Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Uma Heurística de Escalonamento Adaptativa à Disponibilidade da Informação para Aplicações Bag-of-Tasks Data-Intensive em Grids Computacionais

Leonardo de Assis

Campina Grande, Paraíba, Brasil
Setembro - 2009

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Uma Heurística de Escalonamento Adaptativa à Disponibilidade da Informação para Aplicações Bag-of-Tasks Data-Intensive em Grids Computacionais

Leonardo de Assis

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação (M.Sc).

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Francisco Brasileiro

(Orientador)

Campina Grande, Paraíba, Brasil

©Leonardo de Assis, Setembro - 2009

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

A848h
2009 Assis, Leonardo de.
Uma heurística de escalonamento adaptativa à disponibilidade da informação para aplicações Bag-of-Tasks Data-Intensive em Grids computacionais / Leonardo de Assis. — Campina Grande, 2009.
63 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Prof. Dr. Francisco Brasileiro.

1. Redes de Computadores. 2. Escalonamento de Aplicações em Grids. I. Título.

CDU – 004.75(043)

"UMA HEURÍSTICA DE ESCALONAMENTO ADAPTATIVA À DISPONIBILIDADE DA INFORMAÇÃO PARA APLICAÇÕES BAG-OF-TASKS DATA-INTENSIVE EM GRIDS COMPUTACIONAIS"

LEONARDO DE ASSIS

DISSERTAÇÃO APROVADA EM 02.09.2009

FRANCISCO VILAR BRASILEIRO, PH.D
Orientador(a)

ANTONIO MARINHO PILLA BARCELLOS, PH.D
Examinador(a)

LUIZ EDUARDO BUZATO, PH.D
Examinador(a)

CAMPINA GRANDE - PB

Resumo

A tecnologia de *grids* foi criada com o objetivo de facilitar o compartilhamento de recursos entre indivíduos que estão vinculados a diferentes domínios administrativos. Nos últimos anos, o uso de *grids* computacionais está cada vez mais comum devido ao grande poder computacional que esta tecnologia pode prover a um baixo custo. Devido a isso, a execução de aplicações paralelas que processam uma grande quantidade de dados (*data-intensive*) está cada vez mais comum neste tipo de plataforma. Uma aplicação paralela pode ser vista como uma coleção de tarefas que podem ser executadas em paralelo. Para algumas destas aplicações, essas tarefas são independentes e podem ser escalonadas para execução paralela em qualquer ordem. Este tipo de aplicação paralela é referenciada na literatura como aplicações *Bag-of-Tasks* (BoT). Com o intuito de escalonar tarefas em recursos de uma maneira eficiente, escalonadores de aplicações em *grid* utilizam heurísticas de escalonamento. As heurísticas de escalonamento existentes podem ser classificadas em duas abordagens: i) heurísticas *bin-packing*, e ii) heurísticas baseadas em replicação. A primeira abordagem requer informação completa e precisa sobre o ambiente de execução e a aplicação. A segunda abordagem não utiliza informação alguma, mas, ao invés disso, ela aplica o princípio da replicação de tarefas para atingir um bom desempenho. Porém, ambas abordagens têm desvantagens; obter informação completa e precisa sobre o ambiente de execução e a aplicação não é sempre possível em um ambiente de *grid* computacional, enquanto que a redundância das heurísticas baseadas em replicação ocasiona no desperdício de recursos. Em um trabalho recente, foi investigado que apesar de que em um ambiente de *grid* a informação precisa é difícil de se obter, o acesso a ela não é impossível. Na prática, parte da informação pode ser obtida usando serviços que coletam informação sobre o ambiente de execução e publicada em serviços de informação do *grid*. Aquele mesmo trabalho mostrou que é possível reduzir o custo de execução de aplicações *CPU-intensive*, mantendo a mesma eficiência, usando qualquer informação que esteja disponível. Com base no pressuposto daquele trabalho, este trabalho apresenta uma heurística de escalonamento para aplicações BoT *data-intensive*, que é adaptativa à disponibilidade da informação, chamada de Adaptive Data-Intensive. Os resultados obtidos pela heurística Adaptive Data-Intensive mostraram

que o uso racional da informação que estiver disponível leva a uma redução no tempo de execução da aplicação e no desperdício dos recursos.

Abstract

The technology of grid was created to facilitate the resource sharing among individuals belonging to different administrative domains. In recent years, the use of grid computing is increasingly common due to the large computational power that this technology can provide at a low cost. Because of this, the execution of parallel applications that process a large amount of data (data-intensive) is increasingly common in this type of platform. A parallel application can be viewed as a collection of tasks that can be executed in parallel. A parallel application can be viewed as a collection of tasks that can be executed in parallel. For some of these applications, these tasks are independent and can be scheduled to run parallel in any order. This type of parallel application is referenced in literature as Bag-of-Tasks (BoT) applications. In order to schedule tasks onto resources in an efficient manner, grid applications schedulers use scheduling heuristics. The scheduling heuristics can be classified into two approaches: i) bin-packing heuristics, and ii) heuristics based on replication. The first approach requires complete and accurate information about the execution environment and the application. The second approach does not use any information, but, instead, it applies the principle of tasks replication to achieve good performance. But both approaches have disadvantages, complete and accurate information about the execution environment and the application is not always possible in a grid computing environment, while the redundancy of replication heuristics causes resource waste. In a recent work, it was investigated despite the fact that in a grid environment, the accurate information is difficult to get, it is not impossible to have it. In practice, the information can be obtained by using services that collect information about the environment and the application and publish it on grid information services. That same study showed that it is possible to reduce the execution cost of CPU-intensive applications, while maintaining the same efficiency, using any information that is available. Based on the assumption of that work, this dissertation presents a scheduling heuristic for BoT data-intensive applications that is adaptive to the information availability, called Adaptive Data-Intensive. The results obtained by heuristic Adaptive Data-Intensive indicated that the rational use of available information leads to a reduction of application execution time and resource waste.

Agradecimentos

Agradeço à minha família pelo apoio imenso durante toda minha carreira acadêmica, especialmente Emmanuelle que em todo esse período ela foi minha namorada, noiva e esposa.

Ao meu orientador Fubica que me mostrou o caminho da pesquisa e está me orientando desde 2005. Ao professor Walfredo por ter me dado a oportunidade de trabalhar no LSD e pelo apoio dado quando eu estava nos Estados Unidos.

Aos amigos de trabalho que sempre estavam de plantão para tirar dúvidas, especialmente Joãozinho e Raquel que leram boa parte da minha dissertação e me deram ótimos *feedbacks*, Manel com seus comentários com duplo sentido, Toquinho, Giovanni, Jonhnny, Paulinho e Marquinhos.

À equipe do OurGrid por ter fechado uma versão estável do *middleware* reduzindo drasticamente o tempo de execução das simulações dessa dissertação.

Ao pessoal do LSD por fazer desse laboratório um excelente local de trabalho.

Conteúdo

1 Introdução **1**

2 Heurísticas de escalonamento em *grids* computacionais **4**

2.1 Heurísticas *bin-packing* 5

2.1.1 *XSufferage* 6

2.1.2 *RUMR*, *UMR* e *Weighted Factory* 8

2.1.3 Heurística de escalonamendo do *middleware GridBus* 10

2.1.4 Computação Voluntária 13

2.2 Heurísticas baseadas em replicação 14

2.2.1 *Workqueue with Replication* 15

2.2.2 Heurística de escalonamento para aplicações que possuem a carga divisível 16

2.2.3 *Storage Affinity* 19

2.3 *Adaptive WQR* 22

3 Modelo para escalonamento de aplicações *BoT* em *grids* computacionais **27**

3.1 Modelo de *grid* 27

3.2 Modelo da aplicação 30

3.3 Modelo da informação 31

3.4 Escalonamento 33

3.5 Métricas de desempenho 34

4 Heurística Adaptive Data-Intensive **36**

5 Avaliação de desempenho **41**

5.1 Descrição dos cenários 42

5.1.1 Heterogeneidade do *grid* 42

5.1.2 Granularidade e heterogeneidade das tarefas da aplicação 43

5.1.3 Disponibilidade da informação 43

5.1.4 Grau de replicação máximo 44

5.1.5 Tipos de aplicações *data-intensive* 44

5.1.6 Repetição da execução 45

5.2 Validade dos resultados 46

5.3 Resultados 46

5.3.1 Impacto da disponibilidade da informação no *makespan* 47

5.3.2 Impacto da disponibilidade da informação no desperdício de recursos 49

5.3.3 Impacto do grau de replicação máximo no desperdício de recursos . 52

5.3.4 Aplicações do tipo *não-proporcional* 52

6 Conclusão **57**

Lista de Símbolos

BoT - *Aplicação do tipo Bag-of-Tasks*

GIS - *Serviço de Informação do Grid (Grid Information Service)*

CPU - *Unidade Central de Processamento (Central Processing Unit)*

MCT - *Tempo Mínimo de Conclusão (Minimun Completion Time)*

WQ - *Heurística de Escalonamento Workqueue*

RUMR - *Heurística de Escalonamento Robust Uniform Multi-Round*

UMR - *Heurística de Escalonamento Uniform Multi-Round*

BOINC - *Berkeley Open Infrastucture for Network Computing*

SETI - *Search for Extra-Terrestrial Intelligence*

WQR - *Heurística de Escalonamento Workqueue with Replication*

FPLTF - *Heurística de Escalonamento Fastest Processor to Large Task First*

DFPLTF - *Heurística de Escalonamento Dynamic Fastest Processor to Large Task First*

P2P - *Entre-Pares (Peer-to-Peer)*

LAN - *Local Area Network*

WAN - *Wide Area Network*

PT - *Tempo Estimado de Processamento*

TTI - *Tempo Estimado para Transferência dos Dados de Entrada*

TTO - *Tempo Estimado para Transferência dos Dados de Saída*

CT - *Tempo Total Estimado de Execução*

NWS - *Network Weather Service*

JDF - *Arquivo de Descrição do Job (Job Description File)*

Lista de Figuras

1.1 Visão geral de um *grid* computacional 1

2.1 Maximizando o paralelismo de uma aplicação 5

2.2 Resultados das simulações das heurísticas *RUMR*, *UMR* e *Weighted Factory* 10

2.3 Avaliação de desempenho da heurística de escalonamento do *middleware* *GridBus* 13

3.1 Modelo proposto de *grid* 30

5.1 Média do *makespan* em relação à disponibilidade da informação 50

5.2 Média do desperdício de recursos em relação à disponibilidade da informação 53

5.3 Média do desperdício de recursos em relação ao grau de replicação máximo 54

5.4 Média do *makespan* para cenários que utilizam aplicações do tipo *não-proporcional* 56

Lista de Tabelas

2.1 Análise comparativa entre heurísticas de escalonamento para aplicações *BoT data-intensive* . . . 22

2.2 Análise comparativa entre heurísticas de escalonamento para aplicações *BoT CPU-intensive* com informação parcial . . . 25

3.1 Tipos de informação . . . 32

5.1 Heterogeneidade do *grid* . . . 42

5.2 Resultados do *makespan* para os cenários com 10% de informação disponível 48

5.3 Resultados do *makespan* para os cenários com 25% de informação disponível 48

5.4 Resultados do *makespan* para os cenários com 50% de informação disponível 49

5.5 Resultados do *makespan* para os cenários com 75% de informação disponível 49

5.6 Resultados do *makespan* para os cenários com 100% de informação disponível 51

5.7 Resultados do *makespan* para os cenários sem replicação e 100% de informação disponível . . . 51

5.8 Resultados do *makespan* para os cenários com 50% de informação disponível utilizando aplicações do tipo *não-proporcional* . . . 55

5.9 Resultados do *makespan* para os cenários com 100% de informação disponível utilizando aplicações do tipo *não-proporcional* . . . 56

Lista de Pseudo-Códigos

1 Pseudocódigo da heurística de escalonamento XSufferage 7

2 Pseudocódigo da heurística de escalonamento RUMR 11

3 Pseudocódigo da heurística de escalonamento do *middleware* GridBus . . . 12

4 Pseudocódigo da heurística de escalonamento do *middleware* BOINC . . . 15

5 Pseudocódigo da heurística de escalonamento do WQR 17

6 Pseudocódigo da heurística de escalonamento para aplicações que possuem a carga divisível 18

7 Pseudocódigo da heurística de escalonamento do *Storage Affinity* 21

8 Pseudocódigo da heurística de escalonamento do *Adaptive WQR* 24

9 Pseudocódigo da heurística de escalonamento Adaptive Data-Intensive . . . 39

Capítulo 1

Introdução

A tecnologia de *grids* foi criada com o objetivo de facilitar o compartilhamento de recursos entre indivíduos que estão vinculados a diferentes domínios administrativos. Nos últimos anos, o uso de *grids* computacionais está cada vez mais comum devido ao grande poder computacional que esta tecnologia pode prover a um baixo custo. Este aumento no uso de *grids* computacionais também está ligado ao avanço das tecnologias de transmissão de dados, que permite que recursos localizados em diferentes regiões do planeta possam trocar informações usando canais de comunicação com grande largura de banda e baixo atraso na comunicação. A Figura 1.1 ilustra uma visão geral de um *grid* computacional.

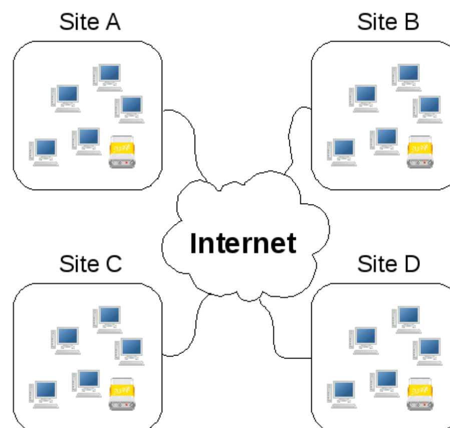


Figura 1.1: Visão geral de um *grid* computacional

Devido a isso, a execução de aplicações paralelas que processam uma grande quantidade de dados (*data-intensive*) está cada vez mais comum neste tipo de plataforma. Uma aplicação

paralela pode ser vista como uma coleção de tarefas que podem ser executadas em paralelo. Para algumas destas aplicações, essas tarefas são independentes e podem ser escalonadas para execução paralela em qualquer ordem. Este tipo de aplicação paralela é referenciada na literatura como aplicações *Bag-of-Tasks* (BoT) [1]. Neste trabalho serão consideradas apenas as aplicações que são dos tipos *data-intensive* e BoT ao mesmo tempo. Apesar da sua simplicidade, existem muitas aplicações que fazem parte desta categoria [2].

O escalonamento eficiente de aplicações BoT é tradicionalmente baseado no uso de informações sobre a infraestrutura de execução e as aplicações. Estes tipos de escalonadores, comumente chamados de escalonadores *bin-packing*, podem ser facilmente aplicados em plataformas homogêneas e dedicadas como supercomputadores e *clusters*, onde estas informações podem ser facilmente descobertas [3]. Entretanto, é sabido que é muito difícil obter informação precisa em ambientes distribuídos [4]. Este tipo de informação ainda é mais difícil de se obter em *grids* por várias razões:

- **Ambiente largamente distribuído**: Em um *grid* computacional, os recursos geralmente estão distribuídos em vários *sites* localizados em diferentes regiões. Isto ocasiona atrasos na comunicação entre os recursos que compõem o ambiente de execução, que pode causar perda da qualidade da informação recebida de outros *sites*.

- **Heterogeneidade dos recursos**: Devido ao *grid* ser um ambiente largamente distribuído, com recursos pertencentes a vários *sites* diferentes, naturalmente esses recursos acabam sendo diferentes tanto no *hardware* (arquitetura, cpu, memória, espaço em disco, etc) como no *software* (sistema operacional, serviços, aplicações instaladas, etc). Isso dificulta estimar o tempo de execução das aplicações em cada recurso do *grid*.

- **Compartilhamento de recursos**: Em muitos casos, os recursos do *grid* não são dedicados. Eles podem ao mesmo tempo executar várias aplicações de vários usuários diferentes, ocasionando constantes variações de carga nestes recursos. Com isso, dificultando a estimativa de tempo de execução feita pelas heurísticas de escalonamento.

- **Diferentes domínios administrativos**: Devido aos recursos estarem localizados em *sites* diferentes com políticas diferentes, cada administrador destes *sites* pode estabele-

cer sua própria política de acesso aos recursos, bem como acesso a informações sobre esses recursos.

Para contornar o problema da falta de informação, escalonadores baseados em replicação foram propostos na literatura [5] [6] [7] [8]. Paranhos et al. [6] mostraram que estes escalonadores podem alcançar a performance comparável à dos escalonadores *bin-packing*, com o custo de consumir ciclos de computação extra, isto é, eles trocam informação por ciclos [6] [8].

Em um trabalho recente [9], este *trade-off* foi investigado mais a fundo. Nobrega-Junior et. al. observaram que apesar de em um ambiente de *grid* a informação precisa ser difícil de se obter, o acesso a ela não é impossível. Na prática, parte da informação pode ser obtida usando serviços que coletam informações sobre os recursos e a rede [10] [11] e publicadas em um serviço de informação do *grid* (GIS) [12] [13], um serviço que provê mecanismos para obtenção de informações a respeito dos recursos disponíveis em um *grid*. Nóbrega-Júnior et al. mostraram que é possível reduzir o custo de execução de aplicações *CPU-intensive*, mantendo a mesma eficiência, usando qualquer informação que esteja disponível. No entanto tal abordagem não foi avaliada considerando as aplicações *data-intensive*.

