Luis Miguel Matos Pereira

Licenciado em Ciência e Engenharia Informática

Plataforma para Computações Paralelas Independentes para o MATLAB

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Pedro Abílio Duarte de Medeiros,
Professor Associado, Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2017

Plataforma para Computações Paralelas Independentes para o MATLAB

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

À família e amigos.

Agradecimentos

Em primeiro lugar gostaria de agradecer à minha família destacando os meus pais e irmão por todo o apoio que me deram ao longo de toda a vida, e o percurso académico não foi exceção. Um muito obrigado pelos sacrifícios e por sempre me proporcionarem todas a oportunidades possíveis, este não é o culminar do meu trabalho mas sim do nosso trabalho. Um paragrafo de agradecimento é insignificante comparado com tudo o que fizeram por mim até hoje. No que depender de mim também não hão de faltar oportunidades para vocês também, porque tudo o que é meu é vosso também.

Gostaria também de agradecer aos amigos que me acompanharam tanto no percurso académico como pessoal. Quer sejam amigos de longa data ou os que se foram juntando pelo caminho e que acabaram por ficar são todos importantes. Sabem que nem sempre foi fácil, mas que sempre me ajudaram a suportar o peso do trabalho e da responsabilidade com muita entreajuda e diversão, porque se tudo fosse fácil não tinha piada.

Por fim mas não menos importante gostaria de agradecer à Faculdade de Ciência e Tecnologias da Universidade Nova de Lisboa que é sem dúvida uma casa cheia de boas pessoas e profissionais, que para além do gosto pelo ensino se mostram sempre disponíveis para ficar mais uns minutos fora de horas para ajudar no que seja preciso, tanto a nível académico como a nível pessoal. Nestes aspetos gostaria de realçar os docentes dos departamentos de Informática e Matemática pois foram os departamentos com os quais tive mais interação ao longo destes últimos seis anos. Gostaria ainda de destacar o professor Pedro Medeiros pela oportunidade de realizar esta tese. Apesar de saber que me encontrava a trabalhar a tempo inteiro sempre mostrou uma enorme flexibilidade para me receber e orientar ao longo da realização deste trabalho mostrando-se sempre disponível. A toda a faculdade um muito obrigado, porque uma casa é tão boa quanto as pessoas que nela habitam.

Um agradecimento especial à Integrity destacando o Hugo Mestre e o Rui Shantilal por ao longo destes três anos me terem dado a flexibilidade de horários que me permitiu frequentar as aulas do mestrado e por fim realizar esta tese.

Resumo

Esta tese pretende desenvolver um sistema de apoio à execução de computações paralelas independentes utilizável a partir do Matlab. Em particular, pretende-se desenvolver de raiz um sistema que explore eficientemente a existência de múltiplos processadores em diferentes configurações *hardware*, com as seguintes características:

- Possibilidade de lançar múltiplas computações independentes em paralelo com diferentes parâmetros de entrada.
- Disponibilidade para interromper computações lançadas, sempre que se verificar que estas não produzem resultados úteis.

O desenvolvimento desta infraestrutura é motivada pela necessidade de reduzir os tempos de execução de problemas de otimização baseados em procura direta direcional. A estrutura algorítmica inerente à procura direta direcional e o caráter dispendioso associado à avaliação da função objetivo sugerem que reduções significativas do tempo de execução poderão ser conseguidas se se usarem múltiplos processadores.

Em particular, esta infraestrutura será usada para paralelizar o código SID-PSM, que é uma implementação sequencial de um algoritmo baseado em procura direta direcional.

Palavras-chave: Matlab, Octave, Otimização, Procura Direta Direcional, Computação Paralela, Algoritmo SID-PSM

Abstract

This thesis aims to develop a system to support the execution of independent parallel computations usable from Matlab. In particular, it is intended to develop from scratch a system that efficiently exploits the existence of multiple processors in different hardware configurations, with the following characteristics:

- Possibility of launching multiple independent computations in parallel with different input parameters.
- Being possible to stop launched computations, as long as it is found that they do not produce useful results.

The development of this infrastructure is motivated by the need to reduce the execution times of optimization problems based on directional direct search. The algorithmic structure inherent to directional direct search and the cost associated with objective function evaluation suggest that significant reductions in runtime can be achieved if multiple processors are used.

In particular, this infrastructure will be used to parallelize the SID-PSM code, which is a sequential implementation of an algorithm based on directional direct search.

Keywords: Matlab, Octave, Optimization, Directional Direct Search, Parallel Computing, SID-PSM Algorithm

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Capítulo 1

Introdução

Este primeiro capítulo introduz o contexto da tese. Fala sobre os problemas existentes e porque é que são relevantes, bem como a abordagem e proposta de resolução. Contém ainda o enquadramento dos objetivos da tese e a descrição da estrutura da mesma.

1.1 Motivação

Várias áreas da ciência como por exemplo astrofísica, medicina, economia e engenharia mecânica lidam bastante com problemas de otimização. Neste tipo de problemas, muitas vezes não é possível utilizar derivadas, sendo um dos motivos o elevado tempo de computação necessário para calcular a função objetivo.

Uma das formas de resolver este problema é utilizar um método de procura direcional. O algoritmo SID-PSM é baseado neste método, e foi desenvolvido pela Professora Ana Luísa Custódio do Departamento de Matemática da FCT NOVA [8], estando implementado em Matlab.

1.2 Problema/Relevância

Inicialmente foi sugerido que o tempo despendido no cálculo da função objetivo poderia ser reduzido significativamente se fossem utilizados múltiplos processadores. No âmbito da unidade curricular PIICEI, o estudante Gonçalo Sousa

Mendes realizou algumas experiências nesta direção. No entanto, a solução apresentada, embora completamente funcional e notoriamente mais rápida, apresentava algumas limitações [14].

A solução desenvolvida baseou-se numa arquitetura *master-worker*, em que os dois programas usam um interpretador de Matlab e seria testada em dois servidores do Departamento de Matemática da FCT NOVA (Markov – 10 processadores, Pitágoras – 8 processadores). Uma das limitações da solução previamente desenvolvida é a necessidade da interface gráfica para utilizar o PCT (*Parallel Computing Toolbox*), que é um pacote desenvolvido pela Mathworks que permite abordar problemas de paralelização com as funções fornecidas. Isto é problemático uma vez que os servidores do Departamento de Matemática não possuem interface gráfica, sendo a interação feita através de linha de comandos. Outra limitação da solução existente é o facto de não conseguir parar um *worker* que já está a executar um determinado cálculo, sendo assim necessário esperar que este acabe. Consomem-se, assim, recursos computacionais inutilmente.

É de realçar que o PCT é um *toolbox* pago, pelo que tem de se adquirir uma licença para a sua utilização. Seria vantajoso desenvolver uma solução de maneira a que não fosse necessária a utilização do PCT devido ao custo da sua licença.

Esta tese pretende então desenvolver um sistema que sirva de apoio à execução do algoritmo SID-PSM que tire proveito da existência dos vários processadores que um computador (ou vários computadores em rede) possa ter, sem recorrer à interface gráfica e com a capacidade de parar computações a meio da sua execução. Desta forma serão mitigadas as limitações da solução já desenvolvida.

1.3 Abordagem

Tendo em conta o trabalho de pesquisa já feito anteriormente pelo estudante Gonçalo Sousa Mendes, bem como as suas experiências realizadas, decidiu-se utilizar a interface para código externo fornecida pelo Matlab. Esta interface surge na forma de ficheiros *mex*. Estes ficheiros estão escritos em linguagem C/C++, e é possível a definição de funções que podem ser depois executadas pelo utilizador utilizando o Matlab.

Ter acesso a código externo ao Matlab vai permitir-nos tratar de todos os aspetos da paralelização do algoritmo explicitamente, ou seja, podemos escolher de que forma queremos fazer a paralelização, bem como a sua gestão ao longo do tempo. Isto implica um trabalho maior do que a solução já desenvolvida, mas desta forma podemos ter em conta fatores vantajosos em termos de melhorias de

tempos de execução, bem como realizar experiências com diferentes configurações, de forma a obter o melhor desempenho possível.

Por fim, utilizando o método proposto, é possível resolver as limitações existentes até agora. Ao sermos nós a definir todo o processo, bem como os seus componentes e a sua maneira de interagir, podemos desenvolver uma solução que não utilize a interface gráfica nem utilize o PCT, não sendo assim necessária a aquisição da sua licença. Desta forma podemos também fazer a gestão explícita dos vários processos nos diferentes processadores, o que permite parar um processo que esteja a decorrer, mesmo que este ainda não tenha terminado os seus cálculos, o que até agora não era possível.

Outra vantagem é que os *workers* podem ser interpretadores Octave (de acesso livre) e também ficheiros executáveis genéricos programados na linguagem que for mais adequada ao cálculo da função objetivo.

1.4 Solução proposta

Utilizando o acesso a código externo do Matlab, é proposto o desenvolvimento de um ficheiro *mex*, cujo propósito é unicamente fazer a ponte entre o Matlab e um programa "Servidor", que também seria desenvolvido em linguagem C/C++. Este servidor será um programa que aceita pedidos feitos pelo ficheiro *mex* (a pedido do utilizador via Matlab). O servidor será então responsável pela gestão de todos os processos, sejam eles iniciar novas execuções, terminar execuções que ainda não tenham terminado, e devolver resultados de execuções de volta ao utilizador.

Podemos pensar no ficheiro *mex* como um *proxy* entre o Matlab e o servidor. Quando é mandado iniciar uma execução, é devolvido ao utilizador um identificador único desta execução. O utilizador pode então utilizar este identificador para parar execuções em curso, bem como pedir os resultados das execuções. Este será o modelo de interação com o utilizador.

Completando a proposta de desenvolvimento do ficheiro *mex* e do servidor, irá haver um outro programa, que irá representar cada execução (cálculo da função objetivo). Este programa serve para o utilizador final implementar a sua função objetivo, e pode estar escrito em qualquer linguagem. Isto dá ao utilizador a vantagem de implementar o cálculo da função objetivo na linguagem que esteja mais familiarizado, ou que seja mais adequada ao problema que se está a tentar resolver. Este programa deve estar preparado para receber o numero da computação que representa, e os argumentos que irão ser utilizados no cálculo. No fim deve escrever o resultado da computação num ficheiro com o numero da computação

que lhe foi atribuída, de forma a ser lido pelo servidor.

Esta arquitetura faz com que a gestão seja transparente ao utilizador, desde a existência de um servidor, ao modo como o trabalho é gerido e distribuído, precisando apenas de invocar simples comandos no Matlab. Sendo uma arquitetura modular, em que cada componente tem a sua função específica, torna-se algo facilmente modificável, caso exista essa necessidade. É também possível o utilizador realizar outras ações no Matlab enquanto estão computações a decorrer (uma vez que estas executam em *background*), não havendo a necessidade do Matlab ficar bloqueado à espera de resultados.

1.5 Contribuição da tese

Após a análise do problemas e tendo em conta as suas características, no final deste trabalho pretende-se:

- Ter a definição de uma arquitetura de suporte à execução de múltiplas computações independentes, utilizável a partir do Matlab que possa ser executada sobre diferentes arquiteturas *hardware* com facilidades para abortar uma computação específica e sem utilizar a interface gráfica.
- Implementar e testar quanto à funcionalidade e desempenho um protótipo dessa arquitetura sobre um *cluster* de duas máquinas Linux.
- Usar o protótipo para paralelizar o algoritmo SID-PSM na otimização de funções objetivo em vários contextos e avaliar os ganhos conseguidos em termos da redução dos tempos de execução.

1.6 Estrutura do documento

Para além deste primeiro capítulo, que introduz o tema da tese, apresenta os problemas e a solução proposta, a tese contém ainda os seguintes capítulos:

- Capítulo 2 - Trabalho relacionado - Aborda possíveis soluções/tecnologias existentes no contexto do problema, vantagens e desvantagens de cada uma, e porque é que é ou não possível a sua utilização.
- Capítulo 3 - Conceção, desenvolvimento e implementação da plataforma - Contém todo o desenvolvimento e experiências realizadas ao longo do trabalho e exposição da plataforma para computações paralelas independentes para o Matlab, bem como exposição e análise de resultados.

- Capítulo 4 - Implementação SID-PSM - Descrição do algoritmo SID-PSM, de modo a compreender melhor de que forma as soluções apresentadas no Capítulo 2 se enquadraram na implementação da plataforma. É ainda utilizada a plataforma para paralelizar o algoritmo SID-PSM. Contém também resultados desta adaptação versus o algoritmo original, bem como a análise dos mesmos.
- Capítulo 5 - Conclusões - São discutidos os resultados obtidos e é feita uma apreciação crítica ao trabalho desenvolvido, bem como possível trabalho futuro.

Capítulo 2

Trabalho relacionado

Tanto o Matlab como o Octave são poderosas ferramentas de cálculo, cada uma com as suas vantagens e desvantagens. Este capítulo fala um pouco sobre estas duas ferramentas, formas de estender as suas funcionalidades e como é que se pode incorporar abordagens de programação paralela e distribuída para aumentar o desempenho do Matlab.

2.1 Execução de programas Matlab/Octave

O Matlab é um programa para realização de cálculo numérico. Na sua maior parte é utilizado para operações com matrizes, processamento de sinais e construção de gráficos, sendo portanto o seu nome uma abreviatura de *Matrix Laboratory* [26]. Tal como qualquer linguagem de programação o Matlab possui a sua própria sintaxe, e o código inserido pelo utilizador é compilado *just-in-time*. Noutro vetor temos o Octave que tem foco nos mesmos objetivos do Matlab, mas de uma forma mais ambiciosa. Para além de ser um *software* gratuito e *open source* (ao contrário do Matlab), pretende não só ser compatível com o Matlab como também estender funcionalidades que o Matlab não suporta (ou suporta de forma diferente) [17]. As diferenças entre o Matlab e o Octave notam-se principalmente na sintaxe (um suporta certas características que o outro não, e vice versa). A título de exemplo temos as strings onde o Matlab apenas suporta strings entre plicas (‘), enquanto o Octave tanto suporta entre plicas ou entre aspas (“). Desta forma há que ter este aspeto em conta caso seja desejável que código desenvolvido seja portável. Tanto um como o outro têm possibilidade de estender as suas funcionalidades. No caso

deste trabalho estamos interessados em extensões que permitam de alguma forma acelerar a execução de código, aproveitando assim potencialmente melhores condições de *hardware* existentes, e diminuindo o tempo que um utilizador tem de esperar até obter os resultados que pretende.

2.2 Extensões à funcionalidade

Para além da elevada panóplia de funcionalidades oferecidas tanto pelo Matlab como pelo Octave, ainda existem várias hipóteses de extensões que permitem ao utilizador obter um maior número de funções disponíveis, ou a flexibilidade de incorporar o Matlab ou o Octave com outro tipo de *software* externo. Estas extensões surgem na sua maioria na forma de pacotes ou em ficheiros *mex*.

2.2.1 Pacotes

Uma das formas de estender as funcionalidades existentes tanto no Matlab como no Octave é a utilização de pacotes de funcionalidades. No Matlab estes pacotes surgem na forma de *Toolboxes*. Uma *Toolbox* é um pacote que adiciona um conjunto de funções à linguagem [13], e podem ser obtidos de duas formas distintas: através da loja do próprio Matlab (algumas são gratuitas enquanto outras são pagas) ou através de outra fonte não oficial. Muitos dos pacotes que estão disponíveis são desenvolvidos e apoiados pela própria Mathworks e vêm recheadas de exemplos e respetiva documentação. É também possível criarmos o nosso próprio *Toolbox* [9]. Para isso basta escrever as funções que queremos num ou vários ficheiros .m e seguir o assistente que nos ajuda a configurar o nosso *Toolbox* e no final será gerado um ficheiro .mltbx que contêm todos os ficheiros e dependências necessárias para ser utilizado, bastando para isso apenas distribuir o ficheiro. No entanto, estes ficheiros têm algumas limitações, nomeadamente na presença de código executável, o que não é permitido.

No Octave o mesmo é possível mas de forma ligeiramente diferente. Estes pacotes surgem na forma de *Packages* e estão disponíveis alguns desenvolvidos por terceiros no site do Octave [20], não havendo propriamente uma loja, uma vez que estes pacotes são gratuitos. Apesar de haver em menor quantidade que no Matlab existe um grande leque de pacotes sobre áreas bastante variadas, o que ajuda ainda mais a tornar o Octave numa alternativa grátis viável em relação ao Matlab. Também aqui é possível criar e distribuir os nossos próprios pacotes [18]. Para tal, apenas temos de criar uma pasta com uma estrutura de pastas/ficheiros bem definida e no final comprimir a pasta raiz no formato .tar.gz. Depois podemos

distribuir este ficheiro como quisermos, bastando apenas importa-lo no Octave quando se quiser utilizar. Estes pacotes são mais permissivos que os do Matlab, uma vez que permitem a presença de código executável.

