UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Uma Solução de Escalonamento para o DPC++

por

ELGIO SCHLEMER

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, abril de 2002.

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Schlemer, Elgio

Uma Solução de Escalonamento para o DPC++ / por Elgio Schlemer. — Porto Alegre: PPGC da UFRGS, 2002.

85 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2002. Orientador: Navaux, Philippe Olivier Alexandre.

1. Escalonamento. 2. Processamento Distribuído. 3. DPC++. 4. Objetos Distribuídos. I. Navaux, Philippe Olivier Alexandre. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof[a]. Wrana Maria Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fernsterseifer
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Senhor, deste-me tanto.
Dá-me uma coisa a mais: um coração agradecido.
(George Herbert)

Agradecimentos

Agradeço à minha mãe, Eroni Luiza König Schlemer e a meu pai Erno Schlemer que com muito esforço e dificuldade não mediram esforços investindo em minha educação. A minha irmã, Elisa Schlemer, a quem muito admiro pelo seu maravilhoso dom de lidar com crianças e por me lembrar algumas vezes, que na vida não existem apenas computadores.

Aos meus amigos, abandonados, que entenderam (ou não) minha dedicação a este trabalho e aos colegas do grupo de Processamento Paralelo e Distribuído, pela ajuda e apoio. Aos amigos Maurício Lima Pilla e Rafael Bohrer Ávila pela ajuda e os inúmeros *coffee-breaks*.

E a Deus, por ter colocado estas pessoas em meu caminho.

"A pior coisa na vida de um ateu é quando ele sente necessidade de agradecer a alguém, mas não sabe a quem". (Anônimo)

Sumário

Lista de Abreviaturas 7

Lista de Figuras 8

Lista de Tabelas 9

Resumo 10

Abstract 11

1 Introdução 12

2 Sistemas Paralelos e Distribuídos 14
2.1 Tipos de Comunicação 14
2.2 Bibliotecas de Comunicação 16
2.2.1 Biblioteca PVM 16
2.2.2 Padrão MPI 18
2.2.3 Padrão OpenMP 20
2.3 *Cluster* de Alto Desempenho 21
2.3.1 Padrão SCI 23
2.3.2 Exemplos de *clusters* 23
2.3.3 Clusters do Instituto de Informática da UFRGS 25
2.4 Considerações Finais 28

3 Escalonamento de Tarefas 30
3.1 Classificação dos escalonadores 30
3.1.1 Escalonamento Estático 31
3.1.2 Escalonamento Dinâmico 32
3.1.3 Escalonamento Adaptativo 33
3.2 Políticas de Escalonamento 33
3.3 Avaliando a Carga 35
3.3.1 A Carga de uma Tarefa 37
3.3.2 Carga em *clusters* 39
3.4 Exemplos de Escalonamento 39
3.4.1 Usando a ociosidade das estações 40
3.5 Considerações Finais 41

4 DPC++ 42
4.1 Estrutura do DPC++ 42
4.2 Objeto Diretório 43
4.3 Tolerância a Falhas no DPC++ 45
4.4 Concorrência entre Métodos no DPC++ 46
4.5 DECK 48
4.5.1 Camadas do DECK 48
4.5.2 Utilização do DECK 48
4.6 Integração do DPC++ com o DECK 49
4.7 Considerações finais 50

5 Modelo Proposto . . . 51
5.1 Análise do DPC++ . . . 51
5.2 Definição do modelo . . . 52
5.2.1 Objetos Espiões . . . 52
5.2.2 Escalonador Central . . . 54
5.3 Considerações Finais . . . 55

6 Implementação . . . 57
6.1 Adaptação do modelo ao DECK . . . 57
6.2 Implementação do Espião . . . 57
6.2.1 Funções internas do espião . . . 59
6.2.2 Estrutura de Dados utilizada . . . 60
6.3 Implementação do Escalonador Central . . . 61
6.3.1 Funções principais do escalonador central . . . 64
6.4 Adaptação . . . 66
6.5 Considerações Finais . . . 66

7 Avaliação e Resultados . . . 68
7.1 Aplicação Implementada . . . 68
7.2 Implementação distribuída do algoritmo . . . 68
7.2.1 Relacionamento da aplicação com o DPC++ . . . 70
7.2.2 Configurações da Aplicação . . . 70
7.3 Testes Realizados . . . 71
7.3.1 Divisão seqüencial . . . 72
7.3.2 Divisão aleatória . . . 75
7.3.3 Divisão otimizada . . . 77
7.4 Considerações Finais . . . 79

8 Conclusões . . . 80

Bibliografia . . . 82

Lista de Abreviaturas

BIP *Base Interface for Parallelizing*
CPU *Central Proccessor Unit*
CCS *Conputing Center Software*
DECK *Distributed Execution Communication Kernel*
DPC++ *Distributed Programming in C++*
DSM *Distributed Shared Memory*
GPPD Grupo de Processamento Paralelo e Distribuído
IP *Internet Protocol*
KLAT2 *Kentucky Linux Athlon Testbed 2*
MPI *Message Passing Interface*
MPICH *Message Passing Interface Chameleon*
NFS *Network File System*
NIS *Network Information Service*
PCI *Peripheral Connection Interface*
PVM *Parallel Virtual Machine*
RAM *Random Acess Memory*
RPC *Remote Procedure Call*
SCI *Scalable Coherent Interface*
SIMD *Single Instruction Multiple Data*
SMP *Symmetric Multi-Processor*
SPMD *Single Program Multiple Data*
TCP *Transfer Control Protocol*
UFRGS Universidade Federal do Rio Grande do Sul

Lista de Figuras

FIGURA 2.1 – Memória Distribuída 15
FIGURA 2.2 – Visão de uma Máquina Virtual no PVM 16
FIGURA 2.3 – Criação de Processos no PVM 17
FIGURA 2.4 – Envio de mensagens no PVM 18
FIGURA 2.5 – Programação em MPI 19
FIGURA 2.6 – Exemplo OpenMP 21
FIGURA 2.7 – Topologia *torus* bidimensional 24
FIGURA 2.8 – Organização do *cluster Myrinet* do GPPD 26
FIGURA 2.9 – Desempenho TCP no cluster *Myrinet* 27
FIGURA 2.10 – Desempenho do *cluster Myrinet* com BIP 27
FIGURA 2.11 – Organização do *cluster SCI* do GPPD 28

FIGURA 3.1 – Classificação dos escalonadores 30
FIGURA 3.2 – Uso dos espiões 33
FIGURA 3.3 – Exemplo de um fractal distribuído. 38

FIGURA 4.1 – Visão Geral do DPC++ 43
FIGURA 4.2 – Invocação de um Método em DPC++ 44
FIGURA 4.3 – Participação do objeto Diretório no DPC++ 44
FIGURA 4.4 – Representação do modelo proposto para o objeto Diretório. . . 46
FIGURA 4.5 – Concorrência entre métodos no DPC++ 47
FIGURA 4.6 – Camadas do deck 48

FIGURA 5.1 – Exemplo de um descritor no DPC++ 52
FIGURA 5.2 – Modelo do escalonador 52
FIGURA 5.3 – Funcionamento do espião 53
FIGURA 5.4 – Escalonador Central 55

FIGURA 6.1 – Serviço de Escalonamento do DECK 57
FIGURA 6.2 – Implementação do espião 59
FIGURA 6.3 – Função *deck_spy_init()* 60
FIGURA 6.4 – Função *deck_spy_main()* 61
FIGURA 6.5 – Função *deck_spy_run()* 62
FIGURA 6.6 – Estrutura de dados usada pelo espião 63
FIGURA 6.7 – Função *getANode()* 65

FIGURA 7.1 – Tempo de cada parte no modelo Mandelbrot 72
FIGURA 7.2 – Desempenho com Carga Baixa 73
FIGURA 7.3 – Desempenho com Carga Baixa e Injeção de Carga 73
FIGURA 7.4 – Desempenho com Carga Alta 74
FIGURA 7.5 – Desempenho com Carga Alta e Injeção de Carga 75
FIGURA 7.6 – Desempenho com Distribuição Aleatória 76
FIGURA 7.7 – Desempenho com Distribuição Aleatória e Injeção de Carga . . 78
FIGURA 7.8 – Desempenho com Distribuição Otimizada 79

Lista de Tabelas

TABELA 2.1 – Comparação de placas de rede 22
TABELA 2.2 – Características do *cluster* hpcline (junho de 2001) 24
TABELA 2.3 – Características do cluster *Myrinet* 26
TABELA 2.4 – Características do cluster SCI 28

TABELA 3.1 – Políticas de escalonamento 34
TABELA 3.2 – Valores de *Load Average* 37

TABELA 5.1 – Características do Modelo 56

TABELA 7.1 – Desempenho para Distribuição Aleatória 77
TABELA 7.2 – Desempenho com Distribuição Aleatória e Injeção de Carga 78

Resumo

Este trabalho descreve uma implementação de um modelo de escalonamento para a linguagem de programação DPC++. Esta linguagem, desenvolvida no Instituto de Informática da UFRGS, possibilita que uma aplicação orientada a objetos seja distribuída entre vários processadores através de objetos distribuídos. Muito mais que uma simples biblioteca de comunicação, o DPC++ torna a troca de mensagens totalmente transparente aos objetos. A integração do DPC++ com o DECK, também em desenvolvimento, trará grandes inovações ao DPC++, principalmente pelo uso de *threads*. O escalonador proposto para este modelo utiliza estes recursos para implantar os chamados processos espiões, que monitoram a carga de uma máquina, enviando seus resultados ao escalonador.

O escalonador implementado possui, desta forma, dois módulos: objetos espiões implementados como um serviço do DECK e o escalonador propriamente dito, incluído no objeto Diretório, parte integrante do DPC++.

Palavras-chave: Escalonamento, Processamento Distribuído, DPC++, Objetos Distribuídos.

TITLE: "DPC++ SCHEDULING"

Abstract

This work describes the implementation of a scheduling model for the DPC++ programming language. This language, developed at the Informatics Institute of UFRGS, allows an object-oriented application to be distributed across several nodes by means of distributed objects. More than a simple communication library, DPC++ provides completely transparent message-passing between the objects of an application. The integration of DPC++ with DECK, also under development at UFRGS, allows for innovations in DPC++, mainly for the use of threads. The scheduler proposed for this model makes use of such resources for carrying out the so-called spy objects, which monitor the status of a machine and send these data to the scheduler.

The implemented scheduler is, thus, composed of two modules: spy objects implemented as a DECK service, and the scheduler itself, whose functionality is included in DPC++'s Directory object.

Keywords: Scheduling, Distributed Processing, DPC++, Distributed Objecs.

1 Introdução

A busca por desempenho em sistemas computacionais tem sido alvo de inúmeras pesquisas desde os primórdios da computação. Uma das formas de se atingir este aumento de desempenho é através do simples aumento do poder computacional dos processadores envolvidos, seja através de sua velocidade, possibilitando realizar um maior número de tarefas por unidade de tempo ou incorporando técnicas que tornam mais rápidas tarefas que oneravam o processador. Os processadores ficaram não apenas mais rápidos, executando mais instruções por unidade de tempo, como também mais eficientes, graças à técnicas como *cache* de instruções, sistema de pré-busca, *pipeline*, entre outras.

Ao mesmo tempo em que o poder individual de cada processador aumentou, uma outra forma de conseguir maior desempenho foi obtido através da interligação de vários processadores, de forma que eles cooperem para a realização de uma tarefa. À medida que o aumento de poder individual de cada processador passou a enfrentar limitações físicas [KIT 96], a idéia de fabricar uma máquina com vários processadores tornou-se uma alternativa vantajosa.

Com a existência de várias unidades processadoras em uma única máquina, alguns problemas precisavam ser resolvidos, como fornecer comunicação entre eles e dividir da melhor forma possível as tarefas existentes.

Tornou-se fundamental o uso de ferramentas que dividissem a carga computacional de forma eficiente entre as várias unidades processadoras existentes. Estas ferramentas são conhecidas como escalonadores.

Um escalonador é um algoritmo que decide qual tarefa será executada em qual processador e quando. Mesmo com a existência de um único processador, em um sistema de compartilhamento de tempo, um escalonador poderá decidir qual processo receberá a CPU em determinada unidade de tempo. Entretanto, com a existência de vários processadores, um escalonador eficiente pode tornar-se fundamental para o desempenho. Já que a interligação de vários processadores traz um certo custo de comunicação, sincronização e compartilhamento de recursos comuns (denominado *overhead*), uma máquina com vários processadores gerenciados de forma equivocada pode até mesmo apresentar um desempenho inferior àquela de um único processador.

Nesta busca de melhor desempenho, várias formas de se interligar unidades processadoras foram pesquisadas e desenvolvidas. A primeira forma originou as chamadas máquinas paralelas, cuja principal característica é a comunicação através da memória. É uma solução cara, pois requer dispositivos de compartilhamento de memória sofisticados e o uso de um barramento ou *switch* e uma memória rápida.

Uma solução mais econômica, foi interligá-los através de uma rede de computadores com a comunicação sendo feita através de troca de mensagens. Várias máquinas monoprocessadas, com uma placa de rede, podem participar de uma máquina virtual com vários processadores. Neste caso, a paralelização de tarefas é feita através de algum *software* que procura esconder a topologia real, se aproximando de uma máquina paralela real.

Neste contexto é que a linguagem DPC++ (*Distributed Programming in C++*) [CAV 94] foi desenvolvida no Instituto de Informática da UFRGS, como uma solução para tornar esta comunicação transparente, com uma expressão de paralelismo simples, baseada na orientação a objetos.

O DPC++ possibilita a implementação de objetos que podem ser executados remotamente em outras máquinas de uma rede, interagindo através de troca de mensagens. Estas trocas de mensagens servem para implementar, de forma remota, o envio de parâmetros necessários para a execução de métodos, o recebimento dos resultados e também mensagens de controle para criar ou destruir um objeto. Estes objetos, no DPC++, são implementados como processos que são disparados de forma remota na máquina especificada. Com isto tem-se uma programação de granulosidade bastante alta, e um baixo *overhead* com comunicação.

Muitas mudanças tecnológicas ocorreram desde a concepção do DPC++, como o aumento significativo da largura de banda disponível nas placas de rede. Com redes mais rápidas e eficientes, novas possibilidades tornaram-se economicamente viáveis.

Tornou-se uma prática comum adquirir equipamentos especialmente montados e configurados para serem usados em conjunto, de forma agregada. Desta forma, popularizou-se o uso dos chamados *clusters*, desde o início concebidos para trabalharem em conjunto.

Um *cluster* possui placas de rede com largura de banda elevada e unidades processadoras totalmente homogêneas e simétricas (mesmo *hardware* e mesma configuração). Com uma comunicação mais rápida e o uso exclusivo para a distribuição de tarefas, a possibilidade de diminuir a granulosidade das tarefas tornou-se viável, bem como simular o compartilhamento de memória através de troca de mensagens.

Tendo em vista os resultados desejados do DPC++ e seu aperfeiçoamento com inclusão de novos recursos, a necessidade de um mecanismo de escalonamento tornou-se indispensável. O objetivo deste trabalho é fornecer este dispositivo de escalonamento para o DPC++.

Inicialmente, no Capítulo 2, as várias formas de comunicação existentes, ferramentas utilizadas e as tecnologias de rede existentes no mercado são estudas. No Capítulo 3 são descritas técnicas clássicas e modernas de escalonamento, bem como a teoria existente, tipos de escalonadores, forma de se avaliar a carga de um processador e estudos de casos.

A descrição da linguagem DPC++, a forma como é implementada e os trabalhos já realizados neste projeto são estudados no Capítulo 4, bem como uma introdução ao ambiente DECK.

Após a avaliação e estudo das técnicas existentes, um modelo de escalonamento para o DPC++ é descrito no Capítulo 5. O Capítulo 6 descreve a forma como o modelo foi implementado, e de que forma o modelo de escalonamento se relaciona com o DECK. Uma avaliação do modelo encontra-se no Capítulo 7, e por fim, no Capítulo 8 conclusões são descritas.

2 Sistemas Paralelos e Distribuídos

Muito se tem buscado aumentar o desempenho através do uso de vários processadores. Inicialmente buscava-se uní-los em uma única máquina, para que compartilhassem a mesma memória e trabalhassem em conjunto na execução de determinada tarefa. Quando vários processadores cooperam para a resolução de um mesmo problema, soluções precisam ser encontradas para resolver problemas de concorrência a um mesmo recurso de *hardware* e para determinar a melhor maneira de mantê-los ocupados o maior tempo possível.

Ao analizar o desempenho destes processadores interligados, percebe-se que o desempenho de uma máquina não cresce proporcionalmente ao número de processadores disponíveis. Ocorre, inclusive, um declínio do desempenho quando o número de processadores atinge um determinado valor. Este declínio se deve a vários fatores, como por exemplo os custos gerados pela comunicação necessária para sincronizar os processadores ou para trocar dados uns com os outros (denominado *overhead*) ou pela dificuldade de manter todos os processadores ocupados.

2.1 Tipos de Comunicação

A forma como estes processadores se comunicam pode ser, basicamente, duas:

- através da memória;
- através de troca de mensagens.

Para que os processadores possam comunicar-se através da memória, é necessário que exista uma área de memória comum a todos os processadores. Máquinas que dispôem destes recursos são conhecidas como "Máquinas com memória compartilhada".

Já o uso de troca de mensagens não necessita que todos os processadores tenham acesso a uma única memória. Máquinas sem uma memória global são conhecidas como "Máquinas com Memória Distribuída".

Basicamente, são quatro os modelos de memória existentes: UMA, NUMA, COMA e NORMA [DER 2002]:

- UMA (*Uniform Memory Access*). Neste modelo, a memória física é compartilhada uniformemente entre os processadores. Todos eles tem tempo igual de acesso a todas as palavras de memória.
- NUMA (*NonUniform Memory Access*). Neste modelo, o tempo de acesso à memória varia de acordo com a localização da palavra (local ou remota). Um exemplo seria uma máquina composta de vários processadores onde cada um possui uma memória local que é compartilhada com os demais (o conjunto de memórias locais formam um espaço de endereçamento global). Neste caso, os processadores acessam a memória local mais rapidamente do que o espaço de endereçamento global.
- COMA (*Cache-Only Memory Architecture*). É um tipo especial de de NUMA, onde as memórias globais distribuídas são substituídas por *caches*. Não existe

uma hierarquia de memória em cada processador. Todas as *caches* formam um espaço de endereçamento global. Um sofisticado mecanismo de coerência de *cache* em *hardware* se faz necessário, o que justifica o alto custo desta tecnologia.

- NORMA (*NO Remote Memory Access*). Neste modelo, não existe um espaço de endereçamento global. Cada processador tem sua memória local e a troca de dados é realizada através de troca de mensagens.

A programação por memória compartilhada, tipicamente máquinas UMA e NUMA, é bastante intuitiva e simples, pois estruturas inteiras de dados podem ser trocadas entre os vários processadores. Basta que os dados sejam colocados na memória e todos os processadores terão acesso. Torna-se viável o paralelismo com granulosidade bastante fina, como *threads*, funções ou mesmo instruções.

Mesmo com a facilidade em termos de programação, o compartilhamento de memória acaba inviabilizando-se à medida que o número de processadores aumenta consideravelmente, seja por limitações técnicas para garantir o acesso sem conflitos, ou por questões econômicas.

Já a distribuição de memória (NORMA) é mais viável economicamente, pois dispensa maiores dispositivos de gerência de memória, provendo a comunicação através de troca de mensagens.

Na Figura 2.1 observa-se que para um processador trocar informações com algum outro, faz-se necessário utilizar algum mecanismo que os interligue, permitindo o envio de mensagens [KIT 95], como uma rede, por exemplo.

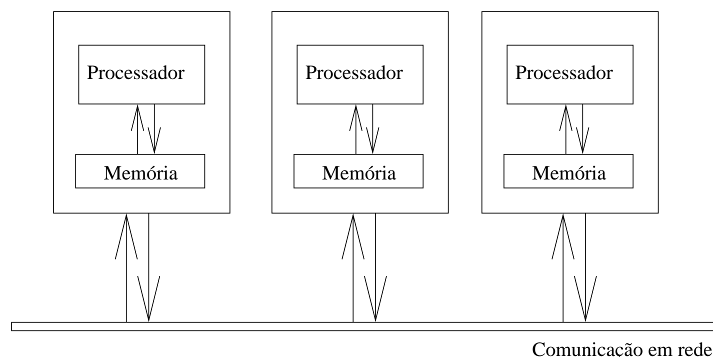

FIGURA 2.1 – Memória Distribuída

As desvantagens da distribuição de memória são basicamente duas:

- a programação é mais complicada e o paralelismo não é tão intuitivo;
- ao se exigir uma comunicação mais intensa, o desempenho acaba comprometido pelo alto custo envolvido na troca de mensagens.

A programação é complicada porque as tarefas devem enviar e receber mensagens de forma explícita e isto deve ser expresso pelo programador.

Para facilitar a programação em máquinas com memória distribuída, buscou-se livrar o programador ao máximo dos detalhes da comunicação através de bibliotecas de comunicação.

2.2 Bibliotecas de Comunicação

As chamadas bibliotecas de comunicação foram desenvolvidas para prover facilidades na comunicação entre processos executando em máquinas diferentes, possibilitando assim a programação por troca de mensagens.

A seguir serão discutidas algumas destas bibliotecas.

2.2.1 Biblioteca PVM

A biblioteca PVM (*Parallel Virtual Machine*) [GEI 94] é uma das mais utilizadas na programação distribuída ainda nos dias de hoje. É uma biblioteca inicialmente baseada em *sockets* (protocolo IP). Possui muitos recursos de gerenciamento remoto de tarefas, não se limitando apenas a prover comunicação entre os processos. A Figura 2.2 ilustra o uso do PVM em uma rede.

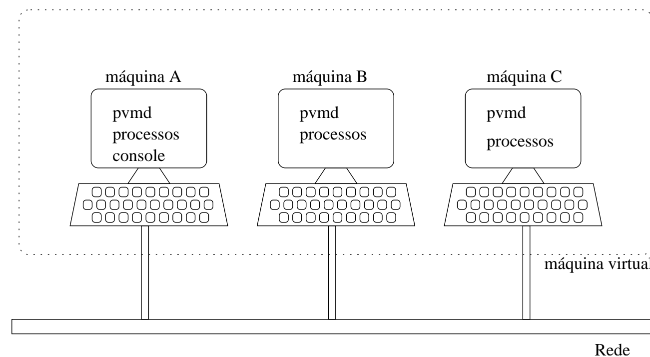

FIGURA 2.2 – Visão de uma Máquina Virtual no PVM

O PVM pode ser dividido em três partes:

- um console de configuração: O PVM tem como objetivo gerenciar uma máquina paralela como se ela fosse real. Para isso, existe um aplicativo que deve ser iniciado em qualquer uma das máquinas onde deseja-se montar uma máquina virtual. A partir deste ponto, pode-se "chamar" outras máquinas a participarem da máquina virtual. De qualquer máquina participante é possível remover ou iniciar processos, remover ou inserir máquinas, bem como monitorar o andamento da execução das tarefas.
- biblioteca de funções: Uma biblioteca é fornecida onde estão implementadas várias funções úteis à comunicação e gerenciamento de processos. Esta biblioteca deve ser inserida junto com a aplicação que fará uso destas funções.
- um *daemon*: cada vez que uma máquina é inserida no PVM, um processo é iniciado na máquina de forma remota, para possibilitar seu gerenciamento. Este processo fica responsável pela máquina na qual se encontra. Dependendo do tipo de troca de mensagens utilizado, o *daemon* é responsável também pela entrega local das mensagens aos processos.

Quanto a comunicação o PVM implementa as primitivas *pvm_send()*, responsável pelo envio de mensagens a um processo, e *pvm_receive()* para receber uma mensagem de um processo. O recebimento de mensagem é do tipo bloqueante, ou

seja, o processo fica bloqueado até que a mensagem desejada esteja disponível. Existem muitas formas de enviar mensagens no PVM, como a possibilidade de estabelecer um tempo máximo de espera de uma mensagem, de poder indicar de qual processo deseja-se receber uma determinada mensagem, ou ainda o envio de mensagens para vários processos, através de comunicação em grupo.

Quanto ao gerenciamento, destacam-se as funções que possibilitam iniciar um determinado processo e as funções que permitem que qualquer processo obtenha o identificador de outro. Este identificador é necessário para que a comunicação se torne possível.

