NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

DUARTE FILIPE DE FREITAS

Licenciado em Engenharia Informática

# SUPORTE DE COMPUTAÇÕES OCTAVE INDEPENDENTES EM MULTIPROCESSADORES DE MEMÓRIA PARTILHADA

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
marco, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

# SUPORTE DE COMPUTAÇÕES OCTAVE INDEPENDENTES EM MULTIPROCESSADORES DE MEMÓRIA PARTILHADA

**DUARTE FILIPE DE FREITAS**

Licenciado em Engenharia Informática

**Orientador:** Pedro Abílio Duarte de Medeiros
*Professor Associado, Universidade NOVA de Lisboa - Faculdade de Ciências e Tecnologia*

**Júri**

**Presidente:** Carla Maria Gonçalves Ferreira
*Professora Associada, Universidade NOVA de Lisboa - Faculdade de Ciências e Tecnologia*

**Arguente:** Carlos Jorge de Sousa Gonçalves
*Professor Adjunto, Instituto Superior de Engenharia de Lisboa*

**Orientador:** Pedro Abílio Duarte de Medeiros
*Professor Associado, Universidade NOVA de Lisboa - Faculdade de Ciências e Tecnologia*

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
marco, 2023

**Suporte de Computações Octave Independentes em Multiprocessadores de Memória Partilhada**



Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeX e o modelo NOVAthesis (v6.10.17) [10].

# Resumo

## Resumo

O objetivo deste projeto é desenvolver uma biblioteca para o *Octave* capaz de paralelizar funções em sistemas de memória partilhada, neste caso, o problema principal que a biblioteca está a resolver são algoritmos de otimização de funções sem derivadas.

O *Octave* é *open source*, ou seja, qualquer pessoa pode criar uma biblioteca para qualquer problema que tenha, por isso existem várias, nomeadamente bibliotecas que correm funções *Octave* em paralelo, como a *parallel package*. No entanto, esta biblioteca foi desenvolvida principalmente para paralelizar usando diferentes máquinas em sistemas distribuídos resultando num sistema de paralelização de memória partilhada bastante simples onde apenas se cria os processos, corre-se a função uma vez e devolve-se os resultados.

Portanto para este projeto criei uma biblioteca do zero. Difere da que já existe porque funciona mais como uma *worker pool*, isto é, cria o número requerido de *workers* que ficam inifinitamente à espera de receber novos *jobs*, *jobs*, no contexto deste projeto são um novo conjunto de valores de entrada que os *workers* recebem para correr a função. A biblioteca tem suporte para funções implementadas em *Octave* (funções interpretadas) e funções implementadas em C/C++ (funções compiladas) compiladas em bibliotecas dinâmicas com duas interfaces diferentes, mas usadas da mesma maneira.

Para implementar a biblioteca usei *octfiles* que são pedaços de código C++ compilados com o API do *Octave* e podem ser usados como funções normais no *Octave*. Além disso, usei bibliotecas padrão do C e sistemas do Linux: os *threads* do C++11 e a *system call* `fork` do Linux para a paralelização. Os *threads* foram usados na interface para funções compiladas e os processos para a interface de funções interpretadas.

Para verificar a eficácia e correção da biblioteca modifiquei um algoritmo de otimização *BoostDMS* para, no passo que avalia a função com os vários valores de entrada gerados, sequencialmente, usar a minha biblioteca e correr este passo concorrentemente e corri-o em máquinas com processadores de vários núcleos de computação onde obti resultados positivos. Os resultados mostram uma têndencia a diminuir o tempo de computação do algoritmo com o aumento de *workers* utilizados maximizando quando o número de

*workers* é igual ou superior ao número valores gerados para analisar.

**Palavras-chave:** Octave, Pralelização, C++, Workers

# Abstract

The objective of this project is to develop a library for *Octave* capable of parallelizing functions on shared memory systems. In this case, the main problem that the library is addressing is optimization algorithms for functions without derivatives.

Octave is open source, which means anyone can create a library for any problem they may have, which is why there are several, including libraries that run Octave functions in parallel, such as the 'parallel' package. However, this library was mainly developed to parallelize using different machines in distributed systems, resulting in a very simple shared memory parallelization system where processes are created, the function is run once, and the results are returned.

Therefore, for this project, I created a library from scratch. It differs from what already exists because it works more like a worker pool, that is, it creates the required number of workers that are infinitely waiting to receive new jobs. In the context of this project, jobs are a new set of input values that workers receive to run the function. The library supports functions implemented in *Octave* (interpreted functions) and functions implemented in C/C++ (compiled functions) compiled into dynamic libraries with two different interfaces, but used in the same way.

To implement the library, I used octfiles, which are pieces of C++ code compiled with the Octave API and can be used as regular functions in Octave. In addition to this, I used standard C libraries and Linux systems: C++11 threads and the Linux fork system call, for parallelization. Threads were used in the interface for compiled functions, and processes were used in the interface for interpreted functions.

To verify the effectiveness and correctness of the library, I modified an optimization algorithm named *BoostDMS* to use my library to run the step that evaluates the function with the various generated input values concurrently instead of sequentially. I ran this modified algorithm on machines with processors with multiple computing cores and obtained positive results. The results show a tendency to decrease the computation time of the algorithm with the increase of workers used, maximizing when the number of workers is equal to or greater than the number of generated values to analyze.

**Keywords:** Octave, Parallelization, C++, Workers

# Índice

# Índice de Figuras

# Índice de Listagens

# 1

# Introdução

Neste capítulo vou começar por explicar o meu raciocínio para escolher este projecto. Em seguida explicarei o objectivo do trabalho, ou seja, o que exactamente vou fazer e porquê. E finalmente, como irei abordar o problema. Explicarei que ferramentas, *frameworks* e metodologias irei utilizar durante este projecto.

## 1.1 Razão da escolha

Este projeto surge da colaboração com o Departamento de Matemática que desenvolveu algoritmos de otimização, implementados em *Matlab*. Estes algoritmos realizam várias computações independentes de funções matemáticas que seriam mais eficientes se realizadas em paralelo. Por isso foi necessário desenvolver uma biblioteca para o *Octave* com uma interface simples e intuitiva capaz disso.

Pessoalmente, estou muito interessado na forma como as ferramentas que nós programadores utilizamos diariamente foram desenvolvidas. Quando ouço falar de uma *framework* que temos de utilizar para um projecto, começo sempre a perguntar-me como funcionam, como é que na raíz de tudo isto sou capaz de enviar dados para uma base de dados apenas através da criação de um objecto. Desde que comecei engenharia informática não gostava de utilizar bibliotecas e ferramentas que não compreendia, começava sempre a pensar como é que funcionam. Por causa disto, fiquei muito interessado na programação de baixo nível.

## 1.2 Problema

O objectivo deste projecto é desenvolver uma biblioteca para *Octave* capaz de realizar cálculos em paralelo através da utilização de múltiplos processadores que partilham os dados, fortemente inspirada na MATLAB's Parallel Computing Toolbox (PCTools).

*Octave* é uma linguagem de *open source*, portanto é grátis e muito semelhante ao MATLAB, o que significa que foi concebido para, facilmente, fazer cálculos e transformações de matrizes, calcular diferentes funções e também traçar estas funções em gráficos.

Muitas das funções matemáticas têm grandes domínios, o que significa que é lento calculá-las sequencialmente. Também, para estudar estas funções é necessário calcular os contradomínios de vários conjuntos diferentes de domínios, o que por sua vez gera novos conjuntos a analisar. Sabendo isto, é do nosso interesse executar estas funções com vários conjuntos de domínios em paralelo, recolher os diferentes resultados, estudá-los, gerar novos conjuntos e executá-los novamente em paralelo.

Para isso, vou desenvolver uma biblioteca capaz de distribuir a mesma função por vários *threads*.

## 1.3 Abordagem

Vou implementar uma biblioteca do zero para ser utilizada no *Octave*. A biblioteca poderá receber uma função, originalmente implementada no *Octave*, e criar um determinado número de *threads*, que ficarão infinitamente a executar a mesma função à medida que vão recebendo novos conjuntos de dados.

### 1.3.1 Objetivos concretos

O principal objectivo da biblioteca será o de executar as mesmas funções do *Octave* em diferentes *threads* como mencionado na secção 1.3. No entanto, pretendo expandir sobre isto.

Para começar, os *threads* poderão continuar a executar a mesma função indefinidamente enquanto vão recebendo diferentes conjuntos de argumentos. Isto significa que os *threads*serão capazes de executar essa função várias vezes sem a necessidade de gerar os *threads* todas as vezes.

Com a ajuda desta biblioteca, vou executar um algoritmo de optimização de uma função matemática que recebe uma lista com mais de 3 números, como domínio, mas esta função vai ser avaliada mais que uma vez, cada vez com um conjunto de números diferente, pois usando os resultados dp primeiro domínio geram-se mais domínios para avaliar a função. Além disso, o algoritmo é optimizado de forma a que seja altamente provável que os domínios gerados primeiro tenham uma maior probabilidade de gerar os resultados relevantes.

## 1.4 Conteúdo

Primeiro, no capítulo 2 vão ser descritos os tipo de máquinas para qual este projeto será desenvolvido, modelos de programação paralela como memória partilhada e troca de mensagens, e bibliotecas de paralelização para funções do *Octave*.

Depois, no capítulo 3 serão mencionadas as funcionalidades da biblioteca, qual a sua interface, que tipo de funções são suportadas bem como o método de utilização

complementado com exemplos. Seguidamente, vão ser apresentadas quais ferramentas usadas no desenvolvimento, o porquê da escolha e o seu impacto no projeto.

