Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Escalonamento Estático de Processos de Aplicações Paralelas MPI em Máquinas Agregadas Heterogêneas com Auxílio de Históricos de Monitoração

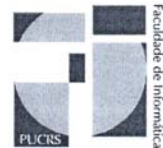
Augusto Mecking Caringi

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. César A. F. De Rose

Porto Alegre
2006

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.



TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Escalonamento Estático de Processos de Aplicações Paralelas MPI em Máquinas Agregadas Heterogêneas com Auxílio de Históricos de Monitoração***", apresentada por Augusto Mecking Caringi, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 19/01/2006 pela Comissão Examinadora:

Prof. Dr. César Augusto Fonticielha De Rose – Orientador — PPGCC/PUCRS

Prof. Dr. Avelino Francisco Zorzo – PPGCC/PUCRS

Prof. Dr. Luiz Gustavo Leão Fernandes – PPGCC/PUCRS

Prof. Dr. Adenauer Correa Yamin – UCPel

Homologada em 28/06/06, conforme Ata No. 17 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.pucrs.br/inf/pos e-mail: ppgcc@inf.pucrs.br

Pontifícia Universidade Católica do Rio Grande do Sul

Dados Internacionais de Catalogação na Publicação (CIP)

C277e Caringi, Augusto Mecking
Escalonamento estático de processos de aplicações paralelas MPI em máquinas agregadas heterogêneas com auxílio de históricos de monitoração / Augusto Mecking Caringi. – Porto Alegre, 2006.
88 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS
Orientador: Prof. Dr. César A. F. De Rose.

1. Processamento Paralelo. 2. Escalonamento (Informática).
3. Sistemas Distribuídos. 4. Informática. I. De Rose, César Augusto Fonticielha. II. Título.

CDD 004.35

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

Dedico este trabalho a minha família e amigos.

Agradecimentos

Agradeço a todos aqueles que me apoiaram nesta jornada.

Resumo

Em um sistema de processamento paralelo heterogêneo, a redução do tempo de resposta das aplicações pode ser alcançada se for levada em consideração a natureza heterogênea do ambiente computacional. Este trabalho enquadra-se neste contexto e descreve o modelo cujo objetivo é otimizar o desempenho de aplicações paralelas MPI executadas sobre máquinas agregadas heterogêneas. Para isto, desenvolve-se uma estratégia de escalonamento global dos processos que compõem a aplicação, a qual visa realizar um mapeamento equilibrado de processos aos nós no início da execução (estático), de modo a balancear a carga e tendo por consequência a minimização do tempo de execução. Isto se dá de forma transparente ao usuário e é gradativamente refinado ao longo das execuções da aplicação, através de um "ciclo de adaptação" apoiado pela análise automática de informações de monitoração obtidas em execuções prévias da mesma. Para avaliar o modelo, foi desenvolvida uma ferramenta que implementa o método proposto. Esta ferramenta foi instalada e configurada no Centro de Pesquisa em Alto Desempenho (CPAD) localizado na PUCRS e uma análise de algumas aplicações paralelas executadas através da ferramenta, no agregado principal do CPAD, é apresentada.

Palavras-chave: Escalonamento. Balanceamento de carga. Máquinas Agregadas Heterogêneas. Monitoração.

Abstract

In a heterogeneous parallel processing system, the reduction of the parallel application's response time can be achieved if the computational environment's heterogeneous nature is taken in consideration. This work fits in this context and describes the model whose goal is to optimize the performance of MPI parallel applications executed on heterogeneous clusters. A strategy for global scheduling of the application's processes was developed, which aims at realizing a balanced mapping of process to nodes in the beginning of the execution (static), in order to balance the load and, by consequence, minimizing the execution time. The process is transparent to the user and is gradually refined during the application's executions through an "adaptation cycle" supported by the automatic analysis of previously acquired monitored information. To evaluate the model, we developed a tool which implements the proposed method. This tool was installed and configured in the Research Center in High Performance Computing (CPAD) located at PUCRS and a analysis of some parallel applications executed through the tool in CPADs main cluster are presented.

Keywords: Scheduling. Load Balancing. Heterogeneous Clusters. Monitoring.

Lista de Figuras

Figura 1 Arquitetura de uma máquina agregada 24
Figura 2 Taxonomia de estratégias de escalonamento 31
Figura 3 Níveis de escalonamento em um agregado 35
Figura 4 Classificação das técnicas de otimização em sistemas heterogêneos . . . 36
Figura 5 Monitoração de aplicação de cálculo do PI 46
Figura 6 Estratégia de escalonamento utilizada 47
Figura 7 Arquitetura do modelo 49
Figura 8 Diagrama da Base de Dados 57
Figura 9 Tela do tempo das execuções de uma aplicação paralela 58
Figura 10 Tela da análise detalhada de uma execução 59
Figura 11 Tela da comparação de execuções da mesma aplicação 59
Figura 12 Tempo das execuções da aplicação de cálculo do PI 68
Figura 13 Comparação entre pior e melhor execução da aplicação de cálculo do PI 68
Figura 14 Tempo das execuções da aplicação N-Gramas 70
Figura 15 Comparação entre pior e melhor execução da aplicação N-Gramas . . . 70
Figura 16 Tempo das execuções do *benchmark* SP 71
Figura 17 Comparação entre pior e melhor execução do *benchmark* SP 72
Figura 18 Tempo das execuções do *benchmark* EP 73
Figura 19 Comparação entre pior e melhor execução do *benchmark* EP 73
Figura 20 Tempo das execuções do *benchmark* IS 74
Figura 21 Comparação entre pior e melhor execução do *benchmark* IS 75
Figura 22 Tempo das execuções da aplicação POV-Ray 76
Figura 23 Comparação entre pior e melhor execução da aplicação POV-Ray . . . 76

Lista de Tabelas

Tabela 1 Características das principais arquiteturas paralelas 21
Tabela 2 Comparativo das abordagens de otimização de desempenho 42
Tabela 3 Número de processos por tipo de nó da aplicação de cálculo do PI . . . 67
Tabela 4 Número de processos por tipo de nó da aplicação N-Gramas 69
Tabela 5 Número de processos por tipo de nó do *benchmark* SP 71
Tabela 6 Número de processos por tipo de nó do *benchmark* EP 73
Tabela 7 Número de processos por tipo de nó do *benchmark* IS 74
Tabela 8 Número de processos por tipo de nó da aplicação POV-Ray 76
Tabela 9 Variação de tempo das aplicações em execuções com mesmos parâmetros 77
Tabela 10 Comparativo das aplicações paralelas testadas 77

