Uma Ferramenta Orientada a Objetos para Monitoramento de Cargas em Sistemas Distribuídos

Paulino Ribeiro Villas Boas e Gonzalo Travieso
Departamento de Física e Informática
IFSC - USP
{paulino, gonzalo}@ifsc.usp.br

Resumo

Este artigo apresenta uma ferramenta que realiza uma das tarefas mais importantes no processo de balanceamento dinâmico de cargas em sistemas distribuídos: o monitoramento. Tal ferramenta é constituída de duas partes: uma interface de programação que permite a fácil inserção de códigos em aplicações paralelas para realizar o monitoramento de cargas e um sistema de monitoramento de cargas em tempo de execução para recolher as informações de carga dos computadores do sistema distribuído. Essas informações podem, então, ser utilizadas nas aplicações paralelas ou em um sistema de balanceamento de cargas através da interface de programação para realizar balanceamento dinâmico de cargas.

1. Introdução

Sistemas paralelos de memória distribuída estão se tornando a melhor escolha para resolver problemas que exigem alto poder computacional, porque eles são escaláveis e apresentam uma boa relação custo-desempenho [4]. No entanto, a programação nesses sistemas é muito mais complexa do que a seqüencial, pois na primeira o programador tem que se preocupar com as diversas linhas de controle executando simultânea e concorrentemente e com a suas interações. Além disso, a preocupação com o desempenho é sempre grande nesses sistemas.

Um dos fatores que limitam o desempenho é o desbalanceamento de carga computacional entre os processadores. Em sistemas de memória distribuída, esse fator se torna muito importante devido ao uso de redes de interconexão "lentas" se comparadas com o poder computacional dos microprocessadores de alto desempenho, logo a granularidade utilizada deve ser grossa para atingir boa eficiência. Com granularidade grossa, as variações estatísticas das cargas tendem a serem altas também, tornando difícil otimizar a distribuição delas entre os processadores, principalmente diante das variações dinâmicas das cargas (em tempo de execução) [11]. Essa dificuldade se agrava ainda mais em sistemas distribuídos heterogêneos multi-usuários em que é impossível prever a variação das cargas antes da aplicação ser executada.

A fim de garantir um desempenho satisfatório em computadores paralelos, é necessário um bom sistema de balanceamento de cargas entre os processadores. O balanceamento de cargas implica na atribuição de carga (quantidade de tarefas ou de dados) proporcional ao desempenho de cada processador. Essa atribuição pode ser *estática*, realizada em tempo de compilação; ou *dinâmica*, realizada durante a execução [12]. Embora o balanceamento estático não gere sobrecarga durante a execução, o dinâmico é preferível para sistemas heterogêneos multi-usuários em que a carga muda em tempo de execução.

Na implementação de um algoritmo de balanceamento dinâmico de cargas, é essencial a existência de um sistema de monitoramento para obter informações como uso de CPU, de memória, de rede, etc, as quais serão utilizadas pelo sistema de balanceamento dinâmico de cargas para tomar as devidas decisões de rebalanceamento. Deste modo, em tempo de execução, é possível saber quais processadores estão ocupados e quais estão ociosos.

Este artigo apresenta uma ferramenta de monitoramento de cargas para sistemas distribuídos como NOWs (*Networks of Workstations*) e *Grids* computacionais. Esta ferramenta é constituída de uma interface e de um sistema de monitoramento de cargas em tempo de execução. A interface, composta por várias classes, permite a fácil inserção de códigos em aplicações paralelas para realizar o monitoramento de cargas. Já o sistema de monitoramento é responsável por obter as informações de cargas dos nós do sistema distribuído e por transmitir esses dados à aplicação paralela que os requisitarem.

O artigo está organizado da seguinte forma: a seção 2 revê conceitos de balanceamento de cargas; a seção 3 apresenta a ferramenta, discutindo como foi desenvolvida; a seção 4 apresenta os resultados obtidos com a utilização

desta ferramenta em uma aplicação específica; e a seção 5 resume e comenta os resultados obtidos com a utilização desta ferramenta.

2. Balanceamento de cargas

O objetivo de um algoritmo de balanceamento de cargas é maximizar a utilização dos recursos computacionais (processadores) a fim de obter alto desempenho [13]. Os esquemas de balanceamento de cargas são geralmente divididos em: *política* e *mecanismo* [9]. A política é o conjunto de escolhas realizadas para distribuir as cargas. Já o mecanismo é a parte que realiza a distribuição física das cargas e provê qualquer informação requerida pelas políticas.

A política toma decisões como: quando um nó precisa migrar parte de sua carga, qual parte deve ser essa, qual nó deve recebê-la, etc. Por essa razão, as políticas de distribuição de cargas são divididas em quatro componentes: *política de transferência*, *política de seleção*, *política de localização* e *política de informação* [9].

