RESUMO

O objetivo de uma grade computacional é o de agregar uma coleção de recursos *distribuídos*, *heterogêneos* e *compartilhados* para oferecer poder computacional para aplicações. Contudo, permanece como desafio a exploração eficiente do desempenho deste ambiente, devido principalmente às três características supracitadas. Um dos fatores cruciais para o bom aproveitamento do potencial de uma grade é uma eficiente alocação das tarefas aos processadores disponíveis no ambiente. O *problema do escalonamento de tarefas* trata exatamente esta questão. Por ser um problema NP-difícil, o desenvolvimento de algoritmos de escalonamento que produzam bons escalonamentos em tempos viáveis é necessário, mas também é um desafio. Este trabalho apresenta uma ferramenta para facilitar o desenvolvimento de estratégias de escalonamento, bem como a análise do desempenho oferecido por diferentes políticas de escalonamento das aplicações paralelas executando em ambientes grades.

SUMÁRIO

1. Introdução 4

1.1 Apresentação – Contexto e Objetivos 7

1.2 Organização 8

2. O problema de escalonamento 9

2.1 Modelo de Aplicação 11

2.2 Modelos de Arquitetura 12

2.3 Processadores Homogêneos ou Heterogêneos 12

2.4 Modelos de comunicação 13

2.4.1 Modelo Latência 13

2.4.2 Modelo LogP 14

2.4.3 Modelo HLogP 15

2.5 Escalonamento de Tarefas 15

2.6 Escalonamento em grades Computacionais 18

2.7 List Scheduling 19

2.7.1 List Scheduling Configurável 20

2.7.2 Escolha da tarefa 21

2.7.3 Escolha do processador 24

2.8 Outras heurísticas para comparação 25

2.8.1 O algoritmo ETF 25

2.8.2 O algoritmo DCP 26

2.8.3 O algoritmo HEFT 26

2.8.4 Algoritmos *List Scheduling* para o modelo HLogP 27

2.9 Resumo 27

3. *Overview* da estrutura da ferramenta 28

3.1 O *Framework EasyGrid* 28
3.2 O Projeto, a Avaliação e a Validação de Algoritmos de Escalonamento 30
3.3 Fluxograma de funcionamento 31
3.4 Implementação, Atualizações e Manutenção 36
3.5 Resumo 37

4. A ferramenta *Task Scheduling Testbed* 38
4.1 Projeto de um algoritmo 38
4.2 Avaliação dos algoritmos de escalonamento estático 43
4.3 Validação dos escalonamentos e a execução de aplicações utilizando o *Portal Easygrid* 46
4.4 Resumo 51

5. Conclusões e trabalhos futuros 52

6. Apêndices 54
Apêndice A - Formatos dos arquivos de entrada e saída da ferramenta 54
Apêndice B – Publicações 60

7. Referências 61

1 INTRODUÇÃO

Atualmente observa-se um número crescente de aplicações, científicas e comerciais, que necessitam de um poder computacional que muitas vezes não pode ser obtido com um computador só. Uma maneira de suprir esta necessidade seria utilizar máquinas com processadores e outros componentes mais velozes, mas além das limitações econômicas temos as limitações da física que precisam ser superadas para o desenvolvimento de máquinas mais poderosas. A conexão de múltiplos processadores de maneira que proporcionem um maior desempenho computacional é uma alternativa mais viável. Estes sistemas são conhecidos como computadores paralelos. Os supercomputadores, por exemplo, são construídos como uma única máquina (com até milhares de processadores interconectados por uma rede especializada) que ocupa muitas vezes um prédio inteiro. Por isso possuem desenvolvimento e manutenção muito caros, o que torna sua utilização extremamente restrita. Uma alternativa mais econômica é conhecida como computação em clusters, que se resume no agrupamento de computadores comuns como estações de trabalho ou PCs interconectados por uma rede de baixo custo disponível no mercado (por exemplo, rede *Ethernet*). Enquanto não oferecem o poder computacional equivalente aos supercomputadores, os clusters buscam oferecer desempenho suficiente para a execução de aplicações de vários tipos.

Contudo, mesmo diante dos supercomputadores e clusters mais velozes existem problemas cuja solução permanece inviável, tanto em termos de tempo como de custo. Com o desenvolvimento crescente de redes de longa distância com alta velocidade, os pesquisadores notaram a possibilidade de tirar proveito dos poderosos recursos computacionais já interconectados via internet. A idéia foi unir tais recursos para que atuassem como um único recurso de alto desempenho. Este tipo de sistema foi denominado Grid (grade) pelos pesquisadores [Fost99]. Grades Computacionais (*Computational Grids*) têm o potencial de se tornarem plataformas poderosas a serem

utilizadas pela comunidade de computação distribuída, tanto científica quanto comercial, para a execução de aplicações de grande importância e alto teor computacional. Além disso, tornam a computação de alto desempenho acessível também a usuários que não necessariamente possuem recursos suficientes e disponíveis localmente para executar suas aplicações.

Computação em grades adotou o nome e o conceito das redes elétricas de potência para capturar a noção de fornecimento eficiente de poder computacional, a um custo razoável, de acordo com a demanda, para qualquer um que precisar [Fost99]. O objetivo da grade é o de agregar uma coleção de recursos distribuídos geograficamente, heterogêneos e compartilhados, sendo vistos como um único recurso computacional. A disponibilidade deste tipo de ambiente representa uma abertura para novos ramos de pesquisa que previamente encontravam-se limitados e sem exploração por razões econômicas e práticas.

Entretanto, ainda existem relativamente poucas aplicações que exploram o poder computacional deste novo ambiente de forma realmente vantajosa. Atualmente, a maioria das aplicações habilitadas às grades tem sido escrita por especialistas de grades e não cientistas, engenheiros ou programadores comuns. Por serem as grades de natureza dinâmica e heterogênea, ou seja, seus recursos podem ou não estar disponíveis em momentos distintos, e suas unidades de processamento possivelmente são de diferentes plataformas, fica muito restrito o aproveitamento neste ambiente de ferramentas de sistemas paralelos já existentes. Isso mostra o grau de dificuldade encontrado, especialmente por leigos em computação, ao propor versões das aplicações preparadas para executar de forma eficiente em grades computacionais. Não mudar essa situação seria motivo suficiente para inibir a aceitação das grades computacionais.

Usuários de sistemas paralelos encontram, com certa freqüência, uma grande dificuldade em atingir uma boa fração do pico de desempenho teórico dos sistemas paralelos e distribuídos. De fato, como as arquiteturas paralelas estão se tornando mais complexas e os softwares estão continuamente evoluindo, um número crescente de problemas relacionados ao desempenho fazem a afinação de aplicações por parte do próprio programador (usuário) uma tarefa complexa e algumas vezes contra-intuitiva.

Por exemplo, a Grade Computacional GridRio [Rebe03, Silv04], em operação desde o final de 2002, está sendo utilizada por especialistas para desenvolver *middleware* voltado para ambientes grade e também por físicos, para executar aplicações cientificas. O poder computacional agregado (e compartilhado entre usuários

de três instituições – IC/UFF, DI/PUC-Rio e CAT/CBPF) atualmente é de 94 processadores com um limite de desempenho de mais de 300 GFLOPs, sendo que os processadores variam desde Pentium II 233Mhz (com 128MB de memória) até Pentium IV 3.2 Ghz (1GB de memória) e sistemas bi-processados [GridRioWeb]. Ainda neste ano espera-se que a infra-estrutura do GridRio venha a atingir mais de 200 processadores e capacidade máxima de processamento de 0.8 TFLOPs. Em um ambiente tão variado é extremamente difícil que usuários (tipicamente cientistas e engenheiros com pouco conhecimento do sistema) sejam capazes de decidir a cada momento de execução e para cada aplicação, quais recursos são mais apropriados para serem utilizados.

O projeto *EasyGrid* [Rebe03, Boer04] propõe-se a atacar e resolver o desafio de habilitar aplicações para executar de forma eficiente em ambientes grade, evitando assim que esse trabalho árduo seja realizado pelo próprio usuário. O ambiente *EasyGrid* permite que programadores possam se concentrar na exploração do paralelismo para resolver o problema em si, deixando que o ambiente gere a *aplicação grade* capaz de utilizar da melhor forma os recursos disponíveis. O ambiente *EasyGrid* deve fazer com que todos os aspectos relacionados a grade sejam transparentes ao usuário. Para isso, aplicações paralelas são transformadas automaticamente em aplicações *system-aware*, ou seja, aplicações cientes do sistema e que se automodificam, visando uma adequação às mudanças dinâmicas ocorridas no ambiente de execução.

Inicialmente, o projeto *EasyGrid* focaliza aplicações paralelas escritas em MPI, devido ao seu grande uso em aplicações científicas e na área da programação paralela. Para a aplicação obter um desempenho aceitável é crucial uma boa alocação dos processos nos processadores disponíveis. O escalonamento de tarefas, que tem como função minimizar o tempo de execução de aplicações paralelas, é um problema muito difícil, dito NP-completo [Ullm75]. Mesmo com simplificações na arquitetura do programa paralelo e na máquina paralela o problema permanece NP - completo, exceto em alguns casos especiais [Gare79]. Por este motivo são propostas heurísticas de escalonamento, que buscam obter bons escalonamentos em tempo viável.

É importante para o bom desempenho destas heurísticas que sejam devidamente modeladas as características da aplicação e da arquitetura em foco. Este conjunto de características é denominado modelo de computação paralela. Existem diversos modelos propostos na literatura, que variam dos mais abstratos aos mais realísticos.

Outra característica importante com relação ao escalonamento é quanto ao momento em que este ocorre em relação à execução da aplicação. Se o escalonamento ocorre no momento da compilação da aplicação é dito estático, e se ocorre durante sua execução é dito dinâmico.

1.1 Apresentação – Contexto e Objetivos

Um dos enfoques do projeto *EasyGrid* está no desenvolvimento de heurísticas de escalonamento que sejam direcionadas especificamente para a grade computacional (considerado um dos maiores desafios da área de pesquisa em grades atualmente [Krau01]). Este trabalho aqui apresentado enfoca o problema de desenvolver heurísticas de escalonamento apropriadas para o *framework EasyGrid* e grades computacionais em geral. Para minimizar a complexidade de um escalonador dinâmico, utilizado durante a execução da aplicação e necessário neste ambiente devido ao seu dinamismo e heterogeneidade, é proposto um escalonador estático que fará o pré-escalonamento das tarefas, restando ao escalonador dinâmico apenas os ajustes necessários [Boer03].

Para isso, é apresentada uma ferramenta gráfica que vem facilitar principalmente o desenvolvimento e análise de algoritmos de escalonamento de tarefas especificamente voltado para atender as características acima. Seus objetivos principais podem ser resumidos em:

- [**Projeto**] Permitir a investigação de abordagens e (combinações de) técnicas de escalonamento (por exemplo, replicação de tarefas, prioridades das tarefas, regras para escolher processadores).
- [**Avaliação**] A avaliação de novos algoritmos em termos de *makespan* (tempo de término da aplicação paralela) e processadores utilizados. A avaliação experimental requer a comparação com outras heurísticas existentes para que assim o comportamento dos novos algoritmos, considerando várias instâncias, seja traçado.
- [**Validação**] A verificação e validação dos escalonamentos gerados pelos algoritmos podem ser feitas por simulação ou via execução num ambiente grade real.

Além de ajudar projetistas de algoritmos de escalonamento, a ferramenta, através de um *portal,* também ajuda usuários comuns nas tarefas (algumas complexas e/ou monótonas) necessárias para acessar os recursos da grade e executar suas aplicações.

1.2 Organização

A Seção 2 apresenta uma descrição do problema de escalonamento, os modelos de aplicação e arquitetura e as principais heurísticas de escalonamento utilizadas. Para o problema de escalonamento em grades computacionais pretende-se focar na abordagem *List Scheduling* e este capítulo também descreve em detalhes as características das heurísticas de L*ist Scheduling*, como por exemplo, as prioridades mais usadas. Ainda neste capítulo são descritos três dos principais algoritmos propostos na literatura para esta classe de heurísticas e que estão disponíveis na ferramenta para serem utilizados nos testes comparativos.

Na Seção 3 inicialmente é apresentada uma descrição do *framework EasyGrid.* Em seguida apresenta-se uma visão geral da ferramenta proposta seguida de um fluxograma que ilustra as funcionalidades oferecidas. No final do capítulo abordamos os principais detalhes de implementação da ferramenta.

A Seção 4 mostra o funcionamento da ferramenta em mais detalhes. Cada fase do desenvolvimento de um algoritmo é apresentada de modo a ilustrar o funcionamento da ferramenta como um todo.

Na Seção 5 são apresentadas as conclusões obtidas, além de propostas de outros trabalhos a serem desenvolvidos com base nos resultados do presente trabalho.

2 O PROBLEMA DE ESCALONAMENTO

Normalmente, as duas razões mais comuns que estimulam o desenvolvimento de programas paralelos em vez de seqüenciais são: achar a solução do problema mais rapidamente e resolver versões mais complexas do problema. Desenvolver tais programas de forma eficiente e com alto desempenho, não é uma tarefa fácil.

O objetivo principal do escalonamento é a execução eficiente de uma aplicação paralela considerando as restrições impostas pelo sistema alvo. Ao se tratar de grades, é muito importante também determinar o uso eficiente dos recursos para as várias aplicações que venham a ser executadas em tal sistema, mas tendo em mente que cada uma dessas aplicações deve ser executada em tempo razoável.

O escalonamento de tarefas, cuja função é minimizar o tempo de execução de programas paralelos através da alocação cuidadosa de tarefas nos processadores disponíveis, é uma atividade crucial quando se fala em rapidez de execução de aplicações paralelas. Porém, esta atividade de escalonar as tarefas de uma aplicação paralela tal que o tempo do escalonamento seja mínimo, é um problema NP-completo [Ullm75]. Mesmo com simplificações na arquitetura do programa paralelo e na máquina paralela, tais como: custo de execução das tarefas uniforme, custo de comunicação entre as tarefas inexistentes e disponibilidade ilimitada de processadores, ainda assim o problema permanece NP-completo exceto por alguns casos restritos [Gare79]. Deste modo, heurísticas de escalonamento são propostas para se tentar obter bons escalonamentos em tempos razoáveis [Kwok96].

Para que uma heurística possa obter bons resultados é necessário também que as características da aplicação e da arquitetura sejam devidamente modeladas, retratando os principais aspectos a serem considerados durante o escalonamento das tarefas. O conjunto destas características é chamado de modelo de computação paralela, e variam desde os mais abstratos aos mais realísticos [Este03]. Um outro fator relevante com

relação ao escalonamento é o instante em que esse ocorre em relação à execução da aplicação. Se ocorrer até o momento da compilação é denominado *escalonamento estático* e se ocorrer durante a execução é denominado *escalonamento dinâmico*.

Em uma grade, o problema de escalonar tarefas torna-se ainda mais complexo por causa de alguns requisitos como:

- Necessidade de um modelo de computação paralela que represente com sensibilidade o custo de comunicação, o dinamismo e a diversidade dos recursos.
- Um escalonador estático de baixa complexidade para que o tempo gasto para gerar um escalonamento não seja demasiadamente alto. Outra característica desejável neste ambiente é um escalonador dinâmico também de baixa complexidade, pois em um ambiente como a grade é preciso tomar decisões no momento em que ocorre o escalonamento, obrigando ao escalonador a possuir baixa complexidade para que ele não acrescente grandes sobrecargas ao sistema;
- Utilização de um número variável de processadores heterogêneos, sendo que, cada um deles possui recursos com disponibilidades variadas;
- Implementação de mecanismos integrados de tolerância a falhas, visto que recursos podem deixar de estar disponíveis durante a execução da aplicação [Nguy00].

