Estudo comparativo de escalonadores de tarefas para grades computacionais

Alvaro Henry Mamani Aliaga

Texto para Qualifição do Mestrado apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo

Programa: Ciências da Computação

Orientador: Prof. Dr. Alfredo Goldman

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro do CNPq, processo número 133147/2009-6

São Paulo, Novembro de 2010

Estudo comparativo de escalonadores de tarefas para grades computacionais

Esta é a versão submetida ao exame de qualificação
de Alvaro Henry Mamani Aliaga

Comissão Julgadora:

- Prof. Dr. Alfredo Goldman (orientador) - IME-USP.
- Prof. Dr. Marcos Gubitoso - IME-USP.
- Prof. Dra. Liria Sato - Poli-USP.

Resumo

Atualmente existem várias grades computacionais em funcionamento baseadas em diferentes *middlewares* para o gerenciamento de recursos. Cada um deles apresenta estratégias específicas para o escalonamento de aplicações nos recursos disponíveis. O escalonamento pode considerar desde um ambiente mais simples, com recursos localizados em uma única rede local, no caso um aglomerado (*cluster*). Mas, ambientes mais complexos podem ser considerados. Dois tipos de contextos estão cada vez mais comuns no escalonamento de aplicações em grades, o escalonamento inter-aglomerado e o escalonamento oportunista. No caso inter-aglomerado, permite-se que as aplicações sejam alocadas em ambientes compostos por diferentes domínios, possivelmente permitindo sua execução simultânea em mais de um domínio. No escalonamento em ambiente oportunista utilizam-se recursos não dedicados para executar aplicações. Em ambientes com essas características, o escalonamento precisa ser dinâmico e adaptativo, isto é, os recursos devem ser alocados no momento da criação das tarefas, possibilitando que somente os recursos mais adequados no momento sejam escolhidos. Nessas grades, esses recursos ficam espalhados em diversos domínios administrativos locais, sendo compartilhados por usuários locais que devem ter prioridade sobre o uso dos mesmos. Neste trabalho, estudamos diferentes escalonadores atualmente utilizados. Em seguida implementamos os escalonadores em um simulador de grades, assim elaboramos uma análise comparativa dos mesmos em diferentes cenários, fazendo uma comparação detalhada dos diversos algoritmos de escalonamento. Para o futuro estudaremos ambientes compostos e escalonamento oportunista.

Palavras-chave: Computação em grade, algoritmos de escalonamento, escalonamento de tarefas, workflows.

Abstract

Currently there are several grids in operation based on different middlewares for resource management. Each one has specific strategies for the scheduling of applications on available resources. The scheduling can be considered from a simple environment, with resources located on a single LAN, e.g. a cluster. But, more complex environments can be considered. Two kind of contexts are more common in the scheduling of grid applications, the inter-cluster scheduling and opportunistic scheduling. The inter-cluster scheduling allows applications to be placed in composed environments of different domains, possibly allowing the spontaneous execution in more than one domain. In opportunistic scheduling environment is common the use of non-dedicated resources to execute applications. In environments with these characteristics, the scheduling must be dynamic and adaptive, that means, resources should be allocated at the moment of creation of tasks, allowing the choice of the most appropriate resources. In these grids, the resources are scattered in various local administrative areas, being shared by local users who should have priority over the usage. In this work, we study different schedulers currently used. Then we will implement them in a grid simulator, as well as develop a comparative analysis of them in different scenaries, making a detailed comparison of several scheduling algorithms. For the future we will study compounds environments and opportunistic scheduling.

Keywords: Grid computing, scheduling algorithms, task scheduling, workflows.

Sumário

Lista de Algoritmos

Lista de Figuras

Capítulo 1

Introdução

Nos anos recentes, tem aumentado a disponibilidade de computadores poderosos assim como a sua interligação com redes de alta velocidade. Este fato tem permitido a agregação de recursos geograficamente dispersos para execução de tarefas de aplicações de grande escala e com uso intensivo de recursos. Esta agregação de recursos tem sido chamada de Computação em Grade (*Grid Computing*) [FKNT02], uma alternativa para obter grande capacidade de processamento.

Grades computacionais compreendem uma complexa infra-estrutura composta por soluções integradas de hardware e software que permitem o compartilhamento de recursos distribuídos sob a responsabilidade de instituições distintas [FK04]. Esses ambientes são alternativas atraentes para a execução de aplicações paralelas ou distribuídas que demandam alto poder computacional, tais como mineração de dados, previsão do tempo, biologia computacional, física de partículas, processamento de imagens médicas, entre outras [BFH03]. Essas aplicações são composta por diversas tarefas que, a depender do tipo de aplicação, podem se comunicar durante a fase de execução.

Na computação em grade, os recursos computacionais são heterogêneos e podem ser agregados ou retirados do ambiente em qualquer momento. Neste cenário o "escalonamento de tarefas" é um grande desafio, que tem como objetivo principal atingir um bom desempenho no tempo de execução de aplicações, independentemente do tipo destas. Assim, o escalonamento de tarefas é um problema importante a ser estudado no âmbito das grades computacionais. O uso de um algoritmo eficiente para a gestão destes recursos é uma questão crucial.

Escalonamento é um problema antigo que motiva muitas pesquisas em diversas áreas [BM06, THW02, dSCB03], nas quais é muito importante determinar o tempo de início das tarefas e a localização do recurso, e outras informações importantes. Um dos objetivos principais nos algoritmos de escalonamento é minimizar o tempo de término das aplicações (*makespan*), escalonando seus componentes de forma a maximizar o paralelismo na execução das tarefas e minimizar a comunicação, conseqüentemente otimizando a utilização dos recursos. No escalonamento de tarefas uma aplicação (ou processo) pode ser composta de tarefas que têm dependências de dados, onde a execução de cada tarefa deve respeitar as suas tarefas precedentes, os custos de comunicação entre tarefas e custos de computação das tarefas componentes do processo [BM06]. No caso de tarefas independentes, as tarefas não possuem dependência, como por exemplo, aplicações do tipo *Bag-of-Tasks*.

1.1 Motivação

O escalonamento (*scheduling* em inglês) de tarefas é um problema NP-Completo [Pin08], mas no ambiente da computação em grade o problema de escalonamento se torna ainda mais desafiador devido às características da grade: dinamicidade e heterogeneidade, recursos fisicamente distantes uns dos outros, etc.

Para que a computação em grade possa atingir um bom desempenho, é preciso fazer um escalonamento adequado, dependendo do tipo de aplicação que será escalonada. Assim, uma análise tanto dos escalonadores quanto dos algoritmos de escalonamentos é necessário.

1.2 Objetivos

Os principais objetivos deste trabalho são os seguintes:

- *Estudo dos diferentes escalonadores*, vamos estudar e comparar os escalonadores existentes nos diversos ambientes, e os motivos porque eles são usados, tipos de tarefas que eles processam, escalabilidade, etc. Se determinará o desempenho dos algoritmos de escalonamento mediante mecanismos de simulação.

- *Simulação dos algoritmos*, A através do uso de um simulador, serão implementados cada um dos algoritmos escolhidos, determinando o desempenho em diferentes cenários.

- *Algoritmos de escalonamento*, também é almejado estudar como se comportam os algoritmos com *workloads* reais.

1.3 Organização do Trabalho

No Capítulo 2, apresentamos os conceitos básicos de computação em grade, escalonadores e algoritmos de escalonamento, necessários para o entendimento do trabalho e seu objetivo. Os escalonadores estudados são apresentados no capítulo 3, os detalhes sobre o ambiente de simulação, como a arquitetura do simulador, implementação, plataforma e ambientes de trabalho são apresentados no capítulo 4.

Os algoritmos de escalonamento para tarefas dependentes e independentes são detalhados no capítulo 5, Resultados experimentais iniciais são mostrados no capítulo 6, comparando os algoritmos apresentados no capítulo 5 em diferentes cenários. No capítulo 7 é apresentado o plano de trabalho e cronograma. Finalmente, no Capítulo 8 discutimos algumas conclusões obtidas.

Capítulo 2

Conceitos

Na computação em grade temos diferentes termos, as vezes dependentes do contexto. Os conceitos básicos de computação em grade e escalonamento são apresentados neste capítulo.

Inicialmente são apresentados conceitos básicos sobre aglomerados e sobre computação em grade e as suas principais características. Depois, são descritos conceitos sobre escalonamento estático e dinâmico, e escalonamento de tarefas.

2.1 Aglomerado

Um aglomerado (também chamado *cluster*) em termos de arquiteturas computacionais, pode ser entendido como uma agregação de computadores de uma forma dedicada (ou não) para a execução de aplicações específicas de uma organização. Um aglomerado dedicado é projetado para rodar exclusivamente aplicações paralelas. Por outro lado na configuração não dedicada, além da execução de aplicações convencionais monoprocessadas, pode ser utilizado como um aglomerado eventual para execução de aplicações que solicitem um maior desempenho computacional agregado.

Os aglomerados (ou agregados, como alguns autores se referem em português) [Dan05], de uma forma geral, são compostos por computadores com uma característica intrínseca de disponibilidade de uma grande quantidade de recursos (processadores, memórias e capacidade de armazenamento) pertencentes a uma única entidade.

2.2 Computação em Grade

A "Computação em Grade" (*Grid Computing*), emergiu como uma importante nova área em meados da década de 1990, nasceu da comunidade de Processamento de Alto Desempenho, motivada pela ideia de se utilizar computadores independentes e amplamente dispersos como plataforma de execução de aplicações paralelas [FK04].

A computação em grade consiste em compartilhar de forma coordenada e dinâmica recursos por *organizações virtuais* entre várias instituições, cada organização virtual é um conjunto de indivíduos ou instituições que fornecem recursos executores e consumidores os quais definem claramente o que é compartilhado e sob que condições o compartilhamento é possível, fazendo uso dos recursos de forma coordenada e controlada. [FKT01].

Este compartilhamento coordenado é feito através de um *middleware*. Um middleware é um pacote de software que faz a interface entre o usuário e o ambiente computacional. No caso de

computação em grade existem diversas infra-estruturas de middleware desenvolvidas até agora, por exemplo, Globus [Fos05], Legion [GW97], InteGrade [GKG+04] e OurGrid [CBA+06]. Estas já permitem que coleções de máquinas heterogêneas distribuídas em aglomerados fisicamente distantes, mas interconectadas por redes de longa distância como a Internet, trabalhem em conjunto para a resolução de problemas computacionalmente pesados.

As principais características das grades são:

- Grandes, pelos recursos potencialmente disponíveis, os quais podem ser agregados na grade;
- Distribuídas: Os recursos localizam-se geograficamente distribuídos, assim, latências em movimentação de dados pode ser significativas em relação ao tempo de execução de uma aplicação;
- Heterogêneos: poder computacional, largura de banda e outras propriedades principais dos recursos podem ser significativamente diferentes;
- Diferentes políticas de acesso: Recurso diferentes e geograficamente distribuídos podem ter diferentes políticas de acesso.

2.3 Escalonamento

O escalonamento é a atribuição de tarefas aos elementos de processamento (recursos), um possível objetivo é que, essa atribuição seja efetuada de forma eficiente para minimizar o tempo das aplicações.

O objetivo pode ser maximizar a utilização dos recursos computacionais disponíveis, e minimizar os custos relativos à comunicação, isto significa minimizar o tempo de término das aplicações, *makespan*. Os diferentes tipos possíveis de escalonamento foram estudados por vários pesquisadores. Uma abordagem de classificação mais aceita está apresentada na figura 2.1 [Dan05].

Na classificação, inicialmente, os métodos de escalonamento são divididos em local e global. O escalonamento local refere-se ao problema de atribuição das tarefas ao processador local, ou seja, é aquele realizado normalmente pelo sistema operacional. O escalonamento global refere-se ao problema de decidir sobre onde executar uma tarefa sendo, portanto, seus métodos aplicáveis aos sistemas distribuídos.

2.3.1 Escalonamento Estático

No escalonamento estático a atribuição de tarefas aos processadores é realizada antes do início do programa. Assim, a atribuição de uma aplicação é estática, e uma estimação do custo computacional deve ser feita com antecedência. Uma das principais vantagens do modelo estático é que é mais fácil programar do ponto de vista do escalonador. A atribuição de tarefas é fixada a priori, e a estimativa do custo da tarefa também é simplificada.

2.3.2 Escalonamento Dinâmico

O escalonamento dinâmico é geralmente aplicado quando é difícil estimar o custo das aplicações, ou as tarefas sendo submetidas em tempo real, ou dinamicamente (*online scheduling*). Estes assumem que muito pouco se sabe a priori acerca das necessidades dos recursos de uma tarefa, ou

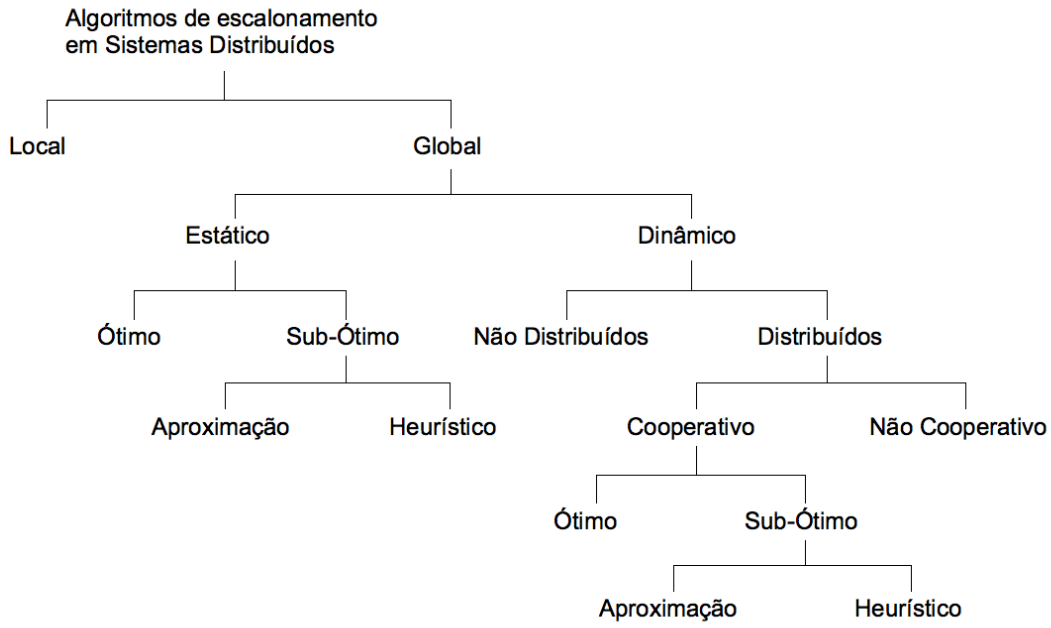


Figura 2.1: Classificação dos métodos de escalonamento [CK88].

do ambiente do qual a mesma irá ser executada. No escalonamento dinâmico pode ser realizada a redistribuição das tarefas aos processadores durante a execução da aplicação.

2.4 Escalonamento de Tarefas

Um dos passos realizados no processo de execução de aplicações em grades é o escalonamento de tarefas que compõem essas aplicações. As tarefas que compõem uma aplicação podem ter dependências entre si. Quando as dependências existem, as tarefas formam grafos direcionados para representar as dependências.

Quando as dependências não existem, as tarefas formam grafos vazios, ou seja, grafos que não possuem arestas, e são mais conhecidas como *Bag-of-Tasks* (BoT). A descrição da aplicação passada como entrada para o escalonador de tarefas depende do tipo de aplicação. Na figura 2.2 é exemplificado o grafo direcionado de uma aplicação na qual há dependências entre as tarefas, enquanto que a figura 2.3 exemplifica o grafo vazio de uma aplicação BoT [Bat10].

Neste trabalho, de acordo com a figura 2.1, nos concentraremos no escalonamento de tipo global, pelas características da computação em grade. Dentro do escalonamento global, estudaremos especificamente heurísticas de tipo estáticas e dinâmicas.

