Universidade Federal de Campina Grande

Centro de Ciência e Tecnologia

Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Escalonamento de Aplicações que Processam Grandes Quantidades de Dados em Grids Computacionais

Elizeu Lourenço dos Santos Neto

Campina Grande, Paraíba, Brasil

Março - 2004

SANTOS NETO, Elizeu Lourenço dos
S237E

Escalonamento de Aplicações que Processam Grandes Quantidades de Dados em Grids Computacionais.

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Março de 2004.

71 p. Il.

Orientador: Walfredo Cirne

Palavras Chave:

1. Sistemas Distribuídos
2. Grids Computacionais
3. Escalonamento
4. Bag of Tasks

CDU - 681.3.066D

Escalonamento de Aplicações que Processam Grandes Quantidades de Dados em Grids Computacionais

Elizeu Lourenço dos Santos Neto

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Walfredo Cirne

(Orientador)

Campina Grande, Paraíba, Brasil

Resumo

Aplicações que processam grandes quantidades de dados demandam grandes transferências de dados quando executadas em grids computacionais. Estas transferências têm um alto custo associado. Portanto, considerar as transferências de dados é fundamental para se obter escalonamentos efi cientes para tais aplicações. Além disso, em ambientes heterogêneos como os grids, as heurísticas que produzem escalonamentos efi cientes tipicamente usam informação dinâmica sobre o grid e as aplicações (disponibilidade de rede e CPU, tempo de execução das tarefas, etc). Porém, estas informações são, em geral, difíceis de se obter com precisão. Embora existam escalonadores que alcançam bom desempenho sem usar informações dinâmicas, eles não são desenvolvidos para considerar o impacto das transferências de dados. Neste trabalho apresentamos *Storage Affinity*, uma nova heurística de escalonamento para aplicações do tipo *Bag-of-Tasks* que processam grandes quantidades de dados sem depender de informação de difícil obtenção. Além disso, o ambiente de execução considerado é um grid computacional. *Storage Affinity* explora os padrões de reutilização de dados, comuns em muitas aplicações, pois isto permite considerar as transferências de dados sem usar informações dinâmicas sobre os recursos, reduzindo o tempo total de execução da aplicação. Através do uso de uma estratégia de replicação de tarefas, Storage Affi nity efetua escalonamentos efi cientes sem depender de informação dinâmica. Os resultados mostram que Storage Affi nity pode alcançar uma performance, em média, 42% melhor do que os escalonadores *estado-da-arte* que dependem de informação, mesmo em situações onde tais escalonadores usam informação perfeita. Em contrapartida, há um acréscimo no consumo de ciclos de CPU (em média, 59%) para alcançar este desempenho devido a replicação de tarefas.

Abstract

Data-intensive applications executing over a computational grid demand large data transfers. These are costly operations. Therefore, taking them into account is mandatory to achieve efficient scheduling of data-intensive applications on grids. Further, within an heterogeneous environment such as a grid, good schedules are typically attained by heuristics that use dynamic information about the grid and the applications (network and CPU loads, completion time of tasks, etc). However, these information are often difficult to be obtained accurately. Although there are schedulers that attain good performance without requiring that kind of information, they were not designed to take data transfer delays into account. This work presents *Storage Affinity*, a novel scheduling heuristic for *Bag-of-Tasks* and *data-intensive* applications running on grid environments. *Storage Affinity* exploits a data reuse pattern, common on many data-intensive applications, allowing it to take data transfer delays into account and reduce the makespan of the application. Further, it uses a replication strategy that yields efficient schedules without relying upon dynamic information that is difficult to obtain. Our results show that Storage Affinity may attain performance that is in average 42% better than that of state-of-the-art knowledge-dependent schedulers, even in the unlikely case when the latter are fed with perfect information. This is achieved at the expense of consuming more CPU cycles (in average, 59% more than not using replication).

Dedicatória

Dedico mais este trabalho à minha querida Mãe (*in memorian*), ao meu Pai e a Lu. Muito obrigado pelo carinho, dedicação e a constante presença de vocês.

Agradecimentos

Agradeço a minha família, aos meus queridos e carinhosos irmãos: Chris, Sílvia, Lula e Arnaldinho e aos meus avós que são um exemplo de perseverança.

Um muito obrigado especial a Walfredo, muito mais que um orientador, um grande amigo; e a Fubica pelos ensinamentos, discussões e pela fundamental descontração.

Agradeço também a Leandro, meu melhor amigo, pelo bom humor e pela paciência. Nazaga pela grande amizade que construímos e pelas discussões científi co-artístico-culturais. Aliandro, sem dúvida teria sido muito mais difícil sem sua ajuda, muito obrigado. Aos caras *sem comentários*: Bruce Dilson e Calvanha (valeu amigos).

Às minhas amigas Glau-Mau, Eliane e Robs; ao Prof. Paranhos, Laurinho, Pipinho (o fenômeno), Tavinho, Nigini, Loreno e Randolph e a todos do LSD (Lab. de Sistemas Distribuídos) pela compreensão quando sequestrei as almas das máquinas de vocês para executar as simulações, bem como pelos momentos de diversão. Serei eternamente grato a Rodrigo e seus pais (D. Gracinha e S. Paulo) pelo acolhimento espontâneo em Natal.

Zane Cirne pela competência na administração dos recursos do LSD. A Marcelo Meira pela cobrança saudável nos prazos a cumprir. Muito obrigado também a Keka por sempre nos auxiliar com carinho e atenção.

Ao professor Jacques Sauvé pelo incentivo, Aninha e Vera pela paciência. A todos os professores e funcionários do Departamento de Sistemas e Computação da UFCG.

A todos da HP Brasil e CAPES pelo suporte. Jornada (HP), Caio e Thiago (PUC-RS), Fabrício (UniSantos) e Carla (LNCC) pela receptividade em nossas visitas. Ao pessoal do Instituto Eldorado, Unisantos, Grid Lab/UCSD, LCAD/UFES, NACAD/UFRJ, Typhon Cluster/UFAL e LNCC por terem cedido seus recursos para execução das simulações e experimentos.

Finalmente, devo um agradecimento ao pessoal do *futebol da terça*, ao Dr. Geovanini, aos descobridores dos efeitos da cafeína, ao Sepultura por lançar um CD ótimo em 2003 e ao RUSH por tocar no Brasil. Valeu! :-P

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Cada vez mais encontramos aplicações que necessitam processar grandes quantidades de dados. Por exemplo, bioinformática [1, 2], física de altas energias [3], visualização científica [4] e processamento de imagens [5, 6] são áreas onde existem aplicações que trabalham com dados de forma intensiva. Além disso, um grande crescimento na geração de informação tem sido registrado nos últimos anos [7]. Para processar grandes conjuntos de dados, em geral, é necessário uma infraestrutura de computação de alto desempenho. Felizmente, na maioria das vezes, o procedimento de particionamento dos dados é fácil e as partições podem ser processadas em paralelo e independentemente, o que facilita o uso de recursos amplamente distribuídos para execução destas aplicações.

Considerando que cada partição é processada por uma tarefa, a independência durante o processamento de cada partição de dados caracteriza as aplicações do tipo *Bag-of-Tasks* (BoT) [8, 9]. Uma aplicação BoT é composta de tarefas que não necessitam se comunicar entre si para continuar com suas computações. Neste trabalho, estamos interessados em aplicações BoT que processam grandes quantidades de dados. A classe de aplicações que possuem estas características é denominada *processors of huge data* (pHd) [10] (ver Apêndice A).

Devido ao fato de suas tarefas serem independentes, as aplicações BoT são normalmente adequadas para executar em ambientes onde os recursos são amplamente distribuídos, por exemplo, os grids computacionais [8, 11, 12]. Porém, uma vez que os domínios que abrigam os recursos dos grids são conectados por redes onde, em geral, há limitação de banda, grandes transferências de dados influenciarão as execuções de aplicações pHd nesses am-

bientes. Portanto, utilizar estratégias que reduzam o impacto das transferências de grandes quantidades de dados será particularmente relevante para o desempenho das aplicações PHD.

Em aplicações PHD é comum haver reutilização de dados. Naturalmente a reutilização de dados pode ser explorada no intuito de obter um melhor desempenho para a aplicação. A reutilização de dados da entrada pode acontecer entre tarefas durante a execução de uma aplicação ou entre execuções sucessivas da aplicação (ver Seção 3.2). Por exemplo, no processo de visualização dos resultados de simulações de óptica quântica [4] é comum a realização de uma seqüência de execuções da mesma aplicação modifi cando alguns poucos parâmetros (e.g. zoom, ponto de vista) e preservando a maior parte dos dados de entrada usados nas execuções anteriores.

Existem algoritmos que consideram as transferências de dados durante o escalonamento das aplicações PHD em grids computacionais [13–16].Porém, estes algoritmos requerem conhecimento sobre o ambiente e sobre a aplicação que não é fácil de ser obtido precisamente na prática, principalmente em ambientes amplamente dispersos como os grids computacionais. Por exemplo, *XSufferage* [14] usa informações sobre os níveis de disponibilidade de CPU, da rede e os tempos de execução das tarefas. Estas informações devem ser conhecidas antes do escalonamento e são utilizadas para decidir qual tarefa deve ser escalonada em que processador.

Por outro lado, para aplicações BOT que não processam dados de forma intensiva, existem escalonadores que não usam informação sobre o ambiente nem sobre a aplicação, mesmo assim conseguem boa performance (e.g. *Workqueue with Replication* - WQR [17]). Através do uso de replicação é possível tolerar decisões de escalonamento inefi cientes que ocorreram pela falta de informação precisa sobre o ambiente e sobre a aplicação. Contudo, estes escalonadores não foram concebidos para tratar com aplicações que processam grandes quantidades de dados (*i.e.* o impacto das transferências de dados não é considerado).

Neste trabalho, nós introduzimos *Storage Affinity*, uma nova heurística de escalonamento para aplicações PHD em grids computacionais. *Storage Affinity* leva em conta o fato dos dados de entrada da aplicação serem freqüentemente reutilizados em execuções sucessivas. Um registro da localização dos dados utilizados pela aplicação é mantido e permite que o escalonador associe as tarefas de forma a evitar, tanto quanto possível, transferências desnecessárias de grandes quantidades de dados. Além disso, o efeito de decisões inefi cientes de

escalonamento é reduzido através do uso de replicação de tarefas.

O restante desta dissertação está organizada da seguinte forma. No próximo capítulo são apresentados os conceitos relacionados aos grids computacionais, as motivações de se utilizar grids computacionais para se executar aplicações PHD, bem como as questões que devem ser abordadas para se conseguir bom desempenho em aplicações PHD executando em grids computacionais. No Capítulo 3 apresentamos as soluções de escalonamento de aplicação existentes e introduzimos a heurística *Storage Affinity*. No Capítulo 4 são apresentados os resultados das comparações de performance entre os escalonadores, ressaltando quais aspectos do ambiente e da aplicação influenciam no desempenho da aplicação de acordo com cada escalonador. No Capítulo 5 são apresentadas as conclusões finais e idéias de possíveis trabalhos futuros. Para complementar, foi incluído o artigo "*Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids*", este artigo foi submetido ao *10th Workshop on Job Scheduling Strategies for Parallel Processing* e descreve de forma mais concisa a heurística *Storage Affinity* bem como os resultados da avaliação de desempenho.

Capítulo 2

Grids Computacionais

Grids computacionais são ambientes que agregam recursos heterogêneos (i.e. processadores, servidores de dados, instrumentos, serviços, etc) que estão amplamente distribuídos e conectados através de redes de longa distância. Além disso, estes recursos estão, em geral, sob o domínio de diversas entidades administrativas independentes [8, 11, 12].

O termo Grid Computacional (*computational grid*) é uma analogia com a rede de energia elétrica (*electric grid*). Como sabemos, a rede de energia elétrica fornece energia elétrica de forma transparente e sob demanda. Sendo assim, a proposta de um grid computacional é ser uma infraestrutura de computação que atenda às necessidades computacionais dos usuários abstraindo os detalhes de como esse atendimento é realizado.

Um primeiro passo para a concretização desta visão foi a utilização de ciclos de CPU ociosos em escala global para execução de aplicações BoT (*Bag-of-Tasks*). Vários projetos mostraram que é possível explorar esse poder computacional latente [8, 18–20]. Um aspecto importante que viabilizou estes projetos foi a facilidade em paralelizar suas aplicações alvo (estas aplicações são compostas de várias tarefas que podem ser processadas de forma independente). Desta forma, aplicações como fatoração de números [21], processamento de imagens [22], física de partículas [23], biologia molecular [24] e processamento de sinais [18] foram extremamente benefi ciadas com a utilização de recursos em escala global.

Os grids computacionais foram se tornando cada vez mais abrangentes. Deixaram de ser um método de agregação de recursos amplamente distribuídos em larga escala, com o foco apenas na comunidade de computação de alto desemepenho (grids de alto desempenho), para ser uma infraestrutura de integração de sistemas orientada a serviço (grids de serviços). As

novas possibilidades de aplicação dos grids computacionais, incentivaram a convergência das tecnologias grid com tecnologias comerciais como os *Web Services* [25]. Neste capítulo, discutiremos como ocorreu essa evolução, as vantagens do uso de grids para o processamento de grandes quantidades de dados e quais são as peças chave no desenvolvimento dos grids de serviços.

2.1 Grids de alto desempenho

O surgimento dos grids foi impulsionado pelos esforços de pesquisa nas área de computação paralela e sistemas distribuídos em geral. Assim, o grid foi idealizado como uma evolução dos paradigmas de computação paralela voltados para arquiteturas fortemente acopladas como SMPs e clusters [26–29]. O aspecto que mais encorajou os esforços com relação ao desenvolvimento dos grids foi a possibilidade de utilizar sob demanda uma quantidade muito grande de recursos sem o ônus da obtenção e manutenção de uma grande infraestrutura central com todos estes recursos.

Durante a conferência *Supercomputing'95* (www.sc-conference.org) foi feita uma demonstração do projeto I-WAY (Information Wide-Area Year) [12]. Este projeto consistia, basicamente, de uma infraestrutura de hardware e software que permitiu executar um conjunto de mais de 60 aplicações durante o evento. Os recursos desta infraestrutura experimental se encontravam distribuídos em 17 sites. Abordando aspectos de segurança foi possível garantir acesso seguro aos recursos. A partir desta experiência houve uma expansão das possibilidades de pesquisa em grids computacionais, estimulando novos projetos e tornando a visão com relação aos grids mais abrangente e ambiciosa.

Os estímulos gerados pelo I-WAY foram refletidos em forma de refi namento de projetos já existentes e na criação de novos projetos para o desenvolvimento de aspectos particulares dos grids computacionais. Por exemplo, o *Condor* [30, 31] (projeto anterior ao I-WAY) evoluiu alinhado ao desenvolvimento de grids mantendo seus objetivos iniciais. O foco do *Condor* se mantém na investigação e desenvolvimento de mecanismos e políticas para suportar *High-Throughput Computing* numa infraestrutura distribuída e heterogênea. O principal objetivo do *Condor* é explorar a capacidade de execução de recursos amplamente distribuídos considerando longos períodos de tempo. Assim, a idéia foi disponibilizar um conjunto

de funcionalidades que permitisse a execução de aplicações em lote através da exploração de ciclos ociosos de CPU. Desta forma, os mecanismos providos pelo *Condor* tornam possível a construção de ambientes de computação de alto desempenho formados por recursos amplamente distribuídos. Uma prova dessa evolução é o *Condor-G*, um projeto que une os protocolos de segurança e acesso a recursos localizados em domínios independentes (Globus) com as tecnologias de gerenciamento e aproveitamento de ciclos ociosos dentro de um site (Condor).

Outros projetos como Globus [32] e Legion [33], são mais voltados para a definição e implementação das camadas de mais baixo nível no desenvolvimento dos grids computacionais. Estas camadas permitem a criação de abstrações e o desenvolvimento de várias funcionalidades básicas para os grids, como infraestrutura de segurança e acesso aos recursos. A proposta do Legion é fornecer um conjunto de componentes para a construção de grids, isso inclui sistema de arquivos, segurança e comunicação entre processos, além de uma visão de uma máquina virtual composta de vários recursos distribuídos. Por sua vez, Globus consiste de um conjunto de serviços essenciais que servem como base para a construção de grids (i.e. alocação de recursos, autenticação, acesso a dados).

Um aspecto bastante relevante é como tornar eficiente, do ponto de vista da aplicação, o acesso e uso da infraestrutura do grid. Desenvolver mecanismos de escalonamento voltados às necessidades da aplicação, como proposto pelo projeto AppLeS [34,35], é uma abordagem bastante utilizada. Por exemplo, APST (AppLeS Parameter Sweep Tempalte) [36] é um esforço que utiliza técnicas de escalonamento de aplicação para tornar eficiente a execução de aplicações do tipo *parameter sweep* em grids computacionais. A idéia consiste em efetuar o escalonamento das tarefas que compõem a aplicação de forma a reduzir o tempo total de execução, sem deixar de atender aos requisitos da aplicação. As aplicações *parameter sweep* são consideradas aplicações paralelas leves, pois necessitam de pouca ou nenhuma comunicação entre suas tarefas. Além disso, suas execuções são geralmente repetidas com freqüência, com uma pequena alteração em seus parâmetros de entrada. Aplicações com estas características são encontradas em várias áreas do conhecimento, por isso a importância em prover a possibilidade de executá-las de forma eficiente.

Apesar desses esforços, ainda não era fácil para usuários de aplicações BoT executar suas aplicações em um grid formado pelos recursos que o mesmo tivesse acesso. Esta difi-

culdade estava ligada a questões de administração dos recursos e ao fato de que as soluções para grids não estavam largamente disponíveis. Assim, o usuário necessitava de uma solução simples que permitisse a execução de aplicações BoT em grids. Portanto, MyGrid [8] foi desenvolvido pensando em prover uma solução prática voltada para o usuário de aplicações BoT, permitindo o mesmo executar suas aplicações em todos os recursos que tem acesso sem necessitar a intervenção de administradores para instalação de pacotes particulares de software. MyGrid é construído usando virtualização de recursos. A virtualização é o encapsulamento de várias tipos de recursos que fornecem uma funcionalidade através de uma interface comum a todas as implementações. Usando esse conjunto de abstrações o usuário pode agregar diversos tipos de computadores ao seu grid (Figura 2.1) [8]. Para efetuar a agregação o usuário deve informar ao MyGrid, quais recursos que compõem seu grid e como acessá-los (transferência de arquivos e execução remota). Semelhante à forma como são descritos os recursos que compõem o grid, o usuário descreve as tarefas que formam a aplicação. A descrição das tarefas consiste em indicar scripts de pré e pós-processamento, de execução remota e os arquivos de entrada e de saída de cada tarefa. O formato de descrição da aplicação permite que o usuário indique requisitos para a aplicação, no que se refere à plataforma de hardware e software desejada. Para uma descrição detalhada dos métodos de acesso suportados pelo MyGrid, sugerimos ao leitor acessar o site do projeto (*http://www.ourgrid.org/mygrid*).