Lista de Abreviaturas

API	Application Programming Interface	54
CC-NUMA	Cache-Coherent Nonuniform Memory Access	21
CFD	Computational Fluid Dynamics	70
CPAD	Centro de Pesquisa em Alto Desempenho	18
CPU	Central Processing Unit	45
DSM	Distributed Shared Memory	21
E/S	Entrada/Saída	21
IA32	Intel Architecture 32 bits	29
IA64	Intel Architecture 64 bits	29
IPC	Interprocess Communication	21
IPF	Itanium Processor Family	66
LAN	Local Area Network	22
MFLOPS	Millions of Floating-point Operations per Second	37
MPI	Message Passing Interface	17
MPP	Massively Parallel Processors	15
NORMA	Non-Remote Memory Access	16
NPB	NAS Parallel Benchmarks	70
POV-Ray	Persistence of Vision Ray-Tracer	67
PVM	Parallel Virtual Machine	27
RPC	Remote Procedure Call	21
SCI	Scalable Coherent Interface	16
SMP	Symmetrical Multi Processing	15
SO	Sistema Operacional	21
SPMD	Single-Program Multiple-Data	27
TCP	Transport Control Protocol	55
XDR	eXternal Data Representation	29

Sumário

1 Introdução . . . 15
1.1 Contexto do trabalho . . . 18
1.2 Principais contribuições deste trabalho . . . 18
1.3 Estrutura do texto . . . 19

2 Máquinas agregadas . . . 21
2.1 Arquitetura . . . 23
2.2 *Hardware* . . . 23
2.2.1 Rede de interconexão . . . 24
2.3 *Software* . . . 25
2.3.1 Sistema Operacional . . . 25
2.3.2 Sistemas de gerenciamento de recursos e monitoração . . . 25
2.3.3 Ambientes de programação paralela . . . 26
2.4 Agregados Heterogêneos . . . 28
2.4.1 Suporte da biblioteca MPICH a agregados heterogêneos . . . 29

3 Escalonamento em sistemas distribuídos . . . 30
3.1 Taxonomia de escalonamento . . . 30
3.2 Classificação hierárquica . . . 30
3.2.1 Escalonamento local versus global . . . 31
3.2.2 Escalonamento estático versus dinâmico . . . 31
3.2.3 Escalonamento ótimo versus sub-ótimo . . . 32
3.2.4 Escalonamento aproximado versus heurístico . . . 32
3.2.5 Soluções dinâmicas . . . 32
3.2.6 Escalonamento global distribuído versus não-distribuído . . . 33
3.2.7 Escalonamento cooperativo versus não-cooperativo . . . 33
3.3 Classificação não-hierárquica . . . 33
3.3.1 Escalonamento adaptativo versus não-adaptativo . . . 33
3.3.2 Balanceamento de carga . . . 33
3.4 Escalonamento dinâmico versus adaptativo . . . 34
3.5 Política versus mecanismo de escalonamento . . . 34
3.6 Relação entre balanceamento de carga e escalonamento . . . 34
3.7 Escalonamento em máquinas agregadas . . . 35

4 Estado da arte . . . 36
4.1 Técnicas dinâmicas . . . 37
4.1.1 DRUM . . . 37
4.1.2 MOSIX . . . 38
4.2 Técnicas estáticas . . . 39
4.2.1 Balanceamento de carga assimétrico . . . 40
4.2.2 Sputnik . . . 40
4.2.3 Otimizador de desempenho por força bruta desenvolvido no CPAD . . . 41
4.3 Comparação das abordagens . . . 42

5 Escalonamento estático auxiliado por históricos de monitoração . . . 44
5.1 Motivação . . . 44
5.1.1 Definição do problema . . . 45
5.2 Proposta . . . 45
5.2.1 Estratégia de escalonamento utilizada . . . 46
5.3 Objetivos do modelo . . . 48
5.4 Forma de avaliação . . . 48
5.5 Arquitetura do modelo . . . 48
5.6 Algoritmo de cálculo do número de processos por nó . . . 50
5.6.1 Aplicações com restrições quanto ao número de processos . . . 52
5.6.2 Ciclo de adaptação . . . 53

6 Ferramenta DAPSched . . . 54
6.1 Sistema de monitoração . . . 54
6.1.1 Módulo agente . . . 54
6.1.2 Módulo coletor . . . 55
6.1.3 Módulo monitor de execuções . . . 56
6.2 Base de dados . . . 56
6.3 Interface web . . . 57
6.4 Disparador de aplicações . . . 59
6.4.1 Arquivo de Máquinas . . . 61
6.5 Utilização da ferramenta . . . 62

7 Testes e resultados atingidos . . . 63
7.1 Classificação de aplicações paralelas . . . 63
7.1.1 Aplicações mestre/escravo . . . 64
7.1.2 Aplicações SPMD . . . 65
7.1.3 Aplicações *pipelining* . . . 65
7.1.4 Aplicações de divisão e conquista . . . 65
7.1.5 Aplicações de paralelismo especulativo . . . 66
7.2 Metodologia empregada nos testes . . . 66
7.3 Aplicações selecionadas . . . 67
7.3.1 Aplicação para cálculo do número PI . . . 67
7.3.2 Aplicação N-Gramas . . . 68

7.3.3 *NAS Parallel Benchmarks* 70
7.3.4 **POV-Ray** 75
7.4 **Variabilidade dos resultados** 77
7.5 **Considerações sobre os testes** 77

8 **Conclusão e trabalhos futuros** 79

Anexo A **Código da aplicação de cálculo do PI** 81

Anexo B *Script* **SQL de especificação do banco de dados** 82

Referências 85

1 Introdução

Atualmente, o Processamento de Alto Desempenho é considerado uma ferramenta fundamental para as áreas de ciência e tecnologia, especialmente em campos como Bioinformática, Meteorologia, Química e Física, entre outros. Sua importância estratégica é caracterizada pela quantidade de iniciativas em pesquisa e desenvolvimento, nesta área, financiadas por governos e organizações de todo o mundo. Um exemplo é a definição nos Estados Unidos de um conjunto de aplicações prioritárias - *Grand Challenge Applications* [1], ou aplicações de grande desafio - as quais possuem grande impacto econômico e científico e têm uma demanda muito grande por alto desempenho computacional, exigindo muitos investimentos do governo para resolvê-las.

