



PEDRO MIGUEL DUARTE FEITEIRA

Licenciado em Ciência de Engenharia Informática

EXECUÇÃO A PEDIDO DE TAREFAS EM RECURSOS REMOTOS

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2021



NOVA

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO
DE INFORMÁTICA

EXECUÇÃO A PEDIDO DE TAREFAS EM RECURSOS REMOTOS

PEDRO MIGUEL DUARTE FEITEIRA

Licenciado em Ciência de Engenharia Informática

Orientador: Vítor Manuel Alves Duarte

Professor Auxiliar, Faculdade de Ciências e Tecnologia da UNL

Júri:

Presidente: Artur Miguel de Andrade Vieira Dias
Professor Auxiliar, Faculdade de Ciências e Tecnologia da UNL

Arguente: Carlos Jorge de Sousa Gonçalves
Professor Adjunto, ISEL

Orientador: Vítor Manuel Alves Duarte
Professor Auxiliar, Faculdade de Ciências e Tecnologia da UNL

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2021

Execução a pedido de tarefas em Recursos Remotos

Copyright © Pedro Miguel Duarte Feiteira, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

*Dedico esta dissertação à minha avó, Albertina, ao meu avô,
Manuel e à minha bisavó, Maria.*

AGRADECIMENTOS

Esta dissertação foi, de longe, o projeto mais complexo e exigente que realizei, mas também o mais gratificante. Este permitiu-me a consolidação de tudo o que aprendi ao longo de todo o meu percurso académico. Posto isto, e como não poderia deixar de o fazer, queria realizar alguns agradecimentos.

Começando pela Casa, responsável por me ensinar tudo o que sei até hoje, queria deixar o meu grande agradecimento à FCT NOVA, principalmente ao Departamento de Informática (DI). Esta Casa proporcionou-me um ambiente fantástico que fez com que estes 5 anos passassem a correr.

Ao orientador, professor Vitor Duarte, queria deixar o meu agradecimento pelo desafio proposto e pelos conselhos proporcionados, que permitiram a realização desta dissertação.

Quero deixar um especial obrigado ao meu colega, Sérgio Tavares, por disponibilizar toda a informação importante que usou na sua Dissertação e também pela sua disponibilidade para me ajudar. Ajuda essa crucial para a realização deste projeto.

Nada disto teria sido possível sem o apoio da família e amigos. Começo por agradecer à minha mãe e à minha avó, Maria. Sem este apoio incondicional, nada disto teria sido possível de concretizar. Elas sempre me deram a coragem e a força de vontade para alcançar todos os meus objetivos na vida, onde este não foi exceção.

Agradeço ao meu avô, Manuel Bento e à minha tia que estiveram sempre lá para me apoiar nos bons e nos maus momentos do meu percurso, tentado sempre mostrar-me que tudo tem um lado positivo.

Deixo aqui um enorme agradecimento ao meu primo, Daniel e à Rita Palma, duas pessoas bastante especiais para mim, que contribuíram bastante para o meu equilíbrio. Sem este apoio, alguns desafios teriam sido bastante difíceis de superar.

Para finalizar, quero agradecer a todos os meus amigos, desde as amizades mais antigas até às amizades feitas durante este percurso académico. Das minhas amizades mais antigas não posso escolher ninguém como mais importante, porque todos o foram à sua maneira. Das amizades feitas durante o percurso académico, quero agradecer aos meus

colegas, André Rodrigues, Diogo Pereira, João Antão, João Teixeira, José Duarte, Li Zixiang e Tiago Ventura. Um obrigado por todo o companheirismo e por toda a troca de ideias que me fizeram crescer, não só para me tornar um bom aspirante à profissão que pretendo exercer, como também para evoluir como pessoa.

RESUMO

As áreas de investigação das ciências e engenharias necessitam de uma grande capacidade de processamento, que permita encontrar soluções para diversos problemas de elevada complexidade, como o cálculo de fórmulas complexas e a realização de simulações. Tal capacidade não se consegue obter através de um computador pessoal. Assim, esta classe de problemas foi migrada para novos sistemas de computação como *clusters* e, mais recentemente, para a computação em nuvem. Estes sistemas, para além de oferecerem a computação desejada, requerem que os seus utilizadores possuam conhecimento na área de informática, para efetuar a sua gestão. Posto isto, podemos concluir que estes foram desenvolvidos apenas para otimizar o desempenho de execuções, não oferecendo uma usabilidade simplificada, para serem aproveitados por utilizadores inexperientes (como matemáticos, biólogos, entre outros).

Esta dissertação teve como objetivo a realização de um sistema que permita, tirando partido das propriedades dos recursos remotos, a utilizadores inexperientes usufruirem destes, na realização das suas execuções e não tendo outra preocupação, para além do pedido realizado ao sistema e dos seus resultados. A Solução Proposta oferece uma arquitetura em paralelo, que abstrai os detalhes dos recursos, oferecendo funcionalidades para automatização de reserva e lançamento de computações em recursos remotos com os respetivos dados.

Para a avaliação da Solução Proposta, foram realizadas análises funcionais e de desempenho para averiguar o momento em que a Solução, passa a ser uma mais-valia para o utilizador. No desempenho foi testado o impacto provocado pelas comunicações realizadas aos recursos remotos e a partir de que instante, este impacto passa despercebido. Foram analisadas algumas métricas de desempenho e, por fim, foi utilizado um problema realista para demonstrar o comportamento do sistema desenvolvido.

Palavras-chave: Computação em nuvem, execução a pedido, paralelismo, clusters, desempenho

ABSTRACT

Science and Engineering research areas need a high computation power, used to find solutions for the most diverse problems as complex equations calculations, simulations, etc. This kind of computation power cannot be obtained with a personal computer. Thus, this class of problems was migrated to remote computational resources as clusters and, more recently, for clouds. These systems, besides the high computation power provided, require users to have computer science knowledge, to successfully manage them. Therefore, we can conclude that these systems were developed just for performance optimization, not to offer a simplified usability for inexperienced users (such as mathematicians, biologists, and so on).

So, this dissertation aimed to create a system that allows, by taking advantage of remote resources properties, inexperienced users to use these latter, having no other concern than the request made and, consequently, its result. This Proposed Solution offers a parallel architecture, that abstracts the resources details, offering automation functionalities for reservation and launching computations in remote resources, with their data.

The Proposed Solution was evaluated using functional and performance analysis to get the moment that it can be valuable for the user. For the performance, it was tested the remote resources access impact and when it becomes irrelevant. Performance metrics were analysed too and, lastly, it was used a realistic problem to demonstrate how this system reacts towards real life challenges.

Keywords: Cloud Computing, on demand executions, parallelism, clusters, performance

ÍNDICE

Índice de Figuras	xi
Índice de Tabelas	xiii
1 Introdução	1
1.1 Descrição do Problema	2
1.2 Objetivos	3
1.3 Estrutura do Documento	5
2 Trabalho Relacionado	6
2.1 Otimização sem derivadas (OSD)	6
2.1.1 Métodos	7
2.1.2 SID-PSM(Simplex Derivatives - Pattern Search Method)	8
2.2 Paralelismo	9
2.2.1 Arquiteturas Base	10
2.2.2 Métricas fundamentais do desempenho	11
2.3 Computação em Nuvem	14
2.3.1 Tipos e Modelos	15
2.3.2 Aplicações e Paradigmas	16
2.3.3 Conclusão	21
2.4 Big Data e Computação de Alto Desempenho (CAD)	22
2.4.1 Master-worker	23
2.5 Paralelismo em MATLAB/Octave	24
2.5.1 MATLAB	24
2.5.2 Octave	25
2.5.3 Dragonfly	26
3 Solução Proposta	28
3.1 Descrição Geral	28
3.1.1 Arquiteturas	29

ÍNDICE

3.2 Funcionamento	33
3.2.1 Reserva de recursos e Lançamento dos componentes do sistema	34
3.2.2 Funcionalidade de Execução Genérica	39
3.2.3 Interface de programação Octave	40
3.3 Comparação com as ferramentas existentes	44
4 Avaliação	46
4.1 Hardware Disponível	46
4.2 Cenários de Avaliação	48
4.2.1 Escolha dos cenários a utilizar	49
4.3 Avaliação da Funcionalidade Genérica	52
4.4 Avaliação da Interface de Programação Octave	54
4.4.1 Problemas de Otimização Utilizados	54
4.4.2 Considerações Relevantes	56
4.4.3 Cálculo do peso dos <i>overheads</i> , $C_{overheads}$	57
4.4.4 Avaliação com dois Executores	65
4.4.5 Avaliação do sistema usando o problema STYRENE	66
5 Conclusões	69
5.1 Trabalho futuro	71
Bibliografia	74
Anexos	
I Anexo 1 Exemplo de uma execução da Solução Proposta	79
II Anexo 2 Documentação do código-fonte da Solução Proposta	87
II.1 Cliente	87
II.1.1 Funcionalidades	87
II.1.2 Ficheiros de configuração	90
II.1.3 Exemplos	90
II.1.4 Programas de lançamento dos recursos	90
II.1.5 Componentes Java	92
II.2 Proxy	92
II.3 Executor	93

ÍNDICE DE FIGURAS

2.1	Comparação dos custos necessários, entre servidores tradicionais e a computação <i>serverless</i> , retirado de [21].	17
2.2	<i>Workflow</i> do Clowdr, retirado de [26].	20
2.3	Execução do <i>Map Reduce</i> , retirada de [36].	24
2.4	Passos da execução do DragonFly, retirada de [42].	26
3.1	Arquitetura do <i>hardware</i> , utilizado pelo modelo de execução local.	29
3.2	Arquitetura do <i>software</i> , utilizado pelo modelo de execução local.	30
3.3	Arquitetura <i>hardware</i> e <i>software</i> , utilizada pelo modelo de execução distribuída.	32
3.4	Distribuição de carga, na Proxy	34
3.5	Ficheiro de configuração das máquinas remotas, sem necessidade de reserva.	36
3.6	Ficheiro de configuração da Azure.	37
3.7	Ficheiro de configuração para o <i>Cluster</i> usado.	38
3.8	Comparação entre a utilização da função <i>parfeval</i> , presente na Parallel Toolbox do MATLAB, e a utilização da operação <i>execute(parameters, index)</i> , desenvolvida.	41
3.9	Comparação entre a utilização da função <i>parfor</i> , presente na Parallel Toolbox do MATLAB, e a utilização da operação <i>execute(parameters, index)</i> , desenvolvida.	41
3.10	Comparação entre a utilização da função <i>fetchNext</i> , presente na Parallel Toolbox do MATLAB, e a utilização da operação <i>next</i> , desenvolvida.	41
3.11	Comparação entre a utilização da função <i>cancel</i> , presente na Parallel Toolbox do MATLAB, e a utilização da operação <i>cancel</i> , desenvolvida.	42
3.12	Funcionamento interno da Proxy e do Executor	43
4.1	Código-fonte exemplo programa, desenvolvido em Java.	53
4.2	Resultado do programa, utilizando a máquina pessoal e a funcionalidade genérica.	53

ÍNDICE DE FIGURAS

4.3	Comparação entre sistemas sequenciais com sistemas paralelos, onde o último utiliza sobreposição entre a comunicação e a computação de tarefas.	59
4.4	Representação, em gráfico, da coluna $C_{par/aval}$ da Tabela 4.11.	62
4.5	Tempo total de execução sequencial vs. paralelo, para os problemas de otimização <i>bertsekas</i> e <i>activefaces</i> , utilizando 0,3 segundos/avaliação para $C_{par/aval}$	64
4.6	Tempo total de execução sequencial vs. paralelo, para os problemas de otimização <i>bdqrtic</i> e <i>arwhead</i> , utilizando 0,3 segundos/avaliação para $C_{par/aval}$	64
4.7	Tempo total de execução sequencial vs. paralelo, para o problema de otimização <i>bdvalue</i> , utilizando 0,3 segundos/avaliação para $C_{par/aval}$	65
4.8	Execução do <i>STYRENE</i> na versão sequencial (original) e em paralelo (Solução desenvolvida), variando o número de núcleos.	68
I.1	Utilização das funcionalidades da Solução Proposta.	81
I.2	Preenchimento do ficheiro principal de configurações e compilação da ferramenta.	81
I.3	Pasta <i>remote</i> , onde é guardada a última configuração utilizada pela Solução Proposta.	82
I.4	Preenchimento do ficheiro das propriedades do <i>Cluster</i> utilizado e a inicialização da reserva dos recursos remotos.	82
I.5	<i>Output</i> do programa de reserva dos recursos remotos.	83
I.6	Estado da reserva dos nós e a criação do ficheiro com as informações referentes ao componente Executor da Solução.	84
I.7	Lançamento da Solução, no <i>Cluster</i>	84
I.8	Execução de um exemplo da <i>framework SID-PSM</i>	85
I.9	Relatório final da execução da <i>framework SID-PSM</i> e a libertação dos recursos remotos.	85
I.10	Configuração para utilizar a Microsoft Azure ou máquinas remotas, sem necessidade de reserva, onde se conhecem IPs e portas, disponíveis, para lançar os componentes Proxy e Executor	86
I.11	Antes de se utilizar a Solução, adiciona-se a pasta <i>main</i> ao projeto, com a função <i>addpath</i>	86

ÍNDICE DE TABELAS

2.1	Comparação entre serviços <i>serverless</i> oferecidos pela Amazon, Google e Microsoft, retirado de [23].	18
3.1	Comparações relevantes entre as ferramentas já existentes, em [42], e o sistema desenvolvido, adicionando as novas funcionalidades pensadas.	45
4.1	Visão geral dos cenários de avaliação apresentados.	49
4.2	Média e desvio padrão do tempo, em segundos, de 10 execuções realizadas da framework SID-PSM, para o problema de otimização <i>bertsekas</i> , com 1 unidade de processamento, para cada um dos cenários envolvidos na avaliação da dissertação.	51
4.3	Cenários considerados para a avaliação do sistema.	52
4.4	Problemas de Otimização utilizados para a avaliação, retirado de [7].	55
4.5	Número de avaliações por iteração, da bateria de testes, na versão sequencial da framework SID-PSM.	55
4.6	Valores médios de C_{par} e C_{seq} para a bateria de testes.	58
4.7	Resultados obtidos da execução da bateria de testes do cenário OD e do cenário DDiC com 1 unidade de processamento, realizadas pela framework SID-PSM.	59
4.8	Peso médio dos <i>overheads</i> , $C_{Overheads/aval}$, para cada um dos problemas de otimização constituintes da bateria de testes, calculado através da Equação 4.7.	59
4.9	Variação do peso dos <i>overheads</i> , $C_{Overheads/aval}$, com o aumento das unidades de processamento, p , para o problema de otimização <i>bertsekas</i> , utilizando o cenário DDiC	60
4.10	Peso dos <i>overheads</i> , $C_{Overheads/aval}$, estimado, utilizando o tempo e o número de avaliações, calculados com a Equação 4.7 para o problema de otimização <i>bertsekas</i>	60
4.11	Estimativas para a bateria de testes, para $C_{Overheads/aval}$ e para $C_{par/aval}$, através da utilização das Equações 4.7 e 4.5, respetivamente.	61

ÍNDICE DE TABELAS

4.12 Verificação dos resultados do problema base (<i>bertsekas</i>) e dos dois extremos dos problemas de otimização (<i>activefaces</i> e <i>bdvalue</i>), utilizando os valores estimados da Tabela 4.11.	62
4.13 Métricas de desempenho dos problemas de otimização <i>bertsekas</i> , <i>activefaces</i> , <i>bdvalue</i>	65
4.14 Execução da bateria de testes com dois Executores de 8 unidades de processamento cada e de um Executor de 16 unidades de processamento	66

INTRODUÇÃO

A explosão da ciência da computação, nas últimas décadas, permitiu às ciências naturais e à engenharia adquirirem a oportunidade de melhorar o conhecimento humano, com provas práticas de conceitos empíricos, há muito existentes.

Esta nova visão para o conhecimento trouxe muitas respostas, mas, em simultâneo, novos desafios. Com a evolução das tecnologias, muitos destes problemas adquiriram, cada vez mais complexidade de resolução e, consequentemente, uma necessidade de aumento no poder computacional. Nos seus primórdios, as primeiras aplicações desenvolvidas, devido à sua simplicidade, eram processadas em máquinas pessoais, em tempo razoável. Mas, com o aumento exponencial da complexidade destas aplicações, temporalmente, e, devido ao desenvolvimento da tecnologia, permitiu-se a criação de novas ferramentas de desenvolvimento, tornando o poder computacional das máquinas pessoais, incapaz de satisfazer estas novas necessidades. Posto isto, para a evolução e obtenção de novas respostas foi necessária a criação de novas estruturas capazes de gerar um aumento significativo no poder computacional, para atender estas novas necessidades.

Uma possível forma de aumentar o poder computacional necessário, foi através da agregação de recursos de forma remota, utilizando arquiteturas cliente-servidor ou *peer-to-peer*. Estes promovem a ideia de acesso remoto, de baixo custo, ao armazenamento e ao processador, dos sistemas participantes nesta agregação. Assim, um computador pessoal, perante aplicações desenvolvidas para processamento em massa, por exemplo, tem um baixo poder de computação, mas, se vários computadores pessoais se ligarem entre si, é possível obter, de forma partilhada, uma maior capacidade de armazenamento e/ou de poder de computação. Um sistema foi popularizado em 1999, pelo *Berkeley SETI Research Center*, com o projeto SETI@home [1], onde voluntários disponibilizam os seus computadores para participarem num agregado que executava a deteção de potenciais sinais extraterrestres. O aparecimento de estes sistemas contribuiu para o objetivo principal: aumentar o desempenho no processamento de tarefas complexas. Contudo, levantou também diversos problemas. Por exemplo, os agregados (*clusters*) dedicados são dispendiosos na aquisição e manutenção, exigindo também alguns conhecimentos especializados aos

programadores e utilizadores destes sistemas. Por outro lado, nos participantes voluntários, cada um tem o seu computador, sistema operativo e aplicações, e cada um destes participantes gera a sua própria segurança, o que cria uma falha grave de compatibilidade e vulnerabilidade desta agregação. Além disso, estes sistemas oferecem uma fraca confiabilidade causada pela disponibilidade dos seus participantes, que mudam constantemente e, devido à heterogeneidade do hardware e das arquiteturas de cada participante, torna-se complexa a gestão destes sistemas. Resumindo, a ideia de recursos remotos e o aumento de poder computacional foram alcançados, mas geraram-se novas questões que precisaram de ser resolvidas.

Em 1996, a computação em nuvem apareceu pela primeira vez, mas só em 2006 é que a Amazon [2] a popularizou, lançando o seu produto, *Elastic Compute Cloud*. A NIST¹ define computação em nuvem como "um modelo que permite um acesso ubíquo, conveniente e a pedido, a uma rede provida de recursos partilhados. Estes são configuráveis e podem ser rapidamente requisitados ou libertados, com o mínimo de esforço na sua gestão e interação com o fornecedor de serviços"[4]. Este sistema, comparado com outras opções que oferecem recursos remotos, trouxe muitas melhorias na segurança, na gestão de recursos e nos custos. A segurança e a gestão de recursos foram melhoradas devido à virtualização dos recursos, onde tudo é gerido também virtualmente e com comunicação intranet, ou seja, onde entidades não autorizadas são impedidas de aceder a estes recursos. A última propriedade foi introduzida pelo modelo de negócio *pay-as-you-go*.

1.1 Descrição do Problema

Como referido anteriormente, o uso de *clusters* e a computação em nuvem com todas as suas vantagens, oferecem alguns desafios onde, para esta dissertação, uma das mais importantes é o acesso e uso fácil por utilizadores não especialistas nestas arquiteturas. Porém, estes recursos não são de fácil utilização para qualquer utilizador. É necessário algum conhecimento em informática para os gerir e configurar, tal como com outros recursos remotos como grandes computadores ou *clusters* (normalmente geridos por especialistas que ajudam o trabalho dos utilizadores destas aplicações). Posto isto, considerou-se duas vertentes: possibilidade de construir aplicações que, executando no computador do utilizador, pedem a execução em paralelo de tarefas em recursos remotos; o serviço que atende estes pedidos e gere estes recursos remotos. A primeira vertente pode ser comparada a uma API, que possibilita o utilizador final a, quando necessário, realizar execuções em paralelo utilizando funções simples, onde toda a lógica, comunicações, com os recursos remotos, e complexidade permanecem transparentes. Assim, na visão do utilizador final, este cria uma aplicação, utiliza a API e obtém os seus resultados. Já a segunda vertente é definida como um serviço, lançado nestes recursos remotos, em forma de aplicação (frequentemente uma *web app*), cuja função é receber pedidos de execução para tarefas de

¹National Institute of Standards and Technology, um dos laboratórios de física mais antigos do mundo. Este providencia as medidas e valores *standard* da tecnologia [3].

elevada complexidade e, consequentemente, realizar o seu processamento, sempre que lhe é pedido. Este serviço também consegue gerir os recursos utilizados de forma autónoma e transparente ao utilizador final.

Recentemente, a Amazon lançou a AWS Lambda [5], onde as execuções a pedido receberam um novo serviço, as funções *serverless*. Este trouxe uma nova perspetiva de como um utilizador pode lançar as suas aplicações, na nuvem. Uma das suas grandes vantagens passa por simplificar a gestão dos recursos utilizados para o lançamento de aplicações simples e/ou *scripts*: estes são agora preocupações do próprio fornecedor.

A arquitetura deste serviço promoveu a criação de novos paradigmas que tornaram a utilização da nuvem mais cómoda e barata. Contudo, existem algumas desvantagens. Uma das mais importantes refere-se às próprias funções, desenvolvidas para problemas computacionais simples, pois utilizam um *timeout* para executar qualquer operação requisitada [6]. Também existem limitações no *debugging* das aplicações executadas, isto porque, como a gestão é realizada pelo fornecedor, é impossível monitorizar os passos intermédios de uma execução.

Como exemplo da realidade de muitos destes utilizadores, temos como cenário de exemplo para esta dissertação aplicações para resolver problemas de otimização, em particular para funções sem derivadas (OSD). OSD é um "campo específico de otimização não linear caracterizado pela falta de informação sobre as suas derivadas" [7] (discutido na Secção 2.1). Os testes e avaliação da proposta desta dissertação terão como base um conjunto de problemas de otimização, executados numa variante do framework SID-PSM (**S**Iplex **D**erivatives - **P**attern **S**earch **M**ethod) [8] adaptado ao novo sistema. Os problemas de otimização utilizados, são bastante comuns em áreas industriais e em ambiente académico, necessitando muitas vezes de grandes recursos para produzir melhores soluções em tempo útil. O SID-PSM, como tantas outras aplicações do âmbito científico, é frequentemente utilizado por quem conhece perfeitamente o domínio do problema a otimizar, mas não tem conhecimentos informáticos. É assim difícil (ou impossível) para estes utilizadores tirarem partido das tecnologias e dos serviços que existem hoje em dia para aceder a abundantes e variados recursos computacionais.

1.2 Objetivos

O objetivo principal desta dissertação passa pelo desenho e desenvolvimento de uma plataforma e interface de programação que permita a programação de aplicações e a sua execução por utilizadores inexperientes no uso de recursos remotos, *clusters* e *cloud*.

Quando a aplicação do utilizador necessitar apenas de paralelismo, este pode ser obtido através da sua máquina pessoal. Porém, se existir uma maior necessidade de poder de computação, será permitido ao utilizador escolher o ambiente, que este pretende utilizar. Este ambiente poderá ser composto por computadores remotos de maior capacidade, ou *clusters*, ou de máquinas na nuvem (*cloud*). A reserva desses recursos (quando necessária),

CAPÍTULO 1. INTRODUÇÃO

a configuração, *deployment* e execução deve ser facilitada ou mesmo transparente para o utilizador, ou seja, o sistema encarrega-se da reserva e lançamento dos recursos.

A solução pensada é genérica e deverá ser, futuramente, independente da classe de aplicações, a suportar, e pode ser usada em quaisquer recursos remotos, quaisquer computadores acessíveis por rede, com melhores recursos que o computador do utilizador.

Considerando a nuvem como um dos possíveis recursos remotos que podem ser utilizados, a interface de programação deve ser independente para evitar o *vendor lock-in*, problema que pode dificultar a difusão de aplicações que acedem a recursos obtidos dos fornecedores de serviços de nuvem. Em suma, esta infraestrutura é pensada para gerir quaisquer recursos remotos. Devido ao tempo limitado para desenvolvimento da dissertação, a implementação e avaliação abrange apenas um conjunto limitado de aplicações e classes de problemas, e suporta um subconjunto de recursos remotos específicos.

Para o desenvolvimento desta dissertação, começar-se-à com o estudo da versão original da *framework* SID-PSM. Esta será executada e posteriormente avaliada, obtendo-se assim acesso ao seu *workflow* e comportamento. A seguir, com base nos resultados anteriores será desenhado um sistema, que permite, com as mínimas alterações possíveis à versão original, executar a *framework*, que até agora era sequencial e local (uma avaliação de cada vez, na máquina do utilizador), em paralelo e de forma distribuída, utilizando máquinas remotas. A concretização do ambiente remoto será baseada no paradigma cliente-servidor, visto que esta transmite uma ideia simples de comunicação entre máquinas. Posto isto, terá de haver um componente cliente, que prepara as mensagens da aplicação do utilizador para serem enviadas e processadas pelo(s) servidor(es), aplicando uma estratégia de avaliação remota [9].

Depois do desenho e implementação, será realizada a avaliação do sistema. Para esta, utilizar-se-à a *framework* SID-PSM, através de uma bateria de testes específica. Não só será considerado o problema de otimização a realizar, como também os seus respetivos parâmetros do ambiente de execução, como, por exemplo, o número de *CPU* a usar. Será utilizada sempre a versão original da *framework* como comparação com o que foi desenvolvido, para permitir retirar conclusões face às melhorias e/ou limitações do que foi desenvolvido. Como um dos pontos principais desta dissertação é, para além de permitir usar recursos remotos de maior capacidade, o uso de paralelismo nestes recursos, nesta avaliação serão procuradas as situações em que tal se torna vantajoso e procura-se exprimir em expressões teóricas a previsão de desempenho para cada arquitetura e em função do número de *CPU* usados. Pretende-se que tal seja avaliado e confirmado experimentalmente. Também serão avaliadas as métricas de desempenho, como o *speedup*, eficiência, redundância, entre outros.

Assim, em suma:

- Realizar-se-à a análise da *framework* SID-PSM original, para compreender o seu funcionamento e como se pode criar um sistema que suporte paralelismo e recursos remotos;

- Realizar-se-à o desenho e a implementação de um sistema paralelo e com suporte a recursos remotos e, ainda capaz de gerir automaticamente esses recursos, utilizados por aplicações com uma elevada necessidade de poder computacional. Este deverá oferecer uma usabilidade simples ao utilizador, permitindo assim, o lançamento e obtenção de resultados, de forma eficaz;
- Será Finalmente realizada a avaliação do sistema, através de métricas definidas para a sua arquitetura e em relação ao instante em que esta se mostra uma mais-valia, face à versão sequencial original.

1.3 Estrutura do Documento

Neste Capítulo foi realizada uma explicação de qual é o principal problema que esta dissertação tenta resolver. Também foram introduzidos os desafios e os objetivos da resolução do problema.

No Capítulo 2 é discutido trabalho relacionado e todos os conceitos básicos que serão necessários para entender e resolver o problema.

No Capítulo 3 encontra-se descrita, de forma detalhada, o sistema desenvolvido. Isto é, a descrição da Solução Proposta e de como esta pode ser utilizada de forma simples e prática.

No Capítulo 4, é realizada uma avaliação do sistema, nos aspetos que se acharam relevantes, como o seu desempenho, métricas e usabilidade.

No último capítulo, Capítulo 5, serão apresentadas as conclusões, onde constam as principais contribuições, alguns contratemplos encontrados durante todo o desenvolvimento deste trabalho e também serão referenciados pontos para trabalho futuro.

Nos Anexos I e II, encontra-se, respetivamente, um exemplo de execução com uma aplicação e a documentação do código-fonte implementado.

TRABALHO RELACIONADO

Para alcançar, com sucesso, o principal objetivo é necessário adquirir conhecimento, *a priori*, sobre conceitos teóricos e avaliar algum do trabalho relacionado ao tema em questão. Deste modo, ao longo deste Capítulo, serão apresentados aspectos que contribuem para uma melhor percepção do que já foi conseguido, mas também do que ainda falta conseguir.

Na Secção 2.1, é apresentado o domínio dos *problemas de otimização* que serve de cenário para este trabalho, em particular na *otimização de funções sem derivada* (OSD). Está descrito também o funcionamento da *framework* SID-PSM, um componente que se usou como teste e para a avaliação do sistema desenvolvido. Na Secção 2.2, são introduzidos alguns conceitos sobre paralelismo. Incluindo algumas arquiteturas consideradas relevantes e as métricas tipicamente usadas. Na Secção 2.3 são introduzidos conceitos sobre computação em nuvem. Esta secção é apresentada como uma linha do tempo onde estão expostos, não só os problemas que existiam antes do aparecimento da nuvem, como também os benefícios obtidos pela introdução desta (como paradigmas e arquiteturas). Na Secção 2.4, apresentam-se conceitos sobre *computação de alto desempenho* (CAD) e arquiteturas ao nível de plataforma, nomeadamente a arquitetura *master-worker*. Finalmente, na Secção 2.5, encontram-se soluções que existem atualmente, para melhorar o problema em questão.

2.1 Otimização sem derivadas (OSD)

Os problemas de otimização sem derivadas (OSD), são um campo específico de otimização, não linear, caracterizado pela falta de conhecimento sobre as derivadas de uma função. Quando existe informação sobre as derivadas de uma função, algoritmos para calcular as derivadas são, sem margem para dúvida, mais eficientes que os métodos utilizados em problemas de OSD. No entanto, os problemas reais são complexos, não lineares e não oferecem uma expressão analítica para minimizar uma função, sendo assim necessária uma implementação eficiente, para melhorar esta classe de problemas [7].

Estas derivadas, podem estar indisponíveis para utilização ou podem ser impossíveis de aproximar numericamente (frequentemente, devido à presença de ruído). Como é

possível apurar, métodos comuns de otimização não linear, como o método de Newton¹, não se podem aplicar devido à ausência de derivadas, impossibilitando uma aproximação. Atualmente, esta é uma área de grande necessidade de algoritmos eficientes e robustos, devido ao aumento na complexidade em modelos matemáticos, usados em variadas áreas das ciências e engenharias.

Estes problemas podem ser classificados com base na sua diferenciabilidade. Existem funções suaves, sendo contínuas e potencialmente diferenciáveis, em qualquer ordem do seu domínio. Esta propriedade permite uma aproximação local da função, através de dados disponíveis, utilizando modelos com regiões de confiança. Desta forma, é possível explorar as características de uma função e acelerar a sua convergência num algoritmo de otimização.

Funções contaminadas com ruído ou que carecem das propriedades anteriormente referidas são consideradas não suaves que, de modo geral, são as mais difíceis de otimizar [7].

