Universidade Federal de Santa Maria — UFSM

Dissertação de Mestrado

SUPORTE AO CONTROLE E ALOCAÇÃO DINÂMICA DE COMPUTADORES EM JAVA

Márcia Cristina Cera

Programa de Pós-Graduação em Engenharia de Produção — PPGEP

Santa Maria, RS, Brasil

2005

SUPORTE AO CONTROLE E ALOCAÇÃO DINÂMICA DE COMPUTADORES EM JAVA

por

Márcia Cristina Cera

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-Graduação em Engenharia da Produção, área de concentração em Tecnologia da Informação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Engenharia da Produção**

PPGEP

Santa Maria, RS, Brasil

2005

Universidade Federal de Santa Maria
Centro de Tecnologia
PPGEP

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

SUPORTE AO CONTROLE E ALOCAÇÃO DINÂMICA DE COMPUTADORES EM JAVA

elaborada por
Márcia Cristina Cera

como requisito parcial para obtenção do grau de
Mestre em Engenharia da Produção

COMISSÃO EXAMINADORA:

Prof. Dr. Marcelo Pasin
(Presidente/Orientador — Departamento de Eletrônica e Computação — UFSM)

Prof. Dr. Nicolas Maillard
(Departamento de Computação Aplicada — UFRGS)

Prof. Dr. Andréa Schwertner Charão
(Departamento de Eletrônica e Computação — UFSM)

Santa Maria, 01 de abril de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Cera, Márcia Cristina

Suporte ao Controle e Alocação Dinâmica de Computadores em Java / Márcia Cristina Cera. – Santa Maria: PPGEP, 2005.

117 f.: il.

Dissertação (mestrado) – Universidade Federal de Santa Maria. PPGEP, Santa Maria, BR–RS, 2005. Orientador: Marcelo Pasin.

1. Java. 2. Programação paralela. 3. Dinamicidade. 4. Transparência. I. Pasin, Marcelo. II. Título.

UNIVERSIDADE FEDERAL DE SANTA MARIA
Reitor: Prof. Paulo Jorge Sarkis
Vice-Reitor: Prof. Clovis Silva Lima
Pró-Reitor de Pós-Graduação e Pesquisa: Prof. Paulo Tabajara Chaves Costa
Diretor do Centro de Tecnologia: Prof. Dr. Felipe Martins Müller
Coordenador do PPGEP: Prof. Dr. João Hélvio de Oliveira Righi
Coordenador da Área de Tecnologia da Informação: Prof. Dr. Marcos C. d' Ornellas

✍ A confecção desta dissertação foi realizada através do uso
de recursos de processamento de textos da ferramenta LaTeX 2ε e do editor *Vim*

Um Pito

Nenito Sarturi, Cláudio Patias e Nelcy Vargas

Olha guri, reparas o que estás fazendo,
depois que fores é difícil de voltar.
Passei-te um pito e continuas remoendo
teu sonho moço deste rancho abandonar.

Olha guri, lá no povo é diferente,
e certamente faltará o que tens aqui.
Eu só te peço: não esqueças de tua gente,
de vez em quando manda uma carta, guri.

Se vais embora, por favor não te detenhas;
segue em frente não olhes para trás
e assim não vais ver a lágrima insistente
que molha o rosto do teu velho, meu rapaz.

Olha guri, pra tua mãe cabelos brancos e
este velho que te fala sem gritar.
Pesa teus planos, eu quero que sejas brando;
se acaso fores pega o zaino para enfrenar.

Olha guri, leva uns cobres de reserva;
pega uma erva pra cevar teu chimarrão
e leva um charque que é pra ver se tu conservas
uma pontinha de amor por este chão.

Esta dissertação é dedicada a Deus, a minha família e a meus amigos.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter-me guiado no caminho que me trouxe até aqui.

Aos meus pais por plantarem em mim, ainda quando criança, a semente do interesse pelos estudos e a ensinar-me que é preciso fazer o melhor que se pode sempre, sem perder o respeito pelas outras pessoas. Agradeço também a toda a minha família que sempre me deu o apoio que necessitei e que se preocupa com o meu bem estar. É graças a vocês que alcanço cada uma de minhas vitórias e que supero minhas derrotas.

Ao Marcelo Pasin, meu orientador, por oferecer os meios e o apoio necessário durante a realização do trabalho. Também o agradeço por ter sido, e continuar sendo, o amigo que mostra o caminho, que incentiva, que se preocupa, que critica os erros e elogia os acertos. Obrigada pela confiança, pela paciência e por ter acreditado no meu potencial.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro durante a realização desse mestrado. Ao Programa de Pós-Graduação em Engenharia de Produção (PPGEP) por ter me acolhido e à Universidade Federal de Santa Maria (UFSM) pela oportunidade de acesso ao ensino público.

Aos professores do PPGEP pela contribuição no processo de formação e aos professores Andréa Charão e Nicolas Maillard por aceitarem fazer parte da banca examinadora e contribuírem para essa dissertação.

A todos os integrantes do Laboratório de Sistemas de Computação (LSC) que estiveram ao meu lado sempre que precisei. Principalmente pelo muito que me ensinaram, tanto no âmbito profissional quanto pela experiência de vida. Agradeço em especial ao Elton, pelas várias vezes em que ouviu meus problemas e procurou ajudar-me a encontrar as soluções.

Aos membros da Tribo Tupiniquim, Diego, Lucas, Marlon e Rodrigo, com quem tive a oportunidade de descobrir que a união e o companheirismo são o melhor energético nos momentos mais conturbados. Aprendi e cresci muito com cada um de vocês, muito obrigada por tudo.

Agradeço aos meus amigos que de alguma forma me ajudaram durante os dois anos de mestrado. Em especial, a Silviana e ao João, que me acompanharam de perto e por compartilharam comigo todas as alegrias e decepções que encontrei pelo caminho. À Elisane e Andréia minhas amigas, praticamente irmãs, que sempre ouviram todas as minhas queixas e contribuíram para o meu bem estar, dando a minha rotina momentos de muita alegria e a certeza de uma amizade sincera.

Por fim, agradeço também a todos aqueles que direta ou indiretamente contribuiram para a realização desse trabalho.

RESUMO

Esta dissertação apresenta a concepção e uma implementação de um sistema de alocação de computadores em um sistema distribuído baseado na ociosidade dos mesmos. Este sistema, chamado de Cadeo (Controle e alocação dinâmica de estações ociosas), tem a finalidade de simplificar a criação de aplicações paralelas que possam ser executadas em sistemas distribuídos. Ele oferece um modelo de programação simples, semelhante ao modelo de aplicações paralelas que executam em computadores com memória compartilhada.

A plataforma de execução do Cadeo é chamada de aglomerado dinâmico, que é um aglomerado (*cluster*) composto por computadores momentaneamente disponíveis, ou ociosos. A execução paralela se dá pelo lançamento concorrente de tarefas, implementadas por invocações assíncronas de métodos remotos em linguagem Java. O mapeamento de aplicações aos recursos reais se dá em dois níveis: no primeiro, associa-se aplicações a aglomerados dinâmicos, no segundo, associa-se tarefas de aplicações a computadores destes aglomerados. O Cadeo gerencia estas associações oferecendo transparência de localização de tarefas e da dinamicidade dos aglomerados, simplificando assim o desenvolvimento de aplicações.

A implementação foi feita com o objetivo de obter uma estrutura básica e funcional, capaz de suportar adaptações futuras. Não houve preocupação em incorporar os melhores algoritmos conhecidos para implementar as associações entre aplicações e aglomerados, ou tarefas a computadores. Entretanto, a versão implementada segue estritamente o modelo projetado, já permitindo escrever aplicações que tiram proveito da transparência e do assincronismo de lançamento segundo este modelo. Com a versão implementada, foi possível comprovar que a sobrecarga gerada pela utilização do sistema é muito pequena.

Palavras-chave: Java, programação paralela, dinamicidade, transparência.

Support to Control and Dynamic Allocation of Computers in Java

ABSTRACT

This thesis presents the project and an implementation of a distributed computer allocation system, based on computer idleness. The system, called Cadeo (control and dynamic allocation of idle workstations) aims to simplify the creation of parallel applications that can be executed on distributed systems. It supports a simple programming model, similar to the model found in parallel applications that run on shared memory computers.

The Cadeo's execution platform is called dynamic cluster, which is a cluster composed by momentarily avaliable computers, or idle computers. Parallel execution is obtained by concurrent task execution, implemented as asynchronous remote method invocations in Java. Mapping application onto resources happens in two levels, first, associating applications to dynamic clusters, and, second, associating application tasks to computers of such clusters. Cadeo manages these associations offering both transparent task location and cluster dynamism, thus simplifying application development.

The implementation was done with the purpose of having a basic and functional structure, which should be easily adapted in the future. This version was not focused on incorporating the best known algorithms to associate applications with clusters, or tasks with computers. Nevertheless, the implemented version strictly follows the proposed model, being now possible to write applications using the transparency and assynchrony accordingly. It was possible to demonstrate that, with the version implemented, the system overhead was very low.

Keywords: Java, parallel programming, dynamism, transparency.

LISTA DE FIGURAS

Figura 2.1: Modelo de um aglomerado de computadores 7
Figura 2.2: Modelo de uma rede de estações de trabalho 9
Figura 2.3: Modelo de uma grade de computadores 11

Figura 3.1: Modelo de uma aplicação implementada com MPI 24
Figura 3.2: Modelo de uma aplicação implementada com PVM 25
Figura 3.3: Modelo de uma aplicação implementada com DECK 27
Figura 3.4: Representação da estrutura do Globus 30
Figura 3.5: Legion: hierarquia de objetos básicos (CHAPIN et al., 1999) 32
Figura 3.6: Estrutura de escalonamento do Condor 33
Figura 3.7: Interface da *grid machine* e suas implementações (COSTA et al., 2004) . . 36
Figura 3.8: Componentes de um objeto ativo 42

Figura 4.1: Cenário do Sistema Cadeo 50
Figura 4.2: Notificação de ociosidade 52
Figura 4.3: Alocação de aglomerado dinâmico 53
Figura 4.4: Lançamento de Tarefas 54
Figura 4.5: Saída de um computador de um aglomerado dinâmico 55
Figura 4.6: Comunicação da ociosidade e destino do novo computador disponível . . . 62
Figura 4.7: Métodos invocados a partir do método idle( ) 64
Figura 4.8: Métodos invocados a partir do método busy( ) 65
Figura 4.9: Classe que imprime uma mensagem 71
Figura 4.10: Classe cliente da aplicação 72

Figura 5.1: Representação do modelo de programação mestre-escravo da aplicação paralela 77

Figura 5.2: Representação de (a) uma tarefa, (b) n tarefas consecutivas e (c) n tarefas consecutivas com concorrência m 79
Figura 5.3: Gráfico com as médias dos tempos de execução para a granularidade 1 . . . 82
Figura 5.4: Gráfico dos intervalos de tempos de execução sobre um ambiente dinâmico 86

LISTA DE TABELAS

Tabela 5.1: Tempos obtidos na estimativa de granularidades 81
Tabela 5.2: Médias dos tempos de execução com tarefas de diferentes granularidades . 83
Tabela 5.3: Médias dos tempos de execução em diferentes granularidades 84

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
AppLeS	*Application-Level Scheduling*
APST	*AppLeS Parameter Sweep Template*
ARMI	*Asynchronous RMI*
Cadeo	**C**ontrole e **a**locação **d**inâmica de **e**stações **o**ciosas
CPU	*Center Processing Unit*
DECK	*Distributed Execution and Comunication Kernel*
FIFO	*First In First Out*
FTP	*File Transfer Protocol*
GRAM	*Grid Resource Allocation Manager*
GRACE	*Grid Architecture for Computional Economy*
GRIP	*Grid Resource Information Protocol*
GSI	*Grid Security Infrastructure*
JIT	*Just In Time*
JNI	*Java Native Interface*
JPVM	*Java Parallel Virtual Machine*
JVM	*Java Virtual Machine*
LDAP	*Lightweight Diretory Access Protocol*
MARS	*Metacomputer Adaptive Runtime System*

MPI *Message Passing Interface*

MPMD *Multiple Program, Multiple Data*

NOW *Networks of Workstations*

PAD Processamento de Alto Desempenho

PVM *Parallel Virtual Machine*

P2P *Peer-to-peer*

RMI *Remote Method Invocation*

SMP *Symmetric Multiprocessing*

SPMD *Single Program, Multiple Data*

UNICORE *UNiform Interface COmputer REsources*

URL *Uniform Resource Locator*

WQR *Work Queue with Replication*

XML *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 1

2 ARQUITETURAS PARALELAS ALVO 6

2.1 Motivação 6

2.2 Aglomerados de Computadores 7

2.3 Redes de Estações de Trabalho 8

2.4 Grades de Computadores 10

2.5 Aproveitamento da Ociosidade Computacional 14

2.5.1 Descoberta de Computadores 15

2.5.2 Distribuição e Localização de Tarefas 17

2.5.3 Escalonamento e Balanceamento de Cargas 18

2.6 Síntese 19

3 PROGRAMAÇÃO NAS ARQUITETURAS PARALELAS ALVO 21

3.1 Motivação 21

3.2 Bibliotecas de comunicação 22

3.2.1 MPI 22

3.2.2 PVM 24

3.2.3 DECK 26

3.3 Gerenciadores de Recursos 28

3.3.1 Globus 28

3.3.2 Legion 31

3.3.3 Condor 32

3.3.4 MyGrid 35

3.4 Utilização de Java em PAD . . . 38
3.4.1 Iniciativas de melhora de Desempenho . . . 38
3.4.2 A Biblioteca ProActive . . . 41
3.5 Síntese . . . 43

4 SISTEMA CADEO . . . 45
4.1 Motivação . . . 45
4.2 Apresentação do Sistema . . . 46
4.2.1 Idéia Geral . . . 47
4.2.2 Definições para o Sistema Cadeo . . . 47
4.3 Estrutura do sistema . . . 49
4.3.1 Funcionamento do sistema . . . 50
4.3.2 Módulos do sistema . . . 51
4.4 Implementação . . . 55
4.4.1 Decisões de Projeto . . . 56
4.4.2 Implementação dos Módulos . . . 60
4.4.3 Interação entre Cadeo e Aplicação Paralela . . . 69
4.5 Síntese . . . 72

5 AVALIAÇÃO . . . 76
5.1 Motivação . . . 76
5.2 Aplicação Desenvolvida . . . 76
5.2.1 Funcionamento da Aplicação . . . 77
5.2.2 Implementação da Aplicação . . . 78
5.3 Resultados . . . 80
5.3.1 Ambiente de execução . . . 80
5.3.2 Dados Coletados . . . 81
5.4 Síntese . . . 87

6 CONCLUSÃO . . . 89

REFERÊNCIAS . . . 94

APÊNDICE A CLASSE ALLOCATOR . . . 103

APÊNDICE B CLASSE DYNAMICCLUSTER 106

APÊNDICE C CLASSE WORKER 109

APÊNDICE D CLASSE SCHEDULER 112

APÊNDICE E CLASSE CADEO 116

1 INTRODUÇÃO

As tecnologias de *hardware* de computadores passaram por grandes evoluções nos últimos anos e continuam evoluindo. A cada nova geração de computadores, surgem máquinas cada vez mais potentes. Mesmo assim, ainda existe uma demanda por poder de processamento não atingida. Problemas que envolvem uma quantidade muito grande de processamento cujo tempo gasto em sua solução seria inviável. Como opção para atender grandes demandas de processamento podem ser usados computadores especialmente arquitetados. Esses computadores possuem um conjunto de processadores que trabalham cooperativamente compartilhando uma mesma memória (multiprocessadores) (WILKINSON; ALLEN, 1999). Porém, por serem produzidos em baixa escala, esses computadores acabam possuindo um custo de aquisição consideravelmente elevado, tornando essa solução impraticável em muitos casos.

Uma alternativa aos multiprocessadores e que são economicamente mais acessíveis são as arquiteturas paralelas com memória distribuída (multicomputadores). Nesse tipo de arquitetura, computadores são interligados e passam a trabalhar juntos apresentando uma imagem única do sistema, ou seja, um conjunto de computadores trabalham como se fossem um só computador. Este modelo de arquitetura paralela pode ser generalizado como sendo um sistema distribuído (TANENBAUM; STEEN, 2002). Os multicomputadores oferecem às aplicações grande poder de processamento resultante da agregação de computadores, possibilitando execuções mais rápidas. Um importante fator relacionado a essas arquiteturas é sua grande escalabilidade. É possível alterar o conjunto de computadores que integra a arquitetura a fim de proporcionar variações na capacidade de processamento para suprir as demandas existentes. Essas arquiteturas, em sua maioria, são compostas por computadores comuns, de fácil aquisição, e também pela sua estrutura flexível, permitindo inclusão ou exclusão de componentes.

Agregar recursos computacionais convencionais é mais barato que fazer investimentos em

dispositivos eletrônicos[1] ou *hardware* para conseguir grande quantidade de processamento. Por se tratar de uma solução economicamente atrativa entre outros fatores, as arquiteturas paralelas com memória distribuída têm evoluído constantemente. Essa evolução pode ser vista a partir de iniciativas como os aglomerados de computadores, passando pelas redes de estações de trabalho até as almejadas grades de computadores. Embora algumas questões ainda dificultem o uso desses sistemas, tais como a carência de *softwares* apropriados, problemas com desempenho e segurança de rede, ainda assim estes se mostram como uma boa alternativa. Este trabalho tem seu foco em arquiteturas paralelas com memória distribuída.

Aplicações paralelas e distribuídas tiram proveito do poder de processamento das arquiteturas paralelas. Porém, programar paralelo é uma tarefas mais complicada que a programação seqüencial. Um programa paralelo pode ser obtido pelo emprego de compiladores paralelizadores ou então pela utilização de um modelo de programação paralelo. Os compiladores paralelizadores possibilitam que um programa seqüencial seja compilado em um programa paralelo. Este tipo de compilador possibilita o paralelismo em nível de dados e nem sempre atende as necessidades da aplicação. A segunda alternativa, que costuma ser mais empregada, deixa o controle do paralelismo por conta do programador. Os modelos de programação paralela possibilitam a paralelização de procedimentos das aplicações. Ou seja, desde de que não hajam dependências de dados, procedimentos que seriam executados seqüencialmente passam a ser executados em paralelo, melhorando o desempenho da aplicação.

O modelo de programação paralela mais utilizado em arquiteturas com memória distribuída é o modelo com trocas de mensagens. Nesse modelo, as aplicações paralelas são compostas por um conjunto de processos distribuídos entre os computadores da arquitetura paralela. Esses processos possuem os procedimentos necessários para a execução da aplicação paralela e estabelecem trocas de mensagens para atualização de dados e sincronização. As questões que envolvem o modelo de programação empregado e o aproveitamento das potencialidades de arquiteturas paralelas com memória distribuída são focos de muitas pesquisas. A área de Processamento de Alto Desempenho (PAD) contempla essas pesquisas. Os estudos realizados possibilitaram o surgiram várias iniciativas que oferecem condições para o desenvolvimento de *softwares* para essas arquiteturas. Um exemplo desse tipo de iniciativa foi a elaboração de bibliotecas de comunicação. Essas bibliotecas facilitam a implementação de aplicações paralelas pois oferecem primitivas que viabilizam a comunicação necessária. Embora se encarreguem da

[1] *chips*

comunicação, a localização dos computadores presentes na arquitetura não é transparente. Com tal transparência seria possível implementar aplicações paralelas de forma simples e intuitiva.

Conforme a dimensão das arquiteturas paralelas com memória distribuída aumenta, começam a surgir problemas relacionados ao gerenciamento do grande número de recursos envolvidos. Para facilitar o controle de arquiteturas de grande porte foram desenvolvidos os sistemas gerenciadores de recursos. Esses sistemas disponibilizam recursos aplicando políticas de escalonamento e balanceamento de cargas. Além de se fazer necessário o gerenciamento dos computadores, também são buscados meios para extrair todo o potencial que esse tipo de arquitetura paralela tem a oferecer. Por exemplo, o aproveitamento de computadores que estejam com baixa carga ou sem realizar processamento para realizar algum tipo de processamento útil. O sistema Condor (LITZKOW; LIVNY; MUTKA, 1988) é um exemplo dessa iniciativa, onde o sistema disponibiliza computadores ociosos de arquiteturas paralelas para a execução de aplicações que necessitem de recursos. Um outro exemplo de iniciativa é a aplicação seti@home (ANDERSON et al., 2002; ANDERSON, 2004) que ocupa computadores ociosos do mundo todo através da Internet. A finalidade dessa aplicação é analisar dados em busca indícios de vida extraterrestre.

Atualmente, a programação orientada a objetos vem sendo empregada em PAD visando uma maior produtividade e facilidade na implementação de aplicações paralelas. As linguagens orientadas a objetos possuem vantagens atrativas, uma vez que permitem a construção de tipos abstratos, reusabilidade de código, herança e polimorfismo. Entre essas linguagens, Java tem se destacado por ser uma linguagem simples e portável, tornando-se popular e amplamente utilizada. Além disso, Java possui, nativamente, suporte a programação paralela e concorrente com múltiplos fluxos de execução (*multithreading*) e com memória distribuída. As características inerentes de Java têm contribuído para sua difusão e aceitação como uma alternativa para o desenvolvimento de aplicações para arquiteturas paralelas com memória distribuída.

Porém, as vantagens da linguagem Java acarretam em sobrecarga no tempo de execução de aplicações quando comparada ao uso de outras linguagens. Uma série de esforços vem sendo realizados a fim de amenizar tal sobrecarga e os avanços têm sido promissores. As facilidades e vantagens obtidas pelo uso de Java, para a maioria dos casos, compensam o desempenho inferior obtido na execução das aplicações. A invocação de métodos nativos e compiladores JIT (*Just In Time*) são exemplos de iniciativas para a melhora do desempenho de aplicações Java. Também há iniciativas que primam por oferecer meios para a construção de aplicações

Java mais eficientes. Uma dessas iniciativas é a biblioteca Java para programação paralela e distribuída ProActive (CAROMEL; KLAUSER; VAYSSIERE, 1998). Essa biblioteca oferece invocação de métodos assíncrona, objetos futuros, espera por necessidade, migração entre outras características empregadas na implementação de aplicações paralelas e distribuídas.

Embora já se tenha atingidos grandes avanços, ainda se fazem necessários investimentos em *softwares* capazes de facilitar e automatizar a programação em arquiteturas paralelas com memória distribuída. Dentro desse contexto se insere o sistema Cadeo (**C**ontrole e **a**locação **d**inâmica de **e**stações **o**ciosas). O Cadeo é um sistema em Java e que possibilitá implementar aplicações paralelas e distribuídas de forma simples e intuitiva. O sistema se encarrega de disponibilizar e gerenciar computadores de forma totalmente transparente às aplicações. A plataforma de execução gerenciada pelo Cadeo concilia computadores provenientes de diferentes tipos de arquiteturas paralelas com memória distribuída.

A disponibilização dos computadores à plataforma de execução do Cadeo pode ocorrer de maneiras diferentes e depende diretamente das características das arquiteturas envolvidas. As formas de disponibilização podem ser por alocação, por compartilhamento ou através da detecção da ociosidade de computadores. Por exemplo a plataforma de execução pode possuir um aglomerado de computadores alocado ao sistema, computadores compartilhados em uma rede ponto-a-ponto ou computadores ociosos de uma rede ou grade. Esse modelo de aceitação de recursos faz com que o Cadeo possua uma grande abrangência e possa ser utilizado com as principais arquiteturas paralelas com memória distribuída.

A plataforma de execução do Cadeo é dinâmica pois o conjunto de computadores disponíveis tende a variar constantemente em função das formas de disponibilização de computadores. Essa plataforma foi denominada de aglomerado dinâmico e sua dinamicidade é totalmente controlada, de forma transparente, pelo Cadeo, que também se encarrega do escalonamento dos computadores. Além de oferecer transparência da dinamicidade do aglomerado dinâmico o sistema também deixa transparente a localização dos computadores. Através dessas duas características o Cadeo fornece um forma simples de implementar aplicações paralelas já que livra o programador do cuidado com a localização dos computadores. Isso implica que, a implementação de uma aplicação paralela e distribuída passa a assemelhar-se com o modelo de implementação de aplicações multiprocessadas.

O sistema Cadeo como um todo, envolve uma série de questões que influenciam diretamente o seu funcionamento, principalmente as questões que envolvem o escalonamento de computado-

res e tarefas. Por questões de tempo, foram fixados objetivos específicos para o trabalho que será apresentado no decorrer desse texto. Primeiramente, objetivou-se definir como seria a estrutura do Cadeo e o modelo de programação que seria empregado. Após, buscou-se, para uma primeira versão do Cadeo, a implementação de uma estrutura básica para o sistema que permitisse a implementação de aplicações paralelas de forma simples. Essa estrutura deve conseguir suportar um ambiente de execução dinâmico e oferecer transparência de localização e de dinamicidade dos computadores. Devido à complexidade dos aspectos que envolvem um sistema desse tipo, algumas questões, principalmente às referentes ao escalonamento e ao balanceamento de cargas, receberam soluções *ad hoc*. Não foi realizado um estudo detalhado para cada uma das questões, considerando as características da plataforma de execução e do sistema, para que as soluções mais apropriadas fossem encontradas. Porém, durante o desenvolvimento da estrutura básica do sistema, tomou-se o cuidado de construí-la adaptável, assim, todas as soluções *ad hoc* poderão ser facilmente substituídas futuramente sem que seja necessário alterações em sua estrutura.

O restante desse texto está dividido da seguinte forma. O próximo capítulo apresenta um levantamento das principais arquiteturas paralelas com memória distribuída, alvo do Cadeo, e iniciativas para o melhor aproveitamento dos computadores dessas arquiteturas. O capítulo seguinte descreve ferramentas utilizadas na programação das arquiteturas alvo do Cadeo. Após, tem-se o capítulo que descreve o sistema Cadeo, onde é apresentado a idéia geral, a estrutura proposta e como se deu a implementação da primeira versão do sistema. Em seguida, tem-se o capítulo destinado a avaliar o comportamento do sistema considerando as decisões tomadas na implementação de sua primeira versão. E por fim tem-se a conclusão onde são ressaltados os principais aspectos mencionados no decorrer desse texto.

2 ARQUITETURAS PARALELAS ALVO

2.1 Motivação

De modo geral, o principal objetivo de arquiteturas paralelas é oferecer grande quantidade de processamento para obter alto desempenho. Um dos primeiros exemplares de arquiteturas paralelas a serem construídos e utilizados foram os supercomputadores. O *hardware* dos supercomputadores são especialmente planejados e integram um conjunto de processadores que acessam uma memória em comum. Os supercomputadores também são conhecidos como multiprocessadores ou arquiteturas paralelas com memória compartilhada. Por essa arquitetura fazer uso de *hardware* específico e com produção em baixa escala, os custos que envolvem a aquisição e manutenção dos supercomputadores é bastante elevado.

Alternativamente, notou-se que computadores paralelos podiam ser construídos com elementos processadores sem memória compartilhada. Hoje em dia, a maioria dos computadores paralelos é algum tipo de aglomerado de computadores (BAKER; BUYYA, 1999), onde computadores comuns são conectados por uma rede de alto desempenho. Existindo uma certa flexibilidade pode-se generalizar este modelo de computador como sendo um sistema distribuído (TANENBAUM; STEEN, 2002). Uma importante vantagem desse tipo de arquitetura paralela é que o custo envolvido é bastante inferior ao de supercomputadores. Isso porque, seus componentes são facilmente encontrados e produzidos em larga escala.

O sistema Cadeo disponibiliza em sua plataforma de execução computadores provenientes de arquiteturas paralelas com memória distribuída. Este capítulo descreve, brevemente, as arquiteturas paralelas alvo para o Cadeo. Primeiramente serão apresentados os aglomerados de computadores e suas principais características. Em seguida serão mostradas as redes de estações de trabalho e por fim as grades de computadores.

2.2 Aglomerados de Computadores

Os aglomerados de computadores (BAKER; BUYYA, 1999) são o tipo de arquitetura voltada a computação paralela mais difundida atualmente. Um aglomerado consiste de um conjunto de computadores independentes interligados por uma rede de interconexão de alta velocidade e que oferecem uma visão única do sistema (BUYYA, 1999; PFISTER, 1998). Uma representação desse tipo de sistema pode ser vista na figura 2.1, onde o *middleware* é o software que dá a visão única aos programas em execução em cada nó.

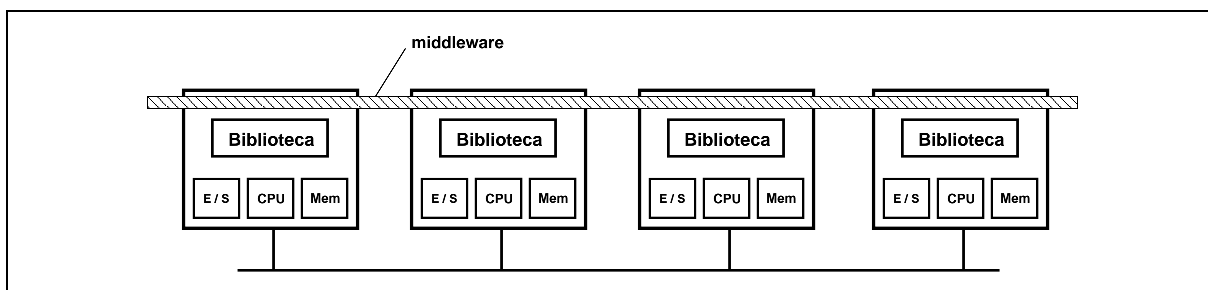

Figura 2.1: Modelo de um aglomerado de computadores

A popularidade dos aglomerados de computadores decorre, principalmente, da sua favorável relação custo/benefício. Um aglomerado pode ser composto por um conjunto de computadores de qualquer tamanho, até mesmo com computadores pessoais, interligados por uma rede de alta velocidade. Assim, é possível compor um aglomerado capaz de atingir níveis de desempenho similares aos oferecidos por supercomputadores, com um custo de aquisição consideravelmente inferior. Outro fator relevante é a sua escalabilidade. Teoricamente, havendo a necessidade de ampliar o desempenho de um aglomerado, bastaria agregar mais computadores ao mesmo. A composição do sistema torna possível adicionar ou remover computadores ao sistema sem afetar sua estrutura de funcionamento. O tamanho do sistema seria diretamente proporcional a demanda por desempenho e ao montante de investimento disponível a aquisição de recursos.

Os aglomerados de computadores foram projetados com o intuito de fornecer grande quantidade de processamento. Por esse motivo, os aglomerados apresentam otimizações capazes de melhorar o desempenho de seus recursos. Os computadores que integram um aglomerado, geralmente, não possuem periféricos de interface humana (monitor, mouse, teclado, etc) uma vez que seu uso é dedicado. Além disso, otimizações de *software* também são realizadas, onde o sistema operacional pode ser simplificado, não havendo a necessidade de se ter certos tipos de serviços habilitados já que os computadores terão um fim específico. Também em virtude do

uso dedicado, algumas camadas de rede podem ser simplificadas e até mesmo eliminadas. Essas otimizações melhoram o desempenho do sistema e é o principal diferencial dos aglomerados de computadores (DE ROSE; NAVAUX, 2003).

Em geral, os computadores que integram um aglomerado são todos da mesma arquitetura, ou seja, um aglomerado é composto por computadores e dispositivos homogêneos. Porém, a utilização de aglomerados compostos por recursos heterogêneos vem aumentando e sendo alvo de muitos estudos (AUMAGE, 2002; KREUTZ; CERA; STEIN, 2004). Isso porque um aglomerado heterogêneo proporciona o aproveitamento dos recursos computacionais existentes, já que a diversidade é uma de suas características. Barreiras tecnológicas podem ser transpostas uma vez que os componentes de aglomerados evoluem rápida e constantemente. Entretanto, para obter um melhor aproveitamento do sistema, faz-se necessário o uso de ferramentas que ofereçam uma interface entre os diferentes dispositivos do sistema. Também é importante existir políticas eficientes que amenizem os efeitos da diversidade do sistema no escalonamento e balanceamento das cargas de trabalho.

Mesmo que uma das principais características dos aglomerados seja oferecer uma imagem única do sistema, o modelo de programação empregado nesses sistemas não é transparente. Isso implica que a localização das tarefas de uma aplicação paralela deve ser conhecida. Buscando facilitar a elaboração dos programas paralelos, foram propostas diferentes ferramentas de programação para aglomerados. Entre elas, MPI (*Message Passing Interface*) (GROPP; LUSK; SKJELLUM, 1994) se destaca por ser a bibliotecas de comunicação mais utilizada e completa.

2.3 Redes de Estações de Trabalho

As redes de estações de trabalho (*Networks of Workstations* - NOW) (POLLATOS; CANDLIN, 1996; ANDERSON; CULLER; PATTERSON, 1995) são uma opção de arquitetura paralela com memória distribuída. Elas disponibilizam, de maneira simples e prática, grande quantidade de memória, disco e processamento. As NOW são compostas por computadores tradicionais interligados por uma tecnologia de rede tradicional (DE ROSE; NAVAUX, 2003). A idéia da sua concepção propunha criar máquinas paralelas simplesmente através do uso da programação simultânea e ordenada de computadores pessoais (estações de trabalho), usando a rede que já os interligava. Nela, as tarefas que compõem uma aplicação paralela são distribuídas entre as estações de trabalho, possibilitando sua execução de forma concorrente. O auge das pesquisas no desenvolvimento de NOW ocorreu na década de 90, mas este tipo de sistema

é utilizado e pesquisado ainda nos dias atuais (ORSOLETTA et al., 2003). Um modelo que representa esse tipo de ambiente pode ser visto na figura 2.2.

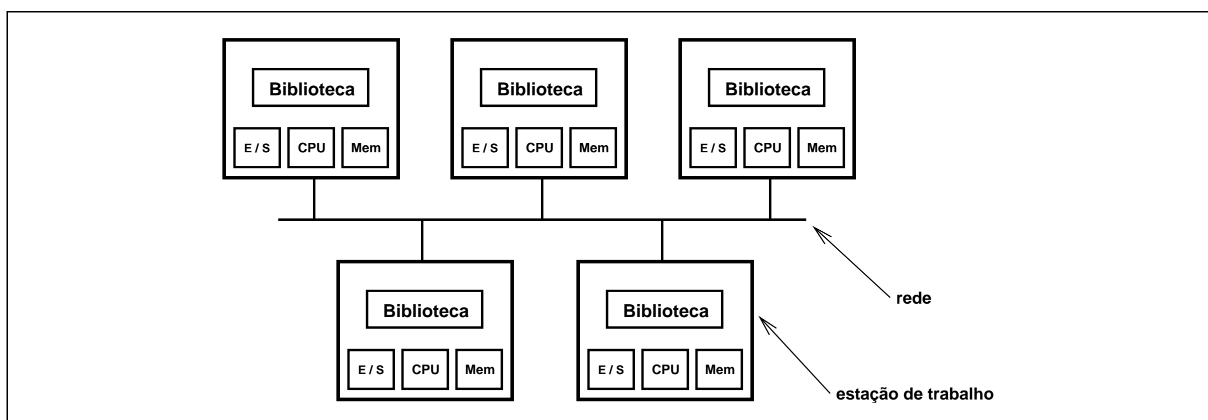

Figura 2.2: Modelo de uma rede de estações de trabalho

Uma NOW pode vir a oferecer taxas de desempenho similares às obtidas por supercomputadores (CULLER et al., 1997). Porém, o custo para a construção de uma NOW é significativamente inferior à aquisição de um supercomputador. Supercomputadores são arquiteturas especialmente desenvolvidas o que torna seu *hardware* caro e com baixa demanda de mercado. Já as NOWs são construídas a partir de computadores comuns, produzidos em larga escala, o que as tornam mais acessíveis. Outra vantagem é a escalabilidade de uma NOW. Ela pode ser facilmente ampliada com a adição de novas estações, também denominadas de nós, ao sistema.

As NOW possuem a desvantagem que tanto as estações de trabalho quanto a rede de interconexão não são dedicadas a execução de aplicações paralelas. Os recursos estão propensos a sofrerem interferência de fatores externos às tarefas de uma aplicação paralela. Outras aplicações (principalmente aquelas do dono da estação) podem utilizar os recursos e assim diminuir o desempenho final da aplicação. Essa característica decorre do fato que redes de estações de trabalho não foram concebidas para executar aplicações paralelas e sim para compartilhar recursos. Por este motivo, as redes não possuem uso dedicado nem as mesmas otimizações presentes nos aglomerados de computadores. Outro aspecto é que a administração dos nós que integram o sistema é realizada localmente pelo seu dono. Dessa forma, o controle da execução dos processos nas estações tem que passar pelas diferentes restrições administrativas de cada uma das estações do sistema.

As características das redes de estações de trabalho conduziram a utilização de seus recursos enquanto os mesmos encontram-se em ociosidade. A utilização de computadores com baixa ou

nenhuma carga de processamento possibilita um melhor aproveitamento dos recursos. Para tanto, faz-se necessária uma infra-estrutura, via *software* , que possibilite o gerenciamento e disponibilização dos computadores. As questões que envolvem o aproveitamento de recursos computacionais ociosos serão apresentadas detalhadamente na seção 2.5.

