UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Convertendo Recursos Compartilhados No Espaço Em Recursos Intermitentes

Lauro Beltrão Costa

Dezembro 2005

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Convertendo Recursos Compartilhados No Espaço Em Recursos Intermitentes

Lauro Beltrão Costa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Informática (MSc).

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Walfredo Cirne
(Orientador)

Dezembro 2005

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O148a Costa, Lauro Beltrão

2006 Convertendo Recursos Compartilhados No Espaço Em Recursos Intermitentes – Campina Grande, 2005

123.: il.

Inclui bibliografia

Dissertação (Mestrado em Informática) - Universidade Federal de Campina Grande,

Centro de Engenharia Elétrica e Informática.

@Orientadores 1– Aprendizagem de Máquina 2– Negociação Automatizada 3– Inteligência Artificial

4– Redes Neurais I-Título

CDU 681.3.06

"CONVERTENDO RECURSOS COMPARTILHADOS NO ESPAÇO EM RECURSOS INTERMITENTES"

LAURO IVO BELTRÃO COLAÇO COSTA

DISSERTAÇÃO APROVADA EM 05.12.2005

PROF. WALFREDO DA COSTA CIRNE FILHO, Ph.D
Orientador

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Examinador

PROF. PHILIPPE OLIVIER ALEXANDRE NAVAUX, Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

Grids Computacionais estão se transformando de promessa em realidade. Naturalmente, isto não está acontecendo para todas as aplicações em uma vez. As *aplicações Bag-of-Tasks* (BoT) (aplicações paralelas compostas por tarefas independentes) são, devido à sua simplicidade, a primeira classe de aplicações a serem executadas de maneira maciça em grids (e.g. SETI@home). As aplicações BoT são especialmente adequadas para a execução em grids pois podem rodar em *recursos intermitentes* (i.e. recursos sem garantias de disponibilidade ou de confiabilidade). Neste cenário, bom desempenho e resultados confiáveis são fornecidos por *escalonadores gulosos* (*eager schedulers*), que utilizam replicação para compensar más associações tarefa-processador. Todavia, escalonadores gulosos não são preparados para usar *recursos compartilhados no espaço* (e.g. supercomputadores paralelos). Isto acontece porque usar um recurso compartilhado no espaço envolve submeter um pedido detalhado ao escalonador do recurso, especificando o número dos processadores e a quantidade de tempo estes processadores devem ser alocados - uma informação que os escalonadores gulosos não estão preparados para fornecer. Uma vez que recursos compartilhados no espaço são os mais poderosos recursos computacionais, o uso deles poderia melhorar o tempo de execução de aplicações BoT. Este trabalho propõe uma maneira automática de elaborar tais pedidos ao recurso compartilhado no espaço a fim de converter recursos compartilhados no espaço em intermitentes, conseqüentemente tornando-os naturalmente usáveis por escalonadores gulosos. Tal conversão é baseada em heurísticas que visam diminuir o tempo de execução das aplicações e permite aos escalonadores gulosos o uso destes poderosos recursos computacionais sem modificações.

Abstract

Grid Computing is turning from promise into reality. Naturally, this is not happening for all applications at once. Bag-of-Tasks (BoT) applications (those parallel applications whose tasks are independent) are, due to their simplicity, the first class of applications to be massively executed on grids (e.g. SETI@home). BoT applications are especially amenable to grid execution because they can run on *intermittent resources* (i.e. resources with no guarantees on availability or reliability). In this scenario, good reliability and performance are provided by *eager schedulers*, which rely on replication to overcome unfortunate task-to-processor assignments. However, eager schedulers are not prepared to use *space-shared resources* (e.g. parallel supercomputers). This happens because using a space-shared resource involves submitting a detailed request to the resource scheduler, specifying the number of processors needed and the amount of time these processors are to be allocated - an information that eager schedulers are not prepared to provide. This is unfortunate because space-shared resources are the most powerful computing resources available today and thus could greatly improve the execution time of BoT applications. This work proposes an automatic way to craft such requests in order to convert space-shared resources into intermittent ones, therefore rendering them naturally usable by eager schedulers. Such conversion is based on adaptive heuristics and allows eager schedulers use such powerful computing resources without modifications.

Agradecimentos

Há pouco menos de seis anos eu chegava a Campina Grande para fazer vestibular. Os períodos de graduação e mestrado se confundem. Foi o tempo em que conheci muita gente, novas amizades se formaram e antigas se firmaram. É impossível agradecer da maneira como deveria a todos aqui, mas tentarei demonstrar uma parcela disso tudo.

A Deus.

A minha família, a quem devo não só o início da minha formação e o apóio, mas também uma relação de grande amizade que construímos.

Ao grande guru Walfredo Cirne, com quem trabalho há mais de quatro anos e que me ofereceu diversas oportunidades de crescimento ao longo deste período. Agradeço ainda a W pelo clima de descontração que me deixou à vontade para em nossos encontros expor minhas dúvidas e os diversos problemas que encontrava. Ao meta-guru Jacques Sauvé, com quem nunca trabalhei diretamente através de projetos, mas que teve uma enorme influência em minha formação (não só acadêmica) ao longo de nossas idas à natação e nossos almoços.

A Daniel Fireman, "fiel escudeiro" durante meu mestrado, que me proporcionou uma experiência bastante enriquecedora como "guru". Experiência que me lembrei como ponto positivo nos momentos em que fiquei desmotivado com o trabalho.

A todos os "vagabundos" (Gustavo, Fábio, Filipe e Flávio) com quem dividi casa durante a maior parte do mestrado e conheço deste o início da graduação. Momentos de discussões "óbvias" e "não óbvias", brigas pela conta de energia, telefone e pelo conserto da geladeira ;-). Apesar disto tudo, agradeço principalmente a oportunidade de conhecê-los melhor e os momentos de não-solidão.

O LSD cresceu muito deste que comecei a trabalhar nele (acredito que hoje tenha uma área 20 vezes maior). Além de acompanhar e me sentir feliz com este crescimento, as pessoas que o formam me proporcionaram a maior parte dos melhores momentos que passei em Campina Grande durante o mestrado. Tendo foco no período do mestrado, agradeço ao professor Fubica e a Aliandro, Alisson, Ayla, Daniel Paranhos, Eliane (pequerrucha), Elizeu, Érica, Kátia Saikoski, Kêka, Lívia, Nazareno, Milena, Mirna, Raquel, Randolph, Robson, William, Zane e a todo pessoal do LSD.

Além das pessoas do LSD, outras de Campina estiveram presentes. Agradeço especialmente a Cássio, Nivaldo Catão, Eloi, Eusa, Heliton e Rebouças. Agradeço também a Aninha, com seu sorriso sempre presente, pela assistência da COPIN e ao pessoal da Fundação Sementes de Vida.

Além das pessoas de Campina, gostaria de agradecer a alguns amigos que estão distantes: Klebson, Ely, João Carlos e João Paulo.

Por fim, meio que parafraseando Nazareno e Dalton, agradeço a todos aqueles que, através do Governo Brasileiro, financiaram parte da minha formação.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Grids computacionais são plataformas para a execução de aplicações paralelas que agregam recursos distribuídos e heterogêneos, oferecendo acesso a esses recursos independente da localização física [37]. Grids tornam possível a integração de vários recursos dispersos como um único computador virtual tornando mais rápida a execução de várias aplicações paralelas.

A heterogeneidade, associada aos requisitos de escalabilidade e descentralização, inerentes a um ambiente como o de grids computacionais, dificulta a coordenação dos recursos e a submissão de uma aplicação, especialmente se o usuário as realiza sem nenhuma automação. É neste cenário que surge a figura do *broker* de recursos (e.g. Nimrod/G [15], MyGrid [23] e Condor-G [39]). Um *broker* de recursos, ou simplesmente *broker*, é o responsável por acessar aos recursos nos diversos domínios administrativos que compõem o grid e coordená-los de modo a fornecer ao usuário a abstração de um meta-computador. O *broker* também é responsável por receber a aplicação do usuário e decidir em quais recursos ela deverá executar (i.e. o *broker* contém um escalonador). Assim, cabe ao *broker* lidar com a heterogeneidade dos recursos e fazer com que o uso do grid seja uma tarefa fácil para o usuário.

O uso de *brokers* como abordagem para utilização de grids é bem consolidado tanto em ambientes empresariais quanto em ambientes científicos. Como exemplos científicos, podemos citar Nimrod/G [15], MyGrid [23] e Condor-G [39]. Um ambiente empresarial que também utiliza *brokers* é o *Community Scheduler Framework* (CSF) [67]. O CSF é uma implementação de código aberto feita pela Platform Computing Inc. [8] e apoiada

por várias empresas (e.g. HP, IBM, AMD, JP Morgan e BEA Systems).

Um tipo de aplicação paralela que executa sobre um grid são as aplicações *Bag of Tasks* (BoT). Elas são aplicações paralelas compostas por tarefas independentes entre si. Independente significa que não há comunicação entre as tarefas e a ordem da execução não influencia o resultado. Apesar desta simplicidade, aplicações BoT são usadas em várias áreas como: mineração de dados [64], buscas massivas [15] (por exemplo, quebra de chaves), simulações Monte Carlo, aplicações em bio-informática [70], processamento de imagens [65, 66] e cálculo de fractais (como o Mandelbrot).

Devido à independência entre as tarefas (o que diminue a necessidade de comunicação), *brokers* para aplicações BoT têm conseguido obter bons resultados [11, 15, 23, 39, 40, 56, 59, 62]. Inclusive já obtiveram sucesso sobre grids computacionais de escala planetária. Exemplos disto são os projetos SETI@home [5, 11] e United Devices [6]. O SETI@home já conseguiu cerca de 2.000.000 de anos de CPU através de 5.000.000 de voluntários distribuídos em 226 países diferentes [4].

Tipicamente *brokers* para aplicações BoT estão preparados para lidar com recursos intermitentes. Um recurso *intermitente* é um recurso computacional que (i) se torna disponível e indisponível para o grid sem aviso prévio, (ii) tem poder de processamento desconhecido e variável e (iii) não oferece garantias sobre a computação realizada (i.e. resultados incorretos são possíveis). Apesar destas características, bom desempenho e resultados confiáveis ainda podem ser obtidos através de escalonadores gulosos (*eager schedulers*) [40, 56, 59, 62] que estão contidos no *broker*. Eles são escalonadores que fazem uso de replicação para tolerar a variação de desempenho bem como garantir a credibilidade dos resultados [63].

Recurso intermitente é uma abstração. Na prática, cada recurso tem seu próprio escalonador que controla o recurso de acordo com as políticas locais, de maneira que não é possível utilizar o recurso sem passar por tal escalonador. Projetos de computação voluntária [6, 11] disponibilizam o *download* de programas para que os voluntários possam prover seus processadores. Assim, quando a máquina fica ociosa, o programa (*worker*) executa e torna a máquina disponível para que os *brokers* destes projetos enviem tarefas para serem executadas. Quando o usuário volta a utilizar a máquina o (*worker*) não permite mais que tarefas sejam executadas até que o recurso se torne

ocioso novamente. Ou seja, a máquina se torna indisponível para o *broker*.

Por outro lado, recursos compartilhados no tempo (*time-shared*) podem ser utilizados a qualquer momento por um usuário, enquanto os escalonadores de recursos compartilhados no espaço (*space-shared*) gerenciam requisições de vários usuários de maneira que uma partição do recurso (um subconjunto dos processadores) atende a apenas uma requisição de um usuário por vez. Há portanto a possibilidade que requisições esperem até que a partição requisitada esteja livre. Assim, existem diferentes procedimentos para prover a abstração de recurso intermitente aos *brokers*.

O uso de recursos compartilhados no tempo (e.g. máquina Linux onde o usuário tem acesso via *secure shell - ssh*) por esses *brokers* é um problema bem resolvido. Em particular, um recurso compartilhado no tempo pode ser visto como um recurso intermitente que está disponível todas as vezes que está ligado. Dessa maneira, utilizar um recurso compartilhado no tempo como um recurso intermitente para as soluções de escalonamento existentes é trivial (o *broker* submete uma tarefa quando detecta que a máquina está ligada).

No entanto, quando se quer usar recursos compartilhados no espaço via um *broker* para aplicações BoT, a situação não é tão simples. O uso deste tipo de recurso requer a submissão de uma requisição detalhada que especifica a quantidade de processadores (p) e o montante de tempo (t^r) durante o qual o usuário deseja utilizar os recursos. O problema é que escalonadores gulosos não estão preparados para realizar uma requisição desse tipo. Estes escalonadores assumem que tudo o que é preciso ser feito é enviar uma tarefa para ser executada quando o recurso se torna disponível.

Recursos compartilhados no espaço são projetados com o objetivo de prover grande poder de processamento. Eles incluem supercomputadores paralelos como o IBM SP2 e Cray XT3, como também *clusters* de menor porte montados pelo próprio usuário. A utilização traria maior capacidade de processamento ao grid e foi uma funcionalidade procurada por diversos os usuários acadêmicos (e.g. UFCG, UFES, UFRJ, PUCRS) do projeto MyGrid/OurGrid [22,23]. O OurGrid é um projeto desenvolvido na UFCG que tem como objetivo prover fácil acesso a recursos computacionais aos usuários através da comunidade para troca de recursos ou a partir dos recursos aos quais o usuário tem acesso diretamente. Para utilizar os recursos compartilhados no espaço, estes usuários

tinham que especificar as requisições (tipicamente requisitavam os valores máximos permitidos para os parâmetros) e preparar (e.g. instalação de software e execução de *daemons*) as máquinas para executar tarefas do grid.

Para que os usuários possam automaticamente utilizar no grid os recursos compartilhados no espaço, duas abordagens podem ser seguidas: (i) modificar os escalonadores gulosos atuais para lidarem com recursos diferentes de intermitentes ou (ii) ter uma maneira de realizar a conversão dos recursos compartilhados no espaço em recursos intermitentes. Neste trabalho, nós optamos pela segunda abordagem: elaborar um modo de converter dos recursos compartilhados no espaço (que necessitam receber requisições ricas em informação) em recursos intermitentes. Optamos por esta abordagem pois ela nos permite utilizar toda a pesquisa já realizada no sentido de utilizar recursos intermitentes de um grid. Assim, essa conversão possibilita que várias soluções de escalonamento que utilizam recursos intermitentes possam ser empregadas com recursos compartilhados no espaço sem modificação.

Com base nessa decisão, propomos a criação de um módulo para realizar requisições ao escalonador do recurso compartilhado no espaço. Isto implica em definir os parâmetros tempo necessário e número de processadores. Uma vez definidos esses parâmetros, o módulo submete um programa (*worker*) que será o responsável por receber as requisições para executar tarefas provenientes da aplicação do usuário. O objetivo é elaborar uma maneira automática de definir esses parâmetros através de heurísticas, visando diminuir o tempo necessário para o término da aplicação submetida pelo usuário. Nossa solução deve atender aos seguintes requisitos:

1. ser automática (elabora as requisições sem intervenção do usuário);
2. não estar acoplada à aplicação (deve funcionar para várias aplicações);
3. apresentar algum mecanismo que visa diminuir o tempo necessário para o término da aplicação.

Note que estamos elaborando heurísticas que adquirem conhecimento sobre o recurso e a aplicação durante a execução das tarefas, sendo capazes de tomar decisões durante a execução da aplicação. Essa capacidade de adquirir conhecimento e utilizá-

lo nas decisões nos fornece a possibilidade de adaptação do escalonamento em um ambiente dinâmico como é um grid computacional.

O restante da dissertação está organizado da seguinte forma. O próximo capítulo provê contextualização nas áreas envolvidas e apresenta trabalhos relacionados, definindo o conceito de grids computacionais, recursos compartilhados no espaço e fazendo uma pequena análise sobre as *workloads* destes recursos. O Capítulo 3 propõe nossa solução, descrevendo a arquitetura utilizada e as heurísticas para conversão dos recursos compartilhados no espaço em recursos intermitentes, bem como discutindo as questões que nos levaram à elaboração dessas heurísticas. No Capítulo 4, avaliamos o desempenho da solução através de simulação com utilizando o tempo para o término dos jobs como métrica de avaliação. O Capítulo 5 apresenta a implementação da solução utilizando o *broker* MyGrid [23]. As conclusões finais e idéias para trabalhos futuros são apresentadas no Capítulo 6.

Capítulo 2

Estado da Arte

Neste capítulo apresentamos as duas áreas nas quais estivemos envolvidos durante este trabalho: grids computacionais e escalonamento de recursos compartilhados no espaço. Além de apresentarmos os principais conceitos destas áreas, também mostramos o estado atual delas bem como introduzimos alguns conceitos básicos de aplicações paralelas comuns às duas áreas.

2.1 Aplicações Paralelas

Uma *aplicação paralela* é resultado da divisão de uma aplicação em vários "pedaços" ou unidades de processamento, chamadas *tarefas*, que executam em processadores distintos mas que, dependendo das características da aplicação, podem se comunicar. Esta divisão de trabalho faz com que a aplicação tenha um tempo de execução menor do que se executasse seqüencialmente, como uma única grande tarefa. De acordo com a troca de informações entre as tarefas, aplicações paralelas podem ser divididas como:

- **Fracamente acopladas:** Aplicações que não necessitam de uma grande troca de informações entre as tarefas. Por exemplo: Fatoração de números.
- **Fortemente acopladas:** Aplicações que necessitam de uma grande troca de informações entre as tarefas. Por exemplo: Um resolvedor Jacobi.

A Figura 2.1 mostra de forma simplificada como o desempenho das aplicações paralelas tende a se comportar. Inicialmente, quanto maior o número de máquinas. mais

rápida é a execução da aplicação, até chegar a um número de máquinas que minimiza o tempo de execução. Depois o tempo de processamento da aplicação volta a aumentar, devido ao aumento da comunicação entre as tarefas (*overhead*), podendo até ultrapassar o tempo da aplicação caso não houvesse paralelização. Esse comportamento pode variar dependendo da velocidade de conexão entre as máquinas, do poder de processamento destinado às tarefas e da aplicação.

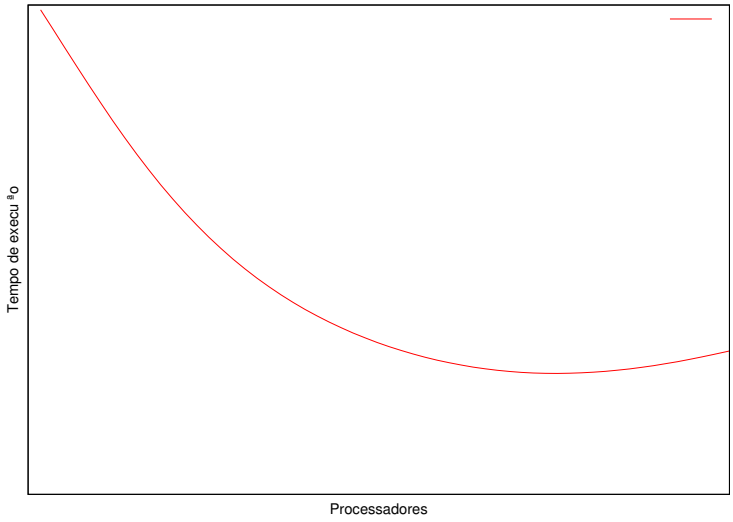


Figura 2.1: Comportamento comum de uma aplicação paralela.

Uma classe especial de fraco acoplamento é formada por as aplicações *Bag of Tasks* (BoT). Aplicações BoT são aplicações paralelas compostas por tarefas independentes entre si. Por independente entende-se que não há comunicação entre as tarefas e que a ordem de execução das tarefas não influencia no resultado. Ou seja, elas têm grau mínimo de acoplamento. Apesar desta simplicidade, aplicações BoT são usadas em vários cenários como: mineração de dados [64], varredura de parâmetros (*parameter sweep*) [15], simulações Monte Carlo, aplicações em bio-informática [70], processamento de imagens [65, 66] e cálculo de fractais (como o Mandelbrot).

2.2 Recursos compartilhados no espaço

Recursos compartilhados no espaço, como supercomputadores paralelos com memória distribuída, são recursos projetados para suportar a execução de aplicações parale-

las. Um recurso compartilhado no espaço é composto por vários nós (cada um com sua própria memória) interconectados por uma rede local. A característica de compartilhamento no espaço visa promover o desempenho das aplicações paralelas. Ou seja, cada aplicação recebe uma partição (subconjunto dos processadores) dedicada para executar por um período de tempo pré-determinado. Como exemplos de recursos deste tipo temos: os supercomputadores IBM SP2, Cray XT3, BlueHorizon assim como pequenos *clusters* com um gerente de recursos que usam compartilhamento no espaço.

