Uma heurística de particionamento de carga divisível para grids computacionais

Leonardo de Assis, Nelson Nóbrega-Júnior, Francisco Brasileiro, Walfredo Cirne

Laboratório de Sistemas Distribuídos – Departamento de Sistemas e Computação – Universidade Federal de Campina Grande (UFCG)
Caixa Postal 10.106 – 58.109-970 – Campina Grande – PB – Brasil

{leoassis,nelson,fubica,walfredo}@dsc.ufcg.edu.br

Abstract. *Many applications that process a large amount of data can easily be transformed into parallel applications composed of independent tasks, i.e., Bag-of-Task applications (BoT). For this, it is enough to divide the data to be processed in small portions and to associate each portion to a task that can independently process it. Such applications are known as divisible workload applications. Although it is easy to transform them in BoT, it is not trivil to identify what is the best way to partition the application, so that a good performance is achieved. Previously known partitioning and scheduling heuristics assume either a homogeneous execution environment or the availability of information on the application profile and the load to which the execution infrastructure is subject. These are assumptions that are not always possible to be made. In particular, computational grids comprise very heterogeneous infrastructures, and the absence of a centralized control makes it very difficult to obtain complete and accurate information about system load. This paper presents a divisible workload heuristic that dynamically calculates the ideal application granularity without requiring any external information about the execution environment, being therefore applicable in computational grid environments.*

Resumo. *Muitas aplicações que processam uma grande quantidade de dados podem ser facilmente transformadas em aplicações paralelas compostas por tarefas independentes, i.e., aplicações Bag-of-Task (BoT). Para isso, basta particionar os dados a serem processados em várias porções menores e associar cada porção a uma tarefa, que pode ser processada independentemente das outras. Tais aplicações são conhecidas como aplicações que possuem carga divisível. Entretanto, embora o particionamento seja simples, identificar qual o particionarmento mais adequado para uma determinada carga não é uma tarefa trivial. As heurísticas de particionamento e escalonamento conhecidas ou assumem um ambiente de execução homogêneo ou a existência de informações completas e precisas sobre o perfil da aplicação e a carga à qual a infra-estrutura de execução está sujeita. Essas são hipóteses que nem sempre podem ser assumidas. Em particular, grids computacionais são ambientes heterogêneos, em que a dinamicidade e o controle distribuído dificultam a obtenção das informações requeridas por essas heurísticas. Este artigo apresenta uma heurística de particionamento de carga divisível que calcula dinamicamente a granularidade ideal da carga da aplicação sem necessitar de qualquer informação externa sobre o ambiente de execução, sendo portanto aplicáveis a ambientes de grids computacionais.*

1. Introdução

Aplicações Bag-of-Task (BoT) são aplicações paralelas que possuem tarefas independentes. Este tipo de aplicação pode ser facilmente adaptável para rodar em uma

plataforma de grid computacional, bastando apenas alocar as tarefas da aplicação às diferentes máquinas do grid. Isso acontece devido ao fato dessas aplicações não necessitarem de comunicação entre elas, e dessa forma não sofrem maiores impactos por conta de uma possível alta latência de comunicação entre os recursos de um grid.

Dentre as aplicações BoT, existem aquelas que processam uma grande quantidade de dados (renderização de imagens, busca por padrões, mineração de dados, etc.). Algumas dessas aplicações são bastante flexíveis, podendo ter os seus dados particionados de qualquer forma e sem grandes restrições ao tamanho dessas partições. Tais aplicações são conhecidas como aplicações com *carga* divisível (*divisible workloads*). Entretanto, embora o particionamento seja simples, definir qual o particionamento mais adequado, de forma que a aplicação tenha um bom desempenho, não é uma tarefa trivial.

Em um ambiente de computação paralela (supercomputador, *cluster*, grid, etc.), a associação de tarefas a recursos é geralmente feita por um software denominado *escalonador*, através do qual o usuário submete sua aplicação para execução. O escalonador é responsável por selecionar os recursos disponíveis mais apropriados para a aplicação e submeter suas tarefas para execução. Um escalonador utiliza-se de heurísticas para realizar as decisões de escalonamento. No caso de aplicações com carga divisível, antes de processar o escalonamento é preciso também decidir como se dará o particionamento da aplicação em tarefas.

Ao longo dos anos foram propostas várias heurísticas que tentam resolver o problema de calcular o tamanho ideal das tarefas para subseqüentemente escaloná-las. Entretanto, a maior parte dessas heurísticas usam informações sobre a plataforma de execução, como por exemplo *Weighted Factoring* [4], *UMR* [5] e *RUMR* [6]. Outras heurísticas, como *GSS* [2] e *Factoring* [3], assumem que os recursos da plataforma de execução são homogêneos. Em ambos os casos, as hipóteses são difíceis de serem garantidas em um grid computacional. Grids computacionais são formados por recursos que pertencem a domínios administrativos diferentes, e que portanto, estão normalmente sujeitos a políticas diferentes. Não é factível assumir que todos os domínios que oferecem recursos para o grid possam disponibilizar informação sobre a sua infra-estrutura. Além disso, grids são ambientes altamente heterogêneos, que evoluem de forma extremamente dinâmica. Dessa forma, as heurísticas existentes não se aplicam à execução eficiente de aplicações de carga divisível em grids computacionais.