A política de transferência determina quando um nó se torna apropriado para participar de uma transferência de cargas, seja como remetente seja como destinatário; a política de seleção determina qual carga do remetente deve ser transferida; a política de localização se encarrega de encontrar um parceiro (destinatário) para o remetente; e a política de informação é responsável por recolher informações dos nós do sistema distribuído para serem usadas nas decisões de distribuição de cargas. O objetivo do último componente é realizar decisões mais precisas, embora ele sobrecarregue o sistema. Há várias questões relacionadas à política de informação, a saber, que informação precisa ser coletada, onde essa informação deve ser guardada e quando realizar a coleta.

2.1. Classificação

Vários foram os algoritmos ou políticas de balanceamento de cargas propostos no passado [9]. Como são muitos algoritmos diferentes, uma classificação se faz útil para melhor compreender as questões envolvidas e para comparação entre eles [14]. Várias tentativas de classificação desses algoritmos foram desenvolvidas [2, 3, 6, 16, 17], todavia aqui será apresentada apenas a proposta por Casavant e Khul [3] por ser uma das mais usadas atualmente.

Distribuição global e local No nível mais alto da hierarquia, algoritmos gerais de distribuição de cargas são divididos em locais e globais. Os globais são, em geral, referenciados na literatura como os algoritmos de distribuição de cargas entre os processadores de um sistema distribuído. Os locais são os algoritmos usuais de escalonamento de processos usados em sistemas operacionais centralizados, portanto sem importância neste artigo.

Estática e dinâmica A distribuição de cargas entre os processadores pode ser dinâmica ou estática, dependendo do tempo em que a distribuição é feita. Distribuição estática assume que a informação de carga do sistema e os requerimentos de recursos de um programa estejam disponíveis no momento de compilação do programa e, portanto, a escolha do processador que executará uma certa carga é decidida nesse instante. Distribuição dinâmica considera a informação do estado atual e prévio do sistema para tomar as decisões de distribuição de cargas e, portanto, prover melhores resultados do que a estática. Por essa razão, não serão discutidas as subclassificações da distribuição estática.

Distribuída e centralizada Aqui, a preocupação é com quem toma as decisões de distribuição de cargas. Se a decisão for feita por apenas um nó, a distribuição é centralizada, ou se for compartilhada por vários, ela é distribuída. Algoritmos centralizados são mais simples de implementar, mas não escalam bem, pois o coordenador pode se tornar o gargalo do sistema, além de ser um ponto crítico de falha, o que não pode ser tolerado em sistemas distribuídos.

Cooperativa e não-cooperativa Os algoritmos distribuídos podem ainda ser subdivididos em não-cooperativos e cooperativos. A questão aqui está relacionada com o grau de autonomia de cada nó em determinar como os seus próprios recursos serão utilizados. No caso não-cooperativo cada processador atua como entidade autônoma e toma decisões com respeito ao uso de seus recursos independente do efeito de sua decisão no resto do sistema. No caso cooperativo, cada processador deve realizar a sua própria porção da tarefa de distribuição, mas todos os processadores trabalham em conjunto para o objetivo comum do sistema.

2.2. Exemplos de algoritmos de distribuição de cargas

Esta subseção apresentará alguns algoritmos simples distribuídos e heurísticos de balanceamento de cargas [13]. Em sistemas distribuídos nos quais as cargas são imprevisíveis, a distribuição de cargas entre processadores não pode ser feita determinística ou matematicamente. Para desenvolver algoritmos de balanceamento de cargas nesse caso, é necessário usar técnicas heurísticas *ad-hoc* para redistribuir as cargas entre os nós de uma maneira sensata e transparente.

Iniciados pelo remetente No caso de algoritmos iniciados pelo remetente, o processo de distribuição de cargas é iniciado pelo nó sobrecarregado (remetente) tentando enviar uma parte de sua carga para um nó menos carregado (destinatário).

Iniciados pelo destinatário Nos algoritmos iniciados pelo destinatário, quando um nó se torna ocioso ele tenta obter carga de um nó mais ocupado.

Simetricamente iniciado Um algoritmo simples iniciado simetricamente pode ser construído combinando as políticas de transferência e de localização usadas nos algoritmos iniciados pelo remetente e pelo destinatário [10]. Ambos os tipos de nós (os ocupados e os ociosos) iniciam a distribuição de cargas.

3. Descrição da ferramenta

Em termos de teoria de balanceamento de cargas discutida na seção anterior, a ferramenta apresentada aqui constitui um mecanismo responsável por fornecer as informações de carga dos nós do sistema distribuído à política de informação para tomar as devidas decisões de rebalanceamento de cargas em tempo de execução.

