**UNIVERSIDADE FEDERAL DE SÃO CARLOS**
**DEPARTAMENTO DE COMPUTAÇÃO**
**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

# Escalonamento de Aplicações Paralelas: de clusters para grids

Daniele Santini Jacinto

**São Carlos – SP**
**Agosto/2007**

# Universidade Federal de São Carlos

## Centro de Ciências Exatas e de Tecnologia

### Programa de Pós-Graduação em Ciência da Computação

# *"Escalonamento de aplicações paralelas: de clusters para grids"*

**DANIELE SANTINI JACINTO**

**Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.**

**Membros da Banca:**

**Prof. Dr. Hélio Crestana Guardia**
**(Orientador – DC/UFSCar)**

**Prof. Dr. Luis Carlos Trevelin**
**(DC/UFSCar)**

**Prof. Dr. Bruno Richard Schulze**
**(LNCC)**

**São Carlos**
**Agosto/2007**

**Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar**

J12ea

Jacinto, Daniele Santini.
Escalonamento de aplicações paralelas : de clusters para grids / Daniele Santini Jacinto. -- São Carlos : UFSCar, 2007.
116 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2007.

1. Grade computacional. 2. Escalonamento. 3. MPI. Título.

CDD: 004.36 (20ª)

# Agradecimentos

O Mestrado será um período da minha vida que tenho certeza que nunca esquecerei. Mais do que conhecimentos, tive muitas lições de vida. Aprendi que a perseverança e a troca de informações resolvem dúvidas. Aprendi que o tempo, a paciência, a leitura e a vontade de fazer algo acontecer transformam assuntos desconhecidos em trabalhos interessantes. Aprendi que a disciplina pode cumprir *deadlines* e aprendi que o trabalho em equipe pode construir algo maior.

Agradeço ao Prof. Dr. Hélio Crestana Guardia pela confiança em mim depositada, pela orientação que recebi ao longo de todo o desenvolvimento deste trabalho, por sua solicitude e dedicação em cada artigo escrito, em cada dúvida que lhe foi repassada e na conclusão dessa dissertação.

Agradeço ao amigo de mestrado Ricardo Araújo Rios, com quem tive o prazer de realizar pesquisas conjuntas que contribuiram imensamente para os resultados obtidos por esse trabalho.

Agradeço ao meu ex-chefe Edson Fernando Italiano, e ao atual, Antonio José da Costa Filho, por sempre terem colaborado para que eu pudesse adequar meus horários de trabalho e de estudo.

Agradeço a meus pais, Norivaldo e Maria, por sempre terem me ensinado a agradecer tudo o que a vida me proporciona.

Agradeço a Deus por mais essa oportunidade, por tudo que aprendi e por todas as pessoas que conheci.

# Resumo

Algoritmos diferentes possibilitam o escalonamento eficiente de aplicações paralelas em plataformas computacionais heterogêneas e distribuídas, tais como *grids* computacionais. Vários algoritmos de escalonamento para esses ambientes necessitam de um modelo de aplicação representado por um grafo acíclico direcionado (GAD), selecionando tarefas para execução de acordo com suas características de comunicação e de processamento.

A obtenção de um GAD para uma aplicação real, contudo, não é uma questão simples. O conhecimento necessário sobre as tarefas da aplicação e as comunicações entre elas, considerando ciclos de transmissão, dificulta a elaboração de um grafo apropriado.

Particularmente, programas MPI, os quais representam uma parcela significativa das aplicações paralelas, apresentam um modelo de comunicação cíclico entre o nó *master* e os nós de processamento. Esse comportamento impede a utilização de muitos algoritmos de escalonamento devido ao fato de eles percorrerem o grafo recursivamente para priorizar as tarefas.

Nesse sentido, esse trabalho apresenta um mecanismo para a criação automática de GADs para aplicações MPI reais originalmente desenvolvidas para *clusters* homogêneos. Para essa implementação, aplicações são monitoradas durante a execução em um *cluster* e os dados coletados são usados para a elaboração de um GADs apropriados. Dependências de dados são identificadas e ciclos existentes entre as tarefas são eliminados.

O algoritmo de escalonamento HEFT é usado para avaliar o modelo de aplicação e o escalonamento obtido é então automaticamente convertido em um arquivo RSL (*Resource Specification Language*) para execução em um *grid* com Globus.

Resultados de execuções de aplicações reais e simulações demonstram que o uso de *grid* pode ser vantajoso.

**Palavras-chave:** *cluster*, MPI, *grids* computacionais, grafos acíclicos direcionados (GAD), escalonador de aplicação, HEFT, Globus, GridSim.

# Abstract

Different algorithms provide efficient scheduling of parallel applications on distributed and heterogeneous computational platforms, such as computational grids.

Most scheduling algorithms for such environments require an application model represented by a directed acyclic graph (DAG), selecting tasks for execution according to their processing and communication characteristics.

The obtainment of DAGs for real applications, however, is not a simple quest. The required knowledge about the application tasks and the communication among them, considering existing transmission cycles, harden the elaboration of appropriate graphs.

Particularly, MPI programs, that represent a meaningful portion of existing parallel applications, usually present a cyclic communication model among the master and the processing nodes. This behavior prevents most scheduling algorithms to be employed as they recursively traverse the graphs to prioritize the tasks.

In this sense, this work presents a mechanism for the automatic creation of DAGs for real MPI application originally developed for homogeneous clusters. In order to do so, applications go through a monitored execution in a cluster and the collected data are used for the elaboration of an appropriate DAGs. Data dependencies are identified and existing cycles among the tasks are eliminated. The HEFT scheduling algorithm is used to evaluate the application model and the schedule obtained is then automatically converted into an RSL (Resource Specification Language) file for execution in a grid with Globus.

Results from running real applications and simulations show using the grid can be advantageous.

**Key-Words:** cluster, MPI, computational grids, direct acyclic graph (DAG), application scheduler, HEFT, Globus, GridSim.

# Lista de Figuras


2.1 Protocolos que compõem o *middleware Globus* . . . . . p. 6

2.2 Estrutura de uma Organização Virtual . . . . . p. 7

2.3 Hierarquia de certificados digitais . . . . . p. 8

2.4 GRAM integra solicitações externas com ambiente interno de um recurso do *grid* . . . . . p. 9

2.5 Interação entre o serviço GRAM (*Globus*) e o escalonador de recurso . . . . p. 10

2.6 Componentes do *Globus Toolkit* 4 . . . . . p. 11

3.1 Escalonamento em *grids* computacionais . . . . . p. 13

3.2 Estrutura de um escalonador de aplicação . . . . . p. 14

3.3 Modelo de aplicação . . . . . p. 14

3.4 Escalonamento utilizando o modelo LogP . . . . . p. 16

3.5 Classificação dos algoritmos de escalonamento estático . . . . . p. 17

4.1 Ambiente de *grid* Simulado . . . . . p. 26

5.1 Ilustração dos mecanismos implementados nesse trabalho . . . . . p. 29

5.2 Visualização gráfica da tarefa 2 antes da sintetização das comunicações . . . p. 34

5.3 Grafo condensado do NPB IS classe C gerado pela ferramenta *Graphviz* . . . p. 36

5.4 NPB classe C após a remoção dos ciclos . . . . . p. 37

6.1 Ambiente de *grid* Simulado . . . . . p. 44

6.2 Grafo gerado do MPI *MergeSort* . . . . . p. 45

6.3 Análise do *Intel Trace Analyzer* . . . . . p. 45

6.4 Grafo gerado da Mutilplicação de Matrizes Paralela . . . . . p. 47

8.1 NAS *Benchmark* classe A com NP=4 . . . . . p. 62

8.2 NAS *Benchmark* classe C com NP=4 . . . . . p. 63

8.3 NAS *Benchmark* classe C com NP=8 . . . . . p. 63

# Lista de Tabelas


2.1 Arquivo RSL para submissão no Globus . . . . . p. 10

3.1 Arquivo de submissão no escalonador de recurso Condor . . . . . p. 20

4.1 Trecho da classe *Program* onde o usuário é criado . . . . . p. 24

4.2 Trecho do método *createGridle*t onde a tarefa 0 de uma aplicação é criada . . p. 24

4.3 Trecho da classe *Program* que cria um recurso no simulador GridSim . . . . p. 25

4.4 Trecho da classe *Program* onde os roteadores são criados . . . . . p. 26

4.5 Trecho da classe *Program* onde o usuário e recursos são conectados a um roteador . . . . . p. 27

4.6 Trecho da classe *Program* onde os roteadores são conectados entre si . . . . . p. 27

5.1 Partes do *Tracefile* do IS NAS classe C . . . . . p. 32

5.2 Partes do *Tracefile* do IS NAS classe C - mostrando o custo computacional de uma tarefa . . . . . p. 33

5.3 Partes do *Tracefile* do IS NAS classe C - mostrando o peso de uma comunicação p. 33

5.4 Partes do *Tracefile* do IS NAS classe C - comunicações globais . . . . . p. 34

5.5 Arquivo de entrada de dados da ferramenta *Graphviz* - gerado pela análise do *tracefile* . . . . . p. 35

5.6 Algoritmo de escalonamento HEFT . . . . . p. 38

5.7 Arquivo RSL . . . . . p. 41

5.8 Resultado do escalonamento da Multiplicação de Matrizes com NP=5 . . . . p. 42

6.1 Remoção de ciclos do grafo aleatório 1 . . . . . p. 47

6.2 Remoção de ciclos do grafo aleatório 2 . . . . . p. 47
6.3 Trecho do método *body()* - mensagens iniciais . . . . . p. 48
6.4 Trecho do método *processOtherEvent()* - *tag* SEND_PKT . . . . . p. 49
6.5 Trecho do método *processOtherEvent()* - *tag* RECV_PKT . . . . . p. 49
6.6 Trecho do método *body()* - mensagens finais . . . . . p. 49
6.7 Trecho do método *processOtherEvents()* - *tag* SEND2_PKT . . . . . p. 50
6.8 Tempo de execução para o *cluster* real e para o simulado - NP=5 . . . . . p. 50
6.9 Tempo de execução para o *cluster* real e para o simulado - NP=32 . . . . . p. 51
6.10 Tempos de execução no *cluster* para MPICH e MPICH-G2 - NP=5 . . . . . p. 52
6.11 Tempos de execução no *cluster* para MPICH e MPICH-G2 - NP=32 . . . . . p. 52
6.12 Tempo de execução no *cluster* para MPICH e MPICH-G2 - matriz com dimensão 4096x4096, usando diferentes valores de NP . . . . . p. 52
6.13 Execução no *grid* simulado com rede *Fast Ethernet* - NP=5 . . . . . p. 53
6.14 Execução em *grid* simulado com rede *Fast Ethernet* - NP=32 . . . . . p. 55
6.15 Execução no *grid* com rede *Gigabit* - NP=5 . . . . . p. 56
6.16 Execução no *grid* com rede *Gigabit* - NP=32 . . . . . p. 57

# Sumário

**1 Introdução** p. 1

**2 *Grids* computacionais** p. 4

2.1 Considerações Iniciais . . . . . p. 4

2.2 *Globus Toolkit* 4.0 . . . . . p. 5

**3 Escalonamento em *grids* computacionais** p. 12

3.1 Considerações Iniciais . . . . . p. 12

3.2 Escalonador de Aplicação . . . . . p. 13

3.2.1 Modelo de aplicação . . . . . p. 14

3.2.2 Modelo arquitetural . . . . . p. 15

3.2.3 Critério de desempenho . . . . . p. 16

3.3 Escalonador de recursos . . . . . p. 19

**4 Simulação de ambientes de *grids* computacionais** p. 21

4.1 Considerações Iniciais . . . . . p. 21

4.2 GridSim: Criação de tarefas e usuários . . . . . p. 23

4.3 GridSim: Criação de recursos . . . . . p. 24

**5 Escalonamento de aplicações MPI em *grids* computacionais** p. 28

5.1 Considerações Iniciais . . . . . p. 28

5.2 Geração de GAD . . . . . p. 30
5.2.1 Monitoramento em *cluster* . . . . . p. 30
5.2.2 Análise do *tracefile* . . . . . p. 32
5.2.3 Sintetização do grafo . . . . . p. 35
5.2.4 Remoção de ciclos . . . . . p. 36
5.3 Adequando o algoritmo HEFT . . . . . p. 38
5.3.1 Classificação das tarefas . . . . . p. 39
5.3.2 Descoberta e seleção de recursos . . . . . p. 39
5.3.3 Relação entre tarefas e recursos do *grid* . . . . . p. 40
5.3.4 Algoritmo EFT . . . . . p. 40
5.4 Submissão no *grid* . . . . . p. 41

**6 Resultados** p. 43

6.1 Considerações Iniciais . . . . . p. 43
6.2 Validação do grafo produzido . . . . . p. 44
6.3 Validação da fase de remoção de ciclos . . . . . p. 46
6.4 Simulando aplicações MPI no GridSim . . . . . p. 46
6.5 Ambiente real X Ambiente simulado . . . . . p. 50
6.6 MPICH X MPICH-G2 . . . . . p. 51
6.7 Computação X Comunicação . . . . . p. 52
6.7.1 Análise de desempenho para NP=5 . . . . . p. 53
6.7.2 Análise de desempenho para NP=32 . . . . . p. 54
6.8 Seleção de recursos aleatória X escalonador de aplicação . . . . . p. 54
6.8.1 Análise para NP=5 . . . . . p. 54
6.8.2 Análise para NP=32 . . . . . p. 55
6.9 Discussão . . . . . p. 56

6.10 Considerações sobre os trabalhos experimentais . . . . . p. 56

**7 Conclusões** p. 60

**8 Trabalhos Futuros** p. 62

**9 Referências** p. 65

**10 Anexo A - Código que simula o *grid* computacional usado nesse trabalho** p. 69

**11 Anexo B - *Script* que realiza a análise do *tracefile*** p. 87

**12 Anexo C - *Script* que sintetiza os dados coletados** p. 92

**13 Anexo D - Código utilizado para remover ciclos dos grafos** p. 95

**14 Anexo E - Código que gera um modelo arquitetutal** p. 98

**15 Anexo F - Implementação do HEFT** p. 109

# 1. Introdução

Um ambiente de processamento paralelo é constituído por múltiplos processadores, memória centralizada ou distribuída, um sistema operacional que permita a manipulação desse processamento e um algoritmo paralelo.

Na implementação dessa abordagem, pode-se citar a utilização de um sistema multicomputador projetado com base na passagem de mensagens.

Esse sistema se caracteriza por apresentar diversos computadores interconectados através de uma tecnologia de rede. Essa arquitetura, também denominada *cluster*, não possui memória compartilhada. Cada computador tem acesso apenas às informações armazenadas em sua memória local e uma das alternativas para a troca de dados é a utilização de bibliotecas de passagem de mensagens baseadas nos protocolos da arquitetura TCP/IP (*Transmission Control Protocol/Internet Protocol*).

A popularidade desse sistema é decorrente do baixo custo, quando comparado ao dos computadores paralelos, e da escalabilidade apresentada. Não existem limites para o número de processadores, e estes podem ser adicionados ao sistema de acordo com a necessidade da aplicação.

A impressionante melhoria de desempenho que as redes de computadores vêm apresentando despertou a idéia de utilizar este sistema multicomputador e outros recursos computacionais, conectados de forma independente, dando origem à computação em grade (*grid computing*) [1].

Esse novo ambiente dispõe de muitos recursos compartilhados entre diferentes domínios institucionais e esses compartilhamentos envolvem acesso direto a computadores, software e dados e necessita ser altamente controlado. Esse conjunto de recursos, usuários e regras de compartilhamento forma uma Organização Virtual (*Virtual Organization*) [1].

Para estabelecer relacionamentos entre todos esses componentes, interoperabilidade passa

a ser um dos pontos centrais da computação em grade e isso significa utilização de protocolos comuns. Protocolos para informação de serviços, gerenciamento de recursos, transferência de dados e autenticação segura de usuários compõem um *middleware* que será responsável por gerenciar essa organização [2].

Uma especialização da computação em grade são os sistemas destinados aos serviços de processamento, denominados grades computacionais (*computational grids*), utilizados para a execução de aplicações paralelas. Neste trabalho adotou-se chamar essa infra-estrutura computacional de *grid*.

Com a possibilidade de englobar sistemas multicomputadores em *grids* surge a necessidade de processar os códigos paralelos desenvolvidos para estes sistemas nesse novo ambiente computacional. O desafio é executar estes códigos de forma eficiente e robusta sem colocar essa responsabilidade sobre o programador ou sobre o usuário.

Utilizando a técnica de escalonamento baseado em *task-graph* [3, 4], um modelo de aplicação pode ser criado para representar suas tarefas e as comunicações entre elas. Comunicações representam a transmissão de dados e a relação de dependência entre as tarefas [5].

O conhecimento das características relevantes de uma aplicação, modeladas por um *task-graph*, pode ser utilizado para melhorar o escalonamento. Se a tarefa $v_j$, que depende da tarefa $v_i$, é escalonada antes de seu predecessor, o recurso selecionado é alocado por mais tempo do que o necessário, devido ao fato de $v_j$ precisar de um resultado produzido por $v_i$. Técnicas de escalonamento baseadas em *task-graph* resultam em uma sequência apropriada de escalonamento das tarefas de uma aplicação.

A obtenção de um *task-graph* para aplicações reais que utilizam passagem de mensagens, contudo, é ainda um problema em aberto para muitos sistemas de escalonamento baseados nesse tipo de modelo de aplicação [6].

Dessa forma, utilizando príncipios de escalonamento de tarefas baseados em *task-graph*, esse trabalho apresenta a implementação de mecanismos que viabilizam a modelagem de uma aplicação que foi desenvolvida para *clusters*. Essa modelagem permite que algoritmos de escalonamento selecionem os melhores recursos de um *grid*, de acordo com as necessidades de cada aplicação, tornando assim sua execução nesses ambientes eficiente.

Esse trabalho está organizado em diversos capítulos sendo que o Capítulo 2 apresenta o *middleware Globus*, com o objetivo de demonstrar como um *grid* computacional é constituído

e gerenciado, como são incluídos usuários e recursos em uma Organização Virtual e como as aplicações paralelas são executadas nesse ambiente. Esse *middleware* é responsável por compor o *grid* computacional sobre o qual os mecanismos apresentados por esse trabalho atuam.

O Capítulo 3 descreve o processo de escalonamento em *grids* computacionais e introduz conceitos, técnicas, heurísticas e algoritmos de escalonamento primordiais para o desenvolvimento dos mecanismos propostos.

O Capítulo 4 trata da simulação de *grids* computacionais. Particularmente, mostra como usuários, tarefas e recursos são criados no simulador *GridSim*, um simulador para ambientes multi-usuários implementado em *Java*, e que foi utilizado para realizar testes de comparação entre tempos de execuções em *cluster* e em *grid*.

O Capítulo 5 descreve como foram implementados os mecanismos para modelagem de aplicações reais. São descritos os processos de monitoramento dessas aplicações pela ferramenta *Trace Collector*, a análise desse monitoramento, a remoção de ciclos da modelagem obtida e a adequação de um algoritmo de escalonamento ao modelo de aplicação gerado.

O Capítulo 6 valida o modelo de aplicação gerado, testa o mecanismo de remoção de ciclos e apresenta os tempos de execuções das aplicações modeladas em ambientes de *clusters* e *grids*, reais e simulados. O Capítulo 7 apresenta as conclusões e o Capítulo 8 descreve pesquisas futuras que podem tornar esse trabalho mais completo.

# 2. *Grids* computacionais

Esse capítulo descreve os conceitos de *grid* computacional e Organização Virtual, os componentes necessários a essas arquiteturas, suas funções e a interação existente entre eles. Como exemplo de implementação de uma Organização Virtual, descreve os princípios de funcionamento do *middleware Globus*.

## 2.1 Considerações Iniciais

Em 1969, *Leonard Kleinrock* [1] sugeriu premeditadamente que teríamos uma quantidade de serviços de computadores, os quais, como os serviços de eletricidade e telefone, serviriam casas e escritórios através dos diversos países [1].

Em meados de 1990, o termo "*O grid*" foi formulado para representar uma nova infraestrutura de computação distribuída proposta para a área de ciência e engenharia avançada [2] e em 1998 *Carl Kesselman* e *Ian Foster*, definiram *grid* no livro "*The Grid: Blueprint for a New Computing Infraestructure*"[7], como uma infraestrutura de *hardware* e *software* que oferece confidencialidade, consistência, ampla disponibilidade e acesso à computação de alto desempenho sem altos custos.

Em um artigo [2] subsequente ao livro acima citado, no ano de 2000, os autores refinaram a definição de *grid* utilizando conceitos de compartilhamento de recursos coordenados e solução de problemas através de organizações virtuais dinâmicas e multi-institucionais. O conceito chave é a capacidade de negociar recursos compartilhados entre um conjunto de participantes (provedores e consumidores) e então utilizar todos esses recursos para algum propósito.

Essa nova abordagem deu origem ao termo Organização Virtual onde o compartilhamento mencionado não é a simples troca de arquivos mas inclui o acesso a computadores, *software*,

[1] *Leonard Kleinrock* criou os princípios básicos de comutação de pacotes durante sua graduação no MIT e em Setembro de 1969 seu computador, localizado na UCLA, foi o primeiro nó da *Internet* atual

dados e outros recursos. Esse compartilhamento é, necessariamente, altamente controlado, com provedores de recursos e consumidores, definindo claramente e cuidadosamente o que é compartilhado, quem pode compartilhar, e as condições sobre as quais os compartilhamentos ocorrem. Um conjunto de indivíduos ou instituições definido por tais regras de compartilhamento forma uma Organização Virtual [2].

A partir dessa época, pesquisas e esforços de desenvolvimento na comunidade *grid* têm produzido protocolos, serviços e ferramentas que descrevem precisamente os desafios que aparecem quando deseja-se constituir Organizações Virtuais escaláveis. Essas tecnologias incluem soluções que suportam (i) gerenciamento de credenciais e políticas quando a computação reúne múltiplos domínios; (ii) protocolos para gerenciamento de recursos e serviços para acesso remoto seguro, (iii) protocolos para solicitação de informação que fornecem informações e *status* dos recursos, organizações e serviços; (iv) serviço de gerenciamento de dados para localizar e transportar conjunto de dados entre sistemas de armazenamento e aplicações.

Pode-se identificar três camadas distintas, que se comportam como um sistema computacional único e bem integrado, constituindo assim um ambiente de *grid* [8]:

- Infra-estrutura: componentes de *software* e *hardware*, integrados por uma rede física;
- *Middleware*: camada que oferece transparência de acesso aos recursos disponíveis, com ferramentas para o gerenciamento e o controle da infra-estrutura e das aplicações em *grid*,
- Aplicações: programas desenvolvidos e otimizados para tirar vantagem dos recursos distribuídos e do comportamento dinâmico do *grid*.

O próximo item desse capítulo descreve um *middleware* para *grid*, de código aberto, desenvolvido e gerenciado pelo Laboratório Nacional de *Argonne*, da Universidade do Sul da Califórnia e pela Universidade de Chicago.

## 2.2 *Globus Toolkit* 4.0

Falar sobre *Globus* significa descrever um *middleware* composto por vários serviços baseados em protocolos para informação de serviços, gerenciamento de processos, transferência de dados e segurança de acesso, conforme ilustra a Figura 2.1.

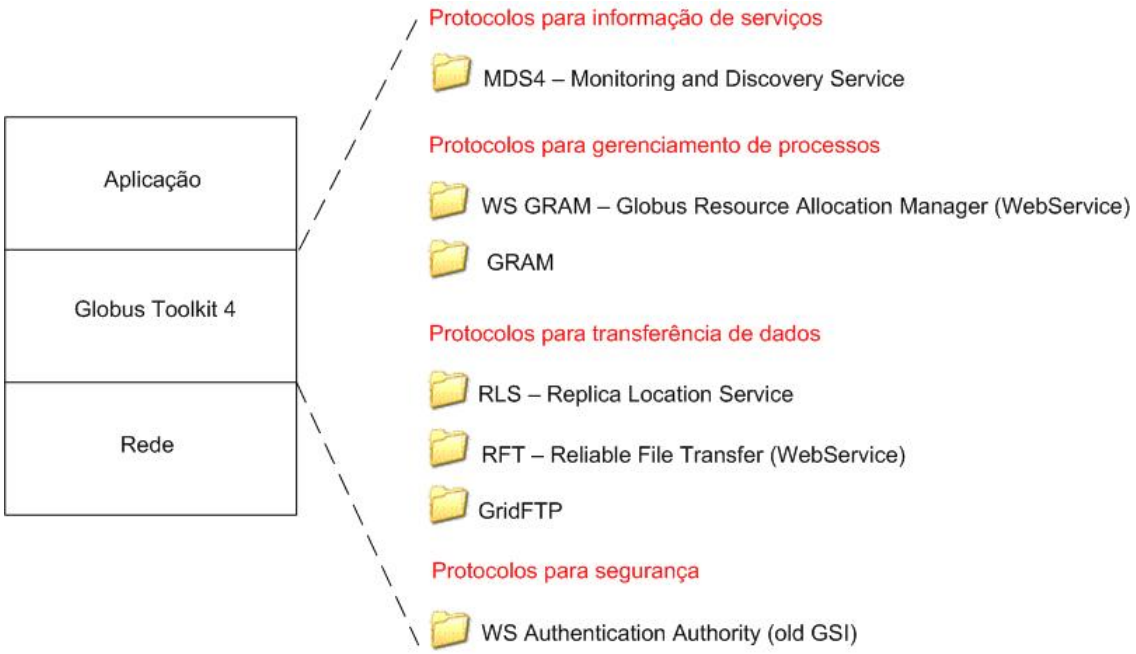



Figura 2.1: Protocolos que compõem o *middleware Globus*

Toda essa estrutura de protocolos atua sobre uma Organização Virtual (*Virtual Organization* - VO), a qual tem seus limites definidos pela presença de certificados digitais X.509, tanto no nível do usuário quanto no nível dos recursos computacionais como demonstra a Figura 2.2.

O processo de autenticação, tanto dos usuários como dos recursos, é gerenciado pelo serviço denominado Pre-WS *Authentication Authority* (antigo GSI - *Globus Security Infraestructure*). As principais motivações por trás desse serviço são: (i) a necessidade de comunicação segura (autenticação e talvez confidencialidade) entre elementos de uma Organização Virtual, (ii) a necessidade de implementar um sistema de segurança não centralizado através de diversos domínios institucionais, e (iii) a necessidade de suportar um *login* único para os usuários da Organização Virtual, incluindo delegações de credenciais para computações que envolvem múltiplos recursos ou *sites* [9].

Baseado nesse sistema de autenticação, cada Organização Virtual cria uma Autoridade Certificadora (*Certification Authority* - CA). Esta, por sua vez, gera um pacote contendo sua chave pública e outros aplicativos necessários aos procedimentos de autenticação. A instalação desse pacote e a presença de um certificado devidamente assinado pela CA são os requisitos necessários para que um recurso computacional pertença a uma Organização Virtual.

Da mesma forma, para que um usuário utilize os recursos disponíveis em uma Organização Virtual, o mesmo deve ter acesso a um dos recursos disponíveis. Através desse recurso, ele

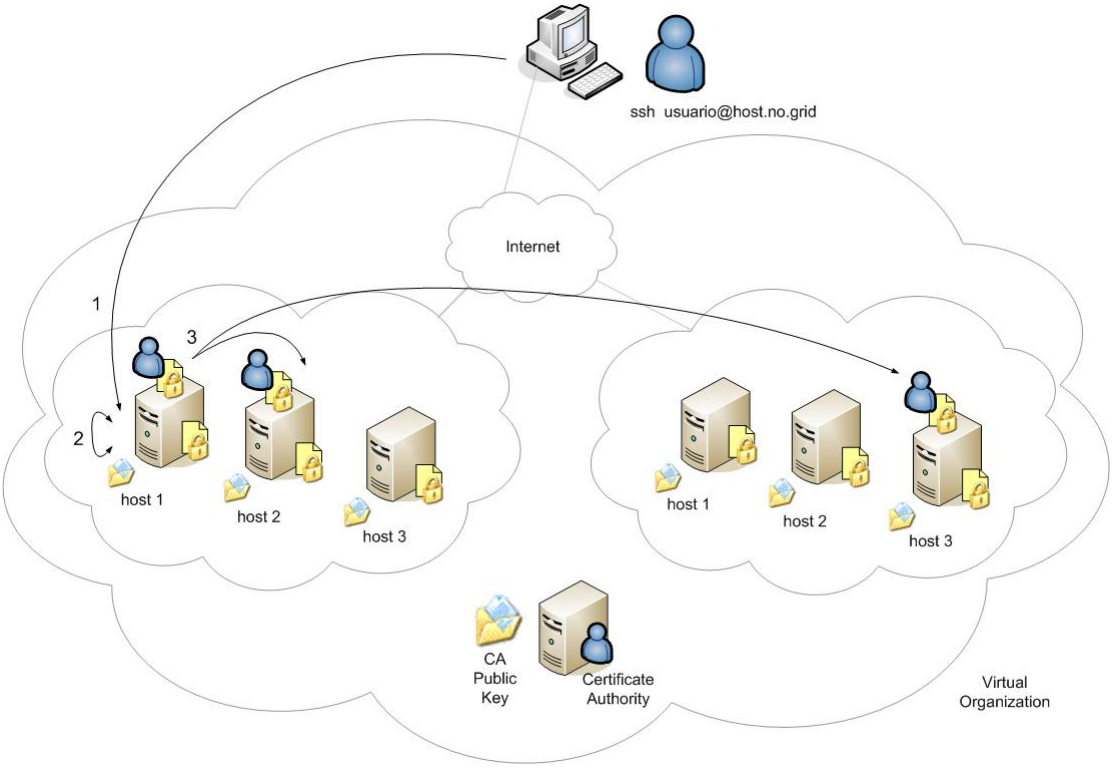



Figura 2.2: Estrutura de uma Organização Virtual

poderá gerar um certificado e enviá-lo à CA para que seja assinado. Com esse certificado devidamente assinado, o usuário envia um *email* aos administradores dos recursos disponíveis na Organização Virtual, solicitando acesso. No caso de aceitação, o certificado desse usuário é armazenado em um arquivo denominado *grid-mapfile* de cada recurso onde seu acesso foi autorizado.