2.2.2 Ficheiros *mex*

Tanto o Matlab como o Octave oferecem uma interface para utilização de código externo, que surge na forma de ficheiros *mex* [10, 19]. O nome é uma abreviatura de *Matlab Executable*, e o código é em linguagem C/C++. Através destes ficheiros é possível receber parâmetros do Matlab/Octave e devolver resultados para variáveis dadas como *output*. Toda a computação fica ao critério do programador, e é completamente transparente ao utilizador, que executa estas funções como se de funções nativas de Matlab/Octave se tratassem.

Este aspeto permite ao programador uma enorme flexibilidade uma vez que, estando o código noutra linguagem, as possibilidades são praticamente ilimitadas. No caso concreto do nosso problema, a utilização de código externo vai permitir tirar partido a nível de liberdade de utilização de recursos, bem como aumentar a possibilidade de ter várias implementações diferentes, pois podemos utilizar bibliotecas externas ou executar outros programas, que por sua vez podem estar numa linguagem diferente. Podemos então tirar partido desta flexibilidade para acelerar a execução de código de maneiras diferentes, ou até mesmo uma combinação de diferentes métodos. Por exemplo, torna-se possível a execução de cálculos em paralelo, seja localmente, ou através da rede utilizando outros computadores. Existe ainda a hipótese de utilizar outros programas/linguagens mais adequadas à resolução de determinados problemas, como por exemplo CUDA [16].

2.3 Incorporar paralelismo e distribuição em programas Matlab

Existem disponíveis para o Matlab algumas ferramentas que permitem ao utilizador explorar várias formas de acelerar as suas computações, caso estas sejam propícias a tal. Estas ferramentas passam por tentar utilizar vários processadores disponíveis numa máquina ou em várias máquinas em *cluster*, bem como a possibilidade de utilizar um ou vários GPUs.

2.3.1 Multi-processamento com memória partilhada

Nos últimos anos têm sido vários os esforços para desenvolver ferramentas que permitissem tirar melhor proveito de condições de *hardware* no Matlab, com foco na computação paralela [1]. Algumas das mais importantes ferramentas desenvolvidas foram o MatlabMPI [5] e o pMATLAB [6] desenvolvidos pelo MIT e o *Parallel Computing Toolbox* (PCT) [12] desenvolvido pela própria Mathworks, uma coleção de funcionalidades que permitem tirar proveito de condições propícias ao paralelismo.

2.3.1.1 MatlabMPI

MPI (*Message Passing Interface*) [3, 28] é a definição de um protocolo de comunicação, normalmente utilizada em computação paralela onde são executadas várias tarefas que trocam mensagens entre si. Tem a vantagem de ser utilizado para executar várias tarefas numa única máquina ou num *cluster* de várias máquinas. A MPI consiste então na definição de um conjunto de regras, práticas e funcionalidades padrão que as suas implementações devem seguir. Algumas implementações conhecidas de MPI são o Open MPI [23] ou o MPICH [15].

O sucesso da MPI deve-se às características suportadas pelas suas várias implementações, muitas delas *open source* (como o caso do Open MPI e do MPICH), sendo as principais:

- Suporte para várias configurações de *hardware*.
- Suporte em vários sistemas operativos, sejam estes 32 ou 64 bits.
- Suporte em várias configurações de rede.
- Capacidade de portabilidade.

Embora a MPI não seja padrão IEEE ou ISO, pode na realidade ser considerada um padrão visto suportar praticamente todas as plataformas atuais, e tendo acabado por substituir todas as bibliotecas de comunicação por mensagens anteriores.

O MatlabMPI é então uma implementação da MPI para Matlab, disponibilizando ao utilizador algumas funções para acelerar o seu código Matlab de forma paralela. A implementação do MatlabMPI foi conseguida de forma a que seja portável para qualquer instância Matlab, e o desempenho com mensagens grandes (maiores que 3,5 MB) é semelhante ao desempenho do MPI em linguagem C [7] (Figura 2.1)

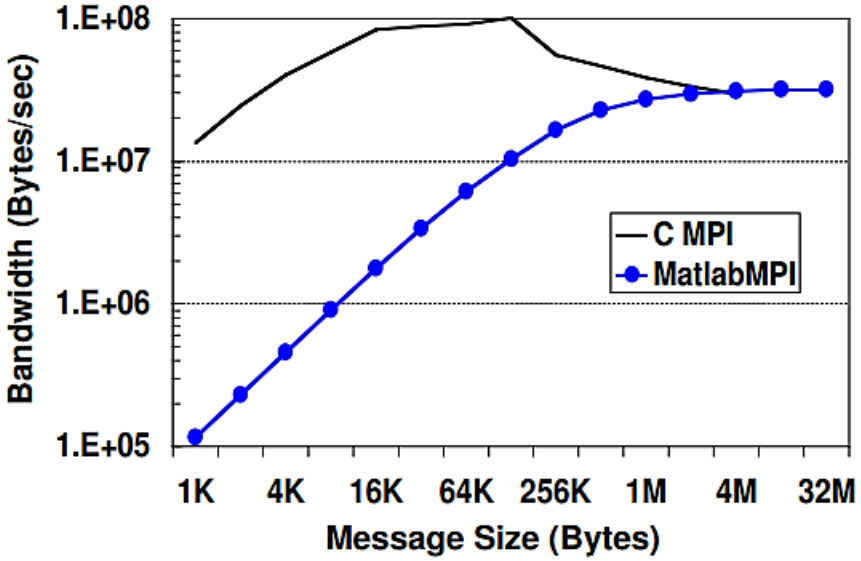

Figura 2.1: Desempenho do MatlabMPI comparativamente ao MPI em linguagem C (página 3 da referência [7])

Eventualmente o MatlabMPI acabou por ser descontinuado como versão única, passando a fazer parte do pMatlab, que utiliza o MatlabMPI para lançar programas e realizar comunicações entre processos. São necessárias tantas licenças Matlab quantas máquinas houver, caso se utilize um sistema distribuído.

2.3.1.2 pMatlab

O pMatlab [4] surgiu como objetivo de simplificar a paralelização utilizando o MatlabMPI. Isto quer dizer que, utilizando o MatlabMPI, o programador tinha que alterar a lógica inerente ao seu programa para acomodar a troca de mensagens fornecida pelo MPI, sendo quase necessário reescrever o programa. Desta forma, o pMatlab proporciona ao programador funções de nível mais alto que as do MatlabMPI, bem como tipos específicos de estruturas que permitem a manipulação de forma paralela, o que dá uma maior abstração dos processos que estão por trás ao programador, permitindo-lhe assim um foco maior no problema que está a tentar resolver visto que apenas necessita de alterar e adicionar algumas linhas ao programa existente.

Assim sendo, o pMatlab introduz um novo tipo de matriz, chamado *dmat*, abreviatura de *distributed matrix*. Por si só este tipo de matriz não é útil, pois temos primeiro que especificar onde e de que forma é que esta matriz se distribui pelos diversos processadores. Isto é feito associando um objeto *map* à matriz distribuída. Este objeto *map* tem quatro componentes que permitem especificar a distribuição dos dados pelos processadores disponíveis:

- *grid* (grelha) - permite especificar como é que cada dimensão da matriz *dmat* é dividida (Figura 2.2).
- *distribution* (distribuição) - especifica como é que cada dimensão da matriz *dmat* é distribuída pelos processadores, e suporta três modos de distribuição (Figura 2.3):
  - *block* (bloco) - cada processador contém um único bloco.
  - *cyclic* (cíclico) - cada bloco é intercalado entre os processadores disponíveis.
  - *block-cycle* (misto) - funciona como o cíclico, mas é possível especificar o tamanho do bloco.
- *processor list* (lista de processadores) - permite atribuir individualmente cada bloco da bloco da grelha a cada processador.
- *overlap* (sobreposição) - permite especificar a quantidade de dados sobrepostos entre os processadores (Figura 2.4).

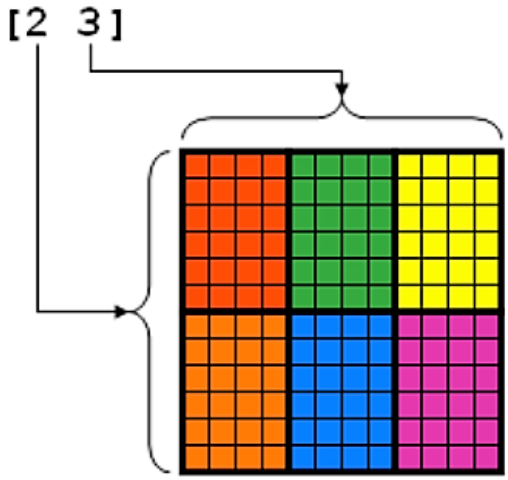

Figura 2.2: Divisão de uma matriz *dmat* em vários blocos (página 15 da referência [4])

Estando os dados das matrizes divididos em matrizes de dimensão mais pequena levou à criação de funções de acesso às matrizes em diferentes contextos, nomeadamente global e local. Contexto global consiste no acesso aos dados pelas coordenadas da matriz original, enquanto que contexto local o acesso aos dados é feito pelas coordenadas da matriz (bloco) que está em cada processador localmente.

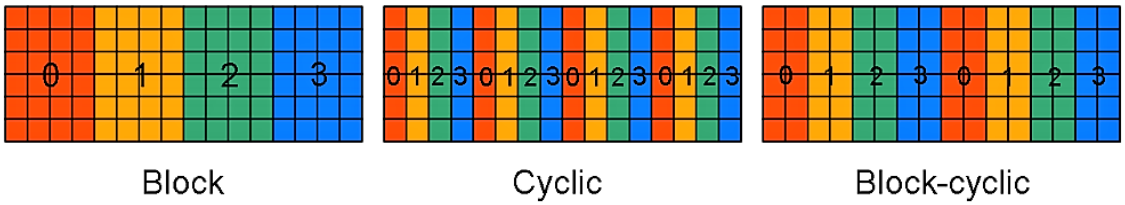

Figura 2.3: Modos de distribuição de blocos de uma matriz *dmat* por processadores (página 15 da referência [4])

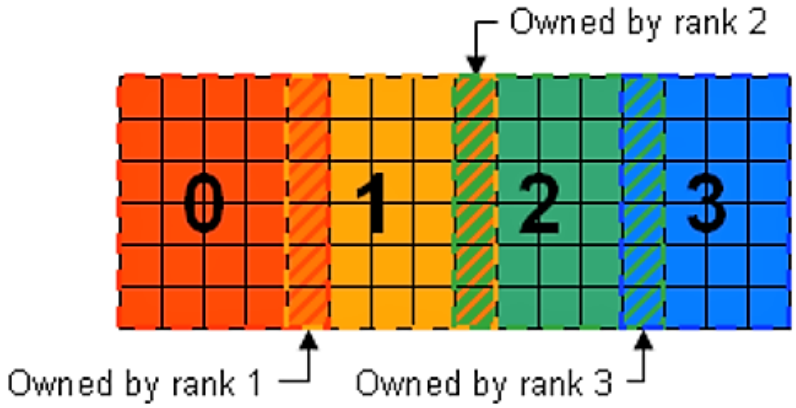

Figura 2.4: Sobreposição de blocos de uma matriz *dmat* por processadores (página 16 da referência [4])

Apesar de todas as funcionalidades que o pMatlab fornece, este não suporta a utilização interativa no Matlab. Isto quer dizer que só é possível utilizar o pMatlab por invocação de scripts, não sendo suportada a sua execução pela linha de comandos.

2.3.1.3 Parallel Computing Toolbox

O PCT fornece ao utilizador um conjunto de funções que permitem o controlo explícito de paralelismo em determinados aspetos. Um exemplo disto é o comando *parfor*, que permite que as iterações de um ciclo for sejam executadas em paralelo. Cada iteração do ciclo é executada por um *worker* de Matlab independente, o que envolve cuidados de sincronização entre os vários *workers*. No entanto, tudo isto é transparente para o utilizador, bastando especificar alguns parâmetros da função e tudo será executado e gerido de forma automática. É de notar que isto só pode ser aplicado caso as iterações sejam independentes uma das outras, caso contrário o resultado seria imprevisível uma vez que a ordem pela qual cada iteração é executada é arbitrária.

Os comandos do PCT, como por exemplo o *parfor* fazem uso da quantidade

de *workers* disponível, que pode ser configurada com o comando *parpool*, especificando o número de *workers* desejados.

Também disponível no PCT temos o *spmd* (*single program*, *multiple data*). Com este comando podemos executar código idêntico nos vários *workers*, mas cada *worker* tem dados diferentes. À primeira vista podemos pensar que a funcionalidade do comando *spmd* é idêntica à do comando *parfor*. Acontece que no *spmd* cada *worker* tem um identificador único, o que nos permite atribuir um conjunto específico de dados a um determinado *worker*, se assim o entendermos. Para além disto o Matlab possui ainda algumas funções que permitem controlar o fluxo das computações do *spmd* (Figura 2.5). Estas funções permitem por exemplo a comunicação entre os vários *workers*, enviando e recebendo mensagens entre si utilizando os comandos *labSend* e *labReceive* respectivamente.

Communication Function	Description
labBarrier	Block execution until all workers reach this call
labBroadcast	Send data to all workers or receive data sent to all workers
labProbe	Test to see if messages are ready to be received from other lab
labReceive	Receive data from another lab
labSend	Send data to another lab
labSendReceive	Simultaneously send data to and receive data from another lab
gcat	Global concatenation of an array across all workers. The result array is duplicated on all workers, unless *targetlab* is specified.
gop	Global reduction using binary associative operation across all workers. The result is duplicated on all workers (unless *targetlab* is specified). For example, *gop*(*@max*,*data*).
gplus	Global addition performed across all workers. The result is duplicated on all workers (unless *targetlab* is specified).

Figura 2.5: Comandos para controlar o fluxo da computação utilizando o *spmd* (página 294 da referência [1])

Semelhante ao comando *spmd* também existe o comando *pmode*, tendo este a possibilidade de interagir com os diferentes *workers* utilizando a interface gráfica. De modo a dar suporte à distribuição de diferentes dados para diferentes *workers*, o Matlab suporta alguns comandos que permitem criar matrizes em que cada *worker* tem uma área da matriz que lhe pertence. É possível a um *worker* aceder a valores fora da área da matriz delimitada para si, embora este processo demore mais tempo do que aceder aos valores que lhe competem.

Ao utilizar os comandos *parfor* e *spmd* a execução espera que os comandos acabem de executar. Outra limitação é que não é possível utilizar comandos que alterem o Matlab a nível de desenho de gráficos, pois cada *worker* executa numa instância de Matlab independente.

2.3.2 GPUs

O PCT também permite efetuar computações paralelas em GPUs quando estes suportam CUDA. O GPU é apropriado para computações paralelas quando o trabalho a ser realizado pode ser dividido em milhares de unidades independentes. Isto deve-se à maneira da arquitetura do *hardware* do GPU, uma vez que originalmente o seu propósito era acelerar cálculos de computações gráficas que consistem na sua maioria em aplicar a mesma operação em milhares de dados diferentes, ou seja, uma espécie de *spmd* em larga escala.

Apesar do PCT permitir programação específica em CUDA também tem disponíveis funções que permitem a um utilizador sem qualquer conhecimento de CUDA efetuar computações em paralelo no GPU, sem ter de alterar código, bastando para isso transformar os dados para o tipo *gpuArray*. Uma vez em *gpuArray*, fica disponível um leque de funções para enviar os dados para o GPU, interagir com eles e retornar os resultados. É claro que o *speedup* potencial não irá ser tão grande como poderia ser se fosse utilizada a linguagem específica CUDA, mas mostra claramente o potencial da utilização do GPU.

No caso da possibilidade de utilizar vários GPUs, o PCT também proporciona uma solução, que consiste em utilizar os comandos *spmd* ou *pmode* para dividir trabalho por diferentes CPUs. A partir daqui, cada processo poderá utilizar um GPU diferente. Desta forma temos a possibilidade de desenvolver uma solução extremamente eficiente, fazendo uso não só dos vários CPUs que um computador possa ter, bem como dos potenciais GPUs disponíveis.