Uma vez que uma aplicação tem acesso dedicado a uma partição do recurso, uma requisição para rodar uma aplicação pode não encontrar uma partição livre para atendê-la imediatamente. Neste caso, a requisição é colocada em uma fila de espera até que processadores suficientes se tornem disponíveis para ela. Isto implica que uma requisição pode não ganhar o recurso imediatamente e cabe ao escalonador do recurso gerenciar a fila de espera [17, 33].

O escalonador do recurso, ou gerente do recurso, decide qual a próxima requisição a ser atendida e quais processadores serão utilizados por ela. Para tomar esta decisão, estes recursos tipicamente solicitam que os usuários especifiquem quantos processadores serão utilizados (p) e quanto tempo (t^r) a partição deverá ser alocada para a execução da aplicação. Note que o gerente do recurso encerra a aplicação caso sua execução exceda o tempo (t^r).

2.2.1 Atendimento das requisições

A necessidade de especificar o tempo durante o qual a partição será alocada ocorre devido aos escalonadores de recursos compartilhados no espaço utilizarem algoritmos com *backfilling* [55] para decidir qual requisição será a próxima a executar. Tais algoritmos tornam possível utilizar recursos que seriam desperdiçados utilizando o algoritmo primeiro-que-chega-é-o-primeiro-servido (*First-Come-First-Serve - FCFS*) pois permitem que requisições novas possam ser atendidas antes de requisições mais antigas. Por exemplo: com FCFS, uma requisição que pede por todos os processadores de um recurso compartilhado no espaço impede que qualquer outra requisição mais nova ganhe uma partição do recurso mesmo que existam processadores disponíveis em quantidade suficiente para executar as novas requisições.

Atualmente, existem vários escalonadores de recursos compartilhados no espaço com algoritmos de *backfilling*. Exemplos incluem PBS [13], Maui [45], Easy [51] e Crono [58]. Todavia, na prática, o comportamento de cada escalonador varia de recurso para recurso mesmo que o software utilizado seja idêntico. Isto ocorre porque existem diversas políticas que são configuradas diferentemente em cada recurso. Apesar dessas diferenças, todos esses escalonadores têm em comum as seguintes características [17]:

1. O atendimento das requisições não necessariamente tem que obedecer a ordem de submissão. Ou seja, a requisição B pode chegar depois da requisição A e ainda assim começar a execução antes.

2. O tempo que estava alocado para uma requisição pode ser utilizado para outra caso a primeira acabe a execução antes da decorrência do tempo requerido.

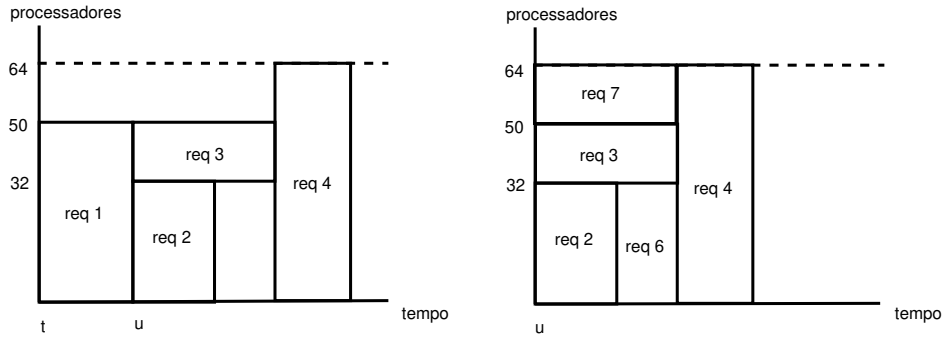
3. Requisições antigas têm mais chances de ganhar o recurso que as requisições novas.

4. Há algum mecanismo para evitar que requisições grandes sejam postergadas indefinidamente (i.e. evita-se *starvation*).

Neste trabalho, vamos considerar o algoritmo *conservative backfilling* como algoritmo para atendimento das requisições nos recursos compartilhados no espaço. Tomamos esta decisão porque *conservative backfilling* é um exemplo representativo dos algoritmos atualmente em produção (atendendo as 4 características citadas acima) e provê bom desempenho [55]. A principal idéia deste algoritmo é que uma nova requisição é inserida no primeiro espaço no qual ela encaixa (considerando como início da fila o instante da submissão da requisição). Ou seja, qual o instante mais breve que terá p processadores livres pelo tempo t^r. Quando uma requisição se encerra, o escalonador verifica se a primeira requisição pode começar a utilizar o recurso. Se esta não puder pois não há processadores livres pelo tempo necessário, o escalonador varre a fila procurando a primeira requisição que: (i) pode ser executada com a quantidade atual de processadores livres e (ii) não irá atrasar qualquer outra requisição na fila. Esta abordagem garante previsibilidade, gerando um limite superior para o término da requisição quando da submissão do job [55].

Algoritmos com *backfilling* fazem com que o atendimento de uma requisição seja afetado pela sua forma ou área ($p \times t^r$) porque quanto maior for sua área mais difícil será para encaixar a requisição em um espaço (*slot*) da fila.

A Figura 2.2 exemplifica o funcionamento do algoritmo *conservative backfilling*. Temos os estados da fila de um recurso compartilhado no espaço composto por 64 processadores em dois momentos. A Figura 2.2(a) apresenta o estado em um dado momento t. Após algum tempo, no momento u, as requisições 6 e 7 são submetidas ao sistema (Figura 2.2(b)) e conseguem ganhar acesso ao recurso antes da requisição 4 pois suas áreas menores permitem que utilizem os processadores sem atrasar nenhuma requisição que tenha chegado antes.



(a) (b)

Figura 2.2: Uma fila de um recurso compartilhado no espaço que utiliza *conservative backfilling*

2.2.2 Aplicações com flexibilidade nas requisições

Voltando a atenção para o escalonamento de recursos compartilhados no espaço e o impacto das requisições no desempenho das aplicações, encontramos diversos trabalhos que analisam essas questões. Dror Feitelson e Larry Rudolph [32] classificam as aplicações para os recursos compartilhados no espaço. Segundo esses autores, a principal característica que influencia a área é a flexibilidade (escolha da quantidade de processadores). Sob este critério, os principais tipos de aplicações são:

- **Rígidas:** O número de processadores que a aplicação utilizará é especificado pelo usuário ou pelo programador e não pode ser mudado.

- **Moldáveis:** O número de processadores dedicados à aplicação pode ser decidido pelo sistema de acordo com algumas restrições da aplicação e da carga do recurso. Neste caso, o usuário pode fornecer algumas possibilidades ao escalonador no momento da submissão e cabe ao escalonador escolher a melhor possibilidade que será de fato submetida à fila. Este número não muda durante a execução da aplicação.

- **Maleáveis:** O número de processadores dedicados à aplicação pode mudar durante a execução como resultado de decisões do escalonador do recurso.

Atualmente, os escalonadores de recursos compartilhados no espaço aceitam requisições para aplicações rígidas [44, 51] (os escalonadores oferecem apenas a interface para submissão que recebe (p, t^r) e, desta maneira, a maior parte da pesquisa realizada assume esse tipo de aplicação [9, 10, 42, 53, 55]. Todavia, existem evidências de que o desempenho da aplicação e a utilização do recurso podem ser melhorados permitindo que o usuário submeta aplicações moldáveis [20, 21, 28, 54] ou mesmo maleáveis [60].

Os trabalhos que tratam requisições não rígidas conseguem obter melhor desempenho devido à manipulação que é feita com a forma da requisição. No caso das aplicações moldáveis, o usuário passa ao escalonador do recurso uma coleção de possíveis formas para a execução da aplicação. O escalonador verifica o estado da fila e escolhe a área que proveja o melhor desempenho para a aplicação. As aplicações maleáveis permitem uma flexibilidade ainda maior pois o escalonador pode dar ou retirar processadores para uma aplicação que já está em execução.

Apesar de Dror Feitelson e Larry Rudolph em [32] terem atribuído ao escalonador do recurso a responsabilidade da escolha das formas, essa escolha pode ser implementada a nível da aplicação. Aplicações moldáveis podem usar um programa que recebe a coleção de formas possíveis, consulta o escalonador do recurso sobre o estado da fila, escolhe a melhor forma e faz a requisição. De maneira semelhante, as aplicações maleáveis podem utilizar essa característica para melhorar seu desempenho mesmo sem que escalonador do recurso dê suporte para isso: a aplicação pode monitorar a fila e verificar quando é um bom momento para submeter uma nova requisição, a nova requisição ganha a partição do recurso e os processadores dessa nova requisição podem executar tarefas

da mesma aplicação. Soluções deste tipo permitem que os resultados das pesquisas possam ser aplicados em sistemas que estão em produção atualmente, mesmo que os escalonadores só aceitem aplicações rígidas.

Consideramos agora o caso das aplicações BoT executadas através de um *broker*. Como as aplicações não têm restrições com relação ao número de processadores que irão conseguir, o *broker* pode tentar verificar o estado da fila e submeter as requisições que possam promover um melhor desempenho para a aplicação. Como os escalonadores de recurso não aceitam que a requisição mude o formato durante a execução, a quantidade de processadores de uma requisição é fixa mas nada impede que uma nova requisição da mesma aplicação seja submetida e assim a quantidade de processadores dedicados a uma aplicação mude. Isto faz com que aplicações BoT sejam um caso especial de aplicações maleáveis onde as tarefas não precisam executar simultaneamente. Também podem ocorrer momentos após o início da execução quando nenhum processador está alocado para a aplicação e a execução ainda não foi concluída. Entretanto, do nosso conhecimento, não existem trabalhos que analisem o impacto deste tipo em específico de aplicação .

Assim, em [25] realizamos um trabalho preliminar ao apresentado nesta dissertação e verificamos o impacto caso aplicações BoT pudessem ser tratadas como tais pelos escalonadores de recursos compartilhados. A maior parte dos jobs BoT consegue um melhor de desempenho, sendo a melhora maior para jobs maiores.

2.2.3 Tempo requisitado

Diversos trabalhos estudaram as características de *traces* de sistemas em produção mostrando a relação entre o tempo requisitado (t^r) e o tempo de execução. Cirne e Berman mostram em [19] que em quatro diferentes sistemas 50% a 60% das requisições usam menos de 20% do tempo requisitado. Ward et al. em [74], apresentam um sistema Cray T3E com uma média de apenas 29% de utilização dos tempos pedidos. Outros trabalhos [16, 31, 48, 55, 73] apresentam características semelhantes de vários sistemas.

Com essas diferenças, os usuários pagam com um tempo de espera na fila maior do que se requisitassem o tempo baseado em uma boa estimativa. Estes mesmos trabalhos apresentam várias justificativas para este comportamento onde mesmo com

as vantagens de requererem um tempo menor, os usuários continuam pedindo muito mais do que o necessário. Três principais razões são apontadas para tal comportamento: (i) o tempo requerido não é uma estimativa do tempo de execução; na verdade funciona como um limite máximo e se a aplicação o excede, a requisição será cancelada antes do seu término, perdendo assim a execução e fazendo com que os usuários superestimem o tempo; (ii) o tempo ganho na espera em fila não é um estímulo suficiente para os usuários tentarem melhorar as requisições e (iii) parte das requisições falha logo no início da execução devido a erros de configuração (e.g. nome de arquivo errado), o que nos leva a pensar que a requisição teve um tempo super estimado quando na verdade ela utilizou muito menos tempo porque falhou.

Lee et al. em [48] apresentam um estudo sobre os resultados das análises relacionadas ao tempo pedido de diversos trabalhos. Os autores tentam analisar se os fatores anteriormente apontados são justificativas para a grande diferença e o tempo realmente utilizado. O artigo foca em dois fatores que podem causar tais diferenças: (i) tempo requisitado é utilizado como um limite máximo para execução e (ii) as vantagens de especificar um tempo melhor (conseguir acesso mais rápido ao recurso) não motivam suficientemente os usuários.

Visando evitar estes fatores, os usuários de Blue Horizon no SDSC (*San Diego Supercomputer Center* - Centro de Supercomputação de San Diego) foram convidados a participar de um experimento em que se pediam alguns parâmetros a mais na submissão da requisição. Além do limite máximo para a execução eles também forneciam um tempo estimado (que não era levado em consideração pelo escalonador do recurso) e um grau de confiança (*confidence*) para as suas estimativas. Para motivar os usuários a fornecerem boas estimativas, foram oferecidos prêmios como *MP3 players* e *USB pen drives* para usuários com as melhores estimativas. O grau de confiança serviu para melhorar a análise dos autores do trabalho. Assim, além de convidar os usuários a participarem (evitando que eles simplesmente preenchessem os parâmetros com qualquer valor) os autores também gostariam de observar se os usuários acreditavam que sabiam o que estavam fazendo.

Mesmo com tais incentivos, os usuários continuaram cometendo grandes erros (a média de utilização do tempo estimado aumentou apenas de 49% para 51%) e muitos

usuários (50%) nem forneceram uma estimativa diferente do tempo máximo da requisição. Um último fator ao qual queremos chamar atenção é o grau de confiança em que os usuários se classificaram. Em uma escala que de 0 a 5, a maior parte (70%) se classificou como 4 ou 5.

Para finalizar a discussão sobre as diferenças entre o tempo requerido e o tempo real de execução das requisições, apresentaremos rapidamente um trabalho que modela o tempo requisitado apresentado por Tsafrir, Etsion e Feitelson [73] em junho de 2005. Os autores deste trabalho verificaram que uma pequena quantidade de valores para o o tempo requisitado (cerca de 20 valores - *"20 top hits"*) é utilizada por cerca de 90% dos usuários. Para modelar o tempo requisitado, os autores observaram que os valores que aparecem com bastante freqüência são os valores máximos permitidos para grupos de usuários. Ou seja, é bastante comum os usuários simplesmente utilizarem o valor máximo como tempo estimado, ao invés de tentarem prover uma boa estimativa para o tempo.

Estes trabalhos mostram que o tempo requisitado se afasta muito do tempo de execução não só porque o (i) tempo requerido é utilizado como um limite máximo para execução e (ii) as vantagens de especificar um tempo melhor (conseguir acesso mais rápido ao recurso) não motivam suficientemente os usuários. Os usuários parecem não se preocupar com este parâmetro e fornecem o valor máximo permitido de maneira que essas diferenças entre os tempos requisitado e utilizado são inerentes às estimativas dos usuários.

2.2.4 Aplicações BoT em recursos compartilhados no espaço

Alguns dos recursos compartilhados no espaço são projetados para prover alta velocidade entre os nós que compõem o recurso (e.g. supercomputadores). Esta velocidade é provida visando promover o desempenho das aplicações paralelas acopladas onde a comunicação pode se tornar o gargalo da execução.

Essa característica pode nos levar a pensar que recursos compartilhados no espaço não executam aplicações BoT. Entretanto temos evidências mostrando o contrário. Primeiro, uma parte considerável das requisições enviadas aos recursos compartilhados no espaço pedem pela menor quantidade de processadores possível: sistemas como SDSC

SP2, SDSC BlueHorizon, CTC SP2 e Los Alamos apresentam no mínimo 35% das requisições com essa característica. Segundo, em [19] encontramos uma pesquisa realizada com usuários do SDSC onde 69,4% dos usuários respondem que suas aplicações não têm restrições com relação à quantidade de processadores que serão requisitados, o que sugere que mais aplicações são BoT. Terceiro, diversos usuários do OurGrid utilizam recursos compartilhados no espaço para executar aplicações BoT (como citado no Capítulo 1). Por fim, nossa experiência mostra que os administradores desses recursos, quando vão prover aos usuários acesso ao recurso, dão mais prioridade à funcionalidade da aplicação (ou ao valor da aplicação) do que a características como a freqüência de comunicação.

Além disto, note que nem todos os recursos compartilhados no espaço apresentam a conectividade de um supercomputador, alguns recursos deste tipo são *clusters* que têm uma rede local "normal" (com velocidade típica encontrada para conectar estações de trabalho) para interconectar os nós.

2.3 Grids computacionais

Grids computacionais surgiram na década de 1990 com a promessa de oferecer serviços computacionais de maneira similar à energia oferecida por nossa rede elétrica (*eletric grid*). Ou seja, tais serviços seriam providos sob demanda e os detalhes seriam transparentes aos clientes (como na rede elétrica).

Para atingir esses objetivos, a infra-estrutura que dá suporte a tais serviços deve prover uma maneira de diversos domínios administrativos dispersos geograficamente colaborarem dinamicamente, através do compartilhamento de seus recursos heterogêneos que estão sob políticas administrativas locais (controle distribuído).

Apesar de todo o esforço feito até o momento [1], a computação em grid está longe de lidar com a complexidade inerente às funcionalidades dessa infra-estrutura e de prover serviços computacionais de maneira similar ao fornecimento de energia feito pela rede elétrica.

[1] O termo *grid computing* retorna mais de 480 mil ocorrências em `http://scholar.google.com/` (Outubro, 2005)

Nosso trabalho se foca na idéia de grids de alto desempenho, responsáveis pelo nascimento da computação em grid. Estes grids sugiram na comunidade de Processamento de Alto Desempenho com o objetivo de agregar diversos processadores espalhados e prover plataformas de execução para aplicações paralelas mais baratas que os recursos tipicamente utilizados para isto até então (e.g. supercomputadores).

Desta maneira, definimos *grids computacionais* como plataformas para a execução de aplicações paralelas em recursos potencialmente heterogêneos e distribuídos por diversos domínios administrativos, oferecendo acesso a esses recursos independente da localização física [18,37,38]. Tais sistemas tornam possível a integração de vários recursos dispersos em um único computador virtual (meta-computador) acelerando a execução de várias aplicações paralelas. Isto possibilitou nos últimos anos a melhoria de desempenho com redução de custo através do compartilhamento de recursos, construindo sistemas em escalas maiores que as anteriores sem a compra de mais equipamentos. Outras definições de grids computacionais podem ser encontradas em [14, 23, 35, 49].

As soluções para grids de alto desempenho tipicamente utilizam um componente que esconde a heterogeneidade das máquinas que serão utilizadas para executar as tarefas no grid. O uso desses componentes simplificam as implementações dos *brokers*. Isto acontece porque a heterogeneidade é reduzida através de uma interface comum provida por tais componentes de maneira que os *brokers* só precisam conhecer uma única interface.

O componente mais conhecido que provê tal função é o Globus Resource Allocation Manager (GRAM). O GRAM foi proposto com o objetivo de ser um módulo responsável pela gerência local de um recurso em uma arquitetura de meta-computador [27]. Ele processa uma requisição RSL (*Resource Specification Language*) que representa um pedido a um recurso, disponibiliza a monitoração e gerência de tarefas e atualiza um serviço de informação sobre o recurso. Neste caso, cabe ao *broker* construir a especificação na linguagem RSL e submeter ao GRAM.

O GRAM é a solução mais usada de submissão de aplicações paralelas para recursos compartilhados no espaço. Existem diversas implementações do componente para fazer com que uma requisição RSL seja convertida em um pedido de alocação para este tipo de recurso. O simples uso desse componente para realizar a submissão de requisições

possibilita que uma aplicação acoplada execute em um recurso compartilhado no espaço mas não garante que partições de diferentes recursos compartilhados no espaço estejam disponíveis ao mesmo tempo de maneira a permitir que uma aplicação execute em recursos distribuídos.

Como solução para possibilitar a execução distribuída, os escalonadores de recursos compartilhados no espaço estão estendendo suas interfaces para prover reservas no futuro (*advance reservation* [36]). Estas reservas têm um momento pré-determinado para iniciar a execução e portanto não são tratadas como as outras requisições. Desta maneira, o responsável pela elaboração da requisição RSL contata serviços que provêem informações sobre a disponibilidade de recursos (e.g. MDS) e elabora um conjunto de requisições com reservas no futuro fazendo com que as partições de diversos recursos possam estar disponíveis ao mesmo tempo. Isto fornece um conjunto de processadores que serão utilizados para a execução de aplicações acopladas a partir de diversas partições.

