FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Optimização do tempo de processamento de aplicações em clusters em ambiente multi-utilizador

Belmiro Daniel Rodrigues Moreira

Dissertação

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Jorge Manuel Gomes Barbosa (Prof. Auxiliar da FEUP)

3 de Março de 2009

Optimização do tempo de processamento de aplicações em clusters em ambiente multi-utilizador

Belmiro Daniel Rodrigues Moreira

Dissertação

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo júri:

Presidente: António Augusto de Sousa (Prof. Associado da FEUP)

Arguente: João Luís Ferreira Sobral (Prof. Auxiliar da Universidade do Minho)

Vogal: Jorge Manuel Gomes Barbosa (Prof. Auxiliar da FEUP)

19 de Março de 2009

Resumo

Este trabalho aborda o problema do escalonamento dinâmico de aplicações, num cluster, em ambiente multi-utilizador.

O problema do escalonamento de aplicações é conhecido por ser, de uma forma geral, NP-completo. Assim, a utilização de um método que permita o escalonamento eficiente das aplicações é fundamental para se atingir um elevado desempenho.

Neste trabalho é proposto um método para o escalonamento de aplicações, que utiliza uma abordagem dinâmica por não se conhecer a frequência de chegada de uma nova aplicação. O objectivo é minimizar o *makespan* de um conjunto de aplicações.

Considera-se que cada aplicação é constituída por um conjunto de tarefas dependentes, com custos de comunicação, representada através de um DAG.

Os algoritmos utilizados para o escalonamento de aplicações constituídas por tarefas dependentes, num *cluster* heterogéneo, consideram apenas o escalonamento de uma aplicação e, para além disso, são estáticos. Os algoritmos de escalonamento dinâmico, por outro lado, consideram aplicações constituídas por uma única tarefa independente, atribuindo a cada uma, um processador. O método de escalonamento desenvolvido destaca-se dos restantes por considerar o escalonamento dinâmico de um conjunto de aplicações, sem efectuar uma reserva fixa de recursos a cada uma delas.

O método de escalonamento desenvolvido é constituído por duas partes: a estratégia e o algoritmo de escalonamento. A estratégia define o momento em que o algoritmo de escalonamento é invocado. A estratégia utilizada é do tipo *batch mode*. O algoritmo de escalonamento é estático, sendo necessário conhecer os tempos de execução e comunicação de cada tarefa.

Para a avaliação do método, consideram-se dois algoritmos de escalonamento, o HEFT e o HPTS. Estes algoritmos foram seleccionados por terem sido apresentados como boas soluções para o escalonamento de DAGs em máquinas heterogéneas. Propõe-se também uma optimização ao HPTS que se designou por iHPTS.

Os resultados mostram que o método desenvolvido apresenta um melhor desempenho quando comparado com as abordagens comuns, que fixam um número de processadores por aplicação ou atribuem a cada uma delas apenas um processador. Estes resultados foram obtidos tanto num *cluster* homogéneo como num heterogéneo.

Abstract

This thesis addresses the problem of scheduling dynamically multi-user and independent jobs on clusters.

The scheduling problem is known as NP-complete in the general case. Therefore, an efficient application scheduling method is critical for achieving high performance.

In this work it is presented a dynamic scheduling method, because the arrival rate of applications is not known a priori. The aim is to minimize the completion time, *makespan*, of batch of jobs.

It is assumed that one application is represented by a directed acyclic graph, DAG, of dependent tasks, with inter-tasks communication costs.

The algorithms proposed for scheduling DAGs in heterogeneous systems consider only one DAG (or job) to schedule and therefore they are all static approaches. Otherwise, on dynamic scheduling for heterogeneous clusters consider exclusively independent tasks and assign one tasks to one processor.

The dynamic method consists in a scheduling strategy and a scheduling algorithm. The scheduling strategy defines the instants when the scheduling algorithm is called. In this research it is considered a batch mode strategy. The scheduling algorithm is static, therefore, it is assumed that the task computation and communications sizes can be known or predicted before task scheduling.

Two scheduling algorithms are considered, HEFT and HPTS, in method evaluation, because they are presented as best solutions for DAG scheduling in heterogeneous systems. Also, is proposed a HPTS optimization, called iHPTS.

The results show the behavior of the scheduling method for the short completion time of a batch of jobs. These results show better performance when compared to the common schedulers strategies that fix the number of processors per job or assign one processor per job.

Agradecimentos

Queria aproveitar este espaço para expressar o meu profundo agradecimento e respeito ao Professor Doutor Jorge Barbosa, orientador do trabalho, por todas as suas sugestões, ensinamentos e disponibilidade que sempre manifestou durante todo este período;

Agradeço aos meus pais e irmão pelo seu carinho, confiança e apoio incansável em todos os momentos;

Finalmente, um agradecimento muito especial à Catarina, por todo seu amor e paciência ao longo destes anos.

Belmiro Moreira

Conteúdo

1 Introdução **1**
- 1.1 Contexto/Enquadramento . . . 1
- 1.2 Motivação e Objectivos . . . 2
- 1.3 Estrutura da Dissertação . . . 4

2 Revisão Bibliográfica **5**
- 2.1 Definição do problema . . . 5
  - 2.1.1 Problema NP-completo . . . 7
  - 2.1.2 Heterogeneidade de um sistema . . . 8
- 2.2 Trabalho relacionado . . . 9
  - 2.2.1 Algoritmos estáticos . . . 10
  - 2.2.2 Algoritmos dinâmicos . . . 15
- 2.3 Resumo e Conclusões . . . 16

3 Método de escalonamento **18**
- 3.1 Estratégia de escalonamento . . . 18
- 3.2 Algoritmo de escalonamento . . . 21
  - 3.2.1 Algoritmo HEFT . . . 22
  - 3.2.2 Algoritmo HPTS . . . 24
  - 3.2.3 Algoritmo iHPTS . . . 28
- 3.3 Tarefas não-paralelas vs paralelas . . . 29

4 Implementação **33**
- 4.1 Ferramenta jScheduler . . . 33
  - 4.1.1 Requisitos Funcionais . . . 34
  - 4.1.2 Diagrama de Classes . . . 36
- 4.2 Ferramenta jEditor . . . 38
  - 4.2.1 Requisitos Funcionais . . . 38
  - 4.2.2 Diagrama de Classes . . . 39

5 Resultados e Discussão **42**
- 5.1 Sistema de Computação . . . 43
  - 5.1.1 Heterogeneidade . . . 44
- 5.2 Aplicações . . . 45
- 5.3 Resultados Experimentais . . . 46
  - 5.3.1 Sistema de Computação Homogéneo . . . 47
  - 5.3.2 Sistema de Computação Heterogéneo . . . 48

6 Conclusões e Trabalho Futuro **53**
6.1 Satisfação dos Objectivos 53
6.2 Trabalho Futuro 55

Referências **57**

A Manual da Ferramenta jScheduler **58**
A.1 Modo de Instalação 58
A.2 Modo de Utilização 58
A.2.1 Submeter uma aplicação 59

B Manual da Ferramenta jEditor **60**
B.1 Modo de Instalação 60
B.2 Modo de Utilização 60
B.2.1 Adicionar tarefas 61
B.2.2 Adicionar aresta 61
B.2.3 Remover tarefa 62
B.2.4 Remover aresta 62
B.2.5 Mover tarefa 63
B.2.6 Seleccionar objectos 63
B.2.7 Submeter uma aplicação 63
B.2.8 Outras funcionalidades 64

C Descrição dos métodos das classes das Ferramentas **65**

Lista de Figuras

2.1 Aplicação com 10 tarefas, representada através de um DAG 6
2.2 Classificação do tipo de escalonamento de aplicações 10
2.3 Classificação dos algoritmos estáticos de escalonamento 11
2.4 Caminho crítico da aplicação definida em 2.1 13

3.1 Exemplo do DAG onde são inseridos os jobs para serem escalonados . . . 19
3.2 Exemplo do modo de funcionamento da estratégia de escalonamento . . . 20
3.3 Mapa do escalonamento efectuado pelo HEFT de uma aplicação num cluster homogéneo com 10 processadores 24
3.4 Mapa do escalonamento efectuado pelo HPTS de uma aplicação num cluster homogéneo com 10 processadores 28
3.5 Carga de cada processador do cluster - Usando o algoritmo HEFT 31
3.6 Carga de cada processador do cluster - Usando o algoritmo HPTS 31

4.1 Diagrama de actividades do caso de utilização - Submeter um job 35
4.2 Diagrama de classes da ferramenta jScheduler 36
4.3 Diagrama de classes da ferramenta jEditor 39
4.4 Diagrama de classes da ferramenta jEditor 40

5.1 Tempo global de escalonamento num cluster homogéneo em 10 processadores 47
5.2 Tempo global de escalonamento num cluster homogéneo em 20 processadores 48
5.3 Tempo global de escalonamento num cluster com baixa heterogeneidade em 10 processadores 49
5.4 Tempo global de escalonamento num cluster com baixa heterogeneidade em 20 processadores 50
5.5 Tempo global de escalonamento num cluster com alta heterogeneidade em 10 processadores 51
5.6 Tempo global de escalonamento num cluster com alta heterogeneidade em 20 processadores 51

B.1 Janela principal do jEditor 61
B.2 Definição dos atributos de uma tarefa 62
B.3 Exibição de uma aresta entre duas tarefas 62
B.4 Propriedades de uma tarefa do DAG 63
B.5 Janela de diálogo que permite o envio de aplicações 64

Lista de Tabelas

2.1 Tempos de execução das tarefas num cluster com 7 processadores, referido em [ASM+00] . . . 9
2.2 Os valores do t-level e b-level das tarefas da aplicação da Figura 2.1 . . . 14

5.1 Capacidade de processamento relativa e número de processadores dos clusters C1 e C2 . . . 43
5.2 Capacidade de processamento relativa e número de processadores dos clusters C3 e C4 . . . 43
5.3 Tempos de execução das tarefas de uma aplicação em cada um dos processadores do cluster C1 . . . 44
5.4 Tempos de execução das tarefas de uma aplicação em cada um dos processadores do cluster C3 . . . 45
5.5 Custos relativos de computação e comunicação das operações algébricas consideradas . . . 46
5.6 Média e Desvio Padrão dos escalonamentos em relação ao algoritmo utilizado, num cluster homogéneo . . . 48
5.7 Média e Desvio Padrão dos escalonamentos em relação ao algoritmo utilizado, num cluster pouco heterogéneo . . . 49
5.8 Média e Desvio Padrão dos escalonamentos em relação ao algoritmo utilizado, num cluster muito heterogéneo . . . 52

Abreviaturas e Símbolos

HPTS	Heterogeneous Parallel Tasks Scheduler
iHPTS	improved Heterogeneous Parallel Tasks Scheduler
HEFT	Heterogeneous Earliest Finish Time
DAG	Directed Acyclic Graph
EST	Earliest Start Time
EFT	Earliest Finish Time
QoS	Quality of Service
ETC	Expected Time to Compute
CCR	Comunications to Computation Ratio
XML	Extensible Markup Language
MPI	Message Passing Interface

Capítulo 1

Introdução

Neste capítulo descreve-se o contexto e enquadramento do trabalho, bem como os seus objectivos e motivação. No final é apresentada a sua estrutura.

1.1 Contexto/Enquadramento

O interesse por problemas cada vez mais complexos cria a necessidade de computadores cada vez mais potentes para os resolver. Contudo, limitações físicas e económicas tem restringido o aumento da velocidade dos computadores sequenciais, levando a adopção, cada vez mais frequente, de novas abordagens, nomeadamente a utilização de sistemas computacionais que permitam a execução de aplicações paralelas.

Um *cluster* é um sistema de computação de memória distribuída que se caracteriza por ser formado por um conjunto de máquinas individuais, ligadas entre si através de uma rede de comunicações. Estes sistemas dizem-se heterogéneos quando os componentes que os constituem têm capacidades diferentes.

Os *clusters* são essencialmente usados quando se pretende ter um sistema de elevado desempenho e/ou disponibilidade. Estes, são usados para a execução de aplicações computacionalmente intensivas, que podem ser executadas de forma paralela, como problemas relacionados com a análise de modelos climáticos, mecânica quântica, dinâmica de fluidos, simulação de experiências nucleares, etc. Dada a natureza dos problemas, são essencialmente instituições universitárias, laboratórios de pesquisa e agências militares, os maiores utilizadores de deste tipo de sistemas. No entanto, começam também a vulgarizar-se na indústria e banca, como uma forma de aumentar a eficiência e competitividade das suas estratégias e produtos. Muitos dos super computadores que pertencem ao TOP500[1], são *clusters*.

[1] Projecto que lista os 500 computadores mais potentes do mundo.

Neste trabalho, considera-se um *cluster* como um conjunto P de processadores, que podem ter capacidades diferentes, interligados através de uma rede de comunicações dedicada. Quando a capacidade de todos os processadores é igual diz-se que o *cluster* é homogéneo, pelo contrário, se isto não se verificar diz-se que o *cluster* é heterogéneo.

Uma aplicação, de uma forma geral, é constituída por um conjunto de tarefas dependentes, isto é, por um conjunto de sub problemas com uma relação de precedência entre eles. Estas aplicações podem ser representadas através de um DAG, onde cada nó representa uma tarefa (sub problema) e as arestas representam as relações de precedência e custos de comunicação.

O desempenho de uma aplicação num *cluster* depende da eficiência do escalonamento das tarefas que a constituem.

O problema de escalonamento de tarefas consiste em alocar recursos (processadores) às tarefas e em estabelecer a sua ordem de execução. Este problema é conhecido por ser NP-completo, com excepção de alguns, poucos, casos particulares. Devido à sua importância no desempenho das aplicações e aproveitamento dos recursos disponíveis, este problema tem vindo a ser amplamente estudado, tendo sido propostas várias técnicas de resolução, nomeadamente, heurísticas e métodos de pesquisa.

O escalonamento de tarefas pode ser classificado em dois tipos: estático e dinâmico. O escalonamento estático caracteriza-se por ser efectuado em tempo de compilação ou na inicialização. Este, baseia-se no conhecimento prévio da estrutura da aplicação e das características das tarefas que a constituem, nomeadamente, o tempo de execução de cada uma delas em cada um dos diferentes recursos, e os seus custos de comunicação. A utilização do escalonamento estático tem, geralmente, como objectivo, a minimização do tempo de execução global das tarefas, ou seja, a minimização do *makespan*.

Em contraste, recorre-se ao escalonamento dinâmico quando não se conhece as características das aplicações e/ou a sua frequência de chegada. Este tipo de escalonamento tem, geralmente, como objectivo, minimizar alguma medida de desempenho relacionada com a qualidade do serviço (QoS). O desconhecimento ou impossibilidade de previsão do tempo de execução das tarefas, remete a outro problema conhecido como *load balancing*. Por outro lado, o desconhecimento da frequência de chegada das aplicações a escalonar, remete a outro tipo de abordagem dinâmica onde é necessário utilizar uma estratégia que defina o momento em que as novas aplicações são escalonadas. É esta segunda abordagem que vai ser discutida neste trabalho.

1.2 Motivação e Objectivos

Os algoritmos utilizados para o escalonamento de aplicações constituídas por tarefas dependentes, num *cluster* heterogéneo, consideram apenas o escalonamento de uma

aplicação e, além disso, são estáticos [BMNM05, THW02]. Os algoritmos de escalonamento dinâmico, por outro lado, consideram aplicações constituídas por apenas uma tarefa independente, atribuindo a cada uma, um processador.

A motivação para este trabalho foi a criação de um método dinâmico para o escalonamento de múltiplos e independentes DAGs, num *cluster*, tanto homogéneo como heterogéneo. A abordagem dinâmica que se considera, advém do desconhecimento da frequência de chegada de cada uma das aplicações. As tarefas são reescalonadas quando uma nova aplicação é submetida. Contudo, em cada aplicação, o tempo de execução e comunicação das suas tarefas é conhecido.

O método desenvolvido pode ser divido em duas partes: a estratégia e o algoritmo de escalonamento. A estratégia define o instante em que o algoritmo de escalonamento é invocado. A estratégia considerada é do tipo *batch mode*. O algoritmo de escalonamento faz a atribuição dos processadores a cada tarefa e estabelece a ordem de execução. Este algoritmo é estático.

Neste trabalho consideraram-se dois algoritmos de escalonamento, o HEFT [THW02] e o HPTS [BMNM05]. Estes dois algoritmos foram escolhidos, por terem sido apresentados como tendo um bom desempenho no escalonamento de DAGs em *clusters* heterogéneos. A principal característica diferenciadora destes dois algoritmos, baseia-se no facto do HPTS considerar que uma tarefa pode executar de forma paralela, ou seja, a cada tarefa pode ser atribuído mais do que um processador. É também proposta uma optimização ao algoritmo HPTS, que se designou por iHPTS. Ambos os algoritmos usam uma heurística do tipo *list scheduling*.

O objectivo deste trabalho, é melhorar o desempenho de *clusters*, num ambiente multi-utilizador, onde múltiplas aplicações independentes são submetidas, cada uma composta por um conjunto de tarefas dependentes. O desempenho é avaliado pelo menor tempo de execução global de um conjunto de aplicações, como em [SZI07].

Um exemplo onde o escalonamento de um conjunto de aplicações pode ocorrer é na análise e processamento de sequências de imagens para identificação e seguimento de objectos. Os objectos podem sofrer alterações ao longo de cada *frame* sendo o problema conhecido como "*deformable object tracking*". Este problema é utilizado em aplicações de vídeo vigilância, processamento de imagem microscópica, análise biomédica, etc. Neste contexto, pode-se considerar ambiente multi-utilizador, o conjunto dos vários instrumentos de aquisição de imagens.

A abordagem comum no escalonamento de aplicações consiste no utilizador fixar o número de processadores para cada aplicação, estando-lhe, estes, exclusivamente reservados. Cada utilizador tenta alocar para as suas aplicações a maior capacidade possível, não existindo desta forma uma gestão global. Pelo contrário, o método desenvolvido não efectua qualquer reserva de processadores por aplicação ou utilizador. Os resultados apresentados comparam as duas abordagens.

Neste trabalho também se tem como objectivo a implementação do método, utilizando uma linguagem de programação orientada a objectos, de forma a poder ser executado e testado num *cluster*, mostrando a sua aplicabilidade.

1.3 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 5 capítulos. No capítulo 2 é feita a apresentação do problema e descrito o estado da arte. No capítulo 3 é elaborada a análise da estrutura e funcionamento do método de escalonamento que foi desenvolvido. Inicialmente, é analisada a estratégia que é utilizada. Seguidamente, são descritos os dois algoritmos estáticos de escalonamento que foram considerados no trabalho, o HEFT e o HPTS. Por fim, é apresentada um proposta de optimização do HPTS, designada por iHPTS. É também abordado o modelo computacional com que cada algoritmo foi apresentado. No capítulo 4 é descrita a implementação do método de escalonamento, através da análise das ferramentas que foram desenvolvidas, nomeadamente o *jScheduler* e o *jEditor*. No capítulo 5 são analisados os resultados obtidos através do novo método, utilizando os algoritmos de escalonamento HEFT, HPTS e iHPTS e comparados com outras abordagens de escalonamento. Finalmente, no capítulo 6 são apresentas as conclusões e o trabalho futuro.

Capítulo 2

Revisão Bibliográfica

Neste capítulo é apresentado o problema do escalonamento dinâmico de aplicações constituídas por um conjunto de tarefas dependentes, num *cluster*. Seguidamente, é apresentado o estado da arte, através da análise das principais abordagens no escalonamento de tarefas em *clusters*.

2.1 Definição do problema

Com este trabalho pretende-se propor um método para optimizar o escalonamento de aplicações, num *cluster*, em ambiente multi-utilizador.

Considera-se um *cluster* como um conjunto de processadores P, ligados através de uma rede de comunicações dedicada, ou seja, um sistema de memória distribuída. Seja S_i a capacidade de um processador i, com $i \in P$. Diz-se que um *cluster* é homogéneo quando:

$$\forall_{i,j} \in P,\ i \neq j \Rightarrow S_i = S_j$$

Se pelo contrário, a propriedade anterior não se verificar diz-se que o *cluster* é heterogéneo.

Neste trabalho, considera-se que uma aplicação, também designada por *job*, é constituída por um conjunto de tarefas dependentes, sendo usualmente representada através de um DAG (directed acyclic graph). Um DAG é uma estrutura formada por um conjunto de nós e de arestas, Figura 2.1. As arestas ligam os nós e caracterizam-se por terem sentido, estabelecendo desta forma uma relação de precedência entre eles. Esta estrutura é acíclica, ou seja, não existe nenhum caminho que tenha início e fim no mesmo nó. Diz-se que um nó é inicial quando nenhum outro se lhe liga (não tem pais), por outro lado, diz-se que um nó é final quando este não se liga a nenhum outro (não tem filhos). Considera-se que

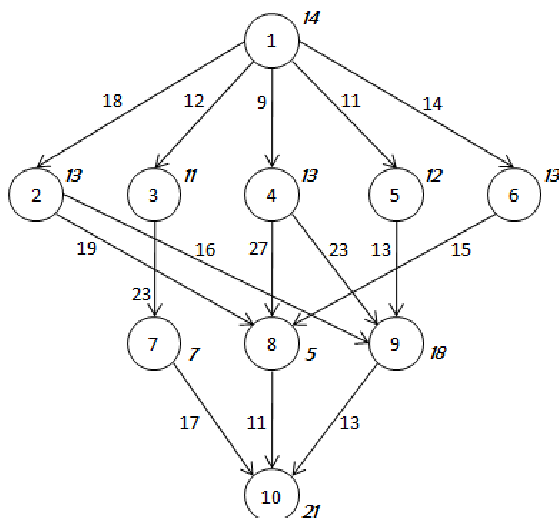

Figura 2.1: Aplicação com 10 tarefas, representada através de um DAG

cada nó é uma tarefa da aplicação, e as arestas representam as relações de precedência e os custos de comunicação entre as diferentes tarefas.