Neste trabalho, será proposta e avaliada uma heurística de escalonamento para aplicações BoT que processam uma grande quantidade de dados (*data-intensive*), que é adaptativa à disponibilidade da informação. A heurística proposta, *Adaptive Data-Intensive*, utiliza a mesma estratégia da heurística *Storage Affinity* [8] que realiza seu escalonamento de acordo com uma métrica de afinidade, porém sem considerar qualquer tipo de informação sobre o ambiente de execução. Diferentemente da *Storage Affinity*, a heurística *Adaptive Data-Intensive* utiliza qualquer informação sobre o ambiente de execução e/ou da aplicação que esteja disponível para otimizar o tempo de execução e o desperdício de recursos.

O restante desta dissertação está organizado da seguinte forma. O Capítulo 2 apresenta uma fundamentação teórica sobre heurísticas de escalonamento em *grids* computacionais. No Capítulo 3 é apresentado o modelo para escalonamento de aplicações *BoT* em *grids* computacionais que foi utilizado neste trabalho. O funcionamento da heurística de escalonamento *Adaptive Data-Intensive* é descrito no Capítulo 4. A avaliação de desempenho e seus resultados são discutidos no Capítulo 5. Por fim, no Capítulo 6 são feitas as considerações finais deste trabalho.

Capítulo 2

Heurísticas de escalonamento em *grids* computacionais

Na maioria das vezes, heurísticas de escalonamento para aplicações BoT têm o objetivo de reduzir o *makespan* da aplicação [14], que significa o tempo decorrido desde o momento em que a aplicação foi submetida para execução até o tempo em que a última tarefa terminou. Para isso, os escalonadores tentam utilizar os recursos disponíveis de uma maneira que seja maximizada a paralelização das aplicações. Por exemplo, supondo que no início de uma execução de uma aplicação BoT existem *n* processadores disponíveis. Considerando-se T_{max} o momento em que o primeiro processador do *grid* fica ocioso porque não existem mais tarefas para executar, e T_{final} o momento em que a aplicação termina, temos que para se atingir um maior paralelismo de uma dada aplicação é necessário ter o momento T_{final} o quanto antes possível, isto é, o escalonador tem que reduzir a diferença entre T_{final} e T_{max} (ver Figura 2.1). Em um cenário ótimo teríamos $T_{max} = T_{final}$, e a utilização máxima de paralelismo no *grid* [15].

O paralelismo da aplicação é influenciado pela maneira como as tarefas são mapeadas nos recursos disponíveis. Se um escalonador tem informação completa e precisa sobre a aplicação e o ambiente de execução, ele pode fazer um mapeamento eficiente visando minimizar a diferença entre T_{final} e T_{max} e, com isso, ele pode reduzir o *makespan* da aplicação. A complexidade de uma solução ótima é alta, e portanto, usa-se heurísticas para se fazer o escalonamento. Se o mapeamento não é adequado, devido à falta de informação, por exemplo, muitos recursos podem ficar ociosos durante o intervalo entre T_{max} e T_{final}. Quanto

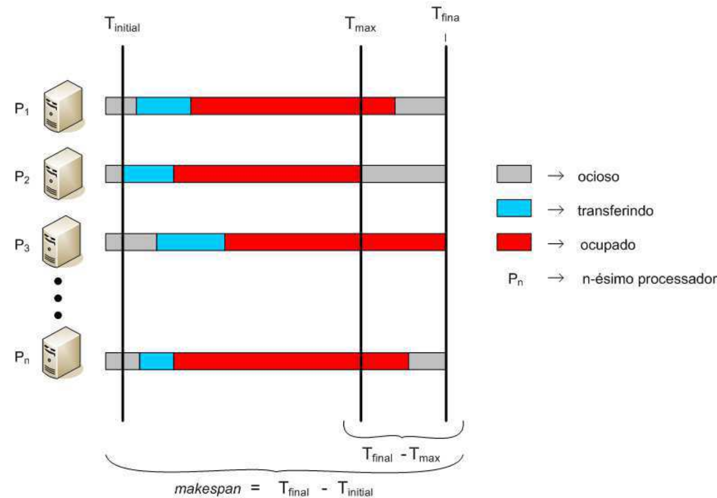


Figura 2.1: Maximizando o paralelismo de uma aplicação

mais longo é esse intervalo, menos eficiente é o escalonamento. Escalonadores baseados em replicação tentam reduzir este problema utilizando os recursos ociosos para executar réplicas adicionais de tarefas que ainda estão executando. Note que isto ocorre após o momento T_{max}. Logo que uma das réplicas de uma tarefa completa sua execução, todas as outras réplicas são canceladas. Isto tem um efeito potencial para diminuir o *makespan* da aplicação (antecipar o momento T_{final}), pois uma réplica pode finalizar sua execução antes da tarefa original que estava executando. Em contrapartida, mais recursos serão utilizados para completar a execução da aplicação.

A seguir, serão descritas algumas heurísticas de escalonamento *bin-packing* e outras baseadas em replicação. O primeiro tipo necessita de informação completa e precisa sobre o ambiente de execução e a aplicação, enquanto o segundo realiza um processo de replicação no final da execução para lidar com a falta de informação e manter uma boa performance. Após isso, serão apresentados e comentados resultados de trabalhos anteriores que confrontaram esses dois tipos de heurísticas de escalonamento.

2.1 Heurísticas *bin-packing*

Várias heurísticas de escalonamento *bin-packing* foram propostas com o objetivo de escalonar tarefas independentes em ambientes heterogêneos [3] [16] [17] [18] [19] [20] [21] [22].

Ainda mais, existem algumas heurísticas propostas para escalonar aplicações *data-intensive* [16] [23] [24] [25]. Estas heurísticas necessitam da disponibilidade de informação completa e precisa sobre o ambiente de execução e das aplicações. As informações sobre os recursos podem ser estáticas (velocidade do CPU, tamanho da memória) ou dinâmicas (carga da CPU, espaço em disco), enquanto que a informação sobre as aplicações é normalmente uma estimativa de tempo que cada tarefa precisa para terminar sua computação em cada recurso disponível. Para aplicações *data-intensive*, é preciso considerar informação sobre pares de recursos como largura de banda e latência da rede, visto que os tempos de transferências de dados podem aumentar significativamente o *makespan* da aplicação. Com esse tipo de informação nas mãos, estas heurísticas analisam qual o melhor mapeamento de tarefas e recursos, de forma que o *makespan* da aplicação seja minimizado.

2.1.1 *XSufferage*

A *XSufferage* [16] é um exemplo de heurística *bin-packing* para escalonar aplicações *data-intensive*. Ela escalona tarefas de acordo com a performance dos recursos do ambiente de execução. A heurística *XSufferage* é uma extensão da heurística *Sufferage* [26]. O objetivo da heurística *Sufferage* é calcular o quanto uma tarefa seria prejudicada se ela não fosse escalonada no recurso que a executaria de forma mais eficiente. Então, a heurística *Sufferage* escolhe tarefas de acordo com o valor de prejuízo. Este valor é chamado de *sufferage* e ele é calculado como a diferença entre o melhor e o segundo melhor tempo de execução esperado, levando em consideração todos os recursos do *grid*. A principal diferença entre as heurísticas *XSufferage* e *Sufferage* é a forma como elas calculam o valor *sufferage*. A primeira considera a transferência dos dados de entrada e saída das tarefas quando é calculada a estimativa de tempo de execução. Isto implica que a *XSufferage* usa as mesma informações que a heurística *Sufferage* usa, mais a informação da largura de banda entre os recursos e o tamanho dos dados de entrada e saída das tarefas. A heurística *Sufferage* apenas usa informações relacionadas à velocidade de CPU dos recursos para prever o tempo de execução de uma tarefa.

O algoritmo da *XSufferage* é apresentado em pseudocódigo no Algoritmo 1. O algoritmo de escalonamento da heurística *XSufferage* funciona da seguinte forma: para cada tarefa da aplicação, é identificado o *site* no qual a tarefa terá o menor tempo de execução (Linhas 11 a 14). Com o *site* identificado, o valor do prejuízo é calculado levando em consideração os

recursos com o menor tempo de execução e o segundo menor (Linhas 16 e 17). Por fim, quando todas as tarefas tiverem o seu valor de prejuízo calculado, a tarefa com o maior valor de prejuízo será escalonada (Linhas 19 a 25), porque esta causaria o maior prejuízo se rodasse no recurso com segundo menor tempo de execução, então deve-se começar por ela. O processo de escalonamento pára quando não há mais tarefas a serem escalonadas (Linha 3).

Algoritmo 1 Pseudocódigo da heurística de escalonamento XSufferage

```
 1 FUNCTION schedule( Grid grid, Job job )
 2
 3 WHILE( job.hasTaskNotScheduled() ) DO
 4    maxSufferage = MIN
 5    nextTask = NULL
 6    nextResource = NULL
 7    FOR( Task task : job.getTasksNotScheduled() ) DO
 8       bestCompletionTime = MAX
 9       bestSite = NULL
10       FOR( Site site : grid.getSites() ) DO
11          IF( getCompletionTime( task, site ) < bestCompletionTime ) THEN
12             bestCompletionTime = getBestCompletionTime( task, site )
13             bestSite = site
14       ENDFOR
15
16       secondBestCompletionTime = getSecondBestCompletionTime( task, bestSite )
17       sufferage = secondBestCompletionTime - bestCompletionTime
18
19       IF( sufferage > maxSufferage ) THEN
20          maxSufferage = sufferage
21          nextTask = task
22          nextResource = getResourceWithBestCompletionTime( task, bestSite )
23    ENDFOR
24
25    ASSIGN( nextTask, nextResource )
26 ENDWHILE
```

Os resultados obtidos através de simulações são apresentados no trabalho feito por Casanova et al. [16], eles mostraram que a heurística *XSufferage* obteve bons resultados comparados aos resultados das outras heurísticas confrontadas. As heurísticas utilizadas para comparação foram:

- **Min-min**: Esta heurística utiliza o mínimo MCT (*Minimun Completation Time*) como métrica. Ela priorizada para o escalonamento a tarefa que executará mais rápido.

- **Max-min**: Utiliza a métrica máximo MCT, onde é prioriza para o escalonamento a tarefa que irá demorar mais para finalizar sua execução.

- **Workqueue**: Também conhecida como WQ, essa heurística associa as tarefas aos recursos de maneira aleatória.

- **Sufferage**: Heurística na qual *XSufferage* foi baseada. Esta por sua vez não leva em consideração a informação sobre a rede (largura de banda, latência, etc).

De acordo com os resultados apresentados, a heurística *XSufferage* obteve a média do *makespan* 7% melhor que a heurística *Sufferage*, 11% melhor que a Max-min, 15% melhor que a Min-min e 36% melhor que a WQ, nos cenários sem pertubação. Para testar melhor a eficiência das heurísticas, em alguns cenários foram introduzidas pertubações. A pertubação de um cenário consiste em introduzir algumas depedências entre as tarefas da aplicação. Para este tipo de cenário, a XSufferage foi 8% melhor que a Sufferage, 10% melhor que a Max-min, 15% melhor que a Min-min e 40% melhor que a WQ.

2.1.2 *RUMR*, *UMR* e *Weighted Factory*

A *RUMR* [24] é outro exemplo de heurística *bin-packing* para escalonar aplicações *data-intensive*, especificamente aplicações que possuem a carga divisível (*divisible workloads*). Normalmente, em aplicações com a carga divisível, a heurística de escalonamento fica responsável por dividir os dados de entrada em pedaços menores formando as tarefas da aplicação. Nesta heurística, tarefas com granularidades pequenas (tarefas com tamanho pequeno para os dados de entrada) são escalonadas no início da execução da aplicação. Isso objetiva colocar os recursos para processar o mais rápido possível, evitando que recursos fiquem por muito tempo ociosos no início da execução, aguardando a transferência dos dados. Durante o processo de escalonamento, a granularidade da tarefa vai aumentando enquanto a aplicação está executando. Após certo ponto, perto do final da execução da aplicação, a granularidade da tarefa diminui visando evitar o escalonamento de tarefas longas no final da execução da aplicação.

A heurística RUMR é uma extensão da heurística UMR. O que diferencia as duas heurísticas é que a RUMR é otimizada para lidar com erro na informação sobre o ambiente de execução, característica comum em um ambiente de *grid*.

O processo de escalonamento da heurística UMR é realizado da seguinte forma: inicialmente, tarefas com a granularidade pequena são escalonadas, com essa granularidade aumentado ao longo da execução da aplicação com o intuito de obter um paralelismo entre transferência da tarefa para o recurso e a execução da tarefa no recurso.

Em um *grid* computacional, onde é comum o erro na informação, escalonar tarefas grandes no final da execução não é uma boa ideia, pois uma tarefa grande pode ser associada a um recurso sobrecarregado, desta forma afetando substancialmente no tempo de execução da aplicação. Outra estratégia utilizada para o escalonamento de aplicações que possuem a carga divisível é a *Weighted Factory* [27]. Nesta estratégia, o escalonamento inicia com tarefas com granularidade grande e ao longo da execução esta granularidade vai diminuindo, ou seja, o seu comportamento é o contrário da heurística UMR. O problema nesta solução, ao contrário da heurística UMR, está no início da execução, onde escalonar tarefas com as maiores granularidades no início da execução vai fazer com que haja uma demora para os recursos começarem a executar as tarefas, pois elas levarão um bom tempo para serem transmitidas. Isto pode impactar o paralelismo entre transmissão e execução.

A heurística RUMR utiliza as vantagens das duas heurísticas acima. Da mesma forma que no UMR, a granularidade vai aumentando ao longo da execução para se obter um paralelismo entre transferência e execução. Mas diferentemente da UMR, a RUMR após uma certa quantidade da aplicação processada, adota o mesmo comportamento da heurística *Weighted Factory*, onde as tarefas vão diminuindo a granularidade ao longo da execução com o objetivo de ter tarefas com granularidade pequena ao final da execução da aplicação. Esta estratégia diminui o impacto que o erro na informação causa no *makespan* da aplicação.

O Algoritmo 2 apresenta o pseudocódigo da heurística RUMR. No começo, o algoritmo de escalonamento da heurística *RUMR* cria um conjunto de tarefas iniciais com granularidade pequena (Linha 3). As tarefas são, então, escalonadas para os recursos (Linhas 10 a 15). Após isso, a heurística entra na fase *UMR*, onde será calculado o número de iterações que ocorrerá nesta fase (Linha 19). Nesta fase, a cada iteração que ocorre, a granularidade das tarefas que os recursos recebem aumenta (Linhas 20 a 24). Quando a última iteração da

fase *UMR* termina, a heurística entra na fase *Weighted Factory*. Nesta fase, a granularidade das tarefas vai diminuindo a cada iteração até que toda a carga da aplicação seja processada (Linhas 31 a 41).

De acordo com o trabalho de Yang et al. [24], os resultados obtidos pela heurística *RUMR* são apresentados na Figura 2.2. Os resultados são definidos como a percentagem de cenários em que a heurística *RUMR* obteve melhor *makespan* que as demais, separados por percentagem do erro na informação disponível. Os resultados obtidos através das simulações mostraram que a RUMR consegue manter um bom desempenho mesmo quando o erro na informação aumenta, enquanto que as heurísticas UMR e *Weighted Factory* são prejudicadas.

Algorithm	Ranges of $error$				
	0–0.08	0.1–0.18	0.2–0.28	0.3–0.38	0.4–0.48
UMR	54.96	56.60	73.45	81.99	86.48
Factoring	98.21	94.06	93.84	90.16	84.74

Figura 2.2: Resultados das simulações das heurísticas *RUMR*, *UMR* e *Weighted Factory*

2.1.3 Heurística de escalonamendo do *middleware GridBus*

O *GridBus* [28] é um *middleware* de *grid* que estende o *middleware Nimrod-G* [29] adicionando funcionalidades para executar aplicações que são ao mesmo tempo *data-intensive* e *parameter-sweep* (varredura de parâmetros), um tipo particular de aplicação BoT. Algumas dessas funcionalidades adicionais são: acesso remoto a repositórios de dados, otimização para transmissão de dados e suporte para execução de aplicações com parâmetros dinâmicos. No *GridBus*, o processo de escalonamento é feito de acordo com a proximidade entre os recursos computacionais e os repositórios de dados. O poder computacional dos recursos também é levado em consideração. Então, é utilizada uma função para calcular o melhor mapeamento entre tarefas e recursos.