O *login* único na Organização Virtual é obtido através de um certificado *proxy*. Este certificado é criado com o comando *grid-proxy-init* e será assinado pelo usuário. Um certificado de *proxy* também representa um certificado de sessão com tempo de vida limitado (geralmente 24 horas).

Uma vez criado o certificado *proxy*, o mesmo é usado para autenticar o usuário em qualquer recurso da Organização Virtual desde que ele tenha permissão de acesso. Esse modelo funciona pois cria uma hierarquia de certificados confiáveis, como ilustra a Figura 2.3.

Essa hierarquia de certificados pode ser descrita da seguinte forma [8]:

1. O recurso remoto do *grid* confia na CA porque instalou o pacote de autenticação cedido por ela e que contém a chave pública da CA.

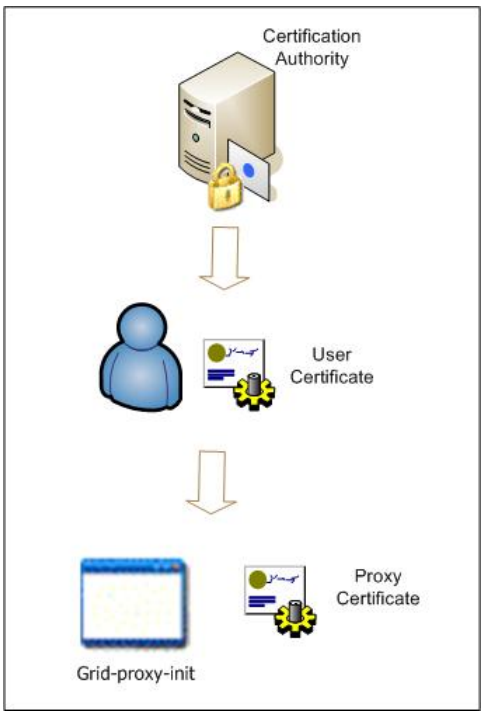



Figura 2.3: Hierarquia de certificados digitais

2. O recurso remoto do *grid* pode autenticar o usuário certificado porque o certificado dele é assinado digitalmente pela CA e ele têm a chave pública da CA.

3. O recurso remoto do *grid* pode autenticar o certificado *proxy* do usuário porque ele é assinado digitalmente pela chave privada do usuário e conforme descrito no passo 2, ele pode autenticar o usuário.

Após concluir o processo de autenticação, o usuário está apto a pesquisar quais são os recursos disponíveis na Organização Virtual, a solicitar permissão de acesso a esses recursos e a realizar a submissão, o monitoramento e o controle de *jobs* em todos os recursos da Organização Virtual para os quais ele conseguir tal permissão.

A descoberta de recursos pertencentes à Organização Virtual é realizada através do serviço de informação denominado MDS (*Monitoring and Discovering Service*). Este serviço fornece informações sobre os recursos que compõem a Organização Virtual (CPU, número de CPUs, memória real, memória virtual, sistema de arquivos, rede, etc) e sobre o estado de cada recurso (carga, disponibilidade, etc).

O conjunto de serviços denominado *Grid Resource Allocation and Management* (GRAM) fornece uma interface padrão para a solicitação e uso dos recursos dos sistemas remotos para a

execução de *jobs*. O uso mais comum do serviço GRAM é para a submissão e para o controle remoto de *jobs* [10].

Esse conjunto de serviços reduz o número de mecanismos necessários para a utilização de recursos remotos. Sistemas locais podem usar uma variedade de mecanismos de gerenciamento (escalonadores, sistemas de filas, sistemas de reserva e controle de interfaces), mas os usuários e os programadores precisam aprender apenas como utilizar um (GRAM) para solicitar e usar esses recursos. Pode-se comparar essa estrutura ao funcionamento de uma ampulheta: o conjunto de serviços GRAM pode ser representado como sendo o "pescoço" da ampulheta, com aplicações e serviços de alto-nível sobre ele e controles locais e mecanismos de acesso abaixo dele, como ilustra a Figura 2.4.


Figura 2.4: GRAM integra solicitações externas com ambiente interno de um recurso do *grid*

Como ambos os lados necessitam trabalhar apenas com o GRAM, o número de interações, APIs e protocolos que necessitam ser usados é bem reduzido.

A submissão de um *job* têm início com o GRAM *Client*, que pode ser um escalonador de aplicação ou o próprio usuário. Nesse instante, uma requisição em formato RSL (*Resource Specification Language*) é enviada ao serviço *gatekeeper* conforme ilustra a Figura 2.5 [10], no passo 3.

O arquivo RSL nada mais é do que uma linguagem utilizada para descrever recursos. Os vários componentes do protocolo de gerenciamento de processos manipulam *strings* RSL para desenvolver suas funções de gerenciamento em cooperação com os outros componentes do sistema. Cada atributo em uma descrição de recursos serve como parâmetro para controlar o comportamento de um ou mais componentes no sistema de gerenciamento de processos [11]. Um exemplo de arquivo RSL é ilustrado na Tabela 2.1. Nesse caso, serão criadas de 5 a 10 instâncias do código *myprog*, cada um em uma máquina com no mínimo 64 MB de memória que esteja disponível durante 4 horas.

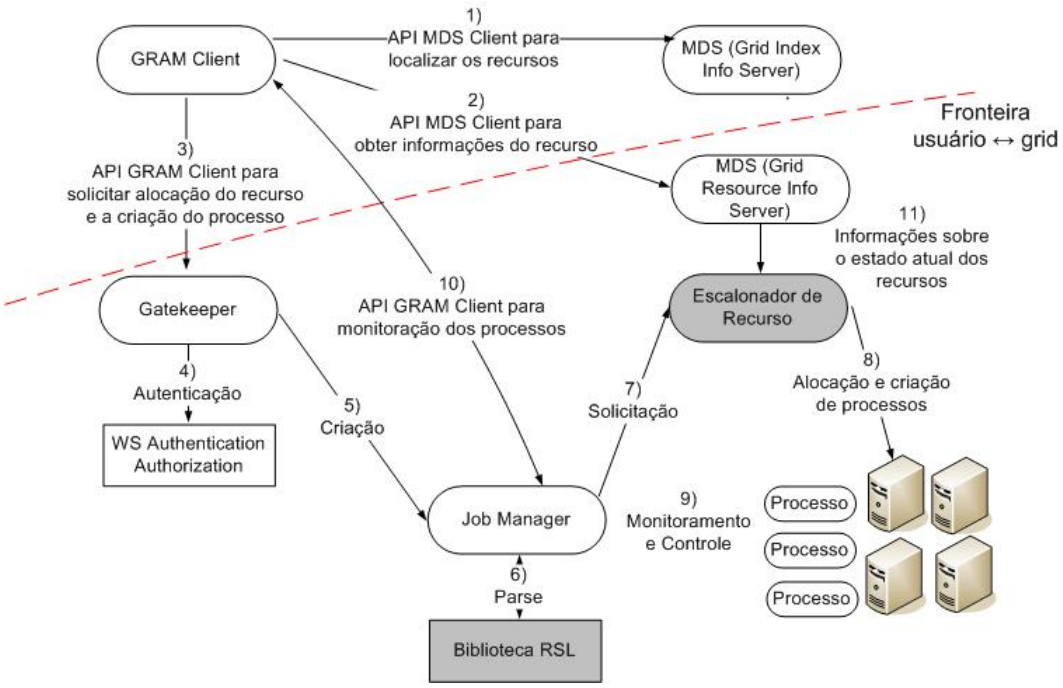



Figura 2.5: Interação entre o serviço GRAM (*Globus*) e o escalonador de recurso

Tabela 2.1: Arquivo RSL para submissão no Globus

```
&(count>=5)(count<=10)
(max_time=240)(memory>=64)
(executable=myprog)
```

Ao receber um arquivo RSL, o serviço *gatekeeper* realiza a autenticação do usuário através de uma requisição ao serviço Pre-WS *Authentication Authority*. Caso o usuário tenha permissão, uma submissão de tarefa cria um *Job Manager* que será responsável por iniciar e monitorar a tarefa. Requisições sobre o estado da tarefa serão encaminhadas diretamente ao *Job Manager*, como ilustra a Figura 2.5 nos passos de 5 a 9.

O *Job Manager* também é responsável por converter a requisição de formato RSL em um formato que o escalonador de recurso local possa entender. Atualmente, há versões de *Job manager* que interagem com Condor, NQE, COdine, EASY. LSF, LoadLeveler, PBS, Unix e Windows [12].

Por fim, o componente *Grid Resource Info Server* do serviço MDS obtém informações do estado e da carga da máquina junto ao escalonador de recursos local. O serviço MDS, por sua vez, torna estas informações disponíveis sob demanda para os outros componentes da

arquitetura *Globus*. Por exemplo, um escalonador de aplicação pode basear-se nas informações fornecidas pelo MDS para decidir quais recursos utilizar na execução de uma dada aplicação.

A ferramenta *Globus* também dispõe de componentes para o gerenciamento de dados e estes podem ser classificados em duas categorias: movimentação de dados e replicação de dados.

Na movimentação de dados há dois componentes: a ferramenta *GridFTP* e o serviço *Reliable File Transfer* (RTF) [13].

Na replicação de dados, o serviço de gerenciamento de dados do middleware *Globus* disponibiliza o *Replica Location Service* (RLS). RLS é uma ferramenta que tem a capacidade de manter uma ou mais cópias de arquivos em uma Organização Virtual. Essa ferramenta é especialmente útil para usuários que necessitam localizar onde determinados arquivos estão armazenados [13].

Atualmente, o *middleware Globus* é composto por componentes *web services* e por componentes não *web services*, como ilustra a Figura 2.6.

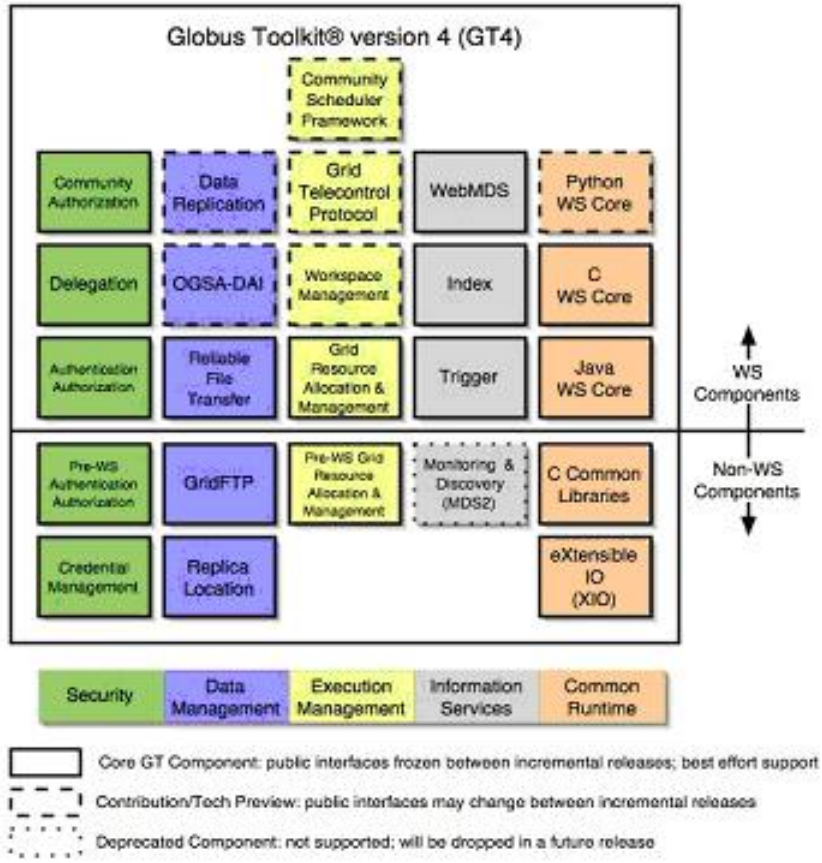



Figura 2.6: Componentes do *Globus Toolkit* 4

# 3. Escalonamento em *grids* computacionais

Em *grids* computacionais, o processo de escalonamento pode ser decomposto em nível de aplicação e em nível de recursos. O nível de aplicação pode utilizar técnicas de escalonamento estático, dinâmico ou híbrido, e algoritmos heurísticos com o objetivo de minimizar o tempo de processamento de uma aplicação. O nível de recurso utiliza especificações de hardware que são necessárias para a execução de uma determinada aplicação.

## 3.1 Considerações Iniciais

A utilização dos melhores recursos que um usuário tem à sua disposição para a execução de uma tarefa constitui um dos grandes desafios na área de *grids* computacionais. A solução para essa questão é obtida através das técnicas de escalonamento.

Geralmente, um recurso computacional é associado a um mecanismo de escalonamento que trabalha com prioridades e filas a fim de compartilhar o tempo de utilização de seus processadores. Além de submeter-se a esse sistema de escalonamento, algumas tarefas necessitam de uma determinada quantidade de memória e/ou recursos de entrada/saída. Apenas quando esses dois pré-requisitos são consentidos, o processador é liberado para a execução das tarefas.

Em ambientes de *grids* deve-se imaginar cada recurso disponível com o mesmo mecanismo de escalonamento acima citado. Aliados a esse mecanismo têm-se as regras que cada domínio institucional pode estabelecer quando disponibiliza um recurso.

Diante desse cenário pode-se compreender a complexidade de criação de um escalonador único capaz de gerenciar todos os recursos disponíveis em um *grid*.

Para resolver tal problema, o escalonamento nesses ambientes pode ser dividido em nível de aplicação e nível de recursos, compondo um sistema de gerenciamento de recursos com múltiplas camadas (*multi-layer Resource Management System* - RMS) [14], como ilustra a Figura

3.1.

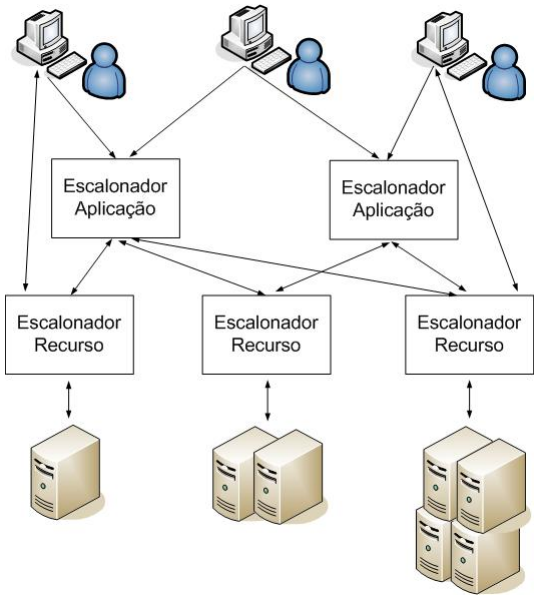



Figura 3.1: Escalonamento em *grids* computacionais

Um escalonador de aplicação aloca tarefas para os recursos do *grid* baseado em um modelo de aplicação, em um modelo arquitetural e em um critério de desempenho [5]. Esse tipo de escalonamento considera todas as relações de dependência existentes entre as tarefas. Um escalonador de recurso, por sua vez, baseia suas decisões de escalonamento apenas em pré-requisitos de *hardware* [8] tais como memória necessária, tipo de CPU, etc., mas não considera nenhuma depedência entre as tarefas. De maneira geral, trata da ordenação e do compartilhamento do uso do recurso ao qual está associado.

## 3.2 Escalonador de Aplicação

Supondo um *grid* computacional com uma estrutura simples e bem conhecida pelo usuário, a tarefa de gerar uma descrição dos recursos mais adequados a uma dada aplicação não é uma atividade tão complexa. Diante desse cenário, classifica-se o usuário como sendo o escalonador de aplicação.

Em um cenário mais complexo, onde o usuário dispõe de dezenas de recursos, escolher os melhores dentre eles pode tornar-se uma tarefa não trivial. Uma das soluções para esse caso é a utilização da técnica de escalonamento com base em grafos acíclicos direcionados (GAD [3, 4]), também denominada *task graph*, que permite implementar um escalonador de aplicação

estático.

Esse tipo de escalonamento envolve um modelo de aplicação, um modelo de ambiente computacional de destino (modelo arquitetural) e um critério de desempenho, como ilustra a Figura 3.2.

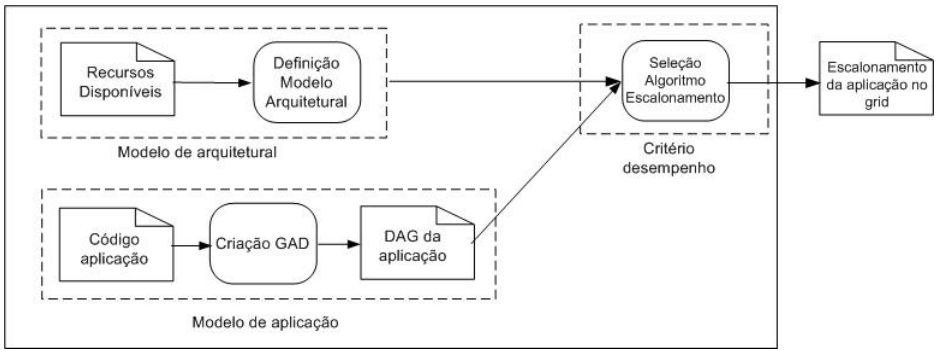



Figura 3.2: Estrutura de um escalonador de aplicação

O relacionamento entre o modelo de aplicação e o modelo arquitetural é definido pelo critério de desempenho, isto é, um algoritmo que relaciona recursos e tarefas.

### 3.2.1 Modelo de aplicação

Quando as caraterísticas de uma aplicação, as quais incluem tempo de execução das tarefas, quantidade de dados transferidos entre essas tarefas e, devido a essa comunicação, as dependências entre elas, são conhecidas a priori, ela pode ser representada por um modelo estático [15].

Esse modelo pode ser um grafo acíclico direcionado, ou GAD, denotado por $G=(V,E,\varepsilon,\omega)$, onde $V$ é o conjunto de $n$ nós do grafo e $E$ é o conjunto de arcos. Cada nó $v \in V$ representa uma tarefa com tempo de execução $\varepsilon(v)$ e cada arco $(i,j) \in E$ representa a restrição de dependência entre as tarefas $(v_i,v_j)$. Um peso $\omega_{(i,j)}$ pode estar associado ao arco $(i,j)$, representando a quantidade de dados a serem enviados da tarefa $v_i$ para a tarefa $v_j$, como ilustra a Figura 3.3.

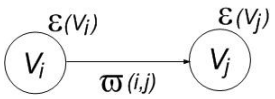



Figura 3.3: Modelo de aplicação

Em um GAD, um nó que representa uma tarefa sem predecessores é chamada tarefa de entrada e uma tarefa sem sucessores é chamada tarefa de saída. Alguns dos algoritmos de

escalonamento necessitam de grafos com uma única entrada ou uma única saída. Se há mais do que uma tarefa de saída (ou entrada), elas podem ser conectadas em uma pseudo tarefa de saída (ou entrada) com arcos de custo zero, sem afetar o escalonamento [15].

Baseado nesse modelo, um escalonador de aplicação pode determinar as dependências existentes entre as tarefas, o tempo de execução previsto para cada tarefa e para a aplicação como um todo, e o montante de comunicação existente entre as tarefas.

Essas informações, conciliadas com os dados de um modelo arquitetural, permitem definir de forma mais precisa a relação entre as tarefas de uma aplicação e os recursos disponíveis no *grid.*

### 3.2.2 Modelo arquitetural

Considerando *grid* computacional como um ambiente de execução, é possível definir um modelo arquitetural para esse ambiente através de (i) um conjunto *P* que representa um número limitado de processadores heterogêneos, (ii) uma matriz de dimensão *pxp*, utilizada para armazenar as taxas de transferências de dados entre os processadores, e (iii) um vetor *h* com os fatores de heterogeneidade dos processadores.

Esse modelo arquitetural também é composto por um modelo de comunicação que representa as características de comunicação dos nós existentes de uma forma mais precisa e realista.

Há vários modelos de comunicação, incluindo LogP [16], HlogP [17], LogGP [18], LogGPC [19], cada um explorando diversos fatores e características da arquitetura e, particularmente, da comunicação.

O modelo LogP é baseado no fato de que, nas máquinas paralelas atuais, o processador deve tratar, ou ao menos iniciar, cada comunicação e, além disso, comunicação e computação não podem ser sobrepostas totalmente. O nome dado para o modelo define os parâmetros considerados, ou seja [16]:

L - latência de comunicação;

o - sobrecarga (*overhead*), que é o tempo durante o qual o processador permanece preparando o recebimento (sobrecarga de recebimento) ou o envio (sobrecarga de envio) de uma mensagem, tempo esse durante o qual o processador não pode realizar outras operações;

g - *gap* é o intervalo mínimo necessário entre as transmissões ou recebimentos de men-

sagens consecutivas no processador,

P - o conjunto dos processadores disponíveis.

Sempre que uma tarefa $v_i \in V$ for escalonada num processador $p_j \in P$ no qual um de seus predecessores imediatos não se encontra, duas tarefas extras deverão ser escalonadas: (i) uma tarefa de envio e (ii) uma tarefa de recebimento, como ilustra a Figura 3.4. Uma tarefa de envio deve ser escalonada imediatamente após cada predecessor de $v_i$ que não se encontra em $p_j$. Para cada uma dessas tarefas de envio deve estar associada uma tarefa de recebimento, que será escalonada em $p_j$ antes de $v_i$. Tais tarefas representam a sobrecarga gasta pelo processador para o envio ou o recebimento de uma mensagem. Alguns autores preferem utilizar os termos tempo de empacotamento e tempo de desempacotamento para se referirem ao tempo de preparação do envio e do recebimento de uma mensagem, respectivamente [20].

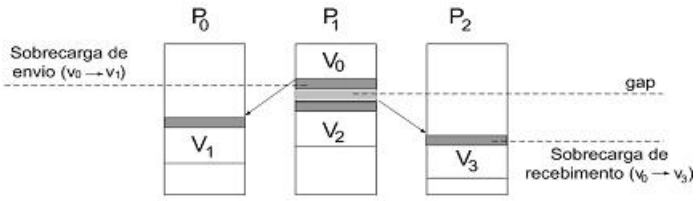



Figura 3.4: Escalonamento utilizando o modelo LogP

Uma vez definidos os modelos e estabelecidos os critérios sob os quais se realizará o escalonamento, é possível determinar e selecionar o melhor *makespan*, ou tempo de processamento, para uma dada tarefa. Se o modelo de comunicação adotado for o LogP, o *makespan* pode ser representado pela fórmula $maxST(v_i,P_j) + \varepsilon(v_i)\ x\ h(P_j)\ \forall\ v_i \in V$ [21].

Nesse caso, $ST(v_i,P_j)$ é o tempo de início de uma determinada tarefa $v_i$ em um processador $P_j$. $\varepsilon(v_i)$ é o peso computacional da tarefa $v_i$ e $h(P_j)$ é o grau de heterogeneidade do processador $P_j$. O objetivo das heurísticas de escalonamento é minimizar esse tempo de processamento.

### 3.2.3 Critério de desempenho

O problema de escalonamento utilizando GAD é classificado como NP-completo [1] em sua forma geral, sendo que soluções com tempo polinomial são conhecidas apenas para um número restrito de casos. Devido a essa complexidade e à impossibilidade de obter todas as soluções

[1]NP é o acrônimo em inglês para tempo polinomial não determinístico (*Non-Deterministic Polynomial time*). É um conjunto de problemas que requerem tempo polinomial para verificação por uma máquina de *Turing* determinística.

possíveis, algoritmos de escalonamento estáticos podem ser divididos em dois grupos principais: (i) baseados em heurísticas e (ii) baseados em pesquisas aleatórias direcionadas, como ilustra a Figura 3.5 [15].

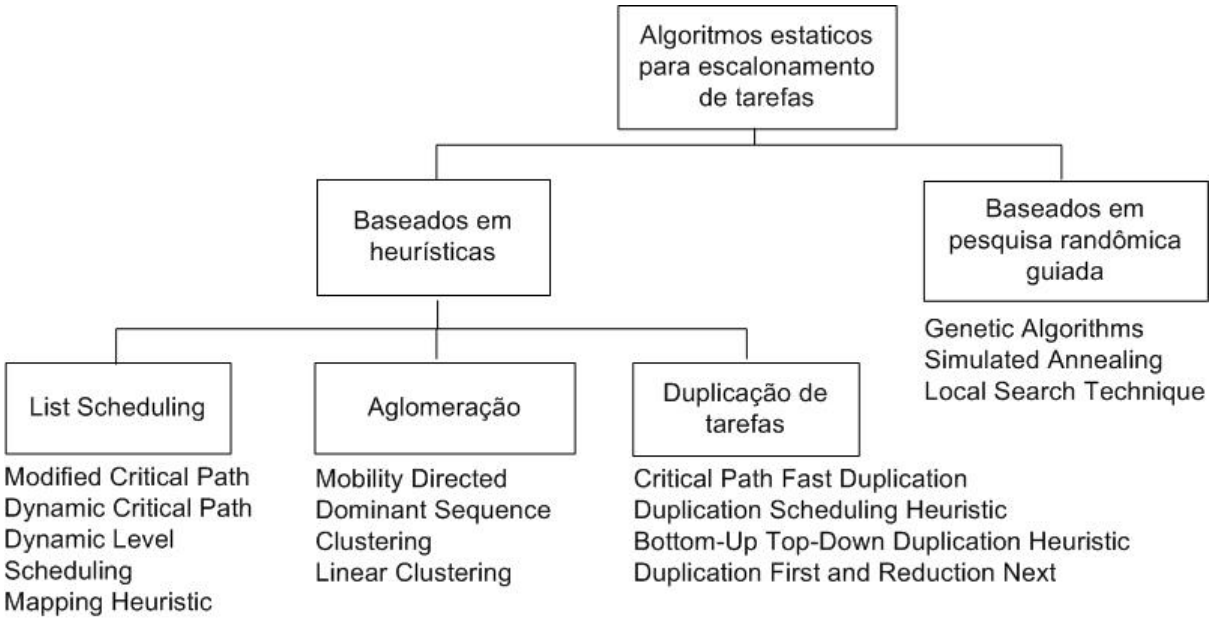



Figura 3.5: Classificação dos algoritmos de escalonamento estático

Os algoritmos baseados em heurísticas podem ser classificados em três grupos: *list-scheduling*, aglomeração e duplicação de tarefas [15]

Estes algorimos baseados em heurísticas mantêm uma lista de todas as tarefas de um grafo de acordo com suas prioridades e implementam duas fases: a priorização das tarefas, que é a fase para selecionar a tarefa com a mais alta prioridade, e a fase de seleção do processador, para selecionar o processador que irá minimizar a função pré-definida de custo de execução. Essa função pode ser o início do tempo de execução, como mencionado na fórmula do *makespan* apresentada anteriormente *($ST(v_i,P_j)$)*.

Há, contudo, numerosas variações nos métodos utilizados para priorizar as tarefas e para selecionar os processadores. Algumas dessas variações são [22]:

- Atribuição de prioridades aos nós: utiliza os atributos de *top-level* e *bottom-level*. O *top-level* de um nó $n_i$ é o comprimento do caminho mais longo de um nó de entrada para o nó $n_i$, excluindo $n_i$. Esse comprimento é a soma de todos os pesos dos nós e arestas ao longo desse caminho. O *bottom-level* de um nó é o comprimento do caminho mais longo desse nó $n_i$ para um nó de saída. Devido ao fato das prioridades no *bottom-level* serem

computadas recursivamente, através do percurso do grafo no sentido *upward*, iniciando da tarefa de saída e finalizando no nó $n_i$, ele também é denominado *upward rank* ou *rank up*. De forma análoga, define-se o termo *downward rank*, ou *rank down*, para o cálculo de prioridades usando *top-level*, onde o grafo é percorrido recursivamente iniciando no nó de entrada e seguindo até o nó $n_i$.