2.1.1 Métodos

Para a resolução dos problemas de OSD, estão disponíveis várias classes de métodos onde a eficácia de cada uma depende das características da função objetivo e dos seus parâmetros. Nas alternativas, o foco em [7], são métodos com análise de convergência bem estabelecida, através de métodos determinísticos como a procura direta direcional e modelos de regiões de confiança, baseados em interpolação polinomial ou regressão.

2.1.1.1 Procura direta direcional

Os métodos de procura direta direcional, usam apenas valores da função, não aproximando derivadas para modular a função objetivo, tanto de forma explícita como implícita. Neste método, a cada iteração, são comparados os valores da função com um conjunto finito de pontos e, consequentemente, são determinados em que novos pontos esta função deve ser avaliada [10]. Considerando um conjunto de pontos previamente processado, é obtido o seu melhor ponto disponível, x_k . De seguida, obtém-se um novo conjunto de pontos, com base num comprimento definido, na iteração. Consequentemente, compara-se cada um destes pontos com o melhor ponto encontrado até então, x_k . Para o passo seguinte, pode-se utilizar uma, de duas estratégias disponíveis:

- Quando um ponto melhor é encontrado, onde a sua avaliação representa uma diminuição no resultado da função objetivo, a iteração chega ao fim e um novo melhor ponto é definido, x_{k+1} . Está estratégia designa-se por avaliação oportunista;
- Outra estratégia baseia-se na avaliação completa, onde todos os pontos considerados são avaliados e o melhor é escolhido.

¹O método de Newton é frequentemente utilizado na matemática, para estimar as raízes de uma função.

Se nenhum ponto encontrado for melhor que o atual, x_k , o comprimento entre pontos é diminuído e x_{k+1} é definido para x_k . Este processo continua durante várias iterações até o critério de paragem ser alcançado [7].

2.1.1.2 Regiões de Confiança

Os métodos com base em regiões de confiança recorrem a previsões, que funcionam como uma substituição da função objetivo. Para os problemas de otimização em questão, OSD, estes modelos são frequentemente construídos a partir de amostras e de interpolação ou regressão, dependendo do número de pontos disponíveis. Deste modo, o modelo cria uma aproximação local da função, e tenta reproduzir a sua curvatura.

A cada iteração, é considerada a minimização do modelo, numa região à volta do atual x_k . Esta região, pode também ser definida como uma Bola $(x_k; r)$, com centro em x_k e r como o raio da região de confiança [7].

2.1.2 SID-PSM(Simplex Derivatives - Pattern Search Method)

O SID-PSM é um método de pesquisa direta direcional, onde é realizada uma ordenação induzida por gradientes simpléticos e, consequentemente, a avaliação da sondagem produzida. Este método é utilizado para resolver problemas de otimização de funções sem derivada. A análise da sua convergência global, garante que as sequências geradas pelo algoritmo, irão convergir para um ponto estacionário do problema de otimização.

A estrutura deste método é constituída por diferentes passos, descritos de seguida: Inicialização, Procura, Sondagem, Processamento dos Indicadores de Descida, Atualização do tamanho da grelha e Ordenação das Direções [7]. É de se notar que, à exceção da inicialização do método, todos os passos são repetidos, a cada iteração, até se alcançar o critério de paragem definido.

2.1.2.1 Inicialização

Escolhe-se x_0 , $\alpha_0 > 0$ e um conjunto de direções, com boas propriedades geométricas, que abrange conjuntos D positivos. Estes últimos geram qualquer vetor, em \mathbb{R}^n , através de combinações lineares não negativas dos seus elementos. Também são inicializadas outras constantes necessárias, incluindo o tamanho da grelha.

2.1.2.2 Procura

Este passo é baseado na minimização dos modelos de interpolação quadrática ou de regressão, dentro de uma região de confiança, onde estes são construídos a partir de conjuntos avaliados anteriormente. Se o ponto x , correspondente à minimização do modelo, satisfazendo $f(x) < f(x_k)$, então substitui-se $x_{k+1} = x$ e declara-se a iteração como bem sucedida, ignorando o passo de sondagem do algoritmo.

2.1.2.3 Sondagem

A cada iteração que o passo de procura falha, é avaliado um conjunto para sondagem definido como $P_k = \{x_k + \alpha_k d : d \in D_k\}$. Se um determinado ponto $x_k + \alpha_k d_k$ for encontrado, tal que $f(x_k + \alpha_k d_k) < f(x_k)$, a votação termina e declara-se $x_{k+1} = x_k + \alpha_k d_k$ e que a iteração foi bem sucedida. Caso contrário, declara-se que a iteração não foi bem sucedida e $x_{k+1} = x_k$.

Este passo é bastante dispendioso, pois envolve calcular a função para cada ponto que, dependendo da dimensão do domínio e da função poderá demorar até vários segundos ou mesmo minutos por cada avaliação. Sendo estas avaliações independentes, a sua execução em paralelo tem enormes vantagens.

2.1.2.4 Processamento dos Indicadores de Descida

Se não existirem pontos suficientes para a computação, este passo deve ser ignorado. Caso contrário, este passo é usado para computar derivadas simpléticas para obter um indicador de descida de qualidade. Este indicador é usado no passo de ordenação.

2.1.2.5 Atualização do tamanho da grelha

Se uma iteração é considerada mal sucedida, o tamanho da grelha, α_k , é diminuído. Caso contrário, este permanece constante ou é aumentado.

2.1.2.6 Ordenação das Direções

Nesta etapa é selecionado o conjunto positivo $D_k \subseteq D$ para ser usado na próxima iteração (frequentemente contém $2n + 2$ vetores, onde n é a dimensão do problema). Se um indicador de descida está disponível, as direções são ordenadas num conjunto de abrangência positivo, conforme o menor ângulo obtido pelo mesmo. Estas direções serão usadas para processar os pontos da sondagem e a ordenação é utilizada para avaliar, em primeiro lugar, os pontos mais promissores ao valor da função objetivo.

Finalmente, o algoritmo volta ao passo da procura, até o critério de paragem ser alcançado.

2.2 Paralelismo

Um computador de arquitetura paralela pode ser definido simplesmente como um conjunto de processadores que cooperam, em simultâneo, para executar uma computação. Assim, é permitida a execução de aplicações que realizam várias instruções em simultâneo, denominando-se *execuções em paralelo*.

Utilizando este modelo de execução, geralmente em que as instruções são independentes entre si, permite um aumento significativo no desempenho da execução de uma aplicação.

Desde 1945, a rapidez dos processadores tem aumentado exponencialmente. Atualmente, pode-se concluir que este crescimento encontra-se estagnado [11]. Contudo, a necessidade para um elevado poder computacional, continua a aumentar em conjunto com a complexidade lógica das aplicações, tornando importante a utilização de paralelismo, quando possível. Contudo, este trouxe um compromisso importante para a computação, entre desempenho e a utilização de recursos. Nas execuções sequenciais, enquanto o desempenho é baixo (todas as tarefas são executadas uma de cada vez), a utilização dos recursos também é baixa (para esta execução é utilizada apenas uma unidade de processamento). Utilizando paralelismo, o desempenho das computações tem tendência para melhorar, provocado pela divisão das tarefas, pelos diversos processadores. Mas, aumentando este número de processadores, aumenta a utilização dos restantes recursos do sistema (por exemplo, memória, disco e rede). Por outro lado, nem todas as operações são independentes, levando à limitação do grau de paralelismo conseguido. Numa ideologia de paralelismo "perfeito", o equilíbrio seria absoluto entre os recursos utilizados e o desempenho das computações. Como não existe perfeição, no mundo real, o grande desafio é convergir o mais próximo possível, para este ponto de equilíbrio.

2.2.1 Arquiteturas Base

2.2.1.1 Sistemas com partilha de memória

Existem arquiteturas baseadas em vários processadores e/ou processadores com múltiplos núcleos, que partilham a memória. Nesta arquitetura, estes processadores podem aceder a um espaço comum de memória através de um barramento partilhado que permite facilmente, partilhar o estado da computação. Nesta arquitetura, o acesso simultâneo à memória principal partilhada pode provocar congestionamentos no barramento de acesso à memória e, consequente um aumento de latência. Considerando que um processador lê uma instrução, todos os outros processadores têm de esperar até o barramento ficar livre, novamente. Se existirem apenas dois processadores, estes podem funcionar próximos da sua taxa máxima, porque o acesso ao barramento vai alternando entre eles. Mas, se um terceiro processador é adicionado, o desempenho começa a degradar-se. Considera-se que, a partir de 10 núcleos, a melhoria é mínima. Assim sendo, cada um destes contém em cache para reduzir os acessos à memória principal. Esta, para além de reduzir o custo de acessos, necessita de garantir coerência com os valores guardados, na memória partilhada [12]. Tal requer a invalidação de entradas em cache e distribuição de novas versões dos dados em cache para cada processador, sempre que um destes efetua escritas na sua cópia [13].

2.2.1.2 Sistemas de memória distribuída

Um sistema de memória distribuída é caracterizado por processadores individuais, onde cada um tem a sua própria memória. Isto significa que, acessos a dados remotos devem

ser realizados através do envio de mensagens por uma rede de interconexão.

Esta arquitetura apresenta alguns benefícios:

- Evasão na contenção, onde cada processador pode utilizar toda a sua largura de banda para aceder à sua memória local, sem interferir com outros processadores;
- Devido aos processadores constituintes destes sistemas não estarem diretamente ligados, não existe nenhum limite inerente ao número de processadores que se podem utilizar. O tamanho desta arquitetura é condicionado apenas pela rede usada para conectar os processadores;
- Finalmente, como cada processador gera apenas os seus dados, não existem problemas de coerência em cache.

A maior desvantagem, passa pela dificuldade no *design* na comunicação entre processadores desta arquitetura e na gestão de coerência entre cópias dos mesmos dados, em diferentes memórias. Esta abordagem introduz um *overhead* nos sistemas de memória distribuída, devido ao tempo para construir e enviar uma mensagem de um processador para outro e para interpretar as mensagens recebidas de outros processadores. Desta forma, pode ser introduzida uma elevada latência, agravando o desempenho desta arquitetura [13].

2.2.2 Métricas fundamentais do desempenho

Para medir a qualidade da arquitetura paralela ou de uma aplicação, recorre-se a métricas que avaliam não só o seu desempenho, como também algumas das suas propriedades. O *speedup*, a eficiência, o custo, a redundância e a utilização, são algumas das métricas habitualmente usadas. Também convém ter presentes conceitos como a escalabilidade, a lei de Amdahl e a lei de Gustafson como referências dos limites superiores para o *speedup*.

2.2.2.1 Speedup

O *speedup* é usado para analisar a variação no desempenho de um sistema. Este mede o rácio entre os tempos de execução para os dois casos a comparar. Por exemplo, o tempo de executar sequencialmente ou com um único processador, $T(1)$ e em paralelo, usando p processadores, $T(p)$ [14]. Assim sendo, o *speedup*, $S(p)$, é representado como:

$$S = \frac{T(1)}{T(p)}$$

2.2.2.2 Eficiência e custo

A eficiência e o custo são dois conceitos semelhantes [12]. A eficiência, $E(p)$, é usada como medida de capacidade computacional, relacionando o *speedup* $S(p)$ com o número de processadores utilizados, p . Já o custo computacional, $C(p)$, relaciona o tempo de

execução, $T(p)$, com o número de processadores utilizados, p [14]. Assim, a eficiência é definida como:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)}$$

e o custo é definido como:

$$C = T(p) \times p$$

2.2.2.3 Redundância

A redundância, $R(p)$, mede o aumento da necessidade da capacidade computacional, quando se aumenta o número de unidades de processamento [14]. Assim, esta métrica mede o rácio entre o número de avaliações realizadas de uma execução em paralelo, não considerando as comunicações, por p unidades de processamento, $O(p)$, pelo número de operações durante a execução sequencial, $O(1)$:

$$R(p) = \frac{O(p)}{O(1)}$$

2.2.2.4 Utilização

A utilização, $U(p)$, é uma medida do bom uso da capacidade computacional. Mede o produto entre a capacidade computacional usada durante uma execução paralela, com a capacidade que estava disponível, isto é:

$$U(p) = R(p) \times E(p)$$

2.2.2.5 Escalabilidade

Na análise de escalabilidade, é avaliado o comportamento de um algoritmo paralelo quando o número de processadores aumenta e os dados ou trabalho também aumentam. Estes dois parâmetros, definem duas categorias de escalabilidade: *escalabilidade forte* e *escalabilidade fraca*. Na *escalabilidade forte*, a eficiência é medida avaliando o desempenho quando aumenta o número de processadores com um tamanho de dados fixo. A lei de Amdahl é usada como meta para o melhor caso. A *escalabilidade fraca* é medida avaliando o desempenho com um número de processadores e quando se aumenta o tamanho dos dados de entrada. A lei de Gustafson é usada para esta avaliação.

A lei de Amdahl assume que o problema analisado usa um tamanho fixo e o trabalho realizado é independente do número de processadores. Considerando C_{seq} as computações de um programa que não beneficiam de paralelização, C_{par} as computações

que beneficiam e ainda C_{com} as computações relacionadas às comunicações e sincronização da arquitetura, pode-se concluir que o tempo de execução de um programa sequencialmente, $T(1)$ ou T_{seq} , é definido como:

$$T(1) = T_{seq} = C_{seq} + C_{par} \quad (2.1)$$

E que o tempo de execução de um programa em paralelo, $T(p)$ ou $T_{par}(p)$, é definido como:

$$T(p) = C_{seq} + \frac{C_{par}}{p} + C_{com} \quad (2.2)$$

Assim podemos concluir que o *speedup* pode ser calculado como:

$$S(p) = \frac{T(1)}{T(p)} = \frac{C_{seq} + C_{par}}{C_{seq} + \frac{C_{par}}{p} + C_{com}}$$

Sabendo que $C_{com} \geq 0$:

$$S(p) \leq \frac{C_{seq} + C_{par}}{C_{seq} + \frac{C_{par}}{p}}$$

Considerando f a fração da computação que tem de ser realizada sequencialmente vem:

$$f = \frac{C_{seq}}{C_{seq} + C_{par}} \text{ e } S(p) \leq \frac{\frac{C_{seq}}{f}}{C_{seq} + \frac{C_{seq} \times (\frac{1}{f} - 1)}{p}}$$

Que, simplificando, nos dá como limite para o *speedup*:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}}$$

Sabendo que $0 \leq f \leq 1$, é possível prever o *speedup* que se pode alcançar, teoricamente, utilizando múltiplos processadores [12] [14]. Contudo, admitimos um trabalho fixo, ignorando que com mais processadores mais trabalhos podem ser obtidos, mudando a relação entre a parte sequencial e a parte paralela.

A lei de Gustafson surge para amortizar as limitações anteriormente descritas. Esta afirma que, considerando o *speedup* anteriormente definido na lei de Amdahl:

$$S(p) \leq \frac{C_{seq} + C_{par}}{C_{seq} + \frac{C_{par}}{p}}$$

Seja f agora a fração da computação gasta nas computações sequenciais. Assim $(1-f)$ é a fração do tempo gasto na parte em paralelo da computação. Ou seja:

$$f = \frac{C_{seq}}{C_{seq} + \frac{C_{par}}{p}} \text{ e } (1-f) = \frac{\frac{C_{par}}{p}}{C_{seq} + \frac{C_{par}}{p}}$$

Simplificando em ordem a C_{seq} e C_{par} , vem:

$$C_{seq} = f \times (C_{seq} + \frac{C_{par}}{p}) \text{ e } C_{par} = p \times (1-f) \times (C_{seq} + \frac{C_{par}}{p})$$

Substituindo no *speedup* e simplificando:

$$S(p) \leq p + f \times (1-p)$$

Enquanto que na lei de Amdahl começa-se pelo tempo de execução sequencial, para estimar o máximo *speedup* utilizando várias unidades de processamento, a lei de Gustafson realiza o oposto, começa pelo tempo de execução em paralelo [14].

2.3 Computação em Nuvem

Para satisfazer as necessidades de um elevado poder de computação, foram utilizadas novas abordagens para processar a informação. Com o desenvolvimento das redes e de grandes *Clusters*, a ideia de execução de programas, via Internet, mostrou gradualmente que é possível a partilha de recursos e o processamento de informação, de forma eficiente. Este conceito evoluiu ao longo dos anos e permitiu a criação, no início dos anos 90, de um modelo de computação considerado um dos predecessores mais importantes da computação em nuvem, a computação em grelha.

A computação em grelha é caracterizada por agregar vários conjuntos de recursos num sistema heterogéneo distribuído, que se encontra geograficamente disperso por diferentes domínios. Este modelo de computação, para além de uma nova perspetiva na abordagem dos recursos, permitindo a utilizadores de um domínio o acesso a recursos de outros domínios, trouxe problemas sérios.

Como este sistema é heterogéneo, é extremamente difícil o desenvolvimento de aplicações e a sua mobilidade (muito dificilmente uma aplicação, desenvolvida para o recurso A, funcionaria em B sem sofrer alguma alteração). Este modelo, ao ser distribuído por diferentes domínios, cria um grande desafio no que toca à sua segurança e gestão.

Mesmo sendo revolucionário e um sucesso em alguns domínios académicos, não provocou um grande impacto a nível comercial.

Em 2005, apareceram novos conceitos como *utility computing* e computação em nuvem. A *utility computing* oferece *hardware* e *software*, como um serviço, aos utilizadores e usa um novo modelo de negócio: pagar apenas o que é utilizado (*pay-as-you-go*). Assim, o modelo maximiza o uso eficiente dos recursos, associados aos custos, tornando-se a base do conceito da computação a pedido.

Mas, para além das propriedades oferecidas pela *utility computing*, a filosofia de nuvem acabou por trazer atributos adicionais. Um deles foi a possibilidade de oferecer um serviço que consegue adquirir recursos para computação, de forma dinâmica, com suporte para cargas de processamento variável. São normalmente os seus fornecedores as entidades que asseguram a manutenção e a segurança, dos recursos disponibilizados.

As máquinas utilizadas na nuvem são tipicamente homogéneas e a sua concentração de recursos encontra-se num único domínio, ao contrário da computação em grelha [15].

2.3.1 Tipos e Modelos

O termo computação em nuvem apresenta uma vasta gama de opções, que se distinguem a partir de um conjunto de propriedades, como o tamanho, gestão e até o tipo de utilizador:

- A nuvem privada, onde a sua infraestrutura é mantida por uma organização;
- A nuvem de comunidade que, como o nome sugere, é partilhada por diferentes organizações que partilham os mesmo interesses e valores;
- A nuvem pública que é a infraestrutura mais conhecida, é caracterizada por estar disponível para qualquer utilizador, através de um fornecedor;
- E, finalmente, existe a nuvem híbrida onde a sua infraestrutura é uma composição de duas ou mais nuvens, que permanecem únicas, mas que se encontram ligadas.

Para além dos tipos de nuvem, existem também modelos de oferta baseados no nível de como os serviços são fornecidos ao utilizador: *Software como serviço* (SaaS), Plataforma como serviço (PaaS), Infraestrutura como serviço (IaaS) e, recentemente, Função como serviço (FaaS).

O modelo de **Software como serviço** é caracterizado por oferecer ao utilizador aplicações criadas pelo fornecedor, na infraestrutura da nuvem. Do ponto de vista do utilizador, este é o modelo de mais alto nível, de acesso aos recursos da nuvem, visto que é liberado apenas *software*. Um exemplo bastante conhecido deste modelo é o Google Docs [16].

Com o modelo de **Plataforma como serviço**, é possível lançar aplicações criadas pelo utilizador usando linguagens de programação, suportadas pelo fornecedor. Este modelo abstrai o programador dos detalhes da infraestrutura, facilitando o desenvolvimento, mas limita quando necessita de ser personalizada para melhorar ou responder a requisitos específicos da aplicação. A Google App Engine [17] é um bom exemplo deste modelo.

O modelo de **Infraestrutura como serviço**, oferece a capacidade de requisitar os recursos de computação base, da nuvem. Este é o modelo com menos limitações para a utilização de recursos e, desta forma, um utilizador está apto para lançar *software* arbitrário, incluído sistemas operativos e aplicações. Os recursos oferecidos por este modelo suporta escalabilidade dinâmica sendo baseado no modelo de custos da *utility computing*. Considerando que os recursos são virtuais, o *hardware* é partilhado por diversos utilizadores [15].

Para finalizar, o modelo de **Função como serviço** permite criar fragmentos de código, que podem ser executados em resposta a eventos, como um clique numa página Web, por exemplo. Este modelo permite escalar o código de forma simples, permitindo também uma implementação económica de microserviços². Utilizando as Funções como serviço, os programadores melhoram a velocidade do desenvolvimento de aplicações simples. Visto que os servidores e a sua gestão são transparentes, aplicações que exijam alto tráfego de dados ou uma utilização pesada, não são preocupações. Contudo, esta transparência traz um inconveniente para o controlo do sistema [19]. Este modelo usa o paradigma de computação *serverless*, discutido na Secção 2.3.2.1.

2.3.2 Aplicações e Paradigmas

A computação em nuvem oferece uma infraestrutura sempre disponível e elástica. A primeira propriedade sugere que os utilizadores podem livremente preocupar-se primeiro com o design das suas aplicações, sem as limitações sobre onde e como estas serão executadas. Já a segunda propriedade assegura-se que a distribuição da carga de trabalho da aplicação é dinâmica, sem qualquer necessidade da intervenção do utilizador.

A ciência e a engenharia recorrem a estes sistemas porque, nestas áreas, são usadas aplicações que requerem um elevado poder de computação, sem grandes custos associados. Neste caso, sem custos de aquisição e manutenção, para além dos recursos oferecidos serem bastante diversificados entre armazenamento, quantidade e potência dos processadores, entre outros.

A computação em nuvem é baseada no paradigma cliente-servidor. Ou seja, o cliente é executado na máquina do utilizador, enquanto as tarefas são levadas para a nuvem [20]. A comunicação pedido/resposta, oferecida por este paradigma (entre clientes e servidores sem estado³), é habitualmente utilizada pela maioria das aplicações. A utilização de um servidor sem estado, torna uma aplicação mais simples, robusta, escalável e resistente a falhas⁴.

Recentemente, como referido na Secção 2.3.1, um novo modelo veio revolucionar a forma de como se podem utilizar os recursos oferecidos pela nuvem. Esta utilização, para além de simplificada, permitiu ainda uma nova filosofia de desenvolvimento de aplicações, através da computação *serverless*.

²Microserviços usam uma arquitetura que permite desenvolver serviços independentes entre si, mas que podem trabalhar em simultâneo [18].

³Um servidor sem estado, como o nome indica, é um servidor que não guarda nenhuma informação sobre o seu estado e/ou entidades que o utilizem. Isto significa que cada pedido é uma transação independente e, mesmo que um cliente realize mais do que um pedido, este servidor não consegue reconhecê-lo.

⁴Se ocorrer uma falha, este servidor terá apenas de reiniciar, não sendo necessária a recuperação de dados. Isto remove alguma complexidade na sua implementação e *overheads* associados, como a criação e manutenção temporária de *backups*.

2.3.2.1 Computação Serverless

Mesmo com todas as vantagens que surgiram com o aparecimento da computação em nuvem, estas falharam em alguns aspectos. A nuvem aliviou os utilizadores da gestão de uma infraestrutura física, mas deixou recursos virtuais para gerir. Contudo, não é tão trivial para utilizadores que não têm o conhecimento necessário, para a administração de um sistema.

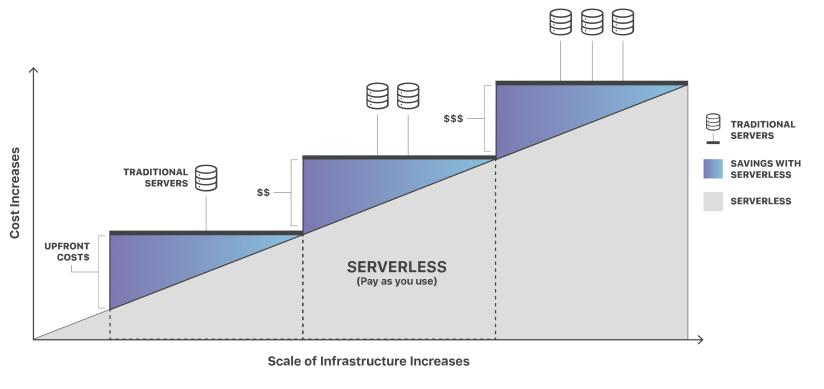


Figura 2.1: Comparação dos custos necessários, entre servidores tradicionais e a computação *serverless*, retirado de [21].

Assim, olhando para todos os problemas, a Amazon criou o serviço *AWS Lambda*. Este oferece a possibilidade de criação de funções na nuvem, popularizando o conceito da computação *serverless*, que sugere apenas a necessidade da escrita do código-fonte, não sendo necessária a preocupação com a administração dos recursos que serão utilizados. Portanto, desenvolvedores perdem a preocupação de reservar capacidade de processamento e as suas aplicações são pagas, com base no número de vezes que o código é acionado. Esta abordagem é importante para aplicações, de uso esporádico, que necessitam de alta disponibilidade [22]. Contudo, esta nova filosofia trouxe também melhorias, em comparação com os serviços tradicionais, como se pode verificar na Figura 2.1, onde à medida que a complexidade da infraestrutura aumenta, os serviços *serverless*, sendo *pay-as-you-go*, promovem uma diminuição nos custos monetários.

Estas funções criadas, respeitam algumas propriedades: Primeiro, cada instância da função executa de forma independente. Isto significa que se ocorrerem vários eventos para a mesma função, estes podem ser tratados em simultâneo, cada um dentro da sua instância. É utilizada uma arquitetura orientada a eventos, através de *triggers*, realizados por outros serviços do fornecedor ou até por programas de terceiros.

Para além da Amazon, outros fornecedores de serviços de nuvem como a Google e a Microsoft, criaram os seus próprios serviços de computação *serverless*, as *Google Cloud Functions* e as *Microsoft Azure Functions*, respectivamente.

Na Tabela 2.1 é apresentada a comparação dos serviços oferecidos por estes três fornecedores. Verificar-se que, por exemplo, a *AWS Lambda* reage a eventos apenas provenientes

dos serviços da própria Amazon e permite a criação/desenvolvimento ilimitado de funções. A *Azure Functions* suporta eventos provenientes de terceiros e o número máximo de funções depende dos recursos disponíveis, no momento. Já a *Google Functions* permite apenas a criação de, no máximo, 20 funções por projeto, suporta apenas Javascript, mas realiza execuções concorrentes de funções de forma ilimitada.

Finalmente, a computação *serverless* promove um alto desempenho e isolamento para a segurança, possibilitando assim a partilha de *hardware*.

Tabela 2.1: Comparação entre serviços *serverless* oferecidos pela Amazon, Google e Microsoft, retirado de [23].

	AWS Lambda	Google Cloud Function	Microsoft Azure Function
Introdução	2015	2016	2016
Escalabilidade	Automática	Automática	Automática
Máximo de funções	Ilimitado	20 por projeto	Depende do evento & recursos disponíveis
Linguagens de programação suportadas	Javascript, Java Python, NodeJS	Javascript	C#, F#, NodeJS, Python, PHP, Bash
Execuções concorrentes	100 execuções por conta	Ilimitadas	Baseado no <i>App service</i>
Modo de <i>deploy</i> para o fornecedor	ZIP	ZIP, armazenamento na nuvem	Git integrado, REST API
Modelo de preço	Pagamento por execução	Pagamento por execução	Pagamento por execução
Serviços que acionam os eventos	S3, SNS, Dynamo DB, Kinesis, Cloud Watch	Cloud Pub, Objetos de armazenamento na nuvem	Azure e serviços de terceiros

Este modelo *serverless* traz uma grande vantagem: a abstração, do ponto de vista do utilizador, da gestão dos recursos remotos. Nesta linha de pensamento, serão apresentados, de seguida, **três sistemas** exemplificativos da Solução pretendida:

- O FnSched, um escalonador desenhado para um serviço *serverless* que optimiza o uso dos recursos, baseados no desempenho requisitado pelo utilizador [24];
- Um sistema de otimização para um problema de regressão (da área de logística) [25], desenvolvido sobre AWS Lambda;
- Clowdr, uma ferramenta para lançamento de experimentos em sistemas de computação de alto desempenho (CAD) e na nuvem, permitindo assim, um acesso partilhado e o lançamento de resultados experimentais [26].

O FnSched [24] foi criado para melhorar um problema específico de agendamento, causado pelo aumento de tráfego na computação *serverless*. Pretende-se que as funções sejam lançadas nos servidores, com o intuito de minimizar os gastos dos fornecedores, enquanto são oferecidas latências aceitáveis. Ao classificar as funções *serverless*, em diferentes categorias: *Edge Triggered*, que se refere a aplicações temporárias e/ou baseadas em eventos e *Massive Parallel*, que utiliza recursos de forma intensa e são perfeitamente paralelos. O FnSched, conforme o estado, no momento, determina:

- Em que máquina deve ser atribuída a próxima função, com base na regulação dinâmica da partilha do processador, em tempo de execução;
- A decisão de quantas máquinas, e respetivas instâncias, devem ser lançadas.

Este escalonador insere e remove o número de máquinas, de forma elástica, baseando-se na carga de trabalho. Esta propriedade usa um algoritmo ganancioso que despacha pedidos, utilizando sempre o menor número possível de máquinas. Se, eventualmente, algumas destas máquinas deixam de ser necessárias, estas são desligadas por inatividade.

O sistema de otimização para um problema de regressão foi criado para explorar recursos que permitam a execução, em paralelo, de aplicações que exigem uma elevada capacidade de processamento, usando funções *serverless*. Neste caso, é implementado um algoritmo de otimização para resolver um problema de regressão, na área da logística. Este problema é composto por uma função de perda, frequentemente utilizada em aprendizagem automática. Esta decompõe-se em duas partes: a soma de N funções suaves, uma função não suave e um vetor de decisão de d dimensões. Com o auxílio do método das direções alternadas para multiplicadores (ADMM)⁵ torna-se possível o uso do modelo *master-worker* (Secção 2.4.1) como base, para a arquitetura deste sistema.

Todos os recursos utilizados na construção deste modelo pertencem à AWS Lambda. Devido a limitações deste serviço, foi criada uma rede, em forma de estrela e atribuído um escalonador (um servidor local) como nó central, onde todos os nós restantes estão ligados a este, via ponto-a-ponto. São utilizadas bibliotecas como:

- ZMQ, para controlar as entradas e saídas, da rede, dos *workers* de forma dinâmica;
- cereal, para serializar e desserializar os dados;
- cURL, para lançar as funções na AWS Lambda.

O escalonador é responsável por lançar e gerir os *masters* e os *workers* da rede. Para problemas de tamanho fixo, o escalonador lança W *workers* e uma *master thread* local. Esta última é processada pelo próprio escalonador. Cada *master* mantém o conhecimento de um vetor de decisão comum e é responsável pela parte da função de perda não suave. A função dos *workers* é obter os resultados da parte suave, da função de perda. Estes, podem interagir com os seus *masters*.