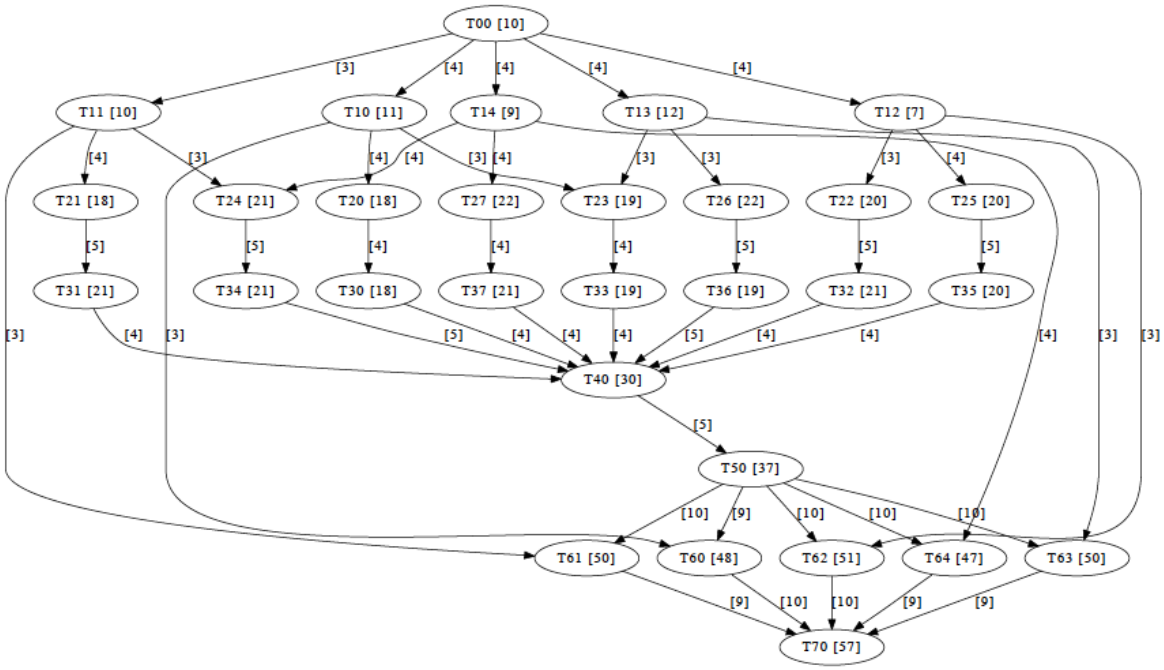


Figura 2.2: Exemplo da descrição de uma aplicação com tarefas dependentes

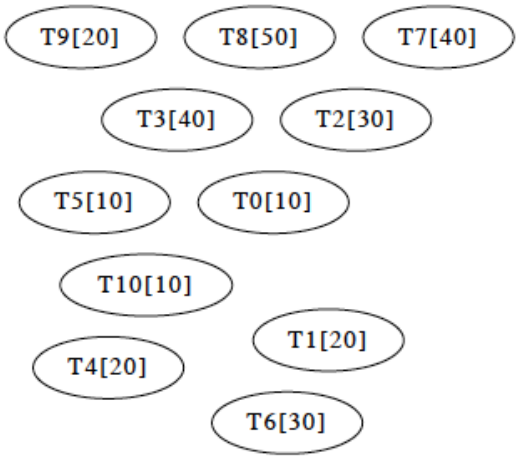


Figura 2.3: Exemplo da descrição de uma aplicação com tarefas independentes

Capítulo 3

Escalonadores

Em sistemas computacionais, podemos entender o gerenciamento de recursos quando as entidades de um recurso (CPU, memória e disco) podem ser gerenciadas. O escalonador de tarefas ou também chamado gerenciador de recursos é aquele software responsável por garantir o bom funcionamento de um sistema computacional, entre as quais temos: recebimento de tarefas por recursos e a atribuição de recursos a essas tarefas, realizando a alocação do que é buscado com o que é oferecido pela infra-estrutura.

O emparelhamento é realizado com o intuito de maximizar a qualidade de serviço estabelecida pelos usuários da grade. A qualidade de serviço pode ser dada pela minimização do tempo de espera pelos resultados de uma tarefa. Assim, para atingir essa qualidade de serviço, existem políticas de escalonamento que ditam como, quando e onde determinada aplicação deve ser executada.

Para fazer a comparação foram escolhidos os escalonadores mais representativos na literatura atual, os quais são usados por instituições importantes na área de computação de alto desempenho, tomando somente escalonadores de código não proprietário.

3.1 OAR

O OAR [CDCG+05] é um escalonador de recursos para aglomerados de grande porte. Desenvolvido no Instituto Politécnico Nacional de Grenoble na França. O OAR investe na simplicidade e nos benefícios da linguagem SQL, usando ferramentas de alto nível como a linguagem *Perl*, o banco de dados *MySql* e uma ferramenta opcional chamada Taktuk [CHR09]. O OAR é livre e possui código aberto com licença GPL.

A seguir as principais características do OAR:

- Execução de tarefas interativas ou de lote;
- Possui controle de admissão;
- *Walltime*;
- Emparelhamento de tarefas/recursos;
- Propriedade de preempção;
- Suporte para Multi-escalonadores (FIFO simples e FIFO com emparelhamento);

- Multi-filas com prioridade;
- Filas de Melhor-esforço (para explorar recursos ociosos);
- Verificação de nós antes de executar uma tarefa;
- Ferramentas de visualização (Monika e DrawGantt)[1];
- Ausência de *daemons* em nós executores;
- RSH e SSH como protocolos de execução remota (gerenciados pelo Taktuk);
- Inserção/Supressão dinâmica de nós de computação;
- *Logging* de informações;
- Tarefas moldáveis;
- Notificações de *Checkpoint*;
- Resubmissões;
- Mecanismos de políticas de *Backfilling*;
- Mecanismos de "reserva" avançada;
- Grid integração com o sistema CIGRI[2];

3.1.1 Arquitetura

O OAR é baseado sobre um nível mais abstrato que minimiza a complexidade de concepção de seu software. A arquitetura interna é construída em cima de dois componentes principais: uma ferramenta genérica e escalável para a administração do aglomerado escrita na linguagem de programação Perl e um banco de dados MySql, como único jeito de compartilhar informação. Como é mostrado na figura 3.1.

No banco de dados são armazenados todos os dados internos sobre aplicações e recursos, o acesso é unicamente por meio de comunicação entre módulos. O casamento entre recursos e o armazenamento e consulta de logs do sistema são realizados através de chamadas SQL. Essa prática atribui características de robustez e eficiência ao OAR, posto que os bancos de dados têm poucas chances de se tornar um gargalo na escalabilidade do sistema por serem capazes de processar eficientemente milhares de consultas simultaneamente. A robustez somente depende dos módulos que precisam deixar o sistema em um estado coerente.

A outra parte do servidor é composta por um conjunto de módulos independentes implementados como *scripts* Perl. Cada um dos módulos é responsável por tarefas específicas, como por exemplo, iniciar e controlar a execução de aplicações, assim para alcançar a totalidade destas tarefas, os módulos interagem com o banco de dados através do consultas SQL.

[1] http://oar.imag.fr/users/tools/
[2] http://cigri.imag.fr/

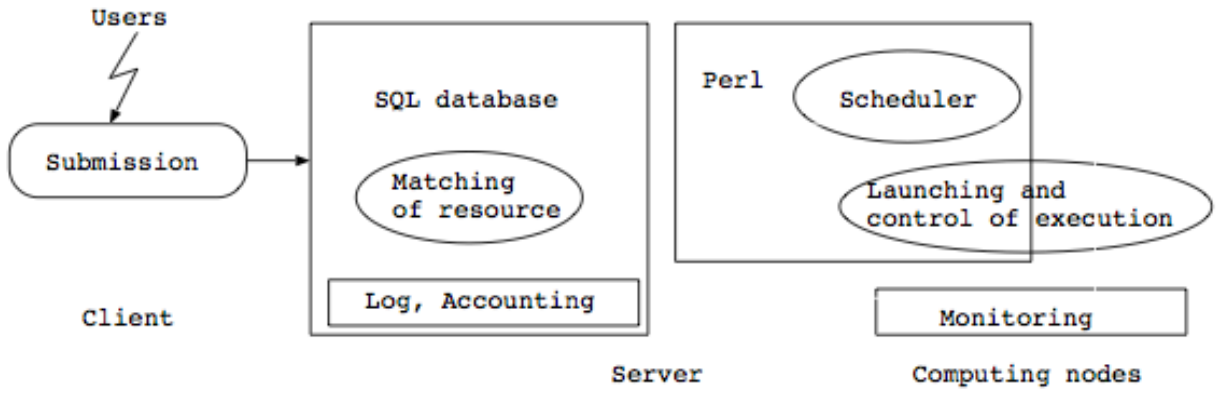


Figura 3.1: Arquitetura global do OAR [CDCG+05].

3.1.2 Monitoramento

O monitoramento das tarefas é tratado por uma ferramenta separada chamada Taktuk. O Taktuk permite a execução paralela de comandos em grandes aglomerados. Taktuk é altamente paralelizado e distribuído.

Dentro do OAR, essa ferramenta é utilizada para realizar tarefas administrativas nos nós dos aglomerados através de um serviço de execução remota. Através do Taktuk, nós falhos podem ser detectados pelo tempo de resposta dos mesmos, respeitando-se um tempo limite que pode ser modificado pelo administrador do escalonador. Todavia, apesar da sua versabilidade, o Taktuk não faz análise de padrões de uso dos recursos.

3.1.3 Escalonamento

Um dos objetivos do OAR, é a sua simplicidade e oferecer uma plataforma aberta para experimentos e pesquisa. Assim, embora o módulo de escalonamento implementado em OAR possua muitas funcionalidades, os algoritmos que ele usa são ainda bastante simples.

Assim, todas as funcionalidades mais importantes, tais como, prioridades sobre tarefas, reserva de recursos e *backfilling* são implementados.

O algoritmo padrão implementado no OAR consiste no FCFS (*First Come First Served*), assim todas as aplicações são ordenadas de acordo com o seu tempo de chegada na fila. Políticas de escalonamentos podem ser implementados no OAR com outras linguagens de programação.

3.1.4 Lugares de Ação

O OAR é o responsável pelo gerenciamento de recursos e agendamento de tarefas do projeto Grid5000[3] e do projeto CIMENT[4].

Realiza principalmente três tarefas

- Reserva de nós para um determinado período, em nome de um usuário solicitante;

[3] www.grid5000.fr
[4] ciment.ujf-grenoble.fr

- Agenda as tarefas dos usuários sobre os nós reservados, garantindo um tempo de utilização coerente;
- Liberar recursos no final de cada reserva.

Este ano, é o terceiro ano consecutivo que o OAR foi escolhido para ser o *mentoring* organizador no *Google Summer of Code* (GSoC)[5].

3.2 OurGrid

O middleware OurGrid [CBA+06] é um projeto de software livre com licença GPL para computação em grade. Permite a criação de uma grade *peer-to-peer free-to-join*. A primeira versão foi liberada no dezembro de 2004 e foi usada por centenas de usuários para acelerar a execução de aplicações *Bag-of-Tasks* (Saco de Tarefas), ou seja, aplicações paralelas cujas tarefas são independentes. Este tipo de aplicações é executada de forma paralela na grade, porém não há comunicação entre si.

O OurGrid tem uma comunidade ativa de usuários e desenvolvedores. O software é escrito na linguagem de programação Java, permitindo que qualquer recurso capaz de executar uma máquina virtual Java seja aproveitado pela grade.

3.2.1 Arquitetura

Como é mostrado na figura 3.2 O OurGrid possui principalmente três componentes:

- *The MyGrid broker*;
- *The OurGrid peer*;
- *The SWAN security service*.

Através do MyGrid [CPC+03], o usuário consegue executar suas aplicações em todos os recursos que ele tenha acesso. O MyGrid manda uma requisição para o OurGrid Peer, assim o *OurGrid Peer* tentará obter recursos nos diferentes peer da grade, desta forma é possível a cooperação dos diferentes *peers* (nós) da grade. Neste processo os usuários locais sempre tem prioridade maior que os outros usuários da grade. Se não houver nenhuma requisição de usuários locais, os recursos disponíveis são considerados ociosos (*idle*). Então um esquema chamado *Network of Favors* [CBA+06] é utilizado. *Network of Favors* é um esquema de alocação de recursos baseado em reputação. *Peers* que são mais usados tem uma melhor reputação e, desta forma recebem uma prioridade maior no momento que requisitam recursos de outros *peers*. Esta abordagem evita o fenômeno de *free-riding*, no qual *peer* somente consome recursos.

O SWAN (*Sandboxing Without A Name*), é uma solução de segurança do OurGrid, baseada em máquinas virtuais Xen que isola o código desconhecido em um *sandbox*. Desta forma, as tarefas da grade que executam em uma máquina específica não podem danificá-la ou utilizar a rede de maneira indevida.

[5] http://wiki-oar.imag.fr/

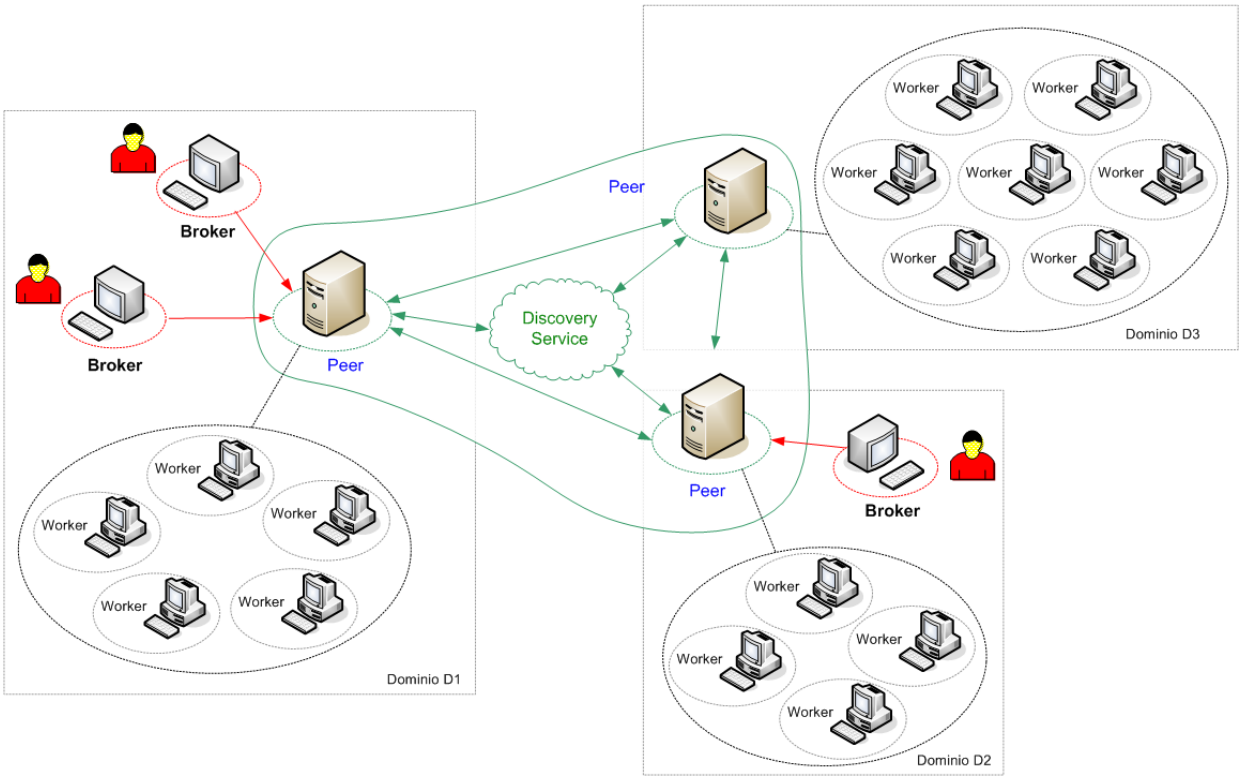


Figura 3.2: Arquitetura do OurGrid [CBA+06].