Neste artigo apresentamos uma heurística de particionamento de aplicações com carga divisível que consegue obter um bom desempenho sem necessitar de informações externas precisas sobre o ambiente de execução. A heurística calcula a granularidade ideal da aplicação, isto é, o tamanho ideal dos dados a serem processados por cada tarefa (tamanho da tarefa), de forma a minimizar o tempo total de execução da aplicação (*makespan* [11]). Ela baseia-se no tempo de execução das tarefas anteriores para definir o tamanho da próxima tarefa a ser escalonada.

Com o intuito de avaliarmos o nosso trabalho, realizamos experimentos com uma aplicação real em um grid. Comparamos os tempos de execução da aplicação com e sem o nosso serviço. Os resultados dos experimentos iniciais mostram que a utilização do serviço proposto pode reduzir o *makespan* das aplicações em até 83%. Além disso, realizamos ainda simulações com o intuito de fazer uma avaliação de desempenho entre nossa heurística e a heurística RUMR [6], que representa o estado-da-arte das heurísticas de particionamento de aplicação com a carga divisível.

O restante deste artigo é organizado da seguinte forma. Na Seção 2 são apresentados alguns trabalhos relacionados. A heurística de particionamento proposta é apresentada na Seção 3. Na Seção 4 descrevemos a implementação de nossa heurística como uma funcionalidade adicional do *broker* MyGrid [1]. Os resultados dos experimentos e simulações são apresentados e analisados na Seção 5. A Seção 6 conclui o artigo.

2. Trabalhos Relacionados

Heurísticas para escalonamento de aplicações com carga divisível não são uma novidade. Ao longo dos anos foram propostos vários trabalhos nesta área. A seguir, discutiremos algumas heurísticas relacionadas ao nosso trabalho.

As heurísticas *GSS* [2], *Factoring* [3] e *Weighted Factoring* [4] iniciam o processo de escalonamento com tarefas de tamanho grande. A granularidade da tarefa vai diminuindo ao longo do processo a fim de obter um bom balanceamento de carga e reduzir as incertezas dos tempos de execução das tarefas. A alocação de uma tarefa grande quando o *pipeline* está prestes a começar a esvaziar pode desperdiçar um percentual grande de recursos. Embora tenham um bom desempenho em ambientes de execução homogêneos e fortemente acoplados, como *clusters*, quando essas heurísticas são aplicadas a ambientes fracamente acoplados como um grid, o escalonamento de tarefas muito grandes no início do processo faz com que as máquinas do grid fiquem ociosas por um longo tempo enquanto estão sendo feitas as transferências das primeiras tarefas, impactando o *makespan* da aplicação.

Objetivando diminuir o problema apontado acima, a heurística UMR [5] inicia o escalonamento com tarefas de tamanho pequeno, aumentando ao longo do processo a granularidade da tarefa para obter um *pipeline* de comunicação e computação. Entretanto, como discutido anteriormente, as máquinas também poderão ficar ociosas, desta vez no final do escalonamento, por causa das últimas tarefas com tamanho grande.

O trabalho feito em *A Scheduling Method for Divisible Workload Problem in Grid Environments* [15] apresenta uma heurística de escalonamento de cargas divisíveis em grids computacionais. Essa heurística é baseada em [5], mas com um método de estimativa de quanto poder de processamento cada *worker* irá doar para a aplicação do grid, este método é baseado nos trabalhos feitos em [16,17]. Da mesma forma que em [5], esta heurística necessita de informações do ambiente computacional (velocidades dos *links* e máquinas) que, em um ambiente como grid computacionais, são difíceis de obter com precisão. Nossa proposta elimina essa necessidade.

A heurística RUMR [6] une características dos dois tipos de heurísticas citadas acima com o objetivo de corrigir suas limitações. Nesta heurística, tarefas com tamanho pequeno são escalonadas no início, com a granularidade da aplicação aumentando no decorrer do processo. Entretanto, a partir de um certo ponto, as tarefas vão diminuindo de tamanho até que o escalonamento seja concluído.

A heurística proposta nesse trabalho é inspirada na heurística RUMR, no sentido que ela inicialmente utiliza uma granularidade baixa para particionar a carga da aplicação, aumentando a granularidade paulatinamente e finalizando também com uma granularidade baixa para obter um maior paralelismo. Porém, diferente do RUMR que se baseia em informações precisas sobre o grid (velocidade das máquinas e velocidade dos links) para particionar a carga, nossa heurística baseia-se apenas no tempo de execução das tarefas já executadas.