O processo de escalonamento de uma aplicação, consiste em alocar processadores às tarefas e em definir o início e fim de execução de cada uma delas. Estes dois procedimentos são designados por *match* e *scheduling*, respectivamente [MAS+99, KSS+07]. Neste trabalho utiliza-se a designação de escalonamento, para os dois procedimentos anteriores.

O escalonamento de aplicações é, em geral, um problema NP-completo [KA99]. A intratabilidade do problema para a obtenção da solução óptima, levou à proposta vários algoritmos, que obtêm em tempo polinomial uma solução, que embora não seja óptima, é considerada na generalidade das situações, como uma boa solução [BMNM05, THW02, SZI07, KSS+07, MAS+99]. Estes algoritmos baseiam-se, normalmente, em heurísticas, contudo, são geralmente muitos restritivos, não se adequando à generalidade das aplicações e ambientes de execução. Os algoritmos propostos para o escalonamento de DAGs num ambiente heterogéneo, consideram apenas o escalonamento de um DAG, além de serem abordagens estáticas. Os trabalhos sobre escalonamento dinâmico, por outro lado, consideram que uma aplicação é uma tarefa independente, alocando um processador por aplicação. O problema que se pretende resolver neste trabalho é mais complexo, pois considera um ambiente multi-utilizador, onde múltiplas aplicações independentes são submetidas ao longo do tempo, cada uma composta por um conjunto de tarefas dependentes.

Um *cluster* pressupõe uma utilização multi-utilizador, isto é, cada utilizador pode submeter aplicações para execução. Considere-se n o número total de aplicações submetidas num *cluster*, por todos os utilizadores, num dado momento.

Seja $G_a = (V_a, E_a)$ uma aplicação a, com $a \in [1, n]$, onde V_a representa o conjunto das tarefas dependentes da aplicação, e E_a representa o conjunto das precedências entre as

suas tarefas e os custos de comunicação. Para cada tarefa $i \in V_a$ é definido o tempo de início $ST(i)$ e de fim $FT(i)$ de execução.

O tempo de execução global de todas as aplicações, também designado por *makespam*, é expresso através de:

$$\forall_{i,j} \in \bigcup_{a=1}^{n} V_a \; : \; makespam \; = \; max\{FT(i)\} \; - \; min\{ST(j)\}$$

Neste trabalho, a função objectivo é minimizar o *makespan*, considerando o problema de escalonamento com o seguinte domínio:

- Uma aplicação é constituída por um conjunto de tarefas dependentes;
- As aplicações são inseridas em tempo de execução, por múltiplos utilizadores;
- Não se conhece a frequência de chegada de uma nova aplicação;
- Os tempos de execução e comunicação de cada tarefa podem ser estimados;
- O *cluster* pode ser heterogéneo;

2.1.1 Problema NP-completo

Um problema diz-se NP-completo (Nondeterministic Polynomial time) quando apresenta as seguintes propriedades:

- Qualquer solução pode ser verificada em tempo polinomial (problema da classe NP);
- Se um problema pode ser resolvido em tempo polinomial, também o pode qualquer problema da classe NP.

Embora se possa verificar, em tempo polinomial, a solução de um problema, não existe um método que permita obter essa solução em tempo polinomial. O tempo necessário para calcular a solução óptima de um problema NP-completo aumenta de forma rápida à medida que o tamanho do problema cresce.

O escalonamento de aplicações é um problema NP-completo na generalidade das situações, existindo apenas algumas situações particulares em que isto não se verifica [KA99].

Assim, é necessária a utilização de técnicas que possibilitem o cálculo de soluções em tempo polinomial que permitam dar resposta ao problema geral do escalonamento de tarefas.

2.1.2 Heterogeneidade de um sistema

Um sistema diz-se heterogéneo quando existe uma variação nas características dos componentes que o constituem. Neste trabalho, considera-se um *cluster*, como um conjunto P de processadores, que podem ter capacidades de processamento diferentes, ligados através de uma rede de comunicações dedicada. Assim, neste trabalho, a única fonte de heterogeneidade que se considera, é a variação da capacidade dos processadores que constituem o *cluster*.

O problema do escalonamento de tarefas num *cluster* heterogéneo difere daquele que é considerado num homogéneo, pois como os processadores têm capacidades diferentes, o tempo de execução de cada uma das tarefas depende do processador em que é executada. A aplicabilidade e a força dos sistemas heterogéneos reside na habilidade de utilizar apropriadamente os recursos disponíveis para a execução de tarefas [MAS+99].

Os algoritmos estáticos de escalonamento, geralmente, lidam com a heterogeneidade dos *clusters* através da estimativa do tempo de execução de cada uma das tarefas nos diferentes processadores, de forma que seja atribuído a cada uma, os processadores que minimizam a métrica que está a ser considerada. Existem diversos processos para o cálculo da estimativa do tempo de execução de uma tarefa. Normalmente é utilizado o *benchmark* de cada um dos processadores, os tempos de execução das tarefas anteriores, ou os parâmetros fornecidos pelos utilizadores [ASM+00].

O modelo ETC (Expected Time to Compute), proposto em [ASM+00], caracteriza um sistema através das seguintes propriedades: heterogeneidade das tarefas, e heterogeneidade dos processadores. A heterogeneidade das tarefas reflecte-se na variação do seu tempo de execução num mesmo processador. A variação do tempo de execução de uma tarefa nos diferentes processadores de um *cluster*, caracteriza a heterogeneidade dos processadores. A análise de cada linha da Tabela 2.1, permite verificar a variação dos tempos de execução de uma mesma tarefa pelos diferentes processadores disponíveis. A análise de cada coluna, mostra a variação dos tempos de execução das diferentes tarefas num mesmo processador.

A partir da caracterização anterior, um sistema pode ser classificado em duas categorias: consistente ou inconsistente. Seja t_i e t_j, o tempo de execução das tarefas i e j, num processador p, com $p \in P$. Diz-se que um sistema é consistente quando: se $t_i < t_j$ para um processador p, então isto também se verifica para qualquer outro processador do *cluster*. Num sistema inconsistente a relação anterior não se verifica. Neste trabalho considera-se que um *cluster* é consistente.

A heterogeneidade de um sistema em relação às máquinas utilizadas é calculada através da análise de todos os tempos de execução de todas as tarefas. Quanto maior a variação que se verifica nos tempos de execução maior será a heterogeneidade.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
t_1	440	762	319	532	151	652	308
t_2	459	205	457	92	92	379	60
t_3	499	263	92	152	75	18	128
t_4	421	362	347	194	241	481	391
t_5	276	636	136	355	388	324	255
t_6	89	139	37	67	9	113	0
t_7	404	521	54	295	257	208	539
t_8	49	114	279	22	93	39	36
t_9	59	35	184	262	145	287	277
t_{10}	7	235	44	81	330	56	78
t_{11}	716	601	75	689	299	144	457
t_{12}	435	208	256	330	6	394	419

Tabela 2.1: Tempos de execução das tarefas num cluster com 7 processadores, referido em [ASM+00]

Em [ASM+00] define-se o coeficiente de variação V, como uma medida da heterogeneidade. Seja $V = \frac{\sigma}{\mu}$, onde σ é o desvio padrão e μ a média de um conjunto de tempos de execução. O coeficiente de variação, expressa o desvio padrão como percentagem da média dos valores considerados. Neste trabalho, o coeficiente de variação é utilizado para quantificar a heterogeneidade do conjunto de processadores de um *cluster*.

2.2 Trabalho relacionado

As aplicações são constituídas por tarefas que podem ser classificadas como independentes ou dependentes. As tarefas independentes caracterizam-se por não possuírem nenhuma relação de precedência com qualquer outra. Neste caso, considera-se que cada tarefa independente é uma aplicação. As tarefas dependentes, caracterizam-se por possuírem uma relação de ordem de execução entre elas. Ou seja, uma tarefa não pode executar sem todas que a precedem terem terminado. Uma aplicação constituída por tarefas dependentes pode ser representada através de um DAG.

O escalonamento de aplicações, é um procedimento que consiste em alocar processadores às tarefas e em definir o início e fim de execução de cada uma delas [BMNM05, BM08, SZI07, KSS+07, MAS+99]. Este procedimento tem como objectivo aumentar o desempenho de uma métrica definida, sendo as mais usadas: a minimização do tempo de execução global das tarefas e a maximização da utilização dos recursos. Contudo, outras métricas podem ser consideradas. Em [KSS+07] são analisadas e testadas um conjunto heurísticas para tarefas independentes, onde o objectivo é maximizar a qualidade de serviço (QoS). A QoS está associada a uma métrica definida, que tem em conta a prioridade, *deadline* e desempenho das tarefas escalonadas.

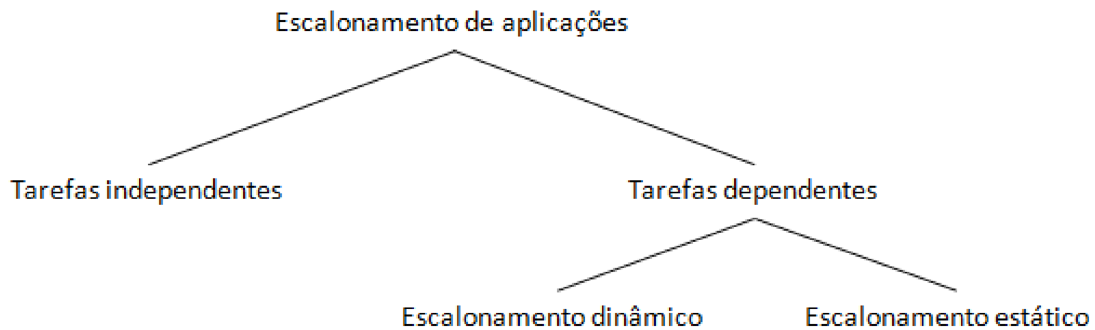

Figura 2.2: Classificação do tipo de escalonamento de aplicações

O problema do escalonamento de aplicações, pode ser dividido em duas categorias [KA99], de acordo com o tipo de tarefas que as constituem. A Figura 2.2 esquematiza essa divisão.

Quando se consideram tarefas independentes, como não existe relação de precedência entre elas, o escalonamento é geralmente realizado em tempo de execução. O objectivo é aumentar o desempenho global do sistema. No caso de as aplicações serem constituídas por tarefas dependentes, podem ser consideradas duas abordagens diferentes para o seu escalonamento: escalonamento estático e dinâmico [KA99, THW02, BM08].

O escalonamento diz-se estático quando é realizado em tempo de compilação ou de inicialização, sendo necessário conhecer à priori os tempos de chegada, execução, comunicação, e as dependências das tarefas das aplicações. Por outro lado, diz-se que o escalonamento é dinâmico, quando é realizado em tempo de execução, por não conhecer ou não ser possível prever o tempo de execução das tarefas, ou não se conhecer a frequência de chegada de cada uma das aplicações.

2.2.1 Algoritmos estáticos

Os algoritmos estáticos são utilizados quando é possível à partida, ter o conhecimento das características das tarefas e do sistema computacional, pressupondo-se que estas não se alteram ao longo do tempo. Quando se utiliza este tipo de algoritmos, o escalonamento é, geralmente, realizado em tempo de compilação ou de inicialização. O objectivo deste tipo de métodos é, normalmente, a minimização do tempo de execução global, *makespan*, de um conjunto de tarefas [BMNM05, KA99, KA05, THW02].

Os algoritmos estáticos de escalonamento de tarefas podem ser classificados em dois grandes grupos, como esquematizado na Figura 2.3: os que são baseados em heurísticas e os baseados na pesquisa aleatória [THW02].

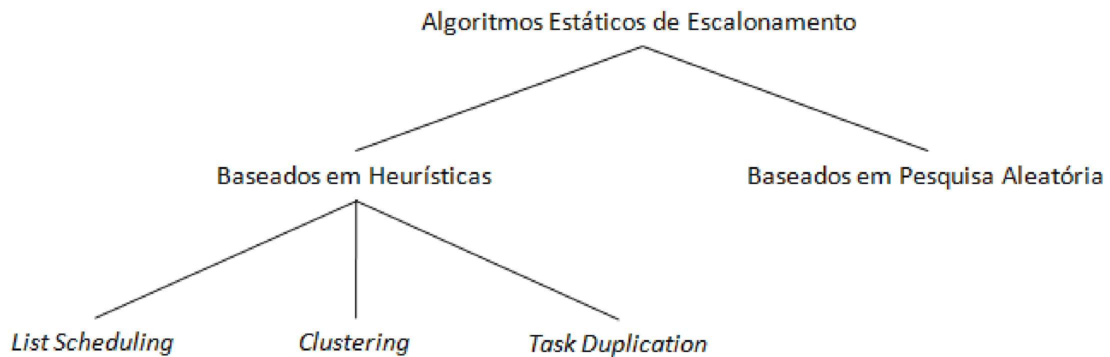

Figura 2.3: Classificação dos algoritmos estáticos de escalonamento

Uma heurística é uma técnica utilizada para a resolução de problemas onde o custo para a obtenção da solução óptima é elevado. A utilização de uma heurística não assegura a obtenção da solução óptima de um problema, mas somente uma solução válida, geralmente considerada uma boa solução. As heurísticas utilizam-se com o objectivo de aumentar o desempenho ou a simplicidade conceptual de um problema, contudo, sacrificando a precisão e exactidão dos resultados obtidos. Os algoritmos baseados na pesquisa aleatória, combinam o conhecimento adquirido nas pesquisas e resultados anteriores com algumas características aleatórias para gerar novos resultados. Os algoritmos genéticos pertencem a este grupo, e caracterizam-se por produzirem bons resultados no escalonamento de tarefas, contudo, geralmente apresentam um tempo de execução superior quando comparados com os algoritmos baseados em heurísticas.

Os algoritmos estáticos de escalonamento, baseados em heurísticas, caracterizam-se pelo tipo de heurística que utilizam. Em [THW02] identificam-se três tipos de heurísticas, nomeadamente: *Clustering*, *Task Duplication* e *List Scheduling*.

2.2.1.1 Clustering

Os algoritmos que utilizam uma heurística do tipo *Clustering*, começam por alocar cada uma das tarefas a um conjunto diferente, cada um deles designado por *cluster*. Cada nova iteração refina a anterior através da reunião de alguns conjuntos. Quando duas tarefas pertencem ao mesmo conjunto, significa que irão executar no mesmo processador. É necessário efectuar as iterações necessárias para que, através da operação reunião, o número de conjuntos seja igual ao número de processadores. Por fim, é necessário efectuar a alocação das tarefas de cada um dos conjuntos nos respectivos processadores e estabelecer a ordem de execução.

2.2.1.2 Task Duplication

As heurísticas do tipo *Task Duplication* caracterizam-se por utilizarem a redundância na alocação de tarefas, atribuindo a mesma tarefa a vários processadores, com o objectivo de reduzir o *overhead* nas comunicações entre tarefas. Este tipo de heurísticas diferem na forma de como efectuam a selecção das tarefas que são duplicadas. Os algoritmos que utilizam este tipo de heurísticas, apresentam uma maior complexidade temporal do que os que utilizam as dos outros dois grupos.

2.2.1.3 List scheduling

Os algoritmos de escalonamento que são utilizados neste trabalho são baseados na heurística *list scheduling*. Esta, técnica caracteriza-se por manter uma lista ordenada com todas as tarefas a escalonar, segundo as suas prioridades [KA99, THW02]. Consideram-se as seguintes etapas:

1. determinar as tarefas disponíveis para execução;

2. definir a sua prioridade;

3. até todas as tarefas estarem escalonadas, seleccionar a tarefa com maior prioridade e atribuí-la ao processador que minimiza a função de custo (por exemplo, o que está disponível primeiro).

Os algoritmos de escalonamento que utilizam esta técnica, geralmente apresentam uma menor complexidade e apresentam melhores resultados que os restantes [THW02].

Os atributos que são usualmente usados na definição das prioridades das tarefas, são o *t-level* (top-level) e o *b-level* (bottom level). *O t-level* de uma tarefa i é o caminho mais longo entre a tarefa inicial e i, excluindo o custo de computação da tarefa i. O *b-level* de uma tarefa j é o comprimento do caminho mais longo entre j e a tarefa final. O comprimento do caminho obtêm-se através da soma do custo de computação e de comunicação das tarefas consideradas. A tarefa que em cada nível tem o maior *b-level*, pertence ao caminho crítico do DAG [BM08].

O caminho mais longo que se pode definir num DAG é designado por caminho crítico, Figura 2.4. Ou seja, é a sequência de tarefas que comporta o maior custo global. O tempo de execução das tarefas desta sequência determina o tempo mínimo de execução da aplicação. Qualquer atraso que se verifique na execução destas tarefas irá repercutir-se no tempo de execução da aplicação. O caminho crítico pode variar ao longo da execução das tarefas, dependendo do escalonamento efectuado em cada momento [BMNM05].

Descreve-se, seguidamente, uma forma de cálculo do *t-level* e do *b-level*, como apresentado em [KA99].

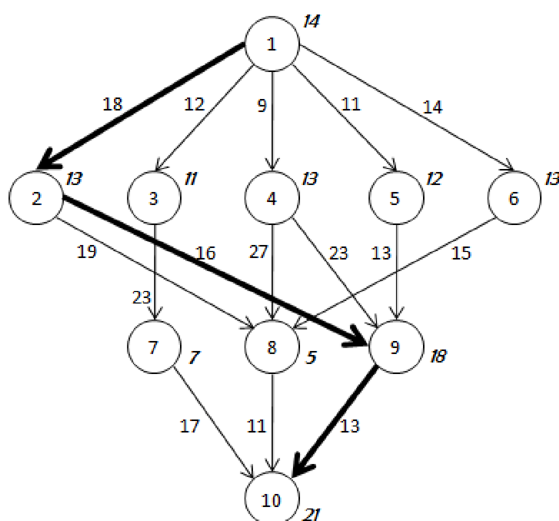

Figura 2.4: Caminho crítico da aplicação definida em 2.1

Cálculo do t-level:

1. Constrói uma lista (TopList) com as tarefas ordenadas topologicamente
2. Para cada tarefa i na TopList
3. max=0
4. Para cada pai x de i
5. Se t_level$(x)+w(x)+c(x,i)>$ max
6. max=t_level$(x)+w(x)+c(x,i)$
7. t_level$(i)=$ max

Inicialmente, as tarefas são ordenadas topologicamente numa lista (linha 1). Seguidamente, nas linhas 4, 5 e 6, para cada um dos pais x da tarefa i é calculada a soma entre o seu *t-level*, o seu custo de computação e o custo de comunicação entre x e i. O maior valor obtido na soma anterior será definido como o *t-level* da tarefa i (linha 7).

Cálculo do b-level:

1. Constrói uma lista (RevTopList) com as tarefas ordenadas de forma inversa à ordem topológica
2. Para cada tarefa i na RevTopList
3. max=0
4. Para cada filho y de i
5. Se $c(i,y)+$ b_level$(y)>$ max
6. max= $c(i,y)$ b_level(y)
7. b_level$(i)=w(i)+$ max

Inicialmente as tarefas são ordenadas por ordem inversa da topológica numa lista (linha 1). Nas linhas 4, 5 e 6, para cada um dos filhos y da tarefa i é calculada a soma entre o seu custo de comunicação e o valor do *b-level* de y. O valor do *b-level* da tarefa i será definido como o maior valor obtido na soma anterior, adicionado do custo de computação,

$w(i)$ da tarefa i (linha 7).

A complexidade temporal do cálculo do *t-level* e *b-level*, apresentado anteriormente, é em ambos os casos $O(e+v)$, onde e é o número de arestas e v o número de tarefas.

Os valores do *b-level* e *t-level* da aplicação apresentada na Figura 2.1 estão apresentados na Tabela 2.2.

Tarefa	t-level	b-level
1	0	113
2	32	81
3	26	79
4	23	88
5	25	77
6	28	65
7	60	45
8	64	37
9	61	52
10	92	21

Tabela 2.2: Os valores do t-level e b-level das tarefas da aplicação da Figura 2.1

Nos trabalhos [THW02, BMNM05, BM08] são apresentados e analisados algoritmos estáticos de escalonamento, que utilizam a técnica de *list scheduling*. Estes algoritmos realizam o escalonamento de aplicações, constituídas por um conjunto de tarefas dependentes que podem ser representadas através de um DAG, num *cluster* heterogéneo. O objectivo considerado nestes trabalhos é minimizar o tempo de execução global de um conjunto de aplicações.

Em [THW02], são apresentados dois novos algoritmos, nomeadamente o HEFT (Heterogeneous Earliest Finish Time) e o CPOP (Critical Path On a Processor). O desempenho destes dois algoritmos é comparado com outros existentes, concluindo que o HEFT supera os resultados atingidos pelos restantes. O HEFT é apresentado como a melhor solução para o escalonamento de aplicações, que podem ser representadas através de um DAG.

A abordagem usual no escalonamento de aplicações, constituídas por tarefas dependentes, é atribuir um processador a uma tarefa, como considerado em [THW02]. Assim, está-se a considerar uma perspectiva sequencial na execução de cada uma das tarefas. A utilização de tarefas paralelas, *malleable tasks*, permite que a cada tarefa sejam atribuídos vários processadores, sendo o seu tempo de execução função do número de processadores alocados a ela.