Posteriormente, uma secção com o código realizado e a explicar a lógica para conseguir implementar a interface anteriormente descrita, no capítulo seguido por testes a algoritmos de otimização comparando o tempo de execução ao usar a biblioteca com a não utilização da mesma. Os resultados dos testes serão apresentados através de gráficos e discutidos, se estão dentro do esperado e se fazem sentido, nos capítulos 4 e 5, respetivamente.

Por fim, as conclusões e trabalhos futuros, i.e, se o projeto está dentro das expetativas e maneiras de o melhorar, novas características e melhorar as que já existem, no capítulo 6.

# 2

# Estado de Arte

Neste capítulo começarei por cobrir os conhecimentos básicos necessários para depois explicar as formas de paralelismo que já existem no *Octave*, porquê de serem insuficientes e porque é necessário criar uma biblioteca nova do zero.

## 2.1 Matlab & Octave

MATLAB (uma abreviação de "MATrix LABoratory") é uma linguagem de programação multiparadigma proprietária e um ambiente de computação numérica desenvolvido pela MathWorks. Foi desenvolvido principalmente para cálculos numéricos, nomeadamente manipulações de matrizes e desenho de funções, mas também permite a implementação de algoritmos, criação de interfaces, e interação com programas escritos em outras linguagens.

MATLAB oferece aos seus utilizadores várias ferramentas, i.e *toolboxes* que são uma coleção de funções construídas sobre o ambiente de programação MATLAB. O MathWorks oferece várias *toolboxes* para ajudar em diferentes campos científicos, como Inteligência Artificial, ciência de dados, suporte a bases de dados, etc. Uma delas é o *PCTools*, que é a única maneira de usar paralelismo no MATLAB. Para ter acesso a essas Toolboxes e ao próprio MATLAB é necessária uma licença paga.

GNU Octave [8] é também uma linguagem de programação de alto nível inspirada no MATLAB, uma vez que é utilizada para os mesmos fins, no entanto é grátis e *open source*. Sendo *open source*, é possível qualquer pessoa desenvolver uma biblioteca para resolver os seus próprios problemas. As bibliotecas podem ser vistas da mesma forma que as Toolboxes, pois são um conjunto de funções utilizadas para optimizar tarefas.

## 2.2 Hardware

Nesta secção vão ser descritas vários tipos de máquinas e hardware para os quais esta biblioteca será desenvolvida. A biblioteca será orientada para sistemas com uma única máquina com uma ou mais unidades multiprocessadoras.

### 2.2.1 Processadores MP

Este projecto será centrado no paralelismo de uma única máquina, ou seja, multiprocessadores com memória partilhada. Os CPU tornaram-se muito mais rápidos ao longo dos anos em termos de velocidade de *clock* de um único *thread*, mas também no número de núcleos por CPU. Hoje em dia, os CPU mais potentes e mais recentes têm mais de cem núcleos e como estão num único CPU têm de partilhar o espaço de memória, isto deu lugar a problemas sobre como permitir aos vários núcleos acederem aos espaços de memória para os seus programas de forma eficiente. Até que finalmente, foi desenvolvida uma arquitectura que se tornou o padrão de CPU modernos: *Uniform Memory Access* (UMA).

UMA é uma arquitetura utilizada em processadores paralelos em que os diferentes núcleos acedem uniformemente à memória, o que significa que o tempo de acesso a uma determinada localização de memória é independente de qual núcleo tenta acedê-la

No entanto, apenas um núcleo pode aceder ao espaço de memória de cada vez, o que serializaria o processo quando pretendemos paralelização. Por isso, também é necessário ter uma cache entre a memória e os núcleos para que estes possam primeiro verificar a cache e só quando necessário aceder à memória. Isto reduz imensamente a probabilidade de dois núcleos acederem à memória ao mesmo tempo, mas quando isso acontece, existe um controlador que decide qual núcleo acede primeiro à memória. Esta arquitetura é ilustrada na Figura 2.1, os diferentes núcleos do CPU acedem ao mesmo espaço de memória através de um controlador que controla a ordem.

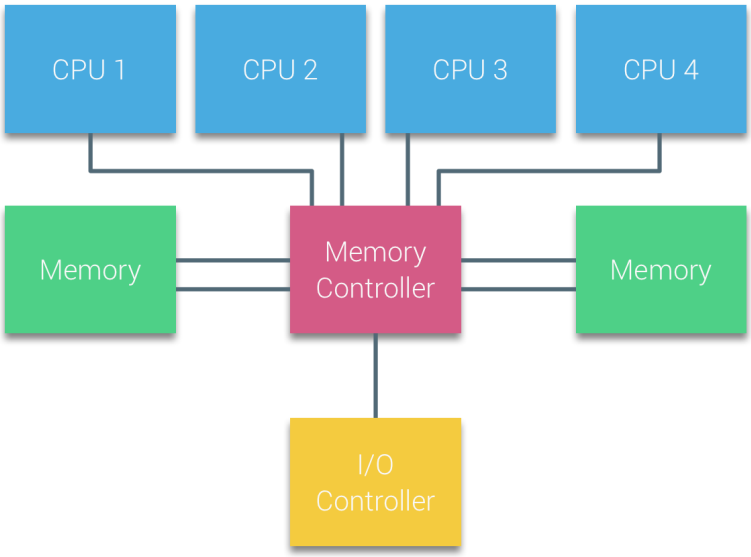



Figura 2.1: Arquitetura UMA [6]

À medida que foi necessário uma maior velocidade de processamento para paralelização, foi desenvolvida uma nova arquitetura: *Non-Uniform Memory Access* (NUMA). Esta arquitetura pode ser vista na Figura 2.2. Basicamente, são vários processadores UMA juntos.

Na figura, podemos ver diferentes nós NUMA, cada um pode ser visto como um CPU completo e cada nó tem o seu próprio espaço de memória. Apesar disso, cada nó vê o espaço de memória completo, não apenas o seu próprio, o que significa que por vezes um nó pode precisar de aceder à memória de outro nó, o que, como seria de esperar, levaria muito mais tempo a aceder. Assim, nesta arquitetura, software é desenvolvido de forma diferente para que os programas sejam salvos na memória mais próxima do CPU que os executa, de modo a reduzir a necessidade de aceder à memória de outro nó.

Esta arquitetura é utilizada, por exemplo, em máquinas que possuem dois CPU físicos em diferentes *sockets*, no entanto, existem hoje em dia processadores que compostos por dois processadores mais pequenos que utilizam a arquitetura NUMA, em vez da UMA. Para o contexto deste projeto, a biblioteca será desenvolvida para arquiteturas UMA padrão.

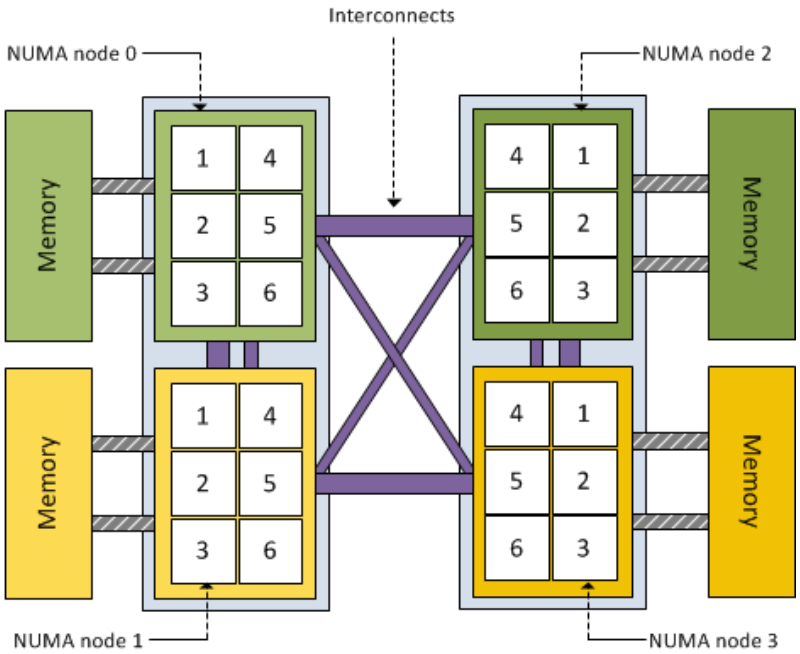



Figura 2.2: Arquitetura NUMA [5]

Para acelerar ainda mais o processamento e a paralelização, existe também *General-purpose computing on graphics processing units* ou GPGPU, quando abreviadamente. As GPU, que são unidades de processamento especificamente fabricadas e geralmente utilizadas para cálculos gráficos de computadores, podem também ser utilizadas em cálculos que são normalmente tratados pelo CPU.

Essencialmente, um pipeline GPGPU é uma espécie de processamento paralelo entre uma ou mais GPUs e CPUs que analisa dados como se estivessem em formato de imagem ou outro formato gráfico. Embora as GPUs funcionem com frequências mais baixas, normalmente têm bastantes mais núcleos. Assim, as GPUs podem processar muito mais imagens e dados gráficos por segundo do que um CPU tradicional. A migração de dados para formato gráfico e, em seguida, a utilização da GPU para analisá-los pode criar um

aumento de velocidade de computação significativo.

## 2.3 Modelos de programação paralela

Este capítulo cobrirá alguns modelos e técnicas de programação utilizados na programação concorrente. Primeiro serão mencionadas as técnicas mais comuns tais como *forks*, *OpenMP* e a biblioteca *pthread* em C/C++, as suas vantagens e desvantagens e se são relevantes para o projecto. Depois, uma breve descrição sobre a passagem de mensagens como método de paralelização, embora apenas superficialmente, uma vez que não será muito relevante na elaboração do projeto.

### 2.3.1 Memória partilhada

Existem muitas bibliotecas construídas com o objectivo de utilizar os diferentes núcleos dos CPU's atuais de forma eficiente para paralelizar algumas computações. Uma API muito conhecida é OpenMP para C e C++, esta API é muito simples de usar transformando o que seria um programa extensivo de muitas linhas para paralelizar um loop, numa única linha..

