Universidade Estadual de Campinas
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Um Mecanismo para Distribuição de Carga em Ambientes Virtuais de Computação Maciçamente Paralela

Autor: Fabiano de Oliveira Lucchese

Orientador: Prof. Dr. Marco Aurélio Amaral Henriques

Comissão Julgadora

Prof. Dr. César Augusto Fonticielha de Rose - PUC-RS
Prof. Dr. Clésio Luis Tozzi - FEEC/UNICAMP
Prof. Dr. Fernando José von Zuben - FEEC/UNICAMP
Prof. Dr. Marco Aurélio A. Henriques - FEEC/UNICAMP

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos necessários para a obtenção do título de Mestre em Engenharia Elétrica.

Campinas
Dezembro, 2002

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

L963m Lucchese, Fabiano de Oliveira
Um mecanismo para distribuição de carga em ambientes virtuais de computação maciçamente paralela / Fabiano de Oliveira Lucchese.–Campinas, SP: [s.n.], 2003.

Orientador: Marco Aurélio Amaral Henriques.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Processamento eletrônico de dados - Processamento distribuido. 2. Redes de computação - Carga de trabalho. 3. Sistemas operacionais. 4. Internet (Redes de computação). I. Henriques, Marco Aurélio Amaral. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Agradecimentos

À minha família, que jamais mediu palavras e gestos de apoio ao meu trabalho.

Ao meu orientador, Prof. Marco Aurélio Amaral Henriques, cuja convivência ao longo destes anos esteve muito além de ser apenas intelectualmente enriquecedora.

Aos meus amigos, que envolvidos ou não neste trabalho, contribuíram fundalmentalmente para o estabelecimento de meu ponto de equilíbrio.

À CAPES, pelo auxílio financeiro concedido durante a realização deste trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Resumo

O recente desenvolvimento de infra-estruturas de telecomunicações como as redes internacionais de dados, capazes de interconectar milhões de computadores espalhados pelo mundo inteiro, tornou possível a utilização de extensos recursos computacionais a custos relativamente baixos. Desta nova realidade surgiram as pesquisas e projetos relacionados aos *computadores virtuais maciçamente paralelos*, que consistem em ambientes virtuais formados por um grande número de computadores que procuram trabalhar cooperativamente na busca de soluções para problemas até então impossíveis de serem tratados pelos sistemas computacionais disponíveis. Entretanto, como em todo novo domínio de pesquisa, neste também há muitas questões ainda sem solução, especialmente para o problema de gerenciamento da carga de processamento no sistema. Neste trabalho, será abordado o problema de balanceamento de carga em ambientes virtuais de computação maciçamente paralela introduzindo-se o algoritmo de Escalonamento Geracional com Replicação de Tarefas (GSTR), o algoritmo de Particionamento de aplicações em Grupos Proporcionais e Paralelos (PPCP) e a Linguagem de Especificação de Aplicações Paralelas (PASL). Um abrangente conjunto de testes é realizado a fim de validar as soluções propostas. A plataforma JOIN é utilizada como bancada de testes para a verificação do desempenho da proposta em ambientes computacionais reais.

Abstract

The recent development of telecomunication infra-structures such as the world-wide data networks, interconnecting millions of computers spread all over the world, has made possible the use of large computational resources at a rather low cost. Within this new reality there has emerged research activities and projects related to *massively parallel virtual computers*, which are virtual environments made up by a large number of computers that work cooperatively in order to solve problems impossible to be tackled by nowadays computer systems. However, as in every new domain of research, in this one there are many unsolved questions, in particular those related to the management of the processing load into the system. In this work, the problem of balancing processing loads on massively parallel virtual computers is approached by the introduction of the Generational Scheduler with Task Replication (GSTR) algorithm, the Parallel and Proportional Cluster Partitioning (PPCP) algorithm and the Parallel Application Specification Language (PASL). A conprehensive set of tests is carried out in order to validate the proposed solutions. The JOIN platform is used as a testbed for the evaluation of the performance of the solution when applied to real computational environments.

Capítulo 1

Introdução

1.1 A Internet como um computador virtual maciçamente paralelo

Nas últimas três décadas, o mundo tem observado, com certo grau de perplexidade, uma silenciosa revolução nos meios de produção e comunicação da economia e da sociedade. A chamada 'era digital', assim definida por embasar-se no emprego generalizado de dispositivos digitais para a resolução de tarefas dos mais variados níveis de complexidade, tem proporcionado às indústrias, às empresas provedoras de serviços e até mesmo ao cidadão comum, a capacidade de suplantar barreiras de maneira nunca antes imaginada.

Dentre os inúmeros campos tecnológicos beneficiados com o advento da era digital, aquele que parece estar sofrendo a maior revolução é, sem sombra de dúvida, o das telecomunicações. Impulsionado pelo desenvolvimento das redes de telecomunicação, sobretudo das redes digitais de médio e longo alcance, este campo desfruta hoje de uma complexa infra-estrutura capaz de interligar, em escala mundial, os mais variados tipos de dispositivos eletrônicos (computadores, telefones celulares, telefones fixos e eletrodomésticos, entre outros).

A integração, via redes digitais, de computadores nos mais variados setores da sociedade e a integração destes setores em uma rede mundial, como a Internet, premitiu que um grande número de dispositivos dotados de poder computacional e de armazenamento de dados pudesse interagir de maneira a resolver problemas complexos até então impossíveis de serem resolvidos por um, ou poucos, destes dispositivos isoladamente. Com redes de interconexão cada vez mais rápidas e que conectam um número crescente de computadores, as possibilidades tornaram-se muito grandes.

Entretanto, são também inúmeras as dificuldades em se tirar proveito deste tipo de recurso; provavelmente as maiores dificuldades residam na impossibilidade de se avaliar com precisão as características de uma rede de computadores como a Internet, em que os conjuntos de computadores e enlaces de comunicação são extremamente heterogêneos e variam segundo uma dinâmica difícil de modelar.

Diante deste grande desafio, vários grupos de pesquisa espalhados pelo mundo iniciaram estudos que procuram apresentar propostas que tratem os inúmeros probelmas decorrentes dessa dificuldade. Algumas

das propostas mais conhecidas são as plataformas SETI@Home [SET02], United Devices' Frontier [Fro02], Parabon [Par02] e Entropia [Ent02], que serão discutidas no Cap. 2.

Neste trabalho será abordada a questão da distribuição de carga computacional em um sistema distribuído tão heterogêneo e passível de falhas quanto se pode esperar de um sistema baseado na Internet. Para isso, serão apresentadas na seção a seguir algumas definições preliminares sobre as quais se baseará todo o desenvolvimento teórico deste trabalho.

1.2 Definições preliminares

Apesar das propostas citadas na seção anterior possuírem muitas características em comum, poucas foram as iniciativas em se formalizar o modelo de computação promovido por este tipo de plataforma [WTH95]. Algumas das iniciativas de formalização de modelos de computação paralela mais bem estabelecidas, tais como os descritos em [Sun90] e [BWL00], referem-se a sistemas com características diferentes daquelas presentes nos computadores virtuais maciçamente paralelos (MPVCs), sobretudo no que tange à escalabilidade. Por esta razão, nesta seção será apresentada uma proposta de modelo de computação adequado às características das plataformas que procuram desenvolver o tipo de processamento descrito anteriormente.

Os conceitos-chave a serem utilizados neste trabalho são os de **aplicação** e de **MPVC**. Definida informalmente, uma aplicação pode ser entendida como um trabalho qualquer a ser desenvolvido sobre um conjunto de dados com o objetivo de se produzir algum resultado esperado. Já um MPVC, definido também de modo informal, é a estrutura computacional necessária para se realizar o trabalho representado por uma aplicação de grande porte que possa ser dividida em um grande número de tarefas.

As plataformas citadas na seção anterior são exemplos de MPVC, uma vez que implementam uma estrutura capaz de executar diversos tipos de trabalho utilizando-se um conjunto arbitrariamente grande de computadores. Exemplos de aplicações para estes MPVCs são a montagem de fragmentos de DNA [LH00], o cálculo dos fatores primos de números inteiros grandes [Sil91] [Pri94], análises moleculares [Str93] além de outros.

Apesar de aparentemente simples, estes conceitos, da maneira informal como foram definidos, possuem ainda muitos pontos ambíguos e, por isso, devem ser definidos segundo critérios mais rigorosos. Nas subseções a seguir, serão apresentadas formalizações para estes conceitos, as quais serão utilizadas ao longo deste trabalho.

1.2.1 O modelo de aplicação

Definição 1 *Define-se um* **bloco de dados** *como sendo uma estrutura capaz de armazenar conjuntos arbitrariamente grandes de dados serializáveis, ou seja, que possam ser transmitidos por meio de um canal serial de comunicação.*

Definição 2 *Define-se uma* **tarefa** τ *como sendo uma seqüência ordenada de operações aritméticas, matemáticas, de desvio condicional ou de atribuição/entrada/saída de dados, que implementa algum tipo de processamento determinístico e finito sobre um bloco de dados de entrada a fim de produzir um bloco de dados de saída.*

Definição 3 *Define-se um* **lote de tarefas** β^c *de cardinalidade c, onde* $c \in \aleph^*$*, como sendo a aplicação de uma determinada tarefa sobre c blocos de dados de entrada, potencialmente distintos, com o objetivo de se produzir c blocos de dados de saída.*

Definição 4 *Se* $\beta_x^{c_x}$ *e* $\beta_y^{c_y}$ *são dois lotes de tarefas distintos e se um sub-conjunto não vazio dos blocos de dados de entrada de* $\beta_y^{c_y}$ *é formado por um sub-conjunto não vazio dos blocos de dados de saída de* $\beta_x^{c_x}$*, então diz-se que há uma* **relação de dependência de dados** *entre* $\beta_x^{c_x}$ *e* $\beta_y^{c_y}$ *e essa relação é denotada por* δ_{xy} *Nessas circunstâncias,* c_y *deve ser um múltiplo ou um sub-múltiplo de* c_x *e essa relação numérica existente entre as cardinalidades dos lotes é chamada de* **relação multiplicativa** *entre* $\beta_x^{c_x}$ *e* $\beta_y^{c_y}$*.*

Definição 5 *Define-se uma aplicação paralela como sendo formada por:*

- *um conjunto* $B = \{\beta_1^{c_1}, \beta_2^{c_2}, \ldots, \beta_N^{c_N}\}$ *de N lotes de tarefas,*
- *um conjunto* $D = \{\delta_{x_1 y_1}, \delta_{x_2 y_2}, \ldots, \delta_{x_W y_W}\}$ *de W relações de dependência de dados entre lotes de tarefas de* B*, em que as seguintes restrições sejam obedecidas:*

  1. $\forall \delta_{xy} \in D, y > x.$
  2. $\forall \beta_x^{c_x} \in B, x < N, \exists \delta_{ij} \in D \mid i = x.$
  3. $\forall \beta_x^{c_x} \in B, x > 1, \exists \delta_{ij} \in D \mid j = x.$

Para se entender melhor o modelo de aplicação aqui adotado, considere a representação esquemática da aplicação paralela mostrada na Fig. 1.1.

Nesta aplicação hipotética, podem ser identificados cinco lotes de tarefas, cada qual representado por um círculo rotulado por $\beta_x^{c_x}$, onde x é um número inteiro de 1 a 5. A cada um destes lotes está relacionada uma determinada tarefa, não evidenciada na figura. Os enlaces direcionais que conectam os círculos representam as cinco relações de dependência existentes entre os lotes.

As restrições sobre as relações de dependência (Def. 5), que podem ser verificadas na aplicação representada pela Fig. 1.1, são necessárias para que não haja a ocorrência de inconsistências ou incoerências nos fluxos de dados gerados durante a execução da aplicação: a violação da primeira restrição pode criar ciclos no grafo de relações de dependência; as duas últimas restrições garantem que nenhum lote, a não ser o primeiro, deixará de receber dados de outro lote, e nenhum lote, a não ser o último, deixará de despachar seus resultados para outro lote. Além disso, a restrição imposta sobre as cardinalidades dos lotes de tarefas que possuem uma relação de dependência se faz necessária já que sua violação criaria indeterminações na maneira como o conjunto de blocos de dados gerados por um lote de tarefas é repartido ao ser remetido a outro(s) lote(s).

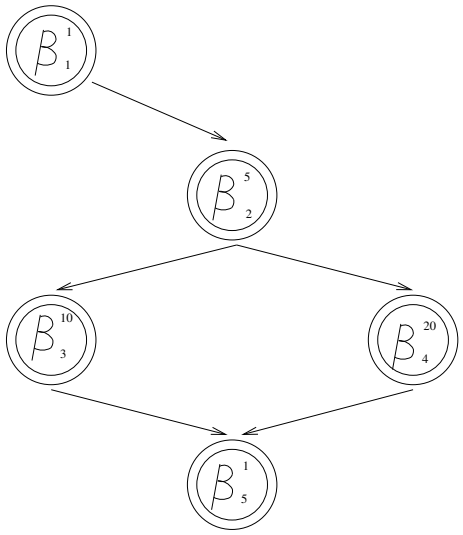

Figura 1.1: Representação gráfica de uma aplicação paralela baseada no modelo proposto.

1.2.2 O modelo de MPVC

Definição 6 *Define-se uma unidade de processamento ϕ_i como sendo uma entidade capaz de executar qualquer tarefa sobre qualquer bloco de dados a ela submetido, e capaz de se comunicar com outras unidades de processamento. Suas características são definidas pelas relações Vp_i e Vc_i.*

- *Dado um conjunto de tarefas $T_i = \{\tau_1, \tau_2, \ldots, \tau_P\}$, define-se $Vp_i : T_i \rightarrow \Re$ como a relação de* **velocidades de processamento** *não-concorrentes no processador ϕ_i das tarefas definidas em T_i.*
- *Dado um conjunto $P_i = \{\phi_1, \phi_2, \ldots, \phi_K\}$ de K unidades de processamento distintas de ϕ_i ($\phi_i \notin P_i$), define-se $Vc_i : P_i \rightarrow \Re$ como a relação de* **velocidades médias de comunicação** *de ϕ_i com as unidades de processamento definidas em P_i.*

Definição 7 *Define-se* **sistema computacional** *como sendo um conjunto $H = \{\phi_1, \phi_2, \ldots, \phi_M\}$ de M unidades de processamento.*

Definição 8 *Define-se um* **grupo** *G como sendo um sistema computacional onde existe pelo menos uma unidade de processamento* **coordenadora***, cuja principal característica é a capacidade de comunicar-se não só com as demais unidades de processamento do próprio sistema como também com unidades de processamento de outros sistemas. Qualquer outra unidade de processamento existente neste sistema é denominada unidade* **trabalhadora** *e só pode se comunicar com a unidade coordenadora do sistema em questão.*

Definição 9 *Define-se um* **computador paralelo** *como sendo:*

- *Um conjunto $M = \{G_1, G_2, \ldots, G_Q\}$ de Q grupos distintos e mutuamente exclusivos, ou seja, que não possuam unidades de processamento em comum.*
- *Um conjunto $C \subset M \times M$, de conexões não-orientadas entre os grupos de M.*

Definição 10 *Define-se um **computador maciçamente paralelo** (MPC) como sendo um computador paralelo dotado de uma infra-estrutura tal que a adição de uma unidade de processamento qualquer em um de seus grupos implique em um aumento no desempenho global do computador.*

Sabe-se que, da maneira como foi definido, um computador maciçamente paralelo é impossível de ser construído na prática, uma vez que a escalabilidade infinita implicada por esta definição reflete uma condição ideal. Entretanto, neste trabalho será utilizado o termo "maciçamente paralelo"para se referenciar sistemas cuja escalabilidade garanta a validade desta definição até para um número de unidades de processamento da mesma ordem de grandeza do número de computadores conectados à Internet.

Os computadores paralelos e maciçamente paralelos podem ser *reais* ou *virtuais*. Um computador paralelo real caracteriza-se por possuir unidades de processamento e enlaces de comunicação originalmente projetados para fazer parte deste computador paralelo. Já os computadores paralelos virtuais são formados por unidades de processamento e enlaces de comunicação que não foram originalmente concebidas para serem utilizadas num computador paralelo. Um computador paralelo formado por computadores conectados pela Internet é um exemplo de computador paralelo virtual.

A partir deste ponto,a sigla MPVC será utilizada para se referenciar os computadores maciçamente paralelos virtuais.

1.3 A plataforma JOIN

Nascida como fruto de pesquisas iniciadas em 1996 [Yer98] [Hen99] e que, desde então, vem sendo constantemente desenvolvida e aperfeiçoada, a plataforma de software JOIN se propõe a ser um computador virtual maciçamente paralelo baseado na Internet.

Desenvolvida inteiramente em linguagem Java e dotada de uma estrutura baseada em componentes, a plataforma JOIN, ou simplesmente JOIN, tem como principais características sua grande portabilidade, já que pode ser executado em qualquer computador que disponha de uma implementação da Máquina Virtual Java [Mic02], e sua flexibilidade de operação e configuração.

JOIN é um sistema composto por quatro tipos distintos de **componentes**: *servidor, coordenador, trabalhador e de administração (jack)*. Cada computador participante na plataforma JOIN executa um destes componentes que, dependendo de sua função, poderá estar presente em um ou muitos computadores simultaneamente. A Fig. 1.2 apresenta um diagrama que ilustra esta organização.

Nesta figura, cada um dos círculos pequenos representa um computador participante na plataforma, enquanto que a cor do círculo define o tipo de componente que está sendo executado no respectivo computador. Brevemente descritas, as atribuições de cada um destes componentes são:

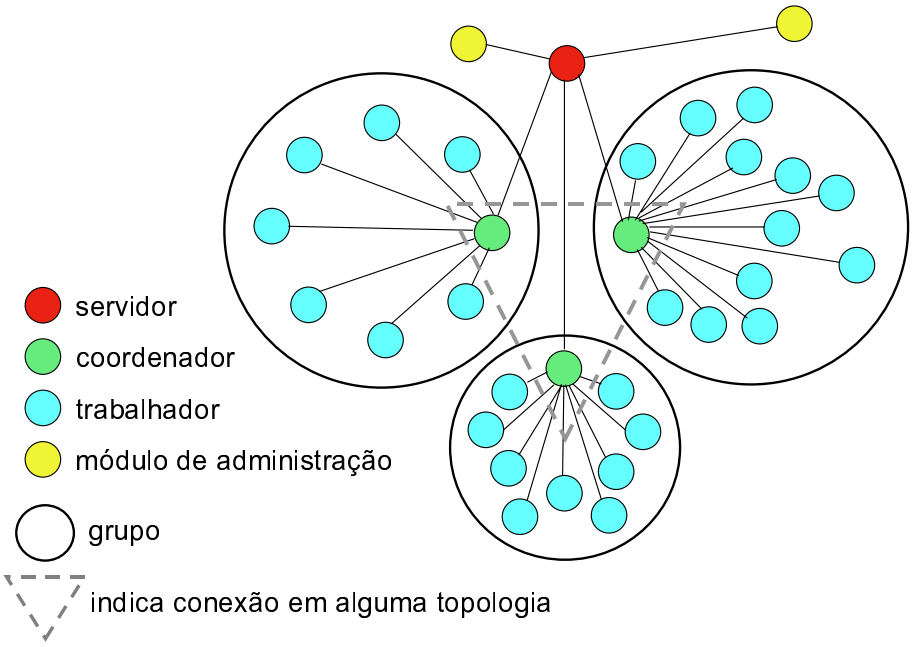

Figura 1.2: A plataforma JoiN.

1. **Servidor**: tem como objetivo gerenciar as operações de mais alto nível da plataforma. Alguns exemplos de operações deste tipo são: iniciar a execução de uma aplicação, receber novos computadores que desejem integrar-se à plataforma ou fornecer informações globais a respeito do funcionamento da mesma.

2. **Coordenador**: é o responsável pelo gerenciamento das atividades de computação realizadas em cada um dos **grupos** da plataforma. Além disso, é responsável também pela comunicação dos grupos com entidades externas a eles. Cada grupo admite somente um coordenador.

3. **Trabalhador**: é o responsável pela realização do trabalho computacional útil, ou seja, pela execução das tarefas de aplicações. Conjuntos de trabalhadores formam com o coordenador os grupos da plataforma.

4. **Módulo de Administração**: este componente, também chamado de JACK (JoiN Administration and Configuration Kit) tem a função de auxiliar a configuração e a operação da plataforma por parte de seus administradores. Para isso, mantém-se conectado ao componente servidor, a quem pode

submeter ordens específicas e/ou buscar informações para o monitoramento da plataforma.

Como pode ser deduzido, o conceito de **grupos** em JOIN é facilmente mapeado no conceito de grupo definido na seção anterior: o computador coordenador e os trabalhadores desempenham exatamente os papéis do coordenador e dos trabalhadores introduzidos na definição 8.

Os componentes servidor e de administração podem ser vistos como pertencentes a grupos especiais onde não há trabalhadores, o que garante total compatibilidade da estrutura desta plataforma com aquela exigida pelo modelo de MPVC proposto. Cabe aqui ainda destacar que a maneira como os grupos se interconectam logicamente pode ser escolhida arbitrariamente, de acordo com os requisitos do ambiente em que a plataforma será executada.

Maiores detalhes a respeito do funcionamento interno de cada um destes componentes poderão ser vistos no Cap. 4 e no Apêndice A.

1.4 Objetivos do trabalho e organização do texto

Conforme visto na seção introdutória deste capítulo, o objetivo deste trabalho é a concepção de um mecanismo de balanceamento de carga computacional para computadores maciçamente paralelos virtuais. Para isto, nos capítulos seguintes será apresentado o embasamento sobre o qual esta proposta se apoiará, assim como um conjunto de simulações e testes que procurarão validar a proposta.

Esta dissertação encontra-se dividida da seguinte maneira:

- Cap. 2 - *Balanceamento de Carga em MPVCs*: apresenta um breve resumo das principais técnicas de balanceamento de carga utilizadas em sistemas distribuídos, bem como alguns exemplos de mecanismos de balanceamento de carga aplicados a sistemas reais.

- Cap. 3 - *Proposta de Balanceamento de Carga para MPVCs*: propõe um mecanismo para o balanceamento de carga em computadores virtuais maciçamente paralelos. Para isso, serão introduzidos o escalonador GSTR, o particionador PPCP e a linguagem de especificação de aplicações paralelas PASL.

- Cap. 4 - *Balanceamento de carga na plataforma* JOIN: apresenta em detalhes a estrutura interna dos componentes JOIN para que, em seguida, sejam descritos os detalhes de implementação na plataforma do esquema de balanceamento de carga proposto.

- Cap. 5 - *Simulações*: apresenta e analisa os resultados obtidos nas simulações realizadas com vários esquemas de balanceamento de carga, dentre os quais o proposto neste trabalho.

- Cap. 6 - *Testes*: apresenta e analisa os resultados obtidos nos testes realizados com vários esquemas de balanceamento de carga, dentre os quais o proposto neste trabalho.

- Cap. 7 - *Conclusões e Trabalhos Futuros*: apresenta as conclusões e perspectivas deste trabalho.

Capítulo 2

Balanceamento de Carga em MPVCs

Neste capítulo, serão apresentadas algumas das principais técnicas de balanceamento de carga empregadas em sistemas distribuídos, procurando-se ressaltar os pontos que sejam relevantes à aplicação destas técnicas em MPVCs. Em seguida, serão vistos alguns exemplos de mecanismos de balanceamento de carga implementados em sistemas computacionais reais. Espera-se, com isso, que possam ser traçadas as linhas gerais nas quais um balanceador de carga para MPVCs deva ser concebido.

2.1 Técnicas básicas de balanceamento de carga

2.1.1 Escalonadores estáticos e dinâmicos

Escalonadores de tarefas são componentes de software comumente integrados a sistemas operacionais paralelos e/ou distribuídos e que têm como função a distribuição de trabalho computacional às unidades de processamento integrantes do sistema, de modo a maximizar o desempenho global do processamento realizado, isto é, promover o balanceamento de carga entre as unidades de processamento envolvidas [Bac89] [BL02] [CWF+98].

Em sistemas homogêneos, o problema de balanceamento de carga pode ser reduzido à divisão de um determinado trabalho computacional em N porções iguais e que possam ser distribuídas e executadas por N unidades de processamento - supostamente idênticas - do sistema. Neste caso, o problema está fortemente relacionado à maneira como representar o trabalho computacional a ser processado e à melhor maneira de dividi-lo em várias partes iguais [KA99].

Em sistemas heterogêneos, o problema de balanceamento de carga é consideravelmente mais complexo e, nestas circunstâncias, o escalonador de tarefas ganha especial importância. Para que o conjunto heterogêneo de unidades de processamento possa ser utilizado de maneira eficiente, questões como predição e monitoramento de desempenho passam a integrar o problema de balanceamento de carga [CWF+98]. Como o domínio de interesse deste trabalho é o de sistemas heterogêneos, tais como os MPVCs, este será o foco das considerações feitas deste ponto em diante.

O problema de atribuição ótima de um conjunto de tarefas a um conjunto heterogêneo de unidades de processamento é NP-difícil [Bac89]. Isso significa que um bom compromisso entre o tempo de processamento despendido na busca por uma solução e a qualidade da solução encontrada deve ser satisfeito e é no contexto deste compromisso que as principais linhas de desenvolvimento de escalonadores ganham forma: a dos escalonadores estáticos e a dos dinâmicos.

Um importante aspecto dos escalonamentos estáticos é que seu cálculo se faz de maneira totalmente independente da distribuição das tarefas [CWF+98]. Dito de outra forma, o escalonamento é feito em duas etapas: na primeira etapa o cálculo do escalonamento é realizado, ou seja, a atribuição das tarefas às unidades de processamento é definida; no segundo momento, um mecanismo de distribuição de tarefas deve entrar em ação para promover a distribução previamente calculada.

Uma importante consequência deste modelo de escalonamento é a necessidade de informações precisas sobre o sistema considerado. Assim, o bom funcionamento de um escalonamento de tarefas estático requer uma estimativa precisa do desempenho do sistema em questão e a qualidade deste escalonamento é um resultado direto da precisão com que estas estimativas são obtidas. Nestas circunstâncias, estimativas imperfeitas ou ocorrências de eventos inesperados que afetem o desempenho do sistema durante a execução das tarefas previamente escalonadas podem fazer com que seu desempenho global sofra significativos decréscimos.

Apesar desta aparente limitação, o escalonamento estático é largamente utilizado em sistemas paralelos reais [ADJ+91] [LWY95] [BL02], uma vez que sua simplicidade de implementação lhe confere grande robustez e facilidade de manutenção. Além disso, nestes sistemas a ocorrência de eventos que afetem significativamente o desempenho do escalonamento é rara e os resultados são frequentemente satisfatórios.

Em oposição a esta técnica está a dos escalonadores dinâmicos. O escalonamento dinâmico pode ser entendido como a aplicação de sucessivos escalonamentos estáticos sobre estados intermediários de execução da aplicação, à medida que ela é executada. Os momentos em que cada um desses escalonamentos é realizado varia de escalonador para escalonador, mas o aspecto mais importante dos escalonadores dinâmicos, e o que justifica o emprego do termo “dinâmico”, é o fato de o escalonamento ser feito concorrentemente à distribuição e execução das tarefas das aplicações.

Ao produzir-se um escalonamento com essas características, beneficia-se da habilidade em lidar com grande parte das decisões de escalonamento em tempo real, o que elimina muitos dos problemas do caso estático. Embora as decisões ainda se baseiem em estimativas de desempenho do sistema e, conseqüentemente, estimativas imprecisas ainda podem significar um escalonamento ineficiente, as conseqüências de um mal escalonamento não são tão impactantes quanto seriam no caso estático. Assim, atrasos ou adiantamentos no tempo de conclusão de uma determinada tarefa podem ser utilizados em tempo real para o re-escalonamento das tarefas ainda a serem executadas. Uma vantagem adicional do fato de seu processamento ser realizado concorrentemente à execução da aplicação escalonada é que isso pode significar economia de tempo global com relação ao caso estático [CWF+98].

Entretanto, os escalonadores dinâmicos possuem seus inconvenientes. Em contrapartida aos escalona-

dores estáticos, a implementação dos escalonadores dinâmicos é laboriosa e requer a manipulação e gerência de estruturas de dados frequentemente complexas. Esse fato torna este tipo de escalonador pesado, sob o ponto de vista da implementação e execução, e menos robusto já que, na eventual ocorrência de uma falha, um grande trabalho de recuperação de estado deverá ser feito [MSd+95].

Em uma plataforma que se propõe a utilizar de forma estruturada o poder computacional representado por muitos computadores conectados via Internet, o paradigma de escalonamento dinâmico parece ser mais adequado, apesar de seus inconvenientes. A dificuldade em se predizer com precisão o desempenho que cada um dos computadores participantes apresentará ao longo do período de processamento de uma determinada tarefa, aliada à possível ocorrência de eventos estocásticos que poderão afetar o desempenho global do sistema, parecem apontar na direção de um mecanismo capaz de se adaptar em tempo real às inúmeras variáveis envolvidas.

2.1.2 Paradigmas de balanceamento de carga

Outro importante aspecto no projeto de escalonadores de tarefas é o paradigma de operação adotado. A existência de diferentes paradigmas advém do fato da implementação do escalonador de tarefas estar diretamente vinculada à maneira como as unidades de processamento do sistema distribuído em questão estejam conectadas entre si - tanto física quanto logicamente. Assim, existe um grande número de paradigmas de balanceamento de carga, emergidos de diferentes topologias de interconexão, cada qual adaptado às características do ambiente computacional no qual foi concebido.

Nas sub-seções a seguir são apresentadas breves descrições dos mais comuns paradigmas de balanceamento de carga.

Cliente/Servidor ou Mestre/Escravo

Neste paradigma, o trabalho de balanceamento de carga é promovido por uma única unidade de processamento, que se responsabiliza por todas as atividades envolvidas, tais como: cálculo da divisão do trabalho computacional entre as unidades de processamento, distribuição do trabalho computacional, coleta de resultados, monitoramento do sistema etc.

Este paradigma, além de possuir a vantagem da simplicidade de implementação e operação, mobiliza uma única unidade de processamento para a execução do mecanismo de balanceamento. Este paradigma pode entretanto sofrer limitações de escalabilidade em face desta centralização.

Exemplos de trabalhos que exploram escalonadores deste tipo são [CWF+98] e [Sha01].

Hierárquico

Este paradigma, que pode ser entendido com uma extensão do paradigma mestre/escravo, consiste na utilização de múltiplos níveis hierárquicos para a distribuição de carga computacional. Assim, uma UP

central inicia a distribuição da carga para as UPs conectadas a ela como escravos, estas redistribuem suas cargas para as UPs também conectadas a elas como escravos e assim sucessivamente.

Nota-se que neste processo cada UP, com exceção da primeira, recebe trabalho computacional de apenas uma UP, ou seja, que cada UP escrava possui apenas um mestre. Esta estrutura, além de mais flexível que a estrutura mestre/escravo, é também mais escalável; nela, a carga computacional correspondente à execução do mecanismo de escalonamento de tarefas é distribuída entre todas as UPs que desempenham o papel de mestres sem que haja aumento na quantidade de comunicação. As desvantagens deste método com relação ao mestre/escravo estão no fato de o balanceamento ser feito em múltiplos passos (um para cada nível hierárquico), o que pode aumentar o tempo total de execução das aplicações, e também no fato de sua implementação e manutenção serem mais laboriosas.

Um exemplo de trabalho que explora este tipo de paradigma é [GGV96].

De difusão

Neste paradigma, que pode ser entendido como uma extensão do paradigma hierárquico, as unidades de processamento encontram-se conectadas de modo a formar redes de interconexão que podem conter ciclos. Exemplos de estruturas deste tipo são a anelar, a toroidal, a malha n-dimensional e o hipercubo.

Neste caso, o processo de balanceamento de carga se dá pela difusão da carga a ser balanceada pelas UPs participantes do sistema até que seja encontrado um estado de equilíbrio satisfatório. Esta estrutura é garantidamente escalável mas pode apresentar um tempo de convergência ao estado de equilíbrio excessivamente alto. Além disso, a existência de ciclos introduz a possibilidade de instabilidades no sistema.

Exemplos de trabalhos que exploram este paradigma são [HB99] e [HC97].

Descentralizado

Nestes sistemas, a rede de interconexões entre UPs é completa, ou seja, há uma conexão existente entre todo par de UPs e, por esta razão, pode ser entendido como o caso geral de todos os casos apresentados anteriormente.

A existência de conexões entre todas as UPs compromete seriamente a escalabilidade deste tipo de mecanismo e limita seu escopo de utilização a situações bem particulares. Seu principal domínio de aplicação é o de sistemas inteligentes e/ou adaptativos, em que cada UP desempenha o papel de um agente autônomo responsável pela resolução de parte do problema de balanceamento de carga.

Exemplos clássicos da aplicação deste paradigma podem ser encontrados em [DL94].

Na Fig. 2.1 podem ser vistas representações gráficas ilustrativas dos paradigmas de escalonamento citados nesta seção.

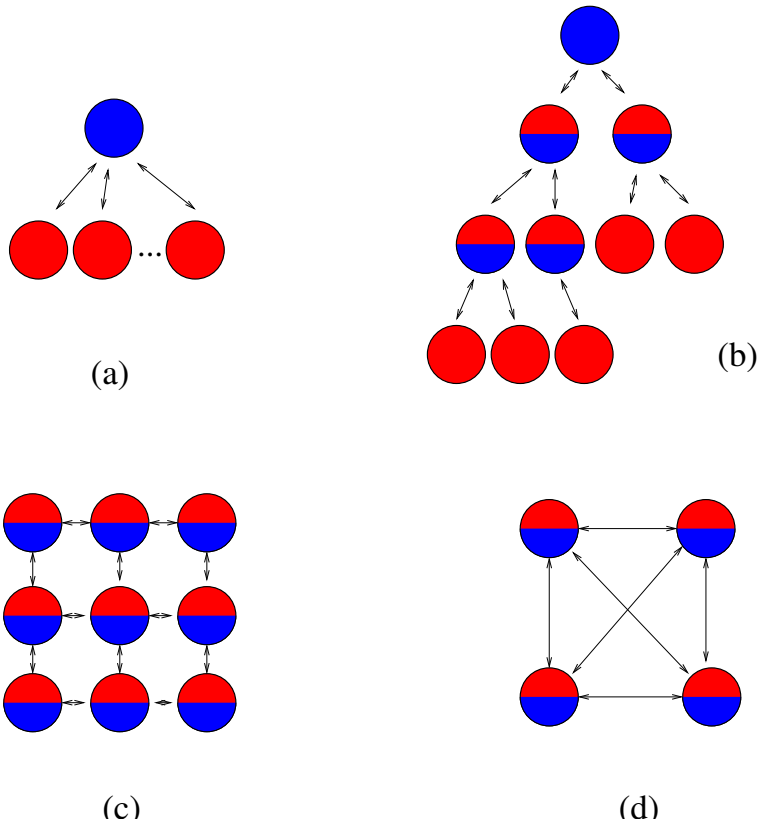

Figura 2.1: Paradigmas de balanceamento: (a) mestre/escravo, (b) hierárquico, (c) difusão e (d) descentralizado.

2.2 Balanceamento de Carga em MPVCs

Nesta seção serão vistos alguns exemplos de sistemas balanceadores de carga em MPVCs. Com isso, espera-se fazer um apanhado das principais técnicas atualmente em uso, suas vantagens e deficiências.

2.2.1 Legion

Em [GWT97] é apresentado o Legion, que consiste basicamente na proposta de um conjunto de interfaces que definem mecanismos para a constituição de um MPVC orientado a objetos.

Dada a generalidade desta abordagem, basicamente determinada pela possibilidade de múltiplas implementações da mesma funcionalidade, o Legion não disponibiliza escalonadores de tarefas genéricos para suas aplicações; ao contrário, incentiva que os programadores de aplicações criem seus próprios escalonadores, adaptados às necessidades particulares de suas aplicações e aos ambientes onde serão executadas.

Em [CkkG99] é apresentado um exemplo de escalonador de tarefas para esta plataforma. Como pode ser verificado neste trabalho, a estrutura básica em que se organizam as unidades de processamento no Legion é claramente hierárquica. Desta forma, a escalabilidade da plataforma será garantida pela existência

de múltiplos níveis hierárquicos, que também definirá como será organizado o escalonamento de tarefas, independentemente da política escolhida para este escalonamento.

As interfaces do mecanismo de escalonamento de tarefas prevêem a possibilidade de tratamentos de re-escalonamento mas não no caso em que haja falhas em uma UP. Como será visto no Cap. 3, este fator pode comprometer significativamente o nível de tolerância a falhas do sistema.

2.2.2 Globus

O projeto Globus [FK97], desenvolvido com a mesma filosofia do Legion, consiste numa especificação na qual implementações de componentes devam se basear para que possam compor um ambiente virtual de computação maciçamente paralela.

Ainda analogamente ao Legion, Globus não disponibiliza escalonadores genéricos para sua aplicações mas sim interfaces que procuram guiar implementações adaptadas a aplicações específicas. As principais implementações que seguem o padrão Globus são as plataformas Java CoG Kit [vLFGL01], MPICH-G2 [KTF03] e Nexus [FKT96].

Nestes trabalhos, pode-se verificar que a estrutura básica em que se organizam as unidades de processamento no Globus é hierárquica, o que define também o paradigma de balanceamento de carga a ser adotado pelos escalonadores de tarefas.

As principais diferenças existentes entre o Legion e o Globus residem no fato deste último não possuir um projeto puramente baseado em técnicas de orientação a objetos e também ao fato dele não prover suporte para uma autonomia das UPs como em Legion. Por essa razão, esta plataforma tende a ser ligeiramente mais eficiente porém menos flexível e robusta que o Legion.

Atualmente existem diversas implementações de MPVCs que seguem o padrão do *middleware* Globus, sendo a mais conhecida delas o sistema Nimrod/G. Maiores informações acerca deste projeto podem ser encontradas em [BAG00].

2.2.3 Condor

O Condor foi umas das primeiras plataformas a serem concebidas com o objetivo de se aproveitar os ciclos de CPU de máquinas ociosas. Inteiramente desenvolvida em C/C++ e inicialmente dedicada aos sistemas operacionais padrão Unix, foi utilizado em diversos laboratórios de universidades espalhadas pelo mundo inteiro para aplicações científicas.

O fato deste sistema não ter se estabelecido como uma opção de MPVC para a Internet se deve basicamente às deficiências de escalabilidade de seu projeto original. Por ser baseado no paradigma mestre-escravo, esta plataforma não suportava um grande número de trabalhadores já que a coordenação de todo o processamento era centralizada num único servidor.

Mais recentemente, o modelo inicial do Condor foi estendido de modo a implementar os protocolos especificados pelo projeto Globus, o que lhe conferiu maior apelo entre os interessados em computação

sobre a Internet.

Há uma extensa produção acadêmica relacionada ao projeto Condor. Em [HSLL99] é apresentada uma descrição em linhas gerais do funcionamento deste sistema. Em [Wri01] é apresentado um trabalho que explora as possibilidades de escalonamento de tarefas oferecidas pela plataforma.

2.2.4 Mosix

MOSIX, um acrônimo para *Multicomputer Operating System for UNIX*, implementa um computador virtual paralelo em ambiente UNIX. Sua principal característica é o alto nível de integração entre as unidades de processamento integrantes do sistema, que podem variar de computadores pessoais a multiprocessadores com ou sem memória compartilhada. A integração entre estas unidades de processamento prevê transparência de localização na rede, cooperação entre processadores que permita a disponibilização de serviços multiprocessados, suporte para configuração dinânica e balanceamento de carga baseado em migração de processos.

Apesar de apresentar sua arquitetura como arbitrariamente escalável, o sistema MOSIX pressupõe a existência de uma rede de interconexão de alta velocidade entre suas unidades de processamento, o que indica uma grande demanda de comunicação entre estas unidades. De fato, como pode ser verificado em [BAR93], a topologia de interconexão entre as UPs do sistema é descentralizada, ou seja, exige que todos os nós contenham referências (neste caso endereços IP) a todos os demais nós. Esta abordagem compromete consideravelmente a escalabilidade do sistema.

O desenvolvimento deste sistema foi iniciado em 1981 e, apesar de não conter as propriedades básicas de um MPVC, foi aqui citado por sua importância histórica na pesquisa de sistemas operacionais distribuídos.

2.2.5 Plataformas Comerciais

Nos últimos dois anos, diversas plataformas que se propõem a realizar processamento maciçamente paralelo na Internet surgiram como iniciativas de empresas que enxergavam nesta possibilidade uma importante oportunidade de negócios. Em razão de seu caráter comercial, as empresas responsáveis por estas plataformas não disponibilizaram nenhuma documentação técnica mais aprofundada a respeito de seus produtos, o que inviabiliza a classificação dos mesmos segundo os critérios introduzidos neste capítulo. Entretanto, para que o ramo comercial de computação maciçamente paralela baseada na Internet possa ser melhor caracterizado, são apresentadas a seguir as principais plataformas comerciais atualmente em operação.