- Inserção ou não-inserção: considera a possível inserção de uma tarefa $n_i$ em um processador $p$ próximo ao tempo final de uma tarefa $n_j$ já escalonada nesse processador $p$.

- Caminho crítico: algoritmos baseados em caminho crítico atribuem maior prioridade às tarefas que pertecem ao caminho crítico de um GAD, isto é, tarefas que estão no caminho mais longo entre o nó de entrada e o nó de saída.

Alguns dos exemplos de algoritmos que utilizam heurísticas de *list-scheduling* são o *Modified Critical Path* (MCP), *Dynamic Critical Path* (DCP), *Dynamic Level Scheduling*, *Mapping Heuristic* (MH), *Insertion-Scheduling Heuristic* e *Earliest Time First* (ETF) [15].

A maioria dos algoritmos *list-schedulig* são para um número limitado de processadores homogêneos totalmente conectados. Os algoritmos que suportam processadores heterogêneos são *Dynamic Level Scheduling Algorithm*, *Levelized-Min Time Algorithm* e o *Mapping Heuristic* [15].

Algoritmos que utilizam heurísticas *list-scheduling* realizam o processo de escalonamento em um tempo inferior quando comparados aos algoritmos baseados em heurísticas de aglomeração e ou de duplicação de tarefas.

Nos algoritmos baseados em aglomeração de tarefas, uma coleção de tarefas, ou *cluster*, consiste em um conjunto de tarefas que devem ser executadas em um mesmo processador. As tarefas são aglomeradas em uma coleção visando minimizar o tempo de execução da aplicação através da minimização do custo de comunicação e do tempo de computação gasto por um processador. Isto é possível pois a comunicação entre as tarefas pertencentes a uma mesma coleção tem custo nulo [20]. Exemplos de algoritmos de aglomeração são *Dominant Sequence Clustering* (DSC), *Linear Clustering Method*, *Mobility Directed* e *Clustering and Scheduling System* (CASS) [15].

Finalizando os algoritmos baseados em heurísticas, a idéia por trás dos algoritmos de escalonamento baseados na duplicação de tarefas é escalonar as tarefas de um grafo e mapear algumas dessas tarefas redundantemente, o que reduz o *overhead* de comunicação entre os

processos. Algoritmos baseados em duplicação diferem de acordo com a estratégia de seleção das tarefas que serão duplicadas. Os algoritmos nesse grupo são geralmente para um número ilimitado de processadores idênticos e eles apresentam uma complexidade muito maior do que os algoritmos baseados em *list-scheduling* ou aglomeração.

Além dos algoritmos baseados em heurísticas têm-se os algoritmos baseados em técnicas de pesquisa aleatória orientada. Esses algoritmos utilizam escolhas aleatórias e estas orientam novas escolhas aleatórias através do contexto do problema. O desempenho meramente aleatório não caminha da mesma forma que os métodos aleatórios de pesquisas. Essa técnica combina o conhecimento adquirido de uma pesquisa anterior com algumas características aleatórias para gerar novos resultados. Algoritmos genéticos [2], que são representantes dessa técnica, resultam em uma boa qualidade de escalonamento, contudo, seus tempos de execução são geralmente muito maiores do que as técnicas baseadas em heurísticas.

## 3.3 Escalonador de recursos

As necessidades de uma tarefa em um *grid* computacional são determinadas pelo usuário no momento da submissão. Em ambientes mais complexos, podem ser tratadas pelo escalonador de aplicação como descrito anteriormente. Por outro lado, as restrições de uso de um recurso são determinadas pela instituição que o disponibilizou no *grid*. Por exemplo, uma instituição pode preferir que um recurso execute as aplicações do usuário *A*, seguidas das aplicações do grupo de sistemas operacionais, e que nunca execute as aplicações do usuário B. Ou seja, as restrições permitem à instituição determinar como sua máquina será utilizada. Essa é a principal função de um escalonador de recursos. Ele deve receber as solicitações de vários usuários e gerenciar o uso dos recursos que ele controla para atender essas solicitações. O escalonador de recursos decide quando e como cada processo executa.

A tabela 3.1 contém alguns pré-requisitos que o escalonador de recurso *Condor* [23] utiliza. Esse arquivo de submissão solicita a execução do programa *myprog* 150 vezes. Esse programa foi compilado e teve o código gerado para estações *SiliconGraphics* executando o sistema operacional IRIX 6.5. Também é descrito que a execução do programa deve ocorrer em máquinas as quais devem ter mais do que 32 MB de memória, e é expressa a preferência para

[2]Os algoritmos genéticos são de uma família de modelos computacionais inspirados na evolução, que incorporam uma solução potencial para um problema específico numa estrutura semelhante à de um cromossomo e aplicam operadores de seleção e *cross-over* a essas estruturas de forma a preservar informações críticas relativas à solução do problema.

executar o programa em máquinas com mais de 64 MB, se tais máquinas estiverem disponíveis. Também é descrito que o programa irá utilizar 28 MB de memória durante sua execução. Cada uma das 150 execuções do programa irá receber seu próprio número de processo, iniciando com o processo de número 0. Dessa forma, os arquivos *stdin*, *stdout*, e *stderr* serão referenciados como in.0, out.0, e err.0 para a primeira execução do programa, in.1, out.1, e err.1 para a segunda execução do programa e assim sucessivamente. Um arquivo de *log*, identificado como *foo.log*, irá conter as informações sobre quando e onde o *Condor* executa os processos, realiza os *checkpoints* e a migração de processos, para as 150 execuções solicitadas.

Tabela 3.1: Arquivo de submissão no escalonador de recurso Condor

```
Executable = myprog
Requirements = Memory >= 32 && OpSys == "IRIX65"&& Arch =="SGI"
Rank = Memory >= 64
Image_Size = 28 Meg
Error = err.$(Process)
Input = in.$(Process)
Output = out.$(Process)
Log = foo.log
Queue 150
```

Em um ambiente de *grid* computacional baseado no *middleware Globus*, a submissão pode ser realizada a partir de um arquivo RSL, como foi mencionado no item 2.2. Sendo o escalonador de recurso específico a cada recurso ou a um conjunto de recursos que compõem o *grid*, nesse ambiente, o serviço GRAM é responsável por realizar a interface entre o escalonador de aplicação e o escalonador de recurso (*Condor, NQE, Condine, EASY, LSF, LoadLeveler, PBS* [8]), ou seja, realizar a conversão de um arquivo RSL em dados que um escalonador de recurso entenda, como ilustra a Figura 2.5 nos passos 5, 6 e 7.

# 4. Simulação de ambientes de *grids* computacionais

A análise da escalabilidade de algoritmos para *grids*, bem como a criação de um ambiente repetitivo e em condições de heterogeneidade que extrapolam as condições existentes, podem normalmente ser tratadas com o auxílio da simulação com o objetivo de realizar testes mais adequados a uma determinada necessidade.

## 4.1 Considerações Iniciais

Obter resultados de técnicas de escalonamento em ambientes reais de *grids* computacionais não é trivial. A limitação desse ambiente e de sua heterogeniedade, características necessárias para a realização de testes consistentes, são os fatores mais agravantes. Uma forma de análise empregada para superar essas limitações é o uso de simuladores que são ferramentas capazes de representar aspectos de uma situação ou de um processo de forma realista.

A motivação para o uso de simuladores, especialmente para a análise de algoritmos e modelos em fase de implementação, pode ser baseada nos seguintes fatores [24]:

- Configurar um ambiente de desenvolvimento para *grid* é caro, envolve muitos recursos e consumo de tempo. Mesmo que um ambiente assim seja configurado, ele geralmente é limitado a uma área física restrita.
- O uso de um ambiente de desenvolvimento real poderá gerar custos reais pois, geralmente, os recursos disponíveis em *grids* trabalham sobre um modelo de economia. Como a análise de algoritmos e modelos novos necessitam de um grande número de testes envolvendo o maior número de recursos disponíveis, o custo desses testes poderá tornar a modelagem desses algoritmos e modelos inviável.
- O uso de um ambiente de desenvolvimento real com tarefas reais implica em tempo de execução real. Horas de execução dessas tarefas podem ser simuladas em segundos por

um simulador que esteja executando em um recurso com poder computacional suficiente para isso.

- Um ambiente de desenvolvimento real não provê um ambiente repetitivo e controlável para a experimentação e avaliação de estratégias de escalonamento.

Há uma diversidade de simuladores de *grids* disponíveis na *Internet*. Dentre eles, destacam-se: *OptorSim* [25], *GridNet* [26], *Bricks* [27], *MicroGrid* [28], *Simgrid* [29], e *GridSim* [24].

Dentre os simuladores mencionados, o *GridSim* mostra-se como a plataforma mais adequada do ponto de vista da flexibilidade. Por ser escrito em *Java* (usando como base o *SimJava*), o *GridSim* oferece portabilidade e a vantagem de trabalhar com múltiplas *threads*, permitindo que diversas entidades "usuários" submetam tarefas para execução simultânea. Sua documentação está organizada em dez programas exemplos, que acompanham o código do simulador, e na documentação da API. Suas principais características são [24]:

- Permite a modelagem de tipos heterogêneos de recursos;
- Os recursos podem ser modelados para operar em modo *space-sharing* ou *time-sharing*;
- A capacidade dos recursos pode ser definida em MIPS (*Million Instrucions per Second*) ou baseadas no *benchmark* SPEC (*Standard Performance Evaluation Corporation*);
- Recursos podem ser localizados em diferentes *time zones*;
- Finais de semanas e feriados podem ser mapeados dependendo da localização do recursos para modelar a carga de utilização do recurso;
- Aplicações com diferentes modelos de paralelismo podem ser simuladas;
- O número de tarefas de uma aplicação, que podem ser submetidas à um recurso, não é limitado;
- Múltiplas entidades "usuários" podem submeter tarefas para execução simultaneamente em um mesmo recurso;
- A velocidade de rede entre os recursos pode ser especificada;
- Suporta a simulação de escalonadores estáticos e dinâmicos,
- Estatísticas de todas as operações, ou apenas das selecionadas, podem ser registradas.

Esse simulador provê alta extensibilidade e portabilidade através das tecnologias *Java* e *thread*. Todos os componentes no modelo de sistema são iniciados como uma *thread* com um nome único. Cada componente executa individualmente com filas de eventos separadas e um evento é transmitido para a fila de eventos do componente de destino diretamente.

Embora muito flexível, o *GridSim* não é escalonável, uma vez que ele depende de um número de *threads* que é bem limitado. Em adicional, o gerenciamento de *threads* em *Java* cria um *overhead* consideravél que pode resultar em um tempo de execução muito grande [30].

## 4.2 GridSim: Criação de tarefas e usuários

O *GridSim* não define nenhum modelo de aplicação específico. Isso fica a critério do desenvolvedor que utiliza o *GridSim* para realizar seus testes.

Esse simulador tem sido testado com o modelo de aplicação *task-farming* [1] e seus desenvolvedores acreditam que outros modelos de aplicações paralelas como GADs, dividir para conquistar, etc, também podem ser modelados e simulados usando *GridSim* [24].

Nesse simulador, cada tarefa independente pode ser heterogênea em termos de tempo de processamento e tamanhos de arquivo de entrada e de saída. Tais tarefas podem ser criadas e suas necessidades podem ser definidas através do objeto *Gridlet*.

Um *Gridlet* é uma pequena aplicação para *grid* que contém todas as informações relativas a uma tarefa. Também descreve os detalhes de gerenciamento da execução dessa tarefa como o custo de processamento, expresso em MIPS, o tamanho do arquivo de entrada, etc. Essas informações auxiliam no cálculo do tempo de execução dessa aplicação em um recurso remoto.

Devido ao fato do *GridSim* simular ambientes multi-usuários, cada usuário define as tarefas que irão compor sua aplicação.

Observando o código da Tabela 4.1, vê-se que deve-se definir o usuário como sendo uma máquina conectada ao *grid*. Deve ser especificada a taxa de transmissão do *link* de comunicação que o usuário tem com o *grid* bem como o *delay* e o MTU desse *link*.

As tarefas que compõem a aplicação de um usuário podem ser definidas como descrito na Tabela 4.2. Cria-se uma instância do objeto *Gridlet* atribuindo a ela uma identificação do

[1] Aplicações paralelas *task-farming* são aquelas em que um conjunto de dados de entrada deve ser processado independentemente por diversas tarefas executando o mesmo código. Cada tarefa eventualmente gera um resultado, que pode ser usado independentemente dos outros

Tabela 4.1: Trecho da classe *Program* onde o usuário é criado

```
...
//////////////////////////////////////////
// Third step: Creates one or more grid user entities
System.out.println(-----------------------------------------");
System.out.println("Creating Grid Users");
System.out.println(-----------------------------------------");
// number of Gridlets that will be sent to the resource
int totalGridlet = 4;
baud_rate = 1000000000; // 1 Gbits/sec
double propDelay = 0.23; // propagation delay in millisecond
int mtu = 1500; // max. transmission unit in byte

// create users
ArrayList userList = new ArrayList(num_user);
    NetUser user = new NetUser("User_0", totalGridlet, baud_rate, propDelay, mtu,trace_flag);

// add a user into a list
userList.add(user);
...
}
```

usuário, o custo de computação dessa tarefa e o tamanho dos dados de entrada e de saída.

Tabela 4.2: Trecho do método *createGridlet* onde a tarefa 0 de uma aplicação é criada

```
/**
This method will show you how to create Gridlets
@param userID owner ID of a Gridlet
@param numGridlet number of Gridlet to be created
/
private void createGridlet(int userID, int numGridlet)
{
// Creates a Gridlet
// Gridlet(int id, double length, long file_size, long output_size)
Gridlet gl = new Gridlet(0, 7536826, 0, 0);

// setting the owner of these Gridlets
gl.setUserID(userID);

// add this gridlet into a list
this.list_.add(gl);
...
```

Dessa forma, os tempos de processamento de aplicações *task-farming* podem ser estimados para qualquer recurso disponível no *grid*.

## 4.3 GridSim: Criação de recursos

No *toolkit GridSim*, pode-se criar CPUs (também denominadas *Processing Elements* (PEs)) com diferentes capacidades especificadas em MIPS. Dessa forma, um ou mais PEs podem ser colocados juntos para criar uma máquina (um único nó). Da mesma forma, uma ou mais

máquinas podem ser colocadas juntas para criar um recurso do *grid*. O recurso resultante pode ser um único processador, um multiprocessador de memória compartilhada (SMP - *Shared Memory Processors*), ou um *cluster* de computadores com memória distribuída. Esses recursos do *grid* podem ser gerenciados por sistemas de escalonamento baseados em *time-sharing* [2] ou *space-sharing*, dependendo do tipo de recurso [24].

Para cada recurso do *grid*, a carga local pode ser estimada e o *time zone* do recurso definido. A velocidade da rede de comunicação entre o usuário e os recursos é definida pela taxa de transferência de dados.

Quando uma entidade é criada, essa entidade registra suas informações de recurso e detalhes de contato na entidade *Grid Information Service* (GIS) do simulador.

Considerando o ambiente de *grid* da Figura 4.1, a Tabela 4.3 exibe o código de como os recursos de A0 a A8 são criados. Primeiro, define-se a taxa de processamento em MIPS, o total de processadores que operam nessa taxa de processamento e o total de máquinas que terão processadores com essas características. Também define-se o canal de comunicação que esse recurso irá ter com o roteador A.

Outras definições do recurso, como tipo de sistema operacional, *time zone*, custo de computação desse recurso, se ele opera em *time sharing* ou *space sharing*, a carga local, o *delay* de comunicação, o calendário de dias úteis e feriados, são definidos no métdo *createGridResource()* que pertence à classe *Program* e se encontra no Anexo A desse trabalho.

Tabela 4.3: Trecho da classe *Program* que cria um recurso no simulador GridSim

```
// Creating NODE_A0 a NODE_A8 Resources
int rating = 6769; // rating of each PE in MIPS - Intel Core 2 X6800
int totalPE = 4; // total number of PEs for each Machine
int totalMachine = 1; // total number of Machines
double baud_rate = 1000000000; // bits per second

String resName = "A";
int i = 0;

NewGridResource resNode = null;

for(i = 0;i < 9;i++)
{
   resNode = createGridResource(resName+i, rating, totalMachine, totalPE, baud_rate);
   // add a resource name into an array
   resList.add(resNode);
}
```

[2] Geralmente, um único PE or um SMP é gerenciado por um sistema *time-sharing* usando política de escalonamento *round-robin* e um *cluster* é gerenciado por escalonadores baseados em *space-sharing* usando diferentes políticas de escalonamento, como *First-Come-First-Served* (FIFO), *Shortest-Job-First* (SJF), entre outras [24].

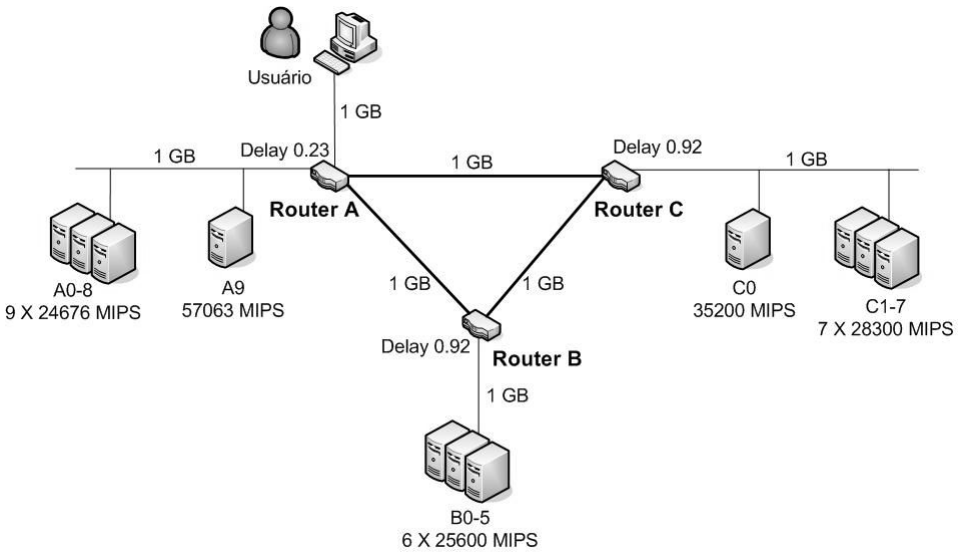



Figura 4.1: Ambiente de *grid* Simulado

Esse ambiente de *grid* também é composto por 3 roteadores que interligam as instituições A, B e C. A criação dessas entidades é realizada pelo objeto *Router*, como demonstra o conteúdo da Tabela 4.4.

Tabela 4.4: Trecho da classe *Program* onde os roteadores são criados

```
// create the routers.
Router r1 = new RIPRouter("router1", trace_flag); // router A
Router r2 = new RIPRouter("router2", trace_flag); // router B
Router r3 = new RIPRouter("router3", trace_flag); // router C
```

A conexão do usuário e dos recursos aos seus devidos roteadores inicia-se com a criação de um método de escalonamento (First In-First Out (FIFO) ou Self-Clocking Fair Queuing (SCFQ)), que será aplicado aos pacotes transferidos entre o roteador escolhido e a entidade conectada a ele, como descrito pelo código da Tabela 4.5. Após essa criação, utiliza-se o método *attachHost()* para conectar essas entidades.

O passo final para compor esse ambiente de *grid* é a interconexão dos roteadores através de uma rede *Gigabit*. Como descrito no conteúdo da Tabela 4.6, deve-se criar um *link* de comunicação entre essas duas entidades especificando a taxa de transmissão, o *delay* desse *link* e o MTU dos pacotes. Como citado anteriormente, também deve-se criar um método de escalonamento e, finalmente, através do método *attachRouter()* interconectar os roteadores.

Esse simulador não é apenas destinado para pesquisar tempo de execução de escalonadores, mas também para testar algoritmos de escalonamento direcionados pelos modelos de economia

Tabela 4.5: Trecho da classe *Program* onde o usuário e recursos são conectados a um roteador

```
// connect all USER entities with ROUTER 1
// For each host, specify which PacketScheduler entity to use.
NetUser obj = null;
for (i = 0; i ⟨ userList.size(); i++)
{
// A First In First Out Scheduler is being used here.
// SCFQScheduler can be used for more fairness
FIFOScheduler userSched = new FIFOScheduler("NetUserSched_"+i);
obj = (NetUser) userList.get(i);
r1.attachHost(obj, userSched);
}
System.out.println("User connected to R1");

// connect ALL RESOURCE from instituion A with ROUTER 1
// For each host, specify which PacketScheduler entity to use.
GridResource resObj = null;

for (i = 0; i ⟨ 10; i++)
{
FIFOScheduler resSched = new FIFOScheduler("GridResSched_"+i);
resObj = (GridResource) resList.get(i);
r1.attachHost(resObj, resSched);
}
System.out.println("Resources from Institution A connected to R1");
```

Tabela 4.6: Trecho da classe *Program* onde os roteadores são conectados entre si

```
// then connect ROUTER 1 to ROUTER 2 with 1 Gb/s connection
baud_rate = 1000000000; // 1 Gbits/s
Link link = new SimpleLink("r1_r2_link", baud_rate, propDelay*4, mtu);

//FIFOScheduler r1Sched = new FIFOScheduler("r1_Sched");
FIFOScheduler r1Sched = new FIFOScheduler("r1_Sched");
FIFOScheduler r2Sched = new FIFOScheduler("r2_Sched");

// attach Router 2 to Router 1
r1.attachRouter(r2, link, r1Sched, r2Sched);
```

baseados em mercado.

# 5. Escalonamento de aplicações MPI em *grids* computacionais

Para utilizar um escalonador de aplicação é necessário ter um modelo de aplicação, um modelo arquitetural e um algoritmo de escalonamento. Monitorando as execuções de aplicações em *cluster* é possível modelar uma aplicação através de um GAD que é utilizado pelos algoritmos de escalonamento para estabelecer as relações de dependência entre as tarefas de uma aplicação e para produzir um escalonamento mais eficiente em *grids* computacionais.

## 5.1 Considerações Iniciais

Modelar o comportamento de uma aplicação real em forma de GAD não é uma tarefa simples quando deve-se estimar o tempo de execução das tarefas e o peso das comunicações entre essas tarefas.

Consideremos que o desenvolvedor de uma aplicação modelou a mesma em forma de GAD, informando apenas as relações de dependências existentes entre as tarefas da aplicação. Sem os dados mencionados no parágrafo anterior, mesmo que o escalonador de aplicação saiba qual a relação de dependência existente entre as tarefas, ele não consegue prever o tempo de execução de uma tarefa em um processador selecionado. Também não consegue determinar qual é o impacto de um determinado canal de comunicação sobre o tempo de execução dessa tarefa.

Com o objetivo de automatizar a modelagem de aplicações MPI reais, para obter os dados necessários a um escalonador de aplicação e este, por sua vez, permitir a execução dessas aplicações de forma eficiente em *grids* computacionais, diversos mecanismos foram implementados nesse trabalho. Esses mecanismos podem ser organizado em 6 módulos, como ilustra a Figura 5.1.

Os módulos 1, 2, 3 e 4 definem o modelo de aplicação, o módulo 5 define o modelo arquitetural e o módulo 6 contém o algoritmo de escalonamento, que relaciona o modelo de aplicação com o modelo arquitetural.

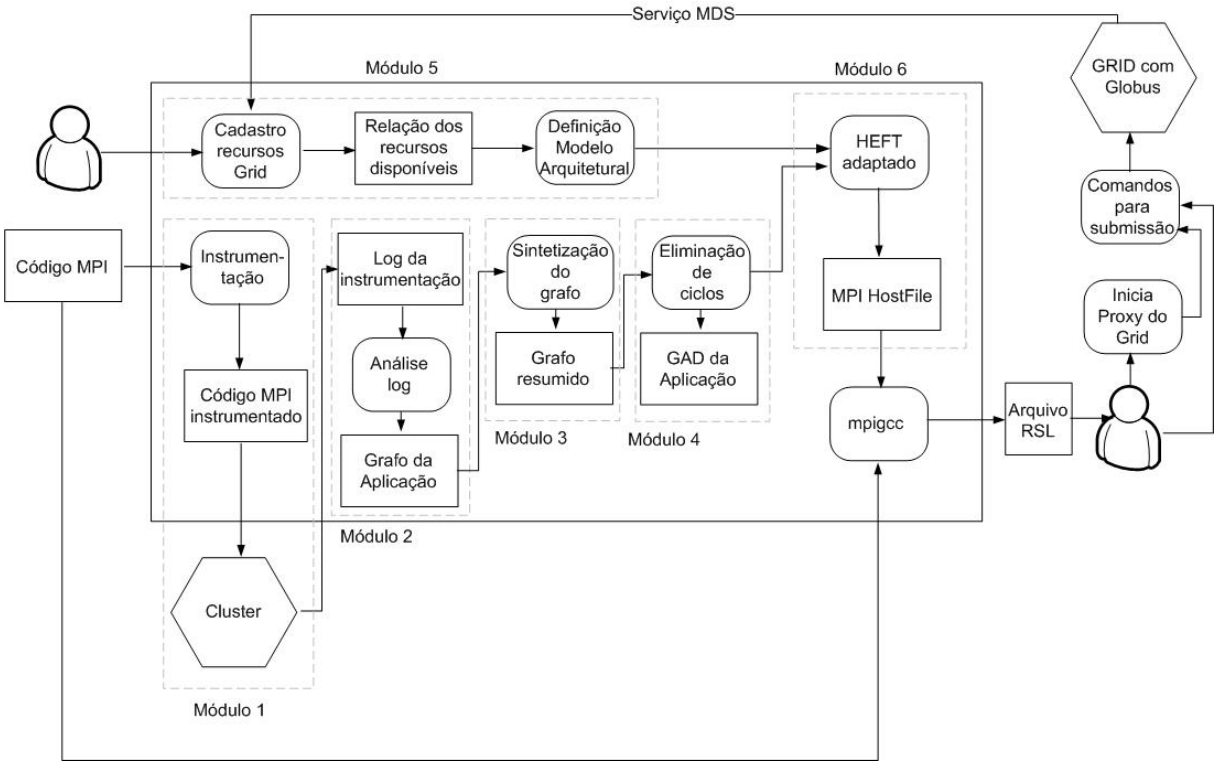



Figura 5.1: Ilustração dos mecanismos implementados nesse trabalho

De forma resumida, o módulo 1 realiza o monitoramento da execução de uma aplicação em um *cluster*. Os dados resultantes desse monitoramento serão utilizados pelos demais módulos para gerar um GAD da aplicação. Esse módulo é descrito no item 5.2.1.

No módulo 2 é realizada a análise dos dados coletados e no módulo 3 a sintetização desses dados. O resultado final desses módulos será um grafo cíclico, que ainda não é adequado aos algoritmos de escalonamento. Esses módulos são comentados nos itens 5.2.2 e 5.2.3 e os códigos implementados estão no Anexo B e no Anexo C, respectivamente.

Para transformar um grafo cíclico em acíclico foi implementado o módulo 4 que realiza uma busca em profundidade no grafo com o objetivo de identificar e remover os ciclos existentes entre os nós. Esse módulo é descrito no item 5.2.4 e o código implementado está no Anexo D.

O módulo 5 realiza a coleta dos recursos disponíveis no *grid*. A descrição mais detalhada desse módulo é realizada no item 5.3.2 e o código implementado está no Anexo E.

Finalizando, o módulo 6 executa o escalonamento como está descrito no item 5.3. No Anexo F estão os principais códigos implementados nesse módulo.

## 5.2 Geração de GAD

Escalonadores de aplicação baseados em grafos acíclicos direcionados necessitam de dados tais como o número de tarefas processadas, as comunicações entre essas tarefas e, em decorrência dessas comunicações, a dependência entre essas tarefas. Para o algoritmo, essas dependências determinam qual a sequência de escalonamento das tarefas é mais apropriada para uma dada aplicação [15].

A criação de um modelo realista do comportamento de uma aplicação pode ser complexa e geralmente recai sobre o desenvolvedor da aplicação.

Conhecimento sobre as características das aplicações pode geralmente ser obtido através de monitoramento. Tal monitoramento, o qual nesse trabalho é realizado durante a execução da aplicação em *cluster*, provê os dados acima mencionados além de dados complementares, tais como o custo computacional das tarefas (tempo total de processamento) e o peso de cada comunicação (tempo total da comunicação e a quantidade de dados transferida).

### 5.2.1 Monitoramento em *cluster*

A ferramenta *Intel Trace Collector* (ITC) gera arquivos de *log* (*tracefiles*) de aplicações MPI os quais podem ser interpretados pela ferramenta de análise de desempenho *Intel Trace Analyzer* (ITA). Algumas versões de ITC também são capazes de rastrear aplicações não MPI, como processos *Java* e comunicações *sockets* em aplicações distribuídas. Ela era anteriormente conhecida como *Vampirtrace* [31].

Devido ao fato do ITC ser uma extensão das implementações MPI existentes e explorar os recursos da MPI *Profiling Interface* [1], sua utilização em um nível básico, ou seja, o rastreamento de todas as chamadas das rotinas MPI e também de todas as comunicações ponto-a-ponto e coletivas, é possível apenas recompilando a aplicação com *link* para a biblioteca apropriada (libVT.a).

