UNIVERSIDADE FEDERAL FLUMINENSE

CARLOS EDUARDO CABRAL DA CUNHA

OSCAR-PBS: NOVAS FUNCIONALIDADES PARA UM GERENCIADOR E ESCALONADOR DE JOBS PARA CLUSTERS

NITERÓI

2013

UNIVERSIDADE FEDERAL FLUMINENSE

CARLOS EDUARDO CABRAL DA CUNHA

OSCAR-PBS: NOVAS FUNCIONALIDADES PARA UM GERENCIADOR E ESCALONADOR DE JOBS PARA CLUSTERS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: REDES E SISTEMAS DISTRIBUÍDOS E PARALELOS.

Orientador:
LÚCIA MARIA DE ASSUMPÇÃO DRUMMOND

Co-orientador:
CRISTIANA BARBOSA BENTES

NITERÓI

2013

CARLOS EDUARDO CABRAL DA CUNHA

OSCAR-PBS: NOVAS FUNCIONALIDADES PARA UM GERENCIADOR E ESCALONADOR DE JOBS PARA CLUSTERS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: REDES E SISTEMAS DISTRIBUÍDOS E PARALELOS.

Aprovada em _____ de 2013.

BANCA EXAMINADORA

Prof. Lúcia Maria de Assumpção Drummond / Orientadora, UFF

Prof. Cristiana Barbosa Bentes / Orientadora, UERJ

Prof. Ricardo Cordeiro de Farias / UFRJ

Prof. Eugene Francis Vinod Rebello / UFF

Niterói

2013

Aos meus pais.

Agradecimentos

Agradeço primeiramente aos mistérios da vida e do universo, principalmente as que permitiram a nossa existência e a de tantas outras coisas maravilhosas.

Agradeço as minhas orientadoras Lúcia e Cristiana. A Lúcia agradeço por tudo que me ensinou desde a minha graduação, depois ao me incentivar a ingressar no mestrado e por toda sua dedicação e paciência. A Cristiana agradeço por toda ajuda, por tudo o que me ensinou durante o desenvolvimento da dissertação e pela maneira paciente com que transmite esses ensinamentos.

Aos meus pais, agradeço a minha mãe Sandra pela infinita dedicação e ao meu pai Eduardo, que sei que está me apoiando de onde estiver. Agradeço aos meus familiares por compreenderem minha ausência e pelo apelido de ufólogo que me deram por causa dos vários finais de semana em que eu vim para a UFF.

Gostaria de agradecer as pessoas que me apoiaram durante o desenvolvimento da dissertação, em especial a Patricia ao Júlio e a Ariane e a todas as pessoas que me ajudaram e que compreenderam a minha ausência durante o desenvolvimento desse trabalho.

Resumo

Esse trabalho apresenta um escalonador de *jobs*, ou *batch scheduler*, chamado de OscarPBS para *clusters* de máquinas *multicore*. Um *batch scheduler* tem como objetivo determinar onde e quando aplicações sequenciais ou paralelas submetidas por usuários de um *cluster* podem ser executadas. A principal motivação para o desenvolvimento deste escalonador foi a de melhorar um Gerenciador de Recursos Distribuídos conhecido como TORQUE, dando a ele novas funcionalidades. O TORQUE gerencia *jobs*, mas seu escalonador possui muitas limitações. Nesse contexto, o OscarPBS substitui o escalonador padrão do TORQUE, adicionando a ele novas rotinas que alocam os *jobs* em recursos do *cluster* que não seriam utilizados pelo escalonador padrão do TORQUE. O OscarPBS também adiciona novas funcionalidades normalmente encontradas em escalonadores de versões comerciais de Gerenciadores de Recursos Distribuídos, tais como *Backfilling* e suporte a qualidade de serviço para *jobs* com prazo e *jobs* com urgência. Esse trabalho também mostra testes experimentais, tendo como objetivo comparar o OscarPBS com o escalonador padrão do TORQUE, utilizando métricas normalmente adotadas na avaliação de escalonadores para *jobs*. O OscarPBS apresenta resultados melhores do que o escalonador padrão do TORQUE em todos os experimentos.

Abstract

This work presents a job scheduler, or batch scheduler, for multicore clusters, called here OscarPBS. The main motivation for developing this scheduler was to enhance the open source Distributed Resource Manager TORQUE, by adding new functionalities to it. TORQUE manages jobs but its scheduler has many limitations. In this context, OscarPBS replaces the standard scheduler of TORQUE, bringing with it new procedures that assign jobs to cluster´s resources that are not available to the standard scheduler of TORQUE. OscarPBS also adds to TORQUE new functions, usually found in more efficient schedulers present in commercial versions of Distributed Resource Managers, such as Backfilling and QoS support for jobs with deadlines and emergency jobs. This work also shows experimental tests aiming to compare OscarPBS with the standard scheduler of TORQUE, using the usual metrics to evaluate job schedulers. OscarPBS outperformed the standard scheduler of TORQUE in all experiments.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A presença cada vez mais comum de *clusters* de máquinas *multicore* e a utilização de aplicações paralelas por mais usuários tem tornado necessário administrar esses *clusters* com softwares de gerência de *jobs*. Esses softwares podem garantir que os usuários obtenham uso exclusivo durante um certo período de tempo de recursos que necessitem, possibilitar políticas de utilização controladas automaticamente, com parâmetros configurados por um usuário administrador [53]. Além disso devem permitir uma utilização eficiente dos recursos, combinando políticas orientadas a satisfação do usuário e administradores e métodos de alocação orientados ao desempenho do sistema de maneira equilibrada.

Há uma série de softwares de gerência de recursos comerciais [1, 16, 14] e muitos também de domínio público [2, 3, 45, 10, 21]. Cada um destes softwares provê uma gama de funcionalidades aos usuários e muitos deles fornecem escalonadores de *jobs*. A função de escalonamento dentro de um sistema de gerência de *jobs* é crucial para garantir uma boa utilização dos recursos do sistema. Entretanto, dependendo das políticas de escalonamento empregadas, muitas vezes as necessidades dos *jobs* dos usuários podem não ser atendidas de forma satisfatória. Por exemplo, um dos escalonadores comerciais mais populares para *clusters*, o PBS Professional [16], em sua política de escalonamento, faz uma verificação muito limitada quanto os possíveis encaixes de um *job multicore* em recursos que estejam disponíveis, fazendo quase sempre uma escolha gulosa de escalonamento que poderia ser melhorada com outros métodos. Já o escalonador TORQUE [2], um escalonador de código fonte aberto e disponível para uso gratuito, não tem suporte a execução adiantada de *jobs*, conhecido como *backfilling* [52], que poderiam ser alocados em recursos livres enquanto existem outros *jobs* que estão esperando a muito tempo para serem executados, num estado conhecido como *starve* ou *starving jobs* [43].

Vários softwares de gerenciamento e de escalonamento de código aberto como TORQUE,

o SLURM e o MAUI [2, 40, 3] possibilitam a inclusão de funcionalidades ou de módulos de escalonamento mais sofisticados ou que atendam a necessidades específicas da instituição que utiliza tais escalonadores, de modo que um estudo sobre o funcionamento de tais softwares e de métodos de escalonamento se torna interessante.

Diante deste cenário, neste trabalho estudamos as deficiências do escalonadores descendentes do OpenPBS [35], os quais serão tratados apenas como PBS e propomos a inclusão de novas funcionalidades para atender demandas de *jobs* com requisitos de Qualidade de Serviço ou de alta prioridade. Nossa ideia é modificar as políticas existentes no PBS para o tratamento das questões de *jobs* com prazo de execução e de *jobs* com urgência e grande prioridade, sem deixar de considerar *jobs* convencionais, oferecendo a esses *jobs* boas possibilidades de competir com *jobs* com prazo. Utilizaremos para isso métodos como *backfilling* [52, 28] e ao mesmo tempo trataremos *jobs* convencionais que eventualmente entrem em *starve*, dando a esses últimos uma prioridade diferenciada. Também propomos alocar todos esses tipos de *jobs* utilizando novos métodos que sejam mais eficientes do que métodos que se baseiem apenas em listas ordenadas, como acontece nas versões de PBS estudadas nesse trabalho.

Realizamos um estudo inicial sobre como o PBS trata estas questões e consideramos que as políticas utilizadas geram uma baixa utilização dos recursos do *cluster*.

Quanto aos *jobs* convencionais, é possível verificar que em várias ocasiões o PBS deixa de fazer um escalonamento eficiente devido a forma muito simplificada como escolhe um *job* de uma fila de *jobs* pendentes e aloca o primeiro e coloca em execução sem considerar as necessidades de outros *jobs*. Por exemplo, se um *job* que não faz questão de nenhuma máquina especial, for escolhido primeiro porque tinha prioridade sobre os demais, o PBS pode alocar esse *job* em uma máquina que é indispensável para o *job* seguinte da fila, mesmo que existam outras máquinas livres. O escalonador simplesmente ordena e coloca os *jobs* em execução sem considerar combinações mais favoráveis a todos os *jobs* da fila. De maneira similar, quando um *job* entra em *starve* por ter esperado muito na fila, o PBS passa a conter recursos indiscriminadamente até que o primeiro *job* em *starve* possa entrar em execução. Tal retenção de recursos não considera que o *job* possa estar requisitando máquinas especificas e que os recursos que o escalonador retém nunca serão utilizados por aquele *job*. O escalonador também não verifica que enquanto um determinado recurso não pode ser liberado, um outro recurso livre poderia ser utilizado por um *job* de tamanho menor sem atrasar o início do *job* em *starve*. O PBS também não disponibiliza a possibilidade de um usuário submeter um *job* com prazo limite, ou

seja, um *job* com características de qualidade de serviço quanto a ter garantias sobre seu tempo de finalização.

Nossa proposta é implementar o escalonador o qual demos o nome de OscarPBS e que é baseado no PBS, porém com novas funcionalidades. Uma delas é sempre tratar todos os *jobs* na fila, verificando as possibilidades de encaixe por meio de um método chamado *Empurrar*. Este método permite a realocação de *jobs* em máquinas diferentes daquelas onde os *jobs* estavam alocados originalmente. Isso resolve problemas sobre a questão de um *job* ocupar uma máquina que seria fundamental para outro *job*. Quanto aos *jobs* em *starve*, o escalonador proposto aloca esses *jobs* antes dos *jobs* convencionais, mas permite a *jobs* convencionais serem alocados em regiões livres, mesmo que isso faça com que eles sejam executados antes dos *jobs* em *starve*. Os *jobs* convencionais também podem empurrar *jobs* em *starve* já alocados, durante suas tentativas de encaixe. *Jobs* com qualidade de serviço, ou *job* QoS, [38] seriam implementados, de modo a ter alta prioridade dentro do espaço de tempo entre o instante da submissão e o prazo do *job*. Tal *job* QoS pode ocasionalmente desalocar *jobs* em *starve* desde que o *job* em *starve* não tenha sido desalocado até uma quantidade limite. *Jobs* QoS também teriam prioridade maior que qualquer *job* não QoS, mesmo aqueles que estivessem em *starve* e podem empurrar qualquer *job* alocado. Uma outra funcionalidade do OscarPBS seria de permitir o tratamento de *jobs* de prioridade muito alta, chamados aqui de *jobs* de emergência. Tais *jobs* teriam caráter de grande urgência e o escalonador permite ao administrador que a submeteria com um prazo definido, enquanto e escalonador tentaria fazer o melhor escalonamento possível considerandos todos os *jobs* e sem deixar de alocar o *job* de emergência. O *job* de emergência teria o poder de desalocar ou parar qualquer outro *job* em execução, mas o escalonador tentaria evitar tais ações. Em caso de precisar desalocar algum *job* para liberar recursos para um *job* emergência, o escalonador consideraria os *jobs* menos importantes antes de desalocar algum deles.

Realizamos experimentos comparando OscarPBS com o PBS em um cenário de submissão de *jobs* baseado em um histórico de submissões reais, onde o tempo entre as submissões inicialmente é escolhido como um tempo fixo e depois seguindo uma distribuição exponencial. Outros testes foram realizados com um tempo entre submissões de *job* obedecendo a uma distribuição exponencial com uma média de tempo obtida a partir do histórico de *jobs* reais. Os experimentos foram realizados em diversas situações considerando a possibilidade de tratar ou não os *jobs* em *starve*. Também realizamos experimentos para comparar os escalonadores OscarPBS com o PBS onde alguns *jobs* possuíam prazo, que chamamos de *jobs* QoS e experimentos onde estavam presentes *jobs*

de alta prioridade.

Nossos resultados mostraram que o OscarPBS teve melhor desempenho que o escalonador PBS quando utilizados cenários de submissão com tempo fixo e exponencial no caso de considerar os não *jobs* em *starve*. Um melhor desempenho também foi observado nos experimentos com *jobs* QoS e nos com *jobs* de alta prioridade.

Concluímos que o estudo e implementação do escalonador OscarPBS trouxe grande conhecimento sobre o funcionamento do gerenciador de recursos do PBS e o escalonador. Os novos métodos propostos para esse escalonador conseguiram melhorar de forma bastante significativa o escalonamento dos *jobs*. Também percebemos novas funcionalidades podem ser acrescentadas com facilidade, de modo a atender a necessidade dos usuários do *cluster* e que também é possível criar uma interface para controle das submissões dos *jobs* dos usuários baseada em um banco de dados, o que facilitaria a criação de uma interface *web* para a submissão e acompanhamento de *jobs*, o que também motivaria uma maior utilização do *cluster* por parte dos usuários.

De maneira resumida, os objetivos desse trabalho, foram o de implementar e testar algumas funcionalidades encontradas em escalonadores, comerciais e de domínio público, em um novo escalonador chamado de OscarPBS. O novo escalonador foi desenvolvido para ser acrescentado ao TORQUE com o objetivo de ser utilizado na submissão de *jobs* de usuários em um *cluster* real. Além disso, nesse trabalho foi implementado um método de escalonamento chamado de *Empurrar* que consiste em realocar *jobs* em máquinas de uma maneira que não é encontrada em escalonadores da família PBS e que tem o objetivo de melhorar o desempenho do OscarPBS em relação ao escalonadores da família PBS.

Dentre as contribuições esperadas, o trabalho desenvolvido pode permitir a utilização do novo escalonador no *cluster* Oscar [11], obtendo possivelmente uma melhor utilização desse *cluster* do que seria possível com o uso de escalonadores disponíveis da família PBS. O trabalho também deve permitir, devido a compreensão que foi adquirida no estudo e implementação de um escalonador utilizado em um *cluster* real, a implementação, teste e utilização de novos métodos de escalonamento. Esses métodos podem a vir ser usados para a submissão de *jobs* reais no *cluster* Oscar.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 apresenta os trabalhos relacionados, considerando outros softwares de gerenciamento de recursos distribuídos (D-RMS) e softwares de escalonamento além do PBS. O Capítulo 3 trás maiores detalhes sobre o funcionamento do próprio PBS e explicações de outros softwares baseados em PBS. O Capítulo 4 apresenta um histórico sobre o desenvolvimento do escalonador

OscarPBS, explica as metodologias de escalonamento em que ele se baseia, o método *Empurrar* e apresenta o algoritmo do escalonador OscarPBS e suas rotinas principais. O Capítulo 5 descreve resultados da execução do OscarPBS com um lote de *jobs* sintéticos e compara o desempenho do OscarPBS com o escalonador PBS convencional. O Capítulo 6 mostra as conclusões do trabalho.

Capítulo 2

Definição do Problema

2.1 Definição de termos utilizados

Neste trabalho tratamos do problema de gerenciamento e escalonamento de *jobs* de aplicações sequenciais e paralelas em um *cluster* de máquinas *multicore*. A seguir apresentamos a definição de alguns termos fundamentais para o entendimento do problema e da solução apresentada.

- **Administrador:** Pessoa que pode submeter *jobs* no *cluster* e que possui poder de administração sobre todos os *jobs* do *cluster*. Ele pode submeter qualquer tipo de *job*.
- **Alocar:** Reservar recursos solicitados por um *job*, associando os recursos a máquinas do *cluster*.
- **Aplicação paralela:** Aplicação que utiliza mais de uma *CPU*, envolvendo memória compartilhada e/ou troca de mensagens pela rede. Aplicações tipo *mpi* ou *multi-thread*.
- **Aplicação sequencial:** Aplicação que utiliza uma única *CPU*.
- **Cluster:** Conjunto de máquinas *multicore* com Hardware e Software razoavelmente uniformes, conectadas em uma mesma rede local e possuindo um mesmo conjunto de usuários e de diretórios de rede compartilhados.
- **CPU:** Uma *CPU* representa um *core* ou núcleo de um processador *multicore*.
- **D-RMS:** Software de submissão de *jobs* em *cluster*. Envolve gerenciamento, escalonamento, controle e acompanhamento dos *jobs*.

- **Empurrar:** Ato de mover fragmentos de *jobs* alocados em máquinas para outras máquinas.
- **Escalonador:** Módulo do *D-RMS* responsável pelo gerenciamento de *jobs*.
- **Executar:** Colocar o *script* de um *job* em execução nas máquinas alocadas para ele no *cluster*.
- **Fragmento de *job*:** Partição do conjunto de *CPUs* de um *job* que devem caber completamente e ocupar uma única máquina do *cluster*. Apesar de um fragmento de *job* poder ocupar uma única máquina, uma máquina pode comportar mais de um fragmento.
- **Job:** Aplicação de um usuário, sequencial ou paralela, que pode ser executada a partir de um conjunto de comandos (script) em um conjunto de recursos do *cluster* solicitados pelo usuário. A esse conjunto de recursos, o usuário deve incluir um tempo máximo estimado para a execução de sua aplicação (*walltime*) e um conjunto de uma ou mais CPUs que devem estar agrupadas em partições chamadas de fragmentos de *jobs*. O *job* também precisa ter seu tipo especificado pelo usuário ou administrador, podendo ser classificado como Comum, *QoS* ou de Emergência. O *job* ainda pode ter especificado um prazo de término e, caso seja um *job* de Emergência, também pode ter valores booleanos que indicam poderes que ele possui para desalocar outros *jobs*.
- **Jobs Comuns:** *Jobs* sem restrição de prazo.
- **Jobs de Emergência:** *Jobs* com prazo, submetidos pelo administrador e que podem ter poderes para desalocar outros *jobs*.
- **Jobs QoS:** *Jobs* que possuem restrição de prazo e prioridade sobre *jobs* Comuns.
- **Jobs Starve:** *Jobs* que inicialmente eram comuns, mas por demorarem demais a serem alocados passam a ter prioridade maior que os *jobs* comuns e obedecer a uma política *FIFO*.
- **Lista:** Estrutura de dados que armazena *jobs*.
- **Máquina:** Máquina ou nó de computação, define um computador *multicore* do *cluster*. Cada computador possui oito *CPUs* ou *cores*.
- **Requisição:** Requisição ou submissão de *job*, ato de um usuário ou administrador submeter um *job* para o *D-RMS*.

- **Script:** Sequencia de comandos de terminal que representam a aplicação sequencial ou paralela de um usuário.

- **Usuário:** Pessoa que pode submeter *jobs* no *cluster*, possuindo poderes de administração limitados a seus próprios *jobs*. Pode submeter qualquer tipo de *job*, menos os de Emergência.

2.2 Características do Problema

De forma simplificada, o problema pode ser definido pelo seguinte conjunto de características:

- Usuários do *cluster* podem submeter *jobs* em qualquer instante de tempo. Os *jobs* submetidos ficam armazenados em filas, esperando pelo momento em que poderão ser colocados em execução no *cluster*.

- Os *jobs*, quando submetidos, devem ser verificados o mais breve possível. Inicialmente é preciso ordenar esses *jobs*, decidindo-se quais deles devem ser alocados primeiro. Também é preciso alocar os *jobs* no *cluster*, fazendo um mapeamento de todas as CPUs que foram requisitadas pelos usuários com CPUs reais do *cluster*. Dentre os *jobs* que são tratados, alguns podem ser alocados e colocados em execução imediatamente, outros podem ser alocados para o futuro, ou seja, podem reservar recursos do *cluster* para garantir a execução futura e outros ainda, mais especificamente *jobs* que não podem reservar para o futuro, podem ser obrigados a esperar nas filas até que existam recursos disponíveis para que os mesmos possam sejam colocados em execução.

- Ainda sobre os *jobs* que não puderam ser alocados quando submetidos, esses só serão verificados novamente na ocorrência de término ou submissão de outros *jobs*, ou após decorrido um certo período de tempo.

- Todos os *jobs* devem ter definidos seus tempos estimados de execução e seus recursos no momento da submissão, não havendo a possibilidade de modificações dos recursos solicitados durante a período de vida do *job*.

- Não há compartilhamento de tempo, ou seja, não há *time sharing* de recursos. Isso significa que as CPUs alocadas para um *job* são exclusivas para ele durante todo seu tempo de execução, não podendo ser utilizadas por outros *jobs*.

- Os *jobs* não possuem restrições definidas pelos usuários quanto à ordem em que podem ser executados. Fica a cargo do escalonador decidir a ordem em que os *jobs* iniciam suas execuções, ordem essa que tem base em políticas e técnicas que procuram melhorar o desempenho do escalonamento.

- Os *jobs* não podem ficar esperando para sempre de modo que uma política contra *starvation* de *jobs* é implementada.

- *Jobs* podem possuir um prazo que é definido pelo usuário no momento da submissão e que não pode ser modificado. *Jobs* de caráter emergencial podem ter permissões, ou poderes, para desalocar outros *jobs*, sendo que existem diferentes tipos de poder para diferentes tipos de *job*. Os poderes dados a um *job* de emergência para desalocar outros *jobs* são definidos durante a submissão e não podem ser modificados. Os *jobs* emergenciais, caso não possam obter recursos durante sua alocação, são os únicos que podem ter o poder para fazer a preempção de outros *jobs* em execução. Caso um *job* sofra preempção, ele precisa ser alocado novamente e recomeçar toda a sua execução.

- Cada *job* pode ocupar uma ou mais *CPUs* de uma ou mais máquinas do *cluster*. Todas as *CPUs* requisitadas por um *job* ocupam o mesmo intervalo de tempo solicitado pelo usuário, sendo alocadas simultaneamente. As *CPUs* alocadas durante a execução de um *job* não podem ser modificadas enquanto o *job* estiver em execução.

- Na solicitação de um *job*, o usuário pode especificar de quais máquinas ele pretende alocar as *CPUs*, e/ou deixar que o escalonador decida quais máquinas utilizar.

Capítulo 3

Trabalhos Relacionados

3.1 Sistemas de Gerência de Recursos Distribuídos (D-RMS)

Um Sistema de Gerência de Recursos Distribuídos ou *Distributed Resource Management System* (DRM ou D-RMS) [53] é um software que tem por objetivo possibilitar a um grupo de usuários utilizar de maneira eficiente um *cluster* de computadores. O *D-RMS* oferece funcionalidades, das quais se destacam as seguintes: Gerenciamento dos recursos do *cluster*, enfileiramento de *jobs*, escalonamento de *jobs*, execução de *jobs*. O software oferece aos usuários a possibilidade de solicitar um subconjunto de recursos do *cluster*, geralmente associando a esses recursos um conjunto de programas para ser executado. Uma solicitação de um usuário, recursos + programas, define um *job* que será enfileirado com outros *jobs*, depois escalonado entre outros *jobs* e finalmente posto em execução dentro dos recursos solicitados, adquirindo, quando possível, uso exclusivo desses recursos até o término de seu conjunto de programas ou do término do tempo máximo para a execução do *job*. Dentre os possíveis recursos solicitados pelo usuário destacam-se *CPUs*, memória, máquinas inteiras e licenças de software. Os usuários também podem ajustar parâmetros de qualidade de serviço para seus *jobs*. As solicitações dos usuários serão levadas em conta no momento do escalonamento e os *jobs* serão colocados para executar quando os recursos estiverem disponíveis, em uma ordem visando atender a solicitação dos usuários e as configurações do administrador do *D-RMS*.

3.2 Classificação

Os *D-RMS* são programas que geralmente são configurados e funcionam a nível de *cluster*. Essa categoria de software recebe alguns nomes diferentes na literatura, como: *job management systems, resource management systems (RMS), schedulers, queue systems, or batch systems*, ou seja, Sistema de Gerenciamento de *Jobs*, Sistema de Gerenciamento de Recursos, Escalonadores, Sistema de Filas, ou sistemas de arquivo de comandos em lote (*batch*).

Os *D-RMS* em geral centralizam em um ponto do *cluster* a submissão, o escalonamento e o acompanhamento dos *jobs*, podendo ter tolerância a falhas. O software em si pode ser separado em alguns serviços que se comunicam via *TCP/IP*. Mais comumente, um serviço controla os usuários, grupos e permissões, possui as listas de recursos e as filas, além de manter os *jobs*. Um segundo serviço escalona os *jobs* nos nós de execução. Um terceiro serviço, normalmente com uma instância por nó de execução, coloca o programa do usuário em execução, monitorando e possibilitando o controle do mesmo.

Geralmente estes possuem um escalonador com algumas funcionalidades, ainda que possibilitem a integração com outros softwares apenas escalonadores, podendo ser esses últimos mais sofisticados. O MAUI e o MOAB podem ser citados como exemplos de escalonadores. Os escalonadores de *jobs* ou *batch schedulers* [39], tratam de processos de usuário que ainda não estão na fila dos prontos, sendo eles também conhecidos como escalonadores de *jobs*.

Os *D-RMS* podem servir como base para outros softwares, alguns deles funcionando em camadas de abstração mais altas, como softwares para submissão de *jobs* em *grid* e trabalhar junto com software de escalonamento e monitoramento. Softwares como *Globus*, *Virtual Grid Execution System* (vgES), MOAB [9, 13, 1, 33, 50], trabalham em conjunto com softwares *D-RMS*.

3.3 Softwares D-RMS mais comuns

Nesta seção descreveremos os softwares *D-RMS* mais comuns [40, 14, 37, 47, 21, 36, 35, 3, 1, 38, 22, 16, 45, 51, 2], fazendo uma breve comparação entre eles, que serão melhor descritos mais adiante.

- A família PBS

São softwares baseados originalmente no *Network Queuing System* (NQS) [42] desenvolvido pela NASA, como o OpenPBS [35] e o TORQUE, ambos com códigos abertos.

  - O TORQUE foi originado do OpenPBS e mantido pela comunidade, podendo ser baixado no site do Cluster Resources Inc.
  - O OpenPBS está descontinuado a alguns anos.
  - O PBS Professional (PBS Pro) é um software proprietário mantido pela Altair. Ele possui funcionalidades do OpenPBS e do MAUI (como reservas de recursos).

- O Sun Grid Engine (SGE) - Oracle Grid Engine (OGE)

  O OGE, assim chamado desde que a Oracle adquiriu a Sun, originalmente conhecido como SGE [15] é um software proprietário baseado no CODINE [49] existindo um versão em código aberto do mesmo também conhecida com Grid Engine [14].

- LoadLeveler da IBM

  Foi originado de um *Batch System* para *clusters* da IBM que usavam o Unix AIX, sendo portado para o Linux. Ainda que seja portado para o Linux, para que se possa usar todas as funcionalidades, um hardware da IBM é requerido.

- Load Sharing Facility (LSF)

  Software proprietário da IBM, muito utilizado. É usado pelo departamento de defesa Americano e pelo departamento de energia.

- Simple Linux Utility for Resource Management (SLURM)

  É um gerenciador de recursos *open source*, tolerante a falhas e extremamente escalável. Usados em sistemas muito grandes, como o IBM BlueGene/L do LLNL com mais de 64000 processadores.

3.4 Comparação

A Tabela 3.1 lista algumas características de alguns dos softwares *D-RMS* mais comuns [2, 16, 15, 40, 37, 36, 35], na mesma, a coluna *Código Aberto* significa que o código fonte do programa está disponível. *Contrato de Suporte Disponível* significa que é possível contratar suporte técnico, ajuda na instalação e manutenção do programa. *Redundância*

Propriedades	TORQUE	PBS Pro	SGE	SLURM	LSF	LoadLev.	OpenPBS
Código Aberto	Sim	Não	Sim	Sim	Não	Não	Sim
Contrato de Suporte Disponível	Sim	Sim	Sim	Sim	Sim	Sim	Não
Redundância de Servidor	Sim	Sim	Sim	Sim	Sim	Sim	Não
Requer Sist. de Arquivos Compartilhado	Não	Não	Sim	Não	Não	Não	Não
Job Arrays	Sim	Sim	Sim	Sim	Sim	Sim	Não
Job Interativo	Sim	Sim	Sim	Sim	Sim	Sim	Sim

Tabela 3.1: Algumas características dos softwares *D-RMS* mais comuns.

de Servidor significa que existe um processo servidor extra, que entra em atividade em caso de falha do processo servidor padrão. *Requer Sistema de Arquivos Compartilhado* mostra a necessidade de o programa precisar de um servidor de arquivos que compartilhe pasta e arquivos em rede. *Job Arrays* indica o suporte a submissão de múltiplos *jobs* semelhantes e baseados no mesmo arquivo de *script* em que o usuário utiliza um único comando de submissão. Os *jobs* pertencentes a um mesmo *Job Array* podem ser diferenciados com o uso de uma variável de ambiente especial que pode ser utilizada no *script* do usuário. Os *Job Arrays* podem facilitar a administração e a submissão de vários *jobs* semelhantes onde apenas algum parâmetro precisa ser modificado. *Job Interativo* indica a possibilidade ter acesso direto ao terminal das máquinas escolhidas pelo escalonador quando o *job* do usuário entra em execução, útil para a depuração do *script* do usuário.

A Tabela 3.2 compara as características de alguns escalonadores que são integrados, ou internos, aos *D-RMS* com softwares escalonadores independentes [2, 16, 15, 36, 37, 3, 1, 21, 47]. A primeira coluna dessa tabela, *D-RMS Suportados*, mostra uma lista de gerenciadores de recursos compartilhados distribuídos que são suportados pelo escalonador em questão. A segunda coluna *Código Aberto* se refere ao código fonte do escalonador estar disponível para o usuário. A coluna *Redundância de Servidor* se refere à existência de outro processo que assume o escalonamento se o escalonador principal falhar. A coluna *Fair-share* indica se o escalonador suporta divisão pré-estabelecida de recursos entre grupos e usuários. A coluna *Reservas e Backfill* indica se o escalonador suporta reserva

Escalonador	D-RMS Suportados	Código Aberto	Redundância de Servidor	*Fair-share*	Reservas e *Backfill*	QoS
TORQUE interno	TORQUE	Sim	Não	Sim	Não	Não
PBS Pro interno	PBS Pro	Não	Sim	Sim	Sim	Não
SGE interno	SGE	Sim	Sim	Sim	Sim	Não
LoadLeveler int. padrão	LoadLeveler	Não	Sim	Sim	Não	Não
LoadLeveler int. backfill	LoadLeveler	Não	Sim	Sim	Sim	Não
LSF interno	LSF	Não	Sim	Sim	Sim	Não
MAUI	LoadL., LSF, PBS e SGE	Sim	Não	Sim	Sim	Sim
MOAB	LoadL., SLURM, LSF, PBS e SGE	Não	Sim	Sim	Sim	Sim
Catalina	LoadL. e PBS	Sim	Não	Não	Sim	Não
PluS	PBS e SGE	Sim	Não	Não	Sim	Não

Tabela 3.2: Algumas características de escalonadores independentes e internos aos *D-RMS*.

adiantada de recursos e se o escalonador suporta a funcionalidade de adiantar *jobs* que não estejam no início da fila, caso existam recursos livres onde esses *jobs* possam ser executados. A coluna QoS indica se o escalonador permite que o administrador possa criar e configurar objetos que representam políticas de qualidade de serviço. Cada política é criada como um objeto independente dentro do escalonador, sendo que cada uma delas pode ter características como possuir nível de prioridade, permitir preempção ou não de *jobs*, possuir restrições e ter acesso a certos recursos. Depois que objetos de política de QoS são criadas, é possível associar usuários, grupos de usuários e *jobs* a esses objetos.

Capítulo 4

O Sistema PBS

Desde que começou a ser utilizado na NASA em 1994 até os dias de hoje, o *Portable Batch System* recebeu várias melhorias e atualizações, se dividindo em três versões diferentes, utilizados em muitos *clusters*. Essas versões são mais conhecidas pelos nomes OpenPBS, TORQUE e PBS Professional. Nessa seção é apresentado um histórico do desenvolvimento do PBS, bem como uma comparação entre as principais versões, algumas interfaces web que tornam mais portáveis e facilitam a utilização e por último alguns escalonadores desenvolvidos para o PBS ou compatíveis com ele. A Figura 4.1 [42, 8, 25, 24, 23, 12, 4] mostra a evolução das versões PBS, desde seu antecessor o NQS até os dias atuais.

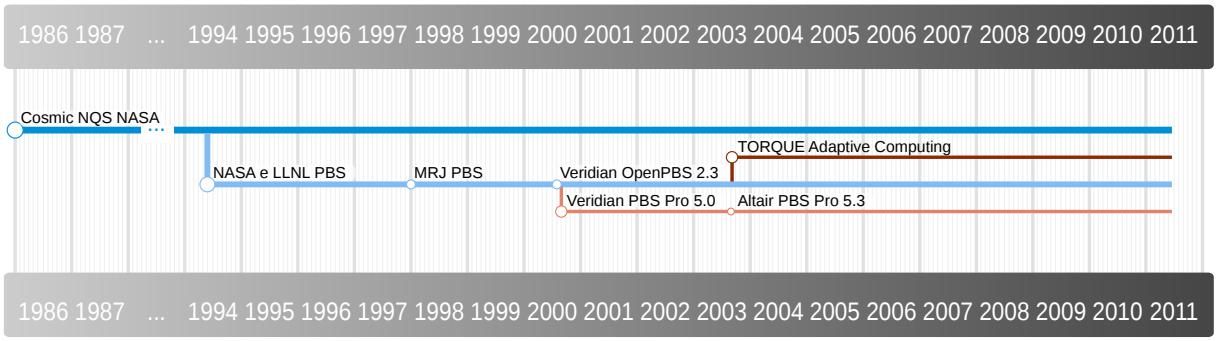

Figura 4.1: Linha do tempo das versões do PBS.

4.1 Histórico

4.1.1 O antecessor NQS

O Network Queuing System (NQS) é um sistema de enfileiramento de *jobs* conhecido desde a década de 80 e considerado o progenitor de todos os outros softwares de gerenciamento de *jobs*. A sua primeira versão, o Cosmic NQS, foi desenvolvido pela NASA em 1985,

em conjunto com a empresa Sterling Software Corporation, para servir como um sistema para controle de *jobs* em uma rede com várias máquinas UNIX. Muitos programas eram grandes demais para serem executados de modo interativo tanto em termos de tempo como de recursos. Por essa questão, os mesmo eram submetidos para filas de um sistema gerenciador de *jobs* e lá permaneciam até que os recursos estivessem liberados. Assim os usuários usavam comandos do NQS para submeter seus *jobs*, que eram arquivos de lote contendo seus comandos. Esses arquivos de lote, além de possuírem os comandos dos usuários, carregavam informações sobre as características do *job*, que seriam usadas pelo NQS. Na época em que o PBS foi desenvolvido, o NQS suportava filas que serviam como filtros (chamadas de filas *pipe*) para alimentar outro tipo de filas, chamadas filas de execução. Os *jobs* na fila de execução eram executados em ordem *FIFO*, o primeiro que entrava era o primeiro que saía. Os administradores poderiam exercer algum controle, ligando e desligando filas e controlando o máximo de *jobs* por fila, porém não havia como examinar a utilização dos recursos.

A utilização do NQS era feita via linha de comando, sendo relativamente simples de usar. Alguns comandos de usuário eram:

- qsub: Submete um *job* no sistema NQS
- qdel: Exclui um *job*
- qstat: Exibe o status dos *jobs* na fila

Após o término do *job*, eram gerados automaticamente dois arquivos, um representando toda a saída normal gerada na saída padrão (normalmente a tela), outro representando a saída de erro gerada pelo arquivo de lote do usuário.

Como a NASA estava em busca de ferramentas que permitissem um bom aproveitamento de seu parque de máquinas, no início dos anos 90, além de verificar os gerenciadores disponíveis no mercado, uma das opções foi desenvolver um sucessor de seu NQS. O PBS foi desenvolvido sobre a primeira versão do NQS, que era conhecida, como já dito, por Cosmic NQS devido a um canal de publicação de softwares livres chamado de Cosmic, que disponibilizava o NQS gratuitamente.

4.1.2 O PBS da NASA

No início dos anos 90, a NASA criou vários relatórios com listas de requisitos para avaliar ferramentas que auxiliassem no melhor aproveitamento de seu parque de máquinas da

época e que pudessem continuar sendo usados nas máquinas que seriam adquiridas. Como não encontrou no mercado sistemas que atendessem a todos os requisitos especificados, ela mesma criou um projeto para criar um sucessor para o sistema NQS.

Assim, em um esforço conjunto entre uma divisão da NASA chamada *Numerical Aerospace Simulation* (NAS) e o centro de pesquisa *National Energy Research Supercomputer Center* (NERSC), divisão do *Lawrence Livermore National Laboratory* (LLNL), foi desenvolvido o novo sistema da NASA, que foi chamado de *Portable Batch System*, com a meta de atender a alguns requisitos buscados nos relatórios criados pela NASA, como o de ser portável, ou seja, funcionar em todas as versões de Unix existentes dos laboratórios da NASA e se manter compatível com máquinas que seriam adquiridas. Para atingir esse objetivo, o PBS foi implementado sobre o padrão POSIX da IEEE, mais especificamente o padrão 1003.2d para *Batch Queuing Extensions* bem como outros padrões relacionados ao POSIX, como o POSIX.2, que define comandos usados em arquivos de execução em lote. O padrão 1003.2d surgiu da própria pesquisa da NASA em busca de um conjunto de características desejáveis para um sistema gerenciador de filas ou recursos. Outro objetivo buscado no desenvolvimento do PBS seria o de que o sistema deveria ser modular. Como o NQS não era flexível em termos de configuração de políticas para o escalonamento dos *jobs*, os desenvolvedores do PBS buscaram o extremo oposto, separando o escalonador em um módulo, que poderia ser facilmente substituído por qualquer módulo de escalonamento que implementasse a política que o usuário do sistema tivesse necessidade. O PBS começou a ser utilizado em 1994 nos laboratórios da NAS, tendo um bom desempenho apresentado em relatórios dos anos seguintes, que a NASA fazia para comparar os sistemas de gerenciamento de *jobs*. O PBS da NASA também oferecia bibliotecas que facilitavam o desenvolvimento de escalonadores sob medida, era bem documentado, com manuais de referência detalhando as APIs e as interfaces de comunicação entre os módulos do programa, como o *External Reference Specification* (ERS) e o *Internal Design Specification* (IDS). O PBS da NASA também já oferecia uma interface gráfica ao usuário (xpbs), que facilitava a submissão e o acompanhamento dos *jobs*.

4.1.3 O PBS da MRJ

Em certo momento do ciclo de vida do PBS, entre os anos de 1997 e 1998, a empresa MRJ Technology Solutions passou a deter os direitos do PBS da NASA e começou a lançar as novas versões do software, que já vinha sendo usado por outras entidades, devido a sua forte portabilidade e boa documentação.

Era possível baixar versões do PBS, com documentação, como o manual do administrador, o manual com as especificações para desenvolvimento de novos escalonadores (*External Reference Specification*) e o manual com as especificações do código (*Internal Design Specification*). O PBS ficou sendo mantido pelo MRJ até sua versão 2.2 lançada em novembro de 1999.

4.1.4 OpenPBS e PBS Professional

No final do ano 1999, a empresa MRJ foi adquirida pela Veridian, que passou a ter parte dos funcionários da MRJ, inclusive os que mantinham o PBS. Passando a cuidar do PBS no ano 2000, a Veridian criou duas versões do mesmo, uma ela chamou de OpenPBS, que era uma última versão do PBS da MRJ. A outra versão foi chamada de *PBS Professional 5.0*, também chamada de *PBS Pro 5.0*. As duas versões foram lançadas no ano 2000, sendo que o OpenPBS, que em quase nada era diferente dos PBSs lançados pela MRJ e pela NASA, parou de ser atualizado. O PBS Professional, por outro lado, recebeu algumas novas funcionalidades e continuou a ser desenvolvido pela Veridian. Entre as características que foram surgindo no PBS Professional em relação ao OpenPBS, podem ser destacadas: Servidor redundante para tolerância a falhas, possibilidade de fazer reservas de recursos, política de *backfilling* para *jobs* (onde *jobs* menores podem passar a frente de *jobs* no início da fila, caso existissem recursos ociosos capazes de atender a esses *jobs* menores), possibilidade de alocar *jobs* em máquinas de trabalho baseando-se em atividade de mouse e teclado e suporte a Windows a partir da versão 5.2. A Veridian deu suporte ao PBS Professional até a versão 5.3.

4.1.5 PBS Professional da Altair

A empresa Altair [6] adquiriu os direitos do PBS Professional e a partir do ano 2003, desde a versão 5.3, podem ser encontrados manuais referentes a versão 5.3 tanto da Veridian como da Altair. A empresa continuou acrescentando novos recursos ao PBS. A versão 11 do PBS Professional conta com suporte a GPUs. Um usuário ao requisitar recursos para executar seu *job*, pode selecionar a quantidade de GPUs que pretende utilizar.

A empresa criou um grupo de softwares voltados para computação sob demanda, a qual deu o nome de *PBS Works*. Esse grupo inclui o PBS Professional, daí seu nome. Outros softwares pertencentes a Altair são PBS Analytics e PBS Catalyst. A página web para o software é www.pbsworks.com.

4.1.6 TORQUE

No ano de 2003 a empresa *Cluster Resources* passou a dar suporte a uma versão código aberto do PBS chamada de *Tera-scale Open-source Resource and QUEue manager* (TORQUE), essa versão é uma divisão do OpenPBS versão 2.3.12, sendo que o TORQUE ganhou algumas melhorias em relação ao seu predecessor. A empresa vende uma grupo de softwares para *cluster* e grid chamado MOAB, que vem com um escalonador e que pode ser usado em conjunto com um gerenciador de *jobs*, como o TORQUE, OpenPBS ou o PBS Professional. O TORQUE possui o mesmo escalonador nativo padrão do OpenPBS (um escalonador simples escrito em linguagem C chamado de FIFO) e oferece a possibilidade para a criação de novos escalonadores baseados na linguagem interpretada TCL [19], ou nas linguagens BASL(*Batch Scheduling Language*) ou C. O manual do TORQUE sugere o uso de escalonadores como o MOAB ou o MAUI. Esse último também é mantido pela Cluster Resources, sendo disponibilizado gratuitamente. O TORQUE, em suas atualizações, se tornou mais facilmente compatível com o MAUI, apesar de o OpenPBS também ser, corrigindo pequenas falhas e facilitando o suporte para algumas funcionalidades do MAUI, como qualidade de serviço (QOS).

4.1.7 Comparando OpenPBS e TORQUE

Em termos gerais o OpenPBS e o TORQUE são muito parecidos, quase idênticos na primeira versão. O TORQUE é uma divisão do OpenPBS 2.3.12 que aconteceu em 2003, em que a maior diferença vem do fato de que o OpenPBS praticamente não é mais atualizado, enquanto o TORQUE continua sendo atualizado e disponibilizado gratuitamente pela empresa Adaptive Computing [5]. Ambos possuem código aberto, o que permite que pacotes de correções e adaptações apareçam para ambos, mas o TORQUE recebe constantemente pequenas melhorias e adaptações, principalmente para dar melhor suporte aos escalonadores MAUI e MOAB, que aos poucos o separa de sua origem, o OpenPBS.

Uma das melhorias disponíveis nas versões mais novas de TORQUE é a de permitir ao usuário definir o número de GPUs utilizadas quando submete um *job*. O suporte a *Non-Uniform Memory Archtecture* (NUMA) também é possível a partir da versão 3.0.

Quanto ao escalonador que vem em ambos os *D-RMS*, praticamente não há acréscimo de funcionalidades. Ao escalonador nativo do TORQUE se dá o nome de *FIFO*, que é o mesmo que vem com o OpenPBS, mas o TORQUE, assim como seu antecessor, pode ser usado com outros escalonadores. A própria empresa que mantém o TORQUE, a Adaptive

Computing, sugere o uso do MOAB ou do MAUI como escalonadores.

Dentre os principais acréscimos entre o TORQUE e seu antecessor, OpenPBS, podem ser destacadas melhorias na tolerância a falhas, pois o TORQUE possui redundância de servidor e continua a execução caso o processo do servidor principal falhe. Melhoramentos na interface de comunicação entre o processo escalonador do TORQUE e os processos monitores de recursos (*Machine Oriented Mini-server*, ou MOMs) que existem em cada computador de trabalho, permitindo ao escalonador fazer um monitoramento de recursos mais preciso. O TORQUE também permite ao escalonador obter dados sobre *jobs* que já foram finalizados. A escalabilidade em relação ao OpenPBS foi aumentada, o TORQUE suporta *clusters* com mais de 2500 processadores, suporta *jobs* de até 2000 processadores e suporta mensagens de servidor de tamanho maior. Os registros de execução (*logs*) se tornaram mais detalhados e vem com mensagens mais legíveis, ao invés de possuírem somente códigos de erro, o que facilita o trabalho de administração e correção de problemas.

4.1.8 Comparando OpenPBS e PBS Pro

O OpenPBS e o PBS Professional foram criados pela Veridian no ano 2000, a partir do PBS da MRJ versão 2.2. Como o OpenPBS era uma continuação do PBS 2.2, ele ficou sendo designado como OpenPBS 2.3. A primeira versão do PBS Professional recebeu 5.0 como número de versão. As diferenças entre o PBS Professional 5.0 e o OpenPBS 2.3 eram pequenas, sendo que o primeiro passou a ser vendido, enquanto o segundo continuou sendo disponibilizado gratuitamente. O PBS Professional 5.0 trazia um escalonador baseado no *FIFO* do OpenPBS com algumas melhorias, tal escalonador passou a incluir o recurso de *jobs* que podem passar a frente de outros para ocupar recursos ociosos, chamado de *backfill* de *jobs*. Os escalonadores das versões mais recentes de PBS Professional incluíram novos recursos, como novas formas de priorizar os *jobs* e suporte a reservas de recursos. Porém, a questão do escalonador limitado do OpenPBS pode ser contornada ao se adotar um escalonador externo como o MAUI, escalonadores, como esse, oferecem algumas funcionalidades que estão no PBS Pro. Outras funcionalidades disponíveis apenas no PBS Professional em relação ao OpenPBS são: redundância de servidor, possibilidade de escolher GPUs na seleção de recursos e suporte a mais Sistemas Operacionais, como é o caso do Windows, suportado desde a versão 5.2. As principais vantagens do OpenPBS é que ele é gratuito e de código aberto, oferece grande parte das funcionalidades que estão no Professional se usado junto com um escalonador externo como o MAUI (também gratuito e código aberto), podendo atender a uma grande parte de usuários que usam ou usariam

o PBS Professional.

4.1.9 Comparando TORQUE e PBS Pro

A diferença entre o TORQUE e o PBS Professional se torna bastante sutil, uma vez que ambos foram desenvolvidos sobre o OpenPBS, eram praticamente iguais no início e de lá para cá vem sempre recebendo pequenas atualizações ao longo do tempo, atualizações muito parecidas como suporte a GPUs, melhorias de tolerância a falhas, suporte a maior quantidade de máquinas e suporte a *jobs* maiores. A diferença mais significativa fica em relação ao escalonador do PBS Professional, uma vez que esse possui funcionalidades como o *backfilling* de *jobs* e reservas de recursos. Porém ao se integrar escalonador MAUI ao TORQUE, esse também passa a possuir as mesmas funcionalidades. Tirando as diferenças menores entre essas duas versões, talvez o que mais motive a adoção de uma ou outra por parte do usuário final esteja na fato de o PBS Professional poder ser instalado em máquinas Windows, ou por outro lado, de o TORQUE + MAUI estarem disponíveis gratuitamente e serem de código aberto.

4.2 Estrutura do PBS

O PBS herdou do NQS os principais comandos, como qsub, qstat e qdel. O NQS já era bastante utilizado e era o precursor de vários sistemas de gerenciamento de recursos/*jobs*. Os objetos do NQS como seu conjunto de comandos, as filas, recursos e parâmetros dos *jobs*, e muito do comportamento, como a listagem de *jobs* na fila e a geração de arquivos de saída quando o *job* termina a execução (arquivo de saída normal do script e arquivo de saída de erros do script) foram adotados pelo PBS. Segue uma relação dos comandos do PBS (OpenPBS 2.3) e seus significados.