Este método de programação paralela que usa *threads* é a mais comum e intuitiva de programar multiprocessadores de memória partilhada. No entanto existem outros métodos de o fazer, como por exemplo passagem de mensagens que será brevemente mencionado, na seccção 2.3.2.

OpenMP [15] é uma implementação de multithreading, um método de paralelização onde um *thread* primário cria um número especificado de *sub-threads* e o sistema divide uma tarefa entre eles. Os *threads* executam em simultâneo, com o ambiente de *runtime* a atribuir *threads* a diferentes processadores.

Outra biblioteca comum utilizada para programar com threads em C e C++ é a pthreads. Em comparação com a API *OpenMP*, esta biblioteca é muito mais verbosa e muito mais difícil de programar com segurança, porque é menos abstraída. Os *threads* têm de ser criados e fechados manualmente, assim como a gestão dos *locks*, ou seja, a obtenção e libertação dos *locks*. Isto abre muitas possibilidades de gerar *deadlocks* que o OpenMP seria capaz de impedir. Contudo, por ser mais difícil, não significa que seja pior, porque o facto de ser só o essencial dá mais flexibilidade e mais opções.

Este método de programação paralela utilizando threads é a forma mais comum e intuitiva de programação de multiprocessadores de memória partilhada. Contudo, existem outros métodos para o fazer, tais como a passagem de mensagens.

### 2.3.2 Mensagens

As mensagens [12] são também uma forma de programação de cálculos paralelos com memória partilhada, contudo o método de acesso aos dados é ligeiramente diferente. Como o nome indica, os processos interagem entre si através da troca de mensagens. Em

vez de acederem às variavéis que precisam diretamente, um processo envia os dados para outro processo.

Desta forma, cada processo, bem como o processo principal, sabem o que cada instância da função está a fazer. Conseguem também verificar se um processo terminou, trocando mensagens entre si, embora isto signifique que os cálculos paralelos não são independentes uns dos outros.

## 2.4 Paralelismo no *Octave*

GNU Octave é software *open source*, o que significa que qualquer pessoa no mundo pode desenvolver as bibliotecas que quiser, portanto, há uma série delas que permitem a paralelização de código *Octave*, são elas: *parallel*, *MPI* e *dragonfly*.

A biblioteca *parallel* [9] tem uma série de funções que podem ser utilizadas em scripts *Octave*. Tem funções para execução local em memória partilhada, bem como funções para configurar a paralelização sobre *clusters* com memória distribuída.

No contexto deste projecto, uma vez que está mais centrado na execução local em multiprocessadores com memória partilhada, vou apenas cobrir as funcionalidades de execução local da biblioteca *parallel*. A biblioteca acrescenta duas funções principais para permitir a paralelização local, são elas:

- `pararrayfun`: argumentos dados sob forma de *array*.
- `paracellfun`: argumentos dados sob forma de *cell array*.

Existe uma terceira função chamada `parcellfun_set_nproc` que é responsável por estabelecer o número padrão de processos de paracellfun. O número de processos utilizados é determinado pelo primeiro argumento de `paracellfun`. Se necessário, é possível gerar novos processos, mas nunca será removido nenhum deles, mesmo que não sejam todos utilizados.

A função `pararrayfun` tem três argumentos, o primeiro é o número de processos, que está limitado ao número de núcleos do cpu. O segundo argumento é o nome da função *Octave* (.m) a executar nos diferentes processos, por último, recebe um array, este array tem os argumentos da função. Além disso, existe também a função paracellfun, cuja principal diferença é que recebe um *cell array* com os argumentos para a função em vez de um vetor como no `paracellfun`.

```
1 fun = @(x) x^2;
2
3 vector_x = 1:10;
4
5 vector_y = pararrayfun(nproc, fun, vector_x)
```

Listing 1: Exemplo de uso da função `pararrayfun`.

```
Output:
parcellfun: 10/10 jobs done

vector_y =

     1     4     9    16    25    36    49    64    81   100
```

Listing 2: Resultados do exemplo

Olhando para o código de exemplo na listagem 1, cria um array com todos os números de um a dez e a função passada como argumento para `pararrayfun` calcula o resultado com cada um dos valores do array.

Para testar as funcionalidades da biblioteca fiz um programa simples do método Monte Carlo para calcular uma aproximação do Π. Para este método existe um quadrado de 1 por 1 e dentro dele, um círculo com um raio de 1, de seguida é gerada uma determinada quantidade de pontos escolhendo aleatoriamente as coordenadas x e y, finalmente divide-se o número de pontos gerados que estão dentro do círculo pelo número total de pontos gerados e multiplica-se esse número por quatro. Quanto mais números forem gerados, mais próximo o resultado de estará do valor do Π.

```
1  function pi = approx_pi_parallel(total_points, n_processors)
2
3
4    vector = create_array(total_points/n_processors, n_processors)
5
6    points_in = pararrayfun(n_processors, @generate_points, vector);
7    pi = (sum(points_in)/total_points)*4;
8    disp(points_in)
9    disp(pi)
10 endfunction
```

Listing 3: Código do método de Monte Carlo.

O código acima mostra o método Monte Carlo, primeiro crio a matriz para a `pararrayfun` dividindo o número de pontos totais pelo número de processos para obter o número de pontos que cada processador irá gerar, e crio uma matriz do tamanho do número de processos com esse número repetido. Depois de cada processo gerar os pontos, devolve o número de pontos dentro do círculo. Uma vez que o `pararrayfun` mapeia a função que recebe por cada valor no array, a soma dos valores no array devolvido é o número de pontos dentro do círculo, dividido pelo número total de pontos e multiplicado por quatro, obtenho a aproximação de Π.

Desta forma, a quantidade total de pontos é dividida pelo número de processos e cada processo gera uma fracção dos pontos, acelerando bastante o processo. Depois de testar, obtive uma velocidade de aproximadamente 2 ao duplicar o número de processadores, como esperado.

## 2.5 Discussão

O sistema a construir tem como principal objetivo suportar a execução de dos algoritmos de otimização sem derivadas usados no projeto BoostDFO. Esses algoritmos seguem uma estrutura iterativa em que, em cada passo, são efetuadas C avaliações da função objetivo; o número C é sempre o mesmo e as C computações são independentes umas das outras.

Em relação à condição de finalização de um passo, os algoritmos podem usar duas estratégias:

- completa: o passo só termina quando todas as C computações devolveram o resultado
- oportunista: perante um resultado concreto de uma avaliação, as avaliações pendentes são canceladas

Quanto à ordem pela qual as soluções são recolhidas, há também duas alternativas:

- aleatória: os resultados das avaliações são recolhidas à medida que vão estando disponíveis
- ordenada: os resultados das avaliações são recolhidas pela ordem pela qual foram lançadas

O sistema também deverá suportar diferentes estratégias para atribuir as C computações aos P processadores disponíveis em situações em C é maior do P. O sistema deverá suportar a escolha da estratégia pelo desenvolvedor dos algoritmos.

Nenhum dos sistemas analisados suporta todos estes aspetos pelo que se irá desenvolver uma biblioteca de raiz. Esta opção tem também a vantagem de permitir a construção de uma biblioteca customizada para o suporte de algoritmos de otimização sem derivadas e também a experimentação de diferentes estratégias para o suporte das duas formas de obter resultados e do lançamento das computações.

# 3

# Organização da Solução

Este capitulo destina-se a primeiro por explicar as funcionalidades implementadas, como as utilizar e também se estas estão de acordo com o planeado para este projeto. Depois de explicar as funcionalidades, serão descritas as ferramentas utilizadas para o conseguir.

## 3.1 Funcionalidades

A objetivo principal da biblioteca é receber uma função através do *Octave* e corrê-la em paralelo. No entanto, os algoritmos para os quais este projeto foi desenvolvido requerem que esta função seja executada várias vezes com vários tipos de dados de entrada diferentes, por isso não seria muito eficiente estar a criar trabalhadores, ou *workers* (O que corre a função recebida), todas as vezes que se quer correr a função outra vez com dados diferentes, ou seja o desejado para este trabalho será criar uma *pool* de *workers* que correm infinitamente à espera de novos dados para correr a função.

A biblioteca tem suporte para dois tipos de funções que pode paralelizar: funções desenvolvidas em C/C++, compiladas como uma biblioteca dinâmica [14], cujo nome da biblioteca terá de ser o mesmo nome do ficheiro e da função a ser usada dentro da biblioteca. E funções interpretadas, desenvolvidas em *Octave*.

Portanto, a biblioteca tem uma interface simples e intuitiva composta por três funções. Uma para criar a *pool* de *workers*, outra para enviar os dados, ou *jobs*, para os *workers* e uma última para os parar. Com algumas diferencas dependendo do tipo de funções que se está a paralelizar.

As funções foram implementadas em C++ que são depois compiladas para octfiles. O octave fornece um compilador que faz *link* da biblioteca *octave* com o código realizado em C++ chamada *mkoctfile*, com esta ferramenta consigo compilar o código que desenvolvi e criar a octfile que o octave reconhece como uma função e é capaz de carregar e correr. Visto que foram desenvolvidas em C++ utilizou-se principalmente bibliotecas *standard* do C++, como os *threads* e os processos através da `fork()` system call. Cada uma destas técnicas foi utilizada para cada um dos tipos de funções suportadas que irei explicar quais e porquê de seguida.

### 3.1.1 Octave

- `spawn_workers`: Função que cria a *pool*. Recebe três argumentos, primeiro o número de workers a serem criados, segundo a função a ser corrida pelos *workers* e terceiro o número de dados de saída da função que é recebida como segundo argumento. A função no segundo argumento é a *function handle* da função a ser corrida pelos *workers*. Isto é o nome da função implementada num ficheiro ".m" precedido de um "@", ou apenas uma variável que guarda a *function handle* através da função do *Octave str2func*.