Também é possível utilizar GPUs utilizando ficheiros *mex*. Esta abordagem já era utilizada antes ainda de serem suportadas funções no PCT que permitissem esta interação. Neste caso, é executado um ficheiro *mex* a partir do Matlab e que por sua vez chama o CUDA. Utilizar ficheiros *mex* para interagir com o GPU também tem as suas vantagens, sendo uma delas a possibilidade de utilizar o processamento do GPU sem ter o PCT instalado, ou em versões mais antigas do Matlab, visto que os ficheiros *mex* já eram suportados antes de existir o PCT. Outra vantagem é a utilização de vários GPUs em condições que o PCT não permite, pois o tipo *gpuArray* apenas suporta a utilização de um GPU. Além disto, permite ao utilizador mais flexibilidade, não só pela possibilidade de interagir com ficheiros CUDA nativos bem como a existência de várias bibliotecas dedicadas à utilização de GPUs em linguagem C++ por exemplo, que é totalmente suportada no formato *mex*.

2.3.3 Multi-processamento com memória distribuída

De forma a tirar partido da utilização de vários computadores em rede (*cluster*, *grid*, *cloud*) a Mathworks disponibiliza um pacote para o efeito, o *Matlab Distributed Computing Server* (MDCS) [11].

Desta forma torna-se possível realizar computações no Matlab de forma distribuída. De um modo geral, as computações podem ser configuradas num escalonador genérico (escalonador é um programa responsável pela gestão dos recursos disponíveis, distribuindo por estes os trabalhos que lhe vão sendo submetidos conforme os requisitos pedidos e os recursos disponíveis), embora seja oferecido suporte para outros escalonadores. Ao utilizar o MDCS temos acesso a licenças para todos os *toolboxes* que possuímos, o que dispensa licenças adicionais quando são utilizados os vários computadores de um *cluster*. O MDCS tem ainda a vantagem de, para além da computação ser feita de forma distribuída, é ainda suportada computação paralela em cada um dos nós disponíveis, tanto a nível de CPUs como a nível de GPUs (utilizando por exemplo o PCT). Obviamente que o tempo de comunicação dos *workers* em computadores remotos vai ser maior do que se estes apenas estivessem disponíveis localmente. A interface do escalonador proporciona ao utilizador um nível de abstração que lhe permite submeter trabalhos a serem realizados sem se preocupar com a distribuição pelos vários *workers*, bem como as diferenças dos sistemas operativos entre eles. O escalonador disponível por omissão no MDCS é o *Matlab Job Scheduler* (MJS), havendo também suporte a outros tipos de escalonador no *Common Job Scheduler* (CJS).

Ao utilizar o MDCS é possível definir várias configurações, desde o número de *workers* disponíveis até ao local onde são armazenados os dados, entre outros. Os *jobs* submetidos ao MDCS podem ser de dois tipos:

- Independentes, tal como o nome indica não precisam de comunicar entre si, o que significa que não precisam de ser executados ao mesmo tempo. Estes não precisam de atrasar o seu início, sendo as tarefas distribuídas à medida que os *workers* ficam disponíveis, podendo até várias tarefas do mesmo *job* serem executadas pelo mesmo *worker*.

- Comunicativos, que precisam de comunicar uns com os outros durante a execução das suas tarefas. Tendo isto em conta, *jobs* que comuniquem entre si só podem começar a ser executados quando existem *workers* suficientes para executar as tarefas ao mesmo tempo.

Os *jobs* passam por várias fases desde a sua criação até ao fim da sua execução

(Figura 2.6), sendo elas *pending*, *queued*, *running*, *finished/failed*. No final da execução de um *job*, os dados são guardados no local configurado, onde podem ser posteriormente consultados. É ainda possível submeter um *job* a um *cluster* para executar em *background*, seja porque não queremos que o *input* fique bloqueado, ou porque queremos fechar o Matlab e voltar depois para ver os resultados. Isto é possível utilizando o comando *batch*.

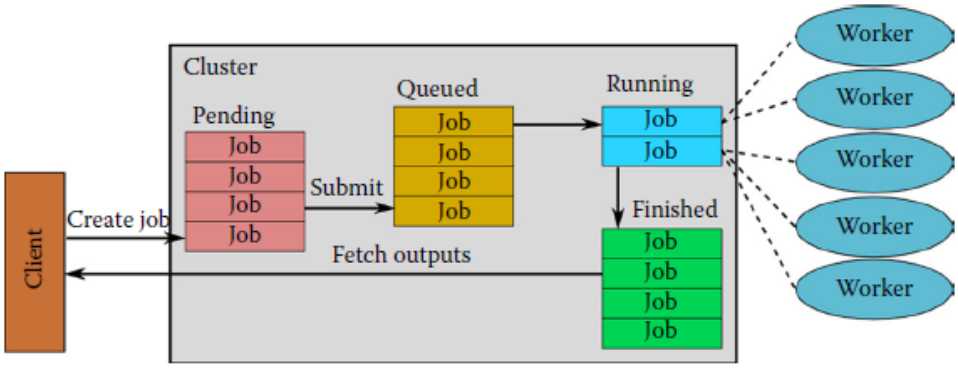

Figura 2.6: Estados da execução de *jobs* no *cluster* (página 332 da referência [1])

2.4 Futuros e promessas

Muitas vezes na perspetiva de programação concorrente é habitual a utilização de futuros e promessas [22]. Futuro é um valor, que pode ainda não estar atribuído, uma vez que a sua computação pode ainda não ter terminado. O valor do futuro vai ser atribuído por uma promessa, quando esta terminar a sua execução. A conclusão de uma promessa e a consequente atribuição do valor ao futuro chama-se resolução.

A vantagem de utilizar este método é a possibilidade separar a definição de um valor da forma como este é calculado, o que permite mais flexibilidade em programação paralela.

A utilização de promessas pode ser implícita ou explícita:

- Utilização implícita significa que o valor do futuro é automaticamente atribuído quando a promessa termina.

- Utilização explícita significa que o utilizador tem que chamar uma função para atribuir o valor obtido pela promessa ao futuro.

A utilização de futuros pode ser bloqueante ou não-bloqueante. No caso de ser bloqueante, quando tentamos utilizar um futuro que ainda não foi resolvido, a execução bloqueia à espera da resolução. Por outro lado, caso o futuro não tenha sido resolvido, mas é possível executar outras ações enquanto se espera pela sua resolução, é uma utilização não bloqueante.

As promessas podem ser executadas logo no momento da criação dos futuros (avaliação *eager*) ou apenas quando se deseja utilizar o valor do futuro (avaliação *lazy*). Cabe ao programador decidir o método mais vantajoso tendo em conta o problema que se está a tentar resolver.

No caso do Matlab não é suportada a utilização de futuros. No entanto utilizando a interface de acesso a código externo pode-se utilizar uma linguagem que o suporte.

2.5 Desenvolvimento de uma solução de raiz

Apesar da qualidade das soluções descritas na secção anterior, elas não são suficientemente flexíveis para as necessidades de paralelização do algoritmo SID-PSM, nomeadamente na parte que tem a ver com a possibilidade de descartar computações cujos resultados já não são relevantes. Há ainda a assinalar que os produtos acima referidos (nomeadamente os *toolboxes* PCT e MDCS) não são gratuitos, e seria preferível uma solução que utilizasse componentes gratuitos. Outra dificuldade nas soluções é a necessidade de utilização da interface gráfica, que não é suportada pelo *cluster* de computadores do Departamento de Matemática. Existem ainda dificuldades com o controlo de licenças quando são utilizados vários nós para realizar as computações.

Por estes motivos, nesta tese vai-se desenvolver, de forma incremental, um sistema cujo comportamento é completamente controlado por nós e que suporta a execução de computações especificadas em várias linguagens. No caso de serem especificadas funções objetivo em linguagem Matlab, estas serão executadas utilizando o Octave.

O sistema irá sendo estendido de forma a incorporar vários tipos de hardware:

- Numa primeira fase, serão utilizados vários processadores num único computador
- Posteriormente serão utilizados vários processadores distribuídos por diferentes computadores.

No desenvolvimento da solução final serão utilizadas algumas abordagens deste capítulo, nomeadamente a utilização da interface de acesso código externo do Matlab através de ficheiros *mex*, algumas mecânicas dos futuros e promessas adaptadas ao problema em questão e MPI.

Capítulo 3

Conceção, desenvolvimento e implementação da plataforma

Durante a elaboração da tese foram realizados testes e desenvolvidas soluções que culminaram numa primeira prova de conceito que mostrou que era efetivamente possível contornar os problemas apresentados no primeiro e segundo capítulos. Estes testes decorreram tanto no Matlab como no Octave, e foram realizados em ambientes de um único computador ou de um *cluster* de quatro computadores em rede.

Após aferir a viabilidade do primeiro protótipo, foi feita uma iteração sobre o mesmo de forma a tornar o protótipo numa plataforma de utilização genérica, ao mesmo tempo que acomodava as necessidades do algoritmo SID-PSM de forma a poder realizar testes e aferir resultados.

3.1 Primeiras abordagens

Numa primeira fase do projeto foi desenvolvido um protótipo que tinha como intuito tentar cumprir o máximo dos seguintes objetivos:

- Lançar várias computações independentes a partir do Matlab.
- Desenvolver a solução de forma a utilizar apenas a licença base do Matlab.
- Possibilidade de abortar computações antes de estas terem terminado.
- Não ter necessidade de utilização da interface gráfica.

3.1.1 Pacotes Octave

Numa primeira fase começou-se por tentar desenvolver um pacote para o Octave. A escolha do Octave para começar a desenvolver um conjunto de experiências deveu-se ao facto de ser um *software* gratuito, ao contrário do Matlab, mantendo de um modo geral as mesmas funcionalidades. O objetivo do pacote seria a fácil instalação por parte de um futuro utilizador.

Um pacote para o Octave é relativamente simples de desenvolver, pois apenas consiste numa pasta comprimida no formato .tar.gz com uma estrutura específica de pastas/ficheiros. De todas as opções possíveis de ficheiros, são apenas obrigatórios dois: um que contém informações sobre o pacote (como por exemplo o nome do programador, versão, etc.) e outro com uma listagem de todos os ficheiros contidos no pacote e respetivas licenças. As funções que queremos definir e que ficarão acessíveis no Octave após instalar o pacote ficam alojadas em ficheiros .m numa pasta de nome *inst*. Esta é a composição básica de um pacote de Octave, e foi implementada com sucesso.

3.1.2 Ficheiros mex e oct

Numa fase seguinte, explorou-se o acesso a código externo ao Octave. Existem duas maneiras de o conseguir: utilizando ficheiros *mex* ou ficheiros *oct*. Ficheiros *mex* são ficheiros escritos em linguagem C/C++, que são posteriormente compilados para ficarem no formato *mex*, que depois irá ser executado pelo Octave. Por outro lado, ficheiros *oct* são ficheiros escritos em linguagem C++ e que, à semelhança dos ficheiros *mex*, são compilados no formato *oct* para serem executados pelo Octave. Esperava-se os ficheiros *oct* fossem mais rápidos no Octave que os ficheiros *mex*, pois seria um ficheiro com um formato específico para ser lido pelo Octave, que está também implementado em C++. Já os ficheiros *mex* poderiam ser lidos tanto pelo Octave como pelo Matlab, o que significava que desenvolvimentos que fossem feitos desta forma no Octave iriam de certeza funcionar também no Matlab.

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(m-1,1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m-1,A(m,n-1)) & \text{se } m > 0 \text{ e } n > 0. \end{cases}$$

Figura 3.1: Função de Ackermann

Continuaram os testes utilizando tanto ficheiros *oct* como ficheiros *mex*, a fim de no final obter um comparativo de desempenhos entre os dois. As experiências

consistiram em vários detalhes, desde utilizar funções externas à principal, testar paralelização utilizando *pthreads* e receber vários *inputs* do Octave, bem como devolver vários *outputs*. Por fim, foi comparado o tempo de execução entre um ficheiro *oct* e um ficheiro *mex* utilizando a função de Ackermann [25] (Figura 3.1), que dependendo dos parâmetros fornecidos pode tornar-se numa função com elevado custo computacional (os valores utilizados foram m a variar entre 0 e 3 e n a variar entre 0 e 17). Após vários testes verificou-se que os tempos de execução eram iguais, e como tal, abandonou-se o desenvolvimento dos ficheiros *oct*, uma vez que, para além de não conferirem qualquer vantagem adicional, não iriam funcionar no Matlab.

3.1.3 Paralelização

Neste momento começaram os testes no Matlab, pedindo uma licença *trial* para o efeito. Verificou-se que os ficheiros *mex* tinham o mesmo comportamento no Matlab que no Octave. Fez-se nesta altura um teste que consistia em utilizar funções do Matlab a partir de código externo. Tudo funcionou corretamente, exceto no caso em que eram feitas de forma concorrente pelas *pthreads*. Quando isto acontecia, o Matlab fechava inesperadamente. No entanto, no Octave funcionou tudo como era esperado, o que nos levou a especular que seria algum problema com o licenciamento do Matlab. Testou-se ainda uma forma de contornar este problema, que consiste em executar essas funções pelo Octave em vez do Matlab, e obteve-se sucesso. Esta é a solução utilizada quando o código da função objetivo está implementado em Matlab.

3.1.4 Primeiro protótipo

A partir daqui começou a moldar-se a arquitetura que iria compor a primeira versão de um protótipo. Começou-se por desenvolver um ficheiro *mex* que mandava executar um programa *master* que tratava dos cálculos em background, permitindo assim ao utilizador continuar a utilizar o Matlab sem ter de ficar à espera que a computação terminasse. O resultado era guardado num ficheiro, que tinha que de consultado pelo utilizador. Numa iteração seguinte desenvolveu-se uma interface de comandos simples que permitiam ao utilizador interagir com o programa, nomeadamente através dos comandos *start*, *stop* e *result*. Quando o utilizador invoca o comando *start* entra em execução o tal programa *master* que executa os cálculos (no caso dos testes estes cálculos consistem na execução da função de Ackermann, por simplicidade) e é devolvido ao utilizador um identificador

da computação. Este identificador pode então posteriormente ser utilizado com os comandos *stop* e *result*, que se traduz respetivamente em parar a execução correspondente ao identificador e obter os resultados da computação correspondente ao identificador. Nesta arquitetura o programa *master* (que executa os cálculos em *background*) escreve os resultados num ficheiro correspondente ao identificador que é devolvido ao utilizador, que é posteriormente lido quando é invocado o comando *results*, seguido do identificador. Desta forma, toda a interação é feita pelo Matlab, não tendo o utilizador de consultar nem gerir nada externo.

Numa versão final deste primeiro protótipo, a maneira de interação do utilizador permaneceu inalterada, mudando apenas aspetos na arquitetura interna. Já nesta fase foi possível comprovar que a arquitetura modular permite atualizar o protótipo sem que tal seja percetível ao utilizador, o que é uma grande vantagem. Os aspetos alterados consistiram na criação de um programa "Servidor", que iria receber os comandos do utilizador através do ficheiro *mex*, e fazer a gestão do início de execuções, parar execuções em curso e devolver resultados. Desta forma todos os componentes do protótipo se tornam mais simples. O ficheiro *mex* serve apenas para comunicar comandos do utilizador ao servidor e mostrar o resultado dos mesmos. O servidor fica encarregue de receber comandos vindos do ficheiro *mex* e agir de acordo com eles, nomeadamente começar/parar execuções, mostrar resultados de execuções terminadas e fazer a gestão das execuções, sejam as que estão a decorrer ou as que já terminaram. O programa *master* representa então uma execução, e caso esteja implementado da forma paralela (no caso dos testes utilizando *pthreads*) faz a gestão de vários programas *worker* de modo a aproveitar vários processadores para obter os resultados pretendidos. Pode também estar implementado sem recorrer a paralelização, sendo os *workers* descartados. A comunicação entre o ficheiro *mex* e o programa servidor é efetuada utilizando *named pipes*, uma vez que o programa servidor está sempre em execução e o ficheiro *mex* começa e termina cada vez que o utilizador invoca um comando a partir do Matlab.