O modelo de programação a ser seguido na elaboração das aplicações paralelas para NOW não é trivial. Durante a implementação, o programador deve preocupar-se com a localização das tarefas, ou seja, o programador deverá saber onde e em quais estações serão lançadas as tarefas da aplicação. Para facilitar a confecção de programas paralelos para essa arquitetura, o programador tem à disposição bibliotecas de comunicação que implementam primitivas de comunicação entre as estações. Como exemplos desse tipo de biblioteca podemos citar MPI e PVM (*Parallel Virtual Machine*) (SUNDERAM, 1990). Estas bibliotecas, além de fornecer um conjunto amplo, otimizado e padronizado de funções de comunicação, possuem mecanismos para o lançamento remoto de processos nos nós da rede.

2.4 Grades de Computadores

As grades de computadores (FOSTER; KESSELMAN, 2003; BERMAN; HEY; FOX, 2003) são a mais recente alternativa de arquitetura para execução de programas paralelos. O princípio básico das grades é a integração de redes de interconexão, comunicação, computação e informação provendo uma plataforma virtual para computação e gerenciamento de dados da mesma forma que a Internet integra recursos em uma plataforma virtual de informação. Através delas seria possível usufruir de recursos computacionais localizados em qualquer parte do planeta, e assim, ter a disposição um poder de processamento e armazenamento potencialmente tão grande quanto se queira (BERMAN; FOX; HEY, 2003). Os recursos computacionais das grades abrangem uma gama bastante variada, indo de computadores pessoais, aglomerados e supercomputadores até equipamentos específicos, tais como sensores, microscópios, entre outros. Um modelo representando esse tipo de arquitetura pode ser visto na figura 2.3.

Para entender melhor a idéia de grade de computadores podemos fazer uma analogia com outra forma de infraestrutura mundialmente dispersa: a rede elétrica (CHETTY; BUYYA, 2002). A rede elétrica oferece energia para seus usuários e estes a utilizam conforme as suas necessidades. Não importa onde vá, qualquer pessoa nos dias de hoje espera poder extrair uma quantidade razoável de energia de qualquer tomada do planeta (por exemplo, recarregar a bateria de seu celular). No contexto dos computadores, os usuários da grade teriam disponível poder de

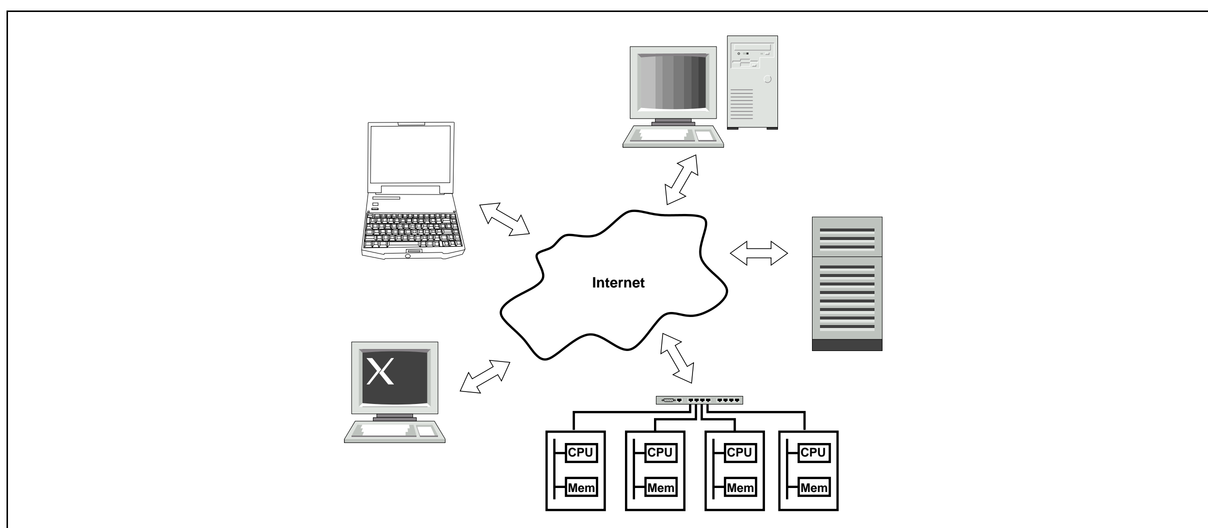

Figura 2.3: Modelo de uma grade de computadores

processamento em quantidade suficiente para atender suas demandas.

As grades de computadores apresentam algumas características inerentes à natureza do sistema onde destaca-se heterogeneidade, distribuição e dinamismo. Devido à potencial escala planetária do sistema, seus recursos possuem arquiteturas heterogêneas entre si e podem estar dispersos pelo mundo configurando uma alta dispersão geográfica. Em uma grade, em função das suas proporções, não é interessante que todo o sistema seja de uso dedicado a uma única aplicação, logo, compartilhamento é outra das suas características. Também faz parte de suas características a capacidade de congregar vários domínios administrativos (acesso e autenticação nesses domínios), assim como o controle distribuído da grade uma vez que ela pode ser mantida por mais de uma entidade, cada uma com poderes sobre partes dela.

A idéia de congregar os recursos computacionais dispersos pelo planeta em um ambiente de grade é bastante atrativa. Com essa infraestrutura é possível atender demandas de pequeno a grande porte (uma vez que o sistema é de larga escala) de forma simples e econômica. O modelo para o acesso ao poder de processamento da grade é simples e semelhante ao modelo existente na rede elétrica, ou seja, ao se requerir processamento a grade teria condições de suprir a demanda. A fim de garantir que as grades serão estáveis e que promoverão desempenho a todos os seus usuários vem sendo estruturadas propostas de economias de grade (*grid economy*) (BUYYA; ABRAMSON; GIDDY, 2000a; WOLSKI et al., 2003). Essas economias forneceriam, principalmente, mecanismos para as negociações entre consumidores e fornecedores e formas seguras para o pagamento pelo processamento obtido. Embora ainda não se tenha estabelecido um pa-

drão para as negociações, com certeza a forma de pagamento pelo processamento será mais econômica que a aquisição dos recursos necessários para obter o processamento desejado.

As grades também contribuem para solucionar problemas que demandam grande quantidade de processamento e armazenamento em sua solução, chamados de problemas de grande desafio (*"Grand Challenge"*). Nos dias atuais a ciência baseia-se fortemente na computação, análise de dados e na colaboração e troca de experiências entre diferentes centros de pesquisa (FOSTER, 2002). Como exemplos de aplicações que são favorecidas pelo ambiente de grade tem-se simulações do comportamento de complexas estruturas moleculares frente a vários tipos de drogas, simulação e modelagem do comportamento financeiro provendo previsões para a bolsa de valores, busca de dados em bases com grande quantidade de informações distribuidamente armazenadas, entre outros.

A aplicação mais conhecida que executa sobre a filosofia de grade é o SETI@home (ANDERSON et al., 2002; ANDERSON, 2004) onde milhões de computadores ligados à Internet analisam dados em busca de indícios de vida extraterrestre. Seguindo esse modelo de aplicação temos o Parabon cciteparabon que, também, utiliza o poder computacional de máquinas ligadas à Internet, o qual testa drogas contra o câncer entre outras doenças. Além dessas duas citadas anteriormente, existem uma série de aplicações que utilizam recursos ligados à Internet enquanto estes não estão sendo utilizados por seus proprietários (ver seção 2.5). Ainda merece destaque o Entropia (CHIEN et al., 2003; SMARR, 2004) que é um sistema capaz de fornecer processamento, onde a integração é feita através de servidores de recursos dentro de uma grade empresarial (*enterprise grid*). Uma grade empresarial é composta por todo o poder de processamento existente dentro da empresa (de grandes e pequenos servidores e aglomerados de computadores em centros de dados até todos os computadores pessoais conectados via rede) trabalhando conjuntamente para solucionar um problema (ANDRES, 2004).

Embora muito se deseje e espere das grades de computadores, muitos aspectos ainda não permitem que elas sejam uma realidade. O que já existe e vem encontrando bons resultados é a interação entre centros de pesquisa geograficamente dispersos para a solução de um mesmo problema (BUNN; NEWMAN, 2003; BECKMAN, 2004; JOHNSTON, 2004; JACKSON, 2004). Além de compartilharem seus recursos físicos e obterem resultados em tempo menor que se tentassem isoladamente, os cientistas trocam experiências e juntos conseguem melhor analisar os dados obtidos. Com a ajuda das grades de computadores, a ciência tem conseguido evoluir e solucionar importantes problemas, o que favorecem a toda humanidade.

Dentro da expectativa e esforços para se ter um ambiente de grade funcional podemos citar projetos como o Globus (FOSTER; KESSELMAN, 1998), Legion (GRIMSHAW; WULF; TEAM, 1997), NetSolve (AGRAWAL et al., 2003), UNICORE (*UNiform Interface COmputer REsources*) (ERWIN; SNELLING, 2001), Ninf (TANAKA et al., 2003), MyGrid (CIRNE et al., 2003), GRACE (*Grid Architecture for Computional Economy*) (BUYYA; ABRAMSON; GIDDY, 2000a) entre outros. Esses projetos buscam oferecer gerenciamento dos recursos das grades possibilitando seu uso como plataforma de execução. De modo geral, todas as iniciativas buscam viabilizar o compartilhamento de recursos e informações, segurança, autonomia, transparência de programação e tolerância a falhas.

Um ambiente de grade é envolvido por uma série de aspectos que influenciam o desenvolvimento de sistemas para o seu gerenciamento. Como por exemplo: quem são os recursos que integram a grade, como será realizada a distribuição deles entre os usuários e quanto custará a eles a disponibilização dos recursos. Por se tratar de uma área de pesquisa recente, foram planejados uma série de sistemas e cada qual oferece uma abordagem diferenciada para solucionar o problema. Por exemplo, o MyGrid e o Legion oferecem soluções que aproveitam-se das vantagens da orientação a objetos para viabilizar o uso das grades. Além da diferenciação de paradigmas de linguagem, esses sistemas também exploram diferentes metodologias para o tratamento dos recursos e das questões que envolvem as grades.

Outro aspecto é que esses sistemas foram direcionados a resolver problemas específicos. Com esse direcionamento do trabalho foi possível encontrar soluções mais eficientes. Porém, para que os sistemas possam ser empregados na prática, os serviços que lhes faltam devem ser oferecidos por outros e assim, tem-se uma dependência entre eles. Por exemplo, o Globus oferece importantes características como autenticação, autorização, comunicação segura, diretório de serviços, porém necessita que lhe seja oferecido o escalonamento dos recursos por algum outro sistema. O GRACE não disponibiliza recursos, os quais podem ser obtidos por intermédio do Globus, porém oferece escalonamento e um modelo econômico para a grade. Os sistemas Ninf e NetSolve, também podem ser utilizados conjuntamente com o Globus, oferecem uma implementação simples e eficiente de um modelo cliente-servidor chamado GridRPC (SEYMOUR et al., 2002). Enfim, cada sistema busca melhor solucionar determinadas carências das grades e a associação dos sistemas existentes pode gerar soluções mais completas e eficientes.

A diversidade de recursos de uma grade de computadores tende a ser muito grande em função da sua escala mundial. As aplicações que utilizarem esta arquitetura terão dificuldade

para obter um bom escalonamento de tarefas e balanceamento de cargas. Visando encontrar as melhores técnicas de escalonamento e balanceamento de cargas usando grades, vêm sendo desenvolvidos alguns projetos, tais como Condor (LITZKOW; LIVNY; MUTKA, 1988), AppLeS (*Application-Level Scheduling*) (BERMAN et al., 1996), APST (*AppLeS Parameter Sweep Template*) (CASANOVA et al., 2000), MARS (*Metacomputer Adaptive Runtime System*) (GEHRING; REINEFELD, 1996), Nimrod/G (BUYYA; ABRAMSON; GIDDY, 2000b) entre outros.

Um bom escalonamento e balanceamento de cargas é peça fundamental para que se possa tirar o melhor proveito de uma arquitetura heterogênea e dinâmica como as grades. É por esse motivo que existem uma série de projetos pesquisando alternativas para conseguir manter um equilíbrio de distribuição de cargas de processamento. Por exemplo, o AppLeS e sua continuação, o APST, mantêm um banco de informações contendo dados da aplicação, do sistema e dos recursos para, a partir dessas informações e do uso de predições, escalonar as cargas de processamento. O MARS também baseia-se em informações tanto da aplicação quanto dos recursos para determinar migrações que possibilitem redução no tempo de execução de aplicações. O problema, nesses casos, é que obter e manter as informações sobre os recursos dinâmicos da grade acaba sendo uma tarefa difícil. O Condor disponibiliza uma estrutura de escalonamento onde em um nível são escalonados recursos provenientes de um banco de recursos e em outro acontece o escalonamento das cargas de processamento de aplicações. Enfim, os projetos buscam, através de diferentes técnicas, contornar a dinamicidade e heterogeneidade para obter um melhor aproveitamento dos recursos da grade.

2.5 Aproveitamento da Ociosidade Computacional

Mantendo a linha que trata de arquiteturas paralelas com memória distribuída, será abordada uma outra opção para a disponibilização de computadores. Esta opção é baseada na taxa utilização (períodos de processamento) dos computadores. Ao se analisar os períodos de utilização de recursos computacionais, em modo geral, pode-se facilmente constatar períodos de pleno processamento, alternados por períodos de baixa utilização. Em outras palavras, durante alguns períodos o poder de processamento de um computador é totalmente utilizado pelos processos em execução, e em outros apenas uma pequena parcela do potencial computacional é utilizado. Em geral, os períodos de baixa utilização são predominantes (LITZKOW; LIVNY; MUTKA, 1988; KRUEGER; CHAWLA, 1991). Essa análise dos períodos de utilização pode ser estendida as arquiteturas paralelas descritas anteriormente.

Os computadores que encontram-se ociosos ou com baixa utilização, poderiam estar realizando alguma espécie de processamento útil durante esses períodos . Dessa forma seria possível obter um melhor aproveitamento do poder de processamento desses computadores. Também se teria disponível poder de processamento para atender a demanda de problemas de difícil solução, que necessitam de grandes quantias de processamento para que seja obtida solução em tempo hábil. Uma outra possibilidade para o uso de arquiteturas compostas por computadores ociosos é disponibilizá-los em uma plataforma de execução de aplicações distribuídas quaisquer. Pesquisas constataram que a utilização de computadores ociosos na execução de aplicações paralelas é relevante e oferece bons resultados (ACHARYA; EDJLALI; SALTZ, 1997; JU; XU; TAO, 1993).

O aproveitamento dos períodos de ociosidade dos computadores é uma possibilidade bastante atrativa que tem despertado o interesse em muitos projetos de pesquisa já hà alguns anos. Nessa seção serão apresentadas as questões de projeto inerentes ao desenvolvimento de sistemas que visam o aproveitamento de computadores ociosos. Para facilitar a compreensão, a seção foi dividida em questões referentes à descoberta de computadores, à distribuição e localização de tarefas e ao escalonamento e balanceamento de carga. Além das questões, serão mostradas algumas alternativas que buscam resolvê-las e que já foram empregadas em sistemas conhecidos na literatura.

2.5.1 Descoberta de Computadores

Ao se pretender fazer uso de computadores em estado de ociosidade, uma das primeiras questões que surgem é definir que situação ou estado de utilização de um computador pode ser considerado ociosidade. Isso implica em definir qual ou quais critérios serão avaliados na descoberta de computadores. Por exemplo, para algumas aplicações pode ser mais relevante ter à disposição grande quantidade de memória, já para outras uma alta disponibilidade de rede pode ser mais relevante. O ideal é tentar resolver esse problema fornecendo um método genérico para a descoberta de recursos ociosos onde seja possível atender o maior números de casos possíveis.

Além de definir quais componentes serão monitorados nos computadores potencialmente ociosos, também é necessário definir limites máximos de utilização para que seja constatada a ociosidade. Todos os computadores cujos componentes apresentarem utilização inferior a esse limite máximo serão considerados ociosos. Nesse ponto outra questão pode ser levantada, como definir o que pode ser considerado um bom limite máximo de utilização, ou seja, que

percentual de utilização pode ser considerado baixo? Por exemplo, restringindo-se ao critério de uso de CPU, se o limite máximo de utilização da CPU fosse 30%, aplicações que necessitassem até 70% do poder de processamento daquela CPU seriam favorecidas. Já para aplicações que necessitassem de mais de 70% do poder da CPU, um índice de até 30% não poderia satisfazê-las. Um outro aspecto é que nem sempre se tem disponível previsões de percentuais de necessidades das aplicações e, na maioria das vezes, tal necessidade envolve mais de um critério, o que dificulta a previsão de um limiar de utilização.

Para que se tenha meios para detectar a ociosidade de um computador deve ser realizada uma coleta de métricas de ocupação e em seguida um tratamento dos dados coletados. A partir desse ponto, o enfoque da questão passa a ser estipular a dimensão do período de ociosidade ou disponibilidade daquele computador. Isto é, buscar estimar por quanto tempo o computador estará disponível sendo que ele pode deixar sua condição de ociosidade a qualquer momento. Essa estimativa é importante para que se evite, por exemplo, de destinar tarefas muito demoradas para computadores que ficarãp disponíveis por um período muito curto de tempo. Existem uma série de algoritmos que procuram oferecer uma estimativa desse tempo de disponibilidade dos computadores (GOLDING et al., 1995). O princípio básico de funcionamento desses algoritmos é estimar o tempo de disponibilidade de um computador baseado na amostragem de seu comportamento em situações passadas. Ou seja, a previsão de disponibilidade é dada através da análise de dados coletados, os quais refletem o comportamento recente do computador. Outro aspecto da busca por determinar o tempo de disponibilidade dos computadores é analisar também o comportamento de seus usuários. Considerando uma rede local, nos períodos como madrugada e horário de almoço há uma maior probabilidade dos computadores permanecerem ociosos do que, por exemplo, durante a manhã.

Outro aspecto a ser planejado é o modo de representação dos recursos disponíveis. Ao se tornar ocioso ou disponível a realizar processamento, é preciso que as informações daquele computador sejam expostas de alguma forma. Preferencialmente, essa forma de representação, ou linguagem de descrição, deve ser maleável e passível a adaptações conforme as necessidades dos sistemas que as utilizam. Nesse sentido, existem iniciativas para representar as informações dos recursos fazendo uso de linguagens como XML (*Extensible Markup Language* (SCHAEFFER FILHO et al., 2004)). XML é uma linguagem que não possui um formato fixo já que sua concepção almejou fornecer maior flexibilidade e adaptabilidade na identificação de informações. Essa linguagem vem sendo empregada para representar recursos de sistemas em geral, e

pode ser empregada para o tipo de sistema em questão.

Além de se fazer necessária uma linguagem de descrição, a identificação de quais informações dos computadores deve ser armazenada é mais um aspecto a ser decidido. Normalmente, além de informações de *hardware* , como velocidade de CPU, espaço de memória e disco livres, entre outras, também são buscadas informações de *software* . Informações tais como: o sistema operacional e sua versão, tipo de *software* suportado, entre outras. Essas informações referentes aos computadores também são conhecidas como predicados.

Após detectada a ociosidade e existir uma representação das informações, os computadores precisam ser disponibilizados para as aplicações que carecem de processamento. Uma forma de disponibilizá-los pode ser com a sua inclusão em um banco de recursos (*pool of resources*). Sistemas que utilizam recursos ociosos tais como Condor, Butler (NICHOLS, 1987), AdJava (FUAD; OUDSHOORN, 2002), entre outros, mantém seus computadores disponíveis em bancos de recursos. Nesses sistemas, ao ser detectada a ociosidade, os computadores se inserem nos bancos de recursos e permanecem lá até que sejam solicitados por alguma aplicação ou até que deixem sua condição de disponibilidade.

Uma outra alternativa para a questão da forma de disponibilizar computadores em ociosidade é uma abordagem semelhante à oferecida pela arquitetura Jini (ARNOLD, 1999). A arquitetura Jini fornece uma infra-estrutura para definir, advertir e encontrar serviços em uma rede, onde os serviços são definidos por uma ou mais interfaces ou classes Java. Essa arquitetura permite que um serviço seja disponibilizado em uma rede onde alguém poderá lê-lo e executá-lo de forma segura e robusta. Tal abordagem implica em se manter um banco de serviços ao invés de um banco de recursos e sua idéia pode ser considerada como uma alternativa na definição de projeto de um sistema que se baseia em computadores que estejam disponíveis a executar tarefas.

2.5.2 Distribuição e Localização de Tarefas

Em sistemas que buscam aproveitar computadores em ociosidade para executar alguma espécie de processamento útil, uma de suas características marcantes é a alta dinamicidade dos computadores envolvidos. Um computador que está disponível (ocioso) em um determinado momento, pode deixar de estar disponível a qualquer instante. Para simplificar a implementação de aplicações para esse tipo de plataforma de execução, seria interessante que sua dinamicidade não fosse percebida pelos usuários. A fim de possibilitar essa simplificação, o lançamento e a distribuição de tarefas (cargas de processamento) deverá ocorrer de forma transparente. Em

outras palavras, existem tarefas a serem processadas e deseja-se que sejam executadas independente do local (computador) onde essa execução aconteça. Dessa forma, ficaria sob responsabilidade do sistema fornecer meios para realizar o lançamento e a distribuição de tais tarefas, sem que sua localização seja conhecida de antemão.

Em virtude da localização transparente proporcionada por estes sistemas, a implementação de aplicações destinadas a ambientes dinâmicos pode ser realizada exatamente como se a aplicação executasse localmente. Essa característica facilita o desenvolvimento das aplicações, uma vez que a complexa manipulação da dinamicidade dos computadores já está resolvida. Assim como o controle da dinamicidade dos recursos deve ficar a cargo do sistema, também deve ser sua responsabilidade garantir a execução de tarefas de computadores que deixam a condição de ociosidade. Isso possibilita que o sistema se adapte à perdas e ao recebimento de computadores, proporcionando até mesmo tolerância às falhas que possam ocorrer durante a execução.

Existem três principais formas de contornar a perda computadores e garantir que suas tarefas venham a ser executadas. Uma delas seria manter a execução da tarefa no computador não mais disponível, porém com uma prioridade baixa. Assim, as demais atribuições do mesmo não sofreriam grande interferência ou seriam comprometidas (JU; XU; TAO, 1993). Uma outra alternativa seria abortar a execução da tarefa no computador não mais disponível. Futuramente, a tarefa abortada seria relançada em algum computador que esteja disponível (NICHOLS, 1987). Por fim, poderiam ser registrados pontos de verificação (*checkpoints*) durante a execução da tarefa. Quando o computador não estivesse mais disponível, a tarefa poderia ser migrada para outro computador, onde sua execução continuaria a partir do último ponto de verificação registrado (LITZKOW; LIVNY; MUTKA, 1988).

2.5.3 Escalonamento e Balanceamento de Cargas

O escalonamento das tarefas nos computadores da plataforma de execução procura proporcionar o casamento entre as necessidades das tarefas e as especificações dos computadores (THAIN; TANNENBAUM; LINVY, 2003). Por exemplo, se uma determinada tarefa faz uso intensivo de CPU ela, provavelmente, oferecerá melhor desempenho se for executada em um computador que apresente um processador mais potente. Nesse ponto, vê-se a importância da forma de descrição das especificações dos computadores (predicados) para que a busca por um computador apropriado para uma tarefa seja eficiente.

Outra questão é como identificar as necessidades de uma tarefa. Uma alternativa é o propri-

etário da aplicação fornecer previamente tais informações já que ele conhece as necessidades da mesma. Porém essa alternativa tornaria o escalonamento dependente da precisão da descrição fornecida pelo usuário ou programador. Uma forma de solucionar possíveis falhas ocorridas durante o escalonamento das tarefas é através do emprego de políticas de balanceamento de cargas. Para este caso, após o inicio da execução das tarefas, as cargas de processamento poderiam ser ajustadas a fim de oferecer um melhor desempenho.

Como planeja-se construir uma plataforma de execução aproveitando computadores que não estejam sendo utilizados em suas atividades usuais, isso implicará em uma plataforma heterogênea. Em virtude de tal heterogeneidade, o emprego de políticas de escalonamento e balanceamento de cargas se fazem necessárias para obter um melhor desempenho da arquitetura paralela. O desempenho de uma aplicação será diretamente relacionado à qualidade da distribuição das suas tarefas nos computadores que estão disponíveis a sua execução. A busca por amenizar o efeito da heterogeneidade através de algoritmos de escalonamento e balanceamento de cargas também é alvo de interesse em sistemas como as grades de computadores. As soluções encontradas para as grades também se aplicam a sistemas que se utilizam de computadores ociosos (BERMAN, 1999).

O balanceamento de cargas torna possível controlar a sobrecarga ou baixa carga dos computadores disponibilizados para uma determinada aplicação. Os algoritmos de balanceamento buscam proporcionar o equilíbrio de carga de trabalho entre os computadores existentes e assim reduzir o tempo de execução de aplicações através de uma distribuição de tarefas mais apropriada (BOZYIGIT, 2000). Tal equilíbrio é obtido através da identificação dos computadores com sobrecarga de processamento onde algumas de suas tarefas são destinadas a outros computadores que estejam menos sobrecarregados. Através do balanceamento de cargas é possível contornar problemas ocorridos durante o escalonamento como por exemplo problemas de precisão das necessidades das tarefas. O balanceamento também é importante porque possibilita correções de carga, uma vez que, devido a dinamicidade dos computadores, o equilíbrio de cargas entre eles pode sofrer alterações.

2.6 Síntese

Esse capítulo destinou-se a apresentar as arquiteturas paralelas com memória distribuídas que serão alvo do sistema Cadeo. Inicialmente foram descritos os aglomerados de computadores, os quais são bastante difundidos atualmente e empregados em muitos centros de pesquisas

e empresas. Essa arquitetura foi especialmente construída para processar aplicações paralelas e oferecer alto desempenho. Após, foram apresentadas as redes de estações de trabalho, as quais disponibilizam computadores convencionais interligados via rede como uma arquitetura paralela. Essa é uma alternativa mais econômica, uma vez que aproveita computadores comuns existentes em uma rede. Uma terceira arquitetura paralela foi apresentada a seguir, as grades de computadores. As grades, apesar de ainda não serem uma realidade plena, são tidas como o futuro das arquiteturas paralelas. Através das grades será possível congregar diferentes tipos de arquiteturas paralelas e recursos computacionais num sistema que oferecerá poder de processamento e armazenamento potencialmente infinito.

Atualmente, as tendências das pesquisas e o processo evolutivo apostam nas grades de computadores como o futuro das arquiteturas paralelas. Muitos estudos vem sendo realizados com o intuito de viabilizar o ambiente de grades que envolve muitos aspectos, principalmente em função de sua grande escala, heterogeneidade e dinamicidade dos recursos. Tais estudos, em sua maioria, buscam oferecer facilidade de programação, melhor aproveitamento dos recursos, segurança, eficiência de comunicação, entre outros.

Outro aspecto que envolve as arquiteturas paralelas com memória distribuída é quanto a busca pelo melhor aproveitamento dos recursos por elas oferecido. Nesse capítulo também foi apresentado uma maneira de melhor aproveitar o poder de processamento através do uso de computadores em estado de ociosidade. Dentro desse contexto, existem projetos que aproveitam períodos de ociosidade dos computadores de sistemas distribuídos disponibilizando ciclos de processamento a aplicações que os necessitem. A quarta seção desse capítulo destinou-se a detalhar essa iniciativa. Nela foram apresentadas as principais características desses sistemas, tais como heterogeneidade e dinamicidade entre outras. Também foram expostas as questões de projeto que envolvem o desenvolvimento de sistemas capazes de disponibilizar recursos ociosos.

3 PROGRAMAÇÃO NAS ARQUITETURAS PARALELAS ALVO

3.1 Motivação

A maior dificuldade na programação das arquiteturas paralelas alvo é a carência de *softwares* apropriados capazes de facilitar o processo de implementação das aplicações. Embora uma das principais características dessas arquiteturas seja fornecer uma imagem única, ou seja, o conjunto de computadores comportar-se como se fosse um único computador, programar sobre essas arquiteturas não é uma tarefa trivial. Devido ao fato do ambiente possuir memória distribuída, fazem-se necessárias trocas de informações entre os processos presentes nos computadores que compõem as arquiteturas. Programar sobre esse tipo de ambiente, no nível da aplicação, requer o conhecimento da localização dos processos para que sejam estabelecidas as trocas. O processo de implementação de aplicações seria simplificado se fosse fornecida, via *software* , transparência da localização dos computadores da arquitetura paralela.

Buscando suprir algumas das necessidades envolvidas no processo de implementação de aplicações, foram desenvolvidas e vem sendo empregadas algumas ferramentas e sistemas. Esse capítulo irá descrever algumas das ferramentas mais conhecidas para as arquiteturas paralelas alvo do Cadeo. Primeiramente, tem-se uma seção destinada as bibliotecas de comunicação, as quais simplificam a implementação de trocas de mensagens entre processos remotos. A seguir, tem-se a seção que descreve alguns sistemas gerenciadores de recursos empregados nas arquiteturas de grande porte. Por fim, serão mostradas as vantagens das linguagens orientadas a objetos, em especial Java, que tem impulsionado seu uso em ambientes desse gênero.

3.2 Bibliotecas de comunicação

Uma das características marcantes das arquiteturas paralelas alvo do Cadeo é que elas apresentam memória distribuída. Uma vez que são compostos por um conjunto de computadores comuns inter-conectados via rede. Os processos ou tarefas das aplicações paralelas que executam sobre esse tipo de ambiente comunicam-se durante sua execução para estabelecer trocas de dados e cooperarem entre si. Implementar a comunicação entre esses processos é um procedimento oneroso, complicado e repetitivo. Para agilizar a implementação de aplicações foram desenvolvidas as bibliotecas de comunicação. Estas bibliotecas possuem definidas rotinas e operações associadas, que viabilizam a comunicação entre processos de aplicações. As rotinas das bibliotecas são invocadas no código das aplicações e a comunicação será estabelecida através dela. O uso das bibliotecas de comunicação possibilitam a implementação de aplicações paralelas usando os mesmos meios empregados em implementações seqüenciais. Em outras palavras, é possível desenvolver aplicações paralelas sem a necessidade de utilizar novas linguagens de programação ou compiladores. A seguir, estão descritas três bibliotecas de comunicação utilizadas para a programação nas arquiteturas paralelas alvo.

3.2.1 MPI

A biblioteca MPI (*Message Passing Interface*) (GROPP; LUSK; SKJELLUM, 1994; GROPP et al., 1996) é, provavelmente, a biblioteca de comunicação mais utilizada atualmente para o desenvolvimento de aplicações paralelas e distribuídas. Esta biblioteca foi convencionada como padrão no MPI Fórum (MPI Forum, 2005) e foi o primeiro padrão portável de biblioteca a oferecer bom desempenho. É uma biblioteca de funções e macros que pode ser utilizada em aplicações escritas em linguagens de programação como C, FORTRAN e C++. Também existem iniciativas que possibilitam o acesso à biblioteca MPI com a linguagem Java. Esse acesso se dá através de invocação de métodos nativos JNI (*Java Native Interface*), onde pode ser citado o DOGMA (JUDD; CLEMENT; SNELL, 1998) e a mpiJava (BAKER et al., 1999). Como seu próprio nome supõe, MPI foi projetada para ser utilizada em programas que exploram a existência de múltiplos processos e estabelece a comunicação entre eles através da troca de mensagens.

Entre as características de MPI pode-se destacar alguns aspectos. A biblioteca apresenta **portabilidade** uma vez que os programas podem ser facilmente portados a plataformas diferentes daquela na qual foram implementados. A **padronização** da sintaxe e comportamento das rotinas de MPI possibilitam o uso de diferentes versões da biblioteca. Quando comparada a

utilização de outras bibliotecas de comunicação, MPI apresenta um bom **desempenho**, sendo, na maioria das vezes, melhor que as demais (KREUTZ et al., 2003). MPI também oferece **qualidade de implementação** já que prevê comunicação assíncrona, operações de comunicação coletiva, gerenciamento de mensagens, comunicação em diferentes formas e redes heterogêneas. E, ainda, ela também oferece **sincronização** da comunicação coletiva através do conceito de barreiras de sincronização (ANDREWS, 2000).

MPI possibilita escrever programas que utilizam o modelo de um único programa para múltiplos dados (SPMD - *Single Program, Multiple Data*) (WILKINSON; ALLEN, 1999). Na execução de um programa MPI, é lançado um conjunto de processos, criados na inicialização da biblioteca. Cada um dos processos está destinado a um determinado elemento processador disponível para a execução do programa. Uma cópia do mesmo programa é disparada em todos os computadores do sistema, sendo executado um trecho específico do programa em cada um deles. Para verificar qual região de código deve ser executada, realiza-se uma seleção baseada no identificador do processo corrente. Um fator que dificulta a implementação de aplicações é que se faz necessária a especificação do endereço dos elementos processadores tanto na distribuição dos processos quanto na comunicação entre eles. O ideal seria que a localização dos processos acontecesse transparentemente.

As primeiras versões da biblioteca MPI permitiam a execução em um ambiente estático onde ocorre o lançamento de um conjunto fixo de processos. Essa característica dificultava o uso da biblioteca em ambientes dinâmicos, ou seja, ambientes com grande variação no conjunto de computadores que os integram. Para suprir essa carência foi elaborada uma nova versão da biblioteca capaz de suportar e implementar o lançamento de processos dinâmicos (GROPP; LUSK, 1995), seguindo as normas da versão MPI-2. Conforme os rumos das evoluções dos sistemas distribuídos, desejava-se que a biblioteca também suportasse ambientes heterogêneos. Para que tal característica fosse inserida a biblioteca, recentemente surgiram duas implementações portáveis da biblioteca MPI, as quais são MPICH-2 (GROPP; LUSK, 1997) e a versão 7 da LAM-MPI. Essas versões suportam comunicação ponto-a-ponto, processos dinâmicos, comunicação em grupo e a utilização tanto de computadores monoprocessados quanto multiprocessados (SMP - *Symmetric Multiprocessing*). Porém, durante a implementação de aplicações paralelas o controle relacionado a dinamicidade do ambiente de execução não acontece automaticamente.

A comunicação entre processos, exceto quando se trata de comunicação coletiva, ocorre ponto-a-ponto, isto é, com a existência de um transmissor e um receptor. As primitivas mais

utilizadas para a troca de mensagem nesta biblioteca são MPI_Send( ) e MPI_Recv( ) para o envio e recepção de dados respectivamente. No total, existem mais de uma centena de primitivas disponíveis na interface dessa biblioteca. Porém, a maioria das implementações podem ser realizadas com um subconjunto dessas primitivas, visto que grande parte delas atende a problemas específicos. Ao se programar com MPI, as estruturas de dados enviadas em mensagens necessitam ser definidas e registradas.

Um exemplo de aplicação paralela implementada com a biblioteca MPI pode ser visto na figura 3.1. Ela apresenta uma aplicação no modelo mestre-escravo executando sobre computadores de um arquitetura paralela. A presença do identificador determina qual parte do programa será executada, se a parte correspondente ao mestre ou a parte correspondente ao escravo. O mestre coordena as execuções dos escravos através de um vetor que contém identificadores dos seus escravos. É através destes identificadores que as mensagens chegam aos seus devidos destinos.

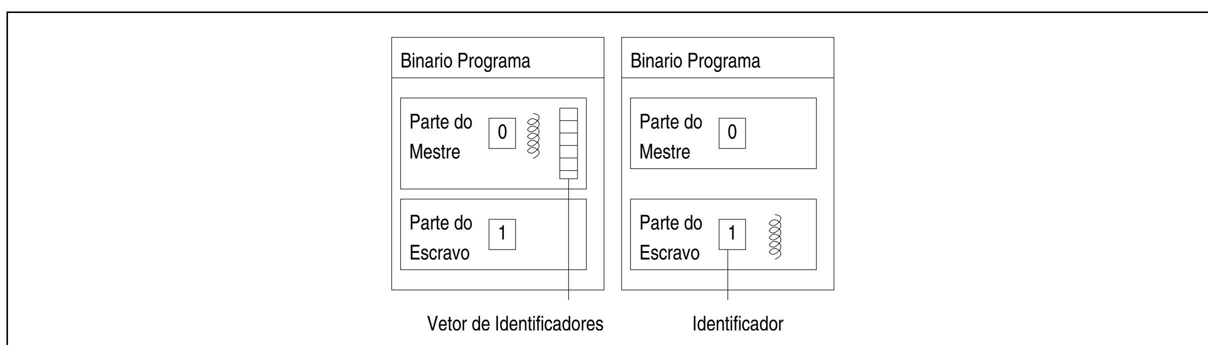

Figura 3.1: Modelo de uma aplicação implementada com MPI

3.2.2 PVM

PVM (*Parallel Virtual Machine*) (SUNDERAM, 1990) é uma biblioteca que proporciona uma infra-estrutura de *software* que emula uma máquina virtual. A máquina virtual pode ser composta por computadores heterogêneos, ou seja, composta por máquinas de diferentes arquiteturas, trabalhando cooperativamente. Para suportar a heterogeneidade, PVM manipula, transparentemente, todo o roteamento de mensagens, conversão de dados e escalonamento de processos tratando as incompatibilidades entre as arquiteturas. Entretanto, para que os programas paralelos possam funcionar adequadamente sobre uma máquina virtual heterogênea, faz-se necessário que eles sejam compilados para todas as arquiteturas pertencentes a máquina virtual.