3. Uma nova heurística de particionamento automatizado

O objetivo principal da heurística consiste em reduzir o *makespan* da aplicação, aumentando o paralelismo da mesma. Considerando-se T_{1st_free} como o momento em que a primeira máquina do grid fica ociosa (sem tarefas para processar) e T_{Final} como sendo o momento em que a execução da aplicação termina, temos que para se atingir um maior paralelismo de uma dada aplicação é necessário ter T_{1st_free} tão tarde quanto possível, isto é, nós temos que reduzir a diferença entre T_{Final} e T_{1st_free} (ver Figura 1). Em um cenário ótimo teríamos $T_{1st_free} = T_{Final}$, e a utilização máxima de paralelismo no grid.

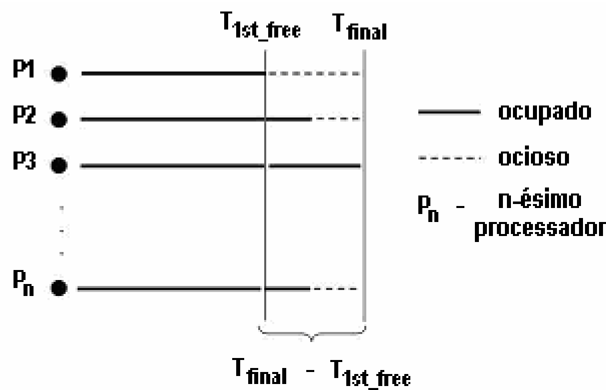

Figura 1. Relação entre T_{Final} e T_{1st_free}

O objetivo de diminuir toda a diferença entre T_{Final} e T_{1st_free} consiste em fazer com que os recursos disponíveis sejam utilizados da melhor maneira possível durante a execução da aplicação de forma que o seu *makespan* seja reduzido. Em um cenário ótimo teríamos $T_{1st_free} = T_{Final}$, e a utilização máxima de paralelismo no grid.

Para ajustar a granularidade da aplicação de maneira que a diferença (T_{Final} - T_{1st_free}) seja reduzida, a heurística segue os mesmos princípios da heurística RUMR. No início da execução a granularidade das tarefas é pequena, de tal forma a ocupar rapidamente todos os recursos disponíveis, aumenta durante a execução e sendo reduzida quando o final da execução se aproxima, para se obter um maior paralelismo. Diferentemente do RUMR, a heurística proposta usa uma métrica de desempenho, calculada durante a execução da aplicação, para definir a granularidade usada em cada momento da execução.

A métrica de desempenho usada é a vazão média do grid, em termos de bytes processados por unidade de tempo, dada pela média harmônica das velocidades de cada máquina do grid em um passado próximo (**$G_{throughput}$**). Mais formalmente, essa métrica é definida da seguinte forma:

$$G_{\text{throughput}} = \frac{n}{\sum_{1}^{n} \frac{1}{P_{\text{throughput,i}}}}$$

onde, **n** é o número de máquinas do grid e **$P_{throughput,\,i}$** é a vazão da máquina **i,** que é calculada através da média das velocidades desta máquina em relação à granularidade atual das tarefas, podendo ser definida mais formalmente como:

$$P_{\text{throughput},i} = Ga \sum_{1}^{min(w,ki)} \frac{1}{T_j}$$

onde, **Ga** é o tamanho das tarefas atuais, **w** é um valor previamente fixado que limita o número de tarefas anteriores consideradas para o cálculo, $\mathbf{k_i}$ é o número de tarefas com granularidade **Ga** terminadas pela máquina **i**, e $\mathbf{T_j}$ é o tempo gasto, incluindo tempo de transferência, pela máquina **i** para terminar a tarefa **j** com granularidade **Ga**.

A heurística proposta consiste de três fases: *Inicial*, *Steady-State* e *Final*. A fase *Inicial* corresponde ao momento em que a aplicação é submetida. Como ainda não há informação coletada, uma percentagem pré-definida da carga é processada com uma certa granularidade pequena, de tal forma a preencher o *pipeline* rapidamente. Durante esta fase, a vazão das máquinas começa a ser constantemente monitorada até o fim da execução da aplicação. Todas as informações coletadas serão utilizadas pela fase *Steady-State*.

Durante a fase *Steady-State*, a granularidade da aplicação é ajustada com base nas informações coletadas sobre as velocidades das máquinas. Esta fase tem o intuito de aumentar o valor de $\mathbf{G_{throughput}}$. Se o valor anterior de $\mathbf{G_{throughput}}$ é menor que o atual (considerando-se uma percentagem mínima de variação para indicar o aumento ou diminuição da vazão), a granularidade da aplicação é aumentada. Caso contrário, a granularidade é reduzida. Quando a granularidade da aplicação é reduzida, são escalonadas tarefas com três tamanhos diferentes para cada máquina, tarefas com tamanho menor, maior e igual à granularidade atual. Isto acontece devido a maior parte da execução da aplicação ocorrer durante a fase *Steady-State*, podendo então o grid, neste período, sofrer modificações que impactam na sua vazão. Desta forma, escalonando tarefas com tamanhos maiores e menores em relação à granularidade atual, é possível detectar mudanças no grid mais eficientemente.