Na Figura 2.3, um exemplo de código criando processos no PVM é mostrado. Neste exemplo, um console deve ter sido previamente construído pelo programador antes de usá-lo. A função *pvm_spawn* tenta criar dez processos de nome "hello_word" em quaisquer nodos existentes. Logo após, espera-se que cada um dos processos criados envie uma mensagem ao processo que os criou, denominado de "processo pai".

```
int main()
{
    int tid[TAM], i, quem;
    char buf[50];

    cc = pvm_spawn("hello_other", (char**)0, 0, "", 10, &tid);
        /* criando 10 processos de nome "hello_other" nas maquinas
           participantes da maquina virtual */

    if (cc == 10){          /* verifica se criou todos */
        for (i=0;i<10;i++){
            cc = pvm_recv(-1, -1); /* recebe uma mensagem */
            pvm_bufinfo(cc, &tam, (int*)0, &quem);
                /* Obtem informacoes da mensagem */
            pvm_upkstr(buf); /* desempacota a mensagem em buf */
            printf("Recebeu %s de %d\n", buf, quem);
        }
    }else {
        fprintf(stderr, "Erro ao criar processos"\n);
        exit(0);
    }
    pvm_exit();
}
```

FIGURA 2.3 – Criação de Processos no PVM

Já na Figura 2.4 tem-se o código responsável pelo envio destas mensagens ao processo pai. Observa-se, principalmente, a necessidade de empacotar as mensagens antes de enviar e de desempacotar ao receber. Isto é necessário pelo fato do PVM poder conectar máquinas heterogêneas de arquiteturas diferentes, como Intel e Sun, por exemplo. O empacotamento converte os dados para um padrão PVM.

```
int main()
{
    int pai, eu;
    char buf[50];

    pai = pvm_parent();              /* obtem o identificador de quem criou */
    eu  = pvm_mytid();               /* obtem o seu identificador */
    sprintf(buf,"Ola de %d", eu);  /* Prepara mensagem */
    pvm_pkstr(buf);                  /* empacota a mensagem */
    pvm_send(pai, 1);                /* envia mensagem para quem o criou */
    pvm_exit();                      /* encerra e sai do pvm */
}
```

FIGURA 2.4 – Envio de mensagens no PVM

O PVM não deixa transparente o gerenciamento de nodos e a troca de mensagens entre os processos da máquina virtual. É o programador quem decide quando um processo deve ser criado. A troca de mensagens também é explícita e muitas vezes burocrática, porque o programador precisa escolher a forma de envio, empacotar todos os seus dados e enviá-los especificando o destino (através do identificador do processo).

Outro ponto negativo do PVM é a necessidade da construção de uma máquina virtual antes de iniciar a aplicação distribuída. Esta criação pode ser feita dentro do código da aplicação, mas isto não é recomendável. Uma boa prática é construir a máquina virtual e deixar os *daemons* em permanente execução. Desta forma pode-se usar a máquina virtual várias vezes, sem a necessidade de configurá-la constantemente.

2.2.2 Padrão MPI

O MPI (*Message-Passing Interface*) não é exatamente uma biblioteca de comunicação, mas sim um padrão de comunicação desenvolvido pelo Fórum MPI [MPI 94]. A comparação com seu antecessor, o PVM, é inevitável [RÍM 97], mas as implementações do MPI tem um propósito diferente.

A forma como os processos são criados no PVM possibilita a criação de uma arquitetura onde qualquer processo pode ser criado a qualquer momento em qualquer máquina. No PVM estes processos podem ser de códigos diferentes. O MPI não permite este tipo de criação, pois sua estrutura segue o padrão SPMD (*Single Process, Multiple Data*), onde um mesmo cógido binário estará em execução em todos os nodos.

O MPI não possui o conceito de máquina virtual, onde máquinas podem ser inseridas ou removidas durante a execução da aplicação e o programador não tem a possibilidade de iniciar a execução de um processo de forma remota. Quando uma aplicação MPI é iniciada, o nome do processo deve ser fornecido. Este entrará em execução em todos os nodos, sendo o único processo, pertencente a aplicação, em execução em cada nodo. Como não é possível ao programador criar outros processos, o custo de inicialização ocorre somente no início.

Quanto a troca de mensagens, o MPI também possui algumas limitações em relação ao PVM, visando eficiência. Uma mensagem não é enviada a um processo específico mas sim para um nodo. Quando a aplicação é iniciada, o nodo que originou o disparo da aplicação recebe o identificador 0, indicando tratar-se do nodo principal. Cada um dos demais nodos participantes será numerado de 1 a $N-1$, sendo N o número total de nodos existentes.

Por estas características, não é necessária a existência de qualquer *daemon* ou sistema de identificação de processos. Um sistema de escalonamento também não tem sentido, pois todos executam o mesmo processo e a carga computacional destes processos dependerá tão somente dos dados manipulados por cada um.

É possível ao programador simular vários processos no MPI. Como existe a possibilidade de um nodo identificar-se perante os demais, conhecendo seu identificador e também o número total de participantes, comandos de seleção podem ser colocados no código, fazendo com que determinados trechos sejam executados somente por este ou aquele nodo. Basicamente este recurso é frequentemente utilizado para diferenciar o *nodo 0* dos demais, elegendo-o como um servidor de dados, por exemplo.

Na Figura 2.5 um código MPI é mostrado, onde pode-se observar a existência de apenas um código e que todos os nodos diferentes de zero, ou seja, exceto o nodo principal, irão enviar mensagens para o *nodo 0*. Este, por sua vez, ficará aguardando mensagens de todos os demais nodos.

```
#include <mpi.h>

int mpierr, rank, size,c;
MPI_Status status;
char *msg=”OK”, temp[10];

int main()
{
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        /* Eh o nodo principal , nodo 0 */
        for (c=1;c<size; c++){
            MPI_Send(msg, 2, MPI_CHAR, c, 1,MPI_COMM_WORLD);
            MPI_Recv(temp, 10, MPI_CHAR, c, 1, MPI_COMM_WORLD,&status);
        }
    } else {
        MPI_Recv(temp, 10, MPI_CHAR, 0, 1,MPI_COMM_WORLD,&status);
        MPI_Send(msg, 2, MPI_CHAR, c, 1,MPI_COMM_WORLD);
        MPI_Finalize();
    }
}
```

FIGURA 2.5 – Programação em MPI

Percebe-se que enquanto o PVM foi desenvolvido para ser uma ferramenta bastante flexível, possibilitando a comunicação entre máquinas de arquiteturas diferentes (Sun com Intel, por exemplo), o MPI já nasceu voltado ao conceito de *cluster* [LAU 97], onde todas as máquinas são homogêneas e executam o mesmo código.

O padrão MPI-2 [MPI 97], surgido em 1997, mudou muitas coisas no padrão MPI original, provendo outros tipos de comunicação e incorporando ao padrão a rotina *MPI_spawn()*, responsável pela criação de novos processos. Com isso, o padrão deixou de ser puramente SPMD, podendo o programador construir programas quase iguais ao PVM.

Alguns fabricantes de placas de rede ou fabricantes de *hardware* construíram suas próprias implementações do padrão MPI, otimizando-o para sua plataforma.

Estão disponíveis desde MPI para rede IP, usando *sockets* como no PVM, até os específicos para determinada placa de rede.

2.2.3 Padrão OpenMP

O OpenMP não é uma biblioteca de comunicaçã, pois tem como objetivo prover a comunicação entre máquinas com memória compartilhada, mais especificamente, àquelas que simulam memória compartilhada em um *hardware* com memória distribuída [OPE 2001].

A necessidade de técnicas deste tipo deve-se à falta de portabilidade de programas escritos para este tipo de arquitetura, pois cada fabricante tem sua própria maneira de compartilhar áreas da memória com os outros processadores. Isto fez com que muitos programadores passassem a usar bibliotecas de comunicação, mesmo dispondo de recursos de compartilhamento de memória. Somente neste aspecto é que uma comparação do MPI com o OpenMP pode ser feita, já que o OpenMP não serve para comunicação por troca de mensagens.

O OpenMP é um conjunto de diretivas de programação com memória compartilhada, cujo principal foco é a linguagem FORTRAN, que é muito limitada na manipulação de *threads*.

O mais interessante é que o OpenMP não exige que o usuário se preocupe com detalhes da paralelização. Pode-se escrever um programa praticamente seqüencial e o OpenMP se encarrega de dividí-lo entre várias *threads* em execução nos vários processadores existentes.

Na Figura 2.6 pode-se verificar um pequeno trecho de código em FORTRAN usando as diretivas do OpenMP. Todas as chamadas do OpenMP são comentários em um compilador FORTRAN normal. Na linha 3 do exemplo, o programador abriu uma sessão paralela determinando uma variável privada X e uma compartilhada w. Na linha 4 determinou uma variável do tipo redução em soma. As variáveis globais (como o I do laço) são classificadas como privadas em cada *thread*.

O que o OpenMP faz é dividir as iterações do laço entre os processadores disponíveis. Pode-se veriricar que isto é quase que um paralelismo implícito, pois o programador não necessita informar quantos são os processadores e em quantas partes o laço deve ser dividido. Na linha 8 está o final da sessão paralela, onde todas as *threads* são sincronizadas. Sempre no fim de cada sessão paralela as *threads* sincronizam implicitamente.

```
1    C trecho de codigo para calcular o PI
2
3    !$OMP PARALLEL DO PRIVATE(x),SHARED(w)
4    !$OMP REDUCTION (+:sum)
5         do i = 1, n
6              x = w * (i - 0.5d0)
7              sum = sum + f(x)
8         enddo
9    C Continuacao do codigo
```

FIGURA 2.6 – Exemplo OpenMP

O programador, se quiser, pode gerenciar alguns detalhes de paralelização:

- especificar o número máximo de *threads* geradas;
- colocar sincronização explícita, como barreiras;
- determinar o tipo de escalonamento usado;
- usar diretivas para avaliar o desempenho, entre outros.

Quanto ao escalonamento, o OpenMP oferece basicamente três possibilidades:

- Estático: O OpenMP dividirá uma sessão em n tarefas e usará n *threads* para executá-las.
- Dinâmico: O OpenMP criará n *threads* para executar as tarefas. Se o número de tarefas for maior que o de *threads* disponíveis, uma *thread*, ao terminar, ganhará mais uma porção para calcular.
- *Runtime*: O tipo de escalonamento será escolhido em tempo de execução.

Outros recursos estão disponíveis ao programador, como a manipulação mais detalhada das *threads* existentes, a definição de áreas sequenciais, uso de funções atômicas, áreas de exclusão mútua, entre outras.

Particularmente, OpenMP e MPI possuem características que os tornam bastante atraentes no uso de máquinas de alto desempenho. Com redes cada vez mais rápidas a um custo satisfatório, tornou-se prática comum a construção de agregados, conhecidos como *clusters* de alto desempenho.

2.3 *Cluster* de Alto Desempenho

Um *cluster* tem características mais complexas que as máquinas paralelas virtuais implementada pelo PVM ou por bibliotecas de comunicação. Enquanto estas bibliotecas possibilitam a interligação de máquinas comuns, construidas para serem consoles, interligadas por uma rede abrangendo várias salas ou prédios, um *cluster*, também chamado de "agregado", foi concebido desde o início com a finalidade de obter alto desempenho na distribuição de tarefas. Não são máquinas espalhadas por uma sala ou prédio, mas computadores interligados com uma rede de alta velocidade, com o mínimo de serviços.

Não se trata, então, de simplesmente pegar uma porção de máquinas ociosas, colocar alguma camada de *software* para possibilitar a comunicação entre processos e usá-las como uma máquina paralela, como é a proposta das bibliotecas de comunicação. Um *cluster* é tecnicamente construído, interligado, exclusivamente para esta finalidade, mesmo que cada nodo seja efetivamente uma máquina independente.

O objetivo da construção de qualquer agregado é o de aproximar-se do desempenho de uma máquina verdadeiramente paralela, diminuindo os custos provenientes da comunicação, sincronismo e gerenciamento. Outra característica desejável nestes equipamentos é a sua escalabilidade, ou seja, a possibilidade de aumentar-se o número de máquinas participantes sem que isto corresponda a um declínio significativo de desempenho. Máquinas puramente paralelas, utilizando uma única memória tornam-se muito caras e complexas quando utilizadas com muitos processadores, não sendo escaláveis.

Um dos fatores limitantes à clusterização era a largura de banda disponível das placas de rede. Se a largura de banda é fixa e o meio é compartilhado, como é no caso de redes utilizando o padrão *Ethernet*, aumentar o número de participantes implica em aumentar a concorrência ao meio de comunicação. Neste sentido buscou-se, não apenas o desenvolvimento de redes com largura de banda maior, mas também alternativas de interligação, como o uso de *switches*, por exemplo. O uso de várias placas de rede e várias formas de interligar estas placas também é amplamente utilizado.

Redes de altíssimo desempenho, especialmente para uso em agregados, também estão em desenvolvimento. Uma destas redes é a rede *Myrinet*, desenvolvida pela *Myricon*, que possui, em sua primeira versão, uma largura de banda de 1,28 Gb/s. Esta largura de banda é maior, inclusive, que a de alguns barramentos disponíveis. A Tabela 2.1 traz uma relação entre as redes atualmente disponíveis e o barramento existente em uma arquitetura Intel típica.

TABELA 2.1 – Comparação de placas de rede

Nome	Meio físico	Velocidade
Ethernet	Compartilhado ponto a ponto	10 Mbits/s
Fast Ethernet	Compartilhado ponto a ponto ou switch	100 Mbits/s
Myrinet	*Switch* (estrela)	1,28 + 1,28 Gbits/s
SCI	Anel (com ou sem switch)	4 Gbits/s
Barramento PCI	133 Mhz/16 bits compartilhado	2,00 Gbits/s

A comunicação entre cada processo está, algumas vezes, além da capacidade do barramento, o que torna impossível utilizar todos os recursos disponíveis pela tecnologia. Um agravante é o fato da placa de rede utilizar o mesmo barramento, que é disputado pelos demais periféricos existentes em uma máquina. Com tais características, ferramentas de gerenciamento de agregados e alternativas para diminuir o uso do barramento são constantemente utilizadas, sendo muitas vezes fator decisivo no desempenho.

Configurações onde as máquinas não rodam quaisquer serviços de autenticação de usuários (NIS, por exemplo) ou de compartilhamento de arquivos (como o NFS), são desejáveis, além do uso de ferramentas de gerenciamento. Com um conjunto de

máquinas é necessário ter dispositivos que permitam atualizações remotas e monitoramento, inclusive com a possibilidade de detectar máquinas com defeitos, evitando que ela comprometa alguma aplicação.

2.3.1 Padrão SCI

Uma das idéias mais interessantes é a simulação de uma memória compartilhada em uma memória fisicamente distribuída. Isso permite que a programação seja feita usando o paradigma de compartilhamento de memória, aliado ao baixo custo dos *clusters*.

Esta simulação pode ser feita por uma camada do Sistema Operacional que a implementaria, de forma transparente, através de troca de mensagens [AMZ 96]. São as conhecidas máquinas DSM (*Distributed Shared Memory*). Como a simulação em *software* traz a desvantagem do alto custo envolvido na troca de mensagens, buscou-se a sua simulação em *hardware*. Atualmente existem placas específicas de rede que fazem esta simulação, oferecendo às aplicações distribuídas uma memória compartilhada, como é o caso da tecnologia SCI.

A tecnologia SCI, especificada na norma IEEE 1596 [IEE 92], possibilita que um processo mapeie em seu espaço de endereçamento um segmento de memória que está fisicamente localizado em outra máquina, simulando, desta forma, memória compartilhada.

O padrão SCI (*Scalable Coherent Interface*) não surgiu com o propósito de ser utilizado como uma rede de comunicação. Seu objetivo é ser utilizado para interligar quaisquer dispositivos, não apenas máquinas [IEE 92]. Estes dispositivos podem ser periféricos, bancos de memória, processadores, etc.

Uma implementação deste padrão é fornecido pela empresa Dolphin, da Alemanha. Esta implementação inicialmente possui uma largura de banda nominal de 4 Gbits/s. Esta largura de banda ultrapassa a velocidade oferecida por alguns barramentos existentes.

A forma de ligação de uma rede SCI é realizada tipicamente em anel ou torus, de forma que um meio físico está sempre disponível entre um ponto e outro, mesmo que alguns nodos precisem percorrer uma distância maior para alcançar outros processadores.

2.3.2 Exemplos de *clusters*

Com a popularização das redes com largura de banda elevadas, muitos agregados tem sido construídos e muita pesquisa sobre a melhor forma de interligá-los vem sendo realizada. Existe, inclusive, uma lista dos mais poderosos, conhecida como TOP 500, que pode ser conferida em [CLU 2001].

São poucos, ainda, os representantes SCI nos *TOP500*, mas isto pode mudar rapidamente, pois a tecnologia SCI ainda é recente e está em processo de popularização. Atualmente existem apenas quatro representantes SCI, sendo as primeiras posições ocupadas por representantes *Myrinet*. A maioria das configurações utiliza tecnologias mais antigas, como redes *Fast Ethernet*, onde se destacam as técnicas usadas para ligar estes nodos, bem como o número de processadores existentes (dados de Junho de 2001).

Um representante desta lista é o *hpcline* da universidade de Paderborn [PAD 2001], composto de 96 nodos, cujas características são mostradas na

Tabela 2.2.

TABELA 2.2 – Características do *cluster* hpcline (junho de 2001)

Número de nodos	96, cada um com 2 processadores
Processador	Pentium III 850 Mhz
Memória/proc.	512 Mbytes
Disco rígido	4 Gbytes SCSI
Tipo de rede	Dolphin SCI 500 Mbytes/s e *Fast Ethernet*
Conexão	Torus bidimensinal (com *switch*)
Sistema	Linux Red Hat 6.2
Desempenho	163,2 GFlops/s

O sistema todo foi montado utilizando-se 48 gabinetes, sendo que cada gabinete contém duas placas e cada placa possui dois processadores. A interligação *torus* bidimensional permite que cada nodo pertença a dois anéis, da forma representada na Figura 2.7.

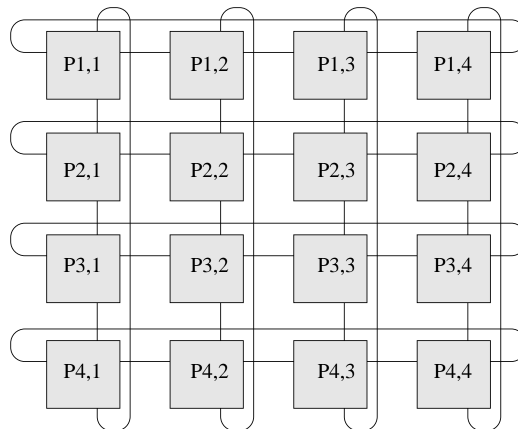

FIGURA 2.7 – Topologia *torus* bidimensional

Uma ferramenta de gerenciamento destes nodos foi desenvolvida, a CCS (*Computing Center Software*) [PAD 2001a], que tem a responsabilidade de gerenciar todos os nodos que participam do *cluster*. Isto se torna necessário pela dificuldade de gerenciar individualmente várias máquinas. Esta ferramenta é responsável por:

- acesso ao *cluster* através de uma única máquina (console);
- gerenciamento e agendamento de *jobs*;
- definição de autorização de acesso ao *cluster*;
  - quem pode utilizar quais recursos;
  - horários que um usuário pode usar;
  - limite de recursos para cada usuário;
- operações administrativas nos nodos;

- escalonamento dos pedidos dos usuários;
  - um usuário pode usar menos nodos que o total existente;
  - decide quais nodos ele usará;
- tolerância a falhas: controla nodos falhos e notifica a administração.

A placa de rede *Fast Ethernet* é conectada de forma tradicional, com o uso de um *switch*, e é usada basicamente para a manutenção e gerenciamento dos nodos.

Dentre as ferramentas de desenvolvimento disponíveis para o *hpcline*, encontram-se disponíveis o PVM e MPI para TCP/IP (usando a *Fast Ethernet*) e duas versões de MPI específicas para SCI: ScaMPI e MPICH.

É importante considerar que a largura de banda destas placas é muito alta, superior, muitas vezes, à do próprio barramento de dados da máquina. Neste caso uma otimização do tráfego gerado pelo gerenciamento é ponto fundamental no desempenho. O *hpcline* evita utilizar serviços do tipo NIS e NFS para não gerar comunicação desnecessária.

Um outro *cluster* que se destaca é o da Universidade de Delaware [DEL 2001], o *SAMson*. Baseado em processadores AMD, tipicamente mais baratos que os Intel e usando apenas um processador por nodo, ele ocupou a posição 22 na lista dos mais rápidos. Obteve um um desempenho de 132 Gflops/s (dados de Junho de 2001).

O *SAMson* possui 132 nodos conectados por uma rede SCI, cada um deles com um processador AMD *Athlon* de 1 Ghz e 512 Mbytes de memória. Destaca-se pelas ferramentas de gerenciamento empregadas e pela forma de conexão. Até detalhes como a padronização do comprimento dos cabos foram considerados na sua construção.

Resultados favoráveis também são atingidos com as redes tradicionais, como no caso do *cluster* KLAT2 (*Kentucky Linux Athlon Testbed 2*).

O KLAT2, da Universidade de Kentucky, na cidade de Lexington, USA, é baseado em estações AMD *Athlon*, composto de 64 nodos e dois outros exclusivos para gerenciamento. Cada um dos nodos possui um processador de 700 Mhz com 128 Mbytes de memória.

Mas não é na quantidade de nodos ou velocidade de processamento que o KLAT2 se destaca, mas sim na forma de conexão existente. Baseado em rede *Fast Ethernet*, os nodos foram interligados através de uma técnica chamada de *Flat Neighborhood Network*, onde qualquer nodo pode se comunicar com qualquer outro passando por apenas um *switch*. Dependendo do número de nodos existentes, um número maior de placas de rede se faz necessário em cada nodo e uma quantidade maior de *switches*. No caso do KLAT2, cada nodo possui quatro placas de rede e um total de 10 *switches* foram empregados.

Por utilizar um *hardware* mais barato que os demais *clusters* o KLAT2 se define como o *Giga Flop* mais barato do mercado, ficando perto de seiscentos dólares (em Junho de 2001).

2.3.3 Clusters do Instituto de Informática da UFRGS

A Universidade Federal do Rio Grande do Sul, através do seu Grupo de Processamento Paralelo e Distribuído (GPPD), foi uma das pioneiras no Brasil a pesquisar

na área de *cluster*, possuindo atualmente dois *clusters* de alto desempenho, um empregando rede *Myrinet* e outro SCI.

O primeiro cluster vem sendo montado desde 1997, tendo sido concluido com a aquisição de uma rede *Myrinet* [BOD 95] em novembro de 1998. Atualmente ele se encontra instalado e operando segundo a topologia mostrada na Figura 2.8 (em Junho de 2001).

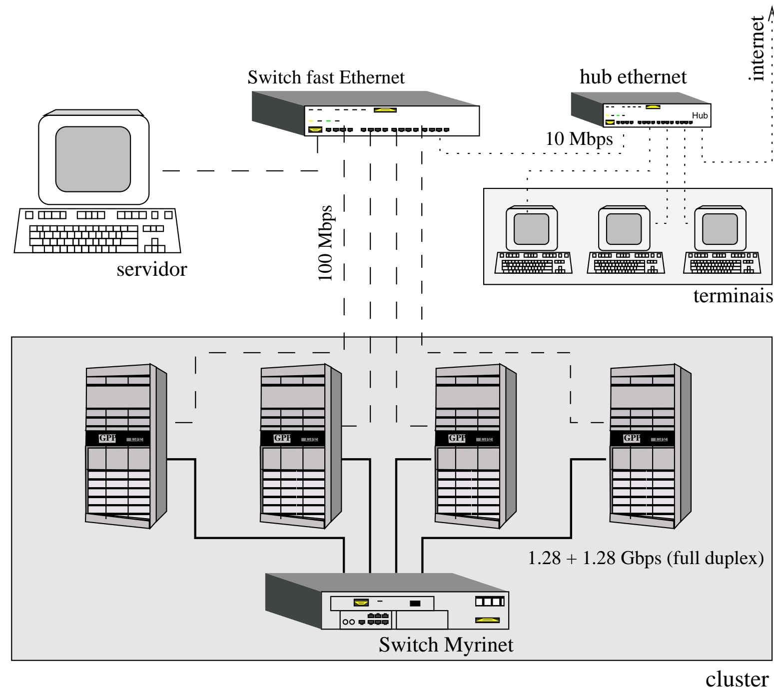

FIGURA 2.8 – Organização do *cluster Myrinet* do GPPD

A rede *Myrinet* está isolada fisicamente, estando disponível de forma exclusiva para comunicação remota entre as tarefas. O acesso externo é realizado através de uma rede *Fast Ethernet*, sendo esta utilizada também para serviços do Sistema Operacional.

Os nodos pertencentes ao *cluster* são compostos de quatro máquinas Pentium PRO com as características mostradas na Tabela 2.3:

TABELA 2.3 – Características do cluster *Myrinet*

Nodos	4
Processador	2 Pentium PRO 200 Mhz
Memória	128 Mb
Interconexão	*Fast Ethernet* e *Myrinet*
Sistema	Linux Debian 2.2 SMP
Disco	SCSI 2Gb

Inicialmente instalou-se o protocolo TCP/IP na rede *Myrinet* do *cluster*. Foram feitos testes de largura de banda utilizando PVM e MPI, de forma a comparar a rede *Fast Ethernet* com a *Myrinet*.