Uma das principais características do ITC é o fato do processo de rastreamento das informações não interferir no processamento da aplicação. Para isso, as informações coletadas são armazenadas em memória.

[1] MPI *Profiling Interface* fornece um encapsulamento para cada chamada da *interface*, com prefixo PMPI_ ao invés de MPI_. Dessa forma, é possível trocar qualquer subconjunto de chamadas da *interface* MPI por rotinas do sistema de monitoração.

Normalmente, se uma aplicação MPI falha ou é abortada, todos os dados coletados são perdidos. A biblioteca responsável por gerenciar o rastreamento (libVT) necessita da MPI para escrever o *tracefile*, mas o padrão MPI não garante funcionalidade após falha. Na prática, várias implementações simplesmente abortam a aplicação. Para resolver esse problema, a aplicação deve ser recompilada com *link* a uma biblioteca que implementa recursos de tolerância a falhas (libVT *fail-safe*).

Dessa forma, em condições normais, o processo de rastreamento irá trabalhar como a bibloteca padrão (libVT), mas as comunicações para a escrita do *tracefile* serão realizadas via *socket* TCP. Esse processo de comunicação é um pouco mais lento, mas a escrita do *tracefile* em disco é garantida.

O arquivo com os dados do monitoramento, gerado pela ferramenta *Intel Trace Collector*, é armazenado em disco em formato binário e a conversão para o formato ASCII é possível através da ferramenta *stftool* que acompanha o ITC. A principal vantagem do formato ASCII é o fato desse arquivo ser interpretado por qualquer linguagem de programação.

Cada linha do *tracefile* contém um único evento ou comando escritos de acordo com a sintaxe: [tempo] ⟨commando⟩ [parâmetro [parâmetro]].

Geralmente, uma linha inicia com um número designando o tempo no qual o evento finalizou. Esse tempo é dado em *ticks* relativos ao início do programa. O tempo também é utilizado para ordenar os eventos no *tracefile*. Um número de *tick* não necessita estar na primeira coluna. O software distinguirá um comando de uma especificação de tempo ao verificar o formato do número [32].

O comando é composto por números ou letras. O primeiro dígito sempre será uma letra, caso contrário, o *software* poderá entender o comando como sendo um valor de *tick*. Não deve haver nenhum espaço em branco devido ao fato desses espaços serem utilizados para separar comandos e parâmetros. A lista de parâmetros depende de cada comando utilizado.

Os comandos reconhecidos pelo ITC podem ser classificados em duas categorias:

(i) Comandos de configuração: descrevem parâmetros do rastreamento utilizados pela ferramenta de análise,

(ii) Comandos de evento: refletem eventos que ocorrem durante a execução do programa.

A Tabela 5.1 apresenta um *tracefile* que descreve os comandos e eventos relativos a exe-

cução da aplicação *Integer Sort* (IS) classe C do *NAS Parallel Benchmark* (NPB) [33].

Tabela 5.1: Partes do *Tracefile* do IS NAS classe C

```
CLKPERIOD 1.2207E-13
EVENTBITS 13
DURATION 0 54913662976000
COMDEF 2 4 0:3:1 NAME "COMM_WORLD"
...
NCPUS 4
...
DEFSTATE 202 ACT 10 "MPI_Send"
DEFSTATE 258 ACT 10 "MPI_Reduce"
DEFSTATE 224 ACT 10 "MPI_Wait"
DEFSTATE 506 ACT 5 "User_Code"
...
GLOBALOPTOKEN 1 "MPI_Barrier"
GLOBALOPTOKEN 2 "MPI_Bcast"
GLOBALOPTOKEN 3 "MPI_Gather"
GLOBALOPTOKEN 4 "MPI_Gatherv"
..
GLOBALOPTOKEN 7 "MPI_Allgather"
GLOBALOPTOKEN 8 "MPI_Allgatherv"
GLOBALOPTOKEN 9 "MPI_Alltoall"
GLOBALOPTOKEN 10 "MPI_Alltoallv"
GLOBALOPTOKEN 11 "MPI_Reduce"
GLOBALOPTOKEN 12 "MPI_Allreduce"
GLOBALOPTOKEN 13 "MPI_Reduce_Scatter"
..
466944000 EXCHEXT CPU 4 DOWNTO 506
18235392000 EXCHEXT CPU 3 DOWNTO 506
46358528000 EXCHEXT CPU 1 DOWNTO 506
...
54722674688000 EXCHEXT CPU 1 DOWNTO 202
54722674688000 SENDMSG 2 1000 FROM 1 TO 2 LEN 4 FUNCTION 202
54724583424000 EXCHEXT CPU 2 UPTO 506
54724583424000 RECVMSG 2 1000 BY 2 FROM 1 LEN 4 FUNCTION 224
54769082368000 EXCHEXT CPU 3 DOWNTO 213
...
54913523712000 EXCHEXT CPU 1 UPTO 506
54913523712000 EXCHEXT CPU 1 UPTO NOACT
...
```

Os comandos de configuração são aqueles em que as linhas não iniciam com números.

### 5.2.2 Análise do *tracefile*

Uma análise do *tracefile* é realizada para criar o modelo da aplicação representado por um grafo. Agrupando as informações das linhas de eventos, determina-se o tempo total de execução de cada nó e o tempo e a quantidade de dados de cada comunicação.

A Tabela 5.2 ilustra os dados utilizados para calcular o tempo de execução da tarefa 1 da aplicação IS NAS classe C. A tarefa caminha para o estado 506, o qual corresponde a "User_code" segundo a definição de estado (DEFSTATE) apresentada na Tabela 5.1, no *tick* 46358528000, e entra no estado "NOACT"no *tick* 54913523712000. Subtraindo os valores de

*tick* final e *tick* inicial e multiplicando esse valor por 1.2207E-13, o qual corresponde ao período de *ticks* em segundos (Tabela 5.1), obtém-se o tempo total de execução dessa tarefa.

Tabela 5.2: Partes do *Tracefile* do IS NAS classe C - mostrando o custo computacional de uma tarefa

```
46358528000 EXCHEXT CPU 1 DOWNTO 506
...
54913523712000 EXCHEXT CPU 1 UPTO NOACT
```

De forma análoga, é possível determinar as comunicações entre as tarefas, a quantidade de dados transferidos e o tempo dessas comunicações, como ilustram os dados da Tabela 5.3. A tarefa 2 envia 4 *bytes* para a tarefa 3 no *tick* 53411512320000 utilizando a primitiva "MPI_Send" (Tabela 5.1). No *tick* 54770655232000, a tarefa 3 recebe 4 *bytes* da tarefa 2 através da primitiva "MPI_Wait" (Tabela 5.1).

Tabela 5.3: Partes do *Tracefile* do IS NAS classe C - mostrando o peso de uma comunicação

```
53411512320000 SENDMSG 2 1000 FROM 2 TO 3 LEN 4 FUNCTION 202
....
54770655232000 RECVMSG 2 1000 BY 3 FROM 2 LEN 4 FUNCTION 224
```

As comunicações globais são identificadas pelas linhas que contém a *tag* GLOBALOP e, considerando os espaços em branco como delimitadores de campos dessas linhas, o primeiro dado representa o *tick* em que esse evento ocorre, o terceiro, o tipo de primitiva de comunição global utilizada (Tabela 5.1), o quinto, a identificação de um processo envolvido na comunicação, o sétimo o *rootcpuid*, ou seja, o processo que executou a primitiva de comunicação. O oitavo, nono e décimo campo representam a quantidade de dados enviados, a quantidade de dados recebidos e a duração dessa comunicação, respectivamente. Quando o campo dados enviados ou recebidos apresenta o valor -1, significa que nenhum dado foi enviado ou recebido.

Para computar essas comunicações, são analisadas quatro possibilidades:

1. Se a quantidade de *bytes* enviados e recebidos é menor do que 0 é criada uma aresta do processo *rootcpuid* para o outro processo com custo de comunicação 0.

2. Se a quantidade de *bytes* enviados é menor do que 0 e a quantidade de *bytes* recebidos é maior do que zero é criada uma aresta do processo *n* para o processo *rootcpuid*, sendo o custo dessa comunicação igual à quantidade de *bytes* recebidos.

3. Se a quantidade de *bytes* enviados é maior do que 0 e a quantidade de *bytes* recebidos é menor do que 0 é criada uma aresta do processo *rootcpuid* para o outro processo sendo o custo dessa comunicação igual à quantidade de *bytes* enviados.

4. Se a quantidade de bytes enviados e de bytes recebidos é maior do que 0, são criadas 2 arestas. Uma do processo *rootcpuid* para o processo *n* e outra do processo *n* para o processo *rootcpuid*.

Tabela 5.4: Partes do *Tracefile* do IS NAS classe C - comunicações globais

```
172982386688000 EXCHEXT CPU 7 DOWNTO 259
172982386688000 GLOBALOP 12 ON 7 2 1 4116 4116 27115520000
...
172995829760000 EXCHEXT CPU 1 DOWNTO 255
172995829760000 GLOBALOP 9 ON 1 2 1 32 32 43704320000
...
2532929372160000 EXCHEXT CPU 4 DOWNTO 258
2532929372160000 GLOBALOP 11 ON 4 2 1 8 -1 3276800000
```

Após a primeira análise do *tracefile*, a visualização do grafo criado é totalmente incompreensível pois o número de comunicações entre as tarefas geralmente é elevado. A Figura 5.2 ilustra apenas a tarefa 2 da aplicação IS NAS Classe C após essa primeira análise.

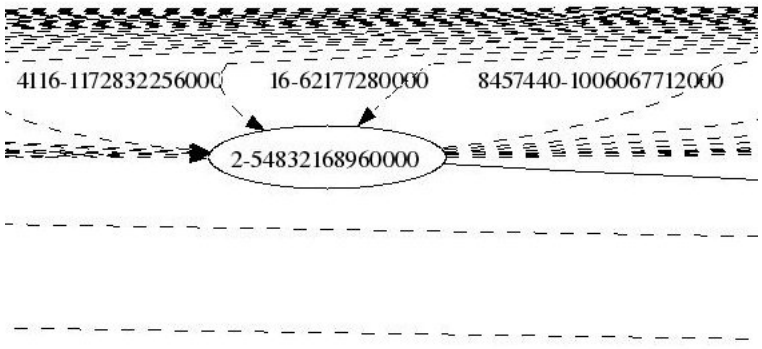



Figura 5.2: Visualização gráfica da tarefa 2 antes da sintetização das comunicações

Uma sintetização das comunicações é realizada neste trabalho com o objetivo de tornar o modelo de aplicação visível graficamente.

### 5.2.3 Sintetização do grafo

Neste trabalho, um grafo condensado é criado para melhorar a visualização gráfica da aplicação e das suas tarefas, como ilustrado na Figura 5.3. Esse grafo provê informações sobre o número de comunicações e sobre a quantidade de dados transferidos entre os nós.

Esse modelo pode ser representado graficamente através da ferramenta *Graphviz* da AT&T, como ilustrado na Tabela 5.5 e na Figura 5.3.

Tabela 5.5: Arquivo de entrada de dados da ferramenta *Graphviz* - gerado pela análise do *trace-file*

```
digraph G {
"n0"[ label = "0-2671629954"];
"n1"[ label = "1-2615497158"];
"n2"[ label = "2-2671715575"];
"n3"[ label = "3-2671709921"];
"n0"→ "n0"[ label="68-2954070224-5053621283"];
"n0"→ "n1"[ label="34-1475308904-1971805275"];
"n0"→ "n2"[ label="33-1487534468-1997653470"];
"n0"→ "n3"[ label="33-1465288588-1857006648"];
"n1"→ "n0"[ label="35-1476440340-1971815362"];
"n1"→ "n2"[ label="1-4-73459617"];
"n2"→ "n0"[ label="35-1476440340-2008094538"];
"n2"→ "n3"[ label="1-4-10449665"];
"n3"→ "n0"[ label="35-1476440340-1857022819"];
}
```

Na Figura 5.3, o *label* dos nós representa o número das tarefas e seus pesos de computacação ($\mu$s), enquanto o *label* das arestas representa o número de transmissões entre os nós e a quantidade de dados transferidos (*bytes*), respectivamente. Na Tabela 5.5, há um dado adicional, além dos apresentados na figura 5.3, que corresponde ao tempo de comunicação ($\mu$s) entre dois nós.

Por exemplo, o tempo total de execução do nó 1 é 2615497158 $\mu$s, enquanto o tempo de execução do nó 2 é 2671715575 $\mu$s. O nó 0 envia 34 mensagens para o nó 1, totalizando 1475308904 *bytes*, e o tempo de comunicação entre esses nós é 1971805275 $\mu$s; o nó 1 envia 1 mensagem para o nó 2 com 4 *bytes*.

Usando esse grafo condensado, o tempo gasto para escalonamento também melhora. Nesse caso em particular, da aplicação NAS IS *Benchmark*, ao invés de analizar um arquivo com 561 linhas o escalonador implementado por esse trabalho operou sobre um arquivo equivalente com 31 linhas. Enquanto 17 segundos foram necessários para aplicar o HEFT [15] no primeiro caso, apenas 0,00001 segundos foram necessários para o grafo condensado. Em ambos os casos, o escalonamento produzido foi o mesmo, o que significa que os mesmos nós foram selecionados

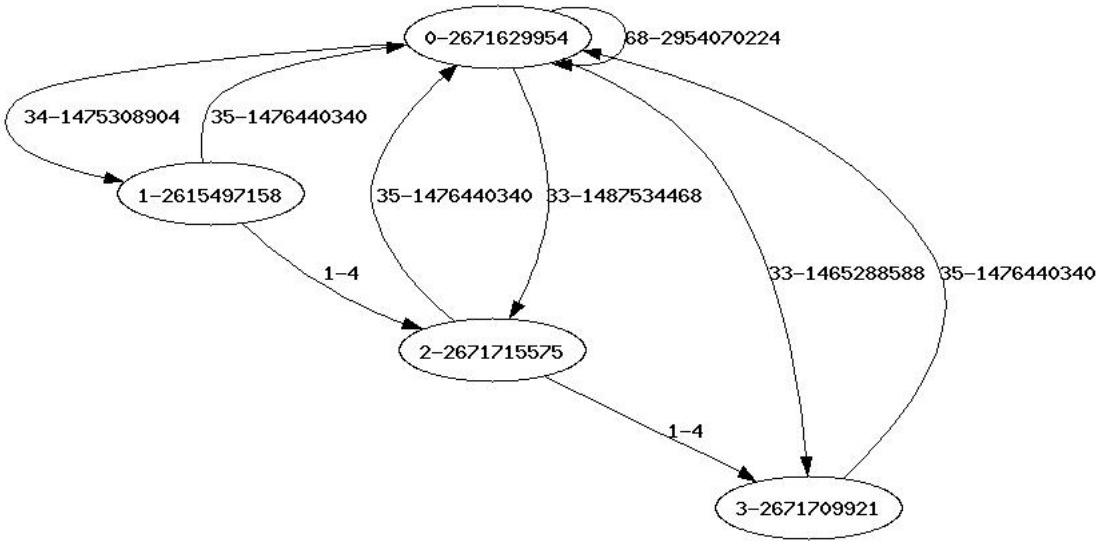



Figura 5.3: Grafo condensado do NPB IS classe C gerado pela ferramenta *Graphviz*

para a execução das tarefas.

Contudo, o grafo obtido ainda não é adequado para escalonamento, devido ao fato dele apresentar ciclos entre os nós.

### 5.2.4 Remoção de ciclos

Aplicações MPI geralmente apresentam um comportamento SPMD (*Simple Program Multiple Data*). Baseado em um *rank*, o nó mestre envia dados para serem processados pelos nós escravos, os quais, posteriormente, retornam os resultados. Geralmente, algumas primitivas para coordenação entre os nós também são utilizadas.

Um modelo de aplicação MPMD (*Muliple Program Multiple Data*) também é suportado. Códigos diferentes são usados para implementar tarefas diferentes mas as primitivas utilizadas para realizar a coordenação entre eles ainda são necessárias.

Em ambos os casos, as comunicações entre as tarefas podem produzir ciclos entre os nós. A Figura 5.3 mostra um exemplo de um grafo que reflete o comportamento da aplicação NAS NPB *Benchmark*, classe C.

Um grafo é considerado cíclico se há um caminho $\langle v_0, v_1, ..., v_k \rangle$ onde $v_0 = v_k$.

Antes do algoritmo de escalonamento ser executado, os ciclos existentes entre os nós de um grafo devem ser removidos. Nesse sentido, esse trabalho implementou um algoritmo baseado

em busca em profundidade (*Depth-first Search algorithm* - DFS) [34] para a remoção dos ciclos.

O algoritmo DFS explora, em profundidade, todas as arestas de um nó recentemente descoberto. Quando todas as arestas desse nó forem exploradas, o algoritmo retorna para analizar outras arestas do nó pai [34]. Nós são marcados durante a pesquisa indicando que eles já foram visitados.

Na implementação DFS desse trabalho, o algoritmo original foi estendido para pesquisar por qualquer nó $n_1$ que precede um nó $n_2$ o qual já foi visitado. Se isso ocorrer, um novo nó é criado, chamado de nó virtual, e todas as arestas que anteriormente uniam $n_1$ com $n_2$ serão direcionadas para o nó virtual, eliminando-se assim o ciclo. Estabelece-se uma relação entre o nome do nó real e do nó virtual para que operações de conversão desses nomes, futuramente, sejam realizadas de forma eficaz, como na fase de *rank* das tarefas no algoritmo de escalonamento.

O resultado final da remoção de ciclos é a transformação de um grafo cíclico, criado pelo monitoramento de uma aplicação MPI real, em um grafo acíclico. A Figura 5.4 exibe o grafo após a remoção dos ciclos. O nó virtual é representado pelo nó tracejado.

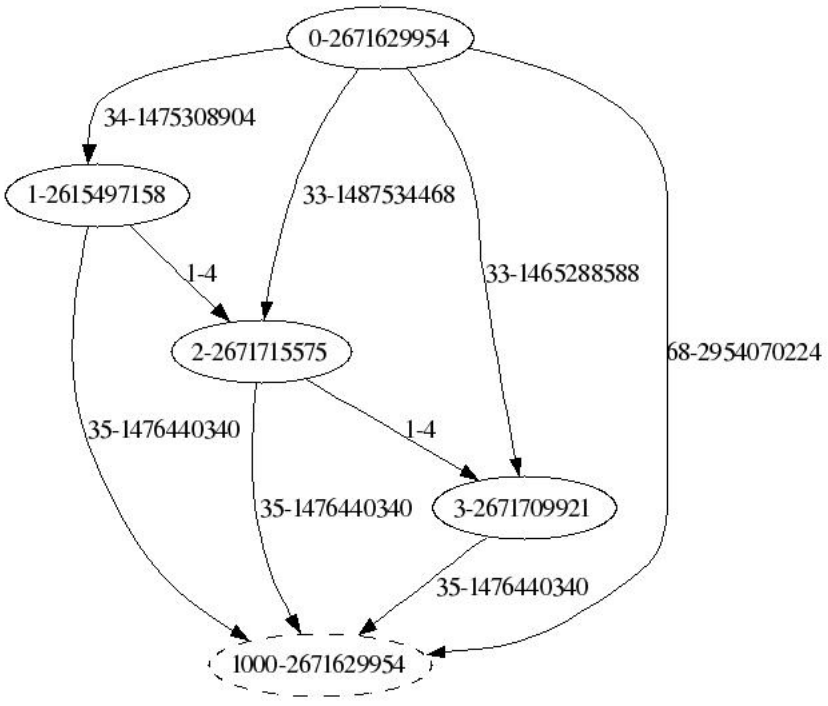



Figura 5.4: NPB classe C após a remoção dos ciclos

## 5.3 Adequando o algoritmo HEFT

O algoritmo HEFT (*Heterogeneous Earliest Finish Time*) proposto por Haluk Topcuoglu et al. em [15] é uma otimização do algoritmo EFT (*Earliest Finish Time*)visando um ambiente de processadores heterogêneos e utilizando heurísticas *list-scheduling*. Destaca-se como um dos principais algoritmos de escalonamento baseados em *list-scheduling*, por apresentar um tempo de escalonamento inferior a outros algoritmos [15].

Este algoritmo inicializa o custo de computação e o custo de comunicação das tarefas utilizando uma fórmula para calcular os valores médios e realiza o escalonamento com o objetivo de obter resultados melhores do que esses valores iniciais.

Nesse trabalho, esses valores médios são substituídos pelos valores obtidos durante a execução da aplicação em *cluster* fornecendo ao algoritmo dados mais precisos que permitem o cálculo do tempo de execução da aplicação no grid de uma forma mais realista.

A Tabela 5.6 descreve nos passos de 1 a 3 a primeira fase do algoritmo, que está relacionada à priorização das tarefas. Nessa etapa é utilizada a heurística de *top-level*, como descreve o passo 2. A segunda fase do algoritmo, relacionada à seleção do processador para uma dada tarefa, tem início no passo 4. Essa fase utiliza uma heurística de inserção para selecionar o melhor processador, como descreve o passo 4.2.1.

Tabela 5.6: Algoritmo de escalonamento HEFT

1) iniciar os custos de computação das tarefas e os custos de comunicação das arestas com os valores médios

2) calcular o *rank* para todas as tarefas através de um percurso pelo grafo em direção aos pontos mais altos começando de uma tarefa de saída

3) utilizando os valores do *rank*, organizar as tarefas em uma lista de escalonamento em ordem decrescente

4) enquanto há tarefas $(v_i)$ na lista de escalonamento faça
  4.1) selecione a primeira tarefa $(v_i)$ da lista de escalonamento
  4.2) para cada processador $(p)$ no conjunto de processadores faça
    4.2.1) calcule o valor de EFT$(v_i, pj)$ usando a política de escalonamento baseada em inserção
  4.3) atribua a tarefa $v_i$ para o processador $p_j$ que minimizar o EFT dessa tarefa

5) fim-enquanto

### 5.3.1 Classificação das tarefas

Baseado no GAD criado pela remoção de ciclos do grafo gerado pela análise do *log* do *Trace Collector*, a aplicação pode ser representada por uma estrutura de dados *G = (V,E,ε,ω)* onde *V* é o grupo de *n* nós e *E* é o grupo de arestas. *ε(i)* é o tempo de execução do nó *i*, e $\omega_(i,j)$ representa a quantidade de dados enviados entre os nós *i* e *j*.

Devido ao fato do *rank* ser computado recursivamente através do percurso realizado no grafo, iniciando na tarefa de saída com destino à tarefa de entrada, ele é denominado *upward rank* ou *top-level* e é defindo por [15]:

$$\mathtt{rank_u}(\mathtt{n_i}) = \varepsilon_{(\mathtt{i})} + \max_{n_j \in succ(n_i)} (\overline{\mathtt{c_{(i,j)}}} + \mathtt{rank_u}(\mathtt{n_j})) \tag{5.1}$$

$$\mathtt{rank_u}(\mathtt{n_{exit}}) = \varepsilon_{(\mathtt{exit})} \tag{5.2}$$

O valor $\overline{c_{(i,j)}}$ corresponde ao tempo de execução estimado da tarefa *i* no processador *j*, como mencionado no item 5.3.3.

Os nós virtuais que estão sendo criados para eliminar os ciclos recebem tratamento especial nessa etapa. Considerando que o maior valor de *rank* corresponde ao *rank* real de um nó, durante a inserção do nó na classificação final do *rank*, a implementação do algoritmo *upward rank* desse trabalho verifica se o nó é um nó real ou virtual. No caso de um nó ser virtual, seu nome é convertido para o nome do nó real. Em seguida, a lista com os *ranks* é pesquisada para verificar se esse nó já está na lista ou não. Se o nó estiver cadastrado na lista de *rank*, o valor do *rank* é comparado e o maior valor irá ser incluído na lista.

Essa etapa é finalizada com uma lista ordenada com as tarefas em uma ordem decrescente, criando uma lista de escalonamento.

### 5.3.2 Descoberta e seleção de recursos

Se o *grid* está sendo usado como uma arquitetura alternativa para a execução de aplicações paralelas é comum que existam mais recursos do que tarefas, de forma que, apenas os recursos mais adequados devem ser considerados. Entre os recursos disponíveis no *grid*, o modelo

arquitetural desse trabalho considera apenas os recursos com a mesma ou maior capacidade de processamento comparada com a melhor capacidade encontrada no *cluster*. Os processadores são organizados em um vetor *P* e as velocidades são representadas em MIPS. A largura de banda dos *links* de comunicação entre os recursos é representada por uma matriz *B*, de tamanho *pxp*. Os custos de início de comunicação (latência) de cada processador são armazenados em um vetor *L*. O vetor *avail* armazena os tempos nos quais o processador $p_i$ está pronto para executar uma tarefa.

Até o presente momento, o vetor *p* é obtido manualmente. Mecanismos para obtenção automática utilizando o *Globus* MDS [35] estão sendo desenvolvidos.

### 5.3.3 Relação entre tarefas e recursos do *grid*

O tempo estimado de execução para cada tarefa em cada processador disponível no *grid* é representado pela matriz *W* de dimensão *p x n*. O custo da tarefa *n* (em segundos) é multiplicado pelos MIPS do recurso do *cluster*, usado no monitoramento, e dividido pelos MIPS do recurso *p* do *grid*.

$$\overline{\mathtt{w}_{(\mathtt{n},\mathtt{p})}} = \frac{\mathtt{cost}_{(\mathtt{n})} * \mathtt{MIPS}_{(\mathtt{cluster})}}{\mathtt{MIPS}_{(\mathtt{p})}} \tag{5.3}$$

A média do custo de comunicação de uma aresta *(i,j)* é definida através da divisão do peso dessa aresta (em *bits*) pela média da largura de banda do *grid*. Esse resultado é somado com a média da latência de comunicação.

$$\overline{\mathtt{c}_{(\mathtt{i},\mathtt{j})}} = \overline{\mathtt{L}} + \frac{\mathtt{data}_{(\mathtt{i},\mathtt{j})}}{\overline{\mathtt{B}}} \tag{5.4}$$

### 5.3.4 Algoritmo EFT

As definições de *EST* e de *EFT* são derivadas de um escalonamento parcial. *EST*($n_i$,$p_j$) e *EFT*($n_i$,$p_j$) são o *earliest execution start time* e o *earliest execution finish time* da tarefa $\mathrm{n}_i$ no processador $\mathrm{p}_j$, respectivamente. *EFT* é definido por [15]:

$$\mathtt{EFT}(\mathtt{n_i}, \mathtt{p_j}) = \mathtt{w}_{(\mathtt{i},\mathtt{j})} + \mathtt{EST}(\mathtt{n_i}, \mathtt{p_j}) \tag{5.5}$$

$$\mathtt{EST}(\mathtt{n_i}, \mathtt{p_j}) = \mathtt{max}\left\{ \mathtt{avail[j]},\ \max_{n_m \in succ(n_i)} \left(\mathtt{AFT}(\mathtt{n_m}) + \mathtt{c}_{(\mathtt{m},\mathtt{i})}\right) \right\} \tag{5.6}$$

$$\mathtt{EST}(\mathtt{n_{entry}}, \mathtt{p_j}) = 0 \tag{5.7}$$

Após a tarefa $n_m$ ser escalonada no processador $p_j$, o *earliest start time* e o *earliest finish time* de $n_m$ no processador $p_j$ é igual ao *actual start time*, *AST($n_m$)* , e o *actual finish time*, *AFT($n_m$)*, da tarefa $n_m$, respectivamente [15].

## 5.4 Submissão no *grid*

Após a execução do escalonador de aplicação, o resultado do escalonamento, descrito na Tabela 5.8, é submetido para execução no *grid*. Esse procedimento é realizado com a geração de um arquivo RSL (*Resource Specification Language*), o qual é submetido para execução. Esse arquivo contém o *hostname* do recurso onde a tarefa será executada, o *label* da tarefa e o *path* do arquivo executável, como ilustra a Tabela 5.7.

Tabela 5.7: Arquivo RSL

```
+ ( &(resourceManagerContact="A9")
(count=1)
(label="subjob 0")
(environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
(LD_LIBRARY_PATH /opt/globus/lib/))
(directory="/home/user/prg-paralel")
(executable="/home/user/prg-paralel/mult-4096")
)
( &(resourceManagerContact="C1")
(count=1)
(label="subjob 1")
(environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
(LD_LIBRARY_PATH /opt/globus/lib/))
(directory="/home/user/prg-paralel")
(executable="/home/user/prg-paralel/mult-4096")
)
...
```

A Tabela 5.8 apresenta todas as etapas do escalonamento de uma aplicação com 5 tarefas. Inicia-se com a remoção de 5 ciclos do grafo dessa aplicação. Em seguida, o modelo arquitetural é selecionado e o *rank* das 5 tarefas é computado. O cálculo do EFT de cada tarefa em cada

processador do modelo arquitetural é executado e o processador que minimiza esse tempo é selecionado. O escalonamento é finalizado com a criação do arquivo *machines* que será utilizado pela biblioteca MPICH-G2.