- **Popular Power**: em operação desde março de 2000, esta plataforma utiliza o poder computacional dos computadores disponíveis na Internet em projetos acadêmicos e sem fins lucrativos. Dentre os principais projetos estão as simulações para o desenvolvimento da vacina contra a gripe. Segundo a empresa que o administra, o Popular Power chegou a trabalhar com cerca de 10.000 computadores simultaneamente. Maiores informações podem ser encontradas em [Pop02].

- **Parabon**: em operação desde junho de 2000, esta plataforma segue a mesma filosofia do Popular Power, mas em projetos acadêmicos e comerciais. Dentre suas aplicações acadêmicas, as mais destacadas são aquelas que procuram contribuir à pesquisa da cura do câncer. Pode ser considerado um dos projetos mais bem sucedidos sob o ponto de vista comercial, pois possui diversas parcerias com institutos de pesquisa e com a renomada empresa Celera Genomics. Maiores informações podem ser obtidas em [Par02].

- **United Devices' Frontier**: também em operação desde junho de 2000, esta plataforma faz uso do poder computacional de computadores conectados à Internet ou a redes corporativas para a solução de problemas financeiros. Também disponibiliza ferramentas para o desenvolvimento de aplicações a serem executadas sobre esta plataforma. É outra bem sucedida iniciativa de exploração do poder de computadores ociosos, pois hoje conta com parceiros comerciais como a *Intel Corp.* e com colaboradores que lhe garantem um poder computacional de 25 teraflops. Maiores informações em [Fro02].

- **Entropia**: esta plataforma começou a ser utilizada comercialmente em agosto de 2000 e de uma forma bastante similar ao projeto da empresa United Devices, pois procura focalizar sua área de atuação no mercado financeiro. Também possui aplicações médicas, especialmente voltadas à pesquisa da cura da AIDS. Atualmente, a empresa encontra-se ativa e atuante no mercado e maiores informações podem ser obtidas em [Ent02].

2.2.6 SETI@Home

Iniciado em 1998, o projeto *SETI@Home*, cuja sigla significa *Search for Extra-Terrestrial Intelligence at Home*, é considerado a mais bem sucedida iniciativa de exploração do poder computacional de máquinas ociosas conectadas pela Internet.

Este projeto distingue-se dos demais projetos apresentados anteriormente por implementar um único tipo de processamento, exclusivamente dedicado a análise dos sinais captados pelo radio-telescópio de Arecibo, em Porto Rico, com o objetivo de identificar sinais de inteligência extra-terrestre. Esta simplicidade facilita a distribuição e a manutenção da plataforma, permitindo que um número maior de voluntários possa colaborar com o projeto. Estima-se que este projeto conte atualmente com cerca de 25 milhões de colaboradores em todo o mundo, representando um potencial superior a 40 teraflops.

A topologia utilizada para o gerenciamento do processamento realizado pelos componentes da plataforma é dividida em dois níveis. Num primeiro nível, há os servidores da plataforma que se interconectam segundo um anel, portanto numa topologia de difusão. No segundo nível estão os componentes trabalhadores, que se conectam ao anel de servidores para receber os dados a serem processados e retornarem os resultados obtidos.

A utilização destes dois modelos de interconexão dos componentes desta plataforma lhe confere grande versatilidade e escalabilidade. Como pôde ser visto no Cap. 1, a plataforma JOIN utiliza uma filosofia

similar a esta, permitindo entretanto diversas implementações para a topologia de interconexão das unidades coordenadoras.

Maiores informações acerca deste projeto podem ser encontradas em [SET02]

2.2.7 Plataformas Peer-to-Peer

Desenvolvidas com o propósito de permitir principalmente o compartilhamento de arquivos entre usuários de Internet, as plataformas *peer-to-peer* tornaram-se nos últimos anos uma das tecnologias mais populares. Apesar de não promoverem processamento distribuído, estas tecnologias utilizam técnicas de interconexão lógica que se assemelham às descritas no contexto dos mecanismos de balanceamento de carga.

A primeira plataforma a ser largamente utilizada com a finalidade de permitir a troca de arquivos entre usuários de Internet foi a *Napster.* Esta plataforma utilizava uma topologia hierárquica de três níveis para a interconexão dos nós sendo que os dois primeiros níveis eram dedicados aos servidores e o terceiro aos clientes.

Em virtude dos problemas legais relacionados à violação dos direitos autorais dos artistas cujos trabalhos eram distribuídos por esta plataforma, a mesma foi desativada pelo governo americano em meados de 2000. Após algumas semanas suspensa, a operação desta plataforma foi retomada sob a condição de que fossem compartilhados apenas trabalhos cuja distribuição não fosse impedida pelas leis internacionais de *copyright.* Depois disso, seu uso se tornou bastante restrito e ela acabou sendo desativada definitivamente em 2002.

No mesmo período em que a utilização do *Napster* se restringia, ganhou força o padrão *GnuTella*. Este padrão especifica protocolos e interfaces que definem a maneira como nós espalhados pela Internet devam ser implementados para que possam compartilhar arquivos de maneira escalável. Assim, diversas implementações deste mesmo padrão foram lançadas e hoje estas implementações formam a chamada rede *GnuTella.*

A principal implementação do padrão *GnuTella* é a plataforma *iMesh.* Esta é a primeira implementação que se utiliza da possibilidade de conexão dinâmica entre servidores. Através deste mecanismo, a topologia de interconexão entre os nós cresce segundo dois paradigmas: o primeiro, entre os servidores, é uma topologia de difusão que cresce de maneira não determinística e segundo as contingências locais; já a conexão entre servidores e clientes é feita segundo o paradigma mestre-escravo. Esta característica confere grande escalabilidade à plataforma que, por essa razão, é a mais utilizada atualmente entre os usuários de Internet.

Outra importante característica desta plataforma é que o caráter dinâmico de sua topologia dificulta em muito a suspensão de seu funcionamento por parte das autoridades, justificando a sua existência mesmo depois de plataformas similares já terem sido desativadas.

Todas as implementações mais recentes do padrão *GnuTella* de interconexão têm se valido desta importante versatilidade. Dentre eles, destacam-se o *KaZaA* e o *Morpheus (v 2.0).*

2.3 Conclusões

Neste capítulo, foi apresentado um panorama dos principais sistemas distribuídos e suas técnicas de balanceamento de carga, assim como algumas considerações relevantes à aplicação das mesmas em MPVCs.

No Cap. 3 será proposto um mecanismo de balanceamento de carga baseado em algumas das técnicas vistas neste capítulo, o qual procura atender os requisitos dos ambientes virtuais de computação maciçamente paralela.

Capítulo 3

Proposta de Balanceamento de Carga para MPVCs

Neste capítulo, será apresentado um mecanismo de balanceamento de carga especialmente desenvolvido para ambientes com as características comumente encontradas nos computadores virtuais maciçamente paralelos: a heterogeneidade das unidades de processamento e dos *links* de comunicação, o fraco acoplamento das unidades de processamento com seus coordenadores, a ampla disponibilidade de unidades de processamento e a grande magnitude da razão processamento / comunicação de suas aplicações.

A partir das técnicas apresentadas no capítulo 2 e dos requisitos a serem atendidos no domínio de interesse deste trabalho, será proposto um algoritmo de balanceamento de carga para MPVCs denominado *Escalonador de Tarefas Geracional com Replicação de Tarefas (GSTR)*.

Serão ainda introduzidas neste capítulo técnicas para o balanceamento de carga entre grupos heterogêneos distintos e uma linguagem de especificação de aplicações paralelas que, em conjunto com o escalonador GSTR, formarão o mecanismo de balanceamento de carga adequado aos modelos de aplicação e de MPVC apresentados no Cap. 1.

3.1 Balanceamento de carga intra-grupo

3.1.1 Escalonador Geracional - GS

Em [CWF+98] é apresentado um algoritmo cíclico de escalonamento de tarefas em sistemas distribuídos heterogêneos denominado Escalonamento Geracional (*Generational Scheduling* ou GS). Este algoritmo dinâmico utiliza uma heurística bastante simples na fase estática do escalonamento, o que lhe confere baixa complexidade computacional. O funcionamento deste escalonador pode ser dividido em quatro etapas.

1. Na primeira etapa, o problema de escalonamento é formalizado utilizando-se o conjunto de tarefas a serem executadas e as relações de precedência entre elas.

2. Na segunda etapa, todas as tarefas que dependem da execução de outras tarefas são temporariamente ignoradas, reduzindo assim o problema de escalonamento a um problema menor onde não há restrições de precedência. Tarefas já completadas e tarefas correntemente em execução são também desconsideradas.

3. Na terceira etapa, um algoritmo qualquer de escalonamento é utilizado para a resolução estática do subproblema formalizado na etapa anterior. As tarefas presentes nesta etapa serão aquelas que ainda não foram executadas, mas que já se encontram aptas para tal, e aquelas previamente escalonadas mas cuja execução ainda não foi iniciada.

4. Finalmente, na última etapa são detectados eventos de re-escalonamento, que dispararão novamente o algoritmo, de maneira a atualizar o estado do problema de escalonamento completo, ou seja, o que considera as relações de precedência entre tarefas.

Por evento de re-escalonamento entenda-se todo evento que potencialmente modifique a estrutura do conjunto de tarefas elegíveis à execução. Alguns exemplos de tais eventos são a conclusão de uma ou mais tarefas previamente escalonadas, a chegada de novas tarefas a serem escalonadas ou um tempo de execução de uma tarefa que seja significativamente diferente daquele que havia sido previamente estimado.

Como é comum nos escalonadores dinâmicos, pode-se perceber claramente a liberdade com que diferentes algoritmos de escalonamento podem ser empregados na etapa de resolução do problema parcial e na facilidade com que diferentes eventos, adaptados às condições e necessidades particulares de um determinado sistema, podem ser incorporados à etapa de re-escalonamento.

3.1.2 Escalonador Geracional com Replicação de Tarefas - GSTR

Nesta seção, será apresentada uma proposta de escalonador geracional com replicação de tarefas (GSTR), que pode ser entendido como uma extensão do escalonador apresentado na seção anterior, mais adequada ao domínio de interesse deste trabalho.

Conforme mencionado no preâmbulo deste capítulo, ambientes virtuais de computação maciçamente paralela têm como características o fraco acoplamento entre as unidades de processamento e seus respectivos coordenadores e a ampla disponibilidade de unidades de processamento. Uma das implicações dessas duas características é o fato de o sistema estar exposto à ocorrência de falhas em suas UPs.

A ocorrência de falhas, transitórias ou permanentes, nas UPs de um MPVC não deve comprometer o funcionamento da plataforma como um todo. Mais especificamente, a falha em uma UP não deve tornar o estado do sistema inconsistente e não deve impedir a conclusão das aplicações que possuam tarefas em execução nesta UP. Desta forma, um mecanismo de tolerância a falhas deve ser agregado ao sistema de modo a torná-lo robusto à ocorrência de falhas deste tipo.

Há diversas maneiras de se conseguir tolerância a falhas em um sistema distribuído: replicação ativa de componentes, mecanismos de *checkpoint e rollback* e utilização de transações, entre outros [Jal94]. Em um

MPVC, o emprego de mecanismos deste tipo, que invariavelmente impõem uma considerável carga sobre o sistema, é mais adequado em componentes centralizadores de informações, como os coordenadores e o servidor.

Já em componentes trabalhadores, a utilização de sofisticados mecanismos de recuperação de falhas está vinculada a uma alta taxa *custo / benefício* quando se considera a grande quantidade de componentes deste tipo. Portanto, a pergunta que se faz é: até que ponto é vantajosa a utilização de mecanismos recuperadores de falhas nos componentes trabalhadores se estes estão disponíveis em grande número e a baixo custo na plataforma ?

Diante destes argumentos, é razoável esperar que o escalonador de tarefas, que é o módulo responsável pelo gerenciamento das execuções das tarefas, seja capaz de contornar eventuais falhas ou mudanças de carga nas UPs às quais tenham sido enviadas tarefas de aplicações. Para isto, é necessário que o escalonador possua algumas propriedades.

Propriedade 1 *O escalonamento deve ser tolerante a falhas nas unidades de processamento.*

Os escalonadores estático e GS não prevêem nenhum mecanismo para contornar a ocorrência de falhas nas unidades de processamento e, por isso, não satisfazem essa condição.

Propriedade 2 *O escalonamento deve ser capaz de se adaptar a mudanças imprevistas nas cargas das unidades de processamento.*

De fato, em um ambiente como a Internet, no qual as unidades de processamento são computadores voluntariamente cedidos à plataforma e, com isso, não-dedicados ao processamento por ela realizado, mudanças imprevisíveis e significativas em suas cargas são não só possíveis como prováveis. A execução periódica de *benchmarks* para se avaliar em tempo real a carga dessas máquinas é quase sempre impraticável dadas as restrições de comunicação, de escalabilidade e de disponibilidade de poder computacional.

O que se propõe é introduzir um mecanismo de replicação de tarefas ao algoritmo GS para se conseguir satisfazer estes dois requisitos. A utilização de replicação de tarefas tem sido largamente explorada pela comunidade científica, porém raras são as publicações em que este tipo de mecanismo é associado a escalonadores de sistemas paralelos heterogêneos [AK94] [TOK98]. O algoritmo proposto pode ser descrito da seguinte forma.

1. Na primeira etapa, o problema de escalonamento é formalizado utilizando-se o conjunto de tarefas a serem executadas e as relações de precedência entre elas.

2. Na segunda etapa, todas as tarefas que dependam da execução de outras tarefas são temporariamente ignoradas, reduzindo assim o problema de escalonamento a um problema menor onde não há restrições de precedência. Tarefas já completadas são desconsideradas, mas tarefas em execução são levadas em conta no escalonamento.

3. Na terceira etapa, um algoritmo de escalonamento (trivial, *best-fit* ou ótimo) é utilizado para a resolução estática **de parte** do subproblema formalizado na etapa anterior. **A parte restante deste subproblema é escalonada dinamicamente à medida que as tarefas escalonadas estaticamente forem sendo completadas**. As tarefas presentes nesta etapa serão aquelas que ainda não foram executadas mas que já se encontram aptas para tal, tarefas previamente escalonadas mas cuja execução ainda não foi iniciada **e tarefas correntemente em execução.**

4. Finalmente, na quarta etapa são detectados eventos de re-escalonamento, que dispararão novamente o algoritmo de maneira a atualizar o estado do problema de escalonamento completo, ou seja, o que considera as relações de precedência entre tarefas.

Como pode ser observado pela descrição apresentada acima, as diferenças fundamentais entre os algoritmos GS e GSTR residem no fato de que, neste último, a etapa de escalonamento do subproblema é dividida em duas fases: uma de escalonamento estático e outra de escalonamento dinâmico. Além disso, nesta etapa são consideradas também as tarefas correntemente em execução.

A introdução do escalonamento dinâmico na terceira etapa deste algoritmo confere-lhe um grau adicional de flexibilidade em comparação ao GS. A proporção das tarefas que será escalonada estaticamente e dinamicamente é um problema em aberto e merece um estudo mais aprofundado. No momento foi arbitrado o limiar de 50 %, o que significa que metade das tarefas são escalonadas estaticamente e a metade restante é escalonada dinamicamente.

O escalonamento de tarefas em execução garante que a falha em uma unidade de processamento não impeça que a(s) tarefa(s) que estava(m) em execução nesta unidade seja(m) completada(s) já que, mais cedo ou mais tarde, ela(s) deverá(ão) ser re-escalonada(s) em outra unidade. Também garante que aumentos bruscos nas cargas de unidades de processamento participantes, que afetem significativamente o balanceamnto de carga promovido pelo escalonamento estático, tenham seu impacto atenuado sobre o desempenho global da execução da aplicação, já que o GSTR procurará distribuir as tarefas afetadas para outra(s) unidade(s).

Para que fique mais claro o funcionamento deste escalonador, considere a sequência de estados representada pelo sistema hipotético da Fig. 3.1. Nesta figura, o grande triângulo representa um sistema heterogêneo; nele, o hexágono representa a unidade coordenadora, a partir de onde serão distribuídas as tarefas de aplicação, e os retângulos representam as unidades de processamento. As unidades de processamento mais à esquerda, e desenhadas em tons mais fortes, são consideradas mais velozes que as mais à direita de acordo com algum critério de avaliação de desempenho.

Supondo que o sistema esteja inicialmente descarregado, ou seja, que nenhuma tarefa de aplicação esteja sendo executada, e que a unidade coordenadora receba um lote de seis tarefas (quadro I), considera-se que a primeira rodada do algoritmo GSTR defina o escalonamento estático apresentado no quadro II da figura. Supondo ainda que a primeira tarefa completada seja uma das enviadas ao primeiro trabalhador (quadro III), então é natural esperar que o algoritmo GSTR atribua a tarefa remanescente a esta unidade de processamento (quadro IV).

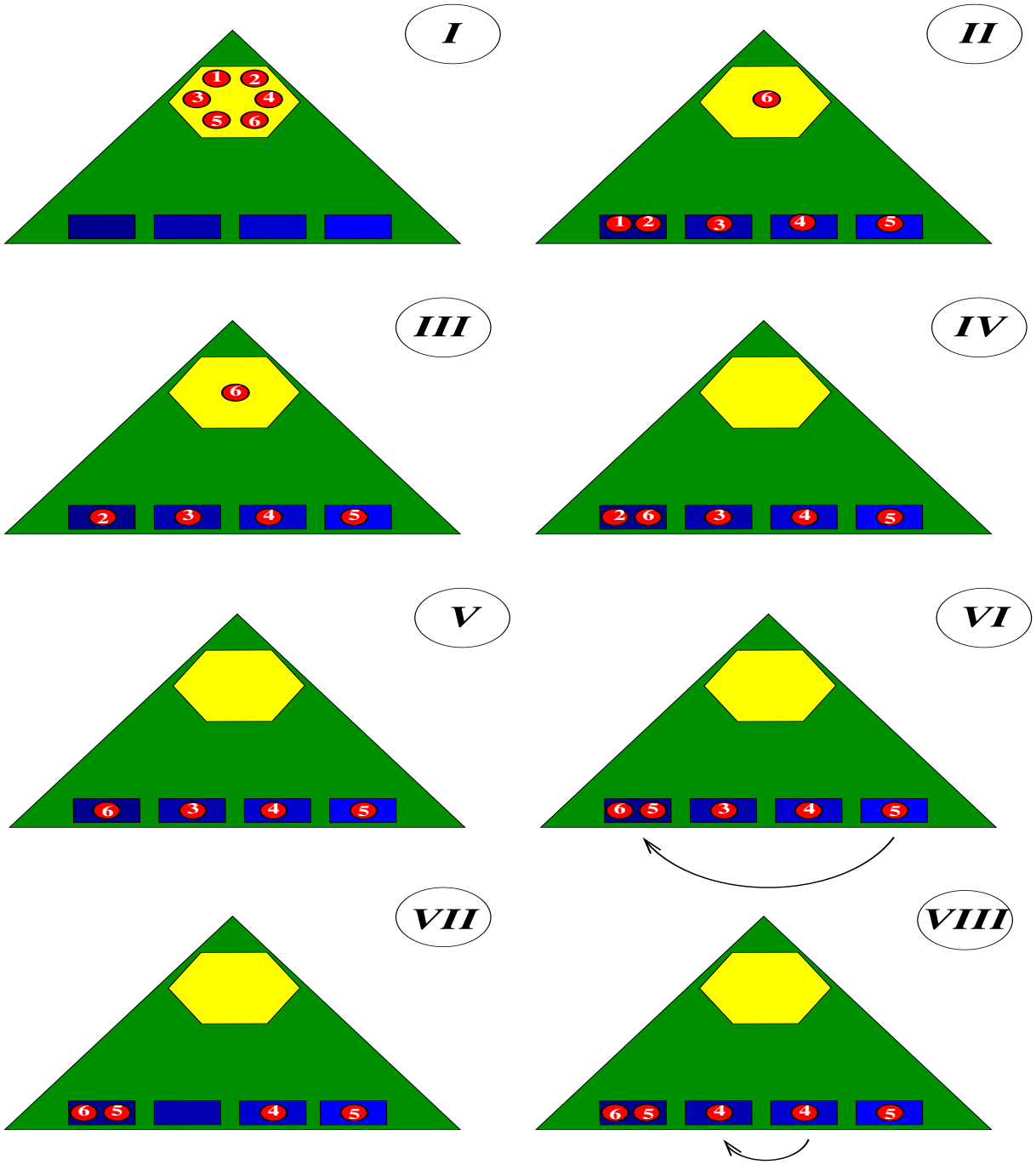

Figura 3.1: Exemplo de execução do algoritmo de escalonamento GSTR.

Cabe aqui destacar que o término desta primeira tarefa gerou um evento de re-escalonamento que induziu o GSTR a calcular a distribuição ideal para a tarefa remanescente, que foi atribuída à primeira unidade de processamento.

Depois de ter sido escalonada a última tarefa, supõe-se que a próxima tarefa a ser concluída seja a mais antiga tarefa em execução na primeira unidade de processamento (quadro V). O término desta tarefa, que também é encarado pelo sistema como um evento de re-escalonamento, induz o escalonador GSTR a reavaliar sua situação. Neste ponto, não há mais nenhuma tarefa pendente (pronta para execução mas em espera) e por isso o escalonador decide replicar a tarefa que está em execução na quarta unidade de processamento (mais lenta) para a primeira, de onde foi gerado o evento de re-escalonamento (quadro VI).

Supondo-se agora que a próxima tarefa a ser concluída seja aquela em execução na segunda unidade de processamento (quadro VII), o evento de re-escalonamento gerado pelo término desta tarefa pode, similarmente ao ocorrido na etapa anterior, induzir a replicação na segunda unidade de processamento da tarefa em execução na terceira unidade de processamento (quadro VIII).

Pode-se perceber por esta sequência que, a cada evento de re-escalonamento (neste caso foram considerados apenas os términos das tarefas), o GSTR pôde alocar uma tarefa pendente ou em execução a uma unidade de processamento total ou parcialmente ociosa. Outros eventos de re-escalonamento poderiam ser considerados, como por exemplo o término de um período de tempo pré-determinado, que forçaria o GSTR a re-escalonar suas tarefas periodicamente.

Uma vez apresentado informalmente o algoritmo de escalonamento GSTR, pode-se agora descrevê-lo com maior rigor. A Fig. 3.2 apresenta uma descrição formal deste algoritmo.

Na seção a seguir, será apresentada uma proposta de escalonamento entre sistemas heterogêneos que pode ser utilizada em conjunto com o escalonamento GSTR.

3.2 Balanceamento de carga inter-grupos

Os algoritmos de escalonamento de tarefas apresentados na seção anterior pressupõem a existência de uma unidade coordenadora, onde será realizado todo o gerenciamento da distribuição das tarefas entre as unidades de processamento. Este cenário corresponde ao conceito de grupo (definição 8), mas para se realizar o balanceamento de carga em um PVC/MPVC, considerações adicionais devem ser feitas.

Para se distribuir a carga computacional representada por uma aplicação paralela entre grupos heterogêneos, deve-se promover algum particionamento na aplicação de modo a se obter o maior grau possível de paralelismo na execução de suas tarefas. A dificuldade neste caso é encontrar um particionamento tal que distribua a carga computacional de forma equânime entre os grupos heterogêneos (GHs) participantes e ainda permita que as partes direcionadas a cada um dos GHs possam ser executadas o mais independentemente possível umas das outras.

```
- receber aplicação paralela.
- definir conjunto Te de tarefas em execução como sendo vazio.
- definir o número máximo de tarefas que podem ser alocadas
simultaneamente para cada UP como uma função do poder computacional
dessa UP. O poder computacional das UPs deve ser estimado a partir
da execução de um conjunto de benchmarks.
- enquanto a aplicação não for concluída:
    - verificar se existe alguma tarefa pronta para ser executada que
    ainda não tenha sido enviada a nenhuma UP,
    - se existir,
        - essa tarefa é selecionada para escalonamento,
    - senão,
        - selecionar a tarefa menos replicada e em execução há menos
        tempo para ser replicada,
    - com a tarefa selecionada, fazer o seguinte:
        - verificar se existe disponibilidade em alguma das UPs do grupo,
            - se existir mais de uma UP com disponibilidade, então,
            selecionar a UP de maior poder computacional,
            - se não existir, aguardar até que alguma UP finalize a
            execução de alguma tarefa,
        - escalonar a tarefa na UP selecionada,
        - atualizar o conjunto Te de tarefas em execução,
    - verificar a ocorrência de eventos de re-escalonamento,
    - se ocorreu,
        - se evento foi ''fim-de-tarefa'',
            - se este for o primeiro resultado para esta tarefa,
                - armazenar resultado recebido,
                - cancelar execução de eventuais réplicas desta tarefa,
            - senão,
                - desconsiderar resultado recebido,
            - atualizar carga das UPs envolvidas, diminuindo-se
            a carga da UP que enviou o resultado e de todas as UPs
            que estavam executando uma réplica desta tarefa,
```

Figura 3.2: Algoritmo de balanceamento de carga intra-grupos GSTR

Enunciado desta forma, o problema de balanceamento de carga entre GHs parece ser muito semelhante ao problema intra-GH, discutido na seção anterior. De fato, é necessário conhecer o poder computacional de cada GH para se realizar uma distribuição justa de trabalho, o que pode ser feito utilizando-se os mesmos resultados obtidos anteriormente durante a avaliação de desempenho das UPs de um grupo, necessária ao funcionamento do algoritmo de balanceamento intra-grupo GSTR.

Após estas considerações, pode ser apresentado o algoritmo de balanceamento de carga entre grupos.

3.2.1 Particionamento em Agrupamentos Paralelos Proporcionais - PPCP

Nesta seção, será proposto um algoritmo para o particionamento de aplicações paralelas denominado **Particionamento em Agrupamentos Paralelos Proporcionais (ou Parallel and Proportional Cluster Partitioning - PPCP)**, que tem como objetivo dividir uma aplicação em partes que possam ser executadas em paralelo e que demandem aproximadamente a mesma carga relativa dos grupos para onde forem submetidas.

Para atender ao primeiro destes dois requisitos, isto é, realizar o particionamento da aplicação em porções que possam ser executadas o mais independentemente possível, aproveita-se a relação multiplicativa existente entre os lotes de tarefas da aplicação (definição 4). Por esta relação multiplicativa, sabe-se que se um lote de tarefas fornece seus resultados como entrada de dados a outro lote de tarefas, então a multiplicidade de um destes lotes é um múltiplo ou sub-múltiplo da multiplicidade do outro. Através desta relação multiplicativa, é sempre possível dividir um conjunto de lotes de tarefas que possuam relações de dependência de dados em grupos totalmente independentes.

Para que isto fique mais claro, considere como exemplo dois lotes de tarefas com, por exemplo, 5 e 10 tarefas, que possuam alguma relação de dependência de dados entre si. Tendo-se como referência o lote de cardinalidade 5 e sabendo-se que cada tarefa deste lote se liga logicamente a duas tarefas do lote de cardinalidade 10, então pode-se perceber que qualquer particionamento realizado no primeiro automaticamente indicará um particionamento no segundo se forem preservados os fluxos de dados entre as tarefas. Assim, se por exemplo o primeiro lote for dividido em dois outros lotes com 3 e 2 tarefas, então o segundo lote será dividido em dois outros com 6 e 4 tarefas formando dois grupos independentes com dois lotes cada. A Fig. 3.3 ilustra este particionamento.

Pode-se generalizar esta regra utilizando-se o conceito de *divisor comum*. Dado um conjunto de lotes de tarefas que possuam quaisquer relações de dependência de dados entre si e sendo K um divisor comum destes lotes de tarefas, então qualquer particionamento realizado em K automaticamente indicará um particionamento em todos os lotes deste conjunto já que a cardinalidade destes lotes é necessariamente um múltiplo de K. Além disso, num conjunto de lotes que contenham relações de dependência de dados, sempre existirá um divisor comum $K \geq 1$ em função da relação multiplicativa entre as cardinalidades destes lotes.

Para ilustrar esta técnica, considere um lote de tarefas de cardinalidade 15, que forneça dados a outro lote de tarefas de cardinalidade 30, que forneça dados a um lote de tarefas de cardinalidade 10, que forneça dados a um lote de tarefas de cardinalidade 20. Um divisor comum de $\{10,15,20,30\}$ é 5. Qualquer partici-

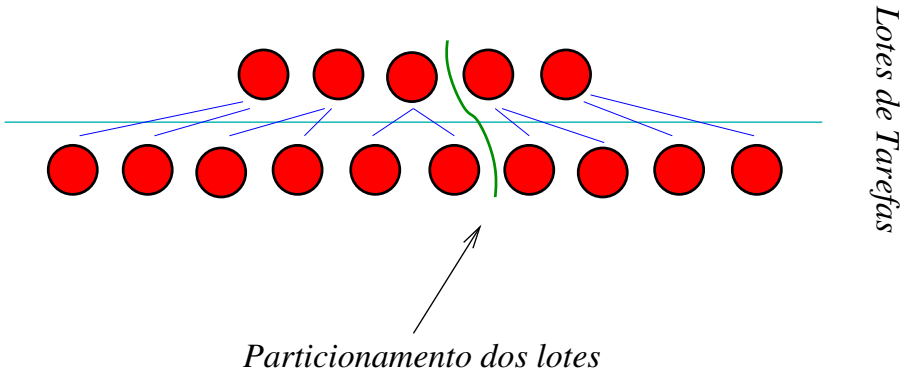

Figura 3.3: Exemplo de particionamento de dois lotes de tarefas com dependência de dados.

onamento realizado em 5, como por exemplo $\{1,4\}$ ou $\{2,3\}$, automaticamente indica o particionamento a ser feito nos lotes do conjunto considerado. Percebe-se ainda que, quanto maior o valor do divisor comum tomado como referência, maior o número de particionamentos possíveis, sendo portanto o *maior divisor comum* o melhor candidato para este papel.

Assim, para se particionar uma aplicação em porções que possam ser executadas o mais independentemente possível, o primeiro passo consiste em agrupar os lotes que possuam alguma relação de dependência de dados entre si e extrair o MDC das cardinalidade deste lotes. Esse valore de MDC servirá como "guia" de particionamento, conforme descrito no parágrafo anterior.

Uma vez identificado um mecanismo para o particionamento da aplicação em grupos independentes, resta saber como fazer esta divisão de maneira a manter o equilíbrio de cargas entre os grupos. Para isso, retoma-se aqui o conceito introduzido ao final da seção anterior. Tendo-se em mãos o poder computacional de cada um dos grupos do MPVC e o números de tarefas correntemente alocadas a estes grupos, pode-se calcular a carga computacional nos grupos e utilizar este parâmetro para balizar o particionamento da aplicação.

Para se ponderar a divisão dos lotes de tarefas entre os GHs, introduz-se aqui o conceito de *potencial relativo* (PR) de um GH como sendo o quociente *poder computacional do GH / somatória dos poderes computacionais de todos os GHs (ou poder do MPVC)*. Os poderes relativos de cada GH formam um conjunto de valores que refletem a prioridade com que tarefas deverão ser alocadas a cada GH, e por isso indicam a maneira como os MDCs deverão ser particionados para se manter a carga equilibrada.

Um ponto adicional que deve ser destacado é o que diz respeito ao arredondamento na divisão dos lotes de tarefas entre os grupos. Como é bem provável que os potenciais relativos não sejam números inteiros, ou que a multiplicação destes pelos MDCs não forneçam números inteiros, é preciso definir uma política de arredondamento que preserve o balanceamento de carga entre grupos. Para isso, introduz-se neste ponto o conceito de *carga relativa* (CR) de um GH, que é o quociente *número de tarefas já alocadas ao GH / poder computacional do GH*. A carga relativa fornece uma medida do grau de balanceamento de carga entre os

grupos, e pode ser utilizada no arredondamento da divisão dos lotes da seguinte forma: partindo-se do GH de menor carga relativa ao GH de maior carga relativa, executa-se a divisão do MDC segundo os potenciais relativos arredondando-se sempre para cima o valor calculado, caso este seja fracionário.

Integrando-se todas essas idéias e adicionando-se cláusulas que tratem os casos especiais de lotes de tarefas pequenos, tem-se o algoritmo de balanceamento descrito na Fig. 3.4.

```
- calcular a carga relativa (CR) em cada um dos GHs,
    - calcular o quociente entre o número de tarefas já alocadas ao GH
    e o poder computacional deste GH,
- atribuir os lotes de cardinalidade pequena para o GH com menor CR:
    - selecionar o GH com menor CR,
    - atribuir a ele os lotes de tarefas com multiplicidade
    menor ou igual ao número de UPs deste grupo,
- dividir a aplicação em conjuntos de lotes que possuam alguma
dependência de dados entre si:
    - para cada lote que ainda não faça parte de nenhum
    grupo e que não tenha sido atribuído ao GH inicial,
        - incluir este lote num novo grupo e todos os demais lotes
        que puderem ser atingidos por uma busca em profundidade
        a partir dele,
- para cada conjunto de lotes,
    - calcular o MDC das cardinalidades dos lotes do conjunto e
    armazená-la na variável K,
    - definir o particionamento que será feito nos lotes do grupo,
        - para cada GH, indo do de menor CR ao de maior CR,
            - calcular o potencial relativo (PR) deste GH considerando-se
            somente os GHs que ainda não receberam tarefas,
            - multiplicar K ao PR deste GH,
            - se resultado não for inteiro, arredondar para cima,
            - subtrair resultado de K,
    - aplicar o particionamento definido pelos valores de K sobre todos os
    lotes do conjunto:
        - para cada lote,
            - aplicar um particionamento proporcional ao feito em K,
- fim.
```

Figura 3.4: Algoritmo de balanceamento de carga inter-grupos PPCP

Para que o funcionamento do procedimento descrito acima fique mais claro, a seguir é apresentado um exemplo ilustrativo.

- Um PVC é formado por 3 GHs de poderes computacionais 3, 5 e 2 respectivamente. O primeiro GH já possui 10 tarefas alocadas; o segundo possui 22 e o terceiro possui 8.

- Ao PVC é submetida a ordem de execução da aplicação apresentada na Fig. 1.1. Para a execução do particionamento desta aplicação, são seguidos os seguintes passos:
  - Calcula-se a carga relativa dos GHs. Pelo enunciado anteriormente, as CRs são portanto 10/3=3,33, 22/5=4,44 e 8/2 = 4, respectivamente.
  - O GH de menor CR é o primeiro GH. A ele deverão ser atribuídas os lotes de tarefas de multiplicidade menor ou igual a 3, que é o número de UPs neste GH. Os lotes com essas características são o 1 e o 5.
  - Divide-se a aplicação em grupos de lotes. Partindo-se do lote 2, é possível atingir os lotes 3 e 4 por uma busca em profundidade no grafo de dependências de dados entre lotes. Esses lotes formarão o único grupo de lotes.
  - Sobre este grupo de lotes é realizado o seguinte processamento:
    * Calcula-se o MDC das multiplicidades dos lotes do grupo: 5, 10 e 20. Neste caso, o MDC é 5. Atribui-se este valor a K.
    * Toma-se o GH de menor CR, ou seja, o primeiro GH.
    * O cálculo do PR deste GH se dá da seguinte forma: este GH possui poder computacional igual a 3 e todos os GHs que ainda não receberam tarefas possuem juntos poder igual a 3+5+2=10 UPs. Então, seu PR é igual a 3/10 = 0,3.
    * A multiplicação de 5 (valor inicial de K, definido pelo MDC) por 0,3 resulta em 1,5. Isso significa que de um lote de 5 tarefas, 1,5 tarefa deve ser enviada para o primeiro GH de acordo com seu potencial relativo.
    * Não é possível atribuir 1,5 tarefa a um GH. Arredonda-se esse valor para cima, ou seja, para 2. Subtrai-se em seguida 2 de K.
    * Toma-se agora o terceiro GH, que possui CR igual a 4. Seu PR será calculado por 2/(5+2)=0,285.
    * A proporção de tarefas que devem ser alocadas a este GH é calculada por 3 (valor de K) $\times$ 0.285 (PR) = 0.857.
    * Analogamente ao realizado para o GH anterior, este valor é arredondado para cima obtendo-se 1 tarefa para o terceiro GH. O valor de K passa a ser 2.
    * Como só resta um GH, a proporção de tarefas restante é atribuída a ele, ou seja, 2 tarefas.
    * O particionamento do MDC é então igual a $\{2,2,1\}$. Para se calcular a distribuição dos lotes reais do grupo, basta verificar a razão entre a multiplicidade de cada um destes lotes e o valor do MDC. O procedimento é o seguinte:
      · O primeiro lote do grupo possui multiplicidade 10. A razão entre a multiplicidade deste lote e o MDC é 10/5=2. A distribuição deste primeiro lote deverá então ser $2\times\{2,2,1\} = \{4,4,2\}$.
      · O segundo lote do grupo possui multiplicidade 5. A razão entre a multiplicidade deste lote e o MDC é 5/5=1. A distribuição deste segundo lote deverá então ser $1\times\{2,2,1\} = \{2,2,1\}$.
      · O terceiro lote do grupo possui multiplicidade 20. A razão entre a multiplicidade deste lote e o MDC é 20/5=4. A distribuição deste terceiro lote deverá então ser $4\times\{2,2,1\} = \{8,8,4\}$.
    * Não há mais lotes neste grupo. A distribuição está completa.

A Fig. 3.5 ilustra o resultado do particionamento realizado. Como pode ser verificado, este algoritmo tem como característica particionar os grupos de lotes de tarefas de forma tal que cada parte possa ser

submetida a um GH e executada sem que este necessite comunicar-se durante sua execução. Desta forma, os únicos pontos de sincronização são os correspondentes aos lotes de tarefas inicialmente alocados ao GH de menor CR, pontos esses de onde partem todas as ordens de execução das partições e onde chegam os resultados dessas execuções.

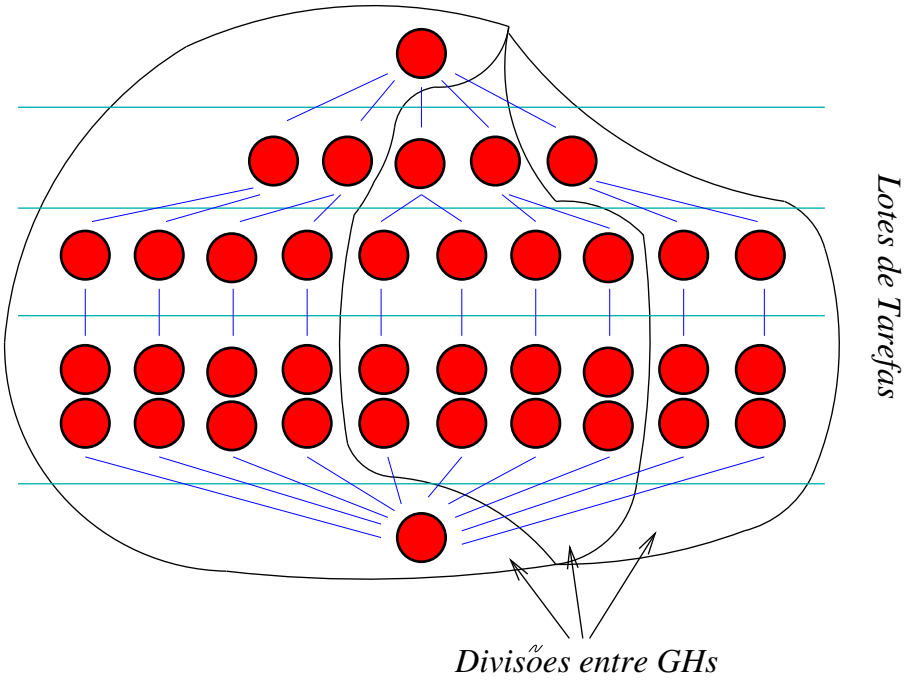

Figura 3.5: Particionamento promovido pelo algoritmo PPCP na aplicação paralela exemplificada.

Na próxima seção, será apresentada a linguagem PASL, desenvolvida para a especificação de aplicações paralelas aderentes ao modelo de aplicações introduzido pela definição 5.

3.3 Uma linguagem de especificação de aplicações paralelas - PASL

Nesta seção será introduzida uma linguagem para a especificação de aplicações paralelas que, juntamente com os algoritmos GSTR e PPCP, forma o mecanismo de balanceamento de carga proposto neste trabalho. A motivação da introdução desta linguagem de especificação reside na necessidade de um mecanismo para a especificação das aplicações, suas tarefas, seus lotes de tarefas e as relações de precedência entre esses lotes, para que um algoritmo de escalonamento de aplicações paralelas possa ser implementado em um PVC qualquer.