O processamento de alto desempenho, contudo, nos dias atuais, está fortemente dependente de técnicas e conceitos provindos do processamento paralelo e distribuído, a fim de prover o desempenho necessário a estas aplicações. O processamento paralelo e distribuído, em essência, consiste em juntar dois ou mais processadores, fazendo-os trabalhar em conjunto para resolver o mesmo problema [2].

Os sistemas de alto desempenho disponíveis atualmente abrangem desde processadores fortemente acoplados, conhecidos como MPPs (*Massively Parallel Processors*), até conjuntos de computadores interligados por redes convencionais. A contínua inovação na tecnologia de *hardware* reduz o tempo de comunicação entre os processadores e a latência de acesso à hierarquia de memória para, desta forma, suportar a computação neste novo ambiente. Infelizmente, a infraestrutura de *software* (e.g. linguagens de programação, compiladores, sistemas operacionais e ferramentas de otmização de desempenho) disponível nos dias de hoje ainda não acompanha o estado da arte do multiprocessamento em *hardware*.

Desde a década de 1990, vem-se observando uma tendência na substituição dos caros supercomputadores com arquiteturas proprietárias por agregados. Dessa forma, um agregado ou máquina agregada - em inglês conhecido como *cluster* - é atualmente uma das arquiteturas mais utilizadas na obtenção de alto desempenho. Máquinas agregadas são sistemas computacionais de alto desempenho construídos a partir de componentes de prateleira. Os computadores, chamados nós, podem ser servidores SMP (*Symmetrical Multiprocessors*), estações ou simples computadores pessoais, interligados através de uma rede de interconexão, que trabalham coletivamente como um sistema único, apresentando desempenho equivalente a um supercomputador. A utilização de máquinas agregadas traz alguns benefícios, como:

- Ótima relação custo-benefício: As máquinas agregadas, por serem construídas com "componentes de prateleira", tornam-se muito mais baratas do que supercomputadores comer-

ciais e podem possuir o desempenho aproximado de um sistema MPP, por exemplo, o qual pode ter um custo várias vezes maior;

- Baixo custo de manutenção: Com componentes de prateleira produzidos em grande escala, pode ser feita a substituição de peças com um baixo custo;

- Escalabilidade: Com a utilização dessas máquinas, tem-se uma capacidade de expansão muito grande, bastando "agregar" mais nós à medida das necessidades.

A rede de interconexão interliga todos os nós do agregado. Ela pode ser uma rede padrão como Ethernet ou Fast-Ethernet, ou uma rede de baixa latência como Myrinet [3] ou SCI (*Scalable Coherent Interface*) [4]. A utilização de uma rede de baixa latência leva a uma melhoria dos resultados devido à redução do gargalo de rede, o que é comum em muitas das aplicações paralelas que utilizam intensivamente troca de mensagens, paradigma padrão em multicomputadores do tipo NORMA (*Non-Remote Memory Access*); no entanto, este tipo de rede possui custo relativamente elevado. Os nós, que são responsáveis pelo processamento da aplicação, normalmente não possuem periféricos, como monitor, mouse e teclado, devido a sua utilização específica para execução de aplicações paralelas [5].

Hoje em dia, a tendência crescente na construção de máquinas agregadas pode ser confirmada pela lista das 500 máquinas mais rápidas do mundo [1], a qual demonstra que o tipo de máquina que mais cresce são os agregados, já existindo algumas com milhares de nós. Além disso, atualmente, entre as dez primeiras máquinas do *ranking*, sete são agregados [6].

Entretanto, as vantagens trazidas pelas máquinas agregadas, tais como alta configuralibilidade e escalabilidade, trazem também alguns desafios para o desenvolvimento de ambientes, sistemas e aplicações utilizados neste tipo de máquina. Alguns dos principais desafios trazidos pelas máquinas agregadas são:

- Gerência complexa: Devido às facilidades de expansão do número de nós e dos tipos de nós que podem ser utilizados na construção de um agregado, a gerência destes recursos torna-se mais complexa, pois deve ser eficiente, independentemente do número de nós e do tipo de nós utilizados;

- Difícil programação: A programação em máquinas agregadas é tida como mais complicada do que em outras arquiteturas paralelas, pois é dependente da disponibilidade de ferramentas e ambientes adequados. Além disso, podemos destacar alguns desafios como: distribuição e particionamento dos dados, escalonamento, balanceamento de carga, tolerância a falhas, heterogeneidade, entre outros.

Em relação à heterogeneidade, quando um agregado é composto de nós idênticos, isto é, quando todos os nós possuem a mesma arquitetura e a mesma capacidade de processamento,

[1] http://www.top500.org

ele é denominado um agregado homogêneo; em contrapartida, quando os nós diferem entre si, ele é denominado um agregado heterogêneo.

Os agregados heterogêneos introduzem uma dificuldade adicional na sua programação, porque é necessário levar-se em consideração a existência de diferentes tipos de nós. Desse modo, para que o tempo de execução de uma aplicação paralela executando em um sistema computacional heterogêneo seja minimizado, é necessário haver um mapeamento otimizado das tarefas ou processos da aplicação aos processadores, levando-se em consideração suas diferentes capacidades de processamento [7].

Uma aplicação paralela MPI pode por si só realizar balanceamento de carga, isto é, distribuir os dados ou tarefas da aplicação de uma forma equilibrada, levando em consideração os fatores que geram desequilíbrio, como a natureza heterogênea da máquina onde ela está sendo executada. Contudo, isto impõe um aumento da complexidade no projeto da aplicação. Aplicações paralelas legadas também podem ser modificadas para realizarem balanceamento de carga, porém isto geralmente não é uma tarefa fácil, envolvendo mudanças significativas no projeto e na estrutura da aplicação [8].