As análises deste sistema, através da execução da função de perda com quantidades de *workers* diferentes, permitiu a obtenção do tempo em que um *worker* está inativo ou em funcionamento. Foi concluído que os problemas de otimização ficam com alguma limitação, devido à computação *serverless* apresentar tempos máximos definidos (*timeouts*), para cada computação, e de não ter estado [25].

⁵O método das direções alternadas para multiplicadores (ADMM) é um algoritmo que resolve problemas convexos de otimização, através da fragmentação destes problemas em pedaços menores, que se tornam de execução mais simples [27].

O **Clowdr** é uma *framework*, escrita em Python, para aglomerar todas as ferramentas computacionais já existentes da neurociência, facilitando assim a sua utilização pelos investigadores. Esta realiza computações na máquina pessoal, em *Clusters*, e na nuvem, onde o utilizador tem apenas de definir qual quer utilizar, através de um simples argumento, na linha de comandos.

A Figura 2.2 mostra o *workflow* do Clowdr. Este utiliza a *framework* Boutiques [28], utilizada para realizar a descrição completa do que se vai executar, garantindo que tudo é definido antes de ser processado. Depois desta descrição, o investigador precisa apenas de realizar o pedido de execução dos seus experimentos no Clowdr, através da escolha dos recursos.

Esta ferramenta ainda oferece um portal onde é possível visualizar, explorar execuções realizadas e filtrar resultados, através de parâmetros definidos. Os experimentos podem ser executados novamente quando, e se, desejado.

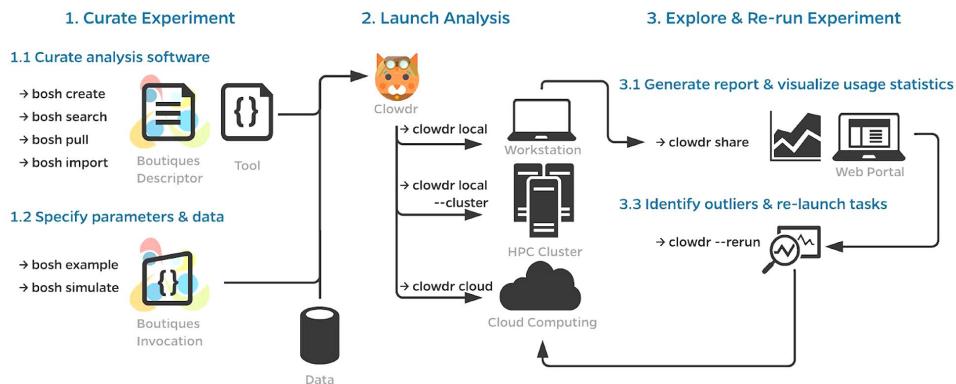


Figura 2.2: *Workflow* do Clowdr, retirado de [26].

2.3.2.2 Vantagens e Desvantagens

Atualmente, a computação *serverless* diferencia-se dos seus predecessores através de propriedades bastante importantes: apresenta isolamento forte, flexibilidade na plataforma e automatização na gestão dos recursos.

Com este novo paradigma, onde o lançamento de aplicações se tornou mais simples, novos utilizadores podem lançar funções sem necessitarem de gerir a infraestrutura ou de grandes conhecimentos da nuvem. Utilizadores mais experientes podem até migrar sistemas já existentes para este novo paradigma e, em muitos casos, diminuir os gastos. Em [22], concluiu-se que migrar uma aplicação, neste caso a MindMup [29], para os serviços *Lambda* reduziu entre 66% e 95% os custos de hospedagem, mas mudou o *design* e grande parte da implementação deste sistema.

Em [30], para comparar o desempenho entre a *Google compute engine* e a *Google cloud function*, foi utilizado um algoritmo implementado através de programação dinâmica, para computar semelhanças entre duas sequências de proteínas. Os resultados mostraram

que, para a mesma configuração, a *Google cloud function* era, neste exemplo, mais barata que a *Google compute engine* (a *Google cloud function* pode custar entre 14% a 40% menos do que a *Google compute engine*), mas é a *Google compute engine* que alcança o máximo desempenho.

Em [31] é mostrada uma comparação geral entre sistemas, adaptados da abordagem tradicional da nuvem, para a abordagem *serverless*. Com base nos seus resultados verifica-se que, para além dos desafios apresentados na Secção anterior, existem outros que necessitam de ser abordados:

- O fraco desempenho para algumas operações comuns de comunicação como disseminação e/ou agregação de pacotes. Isto ocorre por a função ser lançada dentro de uma máquina virtual (MV) aleatória, que está disponível no momento e, considerando que é impossível ter conhecimento se as outras funções pertencem à mesma instância, a comunicação aumenta da complexidade $O(N)$, para $O(N \times K)$, onde N é o número de instâncias no sistema e K o número de funções por MV.
- O desempenho imprevisível, devido ao tempo que pode ser necessário para inicializar uma nova instância. Ou seja, a inicialização de cada instância é composta por três fatores: o tempo que demora a começar a função; o tempo que demora a inicializar o ambiente de execução; e o lançamento do código da aplicação do utilizador. Inicializar uma função pode demorar menos de um segundo, mas, inicializar o ambiente pode demorar vários segundos. Considerando que a computação *serverless* usa um tempo limite para a completar cada execução, a função, pode nunca ser completada, bloqueando processos que poderão estar dependentes deste.

2.3.3 Conclusão

A computação em nuvem trouxe, geralmente, diversas vantagens extraordinárias que auxiliou a criação de uma nova perspetiva para desenvolver uma aplicação:

- Está numa boa posição para incentivar a exploração dos avanços recentes das tecnologias, em *software*, redes, armazenamento e processadores;
- A segurança, a gestão de recursos e a qualidade de serviço são menos desafiantes de manter, comparando com os seus sistemas predecessores;
- É focada para computações em empresas. A sua adoção em indústrias, institutos financeiros, organizações de saúde, contribuiu para um grande impacto na economia;
- Oferece a ilusão de recursos infinitos;
- Elimina a necessidade de um primeiro compromisso financeiro sendo baseada no modelo de negócio *pay-as-you-go*, que pode atrair novas aplicações e novos utilizadores.

Porém, para além dos pontos discutidos anteriormente, este serviço continua a mostrar alguns desafios para superar:

- O *Vendor lock-in*, ou seja, quando um utilizador usa um fornecedor, existe uma elevada dificuldade de mudar para outro.
- Confidencialidade dos dados.
- A imprevisibilidade do desempenho. Esta é uma das consequências da partilha de recursos.
- Elasticidade, a habilidade para aumentar ou diminuir a escala rápida das aplicações. São necessários novos algoritmos para controlar a alocação dos recursos e a carga de trabalho;
- Necessidade de conhecimentos específicos para usar e programar as aplicações para estes ambientes. Este é um dos problemas que se pretende superar na realização desta dissertação.

2.4 Big Data e Computação de Alto Desempenho (CAD)

A *big data* refere-se ao processamento de dados que, dado o seu volume, são tipicamente muito difíceis de armazenar, gerir e analisar, usando sistemas de bases de dados tradicionais. Com alguma frequência, estes sistemas processam dados em tempo real, podendo escalar para suportar grandes débitos de informação [32]. O conceito *big data* surgiu com o aumento exponencial da produção diária de dados, pelas organizações e pelo surgimento da “Internet das coisas⁶”. Existem tipicamente entidades produtoras e entidades consumidoras de dados. Frequentemente, a produção de dados é realizada por aplicações *online* com imensos utilizadores ou por simulações de alto desempenho, ou seja, que já usam uma infraestrutura de CAD. Por outro lado, estes dados são consumidos e analisados através de uma infraestrutura de *big data*, que necessita de uma grande capacidade de IO e/ou capacidade computacional. Assim, pode-se concluir que os conceitos *big data* e computação de alto desempenho, estão relacionados [34].

De forma trivial, computação de alto desempenho (CAD) pode ser definida como a agregação de recursos de computação, que visam oferecer soluções rápidas para problemas complexos, que um computador pessoal não consegue resolver, em tempo útil. Cada máquina de um sistema de CAD é conhecida como um nó. Este é composto por múltiplos processadores, memória, armazenamento e unidades de processamento gráfico (GPU). Todos os componentes de cada nó são conectados internamente através de uma rede privada ao *Cluster*. Esta abordagem, quando muitos utilizadores diferentes tendem em utilizá-la, requer um escalonador para gerir os recursos e suportar todos os seus acessos.

⁶A Internet das coisas (IoT) são todos os equipamentos ou objetos que se encontram permanentemente *online* e se conseguem identificar e comunicar [33].

Estes serviços, para além de poderem ser desenhados e implementados pelas organizações que os utilizam, também são disponibilizados, hoje em dia, como um serviço na nuvem. Esta permite tirar partido do processamento em massa de dados, de forma simples, devido às vantagens da computação em nuvem (a inexistência de custos em aquisição e manutenção de máquinas; o modelo de negócio, onde se paga apenas o que se utiliza; a fácil configuração, para uma utilização pré definida). De outro modo, muitas organizações e/ou utilizadores não teriam acesso a este sistema.

Contudo, as implementações existentes na nuvem estão desenhadas para serem aplicadas em casos de uso muito específicos, ou seja, não há liberdade total no que toca a personalização do serviço. Se o que se pretende não for suportado, tem de se desenhar e implementar uma solução própria, que pode não ser trivial.

2.4.1 Master-worker

Na execução de aplicações em paralelo nos sistemas de CAD, recorre-se muitas vezes ao modelo *master-worker*, caracterizado por duas funções: um *master* e vários *workers*, sendo estes últimos distribuídos pelo sistema de computação paralela. O *master*:

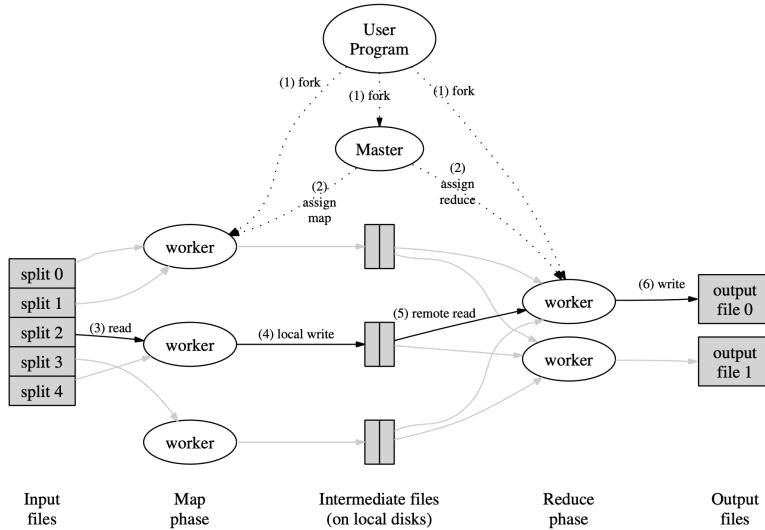
- Divide o trabalho por todos os *workers* e recebe todos os seus resultados;
- Assimila toda a informação recebida, para obter o resultado;
- Mantém os metadados da informação crucial: a informação o particionamento dos dados, quantos *workers* acabaram a execução, entre outros.

A única comunicação que existe, neste padrão, é entre o *master* com cada *worker*, ou seja, os *workers* normalmente não comunicam diretamente entre si [35]. Por isso, usa-se nos casos em que as computações nos *workers* são independentes (não necessitam de trocar informação).

Este modelo é utilizado pela framework *Map Reduce* da Google [36], Figura 2.3. Este é composto por duas funções, executadas em sequência: o *Map* (ou *Mapper*) e o *Reduce* (ou *Reducer*). A primeira obtém o *input* proveniente do disco em tuplos <chave,valor>, processa-o e produz um novo conjunto de tuplos intermédios, <chave_int,valor_int>, como *output*. O *Reduce* usa este conjunto intermédio, e produz o conjunto de tuplos final <chave,valor>, como *output*. Existem duas considerações a realizar, antes do processamento dos *reducers*:

- Qualquer *Reducer* não pode iniciar enquanto existir um *Mapper* em processamento;
- Todos os tuplos que têm a mesma chave são agrupados para o mesmo *Reducer*, que irá agregar os valores para essa chave.

Para além dos passos anteriormente referidos, existe um passo opcional, o *Combine*. Este tem a mesma funcionalidade de um *Reducer* e corre individualmente, em cada *Mapper*. É usado para amortizar a complexidade do processamento dos *Reducers* [37].


 Figura 2.3: Execução do *Map Reduce*, retirada de [36].

2.5 Paralelismo em MATLAB/Octave

Normalmente, os programas pertencentes às ciências e engenharias, são implementados em linguagens de programação comuns nesses domínios e permitem obter um bom desempenho. MATLAB [38] e Octave [39] são exemplos de linguagens de programação bastante utilizadas e que melhor correspondem a estas necessidades. O algoritmo SID-PSM, anteriormente descrito, encontra-se implementado em MATLAB, correndo também grande parte em Octave, daí o foco nestes dois ambientes.

Mas, mesmo utilizando linguagens de programação adequadas, o desempenho pode ser melhorado, quando possível, através de paralelismo. Posto isto, tanto MATLAB como Octave dispõem de extensões à própria linguagem, que permitem execuções em paralelo. Estas são a MATLAB Parallel Computing Toolbox [40] e a Octave Parallel [41], respetivamente.

Para além destas extensões originais às linguagens de programação, existem várias ferramentas criadas pela comunidade, para realizar o mesmo tipo de execuções. Nas inúmeras ferramentas desenvolvidas, destacamos o DragonFly [42].

2.5.1 MATLAB

Para analisar e compreender como se pode criar uma conexão entre a aplicação, que irá ser utilizada como caso de teste [7] e a Solução criada nesta dissertação, é necessária a compreensão da sua linguagem de programação, MATLAB [38]. O MATLAB é usado frequentemente por cientistas e engenheiros para desenvolver algoritmos e/ou simulações. Para além de todas as funcionalidades disponibilizadas, é uma linguagem de programação

closed-source, ou seja, esta carece de uma licença, de custo moderado a elevado, para a sua utilização, que dificulta bastante o acesso a qualquer utilizador e/ou organização.

2.5.1.1 Parallel Computing Toolbox

O MATLAB oferece uma extensão que permite ao programador, quando possível, utilizar paralelismo para melhorar o desempenho da sua aplicação. Esta extensão designa-se por Parallel Computing Toolbox [40]. Esta inclui ciclos *for* que executam em paralelo, tipos especiais de vetores e a possibilidade de lançar *workers* localmente ou utilizando um *Cluster*. Inclui também suporte para a arquitetura *master-worker*, discutida na Secção 2.4.1.

Os ciclos, em paralelo, designam-se por: *parfor* e *parfeval*.

O ciclo *parfor* executa todo o código no corpo do ciclo de forma paralela e independente. Sabendo que cada *worker* executa uma instância de MATLAB em cada núcleo, o número de núcleos da máquina utilizada torna-se a limitação do processamento. Uma execução do *parfor*, é uma iteração. Se existirem mais iterações do que *workers*, estes realizarão mais do que uma iteração do ciclo.

O modelo de execução do *parfeval*, é semelhante ao *parfor*. Contudo, realiza execuções de forma assíncrona, baseada em objetos *Future*. Ou seja, a tarefa é enviada para os *workers* disponíveis e o fluxo do programa não é bloqueado. No *parfeval*, os resultados podem ser obtidos um de cada vez (depois de cada *worker* acabar o seu processamento) [7].

A interface de programação, desenvolvida e apresentada na Secção 3.2.3, tem como base esta extensão.

2.5.2 Octave

Octave é uma linguagem de programação de código aberto, desenvolvida pela GNU, com funcionalidades bastante semelhantes ao MATLAB. Esta pode ser utilizada para desenvolver aplicações, que utilizam funções não nativas do MATLAB e, pode ser considerada uma ótima alternativa a este, visto que não é necessária qualquer licença para a sua utilização. No entanto, apresenta alguma simplicidade na execução de código, pelo terminal do Sistema Operativo, comparativamente ao MATLAB. Estes detalhes estão descritos no Capítulo 5.

Devido a ser uma ferramenta grátil e à facilidade de executar código-fonte de forma direta, o Octave foi a linguagem de programação utilizada para desenvolver o sistema.

2.5.2.1 Octave Parallel

A extensão de execuções em paralelo, oferecida para Octave, não oferece funcionalidades tão ricas como as disponibilizadas em MATLAB. A Octave Parallel, utiliza apenas o envio e receção de informação, através de *sockets*:

- *pconnect/sclose*, utilizados para estabelecer/remover a conexão;

- *psend, recv, eval/rfeval* utilizados para enviar variáveis, receber variáveis e realizar a avaliação de um comando, respetivamente;
- *netcellfun* e *netarrayfun*, que avaliam funções, em *Clusters*, de forma paralela sendo responsáveis por obter os resultados destas avaliações;
- *parcellfun*, que avalia funções numa única máquina, utilizando vários processos em paralelo.

2.5.3 Dragonfly

O DragonFly [42] é uma ferramenta flexível, transparente e robusta, de código aberto, que torna possível a paralelização de código MATLAB e/ou Octave em sistemas computacionais heterogéneos. Este oferece paralelismo de baixo nível, permitindo que os utilizadores escrevam código em paralelo, usando funções MATLAB/Octave, de forma direta. Também conta com paralelismo de alto nível, através da possibilidade de definição de um *Cluster* e, da distribuição de carga, entre os nós.

Suporta também o desenho de aplicações em paralelo, usando diferentes paradigmas: *Single Program Multiple Data*⁷ e *master-worker* (Secção 2.4.1).

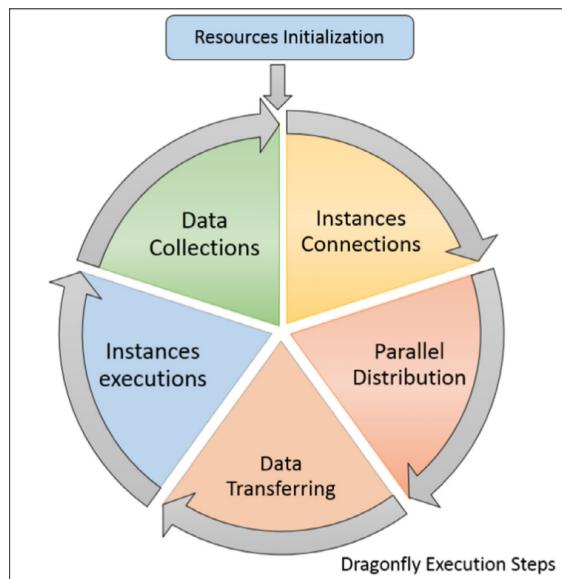


Figura 2.4: Passos da execução do DragonFly, retirada de [42].

O ambiente distribuído do DragonFly:

- Oferece os dados de *input* para qualquer *thread*, incluindo resultados obtidos de execuções anteriores;

⁷*Single Program Multiple Data* significa que cada *worker* pode processar diferentes dados, para aumentar o desempenho da aplicação [7]

- Realiza distribuição dinâmica de carga e o balanço automático, entre os recursos computacionais;
- Obtém o *output* e assegura coerência dos resultados, como a execução sequencial original.

Já na estrutura da sua arquitetura, Figura 2.4, observa-se que:

- Primeiro os utilizadores têm de definir os recursos computacionais. Estes podem ser máquinas com vários processadores, de um ou vários núcleos, *Clusters* ou a nuvem;
- De seguida inicializa estes recursos e configura o ambiente;
- Depois transfere os ficheiros e informações necessárias, para inicializar a execução e, consequentemente, lança as instâncias MATLAB/Octave;
- Finalmente agrupa os dados de *output*, de todos os *workers*, e empacota-os no *master*, com o formato definido pelo utilizador.

Contudo, para além de estas funcionalidades fornecidas, esta ferramenta não está preparada para ser utilizada por utilizadores comuns. Antes de qualquer utilização, o Dragonfly tem de ser reservado e configurado manualmente. A reserva exige algum conhecimento de como o escalonador dos recursos remotos funciona e a configuração exige algum conhecimento de redes de computadores.

SOLUÇÃO PROPOSTA

Neste Capítulo será discutido o sistema desenvolvido. Este visa simplificar e facilitar aos programadores e utilizadores a realização de execuções a pedido, em recursos remotos, como *Clusters* e/ou a nuvem.

Inicialmente, na Secção 3.1, são descritos pontos gerais do desenvolvimento do sistema. São abordados pontos importantes dos seus componentes e de como foi pensada toda a sua arquitetura.

Na Secção 3.2, será explicado o funcionamento da infraestrutura, passo a passo, comecando pela descrição das operações, disponibilizadas por esta, os requisitos necessários para a sua utilização, e execução.

Finalmente, na Secção 3.3, são apresentadas comparações, entre o sistema desenvolvido e as ferramentas existentes para solucionar o problema proposto: como o Dragonfly, a Parallel Toolbox do MATLAB e o Parallel Octave.

É importante constatar que está disponível um exemplo de execução do funcionamento deste sistema e a documentação detalhada de todo o seu código-fonte, nos Anexos I e II, respetivamente.

3.1 Descrição Geral

Para além da facilidade atualmente na obtenção de acesso a recursos remotos¹, é necessário algum conhecimento em informática, para saber gerir e/ou configurar todos eles. Assim, este sistema foi desenvolvido utilizando sempre estratégias que visam simplicidade, desde as funcionalidades até à utilização dos recursos remotos. Este foi programado para inicialmente, sobre a forma de teste, realizar execuções locais, ou seja, na própria máquina do utilizador e, de seguida foi criado um ambiente para execuções distribuídas.

Este último modelo desenvolvido permite ao utilizador usufruir de estas funcionalidades, utilizando máquinas remotas, a que este tenha acesso direto, ou através de sistemas onde é feita a reserva dos recursos, como a nuvem ou nós que pertencem a um *Cluster*.

¹Na nuvem, por exemplo, mesmo sendo necessário pagar para os obter, os recursos remotos foram convergindo, temporalmente, para preços mais baixos e aceitáveis, devido à sua banalização e otimização de custos.

3.1.1 Arquiteturas

Como o sistema foi desenvolvido para suportar execuções locais e distribuídas, existem alguns pontos que diferem entre estes dois modelos de execuções que, consequentemente, deram origem a diferentes versões da solução.

Cada uma das arquiteturas apresentadas, de seguida, servem de referência e serão descritas com base no *hardware* utilizado, *software* desenvolvido e como estes interagem entre si.

3.1.1.1 Arquitetura Local

De modo geral, o modelo de execução local, oferecido pelo sistema desenvolvido, é bastante simples sendo utilizado apenas para testes devido aos seus *overheads* implícitos. Este modelo, utilizado numa máquina local, não contém componentes remotos.

Começando pelo *hardware*, Figura 3.1, nesta arquitetura consideramos uma unidade de processamento, com suporte a vários núcleos, onde cada núcleo pode realizar, de forma independente, uma sequência de avaliações da aplicação do utilizador. Já a memória da máquina é partilhada por todas as computações, para guardar os resultados das avaliações realizadas.

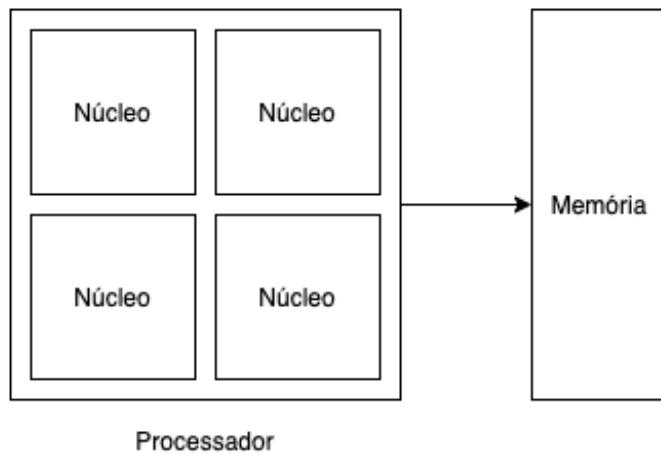


Figura 3.1: Arquitetura do *hardware*, utilizado pelo modelo de execução local.

Olhando agora para a arquitetura do *software*, Figura 3.2, esta é composta por vários componentes. Começando pela **Aplicação**, esta representa a aplicação do utilizador, desenvolvida pelo programador com base na *framework* (uma biblioteca permite a interação desta com o restante sistema), utilizando as funcionalidades implementadas. O **Ficheiro de Configuração** permite a configuração de todo o sistema de forma fácil e simples, auxiliando a transparência para o utilizador sem conhecimentos técnicos. Já o componente

Sistema é composto pelo módulo de processamento responsável por, com base na configuração, gerir a execução das várias computações no processador da máquina local. Este inclui uma estrutura de dados, que guarda os resultados do processamento até serem consumidos pela aplicação. Sabendo que a estrutura de dados utilizada é uma fila de espera, os resultados são consumidos seguindo uma ordem FIFO² à medida que as avaliações vão terminando.

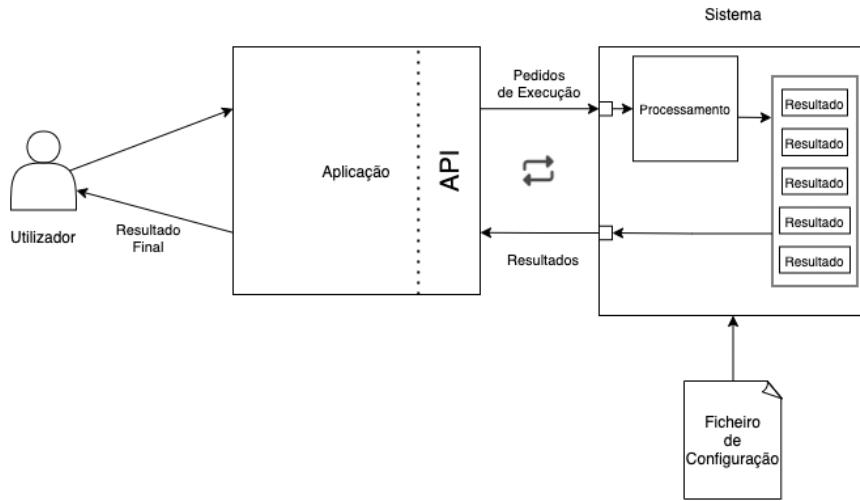


Figura 3.2: Arquitetura do *software*, utilizado pelo modelo de execução local.

As avaliações pedidas pela aplicação podem corresponder à funcionalidade genérica que permite executar outros programas em paralelo com a aplicação (descrito na Secção 3.2.2); ou podem, com base numa interface de programação Octave, executar avaliações de funções em paralelo com a aplicação (descrito na Secção 3.2.3).

Para a utilização do sistema, quando se utiliza a interface de programação Octave, é fundamental o preenchimento do ficheiro de configuração, com as informações requisitadas para a sua execução.

Olhando agora para a execução do sistema, a funcionalidade genérica é direta: Há um programa cliente que permite efetuar o pedido de execução, a aplicação indicada no pedido é executada pelo Sistema (eventualmente em paralelo com outras em curso) e, quando termina, o seu resultado (o *output*) é devolvido ao utilizador. Esta funcionalidade é independente de linguagem de programação, ou seja, qualquer aplicação é suportada desde que a máquina remota tenha suporte à sua linguagem de programação. Já com a interface de programação, a sua execução apresenta alguns pontos que requerem mais detalhe:

²FIFO - First In, First Out, um algoritmo que, como o nome indica, o primeiro elemento a chegar à estrutura de dados é o primeiro a ser servido.

1. Após a configuração da diretoria onde se encontra o código da aplicação, basta ao utilizador lançar a sua aplicação. Toda a distribuição de carga da execução, e pedidos ao Sistema inseridos pelo programador usando as funcionalidades implementadas, são transparentes ao utilizador final;
2. Durante a execução, a aplicação e o sistema desenvolvido estão em constante interação. A aplicação vai enviando pedidos ao sistema; este executa e guarda na memória, da máquina utilizada, os resultados obtidos de cada pedido. À medida que são guardados, vão sendo consumidos pela aplicação;
3. Finalmente, o resultado é mostrado ao utilizador.

Contudo, a funcionalidade genérica foi desenvolvida em C, com bibliotecas *standard*. Já para a interface de programação Octave, foi utilizada uma biblioteca Octave, que permite o *bind* entre o próprio código Octave com outras linguagens de programação como C++, C e Fortran, a *liboctave-dev*.

3.1.1.2 Arquitetura Distribuída

Para proporcionar ao utilizador a possibilidade de usufruir do poder de processamento paralelo em máquinas remotas, foi criado um modelo de execução distribuído. Este modelo é caracterizado através da sua arquitetura, Figura 3.3, onde se podem observar novos componentes, para além da **Aplicação** e do **Ficheiro de Configuração** semelhante ao da arquitetura anterior. Os restantes componentes presentes na Figura: o **Cliente**, a **Proxy** e o **Servidor** ou **Executor** compõem o sistema implementado nesta dissertação e permitem a distribuição dos pedidos de execução pelos restantes computadores.

Começando pelo **Cliente**, este está ligado à biblioteca com a API que permite usar este sistema e é responsável pelo contacto entre a aplicação do utilizador e os componentes remotos do sistema, ou seja, que se encontram nas máquinas remotas e vão processar as avaliações necessárias. Contudo, a comunicação é feita apenas com a **Proxy**. Esta **Proxy** serve de intermediária e gestora de carga entre o **Cliente** e o(s) **Executores(es)**.

Neste sistema existe apenas uma **Proxy**, que gere e interage com todos os **Executores** e atende o **Cliente**. Cada computador terá o seu **Executor** (ou seja, 1:1), responsável por executar e gerir os pedidos que lhe chegam via **Proxy**.

Finalmente temos o **Sistema de Reserva**, responsável pela reserva inicial das máquinas remotas que serão utilizadas para alojar os recursos e instanciar o sistema que permite a execução desta arquitetura. De momento existe a implementação da reserva de recursos para dois cenários: quando é necessária a reserva de máquinas virtuais (MV), na nuvem, e quando se utiliza um sistema de reserva de nós, de um *Cluster*. Sabendo que os componentes remotos do sistema, **Proxy** e **Executor(es)**, serão lançados nas máquinas a que o utilizador tem acesso, estas têm de estar previamente ativas. Por isso, para poupar este procedimento, o sistema também dispõe de um conjunto de comandos que realizam a reserva, a cópia dos componentes para estas máquinas e o lançamento destes componentes,

sem a necessidade da intervenção do utilizador. Esta arquitetura incorpora uma ideologia bastante semelhante à *framework* Clowndr, Figura 2.2. Porém, antes deste lançamento, são necessárias algumas configurações. Esta abordagem encontra-se descrita na Secção 3.2.1.