Esta linguagem de especificação foi concebida de forma a seguir o modelo de aplicação paralela proposto no Cap. 1, atendendo os seguintes requisitos:

- definir as tarefas da aplicação atribuindo um identificador lógico a um bloco de código que possua

interfaces de entrada e saída bem definidas,

- definir os lotes de tarefas da aplicação,
- definir as relações de precedência entre os lotes de tarefas da aplicação descrevendo os fluxos de dados entre eles,
- descrever o tipo de computação realizado pelas tarefas.

Para a definição proposta no primeiro item, escolheu-se uma forma sintática intuitiva e que descrevesse de maneira genérica o modo como o código de tarefas de aplicações paralelas são geralmente dispostas em sistemas distribuídos. Assim, foram definidos a primitiva *path*, que procura relacionar os tipos de tarefas à localização lógica de seu código, seja ela dentro de um sistema de arquivos local ou em um servidor remoto, e os identificadores de tipos de tarefas que, seguindo o padrão proposto em [VG96], são formados pela justaposição da letra *L* com um identificador numérico.

Assim, para identificar três tipos de tarefas que estejam localizados, por exemplo, em um diretório local de nome /app/test, utiliza-se a seguinte seqüência de instruções:

```
path = "/app/test";
L1 = "block1";
L2 = "block2";
L3 = "block3";
```

A primitiva *path* pode ser utilizada mais de uma vez dentro de uma mesma especificação caso os blocos de computação de uma determinada aplicação encontrem-se em locais distintos. Neste caso, a ordem com que as instruções são apresentadas é de fundamental importância para a localização dos blocos de código. No exemplo apresentado a seguir, os blocos 0 e 1 encontram-se no diretório /app/test1 enquanto que o bloco 2 encontra-se no diretório /app/test2.

```
path = "/app/test1";
L1 = "block1";
L2 = "block2";
path = "/app/test2";
L3 = "block3";
```

As definições dos lotes de tarefas da aplicação e suas relações de precedência se dão em um mesmo bloco de instruções na PASL. A sintaxe das instruções deste bloco é descrita como:

$$B[bid] = L[tid](c) \ << \ [B[lid] \ [, B[lid]]^{+} \ |Storage|NULL] \ [>> \ Storage]^{*} \ ;$$

Nesta expressão, bid representa um identificador de lote de tarefa e tid representa um identificador de tarefa, ambos formados por números inteiros. O termo c indica a cardinalidade do lote; o operador

$<<$ é utilizado para se indicar a origem dos dados a serem entregues ao lote; estes dados podem vir de outro(s) lote(s) ou de um repositório qualquer de dados, tal como um arquivo local ou um arquivo remoto. A utilização da constante $NULL$ indica que o lote de tarefas não necessita de dados de entrada para ser executado. O operador $>>$ é opcional e deve ser utilizado quando os dados de saída de um lote tiverem de ser armazenados em um repositório de dados.

Para que a ligação de dados entre lotes de cardinalidades distintas seja possível, admite-se aqui que suas respectivas tarefas possuirão interfaces de entrada e saída de dados que suportarão essa ligação. Essas interfaces dependerão da linguagem de programação em que as tarefas forem escritas e por isso não são especificadas pela linguagem PASL.

Uma importante restrição imposta pela linguagem PASL deve ser destacada neste ponto: operações de escrita e leitura em repositório de dados só podem ser feitas por lotes de tarefas de multiplicidade um. Esta restrição se faz necessária para que a linguagem PASL seja completamente independente da linguagem de programação em que as tarefas estejam escritas. Se esta restrição não fosse imposta, não seria possível deduzir, pela simples apreciação da especificação PASL, como os dados de um repositório seriam distribuídos entre as m tarefas de um lote ou como estas m tarefas escreveriam num repositório de dados já que, para isso, seria necessário se saber o tipo de estrutura de dados manipulado por cada uma das tarefas.

O exemplo a seguir mostra como a aplicação hipotética representada pelo diagrama apresentado na Fig. 1.1 poderia ser descrita pela linguagem PASL:

```
path = "/app/test1";
B1 = L1(1) << NULL;
B2 = L2(5) << B1;
B3 = L3(10) << B2;
B4 = L4(20) << B2;
B5 = L5(1) << B3,B4 >> ``results'';
```

Com o objetivo de promover maior fatoração nas especificações geradas pela linguagem PASL, introduz-se aqui as primitivas $beginblock(n)$ e $endblock$. Estas primitivas delimitam blocos aninháveis de lotes de tarefas cuja execução se repetirá por n vezes. O exemplo a seguir ilustra a aplicação destas duas primitivas.

```
B1 = L1(1) << NULL >> ``temp'';
beginblock(10);
    B2 = L2(1) << ``temp'';
    B3 = L3(25) << B2;
    B4 = L4(1) << B3 >> ``temp'';
endblock;
```

Na primeira execução deste bloco, o arquivo “temp” conterá os dados gerados pela execução do lote 0. O lote 1 utilizará estes dados em sua execução e, à medida que sucessivas execuções do lote 3 forem

sendo realizadas, novos dados serão armazenados no arquivo "temp" e serão utilizados pelo lote 1 em sua execução seguinte. O bloco do exemplo será executado 10 vezes.

A restrição imposta pela linguagem PASL, de que escritas e leituras em repositórios só podem ser feitas por tarefas de multiplicidade 1, gera uma certa redundância na especificação do fluxo de dados da aplicação hipotética do exemplo. De fato, esta redundância faz parte do preço a ser pago pela generalidade com que aplicações paralelas podem ser descritas pela linguagem PASL e pelas restrições impostas pelo modelo de aplicação proposto. Um artifício que violaria o modelo inicial mas que reduziria grandemente esta redundância seria:

```
beginblock(10);
    B1 = L1(1) << B3;
    B2 = L2(25) << B1;
    B3 = L3(1) << B2;
endblock;
```

A ambigüidade existente nas relações de precedência entre as tarefas pode ser resolvida pela ordem com que estas são apresentadas no bloco repetitivo. Desta forma, a primeira execução do lote 0 é realizada sem que nenhum dado seja enviado ao mesmo. Nas execuções subseqüentes, serão enviados os dados gerados pelo lote B2 eliminando-se assim a necessidade da manipulação redundante de arquivos temporários. Entretanto, apesar de mais simples, esta especificação introduz outras ambigüidades que dificultam o entendimento e, consequentemente, a manutenção de aplicações paralelas assim desenvolvidas. Além disso, este artifício claramente viola a primeira restrição imposta sobre as relações de precedência do modelo de aplicação adotado. Por essas razões, este tipo de construção não será adotado neste trabalho, tendo sido aqui apresentado apenas por razões ilustrativas.

Laços de repetição condicional também poderiam ser adicionados à sintaxe da linguagem. Porém, este recurso traz uma série de conseqüências e será reservado para extensões futuras da linguagem PASL.

Para a descrição do tipo de computação realizado pelas tarefas, será adotada a sintaxe da linguagem de modelagem PAMELA, proposta em [VG96]. Nesta linguagem, são utilizadas abstrações que procuram relacionar cada tipo de operação realizada no bloco de computação com um período de tempo característico. Assim, seqüências de operações são descritas como sucessões de "atrasos", cada qual relacionado a um tipo de operação (operação matemática em ponto flutuante, atribuições, desvios condicionais etc). A utilização deste tipo de descrição na modelagem dos tipos de tarefa de uma aplicação possibilita uma resolução mais precisa do problema de escalonamento estático, além de viabilizar uma predição analítica de performance de execução das aplicações.

O agrupamento destes blocos em uma única especificação de aplicação é trivial. Adicionam-se ainda a este conjunto duas primitivas que definam o nome da aplicação e provejam uma breve descrição da mesma, além de suporte a comentários no estilo utilizado em programação C/C++ e Java.

Um exemplo de uma especificação completa é mostrado a seguir.

```
// Header Section

name =          "Sample Application";
description =   "A simple test";

%%      // This is a separator

// Assigment section

path =  "/app/test1";
L1 =    "block1";
L2 =    "block2";
path =  "/app/test2";
L3 =    "block3";
L4 =    "block4";

%%      // This is a separator

// Data link section

B1 = L1(1) << NULL >> ``temp'';
beginblock(10);
    B2 = L2(1) << ``temp'';
    B3 = L3(25) << B2;
    B4 = L4(1) << B3 >> ``temp'';
endblock;
```

Como o modelo PAMELA de cada tipo de tarefa está unicamente relacionado ao seu código, independentemente da(s) aplicação(ões) em que esta tarefa é utilizada, não parece razoável incluir estes modelos nas especificações das aplicações. Para cada tipo de tarefa, deverá existir uma descrição PAMELA do tipo de computação realizada por ela. A seguir é apresentado um exemplo de especificação PAMELA para uma tarefa hipotética.

```
L1 = delay(t1);A*delay(t2);B*delay(t3);
```

Segundo este modelo, a execução da tarefa L0 demanda uma unidade de tempo t1 mais “A” unidades de tempo t2 mais “B” unidades de tempo t3. Cada uma destas unidades de tempo deve estar relacionada a uma operação ou a um conjunto pré-determinado de operações que possuam tempos de execução conhecidos.

3.4 Conclusões

Neste capítulo foi apresentado um mecanismo de balanceamento de carga para MPVCs definidos segundo o modelo introduzido no Cap. 1. Este mecanismo pode ser dividido em três partes: o particionador PPCP, responsável pelo balanceamento de carga computacional entre grupos distintos, o escalonador GSTR, responsável pelo balanceamento de carga computacional dentro de um mesmo grupo, e a linguagem PASL, responsável pela especificação das aplicações paralelas que serão executadas por estes componentes.

Uma vez estabelecidas as bases teóricas sobre as quais o escalonamento de tarefas será realizado, o próximo passo é aplicar estes conceitos sobre um MPVC. Desta forma, no capítulo seguinte será apresentada a descrição do funcionamento interno dos componentes da plataforma JOIN e, em seguida, serão descritos os detalhes da implementação deste mecanismo de escalonamento para esta plataforma.

Capítulo 4

Balanceamento de carga na plataforma JOIN

Neste capítulo serão descritos os módulos desenvolvidos para a implementação, na plataforma JOIN, do mecanismo de balanceamento de carga proposto no Cap. 3. Uma descrição mais aprofundada dos mecanismos apresentados neste capítulo pode ser encontrada no Apêndice A.

4.1 A estrutura dos serviços em JOIN

Para que o mecanismo de balanceamento de carga proposto no Cap. 3 pudesse ser implementado na plataforma JOIN, diversas considerações referentes à estrutura interna dos componentes JOIN tiveram de ser feitas.

Conforme já citado no capítulo introdutório, há quatro componentes distintos que definem a estrutura lógica da plataforma JOIN: o servidor, os coordenadores, os trabalhadores e os *jacks*, que são componentes especiais de configuração e administração. Além de introduzir uma estrutura **escalável**, esta organização em componentes também define a organização lógica com que todo software de sistema deverá ser concebido para que seja plenamente compatível com a plataforma. Em se tratando dos módulos de serviço, que são os responsáveis pela implementação de todas as funcionalidades presentes na plataforma, essa organização é imperativa.

Sendo assim, cada módulo de serviço é formado sempre por quatro partes distintas, cada uma delas dedicada a um componente do JOIN. Desta forma, cada módulo de serviço possui um sub-módulo do tipo servidor, um sub-módulo do tipo coordenador, um sub-módulo do tipo trabalhador e um sub-módulo do tipo jack. Além disso, estes sub-módulos de serviço devem interagir segundo a estrutura imposta pela plataforma JOIN ou seja, trabalhadores comunicando-se somente com seu coordenador, coordenadores comunicando-se com trabalhadores, coordenadores e servidor, e servidor comunicando-se com coordenadores e jacks.

Estas restrições impostas sobre os fluxos de dados da plataforma estão presentes no modelo de MPVC

apresentado no Cap. 1 e existem por razões de escalabilidade; é possível perceber como o estabelecimento indiscriminado de conexões entre sub-módulos de um mesmo serviço pode levar a plataforma rapidamente a um estado de saturação. Entretanto, não é possível garantir que o fluxo de informações entre sub-módulos de serviços siga sempre o curso pretendido pela organização lógica do JOIN; de fato, não há nada que impeça um serviço de distribuir informações entre seus sub-módulos que permitam, por exemplo, que trabalhadores se comuniquem entre si. Esta prática é, no entanto, francamente desincentivada.

Um outra restrição imposta por razões de modularidade e escalabilidade diz respeito à comunicação entre módulos de serviços; serviços diferentes só podem comunicar-se dentro de um mesmo componente da plataforma JOIN. Assim, os sub-módulos de serviço que estejam sendo executados em um coordenador da plataforma só podem interagir com outros módulos de serviço através dos seus respectivos sub-módulos que estejam sendo executados neste mesmo coordenador. Caso um sub-módulo de serviço necessite de uma informação detida por um sub-módulo de outro serviço que esteja localizado em outra máquina, deverá efetuar a requisição ao sub-módulo que se encontra localmente e este se encarregará de encaminhar esta requisiçào ao sub-módulo remoto que detém a informação.

Estes modelos de comunicação encontram-se representados graficamente pelo diagrama da Fig. 4.1.

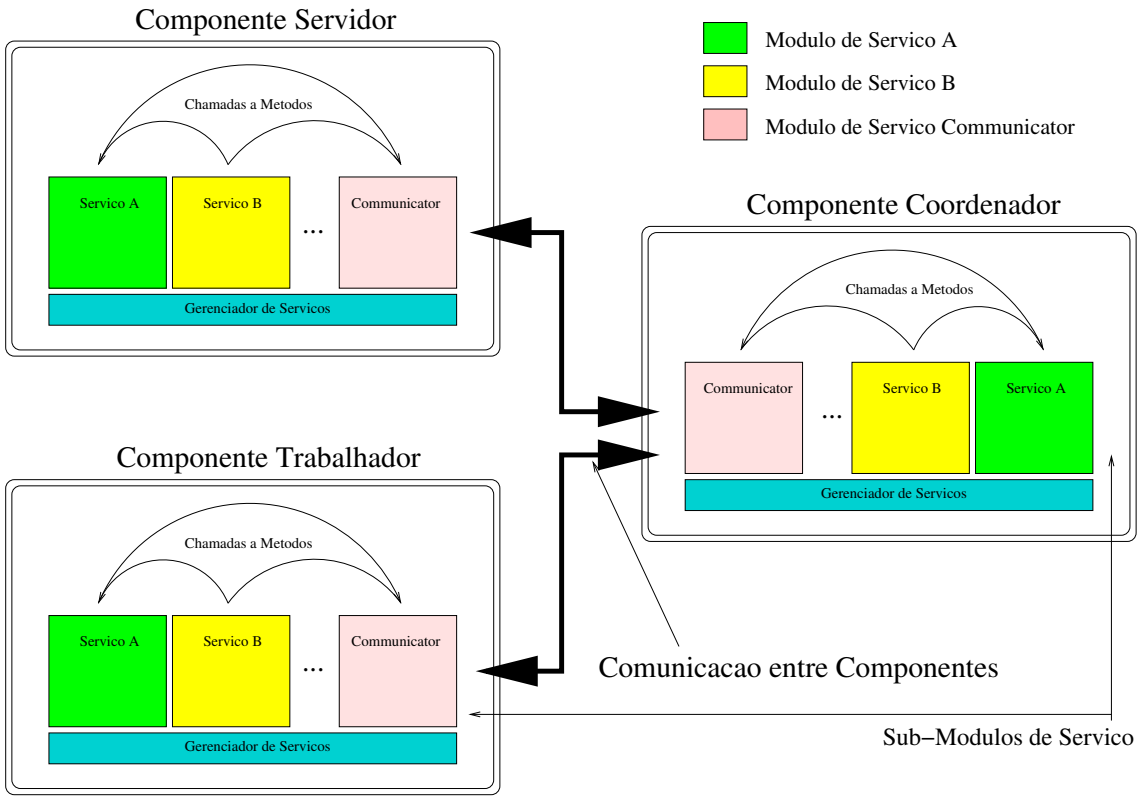

Figura 4.1: Modelo de interação entre sub-módulos de serviço da plataforma JoiN.

Além da organização em sub-módulos, cada serviço deve possuir sua estrutura interna definida pelas interfaces de software correspondentes à funcionalidade que pretende implementar. Assim, para cada funcionalidade implementável por um módulo de serviço, há um conjunto de interfaces pré-definidas abrangendo todos os seus sub-módulos.

Há ainda uma interface básica que todos os sub-módulos de serviço devem implementar. Denominada *ServiceCode*, essa interface procura organizar algumas das tarefas críticas a serem realizadas pelos sub-módulos em métodos claramente definidos. Com isso, espera-se que, controlando-se o acesso dos sub-módulos a certos recursos compartilhados, minimize-se o risco de que, num dado momento, o estado interno de algum serviço torne-se inconsistente. Maiores detalhes sobre esse interface podem ser encontradas no Apêndice A.

A seguir, são apresentadas as propostas de módulos de serviço para a implementação do mecanismo de balanceamento de carga.

4.2 O gerenciador de aplicações

Uma vez apresentada a estrutura geral dos módulos de serviço da plataforma JOIN, será descrito o serviço gerenciador de aplicações que procura implementar nesta plataforma o mecanismo de distribuição de carga discutido no Cap. 3.

4.2.1 Adaptação do modelo de aplicação

Nesta seção, serão feitas algumas considerações a respeito da adaptação do modelo de aplicação proposto no Cap. 1 para o serviço de gerenciamento de aplicações desenvolvido.

A plataforma JOIN foi completamente desenvolvida utilizando-se a linguagem Java, principalmente em razão de sua portabilidade. Sendo assim, todas as tarefas das aplicações JOIN devem conter uma interface em Java que permita à plataforma utilizá-las convenientemente na execução das aplicações. O conteúdo desta interface, denominada *AppCode()*, é reproduzido a seguir.

```
package join.app;

import java.io.*;

public interface AppCode {
    public Serializable taskrun(Serializable par);
}
```

Cabe aqui ressaltar que as tarefas de aplicações não precisam ser inteiramente desenvolvidas na linguagem Java uma vez que é possível realizar chamadas entre classes Java e programas escritos em código nativo. Entretanto, nestes casos seria necessária a criação de interfaces Java que estabelecessem a ligação entre estas tarefas e a plataforma JOIN.

A interface descrita acima define um único método denominado *taskrun()* que tem como função abrigar o código a ser executado pela tarefa. Desta forma, sempre que uma classe Java for desenvolvida para ser utilizada como tarefa de aplicação na plataforma JOIN, terá de implementar a interface *AppCode* e incluir em seu método *taskrun()* o código necessário à sua execução.

Nota-se ainda que este método utiliza objetos serializáveis como parâmetros de entrada e de saída. Esta é a maneira como o conceito de *blocos de dados* (definição 1) foi agregado à estrutura das tarefas utilizando-se os recursos da linguagem Java.

Para a representação das especificações de aplicações, foi desenvolvida uma estrutura de dados que tem como principal característica poder ser facilmente construída a partir da descrição textual da especificação em linguagem PASL.

Essa estrutura é definida por uma classe Java denominada PAS (especificação de aplicação paralela, ou *Parallel Application Specification*). A geração de objetos PAS que representem aplicações paralelas pode ser feita de maneira totalmente independente das tarefas de aplicação, que podem portanto ser utilizadas livremente como blocos de computação a serem combinados para se promover algum processamento específico.

Conhecendo-se a estrutura das tarefas e das especificações de aplicação a serem utilizadas pelo gerenciador de aplicações, segue-se com a descrição dos diversos componentes deste serviço.

4.2.2 Descrição dos sub-módulos do serviço

Nesta seção serão descritos os módulos de serviço desenvolvidos para a implementação, na plataforma JOIN, da proposta de balanceamento de carga apresentada no Cap. 3. Entretanto, cabe aqui esclarecer que, apesar da especificação destes módulos de serviço estar completa, não foi possível realizar testes com a etapa de balanceamento de carga inter-grupos (particionamento PPCP) da proposta. Esta impossibilidade se deu em razão da plataforma JOIN ainda não disponibilizar módulos de serviço necessários ao suporte de múltiplos grupos. Maiores informações acerca do estado atual de desenvolvimento desta plataforma podem ser encontradas no Apêndice A.

Com relação aos módulos de serviço, sabe-se que, conforme descrito na seção 4.1, cada serviço deve conter um sub-módulo servidor, um coordenador, um trabalhador e um jack. Assim, nas seções a seguir serão descritos cada um dos sub-módulos desenvolvidos para este serviço de gerenciamento de aplicações.

Sub-módulo jack

É através deste sub-módulo que todas as ações administrativas relacionadas às aplicações são realizadas. Estas ações são:

1. instalar tarefas e especificações de aplicações: para que uma aplicação seja distribuída e executada na plataforma, suas tarefas e sua especificação devem estar armazenadas na máquina servidora do JOIN. Armazenar esses objetos no servidor para que possam ser posteriormente utilizados corresponde à

"instalação" da aplicação e essa instalação é feita a partir de um componente jack utilizando-se a interface apropriada do gerenciador de aplicações. Cabe aqui ressaltar que a geração dos objetos PAS se dá no próprio componente jack, antes que estes sejam enviados ao servidor. Para isso, analisadores léxico e sintático encarregam-se de ler a descrição textual, em liguagem PASL, da especificação da aplicação e de construir o objeto PAS correspondente.

2. Desinstalar tarefas e especificações de aplicações: analogamente à instalação destes objetos, o sub-módulo jack do gerenciador de aplicações permite que tarefas e especificações de aplicações previamente instaladas no servidor sejam removidas.

3. Iniciar e interromper a execução de uma aplicação: se uma determinada aplicação estiver devidamente instalada no servidor do JOIN, sua execução poderá ser requisitada. Durante a execução desta aplicação é possível solicitar sua interrupção temporária ou definitiva.

4. Monitorar o andamento de uma execução: neste sub-módulo é ainda possível monitorar o andamento da execução de uma aplicação. Esse monitoramento permite saber o ponto em que a execução se encontra bem como os resultados parciais produzidos.

Como pode ser visto, o sub-módulo jack do gerenciador de aplicações tem como funções básicas a administração de aplicações na plataforma. Uma vez instalada uma aplicação, caberá aos demais sub-módulos deste serviço levar essa execução a cabo, como será visto nas seções a seguir.

Na Fig. 4.2 é apresentada uma transcrição comentada da interface Java deste sub-módulo.

Sub-módulo servidor

O sub-módulo servidor do serviço gerenciador de aplicações tem como atribuição atender às requisições feitas pelo jack para instalação/desinstalação/execução/interrupção de aplicações e interagir com os coordenadores de modo a promover a distribuição e execução das aplicações na plataforma.

Para a instalação e desinstalação de objetos, o sub-módulo servidor deve apenas interagir com o sistema de arquivos local.

Já para a execução de aplicações, este sub-módulo deve consultar o sub-módulo servidor do serviço escalonador a fim de que este último lhe indique um particionamento/escalonamento da aplicação entre os grupos de trabalhadores. Conforme visto na Seção 3.2.1, este particionamento é realizado segundo o modelo PPCP e, ao contrário do que ocorre com a distribuição das tarefas dentro de um mesmo grupo, todas as partes da aplicação deverão ser submetidas a um único coordenador, a partir do qual será feita a difusão das demais partes aos outros coordenadores; feito isso, caberá ao servidor apenas aguardar que a aplicação seja finalizada. Se durante este período for solicitada a interrupção da execução da aplicação, uma mensagem deverá ser enviada a este coordenador, que se encarregará da interrupção da mesma.

```
1  package applicationManager.jack;
2
3  import java.util.*;
4
5  import join.app.*;
6
7  public interface J_ApplicationManager {
8      public PAS getPAS(String appName);     // Estes três métodos são utilizados pela
9      public Map getLocalApps();             // GUI através da qual o gerenciamento de
10     public Map getRemoteApps();            // aplicações é feito na plataforma
11     public void addComponents(String appName, PAS aPAS, Collection componentsToAdd);
12     public void removeComponents(String appName, Collection componentsToRemove);
13                                            // Os dois métodos anteriores são
14                                            // utilizados para o gerenciamento de
15                                            // componentes de aplicações
16     public String uninstallApp(String appName);// Desinstala aplicações
17     public String synchronize(String appName); // Atualiza a versão de aplicações
18     public String submitApp(String appName);   // Inicia a execução de uma aplicação
19     public Collection getAppComponents(String appName);
20                                            // Recebe referências aos componentes de
21                                            // uma aplicação
22 }
```

Figura 4.2: Inteface do sub-módulo jack do serviço Gerenciador de Aplicações

Nota-se aqui que apesar do servidor submeter a aplicação a apenas um de seus coordenadores, o trabalho de execução da aplicação poderá ser realizado em outros grupos. Essa distribuição de trabalho se fará no nível dos coordenadores, como será visto na próxima seção.

Na Fig. 4.3 é apresentada a interface Java comentada deste sub-módulo.

Sub-módulo coordenador

Conforme mencionado anteriormente, sempre que o componente jack solicitar a execução de uma aplicação ao sub-módulo servidor, esta será submetida a um dos coordenadores indicados pelo escalonamento realizado no sub-módulo servidor do serviço escalonador.

O coordenador responsável pelo recebimento da aplicação diretamente do servidor é também o coordenador do grupo escolhido para executar as tarefas de sincronização do particionamento PPCP. Assim, caberá a este coordenador não só administrar a execução das tarefas de sua partição como também coordenar a execução das demais partes nos outros coordenadores. Ele deverá enviar aos demais coordenadores suas respectivas partes e aguardar suas respostas para efetuar a ligação dos dados recebidos de diferentes coordenadores.

```
1  package applicationManager.server;
2
3  import applicationManager.common.*;
4
5  public interface S_ApplicationManager {
6      private void initializeInstalledApps();// Método interno para inicialização de
7                                             // base de dados de aplicações instaladas
8      private void saveInstalledApps();      // Método interno para salvar arquivo de
9                                             // aplicações instaladas
10 }
```

Figura 4.3: Inteface do sub-módulo servidor do serviço Gerenciador de Aplicações

A ordem de execução de parte de uma aplicação é composta por uma especificação de aplicação (que na verdade é parte da especificação completa da aplicação) e pelas classes Java que definem as tarefas de aplicação referenciadas por essa especificação. Uma vez recebidos esses dados, o coordenador deverá consultar novamente o serviço escalonador para que ele indique uma distribuição das tarefas entre os trabalhadores do grupo segundo algum algoritmo de alocação estática de tarefas.

Depois de definida e realizada a distribuição das tarefas prontas entre os trabalhadores, o coordenador deverá aguardar a conclusão das mesmas de acordo com o algoritmo de escalonamento GSTR (Seção 3.1.2). Quando todo o trabalho for completado, os resultados serão retornados à entidade que lhe submeteu a requisição de execução, seja ela um sub-módulo servidor ou um coordenador.

Este sub-módulo não exporta nenhuma interface em particular, já que recebe requisições apenas dos sub-módulos servidor e trabalhador do serviço gerenciador de aplicações. Conforme visto anteriormente, sub-módulos de um mesmo serviço comunicam-se por meio de trocas de mensagens entre componentes da plataforma.

Sub-módulo trabalhador

Finalmente, é neste sub-módulo que o processamento propriamente dito das tarefas é realizado. Para isso, são enviadas para cada sub-módulo as classes Java correspondentes às tarefas que devem ser executadas, bem como os blocos de dados sobre os quais o processamento se realizará.

Assim que cada uma das tarefas enviadas ao trabalhador é concluída, seus resultados são atualizados no servidor de arquivos do JOIN ou diretamente enviados ao coordenador do grupo, que se encarregará de dar o destino correto ao mesmo, fechando assim a ciclo de execução da aplicação.

O mecanismo empregado por este sub-módulo para a execução de tarefas permite que tarefas originais e replicadas sejam executadas concorrentemente mas com prioridades diferentes. Um estudo mais aprofundado seria necessário para se avaliar o impacto de diferentes prioridades neste mecanismo de execução. Entretanto, este estudo é reservado para uma extensão futura deste trabalho e, por hora, arbitra-se que as

tarefas originais são executadas com o dobro da prioridade das tarefas replicadas.

Na Fig. 4.4, é apresentada a interface Java comentada deste sub-módulo.

```
1  package applicationManager.worker;
2
3  import applicationManager.common.*;
4
5  public interface W_ApplicationManager implements DataGatherer {
6
7      void startTask(String fullName, String extension, long tid,
8                              Serializable par, Boolean isReplica, double key);
9                                              // Este é o método invocado sempre que uma
10                                             // nova tarefa é enviada ao trabalhador.
11                                             // Sua função é colocár a tarefa na fila
12                                             // de execução
13     void killTask(long tid);
14                                             // Método invocado quando a execução de
15                                             // uma tarefa deve ser interrompida
16     void putNewData(AppTask task);
17                                             // Este método é utilizado pelos executores
18                                             // de tarefas quando a execução de uma
19                                             // tarefa é concluída
20 }
```

Figura 4.4: Inteface do sub-módulo trabalhador do serviço Gerenciador de Aplicações

4.3 O escalonador

4.3.1 Descrição dos sub-módulos do serviço

Analogamente ao realizado na descrição do gerenciador de serviços, nas seções a seguir serão descritos cada um dos sub-módulos desenvolvidos para o serviço escalonador.

Sub-módulo jack

O serviço escalonador tem como particularidade o fato de receber requisições somente do gerenciador de aplicações. Sendo assim, seu conteúdo poderia ser agregado ao do gerenciador de aplicações mas, por razões de modularidade, este foi mantido como um serviço independente.

Nestas circuntâncias, uma das consequências em se manter o escalonador como um serviço independente é o fato deste oferecer ao usuário do sistema poucos parâmetros configuráveis. De fato, as únicas

opções de configuração se fazem sobre parâmetros que balizam a maneira como as aplicações são particionadas e como as tarefas são distribuídas dentro dos grupos do JOIN.

Atualmente há a possibilidade de se escolher o esquema de escalonamento a ser utilizado nos grupos dentre escalonador estático, GS ou GSTR, combinados com os algoritmos de escalonamento trivial e *best-fit*, perfazendo um total de seis possíveis combinações. Essas seis possibilidades são as exploradas nos testes apresentados no Cap. 6.

Na Fig. 4.5 é apresentada a interface Java comentada deste sub-módulo.

```
1  package scheduler.jack;
2
3  import scheduler.common.*;
4
5  public interface J_Scheduler {
6
7      public setSchedulerType(SchedulerType st);
8                                              // seleciosa tipo de escalonamento
9      public setSchedulerAlgorithm(SchedulerAlgorithm sa);
10                                             // seleciona algoritmo de escalonamento
11 }
```

Figura 4.5: Inteface do sub-módulo jack do serviço Escalonador

Sub-módulo servidor

De acordo com a proposta de balanceamento de carga entre grupos (seção 3.2.1), para se distribuir o trabalho computacional de uma aplicação entre diferentes sistemas heterogêneos, basta aplicar um particionamento à aplicação e então distribuir as partes entre os grupos. É basicamente esta a atribuição do sub-módulo servidor do serviço escalonador.

Quando do gerenciador de aplicações é solicitada a execução de uma aplicação, este submete-a ao serviço escalonador para que seja particionada e escalonada entre os grupos de trabalhadores. Para realizar o particionamento da aplicação e a atribuição das partes aos grupos de trabalhadores, o serviço escalonador vale-se do algoritmo apresentadas em 3.2.1.

Uma vez realizada esta etapa, todos os resultados são remetidos ao coordenador para o qual foi atribuída a primeira parte da aplicação, conforme descrito anteriormente. Este mecanismo segue recursivamente até que todas as partes sejam entregues a seus respectivos coordenadores.

Concluída a execução da aplicação, caberá ao sub-módulo servidor do gerenciador de aplicações informar ao serviço escalonador este evento. Com isso, o escalonador poderá atualizar suas tabelas internas de carga nos grupos de trabalhadores de modo a poder escalonar futuras aplicações eficientemente.

Na Fig. 4.6 é apresentada a interface Java comentada deste sub-módulo.

```
1  package scheduler.server;
2
3  import join.task;
4
5  public interface S_Scheduler {
6      public PAS[] applyPPCP(PAS pas);       // Particiona aplicação segundo o PPCP
7  }
```

Figura 4.6: Inteface do sub-módulo servidor do serviço Escalonador

Sub-módulo coordenador

Ao sub-módulo coordenador do serviço escalonador cabe a tarefa de atender requisições de escalonamento de tarefas de aplicações no próprio grupo de trabalhadores do qual faz parte. Assim, é neste sub-módulo que os algoritmos para a resolução da etapa estática do balanceamento GSTR são implementados.

Para a realização do escalonamento, o escalonador se vale das características das tarefas de aplicação a ele entregues e das estimativas dos poderes computacionais das máquinas trabalhadoras do grupo. Essas estimativas são enviadas pelos sub-módulos trabalhadores do serviço escalonador.

Na Fig. 4.7 é apresentada a interface Java comentada deste sub-módulo.

```
1  package scheduler.coordinator;
2
3  import join.task;
4  import join.service.*;
5
6  public class C_Scheduler implements ServiceCode, C_Scheduler {
7
8      public Long allocateExecutionRequest(JAS.ModuleSpec req);
9                                             // É através desta função que o serviço
10                                            // gerenciador de aplicações submete
11                                            // requisições de execução de tarefas ao
12                                            // escalonador. Como resultado, recebe o
13                                            // identificador do trabalhador para onde
14                                            // a tarefa deverá ser submetida
15     public void localInit();               // Estes quatro métodos são os métodos da
16     public void globalInit();              // interface ServiceCode. É neles que é
17     public void serviceRun();              // implementado o código de inicialização,
18     public void serviceFinally();          // finalização e operação do serviço
19 }
```

Figura 4.7: Inteface do sub-módulo servidor do serviço Escalonador

Sub-módulo trabalhador

Neste sub-módulo, são realizadas operações que visam estimar o poder computacional da máquina trabalhadora em que são executadas. Para isso, um conjunto de *benchmarks* é periodicamente executado e os tempos de execução resultantes são remetidos ao sub-módulo coordenador para que possam ser utilizados no escalonamento das tarefas de aplicação.

O conjunto de *benchmarks* selecionado procura avaliar o desempenho das máquinas trabalhadoras na execução de operações aritméticas, de atribuição, de *loop* e matemáticas, sendo que nesta última categoria são avaliadas operações trigonométricas, exponenciais, logarítmicas e não lineares.

Este sub-módulo não exporta nenhuma interface em particular, já que os resultados de todas as suas operações são destinados ao sub-módulo coordenador do serviço escalonador, com o qual se comunica por meio de serviço comunicador.

4.4 Conclusões

Neste capítulo foi descrita a estrutura interna dos módulos escalonador e gerenciador de aplicações, especialmente desenvolvidos para a implementação no JOIN do mecanismo de balanceamento de carga desenvolvido no Cap. 3. Também foi descrita a maneira como o modelo de aplicação (Seção 1.2.1) foi adaptado à estrutura da plataforma.

No próximo capítulo, serão apresentadas as baterias de simulações realizados com a plataforma JOIN, que procuraram validar a escolha do esquema de balanceamento de carga proposto para um MPVC.

Capítulo 5

Simulações

Neste capítulo, são apresentadas algumas simulações realizadas com o objetivo de se verificar as vantagens e desvantagens das proposta de escalonamento intra-grupo e inter-grupos introduzidas no Cap. 3 frente aos mecanismos comumente empregados em sistemas distribuídos.

5.1 Simulações de escalonamentos intra-grupo

5.1.1 Características das simulações

As simulações desenvolvidas nesta seção procuram apresentar as vantagens e desvantagens do algoritmo de balanceamento GSTR frente a dois esquemas de balanceamento de carga estáticos e dois outros algoritmos de balanceamento de carga dinâmicos.

Nestas simulações, é feita a simplificação de que o tamanho dos blocos de dados utilizados por todas as tarefas é constante. Com isso, espera-se poder avaliar de maneira mais controlada o impacto da relação computação / comunicação sobre o desempenho dos esquemas de balanceamento de carga simulados.

Para a realização destas simulações, foi desenvolvido um modelo matemático que procurou representar fielmente as características do sistema real correspondente. Este modelo, construído a partir de um conjunto de funções **MATLAB**, descreve não só o estado interno das entidades modeladas, tais como as unidades de processamento do grupo, como também as eventuais indeterminações inerentes aos ambientes de computação maciçamente paralela, tais como falhas nas UPs e variações em suas cargas.

As funções desenvolvidas foram:

- **generateApp**: esta função é responsável pela geração aleatória de especificações de aplicações paralelas que sigam o modelo apresentado pela Def. 5 (Seção 1.2.1). Para isso, esta rotina deve receber como parâmetros de entrada o número de lotes de tarefas e o número total de tarefas que a aplicação deverá conter. Nota-se ainda que nem sempre é possível gerar uma aplicação aleatória que contenha exatamente o número de tarefas solicitado, especialmente em virtude da dependência entre as cardinalidades dos lotes de tarefas imposta pela relação multiplicativa introduzida pela Def. 4. Neste caso,

o número final de tarefas geradas apresenta apenas uma pequena divergência em relação ao número inicial solicitado. A saída desta função é uma matriz de adjacências representando as relações de dependência entre os lotes e as multiplicidades destes lotes.

- **generateGroup**: esta função é utilizada para se gerar a especificação de um grupo de computadores. Para isso, devem ser informados o número de unidades de processamento trabalhadoras, o número de tarefas que serão executadas nessas UPs, informações relativas aos tempos de procesamento e comunicação e a maneira como estes tempos podem variar de uma UP a outra. As saídas desta função são uma matriz contendo os tempos de execução de cada uma das tarefas hipotéticas em cada uma das UPs e uma matriz contendo os tempos de comunicação entre as UPs para a transmissão de um bloco de dados que, conforme explicitado anteriormente, será considerado como de tamanho fixo.

- **simpleSchedule**: esta função é a responsável pelo cálculo do escalonamento trivial de uma aplicação paralela (a partir de sua especificação) em um grupo de UPs. O escalonamento é dito trivial porque procura atribuir o mesmo número de tarefas a todas as UPs, independentemente dos tempos de execução de cada tarefa em cada UP. Os parâmetros de entrada desta função são as especificações do grupo de UPs e da aplicação a ser escalonada e sua saída é uma sequência de ordens de envio de tarefas a UPs.

- **bestfitSchedule**: esta função implementa o cálculo do escalonamento *best-fit* de uma aplicação paralela em um grupo de UPs. Este algoritmo necessita dos mesmos parâmetros do algoritmo anterior e também retorna uma lista de ordens de envio de tarefas, mas procura distribuir estas tarefas proporcionalmente aos poderes computacionais das UPs.

- **staticScheduler**: esta função é a responsável pela simulação propriamente dita do escalonamento estático. Para isso, ela promove o escalonamento da aplicação no grupo de UPs passados como parâmetros e calcula o tempo total de execução desta aplicação segundo o escalonamento proposto. Os escalonamentos possíveis de serem executados são o trivial e o *best-fit*, devidamente implementados pelas duas funções descritas acima. Pode ainda ser especificado um nível de erro a ser introduzido na especificação do grupo de UPs para que a simulação do escalonamento se faça em condições mais próximas da realidade.

- **gsScheduler**: analogamente à função anterior, esta implementa a simulação de um escalonador geracional. Os escalonamentos (feitos em múltiplas etapas) podem também utilizar os dois algoritmos propostos acima (trivial e *best-fit*) e também há a possibilidade de introdução de erros na especificação do grupos de UPs.

- **gstrScheduler**: esta função implementa um simulador para o escalonamento geracional com replicação de tarefas. Além da especificação de um nível de erro a ser introduzido nos tempos de execução e comunicação do grupo de UPs, é ainda possível especificar um nível de falhas em

UPs. Isso permite a avaliação do impacto de falhas em unidades de processamento no tempo final de execução da aplicação.

Maiores detalhes a respeito de cada uma destas funções podem ser encontradas no Apêndice B, incluindo o código-fonte das mesmas.

Com o objetivo de estabelecer uma base comparativa para as simulações do sistema real, é primeiramente realizado um conjunto de simulações sobre um sistema homogêneo hipotético. Desta forma, pode-se comparar posteriormente o impacto da heterogeneidade do grupo de UPs sobre o desempenho de cada um dos escalonadores.