Contudo, mesmo no caso das aplicação paralelas que não realizam balanceamento de carga, é possível otimizar o seu desempenho quando executadas em agregados heterogêneos, mesmo sem modificações no código da aplicação, através de mecanismos de escalonamento de processos e/ou balanceamento de carga que operem em nível de sistema. Um exemplo é o sistema MOSIX [9], que possui um mecanismo de migração de processos. Este mecanismo realiza escalonamento dinâmico de processos transparentemente, transferindo processos entre os nós de forma a balancear a carga total do agregado.

Para que esses mecanismos funcionem, é necessário que a aplicação seja quebrada em um número de partições maior do que o número de processadores físicos disponíveis [8]. No caso das aplicações MPI, isto pode ser realizado em nível de sistema, bastando, em princípio, que seja informado ao disparador de aplicações um número maior de processos do que o número de processadores do agregado. Desse modo, um sistema como o MOSIX seria capaz de mapear e remapear dinamicamente os processos, de forma a balancear a carga.

Entretanto, sistemas capazes de realizar migração dinâmica de processos não são usualmente utilizados em máquinas agregadas de propósito geral, pois possuem desvantagens, podendo-se destacar a sobrecarga causada por tais sistemas. Além disso, os sistemas de migração de processos existentes não suportam migração entre nós de arquiteturas diferentes, o que dificulta o seu uso quando a máquina agregada alvo é composta por nós pertencentes a mais de um tipo de arquitetura (IA32 e IA64 por exemplo). Observa-se, então, que se faz necessário um método capaz de otimizar o desempenho de aplicações paralelas genéricas executadas em agregados heterogêneos sem os complexos mecanismos de migração de processos e ao mesmo tempo sem necessidade de modificação ao código da aplicação.

1.1 Contexto do trabalho

O grande desafio da atualidade no campo do Processamento de Alto Desempenho é obter um melhor aproveitamento da capacidade de processamento das máquinas existentes. Para tal, quatro áreas principais devem ser atacadas: as arquiteturas paralelas; o gerenciamento dessas máquinas; as linguagens que consigam explorar melhor o paralelismo dos programas; e, por último, a obtenção de uma maior concorrência nos algoritmos [10]. O objeto deste trabalho é atacar o problema no campo do gerenciamento dessas máquinas.

O gerenciamento das máquinas ocupa-se principalmente com os aspectos de sistemas operacionais em ambientes com mais de uma máquina, tanto paralelos como distribuídos. Essa área compreende tanto a sincronização de processos, quanto aspectos de bloqueio, como mecanismos de atribuição de processos aos processadores, entre outros.

A redução do tempo de resposta de programas paralelos tem sido alvo de um grande esforço de pesquisa. O tempo de execução de um programa paralelo é definido pelo intervalo entre o instante em que o primeiro processador inicia a sua parte do processamento e o momento em que o último encerra a sua parte. Como visto, a adequada distribuição de tarefas aos diversos processadores é fundamental para a minimização do tempo de execução do programa paralelo e maximização da utilização dos recursos computacionais disponíveis. A identificação desta distribuição ideal de tarefas caracteriza o problema de balanceamento de carga [11]. O balanceamento de carga, por sua vez, pode ser obtido através de uma estratégia de escalonamento de processos distribuídos, atribuindo-se um número maior de processos aos nós mais rápidos e menos processos aos nós com menor capacidade de processamento.

Este trabalho se enquadra nesse contexto e busca explorar um modelo adaptativo de balanceamento de carga para aplicações paralelas MPI (*Message Passing Interface*) através do escalonamento estático de processos, baseando-se em informações de monitoração de execuções prévias da aplicação para balancear a carga entre os nós (número de processos por máquina) ao longo das execuções. O termo adaptativo refere-se à capacidade do escalonador de alterar os parâmetros de distribuição dos processos em resposta a um *feedback* que, neste caso, é representado pelas informações de monitoração das execuções anteriores da aplicação.

1.2 Principais contribuições deste trabalho

Este trabalho é encadeado em uma seqüência de outros que vêm sendo desenvolvidos no Centro de Pesquisa em Alto Desempenho (CPAD) da PUCRS, relacionados com gerência, monitoração e otimização de aplicações para máquinas agregadas, entre as suas principais contribuições, destacam-se:

- definição de um modelo para otimizar o desempenho de uma aplicação paralela MPI

genérica, executada sobre um agregado heterogêneo, de forma automática ao longo das execuções;

- elaboração de um algoritmo para calcular o número de processos que serão atribuídos a cada tipo de nó a cada nova execução da aplicação, baseando-se nos históricos de monitoração das execuções prévias;
- implementação de uma ferramenta baseada no modelo e no algoritmo propostos;
- integração da ferramenta ao *middleware* do agregado Amazônia;
- avaliação do modelo através da análise de testes envolvendo algumas aplicações paralelas selecionadas.

1.3 Estrutura do texto

O trabalho está organizado da seguinte forma:

- o Capítulo 2 apresenta primeiramente uma visão das principais arquiteturas paralelas atuais, para então focar-se nas máquinas agregadas, analisando-se tanto aspectos de *hardware* quando de *software*. Paralelamente, são trazidas à tona questões pertinentes ao tema deste trabalho, incluindo aspectos do padrão MPI, a definição de agregados heterogêneos e o suporte da biblioteca MPICH a este tipo de agregado;
- o Capítulo 3 apresenta um estudo sobre escalonamento em sistemas distribuídos, traçando conceitos básicos e analisando uma taxonomia de escalonamento amplamente aceita;
- o Capítulo 4 analisa o estado da arte em relação à otimização de desempenho de aplicações paralelas em ambientes computacionais heterogêneos;
- o Capítulo 5 apresenta os principais fatores que motivaram o desenvolvimento deste trabalho, resultando na atual proposta do mesmo. Além disso, são também apresentados os objetivos do modelo, juntamente com a sua descrição, incluindo arquitetura básica e aspectos relacionados.
- o Capítulo 6 relata os aspectos de implementação da ferramenta baseada no modelo proposto;
- o Capítulo 7 relata a avaliação do modelo em questão a partir da análise do comportamento de algumas aplicações paralelas selecionadas quando executadas através da ferramenta implementada. Também é realizado um breve estudo sobre os diferentes tipos de aplicações paralelas existentes, a fim de ter-se um maior entendimento do comportamento observado nas aplicações testadas.

- o Capítulo 8, por último, conclui o trabalho, analisando vantagens e desvantagens, e também apresentando algumas idéias para trabalhos futuros.