- Comandos de usuário:
  - qsub -especificação de recursos- script.pbs : Comando para a submissão de *jobs*, onde se pode especificar os recursos a serem utilizados pelo *job* e se passa como parâmetro o arquivo de script (que contém os comandos do usuário). Em alguns casos pode se especificar o *flag* -I (no lugar do script) para se executar em modo comandos, com o recurso solicitado de modo interativo
  - qstat: Lista as filas e os *jobs*, mostrando os donos, tempo e status
  - qdel jobid: Remove o *job*

  - qmove fila_de_origem fila_de_destino: Move o *job* de uma fila para outra. Pode ser local ou remota
  - qalter : Muda parâmetros de um *job* já submetido
  - qmsg : Manda uma mensagem para os arquivos de saída do *job*
  - qorder jobid1 jobid2 : Troca a posição de dois *jobs* em uma mesma fila
  - qrerun: termina um *job* que estava em execução e submete novamente esse *job*
  - qhold jobid: Bloqueia um *job* na fila, para que o mesmo não possa ser executado
  - qrls jobid: Libera um *job* bloqueado, para que o mesmo possa ser escolhido para execução
  - qselect: Faz uma pesquisa e mostra somente os *jobs* que obedecerem aos parâmetros da pesquisa
  - qsig: Manda um sinal para o processo principal do *job*
  - nqs2pbs: Converte um script do NQS em um script de PBS

- Comandos de administrador:
  - qenable fila: Fila aceita novos *job*
  - qdisable fila: Fila não aceita novos *jobs*
  - qstart fila: *Jobs* da fila podem ser executados
  - qstop fila: Nenhum *job* da fila pode ser colocado em execução
  - qterm -t tipo servidor: Para um servidor PBS
  - qrun: Força a execução de um *job*
  - qmgr: Gerenciador de parâmetros das filas, de permissões de usuário e de parâmetros gerias do servidor

- Comandos para interface gráfica TCL:
  - xpbs: Interface gráfica que permite a submissão, modificação, exclusão de *jobs* e acompanhar as filas(Figuras 4.2 e 4.3).
  - xpbsmon: Permite monitorar o estado das máquinas do *cluster* (Figura 4.4).

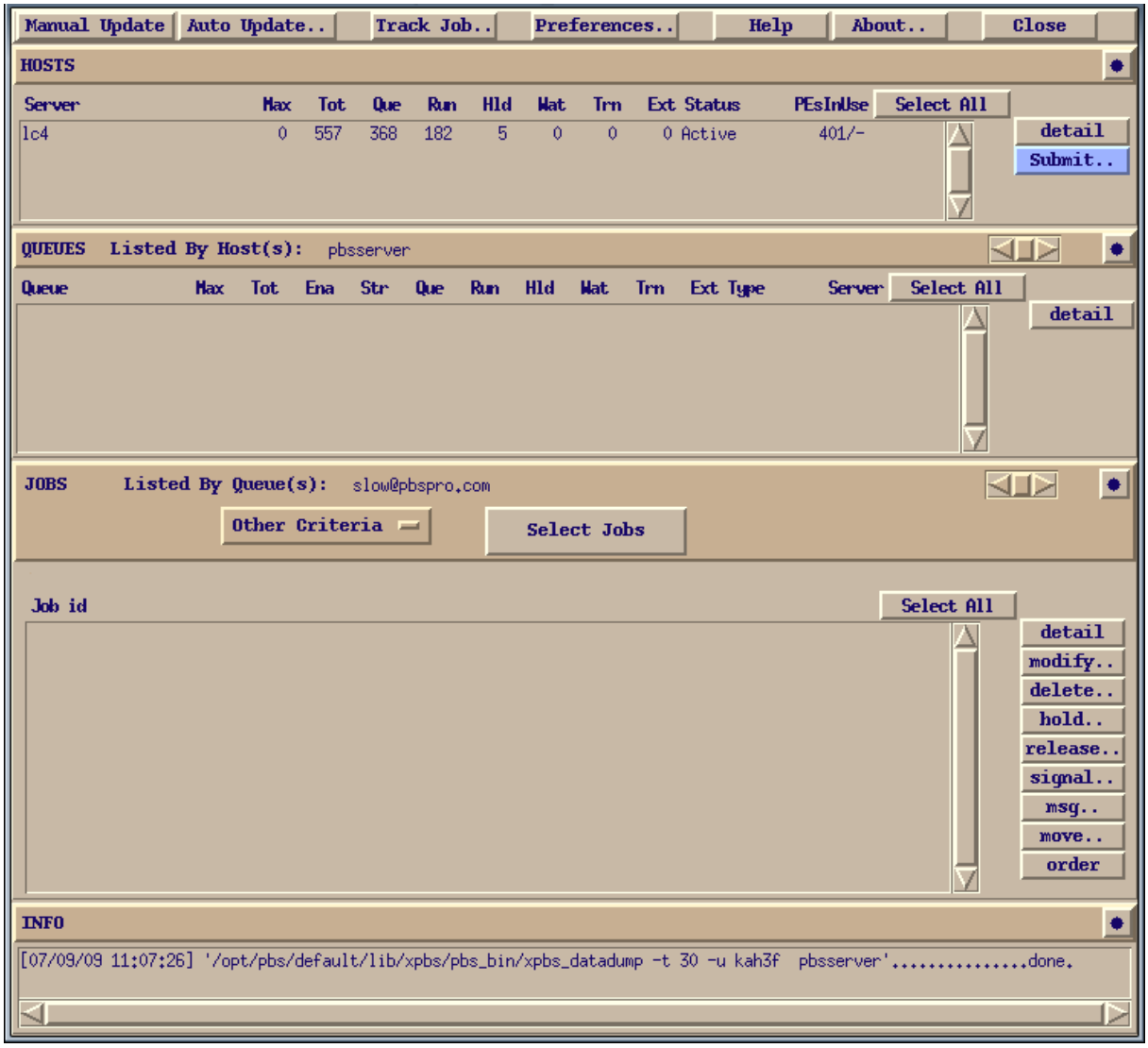

Figura 4.2: Tela da lista de filas e *jobs* do xpbs.

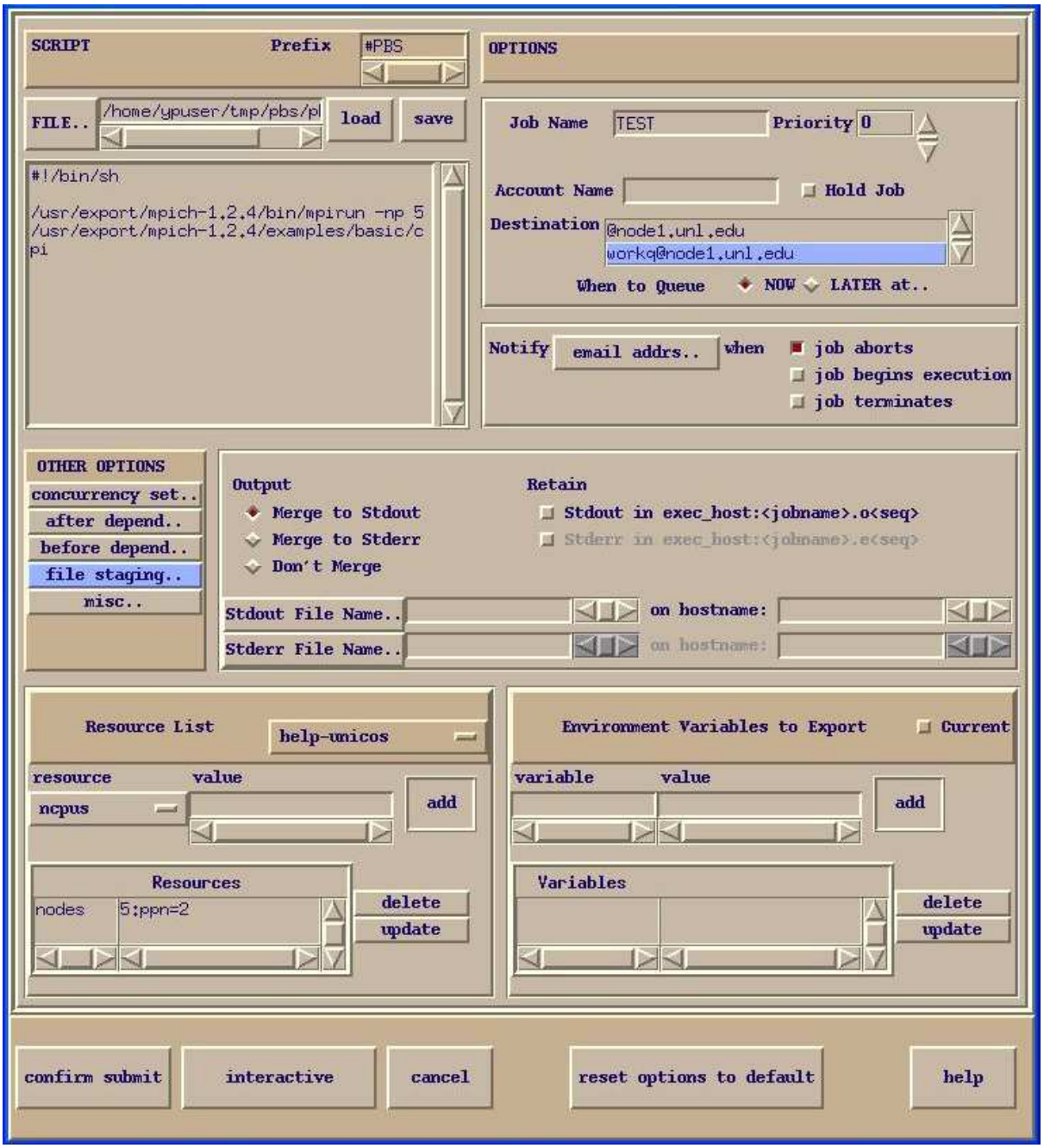

Figura 4.3: Tela de submissão de *job* do xpbs.

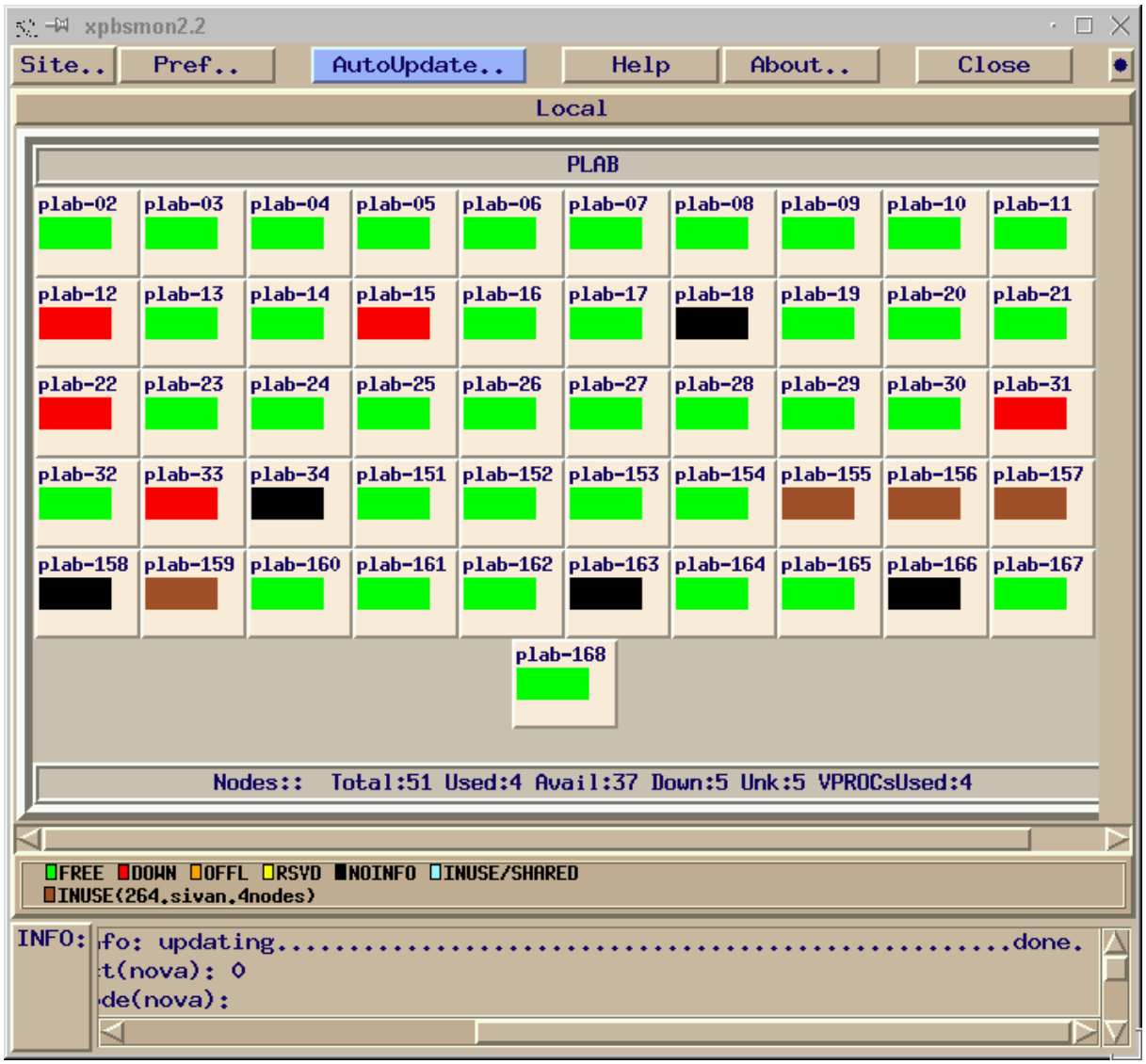

Figura 4.4: Tela do xpbsmon.

4.3 Módulos do PBS

Uma das metas do PBS era ter um escalonador separado, que permitisse a liberdade para a implementação de novas políticas de escalonamento, de acordo com a necessidade de quem utilizaria o PBS. Para isso, ele foi desenvolvido de maneira modular. Um módulo cuidaria apenas das filas dos *jobs* e dos privilégios dos usuários, esse modulo é o servidor. Outro módulo cuidaria do escalonamento e por último, um módulo cuidaria dos recursos e dos processos em cada nó de computação. Os três módulos podem ser vistos na Figura 4.5

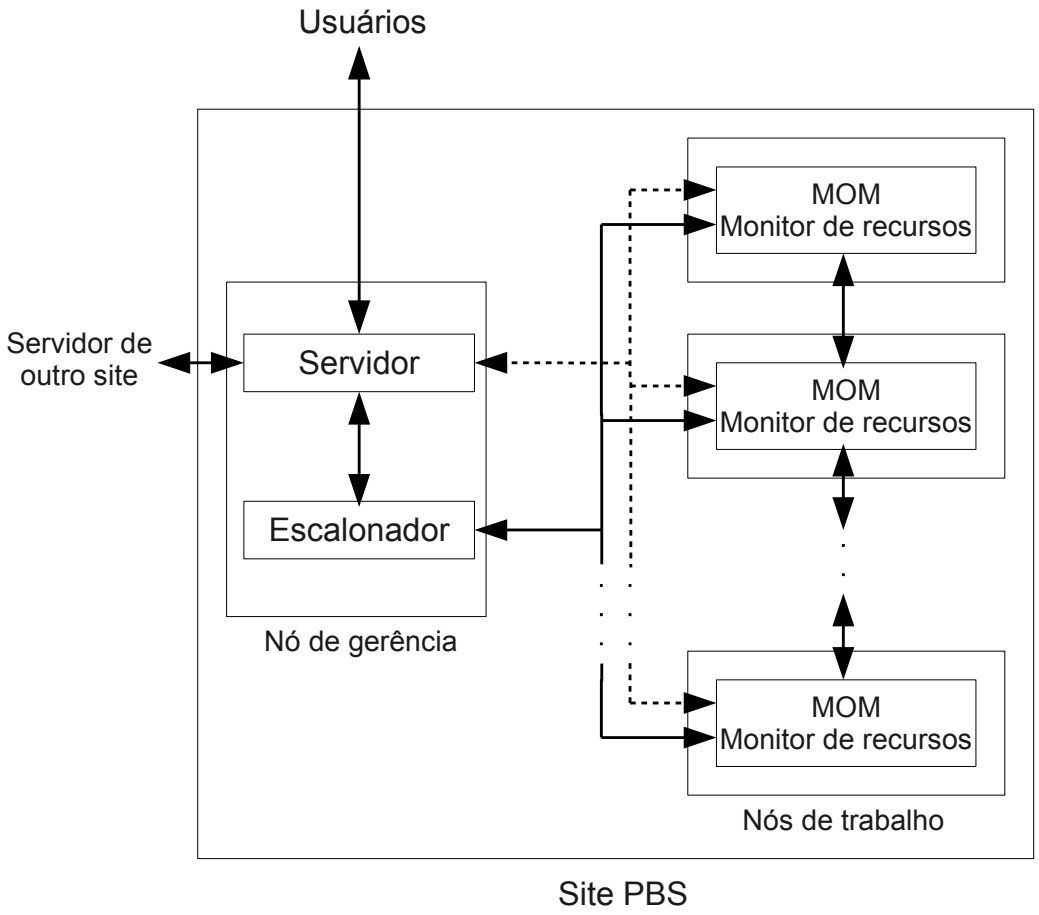

Figura 4.5: Os três tipos de módulos do PBS: servidor, escalonador e os MOMs.

4.4 Serviços do PBS

Segue abaixo uma breve descrição das funções dos três serviços (*daemons*) existentes no PBS. Cada um dos três normalmente é colocado em execução durante a inicialização das máquinas e ficam aguardando por conexões *TCP/IP* vindas dos outros dois serviços, ou de conexões vindas de comandos do administrador (tem controle total sobre as filas, os serviços, configurações, contas e *jobs* de usuários), operador (pode parar e iniciar filas, os serviços e os *jobs* dos outros usuários) e usuário comum (tem poder apenas sobre seus *jobs*). O *daemon* servidor também aceita conexões vindas de servidores remotos, para transferência de *jobs* remotos. Esses três serviços são: o servidor, o módulo escalonador e os módulos de controle dos processos em execução e dos recursos utilizados, que executam em cada máquina de trabalho (normalmente um por máquina, enquanto o servidor e o escalonador executam em apenas uma máquina) conhecidos como MOMs (*Machine Oriented Mini-server*).

- O módulo servidor: O servidor é responsável por manter e gerenciar as filas, os *jobs* e as máquinas disponíveis. Os usuários geralmente submetem *jobs* para o servidor através do comando qsub, especificando a fila, os recursos e seu arquivo de comandos (*script*) e consultam os *jobs* no servidor via qstat. Os administradores, operadores ou o módulo escalonador, solicitam ao servidor que execute os *jobs* nas máquinas, habilite ou desabilite filas, mude alguns atributos gerais, de fila ou dos nós, entre outras ações. O servidor, obedecendo ao escalonador ou aos administradores, manda que os módulos MOMs executem os *jobs*, ou que finalizem os *jobs*. Os MOMs por sua vez mantém o servidor informado sobre os *jobs* que já terminaram a execução, para que o servidor possa atualizar seu estado e chamar o escalonador para um novo ciclo de escalonamento. Existe o comando qmgr, que possibilita aos administradores mudar a configuração geral do módulo servidor, como atributos globais, atributos de filas, atributos de máquinas e permissões de usuários sem precisar reiniciar o servidor.

- O módulo escalonador: O módulo escalonador é responsável por escolher, dentre os *jobs* candidatos nas filas do servidor, qual deles vai ser o próximo a ser executado e em quais máquinas o *job* deve ser alocado. O escalonador, após consultar o servidor e os MOMs para obter os recursos das máquinas, decide se vai executar ou excluir algum *job* e manda os comandos ao servidor para que faça isso. Do ponto de vista do servidor, o escalonador é um usuário com poder de administrador. O software do

escalonador pode utilizar uma API de comandos para se comunicar com o servidor, e os comandos disponíveis nessa API é equivalente aos comandos disponíveis aos usuários, que eles usam quando precisam controlar os *jobs* no servidor.

O Servidor é programado para chamar o escalonador (ou gerar um ciclo de escalonamento) sempre que um novo *job* chega a fila e sempre que um MOM termina um *job*. Ele também chama o escalonador em intervalos de tempo definidos pelo administrador, por exemplo, de 10 em 10 minutos. Finalmente, é importante lembrar que o PBS tem como uma de suas metas permitir uma fácil substituição do escalonador, oferecendo documentação e uma Interface de Programação de Aplicações (Application Programming Interface - API) de comunicação entre o escalonador e o servidor e de comunicação entre o servidor e os MOMs na linguagem C, para a implementação de novos escalonadores capazes de se comunicar com o servidor e os MOMs. Assim o escalonador do PBS é o módulo que tem mais versões.

- O módulo Machine Oriented Miniserver (MOM) / monitor de recursos: é um *daemon* que reside em cada máquina de trabalho que o PBS gerencia, sendo responsável por controlar a execução dos *jobs* que o servidor mandou para ele, informando ao servidor sempre que um *job* termina. O MOM também age como um monitor de recursos da máquina de trabalho onde está, respondendo a consultas enviadas pelo escalonador. Ele informa ao escalonador o total de recursos disponíveis da máquina, os recursos utilizados e informa sobre alguns dados estáticos (definidos no arquivo texto de configuração do MOM daquela máquina pelo administrador) e oferece a opção de obter outros dados da máquina, através da execução de outro programa qualquer, definido no arquivo de configuração do MOM.

4.4.1 Servidor PBS

Os principais atributos globais do servidor PBS são:

- default_queue: a fila padrão para onde os *jobs* são submetidos se o usuário não especificar a fila quando submete um *job*.
- acl_hosts: lista de controle de acesso, com as máquinas *hosts* que podem acessar o servidor.
- acl_hosts_enable: habilitar ou não a lista acl_hosts.

- default_node: define os nós onde os *jobs* são executados se isso não estiver especificado.
- managers: define uma lista de usuários, em máquinas definidas ou redes definidas, que podem ter acesso com o poder de administrador.
- node_pack: define como *jobs* são agrupados em máquinas de vários processadores. Os *jobs* podem ser agrupados em um conjunto mínimo de máquinas possível (valor true) ou espalhado pelas máquinas (valor false), ou ainda seguir o arquivo de nodes, se node_pack não estiver configurado.
- operators: define uma lista de usuários, em máquinas definidas ou redes definidas, que podem ter acesso com o poder de operador.
- query_other_jobs: Define se os usuários podem ver o status dos *jobs* de outros usuários pelo comando qstat.
- resources_defaults: define valores padrões para recursos, se eles não forem especificados pelos usuários durante uma submissão via qsub. Pode ser atribuído para qualquer recurso, exemplo: resources_defaults.mem=4mb, faz com que os jobs submetidos sem o uso de memória especificado, possuam um limite máximo de 4 megabytes.
- resources_max: define o máximo de um recurso, para um *job* submetido. Só tem efeito se a fila para a qual o *job* foi submetido não tiver limite para o mesmo recurso.

Os atributos da fila do servidor PBS são:

- queue_type: define se a fila é de roteamento ou de execução. Filas de roteamento direcionam jobs para outras filas, as de execução são de onde os *jobs* podem ser escolhidos para execução.
- enabled: define se uma fila aceita *jobs* ou não.
- started: define se os *jobs* nessa fila serão processados. Roteados no caso de uma fila de roteamento, escolhidos para execução no caso de uma fila de execução.
- resources_max: Define o máximo que o *jobs* devem ter de algum recurso para serem aceitos nessa fila. Pode ser usado para definir máximo de vários recursos para uma mesma fila.

- resources_min: Define o mínimo que *jobs* devem solicitar de algum recurso para serem aceitos na fila. É possível repetir esse atributo para uma mesma fila, com vários tipos de recurso.

Os atributos de fila apenas de roteamento são:

- route_destinations: lista de filas locais ou remotas para onde o *job* pode ser enviado. A fila ser remota significa que ela pertence a um outro servidor PBS.

Os atributos de fila apenas de execução são:

- resources_default: Define a quantidade de recursos para um recurso de um *job* colocado nessa fila, caso a mesma não tenha sido especificada pelo usuário. Esse atributo, caso tenha sido especificado, substitui o atributo resources_default do servidor. Pode ser especificado para cada tipo de recurso.

4.5 Interfaces Web

Há uma série de interfaces web que foram propostas para simplificar o acesso ao sistema PBS. A seguir apresentamos as principais.

4.5.1 PBSWeb

O PBSWeb [45] foi criado na universidade de Alberta no ano 2000, com o intuito de facilitar e ajudar o uso do PBS, através de uma interface web. Sintetiza as etapas da utilização do PBS em uma página, tornando seu uso mais intuitivo, com as fases: carregar os arquivos de código fonte, compilar, criar o script de submissão para poder executar os binários compilados e submeter o script, acompanhar o andamento da submissão, pegar os resultados. A interface cria o script de submissão automaticamente, tomando como base os comandos que o usuário deseja executar e nos recursos que ele escolheu (estando os campos desses recursos disponíveis na tela), simplificando essa etapa para os novatos e ajudando na organização para os mais experientes. As telas sugerem ao usuário os próximos passos a serem tomados, como em um tutorial (wizard). Também oferece links de ajuda. Como funciona via *browser*, é independente de plataforma, evitando a necessidade de instalação de programas clientes, específicos para a plataforma de onde o

usuário precise executar seus comandos, tornando necessário apenas que a plataforma do usuário possua um *browser* que atenda a requisitos mínimos da interface web.

As telas do PBSWeb foram desenvolvidas numa linguagem para construção de páginas web dinâmicas, muito conhecida e utilizada, chamada de *PHP*, as telas fazem uso de um protocolo de criptografia conhecido como protocolo de *sockets* seguro, ou *Secure Sockets Layer* (SSL). O PBSWeb também faz uso da linguagem Perl (linguagem muito utilizada no desenvolvimento de páginas web). As telas do PBSWeb foram testadas e utilizadas principalmente em ambientes que executam o servidor de páginas web chamado APACHE, além de também utilizar a linguagem javascript para fazer a validação dos dados escritos pelos usuários nas telas web. O PBSWeb utiliza um banco de dados PostgreSQL para armazenar seus dados e utiliza o comando de linux *ssh* para se logar de modo seguro nos servidores PBS, como se fosse o próprio usuário se conectando para executar comandos do PBS.

Os *jobs* que os usuários já submeteram pelo PBSWeb ficam armazenados em um histórico, podendo ser reaproveitados ou seja, submetidos novamente, com novas modificações se for preciso.

O software PBSWeb pode ser obtido a partir de uma página da *University of Alberta* [17].

As telas Principais do PBSWeb são:

- Página de login
- Tela com os links para as etapas, semelhante a um menu principal (Figura 4.6) útil por dar ao usuário uma visão geral das etapas do PBS, mesmo havendo um menu de navegação em todas as outras páginas, que possui a mesma funcionalidade dessa tela. Também se destacam os links para troca de senha e um link para sair da sessão autenticada.
- Página para carregar o arquivo empacotado (um ou mais arquivos e pastas agrupados em apenas um arquivo, gerados com a ferramenta *tar*) e decidir se vai fazer o compilar ou não (a compilação é feita com o comando *make*).
- Tela que exibe o resultado da descompactação do arquivo empacotado enviado, mostrando os arquivos obtidos pela ferramenta *untar* e caso seja escolhida a opção de de compilação (*make*) na tela anterior (carregar o arquivo), mostra o resultado da compilação gerado pelo comando *make*.

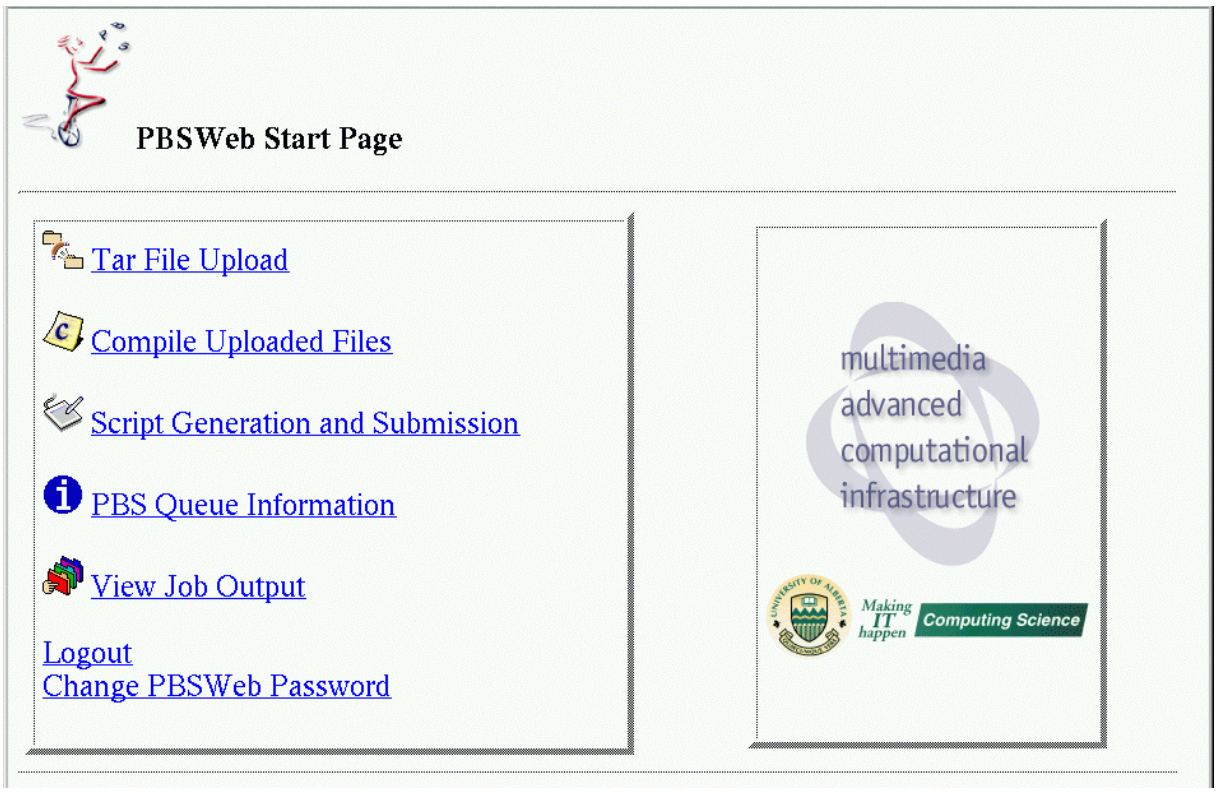

Figura 4.6: Menu principal do PBSWeb.

- Tela para criação do script de submissão do *job* PBS, permite ao usuário definir quais comandos o script deve executar, definir o nome do *job*, escolher para qual fila deseja submeter (inclusive a fila padrão), quais recursos deseja utilizar, possível carregamento de arquivos de uma máquina remota antes da execução do *job* (*stagein*) - possível cópia de arquivos para uma máquina remota após a execução do *job* (*stageout*) e a possibilidade de escolher um email para onde o PBS deve enviar mensagens sobre o andamento do *job*.

- Página com o resultado da submissão, exibindo o script de submissão que foi gerado pelo PBSWeb e o jobid que o *job* recebeu caso a submissão tenha sido aceita pelo PBS. O script de submissão gerado é automaticamente salvo em um arquivo.

- Tela que exibe o status de todos os *jobs* e filas disponíveis ao usuário, possibilitando ao mesmo escolher qual servidor deseja visualizar. O usuário pode excluir seus *jobs* que estejam esperando na fila ou em execução, caso queira.

- Tela que permite visualizar a saída dos *job*, após sua execução.

4.5.2 PBS-Libra Web

No trabalho de implementação do escalonador Libra [51], o código do PBSWeb [45] foi modificado para oferecer suporte aos parâmetros do Libra, Tempo estimado (*Estimate*), prazo (*Deadline*) e orçamento (*Budget*) como pode ser visto na Figura 4.7 . Assim esse escalonador aproveitou todos os benefícios oferecidos pela interface web, como a alta portabilidade e facilidade de uso.

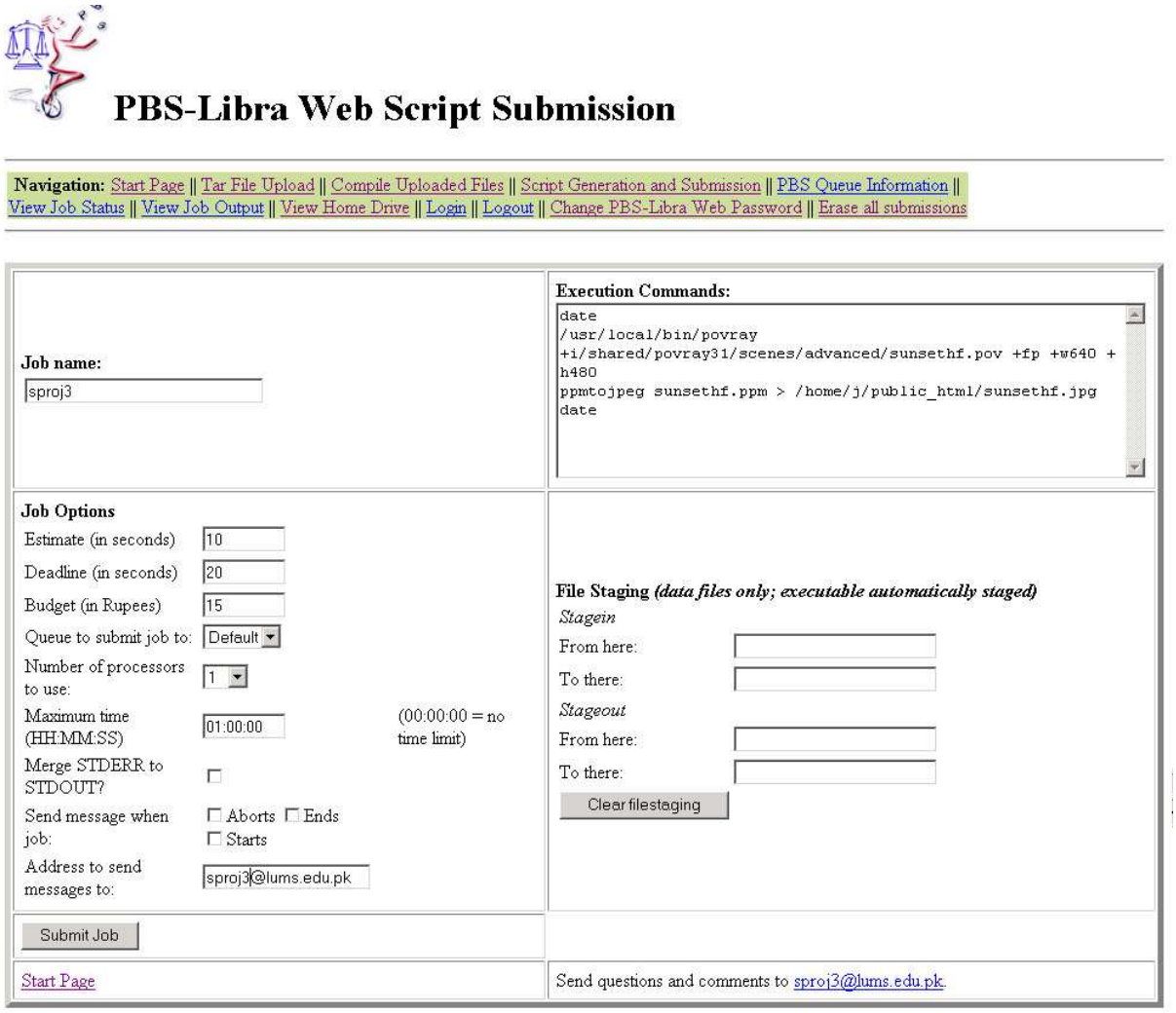

Figura 4.7: Página de submissão de *jobs* do PBS-Libra Web.

4.6 Escalonadores Disponíveis

O código fonte de vários tipos de PBS (no caso, o antigo PBS da NASA, OpenPBS, TORQUE e primeiras versões do PBS Professional) trazem com eles alguns modelos diferentes de escalonador.

4.6.1 Escalonadores Nativos

No momento da instalação, ou compilação dos fontes do PBS, o usuário que está instalando pode escolher um escalonador, que já vem junto com os fontes do próprio PBS, utilizando parâmetros especiais durante a instalação. Uma instalação feita sem se especificar um dos escalonadores disponíveis, faz com que um escalonador escrito em linguagem C, chamado de FIFO, seja instalado por padrão. Apesar do nome, o escalonador padrão faz mais do que simplesmente escalonar os *jobs* em fila FIFO (o primeiro a entrar é o primeiro a sair) e não faz isso por padrão, ele permite ordenar os *jobs* com base em número de CPUs requeridas, por exemplo e permite um tratamento simples para prevenir que *jobs* sejam preteridos indefinidamente. Além do FIFO, o PBS traz escalonadores baseados em linguagem TCL e BASL (*BAtch Scheduling Language*). O administrador poderia modificar os *scripts* para ter uma política a seu gosto. Ambos os *scripts* poderiam consultar os objetos disponíveis no servidor e nas máquinas, para tomar decisões. Os fontes do PBS também trazem outros escalonadores escritos na linguagem C além do próprio *FIFO*, sendo esses outros escalonadores destinados a arquiteturas específicas. Esses precisam ser compilados durante a instalação do PBS e funcionam como o *daemon* escalonador.

4.6.2 Escalonador Padrão

Praticamente todas as versões do PBS vem com um escalonador padrão chamado *FIFO*, ou que toma esse como base, no caso do PBS Pro. Esse escalonador está presente como o padrão desde versões bastante antigas do PBS (versões da NASA). O OpenPBS e o TORQUE herdaram o escalonador *FIFO* dos PBSs anteriores, sem muitas alterações. Mesmo o PBS Professional 11, que é o mais recente no momento em que é realizado esse estudo, também utiliza uma versão melhorada mas fortemente baseada no *FIFO*, com algumas funcionalidades extras, como *backfilling* (*jobs* pequenos de menor prioridade podem passar a frente de outros *jobs* na fila para utilizar recursos que estejam ociosos) e permitir definir uma fórmula para cálculo de prioridade do *job*. Apesar do nome, o escalonador padrão não se comporta como um *FIFO* (*first in, first out*, ou o primeiro *job* a entrar é o primeiro a sair), a menos que sejam feitas várias modificações em seu arquivo de configurações original. Em sua configuração padrão, ele funciona como um simples método guloso para otimizar o uso de CPU, colocando os *jobs* que pedem menor número de CPUs com prioridade e possuindo uma proteção para *jobs* não sejam preteridos indefinidamente, dando prioridade maior a esses *jobs* caso um determinado prazo de espera na fila tenha sido ultrapassada.

O escalonador *FIFO* pode ser facilmente substituído por outro com mais recursos ou com políticas de escalonamento muito específicas. O *FIFO* também pode ser modificado uma vez que seu código fonte em linguagem C está disponível nas versões do PBS que possuem código aberto.

4.6.3 Escalonador MAUI

O MAUI é um escalonador de código aberto e disponibilizado gratuitamente pela empresa Adaptive Computing que pode ser integrado ao PBS, substituindo o escalonador padrão. O manual do TORQUE recomenda fortemente a substituição do escalonador padrão pelos escalonadores MAUI ou MOAB, uma vez que o TORQUE, o MAUI e o MOAB são mantidos pela Adaptive Computing.

O MAUI possui funcionalidades que o escalonador *FIFO* do OpenPBS e do TORQUE não possuem, como *Backfill* de *jobs* (apenas a versão comercial, o PBS Professional, possui *Backfill* incorporado ao escalonador nativo) e reserva adiantada de recursos e suporte a qualidade de serviço, que o tornam o MAUI interessante para as pessoas que usam esses tipos de PBS. No caso do PBS Professional, as versões mais novas possuem alguns desses recursos em seu escalonador padrão, um *FIFO* melhorado, como reservas adiantadas de recursos e *backfill*, a maior diferença fica por conta do parâmetro de qualidade de serviço do MAUI, que o PBS Professional não implementa.

Do ponto de vista da interface, os usuários executam os comandos do PBS como antes, ou seja, submetem os *jobs* via *qsub* e acompanham os *jobs* com o comando *qstat*. Além dos comandos comuns do PBS, o MAUI oferece comandos extras, para novas funcionalidades e informações.

Em maiores detalhes, seguem as funcionalidades do MAUI:

- Backfill: Permite alguns *jobs* executarem fora de ordem normal de prioridade, desde que não atrasem o *job* de maior prioridade da fila. Os *jobs* precisam ter o tempo de execução estimado, onde uma estimativa de boa qualidade é desejável. Tempos de execução muito pequenos, podem fazer o *job* ser morto antes de terminar, enquanto tempos de execução muito grandes podem atrasar a execução de um *job* sem necessidade. O MAUI oferece o comando *showbf*, que mostra aos usuários que recursos estão disponíveis para uso imediato. Isso ajuda aos usuários dimensionarem seus *jobs* para tirar proveito dos recursos disponíveis no momento e do mecanismo que o *backfill* oferece. O *Backfill* melhora a ocupação do sistema e diminui o tempo de

permanência *turnaround time*, em relação ao escalonador *FIFO* padrão do PBS, ao utilizar os espaços livres.

- Gerenciamento de alocação: Permite oferecer aos usuários uma estrutura similar a contas bancárias, que possuem créditos que podem ser trocados por recursos computacionais disponíveis. Um usuário pode ter mais de uma conta com créditos e ao submeter um *job*, pode escolher qual conta será debitada pelo recurso utilizado.

- Reservas Adiantadas: Permite configurar a reserva de alguns recursos para certos usuários por um período definido de tempo. O acesso a uma reserva é feita por lista de controle de acesso (ACL) que define quais usuários poderão acessar essa reserva. O MAUI pode forçar os *jobs* a executarem em reservas, de acordo com parâmetros de qualidade de serviço.

- Qualidade de serviço (QoS): Permite definir privilégios especiais para alguns usuários, como acesso a recursos extras, isenções de algumas políticas, acesso a funcionalidades especiais e priorização de *job* melhorada. É possível definir quais vantagens estão disponíveis. O usuário com permissão de usar alguma qualidade de serviço, usa o parâmetro QoS no comando qsub do PBS.

- Estatísticas: O MAUI cria um grande número de estatísticas para ajudar os usuários a determinarem como e com que frequência seus *jobs* estão sendo executados. O comando *showstats* fornece estatísticas detalhadas com base no usuário , ou grupo. O comando *showgrid* pode ser usado para determinar quais tipos de *job* tem a melhor desempenho nos escalonamentos, servindo como modelo para os usuários ajustarem seus *jobs* e obter melhores tempos de permanência (*turnaround time*).

- Diagnósticos: O comando *checkjob* permite aos usuários verem o estado detalhado de cada *job* que eles submeteram, utilizando número o identificador do *job jobid*. Esse comando mostra todos os atributos do *job* e informação do estado e também provê uma análise se o *job* poderá executar ou não. Se o *job* não for capaz de executar, o comando mostra razões para isso. O comando *showstart* fornece uma estimativa de quando o *job* pode iniciar.

- Informação da carga de trabalho: Os comandos que listam os *jobs* e as filas, como o *qstat* do PBS, não refletem a ordem em que os *jobs* estão para ser escalonados, nem com que ordem os que estão em execução foram escalonados. O comando *showq* do MAUI oferece uma lista mais detalhada para os *jobs* na fila e em execução.

4.6.4 MOAB

O MOAB é um grupo de softwares, tanto em versão para *grid* como para *cluster* comercializado pela empresa Adaptive Computing que possui um escalonador. Basicamente, o escalonador do MOAB Cluster Suite oferece as mesmas funcionalidades do MAUI, possuindo alguns itens extras como interface web e suporte a um número maior de arquiteturas. Assim como o MAUI, o MOAB pode ser usado para substituir o escalonador padrão do OpenPBS, TORQUE e o PBS Professional.

O MOAB oferece uma interface web e java ao usuário e oferece vários tipos de relatórios e gráficos de uso.

4.6.5 Libra

O escalonador Libra [51], faz o escalonamento (alocação de recursos) baseado nas requisições de qualidade de serviço dos usuários. Ele se baseia na ideia de economia de mercado, tomando como parâmetros o tempo estimado de duração do *job*, o orçamento e o prazo do cliente. Com isso, o foco desse escalonador está em usar a satisfação dos usuários como métrica de escalonamento, uma vez que o próprio usuário escolhe quanto de seus créditos ou quota está disposto a oferecer para que um de seus *jobs* execute em um prazo determinado. Uma situação mais flexível seria obtida se os usuários pudessem barganhar os créditos entre si, como no sistema para *grids* Nimrod-G. Se uma pessoa realmente tem um prazo apertado para cumprir ela vai procurar não desperdiçar alocação de recursos para conseguir alocar no momento certo e será recompensada ou punida pelo seu próprio uso dos recursos.

Os principais parâmetros que o escalonador considera para cada *job* são:

- tempo estimado de execução do *job*, representado pela letra E,
- prazo para a finalização do *job* D,
- orçamento (*budget*) que ele esteja disposto a pagar pela término do *job* no prazo B

O escalonador assume as condições:

- todos os nós só executam *jobs* aprovados pelo servidor. Os nós são heterogêneos (hardware e software)

- os nós aceitam o parâmetro porcentagem de CPU alocado ao *job* e deve ser capaz de forçar isso, a soma de todas as fatias é menor ou igual a 100.
- O tempo estimado(E) dado pelo usuário é correto para um *job*, rodando em qualquer no do *cluster*.

Como dito anteriormente, esse escalonador não oferece mecanismo de barganha de créditos entre os usuários, para que eles possam negociar com os outros pelo uso de recursos, de acordo com as suas considerações, como acontece no ambiente de *grid* de projetos como o Nimrod-G

O *budget* (B) oferecido é aceito ou não com base numa fórmula de custo mínimo:

$Custo = \alpha * E + \beta * (E/D)$ (α e β são coeficientes da fórmula)

O valor de α tem impacto no custo do recurso por *job* pode ser levado pela demanda e estoque de recursos. O valor de β tem impacto no incentivo oferecido ao usuário por especificar o prazo. Maiores incentivos vem para usuários com prazos maiores.

O primeiro termo da função de custo, indica que o valor aumenta, se o tempo estimado for maior, independente do prazo. O segundo termo indica que o custo depende de uma relação entre o tempo estimado e o prazo. Prazos maiores devem impactar na fórmula, diminuindo o custo.

Se os nós não puderem atender ao prazo do usuário, ele deve tentar depois, ou tentar um prazo mais relaxado. Se aceito, o *job* é mandado para os nós com menos carga, entre aqueles que aceitaram.

O Libra foi implementado como um escalonador para o PBS. Um exemplo de comando de submissão de *jobs*, com os parâmetros direcionados para o Libra seria: qsub script.pbs -w deadline=x, budget=y, estimate=z Onde *deadline* é o prazo, *budget* é o orçamento, *estimate* é o tempo estimado e script.pbs é o arquivo de comandos submetido pelo usuário.

A interface PBSWeb foi modificada para dar suporte aos parâmetros de qualidade de serviço do Libra.

4.6.6 UnderLord

O escalonador UnderLord [22] foi desenvolvido em 2001, pelo TJNAF - Thomas Jefferson National Accelerator Facility na Southeastern Universities Research Association (SURA) a pedido de United States Department of Energy.

O projeto do escalonador UnderLord faz parte de um projeto maior, feito em conjunto com o MIT, para o desenvolvimento de um MetaEscalonador a ser usado entre algumas instituições. O projeto maior possui dois escalonadores, um deles chamado OverLord, responsável por migrar *jobs* entre sites. E o UnderLord, responsável por tratar os *jobs* localmente.

O escalonador UnderLord foi desenvolvido em linguagem C++ para facilitar a modularidade e o reuso do código. As bibliotecas do UnderLord fazem uso das bibliotecas disponibilizadas para a comunicação entre o escalonador com o servidor e os monitores de recurso do PBS.

Com alguns ajustes, o código fonte do escalonador UnderLord pode ser compilado com a versão 2.3.12 do OpenPBS. Assim como o *FIFO* do PBS, o UnderLord possui um arquivo de configuração onde o administrador pode ajustar alguns parâmetros e definir como os *jobs* devem ser priorizados.

O Escalonador é multi-estágio, onde cada estágio tem opções de escalonamento para definir seu comportamento.

Para priorizar os *jobs*, o algoritmo do escalonador trabalha com o peso dos vários estágios, ou seja, os *jobs* são ordenados de acordo com seu tempo na fila, duração prevista, número de nós requisitados, o uso feito de recursos pelo usuário e a fila, entre outros.