Na fase *Final*, quando a aplicação já processou uma quantidade considerável de carga (previamente fixada), a sua granularidade é reduzida para evitar que uma associação de tarefas grandes para máquina lentas no final da computação aumente o intervalo de tempo durante o qual o grid tem máquinas ociosas e possa dessa forma comprometer o *makespan* da aplicação como um todo. Se pouco antes de entrar na fase *Final,* a granularidade atual for menor que a granularidade final (também previamente fixada), então a granularidade atual é adotada na fase *Final*.

A vazão do grid é calculada a partir do momento em que todas as máquinas possuem pelo menos uma velocidade calculada para a granularidade atual, ou o número de tarefas processadas com a granularidade atual seja igual ao número de máquinas do grid multiplicado por uma constante. Neste último caso, é assumido que as máquinas que ainda não terminaram nenhuma computação, terminam neste exato momento.

Em nossos experimentos arbitramos o valor 4 para a constante citada no parágrafo anterior. A explicação para essa escolha é baseada na lei de Moore, que diz que as velocidades das máquinas duplicam em um período de um ano e meio. Levando em consideração que o tempo de vida útil de uma máquina seja de quatro a cinco anos, um grid heterogêneo pode ter uma máquina quatro a cinco anos mais antiga que outra, ou seja, a máquina mais nova pode ser aproximadamente oito vezes mais rápida que a máquina mais antiga. Então, calculando-se a média das velocidades relativas das máquinas, o resultado é aproximadamente quatro.

Em resumo, a heurística utiliza quatro parâmetros: **Gi**, **Gf**, **V** e **W**, os quais são descritos na Tabela 1.

Tabela 1. Parâmetros da Heurística

Nome	*Descrição*
G_i	Granularidade utilizada na fase *Inicial*.
G_f	Granularidade adotada durante a fase *Final*. Normalmente deve ser utilizada uma granularidade um pouco maior que G_i.
V	Percentagem de variação máxima, para mais ou para menos, admissível entre o valor anterior e o valor atual de $G_{throughput}$ sem que seja caracterizado, respectivamente, um aumento ou diminuição da mesma
W	Tamanho da janela que delimita o número de vazões consideradas para o cálculo de $P_{throughput}$ para cada máquina

A quantidade da carga processada nas fases *Inicial* e *Final* é calculada de acordo com as seguintes fórmulas:

$WL_i = G_i \cdot 4 \cdot n$, $WL_f = G_f \cdot 4 \cdot n$, onde n é o número de máquinas disponíveis.

Isto indica que é preciso ter processado pelo menos um número de tarefas equivalente a quatro vezes o número de máquinas para se calcular a vazão do grid, como explicado anteriormente.

4. Detalhes de implementação

Para validar a heurística, implementamos um serviço que automatiza o processo de particionamento de aplicações com carga divisível. Tal serviço foi implementado como uma funcionalidade adicional do MyGrid [1]. O MyGrid é um *broker* que suporta a execução de aplicações BoT em grids computacionais, mais especificamente, no OurGrid [13]. Entre outras funcionalidades, o MyGrid implementa vários escalonadores de tarefas. Por sua vez, o OurGrid consiste em um grid aberto para execução de aplicações BoT, que está em operação desde dezembro de 2004 (veja http://status.ourgrid.org/ para ter uma visão do estado do sistema em tempo-real). Ele provê uma infra-estrutura *peer-to-peer* que permite o compartilhamento de recursos computacionais em escala mundial.

Na Seção 4.1, apresentamos uma visão geral do MyGrid. Na Seção 4.2 descrevemos a arquitetura geral do serviço de particionamento.

4.1. Visão geral do MyGrid

No contexto do MyGrid, uma aplicação consiste em um conjunto de tarefas independentes, que podem ser executadas em qualquer máquina disponível no grid. O MyGrid provê abstrações que escondem do usuário a heterogeneidade das máquinas que compõem o grid. Para o MyGrid, existem dois tipos de máquinas: *home machine* e *grid machine*. A *home machine* é o componente central do MyGrid, atuando como um

coordenador que escalona e gerencia a execução das tarefas nas *grid machines* (máquinas do grid responsáveis pela execução das tarefas). A partir da *home machine*, o usuário do MyGrid submete sua aplicação, cujas tarefas executarão nas *grid machines* (*GuMs*). A Figura 2 ilustra a organização dos componentes do MyGrid.