2.3.1 Grids computacionais para aplicações BoT

Apesar do esforço para prover soluções que viabilizem aplicações fortemente acopladas serem executadas em grids, as aplicações BoT podem ser vistas como as melhores possíveis para grids computacionais onde a comunicação facilmente se torna o gargalo para aplicações fortemente acopladas. Exemplos disto são os projetos projeto United Devices [6] e SETI@home [11]. Outros dois fatores influenciam para a facilidade da execução das aplicações BoT: (i) *firewalls* e endereços privados; e (ii) falhas. Com relação a (i), a presença de *firewalls* e IP privados no grid tornam praticamente inviável a comunicação simétrica entre os processadores que o compõem, complicando a execução de aplicações fortemente acopladas [69]. Com aplicações BoT, não é necessário que todos os nós tenham que se comunicar entre si, é preciso apenas que um nó (responsável por distribuir as tarefas) tenha essa característica.

A facilidade de lidar com falhas deve-se ao fato de que quando a execução de uma tarefa falha devido à indisponibilidade do processador (e.g. máquina reiniciada ou falha na comunicação), uma nova execução pode ser iniciada em um outro processador. Se a aplicação fosse fortemente acoplada e um processador falhasse, teríamos o caso onde

seria necessário reiniciar toda a aplicação.

Tais facilidades aliadas à grande quantidade de aplicações BoT fizeram com que diversas soluções de grids para este tipo de aplicação entrassem em produção e se tornassem casos de sucesso. Como exemplos, podemos citar: United Devices [6], SETI@home [11], Nimrod/G [15], Condor [72] e OurGrid [2, 22].

O projeto Condor é um dos exemplos citados. Ele se focou em aplicações BoT, atingiu uma boa maturidade e hoje é uma das soluções grid mais utilizadas. Condor é o projeto mais antigo de grid, iniciou no fim da década de 1980 com o objetivo de aproveitar os ciclos ociosos de um site [52] e depois evoluiu para que usuários de um site pudessem utilizar recursos de outros sites [30]. Hoje, ele provê uma solução completa para grids: cobrindo desde a submissão do job, o escalonamento, controle da execução até a obtenção dos resultados. Cada máquina roda um conjunto de *daemons* responsáveis por prover funcionalidades similares às do Globus. Apesar deste sucesso, Condor-G [39] surgiu como uma iniciativa do casamento entre as tecnologias dos projetos Globus e Condor. Condor-G tira vantagem dos GRAM já implantados (*deployed*) para execução de tarefas.

Essas soluções têm em comum o uso de *brokers* preparados para utilizar recursos intermitentes. Um *broker* é a porta de entrada do usuário no grid, ele é o responsável por receber a aplicação, coordenar os recursos, executar as tarefas e retornar o resultado da aplicação. Um *recurso intermitente* é uma abstração de um recurso computacional que se torna disponível e indisponível para o grid sem aviso prévio, com poder de processamento desconhecido e variável e que pode não oferecer garantias sobre a computação realizada (resultados incorretos são possíveis). Apesar destas características, bom desempenho e resultados confiáveis ainda podem ser obtidos através de escalonadores gulosos (*eager schedulers*) [40, 56, 59, 62]. Eles são escalonadores que fazem uso de replicação para tolerar a variação de desempenho bem como a credibilidade dos resultados [63].

Recursos de computação voluntária são bons exemplos para este tipo de abstração: quando uma máquina se torna ociosa, é executado um *worker* que se contacta com o *broker* e torna a máquina disponível para execução; quando o usuário dono da máquina volta a utilizá-la, o recurso se torna indisponível. O uso de recursos compartilhados no

tempo (e.g. máquina Linux onde o usuário tem acesso via *secure shell - ssh*) por esses *brokers* é um problema bem resolvido. Em particular, um recurso compartilhado no tempo pode ser visto como um recurso intermitente que está disponível todas as vezes que está ligado. Dessa maneira, utilizar um recurso compartilhado no tempo como um recurso intermitente para as soluções de escalonamento existentes é trivial (o *broker* submete uma tarefa quando detecta que a máquina está ligada).

Uma solução que utiliza recursos compartilhados no espaço para aplicações BoT está sendo desenvolvida pela CPAD/PUCRS. Esta solução utiliza uma abordagem bastante semelhante à computação voluntária onde os recursos ociosos são doados ao grid. O escalonador do recurso executa um *worker* para cada processador ocioso, o *worker* então se anuncia para recurso disponível para o grid. Quando um usuário local precisa utilizar o processador, o escalonador mata o *worker* e qualquer outro processo iniciado para o grid. Note que esta abordagem, assim como as outras apresentadas para aplicações BoT no grid, não oferece garantias sobre disponibilidade dos recursos.

No entanto quando queremos utilizar o recurso compartilhado no espaço como usuário local que tem acesso ao recurso e não quer depender da ociosidade, nos deparamos com um problema. Isto ocorre, pois, mesmo usando componentes como o GRAM para esconder a heterogeneidade dos recursos, recursos compartilhados no espaço têm uma forma diferente de uso onde é necessário especificar quantos processadores serão usados e por quanto tempo.

Smallen et al. em [66] realiza automaticamente esta escolha mas de maneira acoplada à aplicação. Os autores realizam a requisição aos recursos compartilhados no espaço de duas maneiras: (i) imediata, onde é pedido a partição livre que inicia no instante da submissão da aplicação e (ii) estática, onde o pedido era realizado de acordo com uma execução prévia das tarefas (realizada pelos autores e não automatizada pela aplicação) para estimar o tempo de execução em um processador do recurso compartilhado no espaço utilizado e a quantidade de processadores variava em três possibilidades pré-determinadas.

Apesar de alguns trabalhos descreverem o funcionamento do Condor-G, não encontramos nenhum que especificasse como estes parâmetros são escolhidos. O trabalho de Derek Wright [75] é o que mais se aproxima de uma discussão sobre esta escolha

relacionando ao Condor. Derek Wright apresenta uma solução que combina recursos intermitentes e *clusters* que disponibilizam processadores dedicados onde não existe especificação de tempo requerido. Esta questão está presente não só no Condor mas em diversos outros *brokers* para BoT (e.g. [15, 23]) e é o foco do nosso trabalho.

Capítulo 3

Elaborando requisições automaticamente

Este capítulo descreve a solução para conversão dos recursos compartilhados no espaço em recursos intermitentes. Apresentamos o modelo utilizado para representar um grid para aplicações BoT que utiliza recursos compartilhados no espaço. Um dos componentes deste modelo é um adaptador de requisições para cada recurso compartilhado no espaço. Esse adaptador utiliza heurísticas, também descritas neste capítulo, para submeter requisições ao recurso compartilhado no espaço e tornar os processadores desse recurso disponíveis para a execução de tarefas dos jobs do grid.

3.1 Modelo do grid com recurso compartilhado no espaço

O modelo utilizado é apresentado na Figura 3.1. O usuário do grid submete um job ao *broker* que está preparado para lidar com recursos intermitentes solicitados aos provedores de recursos. Um provedor de recursos é o responsável por conseguir o acesso ao recurso e prover ao *broker* processadores prontos para executar tarefas. Dessa forma, cada provedor tenta obter os processadores solicitados, baseados na disponibilidade e em políticas de escalonamento locais, e disponibilizá-los para o *broker*. Quando o *broker* tem um novo processador disponível, é iniciada a execução de uma nova tarefa.

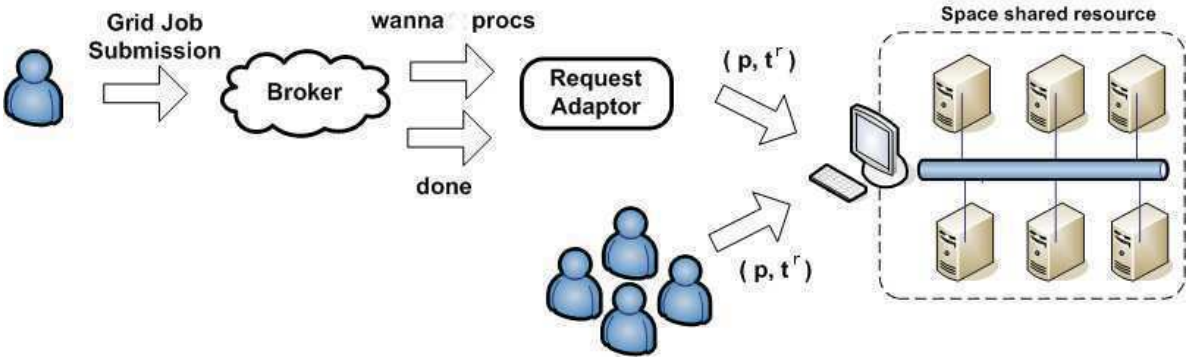


Figura 3.1: Modelo da situação analisada

Se os recursos utilizados são compartilhados no espaço, o processo de obtenção de processadores utiliza nossa solução. Nesse caso, existe um adaptador de requisições para cada recurso compartilhado no espaço. Esse adaptador recebe o pedido do *broker* e, baseado em heurísticas, produz requisições para o recurso compartilhado no espaço. Tipicamente, esperamos que esse adaptador seja um componente do provedor de recursos e por este motivo omitimos o provedor de recursos da Figura 3.1.

As requisições produzidas, além de especificarem o tempo e a quantidade de processadores, também especificam o programa que deve ser executado em cada processador quando a requisição começar a ser atendida. O programa será um *worker* (programa responsável por receber uma tarefa e executá-la) como encontrado em diversas soluções para grid (e.g. [11, 23, 39, 72]). Quando o *worker* inicia a execução, ele se comunica com o *broker* que fica ciente da existência de mais um processador disponível e envia uma tarefa para execução. No final da execução da aplicação, o *broker* informa ao adaptador que não são necessários mais processadores (i.e. que o job do grid acabou).

3.1.1 Notação

Seja $\mathcal{R}$ o conjunto de $|\mathcal{R}|$ requisições submetidas ao recurso compartilhado no espaço (provenientes dos usuários locais do recurso ou do adaptador de requisições). Seja $R_i \in \mathcal{R}$ e $R_i = (t_i^s, p_i, t_i^r, t_i^i, t_i^e)$; onde t_i^s é o instante que a requisição é submetida ao recurso compartilhado no espaço, p_i é a quantidade de processadores requisitados, t_i^r é o tempo pelo qual os processadores devem ser alocados, t_i^i é o instante que a requisição

inicia a utilização dos processadores, t_i^e é o tempo pelo qual os processadores ficaram alocados para a requisição e o instante final de execução de de R_i pode ser obtido por $t_i^i + t_i^e$. No momento da submissão, são conhecidos apenas t_i^s, p_i e t_i^r (sendo p_i e t_i^r os parâmetros fornecidos ao escalonador do recurso); os demais elementos da tupla R_i são conhecidos apenas ao término da requisição. Definimos área de R_i como sendo $p_i \times t_i^r$. O tempo de espera em fila de R_i pode ser obtido por $t_i^i - t_i^s$. $\mathcal{R}_t^w$ é o conjunto de requisições submetidas até o instante t e que ainda não começaram a utilizar os processadores (requisições que estão em espera na fila).

Seja o job do grid $G = (u_g^s, u_g^f, \mathcal{T})$, onde $\mathcal{T}$ é um conjunto de tarefas do grid, u_g^s é o instante de submissão do job G e u_g^f é o instante final de execução de G (não conhecido no instante u_g^s). Seja $T_k \in \mathcal{T}$ e $T_k = (v_k^i, v_k^e)$, onde v_k^i é o instante que a tarefa inicia a utilização do processador, v_k^e é o tempo de execução da tarefa (ambos não são conhecidos na submissão do job G). O instante do fim da execução de T_k é obtido por $v_k^i + v_k^e$. O instante do fim de execução de G é dado por $u_g^f = \left(max_{T_k \in \mathcal{T}}(v_k^i + v_k^e)\right) + u_g^s$. $\mathcal{T}_t^p$ é o conjunto de tarefas do grid que estão pendentes (ainda não tiveram a execução completada) no instante t, onde $\forall t, \mathcal{T}_t^p \subset \mathcal{T}$. G é executado através do conjunto requisições RG para o recurso compartilhado no espaço escolhidas pelo adaptador de requisições, onde $RG \subset \mathcal{R}$. Finalmente, $T(R_i)$ é o conjunto de tarefas executadas pela requisição R_i, onde $\forall R_i, T(R_i) \subset \mathcal{T}$.

3.2 Informações necessárias para adaptação de requisições

Para a escolha dos parâmetros das requisições (p, t^r) que devem ser submetidos ao escalonador do recurso compartilhado no espaço, o adaptador de requisições precisa obter algumas informações sobre o job G do grid, sobre o estado do recurso compartilhado no espaço e sobre as políticas administrativas que impõem restrições às requisições. Assumimos as seguintes informações sobre o recurso compartilhado no espaço e sobre suas restrições estão disponíveis:

1. A quantidade máxima de requisições pendentes para o adaptador de requisições

($maxPR$).

2. A quantidade máxima permitida de processadores por requisição ($maxP$).

3. O tempo máximo permitido para cada requisição ($maxT^r$).

4. O estado da fila de requisições. i.e. o conjunto $\mathcal{R}_t^w$ onde de cada requisição $R_i \in \mathcal{R}_t^w$ só temos (p_i, t_i^s e t_i^r) conhecidos.

As informações dos itens 1, 2 e 3 são estáticas e definidas pela política administrativa do recurso compartilhado no espaço. Assim, elas podem ser fornecidas ao adaptador de requisições no momento da instalação desse módulo. Outra alternativa para obter essas informações é o adaptador de requisições realizar pedidos incrementais até obter um erro, estabelecendo assim o limite. O estado da fila (item 4) tem valor dinâmico e deve ser obtido usando um comando ou serviço oferecido pelo sistema.

As informações que serão necessárias sobre o job do grid G são: (i) quantidade $|\mathcal{T}|$ de tarefas do job G e (ii) tempo estimado $E(\mathcal{T})$ para a execução de uma tarefa do job G. Definimos $E(\mathcal{T})$ como o tempo de execução estimado para qualquer tarefa $T_k \in \mathcal{T}$. A quantidade de tarefas é conhecida no momento da submissão do job e, portanto, uma informação fácil de ser obtida. O tempo estimado para a execução da tarefa é complicado de ser obtido do usuário ou do *broker*. Assim, a heurística infere essa informação durante a execução do job (como veremos adiante).

3.3 Heurísticas para elaboração de requisições ao recurso compartilhado no espaço

O adaptador de requisições deve escolher o conjunto RG de forma a minimizar u_g^f. A escolha pela conjunto ótimo é impossível de ser realizada pois depende (i) do comportamento das requisições submetidas ao recurso compartilhado no espaço ($\mathcal{R}$ - que só é conhecido no final da execução das requisições) e (ii) do tempo de execução das tarefas de G ($v_k^e, \forall T_k \in \mathcal{T}$ - que só é conhecido ao final da execução de G). Ou seja, o adaptador de requisições precisaria conhecer (i) e (ii) no instante u_g^s (o que implica em conhecer o futuro) para realizar a escolha do conjunto RG ótimo.

Devido a essa impossibilidade, o adaptador de requisições realiza a escolha baseado em heurísticas. Ele executa uma heurística para elaborar requisições no *instante da submissão de G* (u_g^s - quando o *broker* requisita processadores). Essa heurística continua a ser executada *no instante que uma nova requisição pode ser submetida* ao recurso compartilhado no espaço (i.e. quando uma requisição anterior já foi atendida) até que não existam mais tarefas a executar. Definimos esses instantes como *momentos de tomada de decisão*. Assim, em um instante t, quando é possível submeter uma (ou mais) requisições, uma heurística escolhe o conjunto $\mathcal{M}_t$ de requisições para execução do conjunto $\mathcal{T}_t^p$ de tarefas do job do grid G. A Equação 3.1 mostra como o conjunto RG é formado. Definimos também $\mathcal{M}_t^p$ como o conjunto das requisições pendentes (já submetidas, mas que não finalizaram a execução) no instante t.

$$RG = \bigcup_{t=u_g^s}^{u_g^f} \mathcal{M}_t \tag{3.1}$$

O processo de elaboração de heurísticas para produzir as requisições começou com uma solução simplória já utilizada pelos usuários do OurGrid: a heurística estática (ver Seção 3.3.1). Baseados nessa heurística, criamos uma nova (heurística inicial - ver Seção 3.3.2) que não obteve resultados melhores que a heurística estática. Apesar disso, a análise da heurística inicial e da heurística estática nos deu base para a criação da heurística adaptativa (ver Seção 3.3.3) que é a nossa solução final. Essa seção apresenta estas heurísticas bem como os fatores que nos levaram à criação de cada uma.

3.3.1 Heurística Estática

Uma solução simplória para realizar a requisição é especificar os valores máximos permitidos ao usuário para ambos os parâmetros (p, t^r). Essa solução era utilizada pelos usuários do OurGrid. Realizamos uma pequena modificação nela e apresentamos seu funcionamento no Algoritmo 3.1. Note que t é o instante do momento de tomada de decisão no qual a heurística está sendo executada.

A pequena melhoria realizada em relação à solução de nossos usuários é na escolha de p. O número de processadores nem sempre é o máximo pois a heurística verifica a quantidade $|\mathcal{T}_t^p|$ de tarefas pendentes e divide pela quantidade de requisições que podem

```
for i = 1 to (maxPR − |M_t^p|) do
    p ← min(maxP, ⌈|T_t^p|/(maxPR − |M_t^p|)⌉);
    submeteRequisicao(p, maxT^r);
end
```

Algoritmo 3.1: Heurística Estática

ser efetuadas $(maxPR - |\mathcal{M}_t^p|)$. O valor de p será o menor valor entre o resultado dessa divisão e $maxP$. Uma vez que p tem que ser um valor inteiro (não podemos pedir $0,5$ processador, por exemplo), nós utilizamos $\lceil |\mathcal{T}_t^p|/(maxPR - |\mathcal{M}_t^p|) \rceil$.

Se a quantidade de tarefas de um job for grande ($|\mathcal{T}| > maxP \times maxPR$), várias requisições novas podem ser submetidas quando as antigas acabarem, de maneira que não se exceda a quantidade de requisições pendentes permitidas. Ou seja, $\forall t, |\mathcal{M}_t^p| \leqslant maxPR$. Todavia, essa heurística não precisa obter o estado da fila do recurso compartilhado no espaço.

Essa solução freqüentemente gera a maior área possível para uma requisição pois freqüentemente solicita $p = maxP$ e sempre usa $t^r = maxT^r$. Isso pode provocar mais tempo de espera em fila (ver Seção 2.2.1) e, conseqüentemente, aumento de u_g^f.

3.3.2 Heurística Inicial

Devido ao impacto da área da requisição no tempo de espera em fila, nós acreditávamos que um bom valor para o parâmetro t^r seria o tempo necessário para executar uma tarefa (v_k^e). Tal valor é o menor possível (reduzindo a área e conseqüentemente o tempo de espera em fila) capaz de realizar processamento que produza algum resultado. uma forma de especificar o tempo seria pedir ao usuário para informar quanto tempo deve ser requisitado ao escalonador do recurso (ou seja, quanto tempo é necessário para executar uma tarefa).

No entanto, isso dificultaria o uso do grid uma vez que os usuários teriam que conhecer a duração de suas aplicações em todos os recursos compartilhados no espaço disponíveis no grid. Não optamos por essa abordagem pois acreditamos que os *brokers* para aplicações BoT obtêm grande sucesso devido à facilidade de uso.

Outro fator é que acreditamos tornar inviável pedir essa informação ao usuário é

que, como mostrado na Seção 2.2.3, essa estimativa é feita com grandes erros mesmo para recursos homogêneos como supercomputadores. Ou seja, se a estimativa já é difícil para supercomputadores com poder de processamento bem conhecido, como o usuário especificaria para o grid que é composto por recursos tão heterogêneos?

Estimando o tempo da tarefa

A solução desenvolvida inicialmente foi uma heurística para estimar o tempo a ser requisitado t^r. Essa heurística estima o valor do tempo a ser requisitado t^r de acordo com as execuções anteriores das tarefas visando aproximar t^r do tempo de execução de uma tarefa (v_k^e).

A estimativa $E(\mathcal{T})$ é realizada de acordo com o Algoritmo 3.2 que é executada nos momentos de tomada de decisão (representados por now). Caso seja a primeira estimativa ($now = u_g^s$), o algoritmo utiliza um tempo padrão pré-definido ($DEFAULT\_TIME$) como valor para $E(\mathcal{T})$.

if $now = u_g^s$ **then**
| $E(\mathcal{T}) \leftarrow DEFAULT\_TIME$
else
| $est_1 \leftarrow 1,1 \times max_{T_k \in (\mathcal{T} - \mathcal{T}_{now}^p)}(v_k^e)$
| $est_2 \leftarrow 2 \times max_{R_i \in \bigcup_{t=u_g^s}^{now} \mathcal{M}_t} \left(t_i^e - \sum_{T_k \in T(R_i)} v_k^e \right)$
| $E(\mathcal{T}) \leftarrow max(est_1, est_2)$
end

Algoritmo 3.2: Estimativa de tempo da heurística inicial

Em uma situação na qual o tempo requisitado t^r foi suficiente para executar uma tarefa ($t^r > v_k^e$), a próxima solicitação deverá pedir t^r um pouco maior (usamos 10% maior) que o maior tempo de execução de uma tarefa de G entre as tarefas executadas até o momento (ver est_1 no Algoritmo 3.2). O tempo é um pouco maior para obter uma pequena margem de segurança para a execução das próximas tarefas.