Os algoritmos apresentados em [BMNM05, BM08], consideram que as tarefas que constituem as aplicações podem executar de forma paralela. Em [BMNM05] é proposto um novo algoritmo de escalonamento e os seus resultados são comparados com

a execução sequencial das tarefas e com a existência de um número ilimitado de processadores, permitindo que cada tarefa execute no seu tempo óptimo. Estes resultados são obtidos utilizando diferentes cenários de CCR (Comunications to Computation Ratio). Em [BM08] é apresentado um algoritmo que se baseia em [BMNM05]. O seu desempenho é comparado com o algoritmo HEFT apresentado em [THW02]. É mostrado, que o escalonamento de tarefas paralelas pode melhorar o desempenho de um *cluster*, permitindo uma utilização mais eficiente dos recursos disponibilizados pelo *cluster*. Embora seja sugerida uma abordagem dinâmica, todo o trabalho é apresentado numa perspectiva estática.

O escalonamento de tarefas paralelas também é tratado em [BMW+04]. Neste trabalho é apresentado um algoritmo polinomial, contudo, as tarefas consideradas são independentes. É assumido que o número de processadores é suficiente para processar todas as tarefas simultaneamente. O objectivo é a minimização do tempo global de execução.

2.2.2 Algoritmos dinâmicos

O escalonamento dinâmico de aplicações ocorre quando é efectuado em tempo de execução. Utiliza-se este tipo de algoritmos quando não é possível estimar o tempo de execução das tarefas que constituem cada aplicação, ou não se conhece a frequência de chegada das aplicações. Esta última abordagem será a discutida no trabalho.

O desconhecimento ou impossibilidade de previsão do tempo de execução das tarefas, implica que o seu escalonamento seja efectuado em tempo de execução. Esta abordagem dinâmica utiliza algoritmos que permitem alterar o escalonamento das tarefas ao longo do tempo, com o objectivo de equilibrar a carga pelos diferentes processadores. Este problema é conhecido por *load balancing*. Um escalonamento onde cada processador está sujeito à mesma carga, permite, geralmente, um aumento da eficiência e consequentemente, a redução do tempo de processamento [WT98]. O processo de balanceamento da carga, segundo [WT98], tem as seguintes fases:

1. Calcula uma estimativa da carga de cada processador e da computação de cada uma das tarefas;

2. Verifica se a carga está balanceada. Se não, determina se é proveitoso efectuar seu o balanceamento;

3. Calcula a carga que é necessária transferir para esta ficar balanceada;

4. Selecciona as tarefas que melhor correspondem ao calculado no ponto anterior. A escolha destas tarefas é geralmente constrangida pelas comunicações e tamanho de cada uma delas;

5. Uma vez seleccionadas, as tarefas são transferidas entre processadores. A integridade do estado de cada uma das tarefas deve ser garantida.

Pode-se classificar os algoritmos de escalonamento dinâmicos como *preemptivos* e *não-preemptivos*. O escalonamento *preemptivo* permite que a execução de uma tarefa seja interrompida e que a porção da tarefa que falta executar seja reescalonada para outros processadores. Pelo contrário, se o escalonamento for *não-preemptivo*, a tarefa depois de iniciar a sua execução, não pode ser reescalonada.

Por outro lado, o desconhecimento da frequência de chegada das aplicações a escalonar, remete a outro tipo de abordagem dinâmica [MAS+99, SZI07]. Os algoritmos que atendem a esta abordagem podem ser classificados em dois tipos: *immediate mode* e *batch mode* [MAS+99, SZI07]. O *immediate mode* caracteriza-se por invocar o algoritmo de escalonamento quando uma nova aplicação é submetida, sendo apenas esta escalonada. No *batch mode* o algoritmo de escalonamento é invocado quando uma nova aplicação é submetida, contudo considera não só a nova aplicação, mas também o conjunto das tarefas já escalonadas que aguardavam para serem executadas. Enquanto no *immediate mode* as tarefas apenas são escalonadas um vez, no *batch mode* as tarefas são escalonadas sempre que uma nova aplicação é submetida, até serem executadas.

Em [MAS+99] é comparado o desempenho de dois conjuntos de algoritmos, uns que utilizam uma heurística do tipo *immediate mode* com outros, que usam uma do tipo *batch mode*. Conclui-se que o uso de uma heurística do tipo *batch mode* permite atingir um melhor desempenho, quando se pretende minimizar o tempo de execução global do escalonamento. Em [SZI07] é apresentado um método para o escalonamento de tarefas independentes que inclui uma estratégia e um algoritmo genético. A estratégia e o algoritmo trabalham em conjunto de forma a implementar uma abordagem dinâmica. Foi comparado o desempenho do método com outras propostas, através da simulação, utilizando um elevado número de tarefas. Em ambos os trabalhos é considerado o escalonamento de tarefas independentes, num *cluster* heterogéneo, alocando apenas um processador a cada tarefa, ou seja, não consideram tarefas paralelas [MAS+99, SZI07]. Também em ambos os casos, se conhece à priori, os tempos de execução de cada tarefa.

2.3 Resumo e Conclusões

Depois da análise anterior, verifica-se que os algoritmos que são propostos para o escalonamento de aplicações constituídas por tarefas dependentes, num *cluster* heterogéneo, apenas consideram um DAG, além de serem abordagens estáticas. Os trabalhos sobre escalonamento dinâmico, por outro lado, consideram que uma aplicação é uma tarefa independente, alocando um processador por aplicação.

Neste trabalho pretende-se efectuar o escalonamento de várias aplicações independentes, representadas através de DAGs, não se conhecendo a sua frequência de chegada. Assim, desenvolveu-se um método de escalonamento que tem uma abordagem dinâmica, embora seja utilizado um algoritmo estático. O método, como em [SZI07], é dividido numa estratégia e num algoritmo de escalonamento. A abordagem dinâmica é conseguida com a utilização de uma estratégia que define o momento em que o algoritmo estático é invocado. Esta estratégia utiliza uma heurística do tipo *batch mode*, em detrimento de uma do tipo *immediate mode*, pois permite atingir um melhor desempenho, como referido em [MAS+99]. Assim, quando uma nova aplicação é submetida são avaliadas, pelo algoritmo estático de escalonamento, todas as tarefas que ainda não foram executadas, sendo novamente escalonadas, tendo em conta a nova realidade. Os algoritmos estáticos considerados no método desenvolvido, permitem o escalonamento de aplicações constituídas por tarefas dependentes. Embora o método utilize uma abordagem dinâmica, por não se conhecer a frequência de inserção de uma nova tarefa, é necessário conhecer à priori os tempos de execução e de comunicação de cada uma das tarefas, para ser possível a utilização do algoritmo estático de escalonamento.

Capítulo 3

Método de escalonamento

Neste capítulo é realizada a analise da estrutura e funcionamento do método de escalonamento dinâmico que foi desenvolvido neste trabalho. Este método pode ser dividido em duas partes: a estratégia e o algoritmo estático de escalonamento. A estratégia, determina o momento em que o algoritmo de escalonamento é invocado e que tarefas devem ser escalonadas. O algoritmo de escalonamento atribui os processadores às tarefas e define a sua ordem de execução.

Inicialmente, é analisada a estratégia utilizada. Seguidamente, são analisados os dois algoritmos estáticos de escalonamento que foram considerados neste trabalho, o HEFT, o HPTS, e a proposta de optimização do HPTS, designada por iHPTS. Para cada um deles é descrito o modelo computacional considerado.

3.1 Estratégia de escalonamento

A estratégia de escalonamento especifica o tipo de escalonamento dinâmico que se adopta, *immediate mode* ou *batch mode*, estabelecendo o momento em que o algoritmo de escalonamento é invocado e que tarefas serão escalonadas. Neste trabalho adoptou-se uma estratégia do tipo *batch mode*.

Algoritmo: Estratégia

```
1. while(1)
2.   Aguarda por um novo job
3.   Insere o job no DAG principal
4.   Invoca o algoritmo de escalonamento
```

Inicia-se um novo ciclo quando uma aplicação, ou *job*, é submetida no *cluster*. Uma aplicação é inserida no DAG principal, que contém todas as aplicações que estão a ser processadas, como mostra a Figura 3.1. Este DAG é caracterizado por possuir um nó de

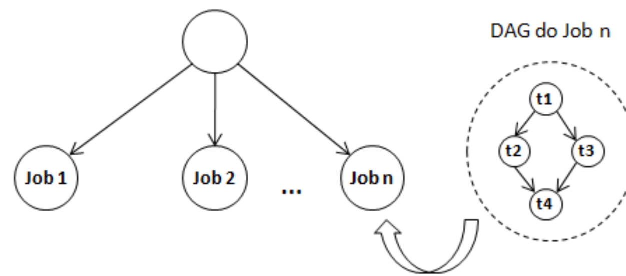

Figura 3.1: Exemplo do DAG onde são inseridos os jobs para serem escalonados

entrada, sem custos de computação, o qual está ligado a todas as aplicações já processadas pelo algoritmo de escalonamento, embora as suas tarefas ainda não tenham sido todas executadas. Esta ligação, é a única que se verifica entre aplicações, pois não existe nenhuma relação de precedência entre si. Como também não há custos de comunicação, não se estabelece qualquer relação de prioridade nem de ordem entre as aplicações. As aplicações são independentes.

Após efectuada a sua inserção, o algoritmo de escalonamento é invocado para escalonar as tarefas da nova aplicação e reescalonar as dos restantes que ainda não haviam sido executadas.

Se a capacidade do *cluster* for excedida, devido ao elevado número de tarefas que podem executar simultaneamente num dado momento (*número de tarefas* > *número de processadores disponíveis*), estas serão executadas quando houver disponibilidade, todavia sem qualquer reserva de recursos para uma aplicação específica. Esta é uma característica importante do método de escalonamento proposto, como mostrado no Capítulo 5, pois aumenta o desempenho do *cluster* atendendo ao menor tempo de execução global das aplicações submetidas. Esta abordagem possibilita, ao contrário da reserva estática de processadores, minimizar o estado *idle* dos processadores, pois a capacidade de processamento que não é utilizada por uma aplicação pode ser utilizada pelas restantes.

Na figura 3.2 está esquematizado o modo de funcionamento da estratégia de escalonamento do método desenvolvido, num cluster homogéneo constituído por 10 processadores. Na secção (A) está representado o mapa de escalonamento de uma aplicação onde a tarefa número 1 está em execução nos processadores 1, 2, 3 e 4. Decorridos 10 segundos, é submetida uma nova aplicação no cluster. A estratégia implementada elimina o escalonamento anterior, como representado na secção (B), e é invocado o algoritmo de escalonamento que irá atender às tarefas das duas aplicações para efectuar um novo escalonamento, secção (C). É considerado que as aplicações após iniciarem a sua execução não podem ser reescalonadas.

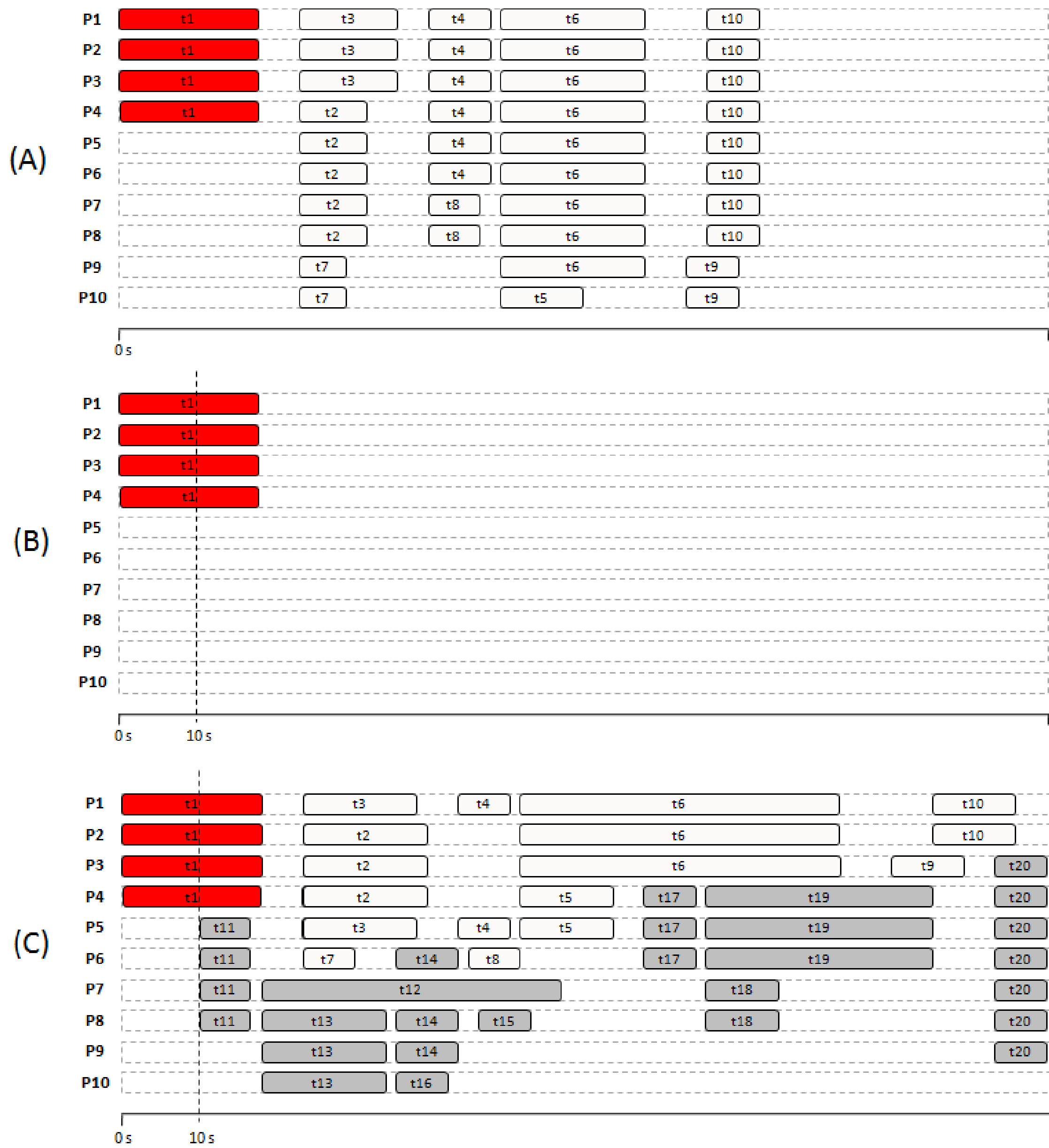

Figura 3.2: Exemplo do modo de funcionamento da estratégia de escalonamento

3.2 Algoritmo de escalonamento

O algoritmo de escalonamento é invocado quando uma nova aplicação é inserida no DAG descrito anteriormente. Este algoritmo determina em que processadores cada uma das tarefas irá executar (*match*) e quando irá executar (*scheduling*), baseado na informação que dispõem de cada uma delas e da disponibilidade do *cluster* em cada instante. Neste trabalho foram considerados dois algoritmos para o escalonamento das aplicações, nomeadamente o *Heterogeneous Earliest-Finish-Time* (HEFT) [THW02] e o *Heterogeneous Parallel Task Scheduler* (HPTS) [BMNM05], por terem sido apresentados como boas soluções no escalonamento de DAGs em *clusters* heterogéneos. Da análise e estudo destes dois algoritmos desenvolveu-se uma proposta de optimização do algoritmo HPTS, que se designou de *improved Heterogeneous Parallel Task Scheduler* (iHPTS).

O algoritmo HEFT é apresentado como o melhor algoritmo para o escalonamento de DAGs em *clusters* heterogéneos, em [THW02]. Este algoritmo atribui um processador a uma tarefa. Por outro lado, o HPTS considera que as tarefas podem executar paralelamente podendo atribuir a uma tarefa vários processadores. No entanto, em ambos os algoritmos, cada processador está associado apenas a uma tarefa, em cada instante. O escalonamento de tarefas de forma paralela depende da carga do *cluster*, ou seja, para situações de elevada carga (número de tarefas $\geq$ número de processadores) o HPTS tende a atribuir apenas um processador por tarefa como o HEFT. Estes dois algoritmos foram implementados utilizando o modelo computacional com que originalmente foram propostos em [THW02] e [BMNM05]. É mostrado em [BM08] que quando a paralelização de tarefas pode ser utilizada, o HPTS atinge um melhor desempenho do que o HEFT. Outros algoritmos de escalonamento podem ser utilizados nesta fase.

Os algoritmos de escalonamento considerados, são baseados na técnica *list scheduling* [KA99] que consiste nos seguintes passos: a) determinar as tarefas disponíveis para execução, b) definir a sua prioridade, c) até todas as tarefas estarem escalonadas, seleccionar a tarefa com maior prioridade e atribuí-la ao processador que está disponível primeiro. Em *clusters* heterogéneos, o último passo deve ser readaptado pois o processador que está disponível primeiro pode não ser aquele permita a execução da tarefa no menor tempo global, podendo ser vantajoso analisar os restantes. Para o escalonamento de tarefas paralelas, o último passo selecciona não apenas um processador mas vários [BMNM05]. Os algoritmos analisados utilizam os atributos *b-level* e o *t-level* para a definição das prioridades de cada uma das tarefas.

3.2.1 Algoritmo HEFT

3.2.1.1 Modelo Computacional

O sistema computacional que se considera é um *cluster*, constituído por um conjunto P de processadores, que podem ter capacidades de processamento diferentes, e estão ligados através de uma rede *Ethernet* dedicada. A rede suporta comunicações simultâneas entre os diferentes processadores. Cada uma das aplicações, *job*, é representada através de um DAG e o tempo de execução de cada uma das suas tarefas pode ser estimado em tempo de compilação ou antes de ser efectuado o seu escalonamento. É assumido que as tarefas são não-preemptivas e que as comunicações podem ocorrer sobrepostas à execução de tarefas.

Seja W uma matriz $v \times q$, onde v é o número de tarefas e q o número de total de processadores, em que cada elemento $w_{i,j}$ é o custo estimado de computação da tarefa n_i no processador p_j. Antes de ser efectuado o escalonamento, as tarefas são iniciadas com o tempo médio de execução. Considere-se uma tarefa n_i, então o seu tempo médio de execução é definido por:

$$\overline{w_i} = \sum_{j=1}^{q} \frac{w_{i,j}}{q}$$

A informação dos tempos de comunicação é armazenada numa matriz B de tamanho $q \times q$. Os tempos de comunicação iniciais dos processadores são dados através de um vector L de dimensão q. Assim, o tempo de comunicação entre duas tarefas, n_i e n_k, escalonadas em processadores diferentes é definido por:

$$c_{i,k} = L_m + \frac{data_{i,k}}{B_{m,n}}$$

Quando ambas as tarefas são escalonadas no mesmo processador, considera-se que $c_{i,k}$ é zero.

Antes de ser efectuado o escalonamento, o tempo de comunicação de cada uma das arestas é iniciado com a média do tempo de comunicação entre as duas tarefas. Este é definido por:

$$\overline{c_{i,k}} = \overline{L} + \frac{data_{i,k}}{\overline{B}}$$

onde $\overline{B}$ é a média dos tempos de comunicação entre os processadores e $\overline{L}$ a média dos tempos de comunicação de inicialização.

Define-se o $EST(n_i, p_j)$ e $EFT(n_i, p_j)$ como o momento em que uma tarefa n_i pode iniciar e terminar, respectivamente, num processador p_j. O EST é definido por:

$$EST(n_i, p_j) = max\left\{avail\,|j|\,, \max_{n_m \in pred(n_i)} (AFT(n_m) + c_{m,i})\right\}$$

onde $pred(n_i)$ é o conjunto das tarefas antecessoras da tarefa n_i, e $avail\,|j|$ representa o momento em que o processador p_j está livre, pronto para executar uma nova tarefa. O $AFT(n_m)$ é o momento em que a tarefa n_m termina. Assim, o tempo de início da tarefa n_i, num processador p_j é igual ao máximo entre: o momento em que o processador está disponível e o valor máximo do tempo de fim das tarefas antecessoras adicionado com o tempo de comunicações.

O EFT é definido por:

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j)$$

sendo obtido através da soma do tempo de execução da tarefa, com o momento em que esta foi iniciada.

3.2.1.2 Descrição do Algoritmo

O algoritmo HEFT pode ser dividido em duas fases: o estabelecimento das prioridades de todas as tarefas e a selecção do processador para cada tarefa [THW02].

Algoritmo 2: HEFT

1. Atribui o custo de computação às tarefas e de comunicação às arestas com valores médios
2. Calcula o b_level
3. Ordena, de forma decrescente, as tarefas pelo seu b-level
4. Enquanto existem tarefas por escalonar
5. Selecciona a primeira tarefa n_i
6. Para cada processador p_j do conjunto de processadores
7. Calcula $EFT(n_i, p_j)$ utilizando a política de inserção
8. Aloca a tarefa n_i ao processador p_j que minimiza o seu EFT

Na primeira fase, linha 1 até à linha 3, o algoritmo calcula o *b-level* de cada uma das tarefas mediante os valores do custo de computação e comunicação que estão definidos em cada tarefa. Seguidamente, é criada uma lista de tarefas ordenadas de forma decrescente do seu *b-level*. Esta lista proporciona uma ordenação topológica das tarefas que constituem as diferentes aplicações preservando as dependências entre tarefas.