3.2.2 Escalonamento

Ainda com a simplicidade das aplicações Bag-of-Tasks, o escalonamento delas sobre grades é difícil. Primeiro, escalonamentos eficientes dependem de informações sobre a aplicação (por exemplo tempo de execução estimado, tamanho das tarefas) e recursos (velocidade de processamento, topologia da rede, carga dos recursos, etc.). Mas, é muito complexo obter este tipo de informações num sistema tão grande e amplamente disperso, como uma grade. Assim, o MyGrid, o escalonador do OurGrid, tenta usar algoritmos de escalonamento que não dependem desse tipo de informações:

- *Workqueue*;
- *Workqueue with Replication* [dSCB03];
- *Storage Affinity* [SNCBL04].

O *Workqueue* simplesmente escalona as tarefas submetidas aos recursos disponíveis em uma ordem arbitrária. O WQR (*WorkQueue with Replication*) é uma extensão do *Workqueue* sendo que, após a submissão de todas as tarefas, o escalonador passa a submeter réplicas das tarefas em execução até que não haja mais recursos disponíveis. Como a estratégia de *Workqueue* não utiliza qualquer informação acerca das aplicações ou dos recursos, a replicação funciona como um mecanismo que procura compensar alocações más sucedidas (por exemplo, escalonar tarefas em

recursos lentos ou sobrecarregados). Isso faz com que o WQR consuma mais recursos do que os escalonadores que utilizam informações sobre a disponibilidade dos recursos. Cientes deste problema, os desenvolvedores lançaram a segunda versão do MyGrid com uma nova opção de escalonamento: o *Storage Affinity*. Esse algoritmo de escalonamento mantém informações sobre a quantidade de dados que os nós contém sobre uma determinada aplicação. Dessa forma, sempre que uma decisão de escalonamento precisa ser feita, o *Storage Affinity* escolhe o recurso que já contém a maior quantidade de dados necessários para o processamento. Essa abordagem é mais adequada para as aplicações do tipo saco de tarefas (BoT) que processam grandes quantidades de dados, já que o tempo de transferência dos dados para as máquinas que irão processá-los representam uma sobrecarga considerável no tempo total de execução das aplicações.

3.3 Condor

O Condor [LLM88, FTF+02, TTL05], é um software especializado para gerenciar aplicações de computação intensiva. Como outros escalonadores de tarefas (*batch systems*), o Condor provê mecanismos de enfileiramento e priorização de aplicações, políticas de escalonamento e monitoração de recursos. Os usuários submetem aplicações paralelas ou seriais ao Condor, e o Condor as enfileira, escolhe quando e onde executar os trabalhos baseado em uma política, monitora cuidadosamente o seu progresso e, finalmente, informa ao usuário após a conclusão.

O Condor é um dos sistemas pioneiros na área da computação oportunista, lançado em 1984, desenvolvido pela equipe Condor na universidade de *Wisconsin-Madison*, influenciou o interesse acadêmico na busca de soluções que permitissem o uso de ciclos ocioso de estações de trabalho para a execução de aplicações paralelas de alto processamento. O Condor é software livre, possui licença Apache versão 2.0.

3.3.1 Matchmaking

O escalonamento no Condor é feito através de um mecanismo chamado *matchmaking* [RLS98], o qual decide quando, onde e como será executada uma determinada tarefa. Como mostrado na figura 3.3.

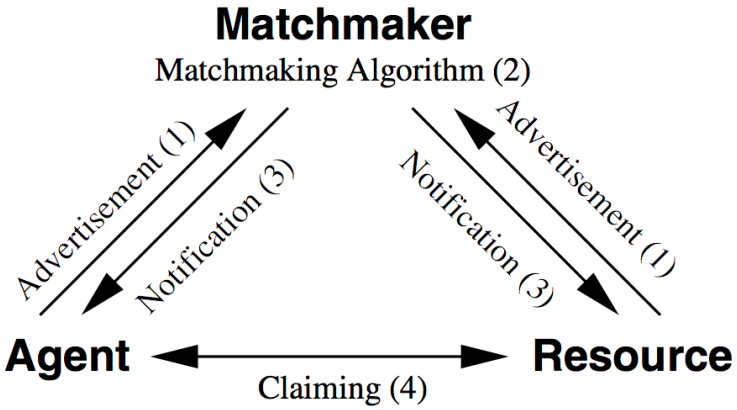


Figura 3.3: Matchmaking [TTL05].

Cada recurso e tarefa que precisa de ser executada, através de um componente denominado agente, anunciam suas respectivas existências a entidades de *matchmaker*. Essa atividade é chamada de *Classified advertisements* ou *ClassAds*. Uma vez anunciados, o *matchmaker* cria pares que satisfazem às necessidades e limitações um do outro. Esses pares são notificados à tarefa e recurso envolvidos no passo 3 e, finalmente, no passo 4 essas partes negociam possíveis termos e começam a execução. O mecanismo de *classAds* é feito em cima de uma linguagem própria, da forma *nome* = *valor*, para a especificação de um tarefa de uma máquina. Um exemplo encontra-se na figura 3.4.

Job ClassAd	Machine ClassAd
[	[
MyType = "Job"	**MyType** = "Machine"
TargetType = "Machine"	**TargetType** = "Job"
Requirements =	**Machine** = "nostos.cs.wisc.edu"
((other.Arch=="INTEL" &&	**Requirements** =
other.OpSys=="LINUX")	(LoadAvg <= 0.300000) &&
&& other.Disk > my.DiskUsage)	(KeyboardIdle > (15 * 60))
Rank = (Memory * 10000) + KFlops	**Rank** = other.Department==self.Department
Crod = "/home/tannenba/bin/sim-exe"	**Arch** = "INTEL"
Department = "CompSci"	**OpSys** = "LINUX"
Owner = "tannenba"	**Disk** = 3076076
DiskUsage = 6000	]
]	

Figura 3.4: Dois exemplos de *ClassAds* do Condor [TTL05].

3.3.2 Arquitetura

O Condor pode ser usado como gerenciador de um aglomerado com nós computacionais dedicados (por exemplo, um aglomerado *Beowulf*). Mecanismos próprios do Condor permitem aproveitar o poder computacional ocioso de estações de trabalho.

Uma infra-estrutura Condor, é formada por um nó chamado gerenciador central e um número arbitrário de outros nós divididos entre nós de execução (recursos que doam poder computacional) e nós submissores, este conjunto de recursos é chamado *pool*, assim como é mostrado na figura 3.5. Cada componente da figura apresenta funcionalidades bem definidas, implementadas através de *daemons* (representados dentro de retângulos de cantos arredondados).

- Gerenciador Central (*Central Manager*). Para cada *pool* no Condor existe um único gerenciador central. O gerenciador central é responsável por coletar informação a respeito dos estados dos recursos das máquinas executoras e dos pedidos de execução das máquinas submissoras. Dessa maneira, ele apresenta um *daemon condor_collector* responsável por receber notificações regulares de *classAds* e armazená-las para posteriores consultas. O gerenciador central também é responsável dos *matchmaking*, posto que possui todas as informações necessárias.

- Nós de execução (*Execution Machine*). O nó de execução que é representado pelo *daemon*

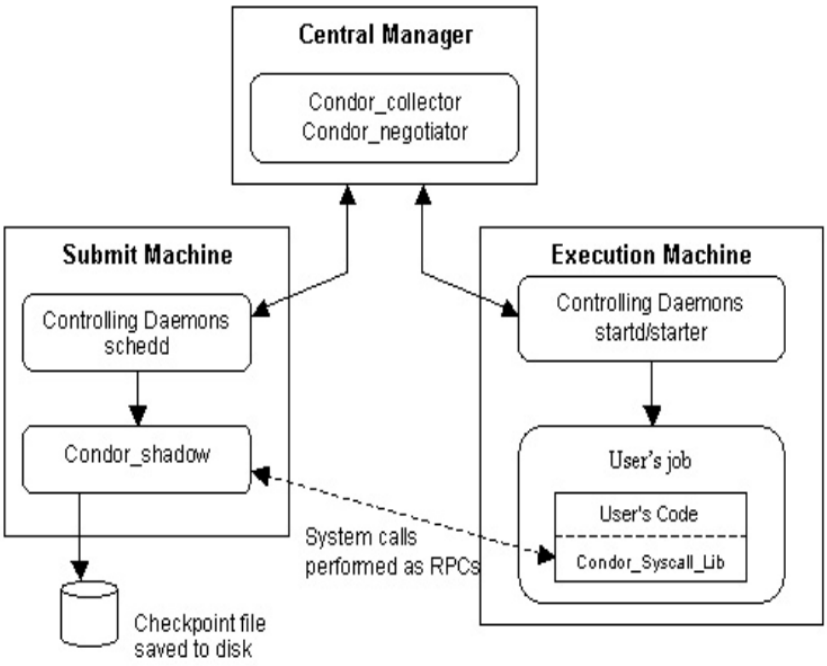


Figura 3.5: Arquitetura de um *pool* no Condor [CWT+04].

startd, permite executar aplicações garantindo a política na qual aplicações remotas serão iniciadas, interrumpidas ou finalizadas. Ele anuncia as suas capacidades e informações de uso bem como os requisitos e preferências sobre um *match* ao gerenciador central, e gerencia a execução local do trabalho (através do *daemon starter*). O *daemon starter* configura a execução e monitora a aplicação durante o seu tempo de vida.

- Nós submissores (*Submit Machine*). Estes nós permitem aos usuários submeter aplicações através do *daemon schedd* e inserí-los em uma fila. Será pedida a alocação de recursos para a aplicação ao gerenciador central durante um ciclo de negociação. Uma vez que a aplicação foi alocada aos recursos, o *daemon schedd* disparará o *daemon shadow*, responsável para gerenciar a execução remota da aplicação, e executar tarefas como o estado de checkpointing, o reescalonamento do trabalho no caso de falha, ou realizar chamadas do sistema feitas pela aplicação executadas remotamente na máquina local.

3.4 PBS/OpenPBS/Torque

O PBS, Portable Batch Scheduler[6] [Hen95], é um sistema gerenciador distribuído de carga (*workload management*) e gerenciador de tarefas (*job scheduling*), originalmente deselvolvido para gerenciar recursos de computação aeroespacial da NASA. O PBS desde então se tornou o líder na gestão da carga em supercomputadores e o padrão para gerenciar tarefas no Linux [WET03].

O PBS foi feito para administrar, monitorar a carga computacional sobre um conjunto de um ou

[6] Altair Engineering, http://www.pbsgridworks.com. Último acesso no 22 Nov, 2010

mais computadores e gerenciar as tarefas. Foi desenvolvido inicialmente pela *Veridian Systems* para a NASA, posteriormente a *Veridiam Systems* foi adquirida pela *Altair Engineering*, que distribui duas versões do PBS:

- *PBS Professional*, versão comercial,
- *OpenPBS*, distribuição livre, que não é mantida pela *Altair Engineering*.

O PBS possui três principais papéis:

- Fila. A coleta de tarefas para serem executadas em um computador. Os usuários enviam as tarefas para o sistema de gerenciamento de recursos onde eles são colocados em fila até que o sistema fique pronto para executá-los.
- Escalonamento: O processo de seleção de tarefas para executá-las, quando e onde, de acordo a uma política pré-determinada. O escalonador provê diversas maneiras de distribuir a carga de tarefas entre os recursos disponíveis baseado na configuração de hardware, na disponibilidade de recursos ou mesmo na utilização de dispositivos de entrada, com o objetivo de maximizar o uso eficiente de recursos.
- Monitoramento: O processo de rastreamento e reserva dos recursos. Isso abrange o monitoramento tanto a nível de usuário e de nível de sistema, bem como a monitorização de tarefas em execução.

A figura 3.6. mostra os componentes do PBS.

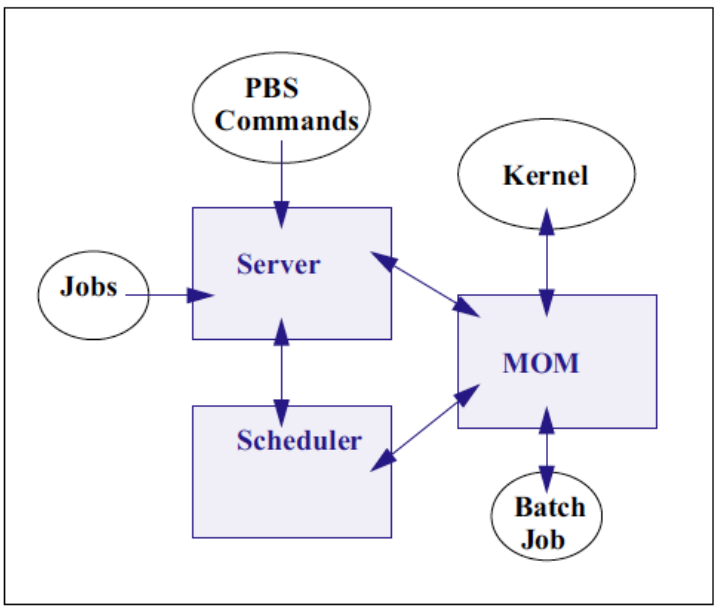


Figura 3.6: Componentes do PBS [CWT+04].

Alem das duas versões do PBS, existem mais uma chamada *Torque* [Sta06], que é um derivado do OpenPBS e é ativamente desenvolvido, suportado e mantido pela *Cluster Resources Inc.*.

3.4.1 Torque

O *Terascale Open-Source Resource and QUEue Manager* Torque [Sta06] é um gerenciador de recursos de código aberto. É um esforço da comunidade baseado no PBS, especificamente na versão 2.3.12 do *OpenPBS*, com mais de 1200 linhas de código modificadas, o Torque incorporou melhorias significativas ao PBS, por exemplo: na escalabilidade, tolerância a falhas e extensões das caracteristicas básicas do PBS, com contribuições de importantes organizações tais como NCSA (*National Center for Super-computing Applications*), OSC (*Ohio Supercomputer Center*), USC (*University of Southern California*), a *U.S. Dept. of Energy*, Sandia, PNNL, *U. de Buffalo*, TeraGrid, e muitas outras organizações líderes no HPC (*High Performance Computing*).

O Torque é software livre, usado em milhões de sítios de pesquisa no globo. Com os comandos disponibilizados é possível alocar recursos, escalonar e gerenciar a execução, o monitoramento do estado das tarefas submetidas.

Algumas características inseridas ao OpenPBS pelo Torque são:

- **Tolerância a Falhas**, foram adicionadas novas condições, detecções e tratamento de falhas. Condições de falha são verificadas e tratadas, há suporte de verificação nos *scripts* dos nós.
- **Interface de Escalonamento**
  - Extensão da interface de consulta proporcionando ao escalonador com informações adicionais mais precisas;
  - Extensão da interface de controle que permite ao escalonador um maior controle sobre o comportamento das tarefas e seus atributos;
  - Permite recolher estatísticas das tarefas concluídas.
- **Escalabilidade**
  - Melhorias significativas no servidor para a implementação do modelo MOM (*Machine-Oriented Miniserver*) [Hen95] de comunicação;
  - O gerenciador tonou-se escalável, possui a habilidade para lidar com aglomerados maiores (mais de 15 TF/2500 processadores);
  - Habilidade para lidar com tarefas maiores (mais de 2000 processadores).
  - Capacidade para suportar mensagens maiores no servidor.
- **Usabilidade**, adição de novas funcionalidades de *logging*.

Arquitetura

Um aglomerado com Torque consiste de um nó principal e muitos outros nós. O nó principal executa o *daemon pbs_server* e os outros nós executam o *daemon pbs_mom*. Os comandos clientes para submeter e gerenciar tarefas podem ser instalados em qualquer nó (incluindo nós onde não foram executados os *daemons pbs_server* ou *pbs_mom*).

