online; accessed 18-September-2022].

[Quinn 2004] Quinn, M. J. (2004). Parallel programming in c with mpi and openmp.

[Raspberry Pi 2022] Raspberry Pi (2022). How to build a raspberry pi cluster. https://www.raspberrypi.com/tutorials/cluster-raspberry-pi-tutorial/. [Online; accessed 25-April-2022].

[Sankalp Hariharan; Remmy Martin Kilonzo 2018] Sankalp Hariharan; Remmy Martin Kilonzo (2018). Amd: The autonomous drive to profitability. https://iveybusinessreview.ca/5944/amd-autonomous-drive-profitability/. [Online; accessed 25-April-2022].

[Washington University in St. Louis 2021] Washington University in St. Louis (2021). Analog computers now just one step from digital: Work from mckelvey school of engineering may help usher in. www.sciencedaily.com/releases/2021/12/211209082557.htm. [Online; accessed 25-April-2022].

[Wikipedia contributors 2017] Wikipedia contributors (2017). Explicit parallelism — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Explicit_parallelism&oldid=802845156. [Online; accessed 18-September-2022].

[Wikipedia contributors 2022a] Wikipedia contributors (2022a). Moore's law — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=1083232534. [Online; accessed 20-April-2022].

[Wikipedia contributors 2022b] Wikipedia contributors (2022b). Partial differential equation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Partial_differential_equation&oldid=1100838869. [Online; accessed 8-September-2022].

[Wikipedia contributors 2022c] Wikipedia contributors (2022c). Sisal — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SISAL&oldid=1088687740. [Online; accessed 18-September-2022].

[Wikipedia contributors 2022d] Wikipedia contributors (2022d). Transistor count — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Transistor_count&oldid=1084619873. [Online; accessed 25-April-2022].