Uma das principais funcionalidades do MyGrid é o escalonamento de aplicações. MyGrid tenta executar a aplicação da forma mais efi ciente possível através do uso de heurísticas de escalonamento. Os escalonadores atualmente implementados no MyGrid foram desenvolvidos com a mesma fi losofi a de simplicidade, mesmo assim se preocupando com o desempenho da aplicação. Na versão atual, o MyGrid possui três opções de escalonadores: *Workqueue*, *Workqueue with Replication* e *Storage Affinity*. Os dois primeiros são recomendados para aplicações que não processam grandes quantidades de dados, enquanto o último é mais adequado para aplicações que processam dados de forma intensiva. Como podemos constatar em Paranhos, et al. [17] e no Capítulo 4, apesar de serem heurísticas de escalonamento simples, o desempenho atingido pelo *Workqueue with Replication* e pelo *Storage Affinity*, respectivamente em suas aplicações alvo, é melhor do que outros escalonadores que utilizam heurísticas mais sofi sticadas e dependentes de acesso a informação sobre o ambi-

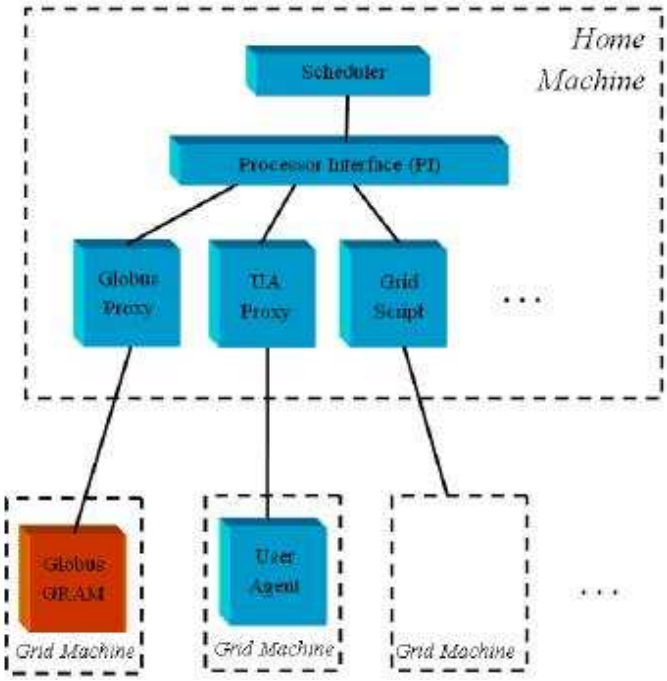


Figura 2.1: Abstrações usadas na construção do MyGrid

ente. Uma vez que *Storage Affinity* utiliza replicação, o bom desempenho é obtido com o gasto extra de recursos do grid. No Capítulo 3 as heurísticas são descritas em detalhes e no Capítulo 4 são mostrados os resultados da análise do impacto de vários aspectos do grid e da aplicação nos escalonamentos efetuados por cada heurística e conseqüentemente suas influências no desempenho da aplicação.

2.2 Aplicações pHd em grids

Aplicações PHD são aplicações compostas por um conjunto de tarefas independentes que têm a necessidade de processar grandes quantidades de dados [10]. As aplicações PHD são particularmente importantes para várias áreas do conhecimento. Facilmente encontramos aplicações fracamente acopladas que necessitam processar grandes quantidades de dados. Podemos citar exemplos nas áreas de bioinformática [1, 37, 38], biologia [39], computação gráfica [22] e processamento de sinais [18], que necessitam processar grandes quantidades de dados e podem ser compostas de tarefas que executam de forma independente.

A característica de independência entre as tarefas das aplicações pHd torna tais aplica-

ções adequadas para executar em grids computacionais. Apesar disso, o requisito de processar grandes quantidades de dados aliado à ampla distribuição dos recursos (em geral conectados por redes de longa distância) influenciam fortemente o desempenho da aplicação. Sendo assim, ambientes de execução e heurísticas de escalonamento para estas aplicações devem considerar as transferências de dados para melhorar o desempenho da aplicação.

É importante citar a existência de uma infraestrutura de armazenamento de dados em larga escala nos moldes de um grid computacional. Portanto, a pesquisa sobre técnicas, mecanismos [40, 41] e protocolos [42] que venham a permitir a manipulação eficiente de enormes quantidades de dados em ambientes com ampla distribuição de recursos heterogêneos está extremamente ligada aos *Data Grids* [43, 44]. Juntamente com essas soluções foram desenvolvidas heurísticas e arquiteturas de escalonamento voltadas para aplicações que processam dados de forma intensiva [14, 45].

Muitas heurísticas de escalonamento para aplicações PHD se baseiam em informações sobre o grid e sobre a aplicação para efetuar as associações entre tarefas e processadores. Contudo, apesar do uso destas informações proporcionar a geração de escalonamentos eficientes, o acesso a informações precisas sobre a carga de utilização de CPU, largura de banda disponível na rede e tempo de execução das tarefas da aplicação nem sempre é possível. A falta de um controle central e a ampla distribuição dos recursos dificulta a implantação de um serviço de monitoração que forneça tais informações. Além disso, políticas de segurança podem restringir o acesso a informação deste tipo. Sendo assim, com o intuito de evitar o uso de informações dinâmicas sobre os recursos e estimativas sobre o tempo de execução da aplicação e ainda assim obter bom desempenho, *Workqueue with Replication* e *Storage Affinity* aplicam estratégias de replicação de tarefas. É possível verificar que o desempenho da aplicação é melhorado e alcança níveis melhores que as heurísticas baseadas em informação. Todavia, a replicação de tarefas utilizada na heurística implica em um consumo extra de recursos. Uma discussão detalhada sobre o comportamento das heurísticas de escalonamento e seus desempenhos é encontrado no Capítulo 4.

2.3 Grids de serviços

A experiência prática adquirida no desenvolvimento dos grids de alto desempenho tornou possível identificar os problemas que dificultam a implantação de uma infraestrutura como esta. A questão central foi determinada como sendo a integração de recursos e serviços distribuídos de forma transparente e eficiente. Percebeu-se então que este requisito era comum às comunidades científica e comercial. Desta forma, a crescente necessidade de cooperação científica entre grupos amplamente distribuídos através do compartilhamento de recursos e serviços, bem como a necessidade de integração de sistemas comerciais impulsionaram a convergência de tecnologias de ambas as comunidades.

Como resultado, os grids evoluíram para utilizar uma abordagem orientada a serviços. Partindo de um conjunto de serviços básicos, como autenticação e transferência de dados, a idéia é a construção de grids baseados na composição de serviços, criando serviços de alta ordem (serviços compostos de outros serviços), que possam usar esse conjunto básico e outros serviços como alicerce.

A convergência de tecnologias das duas comunidades veio através da união de tecnologias e conceitos relacionados aos grids com as tecnologias de *web services* [25], no intuito de realizar a visão da orientação a serviços. A partir disto foi definida uma arquitetura de serviços básicos para a construção de uma infraestrutura de grids computacionais. Esta arquitetura foi denominada *Open Grid Services Architecture* (OGSA) [46, 47]. Esta definição contempla a idéia de interconexão de sistemas e a criação de ambientes virtuais multi-institucionais. Além disso, segundo as definições da OGSA, os recurso que podem ser agregados ao grid são representados por serviços e estes serviços são chamados de *Grid Services* [46]. Os *grid services* são essencialmente *web services* que seguem convenções estabelecidas na especificação da OGSA e suportam interfaces padrões para garantir algumas operações adicionais, como gerenciamento do ciclo de vida do serviço.

As interfaces padrões definidas na OGSA facilitam a virtualização de recursos e serviços. Isso possibilita o uso de vários tipos de recursos de forma transparente. Outra vantagem da virtualização é que interfaces padrões geram um baixo acoplamento entre o cliente e o provedor do serviço. Esse baixo acoplamento facilita a mudança na implementação dos serviços sem causar, necessariamente, mudanças na implementação do cliente, bem como o

caso inverso.

Após a definição do modelo da arquitetura e identificação de serviços básicos através da criação da OGSA foi necessário a especificação do comportamento desses serviços. Sendo assim, o passo seguinte foi a especificação dessa infraestrutura básica de serviços, no intuito de permitir a implementação do modelo da arquitetura definida pela *OGSA*. A nova especificação foi denominada *Open Grid Services Infrastructure* (OGSI) [48] e tem o objetivo de definir as interfaces básicas e os comportamentos de um *grid service* [47]. OGSI é a materialização da arquitetura definida pela OGSA, pois os serviços especificados servem como base para construção dos grids. O *Globus Toolkit 3.0* (GT3) [32] forneceu a primeira implementação da OGSI 1.0 em julho de 2003.

Tendo em vista a nova perspectiva que os *grid services* trouxeram para o desenvolvimento dos grids computacionais, a tendência é que os grids se tornem uma infraestrutura global de computação orientada a serviços, atendendo tanto a comunidade de computação científica de alto desemepenho, quanto a comunidade de computação comercial.

Apesar de ter sido idealizado como uma solução para grids de alto desempenho, *Storage Affinity* pode facilmente ser aplicado em um contexto de grids de serviços como um serviço de escalonamento de aplicações PHD. *Storage Affinity* pode ser visto como um serviço que deve interagir com serviços de armazenamento e gerência de dados, se enquadrando assim na perspectiva futura dos grids computacionais.

Capítulo 3

Escalonamento de aplicações PHD

Neste capítulo discutiremos os problemas relacionados ao escalonamento de aplicações que processam dados de forma intensiva em grids computacionais. Além disso, descreveremos algumas heurísticas de escalonamento voltadas para aplicações BOT.

Como já definimos no Capítulo 1, as aplicações PHD possuem todas as características de uma aplicação BOT. Portanto, devido ao fato das tarefas que formam este tipo de aplicação não necessitarem se comunicar durante a computação, aplicações PHD são adequadas para executar em um grid computacional. Contudo, o escalonamento eficiente destas aplicações não é uma tarefa trivial. Existem duas principais dificuldades no escalonamento de aplicações PHD em grids computacionais. Primeiro, as transferências de dados que ocorrem durante a execução das tarefas afetam bastante o desempenho da aplicação devido à limitação de banda nas conexões entre os sites, os quais estão, em geral, conectados por redes de longa distância. O segundo problema consiste da dificuldade na obtenção de informação precisa sobre o desempenho dos recursos do grid e o tempo de execução das tarefas. Muitas heurísticas de escalonamento além de assumir que estas informações estão disponíveis, consideram as informações precisas. Porém, estas informações são bastante dinâmicas e não se encontram, em geral, disponíveis no instante do escalonamento.

Muito trabalho já foi empregado na pesquisa sobre técnicas de previsão de desempenho de CPU e de rede, bem como o tempo de execução de aplicações [49–53]. Avaliando a qualidade das previsões obtidas nestes trabalhos, conclui-se que efetuar previsões precisas sobre tais informações não é uma tarefa fácil. Além disso, o fato de não haver uma entidade central que controle o grid dificulta a implantação de uma infraestrutura de monitoração de

recursos que forneça estas informações.

Entretanto, sabemos que é possível contornar o problema da obtenção de informações precisas sobre o grid e as aplicações através da replicação de tarefas [17]. Com replicação, há a chance das réplicas serem associadas a processadores mais rápidos do que os processadores que estão executando as tarefas originais. Resultados anteriores mostraram que o desempenho da aplicação pode ser bastante melhorado através da técnica de replicação [8, 17]. Contudo, estes resultados não consideram aplicações que processam dados de forma intensiva. Como será mostrado no Capítulo 4, o uso de replicação sem levar em conta as transferências de dados resulta em um baixo desempenho para as aplicações PHD.

É claramente possível melhorar o desempenho de aplicações PHD evitando as transferências de dados desnecessárias. Em particular, transferências desnecessárias podem ser identificadas e evitadas quando a aplicação reutiliza dados (isso pode ocorrer tanto entre execuções, quanto durante uma mesma execução). Reutilização de dados é uma característica comum entre as aplicações PHD [4, 54].

Existem heurísticas conhecidas que exploram a reutilização de dados e a capacidade das conexões entre os processadores para promover o desempenho da aplicação [13, 14]. Porém, estas heurísticas dependem de informações sobre o ambiente (i.e. utilização de CPU, largura de banda disponível na rede e tempo de execução das tarefas).

Pode-se portanto observar que a dificuldade em obter informações dinâmicas e o impacto das transferências de dados foram individualmente atacados. Neste capítulo, discutiremos duas heurísticas de escalonamento que tratam estes problemas de forma separada, *Workqueue with Replication* (WQR) [17] e *XSufferage* [14]. Além disso, apresentaremos *Storage Affinity* [10], uma nova heurística que aborda, de forma conjunta, os dois problemas referentes ao escalonamento de aplicações PHD descritos anteriormente. Porém, antes disso definiremos formalmente o problema de escalonamento abordado neste trabalho.

3.1 Modelo e notação

Nesta seção apresentamos o modelo que é considerado nas simulações e a notação utilizada no restante do trabalho. Consideramos o escalonamento de um seqüencia de *jobs*[1] em um

[1] Os termos job e aplicação são utilizados aqui com o mesmo significado.

grid computacional.

3.1.1 O Grid

Um grid G é formado por uma coleção de sites. Cada site é composto de um conjunto de processadores e de um conjunto de servidores de dados. Os processadores têm a capacidade de executar tarefas, enquanto os servidores de dados desempenham a função de armazenar os dados de entrada necessários para a execução das tarefas, bem como os dados de saída gerados. De maneira mais formal, temos:

$$G = \{site_1, \ldots, site_g\}, g > 0, \text{ e } site_i = P_i \cup S_i,$$

onde P_i é um conjunto não-vazio de processadores no $site_i$ e S_i é o conjunto de servidores de dados do $site_i$. Assumimos que os conjuntos de recursos localizados em cada site são disjuntos, ***i.e.*** $\forall i, j, i \neq j, site_i \cap site_j = \varnothing$.

Processadores pertencentes ao mesmo site estão conectados através de uma rede local, enquanto os processadores que estão localizados em diferentes sites estão conectados por uma rede de longa distância. Esta consideração implica em que os processadores e servidores de dados dentro de um mesmo site se comunicam através de conexões mais rápidas, ao contrário das conexões entre recursos de sites diferentes que apresentam limitação de banda e alta latência.

Defi nimos dois conjuntos que representam todos os processadores (P_G) e todos os servidores de dados (S_G) presentes em um grid G. Desta forma, temos:

$$P_G = \bigcup_{1 \leq i \leq |G|} P_i, \quad \text{e} \quad S_G = \bigcup_{1 \leq i \leq |G|} S_i.$$

O usuário inicia a execução das aplicações a partir de uma *máquina base* (P_{home}), esta máquina não faz parte do grid, ou seja, $p_{home} \notin G$. Além disso, nós assumimos que antes da primeira execução de uma aplicação, todos os dados de entrada são armazenados em um sistema de arquivos local da máquina base representado por S_{home}. Uma ilustração desse ambiente de execução pode ser vista na Figura 3.1.

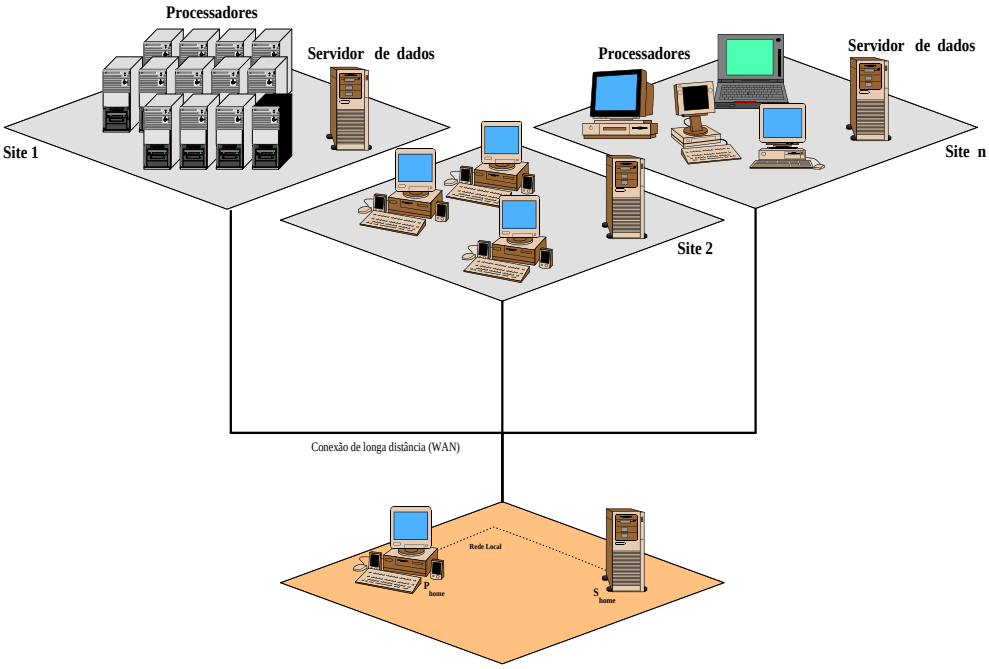


Figura 3.1: O modelo do sistema

3.1.2 A aplicação e os dados de entrada

O job J_j, $j > 0$, é a $j-$ésima execução da aplicação J. Um job é composto por uma coleção não-vazia de tarefas, onde cada tarefa é defi nida por dois conjuntos de dados: o conjunto de entrada e o de saída. Formalmente:

$$J_j = \{t_1^j, \ldots, t_n^j\}, n > 0, \text{ e } t_t^j = (I, O), I \cup O \neq \varnothing,$$

onde $t_t^j.I$ e $t_t^j.O$ são os conjuntos de dados de entrada e saída da tarefa t_t^j, respectivamente.

Para cada servidor de dados $S_i, S_i \in S_G$, temos que $ds_j(S_i)$ é o conjunto dos elementos de dados que estão armazenados em S_i antes da execução do $j-$ésimo job iniciar, e $ds(S_{home})$ é o conjunto dos elementos de dados que estão armazenados na máquina base. Defi nimos D_j como sendo o conjunto de todos os elementos de dados que podem ser usados como entrada na $j - sima$ execução do job J. D_j é dado por:

$$D_j = ds(S_{home}) \cup \left\{ \bigcup_{S_i \in S_G} ds_j(S_i) \right\}.$$

Temos que $t_t^j.I \subseteq D_j$ e após a execução do $j-$ésimo job, o conjunto dos elementos de dados disponíveis D_{j+1} é dado pela união de D_j com todos os elementos de dados de saída

que resultaram da execução do job J_j:

$$D_{j+1} = D_j \cup \left\{ \bigcup_{1 \leq t \leq |J_j|} t_t^j.O \right\}.$$

Defi nimos também que o conjunto de entrada de dados de toda a aplicação como sendo a união dos conjuntos de dados de entrada de todas as tarefas quem formam a aplicação. Isto pode ser expresso pelo seguinte:

$$J_j.I = \bigcup_{k=1}^{|J_j|} t_k^j.I.$$

3.1.3 Métricas de desempenho

Um escalonamento Σ_j de um job J_j é composto pelo escalonamento de cada uma das tarefas que formam J_j. O escalonamento de uma tarefa particular t_t^j de J_j especifi ca o processador no qual a tarefa t_t^j irá executar. Note que é possível que um mesmo processador seja associado a mais de uma tarefa em momentos distintos. Formalmente,

$$\Sigma_j = \{p_1^j, \ldots, p_n^j\}, n = |J_j|, p_t^j \in P_G, 0 \leqslant t \leqslant n.$$

Assumimos que uma tarefa pode apenas acessar os servidores de dados do mesmo site do processador onde está executando. Uma vez que nesse modelo as conexões dentro de cada site são muito mais rápidas que as conexões entre sites diferentes, o tempo gasto com transferências de dados entre um processador qualquer $p \in P_i$ e qualquer servidor de dados $s \in S_i$ é desprezível para o tempo de execução das tarefas. Sendo assim, a escolha do servidor de dados dentro de cada site pode ser feita de forma aleatória sem penalizar o desempenho da aplicação. Assim, após t_t^j ser executada em $p_t^j \in site_i$, $s_t^j \in S_i$ terá armazenado todos os elementos de dados do conjunto $t_t^j.O$. Conseqüentemente, se nem todos os elementos de dados do conjunto $t_t^j.I$ estiverem armazenados em algum servidor $S'_{t'} \in S_i$, os elementos de dados que faltarem devem ser transferidos de S_{home} para s_t^j antes da execução de t_t^j iniciar em p_t^j.