Nosso objetivo é tornar o tempo requisitado próximo do tempo de execução da tarefa. Note que as tarefas podem variar no tempo de execução, por isso a heurística toma as decisões baseada no tempo de execução da maior tarefa até o momento. Se o tempo requisitado não foi suficiente para executar alguma tarefa (ver est_2 no Algoritmo

3.2), o valor de tempo a ser requisitado na próxima tentativa será multiplicado por um fator inteiro (2). O Algoritmo 3.2 apresenta $E(\mathcal{T})$. Ele é baseado no controle de congestionamento do TCP [46] para essas estimativas: em boas situações ser cuidadoso (multiplicação por $1,1$) e em situações ruins tomar decisões abruptas (multiplicação por 2).

Estabelecendo o número de processadores

A diminuição do parâmetro p não causa a impossibilidade de término da tarefa como acontece com o parâmetro t^r. Esse fato nos leva a pensar que a melhor forma de diminuir a área é dividir o número de processadores solicitados pelo *broker* em várias pequenas requisições (cada uma de até *um* processador) o que implicaria em um menor tempo em fila [25].

Todavia, a maioria dos escalonadores de recursos compartilhados no espaço (através de políticas administrativas) impõe limites ao número de requisições pendentes que o usuário pode ter no sistema ($maxPR$). A premissa é impedir uma "monopolização" do recurso. Tais políticas fazem com que a especificação do número de processadores p não seja uma tarefa tão fácil, pois não podemos simplesmente pedir várias requisições de um processador.

Dessa maneira, a heurística que inicialmente desenvolvemos para a conversão não poderia apenas gerar várias requisições com $p = 1$. Levamos em consideração a carga do recurso (tamanho da fila de requisições) antes de realizar a requisição. A idéia é quanto maior a carga, menor o valor da quantidade de processadores por requisição pois menores são os espaços livres na fila. Assim, elaboramos várias versões de uma função que recebe a carga do recurso e retorna a quantidade de processadores a ser pedida que retornava $maxP$ para uma fila sem requisições em espera e 1 para uma fila de tamanho infinito (valores intermediários eram retornados quando a fila não estava vazia). A Equação 3.2 mostra um exemplo de função utilizada.

$$maxP \cdot \frac{1}{|\mathcal{R}_t^w|} \tag{3.2}$$

Nossa esperança era que essa abordagem poderia não oferecer imediatamente ao *broker* todos os recursos de que ele necessita, mas possibilitaria a obtenção mais rápida

dos recursos e portanto esperávamos que o tempo de execução da aplicação diminuísse.

Refinando a requisição

Podemos usar o estado da fila de requisições para determinar novas requisições possíveis que começam imediatamente. Com essa informação, a heurística dá forma à primeira requisição realizada pela heurística (quando ainda não existe conhecimento sobre execuções anteriores) assim como sabe a maior quantidade de processadores que pode ser requisitada sem que exista espera em fila.

Note que há a possibilidade de nenhum processador estar disponível. Nesse caso, a requisição é feita com base em um tempo padrão pré-definido. Outra vantagem da utilização desse comando do sistema é a possibilidade de pedir um tempo maior que o estimado sem que isso implique em maior tempo de espera, o que diminue a possibilidade do tempo não ser suficiente para uma tarefa terminar. Apesar de considerarmos que toda a fila está disponível (como mostrado na Seção 3.2), quando elaboramos essa heurística trabalhamos com a possibilidade de utilizar apenas as novas requisições possíveis que começam imediatamente.

Comportamento da heurística inicial

Essa heurística foi implementada e apresentou alguns resultados preliminares bons [24]. Todavia, quando continuamos a analisar o desempenho da heurística verificamos que o desempenho não era melhor em comparação à heurística estática na grande maioria de casos. A diferença da análise preliminar e a realizada posteriormente foi os tipos de jobs que levamos em consideração. A heurística inicial obtinha bons resultados para jobs bem pequenos, mas não para jobs maiores. A Seção 4.2 mostra os casos que consideramos para a análise final.

A principal dificuldade encontrada nessa heurística foi elaborar uma função que provesse um bom mapeamento entre a carga do recurso e a quantidade de processadores. Durante o processo de analisar os resultados e melhorar a heurística, fizemos várias versões da função. Os resultados melhoravam apenas para uma determinada categoria de jobs e não obtínhamos o sucesso global esperado.

3.3.3 Heurística Adaptativa

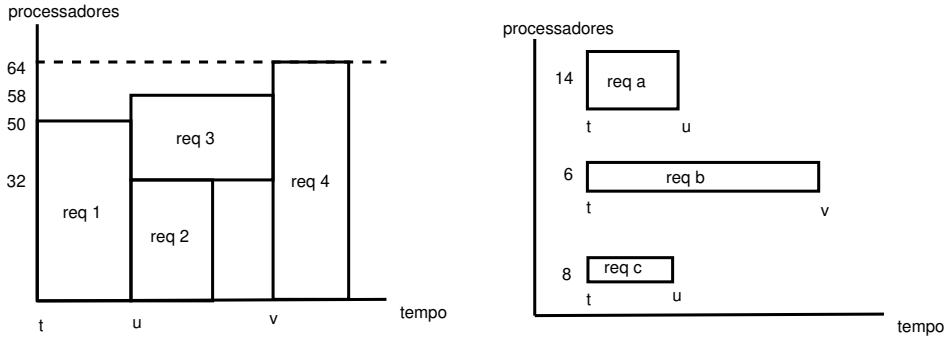
Apesar dos resultados fracos obtidos inicialmente, o processo de analisá-los nos fez perceber as qualidades da heurística estática. Ela provê uma boa vazão de tarefas fazendo com que o job do grid termine mais rápido. Suas requisições demoram a ganhar os processadores mas, quando ganham, executam uma grande quantidade de trabalho.

Dessa maneira, mudamos a abordagem de tentar prover a menor área possível capaz de executar uma tarefa. Nosso objetivo passou a ser produzir requisições que provessem a melhor vazão de tarefas possível. Para isso, precisávamos ainda saber o tempo de execução da tarefa. Como essa informação só está disponível no fim de execução da tarefa, utilizamos o mesmo mecanismo utilizado pela heurística inicial para estimar o tempo de execução, mas não utilizamos a função para estabelecer a quantidade de processadores. A Equação 3.3 mostra como obter a vazão para um conjunto de requisições $\mathcal{M}_t$ baseado em $E(\mathcal{T})$ (ver Algoritmo 3.2).

$$V(\mathcal{M}_t, E(\mathcal{T})) = \sum_{R_m \in \mathcal{M}_t} \frac{\frac{t_m^r}{E(\mathcal{T})} \cdot p_m}{max_{R_m \in \mathcal{M}_t}(t_m^i + t_m^r)} \tag{3.3}$$

Essa nova heurística tornou-se nossa solução final e a chamamos de heurística adaptativa. Essa heurística constrói o conjunto de requisições $\mathcal{M}_t$ explorando o conjunto requisições possíveis em um instante t ($\mathcal{P}_t$), usando $V(\mathcal{M}_t, E(\mathcal{T}))$ como critério de escolha. De maneira semelhante à nossa heurística inicial, o desenvolvimento da heurística adaptativa passou por várias etapas que resultaram em algumas versões até obtermos a versão final. Uma das nossas principais dificuldades para implementar a heurística adaptativa foi a função para obter um conjunto $\mathcal{P}_t$ para ser utilizado na heurística. Isso ocorreu porque um dado instante pode oferecer várias possibilidades de requisições mutuamente excludentes. A Figura 3.2 apresenta um exemplo onde existem várias possibilidades de requisições mutuamente excludentes. O estado da fila é exibido na Figura 3.2(a), se a heurística escolher pela requisição a ela não pode optar nem pela b nem pela c (ver Figura 3.2(b)).

Para formar o conjunto $\mathcal{P}_t$ escolhemos as requisições que preenchem os espaço livres na fila de espera. Quando os espaços livres provêem requisições mutuamente excluden-



(a) Estado da fila

(b) Algumas requisições possíveis que começam no instante t

Figura 3.2: Exemplo de uma fila com escolha de requisições mutuamente excludentes

tes, escolhemos a de melhor vazão. Além dessas requisições, o conjunto $\mathcal{P}_t$ tem $maxPR$ requisições com parâmetros $p = maxP$ e $t^r = maxT^r$ para representar as requisições possíveis que iniciam depois do fim da fila de espera pelo recurso. Essa abordagem limita $|\mathcal{P}_t|$. Essa medida é necessária pois se fosse considerada todas as requisições possíveis em um instante t, $|\mathcal{P}_t| = \infty$.

O Algoritmo 3.3 mostra o funcionamento da heurística adaptativa que é executada em todo momento de tomada de decisão. O funcionamento da heurística é varrer a fila de espera do recurso compartilhado no espaço, verificar um conjunto $\mathcal{P}_t$ de requisições possíveis e escolher dentre elas o conjunto $\mathcal{M}_t$ que proveja a melhor vazão. Acreditamos que o problema de escolher as melhores requisições seja um problema NP-Completo, apesar de não termos conseguido provar isso. Assim, nossa estratégia para escolher as requisições foi uma estratégia gulosa: escolher as $maxPR$ primeiras requisições possíveis como o conjunto candidato, escolher uma nova requisição possível, verificar se a troca de alguma das requisições do conjunto escolhido por essa nova requisição melhora a vazão do conjunto, se melhorar realizar a troca, continuar a varredura até o fim de $\mathcal{P}_t$.

Além do funcionamento já descrito, note que a heurística pode diminuir a quantidade de processadores (p_m) baseada no número de tarefas pendentes antes de realizar a submissão das requisições (ver fr no final do Algoritmo 3.3). Essa melhoria visa diminuir a quantidade de processadores das requisições caso o conjunto escolhido pro-

$\mathcal{M}_t \leftarrow \varnothing$

for $m = 1$ **to** $(maxPR - |\mathcal{M}_t^p|)$ **do**
| $\mathcal{M}_t \leftarrow \mathcal{M}_t \cup P_m$

end

foreach $P_m \in \mathcal{P}_t$ **do**
| **foreach** $M_m \in \mathcal{M}_t$ **do**
| | $\mathcal{M}_t^{aux} \leftarrow \mathcal{M}_t - M_m \cup P_m$
| | **if** $V(\mathcal{M}_t, E(\mathcal{T})) < V(\mathcal{M}_t^{aux}, E(\mathcal{T}))$ **then**
| | | $\mathcal{M}_t \leftarrow \mathcal{M}_t^{aux}$
| | **end**
| **end**

end

$fr \leftarrow \left(|\mathcal{T}_t^p| / \sum_{m=1}^{|\mathcal{M}_t|} p_m \right)$

if $fr < 1$ **then**
| **foreach** $M_m \in \mathcal{M}_t$ **do**
| | $p_m \leftarrow \lceil fr \cdot p_m \rceil$
| **end**

end

$submetaRequisicoes(\mathcal{M}_t)$

Algoritmo 3.3: Heurística Adaptativa

veja um número maior de processadores do que de tarefas pendentes. A diminuição tende a evitar duas situações: (i) processadores alocados e não utilizados e (ii) áreas de requisições maiores que provocam maior espera em fila.

3.3.4 Heurística Mista

Mesmo com a evolução da nossa solução para a heurística adaptativa ainda tivemos casos onde a heurística estática teve um desempenho melhor (ver Seção 4.4). Este foi o motivo de elaborar mais um tipo de heurística que chamamos de heurística mista. Ela combina o funcionamento das heurísticas estática e adaptativa da seguinte forma: a primeira requisição é submetida como se o algoritmo fosse o da heurística estática (i.e. requisição de máxima área).

Assim, quando o momento de tomada de decisão é u_g^s a escolha do conjunto $\mathcal{M}_t$ é mostrado pelo Algoritmo 3.4. Os demais momentos de decisão executa a heurística adaptativa já apresentada. O efeito da combinação dessas características foram resultados intermediários (ver Seção 4.7).

3.4 Desperdício do recurso

Todas as heurísticas têm a desvantagem de poder apresentar desperdício de recurso, onde a partição do recurso está alocada mas ociosa. Isto acontece em duas situações: (i) uma requisição se encerra antes do término da tarefa (não produzindo nenhum resultado) e (ii) existem menos tarefas pendentes do que processadores atualmente alocados.

O caso (i) acontece porque o tempo requisitado não casa exatamente com a soma dos tempos das tarefas que serão executadas. Nesse caso, o *broker* considera que os processadores ficaram indisponíveis e submete novamente as tarefas abortadas para outros processadores.

O caso (ii) pode ocorrer quando o *broker* já executou várias tarefas. Uma requisição submetida quando existiam mais tarefas pendentes ganha os processadores, de maneira que ela provê uma quantidade de processadores maior que a necessária deixando alguns processadores alocados porém ociosos.

$\mathcal{M}_t \leftarrow \varnothing$

for $m = 1$ **to** $(maxPR - |\mathcal{M}_t^p|) - 1$ **do**
| $\mathcal{M}_t \leftarrow \mathcal{M}_t \cup P_m$

end

foreach $P_m \in \mathcal{P}_t$ **do**
| **foreach** $M_m \in \mathcal{M}_t$ **do**
| | $\mathcal{M}_t^{aux} \leftarrow \mathcal{M}_t - M_m \cup P_m$
| | **if** $V(\mathcal{M}_t, E(\mathcal{T})) < V(\mathcal{M}_t^{aux}, E(\mathcal{T}))$ **then**
| | | $\mathcal{M}_t \leftarrow \mathcal{M}_t^{aux}$
| | **end**
| **end**

end

$fr \leftarrow \left(|\mathcal{T}_t^p| / \sum_{m=1}^{|\mathcal{M}_t|} p_m \right)$

if $fr < 1$ **then**
| **foreach** $M_m \in \mathcal{M}_t$ **do**
| | $p_m \leftarrow \lceil fr \cdot p_m \rceil$
| **end**

end

$\mathcal{M}_t \leftarrow \mathcal{M}_t \cup \{(maxP, maxT^r)\}$

$submetaRequisicoes(\mathcal{M}_t)$

Algoritmo 3.4: Heurística Mista para o momento de tomada de decisão u_g^s.

A Equação 3.4 mostra como obter o tempo de processamento desperdiçado (TD) por um job do grid. Apesar do foco de nosso trabalho ser diminuir o tempo de execução do job do grid (diminuir u_g^f), TD também é uma métrica analisada nesta dissertação (ver Seção 4.6).

$$TD = \sum_{R_i \in RG} (p_i \cdot t_i^e) - \sum_{T_k \in \mathcal{T}} v_k^e \tag{3.4}$$

Capítulo 4

Avaliação de Desempenho

Para avaliar o desempenho da nossa solução, comparamos o comportamento das heurísticas desenvolvidas tendo foco nas heurísticas adaptativa e estática. A avaliação é comparativa e utiliza as seguintes métricas: (i) tempo de execução do job do grid ($u_g^f - u_g^s$) e (ii) tempo de processamento desperdiçado (TD - ver Seção 3.4). Desenvolvemos um simulador para analisar o comportamento das heurísticas. Preferimos a técnica de simulação pois ela permite (i) uma resolução mais fácil do modelo do que se resolvêssemos o modelo matematicamente (o que nem sempre é possível), (ii) cobrir um espaço de possibilidades maior, permitindo a análise de vários cenários, e (iii) mais controle dos fatores que afetam a execução do que se fosse utilizada a implementação real da solução.

4.1 Grid simulado

O modelo utilizado é apresentado na Figura 3.1 e descrito na Seção 3.1. Vamos aqui recapitular o funcionamento e apresentar alguns detalhes adicionais sobre o escalonamento que utilizamos para simular o grid. O usuário do grid submete um job ao *broker* que está preparado para lidar com recursos intermitentes solicitados aos provedores de recursos. Cada provedor tenta conseguir os processadores solicitados baseado na disponibilidade e em políticas de escalonamento locais. À medida que o *broker* recebe novos processadores (i.e. é contactado por novos *workers*), ele inicia a execução das tarefas.

O algoritmo de escalonamento utilizado pelo *broker* é conhecido como *workqueue*: para cada processador livre uma tarefa ($T_k \in \mathcal{T}$) é escolhida aleatoriamente e enviada para execução nesse processador. O processo de submissão de tarefas se repete toda vez que um processador acaba de executar uma tarefa (i.e. fica livre novamente) até que não existam mais tarefas disponíveis para execução (i.e. $|\mathcal{T}_t^p| = 0$). O escalonamento das tarefas pode ser refinado visando melhorar o desempenho das aplicações ou a utilização do sistema [40, 56, 59, 62]; no entanto este não é o nosso foco e, portanto, não tratamos aqui com algoritmos diferentes de *workqueue* a nível de *broker*.

O provedor de recursos tem um adaptador de requisições para cada *cluster* ou supercomputador. Este adaptador recebe o pedido do *broker* e produz requisições para o recurso compartilhado no espaço baseado nas heurísticas descritas no Capítulo 3. Essas requisições especificam o tempo t^r e a quantidade de processadores p a serem alocados. O algoritmo utilizado para escalonar as requisições no recurso compartilhado no espaço é o *conservative backfilling* (descrito na Seção 2.2.1) . Quando a requisição ganha o recurso, o *broker* é informado que existem mais processadores disponíveis. A partir desse momento, o funcionamento do escalonador do *broker* é o já descrito.

4.2 *Workload*

Para alimentar o simulador precisamos de *workload* descrevendo os jobs. Note pela Figura 3.1 que existem dois tipos de fontes de jobs: (i) do usuário do grid que são submetidos ao *broker* e (ii) dos usuários locais dos recursos compartilhados no espaço (aqui tratados apenas como requisições ou o conjunto $\mathcal{R}$). *Clusters* e supercomputadores são sistemas em produção há mais tempo do que grids e, portanto, já é possível encontrar *traces* de tais sistemas disponíveis na *web* de maneira que analisamos uma *workload* real. Outra vantagem de utilizar tais *traces* é o fato de que já foram estudados em diversos trabalhos de pesquisa o que pode ajudar a análise dos resultados. A desvantagem é que avaliar o cenário para uma carga específica pois diferentes *workloads* podem causar grandes diferenças no desempenho dos escalonadores [31]. Assim, decidimos analisar o comportamento com alguns *traces* reais, amenizando a desvantagem citada. O repositório [3] nos serviu como fonte para *workloads* de recursos compartilhados no

espaço, do qual utilizamos os *traces* apresentados na Tabela 4.1.

Trace	**Sistema**	**Quantidade de processadores**	**Quantidade de requisições**	**Período**
SDSC SP2	San Diego Supercomputer Center SP2	128	73496	Abril/1998 a Dezembro/2000
SDSC BlueHorizon	San Diego Supercomputer Center BlueHorizon	1152	250440	Abril/2000 a Dezembro/2000
CTC SP2	Cornell Theory Center SP2	512	79302	Julho/1996 a Julho/1997

Tabela 4.1: Traces utilizados

A disponibilidade de *workloads* para grids não é a mesma. Não temos acesso a *traces* de sistemas em produção nem bibliografia que descreva *workloads* sintéticas para grids em produção. Nesse aspecto, parte da produção científica utiliza uma parcela ou todo um *trace* de supercomputador (e.g. [29, 61]). Todavia, estes *traces* não fornecem algumas informações necessárias para nossa avaliação. Isto ocorre porque parte das características da aplicação não estão disponíveis para o escalonador (responsável por registrar o *trace*). Desse modo, não utilizamos tal abordagem pois não teríamos como obter algumas das informações necessárias para a simulação: (i) quantidade de tarefas $|\mathcal{T}|$ e (ii) tempo de execução de cada tarefa $v_k^e, \forall T_k \in \mathcal{T}$.