2 Máquinas agregadas

Este capítulo apresenta inicialmente uma visão das principais arquiteturas paralelas atuais, para então aprofundar-se no tópico relativo às máquinas agregadas, incluindo tanto aspectos de *hardware* quando de *software*. Paralelamente, são focadas questões diretamente relacionadas ao tema deste trabalho.

Durante a última década, emergiram muitos sistemas computacionais diferentes suportando processamento de alto desempenho. A sua taxonomia é baseada na maneira como seus processadores, memória e interconexão são dispostos [12]. É possível destacar: processadores massivamente paralelos (MPP), multiprocessadores simétricos (SMP), sistemas com acesso à memória não uniforme e coerência de *cache* (CC-NUMA), sistemas distribuídos e máquinas agregadas. A Tabela 2.1 apresenta uma comparação entre estas arquiteturas.

Característica	*MPP*	*SMP/CC-NUMA*	*Cluster*	*Distribuído*
Número de nós	100-1000	10-100	100 ou menos	10-1000
Granularidade	grão fino ou médio	grão médio ou grosso	grão médio	muito grosso
Comunicação inter-nó	troca de mensagens, variáveis compartilhadas para DSM	memória centralizada ou DSM	troca de mensagens	arquivos compartilhados, RPC, troca de mensagens e IPC
Escalonamento de *jobs*	única fila de execução	única fila de execução	múltiplas filas coordenadas	filas independentes
SO do nó	N micro-kernel	SMP: um monolítico, NUMA: vários	N SO	N SO
Espaço de endereçamento	múltiplos - único para DSM	único	múltiplos ou único	múltiplo
Segurança inter-nó	desnecessária	desnecessária	necessária se exposta	necessária
Dono	uma organização	uma organização	uma ou mais organizações	várias organizações

Tabela 1 – Características das principais arquiteturas paralelas

Um MPP geralmente é um grande sistema de processamento paralelo, consistindo tipicamente de várias centenas de elementos processadores (nós) independentes, interconectados através de uma tecnologia de rede proprietária. Sistemas SMP, por sua vez, possuem entre 2 e 64 processadores, cada um deles compartilhando os recursos (barramento, memória, sistema de E/S) com todos os outros e executando uma única cópia do sistema operacional.

CC-NUMA propõe-se a ser uma arquitetura de multiprocessadores mais escalável que SMP, onde todos os processadores têm acesso a um mesmo espaço de endereçamento global, porém

cada um está mais perto de uma determinada região da memória com o objetivo de desafogar-se o barramento. Sistemas distribuídos, por sua vez, podem ser considerados, de forma genérica, as redes convencionais de computadores independentes, cada um com seu próprio sistema operacional.

Desde o início dos anos 90, tem-se percebido uma tendência de substituição dos caros e especializados supercomputadores paralelos proprietários (como MPPs) por supercomputadores construídos com máquinas convencionais (PCs, estações de trabalho, SMPs), interconectadas através de LAN (*Local Area Network*). Dentre as principais forças por trás desta transição, está a disponibilidade cada vez maior de componentes para computação de alto desempenho, que podem ser adquiridos pelo grande público.

Dessa forma, máquinas agregadas (*clusters*) [12] estão rapidamente tornando-se a plataforma padrão para computação de larga escala e de alto desempenho. O principal atrativo destes sistemas é que eles são construídos a partir de "componentes de prateleira", isto é, *hardware* de baixo custo, como máquinas baseadas em processadores Pentium e tecnologia de rede rápida como Myrinet, aliados a componentes de *software* amplamente aceitos, como o sistema operacional Unix e seus derivados, e ambientes de programação paralela, como MPI

Os computadores que formam a máquina agregada, chamados de nós, podem ser servidores SMP (*Symmetrical Multiprocessors*), estações de trabalho ou simples computadores pessoais que trabalham coletivamente como um sistema único, apresentando desempenho equivalente a um supercomputador. Conforme foi visto, a utilização de máquinas agregadas traz alguns benefícios, como:

- Ótima relação custo-benefício: As máquinas agregadas, por serem construídas com "componentes de prateleira", tornam-se muito mais baratas do que supercomputadores comerciais e podem possuir o desempenho aproximado de um sistema MPP (*Massively Parallel Processors*), por exemplo, o qual pode ter um custo várias vezes maior.

- Baixo custo de manutenção: Com componentes de prateleira produzidos em grande escala, pode ser feita a substituição de peças por um baixo custo.

- Escalabilidade: Com a utilização dessas máquinas, tem-se uma capacidade de expansão muito grande, bastando "agregar" mais nós à medida das necessidades.

Hoje em dia, a construção de máquinas agregadas tornou-se uma tendência, já existindo algumas com milhares de nós. A lista das 500 máquinas mais rápidas do mundo [1] confirma esta tendência, demonstrando que o tipo de máquina que mais cresce são os agregados. Além disso, atualmente, entre as dez primeiras máquinas do *ranking*, sete são agregados [6].

[1] http://www.top500.org

2.1 Arquitetura

Quanto a sua arquitetura, os sistemas paralelos atuais podem ser divididos em duas grandes classes: Multiprocessadores, sistemas com memória compartilhada, e Multicomputadores, sistemas com memória não-compartilhada [10]. Os agregados enquadram-se nesta segunda categoria.

Nos multiprocessadores, todos os processadores acessam, através de uma rede de interconexão, uma memória compartilhada. Nestas máquinas, a comunicação entre os processadores é realizada através de operações convencionais de *load/store* na memória.

Já nos multicomputadores, cada processador possui sua própria memória local e só é capaz de endereçar esta memória; devido a isto, estas máquinas também são chamadas de NORMA (*Non-remote Memory Access*). A comunicação entre os processadores é realizada através do mecanismo de troca de mensagens.

Desse modo, apesar de uma máquina agregada poder ser formada por multiprocessadores, o sistema como um todo é um multicomputador. Conforme a Figura 1, os principais componentes de um agregado são:

- múltiplos computadores, atualmente baseados, em geral, em processadores Intel Pentium ou Intel Itanium;
- sistema operacional, usualmente derivados do UNIX como Linux;
- rede de interconexão, abrangendo desde redes convencionais, como Ethernet, até redes de baixa latência, como Myrinet [3];
- protocolos de comunicação, como TCP/IP, ou protocolos de alto desempenho, como GM (*Glenn's Messages*);
- middleware, incluindo sistemas de gerência e monitoração;
- ambientes de programação paralela, como PVM [13] ou MPI [14];
- aplicações paralelas.