Os algoritmos que são analisados neste trabalho têm como objetivo a redução do tempo total de execução da aplicação. Sendo assim, a avaliação da efi ciência de cada heurística de escalonamento será efetuada através da verifi cação do tempo de execução da aplicação. O

tempo total de execução da aplicação, normalmente referenciado como o *makespan* [55] da aplicação, é definido como o intervalo de tempo entre o momento em que a primeira tarefa do job é iniciada e o primeiro instante em que todas as tarefas finalizaram suas execuções. Portanto, a métrica de desempenho adotada é baseada na função objetivo dos algoritmos de escalonamento, a redução do *makespan* da aplicação.

Seja T_i o instante de tempo no qual a primeira tarefa da aplicação iniciou sua execução e T_f o instante no qual a última tarefa finalizou a execução. Desta forma, temos que o makespan da aplicação é dado por $T_f - T_i$. Veja na Figura 3.2 uma ilustração para a definição de *makespan*, considerando um job qualquer formado por n tarefas, $J_x = \{t_1^x, ..., t_n^x\}$, onde t_1^x foi a primeira a iniciar a execução e a última a finalizar foi a tarefa t_n^x.

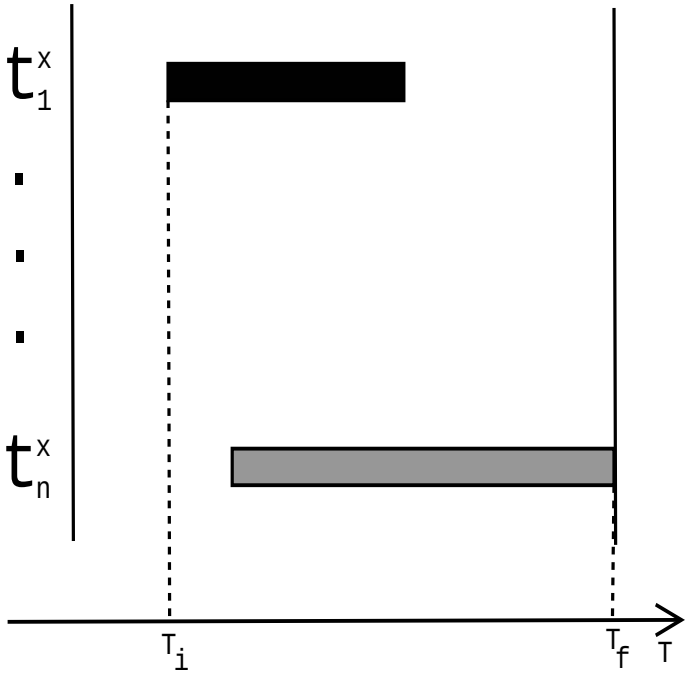


Figura 3.2: Ilustração do cálculo do *makespan* de um job

3.2 Reutilização de dados

É muito comum em aplicações PHD a presença de algum padrão de reutilização de dados [1, 4–6, 22, 37–39, 54] que pode ser explorado para melhorar o desempenho da aplicação. Classificamos os padrões de reutilização em dois tipos. Os tipos variam de acordo com

a forma de compartilhamento de dados entre as tarefas. Assim, temos dois tipos básicos: *inter-job* e *inter-task*. O primeiro tipo ocorre quando um job usa dados já usados ou produzidos por outro job que executou anteriormente, enquanto o segundo padrão de reutilização ocorre em aplicações onde as tarefas compartilham dados de seus conjuntos de dados de entrada. Mais formalmente, o padrão de reutilização de dados *inter-job* ocorre se a seguinte relação for verdadeira:

$$(j < k) \wedge ((J_j.I \cup J_j.O) \cap J_k.I \neq \varnothing).$$

Como exemplo de um padrão de reutilização *inter-job* podemos citar uma aplicação de visualização [4, 6]. Em geral, esse tipo de aplicação é composto por tarefas que processam partições distintas do conjunto total de entrada da aplicação. Estas tarefas são executada inúmeras vezes com a modificação de uma pequena porção dos parâmetros de entrada (ex. zoom, ponto de vista). Sendo assim, entre as inúmeras execuções, a aplicação irá aproveitar os dados previamente transferidos pelas execuções anteriores.

Por outro lado, o padrão de reutilização de dados *inter-task* ocorre em uma aplicação se a seguinte relação for verdadeira:

$$\bigcap_{t=1}^{|J_j|} t_t^j.I \neq \varnothing.$$

Nesse caso, um exemplo de aplicação seria uma aplicação de busca de padrões em um banco de dados textual [1]. Essa aplicação poderia ser estruturada de forma que todas as suas tarefas utilizem o mesmo arquivo de entrada (o banco de dados textual) e uma chave para a busca. Assim, durante uma mesma execução da aplicação as tarefas compartilharam parte dos seus conjuntos de dados de entrada. A melhora no desempenho da aplicação pode ser alcançada evitando a transferência dessa porção de dados em comum entre as tarefas. Essa técnica faz parte da estratégia da heurística *Storage Affinity* para melhorar o desempenho das aplicações PHD executando em grids.

Vale ressaltar que é possível que algumas aplicações apresentem padrões de reutilização de dados híbridos. Como veremos a seguir, para um escalonamento eficiente das aplicações PHD é importante que a heurística de escalonamento considere estes padrões de reutilização no intuito de evitar transferências desnecessárias de grandes quantidades de dados.

Uma aplicação de visualização de dados [4, 6] sugere um padrão de reutilização do tipo *inter-job*. Em geral, uma aplicação de visualização é executada inúmeras vezes sobre o

mesmo conjunto de dados, sendo modifi cados apenas alguns parâmetros de sua entrada entre as execuções (zoom, ponto de vista, etc). Cada tarefa da aplicação recebe uma fatia distinta do conjunto de dados de entrada e os parâmetros da visualização.

Por outro lado, uma aplicação de busca de padrões texto pode ser estruturada de forma que todas as tarefas recebam como parâmetro uma seqüência de caracteres diferente (alguns bytes) e um mesmo banco de dados textual. Nesse caso, durante a execução da aplicação as tarefas estarão compartilhando dados entre si, o que caracteriza uma aplicação que apresenta um padrão de reutilização de dados *inter-task*.

3.3 Algoritmos

3.3.1 Workqueue with Replication

A heurística de escalonamento *Workqueue with Replication* (WQR) [17] foi concebida para solucionar o problema da obtenção de informações precisas sobre a aplicação e a carga de utilização dos recursos do grid. No Algoritmo 1 vemos que em sua fase inicial o WQR é similar a heurística de escalonamento *Workqueue* tradicional. As tarefas são enviadas para execução nos processadores que se encontram disponíveis em uma ordem aleatória. Quando um processador fi naliza a execução de uma tarefa, este recebe uma nova tarefa para processar.

As heurísticas WQR e *Workqueue* passam a diferir no momento em que um processador se torna disponível e não há mais nenhuma tarefa pendente para executar. Neste momento, *Workqueue* já fi nalizou seu trabalho e apenas aguarda a fi nalização de todas as tarefas do job. Porém, o WQR inicia sua fase de replicação para as tarefas que ainda estão executando. Note que, na fase de replicação, quando uma tarefa original fi naliza sua execução primeiro que suas réplicas, estas são interrompidas. Caso contrário, quando alguma réplica fi naliza primeiro, a tarefa original e as demais réplicas dela são interrompidas.

WQR alcança bons níveis de desempenho sem a utilização de informação dinâmica sobre os processadores, conexões de rede ou mesmo tempo de execução das tarefas, considerando aplicações que não processam dados de forma intensiva [17]. Há porém, um efeito colateral no uso da replicação. As réplicas que são interrompidas ocasionam um desperdício de ciclos de CPU, pois o resultado de sua computação interrompida não será útil. Devido ao fato da

heurística WQR não considerar os padrões de reutilização de dados durante o escalonamento, um desempenho ruim para aplicações PHD pode ser verifi cado na análise que efetuamos e será apresentada no Capítulo 4.

```
Entrada : G, J_j
Saída   : Σ_j
while J_j ≠ ∅ do
    k ← 1
    while (k ⩽ |J_j|) and (∃ p ∈ G | p is not busy) do
        while (m ⩽ |G|) and (some task-to-host assignment does not occur) do
            if (∃ p ∈ site_m | p is not busy) then
                p_chosen ← getFirstAvaiableProcessorAt(site_m)
                assign(t_k, p_chosen)
                J_j ← J_j − {t_chosen}
            end
            m ← m + 1
        end
        k ← k + 1
    end
    if ∀ p ∈ G | p is busy then
        waitForATaskCompletionEvent()
    end
end
J_r ← getAllRunningTasks()
while (J_r ≠ ∅) do
    p_chosen ← nil
    while (k ⩽ |J_r|) and (∃ p ∈ G | p is not busy) do
        while (m ⩽ |G|) and (some task-to-host assignment does not occur) do
            if (∃ p ∈ site_m | p is not busy) then
                p_chosen ← getFirstAvaiableProcessorAt(site_m)
                {createReplica(task) is a function that create a replica to the specifi ed task.}
                {That function updates the task replication degree, denoted by d.}
                replica^d_{t_kr} ← createReplica(t_kr)
                assign(replica^d_{t_kr}, p_chosen)
            end
            m ← m + 1
        end
        k ← k + 1
    end
    if (∀ p ∈ G | p is busy) then
        waitForATaskCompletionEvent()
        killAllReplicasOfTheCompletedTask()
    end
    J_r ← getAllTasksRunning()
end
```

Algoritmo 1: Workqueue with Replication

3.3.2 XSufferage

XSufferage [14] é uma heurística de escalonamento que se baseia nas informações sobre o desempenho dos recursos e para associar tarefas aos processadores. Esta heurística aborda o impacto das grandes transferências de dados nas aplicações PHD executando em grids computacionais através da exploração da reutilização de dados. *XSufferage* é uma extensão de heurística de escalonamento *Sufferage* [56]. A idéia básica da heurística *Sufferage* é determi-

nar quanto cada tarefa seria prejudicada (“sofreria”) se não fosse escalonada no processador que a executaria de forma mais eficiente. Portanto, *Sufferage* prioriza as tarefas de acordo com o valor que mede o prejuízo (“sofrimento”) de cada tarefa. Este valor é denominado de *sufferage* e definido como a diferença entre o melhor e o segundo melhor tempo de execução previsto para a tarefa, considerando todos os processadores do grid.

Entrada : G, J_j

Saída : Σ_j

while $J_j \neq \varnothing$ **do**
- $sufferage_{max} \leftarrow 0$
- $CompletionTime_{min_1} \leftarrow \infty$
- $CompletionTime_{min_2} \leftarrow \infty$
- $p_{chosen} \leftarrow nil$
- $t_{chosen} \leftarrow nil$
- **for** $k \leftarrow 1$ ***to*** $|J_j|$ **do**
  - **for** $m \leftarrow 1$ ***to*** $|G|$ **do**
    - **if** $(getBestSiteLevelCT(t_k^j, site_m) \leqslant CompletionTime_{min_1})$ **then**
      - $CompletionTime_{min_2} \leftarrow CompletionTime_{min_1}$
      - $CompletionTime_{min_1} \leftarrow getBestSiteLevelCT(t_k^j, site_m)$
    - **end**
  - **end**
  - **if** $(sufferage_{max} \leqslant (CompletionTime_{min_2} - CompletionTime_{min_1}))$ **then**
    - $sufferage_{max} \leftarrow CompletionTime_{min_2} - CompletionTime_{min_1}$
    - $t_{chosen} \leftarrow t_k^j$
    - $p_{chosen} \leftarrow p_{t_k^j}$
  - **end**
- **end**
- **assign**(t_{chosen}, p_{chosen})
- $J_j \leftarrow J_j - \{t_{chosen}\}$

end

Algoritmo 2: Heurística de escalonamento XSufferage

A principal diferença entre *Sufferage* e *XSufferage* é o método usado para calcular o valor do *sufferage*. Na heurística *XSufferage* há o conceito de site, o que não ocorre em *Sufferage*. Portanto, em *XSufferage* inicialmente é determinado para cada tarefa o seu melhor tempo de execução em cada site, estes valores são chamados de *site-level completion times*. Em seguida é calculado o *site-level sufferage* para cada tarefa, isso é feito calculando a diferença entre os dois melhores valores encontrado para *site-level completion time* da tarefa. Sendo assim, a tarefa que apresentar o maior *site-level sufferage* terá prioridade e será escalonada no site que a executaria mais rápido. Além da consideração dos sites, há outro aspecto distinto entre estas heurísticas. *XSufferage* considera a transferência dos dados de entrada da tarefa durante o cálculo dos tempos de execução da tarefa. Isso implica em uso das informações usadas também por *Sufferage* mais a largura de banda disponível na rede que conecta os recursos, diferente de *Sufferage* que necessita apenas das informações relacionadas a CPU e

o tempo estimado de execução da tarefa.

No Algoritmo 2, vemos que *XSufferage* recebe como entrada um job J_j e um grid G. O escalonamento é iniciado com uma iteração sobre o conjunto J_j. Nesta iteração a tarefa t_t^j que tem o maior valor de *site-level sufferage* é determinada. Esta tarefa é associada a um processador que apresentou a melhor previsão para o tempo total de execução da tarefa. Vale relembrar que, a previsão para o tempo total de execução da tarefa é determinado somando as previsões do tempo de transferência e do tempo de execução. Portanto, a tarefa não será associada necessariamente ao processador mais rápido do grid, mas àquele que estiver melhor conectado e/ou já possuir os dados armazenados no site. Esta operação é repetida até que todas as tarefas em J_j sejam escalonadas.

A definição do *site-level sufferage* permite que a localização dos dados de entrada da tarefa seja considerada no momento da alocação de recursos. O efeito esperado é a redução do impacto das transferências de dados desnecessárias no tempo total de execução da aplicação. A avaliação de *XSufferage* mostra que evitando transferências desnecessárias de dados o desempenho da aplicação é melhorado [14]. Porém, os cálculos para a determinação dos valores do *site-level sufferage* são baseados no conhecimento sobre a carga de CPU, largura de banda da rede e tempos de execução das tarefas. Em geral, não é fácil obter este tipo de informação. Em alguns casos, dependendo das restrições administrativas sobre os recursos que compõem o grid, tais informações podem nem mesmo estar disponíveis.

3.3.3 Storage Affinity

Na Seção 3.2 os padrões de reutilização de dados foram classificados em dois tipos básicos, *inter-job* e *inter-task*. *Storage Affinity* foi concebido no intuito de explorar ambos os padrões de reutilização de dados para melhorar o desempenho de aplicações PHD. Para tanto, foi definida um método de priorização de tarefas utilizando um conceito chamado de *storage affinity* (afinidade). O valor da *afinidade* entre uma tarefa e um site determina *quão próximo* do site esta tarefa está. A semântica do termo *próximo* está associada à quantidade de bytes da entrada da tarefa que já está armazenada remotamente em um dado site, ou seja, quanto mais bytes da entrada da tarefa já estiver armazenado no site, mais próximo a tarefa estará do site, pois possui mais *storage affinity*. Assim, o valor do *storage affinity* de uma tarefa com um site *é o número de bytes pertencentes à entrada da tarefa que já estão armazenados*

no site. Formalmente, o valor de *storage affinity* entre t_t^j e $site_i$ é dado por:

$$SA(t_t^j, site_i) = \sum_{d \in (t_t^j . I \cap ds_j(S_i))} |d|$$

onde, $|d|$ representa o número de bytes de um elemento de dados d.

Note que *Storage Affinity* utiliza tão somente as informações sobre o tamanho e a localização dos arquivos de entrada. Argumentamos que tais informações podem estar disponíveis no instante do escalonamento sem dificuldade e perda de precisão. Por exemplo, as informações relacionadas aos dados de entrada de uma tarefa podem ser obtidas através de requisições aos servidores de dados. Caso estes servidores não sejam capazes de responder às requisições sobre *quais elementos* de dados eles armazenam e *qual o tamanho* de cada elemento de dado, uma implementação alternativa da heurística *Storage Affinity* poderia facilmente manter um histórico das informações sobre as transferências efetuadas para cada tarefa e assim possuir tais informações.

Além de aproveitar a reutilização dos dados, *Storage Affinity* também trata a dificuldade na obtenção de informações dinâmicas sobre o grid e informações sobre o tempo de execução das tarefas da aplicação. Para resolver este problema, *Storage Affinity* efetua *replicação de tarefas*. A idéia é que as réplicas tenham a chance de serem submetidas a processadores mais rápidos do que aqueles associados às tarefas originais. Com isso espera-se uma redução do tempo total de execução da aplicação.

No Algoritmo 3 é apresentada a heurística de escalonamento *Storage Affinity*. Note que a mesma é dividida em duas fases. Na primeira fase, Storage Affinity associa cada tarefa $t_t^j \in J_j$ a um processador $p \in G$. A ordem de escalonamento é determinada através do cálculo do valor de *storage affinity* das tarefas. Após determinar as afinidades, a tarefa que apresentar o maior valor para *storage affinity* é escalonada em um processador do site com o qual a tarefa apresentou maior afinidade. A segunda fase consiste da replicação de tarefas. Esta fase inicia quando não há mais tarefas aguardando para executar e há, pelo menos, um processador disponível. Uma réplica *pode* ser criada para qualquer tarefa em execução, porém nesta fase há um critério e uma ordem de prioridade para criação de réplicas. Considerando que o grau de replicação de uma tarefa é o número de réplicas criadas para esta tarefa, então ao iniciar a fase de replicação, os seguintes critérios são verificados na escolha de qual tarefa deve ser replicada: i) a tarefa deve estar executando e ter um valor de afinidade positivo, ou

seja, alguma porção de sua entrada já deve estar presente no site de algum dos processadores disponíveis no momento; ii) o grau de replicação corrente da tarefa deve ser o menor entre as tarefas que atendem o critério (i); e iii) a tarefa deve apresentar o maior valor de afinidade entre as tarefas que atendem o critério *(ii)*.

```
Entrada : G, J_j
Saída   : Σ_j
while (J_j ≠ ∅) do
    affinity_max ← affinity_current ← 0; p_chosen ← t_chosen ← nil; k ← m ← 1
    while (t ⩽ |J_j|) do
        while (i ⩽ |G|) do
            if (∃ p ∈ P_i | p is not busy) then
                if (affinity_max ⩽ SA(t_t^j, site_i)) then
                    affinity_max ← SA(t_t^j, site_i)
                    p_chosen ← getFirstAvaiableProcessorAt(site_i)
                    t_chosen ← t_t^j
                end
            end
            i ← i + 1
        end
        t ← t + 1
    end
    assign(t_chosen, p_chosen)
    J_j ← J_j − {t_chosen}
    if (∀ p ∈ P_G | p is busy) then
        waitForATaskCompletionEvent()
    end
end
J_r ← getAllRunningTasks()
while (J_r ≠ ∅) do
    affinity_max ← 0; p_chosen ← t_chosen ← nil; t ← i ← 1;
    J_r ← J_r − {t_t^r ∈ J_r | ∀ site_i ∈ G, SA(t_t^r, site_i) > 0}
    J_r ← J_r − {t_t^r ∈ J_r | getReplicationDegree(t_t^r) = replicationDegree_min }
    while (t ⩽ |J_r|) do
        while (i ⩽ |G|) do
            if (∃ p ∈ P_i | p is not busy) then
                if (affinity_max < SA(t_t^r, site_i)) then
                    affinity_max ← SA(t_t^r, site_i)
                    t_chosen ← t_t^r
                    p_chosen ← getFirstAvaiableProcessorAt(site_i)
                end
            end
            i ← i + 1
        end
        t ← t + 1
    end
    if (t_chosen ≠ nil) then
        replica^d_{t_chosen} ← createReplica(t_chosen)
        assign(replica^d_{t_chosen}, p_chosen)
    else
        waitForATaskCompletionEvent()
    end
    if (∀ p ∈ P_G | p is busy) then
        waitForATaskCompletionEvent()
        killAllReplicasOfTheCompletedTask()
    end
    J_r ← getAllTasksRunning()
end
```

Algoritmo 3: Heurística de escalonamento Storage Affinity

De forma semelhante ao WQR, quando uma tarefa completa sua execução as outras réplicas da tarefa são interrompidas. O algoritmo finaliza quando todas as tarefas que estão executando completam. Até isto ocorrer, a replicação de tarefas continua. Caso seja pertinente uma implementação particular de *Storage Affinity* pode limitar o grau de replicação máximo no intuito de conter o desperdício de recursos ocasionado pela replicação. Nas simulações que discutiremos a seguir (Capítulo 4) utilizamos replicação infinita.