No Algoritmo 3, é mostrado o pseudocódigo da heurística de escalonamento do *middleware GridBus*. O algoritmo de escalonamento do *GridBus* funciona de forma iterativa, a cada iteração, o algoritmo escalona uma tarefa (nomeada como *job* no trabalho de Venugopal et al. [28]) em um recurso onde sua execução irá terminar mais rápido (Linhas 12 a 17), levando em consideração os atributos deste recurso e a largura de banda entre o repositório

Algoritmo 2 Pseudocódigo da heurística de escalonamento RUMR

```
1 FUNCTION schedule( Grid grid, Job job )
2
3    job.createInitialTasks()
4
5    scheduleFirstRound( grid, job )
6    scheduleUMRPhase( grid, job )
7    scheduleWFPhase( grid, job )
8 ENDFUNCTION
9
10 FUNCTION scheduleFirstRound( Grid grid, Job job )
11
12    FOR( Task task : job.getTasksNotScheduled() ) DO
13       ASIGN( task, task.getResource() )
14    ENDFOR
15 ENDFUNCTION
16
17 FUNCTION scheduleUMRPhase( Grid grid, Job job )
18
19    umrIteractions = getNumberOfUMRIteractions( grid, job )
20    FOR( int i = 1; i <= umrIterations; i++ ) DO
21       FOR( Resource resource : grid.getResources() ) DO
22          job.addTask( createNextUMRPhaseTask( resource ) )
23       ENDFOR
24    ENDFOR
25
26    FOR( Task task : job.getTasksNotScheduled() ) DO
27       ASIGN( task, task.getResource() )
28    ENDFOR
29 ENDFUNCTION
30
31 FUNCTION schedulerWFPhase( Grid grid, Job job )
32
33    WHILE(! job.getWorkload().isCompleteProcessed() )
34       FOR( Resource resource : grid.getResources() ) DO
35          job.addTask( createNextWFPhaseTask( resource ) )
36       ENDFOR
37    ENDWHILE
38
39    FOR( Task task : job.getTasksNotScheduled() ) DO
40       ASIGN( task, task.getResource() )
41    ENDFOR
42 ENDFUNCTION
```

de dados, onde os dados da tarefa estão, e o recurso (Linhas 12). O algoritmo continua executando este laço até que não tenha mais tarefas a serem escalonadas (Linha 2). Se em um dado momento não há recurso disponível para execução, o escalonador espera até que um novo recurso fique ocioso para poder escalonar a próxima tarefa.

Algoritmo 3 Pseudocódigo da heurística de escalonamento do *middleware* GridBus

```
 1 FUNCTION schedule( Grid grid, Job job )
 2    WHILE( job.hasTaskNotScheduled() ) DO
 3       bestCompletationTime = MAX
 4       bestDataHost = NULL
 5       bestResource = NULL
 6       bestTask = NULL
 7
 8       FOR( Task task : job.getTasksNotScheduled() ) DO
 9          FOR( DataHost dataHost : grid.getDataHosts() ) DO
10             FOR( Resource resource : grid.getResources() ) DO
11
12                tempCompletationTime = getCompletationTime( task, dataHost, resource )
13                IF( tempCompletationTime < bestCompletationTime ) THEN
14                   bestCompletationTime = tempCompletationTime
15                   bestDataHost = dataHost
16                   bestResource = resource
17                   bestTask = task
18             ENDFOR
19          ENDFOR
20       ENDFOR
21
22       ASSIGN( bestTask, bestDataHost, bestResource )
23    ENDWHILE
24 ENDFUNCTION
```

A avaliação de desempenho desta heurística foi feita no trabalho de Venugopal et al. [28]. Esta heurística foi comparada com duas outras heurísticas que não utilizam qualquer tipo de informação sobre o ambiente de execução. A primeira heurística executa as tarefas da aplicação apenas nos recursos localizados nos *sites* onde estão localizados os repositórios de dados, não havendo transferência dos dados de entrada das tarefas entre *links* de diferentes *sites*. A segunda heurística executa as tarefas nos recursos de forma aleatória, independentemente da localização dos recursos e dos repositórios de dados. Os resultados da avaliação de desempenho são apresentados na Figura 2.3. Na figura, o Eixo x representa o tempo de execução, enquanto que o Eixo y representa cada heurística avaliada. De acordo com os

resultados apresentados, é possível observar que o escalonamento que utilizou informações sobre o ambiente de execução conseguiu reduzir substancialmente o tempo de execução da aplicação quando comparado aos escalonamentos que não utilizaram informação alguma.

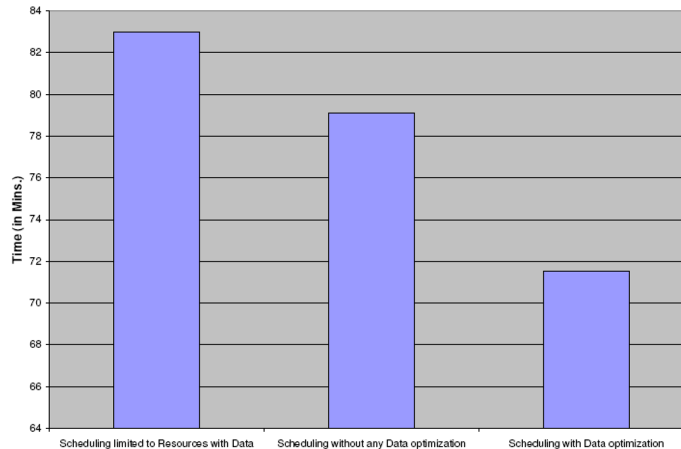


Figura 2.3: Avaliação de desempenho da heurística de escalonamento do *middleware GridBus*

2.1.4 Computação Voluntária

BOINC[1] e *World Community Grid*[2] são projetos de *grid* para computação voluntária (*voluntary computing*). Neste tipo de projeto, usuários domésticos disponibilizam seus computadores quando estes estão ociosos. Estes projetos são bastante conhecidos devido aos tipos de aplicações que são executadas em seus *grids*. Uma das mais conhecidos representantes desse tipo de aplicação é a aplicação do projeto SETI@home[3] que visa procurar vida extra terrestre através da captura de sinais de rádio no espaço. A maior parte dessas aplicações tem características de aplicações *data-intensive*. A heurística de escalonamento no *BOINC* [30] utiliza um *software* agente nos computadores voluntários. Este agente realiza testes de performance (*benchmarks*) nestes computadores, obtendo assim informações de desempenho e disponibilidade. Então, o escalonador cria uma tarefa para um determinado computador de acordo com as informações que foram obtidas sobre ele.

[1] http://boinc.berkeley.edu/
[2] http://www.worldcommunitygrid.org/
[3] http://setiathome.berkeley.edu/

O Algoritmo 4 apresenta o pseudocódigo do algoritmo de escalonamento do *BOINC*. O escalonamento no *BOINC* ocorre em duas etapas:

1. **Seleção das tarefas** (tratadas como *jobs* no trabalho de Anderson et al. [30]): Nesta etapa é criada a lista de execução de tarefas, onde são colocadas as tarefas que são preparadas de acordo com a configuração dos recursos disponíveis para execução. No *BOINC*, recursos disponíveis não é o mesmo que recursos ociosos. Durante a execução de uma tarefa em um recurso, esta tarefa pode ter sua execução cancelada e o recurso pode ser escalonado para executar em uma nova tarefa. Mais detalhes sobre cancelamento da execução são descritos na próxima etapa.

2. **Execução das tarefas**: Nesta etapa as tarefas são escalonadas nos recursos para execução. No *BOINC*, cada tarefa tem o seu *deadline* especificado. Deste modo, o escalonador submete as tarefas aos recursos de maneira iterativa, onde em cada *loop* do algoritmo, é escalonada a tarefa cujo *deadline* está mais próximo (Linhas 4). Quando não existem recursos ociosos e o escalonador precisa submeter uma tarefa, será escolhido o recurso cuja tarefa fez *checkpoint* a menos tempo (Linhas 13), e deste modo, esse recurso é escalonado para executar essa nova tarefa. As tarefas do *BOINC* frequentemente fazem *checkpoint* da sua execução com o objetivo de diminuir o desperdício de recursos ocasionados pelas preempções dos recursos.

No caso do BOINC, o seu escalonador não visa diminuir o *makespan* das aplicações. Como descrito anteriormente, sua função é escalonar tarefas adequadas aos recursos disponíveis e diminuir o desperdício de recursos preemptando os recursos cujas tarefas fizeram *checkpoint* a menos tempo.

2.2 Heurísticas baseadas em replicação

Como discutido anteriormente, heurísticas baseadas em replicação replicam tarefas que ainda estão executando para contornar penalidades de performance de possíveis mapeamentos mal feitos de tarefas a recursos. *Workqueue with Replication* (WQR) é uma heurística de escalonamento baseada em replicação para aplicações *BoT CPU-intensive* [6], enquanto que *Storage Affinity* [8] é uma heurística de replicação otimizada para aplicações *data-intensive*.

Algoritmo 4 Pseudocódigo da heurística de escalonamento do *middleware* BOINC

```
 1 FUNCTION schedule( Grid grid, JobCollection jobs )
 2
 3    WHILE( jobs.hasUnfinishedJob() AND grid.hasAvailableResource() ) DO
 4       nextJob = jobs.getJobWithEarliestDeadline()
 5       nextResource = grid.getNextAvailableResource()
 6       ASIGN( nextJob.createTask( nextResource ), nextResource )
 7    ENDWHILE
 8
 9    SLEEP( SLICE_TIME )
10
11    WHILE( jobs.hasMissedDeadlineJob() ) DO
12       nextJob = jobs.getJobWithEarliestDeadline()
13       nextResource = grid.preemptResourceWithLatestCheckpoint()
14       ASIGN( nextJob.createTask( nextResource ), nextResource )
15    ENDWHILE
16 ENDFUNCTION
```

Já o trabalho de Assis et. al. [15] apresenta uma heurística de escalonamento baseada em replicação para aplicações que possuem a carga divisível (*divisible workloads*). Elas obtêm, em ambientes completamente sem informação, um *makespan* que é comparável ao obtido por heurísticas *bin-packing* em ambientes com informação completa e precisa. Entretanto a boa performance é obtida com o ônus de consumir mais recursos devido à computação redundante das réplicas que são iniciadas no final da execução da aplicação.

2.2.1 *Workqueue with Replication*

A *Workqueue With Replication* (*WQR*) [6] é uma heurística baseada em replicação para executar aplicações *BoT*. Ela tem o seu funcionamento parecido com a clássica heurística *Workqueue* (*WQ*), onde tarefas são escalonadas de forma aleatória em recursos ociosos no *grid*. A diferença entre essas duas heurísticas está no fato de que a heurística *WQR* realiza o processo de replicação de tarefas quando ainda existem tarefas executando e máquinas ociosas no *grid*. Este processo de replicação é feito com o o objetivo de diminuir o *makespan* da aplicação, evitando que uma tarefa fique executando por muito tempo em um recurso com pouco poder computacional. Note que apesar da replicação ocasionar uma melhora do *makespan*, ela acaba gerando um desperdício de recursos. A heurística *WQR* ainda define um fator máximo para o número de réplicas de uma tarefa, visando diminuir o desperdício de

recursos. Por exemplo, se este fator é 3, uma tarefa só poderá ter no máximo 3 réplicas, isto é, uma tarefa com 2 réplicas, só é replicada novamente se todas as outras tarefas já têm pelo menos duas réplicas.

No trabalho de Paranhos et al. [6] são comparados os desempenhos das heurísticas *WQR*, *WQ*, *Sufferage* e *Dynamic FPLTF*[4]. Os resultados apresentados no trabalho mostraram que a heurística *WQR* obteve o *makespan* ligeiramente melhor que o das outras heurísticas comparadas na maioria dos cenários simulados. Porém, com o desperdício de recursos devido à replicação de tarefas no final da execução.

O Algoritmo 5 apresenta o pseudocódigo do algoritmo de escalonamento da heurística *WQR*. Primeiramente a heurística escalona as tarefas nos recursos de maneira aleatória (Linhas 3 a 7). Se não houver recurso disponível, o algoritmo espera até que um recurso fique disponível (Linhas 4 e 5). No final da execução, quando não há mais tarefas para escalonar, mas ainda existem tarefas executando e também recursos disponíveis, a heurística entra na fase de replicação. Nesta fase, tarefas que ainda estão executando são replicadas em recursos disponíveis (Linhas 9 a 16). Quando uma tarefa finaliza sua execução, obrigatoriamente todas as réplicas dessa tarefa são canceladas (Linhas 18 a 20).

2.2.2 Heurística de escalonamento para aplicações que possuem a carga divisível

A heurística apresentada no trabalho de Assis et al. [15] é baseada em replicação para aplicações que possuem a carga divisível. Diferentemente da heurística RUMR, apresentada anteriormente, esta heurística não utiliza nenhuma informação sobre o ambiente de execução para realizar o escalonamento. A estratégia desta heurística para calcular o tamanho ideal da tarefa é monitorar frequentemente a vazão do *grid*. Isto permite a heurística saber qual a granularidade que deixa o *grid* com maior vazão.

A variação da granularidade das tarefas durante o processo de escalonamento é similar à variação da heurística *RUMR*, onde, no início do escalonamento, tarefas menores são submetidas, com este tamanho aumentando ao longo da execução. No final da execução, tarefas

[4]A heurística *Dynamic FPLTF* (*Fastest Processor to Large Task First*) é uma variação da heurística *FPLTF* [31], que é uma heurística *bin-packing* para aplicações *CPU-intensive*.

Algoritmo 5 Pseudocódigo da heurística de escalonamento do WQR

```
 1 FUNCTION schedule( Grid grid, Job job, int replicationFactor )
 2
 3    WHILE( job.hasTaskNotScheduled() ) DO
 4       IF(! grid.hasAvailableResource() ) THEN
 5          sleepUntilThereIsAvailableResource()
 6       ASSIGN( job.getNextUnscheduledTask(), grid.getNextAvailableResource() )
 7    ENDWHILE
 8
 9    WHILE( job.hasAllTasksScheduled() AND job.hasRunningTask() ) DO
10       IF(! grid.hasAvailableResource() ) THEN
11          sleepUntilThereIsAvailableResource()
12       nextTask = job.getNextRunningTaskWithMinReplicas()
13       IF( nextTask.getNumberOfReplicas() < replicationFactor ) THEN
14          REPLICATE( nextTask, grid.getNextAvailableResource() )
15    ENDWHILE
16 ENDFUNCTION
17
18 FUNCTION whenTaskFinishes( Task task )
19    task.cancelAllRunningReplicas()
20 ENDFUNCTION
```

pequenas são escalonadas para que a etapa de replicação seja feita com tarefas menores para não se ter um alto desperdício de recursos.

O Algoritmo 6 apresenta o pseudocódigo do método de escalonamento desta heurística. O funcionamento do algoritmo de escalonamento ocorre da seguinte maneira: inicialmente são criadas tarefas com granularidade pequena, e estas são escalonadas para os recursos (Linhas 3 a 8). Quando um recurso finaliza a execução de uma tarefa tornando-se disponível, a vazão do *grid* é calculada (Linhas 12 e 13). Se a vazão do *grid* aumentou em relação à última vazão calculada, a granularidade das novas tarefas a serem criadas é aumentada, caso a vazão seja menor, a granularidade é diminuída (Linhas 15 a 18). O algoritmo continua neste laço até que a carga processada da aplicação atinja a fase final (Linha 5), onde tarefas com granularidade pequena são escalonadas (Linhas 21 a 28). Quando toda a carga da aplicação for processada, a heurística ainda pode entrar na fase de replicação (Linhas 30 a 36).

No trabalho de Assis et al. [15], esta heurística foi comparada com a *RUMR* e a *WQR*, ambas descritas anteriormente. Para utilizar a heurística *WQR* com este tipo de aplicação, a carga da aplicação foi fatiada em tarefas de tamanhos iguais antes de iniciar o processo de escalonamento.

Algoritmo 6 Pseudocódigo da heurística de escalonamento para aplicações que possuem a carga divisível

```
1 FUNCTION schedule( Grid grid, Job job, int replicationFactor )
2
3    job.createFirstsTasks()
4
5    WHILE( job.processedWorkload() < lastPhaseWorkload ) DO
6       WHILE( grid.hasAvailableResources() ) DO
7          ASSIGN( job.createTask(), grid.nextAvailableResource() )
8       ENDWHILE
9
10      sleepUntilThereIsAvailableResource()
11
12      lastThroughput = currentThroughput
13      currentThroughput = grid.calculateThroughtput()
14
15      IF( currentThroughput > lastThroughput ) THEN
16         job.increaseTaskSize()
17      ELSE IF( currentThroughput < lastThroughput ) THEN
18         job.decreaseTaskSize()
19   ENDWHILE
20
21   job.decreaseTaskSizeToMinimum()
22
23   WHILE( job.processedWorkload() < completeWorkload ) DO
24      WHILE( grid.hasAvailableResources() ) DO
25         ASSIGN( job.createTask(), grid.nextAvailableResource() )
26      ENDWHILE
27      sleepUntilThereIsAvailableResource()
28   ENDWHILE
29
30   WHILE( job.hasAllTasksScheduled() AND job.hasRunningTask() ) DO
31      IF(! grid.hasAvailableResource() ) THEN
32         sleepUntilThereIsAvailableResource()
33      nextTask = job.getNextRunningTaskWithMinReplicas()
34      IF( nextTask.getNumberOfReplicas() < replicationFactor ) THEN
35         REPLICATE( nextTask, grid.getNextAvailableResource() )
36   ENDWHILE
37 ENDFUNCTION
38
39 FUNCTION whenTaskFinishes( Task task )
40   task.cancelAllRunningReplicas()
41 ENDFUNCTION
```

Os resultados das simulações mostraram que esta heurística obteve resultados melhores que a heurística *RUMR*, mesmo não utilizando informações sobre o ambiente de execução, devido à replicação de tarefas no final da execução. Por outro lado, por ser uma heurística baseada em replicação, houve desperdício de recursos. Porém, esta heurística obteve um desperdício de recursos em média menor que a heurística *WQR*, porque ao contrário da *WQR*, esta heurística controla a granularidade das tarefas, utilizando tarefas pequenas no final da execução e consequentemente durante o processo de replicação.