Apesar de parecer simples, os estágios de escalonamento do UnderLord não podem ser considerados um de cada vez. Isso levaria a situações de *jobs* preteridos indefinidamente *starvation*. Por exemplo, se o estágio de duração do *job* for considerado primeiro, os demais estágios trabalhariam apenas como critério de desempate entre os *jobs* que pedissem menores quantidades de tempo, sendo que os *jobs* maiores ficariam em desvantagem e poderiam nunca conseguir executar.

Assim, os estágios são considerados simultaneamente, por meio do cálculo do peso total do *job*, que é a multiplicação do peso de cada um dos estágios. Antes do produto dos pesos para cada estágio, os mesmo valores (que a princípio poderiam assumir qualquer valor entre menos infinito e mais infinito) são mapeados em um intervalo limitado entre 0 e 1. A cada estágio o administrador pode atribuir pesos, sendo que o peso 0 anula o efeito daquele estágio sobre o peso total do *job*.

4.7 Melhorias Propostas

Nessa seção são explicadas as melhorias propostas no escalonador OscarPBS desenvolvido nesse trabalho, sobre os escalonadores dos softwares *D-RMS* sucessores do OpenPBS [35, 2, 16]. Uma das melhorias propostas para o escalonador do OscarPBS é o uso de um método chamado de *Empurrar*, que tira proveito da alocação de *jobs* não exploradas pelos escalonadores dos *D-RMS* da família PBS, tendo impacto no desempenho do escalonamento. Outra melhoria é oferecer aos usuários do OscarPBS uma funcionalidade de prazo, que possibilita aos usuários um melhor controle sobre seus *jobs*.

4.7.1 Melhorias no algoritmo de escalonamento de jobs

Durante o escalonamento de *jobs*, os algoritmos de escalonamento utilizados nos *D-RMS* precisam, em algum momento, fazer a da escolha das máquinas do *cluster* onde os *jobs* serão alocado. O processo de escolha das máquinas afeta o modo como os *jobs* se encaixam nas máquinas consequentemente afetando a forma como os recursos do *cluster* são utilizados. Um algoritmo que fizesse uma escolha ótima para o escalonamento de *jobs* paralelos seria proibitivamente custoso em termos computacionais [29], então os módulos de escalonamento dos *D-RMS* sucessores do OpenPBS [35, 2, 16] utilizam heurísticas de escolha de máquinas que encontram uma solução para a questão de quais máquinas utilizar para alocar um *job*. Porém como as heurísticas não necessariamente encontram a melhor solução de encaixes possível, é possível verificar situações em que uma escolha diferente de máquinas permitiria a execução de *jobs* em CPUs que na prática ficaram ociosas devida a escolha feita pela heurística.

Durante esse trabalho se observou que com algumas mudanças no modo como os *jobs* eram alocados nas máquinas pelos escalonadores dos *D-RMS* da família PBS, algumas situações que levariam a CPUs ociosas e a *jobs* não alocados por não poderem ser encaixados nessas CPUs, poderiam ser resolvidas se partes de *jobs* já alocados fossem movidos para outras máquinas do *cluster*. Para explorar essa possibilidade de mover *jobs* alocados e utilizar melhor os recursos, um método chamado de *Empurrar* foi desenvolvido nesse trabalho e incluído do escalonador do OscarPBS, visando tornar o escalonamento mais eficiente.

4.7.2 Novas funcionalidades

O OscarPBS acrescenta novas funcionalidades aos softwares *D-RMS* da família PBS, principalmente com relação as versões que são de domínio público, OpenPBS e TORQUE. A versão comercial da família PBS, o PBS Professional, apresenta funcionalidades semelhantes as propostas pelo OscarPBS, sendo essas funcionalidades o suporte a *backfill* diante de *jobs* em *starvation* [16] e a possibilidade de reservar recursos antecipadamente para execução de alguns *jobs*, que de alguma forma equivale a funcionalidades de *jobs* com prazo oferecidas pelo OscarPBS.

Pode-se dizer que os acréscimos de recursos foram feitos sobre o *D-RMS* TORQUE. O TORQUE, sendo de domínio público e de código aberto, possibilitou que algumas modificação fossem feitas em seu código de modo que o OscarPBS fosse integrado a ele. Com o OscarPBS substituindo o escalonador padrão do TORQUE, as seguintes funcionalidades não disponíveis no TORQUE foram acrescentadas:

- Suporte a *Backfill*: A funcionalidade *Backfill*, no caso do OscarPBS, se aplica em ocasiões onde existem *jobs Starve*, que são *jobs* que esperaram muito tempo na fila sem conseguirem ser alocados e que passam a ser priorizados diante de outros *jobs* mais novos, forçando esses *jobs* a executarem depois do *job Starve*, obedecendo a uma política *FIFO*. O método *Backfill* se caracteriza por permitir que *jobs* mais recentes que os *jobs Starve* possam ser alocados antes dos *jobs Starve* desde que isso não atrase a execução do *job Starve*. O escalonador padrão do TORQUE não faz *Backfill* retendo todos os recursos do *cluster* até que o *job Starve* possa ser executado. O escalonador do OscarPBS, por outro lado, reserva espaço para os *jobs Starves*, mas permite que *jobs* mais novos que caibam em recursos ociosos do *cluster* sejam alocados, desde que não atrasem a execução dos *jobs Starves* reservados.

- *Jobs* com prazo: O OscarPBS oferece suporte a dois tipos novos de *jobs*, que permitem que os usuários e o administrador tenham controle sobre quando o *job* deve terminar. Nesse trabalho, esses *jobs* são chamados de *jobs* QoS e *jobs* Emergência. Os *jobs* QoS tem prazo definido pelo usuário assim que são submetidos. Se o escalonador conseguir reservar um espaço para o *job* QoS dentro do prazo estabelecido pelo usuário, o usuário fica sabendo que a reserva foi feita e que a execução de seu *job*, na maioria das vezes, está garantida. Um *job* QoS alocado somente perderia seus recursos diante de alguma falha ou se um *job* de Emergência submetido depois do *job* QoS requisitasse seus recursos. Um *job* de Emergência é um *job* com prazo,

de caráter emergencial, que pode ser submetido pelo administrador do OscarPBS. O *job* de Emergência possui poderes extras para desalocar outros *jobs*, que um *job* QoS normalmente não possui. Assim o *job* de Emergência pode, se necessário, desalocar outros *jobs* para conseguir garantir seus recursos. O OscarPBS oferece diferentes tipos de poder, ou capacidade para desalocar outros tipos de *jobs*, ao *job* Emergência, enquanto também procura alocar o *job* Emergência de modo a desalocar o mínimo possível de *jobs*, principalmente *jobs* QoS.

As funcionalidades extras oferecidas pelo OscarPBS requerem a substituição do módulo escalonador do TORQUE, bem como o acréscimo do módulo de interface com o usuário. O OscarPBS então complementa o TORQUE, com o acréscimo de uma interface e de um módulo de escalonar próprio, essa interface utiliza um banco de dados, para guardar informações com respeito a requisições de *jobs* feitas pelos usuários bem como possibilitar a troca de informações entre a interface com o usuário e o módulo escalonador. No capítulo seguinte, o foco principal será o escalonador do OscarPBS. A interface com o usuário e o banco apesar de fazerem parte desse trabalho, serão por conveniência vistos como elementos separados no capítulo que se segue. O módulo de interface com o usuário desse trabalho é representado por um conjunto de programas que simulam o comportamento de usuários e acessam diretamente o banco de dados do OscarPBS para submeter *jobs*. A interface com o usuários reais, quando implementada, utilizará o mesmo banco de dados para armazenar solicitações de usuários e guardar o estado dos *jobs*, que é utilizado pelos programas que simulam as submissões de usuário.

Capítulo 5

Ferramenta OSCAR-PBS

O presente capítulo tem o objetivo de explicar o desenvolvimento e o funcionamento do OscarPBS bem como ilustrar com exemplos o funcionamento de um método desenvolvido nesse trabalho e utilizado pelo escalonador do OscarPBS, que será chamado de método *Empurrar*. O capítulo se encontra organizado nas seguintes seções:

A primeira seção mostra um breve histórico que motivou o desenvolvimento do OscarPBS e da criação do método *Empurrar*. A segunda seção dá uma visão geral do funcionamento do OscarPBS, explicando de modo resumido seu método principal e os métodos importantes para o entendimento de como os *jobs* são alocados. A terceira seção ilustra o funcionamento do método *Empurrar* com alguns exemplos que mostram como ele melhorar a alocação de *jobs* em máquinas do *cluster*. A quarta seção ilustra as estruturas de dados principais e o estados dos *jobs*, que são explicadas tendo o objetivo de ajudar no entendimento do funcionamento do OscarPBS. Detalhes sobre a requisição de *jobs* e a estrutura do Banco de Dados utilizado pelo OscarPBS também são mostrados nesse capítulo. Por último, a quinta seção explica em detalhes o algoritmo de escalonamento do OscarPBS, iniciando pelo seu método principal e mostrando os algoritmos de rotinas importantes na medida em que elas são referenciadas no texto.

5.1 Um Breve Histórico

O *cluster* Oscar [11], adquirido pelos Institutos de Computação, Física e Química da UFF em 2008, trouxe com ele uma licença de software *D-RMS* para o gerenciamento e escalonamento de *jobs* chamado de PBS Professional 10.1. Esse sistema possibilitava o escalonamento de *jobs* de usuários no *cluster*, com base em filas de prioridades diferentes e com *jobs* de cada fila ordenados de acordo com o tempo de CPU estimado. Cada fila

possuía um limite de tempo para os *jobs*, sendo que as filas de tempo menor possuíam maior prioridade. Havia uma fila expressa com prioridade ainda maior e oito máquinas eram deixadas livres para algum uso emergencial.

O PBS Professional também trazia algumas características, como tratamento de *jobs* starve após um determinado período de tempo, com possibilidade de *backfilling*, possibilidade de reserva de recursos e permitia ao administrador utilizar diferentes funções de peso para a escolha de que *job* seria alocado primeiro dentro de uma fila, além de uso de *fairshare*.

Apesar das funcionalidades apresentadas, foi possível notar que o escalonamento feito apenas com base em funções de peso dos *jobs*, fazia escolhas precipitadas com relação a como as máquinas eram utilizadas, o que sugeria que uma heurística que considerasse além do peso, poderia fazer uso mais eficiente das máquinas do *cluster*.

Depois dos primeiros testes com o PBS Professional buscou-se um escalonador semelhante que fosse livre para teste e uso em outras máquinas sem precisar adquirir a licença. O escalonador TORQUE foi o escolhido, ele é semelhante ao PBS Professional, já que ambos são sucessores do OpenPBS. A maior diferença do escalonador TORQUE em relação ao PBS Professional está no escalonador, devido ao suporte a *backfilling* para *jobs* em *starve*, que não está presente no TORQUE. O TORQUE então passou a ser experimentado em outro *cluster*, o *cluster* RIO e depois passou a ser testado também no Oscar, passando a substituir o PBS Professional quando a licença do mesmo expirou.

O escalonador TORQUE é disponibilizado com o código fonte e sua documentação é a mesma que a das ultimas versões do OpenPBS. O OpenPBS traz a ideia de ser principalmente um gerenciador de recursos distribuídos, com um escalonador incluído, mas que é simples de substituir. A documentação do OpenPBS e seu código fonte traz detalhes de como o escalonador pode ser substituído, trazendo também exemplos de outros escalonadores. Enquanto uma versão instalada do TORQUE era utilizada no *cluster* Oscar para o escalonamento dos *jobs* dos usuários, duas novas instalações de TORQUE foram feitas no *cluster*, com a intenção de serem utilizadas neste trabalho. Dessas duas instalações, uma permaneceria inalterada e outra teria o código do módulo escalonador modificado. A instalação modificada, teria uma heurística visando utilizar não apenas o peso mas também observar os *jobs* da fila e permitir modificações buscando um melhor encaixe de *jobs* antes de colocar os *jobs* em execução.

As primeiras versões do OscarPBS foram implementadas dentro do próprio TORQUE utilizando a linguagem C. A ideia principal era ter acesso a todos os eventos que o

pbs_server passava ao escalonador e poder enviar comandos ao módulo pbs_server e pbs_mons utilizando as bibliotecas da própria linguagem. Tal abordagem levava a necessidade de recompilar e reinstalar o TORQUE a cada mudança. Com isso, naturalmente se desejou separar o código fonte do escalonador do código fonte do TORQUE, a ponto de não precisar recompilar o TORQUE a cada mudança ou correção do escalonador. Foi criado um módulo simples, que lê um arquivo de configuração e a partir dele, decide para qual programa mandar um sinal, cada vez que recebe um evento do escalonador. O escalonador principal ficou independente do código do TORQUE, tornando o desenvolvimento simples. O escalonador passou a ser implementado na linguagem Python [18] em uma módulo completamente separado do TORQUE. Tal escalonador acessa um banco de dados PostgreSQL [20] para buscar as requisições de usuário e recebe eventos do TORQUE para término de *job* em conjunto com eventos vindos da interface do usuário para submissão de *jobs*.

5.2 Visão geral do Escalonador

O OscarPBS tem a função de fazer o escalonamento de *jobs* em um *cluster*, estando integrado a um software *D-RMS* chamado de TORQUE. O OscarPBS classifica os *jobs* em quatro tipos diferentes. Ele utiliza um método desenvolvido nesse trabalho chamado de *Empurrar* no escalonamento de *jobs*, que tem o objetivo melhorar o desempenho do escalonamento. O método *Empurrar* funciona em conjunto com outros métodos de escalonamento também implementados no OscarPBS e que são utilizados em outros escalonadores da família PBS.

O OscarPBS é composto dos módulos de interface com o usuário, banco de dados e escalonador mas o foco principal deste capítulo será o módulo escalonador. O termo OscarPBS estará se referindo principalmente ao módulo de escalonamento desenvolvido nesse trabalho.

O escalonador OscarPBS foi idealizado de modo a poder receber em qualquer momento requisições de *jobs* de usuários e tratar os *jobs* no momento em que eles chegam, reservando espaço em máquinas para esses *jobs* e colocando em execução imediatamente se for possível. De maneira semelhante, ele foi idealizado para tratar o término de *jobs* no momento em que esses acontecem, de modo a poder liberar imediatamente as máquinas reservadas ao *job* que estava em execução e poder alocar *jobs* que não estavam alocados por falta de máquinas e também se possível realocar *jobs* para que possam ser executados

imediatamente.

O escalonador então trabalha quando acontece algum evento, de chegada ou término de *job* e depois de verificar todos os *jobs* de suas filas possivelmente tratar alguns deles, o escalonador entra em suspensão esperando pela ocorrência de um novo evento. O trabalho feito pelo escalonador entre a ocorrência de dois eventos será chamado de ciclo de escalonamento. Na implementação a ocorrência de eventos é representada pelo envio de sinais SIGCONT do padrão POSIX para o módulo escalonador. O escalonador fica suspenso aguardando pela chegada desse tipo de sinal.

Assim, de forma simplificada, a rotina do escalonador OscarPBS pode ser descrita como no algoritmo 1:

```
1 enquanto True faça
2     permanecer_suspenso_esperando_por_evento();
3     ciclo_de_escalonamento();
4 fim
```

Algoritmo 1: Loop de eventos e rotina de um ciclo de escalonamento

Além dos eventos gerados pela chegada do sinal SIGCONT, o escalonador também controla o tempo que ficará suspensão de modo a não depender somente de eventos externos para realizar um ciclo de escalonamento. A suspensão é sempre limitada por uma quantidade de tempo máxima pré-definida, ou pode se basear no final estimado de *job* mais próximo ou no início de um *job* reservado. Os eventos de chegada de *job* são gerados pelo módulo de interface com os usuários, quando esses submetem novos *jobs*. A interface com o usuário, guarda a requisição do usuário no banco de dados do OscarPBS e após isso envia um sinal SIGCONT ao módulo escalonador, fazendo que ele saia da suspensão e inicie um novo ciclo de escalonamento imediatamente se estiver suspenso. Se o escalonador estiver em um ciclo de escalonamento no momento em que o sinal é enviado, o sinal fica retido até que o escalonador termine o ciclo atual. Assim que o escalonador tentar entrar em suspensão para esperar o próximo sinal, ele receberá o sinal retido e entrará em um novo ciclo escalonamento imediatamente. Os términos de *jobs* também geram sinais que fazem o escalonador entrar em um novo ciclo de escalonamento. Os *jobs* em execução são monitorados pelo módulo de gerência de *jobs* do TORQUE chamado de pbs_server. Quando um *job* termina, o pbs_server provoca a execução de uma rotina de escalonamento de seu módulo de escalonamento nativo o pbs_sched. No TORQUE integrado ao OscarPBS, o módulo pbs_sched do TORQUE foi modificado de forma que quando ele seja executado pelo pbs_server ele envie um sinal SIGCONT para ao módulo

escalonador do OscarPBS.

Ao entrar em um ciclo de escalonamento, o OscarPBS realiza alguns passos que serão explicados resumidamente nos passos do Algoritmo 2. Mais detalhes sobre o funcionamento do ciclo de escalonamento serão mostradas na seção final deste capítulo.

```
algoritmo ciclo_de_escalonamento()
1 Passo 1. Verificar se existem em execução jobs para forçar o término;
2 Passo 2. Verificar se jobs que estavam em execução terminaram sozinhos;
3 Passo 3. Alocar jobs Emergência que não estejam alocados;
4 Passo 4. Alocar jobs QoS que não estejam alocados;
5 Passo 5. Alocar todos os jobs Starve, procurando alocar os mais antigos para a
  execução imediata;
6 Passo 6. Tentar alocar jobs Comuns para execução imediata;
7 Passo 7. Tentar alocar jobs QoS e Emergência para a execução imediata;
8 Passo 8. Executar os jobs alocados para execução imediata;
fim
```

Algoritmo 2: Algoritmo com os passos de um ciclo de escalonamento

Passo 1: O escalonador verifica se há *jobs* em execução que devem ser finalizados, isso ocorre se algum *job* em execução ultrapassou o tempo solicitado, em outras palavras, ele ultrapassou o *walltime* solicitado pelo usuário mais uma tolerância dada pelo OscarPBS.

Passo 2: O escalonador verifica se algum *job* que estava em execução no ciclo de escalonamento anterior deixou de estar executando. Esses *jobs* ocupam recursos nas estruturas de dados do escalonador OscarPBS mas os recursos reais das máquinas do *cluster* já foram liberadas. Assim a intenção desse passo é liberar os recursos nas estruturas de dados do OscarPBS para que fique sincronizado com os recursos reais. O estado das máquinas também é sincronizado, de modo que o escalonador saiba quais máquinas estão disponíveis.

Passo 3: O escalonador tentará alocar quaisquer *jobs* de Emergência novos ou que não puderam ser alocados em ciclos anteriores. Esse passo visa alocar os *jobs* de Emergência dentro de seu prazo e garantir que eles possuam seu lugar reservado quando o escalonador tentar alocar outros tipos de *job*. Aqui o escalonador tenta alocar os *jobs* o mais tarde possível, tendo em vista obedecer o prazo dos *jobs* mas tentando escolher uma posição no tempo que seja o mais distante possível do instante atual.

Passo 4: Agora o escalonador tentará alocar os *jobs* QoS novos e aqueles que estavam desalocados por não conseguirem ser alocados no ciclo de escalonamento anterior. De forma semelhante aos *jobs* Emergência, o escalonador tentará alocar cada *job* QoS o mais tarde possível, colocando o *job*, de preferência, próximo a seu prazo. A ideia é garantir

a alocação em uma posição mais distante possível do instante atual, por essa posição ser provavelmente menos concorrida.

Passo 5: O escalonador tentará alocar todos os *jobs Starves* obedecendo a ordem em que foram requisitados pelo usuário (*FIFO*). Inicialmente o escalonador tentará alocar qualquer *job Starve*, desde que ele não esteja em execução, no instante atual para que o mesmo seja executado no final do ciclo de escalonamento. Nessa tentativa inicial, o *job Starve* terá poderes para desalocar, caso seja necessário, outros *jobs Starve* que tenham sido submetidos a menos tempo do que ele. Se não for possível alocar o *job Starve* no instante atual e se ele já não estiver alocado em outra posição, o escalonador tentará alocar o *job*, procurando posições cada vez mais no futuro até conseguir uma onde o *job Starve* possa ser alocado. A procura prossegue nem que para isso o *job Starve* precise ser alocado depois do final do último *job*. Um *job Starve* só não conseguirá ser alocado se todas as máquinas disponíveis no *cluster* não puderem atende-lo no momento. A ideia desse passo é tentar alocar todos os *jobs Starve* antes que o escalonador tente colocar *jobs* Comuns em execução.

Passo 6: O escalonador tentará alocar *jobs* Comuns no instante mais cedo possível. Os *jobs* são ordenados pelo seu peso, que de modo simplificado, representa o número de CPUs requisitadas para o *job* vezes tempo de execução estimado pelo usuário. Os *jobs* que tiverem peso menor serão experimentados primeiro.

Passo 7: O escalonador tentará realocar *jobs* QoS e Emergência alocados no instante atual. Esses *jobs* já tem sua posição garantida, mas se puderem ser executados imediatamente o escalonador tentará fazer isso.

Passo 8: O escalonador colocará em execução todos os *jobs* que estejam alocados no instante atual. Para cada um desses *jobs*, o escalonador prepara uma requisição para o TORQUE definindo todas as máquinas que devem ser usadas pelo *job* e o *walltime* do *job*. As máquinas que serão usadas pelo TORQUE são as mesmas que foram definidas pelo escalonador OscarPBS no momento da alocação do *job*. O TORQUE fica encarregado de colocar o *job* em execução nas máquinas certas e vigiar o término do *job*. O escalonador OscarPBS utiliza os comandos *qsub* e *qrun* do TORQUE para submeter e colocar em execução cada *job* alocado no instante atual.

Quando o escalonador decide tentar alocar um *job*, como está descrito nos passos do ciclo de escalonamento acima, ele precisa tomar algumas decisões, como por exemplo qual instante de tempo deve coincidir com o início do *job* e em que máquinas o *job* deve ser alocado. Essas decisões com relação ao instante de tempo escolhido e máquinas escolhidas

serão descritas brevemente a seguir, porém uma descrição mais detalhada será apresentada nas seções seguintes.

Escolha de uma posição no tempo para tentar inserir o job:

Em algumas situações determinar o instante de tempo onde o *job* deve ser alocado é trivial, como por exemplo, quando o escalonador tenta alocar um *job* no instante de tempo atual, ou quando o escalonador tenta alocar um *job* Emergência adjacente ao seu prazo (essa é uma ocasião especial da inserção de um *job* Emergência, vista nas seções seguintes). Em outras situações, um instante de tempo precisa ser determinado antes que o escalonador possa tentar inserir o *job*. Para determinar tal instante de tempo o escalonador trabalha em duas etapas. Inicialmente, o escalonador determina um intervalo de tempo onde um instante de tempo que coincida com o inicio do *job* possa ser procurado. Por exemplo, quando o escalonador tenta alocar um *job* QoS, o intervalo inicial vai do instante de tempo atual até o prazo do *job* QoS. No caso de um *job* Starve, esse intervalo vai do instante atual até o último instante de tempo que determine o fim de algum *job* alocado ou em execução. Uma vez definido o intervalo, o escalonador procura nesse intervalo uma posição onde o *job* tenha a possibilidade de ser inserido (a busca dessa posição de tempo faz uma verificação prévia dos totais de CPUs disponíveis na posição de tempo sem garantir no entanto que o *job* possa ser inserido porque a inserção não depende apenas do número de CPUs disponíveis, mas como as mesmas estão organizadas nas máquinas) Se a busca por um instante inicial naquele intervalo falhar, o escalonador desiste de escalonar o *job* naquele ciclo. Por outro lado se for possível encontrar um instante de tempo no intervalo, o escalonador tenta alocar o *job* nessa posição, podendo ter sucesso ou não. Quando o escalonador não consegue inserir o *job* na posição encontrada, o intervalo de tempo onde a busca é feita é reduzido, de maneira a não incluir mais o instante de tempo que falhou. Se o intervalo reduzido ainda comportar o *walltime* do *job*, uma nova busca por um instante de tempo é feita nesse novo intervalo. As tentativas são repetidas se necessário, até que o *job* não caiba mais no intervalo e nesse caso o escalonador desiste de alocar o *job* naquele ciclo de escalonamento.

Escolha das máquinas a serem utilizadas por um job:

Uma vez definido um instante de tempo que deva coincidir com o início do *job* a ser alocado, o escalonador chamará a rotina inserir_job, que será explicada em mais detalhes nas seções seguintes, para tentar alocar o *job* naquele instante de tempo. Nessa etapa do escalonamento ainda não há garantia que o *job* possa ser inserido nessa posição de tempo que foi definida, pois a posição de tempo é escolhida apenas considerando o total

de CPUs livres e desalocáveis naquele espaço de tempo, sem considerar como essas CPUs estão organizadas nas máquinas. O termo CPUs desalocáveis utilizado aqui representa os totais de CPUs de *jobs* que podem ser desalocados durante a tentativa de inserção do *job* a ser inserido. Um *job* pode ter o poder de desalocar outros *jobs* como será visto com mais detalhes nas seções seguintes. Caberá a rotina inserir_job tentar inserir o *job* nesse intervalo de tempo definido.

A inserção do *job* em si depende da inserção de todos os fragmentos desse *job*. Um *job* é composto de um ou mais fragmentos e esses fragmentos do *job* devem cada um ocupar uma determinada máquina, havendo a possibilidade de uma só máquina possuir mais que um fragmento de um mesmo *job*. A rotina de inserir_job então precisa inserir todos os fragmentos desse *job*, para que a mesma considere que a inserção foi bem sucedida. Caso a rotina não possa inserir algum dos fragmentos do *job*, ela deve desalocar os fragmentos que possam ter sido inseridos e retornar fracasso na inserção do *job*. Durante a inserção de um *job* a rotina inserir_job realiza algumas tarefas com a intenção de inserir cada um dos fragmentos desse *job*. Primeiro ela tenta inserir todos os fragmentos que já possuem máquinas definidas, ou seja, os fragmentos fixos. Depois, se conseguir inserir todos os fixos, ela tenta inserir os fragmentos livres.

Durante a inserção dos fragmentos fixos, uma rotina é chamada para verificar a situação do fragmento em relação a sua máquina. Três situações podem ocorrer. Numa delas que será chamada de *cabe*, a máquina pode aceitar o fragmento sem que seja preciso liberar nenhum espaço. Em outra, chamada de *viável*, a máquina pode aceitar o fragmento, porém há um ou mais fragmentos que precisam ser movidos para outra máquina, que também serão designados como sendo fragmentos *empurráveis*, e/ou desalocados, antes que o fragmento possa ser inserido. Numa terceira situação, chamada de *inviável*, o fragmento não pode ser inserido na máquina, porque já existem um ou mais fragmentos na máquina que coincidam com o fragmento a ser inserido e esses fragmentos que ocupam a máquina não podem ser movidos nem desalocados.

Se todos os fragmentos fixos foram alocados, o escalonador tentará alocar os fragmentos livres. Para cada fragmento livre, a rotina inserir_job tentará encontrar uma máquina *bestfit*, chamando a função encontra_bestfit. Essa função verificará todas as máquinas disponíveis, determinando as máquinas em que o fragmento *cabe*, ou seja as máquinas em que não seja preciso liberar espaço, enquanto que também calcula quanto espaço existe livre nas máquinas do tipo *cabe*. Dentre as máquinas *cabe*, se houver alguma, a *bestfit* será a que tiver menor espaço sobrando dentro do intervalo de tempo que o fragmento a

ser inserido na máquina vai ocupar. Não havendo máquina *bestfit*, a inserir_job chama uma rotina para gerar um lista de máquinas *viáveis*. Nas máquinas *viáveis*, é possível inserir o fragmento desde que antes se mova alguns fragmentos para outras máquinas, ou se desaloque alguns fragmentos. A lista de máquinas viáveis, se houver alguma, é ordenada de modo crescente pela categoria da máquina, sendo essas categorias de máquina, que serão vistas com mais detalhes nas próximas seções, funcionam como um peso calculado a partir dos tipos de *job* que estão alocados na máquina. Havendo empate de categoria, as máquinas *viáveis* que tiverem menos espaço a liberar são escolhidas primeiro. A categoria da máquina visa evitar que *jobs* sejam desalocados sem necessidade. Em outras palavras, as máquinas onde não há possibilidade de deslocar nenhum tem a menor categoria possível (nessa máquina existem apenas fragmentos *empurráveis*) e as máquinas com vários *jobs* importantes com a possibilidade de serem desalocados tem categoria maior.

Escolhida um máquina viável da lista de máquinas, uma rotina chamada *liberar_ espaco(fragmento,maquina)* é chamada com a intenção de empurrar fragmentos e, em último caso, desalocar fragmentos até que a máquina se torne uma máquina onde o fragmento a ser inserido caiba. Se a rotina liberar_espaco falhar para uma máquina *viável*, as outras máquinas viáveis da lista serão verificadas.

Por fim o fragmento livre pode ter conseguido uma máquina *bestfit*, ou caso contrário, conseguido liberar espaço em alguma máquina viável. Se nenhuma das duas situações anteriores acontecer, a inserção do fragmento livre falhou e a rotina inserir_job irá parar de tentar inserir os fragmentos que restam e desalocar os fragmentos que tenha alocado.

A rotina liberar_espaco:

Como visto anteriormente, a escolha do instante de tempo onde o *job* será inserido, seguida da escolha das máquinas onde o *job* será inserido, ou melhor, das máquinas onde cada fragmento do *job* será inserido, são etapas que envolvem a alocação de um *job*. Dentro da escolha das máquinas para cada fragmento, se o fragmento não cabe na máquina, seja o fragmento fixo ou livre, uma rotina chamada liberar_espaco é chamada para tentar transformar uma máquina que é apenas *viável* em uma máquina do tipo *cabe*.

Esse método pode ser descrito como possuindo duas etapas. Na primeira etapa, ele tentará mover todos os fragmentos *empurráveis* para máquinas vizinhas, onde fragmentos empurráveis são aqueles que não são fixos e que não são de *jobs* em Execução. Se não conseguir liberar o espaço necessário apenas empurrando fragmentos, a liberar_espaco passa a desalocar *jobs* que possuem algum fragmento que ocupe a máquina, até que a máquina tenha espaço suficiente.

É importante salientar que os fragmentos que são movidos ou desalocados por essa rotina de alguma maneira coincidem com o fragmento a ser inserido, ou seja, competem pelas mesmas CPUs, em um intervalo de tempo comum entre os fragmentos a ser inserido e o a ser movido ou desalocado.

O rotina liberar_espaco eventualmente pode não conseguir liberar o espaco, quando algum fragmento empurrável não puder ser empurrado para outra máquina e esse mesmo fragmento não puder ser desalocado e não houver outros fragmentos a empurrar ou excluir que possam compensar o fragmento que não pode ser empurrado.

A rotina empurrar_fragmento:

A última rotina importante a ser explicada resumidamente nessa seção será empurrar_fragmento. Ela tem grande importância no desempenho do escalonador buscando utilizar o método *Empurrar* explicado nas seções seguintes, para aproveitar melhor os recursos da máquinas do *cluster*.

Essa rotina tem o objetivo de retirar um fragmento da máquina onde ele está alocado atualmente, movendo o fragmento para alguma máquina vizinha. O fragmento a ser empurrado precisa ser necessariamente um fragmento livre e de um *job* que não esteja em execução ainda. Antes de empurrar o fragmento a rotina precisa encontrar alguma máquina disponível no *cluster* em que o fragmento caiba sem precisar liberar espaço da máquina. Para isso essa rotina chama a função encontra_bestfit passando um parâmetro que indica que a máquina onde o fragmento já está não participa da escolha e máquinas onde existe algum problema de exclusividade de fragmento (com outros fragmentos do mesmo *job*) não participam. Encontrada uma máquina bestfit, a rotina move o fragmento para essa máquina bestfit e retorna *True* para a liberar_espaco. Do contrário retorna False para a rotina liberar_espaco.

5.3 O método *Empurrar*

O método *Empurrar* foi desenvolvido nesse trabalho com o objetivo de explorar situações em que os métodos de escalonamento utilizados pelos *D-RMS* da família PBS deixam recursos ociosos e *jobs* sem poder alocar. Devido a isso, com o uso do método *Empurrar*, se espera que o OscarPBS obtenha melhor aproveitamento de recursos do *cluster* que os escalonadores dos *D-RMS* sucessores do OpenPBS.

Nessa seção o funcionamento do método *Empurrar* será ilustrado com alguns exem-

plos de submissões de *jobs*, semelhantes aos que acontecem durante o escalonamento de *jobs* reais no *cluster* Oscar. Os exemplos mostram situações em que o método consegue alocar *jobs* Candidatos nos recursos do *cluster* onde as outros rotinas do escalonador não poderiam mais fazer nada com relação ao *job* Candidato e o deixariam esperando por ciclos futuros, mesmo havendo recursos que pudessem atender ao *job*.

5.3.1 Exemplo com *jobs* Comuns Candidatos disputando pelo instante atual

O primeiro exemplo ilustra uma situação em que existem quatro *jobs* Comuns Candidatos em espera, quando acontece um evento de término de *job* que libera duas máquinas do *cluster* (ver Figura 5.1), que serão representadas por Máq1 e Máq2.

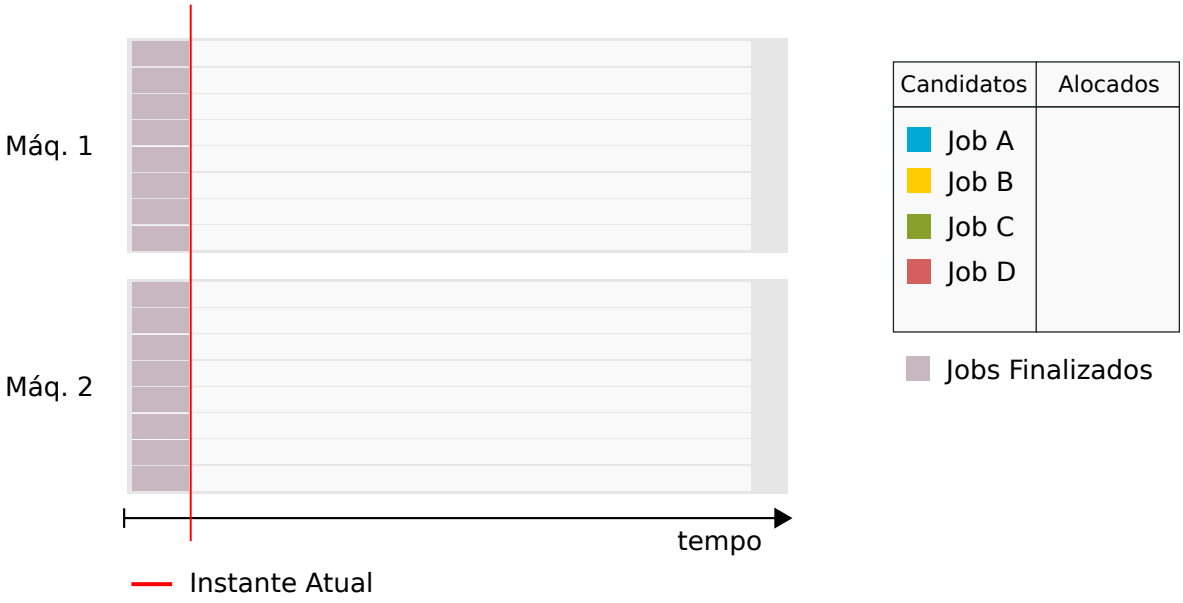

Figura 5.1: Duas máquinas do *cluster* Máq1 e Máq2 ficam vazias após um evento de finalização de um *job*, quando existem quatro *jobs* Comuns Candidatos representados por A,B,C e D.

Com o evento, o OscarPBS entra em um ciclo de escalonamento e vai tentar alocar todos os quatro *jobs*, quando chegar na rotina que trata de *jobs* Comuns. Antes de tratar os quatro *jobs*, o escalonador ordena esses *jobs* pelo peso, definido na seção seguinte e segue tentando alocar cada um deles, procurando máquinas onde o *job* caiba e que deixe o mínimo de espaço livre dentro do intervalo que o *job* ocupa. Os quatro *jobs* aqui representados por A,B,C e D são todos compostos de apenas um fragmento, o que significa que o sucesso na inserção desses *jobs* depende do sucesso da inserção de seu único fragmento. O escalonador na verdade aloca e empurra os fragmentos dos *jobs*, mas para os

jobs com um único fragmento em algumas ocasiões será dito que o escalonador empurrou o *job* ou alocou o *job*.

Os *jobs* deste exemplo tem as seguintes características:

- **Job A**: Um fragmento de 2 CPUs, *walltime* de 100 segundos. Peso total: 200.
- **Job B**: Um fragmento de 3 CPUs, *walltime* de 150 segundos. Peso total: 450.
- **Job C**: Um fragmento de 5 CPUs, *walltime* de 200 segundos. Peso total: 1000.
- **Job D**: Um fragmento de 6 CPUs, *walltime* de 250 segundos. Peso total: 1500.

Onde a unidade de peso é **CPU x segundo**. O escalonador insere primeiro o fragmento do *job* A, por esse *job* ser o de menor peso e como a Máq1 tem o mesmo espaço da Máq2, o escalonador usou a primeira encontrada que no caso foi a máq1. Logo em seguida insere o *job* B, na mesma máquina que o *job* A devido a escolha de máquina *bestfit*, ver Figura 5.2.

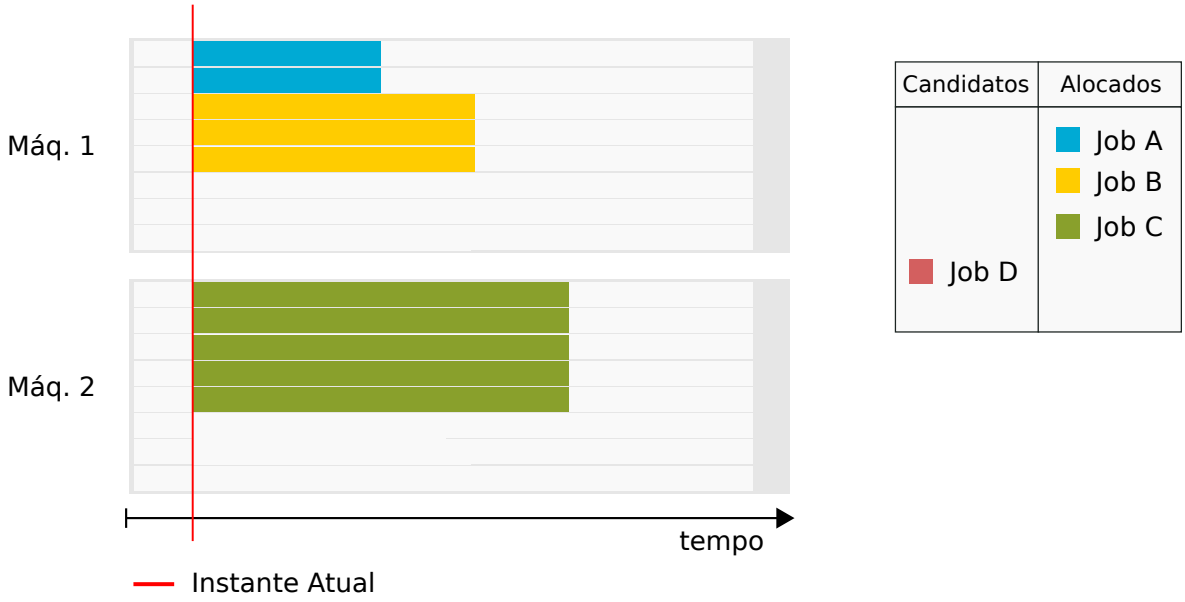

Figura 5.2: O escalonador consegue alocar os *jobs* A e B na Máq1 e o *job* C na Máq2 usando funções de peso para os *jobs* e *bestfit* para máquinas porém o *job* D fica sem espaço.

Agora o escalonador tenta inserir o *job* C, encontrando a Máq2 como a única em que o *job* *cabe*. Quando vai inserir o *job* D, o escalonador não encontra máquinas onde possa alocar esse *job*, mas sabe que o sistema possui espaço ocioso suficiente e que se puder reorganizar os *jobs* talvez o *job* D possa ser encaixado. Esta situação é resolvida

caso pelo método *Empurrar*. O escalonador vai procurar todas as máquinas viáveis para inserir o *job* D e descobre que as máquinas Máq1 e Máq2 são viáveis. Dentre as viáveis, aquela que tiver menor espaço a ser liberado é tentada primeiro, que no caso é a Máq1. Escolhida a máquina onde o escalonador vai liberar o espaço, os fragmentos *Empurráveis* são determinados e ordenados pelo pelo quando cada um pode liberar de espaço. Após feita a seleção e ordenação dos fragmentos o escalonador decide empurrar o fragmento do *job* B. O método *Empurra* então encontra a Máq2. como sendo a *bestfit* onde o fragmento do *job* B pode ser inserido e insere o fragmento nessa máquina ver Figura 5.3.

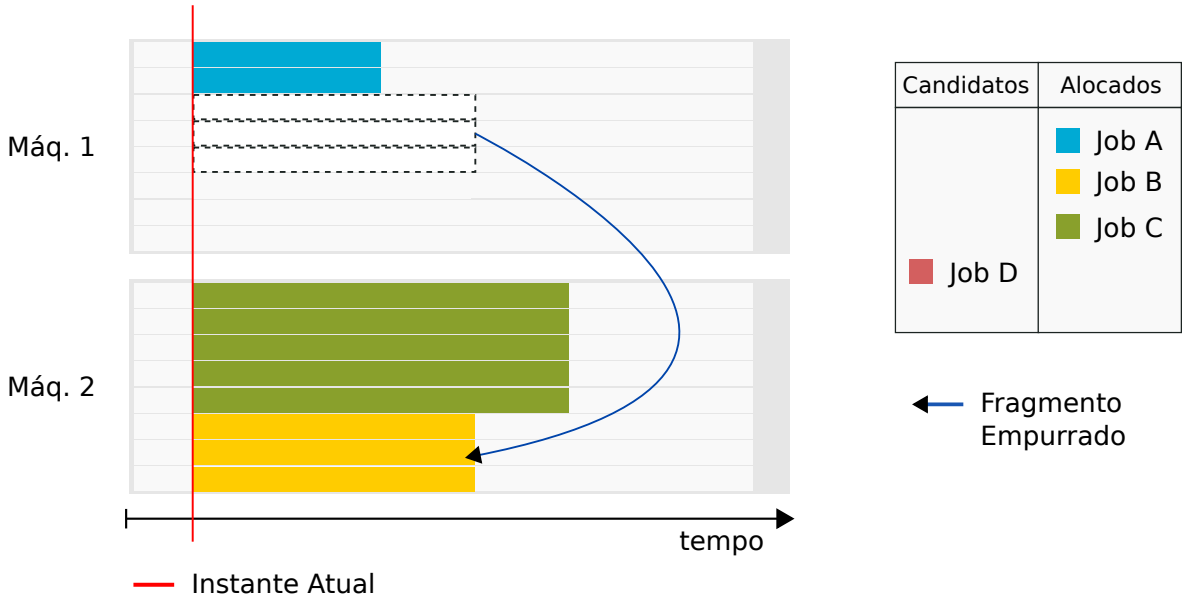

Figura 5.3: O método *Empurrar* move o fragmento do *job* B, a fim de liberar espaço na Máq1. para que o fragmento do *job* D possa ser inserido

Após esse processo, a máquina Máq1 passa a ter espaço suficiente para abrigar o fragmento do *job* D, então a tentativa de liberar espaço termina e o *job* D é inserido na Máq1 ver Figura 5.4.

5.3.2 Exemplo de dois usuários submetendo dois *jobs*

Esse exemplo ilustra uma situação que é acontece no escalonamento de *jobs* reais. Um usuário submete dois *jobs* com a mesma solicitação de CPUs e *walltime*, enquanto um outro usuário também submete outros dois *jobs* com mesma solicitação de CPUs e *walltime*.

De modo similar ao exemplo anterior, um evento de término de *job* ocorre causando a liberação duas máquinas quando existem 4 *jobs* E, F, G e H Comuns Candidatos na fila.

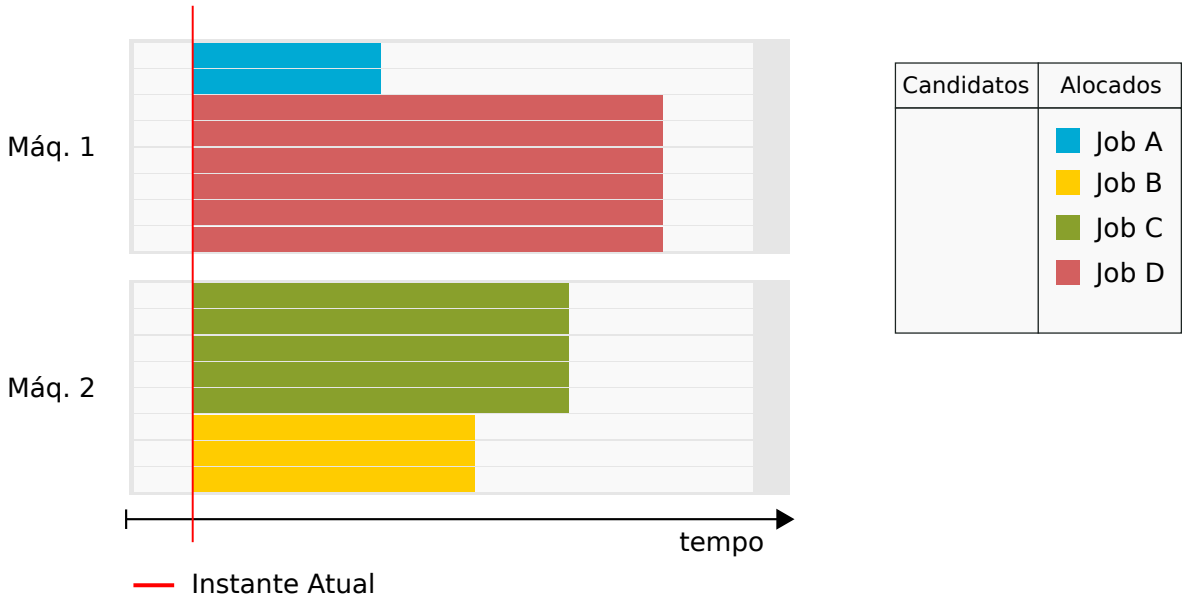

Figura 5.4: Agora com uma nova configuração na alocação dos *jobs*, foi possível inserir o *job* D na Máq1

Ver Figura 5.5.

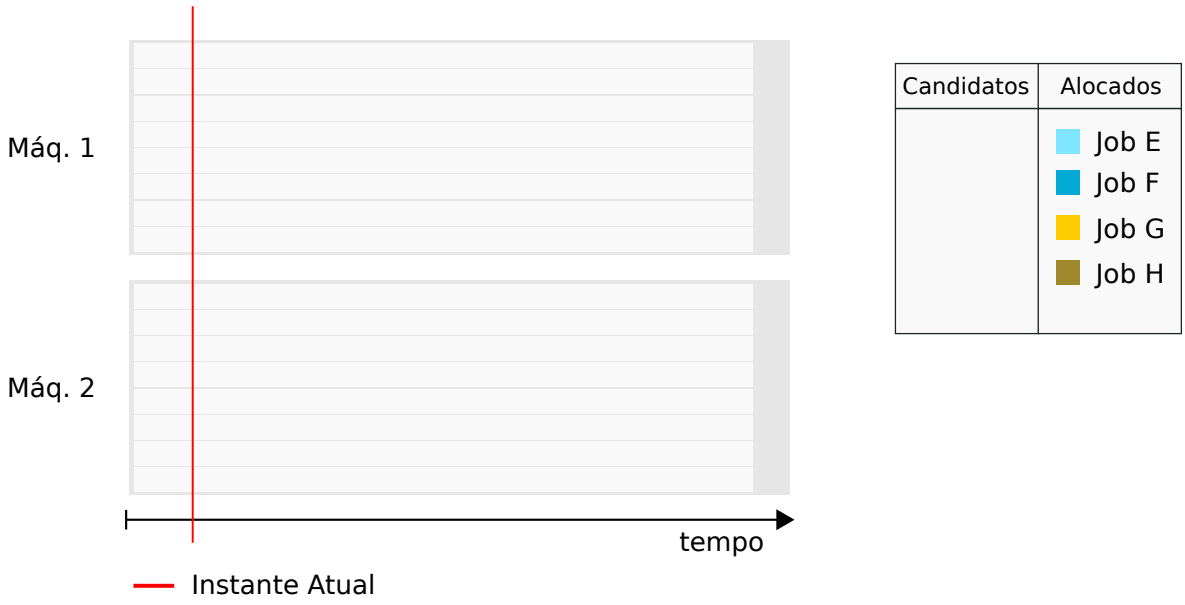

Figura 5.5: Duas máquinas ficam livres após um evento de término de *job*, enquanto existiam 4 *jobs* Comuns a espera de recursos

Os *jobs* E e F são submetidos por um usuário, possuindo as mesmas características quanto ao a número de CPUs e o *walltime* requisitado. Os *jobs* G e H são submetidos por outro usuário, sendo que esses também possuem características semelhantes quanto aos recursos solicitados porém sendo diferentes das dos *jobs* E e F.