Capítulo 4

Avaliação de desempenho

Na avaliação de desempenho a heurística *Storage Affinity* foi comparada às heurísticas WQR e *XSufferage*. A decisão de comparar a nova abordagem com estas heurísticas vem do fato que WQR representa uma solução eficiente para escalonamento de aplicações sem utilizar informação dinâmica, enquanto *XSufferage* provê um escalonamento eficiente para aplicações que processam dados de forma intensiva. A metodologia de avaliação do desempenho baseou-se na execução de simulações da execução de aplicações PHD.

Uma vez que o desempenho de um escalonamento é bastante influenciado pela carga de utilização dos recursos e as características da aplicação [57–59], as simulações foram desenvolvidas para cobrir uma variedade razoável de cenários. Os cenários variam na heterogeneidade do grid e da aplicação, bem como na relação entre o número médio de tarefas e de processadores do grid. Portanto, é esperado não apenas identificar qual escalonador é mais eficiente, mas também entender como os diferentes fatores influenciam o desempenho dos escalonadores.

4.1 Simulando um Grid Computacional

Cada tarefa possui um custo computacional. Este custo expressa quanto tempo a tarefa executaria em um *processador-referência* em regime dedicado. Como o grid pode ser formado por processadores que foram adquiridos em diferentes épocas, é uma tendência que os grids sejam bastante *heterogêneos* (i.e. a velocidade dos processadores apresenta uma grande variação). Sabendo que processadores podem executar em várias velocidades, definimos que o

valor da velocidade do *processador-referência* é igual a 1 ($\nu = 1$). Assim, um processador dedicado e com velocidade igual a 2 ($\nu = 2$) executa uma tarefa de 100 segundos em 50 segundos.

Com o propósito de investigar o impacto da heterogeneidade do grid no escalonamento de aplicações, consideramos quatro níveis de heterogeneidade, estes níveis estão listados na Tabela 4.1, onde $U(x, y)$ representa uma distribuição uniforme no intervalo $[x, y]$. Assim, para heterogeneidade $1x$, os processadores sempre possuirão a propriedade $\nu = 10$, ou seja, o grid é *homogêneo*. Por outro lado, se a heterogeneidade do grid for $8x$, o máximo de heterogeneidade, isto significa que a velocidade do processador mais rápido será 8 vezes maior que a do mais lento. Vale ressaltar que, em todos os casos, a média das velocidades dos processadores que formam o grid tem o valor igual a 10.

Heterogeneidade do Grid	**Distribuição da velocidade dos processadores**
$1x$	$U(10, 10)$
$2x$	$U(6.7, 13.4)$
$4x$	$U(4, 16)$
$8x$	$U(2.2, 17.6)$

Tabela 4.1: Heterogeneidade do Grid e distribuições de velocidades dos processadores.

A velocidade total do grid é determinada pela soma das velocidades de todos os seus processadores. Para todas as simulações a velocidade do grid foi fixada no valor $1,000$. Uma vez que a velocidade dos processadores depende da heterogeneidade do grid (ver Tabela 4.1), um grid é "construído" pela adição de um processador por vez, até que a velocidade do grid alcance o valor $1,000$. Como o grid é composto de dois ou mais sites, os processadores são distribuídos entre os sites que compõem o grid em igual proporção. Semelhante ao modelo utilizado nas simulações em Casanova *et al.* [14], no nosso modelo, o número de sites varia de acordo com a seguinte distribuição uniforme: $U(2, 12)$.

Por questões de simplicidade e para manter o foco do trabalho, políticas de gerenciamento de espaço em disco não são consideradas para os servidores de dados, ou seja, assume-se que os servidores possuem espaço em disco ilimitado. Como especificado anteriormente no Capítulo 3, as transferências de dados entre um processador $p \in P_i$ e um servidor de dados $s \in S_i$, onde $site_i = P_i \cup S_i$, não são consideradas.

Desde que os recursos dos grids são compartilhados por várias aplicações, é bastante co-

mum que nem os processadores nem a rede se encontrem em regime dedicado todo o tempo. Portanto, para simular a variação na disponibilidade dos processadores e da rede usamos *traces* obtidos através da monitoração de recursos reais com NWS [60]. Por exemplo, um processador de $\nu = 1$ e disponibilidade $= 50\%$ executa uma tarefa de 100 segundos em 200 segundos. Obviamente, este cálculo assume uma disponibilidade *constante* de 50%.

Entretanto, na prática a disponibilidade do processador é uma função do tempo, ou seja, a carga do processador pode variar durante a própria execução da tarefa. Portanto, sendo τ_i o instante no qual a tarefa inicia sua execução, τ_f o instante em que a tarefa finaliza, C o custo da tarefa (tempo de execução no *processador-referência* em regime dedicado), $A_v(\tau)$ uma função que fornece a disponibilidade do processador num dado instante e ν a velocidade do processador que está executando a tarefa, temos que o cálculo do *tempo de execução efetivo* (τ_e) da tarefa seria dado por:

$$\nu \int_{\tau_i}^{\tau_f} A_v(\tau) d\tau = C \implies \tau_e = (\tau_f - \tau_i) \tag{4.1}$$

Porém, uma vez que o tempo no simulador avança de forma discreta, o cálculo de τ_e é efetuado através de um somatório ao invés de uma integral. Logo:

$$\nu \Sigma_{i=1}^{n} A_v(\tau)_i = C \implies \tau_e = (\tau_f - \tau_i) \tag{4.2}$$

onde n é um número finito de parcelas existentes no intervalo $[\tau_i, \tau_f]$.

Outro aspecto da modelagem de recursos é a rede. A rede é modelada como sendo um único canal compartilhado entre a máquina base e o grid. A capacidade máxima disponível para a aplicação nessa conexão foi definida como sendo $1Mbps$. Tomando como exemplo as estatísticas de tráfego de saída da conexão UFCG(PB) $\mapsto$ RNP(SP) entre Jan/2003 e Jan/2004 é possível ver que a disponibilidade oscila em torno de $1Mbps$ (área verde da Figura 4.1). Para simular essa oscilação são utilizados *traces* NWS [60] que servem como entrada para o simulador que utiliza um procedimento semelhante ao descrito na Equação 4.2 para calcular o tempo efetivo de uma transferência de dados.

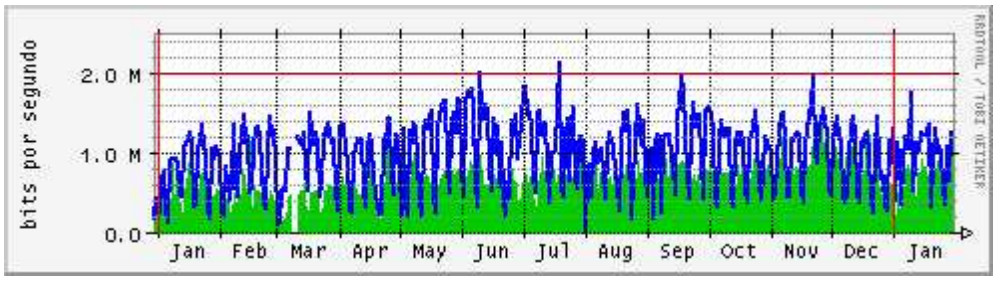


Figura 4.1: Utilização da conexão UFCG(PB) $\mapsto$ RNP(SP). VERDE = Tráfego de Saída. AZUL = Tráfego de Entrada. Fonte: http://www.rnp.br/ceo/trafego/estatisticas

4.2 Simulando as Aplicações PHD

Em aplicações PHD, o tempo de execução da aplicação é tipicamente uma função do tamanho dos dados de entrada. A explicação para isso é bastante simples. Quanto mais dados devem ser processados, mais tempo é necessário para processá-los. De fato, existem algumas aplicações PHD onde o custo das tarefas é *totalmente* determinado pelo tamanho dos dados de entrada da tarefa. Por exemplo, isso ocorre para uma aplicação de visualização científi ca, onde os dados de entrada devem ser processados integralmente para que a saída possa ser produzida [4]. Todavia, existem outras aplicações onde o custo é apenas influenciado pelo tamanho dos dados de entrada. Este é o caso de uma aplicação de busca de padrões, onde o tamanho da entrada determina apenas um limite máximo de tempo de execução (pior caso). Feitas estas considerações, ambos os tipos de aplicações foram simuladas, com o intuito de avaliar o comportamento das heurísticas diante da variação deste aspecto da aplicação.

O tamanho total dos dados de entrada de cada aplicação foi fi xado em $2GBytes$. Como há uma relação *tempo de execução / nº de bytes* que determina o custo de uma tarefa, dados experimentais [4] foram utilizados para converter o tamanho da entrada das tarefas em seus respectivos custos. O mesmo fator de conversão (0.001602171 segundos/KByte) foi utilizado para ambos os tipos de aplicação. Porém, há uma diferença no método de conversão utilizado para cada tipo de aplicação. No caso da aplicação de visualização, a determinação do custo computacional da tarefa é direta, necessitando apenas uma multiplicação do fator pelo tamanho em bytes da entrada da tarefa. Já para a aplicação de busca de padrões, o fator é usado para determinar apenas o limite máximo ($UpperBound$) do custo da tarefa. O custo real é determinado usando a distribuição uniforme $U(1, UpperBound)$.

Outro aspecto analisado foi a influência, no desempenho de um escalonamento, da relação entre o número médio de tarefas do job e o número de processadores no grid. Note que, neste modelo, quando os tamanhos da aplicação e do grid são fi xos, a relação citada acima é inversamente proporcional ao tamanho médio dos dados de entrada das tarefas que compõem a aplicação, i.e. a *granularidade da aplicação*. Assim, um incremento na granularidade da aplicação implica em uma redução do número médio de tarefas que formam o job. Nas simulações foram considerados três grupos de aplicação que são defi nidos pelos seguintes valores de granularidade: $3MBytes$, $15MBytes$ e $75MBytes$. Vale relembrar que há uma relação entre o tamanho dos dados de entrada da tarefa e seu custo computacional, sendo assim, é possível converter a granularidade em função do tamanho dos dados para uma função do custo computacional. Nesse caso: 5000, 25000 e 125000 segundos.

O tamanho das tarefas que formam a aplicação também pode variar. Logo, para simular esta variação, introduzimos um fator de *heterogeneidade de aplicação*. O fator de heterogeneidade determina quanto os tamanhos dos dados de entrada das tarefas variam, conseqüentemente o tempo de processamento também varia. Os tamanhos dos dados de entrada são gerados através de uma distribuição uniforme que contempla os conceitos da granularidade e heterogeneidade da aplicação. A distribuição usada é $U(AverageSize \times (1 - \frac{H_a}{2}), AverageSize \times (1 + \frac{H_a}{2}))$, onde $AverageSize \in \{3MBytes, 15MBytes, 75MBytes\}$ e $H_a \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$.

4.3 Ambiente de Execução das Simulações

Os resultados apresentados a seguir foram extraídos da análise de 3.000 simulações, sendo 1.500 considerando a aplicação de visualização e a outra metade considerando a aplicação de busca de dados. Em cada simulação eram simuladas as três heurísticas de escalonamento. Além disso, em todas as simulações foi considerada uma aplicação com padrão de reutilização de dados *inter-job* numa seqüencia de seis execuções.

A ferramenta de simulação foi desenvolvida a partir do *Simgrid toolkit* [61]. O *Simgrid toolkit* fornece as funcionalidades básicas para a simulação de aplicações distribuídas em grids computacionais. Para desenvolver o simulador foram necessárias algumas adaptações no código original do *Simgrid*. Estas adaptações consistiram da inclusão de algumas abstra-

ções para tornar possível a simulação do escalonamento de aplicações PHD. Sendo assim, foram implementadas estruturas e funções para modelar o comportamento dos servidores de dados e dos elementos de dados manipulados pelas tarefas. Acreditamos que essas extensões efetuadas no *Simgrid toolkit* sejam úteis em futuras investigações envolvendo a execução de aplicações PHD em grids computacionais.

O ciclo de execução e análise dos resultados das simulações ocorreu durante um período aproximado de 30 dias. Cada simulação executou em média 4.5 horas, logo seriam necessários, aproximadamente 564 dias para executar em uma única máquina. Felizmente, como o próprio conjunto de simulações é uma aplicação BOT, as simulações foram executadas em um grid composto por até 107 máquinas distribuídas entre 5 domínios administrativos diferentes (LSD/UFCG, Instituto Eldorado, LCAD/UFES, UniSantos e GridLab/UCSD). Para executar as simulações no grid foi utilizado o *middleware* MyGrid [8, 62], um ambiente que permite a execução de aplicações BOT em todos os processadores que o usuário possa ter acesso (ver Seção 2.1). Vale ressaltar que as máquinas que formavam o grid não estavam dedicadas às execuções das simulações, outros usuários compartilhavam estes recursos.

4.4 Resultados das Simulações

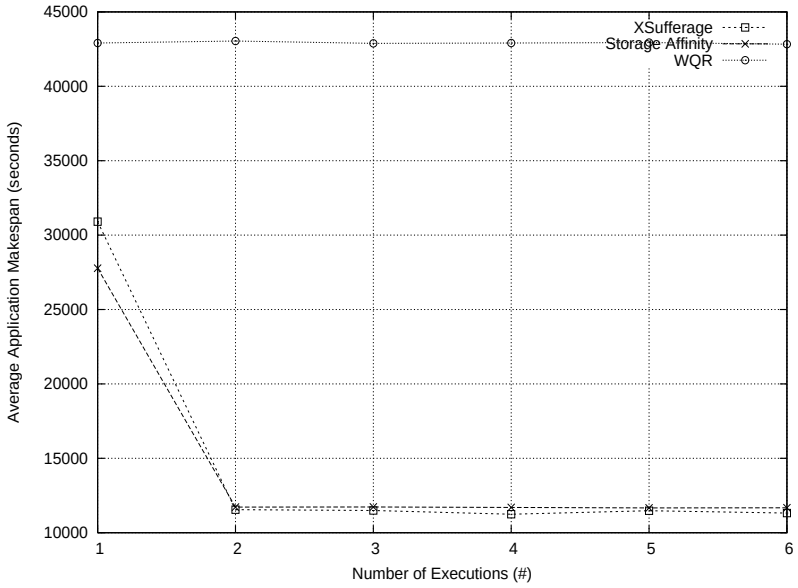
Nesta seção são analisados os resultados obtidos nas simulações das heurísticas de escalonamento. Foram analisadas a influência da granularidade da aplicação e a heterogeneidade do grid e da aplicação no desempenho do escalonamento da aplicação.

Makespan (segundos)	*Storage Affinity*	WQR	*XSufferage*
Média	14377	42919	14665
Desvio Padrão	10653	24542	11451
Desperdício de CPU	*Storage Affinity*	WQR	*XSufferage*
Média	59.243%	1.0175%	–
Desvio Padrão	52.715%	4.1195%	–
Desperdício de Rede	*Storage Affinity*	WQR	*XSufferage*
Média	3.1937%	130.88%	–
Desvio Padrão	8.5670%	135.82%	–

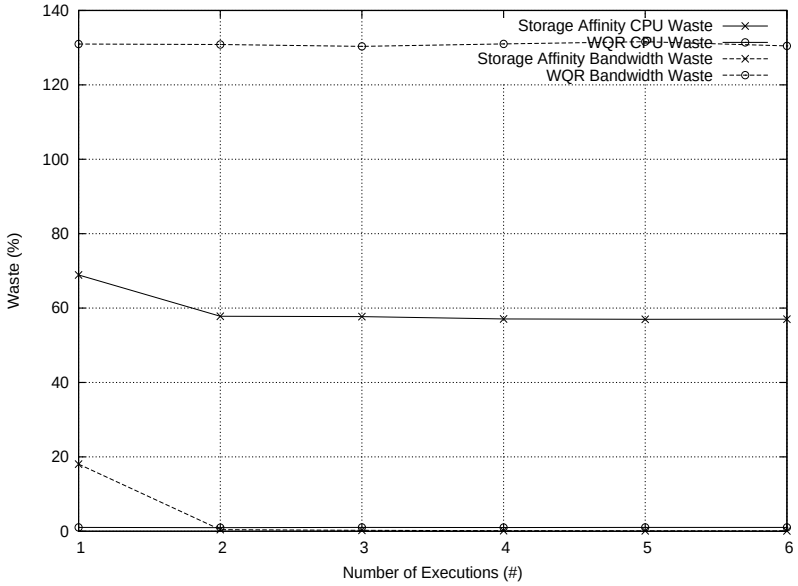
Tabela 4.2: Sumário do desempenho das heurísticas e desperdício de recursos.

Resultados gerais

Na Tabela 4.2 temos um sumário dos resultados das simulações. Os valores exibidos nesta tabela agrupam os resultados obtidos em todas as simulações. É possível notar que na média *Storage Affinity* e *XSufferage* alcançam desempenhos comparáveis, com *Storage Affinity* sendo um pouco melhor.



(a) Tempo de execução da aplicação



(b) desperdício de recursos

Figura 4.2: Sumário do desempenho das heurísticas de escalonamento

Um detalhe importante é a grande diferença de desperdício de recurso entre *Storage Affinity* e WQR. O uso de estratégias de replicação diferentes produz esse efeito. O fato de WQR não evitar transferências reduz o desperdício de CPU elevando bastante o desperdício de largura de banda da rede, ao contrário de *Storage Affinity*.

No intuito de avaliar a precisão e a confiança das médias apresentadas na Tabela 4.2, determinamos o *intervalo de confiança de* 95% [63] para a média da população (μ). Sendo assim, usando a média, o desvio padrão e o tamanho da amostra (número de valores para o makespan resultantes das simulações) foram estimados os intervalos de confiança que são apresentados na Tabela 4.3.

Heurística	*Intervalo de confiança de* 95%
Storage Affinity	$14241 < \mu < 14513$
Workqueue with Replication	$42547 < \mu < 43291$
XSufferage	$14498 < \mu < 14832$

Tabela 4.3: Intervalos de confiança de 95% para *makespan* médio de cada heurística.

Uma vez que as larguras dos intervalos de confiança (w) são relativamente pequenas, quando comparada às médias (ver Tabela 4.4), temos um bom indício que a quantidade de simulações foi suficiente para obter resultados com boa precisão.

Heurística	w	% com relação ao makespan
Storage Affinity	330	2.2%
Workqueue with Replication	330	2.2%
XSufferage	850	2%

Tabela 4.4: Largura dos intervalos de confiança e proporções com relação às médias

Na Figura 4.2 temos a média do tempo total de execução da aplicação e o desperdício de CPU em cada execução da aplicação. Estes dados são os mesmos apresentados na Tabela 4.2, porém separados por execução em cada heurística. Os resultados mostram que as duas heurísticas que levam os dados em consideração alcançam um desempenho muito melhor que WQR. Este efeito é causado pelo tempo gasto com as transferência dos dados que predomina no tempo total de execução da aplicação. Assim, o fato de não considerar os padrões de reutilização de dados prejudica de forma severa o desempenho da aplicação. Como pode ser observado, estes resultados mostram uma melhora muito acentuada a partir da segunda execução para as heurísticas *Storage Affinity* e *XSufferage*, isso é explicado pelo

fato da aplicação na primeira execução efetuar todas as transferências, enquanto que a partir da segunda os dados podem ser reutilizados e as transferências evitadas.