Pelos resultados mostrados na Figura 2.9, pode-se verificar que o desempenho sobre TCP não foi o esperado. Enquanto que na *Fast Ethernet* chegou-se a valores próximos de 9 Mbytes/s, na *Myrinet* o máximo foi 14 Mbytes/s, quase 10 vezes menos que o esperado.

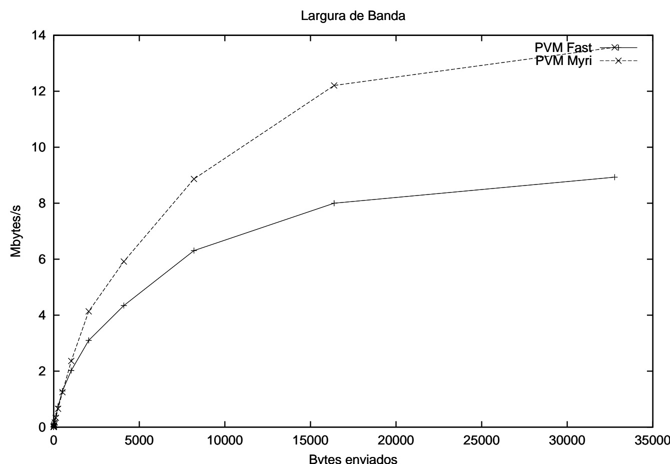

FIGURA 2.9 – Desempenho TCP no cluster *Myrinet*

O alto *overhead* imposto pela camada TCP do núcleo do Linux impossibilitou uma largura de banda maior. A latência, os custos com a pilha do sistema e as chamadas de sistema do Linux acabaram comprometendo seriamente o desempenho.

Na Figura 2.10 tem-se uma segunda avaliação de desempenho utilizando uma biblioteca exclusiva para a *Myrinet*. Valores próximos de 100 Mbytes/s foram obtidos [BAR 2000] neste caso, sendo o teste realizado com duas implementações diferentes do padrão MPI e uma implementação do DECK (descrito na Sessão 4.5), todas utilizando a biblioteca BIP (*Base Interface for Parallelizing*).

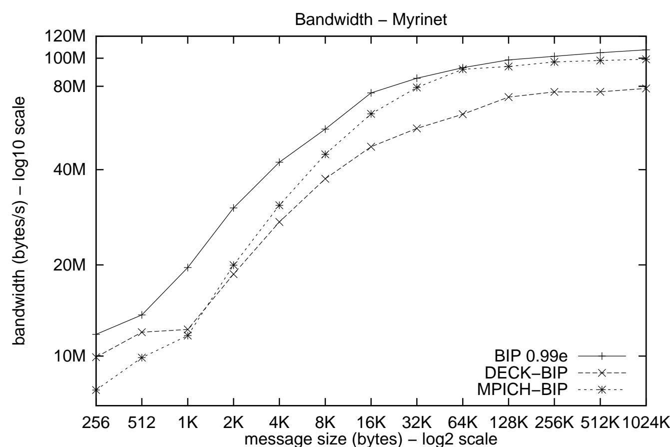

FIGURA 2.10 – Desempenho do *cluster Myrinet* com BIP

O *cluster* SCI do instituto conta com quatro máquinas dual Pentium III 500Mhz, organizadas conforme mostra a Figura 2.11. Os consoles e o servidor são os mesmos utilizados pelo *cluster Myrinet*. As características físicas de cada nodo podem ser vistas na Tabela 2.4.

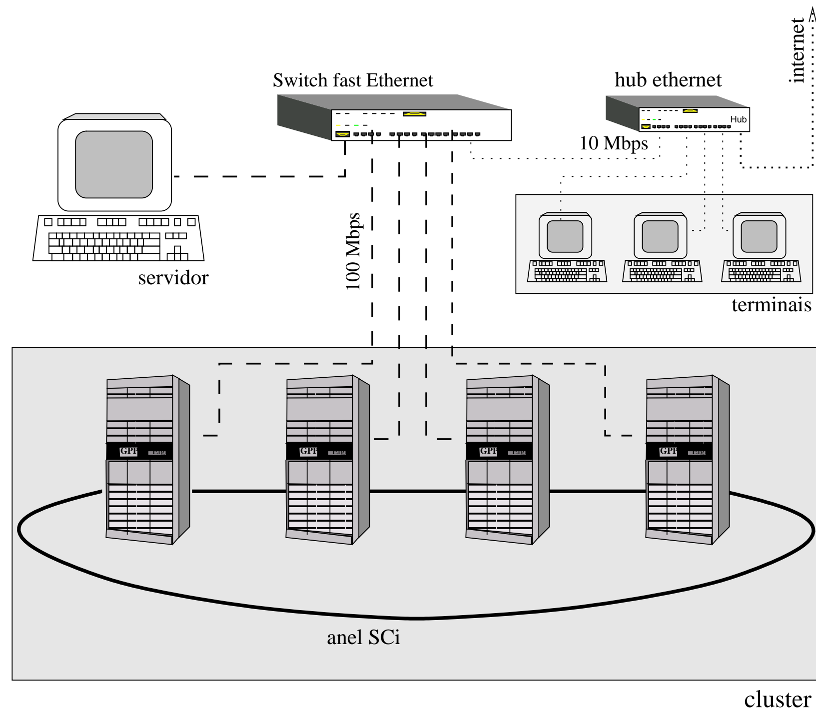

FIGURA 2.11 – Organização do *cluster SCI* do GPPD

TABELA 2.4 – Características do cluster SCI

Nodos	4
Processador	DUAL Pentium III 500Mhz
Memória	256 Mb
Interconexão	*Fast Ethernet* e SCI
Sistema	Linux Debian 2.2 SMP
Disco	SCSI 2Gb

Uma descrição mais detalhada, bem como avaliações de desempenho podem ser encontradas em [ÁVI 99].

2.4 Considerações Finais

Buscar aumento de desempenho pelo uso de vários processadores interligados e cooperando de forma organizada é, na verdade, uma idéia bastante antiga. A maneira mais natural de fazer isto é dispô-los em uma memória compartilhada, de forma que os dados entre eles possam ser facilmente compartilhados.

As bibliotecas de comunicação serviram muito bem a esta necessidade, facilitando a comunicação entre processadores interconectados através de uma rede. Perdeu-se, porém, a facilidade de programação, pois a troca de informações precisou ser explicitada através do uso de funções para enviar e receber mensagens. Outro limite era foi o custo envolvido na comunicação. Como esta comunicação é realizada tipicamente através de uma rede comum, de baixa largura de banda, as aplicações precisam ser divididas em partes maiores, caracterizando um processamento de gra-

nulosidade grossa.

A largura de banda das redes vem sendo constantemente melhorada, e hoje já temos a disposição equipamentos que superam até o próprio barramento de dados de um PC comum.Isto possibilitou voltar a trabalhar com aplicações de granulosidade mais fina e até mesmo implementar camadas em *software* para simular, de forma transparente à aplicação, uma memória global. Nasceram as máquinas conhecidas como DSM. Incluir esta simulação no próprio *hardware* da placa de rede foi o próximo passo, como é o caso do padrão SCI.

Com o *hardware* de máquinas *desktop* cada vez mais barato, e estas com processadores cada vez mais poderosos, a idéia de juntar várias máquinas e simular uma máquina paralela tem ganhado cada vez mais a atenção de pesquisadores.

Sistemas de alto desempenho já são hoje uma realidade disponível a um custo baixo. Formas mais eficientes de interconexão, uso de mais de um protocolo de comunicação, utilização mínima dos serviços de rede e principalmente uso da comunicação e alocação de tarefas de forma eficiente são fatores cada vez mais determinantes do desempenho destes agregados.

Não basta apenas ter vários processadores ligados de uma forma eficiente, se alguns ficarem sobrecarregados e outros ociosos. Sistemas que gerenciem de forma correta estes recursos são necessários para prover uma distribuição de carga satisfatória. É extremamente necessário pensar com seriedade em algoritmos de escalonamento para estes sistemas.

3 Escalonamento de Tarefas

Um escalonador é um algoritmo com a função de decidir qual tarefa será executada em qual processador. Na existência de vários processadores em um *cluster*, por exemplo, um escalonador eficiente pode tornar-se fundamental para o desempenho.

Casavant [CAS 88] descreveu o problema do escalonamento como uma relação entre consumidores, recursos e a política de alocação destes recursos.

Uma técnica especial de escalonamento, conhecida como "balanceamento de carga" [HAC 90, WIL 93], refere-se a forma de balancear uma carga (tarefas) entre os vários processadores existentes. Balanceamento de carga tem preocupações mais complexas do que apenas distribuir as tarefas entre os vários *nodos* existentes, pois procura fazê-lo de forma que a carga fique permanentemente equilibrada. Caso algum nodo esteja com carga menor ou maior que a dos demais, o algoritmo pode proceder a uma migração de tarefas, retirando uma tarefa do nodo sobrecarregado e colocando-a em outro, isto em plena execução, o que envolve um alto custo para restaurar no nodo destino a tarefa tal como ela estava quando foi retirada do nodo de origem.

3.1 Classificação dos escalonadores

Na Figura 3.1 é possível ver a forma como são divididos os tipos de escalonadores existentes.

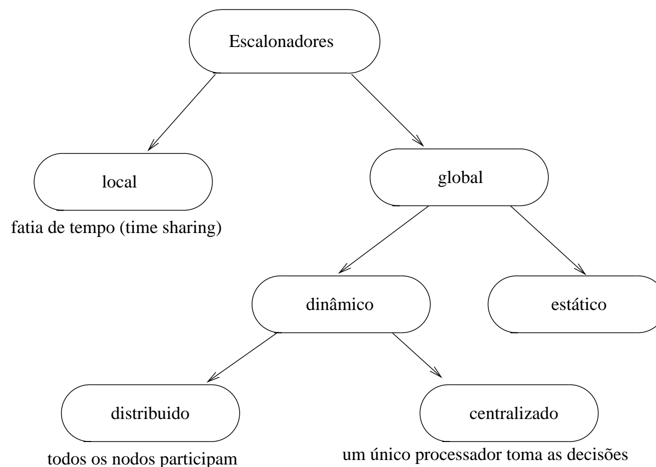

FIGURA 3.1 – Classificação dos escalonadores

Um escalonador pode ser classificado, inicialmente, como local ou global. Escalonador local é aquele existente nos Sistemas Operacionais para dividir porções de tempo de CPU entre os processos que estão na memória. Questões quanto a prioridade de determinados processos em relação aos demais devem ser considerados neste tipo de escalonamento. O escalonamento local, portanto, gerencia os recursos de uma única unidade computacional. Como exemplo, podemos citar o escalonamento de processos empregados no Unix que utiliza uma lista circular *Round Robin* [TAN 92].

Escalonadores globais, por sua vez, procuram alocar tarefas entre vários processadores. Em sistemas distribuídos, o escalonador global tem uma importância fundamental.

O escalonador global pode ser dividido em centralizado e distribuído. No centralizado, a decisão sobre qual processador receberá uma determinada tarefa é tomada em um único nodo e este fica responsável por manter um correto gerenciamento de todos os processadores. Já o escalonador distribuído divide esta tarefa entre todos.

Um escalonador precisa conhecer da forma mais eficiente possível o estado de cada processador, isto é, quanto aquele nodo específico é capaz de suportar. Temos então, uma outra classificação de algoritmos, definida pela forma como o escalonador busca e/ou mantém informações sobre os recursos que gerencia: escalonadores estáticos ou dinâmicos.

3.1.1 Escalonamento Estático

Neste escalonamento, as regras utilizadas são previamente conhecidas e não são alteradas. As informações sobre os processadores envolvidos são fornecidas pelo programador ou pesquisadas pelo algoritmo uma única vez. Basicamente, o escalonador apenas conhece a capacidade computacional de cada nodo, o que pode ser expresso por suas características de *hardware*. Não existe a possibilidade de um nodo informar ao escalonador que encontra-se em uma situação de sobrecarga. O escalonador tem a informação de que o *nodo* X tem mais poder de processamento que o *nodo* Y, logo está apto a receber as tarefas mais pesadas ou um maior número delas.

Um algoritmo bastante simples e rudimentar pode ser a distribuição na ordem de uma lista circular, usando como único critério distribuir as N tarefas entre os M processadores, sem preocupar-se com a carga de cada tarefa ou com o poder computacional de cada processador. Pode parecer ineficiente, mas bastante útil para escalonar tarefas de granulosidade fina. Se um escalonador, por exemplo, toma uma decisão após consumir 100 unidades de tempo, escolhendo o melhor processador para receber uma tarefa que executa também em 100 unidades de tempo, o algoritmo tornou-se um grande consumidor de recursos. Seria mais eficiente colocar esta tarefa até mesmo em um processador não tão adequado, mas tomar esta decisão de forma rápida.

Atualmente, porém, tais métodos mostram-se inapropriados na maioria dos casos, pois:

- em sistemas agregados, todos os *nodos* envolvidos tem características de *hardware* iguais, sendo que cada *nodo* oferece os mesmos recursos computacionais ao escalonador.

- existem muitas variações na carga de cada tarefa e distribuí-las de forma simétrica entre os nodos não significa, necessariamente, obter o melhor desempenho;

- uma máquina pode não estar a disposição de uma única aplicação, então, tarefas alocadas por outras aplicações, ou mesmo por outros usuários, podem interferir no desempenho da aplicação e o algoritmo não teria meios de detectar isso.

Em algumas situações, é mais desejável que o escalonador esteja com informações atualizadas sobre os *nodos*. Quando isto acontece, tem-se o chamado "Escalonamento Dinâmico".

3.1.2 Escalonamento Dinâmico

Neste tipo de escalonamento há uma constante análise das características dos nodos, ou seja, o escalonador tem a possibilidade de "saber" a capacidade de cada nodo em qualquer momento, podendo decidir sobre qual deles está mais apto a receber uma tarefa. Para um escalonador dinâmico a quantidade de tarefas executando no nodo, mesmo tarefas de outras aplicações, e a carga de cada uma são informações relevantes. Escalonadores deste tipo, ao contrário dos estáticos, possuem um custo maior, pois eles possuem mais comunicação. Entretanto, possuem a capacidade de tomar decisões mais precisas.

Para implementar este algoritmo é necessário obter informações atualizadas sobre cada nodo com uma certa freqüência. Uma forma é pesquisar todos os nodos existentes no sistema sempre que uma nova alocação se fizer necessária. Isto envolveria consultar cada *nodo* em separado, pedindo que este lhes remeta informações sobre seu estado de carga.

Uma forma se fazer isso é com a existência de pequenos processos em execução em cada máquina, com o único propósito de "espionar", remetendo suas conclusões ao algoritmo de escalonamento. Estes processos são chamados de "processos espiões" [ELR 94].

Como as informações de cada nodo deve ser do conhecimento de todos os outros, isto pode gerar um grande custo com nodos enviando mensagens de estado para todos os outros de forma generalizada. Uma solução para este problema é a classificação dos nodos envolvidos em receptores e fornecedores [WIL 93, KRU 94]. Neste caso, não há a necessidade de cada nodo conhecer o estado de todos os demais e as mensagens trocadas são as mínimas.

O objetivo desta solução é dispor de parâmetros de carga que permitam a um determinado nodo se enquadrar como receptor ou doador de carga. Se ele for receptor, pode anunciar seu estado aos demais, possibilitando que todos saibam de sua condição de ociosidade. Se ele for doador, pode, da mesma forma, divulgar-se como tal, excluindo-se do processo de escalonamento. Se o sistema provê migração de processos, a interação entre receptores e doadores pode ser muito produtiva. Da mesma forma, se o nodo não se encaixar em nenhum caso, pode permanecer com suas tarefas sem importar-se com o estado dos demais, apenas informando quando trocar de estado. A Figura 3.2 exemplifica estas situações.

Esta técnica é a principal base para os mais modernos sistemas de escalonamento distribuído com migração de processos. Alguns estudos sugerem que a melhor forma é a do escalonamento iniciado pelo *sender*, ou seja, onde processadores sobrecarregados se encarregam de procurar processadores com pouca carga. Outra possibilidade é o iniciado pelo *receiver*, onde nodos com pouca carga oferecem-se para receber novas tarefas. Alguns algoritmos funcionam de forma parecida como em um leilão, onde quem tem a melhor oferta ganha.

Um cuidado com estes algoritmos deve ser tomado no sentido de que uma tarefa não fique "pulando" de nodo para nodo, caso esta tarefa tenha carga alta. Uma tarefa deste tipo tem a capacidade de transformar qualquer nodo receptor em um nodo doador, fazendo com que este tente livrar-se da carga excedente, e

assim sucessivamente, causando a chamada “postergação indefinida”. Uma definição correta das políticas de escalonamento pode facilmente evitar este tipo de problema.

Escalonadores dinâmicos possuem um custo extra, pela necessidade de comunicação. Caso o algoritmo seja centralizado, haverá a possibilidade de um gargalo de comunicação com todos os nodos enviando mensagens para o único escalonador. Caso seja distribuído, haverá uma intensa comunicação de todos com todos. A comunicação intensa é um problema que pode tornar-se relevante no caso de todos os nodos estarem sobrecarregados.

3.1.3 Escalonamento Adaptativo

O escalonamento adaptativo é uma variação do dinâmico. Basicamente, o escalonador deste tipo é capaz de se auto avaliar, calculando sua interferência no sistema ao qual se propõe gerenciar. Se considerar que está consumindo muita CPU para este trabalho, pode reduzir a rigorosidade dos testes ou características consideradas em cada nodo. Por exemplo, pode passar a considerar apenas a quantidade de tarefas existentes em cada nodo como fator de decisão. Se, de outro lado, chegar a conclusão que tomou decisões erradas, pode aumentar a rigorosidade com que avalia a carga dos nodos.

Um sistema adaptativo, portanto, é aquele que possui a capacidade de se adaptar às cargas das tarefas que gerencia, procurando usar o melhor de cada técnica conhecida e no momento certo. Pode começar sendo do tipo iniciado pelo *sender* e depois tornar-se iniciado pelo *receiver*. Pode ser originalmente dinâmico, mas depois, em caso de extrema sobrecarga, deixar de coletar informações sobre os nodos, tornando-se estático, e assim por diante.

3.2 Políticas de Escalonamento

Em [SIN 94] encontra-se a definição de escalonador dinâmico como sendo um sistema que possui um conjunto de políticas que regem o seu comportamento. São ao todo quatro políticas: de transferência, de seleção, de localização e de informação, conforme pode ser visualizado na tabela 3.1.

A política de transferência determina se um processador está ou não sobrecar-

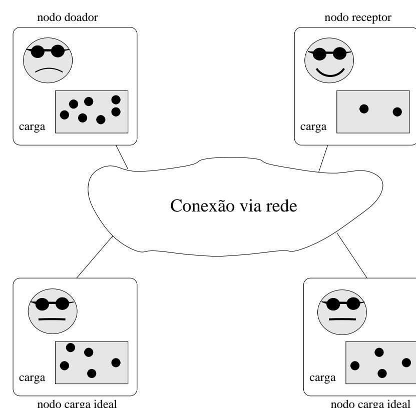

FIGURA 3.2 – Uso dos espiões

regado. Caso a carga do nodo ultrapassar um limite, ele pode considerar-se como sobrecarregado, podendo requisitar ajuda aos demais nodos, caso exista migração de carga. De igual forma, estando ele abaixo de sua capacidade, pode considerar-se apto a receber tarefas, ofertando processamento aos demais, de acordo com o algoritmo em execução.

Esta política deve ser bem definida, para evitar, por exemplo, que uma tarefa com carga suficientemente altafique "pulando" de nodo em nodo sem ser executada. Algumas técnicas podem ser utilizadas para evitar este problema:

- Considerar três casos possíveis: processadores com carga ideal, com pouca carga e com muita carga [KRU 94], como já exemplificado na Figura 3.2. Os processadores com carga ideal não estão aptos nem a ofertar tarefas e nem a recebê-las. Ao enviar uma tarefa a um nodo com pouca carga, pode-se estimar seu impacto antes, não enviando caso o processador torne-se sobrecarregado.

- Um contador que indica quantas vezes uma tarefa migrou, obrigando o processador que o receber com o contador no limite a executá-la de qualquer maneira.

Caso um processador esteja sobrecarregado, definido pela política de transferência, a política de seleção decide qual será a tarefa escolhida para migrar. A escolha pode ser tanto de uma tarefa que ainda não teve sua execução iniciada (candidata mais favorável, pela inexistência de contexto de execução) como de uma tarefa em plena execução (alternativa mais complexa e de custo maior). A política pode determinar que a tarefa iniciada a menos tempo, ou seja, aquela que está apenas a alguns ciclos em execução, seja a escolhida, apostando ter um custo menor com migração de contexto (poucas comunicações estabelecidas, poucos arquivos abertos e manipulados, etc). Outra técnica é escolher a tarefa que causou a sobrecarga, no caso, a última recebida. Esta técnica tem muitas vantagens, seja pelo fato de não ter iniciado sua execução ou pela pouca possibilidade de outras tarefas do nodo terem realizado comunicação com a tarefa.

Depois de escolhida a tarefa a ser migrada pela política de seleção, há a necessidade de se escolher o novo processador que receberá a migração, tarefa de responsabilidade da política de localização. O método mais empregado é o aleatório, mesmo que ele possa escolher um processador com carga ainda maior que a do atual. Isto pode ser resolvido, testando o processador antes e refazendo a escolha, caso necessário. Algoritmos que funcionam como uma espécie de leilão, onde quem fizer uma oferta maior ganha a carga, também podem ser utilizados.

Em um sistema com escalonamento distribuído, os processadores precisam conhecer o estado dos demais, ou pelo menos parte deles. A forma como estas

TABELA 3.1 – Políticas de escalonamento

Política	Descrição
transferência	classifica um processador como sobrecarregado ou não
seleção	decide qual tarefa deve mudar de processador
localização	escolhe quem receberá a tarefa escolhida
informação	de que forma as informações sobre carga são compartilhadas

informações são compartilhadas é definida pela política de informação. Pode-se, por exemplo:

- eleger um processador central que receberá informações de todos os demais, fornecendo-as a quem requisitar. Com um número grande de processadores e troca de mensagens, o desempenho pode ficar comprometido.
- através de *broadcast*, divulgando as informações de cada um. Pode ser inviável, pela quantidade excessiva de mensagens trocadas em uma rede cuja prioridade deve ser a de servir à comunicação entre tarefas.
- definir que cada um possui tão somente informações sobre si próprio e de ninguém mais, podendo fornecê-la a qualquer um que as requisite. Evita a troca de mensagens excessivas, mas torna a migração de tarefas mais burocrática, pois a política de localização fica mais complexa, precisando consultar cada nodo sobre sua carga antes de escolher algum.
- todos os processadores sabem o estado de todos os outros que lhe interessam, ou seja, informações sobre carga somente serão trocadas quando houver necessidade e conveniência. É desnecessário um nodo enviar um *broadcast* a todos os demais, informando sobre sua nova carga, se esta não o fez mudar de estado. Mas se ele era considerado por todos como um receptor de tarefas em potencial e teve sua carga elevada, tornando-o sobrecarregado, é importante que esta nova situação seja comunicada aos demais. Da mesma forma, um nodo que se encontra com carga estável, definido pela política de transferência, pode optar por não mais receber mensagens dos demais com informações sobre carga. Se a comunicação for em grupo, pode excluir-se do grupo.

Um outro fator muito importante nos escalonadores distribuídos é a forma como a carga de um processador é avaliada.

3.3 Avaliando a Carga

Uma das primeiras coisas que o escalonador precisa conhecer é a capacidade física de cada nodo. É uma informação de extrema relevância em sistemas com processadores diversos, de características físicas diferentes.

Isto pode ser configurado uma única vez na instalação do algoritmo e não precisa ser atualizado até que algum nodo mude suas características de *hardware*. Além disto, estas informações são irrelevantes em sistemas de caracterísiticas iguais, conhecidos como "sistemas homogêneos".

Geralmente o que se deseja obter é a carga do nodo, ou seja, quanto ainda está disponível de sua capacidade computacional. Estudos demonstram que uma estação de trabalho típica tem sua capacidade de processamento ociosa 91% do tempo [EFE 95]. Este é um bom motivo para procurar aproveitar esta ociosidade na resolução de algum problema distribuído [TAN 96].

Uma maneira bem simples e bastante empregada é simplesmente usar o tamanho da fila de execução como medida. Quanto maior o número de processos existentes na fila, maior será a disputa pelo processador o que aumenta o tempo de espera na fila por parte dos processos. No entanto, nos sistemas atuais existe

o conceito de prioridade, onde processos de alta prioridade ganham mais CPU que outros, fazendo com que uma lista pequena, com alguns processos de alta prioridade, produza um tempo de espera maior para um processo de baixa prioridade. O tempo médio de espera na fila, neste caso, torna-se mais interessante como parâmetros de carga.