Em PVM os programas paralelos são compostos por programas menores (processos), di-

ferentes entre si. Cada processo resolve uma parcela do problema atacado e pode vir a gerar novos processos. Os programas PVM seguem o modelo MPMD (*Multiple Program, Multiple Data*) que se difere do modelo SPMD das aplicações MPI. Além de suportar um ambiente heterogêneo, PVM também possibilita que esse ambiente seja dinâmico. Em qualquer ponto da execução de um programa paralelo, um processo qualquer pode iniciar ou finalizar outros processos ou adicionar ou remover computadores da máquina virtual. Para coordenar os processos, PVM possibilita a comunicação e/ou sincronização entre eles. Porém, controles específicos de dependências devem ser implementados pelo programador abaixo da biblioteca, através do uso estruturas fornecidas por ela.

Semelhante a MPI, PVM tem suporte às linguagens de programação C, C++ e FORTRAN. Existem também iniciativas para o uso com a linguagem Java, como o JavaPVM onde métodos nativos da biblioteca PVM são acessados. Outra iniciativa é a JPVM (*Java Parallel Virtual Machine*) (FERRARI, 1998) que é uma implementação de uma biblioteca de comunicação com a mesma idéia de PVM, seguindo o modelo de programação orientado a objetos.

Para estabelecer a comunicação entre os processos que integram um programa paralelo, PVM oferece a implementação das primitivas básicas de envio (PVM_send( )) e recebimento de dados (PVM_recv( )). Quaisquer processos PVM podem trocar mensagens sem restrições quanto a sua quantidade e seu tamanho. Além disso, o modelo de comunicação de PVM assume que as mensagens podem ter envio e recepção assíncrona e recepção síncrona. Logo, todas as rotinas de envio de dados serão não-bloqueantes, uma vez que os dados estarão armazenados em uma região de memória intermediária (*buffer*) e os processos não aguardam até que a troca de mensagem seja efetuada. Contudo, o recebimento pode ser tanto bloqueante quanto não bloqueante, possibilitando maior flexibilidade na escrita de uma aplicação.

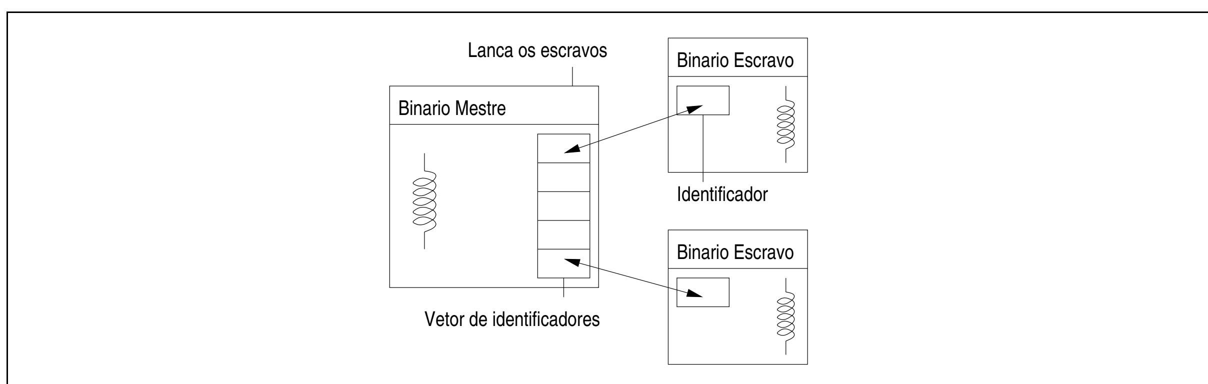

Figura 3.2: Modelo de uma aplicação implementada com PVM

A fim de facilitar a compreensão do modo de funcionamento da biblioteca PVM tomou-se um programa paralelo que segue o modelo mestre-escravo semelhante ao apresentado para MPI. A estrutura desse programa está representada na figura 3.2. Nela pode-se ver a implementação do processo mestre diferenciada da implementação dos processos escravos. O processo mestre lançará os processos escravos e os coordenará a partir de um vetor de identificadores.

3.2.3 DECK

O DECK (*Distributed Execution and Communication Kernel*) (BARRETO; NAVAUX; RIVIÈRE, 1998) é um ambiente que objetiva dar suporte a programação de alto desempenho. Esse ambiente é composto por um sistema de execução, uma biblioteca de comunicação e outros serviços especializados necessários a implementação de programas paralelos e distribuídos. Da mesma forma que MPI, um programa que utiliza o DECK segue o modelo SPMD onde tem-se uma cópia do programa executando em cada computador disponível na arquitetura paralela. O DECK permite troca de mensagens, semelhante a MPI, entretanto também permite o uso de múltiplos fluxos de execução de forma integrada. Assim, é possível conciliar arquiteturas SMP e programação com memória compartilhada e, ainda, fazer uso de tecnologias de rede de alta velocidade.

O DECK foi desenvolvido para a execução em aglomerados de computadores e também para sistemas que seguem o modelo MultiCluster (BARRETO; ÁVILA; NAVAUX, 2000). O modelo MultiCluster prevê a integração de múltiplos aglomerados de computadores. Em função dessa característica, o DECK suporta heterogeneidade da arquitetura paralela sobre o qual executa e ainda oferece um suporte mínimo para tolerância a falhas (BARRETO; NAVAUX; RIVIÈRE, 1998). Entre os aspectos que envolvem esse suporte mínimo a tolerância a falhas está a criação dinâmica de processos, fazendo com que o ambiente consiga administrar a dinamicidade da plataforma de execução. O DECK também permite que seus programas sejam executados, sem que sejam necessárias alterações, sobre diferentes tecnologias de rede por ele suportadas. A princípio o ambiente DECK foi desenvolvido para suportar o DPC++ que é uma extensão da linguagem C++ para programação distribuída (SILVEIRA et al., 2000). Ele também permite a utilização com a linguagem C e já existem esforços para sua utilização com Java (ROSA RIGHI; NAVAUX; PASIN, 2004).

A biblioteca de comunicação do DECK possibilita o uso e controle de múltiplos fluxos de execução, semáforos, *mutexes* e variáveis condicionais. O DECK também estabelece a comuni-

cação através da abstração de caixas de correio em um modelo de comunicação ponto-a-ponto e, ainda, primitivas para comunicação coletiva. Em virtude da comunicação ser realizada através de caixas de correio, todo o processo que desejar receber dados deve criar uma dessas caixas. Ao existirem processos que desejem enviar dados, estes devem clonar a caixa de correio pertencente ao processo destinatário dos dados. Após a clonagem, os dados devem ser empacotados em uma mensagem e inseridos na caixa de correio. O receptor deverá retirar a mensagem da caixa de correio, desempacotá-la e assim terá acesso aos dados que lhes foram enviados. As principais primitivas de comunicação são deck_mbox_post( ) que envia mensagens a caixa de correio e deck_mbox_retrv( ) que retira mensagens recebidas da caixa de correio.

A figura 3.3 apresenta um esquema de uma aplicação no modelo mestre-escravo como as apresentadas anteriormente. Nela tem-se o mestre enviando dados ao escravo através da caixa de correio. O escravo cria sua caixa de correio e o mestre a clona para que seja possível enviar-lhe os dados.

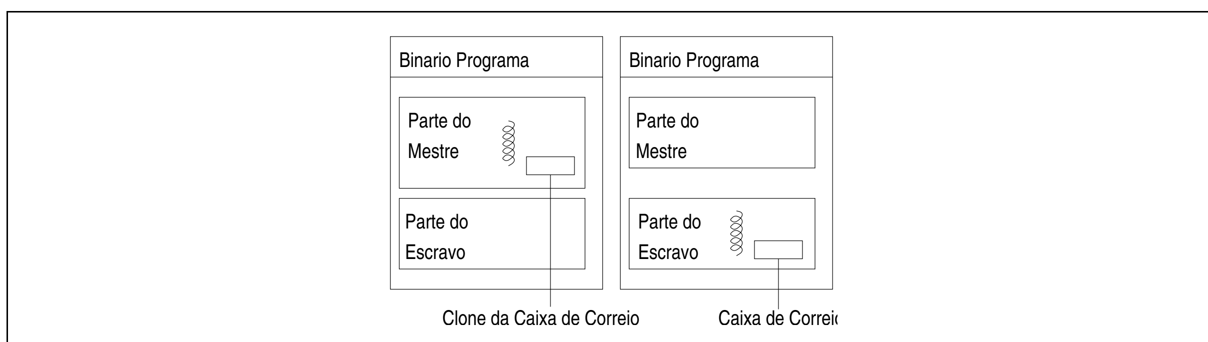

Figura 3.3: Modelo de uma aplicação implementada com DECK

Balanço do Uso de Bibliotecas

O uso de bibliotecas de comunicação facilita a implementação de aplicações destinadas a ambientes com memória distribuída. Isso porque, suas primitivas fornecem ao programador, através de uma interface simples, as operações que viabilizam a comunicação entre os processos. Além disso, as bibliotecas fornecem outros serviços básicos como por exemplo, meios para a detecção e tratamento de falhas que possam vir a ocorrer durante a execução das aplicações. Entretanto, mesmo com as vantagens oferecidas pelas bibliotecas, desenvolver aplicações para sistemas distribuídos não é uma tarefa simples.

Um dos principais motivos para essa dificuldade está na não transparência da localização dos processos comunicantes. Embora as bibliotecas, como as citadas anteriormente, favoreçam

a implementação da comunicação, ainda faz-se necessária a determinação de onde estão localizados os processos. Considerando ambientes de grande porte e dinâmicos como as grades, a implementação de aplicações torna-se mais complexa. Isso porque, o tratamento da dinamicidade deverá ser implementado pelo desenvolvedor da aplicação. Outro aspecto que dificulta a implementação de aplicações é a necessidade de gerenciamento dos recursos. A fim de amenizar este problema, existem sistemas que encarregam-se do gerenciamento. Exemplos desses sistemas podem ser vistos na próxima seção.

3.3 Gerenciadores de Recursos

As bibliotecas de comunicação facilitam a implementação de programas ou aplicações paralelas para sistemas distribuídos. Porém, quando esses sistemas tomam grandes proporções, tais como as grades de computadores, além de usar bibliotecas de comunicação também é necessário ter meios de gerenciar tantos computadores. Isso porque, é comum que os recursos que integram tais sistemas sejam heterogêneos e dinâmicos. Para essas arquiteturas de grande porte fazem-se necessários sistemas gerenciadores de recursos que controlam, principalmente, o conjunto de processadores disponíveis, banda de rede e discos de armazenamento. Sem o uso desses gerenciadores, as aplicações se tornariam muito complexas uma vez que teriam que ater-se a resolver problemas referentes a descoberta e gerenciamento de recursos. Nessa seção serão apresentados alguns sistemas gerenciadores bem como sua estrutura e modo de funcionamento. Os gerenciadores mostrados a seguir foram propriamente desenvolvidos ou adaptados as grades de computadores, por se tratar da arquitetura de grande porte em maior evidência atualmente.

3.3.1 Globus

O Globus (FOSTER; KESSELMAN, 1998, 1997) é um dos projetos mais referenciados na literatura e consiste de uma infraestrutura de *software* que capacita aplicações a manipular recursos computacionais heterogêneos e distribuídos como uma máquina virtual única. O elemento central do sistema Globus é o seu conjunto de ferramentas. Nele estão incluídos *softwares* para segurança, infraestrutura de informação, gerenciamento de recursos e dados, comunicação, detecção de falhas e portabilidade. O Globus apresenta um conjunto de API (*Application Programming Interface*) que suporta uma variedade de aplicações e paradigmas de programação. Essa API possibilita seu uso para aplicações implementadas com a linguagem C e, para o lado de clientes, em Java.

O conjunto de ferramentas do Globus possui uma estrutura modular que permite que as aplicações possam fazer uso apenas das características que lhes são necessárias. Por exemplo, uma determinada aplicação pode fazer uso da infraestrutura de gerenciamento de recursos e informação sem utilizar a biblioteca de comunicação disponibilizada pelo Globus. Embora o conjunto de módulos existentes no Globus varie conforme sua versão, ele engloba, principalmente, os seguintes elementos:

- *Grid Resource Allocation Manager* (GRAM) que é o protocolo que permite alocação remota, reserva, monitoração e controle dos recursos computacionais da grade.

- GridFTP que é uma extensão do protocolo FTP (*File Transfer Protocol*) o qual permite acesso com alto desempenho, segurança e confiabilidade a dados.

- *Grid Security Infrastructure* (GSI) garante a segurança das conexões realizando autenticações, autorizações e mecanismos de proteção de mensagens de maneira uniforme.

- *Grid Resource Information Protocol* (GRIP) atualmente baseado no LDAP (*Lightweight Diretory Access Protocol*) é utilizado para definir um padrão para as informações dos recursos e do modelo das informações associadas.

A questão que envolve a segurança das grades é fundamental para o sucesso do uso desse ambiente. Uma vez que estão envolvidos uma variedade de domínios administrativos, é necessário oferecer as condições mínimas de segurança às máquinas que as compõem. Garantias como autenticação e autorização são fundamentais para incentivar os administradores dos domínios a disponibilizarem seus recursos em um ambiente de grade. O Globus oferece segurança à grade através de seu módulo GSI, o qual baseia-se em certificados de autoridade (*certificate authority* - CA) (WELCH et al., 2003).

A estrutura do Globus volta-se para a disponibilização e uso dos recursos computacionais das grades de forma unificada. A flexibilidade oferecida pelo Globus permitindo o uso independente de seus módulos contribui para que problemas específicos possam ser resolvidos adequadamente. Porém, ainda não existe uma versão do Globus que funcione sem a necessidade de *softwares* adicionais, pelo menos para o escalonamento de recursos. Essa característica faz com que o uso do Globus esteja sempre relacionado ao uso de outras ferramentas que supram suas carências.

A fim de compreender em que nível se enquadra o sistema Globus na estrutura das grades de computadores tem-se a figura 3.4. A figura apresenta em seu nível inferior os recursos computacionais que integram a grade. Acima dos recursos e fornecendo o gerenciamento dos mesmos, tem-se o Globus e seus principais módulos. Como o conjunto de módulos varia conforme a versão do sistema, foram representados os principais e inseridas reticências caracterizando a variação e a possibilidade de inclusão de mais módulos. Acima do sistema Globus tem-se a representação de uma aplicação e de outros serviços que fornecem funcionalidades não contempladas pelo sistema.

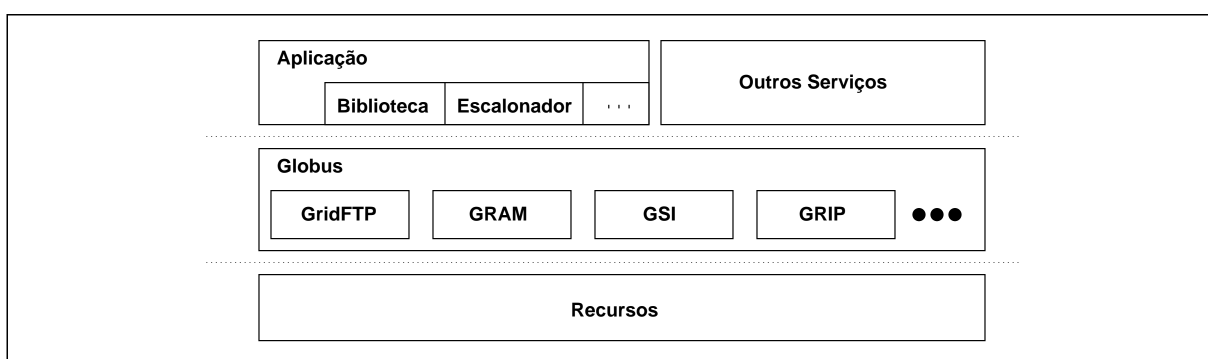

Figura 3.4: Representação da estrutura do Globus

Embora não hajam restrições quanto a quais serviços serão fornecidos ao Globus, o principal deles está relacionado ao escalonamento de recursos. Sistemas como Condor, Legion, AppLeS entre outros, são comumente utilizados para fornecerem o escalonamento dos recursos do sistema. Outro aspecto apresentado na figura diz respeito à aplicação que utiliza o Globus. Embora o sistema ofereça uma biblioteca de comunicação aos seus usuários (chamada Nexus (FOSTER; KESSELMAN; TUECKE, 1996)), seu uso não é obrigatório. Essa característica permite que bibliotecas como as apresentadas na seção 3.2 sejam utilizadas conjuntamente com o Globus. Além disso, permite que aplicações já existentes e implementadas com outro tipo de biblioteca de comunicação, possam executar sobre o Globus sem que sejam necessárias alterações em sua estrutura. Essa característica se estende também a outras ferramentas utilizadas na implementação da aplicação, como por exemplo um escalonador específico para a distribuição das tarefas entre os computadores disponíveis.

3.3.2 Legion

Assim como o Globus, o sistema Legion (GRIMSHAW; WULF; TEAM, 1997; GRIMSHAW et al., 1999; CHAPIN et al., 1999) visa oferecer serviços básicos para a utilização das grades de computadores, tais como segurança e gerenciamento de recursos e dados. O sistema oferece uma máquina virtual única que será composta por computadores, independentemente da dispersão geográfica dos mesmos. O Legion implementa serviços de mais alto nível utilizando para isso sistemas operacionais, ferramentas de gerenciamento de recursos e mecanismos de segurança já existentes. O sistema busca fornecer, dentre outras coisas, autonomia para os domínios, suporte a heterogeneidade, extensibilidade, facilidade de uso, desempenho (através de processamento paralelo), tolerância a falhas e escalabilidade.

Um dos aspectos cruciais da utilização das grades é a segurança, a qual está relacionada ao tipo de recursos acessado e ao nível de segurança desejado. O Legion oferece mecanismos que asseguram integridade e confidencialidade de dados, autenticação e autorização. Também permite que as políticas empregadas possuam uma configuração local controlada pelos administradores dos domínios que compõem a grade. Dessa forma, o Legion permite explicitamente que os domínios possuam políticas diferenciadas e mantenham o controle de seus recursos independentemente do restante da grade.

O Legion é um sistema baseado em objetos composto por um conjunto de objetos independentes que comunicam-se entre si através de invocações remotas de métodos. Tudo o que for relevante para o sistema é representado por objetos, como por exemplo, arquivos, aplicações, interfaces de aplicações, usuários e grupos. Essa é uma característica que o difere do Globus já que fornece ao ambiente de grade o encapsulamento de todos os seus componentes em objetos. Essa metodologia tem todas as vantagens normais de um modelo orientado a objetos, tais como abstração de dados, encapsulamento, herança e polimorfismo (GRIMSHAW et al., 2003).

A hierarquia dos objetos básicos do sistema Legion está representada na figura 3.5. As classes no Legion possuem uma outra funcionalidade além da padrão que é definir o tipo das suas instâncias. As classes atuam como entidades ativas responsáveis pelo gerenciamento das suas instâncias, incluindo sua localização, através de um ClassManager. Por ser um modelo hierárquico, em caso de falha de um objeto, o ClassManager da classe superior poderá tomar as devidas ações, tal como migrá-lo ou reinicializá-lo. A classe LegionClass é a classe mais geral e a partir dela são estendidas todas as demais classes do sistema, inclusive as de usuário. Os objetos criados a partir da classe HostClass encapsulam as informações sobre a capacidade dos

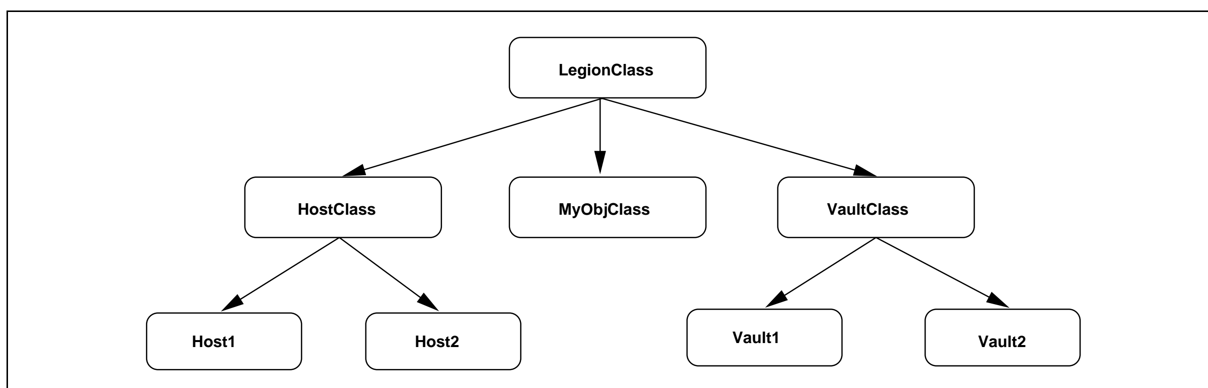

Figura 3.5: Legion: hierarquia de objetos básicos (CHAPIN et al., 1999)

computadores e é responsável pela instanciação de objetos no computador ao qual representa. As classes VaultClass representam uma abstração de unidades de armazenamento genéricas. Cada objeto dentro do Legion possui um objeto dessa classe associado a ele para garantir a persistência do estado do objeto, sendo que este estado será utilizado para fins de migração, reinicialização ou finalização do objeto.

O Legion oferece um escalonador básico que procura casar as informações dos computadores com as necessidades das tarefas das aplicações. Para que o escalonamento seja eficiente é preciso fornecer ao sistema um bom conhecimento sobre as características e as necessidades da aplicação a ser executada, o que nem sempre é possível de ser estimado. O sistema também aceita que novas implementações para os módulos sejam utilizadas.

3.3.3 Condor

O Condor (LITZKOW; LIVNY; MUTKA, 1988; THAIN; TANNENBAUM; LINVY, 2003; LIVNY et al., 1997) é um sistema que oferece localização de recursos e alocação de tarefas de forma automática através do monitoramento dos computadores disponíveis. Ele vem sendo concebido e aperfeiçoado há mais de quinze anos e os resultados dos esforços de seus idealizadores reflete-se em sua popularidade e aceitação. Inicialmente, ele foi planejado para funcionar em NOWs, com o passar do tempo e as evoluções das arquiteturas paralelas ele foi adaptado as novas realidades.

O objetivo do sistema é maximizar a utilização de computadores com a menor interferência possível entre as tarefas escalonadas e as atividades do usuário do computador. Para isso, o sistema aproveita-se de computadores em estado de ociosidade. O sistema identifica os computadores ociosos e os insere em um banco de recursos (*resources pool*). Através de uma estrutura

de escalonamento o sistema agenda tarefas seqüenciais nos computadores presentes no banco de recursos.

O Condor se propõe a oferecer grande quantidade de processamento a médio e longo prazo, ou seja, fornecer um desempenho uniforme a uma aplicação, mesmo que o potencial de desempenho do sistema como um todo seja variável. Tal fato o caracteriza como sendo um sistema capaz de oferecer alta vazão e não alto desempenho (LITZKOW; LIVNY; MUTKA, 1988; FREY et al., 2001; LIVNY et al., 1997).

Outro aspecto do Condor é sua estrutura de escalonamento, a qual é transparente aos usuários e composta por um coordenador e por escalonadores locais (LITZKOW; LIVNY; MUTKA, 1988). O coordenador centraliza informações dos computadores que compõem o sistema e implementa a política de escalonamento escolhida pelo administrador. Ele executa em um dos computadores que integra o sistema, distribuindo tarefas entre os computadores ociosos. Para tal, o coordenador se baseia no tamanho da fila de tarefas que cada computador está executando. Os escalonadores locais estão presentes em todos os computadores que integram o sistema e serão responsáveis pelo gerenciamento da execução das tarefas naquele computador. A figura 3.6 mostra a distribuição da estrutura de escalonamento, onde cada um dos computadores possui um escalonador local e, em um deles, se encontra o coordenador.

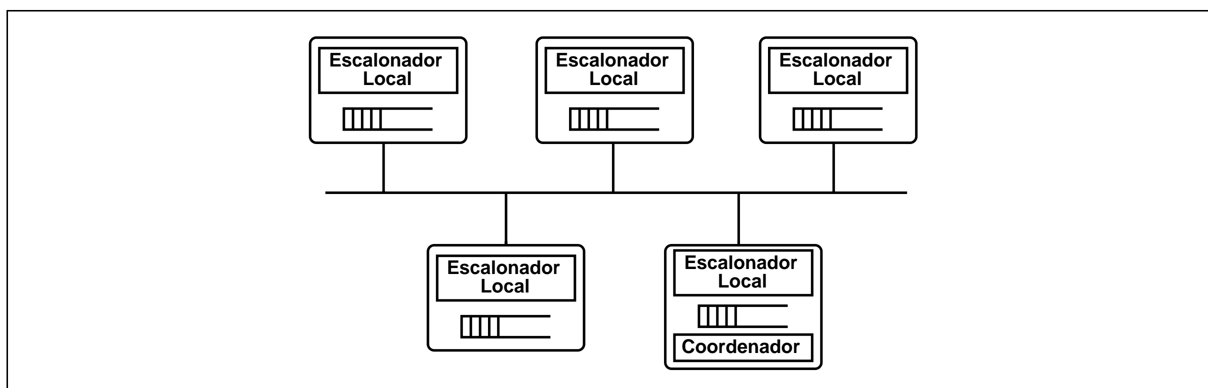

Figura 3.6: Estrutura de escalonamento do Condor

A estrutura de escalonamento do Condor proporciona a centralização do gerenciamento dos computadores presente no banco de recursos através do coordenador. Ele é responsável por empregar políticas para o escalonamento das tarefas a serem executadas. Essa estrutura também proporciona a independência dos computadores que integram o sistema uma vez que o escalonamento local das tarefas é realizado pelo escalonador local. Dessa forma é possível a continuidade da execução das tarefas localmente, sem serem afetadas nos casos de falhas por

parte do coordenador central.

O Condor possibilita a migração de tarefas de um computador ocioso para outro. Dessa forma é possível garantir que o processamento realizado até um dado instante por um computador não seja perdido quando este deixar de estar disponível. Para proporcionar tal vantagem, um estado atual da execução ou ponto de verificação (*checkpoints*) é salvo periodicamente. Ao ser necessário o relançamento de uma tarefa que teve sua execução interrompida, o novo recurso processará a tarefa a partir do último ponto de verificação registrado.

O sistema também preocupa-se em proporcionar equilíbrio de acesso aos recursos computacionais a todos os seus usuários. Para isso, o Condor faz uso de um algoritmo de balanceamento de cargas chamado *Up-Down* (MUTKA; LIVNY, 1987). Através desse algoritmo, tanto os usuários com grande demanda (aqueles que possuem tarefas que demandam grande tempo de processamento) quanto os que demandam pouco tempo de processamento terão acesso aos recursos disponíveis.

Em conseqüência da evolução dos sistemas distribuídos buscou-se integrar dois ou mais desses sistemas e assim ter a disposição um maior potencial computacional. Dentro desse desejo de integração de sistemas distribuídos, o Condor ampliou sua estrutura concebendo o Bando de Condors) (*Flock of Condors* (EPEMA et al., 1996). Em um Bando de Condors é mantida a estrutura de escalonamento do sistema e o conceito de bancos de recursos. O diferencial dessa nova versão do sistema é a possibilidade de integração de bancos de recursos possibilitando o compartilhamento dos computadores disponíveis. Para possibilitar a integração de dois bancos de recursos, cada um deles possuirá uma máquina destinada a funcionar como uma ponte (*gateway*), configurando sempre uma ligação par-a-par sem nenhuma entidade centralizadora entre os bancos.

Uma outra evolução do Condor foi o desenvolvimento do Condor-G (FREY et al., 2001), a fim de possibilitar que o Condor pudesse adaptar-se ao conceito de grades de computadores apresentando uma visão mais heterogênea. O Condor-G representa o casamento do sistema ao Globus (FOSTER; KESSELMAN, 1998), onde além dos bancos de recursos o Condor utiliza recursos via Globus. O Globus disponibiliza protocolos de segurança para comunicação inter-domínios e padrões de acesso entre diferentes sistemas de gerenciamento dos domínios e, assim, ampliando a abrangência do Condor. O Condor-G apresenta uma arquitetura mais centralizada que o *Flock of Condors* onde o escalonador submete tarefas de aplicações tanto ao banco de recursos quanto aos recursos disponibilizados pelo Globus.

Existe um crescente interesse no aproveitamento das vantagens das linguagens orientadas a objetos para a programação de arquiteturas paralelas. Seguindo esta linha, o Condor foi adaptado e possibilita a execução de aplicações em Java (THAIN; LIVNY, 2002).

3.3.4 MyGrid

O MyGrid (CIRNE et al., 2003; CIRNE; MARZULLO, 2001) é um sistema que possibilita a execução de tarefas de aplicações paralelas em computadores da grade com o objetivo de ser simples, completo e seguro. Para evitar problemas quanto a definição de quais são os computadores pertencentes a grade, o MyGrid utiliza-se de uma política que baseia-se nas permissões do usuários do sistema. Para o MyGrid, a grade de um usuário é formada por todos os computadores onde esse usuário possua acesso, independente da localização dos recursos. As aplicações executadas através desse sistema devem seguir o modelo de aplicações do tipo sacola de tarefas (*bag-of-task*) (ANDREWS, 2000), onde suas tarefas são independentes entre si, contribuindo para sua distribuição entre computadores geograficamente dispersos. Embora soe estranho restringir o sistema a executar um único tipo de aplicação, existem muitos problemas que podem ser resolvidos através de aplicações desse gênero (SMALLEN et al., 2000; SMALLEN; CASANOVA; BERMAN, 2002; STILES et al., 2001; CIRNE et al., 2003; ADLER; GONG; ROSENBERG, 2003). O MyGrid visa oferecer escalonamento de tarefas de forma a conseguir tirar bom proveito dos computadores disponíveis na grade, os quais costumam ser bastante heterogêneos. A arquitetura do MyGrid é flexível a fim de suportar a dinamicidade dos computadores para facilitar sua inclusão e exclusão na grade.

Os usuários que desejarem executar suas aplicações com o MyGrid devem inicialmente submeter as tarefas da aplicação para o computador que coordenará a distribuição delas. Esse computador é chamado de máquina base (*home machine*) e costuma ser o próprio computador do usuário. As tarefas serão distribuídas entre os computadores que compõem a grade do usuário, as quais recebem a denominação de máquinas da grade (*grid machines*). As tarefas da aplicação deverão ser formadas por três sub-tarefas chamadas de inicial (*initial*), grade (*grid*) e final (*final*) e essas sub-tarefas devem ser executadas exatamente nessa ordem. As sub-tarefas inicial e final são executadas na máquina base, sendo que a primeira delas inicia o ambiente e transfere dados de entrada caso necessário e a última sub-tarefa é responsável pela espera dos resultados da tarefa caso existam. A sub-tarefa grade executa sobre nos computadores da máquina da grade e representa a computação desejada da tarefa.

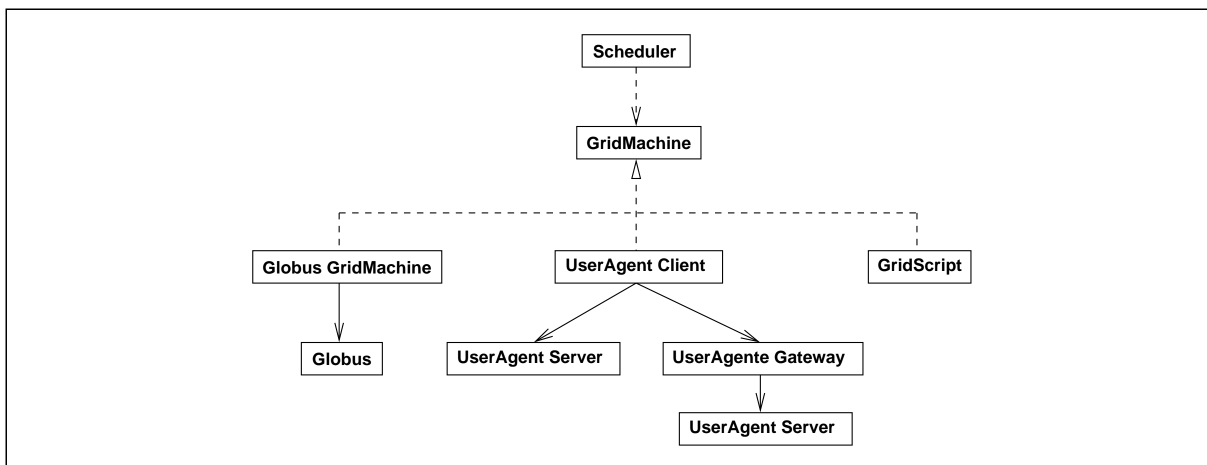

Figura 3.7: Interface da *grid machine* e suas implementações (COSTA et al., 2004)

A figura 3.7 ilustra como acontece a comunicação com as máquinas que fazem parte da grade. O MyGrid disponibiliza três implementações nativas para a realização da comunicação: o agente de usuário (*UserAgent*), *scripts* de grade (*GridScript*) e o Globus. Os agentes de usuários costumam ser usados quando é fácil instalar *softwares* nas máquinas da grade e esses agentes fornecem uma implementação básica, em Java, dos serviços necessários. Os *scripts* de grade fornecem as mesmas funcionalidades dos agentes de usuário porém usando *scripts* ao invés de processos Java. A adequação a utilização do Globus existe por este ser um dos mais populares gerenciadores de recursos para grades. Outro motivo é que o MyGrid tem a oferecer o escalonamento de que o Globus carece.

O escalonador presente no MyGrid é uma importante característica do sistema. Em um ambiente de grade, em função de sua natureza heterogênea e dinâmica, conseguir obter e manter as informações de todos os seus recursos é uma tarefa bastante complicada. Também é difícil obter boas informações sobre as tarefas a serem executadas e, sem poder contar com tais informações, torna-se difícil encontrar soluções capazes de fornecer um bom escalonamento das cargas de processamento. Buscando contornar esse problema, o MyGrid faz uso de um algoritmo de fila com replicação (*Work Queue with Replication* - WQR) (PARANHOS; CIRNE; BRASILEIRO, 2003). Esse algoritmo oferece bom desempenho sem levar em consideração as informações das tarefas ou computadores envolvidos.

O escalonamento dinâmico oferecido pelo WQR baseia-se na replicação das tarefas das aplicações. Ao deixarem de existir tarefas a serem lançadas e algum computador tornar-se disponível, este receberá uma réplica de uma tarefa que ainda não tenha concluído sua execução.

Dessa forma é possível atacar problemas gerados pela heterogeneidade de computadores e tarefas e também a variação dinâmica de disponibilidade de computadores. Porém, segundo os estudos realizados (PARANHOS; CIRNE; BRASILEIRO, 2003), a replicação ajuda a contornar o problema da perda de ciclos oferecendo um desempenho razoável, mas não resolve o problema completamente.

A continuidade do trabalho realizado no MyGrid é o OurGrid (ANDRADE et al., 2003), o qual permite a criação de comunidades em larga escala. O OurGrid trabalha com os conceitos da computação ponto-a-ponto (*peer-to-peer* - P2P) onde os recursos são compartilhados segundo "redes de favores". Nessas redes, alocar um recurso para executar um serviço é um favor e os recursos de um ponto da rede são disponibilizados para os outros pontos. O critério para a disponibilização é a existência de créditos em seu histórico de interações, ou seja, um ponto disponibilizará recursos, prioritariamente, com outros pontos que já tenham lhe prestado algum favor. O compartilhamento de recursos em redes P2P também pode ser considerado como uma iniciativa para o melhor aproveitamento dos recursos de arquiteturas paralelas. Essa iniciativa é recente, mas tende a evoluir e ser cada vez mais empregada como meio de utilização de recursos.

Balanço dos Gerenciadores de Recursos

As arquiteturas paralelas de memória distribuída passaram a ter um número cada vez maior de computadores. Essa tendência é confirmada pelo crescente interesse em grades de computadores. Os sistemas gerenciadores de recursos facilitam a utilização de arquiteturas paralelas de grande porte. Esses sistemas, em linhas gerais, oferecem serviços básicos, tais como descoberta de recursos, autenticação, segurança, escalonamento, entre outros. A maior contribuição desses sistemas é tornar mais simples a utilização e aproveitamento das potencialidades da arquitetura paralela. Os usuários e as aplicações paralelas utilizam os serviços dos gerenciadores livrando-se dos transtornos por trás dos mesmos.

Nessa seção foram apresentados quatro gerenciadores de recursos desenvolvidos para ambientes de grades. O Globus, que é um dos mais populares, apresenta uma estrutura modular que atende a grande maioria das necessidades de um ambiente de grande porte. Para os serviços que não são atendidos pelo Globos é possível a integração com outros sistemas. Por exemplo, o Globus não apresenta escalonamento, porém possibilita a utilização do Condor para suprir essa carência. O Condor também é um sistema gerenciador, cuja característica marcante é sua estrutura de escalonamento. O Condor disponibiliza computadores em ociosidade para o

processamento de tarefas possibilitando um melhor aproveitamento dos recursos. A principal característica dos outros dois gerenciadores apresentados, Legion e MyGrid, é que ambos são orientados a objetos. Existe uma tendência para o aproveitamento das facilidades da orientação a objetos na programação de arquiteturas paralelas como as em questão. Entre as linguagens orientadas a objetos, Java tem se destacado e vem sendo cada vez mais empregada. Essa tendência será detalhada na seção seguinte.