5.1.2 Simulações sobre grupos homogêneos

Na primeira bateria de simulações, foram geradas diversas instâncias de aplicações e grupos homogêneos. Para isso, foram aninhados quatro laços repetitivos, cada qual responsável pela iteração de um parâmetro utilizado na geração das aplicações e dos grupos. Na Fig. 5.1 é apresentado o algoritmo utilizado para a geração dos cenários de simulação.

```
1  para N (UPs) variando de 10 a 50 em intervalos de 10,
2      para K (tempo) variando de 1 s a 7 s em intervalos de 2,
3          para B (lotes) variando de 5 a 11 em intervalos de 2,
4              gerar um grupo homogêneo que contenha N UPs, cada uma delas
5              capaz de executar cada uma das tarefas da aplicação em
6              (50 +/- 5)*K segundos e com tempo de comunicação
7              fixo e igual a 10 segundos (*).
8              para T (tarefas) variando de 100 a 1000 em intervalos de 100,
9                  gerar uma aplicação que contenha T tarefas e B lotes.
10                 simular escalonamento da aplicação gerada no grupo gerado
11                 utilizando-se escalonadores diversos.
12             fim,
13         fim,
14     fim,
15 fim.
16
17 (*) Nota-se aqui que o tempo de execução de uma determinada tarefa será
18     o mesmo em todas as UPs e que a distribuição dos tempos de execução
19     das diferentes tarefas é uniforme.
```

Figura 5.1: Algoritmo de geração dos cenários de simulação para grupos homogêneos

Com o emprego deste esquema de simulação, pode-se analisar a influência de cada um dos parâmetros utilizados sobre os tempos finais de escalonamento em uma ampla faixa de valores.

As faixas de valores arbitrados procuraram, em certo grau, cobrir possibilidades passíveis de serem encontradas na prática. Assim, espera-se inicialmente que um grupo de UPs tenha aproximadamente 30 unidades, que este grupo execute aplicações com aproximadamente 5 lotes e centenas de tarefas, cada qual com um tempo de execução da ordem de centenas de segundos, e que a relação entre os tempos de processamento e comunicação para cada tarefa seja da ordem de dezenas.

Como pode ser verificado multiplicando-se o número de passos dos diversos laços repetitivos, o número de combinações de parâmetros é igual a 800, o que significa que 800 cenários distintos serão gerados e simulados. O gráfico da Fig. 5.2 mostra os tempos de execução obtidos para essa bateria de simulações.

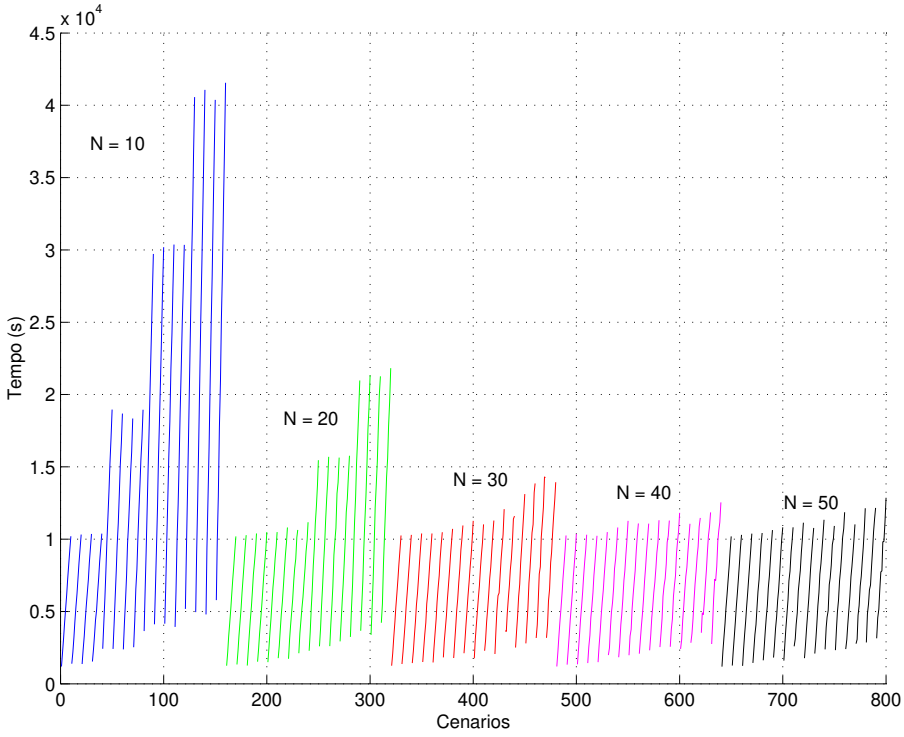

Figura 5.2: Tempos de execução dos 800 cenários gerados com sistemas homogêneos

Como não foram introduzidos erros durante as simulações, os tempos finais de execução gerados pelos escalonamentos estáticos e dinâmicos foi o mesmo. Além disso, em um grupo homogêneo como os aqui simulados, a aplicação dos algoritmos *best-fit* e trivial resultam no mesmo escalonamento, já que cada tarefa demanda exatamente o mesmo tempo de execução em todas as UPs.

O conjunto de curvas resultante é nitidamente formado por cinco setores distintos; cada um destes setores corresponde às simulações feitas com grupos de 10, 20, 30, 40 e 50 unidades de processamento, respectivamente. Fica claro desde já a nítida influência do tamanho do grupo sobre o tempo final de execução das aplicações, como era de se esperar.

Tomando-se o terceiro destes cinco setores para uma análise mais detalhada (Fig. 5.3), podem-se

observar dezenas de segmentos ascendentes. Estes segmentos representam cada uma das execuções do laço mais interno do algoritmo da Fig. 5.1, responsável pela iteração sobre o número de tarefas das aplicações geradas. Pode-se postular pelo padrão de crescimento dos segmentos que a relação entre o número de tarefas e o tempo final de execução da aplicação é claramente linear, pelo menos dentro da faixa de valores simulados.

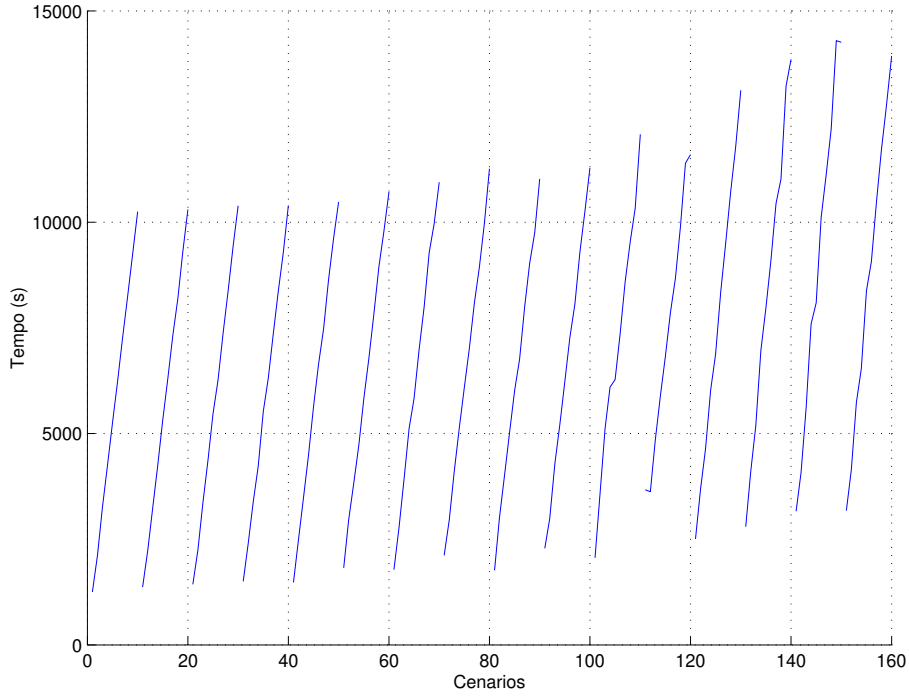

Figura 5.3: Tempos de execução dos 160 cenários “centrais”, onde N = 30.

Para que uma análise mais criteriosa possa ser feita, convém isolar a influência de cada um dos parâmetros de simulação sobre os tempos de execução obtidos. Pelo apresentado até o momento, verificou-se que o aumento no número de UPs tende a diminuir o tempo de execução médio das aplicações, o que é razoável. Verificou-se também que o aumento na quantidade de computação realizada por uma aplicação, representado pelo aumento no número de tarefas, aumenta linearmente o tempo total de execução desta aplicação. Resta agora verificar a influência do número de lotes de tarefas e do tempo de execução das tarefas sobre os tempos totais obtidos.

Desta forma, na Fig. 5.4 são apresentados quatro gráficos. Em cada um deles, as retas representam os tempos de execução obtidos através do aumento do números de tarefas das aplicações simuladas. As múltiplas retas de um mesmo gráfico representam aplicações com diferentes tamanhos de tarefas, com o parâmetro K variando de 1 a 7 em intervalos de 2. Finalmente, cada um dos gráficos apresenta os resultados obtidos para um B diferente, que é o número de lotes de tarefas, variando de 5 a 11 em intervalos de 2. Para

estes casos foi fixado o número de UPs (parâmetro N) em 30.

Pode-se perceber que a influência do número de lotes da aplicação sobre os tempos de execução é relativamente pequena e irregular; de fato, o que pode influenciar o tempo final de execução de uma aplicação é a maneira como as relações de dependência de dados entre lotes se estabelecem; quanto mais paralelamente os lotes puderem ser executados, menor será o número de vezes em que o escalonador terá de aguardar a execução total de um lote para prosseguir na execução de outro(s). Desta forma, o aumento no número de lotes aumenta também o número de combinações entre lotes, o que aumenta essa irregularidade sobre os tempos de execução. Entretanto, a influência dessa irregularidade não é muito significativa, sobretudo quando comparada com o tempo total de execução da aplicação.

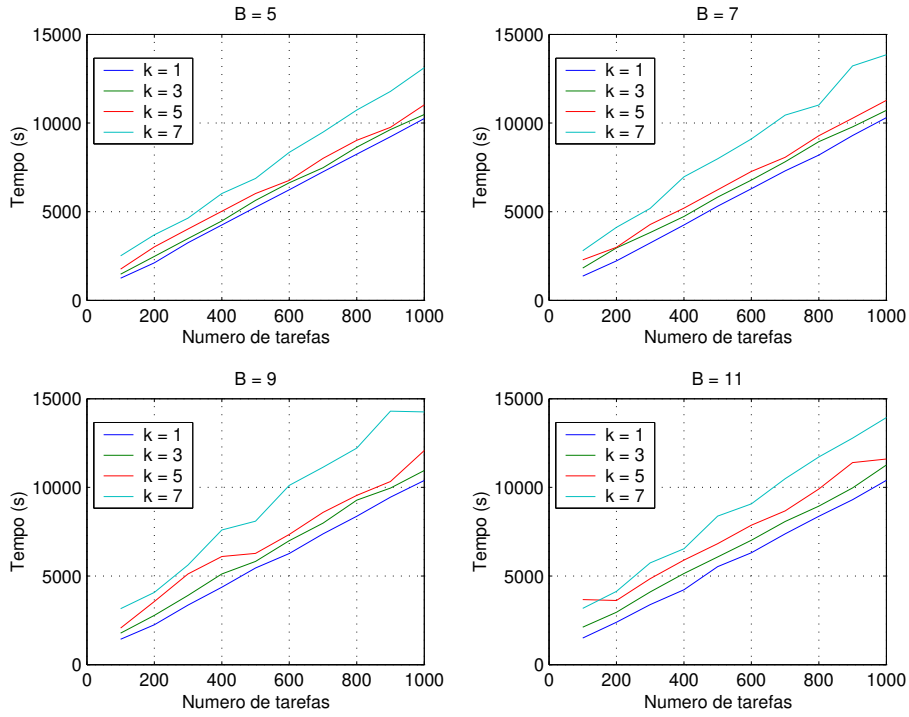

Figura 5.4: Análise dos tempos de execução fixando-se N e B, e variando-se T e K.

Uma análise similar à realizada sobre o parâmetro B pode também ser realizada sobre o parâmetro K. Na Fig. 5.5 são apresentados quatro gráficos onde foram fixados os valores de K (em cada gráfico). Por estes gráficos, pode-se perceber que o comportamento geral das curvas mantém-se uniforme com a variação de K exceto por um incremento homogêneo nos tempos de execução. Este incremento se deve ao fato do parâmetro K determinar o tempo de execução das tarefas e, conseqüentemente, o tempo total de execução de uma aplicação. Observa-se também que, com o aumento de K, as retas tornam-se mais irregulares. Conforme visto no gráfico anterior, esta irregularidade se deve principalmente ao número de lotes (parâmetro B). Com o aumento do tempo de execução das tarefas, o tempo de execução dos lotes

cresce proporcionalmente, aumentando também os tempos em que a execução da aplicação fica bloqueada a espera de resultados parciais.

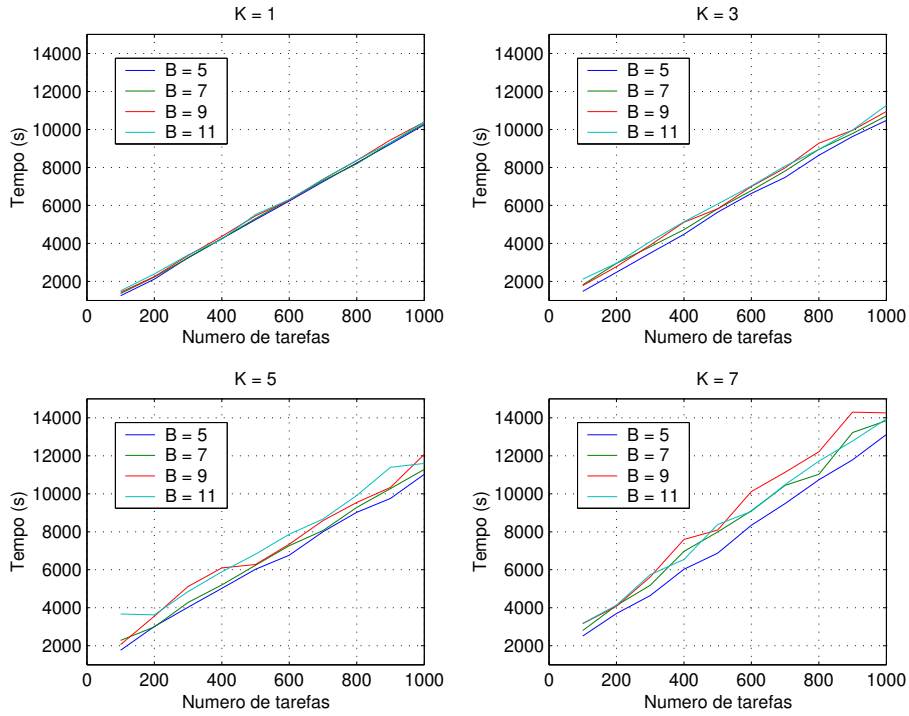

Figura 5.5: Análise dos tempos de execução fixando-se N e K, e variando-se T e B.

Uma análise adicional que pode ser feita sobre os dados obtidos é a de *speed-up*. Para se calcular o *speed-up* em um grupo homogêneo basta dividir o tempo de execução sequencial numa das UPs do sistema pelo tempo de execução de uma versão paralela da mesma aplicação. Na Fig. 5.6 é apresentado este cálculo para os 800 cenários simulados e, em destaque, nos 160 cenários centrais (onde N = 30).

Como pode ser visto pelos gráficos, o valor do *speed-up* muda significativamente ao longo das 800 simulações realizadas. Para que seja possível uma análise mais detalhada destas mudanças, será adotado o mesmo procedimento utilizado anteriormente para o isolamento dos parâmetros B e K. O parâmetro N será fixado em 30.

Na Fig. 5.7, são apresentados os gráficos obtidos através do isolamento do parâmetro B, que define o número de lotes de tarefas da aplicação. Analogamente ao verificado anteriormente, o aumento no número de lotes de tarefas da aplicação provoca um aumento na irregularidade dos resultados obtidos pelas simulações, além de acarretar uma ligeira diminuição nas taxas de *speed-up*. Um outro aspecto revelado pelos gráficos é que o aumento no número de tarefas de uma aplicação provoca um aumento em sua taxa de *speed-up*. Este aumento, entretanto, não cresce de maneira linear, saturando-se ao final das curvas.

Como as aplicações simuladas possuem sempre a mesma relação computação/comunicação para um

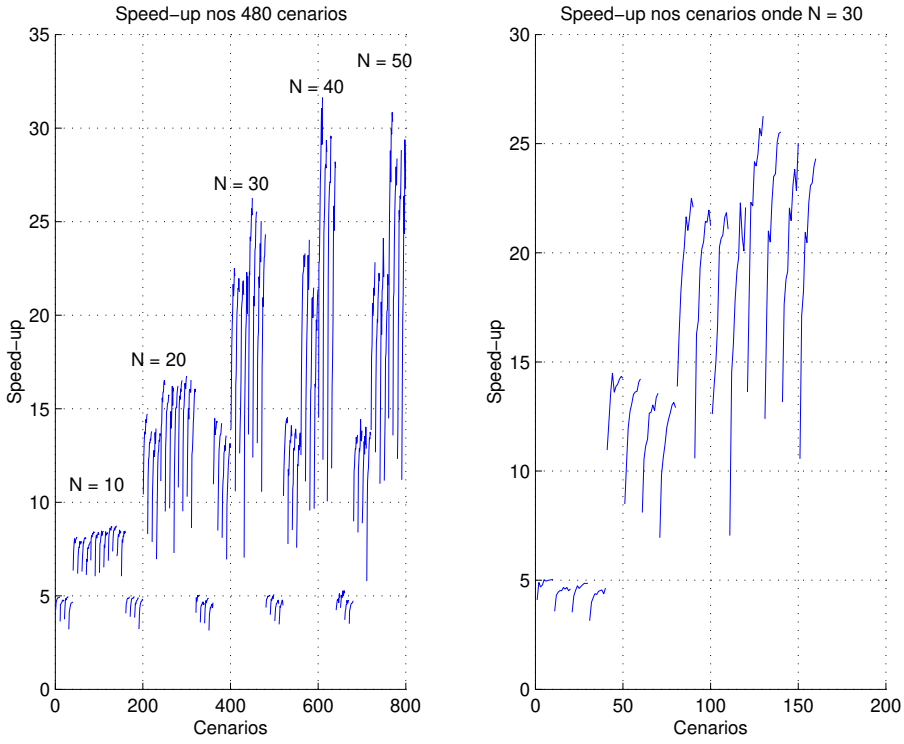

Figura 5.6: Gráficos de speed-up para os 800 cenários simulados (a esquerda) e dos 160 cenários centrais, onde N = 30 (a direita).

determinado valor de K e independentemente do número de tarefas T, seria de se esperar que o *speed-up* fosse constante para K fixo. Na verdade, o que se observa nos gráficos da Fig. 5.7, é que o *speed-up* é relativamente baixo para os valores iniciais de T e tende a uma constante à medida que o parâmetro T aumenta. Isto se deve ao fato dos lotes de tarefas de maior cardinalidade serem executados mais eficientemente que os lotes de menor cardinalidade. Esta diferença de desempenho se deve ao tempo gasto para se preencher o grupo de UPs com as tarefas do lote; quando o lote é mais numeroso, o tempo gasto para a distribuição inicial das tarefas é menos impactante no tempo total de execução do lote.

Também analogamente ao realizado anteriormente, na Fig. 5.8 são apresentados os gráficos onde são fixados os valores de K (atenção à mudança de escala). Percebe-se novamente que o comportamento das curvas é aproximadamente o mesmo em todos os casos com a diferença de que para cada valor de K há um *speed-up* médio diferente. Estas diferenças se devem ao fato de que o aumento de K provoca um aumento na relação computação/comunicação das aplicações uma vez que o tempo de comunicação das tarefas mantém-se constante.

Como pode ser observado pelos gráficos, o *speed-up* é tão melhor (mais alto) quanto:

1. maior a relação computação/comunicação de uma aplicação,

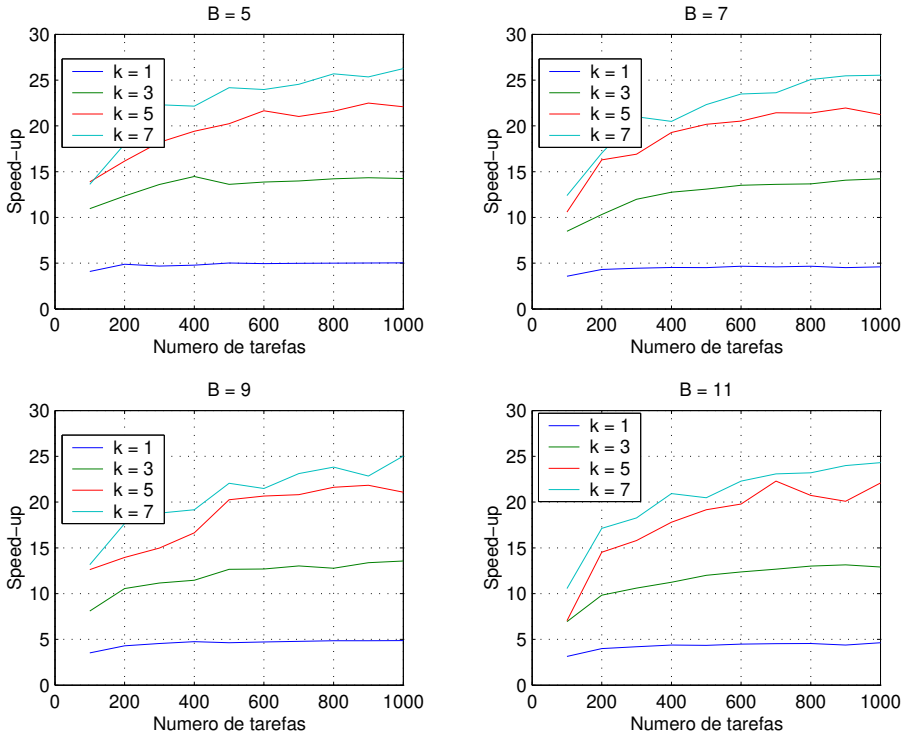

Figura 5.7: Análise das taxas de speed-up fixando-se N e B, e variando-se T e K.

2. maior o número de unidades de processamento do grupo (antes da faixa de saturação que, neste caso, esteve em torno de 40).

Por outro lado, o *speed-up* satura-se quanto:

1. menor a cardinalidade dos lotes de tarefas de uma aplicação,
2. menos paralelamente os lotes de tarefas de uma aplicação podem ser executados,
3. maior o número de unidades de processamento do grupo (depois da faixa de saturação que, neste caso, esteve em torno de 40).

Depois de analisadas as propriedades observadas nos tempos resultantes das simulações feitas sobre grupos de computadores onde não havia falhas, cabe agora realizar simulações que procurem ilustrar o impacto de falhas nas UPs sobre o tempo final de execução da aplicação. Nestes novos cenários será simulado somente o algoritmo GSTR, que é o único tolerante a falhas nas UPs.

Nesta bateria de simulações, o número de UPs do grupo foi fixado em 40 e o número de lotes de tarefas foi fixado em 5. O tempo médio de execução de uma tarefa em uma UP foi arbitrado em 500 s e o tempo de comunicação entre UPs para a troca dos blocos de dados manteve-se fixo em 10 s. Foram variados apenas

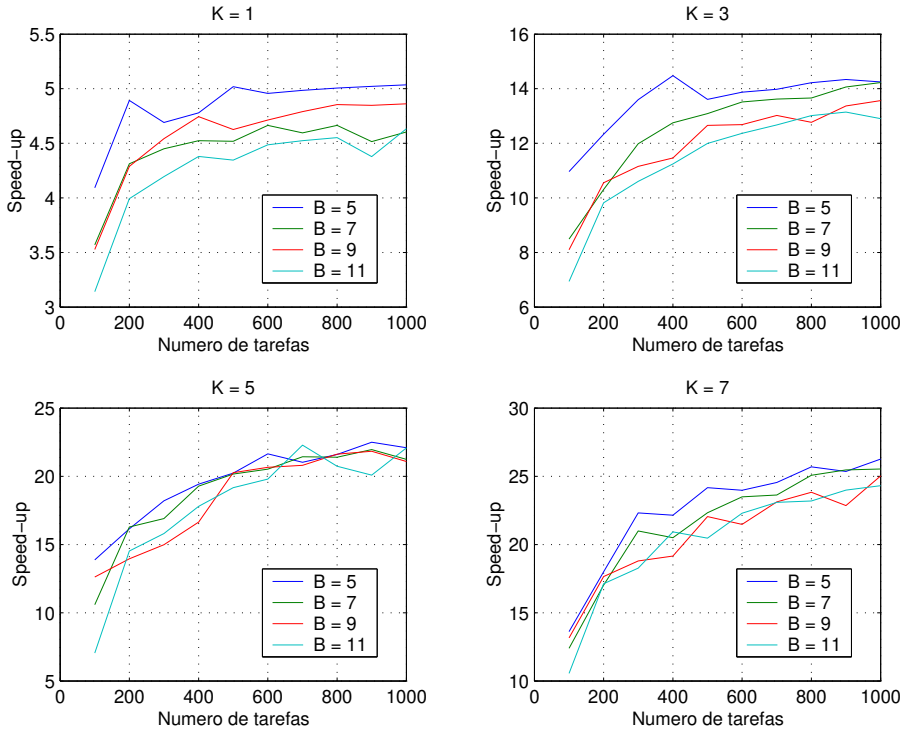

Figura 5.8: Análise das taxas de speed-up fixando-se N e K, e variando-se T e B.

o número de tarefas da aplicação, que cresceu de 100 a 1000 em intervalos de 100, e o número de UPs defeituosas, ou seja, que falhavam no início do processo de escalonamento das tarefas da aplicação. Os resultados podem ser vistos no gráfico da Fig. 5.9.

Como pode ser observado, o aumento linear no número de UPs falhas provoca um aumento não linear no tempo total de execução da aplicação (distâncias entre as curvas são diferentes). Este comportamento pode ser entendido considerando-se que, nestes casos (em que o número de tarefas é bem maior que o númeo de UPs), o tempo de execução paralela da aplicação deverá ser aproximadamente igual ao tempo de execução em uma das UPs, já que o sistema considerado é homogêneo. Como o tempo de execução em cada UP é dado aproximadamente por T_s/N, onde T_s representa o tempo de execução sequencial, N representa o número de UPs não-falhas e considera-se que o tempo de comunicação é consideravelmente menor que o tempo de computação, é esperado que este tempo de execução paralela cresça segundo a curva $1/N$ quando N diminui.

Apresentadas as simulações sobre sistemas homogêneos, serão realizadas agora simulações em grupos de UPs heterogêneos.

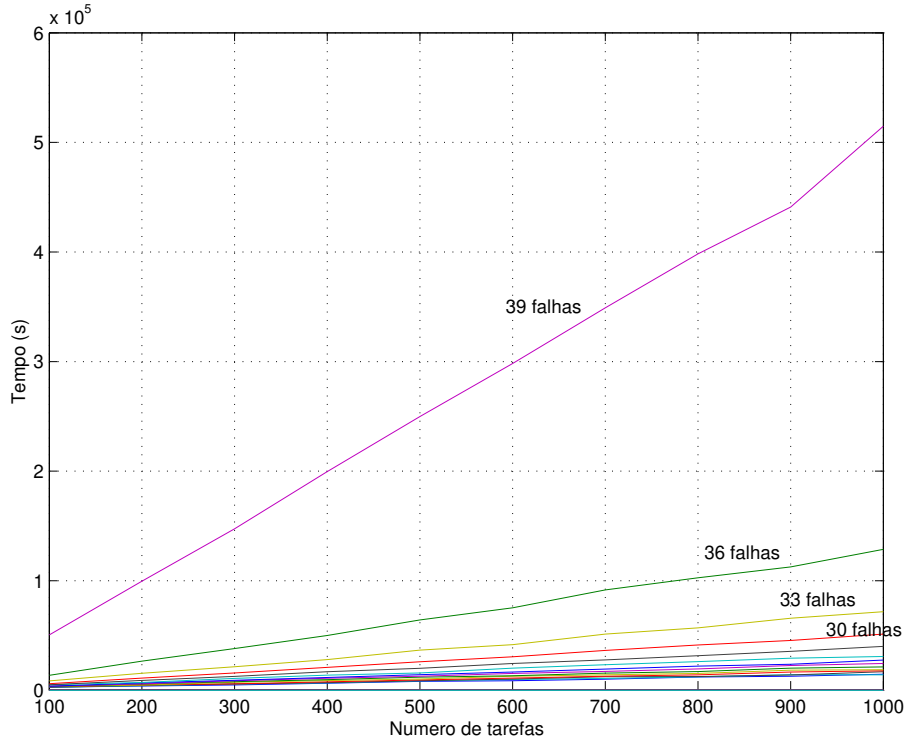

Figura 5.9: Tempos de execução em sistemas com número crescente de falhas.

5.1.3 Simulações sobre grupos heterogêneos

Grupos heterogêneos são formados por unidades de processamento com características distintas e canais de comunicação com capacidades variadas. As características das UPs, definidas por suas configurações de software e hardware, podem influenciar os tempos de processamento das tarefas de duas maneiras.

1. UPs com arquiteturas diferentes podem executar determinados tipos de tarefas de maneira mais ou menos eficiente. Assim, se $t(i)$ denotar o tempo de execução de uma tarefa i, em uma UP pode-se ter $t(i) > t(j)$ enquanto que em outra UP tem-se $t(i) < t(j)$.

2. UPs com arquiteturas similares porém com configurações distintas, tanto de hardware quanto de software, podem executar mais ou menos eficientemente uma mesma tarefa. Dito de outra forma, apesar dos tempos $t(i)$ e $t(j)$ poderem mudar de UP para UP, se $t(i) < t(j)$ numa determinada UP, então esta relação sempre se verificará nas UPs de mesma arquitetura.

Desta forma, há dois fatores que definem a heterogeneidade de processamento num grupo: as diferenças de arquitetura das UPs, que fazem com que determinadas operações sejam executadas mais eficientemente que outras em determinadas UPs, e as diferenças de versões de hardware e software das UPs, que fazem

com que UPs com arquiteturas semelhantes apresentem desempenhos globais diferentes. Para que fique mais clara a influência destes fatores sobre os tempos de execução de tarefas em grupos heterogêneos, na Fig. 5.10 são apresentados quatro gráficos que ilustram quatro níveis possíveis de heterogeneidade em um grupo de UPs.

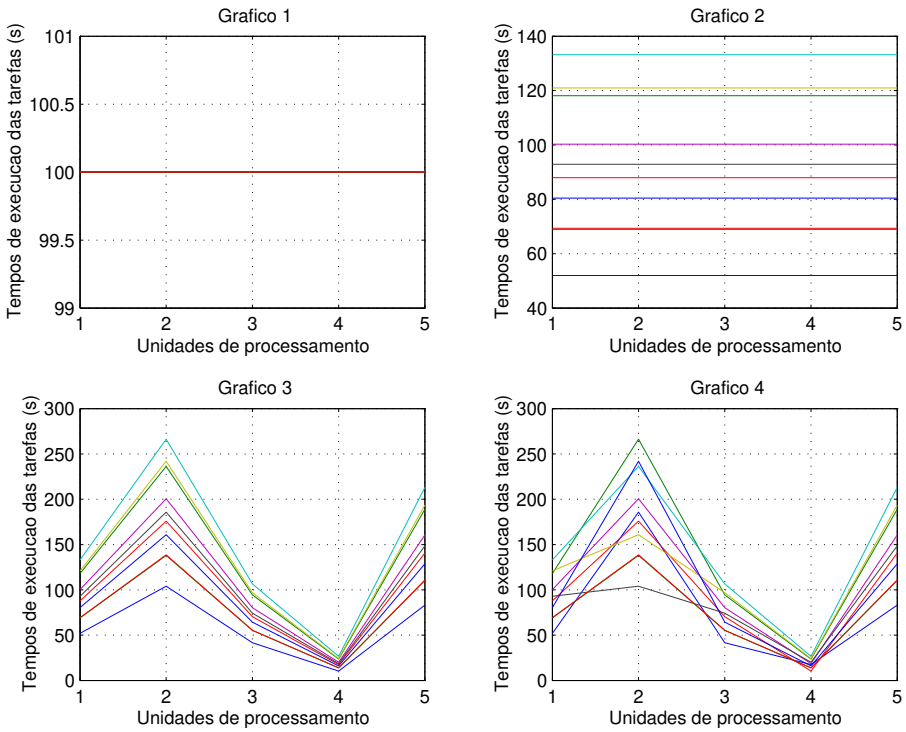

Figura 5.10: Tempos de execução de 10 tarefas em grupos com níveis crescentes de heterogeneidade.

Nestes gráficos, são apresentados os tempos de execução de 10 tarefas em 5 UPs hipotéticas. No eixo das abcissas do gráfico são representadas cada uma das UPs enquanto que cada curva representa os tempos de execução de uma determinada tarefa nas 5 UPs. Cada um dos gráficos desta figura possui a seguinte interpretação:

1. No primeiro gráfico é apresentada uma reta que mantém-se em um valor constante. Este gráfico representa o caso em que todas as tarefas de uma aplicação são executadas no mesmo tempo em todas as UPs. Este caso não é factível de ser encontrado na prática já que mesmo num sistema homogêneo tarefas com características diferentes são executadas em tempos diferentes nas UPs.

2. No segundo gráfico são apresentadas diversas retas que também mantém-se em valores constantes, cada qual representando os tempos de execução de uma determinada tarefa nas UPs do grupo. Este caso ilustra a distribuição dos tempos de execução de tarefas com características distintas num grupo homogêneo, já que uma mesma tarefa é executada no mesmo tempo em todas as UPs.

3. No terceiro gráfico são apresentadas curvas lineares por partes e que nunca se cruzam. Este gráfico ilustra a distribuição dos tempos de execução de tarefas distintas em um grupo heterogêneo onde as UPs possuem desempenhos globais diferentes mas a mesma arquitetura interna. Pelo fato das UPs possuírem a mesma arquitetura, as curvas não se cruzam já que as diferenças existentes entre as UPs afetam seus desempenhos de maneira global, dilatando ou contraindo os tempos de execução das tarefas.

4. No quarto gráfico são apresentadas curvas lineares por partes e que se cruzam. Este gráfico ilustra a distribuição dos tempos de execução de tarefas distintas em um grupo heterogêneo onde as UPs possuem desempenhos globais e arquiteturas internas diferentes. Pelo fato de possuírem arquiteturas diferentes, algumas UPs executam determinadas tarefas mais ou menos eficientemente que outras, o que se reflete graficamente no cruzamento de curvas de tempos de execução.

Seguindo o princípio ilustrado por estes gráficos, para a geração de grupos heterogêneos nas simulações, foi adotado o procedimento descrito a seguir.

1. Foram gerados aleatoriamente os tempos de execução das tarefas no grupo como se este fosse homogêneo. Assim, partindo-se de um tempo médio geral de execução de tarefas (quadro 1 da Fig. 5.10) e de um desvio padrão, foram gerados os tempos de cada tarefa segundo uma distribuição uniforme (quadro 2 da Fig. 5.10).

2. Os tempos de execução das tarefas em cada UP foram dilatados ou contraídos para se simular as eventuais diferenças nas versões de software e hardware das UPs (quadro 3 da Fig. 5.10).

3. Por fim, os tempos de execução das tarefas em cada UP foram embaralhados de modo a se simular as diferenças de arquitetura entre as UPs (quadro 4 da Fig. 5.10). Este embaralhamento foi feito redefinindo-se os tempos de execução das tarefas em cada UP como sendo uniformemente distribuídos dentro do intervalo de tempo definido pela tarefa mais rápida e mais lenta em cada UP.

Ainda foi necessária a determinação do grau de heterogeneidade da comunicação entre UPs que, neste caso, restringiu-se à comunicação entre unidades trabalhadoras e a unidade coordenadora. Os tempos de comunicação foram gerados como sendo uniformemente distribuídos em torno de uma média geral.

Na primeira bateria de simulações sobre grupos heterogêneos, é adotado o mesmo procedimento utilizado nas simulações sobre grupos homogêneos: cinco laços são encadeados de modo a promover a iteração dos vários parâmetros utilizados nas simulações. Uma transcrição destes laços pode ser vista na Fig. 5.11.

Uma diferença com relação aos valores utilizados na seção anterior está no parâmetro T, correspondente ao número de tarefas das aplicações, que varia de 200 a 1000 em passos de 200. Essa simplificação, que visa diminuir o volume de dados gerados e o tempo consumido nas simulações, se justifica pelo fato da variação deste parâmetro provocar uma variação previsível (linear) no tempo total de execução da aplicação.

```
para N (UPs) variando de 10 a 50 em intervalos de 10,
    para K (tempo) variando de 1 s a 7 s em intervalos de 2,
        para B (lotes) variando de 5 a 11 em intervalos de 2,
            para H (adimensional) variando de 10 a 25 em intervalos de 5
                gerar um grupo heterogêneo que contenha N UPs. Os tempos de
                execução das tarefas são definidos da seguinte forma:
                    gerar tempos de execução de tarefas como se grupo fosse
                    homogêneo. Utilizar como tempo médio 50*K e dispersão de 10 \%,
                    dilatar ou contrair tempos de execução de modo a alterar as
                    médias de tempo de cada UP em H*K em média,
                    embaralhar tempos de execução em cada UP,
                os tempos de comunicação são definidos por 10 +/- 3 s,
                para T (tarefas) variando de 200 a 1000 em intervalos de 200,
                    gerar uma aplicação que contenha T tarefas e B lotes,
                    simular escalonamento da aplicação gerada no grupo gerado
                    utilizando-se escalonadores diversos,
                fim,
            fim,
        fim,
    fim,
fim.
```

Figura 5.11: Algoritmo de geração dos cenários de simulação para grupos heterogêneos

Além disso, há também a introdução do parâmetro H, que influencia o grau de heterogeneidade de processamento nas UPs. A variação deste parâmetro de 10 a 25 em intervalos de 5, implica na variação dos tempos médios de execução de tarefas nas UPs em 20, 30, 40 e 50 % em relação à média, tanto para cima quanto para baixo. Como pode ser verificado na figura, a variação nos tempos de processamento das tarefas dentro de cada UP mantém-se constante em 10 % do valor da média local e os tempos de comunicação variam uniformemente em 3 s ao redor da média de 10 s.

Os resultados desta primeira bateria de simulações podem ser vistos na Fig. 5.12.

No gráfico desta figura há duas curvas: como não foram introduzidos erros de estimativas dos tempos de processamento e comunicação nestas simulações, os tempos gerados pelos escalonadores estáticos e seus correspondentes dinâmicos foi o mesmo. Assim, a curva superior corresponde ao tempo de execução obtido pelo emprego do algoritmo trivial, enquanto que a curva inferior representa o tempo de execução obtido pelo emprego do algoritmo *best-fit*.

Como se pode observar, o aspecto geral deste gráfico é muito semelhante ao do gráfico da Fig. 5.2, evidenciando o fato de que, num sistema heterogêneo, os parâmetros das simulações afetam globalmente os tempos finais de execução da mesma forma que nos sistemas homogêneos. No gráfico apresentado na Fig. 5.13, é destacado o primeiro terço do gráfico anterior.

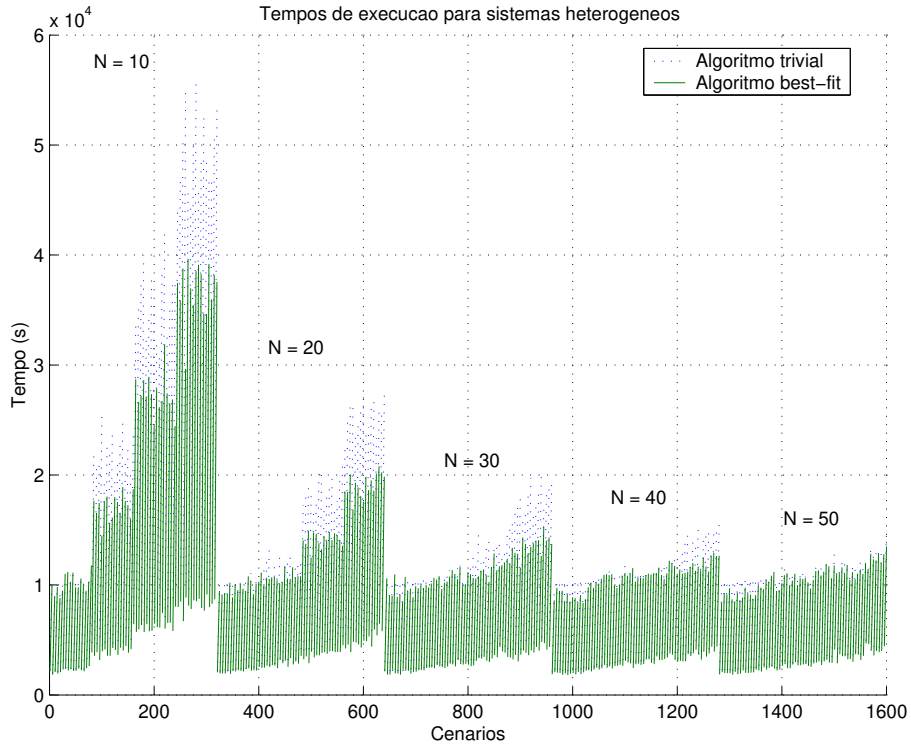

Figura 5.12: Tempos de execução dos 1600 cenários gerados com sistemas heterogêneos.