Um dos segmentos do projeto EasyGrid consiste em buscar heurísticas e técnicas de escalonamento que sejam especificamente direcionadas ao novo enfoque de computação paralela, as grades computacionais[EasyGridWeb]. Devido à característica dinâmica dos recursos computacionais nas grades computacionais torna-se necessário um escalonamento de tarefas que possa ser modificado durante a execução da aplicação, ou seja, um escalonamento dinâmico. Porém, para que o tempo de escalonamento da aplicação seja razoável, é necessário que a sobrecarga gerada ao sistema pelas atividades do escalonador dinâmico seja minimizada. Uma estratégia para reduzir o overhead provocado pelo escalonador dinâmico é através de um prévio escalonamento estático que já ajusta a aplicação à configuração momentânea da máquina alvo [Mahe99, Whal01].

Parte dos objetivos do projeto EasyGrid então se resume na construção de um

escalonador estático que empregue uma heurística que possa ser utilizada também pelo escalonador dinâmico. Este não deve possuir nenhuma restrição quanto ao modelo da aplicação, custo de execução, custo de comunicação, número de processadores, ou velocidades de processadores. Restando ainda a missão de ser rápido e eficiente, em termos de *makespan* (tempo de término do programa paralelo) e uso dos processadores. Além disso, é desejado também que seja implementado sobre um novo modelo de computação paralela denominado *HLogP* [Mend04], tendo em vista este modelo ser mais realístico que o modelo de latência [Papa90] usado pela maioria dos pesquisadores na área de escalonamento, e também por já ter mostrado melhores resultados que o modelo de latência para as máquinas paralelas atualmente utilizadas [Mart97].

Esse trabalho é parte integrante do projeto EasyGrid e tem por finalidade facilitar o estudo de técnicas de escalonamento estático, dinâmico e híbrido e o desenvolvimento de algoritmos de escalonamento específicos para classes de aplicações *system-aware* que então possam explorar melhor o desempenho de grades computacionais.

2.1 Modelo de Aplicação

As aplicações paralelas consideradas neste trabalho são representadas por um grafo acíclico direcionado (GAD) cujas tarefas mantêm uma relação de precedências. Um GAD é denotado por $G = (V, E, \varepsilon, \omega)$, onde V é o conjunto vértices que representam as tarefas, e E o conjunto de arcos que representa a relação de precedência entre as tarefas. Cada arco $(v_i, v_j) \in E$ representam a relação de precedência entre as tarefas v_i e v_j, ou seja, a tarefa v_i deve completar sua execução antes que v_j comece. Um peso $\omega(v_i, v_j)$ pode estar associado ao arco(v_i, v_j) representado a quantidade de dados a serem transmitidos da tarefa origem v_i para a tarefa destino v_j. A cada nó $v_i \in V$ é associado um *peso de computação* $\varepsilon(v_i)$ representando a quantidade de trabalho da tarefa em um determinado processador.

Para representar as adjacências entre as tarefas, são utilizadas duas notações: o conjunto de predecessores imediatos de uma tarefa $v_i \in$ V é denotado por pred$(v_i) = \{v_j \in V \mid (v_j, v_i) \in E\}$ e o conjunto de sucessores imediatos de v_i é denotado por succ$(v_i) = \{v_j \in V \mid (v_i, v_j) \in E\}$.

2.2 Modelos de Arquitetura

O modelo arquitetural define as características da arquitetura paralela consideradas relevantes ao problema de escalonamento. Tipicamente, os modelos utilizados supõem que o sistema computacional consiste de unidades de processamento distribuídas (cada um com sua própria memória local) que se comunica sobre uma rede de interconexão via mensagens por trocar informação. Esses aspetos são usualmente especificados em modelos separados:

- Um modelo de processamento define o conjunto, geralmente limitado, de elementos de processamento por $P = \{p_0, p_1, ..., p_n\}$, onde o poder computacional de cada elemento pode ou não ser diferente (ver Seção 2.3).
- Um modelo de comunicação tenta capturar os custos de comunicação entre os processadores e a topologia das interconexões. Uma representação que pode ser utilizada para a rede é por meio de um grafo não direcionado $G = (P, E_p)$, onde P é conjunto de vértices que representam os processadores e E_p o conjunto de arestas que representam os canais de comunicação que interligam esses processadores. A maioria dos trabalhos de escalonamento considera que os processadores são totalmente conectados [Kwok99a]

2.3 Processadores Homogêneos ou Heterogêneos

O principal objetivo de se modelar o problema do escalonamento de tarefas é avaliar como uma aplicação será executada na máquina alvo, possibilitando uma tomada de ações e decisões que resultem em um melhor desempenho. Sendo assim, escolher entre utilizar uma arquitetura que considera processadores heterogêneos, ao invés de homogêneos, é importante para retratar de forma mais exata o ambiente da máquina destino.

A arquitetura homogênea considera que todas as máquinas possuem a mesma velocidade de processamento e as configurações das máquinas são idênticas (o que na

prática é muito raro) [Mend04]. Por isso, os algoritmos que utilizam essa arquitetura normalmente obtêm escalonamentos que pouco se assemelham aos resultados reais. Entretanto pode-se utilizar uma arquitetura heterogênea que considera a diferença de velocidade entre as máquinas, para realizar escalonamentos mais precisos. O tempo de execução de uma tarefa v_i no processador p_j pode ser dado pela expressão $\varepsilon(v_i) \times fh(p_j)$, onde $fh(p_j)$ representa o fator de heterogeneidade do processador p_j.

2.4 Modelos de comunicação

Devido a constantes desenvolvimentos na máquina paralela com memória distribuída, se tornou necessário a utilização de um modelo de computação paralela que representasse de forma mais realista as máquinas e que definisse as características de comunicação de forma mais precisa. Esta subseção vai apresentar os principais modelos de comunicação adotados pela comunidade da área de escalonamento de tarefas.

2.4.1 Modelo Latência

Atualmente é o modelo padrão de comunicação utilizado na área de escalonamento de tarefas. O único parâmetro arquitetural associado ao custo de comunicação é a latência, que representa o atraso no tempo de transmissão para enviar uma unidade de dado no canal de comunicação entre dois processadores (e é denotada por L).

Neste modelo, assume-se que um processador não gasta tempo preparando o envio ou recebimento de mensagens, e assim, comunicação e computação de tarefas podem se sobrepor totalmente, e além disso, um processador pode enviar várias mensagens distintas, simultaneamente, para destinos diferentes, o que se chama *multicast*.

2.4.2 Modelo *LogP*

Na tentativa de se desenvolver um modelo mais prático foi proposto o modelo *LogP* [Cull93], que considera a existência de parâmetros como as sobrecargas de envio e de recebimento e latência. O modelo *LogP* é direcionado a rede de comunicação, não impondo nenhum estilo de programação para capturar os custos associados aos eventos de comunicação. Este modelo considera parâmetros que capturam os custos de comunicação individualmente, e que comunicação e computação não podem ser sobrepostos totalmente. Os parâmetros principais deste modelo são:

- A *latência* ***L*** na comunicação como definido no modelo de latência;
- A *sobrecarga* (*overhead* ***o***) de envio (designado aqui como O_s) ou recebimento (O_r), que é o tempo que o processador gasta preparando uma mensagem para ser enviada ou recebida respectivamente. Durante esse período o processador não pode executar nenhuma outra tarefa;
- O intervalo entre envios ou recebimentos consecutivos em um mesmo processador (*gap* ***g***); e
- ***P***, a quantidade de processadores disponíveis.

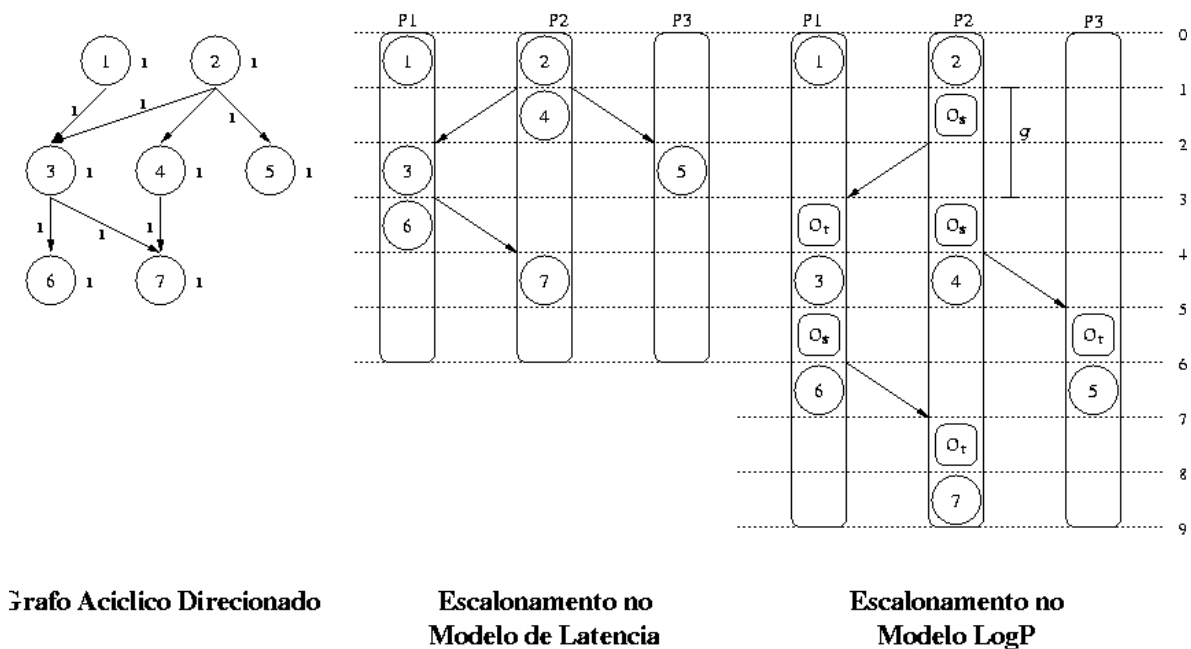

Figura 2.1 Uma comparação de escalonamentos sobre os modelos Latência e LogP

A Figura 2.1 mostra os efeitos dos parâmetros do modelo LogP no escalonamento de um grafo em comparação com o escalonamento dele no modelo latência.

Embora as características desse modelo o tornam bastante interessante e que outras variantes deste modelo já tenha sido propostas (por exemplo, considerando que os parâmetros podem variar dependendo do tamanho da mensagem) para modelar clusters [Wang99], o modelo ainda não é apropriado para ambientes grades.

2.4.3 Modelo *HLogP*

O modelo *HLogP* [Mend04] é uma extensão do modelo *LogP*, e foi proposto com a finalidade de se adequar melhor ao ambiente grade, pois considera relevante as características heterogêneas desse ambiente. A principal diferença entre o modelo LogP e o modelo HLogP é que o modelo *LogP* considera a comunicação entre as máquinas, os *overheads* de comunicação e a diferença de capacidade de processamento das unidades de processamento constantes em relação ao tamanho da mensagem. Já o modelo *HLogP* considera que esses mesmos parâmetros podem variar em função do tamanho da mensagem, fazendo com que o modelo *HLogP*, dentre os apresentados, seja o mais apropriado para o ambiente grade.

2.5 Escalonamento de Tarefas

Como primeiro passo para compreender a discussão sobre escalonamento de tarefas é importante destacar as diferenças existentes nesta área. Inicialmente o escalonamento pode ser classificado de acordo com o número de processadores a serem considerados no sistema. Quando a atribuição das tarefas de um programa paralelo é feita a um único processador disponível, o escalonamento é dito local. No caso onde as tarefas são designadas a vários processadores, que compõem a plataforma computacional distribuída, o escalonamento é denominado global [Este03].

Outra classificação que diz respeito ao escalonamento global é o fato do escalonamento ser estático ou dinâmico. Se as decisões de escalonamento ocorrem no máximo em tempo de compilação do programa, ou seja, antes de sua execução o escalonamento é dito estático. Já no escalonamento dinâmico as decisões são tomadas

durante a execução da aplicação e normalmente há muito pouco ou nenhum conhecimento prévio sobre a aplicação e a arquitetura.

O escalonamento estático possui como vantagem o fato de não impor sobrecarga para determinar o escalonamento em tempo de execução da aplicação. Por outro lado, suas desvantagens estão na exigência de um alto grau de conhecimento prévio a respeito do comportamento da aplicação em relação à plataforma usada e na ineficiência dos mecanismos que estimam os custos de computação e comunicação, o que causa sérias degradações no desempenho esperado.

O escalonamento dinâmico tem como vantagem à característica de se adequar melhor a plataformas onde a estrutura computacional pode ser dinamicamente alterada. Já a sua maior desvantagem, como já citado anteriormente, é a sobrecarga gerada no sistema devido às decisões tomadas durante o escalonamento em tempo de execução [Este03].

Além disso, uma vez que a tarefa começa a ser executada ela não poderá ser interrompida, para a execução de uma outra tarefa, o que é conhecido como execução em modo de não preempção. Na realidade, o processo corrente pode estar sujeito a uma faixa de tempo compartilhado, isso depende exclusivamente da política utilizada pelo sistema operacional, contudo o problema considera que duas tarefas pertencentes a um mesmo GAD não podem compartilhar uma mesma faixa de tempo no mesmo processador.

Um escalonamento S é um conjunto finito de tuplas (v_i, p_j, tl), cada uma especificando que uma tarefa $v_i \in V$ é executada no processador $p_j \in P$ no tempo tl. Dessa forma, cada tarefa $v_i \in V$ deve ser escalonada em pelo menos um processador $p_j \in P$ e essa tarefa só pode ser executada depois que todos os seus predecessores imediatos tenham completado a sua execução e que os respectivos dados necessários para o início da sua execução estejam disponíveis no processador p_j. Foi considerada a possibilidade dos algoritmos utilizarem a técnica de replicação de tarefas [Papa90].

Considerando que este estudo, também, adota o modelo de comunicação *HLogP*, definido anteriormente, é preciso enfatizar que após a execução de uma tarefa de computação v_i em um processador p_j , uma série de tarefas de sobrecargas de envio da comunicação, representadas por O_s, são escalonadas no processador p_j, sempre que existir sucessores imediatos da tarefa v_i, alocados a processadores diferentes. Desta mesma forma, uma série de tarefas de sobrecargas de recebimento de comunicação,

representadas por O_r, precisam ser escalonadas nos respectivos processadores que executarão as tarefas sucessoras de v_i, antes da execução de qualquer outra tarefa. Assim, é garantido que uma tarefa somente inicia a sua execução após receber todos os dados, enviados por mensagens vindas de seus predecessores remotos.