2.2 *Hardware*

A principal característica do *hardware* de um agregado é que ele é *hardware* comum, isto é, está amplamente disponível no mercado. O nós são geralmente PCs com microprocessadores Intel Pentium ou Itanium; entretanto, outras arquiteturas, como Digital Alpha, Sun SPARC e PowerPC, também são utilizadas.

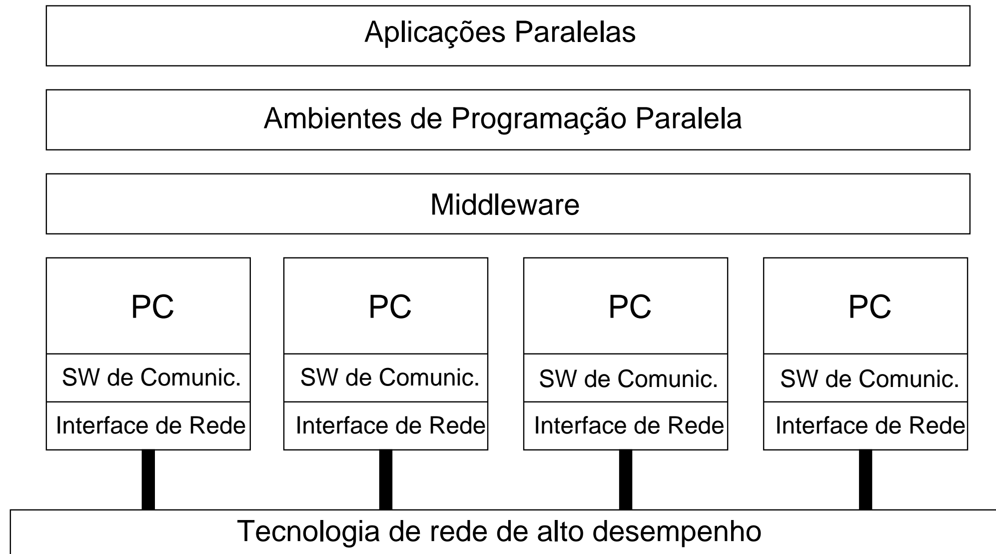


Figura 1 – Arquitetura de uma máquina agregada

2.2.1 Rede de interconexão

Os nós de uma máquina agregada comunicam-se através de uma rede de interconexão, que pode tanto ser uma rede padrão Ethernet como uma rede de alta velocidade, também caracterizadas por sua baixa latência, como Myrinet ou SCI. O padrão *Ethernet* é amplamente utilizado; no entanto, devido ao seu relativo baixo desempenho (alta latência e baixa vazão), ele vem dando lugar às redes de alto desempenho.

Ainda assim, mesmo com a adoção de redes rápidas, o padrão *Ethernet* muitas vezes é utilizado como rede secundária, em conjunto com a rede primária. A taxa de vazão de 10MBits/s do padrão *Ethernet* já não é mais suficiente para ambientes onde ocorrem transferências de grandes volumes de dados; dessa forma, um novo padrão denominado *Fast Ethernet* foi desenvolvido, aumentando a taxa de vazão para 100Mbit/s. Atualmente, a *Ethernet* em estado da arte é a *Gigabit Ethernet*, que, apesar de ter uma alta taxa de vazão, não apresenta a baixa latência necessária em um ambiente de computação de alto desempenho, onde ocorre a troca de muitas mensagens de pequeno tamanho, em vez de longas transferências de dados.

A utilização de redes padrão (Ethernet, Fast Ethernet) é uma tendência impulsionada por grandes fabricantes, como HP, IBM e Dell, os quais estão interessados na construção de máquinas paralelas poderosas, agregando milhares de estações de trabalho de baixo custo. Para interligação de tantas máquinas, fica muito caro investir em uma rede especial, sendo utilizada, então, uma rede padrão [10].

Já a tendência de utilização de redes de alto desempenho é impulsionada por pequenas empresas que fabricam placas de interconexão especificamente para máquinas agregadas. Estas placas implementam protocolos de rede de baixa latência, otimizados para as características de comunicação de aplicações paralelas. A principal desvantagem destas redes são o seu alto

custo, tornando-se muito cara a construção de máquinas com muitos nós (mais do que algumas centenas) [10].

Dentre as redes de baixa latência, destaca-se a rede Myrinet, consistindo em uma tecnologia de interconexão *full duplex* desenvolvida pela empresa Myricom [3]. Seu custo é relativamente alto se comparado com o padrão *Fast Ethernet*, mas as vantagens são muito grandes: latência muito baixa (5 microsegundos), vazão muito elevada (1.28Gbits/s), e um processador programável *on-board* que permite grande flexibilidade.

A principal máquina agregada do CPAD, chamada Amazônia, possui uma rede primária Myrinet e uma rede secundária Fast Ethernet; sua configuração detalhada encontra-se no capítulo 7.

2.3 *Software*

Os componentes de *software* que compõem o ambiente de um agregado podem ser divididos em duas grandes categorias: ferramentas de programação e sistemas de gerenciamento. Ferramentas de programação incluem linguagens, bibliotecas, depuradores. Sistemas de gerenciamento, por sua vez, incluem ferramentas de instalação e administração e mecanismos de escalonamento e alocação, tanto de *hardware* quanto de *software* [15].

2.3.1 Sistema Operacional

A grande maioria dos agregados utiliza sistemas operacionais derivados do UNIX como Solaris e principalmente Linux. Entre as vantagens oferecidas pelo sistema operacional Linux, destacam-se: suporte a muitas arquiteturas de *hardware*, liberdade de distribuição, disponibilidade do código fonte, grande comunidade de desenvolvedores trabalhando no projeto, etc.

Além disto, o Linux oferece características típicas presentes em um sistema UNIX, como multi-tarefa preemptiva, suporte multi-usuário, memória virtual, suporte a vários processadores, entre outras. Desta forma, torna-se uma tarefa relativamente fácil portar programas desenvolvidos em outras variantes do UNIX.