Na versão final do primeiro protótipo (Figura 3.2) o programa *master* encontra-se desenvolvido de três formas distintas. Numa delas, a versão mais simples, executa a função de Ackermann uma vez. Uma segunda forma é a execução da função de Ackermann duas vezes de maneira sequencial, e a terceira é a execução da função de Ackermann duas vezes, mas de forma concorrente utilizando *pthreads*. Desta forma foi possível aferir alguns resultados comparativos entre os três, nomeadamente o tempo de execução (Tabela 3.1). Mais uma vez, os parâmetros utilizados para a função de Ackermann foram m a variar entre 0 e 3 e n a variar entre 0 e 17. A versão simples que executa o ciclo uma vez demorou

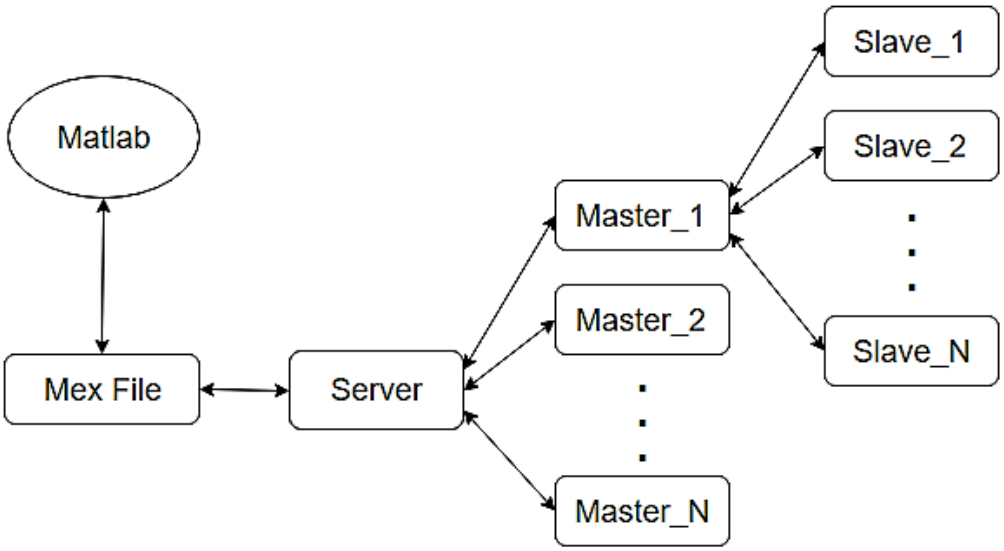

Figura 3.2: Arquitetura do protótipo final

um total de 125 segundos, sendo este o resultado base. Na segunda forma o ciclo é executado sequencialmente duas vezes e o resultado obtido foi o dobro (249 segundos), que era o resultado esperado. Já a versão em paralelo demorou apenas mais 20 segundos do que a versão simples, o que mostra uma vantagem de 104 segundos. É de realçar que estes resultados foram obtidos fazendo a média de 5 rondas de testes. Os testes foram feitos em ambiente Linux (Ubuntu 16.04) e o computador utilizado tem dois processadores e 6GB de memória RAM. A versão Matlab utilizada foi a R2016b.

Programa	Sequencial	Sequencial x 2	Paralelo x 2
Tempo (segundos)	125	249	145

Tabela 3.1: Resultados de diferentes implementações do *master*

Concluindo a fase de testes, bem como esta versão inicial do protótipo e analisando os resultados, mostrou-se a viabilidade do projeto, bem como da arquitetura da solução desenvolvida até agora uma vez que cumpria os objetivos inicialmente propostos.

3.2 Plataforma para computações paralelas independentes para o MATLAB

Para utilizar o protótipo desenvolvido como uma plataforma genérica é necessário que este seja adaptável a diferentes tipos de problemas. Isto significa que tem de

ser oferecida ao utilizador uma interface de utilização flexível e cujo funcionamento interno da paralelização seja transparente, abstraindo-o assim de detalhes de implementação e permitindo-lhe apenas focar-se na sua própria solução.

Assim sendo, de modo a completar o protótipo para desenvolver as características necessárias é preciso:

- Suportar a utilização de processadores distribuídos por diferentes computadores, nomeadamente para utilização em *clusters*.
- Definir uma forma do utilizador poder especificar o número de processadores que vão ser utilizados.
- Permitir a definição da função a paralelizar.
- Permitir atribuir um número arbitrário de argumentos à função que executa em paralelo, suportando assim qualquer função definida.

No final será possível utilizar a plataforma para adaptar qualquer problema para tirar partido da paralelização.

3.2.1 Plataforma de utilização genérica

O primeiro passo tomado foi solidificar a comunicação entre os diferentes componentes da plataforma. Em concreto, foi necessário encontrar uma maneira de suportar a utilização de diferentes computadores em rede. Para tal, foi utilizada a MPI (*Message Passing Interface*) no servidor, ficando assim com as seguintes características:

- Quando inicializado, o primeiro processo fica com o papel de servidor principal. Este servidor fica sempre na máquina em que é executado o Matlab, e é responsável pela comunicação entre este e os restantes servidores distribuídos pelas restantes máquinas disponíveis.
- Os servidores secundários apenas comunicam com o servidor principal, e são responsáveis apenas pelas ações na sua máquina respetiva.

Com esta alteração (Figura 3.3) o comportamento do ficheiro *mex* mantém-se igual, ou seja, apenas serve para comunicar as instruções do utilizador dadas através do Matlab ao servidor principal utilizando os *named pipes*. Esta é a razão pela qual o servidor principal fica sempre na máquina onde é executado o Matlab: a cada instrução dada ao ficheiro *mex* equivale a uma execução do mesmo. É

necessário manter um canal de comunicação permanente para que as execuções seguintes consigam comunicar com o servidor principal.

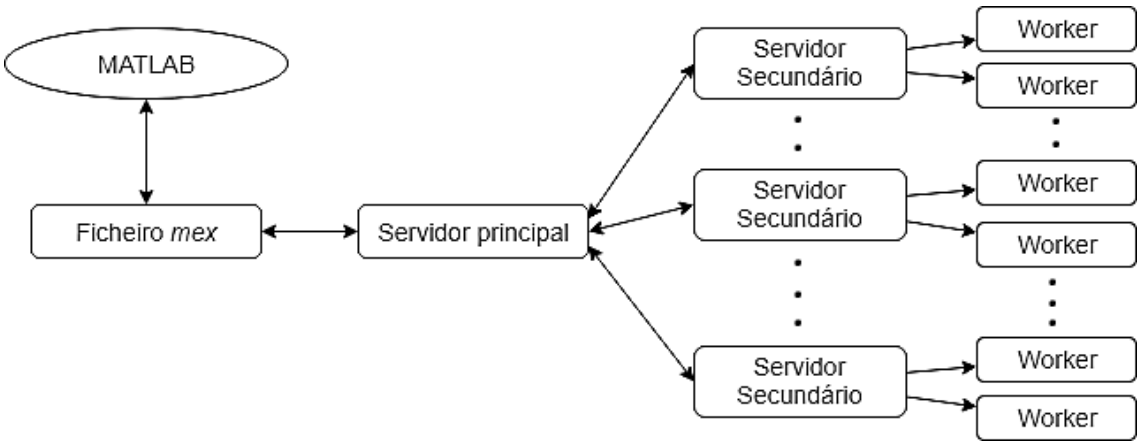

Figura 3.3: Arquitetura final da plataforma

O servidor principal tem como responsabilidades receber as instruções do utilizador recebidas pelos *named pipes*. A atribuição de computações aos servidores secundários é feita utilizando um escalonamento *round-robin* [27]. Isto significa que as computações vão sendo atribuídas aos servidores secundários de forma cíclica. É necessário guardar a que servidor secundário correspondem os números das computações, para posteriormente saber a que servidor pedir os resultados de uma determinada computação ou abortar uma computação. Na perspetiva do servidor secundário é necessário receber instruções do servidor principal (começar computação, abortar ou obter resultados), bastando para isso guardar apenas os números de computações que lhe competem. Os ficheiros com os resultados são guardados localmente em cada servidor secundário.

A execução do servidor utilizando a MPI faz uso de um ficheiro chamado *mpihostfile*, que vai ser consultado sempre que for inicializado o servidor. Este ficheiro está disponível para alteração por parte do utilizador, permitindo-lhe assim controlar quantos nós em cada máquina estão disponíveis.

Implementar o servidor com MPI trouxe a necessidade de iniciar e terminar explicitamente o processo de paralelização. Foram então criados os comandos *init* e *finish*:

- O comando *init* faz com que o ficheiro *mex* inicialize os servidores (principal e secundários) e crie os *named pipes* que serão utilizados nas execuções *mex* seguintes. O comando *init* inclui ainda um argumento que corresponde ao número de processadores que vão ser utilizados, sendo essa informação utilizada aquando da inicialização dos servidores.

- O comando *finish* serve para terminar todos os processos, informando o servidor principal. Isto faz com que seja enviada uma mensagem a todos os servidores secundários para abortarem qualquer computação que ainda esteja a decorrer, limpar os ficheiros de resultados guardados localmente e terminar. Assim que todos os servidores secundários terminam, o servidor principal também termina, sendo posteriormente removidos os *named pipes*.

Ao utilizar a MPI como solução para implementar o servidor significa que sempre que a plataforma for utilizada têm de existir no mínimo dois nós, um para o servidor principal e outro para o servidor secundário.

De seguida foram adaptados os componentes da plataforma para suportar a possibilidade do utilizador inserir um número arbitrário de argumentos em cada computação. Estes argumentos são transmitidos desde o ficheiro *mex* até ao *master* onde serão utilizados para executar as operações definidas pelo utilizador.

A função a executar em paralelo fica ao critério do utilizador. O ficheiro *master* está implementado em linguagem C e está preparado para receber o identificador da computação a realizar e um número arbitrário de argumentos. A partir daqui o utilizador fica livre para implementar a sua função. A função pode ser diretamente implementada no *master* ou ser outro programa diferente escrito noutra linguagem. As únicas condições a ter em conta para manter o funcionamento correto da plataforma são:

- O programa definido pelo utilizador tem de ser executado a partir do *master*. Isto deve-se ao facto da plataforma estar programada para executar o *master* sempre que existe uma nova computação, fornecendo-lhe o identificador e os argumentos.

- O resultado da computação tem que ser escrito num ficheiro com o formato *output_X.txt* onde X corresponde ao identificador da computação. A plataforma está programada para ler os resultados em ficheiros com este formato no nome. O ficheiro apenas deve conter uma linha com o resultado obtido.

Esta característica permite ao utilizador da plataforma utilizar a linguagem que lhe é mais familiar ou mais adequada à resolução do problema em questão. Por exemplo, para além da execução de várias instâncias do *master* em paralelo é ainda possível implementar o próprio *master* de forma paralela, como foi demonstrado nos testes do primeiro protótipo utilizando *pthreads*.

Esta liberdade de implementação da função objetivo permite ainda realizar um dos objetivos desta implementação, que é utilizar o Octave para executar código

Matlab nos nós onde não está instalado o Matlab, não sendo assim necessárias licenças adicionais.

3.2.2 Operações disponíveis

A interface final da plataforma possui assim cinco comandos distintos que permitem ao utilizador tirar partido da paralelização. O comando *init* (Figura 3.4) prepara o ambiente para a utilização da paralelização, iniciando assim os servidores que vão ser utilizados e estabelecendo os canais de comunicação entre o ficheiro *mex* e o servidor principal através de *named pipes*. O número de servidores que serão utilizados é dado como argumento neste comando.

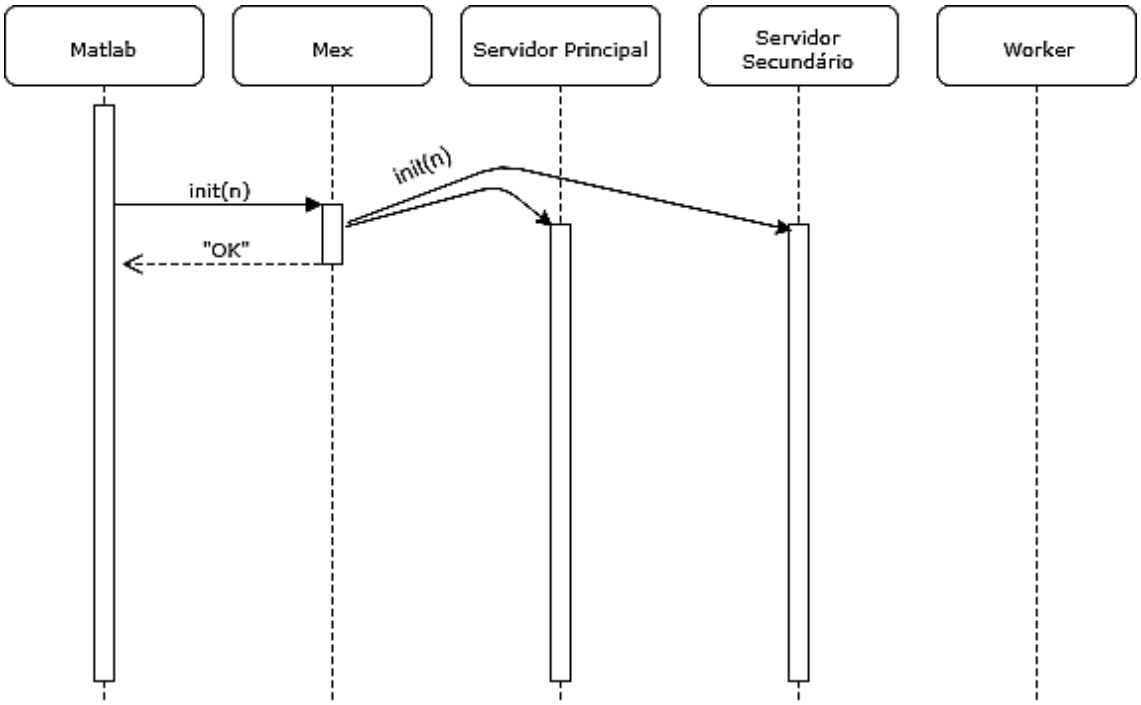

Figura 3.4: Sequência de comunicação utilizada pelo comando *init*

Note-se que o servidor principal e os servidores secundários são executados em simultâneo porque são o mesmo programa, apenas com funcionalidades diferentes consoante o número do processo MPI. Nos diagramas apresentados é ainda possível constatar que a execução do ficheiro *mex* termina sempre que é devolvido o resultado ao Matlab.

Uma vez inicializada a plataforma ficam disponíveis as restantes funcionalidades.

O comando *start* (Figura 3.5) inicializa uma computação em *background*, sendo possível executar várias de forma paralela. É possível ao utilizador utilizar este comando com um número arbitrário de argumentos. O servidor principal atribui

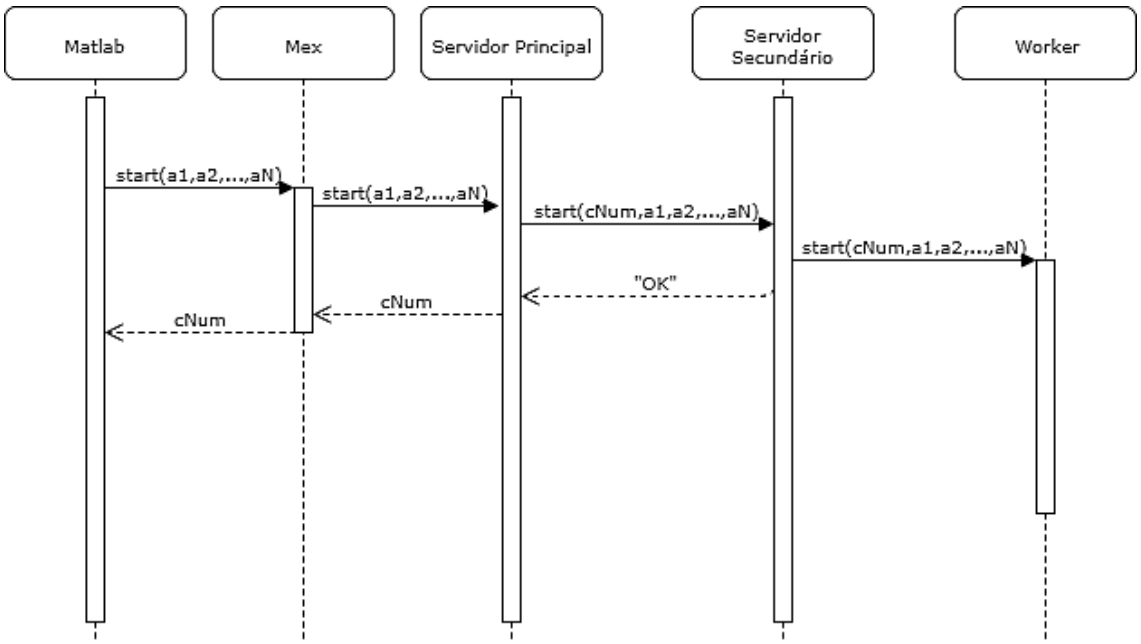

Figura 3.5: Sequência de comunicação utilizada pelo comando *start*

um número de computação a essa execução, transmitindo-o para o próximo servidor secundário disponível, guardando a que servidor corresponde o número da computação gerado. O número de computação é também guardado no servidor secundário para este saber que computações lhe competem para posteriormente devolver os resultados quando pedido ou abortar computações. O servidor secundário executa assim o *master* com os argumentos fornecidos e o número de computação da execução, para que este possa escrever os resultados no ficheiro correspondente. É ainda guardado o PID (*Process Identification Number*) da execução do *master* para o caso de ser dada a instrução para abortar a computação. Por fim é devolvido ao utilizador o número de computação gerado que corresponde à computação que mandou executar.