Apêndice A

Código poisson2D_ksp.c

```
1  static char help[] = "A structured-grid Poisson solver using DMDA+KSP.\n\n";
2
3  #include <petsc.h>
4
5  extern PetscErrorCode formMatrix(DM, Mat);
6  extern PetscErrorCode formExact(DM, Vec);
7  extern PetscErrorCode formRHS(DM, Vec);
8
9  //STARTMAIN
10 int main(int argc,char **args) {
11     PetscErrorCode ierr;
12     DM             da;
13     Mat            A;
14     Vec            b,u,uexact;
15     KSP            ksp;
16     PetscReal      errnorm;
17     DMDALocalInfo info;
18     PetscLogDouble t1, t2;
19     PetscMPIInt rank;
20
21
22     ierr = PetscInitialize(&argc, &args, NULL, help); if (ierr) return ierr;
23     PetscCall(PetscTime(&t1));
24     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
25     // change default 9x9 size using -da_grid_x M -da_grid_y N
26     ierr = DMDACreate2d(PETSC_COMM_WORLD,  // MPI_Comm comm
27                         DM_BOUNDARY_NONE,  // DMBoundaryType bx, type of ghost nodes at
                               boundary
28                         DM_BOUNDARY_NONE,  // DMBoundaryType by   /DM_BOUNDARY_NONE/
                               DM_BOUNDARY_GHOSTED/DM_BOUNDARY_PERIODIC/
29                         DMDA_STENCIL_STAR, // DMDAStencilType stype, /DMDA_STENCIL_BOX/
                               DMDA_STENCIL_STAR/
30                         9,                 // PetscInt M, global dimension in each
                               direction of the array
31                         9,                 // PetscInt N
32                         PETSC_DECIDE,      // PetscInt m, number of process in each
                               dimension
33                         PETSC_DECIDE,      // PetscInt n
34                         1,                 // PetscInt dof, number of degrees of freedom
                               per node
35                         1,                 // PetscInt s, stencil width, the number of grid
                                points away from the center of the stencil
36                         NULL,              // const PetscInt lx[], arrays containing the
                               number of nodes in each process' portions of the grid, or NULL
```

```
                        NULL,               // const PetscInt ly[], if non-NULL, these must
                            be of length m and n, respectivly, and the sum of lx[] and ly[]
                             must be M and N respectively
                        &da);               // DM *da, the resulting DMDA object
    CHKERRQ(ierr);

    // create linear system matrix A
    ierr = DMSetFromOptions(da); CHKERRQ(ierr);
    ierr = DMSetUp(da); CHKERRQ(ierr);
    ierr = DMCreateMatrix(da,&A); CHKERRQ(ierr);
    ierr = MatSetFromOptions(A); CHKERRQ(ierr);

    // create RHS b, approx solution u, exact solution uexact
    ierr = DMCreateGlobalVector(da,&b); CHKERRQ(ierr);
    ierr = VecDuplicate(b,&u); CHKERRQ(ierr);
    ierr = VecDuplicate(b,&uexact); CHKERRQ(ierr);

    // fill vectors and assemble linear system
    ierr = formExact(da,uexact); CHKERRQ(ierr);
    ierr = formRHS(da,b); CHKERRQ(ierr);
    ierr = formMatrix(da,A); CHKERRQ(ierr);

    // create and solve the linear system
    ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
    ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
    ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
    ierr = KSPSolve(ksp,b,u); CHKERRQ(ierr);

    // report on grid and numerical error
    ierr = VecAXPY(u,-1.0,uexact); CHKERRQ(ierr);    // u <- u + (-1.0) uxact
    ierr = VecNorm(u,NORM_INFINITY,&errnorm); CHKERRQ(ierr);
    ierr = DMDAGetLocalInfo(da,&info);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD, "on %d x %d grid:  error |u-uexact|_inf = %g\n",
        info.mx,info.my,errnorm); CHKERRQ(ierr);

    VecDestroy(&u);
    VecDestroy(&uexact);
    VecDestroy(&b);
    MatDestroy(&A);
    KSPDestroy(&ksp);
    DMDestroy(&da);
    PetscCall(PetscTime(&t2));
     ierr = PetscPrintf(PETSC_COMM_SELF, "Processor %d took %f CPU seconds\n", (int)rank, (
         t2-t1)); CHKERRQ(ierr);

    return PetscFinalize();
}
//ENDMAIN

//STARTMATRIX
PetscErrorCode formMatrix(DM da, Mat A) {
    PetscErrorCode ierr;
    DMDALocalInfo  info;
    MatStencil     row, col[5];
    PetscReal      hx, hy, v[5];
    PetscInt       i, j, ncols;

    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
    hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
    for (j = info.ys; j < info.ys+info.ym; j++) {
        for (i = info.xs; i < info.xs+info.xm; i++) {
            row.j = j;           // row of A corresponding to (x_i,y_j)
```

```
            row.i = i;
            col[0].j = j;        // diagonal entry
            col[0].i = i;
            ncols = 1;
            if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
                v[0] = 1.0;      // on boundary: trivial equation
            } else {
                v[0] = 2*(hy/hx + hx/hy); // interior: build a row
                if (i-1 > 0) {
                    col[ncols].j = j;    col[ncols].i = i-1;
                    v[ncols++] = -hy/hx;
                }
                if (i+1 < info.mx-1) {
                    col[ncols].j = j;    col[ncols].i = i+1;
                    v[ncols++] = -hy/hx;
                }
                if (j-1 > 0) {
                    col[ncols].j = j-1;  col[ncols].i = i;
                    v[ncols++] = -hx/hy;
                }
                if (j+1 < info.my-1) {
                    col[ncols].j = j+1;  col[ncols].i = i;
                    v[ncols++] = -hx/hy;
                }
            }
            ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES); CHKERRQ(ierr);
        }
    }
    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    return 0;
}
//ENDMATRIX

//STARTEXACT
PetscErrorCode formExact(DM da, Vec uexact) {
    PetscErrorCode ierr;
    PetscInt       i, j;
    PetscReal      hx, hy, x, y, **auexact;
    DMDALocalInfo  info;

    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
    hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
    ierr = DMDAVecGetArray(da, uexact, &auexact);CHKERRQ(ierr);
    for (j = info.ys; j < info.ys+info.ym; j++) {
        y = j * hy;
        for (i = info.xs; i < info.xs+info.xm; i++) {
            x = i * hx;
            auexact[j][i] = x*x * (1.0 - x*x) * y*y * (y*y - 1.0);
        }
    }
    ierr = DMDAVecRestoreArray(da, uexact, &auexact);CHKERRQ(ierr);
    return 0;
}

PetscErrorCode formRHS(DM da, Vec b) {
    PetscErrorCode ierr;
    PetscInt       i, j;
    PetscReal      hx, hy, x, y, f, **ab;
    DMDALocalInfo  info;

    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
```

```
157     hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
158     ierr = DMDAVecGetArray(da, b, &ab);CHKERRQ(ierr);
159     for (j=info.ys; j<info.ys+info.ym; j++) {
160         y = j * hy;
161         for (i=info.xs; i<info.xs+info.xm; i++) {
162             x = i * hx;
163             if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
164                 ab[j][i] = 0.0;  // on boundary: 1*u = 0
165             } else {
166                 f = 2.0 * ( (1.0 - 6.0*x*x) * y*y * (1.0 - y*y)
167                     + (1.0 - 6.0*y*y) * x*x * (1.0 - x*x) );
168                 ab[j][i] = hx * hy * f;
169             }
170         }
171     }
172     ierr = DMDAVecRestoreArray(da, b, &ab); CHKERRQ(ierr);
173     return 0;
174 }
175 //ENDEXACT
```