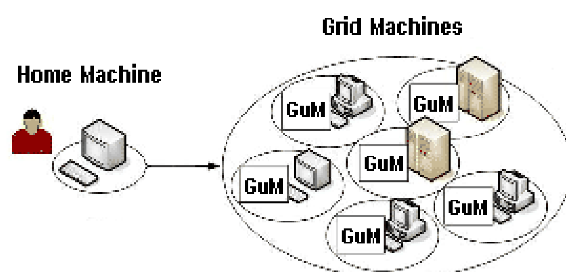

Figura 2. Organização dos componentes do MyGrid

Na versão atual (3.2.1), a arquitetura do MyGrid possui dois módulos: *Scheduler* e *Replica Executor* [7]. O *Scheduler* é responsável por escalonar tarefas para execução em uma *grid machine*. As *grid machines* são fornecidas pelo *OurGrid peer*, este tem a função de localizar e enviar *grid machines* para o *Scheduler*. O *Replica Executor* é responsável por executar uma tarefa em uma *grid machine*. A Figura 3 mostra o projeto arquitetural do MyGrid [7].

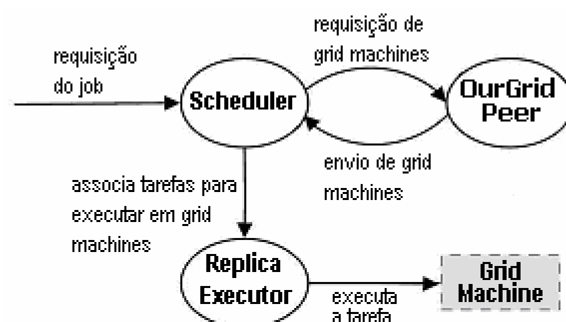

Figura 3. Projeto arquitetural do MyGrid

O *Scheduler* aguarda requisições de submissão de *jobs*. Quando ele recebe uma requisição, ele aplica uma heurística de escalonamento para verificar quantas máquinas são necessárias para executar o *job*. Após isso, o *Scheduler* requisita as *grid machines* necessárias ao *OurGrid peer*. Por fim, o *Scheduler* envia pares de tarefas e *grid machines* para o *Replica Executor*. Este instancia uma réplica da tarefa e a submete à *grid machine* correspondente (tarefas são todas tratadas como réplicas). Uma réplica é uma tentativa de executar uma tarefa.

Atualmente, o MyGrid possui duas heurísticas de escalonamento, *Workqueue With Replication* (WQR)[8] e *Storage Affinity* [9]. WQR é uma extensão do clássico algoritmo *Workqueue*. Essa heurística seleciona tarefas do *job* e as associa às máquinas de maneira aleatória. Quando *grid machines* estão ociosas e todas as tarefas já foram associadas, mais réplicas de tarefas que estão executando são instanciadas e associadas a estas máquinas. As outras réplicas da mesma tarefa são abortadas assim que alguma

réplica termina. Desta maneira, réplicas associadas posteriormente podem finalizar antes das associadas anteriormente, aumentando o desempenho da aplicação. Por outro lado, essa replicação ocasiona desperdício de recursos (*badput*).

A heurística *Storage Affinity* foi proposta com o intuito de tornar mais eficiente o escalonamento de aplicações BoT que processam grandes quantidades de dados. Ela associa tarefas às máquinas de acordo com a afinidade entre elas. A afinidade entre uma tarefa e uma máquina corresponde ao número de *bytes* dos dados a serem processados pela tarefa que já estão armazenados no *site* ao qual a máquina pertence. A heurística calcula a afinidade de cada tarefa com todas as máquinas, escalonando a tarefa com o maior valor de afinidade na máquina na qual esse valor foi obtido. Esse processo é repetido de forma iterativa até que todas as tarefas sejam escalonadas. Assim como *WQR*, *Storage Affinity* também aplica a técnica de replicação para lidar com a falta de informação sobre o grid. Entretanto, *Storage Affinity* segue critérios específicos para criação de uma réplica, diferente de *WQR* que replica tarefas de maneira aleatória.

4.2. Arquitetura do particionador

Atualmente, o usuário do MyGrid tem que particionar a carga da aplicação manualmente. Neste caso, o arquivo de configuração do *job* precisa conter a descrição detalhada de todas as tarefas indicando, entre outras informações, a porção da *carga* a ser processada por cada uma delas.

Utilizando o serviço proposto, o processo de particionar a carga da aplicação é realizado de maneira automática. Dessa forma, o usuário apenas precisa especificar no seu arquivo de configuração do *job* qual a carga a ser particionada. Com base nisso, o serviço particiona a carga e cria as tarefas responsáveis por processar cada porção. Em seguida, as tarefas são submetidas para execução.

O serviço de particionamento foi implementado como um módulo adicional na arquitetura do MyGrid, o qual interage apenas com o módulo *Scheduler* (ver Figura 4). Dessa forma, diferentemente da abordagem normalmente discutida na literatura, em que particionamento e escalonamento são executados de forma integrada, o serviço proposto nesse artigo é independente da heurística de escalonamento utilizada pelo escalonador do MyGrid (*WQR*, *Storage Affinity* ou uma nova que possa vir a ser proposta e implementada), definida pelo usuário.