2.2.3 *Storage Affinity*

A *Storage Affinity* [8] é uma heurística baseada em replicação para executar aplicações *BoT data-intensive*. Ela foi proposta para tornar mais eficiente a execução de aplicações *BoT* que repetitivamente processam uma grande quantidade de dados e que reutiliza uma porção significativa dos dados de entrada, como por exemplo as aplicações de visualização de resultados de experimentos científicos. A *Storage Affinity* associa tarefas a recursos de acordo com a afinidade entre eles. Esta afinidade é calculada entre uma tarefa e um recurso e corresponde ao número de *bytes* que são processados pela tarefa que já estão armazenados no *site* ao qual o recurso pertence. A heurística calcula a afinidade de cada tarefa com todos os recursos, desta forma, escalonando a tarefa com a maior afinidade no recurso que este valor foi obtido. Este processo é repetido de maneira iterativa até que todas as tarefas sejam escalonadas. Como na heurística *WQR*, a *Storage Affinity* também aplica a técnica de replicação para lidar com a ausência de informação sobre o ambiente de execução. Entretanto, a *Storage Affinity* segue um critério específico para criar uma réplica, diferentemente de *WQR* que replica as tarefas de maneira aleatória. A *Storage Affinity* somente replica uma tarefa, se algum outro recurso do mesmo *site* estiver ocioso, evitando que os dados de entrada da tarefa sejam novamente transferidos, pois eles já estão lá armazenados.

O modelo de *grid* computacional que a heurística *Storage Affinity* necessita para realizar o seu processo de escalonamento deve ser especificado como: o *grid* deve ser formado por um ou mais *sites*. Cada *site* é composto por recursos computacionais (executam tarefas) e um recursos para armazenamento de dados. Todos os recursos computacionais devem ter acesso ao recurso para armazenamento de dados do seu *site*.

A razão para o uso deste modelo se deve ao fato de que uma vez que aplicações *data-*

intensive transmitem uma grande quantidade de dados e também podem ser executadas mais de uma vez mantendo os mesmos dados de entrada, os dados transmitidos ficam armazenado nos recursos de armazenamento não precisando ser transmitidos novamente.

Da mesma maneira que a heurística *WQR*, a *Storage Affinity* também não necessita de informação alguma sobre o ambiente de execução (velocidade e carga dos recursos, *links*). A única informação que é necessária para a heurística realizar o seu escalonamento é a quantidade de *bytes* dos dados de entrada das tarefas que já estão armazenados no recurso de armazenamento de cada *site*. É com essa informação que *Storage Affinity* calcula a afinidade entre as tarefas e os recursos.

O Algoritmo 7 apresenta o pseudocódigo do algoritmo de escalonamento da heurística *Storage Affinity*. O funcionamento do algoritmo ocorre da seguinte maneira: primeiramente, para cada tarefa da aplicação, é selecionado um *site* cuja tarefa tem o maior valor de afinidade. A cada iteração do laço, a tarefa que possuir um maior valor para a afinidade é escalonada (Linhas 3 a 17). Quando não existir mais tarefas para serem escalonadas, a heurística ainda pode entrar na fase de replicação Linhas (19 a 26).

No trabalho de Santos-Neto et al. [8], a heurística *Storage Affinity* é comparada com as heurísticas *XSufferage* e *WQR*. A primeira é uma heurística *bin-packing* otimizada para aplicações *data-intensive*. A segunda é uma heurística baseada em replicação. Ambas já foram explicadas anteriormente.

A heurística *WQR* foi comparada com a *Storage Affinity* com o objetivo de mostrar que uma heurística que não é otimizada para aplicações *data-intensive* não obtém boa performance quando está executando este tipo de aplicação. Isto ocorre porque a *WQR* associa tarefas a recursos de forma aleatória. Com isso, durante uma replicação ou re-execução, a heurística pode escalonar uma tarefa a um recurso pertencente a um *site* que não armazena os dados de entrada desta tarefa, tendo que transmiti-los novamente, acarretando em desperdício de banda e afetando também o *makespan* da aplicação. Na Tabela 2.1 temos os resultados obtidos na análise comparativa dessas heurísticas.

Como podemos observar, as heurísticas *Storage Affinity* e *XSufferage* tiveram resultados no *makespan* bem parecidos. Note que para obter esse resultado, a heurística *XSufferage* necessitou de informação completa e precisa sobre o ambiente de execução e a aplicação. Por outro lado, a heurística *Storage Affinity* não precisou desse tipo de informação para

Algoritmo 7 Pseudocódigo da heurística de escalonamento do *Storage Affinity*

```
 1 FUNCTION schedule( Grid grid, Job job, int replicationFactor )
 2
 3    WHILE( job.hasTaskNotScheduled() ) DO
 4       FOR( Task task : job.getTasksNotScheduled() ) DO
 5          IF(! grid.hasAvailableResource() ) THEN
 6             sleepUntilThereIsAvailableResource()
 7          FOR( Site site : grid.getSitesWithAvailableResource() ) DO
 8             tempAffinity = calculateAffinity( task, site )
 9             IF( tempAffinity > maxAffinity ) THEN
10               maxAffinity = tempAffinity
11               nextTask = task
12               nextSite = site
13          ENDFOR
14       ENDFOR
15
16       ASSIGN( nextTask, nextSite.getAvailableResource() )
17    ENDWHILE
18
19    WHILE( job.hasTaskNotFinished() ) DO
20       IF(! grid.hasAvailableResource() ) THEN
21          sleepUntilThereIsAvailableResource()
22       nextTask = job.getNextRunningTaskWithMinReplicas()
23       IF( nextTask.getNumberOfReplicas() < replicationFactor ) THEN
24           Site site = grid.getBestAffinitySitesWithAvailableResource( nextTask ).nextSite()
25           REPLICATE( nextTask, site.getAvailableResource() )
26    ENDWHILE
27 ENDFUNCTION
28
29 FUNCTION whenTaskFinishes( Task task )
30    task.cancelAllRunningReplicas()
31 ENDFUNCTION
```

Heurística	*Makespan* (segundos)	Desperdício
Storage Affinity	14337s	62%
WQR	42919s	131%
XSufferage	14665s	-

Tabela 2.1: Análise comparativa entre heurísticas de escalonamento para aplicações *BoT data-intensive*

obter esse resultado. Ela utilizou a técnica de replicação para obter um bom resultado no *makespan*, ao custo de um desperdício de recursos. Já na heurística *WQR*, os altos valores para o *makespan* e desperdício ocorreram porque a *WQR* não é uma heurística otimizada para aplicações *data-intensive*.

Com esses resultados, é possível afirmar que as heurísticas de replicação são bastante eficientes quando utilizadas em ambientes de *grids* computacionais, que são ambientes dinâmicos e heterogêneos onde a informação é imprecisa ou não está disponível, porém, esse bom desempenho está associado a um custo computacional maior.

2.3 *Adaptive WQR*

Em um trabalho recente [9], foi observado que apesar de o acesso à informação ser difícil em um *grid* computacional, esta informação não é impossível de se obter, mesmo sendo apenas informação sobre parte do *grid*. Naquele trabalho foi mostrado que é possível reduzir o custo de execução de aplicações *CPU-intensive*, mantendo a mesma eficiência, usando qualquer informação que esteja disponível.

A *Adaptive WQR* [9] é uma heurística de escalonamento para aplicações *BoT CPU-intensive* capaz de utilizar qualquer informação disponível sobre o ambiente de execução com o intuito de diminuir o desperdício de recursos durante o processo de replicação. Assim como a *WQR*, ela também utiliza replicação com o objetivo de melhorar o *makespan* da aplicação.

No Algoritmo 8, é apresentado o pseudocódigo da heurística *Adaptive WQR*. O processo de escalonamento da heurística *Adaptive WQR* ocorre da seguinte maneira: primeiramente se o escalonador tiver informação sobre a aplicação (custo de execução das tarefas em um

processador padrão), o *grid* é dividido em dois grupos, um grupo com recursos que possuem informação disponível e outro grupo com os recursos sem informação disponível (Linhas 3 e 4). Após isso, as tarefas da aplicação são ordenadas de maneira decrescente de acordo com seus custos. Depois disso, as tarefas são escalonadas no grupo com informação utilizando a heurística *Dynamic FPLTF* (Linha 8). Quando não há mais recursos do primeiro grupo disponíveis, as tarefas ainda não escalonadas são submetidas aos recursos do grupo sem informação utilizando a heurística *WQR* (Linha 8). Por outro lado, se a heurística não possui informação sobre a aplicação, todas as tarefas são escalonadas nos recursos usando a heurística *WQR* (Linha 11).

Quando o processo de escalonamento chega na etapa de replicação (Linhas 13 e 14), ou seja, quando há recursos ociosos e tarefas ainda executando, uma tarefa somente será replicada se uma das alternativas abaixo for verdadeira:

- Se o escalonador possuir informação sobre a aplicação e sobre o recurso que está executando a tarefa, esta tarefa somente será replicada se o tempo estimado para finalizar a nova réplica for menor que o tempo estimado para finalizar a tarefa que está sendo executada. Para isso, o escalonador também deve ter informação sobre o recurso ocioso que irá executar a réplica (Linhas 21 a 24).

- Se o escalonador possuir apenas informação sobre o recurso que está executando a tarefa, esta tarefa somente será replicada se o recurso ocioso for mais rápido que o recurso que está executando a tarefa. Note que o escalonador também precisa ter informação sobre o recurso ocioso (Linhas 26 a 30).

- Se o escalonador não tiver informação sobre o recurso que está executando a réplica, a tarefa sempre será replicada (Linhas 32 e 33).

No trabalho que foi realizado por Nobrega-Junior et. al. [9], uma análise comparativa foi feita com o objetivo de avaliar heurísticas de escalonamento para aplicações *BoT CPU-intensive* com informação parcial. Naquele trabalho, foi visto que apesar da informação sobre o ambiente de execução ser de difícil acesso num *grid* computacional, ela não é impossível de se obter, mesmo que não seja completa. Isto quer dizer que podemos ter informação apenas sobre parte do ambiente de execução. As heurísticas analisadas naquele

Algoritmo 8 Pseudocódigo da heurística de escalonamento do *Adaptive WQR*

```
 1 FUNCTION schedule( Grid grid, Job job, int replicationFactor )
 2
 3    gridInfo = grid.getResourcesWithInfo()
 4    gridNoInfo = grid.getResourcesWithoutInfo()
 5
 6    IF( job.hasInfo() ) THEN
 7       job.sortTaskInDecreasingOrderByCost()
 8       scheduleUsingDFPLTF( gridInfo, job )
 9       scheduleUsingWQR( gridNoInfo, job )
10    ELSE
11       scheduleUsingWQR( grid, job )
12
13    WHILE( job.hasAllTasksScheduled() AND job.hasRunningTask() ) DO
14       IF(! grid.hasAvailableResource() ) THEN
15          sleepUntilThereIsAvailableResource()
16       nextTask = job.getNextRunningTaskWithMinReplicas()
17       IF( nextTask.getNumberOfReplicas() < replicationFactor ) THEN
18          runningReplica = nextTask.getLastRunningReplica()
19          newReplica = nextTask.createNewReplica( grid.getNextAvailableResource() )
20
21          IF( job.hasInfo() AND runningReplica.getResource().hasInfo() AND
22                      newReplica.getResource().hasInfo() ) THEN
23             IF( newReplica.getEstimatedCost() < runningReplica.getEstimatedRemainingCost() ) THEN
24                REPLICATE( newReplica, newReplica.getResource() )
25
26          IF( job.hasNoInfo() AND runningReplica.getResource().hasInfo() AND
27                      newReplica.getResource().hasInfo() ) THEN
28             IF( newReplica.getResource().getCPUPower() >
29                                      runningReplica.getResource().getCPUPower() ) THEN
30                REPLICATE( newReplica, newReplica.getResource() )
31
32          IF( runningReplica.getResource().hasNoInfo() ) THEN
33                REPLICATE( newReplica, newReplica.getResource() )
34    ENDWHILE
35 ENDFUNCTION
36
37 FUNCTION whenTaskFinishes( Task task )
38    task.cancelAllRunningReplicas()
39 ENDFUNCTION
```

trabalho foram *WQR*, *Adaptive WQR* e *DFPLTF* [6]. Os resultados obtidos são reproduzidos na Tabela 2.2.

Nível de informação →	0% − 10%		30% − 70%		90% − 100%	
Heurística ↓	*Makespan*(s)	Desperdício (%)	*Makespan*(s)	Desperdício (%)	*Makespan*(s)	Desperdício (%)
Adaptive WQR	13151s	159%	12437s	44%	10502s	19%
WQR	11488s	197%	11491s	195%	11511s	197%
DFPLTF	116637s	−	28183s	−	17684s	−

Tabela 2.2: Análise comparativa entre heurísticas de escalonamento para aplicações *BoT CPU-intensive* com informação parcial

De acordo com a Tabela 2.2, a comparação feita entre as heurísticas foi separada em três tipos de cenários distintos. O primeiro cenário considera os ambientes de execução onde existe informação disponível para apenas alguns *sites* (entre 0% e 10% dos *sites*), o segundo considera que a informação está disponível para 30% e 70% dos *sites*, enquanto que no terceiro cenário entre 90% e 100% dos *sites* têm informação disponível. Estes cenários foram propostos com intuito de verificar o comportamento da heurística *Adaptive WQR* sendo executada em ambientes com vários níveis de informação, desde ambientes com pouca informação até ambientes com informação quase completa. De acordo com os resultados apresentados, a heurística *Adaptive WQR* obteve valores para o *makespan* comparáveis aos valores da heurística *WQR*, porém com um desperdício de recursos que vai diminuindo quando o nível de informação do ambiente de execução vai aumentando.

Para a heurística *WQR*, os níveis de informação sobre o ambiente de execução não influenciaram nos resultados tanto do *makespan* quanto no do desperdício, porque a heurística *WQR* não utiliza nenhuma informação sobre o ambiente para realizar o seu processo de escalonamento. Mas como podemos observar, mesmo sem utilizar nenhuma informação, a heurística *WQR* obteve bons resultados no *makespan*. Por outro lado, o seu desperdício de recursos permaneceu alto em todos os cenários executados.

A heurística *bin-packing DFPLTF* em todos os cenários obteve o valor do *makespan* pior do que as heurísticas de replicação. Isto ocorreu devido ao fato de que a *DFPLTF* apenas utilizar a parte do ambiente de execução que possui informação, ou seja, ela só utilizou os recursos que tinham informação. Isto explica o valor alto do *makespan* para os cenários com pouca informação. A *DFPLTF* não possui desperdício de recursos porque ela é uma

heurística *bin-packing*.

De acordo com os resultados obtidos pela heurística *Adaptive WQR* apresentados no trabalho de Nobrega-Junior et. al. [9], foi mostrado que é possível diminuir o desperdício de recursos durante o escalonamento de aplicações *CPU-intensive* (ocasionado pelo processo de replicação) utilizando qualquer informação que esteja disponível sobre o ambiente de execução e/ou a aplicação.

Este resultado motivou investigar se é possível utilizar o mesmo tipo de informação que esteja disponível para melhorar o escalonamento de aplicações que processam uma grande quantidade de dados (*data-intensive*). Para este tipo de aplicação, esta estratégia pode não somente diminuir o desperdício de recursos, como também pode diminuir o tempo de execução da aplicação (*makespan*). Esta diminuição no *makespan* se dá pelo fato de que com aplicações *data-intensive*, transferências de longa duração serão necessárias, com isso o escalonador pode identificar os recursos com melhores *links*.

Capítulo 3

Modelo para escalonamento de aplicações *BoT* em *grids* computacionais

Este capítulo apresentará o modelo para escalonamento de aplicações *BoT* em *grids* computacionais que foi utilizado para definir a heurística de escalonamento *Adaptive Data-Intensive*. Este modelo também foi implementado no simulador onde foram executadas as simulações realizadas para apresentação dos resultados desta dissertação.

O modelo se baseia no proposto por Nobrega-Junior et. al. [32] que por sua vez foi inspirado na arquitetura de *grids* entre-pares (*P2P grids*), mais especificamente no *OurGrid*[1] [33]. O OurGrid é um *grid* computacional P2P, aberto e em produção desde dezembro de 2004. A comunidade de usuários oferece poder computacional a qualquer usuário interessado em se juntar ao grupo e executar suas aplicações *BoT*. Seu poder computacional é obtido através dos recursos ociosos dos seus participantes e é compartilhado de tal forma que recebe mais recursos quem oferece mais recursos ao *grid*. Esse comportamento é garantido pelo uso da rede de favores (*network of favors*) [34] como política de compartilhamento dos recursos.

3.1 Modelo de *grid*

Um grid G é composto por um conjunto de *sites*. Cada *site* é constituído por processadores, máquinas base e servidores de dados. Os processadores são responsáveis por executar as tarefas das aplicações. A máquina base é o local a partir de onde o usuário submete sua apli-

[1] www.ourgrid.org

cação, é lá que é executado o escalonador. Já o servidor de dados é utilizado para armazenar os dados de entrada (*input*) e os dados de saída (*output*) das tarefas. Mais formalmente, um *grid* pode ser definido como:

$$G = \{site_1, site_2, ..., site_g\}, g > 0;$$

$$site_i = P_i \cup S_i \cup B_i;$$

$$S_i \neq \emptyset \wedge P_i \cup B_i \neq \emptyset,$$

onde P_i é o conjunto de processadores do $site_i$, B_i é o conjunto de máquinas base do $site_i$ e S_i é o conjunto de servidores de dados do $site_i$. Note que um *site* deve ter pelo menos um processador ou uma máquina base, bem como pelo menos um servidor de dados.

Para deixar o modelo proposto mais simples, será levado em consideração que um *site* tenha no máximo uma máquina base e apenas um servidor de dados. Desta maneira, temos:

$$S_i = \{\sigma_i\} \text{ } e \text{ } B_i = \{\beta_i\} \vee B_i = \emptyset,$$

onde σ_i corresponde ao servidor de dados e β_i à máquina base.