Listagem A.1: Código completo para o problema de poisson bidimensional utilizando o método de diferenças finitas com PETSc

Apêndice B

Código poisson3D_ksp.c

```
1
2  /*
3  Laplacian in 3D. Modeled by the partial differential equation
4
5     - Laplacian u = 1,0 < x,y,z < 1,
6
7  with boundary conditions
8
9     u = 1 for x = 0, x = 1, y = 0, y = 1, z = 0, z = 1.
10
11    This uses multigrid to solve the linear system
12
13    See src/snes/tutorials/ex50.c
14
15    Can also be run with -pc_type exotic -ksp_type fgmres
16
17 */
18
19 static char help[] = "Solves 3D Laplacian using multigrid.\n\n";
20
21 #include <petscksp.h>
22 #include <petscdm.h>
23 #include <petscdmda.h>
24
25 extern PetscErrorCode ComputeMatrix(KSP,Mat,Mat,void*);
26 extern PetscErrorCode ComputeRHS(KSP,Vec,void*);
27 extern PetscErrorCode ComputeInitialGuess(KSP,Vec,void*);
28
29 int main(int argc,char **argv)
30 {
31   KSP            ksp;
32   PetscReal      norm;
33   DM             da;
34   Vec            x,b,r;
35   Mat            A;
36   PetscLogDouble t1, t2;
37   PetscInt mx, my, mz;
38   PetscMPIInt rank;
39
40   PetscCall(PetscInitialize(&argc,&argv,(char*)0,help));
41   PetscCall(PetscTime(&t1));
42   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
43
44   PetscCall(KSPCreate(PETSC_COMM_WORLD,&ksp));
```

```
45   PetscCall(DMDACreate3d(PETSC_COMM_WORLD,DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
         DM_BOUNDARY_NONE,DMDA_STENCIL_STAR,7,7,7,PETSC_DECIDE,PETSC_DECIDE,PETSC_DECIDE
         ,1,1,0,0,0,&da));
46   PetscCall(DMSetFromOptions(da));
47   PetscCall(DMSetUp(da));
48   PetscCall(KSPSetDM(ksp,da));
49   PetscCall(KSPSetComputeInitialGuess(ksp,ComputeInitialGuess,NULL));
50   PetscCall(KSPSetComputeRHS(ksp,ComputeRHS,NULL));
51   PetscCall(KSPSetComputeOperators(ksp,ComputeMatrix,NULL));
52   PetscCall(DMDAGetInfo(da,0,&mx,&my,&mz,0,0,0,0,0,0,0,0));
53   PetscCall(DMDestroy(&da));
54
55   PetscCall(KSPSetFromOptions(ksp));
56   PetscCall(KSPSolve(ksp,NULL,NULL));
57   PetscCall(KSPGetSolution(ksp,&x));
58   PetscCall(KSPGetRhs(ksp,&b));
59   PetscCall(VecDuplicate(b,&r));
60   PetscCall(KSPGetOperators(ksp,&A,NULL));
61
62   PetscCall(MatMult(A,x,r));
63   PetscCall(VecAXPY(r,-1.0,b));
64   PetscCall(VecNorm(r,NORM_2,&norm));
65   PetscCall(PetscPrintf(PETSC_COMM_WORLD, "on %d x %d x %d grid:  error |u-uexact|_inf = %g
         \n", mx, my, mz, norm));
66
67   PetscCall(VecDestroy(&r));
68   PetscCall(KSPDestroy(&ksp));
69   PetscCall(PetscTime(&t2));
70   PetscCall(PetscPrintf(PETSC_COMM_SELF, "Processor %d took %f CPU seconds\n", (int)rank, (
         t2-t1)));
71   PetscCall(PetscFinalize());
72   return 0;
73 }
74
75 PetscErrorCode ComputeRHS(KSP ksp,Vec b,void *ctx)
76 {
77   PetscInt       i,j,k,mx,my,mz,xm,ym,zm,xs,ys,zs;
78   DM             dm;
79   PetscScalar    Hx,Hy,Hz,HxHydHz,HyHzdHx,HxHzdHy;
80   PetscScalar    ***barray;