Nas Figs. 5.14 e 5.15, são apresentados gráficos que demonstram a influência da variação do parâmetro H (grau de heterogeneidade) sobre os tempos de simulação para B constante e igual a 5 e 11, respectivamente.

Nestes gráficos, cada curva representa o quociente entre o tempo fornecido pelo escalonamento realizado com o algoritmo trivial e o tempo fornecido pelo escalonamento do algoritmo *best-fit* (t/bf). Como pode ser observado, na grande maioria dos casos essa relação é maior que 1, demonstrando que o algoritmo *best-fit* é quase sempre melhor que o trivial.

Podem-se observar três fenômenos distintos nos gráficos destas duas figuras. O primeiro e mais evidente é o causado pelo aumento da heterogeneidade no sistema; como pode ser observado, este aumento provoca um aumento direto na relação entre os tempos fornecidos pelos algoritmos de escalonamento *best-fit* e trivial. Esta diferença deve-se à crescente piora de desempenho sofrida pelo algoritmo trivial na ocorrência de heterogeneidade; de fato, como este algoritmo procura distribuir a mesma carga computacional entre as UPs, então o tempo de computação total sempre será determinado pela máquina mais lenta, o que não ocorre com o algoritmo *best-fit.*

O segundo fenômeno observado nestes gráficos é uma consequência da sensibilidade dos tempos de execução ao aumento do parâmetro K. Observando os diferentes gráficos de uma mesma figura, percebe-

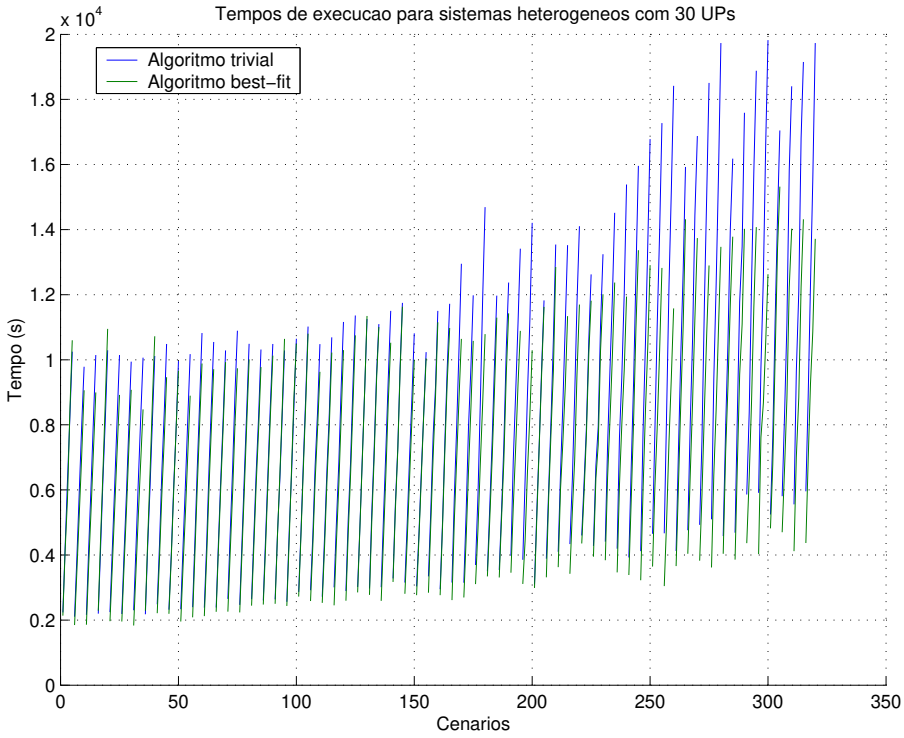

Figura 5.13: Tempos de execução dos 320 cenários centrais, onde N = 30.

se que o aumento de K faz com que a influência da heterogeneidade sobre o quociente dos tempos dos algoritmos seja mais forte. Esta sensibilidade à heterogeneidade também ocorre pelo fato da qualidade do escalonamento fornecido pelo algoritmo trivial diminuir à medida que a heterogeneidade do sistema cresce. Para valores grandes de K, as distorções causadas pela distribuição igualitária de carga entre as UPs são amplificadas, aumentando assim o quociente entre os tempos de execução dos escalonamentos fornecidos pelos dois algoritmos.

O terceiro e menos evidente fenômeno observado diz respeito à influência da variação do parâmetro B. Como pode ser observado, este parâmetro influencia os gráficos de maneira irregular e pouco previsível. O que se pode concluir é que com o aumento de B há uma pequena diminuição nas diferenças entre os tempos gerados pelos dois algoritmos.

A mesma análise realizada para valores fixos de B pode ser feita para valores fixos de K. As Figs. 5.16 e 5.17 apresentam os gráficos resultantes da fixação do parâmetro K em 1 e 7, respectivamente.

Conforme o verificado nos gráficos anteriores, pode-se observar que para determinados valores de K a influência da heterogeneidade do sistema (parâmetro H) sobre os quocientes dos tempos de execução pode ser grande. Na Fig. 5.16, para K igual a 1, a variação do parâmetro H pouco influencia o comportamento das curvas. Para valores maiores de K, como K igual a 7 na Fig. 5.17, a relação entre os tempos obtidos pelos diferentes algoritmos de escalonamento cresce com o aumento de H.

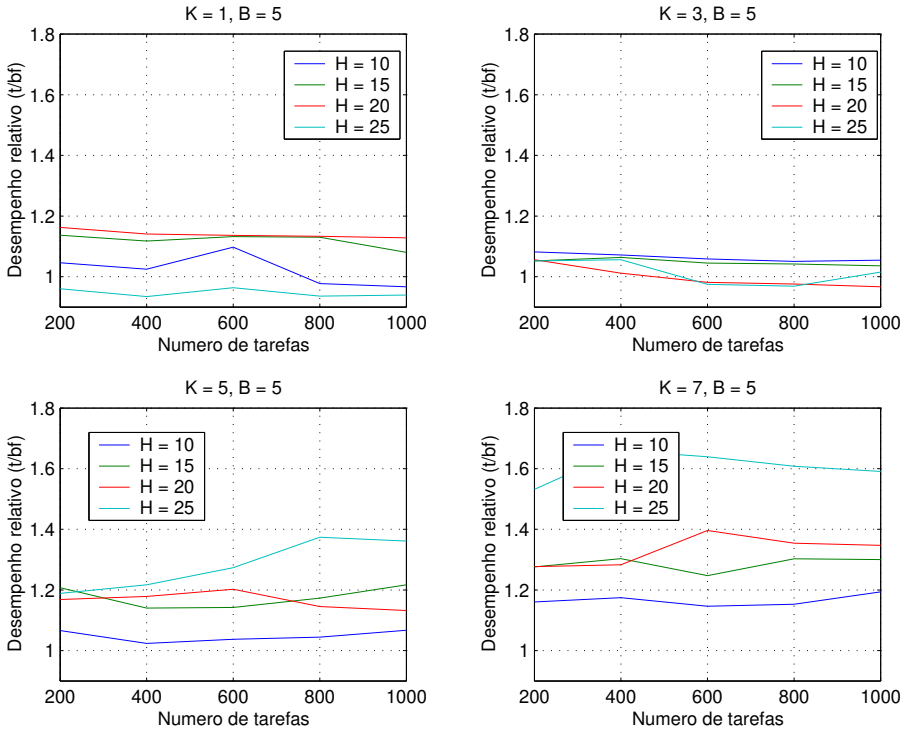

Figura 5.14: Quociente dos tempos de execução dos algoritmos trivial e best-fit obtidos com heterogeneidade crescente, N = 30 e B = 5.

O próximo passo é introduzir erros de estimativas nos tempos de processamento e comunicação das UPs do grupo para se avaliar as diferenças de desempenho entre os escalonadores estáticos e dinâmicos. Para que a introdução de erros nos modelos pudesse ser feita de forma a viabilizar a comparação entre os diferentes mecanismos de escalonamento, procurou-se adotar a introdução de um erro determinístico, de modo que pudesse ser reproduzido em todas as simulações. Para isso, a função *senoidal* serviu plenamente aos propósitos destas simulações já que, além de determinística, possui média 0.

O nível máximo deste erro, ou seja, a amplitude da função senoidal, foi um dos parâmetros variados nas simulações. O número de UPs sobre as quais este erro de estimativas incidia foi o segundo parâmetro de simulação a ser variado.

Nesta etapa foram mantidos fixos o número de UPs em 40, o número de lotes em 5, o número de tarefas em 300, o tempo médio de processamento das tarefas em 500 s e o tempo de comunicação em 10 +/- 3 s. O grau de heterogeneidade de processamento do grupo foi determinado da seguinte forma: tempos iniciais de execução das tarefas distribuídos em uma faixa de 10 % da média geral; dilatações e contrações nestes tempos de maneira a deslocar as médias em cada UP em 150 s; embaralhamento dos tempos. Conforme mencionado no parágrafo anterior, foram variados os parâmetros que determinam o nível de erro das estimativas e o número de UPs em que este erro incidiu; o primeiro parâmetro foi iterado

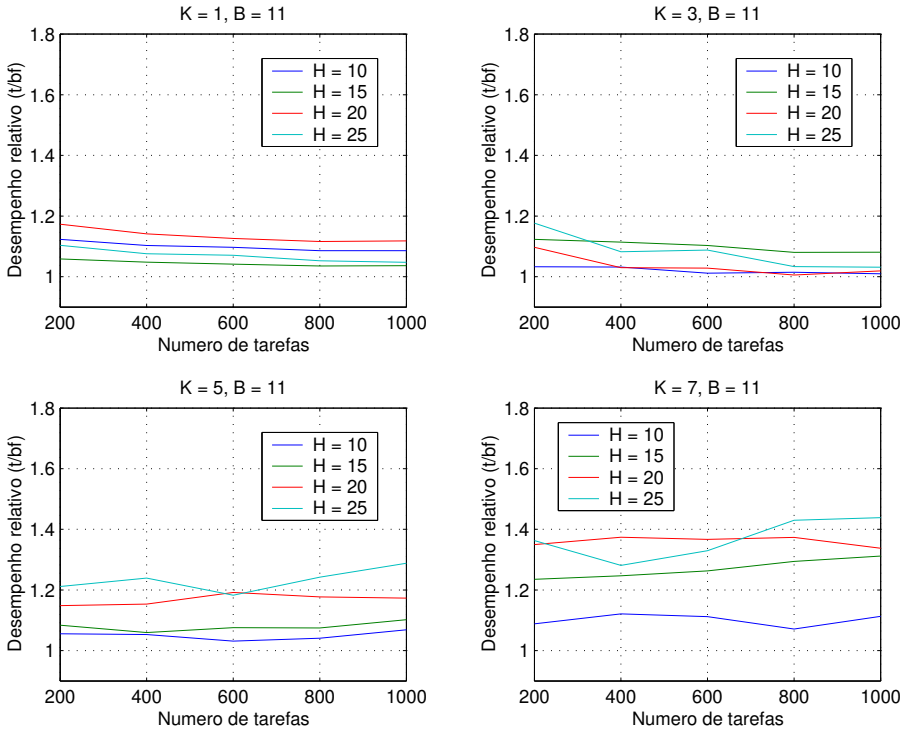

Figura 5.15: Quociente dos tempos de execução dos algoritmos trivial e best-fit obtidos com heterogeneidade crescente, N = 30 e B = 11.

de 0.2 a 0.8 em passos de 0.2, o que significa dizer que os níveis máximos de erro nas simulações foram de 20% a 80% de desvio; já para o segundo parâmetro, foram selecionadas de 1 a 20 UPs para sofrer a influência destes erros, ou seja, de 2,5% a 50% das UPs do grupo.

Na Fig. 5.18, pode ser visto o gráfico com os tempos de execução resultantes desta bateria de simulações.

Pode-se perceber por estes gráficos que o aumento no número de UPs em que incidiram erros nas estimativas dos tempos provoca uma variação irregular nos tempos de execução das aplicações. Entretanto, percebe-se que o aumento no nível de erro introduzido nestas UPs aumentou significativamente a diferença entre os tempos fornecidos por diferentes escalonadores. Nota-se também que os tempos fornecidos pelos escalonadores estático e GS variaram de maneira bem mais irregular que os tempos do escalonador GSTR, que por sua vez apresentou comportamento mais estável e previsível.

Essa grande diferença na estabilidade dos tempos de execução se deve ao fato do escalonador GSTR compensar erros de estimativas com replicação de tarefas. Assim, se uma tarefa previamente alocada a uma determinada UP demorar um tempo demasiadamente grande para ser concluída, esta tarefa será replicada a outra UP diminuindo-se a probabilidade de que este atraso comprometa o tempo global de execução da aplicação. Uma outra possibilidade é a replicação de tarefas em UPs ociosas, que aumenta o desempenho

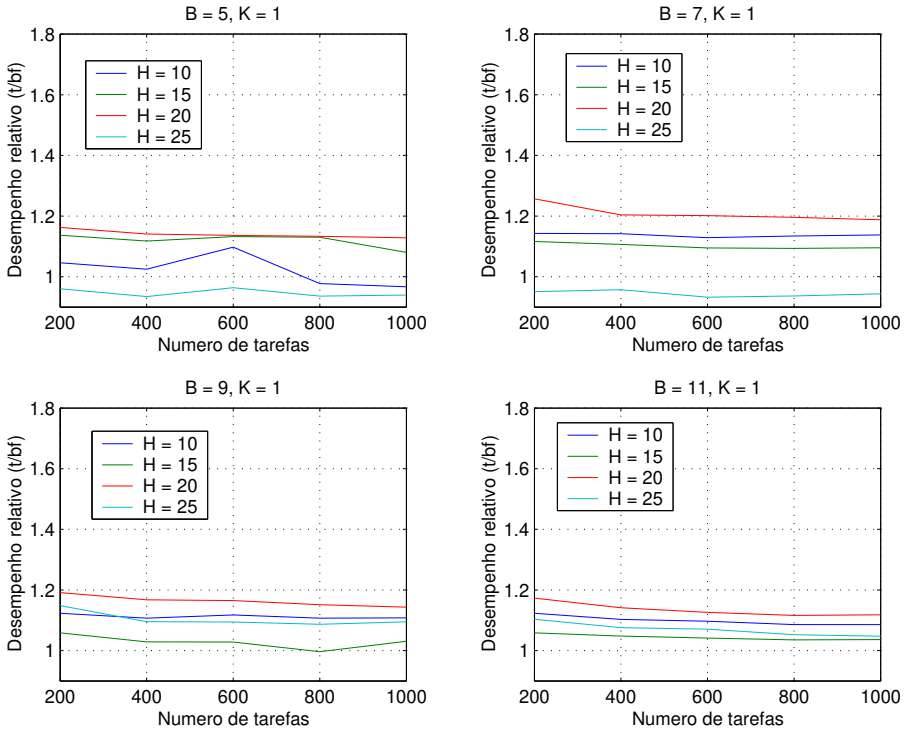

Figura 5.16: Quociente dos tempos de execução dos algoritmos trivial e best-fit obtidos com heterogeneidade crescente, N = 30 e K = 1.

global da execução.

Na próxima bateria de simulações é avaliada a influência das falhas nas UPs sobre o tempo global de execução das aplicações. Semelhantemente ao realizado nas simulações anteriores, para esta bateria são fixados o número de UPs em 40, o número de lotes em 5 e o número de tarefas em 300. Não será introduzido nenhum nível de erro e a heterogeneidade do sistema será constante e definida pelos mesmos valores empregados nas simulações anteriores. O único laço presente na simulação se encarregará de gerar sistemas heterogêneos com falhas em 1 a 39 UPs. Os resultados podem ser vistos na Fig. 5.19.

As duas curvas deste gráfico representam os tempos de execução obtidos pela aplicação do escalonador GSTR com algoritmos trivial e *best-fit*. O comportamento global das curvas é essencialmente o mesmo do apresentado nos sistemas homogêneos a não ser pela suave irregularidade resultante da introdução da heterogeneidade nos sistemas.

5.1.4 Resumo dos resultados das simulações de escalonamento intra-grupo

A seguir é apresentado um resumo das principais conclusões obtidas pela execução das simulações descritas anteriormente.

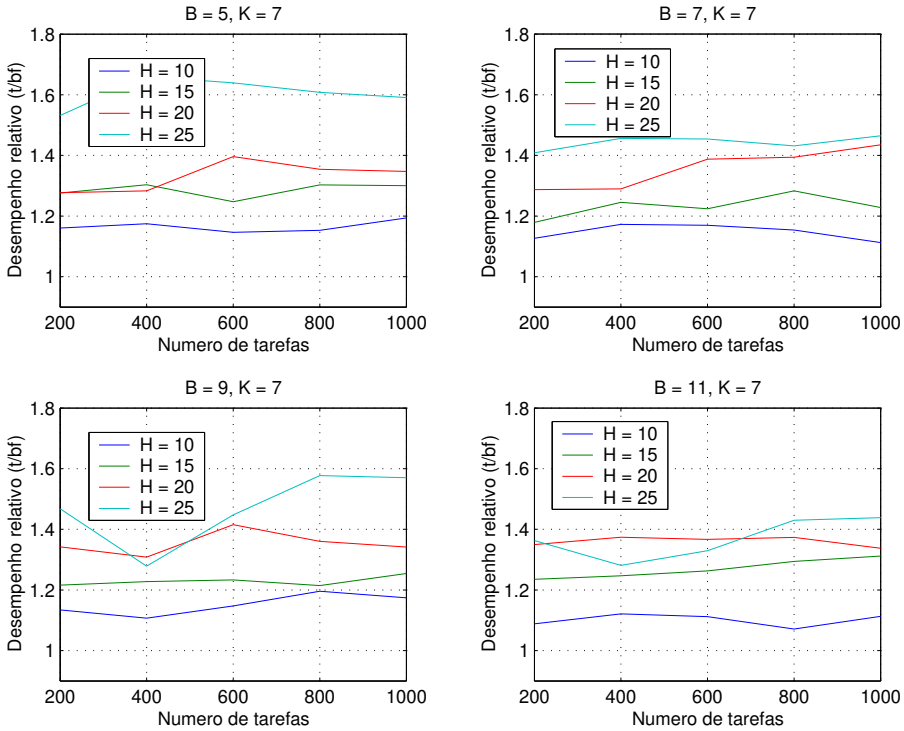

Figura 5.17: Quociente dos tempos de execução dos algoritmos trivial e best-fit obtidos com heterogeneidade crescente, N = 30 e K = 7.

Conclusões válidas para grupos homogêneos e heterogêneos:

Em ambos os tipos de grupos:

1. O tempo de execução de uma aplicação é diretamente proporcional ao número de tarefas dessa aplicação e ao tempo médio de execução destas tarefas.
2. O *speed-up* é maior quando:
   (a) aumenta a relação $\frac{T_p}{T_c}$, onde T_p representa o tempo total de computação e T_c o tempo total de comunicação,
   (b) diminui o número de unidades de processamento do grupo, até um determinado valor de saturação.
3. O *speed-up* satura-se quando:
   (a) diminui a cardinalidade dos lotes de uma aplicação,
   (b) diminui o grau de paralelismo com que os lotes de uma aplicação são executados.

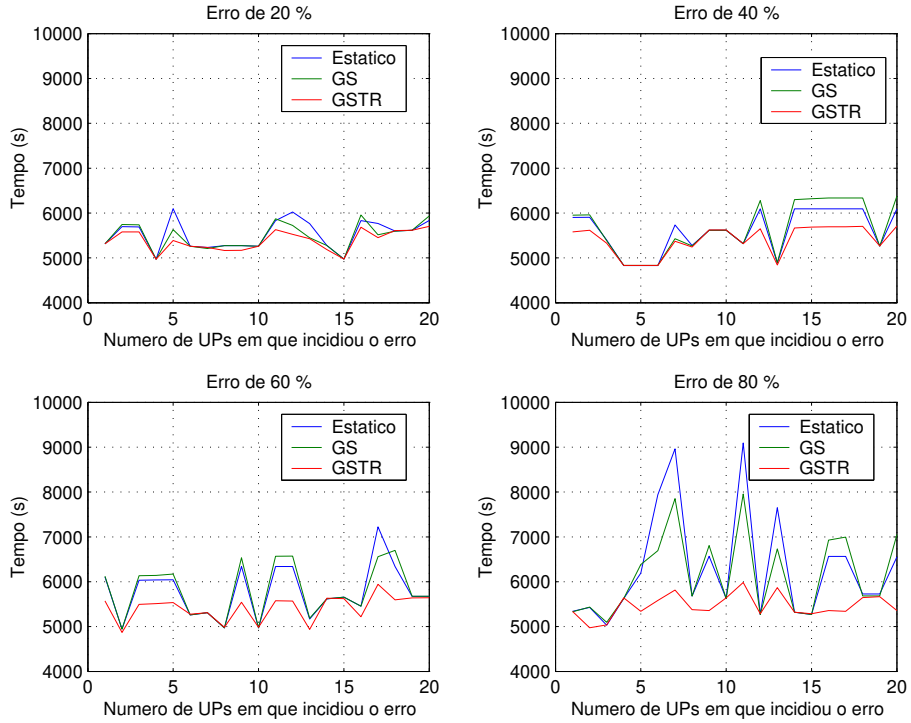

Figura 5.18: Tempos de execução de aplicações escalonadas em sistemas heterogêneos com erros crescentes de estimativas.

(c) aumenta o número de unidades de processamento do grupo além do valor de saturação.

4. O tempo de execução de uma aplicação em um grupo cresce à taxa de $\frac{1}{UPs_{totais} - UPs_{falhas}}$ quando o número de UPs falhas aumenta.

Conclusões válidas somente para os grupos homogêneos

Especificamente para os grupos homogêneos, verificou-se que os algoritmos trivial e best-fit fornecem as soluções ótimas, já que alocam o mesmo número de tarefas a cada UP.

Conclusões válidas somente para os grupos heterogêneos

Em um grupo heterogêneo:

1. os escalonadores que utilizam o algoritmo *best-fit* apresentam, frente aos que utilizam o algoritmo trivial, melhora no desempenho médio dos escalonamentos,

2. os escalonadores estático e GS apresentaram performance irregular quando da introdução de erros nas estimativas dos tempos de computação e comunicação nas UPs. Já o algoritmo GSTR apresentou-se

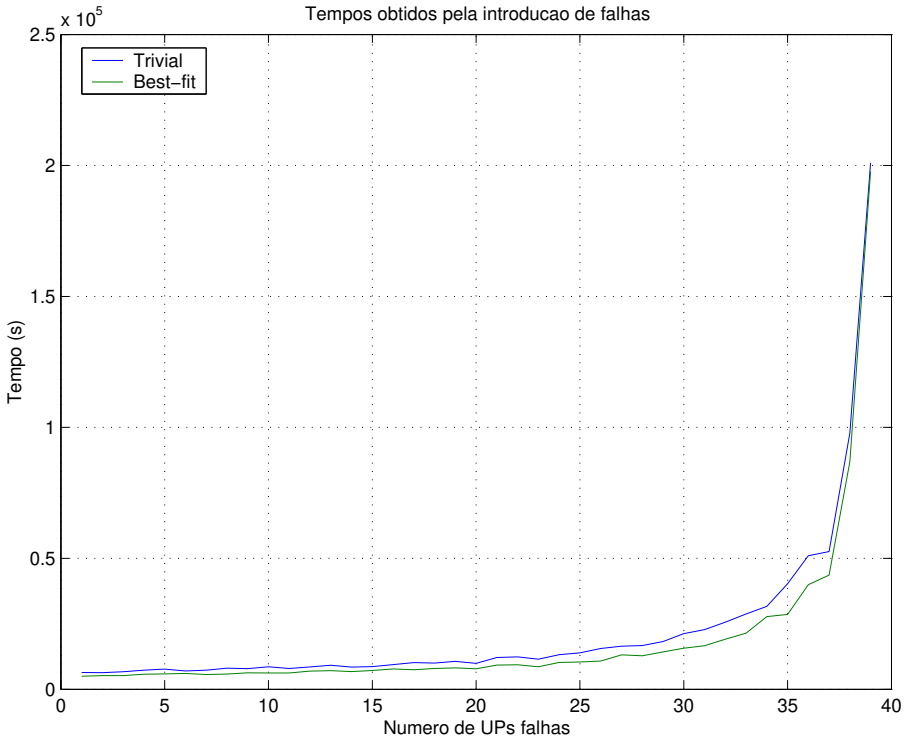

Figura 5.19: Tempos de execução em sistemas com número crescente de falhas.

bastante robusto a esta introdução de erros.

5.2 Simulações de escalonamentos inter-grupos

5.2.1 Características das simulações

As simulações desenvolvidas nesta seção procuram apresentar as vantagens e desvantagens do algoritmo de balanceamento PPCP frente a dois algoritmos de balanceamento de carga entre grupos de UPs.

Nestas simulações, são aplicadas as seguintes restrições:

- são considerados MPVCs com 2, 3 e 4 grupos heterogêneos,
- o poder computacional de cada grupo será aproximado pelo seu número de UPs,
- em cada grupo será aplicado o balanceamento de carga GSTR/best-fit,
- o tempo médio de execução das tarefas nos grupos é constante,
- o tamanho dos blocos de dados utilizados por todas as tarefas é constante.

Como na seção anterior, para a realização destas simulações foi desenvolvido um conjunto de funções **MATLAB** que modelam o comportamento dinâmico de um MPVC formado por um número arbitrário de grupos.

As funções desenvolvidas foram:

- **tp**: esta função implementa o cálculo de um particionamento trivial em uma aplicação qualquer. Para isto, necessita do número de grupos em que a aplicação deve ser particionada e o número de tarefas contidas em cada lote. O particionamento trivial procura distribuir a mesma quantidade de tarefas entre os grupos, privilegiando grupos de menor CR quando arredondamentos se fazem necessários,
- **pp**: esta função implementa o cálculo de um particionamento proporcional em uma aplicação qualquer. Por "particionamento proporcional", entende-se um particionamento sobre os lotes de tarefas que procura seguir as relações de proporcionalidade entre as cardinalidades dos grupos de UPs. Para a realização deste cálculo, a função necessita do número de UPs em cada grupo e da cardinalidade de cada lote de tarefas,
- **ppcp**: esta função implementa o cálculo do particionamento PPCP em uma aplicação qualquer. Para isso, necessita da especificação completa da aplicação e do número de UPs contidas em cada grupo,
- **igp**: esta função é a responsável pela simulação propriamente dita do escalonamento de tarefas entre UPs. Para isso, precisa da especificação do MPVC, da aplicação e do particionamento a ser executado.

Maiores detalhes a respeito de cada uma destas funções podem ser encontrados no Apêndice B, incluindo o código-fonte das mesmas.

Analogamente ao realizado na seção anterior, foram gerados diversos cenários para a simulação dos escalonamentos de tarefas inter-grupos. Na Fig. 5.20 é apresentado o algoritmo geral utilizado para estas simulações.

Como nas seções anteriores, o algoritmo apresentado na Fig. 5.20 apenas descreve de forma genérica a metodologia utilizada para a geração dos cenários de simulação uma vez que não especifica a quantidade nem as características dos grupos de UPs. As particularidades de cada simulação realizada serão apresentadas nas sub-seções a seguir.

5.2.2 Simulações de escalabilidade de UPs

Nesta seção, são apresentadas simulações que procuram avaliar a escalabilidade do modelo de MPVC adotado quando o número de UPs deste MPVC é incrementado pelo acréscimo de novos grupos. Assim, é arbitrado um MPVC inicial com apenas 2 grupos sobre o qual são feitas simulações preliminares e no qual são acrescidos novos grupos gradativamente.

Arbitrou-se que este MPVC possuiria inicialmente 2 grupos com 30 UPs cada. Este valor foi escolhidos por pertencer à faixa de valores simulados para os escalonamentos intra-grupo e por estar abaixo da faixa

```
para B (lotes) variando de 4 a 10 em intervalos de 2,
    para T (tarefas) variando de 400 a 1000 em intervalos de 200,
        gerar um conjunto de grupos heterogêneos que contenham um número
        pré-determinado de UPs. Os tempos de execução das tarefas nestes
        grupos são definidos da seguinte forma:
            gerar tempos de execução de tarefas como se grupos fossem
            homogêneos. Utilizar como tempo médio 500 s e dispersão de 10 %,
            dilatar ou contrair tempos de execução de modo a alterar as
            médias de tempo de cada UP em 100 s em média,
            embaralhar tempos de execução em cada UP,
            os tempos de comunicação são definidos por 10 +/- 3 s,
        gerar uma aplicação que contenha T tarefas e B lotes,
        escalonar aplicação neste grupo de UPs,
    fim,
fim.
```

Figura 5.20: Algoritmo de geração dos cenários de simulação para escalonamento inter-grupos

Lotes		4				6				8				10			
Tarefas (x100)		4	6	8	10	4	6	8	10	4	6	8	10	4	6	8	10
GHs	2	79	104	135	161	88	118	144	179	89	113	156	169	104	137	153	179
	3	58	78	97	111	72	95	104	123	71	91	114	139	80	98	118	139
	(%)	-26	-25	-28	-31	-18	-19	-28	-31	-20	-19	-27	-18	-23	-29	-23	-22
	4	48	63	76	93	57	70	86	100	61	90	96	109	70	96	101	117
	(%)	-39	-40	-43	-42	-35	-40	-40	-44	-32	-20	-38	-36	-33	-30	-34	-35

Tabela 5.1: Tempos de processamento nos GHs (em min).

de saturação superior. Após serem feitas as simulações com 2 GHs, acrescentou-se um GH também com 30 UPs ao MPVC e repetiram-se as simulações. Este processo foi repetido para um MPVC com 4 GHs de 30 UPs.

Na Tabela 5.1 são apresentados os resultados obtidos nestas 3 baterias de simulações.

Nesta tabela, os valores numéricos referem-se à média das execuções de aplicações com 400, 600, 800 e 1000 tarefas. Os valores percentuais referem-se às variações obtidas pela adição de GHs **ao MPVC de 2 GHs**. Assim, a primeira linha de valores percentuais refere-se às variações nos tempos do MPVC de 3 GHs em relação aos de 2 GHs, e a segunda linha refere-se às variações nos tempos do MPVC de 4 GHs **também** em relação ao de 2 GHs.

Um ponto importante a ser destacado nesta tabela é que os tempos totais apresentados não consideram os tempos de comunicação entre os grupos. Isto se deve ao fato destes tempos poderem variar significativamente dependendo do ambiente adotado para a operação das unidades coordenadoras. Caso estas estejam localizadas, por exemplo, em pontos relativamente distantes na Internet, então o tempo de comunicação

Lotes		4				6				8				10			
Tarefas (x100)		4	6	8	10	4	6	8	10	4	6	8	10	4	6	8	10
GHs	2	99	298	597	498	197	150	398	497	396	296	553	744	224	708	682	655
	3	264	398	798	664	264	500	1060	663	264	792	742	1328	425	560	1056	498
	(%)	167	34	34	33	34	233	166	33	-33	168	34	78	90	-21	55	-24
	4	297	447	897	747	375	865	596	1119	441	1110	990	1829	462	657	1192	1060
	(%)	200	50	50	50	90	477	50	125	11	275	79	146	106	-7	75	62

Tabela 5.2: Número de pacotes trocados entre GHs.

entre GHs tende a ser maior do que os tempos de comunicação dentro de um mesmo GH. Para que se possa avaliar o impacto de cada algoritmo de escalonamento sobre estes tempos de comunicação nos cenários simulados, a Tabela 5.2 apresenta o número de pacotes trocados entre os GHs para cada cenário simulado.

Um outro ponto importante a ser detacado nas duas tabelas é que, como os grupos dos MPVCs simulados são iguais, isto é, todos possuem 30 UPs, então todos os algoritmos de particionamento forneceram o mesmo resultado: de fato, mesmo o particionador trivial, que é o mais simples de todos, "acerta" ao supor que os grupos são idênticos. É por isso que é apresentado apenas um resultado válido para todos os algoritmos particionadores.

Nestas duas tabelas, observa-se que:

- os tempos de computação decrescem com o aumento no número de UPs do MPVC,
- o número de pacotes trocados entre GHs varia significativamente de cenário a cenário. Entretanto, observa-se que, em média, este número aumentou com o aumento no número de GHs.

Dentre estas observações, a única que refere-se a um comportamento não trivial é a última. De fato, o número de pacotes trocados entre GHs depende diretamente da relação entre o número de pacotes trocados entre lotes de tarefas e o número de UPs de cada grupo. Como as aplicações são geradas aleatoriamente, as relações de dependência entre os lotes de uma aplicação podem variar grandemente, variando assim a maneira como os pacotes são trocados entre os lotes. Para os valores de número de lotes e número de UPs arbitrados nestes cenários, há casos em que esta relação sofre variações bruscas, refletidas pelas grandes variações percentuais encontradas na tabela (como quando há 400 tarefas e 8 lotes).

Uma análise que ainda pode ser feita diz respeito à eficiência com que a aplicação é executada nos GHs do MPVC. A adição do terceiro e do quarto GHs ao MPVC de 2 GHs, aumenta seu poder computaconal em 50 % e 100 %, respectivamente. Pode-se observar pela Tabela 5.1 que, apesar de estarem aquém destes valores ideais, os valores de desempenho reais cresceram com o aumento do MPVC, como exige a Definição 10.

Na seção a seguir são apresentadas as simulações para avaliação da escalabilidade dos grupos do MPVC.

Lotes		4				6				8				10			
Tarefas (x100)		4	6	8	10	4	6	8	10	4	6	8	10	4	6	8	10
GHs	2	79	104	135	161	88	118	144	179	89	113	156	169	104	137	153	179
	3	79	106	135	165	83	113	138	171	80	115	151	175	104	113	163	183
	(%)	0	1	1	2	-6	-4	-4	-5	-10	2	-3	3	0	-18	6	2
	4	80	101	131	163	89	117	139	169	87	117	149	175	94	132	170	184
	(%)	2	-3	-3	1	2	-1	-3	-6	-2	4	-5	3	-10	-3	11	3

Tabela 5.3: Tempos de processamento nos GHs (em min).

Lotes		4				6				8				10			
Tarefas (x100)		4	6	8	10	4	6	8	10	4	6	8	10	4	6	8	10
GHs	2	99	298	597	498	197	150	398	497	396	296	553	744	224	708	682	655
	3	264	398	531	664	330	660	531	970	499	474	352	1655	330	511	520	747
	(%)	167	34	-11	33	68	340	33	95	26	60	-36	122	47	-28	-24	14
	4	298	447	897	1122	444	447	596	746	597	558	1071	496	296	756	1190	744
	(%)	201	50	50	125	125	198	50	50	51	89	94	-33	32	7	74	14

Tabela 5.4: Número de pacotes trocados entre GHs.

5.2.3 Simulações de escalabilidade de grupos

Nesta seção, são apresentadas simulações que procuram avaliar a escalabilidade do modelo de MPVC adotado quando o número de GHs deste MPVC é incrementado. Assim, é arbitrado um MPVC inicial com apenas 2 grupos sobre o qual são feitas simulações preliminares e no qual são acrescidos novos grupos gradativamente, **mantendo-se constante o número total de UPs**.

O MPVC inicial utilizado foi o mesmo da seção anterior, de modo que seus dados de simulação já são conhecidos. O MPVC de 3 GHs utilizado possui GHs de 20 UPs cada, enquanto que o MPVC de 4 GHs possui GHs de 15 UPs cada. Nota-se portanto que os três MPVCs simulados possuem ao todo 60 UPs, que estão igualmente distribuídas em seus GHs.

As tabelas 5.3 e 5.4 mostram respectivamente os tempos de processamento obtidos em cada MPVC e o número de pacotes trocados entre GHs de cada MPVC.

Por estas tabelas, constata-se que:

- o número de pacotes trocados entre GHs aumenta quando o número de GHs do MPVC aumenta,
- o tempo de processamento mantém-se aproximadamente constante quando o número de GHs do MPVC aumenta.

Percebe-se portanto que a eficiência com que a aplicação é executada tende a diminuir com o aumento do número de GHs no MPVC devido ao aumento no volume de comunicação entre os GHs.

Na seção a seguir são apresentadas as simulações para avaliação do desempenho dos particionadores quando os GHs não são iguais.

Lotes			4				6				8				10			
Tarefas (x100)			4	6	8	10	4	6	8	10	4	6	8	10	4	6	8	10
Caso	I	TP	48	63	76	93	57	70	86	100	61	90	96	109	70	96	101	117
		PP	48	63	76	93	57	70	86	100	61	90	96	109	70	96	101	117
		PPCP	48	63	76	93	57	70	86	100	61	90	96	109	79	96	101	117
	II	TP	56	77	101	125	74	88	110	123	67	90	112	140	89	98	128	139
		PP	44	64	78	93	63	72	92	95	64	81	81	116	87	94	107	115
		PPCP	44	64	78	93	63	72	92	95	71	81	90	116	87	94	107	115
	(%)	TP	17	23	32	34	30	25	28	23	11	0	17	29	27	2	27	18
		PP	-10	1	2	0	10	2	7	-5	6	-10	-1	7	25	-2	6	-2
		PPCP	-10	1	2	0	10	2	7	-5	18	-10	-1	7	25	-2	6	-2
	III	TP	68	100	124	155	76	106	142	166	90	112	135	176	88	117	152	178
		PP	45	68	79	93	67	74	91	102	74	90	95	108	73	91	113	117
		PPCP	45	68	79	93	67	74	91	102	82	98	95	108	75	91	113	129
	(%)	TP	42	59	63	66	34	50	65	65	47	24	40	62	26	23	51	51
		PP	-8	8	4	-1	18	5	5	1	21	0	-1	0	5	-5	12	0
		PPCP	-8	8	4	-1	18	5	5	1	35	9	-1	0	6	-5	12	10

Tabela 5.5: Tempos de processamento nos GHs (em min).

5.2.4 Simulações de heterogeneidade entre os grupos

Nesta seção são apresentadas simulações que procuram avaliar o comportamento de cada algoritmo de particionaento frente a diferenças nos poderes computacionais dos GHs de um MPVC. Para isso, é inicialmente considerado um MPVC com 4 GHs de 30 UPs cada, similar ao utilizado na primeira seção de simulações. Em seguida, são apresentadas simulações com 2 outros MPVCs de 4 GHs contendo 20, 27, 33 e 40 UPs e 15, 25, 35 e 45 UPs, respectivamente.

Desta forma, são arbitrados 3 MPVCs com números iguais de UPs mas que possuem diferentes distribuições de UPs em seus grupos. Mais especificamene, a distribuição das UPs no GHs torna-se gradativamente mais heterogênea do primeiro ao último MPVC. Os resultados destas simulações podem ser vistos nas Tabelas 5.5 e 5.6.

Por estas tabelas, verifica-se que:

- o aumento da heterogeneidade entre os GHs do MPVC piorou o desempenho do particionador trivial, como esperado (ver linhas de % do TP),
- os algoritmos PP e PPCP se mostraram menos sensíveis ao aumento da heterogeneidade entre os GHs do MPVC (ver linhas de % do PP e PPCP),
- a diferença entre os tempos de processamento do particionamento trivial e dos demais cresce à medida que a heterogeneidade entre os GHs aumenta,
- a diferença entre numero de pacotes trocados quando se muda do particionamento trivial para os outros algoritmos cresce à medida que aumenta a heterogeneidade entre os GHs;

Lotes			4				6				8				10			
Tarefas (x100)			4	6	8	10	4	6	8	10	4	6	8	10	4	6	8	10
Caso	I	TP	297	447	897	747	375	865	596	1119	441	1110	990	1829	462	657	1192	1060
		PP	297	447	897	747	375	865	596	1119	441	1110	990	1829	462	657	1192	1060
		PPCP	297	447	897	747	375	865	596	1119	441	1110	990	1829	462	657	1192	1060
	II	TP	298	448	598	748	296	448	1192	746	330	801	952	1043	259	814	1666	1984
		PP	265	398	532	664	263	400	1060	665	312	711	848	924	231	726	1484	1760
		PPCP	265	398	532	664	263	400	1060	665	271	711	848	924	231	726	1484	1760
	(%)	TP	0	0	-33	0	-21	-48	100	-33	-25	-28	-4	-43	-44	24	40	87
		PP	-11	-11	-41	-11	-30	-54	78	-41	-29	-36	-14	-49	-50	11	24	66
		PPCP	-11	-11	-41	-11	-30	-54	78	-41	-39	-36	-14	-49	-50	11	24	66
	III	TP	298	448	598	1122	225	448	600	757	147	564	1192	753	566	713	792	1452
		PP	248	372	498	933	186	372	500	630	167	534	992	628	545	611	664	1233
		PPCP	248	372	498	933	186	372	500	630	120	480	992	628	491	567	664	1131
	(%)	TP	0	0	-33	50	-40	-48	1	-32	-67	-49	20	-59	23	9	-34	37
		PP	-16	-17	-44	25	-50	-57	-16	-44	-62	-52	0	-66	18	-7	-44	16
		PPCP	-16	-17	-44	25	-50	-57	-16	-44	-73	-57	0	-66	6	-14	-44	7

Tabela 5.6: Número de pacotes trocados entre GHs.