O nó principal também executa um *daemon* para o escalonador. O escalonador interage com o *pbs_server* para tomar decisões de política local para o uso dos recursos e assim alocar tarefas aos nós. Um simples escalonador de tipo FIFO, e um código para construir um escalonador mais avançado é fornecido na distribuição fonte do Torque. A maioria dos usuários TORQUE optam por usar um pacote, avançado de escalonador, como Maui ou Moab.

Os usuários submetem tarefas ao *pbs_server* usando o comando *qsub*. Quando *pbs_server* recebe uma nova tarefa informa ao escalonador. Quando o escalonador encontra os nós para a tarefa, envia instruções para executar a tarefa com a lista de nós ao *pbs_server*. Então, o *pbs_server* envia a nova tarefa para o primeiro nó na lista de nós com instruções para iniciar a tarefa.

Em sua versão 2.5.2, o Torque passou a oferecer um mecanismo de redirecionamento do ambiente gráfico X11, e a dar suporte a comandos clientes na plataforma *Windows* via *Cygwin*.

3.5 Maui

O Maui [BHK+00], é um escalonador de tarefas de código livre para aglomerados e supercomputadores. Ele surgiu com o propósito de auxiliar algumas carências de desempenho das políticas de escalonamento implementadas no sistema *IBM LoadLeveler*, por exemplo a grande taxa de ociosidade de recursos paralelos na espera de tarefas.

3.5.1 Características

O objetivo principal no Maui é escalonar as tarefas de forma especializada. Ele pode ser usado como um componente adicional em sistemas de escalonamentos, por exemplo, o PBS e o Torque. Maui estende as capacidades desses sistemas, acrescentando as seguintes características:

- **Priorização de tarefas**, Maui atribui pesos aos diferentes objetivos, assim um valor global ou prioridade pode ser associado no escalonamento;
- **Reserva de Recursos**, cada reserva é constituída por três componentes principais: a lista de recursos, o tempo de reserva e a lista de controle de acesso. A tecnologia de reserva antecipada fornece muitas características ao Maui, incluindo *backfill*, escalonamento baseado em prazos, suporte a QoS (*Quality of Service*) e meta-escalonamento;
- **Suporte a QoS**, a QoS é configurado pelos administradores, cada tipo de QoS apresenta um conjunto de prioridades, políticas de isenções e as configurações de acesso aos recursos. Os administradores podem configurar usuários, grupos, contas e classes como beneficiários dos privilégios de QoS;

- **Abrangente equidade de políticas**, o Maui oferece algumas ferramentas flexíveis que ajudam na necessidades comuns de equidade, isto implica para todos os usuários igualdade de acesso aos recursos computacionais;

- **Políticas de Backfill**, é uma otimização de escalonamento que permite que um escalonador faça melhor uso dos recursos disponíveis, executando trabalhos fora da ordem estabelecida em determinada fila. No caso da tarefas de maior prioridade e com requisições mais simples de recursos pode ser executada no lugar da primeira;

- **Suporte de diagnóstico** o Maui fornece um número de comandos para o diagnóstico do comportamento do sistema. Esses comandos de diagnóstico apresentam detalhadamente aspecto sobre problema no escalonamento, relatório do desempenho e sobre condições inesperadas ou potencialmente erradas de qualquer operação em curso;

- **Modo de Teste**, o Maui suporta o modo de escalonamento chamado “TEST”. Nesse modo, o escalonador funciona como se estivesse executando normalmente, mas o Maui torna-se incapaz de iniciar, interromper, cancelar ou modificar atributos das aplicações ou dos recursos.

Neste capítulo vimos diversos escalonadores usados em grades em funcionamento atualmente. É interessante notar que na maioria dos casos são usados escalonadores relativamente simples. No Capítulo 5 veremos estes algoritmos e mais alguns da literatura.

Capítulo 4

Simulador

Um dos aspectos fundamentais no desenvolvimento de aplicações e novas estratégias em sistemas distribuídos e especificamente na computação em grade é a sua avaliação. Neste cenário geralmente ferramentas de simulação são usadas, isto é principalmente porque a realização de experimentos reais em recursos reais não é apropriado pelas seguintes rações:

- Aplicações reais devem rodar por longos períodos de tempo, e não é viável para executar um grande número de experimentos de simulação neles;
- A utilização de recursos reais, torna difícil a exploração de uma grande variedade de configurações de recursos;
- Variações na carga de recursos ao longo do tempo tornam difícil a obtenção de resultados reprodutíveis.

Ainda a simulação na computação em grade sendo mais complexa pelas características já comentadas das grades, atualmente existem diversos simuladores para grades, entre eles podemos citar ao Bricks [TMN+99], que é uma ferramenta empregada para simular diversos comportamentos de um sistema computacional distribuído, como o de algoritmos de escalonamento, da topologia de sistemas de tipo cliente-servidor, e de estratégias de processamento para redes e servidores; o OptorSim [BCC+03], criado para o estudo de algoritmos de escalonamento dedicados para a migração e replicação dos dados. Foi implementado em Java, o que favorece uma maior portabilidade. Está disponibilizado no *sourceforge*[1] e a última atualização do código é do ano 2008.

O GridSim [BM02] permite modelagem e simulação de entidades em sistemas de computação paralela e distribuída, como usuários, aplicações, recursos e *resource brokers* ou escalonadores, para o desenho e avaliação de algoritmos de escalonamento. GridSim fornece um mecanismo global para a simulação de diferentes classes de recursos heterogêneos, usuários, aplicações, e *resource brokers*. Pode ser usado para simular aplicações de escalonamento para diferentes domínios de sistemas de computação distribuída como aglomerados ou grades. Similar ao Optorsim, o GridSim é feito em Java e está disponibilizado no *sourceforge*[2]. A última versão do GridSim é a 5.0.1, liberada no outubro do 2009 e a lista de usuários está em estado inativo.

[1] http://sourceforge.net/projects/optorsim/
[2] http://sourceforge.net/projects/gridsim/

O SimGrid [CLQ08], é um *toolkit* que fornece importantes funcionalidades para a simulação de aplicações distribuídas em ambientes heterogêneos. Este simulador é implementado em C e possui diversas funcionalidades para simulação de ambientes distribuídos, e uma comunidade ativa[3].

O simulador usado no presente trabalho é o SimGrid, pelas qualidades comentadas anteriormente. Alem disso ela possui exatamente o que é preciso para nossos propósitos. Na seguinte seção será apresentado o SimGrid e suas principais características.

4.1 SimGrid

O SimGrid é uma ferramenta que fornece funcionalidades chave para simulação de aplicações distribuídas em ambientes distribuídos heterogêneos. O objetivo específico do simulador é facilitar a pesquisa na área de aplicações paralelas e distribuídas em plataformas de computação distribuída que vão desde simples redes de estações até grades computacionais.

4.1.1 Arquitetura

A arquitetura do SimGrid é mostrada na figura 4.1, é descrita na forma de camadas, cada uma contendo um ou mais componentes.

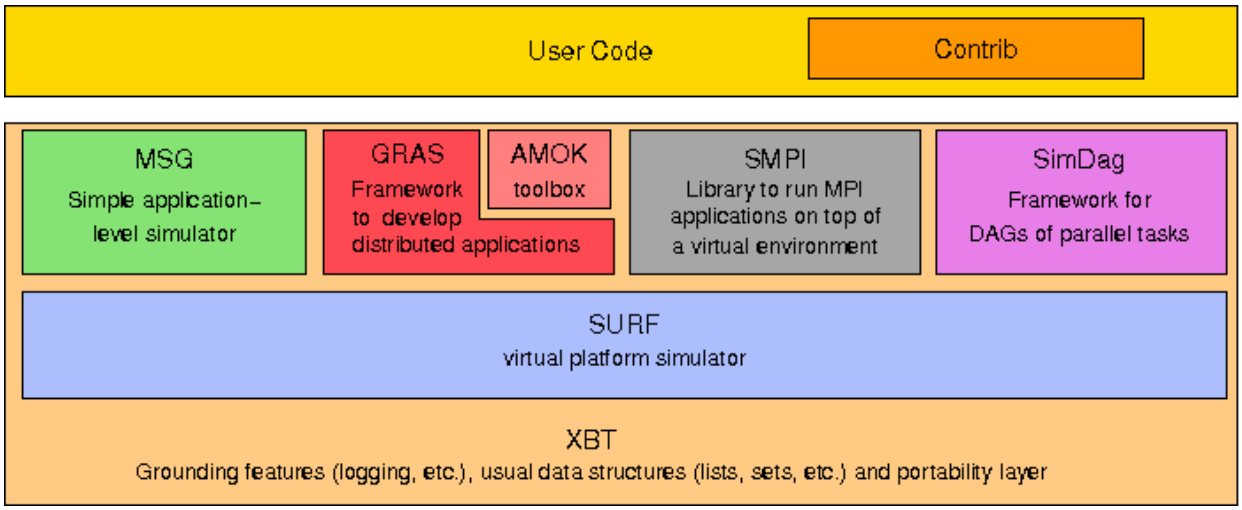


Figura 4.1: Componentes do SimGrid

O código da aplicação escrito pelo usuário, utilizando a biblioteca SimGrid, faz uso de algum dos ambientes de programação situados na primeira camada. O projeto SimGrid disponibiliza também um conjunto de exemplos e aplicações desenvolvidos e compartilhados por usuários.

4.1.2 Componentes do SimGrid

De acordo com o tipo de aplicação que se deseja avaliar, é possível escolher um módulo adequado. Primeiro serão detalhadas as camadas correspondentes do SimGrid, na seção de "ambientes de programação", bem como as situações em que seu uso é indicado.

[3] http://simgrid.gforge.inria.fr/

Camada: Ambientes de Programação

SimGrid fornece diversos ambientes de programação, construído sobre um único núcleo (*kernel*). Cada ambiente objetiva um alvo específico e constitui um paradigma diferente. Para escolher o mais adequado, tem que se pensar sobre que se quer fazer e qual seria o resultado do trabalho. A figura 4.2 mostra os componentes da camada.

- **MSG**: Foi o primeiro ambiente de programação disponibilizado e é o de uso mais difundido. Ele é usado para modelar aplicações como processos seqüenciais concorrentes (*Concurrent Sequential Processes*);
- **SMPI**: (*Simulated MPI*), para simulações de códigos MPI. Simulação do comportamento de uma aplicação MPI usando técnicas de emulação;
- **GRAS**: (*Grid Reality And Simulation*), possibilita a execução de aplicações reais para o estudo e teste dela. Acima da API do GRAS, tem uma *toolkit* chamada AMOK (*Advanced Metacomputing Overlat Kit*) que implementa em alto nível diversos serviços necessários às aplicações distribuídas;
- **SimDag**: ambiente dedicado à simulação de aplicações paralelas, por meio do modelo DAGs (*Direct Acyclic Graphs*). Com este modelo é possível especificar relações de dependência entre tarefas de um programa paralelo.

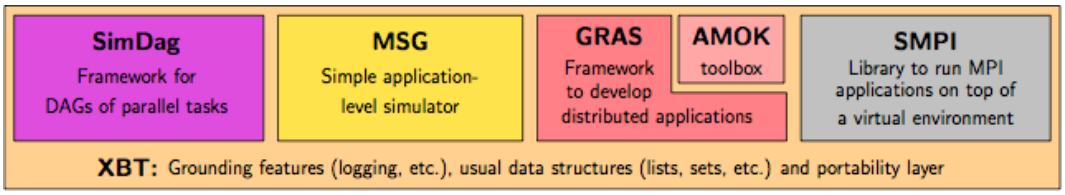


Figura 4.2: Camada: Ambientes de Programação(*Programmation environments*)

Camada: Núcleo de simulação

O núcleo das funcionalidades, para simular uma plataforma virtual é fornecida pelo módulo chamado *SURF*. É de muito baixo nível e não se destina a ser utilizado como tal pelos utilizadores finais. Em vez disso, serve de base para a camada de nível superior.

Uma das principais características do *SURF* é a capacidade de mudar de forma transparente o modelo utilizado para descrever a plataforma. Isto facilita bastante a comparação dos vários modelos existentes na literatura.

Camada: Base

A base da ferramenta é constituída pelo XBT (*eXtended Bundle of Tools*). É uma biblioteca portável, fornecendo alguns recursos como registro de *logs*, lançamento de exceções e suporte a configurações.

O XBT também possui as seguintes estruturas de dados: *Dynar*: matriz dinâmica genérica, *Fifo*: fila genérica, *Dict*: dicionário genérico, *Heap*: um *heap* genérico, *Set*: conjunto de dados genéricos e *Swag*: tipo de dado baseado em listas encadeadas.

4.1.3 Implementação e Documentação

O SimGrid é implementado na linguagem de programação C. Algumas técnicas de otimização são empregadas na manipulação dos *traces*: eles podem ser compartilhados pelos recursos; conjuntos grandes de *traces* são carregados na memória apenas quando necessários, e são descartados após o uso.

O SimGrid que é *opensource*, funciona em modo texto, e está disponível para ambientes Linux, Windows e MacOS. Ele possui uma boa documentação que pode ser obtida no site oficial do projeto[4],

4.1.4 Modelagem da Plataforma da Grade e os Workloads

Uma importante característica do SimGrid é a especificação da plataforma computacional por meio de arquivos com formato XML (*Extensible Markup Language*). É possível especificar todo o conjunto de recursos da grade, assim como a estrutura de interconexão entre eles. Os recursos computacionais que executam tarefas são modeladas pelo seu poder computacional, enquanto que os *links* de comunicação são modelados pela sua largura de banda e latência. Entre dois recursos podem haver mais de um *link*, e um dado *link* pode ser utilizado tanto para enviar quanto para receber tarefas [FQS08].

No caso do ambiente de tarefas, é usado também um arquivo XML. No ambiente de programação MSG o arquivo especifica os processos que serão executados em cada recurso, e argumentos necessários e opcionais pertencentes a cada processo.

No caso do ambiente de programação SimDag, é possível criar todas as tarefas e as dependências diretamente do código. Também, SimGrid possui funções que facilitam ao desenvolvedor na criação de tarefas. O *SD_daxload()* e o *SD_dotload()*, são as duas funções até agora disponíveis, essas funções dão suporte respectivamente para dois formatos pré-estabelecidos, o formato DAX (*Directed Acyclic Graph in XML*) [BCD+08] [5]. e o formato DOT[6], que é uma linguagem para descrever DAGs.

Este capítulo apresentou ao simulador SimGrid, o qual será usado neste trabalho, foram descritas a arquitetura básica e os seus componentes. Também foi apresentando brevemente cada camada, a modelagem da plataforma e dos workloads.

[4] simgrid.gforge.inria.fr/
[5] https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
[6] http://www.graphviz.org/doc/info/lang.html

Capítulo 5

Algoritmos de Escalonamento

No capitulo 3 pudemos perceber que apesar dos algoritmos de escalonamento sofisticados disponíveis na literatura, a maioria dos escalonadores usa uma fila simples, por exemplo uma fila de tipo FIFO (*First In First Out*). Alguns usam técnicas de replicação no escalonamento, para tentar diminuir o tempo de término das tarefas em processamento. Porém a maioria deles é extensível, isto significa que é possível implementar algoritmos de escalonamento especializados.

A utilização de um algoritmo de tipo FIFO é adequada somente em arquiteturas homogêneas, arquitetura representada por $\langle P||C_{max}\rangle$, onde P representa o tipo de máquinas (recursos) idênticas e o C_{max} representa o *Makespan*. Na atualidade existem algoritmos de escalonamento em diferentes cenários, mas já vimos que o problema de escalonamento é NP-Completo [Pin08]. O problema de escalonamento se torna mais desafiador devido a algumas características únicas pertencentes à computação em grade, tais como, a sua heterogeneidade e alta dinamicidade. Infelizmente, algoritmos de escalonamento de sistemas paralelos e distribuídos tradicionais, como é o caso do FIFO, o qual é usado em sistemas homogêneos e dedicados, podem não ser bem adaptados para essas novas características [DA06].