Formalmente, é possível definir o conjunto de todos os processadores do *grid* como:

$$P_g = \bigcup_{i=1}^{|G|} P_i$$

Cada processador é definido por uma tupla formada por dois atributos: velocidade (*cpu_speed*) e percentual do poder de processamento disponível (*cpu_avail*). De maneira formal:

$$\pi = (cpu_speed, cpu_avail), \forall \pi \in P_g,$$

onde *cpu_speed* corresponde à velocidade do processador π e *cpu_avail* ao poder de processamento de π que está disponível. É importante mencionar que o atributo *cpu_avail* varia seu valor ao longo do tempo, ou seja, ele é um atributo dinâmico.

O modelo de rede que foi adotado neste trabalho também foi definido por Nobrega-Junior et. al. [32]. Este modelo assume que os processadores, máquina base e o servidor de dados de um mesmo *site* estão conectados através de uma rede local (*LAN*) de alta velocidade. Além disso, todos os *sites* que compõem o *grid* estão interligados por uma rede de longa

distância (*WAN*) com limitação de banda e alta latência. Vale salientar que todos os *sites* possuem conectividade direta para todos os outros *sites* do *grid*, ou seja, visualizando o *grid* como um grafo onde os *sites* seriam os nós e a conectividade direta entre dois *sites* seria uma aresta, este grafo seria completamente conectado.

Com isso, é possível definir o conjunto de todos os *links* do *grid* como:

$$L_g = \{\lambda_{i,j} | 1 \leq i, j \leq g\}, g = |G|$$

onde $\lambda_{i,j}$ representa um *link* entre os $site_i$ e $site_j$, quando $i \neq j$ e $\lambda_{i,i}$ quando o *link* conecta recursos dentro do mesmo *site*.

Cada *link* é definido por uma tupla formada por três atributos: largura de banda para conexão (*bw_con*), limite da largura de banda (*bw_lim*) e latência (*lat*). De maneira formal:

$$\lambda = (bw_con, bw_lim, lat), \forall \lambda \in L_g$$

onde *bw_con* corresponde à largura de banda que uma conexão recebe quando é aberta. O atributo *bw_lim* corresponde à largura de banda máxima que o *link* pode fornecer levando em consideração todas as conexões abertas neste *link*. Por fim, o atributo *lat* corresponde à latência do *link*, ou seja, o tempo entre o início de uma requisição e o momento da chegada da mesma.

A razão para existir o atributo *bw_lim* em um *link* é para evitar com que muitas conexões sejam abertas em paralelo utilizando o mesmo *link* e sempre obtenham a mesma largura de banda. Por exemplo, quando uma conexão precisa ser aberta, é verificado se esta conexão irá fazer com que a vazão do *link* ultrapasse o valor de *bw_lim*. Em caso afirmativo, cada conexão receberá a largura de banda correspondente à fração do valor de *bw_lim* com o número de conexões que foram abertas naquele *link*. Caso contrário, cada conexão receberá a largura de banda correspondente ao valor do atributo *bw_con*.

Com esse modelo de *links*, é possível modelar tanto *links* de uma rede de longa distância (*WAN*), quanto *links* para uma rede local (*LAN*). Por exemplo, para criar um *link WAN*, basta atribuir um valor para *bw_con* que seja menor que o valor de *bw_lim*. Por exemplo, *10Mbps* e *40Mbps*, respectivamente. Neste caso cada conexão obteria uma largura de banda de *10Mbps* até o limite de quatro conexões. A partir daí a largura de banda obtida seria proporcional ao número de conexões ativas. Por exemplo, com cinco conexões, cada conexão receberia

8Mbps de largura de banda, resultado de *bw_lim* dividido pelo número de conexões. Por outro lado, para criar um *link LAN*, basta atribuir o mesmo valor para *bw_con* e *bw_lim*.

A Figura 3.1 ilustra um *grid* com três *sites* especificado pelo modelo proposto, onde o *Site* 1 contem três processadores e um servidor de dados conectados pelo *Link* $\lambda_{1,1}$, o *Site* 2 contem dois processadores, um processador base e um servidor de dados que estão conectados pelo *Link* $\lambda_{2,2}$ e o *Site* 3 contem um processador base e um servidor de dados conectados pelo *Link* $\lambda_{3,3}$. Os *Sites* 1 e 2 estão conectados pelos *Links* $\lambda_{1,2}$ e $\lambda_{2,1}$ e o mesmo acontece para os *Links* entre os *Sites* 1 e 3, e 2 e 3.

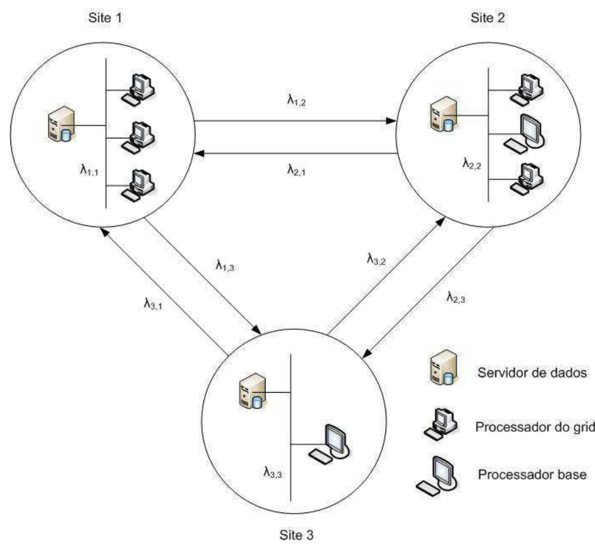


Figura 3.1: Modelo proposto de *grid*

3.2 Modelo da aplicação

Um *job* é formado por um conjunto de *n tasks* independentes e que podem ser executadas em qualquer ordem. De maneira formal, um *job J* é definido como:

$$J = \{\tau_1, \tau_2, ..., \tau_n\}$$

onde τ_n corresponde à n-ésima *task* do *job*.

Uma *task* τ é modelada por três atributos:

- $\tau.input_data$: Quantidade em *bytes* dos dados de entrada da *task*;

- $\tau.output_data$: Quantidade em *bytes* dos dados de saída da *task*;
- $\tau.computational_cost$: Tempo em segundos estimado de execução da *task* em um processador referência com $\pi.cpu_speed = 1$ e $\pi.cpu_avail = 100\%$ durante toda execução.

Além disso, existem quatro medidas de desempenho de uma *task* que são utilizadas durante o processo de escalonamento: tempo estimado para processamento (*PT*), tempo estimado para transferência dos dados de entrada (*TTI*), tempo estimado para transferência dos dados de saída (*TTO*) e tempo total de execução estimado (*CT*). Formalmente temos:

$$PT_{k,\pi} = \frac{\tau_k.computational_cost}{\pi.cpu_speed \times \pi.cpu_avail}$$

onde $PT_{k,\pi}$ corresponde ao tempo estimado para que a *task k* seja executada pelo processador π.

$$TTI_{k,i,j} = \lambda_{i,i}.lat + \lambda_{i,j}.lat + \lambda_{j,j}.lat + \frac{\tau_k.input_data}{min(\lambda_{i,i}.bw_avail, \lambda_{i,j}.bw_avail, \lambda_{j,j}.bw_avail)}$$

$$TTO_{k,i,j} = \lambda_{i,i}.lat + \lambda_{i,j}.lat + \lambda_{j,j}.lat + \frac{\tau_k.output_data}{min(\lambda_{i,i}.bw_avail, \lambda_{i,j}.bw_avail, \lambda_{j,j}.bw_avail)}$$

onde $TTI_{k,i,j}$ corresponde à transferência dos dados de entrada da *task k* do processador base β_i do $site_i$ para o processador π_j e $TTO_{k,i,j}$ corresponde à transferência dos dados de saída do processador π_j do $site_j$ para a máquina base β_i.

$$CT_{k,\pi} = TTI_{k,i,j} + PT_{k,\pi} + TTO_{k,i,j}$$

onde $CT_{k,\pi}$ corresponde ao tempo estimado de execução de uma *task k* que foi submetida a partir de um processador base β_i pertencente ao $site_i$ para ser executada em um processador π no $site_j$.

3.3 Modelo da informação

As informações que são utilizadas pelo modelo descrito estão divididas em três categorias: informação sobre a rede, informação sobre os recursos e informação sobre a aplicação.

As informações sobre a rede que foram descritas anteriormente (latência, largura de banda dos *links*, etc) podem ser obtidas através de ferramentas de monitoramento como o NWS [35] ou o IPerf [36].

Já as informações sobre os recursos representam o poder computacional provido pelos mesmos como velocidade de CPU e carga disponível para processamento. Essas informações podem ser obtidas por meio de ferramentas de monitoramento de recursos como o Ganglia [10].

A informação sobre a aplicação é obtida através de um arquivo de descrição com informações sobre a aplicação. No *middleware OurGrid*, por exemplo, o usuário descreve sua aplicação através do arquivo JDF (*Job Description File*). Informações da aplicação que são importantes durante o processo de escalonamento podem ser: tamanho dos arquivos de entrada, vazão de cada tarefa, etc.

Além disso, as informações podem ser divididas em mais duas categorias: informações estáticas e informações dinâmicas. As informações estáticas são aquelas que não mudam durante todo o processo de escalonamento. Já as informações dinâmicas podem mudar. A Tabela 3.1 lista todos os tipos de informação em suas respectivas categorias.

	Estática	**Dinâmica**
Rede	Limite de largura de banda	Latência, Largura de banda
Recurso	Velocidade de CPU	Disponibilidade de processamento
Aplicação	Tamanho da entrada, Vazão das tarefas	-

Tabela 3.1: Tipos de informação

Formalmente, temos:

$$\Im_{net} = \{lat, bw_con, bw_lim\}$$

$$\Im_{proc} = \{cpu_speed, cpu_avail\}$$

$$\Im_{app} = \{input_set, output_set, computational_cost\}$$

onde $\Im_{net}$ corresponde ao conjunto de informações sobre a rede, $\Im_{proc}$ corresponde ao conjunto de informações sobre os recursos e $\Im_{app}$ corresponde ao conjunto de informações sobre a aplicação.

Como discutido anteriormente, um dos maiores desafios de um ambiente distribuído como um *grid* computacional é a obtenção de informação precisa sobre a infraestrutura de execução. Para modelar essa dificuldade, foram utilizados dois parâmetros que representam a disponibilidade e o erro na informação provida para a heurística de escalonamento. Esses parâmetros são bastante importantes pois eles modelam as condições reais de um ambiente distribuído. Os parâmetros de disponibilidade e erro da informação são definidos formalmente como:

$$i = (\delta, e), \forall_i \in \Im$$

$$\delta \in \{true, false\}$$

$$\Im = \Im_{net} \bigcup \Im_{proc} \bigcup \Im_{app}$$

onde $i.\delta$ e $i.e$ correspondem respectivamente à disponibilidade e a percentagem de erro da informação *i*. A informação está disponível quando $i.\delta = true$, caso contrário $i.\delta = false$.

3.4 Escalonamento

O escalonamento de uma aplicação (*job*) em um *grid* computacional é o processo de submissão das tarefas desse *job* em recursos do grid. O escalonamento representa o mapeamento da execução do *job*, ou seja, em quais recursos as tarefas executarão. Em escalonadores que usam replicação, uma tarefa pode ter mais de uma réplica que por sua vez pode ser executada em recursos distintos. Neste modelo é considerado que em um *job* que finalizou sua execução com sucesso, cada tarefa tem apenas uma única réplica que finalizou sua execução com sucesso. Formalmente, temos:

$$\Sigma_j = \{\mu_1, \mu_2, ..., \mu_n\}, n \geq |J|$$

$$\mu_i = \{\pi, \tau, cp\}, \forall \mu_i \in \Sigma_j$$

$$\forall \tau_k \in J, \exists \mu_i.\tau = \tau_k \wedge \mu_i.cp = 1$$

onde Σ_j é o conjunto de todas as réplicas executadas durante o escalonamento, inclusive as que não chegaram a terminar sua execução. μ_i representa uma réplica submetida, onde $\mu_i.\pi$ corresponde ao recurso π em que essa réplica i foi executada, $\mu_i.\tau$ corresponde à tarefa τ

que a réplica i pertence e $\mu_i.cp$ corresponde ao percentual da réplica i que foi processada pelo recurso π. Neste caso, o valor de $\mu_i.cp$ é um número real entre 0 e 1. Quando a tarefa é executada com sucesso, temos $\mu_i.cp = 1$.

3.5 Métricas de desempenho

Um dos objetivos principais da maioria das heurísticas de escalonamento é a redução do tempo de execução, também conhecido como *makespan* [14], da aplicação. Devido a isso, a métrica *makespan* será utilizada para medir o desempenho das heurísticas analisadas neste trabalho. Por definição, o *makespan* da aplicação corresponde ao tempo entre o momento em que a primeira tarefa é escalonada até o momento em que a última tarefa escalonada finalizou a transferência dos seus dados de saída (veja Figura 2.1, Página 5).

De maneira formal, *makespan* é definido como:

$$makespan = T_{final} - T_{initial},$$

onde T_{final} corresponde ao momento em que a última tarefa escalonada finaliza a transferência dos dados de saída e $T_{initial}$ corresponde ao momento em que é escalonada a primeira tarefa da aplicação.

Como o principal objetivo da heurística de escalonamento descrita nesse trabalho é reduzir o desperdício de recursos e manter um *makespan* comparável ao das heurísticas totalmente baseadas em replicação, para isso, utilizando qualquer informação que esteja disponível sobre a aplicação e o ambiente de execução, é preciso utilizar uma métrica que represente o nível de desperdício de recursos da execução de uma aplicação em um dado *grid*. Formalmente, o nível de desperdício de recursos é definido como:

$$ResourceWaste = \frac{badput}{goodput},$$

onde o $goodput$ equivale ao somatório dos tempos de execução de todas as réplicas finalizadas com sucesso e o $badput$ corresponde ao somatório dos tempos de execução de todas as réplicas que foram canceladas. Note que uma réplica em execução é cancelada quando outra réplica da mesma tarefa finaliza sua execução com sucesso. Formalmente, $goodput$ e $badput$ são definidos como:

$$goodput = \sum_{k=1}^{|J|} CT_{\mu_k.\tau,\mu_k.\pi}, \forall \mu \in \Sigma_j \wedge \mu_k.cp = 1$$

$$badput = \sum_{k=1}^{|J|} CT_{\mu_k.\tau,\mu_k.\pi}, \forall \mu \in \Sigma_j \wedge \mu_k.cp < 1$$

Capítulo 4

Heurística Adaptive Data-Intensive

Este capítulo descreve a heurística de escalonamento Adaptive Data-Intensive. O funcionamento desta heurística é baseado no funcionamento das heurísticas *Adaptive WQR* que foi descrita no trabalho de Nobrega-Junior et al. [9] e *Storage Affinity* que foi descrita no trabalho de Santos-Neto et al. [8], ambas descritas anteriormente. Da heurística *Adaptive WQR*, a heurística *Adaptive Data-Intensive* utiliza o mecanismo de escalonamento adaptativo à disponibilidade da informação sobre a aplicação e o ambiente de execução. Já da heurística *Storage Affinity*, a heurística *Adaptive Data-Intensive* herda a funcionalidade para escalonar tarefas aos recursos de acordo com a afinidade entre eles.

O processo de escalonamento da heurística é feito da seguinte forma: a heurística recebe uma coleção de tarefas (*job*) e uma coleção de recursos (*grid*) como parâmetro. Após isso, as tarefas são escalonadas nos recursos de acordo com a afinidade entre eles. Essa afinidade é calculada da mesma forma que a heurística *Storage Affinity* calcula, que corresponde à quantidade dos dados de entrada da tarefa, em *bytes*, que já está armazenada no *site* onde o recurso está localizado.

Primeiro, a afinidade de todas as tarefas com todos os recursos é calculada. Então, a tarefa com o maior valor de afinidade é escolhida. No caso de mais de uma tarefa ter o mesmo valor de afinidade, é escolhida a tarefa com maior quantidade de dados de entrada. Após a tarefa com o maior valor de afinidade ser escolhida, se esta tarefa tem o mesmo valor de afinidade para mais de um *site*, então é escolhido o melhor *site* para aquela tarefa de acordo com as seguintes regras.

A heurística escolhe o *site* contendo pelo menos um recurso disponível que possui o

menor custo de acordo com a equação que calcula a métrica *CT* descrita no capítulo anterior. Essa equação calcula o tempo de execução de uma tarefa em um dado recurso somando o tempo de transferência dos dados de entrada com o tempo de execução e o tempo de transferência dos dados de saída.

Como a heurística *Adaptive Data-Intensive* precisa lidar com a falta de informação sobre a aplicação e/ou o ambiente de execução, vai haver momentos em que a heurística não terá acesso a todas as informações que ela precisa para fazer o cálculo do *CT*. A solução utilizada pela heurística para contornar esse problema é agrupar os recursos de acordo com a disponibilidade de informação, onde cada grupo terá recursos com o mesmo tipo de informação disponível. O Grupo G_1 terá os recursos com informação completa, já o Grupo G_2 terá os recursos com apenas a informação sobre a rede, o Grupo G_3 terá os recursos com apenas a informação sobre eles e por fim, o Grupo G_4 possui os recursos que não têm informação alguma.