3.4 Utilização de Java em PAD

A popularidade da linguagem Java (GOSLING; MCGILTON, 2004) tem aumentado nos últimos anos em virtude de sua simplicidade e flexibilidade. Ela é uma linguagem de programação orientada a objetos que apresenta características como herança, polimorfismo, reusabilidade de código, portabilidade, entre outras. O modelo de programação orientado a objetos proporciona simplicidade no planejamento e desenvolvimento de aplicações. Além da sua utilização na programação seqüencial, Java apresenta mecanismos próprios que favorecem sua utilização para a programação paralela e distribuída (GETOV et al., 2001). Ela tem suporte à programação com múltiplos fluxos de execução (*multithreading*) e memória distribuída que dispensam o uso de bibliotecas suplementares como é necessário em outras linguagens, por exemplo C e FORTRAN (KIELMANN et al., 2001).

O processo de implementação de aplicações Java é favorecido pelo modelo orientado a objetos. Também é possível a criação de tipos de dados abstratos e a reutilização de códigos já implementados. Embora facilite o processo de implementação de aplicações, as características inerentes de Java, entre elas sua natureza interpretada, oferecem um sobre-custo ao desempenho das aplicações. Ao se comparar o desempenho de aplicações Java com o de outras linguagens não interpretadas, por exemplo C, Java apresenta um desempenho inferior. Muitos investimentos tem sido realizados buscando amenizar a defasagem de desempenho proporcionada por Java. A diferença de desempenho tende a ser cada vez menor já que existem iniciativas que têm conseguido melhoras significativas e encorajado para emprego dessa linguagem na programação de arquiteturas paralelas. A seguir serão apresentadas algumas dessas iniciativas.

3.4.1 Iniciativas de melhora de Desempenho

Alguns aspectos da linguagem Java, tais como a verificação de exceções e o coletor de lixo automático acabam influenciando o desempenho das aplicações. Outro aspecto que influencia

o desempenho de Java é que, para que seja possível oferecer portabilidade a diferentes arquiteturas, Java é uma linguagem interpretada. Nela um código fonte em Java é compilado para uma arquitetura neutra chamada de *bytecode* e este é posteriormente interpretado por uma Máquina Virtual Java (JVM - *Java Virtual Machine*).

Algumas alternativas que buscam melhorar o desempenho de Java concentram seus esforços na redução do tempo no processo de interpretação do *bytecode*. Para isso, buscam compilar parte ou todo o *bytecode* para código de máquina nativo. Um exemplo de compilação parcial de código é o oferecido por compiladores JIT (*Just In Time*) que traduzem, em tempo de execução, arquivos *bytecode* para instruções de máquina oferecendo ganhos de desempenho. Compiladores JIT estão disponíveis na maioria das implementações da JVM (RADHAKRISHNAN et al., 2001) as quais combinam as técnicas de compilação JIT e interpretação, sendo que a compilação é empregada nos trechos de maior acesso no *bytecode*.

Já os compiladores diretos traduzem código fonte Java ou *bytecode* para instruções de código de máquina. Esta tradução é realizada previamente à execução e de forma estática, sendo que esses compiladores não suportam a carga dinâmica de classes (KAZI et al., 2000). Outra conseqüência dos compiladores diretos é a perda da portabilidade uma vez que seus códigos gerados são específicos a uma determinada arquitetura. Como exemplo desse tipo de compiladores pode-se citar o Caffeine (HSIEH; GYLLENHAAL; W. HWU, 1996) que transforma *bytecode* em código nativo otimizado para a arquitetura x86. Também tem-se o GCJ (TROMEY, 2004) que além de transformar arquivos fonte Java em código de máquina, também pode gerar código intermediário C para posterior compilação e execução padrões nessa linguagem.

Além de iniciativas, como as mostradas acima, que buscam obter eficiência durante a execução de programas Java, existem também iniciativas que visam a implementação de aplicações mais eficientes. Nesse contexto inserem-se as bibliotecas Java para programação paralela e distribuída, as quais oferecem criação e manutenção de vários fluxos de execução, mecanismos eficientes de sincronização e passagem de mensagem. Como exemplos desse tipo de bibliotecas têm-se o Java Party (HAUMACHER; MOSCHNY; PHILIPPSEN, 2004; PHILIPPSEN; ZENGER, 1997) e o ProActive (CAROMEL; KLAUSER; VAYSSIERE, 1998). Também pode-se citar o Manta (MAASSEN et al., 2001) que é uma distribuição de JVM otimizada a qual permite alto desempenho na invocação remota de métodos (RMI - *Remote Method Invocation*).

O JavaParty busca oferecer mecanismos mais elegantes e diretos para implementar aplicações paralelas em Java para sistemas com memória distribuída. JavaParty oferece transparência

de localização, ou seja, ele encarrega-se de mapear a localização de objetos e fluxos de execução (*threads*) distribuídos que podem ser utilizados como se fossem locais. Para que a distribuição aconteça, o programador deverá inserir a palavra remote na declaração de classe, indicando que as instâncias daquela classe podem ser movidas para outros processadores. O tratamento dos códigos com a sintaxe do JavaParty é realizado por meio de um pré-processador que gera códigos aptos a realizarem migração, possuírem localização transparente e realizarem serialização mais eficiente (PHILIPPSEN; HAUMACHER; NESTER, 2000). Dessa forma o JavaParty consegue melhoras de desempenho na invocação remota de métodos, além de fornecer uma imagem única do sistema, como uma única máquina virtual.

O ProActive é uma biblioteca Java que busca simplificar o desenvolvimento de aplicações paralelas e distribuídas. Ele oferece invocação remota de métodos assíncrona, espera por necessidade, migração, segurança, polimorfismo entre objetos locais e remotos, entre outras características. Para proporcionar tais características, o ProActive utiliza-se de uma abstração de objetos ativos. Um objeto ativo é um objeto padrão do Java com um fluxo de execução associado a ele, e este fluxo controlará os serviços oferecidos pelo objeto. Além dos objetos ativos o ProActive possui objetos futuros que possibilitam assincronismo na invocação de métodos e espera por necessidade. Para a utilização da biblioteca na implementação de aplicações o programador deve inserir invocações aos métodos da API do ProActive.

O Manta foi desenvolvido para oferecer alto desempenho na implementação de RMI (comunicação) através de um compilador nativo, como JIT, e de seus protocolos próprios para serialização e empacotamento (*marshaling*). Ele oferece otimizações para a JVM as quais possibilitam uma maior eficiência na invocação de métodos remotos e na serialização de dados. O Manta também faz uso de uma biblioteca de baixo nível, chamada Panda (BAL et al., 1998), que minimiza o número de cópias de memória, resultando em uma maior vazão. Além disso, o RMI do Manta combina alto desempenho com flexibilidade e interoperabilidade, ou seja, é possível que o Manta comunique-se com computadores que apresentam a JVM padrão do Java. Essa característica viabiliza seu uso em sistemas de grande porte, como as grades, onde facilmente existirá variedade de JVM em uso. Por se tratar de um compilador, para que o Manta seja utilizado basta compilar o programa Java com os comandos de compilação por ele oferecido.

Essa seção buscou exemplificar algumas iniciativas para a obtenção de melhora no desempenho de aplicações Java. Dentre as opções estudas, optou-se pela utilização da biblioteca ProActive por oferecer assincronismo na invocação de métodos remotos, espera por necessidade e

uma API de simples utilização. Além disso, seu modelo de implementação segue o mesmo modelo do Java padrão, dispensando o uso de pré-processadores ou de compiladores específicos. A seguir tem-se, em mais detalhes, as características, a estrutura e modo de funcionamento do ProActive.

3.4.2 A Biblioteca ProActive

O ProActive (CAROMEL, 2004a; CAROMEL; KLAUSER; VAYSSIERE, 1998; CAROMEL, 1993) é uma biblioteca Java para computação paralela, distribuída e concorrente. Essa biblioteca proporciona simplicidade no modelo de programação de aplicações destinadas a executarem de forma distribuída, independente se a execução ocorrerá em uma rede local, em um aglomerado ou em um ambiente de grade. Essa simplicidade é proporcionada por um conjunto simples e completo de primitivas que integram sua API. O ProActive proporciona ao Cadeo uma interface simples para a distribuição e comunicação entre os objetos do próprio Cadeo e os objetos das aplicações que o utilizam. Essa seção irá detalhar alguns aspectos importantes do ProActive que contribuíram para a estruturação e funcionalidade do Cadeo.

O ProActive é uma biblioteca inteiramente composta por classes Java e apresenta total compatibilidade com o Java tradicional, não sendo necessárias alterações na JVM para o seu funcionamento. Um dos principais objetivos do ProActive é reduzir a distância entre a programação multiprocessada e a programação distribuída. Dessa forma seria possível reutilizar códigos de aplicações com múltiplos processos e executá-las de forma distribuída. Para tornar viável tal objetivo é necessário que os objetos possam ter transparência de localização, a fim de proporcionar polimorfismo entre objetos locais e remotos. No ProActive, a localização de objetos instanciados remotamente é transparente, mas a localização deve ser conhecida no momento da instanciação. Além disso, é necessário que haja transparência nas atividades dos objetos uma vez que no ProActive as invocações de método serão realizadas em uma *thread* existente associada àquele objeto (CAROMEL; KLAUSER; VAYSSIERE, 1998).

A fim de obter sincronização e transparência na localização e nas atividades dos objetos remotos, o ProActive fundamenta-se no conceito de objetos ativos (CAROMEL; KLAUSER; VAYSSIERE, 1998; CAROMEL, 2004b). Um objeto ativo é composto pelo objeto padrão do Java e uma *thread*, chamada de corpo (*body*) associada a ele. A representação dos componentes de um objeto ativo pode ser vista na figura 3.8. O corpo é responsável por receber as invocações de método de um objeto ativo e ordená-las em uma lista de requisições pendentes. Essa lista de

requisições será atendida conforme uma política de sincronização, que por padrão é FIFO (*First In, First Out*), mas que pode ser alterada caso haja necessidade.

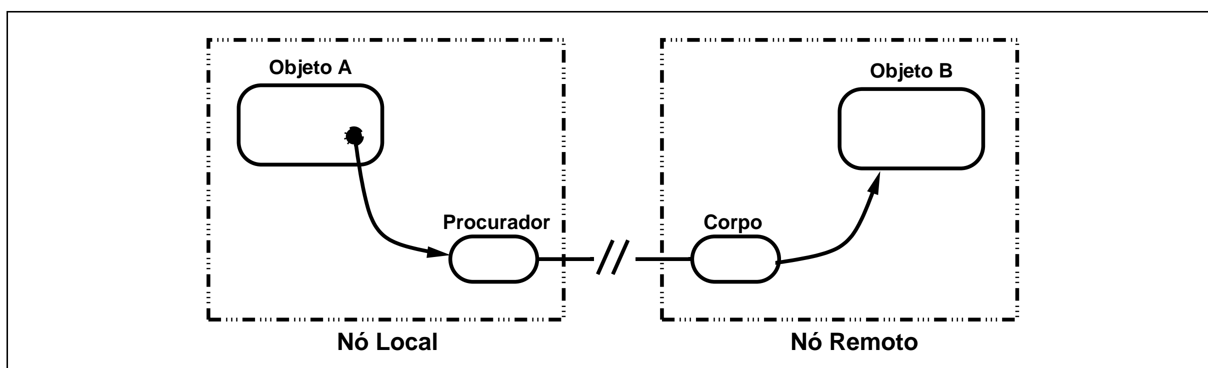

Figura 3.8: Componentes de um objeto ativo

O ProActive possibilita a invocação de métodos remotos de forma assíncrona. Para tanto esta biblioteca faz uso de objetos futuros (CAROMEL; KLAUSER; VAYSSIERE, 1998; CAROMEL; HENRIO; SERPETTE, 2004). Um objeto futuro compreende o retorno imediato de uma invocação de método em um objeto ativo e representa o resultado da invocação de um método ainda não processado. Quando finalizar a execução do método invocado o objeto futuro é automaticamente substituído pelo objeto de retorno. Esse procedimento possibilita que o objeto que invocou um método remotamente possa continuar seu processamento enquanto ocorre o processamento remoto. Caso seja necessária a utilização imediata dos resultados de uma chamada de método remoto, o ProActive possibilita espera por necessidade. A espera por necessidade faz com que a execução de um objeto permaneça bloqueada até que o resultado de uma invocação remota esteja disponível.

O ProActive também oferece migração (BAUDE et al., 2000), a qual permite que um objeto ativo seja migrado, de forma transparente, entre diferentes JVMs. Além de poder mover os objetos entre diferentes JVMs é preciso que se possa comunicar com o objeto migrado independentemente do local onde ele se encontre. O ProActive mantém um rastreamento da localização dos objetos migrados através de um servidor de localização e todo esse processo ocorre de forma transparente ao programador.

A migração em Java pode ser classificada em dois tipos, forte e fraca (BAUDE et al., 2000). Uma migração forte se caracteriza pela transferência de um processo juntamente com seu contexto, isto é, o estado atual da pilha, o valor do ponteiro de programa e todos os objetos relacionados ao processo. Dessa forma, a migração ocorre de forma preemptiva, ou seja, o processo

não precisa tomar conhecimento de que foi migrado. Já a migração fraca envolve apenas os objetos relacionados ao processo migrado e não ocorre de forma preemptiva, ou seja, requer que o processo esteja de acordo em realizar a migração. O estar de acordo em realizar a migração nada mais é do que aguardar a execução de todos os métodos que já haviam sido iniciados e após realizar a transferência do processo. Devido ao modelo de Java, não é possível implementar migração forte sem realizar mudanças no código da JVM (BOUCHENAK; HAGIMONT; PALMA, 2003). O estado de execução de um processo é um dado interno da JVM e não é diretamente acessível aos programadores Java. Por este motivo, a migração oferecida pelo ProActive é do tipo fraca.

Completando as principais características do ProActive tem-se ainda a segurança por ele oferecida (ATTALI; CAROMEL; CONTES, 2003). O ProActive oferece um conjunto de políticas de segurança que vão desde autenticação de comunicação, integridade, confidencialidade até mecanismos de segurança de migração, políticas de segurança hierárquica e políticas de negociação dinâmicas. Estes aspectos, apesar de disponíveis, ainda não foram tratados na versão atual de Cadeo.

3.5 Síntese

A programação em arquiteturas paralelas com memória distribuída não é uma tarefa fácil. Esse capítulo destinou-se a mostrar algumas ferramentas capazes de auxiliar na implementação de programas paralelos para as arquiteturas paralelas alvo do Cadeo. Primeiramente foram apresentadas as bibliotecas de comunicação. Essas bibliotecas oferecem primitivas básicas para o envio e recebimento de dados. A seção apresentou alguns aspectos de três bibliotecas de comunicação: MPI, PVM e DECK.

Buscando suprir grandes demandas por processamento um número cada vez maior de computadores passaram a ser interligados nas arquiteturas paralelas. A própria grade de computadores é uma arquitetura cuja característica inerente é integrar computadores em escala mundial. Para programar uma arquitetura composta por muitos computadores, além da ajuda das bibliotecas de comunicação é preciso administrar tantos computadores. Nesse sentido, foram apresentados exemplos de sistemas gerenciadores de recursos, os quais encarregam-se de oferecer serviços básicos como: controle de disponibilidade e escalonamento dos recursos.

Atualmente, existe uma forte iniciativa para a utilização de linguagens orientadas a objetos, em especial Java, na programação das arquiteturas paralelas alvo. Embora ainda apresente

problemas relacionados a seu desempenho, Java é uma linguagem simples, portável e que torna mais produtivo o processo de desenvolvimento de aplicações. Além de Java possuir suporte nativo a programação distribuída (RMI e múltiplos fluxos de execução), bibliotecas como ProActive viabilizam sua utilização para a programação concorrente.

As ferramentas descritas nesse capítulo colaboram para o melhor e mais fácil aproveitamento das potencialidades das arquiteturas paralelas alvo. Mesmo assim, no nível da aplicação paralela, a localização dos processos não é transparente. Essa característica dificulta o processo de desenvolvimento das aplicações. O próximo capítulo descreve o sistema Cadeo que, entre outras características, oferece uma interface simples para o desenvolvimento de aplicações paralelas. No sistema, a localização e a dinamicidade dos computadores que integram sua plataforma de execução é totalmente transparente.

4 SISTEMA CADEO

4.1 Motivação

A evolução das arquiteturas paralelas foi embasada na busca por ganhos de desempenho para a solução de problemas que demandam grande quantidade de processamento. Aliado à busca por ganhos de desempenho almejou-se também proporcionar alternativas mais econômicas para a construção dessas arquiteturas do que a aquisição de supercomputadores. Computadores comuns passaram a ser interligados via rede para executar tarefas de aplicações paralelas e distribuídas. As aplicações paralelas, em sua maioria, utilizam o paradigma de programação com trocas de mensagens, uma vez que o sistema possui memória distribuída.

Mesmo com a existência de bibliotecas de comunicação capazes de oferecer primitivas eficientes para a comunicação entre as tarefas da aplicação, a pré determinação da localização das tarefas faz-se necessária. Dessa forma, perde-se transparência na implementação de aplicações e o modo de idealizar uma aplicação paralela difere muito de uma implementação multiprocessada, tornando a programação paralela mais difícil. Esse problema poderia ser amenizado se fosse possível programar com transparência de localização dos computadores que compõem sistemas distribuídos.

Uma alternativa para o melhor aproveitamento do poder de processamento das máquinas que integram arquiteturas paralelas é a utilização de computadores em ociosidade. Em muitos casos, os períodos de ociosidade ou baixa utilização dos computadores prevalecem sobre os períodos de pleno processamento. Plataformas de execução compostas por computadores ociosos permitem que sejam atingidos bons níveis de desempenho além de melhor explorar o potencial computacional dos equipamentos.

Além da utilização de ciclos em ociosidade, uma recente iniciativa relacionada a sistemas distribuídos busca o compartilhamento de ciclos computacionais. Essa iniciativa é chamada de

computação ponto-a-ponto (P2P) (LO et al., 2004; BUTT et al., 2003; BARKAI, 2001) e vem sendo cada vez mais aperfeiçoada e difundida. O funcionamento de uma rede P2P consiste basicamente do compartilhamento de ciclos computacionais de um par (*peer*) com os demais pares da rede. O par que compartilhou seus ciclos computacionais poderá usufruir dos ciclos dos demais pares da rede quando necessitar. Alguns sistemas já empregam a filosofia das redes P2P para o compartilhamento de recursos, como por exemplo o OurGrid que utiliza um esquema de rede de favores.

Dentro desse contexto, este capítulo tem por objetivo apresentar o sistema Cadeo. O Cadeo objetiva fornecer meios para implementar aplicações paralelas e distribuídas de forma simples e intuitiva. Para isso, o sistema oferece transparência quanto a localização e a dinamicidade dos computadores que integram a plataforma de execução do Cadeo. O Cadeo também oferece estrutura de escalonamento capaz de distribuir computadores e tarefas e de gerenciar a disponibilidade dos computadores da plataforma de execução. Tal plataforma de execução será composta por computadores disponibilizados por arquiteturas paralelas com memória distribuída.

A próxima seção desse capítulo destina-se a apresentar o sistema Cadeo relatando a idéia geral e as definições do sistema. Após tem-se a estrutura do sistema onde será mostrado seu funcionamento e os módulos que o integram. Em seguida será descrito como se deu a implementação do Cadeo, apresentando a interface de desenvolvimento, as decisões de projeto e a implementação dos módulos do sistema juntamente com um exemplo de aplicação.

4.2 Apresentação do Sistema

Em linhas gerais, o sistema Cadeo foi planejado para oferecer uma infraestrutura capaz de tornar a programação paralela e distribuída simples e intuitiva. Com a utilização do Cadeo é possível implementar aplicações paralelas seguindo o mesmo modelo empregado na programação multiprocessada. Isso é possível graças a transparência, tanto de localização quanto de dinamicidade dos computadores, que o sistema oferece às aplicações. O sistema gerencia uma plataforma de execução composta por computadores disponibilizados por arquiteturas paralelas que atenderão demandas de aplicações paralelas e distribuídas. Essa seção destina-se a descrever a idéia geral e as definições do sistema Cadeo, as quais serão melhor detalhadas a seguir.

4.2.1 Idéia Geral

O sistema Cadeo foi idealizado visando conciliar três aspectos: (i) simplicidade na programação paralela e distribuída, (ii) utilização de arquiteturas paralelas e (iii) aproveitamento de computadores disponíveis. Esse sistema objetiva oferecer meios para implementar aplicações paralelas e distribuídas de forma simples e intuitiva. Além disso, o Cadeo oferece transparência na localização das tarefas de aplicações paralelas e distribuídas, onde o mesmo esconde a localização dos computadores que as executam. O sistema garante que as tarefas serão executadas independentemente de onde esteja localizado o recurso que irá processá-la, tudo isso acontecendo automaticamente, facilitando o desenvolvimento de aplicações que utilizam o sistema.

Para o Cadeo, as aplicações que solicitarem poder de processamento para sua execução serão contempladas por um conjunto de computadores. Esses computadores integrarão a plataforma de execução da aplicação. A plataforma de execução oferecida pelo Cadeo é dinâmica, ou seja, seus computadores estarão disponíveis apenas por determinados períodos. Esse comportamento dinâmico é decorrente da natureza de disponibilização de recursos. O sistema faz uso de computadores pertencentes a arquiteturas paralelas disponibilizados por alocação, compartilhamento ou pela detecção de ociosidade. O conjunto de computadores varia constantemente já que os computadores terão diferentes períodos de disponibilidade.

Com essa breve descrição das características do Cadeo percebe-se que ele pode ser aplicado em qualquer tipo de arquitetura paralela com memória distribuída, demonstrando seu grande potencial de abrangência e aplicabilidade. Outro aspecto é que devido ao Cadeo suportar a alta dinamicidade de computadores ele se enquadra ao conceito de grades de computadores, cujo dinamismo é uma de suas características. A próxima seção apresenta algumas definições do sistema que irão contribuir para o entendimento do funcionamento do mesmo.

4.2.2 Definições para o Sistema Cadeo

Durante o projeto do sistema alguns aspectos foram sendo levantados e fizeram-se necessárias algumas definições relativas a ele. Essa seção tem o objetivo de apresentar as definições que foram fixadas especificamente para o sistema Cadeo.

Computadores Potencialmente Ociosos: é um conjunto de computadores compreendido por todo e qualquer computador que pode vir a ser utilizado pelo sistema Cadeo. Em outras palavras são todos os computadores associados ao sistema. Um computador potencialmente ocioso pode ser um computador indisponível, ou um computador ocioso, ou um

computador ocupado conforme as definições a seguir.

Computador Indisponível: é um computador associado ao Cadeo, porém, momentaneamente não disponível ao sistema, ou seja, não poderá receber uma carga de processamento. Isso implica que o computador está executando algum tipo de processamento, por exemplo atendendo a uma solicitação de seu proprietário, que não tenha ocorrido por meio do Cadeo. Este computador não pode ser considerado ocupado pois não está alocado a uma aplicação e sim realizando processamento sem o intermédio do Cadeo.

Computador Ocioso: todo e qualquer computador que esteja apto a receber uma tarefa a ser processada será tido como ocioso no Cadeo. A disponibilidade de um computador pode ter sido constatada tanto por um detector de ociosidade quanto por algum outro meio alternativo que dependerá das restrições do sistema ou domínio ao qual o computador pertença. Um meio alternativo para a disponibilização de computadores ao Cadeo pode ser o uso de um aglomerado por um determinado período concedido pelo sistema gerenciador do mesmo. Ou então pode ser através da alocação de computadores de uma grade através de sistemas gerenciadores de recursos como Globus ou Legion por exemplo. Outra forma de disponibilização de recursos pode ser o compartilhamento de ciclos de processamento em sistemas P2P.

Computador Ocupado: é um computador que está alocado a uma aplicação paralela por intermédio do Cadeo. Quando uma aplicação paralela solicita computadores ao Cadeo, este destinará um conjunto de computadores ociosos para a execução da aplicação. A partir da alocação, os computadores ociosos passam a serem computadores ocupados.

Aglomerado Dinâmico: essa é a denominação dada a conjuntos de computadores no Cadeo. Quando um computador passa a estar disponível ao sistema (ocioso) ele pertence a um aglomerado dinâmico. Como o tempo de disponibilidade dos nós é variável, o aglomerado se apresenta de forma dinâmica, onde os nós passam e/ou deixam de fazer parte do aglomerado constantemente. Esse aglomerado pode ser particionado em dois ou mais, a fim de atender solicitações de aplicações que carecem de recursos. Cada aplicação que execute através do sistema terá à sua disposição um aglomerado dinâmico, sendo que a dinamicidade do mesmo é totalmente transparente à aplicação.

Computador de Último Recurso: é um computador que está sempre disponível e presente em um aglomerado dinâmico alocado a uma aplicação. Um computador desse gênero é inserido no aglomerado dinâmico para garantir que sempre existirá pelo menos um computador disponível a receber processamento. O computador de último recurso evita problemas com a falta de computadores ociosos a serem alocados e evita casos onde o aglomerado ficaria vazio. Outra situação evitada é a perda de todos os computadores do aglomerado antes do final da execução da aplicação. Normalmente, o computador de último recurso é o computador que gerencia a aplicação.

Aplicação Paralela: no contexto do Cadeo, uma aplicação paralela é composta por um conjunto de tarefas, independentes umas das outras e que podem ser executadas concorrentemente. Essas aplicações são conhecidas como aplicações do tipo sacola de tarefas (*bag-of-tasks*) que foram previamente citadas na seção 3.3.4. As tarefas que compõem a aplicação serão distribuídas entre os nós que integram o aglomerado dinâmico, a independência das tarefas facilita a distribuição das mesmas. Maiores detalhes sobre a estrutura das aplicações paralelas serão dadas na seção 4.4 que detalha a implementação do sistema.

Tarefa: corresponde a uma parcela de processamento necessária para completar a execução de uma aplicação paralela. A tarefa deve ser planejada de tal forma a não possuir dependência de outras tarefas e assim possibilitar sua execução de forma concorrente. Entretanto, uma tarefa poderá gerar novas tarefas a serem executadas. Na seção que descreve a implementação do sistema (4.4) será dado maiores detalhes sobre como é a representação de uma tarefa no Cadeo.

4.3 Estrutura do sistema

Essa seção tem por objetivo dar maiores detalhes sobre a estrutura proposta para o sistema Cadeo. A fim de facilitar o entendimento tem-se, primeiramente, uma visão mais abrangente das funcionalidades oferecidas pelo sistema. Após ter sido formada uma base sobre as funcionalidades do sistema serão apresentados os módulos que foram planejados para contemplar todos os objetivos do sistema.

4.3.1 Funcionamento do sistema

Em linhas gerais, o funcionamento do Cadeo acontece da forma descrita a seguir. Quando um computador torna-se disponível ao sistema (ocioso), ele comunica a um gerenciador sobre sua disponibilidade em receber algum tipo de trabalho [1]. Enquanto não houver nenhuma aplicação que necessite de recursos, os computadores disponíveis permanecerão sem realizar processamento. Ao existir uma aplicação paralela a ser executada, o Cadeo, através do gerenciador, destinará algumas das máquinas disponíveis para a execução dessa aplicação. A figura 4.1 apresenta um cenário hipotético do funcionamento do Cadeo. Nela podem ser identificados os computadores potencialmente ociosos que integram o sistema, sendo que, neste caso, estes podem se apresentar em dois estados: ociosos ou indisponíveis. Também tem-se o gerenciador, com a identificação do conjunto de computadores ociosos (disponíveis) e uma aplicação paralela composta pelo seu conjunto de tarefas. Para possibilitar a execução de suas tarefas a aplicação solicita ao gerenciador um conjunto de computadores, a requisição esta indicada na figura 4.1 pela seta no sentido da aplicação para o gerenciador.

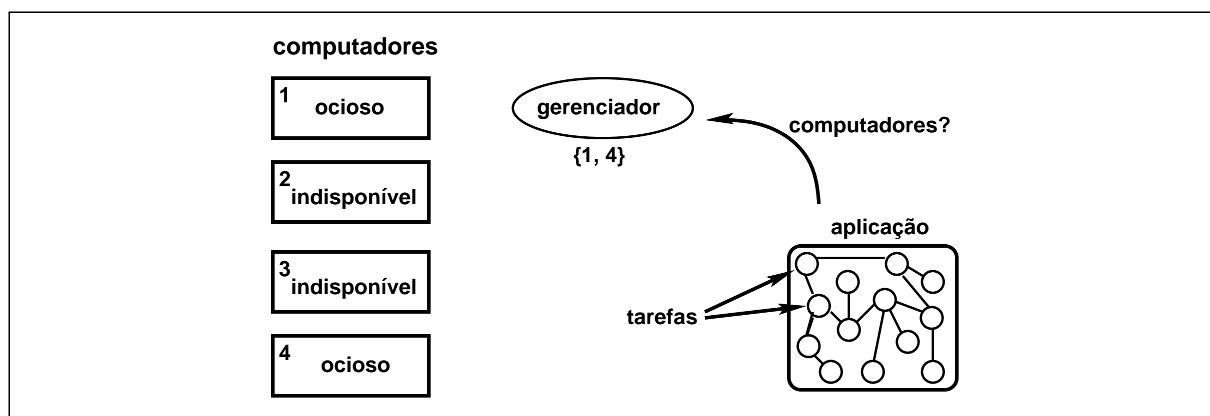

Figura 4.1: Cenário do Sistema Cadeo

O gerenciador, entre outras coisas, tem por tarefa oferecer equilíbrio na distribuição dos computadores disponíveis entre as aplicações paralelas carentes por computadores. A fim de possibilitar esse equilíbrio o gerenciador faz uso de políticas de distribuição. Para definir quantas máquinas serão associadas a uma aplicação o gerenciador pode levar em conta, além da capacidade e da quantidade de máquinas disponíveis, aspectos como a prioridade de aplicações com relação aos computadores existentes ou a quantidade relativa de tarefas associadas às

[1] apesar do gerenciador ser tratado, ao decorrer desse trabalho, como um objeto único, o mesmo deve possuir uma implementação distribuída por questões de desempenho

aplicações.

Um conjunto de computadores associado a uma aplicação paralela, conjunto esse denominado aglomerado dinâmico, pode sofrer alterações através da inclusão ou exclusão de nós. Supondo que uma das máquinas deixe sua condição de ociosidade, por exemplo com a volta da utilização por seu usuário. O sistema tem que permitir à aplicação contornar a perda desse recurso, sem comprometer a sua execução. O gerenciador de recursos também poderá intervir no conjunto de máquinas enquanto ocorre o processamento de aplicações. De acordo com suas políticas de distribuição e balanceamento de cargas o gerenciador poderá adicionar ou remover computadores aos aglomerados dinâmicos a qualquer momento. O comportamento dinâmico dos computadores que compõem um aglomerado dinâmico ocorre de forma transparente aos usuários do Cadeo.

O Cadeo caracteriza-se pela utilização de computadores temporariamente disponíveis. Provavelmente enquanto ocorre a execução de uma aplicação, alguns dos computadores associados a ela deixem de estar disponíveis, ou então, mais computadores podem ser disponibilizados. Em ambos os casos, a execução da aplicação deverá transcorrer normalmente, uma vez que o sistema permite transparência na inclusão e exclusão de computadores ao aglomerado dinâmico.

4.3.2 Módulos do sistema

O Cadeo foi estruturado em três módulos básicos: trabalhador, alocador e escalonador. Cada um desses módulos procura reunir funcionalidades afins e torna fácil seu entendimento e funcionamento. A seguir, cada um dos módulos será apresentado, sendo também apresentado seu comportamento e suas funcionalidades no sistema.

O módulo **trabalhador** é responsável pela execução de tarefas de aplicações paralelas e por anunciar se o computador onde ele se encontra está disponível (ocioso) ou não. Uma cópia do trabalhador estará presente em cada um dos computadores que se disponibilizam a aceitar tarefas externas a serem processadas. Todos esses computadores que possuem um módulo trabalhador compreendem o conjunto de computadores potencialmente ociosos do Cadeo. Quando qualquer um dos computadores entra em estado de ociosidade o seu trabalhador informa a disponibilidade do recurso a um segundo módulo do sistema, o alocador. Na figura 4.2 podem ser visualizados 4 computadores, cada qual com seu módulo trabalhador. Nela o módulo alocador é informado da disponibilidade (ociosidade) do computador 2.

O sistema foi projetado para oferecer um escalonamento em dois níveis. Um primeiro nível

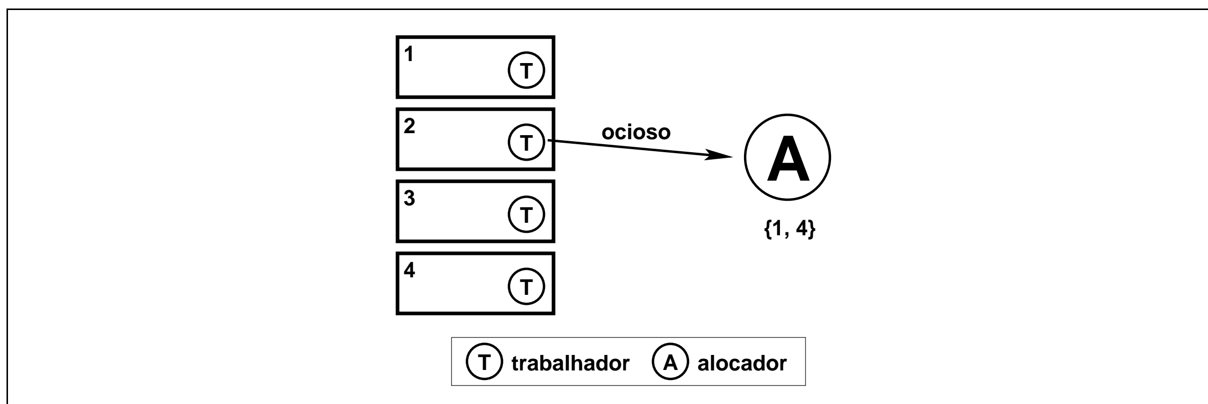

Figura 4.2: Notificação de ociosidade

seria responsável pela distribuição dos computadores disponíveis entre as aplicações paralelas. De posse de um conjunto de computadores disponíveis, um segundo nível de escalonamento realizaria a distribuição de tarefas entre os computadores. O primeiro nível de escalonamento é atendido pelo módulo alocador e o segundo pelo módulo escalonador.

O **alocador** é responsável pela alocação e controle dos computadores ociosos. Na seção anterior, o alocador foi denominado de gerenciador. Este módulo é o núcleo central do sistema Cadeo. Nele, são mantidas referências de todas as máquinas que estão aptas a receber um trabalho a ser executado. Esse conjunto de referências a computadores ociosos compõem um aglomerado dinâmico que permanece em posse do alocador até que haja demanda pelos seus recursos. O aglomerado dinâmico do alocador está representado na figura 4.2 e, com a inclusão de mais um computador, passará a ser {1, 2, 4}. Na versão atual, o Cadeo apresenta um único módulo alocador que centraliza as informações e gerencia a distribuição de todos os computadores que estão disponíveis no sistema. Deseja-se futuramente, que este módulo apresente-se de forma distribuída a fim de atender a um número maior de computadores (sistemas de maior escala como grades) evitando possíveis gargalos do sistema.

O módulo **escalonador** é responsável por distribuir as tarefas de uma aplicação paralela entre os computadores disponibilizados para a sua execução. Para cada aplicação que faz uso do sistema existirá um módulo escalonador associado a ela, o qual servirá como um elo entre a aplicação e o Cadeo. O escalonador proporciona a interação entre a demanda da aplicação paralela e o alocador, uma vez que é o escalonador que realiza o pedido e gerencia os computadores que irão atender a demanda da aplicação. Após o recebimento de computadores o escalonador também proporciona a interação entre o conjunto de tarefas e os recursos que as processarão.

Num primeiro momento, conforme a demanda da aplicação paralela, o escalonador irá solicitar ao alocador um conjunto de computadores. Este conjunto é especificado através de predicados, conforme explicado na seção **??**. O alocador irá avaliar a disponibilidade de computadores do sistema e alocará um aglomerado dinâmico buscando atender a demanda da aplicação. A figura 4.3 mostra a solicitação de computadores por parte do escalonador e a concessão de um aglomerado dinâmico pelo alocador. Os computadores já associados a um aglomerado dinâmico de uma aplicação pertencem exclusivamente àquele aglomerado dinâmico, porém permanecem registrados no alocador. Esse registro identifica a qual aglomerado dinâmico os computadores foram associados, possibilitando posteriores ajustes nos aglomerados.