Neste caso calcula-se quanto cada processo esperou, em média, sem receber sua fatia de tempo da CPU. Quanto maior este tempo, mais a máquina teve problemas para dar conta de sua carga, logo, ela pode estar sobrecarregada. Mesmo que esta técnica se pareça muito com a anterior, pois a média será maior na proporção direta do número de processos existentes, ela é mais completa, pois leva em consideração processos que recebem a CPU e não a utilizam plenamente, talvez por terem feito várias requisições de I/O. Como, neste caso, o processo não é considerado como na fila de espera, uma grande quantidade destes processos não aumenta de forma significativa o tempo médio de espera.

O tempo de espera na fila de execução, por causa de sua simplicidade e baixo *overhead*, é muitíssimo usado para avaliar a carga do nodo.

Muitas informações sobre o estado atual de um processador já são calculadas e mantidas pelo próprio Sistema Operacional. Isto é necessário para que o mesmo possa fazer a correta distribuição das fatias de tempo entre seus processos, no escalonamento local.

Como exemplo, o Unix possui um sistema baseado em prioridades [CLE 87], onde cada processo possui uma prioridade básica, inerente ao processo, que pode variar desde a mais baixa até a mais alta prioridade. Processos de maior prioridade ganham a CPU primeiro, enquanto que os processos de prioridade menor são deixados de lado. A cada intervalo de tempo, um segundo algoritmo entra em ação e calcula novamente a prioridade de cada processo existente, diminuindo a prioridade de quem recebeu muita CPU e aumentando para aquele que foi discriminado. Processos que estavam bloqueados à espera de I/O, retornam à fila de prontos com a prioridade máxima, pois ficaram muito tempo sem receber ciclos de CPU [TAN 92].

Por causa destas características, o Unix já mantém contabilizações sobre o uso da CPU, como a média de carga, chamada de *load average*. Esta medida expressa a quantidade de processadores, iguais ao existente no sistema atual, que seriam necessários para executar todos os processos com o tempo de resposta e espera na fila de preempção mais eficazes. Se o nodo em questão tiver apenas um processador, valores abaixo de 1 indicam que o nodo está conseguindo manter sua carga. Valores, de outro modo, maiores que o número de processadores disponíveis significam um nodo sobrecarregado. A Tabela 3.2 mostra algumas situações possíveis, onde pode-se observar que o valor lido pelo *load average* deve ser interpretado considerando:

- a quantidade de processadores existentes na máquina;
- os valores médios dos últimos 5 e 15 minutos fornecidos também pelo Sistema Operacional;
- o número de processos existentes, pois mais processos podem significar uma tendência do nodo em ficar ocioso;

Para uma melhor avaliação são fornecidos três valores de *load average*: a média do último minuto, dos últimos cinco minutos e dos últimos quinze minutos.

TABELA 3.2 – Valores de *Load Average*

Proc	Load Avg	5 min	Observação
1	0,50	0,30	Nodo tem estado ocioso
1	0,50	1,90	Nodo ocioso agora, mas estava com carga alta
2	1,20	0,30	Nodo está com carga, mas ainda um pouco ocioso
2	3,40	9,80	Nodo sobrecarregado. Muita carga.
2	0,40	8,50	Nodo ocioso agora, mas com muita carga recentemente.

As informações de *load average* podem estar, ainda, disponíveis através de um arquivo simbólico montado em */proc*, desde que devidamente configurado. Se ao compilar o *kernel* o usuário habilitar a geração do diretório simbólico *proc* (o que é desejável), muitas informações úteis a qualquer escalonador ou espião estarão disponíveis, como informações sobre o tipo de processador, velocidade, memória, bem como o próprio *load average.*

Processos espiões podem coletar estas informações e trocá-las com os outros nodos ou simplesmente enviar ao nodo encarregado de fazer o escalonamento. Este espião pode também fazer outras coisas, como analisar o impacto no desempenho do sistema causado pela última tarefa enviada pelo escalonador.

Seria também bastante útil se os algoritmos de escalonamento fossem capazes de conhecer cada tarefa, que tipo de tarefa é, com que intensidade se comunica, etc.

3.3.1 A Carga de uma Tarefa

Uma característica muito difícil de se conhecer mas extremamente útil, se for adquirida, é o custo computacional da carga a ser escalonada. Se o escalonador tiver condições de diferenciar as tarefas pela sua carga computacional, pode tomar decisões bem mais inteligentes, como alocar várias tarefas menores a um nodo e apenas uma grande a outro, de forma que ambos estejam basicamente com a mesma carga, mesmo que um tenha efetivamente mais tarefas que o outro. Este conhecimento também será muito útil para a migração de tarefas. Seria desperdício migrar uma tarefa que está prestes a terminar, visto que os custos de migração são muito altos.

Obter tais informações não é algo trivial, pois tarefas de código pequeno não são, necessariamente, menores. Pode-se obtê-las de algumas formas:

- pedir ao programador da aplicação: mesmo que esta tarefa seja difícil para o programador e não desejável, pois perde-se a transparência fazendo com que ele tome decisões sobre o escalonamento, um simples enquadramento de cada tarefa em um tipo específico já pode ser de grande ajuda. Pode-se, por exemplo, pedir ao usuário que classifique suas aplicações como leves ou pesadas, ou como tarefas com muita ou pouca comunicação.

- com base em execuções anteriores: se esta não for a primeira vez que um mesmo conjunto de tarefas precisa ser escalonado, o escalonador pode ter um conhecimento de um comportamento anterior e saber estimar a carga de cada tarefa.

- compiladores específicos: linguagens de programação desenvolvidas para este propósito podem incorporar ao código informações sobre a quantidade de laços existentes, o tamanho de cada um, etc. Um exemplo desta técnica é descrito em [CAS 99].

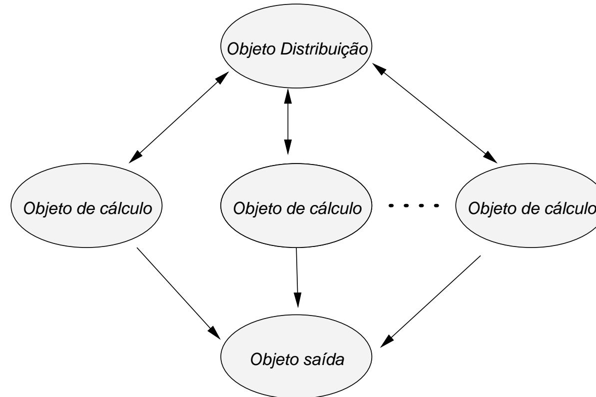

FIGURA 3.3 – Exemplo de um fractal distribuído.

Como exemplo, pode-se citar uma aplicação típica de geração de fractais. No exemplo da Figura 3.3, temos uma aplicação distribuída para isso. Neste caso, a distribuição foi feita da seguinte forma:

- existe um processo responsável em distribuir partes da imagem a ser calculada entre os vários processos cálculo. Este processo recebe o nome de “distribuição”.
- cada processo “cálculo” calcula a parte recebida e envia o resultado para o objeto “saída”, que exibe os resultados na tela;
- existem n processos cálculos iguais e com a mesma carga (mesmo número de pontos para calcular);
- o número de pontos da imagem total é fornecido pelo usuário na linha de comando.

O que pode-se verificar com este exemplo é que cada processo cálculo tem um “peso”, uma carga, diretamente proporcional a quantidade de pontos que receberá para calcular. Um escalonador, de posse desta informação, pode, por exemplo, evitar colocar dois cálculos em um mesmo nodo. Se esta aplicação estiver executando em um ambiente com oito nodos e sete processos de cálculo, tem-se um total de 10 processos a serem escalonados para oito nodos. Como os processos de cálculo fazem muito uso da CPU, uma solução coerente seria colocar os 7 processos de cálculo nos 7 nodos mais rápidos, e os outros 3 processos (principal, distribuição e saída) no nodo de menor desempenho. Porém, esta não é uma solução muito racional do ponto de vista de um escalonador que não consegue diferenciar a carga de cada processo. O mais coerente, para o algoritmo, seria colocar mais de um processo nos nodos mais eficientes. Poderia ocorrer o caso de um único nodo com dois processos cálculo, o que não é desejável.

Avaliar a carga computacional de uma tarefa é bastante relevante, ainda mais quando conta-se com máquinas homogêneas, dispostas em um *cluster*.

3.3.2 Carga em *clusters*

Em um sistema agregado, algumas considerações devem ser feitas quanto ao escalonamento:

- os nodos são de uso exclusivo: a idéia básica de um *cluster* é que seja totalmente isolado da rede e demais usuários, ou seja, dentro da rede que interliga os nodos do *cluster*, só existem processos e comunicação referentes à aplicação distribuída em execução;
- máquinas homogêneas e simétricas: cada nodo de um agregado possui *hardware* desde o início montado para esta finalidade. Além de serem homogêneas, isto é, mesmo processador, mesmo Sistema Operacional, mesmos recursos físicos, os nodos também são simétricos, no sentido de serem iguais em quantidade de memória, velocidade do processador, velocidade e tamanho dos discos rígidos, etc.
- maior largura de banda, possibilitando um paralelismo de grão menor;
- várias placas de rede: o uso de várias placas de rede para fazer a comunicação, inclusive com padrões e protocolos diferentes;

Na programação orientada a objetos distribuídos, como no caso do DPC++ (Capítulo 4), a carga de um nodo não será proporcional aos objetos em execução, mas sim às requisições de métodos realizadas por estes. Se estas requisições forem implementadas como chamadas de funções ou mesmo *threads*, fica difícil para o escalonador controlar a carga. Neste caso, na medida que a granulosidade for diminuindo, passando de simples processos (a mais alta) para funções ou mesmo processos leves (*threads*), mais complicado fica o algoritmo de escalonamento.

Algoritmos de escalonamento podem ser também empregados para escolher a rede mais ociosa para atender a comunicação entre dois nodos específicos, caso os nodos possuam mais de uma placa de rede. Este escalonamento pode, inclusive, usar a rede de mais alto desempenho para a comunicação entre produtores e consumidores, e a rede de menor largura de banda para as demais, como as de gerenciamento de tarefas ou aquelas relativas ao funcionamento do cluster, como NFS (*Network File System*).

3.4 Exemplos de Escalonamento

Muitos sistemas usam algoritmos de escalonamento de carga. Desde os algoritmos mais simples, como o estático, até os mais complexos, envolvendo migração em um sistema distribuído.

As bibliotecas de comunicação existentes, como o PVM e as implementações de MPI, provêm poucos recursos de escalonamento de tarefas. No caso do PVM, o programador pode, caso deseje, forçar a carga de um processo em uma determinada máquina. Se ele deixar que a própria biblioteca escolha o nodo onde deve ser criado o processo, ele irá escolher aquele que tiver o menor número de processos em execução, o que não significa uma carga maior. Trata-se, então, de uma simples lista circular de processadores, onde os M processos serão divididos entre os N processadores. Este

escalonador estático pode cometer erros, quando a aplicação possui dois processos com pouca carga, tipicamente *I/O-bound*, e os outros de carga alta.

Algumas implementações de PVM utilizando migração de processos foram desenvolvidas como a apresentada em [BUY 99]. Neste sistema, a migração de uma tarefa envolve quatro passos:

- o escalonador notifica o *daemon* responsável de que um de seus processos precisa ser migrado;
- este, por sua vez, envia uma mensagem para todos os demais *daemons* existentes na máquina virtual, sinalizando que uma migração ocorrerá, e que nenhuma comunicação deve ser realizada com o nodo em questão até que a migração seja concluída;
- depois de receber uma confirmação de cada nodo, sinalizando sua ciência, o contexto atual do processo é transferido para seu novo local e um novo processo é criado;
- por fim, o novo processo conecta-se ao *daemon* do novo nodo e este notifica todos os demais sobre o novo endereço (identificador) do processo, para que possam comunicar-se com ele.

A migração é assíncrona, podendo ser feita a qualquer momento, envolvendo apenas o processo que será migrado e aqueles que mantém comunicação com ele.

Quanto ao MPI, por ser SPMD, todos os processadores estão executando somente um processo, sendo que este é o mesmo para todos. Então não existe nenhum momento em que deva-se escolher em qual processador um processo deve ser criado. Neste caso as possíveis diferenças entre cargas computacionais serão resultado da distribuição de dados entregues para processamento, e esta divisão faz parte da aplicação e da forma como ela foi organizada logicamente pelo programador.

3.4.1 Usando a ociosidade das estações

Os índices de ociosidade de uma estação de trabalho podem ser bastante elevados em situação normal de uso e até mesmo de aproximadamente 100% em horários específicos, como durante a noite e finais de semana. Aproveitar esta ociosidade para ajudar na realização de determinada tarefa pode tornar-se interessante.

Um exemplo é a proposta Batrun [TAN 96], que monitora máquinas aproveitando sua ociosidade para realizar determinada tarefa. Um *daemon* permanece executando em cada máquina que compõe as chamadas células do Batrun. Cada célula funciona como se pertencesse à um mesmo escalonador local, com uma máquina responsável pelo gerenciamento. O sistema observa as atividades do *mouse* e do teclado para determinar a ociosidade da máquina.

Caso ela não esteja em uso interativo por nenhum usuário no momento, o sistema entra em ação, recebendo uma tarefa para executar. Caso ela volte a ficar inativa, com a presença de um usuário, a tarefa é congelada através de *checkpoints* e seu estado pode ser restaurado em qualquer outra máquina da célula.

A comunicação entre as máquinas é realizada através de RPC (*Remote Procedure Call*), mesmo sistema utilizado pelo Unix para prover sistemas de arquivos em rede.

Existem também alguns programas disponíveis na Internet que podem ser instalados como protetores de tela, sendo ativados quando o sistema estiver sem uso. Alguns exemplos são programas com o propósito de analisar o sequenciamento de DNA, quebrar por força bruta algum algoritmo de criptografia, entre outros. Eles tem a possibilidade de transformar cada microcomputador de usuário comum em um nodo, mesmo que pouco representativo, participante de uma imensa rede. Até mesmo a análise de ondas de rádio com o objetivo de descobrir vidas em outros planetas faz uso desta técnica.

3.5 Considerações Finais

A tarefa de escalonar tarefas entre vários processadores envolve muitos parâmetros. O escalonador pode considerar, por exemplo, que não é o único a alocar recursos para aquele nodo, pela existência de um sistema multiusuário. Assim, mesmo que ele não tenha designado cargas altas para um determinado processador, este pode encontrar-se em uma situação de sobrecarga.

Em um conjunto de máquinas compartilhadas por vários usuários, uma abordagem à migração de carga pode ser justificável. Isto porque o processador pode surpreender o algoritmo de escalonamento, mostrando-se subitamente sobrecarregado.

4 DPC++

A linguagem de programação DPC++ (*Distributed Processing in C++*) foi desenvolvida no Instituto de Informática da Universidade Federal do Rio Grande do Sul (UFRGS) com o objetivo de facilitar a programação distribuída através do conceito de objetos distribuídos [CAV 93, CAV 94]. Objetos distribuídos são objetos que podem ser criados em uma máquina diferente da que está executando a aplicação. O DPC++ baseia-se na linguagem C++, tendo a mesma sintaxe e incluindo mais algumas palavras reservadas. Trata-se, na verdade, de um pré-compilador que gera código em C++ que depois sera compilado pelo compilador C++.

O uso de uma linguagem orientada a objetos como base para uma linguagem distribuída deve-se ao fato de existirem características semelhantes entre ambas:

- as trocas de parâmetros entre objetos podem facilmente serem transformadas em trocas de mensagens.
- um objeto tem seu código encapsulado, ou seja, não é possível, dentro do conceito de uma linguagem orientada a objetos pura, acessar uma variável interna de um objeto. Este conceito muito se assemelha à distribuição de memória existente em máquinas distribuídas.
- um objeto não depende de outro para executar, isto é, não depende da estrutura interna de outro objeto. Qualquer acesso a funções dentro do objetos somente são permitidas através de métodos, de forma semelhante a um processamento distribuído usando RPC.

Em uma linguagem de programação orientada a objetos ocorre a troca de informações entre um objeto criado e o objeto (ou aplicação) que o criou. Esta troca de informações é realizada pelos parâmetros de criação do objeto. Da mesma forma, quando um método deste objeto é executado, uma troca de informações, através dos parâmetros da chamada de método e conseqüente retorno de valor, é realizada. Esta troca de informações é feita através da pilha do Sistema Operacional.

A diferença entre uma linguagem C++ típica e o DPC++ é que esta troca de informações podem ser realizados através de troca de mensagens, permitindo que objetos executem em máquinas distintas, conectadas através de uma rede. É na etapa de pré-compilação que esta troca de mensagens é inserida, seja através de alguma biblioteca específica, como o PVM e o MPI, ou mesmo através de *sockets*.

Na Figura 4.1 encontra-se uma visão geral de uma aplicação DPC++ com as invocações de métodos realizadas através de trocas de mensagens.

4.1 Estrutura do DPC++

Na versão atual do DPC++, o programador precisa criar um arquivo especial chamado de descritor da aplicação. Neste arquivo encontram-se informações sobre quantas classes distribuídas existem na aplicação, quais as máquinas que serão utilizadas para criar os objetos distribuídos e qual máquina deve ser usada para disparar o objeto Diretório.

Quando o pre-compilador for executado, este é o primeiro arquivo que ele processa, procurando e processando os demais arquivos (os que contém o código de uma classe distribuída).

Como os objetos distribuídos que o programador criou podem executar em outra máquina de mesma arquitetura, o DPC++ encapsula seu código dentro de um novo executável para que o mesmo possa ser remotamente iniciado em qualquer máquina do ambiente distribuído. Estes novos executáveis (processos) tem rotinas de recebimento e envio de mensagens para possibilitar a comunicação entre a rede.

Uma aplicação DPC++, depois de compilada, será composta, internamente, das seguintes partes:

- objetos locais definidos pelo programador;
- objetos distribuídos definidos também pelo programador;
- objetos locais chamados de procuradores;
- programa principal responsável por disparar toda a aplicação.
- um objeto distribuído especial, chamado de objeto Diretório;

Os objetos locais, definidos pelo programador, em nada diferem de um objeto normal. Para permitir a ligação de um objeto local com um objeto distribuído, objetos locais chamados de "procuradores" são inseridos pelo DPC++ no código com o objetivo de manter o endereço do objeto distribuído que representam e converter a passagem de parâmetros da invocação de métodos em mensagens. Da mesma forma o objeto procurador é responsável, também, por devolver ao objeto local, na forma de retorno de método, o resultado de uma invocação recebida através de uma mensagem do objeto distribuído. A Figura 4.2 exemplifica a participação dos objetos procuradores em uma execução de método.

O objeto Diretório tem uma importância muito especial neste trabalho e será estudado a seguir.

4.2 Objeto Diretório

O compilador da linguagem inclui dentro de toda aplicação DPC++ um objeto distribuído especial, chamado de objeto Diretório. Este objeto fica encarregado de

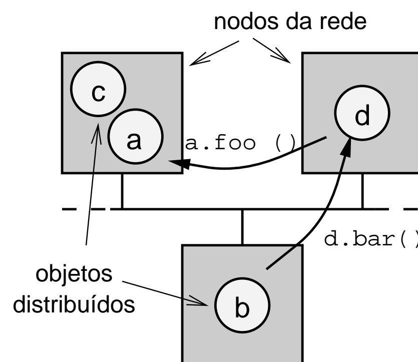

FIGURA 4.1 – Visão Geral do DPC++

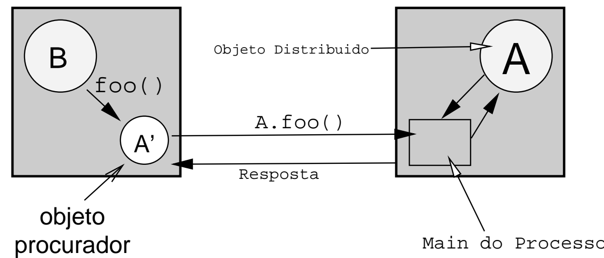

FIGURA 4.2 – Invocação de um Método em DPC++

gerenciar toda a criação e remoção dos demais objetos distribuídos existentes no sistema.

As funções do objeto Diretório são:

- criar objetos distribuídos;
- manter tabelas de endereços que possibilitem localizar um objeto distribuído dentro do ambiente e;
- manter o sistema confiável através de tolerância a falhas.

Toda a criação de objetos distribuídos no ambiente é feita pelo objeto Diretório. Como exemplificado na Figura 4.3, quando um objeto qualquer, seja ele local ou distribuído, solicita a criação de um novo objeto distribuído, esta criação nada mais é do que a criação de um objeto local chamado de procurador.

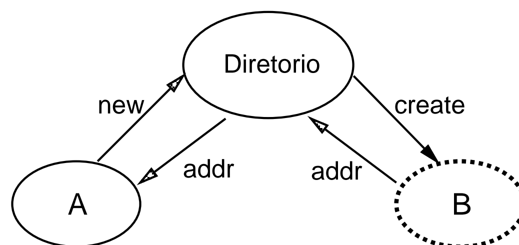

FIGURA 4.3 – Participação do objeto Diretório no DPC++

Este objeto procurador prepara uma mensagem com uma solicitção ao objeto Diretório de criação, em alguma máquina, do objeto distribuído que o procurador representa. O objeto Diretório executará esta tarefa e retornará o endereço do objeto distribuído criado.

Entende-se aqui por endereço uma ou mais informações que possibilitem ao procurador comunicar-se diretamente com seu objeto distribuído. Se o compilador estiver usando PVM, por exemplo, este endereço pode ser o número *pvmid* do processo; se estiver usando *sockets* esta informação será o nome da máquina mais o endereço da porta, e assim por diante.

O objeto Diretório é o único, em todo o sistema, que tem uma visão "global" dos objetos, ou seja, enquanto cada objeto conhece tão somente aqueles criados por ele, o objeto Diretório conhece todos. É por isso, também, que o objeto Diretório é responsável pela tolerância a falhas.

4.3 Tolerância a Falhas no DPC++

Sendo o DPC++ uma linguagem de programação distribuída, a possibilidade de ocorrerem falhas é maior. Por isso, tolerância a falhas foi adicionado ao DPC++ para tratar especificamente dois tipos de falhas: a falha em algum objeto distribuído e a falha do próprio objeto Diretório.

Para o primeiro problema, que caracteriza-se pelo fato de um objeto distribuído não responder à requisições, fez-se uso de *checkpoints*. O sistema de tolerância a falhas para este caso limita-se a detectar e corrigir falhas de *crash*.

O modelo de *checkpoints* distribuídos [PIL 97] é baseado no modelo global de *checkpoint*, onde periodicamente o sistema todo pára sua execução e uma imagem de cada processo é salva em um arquivo. Quando uma falha for detectada, basta restaurar o último *checkpoint* feito e o sistema continua. No caso do DPC++ um objeto só conhece objetos que ele criou e/ou com os quais mantém comunicação, já que somente o objeto Diretório conhece o sistema completo. Então o estado global é relativo apenas aos objetos que trocaram mensagens.

Desta forma, o sistema implementado faz *checkpoints* apenas entre os dois processos envolvidos em uma troca de mensagem e somente quando uma mensagem for enviada.

O sistema implementado na linguagem pode ser resumido da seguinte forma:

- sempre que dois objetos trocam mensagens, um *checkpoint* é criado em ambos.
- o objeto que requisitou um método ou que está esperando alguma mensagem de outro, ficará aguardando por um tempo pré-determinado.
- após este tempo, este objeto presume que uma falha pode ter ocorrido, o que nem sempre é verdade, pois o objeto para o qual atribui-se a falha, pode estar simplesmente executando uma tarefa demorada.
- para verificar se houve falha, uma mensagem especial é enviada ao objeto Diretório perguntando sobre a situação do referido objeto.
- o objeto Diretório poderá informar que o objeto está em execução, ou que houve uma falha. No caso de falha, o objeto Diretório irá criar um novo objeto em outra máquina e restaurar o último *checkpoint* de ambos (o que falhou e o que notificou a falha).
- o sistema, considerando aqui apenas estes dois objetos distribuídos, voltará a um estado consistente e, por conseqüência, todo o ambiente estará consistente.

O sistema todo passou a depender muito da ação do objeto Diretório, tornando-o indispensável. Uma falha neste objeto compromete toda a aplicação.

Para solucionar este problema, uma técnica de replicação do objeto Diretório foi desenvolvida [OLI 98, OLI 99]. Grandes alterações foram sugeridas ao objeto Diretório para que ele trabalhe permanentemente com um *backup*.

O diagrama de estados definido na Figura 4.4 descreve a primeira função do objeto Diretório como sendo a criação do seu objeto *backup*. Na primeira vez que o objeto Diretório é criado, o ambiente DPC++ deverá também criar o mesmo objeto Diretório em outro nodo da rede, que funcionará como um objeto Diretório *backup*.

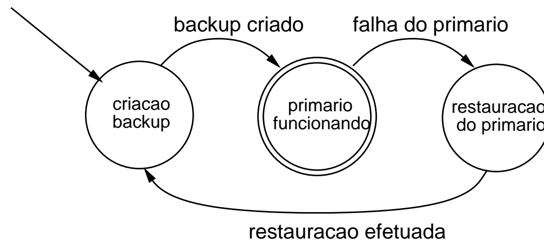

FIGURA 4.4 – Representação do modelo proposto para o objeto Diretório.