Os *jobs* deste exemplo tem as seguintes características:

- **Job E**: Um fragmento de 2 CPUs, *walltime* de 200 segundos. Peso total: 400.
- **Job F**: Um fragmento de 2 CPUs, *walltime* de 200 segundos. Peso total: 400.
- **Job G**: Um fragmento de 6 CPUs, *walltime* de 280 segundos. Peso total: 1680.
- **Job H**: Um fragmento de 6 CPUs, *walltime* de 280 segundos. Peso total: 1680.

Os *jobs* E é escolhido primeiro devido a seu peso menor e alocado na Máq1. O *job* F é selecionado logo em seguida, sendo alocado também na Máq1 devido a política *bestfit*. O *job* G é selecionado e alocado na Máq2, uma vez que ele cabe nessa máquina. O *job* H é então selecionado, mas o escalonador não consegue encontrar uma máquina onde ele caiba, chegando a situação mostrada pela Figura 5.6.

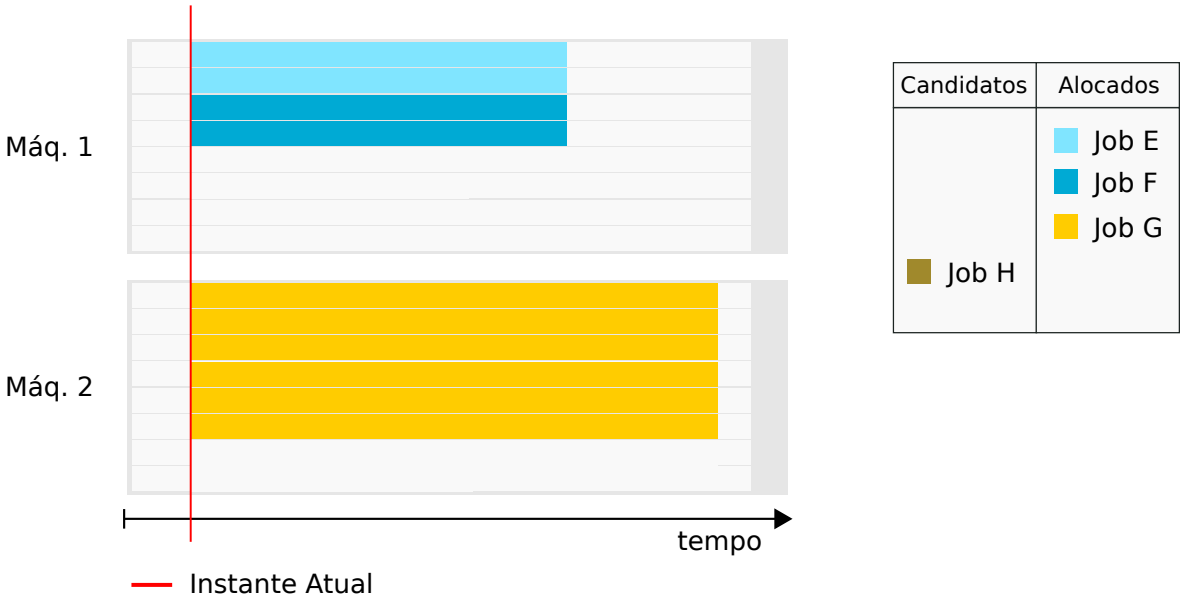

Figura 5.6: Os *jobs* E e F de um mesmo usuário são alocados na Maq1, o *job* G de outro usuário é alocado porém o *job* H deste segundo usuário fica sem lugar para ser inserido

Com isso, a rotina que insere *jobs* procura por máquinas viáveis para o *job* H, onde possa utilizar a rotina que libera espaço nessas máquinas. As máquinas Maq1 e Maq2 são viáveis, sendo que a Máq1 é escolhida primeiro porque nela será preciso liberar menos recursos. A rotina que libera espaço então escolhe o fragmento do *job* F para empurrar primeiro e chama a rotina que executa o método *Empurrar*, procurando uma máquina *bestfit* para o fragmento do *job* F. A Máq2 é então escolhida e o fragmento do *job* F é empurrado para ela como ilustrado na Figura 5.7.

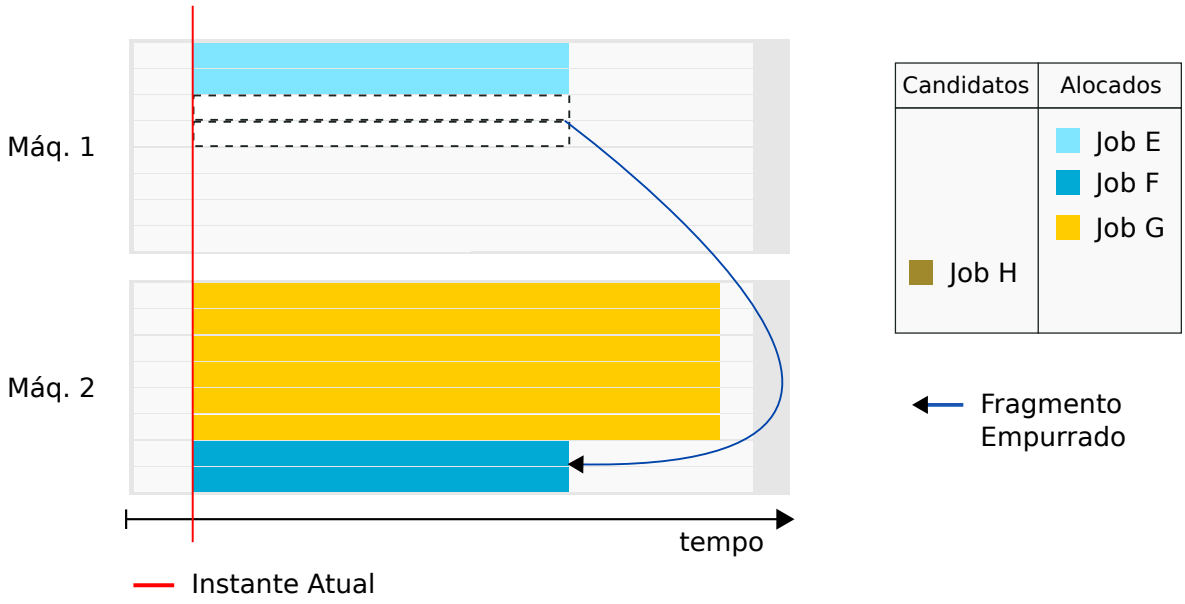

Figura 5.7: A função que insere *jobs* usa o método *Empurrar* no *job* F, a fim de liberar espaço na Máq1 e poder inserir o *job* H

Após ter empurrado o fragmento do *job* F, a rotina que libera espaço conseguiu espaço suficiente para inserir o fragmento do *job* H na Máq1. Por isso a rotina termina e o fragmento é inserido na Máq1, o que equivale a conseguir inserir o próprio *job* H. A inserção do *job* H pode ser vista na Figura 5.8.

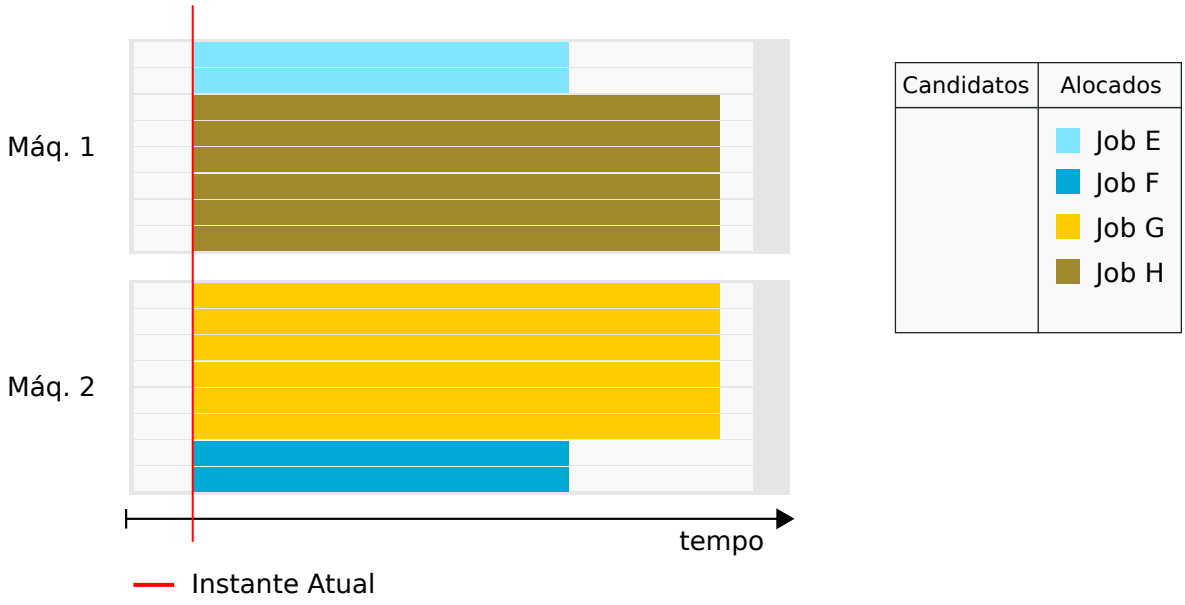

Figura 5.8: Após liberar espaço suficiente na máq1, a rotina que insere *jobs* consegue inserir o *job* H

5.3.3 Exemplo de alocação de *job* Comum após um *job* QoS ser empurrado

Neste exemplo, é ilustrada uma situação em que um *job* Comum de baixa prioridade pode ser alocado após um *job* QoS ter sido alocado antes e garantido um espaço antes de seu prazo. Situações semelhantes a desse exemplo podem acontecer com outros *jobs* de baixa prioridade, quando o escalonador tenta alocar esses *jobs* entre *jobs* já alocados de prioridade maior. Situações parecidas também podem acontecer quando o escalonador tenta inserir um *job* Comum entre *jobs Starves* já alocados *Backfilling* desde que os *jobs Starve* não sejam atrasados com relação a posição no tempo que estão ocupando.

O *job* L, de peso igual a 1000 **CPUs x segundos**, é submetido por um usuário, provocando um evento de chegada de *job*. Quando o escalonador tenta inserir o *job* L, consegue localizar uma única máquina viável máq1 onde encontra um *job* I em execução e um *job* K QoS alocado numa posição adjacente a seu prazo. O *job* K impede a inserção do *job* L na Máq1, mas o *job* K é empurrável e existe uma outra máquina Máq2 que pode comportar o *job* K como mostrado na Figura 5.9.

Os *jobs* deste exemplo tem as seguintes características:

- **Job I**: Um fragmento de 3 CPUs, *walltime* restante de 150 segundos.
- **Job J**: Um fragmento de 6 CPUs, *walltime* restante de 100 segundos .
- **Job K**: Um fragmento de 7 CPUs, *walltime* de 100 segundos. O prazo do *job* K está a uma distância de 290 segundos do instante atual e seu início a uma distância de 190 segundos. Peso total: 700.
- **Job L**: Um fragmento de 4 CPUs e *walltime* de 250 segundos. Peso total: 1000.

A rotina que insere *jobs*, chama a rotina que libera espaço para a máquina Máq1 e a rotina que libera espaço por sua vez, chama a rotina que executa o método *Empurrar* para o fragmento de *job* empurrável do *job* K. O método *Empurrar* localiza a Máq2 onde o fragmento de K cabe e o empurra para essa máquina, como mostrado na Figura 5.10.

Após ter utilizado o método *Empurrar* a Máq1 que era viável, passa a ser uma máquina onde o fragmento do *job* L cabe. Assim o escalonador pode finalmente pode inserir o *job* L na máquina Máq1 como mostrado na Figura 5.11.

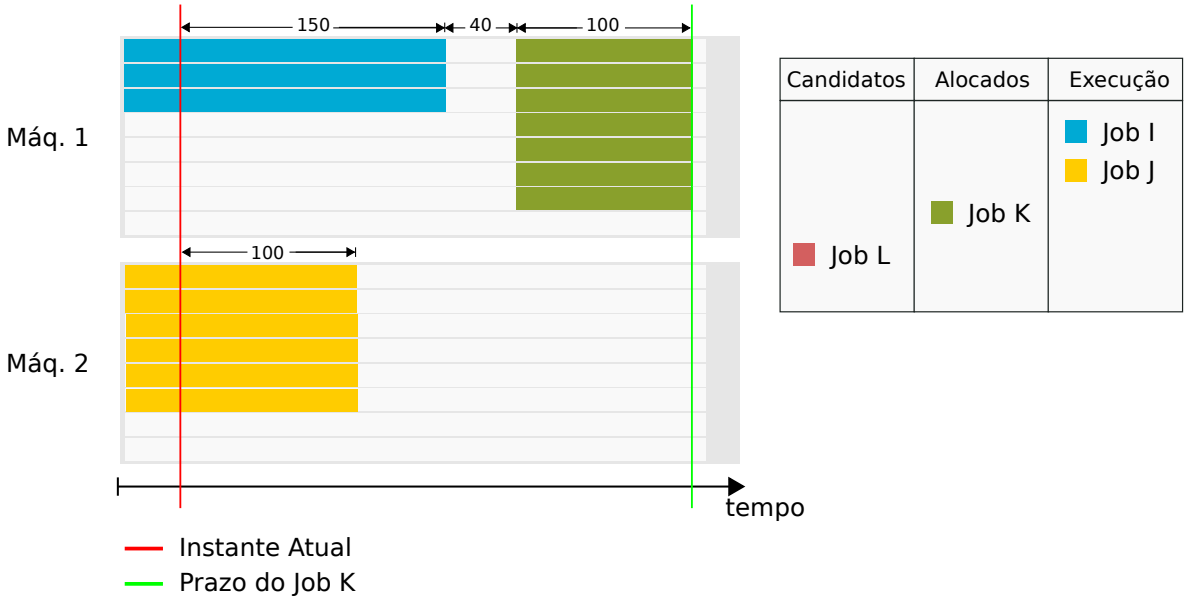

Figura 5.9: Após a submissão do *job* L o escalonador acorda para atender esse *job* e encontra apenas uma máquina viável Máq1 onde pode tentar liberar espaço para o *job* L.

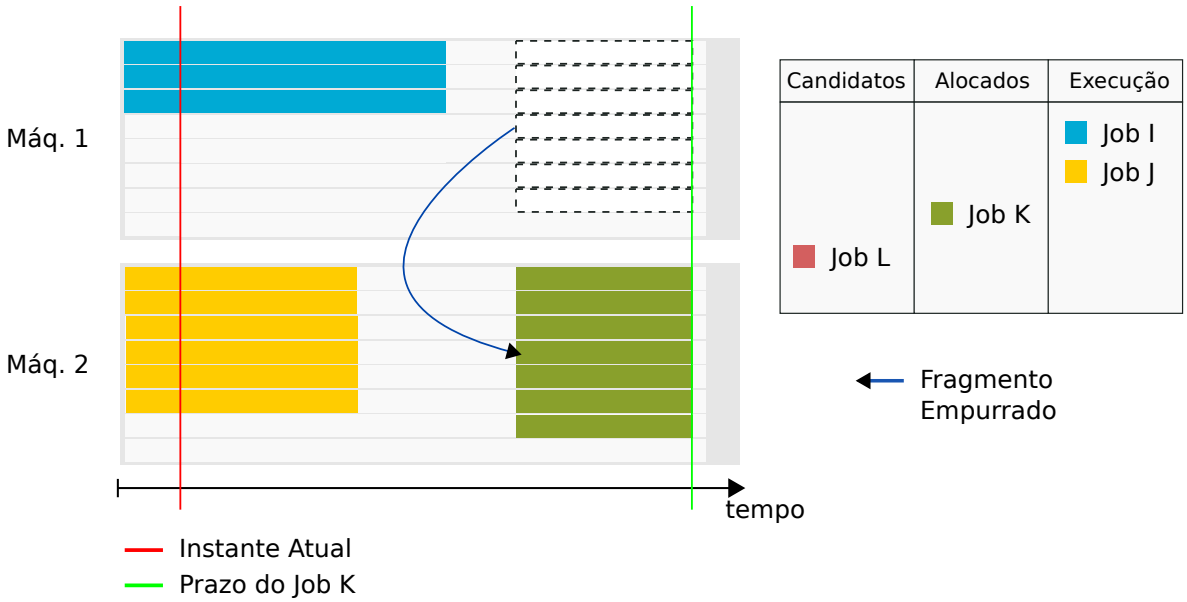

Figura 5.10: Apesar do *job* K ter maior importância que o *job* L, o método *Empurrar* pode empurrar o *job* K sem prejuízo para o mesmo. Como a máquina Maq2 possui espaço onde o *job* K possa caber, o método *Empurrar* consegue mover o *job* para essa máquina.

5.3.4 Exemplo com *jobs* com mais de um fragmento

Este exemplo tem o objetivo de ilustrar o método *Empurrar* sendo utilizado para auxiliar na inserção de um *job* com mais de um fragmento. Após um evento de término de *jobs*, três máquinas do *cluster* passam a ter espaço suficiente para que o escalonador possa tentar

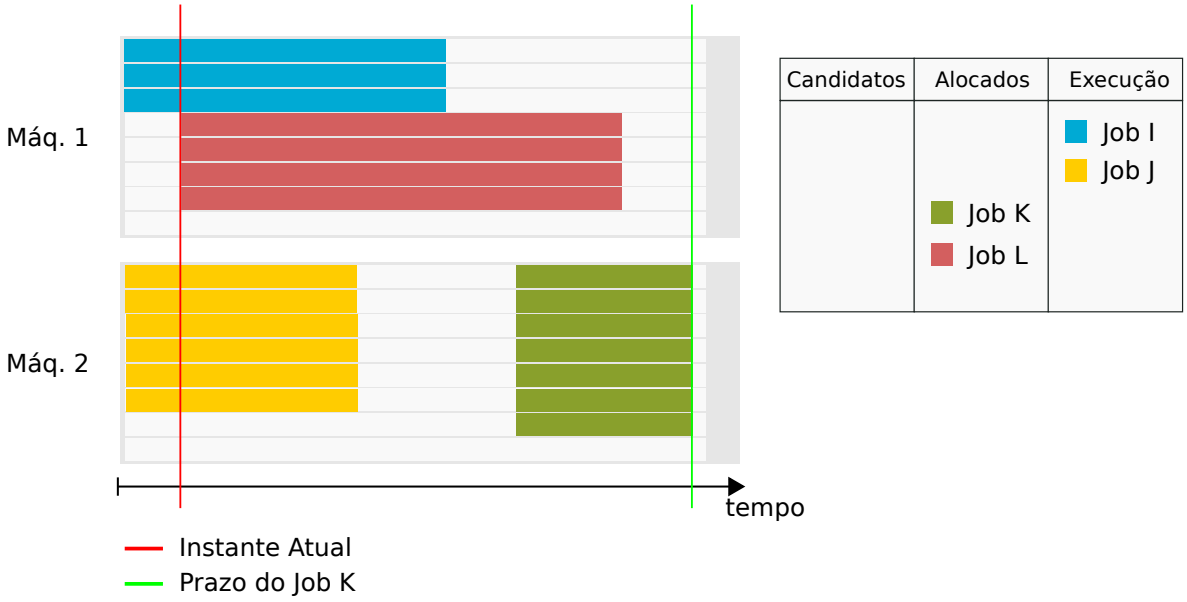

Figura 5.11: Após o espaço necessário ter sido liberado na Máq1, a rotina que insere *jobs* pode alocar o *job* L nessa máquina.

inserir 2 *jobs* Comuns Candidatos de dois fragmentos cada um, que estavam esperando por recursos. Essa situação pode ser vista na Figura 5.12.

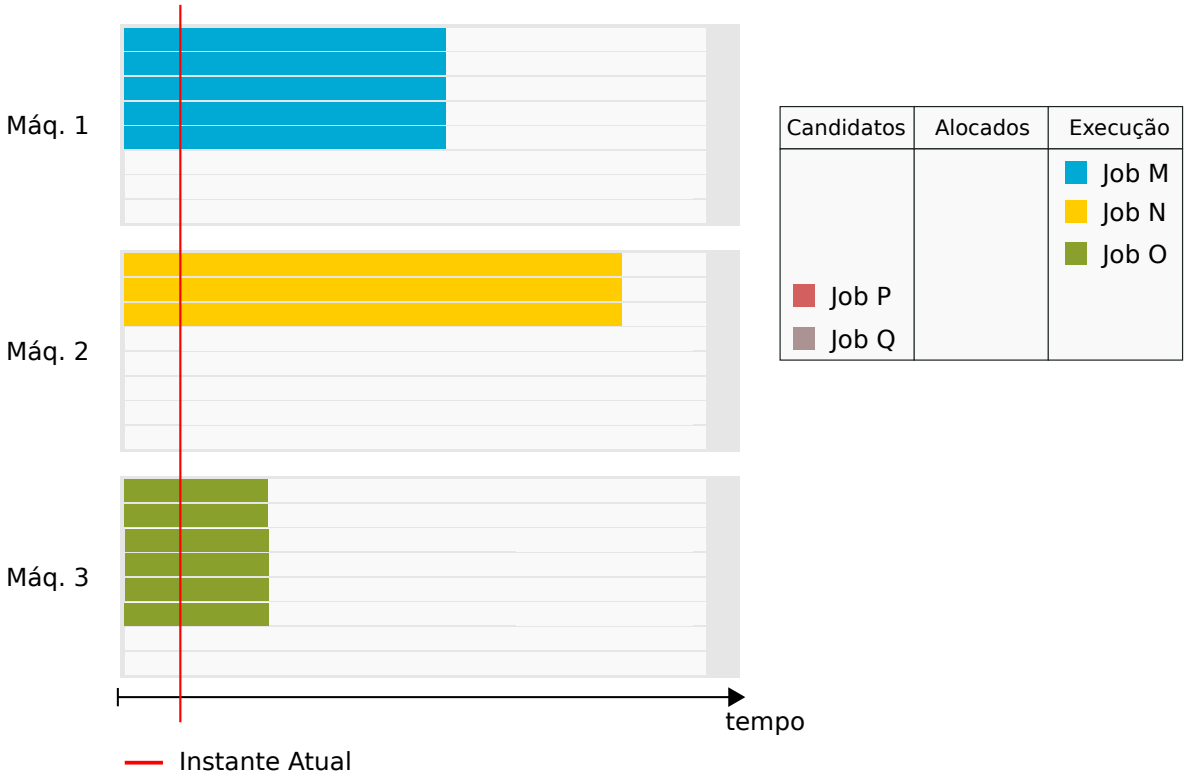

Figura 5.12: Três máquinas que podem ser utilizadas para a inserção de *jobs* candidatos após um evento término de *jobs*

No momento em que o escalonador entra na rotina ciclo de escalonamento onde vai tratar os dois *jobs* Comuns Candidatos P e Q, três *jobs* outros *jobs* se encontram em execução nas máquinas Máq1, Máq2 e Máq3, os *jobs* M, N e O. O *job* P é escolhido primeiro por ter peso menor. O escalonador então escolhe o fragmento do *job* P que utilize o menor número de CPUs e tenta inserir esse fragmento numa máquina bestfit. O escalonador então consegue alocar esse fragmento na máquina Máq1 (Ver Figura 5.13).

Os *jobs* deste exemplo tem as seguintes características:

- **Job M**: Um fragmento de 5 CPUs, *walltime* restante de 150 segundos.
- **Job N**: Um fragmento de 3 CPUs, *walltime* restante de 250 segundos .
- **Job O**: Um fragmento de 6 CPUs, *walltime* restante de 50 segundos .
- **Job P**: Dois fragmento de 2 CPUs, *walltime* de 200 segundos. Peso total: 2 frags x 400 = 800.
- **Job Q**: Dois fragmento de 3 CPUs, *walltime* de 220 segundos. Peso total: 2 frags x 660 = 1320.

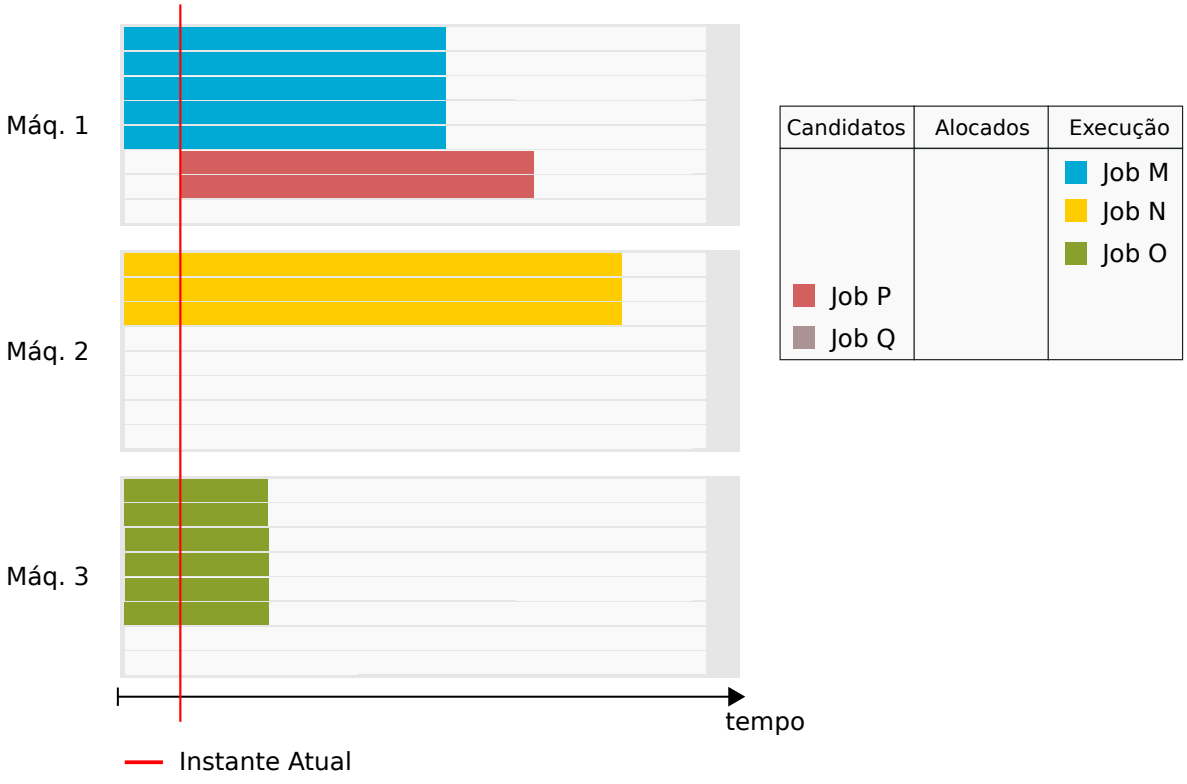

Figura 5.13: O escalonador insere o primeiro fragmento do *job* P na máquina *bestfit* Máq1

Em seguida o escalonador tenta alocar o segundo fragmento do *job* P e consegue alocar esse fragmento na máquina Máq2, que é a *bestfit*, como ilustrado na Figura 5.14.

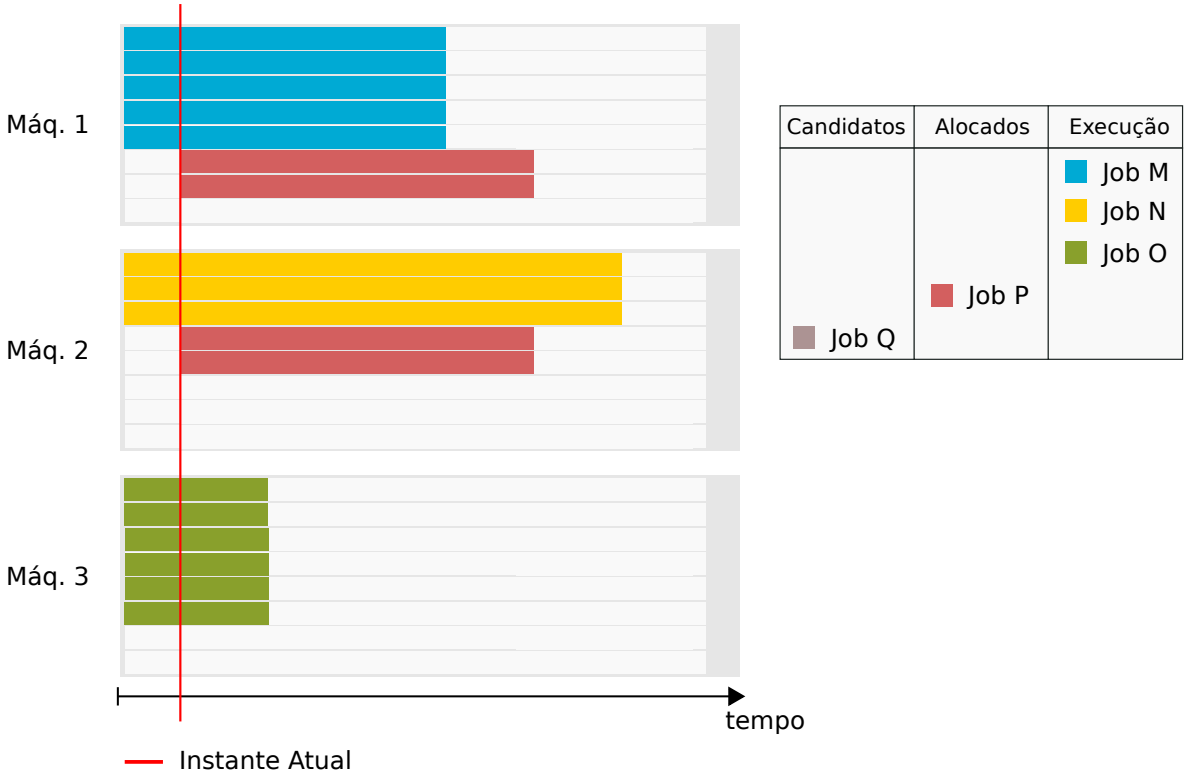

Figura 5.14: O escalonador insere o segundo fragmento do *job* P na máquina *bestfit* Máq 2. O *job* P passa a ser considerado como alocado

Após inserir os dois fragmentos do *job* P, esse *job* passa a ser considerado com alocado pelo escalonador e ele escolhe o próximo *job* que no caso é o *job* Q. O primeiro fragmento escolhido do *job* Q é inserido na máquina Máq2 por essa ser a única máquina onde o fragmento cabe e portanto a máquina bestfit. A inserção desse fragmento pode ser visto na Figura 5.15.

Depois disso, o escalonador tenta inserir o segundo fragmento do *job* Q, porém não encontra uma máquina *bestfit* para esse fragmento. Porém existem duas máquinas viáveis Máq1. e Máq2. onde a rotina de inserir *jobs* pode tentar liberar espaço para o fragmento. A máquina Máq1 é a primeira viável escolhida, porque ela tem menor quantidade de espaço a ser liberado que a Máq2. Ao tentar liberar espaço na Máq1, a rotina que libera espaço encontra um fragmento empurrável do *job* P e consegue empurrar esse fragmento para a máquina Máq3 conforme pode ser visto na Figura 5.16.

Depois que um dos fragmentos do *job* P é empurrado para fora da máquina Máq1, essa máquina passa a ter espaço suficiente para conter o segundo fragmento do *job* Q e

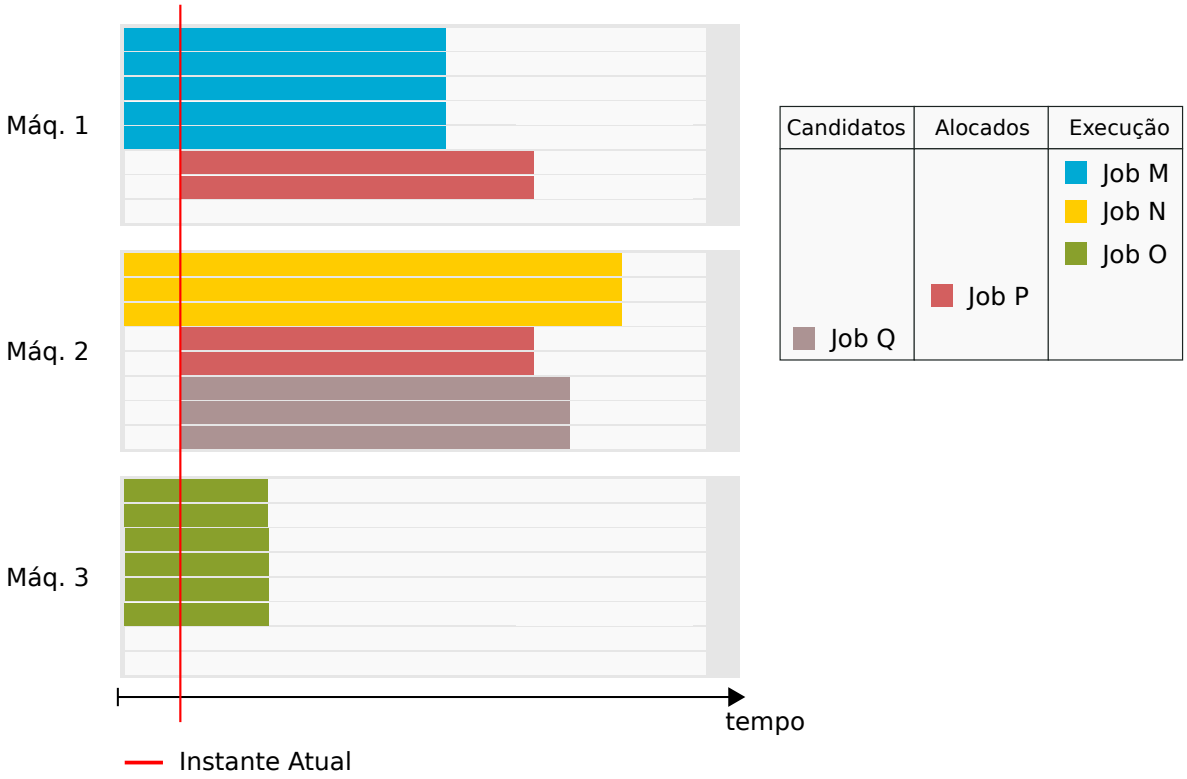

Figura 5.15: O primeiro fragmento do *job* Q é inserido na máquina *bestfit* Máq2. Porém o segundo fragmento do *job* Q não pode ser inserido diretamente

com isso a rotina que insere *jobs* consegue finalizar a alocação de todos os fragmentos do *job* Q e esse passa a ser considerado como alocado, como pode ser visto na Figura 5.17.

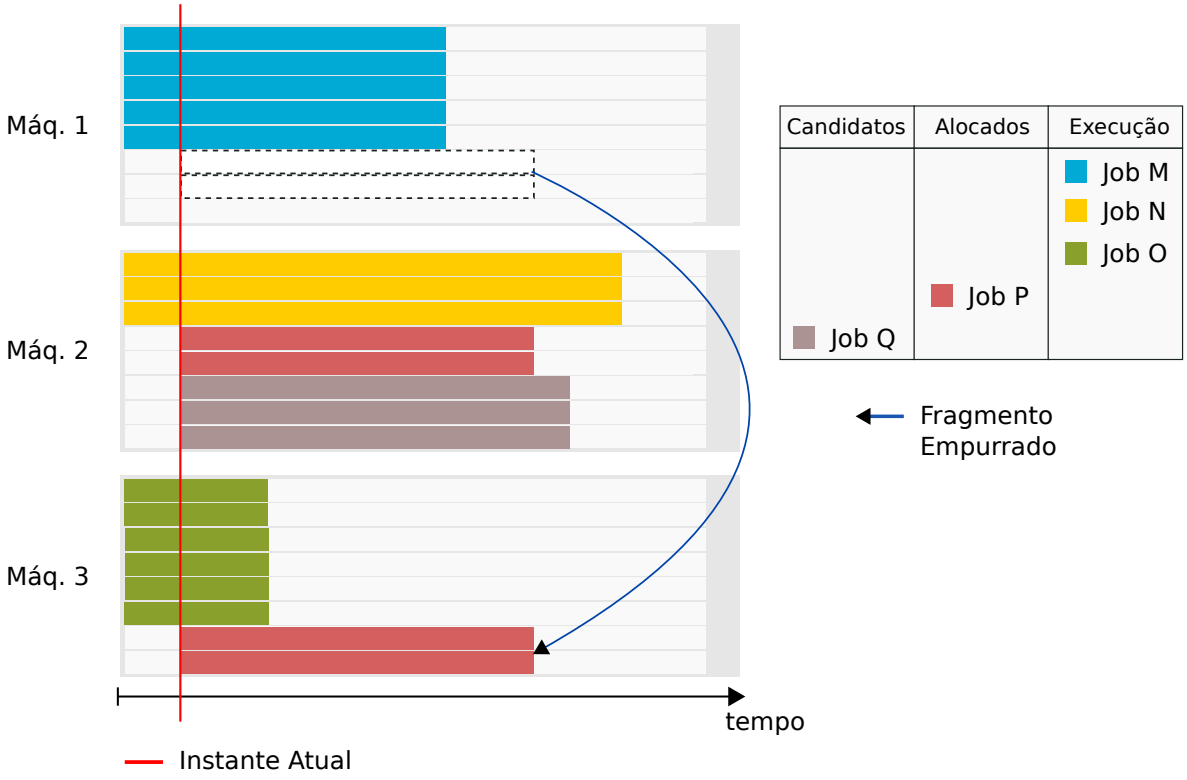

Figura 5.16: Dentre as máquinas viáveis Máq1. e Máq2., o escalonador consegue liberar espaço na Máq1, empurrando um fragmento do *job* P para a máquina *bestfit* Máq3.

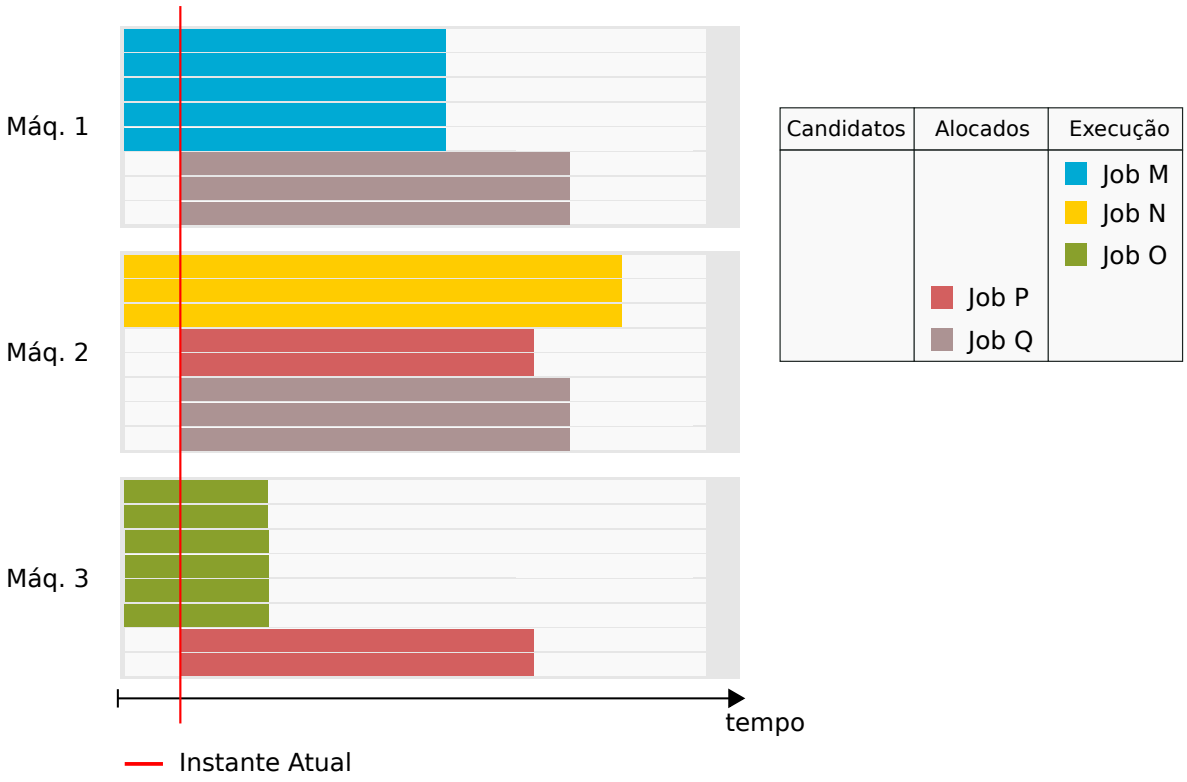

Figura 5.17: Com o espaço liberado na máquina Máq1. a alocação do segundo fragmento do *job* Q passa a ser possível nessa máquina, assim o fragmento é inserido e o *job* Q passa a ser considerado como alocado

5.4 Tipos de *jobs* e Estruturas de dados utilizadas

Essa seção tem o objetivo explicar os tipos de *jobs* existentes no OscarPBS, bem como explicar seus estados durante um ciclo de escalonamento. Também serão explicados aqui as principais estruturas de dados utilizadas pelo escalonador OscarPBS, visando dar maior clareza a explicação de seu funcionamento na seção que se segue. O formato das requisições de usuário e os campos das requisições no Banco de Dados também são mostradas nesta seção, uma vez que essas estruturas podem facilitar o entendimento do escalonador.

5.4.1 Tipos de *Jobs*

Conforme definido no início desse trabalho um *job* é composto de um ou mais fragmentos, que por sua vez podem ocupar todas as CPUs de uma máquina ou apenas parte das CPUs de uma máquina. O *job* pode ter fragmentos na mesma máquina ou em máquinas diferentes e todas as CPUs desses fragmentos do *job* são disponibilizados para um usuário executar uma aplicação possivelmente paralela durante um período de tempo. Nesse conjunto de CPUs de um *job*, o usuário terá exclusividade das CPUs durante o tempo de vida de seu *job*. As aplicações de usuário podem ser compostas de partes que executam em uma mesma máquina e partes que executam em máquinas distintas na rede. Sendo assim, é possível a um usuário especificar um *job* como um conjunto de partes indivisíveis que são chamados de fragmentos de *jobs*. No *job* está incluído um arquivo com uma sequencia de comandos de terminal, que o usuário deseja que seja executado nas máquinas. O escalonador disponibiliza ao ambiente de execução do usuário variáveis de ambiente que indicam arquivos com informação sobre o escalonamento, como quais máquinas foram obtidas e a característica dessas máquinas. O script do *job* é executado a partir de uma das máquinas obtidas e cabe ao script do usuário decidir como iniciar o uso dos recursos obtidos. O usuário deve estar ciente que seu *job* não poderá executar mais tempo do que foi solicitado, nem de usar mais recursos do que foi pedido durante a submissão do *job*, podendo ter acesso negado ou o *job* finalizado.

Há quatro tipos de *jobs* no OscarPBS:

1. *Jobs* Comuns

   Os *jobs* comuns, entram no final da lista de candidatos de *jobs* comuns. Cada *job*, ao entrar na lista, tem seu peso calculado e armazenado. A lista de candidatos é então ordenada por peso crescente. O escalonador tenta alocar cada *job* comum

da lista de candidatos, seguindo a ordem de pesos. Todos os *jobs* comuns são alocados no instante atual se possível. Todos os *jobs* comuns alocados são postos em execução. Todos os *jobs* comuns que não puderam ser alocados no instante atual permanecem na lista de candidatos e serão considerados novamente no próximo ciclo de escalonamento.

2. *Jobs* Starve

   Os *jobs* comuns que permanecem na lista de candidatos após um determinado tempo, passam para a lista de *starve* candidatos. A lista de *jobs Starve* é ordenada em ordem *FIFO*. O escalonador tenta alocar cada *job starve* inicialmente no instante atual, o que possibilitaria a execução imediata do *job*. Não sendo possível, o escalonador avança para posições futuras até encontrar uma posição onde seja possível alocar o *job starve*.

3. *Jobs* QoS

   O *job* QoS possui prioridade maior que *jobs* comuns e segue uma política *FIFO*. Por qualidade de serviço se entende que o *job* tem um prazo de tempo definido pelo usuário e que se o *job* puder ser alocado quando o usuário o requisitar, os *jobs* comuns, *starves* e outros *jobs* QoS não poderão mais atrasar além do prazo definido. Se o *job* QoS não puder ser alocado, ele fica na fila, aguardando a possibilidade de algum *job* terminar prematuramente, porém não estará garantido que ele poderá ser alocado. Um *flag* de status mostra ao usuário que o escalonador já tentou escalonar seu *job* QoS e em caso de não conseguir, o usuário pode escolher entre tentar diminuir ou deixar mais flexível seu *job*, ou então deixar o *job* na fila na esperança que o escalonador o possa escalonar.

   Ao ser submetido, o *job* QoS encontra uma situação em que pode haver *jobs* em execução, *jobs* QoS e emergência alocados, *jobs* starve alocados e *jobs* comuns esperando uma oportunidade de execução. Nesse momento, o escalonador procura reservar um local para encaixar o *job* QoS antes de seu prazo. Os *jobs* comuns que já estavam na fila antes do QoS e os *jobs starve* não críticos podem ser desalocados quando se tenta encaixar o *job* QoS. Se o escalonador não puder reservar no momento, o *job* QoS permanece como um *job* candidato, ainda tendo chance de ser alocado antes de qualquer *job* comum até a próxima liberação de recursos. No momento da liberação de recursos, o escalonador ainda dá prioridade ao *job* QoS, frente a todos os *jobs* comuns candidatos e *starve* alocados, porém se o *job* QoS não conseguir reservar o espaço, o escalonador não deixará de tentar colocar *jobs*

comuns e outros *jobs* em espaços que permitam executar o *jobs* no momento atual, de modo que o *job* QoS não poderá monopolizar o espaço livre que não conseguiu obter, antes de seu prazo.

Os *jobs* QoS candidatos, em cada ciclo de escalonamento, são considerados em duas etapas. A primeira etapa serve para garantir que o *job* QoS foi alocado no prazo. Nesta etapa, o escalonador inicia as tentativas de alocação na posição mais ao futuro possível, ainda no prazo do *job* QoS. Se não conseguir, vem caminhando na direção do instante atual, tentando alocar o *job* QoS. Na primeira etapa, o *job* QoS tem o poder de desalocar *jobs starve* alocados (os mesmos voltam para a fila de *starve* candidatos). A segunda etapa é realizada com *jobs* QoS que estão alocados. O escalonador tenta alocar esses *jobs* no instante atual, se houver recursos ociosos, usando o algoritmo de empurra (que será explicado mais tarde) enquanto respeita todos os *jobs* já alocados.)

4. *Jobs* de Emergência

   O mesmo que acontece com um *job* QoS, acontece com um *job* de emergência recém-submetido. Porém, o *job* de Emergência pode ter poderes extras para desalocar *jobs* próximo a seu prazo final, de modo que sua prioridade pode ser aumentada até que nenhum outro *job* possa impedir o escalonamento de um *job* de emergência, apenas a falta de recursos totais poderia impedir a alocação do *job* Emergência.

   O *job* de Emergência pode experimentar três etapas. Na primeira tentativa, o escalonador tenta reservar a posição do *job* de Emergência, iniciando pelo prazo, tendo o poder de desalocar *jobs starve* alocados, se for necessário. Se não conseguir, o escalonador tenta alocar o *job* de Emergência em seu prazo, usando todos os poderes que o administrador tenha dado ao *job* (desalocar QoS, emergência, *starve* crítico, *job* em execução). No final do ciclo de escalonamento, o escalonador tenta realocar os *jobs* de Emergência alocados no instante atual, respeitando todos os *jobs* alocados.

Os quatro tipos de *jobs* descritos acima, possuem um ciclo de vida enquanto são tratados pelo escalonador, que pode ser descrito por cinco estados. Basicamente, os estados possíveis de qualquer *job* são Candidato, Alocado, Atual, Execução e Pós-escalonamento.

O *job* chega ao escalonador por meio de uma requisição de usuário. A requisição tem as especificações dos recursos necessários ao *job* e *walltime*, também trazendo o tipo de *job* e o prazo. Isso permite ao escalonador tratar a requisição em um *job*, processo que

será chamado de parse da requisição, onde uma estrutura de *job*, com seus fragmentos, é criada.