No arquivo *machines* cada linha representa o número da tarefa de uma aplicação ou seja, no exemplo da Tabela 5.8 a tarefa 0 será executada no recurso A9, a tarefa 1 no recurso C1 e assim, sucessivamente.

Tabela 5.8: Resultado do escalonamento da Multiplicação de Matrizes com NP=5

```
user@myhost: /scheduler/$ ./sched mpi-mult-4096.txt.sched grid.dat cluster.dat latencia.dat
costcomm.dat /home/user/prg-paralel/ /home/user/prg-paralel/mult-4096

O grafo tem 5 ciclos

Estrutura do grid apos filtro
Codigo: 0, Nome: A0, MHz: 27076
Codigo: 1, Nome: A1, MHz: 27076
Codigo: 2, Nome: A2, MHz: 27076
Codigo: 3, Nome: A3, MHz: 27076
Codigo: 4, Nome: A4, MHz: 27076
Codigo: 5, Nome: A5, MHz: 27076
Codigo: 6, Nome: A6, MHz: 27076
Codigo: 7, Nome: A7, MHz: 27076
Codigo: 8, Nome: A8, MHz: 27076
Codigo: 9, Nome: A9, MHz: 57063
Codigo: 10, Nome: B0, MHz: 25600
Codigo: 11, Nome: B1, MHz: 25600
Codigo: 12, Nome: B2, MHz: 25600
Codigo: 13, Nome: B3, MHz: 25600
Codigo: 14, Nome: B4, MHz: 25600
Codigo: 15, Nome: B5, MHz: 25600
Codigo: 16, Nome: C0, MHz: 35200
Codigo: 17, Nome: C1, MHz: 28300
Codigo: 18, Nome: C2, MHz: 28300
Codigo: 19, Nome: C3, MHz: 28300
Codigo: 20, Nome: C4, MHz: 28300
Codigo: 21, Nome: C5, MHz: 28300
Codigo: 22, Nome: C6, MHz: 28300
Codigo: 23, Nome: C7, MHz: 28300

Rank das tarefas
Task: 0, Rank: 38707566210444528.000000
Task: 3, Rank: 25805046657789132.000000
Task: 2, Rank: 25805045558176972.000000
Task: 1, Rank: 25805045473340620.000000
Task: 4, Rank: 25805044957326540.000000

EFT que minimiza a execucao da task 0 no processador 9:528.316150 seg.
EFT que minimiza a execucao da task 3 no processador 16:857.377852 seg.
EFT que minimiza a execucao da task 2 no processador 9:1056.632570 seg.
EFT que minimiza a execucao da task 1 no processador 17:1066.196141 seg.
EFT que minimiza a execucao da task 4 no processador 18:1066.196099 seg.

Arquivo MACHINES
"A9"1
"C1"1
"A9"1
"C0"1
"C2"1
-------------------------------------------------------
Dados sobre o processo de escalonamento
-------------------------------------------------------
Hora i: 1183384192, Hora f: 1183384192, Tempo gasto: 0 seconds

user@myhost: /scheduler/$
```

Nesse caso, a aplicação foi executada usando a biblioteca MPICH-G2, que usa as capacidades do *grid* para iniciar os processos em sistemas remotos, para transferir os arquivos executáveis e os dados para sistemas remotos, e para segurança [36].

# 6. Resultados

O modelo de aplicação gerado pelo mecanismo proposto por esse trabalho é validado utilizando a ferramenta *Intel Trace Analyzer* e o processo de remoção de ciclos é testado com grafos gerados aleatoriamente pela ferramenta *GTGraph*. Com a criação de GADs consistentes para aplicações MPI reais, tempos de execução em *cluste*r e *grids*, reais e simulados, são comparados.

## 6.1 Considerações Iniciais

A avaliação do mecanismo de escalonamento proposto por esse trabalho inicia-se com a compilação de uma aplicação com *link* para biblioteca da ferramenta *Trace Collector*. Essa aplicação é então executada em *cluster*. O *log* gerado pela ferramenta *Trace Collector* é analizado e as informações relevantes (tempo total de execução, troca de mensagens, quantidade de dados transferidas, etc) são extraídas.

O grafo correspondente à aplicação é modelado e percorrido para a remoção dos ciclos existentes. O resultado final dessas etapas é um GAD, o qual descreve o comportamento da aplicação.

A descoberta de recursos é então executada e os resultados são usados para a seleção dos recursos mais apropriados. O algortimo de escalonamento HEFT é aplicado, relacionando o modelo de aplicação e o modelo arquitetural, utilizando o *Earliest Finish Time* como política de inserção. Com os resultados do escalonamento, um arquivo RSL é criado e, após a compilação da aplicação com a biblioteca MPICH-G2 [36], a aplicação é submetida para execução no *grid*.

Execuções reais e simulações foram utilizadas para investigar o uso de *grids* como uma alternativa para a execução de aplicações paralelas que utilizam passagem de mensagem.

O *cluster* e o *grid* real, com sistema operacional Linux Rocks 4.2.1, eram compostos por

16 nós *dual-core* com 4788 *bogomips* e 1 GB RAM. A rede de comunicação era baseada na tecnologia *Fast Ethernet*.

O ambiente elaborado para simular um *grid*, com 24 recursos, foi criado como ilustra a Figura 6.1 usando o simulador *GridSim* [24].

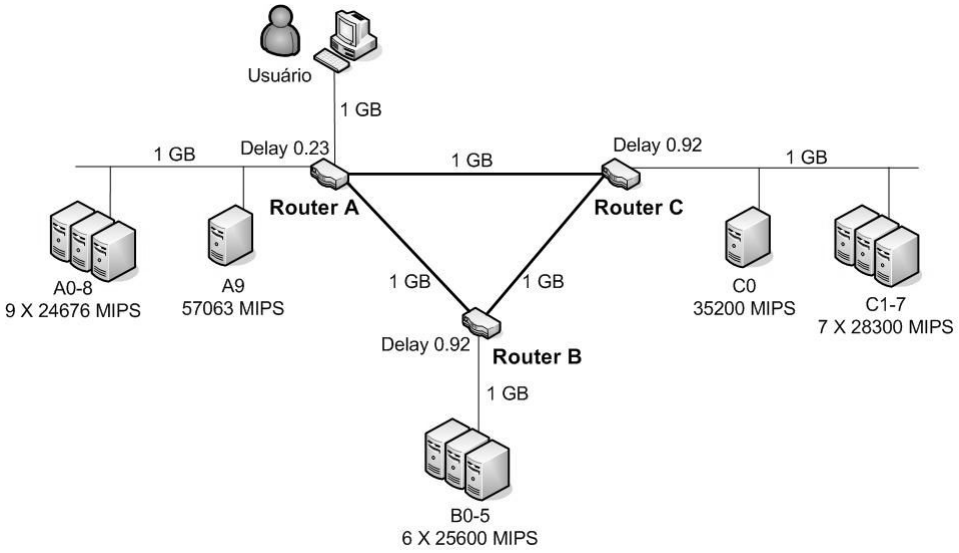



Figura 6.1: Ambiente de *grid* Simulado

Os resultados apresentados nesse capítulo mostram uma significativa melhora no tempo de execução quando a aplicação foi escalonada em *grid*.

## 6.2 Validação do grafo produzido

Para a validação do grafo produzido por esse trabalho, foi utilizada uma versão distribuída do algoritmo *Merge-Sort*.

O mesmo *tracefile* utilizado para gerar o grafo dessa aplicação também foi analizado pelo *Intel Trace Analyzer*, uma ferramenta desenvolvida pela *Intel* para analizar os dados gerados pelo *Intel Trace Collector*.

Usando essa ferramenta, pode-se observar que cada nó do grafo condensado que foi gerado, como ilustra a Figura 6.2, recebeu uma mensagem com 32768 *bytes*. Essas mensagens correspondem às comunicações do tipo *Scatter* , e são ilustradas pelos dados do *Trace Analyzer* na Figura 6.3. Da mesma forma, o nó 0 recebe duas mensagens, uma do nó 2 e a outra do nó 1. Nos dados do *Trace Analyzer*, o nó 0 tem duas comunicações *Recv*, totalizando 80795 $\mu$s, que correspondem á soma do tempo de duração dessas duas mensagens.

Através da comparação dos dados do grafo condensado, gerado por esse trabalho, e as informações analizadas pelo *Trace Analyzer* pode-se observar que todas as transmissões foram corretamente capturadas e representadas pelo modelo de aplicação gerado pelos mecanismos desenvolvidos por esse trabalho.

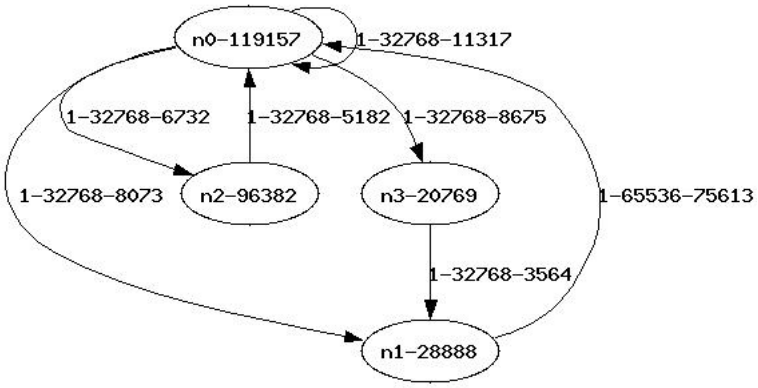



Figura 6.2: Grafo gerado do MPI *MergeSort*

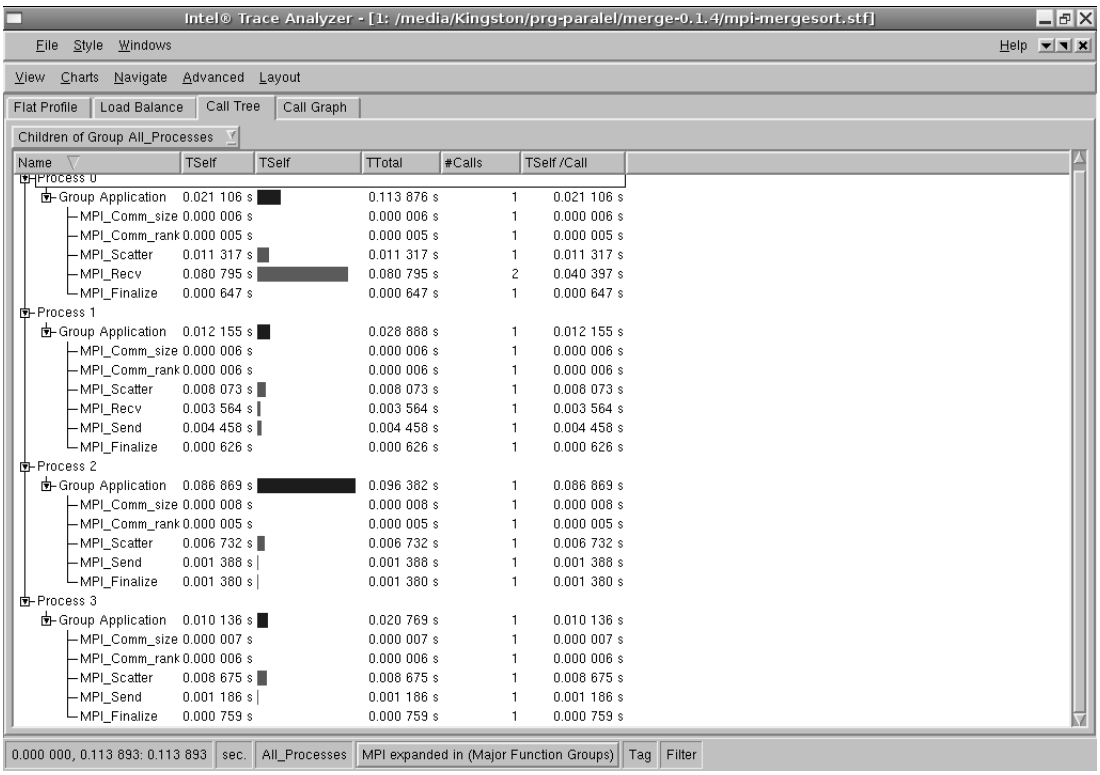



Figura 6.3: Análise do *Intel Trace Analyzer*

## 6.3 Validação da fase de remoção de ciclos

A análise do mecanismo de remoção de ciclos utilizou grafos coletados de aplicações MPI reais, como ilustram as Figuras 5.3 e 5.4, e também grafos gerados aleatoriamente.

Os grafos aleatórios foram obtidos usando o *GTGraph* [37] que é uma ferramenta geradora de grafos sintéticos desenvolvido para o DIMACS *Shortest Paths Challenge*. Essa ferramenta disponibiliza três métodos diferentes para a geração de grafos. Nos testes apresentados abaixo, foi utilizado o método *Random Graph*. Esse método por sua vez é composto por dois geradores de grafos aleatórios diferentes:

1. O *Erdös-Rényi* - gerador de grafos que recebe como argumentos de entrada o número de vértices e a constante de probabilidade de ocorrer uma aresta entre qualquer par de vértices no grafo.

2. O *Random* - gerador de grafos que recebe como argumentos de entrada o número de vértices *(n)* e o número de arestas *(m)*, e adiciona *m* arestas aleatoriamente escolhendo um par de vértice a cada momento.

A Tabela 6.1 ilustra a remoção de ciclos de um grafo gerado pelo pacote *Random* e a Tabela 6.2 um grafo gerado pelo pacote *Erdös-Rényi*.

Todos os grafos gerados aleatoriamente foram processados pelo método de remoção de ciclos desenvolvimento neste trabalho. Considerando que os nós virtuais são a representação de nós que anteriormente continham ciclos, uma análise visual em todos os casos mostra que o grafo resultante está correto e que não há nenhum ciclo existente.

Na Tabela 6.1, pode-se observar que há 4 arestas do nó 1 para o nó 0 no grafo original. Após a remoção de ciclos, essas arestas são redirecionadas do nó 1 para o nó virtual 1000. Da mesma forma, pode-se observar que as 3 arestas do nó 2 para o nó 1 foram redirecionadas para o nó virtual 1001 após a remoção de ciclos. A aresta original do nó 2 para o nó 0 também é redirecionada para o nó virtual 1000. Um total de 8 ciclos foram removidos nesse grafo.

## 6.4 Simulando aplicações MPI no GridSim

A aplicação utilizada para os testes simulados foi uma implementação paralela da multiplicação de matrizes que apresenta o comportamento representado na Figura 6.4, para NP=5. O

Tabela 6.1: Remoção de ciclos do grafo aleatório 1

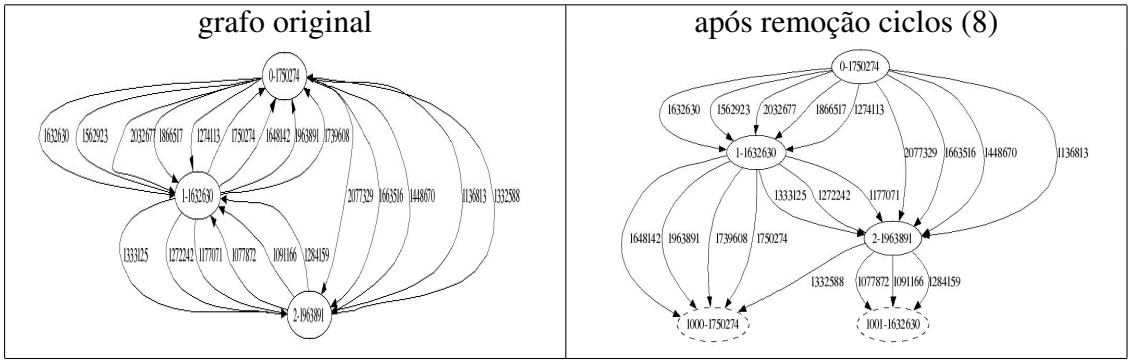
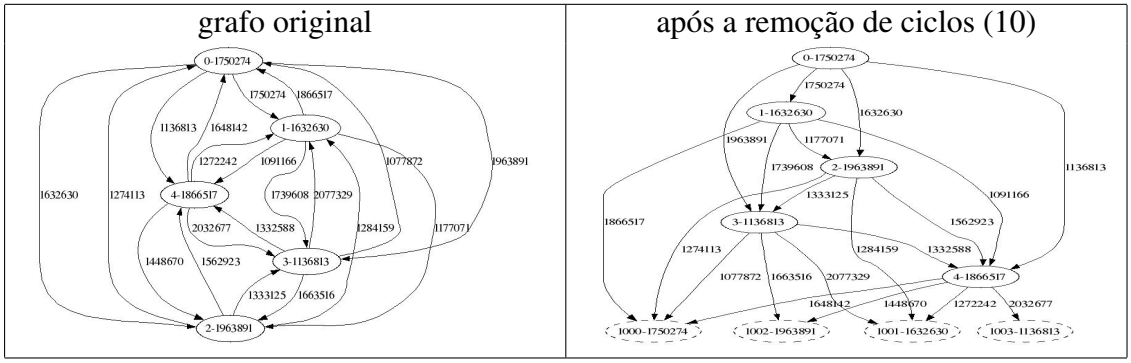



Tabela 6.2: Remoção de ciclos do grafo aleatório 2




comportamento para NP=32, que também foi utilizado nos testes, é análogo variando apenas a quantidade de *bytes* transferidos entre as tarefas.

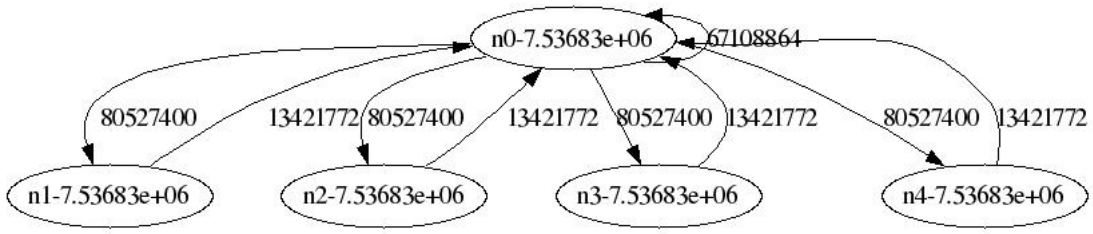



Figura 6.4: Grafo gerado da Mutilplicação de Matrizes Paralela

Pode-se observar que a tarefa 0 envia 80527400 *bytes* para as outras 4 tarefas e, depois, essas 4 tarefas enviam 13421772 *bytes* para a tarefa 0.

Como o simulador trata o tempo de processamento das tarefas nos recursos, para simular esse tipo de aplicação é necessário que o simulador também trate o tempo dessas comunicações.

Mesmo que fosse viável reescrever os métodos que tratam das portas de *input* e *output* dos recursos do *grid*, a cada simulação realizada, durante a escolha aleatória de recursos, as tarefas seriam alocadas em recursos diferentes. Um recurso que em determinada simulação estaria executando a tarefa 0 e, conseqüentemente, deveria enviar mensagens para todas as demais tarefas, em outra simulação poderia executar a tarefa 2 e não precisaria mais enviar mensagens para todas as demais tarefas. Isso mudaria o comportamento das portas de *input* e de *output* desse recurso, ou seja, o código teria que ser reescrito para cada simulação.

O método responsável por iniciar o processo de simulação é o *body()*, que é uma extensão da classe *GridSim*, e ele representa as ações que o usuário do *grid* realiza, como por exemplo, a submissão das tarefas de uma aplicação.

Utilizando uma lista que relaciona tarefas e recursos é possível saber onde uma determinada tarefa está sendo processada no *grid* simulado, e conseqüentemente, é possível enviar uma mensagem para esse recurso.

Para simular as comunicações entre as tarefas, logo após realizar a submissão dessas tarefas nos recursos do grid, e consultando a lista que relaciona tarefa e recursos, o código do usuário envia 4 pequenas mensagens (12 *bytes*, que correspondem ao tamanho aproximado de um objeto *Integer*), com a *tag* SEND_PKT e com o destinatário, para o recurso que executa a tarefa 0, como ilustra o código da Tabela 6.3.

Tabela 6.3: Trecho do método *body()* - mensagens iniciais

```
for (i = 1; i ⟨ list_.size(); i++)
{
   Destiny = new Integer(getEntityId(resourceName[resTask[i]]));

   System.err.println(this.name_ + ": Advising "+ resourceName[resTask[0]] +
      "to send a packet to "+ resourceName[resTask[i]] + "at time "+ GridSim.clock());

   super.send(super.output, 1, Program.SEND_PKT,
     new IO_data(Destiny,12,resourceID[resTask[0]]));
}
```

Foi implementado para cada recurso do *grid*, o método *processOtherEvent* em que o recebimento da *tag* SEND_PKT faz com que esse recurso envie uma mensagem para outro recurso com a *tag* RECV_PKT e com o tamanho BCAST_SIZE, que corresponde a uma constante com o valor de 80527400 *bytes*, para esse exemplo de aplicação, como descrito pelo conteúdo da Tabela 6.4.

Quando um recurso recebe uma mensagem com a *tag* RECV_PKT, ele imprime na tela o

Tabela 6.4: Trecho do método *processOtherEvent()* - *tag* SEND_PKT

```
protected void processOtherEvent(Sim_event ev)
{
try
    {
     case Program.SEND_PKT:
        System.err.println(super.get_name() + ": sending SEND_PKT to "+
          name + "with "+ Program.BCAST_SIZE + "bytes at time "+ GridSim.clock());

        super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.RECV_PKT,
          new IO_data(new Integer(super.get_id()), Program.BCAST_SIZE, obj.intValue()) );
        break;
```

recebimento da mensagem como ilustra o código da Tabela 6.5

Tabela 6.5: Trecho do método *processOtherEvent()* - *tag* RECV_PKT

```
case Program.RECV_PKT:
      System.err.println(super.get_name() + ": received RECV_PKT from "+
         name + "at time "+ GridSim.clock());

      break;
```

Até esse momento foram computadas as mensagens que a tarefa 0 envia para as demais tarefas. Para simular as comunicações que as 4 tarefas enviam para a tarefa 0, o processo é semelhante, mudando apenas a *tag* de envio, para que com isso também seja alterado o tamanho da mensagem, ou seja, a quantidade de *bytes* a serem enviados.

O usuário agora envia 4 mensagens para os recursos que executam as tarefas de 1 a 4 com a *tag* SEND2_PKT e com o destinário sendo o recurso que executa a tarefa 0, como ilustra o código da Tabela 6.6.

Tabela 6.6: Trecho do método *body()* - mensagens finais

```
for (i = 1; i ⟨ list_.size(); i++)
{
   Destiny = new Integer(getEntityId(resourceName[resTask[0]]));

   System.err.println(this.name_ + ": Advising "+ resourceName[resTask[i]] +
     "to send a packet to "+ resourceName[resTask[0]] + "at time "+ GridSim.clock());

   super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.SEND2_PKT,
     new IO_data(Destiny,12,resourceID[resTask[i]]) );
}
```

Quando um recurso recebe uma mensagem com a *tag* SEND2_PKT ele envia uma mensagem para o recurso de destino com a *tag* RECV_PKT e com tamanho ANSWER_SIZE, que

para esse exemplo de comunicação corresponde aos 13421772 *bytes*, como ilustra o conteúdo da Tabela 6.7.

Tabela 6.7: Trecho do método *processOtherEvents() - tag* SEND2_PKT

```
case Program.SEND2_PKT:
    System.err.println(super.get_name() + ": sending SEND2_PKT to "+
      name + "with "+ Program.ANSWER_SIZE + "bytes at time "+ GridSim.clock());

    super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.RECV_PKT,
      new IO_data(new Integer(super.get_id()), Program.ANSWER_SIZE, obj.intValue()) );

    break;
```

O tempo gasto para que o usuário informe um recurso que ele deve enviar uma mensagem para outro recurso não interfere no tempo de execução previsto pelo simulador. Dessa forma, utilizando esse método é possível computar o tempo das comunicações que as tarefas de uma aplicação MPI podem realizar.

## 6.5 Ambiente real X Ambiente simulado

Para melhorar a confidencialidade dos resultados das simulações, primeiro utilizou-se o simulador *GridSim* para modelar um ambiente similar ao *cluster* real. O *benchmark* da multiplicação de matrizes foi então executado no *cluster* real e no simulado. O mesmo teste foi aplicado em diferentes dimensões de matrizes e diferentes números de nós, como mostra a Tabela 6.8, para NP=5, e a Tabela 6.9, para NP=32.

Tabela 6.8: Tempo de execução para o *cluster* real e para o simulado - NP=5

| Multiplicação de Matrizes Paralela - NP 5 | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| Dim | *Cluster* Real | *Cluster* Simulado | % Compatibilidade |
| 512 | 1.05835400 | 7.16849200 | -477.32 |
| 1024 | 7.10559800 | 12.43584800 | 24.99 |
| 2048 | 54.05923400 | 60.57475850 | 87.95 |
| 4096 | 1555.62138000 | 1579.10880195 | 98.49 |
| 8192 | 14644.3968000 | 14649.4240890 | 99.97 |

As quantidades de processos 5 e 32 foram escolhidas levando-se em consideração a quantidade de recursos disponíveis no *grid*. Para 5 processos, o escalonador teria mais recursos

Tabela 6.9: Tempo de execução para o *cluster* real e para o simulado - NP=32

| Multiplicação de Matrizes Paralela - NP 32 | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| Dim | *Cluster* Real | *Cluster* Simulado | % Compatibilidade |
| 512 | 0.861159 | 8.238290 | -756.65 |
| 1024 | 3.760906 | 11.470340 | -104.98 |
| 2048 | 20.806705 | 56.526995 | -71.67 |
| 4096 | 351.512418 | 363.129668 | 96.69 |
| 8192 | 3957.776188 | 3984.492556 | 99.32 |

disponíveis do que tarefas e, para 32 processos, o escalonador teria mais tarefas do que recursos disponíveis, considerando que o *grid* simulado era composto por 24 recursos. Usando esse critério pode-se observar a relação comunicação X computação.

Observa-se diferentes graus de compatibilidade entre os resultados da simulação e da execução no *cluster*. A dimensão da matriz de 4096 elementos foi utilizada nos testes que serão apresentados nas seções a seguir devido ao fato de apresentar uma boa compatibilidade (98,49% para NP=5 e 96,69% para NP=32) entre o tempo de execução em ambiente real e em ambiente simulado.

## 6.6 MPICH X MPICH-G2

O caminho mais simples para portar um aplicação MPI de *cluster* para *grid* é recompilando essa aplicação com a biblioteca MPI com suporte para *grids*. Dessa forma, comparou-se o desempenho da multiplicação de matrizes usando as bibliotecas MPICH e MPICH-G2.

As aplicações foram executadas no *cluster* e no *grid* com *Globus* usando os mesmos nós para o processamento.

Analisando a maior complexidade de compartilhamento, dados os aspectos de segurança e de comunicação existentes em um *grid*, pode-se supor que o *cluster* deveria apresentar um melhor desempenho, mas isto não é verdade para todos os casos.

A implementação melhorada de mecanismos de comunicação permitiu que a versão disponível para *grid* da biblioteca MPI apresentasse um melhor desempenho quando comparada com o da versão para *cluster*, considerando que a aplicação tinha um reduzido números de processos. Isso pode ser observado na Tabela 6.10.

Com um número maior de processos, a versão de MPI para *cluster* passou a apresentar

melhor desempenho, como ilustra a Tabela 6.11.

Os resultados apresentados na Tabela 6.12, mostram que o número de processos de uma aplicação influencia no desempenho de ambas as versões de MPI, provavelmente devido ao modelo interno de comunicação das bibliotecas.

Tabela 6.10: Tempos de execução no *cluster* para MPICH e MPICH-G2 - NP=5

| Multiplicação de Matrizes Paralela - NP 5 | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| Dim | MPICH | MPICH-G2 | Melhora (%) |
| 512 | 1.05835400 | 0.9725650 | 7.65 |
| 1024 | 7.01737600 | 5.70530500 | 18.7 |
| 2048 | 87.51721900 | 41.16028300 | 52.97 |
| 4096 | 1596.4805590 | 1123.1961240 | 29.65 |

Tabela 6.11: Tempos de execução no *cluster* para MPICH e MPICH-G2 - NP=32

| Multiplicação de Matrizes Paralela - NP 32 | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| Dim | MPICH | MPICH-G2 | Melhora (%) |
| 512 | 0.859025 | 6.103257 | -610.48 |
| 1024 | 3.752422 | 13.468464 | -258.92 |
| 2048 | 20.790148 | 53.233273 | -156.05 |
| 4096 | 350.216397 | 568.933196 | -62.45 |

Tabela 6.12: Tempo de execução no *cluster* para MPICH e MPICH-G2 - matriz com dimensão 4096x4096, usando diferentes valores de NP

| Multiplicação de Matrizes Paralela | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| NP | MPICH | MPICH-G2 | Melhora (%) |
| 4 | 1597.494272 | 1166.386959 | 26.98 |
| 8 | 823.115742 | 620.726422 | 24.58 |
| 16 | 547.531631 | 464.800095 | 15.10 |
| 32 | 383.047842 | 577.331566 | -50.72 |

## 6.7 Computação X Comunicação

O efeito da granularidade, como é relatado para a relação de comunicação X computação, também foi considerado nos testes.