A segunda fase do algoritmo, linha 4 até à linha 8, atribui uma tarefa ao processador que lhe permite obter o menor tempo de fim de execução da tarefa (EFT). Este algoritmo

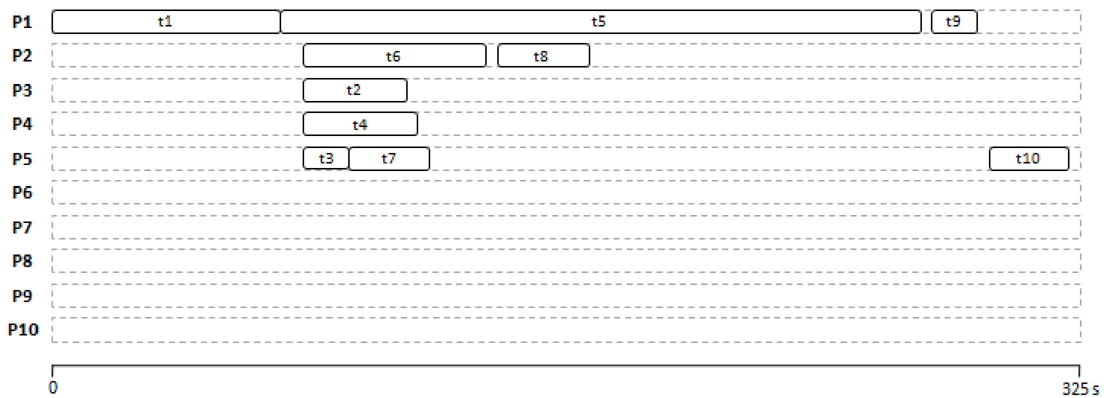

Figura 3.3: Mapa do escalonamento efectuado pelo HEFT de uma aplicação num cluster homogéneo com 10 processadores

implementa uma política de inserção que permite que uma tarefa seja escalonada no primeiro *slot* livre de um processador, entre duas tarefas já escalonadas. O tamanho do *slot* é a diferença entre o tempo de fim e de início das duas tarefas consecutivamente escalonadas no mesmo processador. Este deve ter pelo menos o tempo de execução da tarefa que está a ser escalonada bem como respeitar a precedência entre tarefas.

A Figura 3.3 o mostra mapa do escalonamento de uma aplicação com 10 tarefas, realizado pelo algoritmo HEFT, num *cluster* homogéneo constituído por 10 processadores. Verifica-se, que poucos processadores são utilizados, não se aproveitando a capacidade total que está disponível.

A complexidade temporal do algoritmo HEFT é $O(T^2 \times P)$ [THW02], sendo T o número de tarefas e P o número de processadores.

3.2.2 Algoritmo HPTS

3.2.2.1 Modelo Computacional

O sistema computacional que se considera é um *cluster*, constituído por um conjunto P de processadores, que podem ter capacidades de processamento diferentes, e estão ligados através de uma rede *Ethernet* dedicada. A rede suporta comunicações simultâneas entre os diferentes processadores. Cada uma das aplicações, (*jobs*), são representadas através de um DAG e o tempo de execução de cada uma das suas tarefas pode ser estimado em tempo de compilação ou antes de ser efectuado o escalonamento. É assumido que as tarefas são não-preemptivas.

As comunicações necessárias para a execução de uma tarefa são incluídas no tempo de execução como uma função dos $P_i \subseteq P$ processadores usados na tarefa. A comunicação entre tarefas é definida como uma função do tempo de execução da tarefa emissora e é representada através dos pesos nas arestas do DAG.

O modelo que estima o tempo de execução de cada tarefa, utiliza a capacidade de processamento, S_i, de cada processador $i \in P$, em $Mflops/s$, a latência a rede T_L, e a sua largura de banda ω medida em $Mbit/s$. O tempo total de execução é obtido através da soma do tempo de comunicação T_{comm}, e o tempo despendido nas operações paralelas, $T_{parallel}$. O tempo necessário para transmitir uma mensagem com b elementos é $T_{comm} = T_L + b\omega^{-1}$. O tempo necessário para executar paralelamente uma tarefa, sem qualquer parte sequencial ou tempo de sincronização, em $P_i \subseteq P$ processadores, com $P_i \neq \oslash$ é $T_{parallel} = f(n)/\sum_{i=1}^{p} S_i$. O numerador, $f(n)$ representa a função de custo do algoritmo considerado na tarefa, que depende do tamanho n do problema, e é expressa através do número de operações em vírgula flutuante por segundo.

Como exemplo, considere-se a multiplicação de matrizes quadradas $n \times n$, usando o algoritmo descrito em [GW95]. O número de operações de vírgula flutuante estimado é $f(n) = 2n^3$. O total de informação que é necessária transmitir numa rede de processadores $P = r \times c$ é $n^2(r-1)$ ao longo das linhas de processadores e $n^2(c-1)$ ao longo das colunas de processadores, resultando num total de $n^2(r+c-2)$ elementos. Se uma transmissão *broadcast* ao longo de uma coluna ou linha de processadores é considerada sequencial, então é transformada em $(r-1)$ e $(c-1)$ mensagens, respectivamente.

Finalmente, o tempo de execução do algoritmo de multiplicação de matrizes é dado por:

$$T = T_{comm} + T_{parallel} = \frac{n^2(r+c-2)}{w} + T_L + \frac{2n^3}{\sum_{i=1}^{p} S_i}$$

Esta expressão é executada para todos os $p \in P$, por ordem decrescente de capacidade, de forma a determinar o número de processadores que minimizam o tempo de execução.

3.2.2.2 Descrição do Algoritmo

O algoritmo HPTS considera que uma tarefa pode executar em vários processadores [BMW+04, Jan04, LMT02, Try01].

O tempo de execução $t_{i,p}$ da tarefa i pode ser descrito por uma função não monótona, uma vez que existe um número de processadores p tal que $t_{i,p} < t_{i,p-1}$ e $t_{i,p} < t_{i,p+1}$. Seja $t^*_{i,p}$ o tempo mínimo de execução da tarefa i num *cluster* heterogéneo, que é conseguido quando os processadores mais rápidos são utilizados. Outra combinação de processadores irá resultar numa menor capacidade de computação e, consequentemente, num maior tempo de execução da tarefa.

Da definição anterior, pode-se estimar o tempo mínimo de execução de cada uma das aplicações, o qual resulta da soma do tempo mínimo de processamento das tarefas no caminho crítico. $t_\infty = \sum_i t^*_{i,p}$, é o tempo requerido para executar todas as tarefas assumindo

um número ilimitado de processadores [Try01]. Neste caso, o *cluster* teria os recursos disponíveis para que qualquer tarefa fosse executada no seu tempo mínimo.

O tempo esperado para a execução de uma aplicação é maior, devido a nem todas as tarefas poderem ser executadas nos processadores com maior capacidade, o que pode alterar dinamicamente o caminho crítico. Tarefas com menor prioridade, depois de serem escalonadas, podem ser transformadas em tarefas que fazem parte do caminho crítico se a capacidade do *cluster* é inferior à capacidade requerida para obter o $t^*_{i,p}$. Devido a esta situação o HPTS avalia dinamicamente o *b-level* das tarefas que estão a ser escalonadas e efectua correcções de forma a reduzir o *b-level* máximo das mesmas.

A capacidade de processamento necessária para conseguir $t^*_{i,p}$ para a tarefa i é definida por $S^*_i = \sum_{j=1}^{p} S_j$, e é conseguida com a utilização dos processadores com maior capacidade. É óbvio que se forem utilizados processadores com capacidade inferior, a capacidade com que se consegue obter o tempo mínimo de execução para a tarefa i é $S'_i < S^*_i$, resultando então que $t'_i > t^*_i$. Uma vez que para obter S^*_i é necessário um maior número de processadores, necessariamente estarão implicadas mais comunicações e, consequentemente, mais tempo de processamento. Deste modo, o tempo mínimo de processamento conseguido será certamente superior ao estimado t^*_i.

Algoritmo: HPTS

```
1. Enquanto tarefas ≠ ∅
2.   Calcula o conjunto de tarefas prontas para execução
3.    Para cada tarefa pronta i
4.      Calcula a sua capacidade óptima  S_i^*
5.    Se ∑_i S_i^* > S_max
6.      Para cada tarefa pronta i
7.        S'_i = (S_max / ∑_j S_j^*) S_i^*
8.    caso contrário
9.      Para cada tarefa pronta i S'_i = S_i^*
10.  Calcula tl
11.  enquanto tarefas prontas ≠ ∅
12.     Selecciona a tarefa com o mínimo tl
13.
Selecciona os processadores que permitem tl
14.     Enquanto  S_i < S'_i and t_{i,p} < t_{i,p-1}
          adiciona processador
15.  Calcula bl
16.  Enquanto verdadeiro
17.     Selecciona a tarefa k com o maior bl
18.     Selecciona a tarefa r com o menor bl
19.     Se r já foi máximo
          então termina
20.     Retira um processador à tarefa r
21.     Adiciona-o à tarefa k
22.     Reavalia os tempo de execução
          das tarefas r e k
23.     Reavalia o bl das tarefas r e k
```

No algoritmo HPTS o *t-level* e o *b-level* estão representados por `tl` e `bl` respectivamente. A capacidade total do *cluster* é calculada através da soma da capacidade individual de cada processador pela fórmula $S_{max} = \sum_{i=1}^{P} S_i$. O ciclo `while` na linha 1 refere-se a todas as tarefas do DAG principal. Na linha 2 as tarefas prontas para o escalonamento, são aquelas cujas suas predecessoras foram já escalonadas. Da linha 3 à 9, o algoritmo determina a capacidade computacional que minimiza cada uma das tarefas prontas para serem escalonadas, S_i^*, de acordo com o modelo computacional, e assumindo que os processadores mais rápidos são utilizados. Nesta fase não é importante o número de processadores, mas sim a sua capacidade. De seguida, se a capacidade do *cluster* é excedida, S_{max}, a capacidade definida para cada uma das tarefas é limitada ao peso que cada uma representa. Da linha 10 até à 14, o algoritmo efectua o escalonamento de cada uma das tarefas que estão a ser consideradas e tenta alocar a capacidade de processamento determinada anteriormente a cada uma delas. A capacidade mínima que se pode atribuir a uma tarefa é a capacidade de um processador. Se não existirem processadores disponíveis para iniciar a tarefa no seu *t-level*, esta será atrasada no seu início de execução. Os processadores que são atribuídos às tarefas são aqueles que permitem a sua execução o mais cedo possível,

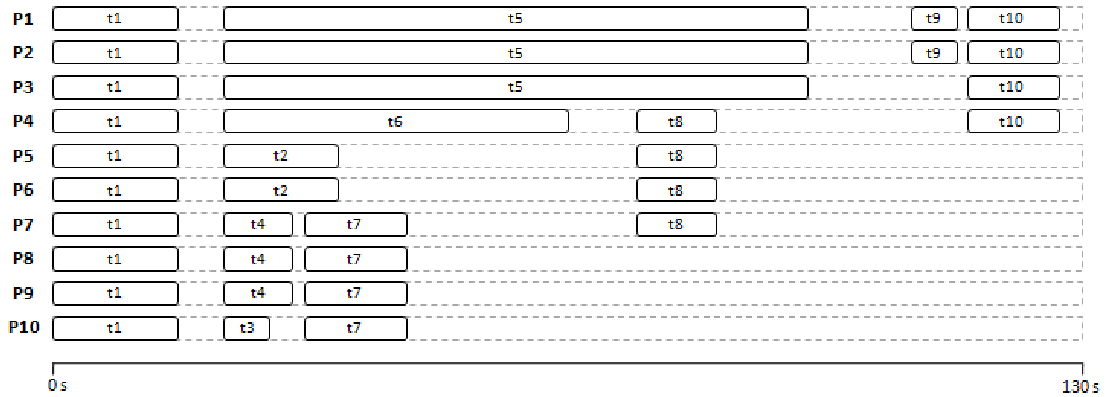

Figura 3.4: Mapa do escalonamento efectuado pelo HPTS de uma aplicação num cluster homogéneo com 10 processadores

porém é sempre verificado se começar mais tarde, com uma maior capacidade alocada (maior número de processadores) conduz a um menor tempo de fim de execução da tarefa. O tempo de execução de uma tarefa necessita de ser testado pois, num ambiente heterogéneo, nada garante que os processadores utilizados sejam os mais rápidos e, consequentemente, o tempo mínimo de execução pode ser atingindo com menos capacidade de processamento, contudo superior a $t^*_{i,p}$. Da linha 15 até à 23, o algoritmo tenta corrigir a alocação de processadores efectuada tentando atribuir mais processadores às tarefas que têm um maior *b-level*. O cálculo do *b-level* referido nas linhas 15 e 23, usa o $t^*_{i,p}$ para as tarefas dos níveis seguintes (as que ainda não foram escalonadas) e para as tarefas do nível actual, usa o tempo calculado nas linhas 14 e 22 respectivamente. O algoritmo termina esta operação, se a tarefa com o *b-level* mínimo se torna máximo durante o processo de minimização. Na linha 22, o tempo de execução das tarefas r e k é reavaliado considerando o novo conjunto de processadores atribuídos, o qual resultou da transferência de um processador da tarefa r para a tarefa k.

A Figura 3.4 o mostra mapa do escalonamento de uma aplicação com 10 tarefas, realizado pelo algoritmo HPTS, num *cluster* homogéneo constituído por 10 processadores. Verifica-se, que a execução paralela das tarefas, permite a utilização dos vários processadores do *cluster*, contudo, existem momentos que nem toda a capacidade disponível é utilizada.

A complexidade temporal do HPTS é também $O(T^2 \times P)$ [BMNM05], onde T é o número de tarefas e P o número de processadores.

3.2.3 Algoritmo iHPTS

Através do estudo e análise algoritmo HPTS, verifica-se que, em situações de elevada carga do *cluster*, a capacidade reservada S'_i para cada tarefa i pode ser pequena pois

$S'_i = (S_{max}/\sum_j S^*_j)S^*_i$, podendo ser mesmo inferior à capacidade dos processadores que constituem o *cluster*. Assim, quando o HPTS aloca um processador a uma tarefa, como a capacidade deste é superior à reservada para a tarefa, ou seja, $S_i < S'_i$, mais nenhum processador é considerado. Quando esta situação se verifica, nada garante que o processador atribuído seja aquele que possibilita o EFT da tarefa. Este problema surge quando se considera um *cluster* heterogéneo, onde o tempo de execução das tarefas varia segundo o processador em que são executadas.

Propõem-se então uma alteração ao HPTS que, como mostrado na secção de resultados, melhora o tempo de execução global, quando se está perante um *cluster* heterogéneo com elevada carga.

Algoritmo: iHPTS

...
11. Enquanto tarefas prontas $\neq \oslash$
12. Selecciona a tarefa com o mínimo tl
13. Selecciona os processadores que permitem tl
14. Enquanto $S_i < S'_i$ and $t_{i,p} < t_{i,p-1}$ adiciona processador
15. Se tarefa tem um processador
16. Desaloca o processador da tarefa
17. Para cada processador p_k do conjunto de processadores
18. Calcula EFT_{p_k}
19. Atribui tarefa ao processador p_j que minimiza o seu EFT
...

A alteração que se verifica entre o iHPTS e o HTPS é a introdução das linhas 15 à 19. Após efectuada a atribuição dos processadores, o algoritmo verifica quantos processadores estão alocados à tarefa. No caso de ser apenas um, calcula o EFT da tarefa em todos os processadores do *cluster* e atribui aquele que permite o menor EFT da tarefa.

Embora esta abordagem seja inspirada no HEFT, não se adopta a política de inserção devido à complexidade que iria originar no algoritmo para determinar se o *slot* livre entre duas tarefas consecutivas de um processador, se deve a este estar no estado *idle* ou a estar a ocorrer uma comunicação entre tarefas.

No algoritmo iHPTS, considera-se o mesmo modelo computacional do que o HPTS.

3.3 Tarefas não-paralelas vs paralelas

A carga de cada processador e a capacidade do *cluster* utilizada, é aproveitada de maneira diferente pelos algoritmos HEFT e HPTS, nomeadamente devido a consideração, ou não, de tarefas paralelas. As Figuras 3.3 e 3.4 ilustram o escalonamento efectuado pelo algoritmo HEFT e HPTS, respectivamente, de uma aplicação com 10 tarefas, num *cluster*

homogéneo constituído por 10 processadores. Estas figuras permitem comparar a forma como cada algoritmo utiliza os recursos disponíveis no *cluster*.

O algoritmo HEFT utiliza uma abordagem não paralela, isto é, a cada tarefa é apenas atribuído um processador. Na Figura 3.3 verifica-se que esta abordagem, aliada à estratégia de minimização do tempo de comunicação entre tarefas, resulta na concentração do escalonamento num pequeno número de processadores, não existindo um balanceamento de carga efectivo. Verifica-se também, que o HEFT não utiliza grande parte da capacidade disponível. Assim, a adição ou reserva de mais processadores no *cluster*, não irá influenciar o escalonamento da aplicação, não se conseguindo um menor tempo de execução global.

O algoritmo HPTS utiliza uma abordagem paralela, isto é, a cada tarefa podem ser alocados vários processadores. Esta característica permite um melhor aproveitamento da capacidade disponível no *cluster*, como mostrado na Figura 3.4. Verifica-se também, que a abordagem paralela, aliada à estratégia de escalonamento que é utilizada pelo HPTS, resulta num balanceamento da carga mais eficaz, quando comparado com o HEFT. Contudo, este também não é perfeito, pois a carga não está distribuída de forma equitativa por todos os processadores, não sendo utilizada a total capacidade disponibilizada pelo *cluster* em todos os períodos de tempo. Isto pode nunca ser atingido devido à natureza das aplicações que são escalonadas, ou seja, às restrições impostas pelas precedências entre tarefas.

O número de tarefas que constituem uma aplicação e a forma como as suas precedências estão definidas, pode influenciar a utilização de um *cluster*. Em [BM08] é analisada a distribuição de carga obtida pelo HEFT e HPTS num *cluster* homogéneo com 20 processadores. São considerados 3 DAGs, com 10, 30 e 90 tarefas. O DAG com 10 tarefas é o apresentado em [THW02], e os dois restantes foram gerados aleatoriamente utilizando o algoritmo apresentado em [SSM+04].

A Figuras 3.5 e 3.6 mostram a distribuição da carga pelos processadores, obtida através do escalonamento das três aplicações pelo algoritmo HEFT e HPTS, respectivamente. Verifica-se que a utilização do algoritmo HEFT, apenas atribui uma carga significativa a 3 processadores no escalonamento da aplicação com 10 tarefas. Para a aplicação com 30 tarefas, apenas 4 processadores são utilizados, mas com uma carga bastante diferente. No escalonamento da aplicação com 90 tarefas, 5 processadores não são utilizados e outros 5 têm uma carga bastante pequena, não sendo utilizada na maior parte do tempo metade das máquinas disponíveis. Desta forma, o escalonamento de uma aplicação com um elevado número de tarefas, não garante a utilização de toda a capacidade disponível no *cluster*. A utilização do algoritmo HPTS revela uma melhor distribuição da carga pelos processadores e consequentemente um melhor aproveitamento dos recursos disponibilizados pelo *cluster*, em cada uma das aplicações consideradas.

O algoritmo HEFT ao aproveitar apenas parte da capacidade disponível de um *cluster*

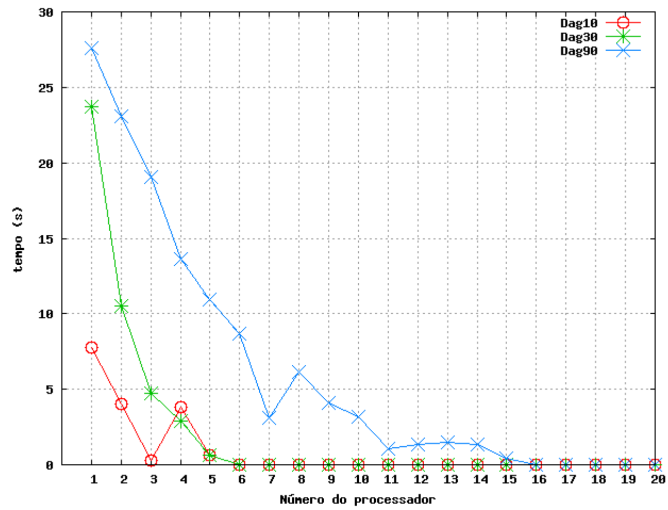

Figura 3.5: Carga de cada processador do cluster - Usando o algoritmo HEFT

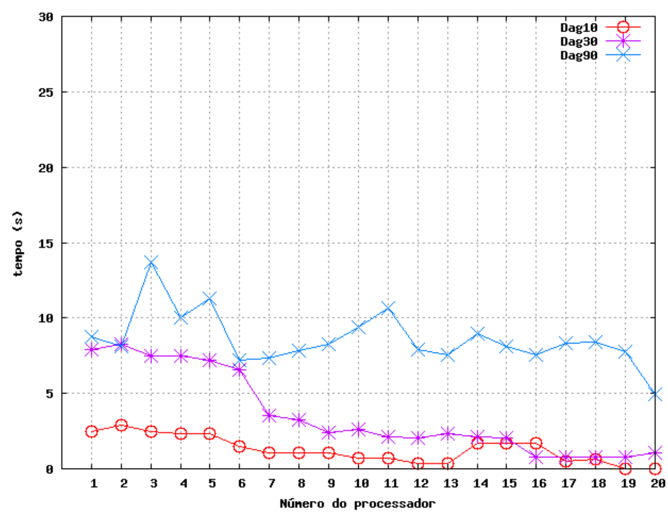

Figura 3.6: Carga de cada processador do cluster - Usando o algoritmo HPTS

permite que novas aplicações sejam escalonados, utilizando os recursos livres. Por outro lado, a flexibilidade que é introduzida pela abordagem paralela do algoritmo HPTS, possibilita que novas aplicações sejam adicionadas através da redução do número de processadores alocados a cada tarefa e aproveitando os recursos não utilizados. Desta forma pode-se atingir uma melhor taxa de utilização dos recursos que são disponibilizados por um *cluster*.