Em termos de programação, a ferramenta dispõe de uma interface, composta por várias classes, e de um sistema de monitoramento de cargas em tempo de execução. Através da interface, o programador pode incluir, em suas aplicações paralelas, códigos que realizem o monitoramento de carga dos nós do sistema distribuído. Com isso, o programador tem às mãos informações de carga de todos os nós necessárias para efetuar o balanceamento dinâmico.

A ferramenta baseia-se no modelo cliente/servidor em que, neste caso, o servidor é o sistema de monitoramento que fornece o serviço de informação de carga do nó que o hospeda, e o cliente é a parte das aplicações paralelas responsável por pedir as informações de carga dos nós do sistema distribuído.

Para o seu desenvolvimento, foram utilizados: a linguagem de programação Java [5], RMI – *Remote Method Interface* [8] e um *cluster* de computadores do tipo Beowulf [15]. Foi escolhida a linguagem de programação Java por ser portátil, orientada a objetos, pela facilidade de comunicação entre os nós do sistema distribuído através de RMI, e pela facilidade de programação devido à simplicidade da linguagem. Dentre essas características, a mais importante é ser portátil, pois é um dos objetivos desta ferramenta garantir que ela seja portátil para qualquer tipo de plataforma, já que sistemas distribuídos como NOWs e *Grids* computacionais podem ser formados por nós com arquiteturas e sistemas operacionais diferentes.

O sistema distribuído utilizado para o desenvolvimento e teste desta ferramenta foi um *cluster* de computadores do tipo Beowulf com 16 nós e com Linux Slackware versão 8.1 instalado em cada nó. Embora o sistema de monitoramento de cargas tenha sido projetado para sistemas distribuídos heterogêneos, o uso deste *cluster* permitiu analisar o impacto do sistema desenvolvido na execução de programas paralelos. Para simular um sistema distribuído heterogêneo, foi colocada carga em alguns nós, desbalanceando a carga geral do sistema.

Cada sistema operacional possui um jeito próprio de guardar as informações de carga computacional, logo para cada um existe uma maneira específica de medi-las. No caso de sistemas Unix, todas as informações do computador são guardadas em arquivos e atualizadas periodicamente; assim, para saber como os recursos do computador estão sendo utilizados, basta ler os arquivos que guardam essas informações. No Slackware, as informações de uso de CPU podem ser obtidas através do arquivo `/proc/stat`; as de uso de memória, em `/proc/meminfo`; e as de tráfego de rede, em `/proc/net/dev`.

Embora obter a carga seja dependente da plataforma, a ferramenta ainda assim é portátil porque, ao utilizar a interface, o programador não precisa saber qual é a plataforma de cada nó do sistema distribuído. Em outras palavras, a interface da ferramenta, a única parte vista pelo programador, é portátil apesar de a implementação não ser.

3.1. Interface

A interface desta ferramenta é composta de duas interfaces Java: `LoadIndex` e `Resource`. A primeira representa o *índice de carga* [13], um índice que mostra o quanto os recursos de um determinado nó estão ocupados. A segunda descreve objetos cujos métodos, destinados a medir um determinado índice de carga, podem ser chamados remotamente.

As classes que implementam essas interfaces são os tipos de medida de carga que se deseja realizar. Três tipos de medidas de carga foram desenvolvidas: uso de CPU, de memória e de rede.

3.1.1. Interface `LoadIndex` É a interface que define os índices de carga, figura 1. Cada tipo de índice de carga é uma classe que implementa esta interface. O método `get()` (linha 3) é o método de acesso a esse índice cujo valor varia entre 0.0 e 1.0, ou seja, entre totalmente ocioso e totalmente ocupado. Embora a maioria dos sistemas de distribuição de cargas utilize um único índice de carga para representar o estado de um dado nó, este sistema de monitoramento define índices de carga individuais de maneira que o programador da aplicação paralela escolha os que melhor lhe convier.

```
1  public interface LoadIndex extends
2      java.io.Serializable {
3          public double get();
4  }
```

Figura 1. Definição da interface `LoadIndex`.

Para realizar as medidas de carga propostas para esta ferramenta, foram desenvolvidas as seguintes classes: `CPULoadIndex`, `MemLoadIndex` e `NetLoadIndex`,

as quais representam os índices de carga de CPU, de memória e de rede, respectivamente. `MemLoadIndex` é um pouco diferente das demais e, além de representar o índice de carga de memória total (RAM + *Swap*), é composta de duas outras subclasses `LoadIndex`: `RamLoadIndex`, que representam o índice de carga de memória física (RAM), e `SwapLoadIndex`, o índice de carga de memória virtual (*Swap*). Além do método `get()`, `MemLoadIndex` possui dois outros métodos: um de acesso a objetos da classe `RamLoadIndex` e outro de acesso a objetos da classe `SwapLoadIndex`.