5.4.2 Estados de um *Job*

Os quatro tipos de *job* descritos anteriormente passam por estados durante o ciclo de vida no escalonador. Quando os *jobs* são submetidos, eles ficam armazenados em uma tabela do banco de dados e assim que o escalonador entra em um novo ciclo de escalonamento, possivelmente iniciado pelo evento de chegada de novos *jobs*, o escalonador coloca esses *jobs* em um estado inicial como sendo Candidatos. E a partir deste estado cada *job* muda de estado dentro do escalonador até chegar a um estado final de Pós-escalonamento, que representa o término do ciclo de vida do *job* no escalonador. São apresentados aqui os cinco tipos de estado para os *jobs*, havendo geralmente estruturas de dados onde cada estado de *job* fica armazenado. O ciclo de vida de um *job*, com relação a seus estado pode ser visto na Figura 5.18. As estruturas de dados onde os *jobs* ficam armazenados podem ser vistos na Figura 5.19. Os *jobs* geralmente possuem uma sequencia natural na transição de estados, porem é possível que essa ordem seja alterada durante o ciclo de escalonamento. A ordem comum dos estados inicia pelo estado de Candidato, passando para o estado de Alocado, depois para o estado de alocado no instante Atual, seguido do estado de Execução terminando no estado de pós-escalonamento. A descrição de cada estado, bem como a transição entre eles é explicada a seguir.

1. Estado Candidato

   Esse estado geralmente representa *jobs* que acabaram de ser reconhecidos pelo escalonador, mas que ainda não têm uma posição definida no *cluster* com relação a quais máquinas irão utilizar e em que instante de tempo iniciar. Em outras palavras, são *jobs* que acabaram de ser submetidos pelos usuários e lidos do banco de dados, mas que ainda não foram tratados. Esse estado também pode ser ocupado por *jobs* que estavam alocados ou em execução mas que tiveram que ceder a *jobs* de maior prioridade, sendo desalocados ou parando a execução e voltando ao estado de candidatos. Há também uma possível migração de *jobs* comuns candidatos para *jobs starve* candidatos. Os *jobs* candidatos podem migrar para o estado de alocados ou pós-execução, significando ou que conseguiram um espaço no *cluster* ou que não é mais possível alocar esses *jobs*. Com relação a estrutura de dados, existe uma lista de candidatos para cada um dos quatro tipos de *job*.

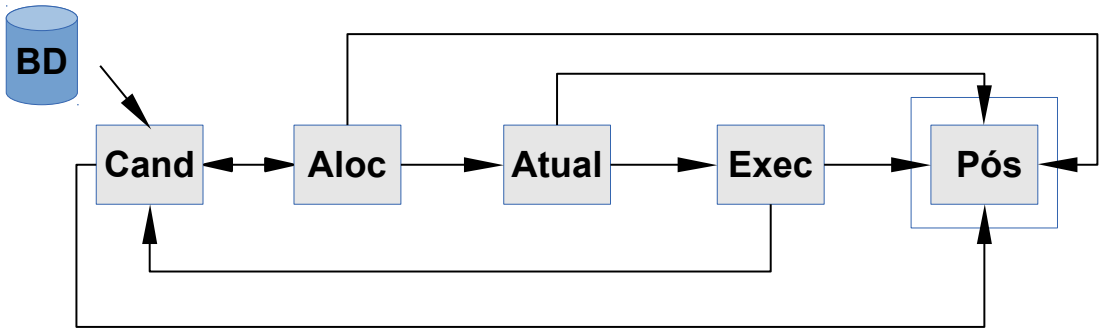

Figura 5.18: Estados dos *jobs* durante um ciclo de escalonamento. Todos os tipos de *jobs* iniciam no estado Candidato representado na Figura por Cand. Com exceção dos *jobs starve* Candidatos, todos os Candidatos vêm da tabela do banco de dados, representada por BD. O estado de *job* Alocado é representado por Aloc. *Jobs* alocados no instante atual estão representados por Atual. O estado de Execução aparece representado por Exec. O estado final de todos os *jobs*, Pós-escalonamento, aparece representado por Pós. As setas representam as possíveis transições entre os estados durante o ciclo de vida de um *job*.

2. Estado Alocado

   Para o *job* nesse estado, estão definidas as máquinas onde ele pode executar e também o intervalo de tempo. Há neste estado uma garantia que o *job* deve executar, a menos que um *job* de prioridade maior precise desalocá-lo. O escalonador geralmente seleciona uma lista de Candidatos, ordena a lista e a partir dai tenta colocar os *jobs* Candidatos no estado de Alocados. Apenas *jobs* no estado de Candidatos podem se tornar Alocados. Quando no estado de Alocados, os *jobs* podem entrar no estado de Alocados no instante atual, podem voltar a ser Candidatos, ou passar para o estado de Pós-escalonamento. Com relação a estrutura de dados há uma lista de alocados para cada tipo de *job*.

3. Estado Alocado no instante atual

   Esse estado representa *jobs* Alocados prontos para serem colocados em execução no ciclo atual de escalonamento. Esse estado representa *jobs* Alocados cujo o início seja igual ao instante de tempo gravado no início do ciclo de escalonamento, ou seja anterior ao instante atual dentro de uma tolerância de tempo definida na configuração do escalonador. O escalonador pode descobrir que alguns *jobs* Alocados se tornaram aptos a serem Alocados no instante atual, ou pode tentar realocar estes *jobs* já alocados para o instante atual. *Jobs* Alocados no instante atual podem passar para o estado de Execução, ou passar para o estado de Pós-escalonamento, caso

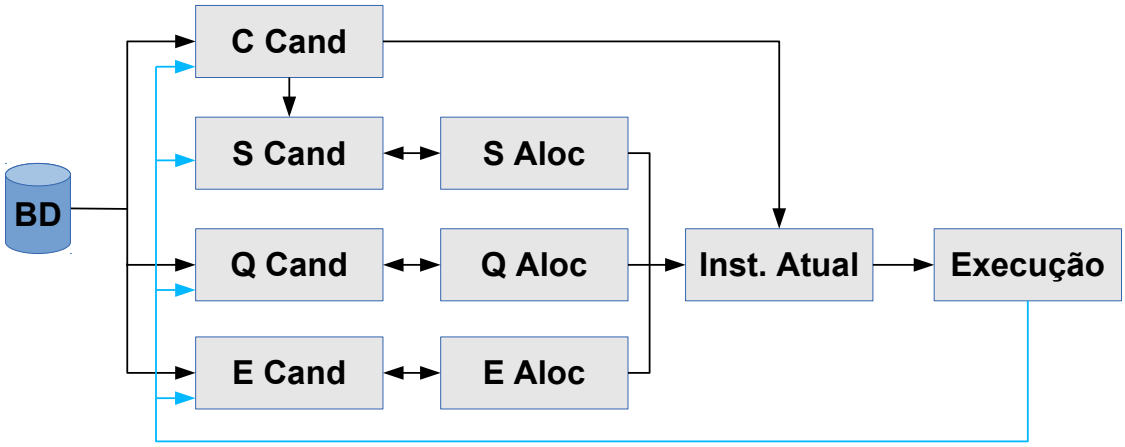

Figura 5.19: Listas que contém *jobs* e que são utilizadas pelo escalonador durante o ciclo de escalonamento. A tabela do banco de dados de onde as requisições de usuário chegam é representada por BD. As listas de *jobs* Comuns Candidatos, *Starve* Candidatos, QoS Candidatos e Emergência Candidatos são representadas por C Cand, S Cand, Q Cand e E Cand respectivamente. As listas com *jobs starve*, QoS e emergência Alocados são representadas por S Aloc, Q Aloc e E Aloc, respectivamente. A lista de todos os *jobs* alocados no instante atual é representada por Inst. Atual. A lista de todos os *jobs* já em execução é representada por Execução. As setas representam as transições dos *jobs* entre as listas durante os ciclos de escalonamento. As setas azuis representam *jobs* em execução que podem ser interrompidos por *jobs* de Emergência (quando esses tem poder suficiente para isso e precisam fazer isso) e voltam para a lista de Candidatos.

ocorra algum erro durante a tentativa de colocar o *job* em execução. Existe uma única lista, em que todos os *jobs* que estão alocados no instante atual são colocados, quando o escalonador chega ao final do ciclo de escalonamento, ele tenta colocar em execução todos os *jobs* dessa lista.

4. Estado Execução

   Esse estado representa *jobs* que estejam em execução no *cluster*. Esses *jobs* ocupam espaço nas máquinas que os *jobs* alocados precisam respeitar. Os *jobs* entram nesse estado após terem passado pelo estado de *jobs* Alocados no instante atual e podem sair do estado de Execução para o estado de Pós-escalonamento, ou para o estado de Candidatos se forem interrompidos por *jobs* de Emergência. Existe uma única lista para todos os *jobs* em Execução.

5. Estado Pós-escalonamento

   Esse estado representa *jobs* que terminaram definitivamente o seu ciclo de vida no escalonador. Representam *jobs* em execução que terminaram espontaneamente

porque o tempo acabou ou porque o usuário o excluiu, ou *jobs* no estado de Candidato ou Alocado que não puderam ser alocados porque o prazo terminou, ou porque o usuário os excluiu. Neste estado, os *jobs* deixam de existir no escalonador, passando a existir apenas na tabela do banco de dados, para que o usuário verifique seu status final.

5.4.3 Submissão de *Jobs*

A sintaxe utilizada pelo OscarPBS para submissão de *jobs* se baseia no comando de submissão utilizado pelo OpenPBS/TORQUE PBS Professional e LibraPBS. Como no caso do PBS Professional permite fragmentar o *job* em conjuntos de CPUs, que podem executar em uma mesma máquina ou não, possibilitando que o fragmento seja exclusivo. No caso do TORQUE, fragmentos nunca ocupam uma mesma máquina. Como no libraPBS, permite a inclusão de prazo. O tipo foi acrescentado para definir um *job* como sendo Comum, QoS ou Emergência.

Os componentes de um comando de submissão são:

- Tipo (*flag* -t): define o tipo do *job* submetido, C: Comum, Q: QoS e E: Emergência
- Prazo (*flag* -p): se o *job* for tipo Q ou E, o usuário deve definir um prazo, que é uma data com dia-mes-ano hora:minuto:segundo
- Walltime (*flag* -l walltime=): tempo estimado de duração do *job*
- Recursos (*flag* -l nodes=): os fragmentos do *job*, que estão definidos como conjuntos de CPUs, exclusivos na máquina ou não, que podem ser livres ou fixos
- Arquivo de script: o caminho para um arquivo de script criado pelo usuário, onde se encontram os comandos que preparam e executam a aplicação do usuário nos recursos solicitados.

Exemplos de comandos de submissão:

```
qsub -l walltime=3:45 -l nodes=1:ppn=5+uff3:ppn=4:e -t C arq_de_script
```

```
qsub -p "3-07-2013 23:31" -l walltime=3:45 -l nodes=1:ppn=5+uff3:ppn=4:e
 -t Q arq_de_script
```

```
qsub -p "3-07-2013 23:31" -l walltime=3:45 -l nodes=1:ppn=5+uff3:ppn=4:e
 -t E arq_de_script
```

Para os recursos e lista de fragmentos, a sintaxe é:

```
-l nodes=fragmento_1[+fragmento_2[+...[+fragmento_n]]]
```

onde `fragmento_x`, tem a seguinte sintaxe:

```
{nfrags|nome_maquina}:ppn=n_cpus_frag[:e]
```

Cada fragmento precisa ter especificado o número de fragmentos do tipo, ou o nome da máquina onde o fragmento ficará (fragmento fixo). O campo *ppn* (processos por nó) indica para o OscarPBS o número de cores que o fragmento precisa reservar. O campo opcional representado por *:e* indica que o fragmento é exclusivo na máquina, em relação ao *job* submetido, ou seja, a máquina que esse fragmento ocupar deve ter apenas esse fragmento do *job* submetido, porém na mesma máquinas podem existir fragmentos de outros *jobs*.

Os usuários e o administrador podem fazer basicamente três tipos de requisições ao escalonador OscarPBS:

1. qsub: Comando destinado a submissão de *jobs*, do tipo Comum, QoS ou Emergência.
2. qstat: Lista o status dos *jobs* tratados pelo escalonador
3. qdel: Remove um *job* que esteja nos estados de candidato, alocado ou execução.

Os três tipos de requisições listadas acima são armazenadas em uma tabela do banco de dados do OscaoPBS, criadas com status de nao_atendida. Assim que uma submissão é armazenada, um sinal é disparado ao OscarPBS para que o mesmo acorde e trate dos registros pendentes.

5.4.4 Banco de Dados e Tabela de Requisições

No banco de dados do OscarPBS, existe uma tabela onde as requisições do usuário ficam armazenadas. Tais requisições são muito importantes para o escalonador e são verificadas em todos os ciclo de escalonamento. Os registros de requisição do banco de dados são

mantidos atualizados pelo escalonador, quando um *job* muda de estado, de modo a permitir que o usuário possa consultar o estado de seu *job*. Os campos da tabela de requisições do banco de dados do OscarPBS estão apresentados na Tabela 5.1.

Nome do campo	tipo	Função
id	inteiro	id único utilizado para identificar o *job*
requisicao	*string*	*string* de requisição feita pelo usuário
dono	*string*	*username* Linux do usuário dono da requisição
tipo	caracter	tipo da requisição
estado	caracter	estado do *job* relativo a submissão
atendida	booleana	requisição foi atendida ou não pelo escalonador
id_req_excluir	inteiro	se for uma requisição de exclusão, indica o id do *job* a ser excluído
tempo_real	*timestamp*	instante de tempo (data, hora, minuto e segundo) em que a requisição foi criada
prazo	*timestamp*	instante de tempo (data, hora, minuto e segundo) para o término de um *job*
sc_aloc	booleano	poder para desalocar *job starve* crítico alocado
q_aloc	booleano	poder para desalocar *job* QoS alocado
e_aloc	booleano	poder para desalocar *job* emergência alocado
c_exec	booleano	poder para desalocar comum em execução
s_exec	booleano	poder para desalocar *starve* em execução
q_exec	booleano	poder para desalocar QoS em execução
e_exec	booleano	poder para desalocar emergência em execução

Tabela 5.1: Tabela de Requisições

Com relação às estruturas globais existentes, há uma estrutura global de lista de máquinas, onde cada máquina é representada por um registro contendo informações úteis relativas à máquina, algumas variáveis utilizadas em cálculos temporários durante um escalonamento e um ponteiro para uma lista de intervalos de tempo de máquina, que representam intervalos de tempo utilizados. Cada objeto desta lista representa um intervalo de tempo, com instante inicial e final, totais de CPUs alocados e uma lista de fragmentos que estejam alocados naquele intervalo.

Há também uma lista de intervalos de tempo onde cada objeto da lista representa um intervalo de tempo que contém uma lista de intervalos de tempo de máquinas. Esses objetos são ordenados de maneira crescente de tempo e pode haver descontinuidades na lista, em intervalos de tempo em que não há nenhum fragmento alocado. Essa lista tem o objetivo de permitir que os métodos que fazem a verificação prévia da viabilidade de um *job* em um intervalo de tempo acessem com facilidade todos os intervalos de tempo máquina e computem os totais de CPUs utilizadas naquele intervalo de tempo específico. Em conjunto com o número total de CPUs disponíveis e de máquinas disponíveis, que é

computado no início de um ciclo de escalonamento, a rotina que verifica a viabilidade de um *job* em um intervalo de tempo utiliza a lista de intervalos globais.

Para a alocação de *jobs*, o escalonador utiliza ainda as seguintes listas:

- Lista dos *jobs* comuns candidatos: É uma lista de ponteiros para *jobs* do tipo Comum, no estado de Candidatos
- Lista dos *jobs starve* candidatos: É uma lista de ponteiros para *jobs* do tipo *Starve*, no estado de Candidatos
- Lista dos *jobs* QoS candidatos: É uma lista de ponteiros para *jobs* do tipo QoS, no estado de Candidatos
- Lista dos *jobs* emergência candidatos: É uma lista de ponteiros para *jobs* do tipo Emergência, no estado de Candidatos
- Lista dos *jobs starve* alocados: É uma lista de ponteiros para *jobs* do tipo *Starve*, no estado de Alocados
- Lista dos *jobs* QoS alocados: É uma lista de ponteiros para *jobs* do tipo QoS, no estado de Alocados
- Lista dos *jobs* emergência alocados: É uma lista de ponteiros para *jobs* do tipo Emergência, no estado de Alocados
- Lista dos *jobs* prontos para execução: É uma lista de ponteiros para *jobs* de todos os tipos, Alocados no instante atual
- Lista dos *jobs* em execução: É uma lista de ponteiros para *jobs* de todos os tipos que estão no estado de Execução

Durante a tentativa de escalonamento de um *job*, os fragmentos de um *job* vão sendo inseridos nas máquinas podendo causar eventos de empurra em fragmentos de *jobs* já alocados e podendo desalocar *jobs* já alocados. Uma lista de eventos de empurra e uma lista de fragmentos excluídos é mantida durante a tentativa de inclusão do *job*.

5.4.5 Estruturas de dados de *jobs*, fragmentos, máquinas e intervalos de tempo

Nesta subseção serão apresentados detalhes de estruturas de dados importantes para o funcionamento do escalonador do OscarPBS. Inicialmente serão mostradas as estruturas

de um *job* e suas quatro listas de fragmentos, depois será mostrada a estrutura de um fragmento de *job*. A estrutura de dados que guarda valores importantes para cada máquina também será apresentada. Por último, serão apresentadas as estruturas que representam como os fragmentos dos *jobs* alocados e em execução ficam armazenados nas máquinas e uma lista global de intervalos de tempo que auxilia na verificação de recursos livres e desalocáveis durante a alocação de um *job*.

A estrutura de dados de um *Job*

Para cada *job* que chega ao escalonador OscarPBS por meio das requisições de usuário, é criada uma estrutura de dados que armazena dados importantes do *job*. Nessa estrutura ficam armazenados dados que não mudam durante toda a vida do *job*, como por exemplo o *walltime* e o prazo e outros que mudam durante o escalonamento, como por exemplo início e estado. Cada *job* possui quatro listas de fragmentos, podendo algumas dessas listas serem vazias. A lista dos fragmentos livres guarda os fragmentos do *job* que não especificam uma máquina em particular e que por isso são fragmentos podem ocupar qualquer máquina disponível e com espaço suficiente. A lista de fragmentos livres exclusivos, guarda os fragmentos que, como os livres, podem ocupar qualquer máquina, porém eles possuem uma restrição. Se um fragmento livre exclusivo estiver alocado em uma máquina, ele deve ser o único fragmento do *job* a ocupar essa máquina. A lista de fragmentos fixos contém os fragmentos do *job* que tem a máquina especificada durante a requisição do usuário. Esses fragmentos só podem ocupar uma máquina, que é a especificada. Sobre a lista de fragmentos fixos e exclusivos, esses devem ocupar a máquina especificada durante a requisição e possuem a restrição de serem os únicos fragmentos a ocuparem suas respectivas máquinas, com relação ao *job*. A estrutura de dados de um *job* está ilustrada na Figura 5.20.

A estrutura dados de um fragmento de *job*

Os *jobs* são necessariamente compostos de um ou mais fragmentos, sendo que cada fragmento, quando alocado, ocupa recursos de uma só máquina do *cluster*. Uma máquina do *cluster* pode possuir vários fragmentos de *job*, mas um fragmento só pode estar alocado em uma máquina. Por outro lado, um *job* pode ocupar mais do que uma máquina do *cluster* se possuir mais do que um fragmento. Todos os fragmentos de um *job* compartilham o mesmo inicio e *walltime* de seu *job*. Alguns outros atributos dos fragmentos também são compartilhados com o *job*, como por exemplo o *poder* para desalocar outros *jobs* durante a alocação. Mesmo assim, os fragmentos de um *job* possuem atributos próprios como por exemplo número de CPUs, qual a sua máquina se esta estiver definida, entre vários outros

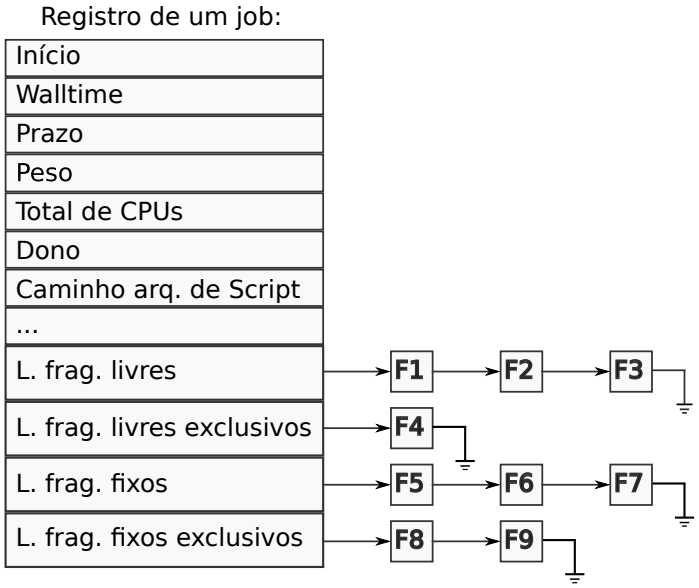

Figura 5.20: Um registro de *job*. Os fragmentos desse *job* são armazenados em quatro listas

atributos úteis ao fragmento durante o processo de alocação.

Os fragmentos de um *job* também determinam o peso do mesmo, que é calculado quando o *job* é criado. Por exemplo, um *job* com M fragmentos livres e N fragmentos fixos tem seu peso calculado com base na Equação (5.1). Nessa equação, FF representa uma taxa adicional de peso criada para penalizar fragmentos fixos, dado que esses fragmentos restringem as possibilidades de escalonamento. Os fragmentos fixos são solicitados quando os usuários necessitam que seus *jobs* ocupem máquinas pré-definidas. Geralmente essa escolha é feita devido a alguma configuração na aplicação do usuário que prende o funcionamento a algumas máquinas.

$$job.peso = job.walltime * (\sum_{i=1}^{M} frag\_livre_i.nCPUs + FF * \sum_{j=1}^{N} frag\_fixo_j.nCPUs) \tag{5.1}$$

Os campos mais importantes de um fragmento de *job* podem ser vistas na Tabela 5.2.

A estrutura dados de uma máquina

O escalonador do OscarPBS cria uma lista de máquinas quando é iniciado e cada uma dessas máquinas possuem atributos que devem estar de acordo com as máquinas reais do *cluster*. Para isso, cada máquina possui um registro onde são armazenados seus atributos. Além dos atributos da máquina, cada máquina possui uma lista de intervalos de tempo que

Nome do campo	Tipo	Função
job	ponteiro	indica o *job* que possui o fragmento.
nCPUs	inteiro	número de CPUs que o fragmento requer em uma máquina.
maquina	ponteiro	indica a máquina onde o fragmento está alocado.
maquina_original	ponteiro	indica qual maquina o fragmento ocupava antes da inserção de *job*.
lista_de_maquinas	ponteiro	guarda uma lista de máquinas onde o fragmento esteve durante uma inserção de *job*.
recursos_a_liberar	inteiro	recursos que o fragmento pode liberar na máquina em CPUs x segundos, ao ser empurrado.

Tabela 5.2: Campos de um fragmento de *job*

representam quais os fragmentos de *jobs* estão alocados na máquina e quando. Atributos importantes de uma máquina são, por exemplo, o número de CPUs da máquina e se ela está disponível ou não. Os atributos mais importantes de um registro de máquina podem ser vistos na Tabela 5.3

Nome do campo	Tipo	Função
nome	*string*	nome da máquina.
nCPUs	inteiro	número de CPUs da máquina.
disponivel	booleano	indica que a máquina está disponível para a submissão de *jobs*.
categoria	inteiro	classifica as máquinas viáveis de acordo com sua conveniência para a inserção de um frag. de *job*.
espaco_a_liberar	inteiro	quanto é preciso liberar em recursos para inserir um fragmento de *job*.
intervalos_de_tempo	ponteiro	faz referência a uma lista de intervalos de tempo onde existem fragmentos alocados.

Tabela 5.3: Campos de um registro de máquina

A lista de intervalos de tempo de uma máquina

Como dito anteriormente, cada máquina possui uma lista de intervalos de tempo que indica quais fragmentos de *job* estão alocados na máquina e quando. Esses fragmentos podem ser de *jobs* alocados e em execução. A motivo da existência desses intervalos é permitir que as rotinas do escalonador possam consultar cada máquina, em busca de espaços onde possam inserir fragmentos de *jobs*. Esses intervalos de tempo permitem as rotinas saberem quantas CPUs estão livres e quantas podem ser liberadas por meio do método *Empurrar* ou por desalocar fragmentos que estejam alocados nesses intervalos. A lista de intervalos de tempo de máquina só mantém intervalos que possuam pelo menos um fragmento. Cada intervalo de tempo da lista possui um registro com detalhes de como os

recursos estão sendo utilizados naquele intervalo e possui uma lista de fragmentos de *jobs* cujo o uma parte do intervalo de tempo coincida com o do intervalo de tempo da máquina. A Figura 5.21 ilustra duas listas de intervalo de tempo de máquina, representado duas máquinas com seus *jobs* alocados ou em execução. A Tabela 5.4 mostra os valores de cada registro de intervalo de tempo de máquina.

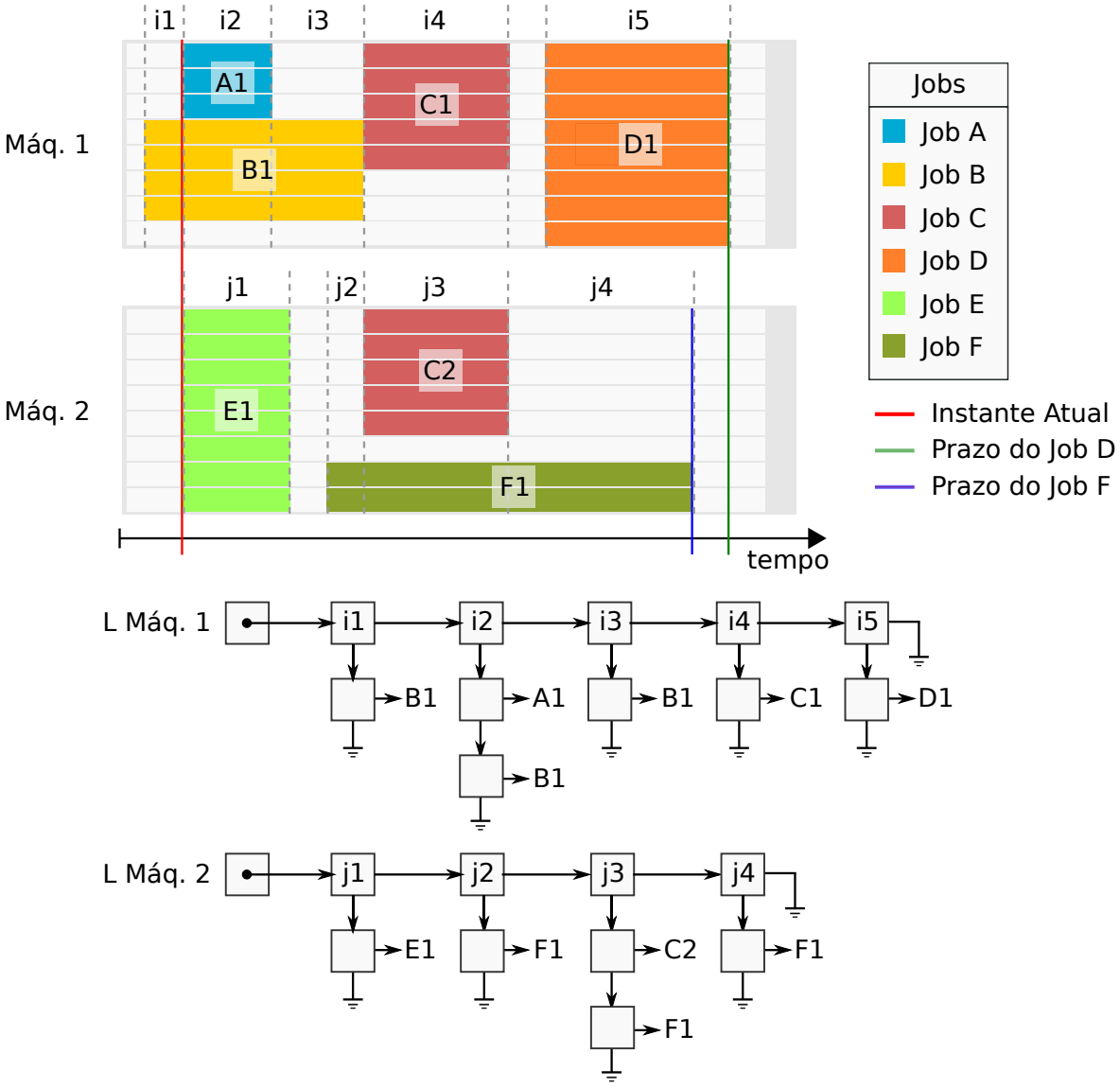

Figura 5.21: Os fragmentos de *job* que ocupam cada máquina ficam representados pelas listas de intervalos de tempo de sua respectiva máquina. Os intervalos de tempo da máquina 1, aparecem representados pelas letras i_1 até i_4. No caso dos intervalos de tempo da máquina 2, eles aparecem representados pelas letras j_1 até j_4.

A lista de intervalos de tempo global

Durante a alocação de um *job*, existe uma fase em que o escalonador OscarPBS precisa definir em que intervalo de tempo deve inserir um *job*. Tentar inserir um *job* em uma

posição de tempo qualquer, sem uma verificação prévia da viabilidade quantos aos recursos livres e alocados que o intervalo possui poderia ser muito trabalhoso para o escalonador. Sem um teste prévio de viabilidade, o escalonador, enquanto tentando inserir um *job*, tentaria empurrar e desalocar *jobs* em mais posições de tempo que são inviáveis para a inserção do *job*.

Assim, o escalonador OscarPBS possui uma lista de intervalos de tempo global, com o objetivo de ajudar as rotinas que procuram por uma posição no tempo a achar uma posição que não seja inviável. Encontrar uma posição não inviável não significa que ela seja viável, mas poupa muito trabalho para o escalonador, pois evita que ele tente inserir o *job* em lugares onde uma verificação prévia poderia indicar que a posição é inviável.

Assim existe uma lista de intervalo de tempos global, que permite as rotinas saberem quantas CPUs livres e desalocáveis existem em cada intervalo de tempo. Cada intervalo de tempo da lista global, possui uma lista dos intervalos de tempo de máquina que possui um tempo em comum ver Figura 5.22. Isso permite que o total de CPUs livres e desalocáveis de um intervalo de tempo qualquer sejam calculadas a partir de cada intervalo de máquina que coincida com aquele intervalo global. A Tabela 5.5 mostra os atributos de um elemento da lista dos intervalos globais.

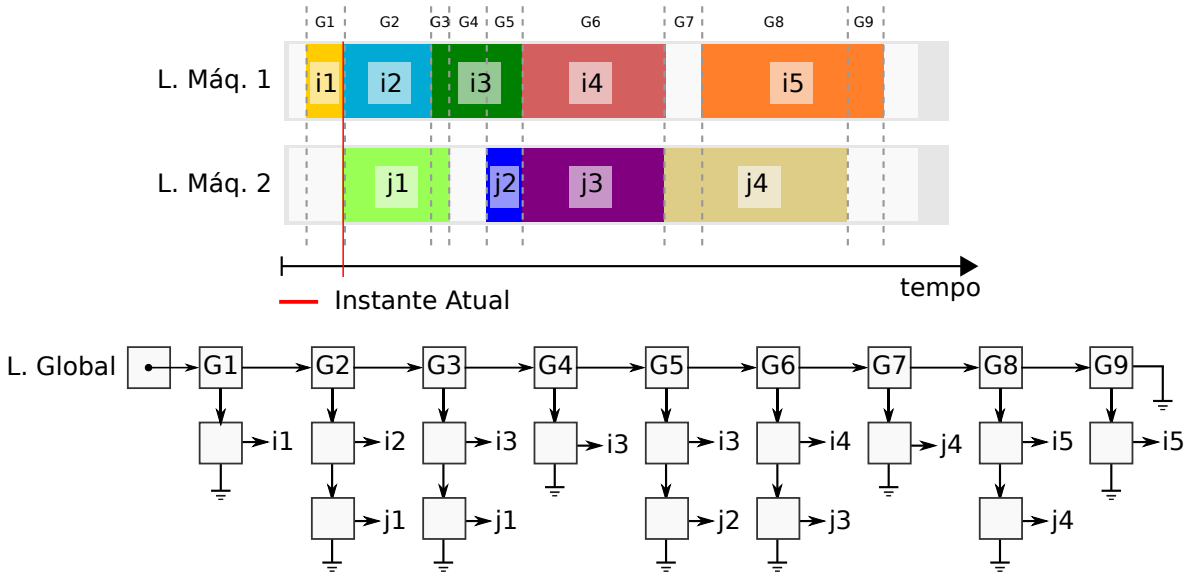

Figura 5.22: Cada intervalo de tempo global ajuda a determinar os recursos somados de todas as máquinas do *cluster* em um intervalo de tempo em que não há mudanças nos totais recursos nem nos intervalos de tempo das máquinas. Os intervalos de tempo globais aparecem representados pelas letras G_n, enquanto os intervalos de tempo das máquinas Máq1 e Máq2 aparecem representados pelas letras i_n e j_n respectivamente.

Nome do campo	Tipo	Função
inicio	inteiro	instante de tempo que define o inicio do intervalo
fim	inteiro	instante de tempo que define o final do intervalo
lista_fragmentos	ponteiro	lista de ponteiros para os fragmentos de *job* que coincidem com esse intervalo
tot_comum_movivel	inteiro	total de CPUs de fragmentos livres de *jobs* Comuns Alocados
tot_comum_nao_movivel	inteiro	total de CPUs de fragmentos fixos de *jobs* Comuns Alocados
tot_comum_exec	inteiro	total de CPUs de fragmentos de *jobs* Comuns em Execução
tot_starve_movivel	inteiro	total de CPUs de fragmentos livres de *jobs* Starve Alocados
tot_starve_nao_movivel	inteiro	total de CPUs de fragmentos fixos de *jobs Starve* Alocados
tot_starve_critico_movivel	inteiro	total de CPUs de fragmentos livres de *jobs Starve* Alocados que já foram descalocados o máx. de vezes permitido
tot_starve_critico_nao_movivel	inteiro	total de CPUs de fragmentos fixos de *jobs Starve* Alocados que já foram descalocados o máx. de vezes permitido
tot_starve_exec	inteiro	total de CPUs de fragmentos de *jobs Starve* em Execução
tot_qos_movivel	inteiro	total de CPUs de fragmentos livres de *jobs* QoS Alocados
tot_qos_nao_movivel	inteiro	total de CPUs de fragmentos fixos de *jobs* QoS Alocados
tot_qos_exec	inteiro	total de CPUs de fragmentos de *jobs* QoS em Execução
tot_emergencia_movivel	inteiro	total de CPUs de fragmentos livres de *jobs* Emergência Alocados
tot_emergencia_nao_movivel	inteiro	total de CPUs de fragmentos fixos de *jobs* Emergência Alocados
tot_emergencia_exec	inteiro	total de CPUs de fragmentos de *jobs* Emergência em Execução

Tabela 5.4: Campos de um nó da lista de intervalos de tempo de uma máquina

Nome do campo	tipo	Função
inicio	inteiro	instante de tempo que define o inicio do intervalo
fim	inteiro	instante de tempo que define o final do intervalo
lista_intervalos	ponteiro	lista de ponteiros para os intervalos de tempo de máquina que coincidem com esse intervalo

Tabela 5.5: Campos de um nó da lista de intervalos de tempo global

5.5 O algoritmo de escalonamento

Esta seção tem o objetivo de explicar o funcionamento do escalonador do OscarPBS de forma detalhada, complementando a visão geral dada na seção 4.2 e com a utilização das estruturas de dados ilustradas na seção 4.4. Também é mostrado quando o método *Empurrar* da seção 4.3 é utilizado e como é implementado. A seção mostra o escalonador em forma de rotinas, iniciando pela rotina do ciclo de escalonamento, seguindo com a explicação das rotinas que aparecem no ciclo de escalonamento. Outras rotinas importantes como as utilizadas na escolha da posição de cada *job* a ser inserido e rotinas responsáveis pela a escolha e liberação de recursos das máquinas para a inserção de *jobs* são mostradas aqui, como por exemplo uma rotina procura máquinas para os fragmentos dos *jobs* utilizando utilizando um critério *bestfit*, a rotina que encontra máquinas viáveis para um fragmento e as rotinas que tentam liberar espaço nessas máquinas com o uso do método *Empurrar* ou desalocando *jobs*.

Quando se mostrou necessário, algumas rotinas foram explicados de modo mais detalhado, com inclusão de um algoritmo que pudesse ilustrar seu funcionamento. As demais rotinas tem seu funcionamento brevemente explicado no texto.

5.5.1 As rotinas principais do ciclo de escalonamento

Nesta subseção são mostradas todas as subrotinas do método principal, que implementa um ciclo de escalonamento. Essas rotinas estão envolvidas em tarefas como a escolha e priorização dos *jobs*, finalização de *jobs* e para colocar *jobs* em execução. Elas estão diretamente relacionadas as Listas e estados de *jobs* mostradas na seção 4.4.

Loop do ciclo de escalonamento

O ciclo de escalonamento que ocorre no *loop* de eventos, conforme descrito na seção 3.2 é na verdade um conjunto de rotinas que tratam os quatro tipos de *job* do escalonador OscarPBS. O algoritmo 3 mostra as rotinas do ciclo de escalonamento. O significado de cada uma dessas rotinas será descrito nesta seção.

```
1  enquanto True faça
2      esperar_eventos();
       /* início do ciclo de escalonamento */
       /* etapa de liberação de recursos */
3      coletar_requisicoes_BD("Exclusao");
4      remover_jobs_requisitados() ;
5      remover_jobs_de_tempo_esgotado();
6      desalocar_jobs_finalizados();
7      desalocar_jobs_alocados_atrasados();
8      sincronizar_maquinas();
       /* garantir alocação dos jobs Emergência Candidatos */
9      coletar_requisicoes_BD("Emergencia");
10     criar_novos_jobs_Candidatos("Emergencia");
11     alocar_jobs_Emergencia_Candidatos_novos();
12     alocar_jobs_Emergencia_Candidatos_antigos();
       /* garantir alocação dos jobs QoS Candidatos */
13     coletar_requisicoes_BD("QoS");
14     criar_novos_jobs_Candidatos("QoS");
15     alocar_jobs_QoS_Candidatos();
       /* garantir alocação dos jobs Starve Candidatos */
16     procurar_novos_Starve_Candidatos();
17     alocar_jobs_Starve_Candidatos();
       /* lista dos jobs alocados no instante atual é iniciada aqui */
18     ListaAlocadosInstanteAtual = [];
       /* realocar jobs starve alocados no instante atual */
19     realocar_jobs_Starve_no_instante_atual();
       /* alocar jobs Comuns no instante atual */
20     coletar_requisicoes_BD("Comum");
21     criar_novos_jobs_Candidatos("Comum");
22     alocar_jobs_Comuns_candidatos_no_instante_atual();
       /* realocar jobs QoS e Emergencia no instante atual */
23     realocar_jobs_QoS_Emergencia_no_instante_atual();
       /* executar os jobs do instante atual */
24     executar_jobs_alocados_no_instante_atual();
       /* fim do ciclo de escalonamento */
25 fim
```

Algoritmo 3: O loop de eventos e as rotinas principais de um ciclo de escalonamento

A rotina esperar_eventos()

O ciclo de escalonamento acontece dentro de um *loop* infinito, então para que o escalonador não consuma recursos enquanto não houver trabalho útil a ser feito, a rotina esperar_eventos coloca o escalonador em suspensão durante um tempo pré-definido, ou a espera de algum evento externo que retire o escalonador da suspensão por meio do recebimento de um sinal enviado por outro processo, aquele que acontecer primeiro.

Quando existem novas requisições de *jobs*, por parte do usuário, ou quando *jobs* que estavam em execução terminam no TORQUE, a interface do usuário ou o TORQUE enviam sinais ao escalonador, fazendo o mesmo sair da suspensão, caso esteja, ou fazendo entrar em um ciclo de escalonamento assim que o ciclo atual terminar.

Além dos sinais citados anteriormente, o próprio escalonador pode prever se haverá trabalho a ser feito em algum momento no futuro. Antes de ficar suspensa a rotina calcula quando será o próximo início de *job* alocado ou fim de *job* em execução (de acordo com o tempo de execução estimado) e o que ocorrer primeiro define quanto tempo o escalonador pode ficar suspenso.

Em algumas situações a rotina permite que o *job* inicie o ciclo de escalonamento sem ficar suspenso. Isso acontece se o ciclo anterior não conseguir executar algum *job* que havia entrado no estado de Alocado no instante atual, ou se *jobs Starve* Candidatos surgirem no ciclo anterior, como será visto adiante.

Ao final dessa rotina, o escalonador grava o instante atual em uma variável e passa a utilizar esse instante de tempo como parâmetro de decisão em todo o ciclo de escalonamento atual.

A rotina coletar_requisicoes_BD(tipo_de_requisicao)

Durante o ciclo de escalonamento, o escalonador em algumas ocasiões consulta a tabela de requisições no banco de dados do OscarPBS para saber se há novas requisições feitas pelo usuário. Cada requisição é lida e tratada pelo escalonador uma única vez. Todas as requisições possuem um campo *atendida* que indica se foi tratada ou não pelo escalonador. Esse campo é iniciada com o valor Falso quando a requisição é criada e é atualizado para Verdadeiro pelo escalonador quando ele lê a requisição pela primeira vez.

O parâmetro tipo_de_requisicao indica o tipo de requisição nova o escalonador está interessado no momento. Assim apenas esse tipo de requisição é lido a atualizado para atendida igual a Verdadeiro, fazendo a rotina coletar_requisicoes_BD retornar apenas requisições do tipo escolhido.

Cada valor possível de tipo_de_requisição é descrito a seguir:

- O valor *"Exclusao"* faz a rotina verificar se há novas requisições de usuário que solicitam a exclusão de *jobs*. Essas requisições obrigatoriamente possuem uma referência ao *job* que o usuário deseja excluir.

- O valor *"Emergencia"* faz a rotina verificar se há requisições de *jobs* de emergência. Essas requisições obrigatoriamente trazem o prazo e os poderes de exclusão do *job* Emergência.
- Quanto ao valor *"QoS"* faz a rotina buscar por requisições do tipo QoS. Nelas o campo prazo é obrigatório.
- Já o valor *"Comum"* faz rotina buscar por requisições de *jobs* comuns. Esses *jobs* não tem prazo nem poderes de exclusão.

A rotina remover_jobs_requisitados()

O usuário pode solicitar a exclusão de seu *job* ao escalonador, estando o *job* em qualquer estado. O usuário administrador pode excluir qualquer *job* que precisar. A rotina remover_jobs_requisitados pára a execução do *job*, se necessário, desaloca o *job* se o mesmo não for Candidato e retira o *job* de sua respectiva lista. O estado do *job* passa a ser Pós-Execução.

A rotina remover_jobs_de_tempo_esgotado()

Todos os *jobs* que são colocados em execução pelo OscarPBS tem um tempo de *walltime* definido pelo usuário, mais uma tolerância que é dada pelo próprio OscarPBS. O *walltime* de cada *job* é passado ao TORQUE pelo OscarPBS e o TORQUE passa a ser responsável por finalizar o *job* caso o mesmo não termine espontaneamente antes do *walltime* terminar. Normalmente o TORQUE finaliza o *job* que tenha estourado seu *walltime* alguns segundos após isso ter acontecido, porém em algumas ocasiões o TORQUE pode levar muito tempo para excluir o *job*, ou ficar esperando o administrador excluir o *job* caso alguma máquina do *cluster* tenha ficado indisponível. Se o TORQUE não excluir o *job* após o *walltime* ter acabado mais um prazo, o próprio OscarPBS verifica se existem *jobs* nessas condições na rotina remover_jobs_de_tempo_esgotado() e finaliza esses *jobs*.

A rotina desalocar_jobs_finalizados()

Os *jobs* que estavam em execução durante o ciclo de escalonamento anterior e que terminaram espontaneamente antes do ciclo atual e os *jobs* em execução que foram finaliza-

dos pela rotina remover_jobs_de_tempo_esgotado() estão na lista de *jobs* em execução quando na verdade não estão mais em execução. Essa rotina tem o objetivo de verificar esses *jobs* que acabaram de executar e sincronizar o estado dos mesmos, limpando a lista de execução e liberando recursos que estavam associados a esses *jobs*. Os *jobs* tratados nessa rotina passam para o estado de Pós-execução.

A rotina desalocar_jobs_alocados_atrasados()

Quando o escalonador entra em seu ciclo atual, alguns *jobs* poderiam estar já alocados para serem iniciados exatamente no instante_atual, ou com possivelmente um pequeno atraso. Normalmente o escalonador acorda e decide colocar esses *jobs* alocados que tenham alcançado o instante_atual, ou que estejam com um atraso pequeno em execução. Porém se acontecer um atraso muito grande, o intervalo de tempo previamente alocado para o *job* pode ficar menor que o *walltime* do *job*. Se existirem *jobs* alocados nessas condições, essa rotina desaloca esses *jobs* e os transforma em *jobs* Candidatos.

A rotina sincronizar_maquinas()

Essa rotina verifica o estado de cada máquina do *cluster*, verificando se a mesma está disponível ou indisponível e atualiza os *flags* de disponibilidade dessas máquinas para que as outras rotinas do escalonador saibam se podem utilizar essa máquina. Essa rotina também calcula o número máximo de CPUs disponíveis e essa variável é importante para determinar previamente se um *job* candidato tem condições de ser alocado.

A rotina criar_novos_jobs_candidatos(tipo)

Esta rotina sempre é chamada após a coletar_requisicoes_novas tendo o objetivo de criar *jobs* do tipo Comum, QoS ou Emergência. A rotina que coleta requisições coloca em uma lista global chamada de *ListaDeRequisicoes* as requisições de um dado tipo de *job* coletadas do Banco de Dados. A rotina criar_novos_jobs_candidatos então verifica cada uma das requisições da lista e para cada uma delas cria um novo *job*. Os campos da requisição são interpretados pelo método *parse_job* que é responsável por criar um *job*, definindo seu tipo, walltime, dono, criando os fragmentos do *job* e colocando cada fragmento em sua respectiva lista de fragmentos do *job*. Após criar o *job*, a rotina

criar_novos_jobs_candidatos coloca o *job* no final da lista de candidatos de acordo com a tipo do *job*. Mais detalhes sobre a rotina podem ser vistas no algoritmo 4.

```
1  criar_novos_jobs_candidatos(tipo)
2  |  para cada requisicao em ListaDeRequisicoes faça
3  |  |  job_novo = parse_job(requisicao);
4  |  |  se tipo == "Comum" então
5  |  |  |  ListaComumCandidatos.append(job_novo);
6  |  |  senão se tipo == "QoS" então
7  |  |  |  ListaQoSCandidatos.append(job_novo);
8  |  |  senão
   |  |  |  /* tipo == "Emergencia" */
9  |  |  |  ListaEmergenciaCandidatos.append(job_novo);
10 |  |  fim
11 |  fim
12 fim
```

Algoritmo 4: A rotina que cria novos *jobs* a partir de uma lista de requisições vindas do Banco de Dados.