Capítulo 4

Implementação

Neste capítulo são apresentadas as duas ferramentas que este trabalho deu origem, o *jScheduler* e o *jEditor*. O *jScheduler* implementa o método de escalonamento desenvolvido e os dois algoritmos de escalonamento seleccionados, o HPTS e o HEFT. Esta ferramenta permite também a gestão da execução de cada uma das tarefas num *cluster*. O *jEditor* é uma ferramenta que possibilita a criação de um DAG, através da definição das características de cada uma das suas tarefas, através de uma interface gráfica. Esta ferramenta possibilita também submeter remotamente *jobs* para um *cluster*. Estas duas ferramentas embora relacionadas, não têm de ser utilizadas em conjunto.

Para a implementação de ambas as ferramentas foi utilizada a linguagem de programação orientada a objectos JAVA, devido à sua característica multi-plataforma, permitindo a sua utilização nos mais variados sistemas, sem ser necessário a sua adaptação e recompilação.

4.1 Ferramenta jScheduler

O *jScheduler* foi desenvolvido de forma a implementar o método de escalonamento dinâmico proposto neste trabalho e descrito no Capítulo 3. Para ser possível a realização de testes, foi também necessário a implementação dos dois algoritmos de escalonamento seleccionados, nomeadamente o HEFT, o HPTS, e da proposta de optimização do HPTS, designada por iHPTS.

Através do *jScheduler*, para além da implementação do método de escalonamento, pretende-se também mostrar a sua aplicabilidade num *cluster*, através da gestão da execução das tarefas das diferentes aplicações. Assim, esta ferramenta não é apenas um escalonador mas também um simples gestor de recursos.

O *jScheduler* considera que a execução de tarefas paralelas, como definido nos algoritmos HPTS e iHPTS, utiliza a especificação MPI. O MPI (Message Passing Interface)

é a especificação para um modelo de programação que se baseia na troca de mensagens, sendo a sincronização e o modo de funcionamento da aplicação da responsabilidade do programador. O MPI é o modelo dominante na computação de alto desempenho, tendo como principais objectivos: aumentar a portabilidade dos programas; aumentar e melhorar a funcionalidade; conseguir implementações eficientes numa vasta gama de arquitecturas; suportar ambientes heterogéneos.

Considera-se que os utilizadores desta ferramenta serão os actuais utilizadores de um *cluster*, passando a dispor de um outro recurso para efectuar o escalonamento e execução das suas aplicações.

A interface com o utilizador do *jScheduler* é em modo texto, sendo utilizado um conjunto de comandos para a inserção de aplicações. Uma aplicação para ser interpretada pelo *jScheduler* necessita de estar descrita através de um ficheiro XML, onde estão especificadas todas as características das tarefas que a constituem. Estes ficheiros podem ser facilmente criados através do *jEditor*.

O resultado do escalonamento e de execução de cada uma das tarefas é disponibilizado aos utilizadores através de um conjunto de ficheiros de texto, para posterior análise.

O *jScheduler* permite que as aplicações sejam submetidas remotamente através da ferramenta *jEditor*.

4.1.1 Requisitos Funcionais

O *jScheduler* deve aceitar aplicações submetidas por utilizadores e ser responsável pelo seu escalonamento e execução, segundo o método que implementa. Na Figura 4.1 é apresentado um diagrama de actividades que descreve o fluxo das operações realizadas, resultante da adição de uma aplicação.

Um novo ciclo de escalonamento é iniciado quando um utilizador insere um *job* no *cluster* através do *jScheduler*. A ferramenta verifica se existem tarefas já escalonadas e que ainda não foram executadas. Se não existirem tarefas por executar, o *jScheduler* faz o escalonamento das tarefas do novo *job* iniciando de imediato a sua execução. Caso contrário, se existirem tarefas por executar, não é permitida a execução de novas tarefas, e o escalonamento anterior é eliminado. Todas as tarefas que ainda não executaram, juntamente com as que constituem o novo *job*, são seleccionadas e escalonadas, tendo em conta a disponibilidade dos processadores, devido à possibilidade de alguns estarem ocupados a executar tarefas já iniciadas antes da inserção do novo *job*. Seguidamente, é iniciada a execução das tarefas segundo o escalonamento efectuado.

A gestão da execução das tarefas é um requisito funcional de forma a mostrar a aplicabilidade do método. O início da execução de uma nova tarefa é controlada pelo *jScheduler*.

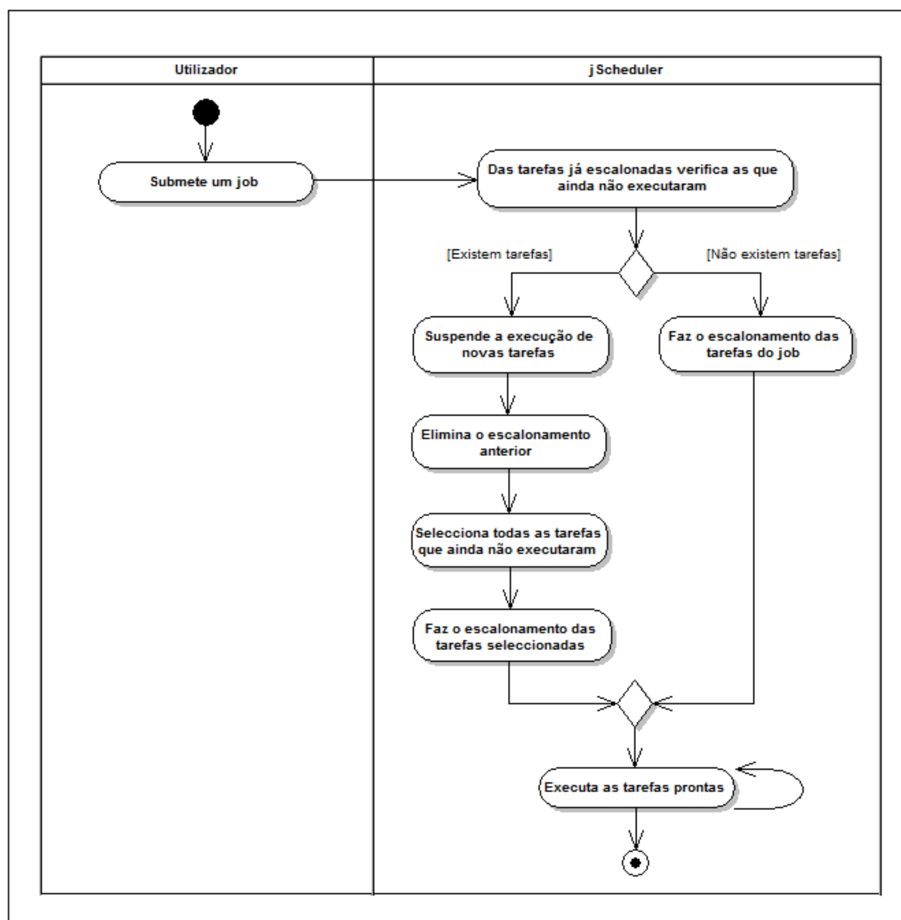

Figura 4.1: Diagrama de actividades do caso de utilização - Submeter um job

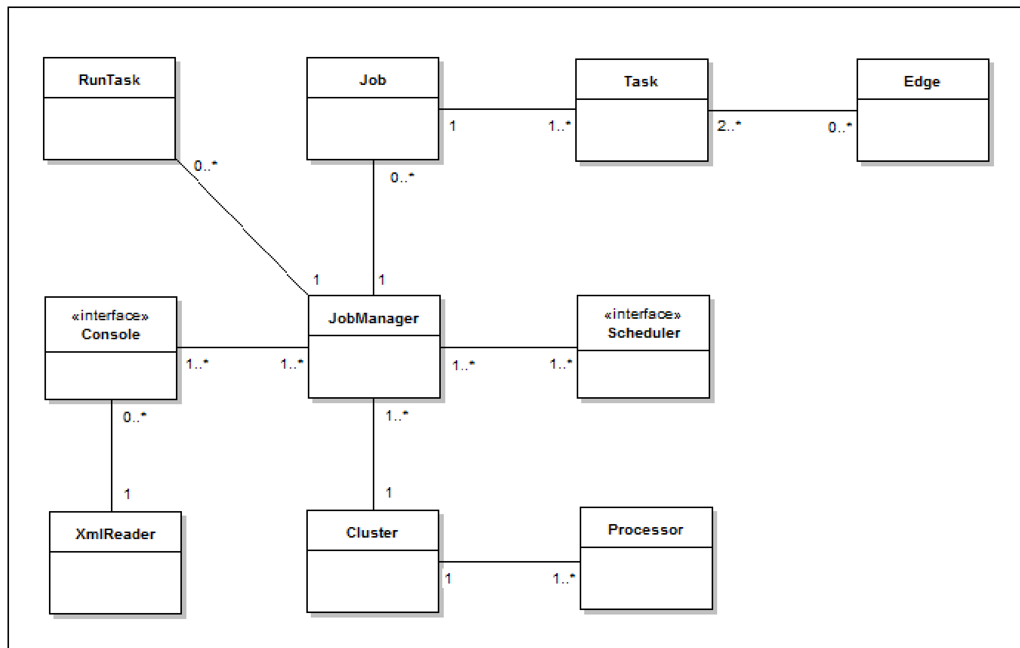

Figura 4.2: Diagrama de classes da ferramenta jScheduler

4.1.2 Diagrama de Classes

O diagrama de classes da ferramenta *jScheduler* é apresentado na Figura 4.2, sendo realizada uma pequena descrição que caracteriza cada uma das classes.

No Anexo C é efectuada uma descrição dos métodos implementados em cada classe.

4.1.2.1 Classe *Task*

Cada instância da classe *Task* representa uma tarefa de uma aplicação, ou seja, um nó de um DAG. Os atributos de cada instância caracterizam a tarefa, sendo usados para efectuar o seu escalonamento. Cada tarefa está associada a um *job* e pode ter relações de precedência com outras tarefas. As relações de precedência são definidas através de instâncias da classe *Edge*.

4.1.2.2 Classe *Edge*

Uma instância da classe *Edge* representa uma aresta entre dois nós de um DAG, ou seja, representa uma relação de precedência entre duas tarefas. Uma aresta tem uma tarefa inicial e outra final, definindo desta forma o seu sentido. Cada instância desta classe tem também associado o custo de comunicação entre as duas tarefas. Entre um par de tarefas apenas pode existir uma aresta.

4.1.2.3 Classe *Job*

Uma instância da classe *Job* representa uma aplicação que foi submetida por um utilizador, para ser escalonada e executada. Cada objecto desta classe contem como atributos as tarefas que constituem uma aplicação e as outras características relevantes para a caracterizar, como por exemplo, o utilizador, data de inserção, etc.

4.1.2.4 Classe *Processor*

Cada objecto da classe *Processor* representa um processador do sistema de computação. Cada processador é caracterizado pela sua capacidade de processamento e pelo seu endereço de localização.

4.1.2.5 Classe *Cluster*

Um objecto da classe *Cluster* caracteriza-se por representar o sistema de computação considerado. Uma instância desta classe é caracterizada por um conjunto de processadores e pelas características da rede de comunicações que os ligam.

4.1.2.6 Classe *XmlReader*

A classe *XmlReader* é responsável por interpretar os ficheiros XML que contêm a descrição de uma aplicação, nomeadamente, as tarefas e suas características. Os ficheiros XML podem ser facilmente criados com a utilização da ferramenta *jEditor*.

4.1.2.7 Interface *Console*

A interface *Console* define a estrutura que terá ser implementada pelas classes que sejam responsáveis pela interacção com os utilizadores. As classes que implementam esta interface devem permitir a introdução de aplicações no *jScheduler*, para escalonamento e execução. Foi tomada a opção de utilizar uma interface, de forma a possibilitar a fácil criação de novas classes que suportem a interacção com o utilizador. No *jScheduler* foram implementadas soluções para a introdução de *jobs* localmente, através da consola, e remotamente, através da ferramenta *jEditor*.

4.1.2.8 Interface *Scheduler*

A interface *Scheduler* define a estrutura das classes que implementam cada um dos algoritmos de escalonamento. No *jScheduler* foram implementados 2 algoritmos de escalonamento nomeadamente, o HPTS, o HEFT e realizada a optimização designada por iHPTS.

4.1.2.9 Classe *RunTask*

Cada instância da classe *RunTask* representa o *thread* responsável por controlar a execução de uma tarefa. Cada objecto desta classe, é responsável por iniciar a execução de uma tarefa, no(s) processador(es) que lhe estão alocados. É também responsável por detectar o fim de execução de uma tarefa.

4.1.2.10 Classe *JobManager*

A classe *JobManager* é responsável pela gestão do escalonamento das tarefas de todas as aplicações. A estratégia de escalonamento, que define o momento em que o algoritmo de escalonamento é invocado, está implementada nesta classe. Além disso, esta classe é também responsável por conservar o estado de execução de cada uma das aplicações.

4.2 Ferramenta jEditor

A definição de cada uma das aplicações que são escalonadas no *jScheduler* é realizada através de um ficheiro XML. Contudo, quando se está perante aplicações com um elevado número de tarefas que têm muitas relações de precedência, a criação desse ficheiro XML torna-se bastante demorada e sujeita a erros. Surge assim a necessidade de uma nova ferramenta, o *jEditor*.

O *jEditor* é uma ferramenta desenvolvida de forma a facilitar a definição de uma aplicação para ser escalonada e executada no *jScheduler*. Com esta ferramenta é possível a criação de um DAG, onde o utilizador define as características de cada uma das tarefas e as suas relações de precedência, através de uma interface gráfica, como se pode ver no Anexo B.

A interface gráfica do *jEditor* é constituída por três áreas: área de desenho; barra de ferramentas; e propriedades dos objectos. O utilizador tem ao seu dispor um conjunto de opções que permite a criação de um DAG de forma visual, como de uma ferramenta de desenho se tratasse. O trabalho pode ser guardado para posterior edição.

Através da interface gráfica também é possível submeter remotamente uma aplicação para o *jScheduler*, através da utilização de *sockets*, eliminando desta forma a necessidade da utilização da linha de comando.

4.2.1 Requisitos Funcionais

O *jEditor* foi desenvolvido de forma a permitir que um utilizador crie e edite DAGs que representam *jobs*, utilizando um processo de desenho simples. Também se teve como requisito a submissão, através de uma interface gráfica, de uma aplicação. Na Figura 4.3 é apresentado o diagrama de casos de utilização do *jEditor*.

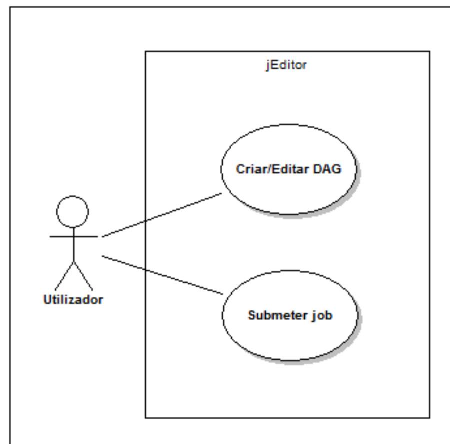

Figura 4.3: Diagrama de classes da ferramenta jEditor

O caso de utilização *criar/editar DAG*, engloba toda a vertente de criação e edição de um DAG, nomeadamente a adição de tarefas e das suas relações de precedência. A gravação do DAG no formato XML que é interpretado pelo *jScheduler* é um requisito fundamental.

A submissão remota de uma aplicação através desta aplicação pretende facilitar a interacção com o *jScheduler*, devido à utilização de uma interface gráfica.

4.2.2 Diagrama de Classes

O diagrama de classes da ferramenta *jEditor* é apresentado na Figura 4.4, sendo efectuada uma pequena descrição que caracteriza cada uma das classes.

4.2.2.1 Classe *Task*

A classe *Task* representa as tarefas de uma aplicação. Cada desta classe *Task* representa uma tarefa de uma aplicação, ou seja, um nó de um DAG. Os atributos de cada instância caracterizam a tarefa, sendo inseridos pelo utilizador. Cada tarefa está associada a um *JobDag* e pode ter relações de precedência com outras tarefas.

4.2.2.2 Classe *JobDag*

Uma instância da classe *JobDag* representa uma aplicação, ou seja um DAG. Este é caracterizado através de um conjunto de tarefas.

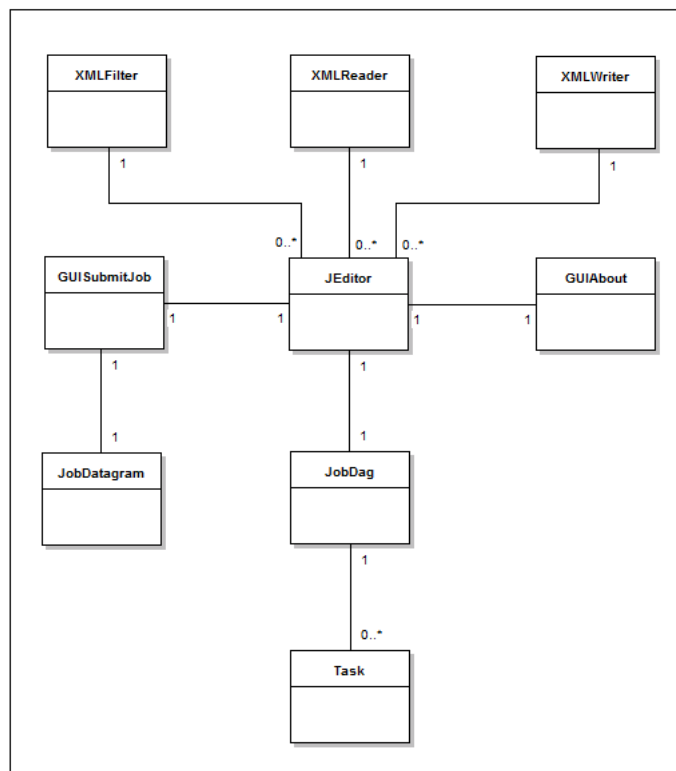

Figura 4.4: Diagrama de classes da ferramenta jEditor

4.2.2.3 Classe *JobDatagram*

A classe *JobDatagram* representa um *job* submetido remotamente. Cada instância desta classe contém a informação necessária a ser enviada remotamente para o *JScheduler* para submeter uma aplicação.

4.2.2.4 Classe *JEditor*

A classe *JEditor* implementa toda a interface gráfica da ferramenta, nomeadamente as ferramentas desenho dos DAGs.

4.2.2.5 Classe *GUISubmitJob*

Esta classe implementa a interface da janela de diálogo utilizada para submeter um *job* remotamente no *jScheduler*. A janela de diálogo contém as opções que são necessárias preencher.

4.2.2.6 Classe *GUIAbout*

Esta classe implementa uma janela de diálogo com os créditos da ferramenta.

4.2.2.7 Classe *XMLFilter*

A classe *XMLFilter* é responsável por verificar e filtrar os ficheiros que serão exibidos nas caixas de diálogo *Open* e *Save* do *jEditor*. São apenas exibidos os ficheiros com extensão *xml*.

4.2.2.8 Classe *XMLReader*

Esta classe é responsável por interpretar os ficheiros XML que contêm a descrição de uma aplicação. Isto permite que o *jEditor* abra os ficheiros para posterior edição.

4.2.2.9 Classe *XMLWriter*

Esta classe é responsável por escrever os ficheiros XML que contêm a descrição de uma aplicação. Isto permite que o *jEditor* guarde os ficheiros para posterior edição.

Capítulo 5

Resultados e Discussão

Neste capítulo é avaliado o desempenho do método de escalonamento dinâmico proposto através da comparação com duas abordagens normalmente utilizadas, nomeadamente:

- Atribuir um processador a cada aplicação;
- A cada aplicação é reservado um número fixo de processadores, exclusivos durante toda a sua execução.

A avaliação do desempenho do método de escalonamento dinâmico proposto, foi efectuada através da comparação do *makespan* de um conjunto de aplicações, com os obtidos através das outras duas abordagens, nas mesmas condições.

Os algoritmos de escalonamento utilizados foram o HEFT e HPTS, e a optimização iHPTS descritos no Capítulo 3.

As ferramentas que foram criadas no âmbito deste trabalho, nomeadamente o *jScheduler* e o *jEditor* foram utilizadas nos testes efectuados. Para verificar a aplicabilidade do *jScheduler*, este foi testado num *cluster* disponível localmente. No entanto, os resultados apresentados foram obtidos através da simulação do ambiente de execução, baseados em medições num *cluster*, como em [BM08]. Foi utilizado um ambiente simulado para se obter um melhor controlo sobre as variáveis, os resultados não serem influenciados por factores externos e ser possível utilizar ambientes de execução distintos, que não estavam disponíveis localmente.

Foram considerados seis *clusters* com diferentes características, nomeadamente, número de processadores, capacidade de processamento e nível de heterogeneidade. Isto permite analisar os resultados e comparar o desempenho do método de escalonamento dinâmico proposto em diferentes ambientes de execução.

O procedimento para a estimativa das comunicações e tempos de execução que se utilizou foi o analisado e discutido em [BTP00];

5.1 Sistema de Computação

A avaliação do método de escalonamento dinâmico é realizada com o recurso a vários sistemas de computação, *clusters*, para se poder analisar o seu comportamento em diferentes ambientes de execução, nomeadamente em *clusters* homogéneos e heterogéneos. São considerados dois *clusters* homogéneos e quatro heterogéneos.