3.1.2. Interface `Resource` Essa interface, apresentada na figura 2, define que os objetos das classes que a implementam possuam um método para cálculo do índice de carga para um determinado tipo de medida (uso de CPU, de memória ou de rede).

```
1  public interface Resource extends
2     java.rmi.Remote {
3        public LoadIndex info()
4           throws java.rmi.RemoteException;
5  }
```

Figura 2. Definição da interface `Resource`.

O cliente (parte da aplicação paralela interessada no monitoramento de cargas) para um determinado tipo de medida de carga só precisa da definição dessa interface para poder realizar a chamada do método `info()` e receber o índice de carga correspondente ao pedido efetuado.

O servidor para um certo tipo de medida, por outro lado, precisa da implementação dessa interface em uma classe, de maneira a poder prover o serviço correspondente. Três classes que implementam a interface `Resource` são apresentadas a seguir: `CPUResourceImpl`, `MemResourceImpl` e `NetResourceImpl`.

CPUResourceImpl É a classe destinada a calcular o índice de carga de CPU. A medida de carga de CPU é realizada através da leitura do arquivo `/proc/stat`. A figura 3 mostra um trecho desse arquivo (que é a única parte importante para os propósitos desta ferramenta). Após a palavra `cpu` da primeira linha, os números seguintes são: *user*, *nice* (modo *user* com baixa prioridade), *system* e *other* [1]. Esses números representam a quantidade de *jiffies* (1 *jiffy* é igual a 10 milisegundos [1]) gasta em cada estado desde que o computador foi ligado, logo esses números sempre aumentam. Cada um tem um significado diferente: *user* representa o tanto de *jiffies* gasto pelo usuário; *nice*, o que foi gasto no modo *user* com baixa prioridade; *system*, o que foi gasto pelo sistema operacional; e *other*, o que não foi gasto com nenhum dos anteriores (tempo em que a CPU ficou ociosa).

```
1  cpu  704475 0 31068 1488631
2  cpu0 704475 0 31068 1488631
3  page 62945 55186
4  swap 1 0
```

Figura 3. Trecho do arquivo `/proc/stat`.

Como o sistema operacional atualiza esse arquivo de 10 em 10 milisegundos, é suficiente fazer duas leituras consecutivas desse arquivo com intervalo maior do que 10 milisegundos para obter a quantidade de *jiffies* gasta em cada estado do sistema num certo instante de tempo e calcular o índice de carga de CPU. Conforme [1], esse cálculo pode ser feito da seguinte forma:

$$CPULoadIndex = 1.0 - \frac{O}{U + N + S + O} \quad (1)$$

onde: $CPULoadIndex$ é o índice de carga de CPU; O é a diferença de *jiffies* do estado *other* medida nesse intervalo de tempo; U, N e S o mesmo para *user*, *nice* e *system*, respectivamente. Após calcular esse índice, constrói-se um objeto da classe `CPULoadIndex` com esse valor e o método `info()` o retorna ao cliente que o requisitou.

MemResourceImpl É a classe que implementa a medida de carga de memória. Os objetos dessa classe calculam três índices de carga: um de memória RAM, `RamLoadIndex`, um de memória *Swap*, `SwapLoadIndex` e um da memória total (RAM + *Swap*), `MemLoadIndex`. Para cálculo desses índices, faz-se uma leitura do arquivo `/proc/meminfo`. Na figura 4, é mostrado um trecho desse arquivo, onde `Mem` é a memória RAM.

```
1          total:      used:       free:      ...
2  Mem:   262299648  116932608  145367040   ...
3  Swap:  1085693952         0  1085693952  ...
```

Figura 4. Trecho do arquivo `/proc/meminfo`.

Aqui, o cálculo do índice de carga de memória é mais fácil. Nos três casos, o índice é obtido dividindo a quantidade de memória usada pelo total. No caso da memória total (RAM + Swap), o índice é obtido somando a quantidade de memória RAM e a de Swap usadas e dividindo o resultado pela soma do total das duas. Após o cálculo desses índices, constrói-se um objeto da classe `MemLoadIndex`, e o método `info()` o retorna para o cliente que solicitou o serviço de medida de carga de memória.

NetResourceImpl Essa classe implementa a medida de carga de rede que é realizada através da leitura do arquivo `/proc/net/dev`. Um trecho desse arquivo pode ser visto na figura 5. Os "..." representam a parte desse arquivo que não é importante para a medida de carga de rede.