Mantendo a mesma tecnologia de rede (*Fast Ethernet*) para o *cluster* e para o *grid*, apenas com maior latência (*delay*) no *grid*, como ilustrado na Figura 6.1, a relação comunicação X computação foi analisada.

Supondo que cada recurso do *grid* apresenta um desempenho melhor do que os recursos do *cluster*, a migração para *grid* pode melhorar a relação comunicação X computação.

Nos testes apresentados a seguir, o pior recurso do *grid* era 41.47% melhor do que os recursos do *cluster*.

### 6.7.1 Análise de desempenho para NP=5

Para a mesma aplicação de multiplicação de matrizes, o tempo de execução para o *cluster* foi 1555.62138 segundos. Como mostra a Tabela 6.13, mesmo escolhendo os recursos do *grid* aleatoriamente, todas as combinações de recursos testadas produziram um ganho de desempenho de aproximadamente 29% no *grid*.

As 5 primeiras colunas da Tabela 6.13 representam as tarefas da aplicação. As linhas dessa tabela representam as 5 execuções da aplicação e cada célula dessas linhas informam o recurso no qual a tarefa foi executada. O tempo de cada execução é descrito na última coluna.

Tabela 6.13: Execução no *grid* simulado com rede *Fast Ethernet* - NP=5

| Multiplicação de Matrizes Paralela - NP=5 | | | | | |
|---|---|---|---|---|---|
| Tempo de Execução (seg) | | | | | |
| Tarefa0 | Tarefa1 | Tarefa2 | Tarefa3 | Tarefa4 | Tempo |
| A8 | C2 | B3 | A7 | C1 | 1118.43323817 |
| A1 | A4 | B3 | B5 | B2 | 1118.43323817 |
| C7 | A2 | A2 | C7 | A5 | 1119.43323817 |
| C2 | A9 | B5 | C7 | C0 | 1118.43323817 |
| B1 | A2 | A5 | A4 | C4 | 1118.43392817 |

Nesse caso, aumentando a capacidade de computação dos recursos, sem se preocupar com os *links* de comunicação entre eles, o tempo de execução em *grid* sempre foi melhor do que em *cluster*. Isso pode ter ocorrido devido ao número de comunicações não ser elevado e a importância da verificação do *link* de comunicação não ser tão relevante.

### 6.7.2 Análise de desempenho para NP=32

Usando um grande número de processos, se a quantidade de dados para ser processada é preservada, o processamento em cada recurso é reduzido. Isso produz uma relação de mais comunicação do que computação e o desempenho no *grid* é pior.

O tempo de execução no *cluster* foi 351.512 segundos e os resultados da Tabela 6.14, que foram obtidos através da escolha aleatória de recursos, são aproximadamente 77% piores do que o tempo de execução no *cluster*.

A primeira coluna da Tabela 6.14 representa cada recurso do *grid* e as outras 3 colunas representam 3 execuções da multiplicação de matrizes. As linhas das colunas que representam as execuções informam o número da tarefa que foi executada no recurso correspondente a essa linha. A última linha descreve o tempo de cada execução.

Esses resultados preliminares mostram que, considerando a escalabilidade da aplicação, a seleção do número apropriado de recursos, bem como do *link* de comunicação entre eles, pode determinar a viabilidade de migrar essa aplicação para o *grid*.

## 6.8 Seleção de recursos aleatória X escalonador de aplicação

O último experimento foi conduzido para determinar a influência do uso de um algoritmo de escalonamento para selecionar os recursos do *grid* que serão mais adequados a uma dada aplicação.

As Tabelas 6.15 e 6.16 mostram os resultados para a seleção de recursos aleatória, para a escolha dos recursos a partir de uma lista ordenada por capacidade de processamento, e para a escolha dos recursos usando o escalonador de aplicação baseado no algoritmo de escalonamento HEFT.

### 6.8.1 Análise para NP=5

Os resultados nas linhas de 1 a 4 da Tabela 6.15 foram obtidos usando uma lista de recursos gerada aleatoriamente (*machine file*). Na linha 5, o *machine file* foi ordenado por capacidade de processamento e, na linha 6 está o resultado da seleção de recursos usando a implementação do algoritmo HEFT e do GAD gerado por este trabalho.

Tabela 6.14: Execução em *grid* simulado com rede *Fast Ethernet* - NP=32

| Multiplicação de Matrizes Paralela - NP 32 | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| Recursos | Tarefas | Tarefas | Tarefas |
| A0 | 16,30 | 15,29,31 | |
| A1 | 4 | 25 | 26 |
| A2 | | 9 | 9,21,24 |
| A3 | 1,26 | 14 | 4,25 |
| A4 | | 10,11,28 | 2 |
| A5 | | 26,30 | 5,8 |
| A6 | 3,25 | 12,23,24 | 3,22 |
| A7 | 13 | 27 | 30 |
| A8 | 5,22 | | 29,31 |
| A9 | | 13 | 15,18 |
| B0 | | 7,16 | 0.19 |
| B1 | 11 | 21 | 12,17 |
| B2 | 6,8 | 22 | |
| B3 | 0,10,27,29,31 | 4 | |
| B4 | 21,23 | 2,17 | 28 |
| B5 | 18,24 | | 6,7,20 |
| C0 | 28 | 5 | 16,27 |
| C1 | | 1 | 1,11 |
| C2 | 7 | | |
| C3 | 2,12,19 | 8,18 | |
| C4 | 15 | 3,6 | 14 |
| C5 | 14,17,20 | | |
| C6 | 9 | 19 | 10,23 |
| C7 | | 0,20 | 13 |
| Tempo | 422.3939 | 745.832300 | 705.8585 |

### 6.8.2 Análise para NP=32

Na Tabela 6.16, nos resultados da coluna 1, o *machine file* estava desordenado. Para a coluna 2, o *machine file* estava ordenado por capacidade de processamento, e a coluna 3 mostra o resultado da seleção de recursos usando a implementação do algoritmo HEFT e do GAD gerado por este trabalho.

Em ambas as análises, para NP=5 e NP=32, os resultados mostram que o uso de um escalonador de aplicação, que considerada as características da aplicação, produz o melhor tempo de execução no *grid*.

Tabela 6.15: Execução no *grid* com rede *Gigabit* - NP=5

| Multiplicação de Matrizes Paralela - NP=5 | | | | | | |
|---|---|---|---|---|---|---|
| Tempo de Execução (seg) | | | | | | |
| Linha | Tarefa0 | Tarefa1 | Tarefa2 | Tarefa3 | Tarefa4 | Tempo |
| 1 | A2 | B3 | C7 | C4 | A6 | 1118.4334680 |
| 2 | A0 | C4 | A7 | A7 | A1 | 1119.4333680 |
| 3 | B5 | A6 | A6 | C3 | B1 | 1169.4332380 |
| 4 | A6 | A4 | C2 | A5 | A2 | 1118.4339380 |
| 5 | A9 | C0 | C1 | C2 | C3 | 271.3240699 |
| 6 | A9 | A9 | C0 | C1 | C2 | 271.3231499 |

## 6.9 Discussão

Uma latência maior, típica das conexões de *Internet*, pode desencorajar o uso de *grids* para executar aplicações paralelas.

Contudo, experimentos e simulações mostram que um escalonador de aplicação apropriado, o qual considera as características de processamento e de comunicação das aplicações, para realizar a seleção dos recursos do *grid* podem tornar os ambientes de *grids* viáveis para essas aplicações e também podem melhorar seus tempos de execução.

A relação comunicação X computação ainda é um fator decisivo quando deseja-se migrar uma aplicação, ou mesmo quando deseja-se explorar a escalabilidade de uma aplicação paralela em *grids*.

## 6.10 Considerações sobre os trabalhos experimentais

Durante a implementação dos mecanismos apresentados no Capítulo 5 e nas seções anteriores, algumas dificuldades foram encontradas destacando-se dentre elas o fato das aplicações MPI reais apresentarem ciclos.

Diante desse problema, todo o escalonamento ficou comprometido pois não era possível calcular o *rank* das tarefas e os algoritmos de escalonamento, em sua grande maioria, baseiam-se nesse *rank*.

Surgiu a idéia da criação de nós virtuais e utilizando os conceitos e técnicas para a manipulação de grafos foi possível conciliar a pesquisa em profundidade no grafo, com o objetivo de localizar os ciclos, com a criação dos nós virtuais possibilitando assim a remoção dos ciclos.

Tabela 6.16: Execução no *grid* com rede *Gigabit* - NP=32

| Multiplicação de Matrizes Paralela - NP=32 | | | |
|---|---|---|---|
| Tempo de Execução (seg) | | | |
| Recursos | Tarefa | Tarefa | Tarefa |
| A0 | 20,26 | 15 | 3 |
| A1 | 13,29 | 16 | 2 |
| A2 | 2,23 | 17 | 26 |
| A3 | 6,22 | 18 | 22 |
| A4 | 25,31 | 19 | |
| A5 | 0,4 | 20 | 31 |
| A6 | 21,28 | 21 | 7 |
| A7 | 7 | 22 | 23 |
| A8 | 15,30 | 23 | 17 |
| A9 | | 0,24 | 0,9,29 |
| B0 | 3 | 9 | 1 |
| B1 | 11,16,27 | 10 | 15,21 |
| B2 | 5,12,17,24 | 11 | 25 |
| B3 | | 12 | 12 |
| B4 | 10 | 13 | 4,6 |
| B5 | 9,14,19 | 14 | 20 |
| C0 | | 1,25 | 11,13 |
| C1 | | 2,26 | |
| C2 | | 3,27 | 5 |
| C3 | | 4,28 | 8,30 |
| C4 | | 5,29 | 14,24 |
| C5 | 1 | 6,30 | 10,16 |
| C6 | | 7,31 | 18,27 |
| C7 | 8,18 | 8 | 19,28 |
| Tempo | 531.0313 | 131.18.03 | 119.3441 |
| Coluna | 1 | 2 | 3 |

Conseguindo calcular o *rank* das tarefas, notou-se que era preciso adequar esse *rank* pois ele passou a calcular a tarefa virtual como se ela fosse uma tarefa real nova. Adotou-se então um mecanismo de nomenclatura para essas tarefas virtuais possibilitando a conversão do nome dessa tarefa de virtual para real e vice-versa.

Durante a implementação do algoritmo HEFT a preocupação maior foi o ajuste de unidades (segundos, ticks, MIPS, ciclos, *bits* e *bytes*) dos dados utilizados, necessário para a execução de um escalonamento correto e preciso.

Com os mecanismos de modelagem de aplicação e de escalonamento prontos, surgiu a necessidade de tentar generalizar o comportamento das aplicações MPI durante o escalonamento. A príncipio foi utilizado o *benckmark* NAS IS (*Integer Sort*) classe A, B e C devido ao fato

de ser um dos *benchmark* mais citados em outros trabalhos acadêmicos. Posteriormente, foi implementada uma versão paralela do MergeSort e da multiplicação de matrizes com o objetivo de diversificar as aplicações a serem escalonadas e de verificar o comportamento dessas aplicações.

Como alguns trabalhos sobre escalonamento citavam a utilização de GAD sintéticos, esse trabalho utilizou a ferramenta GTGraph para modelar esse tipo de grafo. A geração de GADs sintéticos permitiu verificar o desempenho do algoritmo de escalonamento em relação a quantidade de tarefas e arestas de uma aplicação. Foi possível escalonar um grafo com 200 tarefas, 400 arestas em 21 segundos. Esse grafo apresentou 61 ciclos que foram removidos e tratados de forma correta durante o cálculo do *rank* das tarefas. Atualmente, uma otimização no código realizada no trabalho de Ricardo Rios [38], permite que grafos com aproximadamente 2.000 tarefas, 398.840 arestas e 288.926 ciclos sejam escalonados em torno de 758,26 segundos.

A fase de testes foi organizada com dois objetivos principais que eram, validar o modelo de aplicação gerado por esse trabalho e demonstrar que a execução de aplicações MPI em *grids* é viável.

Durante a validação do modelo de aplicação gerado, foram realizadas análises em arquivos com 420.000 linhas o que demonstrou que o tipo de linguagem utilizada no *script* de análise não é a mais adequada pois, para esse caso, a análise demorou aproximadamente 2 horas. Esse tempo não interfere no tempo de escalonamento uma vez que o GAD gerado será um dado de entrada do escalonador mas desejamos que o usúario realize essa modelagem de forma mais eficiente.

Para demonstrar que a execução de aplicações MPI em *grids* é viável, foram utilizados ambientes reais e ambientes simulados. A escolha de um simulador baseou-se nas características desse simulador e na praticidade dos exemplos encontrados. Esse tipo de escolha conduziu à utilização do simulador GridSim.

Devido ao fato desse simulador trabalhar com aplicações *task-farming* em sua implementação mais básica, foi necessário realizar pesquisas nos códigos dos 10 programas exemplos, disponíveis junto ao código desse simulador, e nos métodos das classes da API do simulador para elaborar uma forma de simular aplicações MPI.

Com a simulação de aplicações MPI sendo realizada no GridSim o principal problema passou a ser o tempo que essa simulação apresentava. Eram necessárias aproximadamente 2 horas para simular uma aplicação com 32 tarefas, provavelmente, devido ao fato de cerca de

160 *threads* Java serem criadas. Esse tipo de simulação necessitava de uma máquina com pelo menos 2 GBytes de RAM.

Todos os problemas decorrentes da fase de implementação e de testes contribuiram imensamente no aprimoramento desse trabalho pois a maioria desses problemas não eram imagináveis durante a elaboração da proposta desse trabalho.

# 7. Conclusões

Várias aplicações intensivamente parelelas ainda não utilizam ambientes de *grids* computacionais devido ao fato do custo das comunicações entre as tarefas potencialmente comprometer o tempo de execução dessas aplicações.

Enquanto *clusters* têm permitido que vários desses problemas computacionais sejam resolvidos de forma eficiente, muitos podem também beneficiar-se do uso de *grids* computacionais, ou seja, de um ambiente computacional onde os recursos são totalmente dedicados e interligados por uma rede de comunicação que apresenta um comportamento estável sem grandes oscilações nas taxas de envio/recebimento.

Na computação em *grid*, escalonadores de aplicação são utilizados para melhorar a seleção dos recursos disponíveis considerando as características das aplicações. Isso é extremamente útil para o escalonamento de aplicações multitarefas que apresentam comunicação entre essas tarefas.

Analizando arquivos de *log* de aplicações MPI resultantes do monitoramento em *cluster*, determinamos as características dessas aplicações e geramos um modelo de aplicação baseado em grafos.

Muitos programas MPI apresentam modelos de comunicação cíclicos e esse tipo de comportamento impede que diversos algoritmos de escalonamento sejam empregados devido ao fato deles percorrem recursivamente os grafos para priorizar as tarefas. Com o objetivo de eliminar esses ciclos de comunicação, foi implementado um mecanismo, baseado na busca em profundidade e no conceito de nós virtuais, o qual realiza a conversão de um grafo cíclico em um grafo acíclico, compatível com os algortimos de escalonamento.

Uma implementação do algoritmo HEFT foi desenvolvida e utilizada para validar o modelo de aplicação produzido. Foi necessário adptar a classificação final do *rank* das tarefas para tratar os nós virtuais criados durante a remoção dos ciclos.

Como definido pelo o conceito de RMS (*Resource Management System*), um modelo de aplicação e um modelo arquitetural foram utilizados em um escalonador de aplicação e o escalonamento obtido foi convertido em um arquivo RSL, o qual permitiu a submissão da aplicação em um *grid* com *Globus*.

Os testes realizados tiveram como objetivo a comparação entre os tempos de execução de uma aplicação em *cluster* e em *grid*. Foram utilizados um ambiente real, gerenciado pelo *middleware Globus*, e um ambiente simulado, criado através do simulador *GridSim*.

Os resultados mostram que a solução proposta obteve sucesso na modelagem da aplicação. Essa modelagem, por sua vez, conciliada a um algoritmo de escalonamento, possibilitou a seleção dos recursos do *grid* que minimizaram o tempo de execução das aplicações, concluindo que o uso de GADs permite escalonamento mais eficiente em *grid*.

O mecanismo apresentado por este trabalho gera automaticamente GADs de aplicações MPI simplificando seus escalonamentos em um *grid* e possibilitando melhora no tempo de execução dessas aplicações. Com essa criação automática, acreditamos que aplicações desenvolvidas a princípio para *cluster* podem ser mais facilmente adaptadas para execução em ambientes de *grids* computacionais.

# 8. Trabalhos Futuros

Com o objetivo de tornar esse trabalho mais completo sugere-se os itens de pesquisas relacionados a seguir.

1) A predição de GADs, para viabilizar a modelagem da aplicação sem que seja necessária sua execução em *cluster* quando seu NP ou quando a quantidade de dados a serem processados forem alteradas.

Através de uma análise visual é possível concluir que o grafo da Figura 8.1 apresenta a mesma quantidade de nós e de arestas quando comparado com o grafo da Figura 8.2. As arestas da Figura 8.1 interligam os mesmos nós do grafo da Figura 8.2 apresentando dessa forma o mesmo comportamento. Esses grafos representam as aplicações NAS *Benchmark* classe A e classe C, respectivamente. Nesse caso, o NP permaneceu o mesmo mas a quantidade de dados a ser processada foi alterada.

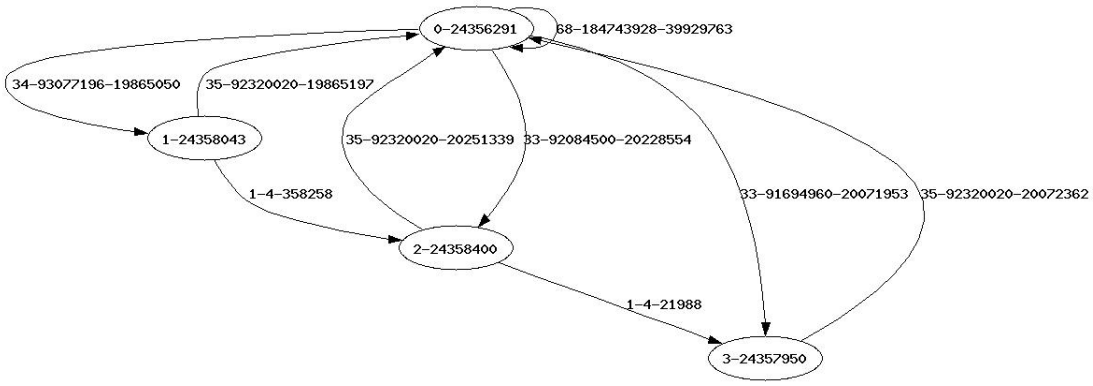



Figura 8.1: NAS *Benchmark* classe A com NP=4

Com outra análise visual, comparando agora os grafos das Figuras 8.2 e 8.3, é possível verificar que, embora o NP tenha sido alterado, o comportamento da aplicação permaneceu o mesmo. A tarefa 0 se comunica com todas as outras $n_i$ tarefas e essas, por suas vezes, retornam uma comunicação para a tarefa 0 e também enviam uma mensagem para a tarefa $n_{(i+1)}$.

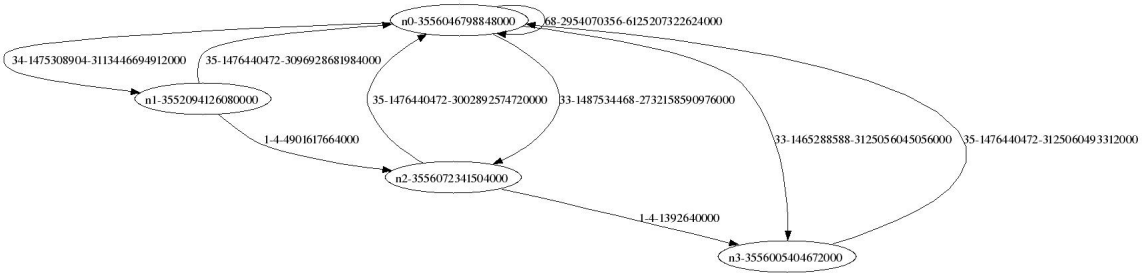



Figura 8.2: NAS *Benchmark* classe C com NP=4

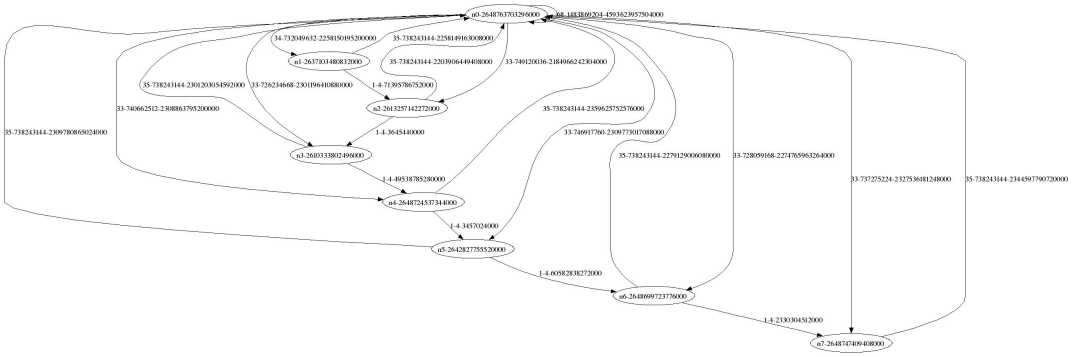



Figura 8.3: NAS *Benchmark* classe C com NP=8

Se o comportamento da aplicação é mantido, novos GADs podem ser criados com peso computacional das tarefas e peso de comunicação das arestas sendo estimados de forma satisfatória.

2) A escolha do melhor número de processos para uma dada aplicação, para explorar de forma correta toda a escalabilidade que um *grid* computacional oferece.

Os resultados dos testes mostram que a relação comunicação X computação pode encorajar ou não a execução de uma aplicação em *grid*. Mas uma das grandes vantagens de portar uma aplicação para *grid* é a possibilidade de explorar toda a escalabilidade que esse ambiente oferece.

Como algoritmos de aglomeração são utilizados para minimizar os custos de comunicações, faz sentido imaginar que esses algoritmos possam permitir explorar a escalabilidade de uma aplicação em um ambiente de *grid*. Se há recursos com um potencial maior de processamento e se o algoritmo de aglomeração irá colocar conjuntos de tarefas para serem executadas em um mesmo processador, eliminado os custos de comunicação, a relação comunicação X computação pode ficar equilibrada e a execucção em *grid* será viável.

3) A escolha de um espaço amostral de recursos mais adequado a uma dada aplicação,

visando minimizar o tempo de escalonamento.

No caso do HEFT implementado nesse trabalho, é realizado o cálculo do tempo de execução de *n* tarefas em *p* processadores. Pode-se concluir então que a escolha dos *n* melhores processadores do *grid* será suficiente para escalonar as *n* tarefas, sendo esse o pior tempo de escalonamento, ou seja *n* X *n*, colocando cada tarefa em um processador.

A escolha dos *n* melhores processadores, ou recursos, disponíveis no *grid* para uma dada aplicação é um fator muito importante para que os processos de escalonamento sejam viáveis, rápidos e eficientes.

4) Um escalonador centralizado pode tornar mais realista os resultados estimados pelos escalonadores de aplicação.

Nesse trabalho, o escalonador de aplicação não têm sua execução centralizada. Portanto, se um usuário A realiza a submissão de uma aplicação X no *grid* e, ao mesmo tempo o usuário B realiza o escalonamento de uma aplicação Y, o escalonador não irá considerar a carga que a aplicação X exerce sobre os recursos pois o modelo arquitetural é obtido quando o escalonador é iniciado. Isso tornará os resultados do escalonamento inadequados ao ambiente disponível nesse determinado instante.

A implementação de um escalonador centralizado, que também pode realizar ou não a submissão das aplicações no *grid*, poderá considerar a carga real de cada recurso tornando os resultados do escalonamento condizentes com o ambiente computacional disponível em um determinado momento.

# 9. Referências

[1] FOSTER, Y. What is the grid? a three point checklist. *Grid Today*, v. 1, n. 6, July 2002. Disponível em: <http://www.gridtoday.com/02/0722/100136.html>.

[2] FOSTER, Y.; KESSELMAN, C.; TUECKE, S. The anatomy of the grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, v. 15, n. 3, p. 200–222, 2001. Disponível em: <http://hpc.sagepub.com/cgi/reprint/15/2/200>.

[3] GANSNER, E. R. et al. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 19, n. 3, p. 214–230, 1993. ISSN 0098-5589.

[4] KOUTSOFIOS, E.; NORTH., S. *Drawing graphs with* D*ot*. Sep 1991. Technical Report 910904-59113-08TM, AT&T Bell Laboratories. Disponível em: <http://www.graphviz.org/Documentation/dotguide.pdf>.

[5] BERMAN, F. D. et al. Application-level scheduling on distributed heterogeneous networks. In: *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1996. p. 39. ISBN 0-89791-854-1.

[6] VIANNA, B. A. et al. A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In: *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*. New York, NY, USA: ACM Press, 2004. p. 41–46. ISBN 1-58113-950-0.

[7] FOSTER, I.; KESSELMAN, C. (Ed.). *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN 1-55860-475-8.

[8] FERREIRA, L. et al. *Introduction to Grid Computing with Globus*. September 2003. Technical Report ISBN 0738427969, IBM International Technical Support Organization.

[9] WELCH, V. *Globus Toolkit Version 4: Grid Security Infraestructure: A Standards Perspective*. September 2005. University of Chicago. Disponível em: <http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf>.

[10] ALLIANCE, G. GRAM*: Key Concepts*. University of Chicago. Disponível em: <http://www.globus.org/toolkit/docs/3.2/gram/key/index.html>.

[11] ALLIANCE, G. GT *2.4: The* G*lobus* R*esource* S*pecification* L*anguage* RSL *v1.0*. September 2003. University of Chicago. Disponível em: <http://www.globus.org/toolkit/docs/2.4/gram/rsl_spec1.html>.

[12] CIRNE., W. Grids computacionais: Arquiteturas, tecnologias e aplicações. *Anais do Terceiro Workshop em Sistemas Computacionais de Alto Desempenho*, Oct 2002.

[13] ALLIANCE, G. *Data Management: Key Concepts*. University of Chicago. Disponível em: <http://www.globus.org/toolkit/docs/4.0/data/key/index.html>.

[14] KRAUTER, K.; BUYYA, R.; MAHESWARAN., M. A taxonomy and survey of grid resource management systems. *Software Practice and Experiance*, v. 32, p. 135–164, Feb 2002.

[15] TOPCUOUGLU, H.; HARIRI, S.; WU, M. you. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 13, n. 3, p. 260–274, 2002. ISSN 1045-9219.

[16] CULLER, D. E. et al. LogP: Towards a realistic model of parallel computation. In: *Principles Practice of Parallel Programming*. [s.n.], 1993. p. 1–12. Disponível em: <citeseer.ist.psu.edu/culler93logp.html>.

[17] MENDES., H. *Hlog*P*: Um modelo de escalonamento para a execução de aplicações* MPI *em grades computacionais*. 2004. Tese de Mestrado, Instituto de Computação, Universidade Federal Fluminense.

[18] ALEXANDROV, A. et al. LogGP: Incorporating long messages into the LogP model. In: *SPAA 95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM Press, 1995. p. 95–105. ISBN 0-89791-717-0.

[19] MORITZ, C. A.; FRANK, M. I. LoGPC: Modeling network contention in message-passing programs. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 12, n. 4, p. 404–415, 2001. ISSN 1045-9219.

[20] VIANNA, L. S.; DRUMMOND, L. M.; OCHI, L. S. Um algoritmo GRASP para o problema de escalonamento de tarefas utilizando o modelo LogP. *XXXII Simpoósio Brasileiro de Pesquisa Operacional*, p. 295–313, 2000. ISSN 1518-1731. Disponível em: <labic.ic.uff.br/conteudo/artigos/sbpo-leo-2000.pdf>.

[21] CARDOSO, V. E. F. R. D. F. Estratégias para minimizar efeitos adversos das sobrecargas de comunicações em ambientes computacionais heterogêneos sob o modelo LogP. In: *III workshop de Grade Computacional e Aplicações*. [S.l.: s.n.], 2005.

[22] KWOK, Y.; AHMAD., I. Benchmarking the task graph scheduling algorithms. In: *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 1998. p. 531.

[23] THAIN, D.; TANNENBAUM, T.; LIVNY, M. Condor and the grid. In: BERMAN, F.; FOX, G.; HEY, T. (Ed.). *Grid Computing: Making the Global Infrastructure a Reality*. [S.l.]: John Wiley & Sons Inc., 2002.

[24] BUYYA, R.; MURSHED, M. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, Volume 14, p. 13–15, May 2002. Disponível em: <citeseer.ist.psu.edu/murshed02gridsim.html>.