2.3.2 Sistemas de gerenciamento de recursos e monitoração

O sistema de gerenciamento de recursos é um dos *softwares* mais importantes de uma máquina agregada. Entre as principais funcionalidades providas pelos sistemas de gerência de recursos para agregados, destacam-se: alocação e liberação de recursos, execução de aplicações utilizando os recursos alocados de forma interativa, ou através de *batch jobs*, e gerência da

fila de alocação, através de uma política de alocação. Outras funcionalidades que podem ser providas pelos sistemas de gerência de recursos são: balanceamento de carga, migração de processos, suporte a várias políticas de alocação, gerência da fila de alocação baseado em prioridades, suspensão e recomeço de aplicações, entre outras. Entre os principais sistemas de gerência de recursos para agregados existentes atualmente, é possível destacar: OpenPBS, CCS, DQS e Condor. No CPAD, utiliza-se um gerenciador desenvolvido no próprio centro, chamado CRONO [16].

O gerenciador de recursos CRONO implementa somente os serviços básicos de gerenciamento necessários para compartilhar uma máquina agregada entre diversos usuários. O CRONO é otimizado para agregados de até médio porte (64 nós) e tem como principais características: suporte para gerenciamento concorrente de diversos agregados, suporte a alocações do tipo *space-shared* e *time-shared*, modo de execução interativo ou através de *batch jobs* e possibilidade de utilização de *scripts* de pré e pós processamento para configuração do ambiente. Além disso, o CRONO conta com um *script* que auxilia o disparo de aplicações paralelas, atuando como um *wrapper* para o *dispatcher* `mpirun`.

Outra classe de *softwares* muito importantes em um agregado são os sistemas de monitoração. Tais sistemas são projetados para coletar parâmetros de desempenho de sistemas, tais como: utilização de CPU dos nós, utilização de memória, taxa de E/S e interrupções, e apresentá-los de uma forma que possam ser facilmente entendidos pelo administrador do sistema [15]. Entre os principais sistemas de monitoração existentes, destacam-se: Co-Pilot, Ganglia e NWS.

O Centro de Pesquisa em Alto Desempenho da PUCRS possui um *software* de monitoração desenvolvido no próprio centro, chamado RVISION [17], o qual tem o intuito de prover uma arquitetura extensível e configurável para monitoração de agregados. Entre as principais características deste sistema é possível citar: (1) acesso às informações de utilização de recursos de forma independente para cada usuário e, (2) possibilidade de diversas sessões de monitoração estarem em execução paralelamente.

2.3.3 Ambientes de programação paralela

Apesar da possibilidade de utilização de outros modelos, o modelo de troca de mensagens é o mais amplamente utilizado para programação de máquinas agregadas [15]. Isto ocorre por ser este modelo diretamente mapeado à arquitetura com memória não compartilhada destes sistemas computacionais e, por consequência, é o que pode ser implementado de modo mais otimizado.

Na programação paralela através deste modelo de comunicação, uma aplicação consiste em uma coleção de processos autônomos, cada um possuindo sua própria memória local e comunicando-se com os demais processos através de funções de envio e recebimento de mensagens. Quando todos os processos executam o mesmo programa, porém cada um deles trabalha

em uma parte diferente dos dados, a aplicação é dita SPMD (*Single-Program Multiple Data* [2].

Desse modo, bibliotecas de troca de mensagens permitem o desenvolvimento de programas paralelos eficientes para sistemas com memória distribuída, Entre as bibliotecas existentes, as mais populares são PVM (*Parallel Virtual Machine*), desenvolvida pelo *Oak Ridge National Laboratory* [13] e MPI (*Message Passing Interface*) definida pelo *MPI Forum* [14].

PVM foi desenvolvida no Laboratório Nacional de Oak Rigde, nos Estados Unidos, com o objetivo de ser utilizada para a implementação de aplicações paralelas, tanto em supercomputadores de alto custo como em agregados.

Já o padrão MPI vem aos poucos substituindo o PVM. Ele foi proposto por um comitê que tinha por objetivo reunir as melhores características de cada ambiente de troca de mensagens desenvolvido até então. Tendo como metas de projeto eficiência, portabilidade e funcionalidade, o padrão somente define a biblioteca de passagem de mensagens, deixando a inicialização e configuração a cargo dos desenvolvedores.

Ambas as bibliotecas estão disponíveis para as principais linguagens utilizadas para programação paralela: Fortran 77, Fortran 90, ANSI C e C++. Atualmente, MPI tornou-se o padrão de fato para biblioteca de troca de mensagens, sendo esta a biblioteca tomada por base neste trabalho.

Padrão MPI

Conforme visto, o padrão MPI foi definido com o objetivo de reunir as melhores características dos sistemas de troca de mensagem mais populares existentes [12]. Resultado do trabalho do MPI-Forum, suas metas de projeto foram portabilidade, eficiência e funcionalidade. Atualmente, ele é amplamente aceito e existem implementações da biblioteca MPI para os mais variados tipos de sistemas paralelos.

O padrão restringe-se a definir uma biblioteca de troca de mensagens, deixando a cargo dos implementadores definir a forma como é realizada a criação e controle dos processos. Porém um fator é importante: MPI trabalha com um grupo fixo de processos, os quais são disparados no início da execução. No enfoque deste trabalho, essas operações de gerenciamento de processos, especialmente a forma como ocorre a criação dos mesmos, é um dos aspectos mais importantes da programação paralela.

A criação de processos pode ser estática ou dinâmica. Na criação estática, o conjunto de processos que irão compor o programa paralelo deve ser definido antes da execução e não pode ser alterado até o final do programa. Já na criação dinâmica, os processos podem ser instanciados a qualquer momento durante a execução do programa.

Cada processo criado em um programa paralelo deve ser associado a algum processador do sistema. Esta distribuição adequada de processos aos processadores, que consiste em um problema de escalonamento, algumas vezes também é chamada na literatura de mapeamento [18]. Quando a criação de processos é estática, o mapeamento é definido no momento da carga do

programa na máquina paralela. Por outro lado, quando os processos são criados dinamicamente, estes são distribuídos aos nós em tempo de execução.

O novo padrão MPI-2 [19] permite o gerenciamento dinâmico de processos, contando com a possibilidade de criação de novos processos em tempo de execução; contudo o presente trabalho não prevê este modelo de programação, tendo como foco realizar a distribuição de processos aos processadores alocados pelo usuário através de uma estratégia de escalonamento estática.