Os tipos de aplicações avaliadas neste trabalho serão: aplicações com tarefas dependentes e aplicações com tarefas independentes. Na seguinte seção, serão apresentados os algoritmos mais relevantes de acordo com a literatura.

5.1 Algoritmos de Escalonamento para Tarefas Independentes

No contexto de aplicações executadas nas grades computacionais, aplicações com tarefas independentes podem ser escalonadas em qualquer ordem. Este tipo de aplicações é referenciada na literatura como aplicações *Bag-of-Tasks* (BoT) [CPC+03].

5.1.1 O Algoritmo WQR

O algoritmo *Workqueue with Replication* (WQR) [dSCB03], foi criado para solucionar o problema de obtenção de informações sobre a aplicação e os recursos da grade.

O WQR em sua fase inicial é similar à heurística de escalonamento tradicional *Workqueue* [dSCB03], onde as tarefas são escolhidas em uma ordem arbitrária e enviadas ao processador quando estes estiverem disponíveis.

As heurísticas WQR e *Workqueue* passam a diferir no momento em que um processador se torna

disponível e não há mais nenhuma tarefa pendente para executar. Neste momento, *Workqueue* já finalizou seu trabalho e apenas aguarda a finalização de todas as tarefas. Porém, o WQR inicia sua fase de replicação para as tarefas que ainda estão executando. Note que, na fase de replicação, quando uma tarefa original finaliza sua execução primeiro que suas réplicas, estas são interrompidas. Caso contrário, quando alguma réplica finaliza primeiro, a tarefa original e as demais réplicas dela são interrompidas.

WQR alcança bons níveis de desempenho sem a utilização de informações dinâmicas sobre os processadores, conexões de rede ou mesmo o tempo de execução das tarefas, considerando aplicações que não processam dados de forma intensiva. Há porém, um efeito colateral no uso da replicação. As réplicas que são interrompidas ocasionam um desperdício de ciclos de CPU, mesmo que não haja interrupção, replicação leva a desperdício.

5.1.2 O Algoritmo XSufferage

Outra heurística popular para tarefas independentes é chamado *Sufferage* [CLZB00]. A lógica por detrás do *Sufferage* é que uma tarefa deveria ser alocada a uma certa máquina e se isto não acontece, ela vai sofrer mais. Isto é, cada tarefa, possui um valor de sofrimento chamado *sufferage* que é definido pela diferença entre seu melhor *Minimum completion time* (MTC) e o segundo MTC. Tarefas com maior valor de *sufferage* tomam precedência.

XSufferage [CLZB00] é uma extensão de heurística de escalonamento *Sufferage*. A ideia básica da heurística *Sufferage* é determinar quanto cada tarefa seria prejudicada "sofreria" se não fosse escalonada no processador que a executaria de forma mais eficiente. Portanto, o *Sufferage* prioriza as tarefas de acordo com o valor que mede o prejuízo, "sofrimento", de cada tarefa. A principal diferença entre *Sufferage* e *XSufferage* é o método usado para calcular o valor do *sufferage*. No *XSuffrage* o valor do *sufferage* não é calculado só com o **tempo de conclusão mínimo**, mas sim considerando o nível de **tempo de conclusão mínimo** dos aglomerados (*cluster-level MCTs*), ou seja, **tempo de conclusão mínimo** pelo cálculo do mínimo sobre todos as máquinas em cada aglomerado. Assim para cada tarefa é calculado o valor do *sufferage* no aglomerado, esse valor é a diferença entre o melhor e o segundo melhor valor *cluster-level MCTs*.

Portanto o algoritmo de escalonamento *XSufferage* usa informações sobre os níveis do cluster, os quais precisam estar disponíveis no momento em que o algoritmo vai alocar as tarefas. Algumas das informações que ele precisa são:

- disponibilidade de CPU;
- disponibilidade da rede;
- Os tempos de execução das tarefas.

Estas informações devem ser conhecidas antes de escalonamento e são utilizadas para decidir qual tarefa deve ser escalonada em qual processador.

Um ponto importante a ser observado é que o algoritmo considera somente os recursos livres no momento em que vai escalonar uma tarefa, pois caso contrário sempre o recurso mais rápido e com a melhor conexão de rede receberia todas as tarefas.

5.1.3 O Algoritmo Storage Affinity

O algoritmo *Storage Affinity* [SNCBL04] leva em conta o fato dos dados de entrada da aplicação serem freqüentemente reutilizados em execuções sucessivas.

O método de escalonamento é definido sobre o conceito fornecido pelo próprio nome do algoritmo, a **afinidade**, o valor da afinidade entre uma tarefa e uma máquina determina quão próximo da máquina esta tarefa está. A semântica do termo *próximo* está associada à quantidade de bytes da entrada da tarefa que já está armazenada remotamente em uma dada máquina, assim, quanto mais bytes da entrada da tarefa estiver armazenado na máquina, mais próximo a tarefa estará da máquina, pois possui mais *storage affinity*.

Alem de aproveitar a reutilização de dados, o *Storage Affinity*, também realiza replicação de tarefas, tratando da dificultade de obtenção de informações dinâmicas sobre a grade, e sobre o tempo de execução das tarefas. A ideia é que as réplicas tenham a chance de serem submetidas a processadores mais rápidos do que aqueles associados a tarefas originais, reduzindo o tempo de execução da aplicação.

Em Cirne et. al. [SNCBL04] compara os algoritmos *Storage Affinity*, o *XSufferage* e o *WQR*. Nesse mesmo trabalho o algoritmo *XSufferage* e o *Storage Affinity* apresentam melhor desempenho que o *WQR*. O algoritmo *XSufferage* supera ao *Storage Affinity* somente quando a granularidade da aplicação é grande, no entanto, se a granularidade em tipos de aplicações *Bag-of-Tasks* fosse reduzida, o makespan do algoritmo *Storage Affinity* supera ao *XSufferage* em 42%.

5.2 Algoritmos de Escalonamento para Tarefas Dependentes

Os algoritmos de escalonamento mostrados nesta seção, são algoritmos que representam aplicações com tarefas dependentes. As heurísticas neste tipo de aplicações são classificadas em varias categorias, principalmente pelas técnicas que usam. São elas:

- *list-scheduling*. A lista ordenada de tarefas é construída através da atribuição de prioridade para cada tarefa. As tarefas são selecionadas pela ordem de suas prioridades e cada tarefa selecionada é escalonada em um recurso que minimiza uma função custo pré-definido;

- *Clustering*. Esta técnica consiste em criar grupos de tarefas (*clusters* de tarefas) que têm dependência de dados entre si. Essas tarefas são escalonadas no mesmo recurso;

- *duplication-based*. Consiste na duplicação de tarefas em mais de um recurso. Assim cada tarefa pode ser escalonada em dois ou mais recursos distintos.

5.2.1 Problema de Escalonamento para Tarefas Dependentes

Uma aplicação com tarefas dependentes é representada por um grafo acíclico dirigido (DAG), $G = (V, E)$, onde V é o conjunto de tarefas v e E é o conjunto de relações de precedência entre

as tarefas. Cada aresta $(i, j) \in E$ representa uma restrição de precedência tal que, a tarefa t_i deve completar sua execução antes que a tarefa t_j inicie sua execução [THW02].

Num DAG, uma tarefa que não tem pais é chamada *tarefa de entrada* e a tarefa que não tem filhos é chamada *tarefa de saída*. Alguns algoritmos de escalonamento requerem somente uma tarefa de entrada e uma tarefa de saída. No caso de existir mais de uma tarefa de saída, estas serão conectadas com custo de comunicação zero para uma pseudo-tarefa de saída, também com custo zero de computação, para não afetar o escalonamento.

Consideramos um conjunto R de r recursos heterogêneos conectados. *Dados* é uma matriz $v \times v$ de comunicação de dados, onde $dados_{i,k}$ é a quantidade de dados necessária para transmitir desde a tarefa t_i para a tarefa t_k.

W é uma matriz $v \times q$ de custo de computação, na qual cada $w_{i,j}$ dá o tempo de execução estimado para completar a tarefa t_i no recurso r_j.

Antes do escalonamento, as tarefas geralmente são etiquetadas com a média dos seus custos de execução. A média do custo de execução de uma tarefa t_i é apresentada na equação 5.1:

$$\overline{w_i} = \sum_{j=1}^{r} w_{i,j}/r. \tag{5.1}$$

A taxa de transferência de dados entre recursos são armazenados na matriz B de tamanho $r \times r$. O custo de comunicação da aresta (i, k), onde para transferir dados da tarefa t_i (escalonado no recurso r_m) para a tarefa t_k (escalonado no recurso r_n) é definido pela equação 5.2:

$$c_{i,k} = \frac{dados_{i,k}}{B_{m,n}}. \tag{5.2}$$

Quando as tarefas t_i e t_k são escalonados no mesmo recurso, $c_{i,k}$ será zero pois assumimos que o custo da comunicação dentro de um mesmo recurso é desprezível em comparação com o custo da comunicação entre recursos. Antes de escalonar, a média dos custos da comunicação são utilizados para etiquetar as arestas. A média do custo de comunicação de uma aresta (i, k) é definido pela equação 5.3:

$$\overline{c_{i,k}} = \frac{dados_{i,k}}{\overline{B}}. \tag{5.3}$$

Onde, $\overline{B}$ é a taxa media de transferência entre recursos no domínio.

Também, é necessário definir os atributos EST e EFT, que são derivados de um escalonamento parcial. $EST(t_i, r_j)$ e $EFT(t_i, r_j)$ são o *earliest execution start time* e o *earliest execution finish time* das tarefas t_i sobre o recurso r_j, respectivamente. Para a tarefa de entrada $t_{entrada}$,

$$EST(t_{entrada}, r_j) = 0. \tag{5.4}$$

Para as outras tarefas no grafo, os valores do EFT e o EST são calculados recursivamente, começando da tarefa de entrada, como mostrado nas duas seguintes equações 5.5 e 5.6. Para cal-

cular o EFT de uma tarefa t_i todas as tarefas predecessoras da tarefa t_i devem ter sido escalonadas.

$$EST(t_i, r_j) = \max\left\{avail[j], \max_{t_m \in pred(t_i)}(AFT(t_m) + c_{m,i})\right\}, \tag{5.5}$$

$$EFT(t_i, r_j) = w_{i,j} + EST(t_i, r_j), \tag{5.6}$$

onde, $prec(t_i)$ é o conjunto de tarefas predecessoras imediatas da tarefa t_i e $avail[j]$ representa o tempo em que o recurso r_j estará disponível para executar uma tarefa. Se t_k é a última tarefa alocada sobre o recurso r_j, então $avail[j]$ é o tempo que o recurso r_j completou a execução da tarefa t_k e está pronto para executar outra tarefa.

O bloco interno do *max* na equação EST devolve o tempo de disponibilidade (*ready time*), ou seja, o tempo quando todos os dados necessários pela tarefa t_i chegaram ao recurso r_j.

Depois que uma tarefa t_m é escalonada no recurso r_j, o *earliest start time* e o *earliest finish time* de t_m sobre o recurso r_j é igual ao tempo de início atual (*actual start time*) $AST(t_m)$ e o tempo de término atual (*actual finish time*) $AFT(t_m)$ da tarefa t_m, respectivamente. Depois de todas as tarefas do grafo estarem escalonadas, o comprimento do escalonamento (ou seja, o tempo de conclusão total) será o tempo de término da tarefa de saída t_{saida}, ou seja:

$$makespan = AFT(t_{saida}). \tag{5.7}$$

Se a aplicação tem múltiplas tarefas finais e a convenção de inserir uma pseudo-tarefa de saída não é aplicada, o comprimento do escalonamento (que também é chamado *makespan*), é definido como:

$$makespan = \max\{AFT(t_{saida})\}. \tag{5.8}$$

5.2.2 Atributos do Grafo usado pelos Algoritmos de Escalonamento

As tarefas são ordenadas nos algoritmos pelas suas prioridades do escalonamento que são baseados na classificação das variáveis *upward* e *downward*.

A *upward rank* de uma tarefa t_i é recursivamente definido pela equação 5.9:

$$rank_u(t_i) = \overline{w_i} + \max_{t_j \in succ(t_i)}(\overline{c_{i,j}} + rank_u(t_j)). \tag{5.9}$$

Onde, $succ(t_i)$ é o conjunto de sucessores imediatos da tarefa t_i, $\overline{c_{i,j}}$ é o custo de comunicação médio da aresta (i, j), e $\overline{w_i}$ é o custo de computação médio da tarefa t_i. A posição é calculada recursivamente percorrendo o grafo de tarefas de trás para frente, a partir da tarefa de saída, ele é chamado de *upward rank*. Para a tarefa de saída t_{saida}, o valor de *upward rank* é igual a:

$$rank_u(t_{saida}) = \overline{w_{saida}}. \tag{5.10}$$

Basicamente, $rank_u(t_i)$ é o comprimento do "caminho crítico" desde a tarefa t_i para a tarefa

de saída, incluindo o custo de computação da tarefa t_i.

Da mesma forma, o *downward rank* de uma tarefa t_i é recursivamente definido pela equação 5.11:

$$rank_d(t_i) = \max_{t_j \in pred(t_i)} \left\{ rank_d(t_j) + \overline{w_j} + \overline{c_{j,i}} \right\}, \tag{5.11}$$

onde $pred(t_i)$ é o conjunto de tarefas predecessoras imediatas de t_i. Os *downward ranks* são calculados recursivamente percorrendo o grafo de tarefas para frente, começando na tarefa de entrada do grafo. Para a tarefa de entrada $t_{entrada}$, o valor *downward rank* é igual a zero.

Basicamente, $rank_d(t_i)$ é a distância mais longa desde a tarefa de entrada à tarefa de t_i, excluindo o custo de computação da tarefa em si [THW02].

Nesta parte, os algoritmos de escalonamento escolhidos para serem avaliados são:

- HEFT: Este algoritmo usa a técnica de *list scheduling*, é o melhor representante e um dos mais usados pela sua simplicidade na implementação. A partir dele foram feitos diferentes algoritmos de escalonamento para tarefas paralelas;
- CPOP: Também usa a técnica de *list scheduling*. Ele agrupa os nós do caminho crítico em um mesmo recurso;
- PCH: Inicialmente proposto para trabalhar no *middlware* Xavantes [CMB04], usa as técnicas de *list scheduling* e *clustering*.

5.2.3 O Algoritmo HEFT

O algoritmo de escalonamento *Heterogeneous Earliest Finish Time* [THW02] utiliza a técnica de *list scheduling*, para escalonar tarefas dependentes em sistemas heterogêneos.

Este algoritmo possui duas fases, a fase de atribuir prioridade nas tarefas, e a fase de seleção de recursos para as tarefas em ordem não crescente das prioridades e escalonar cada tarefa selecionada no recurso que minimiza o tempo de finalização da tarefa.

Priorização de tarefas Na fase de atribuir prioridade às tarefas é calculada a prioridade das tarefas, baseado na média dos custos de computação e custos de comunicação. Isto é, o cálculo da variável *upward rank*, depois de geradas as prioridades de cada tarefa, é gerada uma lista, baseada na ordenação das prioridades das tarefas em ordem não crescente. Assim, pode ser demonstrado que a ordem não crescente de valores $rank_u$ fornece uma ordem topológica das tarefas, que é uma ordem linear que preserva as restrições de precedência.

Seleção de recursos Na fase de seleção de recursos é considerada a possível alocação de uma tarefa t_i a um recurso r_j que minimize o tempo de término (*earliest finish time*). Aqui o escalonador seleciona a tarefa t_i da lista com maior prioridade, depois para cada recurso $r \in R$ é calculado o EST e EFT de cada tarefa t_i, e finalmente a tarefa é alocada ao recurso r_j que minimiza o EFT da tarefa t_i.