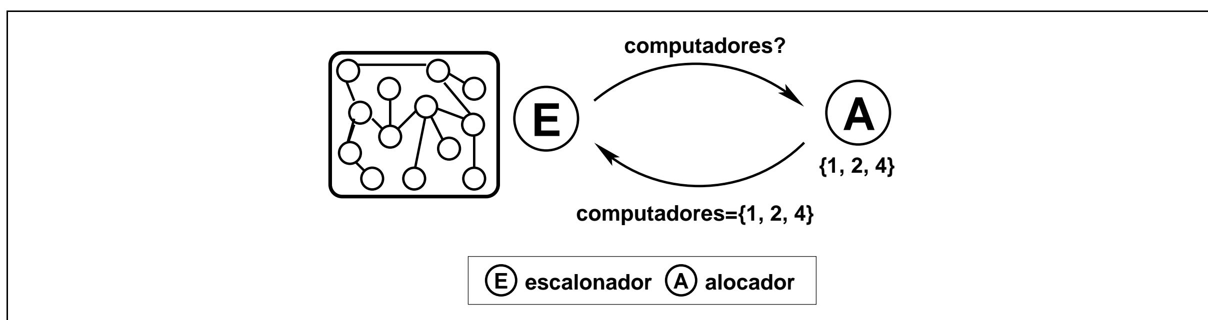

Figura 4.3: Alocação de aglomerado dinâmico

O alocador tem por responsabilidade proporcionar um equilíbrio entre distribuição de computadores disponíveis e a demanda das aplicações paralelas. Para isso o alocador aplica políticas de decisão que buscam melhor atender a todas as demandas por computadores. A princípio, um mesmo computador poderá fazer parte somente de um único aglomerado dinâmico. Porém, para o caso de recursos multiprocessados (arquiteturas SMP) seria interessante poder dividir seus processadores entre diferentes aglomerados dinâmicos. Por esta razão, estuda-se a possibilidade de incluir esta funcionalidade em versões futuras do Cadeo.

Após o alocador destinar um aglomerado dinâmico a uma aplicação, o escalonador estará de posse da identificação do conjunto de computadores pertencentes ao aglomerado dinâmico. O controle do aglomerado dinâmico passa a ser do escalonador que distribuirá as tarefas da aplicação e gerenciará suas execuções. A partir desse ponto, o escalonador se comunica diretamente com os trabalhadores dos computadores que fazem parte do aglomerado dinâmico. Um exemplo da interação entre escalonador e os computadores do aglomerado dinâmico pode ser visto na figura 4.4. Nela, cada um dos computadores do aglomerado dinâmico recebe uma tarefa a ser processada. Ao final da execução de uma tarefa, o resultado encontrado é retornado e o com-

putador estará apto a receber uma nova carga de trabalho. Esse processo se repetirá enquanto houverem tarefas a serem processadas ou enquanto o computador permanecer ocioso.

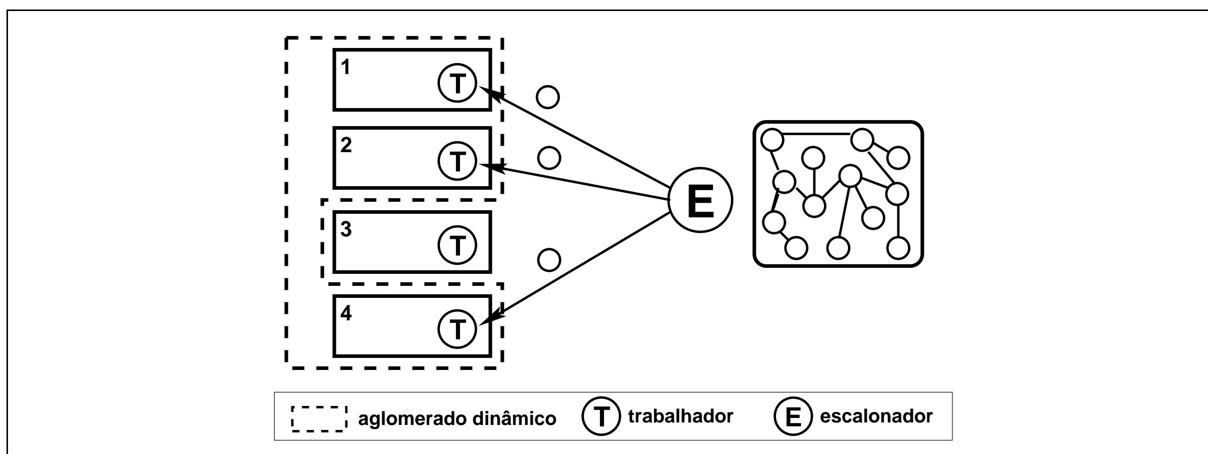

Figura 4.4: Lançamento de Tarefas

Ao ser concluída a execução de todas as tarefas de uma aplicação paralela os computadores serão desassociados daquele aglomerado dinâmico. Assim que receber todos os resultados das tarefas, o escalonador liberará o conjunto de computadores e estes serão devolvidos ao alocador. Esses computadores poderão ser imediatamente repassados para aglomerados dinâmicos que carecem por computadores ou então permanecer em posse do alocador até que uma nova requisição aconteça.

O conjunto de computadores que integram um aglomerado dinâmico não é fixo uma vez que eles podem se tornar ou deixar de estarem disponíveis a qualquer momento. Ao ser identificado o fim do período de ociosidade de um computador, o módulo trabalhador tratará de avisar ao alocador que aquele computador não está mais disponível. Caso o computador esteja alocado a uma aplicação, outra função do trabalhador é providenciar que as tarefas em execução localmente deixem de executar, sendo transferidas ou relançadas em outro computador.

A figura 4.5 ilustra um exemplo das funcionalidades do trabalhador quando acaba o período de disponibilidade de um computador. Nela tem-se a informação passada pelo trabalhador ao alocador sobre a indisponibilidade do computador. Para esse exemplo, a decisão referente as tarefas em execução é migrá-las a outros computadores do aglomerado. A migração está representadas pelos círculos em cinza que estavam presentes no computador 1 que passam para os computadores 2 e 4. Em conjunto com as ações do trabalhador, o alocador, ao receber a notificação do fim da disponibilidade de um computador, verificará se aquele computador está associado a um aglomerado dinâmico alocado a uma aplicação paralela. Em caso afirmativo, o alocador

comunicará a perda do recurso ao escalonador responsável para que não sejam lançadas novas tarefas naquele computador. O alocador também eliminará a referência ao computador não mais ocioso, como pode ser visualizado na figura onde a referência ao computador 1 foi eliminada.

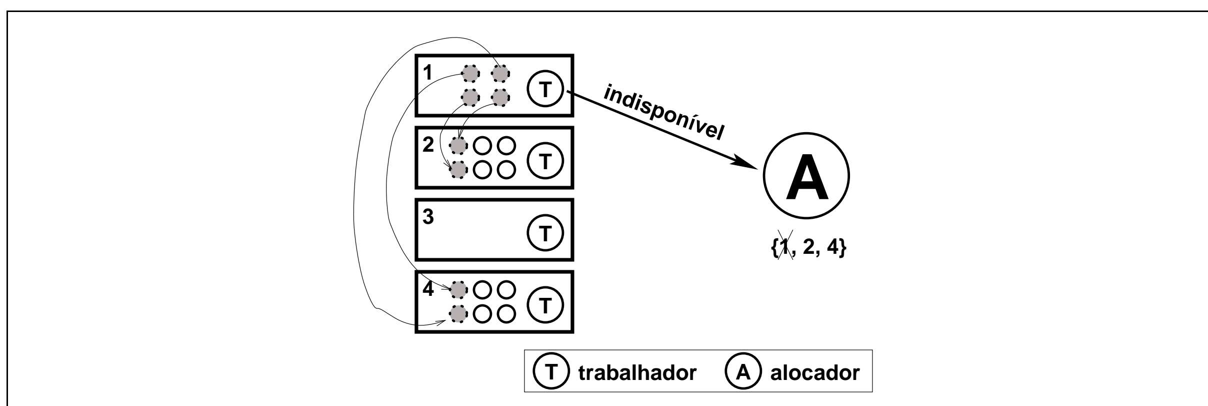

Figura 4.5: Saída de um computador de um aglomerado dinâmico

Uma outra situação que também pode vir a ocorrer é a adesão de novos computadores a um aglomerado dinâmico. Assim como quando da perda de computadores, a execução da aplicação deve prosseguir normalmente. Ao constatar a ociosidade de um computador, o trabalhador notifica o alocador sobre a disponibilidade de mais um recurso. O alocador, caso existam aglomerados dinâmicos carentes, aplicará políticas de alocação para destinar à algum deles o novo computador. Ao reconhecer o novo computador presente no aglomerado, o escalonador passará a lançar tarefas também naquele computador.

4.4 Implementação

Durante o planejamento do sistema Cadeo, primeiramente foi estudado como o sistema funcionaria e como aconteceria a programação utilizando computadores provenientes de diferentes arquiteturas paralelas alvo. Após ser estudado o comportamento esperado do sistema planejou-se uma estrutura que pudesse atender suas funcionalidades e então partiu-se para a implementação do mesmo. Esta seção apresenta como foi realizada a implementação do sistema Cadeo. Primeiramente serão descritas as principais decisões de projeto tomadas durante a implementação. Por fim, tem-se a descrição de como foram implementados os módulos do sistema.

4.4.1 Decisões de Projeto

4.4.1.1 Linguagem de Programação

O primeiro passo para a implementação do sistema Cadeo foi a escolha da linguagem de programação a ser utilizada. As linguagens orientadas a objetos vem sendo cada vez mais utilizadas e oferecem importantes vantagens tais como: polimorfismo, herança, reusabilidade de código, criação de tipos abstratos de dados, entre outras. Tais vantagens facilitam e tornam mais produtivo o processo de desenvolvimento. Almejando simplificar o processo de implementação do Cadeo e deixá-lo flexível, optou-se por uma linguagem de programação orientada a objetos, mais especificamente, Java. Java destaca-se por ser uma linguagem simples, flexível e portável, para maiores detalhes ver seção 3.4.

4.4.1.2 Modelo de Programação

As aplicações paralelas implementadas com o Cadeo serão compostas por um conjunto de tarefas independentes. Ao ser planejado o lançamento dessas tarefas, por se tratar do uso de Java, a solução imediata seria a implementação via invocação remota de métodos (RMI). Porém, o RMI padrão do Java ocorre de forma síncrona, o que implica no bloqueio do objeto que realiza a invocação até que a execução do método no objeto remoto esteja finalizada. A execução síncrona do RMI impossibilita o lançamento simultâneo de tarefas a serem executadas concomitantemente. Sem a possibilidade do lançamento simultâneo, não se conseguiria obter paralelismo na execução das tarefas.

Por esse motivo buscou-se uma alternativa a qual possibilitasse a execução assíncrona de invocação de métodos remotos. Optou-se por avaliar alternativas onde fosse mantida a mesma interface do RMI tradicional do Java. Nas alternativas estudadas, para possibilitar a execução assíncrona das invocações de métodos é utilizado o conceito de objetos futuros (*future*). Quando um objeto realiza uma invocação de método assíncrona, uma instância de uma classe Future é retornada imediatamente ao objeto chamador. Ao final da computação do método seu resultado é inserido dentro do objeto futuro. Dessa forma, o objeto chamador poderá realizar computação concorrentemente com a execução do método remoto. Caso o objeto chamador necessitar do resultado da invocação, ele poderá consultar o objeto futuro em qualquer momento da execução. Além do uso de objetos futuros, um objeto remoto possui um fluxo de execução (*thread*) associado a ele. Esse fluxo de execução controlará o objeto e gerenciará as invocações assíncronas a seus métodos.

A primeira alternativa estudada oferece significantes melhoras no desempenho de invocações de métodos remotos através da seleção dinâmica de protocolos (FALKNER; CODDINGTON; OUDSHOORN, 1999). Ela possibilita invocações assíncronas através de objetos futuros. Porém, esta alternativa faz uso de um pré-compilador para classes de objetos remotos. Como pretende-se que o Cadeo seja o mais simples possível, descartou-se essa opção já ela necessitava de um pré-processador específico. Uma segunda alternativa avaliada foi o mecanismo chamado ARMI (*Asynchronous RMI* (RAJE; WILLIAMS; BOYLES, 1997)). O ARMI foi construído sobre o RMI tradicional do Java e permite a execução concorrente da computação local e remota, através do uso do conceito de objetos futuros. A implementação de programas com o ARMI segue o mesmo modelo empregado quando da utilização do RMI tradicional. Para determinar o assincronismo na invocação dos métodos, o programador deverá usar, ao invés do programa *rmic*, o programa *armic*. Este programa gerará os procuradores (*stubs*) que manipularão o assincronismo da comunicação.

Uma terceira alternativa que proporciona a invocação assíncrona de métodos remotos é oferecida pela biblioteca ProActive (CAROMEL; HENRIO; SERPETTE, 2004). Essa biblioteca foi descrita em maiores detalhes na seção 3.4.2. Da mesma forma que os demais, o ProActive também utiliza-se do conceito de objetos futuros e possibilita espera por necessidade. O ProActive também trabalha com o conceito de objetos ativos (*active objects*) onde seus objetos possuem um fluxo de execução (corpo do objeto) associado a eles. Com o ProActive, os programas são implementados seguindo o mesmo modelo do RMI tradicional e compilado como um programa Java qualquer. O ProActive se encarrega, transparentemente, do registro dos objetos remotos e do assincronismo nas invocações de método, não sendo necessários programas específicos como *rmic* ou *armic*. Embora o ProActive realize por conta própria boa parte do trabalho relacionado às invocações assíncronas, todos os aspectos relacionados a alocação dos objetos é trabalho do programador.

Optou-se pela utilização do ProActive na implementação do Cadeo, por este oferecer uma maior simplicidade na implementação de chamadas de métodos assíncrona. Com a tomada dessa decisão toda a comunicação entre tarefas de uma aplicação paralela será realizada por meio do ProActive. E ainda, toda a comunicação necessária no Cadeo também acontecerá através do ProActive, inclusive a comunicação entre os módulos básicos do sistema. Além do assincronismo de chamada de métodos o ProActive apresenta outras facilidades, citadas na seção 3.4.2, que foram sendo aproveitadas na implementação do Cadeo. Maiores detalhes sobre

tais vantagens e qual a sua influência no Cadeo serão apresentados no decorrer desse texto.

4.4.1.3 Modelo das Aplicações Paralelas

Uma vez que optou-se pela utilização do ProActive buscou-se definir o modelo das aplicações paralelas que utilizam o Cadeo. As aplicações paralelas serão compostas por um conjunto de tarefas independentes. A independência das tarefas possibilitaria a execução concorrente e facilitaria a distribuição das mesmas entre os computadores do aglomerado dinâmico. Transferindo essa idéia de aplicação paralela para o contexto do RMI assíncrono oferecido pelo ProActive, buscou-se avaliar como as tarefas da aplicação seriam representadas.

Uma tarefa de uma aplicação paralela corresponde a uma parcela do processamento esperado daquela aplicação. No contexto da orientação a objetos, as tarefas podem ser representadas pelo processamento realizado em métodos de objetos. Esta aproximação vem ao encontro do modelo de invocação remota de métodos que será usado. Para que seja possível a chamada RMI assíncrona no ProActice, tanto o objeto que realiza a invocação quanto o objeto que executa o método, devem ser objetos ativos (ver seção 3.4.2). Essa característica do ProActive implica que a comunicação deverá acontecer entre objetos ativos, ou seja, que tanto o objeto que lança quanto o que processa as tarefas, deverão ser objetos ativos. Indo um pouco além, pode-se deparar com duas situações: ou as tarefas compreendem as chamadas RMI ou, então, as tarefas seriam os objetos ativos que possuem os métodos invocados.

Caso as tarefas fossem representadas pelas chamadas RMI, a execução das tarefas estaria condicionada a realização da chamada RMI. Em outras palavras, as tarefas seriam dependentes da execução das chamadas remotas. Essa dependência caracterizaria um modelo de programação semelhante ao modelo dividir para conquistar (NIEUWPOORT; KIELMANN; BAL, 2000). Segundo esse modelo, o escalonamento das tarefas aconteceria nas chamadas, ou seja, as chamadas seriam distribuídas entre os computadores disponíveis. É claro que por trás da distribuição das chamadas aconteceria também a distribuição dos objetos ativos com os métodos para processar as tarefas.

Uma outra alternativa seria considerar que as tarefas são representadas pelos objetos ativos, onde as chamadas RMI seriam as interações entre as tarefas. As interações entre as tarefas também representariam dependências entre elas. Porém, um objeto ativo pode ter um fluxo de execução (*thread*) associado a ele e este se encarregaria de realizar as chamadas aos métodos, independente das chamadas RMI. Assim, é possível conceber um modelo de programação do

tipo sacola de tarefas (*bag-of-tasks*, seção 3.3.4) para as aplicações paralelas. Para essa situação, o escalonamento aconteceria durante a instanciação dos objetos, onde os objetos seriam distribuídos entre os computadores disponíveis no aglomerado dinâmico.

Para a implementação atual do Cadeo, optou-se por representar as tarefas por objetos ativos, fazendo com que as aplicações sigam o modelo de sacola de tarefas. Essa decisão foi tomada para facilitar o escalonamento das tarefas, uma vez que é mais fácil escalonar objetos ativos que invocações de métodos.

4.4.1.4 Comportamento na Perda de Computadores

Uma decisão importante tomada durante a implementação do Cadeo diz respeito ao tratamento dado às tarefas quando um computador deixa de estar disponível. As tarefas em execução podem ser abortadas e relançadas futuramente, ou serem mantidas em execução porém com prioridade baixa, ou então, migradas a outros computadores que estejam disponíveis (ver seção 2.5.2). No Cadeo optou-se que sempre que um computador deixe de estar disponível suas tarefas sejam migradas a algum outro computador no aglomerado dinâmico apto a recebê-las. Essa decisão aproveita uma das características do ProActive que é possibilidade de migração de objetos ativos. A representação das tarefas como objetos ativos também favoreceu esta decisão e facilita o gerenciamento das tarefas distribuídas pelos computadores. Maiores detalhes sobre como é realizada a migração das tarefas serão apresentadas na seção que detalha a implementação dos módulos do Cadeo.

4.4.1.5 Enfoque Principal do Trabalho

Como pode-se perceber até aqui, o Cadeo é um sistema que envolve uma série de aspectos. Ele engloba questões desde a descoberta e disponibilização de computadores de diferentes arquiteturas paralelas até políticas de escalonamento e balanceamento de cargas. Como seria inviável ater-se a solucionar todas as questões que envolvem o Cadeo, direcionou-se o trabalho para compor uma estrutura adaptável do sistema. Assim, o enfoque dado a esse trabalho foi a estruturação e criação de uma base funcional para o sistema. Essa base funcional foi construída de tal forma que o tratamento devido as questões remanescentes poderão ser facilmente agregadas no futuro. Na implementação atual o sistema é capaz de oferecer transparência na distribuição e localização das tarefas bem como o suporte necessário a utilização de aglomerados dinâmicos como plataforma de execução. Além disso, procurou-se planejar e desenvolver a base do sistema o mais adaptável possível. Em outras palavras, procurou-se implementar um sistema capaz

de facilmente acoplar novas características e implementações que visem a melhora do sistema.

A base funcional do Cadeo compreende a implementação dos três módulos básicos, alocador, escalonador e trabalhador (ver seção 4.3.2) e da correta interação entre eles. Durante a implementação dos módulos, o ProActive teve um importante papel principalmente para oferecer transparência de distribuição e localização de tarefas. Como o enfoque do trabalho foi a elaboração de uma base funcional para o sistema, muitas das **políticas de decisão** empregadas foram políticas *ad hoc*. O emprego dessas políticas possibilitou o funcionamento do sistema, embora nenhum cuidado tenha sido tomado referente a melhor adequação dessas políticas com o contexto e características do Cadeo. O principal emprego das políticas de decisão *ad hoc* se deu para as questões relacionadas ao escalonamento e balanceamento de cargas das tarefas de aplicações. Em contra-partida, em função de sua estrutura adaptável, o sistema facilmente poderá receber outras implementações de políticas buscando suprir deficiências apresentadas na versão atual.

4.4.2 Implementação dos Módulos

Essa seção apresenta maiores detalhes sobre como foram implementados os três módulos básicos do sistema. Também serão detalhadas as políticas de decisão empregadas e como ocorre a interação entre os módulos do sistema. Para facilitar o entendimento, o texto foi dividido em três seções que detalham cada um dos módulos básicos do sistema.

4.4.2.1 Alocador

O módulo alocador tem um importante papel no Cadeo, uma vez que ele gerencia a alocação de todos os computadores disponíveis ao sistema. Para uma primeira versão do sistema planejou-se a implementação centralizada desse módulo a fim de simplificar o processo de desenvolvimento do mesmo. Porém, tomou-se o cuidado de planejá-lo de tal forma que uma implementação distribuída do módulo pudesse ser desenvolvida futuramente, sem que para isso fossem necessárias alterações na estrutura do sistema. Na versão centralizada são previstas sobrecargas no alocador quando este atender a um conjunto consideravelmente grande de computadores. Uma implementação de alocador distribuída amenizaria tal sobrecarga.

A implementação do módulo alocador foi contemplada através de uma classe Java chamada Allocator. Uma instância dessa classe fornece ao restante do sistema, através de seus métodos, todas as funcionalidades esperadas do módulo. A especificação dessa classe e de seus métodos está descrita com maiores detalhes no Apêndice A. Durante a execução do sistema, existe

um único objeto da classe Allocator instanciado em um dos computadores pertencentes ao sistema (versão centralizada). Os demais módulos do sistema irão fazer requisições aos métodos do alocador para que suas necessidades sejam satisfeitas. Para concretizar uma implementação distribuída, a classe Allocator pode ser reimplementada de forma que vários objetos dessa classe sejam instanciados. Os objetos que necessitam de uma referência a um alocador serão associados a um dos alocadores, o qual poderá melhor atendê-los. Os objetos alocadores interagirão entre si, seguindo um modelo distribuído, por exemplo um modelo hierárquico (não mais centralizado).

Conforme já relatado anteriormente, o Cadeo faz uso da biblioteca ProActive a fim de, entre outras coisas, possibilitar a invocação assíncrona de métodos. Dentro dos conceitos do ProActive, só é possível realizar a invocação assíncrona de métodos se os objetos envolvidos (tanto o objeto que invoca quanto o que executa o método) forem objetos ativos (ver seção 3.4.2). Como deseja-se proporcionar invocação assíncrona de métodos no alocador, o objeto que representa o módulo é um objeto ativo. Também os objetos que representarão os módulos trabalhador e escalonador, os quais realizarão invocações de métodos no alocador, serão objetos ativos.

Para que seja possível o gerenciamento de todos os computadores que estão disponíveis ao Cadeo, os computadores potencialmente ociosos conhecem o alocador e invocam seus métodos conforme necessário. Em virtude disso, o alocador possui dois métodos destinados a adição e exclusão de computadores no conjunto de computadores ociosos. Esse conjunto de computadores ociosos faz parte de um aglomerado dinâmico e está representado no sistema por um objeto da classe DynamicCluster. As instâncias dessa classe armazenam as referências aos computadores disponibilizados ao aglomerado dinâmico e oferecem a implementação dos métodos responsáveis pelas suas funcionalidades. Detalhes sobre a classe DynamicCluster e seus métodos podem ser encontrados no Apêndice B.

A classe Allocator possui o método addWorker( ) que é invocado remotamente toda vez que um computador estiver ocioso. Em outras palavras, quando um computador estiver apto a receber uma parcela de processamento o método addWorker( ) será invocado através do módulo trabalhador. Em posse de um computador ocioso, o alocador terá duas opções. Uma delas é atribuí-lo a um aglomerado dinâmico alocado a uma aplicação e que apresente carência de recursos. Outra opção seria inserí-lo em seu próprio aglomerado dinâmico caso não hajam requisições por recursos. A figura 4.6 apresenta um esquema que representa a informação da disponibilidade de um computador (representado pelo retângulo) através de seu módulo tra-

balhador ao alocador. O alocador possui duas possibilidades para definir o destino do novo computador ocioso. A inclusão de computadores em um aglomerado dinâmico qualquer ocorre através da invocação, pelo alocador, do método addWorker( ) da classe DynamicCluster.

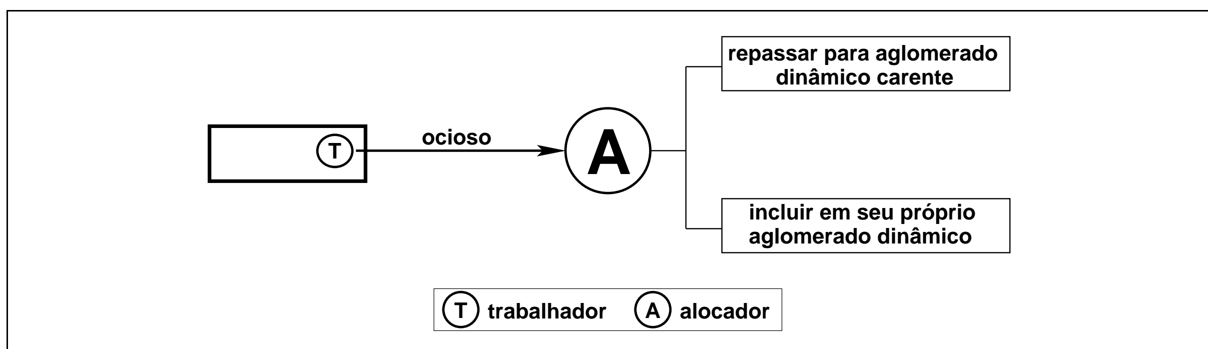

Figura 4.6: Comunicação da ociosidade e destino do novo computador disponível

O método subtractWorker( ) do alocador é invocado na situação contrária, isto é, quando um computador deixa de estar disponível a receber processamento. Nesse caso, o alocador verificará se aquele computador encontra-se em seu aglomerado dinâmico e em caso afirmativo irá excluí-lo. Caso o computador não esteja em posse do alocador, este verificará seus registros para descobrir em qual aglomerado dinâmico o computador fora alocado. Ao descobrir a quem o computador foi atribuído, o alocador transferirá o pedido de exclusão ao aglomerado correspondente.

Completando as funcionalidades previstas para o módulo alocador temos a implementação de mais dois métodos responsáveis pela solicitação e devolução de aglomerados dinâmicos. Toda aplicação que deseje receber computadores para a execução de suas tarefas através do Cadeo deverá solicitá-los através do método getCluster( ). Como resposta a esse método a aplicação receberá um aglomerado dinâmico onde poderão ser lançadas e executadas as suas tarefas. Nesse ponto tem-se uma importante decisão a ser tomada que envolve definir como o alocador irá atender as requisições de aglomerados para proporcionar uma boa distribuição dos computadores e satisfazer a todas as requisições. Na versão atual do sistema, ao receber uma requisição por aglomerado dinâmico o alocador irá transferir todos os computadores ociosos do seu aglomerado dinâmico. Caso venha a receber outra requisição e não possua computadores para atendê-la o alocador poderá re-alocar alguns dos computadores do aglomerado recentemente alocado para o aglomerado carente. Além disso, o aglomerado carente por computadores, terá prioridade no recebimento dos novos computadores ociosos que venham a ser agregados ao

sistema. Essa, muito provavelmente, não é a melhor forma de distribuir os computadores entre as requisições, já que a ênfase dada na escolha dessa solução foi em que a mesma fosse simples, mesmo que não fosse a melhor solução.

Finalizando os métodos da classe Allocator tem-se o método returnCluster( ) que é responsável pela devolução dos computadores pertencentes a um aglomerado dinâmico. Essa devolução acontece ao final da execução da aplicação paralela onde os recursos computacionais não se fazem mais necessários. Nesse método a referência ao aglomerado dinâmico é eliminada já que o mesmo não precisará de novos computadores nem terá computadores do sistema. Os computadores que pertenciam ao aglomerado dinâmico que está sendo devolvido serão re-adicionados (nova invocação do método addWorker( )) ao alocador que decidirá seus destinos.

4.4.2.2 Trabalhador

O módulo trabalhador estará presente em cada um dos computadores potencialmente ociosos do sistema. Este módulo é responsável pela notificação ao alocador da disponibilidade do computador onde se encontra. Além disso, ele é responsável pelo gerenciamento das tarefas de aplicações paralelas que estejam sendo executadas localmente no computador. Esse módulo apresenta um importante papel no sistema, pois representar o elo entre o alocador e os computadores potencialmente ociosos da plataforma de execução (aglomerado dinâmico).

Para possibilitar a implementação do módulo trabalhador foi estruturada a classe Worker. Um objeto dessa classe existirá em cada um dos computadores potencialmente ociosos e sua implementação oferece todos os métodos necessários para atender as funcionalidades do trabalhador. Maiores detalhes sobre a classe Worker, seus atributos e métodos podem ser encontrados no Apêndice C desta dissertação. Assim como o alocador, o objeto que representa um trabalhador no Cadeo será um objeto ativo, possibilitando assincronismo na invocação dos métodos desse objeto.

Durante a instanciação do objeto da classe Worker deve ser passada uma referência ao objeto que atua como alocador. Atualmente, na criação do objeto é informada a URL (*Uniform Resource Locator*) do alocador para que a ligação possa ser realizada. Optou-se por informar previamente a URL do alocador para simplificar o processo de implementação, mas espera-se, futuramente, que os objetos da classe trabalhador possam encontrar automaticamente o alocador. Esse descoberta automática do alocador aconteceria nos moldes de sistemas que utilizam-se de servidores recursos ou serviços (SCHAEFFER FILHO et al., 2004; ARNOLD, 1999) onde o

alocador poderia ser encontrado, versão centralizada, ou o melhor alocador, versão distribuída.

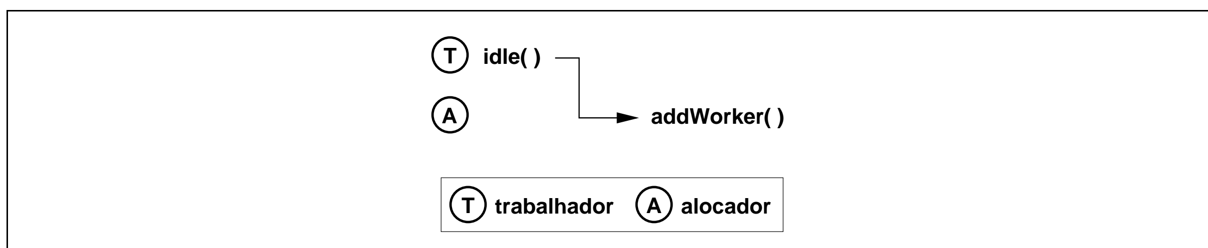

Figura 4.7: Métodos invocados a partir do método idle( )

Dentre os métodos oferecidos pela classe Worker, dois deles encarregam-se de notificar o estado (ocioso ou indisponível) do computador onde se encontram. O método idle( ) é responsável por disponibilizar, ao alocador, as informações do computador onde se encontra. A invocação desse método se dá a partir do momento em que é constatada a disponibilidade (ociosidade) do computador. A disponibilidade de um computador pode acontecer por alocação, detecção de ociosidade ou compartilhamento conforme as características da arquitetura a qual pertence. Com a disponibilidade do computador constatada, o método idle( ) encarrega-se de invocar remotamente o método addWorker( ) do alocador. Em seguida o computador será incluído em um aglomerado dinâmico conforme as opções de destino descritas na seção anterior. Após sua chamada, o computador estará apto a receber tarefas e a processa-las, contribuindo para a execução de uma determinada aplicação paralela. A figura 4.7 ilustra as invocações nos módulos do sistema desencadeadas pelo método idle( ) no trabalhador. Durante a execução do método no trabalhador ocorre a invocação do método addWorker( ) no alocador,representado pela seta na figura, informando que o computador está disponível.

Um segundo método, responsável por informar o fim do período de disponibilidade de um computador, também é disponibilizado. O método busy( ) é invocado quando for identificado o fim do período de disponibilidade (ociosidade) e irá informar ao alocador que aquele computador não está mais disponível. Após a invocação desse método, o computador não receberá mais tarefas a serem executadas. Se existirem tarefas em execução, elas serão migradas para algum computador que possa processá-las. Uma ilustração dos níveis de invocação de métodos desencadeados a partir do método busy( ) pode ser visto na figura 4.8. No nível do trabalhador ocorre a invocação ao método busy( ) e se existirem tarefas em execução será também invocado o método migrateAll( ). A invocação está representada pela seta número 1 e maiores detalhes sobre esse método serão dados a seguir. Através do método busy( ) ocorre a invocação do mé-

todo subtractWorker( ) no alocador informando o fim da disponibilidade do computador (seta número 2).

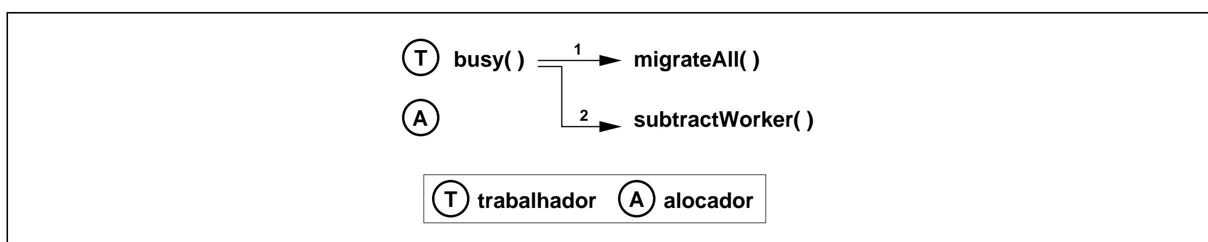

Figura 4.8: Métodos invocados a partir do método busy( )

Nesse ponto percebe-se mais uma vantagem oferecida pelo ProActive, uma vez que ele proporciona a migração de objetos entre diferentes JVMs. Visando aproveitar tal funcionalidade do ProActive e buscando simplificar o controle das tarefas em execução em um determinado computador, optou-se pela migração de tarefas quando terminar o período de ociosidade. Para que seja possível realizar a migração através do ProActive, os objetos que representam as tarefas da aplicação paralela são objetos ativos. Com o uso da migração, a parcela de processamento de uma tarefa já realizada em um computador não será perdida. O restante do processamento necessário será realizado em outro computador que esteja disponível.

A fim de proporcionar a migração das tarefas em execução quando do fim do período de ociosidade de um computador, a classe Worker oferece o método migrateAll( ). Esse método se responsabilizará por migrar todas as tarefas que estavam sendo executadas e é invocado durante a execução do método busy( ), conforme apresentado na figura 4.8. Nesse ponto, outra questão a ser levantada é para onde migrar tais tarefas. A solução dada na versão atual foi transferir essa responsabilidade ao escalonador. Ele informará ao trabalhador qual será o novo destino das tarefas em execução. Como se dá a escolha por esse novo destino será apresentada na próxima seção que detalhará a implementação do escalonador. Além de ser possível migrar todos as tarefas de um computador, da forma mostrada acima, também é possível que uma ou uma parcela do total de tarefas sejam migradas. Essa característica possibilita que sejam realizados ajustes nas cargas de execução dos computadores de um aglomerado dinâmico proporcionando implementações de políticas de balanceamento de cargas. O método da classe Worker que possibilita tal característica é o método migrate( ). Esse método pode ser invocado por qualquer outro objeto do sistema, além de ser invocado quando da execução do método migrateAll( ).

Na versão atual do sistema a migração foi utilizada apenas para transferir as tarefas em

execução de um computador quando da sua exclusão de um aglomerado dinâmico. Embora a migração não tenha sido aplicada para balancear as cargas de trabalho de um aglomerado dinâmico, tomou-se o cuidado de estruturar o sistema para facilmente aceitar tal característica. Porém faz-se necessário um estudo a fim de avaliar se a sobrecarga imposta pelo processo de migração não irá comprometer o desempenho das aplicações no sistema. Atualmente, só é possível realizar migração fraca em Java e esse é o tipo de migração oferecida pelo ProActive (seção 3.4.2). Outra questão que deve ser levada em conta é a influência da dinamicidade do sistema onde podem ocorrer situações críticas. Um exemplo desse tipo de situação seria um computador deixar de estar disponível logo após ter recebido tarefas migradas de outro computador, onde se teria uma dupla sobrecarga de migração para um conjunto de tarefas. Esse tipo de situação poderia ser evitada através do conhecimento de estimativas de tempo de disponibilidade que, em alguns casos (por exemplo quando considera-se computadores ociosos de uma rede) é difícil de ser estimado.

Para finalizar as funcionalidades da classe Worker, ela oferece mais três métodos para a verificação de disponibilidade de execução e registro de tarefas em execução. Dois desses métodos são responsáveis por informar se o computador está apto a receber mais uma tarefa a ser executada. Esses dois métodos possuem o mesmo nome execute( ) variando apenas os parâmetros envolvidos, como pode ser visto no Apêndice C. Se o computador estiver apto a receber uma nova tarefa, o método task( ) será invocado para que a nova tarefa seja registrada. O registro das tarefas acontece para facilitar o controle das tarefas e migrá-las quando se fizer necessário.

4.4.2.3 Escalonador

O módulo escalonador completa os três módulos básicos do Cadeo. Este módulo é responsável pela interação entre a demanda da aplicação paralela e o restante do sistema. Um escalonador estará associado a cada uma das aplicações paralelas que desejarem fazer uso do Cadeo. O módulo, primeiramente, irá solicitar ao alocador um aglomerado dinâmico para servir de plataforma de execução das tarefas da aplicação paralela. Posteriormente, o escalonador controlará a distribuição das tarefas entre os computadores do aglomerado. Além disso, também é tarefa do escalonador controlar os computadores do aglomerado dinâmico bem como informar quais computadores encontram-se menos sobrecarregados.