A rotina alocar_jobs_Emergencia_Candidatos_novos()

Essa rotina trata *jobs* de Emergência que foram coletados na tabela de requisições durante o ciclo atual de escalonamento pela rotina coletar_requisicoes_job_Emergencia. Esses *jobs* são tratados pelo escalonador em ordem *FIFO*. O escalonador tenta inserir cada desses *jobs* inicialmente usando apenas o poder de desalocar *jobs Starve* Alocados. Se um determinado *job* não puder ser inserido com esse poder, a rotina tenta inserir novamente o *job*, com todos os poderes para desalocar outros *jobs*, de acordo com os poderes requisitados pelo administrador. A rotina pode ser vista no algoritmo 5.

```
1  alocar_jobs_Emergencia_Candidatos_novos()
2      para cada job em ListaEmergenciaCandidatos faça
3          se not job.requisitado_no_ciclo_atual então
4              continue;
5          fim
6          inicio_intervalo = instante_atual;
7          fim_intervalo = job.prazo;
8          lista_de_inicios_de_job = gerar_lista_inicios();
9          indice = 0;
10         continuar = True;
11         job_alocado = False;
12         enquanto continuar faça
13             job.inicio = encontrar_inicio(job, inicio_intervalo, fim_intervalo, "Final",
               "DesalocarStarves");
14             se job.inicio == -1 então
15                 continuar = False;
16             senão
17                 job_alocado = inserir_job(job, "DesalocarStarves");
18                 se job_alocado então
19                     continuar = False;
20                 senão
21                     se indice == tamanho(lista_de_inicios_de_job) então
22                         continuar = False;
23                     senão
24                         indice = procurar_novo_final(lista_de_inicios_de_job, indice,
                           fim_intervalo);
25                         se indice == -1 então
26                             continuar = False;
27                         senão
28                             fim_intervalo = lista_de_inicios_de_job[indice];
29                             indice = indice + 1;
30                         fim
31                     fim
32                 fim
33             fim
34         fim
35         se not job_alocado então
36             job.inicio = job.prazo - (job.walltime + tolerancia);
37             job_alocado = inserir_job(job, "DesalocarTodos");
38         fim
39         se job_alocado então
40             ListaEmergenciaCandidatos.remove(job);
41             ListaEmergenciaAlocados.append(job);
42             efetivar_eventos();
43         senão
44             desfazer_eventos();
45         fim
46     fim
47 fim
```

Algoritmo 5: A rotina que trata *jobs* Emergência Candidatos cujas as requisições foram tratadas durante o ciclo atual.

A rotina alocar_jobs_Emergencia_Candidatos_antigos()

Esta rotina tem o objetivo de tratar os *jobs* de Emergência que estão na fila de Candidatos. A diferença desta rotina com relação a alocar_jobs_Emergencia_Candidatos_novos é

que nesta os *jobs* que são tratados não conseguiram garantir a alocação em ciclos anteriores e também não foram removidos pelo usuário. O escalonador OscarPBS então tenta alocar novamente esses *jobs* sempre que tem uma oportunidade. Porém para esses *jobs*, o escalonador tenta alocar utilizando apenas o poder de *DesalocarStarves*, evitando assim que *jobs* Emergência que não puderam garantir seu espaço no momento da requisição cause desalocações indesejadas em um momento posterior. A rotina é apresentada no algoritmo 6

```
1  alocar_jobs_Emergencia_Candidatos_antigos()
2      ordenar_por_ordem_de_chegada(ListaEmergenciaCandidatos);
3      para cada job em ListaEmergenciaCandidatos faça
4          se job.requisitado_no_ciclo_atual então
5              job.requisitado_no_ciclo_atual = False;
6              continue;
7          fim
8          inicio_intervalo = instante_atual;
9          fim_intervalo = job.prazo;
10         lista_de_inicios_de_job = gerar_lista_inicios();
11         indice = 0;
12         continuar = True;
13         job_alocado = False;
14         enquanto continuar faça
15             job.inicio = encontrar_inicio(job, inicio_intervalo, fim_intervalo, "Final",
               "DesalocarStarves");
16             se job.inicio == -1 então
17                 continuar = False ;
18             senão
19                 job_alocado = inserir_job(job, "DesalocarStarves");
20                 se job_alocado então
21                     continuar = False;
22                 senão
23                     se indice == tamanho(lista_de_inicios_de_job) então
24                         continuar = False;
25                     senão
26                         indice = procurar_novo_final(lista_de_inicios_de_job, indice,
                           fim_intervalo);
27                         se indice == -1 então
28                             continuar = False;
29                         senão
30                             fim_intervalo = lista_de_inicios_de_job[indice];
31                             indice = indice + 1;
32                         fim
33                     fim
34                 fim
35             fim
36         fim
37         se job_alocado então
38             ListaEmergenciaCandidatos.remove(job);
39             ListaEmergenciaAlocados.append(job);
40             efetivar_eventos();
41         senão
42             desfazer_eventos();
43         fim
44     fim
45 fim
```

Algoritmo 6: A rotina que trata *jobs* Emergência Candidatos que não puderam ser alocados em ciclos anteriores.

A rotina alocar_jobs_QoS_Candidatos()

Esta rotina tenta alocar todos os *jobs* QoS Candidatos, selecionando os *jobs* da lista ListaQoSCandidatos na ordem em que foram requisitados pelos usuários. De modo semelhante ao que acontece com a rotina alocar_jobs_Emergencia_Candidatos_antigos esta

rotina tenta localizar uma posição para cada *job*, varrendo um intervalo de tempo que vai do instante atual até o prazo do *job*. Sempre que a rotina encontra uma posição dentro do intervalo de tempo, verificada previamente com a rotina encontrar_inicio, ele tenta inserir o *job* nessa posição utilizando a rotina inserir_job. A rotina pode ser vista com mais detalhes no algoritmo 7.

```
1  alocar_jobs_QoS_Candidatos()
2      ordenar_por_ordem_de_chegada(ListaQoSCandidatos);
3      para todo job em ListaQoSCandidatos faça
4          inicio_intervalo = instante_atual;
5          fim_intervalo = job.prazo;
6          lista_de_inicios_de_job = gerar_lista_inicios();
7          indice = 0;
8          continuar = True;
9          job_alocado = False;
10         enquanto continuar faça
11             job.inicio = encontrar_inicio(job, inicio_intervalo, fim_intervalo, "Final",
               "DesalocarStarves");
12             se if job.inicio == -1 então
13                 continuar = False;
14             else
15                 job_alocado = inserir_job(job, "DesalocarStarves");
16                 se if job_alocado então
17                     continuar = False;
18                 else
19                     se if indice == tamanho(lista_de_inicios_de_job) então
20                         continuar = False;
21                     else
22                         indice = procurar_novo_final(lista_de_inicios_de_job, indice,
                           fim_intervalo);
23                         se if indice == -1 então
24                             continuar = False;
25                         else
26                             fim_intervalo = lista_de_inicios_de_job[indice];
27                             indice = indice + 1;
28                         end
29                     end
30                 end
31             end
32         fim
33         se job_alocado então
34             ListaQoSCandidatos.remove(job);
35             ListaQoSAlocados.append(job);
36             efetivar_eventos();
37         else
38             desfazer_eventos();
39         end
40     fim
41 fim
```

Algoritmo 7: A rotina que trata todos os *jobs* QoS Candidatos, procurando alocar esses *jobs*.

A rotina procurar_novos_starve_candidatos()

Esta rotina verifica todos os *jobs* Comuns na lista de *jobs* Comuns Candidatos. Os *jobs* Comuns verificados que tiverem sido requisitados a mais tempo que a tolerância que OscarPBS dá para *jobs* comuns em espera antes desses serem considerados como estando em *starve*, deixam de estar na fila de *jobs* Comuns Candidatos e passam para a fila dos *jobs Starve* Candidatos. A rotina pode ser vista no algoritmo 8.

```
1 procurar_novos_starve_candidatos()
2 |   para cada job em ListaComumsCandidatos faça
3 |   |   se instante_atual > (job.instante_requisicao + tolerancia_starve) então
4 |   |   |   ListaComumsCandidatos.remove(job);
5 |   |   |   ListaStarvesCandidatos.append(job);
6 |   |   fim
7 |   fim
8 fim
```

Algoritmo 8: A rotina que transforma *jobs* Comuns que estão esperando a muito tempo em *jobs Starves* Candidatos.

A rotina alocar_jobs_Starve_Candidatos()

Nesta rotina, são tratados todos os *jobs Starve* Candidatos. O escalonador tenta alocar cada um desses *jobs* em intervalos de tempo que vão do instante atual até o último final do último *job* alocado ou em execução, incluindo o *walltime* e a tolerância desses *jobs*. No intervalo de tempo, o escalonador tenta alocar o *job Starve* inicialmente no instante atual e, se não conseguir, vai avançando a posição de tempo em direção ao último *job* alocado, tentando alocar o *job Starve*. A rotina pode ser vista de maneira mais detalhada no algoritmo 9.

```
1  alocar_jobs_Starve_Candidatos()
2      ordenar_por_ordem_de_chegada(ListaStarvesCandidatos);
3      para cada job em ListaStarvesCandidatos faça
4          inicio_intervalo = instante_atual;
5          lista_de_finais_de_job = gerar_lista_finais();
6          ultimo_indice = tamanho(lista_de_finais_de_job) - 1;
7          se ultimo_indice == -1 então
8              fim_intervalo = instante_atual + job.walltime + tolerancia;
9          else
10             fim_intervalo = lista_de_finais_de_job[ultimo_indice] + job.walltime +
               tolerancia;
11         end
12         indice = 0;
13         continuar = True;
14         job_alocado = False;
15         enquanto continuar faça
16             job.inicio = encontrar_inicio(job, inicio_intervalo, fim_intervalo, "Inicio",
               "NaoDesalocar");
17             se job.inicio == -1 então
18                 continuar = False;
19             else
20                 job_alocado = inserir_job(job, "NaoDesalocar");
21                 se job_alocado então
22                     continuar = False;
23                 else
24                     se indice == tamanho(lista_de_inicios_de_job) então
25                         continuar = False;
26                     else
27                         indice = procurar_novo_inicio(lista_de_finais_de_job, indice,
                           inicio_intervalo);
28                         se indice == -1 então
29                             continuar = False;
30                         else
31                             inicio_intervalo = lista_de_finais_de_job[indice];
32                             indice = indice + 1;
33                         end
34                     end
35                 end
36             end
37         fim
38         se job_alocado então
39             ListaStarvesCandidatos.remove(job);
40             ListaStarvesAlocados.append(job);
41             efetivar_eventos();
42         else
43             desfazer_eventos();
44         end
45     fim
46 fim
```

Algoritmo 9: A rotina que tenta alocar todos os *jobs Starves* Candidatos.

A rotina realocar_jobs_Starve_no_instante_atual()

Esta rotina verifica se existem *jobs Starves* alocados que estão no instante atual e coloca os mesmos na lista ListaAlocadosInstanteAtual. Todos os *jobs* que vão para essa lista serão colocados em execução no final do ciclo de escalonamento a menos que aconteça

algum erro como por exemplo uma máquina que venha a se tornar indisponível. Os *jobs Starves* Alocados que não estejam no instante atual serão temporariamente duplicados pela rotina duplicar_job e se possível realocados no instante atual. Se conseguir realocar o *job Starve*, o escalonador faz com que o *job* duplicado se torne o *job* oficial e o que estava alocado anteriormente é excluído. Os *jobs Starve* requisitados a mais tempo terão poder para desalocar *jobs Starves* mais novos, que não estejam no instante atual. Porém um *job* mais novo nunca pode desalocar um mais antigo, tendo em vista evitar que *jobs Starves* possam fazer outros esperarem por tempo indeterminado. Os *jobs Starves* que puderem ser realocados no instante atual vão para a lista ListaAlocadosInstanteAtual. A rotina pode ser vista com mais detalhes no algoritmo 10.

```
1  realocar_jobs_Starve_no_instante_atual()
2  |  ordenar_por_ordem_de_chegada(ListaStarvesAlocados);
3  |  para cada job em ListaStarvesAlocados faça
4  |  |  se job.inicio <= instante_atual então
5  |  |  |  ListaAlocadosInstanteAtual.append(job);
6  |  |  |  continue;
7  |  |  else
8  |  |  |  job_duplicado = duplicar_job(job);
9  |  |  |  exclusao_provisoria(job);
10 |  |  |  job_duplicado.inicio = instante_atual;
11 |  |  |  job_alocado = inserir_job(job_duplicado, "DesalocarStarvesMaisRecentes");
12 |  |  |  se job_alocado então
13 |  |  |  |  ListaStarvesAlocados.append(job_duplicado);
14 |  |  |  |  ListaAlocadosInstanteAtual.append(job_duplicado);
15 |  |  |  |  efetivar_eventos();
16 |  |  |  else
17 |  |  |  |  desfazer_eventos();
18 |  |  |  end
19 |  |  end
20 |  fim
21 fim
```

Algoritmo 10: A rotina que verifica se existem *jobs Starve* Alocados que podem ser realocados para o instante atual.

A rotina alocar_jobs_Comuns_candidatos_no_instante_atual()

Esta rotina tenta alocar todos os *jobs* Comuns Candidatos no instante atual. O Escalonador sempre tenta alocar os *jobs* Comuns no instante atual, tendo em vista colocar estes *jobs* em execução imediatamente eles nunca reservam recursos como acontece com outros tipos de *job*. No início da rotina a lista de *jobs* Comuns Candidatos é ordenada pelo peso dos *jobs*, tendo em vista colocar *jobs* menores para executar primeiro e reduzir o tempo de resposta médio de todos os *jobs*. Durante as tentativas de inserção de *jobs* Comuns, o poder para desalocar outros *jobs* é totalmente desabilitado, tendo em vista

que *jobs* Comuns tem baixa prioridade em relação aos outros tipos de *job*. A rotina pode ser vista em maiores detalhes no algoritmo 11.

```
1  alocar_jobs_Comuns_candidatos_no_instante_atual()
2  |   ordenar_por_peso(ListaComunsCandidatos);
3  |   para cada job em ListaComunsCandidatos faça
4  |   |   job.inicio = instante_atual;
5  |   |   job_alocado = inserir_job(job, "NaoDesalocar");
6  |   |   se job_alocado então
7  |   |   |   ListaAlocadosInstanteAtual.append(job);
8  |   |   |   efetivar_eventos();
9  |   |   else
10 |   |   |   desfazer_eventos();
11 |   |   end
12 |   fim
13 fim
```

Algoritmo 11: A rotina que tenta alocar os *jobs* Comuns Candidatos no instante atual.

A rotina realocar_jobs_QoS_Emergencia_no_instante_atual()

Esta rotina verifica se existem *jobs* alocados QoS e Emergência que estejam no instante atual e coloca os mesmos na lista ListaAlocadosInstanteAtual para que sejam executados no final do ciclo atual. Com os demais *jobs* QoS e Emergência Alocados, a rotina os coloca em uma lista e ordena pelo peso desses *jobs*. O escalonador então tenta realocar estes *jobs* no instante atual, criando uma duplicata do *job* e tentando inserir a duplicata no instante atual. A tentativa de inserção é feita sem nenhum poder para desalocar outros *jobs*, de modo que o *job* duplicado só poderá ser inserido se existirem recursos suficientes para ele. Se o *job* duplicado puder ser inserido, ele passa a ser o oficial e o *job* que estava alocado é excluído. Os *jobs* que puderem ser realocados no instante atual, são inseridos na lista ListaAlocadosInstanteAtual para que sejam executados no final do ciclo de escalonamento atual. Mais detalhes da rotina podem ser vistos no algoritmo 12.

```
realocar_jobs_QoS_Emergencia_no_instante_atual()
    ListaQoS_Emergencia = [];
    para cada job em ListaQoSAlocados faça
        se if job.inicio <= instante_atual então
            ListaAlocadosInstanteAtual.append(job);
            continue;
        else
            ListaQoS_Emergencia.append(job);
        end
    fim
    para cada job em ListaQoSAlocados faça
        se job.inicio <= instante_atual então
            ListaAlocadosInstanteAtual.append(job);
            continue;
        else
            ListaQoS_Emergencia.append(job);
        end
    fim
    ordenar_por_peso(ListaQoS_Emergencia);
    para cada job em ListaStarvesAlocados faça
        job_duplicado = duplicar_job(job);
        exclusao_provisoria(job);
        job_duplicado.inicio = instante_atual;
        job_alocado = inserir_job(job_duplicado, "NaoDesalocar");
        se job_alocado então
            se job_duplicado.tipo == Q então
                ListaQoSAlocados.append(job_duplicado);
            else
                ListaEmergenciaAlocados.append(job_duplicado);
            end
            ListaAlocadosInstanteAtual.append(job_duplicado);
            efetivar_eventos();
        else
            desfazer_eventos();
        end
    fim
fim
```

Algoritmo 12: A rotina que tenta realocar *jobs* QoS e Emergência já alocados no instante atual, se existirem recursos sobrando.

A rotina executar_jobs_alocados_no_instante_atual()

Esta rotina tenta colocar todos os *jobs* da lista ListaAlocadosInstanteAtual em execução, ou seja, todos os *jobs* do ciclo atual que estiverem alocados no instante atual. Se a tentativa de colocar algum *job* em execução falhar, a variável global algum_job_falhou recebe o valor *True* e isso fará com que o próximo ciclo de escalonamento não precise esperar. O *job* que falhar é desalocado e colocado no estado de Pós-escalonamento. Os recursos que estavam alocados para o *job* que falhou são liberados nesta rotina e voltam a ficar disponíveis para outros *jobs*. A rotina colocar_job_em_execucao monta um comando de submissão seguindo a sintaxe do TORQUE, especificando neste comando as máquinas selecionadas para comportar os fragmentos do *job* alocado, com o número de CPUs que

cada máquina deve disponibilizar e depois submete o *job* para o módulo *pbs_server* com o comando *qsub* do TORQUE. Logo em seguida, o escalonador o coloca em execução com o comando *qrun* do TORQUE. Havendo sucesso com os comandos *qsub* e *qrun*, a rotina atualiza os totais de cada intervalo de tempo nas listas das máquinas utilizadas pelo *job*, de modo a manter atualizado a forma como as CPUs das máquinas estão sendo utilizadas nos intervalos de tempo. Os detalhes da rotina executar_jobs_alocados_no_instante_atual podem ser vistos no algoritmo 13.

```
1  executar_jobs_alocados_no_instante_atual()
2  |  para cada job em ListaAlocadosInstanteAtual faça
3  |  |  conseguiu_executar = colocar_job_em_execucao(job);
4  |  |  se conseguiu_executar então
5  |  |  |  ListaJobsExecucao.append(job);
6  |  |  else
7  |  |  |  desalocar_job(job);
8  |  |  |  algum_job_falhou = True;
9  |  |  end
10 |  fim
11 fim
```

Algoritmo 13: A rotina que tenta colocar todos os *jobs* alocados no instante atual em execução.

As rotinas ordenar_por_ordem_de_chegada(ListaDeJobs) e ordenar_por_peso(ListaDeJobs)

Estas duas rotinas são utilizadas para ordenar as listas de *jobs* durante a execução das seguintes rotinas:

- alocar_jobs_Emergencia_Candidatos_antigos
- alocar_jobs_QoS_Candidatos
- alocar_jobs_Starve_Candidatos
- realocar_jobs_Starve_no_instante_atual
- alocar_jobs_Comuns_Candidatos_no_instante_atual
- realocar_jobs_QoS_Emergencia_no_instante_atual

Aqui é dado uma breve explicação do que fazem essas duas rotinas de ordenação.

Ambas as rotinas ordenam a lista de *jobs* *ListaDeJobs* passada como parâmetro de forma crescente considerando o valor de um campo de cada *job* na lista.

No caso da rotina ordenar_por_ordem_de_chegada, o campo do *job* que considerado para ordenação é job.instante_requisicao, que armazena o instante em que o *job* foi requisitado pelo usuário. Isso faz com que a lista de *jobs* se torne *FIFO*, onde os primeiros elementos da lista são aqueles que foram requisitados primeiro.

No caso da rotina ordenar_por_peso o campo usado é job.peso. Isso faz com que a lista fique ordenada pelo peso dos *jobs*. Isso tem o objetivo de colocar os *jobs* menores no início da lista.

5.5.2 As rotinas responsáveis por encontrar um instante de tempo para alocar *jobs*

As rotinas apresentadas aqui, tem por finalidade auxiliar o escalonador a decidir onde seria conveniente tentar inserir um *job*, com relação ao instante de tempo. Essas rotinas se baseiam em uma estrutura de dados com uma lista de intervalos de tempo globais da seção 4.4. Uma verificação prévia dos intervalos de tempo é feita para saber se uma posição no tempo tem condições mínimas para que o escalonador inicie uma tentativa de inserção do *job*.

As rotinas gerar_lista_inicios() e gerar_lista_finais()

As rotinas gerar_lista_inicios e gerar_lista_finais geram uma lista contendo os instantes iniciais ou os instantes finais de *jobs* alocados ou em execução, respectivamente. O método gerar_lista_inicios gera uma lista em ordem decrescente, enquanto o método gerar_lista_finais gera uma lista em ordem crescente. O método gerar_lista_finais define os finais dos *jobs* com sendo $fim\_job = job.inicio + job.walltime + tolerancia$.

As rotinas procurar_novo_inicio e procurar_novo_final

A rotina procurar_novo_inicio(lista_de_finais_de_jobs, indice, inicio_intervalo) retorna o índice do primeiro elemento da lista a partir do indice passado como parâmetro que seja maior que o parâmetro inicio_intervalo. Se a lista toda for lida mas nenhum elemento for maior que inicio_intervalo, a rotina returna -1. De maneira parecida, o método

procurar_novo_final(lista_de_inicios_de_jobs, indice, final_intervalo) retorna o índice do primeiro elemento da lista a partir do índice passado como parâmetro que seja menor que o parâmetro final_intervalo. Se a lista toda for lida mas nenhum elemento verificado for menor que fim_intervalo, a rotina returna -1.

A rotina encontrar_inicio

Dados um intervalo tempo e um *job*, essa rotina procura dentro desse intervalo um instante de tempo que possa coincidir com o início do *job* e que será usado como a posição inicial no momento em que a rotina inserir_job tentará inserir o *job*. O intervalo precisa ser maior ou igual ao *walltime* do *job* mais uma tolerância. A rotina retorna *True*, se puder encontrar uma posição dentro do intervalo, ou False se o intervalo for pequeno demais ou se a rotina não puder encontrar uma posição inicial para o *job*.

A busca pela posição inicial começa pelo início ou pelo final do intervalo, conforme um parâmetro passado para a rotina e avança em direção ao outro extremo do intervalo enquanto não puder encontrar o início procurado. Nessa rotina existe um *loop*, onde a cada passo, um subintervalo do intervalo de tempo original e do tamanho do *job* (*walltime* + tolerância) é definido e testado. Se o *job* for aceito nesse subintervalo, o início desse subintervalo é retornado pela rotina, senão, um novo subintervalo é criado se ainda for possível e o *loop* avança. O teste do subintervalo é feito pela verificação na lista de tempos globais, onde todos os intervalos de tempos globais que coincidem com o subintervalo devem possuir CPUs livres e desalocáveis suficientes para atender ao número de CPUs do *job* (soma das CPUs de todos os fragmentos do *job*). Esse verificação é feita pela rotina testar_intervalos_globais. Se um dos intervalos de tempo global que coincidem com o subintervalo testado não puder atender ao *job*, um novo subintervalo é definido numa posição imediatamente anterior ou posterior ao subintervalo testado. Posterior se a busca começou no início do intervalo passado como parâmetro, ou anterior se a busca começou pelo fim do intervalo. O parâmetro poder_de_desalocar determina qual o poder que o *job* possui para desalocar, o que influencia no número total de CPUs desalocáveis de cada intervalo de tempo global. Os totais de CPUs de um nó de intervalo de tempo global da lista de intervalos de tempo globais são obtidos indiretamente a partir da soma das CPUs dos intervalos de tempo de máquinas disponíveis que fazem parte da lista de intervalos de máquina daquele intervalo de tempo global. A rotina encontrar_inicio pode ser visto em detalhes no algoritmo 14.

```
1  encontrar_inicio(job, inicio, fim, inicio_da_busca, poder_de_desalocar)
2      se (fim - inicio) < (job.walltime + tolerancia) então
3          retorna False;
4      fim
5      se inicio_da_busca == "Inicio" então
6          inicio_subintervalo = inicio;
7          fim_subintervalo = inicio + job.walltime + tolerancia;
8      senão
9          fim_subintervalo = fim;
10         inicio_subintervalo = fim - (job.walltime + tolerancia);
11     fim
12     enquanto True faça
13         todos_couberam = testar_intervalos_globais(inicio_subintervalo, fim_subintervalo,
           job, poder_de_desalocar);
14         se todos_couberam então
15             retorna True;
16         senão
17             se inicio_da_busca == "Inicio" então
18                 se (fim_subintervalo + job.walltime + tolerancia) > fim então
19                     retorna False;
20                 senão
21                     inicio_subintervalo = fim_subintervalo;
22                     fim_subintervalo = inicio_subintervalo + job.walltime + tolerancia;
23                 fim
24             senão
25                 se (inicio_subintervalo - (job.walltime + tolerancia)) < inicio então
26                     retorna False;
27                 senão
28                     fim_subintervalo = inicio_subintervalo;
29                     inicio_subintervalo = fim_subintervalo - (job.walltime + tolerancia);
30                 fim
31             fim
32         fim
33     fim
34     retorna False;
35 fim
```

Algoritmo 14: A rotina que seleciona uma posição dentro de um intervalo de tempo para que o escalonador possa tentar inserir um *job*.

5.5.3 As rotinas responsáveis por selecionar máquinas para *jobs*

Esta subseção mostra as rotinas que estão diretamente envolvidas com a inserção de um *job* em uma posição de tempo já definida, mostrando a rotina inserir_job, que é uma das mais importantes, também são mostradas as rotinas efetivar_eventos e desfazer_eventos que são responsáveis por efetivar ou desfazer todas as ações de fragmentos de *jobs* empurrados e *jobs* desalocados durante o processo de inserção de um *job* e também são mostradas rotinas responsáveis pela seleção e verificação de máquinas adequadas para atender ao *job*.

A rotina inserir_job(job, poder_de_desalocar)

Esta rotina tem o objetivo de tentar inserir um *job* em um instante de tempo já definido. Como o *job* é composto de um conjunto de fragmentos a serem inseridos cada um em alguma máquina, a rotina então passa a ver o *job* como um conjunto de fragmentos a serem iniciados no mesmo instante de tempo do seu *job* e cada um de desses fragmentos precisa ser inserido com sucesso em alguma máquina, para que o *job* tenha sucesso em sua inserção. Cada fragmento fixo do *job* requer uma máquina que já foi definida pelo usuário durante a requisição. Quanto aos fragmentos livres, esses podem tentar todas as máquinas disponíveis.

Escolhida uma máquina para tentar inserir um fragmento do *job*, o método inserir_job verifica entre três possibilidades como explicado anteriormente. Isso é feito verificando a lista de intervalos de tempo da máquina, para se saber quantas CPUs existem livres em cada intervalo de tempo, ou quantas CPUs tem o potencial para serem liberadas em cada intervalo, seja por meio do uso do método *Empurrar* ou por desalocar *jobs*.

Se todos os intervalos de tempo da máquina que concorrem com o intervalo de tempo do fragmento a ser inserido tiverem CPUs livres suficientes para atender ao fragmento, a máquina é considerada como sendo do tipo *cabe*. Nesse caso a inserção do fragmento é feita imediatamente com o método inserir_fragmento. Se porém um ou mais intervalos de tempo da máquina não disponibilizarem CPUs suficientes, mas contiverem fragmentos que podem ser *Empurrados* ou desalocados e se ao empurrar ou desalocar esses fragmentos o número de CPUs do intervalo de tempo seja suficiente para atender ao fragmento sendo inserido, então a máquina é considerada *viável*. Sendo uma máquina *viável* a rotina inserir_job chamará um outra rotina chamada liberar_espaco, que tentará retirar fragmentos da máquina, usando inicialmente o método *Empurrar* em todos os fragmentos que puder, ou em último caso, desalocando *jobs* que o *job* a ser inserido tiver poder para desalocar. Uma máquina também pode ser considerada *inviável* para um fragmento, o que significa que algum intervalo não possui CPUs livres suficientes para atender ao fragmento e que os fragmentos alocados na máquina não podem ser movidas ou desalocadas em número suficiente. Nesse caso a rotina desiste se o fragmento a ser inserido na máquina for fixo, ou tenta outra máquina se o fragmento for livre.

Na caso de uma máquina ser viável para um fragmento, a rotina liberar_espaco é chamada com o fragmento, a máquina e o poder a ser usado para desalocar passados como parâmetros. Esse poder indica quais fragmentos alocados na máquina podem ser desalocados se isso for necessário. São quatro tipos de poder para desalocar existentes e cada um deles tem o seguinte significado:

NaoDesalocar: Esse poder significa que os fragmentos do *job* a serem inseridos não podem desalocar nenhum outro *job*. Ele é dado a *jobs* Comuns, a *Starve* Candidatos e também é dado a *jobs* QoS e Emergência se esses já estiverem alocados e o escalonador estiver tentando realocar eles no instante_atual.

DesalocarStarves: É um poder dado a *jobs* que ao serem inseridos podem desalocar *jobs Starve* já Alocados, desde que um limite de vezes que um *job Starve* possa ser desalocado não tenha sido alcançado. Esse poder normalmente é concedido a *jobs* de Emergência e a *jobs* QoS quando os mesmos estão no estado de Candidatos.

DesalocarStarvesMaisRecentes: Esse poder é dado apenas a *jobs Starve* Alocados, quando o escalonador tentar realocar esses *jobs* no instante_atual. Esse poder significa que o *job Starve* sendo inserido pode desalocar outro *Starve* desde que o *job* a ser desalocado não esteja no instante_atual e que tenha sido requisitado após o *job* a ser inserido, ou seja, o *job Starve* só poderá desalocar outros *jobs Starve* que sejam mais recentes do que ele.

DesalocarTodos: Esse poder é dado apenas a *job* de Emergência que acabaram de ser requisitados pelo usuário quando os mesmos não poderem ser inseridos após uma tentativa com o poder de DesalocarStarves. Com o poder DesalocarTodos, o *job* Emergência pode desalocar todos os tipos de *jobs* aos que teve permissão de excluir, de acordo com os valores dos campos em sua requisição no Banco de Dados.

A rotina de inserção de *job* pode ser vista em detalhes no algoritmo 15.

```
1  inserir_job(job, poder_de_desalocar)
2      conseguiu_todos_os_fragmentos = True;
3      fragmentos_alocados = [] ;
4      para cada fragmento em job.fragmentos_fixos faça
5          condicao = fragmento_fixo_cabe(fragmento, poder_de_desalocar);
6          se condicao == "cabe" então
7              tem_espaco = True;
8          senão se condicao == "viavel" então
9              tem_espaco = liberar_espaco(fragmento, fragmento.maquina,
               poder_de_desalocar);
10         senão
               /* inviavel */
11             tem_espaco = False;
12         fim
13         se tem_espaco então
14             inserir_fragmento(fragmento, fragmento.maquina);
15             fragmentos_alocados.append(fragmento);
16         senão
17             conseguiu_todos_os_fragmentos = False;
18             break;
19         fim
20     fim
21     se conseguiu_todos_os_fragmentos então
22         para cada fragmento em job.fragmentos_livres faça
23             maquina_bestfit = procurar_maquina_bestfit(fragmento);
24             se maquina_bestfit então
25                 maquina = maquina_bestfit;
26                 conseguiu_maquina = True;
27             senão
28                 conseguiu_maquina = False;
29                 maquinas_viaveis = procurar_maquinas_viaveis(fragmento,
                   poder_de_desalocar);
30                 para cada maquina_viavel em maquinas_viaveis faça
31                     tem_espaco = liberar_espaco(fragmento, maquina_viavel,
                       poder_de_desalocar);
32                     se tem_espaco então
33                         maquina = maquina_viavel;
34                         conseguiu_maquina = True;
35                     fim
36                 fim
37             fim
38             se conseguiu_maquina então
39                 inserir_fragmento(fragmento, maquina)
                   fragmentos_alocados.append(fragmento);
40             senão
41                 conseguiu_todos_os_fragmentos = False;
42                 break;
43             fim
44         fim
45     fim
46     se not conseguiu_todos então
47         para cada fragmento em fragmentos_alocados faça
48             desalocar_fragmento(fragmento);
49         fim
50     fim
51     retorna conseguiu_todos;
52 fim
```

Algoritmo 15: A rotina que tenta alocar um *job* em um instante de tempo definido.

As rotinas efetivar_eventos() e desfazer_eventos()

Durante a tentativa de inserção de um *job*, pela rotina inserir_job, alguns fragmentos de *jobs* já alocados podem ser empurrados para outras máquinas e alguns *jobs* podem ser desalocados se a rotina precisar liberar espaço em máquinas para inserir os fragmentos do *job*.

Se durante a tentativa de inserção de um *job*, os fragmentos de outros *jobs* fossem empurrados e *jobs* fossem desalocados imediatamente, isso poderia ser ruim ao escalonamento se ao final o *job* a ser inserido não obtivesse sucesso na inserção, causando a desalocação de *jobs* sem proveito. Assim, as ações efetuadas durante a tentativa da inserção de um *job* tem apenas caráter provisório, não tendo efeito real nos estados e nas alocações dos *jobs*. Quando o processo de inserção de um *job* termina, é possível saber se vale a pena ou não efetivar as ações de movimentações de fragmentos e desalocação de *jobs* ocorrida.

As ações de movimentação de fragmentos e exclusões de *jobs* são armazenadas em listas de que serão chamadas de listas de eventos. Existe uma lista de eventos para fragmentos que foram empurrados e outra para *jobs* que foram excluídos. Durante o processo de inclusão de um *job*, as movimentações de fragmentos feitas pelo método *Empurrar* e as exclusões de *jobs* feitas pelo método exclusao_provisoria(job) deixam que os fragmentos empurrados e os *jobs* excluídos continuem alocados em suas máquinas originais e permaneçam em seus estados de Alocado ou em Execução como se nada tivesse acontecido. Há porém um desconto dos recursos que esses fragmentos movidos e que *jobs* desalocados provisoriamente ocupavam em suas máquinas originais, o que possibilita que essas máquinas permitam a inserção, ainda que provisoriamente, dos fragmentos do *job* que está sendo inserido.

Com o sucesso ou o fracasso na inclusão do *job*, o escalonador decide se deve efetivar os efeitos dos eventos armazenados nas duas listas de eventos, ou se deve desfazer os efeitos dos eventos.

A rotina efetivar_eventos é chamada se o *job* é inserido com sucesso. Isso faz com que os fragmentos empurrados em caráter provisório passem a considerar a última máquina para onde foram empurrados como sua máquina oficial. Se o fragmento foi alocado em outras máquinas durante o processo de empurra ele é desalocado dessas máquinas, passando a ocupar apenas a última máquina para onde foi empurrado. Os *jobs* que tenham sido desalocados provisoriamente, são realmente desalocados. Se o *job* desalocado em questão

estiver em execução ele é interrompido e depois desalocado das máquinas onde estava. O estado do *job* muda, se ele era um *job* Alocado, ele passa a ser um *job* Candidato e se era um *job* em Execução ele passa a ser um *job* Pós-escalonamento.

A rotina desfazer_eventos é chamada se o *job* não pode ser inserido. Assim, os fragmentos empurrados provisoriamente voltam a ocupar recursos de suas máquinas originais e os *jobs* excluídos provisoriamente voltam a ocupar os recursos das máquinas onde estavam alocados. Todos os fragmentos empurrados e *jobs* excluídos voltam a ficar exatamente como estavam antes da tentativa de inserção do *job*.

A rotina fragmento_fixo_cabe(fragmento, poder_de_desalocar)

Esse método verifica se um fragmento fixo cabe em sua máquina requisitada retornando três possíveis valores: *"cabe"*, *"viável"* ou *"inviável"*. A fim de determinar o valor que precisa ser retornado, essa rotina verifica todos os intervalos de tempo da máquina na lista de intervalos de tempo da mesma que coincidam com o intervalo de tempo do fragmento a ser inserido. Com isso se faz possível determinar quantas CPUs livres cada intervalo de tempo possui e quantas CPUs poderiam ser liberadas no intervalo de tempo utilizando-se o método *Empurrar* para mover fragmentos que estejam ocupando esse intervalo de tempo. Ou então por meio de desalocação de fragmentos, de acordo com o poder para desalocar. Se todos os intervalos de tempo verificados tiverem CPUs livres suficientes para atender ao fragmento, o tipo *cabe* é retornado. Se pelo menos um intervalo não tiver fragmentos livres suficientes mas que possuam CPUs ocupadas que possam ser liberadas em número suficiente, sejam pelo pelo método *Empurrar* ou por desalocar fragmentos que possam ser desalocados de acordo com o poder do *job* e com o parâmetro poder_de_desalocar, isso significa que a máquina pode ser do tipo *viável* mas não mais um tipo *cabe*. Em último caso, se qualquer um dos intervalos da máquina que coincida com o intervalo de tempo do fragmento a ser inserido, não possuir CPUs livres ou que possam ser liberadas em número suficiente para atender o número de CPUs do fragmento, a máquina passa a ser considerada *inviável*

A rotina procurar_maquina_bestfit(fragmento)

Esta rotina procura entre as máquinas disponíveis aquela onde o fragmento *caiba* e ao mesmo tempo que seja a máquina possua o menor espaço livre disponível dentro do intervalo de tempo que o fragmento ocupa. A rotina calcular_espaco_livre(maquina,

fragmento) é usada para saber se a máquina cabe e se for o caso, quanto de espaço livre a máquina teria se o fragmento fosse inserido. Para calcular o espaço livre, a rotina calcular_espaco_livre verifica os nós de intervalo de tempo da lista de intervalos de tempo da máquina que concorrem com o intervalo de tempo do fragmento. Se algum nó de intervalo de tempo não possuir CPUs livres suficientes para atender ao fragmento, a rotina retorna -1 imediatamente. Se por outro lado existirem CPUs suficientes, o espaço livre do intervalo é calculado como número de CPUs livres vezes o comprimento do intervalo e somado a uma variável de espaco_livre_total. Depois de somar o espaço livre de todos os intervalos verificados, na variável espaco_livre_total, a rotina retorna o valor dessa variável. A rotina procurar_maquina_bestfit pode ser vista com mais detalhes no algoritmo 16.

```
1  procurar_maquina_bestfit(fragmento)
2      maquina_encontrada = null;
3      espaco_livre = -1;
4      para cada maquina em ListaMaquinas faça
5          se maquina.disponivel então
6              espaco_da_maquina = calcular_espaco_livre(maquina, fragmento);
7              se espaco_da_maquina != -1 então
8                  se (espaco_livre == -1) ou ( espaco_da_maquina < espaco_livre ) então
9                      espaco_livre = espaco_da_maquina;
10                     maquina_encontrada = maquina;
11                 fim
12             fim
13         fim
14     fim
15     retorna maquina_encontrada;
16 fim
```

Algoritmo 16: A rotina que encontra uma máquina *bestfit* se a mesma existir, para um fragmento de *job*.

A rotina procurar_maquinas_viaveis(fragmento, poder_de_desalocar)

Esta rotina é chamada pela inserir_job caso a rotina procurar_maquina_bestfit não consiga achar nenhuma máquina onde o fragmento livre poderia caber sem a necessidade de liberar espaço. Isso significa que só existem máquinas do tipo *viável* ou *inviável* com relação ao fragmento a ser inserido. A rotina então verifica quais são as máquinas viáveis e coloca todas em uma lista. Enquanto encontra as máquinas viáveis, essa rotina classifica as máquinas quanto a uma categoria e classifica com relação a quanto de espaço total precisaria ser liberado na máquina para que o fragmento pudesse ser inserido. Com esses dois valores de classificação em cada máquina da lista de máquinas viáveis, a lista é ordenada, pela categoria da máquina em ordem crescente e também, como um critério de desempate para máquinas de uma mesma categoria, pelo quanto de espaço seria preciso

Tipo do *job*	Estado do *job*	Valor do incremento
Starve	Alocado	1
QoS	Alocado	2
Emergência	Alocado	4
Comum	Execução	8
Starve	Execução	16
QoS	Execução	32
Emergência	Execução	64

Tabela 5.6: Valores incrementais para o cálculo das categorias das máquinas viáveis, de acordo com os *jobs* que ela pode vir a precisar desalocar

liberar na máquina em ordem decrescente. A ordenação da lista de máquinas viáveis é feita pelas rotinas ordenar_por_espaco_a_liberar e ordenar_por_categoria.

No início da rotina, o valor zero é atribuído a categoria de todas máquinas e também ao quanto seria preciso liberar de recursos em cada máquina para que o fragmento caiba. A rotina zerar_categorias_e_espaco_a_liberar é responsável por zerar essas duas variáveis em cada uma das máquinas.

Com respeito ao valor de categoria de uma máquina, ele é incrementado quando um determinado tipo de *job* á encontrado. Máquinas com *jobs* importantes com risco de serem desalocados pela rotina liberar_espaco devem ser evitadas, assim máquina com *jobs* de menor importância tem categorias menores e são escolhidas antes que máquinas com *jobs* importantes.

Cada tipo de *job*, quando encontrado pela primeira vez em uma máquina, incrementa a categoria da máquina em um valor definido na tabela. Cada tipo diferente participa do incremento uma única vez.

Durante o incremento do valor de categoria da máquina, se leva em consideração o parâmetro poder_de_desalocar. Se o poder é "NaoDesalocar" as categorias nunca são incrementadas. Se o poder é "DesalocarStarve" ou "DesalocarStarvesMaisRecentes" apenas *jobs* do tipo *Starve* Alocado são considerados. Quando o poder utilizado é "DesalocarTodos" toda a tabela de valores é considerada. Uma tabela com os valores incrementais que cada *job* alocado na máquina pode acrescentar a categoria da máquina pode ser vista em 5.6.

O método viabilidade_da_maquina, verifica os nós de intervalo de tempo da máquina

que concorrem com o intervalo de tempo do fragmento. É verificado se o intervalo de tempo é do tipo *cabe*, *viável* ou *inviável*. Se for do tipo *inviável*, a rotina retorna imediatamente um registro de resultados com o valor do campo eh_viavel sendo False, indicando que a máquina não é viável. Se por outro lado, o nó do intervalo de tempo é do tipo *cabe*, esse nó é ignorado e a rotina apenas passa para o nó seguinte. Se o nó de intervalo de tempo da máquina for do tipo *viável* de acordo com os fragmentos que podem ser empurrados ou desalocados com o parâmetro poder_de_desalocar, a rotina calcula quanto espaço precisa ser liberado no intervalo de tempo, multiplicando o número de CPUs a liberar pelo comprimento do intervalo e verifica a categoria da máquina verificando quais tipos de *jobs* existem na lista de fragmentos do nó de intervalo de tempo. A rotina procurar_maquinas_viaveis pode ser vista com mais detalhes no algoritmo 17.

```
1  procurar_maquinas_viaveis(fragmento, poder_de_desalocar)
2      maquinasViaveis = [] ;
3      zerar_categorias_e_espaco_a_liberar() ;
4      para cada maquina em ListaMaquinas faça
5          se maquina.disponivel então
6              resultado = viabilidade_da_maquina(fragmento, maquina, poder_de_desalocar);
7              se resultado.eh_viavel != -1 então
8                  maquina.categoria = resultado.categoria;
9                  maquina.espaco_a_liberar = resultado.espaco_a_liberar;
10                 maquinasViaveis.append(maquina);
11             fim
12         fim
13     fim
14     ordenar_por_espaco_a_liberar(maquinasViaveis);
15     ordenar_por_categoria(maquinasViaveis);
16     retorna maquinasViaveis;
17 fim
```

Algoritmo 17: A rotina procura e cria uma lista de máquina viáveis para um fragmento de *job*.

A rotina inserir_fragmento(fragmento, maquina)

Esse método insere um fragmento em uma máquina onde o fragmento caiba. Isso significa que a lista com nós de intervalos de tempo da máquina serão atualizadas. Os nós de intervalos de tempo da máquina que fizerem interseção com o intervalo de tempo do fragmento a ser inserido, serão atualizados quanto a seus totais de CPUs ocupadas e quanto a suas respectivas listas de fragmentos, que passarão a conter o fragmento a ser inserido. Novos nós de intervalos de tempo podem ser criados caso existam descontinuidades na lista de intervalos de tempo dentro do intervalo tempo onde o fragmento está sendo inserido. Novos nós de intervalos que tempo que venham a ser criados na lista de intervalos de tempo da máquina são inseridos na lista de tempos global. A lista

de tempos global também pode sofrer modificações, como com o acréscimo de novos nós se isso for preciso para que a lista global conter os novos nós de intervalo de tempo da máquina.

A rotina desalocar_fragmento(fragmento)

Esta rotina remove um fragmento da máquina onde estiver inserido. Isso significa que a lista com nós de intervalos de tempo da máquina serão atualizadas. Os nós de intervalos de tempo da máquina que fizerem interseção com o intervalo de tempo do fragmento a ser removido, serão atualizados quanto a seus totais de CPUs ocupadas e quanto as suas respectivas listas de fragmentos, que deixarão de conter o fragmento a ser removido. Nós de intervalos de tempo da máquina podem ser removidos caso suas respectivas listas de fragmentos fiquem vazias após a remoção do fragmento tratado pela rotina. Os nós de intervalos que tempo que venham a ser removidos da lista de intervalos de tempo da máquina, serão removidos da lista de tempos global. Os nós da lista de intervalos de tempo global também podem ser removidos se suas respectivas listas de nós de intervalo de tempo de máquina ficarem vazias.

5.5.4 As rotinas responsáveis por liberar espaço em máquinas e a rotina que implementa o método *Empurrar*

Aqui são mostradas rotinas que auxiliam na liberação de recursos das máquinas escolhidas, tendo foco principal na rotina liberar_espaco e na rotina que implementa o método *Empurrar* chamada de empurra_fragmento.

A rotina liberar_espaco(fragmento, maquina, poder_de_desalocar)

Esta rotina tem o objetivo de liberar espaço em uma máquina que é *viável* para um fragmento, de modo a fazer com que essa máquina passe a comportar o fragmento, ou seja, que se torne uma máquina tipo *cabe*.

Para isso, esse método verifica cada nó de intervalo de tempo da máquina, na lista de intervalos de tempo da máquina verificando quais intervalos concorrem com o intervalo de tempo do fragmento a ser inserido. Os nós que sejam do tipo *cabe* são ignorados, mas os nós que sejam do tipo *viável* são considerados, de modo que a rotina verifica

quantas CPUs é preciso liberar nesses nós e quantas CPUs ele poderia liberar nesses nós, armazenando esses valores em campos temporários de cada nó. Também é verificado quais fragmentos o nó de tempo possui, classificando alguns desses fragmentos como empurráveis e/ou excluíveis e colocando esses fragmentos em listas de empurráveis ou excluíveis de acordo com sua classificação. Durante essa verificação dos nós de intervalo de tempo da máquina, os que são *viáveis* podem ter fragmentos empurráveis e/ou excluíveis. Os fragmentos empurráveis mantém uma variável temporária que armazena um valor de o quanto o fragmento tem de recursos a liberar. Esse valor a ser liberado pelo fragmento é uma soma dos recursos que o fragmento pode liberar em cada intervalo da máquina, sendo que o intervalo de tempo deve coincidir com o tempo do fragmento a ser inserido e com o tempo do fragmento a ser liberado.

Após geradas as listas de fragmentos empurráveis e *jobs* excluíveis, que no algoritmo 18 está abstraído pela *loop* que contém a rotina classificar_fragmentos_do_intervalo_de_tempo_da_maquina(), a rotina ordenará os fragmentos empurráveis pelo quanto cada fragmento liberaria de recursos. Os fragmentos que podem liberar mais recursos são os preferidos. No método a ordenação é feita pela rotina ordenar_por_recursos_a_liberar(lista_de_fragmentos_empurraveis). Após a lista estar ordenada, o método tentará empurrar todos os fragmentos *Empurráveis*. Então a rotina entra em um *loop* para tratar cada fragmento empurrável da lista. No corpo do *loop*, é feita uma verificação se o fragmento atual do *loop* ainda precisa ser empurrado, isso porque durante o processo de empurrar os fragmentos, alguns deles podem não mais precisarem ser empurrados. Isso é feito se verificando os nós de intervalos de tempo da máquina que tem interseção com o fragmento a ser inserido e com o fragmento a ser empurrado. Se algum desses nós ainda tiver CPUs a serem liberadas, então a rotina tenta empurrar o fragmento. O método *Empurrar* é então chamado para o fragmento. O fragmento pode ser empurrado ou não. Se não for empurrado, a rotina desconta de cada intervalo de máquina aquilo que é possível liberar dela e verifica se o intervalo de tempo ainda é viável, ou seja, se o número de CPUs que ainda podem ser liberadas mais as CPUs livres atendem ao fragmento a ser inserido. Se algum intervalo de tempo se tornar inviável, a rotina liberar_espaço termina, retornando False para indicar que não conseguiu liberar o espaço na máquina. Se por outro lado o fragmento for empurrado, a rotina desconta de cada intervalo o número de CPUs que precisam ser liberadas e o número de CPUs que podem ser liberadas. Conseguindo empurrar, o total a liberar também é descontado e se esse valor chegar a zero, isso significa que os recursos necessários foram liberados e a rotina sai com o valor *True*.