81
82   PetscFunctionBeginUser;
83   PetscCall(KSPGetDM(ksp,&dm));
84   PetscCall(DMDAGetInfo(dm,0,&mx,&my,&mz,0,0,0,0,0,0,0,0));
85   Hx = 1.0 / (PetscReal)(mx-1);
86   Hy = 1.0 / (PetscReal)(my-1);
87   Hz = 1.0 / (PetscReal)(mz-1);
88   HxHydHz = Hx*Hy/Hz;
89   HxHzdHy = Hx*Hz/Hy;
90   HyHzdHx = Hy*Hz/Hx;
91   PetscCall(DMDAGetCorners(dm,&xs,&ys,&zs,&xm,&ym,&zm));
92   PetscCall(DMDAVecGetArray(dm,b,&barray));
93
94   for (k=zs; k<zs+zm; k++) {
95     for (j=ys; j<ys+ym; j++) {
96       for (i=xs; i<xs+xm; i++) {
97         if (i==0 || j==0 || k==0 || i==mx-1 || j==my-1 || k==mz-1) {
98           barray[k][j][i] = 2.0*(HxHydHz + HxHzdHy + HyHzdHx);
99         } else {
100          barray[k][j][i] = Hx*Hy*Hz;
101        }
102      }
```

```
    }
  }
  PetscCall(DMDAVecRestoreArray(dm,b,&barray));
  PetscFunctionReturn(0);
}

PetscErrorCode ComputeInitialGuess(KSP ksp,Vec b,void *ctx)
{
  PetscFunctionBeginUser;
  PetscCall(VecSet(b,0));
  PetscFunctionReturn(0);
}

PetscErrorCode ComputeMatrix(KSP ksp,Mat jac,Mat B,void *ctx)
{
  DM             da;
  PetscInt       i,j,k,mx,my,mz,xm,ym,zm,xs,ys,zs;
  PetscScalar    v[7],Hx,Hy,Hz,HxHydHz,HyHzdHx,HxHzdHy;
  MatStencil     row,col[7];

  PetscFunctionBeginUser;
  PetscCall(KSPGetDM(ksp,&da));
  PetscCall(DMDAGetInfo(da,0,&mx,&my,&mz,0,0,0,0,0,0,0,0,0));
  Hx      = 1.0 / (PetscReal)(mx-1); Hy = 1.0 / (PetscReal)(my-1); Hz = 1.0 / (PetscReal)(
      mz-1);
  HxHydHz = Hx*Hy/Hz; HxHzdHy = Hx*Hz/Hy; HyHzdHx = Hy*Hz/Hx;
  PetscCall(DMDAGetCorners(da,&xs,&ys,&zs,&xm,&ym,&zm));

  for (k=zs; k<zs+zm; k++) {
    for (j=ys; j<ys+ym; j++) {
      for (i=xs; i<xs+xm; i++) {
        row.i = i; row.j = j; row.k = k;
        if (i==0 || j==0 || k==0 || i==mx-1 || j==my-1 || k==mz-1) {
          v[0] = 2.0*(HxHydHz + HxHzdHy + HyHzdHx);
          PetscCall(MatSetValuesStencil(B,1,&row,1,&row,v,INSERT_VALUES));
        } else {
          v[0] = -HxHydHz;col[0].i = i; col[0].j = j; col[0].k = k-1;
          v[1] = -HxHzdHy;col[1].i = i; col[1].j = j-1; col[1].k = k;
          v[2] = -HyHzdHx;col[2].i = i-1; col[2].j = j; col[2].k = k;
          v[3] = 2.0*(HxHydHz + HxHzdHy + HyHzdHx);col[3].i = row.i; col[3].j = row.j; col
              [3].k = row.k;
          v[4] = -HyHzdHx;col[4].i = i+1; col[4].j = j; col[4].k = k;
          v[5] = -HxHzdHy;col[5].i = i; col[5].j = j+1; col[5].k = k;
          v[6] = -HxHydHz;col[6].i = i; col[6].j = j; col[6].k = k+1;
          PetscCall(MatSetValuesStencil(B,1,&row,7,col,v,INSERT_VALUES));
        }
      }
    }
  }
  PetscCall(MatAssemblyBegin(B,MAT_FINAL_ASSEMBLY));
  PetscCall(MatAssemblyEnd(B,MAT_FINAL_ASSEMBLY));
  PetscFunctionReturn(0);
}

/*TEST

   test:
      nsize: 4
      args: -pc_type exotic -ksp_monitor_short -ksp_type fgmres -mg_levels_ksp_type gmres -
          mg_levels_ksp_max_it 1 -mg_levels_pc_type bjacobi
      output_file: output/ex45_1.out