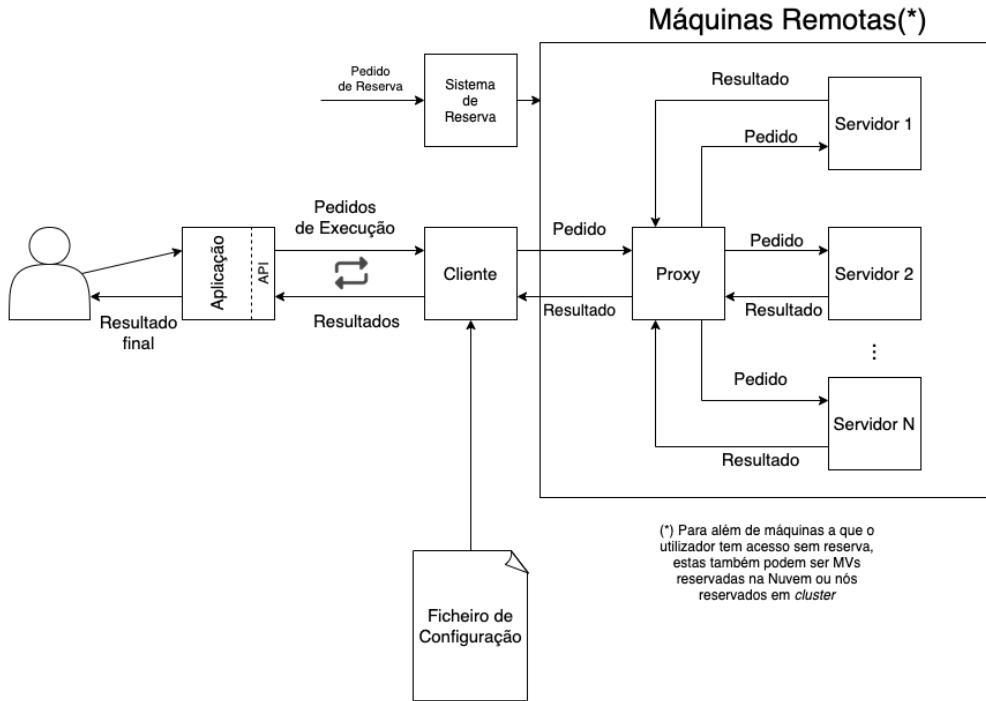


Figura 3.3: Arquitetura *hardware e software*, utilizada pelo modelo de execução distribuída.

No caso da nuvem o utilizador faz a reserva e automaticamente tem acesso à(s) MV(s) alocada(s). Já no segundo cenário, é necessário aceder ao sistema de reservas do *Cluster*, fazer o pedido e esperar até os nós pedidos estejam prontos. No *Cluster* usado, para a reserva é usado o escalonador OAR e depois, para manter a reserva é necessário manter uma sessão SSH³ ligada.

O **Cliente** realiza a comunicação com a **Proxy**, utilizando REST⁴. O primeiro recebe os pedidos, provenientes da aplicação, e efetua depois as respetivas chamadas para a **Proxy**.

Já o **Executor** é responsável pela execução dos pedidos REST, provenientes da **Proxy**. O número de *thread*, que podem ser utilizadas pelo **Executor**, é configurada para ser equivalente ao número de *workers* que o componente disponibiliza devendo ser, tipicamente, igual ao número de *cores* existentes no computador. Este ainda consegue servir múltiplos pedidos em simultâneo e, mesmo se o número de pedidos for maior que o número de

³O SSH é um protocolo de rede utilizado para ter acesso, gerir e/ou modificar remotamente, máquinas ligadas à Internet [43].

⁴Através do protocolo HTTP, *Hypertext Transfer Protocol*, é utilizado na comunicação entre utilizadores, frequentemente através da utilização de navegadores web e servidores [44].

workers disponíveis, estes ficam guardados numa estrutura de dados, executados segundo a ordem FIFO.

Olhando agora para as ferramentas de desenvolvimento, o **Cliente** foi desenvolvido com a linguagem C, utilizadando programação com *sockets*, para criar o cliente REST. A **Proxy** e o **Executor** foram desenvolvidos em Java e utilizam tecnologia REST (biblioteca Jersey), que permitiu a criação destes componentes através de *endpoints* específicos. Para suportar a funcionalidade genérica e a interface de programação Octave, foram utilizadas bibliotecas de execução de comandos, através da criação de processos, diretamente do Java, para as mais diversas linguagens de programação e para a interpretação de funções usando o Octave, respetivamente.

Pode-se ainda observar pela Figura 3.3 que a dinâmica é bastante semelhante ao modelo de execução local, onde a diferença está na utilização de máquinas remotas em vez da própria máquina do utilizador. Posto isto, a execução da funcionalidade genérica mantém-se inalterada, comparativamente ao que foi descrito na Secção anterior. Já o modo de execução da interface de programação Octave, após o preenchimento do **Ficheiro de Configuração** (agora, para além da diretroria da aplicação do utilizador, também é necessário indicar o endereço da **Proxy**):

1. O utilizador lança a aplicação. Toda a fragmentação e distribuição de carga da execução, onde são utilizadas as funcionalidades do sistema, são transparentes ao utilizador final e são agora geridas pela **Proxy**;
2. Para cada avaliação pedida pela aplicação ao **Cliente** (ligado à aplicação), é criado um pedido REST e enviado para a **Proxy**;
3. A **Proxy** redireciona o pedido para um dos **Executores** disponíveis, procurando manter o equilíbrio de carga (utilizando o algoritmo Round Robin, Figura 3.4);
4. Após a avaliação do pedido, o seu resultado retorna à **Proxy**, que o guarda temporariamente;
5. Finalmente, à medida que os pedidos executam, a aplicação vai pedindo os resultados, até o resultado ser obtido. Esta operação é bloqueante, ou seja, se não existirem resultados no momento do *fetch*, o Sistema fica à espera de resposta, para poder prosseguir.

3.2 Funcionamento

Nesta Secção será apresentado o funcionamento do sistema, com base nos modelos de execução discutidos anteriormente. Começar-se-à pela descrição e definição dos conjuntos de comandos utilizados para a reserva de recursos remotos e o posterior lançamento dos componentes do sistema, Secção 3.2.1.

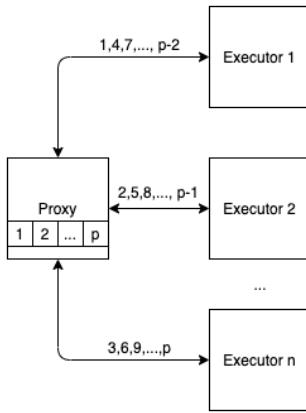


Figura 3.4: Distribuição de carga, na **Proxy**.

De seguida, nos modelos de execução, destacam-se duas variantes de funcionalidades do sistema:

- Uma funcionalidade genérica e extremamente básica, cuja função é permitir ao utilizador usufruir dos recursos remotos. Esta assemelha-se ao lançamento de processos remotos disponível por SSH, sendo descrito na Secção 3.2.2;
- Uma interface de programação para Octave, que facilita a execução em paralelo de funções Octave, descrito na Secção 3.2.3.

Para cada uma das variantes serão descritos os requisitos, as operações fornecidas e um exemplo de uma possível configuração e execução.

3.2.1 Reserva de recursos e Lançamento dos componentes do sistema

Os recursos remotos suportados separam-se em três categorias: máquinas já acessíveis sem necessidade de reservar, máquinas virtuais em Nuvens e nós de um *Cluster*. Estas últimas são reservadas por comandos, desenvolvidos em C, que necessitam de um ficheiro de configurações para realizar todo o processamento, começando pela reserva (do *Cluster* ou de recursos na nuvem). Para simplificar e automatizar a utilização deste sistema como pretendido, estes conjuntos de comandos também realizam a instalação do *software* (os componentes **Proxy** e **Executor**) de forma automática⁵ e a inicialização destes para a execução das aplicações do utilizador.

Nas categorias apresentadas anteriormente, os comandos desenvolvidos para as máquinas remotas são compatíveis apenas para Sistema Operativo Linux/UNIX. Para a utilização de máquinas virtuais em Nuvens os comandos estão, de momento, implementados apenas para a Microsoft Azure. Finalmente, os comandos desenvolvidos para a utilização

⁵Neste momento a instalação ainda não é automática para máquinas Windows.

dos nós de um *Cluster* suportam, de momento, sistemas de reserva OAR [45]. No futuro, é possível a implementação de qualquer outro suporte para os recursos remotos que se pretenda utilizar.

Depois da execução de cada conjunto de comandos de lançamento dos componentes, é gerado pelos comandos um ficheiro designado por *hosts.txt*, que contem a informação necessária para o reconhecimento do(s) **Executor(es)** disponíveis.

3.2.1.1 Máquinas Remotas

No caso de máquinas a que o utilizador tem acesso, via SSH, sem a necessidade de reserva, são oferecidos dois conjuntos de comandos, para a utilização destes recursos. Ou seja, existem os comandos:

- *prepare_hosts* - preparar as máquinas para instalar os componentes do sistema e, consequentemente, realizar o seu lançamento. Este realiza esta preparação, utiliza um ficheiro de configuração, Figura 3.5, que utiliza os seguintes campos:

- A anotação *[Proxy]* e *[Node]*, para definir uma nova máquina, que será utilizada. Ou seja, quando se pretende usar uma máquina para a **Proxy**, antes de qualquer outra configuração, deve ser usada a anotação *[Proxy]*. Só é possível reservar uma máquina para executar a **Proxy**. Para indicar a(s) máquina(s), onde se irá alojar o(s) **Executor(es)**, utiliza-se a anotação *[Node]* e o resto do procedimento é equivalente à anotação anterior. Pode-se utilizar a anotação *[Node]*, para *N*, máquinas, dando origem a *N Executores*;
- O campo *host*, onde o utilizador deve preencher com o endereço da máquina remota, para as conexões SSH;
- *ssh_port*, onde se insere a porta que será utilizada nas comunicações SSH;
- *port*, que corresponde à porta que será utilizada pelo componente;
- *cpus*, dá a hipótese ao utilizador para escolher a quantidade de unidades de processamento que pretende utilizar, da máquina reservada. Por exemplo, se a máquina utilizada oferecer **8 unidades de processamento**, o utilizador pode escolher a quantidade de unidades de processamento que quer utilizar, até ao máximo oferecido. Quando se insere o número **0** neste parâmetro, significa que se utilizará todas as unidades de processamento disponibilizadas;
- *os*, é utilizado para definir o Sistema Operativo da máquina que será reservada. Este parâmetro é utilizado para utilizar os comandos do terminal UNIX ou do terminal Windows;
- O campo *has_service*, que define se a máquina reservada já tem instalado o componente remoto, que lhe será destinado. Este campo é responsável pelo envio do sistema implementado para o recurso remoto e é fundamental ser

ativado na primeira execução. Depois tem deve ser desativado para poupar tempo no lançamento do sistema implementado.

- *shutdown_hosts* –Terminar a execução dos componentes remotos da solução, quando estes não são mais necessários.

```
[Proxy]
host=<user>@<ip>
ssh_port=22
port=8080
#to use all the machine cpus insert 0 (zero)
cpus=0
#unix(*), windows
os=unix
hasService=no
[Node]
host=<user>@<ip>
ssh_port=22
port=8081
cpus=0
#unix(*), windows
os=unix
hasService=no
```

Figura 3.5: Ficheiro de configuração das máquinas remotas, sem necessidade de reserva.

3.2.1.2 Nuvem

Atualmente é apenas suportada a reserva de MV na Microsoft Azure. Para ser possível manter a independência relativamente ao Sistema Operativo, que executa a parte cliente da aplicação, os comandos implementados baseiam-se nos comandos Azure CLI, disponibilizados para variados Sistemas Operativos.

Os comandos desenvolvidos permitem três operações:

- *prepare_azure* - Preparar a Azure com os componentes remotos do sistema, a **Proxy** e **Executor(es)**. Este reserva as VMs conforme o descrito no ficheiro de configuração, e instala o sistema, tornando-o pronto a aceitar pedidos das aplicações dos clientes. A Figura 3.6, contém um exemplo de um ficheiro de configurações que deve ser preenchido pelo utilizador. Este contém praticamente os mesmos campos que para o caso das máquinas remotas, referidas na Secção 3.2.1.1, mais os campos:

- *size*, onde o utilizador insere o tamanho da máquina a reservar. Estes tamanhos diferem, não só no armazenamento que ficará disponível, como no número de processadores e capacidade de processamento destes. Mais informações disponíveis em [46];
- *location*, que permite ao utilizar escolher a zona, no mundo, de reserva da máquina.

Este comando, para além reservar máquinas, lança os componentes remotos e gera ficheiros com algumas informações importantes sobre as máquinas, reservadas. Estes ficheiros não devem ser alterados pelo utilizador;

- *shutdown_azure* - Parar os recursos anteriormente reservados, mantendo as suas configurações. Este é utilizado para permitir ao utilizador, quando o sistema deixa de ser necessário, poupar nos custos associados à reserva. O comando, para realizar este procedimento, utiliza os ficheiros gerados na reserva. Para voltar a lançar a mesma configuração destes recursos, deve-se de voltar a usar o comando *prepare_azure* com o mesmo ficheiro de configuração;
- *clean_azure* - Limpar os recursos quando estes não são mais necessários. Este comando, para além de apagar os recursos, também elimina os ficheiros gerados pela reserva.

```
[Proxy]
ssh_port=22
port=8080
size=Standard_DS1_v2
#to use all the machine cpus insert 0 (zero)
cpus=0
#unix(*), windows
os=unix
location=westeurope
[Node]
ssh_port=22
port=8081
size=Standard_DS1_v2
cpus=0
#unix(*), windows
os=unix
location=westeurope
```

Figura 3.6: Ficheiro de configuração da Azure.

3.2.1.3 Cluster

Nesta configuração foi usado um *Cluster* do Departamento de Informática FCT/UNL gerido pelo sistema OAR. Neste *Cluster* não é permitido que os seus utilizadores tenham privilégios para adicionar e/ou remover *software*, das máquinas constituintes, não incluindo suporte para Octave. Mas, como o *Cluster* oferece Docker⁶, foi criada uma imagem para executar o **Executor**, com suporte para Octave.

Devido a ser necessária a reserva dos recursos, manualmente, antes do seu lançamento foram criados dois comandos:

- *Cluster_reservation* - Reserva de recursos remotos;
- *Cluster_launch* - Lançamento dos componentes remotos (**Proxy** e **Executor(es)**).

Como este processo é mais complexo do que os anteriormente descritos, achou-se por bem a realização deste comando, para auxiliar utilizadores que não tenham conhecimento

⁶Docker é uma plataforma que permite desenvolver, transportar e executar aplicações. Esta, através de configurações descritas pelo utilizador, cria um ambiente virtual, designado por contentor, que pode ser usado como uma máquina virtual ou pode diretamente realizar o lançamento de uma aplicação. Tudo isto depende de um ficheiro de designação específica, Dockerfile, como uma "receita" de preparação deste ambiente virtual [47].

nesta área. Ambos utilizam o ficheiro de configuração, Figura 3.7, que deve ser preenchido com as informações necessárias. Este ficheiro, para além da maioria dos campos definidos nos ficheiros de configuração, da Secção 3.2.1.1, apresenta ainda o parâmetro adicional *host_OS*, que serve para o utilizador definir qual o Sistema Operativo utilizado pela máquina remota. Sendo, este *Cluster* composto apenas por máquinas UNIX, não foi averiguado o comportamento dos comandos de reserva e lançamento, para *Clusters* que suportem máquinas Windows.

No caso da reserva de recursos, o conjunto de comandos é interativo e realiza alguns pedidos ao utilizador, dando dicas de como reservar recursos e de como deve proceder para o próximo passo, o lançamento dos componentes. A única maneira de ter conhecimento das máquinas reservadas e tudo o que é avaliado é através da reserva ativa ou interativa. A única condição passa por manter a linha de comandos, utilizada para a reserva, aberta.

Após a reserva, pode-se executar o lançamento dos componentes remotos do sistema. Este comando tem um parâmetro: o nó que será usado como componente **Proxy**. Este é o nó a que a reserva fica conectada.

Devido a não ser possível o acesso direto a nós internos ao *Cluster* a partir do exterior, após o lançamento, o acesso à **Proxy** é realizado através de um túnel SSH⁷. O **Executor** é lançado, executando a imagem Docker, anteriormente referida.

A partir deste ponto, o sistema está pronto a ser utilizado. Para parar de utilizar os recursos remotos, basta terminar a sessão SSH, criada pela reserva das máquinas remotas, com o *Cluster*.

```
remote_type=cluster
host=<user>@<ip>
#default ssh port: 22
ssh_port=12034
port=8080
#unix(*), windows
host_OS=unix
has_service=yes
```

Figura 3.7: Ficheiro de configuração para o *Cluster* usado.

3.2.1.4 Requisitos para os recursos remotos utilizados

Existem também requisitos, comuns a todos os conjuntos de comandos desenvolvidos, que devem constar, tanto na máquina do utilizador como em todas as máquinas remotas.

⁷Túnel SSH é um mecanismo que permite criar uma ligação direta da(s) porta(s) da máquina do cliente para a máquina remota, ou vice-versa. Assim redirecionando a informação para a porta partilhada é possível enviar dados da máquina local para a máquina remota [48].

Todas as máquinas devem oferecer suporte ao protocolo SSH, isto porque, todas as comunicações entre recursos, que transportam as configurações transparentes aos utilizadores (como o envio de ficheiros, alocação dos componentes do sistema e inicialização destes) são feitas a partir deste protocolo.

Adicionalmente, estas máquinas devem suportar, na sua linha de comandos, os comandos *zip* e *unzip*. É através destes que são comprimidos e descomprimidos os componentes remotos, enviados entre máquinas.

3.2.2 Funcionalidade de Execução Genérica

Esta funcionalidade visa a criação de uma sessão entre o utilizador e o sistema, para executar código compilado, em recursos remotos.

Designa-se genérica, pois não está dependente de uma linguagem de programação em específico. Desde que o *host* onde o **Executor** executa, tenha suporte para a linguagem da aplicação do utilizador, a funcionalidade irá permitir a sua avaliação remota. Nesta fase, Java, C e C++ são algumas das linguagens de programação testadas.

3.2.2.1 Operação

Uma execução desta funcionalidade é bastante direta. O utilizador começa por realizar a reserva e o lançamento dos recursos remotos, utilizando os conjuntos de comandos descritos na Secção 3.2.1, e por preencher o ficheiro de configuração obrigatório.

Esta funcionalidade encontra-se depois disponível através de um comando que recebe como argumentos, o comando literal para realizar a execução. Por exemplo, suponhamos que um utilizador irá executar um programa em Java, previamente compilado (com os ficheiros *.class* do projeto), o comando de lançamento de programas, nesta linguagem de programação é:

```
java nome_da_class_principal argumento1 argumento2 ...
```

Por exemplo, para Windows e considerando a execução Java, mencionada anteriormente, o comando para a utilização desta funcionalidade será:

```
.\remoteSupport.exe java nome_da_class_principal argumento1 argumento2 ...
```

A partir deste ponto, entramos na parte interna da funcionalidade onde:

1. O projeto do utilizador é comprimido e enviado para a **Proxy**, através de um pedido REST. Esta última, reencaminha-o para um **Executor** disponível. A sessão entre o utilizador e a **Proxy** permanece ligada;
2. O projeto, após chegar ao **Executor**, é executado com os argumentos indicados. Após a sua execução, o resultado (*standard output*) é enviado imediatamente à **Proxy** e, consequentemente, passado ao comando *remoteSupport*, que o afixa no *standard output* do utilizador.

Esta funcionalidade apresenta algumas limitações. Por exemplo, se a ligação aos recursos remotos for perdida, não é possível obter o resultado da execução. Ou seja, esta não conta com nenhum mecanismo de tolerância a falhas. Para suportar qualquer linguagem de programação adicional, fica ao critério do utilizador a instalação do seu suporte, nos componentes remotos.

Sabendo que esta funcionalidade cria apenas uma sessão com as máquinas remotas, cabe ao próprio utilizador o controlo do paralelismo da sua aplicação, caso o necessite. Ou seja, esta funcionalidade não oferece opções de paralelismo para embutir no código-fonte. Contudo, é possível lançar várias instâncias, independentes, desta funcionalidade.

3.2.3 Interface de programação Octave

A interface de programação implementada oferece três operações ao programador:

- A função *execute(index, varargin)* permite pedir a execução remota de uma função Octave, com os respetivos argumentos. Tipicamente esta será inserida num ciclo *for*, com finalidade de se aproximar ao comportamento, das duas funções de lançamento em paralelo da extensão *Parallel Toolbox* do MATLAB, as funções *parfor* e *parfeval*. Esta função contém:
 - O argumento *index*, é utilizado para identificação do pedido e pode ser, por exemplo, o índice do elemento, num ciclo *for*. Este permite, quando necessário, a ordenação dos resultados obtidos;
 - *varargin*, uma lista que recebe uma referência para uma função da aplicação e um número variado de argumentos e de tipos possivelmente diferentes [49]. Ou seja, cada lista pode conter números, palavras, objetos, entre outros. Está subentendido que a função será executada com os argumentos indicados nesta lista e pela ordem em que aparecem. Como exemplo, assumindo que numa aplicação é necessária a execução da função 'foo' com argumentos *arg1* e *arg2*, na terceira iteração de um ciclo, a função deve ser usada como: *execute(3,@foo,arg1,arg2)*⁸.

Nas Figuras 3.8 e 3.9 é possível verificar, através de um exemplo, como substituir funções da Parallel Toolbox do MATLAB pela funcionalidade desenvolvida.

- A função *next()*, Figura 3.10, assemelha-se à função *fetchNext()*, presente na Parallel Toolbox do MATLAB⁹. Esta função, realiza um pedido à **Proxy**, para esta devolver o próximo tuplo, pronto a ser consumido. Este tuplo contém o resultado duma execução e o seu respetivo índice para ordenação, caso seja necessário.

⁸O carácter '@' em Octave/MATLAB é utilizado para indicar uma função através do seu nome. Assim sendo @foo será a referência de uma função com o nome 'foo' [50].

⁹*fetchNext(F)*, uma função que devolve o próximo objeto FevalFuture não lido, disponível [51].

- A função `cancel()`, Figura 3.11, é uma função especial, utilizada quando o algoritmo não necessita dos resultados dos pedidos ainda em curso. Tal pode ser, por exemplo, porque o algoritmo baseia-se numa série de iterações onde, perante uma resposta, pode decidir avançar para a próxima iteração sem necessitar dos restantes resultados.

MATLAB	Sistema
<pre>for ii = 1:pointsToEval futures(ii) = parfeval(@eval_point, 1, x_mat(:,ii), f_eval); end</pre>	<pre>for ii = 1:pointsToEval execute(ii, @eval_point, x_mat(:,ii), f_eval); end</pre>

Figura 3.8: Comparação entre a utilização da função `parfeval`, presente na Parallel Toolbox do MATLAB, e a utilização da operação `execute(parameters, index)`, desenvolvida.

MATLAB	Sistema
<pre>parfor (ii = 1:pointsToEval, numWorkers) ftemp = f_eval(x_mat(:,ii)); if isnfinite(ftemp) f_vector(ii) = ftemp; mask_f(ii) = 1; end end</pre>	<pre>for ii=1:pointsToEval execute(ii, f_eval, x_mat(:,ii)); end for ii=1:pointsToEval [idx, ftemp] = next(); if isnfinite(ftemp) f_vector(ii) = ftemp; mask_f(ii) = 1; end end</pre>

Figura 3.9: Comparação entre a utilização da função `parfor`, presente na Parallel Toolbox do MATLAB, e a utilização da operação `execute(parameters, index)`, desenvolvida.

MATLAB	Sistema
<pre>for jj = 1:pointsToEval [completedIdx, ftemp] = fetchNext(futures); func_eval = func_eval + 1; func_iter = func_iter + 1; results(completedIdx) = ftemp; mask_res(completedIdx) = 1; while completedIdx == (lastIndex+1)</pre>	<pre>for jj = 1:pointsToEval [completedIdx, ftemp] = next(); func_eval = func_eval + 1; func_iter = func_iter + 1; results(completedIdx) = ftemp; mask_res(completedIdx) = 1; while completedIdx == (lastIndex+1)</pre>

Figura 3.10: Comparação entre a utilização da função `fetchNext`, presente na Parallel Toolbox do MATLAB, e a utilização da operação `next`, desenvolvida.

MATLAB	Sistema
<pre>cancel(futures); futures(1:pointsToEval) = [];</pre>	<pre>cancel();</pre>

Figura 3.11: Comparação entre a utilização da função *cancel*, presente na Parallel Toolbox do MATLAB, e a utilização da operação *cancel*, desenvolvida.

A interface de programação oferece assim funcionalidades que permitem paralelismo, sendo este transparente ao utilizador.

No programa Octave deve adicionar o caminho para a pasta com o código da API do sistema, usando a função *addpath* do Octave. A partir deste momento, a interface de programação está pronta a ser utilizada para a implementação do código da aplicação do utilizador.

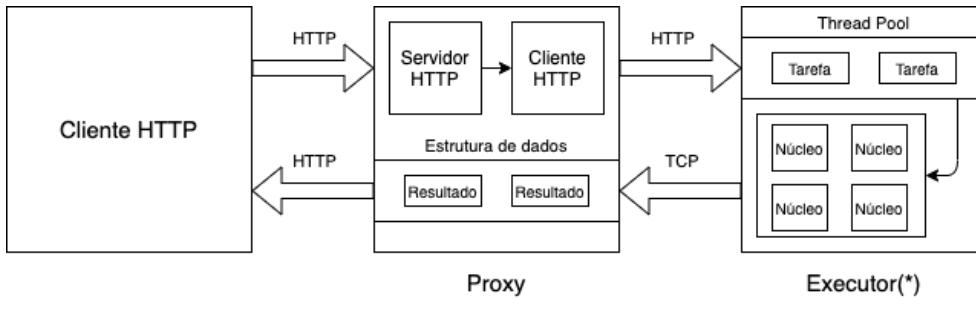
A **Proxy**, para além de receber os pedidos do **Cliente** via REST, redireciona o que foi recebido para uma das instâncias de cada **Executor** disponível, através do algoritmo Round Robin, Figura 3.4. Também conta com uma estrutura de dados, que se vai preenchendo com os resultados, provenientes do **Executor**, através de uma conexão TCP integrada, Figura 3.12.

O **Executor** é composto por uma *Thread Pool*, um serviço que conta com uma fila de espera e uma *pool* de p unidades de processamento (definida pelo utilizador), que vai realizando as execuções recebidas da **Proxy**. A sequência de operações está descrita visualmente na Figura 3.12. Nesta é possível verificar que:

- O pedido (mensagem REST) do utilizador, chega à **Proxy** e é imediatamente reencaminhado para o **Executor**. Após este envio, a **Proxy** avisa o utilizador que o seu pedido foi bem-sucedido;
- Ao chegar ao **Executor**, o pedido é adicionado à sua *Thread Pool* e, pode ter um de dois destinos: ser executado ao chegar, se alguma das unidades de processamento estiver parada ou, se estas estiverem ocupadas, entrar na fila de espera da própria *Thread Pool*;
- Após o término da execução do pedido, o **Executor** envia seu o resultado, através da conexão TCP, para a **Proxy**;
- Por fim, os resultados são guardados na **Proxy**, por ordem de chegada e, consumidos pela aplicação do utilizador.

A interação entre todos os componentes do sistema, é não bloqueante. Isto porque a aplicação não fica à espera do resultado de cada pedido. Este simplesmente envia-o e, recebe uma notificação de sucesso (resposta REST). O mesmo acontece quando a **Proxy** reencaminha o pedido para o **Executor**.

A **Proxy**, posteriormente, recebe, via TCP, os resultados de cada execução e a aplicação usa a função *next()* para verificar quais os resultados que estão prontos, consumindo-os um a um.



(*) Todas as instâncias do Executor são independentes. Assim, para cada uma existe uma conexão HTTP e uma conexão TCP.

Figura 3.12: Funcionamento interno da **Proxy** e do **Executor**.

3.2.3.1 Configurações para utilização do sistema

Para a execução de uma aplicação usando determinados recursos, é necessário um mínimo de configuração, via um ficheiro de configuração. Este ficheiro necessita do tipo de execução pretendido (local ou remota), da diretoria do projeto que utiliza a interface de programação e o *runtime* pretendido (de momento só é suportado Octave). Para as execuções usando recursos remotos também é necessário configurar o endereço e porta do servidor que irá conter a **Proxy**. No caso do *Cluster*, quando a aplicação do utilizador executa no seu computador e acede aos nós do *Cluster* por um túnel SSH, o endereço será *localhost* (127.0.0.1).

Para executar um programa, independentemente de se pretender executar localmente ou usando recursos remotos basta usar o comando habitual para executar programas Octave:

octave nome_do_ficheiro.m

3.2.3.2 Execução Local

Do lado interno do sistema, ou seja, transparente ao utilizador:

1. Para cada pedido efetuado via a função *execute*, a infraestrutura, transforma os argumentos recebidos, numa avaliação de uma função Octave, a ser executada num novo processo. Este é lançado através de uma *thread*¹⁰, em *background*. Após realizada a execução, o seu resultado é guardado numa estrutura de dados;
2. À medida que as execuções vão terminando, os resultados vão sendo armazenados numa estrutura de dados da API;

¹⁰É utilizada a biblioteca *pthread*, uma biblioteca da linguagem C.

3. Quando chamada a função *next*, o resultado à cabeça da fila é enviado para a aplicação. Caso ainda não existam resultados, o *next* espera, bloqueando o sistema. Esta Solução permite controlar possíveis resultados perdidos.

Atualmente esta versão apresenta uma limitação: não usa memória partilhada entre as operações da interface de programação, Secção 3.2.3, e os processos que executam as funções.

Para contornar esta limitação, recorreu-se a armazenar os resultados dentro de um ficheiro temporário. Porém, a cada avaliação realizada, há uma escrita neste ficheiro que, assumindo que poderão ser realizadas milhares de avaliações, serão também feitas milhares de escritas no mesmo ficheiro, o que limita bastante o desempenho do sistema.

3.2.3.3 Execução Remota

Poderá ocorrer um primeiro passo: se for a primeira execução da aplicação, o seu código é enviado para o(s) **Executor(es)** configurado(s). Depois, o Sistema informa a **Proxy** dos endereços de todos os **Executores** disponíveis, assim como de quantas unidades de processamento serão usadas para realizar as execuções, em cada máquina. Já existindo o código do projeto no(s) **Executor(es)** e as respetivas configurações, este passo é ignorado.

De seguida, dá-se o funcionamento interno do sistema, como descrito na Secção 3.2.3.

3.3 Comparação com as ferramentas existentes

A Tabela 3.1 mostra a comparação entre as principais ferramentas apresentadas no Capítulo 2 com o sistema desenvolvido. Como é possível observar, esta solução, tal como o Dragonfly, apresentam algumas vantagens relativamente às restantes ferramentas, para os objetivos pretendidos. Isto acontece porque ambas as ferramentas permitem criar aplicações que vão usufruir de execuções em paralelo, no entanto, do ponto de vista do programador e, principalmente, do utilizador, nem sempre é fácil a instalação da infraestrutura e do sistema que suporta a execução da aplicação. Por exemplo, pode-se observar que o MATLAB Parallel Toolbox, o Parallel Octave e mesmo o Dragonfly, assumem que os recursos estão sempre disponíveis e algum especialista os instala e configura a ferramenta para uso do utilizador. Outro aspeto a ter em conta é que estes sistemas parecem estar vocacionados apenas para programas em MATLAB ou Octave, quando seria bastante vantajoso se estas plataformas suportassem outras linguagens. Também é possível observar que, devido a ser necessária uma licença diferente para utilizar o MATLAB em modo distribuído, acarretam-se custos acrescidos. Outro ponto, negativo de todas as ferramentas já existentes, passa pela não automatização de preparação dos recursos: por exemplo, no Dragonfly as máquinas que irão hospedar estas instâncias têm de estar devidamente configuradas, *a priori*. Estas configurações são à base de instalações de *software*, via linha

3.3. COMPARAÇÃO COM AS FERRAMENTAS EXISTENTES

de comandos, que não é trivial, para utilizadores inexperientes e não sendo fácil estes utilizadores alterarem depois, as configurações ou a terem várias para diferentes conjuntos de recursos.