Nossa abordagem foi utilizar uma *workload* sintética. Visando amenizar o problema de não ter dados reais, fizemos uma ampla varredura de parâmetros, gerando vários cenários variando a quantidade de tarefas por job ($|\mathcal{T}|$), o tempo de execução (ou tamanho) médio de uma tarefa e a heterogeneidade das tarefas de um job. A heterogeneidade de um job é dada pela variação do tempo de execução de suas tarefas.

Variamos o tempo médio de execução em 100, 1000 e 10000 segundos; a quantidade das tarefas em 100, 1000, 10000 e 100000; e a heterogeneidade em 1x, 2x e 4x. Utilizamos a distribuição uniforme - U(valor mínimo, valor máximo) - para variar a heterogeneidade. A heterogeneidade de 1x significa que o job é homogêneo (i.e. todas as suas tarefas têm o mesmo tempo de execução). Heterogeneidade 2x apresenta tarefas de acordo com U(média/2, 3média/2) e 4x com U(média/4, 7média/4).

Definimos um *tipo de job* como o conjunto de jobs que apresentam o mesmo tempo

médio entre as tarefas, a mesma quantidade de tarefas e a mesma heterogeneidade. Por exemplo, um job G com $|\mathcal{T}| = 100$, com duração média de 100 segundos e heterogeneidade 1x (100,100s,1x) é um job homogêneo composto de 100 tarefas de 100 segundos cada uma ($\forall T_k \in \mathcal{T}, v_k^e = 100s$). A Tabela 4.2 resume os parâmetros utilizados para gerar os jobs do grid, produzindo 36 combinações possíveis. Assim, executamos simulações para 36 tipos de jobs diferentes.

Heterogeneidade	1x U(média,média), 2x U(média/2, 3média/2) e 4x U(média/4, 7média/4)
Tempo médio da tarefa	100 segundos, 1000 segundos e 10000 segundos
Quantidade de tarefas	100, 1000, 10000 e 100000

Tabela 4.2: Valores possíveis para cada parâmetro de um job.

4.3 Simulações do grid

4.3.1 Descrição dos cenários estudados

Nosso principal objetivo é avaliar o tempo de execução (*turn-around time* ou *makespan*) dos jobs do grid, i.e. $u_g^f - u_g^s$. Cada rodada do simulador tem apenas um job do grid como entrada, de maneira que nenhum job do grid influencia outro. As simulações consistiram submeter um job do grid a um *broker* que tinha acesso, através de um adaptador de requisições, a um recurso compartilhado no espaço. Esse recurso processa um dos *traces* (o conjunto $\mathcal{R}$) descritos na Tabela 4.1. Note que utilizamos apenas um recurso compartilhado no espaço por execução do simulador e não vários, apesar do nosso trabalho ter como motivação um grid. Isso ocorre porque apenas um recurso desse tipo já é suficiente para avaliarmos se existe ganho com a utilização da heurística adaptativa e de quanto ele é. A adição de mais recursos tornaria o ambiente mais complexo desnecessariamente, uma vez que não há .interferência de um recurso compartilhado no espaço em outro.

Além das *workloads*, tivemos que fornecer também a quantidade de requisições que um usuário poderia ter pendente no sistema ($maxPR$ - parâmetro necessário para as heurísticas, descrito na seção 3.2). Variamos $maxPR$ entre 1 e 6 (maior valor encontrado nos sistemas reais consultados - SDSC BlueHorizon [7]). Assim, um cenário é definido por: (i) tipo do job do grid, (ii) valor de $maxPR$, (iii) *workload* de recurso compartilhado no espaço (um conjunto $\mathcal{R}$) e (iv) a heurística para elaboração de requisições. Desta maneira, a quantidade de cenários possíveis é: 36 tipos de jobs $\times$ 6 valores para $maxPR$ $\times$ 3 *workloads* de recursos compartilhados no espaço $\times$ 3 heurísticas (estática, adaptativa e mista) = 1944. Cada execução do cenário submete um job G do grid em um instante diferente (gerado aleatoriamente) mas dentro do intervalo delimitado pela *workload* do supercomputador. Portanto, u_g^s muda e $min_{R_i \in \mathcal{R}}(t_r^s) \leqslant u_g^s \leqslant max_{R_i \in \mathcal{R}}(t_i^i + t_i^e)$.

Foram executadas 100 simulações para cada cenário. Ou seja, 100 execuções do simulador cada uma com um job G do grid diferente. Utilizando um nível de confiança de 95%, essa quantidade de simulações nos garantiu um erro menor do que 5% para a média do tempo de execução para a maioria dos casos (utilizamos o procedimento descrito em [47] para este cálculo). Apenas os jobs menores ($|\mathcal{T}| = 100$ ou $|\mathcal{T}| = 1000$ tarefas com tempo médio de execução inferior a 10000s) tiveram um erro maior pois devem uma parte maior do seu tempo de execução à espera em fila que é bastante variável. Para estes casos, executamos mais simulações até obter um erro menor que 5%. Nesses casos, cada tipo de job tem um conjunto de 1000 jobs do grid associados a ele.

Para executar as mais de 194.400 simulações com resultado analisados nesta dissertação (1944 cenários $\times$ pelo menos 100 simulações por cenário), utilizamos o OurGrid com recursos espalhados por cerca de 20 diferentes sites. Vale salientar que a quantidade de simulações realmente executadas foi 701.425, pois tivemos várias heurísticas até chegarmos às apresentadas nesta dissertação, além das execuções falhas por problemas diversos (e.g. arquivos de configuração com erros).

4.4 Tempo de execução do grid job

O tempo de execução de um job G do grid é dado por $u_g^f - u_g^s$. A análise comparativa das heurísticas verifica a relação de $u_g^f - u_g^s$ obtido com a heurística estática e o obtido com a heurística adaptativa. Vale salientar que a mudança de um parâmetro apenas em um cenário na provoca nenhuma mudança nos outros. Inclusive, quando o tipo de job é o mesmo o conjunto de jobs utilizados também é o mesmo se o cenário muda, por exemplo, apenas pela heurística aplicada, os jobs do grid são os mesmos.

Para descrever a análise, definimos $\mathcal{G}$ como o conjunto de $|\mathcal{G}|$ jobs do grid submetidos ao *broker* (mas sempre um job em cada execução). Definimos também $TA(G)$ como $u_g^f - u_g^s$ obtido pelo job G com a heurística adaptativa e $TE(G)$ como $u_g^f - u_g^s$ obtido pelo job G com a heurística estática. A análise é baseada no ganho de desempenho (*speedup* médio) da heurística adaptativa em relação à heurística estática. A Equação 4.1 mostra como obter o *speedup* médio.

$$\frac{\sum_{G\in\mathcal{G}} TE(G)}{|\mathcal{G}|} \div \frac{\sum_{G\in\mathcal{G}} TA(G)}{|\mathcal{G}|} = \frac{\sum_{G\in\mathcal{G}} TE(G)}{\sum_{G\in\mathcal{G}} TA(G)} \tag{4.1}$$

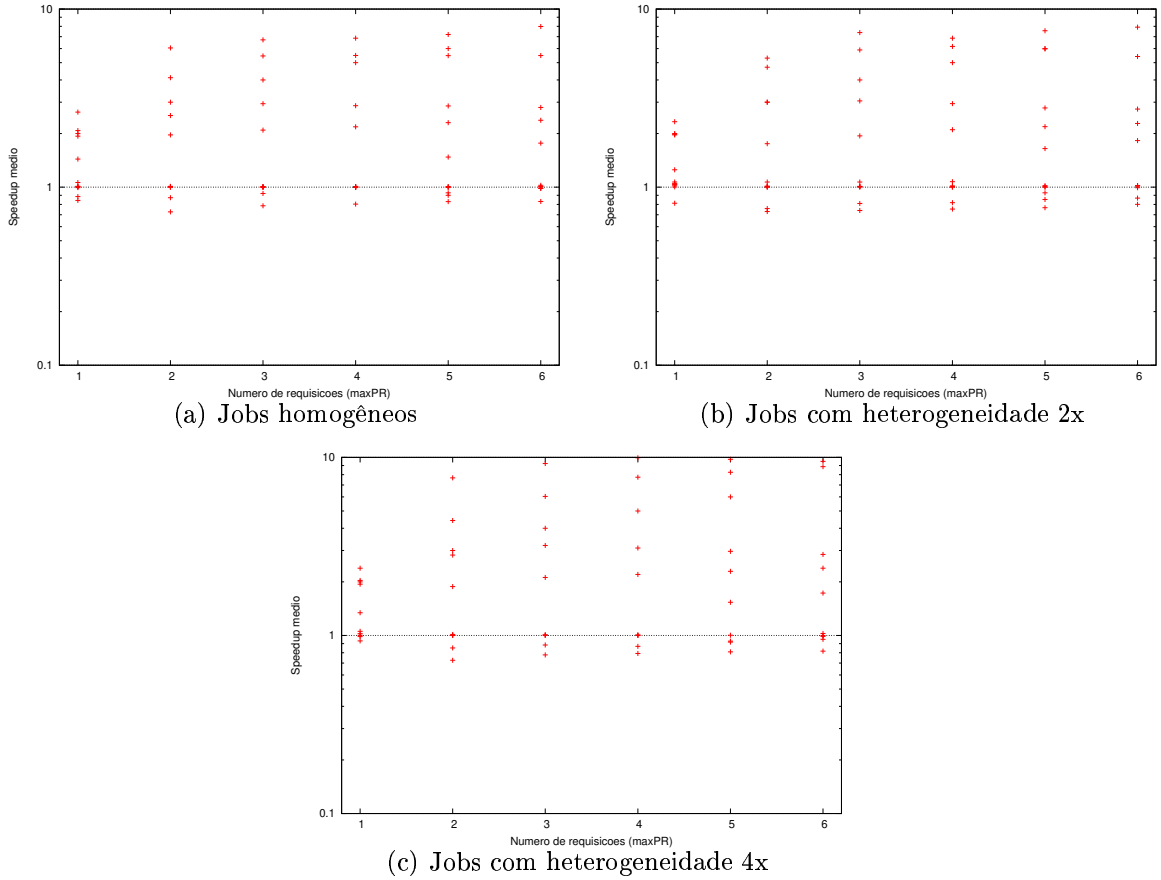
A heurística adaptativa obteve melhor resultado do que a heurística estática na maioria dos casos. A maior diferença entre os resultados foi com a mudança da *workload* e portanto apresentamos os resultados divididos por *workload*. A Tabela 4.3 apresenta um resumo dos resultados exibindo *speedup* médio que a heurística adaptativa obteve sobre a heurística estática para cada *workload*. O restante dessa seção faz uma análise mais detalhada de cada *workload* e de cada tipo de job.

Workload	*Speedup médio*
SDSC SP2	2,05 ± 5%
SDSC BlueHorizon	2,37± 5%
CTC SP2	14,74± 5%

Tabela 4.3: *Speedup* médio da heurística adaptativa sobre a estática.

Para evitar uma explosão na quantidade dos gráficos devido à quantidade de jobs diferentes gerados a partir da combinação de parâmetros da Tabela 4.2, cada ponto

representa o *speedup* médio para um determinado tipo de job. Por exemplo, um ponto com $speedup = 2$ significa que $\frac{\sum_{G \in \mathcal{G}} TA(G)}{|\mathcal{G}|}$ foi a metade de $\frac{\sum_{G \in \mathcal{G}} TE(G)}{|\mathcal{G}|}$.



(a) Jobs homogêneos

(b) Jobs com heterogeneidade 2x

(c) Jobs com heterogeneidade 4x

Figura 4.1: Comparação do desempenho dos jobs utilizando as heurística estática e adaptativa para a workload SDSC SP2

4.4.1 SDSC SP2

A Figura 4.1 apresenta os resultados obtidos quando utilizamos a *workload* SDSC SP2. Os gráficos estão agrupados por heterogeneidade do job do grid. O eixo x apresenta a variação de $maxPR$, o eixo y o *speedup* médio e cada ponto do gráfico é um cenário. Na maior parte dos casos, a heurística adaptativa tem desempenho melhor, especialmente com jobs pequenos (100 ou 1000 tarefas com duração média de 100 ou 1000 segundos). A heterogeneidade que aplicamos nos cenários analisados não

produziu diferenças significativas de desempenho (em média, menor que 2%) e por este motivo suprimimos em alguns casos o detalhamento por heterogeneidade.

Jobs com grande duração média da tarefa (10000s) e poucas tarefas ($|\mathcal{T}| = 100$ ou $|\mathcal{T}| = 1000$) são desafios para a heurística adaptativa. A Figura 4.2 mostra o *speedup* médio para esses casos. Esses jobs apresentam resultados melhores com a heurística estática pois as requisições dela fazem com que pedidos "úteis" (capazes de executar pelo menos uma tarefa) sejam iniciados antes do que os "úteis" da heurística adaptativa. A heurística adaptativa gasta algumas requisições no processo para estimar o tempo da tarefa antes de realizar pedidos que possam oferecer uma boa vazão. Isso implica no tempo do processo de elaboração das requisições: gerar uma requisição, esperar na fila, esperar o tempo de execução, fazer uma nova requisição baseada na última e repetir este ciclo quantas vezes for preciso. Dessa maneira, o tempo gasto nesse processo é maior que o tempo necessário para as requisições geradas pela estática serem atendidas e terminarem a execução.

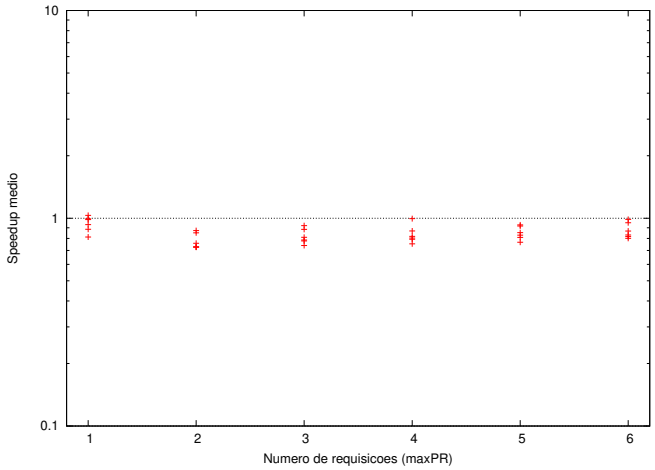


Figura 4.2: Speedup para jobs com poucas tarefas e grande duração média de tarefas (todas as heterogeneidades).

Outro fator que tem grande influência neste comportamento é o tempo t^r requerido pelos usuários do supercomputador que é tipicamente muito maior que o necessário (t_i^e) como já citado neste trabalho (ver Seção 2.2.3). A heurística adaptativa estima um conjunto de requisições que oferece uma vazão melhor que o conjunto oferecido pela heurística estática mas a fila irá abrir espaços diferentes dos previstos (uma vez

que utilizamos o tempo requisitado com o tempo que a requisição deve durar). Assim, mesmo calculada a vazão das requisições a heurística não conhece o que vai acontecer no futuro e pode oferecer um conjunto de requisições que proveja uma vazão menor do que o oferecido pela estática.

Jobs maiores ($|\mathcal{T}| = 10000$ ou $|\mathcal{T}| = 100000$ e tarefas com tempo de execução de 1000s ou 10000s em média) apresentam tempos de execução similares quando usamos a heurística estática e quando usamos a heurística adaptativa (observe na Figura 4.3 que o *speedup* para esses jobs fica em torno de 1). Nesses casos, a heurística estática produz requisições com boa vazão de tarefas em relação às requisições iniciais da adaptativa, mas o tempo necessário para a solução adaptativa estimar o tempo de uma tarefa é pequeno em comparação com o tempo de execução do job. Depois desse tempo de estimativa, as requisições geradas são quase idênticas pois a quantidade de trabalho a ser realizado é tão grande que vale a pena esperar na fila por uma quantidade maior de processadores e por mais tempo.

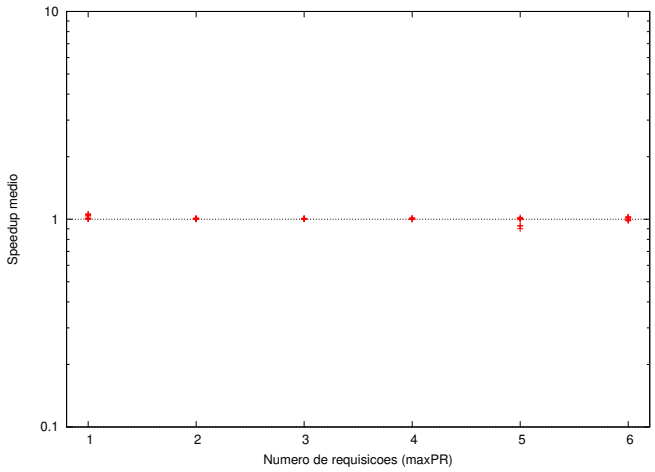
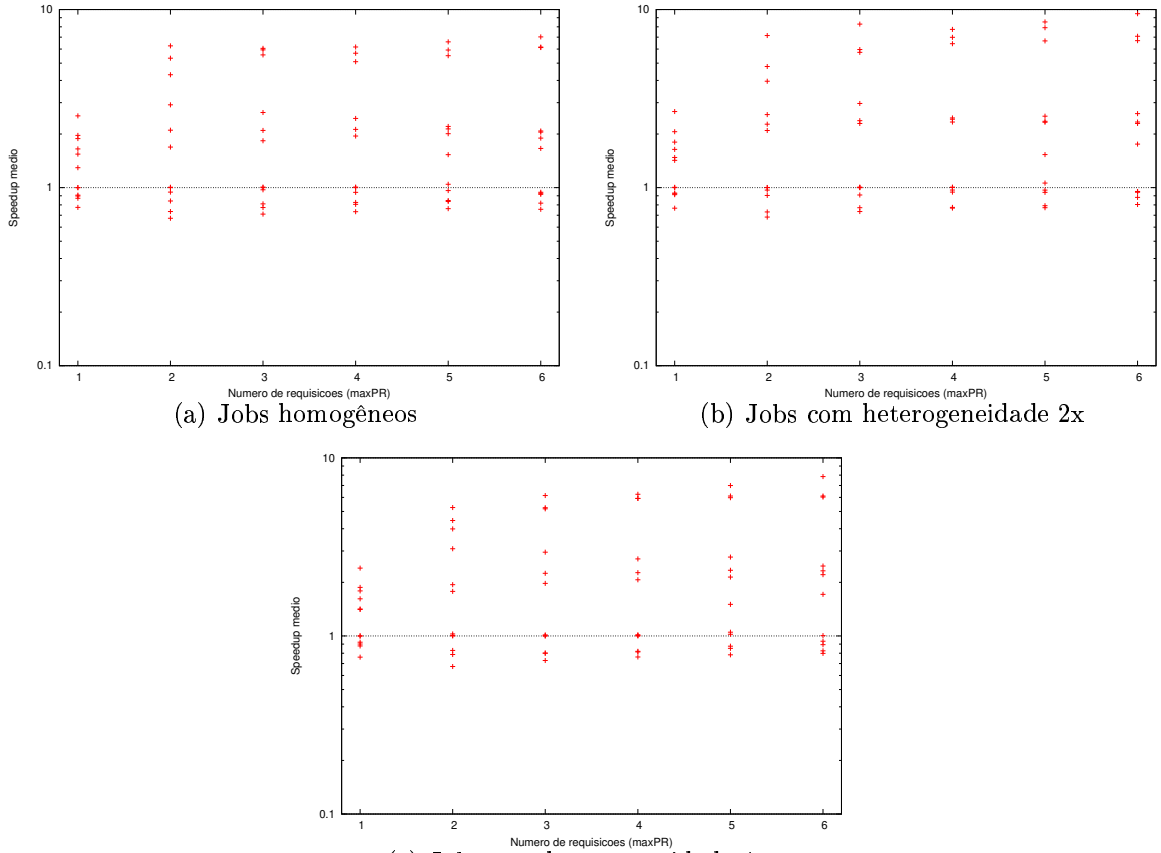


Figura 4.3: Jobs Maiores (todas as heterogeneidades).

4.4.2 SDSC BlueHorizon e CTC SP2

A Figura 4.4 apresenta os resultados obtidos quando utilizamos a *workload* SDSC BlueHorizon. Esses resultados são similares aos resultados obtidos com a *workload* SDSC SP2 tanto nos resultados gerais quanto no comportamento dos diversos tipos de jobs. Também não apresentam grandes diferenças com a heterogeneidade aplicada.