Os dados importante são: a quantidade de *bytes* recebidos e a quantidade de *bytes* transmitidos por uma dada interface de rede que, neste caso, pode ser: eth0, eth1 e eth2.

```
1 Inter-|   Receive              |  Transmit
2  face|  bytes   packets ... | bytes    packets ...
3    lo: 1852196  18675   ...  1852196   18675   ...
4  eth0:15306096 110036   ...  102817704 160434  ...
5  eth1: 3022820  20586   ...  1285216   16793   ...
6  eth2: 3198330  36850   ...  2513430   14839   ...
```

Figura 5. Trecho do arquivo /proc/net/dev.

O cálculo do índice de carga de rede segue o mesmo raciocínio do de CPU: lê-se o arquivo /proc/net/dev duas vezes e se determinam quantos *bytes* foram recebidos e quantos foram transmitidos numa certa interface de rede e num certo intervalo de tempo. Somando essas quantidades, dividindo o resultado pelo intervalo de tempo e, depois, pela largura de banda máxima, obtém-se o índice de carga de rede. Em seguida, com esse índice, constrói-se um objeto da classe NetLoadIndex que é retornado pelo método info().

3.2. Sistema de monitoramento de cargas

O sistema de monitoramento de cargas nada mais é do que servidores RMI os quais fornecem os índices de cargas dos nós do sistema distribuído. Portanto, os servidores e os clientes constituem a base para o funcionamento desta ferramenta. A figura 6 mostra como esses programas funcionam. O cliente é a parte das aplicações paralelas que pedem as informações de cargas dos nós do sistema distribuído; e o servidor é a aplicação executada em todos os nós para medir um certo tipo de carga e atender às requisições dos clientes. A medida de um determinado tipo de carga é realizada pelo objeto de uma classe que implementa a interface Resource, e a única tarefa do servidor consiste em criar esse objeto permitindo-o ser acessado pelos clientes.

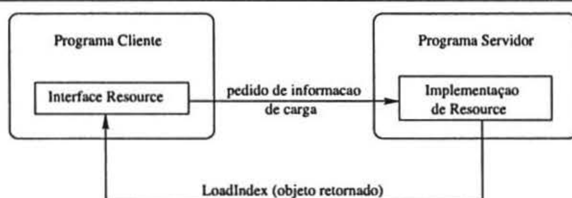

Figura 6. Esquema mostrando como funciona o sistema de monitoramento de cargas.

Para exemplificar como funciona o sistema de monitoramento de cargas, apresentam-se a seguir o servidor e o cliente responsáveis pela medida de carga de CPU. O trecho mais importante do código do servidor é mostrado na figura 7.

```
1  ...
2  Resource cpur = new CPUResourceImpl();
3  Naming.rebind("rmi://localhost:1099/
4     CPUResourceService", cpur);
5  ...
```

Figura 7. Trecho do código do servidor de carga de CPU.

A linha 2 desse código informa que cpur é um objeto da interface Resource cuja implementação é dada por CPUResourceImpl. As linhas 3 e 4 indicam que o objeto cpur está associado ao endereço rmi://localhost:1099/CPUResourceService, onde localhost é o computador que hospeda o servidor e 1099 (porta padrão usada pelo RMI) é a porta onde o serviço CPUResourceService fica aguardando conexões. Por este código se vê que o servidor somente cria um objeto acessível remotamente e fica esperando por pedidos de clientes. Toda a parte de medida de um determinado tipo de carga é deixada para a implementação desse objeto, que, no caso de carga de CPU, é um objeto da classe CPUResourceImpl.

O trecho mais importante do código do cliente é exibido na figura 8. As linhas 2 e 3 indicam que o objeto cpur não é local mas sim remoto e que deve ser procurado no endereço rmi://n1/CPUResourceService, onde n1 é o endereço do nó em que está o objeto remoto, ou seja, é o endereço do computador do qual se quer saber o seu índice de carga de CPU. Após encontrar o servidor, o cliente chama o método info() para obter o índice de carga do nó desejado, conforme indica a linha 4. Também nessa linha, nota-se que é necessário uma conversão do objeto de retorno do método info(): LoadIndex para o tipo desejado: CPULoadIndex que é guardado pelo objeto cpul.

```
1  ...
2  Resource cpur = (Resource)Naming.lookup("rmi://
3     n1/CPUResourceService");
4  CPULoadIndex cpul = (CPULoadIndex)cpur.info();
5  ...
```

Figura 8. Trecho do código do cliente de carga de CPU.