O sistema desenvolvido focou-se, como previsto nos objetivos, em facilitar a reserva e configuração automática de recursos remotos onde possa executar a pedido e em paralelo programas ou tarefas de aplicações que o utilizador corre a partir do seu próprio computador. Assim temos:

- Instalação automática dos requisitos, para executar a infraestrutura. Como referido na Secção 3.2.1. Esta funcionalidade está, no presente, a funcionar apenas em ambientes UNIX;
- Suporte para mais linguagens de programação. Será uma ferramenta genérica, onde a interface de programação Octave, já é oferecida pelo sistema, mas serão disponibilizadas também em outras linguagens de programação. Contudo, até ao presente, a funcionalidade genérica, que permite a execução remota de programas para várias linguagens de programação, é a única que cumpre este requisito;

Para além destas novas funcionalidades, a usabilidade do sistema junta as principais funcionalidades suportadas pelo Dragonfly, mas com a simplicidade de programação das funções inspiradas pela Parallel Toolbox, do MATLAB e muita mais fácil configuração pelo utilizador, para variados recursos.

Tabela 3.1: Comparações relevantes entre as ferramentas já existentes, em [42], e o sistema desenvolvido, adicionando as novas funcionalidades pensadas.

Funcionalidade	MATLAB (Parallel Toolbox)	Parallel Octave	Dragonfly	Sistema desenvolvido
Tipo de licença	Closed Source	Open Source	Open Source	Open Source
Possibilidade para paralelizar as tarefas nos núcleos de uma máquina individual	Sim	Sim	Sim	Sim mas com limitações (descritas na Secção 3.2.3.2)
Recursos remotos suportados	Clusters, Cloud (MATLAB Parallel Server)	Clusters	Máquinas individuais, Clusters, nuvem	Máquinas individuais, Clusters, nuvem (Azure mas como trabalho futuro AWS e Google Cloud)
Reserva dos recursos e instalação do sistema de forma autónoma	Não	Não	Não	Sim (em ambientes UNIX)
Facilidade na definição do ambiente de execução	Não (configuração manual necessária)	Não	Não (configuração manual necessária)	Sim
Capacidade para realizar computações entre diferentes sistemas operativos	Sim, utilizando licenças diferentes	Sim	Sim	Sim
Interface gráfica para definir o ambiente computacional	Sim	Não	Sim	Trabalho futuro
Interoperabilidade entre linguagens (MATLAB/Octave)	Não	Não	Sim	Trabalho futuro
Linguagens de Programação suportadas	MATLAB	Octave	MATLAB/Octave	Octave (Java, C, C++, etc., de forma limitada)

AVALIAÇÃO

Para uma validação dos resultados esperados, o sistema foi submetido a uma avaliação que permitiu obter respostas, quer sobre o seu correto funcionamento, quer sobre o seu comportamento e desempenho em vários cenários de utilização.

Na Secção 4.1 é apresentado o *hardware* disponível para a avaliação da dissertação. Na Secção 4.2 estão descritos todos os potenciais cenários, e a escolha dos realmente usados na avaliação do sistema desenvolvido. Nas Secções 4.3 e 4.4, estão descritos os passos de avaliação para a Funcionalidade Genérica (Secção 3.2.2) e para a Interface de Programação Octave (Secção 3.2.3) do sistema proposto, respetivamente.

Na avaliação da funcionalidade genérica é apenas testada a sua utilização e avaliado o *overhead* da comunicação, face à execução local. A avaliação da interface de programação é baseada no seu desempenho onde, num primeiro contacto, o sistema é comparado com a sua versão sequencial. De seguida, para avaliar o seu desempenho, experimentou-se dividir a carga por diferentes núcleos de um processador e/ou por diferentes computadores e verificar, até que ponto, a comunicação com os recursos remotos é um fator limitante, para este desempenho. Complementarmente, são apresentadas expressões algébricas que permitem prever o comportamento do desempenho do sistema, confirmadas posteriormente pelas experiências efectuadas. Termina-se com a avaliação de um problema próximo da realidade.

4.1 Hardware Disponível

Para cobrir os diferentes infraestruturas consideradas nos nossos objetivos, usou-se uma infraestrutura exemplificativa de cada tipo. Inicialmente, para realizar os primeiros testes, foi utilizada a máquina pessoal (PC - *personal computer*), que apresenta a seguinte configuração:

- CPU Intel Core i5-6600K, com 4 núcleos, 4 *threads* e 3,5 GHz de frequência [52];
- 16 GiB de memória RAM DDR4;
- Sistema operativo Windows 10;

- SSD de 500 GB e HDD de 1TB;

Foi usada uma máquina acessível remotamente, onde não era necessária a sua reserva para utilização. Esta máquina, que será referida como **AMD/64** ao longo deste documento, foi utilizada na Secção 4.4.5, para simular uma utilização real, tanto da *framework* SID-PSM como do sistema desenvolvido, e apresenta a seguinte configuração:

- CPU AMD EPYC, com 64 núcleos;
- 64 GiB de memória RAM DDR4;
- Sistema operativo Linux/Ubuntu;

Como exemplo de utilização de recursos na nuvem, procedeu-se à utilização das MV da Microsoft Azure, onde foi utilizada apenas a configuração mais básica devido à existência de limite nos créditos para a utilização deste serviço da nuvem. Assim, as MV da nuvem utilizadas, apresentam a seguinte configuração:

- CPU virtual;
- 3.5 GiB de memória RAM DDR4;
- Sistema operativo Linux/Ubuntu;
- SSD de 7 GB.

Por último, recorreu-se ao *Cluster* do DI [53], gerido pelo sistema OAR [45]. Devido à grande diversidade de *hardware* por nó que este oferece, foram selecionados apenas **três nós** específicos. Esta escolha deveu-se ao simples facto de estas máquinas apresentarem a mesma configuração de *hardware*, permitindo assim obter resultados comparáveis. Assim, cada máquina utilizada, apresenta a seguinte configuração:

- Dois processadores Intel Xeon E5-2609 v4, cada um com 8 núcleos, 8 *threads* e 1,70 GHz de frequência [54].
- 32 GiB de memória RAM DDR4;
- Sistema operativo Linux/Debian 10;
- SSD de 110 GB.

4.2 Cenários de Avaliação

Considerando os modelos de arquitetura *hardware*, descritas no Capítulo 3, consideraram-se possíveis cenários de utilização para avaliar o comportamento do sistema, quando se utiliza a funcionalidade genérica ou a interface de programação Octave na máquina pessoal; em máquinas remotas onde não é necessária a sua reserva; MV reservadas na nuvem e nós reservados pertencentes a um *Cluster*. É bastante importante ter em consideração que nos nós do *Cluster*, como referido na Secção 3.2.1.3, o lançamento dos componentes que dependem do Octave, só é possível usando Docker (**Cliente** e o **Executor**). Posto isto, esta limitação apresenta um papel importante na definição dos cenários de avaliação desta dissertação.

Para a avaliação da funcionalidade genérica será utilizado um exemplo de um programa Java, que tira partido de paralelismo, como se pode observar na Figura 4.1. Este simula aplicações que necessitem de realizar computações em paralelo, para obter o resultado mais rápido. As primeiras execuções desta funcionalidade serão realizadas na máquina pessoal, dando origem ao cenário **Função Genérica no PC (FGPC)**. Depois, também foram considerados cenários onde esta funcionalidade utiliza recursos remotos. Começando primeiro pela utilização de apenas uma máquina remota, considerou-se o cenário **Função Genérica para 1 máquina remota (FG1)**. De seguida, seria interessante verificar a funcionalidade quando os componentes se encontram distribuídos, entre máquinas, incluindo o próprio **Cliente**, para reduzir o impacto da comunicação PC / computadores remotos, que pode ser variável e de grande *overhead*, dando origem ao cenário **Função Genérica distribuída, incluindo o Cliente (FGDiC)**. Quando o **Cliente** se encontra na máquina pessoal, temos o cenário **Função Genérica distribuída, com o Cliente no PC (FGD)**. Visto que esta funcionalidade baseia-se na plataforma nativa e suporta diretamente o programa a executar, não necessita de imagens Docker no *Cluster*.

Para a avaliação da interface de programação Octave, o desempenho é avaliado experimentalmente, com o auxílio da *framework* SID-PSM [7], conforme os objetivos iniciais do trabalho. Tomou-se como referência a execução do SID-PSM original e, tendo em conta a necessidade da sua execução em Docker no *Cluster*, para este caso usou-se a execução do SID-PSM original em Docker. Foi também feita a comparação entre a execução **Original (O)** e **Original com Docker (OD)** para verificar eventuais *overheads* introduzidos pelo *container*.

Foram também considerados vários cenários para as diferentes Arquiteturas. Numa primeira avaliação, pode-se considerar apenas a execução paralela local, onde se procura explorar vários núcleos do computador (**LPC**) a executar a aplicação (Arquitetura Local). Outro caso será para a Arquitetura Distribuída, começando pela execução de todos os seus componentes também na máquina local (**DPC**), e, noutro cenário mais realista, lançando todos os seus componentes constituintes nos vários recursos remotos disponíveis, para explorar o paralelismo do sistema. Para o caso do *Cluster*, foram criadas **três imagens Docker** distintas:

- *Imagen Docker Completa*, que contem todos os componentes do sistema desenvolvido: **Cliente, Proxy e Executor**;
- *Imagen Docker Cliente*, que contém apenas o **Cliente**;
- *Imagen Docker Servidor*, que contém apenas o **Executor**.

A primeira imagem permite a execução completa do sistema, dentro de apenas um nó do *Cluster*, dando origem ao cenário **Com Docker para 1 nó (CD1N)**; As outras permitem a distribuição dos componentes, entre mais nós, dando origem ao cenário **Distribuído com Docker (incluindo o Cliente) (DDiC)**. É possível ainda considerar o cenário mais flexível para o utilizador, onde o **Cliente** se encontra no PC. Assim, originou-se o cenário **Distribuído com Docker (Cliente no PC) (DD)**.

Nos restantes cenários, com recursos remotos, todos os seus componentes são lançados de forma nativa, ou seja, diretamente na máquina sem a camada criada pelo Docker. As MV da Microsoft Azure e a **AMD/64** são os recursos usados nestes casos. Temos assim o cenário **Sem Docker para um nó (SD1N)**, o cenário **Distribuído sem Docker (incluindo o Cliente) (DiC)** e o cenário **Distribuído sem Docker (Cliente no PC) (D)**, cenários idênticos aos apresentados anteriormente, mas sem uso do Docker.

Os cenários referidos consideram um **Executor** (explorando vários núcleos de uma máquina), mas devemos também considerar vários **Executores**, em particular no *Cluster*. Definimos assim os cenários **Distribuído com 2 Executores com Docker (D2ED)** e o cenário **Distribuído com 2 Executores sem Docker (D2E)**, cobrindo a avaliação do comportamento quando se usam duas máquinas remotas.

Tabela 4.1: Visão geral dos cenários de avaliação apresentados.

	Interface de Programação Octave	Funcionalidade Genérica
PC	O, LPC, DPC	FGPC
AMD/64	SD1N	FG1
Microsoft Azure	DiC, D, D2E	FGDiC
Cluster	OD, CD1N, DDiC, DD, D2ED	FGDC, FGD

4.2.1 Escolha dos cenários a utilizar

Considerando todos os cenários, descritos anteriormente, estes não foram utilizados na totalidade para a avaliação do sistema.

No que toca aos cenários da funcionalidade constituído pelos cenários *Função Genérica (no PC) (FGPC)*, *Função Genérica (1 máquina remota) (FG1)*, *Função Genérica (distribuído, incluindo o Cliente) (FGDiC)* e *Função Genérica (distribuído, Cliente no PC) (FGD)*, sabendo que todos os componentes desta funcionalidade correm nativamente no *Cluster*, foi considerado apenas o cenário **Função Genérica (distribuído, Cliente no PC) (FGD)**, por ser realista e o que pode acarretar maiores *overheads* de rede.

Para ajudar na escolha dos cenários da Interface de Programação Octave, foram então realizadas **10 execuções** da *framework* SID-PSM para um problema em cada um dos cenários com uma só unidade de processamento. Este número de execuções serviu para a escolha não estar dependente de um único resultado, mas sim a partir da média dos resultados obtidos.

Todos os resultados encontram-se na Tabela 4.2. Para melhorar a compreensão dos resultados obtidos e da escolha dos cenários, foram criadas **três categorias**:

- Os cenários com código original: *Original (O)* e *Original com Docker (OD)*. O primeiro cenário foi executado nativamente e o segundo em um *container* Docker, na máquina pessoal;
- Os cenários **sem Docker**. Estes dividem-se em:
 - *Arquitetura Local (LPC)*, a funcionalidade desenvolvida na Secção 3.1.1.1. Esta executa na máquina pessoal;
 - *Arquiteturas Distribuídas*, que se podem sub-dividir conforme as máquinas utilizadas:
 - * *Na máquina pessoal (DPC)*;
 - * Na máquina **AMD/64**, onde se executou o cenário *Sem Docker para um nó (SD1N)*;
 - * Nas MV da Microsoft Azure, que permitiram a execução do cenário *Distribuído (incluindo o Cliente) sem Docker (DiC)* e *Distribuído sem Docker (Cliente no PC) (D)*.
- Os cenários que usam **Docker**: *Com Docker para 1 nó (CD1N)*, *Distribuído (incluindo o Cliente) com Docker (DDiC)* e *Distribuído com Docker (Cliente no PC) (DD)*. Estes cenários foram executados nos nós do *Cluster*.

Começando pelo cenário *Original* e pelo cenário *Original Docker*, pode-se concluir que, não considerando o *overhead* inicial correspondente ao arranque dos *containers*, basta apenas a utilização de um destes cenários, visto que são praticamente idênticos. Em média, o cenário *Original Docker* demora cerca de **0,55208 segundos**, enquanto que o cenário *Original* demora **0,55218 segundos**, de onde se obtém uma diferença de **0,0001 segundos**, ou seja, são execuções praticamente idênticas. Neste caso, e querendo comparar os vários cenários incluindo os que usam Docker, consideramos como referência o cenário **Original Docker (OD)**.

Para os cenários que **não utilizam Docker**, devido às limitações do cenário *LPC*, descritas na Secção 3.2.3.2, e aos resultados obtidos na Tabela, este não vai ser considerado como cenário.

O cenário *DPC*, apesar de bastante importante numa fase inicial de desenvolvimento, não é visto como um cenário real, a ser utilizado pelo sistema. Este apenas é utilizado para

uma comparação entre resultados obtidos inicialmente, e os resultados obtidos quando se distribuí o sistema. Posto isto, não há nenhuma vantagem em mantê-lo como cenário de avaliação.

O cenário **SD1N** engloba todos os componentes desenvolvidos na mesma máquina, assim utilizado também para identificar anomalias no acesso aos recursos remotos nas diversas interações entre componentes. Este cenário foi bastante útil para criar um ambiente real para a execução de um problema de otimização próximo da realidade, *STYRENE*, descrito na Secção 4.4.5.

O cenário **D** foi avaliado com êxito, utilizando os serviços da Microsoft Azure. Já o cenário **DiC** apresentou anomalias na execução e não foi possível completar. Devido ao limite dos custos associados à utilização da nuvem, podendo haver a hipótese da avaliação não se completar, nenhum destes cenários foram considerados na avaliação de desempenho.

Já para os cenários que **utilizam Docker** e começando pelo cenário **CD1N**, podemos observar que o tempo médio de execução do *bertsekas* (Tabela 4.2) está nos **168,55 segundos**, com desvio padrão em **1,38 segundos**. Para o cenário **DDiC**, a média do tempo está em **161,32 segundos**, com desvio padrão em **2,27 segundos**. Já no cenário **DD**, a média do tempo de execução está nos **179,19 segundos**, com **7 segundos** de desvio padrão.

Com os resultados obtidos, é possível verificar que quando utilizado o **Cliente** na máquina pessoal, para além dos tempos, serem mais instáveis, a média do tempo das execuções é maior. Para tal contribui a maior latência e menor *bandwidth* da rede, entre o PC e o nó remoto. Já o cenário **CD1N** e o cenário **DDiC** apresentam resultados bastante semelhantes. Isto significa que, mesmo com a redução do tempo dos *overheads*, utilizando apenas **1 máquina remota**, a divisão da capacidade da máquina entre os componentes remotos, equilibram o tempo da execução.

Posto isto, dentro destes cenários será utilizado o cenário **DDiC**, para realizar as restantes avaliações. Este cenário é um equilíbrio entre um ambiente realista, com eventuais *overheads* introduzidos pelo Docker no *Cluster*, mas melhor estabilidade nos resultados.

Adicionalmente, entre os cenários **D2ED** e **D2E**, foi apenas escolhido o cenário **D2ED**, quer por não existirem recursos disponíveis para se poder distribuir nativamente o sistema, por várias máquinas Azure, quer por o cenário no *Cluster* ser semelhante e diretamente comparável com os restantes cenários já lá definidos.

Assim, todos os cenários considerados para a avaliação do sistema, encontram-se na Tabela 4.3.

Tabela 4.2: Média e desvio padrão do tempo, em segundos, de 10 execuções realizadas da framework SID-PSM, para o problema de otimização *bertsekas*, com **1 unidade de processamento**, para cada um dos cenários envolvidos na avaliação da dissertação.

Cenário	O	OD	LPC	DPC	SDN1	D	CD1N	DDiC	DD
Média	0,55218	0,55208	437,64	275,67	135,68	261,68	168,55	161,32	179,19
Desvio Padrão	0,07253	0,02690	0,76768	9,34880	1,93092	1,23720	1,38004	2,27412	6,99521

Tabela 4.3: Cenários considerados para a avaliação do sistema.

Avaliação	Cenários	
	Nome	Sigla
Funcionalidade Genérica	Função Genérica (distribuído, Cliente no PC)	FGD
Interface de Programação Octave	Original com Docker	OD
	Distribuído (incluindo o Cliente) com Docker	DDiC
	Sem Docker para um nó	SD1N
	Distribuído com 2 Executores com Docker	D2ED

4.3 Avaliação da Funcionalidade Genérica

Observando o código-fonte, Figura 4.1, verifica-se que são definidas, no programa, **duas variáveis globais**: uma estrutura de dados, que neste caso se trata de uma fila, que suporta concorrência e um *THRESHOLD*, que representa o tamanho desta fila.

A variável *available* guarda o número de processadores disponíveis da máquina, através do método *Runtime.getRuntime().availableProcessors()*. Será criada uma *Thread Pool*, com o número de *threads* igual ao valor guardado, na variável *available*. Este valor é dependente da capacidade da máquina que será utilizada para a execução do programa.

Cada *thread* submetida, irá adicionar um número aleatório à fila. Quando acaba este número de submissões, acaba uma iteração. Enquanto existir capacidade para guardar elementos, estes vão sendo lançados, através das submissões para a *Pool*. Serão lançadas e executadas submissões, até acabar a capacidade da fila. Finalmente, será apresentado o número de iterações realizadas, para esgotar o espaço da fila, e o tempo que demorou.

No exemplo apresentado, a fila deste programa tem uma capacidade para **100 elementos**, ou seja, *THRESHOLD* = 100.

O gráfico da Figura 4.2 mostra o resultado da avaliação realizada. Inicialmente, o programa foi executado na máquina pessoal. Esta conta com **4 processadores disponíveis**, ou seja, até chegar à capacidade total da fila, 100, foram submetidas, para a *Pool* de **4 threads**, **8 pedidos** a cada iteração. Como se pode observar na barra de cima do gráfico, foram necessários **39,13 segundos** e **13 iterações** para esgotar a capacidade da fila.

De seguida foi utilizada a funcionalidade genérica, para enviar e executar este programa nos recursos remotos. Foi utilizado um dos nós do *Cluster*, que contém **16 processadores disponíveis**. Observando agora a barra de baixo, foram necessários apenas **12,01 segundos** da execução mais o tempo gasto nas comunicações, **0,448 segundos**, e **4 iterações**, para esgotar a capacidade da fila.

4.3. AVALIAÇÃO DA FUNCIONALIDADE GENÉRICA

```

public class Main {
    private static final int THRESHOLD = 100;
    private static BlockingQueue<Integer> queue;

    Run | Debug
    public static void main(String[] args) {
        queue = new ArrayBlockingQueue<Integer>(THRESHOLD);

        int available = Runtime.getRuntime().availableProcessors();

        System.out.println("Machine with " + available + " available processors, it will be launched " + available*2 + " requests at a time.");

        ExecutorService pool = Executors.newFixedThreadPool(available);

        int interations = 0;
        long start = System.currentTimeMillis();
        while (queue.remainingCapacity() > 0) {
            interations++;
            int counter = 0;
            while (counter < available * 2) {
                pool.submit(() -> {
                    try {
                        Thread.sleep(1000);
                        Random r = new Random();
                        int element = r.nextInt();
                        queue.add(element);
                        System.out.println("Finished! Added " + element + " to the Queue");
                    } catch (InterruptedException e) {
                        System.err.println("Sleep error");
                        System.exit(1);
                    }
                });
                counter++;
            }
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                System.err.println("Sleep error");
                System.exit(1);
            }
        }

        System.out.println("To acheive the size: " + THRESHOLD + " it was needed " + interations
            + " iterations. The execution took " + (System.currentTimeMillis() - start) + " ms");

        pool.shutdown();
        try {
            pool.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
        }
    }
}

```

Figura 4.1: Código-fonte exemplo programa, desenvolvido em Java.

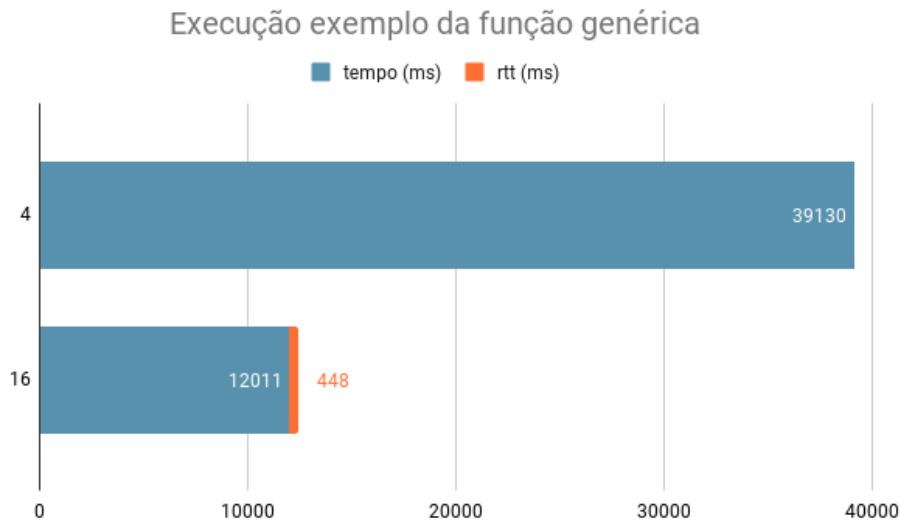


Figura 4.2: Resultado do programa, utilizando a máquina pessoal e a funcionalidade genérica.

Estes resultados demonstram que esta funcionalidade é bastante interessante para executar projetos demorados e/ou que tirem partido de paralelismo na sua implementação, em recursos remotos com maior capacidade e de forma fácil para o utilizador. Contudo, o fator limitante desta funcionalidade está na comunicação entre o **Cliente** e os recursos remotos.

Imagine-se a execução do mesmo projeto, mas agora em máquinas reservadas do outro lado do mundo e/ou projetos com grande volume de código, ou dados a transferir. O tempo dos *overheads*, ao aumentar devido à distância e/ou ao volume de transferência, poderia fazer com que a execução realizada, não demonstrasse um resultado promissor. Para isto acontecer, bastava que a soma dos *overheads* com o tempo de execução fosse igual ou superior ao tempo total da execução do programa, na máquina local. Para este exemplo:

$$39,13 \leq x + 12,011 \Leftrightarrow x \geq 27,119 \text{ segundos}$$

Assim, para este caso e para estas configurações, a transferência do programa para os recursos remotos e a receção da resposta, teriam de demorar, no mínimo, **27,119 segundos**, para não valer a pena tirar partido do recurso remoto.

4.4 Avaliação da Interface de Programação Octave

Para a avaliação experimental, foram usados **6 Problemas de Otimização**, descritos detalhadamente na Secção 4.4.1. Estes problemas compõem uma bateria de testes, habitualmente usada na avaliação de algoritmos e *frameworks* de resolução de problemas de optimização. Na Secção 4.4.2 serão apresentadas algumas considerações iniciais. Aqui é definida a base dos conceitos que serão utilizados, durante esta avaliação. Na Secção 4.4.3 estão descritos todos os cálculos efetuados para estimativa dos *overheads* envolvidos, bem como a sua validação. Na Secção 4.4.4 será analisado o comportamento do sistema desenvolvido, quando se utilizam **dois Executores**. Finalmente, na Secção 4.4.5, é realizada uma avaliação geral de um problema de otimização da indústria, mais realista.

4.4.1 Problemas de Otimização Utilizados

Como anteriormente referido, a aplicação desenvolvida em [7] é utilizada, como caso de teste e, como um possível exemplo de uma aplicação, que pode tirar partido das funcionalidades implementadas. Na Tabela 4.4 podemos observar a descrição dos problemas de optimização utilizados.

4.4. AVALIAÇÃO DA INTERFACE DE PROGRAMAÇÃO OCTAVE

Tabela 4.4: Problemas de Otimização utilizados para a avaliação, retirado de [7].

Problema	Dimensão (n, em \mathbb{R}^n)	Descrição	Função Objetivo
função exemplo	2	Função oferecida pela framework SID-PSM, utilizada como exemplo para resolver problemas de optimização, com restrições.	$f(x) = (x_2 - x_1^2)^2, x \in \mathbb{R}^2$
activefaces	10	Função activefaces em [55].	$F(x) = \max_{1 \leq i \leq n} \{g(-\sum_{i=1}^n x_i), g(x_i)\},$ onde $g(y) = \ln(y + 1)$ $x_i^{(1)} = 1.0$, para todo o $i = 1, \dots, n$.
arwhead	20	Problema descrito em [56].	$F(x) = \sum_{i=1}^{n-1} \{(x_i^2 + x_n^2)^2 - 4x_i + 3\}, x \in \mathbb{R}^n$
bdqrtic	30	Problema descrito em [56].	$F(x) = \sum_{i=1}^{n-4} \{(x_i^2 + 2x_{i+1}^2 + 3x_{i+2}^2 + 4x_{i+3}^2 + 5x_n^2)^2 - 4x_i + 3\},$ $x \in \mathbb{R}^n$
bdvalue	10	Problema de Teste número 28, em [57].	$f_i(x) = 2x_i - x_{i-1} - x_{i+1} + h^2(x_i + t_i + 1)^3/2$ onde $h = 1/(n+1)$, $t_i = ih$, e $x_0 = x_{n+1} = 0$
bertsekas	20	Função do exemplo número 3 em [58].	$f(x) = (1 + \sum_{i=1}^n (i \times x_i))^2,$ onde atinge o seu mínimo em $x^* = (0, \dots, 0)^T$

Estas funções constituem a bateria de testes utilizada para, além de confirmar o correto funcionamento, são habitualmente usadas na avaliação de algoritmos e *frameworks* de resolução de problemas de optimização, em trabalhos da área da matemática [7].

Estes problemas foram escolhidos devido a terem comportamentos diversos, como se pode verificar na Tabela 4.5, da execução referencial original. Os problemas *activefaces* e *bertsekas*, mesmo realizando menos iterações que os restantes problemas de optimização realizam, em média, mais avaliações por iteração em menos tempo. Já o problema *bdvalue* apresenta um comportamento oposto: elevada quantidade de avaliações e de iterações, mas apresenta o menor valor de avaliações, por iteração. Já os problemas *arwhead* e *bdqrtic* são os problemas de optimização com um comportamento intermédio na relação entre o número de avaliações por iterações, porém apresentam, à primeira vista, um comportamento crescente de complexidade.

Tabela 4.5: Número de avaliações por iteração, da bateria de testes, na versão sequencial da *framework* SID-PSM.

	Avaliações	Iterações	Média avaliação/iteração
activefaces	445	26	≈ 17
arwhead	1964	195	≈ 10
bdqrtic	2916	261	≈ 11
bdvalue	6011	1423	≈ 4
bertsekas	902	37	≈ 24

Pode-se também afirmar que nenhum destes problemas é perfeitamente paralelizável, ou seja, o número de avaliações não está igualmente distribuído pelas iterações de cada problema.

O problema base, escolhido para avaliações iniciais do sistema (Secção 4.2.1) foi o *bertsekas*. Esta escolha deve-se ao facto de, para além de uma execução bastante rápida, este é um bom problema para testar a paralelização, onde são realizadas **902 avaliações** em **37 iterações**, ou seja, em média o problema de otimização *bertsekas* realiza **24 avaliações por iteração** (Tabela 4.5), distribuídas pelos vários núcleos do **Executor**, tirando partido do paralelismo. Considerando que seria interessante utilizar um valor, de avaliações por iteração, maior do que o número de unidades de processamento inseridos (nesta avaliação, o número máximo utilizado são **16 unidades de processamento**), de todos os problemas de otimização selecionados, é o que poderá ser mais interessante analisar, em primeiro lugar.

É de assinalar que a versão original do SID-PSM foi desenvolvida em MATLAB e a interface de programação em Octave. Assim, não foi possível tirar partido de todas as funcionalidades disponíveis na *framework*, principalmente do *passo de procura*, referido na Secção 2.1.2, pois este usa operações não suportadas em Octave.

4.4.2 Considerações Relevantes

Devido ao modelo de execução, em paralelo, da interface de programação Octave desenvolvida, só se obtém um melhor desempenho, quando cada execução da função objectivo demora um tempo que compense os *overheads* da paralelização e comunicações. Ou seja, utilizando uma implementação sequencial se cada uma destas avaliações da função objectivo forem muito rápidas, o tempo que se demora, nestas avaliações é demasiado pequeno para beneficiar da paralelização. Tal verifica-se com os problemas, nesta bateria de testes ainda mais por serem efectuadas poucas avaliações por iteração.

Assim, em alguns casos, é muito mais vantajoso utilizar uma abordagem sequencial (sem a utilização de recursos remotos), tanto para a poupança de recursos das máquinas, como para poupar no *overhead*, criado pela distribuição dos pedidos, entre os recursos, e pela comunicação da máquina pessoal com a infraestrutura remota. Já quando as execuções são mais demoradas, surge uma questão: será que o tempo de execução obtido pela arquitetura em paralelo, compensa os *overheads* impostos?