O pseudo-código do algoritmos HEFT é representado no algoritmo 1.

Algorithm 1 O Algoritmo HEFT

1: Calcular o custo de computação das tarefas e o custo de comunicação das arestas com os valores das médias.
2: Calcular $rank_u$ para todas as tarefas percorrendo o grafo, começando na tarefa de saída $task_{saida}$.
3: Ordenar as tarefas em uma lista de escalonamento baseado na ordem não crescente dos valores $rank_u$.
4: **while** (Existem tarefas não escalonadas na lista) **do**
5: Selecionar a primeira tarefa t_i, da lista para o escalonamento.
6: **for** cada recurso r_j no conjunto de recursos ($r_j \in R$) **do**
7: Calcular o valor $EFT(t_i, r_j)$ usando a técnica *scheduling list*.
8: **end for**
9: Alocar a tarefa t_i ao recurso r_j que minimiza o EFT da tarefa t_i.
10: **end while**

5.2.4 O Algoritmo CPOP

O algoritmo de escalonamento *Critical Path On a Processor* [THW02], similar ao algoritmo HEFT, possui as fases de atribuir prioridade às tarefas e seleção de recursos. Ele usa um atributo diferente para estabelecer as prioridades de tarefas e uma estratégia diferente para determinar o melhor recurso para cada tarefa selecionada.

Priorização de tarefas Na fase de atribuir a prioridade às tarefas, tanto a variável *upward* $rank_u$ quanto a variável *downward* $rank_d$, são calculadas para todas as tarefas, usando a média de computação e a média de comunicação.

O algoritmo CPOP usa o caminho crítico (*critical path*) de uma aplicação representada mediante um grafo. O comprimento desse caminho, $|CP|$. A prioridade de cada tarefa é atribuída com a soma de variáveis *upward* e *downward*. O $|CP|$ é igual a prioridade da tarefa entrada. Inicialmente, a tarefa de entrada é a tarefa selecionada e marcada como uma tarefa do caminho crítico. O sucessor imediato (da tarefa selecionada) que tem o valor de maior prioridade é selecionada e é marcada como uma tarefa do caminho crítico. Este processo é repetido até que o nó de saída é atingido. Para desempate o primeiro sucessor imediato que tem a maior prioridade é selecionado.

É mantida uma fila de prioridade (com a chave de $rank_u + rank_d$) para todas as tarefas prontas em qualquer instante. Um heap binário é utilizada para implementar a fila de prioridade, que tem complexidade de tempo $O(log\ v)$ para inserção e exclusão de uma tarefa e $O(1)$ para recuperar a tarefa com a maior prioridade. Em cada etapa, a tarefa com a maior valor $rank_u + rank_d$ é selecionada da fila de prioridade.

Seleção de recursos O recurso de caminho crítico (*critical-path processor*), PCP, é o que minimiza os custos de cálculo cumulativo de tarefas no caminho crítico. Se a tarefa selecionada está no caminho crítico, então é escalonada no recurso de caminho crítico, caso contrário, ela é atribuída a um recurso que minimiza o tempo de terminar mais cedo a execução da tarefa. Em ambos os casos, é considerada uma política de escalonamento baseada em inserção. O algoritmo 2 representa a execução do Algoritmo CPOP.

Algorithm 2 O Algoritmo CPOP

1: Calcular o custo de computação das tarefas e o custo de comunicação das arestas com os valores das médias.
2: Calcular $rank_u$ para todas as tarefas percorrendo o grafo ascendentemente, começando na tarefa saída $task_{saida}$.
3: Calcular $rank_d$ para todas as tarefas percorrendo o grafo descendentemente, começando na tarefa de entrada $task_{entrada}$.
4: Calcular a prioridade $prioridade(t_i) = rank_u(t_i) + rank_d(t_i)$ para cada tarefa t_i no grafo.
5: $|CP| = prioridade(t_{entrada})$, onde $t_{entrada}$ é a tarefa de entrada.
6: $SET_{CP} = \{t_{entrada}\}$, onde SET_{CP} é o conjunto de tarefas no caminho crítico.
7: $t_k \leftarrow t_{entrada}$.
8: **while** t_k não seja a tarefa saída **do**
9: Selecionar t_j, onde $((t_j \in succ(t_k))$ e $(prioridade(t_j) == |CP|))$.
10: $SET_{CP} = SET_{CP} \cup t_j$
11: $t_k \leftarrow t_j$
12: **end while**
13: Selecionar o recurso de caminho crítico (PCP) que minimiza $\sum_{t_i \in SET_{CP}} w_{i,j}, \forall r_j \in R$.
14: Inicia a fila de prioridade com a tarefa de entrada.
15: **while** (Existem tarefas não escalonadas na fila de prioridade) **do**
16: Selecionar a tarefa de maior prioridade t_i, da fila de prioridade.
17: **if** $t_i \in SET_{CP}$ **then**
18: Alocar a tarefa t_i no PCP.
19: **else**
20: Alocar a tarefa t_i no recurso r_j que minimiza o $EFT(t_i, r_j)$.
21: **end if**
22: Atualizar a fila de prioridade com os sucessores da tarefa t_i, se eles se tornam tarefas prontas.
23: **end while**

5.2.5 O Algoritmo PCH

O algoritmo *Path Clustering Heuristic* [BMCB06], é um algoritmo de escalonamento de tarefas dependentes que utiliza as técnicas heurísticas: *list scheduling* e *clustering*. *Cluster* neste contexto é chamado a um agrupamento de tarefas que o algoritmo constrói. As tarefas que fazem parte de um mesmo *cluster* são escalonadas em um mesmo recurso.

Na criação de um *cluster* o algoritmo PCH seleciona um caminho do DAG, fazendo uma busca em profundidade, e escalona seus nós no mesmo recurso, com o objetivo de eliminar comunicação entre nós que têm dependências de dados e entre nós.

A política de criação de *cluster* com tarefas pertencentes a caminhos do grafo tem como objetivo minimizar o *overhead* de comunicação.

Seleção de tarefas e agrupamento Nesta fase o algoritmo seleciona tarefas que formarão cada *cluster* que serão escalonadas no mesmo recurso. A primeira tarefa que compõe um *cluster* cls_k é a tarefa não escalonada com maior prioridade ($rank_u$). A partir dessa tarefa, o algoritmo faz uma busca em profundidade, selecionando novas tarefas para serem adicionadas ao *cluster*. A próxima tarefa selecionada para compor cls_k é a tarefa $t_s \in suc(t_i)$ com o maior $rank_{t_s} + EST_{t_s}$.

Essa soma representa o tamanho do maior caminho que inicia em $t_{entrada}$, passa por t_s e termina em t_{saida}. No caso do PCH o EST é redefinido a seguir:

$$EST(t_i, r_j) = \max\{avail[j], \max_{t_m \in pred(t_i)}(c_{m,i} + w_m + EST_m)\} \tag{5.12}$$

No início do escalonamento a primeira tarefa a ser selecionado será a tarefa de entrada. Após a tarefa de entrada ser selecionada, ela é adicionada ao cluster cls_0, o algoritmo realiza a busca em profundidade, partindo da tarefa de entrada, selecionando a tarefa sucessora t_s que tem o maior $rank_{t_s} + EST_{t_s}$ e adicionando-a ao cluster cls_0. A busca em profundidade continua até que a tarefa de saída seja alcançado. O escalonamento do primeiro *cluster* contém o caminho crítico do grafo inicial. Para formar o próximo *cluster*, o algoritmo seleciona a tarefa t_i não escalonada com maior prioridade, adicionando-a ao *cluster* cls_k. Partindo dessa tarefa o algoritmo efetua uma busca em profundidade de forma análoga à realizada durante a formação do primeiro *cluster*, porém a busca cessa quando atinge uma tarefa sem sucessores não escalonados, incluindo-a no *cluster*. Então o cluster é escalonado e os atributos do grafo recalculados.

O algoritmo 3 mostra essa estratégia, que consome tempo $O(n)$.

Algorithm 3 Gerar_agrupamento

1: $t \Leftarrow$ tarefa não escalonada com a maior prioridade, isto é, maior $rank_u$.
2: $cluster \Leftarrow cluster \cup t$
3: **while** (t tem sucessores não escalonados) **do**
4: $\quad t_{succ} \Leftarrow t_i$ de t com prioridade $rank_{t_i} + EST_{t_i}$.
5: $\quad cluster \Leftarrow cluster \cup t_succ$
6: $\quad t \Leftarrow t_{succ}$
7: **end while**
8: **return** *cluster*

Seleção de recursos A seleção de recursos se dá através do cálculo de valores estimados de início, isto é $rank_u$, EST e EFT. O cálculo em cada recurso tenta prever qual recurso terminará a execução do *cluster* em menor tempo. O fator que determina em qual recurso um *cluster* será escalonado é o EST do sucessor da último tarefa do *cluster* considerado. Um *cluster* cls_k é escalonado no recurso que minimiza o EST do sucessor de cls_k. Para calcular esse valor, o algoritmo primeiro calcula o $EarliestFinishTime(EFT)$ de cada tarefa do *cluster*. É importante salientar que se o recurso contém tarefas do mesmo processo do *cluster* que está sendo escalonado, é necessário antes ordenar as tarefas para que suas precedências não sejam violadas, e então efetuar o cálculo dos $EFTs$. É fácil ver que se as tarefas estiverem ordenadas em ordem decrescente de prioridade, então suas precedências são satisfeitas.

O algoritmo 4 mostra essa estratégia de seleção de recursos.

Path Clustering Heuristic O algoritmo PCH, em [Bit06] e [BMCB06] é composta por um conjunto de fases: seleção de tarefas e agrupamento, seleção de recursos e escalonamento de controladores. A última fase, seleção de controladores acontece no *middleware* chamado *Xavantes*, isto porque o algoritmos foi proposto e testado nesse *middleware*, assim neste cenário, é usada uma

Algorithm 4 Seleciona_melhor_recurso

1: **for all** $r \in recursos$ **do**
2: $escalonamento \Leftarrow Insereclusteremescalonamento$
3: calcula $EST(escalonamento)$
4: $tempo_r \Leftarrow EST(sucessor(cluster))$
5: **end for**
6: **return** $recursorcommenortempo_r$

estrutura chamada de controlador.

Mas, no nosso cenário de simulações vamos usar as fases de seleção de tarefas e agrupamento, e seleção de recursos. Assim, estas fases vão formar o Path Clustering Heuristic (PCH), que é mostrado no algoritmo 5.

O primeiro passo do algoritmo é fazer o cálculo dos atributos iniciais das tarefas. Em seguida o algoritmo inicia a iteração sobre as tarefas do grafo, criando os *clusters* e escalonando-as até que não restem tarefas não escalonadas. A cada *cluster* escalonado, o algoritmo recalcula os atributos, a prioridade ($rank_u$), EST e EFT das tarefas.

Algorithm 5 PCH

1: Atribui o DAG G ao sistema homogêneo virtual
2: Computa os atributos iniciais das tarefas de G
3: **while** existem tarefas não escalonadas **do**
4: $cluster \Leftarrow gera\_agrupamento()$
5: $recurso \Leftarrow seleciona\_melhor\_recurso(cluster)$
6: Escalona $cluster$ em $recurso$
7: Recalcula, prioridade ($rank_u$), $ESTs$ e $EFTs$
8: **end while**

5.3 Apanhado Geral

Neste capítulo, apresentamos os algoritmos de escalonamento para tipos de aplicações para tarefas independentes quanto tarefas dependentes. No caso de aplicações com tarefas independentes, o WQR, XSufferage e o Storage Affinity, esses algoritmos são comparados no trabalho [SNCBL04]. O algoritmo Storage Affinity e o WQR são usado no escalonador OurGrid. Os algoritmos para aplicações com tarefas dependentes, o HEFT, o CPOP e o PCH. Uma avaliação comparativa de 20 diferentes heurísticas pode ser encontrada em [CJSZ08], entre essas heurísticas, a *Heterogeneous Earliest Finish Time* (HEFT), que é uma das mais freqüentemente citadas e utilizadas, tem a vantagem de ser simples e produzir escalonamentos de boa qualidade com um *makespan* menor na maioria dos cenários.

A heurística *Critical Path On a Processor* (CPOP), foi proposta no mesmo artigo que o HEFT [THW02], apresentado bom desempenho, pelo uso de um *heap* no lugar de uma fila de tarefas. Finalmente o *Path Clustering Heuristic* (PCH), foi proposto a partir destes algoritmos. No próximo capítulo, faremos uma comparação de algoritmos para tarefas dependentes usando

workloads reais.

Capítulo 6

Resultados Iniciais

Os *workloads* usados para serem avaliados nos resultados iniciais são de aplicações paralelas (tarefas dependentes), com diferentes quantidades de tarefas, estas aplicações seguem as especificações definidas no artigo [BCD+08]. Os workloads simulados são:

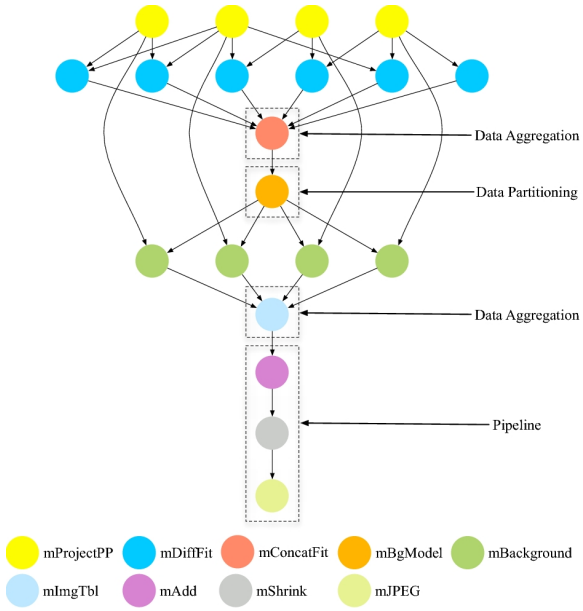
- O *workflow Montage*, criado pela NASA/IPAC, que define pontos de múltiplas imagens de entrada para criar mosáicos personalizados do céu. A estrutura do Montage é mostrada na figura 6.1(a);
- O *workflow* de *CyberShake*, é usado pelo *Southern Calfornia Earthquake Center*, para caracterizar perigos do terremoto na região. A estrutura do CyberShake é mostrada na figura 6.1(b);
- O *workflow* criado pelo *USC Epigenome Center* e o *Pegasus Team*, é usado para automatizar várias operações no processamento da sequência do genoma. A estrutura do Epigenomics é mostrada na figura 6.1(c).

Foram implementados além dos algoritmos de escalonamento HEFT, CPOP e PCH, um escalonamento de tipo FIFO das tarefas, isto é, a medida que percorremos o grafo, começando da tarefa inicial do DAG, até a tarefa final, as tarefas são escalonadas em cada processador. O intuito foi entender a importância de um algoritmo de escalonamento especializado neste tipo de ambientes.

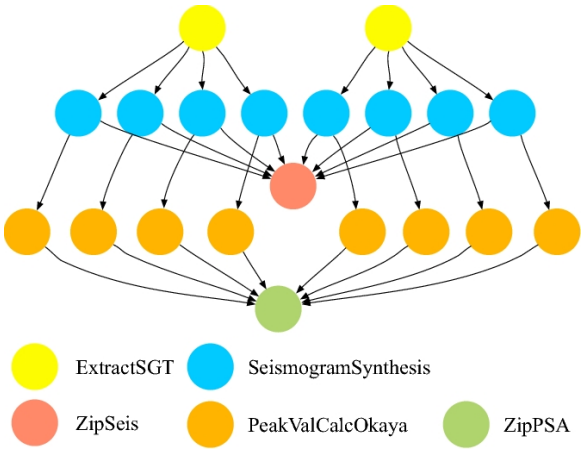
Os algoritmos de escalonamento para tarefas dependentes já comentado no capitulo 5 são simulados no SimGrid. Com o SimDag (módulo do SimGrid) é possível carregar formatos pré-estabelecidos de tipo DAX (*Directed Acyclic graph in XML*) na qual estão especificados os workloads acima comentados. Nesse formato é possível definir um DAG que representará as tarefas e suas dependências, custos de computação e custos de comunicação.