Quando tarefas adjacentes são alocadas a um mesmo processador tais sobrecargas de envio e recebimento podem ser desprezadas, portanto, consideradas nulas. Além disso, é necessário definir as seguintes diretrizes para o escalonamento:

- Cada tarefa $v_i \in GAD$ deve ser executada em pelo menos um processador. Dessa forma, cada tarefa v_i está associada a pelo menos uma tupla do escalonamento S;
- Duas ou mais tarefas não podem ser executadas ao mesmo tempo pelo mesmo processador;
- Uma tarefa de computação não pode enviar qualquer resultado de seus dados, antes de ter completado sua execução;
- Uma mensagem de transferência de dados não pode ser recebida a menos que tenha sido enviada. Existe, portanto uma demora de pelo menos L, que é a latência para a mensagem circular na rede de interconexão, entre uma tarefa de sobrecarga de envio e a respectiva tarefa de sobrecarga de recebimento;
- O escalonamento é concluído quando todas as tarefas do *GAD* estão escalonadas;
- O modo de execução das tarefas nos processadores é sem interrupção, ou seja, não preemptivo;

Uma vez definidos os modelos, da aplicação e da arquitetura, e estabelecidos os critérios sob o qual se realizará o escalonamento, é possível determinar, através da heurística desenvolvida, o tempo de execução paralelo final da aplicação, o *Schedule Length* (SL) ou *makespan* que é dado pela seguinte expressão:

$$\text{Se} \quad T_{\text{final}}(v_i) = \min_{\nabla\, p_j \in \{(v_i,\, p_j,\, tl)\} \subset S} \{T_{\text{inicial}}(v_i, p_j) + \varepsilon(v_i) \times fh(p_j)\}$$

$$\text{então} \quad SL(G) = \max_{\nabla\, v_i \in V} \{T_{\text{final}}(v_i)\}$$

Onde $T_{inicial}(v_i, p_j)$ é o tempo *tl* no qual uma tarefa v_i inicia a sua execução em um processador p_j; $\varepsilon(v_i)$ é o peso de computação de v_i; e $fh(p_j)$ representa o fator de heterogeneidade do processador p_j.

2.6 Escalonamento em grades Computacionais

Uma grade computacional é uma plataforma computacional distribuída e heterogênea, composta por elementos de processamento geograficamente distribuídos e compartilhados que estão conectados por uma rede de comunicação. Devido a estes aspectos pode-se notar que qualquer desenvolvimento de aplicações, gerenciamento de recursos ou escalonamento de tarefas, neste ambiente, será uma tarefa complexa. A natureza dinâmica dos recursos disponíveis em uma grade se deve a vários motivos dentre eles, o compartilhamento de recursos pelos usuários, a política de utilização adotada pelo proprietário destes recursos, possíveis falhas e atualizações. Quanto à natureza heterogênea, esta tem origem nos diferentes tipos de recursos que compõe a *grade*, desde *PCs* individuais e *NOWs (network of workstations*) até supercomputadores paralelos, o que resulta em diferentes níveis de desempenho.

Devido a estas características, o escalonamento em grade*s* computacionais é ainda um tema em aberto. Contudo, é possível inferir que um escalonamento estático não será muito eficaz devido à dificuldade de se modelar previamente o ambiente computacional, requisito indispensável para a tomada de decisões do escalonador estático. Entretanto, também é conhecido que o escalonamento dinâmico, que seria o mais indicado nestas condições, gera um *overhead* significativo ao sistema devido a sua alta complexidade. Diante disto, o projeto EasyGrid decidiu aplicar as duas estratégias, a estática e a dinâmica. Desta forma um escalonamento estático inicial seria feito para fins de reduzir ao máximo, a carga de trabalho imposta ao escalonador dinâmico. Em seguida, um escalonamento dinâmico concluiria o trabalho, utilizando em primeira instância a mesma heurística básica empregada pelo escalonador estático, porém agora com sua carga de trabalho reduzida.

Com a finalidade de viabilizar estudos comparativos entre algoritmos de escalonamento foi implementado um ambiente para o desenvolvimento e avaliação de algoritmos baseado na heurística *List Scheduling*, usando o modelo *HLogP* [Mend04]. A abordagem *List Scheduling* foi escolhida devido a seu uso popular na literatura de escalonamento de tarefas e sua baixa complexidade que permite uma execução rápida. O ambiente permite ser modificado características como qual prioridade será utilizada para ordenar as tarefas e como será escolhido o processador. A ferramenta assim fornece ao usuário a oportunidade de realizar testes comparativos entre algoritmos *List Scheduling* com características diferenciadas. Em seguida, a heurística *List Scheduling* é detalhada em busca de uma estratégia que melhor se adapte as necessidades da grade.

2.7 List Scheduling

A classe de heurísticas do tipo *List Scheduling* é composta de heurísticas de escalonamento nas quais as tarefas de uma aplicação paralela são ordenadas em uma lista, segundo algum tipo de prioridade pré-determinada. As tarefas são posteriormente designadas, uma a uma, aos processadores de acordo com a ordem na lista. Basicamente, esta metodologia segue a seguinte regra: a primeira tarefa livre da lista ordenada deve ser alocada em um processador ocioso tal que o seu tempo de fim seja o mais cedo possível. A estrutura básica do *List Scheduling* está representada no *framework* a seguir:

Algoritmo List Scheduling
{
Definir a prioridade a ser atribuída às tarefas do grafo G=(V,E);
Ordenar as tarefas livres por suas prioridades em uma lista;
Enquanto existir tarefas a serem escalonadas faça
{
- *Selecionar a primeira tarefa* $v_i \in V$ *na lista ordenada de tarefas livres;*
- *Selecionar um processador ocioso qualquer* p_j *onde* v_i *possa começar o mais cedo possível;*
- *Escalonar* v_i *no processador* p_j*;*
- *Determinar as novas tarefas livres;*

}
}

A principal diferença dentre os vários algoritmos desta classe está principalmente na prioridade utilizada para escolher a próxima tarefa a ser escalonada [Kwok99b].

2.7.1 *List Scheduling* Configurável

Com o objetivo de analisar o comportamento da heurística *List Scheduling* foi desenvolvido um algoritmo configurável baseado nessa heurística para ambientes heterogêneos com um número limitado de processadores que considerando o modelo *HLogP*. A estratégia implementa uma lista dinâmica de tarefas e considera diferenças no poder computacional entre os processadores, como também, diferentes latências e sobrecargas de comunicação. Obviamente, ambientes homogêneos e número ilimitado de processadores e modelos de comunicação mais simples como o modelo de latência também podem ser utilizados. É oferecida ao usuário uma variedade de alternativas a serem utilizadas na busca por um melhor escalonamento. As alternativas foram disponibilizadas de forma independentes, possibilitando que o usuário possa combiná-las para analisar a relação entre elas. A seguir serão mostradas algumas dessas alternativas:

- ***Prioridades para escolha da próxima tarefa*** - a maioria das funções utilizadas por algoritmos do tipo *List Scheduling*, (*nível, conível, ALAP, CP,* e outros [Kwok96]) como prioridades, foram adaptadas para ambientes heterogêneos (maiores detalhes são apresentado na Subseção 2.7.2). Por exemplo, o *nível* estático de uma tarefa, que representa o custo do maior caminho da tarefa até um destino, é calculado utilizando o poder computacional médio dos processadores disponíveis e o custo médio de comunicação [Topc02];
- ***Versões dinâmicas das prioridades*** - durante o processo de escalonamento, à medida que tarefas são alocadas, suas prioridades podem mudar devido à eliminação dos custos de comunicação ou a determinação do custo de computação. Assim, a atualização das prioridades após o escalonamento de cada tarefa leva à obtenção de informações mais exatas sobre a real

importância das tarefas nas decisões a serem tomadas nas iterações seguintes;

- ***Três níveis de prioridade*** - em casos de empates entre as prioridades das tarefas é possível utilizar uma segunda prioridade para a escolha da próxima tarefa a ser escalonada, caso ainda persista o empate, uma terceira prioridade poderá ser utilizada. Fundamentalmente, empates eram solucionados de forma aleatória [Kwok99b];
- ***Opções na escolha dos processadores*** - muitos algoritmos propostos apenas buscam o processador que oferece o menor tempo de inicio para a tarefa em questão [Kwok96]. Entretanto, na subseção 2.7.3, definimos outros critérios de decisão, para as situações em que vários processadores oferecem os mesmos tempos de inicio para a tarefa se executada.

2.7.2 Escolha da tarefa

Com o objetivo de determinar qual prioridade poderia ser melhor empregada no escalonador do EasyGrid, foram implementadas algumas prioridades nos algoritmos *List Scheduling* contudo, devido ao modelo arquitetural, para o cálculo do custo de execução de uma tarefa foi definido o fator de heterogeneidade, denotado por $fh(p_i)$, que representa a quantidade de dados que o processador p_i pode processar em uma determinada unidade de tempo. Por outro lado, enquanto uma tarefa não for escalonada o custo de execução dessa tarefa não pode ser calculado utilizando o $fh(p_i)$, pois p_i não é conhecido. Por isso, foi definido o fator de heterogeneidade médio (fhm), que representa a média dos fh para todos os processadores disponíveis.

O custo de execução de uma tarefa escalonada v_i é dado por $\varepsilon(v_i) \times fh(p_i)$, onde p_i é o processador em que v_i está escalonada e o custo de execução de uma tarefa v_i, quando v_i não está escalonada, é dado por $\varepsilon(v_i) \times fhm$. Agora serão descritas as prioridades utilizadas e como é calculado o valor dessas prioridades para as tarefas pertencentes a um GAD:

- ***b-level*** **(*bottom level*) ou nível** – O *nível* de um nó *n* é o tamanho do maior caminho desde o nó *n*, incluindo o custo de execução de *n*, até um nó de saída. Sabendo-se que, o cálculo deste caminho consiste na soma de todos

os *custos de computação* das tarefas v_i, pertencentes a esse caminho, multiplicado pelo fator de heterogeneidade médio das máquinas ($\varepsilon(v_i) \times fhm$) e todas as comunicações $\omega(v_i, v_{i+1})$ das arestas (quantidade de dados a ser enviado) multiplicado pela latência, entre duas tarefas v_i e v_{i+1} adjacentes, pertencentes a esse mesmo caminho.

- ***t-level*** **(*top level*) ou conível** – O *conível* de um nó *n* é o custo do caminho mais longo desde um nó de entrada até *n* , neste caso não é contado o custo de execução do próprio nó *n*. Sendo que, o custo do caminho consiste na soma de todos os *custos de computação* das tarefas v_i multiplicado pelo fator de heterogeneidade médio das máquinas ($\varepsilon(v_i) \times fhm$), e somando todos os *custos de comunicação* ω (v_i, v_{i+1}) das arestas (quantidade de dados a ser enviado multiplicada pela latência) de duas tarefa v_i e v_{i+1} adjacentes, ao longo de todo o caminho. Desta forma, o *t-level* de um nó coincide com o *tempo de início* mais cedo em que este nó pode ser escalonado sem que sejam violadas quaisquer das restrições de precedência.
- ***Critical Path (CP) ou Caminho Crítico*** – O caminho crítico de um *GAD* é o mais longo caminho no *GAD* desde o nó de entrada até o nó de saída. Entende-se aqui por caminho mais longo, o caminho que possui a maior soma dos *custos de computação* das tarefas v_i pertencentes a este caminho, multiplicado pelo fator de heterogeneidade médio das máquinas ($\varepsilon(v_i) \times fhm$) e somando também as comunicações desde o nó de entrada até o nó de saída. É possível que um GAD tenha mais de um caminho crítico.
- ***ALAP (As Late As Possible) start-time*** – O *ALAP* de um nó é a medida de quanto tempo se pode atrasar o início da execução de uma tarefa, sem que se aumente o tempo total do escalonamento. Podemos obter esta medida mediante a realização da subtração do valor do caminho crítico do *GAD* pelo atributo *nível* da tarefa, ou seja, $ALAP(v_i) = CP(G) - nível(v_i)$. Note que as tarefas do caminho crítico possuem o valor da prioridade *ALAP* igual ao valor da prioridade *conível*.
- ***b-level*** **dinâmico ou nível dinâmico:** Esta prioridade é obtida basicamente da mesma forma que a prioridade *nível*. A principal diferença é que esta prioridade tenta ser mais precisa, pois é recalculado o seu valor após o

escalonamento de cada tarefa do grafo. O caminho mais longo é calculado usando a maior soma dos *custos de computação* das tarefas v_i (incluindo a própria tarefa v_i), pertencentes a este caminho, multiplicado pelo fator de heterogeneidade do processador ($\varepsilon(v_i) \times fh(p_i)$), sendo p_i o processador em que v_i está escalonada. Caso v_i não esteja escalonada, esse valor será ($\varepsilon(v_i) \times fhm$). Para o cálculo do caminho mais longo devem ser adicionados também os custos de comunicação das tarefas, caso elas não estejam escalonadas nos mesmos processadores que seus predecessores imediatos, em caso contrário, esse valor é considerado zero.

- ***t-level* dinâmico ou conível dinâmico:** Esta prioridade é obtida basicamente da mesma forma que a prioridade conível. A principal diferença é o valor do conível das tarefas é recalculado após o escalonamento de cada tarefa do grafo. O caminho mais longo de uma tarefa n é obtido escolhendo o caminho que possui a maior soma dos *custos de computação* das tarefas v_i antecessoras de n pertencentes a este caminho, multiplicado pelo fator de heterogeneidade do processador ($\varepsilon(v_i) \times fh(p_i)$), sendo p_i o processador em que n está escalonada, caso v_i não esteja escalonada, esse valor será ($\varepsilon(v_i) \times fhm$). Para o cálculo do caminho mais longo devem ser adicionados também os custos de comunicação das tarefas v_i, caso elas não estejam escalonadas nos mesmos processadores que seus predecessores imediatos, senão zerasse esse valor. É importante lembrar que para a maioria dos algoritmos *List Scheduling* o *t-level* não precisa ser recalculado, pois temos um escalonamento *top dowm*.
- ***Critical Path (CP) dinâmico ou Caminho Crítico dinâmico*** – O caminho crítico dinâmico é obtido de forma bem parecida que a mesma prioridade estática, mas essa prioridade é mais precisa que o Caminho Crítico estático por ela recalcular o seu valor após o escalonamento de cada tarefa. Além disto, ela considera em qual processador a tarefa está escalonada, o que permite reduzir custos de comunicação e calcular o custo de computação da tarefa de forma mais exata.
- ***ALAP (As Late As Possible) dinâmico*** – O ALAP dinâmico consiste em recalcular a prioridade ALAP após o escalonamento de cada tarefa. Para o calculo do ALAP dinâmico é considerado em qual processador a tarefa está escalonada, de forma a reduzir custos de comunicação e calcular de

forma mais exata o custo de computação da tarefa.

Existem vários algoritmos baseados na heurística *List Scheduling* que utilizam uma dessas prioridades como forma de selecionar a tarefa a ser escalonada, por exemplo, o algoritmo HEFT que utiliza o *nível* das tarefas para determinar qual será a ordem em que as tarefas serão escalonadas. Outros algoritmos como MCP[Gajs90] preferem atribuir a mais alta prioridade ao nó que possui o maior *ALAP*, tendendo nesse caso a escalonar os nós tão cedo quanto possíveis. Contudo, é possível que um algoritmo utilize mais de um nível de prioridade, de forma a resolver os casos de desempate de um determinado nível utilizando uma segunda prioridade.