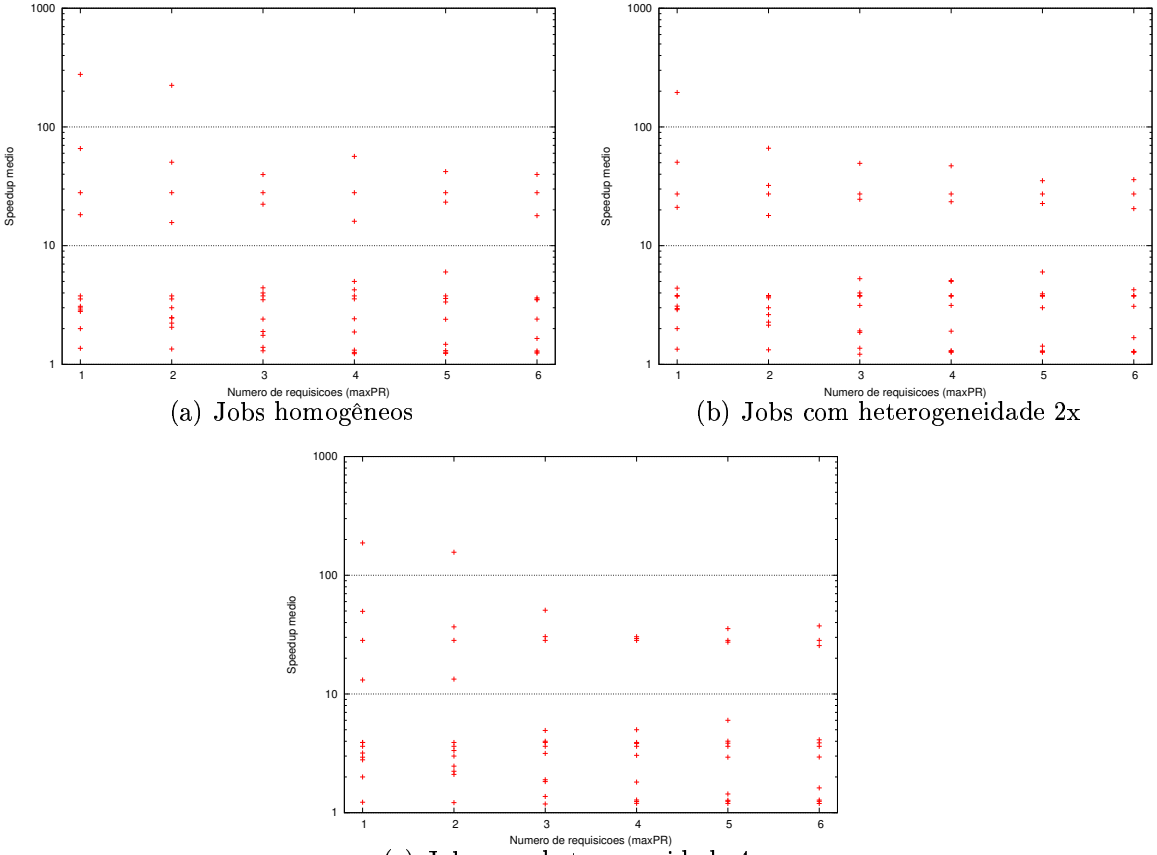
(a) Jobs homogêneos

(b) Jobs com heterogeneidade 2x

(c) Jobs com heterogeneidade 4x

Figura 4.4: Comparação do desempenho dos jobs utilizando as heurísticas estática e adaptativa para a *workload* BlueHorizon

A Figura 4.5 mostra os resultados para CTC SP2. Podemos observar que a heurística adaptativa provê um melhor tempo de execução para todos os casos e que, assim como com SDSC SP2 e SDSC BlueHorizon, a heterogeneidade não apresenta grandes impactos no desempenho.



(a) Jobs homogêneos

(b) Jobs com heterogeneidade 2x

(c) Jobs com heterogeneidade 4x

Figura 4.5: Comparação do desempenho dos jobs utilizando as heurísticas estática e adaptativa para a *workload* CTC

A principal diferença entre as *workloads* é a carga por processador que em CTC SP2 é bem menor do que em SDSC SP2 e em SDSC BlueHorizon provocando mais oportunidades das requisições geradas pela heurística adaptativa conseguirem processadores imediatamente enquanto as estáticas tendem a ir para o fim da fila (devido à sua grande área). A Tabela 4.4 mostra a carga oferecida para cada recurso baseado nas workloads utilizadas. O cálculo da carga oferecida é apresentado na Equação 4.2, onde *procs* é a quantidade de processadores do recurso.

$$\frac{\sum_{R_i \in \mathcal{R}} p_i \cdot t_i^e}{procs \cdot \left(max_{R_i \in \mathcal{R}} (t_i^i + t_i^e) - min_{R_i \in \mathcal{R}} t_i^s \right)} \tag{4.2}$$

Workload	**Carga oferecida**	*Speedup*
SDSC SP2	72%	2,05
SDSC BlueHorizon	73%	2,37
CTC SP2	55%	14,74

Tabela 4.4: Carga oferecida ao recurso e *speedup* obtido pela heurística adaptativa sobre a heurística estática

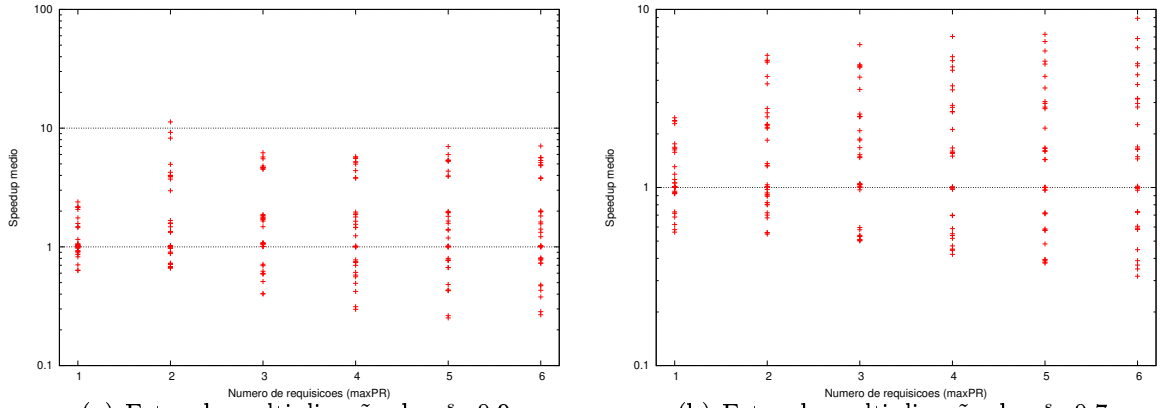
4.5 Impacto da carga

Com base nessa hipótese sobre a alta carga do recurso, resolvemos analisar a situação através do aumento sintético da carga para a *workload* de CTC SP2. Assim, reduzimos o instante de submissão das requisições (t_i^s) dos jobs e mantivemos os demais parâmetros sem alterações (t_i^r, t_i^s e p_i). Isto nos fornece uma situação onde um determinado número de requisições chega em um período de tempo menor que o observado na realidade. Por exemplo, quando multiplicamos t_i^s por $0,7$ é como se submetêssemos os jobs recebidos por 10 meses em um período de 7 meses.

A Figura 4.6 mostra o ganho desempenho da heurística adaptativa sobre a heurística estática para tempos de t_i^s multiplicado por $0,9$ (Figura 4.6(a)) e por $0,7$ (Figura 4.6(b)). A Tabela 4.5 mostra a carga oferecida ao recurso após o aumento sintético. Como os resultados com a variação de heterogeneidade que temos são semelhantes, os valores agregam resultados de todos os jobs. Podemos perceber a diferença que o aumento da carga provocou quando comparamos com os resultados da Figura 4.5 que apresenta as simulações para a *workload* de CTC SP2 sem alterações de carga. Esse cenário tem ganhos de até 200 vezes e nenhum caso onde a heurística adaptativa tenha desempenho inferior ao da heurística estática (ver Figura 4.5), enquanto os casos com aumento da carga apresentam menores ganhos assim como situações onde a heurística estática é melhor.

Workload	Fator para a submissão	Carga oferecida	*Speedup*
CTC SP2	1	55%	14,74
CTC SP2	0,9	61%	1,99
CTC SP2	0,7	78%	1,83

Tabela 4.5: Carga oferecida ao recurso e *speedup* obtido pela heurística adaptativa sobre a heurística estática



(a) Fator de multiplicação de u_g^s: 0,9 (b) Fator de multiplicação de u_g^s: 0,7

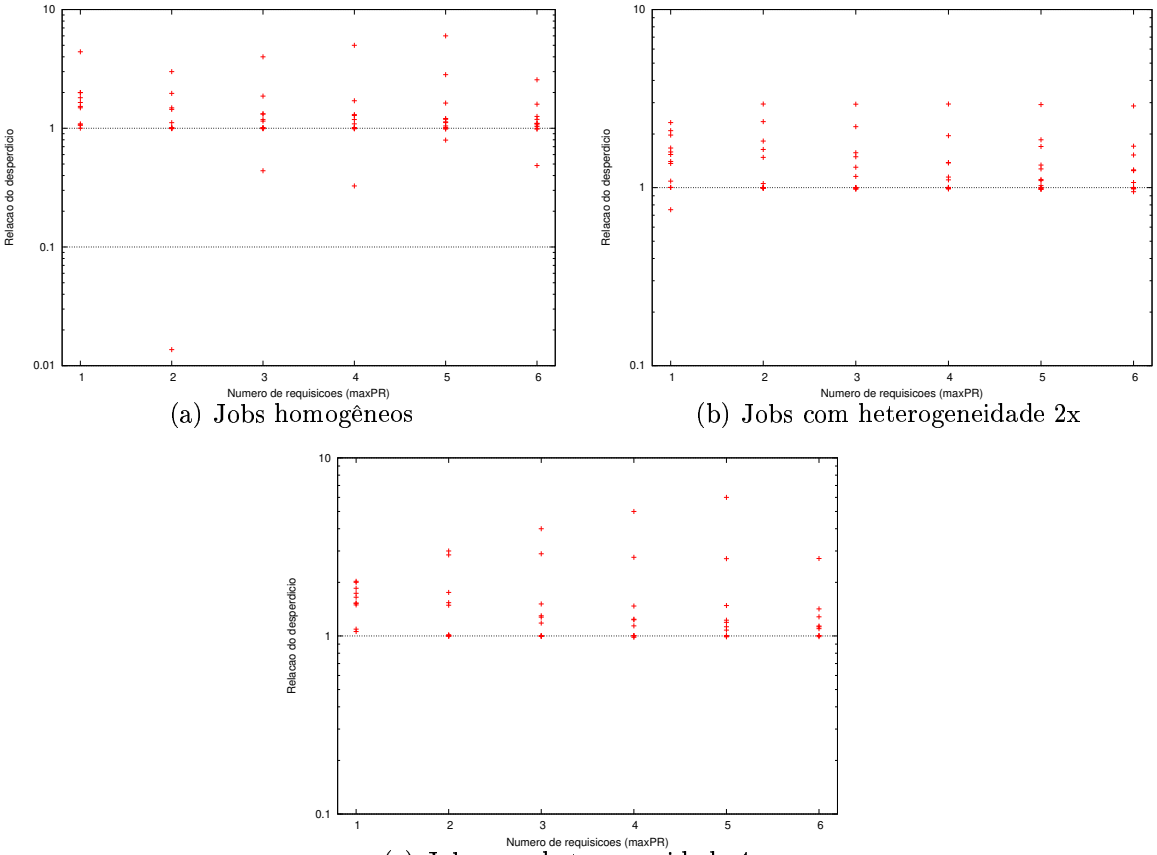
Figura 4.6: Comparação do desempenho dos jobs utilizando as heurísticas estática e adaptativa para a *workload* CTC com aumento sintético de carga

4.6 Desperdício do recurso

Como já citado (ver Seção 3.4), as heurísticas apresentam desperdício do recurso. Para a análise do desperdício, consideramos a relação de desperdício das heurísticas que é tempo de processamento desperdiçado pela estática dividido pelo tempo de processamento desperdiçado pela adaptativa.

As Figuras 4.7, 4.8 e 4.9 apresentam a razão do desperdício das heurísticas. Semelhante ao cálculo da relação do *speedup*, valores acima de 1 significam que adaptativa foi melhor e valores abaixo de 1 significam que estática foi melhor.

As Figuras 4.7 e 4.8 apresentam os resultados para SDSC SP2 e SDSC BlueHorizon, onde na maioria dos casos o tempo desperdiçado pela heurística adaptativa é menor que o tempo desperdiçado pela heurística estática. Esta situação não ocorre para CTC

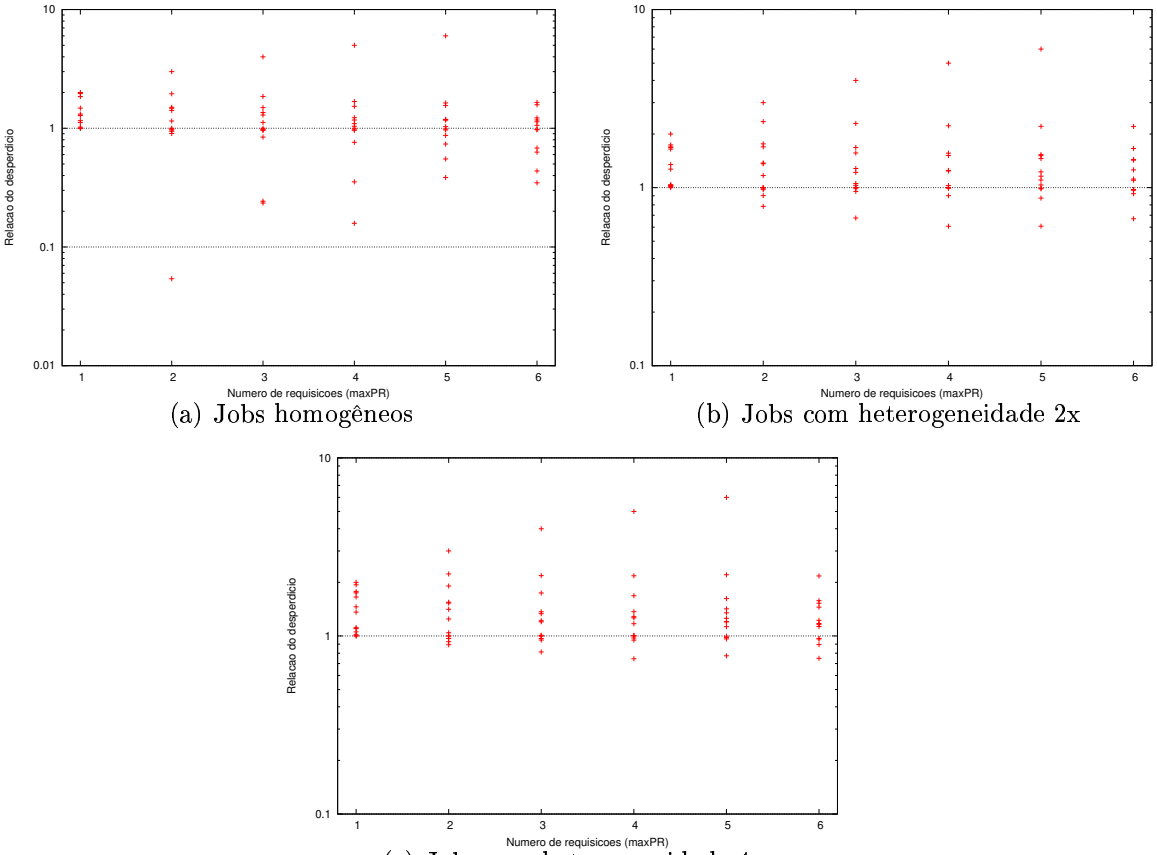


(a) Jobs homogêneos

(b) Jobs com heterogeneidade 2x

(c) Jobs com heterogeneidade 4x

Figura 4.7: Comparação dos ciclos desperdiçados pelos jobs utilizando as heurísticas estática e adaptativa para a *workload* SDSC SP2

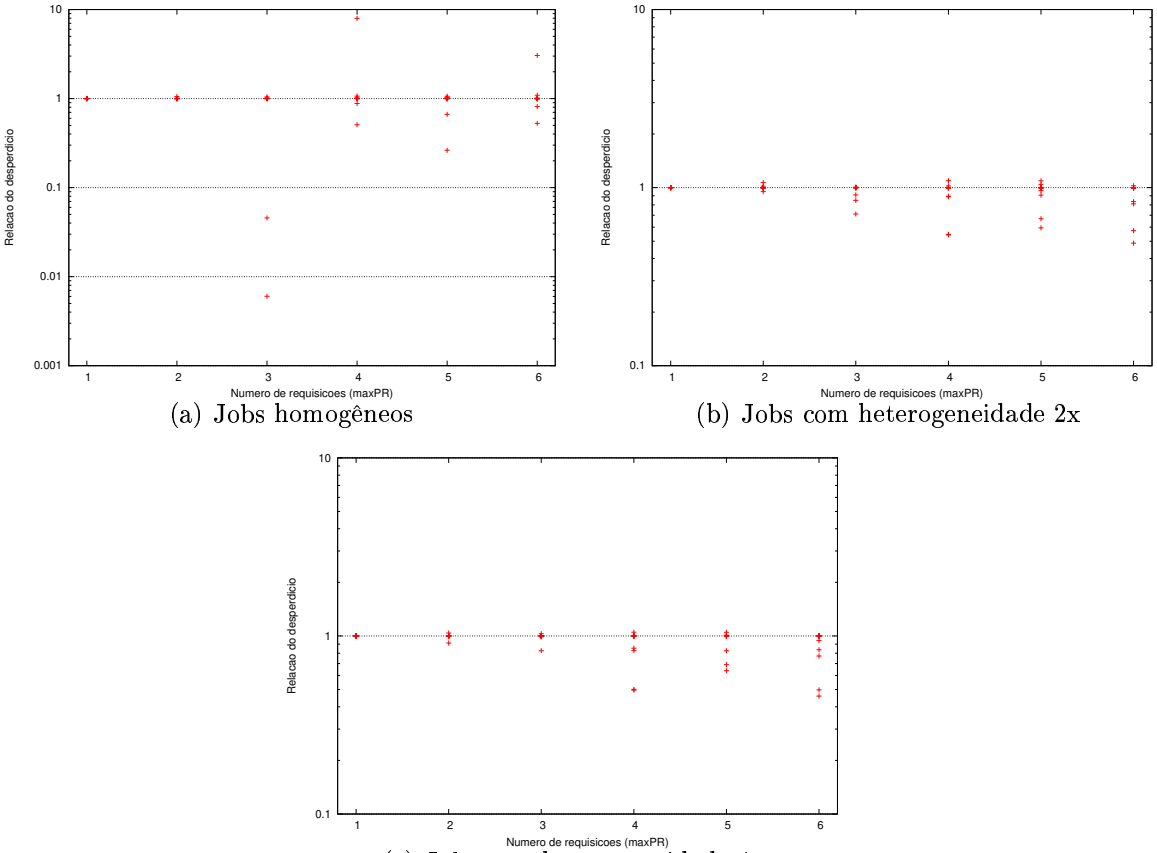


(a) Jobs homogêneos

(b) Jobs com heterogeneidade 2x

(c) Jobs com heterogeneidade 4x

Figura 4.8: Comparação dos ciclos desperdiçados pelos jobs utilizando as heurísticas estática e adaptativa para a *workload* SDSC BlueHorizon



(a) Jobs homogêneos

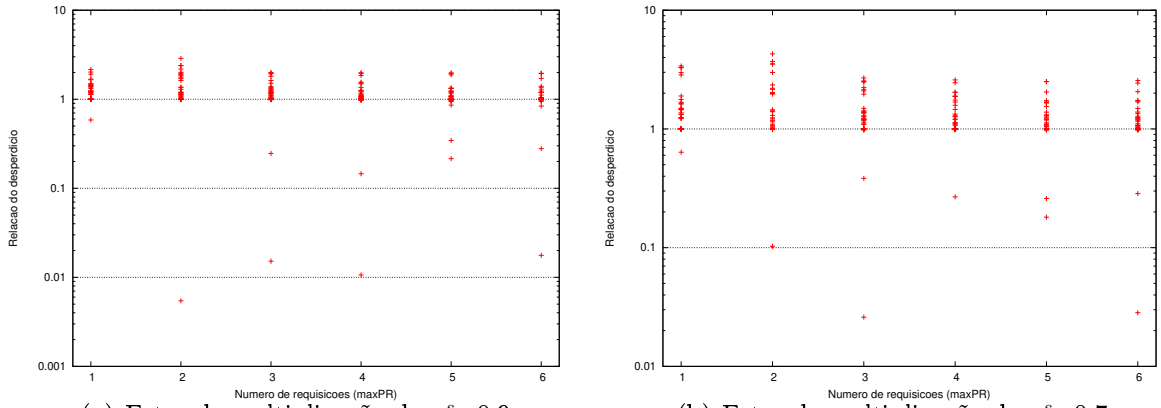
(b) Jobs com heterogeneidade 2x

(c) Jobs com heterogeneidade 4x

Figura 4.9: Comparação dos ciclos desperdiçados pelos jobs utilizando as heurísticas estática e adaptativa para a *workload* CTC SP2

SP2 (ver Figura 4.9) onde na maioria dos casos é a estática que desperdiça menos. A diferença da carga e da quantidade de processadores entre os três sistemas faz com que a heurística adaptativa consiga uma quantidade de processadores maior em CTC (pois a carga faz com que existam requisições que possam ganhar os processadores antes e que os espaços livres sejam maiores), implicando em um maior desperdício.