- os tempos de processamento obtidos pela aplicação dos particionamentos proporcional e PPCP são quase sempre os mesmos, sendo que em alguns casos o particionamento proporcional fornece tempos inferiores ao PPCP,
- o número de pacotes trocados entre GHs pela aplicação dos particionamentos proporcional e PPCP é quase sempre o mesmo, sendo que em alguns casos o número fornecido pelo PPCP é inferior ao particionamento proporcional.

Como pode ser constatado, o desempenho do particionamento trivial é, conforme esperado, inferior ao dos demais particionamentos na grande maioria dos casos. Isto se deve, em primeiro lugar, ao fato dos grupos, de poderes computacionais distintos receberem a mesma carga computacional. Em segundo lugar, o particionamento trivial não considera a relação multiplicativa existente entre as cardinalidades dos lotes que trocam mensagens; desta forma, o número de pacotes trocados entre GHs é maior, o que penaliza o desempenho deste algoritmo.

Também pode ser observado que o desempenho dos particionadores proporcional e PPCP é aproximadamente o mesmo. Entretanto, em alguns casos particulares, o tempo de processamento nos GHs obtidos pela aplicação do particionamento proporcional é inferior aos tempos do PPCP. Isto se deve ao fato deste primeiro realizar um particionamento na aplicação rigorosamente proporcional aos poderes computacionais do GHs, o que resulta num maior equilibrio em suas cargas. O algoritmo PPCP, ao contrário, utiliza critérios diversos para a realização do particionamento da aplicação que, em alguns casos, significam particionamentos não-ótimos.

Por outro lado, pode-se constatar que o número de pacotes trocados entre GHs pela aplicação do PPCP é pelo menos tão pequeno quanto o fornecido pelo PP. De fato, o algoritmo PPCP, que leva em conta a relação multiplicativa entre as cardinalidades dos lotes, foi desenvolvido com o intuito de mini-

mizar a comunicação entre GHs durante a execução da aplicação paralela. Num ambiente onde a relação computação/comunicação deve ser maximizada, este algoritmo destaca-se como a opção mais adequada.

Nestas simulações, é possível observar que a escolha do melhor algoritmo de particionamento, considerando-se o PP e o PPCP, depende fortemente dos tempos de comunicação entre as unidades coordenadoras de grupos distintos. Quanto maior forem estes tempos de comunicação, mais favorável será a escolha do algoritmo PPCP, que apresenta sempre menores demandas de comunicação, ainda que em detrimento do equilíbrio ótimo de carga nos GHs.

5.2.5 Resumo dos resultados das simulações de escalonamento inter-grupo

A seguir é apresentado um resumo das principais conclusões obtidas pela execução das simulações descritas anteriormente.

1. quanto mais similares os GHs, mais próximo é o desempenho dos algoritmos de particionamento,
2. quanto mais diferentes os GHs, pior é o desempenho do particionador trivial frente aos demais,
3. os particionamentos proporcional e PPCP apresentam desempenhos próximos,
4. o particionamento proporcional apresenta os melhores tempos de processamento nos grupos,
5. o particionamento PPCP apresenta o menor volume de comunicação entre grupos.

Na seção a seguir são apresentadas as conclusões deste capítulo.

5.3 Conclusões

Neste capítulo foram apresentadas baterias de simulações que procuraram destacar as vantagens e desvantagens do esquema de escalonamento proposto no Cap. 2 e implementado no Cap. 4.

Em linhas gerais, as simulações indicaram que, para o balanceamento de carga intra-grupo, o emprego de um escalonador GSTR com escalonamento *best-fit* leva a resultados melhores que o de qualquer outro esquema considerado; além dos escalonamentos terem sido executados em menor tempo e de forma mais estável, este escalonador possui a vantagem única de ser tolerante a falhas nas unidades de processamento. Já com relação ao balanceamento de carga inter-grupos, a utilização do particionamento PPCP é a alternativa mais adequada aos MPVCs uma vez que proporciona bom equilíbrio de carga nos grupos e minimiza a comunicação entre eles.

No capítulo seguinte são apresentados testes que procuram avaliar o desempenho destes esquemas de escalonamento em um sistema real.

Capítulo 6

Testes

Neste capítulo são apresentados os resultados de uma bateria de testes realizados com a plataforma JOIN no intuito de se verificar em um ambiente real as propriedades reveladas pelas simulações.

6.1 A aplicação de testes

Para a realização de testes com a plataforma JOIN, a primeira questão a ser resolvida diz respeito à aplicação que será utilizada nestes testes. Para essa escolha, considera-se que:

1. o grau de paralelismo entre os lotes de tarefas deve ser o maior possível para que a interpretação dos resultados frente ao modelo da aplicação seja facilitada,
2. a relação *tempo de processamento / tempo de comunicação* da aplicação deve poder ser variada em uma ampla faixa de valores,
3. o número de tarefas da aplicação deve poder ser variado numa ampla faixa de valores.

Feitas estas considerações, optou-se pelo desenvolvimento de uma aplicação de busca de números primos segundo o algoritmo da **Peneira de Eratosthenes** [CK96]. Para resolver este problema, a aplicação determina e particiona o espaço de busca de interesse e promove a busca distribuída de números primos dentro destes intervalos. Este tipo de processamento encontra aplicação em esquemas para testar a robustez de sistemas de criptografia de chave pública, como por exemplo o RSA.

Esta aplicação é definida por três lotes de tarefas: o primeiro lote, que terá sempre uma única tarefa, é o responsável pelo particionamento do espaço de busca de números primos; o segundo lote, que pode ter cardinalidade variável, é o lote das tarefas que efetua a busca por números primos dentro de cada um dos intervalos; o terceiro lote, também de cardinalidade 1, é o responsável pela coleta dos resultados gerados pelas tarefas do lote 2.

Esta aplicação atende aos requisitos mencionados anteriormente porque:

Instância	Espaço de Busca ($\times 10^9$)	Tarefas
1	[0..1]	50
2	[0..3]	
3	[0..5]	
4	[0..7]	
5	[0..9]	

Tabela 6.1: Características das aplicações utilizadas nos testes

1. a computação propriamente dita é efetuada exclusivamente pelas tarefas do lote 2, o que significa que esta computação será executada com grau máximo de paralelismo uma vez que não dependerá de resultados previamente computados por outros lotes,

2. a quantidade de computação executada por cada tarefa pode facilmente ser alterada aumentando-se a faixa na qual a busca por números primos será realizada. Isso não implica em aumento na quantidade de comunicação entre tarefas,

3. o número de tarefas pode facilmente ser alterado, aumentando-se ou diminuindo-se o espaço total de busca destinado a cada tarefa. Essa mudança também não influi no tempo de comunicação entre tarefas.

No Apêndice C, é apresentado o código-fonte de cada uma das tarefas desta aplicação.

Cabe aqui relembrar que, em razão do estado de desenvolvimento em que se encontrava a plataofmra JOIN, não foram realizados testes da etapa de balanceamento de carga inter-grupos. Assim, analogamente ao apresentado na seção de simulações intra-grupo, são efetuados testes primeiramente em um único grupo homogêneo e em seguida em um único grupo heterogêneo.

6.2 Testes sobre grupos homogêneos

O grupo homogêneo é formado por 14 computadores do Laboratório de Computação Gráfica da FEEC/Unicamp. Essas máquinas dispõem de processador Pentium IV 1.5 GHz, 512 Mb de memória RAM, sistema operacional Windows 2000 e encontram-se conectadas por uma rede local Ethernet 10 Mbits/s. Durante a realização destes testes nenhum outro processo de usuário, além dos processos da própria plataforma JoiN, foi executado.

Para estes testes, foram geradas 5 instâncias da aplicação de busca de números primos, cada qual parametrizada segundo os valores expressos na Tabela 6.1.

Como pode ser observado, a diferença entre as instâncias reside unicamente na faixa de valores na qual a busca será efetuada; todas as versões executadas possuem 50 tarefas, que recebem, cada uma delas, $1/50$ do espaço total de busca.

Instância	Tempo de Execução (s)
1	75.6
2	223.6
3	362.6
4	512.8
5	656.9

Tabela 6.2: Tempos de execução da versão seqüencial da aplicação de busca de números primos

Além da versão paralela da aplicação de busca de números primos, foi também desenvolvida uma versão sequencial desta aplicação. A execução da versão sequencial da aplicação em uma das UPs do sistema homogêneo forneceu os tempos de execução sequenciais necessários aos cálculos de *speed-up* e eficiência. Esta versão foi executada também em 5 instâncias, parametrizadas identicamente às instâncias geradas para o caso paralelo. Os tempos de execução obtidos podem ser vistos na Tabela 6.2.

6.2.1 Análises preliminares

Antes da apresentação e análise dos tempos de execução obtidos na prática, é possível realizar algumas análises preliminares que facilitarão a avaliação dos resultados obtidos. Tendo-se em mãos os tempos de execução da versão sequencial da aplicação de testes, é possível estimar o tempo ideal de execução de cada uma das tarefas da versão paralela em cada uma das UPs como sendo $1/50$ do tempo total de execução da versão sequencial.

Uma vez que os tempos estimados de execução das tarefas da versão paralela da aplicação tenham sido obtidos, é possível calcular o tempo de execução mínimo de um dado escalonamento desconsiderando-se os tempos de comunicação. Desta forma, na tabela 6.3 são apresentados os escalonamentos de tarefas a serem realizados sobre diversas configurações do grupo homogêneo.

Nesta tabela, cada coluna representa uma das UPs participantes e cada linha representa uma configuração do grupo de UPs. Como pode ser verificado, para cada configuração foram escalonadas 50 tarefas entre as UPs participantes, que é o número de tarefas da versão paralela da aplicação de testes.

Calculados os escalonamentos "ïdeais", assim chamados por terem sido calculados sem se considerar os tempos de comunicação entre UPs, e tendo-se em mãos os tempos de execução de cada tarefa da aplicação de testes, pode-se estimar o tempo ideal de execução da aplicação para cada um dos escalonamentos calculados. A Fig. 6.1 apresenta estes tempos de execução.

Como os tempos de execução apresentados na Fig. 6.1 desconsideram os tempos de comunicação entre UPs, podem então ser tomados como um limite inferior dos tempos possíveis de serem obtidos na prática. Desta forma, é possível calcular as taxas de *speed-up* e eficiência relativas a estes tempos de execução e tomá-las como limitantes superiores de desempenho para os testes reais. Nas Figs. 6.2 e 6.3, são apresentados os gráficos de *speed-up* e eficiência, respectivamente.

Configurações	**Tarefas / UP**													
1	50	-	-	-	-	-	-	-	-	-	-	-	-	-
2	25	25	-	-	-	-	-	-	-	-	-	-	-	-
3	17	17	16	-	-	-	-	-	-	-	-	-	-	-
4	13	13	12	12	-	-	-	-	-	-	-	-	-	-
5	10	10	10	10	10	-	-	-	-	-	-	-	-	-
6	9	8	8	8	8	8	-	-	-	-	-	-	-	-
7	8	7	7	7	7	7	7	-	-	-	-	-	-	-
8	7	7	6	6	6	6	6	6	-	-	-	-	-	-
9	6	6	6	6	6	5	5	5	5	-	-	-	-	-
10	5	5	5	5	5	5	5	5	5	5	-	-	-	-
11	5	5	5	5	5	5	4	4	4	4	4	-	-	-
12	5	5	4	4	4	4	4	4	4	4	4	4	-	-
13	4	4	4	4	4	4	4	4	4	4	4	3	3	-
14	4	4	4	4	4	4	4	4	3	3	3	3	3	3

Tabela 6.3: Escalonamentos para diversas configurações do sistema homogêneo.

As taxas de eficiência apresentadas na Fig. 6.3 foram calculadas como o quociente entre o *speed-up* obtido pelo escalonamento e o *speed-up* ideal. Nestes gráficos, aparece apenas uma única curva porque as taxas de *speed-up* foram idênticas para as 5 instâncias da aplicação. A taxa média de eficiência para os 14 escalonamentos foi de 94,12 %.

Como pode ser observado nestes gráficos, mesmo em condições ideais as taxas de *speed-up* e eficiência não atingem os valores ideais. Isto se deve ao fato da aplicação ter sido dividida num número fixo de tarefas, impossíveis de serem distribuídas equanimente entre as UPs para todas as configurações do grupo. O desbalanceamento de carga resultante desta divisão desigual de tarefas é a causa das quedas de desempenho observadas para quase todas as configurações do grupo. Nota-se aqui que, se o número de tarefas fosse muito superior ao número de UPs, o efeito desta distribuição desigual de tarefas seria minimizado.

6.2.2 Análises sobre os dados reais

Uma vez realizadas estas análises preliminares, são agora apresentados os resultados obtidos nos testes reais. Nestes testes, o conjunto formado pelas 5 instâncias da aplicação paralela foi executado sob diferentes cenários: a primeira bateria de execuções ocorreu em um grupo com apenas duas UPs; deste ponto em diante foram sucessivamente adicionadas duas UPs ao sistema e então repetida a execução do conjunto de 5 instâncias até se chegar ao número máximo de 14 UPs.

Os tempos de execução obtidos podem ser observados na Fig. 6.4.

Neste gráfico, pode-se observar que a evolução dos tempos de execução obtidos é bastante regular e

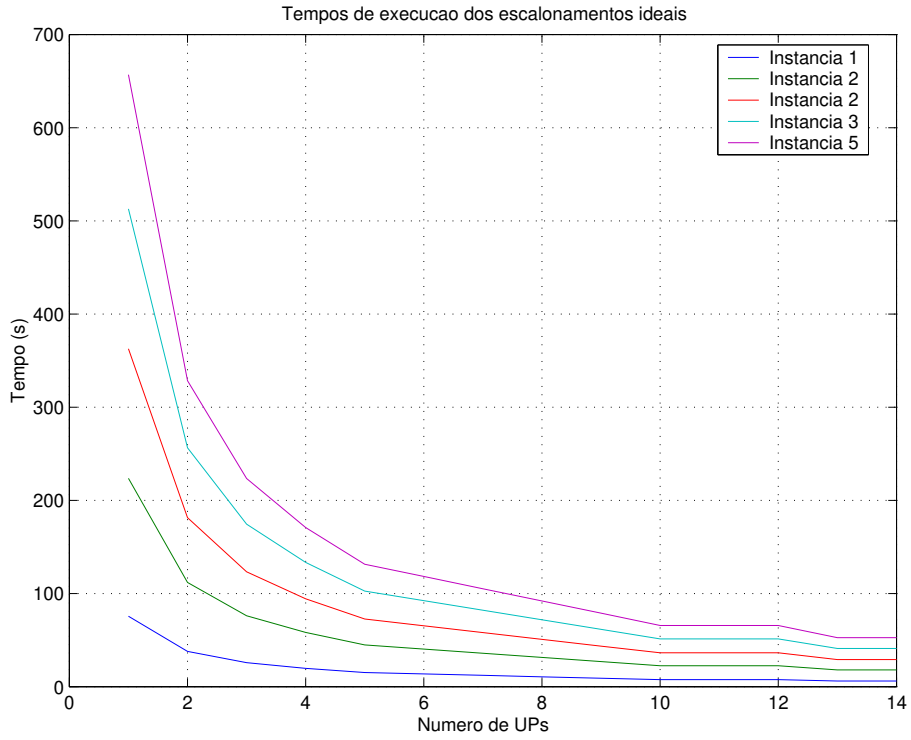

Figura 6.1: Tempos ideais de execução para os escalonamentos apresentados na Tabela 6.3.

previsível. Na Fig. 6.5 são mostradas as curvas de *speed-up* relativas a estes tempos de execução.

Pode-se observar que a evolução das curvas de *speed-up* inicia-se de forma linear mas satura-se na parte final do gráfico. Essa saturação deve-se a dois fatores: o primeiro, já observado no caso ideal, é o desbalanceamento de carga decorrente da divisão da carga computacional de maneira desigual entre as UPs; o segundo deve-se ao fato de que, num sistema homogêneo, todas as suas UPs trabalham num mesmo ritmo e se comunicam com a unidade coordenadora aproximadamente no mesmo instante. Entretanto, na ocorrência de pequenas oscilações nos tempos de execução das tarefas em alguma(s) UP(s), a comunicação entre as UPs e a unidade coordenadora é prejudicada tendo-se em vista a perda de eficiência de comunicação decorrente da colisão de pacotes numa rede padrão Ethernet. Com o aumento no número de UPs, a probabilidade da ocorrência de tais colisões durante a execução de uma aplicação aumenta consideravelmente.

Um outro aspecto importante que pode ser observado é a regularidade na forma das cinco curvas e, como seria de se esperar, o fato das instâncias com menor relação *computação / comunicação* serem também aquelas que resultam em menor *speed-up.*

Finalmente, é possível traçar as curvas de eficiência das execuções realizadas. Essas curvas são apresentadas na Fig. 6.6. Conforme o esperado, essas curvas indicam uma saturação no desempenho das execuções e que se acentua na parte final do gráfico. A eficiência média de todas as execuções realizadas neste sistema

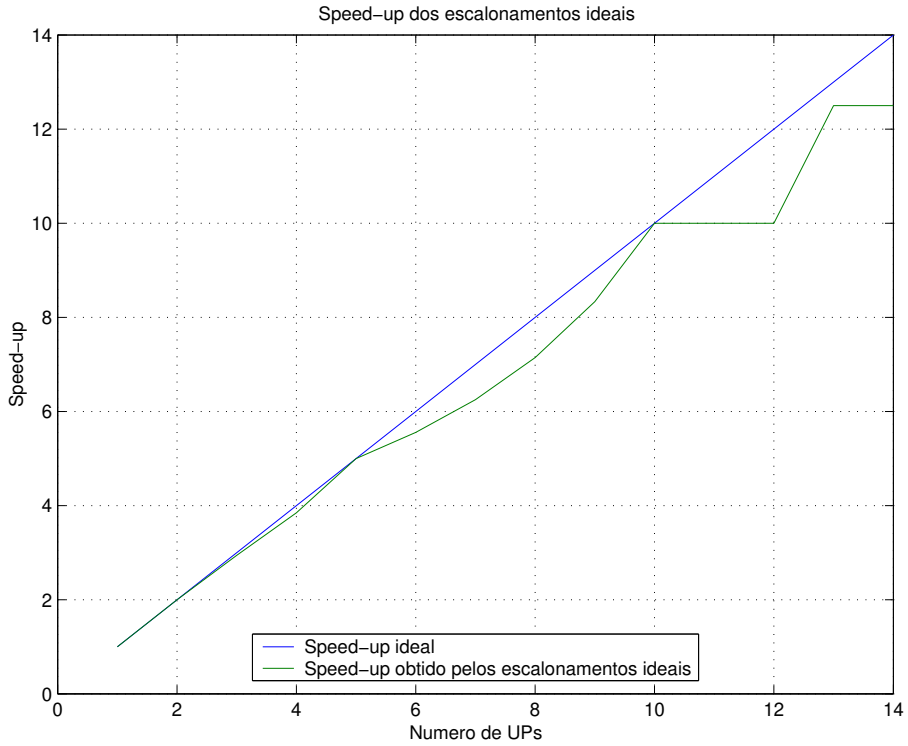

Figura 6.2: Taxas de *speed-up* para os escalonamentos da Tabela 6.3.

homogêneo é igual a 84 %.

A Tabela 6.4 apresenta as taxas médias de eficiência obtidas para cada instância.

A seguir, são apresentados os testes realizados sobre grupos heterogêneos.

6.3 Testes sobre grupos heterogêneos

Para os testes sobre grupos heterogêneos, foram utilizados 14 computadores do Laboratório de Computação e Automação da FEEC/Unicamp. A relação dos computadores participantes com suas características técnicas pode ser vista na Tabela 6.5.

Para que os escalonadores testados pudessem distribuir a carga computacional de maneira a manter o grupo equilibrado, o serviço Gerenciador de Aplicações utilizou um conjunto de *benchmarks* que procurou avaliar o desempenho global dos computadores participantes. Os *benchmarks* utilizados foram os desenvolvidos pelo grupo de pesquisas *Java Grande Forum* (http://www.epcc.ed.ac.uk/javagrande/), que realiza pesquisas em computação científica utilizando a tecnologia Java. A partir dos tempos totais de execução destes *benchmarks* nas UPs, foi possível classificá-las quanto a seus desempenhos relativos, conforme ilustra a segunda coluna da Tabela 6.6.

Instância	Eficiência média
1	76,61 %
2	83,50 %
3	83,74 %
4	87,23 %
5	87,35 %

Tabela 6.4: Taxas médias de eficiência obtidas para os grupos homogêneos.

Nome	**SO**	**Arquitetura**	**Memória (Mb)**
Itapuã	SunOS R 5.7	Ultra SPARC-II 440 Mhz	512
Rocas	SunOS R 5.7	Ultra SPARC-II 360 Mhz	512
Dunas	SunOS R 5.7	Ultra SPARC-II 360 Mhz	256
Juréia	SunOS R 5.7	Ultra SPARC-II 360 Mhz	256
Gorda	SunOS R 5.7	Ultra SPARC-II 360 Mhz	256
Botafogo	SunOS R 5.6	Ultra SPARC-I 200 Mhz	192
Copacabana	SunOS R 5.6	Ultra SPARC-I 167 Mhz	256
Búzios	SunOS R 5.6	Ultra SPARC-I 167 Mhz	256
Grumari	SunOS R 5.6	Ultra SPARC-I 167 Mhz	256
Parati	SunOS R 5.7	SPARCstation 4 143 Mhz	64
Peruibe	SunOS R 5.7	SPARCstation 4 110 Mhz	64
Aracati	SunOS R 5.5.1	SPARCstation 4 110 Mhz	64
Brava	SunOS R 5.7	SPARCstation 4 110 Mhz	32
Caraguá	SunOS R 5.7	SPARCstation 4 110 Mhz	64

Tabela 6.5: Computadores participantes do grupo heterogêneo no qual os testes foram realizados.

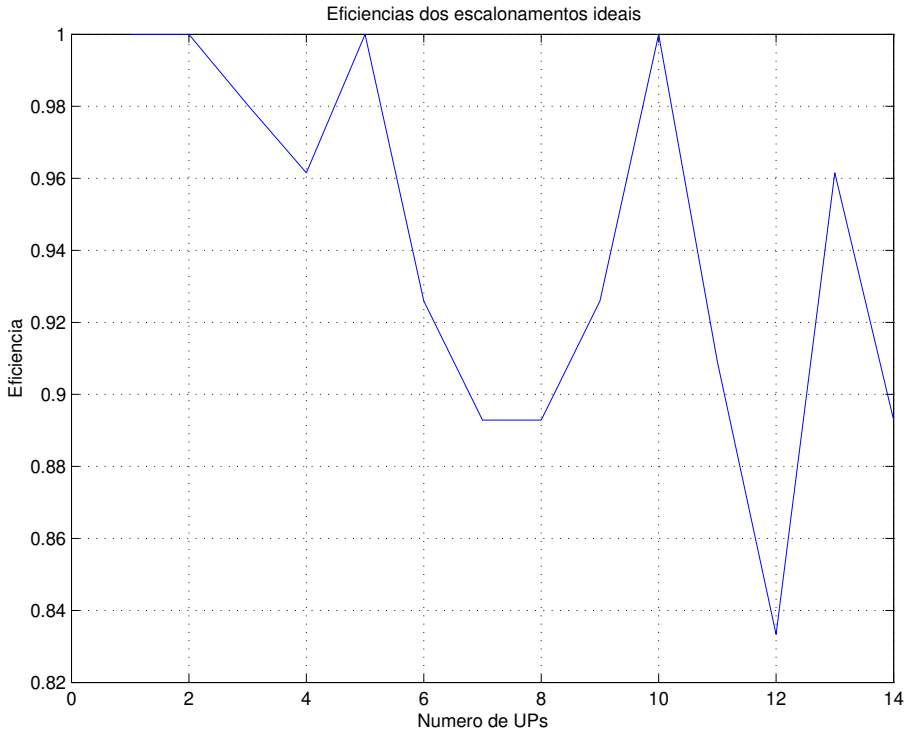

Figura 6.3: Taxas de eficiência para os escalonamentos da Tabela 6.3.

A segunda coluna desta tabela indica que, dentre todas as UPs, aquela que obteve melhor desempenho na execução dos *benchmarks* foi a UP de nome Itapuã. Arbitrando-se a esta UP o fator de desempenho 1,0, foi possível calcular os fatores de desempenho das demais UPs relativos a esta UP. Assim, é possível verificar que, por exemplo, o tempo total de execução dos *benchmarks* na UP Itapuã foi igual a 53 % do tempo de execução dos mesmos *benchmarks* na UP Botafogo.

A terceira coluna da Tabela 6.6 indica os fatores de desempenho reais das UPs. Estes fatores reais foram obtidos executando-se a versão sequencial da aplicação de busca de números primos em cada uma das UPs e calculando-se os quocientes entre os tempos totais de execução na UP de melhor desempenho e nas demais UPs. Na quarta coluna, são apresentadas as diferenças percentuais entre os fatores de desempenho estimados pelos *benchmarks* e os calculados a partir da execução da versão sequencial da aplicação. Como pode ser verificado, estas diferenças, que são na verdade os erros de estimativa de desempenho, são maiores para as UPs de pior desempenho.

Nestes testes, foram utilizadas apenas 3 das 5 instâncias de aplicações utilizadas no caso homogêneo (as de número 1, 3 e 5). Essa simplificação, necessária por razões práticas, não diminuiu a qualidade dos dados gerados dada a linearidade da complexidade desta aplicação dentro da faixa de valores considerados. Também, analogamente ao realizado sobre o grupo homogêneo, as UPs participantes foram incluídas gradativamente no sistema de modo que o impacto da entrada de novas UPs pudesse ser avaliado.

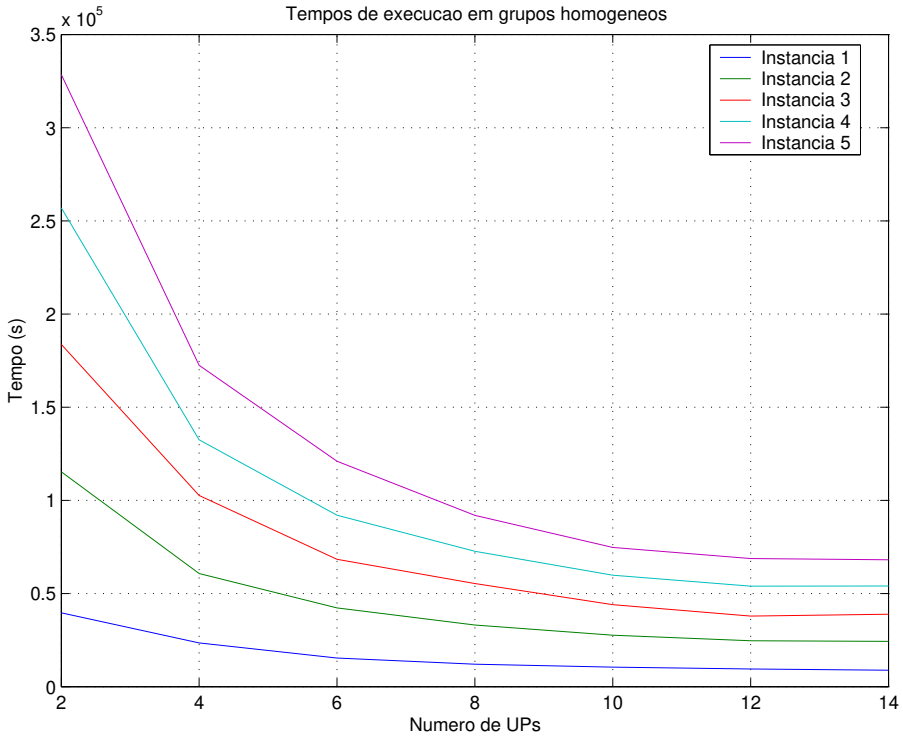

Figura 6.4: Tempos de execução das 5 instâncias da aplicação de busca de números primos com diferentes tamanhos do grupo homogêneo de computadores.

6.3.1 Análises preliminares

Antes da apresentação dos dados obtidos nos testes, algumas análises preliminares podem ser feitas. Como já se conhece o desempenho que cada uma das UPs apresentará durante a execução da aplicação de testes - desempenho este fornecido pelos fatores de desempenho reais - é possível calcular a melhor distribuição de carga possível desta aplicação no grupo de UPs considerado. Este escalonamento ideal servirá como referência para a análise dos resultados obtidos na prática. Na Tabela 6.7, é apresentado o escalonamento ideal para cada uma das 14 situações obtidas pelo acréscimo sucessivo de uma UP ao grupo.

Nesta tabela, cada coluna representa uma das UPs participantes, dispostas na mesma ordem em que aparecem na Tabela 6.6, ou seja, da de melhor a de pior desempenho, e cada linha representa uma configuração do grupo de UPs. Como pode ser verificado, para cada configuração foram escalonadas 50 tarefas entre as UPs participantes, que é o número de tarefas da versão paralela da aplicação de teste. Para o cálculo destes escalonamentos, não foram considerados os tempos de comunicação entre as UPs.

Analogamente ao realizado com os fatores de desempenho reais, na Tabela 6.8 são apresentados os escalonamentos realizados sobre as mesmas configurações do grupo de UPs, mas utilizando-se os fatores de desempenho estimados pelos *benchmarks*, que são os efetivamente utilizados pela plataforma JOIN.

Nome	**Desempenho Estimado**	**Desempenho Real**	**Erro (%)**
Itapuã	1.0	1.0	-
Rocas	0.82	0.81	1,23
Dunas	0.54	0.81	-33,3
Juréia	0.82	0.80	2,50
Gorda	0.67	0.80	-16,2
Botafogo	0.53	0.45	17,7
Copacabana	0.44	0.38	15,7
Búzios	0.25	0.38	-34,2
Grumari	0.44	0.38	15,7
Parati	0.37	0.32	15,6
Peruibe	0.21	0.14	50,0
Aracati	0.21	0.14	50,0
Brava	0.21	0.14	50,0
Caraguá	0.21	0.13	61,5

Tabela 6.6: Fatores de desempenho dos computadores participantes.

Cfg	**Tarefas / UP**													
1	50	-	-	-	-	-	-	-	-	-	-	-	-	-
2	28	22	-	-	-	-	-	-	-	-	-	-	-	-
3	19	16	15	-	-	-	-	-	-	-	-	-	-	-
4	15	12	12	11	-	-	-	-	-	-	-	-	-	-
5	12	10	10	9	9	-	-	-	-	-	-	-	-	-
6	11	9	9	8	8	5	-	-	-	-	-	-	-	-
7	10	8	8	8	8	4	4	-	-	-	-	-	-	-
8	10	8	8	7	7	4	3	3	-	-	-	-	-	-
9	9	7	7	7	7	4	3	3	3	-	-	-	-	-
10	8	7	7	7	7	3	3	3	3	2	-	-	-	-
11	8	7	7	7	6	3	3	3	3	2	1	-	-	-
12	8	7	7	6	6	3	3	3	3	2	1	1	-	-
13	8	7	6	6	6	3	3	3	3	2	1	1	1	-
14	8	6	6	6	6	3	3	3	3	2	1	1	1	1

Tabela 6.7: Escalonamentos calculados utilizando fatores de desempenho reais.

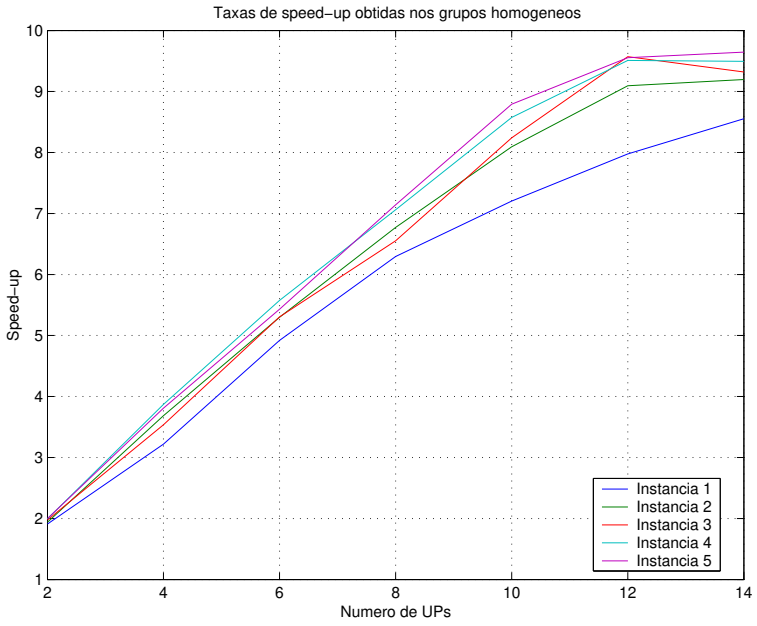

Figura 6.5: Curvas de *speed-up* para os diferentes "tamanhos" do grupo homogêneo.

Como pode ser verificado comparando-se as duas tabelas, as diferenças existentes entre os fatores de desempenho estimados e reais determinam diferenças nos escalonamentos gerados, sobretudo quando são incluídas as UPs que tiveram maior erro de estimativa. Isto significa que os erros nas estimativas do desempenho das UPs fizeram com que o escalonamento efetivamente praticado se diferenciasse do caso ideal provocando assim decréscimos no desempenho global da execução da aplicação.

Para que se possa ter uma idéia melhor do impacto dessas diferenças de escalonamento no desempenho das execuções, na Fig. 6.7 são apresentados os tempos de execução que seriam obtidos aplicando-se estes escalonamentos sobre o grupo real de UPs. Cabe aqui ressaltar que não foram considerados os tempos de comunicação entre as UPs.

Nesta figura, o gráfico da esquerda apresenta os tempos de execução que seriam obtidos pelo melhor escalonamento possível da aplicação de busca de números primos sobre o grupo considerado, enquanto que o gráfico da direita apresenta o melhor escalonamento possível desta aplicação neste grupo, dados os erros nas estimativas de desempenho das UPs. Na verdade, os tempos são maiores já que os tempos de comunicação entre as UPs e as eventuais mudanças de carga nas UPs não foram considerados.

Estes gráficos mostram que os erros nas estimativas de desempenho, maiores nas UPs mais lentas, afetam mais significativamente os tempos de execução quando estas UPs mais lentas são inseridas no grupo de teste. Como o desempenho destas UPs foi superestimado pelos *benchmarks*, são atribuídas a elas cargas desproporcionais a seus poderes computacionais, o que provoca uma piora no desempenho global da execução da aplicação.

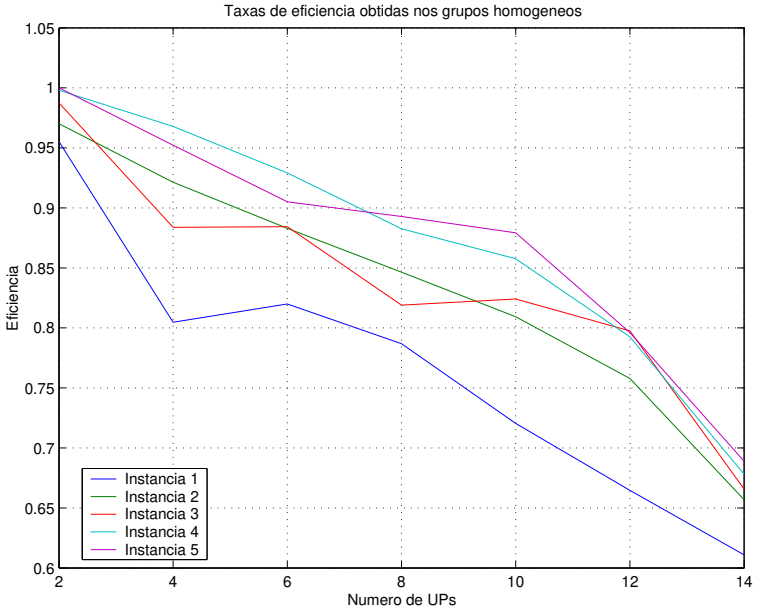

Figura 6.6: Curvas de eficinêcia para as execuções realizadas sobre o grupo homogêneo.

Na Fig. 6.8, são apresentados os gráficos com as taxas de *speed-up* para estes dois casos. Essas taxas de *speed-up* foram calculadas dividindo-se o tempo de execução da versão sequencial da aplicação na UP Itapuã pelos tempos obtidos pelos escalonamentos.

Como o grupo de UPs considerado é heterogêneo, a curva de *speed-up* ideal não é uma reta, mas sim uma curva que representa a somatória dos poderes computacionais das UPs participantes. Assim, o *speed-up* ideal desta aplicação em um grupo que, por exemplo, seja formado pelas UPs Itapuã e Rocas, é de 1,0 + 0,81 = 1,81 do tempo de execução da versão sequencial da aplicação na UP Itapuã. A curva de *speed-up* ideal é também apresentada na figura.

Pode-se constatar que, mesmo para os escalonamentos ideais sem erros de estimativas e sem tempos de comunicação, as taxas de *speed-up* não atingem os valores máximos. Similarmente ao observado nos grupos homogêneos, isto se deve ao fato da aplicação ter sido dividida num número fixo de tarefas cuja distribuição entre as UPs não segue exatamente a mesma relação entre seus poderes computacionais relativos. Assim, a distribuição ideal das 50 tarefas da aplicação num grupo contendo somente as UPs Itapuã e Rocas seria de 27,624 tarefas para a UP Itapuã e 22,376 tarefas para a UP Rocas. Como não existem tarefas fracionárias, foram atribuídas 28 tarefas à UP Itapuã e 22 à UP Rocas.

Por fim, na Fig. 6.9 são apresentadas as taxas de eficiência obtidas para os escalonamentos calculados. Estas taxas de eficiência foram calculadas dividindo-se as taxas de *speed-up* obtidas pelas taxas de *speed-up* ideais.

Percebe-se que os erros nas estimativas dos poderes computacionais das UPs afetam grandemente as

Cfg	Tarefas / UP													
1	50	-	-	-	-	-	-	-	-	-	-	-	-	-
2	27	23	-	-	-	-	-	-	-	-	-	-	-	-
3	21	18	11	-	-	-	-	-	-	-	-	-	-	-
4	16	13	8	13	-	-	-	-	-	-	-	-	-	-
5	13	11	17	11	8	-	-	-	-	-	-	-	-	-
6	12	9	6	9	7	6	-	-	-	-	-	-	-	-
7	11	9	5	9	7	5	4	-	-	-	-	-	-	-
8	10	9	5	8	6	5	4	2	-	-	-	-	-	-
9	9	8	5	7	6	5	4	2	4	-	-	-	-	-
10	9	7	4	7	6	4	4	2	4	3	-	-	-	-
11	9	7	4	7	6	4	4	2	3	3	1	-	-	-
12	9	7	4	7	6	4	2	2	3	3	1	1	-	-
13	8	7	4	7	6	4	3	2	3	3	1	1	1	-
14	8	7	4	7	5	4	3	2	3	3	1	1	1	1

Tabela 6.8: Escalonamentos calculados utilizando fatores de desempenho estimados.

taxas de eficiência. As taxas médias de eficiência obtidas são apresentadas na Tabela 6.9.

6.3.2 Análises sobre os dados reais

Feitas estas análises preliminares, é possível agora apresentar e analisar os resultados obtidos em testes reais com a plataforma JOIN. Diferentemente do realizado na seção anterior, nesta etapa foram utilizados vários esquemas de escalonamento: assim, foram aplicados os escalonadores estático, GS e GSTR, cada qual com os algoritmos de escalonamento trivial e *best-fit*, perfazendo um total de 6 esquemas de escalonamento.

As figuras 6.10, 6.11 e 6.12 apresentam os tempos de execução obtidos com os escalonamentos estático, GS e GSTR para cada uma das instâncias da aplicação, respectivamente.

Fatores de Desempenho	Instância	Eficiência média
Reais	1	91,15 %
	3	90,12 %
	5	90,95 %
Estimados	1	80,13 %
	3	79,02 %
	5	79,90 %

Tabela 6.9: Taxas médias de eficiência obtidas para os escalonamentos ideais.