2.7.3 Escolha do processador

Uma importante fase no escalonamento de tarefas é a escolha do processador onde cada tarefa será escalonada. Nos casos em que o algoritmo de escalonamento encontra situações em que vários processadores oferecem os mesmos tempos de inicio para a tarefa ou situações em que o processador possui vários espaços ociosos é possível que seja tomado algumas decisões dependendo dos dados de entrada fornecidos pelo usuário. Considerando a importância da escolha do processador, foram implementadas algumas funcionalidades para verificar a relevâncias de cada uma delas para minimizar o *makespan*. Essas funcionalidades estão descritas a seguir:

- ***Economia na utilização de processadores*** - pode ocorrer um empate entre o tempo oferecido por um processador já utilizado e um novo processador (ainda não utilizado). A escolha de um novo processador muitas vezes permite que tarefas sucessoras sejam escalonadas neste mesmo processador reduzindo em muitos casos o *makespan*. Por outro lado com esta opção selecionada, o número de processadores utilizados tende a aumentar;
- ***Inserção em espaços ociosos*** - durante o processo de escalonamento é possível que espaços de tempo ociosos nos processadores sejam gerados. Neste caso, escalonar a próxima tarefa livre em um espaço ocioso pode vir a produzir melhores resultados, principalmente quando os processadores são heterogêneos [Topc02];

- ***Redução do número de processadores utilizados*** - após a realização do escalonamento é possível que um conjunto de tarefas escalonadas possa ser deslocado para um outro processador que possui um espaço de tempo ocioso suficiente para executá-las, sem aumentar o *makespan* do escalonamento gerado.

2.8 Outras heurísticas para comparação

Devido à dificuldade de obter provas de optimalidade, algoritmos de escalonamento são, na maioria das vezes, comparados experimentalmente. Com o objetivo de escolher quais algoritmos seriam implementados para testes, foram analisados entre os algoritmos existentes aqueles que possuíam bons resultados e os outros com destaque na literatura. O algoritmo DCP [Kwok96] foi escolhido por utilizar uma heurística de escolha de tarefa um pouco diferente quando comparado com um *List Scheduling* tradicional e principalmente pelos bons resultados obtidos por esse algoritmo. A comparação entre algoritmos que realizam ações que algumas vezes são totalmente diferentes é uma importante forma de investigar novos rumos para o escalonador, e assim incorporar novas idéias ao algoritmo. O ETF [Hwan89] foi implementado principalmente pelo seu destaque na literatura, fornecendo um vasto material de referência. O HEFT é bastante parecido com algoritmo ETF, no entanto o HEFT é considerado um dos melhores algoritmos para ambientes heterogêneos.

2.8.1 O algoritmo ETF

O algoritmo ETF (Earliest Time First) [Hwan89] é um algoritmo do tipo *List Scheduling* projetado para um número finito de processadores homogêneos. O ETF determina a cada iteração o tempo de início mais cedo de todas as tarefas livres nos processadores ociosos no momento corrente. A tarefa selecionada é a que tem o menor tempo de início e é alocada no respectivo processador. Se a tarefa escolhida tem possibilidade de iniciar mais cedo ainda em um processador que não está ocioso no momento, seu escalonamento só é efetivamente realizado em iterações posteriores.

Quando o tempo mais cedo de duas ou mais tarefas forem iguais, o algoritmo escolhe a tarefa com o maior *nível estático*. Portanto, uma tarefa com um maior *nível estático* não necessariamente consegue ser escalonada primeiro, porque o algoritmo considera mais prioritárias as tarefas com o menor tempo de inicio.

2.8.2 O algoritmo DCP

O algoritmo DCP (*Dynamic Critical Path*) [Kwok96] é projetado baseando-se em um atributo que define a mobilidade da tarefa. O algoritmo DCP usa a estratégia *look-ahead* para encontrar o melhor cluster de tarefas para um dado vértice. Por isso, para computar o valor de $T_s(n_i)$ (tempo de inicio de uma tarefa n_i) sobre um processador, o algoritmo DCP também computa o valor de $T_s(n_c)$ (tempo de inicio de n_c), sobre o mesmo processador. Onde n_c é o filho de n_i que tem o maior custo de computação e é também chamado de filho crítico de n_i. O algoritmo DCP escalona uma tarefa n_i no processador que fornecer o valor mínimo da soma desses dois atributos. Esta estratégia de *look-ahead* pode evitar que a tarefa seja escalonada em um cluster que não tem espaço para acomodar o sucessor crítico, impondo uma comunicação pesada entre eles. O algoritmo DCP examina todos os clusters existentes para decidir em qual deles irá escalonar um nó, enquanto outros algoritmos somente procuram um cluster até encontra um que tenha espaço ocioso para armazenar a tarefa da vez. Neste trabalho, tanto o DCP quanto o ETF foram adaptados para um conjunto limitado de recursos heterogêneos.

2.8.3 O algoritmo HEFT

O algoritmo *Heterogeneous Earliest Finish Time* (HEFT) [Topc02] também é uma heurística do tipo *List Scheduling* e considera um número limitado de recursos heterogêneos. O algoritmo HEFT primeiramente ordena todas as tarefas de acordo com a prioridade nível em uma lista. Posteriormente, seguindo a ordem determinada nesta lista, o escalonador vai associando a cada tarefa um processador, de forma que seja minimizado o tempo de fim da tarefa escolhida. HEFT, no entanto, realiza o procedimento de inserção da tarefa em espaços de tempo ociosos nos processadores.

2.8.4 Algoritmos *List Scheduling* para o modelo HLogP

Na literatura existem muito poucos algoritmos que tratam os parâmetros de comunicação definidos nos modelos tipo *LogP* [Cull93]. A maioria adota a estratégia de aglomeração com replicação de tarefas [Boer03a]. O único algoritmo tipo *List Scheduling* é uma versão do algoritmo ETF modificado para o modelo *LogP* proposto por Kalinowski *et al*. [Kali00] que também foi implementado na ferramenta. Utilizando as mesmas técnicas, uma versão *HLogP* do HEFT também foi implementada. Técnicas alternativas para tratar as sobrecargas de comunicação em algoritmos *List Scheduling* estão sendo investigadas [Card04].

2.9 Resumo

Este capítulo apresentou o problema de escalonamento de tarefas, os modelos de computação paralela utilizados, heurísticas de escalonamento de tarefas e o problema do escalonamento em grades. Também foi discutido neste capítulo sobre a necessidade de um escalonador estático e a utilização do modelo HLogP para escalonamentos usando grades. No próximo capítulo será apresentada a ferramenta implementada e será dada uma visão geral do funcionamento dessa ferramenta.

3 *OVERVIEW* DA ESTRUTURA DA FERRAMENTA

Esta seção inicialmente apresenta uma descrição geral do *framework Easygrid*, com o objetivo de localizar a ferramenta dentro do projeto *EasyGrid*. Em seguida são descritos os passos de desenvolvimento de um algoritmo no contexto da ferramenta apresentada. Um fluxograma ilustra as funcionalidades disponíveis durante o processo de projeto, avaliação e validação de novas heurísticas. Ao fim do capítulo detalhes de implementação e manutenção da ferramenta são comentados.

3.1 O *Framework EasyGrid*

O projeto *EasyGrid* [Rebe03, Boer04] tem como meta desenvolver um *framework* para a transformação automática dos programas paralelos baseados na biblioteca MPI em aplicações *system-aware*, ou seja, aplicações cientes do comportamento do ambiente onde estão sendo executadas e capazes de reagir para tirar proveito de mudanças no estado deste ambiente. A metodologia *EasyGrid* visa permitir aos programadores se concentrar em como explorar o paralelismo e deixar a cargo do *framework EasyGrid* gerar uma versão da aplicação capaz de utilizar da melhor forma possível os recursos de um ambiente grade disponíveis ao usuário. O *framework* será usado para validar uma abordagem de *middleware* direcionada para a aplicação e, além disso, estudar o problema de escalonamento estático e dinâmico [Boer03], a integração de tolerância à falhas e as estratégias de escalonamento para aplicações *system-aware* em grades computacionais.

A Figura 3.1 mostra um esboço do *framework* (proposto em [Boer04]) para gerar uma aplicação *system-aware* a partir do código fonte MPI do usuário. Na figura, as funções são representadas por retângulos com cantos arredondados e os arquivos por retângulos regulares. Em adição ao programa MPI do usuário, também é necessário uma

lista de recursos do ambiente grade (Grid Access File) aos quais o usuário tem acesso. O sistema modelador (*System Modeller*) calibra o modelo arquitetural *HLogP*, e também pode obter informações de um serviço de diretórios, tal como o MDS do Globus ou NWS [Wols99], ou medir diretamente cada recurso do usuário que está *online*. Alguns exemplos de informações necessárias incluem: velocidade do processador, carga média e atual do processador e latências de comunicação.

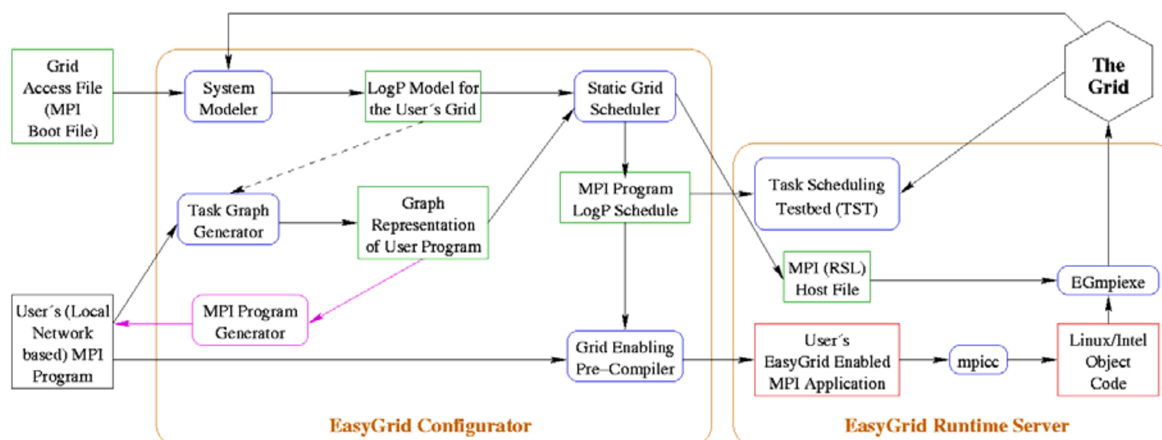

Figura 3.1 O *framework EasyGrid*

Com base nas características dos recursos da grade disponíveis e da aplicação, um mapeamento inicial ou escalonamento é produzido pelo escalonador estático da grade (*Static Grid Scheduler*) para guiar a execução. O arquivo *MPI Host File*, equivalente ao arquivo Globus RSL é gerado pelo escalonador, identificando os recursos para qual cada processo MPI deve ser alocado. O gerador de grafo de tarefas (*Task Graph Generator*) pode ser usado para criar uma representação GAD do programa do usuário [Sinn00], enquanto o gerador de programas MPI (*MPI Program Generator*) cria programas MPI sintéticos a partir de representações GAD. Isso facilita a investigação dos efeitos da granularidade e estrutura do programa (usando GADs pertencentes a *benchmarks* de escalonamento) na eficiência do escalonamento gerado e na sua execução no ambiente grade.

O pré-compilador habilitador da grade (*Grid Enabling Pre-Compiler*) gera uma aplicação MPI *system-aware* usando as informações de escalonamento para reestruturar o programa MPI original do usuário (isso envolve replicação de processos MPI e reordenação das comunicações, se considerado vantajoso) e incorporando o *middleware* apropriado específico da aplicação (o sistema de gerenciamento da aplicação – SGA)

[Frei03]. Em ambos os casos, nenhuma linha de código fonte do usuário é alterada. Enquanto a aplicação do usuário é transformada automaticamente, por motivos de depuração existe uma clara correspondência entre o código escrito pelo usuário e o código executado na grade. Assim, as funções do SGA são embutidas como processos MPI ocultos e integrados dentro da aplicação do usuário por uma camada que aumenta a capacidade das funções MPI padrão [Frei03]. Por razões de portabilidade, nenhuma modificação é feita na biblioteca MPI. O script EGmpiexe cuida de aspectos de autenticação, assegura que cada recurso tem o executável específico correto para o sistema operacional, e inicia a execução da aplicação.

3.2 O Projeto, a Avaliação e a Validação de Algoritmos de Escalonamento

A necessidade de uma ferramenta para facilitar o processo de desenvolvimento e aprimoramento de algoritmos de escalonamento de aplicações para sistemas heterogêneos e distribuídos foi notada inicialmente pelos próprios autores quando estes iniciaram a implementação de um algoritmo do tipo *List Scheduling* para realizar a função de escalonador estático do *framework EasyGrid.* Foi notado que um grande tempo foi gasto com atividades de teste e validação do algoritmo. Para cada combinação de entrada (grafo, arquitetura, prioridades, etc...) era necessário analisar o escalonamento obtido tanto em termos de validade quanto de eficiência. Também foi notado que estes problemas atingiam outros usuários do laboratório de Pós-Graduação da Universidade Federal Fluminense (UFF), dentre estes alunos (mestrandos e doutorandos) e professores que atuam nesta área. Assim surgiu a idéia de uma ferramenta gráfica que facilitasse estas atividades, tornando-as mais rápidas, agradáveis e intuitivas para os usuários.

O processo de desenvolvimento de um algoritmo tipicamente consiste de três fases: o projeto e implementação de um ou mais algoritmos de escalonamento; a avaliação das heurísticas através da comparação experimental dos resultados gerados tanto por novos algoritmos sendo construídos, como pelos já existentes, considerando um conjunto de *GAD*s e modelos arquiteturais escolhidos pelo próprio usuário da ferramenta; a validação por simulação (por exemplo, *SimGrid 2.0* [Casa01] foi proposto

para avaliar algoritmos de escalonamento dinâmicos sem ambientes grades) e/ou por execução num ambiente real como, por exemplo, a Grade Computacional GridRio.

3.3 Fluxograma de Funcionamento

A ferramenta aqui apresentada busca oferecer um ambiente amigável e facilitador para que o usuário possa passar pelas fases de projeto, avaliação e validação de uma forma intuitiva, tendo ao seu dispor uma interface gráfica com uma série de funcionalidades que a tornam uma ferramenta de ensino e aprendizado de algoritmos de escalonamento. Na fase de validação por execução na Grade Computacional GridRio a ferramenta atua como um portal para o *framework EasyGrid*.

Dois usuários imaginários, que serão chamados de João e Maria, ilustrarão a utilização da ferramenta por dois caminhos possíveis (os quais podem ser visualizados nos fluxogramas das Figura 3.2, 3.3 e 3.4). João é aluno do mestrado e deseja realizar testes com seu novo algoritmo de escalonamento. Já Maria, que é professora de métodos numéricos, está interessada em utilizar o poder computacional da grade para executar um novo método que está desenvolvendo para resolução de equações diferenciais.

João solicitou a inclusão de seu algoritmo na ferramenta. Para isso ele seguiu os padrões de arquivos (ver Apêndice A) e forneceu o executável de seu algoritmo. Assim, no dia seguinte ele iniciou seus testes. Inicialmente João definiu que iria escalonar aplicações do tipo GAD. Em seguida João utilizou a função de visualização de GADs, para ter uma idéia exata do formato do grafo e das relações de precedência entre as tarefas. Definido o GAD a ser escalonado, João avança para a página da ferramenta onde é definido o modelo arquitetural. Como está somente realizando seus primeiros testes, João opta por utilizar uma arquitetura teórica, ou seja, uma arquitetura que é definida através de um arquivo e que não corresponde necessariamente a uma grade real. João poderia até criar seu próprio arquivo de arquitetura seguindo o padrão para arquivos de arquitetura. Após selecionar seu arquivo de arquitetura, ele avança para a tela onde deve definir o algoritmo a ser utilizado para escalonamento. João seleciona seu algoritmo e define as prioridades que serão utilizadas.