Para analisar o impacto da carga oferecida ao recurso, verificamos o desperdício para as simulações apresentadas na Seção 4.5 onde aumentamos sinteticamente a carga oferecida da workload CTC SP2. A Figura 4.10 mostra a relação do desperdício para estes casos. Podemos observar que com o aumento da carga oferecida, a heurística adaptativa aumenta a desperdício do recurso quando comparada à heurística estática.



(a) Fator de multiplicação de u_g^s: 0,9 (b) Fator de multiplicação de u_g^s: 0,7

Figura 4.10: Comparação do desperdício do recurso dos jobs utilizando as heurísticas estática e adaptativa para a workload CTC com aumento sintético da carga oferecida

4.7 Heurística Mista

Por fim, apresentamos os resultados para a heurística mista (ver Seção 3.3.4). A Tabela 4.6 mostra os resultados do *speedup* médio para as heurísticas adaptativa e mista usando a heurística estática como referência. Podemos observar que o *speedup* médio da mista é menor que o obtido pela heurística adaptativa. O efeito da combinação das heurísticas não foi intermediário apenas no resumo dos resultados apresentados na Tabela 4.6. Obtivemos resultados intermediários em todos os jobs. Tais jobs finalizam com um

tempo de execução entre o tempo obtido pela heurística estática e o tempo obtido pela heurística adaptativa em todas as simulações. Definindo, $TM(G)$ como $u_g^f - u_g^s$ obtido pelo job G com a heurística mista, temos $TA(G) \leqslant TE(G) \Rightarrow TA(G) \leqslant TM(G) \leqslant TE(G)$ e $TE(G) \leqslant TA(G) \Rightarrow TE(G) \leqslant TM(G) \leqslant TA(G)$.

Dessa maneira, a heurística mista não provê os resultados que esperávamos. Ela apenas ameniza a desvantagem da heurística adaptativa nos casos onde a heurística estática tem desempenho melhor e vice-versa.

Os melhores resultados da heurística mista foram para a *workload* CTC SP2. A única que não apresentava situações nas quais a heurística adaptativa tivesse desempenho pior que a heurística estática. No entanto, os resultados são iguais aos obtidos com a heurística adaptativa para uma grande parte das requisições. Isto ocorre porque a baixa carga oferecida possibilita que a primeira requisição da heurística adaptativa consiga um espaço livre de grande área e que começa no instante da submissão para a maioria dos casos, submetendo uma primeira requisição semelhante à primeira requisição da heurística mista.

Workload	*Speedup* heurística adaptativa	*Speedup* heurística mista
SDSC SP2	2,05	1,37
SDSC BlueHorizon	2,37	1,59
CTC SP2	14,74	14,73

Tabela 4.6: *Speedup* médio das heurísticas adaptativa e mista sobre a heurística estática para todos os jobs.

4.8 Validação do simulador

Apesar das vantagens apresentadas no início deste capítulo que nos fizeram optar por simulações, as simulações podem apresentar falhas que vão desde erros no código do programa utilizado para simulação até falhas no modelo utilizado para a implementação. Com o objetivo de aumentar a confiança no nosso simulador utilizamos: (i) testes de unidade para o desenvolvimento, (ii) testes de regressão baseados em resultados

nossos como também em resultados de outro simulador para recursos compartilhados no espaço (utilizado em [17, 20, 21, 54]), (iii) comparação com um *trace* de um sistema real e (iv) comparação os resultados de alguns cenários pequenos com uma implementação real da solução que será descrita posteriormente (ver Capítulo 5). Apesar do nosso simulador ser mais abrangente do que um simulador exclusivo para recursos compartilhados no espaço, nosso objetivo com os itens (ii) e (iii) foi aumentar a confiabilidade com relação ao comportamento dos recursos compartilhados no espaço nas nossas simulações.

O comportamento das requisições submetidas a outro simulador para recursos compartilhados no espaço (item ii) apresentou resultados similares aos obtidos com nosso simulador. Utilizamos as mesmas *workloads* de recursos compartilhados no espaço descritas na seção 4.2 e uma outra *workload* do Los Alamos National Laboratory (LANL - um sistema CM5 com 1024 processadores onde avaliamos 122233 requisições entre Dezembro de 1999 e Abril de 2000). A Tabela 4.7 mostra um resumo das diferenças encontradas. Nesses casos, dizemos que uma requisição é diferente de outra se a execução de um simulador apresentou t_i^i diferente da execução do outro simulador para uma mesma requisição R_i como entrada. Não existem diferenças para duas das *workloads* analisadas (SDSC SP2 e CTC SP2). As outras duas *workloads* (LANL e SDSC BlueHorizon) apresentaram pequenas diferenças. Analisando as diferenças, percebemos que elas acontecem quando existem requisições submetidas em um mesmo "instante" t_i^s para o recurso (a precisão dos *traces* é de segundos logo requisições submetidas com diferença menores que um segundo podem ser registradas como submetidas no mesmo "instante"). Essas requisições são tratadas de maneira diferente entre os simuladores provocando diferenças nestas requisições e em algumas outras (se a carga estiver alta, a reação é em cadeia).

Com relação ao item (iii). O *trace* de sistema real utilizado para aumentar a confiança no nosso simulador foi obtido no CPAD/PUCRS (um sistema com 32 processadores onde avaliamos 3303 requisições entre Março de 2005 e Setembro de 2005). A comparação dos resultados obtidos pelo nosso simulador e os dados contidos no trace apresentaram diferenças para 2,9% das requisições.

Realizamos a mesma análise descrita acima e verificamos que as diferenças se devem

Workload	Porcentagem de requisições com resultados diferentes
SDSC SP2	0%
CTC SP2	0%
LANL	0,12 %
SDSC BlueHorizon	2,3%

Tabela 4.7: Porcentagem de requisições com resultados diferentes realizando a comparação entre dois simuladores

não só às requisições submetidas no mesmo "instante" t_i^s (como ocorre para LANL e SDSC BlueHorizon). Atribuímos as outras diferenças observadas ao tempo necessário para realizar operações do sistema (e.g. processamento do algoritmo de escalonamento, E/S para registro das operações) e que não são levadas em consideração na implementação do simulador.

O item (iv) é discutido no próximo capítulo.

Capítulo 5

Implementação da solução para conversão de recursos

Com o intuito de validarmos nossa solução, realizamos a implementação dela no MyGrid (um *broker* para aplicações BoT). Este capítulo descreve o MyGrid, como ele passou a lidar com recursos compartilhados no espaço, discute alguns aspectos da implementação e realiza uma pequena análise dos resultados obtidos com esta implementação. O principal objetivo deste capítulo é validar nossa solução e os resultados obtidos na simulador. Relembrando o Capítulo 1, os requisitos da solução que propomos são:

1. ser automática (provida pelos mecanismos descritos Seções 5.1.3 e 5.1.4);
2. não estar acoplada à aplicação (Seção 5.1.2);
3. apresentar algum mecanismo que visa diminuir o tempo necessário para o término da aplicação (que é implementada através da utilização das heurísticas propostas no Capítulo 3).

5.1 MyGrid

Quando o desenvolvimento do MyGrid [23] foi iniciado, poucos usuários de aplicações BoT utilizavam grids apesar de estas aplicações serem adequadas à execução no grid e do grande ganho de desempenho que podiam obter. Acreditamos que isso acontecia devido à grande complexidade em utilizar as tecnologias grid.

O MyGrid teve como objetivo mudar este estado e possibilitar de maneira fácil a execução de aplicações BoT em todas as máquinas que o usuário tivesse acesso. Para tornar isso possível, MyGrid escolheu um *trade-off* diferente comparado a outras soluções de grid: abdicou-se da possibilidade de executar aplicações arbitrárias em troca de suportar somente as aplicações BoT. Entretanto, focando em aplicações BoT, MyGrid pode ser mantido mais simples para a utilização dos usuários. O foco na simplicidade permite ao usuário, por exemplo, executar suas aplicações mais rápido uma vez que não estão interessados em passar muito tempo configurando o ambiente de uma solução mais complexa que fornece mais do que suas necessidades. Vale salientar que o MyGrid não depende da infra-estrutura grid existente. Mas, se alguma infra-estrutura grid estiver disponível, MyGrid pode utilizá-la. Por exemplo, o MyGrid pode utilizar o Globus.

O MyGrid escalona a aplicação nos recursos que o usuário tem acesso, seja este acesso sobre algum middleware existente (tal como Globus [1]) ou através de login remoto simples (e.g. `ssh`). A solução de escalonamento do MyGrid é uma contribuição particularmente interessante, pois usa replicação de tarefas para conseguir bom desempenho sem necessitar de nenhuma informação sobre o grid ou a aplicação [59, 62].

Hoje, o MyGrid [23] é um dos componentes da solução OurGrid [2, 22] mas não pretendemos entrar em detalhes sobre todo o projeto, nosso foco é o broker.

5.1.1 Arquitetura

MyGrid assume que o usuário tem uma máquina que coordena a execução de aplicação BoT pelo grid montado com as máquinas que o usuário tem acesso. Esta máquina é chamada de máquina *home*. A submissão das tarefas que compõe a aplicação é feita para a máquina *home* que é responsável por gerenciá-las no grid. Ela deve ser uma máquina que o usuário use comumente, confiável (pois é nela que será centralizado o escalonamento das tarefas e o armazenamento de dados), onde o usuário tenha configurado seu ambiente de trabalho e onde ele não tenha problemas para instalação de novos programas. Acreditamos que a máquina que será utilizada normalmente para isso seja o *desktop* do usuário.

A máquina home escalona as tarefas para executar nas *máquinas do grid*. Estas

máquinas são descritas pelo usuário em um arquivo de configuração do MyGrid. Ao contrário da máquina home, as máquinas do grid não têm que ser configuradas pelo usuário para se obter um ambiente de trabalho familiar. Todavia, elas normalmente não compartilham o mesmo sistema de arquivos com a máquina home, nem têm os mesmos programas instalados. Uma vez que o ambiente deve ser configurado rapidamente, não se pode requerer que os usuários se preocupem com essas questões. Dessa maneira, o MyGrid deve prover abstrações que facilitem tal configuração.

Contudo, grids são compostos por recursos heterogêneos e prover um ambiente onde o usuário possa se beneficiar de todos os recursos que ele tem acesso não é fácil. A forma como o pedido de execução é feito varia de recurso para recurso. Por exemplo, executar um comando em uma máquina com Globus envolve a implementação de um módulo para troca de várias mensagens com o GRAM, enquanto com *ssh* é suficiente invocar o comando localmente pois a troca das mensagens necessárias já é implementada pelo programa (assumindo que a configuração de chaves público-privadas já foi feita). Diferenças semelhantes acontecem para troca de arquivos.

Dessa forma, verificamos quais são as funcionalidades que uma máquina do grid sempre deve ter para possibilitar a execução de aplicações BoT e foi definido um conjunto mínimo de serviços especificados pela interface *GridMachine* (GuM). Esses serviços são: (i) execução de um comando, (ii) transferência de arquivos da máquina home para a máquina do grid, (iii) transferência de arquivos da máquina do grid para a máquina home e (iv) cancelamento de um comando (matar o processo).

Essa virtualização possibilita que o funcionamento do escalonador seja simples uma vez que ele não precisa conhecer os detalhes sobre o acesso às máquinas do grid. Note que isso facilita também a adição de novos recursos, basta implementar a interface não sendo preciso modificar o restante do código. A Figura 5.1 representa essa arquitetura: *Globus Proxy* e *UAClient* executam na máquina *home* trocando informações com *Globus GRAM* e *UAServer*, respectivamente, para prover as funcionalidades de uma GuM; *GridScript* é responsável por executar *scripts* do usuário (por exemplo, uso do *ssh*) para prover as mesmas funcionalidades.

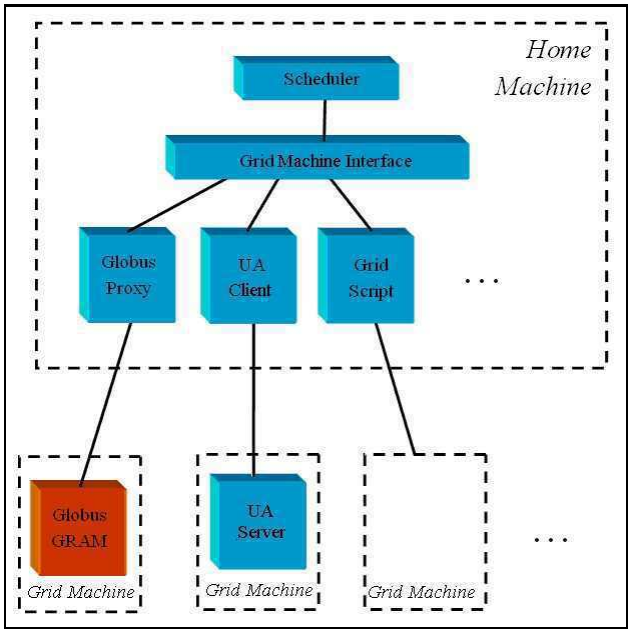


Figura 5.1: Arquitetura do MyGrid [23]

UserAgent

Para a implementação da nossa solução no MyGrid decidimos utilizar o *UserAgent*(UA) como a implementação da interface GridMachine. Ele é implementado em Java facilitando a portabilidade. O usuário o inicializa nas máquinas do grid e o UA funciona como um *daemon* esperando para atender às requisições do escalonador. O escalonador se comunica com o UA através de *Remote Method Invocation* (RMI) [43].

Uma das vantagens de usar o UA é que o uso de software de terceiros como parte da infra-estrutura pode trazer diversos problemas. Por exemplo, é difícil detectar o tipo de erro (comunicação ou execução) quando se usa ssh ou ainda diferentes versões podem trazer problemas com relação à passagem de parâmetros (e.g. tivemos problemas uma versão do ssh que recebia o parâmetro "t" e em outra versão pedia o parâmetro "T" para a mesma funcionalidade). Além disso, temos controle sobre o código para adicionar mais funcionalidades se necessário. Esses fatores fazem com que seja preferível utilizar o UserAgent que é a implementação da GridMachine totalmente desenvolvida pelo projeto OurGrid.

5.1.2 Suporte a recursos compartilhados no espaço

A abordagem de virtualização dos recursos descrita na seção 5.1.1 mantém o escalonador do MyGrid simples e é uma boa maneira de utilizar recursos intermitentes que podem ser usados a qualquer momento. Entretanto, como já comentado neste trabalho, a obtenção do acesso é diferente para recursos compartilhados no espaço.

A interação com o escalonador de recursos compartilhados no espaço também muda de um recurso para outro (e.g. comandos para submissão), o que fez surgir uma segunda abstração, a interface GridMachineProvider. Um GridMachineProvider segue a mesma idéia da GuM com relação à heterogeneidade, mas ele abstrai a forma de conseguir o acesso ao recurso e não a utilização em si. Desta maneira, isolamos a heterogeneidade dos recursos compartilhados no espaço e ainda temos o escalonador do MyGrid funcionando da mesma maneira. O escalonador do MyGrid, então, pede GuMs a um GridMachineProvider que se encarrega em consegui-las e devolver ao escalonador que sabe como utilizá-las.

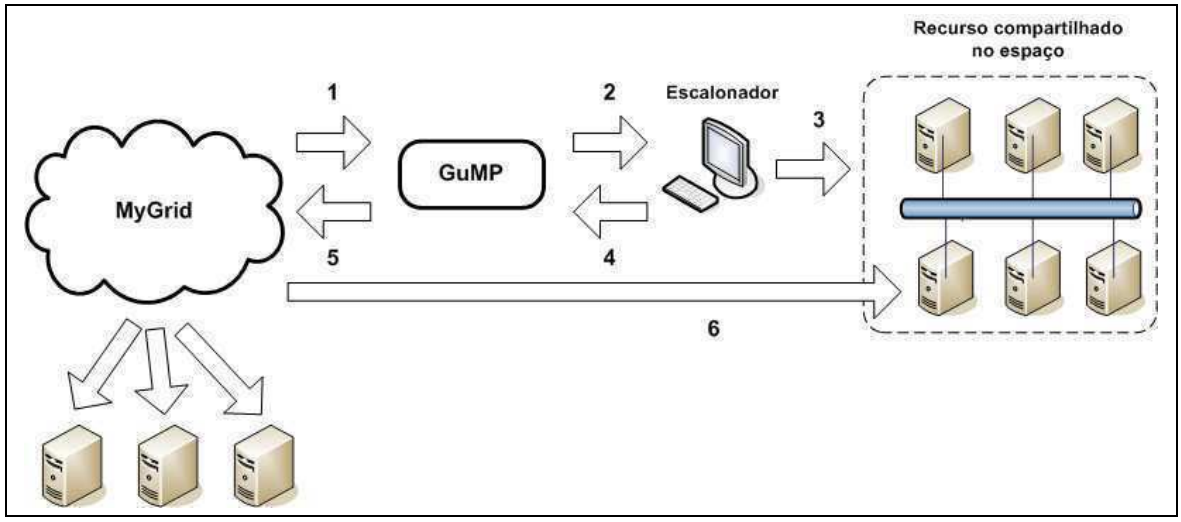


Figura 5.2: Interação Escalonador x GridMachineProvider

A interação é ilustrada na Figura 5.2. O escalonador solicita GuMs (1), o GridMachineProvider se comunica com o gerente de recursos (escalonador de recursos compartilhados no espaço) para obter informações sobre os processadores e realiza as operações necessárias para a utilização destes processadores, como descrito no Capítulo 3, submetendo *workers* que serão disponibilizados ao *broker* através de RMI (2,3,4);

o GridMachineProvider repassa os objetos (*stubs* RMI) para o escalonador (5) que os utiliza como qualquer outra GridMachine (6). Note que o GridMachineProvider atua para o componente adaptador de requisições apresentado na seção 3.1. Os passos são os mesmos para qualquer aplicações submetida ao MyGrid o que satisfaz o requisito 2 da solução (ser desacoplada das aplicações - ver Capítulo 1).

Um problema comum neste cenário é o fato de que as máquinas de um recurso compartilhado no espaço usam IP privado ou estão protegidas por *firewalls* e são acessadas através de uma máquina que serve como porta de entrada (*front-end*). Isto impede que o escalonador possa acessar as máquinas diretamente. Para solucionar este problema, utilizamos a máquina *front-end* para repassar os pedidos para as máquinas que compõem o recurso compartilhado no espaço (i.e. a máquina *front-end* faz *relay*).

Para repassar os pedidos de maneira transparente para o escalonador e para os *daemons* internos (das máquinas que estão com IP privado) resolvemos criar objetos *proxies* que implementam a mesma interface dos objetos das máquinas internas responsáveis por atender às requisições. Cada *proxy* tem uma referência (um *stub*) para um objeto UAServer e as implementações de seus métodos é apenas chamar o método de mesmo nome no UAServer. O escalonador recebe um *stub* para o *proxy* e não para o UAServer em si e faz a requisição ao objeto que recebeu sem se importar se a comunicação é direta ou se existem vários nós intermediários.

5.1.3 Monitorando as tarefas

Para estimar o tempo da tarefa baseado em execuções passadas, precisamos monitorar a execução para verificar se tiveram ou não sucesso. Para isto, a chamada de método no *proxy* para a execução no de um comando registra o início de uma tarefa antes de repassar para o objeto RMI final. Esse procedimento faz com que o objeto *proxy* funcione como um *Decorator* [41]. Quando a tarefa termina, é calculado o tempo que uma tarefa necessitou. Caso a tarefa não tenha finalizado, é guardado o tempo que ela ficou executando e a heurística estima um novo valor.