Na heurística WQR, a execução de cada tarefa sempre implica na transferência de seus dados de entrada (que fi ca evidente pelo grande desperdício de largura de banda e pequeno desperdício de CPU mostrado na Figura 4.2(b) e na Tabela 4.2). A falta de tratamento dos dados reutilizados em WQR impede qualquer possibilidade de melhora no desempenho que a replicação poderia implicar. De forma inversa, devido à exploração dos padrões de reutilização de dados, a estratégia de replicação de *Storage Affinity* torna possível evitar o impacto da falta de informação dinâmica, apresentando uma performance muito similar à heurística *XSufferage*. O efeito colateral de *Storage Affinity* é o consumo extra de recursos (uma média de 59% de ciclos extras de CPU e uma quantidade mínima de largura de banda). Enquanto a principal inconveniência de *XSufferage* é a necessidade de conhecer as informações dinâmicas sobre os recursos. Partindo destes resultados podemos afi rmar que *Storage Affinity* é uma heurística poderosa o sufi ciente para melhorar o desempenho da aplicação, além de ser uma alternativa factível ao uso de informação dinâmica no escalonamento de aplicações PHD.

Impacto da Granularidade

A partir deste ponto, o foco da análise de desemepenho será nos algoritmos *Storage Affinity* e *XSufferage*, pois a heurística WQR não apresentou nenhum comportamento de muito destaque, alcançando desempenhos muito abaixo das duas outras heurísticas pelo fato de não explorar os padrões de reutilização de dados.

Makespan (Média)	*Storage Affi nity*	*XSufferage*	Melhora
$75MBytes$	20228.45	14449.315	-28.57%
$15MBytes$	6228.56	10009.04	37.77%
$3MBytes$	3667.91	6542.88	43.94%

Tabela 4.5: Sumário do desempenho separando por granularidades.

Em seguida investigamos o impacto da granularidade da aplicação no desempenho do escalonamento da aplicação. Na Tabela 4.5 são descritos os níveis de melhora apresentados por *Storage Affinity* em relação a *XSufferage* para cada granularidade. É possível notar que para os grãos menores *Storage Affinity* é melhor que *XSufferage*.

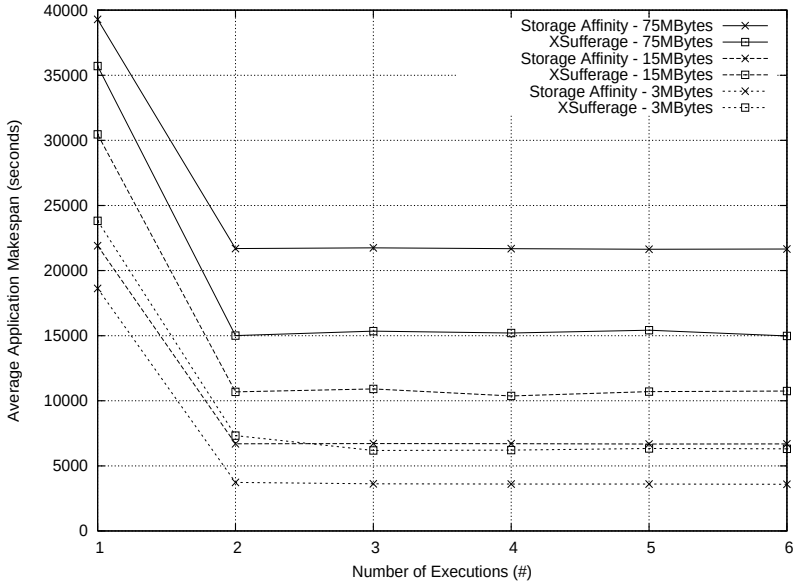
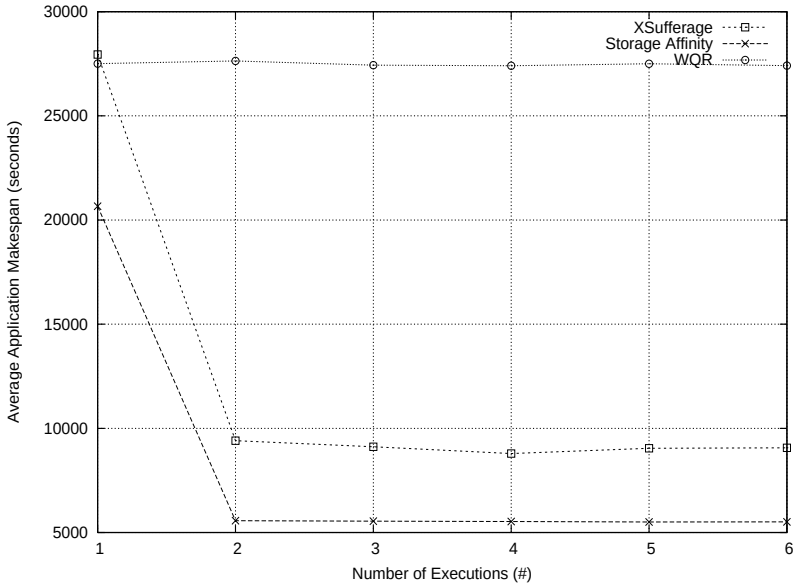
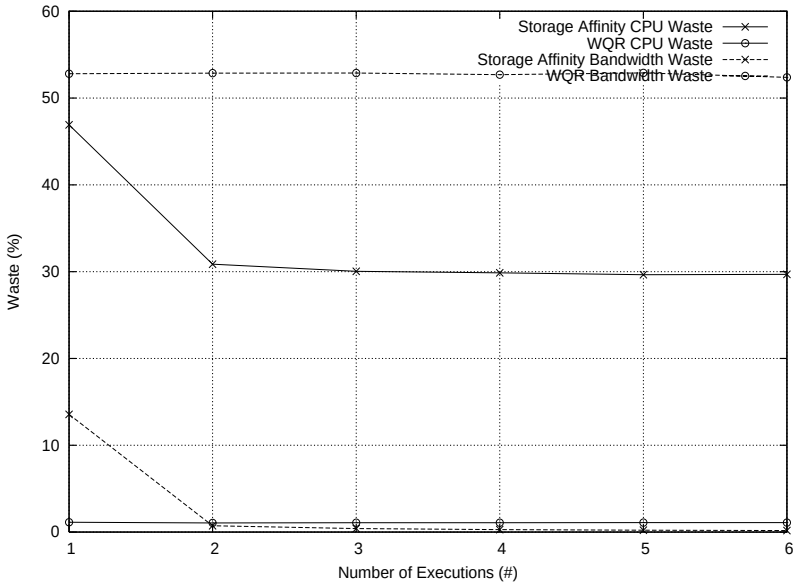


Figura 4.3: Impacto da granularidade da aplicação

Na Figura 4.3 podemos ver a influência dos três tipos de granularidades em *Storage Affinity* e *XSufferage*. Destes resultados pode-se concluir que não importa a heurística usada, menores granularidades implicarão em melhor desempenho. Isto ocorre porque menores tarefas permitem um maior paralelismo na execução da aplicação. Ainda podemos observar que *XSufferage* obtém melhor performance que *Storage Affinity* somente quando a granularidade da aplicação é de $75Mbytes$. Isto decorre do fato que quanto maior é uma tarefa em particular, maior será sua influência no desempenho da aplicação. Assim, caso essa tarefa seja associada a um processador lento, o impacto causado no desempenho geral da aplicação é muito maior do que no caso de uma tarefa de tamanho pequeno. Em outras palavras, a estratégia de replicação de *Storage Affinity* mostra melhores resultados no tratamento de escalonamentos inefi cientes quando a granularidade da aplicação é pequena. Contudo, é possível argumentar que, ao considerar aplicações PHD, é normalmente possível - e bastante fácil - reduzir a granularidade da aplicação convertendo uma tarefa que possui um arquivo de entrada grande, em várias tarefas com arquivos de entrada menores. Esta conversão é tipicamente efetuada através de um simples particionamento dos arquivos de entrada grandes em vários arquivos menores.



(a) Tempo de execução da aplicação



(b) Desperdício de recursos

Figura 4.4: Desempenho do escalonamento da aplicação para granularidades $3Mbytes$ e $15MBytes$

Dada a discussão acima, mostramos na Figura 4.4(a) os valores para o tempo total de execução das aplicações, considerando apenas as granularidades $3Mbytes$ e $15Mbytes$. Para estas simulações, como pode-se constatar na Tabela 4.6, na média, *Storage Affinity* é 32% melhor que *XSufferage*. Além disso, a porcentagem de ciclos de CPU desperdiçados é reduzida

de uma média aproximada de 59% para aproximadamente 33%, como pode ser observado na Figura 4.4(b) e na Tabela 4.6. Portanto, vale enfatizar que a redução da granularidade da aplicação é uma boa política para o aumento do desempenho, uma vez que menores tarefas implicaram em um maior grau de paralelismo (ver Figura 4.2(a)).

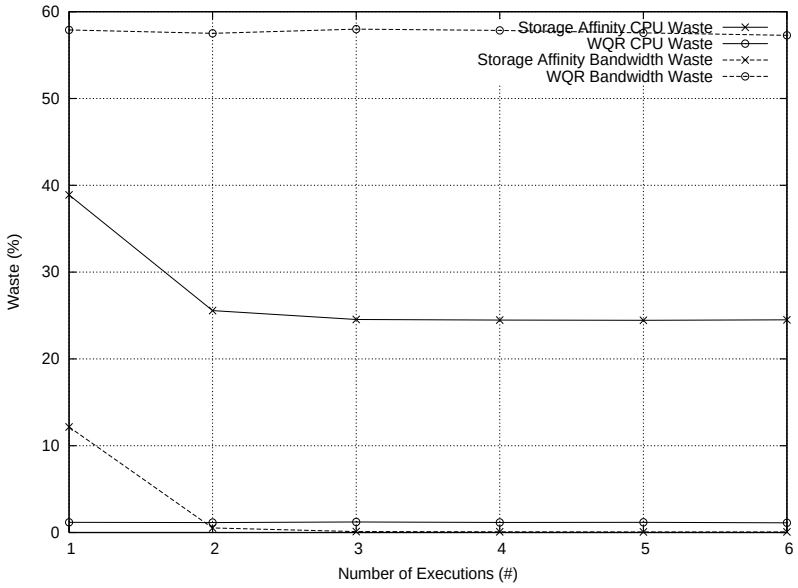
Makespan	*Storage Affinity*	WQR	*XSufferage*
Média	8054	27482	12229
Desvio Padrão	6021	6004	10584
Desperdício de CPU	*Storage Affinity*	WQR	*XSufferage*
Média	32.837%	1.0856%	–
Desvio Padrão	25.681%	2.5105%	–
Desperdício de Rede	*Storage Affinity*	WQR	*XSufferage*
Média	2.5612%	52.758%	–
Desvio Padrão	6.2439%	33.151%	–

Tabela 4.6: Sumário do desempenho para as granularidades $15MBytes$ e $3MBytes$.

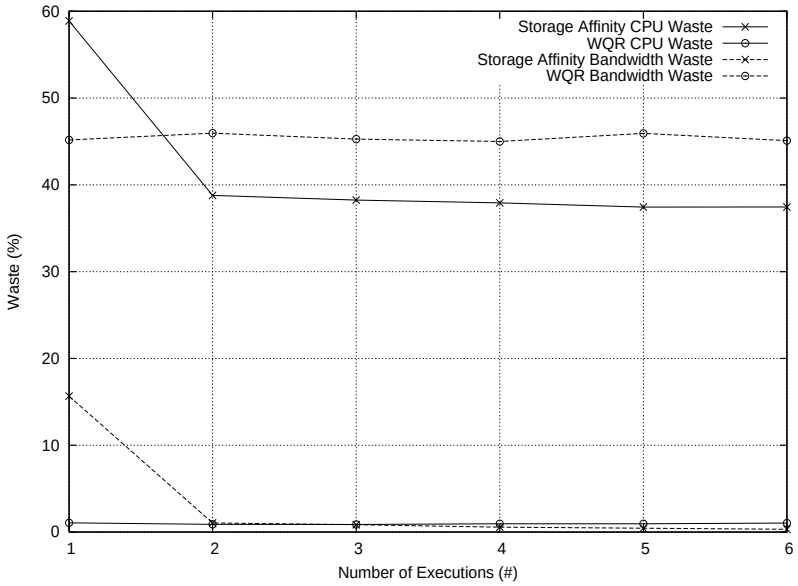
Impacto da relação *tamanho da entrada/custo computacional*

No intuito de analisar a influência das diferentes características das aplicações PHD, em particular a relação tempo de execução/tamanho dos dados de entrada das tarefas, no *makespan* e no desperdício de recursos, foram analisados separadamente os dois tipos de aplicação contempladas nas simulações (ver Seção 4.2).

A partir dos resultados é possível ver que o comportamento das heurísticas, considerando o tempo de execução da aplicação, não é afetado pelas diferentes relações entre tempo de execução e tamanho de entrada das tarefas. Em contrapartida, foi verificado que com relação ao desperdício de recursos, as heurísticas sofrem um pequeno impacto. Na Figura 4.5(b) fica claro que para *Storage Affinity* o nível de desperdício é maior na aplicação de busca de padrões. Enquanto que na Figura 4.5(a) é percebido que WQR gasta mais recursos na aplicação de visualização. Para entender este fenômeno é importante relembrar que na aplicação de visualização, o custo computacional das tarefas é completamente determinado pelo tamanho de suas respectivas entradas. Portanto, como *Storage Affinity* prioriza a tarefa com o maior valor de *afinidade*, isto significa que para aplicações como a aplicação de visualização, a tarefa de maior custo computacional é escalonada primeiro. Logo, a replicação de tarefas iniciará somente quando a maior parte da aplicação já executou.



(a) Aplicação de visualização científica

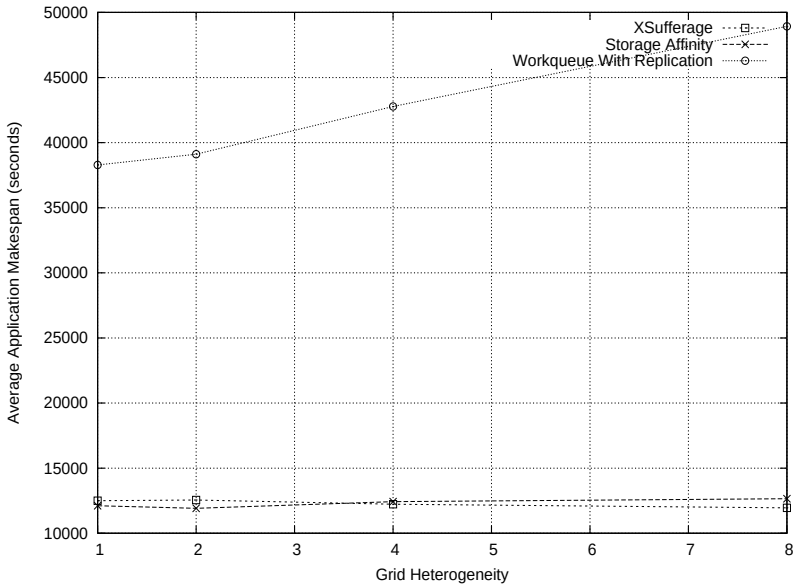


(b) Aplicação de busca de padrões

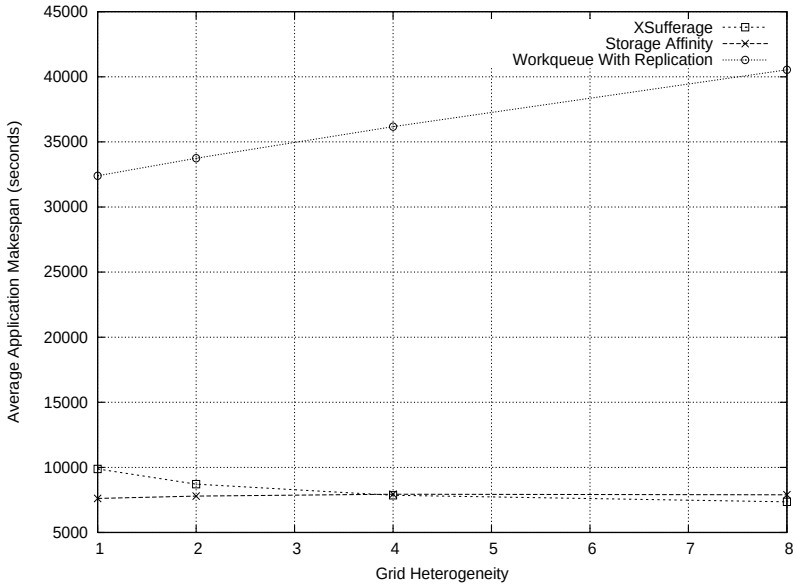
Figura 4.5: Impacto do tipo de aplicação no nível de desperdício de recursos

No caso da aplicação de busca de padrões, o custo computacional das tarefas não é completamente determinado pelo tamanho dos dados de entrada das tarefas, assim tarefas com custo computacional relativamente alto podem ser escalonadas no fi nal da execução da aplicação. Sendo assim, a replicação pode iniciar quando ainda falta executar uma grande porção da aplicação, e conseqüentemente mais recursos são desperdiçados para melhorar o tempo

de execução da aplicação.



(a) Aplicação de visualização de dados



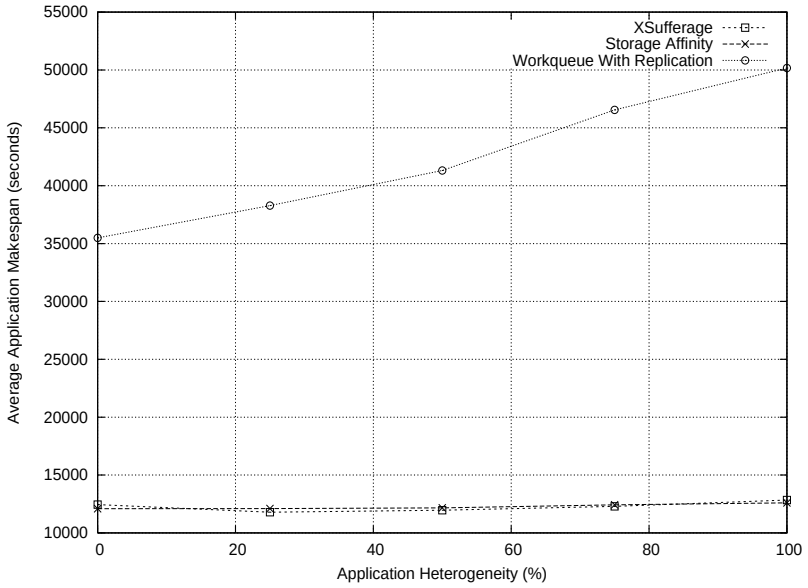
(b) Aplicação de busca de padrões

Figura 4.6: Impacto da heterogeneidade do grid

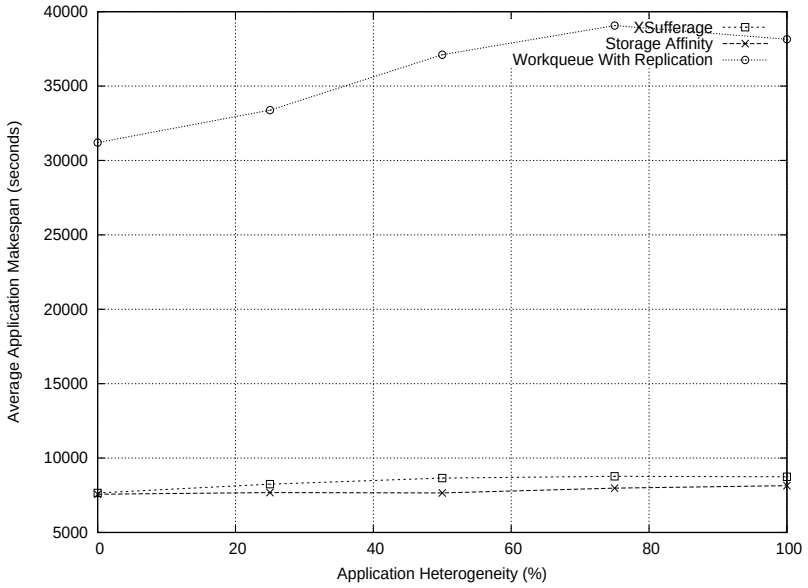
Impacto da heterogeneidade

Finalmente, foram analisadas as influências da heterogeneidade do grid e da aplicação para os dois tipos de aplicação. Na Figura 4.6(a) e na Figura 4.6(b) podemos ver como a heteroge-

neidade do grid influencia no tempo de execução na aplicação de visualização de dados e na aplicação de busca de padrões, considerando as três heurísticas discutidas. *Storage Affinity* e *XSufferage* não sofrem grande impacto por causa da variação da heterogeneidade do grid.