A implementação do escalonador se deu através da estruturação da classe Scheduler. Maiores detalhes sobre os atributos e métodos dessa classe podem ser encontrados no Apêndice D.

Essa classe possui um objeto da classe DynamicCluster que armazena as informações do aglomerado dinâmico disponível à aplicação paralela controlado pelo escalonador. O escalonador também possui um trabalhador exclusivo associado a ele, chamado de trabalhador de último recurso. O trabalhador de último recurso representa o módulo trabalhador presente no computador de último recurso conforme a definição apresentada em 4.2.2. Este trabalhador receberá tarefas caso nenhum outro computador esteja disponível, ou seja, caso não existam computadores no aglomerado dinâmico. Para possibilitar isso, a classe Scheduler possui um objeto da classe Worker o qual também será inserido como um computador do aglomerado dinâmico. Por ser exclusivo de cada escalonador, o alocador do sistema não tem conhecimento do trabalhador de último recurso.

Para o bom funcionamento do sistema é preciso que o escalonador esteja ligado ao alocador. Por esse motivo, ao ser criado um objeto da classe Scheduler é passado a ele a URL do alocador para que ocorra a sua ligação. Aqui, assim como na implementação da classe Worker, foi adotada a alternativa de passar a URL do alocador único por questões de simplicidade na implementação. As mesmas considerações realizadas na seção anterior sobre a localização automática do alocador valem para a classe Scheduler.

Dando início a descrição das funcionalidades do escalonador tem-se o método start( ). Esse método é responsável por solicitar ao alocador computadores para comporem um aglomerado dinâmico (invocação no alocador do método getCluster( )). O alocador, avaliará a disponibilidade de computadores do sistema e atribuirá um conjunto de computadores ao aglomerado dinâmico solicitado pelo escalonador, conforme descrito na seção 4.4.2.1. Durante a execução desse método será criado o trabalhador de último recurso, o qual é inserido juntamente com os demais computador do aglomerado dinâmico. Ao final da execução da aplicação paralela se faz necessária a liberação dos computadores alocados ao aglomerado dinâmico. Essa liberação do aglomerado dinâmico é responsabilidade do escalonador e, para que isso seja possível, a classe Scheduler oferece o método stop( ). Esse método irá informar ao alocador que não se fazem mais necessários novos computadores para o aglomerado dinâmico alocado àquele escalonador. Além disso, esse método devolve ao alocador todos os computadores que estavam alocados a ele (invocação no alocador do método returnCluster( )).

O conjunto de computadores que o escalonador recebe como plataforma de execução da aplicação paralela é dinâmico, ou seja, novos computadores são inseridos e outros excluídos constantemente. Para que o escalonador consiga administrar tal dinamicidade de recursos foram

implementados dois métodos na classe Scheduler. O método addWorker( ) realiza a inclusão de mais um computador ao aglomerado dinâmico, similarmente ao que acontece no método de mesmo nome presente no alocador. A invocação desse método é realizada pelo alocador após receber um novo computador ocioso.

O método addWorker( ) também registra o computador recém incluso com uma maior prioridade para receber novas tarefas, já que ele acaba de ser incluído e ainda não possui nenhuma tarefa a ser processada. Nesse ponto tomou-se outra decisão de projeto que implica em um novo computador permanecer esperando, sem processamento, até que algum outro computador do aglomerado deseje migrar suas tarefas. Dessa forma não é necessária a interferência no conjunto de tarefas em execução no aglomerado o que simplificou a implementação do sistema. Possivelmente a decisão de manter o novo recurso esperando por tarefas interferirá no desempenho da aplicação paralela pois um novo computador pode permanecer um longo tempo disponível sem realizar processamento. Possivelmente, também, isso poderia ser amenizado se logo após a inclusão de um novo computador ele recebesse pelo menos uma tarefa a ser processada. Porém se o tempo de disponibilidade do novo computador for pequeno, a sobrecarga de receber a migração de uma tarefa e ter que migrá-la em seguida possivelmente interferiria mais no desempenho que manter o computador sem tarefas. A decisão de como fazer esta escolha adequadamente foi deixada para um trabalho futuro.

Outro método oferecido pela classe Scheduler é o método subtractWorker( ). Esse método torna possível a exclusão de computadores do aglomerado dinâmico alocado a uma aplicação. O alocador invocará esse método quando receber um pedido de exclusão de um computador e constatar que o mesmo encontra-se alocado ao aglomerado dinâmico do escalonador em questão. Com esse método o escalonador saberá que o computador não estará mais disponível.

O escalonador também é responsável por indicar quem irá receber a carga de processamento de um computador que está deixando de fazer parte do aglomerado. Para isso, a classe Scheduler possui o método getResource( ), o qual retorna o endereço do computador que está apto a receber tarefas a serem processadas. Para a implementação atual, a política empregada para a escolha do computador que receberá as tarefas enfatiza a busca por computadores com carga de trabalho baixa ou nula. O escalonador mantém uma escala de prioridades entre os computadores do aglomerado dinâmico. Nessa escala, os computadores com menor carga de trabalho terão prioridade para receberem tarefas a serem migradas. Computadores inseridos recentemente no aglomerado dinâmico e que estejam sem tarefas encontram-se no topo da escala de prioridade.

Caso não existam computadores na lista de prioridades, será escolhido, aleatoriamente, algum computador presente no aglomerado. Caso o aglomerado dinâmico esteja vazio, o escalonador indicará o trabalhador de último recurso para receber a carga de trabalho. Após obter o retorno desse método, o trabalhador do computador não mais disponível desencadeia a migração das tarefas para o computador indicado.

4.4.3 Interação entre Cadeo e Aplicação Paralela

A seção 4.4.2 mostrou os principais detalhes da implementação dos três módulos básicos do Cadeo. Conhecendo os detalhes de implementação e o modo de funcionamento do sistema apresentado na seção 4.3.1, esta seção mostrará como deve ser implementada uma aplicação paralela que vise fazer uso do Cadeo. Para facilitar a compreensão será apresentado um exemplo simples de aplicação implementada com o Cadeo.

4.4.3.1 Desenvolvimento de Aplicações Paralelas através do Cadeo

Conforme definido na seção 4.2.2 uma aplicação paralela para o sistema Cadeo é composta por um conjunto de tarefas independentes entre si possibilitando sua execução de forma concorrente. Além de apresentar essa característica, faz-se necessária uma forma de estabelecer a interação entre a aplicação e suas tarefas com o sistema Cadeo como um todo. Buscando oferecer essa interface entre aplicação e sistema Cadeo foi desenvolvida a classe Cadeo. O detalhamento dessa classe pode ser encontrado no Apêndice E desse texto.

Durante o planejamento dessa interface almejou-se que a interação entre sistema e aplicação paralela acontecesse de forma transparente a fim de livrar o usuário de preocupações com detalhes de implementação atendidos pelo sistema Cadeo. Dessa forma, foi possível oferecer aos usuários meios para a implementação de aplicações paralelas de forma simples, clara e intuitiva. A classe Cadeo é composta de três métodos que oferecem toda a infraestrutura necessária para que a aplicação utilize-se do Cadeo. Além dos três métodos a classe possui associada a ela um objeto da classe Scheduler, o qual será o escalonador responsável pela distribuição e controle das tarefas. A presença do escalonador na classe Cadeo tem o papel de ser o elo entre a aplicação e todo o restante do sistema Cadeo.

Toda a aplicação paralela que irá interagir com o Cadeo deverá conter em sua implementação um objeto da classe Cadeo. Durante a criação desse objeto também acontece a criação do objeto que representa o escalonador, para isso a URL do alocador do Cadeo deverá ser informado. Essa informação faz-se necessária para que o escalonador possa localizar o alocador e ligar-se

a ele conforme a decisão de projeto tomada e previamente descrita (ver seção 4.4.2.3). Outro aspecto importante da criação do objeto da classe Cadeo é a possibilidade de que seja usada uma implementação de escalonador diferente da implementação presente no Cadeo. Com isso é possível agregar diferentes implementações de escalonadores e testar diferentes algoritmos de distribuição podendo, por exemplo, atender a objetivos específicos de uma aplicação paralela. Qualquer implementação de escalonador pode ser utilizada, desde que siga a interface original do escalonador.

A classe Cadeo oferece três métodos, os quais poderão ser invocados pela aplicação paralela conforme as situações descritas abaixo. As tarefas de uma aplicação paralela serão distribuídas entre os computadores de um aglomerado dinâmico e por isso são representadas por classes Java. Dessa forma, será possível instanciar objetos (que representam as tarefas) nos computadores remotos e migrá-las quando necessário. Para que seja possível a instanciação remota dos objetos que representam as tarefas a classe Cadeo oferece o método fork( ). Os argumentos esperados por esse método são o nome da classe e os parâmetros para a criação do objeto que representa a tarefa. Como retorno desse método tem-se uma referência ao objeto instanciado remotamente que possibilita a invocação de seus métodos.

Pode-se perceber que, durante o processo de instanciação remota das tarefas, em nenhum momento foi necessário informar em que computador remoto a tarefa estaria sendo criada. Tal fato atende a um dos principais objetivos do Cadeo que é oferecer transparência na localização das tarefas de aplicações paralelas. Tanto a distribuição das tarefas quanto a localização das mesmas é de total responsabilidade do Cadeo isentando o programador desse encargo, o que proporciona considerável facilidade para a implementação das aplicações. Outra importante característica desse método é que ele aceita diferentes implementações de tarefas para uma mesma aplicação, desde que seja mantida a independência entre elas. E ainda, o programador tem a liberdade de criar tantas tarefas quanto forem necessárias para execução da aplicação, sem que o sistema lhe ofereça restrições.

Para proporcionar opções de planejamento com relação ao número de tarefas que serão instanciadas para a execução da aplicação, a classe Cadeo oferecer o método size( ). Esse método retorna o número de computadores disponíveis no aglomerado dinâmico. Vale lembrar que o retorno desse método não é necessariamente preciso, já que o aglomerado dinâmico varia constantemente. O valor retornado poderá ser usado conforme o programador achar conveniente, por exemplo, para re-definir o número de tarefas em função do número atual de computadores

do aglomerado. Completando os métodos da classe Cadeo temos o método stop( ). Este método indica o final da execução da aplicação e faz com que o escalonador libere o aglomerado dinâmico comunicando o fim da execução para o alocador. Pela descrição acima, pode-se notar que o uso do sistema Cadeo ocorre de forma simples, que é baseada na criação de um objeto da classe Cadeo e a invocação de seus três métodos.

4.4.3.2 Exemplo de Aplicação Utilizando o Cadeo

Nessa seção será apresentada uma aplicação bastante simples que utiliza o Cadeo para a instanciação remota de um objeto que representa uma tarefa. A aplicação segue o modelo cliente/servidor e seu objetivo é imprimir uma mensagem em um computador remoto. A implementação dessa aplicação pode ser idealizada a partir de dois objetos sendo que um deles realizará a invocação do método remoto (cliente) e outro que executará o método desejado (servidor). A função do Cadeo é oferecer um destino, ou seja, um computador onde o objeto remoto possa ser instanciado. Uma possível implementação para classe que representa a tarefa remota pode ser vista na figura 4.9.

```
01. public class Servidor {
02.     String mensagem;
03.     public Servidor() {
04.     }
05.     public Servidor(String msg) {
06.         mensagem = msg;
07.     }
08.     public void imprime() {
09.         System.out.println("A mensagem é: " + mensagem);
10.     }
11. }
```

Figura 4.9: Classe que imprime uma mensagem

A classe Servidor exposta na figura 4.9 apresenta um atributo mensagem o qual conterá a mensagem a ser impressa e um método imprime( ) responsável pela impressão. Além disso a classe apresenta dois construtores sendo que um deles é sem argumentos e vazio. Essa é uma característica que deve estar presente em todas as classes geradoras de objetos a serem instanciados remotamente através do Cadeo. Isso porque a instanciação remota de objetos do Cadeo é realizada através do ProActive e as restrições estabelecidas por ele foram herdadas. O ProActive necessita que todas as classes que gerarão objetos ativos (ver seção 3.4.2) possuam um construtor vazio e sem argumentos (CAROMEL, 2004b) para que esse tipo de objeto possa ser instanciado.

A classe Servidor foi implementada sem apresentar nenhum contato com o sistema Cadeo. Ela apenas representa o objeto a ser instanciado e foi planejada como uma classe qualquer Java que oferece um método a ser invocado remotamente. A implementação da instanciação de

um objeto da classe Servidor através do Cadeo está presente na classe Cliente que pode ser visualizada na figura 4.10. Essa classe representa o cliente que realiza a instanciação remota do objeto e, além disso, invoca remotamente o seu método imprime( ).

```
1. public class Cliente {
2.     public static void main(String[] args) {
3.         Cadeo c = new Cadeo("sussurro.inf.ufsm.br");
4.         Object[] param = new Object[]{"Olá Mundo!"};
5.         Servidor s = (Servidor) c.fork(Servidor.class.getName(), param);
6.         s.imprime();
7.         c.stop();
8.     }
9. }
```

Figura 4.10: Classe cliente da aplicação

Observando a figura 4.10 pode-se ver, na linha 3, a criação do objeto da classe Cadeo e é através desse objeto que a aplicação interagirá com o sistema Cadeo. Na linha 4 é construído o parâmetro para criação do objeto remoto, que, nesse caso, corresponde a mensagem a ser impressa remotamente. Na linha 5 acontece a instanciação remota do objeto que representa a tarefa da aplicação. Nota-se que a localização dessa tarefa, ou seja, a identificação do computador onde foi instanciado o objeto, não fez-se necessário. O Cadeo encarrega-se de instanciar o objeto em um dos computadores disponibilizados para a execução da aplicação. O objeto s retornado é uma referência ao objeto instanciado remotamente e os métodos invocados a partir dessa referência serão executados remotamente, seguindo o mesmo modelo do RMI Java tradicional. A invocação remota acontece na linha 6, após essa invocação dá-se por encerrada a aplicação acontecendo a finalização do Cadeo na linha 7.

A aplicação descrita nessa seção é bastante simples e exemplifica a facilidade de utilização do sistema Cadeo. Através da descrição dessa aplicação nota-se que o planejamento da aplicação acontece seguindo o mesmo modelo empregado para elaborar aplicações que utilizam o RMI do Java padrão.

4.5 Síntese

O objetivo desse capítulo foi apresentar o sistema Cadeo contemplando aspectos desde a idéia geral, funcionamento, conceitos envolvidos e planejamento do sistema até a forma como foi implementado. O Cadeo busca proporcionar meios para a utilização de computadores provenientes de arquiteturas paralelas com memória distribuída (redes de estações de trabalho, aglomerados e grades de computadores). Através do uso do Cadeo é possível desenvolver aplicações paralelas que serão executadas sobre uma plataforma composta por computadores que estejam disponíveis em arquiteturas paralelas. As aplicações paralelas serão implementadas de forma

simples, sendo que elas são estruturadas de forma semelhante à aplicações multiprocessadas. Ou seja, o programador não necessita ater-se a localização dos computadores que executarão as tarefas da aplicação paralela. Isso porque o Cadeo oferece a aplicação total transparência na localização dos computadores disponíveis em sua plataforma de execução.

O sistema destina-se a executar aplicações paralelas compostas por um conjunto de tarefas independentes, possibilitando sua execução concorrente. As características desse tipo de aplicação vem ao encontro do tipo de plataforma de execução oferecida pelo Cadeo. Os computadores oferecidos pelo Cadeo pertencem a sistemas distribuídos e permaneceram disponíveis ao Cadeo em determinadas faixas de tempo, essa característica configura uma plataforma de execução dinâmica. Tal plataforma é intitulada no sistema de aglomerado dinâmico e todo o controle referente a inclusão e exclusão de computadores é realizado de forma transparente pelo Cadeo. Isso implica que durante o desenvolvimento de aplicações paralelas não é necessário nenhum tipo de controle relacionado a dinamicidade da plataforma de execução. A transparência oferecida pelo Cadeo, tanto de localização quanto de dinamicidade, possibilitam o desenvolvimento de aplicações paralelas de forma bastante simples e intuitiva.

A forma de disponibilização de computadores das arquitetura paralela ao Cadeo estará diretamente relacionada as características do tipo de arquitetura a qual os computadores pertençam. Como foi descrito nesse capítulo, os computadores que estão disponíveis ao sistema podem ser computadores ociosos provenientes, por exemplo, de redes ou grades. Outra forma de disponibilização pode ser através da alocação de computadores ao sistema sendo estes originários de aglomerados de computadores ou grades por exemplo. Podem, ainda, ser computadores compartilhados seguindo o modelo das redes P2P. Essas diferentes formas de disponibilização irão implicar na dinamicidade da plataforma de execução uma vez que o tempo em que os computadores estarão disponíveis é limitado e variável.

A implementação do sistema foi realizada utilizando a linguagem de programação Java. A escolha por essa linguagem baseou-se nas vantagens oferecidas pelas linguagens de programação orientadas a objetos que favorecem o processo de estruturação e implementação de aplicações. Entre as linguagens orientadas a objetos optou-se pelo uso de Java em função de sua grande popularidade, por ela ser uma linguagem simples e portável e pelo crescente interesse em sua utilização em PAD. Para a coerente estruturação do Cadeo e para possibilitar a execução concorrente das tarefas de aplicações paralelas foi utilizado modelo de programação de RMI assíncrono. Para possibilitar o assincronismo na invocação de métodos optou-se pelo

uso da biblioteca Java para programação paralela, distribuída e concorrente, ProActive. Além do assincronismo o ProActive também oferece outras vantagens, como por exemplo migração de objetos, incorporadas no desenvolvimento do Cadeo. Após a escolha da linguagem na qual o sistema foi implementado, para facilitar o processo de confecção estruturou-se o sistema em três módulos básicos. Sendo que cada um desses módulos é responsável por atender a demandas específicas que envolvem o sistema como um todo.

Os módulos básicos do sistema são o alocador, o trabalhador e o escalonador. O alocador é responsável por controlar os computadores disponíveis ao sistema. Mais especificamente o alocador controla tanto a inclusão e exclusão de computadores no sistema quanto a atribuição, retorno e ajustes em aglomerados dinâmicos. O módulo trabalhador está presente em todos os computadores potencialmente ociosos do sistema e é responsável por informar o estado (ocioso ou indisponível) do computador em que se encontre. Além disso esse módulo também controla as tarefas de uma aplicação paralela em execução naquele computador. Por fim, o módulo escalonador é responsável por controlar os computadores alocados para a execução de uma aplicação paralela. Ainda, o escalonador aplica políticas de escalonamento e balanceamento de cargas a fim de atingir um equilíbrio na distribuição das tarefas que integram a aplicação paralela.

Através da descrição do sistema realizada nesse capítulo pode-se constatar que o mesmo envolve uma série de aspectos e questões importantes a serem consideradas. Em virtude da grande abrangência do Cadeo, para que fosse possível concluir uma primeira versão do sistema foi necessário fixar alguns objetivos específicos para o trabalho. A versão atual do sistema apresenta uma base estrutural que enfatiza a transparência no controle da dinamicidade e localização dos computadores disponíveis no sistema e em oferecer uma interface simples para a implementação de aplicações através do Cadeo. O enfoque da implementação foi desenvolver o sistema de tal forma que fosse possível estabelecer o devido controle dos computadores disponíveis no sistema. Além disso, os módulos do sistema foram planejados para que fosse possível adequá-los a novas políticas de alocação, escalonamento e balanceamento de cargas, já que as decisões empregadas atualmente foram decisões *ad hoc*. Futuramente, com a realização de estudos detalhados que considerem as características dos computadores e de sua disponibilidade ao sistema, será possível, facilmente, substituir as decisões *ad hoc* atuais. Dessa forma também poderão ser solucionados possíveis gargalos e deficiências existentes na versão atual do sistema.

A fim de exemplificar o modo de planejamento e implementação de uma aplicação para-

lela utilizando o Cadeo, este capítulo apresentou um exemplo simples de aplicação que utiliza o sistema. Atendendo a um dos objetivos do Cadeo, esse exemplo mostrou que a forma de implementar aplicações é bastante simples. As aplicações seguem um modelo de programação semelhante ao modelo aplicado em implementações de aplicações RMI tradicional do Java. O próximo capítulo dessa dissertação visa apresentar uma avaliação mais detalhada do sistema Cadeo. Nele será apresentada uma aplicação paralela implementada com o sistema e os resultados obtidos em sua execução.

5 AVALIAÇÃO

5.1 Motivação

O capítulo anterior destinou-se a detalhar aspectos sobre a idéia geral do sistema Cadeo, seu funcionamento, a estrutura proposta, detalhes de implementação e, por fim, um exemplo de aplicação que utiliza o Cadeo. Como pôde ser constatado, durante a explicação do Cadeo foi dada ênfase a mostrar o comportamento esperado e as decisões tomadas durante a implementação do sistema. Devido ao grande número de aspectos que envolvem o sistema englobando uma série de questões importantes, as decisões que foram tomadas afetam diretamente o funcionamento e o desempenho do sistema. Tendo em vista a amplitude do sistema, optou-ser por solucionar algumas dessas questões com uma solução provisória *ad hoc*, uma vez que não seria possível atender a todas as questões adequadamente em tempo hábil.

Este capítulo apresentará uma aplicação paralela construída com o Cadeo e executada sobre um conjunto dinâmico de computadores. Através dos resultados obtidos na execução dessa aplicação será possível avaliar alguns aspectos do funcionamento do sistema. Além disso, será possível constatar as influências das decisões *ad hoc* tomadas durante a implementação do sistema.

5.2 Aplicação Desenvolvida

Para avaliar o funcionamento do sistema Cadeo foi implementada uma aplicação paralela simples. O principal objetivo dessa aplicação foi demonstrar que o sistema realmente consegue administrar, de forma transparente, a localização dos computadores do aglomerado dinâmico disponível para a execução de aplicações. Por esse motivo, optou-se por uma aplicação que possibilitasse a variação do tamanho do grão das suas tarefas. Esta seção destina-se a explicar o funcionamento dessa aplicação, como foi implementada e que métricas de tempo foram

extraídas em sua execução.

5.2.1 Funcionamento da Aplicação

Uma aplicação paralela que utilize o Cadeo deve ser composta por um conjunto de tarefas independentes umas das outras possibilitando sua execução concorrente. A fim de testar o sistema Cadeo idealizou-se uma aplicação bastante simples, a qual realiza um conjunto de operações, sem fim específico, e segue o modelo mestre-escravo. A figura 5.1 ilustra este modelo onde aparece um mestre que coordena um conjunto de escravos os quais executarão as tarefas que o mestre lhes destinar. Os retângulos na figura representam computadores, sendo aceitável a presença de mais de um escravo em cada um deles, o que causará concorrência no processador. Os escravos executarão tarefas da aplicação. A cada tarefa é fornecido um vetor de inteiros. Elas realizam cálculos sobre os elementos do vetor e o retornam como resultado. O vetor de inteiros passado como parâmetro é também retornado para fazer com que o tempo gasto para transferir os dados durante a solicitação da tarefa seja igual ao tempo gasto no retorno da tarefa. As tarefas da aplicação são controladas pelo mestre que informa o vetor, solicita as operações e espera o final das execuções recebendo os resultados.

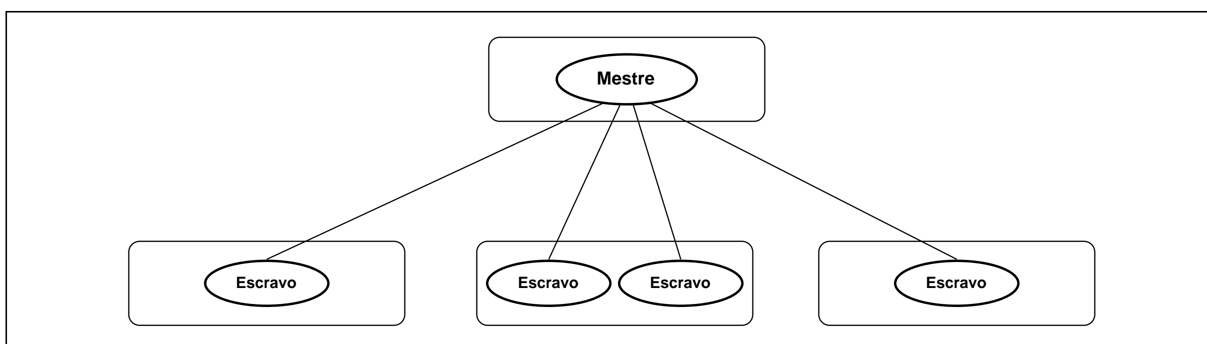

Figura 5.1: Representação do modelo de programação mestre-escravo da aplicação paralela

Um aspecto importante que motivou a escolha dessa aplicação como base para os testes do Cadeo é que, devido a seu comportamento gerenciável, pode-se trabalhar a granularidade das tarefas. Pode-se facilmente aumentar ou diminuir a carga de processamento das tarefas, manipulando sua granularidade. A equação 5.1 representa o cálculo da granularidade de uma tarefa, onde tem-se a razão entre o tempo de computação ($_{comp}$) pelo tempo de comunicação ($_{comun}$) (WILKINSON; ALLEN, 1999). O objetivo em variar a granularidade das tarefas é procurar obter o melhor aproveitamento dos computadores utilizados levando em conta as características do sistema. Isso possibilita que se consiga avaliar a influência da sobrecarga do tempo de

comunicação durante as execuções das tarefas. Partindo-se de valores conhecidos de tempo de execução, pode-se forçar a dinamicidade dos computadores em uso. O tempo gasto a mais (em relação a um grupo fixo de computadores) pode ser considerado como custo dessa dinamicidade.

$$\frac{comp}{comun} \tag{5.1}$$

5.2.2 Implementação da Aplicação

A aplicação paralela, conforme descrito na seção anterior, segue o modelo mestre-escravo. Durante sua implementação foram projetadas três classes Java. Uma delas representa o mestre da aplicação e é responsável por solicitar a instanciação remota e a execução das tarefas e, também, pela espera pelos resultados das tarefas. Tem-se também uma classe representando os escravos da aplicação. Essa classe possui um método que é chamado pelo mestre de modo a ativar sua execução. Completando a aplicação tem-se uma classe que representa os dados de entrada e saída da aplicação.

Uma tarefa a ser processada por um escravo qualquer corresponde a uma invocação de método no objeto escravo presente no computador remoto. O tempo total de execução de uma tarefa é composto pelo tempo gasto na comunicação (invocação do método e retorno do resultado) e pelo tempo de computação da tarefa. Uma representação dos tempos envolvidos na execução de uma tarefa pode ser vista na figura 5.2 (a). Nela aparece o tempo *t1* que representa o tempo de invocação de um método, o tempo *t2* que representa o tempo de computação da tarefa e o tempo *t3* que é necessário para transmitir o resultado. O tempo total de execução da tarefa é obtido pela soma dos tempos *t1*, *t2* e *t3*. A soma de *t1* e *t3* corresponde ao tempo de comunicação da tarefa.

O objetivo da aplicação é executar uma tarefa, como a apresentada, por um determinado número de vezes. A figura 5.2 (b) representa este processo onde a tarefa é realizada em um número n de repetições. Fixou-se o valor de 1000 como padrão para n em todas as execuções da aplicação. Durante o decorrer desse texto, quando for mencionado o tempo de execução da aplicação este se refere ao somatório dos tempos das 1000 tarefas executadas. Outro aspecto que envolve a aplicação é que pode ser executada uma única tarefa em um único escravo (figura 5.2 (b)) ou então, executar mais de uma tarefa simultaneamente nos escravos. A figura 5.2 (c) representa uma execução com mais de uma tarefa em cada escravo. O número de tarefas escolhido para executar em cada escravo é a razão entre 1000 execuções pelo número total de

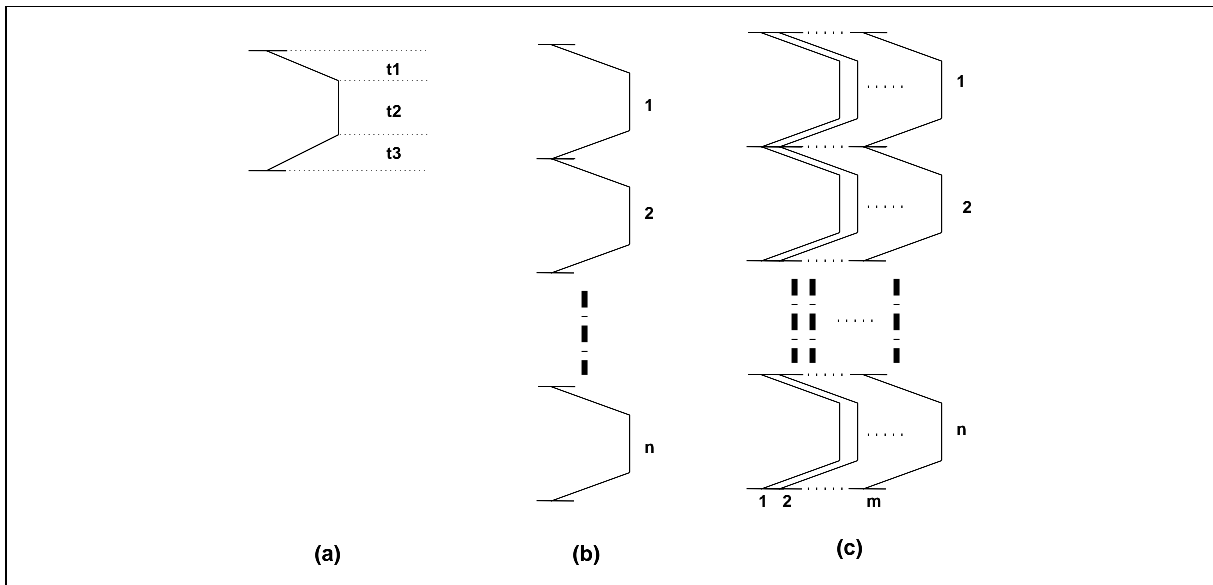

Figura 5.2: Representação de (a) uma tarefa, (b) *n* tarefas consecutivas e (c) *n* tarefas consecutivas com concorrência *m*

escravos disponíveis. Por exemplo, se existirem dois escravos as 1000 tarefas serão divididas entre os 2 escravos onde cada um executará 500 tarefas. Isto permite avaliar problemas de mesmo tamanho em diferentes cenários.

As métricas de tempo foram coletadas em milissegundos. No momento em que um escravo inicia o processamento de uma tarefa é registrado o tempo corrente. Novamente é realizado o registro do tempo ao final da execução da tarefa, sendo que o decréscimo do tempo final pelo tempo inicial corresponde ao tempo de computação da tarefa (conforme ilustrado na equação 5.2). No momento em que lança a tarefa o mestre registra o tempo corrente e ao receber o retorno ele registrará novamente o tempo. O tempo total gasto na execução da tarefa será obtido pela subtração do tempo final pelo inicial, segundo a equação 5.3. Como o tempo gasto pelo escravo na computação da tarefa é conhecido, ao extraí-lo do tempo total de execução registrado pelo mestre tem-se o tempo de comunicação (equação 5.4).

$$comp \qquad final_{escravo} - incial_{escravo} \tag{5.2}$$

$$total \qquad final_{mestre} - incial_{mestre} \tag{5.3}$$

$$comun \qquad total - comp \tag{5.4}$$

Após ter sido planejado como ocorreria a execução da aplicação buscou-se meios para verificar a influência oferecida pela utilização do sistema Cadeo quando comparada a utilização direta do ProActive. Estimava-se que esta influência seria muito pequena e para comprovar isso, foram implementadas duas versões da aplicação paralela. Uma delas foi implementada diretamente sobre o ProActive e outra utiliza o sistema Cadeo. As aplicações possuem comportamento idêntico, uma vez que ambas realizam os mesmos cálculos e utilizam-se dos mesmos tipos de dados e métodos. Os canais de comunicação utilizados durante a execução das versões da aplicação são os mesmos para ambas (já que o Cadeo faz uso do ProActive para estabelecer a comunicação). O conjunto de computadores utilizados nas execuções foi sempre o mesmo para ambas as versões. Isso faz com que os tempos obtidos possam ser comparados uma vez que a única diferença entre as execuções das versões da aplicação é a interface utilizada.

5.3 Resultados

Definidas as implementações da aplicação paralela, foram realizados uma série de testes sobre um mesmo conjunto de computadores. Os resultados encontrados nesses testes bem como o ambiente de execução utilizado serão apresentados nessa seção.

5.3.1 Ambiente de execução

A aplicação paralela foi executada sobre o aglomerado de computadores do Laboratório de Sistemas de Computação (LSC) da UFSM. As máquinas do aglomerado utilizadas foram 7 computadores Pentium III duais de 1GHz, 768 MB de RAM, 20 GB de disco rígido e a rede utilizada foi uma Fast Ethernet. O aglomerado em questão é uma plataforma de execução estática e, em geral, de uso exclusivo a um usuário. Para simular um comportamento dinâmico foram artificialmente provocados períodos de disponibilidade alternados por períodos de não disponibilidade. Esta simulação se deu de uma forma bastante simples, porém suficiente para atender as necessidades dos testes. Do conjunto de computadores do aglomerado dinâmico um deles é escolhido aleatoriamente e desencadeia-se sua exclusão do aglomerado dinâmico. Após um período de tempo, o computador que se encontra fora do aglomerado dinâmico informa sua disponibilidade e novamente é incluído no aglomerado dinâmico da aplicação. A alternância dos períodos de disponibilidade é feita de tal forma que seja possível adequar um certo número de saídas e entradas de computadores no aglomerado dinâmico durante o tempo de execução da aplicação. A determinação do número de saídas e entradas de computadores segue uma

determinada freqüência que será melhor explicada no decorrer do capítulo.

5.3.2 Dados Coletados

Foram realizadas várias execuções de testes e os dados obtidos serão apresentados nessa seção. Inicialmente será mostrado um estudo para a definição do tamanho do grão das tarefas. Após será avaliada a influência causada pela utilização do sistema Cadeo no desempenho de aplicações, quando comparada a utilização direta da biblioteca ProActive. Por fim, tem-se uma avaliação do comportamento da aplicação quando executada sobre uma plataforma de execução dinâmica.

5.3.2.1 Granularidade das tarefas

Segundo a descrição realizada na seção 5.2.1, a aplicação paralela desenvolvida para testar o funcionamento do Cadeo possibilita que a granularidade das suas tarefas seja variada facilmente. As cargas de processamento das tarefas foram variadas até que fosse possível estabelecer três opções de granularidade. As definições de granularidades foram realizadas entre o mestre e um único escravo. A primeira granularidade é aproximadamente igual a 1, isto é, apresenta um equilíbrio entre os tempos gastos na comunicação e computação ($_{comp} \quad * \ _{comun}$). Outra medida de granularidade testada foi aproximadamente 10, ou seja, o tempo gasto na computação da tarefa é 10 vezes maior que o tempo de comunicação ($_{comp} \quad * \ _{comun}$). E, por fim, buscou-se a granularidade aproximada a 100, ou seja, tempo de computação da tarefa sendo 100 vezes superior ao tempo de comunicação ($_{comp} \quad * \ _{comun}$).

A tabela 5.1 apresenta os valores das granularidades e a média dos tempos encontrados durante a execução da aplicação. A segunda coluna da tabela representa a média dos tempos de computação e a terceira coluna possui as médias dos tempos de comunicação. Ambas as médias apresentadas estão em segundos e representam as médias aritméticas de 1000 execuções de tarefas.

Tabela 5.1: Tempos obtidos na estimativa de granularidades

granularidade	**Computação**	**Comunicação**
1	27.0	27.0
10	270.1	27.0
100	2971.0	30.0

5.3.2.2 Influência da utilização do Cadeo

Uma vez que o Cadeo foi implementado fazendo uso do ProActive, buscou-se avaliar qual seria o aumento no tempo de execução gerado pelo sistema Cadeo. Para isso foram implementadas duas versões da aplicação, conforme descrito na seção 5.2.2, onde tem-se uma delas fazendo o uso somente do ProActive e outra com o Cadeo. Comparando os tempos obtidos nas execuções das duas versões da aplicação foi possível verificar a influência que o Cadeo exerce no desempenho das aplicações.

Como o ProActive possibilita a execução de aplicações em um ambiente estático, os testes foram realizados sobre um conjunto fixo de computadores que estão sempre disponíveis. A estaticidade do conjunto de computadores torna viável a comparação esperada, uma vez que, quando se trabalha com um ambiente dinâmico, muitos fatores influenciam o desempenho da aplicação o que dificultaria o processo de comparação. A configuração inicial para a execução da aplicação foi composta por um mestre e um escravo presente em um computador remoto. A partir daí, o número de escravos foi sendo gradativamente aumentado até atingir um total de 12 escravos. O número de escravos foi fixado em 12 pois existiam a disposição 7 computadores biprocessados. Um deles foi utilizado pelo mestre e os 6 restantes receberam 2 escravos cada um. Com essa configuração foi possível ter-se um escravo destinado a cada processador disponível no sistema.