Quando não há dados de entrada das tarefas já armazenados nos recursos de armazenamento dos *sites* do *grid*, os recursos do Grupo G_1 executarão as maiores tarefas, seguido pelos recursos do Grupo G_2, seguido pelos recursos do Grupo G_3 e por fim, seguido pelos recursos do Grupo G_4. A razão para a ordem de execução desses grupos se deve ao fato de que a heurística *Adaptive Data-Intensive* segue uma abordagem otimista, onde ela trata os recursos que possuem mais informação como sendo os melhores. Neste caso, os recursos do Grupo G_2 têm prioridade para serem escalonados em relação aos recursos do Grupo G_3 porque para aplicações *data-intensive*, é mais crítico uma tarefa ser transferida por um *link* ruim ou congestionado do que uma tarefa executar em um recurso ruim ou sobrecarregado.

Como descrito no Capítulo 2, as heurísticas de escalonamento baseadas em replicação replicam tarefas no final da execução quando não há mais tarefas a serem escalonadas e há recursos ociosos. Em trabalhos recentes [8] [6] já foi mostrado que a replicação é uma boa estratégia para lidar com a falta de informação sobre o ambiente de execução. Além disso, no trabalho realizado por Nobrega-Junior et. al. [9] foi visto que é possível reduzir o desperdício de recursos em aplicações *CPU-intensive* utilizando qualquer informação que esteja disponível sobre a aplicação e/ou o ambiente de execução. Devido a isso, um dos objetivos da heurística *Adaptive Data-Intensive* é reduzir o desperdício de recursos durante o escalonamento de aplicações *data-intensive*. A descrição da etapa de replicação da heurística

Adaptive Data-Intensive é explicada a seguir.

A etapa de replicação da heurística *Adaptive Data-Intensive* é feita de forma similar à heurística *Storage Affinity*, onde uma tarefa somente é replicada em um recurso que pertença ao *site* cujo servidor de dados já armazena os arquivos de dados de entrada da tarefa. Se mais de um *site* possui os arquivos de entrada da tarefa, então é escolhido o *site* de acordo com as regras já definidas acima onde é calculado o valor de *CT* para cada recurso disponível pertencente a esses *sites*. É importante mencionar que para este caso o valor da métrica *TTI*, que corresponde à transferência dos dados de entrada para o servidor de dados do *site*, é zero, porque os dados já estão armazenados no servidor de dados do *site*, não necessitando uma nova transferência. Desta forma, os Grupos G_1 e G_3 estão unidos e da mesma forma os recursos dos Grupos G_2 e G_4.

Similar à heurística *Adaptive WQR*, além da escolha do recurso certo para replicar a tarefa, a heurística *Adaptive Data-Intensive* ainda possui um fator de replicação que limita o número de réplicas que uma tarefa pode ter. Além disso, a heurística *Adaptive Data-Intensive* somente replicará a tarefa no recurso escolhido se uma das alternativas descritas a seguir for verdadeira:

- Uma tarefa que está executando em um recurso do Grupo G_1 só será replicada em outro recurso do Grupo G_1 e se o tempo estimado para finalizar a tarefa que está executando é maior que o tempo estimado de execução da nova réplica. Neste caso a heurística de escalonamento estimou que a nova réplica terminará antes da tarefa que está executando.

- Uma tarefa que está executando em um recurso do Grupo G_3 só será replicada em outro recurso dos Grupos G_1 e G_3 e se o tempo estimado para finalizar a tarefa que está executando é maior que o tempo estimado de execução da nova réplica.

- Uma tarefa que está executando em um recurso dos Grupos G_2 ou G_4 será sempre replicada.

É importante mencionar que apesar de não ser necessário a transferência dos dados de entrada das tarefas durante o processo de replicação, pois os dados de entrada já estão armazenados no *site*, é preciso levar em consideração as informações de rede porque os dados de saída das tarefas ainda serão transferidos para a máquina base.

O Algoritmo 9 equivale ao pseudocódigo do processo de escalonamento da heurística *Adaptive Data-Intensive*.

Algoritmo 9 Pseudocódigo da heurística de escalonamento Adaptive Data-Intensive

```
 1 FUNCTION schedule( Grid grid, Job job, int replicationFactor )
 2
 3  job.sortTasksOnDecreasingOrderBy( Job.DATA_INPUT )
 4
 5  /* Alocation Phase */
 6  WHILE( job.hasTaskNotScheduled() ) DO
 7
 8    FOR( Task task : job.getTasksNotScheduled() ) DO
 9      IF(! grid.hasAvailableProcessors() ) THEN
10        sleep()
11      Collection< Site > bestAffinitySites = grid.getBestAffinitySites( task )
12      Site site = getBestSite( bestAffinitySites, task )
13      ASSIGN( task, site.getBestProcessor() )
14    ENDFOR
15  ENDWHILE
16
17  /* Replication Phase */
18  WHILE( job.hasTaskNotFinished() ) DO
19    IF(! grid.hasAvailableProcessors() ) THEN
20      sleep()
21    FOR( Task task : job.getReplicableRunningTasks( replicationFactor ) ) DO
22      Collection< Site > bestAffinitySites = grid.getBestAffinitySites( task )
23      Resource resource = getBestResource( bestAffinitySites, task )
24      Replica runningReplica = task.getLastReplica()
25      Replica newReplica = task.newReplica( resource )
26      IF( newReplica.getCost() < runningReplica.getRemainingCost() ) THEN
27        REPLICATE( newReplica )
28      ENDIF
29    ENDFOR
30  ENDWHILE
31 ENDFUNCTION
```

De acordo com o pseudocódigo, a Linha 1 equivale à assinatura do método onde a função *schedule* recebe como parâmetros um *job*, um *grid* e o fator de replicação, este último equivale ao número máximo de réplicas que uma tarefa pode ter. Na Linha 3, as tarefas do *job* são ordenadas de forma decrescente de acordo com o tamanho dos seus dados de entrada. A fase de alocação corresponde ao bloco entre as Linhas 6 e 15, essa fase permanece executando até que todas as tarefas sejam escalonadas. Se não houver recursos disponíveis

no *grid*, o escalonador vai “dormir”, Linhas 9 e 10. Na Linha 11, são selecionados os *sites* com maior afinidade com a tarefa corrente do laço e na Linha 12, o melhor *site* é escolhido de acordo com a ordem de escalonamento entre os grupos criados. Note que a criação dos grupos está implícita neste método. Então, a tarefa é escalonada para o melhor recurso deste *site* (Linha 13). Se este *site* não possui informação sobre os recursos, um recurso disponível é escolhido de forma aleatória.

A fase de replicação, bloco entre as Linhas 18 e 30, é executada até que todas as tarefas finalizem sua execução. Nas Linhas 19 e 20, da mesma forma que na fase de alocação, se não houver recursos disponíveis, o escalonador vai “dormir”. O bloco entre as Linhas 21 e 29 vai iterar sobre todas as tarefas que estão executando e podem ser replicadas. Neste caso, podem haver tarefas que já atingiram o número máximo de réplicas (fator de replicação), desta forma, elas não são iteradas neste laço. Novamente são selecionados os *sites* com maior afinidade com a tarefa corrente (Linha 22). Após isso, o melhor recurso é escolhido para aquela tarefa (Linha 23), usando as prioridades dos grupos mostradas anteriormente. Então o custo da réplica é comparado com o custo restante da tarefa que está executando (Linha 26). Se o custo da nova réplica for menor que o custo restante da tarefa que está executando, a tarefa é replicada (Linha 27).

Capítulo 5

Avaliação de desempenho

Este capítulo descreve os resultados da avaliação de desempenho da heurística de escalonamento *Adaptive Data-Intensive* que foi descrita no Capítulo 4. A heurística *Adaptive Data-Intensive* foi comparada com as heurísticas *Storage Affinity*, *Adaptive WQR* e *XSufferage*. A heurística *Storage Affinity* foi escolhida por ser bastante eficiente para aplicações *data-intensive* em ambientes sem informação. Por outro lado, *Adaptive WQR* foi escolhida por utilizar qualquer informação disponível sobre o ambiente de execução e/ou da aplicação para diminuir o desperdício de recursos durante o processo de escalonamento de aplicações *CPU-intensive*. Por fim, a heurística *XSufferage* foi escolhida por ser bastante eficiente para aplicações *data-intensive*, mas para ambientes com informação completa. Como a *XSufferage* necessita de informação completa, a ideia é mostrar a perda de desempenho que ela tem quando temos apenas parte do ambiente de execução com informação.

A avaliação de desempenho foi feita através de simulações feitas com cada uma das heurísticas citadas. As métricas comparadas foram o *makespan* (tempo de execução da aplicação) e a nível de desperdício de recursos. Os cenários executados e os resultados obtidos serão descritos a seguir.

Para executar as simulações, foi utilizado o *framework* de simulação *SimGrid* [37]. O *SimGrid* provê funcionalidades para simulação da execução de aplicações paralelas em ambientes distribuídos. Foi desenvolvido um simulador voltado para o escalonamento de aplicações *BoT* em um *grid* computacional que utiliza o modelo descrito no Capítulo 3, com implementações das heurísticas que serão comparadas.

5.1 Descrição dos cenários

Com o objetivo de investigar o dinamismo e a heterogeneidade do ambiente de *grid* computacional durante o processo de escalonamento, as simulações foram executadas considerando um grande conjunto de cenários distintos. As características de cada cenário variavam de acordo com os seguintes aspectos: i) heterogeneidade dos recursos do *grid*; ii) granularidade da aplicação iii) heterogeneidade das tarefas da aplicação; iv) quantidade de informação disponível do ambiente de execução para as heurísticas que utilizam informação; e, v) grau de replicação máximo para as heurísticas que utilizam replicação.

5.1.1 Heterogeneidade do *grid*

Nas simulações, foi utilizado um *grid* composto por 60 recursos que juntos têm uma velocidade de 600. Esses recursos foram distribuídos uniformemente em vários *sites*. O número de *sites* utilizados, por sua vez, foi uniformemente distribuído entre 5 e 10 *sites*. Como um *grid* pode ser composto por recursos que foram adquiridos em diferentes épocas, é bastante comum os *grids* serem bastante heterogêneos. Para definir os parâmetros da heterogeneidade do *grid*, foi utilizada como referência a Lei de Moore [38], que diz que a velocidade dos processadores quase duplica a cada 18 meses. Assim, a velocidade dos processadores aumenta 8 vezes em um período de 54 meses, que é o período médio de depreciação total de um processador.

Foram utilizados quatro tipos de heterogeneidade do *grid*, cada um obedecendo à distribuição uniforme mostrados na Tabela 5.1.

Heterogeneidade do *grid*	**Distribuição da velocidade dos recursos**
1x	U(10, 10)
2x	U(6.7, 10.4)
4x	U(4, 16)
8x	U(2.2, 17.6)

Tabela 5.1: Heterogeneidade do *grid*

Como os recursos podem ser compartilhados com outros usuários, também foi simulado

a carga dos recursos durante a execução. Para isso, foi utilizado arquivos de *trace* que foram criados a partir da monitoração de recursos reais através da ferramenta NWS [35]. O NWS *toolkit* é um serviço distribuído que monitora a performance de recursos computacionais e os *links* que interconectam esses recursos.

5.1.2 Granularidade e heterogeneidade das tarefas da aplicação

Aplicações *BoT* podem ter desde aplicações com todas as suas tarefas do mesmo tamanho, ou seja, mesma quantidade de dados de entrada, ou também podem ter tarefas com tamanhos variados. Para englobar todos esses tipos de aplicações, foram criados vários tipos de cenários com níveis diferentes para a heterogeneidade das tarefas de acordo com a granularidade da aplicação. A granularidade da aplicação refere-se à média do tamanho dos dados de entrada de todas as tarefas da aplicação, enquanto que a heterogeneidade das tarefas corresponde a uma medida da variação uniforme do tamanho dos dados de entrada de cada tarefa da aplicação em relação à granularidade da aplicação.

Foram considerados cinco fatores de heterogeneidade: 0%, 25%, 50%, 75% e 100%. Para calcular o tamanho de entrada de cada tarefa, durante a criação do cenário para simulação, as tarefas eram criadas de acordo com a distribuição uniforme $U(g*(1-h/2), g*(1+h/2))$ até que o tamanho total requerido para aplicação fosse atingido, onde g corresponde à granularidade da aplicação e h o fator de heterogeneidade.

5.1.3 Disponibilidade da informação

No que diz respeito à disponibilidade das informações sobre os recursos, foi considerado que cada *site* ou possui todas ou não possui alguma informação sobre seus recursos. A disponibilidade de informação sobre os recursos foi classificada em cinco níveis, considerando o percentual de *sites* que possuem informação disponível sobre os recursos. Os níveis usados foram: 10%, 25%, 50%, 75% e 100%.

Da mesma forma, para informação sobre os *links*, tanto *links* que conectam os sites (*interlinks*), quanto pra *links* locais (*intralinks*), foram considerados cinco níveis de disponibilidade da informação: 10%, 25%, 50%, 75% e 100%. É importante mencionar que o nível de disponibilidade de informação sobre os *links* não influi no nível de disponibilidade de

informação sobre os recursos. Devido a isso, é possível ter um cenário em que o escalonador não tem informação alguma sobre os recursos de um *site*, mas tem informação sobre os *links* (*interlink* e *intralink*) daquele *site*.

Para a disponibilidade da informação da aplicação, as simulações foram feitas da seguinte forma: a heurística *Adaptive Data-Intensive* foi simulada duas vezes para cada cenário, uma com informação sobre a aplicação e outra sem informação; a heurística *Adaptive WQR* foi simulada com informação sobre a aplicação; a heurística *Storage Affinity* foi simulada apenas sem informação sobre a aplicação, já que ela não necessita desse tipo de informação; a heurística *XSufferage* foi simulada somente com informação sobre a aplicação, pois ela necessita de informação completa sobre a aplicação e o ambiente de execução.

5.1.4 Grau de replicação máximo

Com o objetivo de testar o desperdício de recursos das heurísticas de escalonamento baseadas em replicação, foram feitas simulações variando o grau de replicação máximo das heurísticas durante o processo de escalonamento. Esse grau de replicação máximo corresponde ao número máximo de réplicas que uma tarefa pode ter durante o processo de escalonamento.

Como mostrado nos trabalhos de Paranhos et. al. [6] e Santos-Neto et. al. [8], quanto maior o grau de replicação máximo, maior será o desperdício de recursos. Por outro lado, quando maior o grau de replicação máximo, menor pode ser o *makespan*.

Nos cenários simulados neste trabalho, os valores para o grau de replicação máximo utilizados foram: 3, 5 e infinito. O valor infinito para o grau de replicação máximo significa que a heurística não tem um limite para o número de réplicas de uma tarefa, ou seja, o número de réplicas que uma tarefa possui não influencia na decisão da heurística de escalonamento se ela deve replicar uma tarefa ou não.

5.1.5 Tipos de aplicações *data-intensive*

As aplicações *data-intensive* podem ser classificadas em dois tipos diferentes. O primeiro inclui as aplicações cujas tarefas possuem o custo computacional proporcional ao tamanho dos seus dados de entrada. Estas serão chamadas de *proporcional* neste trabalho. Um tipo comum de aplicação *data-intensive* que se encaixa neste tipo é a contagem do número de

ocorrências de um determinado padrão em um arquivo de entrada, onde todo o arquivo de entrada precisa ser processado. Por outro lado, existem aplicações cujas tarefas não possuem o custo computacional proporcional ao tamanho dos seus dados de entrada. Este tipo de aplicação será chamado de *não-proporcional* neste trabalho. Um exemplo para este tipo de aplicação é a busca por padrões em um arquivo, onde o padrão procurado pode estar tanto no início, meio ou fim do arquivo.

Como será mostrado a seguir, a heurística *Adaptive Data-Intensive* consegue fazer um escalonamento eficiente com aplicações do tipo *proporcional* mesmo sem ter informação sobre o custo das tarefas da aplicação. Isto acontece porque a heurística pode utilizar o tamanho dos dados de entrada da tarefa no lugar do seu custo computacional, já que eles são proporcionais. Para aplicações do tipo *não-proporcional*, a heurística *Adaptive Data-Intensive* perde eficiência se a informação sobre o custo computacional das tarefas não está disponível.

5.1.6 Repetição da execução

Uma característica comum em aplicações *data-intensive* é a necessidade de se repetir a execução da aplicação mais de uma vez utilizando os mesmos dados de entrada. Isto acontece comumente em aplicações de renderização de imagens, onde os dados que descrevem os cenários permanecem os mesmo, mas podem ser renderizados em várias perspectivas[1]. Outro exemplo bastante comum é o de aplicações para previsão do tempo, em que os dados podem ser processados por modelos atmosféricos diferentes[2]. Até nas simulações deste trabalho é usada a repetição de execução, onde os *traces* que descrevem as entidades de um cenário de simulação (recursos, *links*) podem permanecer os mesmos na execução de muitos cenários distintos.

Devido a isso, em cada cenário simulado foi feita a repetição da execução quatro vezes para cada heurística. A diferença entre a primeira execução e as demais é que a partir da segunda execução, os dados de entrada das tarefas já estão armazenados nos *sites* em que elas executaram. As heurísticas que são otimizadas para aplicações *data-intensive* devem lidar com este tipo de cenário realizando um escalonamento eficiente visando o mínimo de

[1] Blender - http://www.blender.org/

[2] BRAMS - http://www.cptec.inpe.br/brams/

transferência de dados possível.

5.2 Validade dos resultados

Para que os resultados das simulações ficassem estatísticamente válidos, foi preciso simular pouco mais de 10000 cenários distintos para as heurísticas *Adaptive Data-Intensive* e *Adaptive WQR*, pouco mais de 2500 para a heurística *XSufferage* e pouco mais de 2000 para a heurística *Storage Affinity*.