O segundo estado é caracterizado pelo fato do objeto Diretório primário estar funcionando perfeitamente, por isso este estado é considerado como estado final do diagrama. A função descrita no terceiro estado é a restauração do objeto Diretório primário. No momento da execução, a situação de falha é assumida pelo objeto distribuído quando este faz uma requisição ao objeto Diretório primário e não obtêm resposta dentro de um tempo limite. A próxima ação a ser realizada pelo objeto distribuído é fazer a mesma requisição ao objeto Diretório *backup*, que, concluindo que o primário falhou, assume a função de primário e cria um novo objeto Diretório *backup* em outro nodo da rede.

4.4 Concorrência entre Métodos no DPC++

O DPC++ vem sendo continuamente melhorado, incorporando novos recursos como a concorrência entre métodos. Na atual versão, sempre que um objeto estiver executando um determinado método, estará indisponível para atender outros. Atualmente, no DPC++, a execução de métodos é considerada síncrona, isto é:

- o objeto que executa o método fica bloqueado esperando;
- o objeto onde o método encontra-se encapsulado também fica indisponível para atender novos métodos.

O primeiro fato não chega a ser um grande problema, pois assemelha-se a execução normal de métodos, onde a execução envia parâmetros e aguarda pelo recebimento do resultado.

Já o segundo é um fator limitante do desempenho, pois um único objeto distribuído pode conter vários métodos e é desejável que um único método tenha várias execuções simultâneas. Cada objeto distribuído no DPC++ funciona da seguinte forma:

- cada objeto é um processo em execução;
- este processo, ao ser criado, aguarda mensagens;
- ao receber mensagem de requisição de um novo método, recebe os parâmetros e o executa localmente;
  - novas mensagens enviadas a este objeto não serão atendidas;

  - um outro objeto pode permanecer bloqueado esperando sua vez para executar algum método remoto deste objeto distribuído;
  - por causa da demora, objetos podem desconfiar que este objeto falhou, enviando mensagens ao objeto Diretório.

- quando o método local termina sua execução, o resultado é retornado localmente;
- este resultado é empacotado em uma mensagem e devolvido ao objeto procurador que fez a requisição do método;
- o objeto, enfim, fica disponível para atender novas requisições.

Isto limita a exploração de paralelismo, principalmente em máquinas com dois ou mais processadores. Se cada nodo tem apenas um objeto distribuído, ou seja, apenas um processo, ele é capaz de atender apenas um método por vez e isto usando somente um dos processadores. Um grande desperdício de recursos.

A nova proposta [ÁVI 99a] utiliza processos leves (*threads*) para obter este paralelismo, possibilitando que uma requisição de métodos dispare uma *thread* responsável por atendê-la, liberando imediatamente o nodo para que possa receber outras requisições. Esta nova abordagem, alvo de estudos neste trabalho, pode ser visualizada na Figura 4.5.

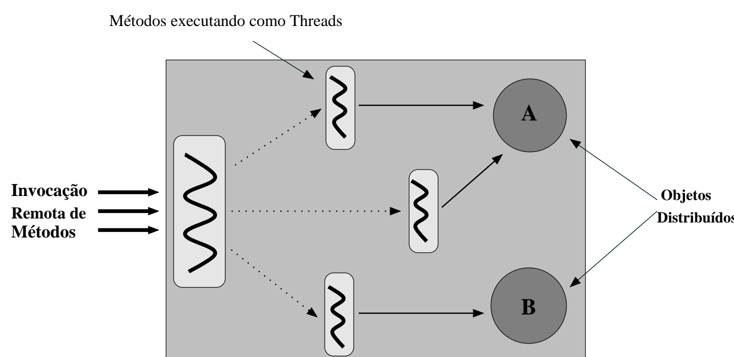

FIGURA 4.5 – Concorrência entre métodos no DPC++

Com esta proposta, um objeto distribuído tem condições de manter sempre uma *thread* com a função de atender as mensagens que chegam. Ao chegar uma requisição de método, um novo processo leve é rapidamente criado e a execução do método é passada para este, liberando prontamente o objeto para que possa atender novas requisições. A cada invocação de método, uma nova *thread* é criada e uma exploração de paralelismo mais eficiente é atingida.

Outra vantagem é que um método pode agora ter várias execuções simultâneas e mesmo a requisição de método pode ser assíncrona, ou seja, quem fez a chamada não precisa aguardar o termino desta. Isto pode ser comparado a métodos *void* em linguagens normais, ou seja, métodos que não retornam valor algum.

A implementação destas características no DPC++ será realizada com a migração do modelo para o DECK.

4.5 DECK

O DECK (*Distributed Execution and Communication Kernel)*, é um modelo para um ambiente de programação paralela que permite o desenvolvimento de aplicações paralelas irregulares, através da combinação de multiprogramação com comunicação [BAR 98]. Suas principais diferenças em relação as bibliotecas de comunicação existentes são o fato de disponibilizar controle de processos leves (*threads*) e comunicação entre elas. Esta comunicação pode ser feita tanto através de memória compartilhada, em uma mesma máquina, ou através de troca de mensagens entre máquinas distintas. Um mecanismo de *mailbox* foi desenvolvido para prover esta última comunicação.

O objetivo do DECK é fornecer um conjunto satisfatório de serviços, bem como operar independentemente da arquitetura de rede existente. Inicialmente, ele foi construído utilizando-se TCP/IP para prover a comunicação, seguindo-se uma versão utilizando BIP para uma rede *Myrinet* [BAR 2000] e uma versão para rede SCI [OLI 2001].

4.5.1 Camadas do DECK

O DECK é internamente dividido em dois subníveis, de acordo com a Figura 4.6. O subnível mais próximo da aplicação, chamado de "camada de serviços", tem a implementação dos serviços disponíveis para a aplicação. Já a camada inferior, denominada *uDECK*, é mais interna ao sistema, sendo ela totalmente dependente do *hardware* da máquina. Desta forma a camada de serviços dispõe de apenas um serviço de envio de mensagem e ele será o mesmo, independente de estar usando TCP ou outro protocolo específico, pois o protocolo é preocupação da camada de baixo nível.

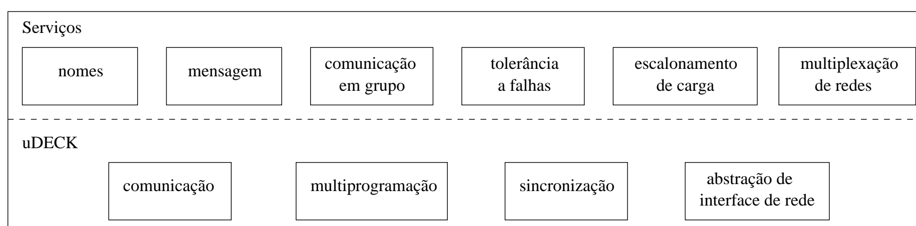

FIGURA 4.6 – Camadas do deck

4.5.2 Utilização do DECK

Como resultado da integração de comunicação e multiprogramação, DECK oferece dois níveis de concorrência:

- processos distribuídos e
- *threads*.

Essa característica é importante, pois permite a sobreposição de operações de comunicação com computação, aumentando assim o desempenho e utilizando de maneira mais eficiente os recursos das arquiteturas baseadas em agregados (máquinas SMP e redes de comunicação de alto desempenho).

A especificação dos nodos utilizados pela aplicação é feita pelo usuário em um arquivo de nodos, com um nome de máquina por linha, de forma análoga ao MPI. A numeração dos nodos, em tempo de execução, é dada pela ordem em que os nomes das máquinas aparecem nesse arquivo: para n nodos, a numeração varia de 0 a $n-1$.

Em cada nodo, uma cópia da aplicação é executada (o DECK é SPMD). Esse processo pode criar múltiplas *threads*, de acordo com suas necessidades computacionais. A comunicação entre *threads* pode ser realizada por memória compartilhada, dentro do mesmo processo, ou por troca de mensagens, entre diferentes processos. No primeiro caso, mecanismos de exclusão mútua são oferecidos para garantir que o acesso concorrente a dados globais seja realizado de forma correta. No segundo caso, diferentes tipos de dados podem ser trocados entre os *threads* através de caixas postais (*mailboxes*).

Cada *thread* pode possuir uma ou mais caixas postais com um identificador único no sistema. Usando esta caixa postal, a *thread* pode comunicar-se com todas as demais.

4.6 Integração do DPC++ com o DECK

A integração do DPC++ com o DECK na construção de um novo compilador trará sensíveis mudanças à linguagem. A principal delas é o uso de *threads*, possibilitando uma programação com granulosidade mais fina e a concorrência entre métodos descrita na Seção 4.4. Uma avaliação das mudanças estruturais na linguagem precisa ser estudada para a construção do novo compilador, pois algumas são significativas:

- o modelo de tolerância a falhas precisa ser revisto, pois a existência de várias *threads* não é prevista na solução inicial;
- a comunicação é realizada através de caixas postais e não mais pelo identificador do processo;
- a programação é SPMD, com todas as máquinas executando a mesma aplicação, cada uma com suas respectivas *threads* executando métodos;
- não será tarefa do DPC++ definir quais nodos serão usados. A necessidade do arquivo descritor precisa ser revista;

A comunicação entre objetos no novo modelo é feita pelo uso de caixas postais. Isto possibilita que cada *thread* tenha uma ou mais caixas postais. Como um objeto pode criar várias *threads*, tem-se a possibilidade de usar algumas delas com o único objetivo de receber e enviar mensagens.

Outro fato é que um objeto distribuído não será mais um processo válido no sistema operacional, com identificador e com espaço em memória. A única estrutura alocada para ele será a *thread* responsável por atender requisições de método e a execução inicial do seu construtor. Somente quando uma requisição for recebida é que uma nova *thread* é criada para atender este método e o objeto passa a estar novamente disponível para atender outros métodos ou até mesmo uma nova requisição para o mesmo método já em execução.

Com o uso do DECK, torna-se desnecessário e mesmo sem sentido a definição de máquinas a serem usadas como nodos no arquivo descritor da aplicação. O número de nodos existentes é determinado em tempo de execução, sendo tarefas do DECK, que carrega a mesma imagem de processo da aplicação em todos os nodos, numerando-os seqüencialmente. O DPC++ terá que definir qual nodo executará o objeto Diretório, podendo convencionar-se, por exemplo, que o *nodo 0* terá esta função. Esta característica retira do DPC++ a responsabilidade de controlar quais máquinas serão usadas no seu arquivo descritor. Uma futura implementação do DPC++ utilizando o DECK pode até mesmo não ter o arquivo descritor da aplicação.

4.7 Considerações finais

A linguagem de programação DPC++, criada inicialmente para máquinas comuns conectadas através de uma rede comum, está sensível às mudanças tecnológicas existentes. Desde sua concepção original até hoje, a idéia de clusterização ganhou força.

Para explorar de forma eficiente estas máquinas, com mais de um processador, alguns problemas que limitavam a exploração do paralelismo no DPC++ precisavam ser resolvidos e soluções foram sugeridas. Estas soluções ainda não foram incorporadas ao compilador DPC++, o que deverá ocorrer através da integração do DPC++ com o DECK.

Ao fazer isto, algumas características importantes do DPC++ serão alteradas:

- objeto distribuído deixa de existir fisicamente;
- sistema de tolerância a falhas baseado em *checkpoints* precisa ser revisto;
- mecanismo de comunicação torna-se mais complexo;

Com o novo modelo, o que executa efetivamente é um método, passando a ser ele independente e sendo visto pelo escalonador do Sistema Operacional como um processo que recebe ciclos de CPU de forma independente. Sendo cada método independente, o mecanismo de tolerância torna-se falho, uma vez que ele atuava sobre dois processos envolvidos na comunicação. Antes apenas dois processos, isto é, dois objetos, precisavam ser restaurados em caso de uma falha, pois a requisição de métodos era síncrona. Agora, um único objeto pode estar atendendo a várias requisições de métodos simultaneamente. Um novo modelo de tolerância a falhas em objetos precisa ser estudado.

Quanto à comunicação, esta era endereçada a processos, seja usando seu identificador aliado ao número da máquina ou algum mecanismo fornecido por bibliotecas de comunicação. Agora o sistema deve comportar comunicação entre *threads*. O uso do conceito de *mailboxes* implementada pelo DECK resolveu este e outros problemas.

5 Modelo Proposto

O modelo de escalonamento proposto neste trabalho procura atender as necessidades de escalonamento da linguagem DPC++. Como mencionado no Capítulo 4, o DPC++ possui muitas características, como tolerância a falhas, uso de concorrência entre métodos, entre outras, sendo que muitas ainda não estão implementadas no compilador. Uma análise das principais características da linguagem será realizada neste Capítulo.

5.1 Análise do DPC++

A linguagem DPC++ foi inicialmente desenvolvida para fornecer uma programação mais simples que as existentes nas bibliotecas de comunicação, implementando objetos distribuídos como processos independentes. Dentre as características desta linguagem que interessam à definição do modelo, pode-se citar:

- existência do objeto principal chamado objeto Diretório;
- necessidade da definição de um arquivo descritor;
- tolerância a falhas implementada através de *checkpoints*;

Pelo fato de existir um único objeto responsável pela criação remota dos demais objetos distribuídos, todo o gerenciamento de tarefas, como a criação e destruição de objetos distribuídos, é realizada pelo objeto Diretório. É ele quem decide em qual máquina será criado o objeto distribuído requisitado, ficando responsável também por devolver ao objeto que requisitou a criação o identificador para que a comunicação seja possível. A importância do objeto Diretório no DPC++ motivou a criação de um sistema de replicação deste objeto, garantindo tolerância a falhas, conforme descrito no Capítulo 4.3.

Para que o objeto Diretório selecione a máquina que executará o processo que representa um objeto distribuído, é necessário que o programador crie um arquivo de descrição, com o nome das máquinas que farão parte do sistema. O programador pode, caso deseje, decidir qual delas receberá o objeto Diretório. É neste arquivo também que o programador descreve o nome de suas classes e seus arquivos correspondentes.

Na Figura 5.1, temos um descritor com duas classes distribuídas utilizando quatro máquinas. A palavra *start* indica que o objeto Diretório deve ser criado na máquina de nome *verissimo.*

O objeto Diretório possui ainda a tarefa centralizada de prover a tolerância a falhas, atendendo consultas de estado de algum objeto, restaurando-o, através do último *checkpoint* em algum outro ponto do sistema, restabelecendo a comunicação.

```
start: Msg.cc
in: verissimo
with: ostermann, verissimo, dionelio, euclides

dclass: ENVIA
at: envia.dc

dclass: RECEBE
at: recebe.dc
```

FIGURA 5.1 – Exemplo de um descritor no DPC++

5.2 Definição do modelo

Pelas características do DPC++, principalmente a existência do objeto central chamado objeto Diretório, o modelo de alocação de tarefas já é, conceitualmente, centralizado. Implementar no DPC++ um escalonamento distribuído não teria muito sentido, pois o objeto diretório é, pela especificação da linguagem, responsável pela criação dos objetos.

O mecanismo de escalonamento proposto é composto de dois módulos: objetos espiões [ELR 94] e o escalonador central. A tarefa dos objetos espiões é fazer estimativas de carga nos nodos e informar ao escalonador o estado dos mesmos. A tarefa do escalonador central é, basicamente, manter uma tabela de nodos com suas estimativas de cargas atualizadas e aguardar requisições de escalonamento de objetos. A Figura 5.2 mostra o modelo proposto.

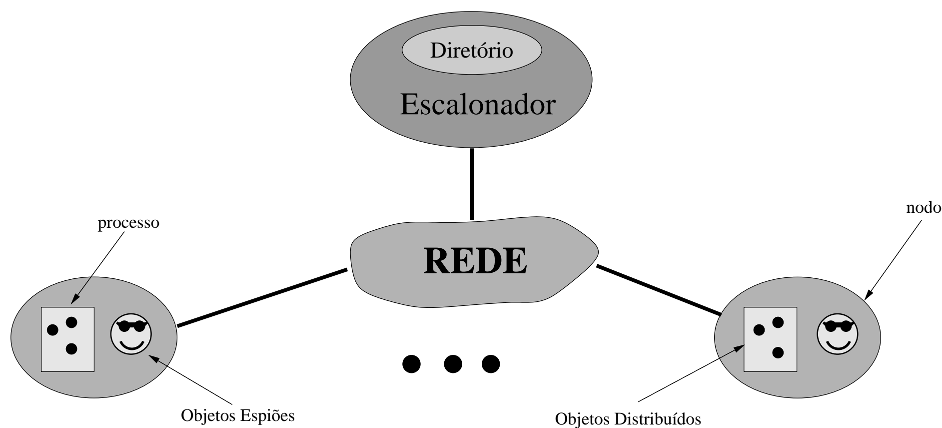

FIGURA 5.2 – Modelo do escalonador

Cada nodo participante da aplicação DPC++ terá um objeto espião responsável pela atualização de sua carga junto ao escalonador. Este objeto tem a função de monitorar o *nodo* onde se encontra e relatar mudanças significativas ao escalonador central.

5.2.1 Objetos Espiões

A existência dos objetos espiões torna o modelo proposto um escalonador dinâmico, pois as informações sobre a carga computacional de cada *nodo* são atualizadas

constantemente. Os espiões são responsáveis por manter o escalonador central informado sobre mudanças na carga computacional de cada nodo. Entende-se por carga não apenas os objetos criados pelo próprio DPC++, mas todos os processos em execução na mesma máquina, mesmo aqueles que não pertencem à aplicação distribuída.

Como o espião envia mensagens ao escalonador freqüentemente, estas mensagens podem gerar um *overhead* considerável. Para minimizar o número de mensagens, os espiões somente notificarão o escalonador central quando houver uma mudança realmente significativa na carga. O que o sistema adotará como "mudança significativa" também depende do estado do sistema.

Inicialmente, por exemplo, pode-se assumir que não interessa ao escalonador tomar conhecimento de mudanças inferiores a 5%. Depois, caso o sistema torne-se carregado, este valor pode ser trocado para, digamos, 20%. Desta forma pode-se variar o *overhead* com as trocas de mensagens entre espiões e escalonador de forma dinâmica.

Uma outra característica do modelo é a avaliação de sua própria influência sobre o nodo que controla (pois ele também é consumidor de recursos), podendo diminuir esta influência, se necessário. Isto classifica o modelo como adaptativo [CAS 88, KRU 94].

Sempre que o espião fizer uma nova avaliação de carga no nodo, verificando a necessidade ou não de notificar o escalonador central, ele permanecerá inativo por algum tempo, no qual não ganhará nenhum recurso de CPU. Obtem-se a adaptação possibilitando aumentar ou diminuir este tempo. Se o nodo estiver com carga alta, o espião pode ficar inativo por um tempo maior, caso contrário, não haverá problemas em se fazer avaliações de carga mais freqüentes.

O funcionamento lógico do espião, mostrado na Figura 5.3 pode ser descrito da seguinte forma:

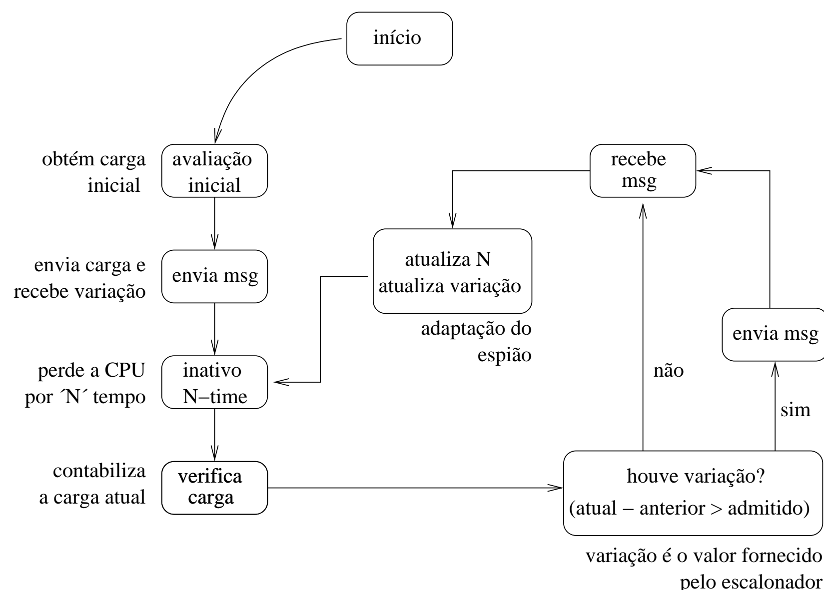

FIGURA 5.3 – Funcionamento do espião

a) ao iniciar, o espião faz uma avaliação inicial do nodo;

b) o resultado é enviado ao escalonador central, no caso o objeto Diretório;

c) o espião deixa de executar por um período de tempo, saindo da fila de prontos;

d) a carga é verificada novamente;

e) a variação ocorrida nos parâmetros de ociosidade é calculada:

   - o escalonador é notificado se a variação se a variação justificar;

f) o espião verifica se existe uma mensagem do escalonador;

   - mensagem para redefinir parâmetros de avaliação ou;
   - mensagem indicando que uma nova carga acaba de ser alocada;

g) os parâmetros são atualizados;

h) o espião recomeça no ítem "c", saindo da fila de prontos;

O espião somente envia uma mensagem quando a variação de carga justificar. O escalonador central pode controlar remotamente este valor, podendo decidir que seus espiões devem comunicar sempre que houver uma variação de carga em 5% no início e depois mudar este valor para 10, 20 ou 30%. O tempo de inatividade do objeto espião também é configurável, só que neste caso pelo próprio espião. Se o espião estiver em uma situação de carga alta ou carga baixa, e esta situação não variou nas últimas N avaliações, o tempo de inatividade pode ser aumentado, diminuindo a influência do espião.

A diferença entre estes dois parâmetros é que a variação visa diminuir o número de mensagens que circulam no sistema, enquanto que o tempo de inatividade apenas a influência do espião no nodo que gerencia.

5.2.2 Escalonador Central

O escalonador central estará incorporado ao objeto Diretório, conforme já justificado anteriormente. Na versão atual do objeto Diretório a escolha de um novo nodo para receber um objeto distribuído é realizada respeitando-se uma lista circular, sem levar em conta condições de carga. O escalonador central proposto é detalhado na Figura 5.4.

As tarefas do escalonador central são:

- manter uma tabela atualizada do estado dos nodos pertencentes à aplicação;
- receber e tratar as informações provenientes de seus espiões e
- atender às requisições de criação de objetos distribuídos.

Quando algum objeto distribuído do sistema requisitar a criação de um novo objeto distribuido, o nodo com menor carga, de acordo com os parâmetros do escalonador, será escolhido. A participação do escalonador neste processo será apenas informar ao objeto Diretório qual máquina deverá ser usada.

O escalonador tem ainda a tarefa de iniciar os objetos espiões nos nodos que participam da aplicação. Como este escalonador é parte integrante do objeto Diretório no DPC++, o construtor dele é responsável por criar os espiões em cada um dos processadores.

Após criados, o endereço de cada espião é retornado ao escalonador e ele aguardará uma mensagem de cada espião, informando o estado inicial dos nodos. Este estado inicial poderá incluir características físicas do nodo, como capacidade de processamento, velocidade de CPU, etc. Entretanto, estas informações não são necessárias caso todos os nodos sejam idênticos.

Após receber a situação inicial, uma tabela contendo o estado dos mesmos é construída e o escalonador está apto a receber requisições, fornecendo o nodo de menor carga ao requisitante (no caso, o objeto Diretório).

5.3 Considerações Finais

A presença do objeto Diretório no modelo de programação DPC++ torna a alocação de tarefas implicitamente centralizada. A princípio pode-se considerar a existência de um "gargalo" nestas condições, mas algumas características devem ser levadas em consideração:

- tipicamente uma aplicação não cria objetos durante toda a sua execução. Os objetos são criados, em sua maioria, no início da aplicação;
- a programação possui uma granulosidade bastante alta, encapsulando objetos distribuídos dentro de processos;

É claro que a presença do objeto Diretório impõe ainda outros problemas, em especial o aspecto de tolerância a falhas. De qualquer forma, ele impõe uma centralização ao modelo de escalonamento, de forma que não haveria sentido um escalonador distribuído para o DPC++.