É possível obter os resultados de uma computação com o comando *result* (Figura 3.6), dando como argumento o número de computação atribuído quando esta foi iniciada. O servidor principal verifica a que servidor secundário corresponde o número de computação pedido e é pedido o resultado, que vai ser lido ao ficheiro correspondente àquela computação pelo servidor secundário. É por fim devolvido ao utilizador, sendo na forma de uma matriz vazia quando a computação ainda não terminou.

Para abortar computações que estão a decorrer pode ser utilizado o comando *stop* (Figura 3.7), utilizando como argumento o número da computação que se deseja terminar.

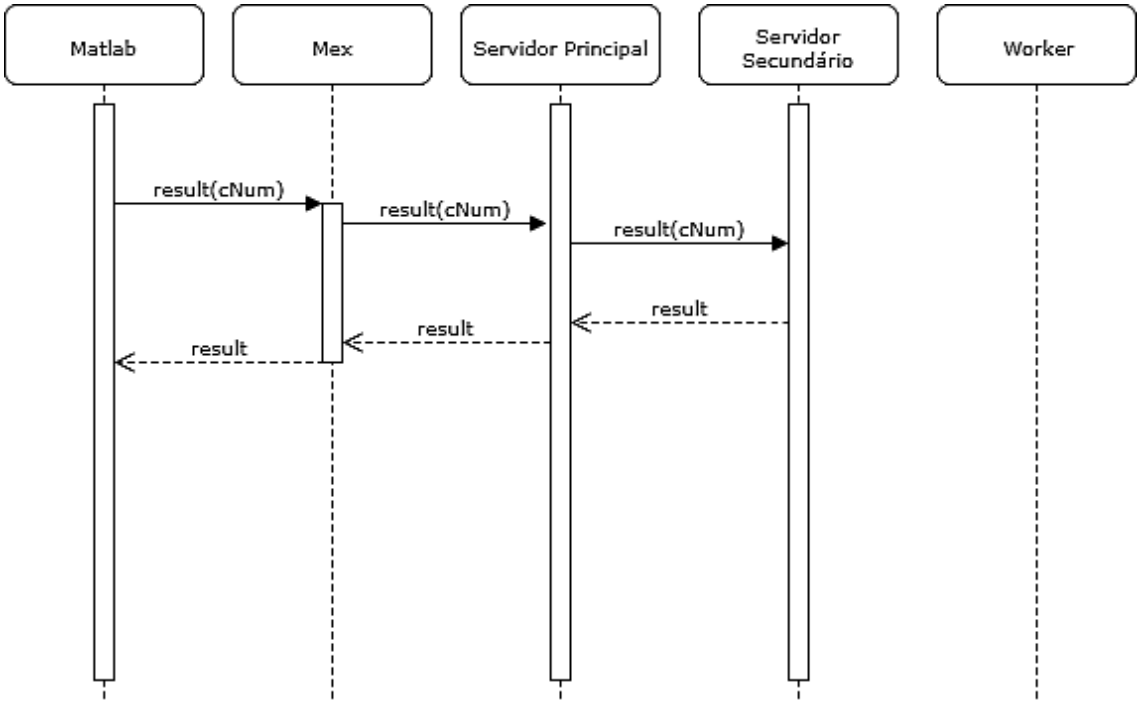

Figura 3.6: Sequência de comunicação utilizada pelo comando *result*

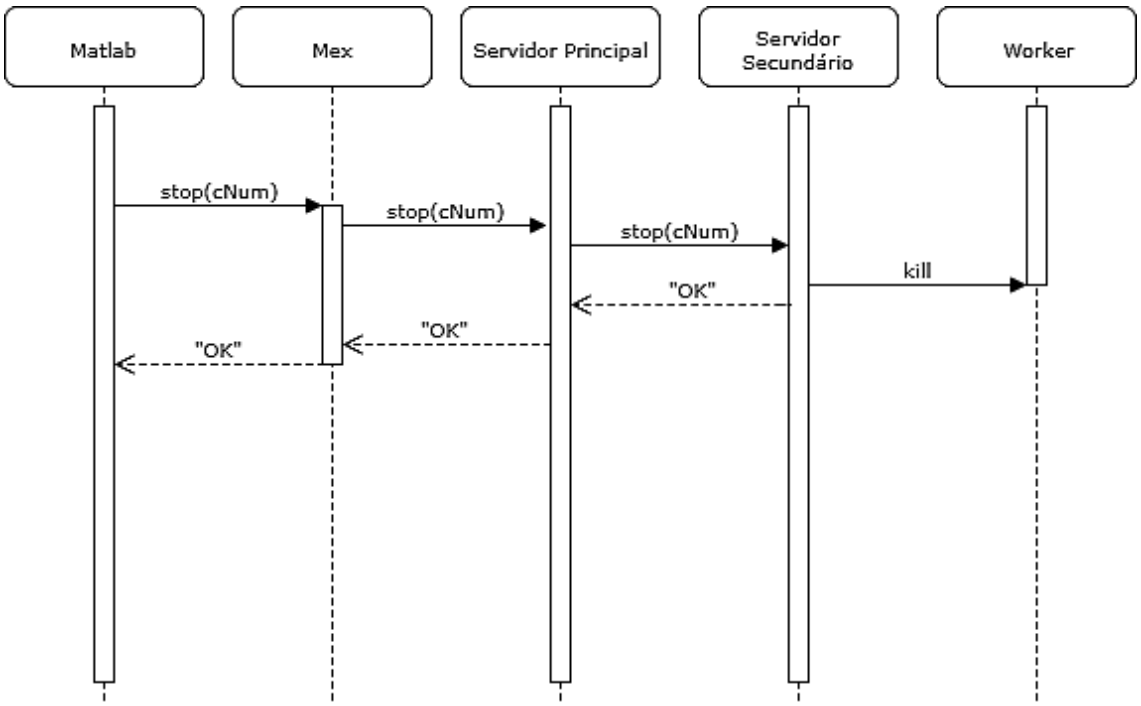

Figura 3.7: Sequência de comunicação utilizada pelo comando *stop*

A forma de funcionamento é parecida com a do comando *result*: o servidor principal verifica a que servidor secundário corresponde o número de computação e é lhe pedido para abortar a mesma. Tendo o PID do processo correspondente à computação identificada pelo número de computação é emitido um sinal para terminar o processo.

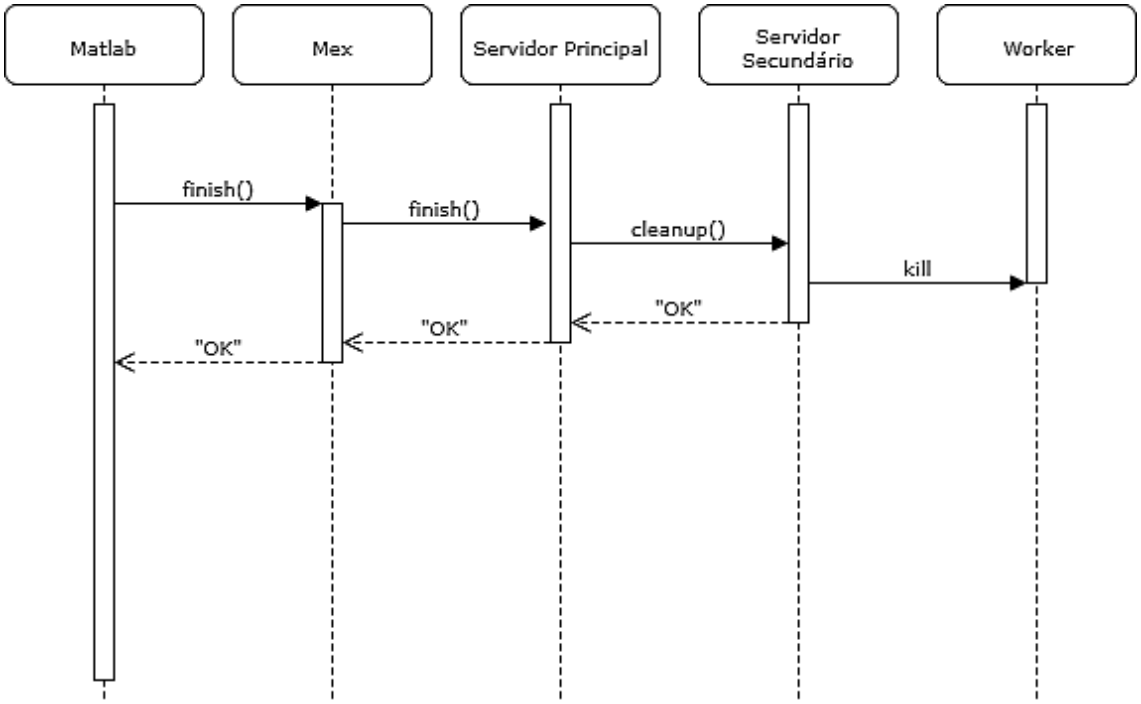

Figura 3.8: Sequência de comunicação utilizada pelo comando *finish*

Por fim o comando *finish* (Figura 3.8) encerra o ambiente da plataforma. Quando o servidor principal recebe este comando é enviado o comando *cleanup* aos servidores secundários. Ao receber este comando os servidores secundários vão abortar todas as computações que ainda estejam a decorrer, e limpar todos os ficheiros de resultados correspondentes. De seguida, informam o servidor principal que estão prontos e terminam. Quando o servidor principal recebe a mensagem de fim de todos os servidores secundários também termina após enviar a mensagem ao *mex*, que irá proceder à limpeza dos canais de comunicação (nomeadamente os *named pipes*). Desta forma nunca são deixados vestígios de computações anteriores, tendo sempre um ambiente limpo igual ao inicial.

3.3 Exposição e análise de resultados

Terminada a implementação da plataforma para computações paralelas independentes para o Matlab foram realizados testes a fim de medir o desempenho da

mesma em diferentes condições de carga e *hardware*. Estes testes decorreram em dois ambientes distintos:

- Ambiente local - computador com quatro processadores e 8GB de memória RAM.
- Ambiente distribuído - *cluster* de quatro nós, cada um com oito processadores e 16GB de memória RAM.

Os testes decorreram na versão R2017a do Matlab em ambos os ambientes, sendo também os dois com sistema operativo Linux (Ubuntu 16.04).

Os testes consistem em executar um certo número de computações que não fazem nada, apenas escrevem um ficheiro em branco indicando que terminaram. Desta forma conseguimos aferir o tempo pago pela paralelização quando é utilizada a plataforma.

Os resultados dos testes foram registados obtendo a média de 5 medições consecutivas em cada teste, num total de 150 execuções.

3.3.1 Ambiente local

No ambiente local testou-se a plataforma utilizando dois e quatro processadores (Tabelas 3.2 e 3.3 respetivamente).

Computações	10	40	160	640	2560
Tempo (segundos)	0,58	0,88	1,78	8,39	28,37

Tabela 3.2: Resultados da execução da plataforma no ambiente local com dois processadores

Computações	10	40	160	640	2560
Tempo (segundos)	0,24	0,64	1,79	7,81	29,43

Tabela 3.3: Resultados da execução da plataforma no ambiente local com quatro processadores

Utilizando os dados obtidos foi gerado um gráfico (Figura 3.9) para melhor analisar os resultados dos testes.

Analisando o gráfico concluímos que a utilização de dois ou quatro processadores não teve efeitos relevantes nos tempos de execução. É também importante referir que no pior caso (2560 computações), se apenas se utilizar o Matlab com um simples ciclo 2560 iterações sem realizar nada, demora menos de 1 milissegundo. No caso da plataforma, a ação "não fazer nada"traduz-se na realidade em:

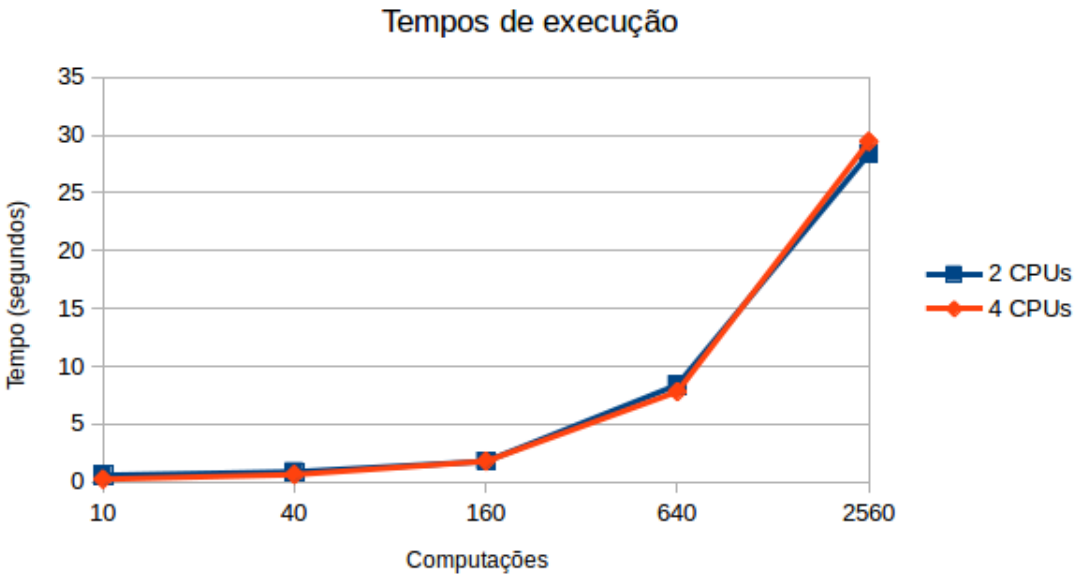

Figura 3.9: Tempos de execução da plataforma no ambiente local

- Inicialização dos *named pipes* e do servidor (principal e secundários) com MPI.
- Execução de 2560 instâncias do programa *master* em que cada uma cria e escreve no seu próprio ficheiro.
- 2560 aberturas de ficheiros para verificar que cada computação já terminou.
- Limpar todos os ficheiros utilizados pela plataforma.

Ter isto em conta reforça o facto de que apenas vale a pena utilizar paralelização em situações que o justifiquem, seja utilizando a plataforma ou outra solução para o mesmo efeito.

3.3.2 Ambiente distribuído

No ambiente distribuído foi utilizada a mesma lógica utilizada no ambiente local para realização de testes. Os testes foram realizados utilizando os quatro nós do *cluster* disponíveis, variando o número de processadores utilizados em cada nó, de modo a variar o número de processadores utilizados com o número de computações realizadas. Foram sempre utilizados os quatro nós para todos os testes, distribuindo os processadores utilizados de maneira igual entre si de modo a testar também a comunicação entre os vários nós:

- 1 processador por nó - total de 4 processadores utilizados (Tabela 3.4).
- 2 processadores por nó - total de 8 processadores utilizados (Tabela 3.5).
- 4 processadores por nó - total de 16 processadores utilizados (Tabela 3.6).
- 8 processadores por nó - total de 32 processadores utilizados (Tabela 3.7).

Computações	10	40	160	640	2560
Tempo (segundos)	1,95	3,37	9,35	32,95	125,25

Tabela 3.4: Resultados da execução da plataforma no ambiente distribuído com um processador por nó (total de 4 processadores)

Computações	10	40	160	640	2560
Tempo (segundos)	1,88	3,07	6,95	27,23	102,8

Tabela 3.5: Resultados da execução da plataforma no ambiente distribuído com dois processadores por nó (total de 8 processadores)

Computações	10	40	160	640	2560
Tempo (segundos)	1,89	2,75	6,52	22,94	88,82

Tabela 3.6: Resultados da execução da plataforma no ambiente distribuído com quatro processadores por nó (total de 16 processadores)

Computações	10	40	160	640	2560
Tempo (segundos)	1,94	2,82	6,13	20,04	80,44

Tabela 3.7: Resultados da execução da plataforma no ambiente distribuído com oito processadores por nó (total de 32 processadores)

Mais uma vez foi gerado um gráfico (Figura 3.10) a partir dos resultados obtidos de modo a fazer uma análise comparativa no mesmo referencial.