A explicação para as diferentes quantidades de cenários simulados para algumas heurísticas se dá pelo fato de que as heurísticas *Adaptive Data-Intensive* e *Adaptive WQR* tiveram cenários variando tanto na disponibilidade da informação (10%, 25%, 50%, 75% e 100% de disponibilidade da informação) quanto no grau de replicação máximo (3, 5 e infinito). A heurística *XSufferage* variou apenas a quantidade de informação disponível, porque ela não utiliza o mecanismo de replicação. Por fim, a heurística *Storage Affinity* variou apenas no grau de replicação máximo, porque ela não utiliza informação sobre o ambiente e a aplicação.

As médias dos resultados obtidos para as duas métricas usadas têm um erro máximo de 1% para mais ou para menos, com um nível de confiança de 95% [39].

5.3 Resultados

Nesta seção, serão mostrados os resultados das simulações feitas com o objetivo de confrontar a heurística *Adaptive Data-Intensive* com as heurísticas do estado da arte *Adaptive WQR*, *Storage Affinity* e *XSufferage*. Devido ao dinamismo e heterogeneidade de um *grid* computacional, apenas parte das informações podem estar disponíveis para um escalonador. Com isso, as simulações tiveram como foco principal o impacto que a disponibilidade de informação causa nas heurísticas de escalonamento. Além disso, foi avaliado o impacto no desperdício de recursos causados pelas heurísticas baseadas em replicação. Por fim, foi avaliado o comportamento das heurísticas em cenários onde o custo computacional das tarefas não é proporcional ao tamanho dos seus dados de entrada.

5.3.1 Impacto da disponibilidade da informação no *makespan*

Nesta seção serão mostrados os resultados para o *makespan* separados pelos cenários que variam na quantidade de informação disponível. A ideia é mostrar que a heurística *Adaptive Data-Intensive* consegue diminuir o *makespan* da aplicação à medida que a quantidade de informação disponível aumenta. Em todos os resultados das simulações desta seção, foi utilizado o valor 3 para o grau de replicação máximo das heurísticas baseadas em replicação. Além disso, o tipo de aplicação utilizada nos cenários mostrados nesta seção é *proporcional*.

A Tabela 5.2 mostra os resultados do *makespan* com 10% de informação disponível. As colunas da tabela correspondem aos *makespans* obtidos da primeira até a quinta execução da aplicação.

De acordo com os resultados mostrados para o *makespan* nos cenários com 10% de informação disponível, é observado que a heurística *XSufferage* obteve um valor muito alto do *makespan* na primeira execução em relação às outras heurísticas. Isto ocorre porque a heurística não está apta para utilizar a parte do *grid* que não possui informação, neste caso, a heurística apenas utilizou 10% dos recursos do *grid*. A heurística *Adaptive Data-Intensive* teve o melhor resultado para o *makespan* porque ela utiliza informação sobre os *links*, podendo ela escolher os melhores *links* para as maiores tarefas. O resultado do *makespan* da primeira execução da heurística *Storage Affinity* foi pior que os das heurísticas *Adaptive Data-Intensive* e *Adaptive WQR* porque a heurística *Storage Affinity* não utiliza informação sobre a aplicação, enquanto que as outras utilizam. Os resultados do *makespan* a partir da segunda execução foram menores para as heurísticas *Adaptive Data-Intensive*, *Storage Affinity* e *XSufferage* porque elas são otimizadas para aplicações *data-intensive*, desta forma, os dados não precisaram ser novamente transferidos nas execuções a partir da segunda. Salienta-se que o valor para a heurística *XSufferage* foi maior porque ela só está utilizando 10% dos recursos do *grid*.

A Tabela 5.3 mostra os resultados para o *makespan* obtidos nos cenários com 25% de informação disponível. Segundo os resultados do *makespan* para os cenários com 25% de informação disponível, é observado que a heurística *XSufferage* melhorou seu *makespan* em todas execuções em relação aos *makespans* dos cenários com 10% de informação disponível. Isto ocorre porque nestes cenários a heurística utilizou 25% dos recursos do *grid*, uma parcela maior que no cenário anterior. É possível notar também que houve um pequeno

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive	2987s	303s	306s	311s	307s
Adaptive WQR	3234s	3245s	3225s	3213s	3221s
Storage Affinity	3493s	302s	302s	303s	301s
XSufferage	13327s	1593s	1595s	1600s	1587s

Tabela 5.2: Resultados do *makespan* para os cenários com 10% de informação disponível

melhoramento do *makespan* nas heurísticas *Adaptive Data-Intensive* e *Adaptive WQR*, isto ocorre pelo fato de que essas heurísticas tem o *makespan* afetado de acordo com a disponibilidade da informação. Os resultados para a heurística *Storage Affinity* são os mesmos porque ela não é afetada pela disponibilidade da informação.

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive	2830s	301s	300s	302s	301s
Adaptive WQR	3188s	3203s	3210s	3204s	3195s
Storage Affinity	3493s	302s	302s	303s	301s
XSufferage	9524s	830s	823s	827s	819s

Tabela 5.3: Resultados do *makespan* para os cenários com 25% de informação disponível

As Tabelas 5.4, 5.5 e 5.6 mostram os resultados do *makespan* para os cenários com 50%, 75% e 100% de disponibilidade da informação, respectivamente.

De acordo com os resultados do *makespan*, é possível observar que a heurística *XSufferage* melhora seu *makespan* à medida que o nível de disponibilidade da informação aumenta. Da mesma forma para as heurísticas *Adaptive Data-Intensive* e *Adaptive WQR*, onde o resultado do *makespan* delas é influenciado pelo nível de disponibilidade da informação, porém bem menos influenciada porque ambas utilizam os recursos sem informação para realizar o escalonamento.

Segundo os resultados dos cenários com 100% de informação disponível mostrados na Tabela 5.6, o valor do *makespan* para a heurística *XSufferage* ainda foi pior que o da heurística *Adaptive Data-Intensive*. Este resultado ocorreu pelo fato de que a heurística *Adaptive Data-Intensive* foi simulada com grau de replicação máximo de 3, que influencia diretamente

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive	2802s	310s	304s	301s	309s
Adaptive WQR	3153s	3151s	3130s	3164s	3147s
Storage Affinity	3493s	302s	302s	303s	301s
XSufferage	5934s	615s	619s	621s	607s

Tabela 5.4: Resultados do *makespan* para os cenários com 50% de informação disponível

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive	2779s	304s	307s	311s	306s
Adaptive WQR	3148s	3160s	3163s	3149s	3157s
Storage Affinity	3493s	302s	302s	303s	301s
XSufferage	3877s	501s	509s	507s	499s

Tabela 5.5: Resultados do *makespan* para os cenários com 75% de informação disponível

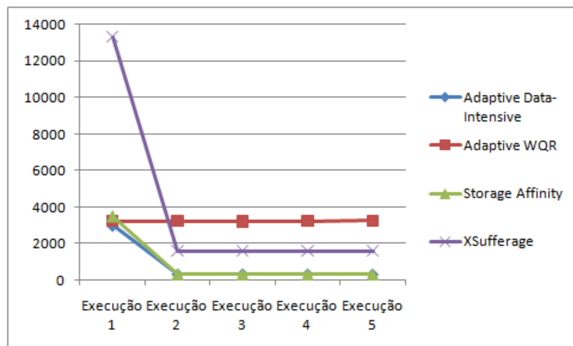
no seu *makespan*.

Para confirmar este comportamento, foram realizadas simulações com cenários onde as heurísticas não utilizaram o mecanismo de replicação, ou seja, elas tinham grau de replicação máximo de 1. Além disso, a disponibilidade de informação utilizada foi de 100%. A Tabela 5.7 mostra os resultados obtidos das simulações nesses cenários. Com isso, é possível observar que a não utilização de replicação para as heurísticas baseadas em replicação, a heurística *XSufferage* tem um *makespan* melhor para todas as execuções.

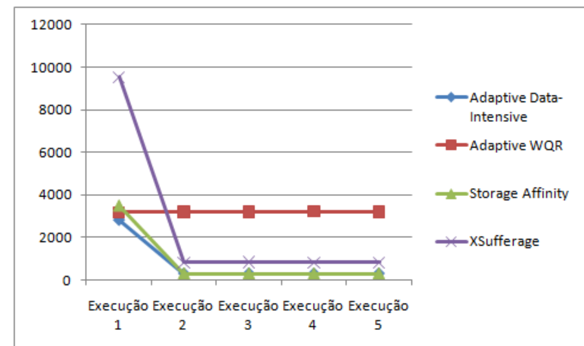
As Figuras 5.1(a) a 5.1(e) ilustram os resultados do *makespan* em relação à disponibilidade da informação, enquanto que a Figura 5.1(f) ilustra os resultados do *makespan* com cenários sem replicação e com 100% de disponibilidade da informação.

5.3.2 Impacto da disponibilidade da informação no desperdício de recursos

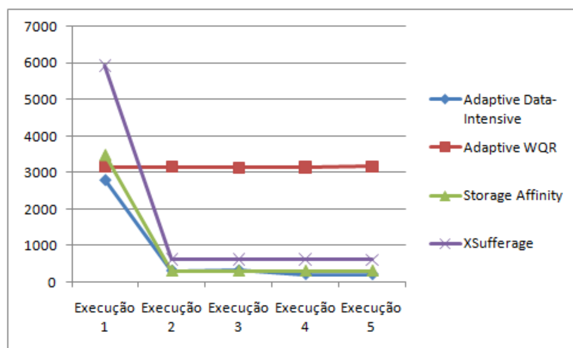
Nesta seção serão mostrados os resultados para o desperdício de recursos separados pelos cenários que variam na quantidade de informação disponível. A ideia é mostrar que a heu-



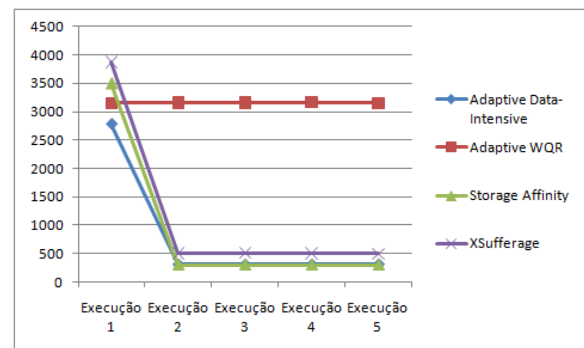
(a) Média do *makespan* para os cenários com 10% de disponibilidade da informação



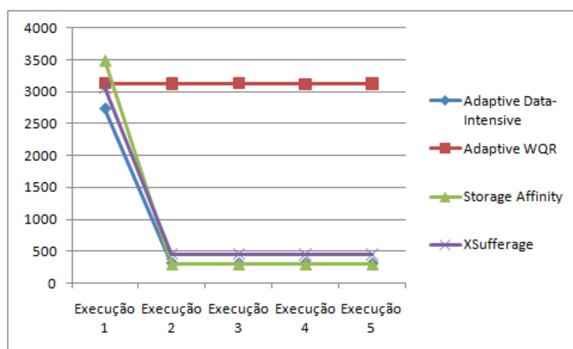
(b) Média do *makespan* para os cenários com para 25% de disponibilidade da informação



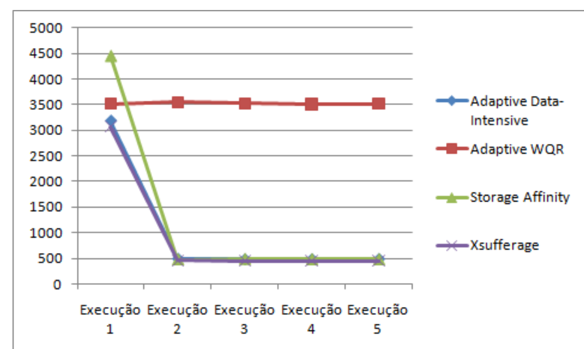
(c) Média do *makespan* para os cenários com para 50% de disponibilidade da informação



(d) Média do *makespan* para os cenários com 75% de disponibilidade da informação



(e) Média do *makespan* para os cenários com 100% de disponibilidade da informação



(f) Média do *makespan* para os cenários sem replicação e com 100% de disponibilidade da informação

Figura 5.1: Média do *makespan* em relação à disponibilidade da informação

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive	2731s	305s	303s	307s	302s
Adaptive WQR	3133s	3124s	3130s	3121s	3124s
Storage Affinity	3493s	302s	302s	303s	301s
XSufferage	3071s	453s	457s	448s	450s

Tabela 5.6: Resultados do *makespan* para os cenários com 100% de informação disponível

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive	3186s	493s	481s	484s	489s
Adaptive WQR	3513s	3547s	3523s	3506s	3514s
Storage Affinity	4460s	475s	484s	480s	479s
XSufferage	3071s	457s	451s	453s	453s

Tabela 5.7: Resultados do *makespan* para os cenários sem replicação e 100% de informação disponível

rística *Adaptive Data-Intensive* consegue diminuir o desperdício de recursos à medida que a quantidade de informação disponível aumenta. Em todos os resultados das simulações desta seção foi utilizado o valor 3 para o grau de replicação máxima das heurísticas baseadas em replicação. Como a heurística *XSufferage* não utiliza replicação, não será avaliado o seu desperdício. Além disso, a aplicação utilizada nos cenários mostrados nessa subseção é do tipo *proporcional*.

As Figuras 5.2(a) a 5.2(e) ilustram o comportamento do desperdício em relação aos cenários com 10%, 25%, 50%, 75% e 100% de informação disponível.

No que se refere ao desperdício de recursos, como esperado, o desperdício de recursos da heurística *Storage Affinity* foi invariante à disponibilidade da informação, já que ela apenas utiliza o grau de replicação máximo e a disponibilidade dos dados de entrada da tarefa no *site* para efetuar a replicação. Como dito anteriormente, a heurística *Storage Affinity* tem a restrição de replicação onde ela só replica uma tarefa em um recurso que pertença a um *site* que já possua armazenados os dados de entrada daquela tarefa. Por outro lado, as heurísticas *Adaptive Data-Intensive* e *Adaptive WQR* obtiveram um desperdício de recursos que

foi inversamente proporcional ao nível de disponibilidade da informação. Como esperado, isto ocorreu porque estas heurísticas replicam melhor à medida que há mais informação disponível. O desperdício de recursos da heurística *Adaptive Data-Intensive* foi melhor que o da heurística *Adaptive WQR* porque a heurística *Adaptive Data-Intensive* também utiliza a restrição para a replicação em que o *site* deve ter os dados de entrada das tarefas a serem replicadas já armazenados nos seus repositórios de dados.

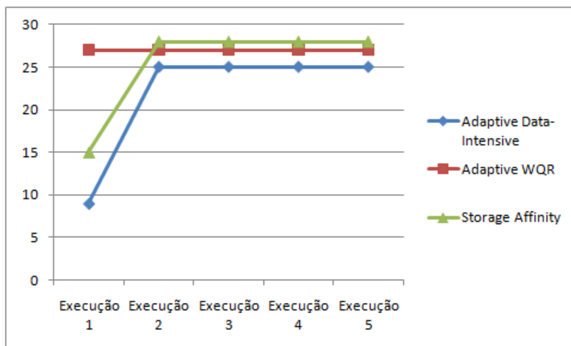
5.3.3 Impacto do grau de replicação máximo no desperdício de recursos

Nesta seção serão mostrados os resultados para o desperdício de recursos separados pelos cenários que variam no valor do grau de replicação máximo. A ideia é mostrar que a heurística *Adaptive Data-Intensive* consegue ter um reduzido desperdício de recursos mesmo que o grau de replicação máximo aumente. Em todos os resultados das simulações desta seção foi utilizado 100% de disponibilidade da informação para as heurísticas que utilizam informação. Além disso, a aplicação utilizada nos cenários mostrados nessa subseção é do tipo *proporcional*.

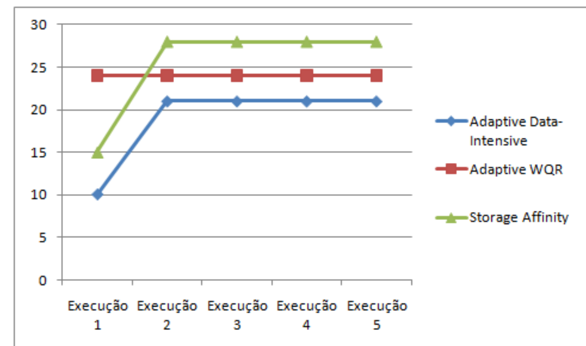
As Figuras 5.3(a), 5.3(b) e 5.3(c) ilustram o comportamento do desperdício em relação aos cenários com grau de replicação máximo de 3, 5 e infinito, respectivamente. De acordo com os gráfico, é possível observar que apesar do grau de replicação máxima aumentar, as heurísticas *Adaptive Data-Intensive* e *Adaptive WQR* conseguem manter um reduzido nível de desperdício de recursos devido às restrições impostas por elas de acordo com as informações sobre os recursos. A heurística *Adaptive Data-Intensive* tem o desperdício melhor porque ela ainda restringe o *site* ao qual será replicada a tarefa. Já a heurística *Storage Affinity* não utiliza informação sobre o ambiente de execução para restringir a replicação de tarefas, por isso o seu desperdício foi impactado pelo grau de replicação máximo.