Os *clusters* homogéneos e heterogéneos considerados são constituídos por 10 e 20 processadores, de forma a avaliar o comportamento do método proposto em diferentes situações de disponibilidade de recursos.

No caso homogéneo considera-se que a capacidade relativa de cada um dos processadores é 1. Nas Tabelas 5.1 e 5.2 está representada a capacidade relativa que corresponde aos processadores que constituem os *clusters* heterogéneos.

Capacidade Relativa	1	0.75	0.5
Processadores do cluster C1	4	3	3
Processadores do cluster C2	8	6	6

Tabela 5.1: Capacidade de processamento relativa e número de processadores dos clusters C1 e C2

Capacidade Relativa	1	0.5	0.25	0.125	0.0625
Processadores do cluster C3	2	2	2	2	2
Processadores do cluster C4	4	4	4	4	4

Tabela 5.2: Capacidade de processamento relativa e número de processadores dos clusters C3 e C4

Em todos os *clusters* considera-se que os seus processadores estão interligados através de uma rede de comunicações *Ethernet*, com 1*Gbit* de largura de banda.

A rede deve permitir comunicações simultâneas entre os diferentes pares de máquinas. Esta é uma característica fundamental para a execução de tarefas paralelas, devido à necessidade de troca de informação dentro de um grupo de processadores que está a executar uma tarefa. No caso geral, quando se efectua a estimativa de comunicações envolvidas, tem de se garantir que dentro do grupo de processadores que processam uma tarefa, não existem conflitos de comunicação. Caso contrário, seria extremamente difícil sincronizar as comunicações, de diferentes tarefas paralelas, sem surgirem conflitos.

5.1.1 Heterogeneidade

Neste trabalho considera-se que um *cluster* é heterogéneo, quando os processadores que o constituem têm diferentes capacidade de processamento. A avaliação da heterogeneidade de cada um dos *clusters* é realizada através do cálculo do coeficiente de variação V, onde V é o quociente entre o desvio padrão e a média, dos tempos de execução de cada tarefa nos diferentes processadores, como descrito no Capítulo 2.

Foram considerados quatro *clusters* heterogéneos, C1, C2, C3, C4, caracterizados nas Tabelas 5.1 e 5.2.

Para a avaliação da heterogeneidade de cada um dos *clusters*, é necessário calcular os tempos de execução de cada uma das tarefas de um *job* em cada um dos processadores disponíveis, como exemplificado pelas Tabelas 5.3 e 5.4, que se referem aos *clusters* C1 e C3, respectivamente. A partir destas tabelas é possível calcular o valor V do *cluster* C1 e C3, respectivamente. Para o cálculo de V em C2 e C4 foram utilizadas tabelas semelhantes.

O cálculo do valor V é efectuado linha a linha, sobre as tabelas com os tempos de execução, quantificando-se desta forma a heterogeneidade relativa aos processadores do *cluster*.

Neste trabalho, como em [KSS$^+$07], considera-se que um *cluster* é pouco heterogéneo se $V = 0,3$ e muito heterogéneo se o $V = 0,9$. Assim, os *clusters* C1 e C2, são considerados pouco heterogéneos e os C3 e C4, são considerados muito heterogéneos.

A distribuição de capacidade pelos processadores de C1, C2, C3, e C4 foi feita de forma aleatória de até se obter os valores desejados para V.

Neste trabalho considera-se que todos os *clusters* são consistentes, como descrito no Capítulo 2.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
t_1	70,4	70,4	70,4	70,4	94,8	94,8	94,8	142,2	142,2	142,2
t_2	24,8	24,8	24,8	24,8	33,4	33,4	33,4	50,0	50,0	50,0
t_3	4,4	4,4	4,4	4,4	5,9	5,9	5,9	8,8	8,8	8,8
t_4	25,3	25,3	25,3	25,3	34,1	34,1	34,1	51,2	51,2	51,2
t_5	209,0	209,0	209,0	209,0	281,5	281,5	281,5	422,2	422,2	422,2
t_6	43,0	43,0	43,0	43,0	57,9	57,9	57,9	86,8	86,8	86,8
t_7	25,6	25,6	25,6	25,6	34,5	34,5	34,5	51,7	51,7	51,7
t_8	24,3	24,3	24,3	24,3	32,8	32,8	32,8	49,2	49,2	49,2
t_9	3,9	3,9	3,9	3,9	5,3	5,3	5,3	8,0	8,0	8,0
t_{10}	27,2	27,2	27,2	27,2	36,7	36,7	36,7	55,0	55,0	55,0

Tabela 5.3: Tempos de execução das tarefas de uma aplicação em cada um dos processadores do cluster C1

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
t_1	35,6	35,6	71,1	71,1	142,2	142,2	284,5	284,5	569,0	569,0
t_2	12,5	12,5	25,0	25,0	50,0	50,0	100,1	100,1	200,2	200,2
t_3	2,2	2,2	4,4	4,4	8,8	8,8	17,6	17,6	35,2	35,2
t_4	12,8	12,8	25,6	25,6	51,2	51,2	102,4	102,4	204,8	204,8
t_5	105,5	105,5	211,1	211,1	422,2	422,2	844,4	844,4	1688,7	1688,7
t_6	21,7	21,7	43,4	43,4	86,8	86,8	173,6	173,6	347,2	347,2
t_7	12,9	12,9	25,9	25,9	51,7	51,7	103,4	103,4	206,9	206,9
t_8	12,3	12,3	24,6	24,6	49,2	49,2	98,4	98,4	196,7	196,7
t_9	24,3	24,3	24,3	24,3	32,8	32,8	32,8	49,2	49,2	49,2
t_{10}	12,3	12,3	24,6	24,6	49,2	49,2	98,4	98,4	196,7	196,7

Tabela 5.4: Tempos de execução das tarefas de uma aplicação em cada um dos processadores do cluster C3

5.2 Aplicações

Uma aplicação é constituída por várias tarefas dependentes, podendo ser representada através de um DAG. Neste trabalho foram consideradas doze aplicações geradas aleatoriamente através do algoritmo apresentado em [SSM+04]. Este algoritmo é utilizado na geração de DAGs e considera que: existem Na nós que não têm antecedentes, apenas têm sucessores, isto é, nós iniciais com *Ids* de 1 até Na; Nb nós, simultaneamente com nós antecedentes e sucessores, com valores de *Ids* a variar entre $Na+1$ até $Na+Nb$, inclusive; Nc nós, apenas com nós antecedentes, isto é, nós finais com *Ids* que variam entre $Na+Nb+1$ até $Na+Nb+Nc$, inclusive. Considera-se que os valores de Na e Nb variam entre 1 e 4, e Nb é igual número dos restantes nós. O mínimo e o máximo de ligações que um nó tem para com os outros, *nodedegree*, é de 2 e 5 respectivamente. Também se considera que o sentido das arestas é dos nós com menor *Id* para os maiores.

Os DAGs gerados aleatoriamente através do método definido em [SSM+04] representam uma colecção de aplicações independentes de um ou vários utilizadores.

Neste trabalho as tarefas de cada uma das aplicações são operações de álgebra linear, nomeadamente, factorização triangular (TRD), cálculo da matriz Q, iteração QR e correlação C.

A dimensão dos dados de entrada de cada tarefa foi gerada aleatoriamente, sendo um valor pertencente ao intervalo [200, 1200].

Os tempos de processamento estimados são baseados em valores reais medidos nos processadores considerados, como em [BM08]. A Tabela 5.5 mostra os valores relativos de computação e comunicação de cada uma das operações algébricas consideradas [BMNM05]. As arestas do DAG representam as relações de precedência e os custos de comunicação entre as tarefas. Neste trabalho considera-se que o CCR (computation to communication ratio) tem o valor de 0,3, isto é, o custo de comunicação entre as tarefas definido por cada aresta é 30% do custo de computação da tarefa anterior.

Tipo de Tarefa	TRD	Q	QR	C
Custo relativo de computação	1	0.82	2	3
Custo relativo de comunicação	1	0.125	0.25	0.50

Tabela 5.5: Custos relativos de computação e comunicação das operações algébricas consideradas

5.3 Resultados Experimentais

A avaliação do desempenho do método dinâmico de escalonamento proposto neste trabalho, foi realizada através da comparação com as seguintes abordagens:

1. Atribuir um processador a cada aplicação;
2. A cada aplicação é reservado um número fixo de processadores, exclusivos durante toda a sua execução.

O primeiro caso corresponde ao escalonamento dinâmico de aplicações independentes constituídas por apenas uma tarefa, como referido em [KSS+07, SZI07]. O segundo caso, corresponde aos simples escalonadores de DAGs independentes, que recebem o número de processadores a alocar ao *job* através de um parâmetro fornecido pelo utilizador. A principal diferença entre o método de escalonamento proposto e as duas abordagens anteriores é este considera as tarefas das aplicações como um todo, logo não reserva nem fixa processadores a aplicações, sendo estes atribuídos às tarefas que estão prontas a executar.

Para a avaliação do método de escalonamento dinâmico, são consideradas doze aplicações, que totalizam 200 tarefas. A frequência de chegada de uma nova aplicação é de 10 segundos.

Nos gráficos apresentados, considera-se que, HPTS, iHPTS e o HEFT correspondem à execução do método dinâmico proposto, utilizando os algoritmos de escalonamento referidos. O HPTS5P e o HEFT5P representam a abordagem de reserva fixa e exclusiva de processadores para cada uma das aplicações, neste caso 5 processadores, utilizando os algoritmos de escalonamento referidos. O HPTS10P, HEFT10P HPTS20P e HEFT20P têm o mesmo significado dos anteriores, só que neste caso são reservados 10 e 20 processadores, respectivamente. Por fim, considera-se que 1J-1P corresponde à atribuição de um processador a cada uma das aplicações.

As tabelas apresentadas mostram a média e o desvio padrão do tempo de execução do conjunto de aplicações, relativas ao algoritmo utilizado em todas as estratégias de escalonamento testadas, nos diferentes *clusters* considerados. Estas tabelas permitem verificar a regularidade de cada um dos algoritmos de escalonamento. A optimização designada por iHPTS não é considerada, pois a sua implementação foi apenas efectuada

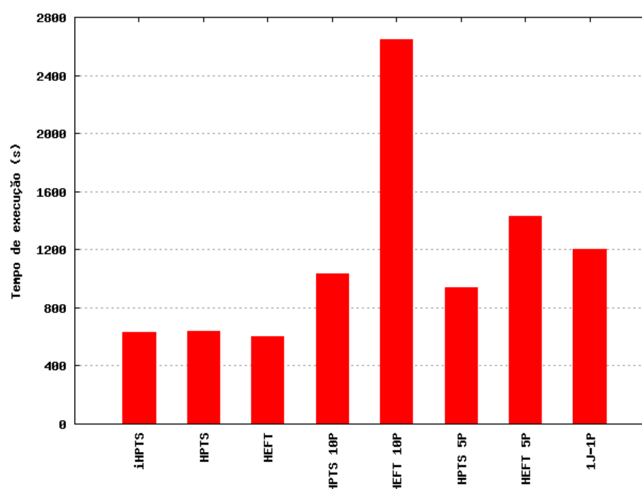

Figura 5.1: Tempo global de escalonamento num cluster homogéneo em 10 processadores

para avaliar e comparar o seu desempenho no método de escalonamento desenvolvido, não tendo sido efectuados testes com outras estratégias de escalonamento.

5.3.1 Sistema de Computação Homogéneo

Para o sistema de computação homogéneo são considerados dois *clusters* com 10 e 20 processadores. Os resultados obtidos nestes dois *clusters* são mostrados através das Figuras 5.1 e 5.2.

Verifica-se que o método de escalonamento proposto, permite atingir um melhor desempenho do que as outras duas abordagens. A reserva fixa e exclusiva de processadores penaliza o desempenho de ambos os *clusters*, utilizando qualquer um dos algoritmos de escalonamento considerados. Verifica-se também que a atribuição de 1 processador a cada aplicação, não se mostra como uma boa opção.

A utilização do iHPTS, não mostra uma melhoria significativa no desempenho quando comparado com a utilização do HPTS.

Quando se utiliza 10 processadores o algoritmo HEFT apresentou ligeiramente um melhor desempenho do que o HPTS, quando aplicados no método de escalonamento proposto. Contudo a situação inverte-se no caso da utilização de 20 processadores. Esta situação verifica-se pois quando existem poucos processadores disponíveis para muitas tarefas o HPTS não tira vantagem da sua principal característica, de considerar que as tarefas podem executar de forma paralela.

As outras duas abordagens de escalonamento, têm um comportamento semelhante nos dois *clusters*.

Quando se analisa o comportamento dos algoritmos de escalonamento em todas as abordagens consideradas, de uma forma geral, verifica-se o HEFT apresenta um pior

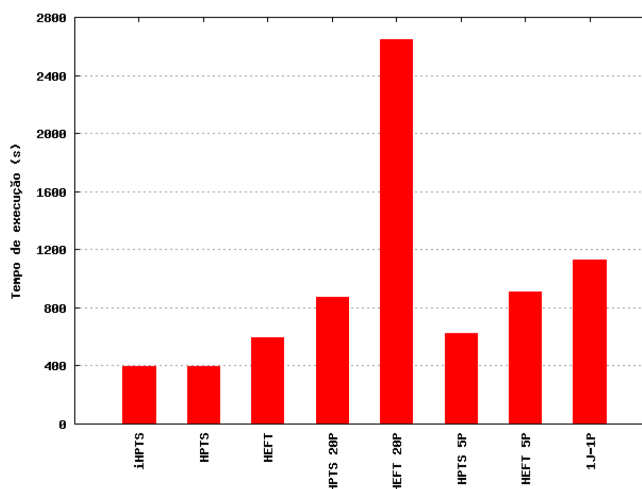

Figura 5.2: Tempo global de escalonamento num cluster homogéneo em 20 processadores

desempenho, quando comparado com o HPTS. A utilização do HPTS como algoritmo de escalonamento, garante um desempenho mais regular como se mostra na Tabela 5.6. Esta tabela compara os valores da média e do desvio padrão dos tempos de escalonamentos em relação ao algoritmo utilizado.

Cluster	Algoritmo	Média (s)	Desvio Padrão (s)
10 Processadores	HPTS	871	209.5
	HEFT	1426	1026.1
	1J-1P	1713	0
20 Processadores	HPTS	631	237.0
	HEFT	1382	1104.4
	1J-1P	1484	0

Tabela 5.6: Média e Desvio Padrão dos escalonamentos em relação ao algoritmo utilizado, num cluster homogéneo

5.3.2 Sistema de Computação Heterogéneo

As experiências realizadas num ambiente heterogéneo foram efectuadas considerando os quatro *clusters*, C1, C2, C3 e C4, caracterizados na secção 5.1.

Inicialmente, é avaliado o desempenho do método proposto num ambiente pouco heterogéneo, utilizando os *clusters* C1 e C2, com 10 e 20 processadores, respectivamente. Os resultados das experiências realizadas nestes dois *clusters* são mostrados através das Figuras 5.3 e 5.4.

Através da análise dos resultados obtidos, verifica-se que o método de escalonamento proposto apresenta melhores resultados, do que as outras duas abordagens consideradas, sendo esta diferença de desempenho, em alguns casos, bastante significativa.

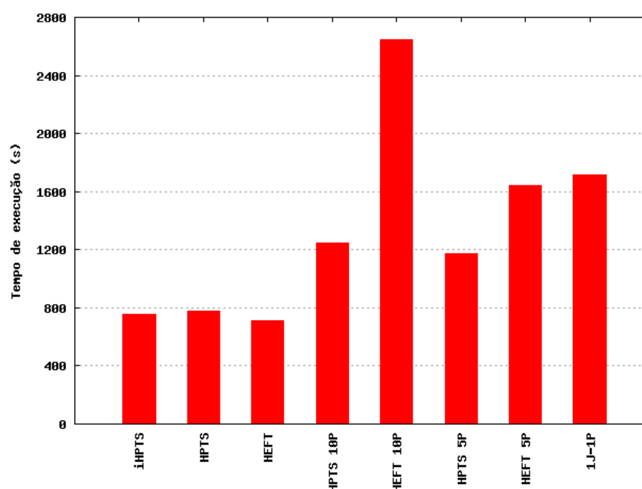

Figura 5.3: Tempo global de escalonamento num cluster com baixa heterogeneidade em 10 processadores

A utilização do iHPTS, neste ambiente heterogéneo, permite uma ligeira melhoria no desempenho quando comparado com a utilização do HPTS.

Apesar da diferença do número de processadores considerada em cada *cluster*, o comportamento do método proposto e das outras duas abordagens é bastante semelhante. Nota-se, no entanto, que no método proposto o algoritmo de escalonamento HEFT tem um melhor desempenho no *cluster* C1, equipado com 10 processadores e que o HPTS tem melhor desempenho no *cluster* C2, equipado com 20 processadores, como já se tinha verificado no ambiente homogéneo.

A análise global ao desempenho dos algoritmos mostra que o HPTS é o algoritmo que apresenta resultados mais regulares, quando comparado com o HEFT, como se mostra na Tabela 5.7. Esta tabela compara os valores da média e do desvio padrão dos tempos de escalonamentos em relação ao algoritmo utilizado.

Cluster	Algoritmo	Média (s)	Desvio Padrão (s)
C1	HPTS	1066	253.3
	HEFT	1666	968.6
	1J-1P	1713	0
C2	HPTS	746	214.8
	HEFT	1564	1027.3
	1J-1P	1484	0

Tabela 5.7: Média e Desvio Padrão dos escalonamentos em relação ao algoritmo utilizado, num cluster pouco heterogéneo

Para completar a avaliação do método dinâmico de escalonamento proposto foi analisado e comparado o seu desempenho num sistema muito heterogéneo, representado pelos

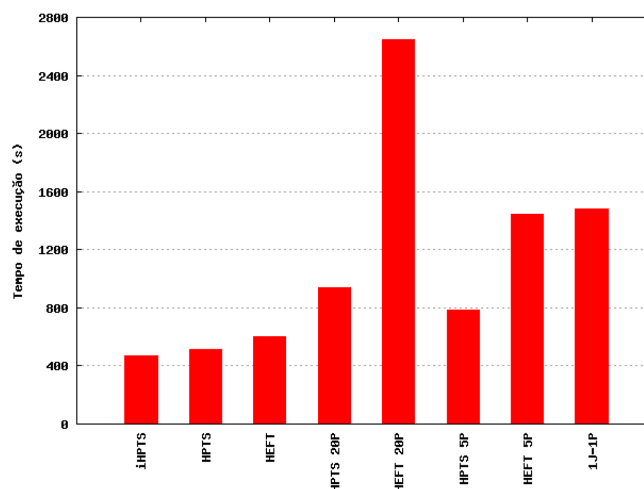

Figura 5.4: Tempo global de escalonamento num cluster com baixa heterogeneidade em 20 processadores

clusters C3 e C4, com 10 e 20 processadores, respectivamente.

As experiências foram realizadas nas mesmas condições das anteriores.

Como verificado nas situações anteriores, também neste caso a abordagem de reserva fixa e exclusiva de processadores a cada uma das aplicações, e a atribuição de um processador a cada aplicação, apresenta piores resultados, quando comparados com o método proposto, seja qual for o algoritmo de escalonamento utilizado. As Figuras 5.5 e 5.6 apresentam os resultados.

A analise do comportamento dos algoritmos de escalonamento, neste ambiente de execução de grande heterogeneidade e com elevada carga, permite verificar que o HEFT apresenta um melhor desempenho nos dois *clusters*, quando comparado com o HPTS, no método de escalonamento proposto. Nas outras duas abordagens o HPTS apresentou sempre um desempenho superior.

No entanto, e como em todas as situações anteriores, quando se analisa o comportamento global dos algoritmos o HPTS apresenta um desempenho mais regular como se mostra na Tabela 5.8, que compara os valores da média e do desvio padrão dos tempos de escalonamentos em relação ao algoritmo utilizado.

A utilização iHPTS, no método de escalonamento proposto, permite uma ligeira melhoria no desempenho quando comparado com o HPTS.

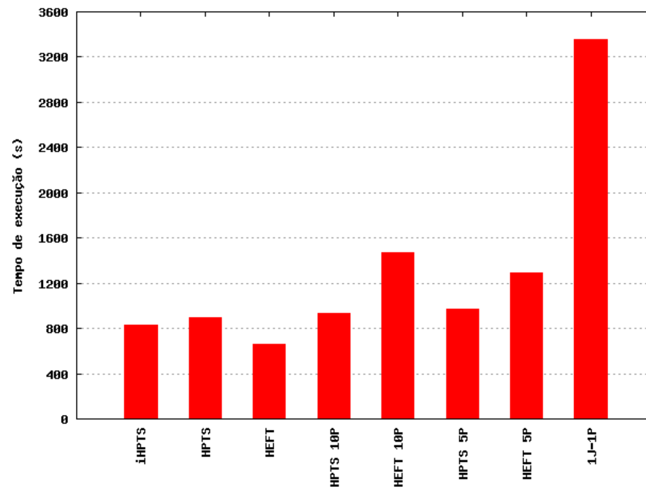

Figura 5.5: Tempo global de escalonamento num cluster com alta heterogeneidade em 10 processadores

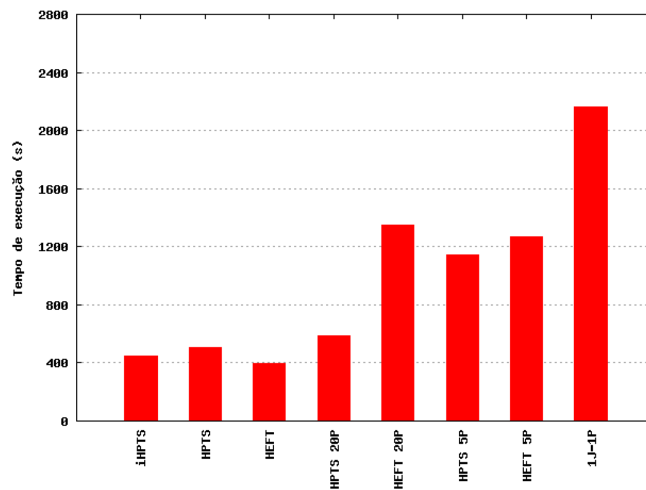

Figura 5.6: Tempo global de escalonamento num cluster com alta heterogeneidade em 20 processadores

Cluster	Algoritmo	Média (s)	Desvio Padrão (s)
C3	HPTS	932	54.4
	HEFT	1138	426.1
	1J-1P	3355	0
C4	HPTS	747	347
	HEFT	1005	528
	1J-1P	2164	0

Tabela 5.8: Média e Desvio Padrão dos escalonamentos em relação ao algoritmo utilizado, num cluster muito heterogéneo

Capítulo 6

Conclusões e Trabalho Futuro

Neste trabalho foi apresentado um método de escalonamento dinâmico, para aplicações constituídas por tarefas dependentes que podem ser representadas através de DAGs. O escalonamento é dinâmico, pois não se conhece a frequência de chegada de cada uma das aplicações, embora os tempos de execução e de comunicação de cada uma das tarefas sejam conhecidos.