Outro fator relacionado a essa monitoração é a quantidade de tarefas que a heurística considera como pendentes. O procedimento do MyGrid é fazer pedidos por processadores e não informar quantas tarefas irão ser executados aos GuMPs. Uma

vez que as heurísticas precisam da informação sobre a quantidade de tarefas pendentes, consideramos que o MyGrid pede uma quantidade de processadores igual à quantidade de tarefas pendentes (de fato, é assim que ele age). Após essa consideração inicial, a quantidade de tarefas a serem executadas é decrementada de acordo com as informações fornecidas pelos objetos *proxies* sobre a finalização das tarefas.

5.1.4 Realizando a submissão

Realizamos a implementação da solução utilizando o CRONO [58] como escalonador de recurso compartilhado no espaço. Optamos pelo CRONO por duas razões: (i) ele é um escalonador simples com uma pequena quantidade de linhas de código quando comparamos com outros escalonadores (e.g. PBS [13] e Maui [45]) e (ii) ele é desenvolvido pelo CPAD/PUCRS, nosso parceiro de pesquisas, o que facilitou a compreensão sobre os detalhes do escalonador e a implementação realizada. Para fazer a submissão, fizemos a integração com a API do CRONO. Como o CRONO é implementado em *C* e o MyGrid em Java, utilizamos JNI (Java Native Interface) [50]. JNI é um mecanismo desenvolvido pela Sun^{TM} que permite ao código escrito em Java utilizar código escrito em outras linguagens e vice-versa. Isso automatiza a submissão de satisfaz o requisito 1 da nossa solução (ver Capítulo 1).

A Tabela 5.1 mostra as funções da API utilizadas pela nossa implementação. Note que para implementar a integração com um novo gerente de recursos, é necessário apenas fornecer ao nosso código Java as funcionalidades destes métodos. Vários métodos recebem o nome do cluster pois podemos encontrar casos onde um mesmo gerente de recursos mantém o controle e vários clusters como recursos independentes (com uma fila de requisições para cada recurso).

5.2 Análise das execuções

Em parceria com o CPAD/PUCRS [1], realizamos alguns experimentos durante o mês de setembro de 2005 com o objetivo de verificarmos a diferença entre o que obtemos no

[1] Centro de Pesquisa de Alto Desempenho da Pontifícia Universidade Católica do Rio Grande do Sul - `www.cpad.pucrs.br`

Método	**Descrição**
`AllocationResponse allocate(String clusterName, int numProc, int time, String bjscript);`	Submete uma requisição pedindo por *numProc* processadores e pelo tempo *time*. *bjscript* é o comando que deverá se executado quando a requisição ganhar a partição requisitada. O retorno é um objeto que tem o *id* da requisição (dentre outros dados).
`String release (String clusterName, int id);`	Cancela uma determinada requisição
`int getMaxRequests (String clusterName);`	Retorna a quantidade máxima de requisições que podem ser realizadas por um usuário
`int getMaxTimePerRequest (String clusterName);`	Retorna o tempo máximo de uma requisição
`int getMaxProcsPerRequest (String clusterName);`	Retorna a quantidade máxima de processadores de uma requisição
`Collection<AllocationResponse> getRequestsInQueue(String clusterName);`	Retorna a fila de requisições do recurso

Tabela 5.1: API do CRONO utilizada pelo MyGrid via JNI

simulador com os resultados da nossa implementação. Com este objetivo, executamos 35 jobs de 100 tarefas com dois valores possíveis para a média das tarefas (100s e 1500s) e duas heurísticas (estática e adaptativa) no recurso compartilhado no espaço do CPAD/PUCRS descrito no fim da Seção 4.8.

Armazenamos os *logs* do *broker* e do *cluster* do qual coletamos o tempo de execução dos grids jobs. Depois, extraímos do *log* do *cluster* as requisições submetidas pelo MyGrid para obter apenas as requisições de usuários locais. Geramos a entrada para o simulador e executamos o simulador para comparar os resultados. Em média a diferença do tempo de execução dos grid jobs foi de 115 segundos (a diferença foi similar inclusive para os jobs com tarefas de 1500s como tempo médio).

Analisando o *log* do MyGrid, observamos que existiu um atraso de cerca de 55 segundos entre a submissão do job e o tempo para começar a execução de uma tarefa (mesmo com processadores livre no momento da submissão do job). Este tempo é gasto no processo de criação dos objetos, criação da listas de requisições, submissão das requisições, início da execução dos *workers*, escalonamento das tarefas do grid e finalmente submissão da tarefa. Além deste atraso, toda vez que um processador fica livre, existe um intervalo para que uma nova tarefa seja escalonada.

Desta maneira, as diferenças nos resultados do simulador se devem aos atrasos para o sistema executar as operações e não a erros na implementação do simulador. Estes atrasos podem ser facilmente levados em consideração na simulação, basta adicionar o tempo correspondente ao tempo das tarefas o que nos dar confiança nos resultados obtidos para a comparação das heurísticas.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho possibilita o uso de processadores de recursos compartilhados no espaço para os *brokers* de grid para aplicações BoT sem modificações nos *brokers* e foi publicado parcialmente em [26]. Esta solução sem modificações nos *brokers* traz a vantagem de reusarmos toda a pesquisa já feita para estes *brokers* (e.g. algoritmos de escalonamento [40, 56, 59, 62]). Outra vantagem é que este modo de uso dos recursos compartilhados no espaço que propormos não pede ao usuário novas informações para usar os recursos. Isto mantém a simplicidade de uso do grid o que acreditamos ser um dos principais fatores de sucesso destes *brokers*.

Os recursos compartilhados no espaço só podem ser utilizados mediante a submissão de requisição que especifiquem a quantidade de processadores a serem utilizados e por quanto tempo eles serão alocados (p, t^r). Para especificar automaticamente estes parâmetros, fizemos uso de heurísticas que tentam maximizar a vazão de tarefas escolhendo requisições dentre as requisições possíveis na fila do recurso. As duas principais heurísticas analisadas foram: heurística estática e heurística adaptativa, que estima o tempo da tarefa e verifica a carga do recurso antes de fazer a requisição.

Realizamos simulações para avaliar o desempenho das heurísticas. A métrica utilizada foi o tempo gasto na execução do job do grid (ou seja, o período de tempo entre a submissão do job e o seu término deste). Na maioria dos casos, a heurística adaptativa foi a de melhor desempenho. Verificamos também que esta heurística tende a obter melhores resultados em situações onde o recurso compartilhado no espaço tem carga baixa e para jobs pequenos.

Devido às situações onde a heurística estática é melhor, criamos uma heurística chamada heurística mista com características das duas citadas. Todavia, a heurística mista não foi melhor como esperávamos, pois provia tempos de execução intermediários para todos os jobs. Desta maneira, recomendamos o uso da heurística adaptativa a menos que o usuário saiba previamente que suas tarefas são longas.

Verificamos também o desperdício do recurso que é inerente a todas as heurísticas. O resultado é bastante semelhante aos observados com o tempo de execução onde a heurística adaptativa tem menor desperdício. Todavia, em situações de carga baixa seu excelente desempenho com relação ao tempo de execução é pago com maior desperdício que a heurística estática.

Com o objetivo de aumentar a confiança nos resultados do nosso simulador, realizamos alguns experimentos além dos testes de unidade durante o desenvolvimento do software. Comparamos o resultado da nossa simulação com o resultado de outro simulador para quatro workloads: duas workloads apresentaram resultados idênticos e as outras duas apresentaram mais de 97% das requisições com resultados idênticos. Outra comparação foi realizada com um trace de um sistema em produção, simulamos o trace e comparamos com o comportamento do sistema real: a grande maioria (mais de 97%) das requisições tiveram resultados idênticos.

Implementamos a solução proposta no OurGrid de maneira que temos um protótipo da nossa solução implementado para um sistema em produção. Este protótipo está em uso atualmente dentro do projeto SegHidro [12].

Ainda com o objetivo de aumentar a confiança nos resultados do nosso simulador, realizamos alguns experimentos e comparamos as simulações. As diferenças apresentadas se devem à execução de operações dos sistemas não levadas em consideração pelo simulador. Todavia, estes atrasos podem ser facilmente levados em consideração adicionando tempo extra em cada tarefa do grid.

Atualmente, estamos realizando um trabalho em conjunto da solução apresentada neste trabalho para recursos compartilhados no espaço com a solução oportunista desenvolvido pelo CPAD/PUCRS [57]. Estamos verificando as vantagens e desvantagens de cada estratégia para utilizar um recurso compartilhado no espaço em grids BoT.

Dentro do projeto OurGrid, observamos que devemos integrar a solução proposta

com o trabalho desenvolvido por Daniel Fireman em [34]. Fireman implementou suporte a aplicações acopladas no ambiente OurGrid sem modificar a visão de ambiente BoT para os usuários. A solução utilizada possibilita que uma tarefa seja descrita com requisitos para utilizar mais de um processador, os *GuMProviders* se encarregam de obter vários processadores e agrupá-los em uma abstração *GridMachine*. O escalonador do MyGrid continua a realizar a associação de uma tarefa por *GridMachine* mas neste caso é como se as tarefas podem ser compostas por subtarefas que executam em processadores diferentes e se comunicam (e.g. aplicações MPI [68]).

Outra possibilidade de trabalho futuro é implementar no adaptador de requisições uma maneira de detectar em qual situação ele está (verificando as características do recurso e da aplicação) e ativar a heurística que melhor obtém resultados em tal situação numa abordagem semelhante à apresentada em [71].

Dois trabalhos que devemos realizar servirão para refinar e consolidar a pesquisa apresentada nesta dissertação são: (i) pesquisa com usuários de recursos compartilhados no espaço, onde obteremos a informação da utilização de aplicações BoT nesses recursos e (ii) caracterização de *workloads* para grids BoT.

O item (i) seria de grande utilidade para uma das possibilidades apontadas para este trabalho durante a apresentação da proposta de dissertação: o usuário local do recurso compartilhado no espaço utilizando as heurísticas para executar suas aplicações mesmo sem fazer através de um grid *broker*. Este item também poderia ser utilizado para realizar o item (ii).

O item (ii) possibilitaria que os nossos resultados fossem mais fortes e que não fosse preciso uma análise com tantas possibilidades diferentes. Relembrando, só efetuamos simulações com um grande número de combinações porque não tínhamos *workloads* que descrevessem os jobs do grid. Neste sentido, começamos um trabalho para realizar esta caracterização com base em *workloads* do OurGrid e em *workloads* de diversos outros projetos de diferentes países.

Referências Bibliográficas

[1] Globus project. http://www.globus.org.

[2] Ourgrid project. http://www.ourgrid.org/.

[3] Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[4] SETI@home Statistics Page. http://setiathome.ssl.berkeley.edu/totals.html.

[5] Seti@home web page. http://www.seti.org/science/setiathome.html.

[6] United devices web page. http://www.ud.com.

[7] NPACI Users Guide: Blue Horizon. http://www.npaci.edu/Horizon/, June 2004.

[8] Platform computing inc web page. http://www.platform.com/, December 2004.

[9] K. Aida. Effect of Job Size Characteristics on Job Scheduling Performance. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 6th Workshop Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.

[10] K. Aida, H. Kasahara, and S. Narita. Job Scheduling Scheme for Pure Space Sharing among Rigid Jobs. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 4th Workshop Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 98–121. Springer-Verlag, 1998.

[11] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[12] E. Araújo, W. Cirne, G. Wagner, N. Oliveira, E. P. Souza, C. O. Galvão, and E. S. Martins. The SegHidro experience: Using the grid to empower a hydro-meteorological scientific network. In *Proceedings of the 1st IEEE Conference on e-Science and Grid Computing*, Dec. 2005.

[13] A. Bayucan. *Portable Batch System Administration.* Veridian System, 2000.

[14] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* Wiley Interscience. John Wiley & Sons, Inc., first edition, April 2003.

[15] R. Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling Parameter Sweep Applications on Global Grids: A Deadline and Budget Constrained Cost-Time Optimisation Algorithm. *International Journal of Software: Practice and Experience (SPE)*, 35(5):419–512, April 2005.

[16] S.-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 103–127. Springer-Verlag, June 2002.

[17] W. Cirne. *Using Moldability to Improve the Performance of Supercomputer Jobs.* PhD thesis, University of California San Diego, February 2001.

[18] W. Cirne. Grids Computacionais: Arquiteturas, Tecnologias e Aplicações. In *Terceiro Workshop em Sistemas Computacionais de Alto Desempenho*, Outubro 2002.

[19] W. Cirne and F. Berman. A Comprehensive Model of the Supercomputer Workload. In *Proceedings of WWC-4: IEEE 4th Annual Workshop on Workload Characterization*, pages 140 – 148, December 2001.

[20] W. Cirne and F. Berman. Using Moldability to Improve the Performance of Supercomputer Jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, October 2002.

[21] W. Cirne and F. Berman. When the Herd is Smart: The Aggregate Behavior in the Selection of Job Request. *IEEE Transactions in Parallel and Distributed Systems*, 14(2):181–192, February 2003.

[22] W. Cirne, F. Brasileiro, N. Andrade, L. B. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! Technical report, UFCG/DSC, Campina Grande - PB, August 2005. `http://www.ourgrid.org/twiki-public/pub/OG/OurPublications/Labs_of_the_World_Unite_v16.pdf`.

[23] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. A. B. da Silva, C. O. Barros, and C. Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of ICPP'2003 - 32th International Conference on Parallel Processing*, pages 407–416. IEEE, October 2003.

[24] L. B. Costa. Convertendo recursos compartilhados no espaço em recursos intermitentes. Technical report, UFCG, Campina Grande - PB, Fevereiro 2005. Proposta de dissertação de Mestrado. Walfredo Cirne, Orientador.

[25] L. B. Costa, W. Cirne, and C. A. F. De Rose. Exploring task independence to schedule Bag-of-Tasks jobs in parallel supercomputers. Technical report, UFCG, Campina Grande - PB, Junho 2004.

[26] L. B. Costa, W. Cirne, and D. Fireman. Converting Space Shared Resources into Intermittent Resources for use in Bag-of-Tasks Grids . In *Proceedings of SBAC-PAD 2005 - 17th Symposium on Computer Architecture and High Performance Computing*. IEEE Press, October 2005.

[27] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 4th Workshop Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, page 62. Spring-Verlag, January 1998.

[28] A. Downey. Using Queue Time Predictions for Processor Allocation. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 3rd Workshop on Job Scheduling Strategies for*

Parallel Processing, volume 1291 of *Lecture Notes in Computer Science*, pages 35–57. Springer-Verlag, 1997.

[29] D. England and J. B. Weissman. Costs and Benefits of Load Sharing in the Computational Grid. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proceedings of 10th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 160–175. Springer Verlag, June 2004.

[30] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[31] D. G. Feitelson. Experimental analysis of the root causes of performance evaluation results: a backfilling case study. *IEEE Transactions in Parallel and Distributed Systems*, 16(2):175–182, Feb 2005.

[32] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1996.

[33] D. G. Feitelson, L. Rudolph, U. Schweigelshohn, K. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 3rd Workshop Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer-Verlag, 1997.

[34] D. Fireman. Implementando suporte a aplicações acopladas no ambiente OurGrid. Technical report, UFCG, Campina Grande - PB, Junho 2005. Walfredo Cirne, Orientador.

[35] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[36] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and

co-allocation. In *Proceedings of the Seventh International Workshop on Quality of Service*, pages 27–36, 1999.

[37] I. Foster, C. Kesselman, J. Nick, S. Tuecke, Open Grid Service Infrastructure WG, and Global Grid Forum. The Physiology of the Grid: An Open Grid Services Architecture for distributed systems integration, 2002.

[38] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, August 2001.

[39] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing HPDC*, pages 55–63, San Francisco, California, August 2001.

[40] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *Proceedings of ICPP'2003 - 32th International Conference on Parallel Processing*, pages 391–398, October 2003.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter Decorator, pages 175–184. Addison Wesley, 1995.

[42] R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 3rd Workshop Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 58–77. Springer-Verlag, June 1997.

[43] W. Grosso. *Java RMI*. O'Reilly, first edition, October 2001. ISBN 1-56592-452-5. `http://java.sun.com/products/jdk/rmi/`.

[44] R. L. Henderson. Job Scheduling Under the Portable Batch System. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 1st Workshop Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer-Verlag, 1995.

[45] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–102. Springer Verlag, 2001.

[46] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, Aug. 1988.

[47] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley Interscience. John Wiley & Sons, Inc., New York, NY, first edition, 1991.

[48] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proceedings of 10th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 253–263. Springer-Verlag, June 2004.

[49] M. J. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proceedings of Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[50] S. Liang. *Java Native Interface: Programmer's Guide and Specification.* The Java Series. Addison-Wesley, first edition, June 1999. ISBN 0201325772. `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`.

[51] D. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 1st Workshop Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer-Verlag, 1995.

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

[53] V. Lo, J. Mache, and K. Windisch. A Comparative Study of Real Workload Traces and Synthetic Workload Models for Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 4th Workshop Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 25–46. Springer-Verlag, March 1998.

[54] J. A. Moreira and W. Cirne. Usando Performance Relativa para Avaliar Desempenho em Supercomputadores ParalelosUsing Relative Performance to Gauge Parallel Supercomputer Performance. In *Anais do Terceiro Workshop em Sistemas Computacionais de Alto DesempenhoProceedings of the Third Workshop on High Perfomance Computing Systems*, October 2002.

[55] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimatesin scheduling the IBM SP2 with backfilling. *IEEE Transactions in Parallel and Distributed Systems*, 12(6):529–543, 2001.

[56] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: scalability issues in global computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000.

[57] M. A. S. Netto, R. Calheiros, R. Silva, C. Northfleet, C. A. F. de Rose, and W. Cirne. Transparent resource allocation to exploit idle cluster nodes in computational grids. In *Proceedings of the First IEEE International Conference on e-Science and Grid Computing*, December 2005.

[58] M. A. S. Netto and C. A. F. de Rose. CRONO: A configurable and easy to maintain resource manager optimized for small and mid-size GNU/Linux Clusters. In *Proceedings of ICPP'2003 - 32th International Conference on Parallel Processing*, pages 555–564. IEEE, October 2003.

[59] D. Paranhos, W. Cirne, and F. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proceedings of Euro-Par 2003: International Conference on Parallel and Distributed Computing*, pages 169–180, August 2003.

[60] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with Carmi. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. Springer-Verlag, 1995.

[61] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proceedings of 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 87–104. Springer Verlag, June 2003.

[62] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 10th Workshop Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 210–232. Springer-Verlag, June 2004.

[63] L. F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[64] F. A. B. Silva, S. Carvalho, H. Senger, E. R. Hruschka, and C. R. G. de Farias. Running Data Mining Applications on the Grid: a Bag-of-tasks Approach. In *International Conference on Computational Science and its Applications ICCSA 2004*, 2004.

[65] S. Smallen, H. Casanova, and F. Berman. Applying Scheduling and Tuning to On-line Parallel Tomography. In *Supercomputing 01*, Colorado, USA, November 2001.

[66] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M.-H. Su, C. Kesselman, S. Young, , and M. Ellisman. Combining workstations and supercomputers to support grid applications: The parallel tomography experience. In *Proceedings of 9th Heterogeneous Computing Workshop - HCW'2000*, pages 241–252, May 2000.

[67] C. Smith. Open source metascheduling for Virtual Oraganizations with Community Scheduler Framework CSF Technical Whitepaper. Technical report, Platform Compuntig Inc, 2004.

[68] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, first edition, 1995. ISBN 0262691841.

[69] S. Son and M. Livny. Recovering internet symmetry in distributed computing. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid - CCGrid'03*, pages 542–550, Tokyo, Japan, May 2003. IEEE Computer Society.

[70] J. R. Stiles, T. M. Bartol, E. E. Salpeter, and M. M. Salpeter. Monte carlo simulation of neuromuscular transmitter release using mcell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

[71] D. Talby and D. G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *Proceedings of 19th Intl. Parallel and Distributed Processing Symposium*. IEEE, April 2005.

[72] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, Feb 2005.

[73] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling User Runtime Estimates. In D. G. Feitelson and L. Rudolph, editors, *Proceedings of 11th Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science. Springer-Verlag, June 2005.

[74] W. A. Ward Jr., C. L. Mahood, and J. E. West. Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 88–102. Springer-Verlag, June 2002.

[75] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, Champaign - Urbana, IL, June 2001.