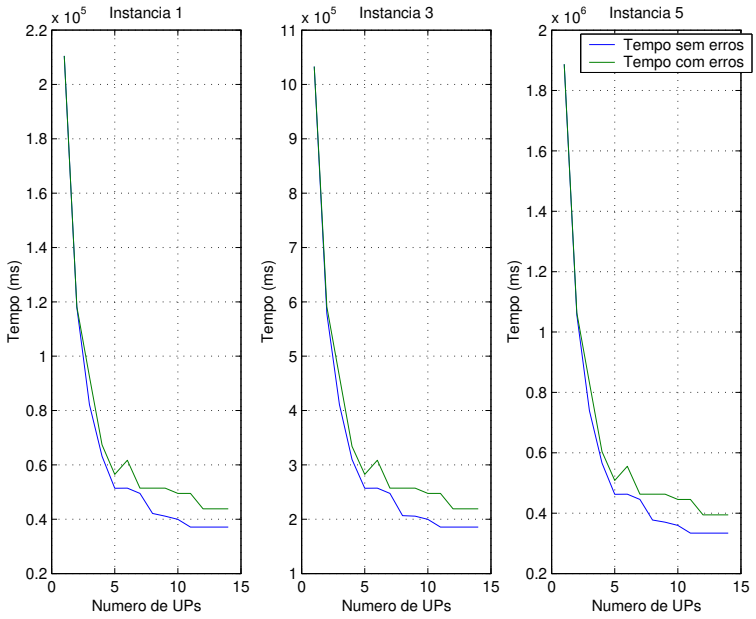

Figura 6.7: Tempos de execução calculados para os escalonamentos ideais, com e sem erros de estimativa.

Para todas as três instâncias da aplicação de teste, o comportamento das curvas de tempo é bastante similar. Pode-se observar nestes gráficos que, quando se utiliza o algoritmo de escalonamento trivial, que atribui a mesma carga computacional a todas as UPs, um considerável decréscimo de desempenho é observado ao se incluir as UPs mais lentas no grupo testado. Este decréscimo de desempenho é atenuado pelos escalonadores GS e GSTR em virtude de suas capacidades de se adaptarem dinamicamente à resposta das UPs, mas o GSTR é visivelmente mais robusto à heterogeneidade do grupo que o GS.

Ao se utilizar o algoritmo de escalonamento *best-fit*, o comportamento das curvas de tempo é consideravelmente mais regular que o das curvas obtidas com o algoritmo trivial. Além disso, nestas curvas não se observa o mesmo decréscimo de desempenho observado nos testes com as UPs mais lentas e o algoritmo trivial. Isto se explica pelo fato das UPs mais lentas receberem menos tarefas que as mais rápidas, o que implica em um balanceamento melhor de carga no grupo.

Um ponto a ser destacado nos resultados obtidos com o algoritmo *best-fit* é o fato do escalonador GSTR apresentar tempos maiores para algumas configurações do grupo de UPs. Isto se deve ao fato de que num grupo onde o desempenho das UPs participantes tenha sido aferido com precisão, os escalonadores estático e GS tendem a fornecer soluções melhores uma vez que a replicação de tarefas nestes casos dificilmente proporciona melhoras de desempenho. De fato, a replicação de tarefas é interessante sob o ponto de vista do desempenho somente em grupos onde os desempenhos das UPs não sejam conhecidos ou onde o erro nas estimativas destes desempenhos estejam acima de um determinado patamar. Fora destes casos, a replicação de tarefas sobrecarrega as UPs e desequilibra suas cargas.

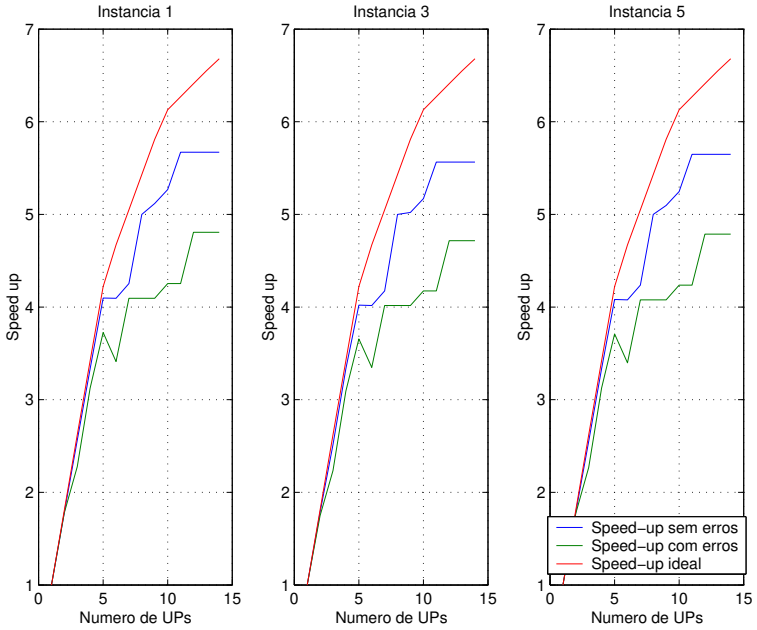

Figura 6.8: Taxas de *speed-up* calculadas para os escalonamentos ideais, com e sem erros de estimativa.

As figuras 6.13, 6.14 e 6.15 apresentam as taxas de *speed-up* relativas aos resultados obtidos. Essas taxas de *speed-up* foram calculadas dividindo-se os tempos de execução da versão sequencial da aplicação de testes na UP Itapuã pelos tempos de execução da versão paralela da aplicação.

Analogamente ao verificado nos gráficos com os tempos de execução obtidos, observa-se grande similaridade no comportamento das curvas de *speed-up* para as três instâncias da aplicação. Observa-se ainda que a queda de desempenho decorrente da aplicação do algoritmo trivial se traduz em uma acentuada queda de *speed-up*. Por fim, fica claro nestes gráficos que o único escalonador que indica um comportamento escalável, isto é, que não se satura com o aumento de UPs, é o GSTR.

Finalmente, nas figuras 6.16, 6.17 e 6.18 são apresentados os gráficos de eficiência. Essas taxas de eficiência foram obtidas dividindo-se o *speed-up* efetivamente obtido pelo *speed-up* ideal.

Na Tab. 6.10, são apresentadas as taxas médias de eficiência para cada caso testado.

6.4 Resumo dos resultados dos testes

Observando-se os resultados obtidos nos testes realizados, tanto sobre os grupos homogêneos quanto sobre os heterogêneos, pode-se dizer que, em linhas gerais, o comportamento apresentado pelo escalonador GSTR está de acordo com os requisitos que orientaram sua concepção; o escalonador GSTR mostrou-se escalável, não saturando-se com o aumento no número de UPs, e bem adaptável aos sistemas heterogêneos em que havia erros acentuados nas estimativas de desempenho das UPs.

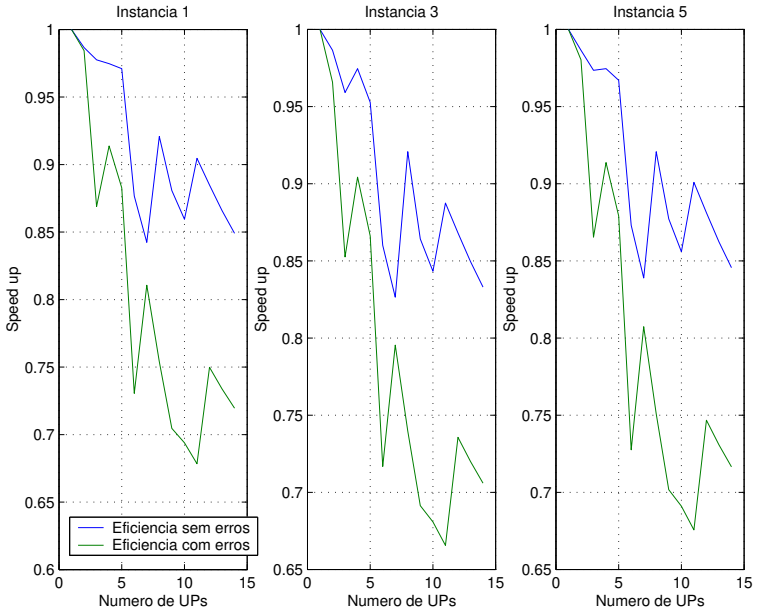

Figura 6.9: Taxas de eficiência calculadas para os escalonamentos ideais, com e sem erros de estimativa.

Cabe aqui destacar que o escopo dos testes foi bastante distante ao das simulações. Isso se deveu basicamente aos seguintes pontos.

1. Foi utilizada apenas uma aplicação com número de lotes e tarefas fixo,
2. Não foi provocada nenhuma variação significativa na carga das UPs participantes, o que significa dizer que os erros nas estimativas de desempenho foram aproximadamente fixos e iguais para todos os esquemas de escalonamento.
3. Não foi provocado nenhum tipo de falha em nenhuma UP.

Algoritmo	Instância	Escalonador		
		Estático	GS	GSTR
Trivial	1	61,70 %	64,12 %	73,33 %
	2	64,30 %	66,29 %	71,18 %
	3	64,40 %	66,47 %	77,53 %
Best-fit	1	74,47 %	75,77 %	74,91 %
	2	75,69 %	77,40 %	77,41 %
	3	76,61 %	78,89 %	80,75 %

Tabela 6.10: Taxas médias de eficiência obtidas para cada esquema de escalonamento.

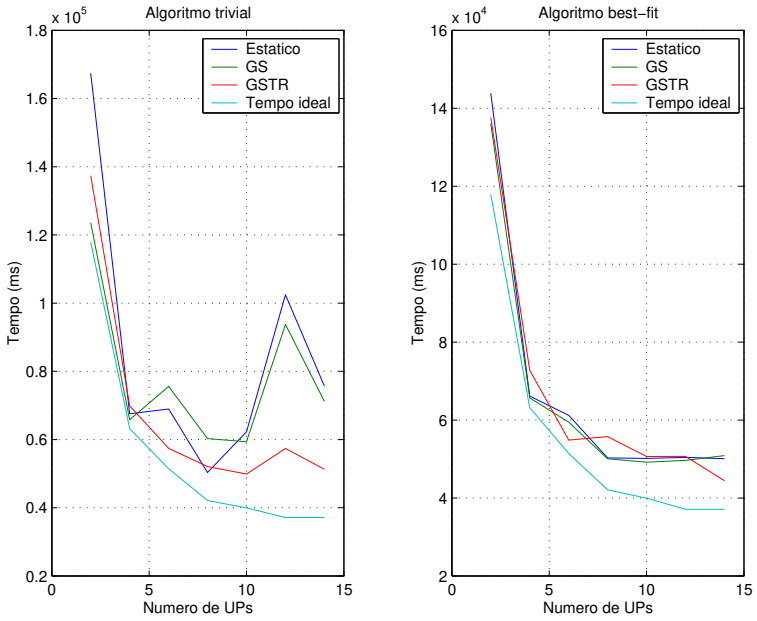

Figura 6.10: Tempos de execução obtidos para a instância 1.

Dadas essas considerações, foi possível verificar que:

1. a utilização do algoritmo de escalonamento *best-fit* proporcionou menor variabilidade ou, em outras palavras, maior regularidade nos tempos de execução e melhor desempenho global;

2. o escalonador GSTR apresentou melhor desempenho global que os demais, sobretudo à medida que o sistema tornou-se mais heterogêneo;

3. o escalonador GSTR mostrou-se bastante robusto à introdução de heterogeneidade no sistema.

6.5 Conclusões

Neste capítulo foram apresentados alguns testes que procuraram validar as conclusões obtidas no capítulo de simulações (Cap. 5).

Apesar de alguns dos outros esquemas de escalonamento terem apresentado resultados ligeiramente melhores em sistemas pouco heterogêneos, cabe aqui destacar que dentro do domínio de interesse deste trabalho, que é o dos computadores virtuais maciçamente paralelos (MPVCs), onde a heterogeneidade e a possibilidade de falhas são aspectos fundamentais, o esquema de escalonamento proposto destacou-se claramente frente aos demais.

Em face destes resultados, consideramos plenamente satisfatória a solução encontrada para o problema inicialmente proposto de balanceamento de carga.

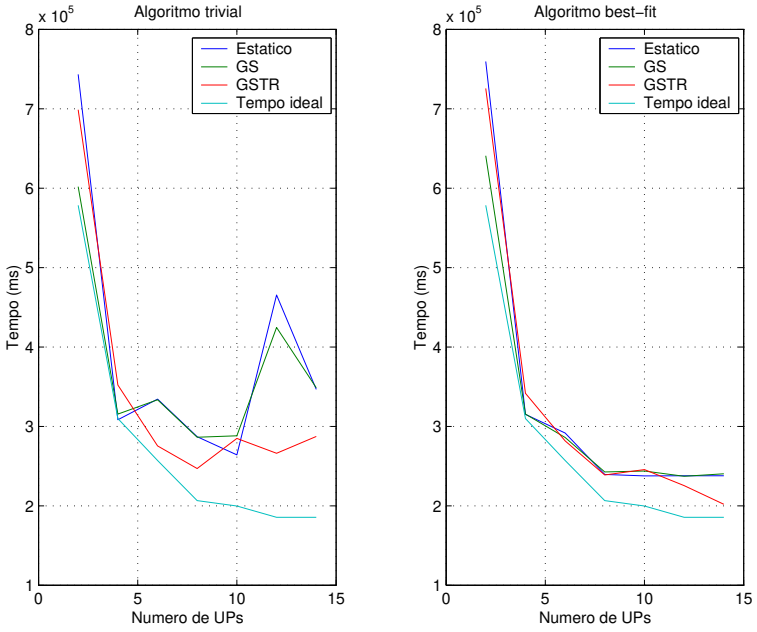

Figura 6.11: Tempos de execução obtidos para a instância 3.

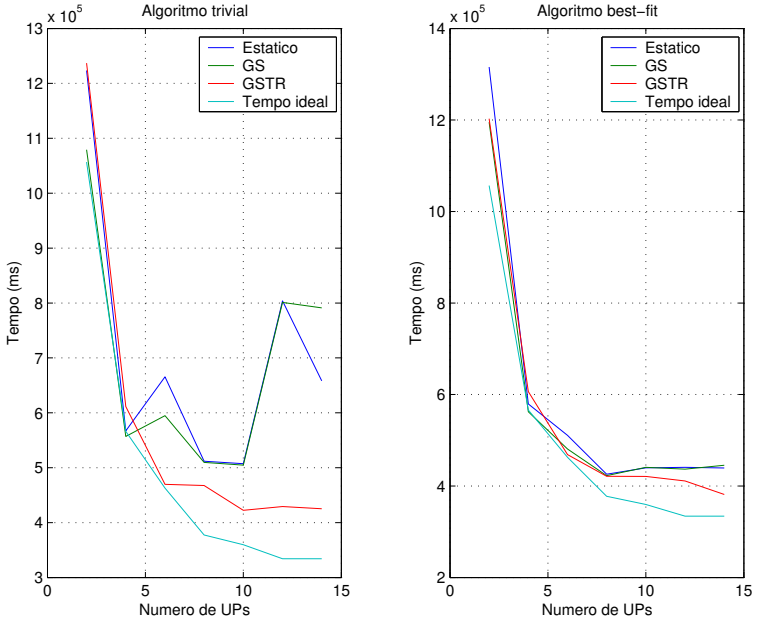

Figura 6.12: Tempos de execução obtidos para a instância 5.

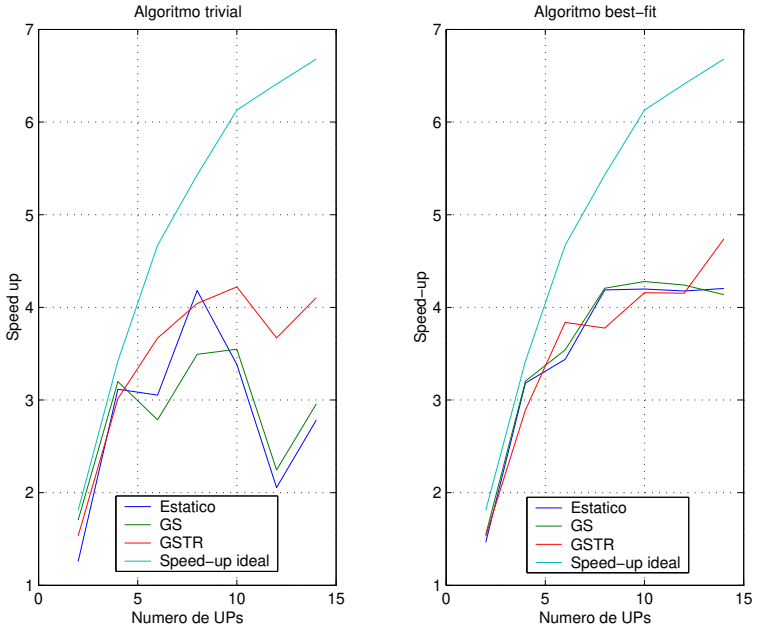

Figura 6.13: Taxas de *speed-up* obtidas para a instância 1.

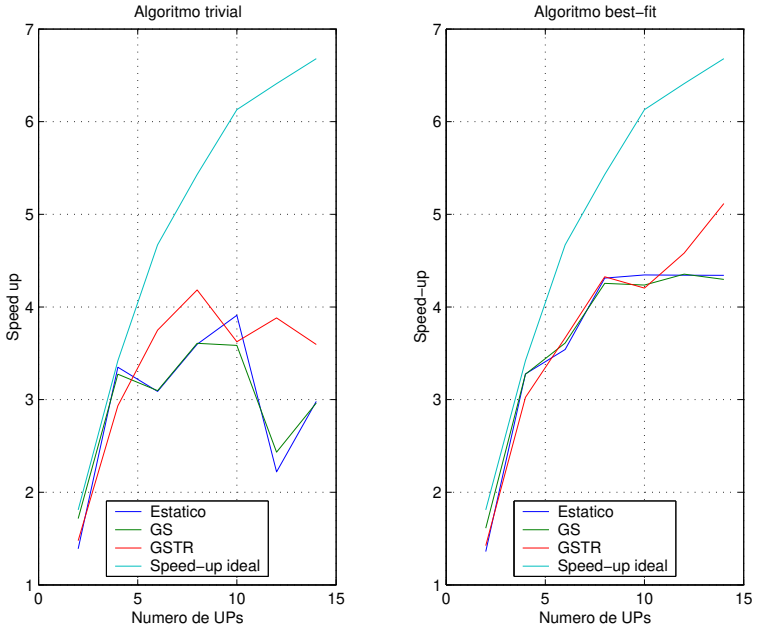

Figura 6.14: Taxas de *speed-up* obtidas para a instância 3.

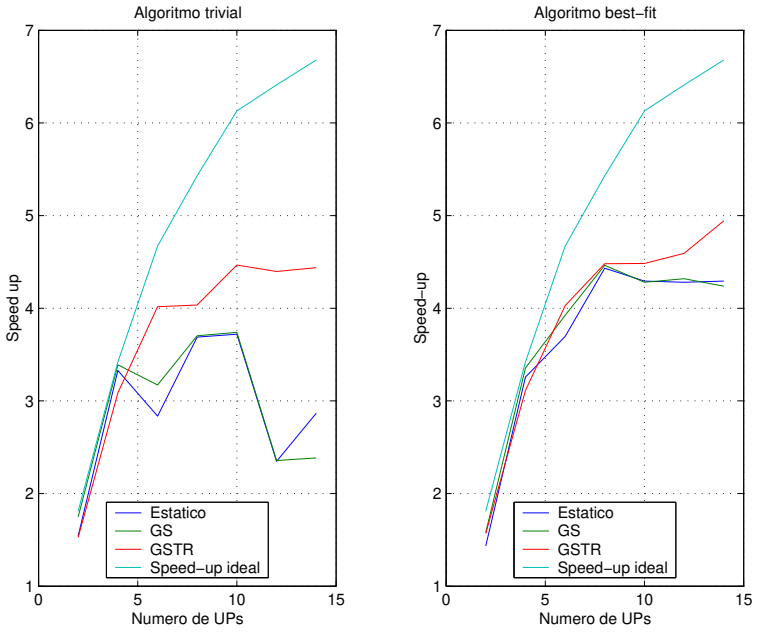

Figura 6.15: Taxas de *speed-up* obtidas para a instância 5.

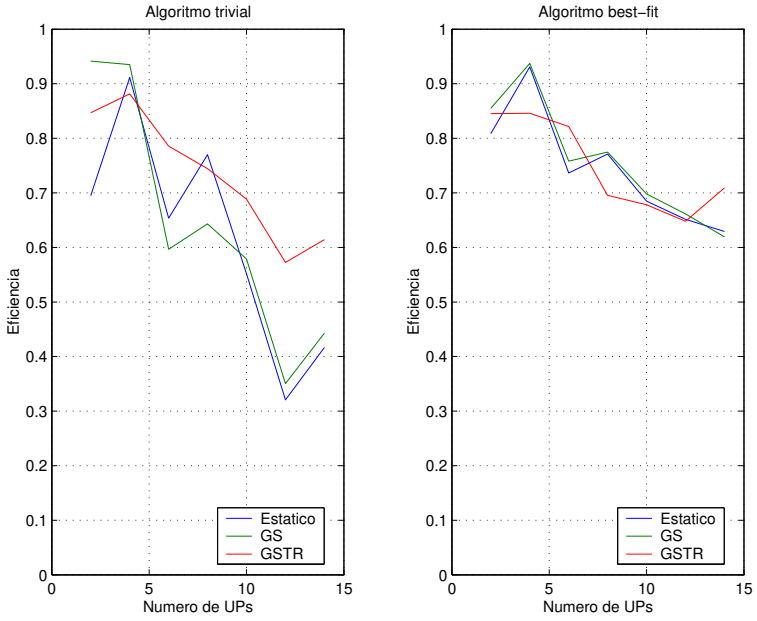

Figura 6.16: Taxas de eficiência obtidas para a instância 1.

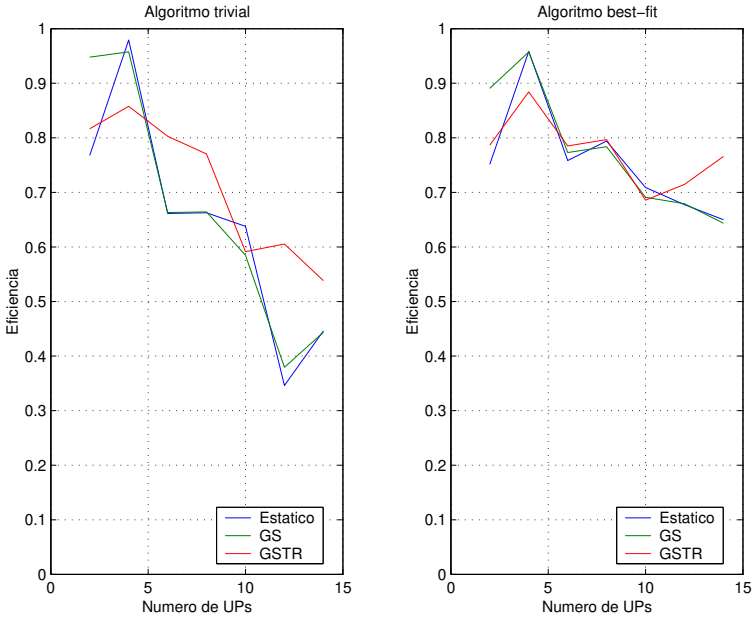

Figura 6.17: Taxas de eficiência obtidas para a instância 3.

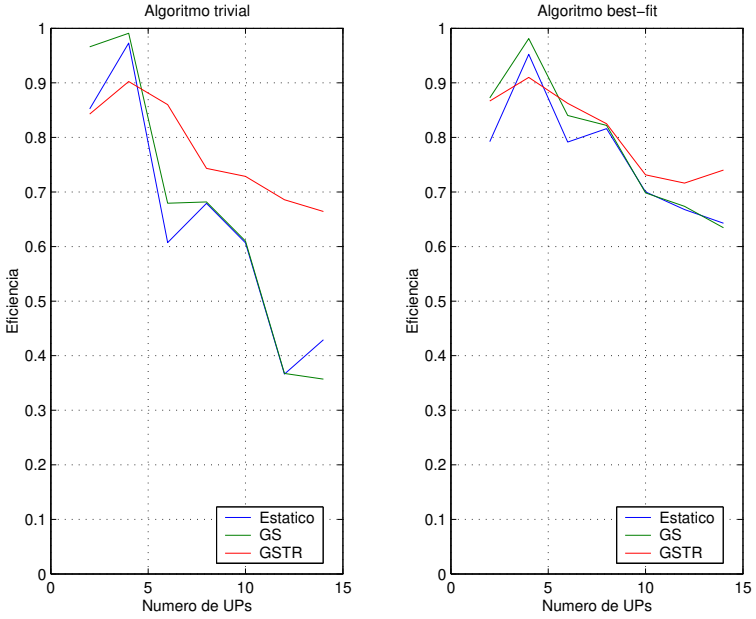

Figura 6.18: Taxas de eficiência obtidas para a instância 5.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Conclusões

Neste trabalho foi abordado um dos principais problemas relacionados ao projeto de computadores virtuais maciçamente paralelos (MPVCs), que diz respeito ao balanceamento de carga computacional entre as unidades de processamento integrantes deste sistema. Nesta abordagem, foram introduzidas definições teóricas que procuraram localizar o domínio exato de interesse do trabalho para que, em seguida, fosse proposto um esquema de balanceamento de carga próprio para o domínio estabelecido.

A solução proposta, que procurou basear-se em alguns dos mais comuns paradigmas de balanceamento de carga em sistemas paralelos e distribuídos, tem em seu núcleo um escalonador de tarefas intitulado "*Escalonador Geracional com Replicação de Tarefas*", ou *GSTR*, que, como o próprio nome procura indicar, vale-se do conceito de replicação de tarefas para atender aos requisitos impostos pelo modelo de referência.

No Capítulo 4, foi visto como os algoritmos de balanceamento GSTR e PPCP e a linguagem PASL foram implementados no sistema de computação maciçamente paralela JOIN.

Finalmente, nos Capítulos 5 e 6 foram apresentadas simulações e testes que procuraram validar a adequação da solução proposta ao domínio de interesse deste trabalho. Conforme pôde ser verificado, os resultados mostraram que, em ambientes caracterizados pela heterogeneidade das unidades de processamento e onde os erros de estimativa de desempenho destas unidades foram significativos, os escalonadores GSTR e PPCP apresentaram performance significativamente superior à dos demais escalonadores considerados, O escalonador GSTR ainda possui a vantagem única de ser tolerante a falhas nas unidades de processamento, um dos principais requisitos dos escalonadores de tarefas em MPVCs.

7.2 Trabalhos futuros

Este trabalho não tem a pretensão de propor uma solução definitiva para o problema de escalonamento de tarefas nestes sistemas paralelos. Algumas importantes melhorias poderiam ser obtidas através do

aperfeiçoamento de módulos da solução proposta, assim como novas propostas podem surgir baseadas nos conceitos introduzidos neste trabalho.

Dentre os principais pontos que podem ser desenvolvidos futuramente, destacam-se os seguintes.

1. Teste do algoritmo de particionamento de aplicações entre grupos (PPCP) na plataforma JOIN.
2. Estudo da possibilidade de utilização dos dados obtidos nos *benchmarks* no particionamento da aplicação entre grupos pelo algoritmo PPCP.
3. Modificação da implementação em JOIN do algoritmo GSTR para que tarefas isoladas que já tenham seus dados de entrada disponíveis sejam iniciadas. Atualmente, as tarefas de um lote são iniciadas somente quando os dados de entrada de todas as tarefas deste lote já se encontram disponíveis.
4. Utilização do serviço gerenciador de arquivos para armazenamento em meio estável dos resultados gerados por todas as tarefas de uma aplicação, o que possibilitará a retomada da execução desta aplicação do ponto em que parou caso haja uma falha no coordenador.
5. Inclusão de mecanismos para o salvamento do estado dos componentes servidor e coordenador dos serviços gerenciador de aplicações e escalonador, para a recuperação de falhas.
6. Introdução de um mecanismo para a execução de aplicações com diferentes prioridades.
7. Extensão da linguagem PASL para a inclusão de instruções de desvio condicional e para a especificação de números variáveis de tarefas nos lotes.
8. Estudo do impacto do limiar de escalonamento estático (da terceira etapa do escalonamento GSTR) no tempo total de escalonamento.
9. Estudo do impacto da utilização de diferentes prioridades na execução de tarefas originais e replicadas.
10. Inclusão de mecanismos inteligentes que detectem a necessidade de escalonamento com ou sem replicação de tarefas.

Bibliografia

[ADJ+91] Anant Agarwal, Geoffrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Daniel Nussbaum, Mike Parkin, and Donald Yeung. The MIT alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[AK94] L. Almand and Y. K. Kwok. A new approach to scheduling parallel programs using task duplication. In *International Conference on Parallel Processing*, volume II, pages 47–51, 1994.

[Bac89] D. Fernandez Baca. Allocation modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, SE=15(11):1427–1436, 1989.

[BAG00] Rakumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G, an architecture for resource management and scheduling system in a global computational grid. *Proceedings of the HPC ASIA 2000, the* 4^{th} *International Conference on High Performance Computing*, 2000.

[BAR93] Guday S. Barak A. and Wheeler R. *The Mosix Distributed Operating System: Load Balancing for Unix*. Springer-Verlag, 1993.

[BL02] Christopher A. Bohn and Gary B. Lamont. Load balancing for heterogeneous clusters of PCs. *Future Generation Computer Systems*, 18:389–400, 2002.

[BWL00] Edward K. Blum, Xin Wang, and Patrick Leung. Architectures and message-passing algorithms for cluster computing: Design and performance. *Parallel Computing*, 26:313–332, 2000.

[CK96] J. H. Conway and Guy R. K. *The Book of Numbers*. Springer-Verlag, 1996.

[CkkG99] Steve Chapin, Dimitrios katramatos, John karpovich, and Andrew Grimshaw. The legion resource management system. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing, Puerto Rico*, 1999.

[CWF+98] Brent R. Carter, Daniel W. Watson, Freund Richard F., Keith Elaine, Mirabile Francesca, and Howard Jay Siegel. Generational scheduling for dynamic task management in heterogeneous computing systems. *Journal of Information Sciences*, 106:219–236, 1998.

[DL94] K. Decker and V. R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, pages 65–84, 1994.

[Ent02] The Entropia Project. http://www.entropia.com, último acesso em 12/2002.

[FK97] I. Foster and C. Kesselman. A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[FKT96] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[Fro02] United Devices' Frontier Project. http://www.ud.com, último acesso em 12/2002.

[GGV96] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Usenix Association Second Symposium on Operating Systems Design*, pages 107–121, 1996.

[GWT97] A. S. Grimshaw, A. Wm, and The Legion Team. The legion vision of a worldwide computer. *Communications of the ACM*, 40(1), 1997.

[HB99] Y. F. Hu and R. J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999.

[HC97] C. C. Hui and S. T. Chanson. Theoretical analysis of the heterogeneous dynamic load balancing problem using a hydro-dynamic approach. *Journal of Parallel and Distributed Computing*, 43:139–146, 1997.

[Hen99] Marco A. Amaral Henriques. A proposal for a Java based massively parallel processing on the Web. In *Proceedings of The First Annual Workshop on Java for High-Performance Computing*, pages 56–66, 1999.

[HSLL99] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptative scheduling for master-worker applications on the computational grid. In *Proceedings of the First IEEE/ACM Internation Workhop on Grid Computing*, Bangalore, India, 1999.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[KA99] P. Kakulavarapu and J. Amaral. A survey of load balancers in modern multi-threading systems. In *Proc. of the 11th Symp. on Computer Architecture and High Performance Computing*, pages 10–16, 1999.

[KTF03] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, a ser publicado em 2003.

[LH00] Fabiano de O. Lucchese and Marco A. A. Henriques. Sequenciamento de cadeias de DNA em ambiente de computação paralelo virtual baseado na internet. In *Anais do Workcomp 2000, Workshop em Computação*, pages 33–38. Instituto Tecnológico de Aeronáutica, 2000.

[LWY95] Cheolwhan Lee, Yuang-Fang Wang, and Tao Yang. Static global scheduling for optimal computer vision and image processing operations on distributed-memory multiprocessors. In *Computer Analysis of Images and Patterns*, pages 920–925, 1995.

[Mic02] Sun Microsystems. The Java Virtual Machine Specification. Documentação on-line: http://java.sun.com/docs/books/vmspec/index.html, 2002.

[MSd+95] Daniel A. Menasce, Debanjan Saha, Stella C. da Silva Porto, Virgilio A. F. Almeida, and Satish K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[Par02] The Parabon Project. http://www.parabon.com, último acesso em 12/2002.

[Pop02] Popular Power Project. http://www.popularpower.com, último acesso em 12/2002.

[Pri94] Pritchard. Improved incremental prime number sieves. In *ANTS: 1st International Algorithmic Number Theory Symposium (ANTS)*, 1994.

[SET02] SETI@Home Project. http://www.seti.org, último acesso em 12/2002.

[Sha01] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California at San Diego, May 2001.

[Sil91] Robert D. Silverman. Massively distributed computing and factoring large integers. *Communications of the ACM*, 34:94–103, 1991.

[Str93] V. Strumpen. Parallel molecular sequence analysis on workstations in the internet. Technical report, Department of computer science, University of Zurich, 1993.

[Sun90] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[TOK98] Tatsuhiro Tsuchiya, Tetsuya Osada, and Tohru Kikuno. Genetics-based multiprocessor scheduling using task duplication. *Microprocessors and Microsystems*, 22:197–207, 1998.

[VG96] Arie Jan Cornelis Van Gemund. *Performance Modeling of Parallel Systems.* PhD thesis, Delft University - Netherlands, 1996.

[vLFGL01] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[Wri01] Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, Champaign - Urbana, IL, June 2001.

[WTH95] Thomas M. Warschko, Walter F. Tichy, and Christian G. Herter. Efficient parallel computing on workstation clusters. *PARS (GI) Mitteilungen*, 14:49–57, 1995.

[Yer98] Eduardo Javier Huerta Yero. Um sistema para o processamento massivamente paralelo na World Wide Web. Master's thesis, Faculdade de Engenharia Elétrica e de Computação - Unicamp, 1998.

Apêndice A

Modificações realizadas na plataforma JOIN

Durante a realização deste trabalho, uma série de atualizações no código da plataforma JOIN se fizeram necessárias. Neste apêndice, serão descritas as principais modificações realizadas procurando-se enfatizar as razões que motivaram as mesmas e o impacto delas na estruturra básica da plataforma.

A.1 O núcleo do JOIN

Durante o período no qual o sistema de balanceamento de carga foi desenvolvido - e juntamente com ele uma série de outros sistemas - o código básico da plataforma JOIN passou por uma grande reforma. A necessidade de parte dessas mudanças decorreu do fato de um sistema de balancemaneto de carga, tal como havia sido delineado, necessitar de informações privilegiadas a respeito do funcionamento da plataforma, somente disponíveis às camadas mais internas do núcleo. Por outro lado, inicialmente o modelo interno da plataforma seguia o paradigma do **núcleo monolítico**, onde o conjunto de funcionalidades básicas do sistema é agrupado em um único bloco hermético. Desta forma, a introdução do mecanismo de balanceamento implicaria na modificação do núcleo da plataforma, o que não era desejável tendo-se em vista o papel crítico desempenhado pelo sistema.

Outros mecanismos igualmente importantes ao funcionamento da plataforma - tais como os de tolerância a falhas e segurança - e que também encontravam-se em fase de codificação, apresentavam as mesmas dificuldades de projeto. Nestas condições, uma reavaliação de todo o código da plataforma foi feita e, após avaliar os prós e contras, decidiu-se que o mais adequado seria alterar este código básico de modo a torná-lo mais simples e flexível.

O novo modelo adotado procurou reduzir ao máximo as atribuições do núcleo da plataforma, deixando a cargo de módulos independentes a implementação das várias funcionalidades necessárias ao sistema. Esses módulos independentes, denominados **módulos de serviço**, ou simplesmente **serviços**, encarregaram-se

então de prover à plataforma funcionalidades de comunicação, gerenciamento de arquivos, gerenciamento de identificadores de tarefas, gerenciamento de eventos e todas as demais funcionalidades que se fizessem necessárias. Neste contexto, o antigo micro-núcleo foi reduzido a um simples **gerenciador de serviços**, cuja função era prover todos os mecanismos necessários ao carregamento, execução e utilização dos módulos de serviço.

Cabe aqui ressaltar que as últimas versões da plataforma já possuiam um mecanismo de manipulação de módulos de serviço semelhante ao descrito no parágrafo anterior. Entretanto, nestas versões nenhum dos módulos de serviços era essencial ao funcionamento da plataforma já que todas as funcionalidades básicas haviam sido incluídas em seu núcleo. A estratégia consistiu portanto em transformar estas funcionalidades básicas, até então contidas no núcleo, em serviços implementados por módulos independentes.

Os principais impactos positivos decorrentes desta mudança de paradigma foram:

1. portabilidade: a simplicidade do gerenciador de serviços lhe conferiu grande versatilidade, permitindo que este pudesse ser utilizado em qualquer um dos componentes da plataforma (servidor, coordenadores, trabalhadores e jacks).

2. facilidade de manutenção: a simplificação drástica do núcleo da plataforma tornou-o mais fácil de ser entendido, manipulado e, eventualmente, modificado. Também conferiu maior estabilidade e eficiência à plataforma.

3. programabilidade: a adoção de uma estrutura bem definida, como a dos módulos de serviço, para a implementação das várias funcionalidades da plataforma, introduziu uma disciplina de desenvolvimento que melhorou a qualidade dos softwares desenvolvidos para a plataforma.

Na seção a seguir será apresentada uma descrição mais detalhada do funcionamento do gerenciador de serviços.

A.2 O gerenciador de serviços

Conforme descrito anteriormente, o gerenciador de serviços tem por objetivo permitir que os módulos de serviço possam ser carregados e operados adequadamente. Para isso, ele deve prover mecanismos que permitam que:

- a plataforma possa ser facilmente configurada através da especificação de quais serviços serão inicialmente carregados e executados,

- os módulos de serviço possam interagir de forma segura através da introdução de mecanismos de controle de acesso,

- as interações entre módulos de serviço não gerem inconsistências em seus estados internos,

- as interações inter-modulares não criem condições de corrida entre módulos, o que potencialmente poderiam causar *deadlocks* na plataforma.

A primeira condição já era satisfeita pelo antigo gerenciador de serviços. Para tal, este se valia da possibilidade de se promover, na linguagem Java, o carregamento de classes de software em tempo de execução a partir de uma cadeia de caracteres contendo o nome destas classes. Desta forma, arquivos de configuração em formato de texto simples podiam ser editados para que o administrador do sistema especificasse quais módulos deveriam ser carregados durante a inicialização da plataforma.

Para que a segunda condição fosse satisfeita, um mecanismo disponibilizado a partir da versão 1.3 do Sun JDK foi de fundamental importância: o *proxy* **de classes**. Este mecanismo permite que chamadas a métodos de uma determinada classe sejam interceptados para que um tratamento qualquer seja realizado antes da invocação efetiva do método em questão. Desta forma, verificações de permissão de acesso podem ser feitas quando serviços realizam chamadas entre si, garantido assim que serviços que desempenham tarefas críticas tenham seu acesso restringido. Neste caso, cabe a um serviço de segurança a definição da política de acesso entre serviços.

A aplicação dos *proxies* de classes aos serviços do JOIN também permitiu que outras tarefas, até então difíceis de serem realizadas, fossem automaticamente introduzidas na plataforma. Uma delas é a de *class profiling*, que consiste basicamente na avaliação da frequência com que os métodos de uma determinada classe são invocados e qual o tempo médio dispendido na execução destes métodos. Trata-se portanto de uma poderosa ferramenta para a avaliação da performance dos serviços e seus graus de interdependência.

A terceira condição pôde ser garantida introduzindo-se pontos de sincronização entre os módulos de serviço de modo a garantir que a concorrência do acesso entre estes módulos fosse feita de maneira controlada. A introdução destes pontos de sincronização será discutida na próxima seção.

Finalmente a quarta condição citada acima é extremamente difícil de ser satisfeita uma vez que para se evitar o encadeamento de chamadas bloqueantes, o gerenciador de serviços teria de conhecer a semântica de operação de cada um dos serviços. Além disso, a introdução de pontos de sincronização a fim de manter os estados internos dos serviços consistentes, potencializa a ocorrência de *deadlocks* na plataforma. A estratégia neste caso consistiu em projetar cuidadosamente as interfaces de serviço de modo a mantê-las com um mínimo grau de interdepedência, o que certamente diminui os riscos de chamadas encadeadas. Maiores detalhes sobre as interfaces dos serviços são apresentadas na seção a seguir.

A.3 Os módulos de serviço

Conforme visto no Cap. 4, os módulos de serviço possuem quatro sub-módulos, cada qual destinado a um componente JOIN. Como a estrutura interna destes sub-módulos é a mesma, nas seções a seguir será apresentada a especificação formal desta estrutura e a implementação adotada para ela na plataforma JOIN.