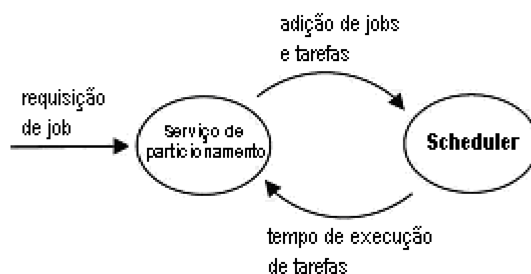

Figura 4. Interação entre o serviço de particionamento e o *Scheduler*

O serviço de particionamento monitora o *job* durante toda sua execução. À medida que as tarefas finalizam sua execução, o *Scheduler* retorna o tempo de execução das mesmas ao serviço. Com isso, o serviço pode calcular o $\mathbf{G}_{\mathbf{Throughput}}$ e realizar as decisões necessárias (ver Seção 3). O objetivo principal do serviço proposto é otimizar o valor de $\mathbf{G}_{\mathbf{Throughput}}$ de forma que possa reduzir o *makespan* da aplicação.

5. Avaliação de Desempenho

Nesta seção, apresentamos resultados de simulações onde comparamos nossa heurística (referenciada por *Dinâmica* no texto) com RUMR, que representa o estado da arte em heurísticas de particionamento e com a heurística WQR sem o uso do particionador (Seção 5.1). Apresentamos ainda, os resultados obtidos nos experimentos realizados para validar a eficiência do nosso serviço de particionamento (Seção 5.2). É importante notar que a nossa heurística faz apenas o particionamento, usando o escalonador WQR para escalonar as tarefas (ver Seção 4).

5.1 Simulações

Para comparar a nossa heurística com a RUMR e WQR, implementamos um simulador baseado no framework de simulação SimGrid [12]. Executamos um total de 12.000 simulações. Uma vez que a execução dessa quantidade de simulações é uma aplicação BoT, nós utilizamos o OurGrid para executá-las em tempo hábil.

Comparamos as heurísticas através dos seus respectivos *makespans* em um conjunto total de 2.000 cenários distintos. Os cenários foram gerados de forma aleatória, variando a velocidade das máquinas do grid. Para simular a heterogeneidade e dinamicidade do grid, as cargas das máquinas e da rede variavam ao longo do tempo durante a execução de um cenário. Ainda mais, em alguns cenários estas informações podiam ser obtidas com um certo valor de erro quando medidas pela heurística. Os parâmetros e seus respectivos valores estão expressos na Tabela 2. Esses valores foram escolhidos baseados em experimentos prévios realizados e em aplicações BoT reais (ver Seção 5.2).

Tabela 2. Parâmetros usados para a construção dos cenários das simulações

Parâmetro	***Valores***
Carga	Fixado em 1.3GB
Número de processadores	Fixado em 10
Velocidade relativa dos processadores (RUMR)	U (0.5 e 2.0)
Carga dos processadores (RUMR)	Variável entre 0% e 100%
Velocidade da rede (RUMR)	Aprox. 90 Mbps
Erro na informação sobre o ambiente (RUMR)	Até 30%
Parâmetros do particionador (Dinâmico)	G_I = 256KB, G_F = 2.3MB, V = 0.02 e W = 5.

A variação de carga no grid foi simulada através de arquivos de *traces* obtidos de medições em um ambiente real utilizando a ferramenta NWS [14]. Tais *traces* contêm a informação de carga do recurso em função do tempo.

Para cada cenário gerado, simulamos a nossa heurística, a heurística RUMR na ausência e presença de erros na informação e a heurística WQR com três granularidades

diferentes (menor, média e maior alcançada na simulação da nossa heurística). No caso da heurística WQR a aplicação foi particionada em tarefas de tamanho igual, dado pelo valor da granularidade usado. Com isso, para cada cenário, executamos um total de seis simulações distintas. Note que a presença de erros na informação no processo de escalonamento, da nossa heurística e da heurística WQR, não influencia no desempenho das mesmas, já que elas não necessitam de tais informações.

Fizemos uma análise estatística para calcular o número de simulações necessárias para que os nossos resultados obtivessem 95% de confiança com um erro de apenas 2 segundos para mais ou para menos. A nossa amostra de 2.000 simulações, para cada caso, já tinha o número de simulações necessárias. O caso que necessitava de mais simulações foi o WQR com a maior granularidade, com um total de 1.164 simulações necessárias.

A média do *makespan* das simulações de cada heurística, em cada caso, estão expressos na Tabela 3.

Tabela 3. *Makespan* (em segundos) das simulações

Dinâmico	RUMR (sem erro)	RUMR (com erro)	WQR (menor)	WQR (média)	WQR (maior)
386	426	427	521	530	549

Os resultados da Tabela 3 mostram que, a nossa heurística obteve uma média do *makespan* melhor do que a média das outras heurísticas. Os resultados da heurística RUMR são bem parecidos devido ao erro máximo da informação ser de até 30%. A heurística RUMR não tem perda de desempenho significativo com erros na informação de até 30%.