O método proposto foi comparado com as abordagens comuns de escalonamento, nomeadamente a atribuição de um processador a cada aplicação, e a reserva fixa e exclusiva de um número de processadores a cada uma das aplicações durante a sua execução.

Estas duas abordagens distinguem-se da implementada no método de escalonamento dinâmico proposto, pois este considera as tarefas disponíveis de todas as aplicações como um todo, não efectuando uma reserva de processadores para cada aplicação.

6.1 Satisfação dos Objectivos

A análise dos resultados obtidos permite concluir que a utilização do método de escalonamento dinâmico permite melhorar de forma significativa o desempenho de um *cluster*, quando comparado com as duas abordagens de reserva de processadores consideradas. Estes resultados verificaram-se tanto para *clusters* homogéneos como para *clusters* com baixo e alto grau de heterogeneidade.

O método de escalonamento dinâmico proposto obteve sempre um melhor desempenho, em todas as situações testadas. A reserva fixa e exclusiva de processadores a cada uma das aplicações não beneficia o desempenho global de um conjunto de aplicações, devido ao tempo de espera que cada aplicação apresenta antes de iniciar a sua execução e também por se alocar recursos a aplicações que por vezes não os utilizam. Assim, a gestão global dos recursos do sistema de computação e a análise de todas as tarefas de

todas as aplicações, como considerado no método proposto, apresenta-se como a melhor solução para a minimização do *makespan* de um conjunto de aplicações. Estes resultados foram obtidos em *clusters* homogéneos e heterogéneos, com um grau alto e baixo de heterogeneidade. Em cada situação os *clusters* estavam equipados com 10 e 20 processadores.

A escolha do algoritmo de escalonamento a utilizar depende do sistema computacional que se considera, da estratégia que se utiliza, e do número de tarefas que esperam por ser executadas.

A análise dos resultados permite concluir que o algoritmo de escalonamento HPTS apresenta sempre um melhor desempenho, em relação ao HEFT, quando se considera uma política de reserva de processadores. Esta vantagem advém do melhor aproveitamento de recursos que é proporcionada pelo algoritmo de escalonamento HPTS, através da execução paralela de tarefas.

No método de escalonamento proposto, o algoritmo HEFT apresentou um melhor resultado, quando se considera um sistema computacional com 10 processadores. Estes resultados traduzem o facto, de que em situações de elevada carga e poucos recursos disponíveis, o algoritmo HPTS não pode tirar vantagem da execução paralela de tarefas. O melhor desempenho do HEFT explica-se também, porque o seu modelo computacional considera que comunicações e processamento podem ocorrer simultaneamente, o que não acontece com o HPTS. Esta característica não está presente em *commodity clusters*, sendo necessário efectuar mais testes, para avaliar o comportamento do algoritmo neste tipo de sistemas.

Quando se considerou um ambiente execução muito heterogéneo o HEFT, teve um melhor desempenho, quando implementado no método proposto. No entanto, ao analisar o desempenho global dos algoritmos, nas diferentes abordagens e ambientes de execução o HPTS é o que apresenta um desempenho mais regular.

A optimização do algoritmo de escalonamento designada por iHPTS, permite melhorar ligeiramente o desempenho do HPTS em *clusters* heterogéneos. Quando se considera um *cluster* homogéneo, não se verifica uma melhoria significativa no desempenho, sendo os resultados muito semelhantes.

Através da utilização da ferramenta *jScheduler* foi possível avaliar o comportamento do método de escalonamento dinâmico, que foi proposto neste trabalho, nas várias situações apresentadas, através da criação de um ambiente simulado. A sua utilização permitiu também demonstrar a aplicabilidade do método, num ambiente real, através do escalonamento e gestão da execução das tarefas, num *cluster* real.

A criação da ferramenta *jEditor* foi fundamental neste trabalho, pois facilitou a criação e análise dos DAGs que representaram cada uma das aplicações.

6.2 Trabalho Futuro

Numa perspectiva de investigação, nenhum trabalho deve ser dado como terminado. Desta forma, o trabalho apresentado deixa algumas portas abertas para possíveis estudos posteriores.

A análise dos resultados permite verificar de forma a superioridade do método de escalonamento dinâmico proposto em relação às outras duas abordagens. Pretende-se num trabalho futuro avaliar o desempenho dos dois algoritmos nas mais variadas condições, podendo o método escolher o algoritmo que garante melhor desempenho em determinado contexto. Como se está a lidar com heurísticas, têm de ser efectuados inúmeros testes, em diversas condições de execução, para minimizar as situações particulares que podem ocorrer, e desta forma generalizar os resultados.

Neste trabalho os sistemas de computação considerados foram *clusters*, homogéneos e heterogéneos, ligados através de uma rede dedicada. A aplicação do método proposto pode-se estender ao *grid computing* onde as características principais são semelhantes aos sistemas heterogéneos, excepto a taxa de transferência flutuante entre *clusters*. Assim, um ponto a desenvolver em trabalhos futuros será o estudo do método em sistemas com redes heterogéneas e com taxas de transferência flutuante.

As ferramentas criadas no âmbito deste trabalho, o *jScheduler* e o *jEditor* destinaram-se a testar os conceitos que foram aplicados, contudo, dada a sua aplicabilidade e potencial pretende-se melhorar as suas funcionalidades em projectos futuros. Uma característica importante a ser implementada é o suporte de tolerância a falhas, de forma que fosse realizado um novo reescalonamento, sempre que uma falha é detectada num nó do *cluster*. A adição desta característica também é fundamental para a aplicação do método de escalonamento num ambiente de *grid computing*.

Dado o novo paradigma da computação pessoal assentar em sistemas *multi-core*, pretende-se num trabalho futuro, a adaptação do método de escalonamento dinâmico, a estes sistemas de computação de memória partilhada.

Referências

[ASM+00] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Debra Hensgen e Sahra Ali. Representing task and machine heterogeneities for heterogeneous computing systems, tamkang. *Journal of Science and Engineering, Special 50 th Anniversary Issue*, 3:195–207, 2000.

[BM08] J. Barbosa e A.P. Monteiro. A list scheduling algorithm for scheduling multi-user jobs on clusters. In *VECPAR 2008*, volume LNCS 5336, pages 123–136. Springer-Verlag, 2008.

[BMNM05] J. Barbosa, C. Morais, R. Nobrega e A.P. Monteiro. Static scheduling of dependent parallel tasks on heterogeneous clusters. In *Heteropar'05*, pages 1–8. IEEE Computer Society, 2005.

[BMW+04] J. Blazewicz, M. Machowiak, J. Weglarz, M. Kovalyov e D. Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, (129):65–80, 2004.

[BTP00] J. Barbosa, J. Tavares e A.J. Padilha. Linear algebra algorithms in a heterogeneous cluster of personal computers. In *Proceedings of 9th Heterogeneous Computing Workshop*, pages 147–159. IEEE CS Press, May 2000.

[GW95] R. Geijn e J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report CS-95-286, University of Tennessee, Knoxville, 1995.

[Jan04] K. Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39:59–81, 2004.

[KA99] Y. Kwok e I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[KA05] Y. Kwok e I. Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *Journal of Parallel and Distributed Computing*, 65:1515–1532, 2005.

[KSS+07] Jong-Kook Kim, Sameer Shivle, Howard Jay Siegel, Anthony A. Maciejewski, Tracy D. Braun, Myron Schneider, Sonja Tideman, Ramakrishna Chitta, Raheleh B. Dilmaghani, Rohit Joshi, Aditya Kaul, Ashish Sharma, Siddhartha Sripada, Praveen Vangari e Siva Sankar Yellampalli. Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous

environment. *Journal of Parallel and Distributed Computing*, 67:154–169, 2007.

[LMT02] R. Lepère, G. Mounié e D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. *European journal of Operational Research*, (142):242–249, 2002.

[MAS+99] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen e R.F. Freund. Dynamically mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59:107–131, 1999.

[SSM+04] Sameer Shivle, H. J. Siegel, Anthony A. Maciejewski, Tarun Banka, Kiran Chindam, Steve Dussinger, Andrew Kutruff, Prashanth Penumarthy, Prakash Pichumani, Praveen Satyasekaran, David Sendek, J. Sousa, Jayashree Sridharan, Prasanna Sugavanam e Jose Velazco. Mapping of subtasks with multiple versions in a heterogeneous ad hoc grid environment. In *Heteropar'04*. IEEE Computer Society, 2004.

[SZI07] Wei SUN, Yuanyuan ZHANG e Yasushi INOGUCHI. Dynamic task flow scheduling for heterogeneous distributed computing: Algorithm and strategy. *IEICE Trans. INF. & SYST.*, E90-D(4):736–744, April 2007.

[THW02] H. Topcuoglu, S. Hariri e M.-Y Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[Try01] Denis Trystram. Scheduling parallel applications using malleable tasks on clusters. In *15th International Conference on Parallel and Distributed Processing Symposium*, 2001.

[WT98] Jerrell Watts e Stephen Taylor. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9:235–248, 1998.

Anexo A

Manual da Ferramenta jScheduler

Este pequeno manual pretende esclarecer todas as potenciais dúvidas que os utilizadores possam sentir durante a utilização do *jScheduler*.

Inicialmente é descrito o procedimento de instalação e os requisitos do programa. Seguidamente, é descrita forma de utilização das funcionalidades que o programa proporciona.

A.1 Modo de Instalação

O *jScheduler* não tem nenhum processador de instalação especifico, bastando copiar os ficheiros para a localização pretendida.

O programa necessita que o Java Runtime versão 1.4.2 ou posterior esteja instalado no sistema. Para ser possível a execução de tarefas é necessário que o MPICHV2 esteja instalado e a executar no sistema.

Antes da execução da ferramenta é necessário editar o ficheiro de configuração *processors.conf*, localizado na pasta da ferramenta. O ficheiro deve ser preenchido com as características e localização de cada processador do cluster. Cada linha representa um processador e deve ter a seguinte configuração:

```
<nº da linha> <localhost> <ip> <capacidade do processador em Mflops>
```

Para iniciar o programa basta utilizar o seguinte comando:

```
java jScheduler
```

A.2 Modo de Utilização

O *jScheduler* é uma ferramenta em que a interacção com o utilizar se faz em modo texto. Antes

A.2.1 Submeter uma aplicação

O utilizador, depois de iniciar o *jScheduler* pode em qualquer momento inserir uma aplicação para ser escalonada e executada pelo *cluster*. Para isso, deve inserir o seguinte comando:

```
addjob <id> <file.xml>
```

A informação relativa ao escalonamento do *job* é exibida no ecrã, bem como do estado de execução de cada uma das tarefas. Esta informação é armazenada nos ficheiros de *log* para posterior análise do utilizador.

Anexo B

Manual da Ferramenta jEditor

Este pequeno manual pretende esclarecer todas as potenciais dúvidas que os utilizadores possam sentir durante a utilização do *jEditor*.

Inicialmente é descrito o procedimento de instalação e os requisitos do programa. Seguidamente, é descrita forma de utilização das funcionalidades que o programa proporciona.

B.1 Modo de Instalação

O *jEditor* não tem nenhum processador de instalação especifico, bastando copiar os ficheiros para a localização pretendida.

O programa necessita que o Java Runtime versão 1.4.2 ou posterior esteja instalado no sistema.

Para iniciar o programa basta utilizar o seguinte comando:

```
javaw jEditor
```

B.2 Modo de Utilização

Ao iniciar o *jEditor* é aberta a janela principal do programa B.1. Nela podemos distinguir 3 áreas diferentes:

- Barra de ferramentas;
- Área de desenho;
- Propriedades dos objectos seleccionados;

Na área superior está localizada a barra de ferramentas, onde estão disponíveis as opções para descrever uma aplicação através de um DAG, nomeadamente, a criação e remoção de tarefas, a criação e remoção de arestas, mover uma tarefa e selecção de um objecto. A área de desenho está situada à direita e é o espaço onde pode ser construído o DAG. A área que está situada à esquerda é reservada para editar as propriedades dos objectos que estão seleccionados.

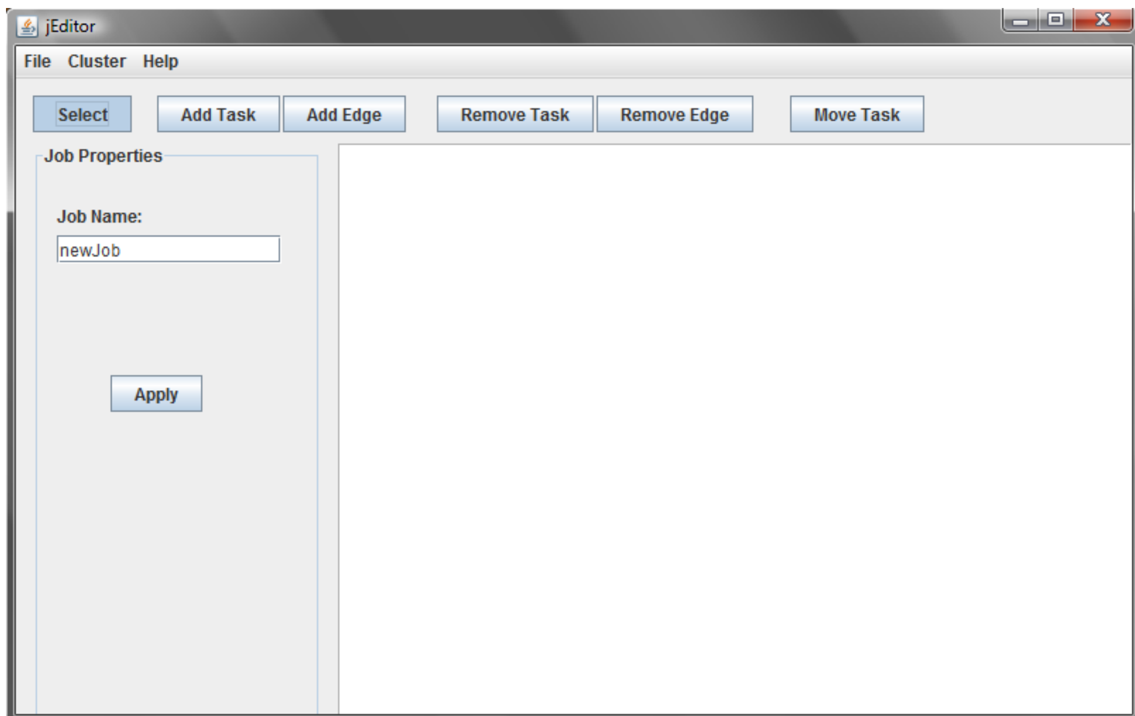

Figura B.1: Janela principal do jEditor

B.2.1 Adicionar tarefas

Para adicionar uma tarefa ao DAG, selecciona-se a opção *Add Task* na barra de ferramentas e efectua-se um *clique* na zona da área de desenho onde se pretende adicionar a tarefa, ver Figura B.2. Por fim, deve-se editar as propriedades da tarefa criada, nomeadamente:

- *Task Name*: o nome da tarefa;
- *Operation*: indica qual é a operação que será realizada pela tarefa;
- *Parameters*: para a adição de parâmetros adicionais a ter com conta no momento de execução;
- *Matrix Dimension*: indica a dimensão da matriz de dados;

Para as alterações sortirem efeito é necessário efectuar *clique* no botão *Apply*.

B.2.2 Adicionar aresta

A adição de uma aresta no DAG, permite criar uma relação de precedência entre duas tarefas. Para criar uma aresta é necessário seleccionar a opção *Add Edge* na barra de ferramentas. Seguidamente, deve-se desenhar a aresta, sempre com o botão direito do rato pressionado, desde a tarefa de origem até à de destino, Figura B.3.

Quando se define uma aresta entre duas tarefas é automaticamente verificado se esta origina um ciclo no DAG. Se isto acontecer é apresentada uma mensagem de erro e a aresta não é considerada.

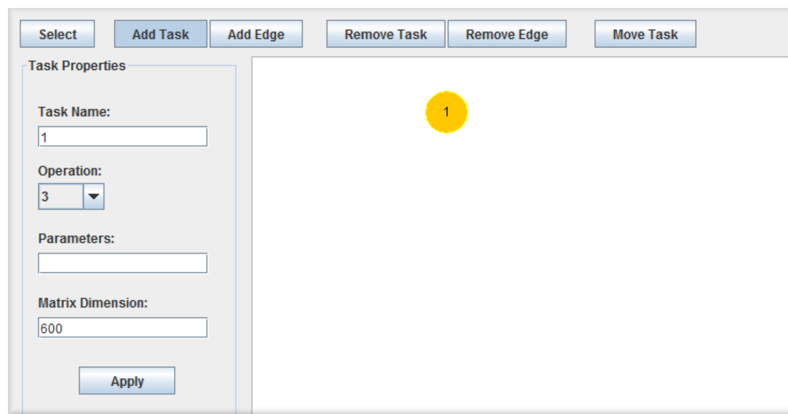

Figura B.2: Definição dos atributos de uma tarefa

Apenas pode existir uma aresta entre duas tarefas.

B.2.3 Remover tarefa

A remoção de uma tarefa pode ser efectuada através da selecção da opção *Remove Task* na barra de ferramentas. Depois, basta efectuar um *clique* na respectiva tarefa. Uma tarefa depois de removida, não pode ser efectuada a sua recuperação.

Todas as aresta que partiam e chegavam à tarefa eliminada, são também removidas.

B.2.4 Remover aresta

Para remover uma aresta deve-se seleccionar a opção *Remove Edge* disponível na barra de ferramentas. O processo para eliminar uma aresta assemelha-se à sua criação. Assim, sempre com o botão direito do rato pressionado, selecciona-se a aresta, desde a tarefa de origem até à de destino. A aresta que liga as duas tarefas será automaticamente removida.

Quando se tenta eliminar uma aresta que não existe, entre duas tarefas, é exibida uma mensagem de erro.

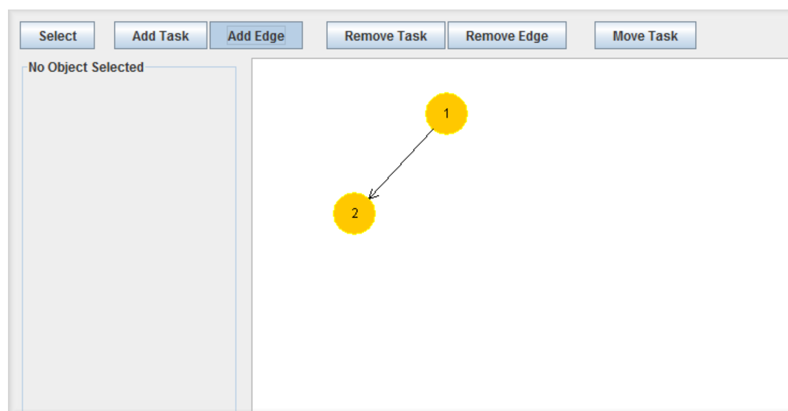

Figura B.3: Exibição de uma aresta entre duas tarefas

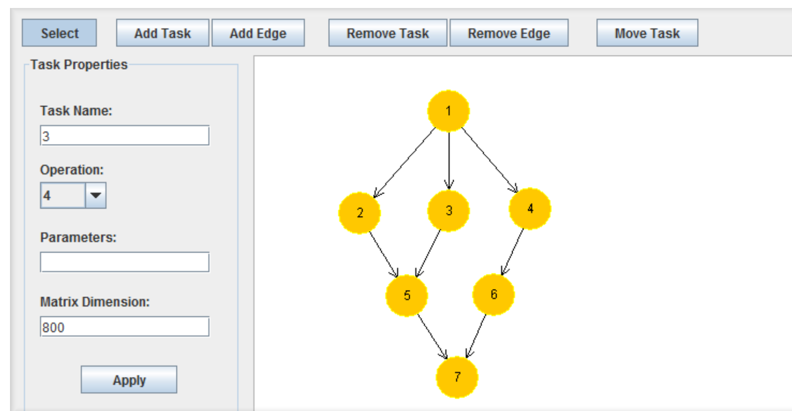

Figura B.4: Propriedades de uma tarefa do DAG

B.2.5 Mover tarefa

O utilizador pode em qualquer momento alterar a disposição das tarefas no DAG, movendo-as para qualquer localização da área de desenho. Para mover uma tarefa deverá seleccionar a opção *Move Task* na barra de ferramentas. Com o botão direito do rato pressionado sobra a tarefa que se deseja mover, deve-se arrastar para a nova localização pretendida.