Após tentar empurrar todos os fragmentos *Empurráveis* se a rotina ainda não saiu

Valor da Categoria	Tipo de *job*	Estado do *job*
0	*Starve*	Alocado
1	*Starve* Crítico	Alocado
2	QoS	Alocado
3	Emergência	Alocado
4	Comum	Execução
5	*Starve*	Execução
6	QoS	Execução
7	Emergência	Execução

Tabela 5.7: Valores de categorias para *jobs*, de acordo com seu tipo e estado

com *True* ou False, isso significa que existem *jobs* a ser excluídos. A partir desse ponto a rotina só precisa excluir os *jobs* da lista de *jobs* excluíveis e verificar a cada *job* excluído se o espaço necessário foi liberado.

Inicialmente a rotina entra em um *loop* para calcular quanto que cada *job* a ser excluído tem de recursos a liberar na máquina. Esses recursos são somados considerando cada intervalo de tempo comum entre o *job* a ser excluído e o fragmento a ser incluído. Alguns *jobs* da lista podem ter deixado ter recursos a liberar, pois seus fragmentos podem ter sido empurrados no *loop* de fragmentos *Empurráveis* ou a fato de alguns fragmentos terem sido empurrados pode ter levado ao a situação em que esse *job* não mais precisa ser excluído. Apenas os *jobs* que ainda tem recursos a liberar continuam na lista.

Após calcular o valor de quanto cada *job* excluível pode colaborar em recursos a liberar, a rotina ordena a lista dos *jobs* excluíveis, primeiro pela categoria do *job*, os de categoria menor ou em outras palavras, aqueles de menor importância, são excluídos primeiro. Depois como critério de desempate da categoria do *job*, é usado o valor de quanto o *job* pode liberar de recursos. Aqueles que puderem liberar mais recursos são excluídos primeiro.

As categorias de *job* são verificadas quando a lista de *jobs* excluíveis é criada pela primeira vez. Nessa lista de categorias, o *job Starve* Alocado é considerado o *job* de menor importância (com seja, um *job* cujo o impacto de desalocar não é grande) enquanto um *job* Emergência em execução é considerado como sendo de importância máxima. Os valores de categoria para os *jobs* excluíveis podem ser vistos na tabela 5.7.

Onde *Starve* Crítico significa um *job Starve* que já foi desalocado o número máximo

de vezes que o OscarPBS permite e que não pode ser ser desalocado com o poder do tipo *DesalocarStarves*

Agora a rotina liberar_espaco executa um *loop* para verificar cada *job* excluível. Primeiro, a rotina verifica se o *job* ainda precisa ser excluído, pois durante a exclusão de *jobs* no *loop* alguns podem não precisar mais serem excluídos. Isso é feito com uma verificação dos intervalos de tempo que são comuns ao *job* a ser excluído e ao fragmento a ser incluído. Se nenhum deles tiver CPUs a liberar, o *job* não precisa mais ser excluído.

Se por outro lado algum intervalo ainda tiver CPUs a liberar, o *job* é excluído provisoriamente e todos os intervalos da máquina tem seu valor de CPUs a liberar descontados daquilo com que o *job* pode colaborar com cada um deles. O total de recursos a liberar é descontado e se o mesmo chegar a zero, a rotina sai retornando *True*.

Os detalhes da rotina liberar_espaco podem ser vistos no algoritmo 18.

```
1  liberar_espaco(fragmento, maquina, poder_de_desalocar)
2      lista_de_fragmentos_empurraveis = [];
3      lista_de_jobs_excluiveis = [];
4      para cada intervalo_de_tempo em maquina.intervalos_de_tempo faça
5          se intervalo_concorre(intervalo) então
6              classificar_fragmentos_do_intervalo_de_tempo_da_maquina();
7          fim
8      fim
9      ordenar_por_recursos_a_liberar(lista_de_fragmentos_empurraveis);
10     para cada fragmento em lista_de_fragmentos_empurraveis faça
11         precisa_empurrar = verificar_se_ainda_precisa_empurrar(fragmento);
12         se precisa_empurrar então
13             conseguiu_empurrar = empurrar(fragmento);
14             se conseguiu_empurrar então
15                 descontar_total_a_liberar();
16                 se total_a_liberar == 0 então
17                     retorna True;
18                 fim
19                 descontar_CPUS_a_liberar_nos_intervalos_de_tempo();
20                 descontar_CPUS_a_liberaveis_nos_intervalos_de_tempo();
21             senão
22                 descontar_CPUS_a_liberaveis_nos_intervalos_de_tempo();
23                 intervalo_inviavel = verificar_viabilidade_intervalos_de_tempo();
24                 se intervalo_inviavel então
25                     retorna False;
26                 fim
27             fim
28         fim
29     fim
30     para cada job em lista_de_jobs_excluiveis faça
31         calcular_total_a_liberar(job);
32         se job.total_a_liberar == 0 então
33             lista_de_jobs_excluiveis.remove(job);
34         fim
35     fim
36     ordenar_por_recursos_a_liberar(lista_de_jobs_excluiveis);
37     ordenar_por_categoria(lista_de_jobs_excluiveis);
38     para cada job em lista_de_jobs_excluiveis faça
39         precisa_excluir = verificar_se_ainda_precisa_excluir(job);
40         se precisa_excluir então
41             exclusao_provisoria(job);
42             descontar_total_a_liberar();
43             se total_a_liberar == 0 então
44                 retorna True;
45             fim
46             descontar_CPUS_a_liberar_nos_intervalos_de_tempo();
47         fim
48     fim
49 fim
```

Algoritmo 18: A rotina que tenta liberar espaço em uma máquina viável para que um fragmento de *job* possa ser inserido na máquina.

A rotina empurrar_fragmento(fragmento)

Aqui será descrito o algoritmo da rotina empurrar_fragmento() que implementa o método *Empurrar* visto em seções anteriores. Essa rotina tenta retirar um fragmento da máquina

onde esse fragmento está, empurrando ele para qualquer outra máquina disponível. Para isso, a rotina precisa encontrar uma máquina *bestfit*.

Se uma maquina *bestfit* é encontrada, é feita uma movimentação provisória da máquina onde o fragmento está para a máquina *bestfit*. Durante a movimentação, o fragmento não é realmente desalocado de sua máquina original, tendo apenas as CPUs que ele ocupa na máquina sendo descontadas dos nós de intervalos de tempo da máquinas em que o fragmento está inserido.

Durante o processo de inclusão de um *job*, um fragmento qualquer pode ser empurrado mais de uma vez, assim as máquinas que o fragmento percorre durante as ações de empurra ficam armazenadas em uma lista do próprio fragmento. A máquina original do fragmento também é salva em uma variável do mesmo, caso o fragmento esteja sendo empurrado pela primeira vez no processo de inclusão de um *job*. Se o fragmento voltar para um máquina onde já esteve durante o processo de inserção do *job*, ele não precisa ser criado nas estruturas de intervalo de tempo da máquina. Nesse caso, o número de CPUs que o fragmento necessita são descontados dos totais relativos aos intervalos de tempo da máquina.

A função remover_totais(fragmento,maquina) desconta de cada intervalo de tempo da máquina que contém o fragmento, o número de CPUS do intervalo de tempo que o fragmento ocupa. Isso faz com que o fragmento ainda esteja alocado nos intervalos de tempo da máquina, mas sem participar dos totais de CPUs. Isso permite que outros fragmentos possam utilizar essas CPUs que foram provisoriamente liberadas.

A função restaurar_totais(fragmento, maquina), acrescenta a cada intervalo de tempo da máquina que possui o fragmento, as CPUs utilizadas por esses mesmo fragmento e que em algum momento foram removidas pela rotina remover_totais. A rotina empurrar_fragmento pode ser vista em detalhes no algoritmo 19.

```
1  empurrar_fragmento(fragmento)
2  |  maquina = encontrar_maquina_bestfit("empurra");
3  |  se maquina != Null então
4  |  |  remover_totais(fragmento,maquina);
5  |  |  se fragmento.maquina_original == Null então
6  |  |  |  fragmento.maquina_original = fragmento.maquina;
7  |  |  fim
8  |  |  se maquina not in fragmento.lista_de_maquinas então
9  |  |  |  inserir_fragmento(fragmento,maquina);
10 |  |  senão
11 |  |  |  restaurar_totais(fragmento, maquina);
12 |  |  fim
13 |  |  se fragmento not in lista_de_eventos_empurra então
14 |  |  |  lista_de_eventos_empurrar.append(fragmento);
15 |  |  fim
16 |  |  retorna True ;
17 |  senão
18 |  |  retorna False;
19 |  fim
20 fim
```

Algoritmo 19: A rotina que tenta empurrar um fragmento para outra máquina.

Capítulo 6

Resultados

Este capítulo apresenta os resultados dos experimentos realizados com o OscarPBS. Apresentamos inicialmente a metodologia utilizada e depois a avaliação do desempenho do escalonador considerando cada uma das funcionalidades propostas.

6.1 Metodologia

6.1.1 O Ambiente de Execução

Todos os experimentos descritos nesse trabalho foram realizados no *cluster* Oscar. O Oscar possui 42 nós, sendo 2 servidores e 40 destinados a execução de *jobs* de usuário. Sobre os dois servidores do Oscar, cujos os *hostnames* são *uff0* e *uff1*, o *uff0* funciona como o servidor principal onde ficam os serviços principais, enquanto o *uff1* funciona como um servidor de manutenção, mantendo os serviços do *uff0* funcionando. O servidor *uff1* não participa do escalonamento diretamente, assim ele será pouco comentado a partir daqui. O Oscar possui dois *switches gigabit* de 48 portas conectados entre si, que são suficientes para conectar todas as máquinas do *cluster*, onde os dois servidores e os 40 nós ficam conectadas, criando uma rede local do próprio *cluster*. Cada um dos dois servidores possui mais uma interface de rede configuradas com endereço de internet, o que permite que os servidores sejam acessados remotamente utilizando um cliente de terminal seguro (*ssh*). O servidor *uff0* está conectado a um *data storage* que faz parte do *cluster* e serve como pasta pessoal para todos os usuários.

Quanto a configuração de hardware e software de cada máquina, o servidor *uff0* é uma máquina com dois processadores Xeon quadcore de 2,0 GHz e 8 GB de memória RAM. Ele possui dois *HDs* em *RAID*, onde fica instalado o sistema operacional, Linux

Redhat 5.3 de 64 bits. O servidor *uff1* é idêntico ao *uff0* em termos de hardware e SO. Os servidores estão conectado diretamente por fibra ótica ao *data storage* e a um dos *switchs Gigabit*. Os nós de computação são todos iguais em termos de hardware e softwares, elas tem *hostname* de *uff2* até *uff41*. Cada nó de computação possui dois processadores Xeon quadcore de 2,66 GHz, 16 GB de memória RAM e um *HD* de 160 GB, onde fica instalado o Sistema Operacional RedHat 5.3 de 64 bits. Todos os nós de computação possuem um diretório compartilhado dentro do próprio *cluster*. Esse diretório existe no *data storage* e é exportado pelo nó *uff0*, para os nós de computação. Esse diretório é a base de todos os diretórios *home* dos usuários. Os nós de computação também possuem diretórios locais temporários que podem ser utilizados pelos usuários como rascunho. O servidor *uff0* possui um serviço TORQUE instalado e esse é responsável por atender as submissões dos usuários. O escalonamento é feito pelo escalonador padrão do TORQUE, que também fica em execução na máquina *uff0*. Todas os nós de computação possuem serviços que atendem a requisições de submissão do TORQUE. Esse serviço é responsável por executar e parar as aplicações paralelas dos usuários nas máquinas de cálculo. Os usuários não têm permissão para acessar as máquinas de cálculo diretamente, assim só podem executar seus experimentos utilizando um script com comandos que são executados pelo TORQUE.

O servidor *uff0* possui compiladores C, C++ e Fortran da Intel e da GNU, assim como bibliotecas para programação paralela, como MPI e OpenMP. A uff0 também possui softwares prontos, para resolução de problemas que podem ser executados em paralelo, como Gaussian, VASP, GAMESS. Os nós de computação possuem em seus discos locais bibliotecas que dão suporte a execução desses programas.

Para realizar os experimentos, foram instalados no *cluster* mais duas instâncias do *D-RMS* TORQUE, totalizando três instalações de TORQUE. Uma das instalações está em produção, sendo utilizado pelos usuários para submissões de *jobs* reais, ele possui dois serviços em execução na *uff0*, o *pbs_server* e o *pbs_sched*, e um serviço em cada nó de computação, o *pbs_mom*. Assim, além dessa primeira instalação do TORQUE, mais duas foram feitas apenas para os experimentos. Uma delas permaneceu com o serviço escalonador TORQUE padrão e a outra teve o serviço escalonador substituído pelo OscarPBS. Cada uma com seus serviços *pbs_server*, *pbs_sched* e *pbs_mom*.

Para que as três instalações de TORQUE convivessem sem conflito no mesmo *cluster*, foi preciso reconfigurar as portas *TCP/IP* utilizadas por duas das instalações. Os recursos do *cluster* requisitados por *jobs* submetidos em cada uma das versões do TORQUE não afetam as outras. Como os *jobs* sintéticos dos experimentos têm processos que não ocupam

CPUs, eles podem ser executados junto com *jobs* reais de usuários sem utilizar recursos reais de forma significativa.

6.1.2 Entrada de Jobs

Coleta de Submissões Reais

Para a realização dos experimentos, se decidiu utilizar um histórico de submissões de *jobs* reais para servir como modelo para as submissões artificiais de *jobs*. Históricos de submissões de *jobs* reais em *clusters* podem ser obtidos a partir de algum *cluster* real a que se tenha acesso, ou a partir de históricos disponíveis na internet, como por exemplo o Parallel Workloads Archive [7]. Nesse trabalho se decidiu utilizar um histórico de submissões reais feita no *cluster* Oscar, por professores e alunos dos Institutos de Computação, Instituto de Física e Instituto de Química foi capturado para se criar um lote de submissões sintéticas que foi utilizado durante os experimentos. As submissões reais foram feitas para uma instalação de TORQUE versão 3.2, utilizando seu escalonador padrão. O serviço *D-RMS* do TORQUE (*pbs_server*) gera um *log* de todas as submissões e do término de cada *job*, guardando detalhes como o dono do *job*, as requisições feitas, o momento em que a submissão foi feita, o instante em que o *job* entra em execução e o tempo *walltime* utilizado pelo *job*, entre outras informações. Os *jobs* do histórico variam bastante de tamanho, em relação ao número de CPUs e de nós. Em alguns desses *jobs*, os usuários especificam as máquinas a serem utilizadas e em outros deixam as máquinas livres. As escolhas dos usuários quanto a seus *jobs*, escolhendo se as máquinas são fixas ou não no *job*, indica se o usuário depende de máquinas pré-definidas quando executa sua aplicação.

Os usuários submetem os *jobs* em um único nó do *cluster*, cujo *hostname* é *uff0*, passando um arquivo de script como parâmetro durante a submissão. Esse arquivo é responsável por chamar a aplicação paralela do usuário. O script do usuário é iniciado em uma das máquinas que foi disponibilizada para o *job* pelo escalonador e é responsável por chamar a aplicação paralela do usuário. Após isso, é possível que a aplicação do usuário gere saída em forma de arquivos e mensagens para a saída padrão. Tais arquivos de saída do *job* do usuário geralmente ficam em um diretório do próprio usuário, sendo que esse diretório é compartilhado por todas as máquinas do *cluster*. As mensagens que seriam geradas na saída padrão ficam armazenadas em um arquivo criado pelo TORQUE e que o usuário pode visualizar após o término de seu *job*.

Geração de Submissões Sintéticas

Com o objetivo de criar um ambiente onde se pudesse utilizar, testar e comparar o desempenho do OscarPBS com o escalonador do TORQUE, foram criados scripts para submeter *jobs* automaticamente, de uma maneira que tal submissão de *jobs* artificial se aproximasse o máximo possível de um cenário de submissões reais, onde usuários reais submetem *jobs* reais para um escalonador. Para que as submissões sintéticas fossem realistas, foram considerados alguns aspectos das submissões reais e as mesmas replicadas nas submissões sintéticas. No histórico coletado a partir das submissões reais, 318 *jobs* foram submetidos por oito usuários diferentes, então foram criados oito usuários no sistema, que serão chamados aqui de usuários sintéticos, com as mesmas características dos usuários reais e foi feito um mapeamento da requisição sintética com seu respectivo usuário sintético, mantendo a mesma relação de usuário e submissão que existe na submissão real. Os *jobs* sintéticos são 318, onde cada um é baseado em um *job* real e as seguintes características do *job* são observadas: No *job* sintético, o pedido por máquinas é idêntico ao que foi feito no *job* real, mantendo os mesmos pedidos por máquinas fixas se for o caso. No caso dos *jobs* reais, não há *walltime* na submissão, porém no *job* sintético correspondente um *walltime* é criado, tomando como base o tempo de *walltime* utilizado pelo *job* real, esse *walltime* utilizado é encontrado no *log* do TORQUE utilizado no *cluster* Oscar. Em cada *job* real existe um script criado pelo usuário que chama a aplicação do usuário. No caso do *job* sintético, o script que faz as submissões cria um diretório com o mesmo nome do *jobid* do *job* real, copia para dentro desse diretório um script sintético que é utilizado por todos os *jobs* sintéticos e copia para esse diretório um aplicativo MPI sintético, para imitar o comportamento de um *job* real. O script sintético chama o aplicativo MPI utilizando as máquinas e o número de CPUs da requisição sintética. O aplicativo MPI cria processos em todas as máquinas na mesma quantidade de CPUs por máquina, porém tais processos executam o comando *sleep* e ficam suspensos. Eles não ocupam as CPUs que pediram, pois o *cluster* executa aplicações reais enquanto tais experimentos são feitos. Como o experimento simula vários *jobs* paralelos, seria muito custoso reservar máquinas para fazer os experimentos do escalonador. Mesmo diante dessas limitações optou-se por não simular o escalonador, pois foi considerado que fazer os experimentos em um *cluster* com máquinas reais, mesmo que as aplicações não possuam carga, poderia ter um comportamento mais realista do que um ambiente de *cluster* simulado. Dois scripts de submissão de *jobs* sintéticos são iniciados simultaneamente, um para submeter *jobs* para o TORQUE e outro para o OscarPBS. O mesmo lote de *jobs* sintético é utilizado pelos dois scripts e o mesmo instante de submissão de cada *job* é utilizado pelos scripts. Ou seja, dado um *job* sintético qualquer do lote, ele é submetido no mesmo instante pelos dois scripts para

cada um dos dois escalonadores. A execução simultânea dos dois escalonadores é feita para que fatores externos como diferentes quantidades de carga real no *cluster* pudesse interferir nos resultados. Quando um dos scripts pretende submeter um *job*, no caso do TORQUE, o script assume a identidade do usuário sintético dono daquele *job* e com poderes desse usuário, submete o *job* sintético utilizando o comando qsub do torque. No caso do script do OscarPBS, uma requisição do *job* sintético é criada no banco de dados do OscarPBS, com o respectivo dono do *job*. O escalonador do OscarPBS, se encarrega de submeter o *job* com as permissões do dono do *job*, quando chega a hora de submeter o *job*. Nesses cenários de execução, foram utilizados intervalos de tempo entre a submissão de *jobs* para simular o comportamento de *jobs* submetidos por usuários reais. Inicialmente foram utilizados intervalos de tempo fixo, nos primeiros experimentos. Posteriormente foram utilizados intervalos de tamanho aleatório, gerados a partir de uma distribuição exponencial [26, 44], tendo em vista que essa distribuição pode simular melhor o intervalo entre as chegadas de *job*. Foi realizada uma coleta de intervalos de tempo de submissões reais de *jobs* no *cluster* Oscar, com o objetivo de gerar um gráfico de frequências das ocorrências de intervalos de tempo entre as submissões, onde os intervalos obtidos são aproximados para quantidades inteiras em segundos. Assim é possível verificar o formato do gráfico de frequências e utilizar uma função densidade de probabilidade que tenha um formato parecido para gerar valores de intervalos a partir desse gráfico. A função que representa a frequência com que ocorrem certos valores de intervalos de tempo entre as requisições de *jobs* foi gerada tomando como base um histórico de 100 dias de submissões reais ocorrida no *cluster* Oscar. O eixo horizontal representa o tempo em segundos do tamanho do intervalo e o eixo vertical representa a frequência com que intervalos daquele tamanho ocorreram. Como certos tamanhos de intervalo não ocorrem (têm frequência igual a zero) o gráfico oscila muito (Figura 6.1), então foram gerados alguns gráficos onde médias de frequências em intervalos de tempo foram utilizadas, tendo em vista reduzir as oscilações e suavizar o formato do gráfico.

A Figura 6.2 representa um gráfico de frequências médias em intervalos de 25 segundos. Outros valores de média foram utilizados, gerando gráficos parecidos com o da figura. Média menores geram gráficos menos suaves e médias maiores geram gráficos mais suaves porém alteram mais o formato da curva.

O perfil é semelhante a de uma distribuição exponencial, apesar de não ser possível gerar uma distribuição exponencial parecida com a média dos intervalos de tempo entre as submissões de *jobs* obtida a partir dos dados do histórico, pois nos intervalos obtidos dados do histórico a variância foi muito maior que a média, provavelmente devido a períodos

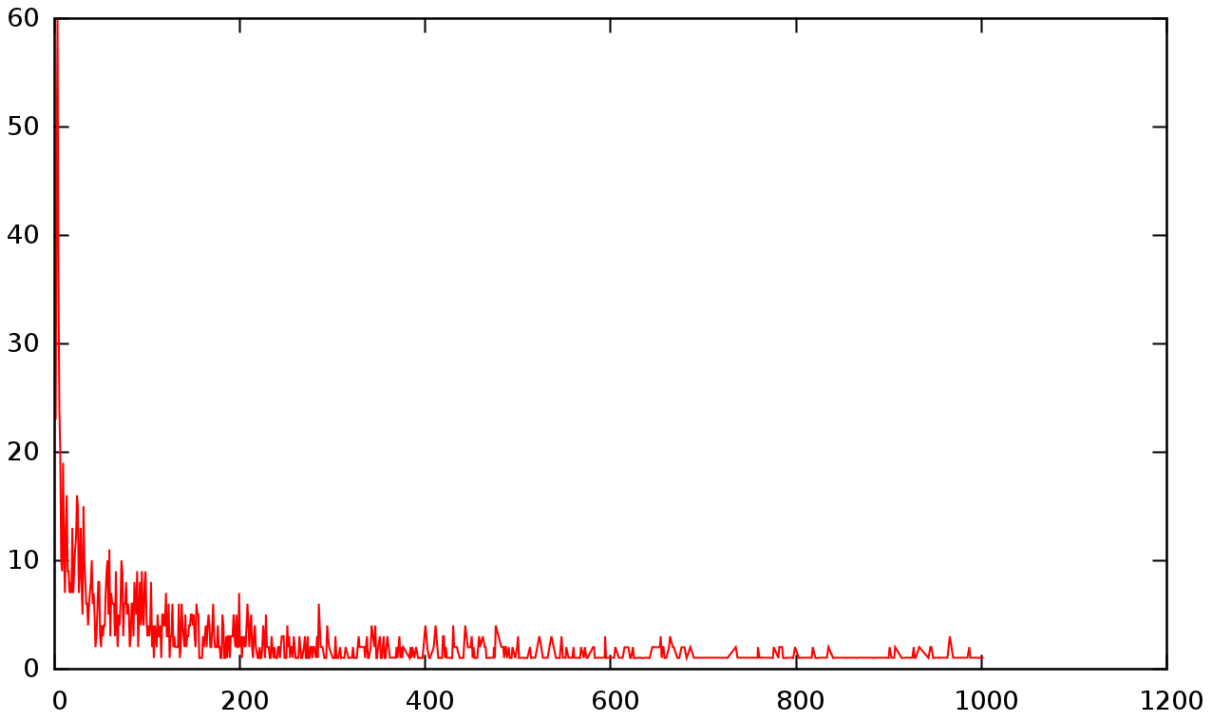

Figura 6.1: Gráfico de frequências de tempos entre submissões obtido a partir do histórico de execução de 100 dias. O eixo horizontal representa o tamanho do intervalo entre submissões e o eixo vertical representa o número de ocorrências de um determinado valor de intervalo.

noturnos onde há pouco submissão de *jobs*.

As Figuras 6.3 e 6.4 representam o gráfico de distribuição exponencial com média de 60 segundos, onde o eixo vertical (segundos) está representado nas escalas linear e logarítmica respectivamente. A distribuição exponencial é utilizada como modelo para a obtenção dos intervalos de tempo entre as submissões.

As figuras 6.5 e 6.6 representam o gráfico da função inversa da função exponencial acumulada [30] de onde os intervalos são obtidos diretamente, onde o eixo vertical representa o tamanho do intervalo em segundos, nas escalas linear e logarítmica respectivamente. Um valor é sorteado dentro de uma distribuição uniforme no intervalo entre 0 e 1 e depois é convertido em um valor intervalo por meio da função inversa da exponencial acumulada.

6.1.3 Métricas escolhidas para a comparação

Alguma métricas como o *Turnaround time* médio, o Tempo de resposta médio e o *Slowdown* médio foram consideradas, sendo escolhidas como métricas de comparação o *Turnaround time* médio e Tempo de resposta médio [32, 31, 41, 46, 27, 34].

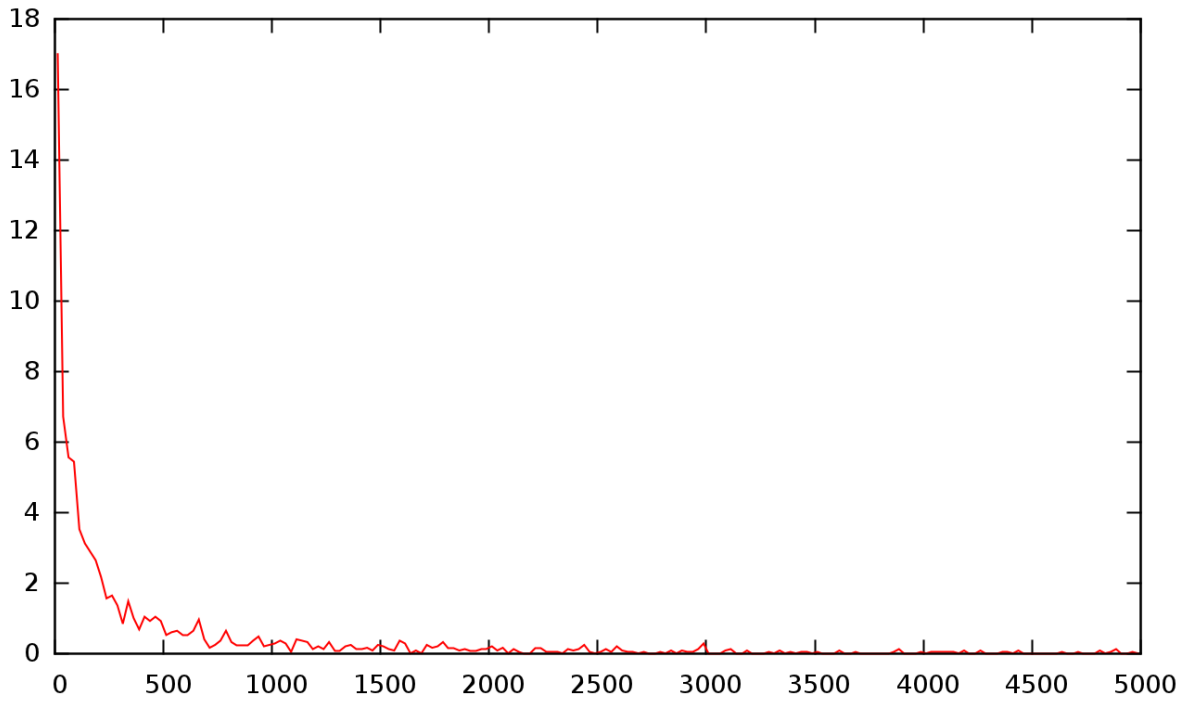

Figura 6.2: Gráfico de frequências de tempos entre submissões obtido a partir do histórico de execução de 100 dias. O eixo horizontal representa o tamanho do intervalo entre submissões e o eixo vertical representa o número de médio de ocorrências de um determinado valor de intervalo. Intervalos de 25 segundos foram utilizados e a frequência média em cada intervalo é utilizada.

Essas métricas são muito utilizadas na avaliação de escalonadores, onde os *jobs* chegam durante o escalonamento. Ambas as métricas refletem o quanto o escalonador consegue utilizar bem o sistema. Um *Turnaround time* médio menor, significa que os *jobs* terminam mais rapidamente em média e um Tempo de resposta médio menor, significa que os *jobs* começam a executar mais rapidamente em média. Tais métricas, quando minimizadas podem trazer maior satisfação aos usuários do *cluster*, que esperam que seus *jobs* comecem a execução o mais cedo possível, ou que terminem o mais cedo possível. O *Turnaround time* médio considera o instante de submissão e o instante de término da execução dos *jobs*, enquanto o tempo de resposta médio considera o instante de submissão e instante de início da execução dos *jobs*.

Para explicar melhor o significado dessas métricas, o *Turnaround time* significa o tempo decorrido, desde o momento da submissão do *job*, até o término de sua execução. O *Turnaround time* médio representa a média aritmética do *Turnaround time* de todos os *jobs* submetidos do lote de *jobs* sintéticos. As fórmulas do *Turnaround time* de um *job* e do *Turnaround time* médio podem ser vistas respectivamente nas Equações (6.1) e (6.2).

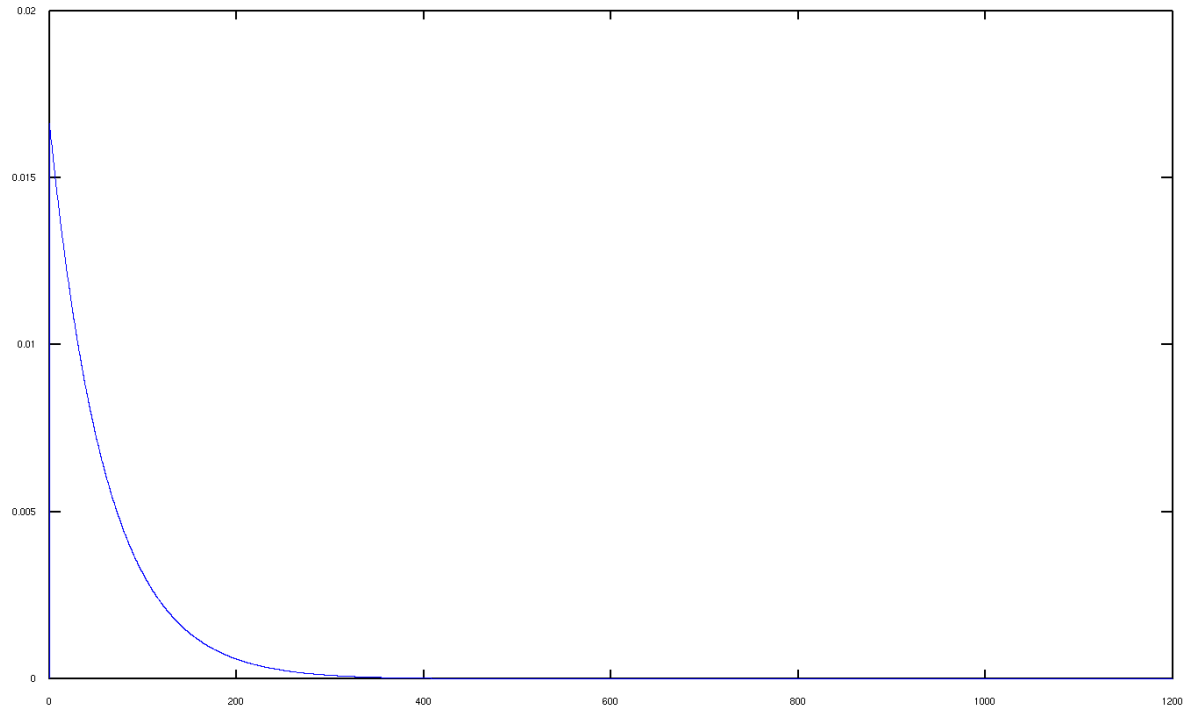

Figura 6.3: Função de distribuição exponencial com média 60 segundos. O eixo horizontal representa tamanho do intervalo em segundos, o eixo vertical representa a probabilidade de um intervalo infinitesimal ocorrer

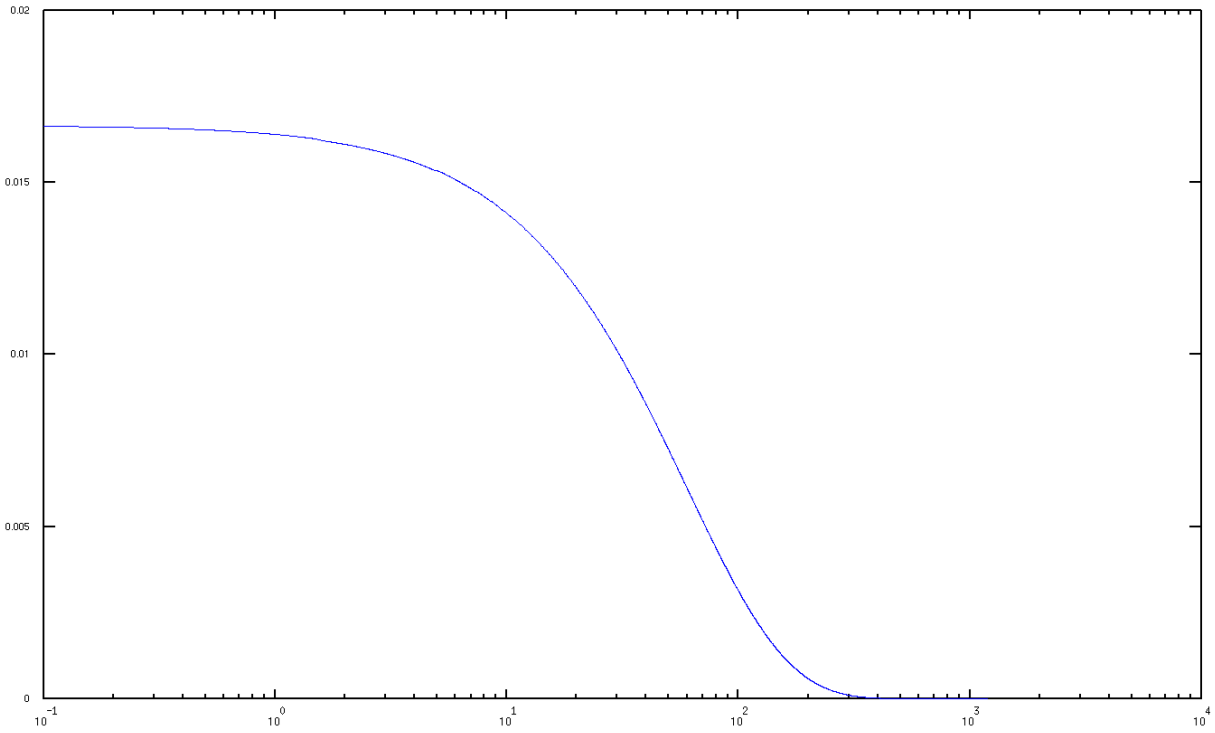

Figura 6.4: Função de distribuição exponencial com média 60 segundos. O eixo horizontal representa tamanho do intervalo em segundos em escala logarítmica, o eixo vertical representa a probabilidade de um intervalo infinitesimal ocorrer

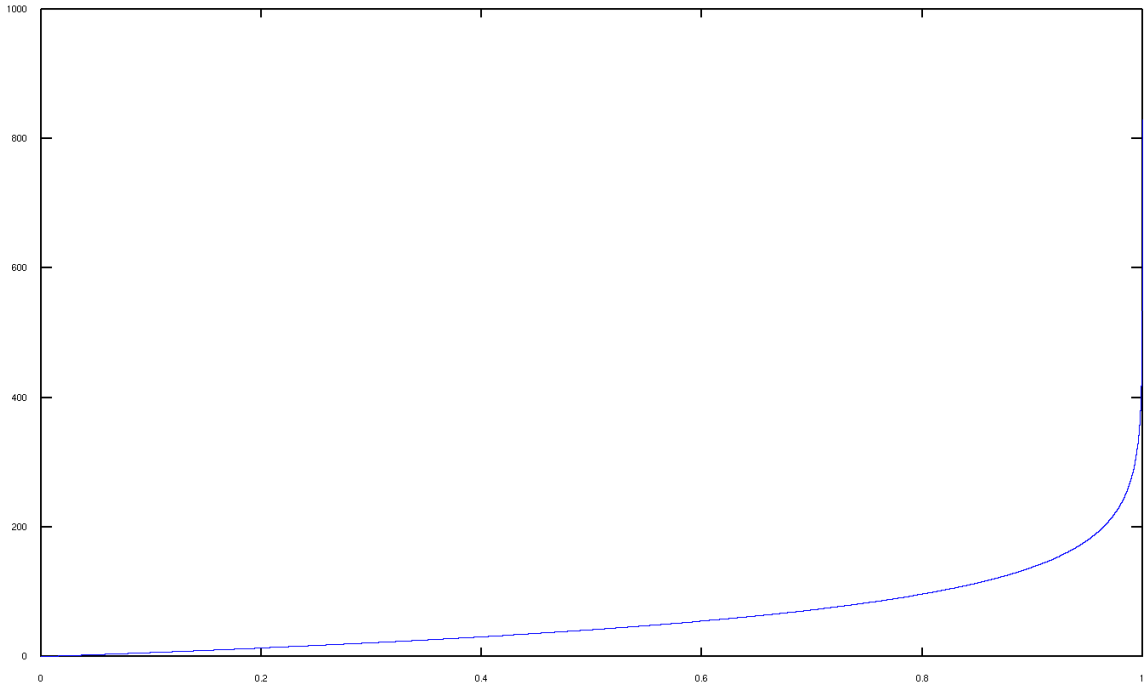

Figura 6.5: Função inversa da distribuição exponencial acumulada com média 60 segundos. O eixo horizontal representa a probabilidade acumulada limitada entre zero e um, o eixo vertical representa o tamanho do intervalo em segundos

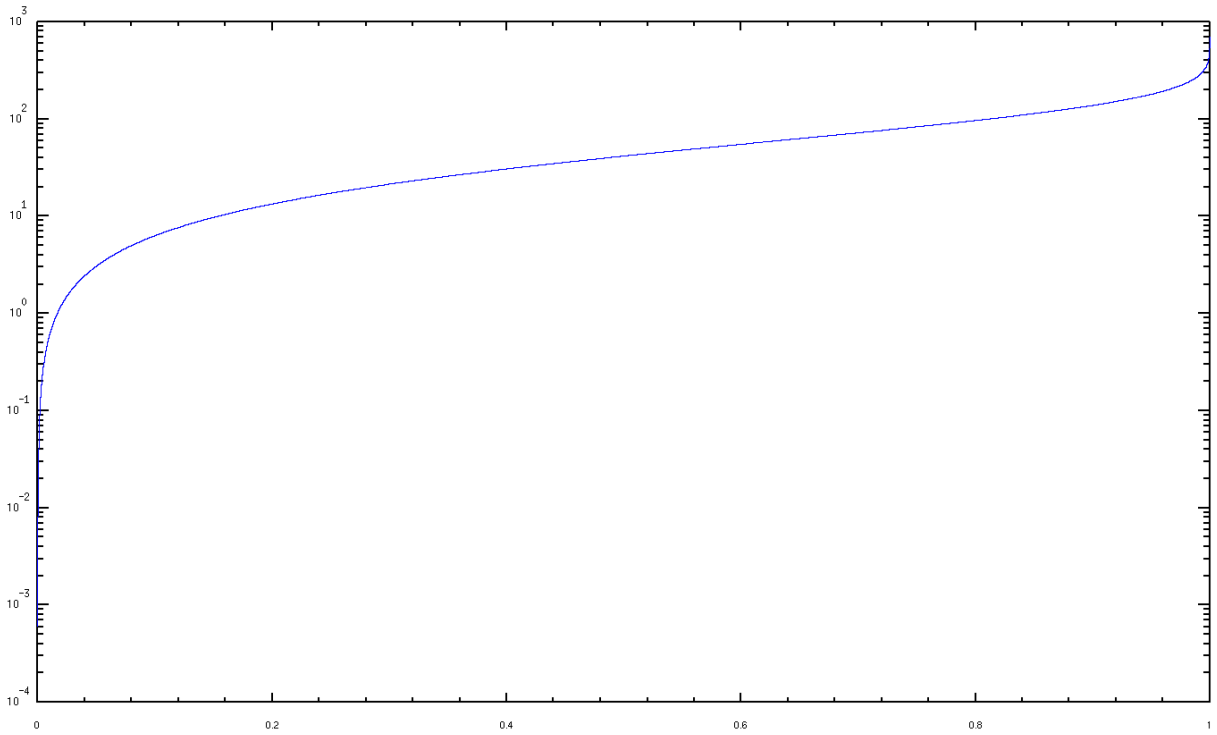

Figura 6.6: Função inversa da distribuição exponencial acumulada com média 60 segundos. O eixo horizontal representa a probabilidade acumulada limitada entre zero e um, o eixo vertical representa o tamanho do intervalo em segundos em escala logarítmica

$$Turnaround\_job_i = Termino\_execucao\_job_i - Submissao\_job_i \quad (6.1)$$

$$Turnaround\_Medio = \frac{\sum_{i=1}^{N} Turnaround\_job_i}{N} \quad (6.2)$$

O N representa o número de *jobs* do lote sintético. O tempo de resposta, também definido como *Response Time* ou *Waiting Time* [27, 34, 48], é dado pelo tempo decorrido entre a submissão do *job* e o início da execução do mesmo, ou seja, do momento que o usuário submeteu o *job*, até o momento em que o escalonador conseguiu iniciar a execução *job*. O tempo de resposta médio é dado pela média aritmética do tempo de resposta de todos os *jobs* sintéticos do lote do experimento. As fórmulas do tempo de resposta de um *job* e do tempo de resposta médio podem ser vistas respectivamente nas equações (6.3) e (6.4).

$$Tempo\_Resposta\_job_i = Termino\_execucao\_job_i - Inicio\_execucao\_job_i \quad (6.3)$$

$$Tempo\_Resposta\_Medio = \frac{\sum_{i=1}^{N} Tempo\_Resposta\_job_i}{N} \quad (6.4)$$

Para realizar as medidas das métricas, foram coletados os *logs* da execução dos *jobs* após os experimentos. Esses *logs* mostram o instante de submissão, de início de execução e de *walltime* utilizado de cada *job*, no caso do experimento realizado com o escalonador padrão do TORQUE. No caso do experimento realizado com o escalonador OscarPBS, o instante de submissão fica registrado no banco de dados e o início da execução e *walltime* utilizados ficam no *log* do serviço *pbs_ server*.

6.1.4 Experimentos Realizados

Ao todo foram realizados oito experimentos, onde em cada um deles o lote de *jobs* sintéticos foi executado pelos escalonadores OscarPBS e o escalonador padrão do TORQUE. Após isso, o histórico gerado por cada um dos experimentos foi verificado por um script que calcula as métricas para cada um dos escalonadores em cada experimento. Em cada experimento, o *Turnaround time* médio e o tempo de resposta médio de cada escalonador são calculados e comparados. No sétimo e oitavo experimentos, uma métrica de taxa de *jobs* bem sucedidos também foi utilizada, pois nesses experimentos existem *jobs* com

prazo.

Cada um dos lote de *jobs* sintéticos utilizados no experimento são baseados no mesmo histórico de *jobs* reais do *cluster* Oscar. Alguns meses de histórico foram coletados continuamente por um script e adicionados no final de um único arquivo. Partes desse arquivo de histórico eram de tempos em tempos salvos em arquivos menores para evitar a perda do histórico caso o arquivo de histórico corrompesse. Um desses arquivos que representavam uma parte do histórico foi escolhido ao acaso, para ser utilizado como modelo para a geração de submissões artificiais utilizadas nos experimentos. O arquivo escolhido possui 318 *jobs* submetidos nos meses de janeiro e fevereiro de 2012. No primeiro experimento, o lote de *jobs* sintéticos foi preparado de forma a que o intervalo de tempo entre a submissão de cada *job* tome exatamente um minuto. Em termos da escala de tempo real, é como se um *job* fosse submetido a cada dez minutos, considerando que nas submissões sintéticas, a duração dos *jobs* é 10 vezes menor que o do *job* real obtido no histórico. Nesse experimento o tratamento de *jobs Starve* é desabilitado, tanto no OscarPBS como no escalonador padrão do TORQUE. Assim, apenas o tratamento de *jobs* Comuns é avaliado nesse primeiro experimento.

O segundo experimento é semelhante ao primeiro, mas o tratamento de *jobs Starve* está habilitado em ambos os escalonadores, considerando que o escalonador padrão do TORQUE tem suporte ao tratamento de *jobs Starve*. O tempo de *starve* utilizado nos experimentos é o equivalente a cinco horas, porém foi dividido por 10 por levar em consideração a escala de tempo utilizada nos experimentos, o que resulta em um tempo de *starve* de 30 minutos.

O terceiro e o quarto experimentos são parecidos com o primeiro e o segundo experimentos respectivamente. A diferença é que nesses experimentos, o intervalo de tempo entre as submissões dos *jobs* não é fixo. Um tempo fixo equivalente a 10 minutos, não é muito realista considerando que isso poderia influenciar no comportamento de ambos os escalonadores comparados, além de que o tempo entre submissões de um cenário real é aleatório. Então se utilizou uma distribuição de probabilidade exponencial com tempo médio de um minuto, para se gerar uma sequência de intervalos de tempo, com média de aproximadamente um minuto. A distribuição exponencial é utilizada para modelar a ocorrência de eventos no tempo, tornando assim os experimentos três e quatro um pouco mais realistas.

Nos quinto e sexto experimentos, foi feito um estudo no histórico de experimentos reais, a fim de se descobrir qual seria um tempo médio entre submissões que fosse mais

realista para se usar na distribuição exponencial. Um histórico de 100 dias seguidos foi então coletado, dos meses de agosto, setembro e outubro de 2012 e o tempo médio entre a submissão desses *jobs* foi calculado. O tempo calculo foi de 3290 segundos, o que dá cerca de 54 minutos entre cada *job*. Ao invés do equivalente a 10 minutos utilizados nos experimentos anteriores. Numa escala de 10 vezes, se descobriu que o tempo médio é de 329 segundos, então esse tempo foi utilizado na distribuição exponencial para criar os intervalos entre submissões do *job*. Posteriormente se viu que a distribuição exponencial com esse tempo não representava muito bem a realidade, pois existem grandes períodos sem submissão de *jobs* (por exemplo, períodos noturno, ou feriados e fins de semana), seguidos de períodos com muitas submissões de *jobs*. A distribuição exponencial distribui a submissão de modo muito uniforme, sem considerar esses picos de submissão, assim o tempo médio real dentro de uma distribuição exponencial acaba por gerar um lote sintético que quase não gera fila de *jobs* candidatos, assim não possibilitando uma comparação significativa entre os dois escalonadores.

O sétimo experimento levou em conta a existência de submissões de *jobs* QoS, Comuns e *Starve*. Neste experimento 20 por cento dos 318 *jobs* do lote, o que, por aproximação, resultou em 64 *jobs*, foram sorteados e submetidos como *jobs* QoS, enquanto os *jobs* restantes foram submetidos como *jobs* Comuns. O prazo dos *jobs* QoS sorteados foi definido em três vezes o *walltime* do *job*, sendo que esse fator de multiplicação foi escolhido ao acaso. A escolha desse fator que definiu o prazo nos experimentos realizados nesse trabalho pode ser refinada em trabalhos futuros, seja por meio de pesquisa em outros trabalhos ou com a realização mais experimentos com diferentes fatores. Nesse experimento se utilizou intervalos de tempo entre as submissões de *jobs* com base numa distribuição exponencial com média de 1 minuto. No TORQUE, os 64 *jobs* QoS são submetidos de modo convencional, por este não suportar *jobs* com prazo de forma semelhantes aos *jobs* QoS do OscarPBS. Nesse experimento o número de *jobs* com prazo que conseguiram finalizar dentro de seu prazo foi considerado.