```

```
162    test:
163       suffix: 2
164       nsize: 4
165       args: -ksp_monitor_short -da_grid_x 21 -da_grid_y 21 -da_grid_z 21 -pc_type mg -
              pc_mg_levels 3 -mg_levels_ksp_type richardson -mg_levels_ksp_max_it 1 -
              mg_levels_pc_type bjacobi
166
167    test:
168       suffix: telescope
169       nsize: 4
170       args: -ksp_type fgmres -ksp_monitor_short -pc_type mg -mg_levels_ksp_type richardson
              -mg_levels_pc_type jacobi -pc_mg_levels 2 -da_grid_x 65 -da_grid_y 65 -da_grid_z
              65 -mg_coarse_pc_type telescope -
              mg_coarse_pc_telescope_ignore_kspcomputeoperators -
              mg_coarse_pc_telescope_reduction_factor 4 -mg_coarse_telescope_pc_type mg -
              mg_coarse_telescope_pc_mg_galerkin pmat -mg_coarse_telescope_pc_mg_levels 3 -
              mg_coarse_telescope_mg_levels_ksp_type richardson -
              mg_coarse_telescope_mg_levels_pc_type jacobi -mg_levels_ksp_type richardson -
              mg_coarse_telescope_mg_levels_ksp_type richardson -ksp_rtol 1.0e-4
171
172    test:
173       suffix: telescope_2
174       nsize: 4
175       args: -ksp_type fgmres -ksp_monitor_short -pc_type mg -mg_levels_ksp_type richardson
              -mg_levels_pc_type jacobi -pc_mg_levels 2 -da_grid_x 65 -da_grid_y 65 -da_grid_z
              65 -mg_coarse_pc_type telescope -mg_coarse_pc_telescope_reduction_factor 2 -
              mg_coarse_telescope_pc_type mg -mg_coarse_telescope_pc_mg_galerkin pmat -
              mg_coarse_telescope_pc_mg_levels 3 -mg_coarse_telescope_mg_levels_ksp_type
              richardson -mg_coarse_telescope_mg_levels_pc_type jacobi -mg_levels_ksp_type
              richardson -mg_coarse_telescope_mg_levels_ksp_type richardson -ksp_rtol 1.0e-4
176
177 TEST*/
```

Listagem B.1: Código completo para o problema de Poisson tridimensional utilizando o método de diferenças finitas com PETSc

Apêndice C

Makefile para instalação do PETSc

```
#!/bin/bash
.SILENT:
.DEFAULT_GOAL := all
HOME_DIR?=${shell cd ~ && pwd}
GIT_DIR=${HOME_DIR}/Git
PETSC_DIR=${GIT_DIR}/petsc/
PETSC_ARCH=arch-linux-c-opt

all: createfolder dependencies clonepetsc instpestc

createfolder:
    (cd ~; \
    echo HOME_DIR=${HOME_DIR}; \
    echo GIT_DIR=${GIT_DIR}; \
    echo PETSC_DIR=${PETSC_DIR}; \
    if [ ! -d ${GIT_DIR} ]; then mkdir -p ${GIT_DIR}; fi)

dependencies:
    sudo apt-get install git -y
    sudo apt-get install gcc -y
    sudo apt-get install make -y

clonepetsc: createfolder dependencies
    (cd ${GIT_DIR}; \
    if [ ! -d ${PETSC_DIR} ]; then \
        git clone -b release https://gitlab.com/petsc/petsc.git petsc --depth 1; \
    fi)

instpestc: clonepetsc
    (cd ${PETSC_DIR}; \
        ./configure --download-mpich --with-debugging=no --download-f2cblaslapack=1; \
        make PETSC_DIR=${PETSC_DIR} PETSC_ARCH=${PETSC_ARCH} all; \
        make PETSC_DIR=${PETSC_DIR} PETSC_ARCH=${PETSC_ARCH} check)
```