5.2. Experimentos

Com o intuito de avaliar o nosso trabalho, realizamos experimentos no MyGrid, comparando o desempenho da aplicação com a utilização e sem a utilização do serviço de particionamento. A heurística do escalonador do MyGrid adotada foi WQR. Como o RUMR necessita de informação, não foi possível executá-la no ambiente de grid no qual realizamos os experimentos. De fato, nossa motivação para propor uma heurística de particionamento para grids vem justamente do fato das heurísticas existentes não serem facilmente adaptáveis a esses ambientes e as heurísticas de escalonamento de grids não tratarem de particionamento de aplicações.

Como aplicação exemplo, utilizamos *Lame* [10] que é uma aplicação que realiza a conversão de arquivos no formato *WAV* para arquivos no formato *MP3*. O tamanho da carga utilizada foi de 1.3GB e consistia de arquivos de áudio no formato WAV.

Os parâmetros do serviço de particionamento foram configurados da mesma forma que nos cenários de simulação anteriormente apresentados, ou seja: $\mathbf{G_I}$=256KB, $\mathbf{G_F}$=2.3MB, **V**=0.02 e **W**=5. Como mencionado anteriormente, esses valores foram obtidos a partir de simulações realizadas com o objetivo de encontrar valores adequados para esses parâmetros. Utilizamos o simulador descrito na seção anterior para encontrar os melhores valores para os parâmetros da heurística. As simulações foram feitas da seguinte forma: fixávamos três parâmetros e fazíamos uma varredura no parâmetro

restante com o objetivo de achar o melhor valor. A cada vez que calibrávamos um parâmetro, refazíamos a varredura em todos os outros que já tinham sido calibrados. Dessa forma, verificávamos se a escolha de um parâmetro afetaria outro parâmetro já calibrado anteriormente.

Analisamos um cenário com dez máquinas utilizando a heurística de escalonamento WQR com nível de replicação 3. As máquinas utilizadas nos experimentos eram de dois tipos diferentes, sete máquinas *Pentium* 4 de 1.8GHz com 640MB de memória e três máquinas *Pentium* 4 de 2.8GHz com 1GB de memória, todas estavam em um mesmo *site*. Entretanto, como a carga sob a qual o grid estava sujeito variava ao longo do tempo, executamos cada cenário quatro vezes, de forma intercalada, no intuito de amenizar a variação do ambiente de execução, e obtendo uma média do *makespan* da aplicação com um desvio padrão mínimo.

Para cada cenário, realizamos um experimento com o uso do serviço de particionamento e três experimentos sem o uso do serviço. Nesses últimos, utilizamos um valor diferente para a granularidade da aplicação. Os valores utilizados foram baseados nas seguintes granularidades alcançadas no experimento com o serviço: granularidade inicial, granularidade média e maior granularidade atingida.

A Tabela 4 apresenta a média dos resultados obtidos nos experimentos.

Tabela 4. Média do *Makespan* (em segs) nos experimentos

Dinâmico	Estático (Gi)	Estático (Média)	Estática (Maior)
477	2822	543	486

Os resultados alcançados mostram que a média do *makespan* da aplicação no experimento com o serviço (Dinâmico) foi aproximadamente 2% menor que o melhor caso e 83% menor que o pior caso do *makespan* nos experimentos sem o serviço (Estático). O pior caso aconteceu quando foi utilizada a menor granularidade.

No caso da menor granularidade, ocorre o pior caso devido ao grande número de tarefas (de tamanho pequeno) que, apesar de permitir um maior paralelismo, ocasionam uma sobrecarga no escalonador. Isso explica a diferença muito grande no resultado das simulações comparado ao resultado dos experimentos, uma vez que o simulador, como é tradicionalmente feito na literatura da área, não incorpora essa sobrecarga em seu modelo.

A pequena diferença entre o resultado da nossa heurística e o resultado do particionamento estático com maior granularidade ocorreu pelo fato de que esta granularidade foi a que obteve uma melhor vazão durante o processo de escalonamento e por isso obteve um bom desempenho no experimento com o particionamento estático. É importante entender que esse valor só foi descoberto após executarmos primeiro o nosso experimento, para depois definir quais os valores usar nos três casos do WQR. Na prática, o usuário teria que definir sem muita informação, qual o tamanho do grão a usar dentro de um universo de escolha bastante amplo.

Além do *makespan*, analisamos também a métrica *badput* que corresponde à soma de todos os tempos de execução das réplicas abortadas. Dessa forma, tal métrica indica o

grau de desperdício de recursos do grid[1]. A Tabela 5 mostra a média do *badput* resultante nos experimentos.

Tabela 5. Média do *BadPut* (em segundos) nos experimentos

Dinâmico	Estático (Gi)	Estático (Média)	Estática (Maior)
20	14	31	74

Como era de se esperar, os resultados apresentados na Tabela 5 indicam que o *badput* é diretamente proporcional à granularidade da aplicação. Quanto maior a granularidade, menos tarefas serão necessárias para executar a aplicação e, assim, uma maior proporção da aplicação será executada de forma replicada (vale lembrar que a fase de replicação só começa quando não há mais tarefas que não tiveram pelo menos uma réplica escalonada).