A.3.1 Modelo formal dos módulos de serviço

Nesta seção serão introduzidas definições que especificam formalmente a estrutura interna de um sub-módulo de serviço em JOIN.

Definição 11 *Define-se um **estado** S como sendo um conjunto $S = \{\theta_1, \theta_2, \ldots, \theta_Q\}$ de Q blocos de dados tais que $\theta_i \cap \theta_j = \oslash \; \forall i, j \in [1, Q], i \neq j$. Os blocos de dados de um estado são também denominados sub-estados, e o bloco θ_1 é denominado sub-estado inicial.*

Definição 12 *Define-se um **bloco de execução** ϵ como sendo uma seqüência finita de instruções que definem operações a serem realizadas sobre um conjunto de sub-estados. Um bloco de execução pode conter outros blocos de execução aninhados em seu interior. Entretanto, neste caso os sub-estados operados pelo(s) bloco(s) de execução aninhados não são considerados entre os sub-estados operados pelo bloco de execução externo.*

Definição 13 *Define-se uma **linha de execução** λ como sendo a entidade abstrata responsável pela execução dos blocos de execução. Como as linhas de execução são fornecidas pelos sistemas operacionais e, neste caso, disponibilizadas por meio da Máquina Virtual Java, sua estrutura não será detalhada neste trabalho. Maiores informações podem ser encontradas em [Mic02]*

Definição 14 *Define-se um sub-módulo de serviço como sendo formado por:*

- *um estado S,*
- *um conjunto $E = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_R\}$ de R blocos de execução com operações definidas sobre os sub-estdos de S.*
- *um conjunto $L = \{\lambda_1, \lambda_2, \ldots, \lambda_K\}$ de K linhas de execução. A linha de execução λ_1 é denominada linha de execução principal e é responsável pela execução do bloco ϵ_1.*

Os sub-módulos de serviço serviços devem ainda atender às seguintes propriedades.

1. *O estado inicial de um serviço é imutável, ou seja, não pode sofrer operações de escrita por nenhum bloco de execução.*
2. *Não é permitido que duas ou mais linhas de execução distintas executem simultaneamente blocos que realizem operações sobre conjuntos de sub-estados que contenham elementos em comum. Para isso, mecanismos de exclusão mútua devem ser utilizados.*
3. *As linhas de execução podem ser disparadas pelo próprio serviço, a partir de um determinado bloco de execução, ou podem ser disparadas por entidades externas ao serviço. Desta forma, deverão ser informados quais os blocos de execução que estão disponíveis para serem executados por estas entidades externas.*

De maneira geral, um sub-módulo de serviço pode ser entendido como um conjunto de blocos de execução que são executados por linhas de execução geradas pelos próprios blocos de execução do serviço ou por entidade externas, como por exemplo outros sub-módulos de serviço, que efetuam chamadas a este serviço. A execução de um bloco de execução pode ou não alterar o estado interno do serviço e, por isso, mecanismos de exclusão mútua devem ser utilizados sempre que necessário para se garantir a consistência destes dados na ocorrência de condições de corrida.

Estes conceitos serão melhor entendidos na próxima seção, onde será descrita a implementação deste modelo na plataforma JOIN.

A.3.2 Implementação dos módulos de serviço

Conforme visto anteriormente, o gerenciador de serviços é o componente da plataforma JOIN responsável pelo carregamento e inicialização dos módulos de serviço nos diferentes componentes da plataforma JOIN. Para isso, cabe a ele efetuar o carregamento dinâmico das classes de software correspondentes aos sub-módulos de serviço e iniciar a execução das mesmas.

Desta forma, cabe ao gerenciador de serviços o fornecimento da linha de execução principal que será responsável pelo bloco de execução inicial. Será nesta linha que as operações internas do sub-módulo de serviço serão realizadas de modo que este possa desempenhar suas funções corretamente.

Segundo o modelo apresentado na seção anterior, um sub-módulo de serviço pode ainda receber requisições de entidades externas a ele sob a forma de linhas de execução que executam blocos exportados. Em termos concretos, isto se dá através da exportação de interfaces correspondentes às funcionalidades que os sub-módulos de serviço implementam. Por exemplo, o sub-módulo servidor do módulo escalonador exporta uma interface que contém o método de particionamento PPCP de uma aplicação paralela. Este método é invocado pelo sub-módulo servidor do gerenciador de aplicações que, por meio de uma de suas linhas de execução, executa o bloco de execução do serviço escalonador que implementa o particionamento PPCP.

Dito de maneira sintética, cada método da interface exportada por um sub-módulo de serviço pode ser entendido como um bloco de execução disponível às linhas de execução externas a este serviço e oriúndas de outros sub-módulo de serviço. Além da possibilidade de um serviço ter os seus blocos de execução executados desta forma, ele ainda dispõe de sua linha de execução principal, que é fornecida pelo gerenciador de serviços.

Para que os sub-módulo de serviço possam ser tratados pelo gerenciador de serviços de maneira homogênea, existe uma interface padrão que estes devem implementar. Denominada *ServiceCode*, esta interface procura separar o processo de inicialização e execução dos sub-módulo de serviço em etapas claramente definidas. Esta organização se faz necessária porque durante o processo de inicialização, um serviço pode realizar chamadas a outro(s) serviço(s), o que pode resultar em condições de corrida. Com isso, espera-se que, controlando-se o acesso dos serviços a certos recursos compartilhados, minimize-se o risco de que, num dado momento, o estado interno de algum serviço torne-se inconsistente. A fim de ilustrar este risco potencial, considere o exemplo a seguir.

1. Um serviço de manipulação de tarefas exporta uma interface que contém métodos dedicados à consulta e à execução de operações sobre tarefas. Um destes métodos de consulta tem como dados de retorno identificadores de tarefas e, quando a resposta a uma consulta for nula, deverá retornar o valor de identificação de tarefas nulo.

2. O valor de identificação de tarefas nulo é definido pelo serviço identificador de tarefas, já que é dele a responsabilidade de definir a estrutura interna destes identificadores. Assim, para se saber qual é o valor deste identificador nulo, o serviço de identificação de tarefas deve ser consultado.

3. Um serviço qualquer precisa, ao iniciar, efetuar uma consulta ao manipulador de tarefas. Se esta consulta resultar nula, o manipulador de tarefas deverá consultar o serviço de identificadores de tarefas para buscar o valor de identificador nulo.

4. Se por alguma razão o serviço de identificadores de tarefas não estiver pronto para responder à requisição, por exemplo por estar com seu estado interno ainda inconsistente e em fase de inicialização, este poderá retornar um valor errado ou bloquear a chamada até que esteja com o valor correspondente pronto. O bloqueio de chamadas torna-se ineficiente, já que será testado para todas as futuras chamadas, e aumenta a probabilidade de *deadlocks* na plataforma.

A solução adotada consistiu em definir métodos que separassem a execução do bloco de execução inicial em etapas de inicialização, processamento principal e finalização. Na Fig. A.1 é apresentada uma transcrição desta interface básica.

```
1 package join.service;
2
3 public interface ServiceCode {
4     void localInit();      // Inicialização independente
5     void globalInit();     // Inicialização com outro serviço
6     void serviceRun();     // laço principal de execução
7     void serviceFinally(); // código de finalização de execução
8 }
```

Figura A.1: Interface básica dos módulos de serviço JOIN

A separação do processo de inicialização de um serviço em dois métodos - *localInit()* e *globalInit()* - define dois pontos de sincronização na execução destes sub-módulos. Quando um sub-módulo de serviço é carregado pelo gerenciador de serviços, a linha de execução principal fornecida a este sub-módulo inicia a execução do mesmo pelo método *localInit()*. Concluída a execução deste método, a linha de execução é bloqueada e fica aguardando a sinalização do gerenciador de serviços para poder prosseguir. Quando todas os serviços já foram carregados e já tiveram seus métodos *localInit()* executados por suas respectivas linhas

de execução principais, então estas linhas são desbloqueadas e executam o método *globalInit()* de seus sub-módulos de serviço. Novamente, ao terminar a execução deste método as linhas de execução aguardam até que seja dada a permissão para que estas prossigam na execução do sub-módulo. Depois que todos os métodos *globalInit()* foram executados, os linhas de execução são desbloqueadas para que possam executar o método *serviceRun()* e, eventualmente, o *serviceFinally()*.

A seguir são descritos em detalhes as funções de cada um destes métodos.

localInit()

Dentro deste método os serviços devem promover as ações e inicialização que possam ser feitas independentemente de outros serviços. Dito de outra maneira, dentro deste método não são permitidos acessos a outros serviços.

Para os serviços que necessitarem de múltiplas linhas de execução que façam chamadas a outros serviços, é também neste método que essas linhas de execução devem ser registradas junto ao gerenciador de serviços. Este procedimento se faz necessário para que o núcleo do sistema mantenha-se informado a respeito das entidades com que ele potencialmente interagirá. Isso evita que, por exemplo, um serviço crie linhas de execução que se façam passar por outro serviço. No entanto, não há nada que impeça um serviço de criar linhas de execução sem o conhecimento do gerenciador de serviços desde que estas não interajam com nenhum outro serviço ou com o gerenciador de serviços. Este caso não é considerado como uma falha de segurança já que não põe em risco o funcionamento de outros serviços.

Cabe aqui ressaltar ainda que para a grande maioria dos serviços este método deverá ser suficiente para a sua completa inicialização. Não deve ser o caso mais comum que para um serviço entrar em plena operação ele necessite de informações detidas por outros serviços. Caso isso aconteça, sua inicialização deve utilizar o método *globalinit()*.

globalInit()

É neste método que os serviços devem executar as ações de inicialização que dependam de outras serviços, ou seja, é a partir deste método que o acesso a outros serviços é possível.

Todo o módulo de serviço que utilizar este método em seu processo de inicialização deverá especificar em sua documentação que somente após a execução do *globalinit()* ele estará disponível para atender a chamadas de outros serviços.

serviceRun()

Este método é chamado após o término da fase de inicialização de todos os serviços. Em seu corpo deverá ser introduzido o código responsável pela realização do ciclo principal de funcionamento dos serviços.

Para se entender melhor a utilização deste método, considere que um determinado serviço denominado "Calculadora" procure implementar um mecanismo capaz de efetuar, no nível do servidor, operações ma-

temáticas simples e complexas. Para isso, seu sub-módulo servidor deverá implementar uma interface que exporte todas as operações matemáticas que este serviço oferecerá; para as operações matemáticas mais simples, o cálculo poderá ser feito localmente dentro do corpo do método que implementa esta operação. Já para as operações mais complexas, o serviço poderá dividir o trabalho de cálculo entre seus sub-módulos coordenadores para que estes ordenem sua execução nos sub-módulos trabalhadores. A comunicação entre servidor e coordenadores e entre coordenadores e trabalhadores não se dá por chamadas de métodos, mas sim por troca de mensagens via serviço *"Communicator"*. Para que estas mensagens sejam recebidas e tratadas convenientemente, um laço de espera de mensagens deve existir e, para este caso, o método *serviceRun()* deve ser utilizado.

Percebe-se portanto que o método *serviceRun()* se presta à implementação da linha de execução necessária ao funcionamento **interno** do serviço. Além da comunicação entre sub-módulos de um mesmo serviço, é neste método que operações como, por exemplo, salvamento de estado, devem ser realizadas.

serviceFinally()

Sempre que um componente JOIN (servidor, coordenador, trabalhador ou jack) for finalizado, o gerenciador de serviços deste componente invocará o método *serviceFinally()* de cada um de seus serviços. É neste método que toda as ações de finalização devem ser implementadas.

Algumas ações de finalização típicas são a atualização das estruturas de dados internas, o salvamento do estado do seviço e o envio de mensagens informando a finalização do sub-módulo.

A.3.3 Serviços especiais e ordinários

Na plataforma JOIN há dois tipos básicos de serviço: os serviços **especiais** e os serviços **ordinários**. Os serviços especiais são aqueles que necessitam de algum tratamento específico por parte da plataforma no momento em que são carregados. Por essa razão, são serviços menos flexíveis já que estão mais fortemente atrelados ao código do gerenciador de serviços. Já os serviços ordinários não necessitam de nenhum tratamento especial ao serem carregados e por isso apresentam estrutura e funções mais genéricas.

A seguir será apresentada uma breve descrição dos principais serviços presentes na plataforma, justificando-se a classificação dos mesmos como especial ou ordinário.

Serviços especiais

A plataforma JOIN dispõe dos seguintes serviços especiais:

Gerenciador de TIDs: um TID é um identificador de tarefa, ou seja, é um número inteiro atribuído a uma linha de execução que procurará identificá-la unicamente dentro da plataforma. A entidade responsável pela atribuição de TIDs aos diversos serviços da plataforma e às linhas de execução que estes criarem é o gerenciador de TIDs, que também poderá ser consultado pelos demais serviços quando estes necessitarem de informações a respeito de um TID qualquer.

O gerenciador de TIDs deve ser um serviço especial porque é dele a responsabilidade de definir o valor de constantes especiais gerenciadas pelo gerenciador de serviços, tal como o TID nulo. Além disso, o gerenciador de serviços precisará do gerenciador de TIDs para a atribuiçao dos TIDs dos serviços que serão criados posteriormente. Assim, este serviço deve ser o primeiro a ser carregado e deve ser consultado pelo gerenciador de serviços.

Gerenciador de Segurança: o gerenciador de segurança é o serviço responsável pela implementação da política de controle de acessos entre serviços e entre usuários e a plataforma. Este serviço é especial porque necessita de informações privilegiadas a respeito do núcleo da plataforma (gerenciador de serviços) para poder operar corretamente.

Serviços ordinários

Os serviços ordinários atualmente disponíveis à plataforma JOIN são:

Gerenciador de Aplicações: conforme visto no Cap. 4, este serviço cuida do controle da execução das aplicações na plataforma. Assim, é dele a responsabilidade pelo envio de tarefas de aplicação aos componentes trabalhadores, pelo recolhimento dos resultados das tarefas e pela ligação de dados entre elas.

Gerenciador de Comunicação: o gerenciador de comunicação, ou *communicator* é o responsável pela comunicação entre sub-módulos de um mesmo serviço. Esta comunicação é feita via troca de mensagens e, por isso, este serviço se vale de conexões TCP/IP entre seus sub-módulos.

Gerenciador de Eventos: este serviço oferece toda a infraestrutura para a geração e o recebimento de eventos assíncronos dentro do escopo de um componente da plataforma. Os eventos podem ser gerados e capturados por quaisquer serviços e sua principal motivação é a possibilidade de rastrear acontecimentos que possam ser úteis para o funcionamento de outros serviços.

Gerenciador de Estado: o gerenciador de estado oferece aos demais serviços a possibilidade de armazenamento de informações em meio estável, o que permite que os serviços recuperem seus estados quando da ocorrência de falhas locais. Nota-se, no entanto, que este serviço está disponível apenas nos componentes servidor e coordenador, que são os componentes que têm acesso ao sistema de arquivos local.

Gerenciador de Topologia: o gerenciador de topologia é o responsável pela definição de como os grupos de computadores irão se conectar. Atualmente, a política em vigor é a de *hipercubos dinâmicos virtuais incompletos*, mas qualquer outra topologia poderia ser implementada por este serviço, tais como topologias hierárquicas e em anel.

Recepcionista: este serviço é o responsável pela implementação do mecanismo de recepção de novos computadores à plataforma. Assim, quando contactado por um computador que ainda não faça parte do JOIN, procura atribuir-lhe um papel (trabalhador ou coordenador) e enviar-lhe os dados necessários para o início de suas atividades. É também responsável pela geração dos eventos correspondentes à entrada de um novo computador na plataforma.

Escalonador: conforme visto no Cap. 4, este serviço, usado exclusivamente pelo gerenciador de aplicações, é o responsável pela repartição de aplicações entre grupos de trabalhadores e pela atribuição de tarefas de aplicação entre os trabalhadores de um grupo. Para isso, implementa as diferentes políticas e algoritmos de escalonamento que sejam de interesse do gerenciador de aplicações.

A.4 Considerações finais

Depois de implementadas e testadas todas as alterações no núcleo do JOIN, ele foi disponibilizado aos demais membros do grupo de pesquisas para que estes pudessem avalia-lo em sua nova versão. A aprovação destas modificações pelo grupo foi unânime e o principal ponto elogiado foi o considerável aumento na facilidade com que novas funcionalidades podiam ser implementadas.

A simplificação do processo de inclusão de novos módulos de serviço motivou a incorporação de inúmeras funcionalidades à plataforma, que a tornaram mais poderosa e mais flexível. A velocidade no desenvolvimento da plataforma aumentou consideravelmente e o tempo gasto em sua manutenção e na correção de bugs caiu drasticamente.

Finalmente, a adoção desta nova estrutura permitiu que o código da plataforma pudesse ser gerenciado de maneira mais eficiente e segura; o código básico, compreendendo o gerenciador de serviço, todas as classes necessárias ao seu carregamente e os serviços especiais, passaram a ser mantidas sob acesso mais restrito, enquanto que os módulos de serviços ordinários podiam ser vistos e eventualmente manipulados por todos os membros do grupo.

Acredita-se hoje que esta nova estrutura ainda poderá sofrer alterações futuras, mas certamente manterá suas características básicas durante um longo período.

Apêndice B

Funções MATLAB utilizadas nas Simulações

B.1 Geradores de aplicações e grupos

Função geradora de aplicações:

```
function [D,M] = generateAPP(b,m)

% {D,M} = generateAPP(b,m)
%
% Generates an hypotetic paralled application.
% The argument b is the number of task batches, and
% the parameter m is the aproximate total number of
% tasks desired.

mul = zeros(b,1);
route = 0;
while(min(size(route)) <= 1), % avoid trivial applications
    dep = zeros(b);
    % Generates the precedence relations matrix
    for i=1:b-1,
        while (sum(dep(i,:)) == 0),
            dep(i,(i+1:b)) = round(rand(1,(b-i)));
        end;
    end;
    route = traceroute(dep);
end;
```

```
r = route(:,2:end-1);
coefs = [1 zeros(1,length(r(:,1))-1)];
piv = m^(1/length(coefs));

while (coefs(end) <= 0),
    for i=2:length(coefs)-1,
        coefs(i) = round(piv*(0.5+rand));
    end;
    ad = 2; ml = 0;
    for j=1:b-2,
        par = 1;
        for i=1:length(coefs)-1,
            if (r(i,j)),
                par = par*coefs(i);
            end;
        end;
        if (r(end,j)),
            ml = ml + par;
        else,
            ad = ad + par;
        end;
    end;
    coefs(end) = round((m-ad)/ml);
end;

for j=1:b-1,
    par = 1;
    for i=2:length(coefs),
        if (route(i,j)),
            par = par*coefs(i);
        end;
    end;
    mul(j) = par;
end;
mul(b) = 1;

M = mul;
D = dep;
```

Função auxiliar utilizada na geração de aplicações:

```
% This function isn't called directly by the user
% It receives a dependency matrix and returns a matrix
```

```
% containing all possible routes inside the graph defined
% by this dependency matrix.
function [R] = traceroute(dep);

b = length(dep);
mul = zeros(b,1);
route = zeros(b);

j = 0;
for i=1:b-1,
    if (mul(i) == 0),
        j = j + 1;
        rt = sort(trace(dep,i,[i]));
        mul(rt) = 1;
        route(j,rt) = 1;
    end;
end;

R = route(1:j,:);

% This is an auxiliary function used by traceroute
function [R] = trace(d,i,r)

for k=i+1:length(d),
    if (d(i,k) ~= 0),
        if isempty(find(r==k)),
            r = trace(d,k,[r k]);
        end;
    end;
end;
R = r;
```

Função geradora de grupos:

```
function [P,C] = generateGroup(n,b,mp,dp1,dp2,mc,dc)

% [P,C] = generateGroup(n,b,mp,dp1,dp2,mc,dc)
%
% Generates a group with n UPs prepared to accept
% b different types of tasks. The arguments mp,
% dp1 and dp2 define the level of heterogeneity of
% the processing power of the UPs. The arguments
% mc and dc define the heterogeneity of the communications
```

```
% links.

% this is the communication matrix
C = mc+dc-2*dc*rand(1,n);

% Intra-UP dispersion
avg = mp*ones(1,n)+(dp1-2*dp1*rand(1,n));

for i=1:n,
    % rows represent tasks and columns UPs
    P(:,i) = avg(i)'+(dp2-2*dp2*rand(b,1));
end;
```

B.2 Algoritmos de escalonamento intra-grupo

Função implementadora do escalonamento trivial:

```
% This function implements the trivial scheduling algorithm.
% The parameters d and m define the dependency relations
% and the batchs' cardinality of the application. The
% matrices p and c define the processing power of the
% units and the speed of its communication links,
% respectively. As a result, the matrix BO indicates
% the order in which the batches are to be scheduled, and
% the matrix TO indicates the order in which the units are
% to receive the tasks.
function [BO,TO] = simpleSchedule(d,m,p,c)

t = length(p(:,1));
n = length(p(1,:));
alloc = zeros(1,n);
bstatus = zeros(t,1);
bo = []; to = [];
offset = 0;

while(sum(bstatus) < 2*t),
    rt = find(sum(d)==0);           % ready or completed batches
    bstatus(bstatus(rt)==0) = 1;    % set status of ready batches
    rti = find(bstatus==1);
    tm = zeros(1,n);
    for i=rti',
        bo = [bo i];
```

```
        for j=1:m(i),
            up = find(alloc == min(alloc)); up = up(1);
            alloc(up) = alloc(up) + 1;
            to = [to up];
        end;
        d(i,1:end) = 0;
        bstatus(i) = 2;
    end;
    bo = [bo 0];
end;

BO = bo;
TO = to;
```

Função implementadora do escalonamento *best-fit*:

```
% This function implements the best-fit scheduling algorithm.
% The parameters d and m define the dependency relations
% and the batchs' cardinality of the application. The
% matrices p and c define the processing power of the
% units and the speed of its communication links,
% respectively. As a result, the matrix BO indicates
% the order in which the batches are to be scheduled, and
% the matrix TO indicates the order in which the units are
% to receive the tasks.

function [BO,TO] = bestfitSchedule(d,m,p,c)

t = length(m);
n = length(p(1,:));
bstatus = zeros(t,1);
bo = []; to = [];

while(sum(bstatus) < 2*t),
    rt = find(sum(d)==0);           % ready or completed batches
    bstatus(bstatus(rt)==0) = 1;    % set status of ready batches
    rti = find(bstatus==1);
    tm = zeros(1,n);
    tmc = 0;
    for i=rti',
        for u=1:m(i),   % m(i) tasks of type i
            % Select best-fit UP
            bfup = 0; df1 = 0; df2 = inf;
```

```
            for j=1:n,
                tmc2 = c(j) + tmc;
                tm2 = tm - c(j)*ones(1,n);
                tm2(j) = tm2(j) + 2*c(j) + p(i,j);
                tm2(tm2<0) = 0;
                df1 = max(tm2+tmc2*ones(1,n)) - max(tm+tmc*ones(1,n));
                if (df1 < df2),
                    df2 = df1;
                    bfup = j;
                    tm3 = tm2;
                end;
            end;
            to = [to bfup];
            tmc = tmc + c(bfup);
            tm = tm3;
        end;
        bo = [bo i];
        d(i,1:end) = 0;
        bstatus(i) = 2;
    end;
    bo = [bo 0];
end;

BO = bo;
TO = to;
```

B.3 Escalonadores

Função implementadora do escalonador estático:

```
% This function implements the static scheduler.
% The parameters d and m define the dependency relations
% and the batchs' cardinality of the application. The
% matrices p and c define the processing power of the
% units and the speed of its communication links, The
% flag sm indicates the scheduling algorithm to be used
% (trivial or best-fit). The parameter erl indicates the
% max error level to be assigned to the performance estimations
% and the erm is a matrix indicating which units are to be
% affected by this error. As a result, the variable T1 indicates
% the total scheduling time, and the variable T2 indicates the
% time spent during the scheduoing calculus.
```

```
function [T1,T2] = staticScheduler(d,m,p,c,sm,erl,erm)

n = length(p(1,:));
tm = zeros(1,n);
ttime = 0;
erri = 0;
tpt = 1;

% Calculate schedule (task allocation order)
tic;
switch(sm)
case(0)
    [bo,to] = simpleSchedule(d,m,p,c);
case(1)
    [bo,to] = bestfitSchedule(d,m,p,c);
end;
T2 = toc;

for i=bo,
    if (i ~= 0),
        % Calculate times
        idx = to(tpt:tpt+m(i)-1);

        % Introduce error
        error = erl*sin(erri*2*pi/sum(m)); % calculate error
        %error = erl*(1-2*rand); % calculate error
        p(:,erm) = p(:,erm) + p(:,erm)*error; % add error
        c(erm) = c(erm) + c(erm)*error; % add error
        erri = erri + m(i);

        for j=idx,
            % Calculate times
            tm(j) = tm(j) + 2*c(j)+p(i,j); % time to send, process and return
            tm = tm - c(j)*ones(1,n); % time-to-send discount
            tm(tm<0) = 0;
            ttime = ttime + c(j);
        end;
        tpt = tpt + m(i);
    else,
        ttime = ttime + max(tm);
        tm = zeros(1,n);
    end;
```

```
end;

T1 = ttime;
```

Função implementadora do escalonador geracional:

```
% This function implements the generational scheduler.
% The parameters d and m define the dependency relations
% and the batchs' cardinality of the application. The
% matrices p and c define the processing power of the
% units and the speed of its communication links, The
% flag sm indicates the scheduling algorithm to be used
% (trivial or best-fit). The parameter erl indicates the
% max error level to be assigned to the performance estimations
% and the erm is a matrix indicating which units are to be
% affected by this error. As a result, the variable T1 indicates
% the total scheduling time, and the variable T2 indicates the
% time spent during the scheduoing calculus.

function [T1,T2] = gsScheduler(d,m,p,c,sm,erl,erm)

t = length(p(:,1));
n = length(p(1,:));

T2 = 0;
ttime = 0;
rtime = 0;
tm = zeros(1,n);
erri = 0;
bpt = 1; tpt = 1;

while(tpt <= sum(m)),
    % Calculate schedule (task allocation order)
    tic;
    switch(sm)
    case(0)
        [bo,to] = simpleSchedule(d,m,p,c);
    case(1)
        [bo,to] = bestfitSchedule(d,m,p,c);
    end;
    T2 = T2 + max(0,(toc - rtime));

    while (bo(bpt) ~= 0),
```

```
        i = bo(bpt);
        % Calculate times
        idx = to(tpt:tpt+m(i)-1);

        % Introduce error
        error = erl*sin(erri*2*pi/sum(m)); % calculate error
        %error = erl*(1-2*rand); % calculate error
        p(:,erm) = p(:,erm) + p(:,erm)*error; % add error
        c(erm) = c(erm) + c(erm)*error; % add error
        erri = erri + m(i);

        for j=idx,
            % Calculate times
            tm(j) = tm(j) + 2*c(j)+p(i,j); % time to send, process and return
            tm = tm - c(j)*ones(1,n); % time-to-send discount
            tm(tm<0) = 0;
            ttime = ttime + c(j);
        end;
        tpt = tpt + m(i);
        bpt = bpt + 1;
    end;
    ttime = ttime + max(tm);
    tm = zeros(1,n);
    bpt = bpt + 1;
end;

T1 = ttime;
```

Função implementadora do escalonador geracional com replicação de tarefas:

```
% This function implements the GSTR scheduler.
% The parameters d and m define the dependency relations
% and the batchs' cardinality of the application. The
% matrices p and c define the processing power of the
% units and the speed of its communication links, The
% flag sm indicates the scheduling algorithm to be used
% (trivial or best-fit). The parameter erl indicates the
% max error level to be assigned to the performance estimations
% and the erm is a matrix indicating which units are to be
% affected by this error. As a result, the variable T1 indicates
% the total scheduling time, and the variable T2 indicates the
% time spent during the scheduoing calculus.
function [T1,T2] = gstrScheduler(d,m,p,c,sm,erl,erm,flm)
```

```
t = length(p(:,1));
n = length(p(1,:));

T2 = 0;
ttime = 0;
rtime = 0;
bstatus = zeros(t,1);
alloc = zeros(1,n);
erri = 0;
old_p = p; old_c = c;

while(sum(bstatus) < 2*t),
    rt = find(sum(d)==0);           % ready or completed batches
    bstatus(bstatus(rt)==0) = 1;    % set status of ready batches
    rti = find(bstatus==1);
    tm = zeros(1,n);
    for i=rti',
        % Introduce failure
        p(:,flm) = p(:,flm) + 10e8*ones(t,length(flm));
        alloc(flm) = alloc(flm) + 10e8;

        % Introduce error
        error = erl*sin(erri*2*pi/sum(m)); % calculate error
        %error = erl*(1-2*rand); % calculate error
        p(:,erm) = p(:,erm) + p(:,erm)*error; % add error
        c(erm) = c(erm) + c(erm)*error; % add error
        erri = erri + m(i);

        for j=1:m(i),
            % Select up
            tic;
            if (sm == 0),
                up = find(alloc == min(alloc)); up = up(1);
                alloc(up) = alloc(up) + 1;
            else,
                base = tm + old_p(i,:) + 2*old_c;
                up = find(base == min(base)); up = up(1);
            end;
            T2 = T2 + max(0,(toc - rtime));
            % Calculate times
            tm(up) = tm(up) + 2*c(up) + p(i,up); % time to send, process and return
```

```
            tm = tm - c(up)*ones(1,n); % time-to-send discount
            old_p(i,tm<0) = p(i,tm<0); % execution time update
            old_c(tm<0) = c(tm<0);     % communication time update
            tm(tm<0) = 0;
            rtime = c(up);
            ttime = ttime + rtime; % time to send
        end;
        d(i,:) = zeros(1,t);
        bstatus(i) = 2;
    end;
    rtime = max(tm(tm<10e5));
    ttime = ttime + rtime;
end;

T1 = ttime;
```

B.4 Algoritmos de escalonamento inter-grupo

Função implementadora do particionamento trivial:

```
% This function implements the trivial partitioning.
% The parameter ngs is an integer that specifies the
% number of groups in which the application has to be
% partitioned. The parameter m is a vector containing
% the cardinality of each application batch.
function result = tp(ngs,m)

b = length(m);
result = [];

for i=1:b,
    column = []; temp = m(i);
    for j=1:ngs,
        temp2 = ceil(temp/(ngs+1-j));
        column = [temp2; column];
        temp = temp - temp2;
    end;
    result = [result column];
end;
```

Função implementadora do particionamento proporcional:

```
% This function implements the proportional partitioning.
% The parameter grs is a vector containing the number of
% UPs in each group. WARNING !! The elements of this vector
% have to be sorted in ascending order. The parameter m is
% a vector containing the cardinality of each application batch.
function result = pp(grs,m)

b = length(m);
ngs = length(grs);
total = sum(grs);
result = [];

for i=1:b,
    column = []; temp = m(i);
    for j=ngs:-1:1,
        temp2 = ceil(temp*grs(j)/sum(grs(1:j)));
        column = [temp2; column];
        temp = temp - temp2;
    end;
    result = [result column];
end;
```

Função implementadora do particionamento PPCP:

```
% This function implements the proportional partitioning.
% The parameter grs is a vector containing the number of
% UPs in each group. WARNING !! The elements of this vector
% have to be sorted in ascending order. The parameters p and
% c define the group in which the app is going to be partitioned
% and the parameters d and m define the app to be particioned.
function result = ppcp(grs,p,c,d,m)

b = size(d,1);
ngs = length(grs);
result = zeros(ngs,b);

% Assign sychronization batchs to the group with the lowest RC

i = find(m < length(grs));
result(ngs,i') = m(i)';

route = traceroute(d);
```

```
s = min(find(sum(result) == 0));
while (s),
    batchgroup = route(find(route(:,s)),:);
    if (size(batchgroup,1) > 1), batchgroup = sum(batchgroup); end;
    batchgroup(find(sum(result))) = 0;
    batchgroupindex = find(batchgroup);
    cardinalities = m(batchgroupindex);
    batchgcd = min(gcd(cardinalities,min(cardinalities)));

    k = batchgcd;
    for i=ngs:-1:1,
        pr = grs(i)/sum(grs(1:i));
        mask = ceil(pr*k);
        result(i,batchgroupindex) = mask*m(batchgroupindex)'/batchgcd;
        k = k - mask;
    end;

    s = min(find(sum(result) == 0));
end;
```

Função implementadora do escalonador inter-grupos:

```
% This function implements the inter-group scheduler.
% The parameter part is the partitioning provided by a
% partitioning algorithm (Tp, PP or PPCP).
% The parameter grs is a vector containing the number of
% UPs in each group. WARNING !! The elements of this vector
% have to be sorted in ascending order. The parameters pp and
% cc define the group in which the app is going to be partitioned
% and the parameter d defines the dependency relations among the
% application batches.
function [ptime,packets] = igp(part,grs,pp,cc,d)

m = sum(part);
b = size(part,2);
ngs = size(part,1);

ptime = 0; packets = 0;
ready = find(sum(d)==0);
while (length(ready) > 0),

    % this loop calculates the number of packets exchenged
```

```
for i=ready,
    sinks = find(d(i,:));
    for j=sinks,
        comm = 0;
        if ((m(i) == 1) | (m(j) == 1)),
            i1 = find(part(:,i));
            i2 = find(part(:,j));
            if (length(i1) == 1),
                comm = sum(part(:,j)) - part(i1,j);
            else,
                comm = sum(part(:,i)) - part(i2,i);
            end;
        else,
            q = 1:ngs;
            if (m(i) > m(j)),
                temp = abs(part(q,i) - part(q,j)*m(i)/m(j));
            else,
                temp = abs(part(q,i)*m(j)/m(i) - part(q,j));
            end;
            comm = sum(temp) / 2;
        end;
        packets = packets + comm;
    end;
end;

pt = zeros(1,ngs);

% this loop calculates the processing times

for i=1:ngs,
    mm = zeros(1,b); mm(ready) = part(i,ready);
    p = pp(:,1:grs(i),i);
    c = cc(i,1:grs(i));
    if (sum(mm) > 0),
        [pt(i),tp] = gstrScheduler(zeros(b),mm,p,c,1,0,[],[]);
    else,
        pt(i) = 0;
    end;
end;

ptime = ptime + max(pt);
```

```
    d(ready,:) = zeros(length(ready),b);
    d(b*(ready-1)+ready) = 1;
    ready = find(sum(d)==0);
end;
```

Apêndice C

A aplicação de testes

C.1 Primeiro lote de tarefas

Neste lote, de cardinlidade 1, é executada a tarefa que distribui o espaço total de busca entre as tarefas do lote seguinte. Cabe aqui destacar que esta é a tarefa que sofre alterações dependendo da instância da aplicação que se deseja executar (1 a 5).

```
package numberSieve;

import java.io.*;

import join.app.*;

/**
 * DistributeData.java
 *
 * Created: Thu Mar 21 15:45:00 2002
 *
 * @author Fabiano O. Lucchese
 * @version 1.0
 */

public class DistributeData implements AppCode {

    static long pcount = 20;            // number of tasks
    static long max = 100000000;        // max limit of the range (first instance)
    long psize;                         // partition size for each thread
    long total = 0;
    int r = 0;
    boolean[] primes;
```

```
    void init() {
psize = max/pcount;

        // compute square-root r
        long p = 0, i = 1;

        while (p < max) {
    p += i; i += 2; r++;
}
        primes = new boolean[r];

        // first array-segment initialization
primes[0] = false; primes[1] = false;
        for (int index = 2; index < r; index++) {
    primes[index] = true;
}

        // removal non-primes from initial array-segment [0..r]
        int pt = 0;
        while (++pt < r)
    if (primes[pt]) {
total++;
for (int index = pt+pt; index < r; index += pt)
    primes[index] = false;
    }

System.out.println("Found "+total+" primes into the initial interval.");
    }

    public Serializable taskrun(Serializable par) {
this.init();

Object[] data = new Object[(int) pcount];
for (int i=0; i<pcount; i++) {
    Object[] pars = new Object[2];
    long[] bounds = new long[2];

    bounds[0] = psize*i;
    bounds[1] = bounds[0]+psize;

    pars[0] = primes;
```

```
    pars[1] = bounds;
    data[i] = pars;
}
((long[]) ((Object[]) data[0])[1])[0] = r;  // correction

return (Serializable) data;
    }
}// DistributeData
```

C.2 Segundo lote de tarefas

Neste lote, de cardinlidade 50, são executadas as tarefa que realizam a busca dos números primos nos intervalos definidos pela tarefa do primeiro lote.

```
package numberSieve;

import java.io.*;

import join.app.*;

/**
 * NumberSieve.java
 *
 * Created: Thu Mar 21 16:12:45 2002
 *
 * @author Fabiano O. Lucchese
 * @version 1.0
 */

public class NumberSieve implements AppCode {

    private long start, end;
    private boolean[] primes;

    // This function finds the number primes in a given partition
    public long countPrimes() {

boolean[] mySieve;
long b, len, stotal = 0;
        int i, p, r = primes.length;
```

```
        // initialize segment [b,b+r]
mySieve = new boolean[r];
        for (i = 0; i < r; i++) mySieve[i] = true;

        // remove multiples of initial primes from succeeding segments
        for (b = start; b < end; b += r) {
    long k, pb;
    // remove multiples of initial primes from [b,b+r]
    p = 0;
    while (++p < r) if (primes[p]) {
k = p*(1+(b-1)/p);
i = (int) (k%b);
for (; i < r; i += p) mySieve[i] = false;
    }

    // count primes in segment [b,b+r] and reset segment array
    p = 0;
    if ((b+r) < end) k = r;
    else k = end%b;
    for (i = 0; i < k; i++) {
if (mySieve[i]) ++p;
mySieve[i] = true;
    }

    stotal += p;
}
        // return total number of primes for this partition
System.out.println("Found "+stotal+" primes into the interval [
                           "+start+","+end+"].");
        return stotal;
    }

    public Serializable taskrun(Serializable par) {
primes = (boolean[]) ((Object[]) par)[0];
start = ((long[]) ((Object[]) par)[1])[0];
end = ((long[]) ((Object[]) par)[1])[1];

return new Long(countPrimes());
    }

}// NumberSieve
```

C.3 Terceiro lote de tarefas

Neste lote, de cardinlidade 1, é executada a tarefa que agrupa os resultados parciais gerados pelas tarefas do segundo lote.

```
package numberSieve;

import java.io.*;

import join.app.*;

/**
 * GatherResults.java
 *
 * Created: Thu Mar 21 17:00:30 2002
 *
 * @author Fabiano O. Lucchese
 * @version 1.0
 */

public class GatherResults implements AppCode{

    public Serializable taskrun(Serializable par) {
long total = 0;
Object[] results = (Object[]) par;

for(int i=0; i<results.length; i++) {
    total += ((Long) results[i]).longValue();
}

System.out.println("Found "+total+" primes (excluding the initial interval).");
return null;
    }
}// GatherResults
```

Apêndice D

Artigos Derivados deste Trabalho

1. **Fabiano de O. Lucchese e Marco A. A. Henriques** - *Sequenciamento de Cadeias de DNA em Ambiente de Computação Paralelo Virtual Baseado na Internet* - Anais do WorkComp 2000, Workshop de Computação - 2000 - Instituto Tecnológico de Aeronáutica - São José dos Campos - SP

2. **Fabiano de O. Lucchese, Francisco S. Sambatti, Eduardo J. H. Yero e Marco A. A. Henriques** - *JoiN: Um Sistema Virtual de Computação Maciçamente Paralela* - Anais do WorkComp 2002, Workshop de Computação - 2002 - Instituto Tecnológico de Aeronáutica - São José dos Campos - SP

3. **Fabiano de O. Lucchese, Francisco S. Sambatti, Eduardo J. H. Yero e Marco A. A. Henriques** - *Um Esquema para Balanceamento de Carga em Ambientes Virtuais de Computação Maciçamente Paralela* - Anais do III WSCAD, Workshop em Sistemas de Computação de Alto Desempenho - 2002 - Vitória - ES

Apêndice E

Siglas Utilizadas no Texto

Sigla	Significado
GH	Grupo Heterogêneo
GS	Generational Scheduling
GSTR	Generational Scheduling with Task Replication
IP	Internet Protocol
IGP	Inter-group Partitioning
JDK	Java Development Kit
MPVC	Massively Parallel Virtual Computer
PAS	Parallel Application Specification
PASL	Parallel Application Specification Language
PVC	Parallel Virtual Computer
PP	Parallel Partitioning
PPCP	Parallel and Proportional Cluster Partitioning
TCP	Transmission Control Protocol
TID	Task IDentifier
TP	Trivial Partitioning
UDP	User Datagram Protocol
UP	Unidade de Processamento
URL	Universal Resource Locator
WWW	World Wide Web