- `send_data`: Função que envia os dados para a *pool*, recebe um argumento que é uma matriz *Octave* onde cada linha serão os dados de entrada para cada *worker*.

- `stop_workers`: Função para parar os *workers*, não recebe qualquer argumento.

### 3.1.2 C/C++ biblioteca dinâmica

- `spawn_workers_c`: Função que cria a *pool*. Recebe três argumentos, primeiro o número de workers a serem criados, segundo a função a ser corrida pelos *workers* e terceiro o número de dados de saída da função que é recebida como segundo argumento. A função no segundo argumento é uma string com o nome da função que está dentro da biblioteca dinâmica, cujo ficheiro tem a terminação ".so", isto é, uma função que calcula a média e implementada com o nome "average" será compilada para biblioteca dinâmica com o nome "average.so" e a função recebe a string "average".

- `send_data_c`: Função que envia os dados para a *pool*, recebe um argumento que é uma matriz *Octave* onde cada linha serão os dados de entrada para cada *worker*.

- `stop_workers_c`: Função para parar os *workers*, não recebe qualquer argumento.

### 3.1.3 Função interpretada

Função interpretada, a que é implementada em *Octave* e paralelizada através do uso de processos, a fork() system call. Optei por fazer as funções interpretadas através de vários processos em vez de *threads*, porque a instância do interpretador que recebo para correr as funções não é thread-safe. Ou seja, se criasse vários threads com uma cópia do interpretador, e tentasse correr a função, só um *thread* de cada vez é que o podia fazer. Por isso optei por criar vários processos, assim cada processo tem a sua stack e heap independente dos outros, ou seja, cada processo tem o seu interpretador e é capaz de correr as funções interpretadas.

Como a memória dos processos não é partilhada tenho de obter os resultados através do uso de pipes, que é também uma funcionalidade da *standard library* do C/C++.

Ao fazer a biblioteca fui tentando fazer com que as funções recebidas pudessem ser o mais gerais possíveis, que o utilizador não precisasse de modificar as suas funções para poder utilizar a biblioteca, contudo isto não foi possível devido as limitações dos pipes. Não é possível escrever objetos diretamente, só é possível escrever bytes, por isso as funções interpretadas tem de receber unicamente um vetor de doubles como dados de entrada.

Um dos principais objetivos da concorrência e paralelismo é acelerar a computação, mas a função interpretada não corre tão rápido quando comparado com uma função já compilada. Ou seja, paralelizar este tipo de funções vai sempre ter um speedup maior que 1 quando comparado com sequencialmente, como esperado, no entanto, não terá um speedup tão grande caso se conseguisse compilar a função *a priori* e corrê-la paralelamente sem recurso a um interpretador.

### 3.1.4 Função compilada

Função compilada, a que é implementada em C, esta função é paralelizada através do uso de threads da biblioteca do C++11. Para ser possível correr esta função com a biblioteca, é primeiro preciso compilá-la como uma biblioteca dinâmica [7], esta será depois *dynamically loaded* utilizando uma biblioteca que só existe no Linux chamada *dlfcn*.

Para além disso, esta função tem também requisitos prévios em relação aos parâmetros: os parâmetros da função compilada são dois apontadores de doubles, um para os dados de entrada e outro para os dados de saída, bem como dois *ints* que guardam o número de valores em cada um dos apontadores de doubles mencionados.

Como foi mencionado na secção anterior, a função a ser paralelizada ser previamente compilada terá maior *speedup* do que se a função for interpretada. Para isso, encontrei uma maneira de tornar a função interpretada numa função compilada que o interpretador do *Octave* seja capaz de correr, utilizando uma ferramenta, não da minha autoria: *Coder*. O *Coder* consegue pegar em código *Octave* e compilá-lo para uma octfile, que no fundo é código C++ compilado que o interpretador do *Octave* reconhece e consegue correr. Assim consegue-se o *speedup* não só através da paralelização como tambem através da passagem de código interpretado para código compilado.

### 3.1.5 Exemplos

Para demonstrar as funcionalidades em prática, desenvolveu-se um script *Octave* que corre uma função também desenvolvida em *Octave*, que calcula a média dos valores de um vetor, ilustrado na figura 4. Começa por gerar uma matriz com 10 linhas e 12 colunas com números naturais entre 1 e 1000 que serão os dados de entrada da função e mostra no ecrã, chama a função para criar os *workers*, neste exemplo crio 4 deles com a função "average" que devolve um valor de saída e a seguir corre um ciclo onde envia a matriz para os *workers*, mostra os resultados no ecrã e gera uma nova matriz para enviar mais duas

vezes vezes para demonstrar o facto dos *workers* estarem efetivamente sempre à espera de novos valores. Finalmente, pára os *workers*.

```
vetor = randi([1 1000], 10, 12);

spawn_workers(4, @average, 1);

for i=1:3
  send_data(vetor)
  vetor = randi([1 1000], 10, 12);
end

stop_workers();
```

Listing 4: Exemplo simples de uso da biblioteca

Ao correr o *script* obtenho o output ilustrado na listagem número 5:

Para realizar os mesmo exemplo com a interface das funções compiladas seria idêntico.

## 3.2 Ferramentas

### 3.2.1 C++

Para realizar este projeto optei por utilizar C++, principalmente porque apesar de também ser possível com C ou Fortran a realização de *octfiles* tem mais suporte para C++.

Primeiramente, o *Octave* dispõe de uma biblioteca com várias macros e estruturas de dados que tornam o manuseamento de dados entre o *Octave* e o C++ mais simples, nomeadamente as matrizes do *Octave*. Estas bibliotecas serão explicadas em mais detalhe em secções próximas.

Em segundo lugar, C++ tem mais funcionalidades quando comparado com o C que fazem programar mais fácil, nomeadamente *smart pointers*. Os *smart pointer* são, como o nome indica, apontadores, mas que sao capazes de se dealocarem quando nao estao a ser mais utilizados o que impede *memory leaks*. Outra razão, a existência de objetos em C++ permite utilizar as estruturas de dados mencionadas que a biblioteca do *Octave* oferece.

Finalmente, com o C++11 foi introduzida uma biblioteca de threads, esta biblioteca permite-me utilizar threads, que são uma principal parte deste projeto, sem recorrer aos pthreads. Isto é positivo pois *pthreads* são de demasiado baixo nível o que torna o manuseamento dos dados que cada thread recebe bastante mais complicado de lidar. Ao utilizar esta biblioteca do C++ consigo enviar objetos e objetos como referência para os threads enquanto que nos pthreads teria de usar apontadores void.

### 3.2.2 Octave/octfiles

Como já foi mencionado anteriormente, *octfiles* são a base deste projeto, são o que o *Octave* consegue carregar e correr como uma função normal, para além das funções escritas em

*Octave* normal. *Octfiles* são pedaços de código C++ compilados com o API do *Octave* em objetos carregados dinamicamente.

Para compilar estes pedaços de código C++ o *Octave* fornece uma ferramenta chamada *mkoctfile* que o faz utilizando automaticamente os *switches* corretos e incluindo os *paths* para os ficheiros *header* que contém a API do *Octave*. Esta ferramenta para além de realizar os passos necessários para conseguir compilar o código C++ numa *octfile*, também consegue correr os inúmeros comandos dos compiladores mais normalmente usados como o *gcc* e o *g++*, podendo assim substitui-los enquanto se realiza código para uma biblioteca do *Octave*, uma vez que chama o *g++* com as opções que recebe e faz link da API do *Octave* para criar o *octfile*.

Neste API do *Octave* existem duas macros que foram cruciais para este projeto, nomeadamente a *DEFUN_DLD* e a *DEFMETHOD_DLD*. Ambas estas macros permitem criar uma função que é dinamicamente carregada no *Octave*, no entanto têm uma pequena diferença, a segunda macro para além de receber os quatro parâmetros que recebe a primeira: o nome que a função terá no *Octave*, a lista de argumentos para a função, o número que argumentos de saída que é omitido e uma *string* com o texto para ajudar a perceber a função quando o comando "help" é chamado. Recebe também uma instância de um objeto do interpretador do *Octave*. Com este interpretador do *Octave* mais o *handle* de uma função implementada em *Octave* que vem na lista dos argumentos é possível correr uma função *Octave* nos *octfiles*, ou em C++, neste caso.

Com estas macros consigo criar as funções da biblioteca, mencionadas em 3.1, no entanto a API fornece também estruturas de dados que foram úteis na realização deste projeto. Entre eles a mais importante pois é a estrutura de dados que guarda a entrada e a saída das funções criadas a partir das macros, *octave_value_list*. Esta estrutura de dados é uma lista capaz de guardar vários *octave_values* que por sua vez são um valor que pode ser qualquer coisa, desde um double, um integer, uma string até uma lista de um tipo de dados. Estas foram as estruturas mais usadas, no entanto também utilizei uma estrutura de dados que representa uma matriz do *Octave*, esta representação da matriz possui um método capaz de tornar a matriz recebida a partir do *Octave* num apontador de doubles sendo assim mais fácil de mexer dentro da linguagem C++ em conjunção com os pipes e os threads. Outro método útil da matriz é um capaz de transpor a matriz, uma vez que os valores quando passados para um apontador estão organizados por coluna em vez de por linha, por isso transpor a matriz resolve este problema.

Além dessas, o *Octave* também fornece duas macros que criam *smart pointers* do C++ e lidam com a memória em vez do programador. São elas *OCTAVE_LOCAL_BUFFER* e *OCTAVE_LOCAL_BUFFER_INIT*, a diferença entre elas é que a segunda cria o apontador inicializado com um valor.