Para se responder a esta questão, será bastante relevante definir, *a priori*, alguns aspetos. Começando pela definição do tempo sequencial, T_{seq} , e em paralelo, $T_{par}(p)$, de uma execução, em p processadores ou *threads*. Considerando as Equações 2.1 e 2.2 obtidas na Secção 2.2.2.5, têm-se:

$$T(1) = T_{seq} = C_{seq} + C_{par} \quad (4.1)$$

$$T_{par}(p) = C_{seq} + \frac{C_{par}}{p} + C_{Overheads}. \quad (4.2)$$

C_{seq} e C_{par} correspondem, respetivamente, ao tempo de execução que é estritamente sequencial e ao tempo da parte da execução que pode ser paralelizável. É de referir que, ao longo deste documento, a variável $T_{par}(p)$, também será referida como T_{par} , para simplificar a sua leitura, tanto em equações, como em tabelas.

Posto isto, é possível relacionar o tempo total de uma execução em paralelo, $T_{par}(p)$, com o tempo total de uma execução sequencial, T_{seq} , sendo o primeiro igual ou inferior:

$$T_{par}(p) \leq T_{seq}$$

Substituindo pelas Equações 4.1 e 4.2, vem:

$$\frac{C_{par}}{p} + C_{overheads} \leq C_{par} \quad (4.3)$$

Resolvendo, em ordem a C_{par} , obtém-se:

$$C_{par} \geq \frac{p \times C_{overheads}}{(p - 1)} \quad (4.4)$$

Este indica o tempo de processamento da parte paralelizada que, dependendo do número de unidades de processamento e dos *overheads*, torna vantajosa a versão paralela em comparação à execução sequencial original. Ou, em média, relativamente ao número de avaliações efetuadas, $C_{par/aval}$:

$$C_{par/aval} = \frac{C_{par}}{num_avals} \geq \frac{p \times C_{overheads/aval}}{(p - 1)} \quad (4.5)$$

Esta divisão é realizada para se obter o tempo, em média, que demora uma avaliação a computar. Contudo, conforme a definição de T_{par} , como ainda não se sabe a influência do peso dos *overheads*, $C_{overheads}$, na execução, deve-se começar com a obtenção deste.

4.4.3 Cálculo do peso dos *overheads*, $C_{overheads}$

Tendo em consideração a Equação 4.2 e resolvendo em ordem a $C_{overheads}$, obtém-se a expressão:

$$C_{overheads} = T_{par}(p) - \frac{C_{par}}{p} - C_{seq} \quad (4.6)$$

E temos, em média, por cada avaliação da função objetivo:

$$C_{overheads/aval} = \frac{C_{overheads}}{num_avals} = \frac{T_{par}(p) - \frac{C_{par}}{p} - C_{seq}}{num_avals} \quad (4.7)$$

Que representa o peso dos *overheads*, por avaliação, que se quer estimar em função de p (número de **unidades de processamento**).

A partir do estudo realizado na Secção 4.2.1 obtivemos os valores de T_{seq} e T_{par} para o problema de otimização *bertsekas*. Foi necessária a obtenção dos componentes da definição

de tempo sequencial, T_{seq} , e paralelo, T_{par} e, em consequência, os valores de C_{seq} e C_{par} , não só para este problema mas também para toda a bateria de testes. Deste modo, obteve-se a Tabela 4.6. Estes valores foram estimados com base na definição de C_{seq} e C_{par} e pela Equação 4.1. Para este caso foi considerada parte paralelizável o ciclo de avaliação da função para cada iteração. De seguida, foi calculado o tempo aproximado da execução que é paralelizável C_{par} : $C_{par} = T_{seq} - C_{seq}$.

Tabela 4.6: Valores médios de C_{par} e C_{seq} para a bateria de testes.

Problema	T_{seq} (segundos)	C_{par} (segundos)	C_{seq} (segundos)
activefaces	0,361	0,149	0,212
arwhead	2,61	0,082	2,528
bdqrtic	9,368	0,195	9,173
bdvalue	6,571	0,455	6,116
bertsekas	0,475	0,034	0,442

Substituindo, na Equação 4.7:

- T_{par} , pela média dos tempos das execuções realizadas, no cenário **DDiC**;
- T_{seq} , pela média dos tempos das execuções realizadas, no cenário **OD**;
- **1 unidade de processamento**, $p = 1$ (pior caso);
- O número de avaliações total da função do problema, num_avals .

E, considerando que as condições de execução são iguais, tanto na versão original (sequencial) como na utilização da arquitetura em paralelo, a Equação 4.7 mostra o peso máximo para este exemplo, que têm os *overheads* da arquitetura que suporta esta *framework* (**Proxy**, **Executor**, etc.). Assim, o objetivo final será encontrar um tempo para a execução da função objetivo que representa um limite para que se começem a obter as melhorias desejadas com as versões paralelas usadas neste trabalho.

Para um resultado não dependente de um único problema de otimização, foram realizadas avaliações com a bateria de testes definida, obtendo-se a Tabela 4.7. Calculando o peso dos *overheads*, $C_{overheads/aval}$, obteve-se a Tabela 4.8.

Assumindo que o comportamento da rede utilizada para fazer os testes é constante, o peso dos *overheads* por avaliação, deveria também de permanecer constante. Ao paralelizar a aplicação do utilizador, o peso dos *overheads* é amortizado quando se sobrepõe a comunicação com a computação, ou seja, enquanto se aguarda a resposta de uma computação, outros pedidos ou respostas e computações podem ser feitas, como exemplificado na Figura 4.3. Assim, o peso dos *overheads* obtido é inversamente proporcional ao número de unidades de processamento, p , utilizados na configuração do sistema.

A Tabela 4.9 demonstra exatamente o que foi descrito. Tal variação não é possível verificar utilizando apenas a Equação 4.7 para calcular $C_{overheads/aval}$, como se pode observar na Tabela 4.10.

4.4. AVALIAÇÃO DA INTERFACE DE PROGRAMAÇÃO OCTAVE

Tabela 4.7: Resultados obtidos da execução da bateria de testes do cenário **OD** e do cenário **DDiC** com **1 unidade de processamento**, realizadas pela *framework* SID-PSM.

	Problema	tempo (segundos)	avaliações	tempo/avaliação
OD	activefaces	0,3613	445	0,00081191
	arwhead	2,61	1964	0,001328921
	bdqrtic	9,368	2916	0,00321262
	bdvalue	6,571	6011	0,001093163
	bertsekas	0,4752	902	0,000526829
DDiC	activefaces	80,84	445	0,18166292
	arwhead	377,8	1964	0,19236253
	bdqrtic	568	2916	0,19478738
	bdvalue	1289	6011	0,21444019
	bertsekas	164,9	902	0,18281596

Tabela 4.8: Peso médio dos *overheads*, $C_{overheads/aval}$, para cada um dos problemas de otimização constituintes da bateria de testes, calculado através da Equação 4.7.

Problema	$C_{overheads/aval}$ (segundos)
activefaces	0,18
arwhead	0,19
bdqrtic	0,19
bdvalue	0,21
bertsekas	0,18
Média	0,19

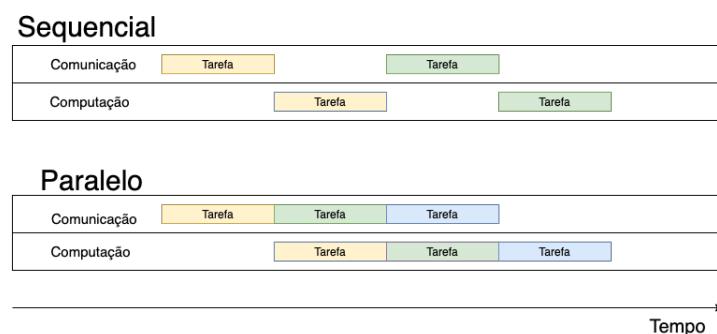


Figura 4.3: Comparação entre sistemas sequenciais com sistemas paralelos, onde o último utiliza sobreposição entre a comunicação e a computação de tarefas.

Também se pode observar que o número de avaliações aumentou com o número de processadores. Parecendo um problema crítico, este resultado não é tão incomum e deve-se ao facto de o sistema estar implementado, usando execuções paralelas e de forma assíncrona. Isto significa que, enquanto na versão sequencial as avaliações param numa iteração assim que um melhor resultado for encontrado, nesta versão serão lançadas p avaliações em paralelo de cada vez. Outro aspeto é que mesmo que os pedidos sejam enviados por ordem, há uma probabilidade da resposta destes pedidos não chegarem ao **Cliente**, por essa mesma ordem. Assim, a ordem das avaliações difere e o algoritmo de optimização pode seguir um percurso diferente na busca da melhor solução, que leva a esta diferença de comportamento e respetivo impacto no tempo total de execução.

Tabela 4.9: Variação do peso dos *overheads*, $C_{\text{Overheads/aval}}$, com o aumento das unidades de processamento, p , para o problema de otimização *bertsekas*, utilizando o cenário **DDiC**.

Unidades de processamento, p	tempo (segundos)	avaliações	$C_{\text{Overheads/aval}}$ (segundos/avaliação)
2	95,09	902	0,10515787
4	66,02	913	0,07218094
8	51,92	937	0,05534749
16	46,72	941	0,04961775

Tabela 4.10: Peso dos *overheads*, $C_{\text{Overheads/aval}}$, estimado, utilizando o tempo e o número de avaliações, calculados com a Equação 4.7 para o problema de otimização *bertsekas*.

tempo (segundos)	avaliações	unidades de processamento, p	$C_{\text{Overheads/aval}}$ (segundos/avaliação)
161,32	902	1	≈0,1783
		2	≈0,1786
		4	≈0,1787
		8	≈0,1788
		16	≈0,1788

Utilizando a Equação 4.5, os valores de $C_{\text{Overheads/aval}}$, previamente calculados para a bateria de problemas, e variando o número de unidades de processamento, p , foram estimados os valores presentes na Tabela 4.11. Podemos também observar o resultado pretendido, $C_{\text{par/aval}}$, de uma maneira mais simplista através da Figura 4.4.

Para confirmar se estas são aproximações válidas da realidade sem tornar a análise exaustiva, acrescentou-se este tempo de espera, $C_{\text{par/aval}}$, na função objetivo apenas ao problema base, *bertsekas*, e aos problemas *activefaces* e *bdvalue*. Estes dois últimos representam os dois extremos da bateria de testes utilizada, no número de avaliações e no tempo de execução. Foram utilizados os cenários, **OD** e **DDiC** e o resultado esperado será um valor para o tempo total de execução, semelhante nos dois cenários. Obtiveram-se os resultados da Tabela 4.12, podendo-se concluir que:

- Se verifica uma oscilação do rácio entre **0,86** e **1,07**, valores muito próximos de **1**, que seria o valor ideal;

4.4. AVALIAÇÃO DA INTERFACE DE PROGRAMAÇÃO OCTAVE

- As piores estimativas foram para o *bdvalue*, onde $C_{par/aval}$ peca normalmente por defeito. Também os tempos de avaliação sequencial e em paralelo apresentaram valores bastante diferentes (por exemplo, para $p = 2$ a versão em paralelo demorou cerca de mais **295 segundos** que a versão original). Para tal, parece contribuir as avaliações extra que se pode observar principalmente para $p = 8$ e $p = 16$, o número de avaliações sobe consideravelmente (por exemplo, para $p = 16$ são realizadas mais **3435 avaliações** que a versão original);
- Como se pode observar na Tabela 4.5, o problema *bdvalue* é, entre os problemas que constituem a bateria de testes, o que conta com menos avaliações por iteração, **4 avaliações por iteração**. Até ao número de unidades de processamento ser igual ao número anteriormente referido, $p = 4$, o tempo melhora consideravelmente. Mas, no momento em que este valor é ultrapassado, o número de avaliações total do problema aumenta drasticamente (**7305** para $p = 8$ e **9478** para $p = 16$, considerando que na versão sequencial o número de iterações é **6011**). Contudo, se for avaliado, não só este tempo, mas sim a sua razão com o número de avaliações realizado:

$$\frac{\text{tempo}}{\text{num_avals}} \Rightarrow \frac{861,5}{7305} = 0,1179 \text{ e } \frac{1030}{9478} = 0,1087,$$

observar-se ainda há melhoria de tempo por avaliação;

- Considerando que os tempos de execução dos dois cenários, para o $C_{par/aval}$ calculado, é muito semelhante, pode-se concluir que se obteve, possivelmente, o que se pretendia: um ponto de viragem, ou seja, o limite inferior onde o sistema começará a proporcionar um melhor desempenho do que a versão original (sequencial);

Tabela 4.11: Estimativas para a bateria de testes, para $C_{overheads/aval}$ e para $C_{par/aval}$, através da utilização das Equações 4.7 e 4.5, respetivamente.

Problema	Unidades de processamento, p	tempo (segundos)	avaliações	$C_{overheads/aval}$ (segundos/avaliação)	C_{par} (segundos)	$C_{par/aval}$ (segundos/avaliação)
activefaces	2	47,99	445	0,1074	95,62	0,2149
arwhead		236,4	1965	0,1196	470,19	0,2393
bdqrtic		351,2	2921	0,1186	693,03	0,2373
bdvalue		876	6018	0,1450	1745,43	0,2900
bertsekas		95,09	902	0,1052	189,70	0,2103
activefaces	4	32,03	446	0,0716	42,59	0,0955
arwhead		183,4	2012	0,0908	243,66	0,1211
bdqrtic		268	2996	0,0887	354,21	0,1182
bdvalue		768,4	6416	0,1223	1046,34	0,1631
bertsekas		66,02	913	0,0722	87,87	0,0962
activefaces	8	25,75	450	0,0571	29,38	0,0653
arwhead		174,6	2148	0,0811	199,17	0,0927
bdqrtic		248,5	3209	0,0771	282,66	0,0881
bdvalue		861,5	7305	0,1178	983,63	0,1347
bertsekas		51,92	937	0,0553	59,27	0,0633
activefaces	16	22,85	453	0,0504	24,35	0,0538
arwhead		188,6	2394	0,0787	201	0,0840
bdqrtic		271,2	3494	0,0775	288,66	0,0826
bdvalue		1030	9478	0,1086	1098,23	0,1159
bertsekas		46,72	941	0,0496	49,80	0,0529

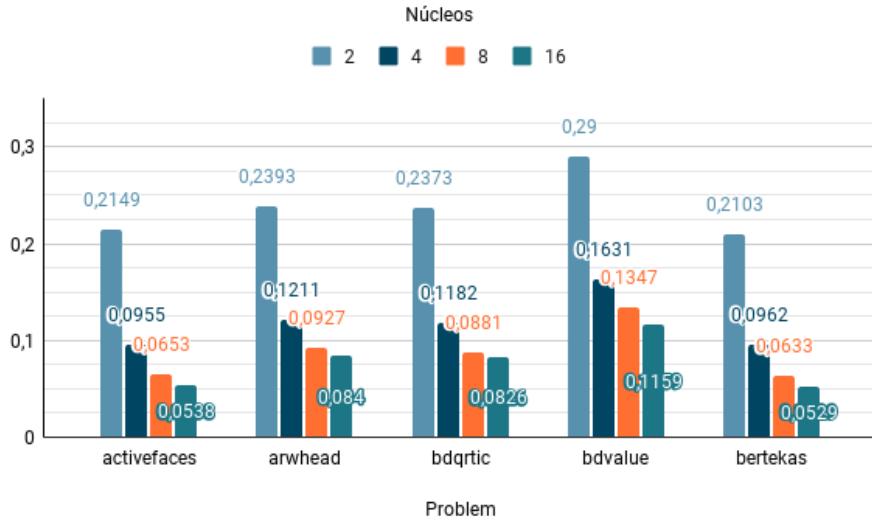


Figura 4.4: Representação, em gráfico, da coluna $C_{par/aval}$ da Tabela 4.11.

Tabela 4.12: Verificação dos resultados do problema base (*bertsekas*) e dos dois extremos dos problemas de otimização (*activefaces* e *bdvalue*), utilizando os valores estimados da Tabela 4.11.

	Problema	$C_{par/aval}$ (segundos)	tempo (segundos)	avaliações	tempo/avaliação	Rácio
OD	bertsekas	0,21	190,4	902	0,2188	0,96
DDiC $p = 2$			197,4	902	0,2111	
OD	activefaces	0,21	93,96	445	0,2111	1,00
DDiC $p = 2$			93,96	446	0,2107	
OD	bdvalue	0,29	1755	6011	0,292	0,86
DDiC $p = 2$			2050	6014	0,3409	
OD	bertsekas	0,096	87,47	902	0,0969	1,00
DDiC $p = 4$			88,14	910	0,0970	
OD	activefaces	0,10	43,02	445	0,0967	0,94
DDiC $p = 4$			45,89	448	0,1024	
OD	bdvalue	0,16	973,5	6011	0,162	0,88
DDiC $p = 4$			1140	6209	0,1836	
OD	bertsekas	0,063	57,74	902	0,0635	1,01
DDiC $p = 8$			58,58	922	0,0640	
OD	activefaces	0,07	29,61	445	0,0665	1,07
DDiC $p = 8$			27,91	448	0,0623	
OD	bdvalue	0,13	821	6011	0,1366	0,92
DDiC $p = 8$			1030	6940	0,1484	
OD	bertsekas	0,053	48,73	902	0,0531	1,02
DDiC $p = 16$			49,42	934	0,0540	
OD	activefaces	0,05	24,51	445	0,0551	1,03
DDiC $p = 16$			24,04	450	0,0534	
OD	bdvalue	0,12	708,5	6011	0,1179	1,07
DDiC $p = 16$			1040	9446	0,1101	

Para uma verificação final do comportamento da bateria de testes foram realizadas novas execuções, tanto para o cenário **OD** como para o cenário **DDiC**, acrescentando na função objetivo uma espera com o tempo $C_{par/aval} = 0,3\text{ segundos}$, o arredondamento do valor máximo encontrado na Tabela 4.11.

Observando os gráficos resultantes, Figuras 4.5, 4.6 e 4.7, das execuções pode-se concluir que a partir das duas unidades de processamento, $p = 2$ todos os problemas da bateria de testes, à exceção do problema *bdvalue* obtém-se a interseção (ou ponto de viragem, referido anteriormente), dos tempos de execução do cenário **OD** e do cenário **DDiC**. Isto significa que, quando utilizadas **2 unidades de processamento**, já se consegue obter um desempenho equivalente, à da versão sequencial original. Por outras palavras, se a função a avaliar demorar mais de $0,3\text{ segundos}$ e um número de avaliações por iteração suficiente, a utilização da interface de programação já é mais aconselhável, do que a versão original (sequencial).

Conforme as métricas de desempenho, estudadas na Secção 2.2.2 e a partir dos resultados obtidos anteriormente e das métricas calculadas e apresentadas na Tabela 4.13, verifica-se que:

- Como era esperável, pela definição de *speedup*, o problema *bdvalue* apresenta os piores valores desta métrica. Apesar do tempo total da execução em paralelo, T_{par} , a partir de $p = 3$, seja melhor que o tempo da execução sequencial, T_{seq} , podemos verificar que a partir de $p = 8$ para $p = 16$ há um aumento de T_{par} (de **1296** para **1329 segundos**). Pode-se verificar também, comparando os restantes problemas, *bertsekas* e *activefaces*, que quanto mais baixo é o valor da diferença $|T_{par} - T_{seq}|$, melhor é o *speedup*;
- A eficiência dos **três** problemas, métrica que mede a utilização da capacidade computacional, mostra que se obteve o comportamento esperado. Pela lógica, à medida que aumenta p , a eficiência dos recursos começa a degradar-se. O problema *bdvalue* apresenta os piores valores desta métrica e o problema *bertsekas* os melhores. Pode-se observar também que, devido à queda do *speedup*, no problema *bdvalue*, de $p = 8$ para $p = 16$, a eficiência deste também piora;
- A redundância avalia o rácio entre o número de operações realizadas numa execução sequencial e em paralelo. Variando p , há um aumento de redundância todos os problemas de otimização. Mas, o maior valor verifica-se no problema *bdvalue*, onde as suas avaliações em paralelo, *avaliações_par*, sobem drasticamente. Este aumento significativo acaba por degradar o desempenho da execução realizada, como se observa entre $p = 8$ e $p = 16$, onde são realizadas **500** avaliações a mais;
- Finalmente, verificando os valores de utilização, a métrica que mede o bom uso da capacidade computacional observa-se que, geralmente, o sistema mostra um mau uso da capacidade computacional disponibilizada pelos recursos remotos, isto

porque a grande parte das computações da *framework*, não se consegue paralelizar, Tabela 4.6.

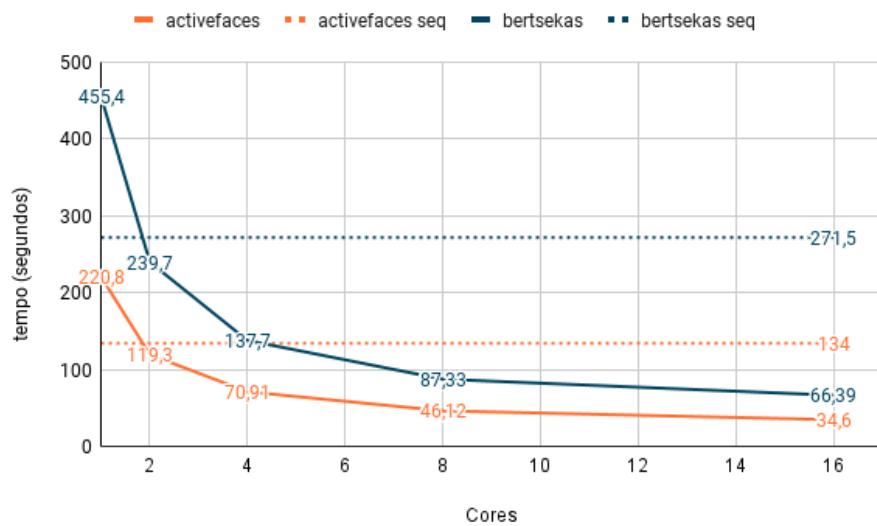


Figura 4.5: Tempo total de execução sequencial vs. paralelo, para os problemas de otimização *bertsekas* e *activefaces*, utilizando **0,3 segundos/avaliação** para $C_{par/aval}$.

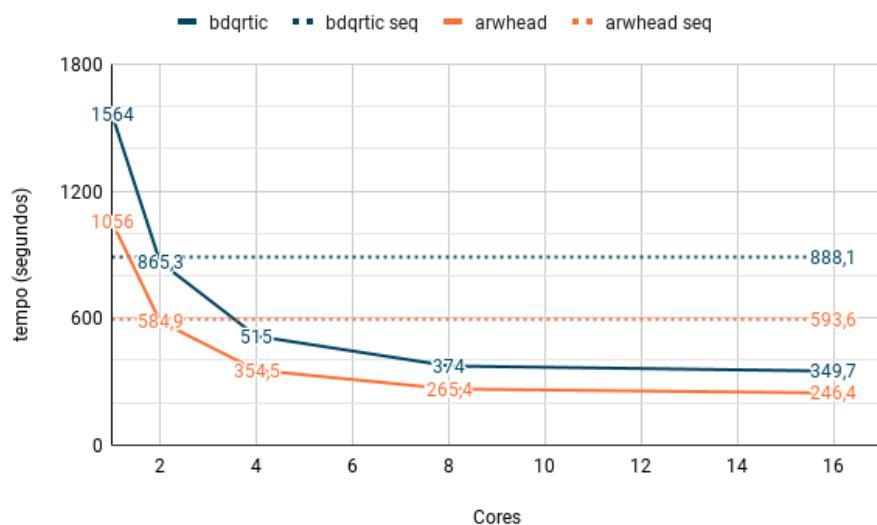


Figura 4.6: Tempo total de execução sequencial vs. paralelo, para os problemas de otimização *bdqrtic* e *arwhead*, utilizando **0,3 segundos/avaliação** para $C_{par/aval}$.

4.4. AVALIAÇÃO DA INTERFACE DE PROGRAMAÇÃO OCTAVE

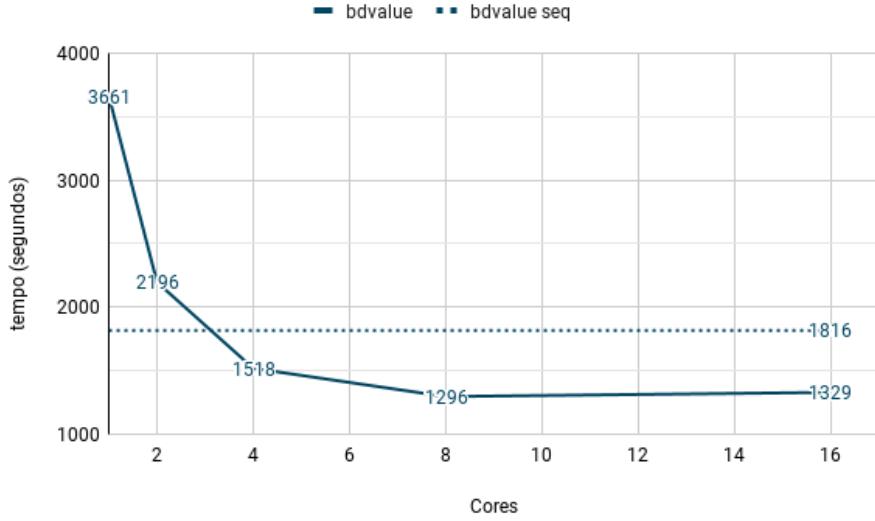


Figura 4.7: Tempo total de execução sequencial vs. paralelo, para o problema de otimização *bdvalue*, utilizando 0,3 segundos/avaliação para $C_{par/aval}$.

Tabela 4.13: Métricas de desempenho dos problemas de otimização *bertsekas*, *activefaces*, *bdvalue*.

Problema (0,3 $C_{par/aval}$)	unidades de processamento, p	T_{par} (segundos)	avaliações_par	T_{seq} (segundos)	avaliações_seq	Speedup	Eficiência	Redundância	Utilização
bertsekas	1	455,4	902	271,5	902	0,596	0,596	1	0,596
	2	239,7	904			1,133	0,566	1,002	0,568
	4	137,7	909			1,972	0,493	1,008	0,497
	8	87,33	917			3,109	0,389	1,017	0,395
	16	66,39	937			4,089	0,256	1,039	0,266
activefaces	1	220,8	445	134	445	0,607	0,607	1	0,607
	2	119,3	445			1,123	0,562	1	0,562
	4	70,91	447			1,890	0,472	1,004	0,475
	8	46,12	450			2,905	0,363	1,011	0,367
	16	34,6	447			3,873	0,242	1,004	0,243
bdvalue	1	3661	6011	1816	6011	0,496	0,496	1	0,4960
	2	2196	6016			0,827	0,413	1,001	0,414
	4	1518	6104			1,196	0,299	1,015	0,304
	8	1296	6609			1,401	0,175	1,099	0,193
	16	1329	8669			1,366	0,085	1,442	0,123

4.4.4 Avaliação com dois Executores

Será interessante avaliar, dividindo a carga de trabalho por mais do que uma máquina remota, o comportamento do sistema, quando se usa a interface de programação Octave. Por exemplo, se será mais vantajoso, utilizar uma máquina remota com **16 unidades de processamento** ou duas máquinas com **8 unidades de processamento**, cada uma.

É assim importante avaliar a interface no cenário **Distribuído com 2 Executores com Docker**, definido na Secção 4.2.1.

Nesta avaliação foi realizada uma execução da bateria de testes para **1 Executor** de **16 unidades de processamento**, $p = 16$, e foi realizada também uma execução, agora com **2 Executores**, cada uma com $p = 8$. Devido à configuração do sistema, onde cada componente remoto fica alocado em apenas uma máquina remota e, como os nós do *Cluster*, com as características pretendidas, são apenas três (agora **1 nó** contém a **Proxy** e

2 **nós**, contém 1 **Executor** cada), para esta análise foi utilizada também a máquina pessoal, como componente **Cliente**.

Os resultados estão presentes na Tabela 4.14, onde se pode observar um comportamento bastante interessante: os tempos de execução, para o mesmo número de unidades de processamento, manteve-se. Há ligeiras diferenças de poucos segundos (a maior diferença pode-se observar no problema *arwhead*, de 21,4 segundos), onde em média, o rácio ronda os 1,036%. Contudo, ainda podemos observar que, para os 2 **Executores**, são realizadas menos avaliações do que na versão de apenas uma.

Podemos assim concluir que efetivamente se nota um peso dos *overheads* crescente no sistema, proporcional ao número de **Executores** utilizados. Contudo, de um ponto de vista geral, este aumento não é significativo, sendo ainda amortizado pela diminuição de avaliações realizadas. Ou seja, em desempenho o aumento de máquinas introduz um pequeno *overhead* extra, mas permite dispor de mais unidades de processamento que será uma grande mais-valia para problemas de maior dimensão, com maior número de avaliações em cada iteração (com maior número de funções para avaliar em paralelo).

Tabela 4.14: Execução da bateria de testes com **dois Executores** de 8 **unidades de processamento** cada e de um **Executor** de 16 **unidades de processamento**.

	Problema	tempo (segundos)	avaliações
2 Executores com 8 unidades de processamento, p	activefaces	37,29	446
	arwhead	366,4	2024
	bdqrtic	672,7	3092
	bdvalue	1276	6560
	bertsekas	80,4	917
1 Executor com 16 unidades de processamento, p	activefaces	35,37	448
	arwhead	345	2199
	bdqrtic	655,7	3322
	bdvalue	1275	7695
	bertsekas	77,58	920

4.4.5 Avaliação do sistema usando o problema STYRENE

Todos os problemas de otimização anteriores são apenas utilizados em ambientes académicos. Como se pode observar na Tabela 4.6, os tempos reais de execução destes problemas, na versão sequencial, são bastante baixos. Para todo o trabalho anterior foi necessária a criação de casos, onde os tempos das suas execuções, na versão sequencial, são forçados a demorar um tempo mínimo para começar a valer a pena o paralelismo. Assim, os resultados anteriores, mesmo sendo promissores, são irrealistas.

Para verificar o comportamento do sistema desenvolvido no mundo real, é necessário também um problema de otimização real. O STYRENE é um problema de otimização referenciado como um caso de *benchmarking*, para a comunidade de otimização de funções sem derivadas (**OSD**). A função a avaliar simula o processo de criação do estireno, visto como uma simulação de caixa preta [59]. A produção do estireno é dividida em quatro passos: preparação dos reagentes, onde é feito um aumento de pressão e realizada a

evaporação; reações catalíticas; recuperação do estireno, com a primeira destilação; e recuperação do benzeno, com a segunda destilação. Este problema de otimização procura a maximização do seu valor, sendo este sujeito a diversas restrições do seu processo incluindo restrições económicas.