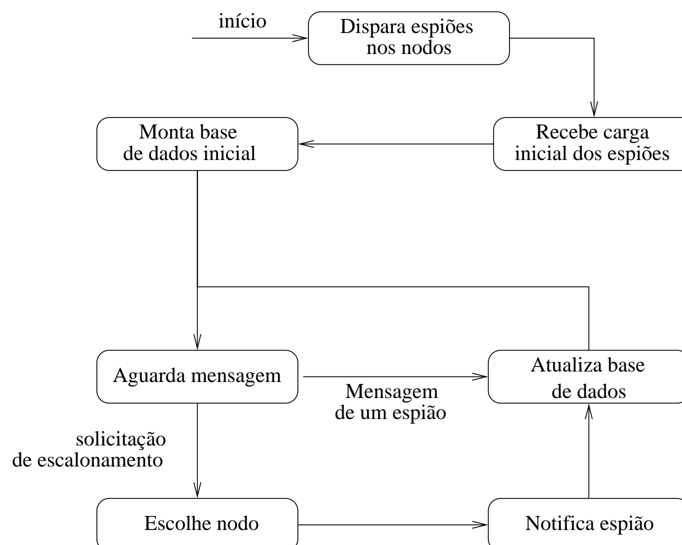

FIGURA 5.4 – Escalonador Central

O escalonador proposto pode ser classificado como centralizado, dinâmico e adaptativo, com as características principais mostradas na Tabela 5.1.

TABELA 5.1 – Características do Modelo

Característica	Descrição:
centralizado	um objeto decide onde outros objetos serão alocados
dinâmico	informações de carga são atualizadas. Uso de espiões
adaptativo	espião inativo por algum tempo: tempo ajustável envia mensagens somente quando variou: variação ajustável

A adaptação é obtida pela variação do tempo de inatividade dos espiões. Este tempo pode ser bastante pequeno quando o nodo estiver ocioso, ou pode ser consideravelmente alto quando estiver com extrema carga, tornando o modelo quase estático. Considera-se, neste caso, que um nodo sobrecarregado não deve ser muito influenciado por seu espião, devendo este moderar um pouco os seus testes.

O envio de mensagens também pode ser ajustado de forma dinâmica. Pode-se ajustar os espiões para informar mudanças que interessam ao escalonador, evitando o envio de mensagens que apenas irão repetir informações já conhecidas pelo escalonador ou mesmo para notificar uma variação de carga insignificante, que não chega a influenciar na decisão do escalonador. Considera-se que o próprio escalonador poderá informar a seus espiões o que ele considera como mudança significativa no nodo.

Este modelo de escalonamento centralizado foi proposto considerando-se o DPC++ sem a concorrência entre métodos. Durante o desenvolvimento deste trabalho, mudanças significativas ao DPC++ foram sugeridas e o modelo de escalonamento procurou se adaptar a estas novos características. A maior delas é que, com o uso do DECK, o modelo passa a ser SPMD, ou seja, cada nodo possuirá apenas um processo em execução e este processo tem condições de ativar qualquer objeto distribuído da aplicação, bem como iniciar qualquer método como um processo leve. A implementação do modelo adaptou-se a estas mudanças esperadas no DPC++.

6 Implementação

O modelo de escalonamento proposto no Capítulo 5 é baseado na implementação atualmente existente do DPC++, onde cada objeto distribuído é um processo independente, com sua própria imagem. Neste modelo, os objetos espiões seriam, da mesma forma, processos em execução em cada uma das máquinas existentes. Entretanto, este modelo do DPC++ já está ultrapassado pelas propostas de concorrência entre métodos, com sua integração com o DECK (Sessão 4.5), e a implementação do modelo de escalonamento proposto considerou estas mudanças.

6.1 Adaptação do modelo ao DECK

Os objetos espiões tem a função de enviar os resultados de sua avaliação para o escalonador central, que no caso é o próprio objeto Diretório. Uma adaptação do modelo proposto implementa os objetos espiões como um serviço do DECK. A definição do DECK prevê um serviço de escalonamento que ainda não foi implementado, conforme pode ser visto na Figura 6.1.

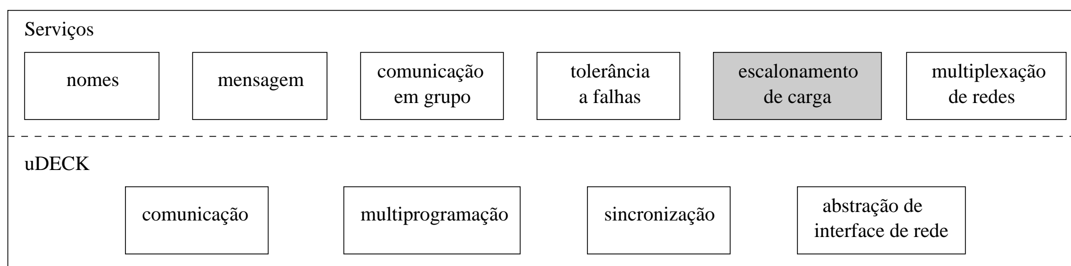

FIGURA 6.1 – Serviço de Escalonamento do DECK

Como o DECK oferece também um serviço de comunicação em grupo, os espiões podem postar uma mensagem do estado do sistema para um grupo. Pode existir até mesmo um sistema baseado em escalonamento distribuído, com a existência de dois grupos, cada um com uma caixa postal:

- um para os nodos conhecidos como *senders*, isto é, aqueles que estão querendo livrar-se de carga em excesso;
- outro grupo de *receivers*, com os nodos que estão ociosos.

O escalonador de cada nodo se encarregaria de incluir-se em um ou outro grupo, de acordo com as condições de sua carga.

O modelo de escalonamento para o DPC++ continua sendo centralizado, pela presença do objeto Diretório, mas o serviço de escalonamento implementado no DECK não precisa, necessariamente, ser centralizado.

6.2 Implementação do Espião

A implementação do modelo retirou a função de análise de carga do DPC++, deixando esta tarefa para ser realizada pelo DECK. Os motivos para esta abordagem

foram:

- os espiões podem ser usados por qualquer aplicação DECK, não apenas em aplicações distribuídas programadas em DPC++;
- o DECK prevê serviço de escalonamento de carga, ainda não implementado;
- o DPC++ fará uso do DECK em uma segunda versão do compilador;
- possibilidade de usar os espiões em um modelo de escalonamento distribuído;
- menor *overhead* e custo de inicialização, pois o espião executará como uma *thread*.

Uma aplicação pode usar ou não serviço de espião. Para utilizar o serviço, uma aplicação DECK deve:

1. inicializar o serviço através de uma chamada específica (*deck_spy_init()*);
2. criar uma caixa postal para receber notificações de carga dos espiões;
3. configurar os parâmetros iniciais para que os espiões comecem seu trabalho.

A caixa postal para envio de notificações de carga pode ser a caixa postal do escalonador central, como no caso do DPC++, ou uma caixa postal de comunicação em grupo, prevista no DECK, no caso de um escalonamento distribuído.

Como o DECK permite a utilização de *threads*, a implementação do espião terá suas funções divididas em duas:

- uma para avaliar a carga notificando o escalonador central sempre que for necessário e
- outra para aguardar mensagens do escalonador, indicando possíveis mudanças dos parâmetros de carga.

Na Figura 6.2 é possível visualizar o esquema lógico do espião implementado. Cada nodo participante da aplicação DECK, ao executar um *deck_spy_init()*, cria uma *thread* responsável pelo serviço de espionagem do nodo. Esta *thread* cria uma caixa postal e fica imediatamente bloqueada, aguardando o ativamento por parte da aplicação.

No caso do DPC++, o ativamento será feito pelo objeto Diretório, que enviará aos espiões de cada nodo o nome da caixa postal do escalonador, bem como os parâmetros iniciais de escalonamento. Ao receber estes dados, uma segunda *thread* com o objetivo único de monitorar a carga é efetivamente criada.

Tem-se, neste momento, duas linhas de execução para o espião. A primeira, referenciada na Figura 6.2 como *thread* principal, tem o único objetivo de atender às mensagens enviadas pelo escalonador. Esta linha de execução fica bloqueada na maior parte do tempo, não representando custo algum ao processamento. Quando uma mensagem chegar, ela rapidamente atualiza os dados necessários (compartilhados pela *thread* "espião") e volta a ficar bloqueada aguardando uma mensagem.

Já a *thread* "espião" faz uma medição inicial na carga e das características de *hardware*, envia estes dados ao responsável por tratá-los, que no caso do DPC++ é

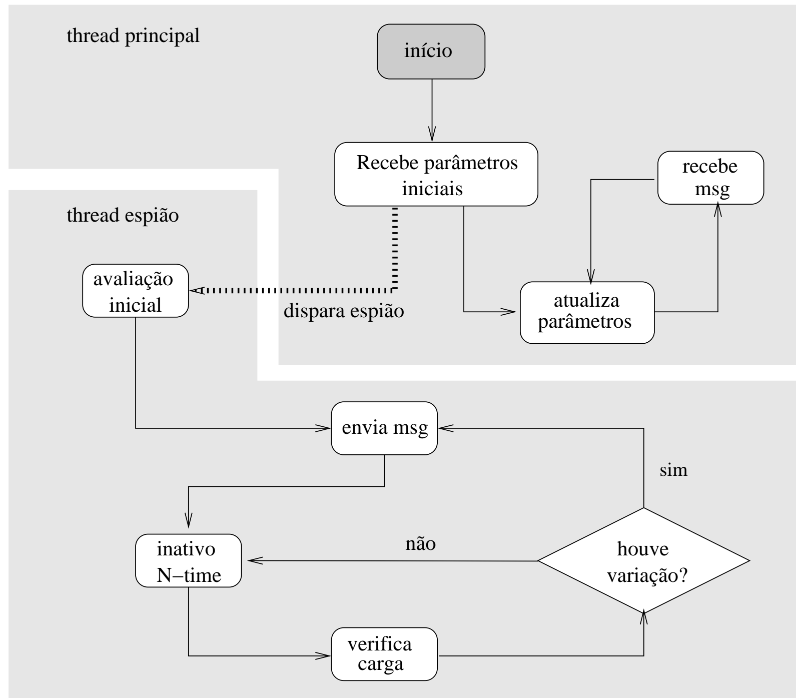

FIGURA 6.2 – Implementação do espião

o escalonador central no objeto Diretório, e entra em um laço infinito, medindo a carga, ficando inativa por um período de tempo, voltando a medir a carga. Se a carga atual variou de forma significativa, este espião envia esta mudança ao escalonador, usando a caixa postal que recebeu na primeira mensagem.

6.2.1 Funções internas do espião

Dentre as funções existentes internamente no espião, destacam-se

- *deck_spy_init()*: é a primeira função a ser executada pela aplicação, fazendo com que o espião seja efetivamente criado. Esta função cria uma caixa postal para receber mensagens e cria uma *thread* utilizando a chamada *deck_thread_create()* do DECK, fazendo com que esta passe a executar na função *deck_spy_main()*. O código desta função pode ser visto na Figura 6.3.

- *deck_spy_main()*: fica permamentemente em execução depois de ser criada. Sua função é receber os parâmetros iniciais do escalonador, configurar o espião e disparar uma *thread* para medir a carga. Após, fica bloqueada esperando mensagens do escalonador. Corresponde à "*thread* principal" visualizada na Figura 6.2. O código desta função pode ser visualizado na Figura 6.4.

- *deck_spy_run()*: esta função é um laço infinito executando como uma *thread*. Corresponde à "*thread* espião" representada na Figura 6.3. Faz uso de outras funções internas para medir a carga de um nodo. Seu código pode ser visto na Figura 6.5.

```
int deck_spy_init()
{
     char name[DECK_MAX_NAME_LEN];    /* Nome da caixa postal */

/* preparando nome: spyN, N=numero do nodo */
     sprintf(name, "spy%u", deck_node());

/* Criando caixa postal */
     deck_mbox_create(&mbox_spy);
     deck_mbox_bind(&mbox_spy, name);

/* Criando thread principal */
     deck_thread_create(&thread_main, deck_spy_main, NULL);
}
```

FIGURA 6.3 – Função *deck_ spy_ init()*

- *mede_ carga()*: a carga é avaliada consultando-se os valores de *load average*, número de processos na fila de execução e número de processos prontos para executar. A utilização destes valores é devido a facilidade em obtê-los, conforme estudado na Seção 3.3. Esta função também é responsável por configurar os valores de inatividade e de porcentagem da adaptação do algoritmo, se houver necessidade (detalhado na Seção 6.4)..

6.2.2 Estrutura de Dados utilizada

Na Figura 6.6 pode ser visto a estrutura de dados usada pelo espião, que possui os seguintes campos:

- número do nodo: para que o escalonador saiba de qual nodo pertence os dados, sendo um valor inteiro;
- número de processos prontos: um valor inteiro que expressa a quantidade de processos existentes na fila de prontos
- número de processos inativos: o número de processos na fila de espera, esperando por I/O;
- valor do *load average* atual;
- valor da média dos últimos 5 minutos para *load average*;
- valor da média dos últimos 15 minutos para *load average*;

```
int deck_spy_main()
{
    deck_msg_t msg;

    deck_msg_create(&msg, MSGSIZE, NULL);
    deck_mbox_retrv(&mbox_spy, &msg);

    spy_config(msg);

    deck_thread_create(&thread_spy, deck_spy_run, NULL);

    while(1) {
        deck_mbox_retrv(&mbox_spy, &msg);
        spy_config(msg);
    }
}
```

FIGURA 6.4 – Função *deck_spy_main()*

Sempre que os valores de algum destes campos variar de forma significativa, a estrutura inteira é enviada para o escalonador central.

Uma segunda estrutura de dados utilizada (Figura 6.6) é enviada uma única vez, podendo até mesmo não ser usada. Ela contém informações sobre o desempenho do *hardware* da máquina, como:

- número de processadores existentes na máquina;
- freqüência do processador em MHz;
- quantidade de memória física;
- valor de *bogomips*, calculado pelo sistema Linux;

O escalonador central possui as mesmas estruturas, com a única diferença de possuir várias delas, uma para cada nodo, dispostas em uma lista.

6.3 Implementação do Escalonador Central

A implementação do escalonador central não envolve o DECK, permanecendo o modelo apresentado. Com isto, livra-se o modelo de escalonamento de ser exclusivamente centralizado. Ele é centralizado no DPC++, pois a presença do objeto Diretório impõe esta condição. Mas, neste caso, o objeto Diretório será apenas um usuário do serviço de espionagem do DECK.

O objeto Diretório fica constantemente aguardando mensagens dos demais objetos distribuídos. Estas mensagens podem ser do tipo serviço ou de controle. Considera-se uma mensagem de serviço aquela que um objeto distribuído envia ao objeto Diretório solicitando a criação de um outro objeto distribuído. Mensagens de controle correspondem a todas as outras que não são solicitações de criação de objetos, como, por exemplo, mensagens do mecanismo de Tolerância a Falhas.

```
int deck_spy_run()
{
     esc_proc config;
     esc_carga *carga_a,  /* valor da ultima carga referenciada */
          *carga_o,      /* valor da carga atual (ultima lida) */
          *carga_temp; /* temporario, para fazer trocas de dados */

     spy_init_struct(carga_a);
     spy_init_struct(carga_o);

     mede_carga(carga_a);

/* Primeira medicao, sempre envia para o escalonador */
     notifica(carga_a);

/* Carga atual e ultima carga eh a mesma (primeira avaliacao)*/
     copia(carga_o, carga_a);

/* inicio do espiao. Fica constantemente medindo a carga e tomando acoes
   necessarias */
     while (1) {
          inativo(dorme);
          mede_carga(carga_a);

          if (mudou(carga_o, carga_a, porcent)) {
               carga_temp = carga_o;
               carga_o = carga_a;
               carga_a = carga_temp;
               notifica(carga_o);
          }
     }
}
```

FIGURA 6.5 – Função *deck_spy_run()*

O modelo incluiu mais um tipo de mensagem de controle, as mensagens dos espiões em execução nos nodos.

A sequência lógica do objeto Diretório, neste modelo, passa a ser:

- ao ser criado envia mensagens aos espiões de todos os nodos, com as informações:
  - a caixa postal do escalonador (próprio objeto Diretório);
  - valor da variação inicial a ser usada pelos espiões;
- aguarda mensagem dos espiões:
  - com características do *hardware*
  - número de processos na fila de prontos;
  - valor da carga inicial.

```
typedef struct proc_carga {
     int prontos;
     int sleeps;
     float lm;
     float l5m;
     float l15m;
     int node;
} esc_carga;

typedef struct proc_config {
     char np;       /* numero de processadores  existentes  no nodo */
     int mhz;       /* Velocidade  de  cada  porcessador  do  nodo */
     float megab;  /* Memoria em Mbytes de cada processador do nodo */
     int bogomips;
} esc_proc;
```

FIGURA 6.6 – Estrutura de dados usada pelo espião

- monta estruturas de dados inicial;
- objeto Diretório pronto para atender requisições;
- fica aguardando novas mensagens:
  - se for de criação de novos objetos, cria no nodo de menor carga;
  - se for dados de espião, atualiza estrutura de dados.

No modelo descrito no Capítulo 5, o objeto Diretório tem a responsabilidade de criar os objetos espiões nos nodos. Esta tarefa, agora, resume-se a apenas ativar os espiões já criados, durante o inicialização do DECK.

O objeto Diretório estará em execução no *nodo 0* somente. Antes de sua criação, portando, os espiões já foram disparados, mas ficam bloqueados esperando uma mensagem inicial. Isto significa que estes espiões podem ser inicializados em qualquer aplicação DECK, pois eles não irão fazer nada enquanto não forem ativados pelo escalonador. Pode-se considerar que uma aplicação DECK sempre terá os espiões, necessitando apenas que sejam ativados. Como estes apenas criam uma caixa postal e permanecem bloqueados, a existência deles, mesmo que ociosos, não acrescenta custo elevado à aplicação.

Uma observação a ser feita é a forma como espiões e escalonador trocam mensagens. Para que ambos possam fazer isto, precisam conhecer o endereço para postar mensagens. Isto, no DECK, é facilitado pela criação de caixas postais com nome e o uso de um servidor de nomes. Definiu-se que cada espião usará sempre uma caixa postal de nome *spyN*, sendo *N* o nodo no qual ele se encontra. Desta forma o objeto Diretório tem meios de enviar a mensagem inicial aos seus espiões.

O contrário já não é verdadeiro, pois os espiões devem enviar seus resultados à caixa postal que receberem na inicialização. Esta caixa postal não é, necessariamente, do escalonador central em execução no *nodo 0*, mas pode ser a de um grupo, onde cada escalonador de cada nodo participa. Isto garante a flexibilidade do modelo.

O construtor do objeto Diretório cria e inicializa as estruturas utilizadas, chamando uma função de inicialização do escalonador.

6.3.1 Funções principais do escalonador central

No Escalonador, algumas funções principais destacam-se:

- inicialização: envia uma mensagem para cada espião com parâmetros iniciais de avaliação. Esta mensagem serve também para ativar o espião no nodo e informar a caixa postal do escalonador.

- atualização: quando o objeto Diretório receber uma mensagem do espião notificando alteração da carga no nodo, esta função é executada e os dados recebidos na mensagem são recuperados.

- novo nodo: esta função é a principal do escalonador, que retorna o melhor nodo para receber uma tarefa. É a função *getANode()*. O critério para a escolha do nodo é o seguinte:

  - atua como uma lista circular, tentando alocar a tarefa ao próximo nodo da lista (*candidato*);
  - procura na tabela algum nodo com melhores condições de carga que o *candidato* atual (escolhido pela lista circular);
  - se a carga do nodo *candidato* for maior que a de algum nodo da tabela, o nodo *candidato* é descartado e o nodo com menor carga é escolhido;
  - elege como próximo nodo a ser escolhido na próxima requisição:
    - o próximo da lista circular, caso *candidato* atual foi o escolhido ou
    - o próprio *candidato*, caso ele não tenha sido eleito nesta fase;

Uma vez que um nodo é escolhido, sua carga e o número de processos da fila de prontos é imediatamente atualizada pelo escalonador. Isto é necessário, pois o espião irá demorar um pouco para enviar a nova situação de carga do nodo que acaba de receber uma nova tarefa. Se o escalonador permanecer com os dados antigos, ele pode eleger o mesmo nodo sucessivas vezes. A forma como isto é feito segue a seguinte regra:

- o número de processos prontos para executar é incrementado. Com a nova alocação de tarefa (que ainda não ocorreu) o nodo escolhido terá, com certeza, mais um processo na fila de prontos.

- a carga principal é modificada levando-se em consideração a carga atual e o número de processos existentes. O critério usado é dividir a carga pelo número de prontos existentes antes da nova alocação, adicionando este valor a carga atual:

$$carga = carga + carga/prontos$$

Esta "penalização" do nodo escolhido é suficiente para evitar sucessivas escolhas do mesmo nodo, pois o espião irá demorar algum tempo para notificar uma mudança de carga. Caso um nodo esteja sem carga, esta solução permite que ele seja escolhido várias vezes sucessivamente, pois mesmo com a penalização realizada o nodo ainda será o de menor carga em todo o sistema.

Outra possibilidade seria descartar como candidato o nodo anteriormente escolhido. Esta abordagem se mostrou ineficiente, por vários motivos:

- com apenas dois nodos, o algoritmo comporta-se como uma lista circular;
- no caso de um nodo ocioso e vários nodos com extrema carga, o escalonador deixará o nodo ocioso nesta condição, pois não alocará duas tarefas consecutivas para ele.

Logo que a carga for influenciada pela nova tarefa, o espião fará uma notificação. Na prática tem-se que objetos de execução rápida, que não chegam a aumentar a carga de um nodo, acabam sendo distribuídos de forma circular nos nodos. Isto porque o espião não irá fazer a notificação por considerar insignificante. O código da função *getANode()* é mostrado de forma resumida na Figura 6.7.

```
int Directory::getANode()
{
     int candidato, i;

/* Semaforo: esta funcao eh atomica. Entrando na secao  critica */
     deck_sem_p(&dpc_mutex);

     candidato = nextNode;

     for (i=0; i< deck_numnodes(); i++) {
          if (i == candidato)
               continue;

          if (compara(i, candidato)) {
               candidato = i;
          }
     }

     if (candidato == nextNode)
          nextNode = (candidato + 1) % deck_numnodes();

/* penalizando o nodo que recebeu a tarefa */
     dpc_carga[candidato].lm+= dpc_carga[candidato].lm/dpc_carga[candidato].prontos;
     dpc_carga[candidato].prontos++;

/* Saindo da secao  critica */
     deck_sem_v(&dpc_mutex);

     return (candidato);
}
```

FIGURA 6.7 – Função *getANode()*

6.4 Adaptação

O modelo implementado é adaptativo, ou seja, ele diminui a influência de seus espiões de acordo com a carga atual. A influência do escalonador no processamento é influenciada por dois fatores:

1. tempo de inatividade de cada espião
2. variação necessária para enviar uma mensagem de notificação

O escalonador Central envia, na primeira mensagem aos seus espiões, os valores que devem ser utilizados para carga baixa, podendo modificá-los a qualquer momento.

A implementação trabalhou com três valores diferentes para cada um dos ítens, um valor para carga baixa, para carga média e para carga alta:

- carga baixa: o tempo de inatividade de cada espião é bastante reduzido, fazendo com que o espião realize medidas mais freqüentes. A variação necessária para enviar uma nova notificação ao escalonador também é pequena.
- carga média: o tempo de inatividade do espião é o dobro do tempo com carga baixa e a variação necessária também é o dobro.
- carga alta: neste caso o tempo de inatividade é quatro vezes maior que o tempo de inatividade para carga baixa e a variação necessária também é quatro vezes maior.

O espião, ao medir a carga do nodo, divide esta carga pelo número de processadores existentes. Esta medida, oferecida pelo Sistema operacional, expressa a quantidade de processadores necessários para executar os processos existes.

Se o resultado obtido nesta divisão for menor que 0.5, o espião se enquadra no caso “carga baixa”, usando os parâmetros apropriados. Valores entre 0.5 e 1.0 se enquadram no caso de “carga média” e valores maiores que 1 são interpretados como “carga alta”.

6.5 Considerações Finais

A separação dos espiões do DPC++ como um serviço do DECK, não alterou o modelo inicialmente proposto. A diferença básica é que os espiões podem ser utilizados por qualquer aplicação DECK, não somente pelo DPC++. O principal problema neste caso, é o fato de um compilador DPC++ para o DECK ainda não ter sido desenvolvido, até porque o DECK ainda está em fase de amadurecimento e portabilidade para outras tecnologias, como a SCI.

Porém, sem dúvida, a maior vantagem foi possibilitar o uso dos espiões em modelos distribuídos de escalonamento.

O uso de *threads* fornecido pelo DECK só trouxe vantagens ao modelo, pois pode-se separar as funções do espião.

Importante observar a existência de concorrência entre as várias partes do modelo:

- as duas *threads* do espião compartilham dados entre si. Deve-se evitar que uma esteja atualizando os dados que a outra está lendo.

- o escalonador central pode ter uma *thread* atendendo uma solicitação de escolha de um novo nodo (*getANode()*) ao mesmo tempo em que uma outra *thread* recebeu notificação de um espião.

Estes dois problemas foram resolvidos com o uso de semáforos existentes no DECK. Sempre que um dado compartilhado for alterado, a *thread* precisa entrar na seção crítica executando um *deck_sem_p()*. Isto garante a exclusão mútua. No caso da função *getANode()*, um semáforo para toda a função foi criado, tornando-a uma função atômica e de exclusão mútua. Não podem existir duas funções *getANode()* executando ao mesmo tempo, de forma que apenas uma entrará em execução deixando as demais em espera.