### 3.2.3 C++11 threads

Os C++11 *threads* são bastantes inspirados nos POSIX *pthreads*, mas simplificam a API enquanto também guarantem segurança de tipos [13].

Com esta biblioteca de *threads* realizei a paralelização das funções compiladas em C/C++, em conjunção com a biblioteca denominada `dlfcn` capaz de aceder a bibliotecas dinâmicas e obter um apontador para uma função dentro desta biblioteca dado o nome.

Os *threads* só por si, não são capazes de lidar com tudo, é preciso certificar-me que estes acedem aos dados corretamente através do uso de variáveis de condição e *mutexes*. Variáveis de condição são primitivas de sincronização que permitem que os *threads* aguardem até que uma condição específica ocorra ou aconteça um *timeout* e *mutexes* são outra primitiva de sincronização para controlar os acessos dos *threads* aos dados em memória partilhada.

### 3.2.4 Linux fork system call

Para paralelizar as funções interpretadas, uma vez que a avaliação da função externa a partir da instância do interpretador recebida através da macro "DEFMETHOD_DLD" não é *thread-safe* tive de recorrer a utilização de processos criados com o `fork()`. Porém a criação destes processos no Linux não é muito mais dispendiosa em termos de recursos e tempo quando comparado com a criação de *threads*.

Para o kernel do Linux não há qualquer diferença entre o que o espaço do utilizador vê como um processo ou um *thread*. Ambos são representados pelas mesmas estruturas de dados e agendados pelo *scheduler* de maneiras semelhantes [3].

No Linux, a principal diferença entre os dois são apenas que os *threads* partilham recursos, particularmente a memória, enquanto que os processos não partilham recursos nenhuns. Ao nível do programador *threads* e processos são criados e manuseados de maneiras diferentes com API's diferentes, nomeadamente *fork* e *wait* para processos e *thread()* e *condition_variables* para *threads*, contudo no fundo destes API's ambos são criados através da chamada ao sistema *clone*.

Assim, utilizar processos para dar a volta a este problema do interpretador não impõe uma maior sobrecarga de recursos de *hardware* nem de tempo comparado com os *threads* usados para as funções compiladas.

No entanto, visto que os processos não partilham recursos tive de usar *pipes* de modo a passar os dados de entrada para cada *worker* bem como obter os dados de saída depois de avaliada a função. Estes pipes são, para o kernel do Linux, ficheiros (representados através de *file descriptors*) e por isso para comunicar entre os processos tenho de escrever bytes nestes ficheiros e não posso passar objetos ou *structs* diretamente. Por este motivo, preciso de passar os dados das estruturas de dos do *Octave* para apontadores de double e de volta às estruturas do *Octave* para passar dados entre os processos.

Para ler e escrever a partir dos pipes utilizam-se duas funções: *read* e *write*, respetivamente. Ambas têm a mesma interface em que recebem o *file descriptor* onde devem realizar

a ação, o apontador com os bytes para ler ou escrever e por fim, a quantidade de bytes. Estas funções podem não completar a ação para todos os bytes mencionados aquando da chamada da função, pelo que então devolvem também a quantidade de bytes lidos ou escritos.

Finalmente, para obter os resultados de cada *worker* tenho de saber quais é que já escreveram os resultados nos respetivos pipes, para conseguir isso utilizo a função *select*. Esta função recebe um set de *file descriptors* e modifica esse set de modo a ter apenas os *file descriptors* que estão prontos a ler ou escrever conforme o necessário.

```
>> example_usage
vetor =

   170   607   424   619   307   592   609   801   729   272   214   304
   100   282   694   325   409   298   688   224   218   736   293   933
   632   344   283   190   652   143    95   576   754   246   925   353
   414   791   288   361   642   535   693   125   353   436   376   363
   576   750   994   476   203   300   466   715   465   993   837   904
   883   242   581   839   137   725   866   339   418   739   431    22
   847   849   523   616   300   225   191   912   466   612   311   847
   189   106   317   613   481   284   457   755   220   821   141   781
   457    16    59   252   969    31   849    45   447   133   499   793
   951    83   639   581   120   474   617   134   111   837   693   912

ans =

   470.67   433.33   432.75   448.08   639.92   518.50   558.25   430.42   379.17   512.6

vetor =

   251   824   527   881   344   242   923   353   366   841    30   248
   692   918   578   184   674   628   125   742   915   116   724   413
   788   741   796   884   782   718   566   238   690   882   634   551
   943   166   147   563   857   407   711    97   721   236   261   833
   283   609   192   398   421   423   975   426    48   155   501   180
   752    10   104   624    43   619    32   872   634   438   168   915
   932   979   766   540   840   203   945   800   587   905    88     9
   590   513    37    48   435   391   849   761   143   246   263   714
   872   950   152   465    79   915   372   341   523   888   645   775
   498   831   667   164   751   622   769    78   765   440   895   867

ans =

   485.83   559.08   689.17   495.17   384.25   434.25   632.83   415.83   581.42   612.2

vetor =

   170   684   430    87   599   110   822   108   325   339   989   268
   221   898   362   419   737   773   923   942    30   644   917   908
   980   885   812   682   540   914   197   818   740   188    58   893
   323   727   233   148   196   237   554   399   833   535   380   269
    38   585   900   414   115   939   855   995   800    81   168   487
   329   145   604   271   318   149   608   849   114   728    14   832
   423   618   373   941   546   318   566   346   341   340   855   366
   803   357   815   810   915   953   303   952   237   643   629   607
   576   339   736   466   476   181   209   472   741   426   439   545
   205   980   760   367   774   610   207   886   394    78   815   669

got terminate signal
got terminate signal
got terminate signal
got terminate signal
```



Listing 5: Exemplo simples de uso da biblioteca

# 4

# Implementação

Este capítulo servirá para explicar as estratégias utilizadas para implementar a interface simples para ambas os tipos de funções suportadas. Será explicado como usei os processos, com os *pipes* e o *select* para conseguir paralelizar funções interpretadas e os *threads* em conjunto com a biblioteca *dlfcn* para implementar a paralelização de funções compiladas.

## 4.1 Funções Interpretadas (Octave)

### 4.1.1 Criar a pool

Para funcções interpretadas, ou funções implementadas em Octave, utilizou-se processos, *pipes* e o *select*, todos da biblioteca padrão do C++/Linux.

Começa-se por receber os inputs: o número de *workers*, a função e o número de outputs desta. Com estes valores é possível realizar o loop que cria os processos e respetivos *pipes*. Os *workers* criados são todos na mesma *layer* e uma *layer* abaixo do processo principal, com dois *pipes* para cada *worker*, um para passar bytes do processo principal para o *worker* e um para passar os resultados do *worker* para o processo principal.

Para guardar os *file descriptors* usou-se dois vetores, um com os pares de *file descriptors* do processo principal para os *workers* e o outro com o par dos *workers* para o processo principal. Nos vetores os *file descriptors* de um *pipe* são inseridos seguidos, isto é, os *file descriptors* do primeiro *pipe* criado serão os elementos nos índices 0 e 1, do segundo nos índices 2 e 3 e assim sucessivamente. Na linha 17 está ilustrado, após os *pipes* do novo processo serem criados, são inseridos nos vetores.

Após a criação dos *pipes* é realizado o `fork()`, se for o processo filho, linha 24, corre a função para os processos, que está explicada na secção 4.1.2, e se for o processo pai, linha 29 guarda o pid do processo e continua o ciclo para criar o resto dos *workers*.

```
DEFMETHOD_DLD(spawn_workers, interp, args, nargout, "Function test")
{
    num_processes = args(0).int_value();
    octave_value function = args(1);
    number_outputs = args(2).int_value();

    for(int i = 0; i<num_processes; i++){
        int to_child[2];
        int to_main[2];
        if(pipe(to_child)==-1 || pipe(to_main) == -1)
            cout <<"pipe creation failed";

        //Guarda-se o maior valor de file descriptor para o select
        update_max_fd(to_child, 2);
        update_max_fd(to_main, 2);

        fd_tochild[i*2 + READ] = to_child[0];
        fd_tochild[i*2 + WRITE] = to_child[1];
        fd_tomain[i*2 + READ] = to_main[0];
        fd_tomain[i*2 + WRITE] = to_main[1];

        pid_t pid = fork();

        if(pid == 0){
            close_unused_file_descriptors();
            process_work(ref(interp), ref(function),
            ↪  fd_tochild[i*2+READ], fd_tomain[i*2+WRITE], nargout);
            _exit(0);

        }else if(pid > 0){
            processes[i] = pid;
            close_unused_file_descriptors();

        }else if(pid < 0) {
            cout << "Creation of processes failed";
            return octave_value_list();
    }
    return octave_value_list();
}
```

Listing 6: Código C++ simplificado que cria a *pool*

### 4.1.2 Trabalho dos processos

Cada processo executa um ciclo infinito onde fica à espera que o *file descriptor* de leitura tenha dados para ler. Após obter um *job*, obtém os valores e desserializa-os para uma *octave_value_list*, usa a instância do interpretador para correr a função e recebe o resultado

também numa *octave_value_list*. Este resultado é serializado e escrito no *file descriptor* para o processo principal. Na listagem 7 está o código em C++ simplificado.

```
1  void process_work(octave::interpreter &interp, octave_value &function,
   ↪  int read_fd, int write_fd, int nargout) {
2      octave_value_list retval;
3      while(true){
4
5          obter valores de entrada;
6
7          desserializar valor para octave_value_list
8
9          correr função com interp
10         obter resultado
11
12         serializar resultado
13         escrever no pipe
14     }
15 }
```

Listing 7: Código C++ simplificado da função que cada processo executa

### 4.1.3 Enviar e receber dados

Para enviar e receber dados, feito através dos *pipes*, são enviados grupos de 3 valores individuais: o primeiro que é um Integer com o indíce que o *job* tomará na matriz de resultados, o segundo que é também um int com o número de valores *double* enviados.