No caso do ambiente distribuído é evidente que existe uma vantagem no tempo de execução ao utilizar mais processadores à medida que se realizam mais computações. Mais uma vez, realizar um ciclo 2560 iterações sem realizar nada utilizando apenas o Matlab demora menos de um segundo. Neste caso, para além das ações descritas no ambiente local existe ainda a comunicação entre os quatro nós distribuídos do *cluster*.

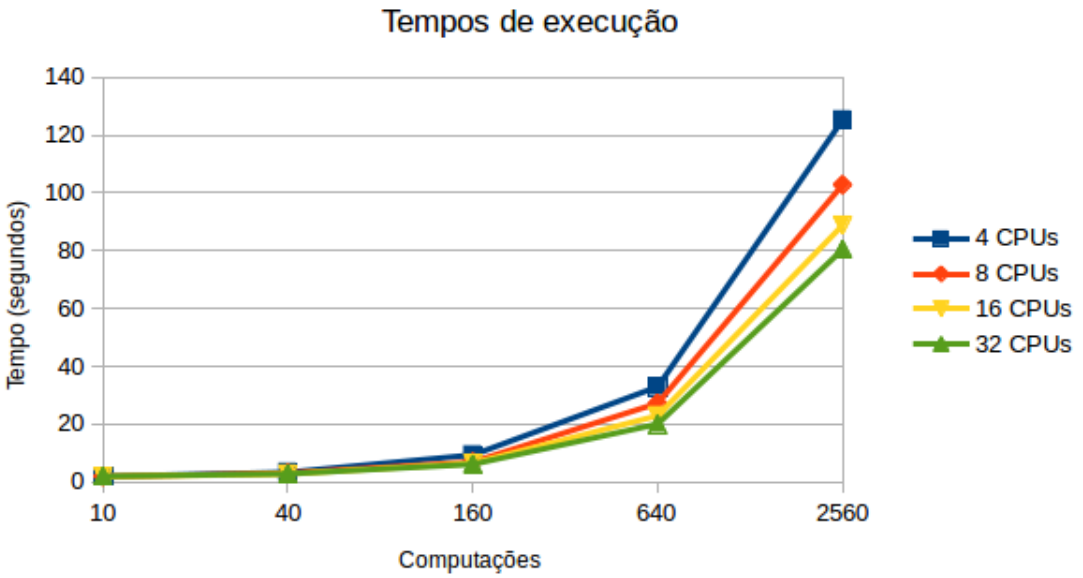

Figura 3.10: Tempos de execução da plataforma no ambiente distribuído

3.3.3 Conclusões

Ao analisar os resultados obtidos durante os testes podemos concluir que a utilização da paralelização não acontece sem custos:

- Inicialização dos *named pipes* e do servidor (principal e secundários) com MPI.
- Execução de tantas instâncias do programa *master* quanto o número de computações.
- Tantas escritas de ficheiros com resultados quanto o número de computações.
- Tantas consultas de resultados quanto o número de computações.
- Limpar todos os ficheiros utilizados pela plataforma.

No entanto apesar do custo da paralelização, quando aplicada a problemas que podem tirar partido dela (como por exemplo múltiplas ações demoradas que podem ser executadas independentemente umas das outras) poderá trazer vantagens, traduzidas numa redução do tempo total de execução do problema. Um destes exemplos é o algoritmo SID-PSM, abordado no capítulo seguinte.

É ainda visível uma redução de cerca de 35% no tempo total de execução no ambiente distribuído quando utilizados 32 processadores em vez de apenas 4

para realizar 2560 computações, sendo este o caso testado mais exigente. O caso típico de utilização da plataforma para paralelização será nestes termos, ou seja, com número elevado de computações.

Capítulo 4

Implementação SID-PSM

O SID-PSM é um algoritmo desenvolvido pela professora Ana Luísa Custódio do Departamento de Matemática da FCT NOVA que procura resolver problemas de otimização não-lineares com ou sem restrições, utilizando métodos sem derivadas [2]. Sendo um algoritmo que implementa um método de pesquisa em padrão, o que o distingue em relação a outras implementações é o número reduzido de cálculos da função objetivo necessários para calcular o resultado final. Sendo o SID-PSM um algoritmo que procura resolver problemas de otimização é utilizado em diversas áreas da ciência como por exemplo astrofísica, medicina, economia ou engenharia mecânica, que têm a necessidade de resolver problemas deste tipo.

Neste capítulo será abordado o algoritmo e os seus fundamentos, bem como de que forma se pode otimizar, contextualizando assim o objetivo final deste trabalho. É ainda utilizada a plataforma para computações paralelas independentes para o Matlab descrita no capítulo anterior para a paralelização do algoritmo SID-PSM de modo a realizar testes e analisar resultados, aferindo se a plataforma se adequa à utilização com um exemplo real.

4.1 Arquitetura e funcionamento

Um algoritmo de pesquisa em padrão consiste na procura do ponto ótimo de uma função objetivo a partir de um ponto inicial, dado como argumento. A pesquisa é feita tendo em conta um valor de alfa, que vai influenciar o tamanho da área onde será feita a pesquisa. O algoritmo pode ter várias condições para terminar, normalmente sendo quando o valor de alfa fica abaixo do valor definido como

critério de paragem. Para a execução do algoritmo são ainda necessárias bases positivas e grelhas:

- Bases positivas - São o conjunto de vetores que determinam a direção dos pontos para os quais testar a função a partir do ponto de teste corrente. O valor de alfa serve para manipular o tamanho dos vetores ao longo da execução do algoritmo.
- Grelhas - Conjunto de pontos próximos do ponto de teste corrente.

Sendo o algoritmo SID-PSM baseado num método de pesquisa em padrão, é composto por três fases:

- Inicialização - São atribuídos os valores do ponto inicial e de alfa, é gerada a base positiva e a grelha para este ponto.
- Procura (opcional) - Tentar encontrar um ponto na grelha para o qual o valor da função objetivo seja melhor que o ponto inicial. Em caso de sucesso duplica o valor de alfa e repete o passo de procura para o novo ponto, caso contrário segue para o passo de sondagem.
- Sondagem - Avaliar a função objetivo nos pontos circundantes ao ponto de teste, obtidos através dos vetores da base positiva e tendo em conta o valor de alfa. Em caso de sucesso duplica o valor de alfa e volta ao passo de procura para o novo ponto. Em caso de insucesso diminui para metade o valor de alfa e volta ao passo de procura. O passo de sondagem garante a convergência do algoritmo.

4.2 Implementação em MATLAB

O SID-PSM distingue-se dos restantes algoritmos devido à utilização de fundamentos e propriedades matemáticas que não são relevantes no contexto deste trabalho. Como tal, serão omitidos para melhor exposição do problema em questão.

A implementação do algoritmo SID-PSM em Matlab segue a estrutura de três passos abordada na secção anterior. Possui um passo de inicialização, onde são inicializadas as variáveis e matrizes que serão utilizadas durante todo o processo. Possui ainda um passo de procura e um passo de sondagem, que funcionam tal como descrito anteriormente. O passo de sondagem consiste num ciclo em que vão sendo sequencialmente calculados novos pontos tendo em conta os vetores de

pesquisa, e onde é avaliada a função objetivo. Uma vez encontrado um valor da função objetivo melhor que o atual, este ciclo termina. Caso contrário o ciclo só termina depois de analisar a função objetivo em todos os pontos calculados com os vetores.

Uma diferença relevante que caracteriza a implementação do SID-PSM é a ordenação dos vetores utilizados para pesquisar pontos para os quais o valor da função objetivo seja melhor que a corrente. Concretamente, isto quer dizer que, em caso de sucesso numa das iterações do ciclo de sondagem, o vetor que foi utilizado para calcular o novo ponto é o primeiro a ser utilizado na próxima execução do ciclo, promovendo assim as direções mais promissoras.

Outro aspeto relevante da implementação do algoritmo SID-PSM é o facto do ciclo de sondagem ser oportunista. Isto quer dizer que termina o ciclo assim que encontra uma solução melhor que a atual, mas não necessariamente a melhor. Num exemplo concreto, supondo que existem seis pontos para avaliar a função objetivo e é encontrado um valor melhor que o atual ao analisar o terceiro ponto, o ciclo termina de imediato, havendo a hipótese de existir um melhor valor para a função objetivo nos quarto, quinto ou sexto ponto.

Podemos nesta altura identificar o ponto onde a paralelização seria vantajosa quando aplicada ao algoritmo SID-PSM: no passo de sondagem. O cálculo da função objetivo é na maioria dos casos a operação mais pesada a nível computacional, pelo que é de esperar que o tempo total de execução do algoritmo seja reduzido se o cálculo da função objetivo para os vários pontos no passo de sondagem for efetuado de forma paralela em vez de sequencial.

Para além dos ganhos esperados ao paralelizar o passo de sondagem, espera-se ainda que estes sejam maiores nos casos em que o passo de sondagem encontra um sucesso. Neste caso, são abortadas as avaliações que ainda não foram calculadas, evitando assim a utilização de recursos de forma desnecessária.

4.3 Adaptação do algoritmo

A operação mais pesada do algoritmo SID-PSM é o cálculo da função objetivo num determinado ponto. No passo de sondagem (Listagem 4.1) é calculado o valor da função objetivo para vários pontos de maneira iterativa, na tentativa de obter um valor melhor que o atual. Tendo estes aspetos em conta foi utilizada a plataforma para computações paralelas desenvolvida no passo de sondagem do algoritmo SID-PSM.

Listagem 4.1: Passo de sondagem do algoritmo SID-PSM (pseudocódigo)

```
1  vetores = ordenar_vetores(vetores);
2  total_vetores = vetores.length();
3  vetor_corrente = 1;
4  sucesso = false;
5  /** passo de sondagem */
6  while(!successo && vetor_corrente <= total_vetores) {
7    x_temp = calc_ponto(x_corrente, vetores[vetor_corrente]);
8    if(admissivel(x_temp)) {
9      f_temp = calc_funcao_objetivo(x_temp);
10     if(f_temp < f_corrente) {
11       sucesso = true;
12       x_corrente = x_temp;
13       f_corrente = f_temp;
14     }
15   }
16   vetor_corrente++;
17 }
```

4.4 Passo de sondagem em paralelo

Para adaptar o passo de sondagem para funcionar em paralelo utilizando a plataforma (Listagem 4.2), este foi dividido em quatro fases:

- Calcular e guardar os pontos para os quais vai ser calculada a função objetivo.
- Iniciar o cálculo da função objetivo para os pontos guardados, em paralelo.
- À medida que as computações forem terminadas, analisar o resultado e agir consoante o valor obtido.
- Em caso de ser encontrado um sucesso, abortar as funções objetivo que ainda não terminaram de calcular.

Desta forma, continua a ser aproveitada a ordenação dos vetores para calcular os pontos, sendo estes guardados pela mesma ordem. Isto garante que vão começar a ser calculados os valores da função objetivo para os pontos mais promissores primeiro. Da mesma forma, quando se esperam pelos resultados, são verificados pela ordem em que foram mandados calcular. Não são guardados os pontos que ficam fora da região admissível do problema (no caso de este ter restrições) uma vez que para estes não é calculada a função objetivo.

Listagem 4.2: Passo de sondagem do algoritmo SID-PSM em paralelo utilizando a plataforma (pseudocódigo)

```
1  vetores = ordenar_vetores(vetores);
2  total_vetores = vetores.length();
3  vetor_corrente = 1;
4  pontos = [];
5  /** guardar os pontos para calcular a funcao objetivo */
6  while(vetor_corrente <= total_vetores) {
7    x_temp = calc_ponto(x_corrente, vetores[vetor_corrente]);
8    if(admissivel(x_temp)) {
9      pontos.push(x_temp);
10   }
11   vetor_corrente++;
12 }
13 total_pontos = pontos.length();
14 /** iniciar o calculo da funcao objetivo para os pontos guardados */
15 for(i = 0; i < total_pontos; i++) {
16   identificador = calc_funcao_objetivo_paralelo(pontos[i]);
17   guardar_identificador(pontos[i], identificador);
18 }
19 sucesso = false;
20 ciclo_corrente = 1;
21 /** tratar os resultados a medida que vao terminando */
22 while(!sucesso && ciclo_corrente <= total_pontos) {
23   x_temp = null;
24   f_temp = null;
25   /** obter o primeiro calculo que acabar */
26   while(x_temp == null && f_temp == null) {
27     for(i = 0; i < total_pontos; i++) {
28       identificador = obter_identificador(pontos[i]);
29       if(calculo_terminou(identificador) && !tratado(pontos[i])) {
30         x_temp = pontos[i];
31         f_temp = resultado(identificador);
32         tratado(x_temp, true); /** marcar como tratado */
33       }
34     }
35   }
36   if(f_temp < f_corrente) {
37     sucesso = true;
38     x_corrente = x_temp;
39     f_corrente = f_temp;
40   }
41   ciclo_corrente++;
42 }
43 /** abortar os calculos da funcao objetivo que ainda nao terminaram */
```

```
44 for(i = 0; i < total_pontos; i++) {
45   if(!tratado(pontos[i])) {
46     identificador = obter_identificador(pontos[i]);
47     abortar_calculo(identificador);
48   }
49 }
```

Ao obter o resultado da função objetivo para um ponto na terceira fase, este é marcado como tratado para não ser abordado nas seguintes iterações. No final do passo de sondagem são abortadas as computações em paralelo que ainda não terminaram. Desta forma evita-se a utilização desnecessária de recursos computacionais.

É preciso ter em conta que o ciclo de sondagem é oportunista, o que significa acaba o ciclo assim que é encontrado o resultado de uma função objetivo melhor que o atual. Sendo o passo de sondagem do algoritmo SID-PSM original executado de forma sequencial, isso significa que mesmo que o primeiro resultado melhor encontrado não seja o melhor possível, o resultado final será sempre o mesmo, o que faz com que seja um algoritmo determinista. Isto significa que sempre que se executar o algoritmo com as mesmas variáveis iniciais, o resultado será sempre igual. Ao executar o passo de sondagem em paralelo, isto já não se verifica, uma vez que pode terminar o cálculo da função objetivo melhor de um ponto diferente daquele que seria primeiro encontrado na versão sequencial. Por exemplo, supondo que existem seis pontos para avaliar a função objetivo e é encontrado um valor melhor que o atual ao analisar o terceiro ponto na versão sequencial, o ciclo termina de imediato, podendo haver um valor melhor nos pontos quatro, cinco ou seis. Na versão em paralelo e na mesma situação, pode acontecer que o cálculo da função para o ponto cinco termine primeiro, sendo esse o ponto considerado para o ciclo seguinte. Isto quer dizer que o algoritmo SID-PSM quando implementado em paralelo não é determinista, podendo obter diferentes resultados em execuções com as mesmas variáveis iniciais. Neste caso o algoritmo continua a ser válido, uma vez que continua a manter o seu carácter oportunista na fase de sondagem, e a convergência continua a ser garantida de mesma forma neste passo.

4.5 Exposição e análise de resultados

Uma vez implementado o algoritmo SID-PSM com o passo de sondagem em paralelo foram realizados testes a fim de medir o desempenho do algoritmo original versus o algoritmo modificado. Estes testes decorreram em dois ambientes distintos:

- Ambiente local - computador com quatro processadores e 8GB de memória RAM.
- Ambiente distribuído - *cluster* de quatro nós, cada um com oito processadores e 16GB de memória RAM.

Os testes decorreram na versão R2017a do Matlab em ambos os ambientes, sendo também os dois com sistema operativo Linux (Ubuntu 16.04). A versão do algoritmo SID-PSM testada foi a 1.3.

Para a função objetivo foi utilizado o problema *Chrosen* [24] (Figura 4.1) implementado em linguagem C;

$$F(\underline{x}) = \sum_{i=1}^{n-1} \left\{ (x_i^2 + x_n^2)^2 - 4x_i + 3 \right\}, \qquad \underline{x} \in \mathcal{R}^n$$

Figura 4.1: Problema *Chrosen* (página 12 da referência [21])

Foi escolhido o problema *Chrosen* porque podemos controlar o número de dimensões do problema, podendo criar diferentes situações de teste tanto a nível do número de processadores utilizados como do número de dimensões do problema, e consequentemente os cálculos a si associados.