6.1 Descrição dos Cenários

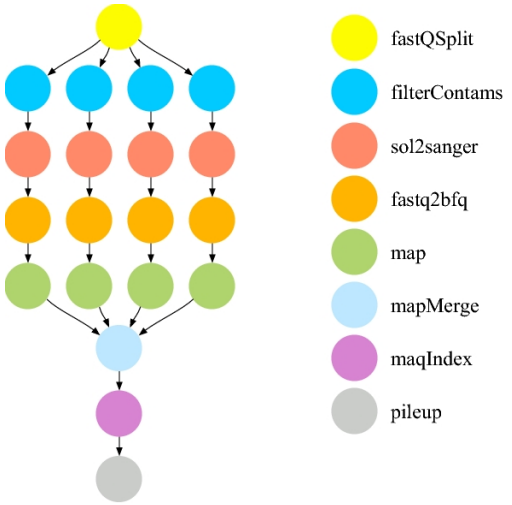
Com o objetivo de pesquisar o dinamismo e a heterogeneidade do ambiente da grade computacional durante o processo de escalonamento, as simulações foram executadas considerando cenários distintos. Primeiro simulamos a heterogeneidade dos workloads, foram avaliadas 20 simulações para os workload de 50 e 1000 tarefas. Também foi avaliada a escalabilidade do *workload*, acrescentando tarefas no workload e entendendo o comportamento dos algoritmos. Depois, a heterogeneidade



(a) Exemplo da estrutura do Montage



(b) Exemplo da estrutura do CyberShake



(c) Exemplo da estrutura do Epigenomics

Figura 6.1: Estrutura dos workloads utilizados

da grade, para comparar o comportamento do algoritmo em cenários onde cada recurso da grade possui diferentes poderes de processamento.

6.1.1 Heterogeneidade dos Tamanhos das Tarefas

Para poder observar o comportamento das heurísticas HEFT, CPOP e PCH, foram testadas 20 simulações por cada número de tarefas, nas quais, em cada simulação o tamanho das tarefas possui uma variação tanto no custo de computação quanto no custo de comunicação, estes workloads foram obtidos do *Workflow Generator* [BCD+08].

Os resultados das simulações no caso de 50 e 1000 tarefas do workload Montage são apresen-

tadas nas figuras 6.2 e 6.3 respectivamente, onde o eixo "Y" representa o *makespan* e o eixo "X", representa o número da simulação das tarefas, neste caso vai desde 1 até o 20.

Na figura 6.2 o algoritmo que apresenta menor instabilidade foi o PCH com um desvio padrão menor com relação ao CPOP e HEFT mostrado na tabela 6.1.

	HEFT	CPOP	PCH
Média	102,89	99,29	87,61
Desvio Padrão	6,23	11,13	1,26

Tabela 6.1: Média e desvio padrão de 20 simulações com 50 tarefas do Montage.

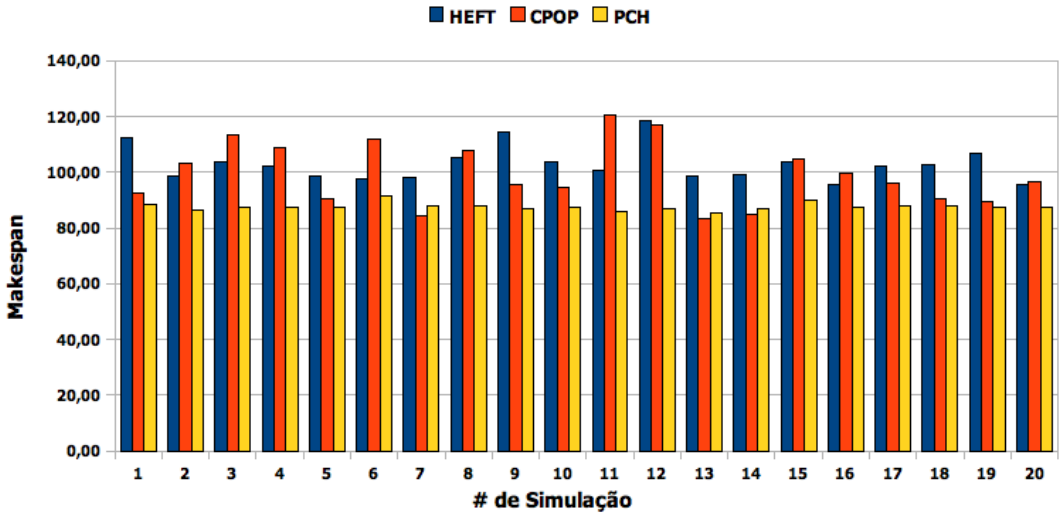


Figura 6.2: Resultados das simulações com 50 tarefas

Na figura 6.3 o algoritmo que apresenta menor instabilidade foi o HEFT com um desvio padrão menor com relação ao PCH e CPOP mostrado na tabela 6.2.

	HEFT	CPOP	PCH
Média	1061,69	1515,22	1721,68
Desvio Padrão	19,10	35,48	158,46

Tabela 6.2: Média e desvio padrão de 20 simulações com 1000 tarefas do Montage.

6.1.2 Escalabilidade do Workload

Para este cenário de simulação foi utilizada uma grade composta por recursos heterogêneos com 10 máquinas apresentadas na tabela 6.3

Foi feita a simulação do workload *Montage*, *CyberShake* e *Epigenomics*, mas com o intuito de testar a escalabilidade, foram simulados workloads com diferente número de tarefas. A simulação começa com 50 tarefas, continuando com 100 tarefas, depois é aumentado de 100 em 100 tarefas,

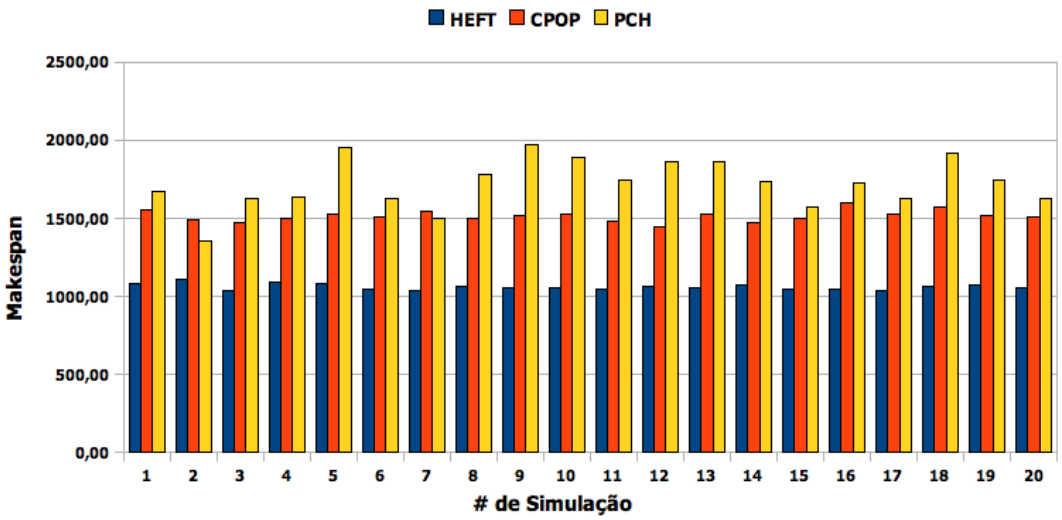


Figura 6.3: Resultados das simulações com 1000 tarefas

Id	Poder Computacional (*flops*)
C1-00	1000000000
C1-01	1000000000
C1-02	1000000000
C1-03	1000000000
C1-04	1000000000
C2-05	5000000000
C2-06	5000000000
C2-07	5000000000
C2-08	5000000000
C2-09	5000000000

Tabela 6.3: Id das máquinas, poder computacional de cada uma.

até chegar a 1000 tarefas. Em cada número de tarefas foram feitas 20 simulações e foi tomado como referência a média dos resultados dessas simulações.

Na figura 6.4 mostra como as heurísticas HEFT e CPOP apresentam uma crescimento uniforme, sendo a HEFT que oferece melhor desempenho a respeito das outras duas, também é possível entender que o uso de um algoritmo de escalonamento especializado melhora significativamente o escalonamento, pois o escalonamento simples de tipo FIFO apresenta um escalonamento com *makespan* muito alto. En todas as figuras podemos ver também o desvio padrão de cada média.

Os resultados de simular o workload *Cybershake* são apresentados na figura 6.5. Neste caso a heurística HEFT não apresenta bom desempenho, o PCH e o CPOP apresentam melhoria enquanto o número de tarefas é acrescentado.

Na figura 6.6, as três heurísticas também apresentaram picos. Aqui o HEFT também mostra melhor desempenho com respeito às outras duas heurísticas.

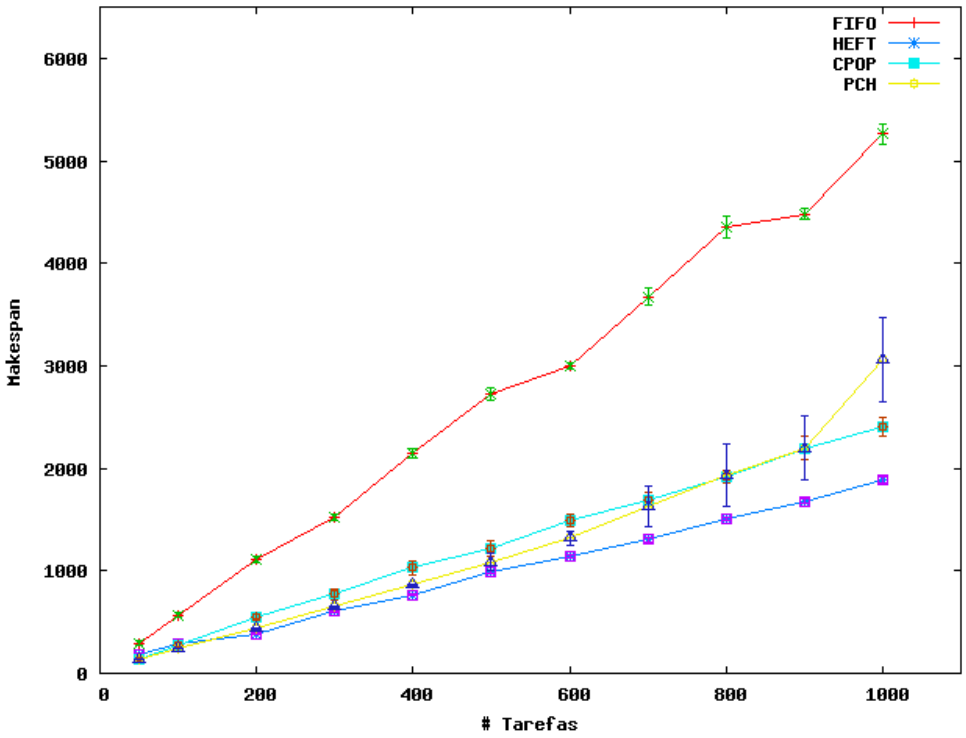


Figura 6.4: Escalabilidade do Workload Montage

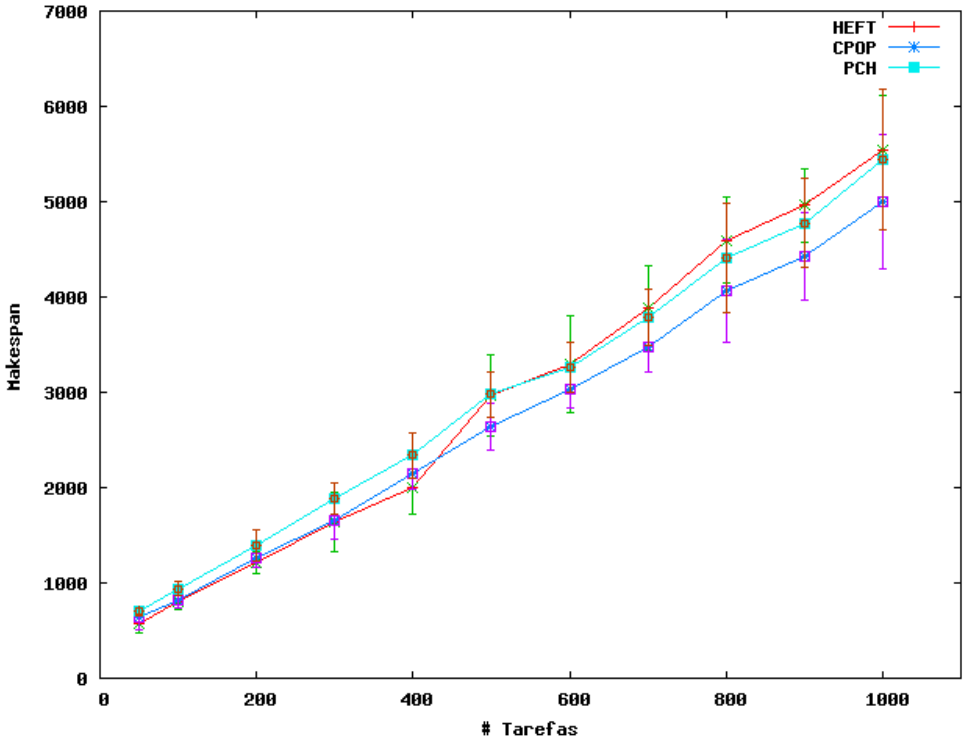


Figura 6.5: Escalabilidade do Workload Cybershake

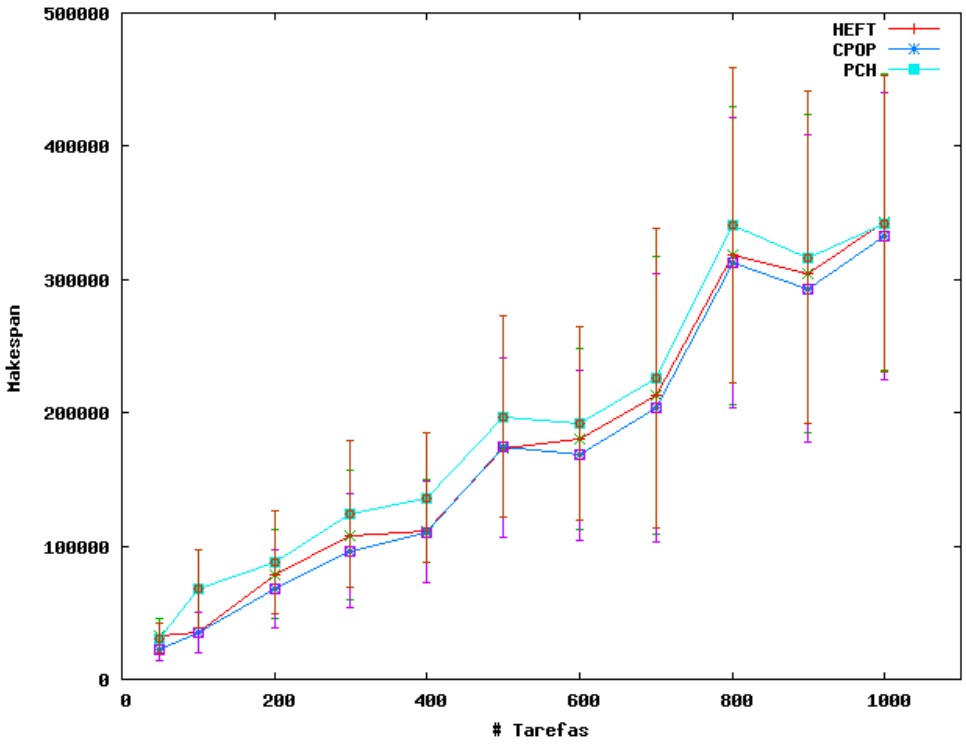


Figura 6.6: Escalabilidade do Workload Epigenomics

6.1.3 Heterogeneidade da Grade

Todos os resultados apresentados na seção anterior foram gerados sobre a plataforma apresentada na tabela 6.3. Vamos mudar os valores de processamento dos recursos para entender o comportamento das heurísticas no caso de uma maior heterogeneidade. Na tabela 6.4

Id	Poder Computacional (*flops*)
C1-00	1000000000
C1-01	2000000000
C1-02	3000000000
C1-03	4000000000
C1-04	5000000000
C2-05	6000000000
C2-06	7000000000
C2-07	8000000000
C2-08	9000000000
C2-09	9000000000

Tabela 6.4: Id das máquinas, poder computacional de cada uma.