(a) Aplicação de visualização de dados



(b) Aplicação de busca de padrões

Figura 4.7: Impacto da heterogeneidade da aplicação

No caso de *XSufferage*, isto não é uma surpresa, pois esta heurística usa informação sobre o ambiente e pode tolerar a variação de heterogeneidade facilmente. Um comportamento

similar é apresentado por *Storage Affinity*, o que mostra que sua estratégia de replicação torna a aplicação robusta aos efeitos das variações de velocidade dos processadores no grid. Por outro lado, WQR sofre bastante impacto com a variação na heterogeneidade do grid. Podemos verificar que com o aumento da heterogeneidade do grid o tempo de execução da aplicação piora.

As heurísticas que consideram os dados também apresentam uma boa tolerância à variação da heterogeneidade da aplicação. Na Figura 4.7, observamos que o tempo de execução da aplicação apresenta uma pequena variação para ambos os tipos de aplicação.

4.5 Validação

No intuito de validar nossas simulações, conduzimos alguns experimentos usando um protótipo da heurística *Storage Affinity*. O protótipo foi desenvolvido como uma nova heurística de escalonamento para o software MyGrid [8, 62].

O grid usado nos experimentos foi formado por 18 processadores localizados em 2 sites distintos (Carcara Cluster/LNCC - Teresópolis, Brasil and GridLab/UCSD - San Diego, USA). A máquina base (p_{home}) residia no Laboratório de Sistemas Distribuídos/UFCG - Campina Grande, Brasil. É importante ressaltar que durante os experimentos os recursos estavam sendo compartilhados com outros usuários.

Com relação a aplicação, usamos o BLAST [1]. O BLAST é uma aplicação que busca uma dada seqüencia de caracteres em um banco de dados. Estes caracteres representam uma seqüencia de proteínas e o banco de dados contém várias seqüencias de proteínas identificadas. A aplicação recebe dois parâmetros: um banco de dados e uma seqüencia de caracteres que deve procurada no banco de dados. O tamanho do banco de dados é da ordem de muitos GBytes, mas pode ser dividido em muitas partições de alguns MBytes. Por outro lado, o tamanho da seqüencia de caracteres que deve ser procurada no banco não ultrapassa 4KBytes.

A aplicação foi formada por 20 tasks. Cada tarefa da aplicação recebe uma partição de 3MBytes de um grande bando de dados copiado do repositório BLAST [2] e uma seqüencia de caracteres menor que 4KBytes. Uma vez que as simulações focaram na aplicações que apresentam padrão de reutilização de dados *inter-job* (ver Seção 3.2), a aplicação foi configurada para apresentar o mesmo padrão. A maior parte da entrada de cada tarefa foi

reutilizada (a partição do banco de dados), enquanto a menor parte da entrada (o alvo da busca de alguns KBytes) era modifi cado entre as execuções.

Metodologia

Nos experimentos foram analisadas duas heurísticas: *Storage Affinity* e *Workqueue with Replication*. *XSufferage* não foi usada devido a falta de uma infraestrutura de monitoração que pudesse fornecer as informações sobre a carga dos recursos, necessárias para o funcionamento da heurística *XSufferage*. Por outro lado, MyGrid já possuía uma versão estável da heurística *Workqueue with Replication* disponível.

Para minimizar o efeito da dinamismo do grid nos resultados, os experimentos consistiram de execuções *back-to-back* das duas heurísticas de escalonamento. Ou seja, os experimentos foram conduzidos de forma que as execuções dos experimentos de cada heurística eram intercaladas. Seguindo esta abordagem, executamos 11 experimentos. Cada experimento consistiu de 4 execuções sucessivas da mesma aplicação para cada heurística, totalizando assim 88 execuções da aplicação.

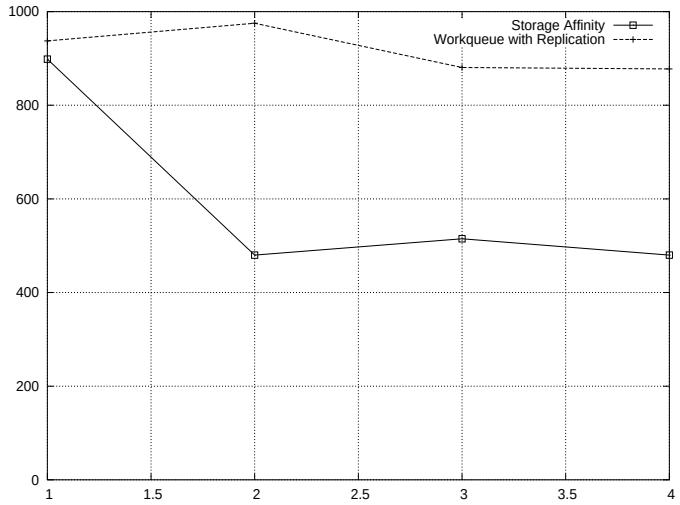


Figura 4.8: Resultados dos experimentos - média do *makespan* da aplicação para cada heurística

Resultados

Na Figura 4.8 apresentamos a média do tempo de execução da aplicação para cada heurística de escalonamento. Enquanto que na Figura 4.9 são apresentados os tempos da simulação do

cenário usado nos experimentos. Os resultados mostram que tanto *Storage Affinity* quanto *Workqueue with Replication* apresentam o mesmo comportamento geral, apontado nos resultados das simulações. Porém, os experimentos diferem da simulação em alguns aspectos. Um desses aspectos é uma maior flutuação nos valores do *makespan* nos experimentos. isto é devido ao alto grau de heterogeneidade do grid.

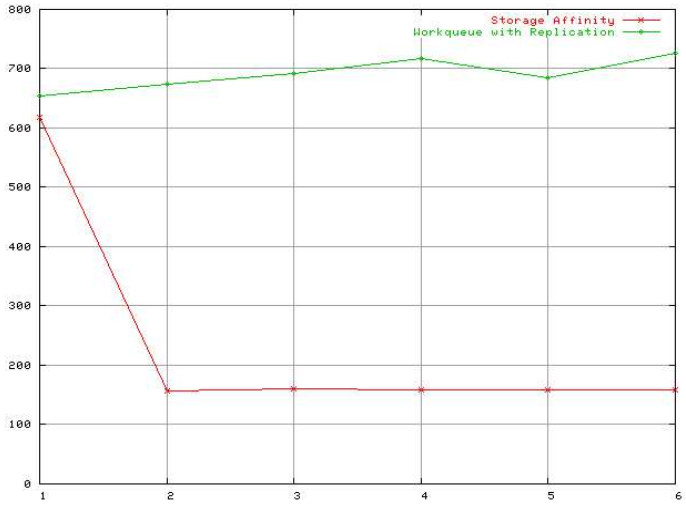


Figura 4.9: Simulação do cenário considerado nos experimentos

Podemos observar que há uma diferença entre o *makespan* obtido nos experimentos e nas simulações. Acreditamos que existem duas razões para esta discrepância. Primeiro, as cargas de CPU e de rede não foram coletadas durante os experimentos. Assim, foram utilizados outros logs NWS. Portanto, o cenário do grid não é foi o mesmo para as simulações e para os experimentos. Segundo, o protótipo *Storage Affinity* sempre faz consultas aos sites para obter informações sobre a existência e sobre os tamanhos dos arquivos. Estas consultas são operações que agregam um custo no escalonamento que não foi considerado no modelo das simulações. porém, uma vez que o escalonador tem a responsabilidade de transferência dos arquivos para os sites, essas informações (localização dos arquivos e tamanho) pode ser armazenada localmente. Desta forma, haveria uma redução da necessidade de invocações remotas durante a execução do *Storage Affinity*.

Capítulo 5

Conclusões e perspectivas

Neste trabalho, foi apresentada *Storage Affinity* uma nova heurística para escalonamento eficiente de aplicações PHD (*Processors of Huge Data*) em grids computacionais. As aplicações PHD são definidas como uma subclasse das aplicações BOT (*Bag-of-Tasks*). As principais característica das aplicações PHD são a necessidade de processar grandes quantidades de dados e o fato de geralmente reutilizarem dados de seu conjunto de entradas. A eficiência de *Storage Affinity* foi comparada com *XSufferage* [14] e WQR [17].

Semelhante a *Storage Affinity*, a heurística *XSufferage* explora os padrões de reutilização de dados no intuito de reduzir o impacto de grandes transferências de dados no desempenho geral da aplicaçao. Porém, *XSufferage* usa informações sobre o ambiente e sobre a aplicação, as quais, em geral, não são fáceis de se obter. WQR tenta contornar a questão do uso de informações sobre o ambiente e sobre a aplicação, contudo não considera o impacto das transferências de dados no desempenho da aplicação. Sendo assim, *Storage Affinity* efetua escalonamentos sem a utilização de informação de difícil acesso (sobre a aplicação apenas), bem como explora os padrões de reutilização de dados para evitar uma degradação no desempenho da aplicação devido à transferência de grandes quantidades de dados através de conexões de longa distância. Vale relembrar que, ao contrário das informações que *XSufferage* depende, *Storage Affinity* utiliza informação sobre a localização dos dados de entrada das tarefas. Essa informação pode ser trivialmente obtida, mesmo considerando a ampla dispersão de recursos nos grids computacionais.

Através de simulações foi constatado que *Storage Affinity* alcança um desempenho melhor que as outras heurísticas. Esse é um efeito da utilização de uma estratégia de replicação

de tarefas e da exploração dos padrões de reutilização de dados, que permitem evitar transferências de dados desnecessárias. Estes resultados mostram que prover políticas para reduzir o impacto das transferências de dados no desempenho da aplicação é algo imprescindível para obter escalonamentos eficientes em aplicações PHD executando em grids computacionais. Além disso, o nível de influência das heterogeneidades do grid e da aplicação no desempenho dos escalonamentos é pequeno. Por outro lado, a granularidade da aplicação causa bastante impacto.

Considerando o aspecto da granularidade foi percebido que independente da heurística utilizada, nas granularidades menores as heurísticas alcançam melhores desempenhos. Porém, um ponto importante é que a granularidade de aplicações PHD pode, em geral, ser reduzida. Assim, a redução da granularidade da aplicação é uma operação que deve ser considerada quando o intuito é ganhar desempenho. Nos cenários favoráveis à *Storage Affinity*, esta heurística efetua escalonamento que resulta em um desempenho, em média, 42% melhor que *XSufferage*. Uma desvantagem de *Storage Affinity* é o desperdício de recursos devido a sua estratégia de replicação. Nossos resultados mostram que a largura de banda desperdiçada é desprezível e o desperdício de CPU pode ser reduzido para menos de 31% com a redução da granularidade da aplicação.

Como trabalhos futuros, pretende-se investigar os seguintes tópicos: i) o impacto do padrão de reutilização *inter-task* no escalonamento da aplicação; ii) gerenciamento de espaço em disco nos servidores de dados; iii) o comportamento emergente de uma comunidade de escalonadores competindo por recursos compartilhados; e iv) a utilização de técnicas introspecção para *data staging* [64], no intuito de fornecer ao escalonador informações sobre a localização dos dados e a utilização de espaço em disco. Além disso, espera-se que a experiência prática do uso de um protótipo de *Storage Affinity*, implementado no MyGrid, nos ajude na identificação de aspectos do nosso modelo que necessitam ser refinados.

Apêndice A

Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids

Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, Aliandro Lima
Universidade Federal de Campina Grande

Abstract: Data-intensive applications executing over a computational grid demand large data transfers. These are costly operations. Therefore, taking them into account is mandatory to achieve efficient scheduling of data-intensive applications on grids. Further, within an heterogeneous environment such as a grid, better schedules are typically attained by heuristics that use dynamic information about the grid and the applications (network and CPU loads, completion time of tasks, etc). However, these information are often difficult to be accurately obtained. Although there are schedulers that attain good performance without requiring dynamic information, they were not designed to take data transfer delays into account. This paper presents *Storage Affinity*, a novel scheduling heuristic for *bag-of-tasks* data-intensive applications running on grid environments. Storage Affinity exploits a data reuse pattern, common on many data-intensive applications, that allows it to take data transfer delays into account and reduce the makespan of the application. Further, it uses a replication strategy that yields efficient schedules without relying upon dynamic information that is difficult to obtain. Our results show that Storage Affinity may attain performance that is in average 42% better than that of state-of-the-art knowledge-dependent schedulers, even in the unlikely case when the latter are fed

with perfect information. This is achieved at the expense of consuming more CPU cycles (in average, 31% more than not using replication).

A.1 Introduction

Several scientific applications generate or process huge amount of data. Genomic sequencing [1], high-energy physics [3] and visualization of quantum optics simulations results [4] are good examples of the so called *data-intensive applications*. In order to process large datasets, these applications, in general, need a high performance computing infrastructure. Fortunately, since the data splitting procedure is easy and each data element can typically be processed independently, a solution based on data parallelism can often be employed.

Task independence is the main characteristic of parallel *Bag-of-Tasks* (BOT) applications [8] [9]. A BOT application is composed of tasks that do not need to communicate to proceed with their computation. In this work, we are interested in the class of applications which has both BOT and *data-intensive* characteristics. We have named it *processors of huge data* (PHD). Shortly, PHD = *BoT* + *data-intensive*.

Due to the independence of their tasks, BoT applications are normally suitable to be executed on grids [11] [8]. However, since resources in the grid are connected by wide area network links (WAN), the bandwidth limitation is an issue that must be considered when running PHD applications on such environments. This is particularly relevant for those PHD applications that present a data reutilization pattern. For these applications, this characteristic can be exploited to achieve better performance. Data reutilization can be either among tasks of a particular application or among a succession of applications executions. For instance, in the visualization process of quantum optics simulations results [4] it is common to perform a sequence of executions of the same parallel visualization application, simply sweeping some arguments (e.g. zoom, view angle) and preserving a huge portion of the data input from the previous executions.

There exists some algorithms that are able to take data transfer into account when scheduling PHD applications on grid environments [13] [14] [15] [16]. However, they require knowledge that is not trivial to be accurately obtained in practice, especially on a widely dispersed environment such as a computational grid [11]. For example, XSufferage [14]

considers the CPU and network loads, as well as the execution time of tasks, all of which must be known a priori, in order to decide how much a given task would *suffer* if not scheduled on a given site.

On the other hand, for CPU-intensive BoT applications, there are schedulers that do not use dynamic information, yet achieve good performance (e.g. Workqueue with Replication - WQR [17]). They use replication to tolerate bad scheduling decisions taken due to the lack of accurate information about both the environment and the application. However, these schedulers were conceived to target CPU-intensive applications and thus data transfer are not taken into account by them.

In this paper we introduce *Storage Affinity*, a new heuristic for scheduling PHD applications on grids. Storage Affinity takes into account the fact that input data is frequently reused in successive executions of many PHD applications. It tracks the location of data to allow schedules that avoid, as much as possible, large data transfers. Further, it reduces the effect of inefficient task-processor assignments via the judicious use of task replication.

The rest of the paper is organized in the following way. In the next section we present the system model that is considered in this work. In Section A.3, we present the Storage Affinity heuristic as well as other heuristics used for comparative purposes. In Section A.4, we outline the workload used on the simulations and present an analysis of the performance of the schedulers discussed. Section A.5 concludes the paper with our final remarks and a brief discussion on future perspectives.

A.2 System Model

This section formally describes the problem investigated and also provides the terminology used in the rest of the paper.

A.2.1 System Environment

We consider the scheduling of a sequence of jobs[1] over a grid infrastructure. The grid G is formed by a collection of sites. Each site is comprised of a number of processors, which are

[1] We use the terms job and application interchangeably.

able to run tasks, and a single data server which is able to store input data required in the execution of a task, and output data generated by the execution of a task. More formally:

$$G = \{site_1, \ldots, site_g\}, g > 0, \text{ and } site_i = P_i \cup \{S_i\},$$

where P_i is the non-empty set of processors at $site_i$ and S_i is the data server at $site_i$. We assume that the resources owned by the various sites are disjoint, *i.e.* $\forall i, j, i \neq j, site_i \cap site_j = \varnothing$.

Processors belonging to the same site are connected to each other through a high bandwidth local area network, whose latency is small and throughput is large when compared to those experienced by the wide are networks that interconnect processors belonging to different sites. Because this assumption we only consider one data server per site, *i.e.* the collection of data servers that may be present in a site is collapsed into a single data server.

We define two sets to encompass all processors (P_G) and all data servers (S_G) present in a grid G. That is to say:

$$P_G = \bigcup_{1 \leq i \leq |G|} P_i, \quad \text{and} \quad S_G = \bigcup_{1 \leq i \leq |G|} \{S_i\}.$$

We assume that the user spawns the execution of applications from a *home* machine that does not belong to the grid ($p_{home} \notin G$). Further, we assume that before the first execution of an application, all its input data are stored at the local file system of the home machine (S_{home}). Figure A.1 illustrates the assumed environment.

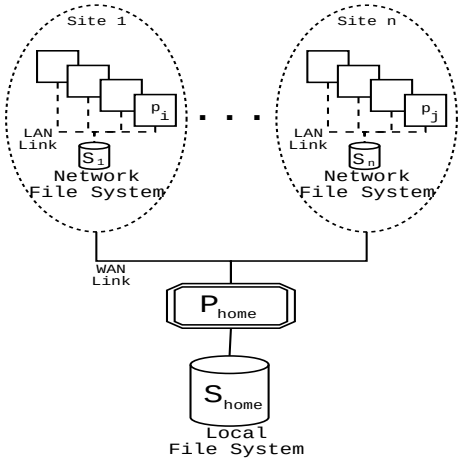


Figura A.1: The sytem environment model

A.2.2 Application

Job J_j, $j > 0$, is the j^{th} execution of the aplication J. A job is composed by a non-empty collection of tasks, in which each task is defined by two datasets: the input and the output datasets. Formally:

$$J_j = \{t_1^j, \ldots, t_n^j\}, n > 0, \text{and } t_t^j = (I, O), I \cup O \neq \varnothing,$$

where $t_t^j.I$ and $t_t^j.O$ are the input and the output datasets of task t_t^j, respectively.