[25] BELL, W. et al. Optorsim - a grid simulator for studying dynamic data replication strategies. *Journal of High Performance Computing Applications, 17(4)*, 2003. Disponível em: <citeseer.comp.nus.edu.sg/bell03optorsim.html>.

[26] LAMEHAMEDI, H. et al. Data replication strategies in grid environments. *Proc. Of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'02), 2002.*, 2002. Disponível em: <citeseer.ist.psu.edu/lamehamedi02data.html>.

[27] AIDA, K. et al. Performance evaluation model for scheduling in global computing systems. *The International Journal of High Performance Computing Applications*, v. 14, n. 3, p. 268–279, Fall 2000. Disponível em: <citeseer.ist.psu.edu/aida00performance.html>.

[28] SONG, H. J. et al. The Microgrid: a scientific tool for modeling computational grids. *Supercomputing*, 2000. Disponível em: <citeseer.ist.psu.edu/song00microgrid.html>.

[29] CASANOVA, H. Simgrid: A toolkit for the simulation of application scheduling. *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, 2001. Disponível em: <citeseer.ist.psu.edu/casanova01simgrid.html>.

[30] PHATANAPHEROM, S.; UTHAYOPAS, P.; KACHITVICHYANUKUL, V. Dynamic scheduling ii: fast simulation model for grid scheduling using hypersim. In: *WSC '03: Proceedings of the 35th conference on Winter simulation*. [S.l.]: Winter Simulation Conference, 2003. p. 1494–1500. ISBN 0-7803-8132-7.

[31] GMBH., I. *Intel* T*race* C*ollector 6.0.1.0 - User's Guide 6.0.1.0.1*. 2005. Disponível em: <http://www.calmip.cict.fr/spip/IMG/pdf/Intel_Trace_Collector_Users_Guide.pdf>.

[32] SEIDL., S. VTF3 - *A Fast* V*ampir* T*race File Low-Level Management Library*. Nov 2003. Technical Report ZHR-R-x0304 - Dresden University of Technology.

[33] BAILEY, D.; BARSCZ, E.; BARTON, J. *The* NAS *Parallel Benchmarks*. 1994. NAS Technical ReportRNR-94-007, NASA Ames Research Center.

[34] CORMEN, T. H. et al. *Introduction to Algorithms*. 2. ed. [S.l.]: The MIT Press, 2001.

[35] FERREIRA, L.; BERSTIS, V.; ARMSTRONG, J. *Introduction to Grid Computing with* G*lobus*. Sep 2003. ISBN 0738499889. ibm.com/redbooks. Disponível em: <http://www.redbooks.ibm.com/abstracts/sg246895.html>.

[36] MPICH-G2. *The* G*lobus* A*lliance*. Aug 2006. Disponível em: <http://www.globus.org/grid_software/computation/mpich-g2.php>.

[37] BADER, D. A.; MADDURI, K. GTG*raph: A Synthetic Graph Generator Suite*. Feb 2006. Disponível em: <http://www-static.cc.gatech.edu/ kamesh/GTgraph/>.

[38] RIOS, R. A. *Escalonamento adaptativo em grades computacionais*. Mar 2007. (Qualificação de Mestrado em Redes e Sistemas Distribuídos) Departamento de Computação, Universidade Federal de São Carlos, São Carlos.

# 10. Anexo A - Código que simula o *grid* computacional usado nesse trabalho

```
1  import gridsim.*;
2  import gridsim.net.*;
3  import java.util.*;
4
5
6  public class Program
7  {
8      /**
9       * Creates main() to run this example
10      */
11     public static final int SEND_PKT = 900;
12     public static final int SEND2_PKT = 901;
13     public static final int RECV_PKT = 902;
14     public static final int BCAST_SIZE = 69206056;
15     public static final int ANSWER_SIZE = 2097664;
16
17     public static void main(String[] args)
18     {
19         System.out.println("Starting network example ...");
20
21         try
22         {
23             int num_user = 1;   // number of grid users
24             Calendar calendar = Calendar.getInstance();
25
26             // a flag that denotes whether to trace GridSim events or not.
27             boolean trace_flag = false;
28
29             // Initialize the GridSim package
30             System.out.println("\n
                   ------------------------------------------------------");
```

```
31          System.out.println("Initializing GridSim package");
32          System.out.println
               ("-------------------------------------------------");
33
34          GridSim.init(num_user, calendar, trace_flag);
35
36          // Create a new GIS entity
37          NewGIS gis = new NewGIS("NewGIS");
38
39          // You need to call this method before the start of simulation
40          GridSim.setGIS(gis);
41
42          System.out.println("\n
               ---------------------------------------------------");
43          System.out.println("Creating Grid Resources");
44          System.out.println
               ("---------------------------------------------------");
45
46          int totalResource = 23;  // total number of Grid resources
47
48          ArrayList resList = new ArrayList(totalResource);
49
50          // Creating NODE_0 a NODE_8 Resources
51          int rating = 6769;        // rating of each PE in MIPS - Intel
               Core 2 X6800
52          int totalPE = 4;         // total number of PEs for each Machine
53          int totalMachine = 1;    // total number of Machines
54          double baud_rate = 1000000000;
55
56          String resName = "ufscar";
57          int i=0;
58
59          NewGridResource resNode = null;
60
61          for(i = 0;i < 9;i++)
62          {
63             resNode = createGridResource(resName+i, rating, totalMachine
                  , totalPE, baud_rate);
64
65             // add a resource name into an array
66             resList.add(resNode);
```

```
67              }
68
69              // Creating UFSCAR9 Resource
70              rating = 57063;      // rating of each PE in MIPS - AMD ATHLON
                    FX-57
71              totalPE = 1;         // total number of PEs for each Machine
72              totalMachine = 1;    // total number of Machines
73
74              resName = "ufscar9";
75              NewGridResource res = createGridResource(resName, rating,
                    totalMachine, totalPE, baud_rate);
76
77              // add a resource name into an array
78              resList.add(res);
79
80              // Creating USP_0 a USP_5 Resources
81              rating = 25600;      // rating of each PE in MIPS - IBM Cell One
                     SPE
82              totalPE = 1;         // total number of PEs for each Machine
83              totalMachine = 1;    // total number of Machines
84
85              resName = "usp";
86
87              for(i = 0;i < 6;i++)
88              {
89                  res = createGridResource(resName+i, rating, totalMachine,
                        totalPE, baud_rate);
90
91                  // add a resource name into an array
92                  resList.add(res);
93              }
94
95              // Creating UNICAMP0 Resource
96              rating = 35200;      // rating of each PE in MIPS - Sony PS3
97              totalPE = 1;         // total number of PEs for each Machine
98              totalMachine = 1;    // total number of Machines
99
100             resName = "unicamp0";
101             res = createGridResource(resName, rating, totalMachine, totalPE
                    , baud_rate);
102
```

```
103             // add a resource name into an array
104             resList.add(res);
105
106             // Creating Unicamp_1 Unicamp_7 Resources
107             rating = 28300;      // rating of each PE in MIPS - Intel Core 2
                    X6800
108             totalPE = 1;         // total number of PEs for each Machine
109             totalMachine = 1;    // total number of Machines
110
111             resName = "unicamp";
112
113             for(i = 1;i < 8;i++)
114             {
115                res = createGridResource(resName+i, rating, totalMachine,
                       totalPE, baud_rate);
116
117                // add a resource name into an array
118                resList.add(res);
119             }
120
121
122             System.out.println("\n
                   ----------------------------------------------------");
123             System.out.println("Creating Grid Users");
124             System.out.println
                   ("----------------------------------------------------");
125
126             // number of Gridlets that will be sent to the resource
127             int totalGridlet = 4;
128             baud_rate = 1000000000;     // bits/sec
129             double propDelay = 0.23;    // propagation delay in millisecond
130             int mtu = 1500;             // max. transmission unit in byte
131
132             // create users
133             ArrayList userList = new ArrayList(num_user);
134             for (i = 0; i < num_user; i++)
135             {
136                 // if trace_flag is set to "true", then this experiment
                       will
137                 // create User_i.csv where i = 0 ... (num_user-1)
138                 NetUser user = new NetUser("User_0", totalGridlet,
```

```
                baud_rate,
139                                     propDelay, mtu, trace_flag);
140
141             // add a user into a list
142             userList.add(user);
143         }
144
145         System.out.println("\n
                ------------------------------------------------------");
146         System.out.println("Building the network topology ");
147         System.out.println
                ("------------------------------------------------------");
148
149         Router r1 = new RIPRouter("router1", trace_flag);   // router
                ufscar
150         Router r2 = new RIPRouter("router2", trace_flag);   // router
                usp
151         Router r3 = new RIPRouter("router3", trace_flag);   // router
                unicamp
152
153         // connect all user entities with r1
154         NetUser obj = null;
155         for (i = 0; i < userList.size(); i++)
156         {
157             // A First In First Out Scheduler is being used here.
158             // SCFQScheduler can be used for more fairness
159             FIFOScheduler userSched = new FIFOScheduler("NetUserSched_
                    "+i);
160             obj = (NetUser) userList.get(i);
161             r1.attachHost(obj, userSched);
162         }
163         System.out.println("User connected to R1");
164
165         // connect all UFSCar resource entities with router1
166         GridResource resObj = null;
167
168         for (i = 0; i < 10; i++)
169         {
170             FIFOScheduler resSched = new FIFOScheduler("GridResSched_"+
                    i);
171             resObj = (GridResource) resList.get(i);
```

```
172                 r1.attachHost(resObj, resSched);
173             }
174             System.out.println("UFSCar resources connected to R1");
175 
176             // connect all USP resource entities with router2
177             resObj = null;
178 
179             for (i = 10; i < 16; i++)
180             {
181                 FIFOScheduler resSched = new FIFOScheduler("GridResSched_"+
                        i);
182                 resObj = (GridResource) resList.get(i);
183                 r2.attachHost(resObj, resSched);
184             }
185             System.out.println("USP resources connected to R2");
186 
187 
188             // connect all Unicamp resource entities with router3
189             resObj = null;
190 
191             for (i = 16; i < 24; i++)
192             {
193                 FIFOScheduler resSched = new FIFOScheduler("GridResSched_"+
                        i);
194                 resObj = (GridResource) resList.get(i);
195                 r3.attachHost(resObj, resSched);
196             }
197             System.out.println("Unicamp resources connected to R3");
198 
199             // then connect router1 to router2 with 1 Gb/s connection
200             baud_rate = 1000000000;
201             Link link = new SimpleLink("r1_r2_link", baud_rate, propDelay
                    *4, mtu);
202 
203             FIFOScheduler r1Sched = new FIFOScheduler("r1_Sched");
204             FIFOScheduler r2Sched = new FIFOScheduler("r2_Sched");
205 
206             // attach r2 to r1
207             r1.attachRouter(r2, link, r1Sched, r2Sched);
208 
209             // then connect router1 to router3 with 1 Gb/s connection
```

```
210             baud_rate = 1000000000;
211             link = new SimpleLink("r1_r3_link", baud_rate, propDelay*4, mtu
                    );
212
213             FIFOScheduler r3Sched = new FIFOScheduler("r3_Sched");
214
215             // attach r3 to r1
216             r1.attachRouter(r3, link, r1Sched, r3Sched);
217
218             // then connect router2 to router3 with 1 Gb/s connection
219             baud_rate = 1000000000;
220             link = new SimpleLink("r2_r3_link", baud_rate, propDelay*4, mtu
                    );
221
222             // attach router3 to router2
223             r2.attachRouter(r3, link, r2Sched, r3Sched);
224
225             // printing table routing of routers
226             r1.printRoutingTable();
227             r2.printRoutingTable();
228             r3.printRoutingTable();
229
230             System.out.println("\n
                    ----------------------------------------------------");
231             System.out.println("Starting Simulation");
232             System.out.println
                    ("----------------------------------------------------");
233
234             GridSim.startGridSimulation();
235
236         }
237         catch (Exception e)
238         {
239             e.printStackTrace();
240             System.out.println("Unwanted errors happen");
241         }
242     }
243
244
245         private static NewGridResource createGridResource(String name, int
                peRating, int totalMachine, int       totalPE, double
```

```
            baud_rate_host)
246     {
247         MachineList mList = new MachineList();
248
249         int rating = peRating;
250         for (int i = 0; i < totalMachine; i++)
251         {
252             PEList peList = new PEList();
253
254             for (int k = 0; k < totalPE; k++)
255             {
256                 peList.add( new PE(k, rating) );
257             }
258
259             mList.add( new Machine(i, peList) );
260         }
261
262         String arch = "Dual Core";       // system architecture
263         String os = "Linux";             // operating system
264         double time_zone = 0.0;          // time zone this resource located
265         double cost = 3.0;               // the cost of using this resource
266
267         ResourceCharacteristics resConfig = new ResourceCharacteristics(
268                 arch, os, mList, ResourceCharacteristics.TIME_SHARED,
269                 time_zone, cost);
270
271         double baud_rate = baud_rate_host;           // communication speed
272         double delay = 0.15;                         // propagation delay
            in millisecond
273         int mtu = 1500;
274
275         long seed = 11L*13*17*19*23+1;
276         double peakLoad = 0.0;       // the resource load during peak hour
277         double offPeakLoad = 0.0;    // the resource load during off-peak
            hr
278         double holidayLoad = 0.0;    // the resource load during holiday
279
280         LinkedList Weekends = new LinkedList();
281         Weekends.add(new Integer(Calendar.SATURDAY));
282         Weekends.add(new Integer(Calendar.SUNDAY));
283
```

```
284             LinkedList Holidays = new LinkedList();
285
286             NewGridResource gridRes = null;
287
288             try {
289                     gridRes = new NewGridResource(name,
290                             new SimpleLink(name + "_link", baud_rate, delay,
                                    mtu),
291                             seed, resConfig, peakLoad, offPeakLoad, holidayLoad
                                    ,
292                             Weekends, Holidays);
293
294             }
295             catch (Exception e) {
296                 e.printStackTrace();
297             }
298
299             System.out.println("Creates one Grid resource with name = " + name
                    + " (" + totalPE + " processor X " + peRating + " MIPS )" );
300
301             return gridRes;
302         }
303
304     } // end class
```

```
1  import java.util.*;
2  import gridsim.*;
3  import gridsim.net.*;
4  import gridsim.util.SimReport;
5  import eduni.simjava.Sim_port;
6  import eduni.simjava.Sim_event;
7
8  class NetUser extends GridSim
9  {
10     private int myId_;                    // my entity ID
11     private String name_;                 // my entity name
12     private GridletList list_;            // list of submitted Gridlets
13     private GridletList receiveList_;     // list of received Gridlets
14     private SimReport report_;            // logs every events
15
16     NetUser(String name, int totalGridlet, double baud_rate, double delay,
17             int MTU, boolean trace_flag) throws Exception
18     {
19         super( name, new SimpleLink(name+"_link",baud_rate,delay, MTU) );
20
21         this.name_ = name;
22         this.receiveList_ = new GridletList();
23         this.list_ = new GridletList();
24
25         // creates a report file
26         if (trace_flag == true) {
27             report_ = new SimReport(name);
28         }
29
30         // Gets an ID for this entity
31         this.myId_ = super.getEntityId(name);
32         System.out.println("Creating a grid user entity with name = " +
33               name + ", and id = " + this.myId_);
34
35         // Creates a list of Gridlets or Tasks for this grid user
36         System.out.println(name + ":Creating " + totalGridlet +" Gridlets")
               ;
37         this.createGridlet(myId_, totalGridlet);
38
39     }
40
```

```
41
42      public void body()
43      {
44          // wait for a little while for about 3 seconds.
45          // This to give a time for GridResource entities to register their
46          // services to GIS (GridInformationService) entity.
47          super.gridSimHold(1.0);
48
49          LinkedList resList = super.getGridResourceList();
50
51          // initialises all the containers
52          int totalResource = resList.size();
53          int resourceID[] = new int[totalResource];
54          String resourceName[] = new String[totalResource];
55
56          // a loop to get all the resources available
57          System.out.println("\n");
58          int i = 0;
59          for (i = 0; i < totalResource; i++)
60          {
61              // Resource list contains list of resource IDs
62              resourceID[i] = ( (Integer) resList.get(i) ).intValue();
63
64              // get their names as well
65              resourceName[i] = GridSim.getEntityName( resourceID[i] );
66
67              System.out.println("Resource " + i + " = " + resourceName[i]);
68          }
69          System.out.println("\n");
70
71
72          Random random = new Random();
73          int index = 0;
74          int[] resTask= new int[5];
75
76          // sends all the Gridlets
77          Gridlet gl = null;
78          boolean success;
79
80          for (i = 0; i < list_.size(); i++)
81          {
```

```
82
83              index = random.nextInt(totalResource);
84              gl = (Gridlet) list_.get(i);
85              System.out.println(name_ + ": Sending Gridlet #" + i + " to " +
                    resourceName[index]+ " at time "+ GridSim.clock());
86
87              success = super.gridletSubmit(gl, resourceID[index]);
88
89              switch(i)
90              {
91  case 0:
92  resTask[0]=index;
93  break;
94  case 1:
95  resTask[1]=index;
96  break;
97  case 2:
98  resTask[2]=index;
99  break;
100 case 3:
101 resTask[3]=index;
102 break;
103 case 4:
104 resTask[4]=index;
105 break;
106 case 5:
107 resTask[5]=index;
108 break;
109              }
110
111         }
112
113
114         // Sending messages between tasks
115         super.gridSimHold(2);
116
117         Integer Destiny = null;
118
119         for (i = 1; i < list_.size(); i++)
120         {
121           if ( resourceName[resTask[i]] != resourceName[resTask[0]] )
```

```
122             {
123             Destiny = new Integer(getEntityId(resourceName[resTask[i]]));
124
125             System.err.println(this.name_ + ": Advising " +
126                     resourceName[resTask[0]] + " to send a packet to "+
                            resourceName[resTask[i]] + " at time "+ GridSim.clock
                            ());
127
128             super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.
                    SEND_PKT, new IO_data(Destiny,1,resourceID[resTask[0]]));
129             super.gridSimHold(0.1);
130             }
131         }
132
133         for (i = 1; i < list_.size(); i++)
134         {
135             if ( resourceName[resTask[i]] != resourceName[resTask[0]] )
136             {
137             Destiny = new Integer(getEntityId(resourceName[resTask[0]]));
138
139             System.err.println(this.name_ + ": Advising " +
140                     resourceName[resTask[i]] + " to send a packet to "+
                            resourceName[resTask[0]] + " at time "+ GridSim.clock
                            ());
141
142             super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.
                    SEND2_PKT, new IO_data(Destiny,1,resourceID[resTask[i]]));
143             super.gridSimHold(0.1);
144             }
145         }
146
147         // receives the gridlet back
148         for (i = 0; i < list_.size(); i++)
149         {
150             gl = (Gridlet) super.receiveEventObject();  // gets the Gridlet
151             receiveList_.add(gl);   // add into the received list
152
153             System.out.println(name_ + ": Receiving Gridlet #" +
154                     gl.getGridletID() + " at time = " + GridSim.clock() );
155         }
156
```

```
157             // shut down I/O ports
158             shutdownUserEntity();
159             terminateIOEntities();
160
161             // don't forget to close the file
162             if (report_ != null) {
163                 report_.finalWrite();
164             }
165
166             System.out.println("\n\n"+ this.name_ + ": sending and receiving of
                    Gridlets" +
167                     " complete at " + GridSim.clock() + " seconds \n\n" );
168         }
169
170
171
172         public GridletList getGridletList() {
173             return receiveList_;
174         }
175
176         private void createGridlet(int userID, int numGridlet)
177         {
178
179             // Creates a Gridlet
180             // Gridlet(int id, double length, long file_size, long output_size)
181
182     Gridlet gl = new Gridlet(0, 1.70991e+06, 0, 0);
183     gl.setUserID(userID);
184     this.list_.add(gl);
185     gl = new Gridlet(1, 1.70992e+06, 0, 0);
186     gl.setUserID(userID);
187     this.list_.add(gl);
188     gl = new Gridlet(2, 1.70992e+06, 0, 0);
189     gl.setUserID(userID);
190     this.list_.add(gl);
191     gl = new Gridlet(3, 1.70992e+06, 0, 0);
192     gl.setUserID(userID);
193     this.list_.add(gl);
194     gl = new Gridlet(4, 1.70992e+06, 0, 0);
195     gl.setUserID(userID);
196     this.list_.add(gl);
```

```
197 gl = new Gridlet(5, 1.70992e+06, 0, 0);
198 gl.setUserID(userID);
199 this.list_.add(gl);
200     }
201 } // end class
```

```
1
2  import gridsim.*;
3  import eduni.simjava.*;
4  import gridsim.net.*;
5  import java.util.*;
6
7
8  public class NewGridResource extends GridResource
9  {
10       * @param name       a Grid Resource name
11       * @param peRating   rating of each PE
12       * @param totalMachine   total number of Machines
13       * @param totalPE        total number of PEs for each Machine
14       */
15       public NewGridResource(String name, Link link, long seed,
             ResourceCharacteristics resource, double peakLoad, double
             offPeakLoad, double relativeHolidayLoad, LinkedList weekends,
             LinkedList holidays) throws Exception
16       {
17            super(name, link, seed, resource, peakLoad, offPeakLoad,
                  relativeHolidayLoad, weekends, holidays);
18       }
19
20      protected void processOtherEvent(Sim_event ev)
21      {
22          try
23          {
24              Integer obj = (Integer) ev.get_data();
25
26              // get the sender name
27              String name = GridSim.getEntityName( obj.intValue() );
28              switch ( ev.get_tag() )
29              {
30                  case Program.SEND_PKT:
31                      System.err.println(super.get_name()
32                              + ": sending SEND_PKT to " + name +
33                              " with " + Program.BCAST_SIZE + " bytes at time
                                    " + GridSim.clock());
34
35                      super.send(super.output, GridSimTags.SCHEDULE_NOW,
                            Program.RECV_PKT,
```

```
36                      new IO_data(new Integer(super.get_id()), Program.
                            BCAST_SIZE, obj.intValue()) );
37                      break;
38
39                  case Program.SEND2_PKT:
40                      System.err.println(super.get_name()
41                              + ": sending SEND_PKT to " + name +
42                              " with " + Program.ANSWER_SIZE + " bytes at
                                    time " + GridSim.clock());
43
44                      super.send(super.output, GridSimTags.SCHEDULE_NOW,
                            Program.RECV_PKT,
45                      new IO_data(new Integer(super.get_id()), Program.
                            ANSWER_SIZE, obj.intValue()) );
46                      break;
47
48
49                  case Program.RECV_PKT:
50                      System.err.println(super.get_name()
51                              + ": received RECV_PKT from " + name +
52                              " at time " + GridSim.clock());
53                      break;
54
55                  default:
56                      break;
57              }
58          }
59          catch (ClassCastException c) {
60              System.out.println(super.get_name() +
61                      ".processOtherEvent(): Exception occurs.");
62          }
63
64      }
65
66      protected void registerOtherEntity()
67      {
68          int SIZE = 12;  // size of Integer object incl. overhead
69
70          // get the GIS entity ID
71          int gisID = GridSim.getGridInfoServiceEntityId();
72
```

```
73          // get the GIS entity name
74          String gisName = GridSim.getEntityName(gisID);
75
76          // register SEND_PKT tag to the GIS entity
77          System.out.println(super.get_name() + ".registherOtherEntity(): " +
78                  "register SEND_PKT tag to " + gisName +
79                  " at time " + GridSim.clock());
80
81          super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.SEND_PKT
                ,
82                  new IO_data(new Integer(super.get_id()), SIZE, gisID) );
83
84
85          // register SEND2_PKT tag to the GIS entity
86          System.out.println(super.get_name() + ".registherOtherEntity(): " +
87                  "register SEND2_PKT tag to " + gisName +
88                  " at time " + GridSim.clock());
89
90          super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.
                SEND2_PKT,
91                  new IO_data(new Integer(super.get_id()), SIZE, gisID) );
92
93
94          // register RECV_PKT tag to the GIS entity
95          System.out.println(super.get_name() + ".registherOtherEntity(): " +
96                  "register RECV_PKT tag to " + gisName +
97                  " at time " + GridSim.clock());
98
99          super.send(super.output, GridSimTags.SCHEDULE_NOW, Program.RECV_PKT
                ,
100                 new IO_data(new Integer(super.get_id()), SIZE, gisID) );
101
102     }
103
104 } // end class
```

# 11. Anexo B - *Script* que realiza a análise do *tracefile*

```
#/bin/bash
#
clear
#
for arquivo in send.txt recv2.txt recv.txt $arqsaida downto.txt noact.txt;
    do
  if [ -f $arquivos ]; then
     rm -f $arquivo
  fi
done
#
echo "-----------------------------------------------------------------"
echo " Script que gera o arquivo de entrada $arqsaida  "
echo " para o software graphviz da AT&T"
echo " "
echo " Arquivos gerados:"
echo "   [inputFile].dot "
echo "-----------------------------------------------------------------"
#
echo
 echo "Digite o nome do arquivo de TRACE em formato ASCII: "
read arq
#
arqsaida=$arq.dot
#
if [[ ! -f $arq ]]; then
     echo "Arquivo $arq nao encontrado"
     exit
fi
# Obtendo a quantia de nos
#
ncpus=`awk -F" " '{if ($1=="NCPUS") print $2}' $arq`
```

```
32 echo $ncpus
33 #
34 # Criando arquivo de saida
35 #
36 echo "digraph G {" >> $arqsaida
37 #
38 # Calculando tempo de processamento e cadastrando nos no arquivo saida
39 #
40 more $arq | grep "NOACT" > noact.txt
41 #
42 usercode=`more $arq | grep "User_Code" | awk -F" " '{print $2}'`
43 more $arq | grep "DOWNTO "$usercode > downto.txt
44 #
45 i=1
46 while (( i <= $ncpus ))
47 do
48   final=`awk 'END{print NR}' noact.txt`
49   j=1
50   while (( j <= $final ))
51   do
52      cmd=`sed -e $j'!d' noact.txt`
53      j=`expr $j + 1`
54      no=`echo $cmd | awk -F" " '{print $4}'`
55      if [ $no == $i ]; then
56         tempof=`echo $cmd | awk -F" " '{print $1}'`
57      fi
58   done
59 #
60   final=`awk 'END{print NR}' downto.txt`
61   j=1
62   while (( j <= $final ))
63   do
64      cmd=`sed -e $j'!d' downto.txt`
65      j=`expr $j + 1`
66      no=`echo $cmd | awk -F" " '{print $4}'`
67      if [ $no == $i ]; then
68         tempoi=`echo $cmd | awk -F" " '{print $1}'`
69      fi
70   done
71   tempot=`expr $tempof - $tempoi`
72 #
```

```
73    echo '"n'$i'" [ label = "'$i'-'$tempot'"];' >> $arqsaida
74    i=`expr $i + 1`
75  done
76  #
77  # Separando os SEND, os RECV e os GLOBALOP
78  #
79  more $arq | grep SEND > send.txt
80  more $arq | grep RECV > recv.txt
81  more $arq | grep "GLOBALOP " > global.txt
82  #
83  # Cadastrando as arestas no arquivo de saida
84  #
85  final=`awk 'END{print NR}' send.txt`
86  i=1
87  while (( i <= $final ))
88  do
89    cmd=`sed -e $i'!d' send.txt`
90  #  echo $cmd
91    i=`expr $i + 1`
92    tempoS=`echo $cmd | awk -F" " '{print $1}'`
93    noS=`echo $cmd | awk -F" " '{print $6}'`
94    noR=`echo $cmd | awk -F" " '{print $8}'`
95    len=`echo $cmd | awk -F" " '{print $10}'`
96  #
97    finalRecv=`awk 'END{print NR}' recv.txt`
98    j=1
99    achou=0
100 #
101 # Obtendo tempo da recepcao da mensagem
102 #
103   while (( j <= $finalRecv && achou == 0 ))
104   do
105     cmdRecv=`sed -e $j'!d' recv.txt`
106     noRecvS=`echo $cmdRecv | awk -F" " '{print $8}'`
107     noRecvR=`echo $cmdRecv | awk -F" " '{print $6}'`
108     if [[ $noS == $noRecvS && $noR == $noRecvR ]]; then
109  #       echo $cmdRecv
110  #       echo
111         tempoR=`echo $cmdRecv | awk -F" " '{print $1}'`
112         sed $j'd' recv.txt > recv2.txt
113         mv recv2.txt recv.txt
```

```
114 #
115         tempoT=`expr $tempoR - $tempoS`
116         echo '"n'$noS'" -> "n'$noR'" [ label = "'$len'-'$tempoT'" ];' >>
                $arqsaida
117 #        echo $tempoT
118         achou=1
119       fi
120       j=`expr $j + 1`
121     done
122 done
123 #
124 # Tratando comunicacoes coletivas
125 #
126 final=`awk 'END{print NR}' global.txt`
127 i=1
128 while (( i <= $final ))
129 do
130   cmd=`sed -e $i'!d' global.txt`
131 #  echo $cmd
132   i=`expr $i + 1`
133   tipoG=`echo $cmd | awk -F" " '{print $3}'`
134   no1=`echo $cmd | awk -F" " '{print $5}'`
135   no2=`echo $cmd | awk -F" " '{print $7}'`
136   qSendG=`echo $cmd | awk -F" " '{print $8}'`
137   qRecG=`echo $cmd | awk -F" " '{print $9}'`
138   tempoG=`echo $cmd | awk -F" " '{print $10}'`
139 #
140   if [ $qSendG -le 0 ]; then
141 #
142 # BARRIER
143 #
144       if [ $qRecG -le 0 ]; then
145         echo '"n'$no1'" -> "n'$no2'" [ label = "0-'$tempoG'" ] [style=
                dashed,color=red];' >> $arqsaida
146 #
147 # Possivelmente um BCAST / SCATTER
148 #
149       else
150         echo '"n'$no2'" -> "n'$no1'" [ label = "'$qRecG'-'$tempoG'" ] [
                style=dashed];' >> $arqsaida
151       fi
```

```
152   else
153  #
154  # Possivelmente um GATHER /SCATTER
155  #
156         if [ $qRecG -le 0 ]; then
157            echo '"n'$no1'" -> "n'$no2'" [ label = "'$qSendG'-'$tempoG'" ] [
                   style=dashed];' >> $arqsaida
158         else
159         {
160  #
161  # ALLGATHER
162  #
163            echo '"n'$no1'" -> "n'$no2'" [ label = "'$qSendG'-'$tempoG'" ] [
                   style=dashed];' >> $arqsaida
164            echo '"n'$no2'" -> "n'$no1'" [ label = "'$qRecG'-'$tempoG'" ] [
                   style=dashed];' >> $arqsaida
165         }
166         fi
167   fi
168  #
169  done
170  #
171  echo } >> $arqsaida
```