Como parâmetro de inicialização do algoritmo foi dado um vetor, em que todas as posições têm o valor um (1), tantos quanto o número de dimensões que pretendemos que o problema tenha. Este parâmetro foi escolhido para os testes porque faz com que o algoritmo SID-PSM nunca encontre um sucesso. Isto é importante para os testes porque desta forma o número de computações realizadas tanto pela versão original (sequencial) como pela versão em paralelo vai ser o mesmo. Em casos em que algoritmo encontre sucessos durante a sua execução, o número total de avaliações da função objetivo entre os dois seria diferente, o que não seria útil para fazer uma comparação direta entre os dois. Consoante o número de dimensões do problema nos testes seguintes, o número de avaliações respetivo é:

- 2 dimensões - 119 avaliações da função objetivo.
- 4 dimensões - 187 avaliações da função objetivo.
- 8 dimensões - 323 avaliações da função objetivo.
- 16 dimensões - 595 avaliações da função objetivo.
- 32 dimensões - 1139 avaliações da função objetivo.

Em todos os casos o número de iterações do algoritmo foi 17, visto que o algoritmo terminava devido ao valor de alfa passar abaixo do limite.

Os testes foram conduzidos de duas maneiras distintas:

- Execução normal de ambos os algoritmos.
- Induzido um atraso de 200 milissegundos no cálculo da função objetivo para simular uma avaliação mais pesada, visto que é nestes casos que a paralelização se torna vantajosa.

Os resultados dos testes foram registados obtendo a média de 5 medições consecutivas em cada teste, num total de 400 execuções.

É ainda importante referir que os ganhos obtidos podem ser ainda maiores nos casos em que o algoritmo SID-PSM é executado em circunstâncias normais, isto é, o passo de sondagem encontra sucessos e as computações não terminadas são abortadas. Estes testes mostram as condições de pior caso do SID-PSM, onde nunca é encontrado um sucesso no passo de sondagem.

4.5.1 Ambiente local

No ambiente local começou-se por testar o algoritmo SID-PSM original de modo a obter uma base de comparação (Tabela 4.1). Podemos desde já constatar que o tempo de execução total quando aplicado o atraso de 200 milissegundos coincide com o número de computações realizadas. Por exemplo, quando o problema tem 8 dimensões são realizadas 323 avaliações da função objetivo, multiplicando pelo atraso de 0,2 segundos obtemos um resultado de 64,6 segundos, somado ao tempo de execução da versão sem atraso (0,12 segundos) obtemos o valor de 64,72 segundos, muito próximo do valor obtido nos testes. A mesma lógica pode ser aplicada aos restantes resultados, que continua a coincidir.

Dimensões	2	4	8	16	32
Tempo (segundos)	0,03	0,07	0,12	0,33	2,78
Tempo (segundos) com atraso de 200ms	23,87	37,51	64,75	120,6	231,8

Tabela 4.1: Resultados da execução do algoritmo SID-PSM original no ambiente local

Após obter o resultado base foi testada a implementação do SID-PSM em paralelo com dois e quatro processadores (Tabelas 4.2 e 4.3 respetivamente).

Neste momento podemos confirmar o que já se esperava: o algoritmo em paralelo só compensa em casos em que o peso do cálculo da função objetivo ultrapasse o peso da paralelização.

Dimensões	2	4	8	16	32
Tempo (segundos)	1,46	2,32	4,36	9,22	25,19
Tempo (segundos) com atraso de 200ms	8,54	9,45	10,81	13,27	27,95

Tabela 4.2: Resultados da execução do algoritmo SID-PSM em paralelo no ambiente local com dois processadores

Dimensões	2	4	8	16	32
Tempo (segundos)	2,56	3,65	7,11	13,66	19,21
Tempo (segundos) com atraso de 200ms	8,57	9,46	11,05	13,76	22,54

Tabela 4.3: Resultados da execução do algoritmo SID-PSM em paralelo no ambiente local com quatro processadores

Após obter todos os resultados foram gerados gráficos a partir destes de forma a poder fazer a análise e comparar as duas implementações.

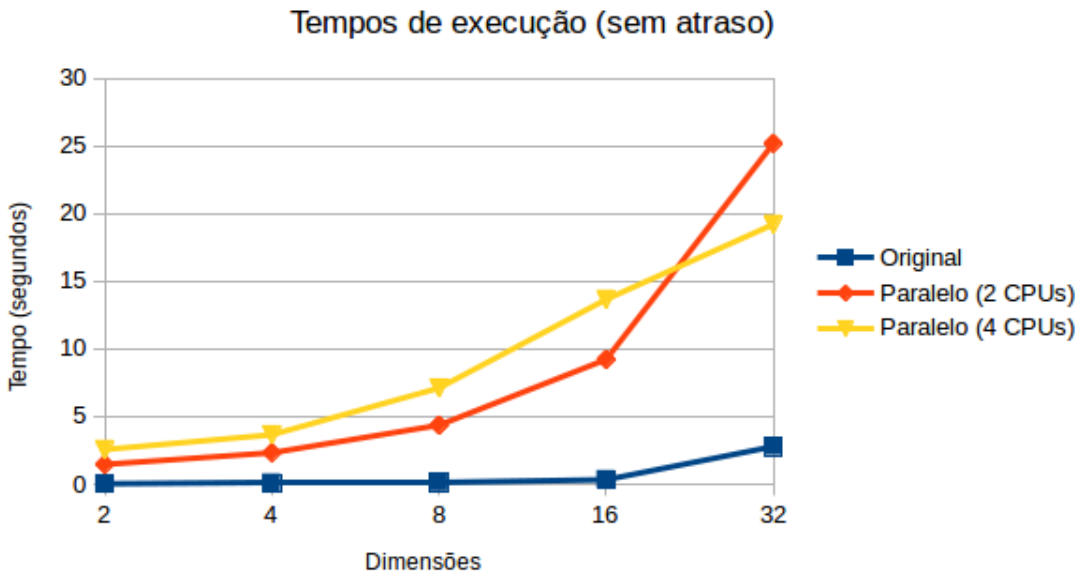

Figura 4.2: Tempos de execução do algoritmo SID-PSM no ambiente local sem atraso no cálculo da função objetivo

Analisando os gráficos das figuras 4.2 e 4.3 podemos mais uma vez observar a vantagem da utilização do algoritmo em paralelo para cálculos de funções objetivo que sejam computacionalmente pesadas.

Através da análise dos gráficos é ainda possível verificar que ao executar o algoritmo SID-PSM em paralelo com quatro processadores obtemos uma vantagem no tempo de execução quando a ordem de dimensões do problema aumenta, neste caso em concreto com 32 dimensões. Isto é possível verificar tanto na versão sem

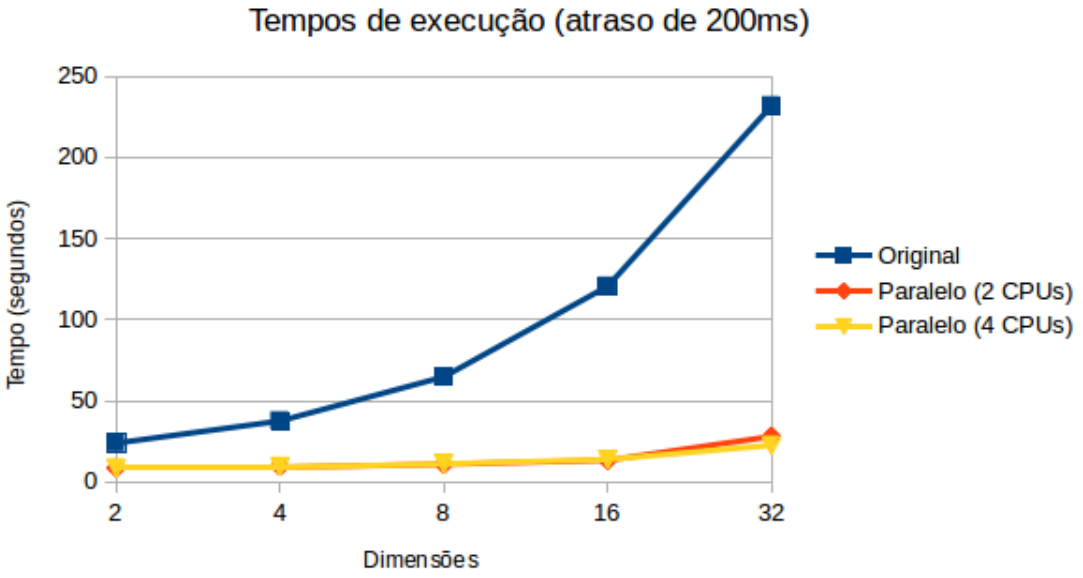

Figura 4.3: Tempos de execução do algoritmo SID-PSM no ambiente local com atraso de 200 milissegundos no cálculo da função objetivo

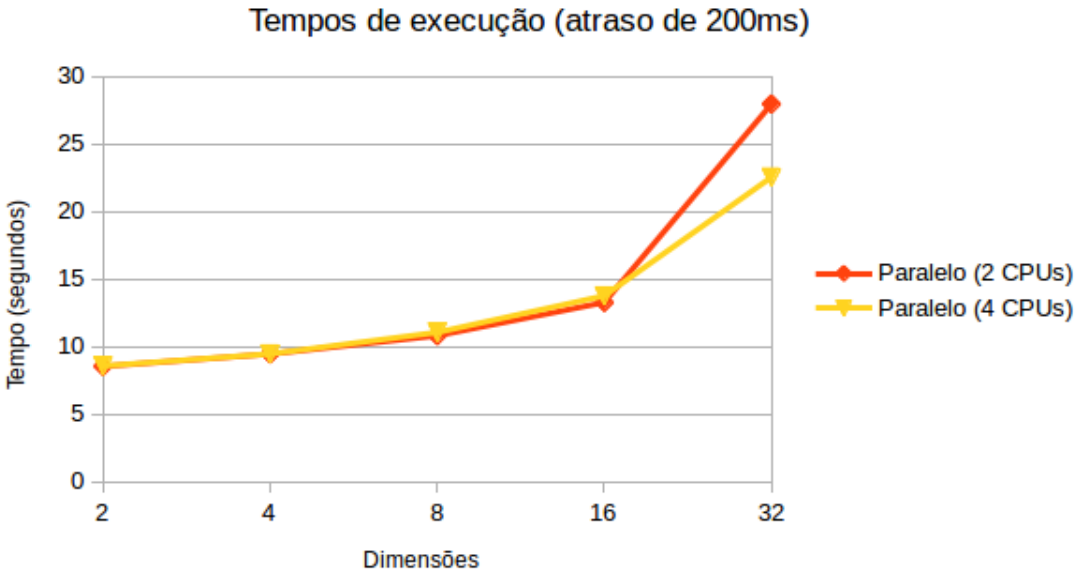

Figura 4.4: Tempos de execução do algoritmo SID-PSM em paralelo no ambiente local com atraso de 200 milissegundos no cálculo da função objetivo

atraso como na versão com atraso de 200 milissegundos, melhor observável na figura 4.4 (que resulta da figura 4.3 removendo o tempo do algoritmo original).

4.5.2 Ambiente distribuído

A lógica de execução dos testes no ambiente local foi reproduzida também no ambiente distribuído. Começou-se por obter os resultados da versão SID-PSM original para ter uma base de comparação (Tabela 4.4). Mais uma vez os tempos de execução com atraso de 200 milissegundos coincidem com o atraso induzido no cálculo da função objetivo tendo em conta o número de cálculos da função para determinado número de dimensões.

Dimensões	2	4	8	16	32
Tempo (segundos)	0,03	0,06	0,16	0,84	10,36
Tempo (segundos) com atraso de 200ms	23,91	37,49	64,83	123,2	238,3

Tabela 4.4: Resultados da execução do algoritmo SID-PSM original no ambiente distribuído

De seguida foi testada a versão em paralelo do algoritmo SID-PSM utilizando os quatro nós do *cluster* disponíveis, variando o número de processadores utilizados em cada nó, de modo a variar o número de processadores utilizados com o número de dimensões do problema. Em todos os casos foram utilizados todos os nós disponíveis independentemente do número de processadores, de modo a contemplar sempre a comunicação entre os mesmos:

- 1 processador por nó - total de 4 processadores utilizados (Tabela 4.5).
- 2 processadores por nó - total de 8 processadores utilizados (Tabela 4.6).
- 4 processadores por nó - total de 16 processadores utilizados (Tabela 4.7).
- 8 processadores por nó - total de 32 processadores utilizados (Tabela 4.8).

Dimensões	2	4	8	16	32
Tempo (segundos)	3,65	4,04	5,79	10,63	28,37
Tempo (segundos) com atraso de 200ms	10,44	11,14	12,96	17,42	36,66

Tabela 4.5: Resultados da execução do algoritmo SID-PSM em paralelo no ambiente distribuído com um processador por nó (total de 4 processadores)

A partir dos resultados obtidos foram gerados gráficos utilizando a mesma lógica utilizada no ambiente local. Continua a verificar-se mais uma vez que utilizar

Dimensões	2	4	8	16	32
Tempo (segundos)	3,36	4,15	6,39	11,55	30,93
Tempo (segundos) com atraso de 200ms	10,41	11,25	12,79	17,55	36,16

Tabela 4.6: Resultados da execução do algoritmo SID-PSM em paralelo no ambiente distribuído com dois processadores por nó (total de 8 processadores)

Dimensões	2	4	8	16	32
Tempo (segundos)	3,58	4,54	6,79	12,36	33,34
Tempo (segundos) com atraso de 200ms	10,42	11,34	13,11	17,58	37,98

Tabela 4.7: Resultados da execução do algoritmo SID-PSM em paralelo no ambiente distribuído com quatro processadores por nó (total de 16 processadores)

Dimensões	2	4	8	16	32
Tempo (segundos)	3,93	5,09	7,54	14,09	44,38
Tempo (segundos) com atraso de 200ms	10,98	11,91	14,21	20,08	50,76

Tabela 4.8: Resultados da execução do algoritmo SID-PSM em paralelo no ambiente distribuído com oito processadores por nó (total de 32 processadores)

o algoritmo SID-PSM em paralelo só compensa em casos em que o peso do cálculo da função objetivo ultrapasse o peso da paralelização. É interessante verificar que o atraso de 200ms para simular uma função mais pesada faz diferença quando se tem que tomar a decisão de utilizar ou não a paralelização.

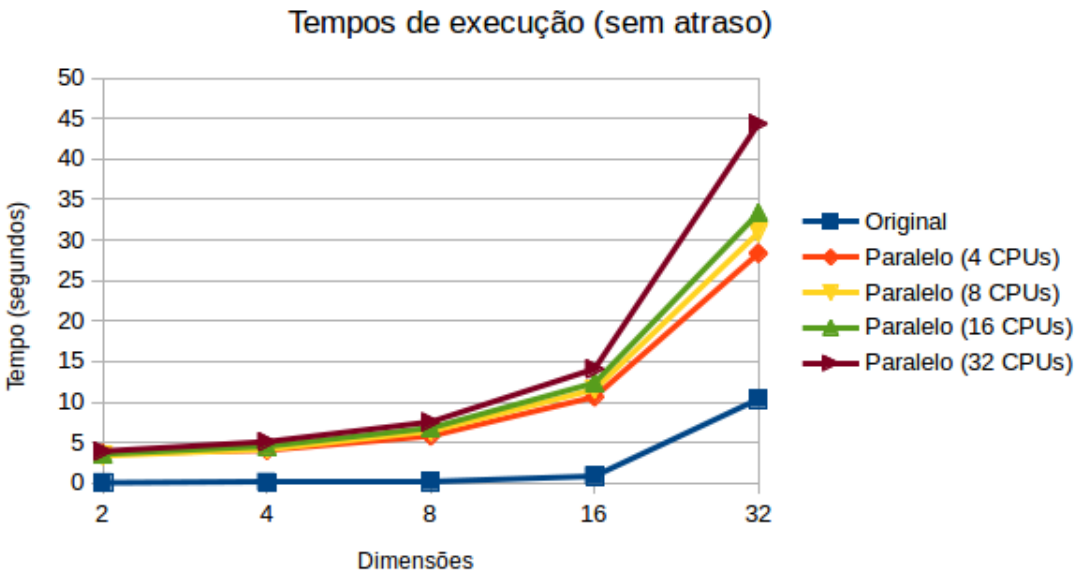

Figura 4.5: Tempos de execução do algoritmo SID-PSM no ambiente distribuído sem atraso no cálculo da função objetivo

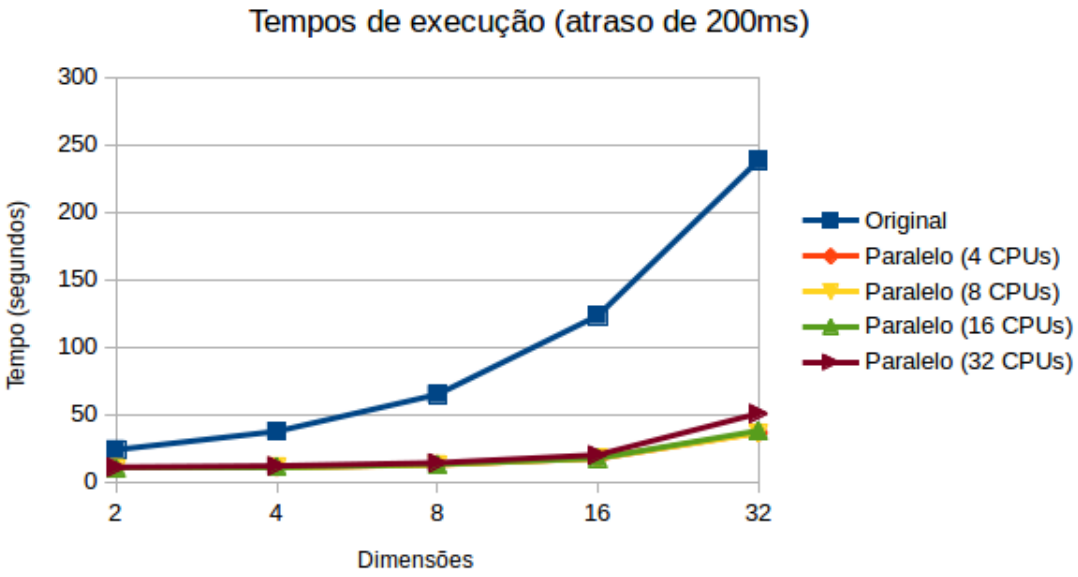

Figura 4.6: Tempos de execução do algoritmo SID-PSM no ambiente distribuído com atraso de 200 milissegundos no cálculo da função objetivo

Analisando os gráficos das Figuras 4.5 e 4.6 podemos mais uma vez observar a vantagem da utilização do algoritmo em paralelo para cálculos de funções objetivo que sejam computacionalmente pesadas, sendo assim consistente com os resultados obtidos no ambiente local.