A fim de que o sistema de monitoramento funcione ainda falta registrar os servidores nos nós do sistema distribuído para os clientes saberem como encontrá-los. Para isso é necessário iniciar o RMI executando o comando: `rmiregistry port`, onde `port` é o número da porta em que o servidor aguarda conexões (se não for especificada nenhuma porta, o RMI utiliza a porta padrão `1099`). Com o `rmiregistry` rodando, coloca-se os servidores para serem executados. Diz-se, então, que o servidor foi registrado no RMI e que já está disponível para receber conexões. Quando o cliente for executado, ele procurará pelo computador onde se encontra o seu objeto remoto, "perguntará" ao RMI qual é o servidor que fornece o serviço que ele está pedindo e, se encontrado, conectará diretamente ao servidor e, depois, fará a chamada do método remoto `info()` para saber a carga do servidor.

4. Testes e resultados

Para testar a ferramenta desenvolvida, duas aplicações foram implementadas: uma realiza balanceamento estático de cargas e a outra, balanceamento dinâmico. A segunda, além de dinâmica, também é distribuída, cooperativa e iniciada pelo destinatário e utiliza a ferramenta de monitoramento apresentada neste artigo.

O problema abordado no desenvolvimento dessas aplicações foi um dos clássicos do xadrez: encontrar o número de soluções do problema do *passeio do cavalo*, ou seja, encontrar todas as solução de o cavalo percorrer um tabuleiro de $M \times N$ casas e visitar cada casa apenas uma vez, partindo de todas as casas do tabuleiro.

Como o cálculo do número de soluções para cada casa inicial (casa onde o cavalo inicia o seu percurso) independe do cálculo das demais, esse critério foi utilizado para paralelizar esse problema, ou seja, cada computador recebe uma quantidade de casas iniciais para calcular o número de soluções referentes àquelas casas.

Na aplicação que realiza balanceamento estático, denominada de `CavParEst`, efetua-se apenas uma divisão inicial das casas do tabuleiro entre os processadores do sistema distribuído. Nessa divisão, cada nó recebe a mesma quantidade de casas, exceto quando a divisão não for inteira, caso em que alguns nós ficam com uma casa a mais que os outros.

Já na outra aplicação, designada de `CavParDin`, também a mesma divisão inicial de casas é realizada, todavia, há redistribuição de casas entre os processadores durante a execução. O processo de redistribuição de casas é iniciado pelo destinatário – quando um nó se torna ocioso, ele "pede" parte das casas restantes de algum dos nós ocupados. A aplicação `CavParDin` determina que seus nós são ociosos ou ocupados conforme seus índices de uso de CPU: acima de 75%, o nó é considerado ocupado, e abaixo desse valor, ocioso. Quando acabar as casas de um nó, ele se torna ocioso e inicia o processo de redistribuição de cargas procurando pelos nós ocupados através do sistema de monitoramento de cargas segundo o índice de uso de CPU. Na figura 9, é mostrado o trecho da aplicação `CavParDin` responsável por procurar os nós ocupados.

```
1   ...
2   for (int i=0; i<nos.length; i++) {
3     cputhread[i] = new CPUResourceThread(nos[i]);
4     cputhread[i].start();
5   }
6   Sring[] nosOcupados = new String[nos.length];
7   int m = 0;
8   CPULoadIndex cpuload;
9   for (int i=0; i<nos.length; i++) {
10    cputhread[i].join();
11    cpuload = cputhread[i].get();
12    if (cpuload.get() > Limiar) {
13      nosOcupados[m] = nos[i];
14      m += 1;
15    }
16  }
17  ...
```

Figura 9. Trecho do código de `CavParDin` que mostra o uso da informação de CPU na procura por nós ocupados.

O `for` das linhas 2 a 5 inicia *threads* para pedir os índices de carga de CPU de todos os nós do *cluster*, onde `nos` é um *array* de *strings* contendo o endereço de todos os nós. Essas *threads* nada mais são do que os clientes descritos na subseção 3.2, e cada uma é responsável por obter o índice de carga de CPU de um nó do *cluster*.

O `for` da linha 9 à linha 16 espera pelo término de cada *thread* (linha 10) e retira o índice de carga obtido através do método de acesso `get()` (linha 11). Se esse índice for maior que `Limiar` (0.75), o nó em questão é marcado como ocupado. A partir do *array* de *strings* contendo os nós ocupados, procura-se o primeiro que possua casas a compartilhar. Se encontrado, ocorre a redistribuição de casas e o nó ocioso volta a trabalhar. Caso contrário, o nó ocioso finaliza os seus trabalhos e espera pelos demais.

Para avaliar as duas aplicações, foram realizados dois testes: um com o tabuleiro de 5×5 e outro com o tabuleiro de 5×6. Os resultados foram obtidos tirando a média de cinco medidas de tempo de execução de cada aplicação para cada caso analisado.

4.1. Primeiro teste: tabuleiro de 5×5

Neste primeiro teste há 1.728 soluções e, além de usar o *cluster*, foi simulado um sistema distribuído desbalanceado

utilizando o mesmo *cluster* com outras aplicações sendo executadas.