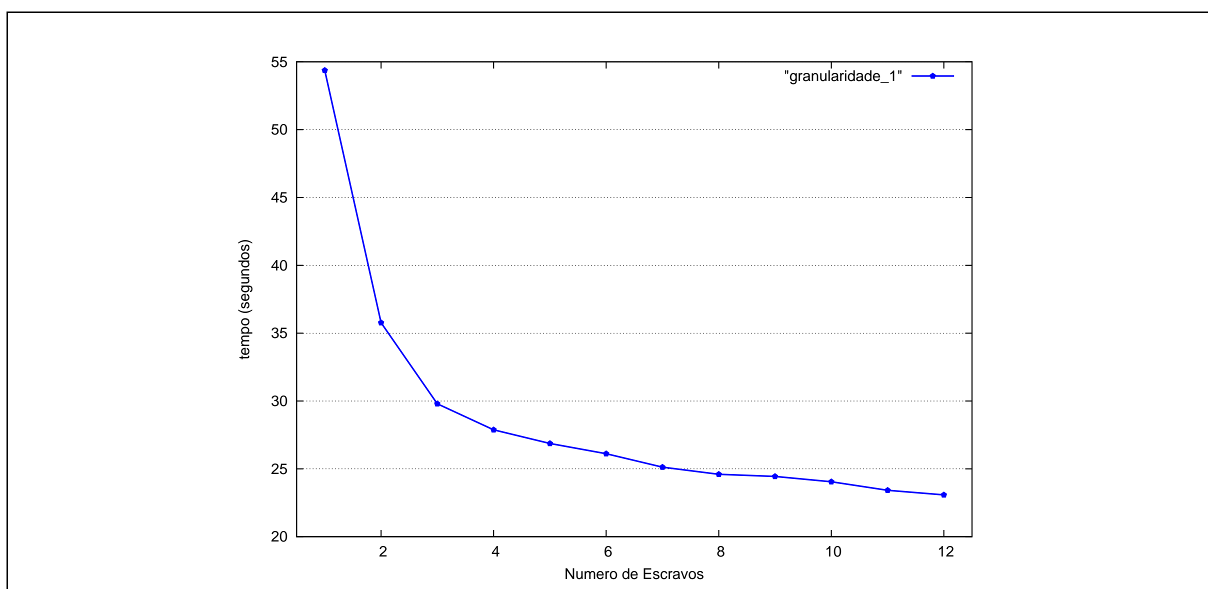

Figura 5.3: Gráfico com as médias dos tempos de execução para a granularidade 1

A figura 5.3 apresenta o gráfico das médias dos tempos de execução para tarefas com gra-

nularidade 1 fazendo uso do ProActive. Cada um dos pontos presentes no gráfico representam as médias, em segundos, de 10 execuções da aplicação para cada número de escravos indicado no eixo horizontal. O gráfico apresenta o comportamento esperado dos tempos de execução da aplicação, onde ocorre uma redução do tempo de execução a cada escravo acrescentado. Essa redução no tempo de execução tende a ser cada vez menor conforme vão sendo inseridos mais escravos. O mesmo tipo de comportamento foi identificado nos testes realizados para as demais granularidades.

Para realizar a comparação entre os tempos obtidos na execução da versão da aplicação implementada sobre o ProActive e a versão implementada sobre o Cadeo, optou-se por avaliar os resultados das execuções da aplicação possuindo 12 escravos. Esta configuração foi escolhida pois ela foi o máximo grau de paralelismo que se pode executar com os computadores disponíveis. A tabela 5.2 apresenta os dados obtidos da execução da aplicação paralela contendo 12 escravos tanto para a versão com ProActive quanto com o Cadeo. Essa tabela possui as médias dos tempos de execução, em segundos, de 10 execuções das duas versões da aplicação para as três granularidades estudadas.

Tabela 5.2: Médias dos tempos de execução com tarefas de diferentes granularidades

granularidade	ProActive	Cadeo
1	23.085	20.447
10	43.043	39.748
100	267.841	265.401

Ao se comparar as médias dos tempos de execução obtidos com a utilização do ProActive e Cadeo pode-se constatar que os tempos com o ProActive foram levemente superiores aos tempos com a utilização do Cadeo. A justificativa para as médias de tempo apresentadas na tabela 5.2 é o maior tempo gasto para a execução da primeira tarefa remota na versão com ProActive. Em ambas as versões da aplicação, ocorre, em uma fase inicial, a instanciação remota dos objetos escravos deixando-os aptos a receberem as invocações de seus métodos. Durante a execução da primeira tarefa em cada um dos escravos na versão com ProActive é gasto um tempo maior para estabelecer a comunicação com a JVM remota. A comunicação da primeira tarefa chega a atingir um tempo 300% maior que a média dos tempos das demais tarefas. Segundo as características da aplicação o tempo de processamento da tarefa mantém-se praticamente constante apresentando um desvio padrão de 1,58 segundos. Já na versão com o Cadeo esse comportamento não ocorre porque a comunicação com JVM remota fôra previamente estabelecida através

do módulo trabalhador presente no computador remoto. Por esse motivo, a primeira tarefa tem um tempo igual ao das demais tarefas e justifica a diferença nos tempos de execução das duas aplicações.

Os dados apresentados na tabela 5.2 comprovam que a utilização do Cadeo, durante a execução de aplicações, não oferece nenhum tipo de sobrecarga ao tempo de execução das tarefas da aplicação. Para conseguir avaliar efetivamente o custo gerado pelos módulos do Cadeo foi realizada uma coleta de tempo diferenciada para este caso. Para a versão da aplicação que utiliza o Cadeo foi, também, medido o tempo necessário para a instanciação remota dos escravos. Dessa forma será possível realizar uma análise mais precisa quanto a utilização do Cadeo. As médias dos tempos obtidos na execução da aplicação de teste estão em segundos e podem ser vistos na tabela 5.3.

Tabela 5.3: Médias dos tempos de execução em diferentes granularidades

granularidade	**ProActive**	**Cadeo**
1	23.085	23.333
10	43.043	43.048
100	267.841	272.088

A tabela 5.3 mostra as médias dos tempos de 10 execuções da aplicação de testes com variação no tamanho da granularidade das tarefas. Como era esperado, os tempos da tabela 5.3 com a utilização do Cadeo foram levemente superiores aos tempos apresentados na tabela 5.2, já que a coleta dos dados passou a considerar mais aspectos da aplicação paralela. Comparando os tempos resultantes da execução com ProActive e Cadeo percebe-se uma diferença muito pequena entre eles. Por trás do processo de instanciação de objetos no Cadeo existe uma estrutura oferecida pelo escalonador para determinar a localização transparentemente. Já com o ProActive foi necessário fixar o conjunto de computadores e fazer a distribuição das tarefas no momento da instanciação. Com os resultados encontrados constata-se que todo o processo que envolve a transparência na localização dos computadores oferecida pelo Cadeo não gera sobrecarga no desempenho das aplicações.

5.3.2.3 Testes com Ambiente Dinâmico

Essa etapa dos testes visa demonstrar a influência da utilização de um ambiente dinâmico. De forma totalmente transparente à aplicação em execução, o Cadeo gerencia um conjunto de computadores dinâmicos possibilitando a inclusão e exclusão de computadores desse conjunto

a qualquer momento. Antes de iniciar essa análise do comportamento do Cadeo, é importante relembrar que na versão atual do sistema foram tomadas algumas decisões *ad hoc* e estas decisões refletirão no desempenho da aplicação durante a utilização do sistema. A maioria dessas decisões são referentes ao escalonamento e balanceamento de cargas das tarefas de aplicações paralelas. Outro fator importante é que, segundo as políticas empregadas na implementação atual, ao ser excluído um computador do aglomerado dinâmico todas as suas tarefas em execução serão migradas para algum outro computador que possa recebê-las. Na implementação atual, para o caso de inclusão de computadores, estes só receberão uma carga de trabalho quando algum outro computador necessitar deixar o aglomerado dinâmico. Todas as decisões tomadas na implementação do sistema foram detalhadas na seção 4.4.2.

A avaliação do comportamento da aplicação paralela de teste em um ambiente dinâmico refletirá a sobrecarga causada pela migração de tarefas. A fim de detectar tal sobrecarga, a aplicação paralela de teste foi executada sobre um aglomerado dinâmico composto por 6 computadores com 2 escravos em cada computador. Como foi descrito na seção 5.3.1, para possibilitar os testes sobre um ambiente dinâmico a dinamicidade do mesmo foi simulada. Para verificar o comportamento do sistema foi estipulado um esquema de freqüências para a dinamicidade dos computadores do aglomerado. No decorrer do texto essa freqüência será referenciada como freqüência de dinamicidade.

Os valores de freqüência representam o número de ocorrências de exclusões de computadores do aglomerado dinâmico durante a execução da aplicação. Por exemplo, uma freqüência igual a 1 representa uma exclusão de um computador do aglomerado dinâmico. Sempre que um computador for excluído, passado um período de tempo aleatório, esse computador será novamente incluído no aglomerado dinâmico. Nota-se que quando ocorre mais de uma exclusão de computadores, estes processos não acontecem concomitantemente, ou seja, apenas um computador estará fora do sistema de cada vez. A escolha do computador que deixará sua condição de ociosidade (ou seja, disponibilidade ao Cadeo) é realizada aleatoriamente.

As tarefas da aplicação utilizadas nos testes possuem a granularidade 100, uma vez que essa granularidade oferece um maior tempo de execução onde se pode perceber melhor os efeitos causados pela dinamicidade do ambiente. O gráfico da figura 5.4 apresenta os tempos encontrados na execução (tempo gasto na execução das 1000 tarefas) da aplicação com variação na freqüência de dinamicidade. O gráfico apresenta uma freqüência de dinamicidade de até 18 inclusões e exclusões de computadores. Para freqüências maiores, o tempo de permanência dos

computadores fora do aglomerado seria muito pequeno. A representação do gráfico mostra o intervalo dos valores encontrados em 10 execuções da aplicação para cada uma das freqüências desejadas, onde o ponto indica a média e os limites do intervalo o maior e o menor tempo obtido.

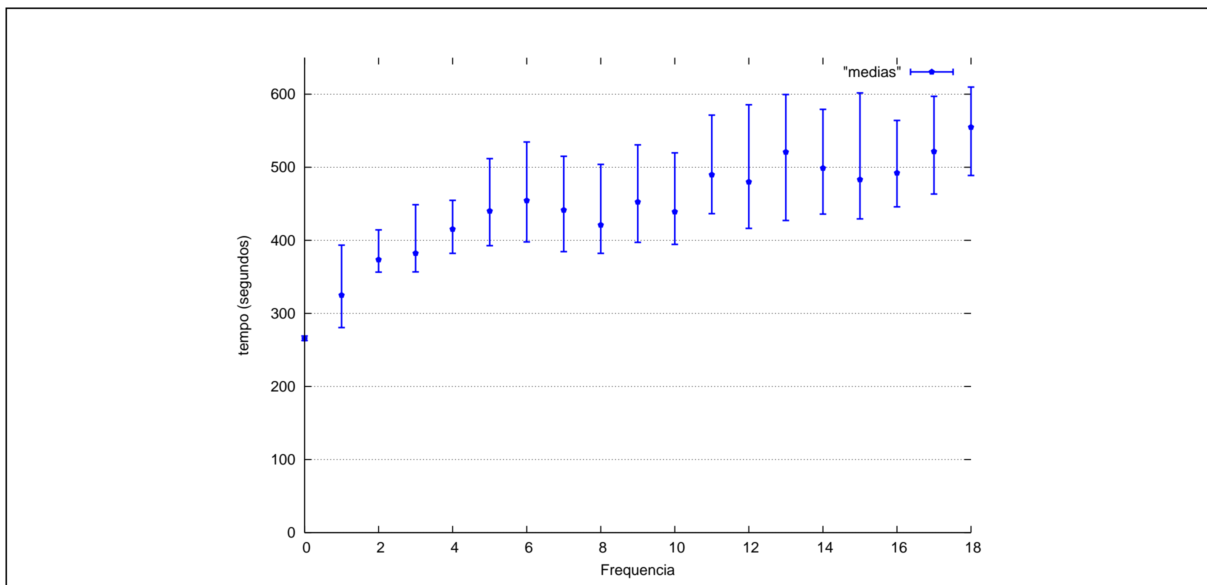

Figura 5.4: Gráfico dos intervalos de tempos de execução sobre um ambiente dinâmico

Pode ser constatado que ocorreu um considerável aumento no tempo de execução quando comparado o tempo obtido com freqüência igual a zero com qualquer outra freqüência de dinamicidade utilizada. Tal comportamento era esperado porque além da aplicação permanecer por períodos executando com um número inferior de computadores do que o número inicial, a migração das tarefas oferece uma sobrecarga ao sistema. Além disso, conforme mencionado na seção 3.4.2, o tipo de migração oferecido pelo ProActive é a migração fraca se fazendo necessária a espera pelo fim da execução de métodos que já tenham sido iniciados para realizar efetivamente a migração das tarefas. Isso implica que mesmo não estando mais disponível um computador permanecerá processando uma chamada de método da aplicação. Esse comportamento pode ser indesejado quando o sistema servir a computadores de uma rede onde seus usuários podem requerer o computador a qualquer momento desejando obtê-lo totalmente disponível imediatamente.

Ao observar o gráfico percebe-se que os tempos comportaram-se de forma bastante inconstante conforme a variação da freqüência e também percebe-se que o gráfico possui uma tendência de elevação dos tempos de execução. Essa variação de tempo das aplicações decorre da política de distribuição *ad hoc* empregada. Quando um computador deixa o aglomerado dinâ-

mico ele irá distribuir suas tarefas entre os computadores restantes no aglomerado. O critério para escolha dos computadores que receberão as tarefas busca primeiramente atribuir tarefas aos computadores que encontram-se sem executar nenhuma tarefa. Se não existir nenhum computador nessa condição o sistema escolherá aleatoriamente computadores do aglomerado dinâmico migrando a eles as tarefas. Em último caso, se não existirem computadores no aglomerado dinâmico as tarefas serão passadas ao computador de último recurso da aplicação.

A política de distribuição empregada não procura manter um equilíbrio entre as cargas dos computadores e podem ocorrer situações onde um determinado computador processa um número maior de tarefas que os demais computadores do aglomerado dinâmico. Esse desequilíbrio reflete diretamente no tempo de execução da aplicação como está representado pelo intervalo dos tempos. Nesse ponto torna-se saliente a importância de realizar um estudo sobre políticas de distribuição e balanceamento de cargas direcionado as características dos ambientes dinâmicos. Uma política de balanceamento de cargas poderia resolver situações como a descrita anteriormente procurando manter equilíbrio entre as cargas dos computadores do sistema. Porém, a migração de tarefas oferece uma sobrecarga de tempo a aplicação sendo um forte ponto a ser considerado para que se consiga melhorar o desempenho sobre ambientes dinâmicos.

5.4 Síntese

Este capítulo destinou-se a avaliar o comportamento do sistema Cadeo utilizado na implementação de aplicações paralelas. Para isso foi desenvolvida uma aplicação de teste bastante simples que seguiu o modelo mestre-escravo. Em suma, a aplicação possui um mestre que é responsável por lançar e receber os resultados de um conjunto de tarefas. A execução das tarefas representa a realização de uma série de cálculos sobre um vetor de dados, as quais serão processadas remotamente por um conjunto de escravos. Essa aplicação não tinha nenhum fim específico mas possuía uma importante característica que é a possibilidade de facilmente alterar a carga de processamento das tarefas. Em virtude dessa característica, pôde-se variar as granularidades das tarefas onde foram definidos três tamanhos de granularidades, sendo deles 1, 10 e 100. Com esses três tamanhos de granularidade foi possível testar o comportamento do sistema sofrendo uma maior ou menor influência do tempo gasto na comunicação durante a execução das tarefas.

Pelo fato de o sistema Cadeo ser implementado utilizando os meios de comunicação oferecidos pelo ProActive, desejou-se descobrir qual seria a influência oferecida pelo uso do Cadeo

quando comparada a utilização direta da biblioteca ProActive. Para isso, foram implementadas duas versões idênticas da aplicação sendo que uma delas utiliza o Cadeo e a outra o ProActive diretamente. Ambas foram executadas sobre uma plataforma de execução estática uma vez que o ProActive não suporta automaticamente uma plataforma dinâmica e também para facilitar as comparações. Analisando os tempos obtidos pode-se constatar que a utilização do Cadeo não causa sobrecarga ao desempenho da aplicação.

Depois de avaliada a influência causada pelo uso do Cadeo em aplicações estáticas, buscou-se avaliar qual seria o comportamento das aplicações diante de um ambiente dinâmico. Para isso a aplicação foi executada sobre um conjunto de computadores que, por simulação, mantinham-se alternando períodos de disponibilidade com não disponibilidade. Estes períodos foram formalizados em um esquema de freqüência de dinamicidade onde um único computador permanecia fora do aglomerado dinâmico de cada vez. Pode-se constatar que independente da freqüência de dinamicidade utilizada houve um acréscimo no tempo total de execução da aplicação. Esse acréscimo era esperado uma vez que a dinamicidade do ambiente afeta diretamente a distribuição da carga de processamento da aplicação e, também, porque não foram buscadas as formas mais eficazes de realizar a distribuição e o balanceamento dessas cargas. E, principalmente, porque o sistema migra as tarefas dos computadores que deixam de fazer parte do aglomerado dinâmico. O processo de migração despende uma certa quantia de tempo além de só ocorrer ao fim do processamento de invocações de métodos já iniciadas, uma vez que a migração oferecida pelo ProActive é migração do tipo fraca.

A execução dos testes sobre um ambiente dinâmico também comprovou que o sistema Cadeo consegue administrar de forma satisfatória a dinamicidade dos computadores. Além de coordenar os recursos disponíveis, o Cadeo também garante que uma execução de aplicação sempre resultará em êxito, sem que tarefas sejam perdidas com a exclusão de computadores no aglomerado dinâmico. O desempenho obtido com a utilização do Cadeo foi diretamente influenciado pelas decisões de projeto tomadas em sua implementação, tais como a utilização de migração e o emprego de políticas de escalonamento *ad hoc*. Esse desempenho poderá ser melhorado futuramente, uma vez que está previsto um estudo aprofundado onde serão buscadas políticas mais propícias a serem empregadas.

6 CONCLUSÃO

As arquiteturas paralelas com memória distribuída oferecem grande quantidade de processamento por meio da agregação de computadores comuns, interligados via rede, e que funcionam como se fossem um único computador. A grande vantagem dessas arquiteturas é que elas são economicamente acessíveis. É mais barato agregar computadores comuns do que investir no desenvolvimento de novas arquiteturas de *hardware* . Elas têm evoluído constantemente e têm se tornado cada vez mais populares. Porém, a programação sobre esse tipo de plataforma de execução não é simples. Existe uma carência grande no que se refere a *softwares* que facilitem a implementação de aplicações para arquiteturas paralelas desse gênero. Entre os motivos que dificultam a programação está o fato de que a localização dos computadores que integram as arquiteturas não é transparente.

Para possibilitar o desenvolvimento de aplicações paralelas de forma simples e intuitiva idealizou-se o sistema Cadeo. O Cadeo oferece um modelo de programação para arquiteturas paralelas com memória distribuída semelhante ao modelo de programação de aplicações multiprocessadas. Para que isso fosse possível, o sistema oferece total transparência quanto a localização dos computadores. A plataforma de execução do Cadeo é composta por computadores de arquiteturas paralelas disponibilizados conforme as características inerentes dessas arquiteturas. Com o uso do sistema é possível construir aplicações paralelas que utilizam computadores pertencentes a diferentes arquiteturas paralelas de forma transparente. O sistema foi implementado em Java e, para atender a todas as necessidades, faz uso da biblioteca ProActive para obter assincronismo na invocação de métodos e migração de objetos.

Por fazer uso de computadores que estejam disponíveis em sistemas distribuídos, a plataforma de execução do Cadeo é dinâmica, ou seja, computadores permanecem constantemente sendo incluídos e excluídos da plataforma. Convencionou-se chamar tal plataforma de aglomerado dinâmico e o Cadeo é capaz de gerenciá-lo de forma transparente, sem que a dinamicidade

do aglomerado afete o modo de implementar as aplicações. Dessa forma, o Cadeo é capaz de fornecer uma interface bastante simples para desenvolver aplicações paralelas. É a transparência, tanto de localização quanto de dinamicidade, oferecida pelo sistema que possibilita a implementação de aplicações paralelas de forma semelhante a implementação de aplicações multiprocessadas.

Uma vez que o sistema administra transparentemente todos os computadores pertencentes ao aglomerado dinâmico é responsabilidade dele também escalonar os computadores. Para melhor controlar o escalonamento necessário ao sistema o Cadeo possui dois níveis de escalonamento: o escalonamento de computadores e o escalonamento de tarefas. Decidiu-se por esta estrutura de escalonamento pois acredita-se que seja possível proporcionar um melhor equilíbrio na distribuição atacando o problema separadamente.

Em virtude do grande número de fatores envolvidos por trás do desenvolvimento do sistema Cadeo, optou-se por fixar alguns objetivos para a primeira versão do sistema. O principal objetivo do trabalho realizado foi planejar e desenvolver uma estrutura básica do sistema, funcional e que suporte a inclusão de adaptações futuras. A versão atual do Cadeo possibilita a implementação de aplicações paralelas de forma simples. Ele é capaz de oferecer transparência tanto da dinamicidade quanto da localização dos computadores, assim como gerenciar o conjunto de computadores disponíveis. Porém, para que fosse possível concluir o trabalho em tempo hábil, o sistema foi implementado fazendo uso de algumas decisões *ad hoc*, principalmente as que se referem a políticas de escalonamento e balanceamento de cargas. Embora tenham sido empregadas soluções *ad hoc* na implementação atual do sistema, foi tomado cuidado para que facilmente pudessem ser acoplar novas decisões em sua estrutura básica. Com isso o sistema fornece meios para que sejam realizados, futuramente, estudos aprofundados e que políticas mais apropriadas as características da plataforma de execução sejam testadas.

Para comprovar o funcionamento do sistema foram realizados testes com uma aplicação paralela. O primeiro objetivo dos testes era avaliar a influência da utilização do Cadeo no desempenho das aplicações. Para realizar a avaliação foram implementadas duas versões idênticas da aplicação, sendo uma delas com o Cadeo e a outra usando diretamente a biblioteca ProActive. Os testes realizados sobre uma plataforma de execução estática comprovaram que a sobrecarga gerada pela utilização do Cadeo foi muito pequena. Principalmente se forem consideradas as vantagens oferecidas pelo Cadeo como por exemplo a transparência de localização dos computadores. Testes realizados sobre um ambiente de execução dinâmico constataram que o sistema

oferece as condições mínimas para a utilização nesse tipo de ambiente. Ou seja, o Cadeo é capaz de administrar um conjunto de computadores dinâmicos de forma coerente e, ainda, garantir que as aplicações terminarão suas execuções com êxito.

O desempenho obtido sobre uma plataforma de execução dinâmica foi baixo, sendo afetado diretamente pelas decisões *ad hoc* utilizadas, como as que envolvem o escalonamento e balanceamento de cargas. Futuramente, espera-se melhorar o desempenho obtido com o Cadeo através da descoberta de soluções mais adequadas às características que envolvem o sistema. Outro aspecto que causa grande influência no desempenho do Cadeo foi a decisão de migrar as tarefas presentes em computadores que deixam de estar disponíveis. Além do custo agregado ao processo de migração de objetos, a migração possibilitada por Java é do tipo fraca. Sempre que se fizer necessário migrar objetos, será necessário esperar o final da execução de métodos que já tenham sido invocados, ou seja, nem sempre a migração ocorre exatamente quando é solicitada. Essa característica pode gerar problemas já que os computadores podem não ser liberados pelo Cadeo imediatamente.

Embora o Cadeo ainda apresente muitas pendências, todos os objetivos propostos para a sua primeira versão foram atendidos. A principal contribuição do trabalho realizado foi o desenvolvimento da estrutura básica do sistema. A versão atual do sistema foi planejada para que possa, futuramente, sofrer adaptações sem que para isso a estrutura do sistema tenha de ser alterada. É possível implementar aplicações paralelas utilizando o Cadeo de forma simples e intuitiva. Isso porque o sistema oferece transparência de localização de computadores e gerencia a dinamicidade da plataforma de execução.

Trabalhos Futuros

Pelo fato do sistema Cadeo estar ainda em sua primeira versão e pela grande variedade de aspectos que envolvem o sistema, é possível enumerar uma série de questões a serem tratadas em trabalhos futuros. A seguir tem-se uma lista das principais possibilidades de trabalhos futuros:

- Desenvolvimento de uma versão distribuída para o módulo alocador para que o Cadeo possa suportar com eficiência arquiteturas paralelas de grande porte.

- Possibilitar que o módulo alocador seja localizado automaticamente pelos módulos escalonador e trabalhador. Associado ao item anterior, será possível, por exemplo, manter um esquema similar a um banco de alocadores. Sendo que o retorno de uma consulta ao banco será o alocador que melhor atenderá as necessidades do módulo que o solicita.

- O Cadeo aceita computadores provenientes de diferentes tipos de arquiteturas paralelas com memória distribuída e em diferentes formas de disponibilização. Atualmente a disponibilização de computadores foi simulada. A estrutura do Cadeo necessita que a informação de disponibilidade parta do módulo trabalhador presente em um computador. Implementar a adaptação dessa característica do sistema as diferentes formas de disponibilização de computadores compreende outra possibilidade de trabalho futuro.

- Atualmente não é mantido nenhum registro das características específicas, tanto de *software* quanto de *hardware* , dos computadores disponíveis ao Cadeo. Essas informações poderiam ser oferecidas quando os computadores passassem a estarem disponíveis ao sistema e poderiam ser empregadas em heurísticas utilizados nas decisões de escalonamento. Além de possibilitar a disponibilização dessas informações, também é necessário encontrar formas ou linguagens de descrição adequadas e flexíveis para a representação das mesmas.

- Realizar um estudo aprofundado para encontrar políticas de escalonamento e balanceamento de cargas que melhor se adaptem as características do sistema. Com esse estudo será possível obter uma melhor distribuição tanto de tarefas quanto de computadores melhorando o desempenho do Cadeo. Também podem ser inseridos aspectos a fim de caracterizar melhor uma requisição por computadores onde um conjunto de características esperadas pudessem ser informadas. Associada a descrição das especificações de cada computador o sistema poderia casar as informações e retornar aglomerados dinâmicos mais próximos das necessidades de aplicações. Essa caracterização dos computadores disponíveis também contribuirá para o escalonamento no nível das tarefas, onde pode ser buscado um equilíbrio das cargas considerando o potencial de cada máquina.

- Encontrar formas de evitar possíveis transtornos causados pela migração fraca de Java. Atualmente, a liberação de computadores que estejam sendo excluídos de um aglomerado dinâmico pode não ser imediata. A solução ideal poderia ser obtida através da migração forte em Java. Enquanto esta última não é possível, a decisão de realizar migração de tarefas poderia ser substituída por alguma outra alternativa que possibilitasse um melhor desempenho para as aplicações paralelas.

- Atualmente, as aplicações paralelas que utilizam o Cadeo devem ser implementadas seguindo o modelo de sacola de tarefas. Seria interessante possibilitar que outros modelos

de programação pudessem ser empregados. Por exemplo, o modelo de dividir para conquistar pode ser adaptado ao contexto do sistema possibilitando que o Cadeo atenda a uma gama maior de problemas.

REFERÊNCIAS

ACHARYA, A.; EDJLALI, G.; SALTZ, J. The Utility of Exploiting Idle Workstations for Parallel Computation. In: ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 1997, Seattle. **Anais. . .** [S.l.: s.n.], 1997.

ADLER, M.; GONG, Y.; ROSENBERG, A. L. Optimal sharing of bags of tasks in heterogeneous clusters. In: ANNUAL ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES (SPAA-03), 15., 2003, New York. **Proceedings. . .** ACM Press, 2003. p.1–10.

AGRAWAL, S.; DONGARRA, J.; SEYMOUR, K.; VADHIYAR, S. NetSolve: past, present, and future - a look at a grid enabled server. In: BERMAN, F.; FOX, G.; HEY, A. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. p.615–624.

ANDERSON, D. P. **SETI@Home Internet Computing**. http://setiathome.ssl.berkeley.edu - último acesso em novembro 2004.

ANDERSON, D. P.; COBB, J.; KORPELA, E.; LEBOFSKY, M.; WERTHIMER, D. SETI@home: an experiment in public-resource computing. **Communications of the ACM**, [S.l.], v.45, n.11, p.56–61, Nov. 2002.

ANDERSON, T. E.; CULLER, D. E.; PATTERSON, D. A. A Case for Networks of Workstations: NOW. **IEEE Micro**, [S.l.], Feb. 1995.

ANDRADE, N.; CIRNE, W.; BRASILEIRO, F.; ROISENBERG, P. OurGrid: an approach to easily assemble grids with equitable resource sharing. In: FEITELSON, D. G.; RUDOLPH, L.; SCHWIEGELSHOHN, U. (Ed.). **Job Scheduling Strategies for Parallel Processing**. [S.l.]: Springer Verlag, 2003. p.61–86. Lect. Notes Comput. Sci. vol. 2862.

ANDRES, C. Enterprise Grid Computing - It isn't just for research anymore. **Linux Magazine - Online**, [S.l.], Oct. 2004.

ANDREWS, G. R. **Foundations of multithreaded, parallel, and distributed programming**. Reading, Massachusetts: Addison-Wesley, 2000.

ARNOLD, K. The JiniTM Architecture: dynamic services in a flexible network. In: DESIGN AUTOMATION CONFERENCE (DAC' 99), 36., 1999, New York. **Proceedings. . .** Association for Computing Machinery, 1999. p.157–162.

ATTALI, I.; CAROMEL, D.; CONTES, A. Hierarchical and Declarative Security for Grid Applications. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING, HIPC, HYDERABAD, INDIA, DECEMBER 17-20, 2003, Springer Verlag. **Anais...** Lecture Notes in Computer Science: LNCS, 2003.

AUMAGE, O. Heterogeneous multi-cluster networking with the Madeleine III communication library. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS '02 (IPPS & SPDP)), 16., 2002, Washington - Brussels - Tokyo. **Anais...** IEEE, 2002. p.85.

BAKER, M.; BUYYA, R. Cluster Computing at a Glance. In: BUYYA, R. (Ed.). **High Performance Cluster Computing**. Upper Saddle River, NJ: Prentice Hall PTR, 1999. v.1, Architectures and Systems, p.3–47. Chap. 1.

BAKER, M.; CARPENTER, B.; FOX, G.; KO, S. H.; LIM, S. mpiJava: an object-oriented java interface to MPI. In: ROLIM, J. et al. (Ed.). **Proceedings of the IPPS/SPDP Workshops on Parallel and Distributed Processing —In Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP'99 (San Juan, Puerto Rico, April 12-16, 1999)**. Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapore-Tokyo: Springer-Verlag, 1999. p.748–762. (LNCS, v.1586).

BAL, H. E.; BHOEDJANG, R.; HOFMAN, R.; JACOBS, C.; LANGENDOEN, K.; RÜHL, T.; KAASHOEK, M. F. Performance Evaluation of the Orca Shared-Object System. **ACM Transactions on Computer Systems**, [S.l.], v.16, n.1, p.1–40, 1998.

BARKAI, D. **Peer-to-Peer Computing**: technologies for sharing and collaborating on the net. Santa Clara, CA, USA: Intel Press, 2001. 332p.

BARRETO, M. E.; NAVAUX, P. O. A.; RIVIÈRE, M. P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented aplication with fault tolerance support. In: IV CONGRESSO ARGENTINO DE CIENCIAS DE LA COMPUTACIóN, 1998, Neuquém, Argentina. **Anais...** Neuquém: Universidad Nacional de Comahue: Facultad de Economia y Administración: Departamento de Informática y Estadística, 1998. p.623–637.

BARRETO, M. E.; ÁVILA, R.; NAVAUX, P. O. A. The Multicluster model to the integrated use of multiple workstation clusters. In: III WORKSHOP ON PERSONAL COMPUTER BASED NETWORKS OF WORKSTATIONS, 2000, Cancun, México. **Anais...** Berlin: Springer-Verlag, 2000. p.71–80. (Lecture Notes in Computer Science, v.1800).

BAUDE, F.; CAROMEL, D.; HUET, F.; VAYSSIERE, J. Communicating Mobile Active Objects in Java. In: HPCN EUROPE 2000, 2000. **Proceedings...** Springer, 2000. p.633–643. (LNCS, v.1823).

BECKMAN, P. **TeraGrid Project**. A Supercomputing Grid Comprising Argonne National Laboratory, California Institute of Technology, National Center for Supercomputing Applications and San Diego Supercomputing Center, http://www.teragrid.org - último acesso em novembro 2004.

BERMAN, F. High-Performance Schedulers. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid**: blueprint for a new computing infrastructure. San Francisco, CA: Morgan Kaufmann, 1999. p.279–309.

BERMAN, F.; FOX, G.; HEY, T. The Grid: past, present, future. In: BERMAN, F.; FOX, G.; HEY, A. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. p.09–50.

BERMAN, F.; HEY, A.; FOX, G. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. 1060p. n.ISBN:0-470-85319-0.

BERMAN, F.; WOLSKI, R.; FIGUEIRA, S.; SCHOPF, J.; SHAO, G. Application-Level Scheduling on Distributed Heterogeneous Networks. In: SUPERCOMPUTING, 1996. **Proceedings...** [S.l.: s.n.], 1996. http://apples.ucsd.edu/hetpubs.html.

BOUCHENAK, S.; HAGIMONT, D.; PALMA, N. D. Efficient Java Thread Serialization. In: ACM INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA (ACM PPPJ'03), 2., 2003, Kilkenny, Ireland. **Proceedings...** [S.l.: s.n.], 2003.

BOZYIGIT, M. History-driven dynamic load balancing for recurring applications on networks of workstations. **J. Syst. Softw.**, New York, NY, USA, v.51, n.1, p.61–72, 2000.

BUNN, J. J.; NEWMAN, H. B. Data-Intensive Grids for High-Energy Physics. In: BERMAN, F.; FOX, G.; HEY, A. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. p.859–906.

BUTT, A. R.; FANG, X.; HU, Y. C.; MIDKIFF, S.; VITEK, J. An Open Peer-to-Peer Infrastructure for Cycle-Sharing. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP'03), 19., 2003, Bolton Landing (Lake George), NY. **Anais...** [S.l.: s.n.], 2003.

BUYYA, R. **High Performance Cluster Computing Vol 1**. Upper Saddle River, NJ: Prentice Hall, 1999.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. An Economy Driven Resource Management Architecture for Global Computional Power Grids. In: THE 2000 INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS (PDPTA 2000), 2000, Las Vegas, USA. **Anais...** [S.l.: s.n.], 2000.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. In: THE 4TH INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING IN ASIA-PACIFIC REGION (HPC ASIA 2000), 2000, Beijing, China. **Anais...** IEEE Computer Society Press: USA, 2000.

CAROMEL, D. Toward a method of object-oriented concurrent programming. **Communications of the ACM**, [S.l.], v.36, n.9, p.90–102, Sept. 1993.

CAROMEL, D. **Oasis Group at INRIA Sophia - Antipolis. ProActive, the Java libary for parallel, distributed, concurrent computing with security and mobility**. Disponibilizado em http://www-sop.inria.fr/sloop/javall/index.html. Atualizado em maio de 2004.

CAROMEL, D. **ProActive Installation & User Guide**. http://www-sop.inria.fr/sloop/javall/index.html.

CAROMEL, D.; HENRIO, L.; SERPETTE, B. Asynchronous and Deterministic Objects. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 31., 2004. **Proceedings...** ACM Press, 2004. p.123–134.

CAROMEL, D.; KLAUSER, W.; VAYSSIERE, J. Towards Seamless Computing and Metacomputing in Java. In: CONCURRENCY PRACTICE AND EXPERIENCE, 1998. **Anais...** Wiley Sons: Ltd., 1998. v.10, n.11–13, p.1043–1061. http://www-sop.inria.fr/oasis/proactive/.

CASANOVA, H.; OBERTELLI, G.; BERMAN, B.; WOLSKI, R. The AppLeS Parameter Sweep Template: user-level middleware for the grid. In: SUPERCOMPUTING'2000 (CD-ROM), 2000, Dallas, TX. **Proceedings...** IEEE and ACM SIGARCH, 2000.

CHAPIN, S. J.; KATRAMATOS, D.; KARPOVICH, J.; GRIMSHAW, A. Resource management in Legion. **Future Generation Computer Systems**, [S.l.], v.15, n.5–6, p.583–594, 1999.

CHETTY, M.; BUYYA, R. Weaving Computational Grids: how analogous are they with electrical grids? **Computing in Science and Engineering**, [S.l.], v.4, n.4, p.61–71, July/Aug. 2002.

CHIEN, A.; CALDER, B.; ELBERT, S.; BHATIA, K. Entropia: architecture and performance of an enterprise desktop grid system. **Journal of Parallel and Distributed Computing**, [S.l.], v.63, n.5, p.597–610, May 2003.