Agora João está pronto para executar o escalonamento. Após a execução ele visualiza o escalonamento gerado por seu algoritmo através de uma tabela “processador *versus* tempo”. Para ter uma visão mais completa do escalonamento, João utiliza a

função que gera um Diagrama de Gantt de seu escalonamento. Ele percebe alguns detalhes de seu escalonamento que podem estar impedindo que melhores resultados sejam obtidos. Para confirmar suas observações ele decide comparar sua heurística com outras heurísticas disponíveis na ferramenta. Para isso, ele retorna à tela de escolha de algoritmos e seleciona as heurísticas que deseja comparar. Em seguida são definidos os grafos que serão utilizados na comparação. Os resultados da comparação são apresentados nos seguintes formatos: *makespan* e número de processadores para escalonamentos gerados para cada grafo; quantidade de melhores *makespans* entre cada par de heurísticas; quantas vezes cada heurística atingiu o melhor *makespan* do conjunto de algoritmos sendo avaliados. Utilizando estas informações, João percebe que seu algoritmo ainda precisa ser bastante refinado, já que o máximo que conseguiu foi empatar com as outras heurísticas em alguns grafos do tipo diamante. João ainda tem muito trabalho pela frente, mas antes de retornar ao seu estudo ele decide imprimir as comparações para no futuro verificar os impactos das modificações que pretende realizar em seu algoritmo.

Enquanto isso, Maria deseja apenas executar seu método para resolução de equações diferenciais. Maria já possui o arquivo fonte MPI correspondente à sua aplicação. Assim, inicialmente ela seleciona a opção “Aplicação MPI”. Maria decide utilizar o escalonador MPI e assim avança diretamente para o Portal que inicia a execução das aplicações. Maria sabe que para executar sua aplicação na grade ela deve criar um *proxy,* que dará acesso aos recursos da grade. Através da ferramenta ela cria um *proxy*, identificando-se através de *login* e senha e definindo o tempo de duração do *proxy.* Em seguida realiza a compilação de sua aplicação e decide utilizar o *middleware EasyGrid* que irá monitorar a execução da aplicação. Depois ela escolhe utilizar o escalonador MPI e inicia a execução de sua aplicação. Como utilizou o *middleware EasyGrid*, ao final da execução Maria pode obter informações sobre como sua aplicação foi executada na grade.

Após alguns dias de trabalho e de mais alguns testes utilizando a ferramenta, João decide que é hora de testar seu escalonador executando uma aplicação GAD na grade. Para isso ele define que deseja executar uma aplicação GAD e em seguida escolhe o grafo que será utilizado. João decide utilizar o modelador disponível na ferramenta para poder definir a configuração da arquitetura que será utilizada por seu escalonador. Assim, ele avança para a página que permite a modelagem. Inicialmente é realizada a leitura do arquivo de configuração da grade, e as máquinas da grade são

apresentadas. João agora define a versão MPI que deseja utilizar (no caso MPI CH-G2) e a ferramenta realiza uma filtragem das máquinas que suportam tal versão. Ele agora define as máquinas da lista apresentada que ele deseja utilizar no seu teste. Após esta fase, João utiliza a função que verifica o *status* das máquinas por ele escolhidas. De posse dos resultados ele verifica que uma das máquinas que desejava utilizar está com uma grande carga de processos e decide removê-la da lista. Agora João possui a lista de máquinas que deseja utilizar no escalonamento e precisa criar um *proxy* para iniciar a modelagem. Após criar o *proxy* e executar o modelador, João pode realizar o escalonamento do GAD utilizando seu algoritmo e o modelo arquitetural recém criado.

Agora já com o escalonamento estático realizado, João avança para o Portal da ferramenta, onde uma aplicação MPI sintética equivalente ao GAD escolhido é gerada e compilada com o *middleware EasyGrid*. João define que será utilizado o escalonamento *EasyGrid*, no caso o escalonamento gerado por seu escalonador. Como já havia criado um *proxy* para realizar a modelagem, João está apto a executar sua aplicação. Após a execução João visualiza as informações da execução e compara a execução real com o resultado simulado de seu escalonador.

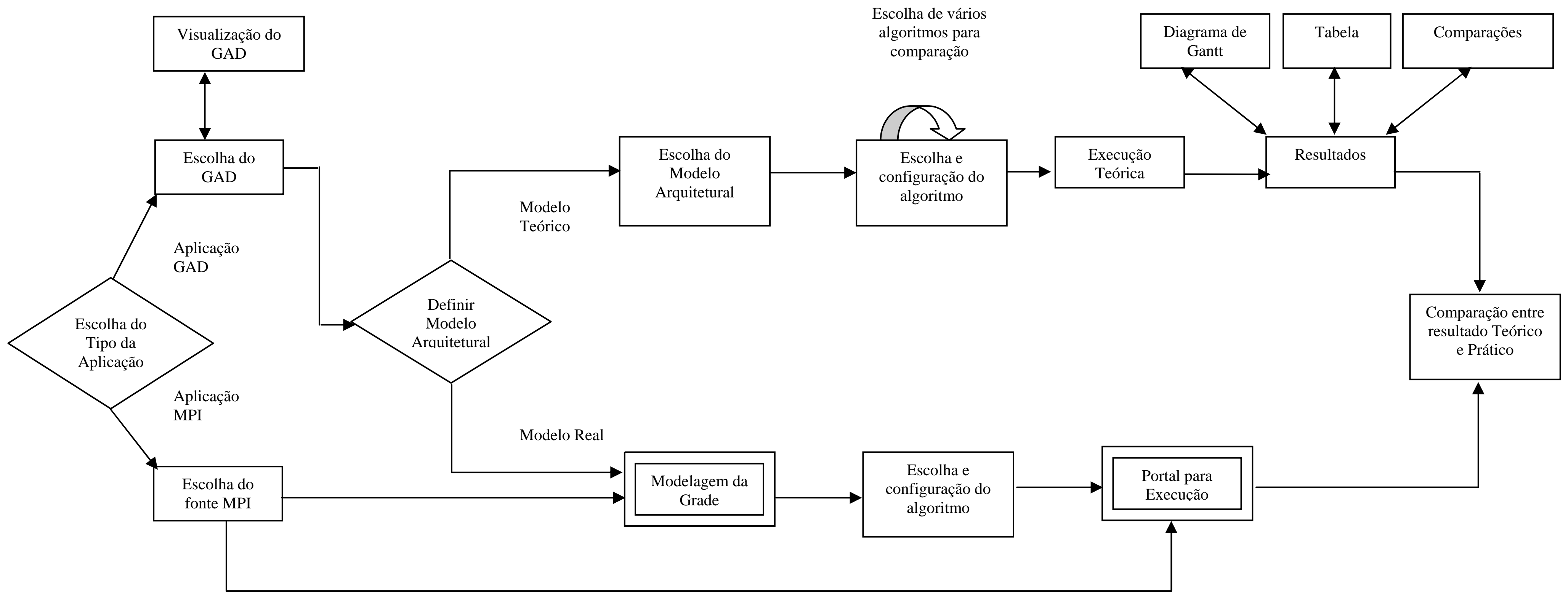

Figura 3.2 Fluxograma Geral

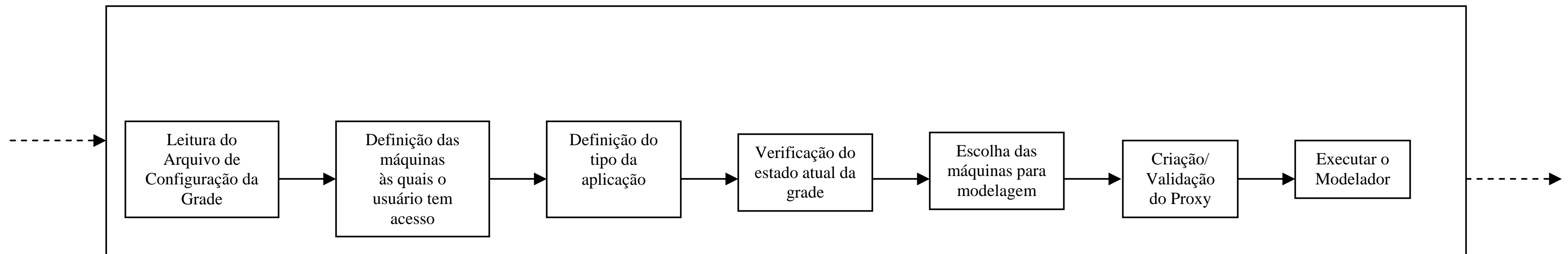

Figura 3.3 Fluxograma da modelagem da grade

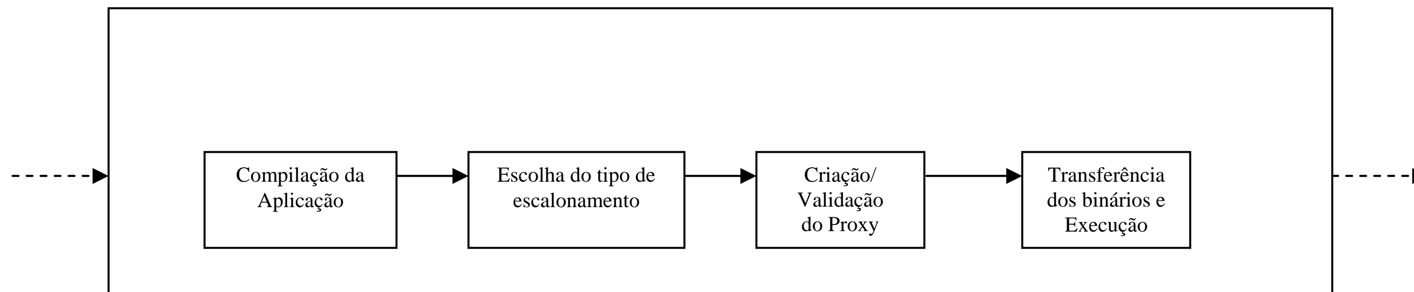

Figura 3.4 Fluxograma do portal para Execução

3.4 Implementação, Atualizações e Manutenção

O desenvolvimento desta ferramenta foi realizado de acordo com os conceitos de programação modular e programação orientada a objetos. Os algoritmos de escalonamento são totalmente independentes da interface gráfica com o usuário (GUI – *Graphical User Interface*). Eles são compilados separadamente e a GUI realiza uma chamada ao sistema, passando para o executável correspondente ao algoritmo selecionado os parâmetros necessários para sua execução. Os valores destes parâmetros são definidos pelo usuário através da GUI. A interface entre os algoritmos e a GUI é realizada através de arquivos com formato pré-definido (os formatos dos arquivos são apresentados no Apêndice A). Em geral, como entrada há um arquivo representando a aplicação (GAD), um arquivo de custos associado ao GAD e um arquivo representando a arquitetura alvo. Como saída os algoritmos produzem um arquivo que representa o escalonamento por eles gerado. Assim, detalhes internos dos algoritmos podem ser modificados sem que seja necessário interferir na GUI, bastando que o formato dos arquivos de entrada e saída seja respeitado.

Esta abordagem no desenvolvimento facilita a manutenção da GUI e torna a integração de novos algoritmos à ferramenta bastante simples. Basta que o executável correspondente ao novo algoritmo seja copiado para o diretório onde se localizam os algoritmos da ferramenta. Em seguida um dos desenvolvedores responsáveis pela ferramenta deve adicionar o novo algoritmo a GUI, tornando-o disponível para execução.

Além da incorporação de novos algoritmos, a adição de novas funcionalidades à GUI também é realizada de forma simples. A GUI foi dividida em páginas (como pode ser visto na Seção 4), cada uma responsável por uma parte específica da ferramenta. Para adicionar uma nova funcionalidade, basta que uma nova página seja adicionada e configurada pelo desenvolvedor. Um exemplo é a parte da ferramenta responsável pelo escalonador dinâmico, que foi incorporada recentemente à ferramenta, com a adição de duas novas páginas [Nilm04].

Alguns algoritmos de escalonamento propostos na literatura foram desenvolvidos durante este trabalho e disponibilizados na GUI. No desenvolvimento destes algoritmos, a idéia de classes foi fortemente utilizada. Cada estrutura de dados (como listas, filas, grafos) foi definida como uma classe. Depois de testadas e validadas, estas classes passam a ser

utilizadas pelos algoritmos, permitindo uma grande reutilização de código. Além da economia de tempo, esta abordagem isola os erros, uma vez que as estruturas de dados já foram previamente testadas. Estas idéias também se aplicam ao desenvolvimento da GUI.

Para o desenvolvimento desta ferramenta foi utilizado o Borland Kylix C++ Builder 3.0 [BorlWeb]. Esta ferramenta foi escolhida por oferecer um ambiente de desenvolvimento completo para a linguagem C/C++ em sistemas operacionais Linux, facilidades para o desenvolvimento de interfaces gráficas e por ser uma ferramenta com um bom reconhecimento no mercado de produção de softwares.

3.5 Resumo

Este capítulo apresentou uma visão geral da ferramenta proposta. Os passos de desenvolvimento de um algoritmo foram descritos no contexto da ferramenta. Um fluxograma ilustrou a utilização da ferramenta por dois usuários típicos, dando uma idéia geral de seu funcionamento. Ao final do capítulo foram abordados os detalhes de implementação.

4 A FERRAMENTA *TASK SCHEDULING TESTBED*

A ferramenta *Task Scheduling Testbed* foi implementada com o objetivo de facilitar o processo de desenvolvimento e aprimoramento de algoritmos de escalonamento de aplicações para sistemas heterogêneos e distribuídos. Esse processo tipicamente consiste de três fases: o projeto e implementação de um ou mais algoritmos de escalonamento; a avaliação das heurísticas através da comparação experimental dos resultados gerados tanto por novos algoritmos sendo construídos, como pelos já existentes, considerando um conjunto de *GAD*s e modelos arquiteturais escolhidos pelo próprio usuário da ferramenta; a validação por simulação utilizando *SimGrid 2.0* [Casa01] e/ou a validação por execução na Grade Computacional GridRio. Neste último caso, a ferramenta atua como um portal para o *framework EasyGrid*, permitindo a modelagem, compilação e execução de aplicações em uma grade real, no caso a grade computacional GridRio. Neste capítulo serão apresentadas as funcionalidades oferecidas pela ferramenta em cada um dos passos de desenvolvimento de um algoritmo de escalonamento.

4.1. Projeto de um algoritmo

Quando um usuário está na fase de projeto de um algoritmo, os primeiros testes são geralmente realizados com modelos (de aplicação e arquitetural) sintéticos que descrevem o comportamento de uma grade teórica. Este modo de utilização também é interessante para comparação entre diversos algoritmos (ou combinações de um mesmo algoritmo).

Na versão atual da ferramenta, o usuário seleciona um algoritmo da lista de heurísticas disponíveis (*List Scheduling* configurável, DCP, ETF e HEFT como foi descrito no Capítulo 2, e outros), depois de ter escolhido a aplicação de entrada (representada por

um *GAD*), e definido o conjunto das características principais da arquitetura alvo (detalhes da aplicação e da arquitetura são lidos de arquivos texto que podem ser escritos pelo usuário, o formato dos arquivos é dado no Apêndice A). Junto com a ferramenta, estão disponíveis todos os GADs dos *benchmarks* de escalonamento de [Kwok99b] e de [Boer02a] (GADs UET-UCT (*unit execution time – unit communication time*)), junto com outros grafos da literatura. A interface amigável da ferramenta facilita o entendimento do usuário na seleção de todas essas características e prioridades necessárias. Para realizar os testes o usuário escolhe o GAD desejado e o arquivo de custos correspondente. A Figura 4.1 mostra o módulo responsável por esta etapa.