4.1.1. *Cluster* balanceado Nesta situação, variando o número de processadores, mediu-se o tempo de execução dessas aplicações e construiu-se o gráfico de *speedup* [7], figura 10.

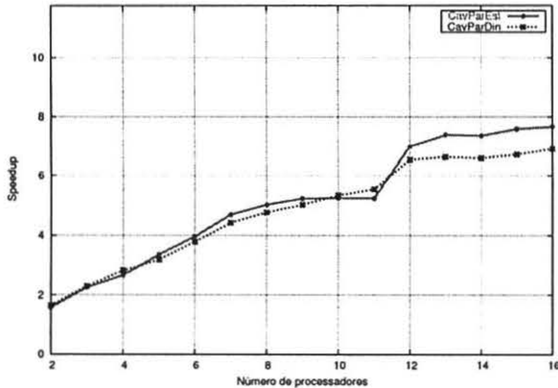

Figura 10. *Speedup* das aplicações paralelas rodando em um *cluster* balanceado. Tabuleiro de 5×5.

A partir do gráfico da figura 10, percebe-se que a aplicação paralela `CavParDin` foi mais lenta do que `CavParEst` na maioria dos casos, mas não muito, indicando que a sobrecarga imposta pelo balanceamento dinâmico de carga não é alta. Este caso não é muito propício para o balanceamento dinâmico porque, após a divisão inicial, não é grande o desequilíbrio de cargas entre os computadores. Isso ocorre devido ao fato de que o tempo de cálculo do número de soluções para cada casa onde o cavalo inicia o seu percurso não é muito diferente das demais, logo, o sistema distribuído já está razoavelmente balanceado e o tempo gasto com o balanceamento dinâmico só tende a prejudicar o desempenho.

4.1.2. *Cluster* desbalanceado Aqui, para desbalancear o *cluster* foi colocada a aplicação `burnCPU` (uma aplicação de uso intensivo de CPU) para ser executada em metade dos nós. Este teste foi realizado da seguinte forma: para um certo número de processadores P, foram colocados dois `burnCPU` em $P/2$ processadores se P for par, ou $(P + 1)/2$ se P for ímpar. Desta maneira, nos nós onde houver `burnCPUs` rodando, a aplicação paralela que estiver sendo executada terá apenas aproximadamente $1/3$ da CPU disponível, enquanto nos outros, ela terá quase 100% da CPU. Os resultados obtidos neste teste podem ser visualizados na figura 11.

No gráfico da figura 11, vê-se que `CavParDin` é mais rápido do que a outra aplicação em todos os casos, evidenciando que, em sistemas distribuídos desbalanceados, o balanceamento dinâmico de cargas é melhor do que o estático.

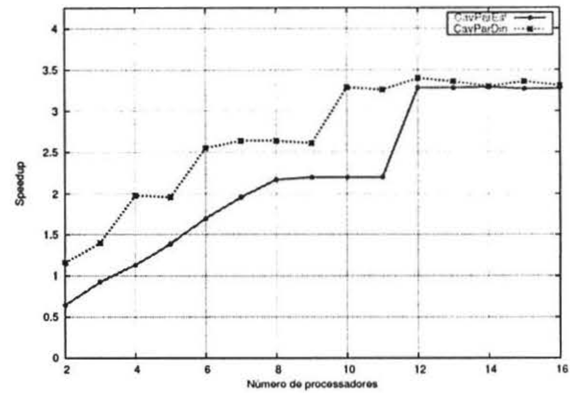

Figura 11. *Speedup* das aplicações paralelas rodando em um *cluster* desbalanceado. Tabuleiro de 5×5.

4.2. Segundo teste: tabuleiro de 5×6

Este teste foi realizado apenas com o *cluster* balanceado, encontrando 37.568 soluções. Os resultados dos *speedups* das duas aplicações são mostrados no gráfico da figura 12.

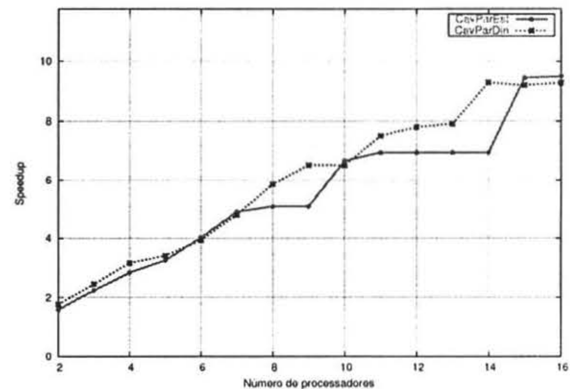

Figura 12. *Speedup* das aplicações paralelas rodando em um *cluster* balanceado. Tabuleiro de 5×6.