O problema é definido por **8 valores de otimização** e **11 constantes**, divididas em dois grupos: **4 restrições** não relaxáveis e não quantitativas e **7 restrições** relaxáveis e quantitativas. A sua execução devolve **12 outputs**, **11 restrições** e o **valor objetivo**. Um ponto é fazível quando os primeiros **11 outputs** são abaixo ou iguais a zero.

Para esta avaliação, o *STYRENE* foi inteiramente executado (versão sequencial e em paralelo) na máquina de 64 núcleos, descrita na Secção 4.1. Como é apenas uma máquina, todos os componentes do sistema desenvolvido (**Cliente**, **Proxy** e **Executor**) foram lançados nesta e, como esta máquina tem suporte para todas as linguagens de programação utilizadas, não foi necessária a utilização de imagens Docker.

Começou-se por realizar a avaliação sequencial e a avaliação em paralelo com **1 núcleo** ($p = 1$) para processamento. Como esperado, a versão sequencial apresenta melhores resultados que a versão em paralelo para $p = 1$. Isto porque, para além da computação do *STYRENE*, também existem os *overheads* impostos na versão em paralelo. Estes *overheads* são atenuados à medida que se aumenta o número de núcleos, começando a melhorar significativamente para $p = 4$ (ver Figura 4.8). Durante todas as avaliações até $p = 16$, alguns dos pedidos de execução entravam na fila de espera para serem avaliados. Quando se executou o *STYRENE* para $p = 32$, verificou-se que nenhum pedido ficou em fila de espera, para ser executado, e obtivemos também algumas estatísticas: no mínimo estavam **15 threads**, no máximo **25 threads** e, em média, **17 threads** ativas, durante estas execuções ($p = 32$). Tal é sinal do número de avaliações que podem ser executadas em paralelo em cada iteração. Ou seja, $p = 32$ já temos processadores mais do que suficientes para a execução do *STYRENE*. Finalmente, foi avaliado o comportamento do sistema para $p = 64$, onde se pode verificar que os resultados são bastante semelhantes aos resultados obtidos para $p = 32$, o que demonstra não haver vantagens para $p > 32$. Assim, o valor mais alto do *speedup* que se pode obter para este problema (para $p = 32$), nas configurações utilizadas é de, 2,24 e uma eficiência de **0,07**. A Figura 4.8 mostra os tempos obtidos de todas as avaliações descritas.

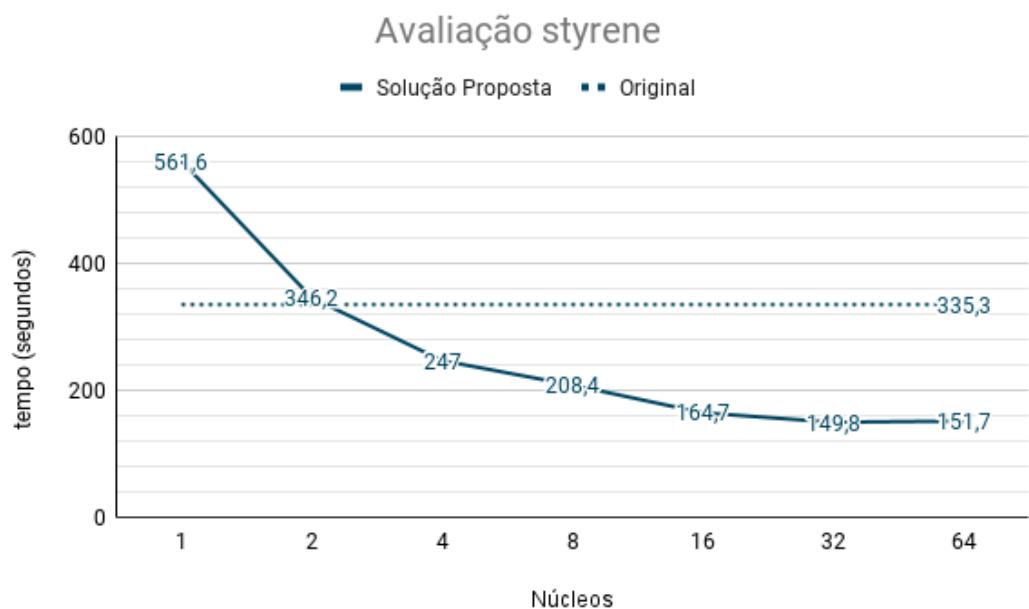


Figura 4.8: Execução do STYRENE na versão sequencial (original) e em paralelo (Solução desenvolvida), variando o número de núcleos.

CONCLUSÕES

Neste último Capítulo, serão discutidos os pontos relevantes, positivos e negativos, de proposta e sua implementação que tornaram os objetivos traçados na Secção 1.2 possíveis, concluindo assim toda a descrição do trabalho desenvolvido. Também, na Secção 5.1, serão apresentadas algumas ideias para trabalho futuro.

Começando pela Funcionalidade Genérica, um primeiro passo para o sistema final, podemos observar que o objetivo principal se cumpriu: permitir ao utilizador do sistema a execução das suas aplicações em recursos remotos, de forma direta e eficaz, onde a sua utilização também pode ser realizada recorrendo à linha de comandos da máquina pessoal. Esta Funcionalidade tem capacidade de enviar a aplicação com os argumentos a executar e suportar qualquer linguagem de programação, desde que o suporte a esta seja devidamente instalado no **Executor** do sistema. A funcionalidade não tolera falhas durante as execuções e, se a ligação entre o utilizador e o **Executor** sofrer uma quebra, todo o progresso é perdido. Conforme os resultados obtidos na avaliação, os *overheads* impostos pela utilização dos recursos remotos foram de **0,448 segundos**, para a aplicação de exemplo e a configuração de recursos utilizada, bastante aceitável para o que se pretende.

Considerando a Interface de Programação Octave, a partir da análise da *framework* SID-PSM conseguiu-se efetivamente obter uma alternativa para a execução deste código em recursos remotos de diferentes tipos e tirando partido de execução em paralelo. Esta já era composta originalmente por duas versões, uma de execução sequencial e uma de execução em paralelo (utilizando a extensão Parallel Toolbox MATLAB). No entanto, durante esta dissertação, como foi utilizado apenas Octave, criando-se mais uma alternativa para a execução do SID-PSM em paralelo, fácil de adaptar para MATLAB, futuramente. Independentemente do seu custo associado à licença necessária no caso do MATLAB, foi averiguado como realizar as mesmas operações que o Octave realiza neste trabalho, especialmente a execução de funções a partir da linha de comando. O Octave, no Windows (*cmd*) ou UNIX (*shell*), pode ser executado através dos comandos:

```
octave --eval "[expressão]" ou octave --eval nome_do_ficheiro.m
```

Para proceder à avaliação da expressão ou de um ficheiro, nessa linguagem. O mesmo

comportamento para o MATLAB pode ser obtido via:

```
matlab -nosplash -nodesktop -minimize -r “[expressão]; exit” -wait -log
```

Onde:

- *-nosplash –nodesktop –minimize* são utilizadas para não abrir, a partir da linha de comandos, o ambiente de programação (*IDE*) do MATLAB;
- *-r* significa que se quer executar, *run*, o próximo parâmetro que se irá inserir;
- *-wait –log* também têm de estar presentes, para o resultado ser mostrado, na linha de comandos utilizada.

Em qualquer dos casos, esta forma de lançar a avaliação, contribui para os *overheads* observados de momento, onde cada vez que este comando é executado, uma nova instância do interpretador é criada. Por exemplo, considerando que se realizarão **10 pedidos**, então serão também inicializadas **10 instâncias**, para executar cada um destes. Cada inicialização tem um custo, em tempo, que multiplicado pelo número de pedidos recebidos, comprometendo bastante o desempenho de todo o trabalho desenvolvido e será de considerar em trabalhos futuros.

Devido à grande diversidade de opções para a implementação de sistemas, que permitem execuções em recursos remotos, a escolha certa foi um grande desafio. Contudo, depois dos resultados obtidos, pode-se concluir que todas as escolhas feitas, durante o desenvolvimento desta dissertação, foram aceitáveis e que a arquitetura do protótipo desenvolvido, baseado em **Proxy** para acesso e distribuição, e **Executor** para a execução em paralelo, facilita a adaptação a novas situações sendo independente da linguagem da aplicação.

Olhando para tudo o que foi desenvolvido, é importante salientar algumas limitações que se encontraram para definir o resultado. Uma destas limitações foi da **Execução Local**, como está discutido na Secção 3.2.3.2. Como este ponto não foi o foco deste trabalho, a atual solução usa ficheiros temporários apenas para guardar resultados temporários, levando a desperdício de tempo e espaço em disco, para além de possíveis problemas devido ao número de ficheiros criados na diretoria. Agora, a solução mais aceitável, para se usufruir da computação em paralelo, será o lançamento da **Proxy** e do **Executor**, na própria máquina pessoal. Como visto na Tabela 4.2, a **Execução Local** corresponde ao cenário da Arquitetura Local (*LPC*) e o lançamento da **Proxy** e do **Executor**, na máquina pessoal corresponde ao cenário Arquitetura Distribuída (*DPC*), sendo os *overheads* **0,49 segundos** para a Arquitetura Local (*LPC*) e **0,31 segundos** para Arquitetura Distribuída (*DPC*).

Ainda no que toca aos objetivos delineados, a avaliação mostrou que mesmo perdendo algum desempenho, devido a *overheads* provocados pelas comunicações e pela distribuição de avaliações entre os **N Executores**, no final torna-se mais vantajosa a avaliação

em paralelo, a partir dos **0,3 segundos** por computação. Porém, também se verificou um aumento no número de avaliações no SID-PSM, proporcional ao aumento de número de execuções/CPU. O problema de otimização *bdvalue*, sendo o problema com o maior número de iterações utilizado, com **1423 iterações** na versão sequencial, foi onde mais se notou este aumento de avaliações (de **6011** para **9446** na versão em paralelo, Tabela 4.12). Tal é específico desta aplicação e destes problemas, onde o algoritmo de procura da melhor solução pode seguir outro caminho, dependendo da ordem de chegada das respostas, ao programa. Também na versão sequencial, a avaliação de uma iteração chega ao fim assim que se obtém uma solução melhor, ao contrário da versão em paralelo, onde são lançadas todas as avaliações paralelas possíveis.

Finalmente, observando agora os objetivos delineados no Capítulo 1, é seguro afirmar que todo o trabalho desenvolvido correspondeu às expectativas. No fim, conseguiu-se obter um sistema paralelo, com suporte a recursos remotos, capaz de gerir automaticamente estes últimos, instalar os seus componentes e a aplicação, de forma simples. Este trabalho, como também foi descrito nos objetivos, mesmo sendo uma solução o mais genérica possível (suporte a várias linguagens de programação e vários recursos remotos), apenas abrange um conjunto limitado de classes de problemas, neste caso aplicações em Octave, e um conjunto limitado de recursos remotos: para além do próprio PC, suporta máquinas acessíveis por SSH a que o utilizador tenha acesso, MV na nuvem Azure e *Clusters* geridos pelo escalonador OAR.

5.1 Trabalho futuro

Para trabalho futuro, os aspetos mais relevantes dizem respeito à reserva e lançamento dos recursos remotos e das funcionalidades Octave, presentes na Solução.

Começando pela reserva e lançamento dos recursos remotos:

- Um ponto inicial bastante importante seria conseguir que a automatização da instalação dos requisitos, utilizados pelos componentes do sistema, também estivesse disponível para máquinas Windows;
- Criar suporte para a Amazon AWS e para a Google Cloud, seria também algo bastante vantajoso. Da forma como o sistema foi realizado, seria até possível contornar o *vendor lock-in*, imposto pelos fornecedores da nuvem. Isto porque, como os componentes da Solução são independentes do fornecedor, não será necessário alterações nestes ou nas aplicações que usem esta solução. Outro potencial benefício era tornar possível configurações que suportassem múltiplas nuvens. Por exemplo, uma configuração onde a **Proxy** estaria alocada na Amazon AWS e vários **Executores**, estariam repartidos entre a Azure e a Google Cloud;

Para as funcionalidades Octave, sendo o desenho da sua arquitetura ainda muito inicial, pode-se discutir vários pontos que se poderiam melhorar e adicionar a esta. Porém,

destacamos apenas os seguintes aspectos:

- Melhorar e completar a execução local, ou seja, a distribuição da carga da aplicação do utilizador pelos núcleos do processador, da máquina pessoal;
- Criar um mecanismo que permitisse ao sistema adaptar, ao longo das execuções, o número de unidades de processamento, segundo as necessidades, em tempo real. Assim o utilizador final, teria menos uma configuração para realizar e ainda conseguiria obter um melhor rendimento, nas suas execuções;
- A criação de uma interface gráfica, que permitisse a configuração interativa do sistema. Este aspeto acabaria por melhorar a usabilidade do trabalho desenvolvido, para utilizadores comuns;
- Adaptar o protótipo desenvolvido ao MATLAB e, consequentemente, criar a interoperabilidade da Solução entre instâncias Octave e instâncias MATLAB, como apresenta o Dragonfly [42]. Assim, seria possível a utilização de instâncias MATLAB, a utilizadores que, necessitem obrigatoriamente destas. Outra possibilidade será permitir a um programa MATLAB tirar partido do paralelismo, sem necessitar de extensões ou licenças, interagindo com os recursos paralelos que executam as operações em Octave. Uma boa analogia para a sua utilização seria a arquitetura *master-worker* onde, neste caso, o *master* seria a instância MATLAB e os *workers* seriam as instâncias Octave;
- Adicionar suporte a mais linguagens de programação às funcionalidades apenas implementadas, de momento, para Octave. Teria de se implementar um componente cliente para cada linguagem de programação e um componente servido, no **Executor** para suportar a execução das funções nessas linguagens. Para o **Cliente**, pode-se explorar a possibilidade de usar a biblioteca *Cliente* em C existente e criar interfaces para as outras linguagens, recorrendo, por exemplo, no caso do Java ao *Java Native Interface* [60], ou à interface do Python para C.
- Melhorar a distribuição de avaliações entre os **Executores** para permitir melhorar a distribuição de carga, visto que agora é utilizado um algoritmo de distribuição simples, *Round-robin*;
- Melhorar o **Executor** para, no caso do Octave, evitar criar novas instâncias para cada avaliação. Parece possível criar as instâncias de início e ir passando os comandos a cada interpretador Octave, para avaliar cada função, sem necessitar de o voltar a executar.

A arquitetura desenvolvida pode, em princípio, acomodar estes novos desenvolvimentos sem grandes problemas.

Finalmente, o último ponto passa pela transformação do paradigma criado para este trabalho, cliente-servidor, para uma solução *serverless* comercial. Depois de todo o estudo, de como funciona este último paradigma, a sua utilização efetiva, não aconteceu. Para todos os requisitos que são necessários pré e pós-utilização do sistema desenvolvido, tornaria o desenho de toda a arquitetura bastante mais complexo, sem a garantia que haveriam melhorias.

BIBLIOGRAFIA

- [1] *SETI@home*. URL: <https://setiathome.berkeley.edu/> (acedido em 11/02/2021) (ver p. 1).
- [2] *Amazon Web Services (AWS) – Serviços de computação em nuvem*. URL: <https://aws.amazon.com/pt/> (acedido em 11/02/2021) (ver p. 2).
- [3] *About NIST*. URL: <https://www.nist.gov/about-nist> (acedido em 20/05/2021) (ver p. 2).
- [4] P. Mell e T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Rel. téc. (Acedido em 30/07/2021) (ver p. 2).
- [5] *Amazon Lambda - AWS*. URL: <https://aws.amazon.com/pt/lambda/> (acedido em 11/02/2021) (ver p. 3).
- [6] *Serverless Architectures*. URL: <https://martinfowler.com/articles/serverless.html> (acedido em 11/02/2021) (ver p. 3).
- [7] S. F. F. Tavares. “Contributions to the development of an integrated toolbox of solvers in Derivative-Free Optimization”. Tese de mestrado. Universidade Nova de Lisboa. (Acedido em 30/07/2021) (ver pp. 3, 6–8, 24–26, 48, 54, 55).
- [8] A. L. Custódio. *SID-PSM: A pattern search method guided by simplex derivatives for use in derivative-free optimization (version 1.3)*. (Acedido em 30/07/2021) (ver p. 3).
- [9] A. Fuggetta, G. P. Picco e G. Vigna. “Understanding code mobility”. Em: *IEEE Transactions on Software Engineering* 24.5 (), pp. 342–361. ISSN: 2326-3881. DOI: [10.1109/32.685258](https://doi.org/10.1109/32.685258). (Acedido em 30/07/2021) (ver p. 4).
- [10] A. L. Custódio. “Aplicações de Derivadas Simplicéticas em Métodos de Procura Directa”. Tese de doutoramento. Universidade Nova de Lisboa. (Acedido em 30/07/2021) (ver p. 7).
- [11] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201575949. (Acedido em 30/07/2021) (ver p. 10).

- [12] B. Schmidt et al. *Parallel Programming: Concepts and Practice*. Morgan Kaufmann. ISBN: 9780128498903. (Acedido em 30/07/2021) (ver pp. 10, 11, 13).
- [13] C. S. E. Project. *Computer Architecture, Copyright (C) 1991, 1992, 1993, 1994, 1995 by the Computational Science Education Project*. URL: <https://www.phy.ornl.gov/csep/ca/ca.html> (acedido em 30/07/2021) (ver pp. 10, 11).
- [14] F. Silva e R. Rocha. *Parallel and Distributed Programming Performance Metrics*. URL: https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_metrics.pdf (acedido em 30/07/2021) (ver pp. 11–14).
- [15] D. Marinescu. *Cloud Computing: Theory and Practice*. 1st. Morgan Kaufmann Publishers Inc. ISBN: 0124046274. (Acedido em 30/07/2021) (ver p. 15).
- [16] Google Docs. URL: <https://www.google.com/docs/about/> (acedido em 14/09/2021) (ver p. 15).
- [17] Google App Engine. URL: <https://cloud.google.com/appengine> (acedido em 14/09/2021) (ver p. 15).
- [18] What are microservices? URL: www.redhat.com/en/topics/microservices/what-are-microservices (acedido em 11/02/2021) (ver p. 16).
- [19] What is Function-as-a-Service (FaaS)? URL: <https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/> (acedido em 27/12/2020) (ver p. 16).
- [20] Client-Server Paradigm - an overview | ScienceDirect Topics. URL: <https://www.sciencedirect.com/topics/computer-science/client-server-paradigm> (acedido em 30/12/2020) (ver p. 16).
- [21] What is serverless computing? | Serverless definition. URL: <https://www.cloudflare.com/learning/serverless/what-is-serverless/> (acedido em 27/12/2020) (ver p. 17).
- [22] G. Adzic e R. Chatley. “Serverless Computing: Economic and Architectural Impact”. Em: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, pp. 884–889. ISBN: 9781450351058. DOI: [10.1145/3106237.3117767](https://doi.org/10.1145/3106237.3117767). URL: <https://doi.org/10.1145/3106237.3117767> (acedido em 30/07/2021) (ver pp. 17, 20).
- [23] A. P. Rajan. “Serverless architecture-a revolution in cloud computing”. Em: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. IEEE, pp. 88–93. DOI: [10.1109/ICoAC44903.2018.8939081](https://doi.org/10.1109/ICoAC44903.2018.8939081). (Acedido em 30/07/2021) (ver p. 18).

- [24] A. Suresh e A. Gandhi. “FnSched: An Efficient Scheduler for Serverless Functions”. Em: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC ’19. Davis, CA, USA: Association for Computing Machinery, pp. 19–24. ISBN: 9781450370387. DOI: [10.1145/3366623.3368136](https://doi.org/10.1145/3366623.3368136). URL: <https://doi.org/10.1145/3366623.3368136> (acedido em 30/07/2021) (ver p. 18).
- [25] A. Aytekin e M. Johansson. “Exploiting Serverless Runtimes for Large-Scale Optimization”. Em: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 499–501. DOI: [10.1109/CLOUD.2019.00090](https://doi.org/10.1109/CLOUD.2019.00090). (Acedido em 30/07/2021) (ver pp. 18, 19).
- [26] G. Kiar et al. “A Serverless Tool for Platform Agnostic Computational Experiment Management”. Em: *Frontiers in Neuroinformatics* 13 (), p. 12. ISSN: 1662-5196. DOI: [10.3389/fninf.2019.00012](https://doi.org/10.3389/fninf.2019.00012). URL: <https://www.frontiersin.org/article/10.3389/fninf.2019.00012> (acedido em 30/07/2021) (ver pp. 18, 20).
- [27] P.-L. Lions e B. Mercier. “Splitting Algorithms for the Sum of Two Nonlinear Operators”. Em: *SIAM Journal on Numerical Analysis* 16.6 (), pp. 964–979. ISSN: 0036-1429, 1095-7170. DOI: [10.1137/0716071](https://doi.org/10.1137/0716071). URL: <http://pubs.siam.org/doi/10.1137/0716071> (acedido em 11/02/2021) (ver p. 19).
- [28] T. Glatard et al. *Boutiques*. DOI: [10.5281/ZENODO.1098558](https://doi.org/10.5281/ZENODO.1098558). URL: <https://zenodo.org/record/1098558> (acedido em 30/12/2020) (ver p. 20).
- [29] *MindMup*. URL: <https://www.mindmup.com/> (acedido em 11/02/2021) (ver p. 20).
- [30] S. Malla e K. Christensen. “HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)”. Em: *Internet Technology Letters* 3.1 (), e137. ISSN: 2476-1508. DOI: [10.1002/itl2.137](https://doi.org/10.1002/itl2.137). URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/itl2.137> (acedido em 12/02/2020) (ver p. 20).
- [31] E. Jonas et al. “Cloud programming simplified: A berkeley view on serverless computing”. Em: *preprint arXiv:1902.03383* (). (Acedido em 30/07/2021) (ver p. 21).
- [32] *What is big data? | Cloud Big Data Solutions*. URL: <https://cloud.google.com/what-is-big-data> (acedido em 28/12/2020) (ver p. 22).
- [33] C. R. d. Junqueira. *A Internet das Coisas (IoT – Internet of Things)*. URL: <https://www.cncs.gov.pt/a-internet-das-coisas-iot-internet-of-things/> (acedido em 11/02/2021) (ver p. 22).
- [34] D. Black. *The convergence of HPC and BigData: What does it mean for HPC sysadmins?* URL: <https://insidehpc.com/2019/02/the-convergence-of-hpc-and-bigdata-what-does-it-mean-for-hpc-sysadmins/> (acedido em 28/12/2020) (ver p. 22).
- [35] *Master-Worker*. URL: <https://java-design-patterns.com/patterns/master-worker-pattern/> (acedido em 28/12/2020) (ver p. 23).

- [36] J. Dean e S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". Em: *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*. OSDI'04. San Francisco, CA, USA: USENIX Association, p. 10. (Acedido em 30/07/2021) (ver pp. 23, 24).
- [37] *MapReduce 101: What It Is & How to Get Started - Talend*. URL: <https://www.talend.com/resources/what-is-mapreduce/> (acedido em 28/12/2020) (ver p. 23).
- [38] *MathWorks - Makers of MATLAB and Simulink*. URL: <https://www.mathworks.com/> (acedido em 30/12/2020) (ver p. 24).
- [39] *GNU Octave*. URL: <https://www.gnu.org/software/octave/index> (acedido em 30/12/2020) (ver p. 24).
- [40] *Parallel Computing Toolbox*. URL: <https://www.mathworks.com/products/parallel-computing.html> (acedido em 30/12/2020) (ver pp. 24, 25).
- [41] *Octave Forge - The 'parallel' package*. URL: <https://octave.sourceforge.io/parallel/> (acedido em 30/12/2020) (ver p. 24).
- [42] I. Azzini, R. Muresano e M. Ratto. "Dragonfly: A multi-platform parallel toolbox for MATLAB/Octave". Em: *Computer Languages, Systems and Structures* 52 (), pp. 21–42. ISSN: 1477-8424. DOI: <https://doi.org/10.1016/j.cls.2017.10.002>. URL: <http://www.sciencedirect.com/science/article/pii/S1477842417300660> (acedido em 30/07/2021) (ver pp. 24, 26, 45, 72).
- [43] *SSH Protocol – Secure Remote Login and File Transfer*. URL: <https://www.ssh.com/ssh/protocol/> (acedido em 11/02/2021) (ver p. 32).
- [44] R. Fielding et al. *Hypertext transfer protocol–HTTP/1.1*. (Acedido em 30/07/2021) (ver p. 32).
- [45] *OAR*. URL: <http://oar.imag.fr/> (acedido em 20/04/2021) (ver pp. 35, 47).
- [46] J. Shim. *VM sizes - Azure Virtual Machines*. URL: <https://docs.microsoft.com/en-us/azure/virtual-machines/sizes> (acedido em 23/02/2021) (ver p. 36).
- [47] *Docker overview*. URL: <https://docs.docker.com/get-started/overview/> (acedido em 05/02/2021) (ver p. 37).
- [48] *SSH port forwarding - Example, command, server config*. URL: <https://www.ssh.com/ssh/tunneling/example> (acedido em 11/02/2021) (ver p. 38).
- [49] *Variable-length input argument list - MATLAB*. URL: <https://www.mathworks.com/help/matlab/ref/varargin.html> (acedido em 11/02/2021) (ver p. 40).
- [50] *Create Function Handle - MATLAB & Simulink*. URL: https://www.mathworks.com/help/matlab/matlab_prog/creating-a-function-handle.html (acedido em 11/02/2021) (ver p. 40).

BIBLIOGRAFIA

- [51] *Retrieve next available unread FevalFuture outputs - MATLAB fetchNext.* URL: <https://www.mathworks.com/help/parallel-computing/parallel.future.fetchnext.html> (acedido em 11/02/2021) (ver p. 40).
- [52] *Intel® Core™ i5-6600K Processor (6M Cache, up to 3.90 GHz) Product Specifications.* URL: <https://ark.intel.com/content/www/us/en/ark/products/88191/intel-core-i5-6600k-processor-6m-cache-up-to-3-90-ghz.html> (acedido em 28/01/2021) (ver p. 46).
- [53] *DI-Cluster Wiki.* URL: <https://cluster.di.fct.unl.pt/docs/> (acedido em 28/01/2021) (ver p. 47).
- [54] *Intel® Xeon® Processor E5-2609 v4 (20M Cache, 1.70 GHz) Product Specifications.* URL: <https://ark.intel.com/content/www/us/en/ark/products/92990/intel-xeon-processor-e5-2609-v4-20m-cache-1-70-ghz.html> (acedido em 28/01/2021) (ver p. 47).
- [55] M. Haarala. *Large-scale nonsmooth optimization: variable metric bundle method with limited memory.* University of Jyväskylä. (Acedido em 30/07/2021) (ver p. 55).
- [56] M. J. D. Powell. “On trust region methods for unconstrained minimization without derivatives”. Em: *Mathematical Programming* 97.3 (), pp. 605–623. ISSN: 0025-5610, 1436-4646. DOI: [10.1007/s10107-003-0430-6](https://doi.org/10.1007/s10107-003-0430-6). URL: <http://link.springer.com/10.1007/s10107-003-0430-6> (acedido em 30/07/2021) (ver p. 55).
- [57] J. Moré, B. Garbow e K. Hillstrom. “Testing Unconstrained Optimization Software”. Em: *ACM Transactions on Mathematical Software* 7.1 (), pp. 17–41. ISSN: 0098-3500, 1557-7295. DOI: [10.1145/355934.355936](https://doi.org/10.1145/355934.355936). URL: <https://doi.acm.org/10.1145/355934.355936> (acedido em 30/07/2021) (ver p. 55).
- [58] C. Bogani, M. G. Gasparo e A. Papini. “Generalized Pattern Search methods for a class of nonsmooth optimization problems with structure”. Em: *Journal of Computational and Applied Mathematics* 229.1 (), pp. 283–293. ISSN: 03770427. DOI: [10.1016/j.cam.2008.10.047](https://doi.org/10.1016/j.cam.2008.10.047). URL: <https://linkinghub.elsevier.com/retrieve/pii/S0377042708005773> (acedido em 30/07/2021) (ver p. 55).
- [59] C. Audet, V. Béchard e S. Le Digabel. “Nonsmooth optimization through Mesh Adaptive Direct Search and Variable Neighborhood Search”. Em: *Journal of Global Optimization* 41.2 (), pp. 299–318. DOI: [10.1007/s10898-007-9234-1](https://doi.org/10.1007/s10898-007-9234-1). URL: <http://dx.doi.org/doi:10.1007/s10898-007-9234-1> (acedido em 30/07/2021) (ver p. 66).
- [60] *Java Native Interface Specification Contents.* URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html> (acedido em 12/02/2021) (ver p. 72).