A função recebe uma matriz onde cada linha é um vetor de entrada para a função paralelizada através dos *workers*, ou seja, cada linha da matriz é um *job*. A matriz é passada para um apontador de *doubles* com as linhas por ordem, e cada uma delas e escrita nos *pipes* livres. Para isso, usa-se o `select()`. Enquanto ouver linhas da matriz para enviar, o select devolve não só os *file descriptors* que já tenham acabado entreatanto, como também os *file descriptors* dos *pipes* livres para escrever os novos *jobs*, caso contrário, usa-se o select apenas para receber os *file descriptors* dos pipes com os resultados das funçoes, ilustrado na linha 17 do código.

Com as dimensões da matriz, consegue-se dividir o apontador em partes, usando o comprimento, pois diz quantos elementos enviar de cada vez e, com o número de linhas, sabe-se quando parar de procurar *pipes* para escrever e apenas esperar pelos resultados.

Como os *pipes* so funcionam com bytes, é necessário serializar as octave_value_lists e desserializar os apontadores, a API do *Octave* providência métodos para a classe das *Matrizes* bem como para a classe *Array* que devolvem apontadores de doubles. Para desserializar apenas se adiciona os doubles do apontador um a um para um *Array*, que são encapsulados num octave_value e este adicionado a uma *octave_value_list*.

```
DEFUN_DLD (send_data, args, ,
                    "Send the data to the workers")
{

    recebe matriz, guarda dimensoes e passa para apontador de doubles;

    int current_line = 0;
    criar sets de file descriptors para o select;

    int counter = 0;
    Matrix results_matrix(dim_vector(number_outputs, lines));

    while(counter < lines){
        fd_set current_read = backup_read_set;
        fd_set current_write = backup_write_set;

        if(current_line<lines){
            usar select para ver quais file descriptors estão prontos
            ↪ para ler e escrever dados

        }else{
            usar select para obter os file descriptors prontos para ler
            ↪ dados
        }

        for(int i = 0; i<num_processes; i++){
            int fd = fd_tomain[i*2 + READ];
            if(FD_ISSET(fd, &current_read)){
                obter valores e inserir na matriz de resultados;
                counter++;
            }
        }
        for(int i = 0;i<num_processes && current_line<lines;i++){
            int fd = fd_tochild[i*2+WRITE];
            if(FD_ISSET(fd ,&current_write)){
                escrever próximo job num pipe livre;
                current_line++;
            }
        }
    }
    octave_value temp(results_matrix);
    return octave_value_list(temp);
}
```

Listing 8: Código C++ simplificado que envia e recebe os dados da pool

### 4.1.4 Apagar pool

Terminar a *pool*, consiste em, em vez de enviar o grupo com os 3 valores, apenas se envia um valor negativo que o processo reconhece como sendo o código para parar e sai do ciclo infinito. Ao sair do ciclo, o processo filho continua a computação, passando para a linha 27 onde vê comando *_exit* que faz com que este processo seja terminado.

O processo principal, após enviar os sinais (valores negativos), faz um `waitpid()`, com os pids guardados na linha 30, até que todos os processos sejam terminados seguramente.

## 4.2 Funções compiladas (C/C++)

Se a função recebida for desenvolvida em C é bastante mais desafiante conseguir não só recebe-la através do octave como também ser capaz de a lançar com os threads.

Primeiro tive de conseguir obter e carregar a função a ser paralelilzada em runtime, para isso utilizei a library "dlfcn", capaz de carregar *dynamic libraries* [1] implementadas em C e obter as funções em runtime, contudo, esta biblioteca não existe para Windows o que pode causar problemas aos utilizadores da biblioteca.

```
typedef void (*fptr)(double *, int, double*, int*);
```

Listing 9: Assinatura do tipo da função a paralelizar

O "dlopen"cria um *handle* que contém o acesso à biblioteca dinâmica. Para obter uma função especifica desta utiliza-se a função *dlsym* que recebe um token, que é na verdade o nome da função dentro da biblioteca, e devolve um apontador para a função. Com o apontador da função, posso usa-la como usaria qualquer outra, porém, uma dificuldade, é o facto de precisar de saber à priori quais os tipos que a função recebe e devolve de modo a conseguir criar o apontador, por isso foi preciso restringir as funções que são possíveis ser paralelizadas a um molde, para o qual criei o *type fptr*, mencionado no capítulo 3 3. Assim é possível enviar este apontador para os *threads* e correr a função dentro do ciclo infinito destes.

O C++ tenta não deixar a gerência de memória somente para o programador, não com apontadores comuns com os *smart pointers*, nem com objetos. Os objetos em C++ quando fora do escopo em que foram criados são destruídos, ou seja a única maneira de conseguir criar um número maior que um de *threads* e conseguir mantê-los a correr e manter controlo sobre eles é a partir de um objeto que os cria, guarda, gere e pára quando não são necessários. Com isto quero dizer que criei uma classe/objeto que representará a *pool* de *threads*.

Esta classe toma partido da estrutura de dados do C++ *vector* para guardar os objetos *thread* e da *queue* que será a fila de *jobs* para os *workers*. Os *jobs* são na mesma uma linha da matriz como é para as funções interpretadas, mencionadas na secção 4.1, porém com umas pequenas diferenças.

Aqui as funções são em C por isso não podem receber objetos C++, ou seja, tenho de passar a linha da matriz para um apontador de doubles o que quer dizer que preciso de saber o seu tamanho. O `index` é o índice que o resultado deste *job* terá no Array dos resultados que é enviado para o *Octave*. Isto é feito de maneira semelhante ao das funções interpretadas onde a matriz é devolvida como um apontador usa-se aritmética de apontadores para empacotar cada linha num *struct* destes que são depois inseridos na fila.

A fim de lidar com problemas de concurrência dos *threads* como *data-races* utilizei variáveis de condição e *mutexes*. Tenho um *mutex* para a fila de *jobs*, que também trata da variável responsável de parar os *threads*, e um *mutex* para o *Array* de resultados também responsável pelo contador de resultados já recebidos. Em termos de variáveis de condição tenho também duas, distribuídas da mesma forma que os *mutexes*. De seguida vou explicar como e por que razão as uso.

### 4.2.1 Criar a pool

Para criar a *pool* recebe-se o número de *workers*, o nome da biblioteca e o número de dados de saída da função, com isso obtenho o apontador da função como explicado anteriormente e chamo o método da classe para criar a *pool*; cria-se os objetos *thread* com a função que todos os *threads* fazem, ilustrado na subsecção 4.2.2 e adiciona-se ao vetor. O apontador da função é guardado dentro da classe. A listagem 10 contém o código C++ simplificado da criação da *pool* de *threads* e também o respetivo método da classe.

### 4.2.2 Trabalho dos *threads*

Cada *thread* está num ciclo infinito onde acedem à fila de *jobs* para obter o próximo *job*, caso esta esteja vazia ficam à espera que esta deixe de estar vazia ou a variável de termino esteja a *true*. Após obter o *job*, extrai os valores, corre a função e insere-os na matriz após conseguir o *lock*. Por fim aumenta a variável usada para verificar se todos os *jobs* estão completos e notifica os restantes *threads* que o *lock* foi libertado.

### 4.2.3 Enviar e receber dados

Tal como é com a função compilada, a matriz recebida é passada para um apontador de doubles, no entanto, para este tipo de funções um *job* é encapsulado numa *struct* com o índice que o *job* vai ter na matriz de resultados, o número de valores de entrada e os valores em si. Este *job* é inserido na fila chamando o método da classe. Para o inserir na fila é primeiro preciso obter o *lock*, ficando à espera se não for possível. Depois de inserir o *job* na fila notifica todos os outros *threads* que podem tentar aceder ao *lock*.

Após todos os *jobs* serem colocados na fila, o *thread* principal fica à espera que todos os *jobs* tenham acabado para poder aceder à matriz de resultados e devolver, ilustrado na linha 19. Isto é alcançado através de uma variável da classe que cada *thread* aumenta

```
1  DEFUN_DLD(spawn_workers_c, args, , "parallelize a function written in C")
2  {
3      int nargin = args.length();
4
5      if(nargin < 3){
6          print_usage();
7      }
8
9      int num_threads = args(0).int_value();
10     charMatrix library = args(1).char_matrix_value();
11     number_outputs = args(2).int_value();
12
13
14     octave_value_list retval;
15     fptr function = load_function(library.row_as_string(0));
16
17     threads = make_unique<Threads>(function, num_threads,
       ↪  number_outputs);
18
19     threads->create_pool();
20     return retval;
21 }
22
23 void Threads::create_pool() {
24     threads.resize(num_threads);
25
26     for(int i = 0; i<num_threads;i++){
27         threads.at(i) = std::thread(&Threads::Thread_work, this);
28     }
29
30 }
```

Listing 10: Código C++ simplificado da classe e da função principal que cria a pool

quando acaba o seu *job*, linha 23 (protegida por um *lock*) e o *thread* principal fica à espera que esta variável seja igual ao número de linhas da matriz de entrada.