Figura 4.1 Especificação da Aplicação GAD

O usuário pode visualizar o grafo escolhido selecionando o arquivo que contém uma descrição textual do grafo. Utilizando *graphviz* (um software livre de *AT&T Research*

Labs) [GraphVizWeb], uma figura do mesmo pode ser gerada para visualização na tela ou impressão. Um exemplo de GAD pode ser visualizado na Figura 4.2, onde: as tarefas da aplicação são representadas pelos vértices rotulados pela sua identificação e seu peso de execução entre parênteses; e as relações de precedência, representadas por arcos, são rotuladas pelos pesos de comunicação. A visualização do grafo é de grande auxílio para que o usuário verifique se seu algoritmo está produzindo um escalonamento válido e que corresponde ao comportamento desejado.

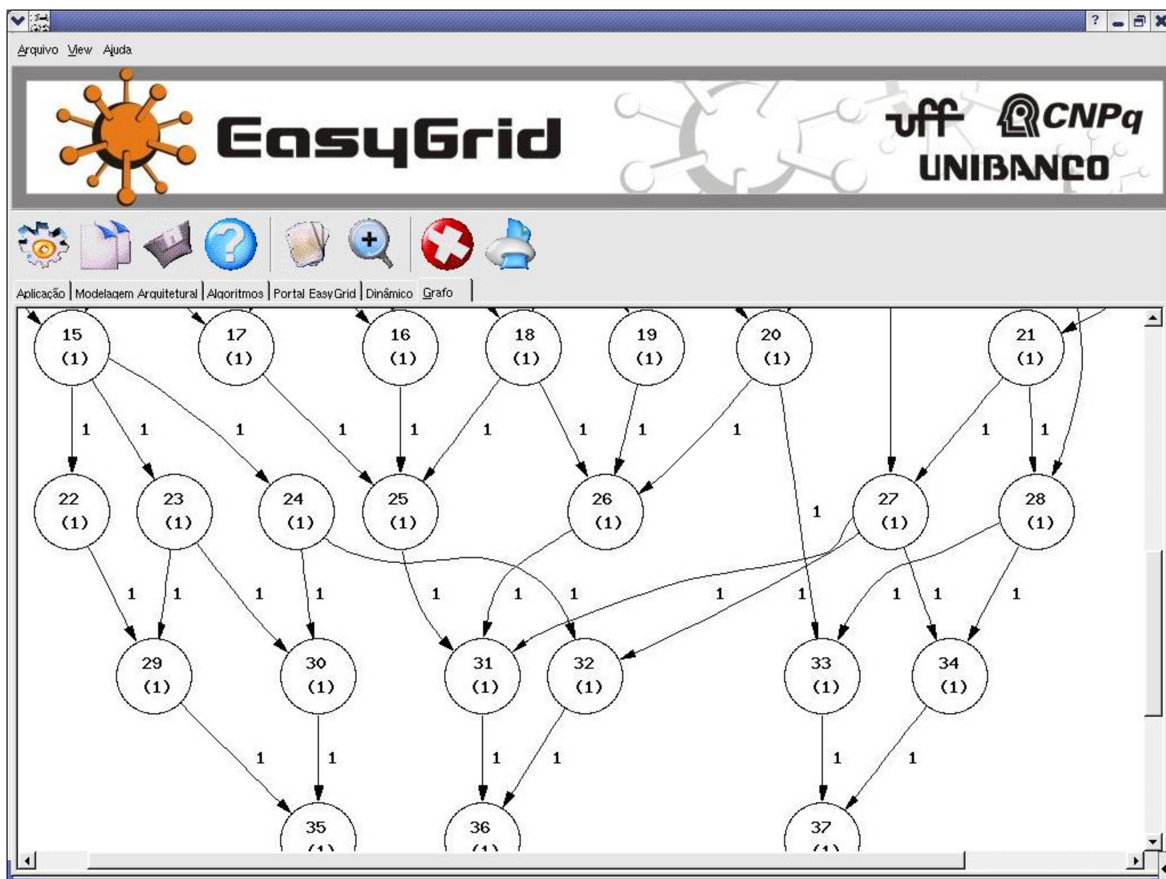

Figura 4.2 Visualização do grafo da aplicação (GAD):

A arquitetura que será considerada pelo algoritmo de escalonamento é lida do arquivo especificado pelo usuário. A ferramenta apresenta um *preview* do arquivo selecionado bem como informações relativas à arquitetura que o arquivo representa. A Figura 4.3 apresenta esta parte da ferramenta.

Figura 4.3 Especificação da Arquitetura Teórica

A fase seguinte é a de escolha do algoritmo de escalonamento. No caso do algoritmo *List Scheduling,* o usuário pode configurá-lo tanto no que se refere à escolha de tarefas (as prioridades descritas na Seção 2.7.2) quanto à escolha do processador (possibilidades descritas na seção 2.7.3). Cada algoritmo possui seu conjunto de configurações, que podem ser realizadas através da interface.

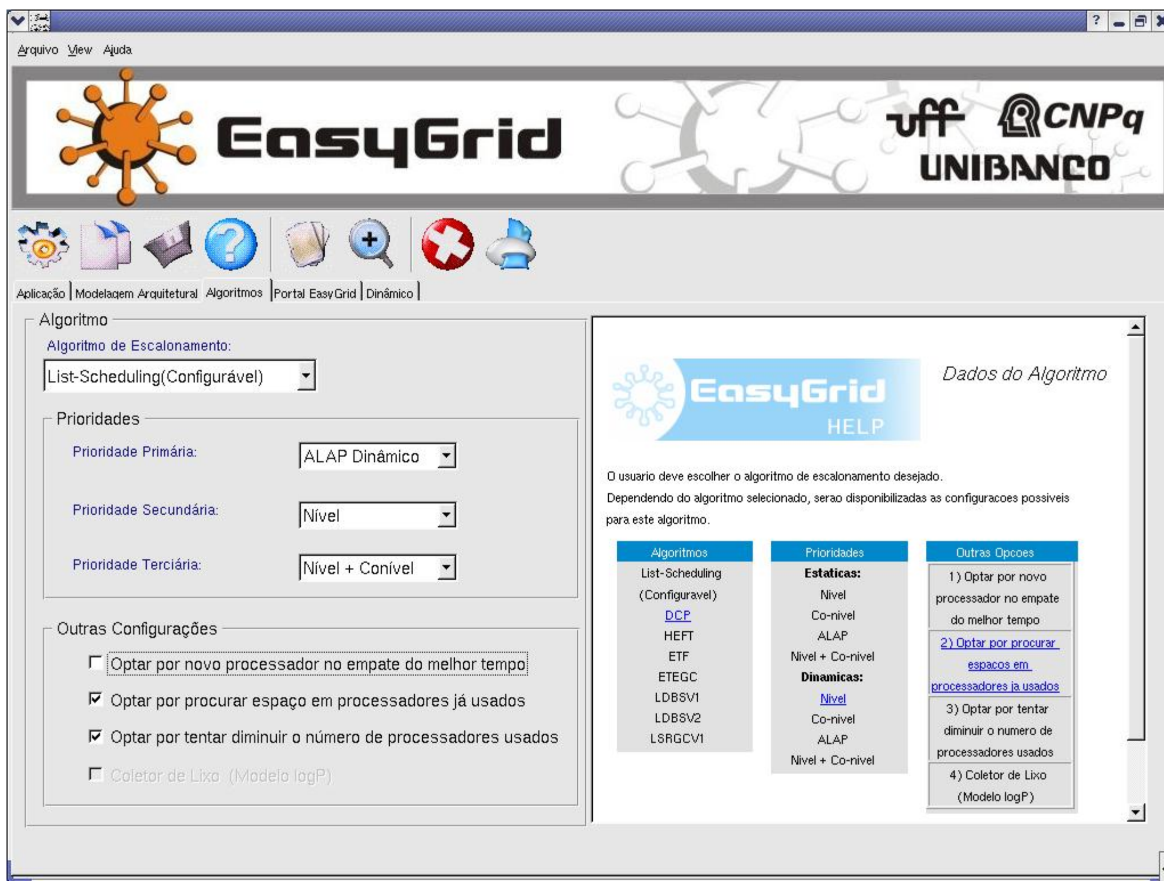

Figura 4.4 Especificação do algoritmo e suas configurações

Na Figura 4.4 pode ser visualizado o módulo da interface que possibilita a especificação da estratégia a ser utilizada, bem como a configuração desta estratégia. No caso da figura o algoritmo escolhido foi *List Scheduling,* que pode ser configurado em relação às prioridades na escolha das tarefas (primária, secundária e terciária) e às configurações quanto à escolha do processador (inserção, economia de processadores, etc). A ferramenta assim, a partir da especificação da política de escalonamento a ser utilizada e seus dados de entrada, apresenta os resultados necessários à avaliação através dos módulos de interface.

O resultado do escalonamento é apresentado em uma tabela *processador versus tempo*, onde o tempo é incrementado em unidades ou por um diagrama de Gantt (que facilita a análise de escalonamentos com *makespan* de alto valor). No exemplo de um diagrama de Gantt na Figura 4.5, cada linha corresponde a um processador utilizado, e cada bloco, representando uma tarefa, tem comprimento proporcional ao seu custo de execução. O usuário pode salvar o escalonamento gerado ou imprimí-lo, para futuro estudo.

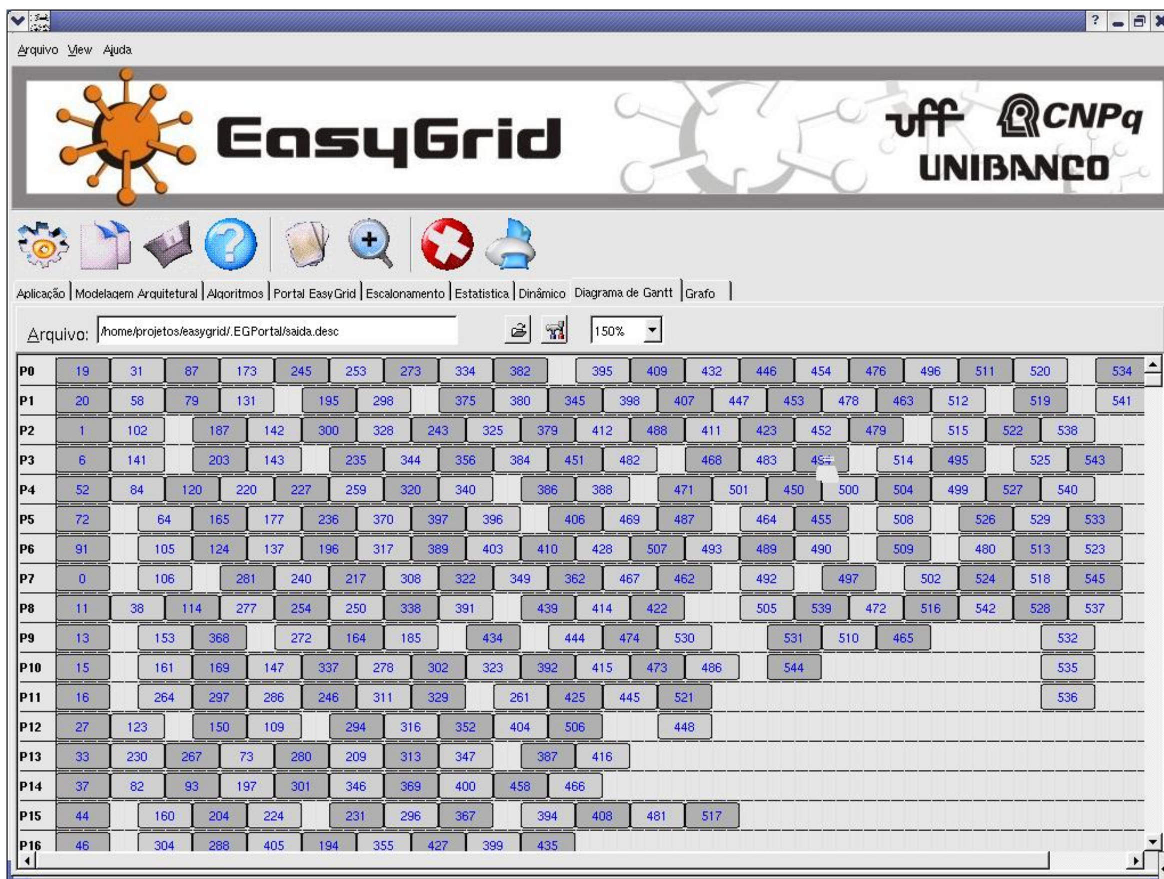

Figura 4.5 Visualização do escalonamento

4.2 Avaliação dos algoritmos de escalonamento estático

A fase de testes de um algoritmo em geral é puramente braçal, já que o desenvolvedor busca comparar os resultados obtidos por seu algoritmo com resultados de

algoritmos reconhecidos como bons na literatura. Para isso é necessário realizar testes com uma grande combinação de variáveis de entrada, tais como: topologias de GADs, granularidades (custos de execução e de comunicação) e tipos de arquitetura diferentes.

O usuário inicialmente define a arquitetura que será utilizada na comparação. Em seguida o usuário tem a possibilidade de selecionar um conjunto de algoritmos e GADs para ser usado como base de comparação dos algoritmos. É gerada uma tabela *algoritmo* × *grafo* com o *makespan* obtido e o número de processadores necessários para o escalonamento gerado por cada algoritmo para cada GAD, fornecendo assim os dados necessários para a análise do desempenho do algoritmo em estudo. Na Figura 4.6, os resultados dos algoritmos DCP, ETF, HEFT e duas novas versões de *List Scheduling* construídos a partir da ferramenta são apresentados para uma série de grafos de entrada.

	DCP	ETF	HEFT(1,1,0)	List-Scheduling(7,4,1,1,1,1)	List-Scheduling(8,1,3,0,1,1)
dag10all1	22(3)	20(3)	20(5)	20(3)	22(3)
Di100	94(10)	94(7)	94(10)	94(7)	108(9)
Di1024	314(32)	314(22)	314(32)	314(22)	372(31)
Di144	114(12)	114(9)	114(12)	114(8)	132(11)
Di16	34(4)	34(3)	34(4)	34(3)	36(3)
Di225	144(15)	144(11)	144(15)	144(10)	168(14)
Di256	154(16)	154(11)	154(16)	154(11)	180(15)
Di25	44(5)	44(4)	44(5)	44(4)	48(4)
Di36	54(6)	54(5)	54(6)	54(4)	60(5)
Di400	194(20)	194(14)	194(20)	194(14)	228(19)
Di9	24(3)	24(3)	24(3)	24(2)	24(2)
fft	28(4)	28(4)	28(8)	28(3)	28(4)
gauss32	140(8)	142(8)	140(26)	142(10)	140(8)
grafo2	52(4)	56(4)	60(6)	60(4)	60(4)
grafo_DCP	1560(3)	1720(4)	1680(9)	1680(5)	1560(5)

Figura 4.6 Resultados comparativos para uma coleção de algoritmos e um conjunto de grafos

Um resumo dos resultados pode ser apresentado como uma coleção de comparações par-a-par (comparando cada par de algoritmos sob investigação) em termos de número de vitórias, empates e derrotas (como pode ser visto na Figura 4.7). Também está disponível o percentual de vezes que cada algoritmo gera o melhor escalonamento do conjunto em teste (é apontado quando o algoritmo é melhor que todos os outros e quando é tão bom quanto algum outro).