Dessa forma, como o experimento utilizando o serviço apresenta granularidade variável ao longo da execução da aplicação, nossa heurística apresenta a vantagem extra de combinar o bom desempenho da granularidade alta, com o baixo *badput* da granularidade pequena.

6. Conclusão

Neste trabalho apresentamos uma heurística de particionamento de carga divisível para grids computacionais que consegue obter um bom desempenho sem necessitar de informações precisas sobre o ambiente de execução. O objetivo do serviço proposto é calcular a granularidade ideal (tamanho da tarefa) de forma a minimizar o *makespan* da aplicação. A heurística adotada baseia-se no tempo de execução das tarefas anteriores para definir o tamanho da próxima tarefa a ser escalonada.

A heurística foi implementada como um módulo adicional ao *broker* MyGrid. Para avaliar o nosso trabalho, realizamos experimentos utilizando uma aplicação real em um grid computacional, onde comparamos os tempos de execução da aplicação com e sem o nosso serviço. Os resultados dos experimentos iniciais mostraram que a utilização do serviço proposto reduziu o *makespan* em até 83%. Além disso, vale ressaltar que o nível de desperdício de recursos do grid nos experimentos usando o serviço foi comparável aos menores valores dos experimentos sem a utilização do serviço.

Realizamos ainda, através de simulação, uma comparação entre a nossa heurística com as heurísticas RUMR e WQR, considerando cenários onde há informação perfeita e cenários onde a informação é imprecisa. Os resultados obtidos através das simulações mostraram que a nossa heurística teve uma média do *makespan* melhor do que a média do RUMR em ambos os cenários (sem erro e com erro na informação). Enquanto nos cenários utilizando a heurística WQR com particionamento estático, a nossa heurística se saiu bem melhor. Dessa forma, concluímos que nossa heurística se adeqüa melhor a ambientes dinâmicos e heterogêneos, como grids computacionais, do que heurísticas

[1] Como mencionado anteriormente, a nossa heurística usa o escalonador WQR para escalonar as tarefas depois de particionadas. Dessa forma, as execuções com a heurística Dinâmica também fazem uso de replicação.

que necessitam de informações precisas para manter um bom desempenho e se adeqüa melhor do que heurísticas de particionamento estático.

Agradecimentos Os autores gostariam de agradecer o apoio financeiro do CNPq/Brasil (processo 300.646/96). Este trabalho foi desenvolvido em colaboração com a HP Brasil P&D.

Referências

[1] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. Silva, C. Osthoff, C. Silveira, *Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approuch,* Proceedings of the ICCP'2003: Internacional Conference on Parallel Processing, outubro 2003.

[2] C. Polychronopoulos and D. Kuck, *Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers*. IEEE Trans. on Computers C-36(12) pp. 1425-1439, dez. 1987.

[3] S. Flynn Hummel. *Factoring: a Method for Scheduling Parallel Loops*. Communications of the ACM, 35(8):90-101, agosto 1992.

[4] 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.

[5] Y. Yang and H. Casanova. *UMR: A Multi-Round Algorithms for Scheduling Divisible Workloads*. In Proceedings of the Internacional Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, abril 2003.

[6] Y. Yang and H. Casanova. RUMR: Robust Scheduling for Divisible Workloads. In *Proceedings of the 12th IEEE* Symposium on High-Performance Distributed Computing *(HPDC-12)*, junho 2003.

[7] OurGrid 3.1 User Manual, 6th maio 2005.

[8] D. Paranhos, W. Cirne, F. Brasileiro. *Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids.* Europar'2003.

[9] E. Santos-Neto, W. Cirne, F. Brasileiro, A. Lima. *Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids.* 10th JSSPP, junho 2004.

[10] Lame. http://lame.sourceforge.net

[11] M. Pinedo, *Scheduling Theory, Algorithms and Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1995

[12] H. Casanova, *Simgrid: A toolkit for the simulation of application scheduling*, in *Proceedings of the First IEEE/ACM Internacional Symposium on Cluster Computing and the Grid*, (CCGrid 2001).

[13] OurGrid website. http://www.ourgrid.org

[14] R. Wolski, *Dynamically Forecasting Network Performance Using the Network Weather Service*. Cluster Computing, 1(1): 119-132, 1998.

[15] Nguyen The Loc, Said Elnaffar, Takuya Katayama, Ho Tu Bao, *A Scheduling Method for Divisible Workload Problem in Grid Environments*. Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'05).

[16] L. Yang, J. Schopf, and I. Foster. *Conservative scheduling: Using predicted variance to improve scheduling decision in dynamic environments*. Super-Computing 2003, Phoenix, Arizona USA, November 2003.

[17] L. Yang, J. Schopf, and I. Foster. *Homeostatic and tendency-based cpu load predictions*. International Parallel and Distributed Processing Symposium (IPDPS'03) Nice, Francec, April 2003.