A figura 6.7 apresenta o resultado da simulação do workload Montage usando a os recursos da tabela 6.4. Comparando esse resultado com o resultado da figura 6.4 é possível observar que a heurística HEFT em ambas figuras apresenta melhor desempenho a respeito das outras heurísticas.

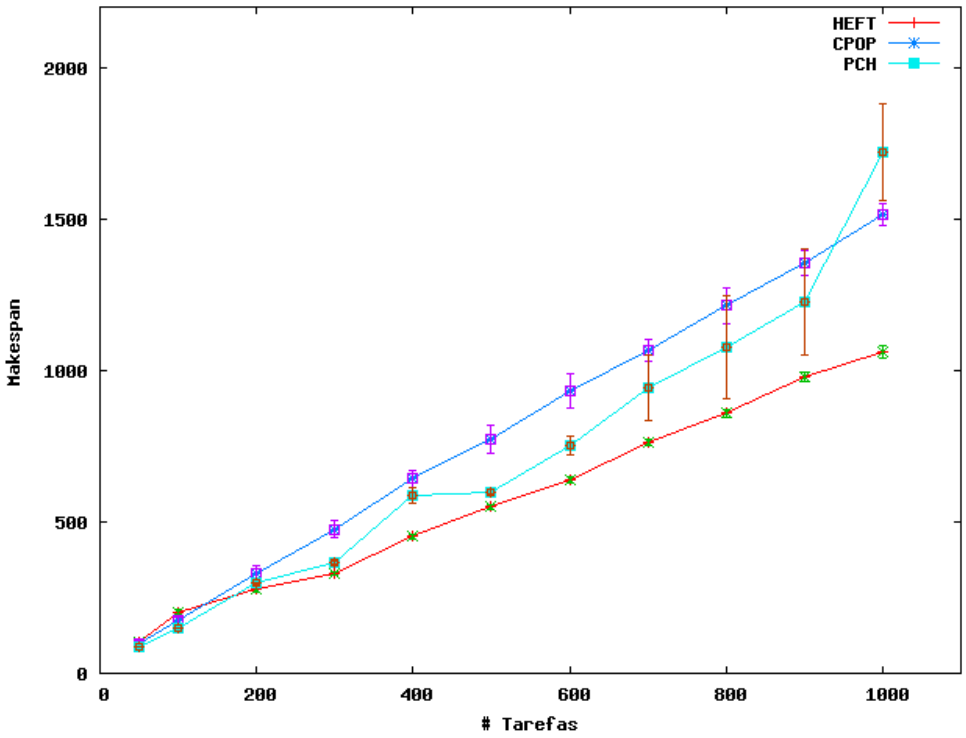


Figura 6.7: Resultados do workload Montage sobre a plataforma da tabela 6.4

Na figura 6.8, as três heurísticas apresentam uniformidade, isto a respeito da figura 6.5, mas o comportamento do HEFT, melhora consideravelmente. O PCH apresentam um desempenho melhor em comparação com o CPOP. Enquanto o número de tarefas é aumentado o desempenho da heurística CPOP não oferece um bom desempenho.

Na figura 6.9, em comparação com a figura 6.6, novamente o HEFT oferece um desempenho bom, melhorando ao PCH e o CPOP. Mesmo com os picos das gráficas, esse desempenho se mantém.

Neste capítulo foram apresentadas as resultados iniciais das simulações realizadas das heurísticas HEFT, CPOP e PCH com o simulador SimGrid. Avaliando a heterogeneidade nos tamanhos das tarefas, a heterogeneidade da grade e escalabilidade do workload.

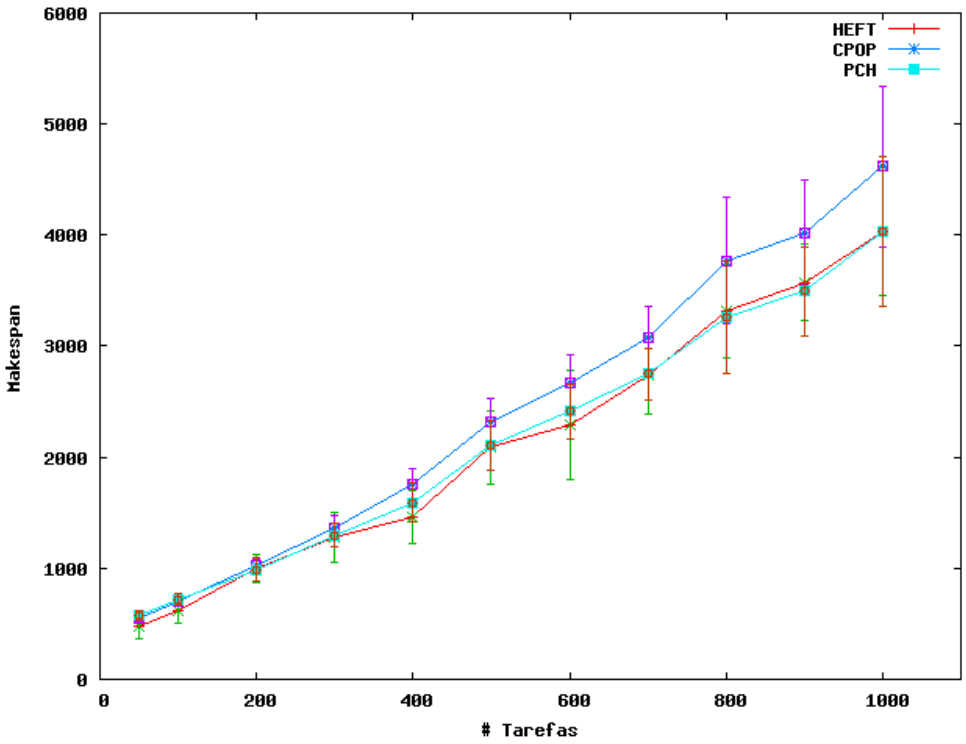


Figura 6.8: Resultados do workload CyberShake sobre a plataforma da tabela 6.4

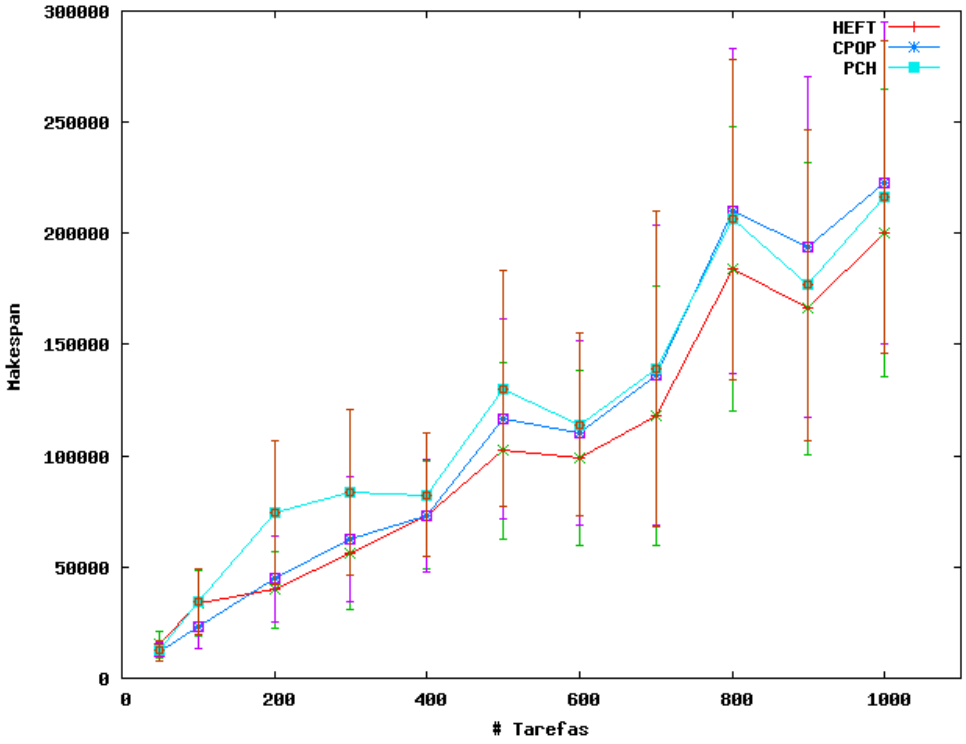


Figura 6.9: Resultados do workload Epigemonics sobre a plataforma da tabela 6.4

Capítulo 7

Plano de Trabalho e Cronograma

7.1 Plano de Trabalho e Cronograma

As atividades descritas e as que são almejadas para o presente trabalho estão na tabela 7.1.

Atividades	**Anos e Semestres**								
	2009		**2010-2011**						
	1^o	2^o	1^o	2^o	Nov.	Dez.	Jan.	Fev.	Mar.
Disciplinas obrigatórias	x	x	x						
Levantamento Bibliográfico	x	x	x						
Estudo dos diversos escalonadores		x	x						
Análise de ambientes de simulação		x	x						
Comparação dos escalonadores					x	x			
Implementação dos algoritmos				x	x	x			
Estudo de métricas de comparação					x	x			
Suporte para mais workloads						x	x	x	
Estudo com workloads reais						x	x	x	
Artigos					x	x	x	x	
Redação da dissertação e defesa								x	x

Tabela 7.1: Cronograma de atividades

Capítulo 8

Conclusões

8.1 Considerações Finais

A computação em grade é uma alternativa para a execução de aplicações paralelas ou distribuídas que demandam alto poder computacional. Essas aplicações são compostas por diversas tarefas que, a depender do tipo de aplicação, podem se comunicar durante a fase de execução. No escopo da computação em grade, o escalonamento é um grande desafio, para atingir um bom desempenho no tempo de execução de aplicações. O problema de escalonamento é um problema NP-Completo [Pin08].

Assim, neste estudo, são avaliados os algoritmos de escalonamento para grades computacionais. o *Path Clustering Heuristic* (PCH) [BM06], o *Critical Path on a Processor* (CPOP) [THW02] e o *Heterogeneuos Earliest Finish Time* (HEFT) [THW02]. Alem deles foi implementado um escalonamento simples de tipo FIFO no qual as tarefas são escalonadas em todos os recursos disponíveis. Foram feitos experimentos em três cenários.

A heurística HEFT apresenta bom desempenho a medida que o número de tarefas foi acrescentado, tanto o PCH quanto o CPOP não apresentaram bom desempenho com relação ao HEFT.

O uso de um algoritmo de escalonamento especializado, é fundamental para obter um "bom" desempenho no escalonamento, pois como foi apresentado no capitulo 6 o escalonamento simples apresenta um desempenho muito ruim em comparação com o escalonamento gerado pelos algoritmos HEFT, COPO e PCH. A medida que o número de tarefas é acrescentada o *makespan* cresce consideravelmente.

Referências Bibliográficas

[Bat10] Daniel Macêdo Batista. *Escalonamento de Tarefas Dependentes para Grades Robustos às Incertezas das Informações de Entrada.* PhD thesis, Unicamp - Universidade Estadual de Campinas. Instituto de Computação, Campinas, São Paulo, Brasil, Fevereiro 2010.

[BCC+03] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger, and Floriano Zini. Optorsim - a grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, pages 403–416, 2003.

[BCD+08] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. *The 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS08), in conjunction with Supercomputing (SC08) Conference.*, November 2008.

[BFH03] Fran Berman, Geoffrey Fox, and Tony Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, New York, NY, USA, first edition, 2003.

[BHK+00] Brett Bode, David M. Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 27–27, 2000.

[Bit06] Luiz Fernando Bittencourt. Uma heuristica de agrupamento de caminhos para escalonamento de tarefas em grades computacionais. Master's thesis, Unicamp - Universidade Estadual de Campinas. Instituto de Computação, Campinas, São Paulo, Brasil, Março 2006.

[BM02] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience (CCPE*, 14(13):1175–1220, 2002.

[BM06] Luiz F. Bittencourt and Edmundo R. M. Madeira. A dynamic approach for scheduling dependent tasks on the Xavantes grid middleware. In *MCG '06: Proceedings of the 4th international workshop on Middleware for grid computing.* ACM, 2006.

[BMCB06] Luiz F. Bittencourt, Edmundo R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. Uma heuristica de agrupamento de caminhos para escalonamento de tarefas em grades computacionais. In *Anais SBRC 2006*, pages 83–98, Curitiba, Brazil, 2006.

[CBA+06] Waldredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro B. Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[CDCG+05] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2, pages 776–783, 2005.

[CHR09] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 91–100, 2009.

[CJSZ08] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. pages 63–74, 2008.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*. IEEE Computer Society Press, March 2008.

[CLZB00] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 349–363, 2000.

[CMB04] Fábio R. L. Cicerre, Edmundo R. M. Madeira, and Luiz E. Buzato. A hierarchical process execution support for grid computing. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 87–92. ACM, 2004.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício A. B. Silva, Carla O. Barros, and Cirano Silveira. Running Bag-of-Tasks applications on computational grids: the Mygrid approach. *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 0:407–416, 2003.

[CWT+04] Clovis Chapman, Paul Wilson, Todd Tannenbaum, Matthew Farrellee, Miron Livny, John Brodholt, and Wolfgang Emmerich. Condor services for the global grid: interoperability between Condor and OGSA. In *Proceedings of 2004 UK e-Science All Hands Meeting*, pages 870–877, Nottingham, UK, August 2004.

[DA06] Fangpeng Dong and Selim G. Akl. Scheduling algorithms for grid computing: State of art and open problems. Technical report, School of Computing, Queen's University, Kingston, Ontario, 2006.

[Dan05] Mario Dantas. *ComputaÃ§ao Distribuida De Alto Desempenho. Redes, Clusters E Grids Computacionais*. Axcel Books, Rio de Janeiro, RJ, Brasil, first edition, 2005.

[dSCB03] Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Euro-Par*, pages 169–180, 2003.

[FK04] Ian Foster and Carl Kesselman. *Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, second edition, 2004.

[FKNT02] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002.

[FKT01] Ian T. Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid - enabling scalable virtual organizations. *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 6–7, 2001.

[Fos05] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *NPC 2005: IFIP International Conf. on Network and Parallel Computing*, pages 2–13, Beijing, China, 2005.

[FQS08] Marc-Eduard Frincu, Martin Quinson, and Frédéric Suter. Handling very large platforms with the new simgrid platform description formalism. Technical Report 0348, Institut National de Recherche en Informatique et en Automatique, INRIA, France, 2008.

[FTF+02] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency - Practice and Experience*, 16(5):449–459, 2004.

[GW97] Andrew S. Grimshaw and Wm. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of ACM*, 40(1):39–45, 1997.

[Hen95] Robert L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, 1995.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[Pin08] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, New York, NY, USA, third edition, 2008.

[RLS98] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140–146, July 1998.

[SNCBL04] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, and Aliandro Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 210–232, 2004.

[Sta06] Garrick Staples. Torque resource manager. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006.

[THW02] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[TMN+99] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 97–104, 1999.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[WET03] Gropp William, Lusk Ewing, and Sterling Thomas. *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge , Massachusetts London, England, second edition, 2003.

Índice Remissivo