7 Avaliação e Resultados

Para testar o modelo implementado foi necessário a definição de uma aplicação distribuída. Esta aplicação deveria atender algumas necessidades:

1. criação e destruição de objetos ao longo da execução da aplicação;
2. objetos criados com características de carga variadas;

O primeiro ítem é necessário porque aplicações que criam todos os seus objetos distribuídos na inicialização eram inadequadas, pois não há o que escalonar nesta abordagem.

A carga destes objetos também deve ser variada, de forma que os processadores terminem suas tarefas de forma totalmente imprevisível.

Uma dificuldade encontrada foi a inexistência de um compilador DPC++ para o DECK. Isto foi resolvido pela implementação de um código final, semelhante ao que o DPC++ geraria. Os espiões foram implementados diretamente no DECK, como um serviço.

Por suas características de computação irregular, uma aplicação de geração de fractais foi implementada para ser usada nesta avaliação.

7.1 Aplicação Implementada

A avaliação utilizou uma aplicação para o algoritmo de fractais de Mandelbrot [MAN 82]. Fractais consistem em imagens abstratas geradas através de operações matemáticas aplicadas sucessivamente sobre os pontos de uma figura.

É possível identificar três fatores que influem no custo de processamento de um fractal de Mandelbrot:

1. as dimensões (ou resolução) da figura que se deseja obter;
2. a região do fractal a ser calculada e;
3. o número de iterações sobre cada ponto.

O número de iterações deve representar um equilíbrio entre o tempo de processamento e a qualidade da figura que se deseja obter, em função das dimensões e da região de cálculo. Quanto maior o número de iterações, maior a qualidade do resultado e uma quantidade maior de processamento será necessária.

O uso desta aplicação deve-se ao fato de que a divisão dos pontos de forma igual entre cada nodo não implica em uma divisão de carga equivalente. Pontos mais internos precisam de muito mais processamento que os demais, pelas características do modelo de fractais empregado.

7.2 Implementação distribuída do algoritmo

Existem muitas formas de se dividir a carga entre os diversos processadores existentes. Uma implementação deste algoritmo usando o DPC++ sem concorrência entre métodos foi demonstrada em [CAV 95], com as seguintes características:

- um objeto chamado de "objeto Distribuição", que atende requisições, fornecendo uma quantidade de pontos a ser calculada;

- objeto saída: recebe os resultados prontos e envia para a tela, gerando a imagem final;

- N objetos de cálculo: calcula o conjunto de pontos recebidos, envia ao objeto saída e solicita mais pontos ao objeto Distribuição.

Cada objeto cálculo calcula os pontos que recebeu, envia o resultado ao objeto saída e solicita mais pontos para calcular. Faz isto sucessivamente até que o objeto Distribuição sinalize que não existem mais pontos para calcular, encerrando a execução dos objetos cálculos.

Este modelo, mesmo que explorando a concorrência entre métodos, não se beneficia do escalonador, pois a distribuição da carga é feita pela própria aplicação. Os objetos cálculos, uma vez criados, ficam solicitando carga ao objeto Distribuição. Tem-se, de forma natural, uma divisão de carga, pois um nodo que recebeu pontos mais difíceis computacionalmente irá demorar mais para solicitar novos pontos, enquanto que um que recebeu pontos menos complexos, solicitará mais pontos ao objeto de Distribuição.

Uma outra forma de divisão da carga, principalmente desejável em arquiteturas SPMD, é fazer com que cada nodo já saiba, de início, quais pontos ele deve calcular. Esta é a forma mais eficiente, pois dispensa o uso de um algoritmo de distribuição de dados. Se existirem, por exemplo, 4 nodos, o *nodo 0* irá calcular o ponto 0 e depois o ponto 4, 8, e assim por diante.

Esta forma de dividir mostra-se muito interessante, pois em virtude das características da geração de fractais, os pontos mais complexos serão também distribuídos. Entretanto esta forma também não se beneficia do algoritmo de escalonamento, pois todos os objetos de cálculo são criados no início da aplicação.

Era necessário uma forma de divisão na qual novos métodos fossem criados ao longo da execução da aplicação, para que o escalonador possa decidir onde eles seriam criados, considerando a carga dos mesmos naquele momento.

A implementação empregada é uma variação do primeiro modelo de divisão, onde cada objeto cálculo realiza o cálculo dos pontos iniciais e solicita novos pontos ao objeto Distribuição. A diferença consiste em que um objeto cálculo, ao terminar os pontos que recebeu, não solicita novos pontos, mas encerra sua execução. O objeto de distribuição procederá à criação de um novo objeto cálculo, usando o algoritmo de escalonamento.

A aplicação implementada possui:

- um objeto gerenciador, que fará a divisão dos pontos;

- vários objetos de cálculo, que serão criados por demanda durante toda a aplicação.

O objeto Diretório continua com as mesmas características de antes, exceto pela inclusão do escalonador central em seu código.

7.2.1 Relacionamento da aplicação com o DPC++

No modelo DPC++ sempre que a criação de um objeto for solicitado, ele é criado no nodo retornado pela função *getANode()*.

Esta função continua existindo, sendo que na versão sem o escalonador ela apenas respeita uma lista circular, entregando as tarefas para os nodos na sequência $0,\ 1,\ 2,\ldots\ N-1,\ 0,\ 1,\ldots$, sendo N o número de nodos existentes. Com o escalonador a tarefa desta função será a de escolher o nodo com melhores condições para receber a nova criação de objeto.

O arquivo descritor do DPC++ foi retirado pois ele existia no DPC++ apenas para informar ao compilador:

- quais são as classes distribuídas;
- quais as máquinas (nodos) que devem ser usadas;
- em qual nodo será criado o objeto Diretório.

O primeiro ítem é desnecessário pois o protótipo já é o código resultante, o código que seria gerado pelo compilador.

A relação dos nodo utilizados é feita pelo DECK no arquivo de nodos e convencionou-se que o objeto Diretório executará no *nodo*0.

7.2.2 Configurações da Aplicação

A aplicação implementada deve ser disparada utilizando-se parâmetros para definir o número de pontos que compõem a imagem, o número de objetos de cálculo que devem ser criados e a quantidade de pontos que cada objeto deve calcular. Para inicializar uma aplicação DECK, utiliza-se o comando *deckrun*, que possui alguns parâmetros:

- arquivo de nodos: um parâmetro determina o nome do arquivo que contém a lista de máquinas que deverão ser usadas pela aplicação. A ausência deste parâmetro fará com que o *deckrun* procure por um arquivo de nome "nodes".
- número de nodos: pode-se determinar quantos nodos devem ser usados. Se o arquivo de nodos possui oito máquinas, pode-se usar apenas quatro delas. A omissão de qualquer valor fará com que a aplicação seja disparada em todos os nodos.
- nome da aplicação: sempre é necessário, pois trata-se do nome da aplicação compilada que deve ter a execução iniciada em todos os nodos.
- parâmetros da aplicação: todos os parâmetros que a aplicação precisa para poder executar.

Um exemplo válido do disparo da aplicação mandel implementada pode ser:

deckrun mandel 200 200 100 4 1

Os valores passados à aplicação mandel indicam o número de pontos horizontais e verticais da figura a ser criada (uma imagem 200 x 200), o tamanho do pedaço que cada objeto cálculo irá calcular (100 pontos) o número de objetos cálculo existentes (4) e quantas vezes o teste será repetido (uma). A aplicação desenvolvida retorna a média de tempo de execução. Como os parâmetros para o deckrun foram omitidos, todos os nodos existentes no arquivo de nodos serão utilizados.

Aumentando-se o tamanho da imagem, para um mesmo número de objetos de cálculo e tamanho de cada parte, mais objetos serão criados e destruídos ao longo da execução. Este efeito também pode ser obtido diminuindo-se a constante que define o tamanho de cada parte. Em contrapartida, ao definir valores maiores para cada parte, cada objeto ficará mais tempo executando, influenciando mais a carga do nodo.

No exemplo, com uma imagem de 200x200, dividida de tal forma que cada parte tenha 100 pontos, tem-se que serão necessários 400 cálculos, ou seja, ao longo de toda a aplicação, 400 objetos cálculo serão criados e cada um calculará um pedaço de 100 pontos. No entanto, o sistema manterá apenas quatro objetos em execução de cada vez e sempre que um destes terminar, um novo objeto será criado até que todas as 400 partes sejam calculadas. O algoritmo de escalonamento será requisitado, neste exemplo, 400 vezes.

Um outro fato importante na avaliação dos resultados é a forma de divisão dos pontos. Isto é importante porque o fractal possui complexidades diferentes entre seus pontos. De maneira geral os pontos mais centrais da figura necessitam de um tempo muito maior de processamento. Isto é interessante para o modelo, pois dividir 200 partes para 4 nodos de forma que cada nodo calcule 50 delas não significa uma boa divisão de carga.

Na Figura 7.1 pode-se ver a irregularidade de execução de cada parte para uma imagem dividida em 200 partes, de maneira que cada parte possuisse 200 pontos no primeiro exemplo e 800 pontos no segundo.

Observa-se uma diferença acentuada entre as partes iniciais e as partes centrais. O tempo necessário para a execução variou na casa dos milhares. No exemplo da Figura 7.1 o pedaço da imagem com maior complexidade para 800 pontos por tarefa é a parte de número 97, que precisou de 18,62 segundos para ser concluído. Já a parte com menor complexidade foi a de número 0, que precisou de apenas 0,009795 segundos, ou seja, 970 milésimos de segundo. Trata-se de uma diferença de 1901 vezes. Obviamente se um determinado nodo receber mais pontos centrais em relação aos demais, ele ficará sobrecarregado.

7.3 Testes Realizados

Os testes de avaliação de desempenho foram executados no cluster *mirynet* do Grupo de Processamento Paralelo e Distribuído. Este cluster foi descrito na Seção 2.3 e possui as características físicas mostradas na Tabela 2.3. Possui quatro nodos com oito processadores no total.

A forma como os pontos do Mandelbrot são distribuídos influencia de forma significativa no desempenho final. Para os testes de desempenho, três formas básicas de distribuição de pontos foram implementadas: seqüencial, aleatório e otimizada.

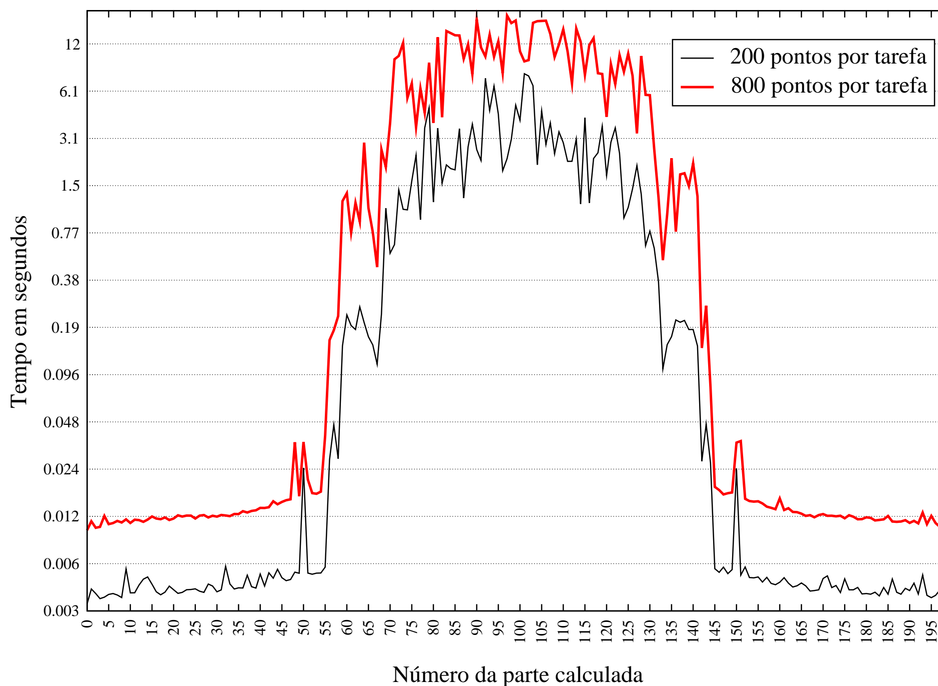

FIGURA 7.1 – Tempo de cada parte no modelo Mandelbrot

7.3.1 Divisão seqüencial

Neste tipo de divisão, o distribuidor entrega as partes de forma seqüencial, ou seja, se a aplicação foi dividida em 200 partes, o algoritmo irá distribuí-las respeitando a sequência 0,1,2,3,4, . . . ,199.

Na Figura 7.2 é mostrado o gráfico de desempenho com tarefas de pouca carga distribuídas de forma seqüencial. Neste exemplo foi usado um um valor bastante baixo para a convergência de cada ponto, com uma qualidade de imagem comprometida.

Os resultados mostrados na Figura 7.3 foram realizados com os mesmos parâmetros do gráfico da Figura 7.2, com a diferença de que programas externos foram disparados no *nodo*1 e *nodo*2 de forma a aumentar a carga. Houve, neste caso, uma injeção externa de carga.

Observa-se nestes dois casos mostrados nas Figuras 7.2 e 7.3 que o algoritmo de escalonamento não se mostrou muito eficiente. Com cargas muito baixas, o escalonador não tem muito o que fazer, tornando-se praticamente uma lista circular. Devido ao *overhead* gerado pelos espiões e pela manutenção das estruturas de dados, ele mostrou-se pior que a solução sem escalonador.

Porém, um comportamento atípico foi obtido para 8 objetos de cálculo e com tamanho de cada parte igual a 3200 pontos. Na Figura 7.2 observa-se que o desempenho neste caso foi ainda pior, mais expressivo que nos demais casos e isto não se deve apenas aos custos extras dos espiões.

O problema, neste caso, é que com apenas oito objetos ativos ao mesmo tempo e dispondo de quatro nodos com dois processadores cada, não há absolutamente nada que o algoritmo de escalonamento possa fazer. A solução é única: um objeto cálculo

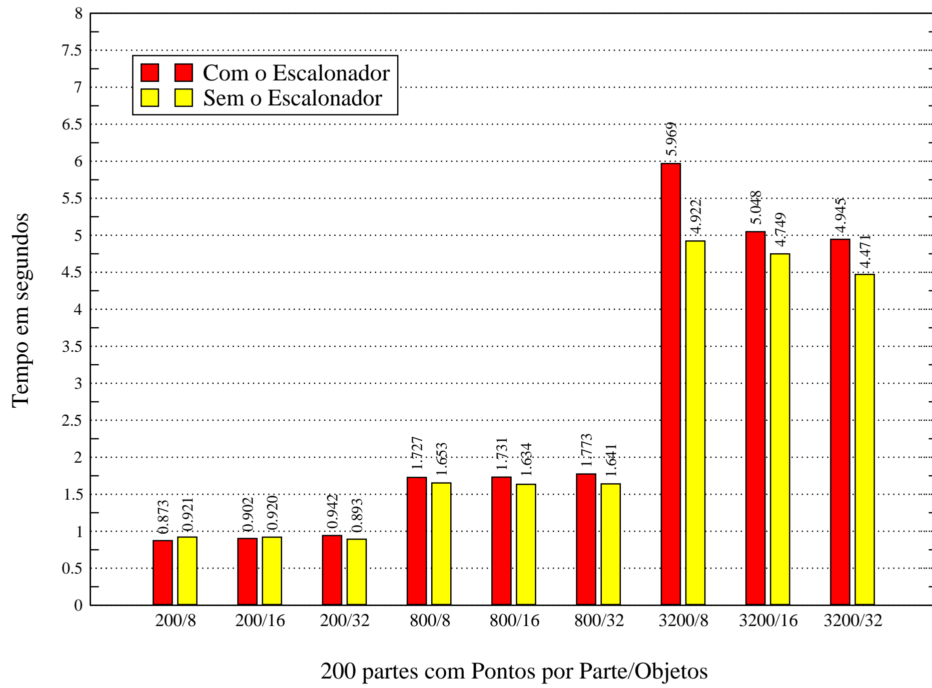

FIGURA 7.2 – Desempenho com Carga Baixa

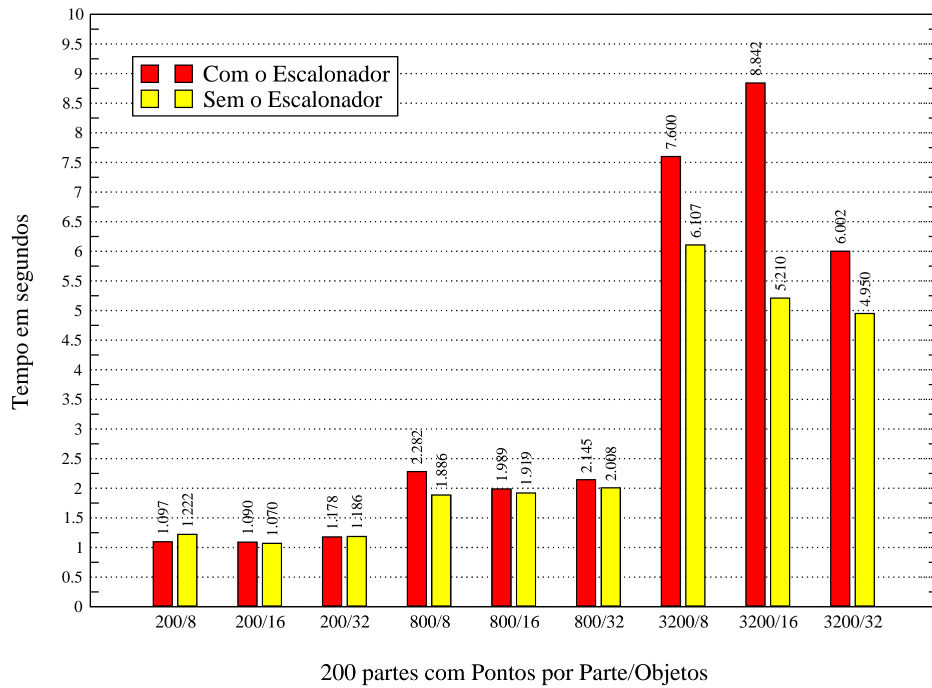

FIGURA 7.3 – Desempenho com Carga Baixa e Injeção de Carga

para cada um dos oito processadores. Assim sendo, quando um objeto de cálculo termina sua execução, um novo objeto deveria ser criado no mesmo processador, pois este ficará totalmente ocioso. Por levar em consideração o número de processo na fila de prontos e a carga de cada nodo, isto não é feito pelo escalonador.

Outro problema foi detectado no teste com injeção de carga, pois obteve-se um desempenho pior em relação aos testes realizados sem carga extra.

Isto ocorreu porque:

- a aplicação usada termina em poucos segundos, não influenciando a carga dos nodos;
- com pontos de complexidade pequena, cada objeto cálculo torna-se praticamente um *I/O Bound*, pois o tempo de comunicação passa a ser mais significativo;
- o escalonador, por considerar a carga externa, tem a tendência de jamais alocar tarefas para nenhum dos nodos com carga alta, carga esta que não pertence a aplicação. Com isto o escalonador deixa os nodos completamente ociosos do ponto de vista da aplicação.

Outros testes foram realizados aumentando-se a complexidade do cálculo de cada ponto. A Figura 7.4 mostra a avaliação de desempenho do algoritmo com tarefas de carga alta. Neste caso, cada ponto precisou de um processamento muito maior para ser calculado, pois aumentou-se a iteração de cada ponto.

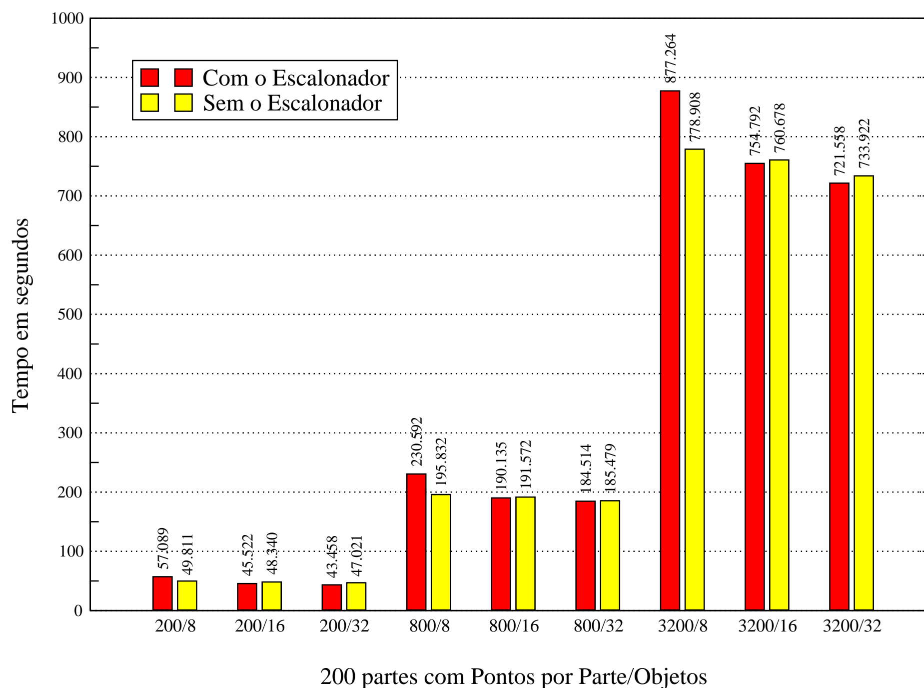

FIGURA 7.4 – Desempenho com Carga Alta

Obteve-se um desempenho melhor para a maioria dos casos, exceto para os testes realizados com oito objetos de cálculo. A explicação para este fato é a mesma

da avaliação com carga baixa: oito objetos de cálculo não são suficientes para manter todos os processadores ocupados o tempo todo, pois a única maneira de fazer isto é manter sempre um objeto em cada processador.

Na Figura 7.5 são mostrados os resultados obtidos com injeção de carga em dois dos quatro nodos utilizados.

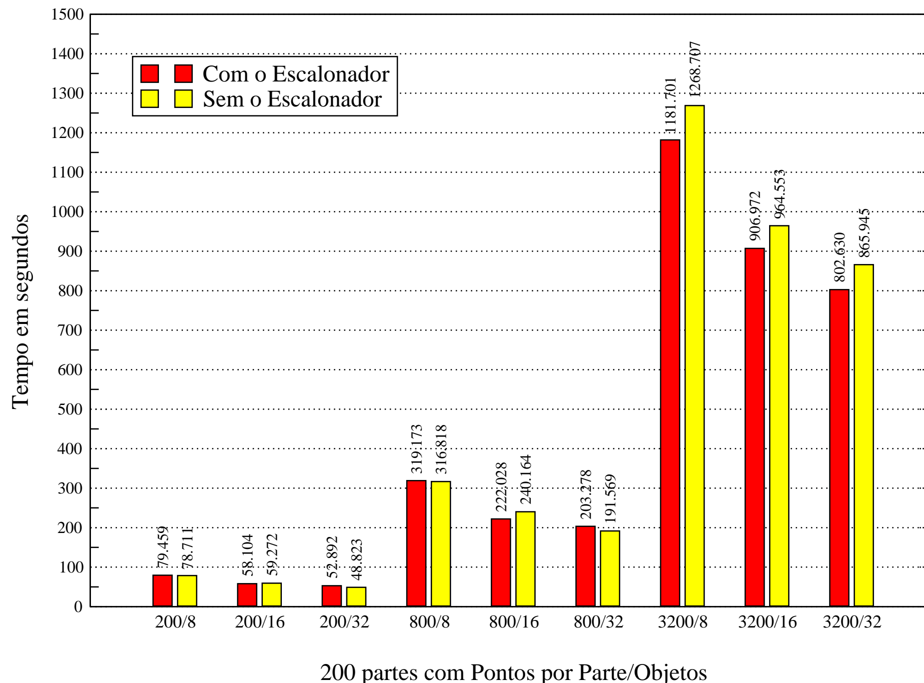

FIGURA 7.5 – Desempenho com Carga Alta e Injeção de Carga

Os testes com injeção de carga mostraram-se mais favoráveis, até mesmo para o exemplo problemático de oito objetos para oito processadores. Neste caso justifica-se o uso do escalonador, pois ele evitou colocar tarefas nestes dois nodos.

7.3.2 Divisão aleatória

A divisão de forma seqüencial, demonstrada anteriormente, possui como característica o fato de os pontos centrais terem complexidade de cálculo maiores. Pode-se considerar que a carga de cada conjunto de pontos é aproximadamente a mesma da parte 0 até a parte 50 (considerando-se a divisão em 200 partes). A partir deste ponto, a complexidade de cada parte aumenta de forma significativa. Na verdade ocorre um salto, com as partes mais complexas no intervalo de 70 até 140 (conforme demonstrado na Figura 7.1)

Podem ser identificados três faixas de carga diferentes:

- carga baixa, nas partes 0 até 50 e 150 até 199, aproximadamente;
- carga alta, na faixa compreendida entre as partes 70 a 140;
- carga média nas demais faixas.