### 4.2.4 Apagar pool

A função principal que termina a *pool* apenas chama o método da classe. O *thread* principal tenta aceder a uma variável booleana através de um *lock* e mudá-la para true, o que faz com que todos os *threads*, que partilham esta variável saiam do ciclo infinito 11. No entanto, para garantir que os *threads* não são terminados a meio, é de seguida feito o `join()` de todos eles.

```
void Threads::Thread_work() {
    while(true){
       Job job;
       {
           unique_lock<mutex> lock(job_mutex);
           if(job_queue.empty()){
               job_var.wait(lock, [this] {
                   return !job_queue.empty() || terminate;
               });
           }
           if (terminate) {
               return;
           }
           job = job_queue.front();
           job_queue.pop();
       }

       obter valores do job e correr a função;
       {
           unique_lock<mutex> lock(results_mutex);
           adicionar resultado a matriz de resultados;
           }
           counter++;
       }
       return_cond.notify_one();

    }
}
```

Listing 11: Código C++ simplificado da função que cada *thread* executa

```
DEFUN_DLD(send_data_c, args, , "send matrix with each line being the
↪  input for each worker"){

    octave_value_list retval(1);
    obter dimensoes da matriz e passar para apontador de doubles;

    threads->resize_results(lines);

    for(int i = 0; i<lines; i++){
        Job j = {
                values + (i*columns),
                columns,
                i
        };
        threads->queue_job(j);

    }
    {
        unique_lock<mutex> lock(threads->get_lock());
        threads->get_cond().wait_for(lock, 180000ms, [&lines] {
            return threads->get_counter() == lines;
        });
    }
    retval(0) = threads->get_results();
    threads->reset_counter();
    return retval;
}
void Threads::queue_job(Job job) {
    {
        unique_lock<mutex> lock(job_mutex);
        job_queue.push(job);
    }
    job_var.notify_one();
}
```

Listing 12: Código C++ simplificado da classe e da função principal que envia e recebe dados da pool

# 5

# Algoritmo BoostDMS

## 5.1 BoostDMS

O primeiro teste da bilbioteca será com o algoritmo de otimização *BoostDMS* [4]. Modificou-se o algoritmo para utilizar a biblioteca para paralelizar alguns dos passos do algoritmo e vai-se comparar o seu desempenho paralelizado usando a biblioteca e correndo sequencialmente.

Primeiro vai ser explicado como funciona o algoritmo, depois quais as modificações feitas e por fim o resultados dos testes. Os testes foram realizado primeiramente numa máquina de um utilizador comum com poucos núcleos de computação e a seguir no cluster do departamento de informática, numa máquina com 12 *cores* / 24 *threads* núcleos de computação e numa com 16 *cores* / 32 *threads*.

### 5.1.1 Descrição

BoostDMS é um algoritmo de otimização de funções sem derivadas, ou seja tem como objetivo procurar no contradomínio da função objetivo os melhores valores que satisfazem uma série de requisitos. O método utilizado para o conseguir é composto por cinco passos principais:

1. **Inicialização**
2. **Seleção de um ponto de iteração**
3. **Passo de procura**
4. **Passo de eleição**
5. **Atualização do parâmetro de tamanho do passo**

O algoritmo centra-se em gerar vários conjuntos de pontos, avaliá-los com a função objetivo, gerar novos e assim convergir os resultados para o valor esperado. O tamanho dos conjuntos de pontos variam dependendo da função objetivo, e podem ter, por exemplo, 32 linhas com 9 números em cada linha ou até mais linhas ou números por linha, ou seja

é necessário correr a função objetivo várias vezes por iteração, consumindo muito tempo do algoritmo.

Por isso modificou-se o algoritmo *BoostDMS* nesta fase, que engloba o passo de procura, o passo de eleição e a atualização do parâmetro de tamanho do passo, que avalia os conjuntos com a função objetivo, para utilizar a minha biblioteca e executá-los em paralelo. Os *workers* são inicializados com a função objetivo e recebem o conjunto de pontos sob forma de matriz que é depois dividida igualmente pelos *workers*, pelo método *round-robin*, reduzindo o tempo gasto a avaliar os pontos.

Set parameters $0 < \gamma_{\text{dec}} < 1 \leq \gamma_{\text{inc}}$
Choose initial point $\boldsymbol{x}_0$ and step size $\alpha_0 > 0$
**for** $k = 0, 1, 2, \ldots$ **do**
    Choose and order a finite set $\boldsymbol{Y}_k \subset \mathbb{R}^n$ // (search step)
    $\boldsymbol{x}_k^+ \leftarrow$ `test_descent`$(f, \boldsymbol{x}_k, \boldsymbol{Y}_k)$
    **if** $\boldsymbol{x}_k^+ = \boldsymbol{x}_k$ **then**
        Choose and order poll directions $\boldsymbol{D}_k \subset \mathbb{R}^n$ // (poll step)
        $\boldsymbol{x}_k^+ \leftarrow$ `test_descent`$(f, \boldsymbol{x}_k, \{\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_i : \boldsymbol{d}_i \in \boldsymbol{D}_k\})$
    **if** $\boldsymbol{x}_k^+ = \boldsymbol{x}_k$ **then**
        $\alpha_{k+1} \leftarrow \gamma_{\text{inc}} \alpha_k$
    **else**
        $\alpha_{k+1} \leftarrow \gamma_{\text{dec}} \alpha_k$
    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k^+$

Figura 5.1: Algoritmo BoostDMS [11]

### 5.1.2 Metodologia e Testes

A função avaliada pelo algoritmo nos testes é uma pré-definida e simples que devolve dois valores. Sendo simples, a grandeza do tempo que leva a correr o algoritmo e demasiado pequena o que torna complicado verificar se a paralelização com a minha biblioteca está a dar resultado e se está a dar resultado eficientemente. De modo a ser possível observar o efeito da biblioteca, adicionei três *nested loops* com 10 vezes, que no fim somam as variáveis dos loops, de modo a adicionar um atraso à função, que consome ciclos do processador.

Nas avaliações começa-se por correr o algoritmo de forma sequencial e de seguida com a versão em paralelo aumentando o número de *workers*, testou-se com 2, 4, 8, 12, 16, 24 e 32 *workers*, as durações serão apontadas e apresentadas num gráfico de barras. As quantidades de números de *workers* que são utilizadas vai depender da capacidade do processador das máquinas utilizadas não utilizando mais *workers* que o número de *threads* do processador. Para os testes que serão discutidos aqui utilizaram-se máquinas do *cluster* do departamento de informática: a *squirtle* com 12 núcleos físicos e 24 *threads* e a *charmander* com 16 núcleos físicos e 32 *threads*.

No gráfico 5.2 estão os tempos de execução do algoritmo nas diferentes máquinas do *cluster*. Representado a azul está a *squirtle* e a *charmander* a laranja; no eixo dos $x$ são o número de *workers* e no eixo dos $y$ o tempo de execução em segundos. Quando comparando os tempos de execução entre as duas máquinas, vemos que os dos testes

realizados na máquina *charmander* são menores que os da máquina *squirtle*, isto deve-se ao facto do processador da primeira ser mais recente com mais ciclos por segundo fazendo com que o atraso da função seja diferente: 0.009 segundos para a *squirtle* e 0.007 segundos para a *charmander*. Além disso, vemos que os tempos têm tendência a diminuir com o aumento do número de *workers* em ambas as máquinas, como esperado, o que revela que a avaliação da função está de facto a ser realizada em paralelo e que a biblioteca está a funcionar.

No entanto, esta diminuição é menos aparente quando o número de *workers* é maior que o número de núcleos físicos do processador que pode ser devido ao facto do algoritmo ter uso intensivo do processador o que pode causar alguns dos núcleos lógicos a correrem de forma descontrolada, por exemplo pararem a meio da computação, mas, contrariamente ao esperado, na máquina *charmander* que tem só 16 núcleos físicos, o tempo de execução diminui consideravelmente quando são utilizados 32 *workers*, isto acontece porque no algoritmo são gerados 31 conjuntos para passar para a função, ou seja, cada *worker* só precisa de correr a função uma vez, enquanto que nas outras situações, pelo menos um *worker* corre a função duas ou mais vezes.

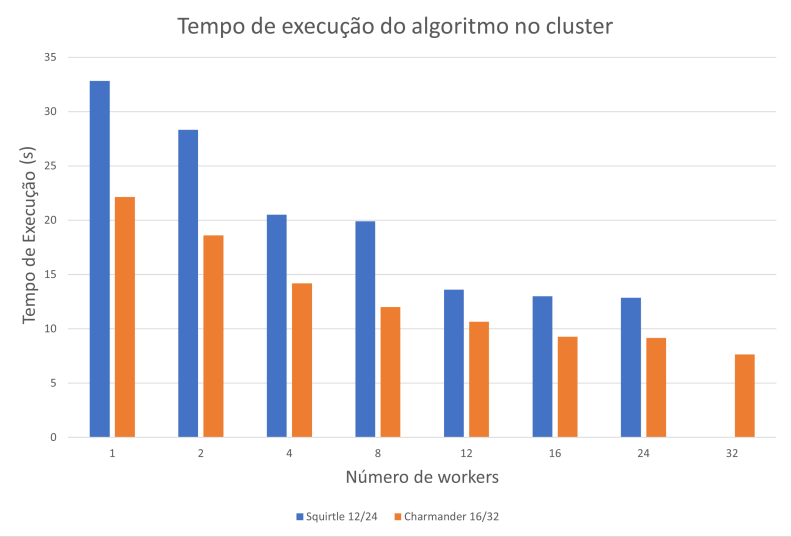



Figura 5.2: Gráfico dos tempos de execução do algoritmo no cluster

## 5.2 STYRENE

### 5.2.1 Descrição

O algoritmo *Styrene* [2] é uma função que recebe um vetor de oito valores entre zero e cem e calcula doze valores, desses doze valores, dependendo da versão do algoritmo utilizada, são devolvidos um subconjunto desses doze valores. Para este trabalho, para testar a biblioteca, este algoritmo é usado em conjunto com o algoritmo descrito anteriormente, *BoostDMS* 5. Para isso, é utilizada a versão multiobjetivo, ou seja, o *Styrene* devolve três

valores do conjunto de doze. Para ser possível usar esta função com a biblioteca e ao mesmo tempo suprimir a necessidade de criar, apagar e recriar ficheiros fez-se mudanças ao algoritmo.