Arquivo View Ajuda

EasyGrid uff CNPq UNIBANCO

Aplicação | Modelagem Arquitetural | Algoritmos | Portal EasyGrid | Teste | Comparacao Grid | Estatistica Grid | Dinâmico | Diagrama de Gantt | Grafo

	DCP	ETF	HEFT(1,1,0)	List-Scheduling(7,4,1,1,1,1)	List-Scheduling(8,1,3,0,1,1)
DCP		V(19); E(37); D(3)	V(8); E(46); D(5)	V(10); E(46); D(3)	V(18); E(38); D(3)
ETF	V(3); E(37); D(19)		V(4); E(39); D(17)	V(5); E(39); D(16)	V(18); E(25); D(17)
HEFT(1,1,0)	V(5); E(46); D(8)	V(17); E(39); D(4)		V(9); E(46); D(5)	V(14); E(44); D(2)
List-Scheduling(7,4,1,1,1,1)	V(3); E(46); D(10)	V(16); E(39); D(5)	V(5); E(46); D(9)		V(16); E(34); D(10)
List-Scheduling(8,1,3,0,1,1)	V(3); E(38); D(18)	V(17); E(25); D(18)	V(2); [illegible] 14); D(14)	V(10); E(34); D(16)	

Figura 4.7 Comparação *par-a-par* da coleção de algoritmos

O usuário pode salvar o escalonamento ou a comparação em um arquivo texto ou *html*, e as figuras em um arquivo *jpeg*, além de imprimi-los.

4.3 Validação dos escalonamentos e a execução de aplicações utilizando o *Portal Easygrid*

O *makespan* calculado pelo algoritmo de escalonamento é, na verdade, somente baseado nas características consideradas pelo modelo arquitetural escolhido. O tempo de execução real da aplicação depende, no entanto, de vários outros aspectos, normalmente omitidos de tais modelos arquiteturais. Assim, para validar o escalonamento produzido é necessário executar a aplicação escalonada em um ambiente real ou simulado. Enquanto a simulação oferece a possibilidade de considerar e controlar vários aspectos de ambientes grade, as conclusões práticas dependem da precisão da modelagem adotada pelo simulador. Resultados exatos podem ser obtidos se uma grade real for utilizada, mas as conclusões só se aplicam para aquela grade. A ferramenta tem a opção de utilizar o simulador *SimGrid* [Casa01], que fornece funcionalidades para simular um ambiente paralelo e distribuído (grade) no estudo de escalonamento de aplicações. A ferramenta fornece para o *SimGrid* as especificações das entidades que representam as tarefas e suas dependências em uma dada aplicação, bem como a configuração dos recursos do ambiente. O modelo de desempenho do *SimGrid* também tem que ser especificado pela ferramenta. A utilização deste simulador facilita a avaliação de algoritmos de escalonamento dinâmicos e híbridos em grades com diferentes características. Atualmente a integração do *SimGrid* como funcionalidade da ferramenta é foco do projeto final de Nilmax T. Moura e Luiz T. Menezes [Nilm04]. A ferramenta também permite a execução de aplicações em um ambiente grade real, no caso a Grade Computacional GridRio. Para executar aplicações MPI em ambientes grade deve-se seguir uma seqüência de passos que pode ser complexa e monótona para usuários inexperientes.

Desta forma, para facilitar este trabalho, a funcionalidade de um *Portal Grade* para o servidor *EasyGrid* (tipo de *Resource Management System*) foi integrado à ferramenta de escalonamento. Este *portal* é capaz de iniciar aplicações *EasyGrid*, coletar informações sobre a execução delas e fornecer comparações entre as execuções reais e estimadas estaticamente. Em particular, a última característica é útil para ajudar a determinar a importância da exatidão do modelo arquitetural e os benefícios do escalonamento estático inicial. Neste *Portal*, as etapas necessárias à execução da aplicação são apresentadas na

ordem requisitada, e a seqüência de passos necessária é feita automaticamente através de um processo de seleção de opções.

Inicialmente o usuário deve escolher que tipo de aplicação deseja executar na grade. Há duas possibilidades: aplicações GAD e aplicações MPI (Figura 4.8). No caso de aplicações GAD, a ferramenta gera uma aplicação MPI correspondente ao GAD selecionado pelo usuário [Frei03]. No caso de aplicações MPI o usuário deve escolher o arquivo fonte com o código MPI de sua aplicação.

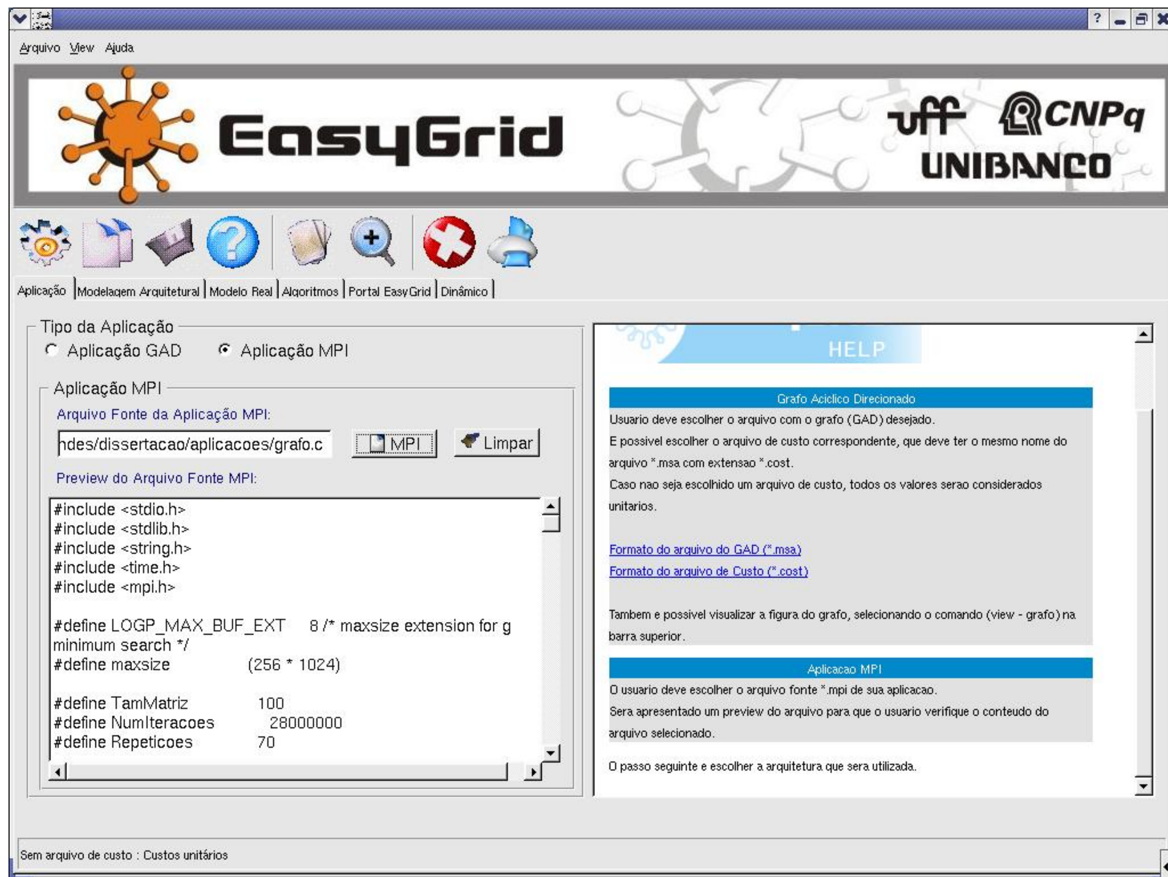

Figura 4.8 Escolha da aplicação MPI

Em seguida o usuário passa para a modelagem arquitetural da grade. A Figura 4.9 apresenta a parte da ferramenta responsável pela modelagem. Em primeiro lugar a ferramenta lê o arquivo de configuração da grade cuja manutenção é responsabilidade do administrador da grade. Neste arquivo estão especificados os *sites* e os *hosts* da grade com

informações necessárias para o modelador, como a versão do MPI suportada por cada host, se o host é servidor de arquivos, entre outras (ver formato do arquivo no Apêndice A). A partir desta lista de *hosts* o usuário deve definir as máquinas às quais tem acesso na grade. Esta informação também pode estar armazenada em um arquivo, que pode ser gerado ou modificado pelo usuário através da ferramenta.

Uma vez que os *hosts* estejam definidos o usuário deve definir a versão MPI de sua aplicação. A partir desta informação é realizada uma nova filtragem dos *hosts*, sendo eliminados aqueles que não suportam a versão do MPI selecionada.

Figura 4.9 Modelagem arquitetural de uma grade real

Agora os *hosts* estão definidos e o usuário deve iniciar a fase de coleta de informações sobre os *hosts* da grade. Para isso, a ferramenta consulta os serviços de diretório da grade (MDS do Globus Toolkit [Fost97]), coletando informações sobre os

hosts da grade que estão ativos naquele instante. Esta fase é importante, pois permite que o usuário verifique naquele momento quais *hosts* estão ativos, qual a carga, a capacidade de processamento, a memória disponível em cada *host* dentre outras informações. Um exemplo é dado na Figura 4.9. A partir destes dados o usuário pode rever sua lista inicial, e definir a lista de máquinas que será considerada pelo modelador. O modelador calibra o modelo HLogP, isto é, coleta os valores de fator de heterogeneidade e os custos de sobrecargas de cada recurso e as latências entre *hosts* da grade [Mend04].

Para realizar a modelagem, é necessário que exista um *proxy* (a ferramenta de segurança do Globus Toolkit [Fost97]) criado pelo usuário. O *proxy* é necessário para o acesso aos recursos da grade. A criação do *proxy* pode ser realizada através da ferramenta se o usuário já possui um certificado guardado na sua conta. O usuário fornece seu *login* e senha e o tempo de duração desejado para o *proxy.* O portal passa a controlar o tempo restante de validade do *proxy*. Neste momento o usuário está pronto para iniciar a modelagem da grade. Após clicar no botão "Modelar a Grade", o usuário deve aguardar até que o modelador gere o arquivo correspondente à arquitetura definida pelo usuário. O arquivo arquitetural gerado pelo modelador será utilizado agora para produzir o escalonamento estático da aplicação do usuário. È bom destacar que o modelo reflete a situação da grade no momento da modelagem e devido ao dinamismo da grade pode não corresponder a situação da grade no momento da execução da aplicação.

Figura 4.10 Portal para a execução das aplicações grade

A Figura 4.10 apresenta a parte da ferramenta responsável pela compilação e execução de aplicações, e onde é possível criar ou destruir um *proxy*. Para executar sua aplicação, o usuário deve em primeiro lugar realizar a compilação. Nesta etapa o usuário deve definir se deseja incluir o *middleware EasyGrid* em sua aplicação. Esta inclusão é realizada de forma automática pela ferramenta. Atualmente o *middleware Easygrid* consegue monitorar a execução da aplicação, coletando informações sobre o andamento da execução e as comunicações realizadas [Frei03]. Futuramente serão incorporadas funcionalidades (tolerância à falhas e escalonamento dinâmico) para tornar a aplicação *system*-aware. Em seguida o usuário define se irá utilizar o escalonamento MPI (*round robin*) ou o escalonamento produzido pelo escalonador estático escolhido na ferramenta.

A aplicação está agora apta a ser executada. Após a execução o usuário pode visualizar o resultado produzido por sua aplicação, utilizando por exemplo um diagrama de

gantt. O *middleware EasyGrid* pode fornecer informações sobre a execução da aplicação e os resultados práticos podem ser agora comparados com os resultados teóricos obtidos pelo escalonador.

4.4 Resumo

Neste capítulo foram apresentadas as funcionalidades oferecidas pela ferramenta *Task Scheduling Testbed* durante o processo de desenvolvimento de um algoritmo de escalonamento, descrevendo os detalhes de funcionamento de cada módulo da ferramenta.

5 CONCLUSÕES E TRABALHOS FUTUROS

O desenvolvimento de softwares de sistema, ferramentas e aplicações paralelas para grades computacionais é um desafio, devido à natureza dinâmica e heterogênea de tais ambientes. O escalonamento de tarefas tem por principal objetivo a execução eficiente de uma aplicação paralela considerando as restrições impostas pelo sistema alvo. No caso das grades, a variação de custo de comunicação, a heterogeneidade dos recursos e a natureza compartilhada tornam ainda mais complexa a atribuição eficiente de tarefas às unidades de processamento.

A ferramenta *Task Scheduling TestBed* apresenta uma interface gráfica bastante atrativa para o usuário, provendo um ambiente de fácil utilização para a construção e análise de algoritmos de escalonamento para grades computacionais através da combinação de diferentes mecanismos. A ferramenta assim oferece a oportunidade de avaliar os escalonamentos gerados por vários algoritmos. A ferramenta integra também as funcionalidades de um portal, que permite a execução e o monitoramento de aplicações MPI *system-aware* em grades baseadas no *Globus Toolkit*.

A estrutura modular da ferramenta facilita não só sua manutenção, mas também sua expansão. Cada algoritmo de escalonamento e funcionalidade de portal é implementado como um executável ou script separado, todos coordenados pela interface gráfica da ferramenta. Um ponto negativo da atual implementação é a necessidade de que todos os módulos, a interface gráfica e as aplicações dos usuários estejam presentes na mesma unidade de processamento da grade. Como trabalho futuro, uma implementação em Java somente da interface gráfica, que se comunicaria via web-services com um nó da grade capaz de executar as funções dos módulos, permitirá que usuários móveis iniciem e

monitorem a execução de uma aplicação na grade através de qualquer máquina com acesso à internet.

Durante este trabalho tivemos a oportunidade de trabalhar como parte de um projeto, o que nos ofereceu uma visão muito importante do trabalho em equipes. Desenvolvendo a parte do *framework EasyGrid* responsável pelo contato com o usuário final, acabamos tendo contato com todas as partes do projeto e adquirindo um bom conhecimento de áreas como o escalonamento de tarefas. O desenvolvimento de um sistema deste porte (certamente o maior que já realizamos) exigiu a aplicação de conhecimentos adquiridos durante o curso, permitindo que o trabalho fosse realizado de forma organizada e padronizada.