A divisão seqüencial não permite que o escalonador obtenha um desempenho elevado, pois, com a divisão seqüencial, todos os nodos recebem partes de complexidade próximas.

No início da aplicação, considerando a existência de 16 objetos cálculos, as partes de 0 a 15 serão entregues para os nodos, possivelmente quatro partes para cada nodo, ou 2 para cada processador. A tendência natural é que todos os nodos terminem suas partes em tempos próximos, recebendo mais tarefas, desta vez da faixa 16 até 31.

Como resultado tem-se que toda a aplicação "caminha" para a faixa mais complexa, de forma que quando o nodo 0 estiver calculando, por exemplo, a parte 70, os demais nodos estarão calculando partes adjacentes.

Esta forma seqüencial de distribuir as tarefas beneficia o modelo sem escalonamento, que as distribui respeitando uma lista circular. O modelo com escalonador não foi prejudicado, pois ele obteve um desempenho favorável mesmo nestas circunstâncias.

Para tornar a aplicação mais irregular, onde o comportamento de cada tarefa é totalmente imprevisível, modificou-se o modelo para que a entrega das partes fosse feita de forma aleatória. A implementação desta técnica exigiu um pouco mais de cuidado, pois precisou-se marcar as partes que já foram entregues. Descartou-se, nestes testes, os resultados obtidos com carga baixa pois eles se comportam de forma análoga aos exemplos com distribuição seqüencial. Em todos os casos de carga baixa o algoritmo de escalonamento mostrou-se ineficiente.

Na Figura 7.6 são mostrados os resultados obtidos com esta técnica de divisão.

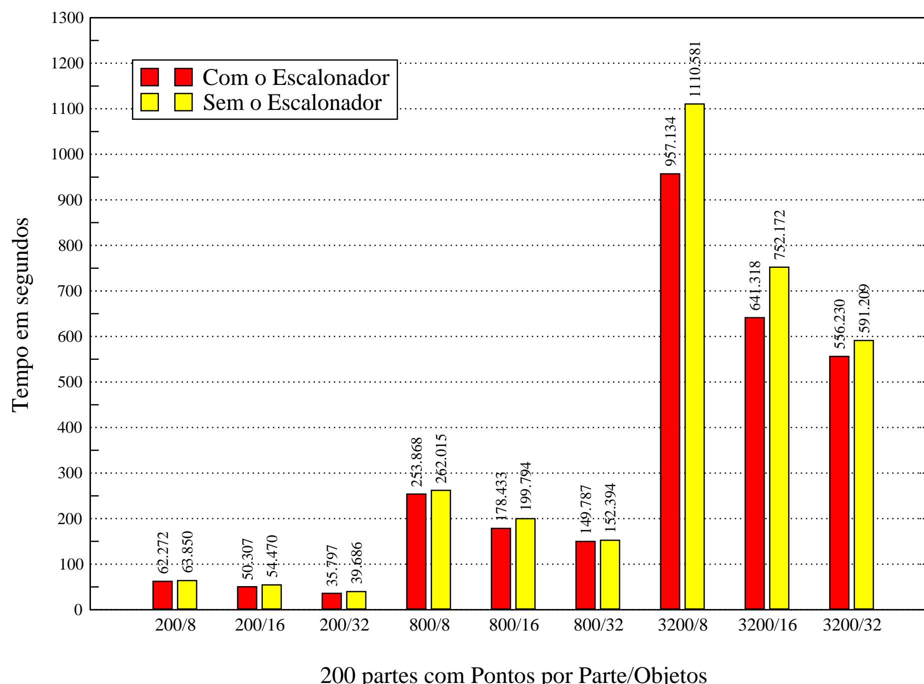

FIGURA 7.6 – Desempenho com Distribuição Aleatória

A Tabela 7.1 mostra os resultados obtidos na avaliação com a relação de au-

mento que o algoritmo sem escalonamento obteve.

TABELA 7.1 – Desempenho para Distribuição Aleatória

Teste	Com escalonador	Sem escalonador	%
200x200 200 8	62.272108	63.850484	2.53
200x200 200 16	50.307158	54.469940	8.27
200x200 200 32	35.796609	39.686379	10.87
400x400 800 8	253.867736	262.015351	3.2
400x400 800 16	178.432694	199.793755	11.97
400x400 800 32	149.786498	152.393795	1.74
800x800 3200 8	957.133870	1110.581292	16.03
800x800 3200 16	641.317998	752.172340	17.29
800x800 3200 32	556.229904	591.209088	6.29

Observa-se que o melhor desempenho obtido foi para 16 objetos, enquanto que para 8 e 32 objetos o desempenho foi bom, mas não na mesma proporção.

O caso da existência de apenas oito objetos já foi devidamente explicado. No entanto, o teste com 32 objetos é um tanto quanto intrigante. Para uma imagem de 400x400 com cada parte possuindo 800 pontos, teve-se que o algoritmo sem escalonamento levou apenas dois porcento a mais para terminar o cálculo.

Deve-se observar, neste caso, as seguintes circunstâncias:

1. são 200 tarefas em execução;

2. destas 200 tarefas, 32 são disparadas logo no início, restando 168 a serem criadas por demanda, sempre que uma tarefa terminar;

3. cada nodo pode receber, por exemplo, 8 tarefas no início;

4. existe uma grande possibilidade de um nodo receber mais de uma tarefa com carga alta, logo no início.

Com pedaços variando em sua complexidade em quase duas mil vezes, ocorre que se um nodo receber várias tarefas com carga alta, o escalonador não terá oportunidade de corrigir isto, pois restam apenas 168 tarefas. Não haverá muitas oportunidades para o algoritmo distribuir a carga nestas circunstâncias.

Outra característica é que a aplicação distribui as tarefas de forma aleatória para os dois casos. Significa que no teste com escalonamento, a sequência escolhida é diferente da usada no teste sem o escalonamento.

Testes com injeção de carga em dois nodos também foram realizados para o caso de distribuição aleatória. Os resultados obtidos podem ser vistos na Figura 7.7.

Com injeção de carga em dois dos quatro nodos obteve-se diferenças de tempo maiores que 30%, como mostrado na Tabela 7.2.

7.3.3 Divisão otimizada

Nesta forma de divisão, as partes são divididas em uma sequência não linear. O nome “otimizada” se refere ao fato de tratar-se do melhor caso para o modelo de escalonamento e, portanto, o pior caso para a versão sem o modelo.

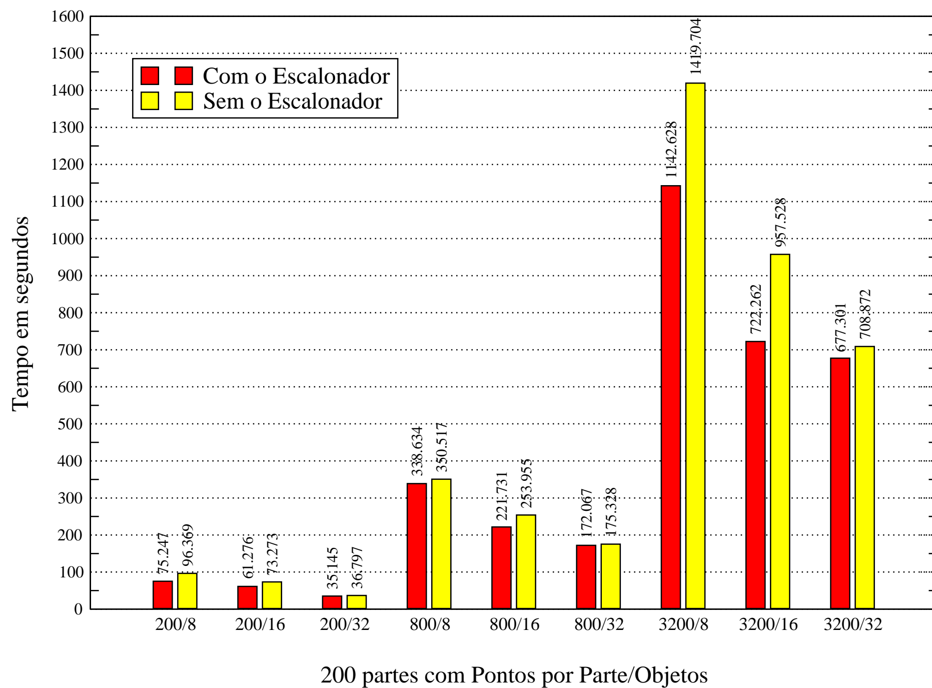

FIGURA 7.7 – Desempenho com Distribuição Aleatória e Injeção de Carga

TABELA 7.2 – Desempenho com Distribuição Aleatória e Injeção de Carga

Teste	Com escalonador	Sem escalonador	%
200x200 200 8	75.246537	96.368888	28.07
200x200 200 16	61.275706	73.273138	19.58
200x200 200 32	35.144835	36.797331	4.70
400x400 800 8	338.633775	350.517335	3.51
400x400 800 16	221.731239	253.955017	14.55
400x400 800 32	172.066703	175.328044	1.90
800x800 3200 8	1142.628404	1419.704419	24.25
800x800 3200 16	722.262216	957.527562	32.57
800x800 3200 32	677.301333	708.872481	4.66

Considerando novamente o exemplo da Figura 7.1, com 200 tarefas, no caso de existir quatro nodos em uso, a divisão das partes será feita respeitando a sequência 0, 50, 100, 150, 1, 51, 101, ...

O propósito desta análise é mostrar em qual situação o escalonador obtem o melhor desempenho.

A Figura 7.8 mostra os resultados obtidos com esta técnica.

O desempenho do escalonador é muito superior neste caso, pois com esta forma de distribuição, a implementação sem o escalonador coloca no nodo 1 a partes da faixa 50 a 99 e no nodo 2 as partes da faixa 100 até a 150, justamente as partes mais complexas. Com isto o nodo 1 e nodo 2 foram seriamente prejudicados pela divisão de tarefas.

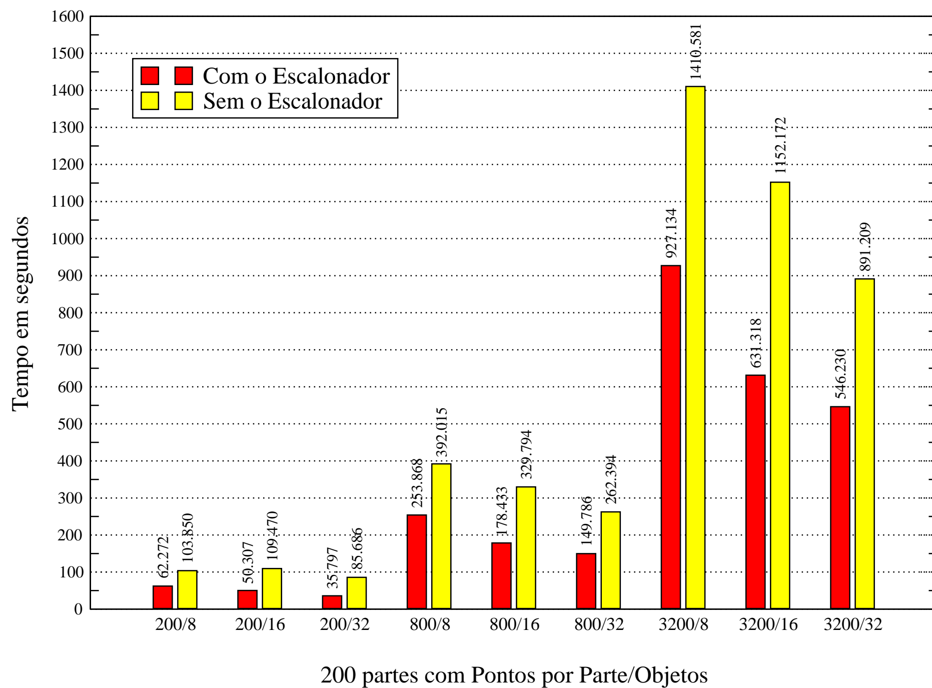

FIGURA 7.8 – Desempenho com Distribuição Otimizada

O teste com injeção de carga obteve resultados onde a implementação sem escalonamento ultrapassou quatro vezes o tempo que o modelo com o escalonador obteve.

7.4 Considerações Finais

Outros testes de avaliação de desempenho foram realizados com várias combinações diferentes de configurações.

Testes com carga muito baixa, utilizando valores de convergência de cada ponto entre 100 e 400 foram realizados, porém descartados. Como a aplicação inteira levou menos de 2 segundos para terminar, muitas delas terminando em décimos de segundo, os resultados obtidos não eram confiáveis.

Cada tarefa, nestas circunstâncias, torna-se um processo *I/O Bound*, pois grande parte do tempo envolvido na execução é de responsabilidade da comunicação envolvida. Os resultados tornaram-se sem sentido, pois pequenas variações na largura de banda da rede influenciavam de forma decisiva no desempenho.

Testes com valores extremamente altos para convergência também foram realizados e alguns levaram horas para terminar, sendo que outros foram abortados por *timeout*. Pelo fato de algumas amostragens terem sido perdidas e o comportamento destes testes seguir a mesma lógica mostrada nos casos com carga alta, descartou-se estes resultados.

8 Conclusões

Nesta dissertação de mestrado foi desenvolvida uma ferramenta de escalonamento de processos como parte do projeto DPC++. Este projeto, em constante atualização e modificação pelo Grupo de Processamento Paralelo e Distribuído, tem por objetivo desenvolver uma linguagem distribuída de programação orientada a objetos.

O DPC++ possibilita criar objetos em outros nodos de forma semelhante à programação Orientada a Objetos. Esta criação, de forma remota e transparente, é gerenciada pelo DPC++ através de um objeto especial chamado "Diretório". Toda a criação e destruição de algum objeto distribuído deve ser requisitada ao "Diretório", em execução no nodo principal.

Esta centralização do DPC++ justificou a implementação de uma ferramenta de escalonamento centralizada. Um algoritimo de escalonamento centralizado se caracteriza por executar em apenas um nodo, centralizando todas as decisões de escalonamento, ao contrário de um algoritmo distribuído, onde todos os nodos dividem este poder de decisão.

A ferramenta também se caracteriza como sendo dinâmica e adaptativa. Um algoritmo dinâmico possui a capacidade de possuir dados recentes sobre cada um dos nodos envolvidos na aplicação. Para alimentar o escalonador com os dados necessários, a ferramenta fez uso de objetos espiões, que ficam constantemente avaliando a carga e remetendo informações ao escalonador. A frequência com que os dados serão enviados para o escalonador pode ser aumentada ou diminuída, gerando mais ou menos comunicação, caracterizando a ferramenta como sendo adaptativa.

O escalonador melhora o desempenho de uma aplicação de carga irregular. Muitas aplicações não terão benefícios com o uso do escalonador, principalmente aplicações que criam todos os seus objetos no início ou fazem uma divisão de dados.

Para a availiação da ferramenta de escalonamento, buscou-se uma aplicação que tivesse como característica principal a irregularidade em sua execução, de forma a aproveitar os recursos oferecidos pelo escalonador. Observou-se que a geração de fractais de *MandelBrot* se enquadra nestas condições, uma vez que que o tempo gasto para calcular um ponto da imagem pode ser até 2000 vezes maior que o tempo gasto para calcular outro ponto. Tipicamente, pontos centrais de um fractal precisam de mais tempo para calcular, consumindo mais recursos de processamento.

Os resultados obtidos podem ser divididos em dois grupos distintos:

- com cargas baixas: neste caso o *overhead* gerado pelos espiões e a comunicação entre espião e escalonador tiveram uma importância significativa no desempenho, de forma que o escalonador apresentou resultados não satisfatórios.
- com cargas altas: o escalonador mostrou-se muito eficiente com cargas altas e variadas. Variadas no sentido de que cada tarefa possui características diferentes.

Com testes de pouca carga, o algoritmo não se mostrou muito eficiente, com um ganho pouco significativo e, para alguns casos, inferior à solução sem o escalonador. Isto se deve aos custos de comunicação existentes no modelo que oneram aplicações que executam em poucos segundos. Estas aplicações não se beneficiam de

algoritmos de escalonamento e seu uso não se justifica nestes casos. Mas com aplicações irregulares e de carga alta, o algoritmo de escalonamento mostrou-se muito eficiente.

Obteve-se ganhos acima de 30% quando a carga da aplicação exigiu alguns minutos para ser calculada em um cluster com oito processadores. Em situações de altíssima carga, com a aplicação levando horas para executar, os testes sem o escalonador foram cerca de quatro vezes mais lentos que a solução empregando o escalonador. Isto demonstra a eficiência do algoritmo de escalonamento nos casos de carga elevada para os quais ele foi desenvolvido.

Como trabalhos futuros, pode-se citar a construção de uma nova versão do compilador usando a concorrência entre métodos para suportar o escalonador desenvolvido. A ferramenta de escalonamento implementada será incorporada neste compilador, permitindo que o usuário desenvolva aplicações DPC++ com o escalonamento dinâmico. O escalonameno poderá ser ativado pelo usuário quando a carga justificar o seu uso. Além disto, o escalonador poderá ser portado para uma futura versão Java do DPC++.

Bibliografia

[AMZ 96] AMZA, C. et al. Treadmarks: shared memory computing on networks of workstations. **IEEE Computer**, Los Alamitos, v.29, n.2, p.18–28, Feb. 1996.

[SBA 97] SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES—PROCESSAMENTO DE ALTO DESEMPENHO, 9., 1997, Campos do Jordão. **Anais...** São Paulo: Escola Politécnica da USP, 1997.

[ÁVI 99] ÁVILA, R. B. et al. Modelagem e avaliação de desempenho de agregados conectados por tecnologia SCI. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH-PERFORMANCE COMPUTING, 11., 1999, Natal, RN. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1999. p.107–112.

[ÁVI 99a] ÁVILA, R. B. **Um modelo de paralelismo de grão fino para objetos distribuídos**. 1999. 75p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAR 98] 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 application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén, Argentina. **Trabajos Seleccionados...** Neuquén: Universidad Nacional del Comahue, 1998. v.2, p.623–637.

[BAR 2000] BARRETO, M. et al. Implementation of the DECK environment with BIP. In: MYRINET USER GROUP CONFERENCE, 1., 2000, Lyon, France. **Proceedings...** Lyon: INRIA Rocquencourt, 2000. p.82–88.

[BOD 95] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29–36, Feb. 1995.

[BUY 99] BUYYA, R. (Ed.). **High performance cluster computing**: programming and applications. Upper Saddle River: Prentice Hall PTR, 1999. 664p.

[CAS 88] CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-porpose distributed computing systems. **IEEE Transactions on Software Engeneering**, New York, v.14, n.2, p.141–154, Feb. 1988.

[CAS 99] CASTRO DUTRA, I. de; COSTA, V. S. Using compile-time granularity information to support dynamic work distribution in parallel logic programming systems. In: SYMPOSIUM ON COMPUTER

ARCHITECTURE AND HIGH-PERFORMANCE COMPUTING, 11., 1999, Natal, RN. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1999. p.248–254.

[CAV 95] CAVALHEIRO, G. G. H. et al. DPC++: an object-oriented distributed language. In: CONFERENCIA INTERNACIONAL DE LA SOCIEDAD CHILENA DE CIENCIA DE LA COMPUTACIÓN, 15., 1995, Arica, Chile. **Actas...** Santiago: Sociedad Chilena de Ciencia de la Computación, 1995. p.92–103.

[CAV 93] CAVALHEIRO, G. G. H.; NAVAUX, P. O. A. DPC++: uma linguagem para processamento distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES—PROCESSAMENTO DE ALTO DESEMPENHO, 5., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. v.2, p.732–744.

[CAV 94] CAVALHEIRO, G. G. H. **Um modelo para linguagens orientadas a objetos distribuído**. 1994. 145p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CLE 87] CLEMENT, M. **Unix internals**: a systems operation handbook. [S.l.]: TAB Books, 1987.

[CLU 2001] CLUSTERS TOP 500. Disponível em: <http://clusters.top500.org>. Acesso em: jun. 2001.

[DER 2002] DE ROSE, C. A. F.; NAVAUX, P. O. A. Fundamentos de processamento de alto desempenho. In: ESCOLA REGIONAL DE ALTO DESEMPENHO , 2., 2002, São Leopoldo, RS. **Anais...** Porto Alegre: SBC, 2002. p.3–29.

[DEL 2001] DELAWARE, U. O. **SAMson**: 132 node linux beowulf cluster. Disponível em: <http://www.bartol.udel.edu/Mri/sam>. Acesso em: jun. 2001.

[EFE 95] EFE, K.; KRISHNAMOORTHY, V. Optimal scheduling of compute-intensive tasks on a network of workstations. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.6, n.6, p.668–673, June 1995.

[ELR 94] EL-REWINI, H. **Task scheduling in parallel and distributed systems**. New Jersey: Prentice Hall, 1994.

[GEI 94] GEIST, A. et al. **PVM**: parallel virtual machine. Cambridge: MIT Press, 1994.

[HAC 90] HAC, A.; JOHNSON, T. J. Sensitivity study of the load balancing algorithm in a distributed system. **Journal of Parallel and Distributed Computing**, Orlando, v.10, p.85–89, 1990.

[IEE 92] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Information Technology—Standard for Scalable Coherent Interface (SCI)**, IEEE 1596-1992. New York, 1992.

[KIT 95] KITAJIMA, J. P. F. W. **Programação paralela utilizando mensagens**. Porto Alegre: Instituto de Informática da UFRGS, 1995.

[KIT 96] KITAJIMA, J. P. Arquitetura de computadores: tendências até o final do século. In: ESCOLA REGIONAL DE INFORMÁTICA, 4., 1996, Canoas, RS. **Anais...** Porto Alegre: SBC, 1996. p.92–109.

[KRU 94] KRUEGER, P.; SHIVARATRI, N. G. Adaptative location policies for global scheduling. **IEEE Transactions on Software Engeneering**, New York, v.20, n.6, p.432–444, June 1994.

[LAU 97] LAURIA, M.; CHIEN, A. MPI-FM: high performance MPI on workstation clusters. **Journal of Parallel and Distributed Computing**, Orlando, FL, v.40, n.1, p.4–18, Jan. 1997.

[MAN 82] MANDELBROT, B. B. **The fractal geometry of nature**. New York: W. E. Freeman and Company, 1982.

[MPI 94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[MPI 97] MPI FORUM. **MPI-2 extensions to the message-passing interface**. Knoxville: University of Tennessee, 1997.

[OLI 2001] OLIVEIRA, F. A. D. de. **Uma biblioteca para programação paralela por troca de mensagens de clusters baseados na tecnologia SCI**. 2001. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[OLI 98] OLIVEIRA JUNIOR, E. R. d.; NAVAUX, P. O. A. Replicação de objetos em um sistema distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES—PROCESSAMENTO DE ALTO DESEMPENHO, 10., 1998, Búzios. **Anais...** Rio de Janeiro: COPPE/UFRJ, 1998. p.157–160.

[OLI 99] OLIVEIRA JUNIOR, E. R. d. **Replicação de objetos distribuídos no DPC++**. 1999. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[OPE 2001] OPENMP. **OpenMP specifications**. Disponível em: <http://www.openmp.org/specs>. Acesso em: jul. 2001.

[PAD 2001] PADERBORN, U. of. **Cluster siemens hpcline**. Disponível em: <http://www.uni-paderborn.de/pc2/services/systems/psc>. Acesso em: jun. 2001.

[PAD 2001a] PADERBORN, U. of. **CCS**: computing center software. Disponível em: <http://www.uni-paderborn.de/pc2/projects/ccs>. Acesso em: jun. 2001.

[PIL 97] PILLA, M. L. et al. Mecanismo de tolerância a falhas para a linguagem distribuída DPC++. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES—PROCESSAMENTO DE ALTO DESEMPENHO, 9., 1997, Campos do Jordão. **Anais...** São Paulo: Escola Politécnica da USP, 1997. p.139–152.

[SBA 99] SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH-PERFORMANCE COMPUTING, 11., 1999, Natal, RN. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1999.

[RÍM 97] RÍMOLO, G. S.; ÁRABE, J. N. C. Balanceamento de carga em um ambiente distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES—PROCESSAMENTO DE ALTO DESEMPENHO, 9., 1997, Campos do Jordão. **Anais...** São Paulo: Escola Politécnica da USP, 1997. p.413–429.

[SIN 94] SINGHAL, M.; SHIVARATRI, N. G. **Advanced concepts in operating systems**. New York: McGraw-Hill, 1994.

[TAN 96] TANDIARY, F.; KOTHARI, S. C. Batrun: utilization idle workstations for large-scale computing. **IEEE Parallel & Distributed Technology**, New York, v.4, n.2, p.41–48, Sept. 1996.

[TAN 92] TANENBAUM, A. S. **Modern operating systems**. Upper Saddle River: Prentice-Hall, 1992.

[WIL 93] WILLEBEEK-LEMAIR, M. H.; REEVES, A. P. Strategies for dynameic load balancing on highly parallel computers. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.4, n.9, p.979–992, Sept. 1993.