O oitavo experimento levou em conta a existência de submissões de *jobs* de Emergência, Comuns e *Starve*. Neste experimento 5 por cento dos 318 *jobs* do lote, o que, por aproximação, resultou em 16 *jobs*, foram sorteados e submetidos como *jobs* Emergência, enquanto os *jobs* restantes foram submetidos como *jobs* Comuns. O prazo dos *jobs* Emergência sorteados foi definido em uma vez e meia o *walltime* do *job*. Nesse experimento se utilizou intervalos de tempo entre as submissões de *jobs* com base numa distribuição exponencial com média de 1 minuto. No TORQUE os 16 *jobs* de Emergência são submetidos de modo convencional, por este não suportar *jobs* com prazo de forma semelhantes aos

Nome da Variável	Valor	Pode desalocar *job* do tipo:
sc_aloc	Verdadeiro	*Starve* crítico alocado
q_aloc	Verdadeiro	QoS alocado
e_aloc	Falso	Emergência alocado
c_exec	Verdadeiro	Comum em execução
s_exec	Verdadeiro	*Starve* em execução
q_exec	Verdadeiro	QoS em execução
e_exec	Falso	Emergência em execução

Tabela 6.1: Tabela com os valores das variáveis booleanas de poder para desalocar outros *jobs* atribuídos aos *jobs* de Emergência submetidos no oitavo experimento

jobs QoS do OscarPBS. Nesse experimento o número de *jobs* com prazo que conseguiram finalizar dentro de seu prazo foi considerado. Os poderes atribuídos a todos os *jobs* de Emergência submetidos no experimento podem ser vistos na Tabela 6.1. Todos os *jobs* em execução que foram desalocados por *jobs* Emergência voltavam para fila de Candidatos. Nesse experimento o número de *jobs* com prazo que conseguiram finalizar dentro de seu prazo foi verificado.

6.2 Avaliação de Desempenho

Nessa seção, vamos apresentar resultados de experimentos que comparam o *turnaround* médio dos *jobs*, o tempo de resposta médio dos *jobs* e o número de *jobs* com prazo bem sucedidos, obtidos do escalonador OscarPBS e do escalonador padrão do TORQUE para lotes idênticos de submissões de *jobs* sintéticos. Com base nesses resultados se espera avaliar o desempenho do escalonador OscarPBS frente ao escalonador padrão do TORQUE, levando em consideração cada um dos tipos de *job* propostos nesse trabalho. O escalonador padrão do TORQUE foi escolhido como referencial de comparação devido sua simplicidade em termos de configuração e políticas utilizadas, o que veio a ser importante para viabilizar os experimentos com o OscarPBS.

6.2.1 Jobs Comuns com intervalos de submissões fixos

Experimento com um lote de 318 *jobs* sintéticos, executados nos escalonadores OscarPBS e padrão do torque, onde o tratamento para *jobs Starve* está desabilitado e o intervalo de

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de Turnaround Médio (s)	10174,52	15689,73	1,54
Tempo de Resposta Médio (s)	6763,18	12280,06	1,82

Tabela 6.2: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs* Comuns e tempo fixo de 60 segundos entre as requisições.

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de Turnaround Médio (s)	13909,81	19543,12	1,40
Tempo de Resposta Médio (s)	10500,43	16133,28	1,54

Tabela 6.3: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs Starve* e tempo fixo de 60 segundos entre as requisições.

tempo entre as submissões é de 60 segundos.

A Tabela 6.2 mostra os resultados para as métricas, bem como mostra na terceira coluna, para indicar quanto o OpenPBS foi mais rápido ou mais lento em relação ao escalonador padrão, representado pelo tempo do escalonador padrão dividido pelo tempo de OscarPBS, em cada métrica.

O escalonador OscarPBS mostrou melhor resultado que o escalonador padrão do torque, apresentando um ganho de 1,54 no *Turnaround* médio e de 1,82 no tempo de resposta médio. Tal ganho de desempenho em ambas as métricas deve ter sido alcançado devido a utilização do método *Empurrar*, uma vez que esse método permite o OscarPBS obtenha melhor encaixe de *jobs* nas máquinas do *cluster* antes de colocar os mesmos em execução.

6.2.2 Jobs Starve com intervalos de submissões fixos

Experimento com um lote de 318 *jobs* sintéticos, executados nos escalonadores OscarPBS e padrão do torque, onde o tratamento para *jobs Starve* está habilitado, com um tempo de *Starve* de 30 minutos, o que equivale a 5 horas quando comparado a escala de tempo real 10 vezes maior e o intervalo de tempo entre as submissões é de 60 segundos.

A Tabela 6.3 mostra os resultados para as métricas e indica, quanto o OpenPBS foi mais eficiente em relação ao escalonador padrão, representado pelo tempo do escalonador padrão dividido pelo tempo de OscarPBS, em cada métrica.

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de Turnaround Médio (s)	10559,08	15108,98	1,43
Tempo de Resposta Médio (s)	7149,07	11698,60	1,64

Tabela 6.4: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs* Comuns e intervalos de tempo entre as requisições obtidos a partir de uma distribuição exponencial com média de 60 segundos.

O escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,40 no *Turnaround* médio e de 1,54 no tempo de resposta médio.

6.2.3 *Jobs* Comuns com intervalos de submissões exponencial

Experimento com média exponencial de 60 segundos

Experimento com um lote de 318 *jobs* sintéticos, executados nos escalonadores OscarPBS e padrão do torque, onde o tratamento para *jobs Starve* está desabilitado e o intervalo de tempo entre as submissões foi gerado a partir de uma distribuição de probabilidade exponencial com média de 60 segundos.

Como a Tabela 6.4 mostra, o escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,43 no *Turnaround* médio e de 1,64 no tempo de resposta médio.

Experimento com média exponencial de 329 segundos

Experimento com um lote de 318 *jobs* sintéticos, executados nos escalonadores OscarPBS e padrão do torque, onde o tratamento para *jobs Starve* está desabilitado e o intervalo de tempo entre as submissões é gerado a partir de uma distribuição exponencial com média de 329 segundos. Esse tempo médio de intervalo entre submissões foi obtido a partir de uma amostra de 100 dias do histórico de submissões real do Oscar.

Como apresentado na Tabela 6.5, o escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,01 no *Turnaround* médio e de 1,01 No tempo de resposta médio. O ganho foi quase desprezível neste caso, devido a distribuição exponencial espalhar os *jobs* de maneira muito uniforme com relação a períodos em que não há submissões, como durante a noite. Com os *jobs* mais espaçados, pouca fila se forma e quase todos os *jobs* podem ser atendidos no momento em que

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de *Turnaround* Médio (s)	43324,24	43654,54	1,01
Tempo de Resposta Médio (s)	39914,78	40246,32	1,01

Tabela 6.5: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs* Comuns e intervalos de tempo entre as requisições obtidos a partir de uma distribuição exponencial com média de 329 segundos. Os ganhos são desprezíveis nesse experimento provavelmente porque como intervalo médio de tempo entre submissões de *jobs* é maior, poucos *jobs* concorrem na fila de espera e o método *Empurrar* é pouco utilizado.

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de Turnaround Médio (s)	13957,94	19283,97	1,38
Tempo de Resposta Médio (s)	10547,91	15873,59	1,50

Tabela 6.6: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs Starve* e intervalos de tempo entre as requisições obtidos a partir de uma distribuição exponencial com média de 60 segundos.

são submetidos, fazendo com que haja pouca diferença entre como os dois escalonadores decidem alocar os *jobs*.

6.2.4 Jobs Starve com intervalos de submissões exponencial

Experimento com média exponencial de 60 segundos

Nesse experimento para *jobs Starve* com intervalos exponencial foi utilizado um lote de 318 *jobs* sintéticos, executados nos escalonadores OscarPBS e padrão do torque, onde o tratamento para *jobs Starve* está habilitado, com tempo de *Starve* configurado para 30 minutos, ou 5 horas considerando o tempo equivalente ao de submissões reais 10 vezes mais lento e o intervalo de tempo entre as submissões é gerado a partir de uma distribuição exponencial com média de 60 segundos.

Segundo a Tabela 6.6, o escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,38 no *Turnaround* médio e de 1,50 no tempo de resposta médio.

Experimento com média exponencial de 329 segundos

Experimento com um lote de 318 *jobs* sintéticos, executados nos escalonadores OscarPBS e padrão do torque, onde o tratamento para *jobs* starve está habilitado com tempo

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de Turnaround Médio (s)	43779,72	44856,31	1,02
Tempo de Resposta Médio (s)	40371,81	41447,32	1,03

Tabela 6.7: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs Starve* e intervalos de tempo entre as requisições obtidos a partir de uma distribuição exponencial com média de 329 segundos.

de *starve* de 30 minutos e o intervalo de tempo entre as submissões é gerado a partir de uma distribuição exponencial com média de 329 segundos.

Na Tabela 6.7, o escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,02 no *Turnaround* médio e de 1,03 no tempo de resposta médio. Como no caso do experimento 5 houve pouco ganho do OscarPBS em relação ao escalonador padrão do TORQUE, devido a distribuição exponencial não representar os períodos noturnos, de feriado e fim de semana, em que há menos submissão de *jobs*.

6.2.5 Jobs Comuns, Starve e QoS com intervalos de submissões exponencial

Esse experimento visa avaliar o desempenho do OscarPBS comparado ao TORQUE padrão quando são levados em conta *jobs* com prazo. Para isso são utilizados os *jobs* QoS do OscarPBS na submissão de *jobs* com prazo, enquanto no TORQUE os mesmos *jobs* são submetidos sem o prazo. Os *jobs* com prazo tem o prazo definido como sendo 3 vezes maior que seu *walltime*. O lote de 318 *jobs* sintéticos foi utilizado e os mesmos *jobs* foram executados executados nos escalonadores OscarPBS e pelo TORQUE padrão, levando-se em consideração que no caso do OscarPBS, os *jobs* com prazo tem seu tipo configurado como sendo um *job* QoS e tem o prazo definido. Em ambos os escalonadores o tratamento para *jobs Starve* está habilitado, com tempo de *starve* configurado para 30 minutos, ou 5 horas considerando o tempo equivalente ao de submissões reais 10 vezes mais lento e o intervalo de tempo entre as submissões é gerado a partir de uma distribuição exponencial com média de 60 segundos.

Como visto na Tabela 6.8, o escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,86 no *Turnaround* médio e de 2,07 no tempo de resposta médio.

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de *Turnaround* Médio (s)	19976,74	37206,10	1,86
Tempo de Resposta Médio (s)	16329,59	33795,91	2,07
Jobs QoS dentro do prazo	40	13	3,08

Tabela 6.8: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs* Comuns, *Starve* e QoS, com tempo exponencial com média de 60 segundos entre as requisições.

Quanto ao número de *jobs* que conseguiram ser executados dentro de seu prazo, o OscarPBS conseguiu satisfazer o prazo para 40 dos 64 *jobs* QoS, enquanto o escalonador padrão do TORQUE terminou 13 dos 64 *jobs* QoS dentro do prazo. O ganho do OscarPBS sobre o TORQUE com relação ao número de *jobs* QoS terminados no prazo foi de 3,08.

6.2.6 Jobs Comuns, Starve e Emergência com intervalos de submissões exponencial

Esse experimento visa avaliar o desempenho do OscarPBS comparado ao TORQUE padrão quando são levados em conta *jobs* com prazo e de caráter emergencial. Para isso são utilizados os *jobs* de Emergência do OscarPBS na submissão de *jobs* com prazo, enquanto no TORQUE os mesmos *jobs* são submetidos sem o prazo. Os *jobs* com prazo tem o prazo definido como sendo uma vez e meia maior que seu *walltime*. O lote de 318 *jobs* sintéticos foi utilizado e os mesmos *jobs* foram executados nos escalonadores OscarPBS e no TORQUE padrão, levando em consideração que no caso do OscarPBS, os *jobs* Emergência tiveram seu tipo, poderes para desalocar *jobs* e prazo definidos na requisição. Em ambos os escalonadores o tratamento para *jobs Starve* está habilitado, com tempo de *starve* configurado para 30 minutos, ou 5 horas considerando o tempo equivalente ao de submissões reais 10 vezes mais lento e o intervalo de tempo entre as submissões é gerado a partir de uma distribuição exponencial com média de 60 segundos.

Na Tabela 6.9, o escalonador OscarPBS mostrou melhor resultado do que o escalonador padrão do torque, apresentando um ganho de 1,41 no *Turnaround* médio e de 1,48 no tempo de resposta médio. Quanto ao número de *jobs* que conseguiram ser executados dentro de seu prazo, o OscarPBS conseguiu satisfazer o prazo para todos os 16 *jobs* de Emergência submetidos, enquanto o escalonador padrão do TORQUE terminou 4 dos 16 *jobs* de Emergência dentro do prazo. O ganho do OscarPBS sobre o TORQUE com relação ao número de *jobs* Emergência terminados no prazo foi de 4,00.

Métrica \Escalonador	OscarPBS	Torque Padrão	Ganho do OscarPBS
Tempo de Turnaround Médio (s)	26452,30	37214,81	1,41
Tempo de Resposta Médio (s)	23042,73	33804,45	1,48
Jobs Emergência dentro do prazo	16	4	4,00

Tabela 6.9: Comparação dos resultados das métricas do OscarPBS com o escalonador do Torque, no experimento com *jobs* Comuns, *Starve* e Emergência, com tempo exponencial com média de 60 segundos entre as requisições.

Capítulo 7

Conclusões

7.1 Considerações Finais

No decorrer desse trabalho verificou-se que é possível melhorar o desempenho ao adicionarem-se funcionalidades em ferramentas bastante conhecidas destinadas as submissões e ao controle de *jobs* em *clusters*. Através do escalonador OscarPBS sobre o *D-RMS* TORQUE foi possível obter um melhor desempenho do escalonamento, uma das metas deste estudo. Constatou-se que a melhora do desempenho manteve-se mesmo com o uso de técnicas que eventualmente tornam o escalonamento mais restrito, como o tratamento de *jobs Starve*. Além disso, verificou-se que as novas funcionalidades de prazo para *jobs* QoS e de Emergência obtiveram exito ao fazerem que uma taxa maior de *jobs* com prazo fossem executados pelo OscarPBS, do que através do escalonador padrão. Mesmo com estas funcionalidades de prazo priorizando *jobs* QoS ou de Emergência, constatou-se uma melhora no desempenho global do escalonamento quanto ao tempo de resposta médio de todos os *jobs*. Concluiu-se que o escalonador desenvolvido nesse trabalho otimizou as políticas de usuário, além de de melhorar o desempenho do sistema.

7.2 Trabalhos Futuros

Com relação a trabalhos futuros, a curto prazo, pode-se pensar em mudanças simples a serem feitas no OscarPBS, como por exemplo no método *Empurrar*, que poderia permitir que os fragmentos empurrados também empurrem outros fragmentos, ou seja, permitir que o *Empurrar* possua mais de um nível. Possíveis novos experimentos baseados em outras mudanças no algoritmo do escalonador à serem implementados, como por exemplo a mudança na ordem em que fragmentos de *job* são selecionados durante a alocação de *jobs*.

Experimentos envolvendo novos históricos de *jobs* reais também viáveis, considerando-se também que nesses, outras formas mais precisas de gerarem-se os intervalos entre as submissões podem ser utilizadas. Outras métricas para avaliação de desempenho como o *Slowdown* médio de *jobs* poderiam ser utilizadas. Outro trabalho interessante a curto prazo seria a implementação de um sistema de créditos para os usuários com QoS, como foi desenvolvido no LibraPBS. A longo prazo, pode-se avaliar a possibilidade de adicionar outras funcionalidades ao OscarPBS, ou mesmo de estudar-se outros escalonadores e *D-RMS* de código aberto para incluir neles o método *Empurrar*. Por exemplo, uma nova funcionalidade possível seria acrescentar ao OscarPBS o suporte a alocação de considerando requisições por GPUs. Outra possibilidade é adicionar e aperfeiçoar funcionalidades presentes nos escalonadores MAUI e o MOAB e de se modificar o próprio MAUI para incluir nele o método *Empurrar* já que ele possui código aberto. Outro Gerenciador de Recursos Distribuídos e escalonador de código aberto, o SLURM, também poderia ser estudado de maneira mais detalhada, visando a inclusão do método *Empurrar* e de aproveitar-se funcionalidades dele para implementação no OscarPBS. A criação de uma interface web para gerenciar e submeter , semelhante às utilizadas em ferramentas como o PBSWeb e o MOAB, que permitam a utilização do *cluster* por usuários menos experientes seria de grande valia em trabalhos futuros.

Referências

[1] Adaptive Computing. Moab Workload Manager Administrator Guide. Disponível em:<http://www.adaptivecomputing.com/resources/docs/>. Acesso em 19 de abril de 2013.

[2] Adaptive Computing. TORQUE Resource Manager Administrator Guide. Disponível em:<http://www.adaptivecomputing.com/resources/docs/>. Acesso em 19 de abril de 2013.

[3] Adaptive Computing. Maui Administrator's Guide. Disponível em:<http://www.adaptivecomputing.com/resources/docs/>. Acesso em 19 de abril de 2013.

[4] Adaptive Computing. The torqueusers Archives. Disponível em:<http://www.supercluster.org/pipermail/torqueusers/>. Acesso em 01 de maio de 2013.

[5] Adaptive Computing. Company History. Disponível em:<http://www.adaptivecomputing.com/company/company-history/>. Acesso em 19 de abril de 2013.

[6] Altair. History. Disponível em:<http://www.altair.com/History.aspx>. Acesso em 19 de abril de 2013.

[7] Dror Feitelson. Parallel Workloads Archive. Disponível em:<http://www.cs.huji.ac.il/labs/parallel/workload/>. Acesso em 04 de maio de 2013.

[8] Generic NQS Project. Generic NQS. Disponível em:<http://gnqs.sourceforge.net/docs/papers/gnqs_papers/gnqs0005.htm>. Acesso em 01 de maio de 2013.

[9] Globus Alliance. Globus Toolkit Homepage. Disponível em:<http://www.globus.org/toolkit>. Acesso em 19 de abril de 2013.

[10] Gompute and Scalable Logic. Open Grid Scheduler/Grid Engine. Disponível em:<http://gridscheduler.sourceforge.net>. Acesso em 19 de abril de 2013.

[11] Instituto de Computação UFF. Portal ST - Cluster Oscar - Passo a passo sobre como usar. Disponível em:<http://suporte.ic.uff.br/index.php/servicos/posgrad/oscarusar>. Acesso em 19 de abril de 2013.

[12] James Patton Jones, editor. PBS Pro 5.3 User Guide. Altair Grid Technologies, 2003.

[13] National Science Foundation. Virtual Grid Execution System (vgES) Design. Disponível em:<http://vgrads.rice.edu/research/execution_system/vges-overview/>. Acesso em 19 de abril de 2013.

[14] Oracle Grid Engine. Disponível em:<http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html>. Acesso em 19 de abril de 2013.

[15] Oracle Grid Engine Installation and Upgrade Guide Release 6.2 Update 7. Disponível em:<http://docs.oracle.com>. Acesso em 25 de abril de 2013.

[16] PBS Works. PBS Professional 11.1 Administrator's Guide, 2011.

[17] PBSWeb: A Web-based Interface to the Portable Batch System. Disponível em:<http://webdocs.cs.ualberta.ca/~pinchak/PBSWeb/>. Acesso em 24 de abril de 2013.

[18] Python Software Foundation. Python Programming Language - Official Website. Disponível em:<http://www.python.org>. Acesso em 19 de abril de 2013.

[19] Tcl Core Team. Tcl Developer Xchange. Disponível em:<http://www.tcl.tk>. Acesso em 19 de abril de 2013.

[20] The PostgreSQL Global Development Group. History. Disponível em:<http://www.postgresql.org/about/history/>. Acesso em 19 de abril de 2013.

[21] University of California. Future home of Catalina Software Distribution Page. Disponível em:<http://www.sdsc.edu/catalina>. Acesso em 19 de abril de 2013.

[22] Akers, W.; Watson, C.; Chen, J.; Chen, Y. The underlord scheduler. a drop-in scheduler for the portable batch system. Disponível em:<http://www.jlab.org/hpc/UnderLord/index.html>. Acesso em 19 de abril de 2013.

[23] Bayucan, A.; Henderson, R. L.; Jones, J. P.; Lesiak, C.; Mann, B.; Nitzberg, B.; Proett, T.; Utley, J. PBS Pro 5.0 Administrator Guide. October 2000.

[24] Bayucan, A.; Henderson, R. L.; Jones, J. P.; Lesiak, C.; Mann, B.; Nitzberg, B.; Proett, T.; Utley, J. Portable Batch System OpenPBS Release 2.3 Administrator Guide. August 2000.

[25] Bayucan, A.; Henderson, R. L.; Proett, T.; Tweten, D. Portable Batch System Project Plan. Numerical Aerospace Simulation Systems Division NASA Ames Research Center, august 1998.

[26] COOPER, J. C. B. The poisson and exponential distributions. In *Applied Probability Trust* (2005), p. 123.

[27] Deng, X.; Gu, N.; Brecht, T.; Lu, K. Preemptive scheduling of parallel jobs on multiprocessors. In *SIAM Journal on Computing (SICOMP) vol. 30* (June 2000), pp. 145–160.

[28] Dimitriadou, S.; Karatza, H. Job scheduling in a distributed system using backfilling with inaccurate runtime computations. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems* (Washington, DC, USA, 2010), IEEE Computer Society, pp. 329–336.

[29] Du, J.; Leung, J. Y.-T. Complexity of scheduling parallel task systems, 1989. pp. 473–487.

[30] Dupree, S. A.; Fraley, S. K. In *A Monte Carlo Primer: A Practical Approach to Radiation Transport* (2002).

[31] Feitelson, D. G. Metrics for parallel job scheduling and their convergence. In *Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP)* (Cambridge, U.S.A., June 2001), pp. 188–205.

[32] Feitelson, D. G.; Rudolph, L. Metrics and benchmarking for parallel job scheduling. In *Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP)* (Orlando, Florida, U.S.A., March 1998), pp. 1–24.

[33] Foster, I.; Kesselman, C.; Lee, C.; Lindell, B.; Nahrstedt, K.; Roy, A. A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on* (1999), pp. 27–36.

[34] HE, Y.; HSU, W. J.; LEISERSON, C. E. Provably efficient adaptive scheduling for parallel jobs. In *Greedy Algorithms* (November 2008), pp. 439–460.

[35] Henderson, R. L. Job scheduling under the portable batch system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* (1995), pp. 279–294.

[36] IBM. LoadLeveler Version 5 Release 1 Resource Manager. 2012.

[37] IBM. Platform Computing Solutions. december 2012.

[38] Islam, M.; Balaji, P.; Sadayappan, P.; Panda, D. K. QoPS: A QoS based scheme for parallel job scheduling. In *Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP)* (Columbus, Ohio, U.S.A., November 2003), pp. 252–268.

[39] Jackson, D.; Snell, Q.; Clement, M. Core algorithms of the maui scheduler. In *Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP)* (Brigham Young University, Provo, Utah, July 2001), pp. 87–102.

[40] Jette, M.; Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *ClusterWorld Conference and Expo* (Lawrence Livermore National Laboratory, California, U.S.A., June 2003).

[41] Kettimuthu, R.; Subramani, V.; Srinivasan, S.; Gopalsamy, T. B.; Panda, D.; Sadayappan, P. An evaluation of preemption strategies for parallel job scheduling. The Ohio State University, november 2001.

[42] Kingsbury, B. A. The Network Queueing System (NQS). Disponível em:<`http://gnqs.sourceforge.net/docs/papers/mnqs_papers/original_cosmic_nqs_paper.htm`>. Acesso em 19 de abril de 2013.

[43] Klusáček, D.; Rudová, H. Performance and fairness for users in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing* (Berlin, 2013), pp. 235–252.

[44] M., P.; J., R. An application of the least flexible job first rule in scheduling and queueing. In *Stern School of Business* (New York University, New York, NY, U.S.A., April 2013).

[45] Ma, G.; Lu, P. Pbsweb:a web-based interface to the portable batch system. In *12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)* (Las Vegas, Nevada, U.S.A., November 2000).

[46] Mudzagada, A.; Mapako, B.; Nyambo, B. M. Perfomance evaluation of resource scheduling techniques in cluster computing. In *International Journal of Scientific & Engineering Research* (May 2012).

[47] Nakada, H.; Takefusa, A.; Ookubo, K.; Kudoh, T.; Tanaka, Y.; Sekiguchi, S. An advance reservation-based computation resource manager for global scheduling. In *National Institute of Advanced Industrial Science and Technology (AIST)* (Sotokanda, Chiyoda-ku, Tokyo, Japan, November 2007).

[48] Niu, S.; Zhai, J.; Ma, X.; Liu, M.; Zhai, Y.; Chen, W.; Zheng, W. Employing checkpoint to improve job scheduling in large-scale systems. In *Job Scheduling Strategies for Parallel Processing* (May 2012), pp. 36–55.

[49] Oracle. Sun ONE Grid Engine and Sun ONE Grid Engine, Enterprise Edition Reference Manual. Disponível em:<`http://docs.oracle.com`>. Acesso em 25 de abril de 2013.

[50] Ramakrishnan, L.; Koelbel, C.; Kee, Y.-S.; Wolski, R.; Nurmi, D.; Gannon, D.; Obertelli, G.; YarKhan, A.; Mandal, A.; Huang, T. M.; Thyagaraja, K.; Zagorodnov, D. Vgrads: enabling e-science workflows on grids and clouds with fault tolerance. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009).

[51] Sherwani, J.; Ali, N.; Lotia, N.; Hayat, Z.; Buyya, R. Libra: An economy-driven job scheduling system for clusters. In *Softw.,Pract. Exper.* (2004), pp. 573–590.

[52] Srinivasan, S.; Kettimuthu, R.; Subramani, V.; Sadayappan, P. Selective reservation strategies for backfill job scheduling. In *Scheduling Strategies for Parallel Processing, LNCS 2357* (2002), Springer-Verlag, pp. 55–71.

[53] Yan, Y.; Chapman, B. Comparative study of distributed resource management systems – sge, lsf, pbs pro, and loadleveler. Department of Computer Science, University of Houston, may 2005.

APÊNDICE A – Instalação do OscarPBS

Este apêndice descreve a instalação do OscarPBS versão 1.0, que é a versão desenvolvida e utilizada nesse trabalho. Inicialmente será feita uma descrição do ambiente necessário incluindo o hardware, Sistema Operacional e softwares utilizados. Será descrita passo a passo a instalação da versão do TORQUE que é utilizado como ponto de partida para a instalação do OscarPBS. Depois será explicado como o TORQUE deve ser modificado para incluir o OscarPBS, o que requer a substituição de alguns arquivos e a modificação de alguns parâmetros, de acordo com o ambiente utilizado. Por último serão descritas as limitações presentes na versão 1.0 do OscarPBS.

A.1 Pré-requisitos

A.1.1 Hardware do *cluster* e Sistema Operacional

Hardware: O hardware recomendado para instalação do OscarPBS é equivalente a um *cluster* de máquinas *multicore* com processadores de 64 bits baseados na arquitetura x86-64 ou AMD64. O *cluster* pode ser composto de duas ou mais máquinas com esse tipo de processador. As máquinas devem estar conectadas entre si, por meio de uma rede local, possivelmente utilizando um *switch ethernet*. É desejável que todas as máquinas possuam armazenamento local, com o uso de um disco rígido em cada máquina. Uma das máquinas deve ser destinada a ser o servidor e essa máquina pode ter mais espaço de armazenamento que as demais, podendo ser esse armazenamento baseado em um *data storage*. O servidor também deve possuir pelo menos duas placas de rede, uma vez que ele deve estar conectado a rede externa e a rede local do *cluster*, onde o servidor é a única máquina do *cluster* que faz a ponte entre a rede externa e as máquinas do *cluster*.

Sistema Operacional: O sistema operacional recomendado para todas as máquinas do *cluster* é o Linux, para sistemas baseados em AMD64, preferencialmente sendo uma

distribuição baseada em Red-Hat Enterprise Linux 5 como por exemplo a distribuição CentOS 5.3 de 64 bits.

A.1.2 Configuração e software do *cluster*

- **Configuração da rede**: As máquinas do *cluster* precisam estar conectadas entre si em uma rede *ethernet*. Elas devem estar configuradas com endereços *IPs* e máscara de uma rede local. A máquina servidor do *cluster* deve possuir duas interfaces de rede, uma delas configurada com um endereço *IP* da rede do *cluster* e outra com um endereço *IP* da rede externa. O endereço de *IP* interno da máquina servidor do *cluster* deve servir de endereço de *gateway* para todas as máquinas do *cluster*. Todas as máquinas do *cluster* devem ter um *hostname* e elas devem ser capazes de reconhecer todas as outras máquinas do *cluster* pelo nome de rede. Em outras palavras, todas as máquinas do *cluster* devem ser capazes de traduzir um *hostname* de qualquer máquina do *cluster* no endereço *IP* daquela máquina.

- **Serviços NIS e NFS**: É recomendável que o servidor do *cluster* funcione como servidor NIS e NFS das máquinas do *cluster*. O NIS deve conter todas as contas dos usuários do *cluster* e essas contas devem ser replicadas para as demais máquinas do *cluster*, o que significa que um usuário existente no servidor do *cluster* pode logar em qualquer máquina do *cluster*. O servidor do *cluster* também deve oferecer um serviço NFS e compartilhar um diretório */home* para todas as máquinas do *cluster*. É desejável que a partição de disco que contém a */home* possua espaço para conter todos os diretórios home dos usuários do *cluster*. Assim, todos os usuários que existem no servidor do *cluster* devem conseguir logar em máquinas do *cluster*, e ter o mesmo diretório *home* compartilhado no *cluster*, comum a todas as máquinas.

- **Serviço *ssh***: É recomendável que todas as máquinas do *cluster* possuam um serviço *ssh*. Um usuário do *cluster* deve ser capaz de se conectar ao servidor do *cluster*, a partir da rede externa, utilizando um software cliente *ssh*. Uma vez logado no servidor do *cluster*, o usuário deve ser capaz de se conectar em qualquer máquina do *cluster* e a partir dessa máquina, deve ser capaz de logar em outra máquina do *cluster*, incluindo o servidor. Uma vez que um usuário esteja logado ao servidor do *cluster* por meio de *ssh* utilizando a senha da conta, é recomendável que todas as conexões *ssh* subsequentes para outras máquinas do *cluster* possam ser realizadas sem a requisição da senha do usuário.

- **Servidor de banco de dados PostgreSQL**: É necessário um servidor de banco de dados instalado na máquina servidor do *cluster*, para armazenar o banco de dados do OscarPBS. O software recomendado é o PostgreSQL 8.1.11

- **Interpretador Python**: O escalonador OscarPBS depende de um interpretador Python instalado no servidor do *cluster* para funcionar. A versão de Python recomendada é a 3.3.0 ou superior.

- **O *D-RMS* TORQUE**: O OscarPBS funciona em conjunto com o TORQUE. A versão recomendada de TORQUE é a 3.0.2 que foi a última versão estável do TORQUE quando o mesmo foi obtido para esse trabalho. Um passo a passo detalhado da instalação do TORQUE é dada a seguir.

A.1.3 Instalação do TORQUE

Uma instalação do TORQUE, inicialmente sem o OscarPBS será descrita nessa subseção. A instalação que inclui o OscarPBS requer algumas mudanças em arquivos, sendo essas explicadas na seção seguinte.

O TORQUE é compilado no servidor do *cluster*. Uma vez compilado no servidor, ele pode ser copiado para as outras máquinas do *cluster*. A versão que fica no servidor e as que ficam em máquinas de trabalho requerem modificações em arquivos responsáveis por iniciar os serviços *pbs_server* e *pbs_sched* no servidor e *pbs_mom* nas máquinas de cálculo, como pode ser visto no passo a passo a seguir:

- Passo 1: Logar-se no servidor do *cluster*.
- Passo 2: Tornar-se *root* com o comando:
  su -
- Passo 3: Ir para o diretório *home* do usuário root:
  cd /root
- Passo 4: Criar um diretório onde o TORQUE será compilado:
  mkdir torque_compilar
- Passo 5: Entrar no diretório de compilação:
  cd torque_compilar

- Passo 6: Obter o arquivo torque-3.0.2.tar.gz do TORQUE no link :
  http://www.adaptivecomputing.com/downloading/?file=/torque/torque-3.0.2.tar.gz
  e copiá-lo para o diretório torque_compilar, ou então obter o arquivo diretamente com o comando:
  wget http://pkgs.fedoraproject.org/repo/pkgs/torque/torque-3.0.2.tar.gz/a2f008c7eeffcb3a0721fbb252397936/torque-3.0.2.tar.gz

- Passo 7: Conferir o *md5sum* do arquivo baixado:
  md5sum torque-3.0.2.tar.gz
  O md5sum deve retornar: *a2f008c7eeffcb3a0721fbb252397936 torque-3.0.2.tar.gz*

- Passo 8: Descompactar o arquivo:
  tar -xvzf torque-3.0.2.tar.gz

- Passo 9: Entrar no diretório gerado:
  cd torque-3.0.2

- Passo 10: Criar os diretórios onde o TORQUE será instalado:
  mkdir /torque ; mkdir /torque/torque_bin ; mkdir /torque/torque_spool

- Passo 11: Executar o script *configure*, indicando onde o TORQUE será instalado e qual o diretório de configuração:
  ./configure –prefix=/torque/torque_bin –with-server-home=/torque/torque_spool

- Passo 12: Compilar o TORQUE:
  make

- Passo 13: Copiar os arquivos gerados para os diretórios de instalação:
  make install

- Passo 14: Ir para o diretório onde ficam os arquivos de configuração do TORQUE:
  cd /torque/torque_spool

- Passo 15: Editar o arquivo *server_name* para que fique com o *hostname* do servidor do *cluster*:
  echo $HOSTNAME > server_name

- Passo 16: Ir para o diretório com os arquivos de configuração do *pbs_server*:
  cd /torque/torque_spool/server_priv

- Passo 17: Editar o arquivo *hosts* para que contenha todos as máquinas de trabalho do *cluster*. Nesse arquivo, cada linha deve conter o *hostname* seguido do número total de CPUs que a máquina possui. Por exemplo, no *cluster* Oscar, onde os 40 nós de trabalho se chamam *uff2*, *uff3* até *uff41* e cada máquina possui 8 CPUs, as 40 linhas do arquivo de *hosts* ficam da seguinte forma:
  uff2 np=8
  uff3 np=8
  ...
  uff41 np=8

- Passo 18: Ir para o diretório que contém os scripts que iniciam os serviços *pbs_server*, *pbs_sched* e *pbs_mom*:
  cd /root/torque_compilar/torque-3.0.2/contrib/init.d

- Passo 19: Editar o arquivo *pbs_server*, de modo que as duas linhas:
  PBS_DAEMON=/usr/local/sbin/pbs_server
  PBS_HOME=/var/spool/torque
  Passem a ser:
  PBS_DAEMON=/torque/torque_bin/sbin/pbs_server
  PBS_HOME=/torque/torque_spool

- Passo 20: Editar o arquivo *pbs_sched*, de modo que as duas linhas:
  PBS_DAEMON=/usr/local/sbin/pbs_sched
  PBS_HOME=/var/spool/torque
  Passem a ser:
  PBS_DAEMON=/torque/torque_bin/sbin/pbs_sched
  PBS_HOME=/torque/torque_spool

- Passo 21: Editar o arquivo *pbs_mom*, de modo que as duas linhas:
  PBS_DAEMON=/usr/local/sbin/pbs_mom
  PBS_HOME=/var/spool/torque
  Passem a ser:
  PBS_DAEMON=/torque/torque_bin/sbin/pbs_mom
  PBS_HOME=/torque/torque_spool

- Passo 22: Copiar os arquivos *pbs_server* e *pbs_sched* para o diretório */etc/init.d*:
  cp -a pbs_server /etc/init.d/
  cp -a pbs_sched /etc/init.d/

- Passo 23: Modificar o dono, grupo dono, dos arquivos *pbs_server* e *pbs_sched*:
  chmod 755 pbs_server pbs_sched
  chown root:root pbs_server pbs_sched

- Passo 24: Acrescentar os diretórios que contém os arquivos executáveis do TORQUE na variável de ambiente *PATH*:
  export PATH=/torque/torque_bin/sbin:/torque/torque_bin/bin:$PATH
  cp /etc/profile /etc/profile_bkp`date +%F`
  echo 'export PATH=/torque/torque_bin/sbin:/torque/torque_bin/bin:$PATH' » /etc /profile

- Passo 25: Copiar o diretório */torque* existente no servidor do *cluster* para o diretório raiz (/) de todas as máquinas de trabalho do *cluster*. Com isso todas as máquinas de trabalho devem possuir um diretório /torque.

- Passo 26: Copiar o arquivo /root/torque_compilar/torque-3.0.2/contrib/init.d/ pbs_mom existente no servidor o diretório /etc/init.d de todas as máquinas de trabalho do *cluster*.

- Passo 27: Em cada máquina de trabalho do *cluster*, modificar as permissões e o dono do arquivo *pbs_mom*:
  chmod 755 /etc/init.d/pbs_mom
  chown root:root /etc/init.d/pbs_mom

- Passo 28: Retornando ao servidor do *cluster* como usuário root, executar o *pbs_server* pela primeira vez, solicitando que ele crie o arquivo de configurações */torque/ torque_spool/server_priv/serverdb*:
  /etc/init.d/pbs_server create
  Obs: Se o comando acima ficar travado por mais de 20 segundos, digitar *Control + C* para finalizar o comando.

- Passo 29: Parar o *pbs_server*: */etc/init.d/pbs_server stop*

- Passo 30: Iniciar o *pbs_server* e criar as algumas filas e configurações básicas do TORQUE:
  /etc/init.d/pbs_server start
  echo set server operators += $USER | qmgr
  echo set server managers += $USER | qmgr
  qmgr -c 'set server scheduling = true'

qmgr -c 'set server keep_completed = 300'
qmgr -c 'set server mom_job_sync = true'
qmgr -c 'create queue batch'
qmgr -c 'set queue batch queue_type = execution'
qmgr -c 'set queue batch started = true'
qmgr -c 'set queue batch enabled = true'
qmgr -c 'set queue batch resources_default.walltime = 1:00:00'
qmgr -c 'set queue batch resources_default.nodes = 1'
qmgr -c 'set server default_queue = batch'

- Passo 31: Verificar se os comandos *qmgr* fizeram efeito:
  qmgr -c 'print server'

- Passo 32: Iniciar o escalonador do TORQUE:
  /etc/init.d/pbs_sched start

- Passo 33: Verificar se os serviços *pbs_server* e *pbs_sched* estão em execução no servidor:
  /etc/init.d/pbs_server status
  /etc/init.d/pbs_sched status

- Passo 34: Iniciar os serviços de monitoramento *pbs_mom* em todas as máquinas de trabalho. Para isso é preciso logar em cada máquina de trabalho e executar o comando:
  /etc/init.d/pbs_mom start

- Passo 35: Retornar ao servidor do *cluster* e verificar se todas as máquinas de trabalho ficaram disponíveis:
  pbsnodes -a

- Passo 36: Quando as máquinas aparecerem com o status *free* por meio do comando *pbsnodes -a*, significa que um usuário existente no *cluster* poderá submeter um *job* para alguma dessas máquinas com o comando:
  qsub -l nodes=1:ppn=1 script

- Passo 37: Se o escalonamento estiver funcionando, os *jobs* submetidos devem poder entrar em execução nas máquinas e gerar arquivos de output e de erro no diretório de onde foi executado o comando *qsub* quando os *jobs* terminarem. Os *jobs* podem ser acompanhados com o comando *qstat -a*.

- Passo 38: É possível e pode ser desejável, bloquear o acesso *ssh* dos usuários a máquinas de cálculo, se os mesmos não tiverem reservados máquinas pelo comando *qsub*. Para isso, é necessário compilar um arquivo de módulo *pam* disponível no diretório do TORQUE e copiar o arquivo gerado, junto com outros arquivos também disponibilizados, para as máquinas de trabalho. Depois é preciso editar o arquivo */etc/authuser* de cada máquina de trabalho. Para mais detalhes sobre o bloqueio do *ssh*, ler o arquivo:
  /root/torque_ compilar/torque-3.0.2/contrib/README.pam_ authuser
  E descompactar o arquivo:
  /root/torque_ compilar/torque-3.0.2/contrib/pam_ authuser.tar.gz.

Após uma instalação bem sucedida do TORQUE, com o teste de algumas submissões e execuções de *jobs*, é possível tentar refazer uma instalação, dessa vez incluindo os arquivos necessários para substituir o escalonador padrão do TORQUE pelo OscarPBS.

A.2 Configuração e Instalação do OscarPBS

A instalação do TORQUE vista na seção anterior, pode ser modificada para incluir o OscarPBS. Para isso, é necessário que alguns arquivos relacionados ao OscarPBS sejam acrescentados. Durante a instalação será considerado que os arquivos relacionados ao OscarPBS estão disponíveis. Esses arquivos podem ter sido obtidos, por exemplo, obtidos com o autor desse trabalho. Os arquivos necessários para a instalação do OscarPBS e uma breve descrição de cada um deles podem ser vistos na tabela A.1:

A.2.1 Criação do banco de dados:

O OscarPBS requer a existência de um banco de dados no serviço PostgreSQL, bem como a existência de um usuário e senha no PostgreSQL que possua as permissões de *SELECT, INSERT, UPDATE, DELETE* nesse banco de dados. O nome padrão do banco de dados do OscarPBS é *banco_ oscarpbs* e o nome padrão do usuário do banco dados é *admin_ oscarpbs*. Uma vez criado o banco de dados, o usuário, a senha e configuradas as permissões, o arquivo de script SQL *tabelas.sql* deve ser executado por esse usuário, a fim de que as tabelas sejam criadas.

Nome do arquivo	Descrição
fifo.c	Substitui o escalonador do TORQUE e apenas repassa os eventos recebidos do *pbs_server* para o OscarPBS.
config_fifo	Contém parâmetros utilizados pelo fifo.c, como qual o nome e o caminho do programa que gera sinais para o OscarPBS.
dispara_evento.c	Executado através do fifo.c, é o programa que envia um sinal SIGINT para o OscarPBS, causando um evento.
oscarpbs.py	É o programa principal do OscarPBS, incluindo o loop de submissões de *jobs*.
tabelas.sql	Script SQL utilizado que gerar as tabelas do banco de dados.
config_oscarpbs	Arquivo de configuração e parâmetros do OscarPBS.
hosts_oscarpbs	Arquivo com os *hostnames* das máquinas de trabalho que o OscarPBS deve considerar.
script_submissoes.py	Script utilizado nos experimentos que faz submissões artificiais para o OscarPBS.
arquivo_submissoes	Arquivo com dados sobre submissões de *jobs*, usado pelo script_submissoes.

Tabela A.1: Arquivos necessários para a instalação do OscarPBS

A.2.2 Preparando a instalação:

Antes da instalação do TORQUE que utiliza o OscarPBS, a estrutura do OscarPBS pode ser criada. Para isso deve-se seguir os seguintes passos:

- Passo 1: Conectar á máquina servidora do *cluster* e se tornar *root*.
- Passo 2: Criar um diretório onde vão ficar os principais arquivos do OscarPBS e um diretório para os arquivos de *log* gerados pelo OscarPBS:
  cd /
  mkdir /oscarpbs
  mkdir /oscarpbs/logs
- Passo 3: Copiar os arquivos do OscarPBS *config_fifo*, *dispara_evento.c*, *oscarpbs.py*, *config_oscarpbs*, *hosts_oscarpbs*, *script_submissoes.py* e *arquivo_submissoes* para o diretório /oscarpbs
- Passo 4: Configurar o dono e o grupo dono dos arquivos:
  cd /oscarpbs

*chown root:root **

- Passo 5: Configurar uma permissão padrão para todos os arquivos:
  *chmod 600 **
- Passo 6: Compilar o arquivo *dispara_evento.c*:
  gcc dispara_evento.c -o dispara_evento
- Passo 7: Dar permissão de execução para os arquivos *oscarpbs.py*, *dispara_evento* e *script_submissoes.py*:
  chmod 700 oscarpbs.py dispara_evento script_submissoes.py
- Passo 8: Editar o arquivo *config_oscarpbs*, colocando no parâmetro *senha_bd* a senha que foi escolhida para usuário *admin_oscarpbs* do banco de dados do OscarPBS:
  senha_bd=senha_escolhida
- Passo 9: Editar o arquivo *hosts_oscarpbs* para conter os *hostnames* das máquinas de trabalho do *cluster* que devem ser levadas em consideração pelo OscarPBS quando for escalonar *jobs* e o número de CPUs que cada máquina. A sintaxe é semelhante a utilizada pelo arquivo *hosts* do TORQUE. Supondo que se queira utilizar 10 máquinas com *hostnames* maquina_de_trabalho1 até maquina_de_trabalho10 e que se queira utilizar 8 CPUs de cada, o arquivo *hosts_oscarpbs* ficaria com 10 linhas:
  maquina_de_trabalho1 ppn=8
  maquina_de_trabalho2 ppn=8
  ...
  maquina_de_trabalho10 ppn=8

Isso prepara o OscarPBS para ser executado pelo TORQUE. Agora é necessário que o próprio TORQUE seja instalado, de modo a ter um módulo *pbs_sched* modificado, que ao invés de fazer o escalonamento, repasse o evento que seria utilizado pelo escalonador padrão do TORQUE para o OscarPBS.

A.2.3 Instalação do TORQUE modificado para utilizar o OscarPBS:

A instalação do TORQUE que utiliza o OscarPBS pode ser feita da mesma maneira que foi mostrada na seção anterior, exceto que alguns passos são modificados. Supondo que

não exista um diretório */torque* no servidor nem nas máquinas de trabalho, todos os passos da instalação do TORQUE da seção anterior podem ser repetidos, a menos das seguintes modificações:

- Imediatamente após o *Passo 10*, (antes de se executar o script *configure*), deve-se copiar o arquivo *fifo.c* do OscarPBS para o diretório:
  /root/torque_compilar/torque-3.0.2/src/scheduler.cc/samples/fifo/
  substituindo o arquivo *fifo.c*:
  /root/torque_compilar/torque-3.0.2/src/scheduler.cc/samples/fifo/fifo.c
  que lá se encontra.

- No *Passo 20* em que duas linhas do arquivo *pbs_sched* são modificadas, deve-se acrescentar a mudança de mais uma linha, totalizando 3 linhas modificadas. A nova linha a ser modificada deve ser mudada de:
  daemon $PBS_DAEMON -d $PBS_HOME
  para:
  daemon $PBS_DAEMON /oscarpbs/config_fifo -d $PBS_HOME

Após finalizado todo o processo de instalação do TORQUE, é preciso iniciar o OscarPBS para que ele receba os eventos lançados pelo TORQUE ou pelo script de submissões:

/oscarpbs/oscarpbs.py &

A submissão de *jobs* para o OscarPBS, na versão 1.0, se dá apenas artificialmente. O script *script_submissoes.py* simula a submissão de *jobs* utilizando dados sobre /textitjobs contidos no arquivo *arquivo_submissoes*. Seria possível colocar o escalonador OscarPBS para trabalhar, utilizando esse script para mandar *jobs* para ele:

/oscarpbs/script_submissoes.py arquivo_submissoes

No entanto o arquivo *arquivo_submissoes* precisaria ser editado para conter máquinas e contas de usuários que existam no servidor onde o OscarPBS será executado, pois nesse arquivo existem contas de usuário e máquinas que existem no *cluster* Oscar.

A.3 Limitações

Atualmente, na versão 1.0 do OscarPBS, a submissão de *jobs* funciona da seguinte forma: A submissão de *jobs* é feita artificialmente por meio de um *script* criado para a realização dos experimentos. Esse *script* lê dados de submissões a partir de um arquivo que contém essas informações (recursos, tipo, prazo, poderes, *walltime*, usuário e instante da submissão) e simula cada uma dessas submissões do arquivo, como se fossem usuários submetendo *jobs* por meio de um comando *qsub*. O *script* então grava a submissão no banco de dados e logo em seguida envia um sinal SIGINT causando um evento no OscarPBS. A gravação no banco de dados e o disparo do evento deveriam ser feitos através de um comando *qsub* específico do OscarPBS e disponível para os usuários reais, mas que não está implementado no OscarPBS atual (a versão 1.0).

De forma semelhante, outros dois comandos de interface de usuário que serão necessárias para acompanhamento e exclusão de *jobs* reais no OscarPBS ainda não estão implementadas: O comando *qdel*, que permite que o usuário possa remover um *job* submetido para o OscarPBS. Esse comando deve criar um registro de banco de dados que solicita uma exclusão de *job* e disparar um evento para o OscarPBS. E o comando *qstat*, que deve permitir que um usuário consulte o banco de dados do OscarPBS para verificar o status de seus *jobs*. Atualmente essas ações só são possíveis através de consulta direta ao banco de dados, no caso do *qstat*, ou pelo criação de um registro de solicitação de exclusão de *job* diretamente no banco de dados, seguida de um comando *kill -SIGINT pid*, onde *pid* é o número de identificação do processo do escalonador OscarPBS.