6 APÊNDICES

Apêndice A – Formatos dos arquivos de entrada e saída da ferramenta

Neste apêndice serão apresentados os formatos dos arquivos utilizados pela ferramenta como entrada ou saída de dados. Para facilitar a compreensão os formatos serão apresentados através de exemplos. São estes: arquivo representando o grafo (GAD), arquivo de custos associados ao GAD, arquivo de modelo arquitetural e arquivo de saída do escalonador.

i) <u>Formato do Arquivo GAD (*filename.msa*)</u>

```
9        {número de tarefas: nove }
1 3      {tarefa:0 - lista de sucessores: 1, 3}
2 4      {tarefa:1 - lista de sucessores: 2, 4}
5        {tarefa:2 - lista de sucessorer: 5}
4 6      {tarefa:3 - lista de sucessores: 4, 6}
5 7      {tarefa:4 - lista de sucessores: 5, 7}
8        {tarefa:5 - lista de sucessores: 8}
7        {tarefa:6 - lista de sucessores: 7}
8        {tarefa:7 - lista de sucessores: 8}
         {tarefa:8 - lista de sucessores: vazia
                        (nestes casos pode-se usar também -1)}
```

ii) <u>Formato do Arquivo de Custos(associado ao GAD acima)</u>

`9`	{número de tarefas do GAD: 9}
`1`	{custo de execução da tarefa 0: 1}
`2`	{custo de execução da tarefa 1: 2}
`6`	{custo de execução da tarefa 2: 6}
`3`	{custo de execução da tarefa 3: 3}
`5`	{custo de execução da tarefa 4: 5}
`1`	{custo de execução da tarefa 5: 1}
`7`	{custo de execução da tarefa 6: 7}
`2`	{custo de execução da tarefa 7: 2}
`1`	{custo de execução da tarefa 8: 1}
`5 2`	{peso da aresta entre a tarefa 0 e seu primeiro sucessor(1): 5, segundo sucessor(3): 2 (referentes ao arquivo *filename.msa*)}
`2 1`	{peso entre 1 e 2: 2, peso entre 1 e 4: 1}
`1`	{peso entre 2 e 5: 1}
`3 4`	{peso entre 3 e 4: 3, peso entre 3 e 6: 4}
`2 2`	{peso entre 4 e 5: 2, peso entre 4 e 7: 2}
`1`	{peso entre 5 e 8: 1}
`4`	{peso entre 6 e 7: 4}
`3`	{peso entre 7 e 8: 3}
	{tarefa 8 não possui sucessores (pode-se utilizar -1)}

iii) <u>Formato do Arquivo de Modelo Arquitetural</u>

As seguinte abreviações serão adotadas:

- maq: número da máquina
- *fh*: fator de heterogeneidade da máquina
- nome: nome da máquina

- O_s: *Overhead* de envio
- O_r: *Overhead* de recebimento

`4`	{número de máquinas na arquitetura: 4}
`5 goiaba.ic.uff.br 2 4`	{maq 0: *fh*: 5, hostnome: goiaba, O_s: 2, O_r: 4}
`3 pequi.ic.uff.br 3 2`	{maq 1: *fh*: 3, nome: pequi, O_s: 3, O_r: 2}
`1 abacate.ic.uff.br 1 1`	{maq 2: *fh*: 1, nome: abacate, O_s: 1, O_r: 1}
`2 pitanga.ic.uff.br 1 2`	{maq 3: *fh*: 2, nome: pitanga, O_s: 1, O_r : 2}
`0 1 4 3`	{matriz 4x4 descrevendo a latência de
`2 0 1 5`	comunicação entre as máquinas. Seja a_{ij} um
`6 1 0 3`	elemento da matriz. Este elemento representa a
`2 4 1 0`	latência de comunicação da máquina *i* para a
	a máquina *j*}

iv) Formato do Arquivo de Saída do Escalonador

`9`	{numero de tarefas escalonadas}
`9`	{numero de processadores considerados para escalonamento}
`20`	{*makespan*}
`0`	{tempo de execução do escalonador em segundos}
`105`	{tempo de execução do escalonador em milisegundos}
`0 abacate 5`	{processador: 0, nome: abacate, 5 tarefas escalonadas}
`0 0 1`	{primeira tarefa: id: 0, tempo_inicio: 0, tempo_fim:1}
`3 2 5`	{segunda tarefa: id: 3, tempo_inicio: 2, tempo_fim: 5}
`1 6 8`	{terceira tarefa: id: 1, tempo_inicio: 6, tempo_fim: 8}
`6 9 12`	{quarta tarefa: id: 6, tempo_inicio: 9, tempo_fim: 12}
`8 19 20`	{quinta tarefa: id: 8, tempo_inicio: 19, tempo_fim: 20}
`1 caju 2`	{processador: 1, nome: caju, 2 tarefas escalonadas}
`2 3 5`	{primeira tarefa: id: 2, tempo_inicio: 3, tempo_fim: 5}
`5 10 13`	{segunda tarefa: id: 5, tempo_inicio: 10, tempo_fim: 13}

```
2 pinha 1       {processador: 2, nome: pinha, 1 tarefa escalonadas}
7 7 10          {primeira tarefa: id: 7, tempo_inicio: 7, tempo_fim:10}
3 pequi 1       {processador: 3, nome: pequi, 1 tarefa escalonadas}
4 3 7           {primeira tarefa: id: 4, tempo_inicio: 3, tempo_fim:7}
4 noname 0      {processador: 4, nome: noname, 0 tarefas escalonadas}
5 noname 0      {processador: 5, nome: noname, 0 tarefas escalonadas}
6 noname 0      {processador: 6, nome: noname, 0 tarefas escalonadas}
7 noname 0      {processador: 7, nome: noname, 0 tarefas escalonadas}
8 noname 0      {processador: 8, nome: noname, 0 tarefas escalonadas}
```

v) Formato do arquivo GRUniverse, de configuração da grade

```
# Esta é uma lista dos recursos do GridRio, i.e. recursos
# para os quais certificados foram expedidos e são
# confiáveis.
#
# Formato do Arquivo:
#    a primeira linha contém o número de recursos, em seguida
# informações são identificado pelos símbolos S e M:
#          S identifica um nome do site seguido pelo número de
#            recursos deste site,  e se site utiliza o sistema
#            NFS [NÃO=0/SIM=1]
#          M identifica o nome do host, se tem MPICH-G2
#            instalado [NÃO=0/SIM=1], se tem MPILAM  [0/1], e
#            se esta máquina é um fileserver [0/1].

     43
     S ic.uff.br 11 1
     M abacate.ic.uff.br 1 1 1
     M caju.ic.uff.br 1 1 0
     M goiaba.ic.uff.br 1 1 0
```

```
M jenipapo.ic.uff.br 1 1 0
M melancia.ic.uff.br 1 1 0
M oiti.ic.uff.br 1 1 0
M pequi.ic.uff.br 1 1 0
M pinha.ic.uff.br 1 1 0
M pitanga.ic.uff.br 1 1 0
M sapoti.ic.uff.br 1 1 0
M siriguela.ic.uff.br 1 1 0

S par.inf.puc-rio.br 32 0
M n00.par.inf.puc-rio.br 1 0 1
M n01.par.inf.puc-rio.br 1 0 1
M n02.par.inf.puc-rio.br 1 0 1
M n03.par.inf.puc-rio.br 1 0 1
M n04.par.inf.puc-rio.br 1 0 1
M n05.par.inf.puc-rio.br 1 0 1
M n06.par.inf.puc-rio.br 1 0 1
M n07.par.inf.puc-rio.br 1 0 1
M n08.par.inf.puc-rio.br 1 0 1
M n09.par.inf.puc-rio.br 1 0 1
M n10.par.inf.puc-rio.br 1 0 1
M n11.par.inf.puc-rio.br 1 0 1
M n12.par.inf.puc-rio.br 1 0 1
M n13.par.inf.puc-rio.br 1 0 1
M n14.par.inf.puc-rio.br 1 0 1
M n15.par.inf.puc-rio.br 1 0 1
M n16.par.inf.puc-rio.br 1 0 1
M n17.par.inf.puc-rio.br 1 0 1
M n18.par.inf.puc-rio.br 1 0 1
M n19.par.inf.puc-rio.br 1 0 1
M n20.par.inf.puc-rio.br 1 0 1
M n21.par.inf.puc-rio.br 1 0 1
```

```
M n22.par.inf.puc-rio.br 1 0 1
M n23.par.inf.puc-rio.br 1 0 1
M n24.par.inf.puc-rio.br 1 0 1
M n25.par.inf.puc-rio.br 1 0 1
M n26.par.inf.puc-rio.br 1 0 1
M n27.par.inf.puc-rio.br 1 0 1
M n28.par.inf.puc-rio.br 1 0 1
M n29.par.inf.puc-rio.br 1 0 1
M n30.par.inf.puc-rio.br 1 0 1
M n31.par.inf.puc-rio.br 1 0 1
```

Apêndice B – Publicações

Foram publicados ou submetidos os seguintes artigos relacionados a este trabalho de Projeto Final e Iniciação Científica feito durante o período de março 2003 a julho 2004:

- J.A. Silva, **A.A. Fonseca**, **B.A. Vianna**, C. Boeres e V.E.F. Rebello, A Grade Computacional GridRio e o Projeto EasyGrid. *Nos Anais do II Workshop em Grade Computacional e Aplicações*, Programa de Verão 2004 do LNCC, Petrópolis, RJ, Brazil, fevereiro, 2004.

- **B.A. Vianna**, **A.A. Fonseca**, N.T. Moura, L.T. Menezes, H.A. Mendes, J.A. Silva, C. Boeres e V.E.F. Rebello, *EasyGrid*: A Tool for the Design and Evaluation of Hybrid Scheduling Algorithms for Computational Grids. Submetido ao *The Proceedings of the 2nd International Workshop on Middleware for Grid Computing*, Toronto, Canadá, outubro, 2004.

- **A.A. Fonseca**, **B.A. Vianna**, N.T. Moura, L.T. Menezes, C. Boeres e V.E.F. Rebello, Um Ambiente para o Desenvolvimento e Avaliação de Algoritmos de Escalonamento para Grades Computacionais. In *5º Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2005)*, Foz do Iguaçu, PR, Brazil, outubro, 2004.

7 REFERÊNCIAS

[Boer02] C. Boeres e V.E.F. Rebello. LogP cluster-based scheduling: Theory and practice. In A. De Souza, C. Amorim, e N. Reis Jr, editors, The *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2002)*, Vitória, Brazil, October 2002. IEEE Computer Society Press.

[Boer02a] C. Boeres and V. E. F. Rebello. On solving the static task scheduling problem for real machines. In *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications*, chapter 3, pages 53-84. Kluwer Academic Publishers, 2002.

[Boer03] C. Boeres, A. Lima e V.E.F. Rebello. Hybrid Task Scheduling: Integrating Static and Dynamic Heuristics. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBACPAD'2003)*, São Paulo, Brasil, Novembro de 2003. IEEE Computer Society Press.

[Boer04] C. Boeres e V.E.F. Rebello. *EasyGrid*: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience*, 16 (5): 425-432, 2004. John Wiley and Sons

[BorlWeb] Borland. http://www.borland.com.br/

[Buyy99] R. Buyya (editor). *High Performance Cluster Computing: Architectures and Systems*. Vol. 1. Prentice Hall, USA, 1999.

[Card04] D. F. Cardoso, Escalonamento Estático de Tarefas em Redes Computacionais Heterogêneas Sob o Modelo LogP. Tese de Mestrado Instituto de Computação Universidade Federal Fluminense, Previsto para agosto de 2004.

[Casa01] H. Casanova, SimGrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of First IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2001

[Cull93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, e T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, USA, May 1993.

[EasyGridWeb] Projeto *EasyGrid*. http://easygrid.ic.uff.br/

[Este03] M.A.N. Esteves *Rumo a uma heurística rápida para geração de escalonamentos eficientes em grids computacionais* , Tese de Mestrado, Instituto de Computação, Universidade Federal Fluminense, junho de 2003.

[Frei03] P. M. S. Freire, *Monitoramento para aplicações MPI system-aware*, Tese de Mestrado, Instituto de Computação, Universidade Federal Fluminense, junho de 2003.

[Fost97] I. Foster e C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2): 115-128, 1997.

[Fost99] I. Foster e C. Kesselman (editors). *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[Gajs90] D. D. Gajski and M. –Y Wu, "Hypertool: A programming Aid for Message-Passing Systems," IEEE Trans. Parallel and Distributed Systems, vol. 1, nº.3, pp. 330-343, July 1990.

[Gare79] M. R. Garey e D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1979

[GCInfoWeb] Grid Computing Infoware. http://www.gridcomputing.com/

[GlobusWeb] The Globus Project. http://www.globus.org/

[GProjWeb] IEEE Distributed Systems Online: Grid computing projects. http://dsonline.computer.org/gc/gcprojects.htm

[GraphVizWeb] GraphViz. http://www.graphviz.org

[GridRioWeb] GridRio Computational Grid. http://easygrid.ic.uff.br/grid/GridRio.html

[Hwan89] J. Hwang, Y. Chow, F. Anger e B. Lee, Scheduling precendence graphs in systems with interprocessor communications times. *SIAM Journal of Computing*, 18(2):1-8, 1989. *Journal of Computer and System Sciences*, 10:384-393, 1975.

[Kwok96] Y-K Kwok e I. Ahmad. Dynamic Critical–Path scheduling: an effective technique for allocating tasks graphs to multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 7(5):506-521, May 1996.

[Kwok99a] Y-K Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), Dec. 1999.

[Kwok99b] Y. K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381-422, December 1999.

[Krau01] K. Krauter, R. Buyya e M. Maheswaran, A Taxonomy and Survey o Grid Resource Management Systems for Distributed Computing, *Software: Practice and Experience*, 32(2):135-164, 2002. John Wiley and Sons.

[Mahe99] M. Maheswaran e H. J. Siegel. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *The Proceedings of 8th Heterogeneous Computing Workshop* (HCN'99), April 1999.

[Mart97] R. P. Martin; A. M. Vahdat, D. E. Culler e T. E. Anderson: Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *The Proceedings of the Annual International Symposium on Computer Architecture*, 24, June 1997, p. 85-97.

[Mend04] H.A. Mendes, *HLogP*: *Um modelo de escalonamento para a execução de aplicações MPI em grades computacionais*, Tese de Mestrado,

Instituto de Computação, Universidade Federal Fluminense, Previsto para agosto de 2004.

[MPI95] Message Passing Forum. *MPI: a Message Passing Interface*. Technical report, University of Tennessee, 1995.

[Nguy00] A. Nguyen-t Uonng: Integrating fault tolerance techniques in grid applications. August2000.(PhD Thesis) - School of Engineering and Applied Science, University of Virginia, 2000.

[Nilm04] N.T Moura e L.T. Menezes, *Políticas de Escalonamento Dinâmico de Aplicações em Ambientes Heterogêneos: Implementação e Análise*, Monografia de Projeto Final, Departamento de Ciência da Computação, Universidade Federal Fluminense, Previsto para dezembro de 2004.

[Papa90] C.H. Papadimitriou, e M. Yannakakis, Towards and architecture-independent analysis of parallel algorithms. SIAMJ Computer, v. 19, p. 322-328, 1990.

[Rebe03] V.E.F. Rebello e C. Boeres, Projeto *EasyGrid*: Um framework para a habilitação automática de aplicações MPI em Grids Computacionais (e a Iniciativa GridRio). Nos *Anais do I Workshop em Grade Computacional e Aplicações*, Programa de Verão 2003 do LNCC, Petrópolis, RJ, Brasil, Janeiro, 2003.

[Silv04] J.A. Silva, A.A. Fonseca, B.A. Vianna, C. Boeres e V.E.F. Rebello, A Grade Computacional GridRio e o Projeto *EasyGrid*. No *Anais do II Workshop em Grade Computacional e Aplicações*, Programa de Verão 2004 do LNCC, Petrópolis, RJ, Brazil, Fevereiro, 2004.

[Sinn00] O. Sinnen and L. Sousa. A platform independent parallelising tool based on graph theoretic models. In *Proc. of the International Conference on Vector and Parallel Processing (VECPAR 2000)*, pages 154-167. Springer Verlag, 2000.

[Topc02] H. Topcuoglu, S. Hariri, e M. Wu, Performanceeffective and low-complexity task cheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13,(3): 260-274, March 2002.

[Ullm75] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384-393, 1975.

[Whal01] R. Whaley, Petitet e J. Dongarra: Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, v. 27, n. 1, p. 3-35, 2001.

[Wols99] R. Wolski, N. Spring, and J. Haye. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 1999.