A partir do gráfico da figura 12, nota-se que `CavParDin` é mais rápido do que `CavParEst` na maioria dos casos, pois, após a divisão inicial, o desequilíbrio de cargas dos nós do *cluster* ainda é grande. Isso ocorre porque o tempo de cálculo do número de soluções para uma dada casa inicial do tabuleiro é grande (muito maior que no caso do tabuleiro de 5×5) e di-

fere muito dos tempos de cálculo das outras casas, logo é possível ganhar muito em desempenho equilibrando dinamicamente a carga do sistema.

5. Conclusões

Neste artigo, foi apresentada uma ferramenta de monitoramento de cargas em sistemas distribuídos. Dentre as características mais importantes desta ferramenta destaca-se o fato dela ser portátil, porque ela pode ser utilizada em grandes sistemas distribuídos que sejam heterogêneos e que até possam possuir nós com sistemas operacionais diferentes. Isso pôde ser obtido graças ao uso da linguagem de programação Java.

Outro aspecto importante desta ferramenta reside no fato de ser orientada a objetos, o que permite o seu fácil uso em outras aplicações. Além disso, utilizar um objeto para representar o índice de carga e não apenas um número torna possível transmitir mais informação a respeito do computador que está sendo analisado. Assim, um único objeto pode conter informações sobre vários tipos de carga, por exemplo. Dados úteis que esse tipo de objeto deve conter são informações como: tipo de CPU, freqüência de operação, capacidade total da memória física, tipo de memória física, interface de rede, enfim algumas informações essenciais para determinar o poder computacional de um certo nó. Também é possível registrar diversos tipos de medidas de carga, inclusive tipos definidos especificamente para cada aplicação.

Os resultados apresentados na seção anterior mostram que o sistema de balanceamento dinâmico de cargas desenvolvido para testar o sistema de monitoramento foi melhor do que o sistema de balanceamento estático na maioria dos casos testados. Isso não só demonstra que o sistema de balanceamento dinâmico de cargas foi eficiente como também comprova a afirmação de que mesmo em sistemas distribuídos cujos nós sejam iguais, o balanceamento de cargas dinâmico é melhor do que o estático [13]. Além disso, conclui-se, a partir desses resultados, que o sistema de monitoramento de cargas impõe pouca sobrecarga no sistema distribuído. Este é um dos resultados mais importantes deste artigo, por satisfazer um dos maiores objetivos desta ferramenta.

A ferramenta apresentada neste artigo constitui uma iniciativa para o desenvolvimento de um sistema de balanceamento de cargas em *Grids* computacionais. Além de fornecer a carga dos nós de um sistema distribuído às aplicações paralelas, ou a um sistema de balanceamento de cargas, a ferramenta desenvolvida também pode ser usada para gerenciamento de um computador paralelo. Desta maneira, o administrador de um sistema distribuído pode saber como os recursos computacionais de seu sistema estão sendo usados.

Referências

[1] N. Basnet, D. Pokharel, and S. Adhakari. STATE of CPU USAGE. Technical report, Department of Computer Science and Engineering, Kathmandu University, 2003.

[2] K. M. Baumgartner and B. W. Wah. Computer Scheduling Algorithms – Past, Present, and Future. *Information Sciences*, pages 319–345, Sep 1991.

[3] T. L. Casavant and J. Khul. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2), February 1988.

[4] D. E. Culler, J. Singh, and A. Gupta. *Parallel Computer and Architecture: A Hardware/Software Approach*. Morgan-Kaufman, 1998.

[5] D. Fanagan. *Java in a Nutshell*. O'Reilly, 3nd edition, May 1997.

[6] D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[7] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[8] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, 2002.

[9] D. Gupta and P. Beard. Load Sharing in Distributed Systems. In *Proceedings of the National Workshop on Distributed Computing*, January 1999.

[10] P. Krueger and M. Livny. The Diverse Objectives of Distributed Scheduling Policies. In *Seventh Int'l Conf. Distributed Computing Systems*, pages 242–249, Los Alamitos, California, 1987. IEEE CS Press.

[11] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. In *ACM Computer Network Performance Symp.*, pages 47–55, 1982.

[12] H. Nishikawa and P. Steenkiste. A general architecture for load balancing in a distributed-memory environment. In *International Conference on Distributed Computing Systems*, pages 47–54, 1993.

[13] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, pages 33–44, December 1992.

[14] P. S. L. Souza. *AMIGO: Uma Contribuição para a Convergência na Área de Escalonamento de Processos*. PhD thesis, Universidade de São Paulo, Instituto de Física de São Carlos, 2000.

[15] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[16] Y. T. Wang and R. J. T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, pages 204–217, March 1985.

[17] C. Xu and F. C. M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Boston, USA, 1997.