ANEXO 1 EXEMPLO DE UMA EXECUÇÃO DA SOLUÇÃO PROPOSTA

As Figuras presentes neste Anexo demonstram uma utilização simples da Solução Proposta. Esta utiliza as configurações dos recursos remotos para *Cluster*. Todas as etapas de uma execução estão numeradas de 1 a 30:

1. Uma possível utilização das funcionalidades *execute* e *next*, da Solução Proposta;
2. A utilização da funcionalidade *cancel*;
3. Preenchimento do ficheiro *config.properties*. Neste é identificado o IP e porta da **Proxy**, as pastas necessárias e o ambiente, para a execução da Solução Proposta;
4. Diretoria do ficheiro de compilação da ferramenta para Windows, *mainCompile.m*. Pode-se observar ainda o ficheiro de compilação UNIX, *unix_mainCompile.m*;
5. Acesso à diretoria *main* da Solução;
6. Comando de compilação do trabalho desenvolvido;
7. Pasta *remote*. Esta pasta guarda as informações da última execução realizada (número de unidades de processamento, *id* do projeto enviado e algumas configurações). Para, por exemplo, alterar o número de unidades de processamento (*workers*), é obrigatório apagar o seu conteúdo;
8. Diretoria do ficheiro de configuração para o *Cluster*;
9. As configurações disponíveis, no ficheiro referido anteriormente;
10. Diretoria do programa, de reserva dos recursos, que serão posteriormente utilizados;
11. Mudança de diretoria para o programa de reserva;
12. Execução do programa de reserva;
13. Primeira parte do *output* do programa, executado anteriormente;

ANEXO I. ANEXO 1 EXEMPLO DE UMA EXECUÇÃO DA SOLUÇÃO PROPOSTA

14. Segunda parte do *output* do programa;
15. Estado da reserva dos nós do *Cluster* utilizado. Esta janela não se deve fechar até não ser mais necessária a utilização da Solução Proposta;
16. Após a reserva é criado um ficheiro especial, *hosts.txt*. Cada linha deste ficheiro corresponde a um componente **Executor** reservado;
17. O conteúdo do ficheiro *hosts.txt*. Cada linha é composta por três parâmetros, separados pelo caractér ":" onde o primeiro é o IP ou nome da máquina que aloca o **Executor**; o segundo parâmetro é a porta do mesmo; o último é o número de *workers*, que este utilizará na execução;
18. Mudança de diretoria para a pasta do programa de lançamento dos componentes remotos;
19. Execução do programa de lançamento. Este contem um parâmetro, o nó que servirá de **Proxy**, que corresponde ao primeiro nó a que a reserva se conecta, neste caso o **nó 5**;
20. *Output* do lançamento dos componentes remotos;
21. Diretoria do exemplo, neste caso da *framework* SID-PSM;
22. Diretoria de ficheiro exemplo para a execução da *framework*;
23. Mudança de diretoria, no terminal, para a pasta do exemplo;
24. Execução do ficheiro exemplo, utilizando a Solução Proposta;
25. Diretoria do relatório, onde mostra o resultado da execução;
26. *Output* do relatório, da execução do ficheiro exemplo;
27. Cancelamento da reserva dos recursos remotos;
28. Diretoria para o ficheiro de configuração para utilizar a Microsoft Azure ou máquinas que não necessitam de reserva;
29. Propriedades de configuração do ficheiro, anteriormente referido;
30. Antes de qualquer utilização, deve-se de adicionar a diretoria *main* da Solução Proposta, ao projeto que utilizará as suas funcionalidades.

```

1
if oportunistic
    for ii = 1:pointsToEval
        %futures(ii) = parfeval(@eval_point, 1, x_mat(:,ii), f_eval);
        execute(ii, @eval_point, x_mat(:,ii), f_eval);
    end

.teration results.

2
if ordered % Force ordering on reception.
    lastIndex = 0;
    results(pointsToEval) = 0;
    mask_res = zeros(pointsToEval,1);
    endcycle = 0;
    for jj = 1:pointsToEval
        %[completedIdx,ftemp] = fetchNext(futures);
        [completedIdx, ftemp] = next();
        func_eval = func_eval + 1;
        func_iter = func_iter + 1;
        results(completedIdx) = ftemp;
        mask_res(completedIdx) = 1;
        while completedIdx == (lastIndex+1)
            actual_eval = actual_eval + 1;
            if isfinite(ftemp)
                f_vector(pointsToEval - completedIdx + 1) = ftemp;
                rotation_vec(pointsToEval - completedIdx + 1) = completedIdx;
                mask_f(pointsToEval - completedIdx + 1) = 1;
            if ftemp < f
                success = 1;
                xtemp = x_mat(:,completedIdx);
                xtempnorm = norm_vector(completedIdx);
                if order_option == 2
                    tmpindex = indexes(completedIdx);
                    if tmpindex % In case its 0, remains the same.
                        first_flag_cold = 1;
                        flag_cold = D(1:n, tmpindex);
                    end
                end
                endcycle = 1;
            break;
        end
    end
end
cancel();
%cancel(futures);
%futures(1:pointsToEval) = [];

```

Figura I.1: Utilização das funcionalidades da Solução Proposta.

```

3
config.properties
1 #the directories should have the full path
2 #local(*), remote
3 type=remote
4 ##### Proxy config #####
5 ip=127.0.0.1
6 port=8080
7 ##### Folder config #####
8 finished_folder=<tool-workspace>\ThesisTool\client\src\main\finished
9 project_folder=<tool-workspace>\ThesisTool\example\BoostSID_PSM_1
10 #octave(*), (java, jar, c future work)
11 environment=octave

4
client\src
> headers
> http
main
> finished
> local
> remote
  cancel.mex
  execute.m
  executionTool.mex
  mainCompile.m
  next.mex
  parseParams.m
  unix_mainCompile.m

5
PS [REDACTED]\ThesisTool> cd .\client\src\main\

6
PS [REDACTED]\ThesisTool\client\src\main> octave .\mainCompile.m

```

Figura I.2: Preenchimento do ficheiro principal de configurações e compilação da ferramenta.

ANEXO I. ANEXO 1 EXEMPLO DE UMA EXECUÇÃO DA SOLUÇÃO PROPOSTA

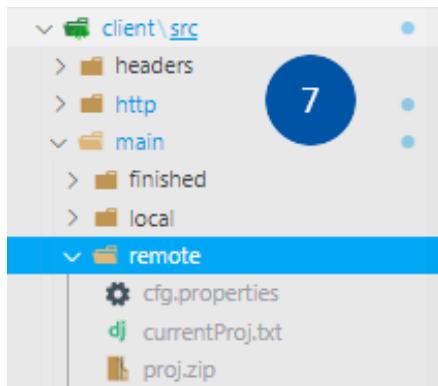


Figura I.3: Pasta *remote*, onde é guardada a última configuração utilizada pela Solução Proposta.

The figure consists of several parts:

- Panel 8:** Shows a file explorer with the 'remote.properties' file selected in the 'docker' folder. The file contains the following configuration:

```
remote_type=cluster
host=[REDACTED]@cluster.di.fct.unl.pt
#default ssh port: 22
ssh_port=12034
port=8080
#unix(*), windows
host_OS=unix
has_service=yes
```
- Panel 9:** A close-up view of the 'remote.properties' file content, identical to the one in Panel 8.
- Panel 10:** Shows the 'init' folder containing the 'remote.properties' file and other files like 'cluster_launch.c', 'cluster_reservation.c', etc.
- Panel 11:** A terminal window showing the command: PS [REDACTED]\ThesisTool> cd .\init\docker\|
- Panel 12:** The terminal window continues with the command: PS [REDACTED]\ThesisTool\init\docker> .\reservation.exe|

Figura I.4: Preenchimento do ficheiro das propriedades do *Cluster* utilizado e a inicialização da reserva dos recursos remotos.

13

```
PS [REDACTED]\ThesisTool\init\docker> .\reservation.exe
Starting cluster's nodes reservation: (Press Enter to continue at each step)

1-Open a terminal (search "cmd" on Windows or "terminal" on Unix Systems)

2-Insert the command:
ssh -p 12034 [REDACTED]@cluster.di.fct.unl.pt

Is it the first cluster usage?
(yes/no)

yes

1-Inside the SSH shell, insert the command:
echo -e "\n" | ssh-keygen -t rsa -q -f "/home/[REDACTED]/.ssh/id_rsa" -N ""

2-Inside the SSH shell, insert the command:
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

3-Inside the SSH shell, insert the reservation command, changing X value to the needed values:
If it is a specific reservation use the command:
[REDACTED] | oarsub -I -l /cluster=X/nodes=X/core=X

If it is a simple reservation use the command:
[REDACTED] | oarsub -I -l nodes=X

More options at cluster's wiki.

4-After the reservation, insert the command:
oarprint host

5-Insert the result of the previous command here, separated by commas (notice that the first one will be the proxy):
node5,node8

Inserted: node5,node8
```

6-Insert the number of cores for each one of the inserted nodes:
For node8:
Number of cores:
4

7-Let the terminal open please.

14

Figura I.5: *Output* do programa de reserva dos recursos remotos.

ANEXO I. ANEXO 1 EXEMPLO DE UMA EXECUÇÃO DA SOLUÇÃO PROPOSTA

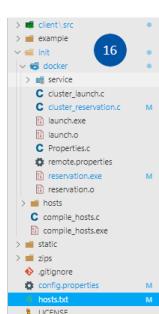
```

@frontend:~$ oarsub -I -l nodes=2
[ADMISSION RULE] Set default walltime to 3600.
[ADMISSION RULE] Modify resource description with type constraints
OAR_JOB_ID=■■■■■
Interactive mode: waiting...
Starting...

Connect to OAR job ■■■■■ via the node node5
@node5:~$ oarprint host
node5
node8

```

15



16

17

Figura I.6: Estado da reserva dos nós e a criação do ficheiro com as informações referentes ao componente **Executor** da Solução.

PS [REDACTED]\ThesisTool> cd .\init\docker\

18

PS [REDACTED]\ThesisTool\init\docker> .\launch.exe node5

19

```

PS [REDACTED]\ThesisTool\init\docker> .\launch.exe node5
ssh -t -p 12034 [REDACTED]@cluster.di.fct.unl.pt "ssh [REDACTED]@node8 'echo [REDACTED]@node8'"
[REDACTED]@node8
Connection to cluster.di.fct.unl.pt closed.

ssh -t -p 12034 [REDACTED]@cluster.di.fct.unl.pt "ssh [REDACTED]@node8 'cd ~/remote/ && docker load --input serv.tar && docker run -d -p 10000:8001 serv &'" 
The image serv:latest already exists, renaming the old one with ID [REDACTED] to empty string
Loaded image: serv:latest
[REDACTED]
Connection to cluster.di.fct.unl.pt closed.

ssh -t -L 8000:node5:8080 -p 12034 [REDACTED]@cluster.di.fct.unl.pt "ssh [REDACTED]@node5 'java -jar ~/remote/service/webservice-1.0-SNAPSHOT.jar 0.0.0.0 8080 & echo $!'" 
[REDACTED]
Feb 13, 2021 12:59:34 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [0.0.0.0:8080]
Feb 13, 2021 12:59:34 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Jersey app started with WADL available at http://0.0.0.0:8080/server/
Hit enter to stop it...

```

20

Figura I.7: Lançamento da Solução, no Cluster.

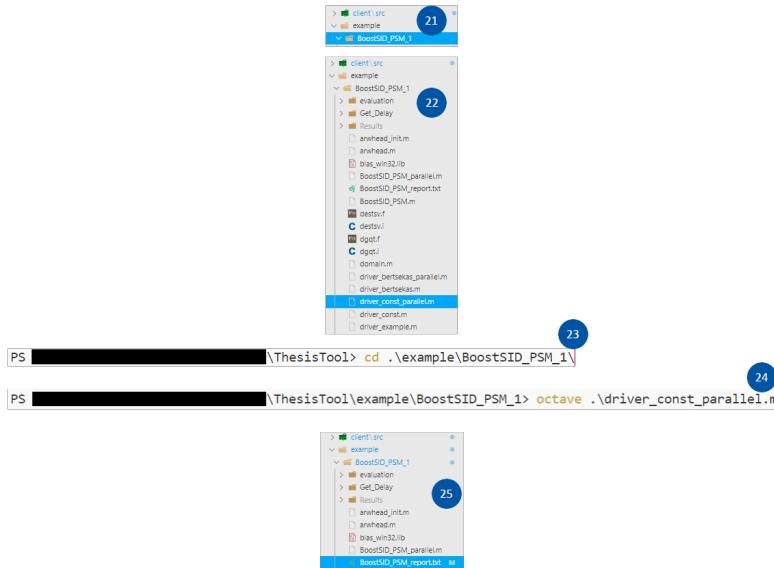


Figura I.8: Execução de um exemplo da *framework* SID-PSM.

Report:	iter	success	#fevals	x(1) +++++++	x(2) +++++++	f_value	alpha	active	search	poised
	0	--	--	-1.2000000e+00	+1.0000000e+00	+1.9360000e-01	+1.2000000e+00	1	--	--
	1	0	2	-1.2000000e+00	+1.0000000e+00	+1.9360000e-01	+1.2000000e+00	1	--	1
	2	0	3	-1.2000000e+00	+1.0000000e+00	+1.9360000e-01	+3.0000000e-01	1	--	1
	3	1	2	-9.0000000e-01	+1.0000000e+00	+3.6100000e-02	+3.0000000e-01	1	--	0
	4	1	1	-9.0000000e-01	+7.0000000e-01	+1.2100000e-02	+3.0000000e-01	1	--	1
	5	0	2	-9.0000000e-01	+7.0000000e-01	+1.2100000e-02	+1.5000000e-01	0	--	1
	6	1	2	-9.0000000e-01	+8.5000000e-01	+1.6000000e-03	+1.5000000e-01	0	--	1
	7	0	2	-9.0000000e-01	+8.5000000e-01	+1.6000000e-03	+7.5000000e-02	0	--	1
	8	0	3	-9.0000000e-01	+8.5000000e-01	+1.6000000e-03	+3.7500000e-02	0	--	1
	9	1	3	-9.3750000e-01	+8.5000000e-01	+8.35571289e-04	+3.7500000e-02	0	--	1
	10	1	1	-9.3750000e-01	+8.6755000e-01	+7.38525391e-05	+3.7500000e-02	0	--	1
	11	0	3	-9.3750000e-01	+8.6755000e-01	+7.38525391e-05	+1.8750000e-02	0	--	1
	12	0	3	-9.3750000e-01	+8.6755000e-01	+7.38525391e-05	+9.3750000e-03	0	--	1
	13	0	3	-9.3750000e-01	+8.7550000e-01	+7.38525391e-05	+4.6875000e-03	0	--	1
	14	1	3	-9.3750000e-01	+8.8212500e-01	+1.52587891e-05	+4.6875000e-03	0	--	1
	15	1	3	-9.3750000e-01	+8.7812500e-01	+6.18351562e-07	+4.6875000e-03	0	--	1
	16	0	3	-9.3750000e-01	+8.7812500e-01	+6.18351562e-07	+2.3437500e-03	0	--	1
	17	0	3	-9.3750000e-01	+8.7812500e-01	+6.18351562e-07	+1.17175750e-03	0	--	1
	18	0	3	-9.3750000e-01	+8.7812500e-01	+6.18351562e-07	+5.89397500e-04	0	--	1
	19	1	1	-9.369146462e-01	+8.7812500e-01	+1.00514038e-07	+5.89397500e-04	0	--	1
	20	0	3	-9.369146462e-01	+8.7812500e-01	+1.00514038e-07	+2.92968750e-04	0	--	1
	21	1	1	-9.37207831e-01	+8.7812500e-01	+5.38338133e-08	+2.92968750e-04	0	--	1
	22	0	2	-9.37207831e-01	+8.7812500e-01	+5.38338133e-08	+1.46484375e-04	0	--	1
	23	1	1	-9.37068047e-01	+8.7812500e-01	+1.80892767e-09	+1.46484375e-04	0	--	1
	24	0	2	-9.37068047e-01	+8.7812500e-01	+1.80892767e-09	+7.32421875e-05	0	--	1
	25	1	3	-9.37068047e-01	+8.78051750e-01	+9.43146921e-10	+7.32421875e-05	0	--	1
	26	0	2	-9.37068047e-01	+8.78051750e-01	+9.43146921e-10	+3.662193037e-05	0	--	1
	27	0	3	-9.37068047e-01	+8.78051750e-01	+9.43146921e-10	+1.83105469e-05	0	--	1
	28	1	1	-9.37042236e-01	+8.78051750e-01	+1.29971043e-11	+1.83105469e-05	0	--	1
	29	0	3	-9.37042236e-01	+8.78051750e-01	+1.29971043e-11	+9.15527344e-06	0	--	1

Figura I.9: Relatório final da execução da *framework* SID-PSM e a libertação dos recursos remotos.

ANEXO I. ANEXO 1 EXEMPLO DE UMA EXECUÇÃO DA SOLUÇÃO PROPOSTA

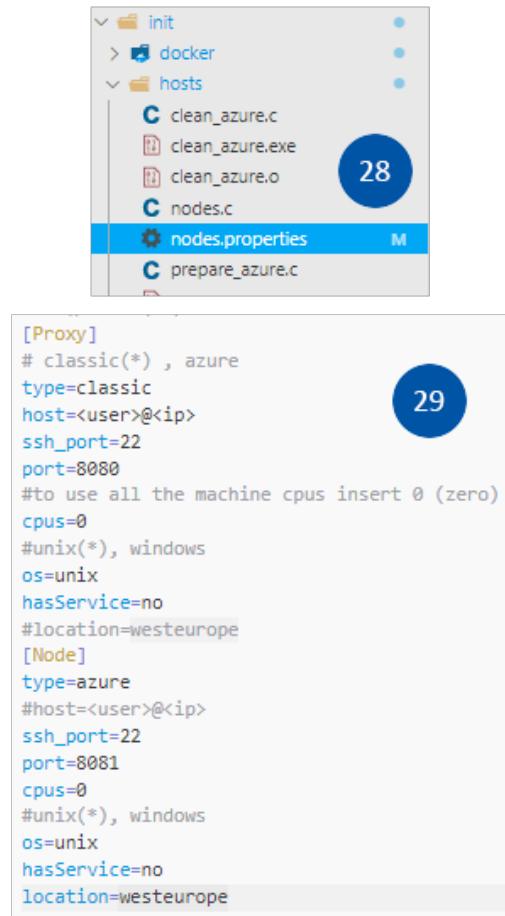


Figura I.10: Configuração para utilizar a Microsoft Azure ou máquinas remotas, sem necessidade de reserva, onde se conhecem IPs e portas, disponíveis, para lançar os componentes **Proxy** e **Executor**.

```
addpath('████████\ThesisTool\client\src\main');
```

A blue circle with the number 30 is positioned to the right of the code snippet.

Figura I.11: Antes de se utilizar a Solução, adiciona-se a pasta *main* ao projeto, com a função *addpath*.

ANEXO 2 DOCUMENTAÇÃO DO CÓDIGO-FONTE DA SOLUÇÃO PROPOSTA

Neste Anexo encontra-se a documentação de todo o código-fonte da Solução Proposta. Contudo, nos repositórios criados, constam alguns ficheiros que não serão documentados, isto porque não são atualmente utilizados pela Solução desenvolvida, mas, podem ser utilizados como um ponto de partida para o seu trabalho futuro.

Foram criados três repositórios, um para cada componente da Solução: **Cliente**, **Proxy** e **Executor**.

II.1 Cliente

O código do **Cliente** é composto, não só pelas suas funcionalidades, como também:

1. Por ficheiros de configuração;
2. Pelos exemplos utilizados para testar a Solução Proposta. Neste caso o SID-PSM é um exemplo bastante básico, de utilização das funcionalidades implementadas;
3. Por programas, que auxiliam o utilizador a lançar os recursos remotos, que serão utilizados;
4. Pelos componentes remotos, ficheiros *Jar*, compactados em ficheiros *.zip*.

II.1.1 Funcionalidades

As funcionalidades encontram-se na pasta *src*, da pasta *client*. É importante referir que a maioria dos ficheiros *.h* são utilizados para definir funções genéricas. Esta definição deveu-se ao facto de algumas funções em C serem dependentes do Sistema Operativo. Por exemplo, para obter o caminho de um ficheiro em Windows, a função utilizada é designada por *_fullpath*. Para UNIX, a função equivalente seria a *realpath*. Assim, definindo a função *GetRealPath*, comum aos dois Sistemas Operativos, evitou repetição de código

ANEXO II. ANEXO 2 DOCUMENTAÇÃO DO CÓDIGO-FONTE DA SOLUÇÃO PROPOSTA

ou até uma compreensão bastante mais complexa, das funções implementadas. O código foi dividido em quatro grupos:

1. *headers*, ficheiros *.h*, comuns a grande parte do código. Estes são:
 - a) *Directory.h*, onde se definem funções genéricas para obtenção do caminho de ficheiros;
 - b) *OS_header.h*, onde são definidas constantes para realizar execuções em linhas de comando (*mkdir*, *cp*, *cat*, entre outras);
2. *http*, onde se encontram implementadas, todas as operações com *sockets* e a criação do Cliente HTTP, que acede à **Proxy**. Estes são:
 - a) *client.c*, que contem:
 - i. A criação do *header* HTTP (*buildHeader*);
 - ii. A inicialização do *socket* que se conectará à **Proxy** (*start*);
 - iii. A função que insere o corpo da mensagem HTTP (*insertFuncionBody*);
 - iv. As funções de envio do pedido HTTP (*sendData*, *sendDataByIndex*);
 - v. As funções de receção do pedido HTTP (*parseHeader*, *readHttpStatus*, *receiveData* e *parseBody*).
 - b) *client.h*, o *header* que define as operações para a utilização dos *sockets*, de forma independente ao Sistema Operativo;
 - c) *operations.c*, que apresenta muitas operações, para além das utilizadas. As utilizadas são apenas:
 - i. A função utilizada pela funcionalidade *execute*, que envia o pedido HTTP, do **Cliente** para a **Proxy** (*sendFunction*);
 - ii. A função responsável pelo envio do projeto, quando necessário, para os Executores remotos (*upload*);
 - iii. A função utilizada pela funcionalidade *next*, que pede à **Proxy** o próximo resultado disponível, para ser consumido pelo **Cliente** (*fetchNext*);
 - iv. A função utilizada pela funcionalidade *cancel*, que descarta execuções que já não são mais necessárias, para determinada iteração (*cancel*);
 - v. A função especial, que permite à **Proxy**, através dos dados recebidos pelo **Cliente**, saber quais os **Executores**, que estão disponíveis (*config*).
 - d) *operations.h*, um *header* criado apenas para guardar diretorias dos ficheiros utilizados.
3. *main*, que contem os ficheiros principais da Solução Proposta. É esta a pasta, que contem os ficheiros de compilação das funcionalidades, disponibilizadas ao utilizador (*mainCompile.m*, que deve ser sempre executado, cada vez que se pretende

utilizar a Solução desenvolvida, em Windows. *unix_mainCompile.m*, cada vez que se pretende utilizar a Solução, em ambientes UNIX). Após a compilação do ficheiro correto, três ficheiros com a extensão *.mex*, aparecerão: *cancel.mex*, *executionTool.mex* e *next.mex*. Estes ficheiros correspondem à interface utilizada, que liga a linguagem Octave à linguagem C. São estes três ficheiros que permitem a utilização das funcionalidades *cancel*, *execute* e *next*, respetivamente. A pasta *main* também contém a pasta *remote*, onde são guardadas as informações necessárias, para utilizar as configurações, da execução anterior. Para renovar as configurações, deve-se de apagar o conteúdo desta pasta.

4. *tool*, constituída pelo código C que compõe o **Cliente** da Solução Proposta. Nestes ficheiros consta:
 - a) *cancel.c*, o ficheiro que implementa a interface interpretada pelo Octave, para a funcionalidade *cancel*;
 - b) *configParser.c*, que contem o *parsing* das configurações principais do projeto, proveniente do ficheiro *config.properties* (*getProperties*), descrito mais abaixo;
 - c) *Directory.c*, um ficheiro utilizado para processar diretórias da Solução Proposta. Neste processamento destaca-se, a obtenção de uma diretoria (*getDirectory*), obter o caminho absoluto da pasta *main* (*getMainFolder*) e ainda, uma função que altera o caminho recebido, conforme o Sistema Operativo (*processPath*);
 - d) *executionTool.c* é o ficheiro que implementa a interface interpretada pelo Octave, para a funcionalidade *execute*;
 - e) *fetch.c*, utilizado apenas para obter o resultado da funcionalidade *next*;
 - f) *next.c* é o ficheiro que implementa a interface interpretada pelo Octave, para a funcionalidade *next*;
 - g) *tool.c* contem funções a mais do que as utilizadas. É apenas utilizada, a função que cria todas as pastas necessárias, ao bom funcionamento da Solução Proposta (*createFolders*) e, é utilizada a função que transforma os valores recebidos do Octave, numa *String*, para ser posteriormente enviada para a **Proxy**, via HTTP (*directRemoteExec*).
 - h) *tool.h* um *header* que contem apenas as bibliotecas utilizadas e, a estrutura utilizada, para facilitar o processamento das matrizes, recebidas pelo Octave;
 - i) *utils.c* contem mais funções do que as realmente usadas. Nas últimas, destacam-se:
 - i. A função para criar um ficheiro *zip*, para enviar à **Proxy** (*zip*);
 - ii. A função para libertar os recursos da matriz, recebida do Octave (*freeMatrix*);
 - iii. A função utilizada para enviar o ficheiro *zip*, anteriormente criado (*addProj*).

ANEXO II. ANEXO 2 DOCUMENTAÇÃO DO CÓDIGO-FONTE DA SOLUÇÃO PROPOSTA

II.1.2 Ficheiros de configuração

A Solução desenvolvida contém apenas dois ficheiros de configuração gerais. Estes ficheiros são:

1. *config.properties*, Figura I.2, o ficheiro mais importante do sistema. Este conta com os parâmetros:
 - a) *type*, onde se define o tipo de execução. De momento está apenas disponível a opção *remote*;
 - b) *ip*, onde se define o IP que a **Proxy** vai utilizar;
 - c) *port*, a porta da **Proxy**;
 - d) *finished_folder*, um campo que atualmente não é utilizado. Servia anteriormente para uma versão simples, da execução local. Não foi removida, para trabalho futuro;
 - e) *project_folder*, o campo onde o utilizador deve inserir a diretoria do projeto, que vai utilizar a Solução Proposta;
 - f) *environment*, que define a linguagem de programação utilizada. Até ao presente é suportado apenas o Octave.
2. *hosts.txt*, o ficheiro gerado quando são reservados os recursos remotos. Cada linha é composta pelo IP de uma máquina, que será tratada como o componente **Executor**, a respetiva porta e número de processadores, que serão utilizados.

II.1.3 Exemplos

Nos exemplos, destaca-se apenas a *framework* SID-PSM com a configuração da Solução Proposta e respetivos problemas de otimização, alguns utilizados para realizar a avaliação, descrita no Capítulo 4.

II.1.4 Programas de lançamento dos recursos

Os programas encontram-se na pasta *remoteResources*, presente do projeto da Solução Proposta. Esta pasta é dividida em três subpastas:

1. *azure*, que contém todos os programas e ficheiros desenvolvidos para a reserva e lançamento da Solução na Microsoft Azure:
 - a) *clean_azure.c*, o programa que remove todas as definições sobre as MV utilizadas pela Solução na Azure;
 - b) *compile_azure.c*, o programa auxiliar para compilar todos os outros programas desenvolvidos;

- c) *installProxy.sh*, um ficheiro que instala todos os requisitos necessários para a utilização da **Proxy**;
 - d) *installService.sh*, um ficheiro que instala todos os requisitos necessários para a utilização do **Executor**;
 - e) *nodes.c*, o ficheiro que faz *parsing* da configuração dos recursos que serão utilizados (*nodes.properties*);
 - f) *nodes.properties*, o ficheiro de configuração dos recursos remotos Azure. Para definir a **Proxy**, deve se inserir, no início da configuração, "[Proxy]". Para definir um **Executor**, inicia-se com "[Node]";
 - g) *prepare_azure.c*, ficheiro utilizados para reservar os recursos e lançar os componentes, da Solução Proposta;
 - h) *shutdown_azure.c*, ficheiro utilizados para desativar os recursos anteriormente reservados. Este programa não remove as informações dos recursos, ou seja, quando necessário, estes podem-se voltar a utilizar.
2. *cluster*, que guarda os programas e ficheiros para a utilização do *Cluster DI*. Este contém:
- a) *service*, a pasta que guarda a imagem Docker utilizada pelo **Executor**, visto que este não executa nativamente nos nós do *Cluster*;
 - b) *cluster_launch.c*, o programa de lançamento dos recursos no *Cluster*;
 - c) *cluster_reservation.c*, o programa para a reserva dos recursos remotos, que serão utilizados;
 - d) *Properties.c*, utilizado para realizar o *parsing* das configurações do *Cluster*, presentes no ficheiro *remote.properties*;
 - e) *remote.properties*, utilizado para configurar o *Cluster*. A ligação aos recursos remotos, é feita por SSH.
3. *hosts*, a pasta correspondente às máquinas que não necessitam de reserva para utilização. Dentro desta pasta destacam-se os ficheiros:
- a) *compile_hosts.c*, o programa auxiliar para compilar todos os outros programas desenvolvidos;
 - b) *installProxy.sh*, um ficheiro que instala todos os requisitos necessários para a utilização da **Proxy**;
 - c) *installService.sh*, um ficheiro que instala todos os requisitos necessários para a utilização do **Executor**;
 - d) *nodes.c*, o ficheiro que faz *parsing* da configuração dos recursos que serão utilizados (*nodes.properties*);

ANEXO II. ANEXO 2 DOCUMENTAÇÃO DO CÓDIGO-FONTE DA SOLUÇÃO PROPOSTA

- e) *nodes.properties*, o ficheiro de configuração dos recursos remotos das máquinas remotas. A sua utilização é semelhante à anteriormente definida para a pasta *azure*;
- f) *prepare_hosts.c*, ficheiro utilizados para reservar os recursos e lançar os componentes, da Solução Proposta;
- g) *shutdown_hosts.c*, com o comportamento semelhante ao *shutdown_azure.c*, definido acima.

II.1.5 Componentes Java

Finalmente, na pasta *zips* encontram-se os ficheiros *Jar*, dos componentes remotos da Solução Proposta. Estes são usados nos programas que envolvem a reserva e o lançamento destes componentes, nos recursos remotos.

II.2 Proxy

O código da **Proxy**, apresenta as seguintes classes:

1. *FuturesSingleton.java*, a classe principal da lógica, deste componente. É por esta que passam todas as operações importantes, da Solução Proposta:
 - a) O envio do pedido do **Cliente**, para o **Executor** (*add*);
 - b) O acesso à estrutura de dados, com os resultados já preparados, consumida, pelo **Cliente** (*next*);
 - c) A configuração do *socket TCP*, utilizado como meio de comunicação interno, entre este componente e o **Executor** (classe *TCPServer*);
 - d) O envio da funcionalidade especial do cancelamento, da iteração (*cancel*).
2. *Host.java*, a classe utilizada para aglomerar dois objetos, utilizados simultaneamente;
3. *HostCollection.java*, a classe que organiza todos os objetos *Host*, anteriormente descritos;
4. *HostFileSingleton.java*, a classe que processa e organiza os **Executores**, provenientes do **Cliente**, que podem ser utilizados pela **Proxy**;
5. *Proxy.java*, a classe que contem todo o código da biblioteca REST, utilizada neste projeto;
6. *ProxyClient.java*, a classe que opera como Cliente HTTP, do **Executor**. É responsável pela comunicação indireta entre o **Cliente** e o **Executor**;

7. *ProxyStart.java*, a classe que inicializa as funcionalidades REST da **Proxy**;
8. *StructureEntry.java*, a classe utilizada para tradução de ficheiros JSON. Esta é a responsável pela organização dos resultados, provenientes do **Executor**.

II.3 Executor

Este componente, é composto por três *packages*:

1. *srv*, que contem as classes principais da biblioteca REST:
 - a) *Main.java*, a classe para inicializar as funcionalidades REST do **Executor**;
 - b) *Resources.java*, a classe que contem todas as anotações e *endpoints* das funcionalidades REST;
2. *srv.res*, onde estão definidas todas as classes dos *endpoints*, disponibilizados pelo componente:
 - a) *Config.java*, a classe que configura o número de *threads*, que serão utilizadas pela *Thread Pool*;
 - b) *Reset.java*, a classe que realiza a funcionalidade *cancel*, no componente **Executor**;
 - c) *SendFuncion.java*, a classe de onde se recebe os pedidos que, posteriormente, vão ser executados por uma *thread*, da *Thread Pool*;
 - d) *Upload.java*, a classe responsável por descompactar o projeto enviado pelo **Cliente**, que será utilizado.
3. *srv.res.common*, onde se encontram algumas classes que são comuns a vários *endpoints*. As que se destacam são:
 - a) *ProcessStructure.java*, a classe principal desta componente. Esta tem várias funcionalidades como:
 - i. Executar um pedido numa *thread* (*newProcess*);
 - ii. Atualizar o controlador de iterações, proveniente da funcionalidade *cancel* (*reset*);
 - iii. *StructureEntry*, uma classe embutida utilizada para formatar o resultado obtido de cada *thread*, em JSON, para posteriormente ser enviada, via TCP, para a **Proxy**;
 - b) *TCPClient.java*, a classe que acede ao *socket* TCP, presente na **Proxy** e que, envia os resultados obtidos pela *Thread Pool*.