Originalmente o algoritmo *STYRENE* está implementado em C++ e é chamado pelo *Octave* através de um script desenvolvido em *Octave*. Este script cria um ficheiro onde escreve os dados de entrada, que é depois passado como argumento do executável compilado com o código do algoritmo. O executável recebe o ficheiro com os dados de entrada e após calcular os resultados escreve-os no *stdout* que são por sua vez redirecionados para um segundo ficheiro. De modo a cortar o *overhead* do uso de ficheiros e para conseguir correr o algoritmo usando a biblioteca, este foi alterado de modo a usar a API do *Octave*, bastante semelhante ao que foi feito no resto deste projeto. O script *Octave* que chama o executável com os ficheiros deixa de ser necessário, e em alternativa a compilar o algoritmo num executável, compila-se num *octfile*. Assim a interface entre o *Octave* e o algoritmo *STYRENE* é mais simplificada onde os dados de entrada são passados como objetos reconhecidos pela API do *Octave* e por sua vez pelo compilador do C++ deixando de ser preciso usar ficheiros.

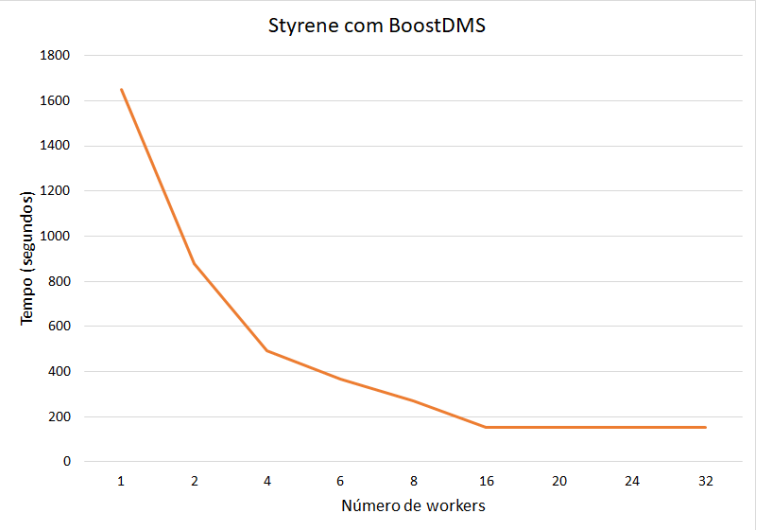
Deste modo o algoritmo é capaz de ser chamado pela biblioteca através da interface para funções interpretadas, podendo assim ser usado em conjunção com o algoritmo anterior: *BoostDMS* uma vez que o *STYRENE* funcionará como uma função *Octave* normal. No entanto, apesar de ser usado o interpretador para correr o algoritmo, este é corrido como se fosse compilado, ou seja, será mais eficiente em relação a custos de computação e tempo, devendo-se ao facto de *octfiles*, apesar de serem funções *Octave*, são no fundo código C++ que o interpretador do *Octave* é capaz de reconhecer e correr mesmo sendo código compilado. Isto é também um exemplo do uso da técnica mencionada na secção 3.1.3.

### 5.2.2 Metodologia e Testes

Funcionando como uma função *Octave* padrão, é possível obter a *function handle* e criar os *workers* como foi explicado na secção 3.1, ou seja, em vez do *BoostDMS* usar a função padrão, utilizará o *Styrene*. No entanto, visto que a função padrão e a função *Styrene* são fundamentalmente diferentes, uma vez que, a primeira não tem restrições de tamanho para o conjunto de entrada e a segunda só aceita conjuntos com oito valores, é preciso alterar os parâmetros do algoritmo *BoostDMS*, nomeadamente a lista na inicialização que terá a opção "0", isto é, a lista de iteração será inicializada com um ponto.

Os testes foram realizados de uma forma idêntica à dos testes com a função padrão. Foram realizados na máquina *charmander* com 16 cores e 32 *threads*, aumentando gradualmente o número de *workers*, entre um e trinta e dois e igualmente vemos uma redução significante do tempo de execução com o aumento de *workers* no gráfico 5.3 abaixo. Contudo, como os dados de entrada da função do *Styrene* são diferentes, a quantidade de conjuntos a serem avaliados numa iteração são também diferentes, ou seja, o número de *jobs* é diferente.

Para a função padrão, que recebia conjuntos de entrada com dezasseis números, eram gerados por iteração, trinta e um conjuntos de entrada. Para a função *Styrene* que recebe obrigatoriamente conjuntos com 8 valores, são criados por iteração 16 conjuntos de valores. Isto quer dizer que para este caso, o número ideal de *workers* são dezasseis, porque como só são criados dezasseis conjuntos, ou dezasseis *jobs*, assim cada *worker* recebe um *job* durante cada iteração do algoritmo. Isto significa que a melhoria do tempo de execução estagna nos dezasseis *workers* e um maior número de *workers* só criava mais custos, como ilustrado pelo gráfico.




(a) Gráfico com a duração do algoritmo em relação ao número de workers

| Styrene com BoostDMS | |
|---|---|
| Número de Workers | Tempo (segundos) |
| 1 | 1649 |
| 2 | 877 |
| 4 | 492 |
| 6 | 367 |
| 8 | 268 |
| 16 | 152 |
| 20 | 152 |
| 24 | 152 |
| 32 | 154 |

(b) Tabela com os resultados

Figura 5.3: Gráfico e tabela ilustrando os resultados da avaliação do algoritmo Styrene no algoritmo BoostDMS

# 6

# Conclusõess

Através do resultado da avaliação do algoritmo de otimização *BoostDMS* é posível visualizar que de facto a avaliação da função com os conjuntos de números gerados está a ser realizado em paralelo e corretamente, porque o tempo de execução diminui conforme o esperado, proporcional ao aumento do número de *workers* e o resultado de correr o algoritmo é igual em todas as iterações sem exceção.

Concluo assim que o objetivo principal deste projeto foi concretizado, uma vez que consegui realizar uma biblioteca para o *Octave* capaz de criar uma série de *workers* capazes de receber uma função, quer compilada em C/C++, quer interpretada, implementada em *Octave*, e ficarem infinitamente à espera de novos dados de entrada para correr a função. No entanto alguns objetivos não foram cumpridos como o facto de não ser possível parar os restantes *workers* quando já se recebeu um valor ótimo e o uso dos processadores gráficos para acelerar adicionalmente a otimização das funções.

## 6.1 Trabalhos futuros

Continuado o projeto seria interessante explorar a eficácia dos processadores gráficos para acelerar a computação e se relevante implementá-lo no projeto. Outras frentes interessantes de melhorar ou adicionar seriam por exemplo:

- adaptar o projeto a mais opções de uso, isto é, maneiras diferentes de passar dados para os *workers* ou ser capaz de enviar dados diferentes a diferentes grupos de *workers*, por exemplo.
- Mais flexíbilidade nos argumentos das funções para paralelizar
- Tornar o código mais robusto face a uso incorreto
- Adicionar à biblioteca suporte para sistemas distribuídos.

# Bibliografia


[1] R. Anthony. *Systems Programming Designing and Developing Distributed Applications*. Morgan-Kauffman, 2016, pp. 277–382 (ver p. 23).

[2] C. Audet, V. Béchard e S. Le Digabel. «Nonsmooth optimization through Mesh Adaptive Direct Search and Variable Neighborhood Search». Em: *Journal of Global Optimization* 41.2 (2008), pp. 299–318. DOI: 10.1007/s10898-007-9234-1. URL: http://dx.doi.org/doi:10.1007/s10898-007-9234-1 (ver p. 30).

[3] E. Bendersky. *Launching Linux threads and processes with clone*. 2018-08. URL: https://eli.thegreenplace.net/pages/about (ver p. 16).

[4] A. L. Custódio et al. «Direct multisearch for multiobjective optimization». Em: *SIAM Journal on Optimization* (2011) (ver p. 28).

[5] F. Denneman. *AMD Magny-Cours and ESX*. 2011-01. URL: https://frankdenneman.nl/2011/01/05/amd-magny-cours-and-esx/ (ver p. 6).

[6] F. Denneman. *NUMA Deep Dive Part 1: From UMA to NUMA*. 2016-07. URL: https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa/ (ver p. 5).

[7] T. W. Doeppner. *Operating Systems In Depth Design and Programming*. 2011, Cap. 3 (ver p. 13).

[8] J. W. Eaton et al. *{GNU Octave} version 7.1.0 manual: a high-level interactive language for numerical computations*. 2022. URL: https://www.gnu.org/software/octave/doc/v7.1.0/ (ver p. 4).

[9] H. Fujiwara, J. Hajek e O. Till. *Octave parallel package documentation*. URL: https://octave.sourceforge.io/parallel/package_doc/ (ver p. 8).

[10] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (ver p. ii).

[11] L. Monteiro. «PYTHON vs JULIA vs MATLAB PARA PROGRAMAÇÃO PARALELA». Universidade NOVA de Lisboa, 2021-11, pp. 29–30 (ver p. 29).

[12] P. Pacheco e M. Malensek. *An Introduction to Parallel Programming*. 2nd Edition. Morgan Kaufmann, 2020-03, pp. 0-496 (ver p. 7).

[13] R. Quinn. *Hands-On System Programming with C++ Build performant and concurrent Unix and Linux systems with C++17*. 2018 (ver p. 16).

[14] M. L. Scott. *Programming language pragmatics*. Morgan Kaufmann Pub, 2006, Cap. 15. ISBN: 9780124104099 (ver p. 11).

[15] G. M. Timothy, H. Yun e E. K. Alice. *The OpenMP Common Core*. 2019. ISBN: 9?802625?8862 (ver p. 7).