For each data server $S_i, S_i \in S_G$, let $ds_j(S_i)$ be the set of data elements that are stored at S_i before the execution of the j^{th} job was started, and let $ds(S_{home})$ be the set of data elements that are stored at the home machine. We define D_j as the set of all data elements that are available to be taken as input by the j^{th} job to be executed. D_j is given by:

$$D_j = ds(S_{home}) \cup \left\{ \bigcup_{S_i \in S_G} ds_j(S_i) \right\}.$$

We have that $t_t^j.I \subseteq D_j$ and after the execution of the j^{th} job, the set of available data elements D_{j+1} is given by the union of D_j with all data elements that have been output by J_j:

$$D_{j+1} = D_j \cup \left\{ \bigcup_{1 \leq t \leq |J_j|} t_t^j.O \right\}.$$

We also define the input dataset of the entire application as the union of the input dataset of each task in the job. It is expressed by:

$$J_j.I = \bigcup_{k=1}^{|J_j|} t_k^j.I.$$

A.2.3 Job Scheduling and Performance Metrics

A schedule Σ_j of the job J_j comprises the schedule of each one of the tasks that form J_j. The schedule of a particular task t_t^j of J_j specifies the processor that is assigned to execute t_t^j. Note that it is possible for the same processor to be assigned to more than one task. Formally,

$$\Sigma_j = \{p_1^j, \ldots, p_n^j\}, n = |J_j|, p_t^j \in P_G, 0 \leqslant t \leqslant n.$$

We assume that a task can only access the data server at the same site of the processor on which the task is running. Thus, after t_t^j is executed at $p_t^j \in site_i$, S_i will have stored all

data elements in the dataset $t_t^j.O$. Consequently, if all data elements in the dataset $t_t^j.I$ are not already stored at S_i, the absent data elements must be first transferred to S_i before the execution of t_t^j can be started at p_t^j.

We measure the application execution time to evaluate the efficiency of a scheduling. Thus, the heuristic we propose in this paper and also the others that we discuss, all have a common goal, which is to minimize this metric. The application execution time, normally referred as its *makespan* [55], is the time elapsed between the moment the first task is started until the moment all tasks have finished their execution.

A.3 Scheduling Heuristics

Despite the fact that BoT applications are suitable to run on computational grids, the efficient scheduling of these applications on grid environments is not trivial. The difficulty related to the scheduling of BoT application is twofold. The first problem is particular to PHD applications, which must deal with a huge amount of data. The issue here is that the large data transfers that occurs during the tasks execution greatly affects the application overall performance. The second problem consists of obtaining precise information about the performance resources will deliver to the application. Such information is input for most schedulers. However, the CPU load, network load and the execution time of tasks are typically not available a priori. In fact, there has been a great deal of research on predicting future CPU and network performance as well as application execution time [49] [50] [51] [52] [53]. As the results of these efforts show, this is by no means an easy task. To complicate matters further, the lack of central grid control make it difficult deploying resource monitoring middleware.

Considering the problem of obtaining accurate dynamic information, we know that it is possible to circumvent it by introducing task replication into the scheduling [17]. The idea behind the task replication is to improve the application performance by increasing the chances of running the tasks on fast/unloaded processors. Previous experiments have shown that the application performance can be vastly improved by this technique [17] [8]. However, these experiments consider only CPU-intensive applications. As we shall see in Section A.4, using replication with no concern for data transfers results in poor performance for PHD

applications.

With respect to the huge amount of data manipulated by PHD applications, it is obvious that one can do a lot better by avoiding unnecessary data transfers. In particular, unnecessary data transfers can be identified and exploited when the application reuses data. Data reutilization is a common characteristic among PHD applications [54] [4]. There are known heuristics that address the large data transfers impact on application performance [14] [13]. Nevertheless, the heuristics depends on dynamic information about the environment (i.e. CPU load, network bandwidth utilization and execution time of tasks).

We can observe that the difficulty in obtaining dynamic information and the impact of large data transfers have been individually attacked. In the following, we comment two scheduling heuristics that deal with these problems separately, Workqueue with Replication (WQR) [17] and XSufferage [14]. We also introduce our approach to address the two PHD scheduling problems together.

A.3.1 Workqueue with Replication

The WQR scheduling heuristic [17] has been conceived to solve the problem of obtaining precise information about the future performance tasks will experience on grid resources. Initially, WQR is similar to the traditional Workqueue scheduling heuristic. Tasks are sent at random to idle processors and when a processor finishes a task, it receives a new task to execute. WQR differs from Workqueue when a processor becomes available and there is no waiting task to start. At this point, Workqueue would just wait for all tasks to finish. WQR, however, starts replicating the tasks yet running. The result from a task comes from the first replica to finish. After the first replica completes, all other replicas are killed.

The rationale behind WQR is to decrease the application makespan by generating more efficient task-to-host assignments to replicas than those performed to the original tasks. WQR achieves good performance for CPU-intensive application [17] without using any kind of dynamic information about processors, network links or tasks. The drawback is that some CPU cycles are wasted with the replicas that do not complete. Moreover, WQR does not take data transfer into account, what results in poor performance for PHD applications, as we shall see in Section A.4.

A.3.2 XSufferage

XSufferage [14] is a knowledge-based scheduling heuristic that deals with the impact of large data transfers on PHD applications running on grid environments. XSufferage is an extension of the *Sufferage* scheduling heuristic [56]. *Sufferage* prioritizes the task that would "suffer" the most if not assigned to the processor that runs it the fastest. How much a task would suffer is gauged by its *sufferage value*, which is defined as the difference between the best and the second best completion time for the task.

The main difference between XSufferage and Sufferage algorithms is the sufferage value determination method. In XSufferage, the sufferage value is calculated using the *site-level* task completion times. The *site-level* completion time of a given task t_t^j is the minimum completion time achieved among all processors within the site. The *site-level sufferage* is the difference between the best and second best *site-level* completion times of the task. The other difference is that XSufferage considers input data transfers in the calculation of the completion time of the task, thus, differently from Sufferage, it requires information about network bandwidth as input.

The algorithm input is a job J_j and a grid G. The algorithm traverses the set J_j until it finds the task t_t^j with the highest sufferage value. This task is assigned to the processor that has presented the earliest completion time. This action is repeated until all tasks in J_j are scheduled.

The rationale behind XSufferage is to consider the data location when performing the task-to-host assignments. The expected effect is the minimization of the impact of unnecessary data transfers on the application makespan. The evaluation of XSufferage shows that avoiding unnecessary data transfers indeed improves the application's performance [14]. However, XSufferage calculates sufferage values based on the knowledge about CPU loads, network bandwidth utilization and task execution times. In general, these information are not easy to obtain. Depending on the administrative restrictions, they may even be unavailable.

A.3.3 Storage Affinity

Storage Affinity was conceived to exploit data reutilization to improve the performance of the application. Data reutilization appears in two basic flavors: *inter-job* and *inter-task*. The

former arises when a job uses the data already used by (or produced by) a job that executed previously, while the latter appears in applications whose tasks share the same input data. More formally, the *inter-job* data reutilization pattern occurs if the following relation holds:

$$(j \neq k) \wedge (J_j.I \cap J_k.I \neq \varnothing).$$

On the other hand, the *inter-task* data reutilization pattern occurs if the relation presented below holds:

$$\bigcap_{t=1}^{|J_j|} t_t^j.I \neq \varnothing.$$

In order to take advantage of the data reutilization pattern and improve the performance of PHD applications, we introduce the *storage affinity* metric. This metric determines *how close* to a site a given task is. By *how close* we mean *how many bytes* of the task input dataset are already stored at a specific site. Thus, *storage affinity* of a task to a site *is the number of bytes within the task input dataset that are already stored in the site*. Formally, the *storage affinity* value between t_t^j and $site_i$ is given by:

$$SA(t_t^j, site_i) = \sum_{d \in (t_t^j.I \cap ds_j(S_i))} |d|$$

in which, $|d|$ represents the number of bytes of the data element d.

We claim that information about data size and data location can be obtained a priori without difficulty and loss of accuracy, unlike, for example, CPU and network loads or the completion time of tasks. For instance, this information can be obtained if a data server is able to answer the requests about *which* data elements it stores and *how large* is each data element. Alternatively, an implementation of a Storage Affinity scheduler can easily store a history of previous data transfer operations containing the required information.

Besides exploiting data reutilization, *Storage Affinity* also tackles the difficulty of obtaining accurate dynamic information about the grid and the application. To circumvent this problem, Storage Affinity applies *task replication*. Replicas have a chance to be submitted to faster processors than those processors assigned to the original task, thus increasing the chance of the task completion time be decreased.

```
Entrada : G, J_j
Saída   : Σ_j
while (J_j ≠ ∅) do
    affinity_max ← affinity_current ← 0; p_chosen ← t_chosen ← nil; k ← m ← 1
    while (t ⩽ |J_j|) do
        while (i ⩽ |G|) do
            if (∃ p ∈ P_i | p is not busy) then
                if (affinity_max ⩽ SA(t_t^j, site_i)) then
                    affinity_max ← SA(t_t^j, site_i)
                    p_chosen ← getFirstAvaiableProcessorAt(site_i)
                    t_chosen ← t_t^j
                end
            end
            i ← i + 1
        end
        t ← t + 1
    end
    assign(t_chosen, p_chosen)
    J_j ← J_j − {t_chosen}
    if (∀ p ∈ P_G | p is busy) then
        waitForATaskCompletionEvent()
    end
end
J_r ← getAllRunningTasks()
while (J_r ≠ ∅) do
    affinity_max ← 0; p_chosen ← t_chosen ← nil; t ← i ← 1;
    J_r ← J_r − {t_t^r ∈ J_r | ∀ site_i ∈ G, SA(t_t^r, site_i) > 0}
    J_r ← J_r − {t_t^r ∈ J_r | getReplicationDegree(t_t^r) = replicationDegree_min }
    while (t ⩽ |J_r|) do
        while (i ⩽ |G|) do
            if (∃ p ∈ P_i | p is not busy) then
                if (affinity_max < SA(t_t^r, site_i)) then
                    affinity_max ← SA(t_t^r, site_i)
                    t_chosen ← t_t^r
                    p_chosen ← getFirstAvaiableProcessorAt(site_i)
                end
            end
            i ← i + 1
        end
        t ← t + 1
    end
    if (t_chosen ≠ nil) then
        replica^d_{t_chosen} ← createReplica(t_chosen)
        assign(replica^d_{t_chosen}, p_chosen)
    else
        waitForATaskCompletionEvent()
    end
    if (∀ p ∈ P_G | p is busy) then
        waitForATaskCompletionEvent()
        killAllReplicasOfTheCompletedTask()
    end
    J_r ← getAllTasksRunning()
end
```

Algoritmo 4: Storage Affinity scheduling heuristic

Algorithm 4 presents *Storage Affinity*. Note that this heuristic is divided in two phases. In the first phase Storage Affinity assigns each task $t_t^j \in J_j$ to a processor $p \in G$. During this phase, the algorithm calculates the highest storage affinity value for each task. After this calculation, the task with the largest storage affinity value is chosen and scheduled. This

continues until all tasks have been scheduled. The second phase consists of task replication. It starts when there are no more waiting tasks and there is, at least, one available processor. A replica could be created to any running task. Considering that the replication degree of a particular task is the number of replicas of the task that have been created, whenever a processor is available, the following criteria are considered to choose the task to be replicated: i) the task must have a positive storage affinity; ii) the current replication degree of the task must the smallest among all running tasks; and iii) the task must have the largest storage affinity value among all remaining candidates. When a task completes its execution, the scheduler kills all other replicas of the task. The algorithm finishes when all the running tasks complete. Until this occurs the algorithm proceeds with replications.

A.4 Performance Evaluation

In this section we analyze the performance of Storage Affinity and compare it against WQR and XSufferage. We have decided to compare our approach to these heuristics because WQR represents the state-of-the-art solution to circumvent the dynamic information dependence, whereas XSufferage is the state-of-the-art for dealing with the impact of large data transfers. We have used simulations to evaluate the performance of the scheduling algorithms performance.

Since the performance attained by a scheduler is strongly influenced by the workload [57] [58] [59], we have designed experiments that cover a wide variety of scenarios. The scenarios vary in the heterogeneity of both the grid and the application, as well as the relation between the average number of tasks and the number of processors in the grid (a factor that strongly impacts the performance of application schedulers [17]). Our hope was not only to identify which scheduler performs better, but also to understand how different factors impact their performance.

A.4.1 Grid Environment

Each task has a computational cost, which expresses how long the task would take to execute in a dedicated reference processor. Processors may run at different speeds. By definition, the reference processor has $speed = 1$. So, a processor with $speed = 2$ (when dedicated) runs

a 100-second task in 50 seconds. Since the computational grid may comprise processors acquired at different points in time, grids tend to be very heterogeneous (i.e. their processors speed may vary widely). In order to investigate the impact of grid heterogeneity on scheduling, we consider four levels of grid heterogeneity, as shown in Table A.1[2]. Thus, for heterogeneity $1x$, we always have $speed = 10$, and the grid is homogeneous. On the other hand, for heterogeneity $8x$, we have maximal heterogeneity, with the fastest machines being up to 8 times faster than the slowest ones. Note that, in all cases, the average speed of the machines forming the grid is 10.

Grid Heterogeneity	Processor Speed Distributions
$1x$	$U(10, 10)$
$2x$	$U(6.7, 13.5)$
$4x$	$U(4, 16)$
$8x$	$U(2.2, 17.8)$

Tabela A.1: Grid heterogeneity levels and the distributions of the relative speed of processors

The grid power is the sum of the speed of all processors that comprise the grid. For all experiments we fi xed the grid power to $1,000$. Since the speed of processors are obtained from the Processor Speed Distributions, a grid is "constructed" by adding one processor at a time until the grid power reaches $1,000$. Note that the average number of processors in the grid is 100. Processors are distributed over the sites that form the grid in equal proportions. Similarly to Casanova et al [14], we assume that a grid has $U(2, 12)$ sites.

For simplicity, we assume that the data servers do not run out of disk space (i.e., we do not address data replacement policies in the present work). As previously indicated, we neglect data transfers within a site. Inter-site communication is modeled as a single shared $1Mbps$ link that connects the *home* machine to several sites.

Since grids are shared by other users, we used NWS [60] real traces to simulate contention for both CPU cycles and network bandwidth. For example, a processor of $speed = 1$ and $availability = 50\%$ runs a 100-second task in 200 seconds.

[2]Note the $U(x, y)$ denotes the uniform distribution in the $[x, y]$ range.

A.4.2 pHD Applications

In pHD applications, the application execution time is typically related to the size of the input data. The explanation for this fact is quite simple. The more data there is to process, the longer the tasks take to complete. In fact, there are pHD applications whose cost is completely determined by the size of the input data. This is the case, for example, of a scientific data visualization application, which processes the whole input data to produce the output image [4]. There are other applications that have the cost influenced, but not completely determined, by the size of the input data. This is the case of a *pattern search* application, in which the size of the input data of each task determines an upper bound for the cost of the task, not the cost of the task itself. We simulated both kinds of applications.

The total size of the input data of each simulated application was fixed to $2GBytes$. For the case of the data visualization application, the total computational cost is completely determined by the size of the input data. Based on experimental data available [4], we were able to convert the amount of input data processed by each task of the application into the number of seconds required to process the data, which is its computational cost. We have used the same proportionality factor (0.001602171 seconds/KByte) to calculate the computational cost of the pattern search application, as a function of the amount of data actually process by its tasks. To determine the computational cost of each task of the application, we used an uniform distribution $U(1, UpperBound)$, in which $UpperBound$ is the computational cost to process the entire input of a particular task.

We also wanted to analyze how the relation between the average number of tasks and the number of processors in the grid would impact the performance of a schedule. Note that when both application size and grid size are fixed, this relation is inversely proportional to the average size of the input data of the tasks that comprise the application, i.e. the *application granularity*. Thus, increasing the application granularity implies in decreasing the average number of tasks that compose the job. We have considered three application groups that are defined by the following application granularity values: $3MBytes$, $15MBytes$ and $75MBytes$.

The tasks that comprise the application can vary in size. Therefore, to simulate this variation, we introduced an application heterogeneity factor. The heterogeneity factor determines how different are the sizes of the input data of the tasks that form the job, and

consequently their costs. The size of the input data are taken from the uniform distribution $U(AverageSize \times (1 - \frac{H_a}{2}), AverageSize \times (1 + \frac{H_a}{2}))$, in which $AverageSize \in \{3MBytes, 15MBytes, 75MBytes\}$ and $H_a \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$.

A.4.3 Simulation Setting and Environment

A total of $3,000$ simulations were performed, with half of them for each type of application. In all simulations, we considered a sequence of six executions of the same job.

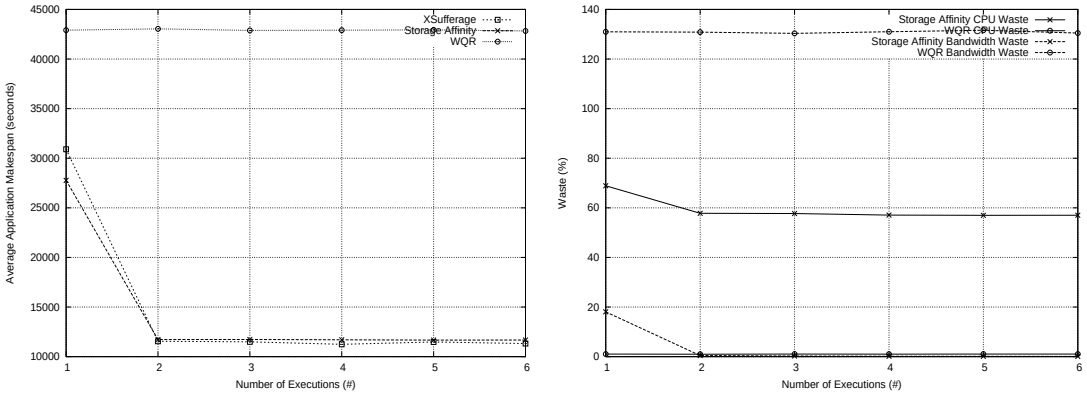
Our simulation tool has been developed using an adapted version of the Simgrid toolkit [61]. The Simgrid toolkit provides the basic functionalities for the simulation of distributed applications on grid environments. Since the set of simulations is itself a BoT application, we have executed it over a grid composed of 107 machines distributed among five different administrative domains (LSD/UFCG, Instituto Eldorado, LCAD/UFES, UniSantos and GridLab/UCSD). We have used the MyGrid middleware [62] [8] to execute it.

A.4.4 Simulation Results

In this section we show the results obtained in the simulations of the scheduling heuristics. We have analyzed the influence of the application granularity, as well as the heterogeneity of both the grid and the application on the performance of the application scheduling.

In Figure A.2 we show the average application makespan and resource waste for all performed simulations with respect to all heuristics analyzed. The results show that both data-aware heuristics attain much better performance than WQR. This is because data transfer delays dominate the makespan of the application, thus not taking them into account severely hurts the performance of the application. In the case of WQR, the execution of each task is always preceded by a costly data transfer operation (as can be inferred from the large bandwidth and small CPU waste shown in Figure A.2(b)). This impairs any improvement that the replication strategy of WQR could bring. On the other hand, the replication strategy of Storage Affinity is able to cope with the lack of dynamic information and yield a performance very similar to that of (the knowledge-centric) XSufferage. The main inconvenience of XSufferage is the need for knowledge about dynamic information, whereas the drawback of Storage Affinity is the consumption of extra resources due to its replication strategy (an

average of 59% of extra CPU cycles and a negligible amount of extra bandwidth). From this result we can state that the Storage Affinity task replication strategy is powerful enough and a quite feasible technique to obviate the need for dynamic information when scheduling PHD applications.