CIRNE, W.; BRASILEIRO, F.; SAUVé, J.; ANDRADE, N.; PARANHOS, D.; SANTOS-NETO, E.; MEDEIROS, R. Grid Computing for Bag of Tasks Applications. In: THIRD IFIP CONFERENCE ON E-COMMERCE, E-BUSINESS AND E-GOVERMENT, 2003. **Proceedings...** [S.l.: s.n.], 2003.

CIRNE, W.; MARZULLO, K. OpenGrid: a user-centric approach for grid computing. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBACPAD 2001), 13., 2001, Pirenópolis, GO. **Anais...** [S.l.: s.n.], 2001.

CIRNE, W.; PARANHOS, D.; COSTA, L.; SANTOS-NETO, E.; BRASILEIRO, F.; SAUVé, J.; SILVA, F. A. B. da; BARROS, C. O.; SILVEIRA, C. Running Bag-of-Tasks Applications on Computational Grids: the MyGrid approach. In: ICCP'2003 - INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2003, Kaohsiung, Taiwan, ROC. **Proceedings...** [S.l.: s.n.], 2003.

COSTA, L.; FEITOSA, L.; ARAúJO, E.; MENDES, G.; COELHO, R.; CIRNE, W.; FIREMAN, D. MyGrid - A complete solution for running Bag-of-tasks Applications. In: SIMPóSIO BRASILEIRO DE REDES DE COMPUTADORES, 22., 2004, Gramado, RS. **Anais...** SBC, 2004.

CULLER, D. E.; ARPACI-DUSSEAU, A. C.; ARPACI-DUSSEAU, R.; CHUN, B. N.; LUMETTA, S. S.; MAINWARING, A. M.; MARTIN, R. P.; YOSHIKAWA, C. O.; WONG, F. Parallel Computing on the Berkeley NOW. In: JOINT PARALLEL PROCESSING SYMPOSIUM, 9., 1997, Kobe, Japan. **Proceedings...** [S.l.: s.n.], 1997.

DE ROSE, C. A. F.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre - RS: Sagra Luzzatto, 2003.

EPEMA, D. H. J.; LIVNY, M.; DANTZIG, R. van; EVERS, X.; PRUYNE, J. A Worldwide Flock of Condors: load sharing among workstation clients. **Journal on Future Generations of Computer Systems**, [S.l.], v.12, 1996.

ERWIN, D. W.; SNELLING, D. F. UNICORE: A Grid computing environment. **Lecture Notes in Computer Science**, [S.l.], v.2150, 2001.

FALKNER, K. E. K.; CODDINGTON, P. D.; OUDSHOORN, M. J. Implementing Asynchronous Remote Method Invocation in Java. In: PARALLEL AND REAL-TIME SYSTEMS (PART'99), 1999, Melbourne, Australia. **Anais...** [S.l.: s.n.], 1999. p.22–34.

FERRARI, A. J. JPVM: network parallel computing in Java. In: ACM 1998 WORKSHOP ON JAVA FOR HIGH-PERFORMANCE NETWORK COMPUTING, 1998, New York, NY, USA. **Anais...** ACM Press, 1998.

FOSTER, I. The Grid: A New Infrastructure for the 21st Century Science. **Physics Today**, [S.l.], February 2002.

FOSTER, I.; KESSELMAN, C. Globus: a metacomputing infrastructure toolkit. **11 International Journal of Supercomputer Applications and High Performance Computing**, [S.l.], p.115–128, 1997.

FOSTER, I.; KESSELMAN, C. Globus: a toolkit-based grid architecture. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid**: blueprint for a future computing infrastructure. San Francisco, CA, USA: MORGAN-KAUFMANN, 1998. p.259–278.

FOSTER, I.; KESSELMAN, C. **The Grid**: blueprint for a new computing infrastructure. 2.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2003. 800 (est.)p.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus Approach to Integrating Multithreading and Communication. **Journal of Parallel and Distributed Computing**, [S.l.], v.37, n.1, p.70–82, 1996.

FREY, J.; TANNENBAUM, T.; LIVNY, M.; FOSTER, I.; TUECKE, S. Condor-G: a computation management agent for multi-institutional grids. In: High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium, 2001, San Francisco, CA, USA. **Anais...** IEEE Computer Society Press, 2001. p.55–63.

FUAD, M. M.; OUDSHOORN, M. J. AdJava - Automatic Distribution of Java Applications. In: COMPUTER SCIENCE 2002, TWENTY-FIFTH AUSTRALASIAN COMPUTER SCIENCE CONFERENCE (ACSC2002), 2002, Monash University, Melbourne, Victoria, Australia. **Anais...** [S.l.: s.n.], 2002. p.65–75.

GEHRING, J.; REINEFELD, A. Mars - a framework for minimizing the job execution time in a metacomputing environment. In: FUTURE GENERAL COMPUTER SYSTEMS, 1996. **Proceedings...** [S.l.: s.n.], 1996.

GETOV, V.; LASZEWSKI, G. von; PHILIPPSEN, M.; FOSTER, I. Multiparadigm communications in Java for grid computing. **Communications of the ACM**, [S.l.], v.44, n.10, p.118–125, Oct. 2001.

GOLDING, R.; BOSCH, P.; STAELIN, C.; SULLIVAN, T.; WILKES, J. Idleness is Not Sloth. In: USENIX TECHNICAL CONFERENCE ON UNIX AND ADVANCED COMPUTING SYSTEMS, 1995, Berkeley, CA, USA. **Proceedings...** USENIX Association, 1995. p.201–212.

GOSLING, J.; MCGILTON, H. **Sun Microsystems. Java Technology**. Disponibilizado em http://java.sun.com. Atualizado em março de 2004.

GRIMSHAW, A.; FERRARI, A.; KNABE, F.; HUMPHREY, M. Wide Area Computing: resource sharing on a large scale. **IEEE Computer**, Seattle, Washington, USA, v.35, n.2, p.29–37, May 1999.

GRIMSHAW, A. S.; NATRAJAN, A.; HUMPHREY, M. A.; LEWIS, M. J.; NGUYEN-TUONG, A.; KARPOVICH, J. F.; MORGAN, M. M.; FERRARI, A. J. From Legion to Avaki: the persistence of vision. In: BERMAN, F.; FOX, G.; HEY, A. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. p.265–298.

GRIMSHAW, A. S.; WULF, W. A.; TEAM, C. T. L. The Legion vision of a worldwide virtual computer. **Communications of the ACM**, New York, NY, USA, v.40, n.1, p.39–45, 1997.

GROPP, W.; LUSK, E. Dynamic Process Management in an MPI Setting. In: SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING (SPDP '95), 1995, Los Alamitos, Ca., USA. **Anais...** IEEE Computer Society Press, 1995. p.530–533.

GROPP, W.; LUSK, E. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. **The International Journal of Supercomputer Applications and High Performance Computing**, [S.l.], v.11, n.2, p.103–114, Summer 1997.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, [S.l.], v.22, n.6, p.789–828, Sept. 1996.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: Portable Parallel Programming with the Message Passing Interface. Cambridge, Massachusetts, USA: MIT Press, 1994.

HAUMACHER, B.; MOSCHNY, T.; PHILIPPSEN, M. **JavaParty, a distributed companion to Java**. Disponibilizado em http://wwwipd.ira.uka.de/JavaParty/.

HSIEH, C.-H. A.; GYLLENHAAL, J. C.; W. HWU, W. mei. Java bytecode to native code translation: the caffeine prototype and preliminary results. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 29., 1996. **Proceedings...** IEEE Computer Society, 1996. p.90–99.

JACKSON, K. **DoE Department of Energy Science Grid**. http://www.doesciencegrid.org- último acesso em novembro 2004.

JOHNSTON, W. E. **NASA Information Power Grid**. http://www.ipg.nasa.gov- último acesso em novembro 2004.

JU, J.; XU, G.; TAO, J. Parallel Computing Using Idle Workstations. **Operating Systems Review**, [S.l.], v.27, n.3, p.87–96, July 1993.

JUDD, G.; CLEMENT, M.; SNELL, Q. DOGMA: Distributed Object Group Management Architecture. In: ACM 1998 WORKSHOP ON JAVA FOR HIGH-PERFORMANCE NETWORK COMPUTING, 1998, New York, NY, USA. **Anais...** ACM Press, 1998. p.??–??

KAZI, I. H.; CHEN, H. H.; STANLEY, B.; LILJA, D. J. Techniques for obtaining high performance in Java programs. **ACM Comput. Surv.**, [S.l.], v.32, n.3, p.213–240, 2000.

KIELMANN, T.; HATCHER, P.; BOUGÉ, L.; BAL, H. E. Enabling Java for high-performance computing. **Communications of the ACM**, [S.l.], v.44, n.10, p.110–117, Oct. 2001.

KREUTZ, D. L.; CERA, M. C.; PASIN, M.; ROSA RIGHI, R. da. Comparativo entre Diferentes Interfaces de Comunicação para Programação Paralela. In: IV WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (WSCAD 2003), 2003, São Paulo - SP. **Anais...** [S.l.: s.n.], 2003.

KREUTZ, D. L.; CERA, M. C.; STEIN, B. O. Alguns Aspectos de Desempenho e Utilização de Aglomerados de Computadores Heterogêneos. In: IV CONGRESSO BRASILEIRO DE COMPUTAçãO (CBCOMP), 2004, Itajaí - SC. **Anais...** [S.l.: s.n.], 2004.

KRUEGER, P.; CHAWLA, R. The Stealth Distributed Scheduler. In: Proceedings of the 11th International Conference on Distributed Computing Systems (11th ICDCS'91), 1991, Arlington, Texas. **Anais...** IEEE Computer Society, 1991. p.336–343.

LITZKOW, M. J.; LIVNY, M.; MUTKA, M. W. Condor : a hunter of idle workstations. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (ICDCS '88), 8., 1988, Washington, D.C., USA. **Anais...** IEEE Computer Society Press, 1988. p.104–111.

LIVNY, M.; BASNEY, J.; RAMAN, R.; TANNENBAUM, T. Mechanisms for High Throughput Computing. **SPEEDUP Journal**, [S.l.], v.11, n.1, June 1997.

LO, V.; ZAPPALA, D.; ZHOU, D.; LIU, Y.; ZHAO, S. Cluster Computing on the Fly: p2p scheduling of idle cycles in the internet. In: THIRD INTERNATIONAL WORKSHOP ON PEERTOPEER SYSTMS, 2004. **Anais...** [S.l.: s.n.], 2004.

MAASSEN, J.; NIEUWPOORT, R. V.; VELDEMA, R.; BAL, H.; KIELMANN, T.; JACOBS, C.; HOFMAN, R. Efficient Java RMI for parallel programming. **ACM Transactions on Programming Languages and Systems**, [S.l.], v.23, n.6, p.747–775, Nov. 2001.

MPI Forum. **Massage Passing Interface Forum**. http://www.mpi-forum.org - último acesso em janeiro 2005.

MUTKA, M.; LIVNY, M. Scheduling Remote Processing Capacity In A Workstation-Processing Bank Computing System. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 7., 1987, Berlin, Germany. **Anais...** [S.l.: s.n.], 1987. p.2–9.

NICHOLS, D. A. Using Idle Workstations in a Shared Computing Environment. **11th ACM Symp. Operating Systems Principles (11th SOSP'87), Operating Systems Review**, Austin, Texas, v.21, n.5, p.5–12, Nov. 1987. InProceedings wilkes%cello@hplabs.hp.com.

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Satin: efficient parallel divide-and-conquer in java. In: Euro-Par 2000 Parallel Processing, 2000, Munich, Germany. **Anais...** Springer, 2000. n.1900, p.690–699. (Lecture Notes in Computer Science).

ORSOLETTA, R. A. D.; AREZI, F.; REBONATTO, M. T.; BRUSSO, M. J. Máquinas NOW heterogêneas. In: ESCOLA REGIONAL DE ALTO DESEMPENHO - ERAD2003, 3., 2003, Santa Maria, RS. **Anais. . .** [S.l.: s.n.], 2003. p.241–244.

PARANHOS, D.; CIRNE, W.; BRASILEIRO, F. Trading Cycles for Information: using replication to schedule bag-of-tasks applications on computational grids. In: EURO-PAR 2003: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING, 2003, Klagenfurt, Austria. **Anais. . .** [S.l.: s.n.], 2003.

PFISTER, G. F. **In search of clusters**. [S.l.]: Prentice Hall, 1998.

PHILIPPSEN, M.; HAUMACHER, B.; NESTER, C. More efficient serialization and RMI for Java. **Concurrency: Practice and Experience**, [S.l.], v.12, p.495–518, 2000.

PHILIPPSEN, M.; ZENGER, M. JavaParty – transparent remote objects in Java. **Concurrency: Practice and Experience**, [S.l.], v.9, n.11, p.1225–1242, Nov. 1997. Special Issue: Java for computational science and engineering – simulation and modeling II.

POLLATOS, S.; CANDLIN, R. Parallelism on a Network of Workstations. In: TDP96: TELECOMMUNICATION - DISTRIBUTION - PARALLELISM, 1996, Agelonde, La Londe Les Maures, France. **Proceedings. . .** [S.l.: s.n.], 1996. p.439–454. Parallelisme sur un Reseau de Stations de Travail, University of Edinburgh,U.K.

RADHAKRISHNAN, R.; VIJAYKRISHNAN, N.; JOHN, L.; SIVASUBRAMANIAM, A.; RUBIO, J.; SABARINATHAN, J. Java Runtime Systems: characterization and architectural implications. **IEEE Transactions on Computers**, [S.l.], v.50, n.2, p.131–146, Feb. 2001.

RAJE, R. R.; WILLIAMS, J. I.; BOYLES, M. Asynchronous Remote Method Invocation (ARMI) mechanism for Java. **Concurrency: Practice and Experience**, [S.l.], v.9, n.11, p.1207–1211, Nov. 1997.

ROSA RIGHI, R. da; NAVAUX, P. O. A.; PASIN, M. Sistema Aldeia: invocação remota e assíncrona de métodos sobre infiniband e deck. In: V WORKSHOP DE COMPUTAçãO DE ALTO DESEMPENHO, WSCAD 2004, 2004, Foz do Iguaçu, PR, Brasil. **Anais. . .** SBC, 2004. p.184–191.

SCHAEFFER FILHO, A. E.; SILVA, L. C. da; YAMIN, A. C.; AUGUSTIN, I.; GEYER, C. F. R. PerDiS: um modelo para descoberta de recursos na arquitetura ISAM. In: VI WORKSHOP DE COMUNICAçãO SEM FIO E COMPUTAçãO MóVEL, 2004, Fortaleza, Brasil. **Anais. . .** [S.l.: s.n.], 2004. p.98–107.

SEYMOUR, K.; NAKADA, H.; MATSUOKA, S.; DONGARRA, J.; LEE, C.; CASANOVA, H. GridRPC: a remote procedure call api for grid computing. In: TECHNICAL REPORT, UNIV. OF TENNESSE, 2002, ICL-UT-02-06. **Anais. . .** [S.l.: s.n.], 2002.

SILVEIRA, A.; ÁVILA, R.; BARRETO, M.; NAVAUX, P. O. A. DPC++: object-oriented programming applied to cluster computing. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 2000, Las Vegas, EUA. **Anais. . .** CSREA Press, 2000. p.2515–2521.

SMALLEN, S.; CASANOVA, H.; BERMAN, F. Applying scheduling and tuning to on-line parallel tomography. **Scientifi c Programming**, [S.l.], v.10, n.4, p.271–289, 2002.

SMALLEN, S.; CIRNE, W.; FREY, J.; BERMAN, F.; WOLSKI, R.; SU, M.-H.; KESSELMAN, C.; YOUNG, S.; ELLISMAN, M. Combining Workstations and Supercomputers to Support Grid Applications: the parallel tomography experience. In: HETEROGENOUS COMPUTING WORKSHOP, 9., 2000. **Proceedings. . .** [S.l.: s.n.], 2000.

SMARR, L. **Entropia Inc.** http://www.entropia.com - último acesso em novembro 2004.

STILES, J.; BARTOL, T.; SALPETER, M.; SALPETER, E.; SEJNOWSKI, T. Synaptic variability: New insights from reconstructions and Monte Carlo simulations with MCell. In: COWAN, W.; SUDHOF, T.; STEVENS, C. (Ed.). **Synapses**. [S.l.]: Johns Hopkins Press, 2001. p.581–731.

SUNDERAM, V. S. PVM: A framework for parallel distributed computing. **Concurrency: practice and experience**, [S.l.], v.2, n.4, p.315–339, Dec. 1990.

TANAKA, Y.; NAKADA, H.; SEKIGUCHI, S.; SUZUMURA, S.; MATSUOKA, S. Ninf-G: A reference implementation of RPC-based programming middleware for Grid computing. **Journal of Grid Computing**, [S.l.], v.1, n.1, p.41–51, 2003.

TANENBAUM, A. S.; STEEN, M. van. **Distributed Systems**: principles and paradigms. Upper Saddle River, NJ: Prentice Hall, 2002. 803p.

THAIN, D.; LIVNY, M. Error Scope on a Computational Grid: theory and practice. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING (HPDC-11 2002), 11., 2002, Edinburgh, Scotland, UK. **Anais. . .** [S.l.: s.n.], 2002. p.199–208.

THAIN, D.; TANNENBAUM, T.; LINVY, M. Condor and the Grid. In: BERMAN, F.; FOX, G.; HEY, A. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. p.299–336.

TROMEY, T. Gcj: the new ABI and its implications. In: GCC Developers Summit ,June 2nd-4th, 2004, Ottawa, Ontario, Canada, 2004. **Proceedings. . .** [S.l.: s.n.], 2004. p.169–174.

WELCH, V.; SIEBENLIST, F.; FOSTER, I.; BRESNAHAN, J.; CZAJKOWSKI, K.; GAWOR, J.; KESSELMAN, C.; MEDER, S.; PEARLMAN, L.; TUECKE, S. Security for Grid Services. In: TWELFTH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING (HPDC-12), 2003, Seattle, Washington, USA. **Anais. . .** IEEE Computer Society Press, 2003.

WILKINSON, B.; ALLEN, M. **Parallel Programming**: Techniques and applications using networked workstations and parallel computers. Upper Saddle River, New Jersey: Prentice-Hall, 1999.

WOLSKI, R.; BREVIK, J.; S.PLANK, J.; BRYAN, T. Grid resource Allocation and Control Using Computational Economies. In: BERMAN, F.; FOX, G.; HEY, A. (Ed.). **Grid Computing**: making the global infrastructure a reality. New York, NY, USA: Wiley, 2003. p.747–771.

APÊNDICE A CLASSE ALLOCATOR

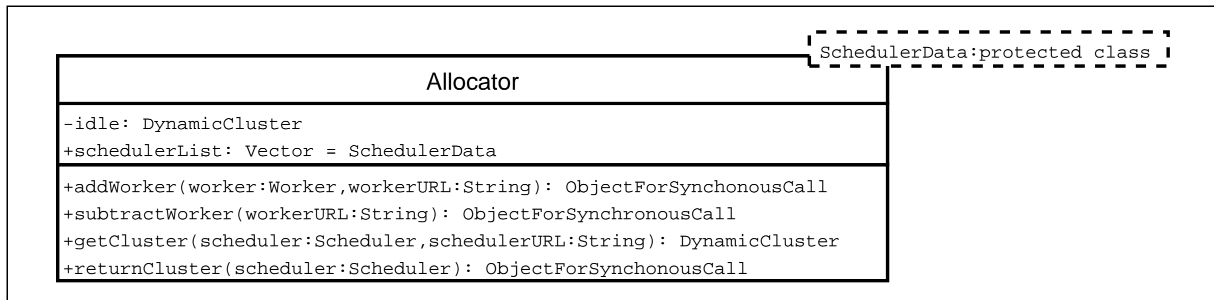

Figura A.1: UML da classe Allocator

A classe Allocator é responsável pelo gerenciamento de todos os computadores disponíveis ao sistema Cadeo. Para isso essa classe mantém armazenadas as referências de todos os computadores disponíveis do sistema em um aglomerado dinâmico próprio. Esse aglomerado dinâmico é representado pelo atributo *idle* que é um objeto da classe DynamicCluster. Além disso, também é responsabilidade dessa classe alocar e gerenciar aglomerados dinâmicos às aplicações paralelas que utilizam o sistema. Para isso, a classe mantém armazenada uma lista das referências dos escalonadores associados às aplicações paralelas e responsáveis por gerenciar os aglomerados dinâmicos.

Atributos:

DynamicCluster idle - aglomerado dinâmico onde ficam armazenados os computadores ociosos

Vector schedulerList - vetor que armazena as referências aos escalonadores que possuem um aglomerado dinâmico associado a ele

Construtor:

Allocator( )

Construtor Vazio

Métodos:

addWorker

public ObjectForSynchronousCall **addWorker**(Worker worker, String workerURL)

Adiciona um computador ocioso ao Cadeo. Se existir um aglomerado dinâmico carente por recursos, o computador ocioso será repassado a ele. Em caso contrário, o computador ocioso será incluído no aglomerado dinâmico associado a essa classe até que seja requerido para compor o aglomerado dinâmico de uma aplicação.

Parâmetros:

worker - referência ao objeto da classe Worker presente no computador ocioso
workerURL - endereço URL da JVM do computador ocioso

Retorno:

objeto específico para sincronização

subtractWorker

public ObjectForSynchronousCall **subtractWorker**(Worker worker)

Remove um computador ocioso do sistema. Verifica se o computador ocioso integra o aglomerado dinâmico associado a essa classe, em caso afirmativo remove-o do aglomerado. Caso contrário, descobre, através da lista de escalonadores, a qual aglomerado dinâmico o computador foi alocado e repassa o pedido de exclusão do computador.

Parâmetros:

worker - referência ao objeto da classe Worker presente no computador ocioso

Retorno:

objeto específico para sincronização

getCluster

public DynamicCluster **getCluster**(Scheduler scheduler)

Aloca um aglomerado dinâmico a uma aplicação paralela. Além disso, esse método armazena a referência do escalonador associado a aplicação paralela a fim de realizar ajustes na composição do aglomerado dinâmico.

Parâmetros:

scheduler - referência ao objeto da classe Scheduler responsável pelo aglomerado dinâmico

Retorno:

objeto da classe DynamicCluster representando o aglomerado dinâmico

returnCluster

public ObjectForSynchronousCall **returnCluster**(Scheduler scheduler)

Elimina a referência a um aglomerado dinâmico alocado a uma aplicação paralela.

Parâmetros:

scheduler - referência ao objeto da classe Scheduler responsável pelo aglomerado dinâmico

Retorno:

objeto específico para sincronização

APÊNDICE B CLASSE DYNAMICCLUSTER

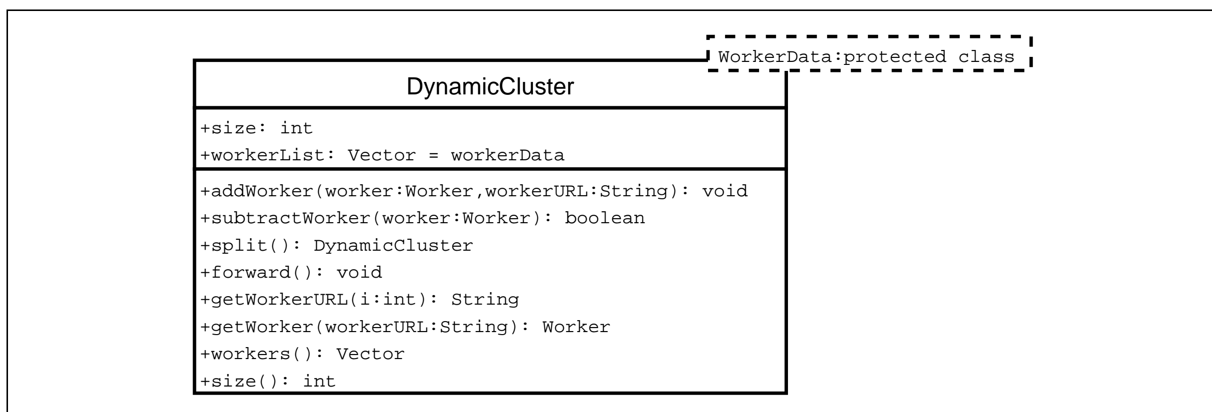

Figura B.1: UML da classe DynamicCluster

A classe DynamicCluster representa os aglomerados dinâmicos no Cadeo. Ela armazena as informações dos computadores que pertencem a um determinado aglomerado dinâmico. Além disso, ela oferece todos os métodos necessários para o gerenciamento do aglomerado dinâmico.

Atributos:

int size - número total de computadores do aglomerado dinâmico

Vector workerList - armazena as informações de todos os computadores disponíveis a um aglomerado dinâmico

Construtor:

DynamicCluster( )

Cria um objeto da classe DynamicCluster

Métodos:

addWorker

public void **addWorker**(Worker worker, String workerURL)

Adiciona um computador ocioso ao aglomerado dinâmico.

Parâmetros:

worker - referência ao objeto da classe Worker presente no computador ocioso

workerURL - endereço URL da JVM do computador ocioso

subtractWorker

public boolean **subtractWorker**(Worker worker)

Remove um computador de um aglomerado dinâmico.

Parâmetros:

worker - referência ao objeto da classe Worker presente no computador

Retorno:

true caso tenha sido possível remover o computador e *false* em caso contrário

split

public DynamicCluster **split( )**

Atribui parte dos computadores disponíveis no aglomerado dinâmico para outro aglomerado dinâmico. O número de computadores atribuídos varia conforme uma política de distribuição empregada. Na versão corrente, na invocação desse método todos os computadores do aglomerado são atribuídos ao novo aglomerado.

Retorno:

o novo aglomerado dinâmico alocado

forward

public void **forward( )**

Repassa todos os computadores de um aglomerado dinâmico ao alocador.

getWorkerURL

public String **getWorkerURL**(int i)

Encontra um computador do aglomerado dinâmico a partir de seu índice.

Parâmetros:

i - índice do computador no aglomerado dinâmico

Retorno:

endereço URL da JVM do computador procurado

getWorker

public Worker **getWorker**(String workerURL)

Encontra um computador do aglomerado dinâmico a partir de seu endereço URL.

Parâmetros:

workerURL - endereço URL da JVM do computador procurado do aglomerado dinâmico

Retorno:

a referencia ao objeto da classe Worker presente no computador procurado

workers

public Vector **workers**()

Retorna os dados de todos computadores pertencentes ao aglomerado dinâmico.

Retorno:

vetor contendo os dados de todos os computadores associados a um aglomerado dinâmico

size

public int **size**()

Retorna o número de computadores associados ao aglomerado dinâmico.

Retorno:

número total de computadores associados a um aglomerado dinâmico

APÊNDICE C CLASSE WORKER

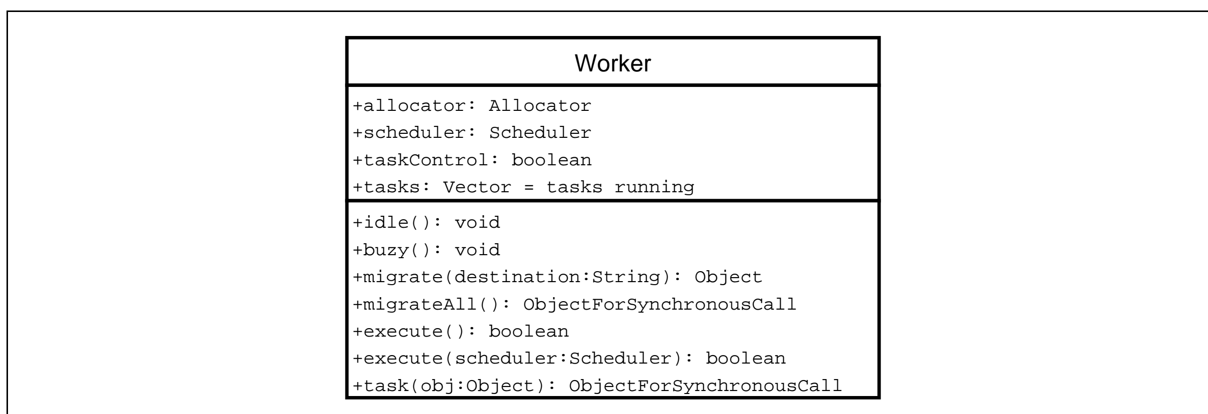

Figura C.1: UML da classe Worker

A classe Worker é responsável pela comunicação do estado de um computador (ocioso ou indisponível) ao alocador e pelo controle das tarefas em execução naquele computador.

Atributos:

Allocator allocator - referência ao alocador

Scheduler scheduler - referência ao escalonador

boolean taskControl - informa se o computador está apto a receber tarefas

Vector tasks - vetor que armazena referências as tarefas em execução

Construtor:

Worker( )

Construtor vazio

Worker(String allocatorAddress)

Cria um objeto da classe Scheduler e conecta-o ao alocador através do endereço passado como parâmetro

Métodos:

idle

public void **idle**( )

Informa ao alocador sobre a disponibilidade do computador, ou seja, que o computador é um computador ocioso.

busy

public void **busy**( )

Informa ao alocador que o computador não está mais disponível, ou seja, que o computador é um computador indisponível.

execute

public boolean **execute**(Scheduler scheduler)

Verifica se é possível executar mais uma tarefa naquele computador.

Parâmetros:

scheduler - referência ao objeto da classe Scheduler responsável pelo aglomerado dinâmico

Retorno:

true se for possível executar uma tarefa ou false caso contrário

execute

public boolean **execute**( )

Verifica se é possível executar mais uma tarefa naquele computador.

Retorno:

true se for possível executar uma tarefa ou false caso contrário

task

public ObjectForSynchronousCall **task**(Object object)

Armazena uma referência a tarefa em execução no computador. Essa referência será usada quando fizer-se necessária a migração das tarefas presentes no computador.

Parâmetros:

object - referência a tarefa em execução nesse computador

Retorno:

objeto específico para sincronização

migrateAll

public ObjectForSynchronousCall **migrateAll**( )

Desencadeia a migração de todas as tarefas em execução no computador.

Parâmetros:

object - referência a tarefa em execução nesse computador

Retorno:

objeto específico para sincronização

migrate

public void **migrate**(String destination)

Migra um objeto que representa uma tarefa para outro computador.

Parâmetros:

destination - endereço do computador de destino

APÊNDICE D   CLASSE SCHEDULER

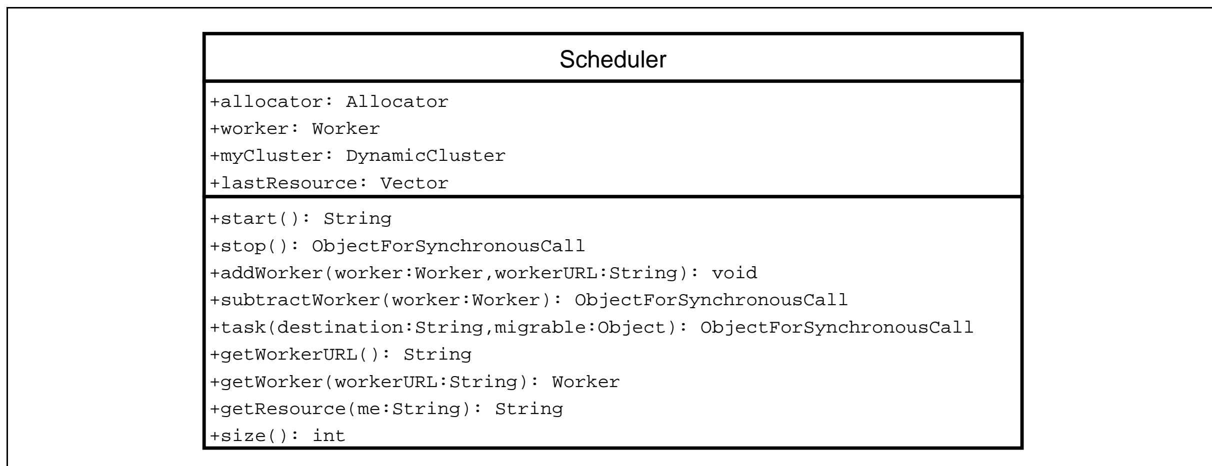

Figura D.1: UML da classe Scheduler

A classe Scheduler é responsável por solicitar um aglomerado dinâmico para a execução de uma aplicação paralela. Além disso, é responsável por distribuir as tarefas dessa aplicação nos computadores disponíveis no aglomerado dinâmico.

Atributos:

Allocator allocator - referência ao alocador

Worker worker - objeto da classe Worker associado ao escalonador

DynamicCluster myCluster - aglomerado dinâmico gerenciado pelo escalonador

Vector lastResource - vetor que armazena por ordem de prioridade os endereços dos computadores que estão aptos a receber tarefas

Construtor:

Scheduler( )

Construtor vazio

Scheduler(String allocatorAddress)

Cria um objeto da classe Scheduler e conecta-o ao alocador através do endereço passado como parâmetro

Métodos:

start

public String **start**( )

Inicia as atribuições do escalonador. Solicita ao alocador um aglomerado dinâmico e cria um objeto da classe Worker que permanecerá associado ao escalonador.

Retorno:

endereço URL da JVM do escalonador

stop

public ObjectForSynchronousCall **stop**( )

Finaliza o escalonador liberando os computadores do aglomerado dinâmico associado a ele.

Retorno:

objeto específico para sincronização

addWorker

public void **addWorker**(Worker worker, String workerURL)

Adiciona mais um computador ocioso ao aglomerado dinâmico.

Parâmetros:

worker - referência ao objeto da classe Worker presente no computador ocioso

workerURL - endereço URL da JVM do computador ocioso

subtractWorker

public ObjectForSynchronousCall **subtractWorker**(Worker worker)

Remove um computador do aglomerado dinâmico.

Parâmetros:

worker - referência ao objeto da classe Worker presente no computador a ser removido

Retorno:

objeto específico para sincronização

task

public ObjectForSynchronousCall **task**(String destination, Object migrable)

Destina uma tarefa a ser executada em um computador específico.

Parâmetros:

destination - endereço do computador de destino

migrable - referência a tarefa que será executada no computador

Retorno:

objeto específico para sincronização

getWorkerURL

public String **getWorkerURL**( )

Retorna um computador qualquer do aglomerado dinâmico.

Retorno:

endereço URL da JVM de um computador do aglomerado

getWorker

public Worker **getWorker**( )

Retorna uma referência ao objeto da classe Worker em um computador remoto.

Retorno:

referência a um objeto da classe Worker

getResource

public String **getResource**(String me)

Retorna o endereço de um computador do aglomerado dinâmico que esteja disponível a receber tarefas. Caso não exista nenhum computador para satisfazer a chamada desse método o endereço do objeto da classe Worker associado ao escalonador (trabalhador de último recurso) será entregue como último recurso.

Parâmetros:

me - endereço do computador corrente

Retorno:

endereço do computador que receberá tarefas

size

public int **size**( )

Retorna o número de computador existentes no aglomerado dinâmico.

Retorno:

número total de computadores associados ao aglomerado dinâmico

APÊNDICE E   CLASSE CADEO

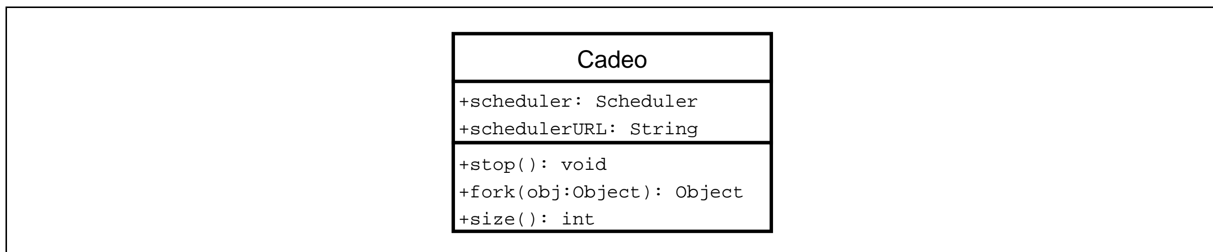

Figura E.1: UML da classe Cadeo

A classe Cadeo é responsável por oferecer uma interface entre o sistema Cadeo e a aplicação paralela que fará uso do sistema.

Atributos:

Scheduler scheduler - objeto da classe Scheduler pelo qual é realizada a interação com o sistema Cadeo

String schedulerURL - endereço URL do escalonador

Construtor:

Cadeo(String allocatorAddress)

Cria um objeto da classe Cadeo. O objeto criado possui um objeto da classe Scheduler que representa o escalonador, o qual conecta-se ao alocador através do endereço passado como parâmetro.

Cadeo(String allocatorAddress , String newSchedulerClass)

Cria um objeto da classe Cadeo. O objeto criado possui um objeto de uma classe do

tipo passado por parâmetro que representa o escalonador, o qual conecta-se ao alocador através do endereço passado como parâmetro.

Métodos:

stop

public void **stop**( )

Anuncia o final da execução desencadeando a liberação dos computadores do aglomerado dinâmico.

fork

public Object **fork**(String className, Object[] param)

Instancia uma tarefa a ser executada remotamente nos computadores presentes no aglomerado dinâmico.

Parâmetros:

className - nome da classe que representa uma tarefa

param - parâmetros necessários para a instanciação do objeto remoto

Retorno:

referência ao objeto instanciado (tarefa)

size

public int **size**()

Retorna o número de computadores associados ao aglomerado dinâmico.

Retorno:

número total de computadores associados ao aglomerado dinâmico