Quando se move uma tarefa, as arestas que estão a ela relacionadas são automaticamente ajustadas para a nova localização da tarefa.

B.2.6 Seleccionar objectos

As propriedades de um objecto podem ser editadas em qualquer momento. Os objectos que se consideram no *jEditor* são as tarefas e o DAG.

Para seleccionar uma tarefa o utilizador deve seleccionar a opção *Select* na barra de ferramentas e efectuar um *clique* sobre a tarefa respectiva. A área relativa às propriedades dos objectos, passa a exibir as propriedades da tarefa, nomeadamente: nome da tarefa; tipo de operação; parâmetros adicionais; tamanho da matriz de dados. Estes dados podem ser editados tendo-se no final de confirmar as alterações com um *clique* no botão *Apply*, ver Figura B.4.

O nome que caracteriza o DAG também pode ser editado, para isso, com a opção *Select* da barra de ferramentas activa, deve-se executar um clique numa zona sem tarefa da área de desenho. Na área de propriedades pode-se alterar o nome de DAG, devendo-se confirmar a alteração através do botão *Apply*.

B.2.7 Submeter uma aplicação

O *jEditor* permite submeter *jobs* para execução. Para utilizar esta funcionalidade o utilizador deverá seleccionar a opção *Submit Job* que está no menu *Cluster*. Seguidamente, é aberta uma janela com algumas caixas de texto que são necessárias preencher, nomeadamente:

- *username*: indica o utilizador responsável pelo *job*;

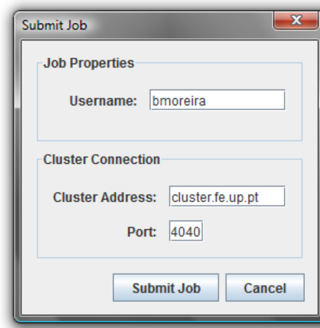

Figura B.5: Janela de diálogo que permite o envio de aplicações

- *Cluster address*: indica qual é o endereço do cluster;
- *port*: indica a porta que está a ser ouvida pelo jScheduler.

Depois da introdução dos dados, deverá ser efectuado um clique no botão OK, ver Figura B.5.

O *job* que será enviado para o cluster é o que está aberto na área de desenho.

A utilização desta funcionalidade implica que o jScheduler esteja a executar no cluster.

B.2.8 Outras funcionalidades

O *jEditor* permite que os DAG sejam guardados e abertos novamente, para continuar a sua edição.

Cada DAG é guardado num ficheiro em formato XML. Estes ficheiros são interpretados pelo *jScheduler*.

Anexo C

Descrição dos métodos das classes das Ferramentas

Neste anexo são apresentados, através de uma breve descrição, os métodos implementados em cada classe do *jScheduler* e do *jEditor*. As classes são as representadas nos diagramas de classes apresentados no Capítulo 4.

Ferramenta *jScheduler*

Classe *Task*

Cada instância da classe *Task* representa uma tarefa de uma aplicação, ou seja, um nó de um DAG. Os atributos de cada instância caracterizam a tarefa, sendo usados para efectuar o seu escalonamento. Cada tarefa está associada a um *job* e pode ter relações de precedência com outras tarefas. As relações de precedência são definidas através de instâncias da classe *Edge*.

Métodos Implementados

void	**addProcessor**(Processor p) Adiciona um processador à tarefa.
void	**completedTask**() Põe a tarefa como executada no job.
boolean	**containsNextTask**(Task task) Verifica se a tarefa está nas nextTasks
double	**getBLevel**() Retorna o bLevel da tarefa
double	**getCommTimeToNextTask**(Task b) Retorna o tempo de comunicação da tarefa actual à tarefa b
double	**getEndTime**() Retorna o tempo de fim de execução da tarefa
String	**getId**() Retorna a identificação da tarefa
Job	**getJob**() Retorna o job ao qual pertence
int	**getMatrixDim**() Retorna a dimensão da matriz de dados
Processor	**getMaxCapacityProcessor**() Retorna o processador com maior capacidade que está atribuído à tarefa
double	**getMaxPrevEndTime**() Calcula o tempo máximo (EndTime) de todas as tarefas que a antecedem (prevTasks).
java.util.LinkedList	**getNextEdges**() Retorna a lista das tarefas que
java.util.LinkedList	**getNextTasks**() Retorna todas as tarefas que a precedem
int	**getNProcessors**() Retorna o número de processadores atribuídos à tarefa.
java.util.LinkedList	**getPrevEdges**() Retorna as arestas que
java.util.LinkedList	**getPrevTasks**() Retorna todas as tarefas que a antecedem
boolean	**getProcessBLevel**() Verifica se o blevel já foi processado
java.util.LinkedList	**getProcessors**() Retorna todos os processadores reservados para a tarefa.
double	**getRealCapacity**() Calcula a capacidade total de processamento reservada à tarefa.
double	**getSCapacity**() Retorna a capacidade óptima da
double	**getStartTime**() Retorna o tempo de início da tarefa.

double	**getSumPrevCommTime**() Soma o custo de comunicação de todas as arestas que se ligam à tarefa (prevEdges)
int	**getTaskOperation**() Retorna a operação matemática associada à tarefa
boolean	**isCompleted**() Verifica se a tarefa já está foi executada
boolean	**isScheduled**() Verifica se a tarefa já está escalonada
void	**removeAllProcessors**() Remove todos os processadores mapeados na tarefa
void	**removeProcessor**(Processor p) Remove um processador que está reservado para a tarefa
void	**setBLevel**(double b) Actualiza o valor do bLevel da tarefa
void	**setComunicationTime**(double time) Atribui o tempo de comunicação a todas as arestas que saem da tarefa
void	**setEndTime**(double e) Actualiza o valor do fim de execução da tarefa
void	**setJob**(Job job) Inicia a tarefa com o job a que pertence
void	**setNextEdge**(Edge edge) Adiciona uma nova aresta (sucessora) à tarefa
void	**setPrevEdge**(Edge edge) Adiciona uma nova aresta (antecedente) à tarefa
void	**setProcessBLevel**(boolean b) Actualiza o valor para true quando processa o blevel
void	**setSCapacity**(double sc) Actualiza o valor da capacidade
void	**setScheduledTask**() Põe a tarefa como escalonada no job
void	**setStartTime**(double s) Actualiza o valor de inicio de execução da tarefa

Classe *Edge*

Uma instância da classe *Edge* representa uma aresta entre dois nós de um DAG, ou seja, representa uma relação de precedência entre duas tarefas. Uma aresta tem uma tarefa inicial e outra final, definindo desta forma o seu sentido. Cada instância desta classe tem também associado o custo de comunicação entre as duas tarefas. Entre um par de tarefas apenas pode existir uma aresta.

Métodos Implementados

Task	**getATask**() Retorna a tarefa de partida.
Task	**getBTask**() Retorna a tarefa de chegada.
double	**getCommTime**() Retorna o tempo de comunicação entre a tarefa de partida e chegada.
void	**setCommTime**(double c) Atribui o tempo de comunicação entre a tarefa de partida e chegada.

Classe *Job*

Uma instância da classe *Job* representa uma aplicação que foi submetida por um utilizador, para ser escalonada e executada. Cada objecto desta classe contem como atributos as tarefas que constituem uma aplicação e as outras características relevantes para a caracterizar, como por exemplo, o utilizador, data de inserção, etc.

Métodos Implementados

java.util.LinkedList	**availableRunTasks**() Retorna todas as tarefas que estão disponíveis para executarem.
java.util.LinkedList	**availableSchedulerTasks**() Retorna todas as tarefas que estão disponíveis para escalonamento.
private java.util.LinkedList	**availableTasks**(java.util.HashSet verifiedTasks) Retorna todas as tarefas que estão disponíveis para escalonamento.
boolean	**completedJob**() Verifica se todas as tarefas do job já foram executadas.
void	**completedTask**(Task task) Coloca uma tarefa no conjunto de tarefas terminadas.
void	**computingBLevel**(java.util.LinkedList topList) Calcula e actualiza o valor do BLevel de cada uma das tarefas do job.
java.util.LinkedList	**computingTopologicalOrder**() Ordena topologicamente as tarefas do job topologicamente numa lista.
java.util.LinkedList	**getAllExecutedTasks**() Retorna todas as tarefas do job que já foram executadas.
java.util.LinkedList	**getAllNonExecutedTasks**() Retorna todas as tarefas do job que ainda não foram executadas.
java.util.LinkedList	**getAllTasks**() Retorna todas as tarefas do job.
java.lang.String	**getId**() Retorna a identificação da job.
java.util.Date	**getInsertionTime**() Retorna a data em que a job foi inserido para escalonamento.
double	**getReservation**()
java.util.Date	**getStartExecutionTime**() Retorna a data em que o job iniciou a execução.
java.lang.String	**getUser**() Retorna o username do dono do job.
boolean	**isCompletedTask**(Task task) Verifica se a tarefa está concluída.
boolean	**isScheduledTask**(Task task) Verifica se a tarefa está escalonada.
void	**scheduledTask**(Task task) Coloca uma tarefa no conjunto de tarefas escalonadas.
private void	**setAllTasksJob**() Associa todas as tarefas ao job.
void	**setInsertionTime**(java.util.Date date) Inicia a data em que o job foi inserido para escalonamento.
void	**setScheduledAllCompletedTasks**() Remove todas as tarefas que já foram executadas.
void	**setStartExecutionTime**(java.util.Date date) Inicia a data em que o job iniciou a execução.

Classe *Processor*

Cada objecto da *Processor* representa um processador do sistema de computação. Cada processador é caracterizado pela sua capacidade de processamento e pelo seu endereço de localização.

Métodos Implementados

void	**addTask**(Task task) Adiciona uma tarefa ao processador para este a executar.
double	**getCapacity**() Obtém a capacidade do processador (em MFLOPS).
double	**getEndTime**() Calcula quando termina a última tarefa processada pelo processador.
java.lang.String	**gethost**() Obtém o host onde se encontra o processador
int	**getId**() Obtém a identificação do processador.
java.lang.String	**getIp**() Obtém o endereço ip da máquina onde está o processador
java.util.LinkedList	**getTasks**() Retorna todas as tarefas que estão associadas ao processador.
void	**removeTask**(Task t) Remove uma tarefa.
void	**removeTasks**() Remove todas as tarefas associadas ao processador.
void	**removeTasksExcept**(java.util.Collection exceptionTa sks) Remove todas as tarefas excepto as indicadas.
void	**setStartMappingTime**(double time) Actualiza a data de referência do processador.

Classe *Cluster*

Um objecto da classe *Cluster* caracteriza-se por representar o sistema de computação considerado. Uma instância desta classe é caracterizada por um conjunto de processadores e pelas características da rede de comunicações que os ligam.

Métodos Implementados

private void	**addProcessor**(int id, java.lang.String host, java.lang.String ip, double capacity) Adiciona um processador à lista de processadores.
void	**cleanProcessors**() Remove todas as tarefas associdas aos processdores
void	**computingComunications**(Job job, double ccr) Calcula e inicia o tempo de comunicação entre tarefas.
void	**computingSOptimal**(Job job) Calcula o SOptimo de cada tarefa (tempo de execução e capacidade de processamento óptimos).
double	**executionTime**(Task task, int nProcessors, double meanProcessorsCapacity) Calcula o tempo de execução de uma tarefa tendo em conta a

	capacidade de processamento envolvida e o número de processadores Considera as operações (TRD, Q, QR e Correlação)
Processor	**getProcessor**(int index) Retorna o processador que está na posição index
java.util.LinkedList	**getProcessors**() Retorna a lista de todos os processadores do cluster.
double	**getProcessorsCapacity**() Calcula a capatidade total dos processadores.
double	**getReferenceTime**() Retorna o tempo referência do cluster.
private void	**readProcessors**() Lê o ficheiro de configuração que contém os processadores do cluster
int	**totalProcessors**() Retorna o número de processadores do cluster

Classe *XmlReader*

A classe *XmlReader* é responsável por interpretar os ficheiros XML que contêm a descrição de uma aplicação, nomeadamente, as tarefas e suas características. Os ficheiros XML podem ser facilmente criados com a utilização da ferramenta *jEditor*.

Métodos Implementados

static boolean	**addEdge**(java.util.LinkedList allTasks, java.lang.String a, java.lang.String b) Adiciona uma edge que liga duas tarefas do job.
static void	**addTask**(java.util.LinkedList allTasks, java.lang.String id, int taskOperation, int matrixDim) Adiciona uma tarefa ao job.
private static boolean	**createEdge**(java.lang.String document, java.util.LinkedList allTasks) Identifica os atributos de uma aresta.
private static boolean	**createTask**(java.lang.String document, java.util.LinkedList allTasks) Identifica os atributos de uma tarefa.
private static void	**findEdge**(java.lang.String document, java.util.LinkedList allTasks) Identifca uma aresta no documento xml.
private static void	**findEdges**(java.lang.String document, java.util.LinkedList allTasks) Identifica o bloco com as arestas.
private static void	**findTask**(java.lang.String document, java.util.LinkedList allTasks) Identifca uma tarefa no documento xml.
private static void	**findTasks**(java.lang.String document, java.util.LinkedList allTasks) Identifica o bloco com as tarefas.
static java.util.LinkedList	**readJobFile**(java.lang.String fileName)
private static void	**readXmlFile**(java.util.LinkedList allTasks, java.lang.String fileName) Lê o ficheiro xml e verifica a sua validade.

Classe *RunTask*

Cada instância da classe *RunTask* representa o *thread* responsável por controlar a execução de uma tarefa. Cada objecto desta classe, é responsável por iniciar a execução de uma tarefa, no(s) processador(es) que lhe estão alocados. É também responsável por detectar o fim de execução de uma tarefa.

Métodos Implementados

void	**run**() Inicia a execução do thread

Classe *JobManager*

A classe *JobManager* é responsável pela gestão do escalonamento das tarefas de todas as aplicações. A estratégia de escalonamento, que define o momento em que o algoritmo de escalonamento é invocado, está implementada nesta classe. Além disso, esta classe é também responsável por conservar o estado de execução de cada uma das aplicações.

Métodos Implementados

void	**addJob**(Job job) Adiciona um job para escalonamento.
private java.util.LinkedList	**availableTasks**() Verifica quais as tarefas que estão disponíveis para execução e actualiza a lista "availableTasks".
void	**completedTask**(Task task) Uma tarefa terminou.
private java.util.LinkedList	**getFreeProcessors**() Retorna os processadores livres.
void	**manageExecutingJobs**() Gere a execução dos jobs.
private boolean	**processorMatch**(Task task) Verifica o "match" entre tarefas e processadores.
private void	**removeAllProcessors**(java.util.LinkedList jobs) Remove os processadores das tarefas que ainda não executaram.
void	**runAvailableTasks**() Executa as tarefas disponíveis.
private void	**runConsole**() Executa o sistema de inserção de jobs (consola, sockets, etc.)
private void	**runReadyTasks**(java.util.LinkedList sortedAvailableTasks) Executa as tarefas prontas a executar.
private void	**setScheduledAllCompletedTasks**(java.util.LinkedList jobs) Todas as tarefas que já foram executadas, são colocadas como escalonadas.
private void	**setScheduledAllRunningTasks**() Todas as tarefas que estão em execução são colocadas como escalonadas para não serem consideradas.

private java.util.LinkedList	**sortAvailableTasks**(java.util.LinkedList availableTasks) Ordena as tarefas disponíveis pelo seu startTime.
private void	**suspendExecution**(Job job) Suspende o modelo de execução actual para gerar um novo devido à adição de um novo job.
private void	**writeGraph**() Cria ficheiros com o mapa da execução.

Interface *Scheduler*

A interface define a estrutura das classes que implementam cada um dos algoritmos de escalonamento.

Métodos Definidos

void	**schedulingJobs**(java.util.LinkedList selectedJobs) Deve implementar o escalonamento dos diferentes jobs que recebe.

Interface *Console*

A interface define a estrutura que terá ser implementada pelas classes que sejam responsáveis pela interacção com os utilizadores. As classes que implementam esta interface devem permitir a introdução de aplicações no, para escalonamento e execução. Foi tomada a opção de utilizar de uma interface, de forma a possibilitar a fácil criação de novas classes que suportem a interacção com o utilizador.

Métodos Definidos

private void	**addCommand**(java.lang.String command) Deve implementar a adição de um job
private void	**readConsole**() Deve implementar a inserção de comandos.
void	**run**() Executa o thread Consola.

Ferramenta *jEditor*

Classe *JobDag*

Uma instância da classe *JobDag* representa uma aplicação, ou seja um DAG. Este é caracterizado através de um conjunto de tarefas.

Métodos Implementados

static void	**addEdge**(java.lang.String startTask, java.lang.String endTask) Adiciona uma aresta
static void	**addEdge**(Task startTask, Task endTask) Adiciona uma aresta
static void	**addTask**(Task task) Adiciona uma tarefa
static void	**drawDag**(java.awt.Graphics g) Desenha todas as tarefas do dag no ambiente de desenho
static void	**drawDag**(java.awt.Graphics g, Task task) Desenha todas as tarefas de um dag quando uma tarefa está a ser movida
static void	**drawEdge**(Task startTask, Task endTask, java.awt.Graphics g) Desenha uma aresta
static void	**drawMoveTask**(int x, int y, java.awt.Graphics g) Desenha a tarefa que está a ser movida
static void	**drawTask**(Task task, java.awt.Graphics g) Desenha uma tarefa
static boolean	**existEdge**(Task task1, Task task2) Verifica se existe uma aresta com a origem e destino dados nos parâmetros
static boolean	**foundCycle**() Verifica se existe um ciclo no dag
static java.util.LinkedList	**getDag**() Retorna a LinkedList com todas as tarefas do dag
static java.lang.String	**getJobId**() Retorna o nome do Job
static void	**newJob**() Elimina a configruração anterior e cria um novo Dag
static boolean	**removeEdge**(Task task1, Task task2) Remove uma tarefa
static void	**removeTask**(Task task) Remove uma tarefa
static Task	**selectedTask**(int x, int y) Selecciona uma tarefa no abiente de desenho
static void	**setJobId**(java.lang.String s) Actualiza o nomo do job

Classe *JobDatagram*

A classe *JobDatagram* representa um *job* submetido remotamente. Cada instância desta classe contém a informação necessária a ser enviada remotamente para o *JScheduler* para submeter uma aplicação.

Métodos Implementados

java.lang.StringBuffer	**getDagFile**() Retorna o ficheiro que xml que representa um dag
java.lang.String	**getUser**() Retorna o nome do utilizador

Classe *Task*

A classe *Task* representa as tarefas de uma aplicação. Cada desta classe *Task* representa uma tarefa de uma aplicação, ou seja, um nó de um DAG. Os atributos de cada instância caracterizam a tarefa, sendo inseridos pelo utilizador. Cada tarefa está associada a um *JobDag* e pode ter relações de precedência com outras tarefas.

Métodos Implementados

void	**addNextTask**(Task task) Adiciona uma nova tarefa precedente
java.lang.String	**getId**() Retorna a identificação da tarefa
int	**getMatrixDim**() Retorna a dimensão dos dados de entrada de uma tarefa
java.util.LinkedList	**getNextTasks**() Retorna todas as tarefas precedentes
java.lang.String	**getParameters**() Retorna os parâmetros de execução de uma tarefa
int	**getTaskOperation**() Retorna a operação matemática atribuída à tarefa
int	**getX**() Retorna a coordenada X, onde a tarefa está representada
int	**getY**() Retorna a coordenada Y, onde a tarefa está representada
void	**removeNextTask**(Task task) Remove uma tarefa precedente
void	**setId**(java.lang.String id) Actualiza o valor de identificação de uma tarefa
void	**setMatrixDim**(int matrixDim) Actualiza a dimensão dos dados de entrada de uma tarefa
void	**setParameters**(java.lang.String parameters) Actualiza os parâmetros de execução de uma tarefa
void	**setTaskOperation**(int taskOperation) Actualiza a operação matemática da tarefa
void	**setXY**(int x, int y) Actualiza o valor das coordenadas

Classe *XmlFilter*

A classe *XMLFilter* é responsável por verificar e filtrar os ficheiros que serão exibidos nas caixas de diálogo *Open* e *Save* do *jEditor*. São apenas exibidos os ficheiros com extensão *xml.*

Métodos Implementados

java.lang.String	**getDescription**() Retorna a descrição de um ficheiro

Classe *XmlReader*

Esta classe é responsável por interpretar os ficheiros XML que contêm a descrição de uma aplicação. Isto permite que o *jEditor* abra os ficheiros para posterior edição.

Métodos Implementados

static void	**readDag**(java.io.BufferedReader fin) Faz a leitura e interpreta um dag descrito num ficheiro XML

Classe *XmlWriter*

Esta classe é responsável por escrever os ficheiros XML que contêm a descrição de uma aplicação. Isto permite que o *jEditor* guarde os ficheiros para posterior edição.

Métodos Implementados

static void	**writeBufferDag**(java.lang.StringBuffer fout) Escreve o ficheiro para XML que descreve o dag para ser enviado por sockets
static void	**writeFileDag**(java.io.PrintWriter fout) Escreve o ficheiro para XML que descreve o dag

Classes *GUI, GUIAbout e GUISubmitJob*

Nestas classes é implementada a interface gráfica. Devido aos inúmeros métodos que correspondem aos vários eventos dos dispositivos de entrada, estes não são aqui enumerados.