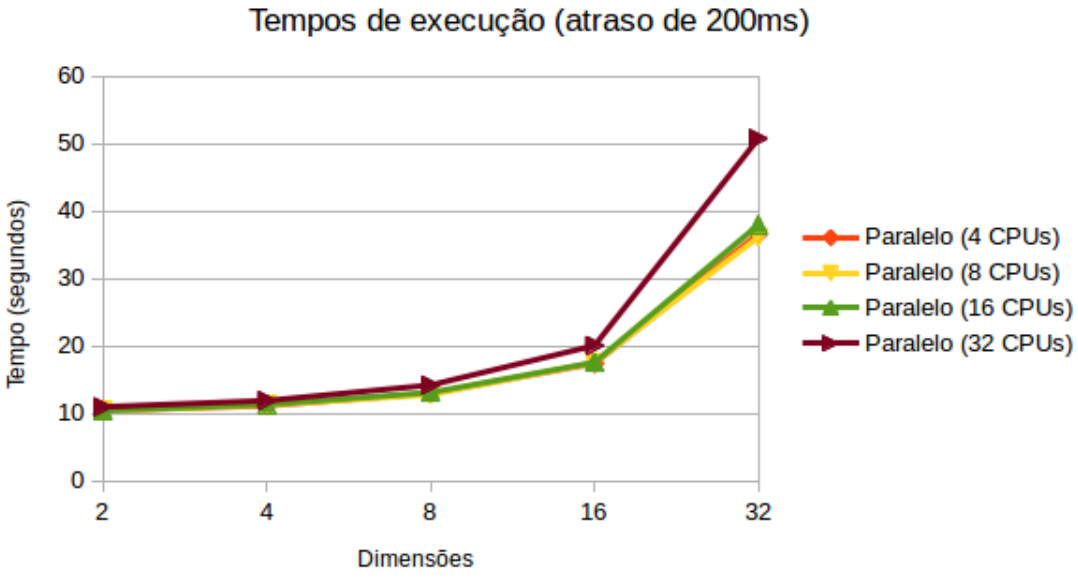

Figura 4.7: Tempos de execução do algoritmo SID-PSM em paralelo no ambiente distribuído com atraso de 200 milissegundos no cálculo da função objetivo

No caso deste ambiente distribuído pode verificar-se que a utilização de mais processadores traz incorre num tempo de execução mais longo. Este aspeto é possível verificar tanto na versão sem atraso como na versão com atraso de 200 milissegundos, melhor observável na figura 4.7 (que resulta da figura 4.6 removendo o tempo do algoritmo original). Este resultado contraria os testes realizados para a plataforma quando testada de forma independente, portanto esta discrepância deve-se à maneira como a plataforma foi utilizada para paralelizar o algoritmo SID-PSM. No entanto, é possível que mesmo assim compense utilizar um grande número de múltiplos processadores em múltiplos nós distribuídos caso o cálculo da função objetivo seja ainda mais pesada do que simulado nesta análise.

4.5.3 Exemplo real

Os testes anteriores apenas focaram-se na performance do algoritmo SID-SPM original contra o algoritmo implementado em paralelo no pior caso, sendo o número de computações e avaliações da função objetivo sempre iguais. Num caso de utilização normal, para além da diferença nos tempos de execução, a versão em paralelo pode produzir resultados diferentes, tanto no resultado final como no número de iterações e avaliações da função objetivo.

Foi utilizada uma função simples (Figura 4.8) para demonstrar que não são precisos cálculos complexos para haver uma discrepância nos resultados. O valor inicial foi um vetor com os valores -1.2 e 1;

$$F(x) = (x_2 - x_1^2)^2$$

Figura 4.8: Função objetivo utilizada no exemplo real

Foi uma vez mais induzido um atraso de 200 milissegundos no cálculo da função objetivo para simular uma computação mais lenta. Foram realizados três tipos diferentes de testes, num total de 15 execuções:

- Execução normal do algoritmo SID-PSM original.
- Execução do algoritmo SID-PSM em paralelo utilizando 4 processadores mas eliminando o fator de divergência devido ao oportunismo do passo de sondagem. As funções eram calculadas em paralelo mas esperava-se que terminassem por ordem.
- Execução do algoritmo SID-PSM em paralelo utilizando 4 processadores sem restrições no funcionamento.

A execução do algoritmo original serviu de base de comparação para os restantes testes, obtendo como resultados:

- 35 iterações do algoritmo.
- 18 sucessos encontrados.
- 143 avaliações da função objetivo.
- Resultado final de 5,60e-20
- Tempo de execução de 28,77 segundos.

De seguida foi testado o algoritmo em paralelo onde não existem discrepâncias de oportunismo no passo de sondagem. Os resultados foram iguais (como seria de esperar) à exceção do tempo de execução que foi de 13,96 segundos, o que resulta numa redução do tempo de execução de cerca de 51% no tempo de execução.

Com este teste podemos constatar uma redução de 51% no tempo de execução sem qualquer redução no número de iterações, cálculos da função objetivo ou abortar computações.

Por fim for testado o algoritmo SID-PSM sem quaisquer restrições para avaliar os diferentes resultados (Tabela 4.9).

Execução	#1	#2	#3	#4	#5
Iterações	35	34	35	35	31
Sucessos	18	17	18	18	14
Avaliações	143	142	143	143	139
Resultado	5,60e-20	4,80e-22	5,60e-20	1,55e-18	1,28e-18
Tempo (segundos)	14,12	13,83	14,02	14,04	13,42

Tabela 4.9: Resultados da execução do algoritmo SID-PSM em paralelo no exemplo real utilizando 4 processadores

Analisando a tabela de resultados podemos afirmar as seguintes conclusões:

- Nos testes 2 e 5 houve uma redução no número de iterações, sucessos encontrados e avaliações da função objetivo. O tempo de execução também foi mais baixo nestes casos.
- O resultado do teste 2 é melhor que o obtido no algoritmo SID-PSM original. Por outro lado o resultado dos testes 1 e 3 são iguais, sendo o 4 e o 5 piores.

Este teste serve para demonstrar que o algoritmo SID-PSM implementado em paralelo não é determinista. Apesar de serem obtidos resultados melhores, iguais

ou piores a validade do algoritmo mantém-se uma vez que as diferenças são de ordem de grandeza muito reduzida.

4.5.4 Conclusões

Utilizando a plataforma para paralelizar o algoritmo SID-PSM permitiu ganhar vantagens nos tempos de execução. Em concreto, um típico caso de utilização do algoritmo SID-PSM onde a paralelização seria mais vantajosa é a resolução do problema *Chrosen* com 32 dimensões nos casos em que foi utilizado o atraso de 200 milissegundos para simular uma utilização normal. Neste caso obteve-se uma redução no tempo de execução de cerca de 90% no ambiente local utilizando 4 processadores. Já no ambiente distribuído a redução foi de cerca de 85% utilizando 4, 8 ou 16 processadores. Utilizar 32 processadores não se provou tão eficaz ficando pelos 78% de melhoria.

Podemos também concluir que a plataforma contribui de forma muito positiva para aumentar a eficácia do algoritmo SID-PSM. Apesar dos melhores casos atingidos durante os testes terem sido 90% no ambiente local e 85% no ambiente distribuído, estes ganhos podem ser ainda maiores em condições de utilização normal do algoritmo, nomeadamente ao abortar computações que ainda estão a decorrer quando é encontrado um sucesso no passo de sondagem, o que não aconteceu durante os testes pois foram realizados pensando no pior caso.

No entanto, estes resultados de melhoria não podem ser considerados para medir a aceleração da execução do algoritmo utilizando a plataforma face ao original, uma vez que é apenas utilizado um atraso. Servem no entanto para ilustrar que a execução em múltiplas máquinas diminui efetivamente o tempo de execução quando a função a calcular tem um tempo de execução suficientemente grande.

Quando testado com um exemplo real foi demonstrado que para além da redução do tempo de execução do algoritmo é ainda possível realizar menos iterações e menos avaliações da função objetivo, podendo por vezes conseguir obter resultados melhores que o original.

Capítulo 5

Conclusões

Neste capítulo é feita uma apreciação crítica sobre o trabalho realizado, comparando os objetivos inicialmente propostos com os que foram obtidos no final. Será também abordado trabalho a realizar no futuro.

5.1 Apreciação da contribuição da tese

Este trabalho apresentou uma plataforma para computações paralelas independentes para o Matlab. A ideia de uma plataforma surgiu quando a análise do algoritmo SID-PSM sugeria que o tempo de execução poderia ser reduzido se fosse possível utilizar paralelização no passo de sondagem. Para paralelizar o algoritmo SID-PSM seria necessário cumprir as seguintes condições:

- Lançar várias computações independentes a partir do Matlab.
- Desenvolver a solução de forma a utilizar apenas a licença base do Matlab.
- Possibilidade de abortar computações antes de estas terem terminado.
- Não ter necessidade de utilização da interface gráfica.

Devido às necessidades do algoritmo SID-PSM não foi possível utilizar nenhuma das soluções existentes atualmente para implementar a paralelização cumprindo todas as condições.

Decidiu-se então desenvolver uma plataforma que cumprisse todas as necessidades do SID-PSM que servisse de apoio à paralelização do algoritmo, mas que fosse genérica o suficiente para ser utilizada por outras soluções.

As características da plataforma resolvem todas as condições acima mencionadas para o algoritmo SID-PSM, sendo ainda possível distribuir as computações realizadas por vários nós computacionais ligados em rede.

Após o desenvolvimento e validação a plataforma foi utilizada para paralelizar o algoritmo SID-PSM para verificar com um caso prático real que poderia ser utilizada numa aplicação de contexto real, obtendo uma redução nos tempos de computação de até 90% num ambiente local e até 85% num ambiente distribuído.

5.2 Trabalho futuro

Como trabalho futuro podem ser estendias a funcionalidades disponíveis na plataforma, desenvolvendo funções que permitam ao utilizador a utilização da plataforma em contextos mais alargados, como por exemplo:

- Após lançar várias execuções em paralelo existir uma função que devolve o primeiro resultado obtido, escusando assim do utilizador ter que verificar resultados de computações individualmente. Esse resultado será marcado para não ser devolvido novamente caso se execute a mesma função outra vez.
- Em vez do utilizador ter que lançar computações individualmente ser possível passar como argumentos os dados necessários para uma série de computações de uma só vez, ficando o seu lançamento individual ao cargo da plataforma.
- Possibilidade de o utilizador poder definir se deseja esperar pelo resultado de uma computação. De momento é devolvida uma matriz vazia no caso de ainda não ter terminado a computação. Da maneira sugerida a execução ficaria bloqueada à espera do resultado.

Para além destas novas funcionalidades seria vantajoso obter testemunhos de utilização da plataforma de forma a ser adaptada com base em casos de utilização real.

Bibliografia

[1] Y. Altman. *Accelerating MATLAB Performance - 1001 tips to speed up MATLAB programs*. Chapman e Hall/CRC Press, 2014.

[2] Ana Luísa Custódio e Luís Nunes Vicente. *SID-PSM: A PATTERN SEARCH METHOD GUIDED BY SIMPLEX DERIVATIVES FOR USE IN DERIVATIVE-FREE OPTIMIZATION*. URL: http://www.mat.uc.pt/sid-psm/sid_psm_manual_1.3.pdf.

[3] L. L.N. L. Blaise Barney. *Message Passing Interface (MPI)*. URL: https://computing.llnl.gov/tutorials/mpi/.

[4] Hahn Kim, Julia Mullen, Jeremy Kepner. *Introduction to Parallel Programming and pMatlab v2.0*. URL: https://ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pMatlab_intro.pdf.

[5] D. J. Kepner. *Parallel Programming with MatlabMPI*. URL: http://www.ll.mit.edu/mission/cybersec/softwaretools/matlabmpi/matlabmpi.html.

[6] D. J. Kepner. *pMatlab: Parallel Matlab Toolbox v2.0.15*. URL: http://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pmatlab.html.

[7] J. Kepner. *Parallel Programming with MatlabMPI*. URL: https://arxiv.org/pdf/astro-ph/0107406.pdf.

[8] A. L. C. e Luís Nunes Vicente. *Métodos de Procura em Padrão em Otimização Sem Derivadas*. URL: http://ferrari.dmat.fct.unl.pt/personal/alcustodio/PresentationNOVA.pdf.

[9] Mathworks. *Create and Share Toolboxes*. URL: https://www.mathworks.com/help/matlab/matlab_prog/create-and-share-custom-matlab-toolboxes.html.

[10] Mathworks. *Introducing MEX Files*. URL: https://www.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html.

[11] Mathworks. *MATLAB distributed computing server*. URL: https://www.mathworks.com/help/mdce/.

[12] Mathworks. *Parallel Computing Toolbox*. URL: https://www.mathworks.com/help/distcomp/.

[13] Mathworks. *Products and Services*. URL: https://www.mathworks.com/products.html?s_tid=gn_ps.

[14] G. S. Mendes. *Paralelização de Algoritmos de Procura Directa Direccional*. 2016.

[15] *MPICH Overview*. URL: https://www.mpich.org/about/overview/.

[16] NVIDIA. *CUDA Parallel Programming and Computing Platform*. URL: http://www.nvidia.com/object/cuda_home_new.html.

[17] G. Octave. *About*. URL: https://www.gnu.org/software/octave/about.html.

[18] G. Octave. *Creating Packages*. URL: https://www.gnu.org/software/octave/doc/v4.0.3/Creating-Packages.html.

[19] G. Octave. *Getting Started with Mex-Files*. URL: https://www.gnu.org/software/octave/doc/v4.0.1/Getting-Started-with-Mex_002dFiles.html.

[20] G. Octave. *Packages*. URL: https://octave.sourceforge.io/packages.php.

[21] On trust region methods for unconstrained minimization without derivatives. *M.J.D. Powell*. URL: https://pdfs.semanticscholar.org/0d9f/c7b84c3d738cf2138b0bf9588af899fec547.pdf.

[22] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. *SIP-14 - Futures and Promises*. URL: http://docs.scala-lang.org/sips/completed/futures-promises.html.

[23] T. O. M. Project. *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/.

[24] P. Toint. *Some numerical results using a sparse matrix updating formula in unconstrained optimization*. Math. Comp., 1978.

[25] E. W. Weisstein. *Ackermann Function*. URL: http://mathworld.wolfram.com/AckermannFunction.html.

[26] Wikipedia. *Matlab*. URL: https://pt.wikipedia.org/wiki/MATLAB.

[27] Wikipedia. *Round-robin scheduling*. URL: https://en.wikipedia.org/wiki/Round-robin_scheduling.

[28] A. S. William Gropp Ewing Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014.