5.3.4 Aplicações do tipo *não-proporcional*

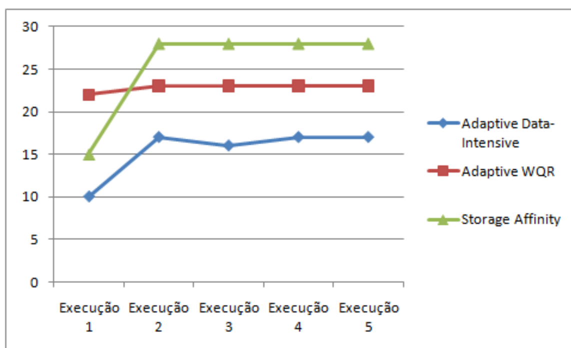
Nesta seção serão mostrados os resultados das simulações feitas com aplicações do tipo *não-proporcional*, que são as aplicações que possuem tarefas com custo computacional não proporcional ao tamanho dos dados de entrada. Para estas simulações, foi utilizado grau de



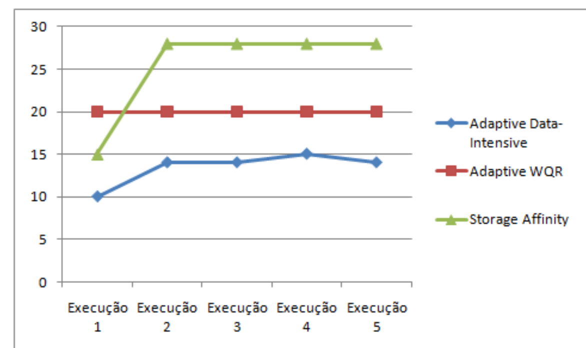
(a) Média do desperdício de recursos para 10% de disponibilidade da informação



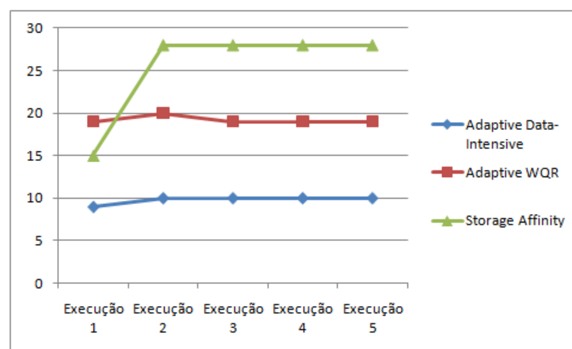
(b) Média do desperdício de recursos para 25% de disponibilidade da informação



(c) Média do desperdício de recursos para 50% de disponibilidade da informação

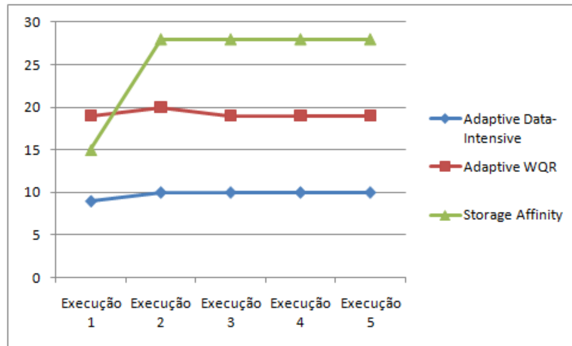


(d) Média do desperdício de recursos para 75% de disponibilidade da informação

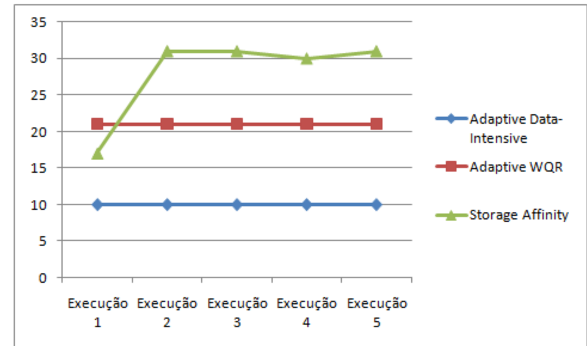


(e) Média do desperdício de recursos para 100% de disponibilidade da informação

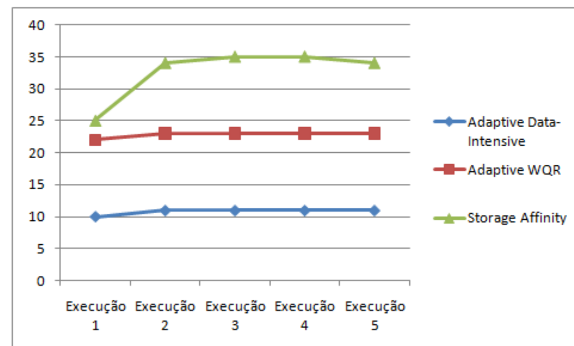
Figura 5.2: Média do desperdício de recursos em relação à disponibilidade da informação



(a) Média do desperdício de recursos para 100% de disponibilidade da informação e grau de replicação 3



(b) Média do desperdício de recursos para 100% de disponibilidade da informação e grau de replicação 5



(c) Média do desperdício de recursos para 100% de disponibilidade da informação e grau de replicação infinito

Figura 5.3: Média do desperdício de recursos em relação ao grau de replicação máximo

replicação máximo de 3, com os cenários variando na disponibilidade da informação. As heurísticas confrontadas nestes cenários foram: *Adaptive Data-Intensive* com disponibilidade de informação da aplicação e sem disponibilidade de informação da aplicação, *Adaptive WQR* e *XSufferage*. O objetivo é mostrar que a heurística *Adaptive Data-Intensive* tem um bom *makespan* em aplicações do tipo *não-proporcional*.

As Tabelas 5.8 e 5.9 mostram os resultados para o *makespan* nos cenários com disponibilidade da informação de 50% e 100%.

Com base nos dados apresentados nas Tabelas 5.8 e 5.9, é possível observar que a heurística *Adaptive Data-Intensive* consegue manter um bom *makespan* mesmo escalonando aplicações do tipo *não-proporcional*, até mesmo sem informação sobre a aplicação que teve resultados um pouco piores em relação aos resultados que utilizaram informação sobre a aplicação. A explicação para isso é que apesar da heurística não ter informação sobre a aplicação, ela ainda tem informação sobre os *links* o que a faz escolher os melhores *links* para as maiores tarefas.

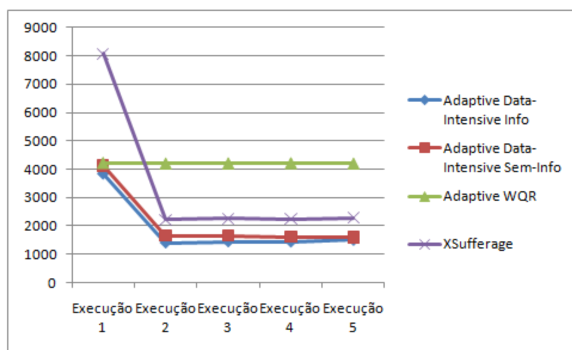
As Figuras 5.4(a) e 5.4(b) ilustram os resultados do *makespan* em relação à disponibilidade da informação nos cenários utilizando aplicações do tipo *não-proporcional*.

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive Info	3847s	1400s	1444s	1436s	1493s
Adaptive Data-Intensive Sem-Info	4130s	1647s	1653s	1605s	1601s
Adaptive WQR	4220s	4205s	4209s	4201s	4196s
XSufferage	8077s	2214s	2253s	2237s	2282s

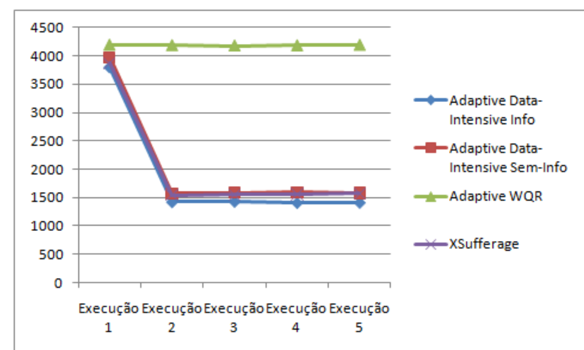
Tabela 5.8: Resultados do *makespan* para os cenários com 50% de informação disponível utilizando aplicações do tipo *não-proporcional*

Heurística ↓ / Execução →	**1**	**2**	**3**	**4**	**5**
Adaptive Data-Intensive Info	3794s	1422s	1429s	1407s	1412s
Adaptive Data-Intensive Sem-Info	3970s	1569s	1584s	1590s	1577s
Adaptive WQR	4193s	4185s	4179s	4183s	4192s
XSufferage	3864s	1543s	1552s	1557s	1569s

Tabela 5.9: Resultados do *makespan* para os cenários com 100% de informação disponível utilizando aplicações do tipo *não-proporcional*



(a) Média do *makespan* para cenários com 50% de informação disponível que utilizam aplicações do tipo *não-proporcional*



(b) Média do *makespan* para cenários com 100% de informação disponível que utilizam aplicações do tipo *não-proporcional*

Figura 5.4: Média do *makespan* para cenários que utilizam aplicações do tipo *não-proporcional*

Capítulo 6

Conclusão

Heurísticas de escalonamento para aplicações *BoT* podem ser consideradas de dois tipos: *bin-packing* e replicação. Heurísticas *bin-packing* são aquelas que utilizam toda informação do ambiente de execução e da aplicação para realizar um escalonamento eficiente, enquanto que as heurísticas baseadas em replicação necessariamente não utilizam informação alguma sobre o ambiente de execução e a aplicação, porém utilizam o mecanismo de replicação de tarefas ao final da execução visando diminuir o *makespan*.

Este trabalho apresentou uma heurística de escalonamento chamada *Adaptive Data-Intensive*. Ela é uma heurística para aplicações *data-intensive* que é adaptativa à disponibilidade da informação sobre o ambiente de execução e a aplicação. A heurística *Adaptive Data-Intensive* tem o seu processo de escalonamento baseado no da heurística *Storage Affinity*, onde o escalonamento é feito de acordo com uma métrica de afinidade entre as tarefas e os *sites* do *grid*. Porém, ao contrário da heurística *Storage Affinity*, a heurística *Adaptive Data-Intensive* utiliza qualquer informação que esteja disponível sobre o ambiente de execução e a aplicação para otimizar o processo de escalonamento visando reduzir o *makespan* e o desperdício de recursos. O desperdício de recursos acontece devido à replicação de tarefas realizada por heurísticas baseadas em replicação.

O desempenho da heurística proposta neste trabalho foi comparado com o desempenho das heurísticas *Adaptive WQR*, *Storage Affinity* e *XSufferage*. As duas primeiras são heurísticas baseadas em replicação, sendo a segunda voltada para aplicações *data-intensive*, enquanto que a terceira é uma heurística do tipo *bin-packing* para aplicações *data-intensive*.

Os resultados apresentados no Capítulo 5 mostraram que a heurística *Adaptive Data-*

Intensive obteve valores para o *makespan*, em média, melhores para a primeira execução, quando os dados de entrada das tarefas precisam ser transferidos para os *sites*. Para as demais execuções, o *makespan* da heurística *Adaptive Data-Intensive* foi equivalente aos *makespans* das heurísticas *Storage Affinity* e *XSufferage*. Em relação aos resultados obtidos para o nível de desperdício de recursos, a heurística *Adaptive Data-Intensive* obteve resultados melhores à medida que a disponibilidade da informação aumentava. Isto mostra que quanto maior a disponibilidade da informação, menor o desperdício de recursos.

Por fim, este trabalho mostrou que é possível diminuir o tempo de execução e o desperdício de recursos das aplicações *BoT data-intensive* utilizando uma heurística de escalonamento que é adaptativa à informação sobre o ambiente de execução e a aplicação em um *grid* computacional, ambiente esse que devido a sua dinamicidade e heterogeneidade pode ter apenas parte dele com informação disponível.

Bibliografia

[1] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício Silva, Carla Osthoff, and Cirano Silveira. Running Bag-of-Tasks Applications on Computational Grids: The Mygrid Approach. In *Proceedings of ICCP'2003 - International Conference on Parallel Processing*, October 2003.

[2] Francisco Brasileiro, Eliane Araújo, William Voorsluys, Milena Oliveira, and Flávio Figueiredo. Bridging the high performance computing gap: the ourgrid experience. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid/First Latin American Grid Workshop*, pages 817–822, Washington, DC, USA, 2007. IEEE Computer Society.

[3] D. Menascé, D. Saha, S. Porto, and G. Hoaks. Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures. *Journal of Parallel and Distributed Computing*, pages 1–18, 1995.

[4] P. Francis, S. Jamin, V. Paxson, and et al. An Architecture for Global Internet Host Distance Estimation Service. In *Proceedings of IEEE INFOCOM*, 1999.

[5] A. Baratloo, M. Karaul, Z.Kedem, and P. Wycko. Charlotte: Metacomputing on the Web. In *Proocedings of the 9th Conference on Parallel and distributed Computing Systems*, 1996.

[6] Daniel Paranhos, Walfredo Cirne, and Francisco Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing*, pages 169–180, August 2003.

[7] Walfredo Cirne, Francisco Brasileiro, Daniel Paranhos, Luís Fabrício W. Góes, and William Voorsluys. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing*, 33(3):213–234, 2007.

[8] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. In *Proceedings of 10th Job Scheduling Strategies for Parallel Processing (JSSPP'2004)*, June 2004.

[9] Nelson Nobrega-Junior, Leonardo Assis, and Francisco Brasileiro. Scheduling cpu-intensive grid applications using partial information. In *The 37th International Conference On Parallel Processing (ICPP-08)*, September 2008.

[10] Matthew Massie, Brent Chun, and David Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Journal of Parallel Computing*, 30(7), July 2004.

[11] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee. S^3: A scalable sensing service for monitoring large networked systems. In *Proceedings of SIGCOMM'06 Workshops*, September 2006.

[12] F. Brasileiro, L. Costa, A. Andrade, W. Cirne, S. Basu, and S. Banerjee. A large scale fault-tolerant grid information service. In *Proceedings of 4th International Workshop on Middleware for Grid Computing*, November 2006.

[13] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceeding of the 10th IEEE Symposium on High-Performance Distributing Computing*, August 2001.

[14] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice hall, New Jersey, USA, 2nd edition, August 2001.

[15] L. Assis, N. Nobrega-Junior, F. Brasileiro, and W. Cirne. Uma heuristica de particionamento de carga divisível para grids computacionais. In *Anais do* $24^{\underline{o}}$ *Simposio Brasileiro de Redes de Computadores (SBRC'2006)*, Curitiba - PR - Brasil, Maio 2006.

[16] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop*, May 2000.

[17] Tracy Braun, Howard Siegel, and Noah Beck. A Comparion of Eleven Static Heuristics for Mapping Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.

[18] Ming Wu and Xian-He Sun. A General Self-adaptive Task Scheduling System for Non-dedicated Heterogeneous Computing. In *Proceedings of IEEE Cluster Computing Conference*, Hong Kong, December 2003.

[19] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *Proceedings of 12th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'2004)*, pages 364–371, February 2004.

[20] H. Casanova, J. Hayes, and Y. Yang. Algorithms and Software to Schedule and Deploy Independent Tasks in Grid Environments. In *Workshop on Distributed Computing, Metacomputing and Resource Globalization*, France, December 2002.

[21] H. James, K. Hawick, and P. Coddington. Scheduling Independente Tasks on Metacomputing Systems. Technical report, University of Adelaide, The University of Adelaide. DHCP-066, 1999.

[22] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *Proceedings of the 9th Heterogeneous Computing Workshop*, May 2000.

[23] Y. Yang and H. Casanova. Umr: A multi-round algorithms for scheduling divisible. In *Internacional Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[24] Y. Yang and H. Casanova. Rumr: Robust scheduling for divisible workloads. In *12th IEEE Symposium on High-Performance Distributed Computing (HPDC-12)*, June 2003.

[25] Nguyen The Loc, Said Elnaffar, Takuya Katayama, and Ho Tu Bao. A scheduling method for divisible workload problem in grid environments. In *Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT05)*, 2005.

[26] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. In *Journal of the ACM (JACM)*, volume 24, pages 280–289, 1977.

[27] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load sharing in heterogeneous systems via weighted factoring. In *Proceedings from 8'th Symposium on Parallel Algorithms and Architectures*, pages 318–328, 1996.

[28] Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton. A grid service broker for scheduling e-science applications on global data grids: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(6):685–699, 2006.

[29] David Abramson, Jon Giddy, and Lew Kotler. High performance parametric modeling with nimrod/g: Killer application for the global grid? In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2000. IEEE Computer Society.

[30] D. P. Anderson. Local scheduling for volunteer computing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[31] D. Menascé, D. Saha, and S. Porto et. al. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. In *Journal of Parallel and Distributed Computing*, pages 1–18, 1995.

[32] Nelson Nobrega-Junior. Avalliação de heurísticas de escalonamento de aplicações bag-of-tasks em grids computacionais adaptativas à disponibilidade de informação, 2006.

[33] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[34] Nazareno Andrade, Francisco Brasileiro, Walfredo Cirne, and Miranda Mowbray. Automatic grid assembly by promoting collaboration in peer-to-peer grids. *Journal of Parallel and Distributed Computing*, 67(8):957–966, 2007.

[35] Rich Wolski. Experiences with Predicting Resource Performance On-line in Computational Grids. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 41–49, March 2003.

[36] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf, 2004.

[37] Arnaud Legrand, L. Marchal, and Henri Casanova. Scheduling Distributed Applications: the Simgrid Simulation Framework. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID03)*, Tokyo, Japan, May 2003.

[38] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 7:341–370, September 1985.

[39] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/-Cole Publishing Company, 1999.