# 12. Anexo C - *Script* que sintetiza os dados coletados

```
#/bin/bash
#
clear
#
for arquivo in Send* ; do
  if [ -f $arquivos ]; then
     rm -f $arquivo
  fi
done
#
echo "-------------------------------------------------------------------"
echo " Script que resume nro de comunicacoes de um DAG"
echo " "
echo " Arquivos gerados:"
echo "  [inputFile].v2 - arquivo DOT resumido"
echo "  [inputFile].sched - arquivo para escalonamento"
echo "-------------------------------------------------------------------"
#
echo
echo "Digite o nro de PROCESSOS: "
read nproc
echo " "
echo "Digite o nome do arquivo .DOT: "
read arq
#
# Obtendo a quantia de nos
#
ncpus=$nproc
echo $ncpus
#
# Criando arquivos de saida
#
```

```
33 dagnovo=$arq".v2"
34 dagsched=$arq".sched"
35 echo $ncpus > $dagsched
36 linhas=`expr $ncpus + 1`
37 echo $linhas
38 #
39 # Acertar os nros dos nos - deve comecar em 0 ao inves de 1
40 #
41 head -$linhas $arq | sed 's/"//g;s/->//g;s/n//g;s/\[//g;s/\]//g;s/label//g;
       s/=//g;s/styledashed//g;s/\;//g;s/\-/ /g' | awk -F" " '{print "\"n"$1
       -1"\" [ label = \"n"$1-1"-"$3"\"];"}' > $dagnovo
42
43 head -$linhas $arq | sed 's/"//g;s/->//g;s/n//g;s/\[//g;s/\]//g;s/label//g;
       s/=//g;s/styledashed//g;s/\;//g;s/\-/ /g' | awk -F" " '{print $1-1" "$3
       }' >> $dagsched
44 #
45 i=1
46 while (( i <= $ncpus ))
47 do
48   j=1
49   while (( j <= $ncpus ))
50   do
51      cmd="\"n"$i\"" -> \"n"$j"\""
52      arqsaida="Send"$i"R"$j".txt"
53 #     echo $arqsaida
54      more $arq | grep "$cmd" >> $arqsaida
55 #    Filtrando os dados
56      final=`awk 'END{print NR}' $arqsaida`
57      if [ $final != "0" ]; then
58       qtdetempo=0
59       qtdedados=0
60       k=1
61       while (( k <= $final ))
62       do
63         cmd=`sed -e $k'!d' $arqsaida`
64         k=`expr $k + 1`
65         dados=`echo $cmd | awk -F" " '{print $7}'`
66         tempo=`echo $dados | sed 's/\"//g' | awk -F- '{print $2}'`
67         dadosS=`echo $dados | sed 's/\"//g' | awk -F- '{print $1}'`
68         qtdetempo=`expr $qtdetempo + $tempo`
69         qtdedados=`expr $qtdedados + $dadosS`
```

```
70          done
71  #       Subtrair 1 pois as tarefas devem comecar em zero
72          echo "\"n"‘expr $i - 1‘\"" -> \"n"‘expr $j - 1‘"\" [ label=\""$final
                "-"$qtdedados"-"$qtdetempo"\" ];" >> $dagnovo
73          echo ‘expr $i - 1‘" "‘expr $j - 1‘" "$qtdedados" "$qtdetempo >>
                $dagsched
74        fi
75  #
76        j=‘expr $j + 1‘
77        rm $arqsaida
78    done
79    i=‘expr $i + 1‘
80  done
81  #
82  echo "}" >> $dagnovo
```

# 13. Anexo D - Código utilizado para remover ciclos dos grafos

```
1  /*
2   * Universidade Federal de Sao Carlos
3   * Departamento de Computacao
4   * Progrid - www.gsdr.dc.ufscar.br/progrid
5   * Programacao Paralela e Distribuida
6   * Copyleft (L) 2006, Daniele S. Jacinto, Ricardo A. Rios e Helio C.
        Guardia
7   */
8
9  /*
10  * Programa : grafo.c
11  * Descricao: Implementacao da Estrutura de Dados Grafo
12  */
13
14 #include "grafo.h"
15
16 /*
17 * HDR = Helio, Dani e Ricardo
18 * HDR = HDR is a DFS to Remove cycles
19 */
20
21 int
22 hdr (node * n)
23 {
24   node *ntmp;
25   int result = 0;
26   arc *atmp = (arc *) malloc (sizeof (arc));
27   atmp = _ahead;
28   n->color = CINZA;
29   while (atmp != NULL)
30     {
31       if (atmp->nodeout == n)
```

```
32          {
33            if ((atmp->nodein)->color == BRANCO)
34              result += hdr (atmp->nodein);
35            else if ((atmp->nodein)->color == CINZA)
36              {
37                if ((ntmp = finder ((atmp->nodein)->id + 1000)) == NULL)
38                  {
39                    ntmp = (node *) malloc (sizeof (node));
40                    ntmp->noden = (node *) malloc (sizeof (node));
41
42                    ntmp->id = ((atmp->nodein)->id) + 1000;
43                    ntmp->cost = (atmp->nodein)->cost;
44                    ntmp->color = BRANCO;
45                    ntmp->noden = NULL;
46                    add_node (ntmp);
47                  }
48                atmp->nodein = ntmp;
49                result += hdr (atmp->nodein);
50                ++result;
51              }
52          }
53        atmp = atmp->arcn;
54      }
55    n->color = PRETO;
56
57    atmp = NULL;
58    free (atmp);
59
60    return result;
61  }
62
63
64  node * finder (int i)
65  {
66    node *tmp = (node *) malloc (sizeof (node));
67    tmp = _nhead;
68    while (tmp != NULL)
69      {
70        if (i == tmp->id)
71          {
72            return tmp;
```

```
73              }
74          tmp = tmp->noden;
75        }
76
77      tmp = NULL;
78      free (tmp);
79
80      return NULL;
81  }
```

# 14. Anexo E - Código que gera um modelo arquitetutal

```
1  /*
2   * Universidade Federal de Sao Carlos
3   * Departamento de Computacao
4   * Progrid - www.gsdr.dc.ufscar.br/progrid
5   * Programacao Paralela e Distribuida
6   * Copyleft (L) 2006, Daniele S. Jacinto, Ricardo A. Rios e Helio C.
        Guardia
7   */
8
9  /*
10  * Programa : arch-model.c
11  * Descricao: implementacao das funcoes definidas no 'header'
12  * Ultima alteracao: 02/11/06 - Daniele
13  * Ultima alteracao: 08/03/07 - rios
14  */
15
16 #include "arch-model.h"
17
18 void
19 model_arch(char *fgrid,char *fcluster, char *frates, char *fcostcomm)
20 {
21   FILE *stream;
22   char line[MAX_LEN], *result, *nproc, *sproc, *rates, *costcomm;
23   int cline = 0,i,j,speed,qtde,cont;
24
25 //  Lendo estrutura do grid
26
27     if ((stream = fopen(fgrid,"rt")) != NULL)
28     {
29       while ((result = fgets(line,MAX_LEN,stream)) != NULL )
30       {
31         nproc=strtok(line," ");
```

```
32          sproc=strtok(NULL," ");
33          insere(&ini_grid,nproc,atoi(sproc));
34          cline++;
35        }
36        if (fclose(stream))
37           printf("fclose error fgrid\n");
38
39        printf("\nEstrutura do grid original\n");
40        mostra(ini_grid);
41      }
42      else
43         printf("fopen error\n");
44
45  // Lendo estrutura do cluster
46
47      if ((stream = fopen(fcluster,"rt")) != NULL)
48      {
49        while ((result = fgets(line,MAX_LEN,stream)) != NULL )
50        {
51          nproc=strtok(line," ");
52          sproc=strtok(NULL," ");
53          insere(&ini_cluster,nproc,atoi(sproc));
54        }
55        if (fclose(stream))
56           printf("fclose error\n");
57
58        printf("\nEstrutura do cluster\n");
59        mostra(ini_cluster);
60      }
61      else
62         printf("fopen error fcluster\n");
63
64  // Filtrando grid em relacao aos recursos do cluster
65      speed=maior(ini_cluster);
66
67      filtra(ini_grid,speed);
68
69      qtde=conta_itens(ini_grid);
70
71  // Lendo matriz com a taxa de transferencia de dados
72
```

```
73      mB=(int *)malloc(qtde*qtde*sizeof(int));
74      if (!mB)
75      {
76          printf("malloc error mB\n");
77          exit(1);
78      }
79      if ((stream=fopen(frates,"rt")) != NULL)
80      {
81        cont=0;
82        i=0;
83        while ((result = fgets(line,MAX_LEN,stream)) != NULL)
84        {
85          if ((busca_cod(ini_grid,i)) == 1)
86          {
87            rates=strtok(line," ");
88            for (j=0;j<cline;j++)
89            {
90              if ((busca_cod(ini_grid,j)) == 1)
91              {
92                  //printf("Taxa transferencia de %i para %i:%s\n",i,j,rates);
93                  mB[cont]=atoi(rates);
94                  rate_transf+=mB[cont];
95                  cont++;
96                  rates=strtok(NULL," ");
97              }
98              else
99                  rates=strtok(NULL," ");
100           }
101         }
102         i++;
103       }
104       rate_transf=rate_transf/cont;
105       if (fclose(stream))
106         printf("fclose error\n");
107     }
108     else
109        printf("fopen error frates\n");
110
111     qtde=conta_itens(ini_grid);
112
113 // Lendo vetor que contem o custo inicial de comunicacao de cada
```

```
        processador
114     vL=(double *)malloc(qtde*sizeof(double));
115     char * pEnd;
116     if (!vL)
117     {
118         printf("malloc error vL\n");
119         exit(1);
120     }
121
122     if ((stream=fopen(fcostcomm,"rt")) != NULL)
123     {
124       cont=0;
125       i=0;
126       while ((result = fgets(line,MAX_LEN,stream)) != NULL)
127       {
128         costcomm=strtok(line," ");
129         for(i=0;i<cline;i++)
130         {
131             //printf("Custo comunicacao do proc %i: %s\n",i,costcomm);
132             if ((busca_cod(ini_grid,i)) == 1)
133             {
134                 vL[cont]=strtod(costcomm,&pEnd);
135                 //printf("Custo comunicacao do proc %i: %lf\n",i,vL[cont]);
136                 cont++;
137                 costcomm=strtok(NULL," ");
138             }
139             else
140             {
141               costcomm=strtok(NULL," ");
142             }
143         }
144       } // fim-while
145     }
146     else
147        printf("fopen error fcostcomm\n");
148
149
150     // Renumera o ID dos processadores devido a exclusao realizada durante a
            filtragem com o cluster
151     renumera(ini_grid);
152
```

```
153     printf("\nEstrutura do grid apos filtro\n");
154
155     mostra(ini_grid);
156
157     // Vendo matriz que contem a taxa de transferencia entre os
            processadores
158     printf("\nMatriz B - taxa de transferencia de dados do processador i (
            linha) para o j (coluna)\n");
159     for (i=0;i<conta_itens(ini_grid);i++)
160     {
161         for (j=0;j<conta_itens(ini_grid);j++)
162               printf("%4i ",mB[i*qtde+j]);
163          printf("\n");
164     }
165
166
167     // Vendo vetor com os custos de comunicacao de cada processador
168     printf("\nVetor com os custos de comunicacao dos processadores\n");
169     for (i=0;i<conta_itens(ini_grid);i++)
170         printf("%lf ",vL[i]);
171     printf("\n");
172 } // fim-programa
```

```
1  /*
2   * Universidade Federal de Sao Carlos
3   * Departamento de Computacao
4   * Progrid - www.gsdr.dc.ufscar.br/progrid
5   * Programacao Paralela e Distribuida
6   * Copyleft (L) 2006, Daniele S. Jacinto, Ricardo A. Rios e Helio C.
        Guardia
7   */
8
9  /*
10  * Programa : resources.h
11  * Descricao: implementacao de estruturas de dados e funcoes para manipular
        recursos do grid e do cluster
12  * Ultima alteracao: 02/11/06 - Daniele
13  * Ultima alteracao: 08/03/07 - rios
14  */
15
16
17 #include "resources.h"
18
19 void insere(processor **cabeca,char nome[SIZE_NOME],int mhz)
20 {
21  processor *aux=NULL;
22  int codigo;
23  processor *novo;
24  if (*cabeca == NULL)
25  {
26    *cabeca = (processor *)malloc(sizeof(processor));
27    (*cabeca)->cod=0;
28    strcpy((*cabeca)->nome,nome);
29    (*cabeca)->mhz=mhz;
30    (*cabeca)->proximo=NULL;
31  }
32  else
33  {
34   aux=*cabeca;
35   while (aux->proximo != NULL)
36         aux=aux->proximo;
37   codigo=(aux->cod+1);
38   novo=(processor *)malloc(sizeof(processor));
39   novo->cod=codigo;
```

```
40    strcpy(novo->nome,nome);
41    novo->mhz=mhz;
42    novo->proximo=NULL;
43    aux->proximo=novo;
44   }
45  }
46
47  /* Exibe os recursos disponiveis na lista encadeada do grid ou do cluster
        */
48  void
49  mostra(processor *noatual)
50  {
51   int i=0;
52   while (noatual != NULL)
53   {
54     i++;
55     printf("Codigo: %i, Nome: %s, MIPS: %i\n",noatual->cod,noatual->nome,
           noatual->mhz);
56     noatual=noatual->proximo;
57   }
58  }
59
60  /* Pesquisar o processador de maior desempenho na lista encadeada do grid
       ou do cluster */
61  int
62  maior(processor *noatual)
63  {
64   int maior;
65   if (noatual == NULL)
66      return -1;
67   else
68   {
69      maior=noatual->mhz;
70      while (noatual != NULL)
71      {
72        if (noatual->mhz > maior)
73             maior=noatual->mhz;
74        noatual=noatual->proximo;
75      }
76      return maior;
77   }
```

```
78 }
79
80 /* Verifica se o processador com id "codigo" ainda eh um recurso disponivel
       no grid (devido a filtragem em relacao aos recursos do cluster) */
81 int
82 busca_cod(processor *noatual,int codigo)
83 {
84  if (noatual == NULL)
85     return -1;
86  else
87  {
88     while ((noatual != NULL) && (noatual->cod != codigo))
89     {
90       noatual=noatual->proximo;
91     }
92     if (noatual == NULL)
93        return -1;
94     else
95        return 1;
96  }
97 }
98
99 /* Conta quantos recursos ha na lista encadeada do grid ou do cluster */
100 int
101 conta_itens(processor *noatual)
102 {
103  int i=0;
104  if (noatual == NULL)
105      return -1;
106  else
107  {
108    while (noatual != NULL)
109    {
110      i++;
111      noatual=noatual->proximo;
112    }
113    return i;
114  }
115 }
116
117 int
```

```
118 exclui(processor **cabeca,int codigo)
119 {
120  processor *anterior=NULL;
121  processor *aux=NULL;
122  aux=*cabeca;
123  anterior=*cabeca;
124  if (aux == NULL)
125     return -1;
126  else
127  {
128   if (aux->cod == codigo)
129   {
130    printf("Resolver exclusão da cabeca da lista encadeada");
131 /* A exclusao da cadbeca da lista está dando erro
132     anterior=aux;
133     cabeca=&(aux->proximo);
134     free(anterior);
135 */     return codigo;
136   }
137   else
138   {
139      while ((aux != NULL) && (aux->cod != codigo))
140      {
141         anterior=aux;
142         aux=aux->proximo;
143      }
144      if (aux != NULL)
145      {
146        anterior->proximo=aux->proximo;
147        free(aux);
148        return codigo;
149      }
150      else
151          return -2;
152   }
153  }
154 }
155
156
157 /* Renumera "codigo" dos recursos do grid apos filtragem */
158 void
```

```
159 renumera(processor *noatual)
160 {
161  int i=0;
162  while (noatual != NULL)
163  {
164    noatual->cod=i;
165    i++;
166    noatual=noatual->proximo;
167  }
168 }
169
170
171 /* Filtra os recursos do grid em relacao ao melhor processador disponivel
       no cluster */
172 int
173 filtra(processor *noatual,int mhz)
174 {
175  processor *temp;
176
177  if (noatual == NULL)
178     return -1;
179  else
180  {
181   temp=noatual;
182   while (temp != NULL)
183   {
184          if (temp->mhz < mhz)
185             exclui(&noatual,temp->cod);
186          temp=temp->proximo;
187   }
188  }
189  free(temp);
190  return -2;
191 }
192
193 int
194 find_mhz(processor *noatual,int codigo)
195 {
196   int result = -1;
197
198   while(noatual != NULL && noatual->cod != codigo){
```

```
199          noatual = noatual->proximo;
200    }
201    if(noatual != NULL)
202    {
203  //      printf("No: %i   Task:%i Custo: %i\n",new->id,task,new->cost);
204       result = noatual->mhz;
205    }
206    noatual=NULL;
207    free(noatual);
208    return result;
209  }
210
211
212  char
213  *find_hostname(processor *noatual,int codigo)
214  {
215
216    while(noatual != NULL && noatual->cod != codigo){
217          noatual = noatual->proximo;
218    }
219    if(noatual != NULL)
220    {
221     strcpy(name,noatual->nome);
222    }
223    noatual=NULL;
224    free(noatual);
225    return name;
226  }
```

# 15. Anexo F - Implementação do HEFT

```
1  /*
2   * Universidade Federal de Sao Carlos
3   * Departamento de Computacao
4   * Progrid - www.gsdr.dc.ufscar.br/progrid
5   * Programacao Paralela e Distribuida
6   * Copyleft (L) 2006, Daniele S. Jacinto, Ricardo A. Rios e Helio C.
        Guardia
7   */
8
9  /*
10  * Programa : algo_eft.h
11  * Descricao: funcoes para implementar escalonamento segundo modelo EFT
12  * Ultima alteracao: 02/11/06 - Daniele
13  * Ultima alteracao: 08/03/07 - rios
14  */
15
16 #include "algo-eft.h"
17
18
19 /* Estima o tempo de execucao de cada tarefa i no processador j */
20 void matriz_W()
21 {
22     int tNos=t_nodes(), tCpus=conta_itens(ini_grid),i,j,cont=0,best_resource
          ;
23     double ciclos;
24     mW=(double *)malloc(tNos*tCpus*sizeof(double));
25     if (!mW)
26     {
27         printf("malloc error matriz_w()\n");
28         exit(1);
29     }
30     best_resource=maior(ini_cluster);
```

```
31      for (i=0;i<tNos;i++)
32      {
33           for (j=0;j<tCpus;j++)
34           {
35                if(cost_node(i) > 0)
36                {
37                  //printf("Cost node %i:%lf  Best_resoure: %i  \n",i,cost_node
                        (i),best_resource);
38                  mW[cont]=(((double)cost_node(i)*best_resource)/((double)
                        find_mhz(ini_grid,j)));
39                }
40                else
41                  mW[cont]=0;
42                cont++;
43           }
44          ciclos=(((double)cost_node(i)*pow(10,-6))*(best_resource*pow(10,6)))
               ;
45          printf("Cost node %i:%lf  Best_resoure: %i  MIPS da tarefa: %lf\n",i
               ,cost_node(i),best_resource, ciclos);
46      }
47  }
48
49  void view_mW()
50  {
51   int tNos=t_nodes(), tCpus=conta_itens(ini_grid),i,j,cont=0;
52   printf("\nMatriz W - custo computacional da task i (linha) no processador
        j (coluna) em segundos\n");
53   for (i=0;i<tNos;i++)
54   {
55         for (j=0;j<tCpus;j++)
56           {
57                printf("%10.8lf    ",mW[cont]);
58                cont++;
59           }
60           printf("\n");
61   }
62  }
63
64  double
65  est(int task, int processor)
66  {
```

```
67    double result = 0, maior = 0;
68
69    if (task == find_entry())
70    {
71       return 0;
72    }
73    else
74    {
75      arc *new = (arc*)malloc(sizeof(arc));
76      new = _ahead;
77
78      while(new != NULL)
79      {
80            if((new->nodein)->id == task)
81            {
82               // a variavel rate_transf eh calculada no codigo arch-model.h
                     quando a mB eh lida
83               result=(double)(vL[processor]+(double)((new->weight*8)/(
                     rate_transf)));
84            }
85            new = new->arcn;
86            if (result > maior)
87               maior = result;
88      }
89      new=NULL;
90      free(new);
91      return maior;
92    }
93 }
94
95 double
96 eft_completo(int task, int processor)
97 {
98     double calc;
99       // Politica de inserção é considerar 95% do tempo de execução de uma
             tarefa em um determinado processador
100      calc=(double)mW[(task*(conta_itens(ini_grid)))+processor]+max(((double
             )avail[processor].rank)*0.95,(double)est(task,processor));
101 //    printf("EFT da task %i no processador %i:%lf\n",task,processor,calc);
102    return calc;
103 }
```

```
104
105 double
106 max(double i, double j)
107 {
108  if (i > j)
109     return i;
110  else
111     return j;
112 }
113
114
115 void
116 heft()
117 {
118   int i,j, task, tNos=t_nodes(), tCpus=conta_itens(ini_grid),cont,cont2;
119   double result,total=0;
120   processor *noatual;
121   t_rankp *p_rank;
122   p_rank=(t_rankp *)malloc(tCpus*sizeof(t_rankp));
123   avail=(t_rankp *)malloc(tCpus*sizeof(t_rankp));
124   schedule=(t_schel *)malloc(tNos*sizeof(t_schel));
125
126   for(i=0;i<tCpus;i++)
127   {
128      avail[i].cod=i;
129      avail[i].rank=0;
130   }
131
132   cont2=0;
133
134   for(i=0;i<(tNos-removed_nodes);i++)
135   {
136     noatual=ini_grid;
137     task=v_rank[i].cod;
138     cont=0;
139     printf("\n");
140     while (noatual != NULL)
141     {
142       result=eft_completo(task,noatual->cod);
143       p_rank[cont].cod=noatual->cod;
144       p_rank[cont].rank=result;
```

```
145         printf("EFT da task %i no processador %i:%lf seg.\n",task,p_rank[cont
                ].cod,p_rank[cont].rank);
146         cont++;
147         noatual=noatual->proximo;
148       }
149       insertSortP(p_rank,conta_itens(ini_grid));
150       printf("EFT que minimiza a execucao da task %i no processador %i:%lf
             seg.\n",task,p_rank[0].cod,p_rank[0].rank);
151       schedule[cont2].task=task;
152       schedule[cont2].proc=p_rank[0].cod;
153       total+=p_rank[0].rank;
154       cont2++;
155       for(j=0;j<tCpus;j++)
156       {
157         if(avail[j].cod == p_rank[0].cod)
158            avail[j].rank+=p_rank[0].rank*0.95;
159       }
160     }
161
162     printf("\nTempo de ocupacao de cada processador\n");
163     for(j=0;j<tCpus;j++)
164        printf("Avail[%i]:%lf\n",avail[j].cod,avail[j].rank);
165     noatual=NULL;
166     free(noatual);
167   }
168
169   void
170   insertSortP(t_rankp a[], int length)
171   {
172         int i, j, cod;
173         double value;
174
175         for(i = 1; i < length; ++i)
176         {
177             value = a[i].rank;
178             cod = a[i].cod;
179             for (j = i - 1; j >= 0 && a[j].rank > value; --j) {
180                 a[j + 1].cod = a[j].cod;
181                 a[j + 1].rank = a[j].rank;
182                 a[j].cod = cod;
183                 a[j].rank = value;
```

```
184             }
185         }
186     }
187
188
189 void
190 insertSortS(t_schel a[], int length)
191 {
192         int i, j, proc,task;
193
194         for(i = 1; i < length; ++i)
195         {
196             proc = a[i].proc;
197             task = a[i].task;
198             for (j = i - 1; j >= 0 && a[j].task > task; --j) {
199                 a[j + 1].task = a[j].task;
200                 a[j + 1].proc = a[j].proc;
201                 a[j].task = task;
202                 a[j].proc = proc;
203             }
204         }
205     }
206
207 void
208 print_rsl(char *dir, char *file)
209 {
210   int i, tNos=(t_nodes()-removed_nodes);
211   char *concat=(char *)malloc(sizeof(char));
212 //   char *itoa=(char *)malloc(sizeof(char));
213   insertSortS(schedule,tNos);
214   printf("\nArquivo RSL\n");
215   printf("+");
216   for(i=0;i<tNos;i++)
217   {
218     strcpy(concat," ");
219     strcat(concat,"echo \"(&(resourceManagerContact=\\\"");
220     strcat(concat, find_hostname(ini_grid,schedule[i].proc));
221     strcat(concat, "\\\"))\"");
222     system(concat);
223     strcpy(concat," ");
224     strcat(concat,"echo \"(count=1)\"");
```

```
225       system(concat);
226       strcpy(concat," ");
227       strcat(concat,"echo \"(label=\\\"subjob ");
228   //    sprintf(itoa,"%i",schedule[i].task);
229   //      strcat(concat,"itoa");
230       strcat(concat,"\\\")\"");
231       system(concat);
232   //      fprintf(stderr,concat);
233   //      system(concat);
234       strcpy(concat," ");
235       strcat(concat,"echo \"(environment=(GLOBUS_DUROC_SUBJOB_INDEX ");
236       strcat(concat,"itoa");
237       strcat(concat,"))\"");
238       system(concat);
239   //      fprintf(stderr,concat);
240       strcpy(concat," ");
241       strcat(concat,"echo \"(LD_LIBRARY_PATH /shared/tools/globus4/lib/)\"");
242       system(concat);
243   //      fprintf(stderr,concat);
244       strcpy(concat," ");
245       strcat(concat,"echo \"(directory=\\\"");
246       strcat(concat,dir);
247       strcat(concat,"\\\")\"");
248       system(concat);
249   //      fprintf(stderr,concat);
250       strcpy(concat," ");
251       strcat(concat,"echo \"(executable=\\\"");
252       strcat(concat,dir);
253       strcat(concat,"/");
254       strcat(concat,file);
255       strcat(concat,"\\\")\"");
256       system(concat);
257   //      fprintf(stderr,concat);
258       strcpy(concat," ");
259       strcat(concat,"echo \")\"");
260       system(concat);
261   //      fprintf(stderr,concat);
262    }
263   }
264
265
```

```
266 void
267 print_rsl_old(char *dir, char *file)
268 {
269   int i, tNos=(t_nodes()-removed_nodes);
270   insertSortS(schedule,tNos);
271   printf("\nArquivo RSL\n");
272   printf("+");
273   for(i=0;i<tNos;i++)
274   {
275     printf("(&(resourceManagerContact=\"%s\")",find_hostname(ini_grid,
          schedule[i].proc));
276     printf("(count=1)");
277     printf("(label=\"subjob %i\")",schedule[i].task);
278     printf("(environment=(GLOBUS_DUROC_SUBJOB_INDEX %i)",schedule[i].task);
279     printf("(LD_LIBRARY_PATH /shared/tools/globus4/lib/)");
280     printf("(directory=\"%s/\")",dir);
281     printf("(executable=\"%s/%s\")",dir,file);
282     printf(")");
283  }
284 }
285
286 void
287 print_machines()
288 {
289   int i, tNos=(t_nodes()-removed_nodes);
290   insertSortS(schedule,tNos);
291   printf("\nArquivo MACHINES\n");
292   for(i=0;i<tNos;i++)
293   {
294     printf("\"%s\" 1\n",find_hostname(ini_grid,schedule[i].proc));
295  }
296 }
```