Listagem C.1: Código makefile para fazer a download, instalação e configuração correta do PETSc

Apêndice D

Makefile para lançar cálculos

```
#!/bin/bash
.SILENT:
.DEFAULT_GOAL := all
HOME_DIR=${HOME}
GIT_DIR=${HOME_DIR}/Git
PETSC_DIR=${GIT_DIR}/petsc
# PETSC_ARCH=arch-linux-c-debug
PETSC_ARCH=arch-linux-c-opt
MPIEXEC=${PETSC_DIR}/${PETSC_ARCH}/bin/mpiexec
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
CFLAGS += -pedantic -std=c99
pmax := 8
FOLDER?=${shell pwd}/results

all: run distclean

compilepoisson2D_ksp: poisson2D_ksp.o
    -${CLINKER} -o poisson2D_ksp poisson2D_ksp.o ${PETSC_LIB}
    ${RM} poisson2D_ksp.o


compilepoisson3D_ksp: poisson3D_ksp.o
    -${CLINKER} -o poisson3D_ksp poisson3D_ksp.o ${PETSC_LIB}
    ${RM} poisson3D_ksp.o

run2Dksp: createfolder compilepoisson2D_ksp
    { \
    set -e ;\
    code="poisson2D_ksp";\
    echo "For code $${code}";\
    for nx in 50 63 64 65 100 128 129 140 150 175 200 225 256 257 300 325 350 400 450 500
        512 513 550 600 650 700 750 850 950 1000 1100 1250 1300 1400 1500 2000 3000 5000
        7500 10000; do \
        for p in {1..${pmax}}; do \
            ny=$$nx; \
            echo "    With $${p} processors and mesh $${nx} x $${ny}"; \
            filename="${FOLDER}/$${code}-p$${p}-mesh$${nx}x$${ny}.txt"; \
            if [ ! -e $${filename} ]; then \
                executeone="./$${code} -da_grid_x $${nx} -da_grid_y $${ny} -log_view"; \
                command="${MPIEXEC} -prepend-rank -np $${p} $${executeone}"; \
                { \
                    $${command} | cat >> $${filename}; \
                } 2>> $${filename}; \
```

```
            fi ; \
        done \
    done \
    }

run3Dksp: createfolder compilepoisson3D_ksp
    { \
    set -e ;\
    code="poisson3D_ksp";\
    echo "For code ${code}";\
    for nx in 5 10 12 15 16 17 18 20 25 30 32 33 35 40 45 50 63 64 65 75 100 127 128 129
        150 180 200 230 250 255 256 257 300 320 350 400 511 512 513; do \
        for p in {1..${pmax}}; do \
            ny=$nx; \
            nz=$nx; \
            echo "    With ${p} processors and mesh ${nx} x ${ny} x ${nz}"; \
            filename="${FOLDER}/${code}-p${p}-mesh${nx}x${ny}x${nz}.txt"; \
            if [ ! -e ${filename} ]; then \
                executeone="./${code} -da_grid_x ${nx} -da_grid_y ${ny} -da_grid_z ${nz
                    } -log_view"; \
                command="${MPIEXEC} -prepend-rank -np ${p} ${executeone}"; \
                { \
                    ${command} | cat >> ${filename}; \
                } 2>> ${filename}; \
            fi ; \
        done \
    done \
    }

createfolder:
    echo HOME_DIR=${HOME_DIR}
    echo GIT_DIR=${GIT_DIR}
    echo PETSC_DIR=${PETSC_DIR}
    echo PETSC_ARCH=${PETSC_ARCH}
    echo MPIEXEC=${MPIEXEC}
    echo FOLDER=${FOLDER}
    if [ ! -d ${FOLDER} ]; then mkdir -p ${FOLDER}; fi


pythonremovefiles: run2Dksp run3Dksp
    python removenonusefulfiles.py

renamefolder:
    python renamefolder.py

run: createfolder pythonremovefiles run2Dksp run3Dksp

distclean:
    @rm -f *~ poisson *tmp
```

Listagem D.1: Código makefile para lançar diversos calculos utilizando os programas poisson2D_ksp.c e poisson3D_ksp.c variando o número de malha n e número de processadores p. Os resultados são automaticamente gravados em pasta *results*