(a) Application Makespan

(b) Resource Waste

Figura A.2: Summary of the performance of the scheduling heuristics

Next we investigate the impact of application granularity on the application scheduling performance. In Figure A.4.4 we can see the influence of the three different granularities on both data-aware schedulers. From the results presented we conclude that no matter the heuristic used, smaller granularities yield better performance. This is because smaller tasks allow greater parallelism. We can further observe that XSufferage achieves better performance than Storage Affinity only when the granularity of the application is $75Mbytes$. This is because the larger a particular task is, the bigger its influence in the makespan of the application. Thus, the impact of a possible inefficient task-host assignment for a larger task is greater than that for a smaller one. In other words, the replication strategy of Storage Affinity is more able to circumvent the effects of inefficient task-host assignments when the application granularity is small. Nevertheless, we argue that, considering pHd applications, it is normally possible - and quite easy - to reduce the application granularity by converting a task with a large input into several tasks with smaller input datasets. The conversion is performed by simply slicing large input datasets into several smaller ones.

Given the above discussion, we show in Figure A.4(a) the values for the makespan of the applications, considering only the granularities $3Mbytes$ and $15Mbytes$. For these si-

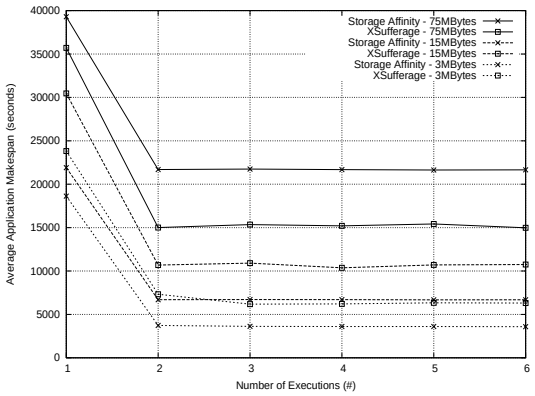
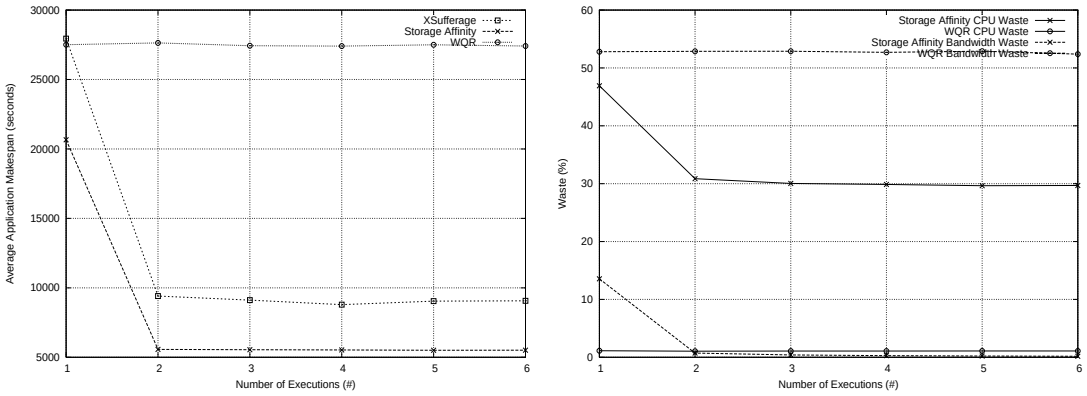


Figura A.3: Impact of the application granularity

mulations, Storage Affi nity outperforms XSufferage by 42%, in average. Further, as can be seen in Figure A.4(b) the percentage of CPU cycles wasted is reduced from 59% to 31%, in average. We emphasize that reducing the application granularity is a good policy as smaller tasks yields more parallelism (see Figure 3).

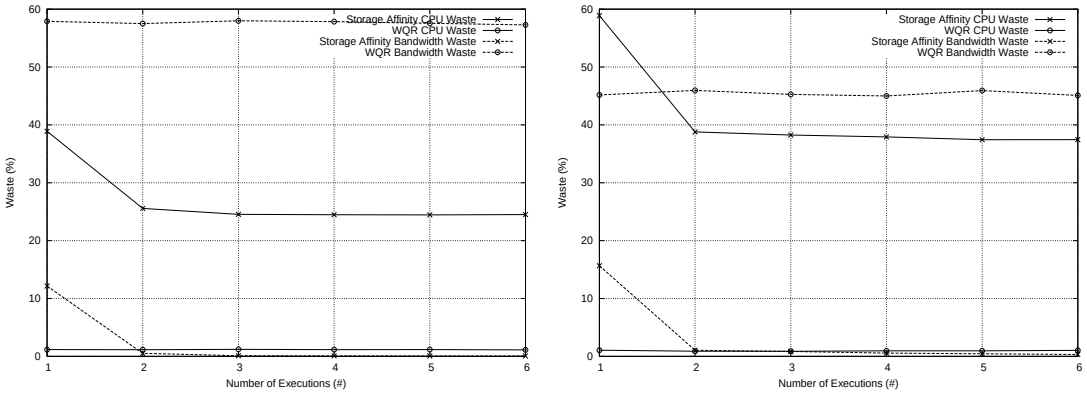


(a) Application Makespan (b) Resource Waste

Figura A.4: Performance of the application scheduling for granularities $3Mbytes$ and $15MBytes$

In order to analyze the influence of the different characteristics of PHD applications on the application makespan and the resource waste, we have considered two types of applications (see Section A.4.2). The results show that the application makespan has not been affected by the different characteristics of the application. On the other hand, we found out that the waste of resources was affected by the type of application considered. Figure A.5(a) and Figure A.5(b) show the results attained.

Recall that in the data visualization application, the computational cost of the task is



(a) Scientific Visualization Application

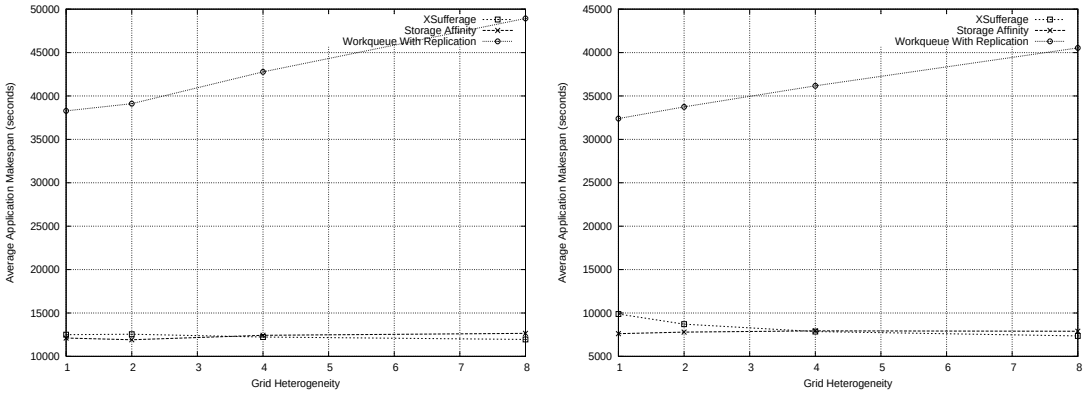
(b) Pattern Search Application

Figura A.5: Impact of application type on the level of resources wasted

completely determined by the size of its input data. Since Storage Affinity prioritizes the task with the largest *storage affinity* value, it means that the largest tasks are scheduled first. Therefore, task replication only starts when most of the application has already been executed. In the case of the pattern search application, the computational cost of the tasks is not completely determined by the size of the input data of the task, thus large tasks can be scheduled at later stages in the execution of the application. Therefore, replication may start when a large portion of the application is still to be accomplished, and consequently more resources are wasted to improve the application makespan.

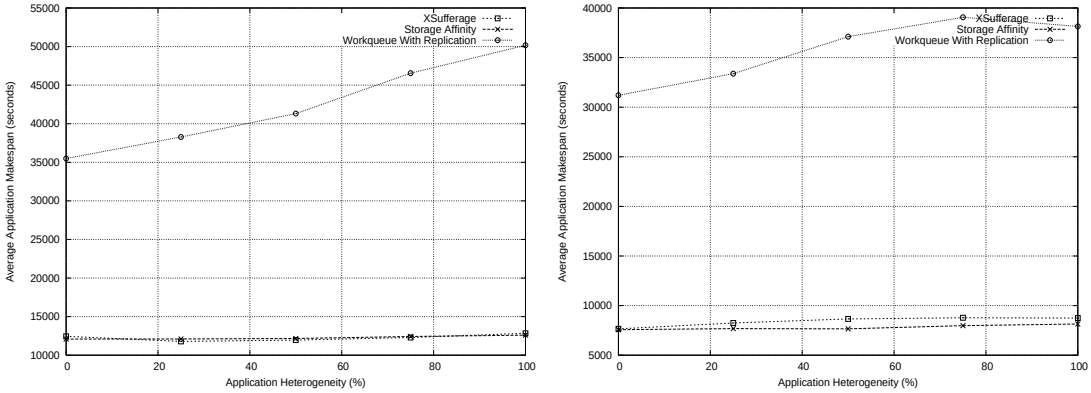
Finally, we have analyzed the heterogeneity of the grid and the application in both the data visualization application and in the pattern search application. In Figure A.6(a) and Figure A.6(b) we can see how the heterogeneity of the grid impacts the makespan of both types of applications, considering the three heuristics discussed. The two data-aware heuristics are not greatly affected by the variation of the grid heterogeneity. It is not surprising that XSufferage presents this behavior, given that it uses information about the environment. The similar behavior of Storage Affinity shows that its replication strategy circumvents the effects of the variations of the speed of processors in the grid.

The data-aware heuristics also present a good tolerance to the application heterogeneity variation. In Figure A.7, we observe that the application makespan presents a tiny variation for both types of application.



(a) Scientific data visualization (b) Pattern search

Figura A.6: Impact of grid heterogeneity



(a) Scientific visualization model (b) Pattern search model

Figura A.7: Impact of application heterogeneity

A.5 Conclusions and future work

In this paper we have presented Storage Affi nity, a novel heuristic for scheduling PHD on grid environments. We have also compared its performance against that of two well-established heuristics, namely: XSufferage and WQR. The former is a knowledge-centric heuristic that takes data transfer delays into account, while the latter is a knowledge-free approach, that uses replication to cope with ineffi cient task-processor assignments, but does not consider data transfer delays. Storage Affi nity also uses replication and avoids unnecessary data transfers by exploiting a data reutilization pattern that is commonly present in PHD applications. In contrast with the information needed by XSufferage and approaches based on network bandwidth knowledge [13], the data location information required by Storage Affi nity is

trivially obtained, even in grid environments.

Our results show that taking data transfer into account is mandatory to achieve efficient scheduling of PHD applications. Further, we have shown that grid and application heterogeneity have little impact in the performance of the schedulers. On the other hand, the granularity of the application has an important impact on the performance of the two data-aware schedulers analyzed. Storage Affinity is outperformed by XSufferage only when application granularity is large. However, the granularity of PHD applications can be easily reduced to levels that make Storage Affinity always outperform XSufferage. In fact, independently of the heuristic used, the smaller the application granularity the better the performance of the scheduler, thus reducing the application granularity is something to be pursued. In the favorable scenarios, Storage Affinity achieves a makespan that is in average 42% smaller than XSufferage. The drawback of Storage Affinity is the waste of grid resources due to its replication strategy. Our results show that the wasted bandwidth is negligible and the wasted CPU can be reduced to less than 31%

As future work, we intend to investigate the following issues: i) the impact of the inter-task data reutilization pattern on application scheduling; ii) disk space management on data servers; iii) the emergent behavior of a community of schedulers competing for shared resources; and iv) the use of introspection techniques for data staging [64] to provide the scheduler with information about data location and disk space utilization. Finally, we are about to release a stable version of Storage Affinity within the MyGrid middleware [62] [8]. We hope that practical experience with the scheduler will help us to identify aspects of our model that need to be refined.

Bibliografia

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 1, no. 215, pp. 403–410, 1990.

[2] "Blast webpage." http://www.ncbi.nlm.nih.giv/BLAST.

[3] G. Group, "http://www.griphyn.org," 2002.

[4] E. L. Santos-Neto, L. E. F. Tenório, E. J. S. Fonseca, S. B. Cavalcanti, and J. M. Hickmann, "Parallel visualization of the optical pulse through a doped optical fiber," in *Proceedings of Annual Meeting of the Division of Computational Physics (abstract)*, June 2001.

[5] T. Davis, A. Chalmers, and H. W. Jensen, "Practical parallel processing for realistic rendering," in *ACM SIGGRAPH*, 2000.

[6] C. Lee and M. Handi, "Parallel image processing applications on a network of workstations," *Parallel Computing*, vol. 21, pp. 137–160, 1995.

[7] P. Lyman, H. R. Varian, J. Dunn, A. Strygin, and K. Swearingen, "How much information?." http://www.sims.berkeley.edu/research/projects/how-much-info-2003, October 2003.

[8] 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 the ICCP'2003 - International Conference on Parallel Processing*, October 2003.

[9] J. Smith and S. K. Shrivastava, “A system for fault-tolerant execution of data and compute intensive programs over a network of workstations,” in *Lecture Notes in Computer Science*, vol. 1123, IEEE Press, 1996.

[10] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima, “Exploiting replication and data reuse to effi ciently schedule data-intensive applications on grids (*submitted to publication*),” October 2003.

[11] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a Future Computing Infrastructure*. 1999.

[12] F. Berman, A. Hey, and G. Fox, eds., *Grid Computing - Making the Global Infrastructure a Reality*. John Wiley and Sons, Ltd, 2003.

[13] O. Beaumont, L. Carter, J. Ferrante, and Y. Robert, “Bandwidth-centric allocation of independent task on heterogeneous plataforms,” in *Proceedings of the Internetional Parallel and Distributed Processing Symposium*, (Fort Lauderdale, Florida), April 2002.

[14] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, “Heuristics for scheduling parameter sweep applications in grid environments,” in *Proceedings of the 9th Heterogeneous Computing Workshop*, (Cancun, Mexico), pp. 349–363, IEEE Computer Society Press, May 2000.

[15] M. Faerman, R. W. A. Su, and F. Berman, “Adaptive performance prediction for distributed data-intensive applications,” in *Proceedings of the ACM/IEEE SC99 Conference on High Performance Networking and Computing*, (Portland, OH, USA), ACM Press, 1999.

[16] K. Marzullo, M. Ogg, A. R. amd A. Amoroso, A. Calkins, and E. Rothfus, “Nile: Wide-area computing for high energy physics,” in *Proceedings 7th ACM European Operating Systems Principles Conference. System Support for Worldwide Applications*, (Connemara, Ireland), pp. 54–59, ACM Press, Sept. 1996.

[17] 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*

the Euro-Par 2003: International Conference on Parallel and Distributed Computing, (Klagenfurt,Austria), August 2003.

[18] "Seti@home site." http://setiathome.ssl.berkeley.edu.

[19] "United devices site." http://www.ud.com.

[20] "Compute against the cancer site." http://www.computeagainstcancer.org.

[21] G. H. Page, "The great internet mersenne prime search." http://www.merssene.org, December 2003.

[22] S. Hastings, T. Kurc, S. Lamgella, U. Catalyurek, T. Pan, and J. Saltz, "Image processing for the grid: A toolkit for building gird-enabled image processing applications," in *3rd IEEE/ACM Internetional Symposium on Cluster Computing and the Grid*, 2003.

[23] J. Basney, M. Livny, and P. Mazzanti, "Harnessing the capacity of computational grids for high energy physics," in *Conference on Computing in High Energy and Nuclear Physics*, 2000.

[24] T. O. Laboratory, "Fight aids@home." http://fi ghtaidsathome.scripps.edu, 2003.

[25] H. Kreger, "Web services conceptual architrecture." www-3.ibm.com/software/solutions/ webservices/pdf/WSCA.pdf, 2003.

[26] MPI, "Mpi forum." http://www.mpi-forum.org, December 2003.

[27] PVM, "Parallel virtual machine." http://www.csm.ornl.gov/pvm/pvm_home.html, December 2003.

[28] HPF, "High performance fortran." http://www.crpc.rice.edu/HPFF/home.html, December 2003.

[29] OpenMP, "Simples, portable, scalable smp programming." http://www.openmp.org, December 2003.

[30] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[31] 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*, vol. 12, pp. 53–65, 1996.

[32] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997.

[33] A. S. Grimshaw, W. A. Wulf, and T. L. Team, "The legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[34] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-level scheduling on distributed heterogeneous networks," pp. ??–??, 1996.

[35] F. Berman and R. Wolski, "Scheduling from the perspective of the application," in *HPDC*, pp. 100–111, 1996.

[36] H. Casanova, G. Obertelli, F. Berman, and R. wolski, "The apples parameter sweep template: User-level middleware for the grid," in *Supercomputing Conference (SC'2000)*, 2000.

[37] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI–BLAST: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, pp. 3389–3402, 1997.

[38] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[39] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne, "The protein data bank," *Nucleic Acids Research*.

[40] D. Thain, J. Basney, S.-C. Son, and M. Livny, "The kangaroo approach to data movement on the grid," in *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, May 2001.

[41] W. R. Elwasif, J. S. Plank, and R. Wolski, "Data staging effects in wide area task farming applications," in *IEEE Internetional Sysmposium on Cluster Computing and the Grid*, (Brisbane, Australia), IEEE Computer Society Press, May 2001.

[42] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tueck, "Gridftp protocol specification."

[43] R. Oldfield, "Summary of existing and developing data grids - draft," in *Grid Forum. Remote Data Access Working Group*, IEEE Computer Society Press, Mar. 2001.

[44] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and C. S. S. Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23:187-200, 2001.

[45] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," in *High Performance Distributed Computing, 2002. Proceedings. 11th IEEE International Symposium*, (Edinburg, Scotland), IEEE Computer Society Press, July 2002.

[46] 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." Global Grid Forum - GGF, 2002.

[47] G. Alliance, "Ogsa site." http://www.globus.org/ogsa, 2003.

[48] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt, "OGSI specification." GGF Document (http://www.ggf.org/ogsi-wg), June 2003.

[49] A. Downey, "Predicting queue times on space-sharing parallel computers," in *Proceedings of 11th International Parallel Processing Symposium (IPPS'97)*, April 1997.

[50] R. Gibbons, "A historical application profiler for use by parallel schedulers," *Lecture Notes in Computer Science*, vol. 1297, pp. 58–75, 1997.

[51] W. Smith, I. Foster, and V. Taylor, "Predicting application run times using his-torical information," *Lecture Notes in Computer Science*, vol. 1459, pp. 122–142, 1998.

[52] R. Wolski, N. Spring, and J. Hayes, "Predicting the cpu availability of time-shared unix systems on the computational grid," in *Proceedings of 8th International Symposium on High Performance Distributed Computing (HPDC'99)*, August 1999.

[53] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jim, "An architecture for a global internet host distance estimation service," in *Proceedings of IEEE INFOCOM*, 1999.

[54] J. Stiles, T. Bartol, E. Salpeter, and M. Salpeter, "Monte carlo simulation of neuromuscular transmitter release using mcell, a general simulator of cellular physiological processes," *Computational Neuroscience*, 1998.

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

[56] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM (JACM)*, vol. 24, no. 2, pp. 280–289, 1977.

[57] D. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson and L. Rudolph, eds.), vol. 1459, pp. 1–24, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[58] D. G. Feitelson, "Metric and workload effects on computer systems evaluation," *Computer*, vol. 36(9), pp. 18–25, September 2003.

[59] V. Lo, J. Mache, and K. Windisch, "A comparative study of real workload traces and synthetic workload models for parallel job scheduling."

[60] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 757–768, 1999.

[61] H. Casanova, "Simgrid: A toolkit for the simulation of application scheduling," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, May.

[62] "Mygrid site." http://www.ourgrid.org/mygrid.

[63] J. L. Devore, *Probability and Statistics for Engineering and The Sciences*, vol. 1. John Wiley and Sons, Inc., 2000.

[64] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Nov. 2000.