Biblioteca MPICH

MPICH [20] é uma implementação portável e de livre distribuição da especificação MPI, aqui comentada por ser a implementação MPI primariamente utilizada no CPAD. A biblioteca MPICH foi desenvolvida através de uma parceria entre o *Argonne National Laboratory* e a *Mississippi State University*, também contando com contribuições da IBM. Seu desenvolvimento deu-se de forma paralela à especificação do padrão MPI, de forma a permitir aos membros do Forum MPI avaliar a viabilidade de suas idéias. As letras "CH" do nome significam "*Chameleon*" - Camaleão - símbolo da adaptação, representando um dos símbolos do projeto: a alta portabilidade da biblioteca.

A arquitetura da biblioteca MPICH é dividida em duas partes, uma implementação MPI independente de plataforma, chamada de camada MPICH, e uma parte dependente de plataforma, denominada Interface de Dispositivo Abstrato - *Abstract Device Interface* (ADI). Desse modo, para portar a biblioteca para uma nova plataforma, somente a implementação ADI precisa ser desenvolvida, enquanto a camada MPICH depende somente da camada ADI e é completamente independente de sua implementação.

2.4 Agregados Heterogêneos

Esta seção define o conceito de agregado heterogêneo. Posto que um agregado homogêneo é todo aquele que possui nós idênticos, por outro lado, um agregado heterogêneo pode ser definido como todo aquele onde os nós não são exatamente iguais, isto é, podem não possuir:

- mesma arquitetura;
- mesmo sistema operacional (incluindo o fato de utilizarem a mesma versão do SO);
- mesmos componentes de *software* como bibliotecas e ferramentas (incluindo o fato de serem da mesma versão).

O primeiro requisito - mesma arquitetura - é um pouco questionável, no sentido do que realmente compõe arquiteturas diferentes. Por exemplo, um agregado composto de dois tipos

diferentes de nós, porém ambos baseados no mesmo processador Pentium 4, mas com diferentes quantidades de memória RAM e diferentes *clocks*, pode tanto ser visto como um sistema homogêneo quanto heterogêneo dependendo do conceito utilizado para defini-lo.

Em um sentido amplo, este pode ser considerado um agregado homogêneo, uma vez que aplicações compiladas em um nó podem ser executadas nativamente em outro. Entretanto, neste trabalho adota-se um sentido mais restrito quanto à homogeneidade, considerando-se o exemplo citado como sendo um sistema heterogêneo, porque os nós possuem diferentes capacidades de processamento.

O agregado principal do CPAD, chamado Amazônia - que tem sua configuração detalhada no capítulo 7 - é um sistema heterogêneo, sendo composto por 4 tipos diferentes de nós, pertencentes a 2 arquiteturas distintas (IA32 e IA64).

É interessante observar que mesmo agregados homogêneos tendem a tornar-se heterogêneos ao longo do tempo devido aos *upgrades* incrementais, onde novos nós mais poderosos vão sendo agregados ao sistema [21]. Este é o caso do *cluster* Amazônia, que inicialmente era um sistema homogêneo.

2.4.1 Suporte da biblioteca MPICH a agregados heterogêneos

O suporte aqui referido diz respeito à capacidade da biblioteca de trabalhar com arquiteturas diferentes, e a biblioteca MPICH possui tal capacidade. Desse modo, a conversão de tipos de dados (*little endian/big endian*) é parte integrante dos serviços da biblioteca.

Em muitos casos a biblioteca realiza apenas uma inversão da ordem dos bytes (*byte swapping*) ou extensão dos tipos de dados (de 32 para 64 bits). Somente números de ponto flutuante requerem tratamento especial, nestes casos o formato XDR (*eXternal Data Representation*) pode ser utilizado quando a norma IEEE não é respeitada [20].

Em relação a utilização de múltiplos binários, em um agregado composto por nós das arquiteturas IA32 e IA64, por exemplo, o *dispatcher* (disparador de aplicações) pode ser informado para utilizar ambas arquiteturas através dos parâmetros `-arch` e `-np`. Desse modo, para executar uma aplicação paralela em 4 nós IA32 e 4 nós IA64, disparando-se um processo por cada nó, poderia-se empregar o comando: `mpirun -arch ia32 -np 4 -arch ia64 -np 4 program.%a`

O nome especial da aplicação "program.%a" permite que sejam especificados diferentes executáveis, devido à impossibilidade de termos um binário que execute sobre as duas arquiteturas. O "%a" é substituído automaticamente pelo nome da arquitetura. Neste exemplo, program.ia32 executa nos nós IA32 e program.ia64 executa nos nós IA64.

3 Escalonamento em sistemas distribuídos

Este capítulo apresenta um estudo sobre escalonamento em sistemas distribuídos, traçando conceitos básicos e analisando uma taxonomia de escalonamento amplamente aceita. O termo “sistemas distribuídos” é aqui empregado de forma ampla, referindo-se a sistemas compostos por múltiplos elementos processadores.

Escalonamento consiste em atribuir processos a processadores e determinar em que ordem estes processos serão executados. Ele é de extrema importância para sistemas paralelos e distribuídos, podendo ser considerado um dos problemas mais desafiantes nesta área [22].

Conforme Casavant [23], escalonamento, de forma genérica, pode ser visto como um problema de gerenciamento de recursos. Uma política ou mecanismo de gerenciamento de recursos, por sua vez, tem por objetivo gerenciar eficientemente o acesso e/ou uso de recursos como processadores, processos, etc.

3.1 Taxonomia de escalonamento

Existem vários tipos de escalonamentos e algumas taxonomias foram propostas [22–24]. Nenhuma destas taxonomias abrange todas as características que um escalonador pode possuir. Dentre as taxonomias estudadas, a proposta por Casavant ([23]) no artigo *“A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems”* é a mais aceita e completa.

O autor acima citado define duas classificações distintas, a primeira delas hierárquica, e uma segunda classificação não-hierárquica, a qual agrega as características que podem ser encontradas em qualquer tipo de sistema de escalonamento, independente da primeira classificação.

3.2 Classificação hierárquica

A classificação hierárquica de escalonamento em sistemas distribuídos, proposta por Casavant, é apresentada na Figura 2. A seguir, temos uma descrição detalhada desta classificação.

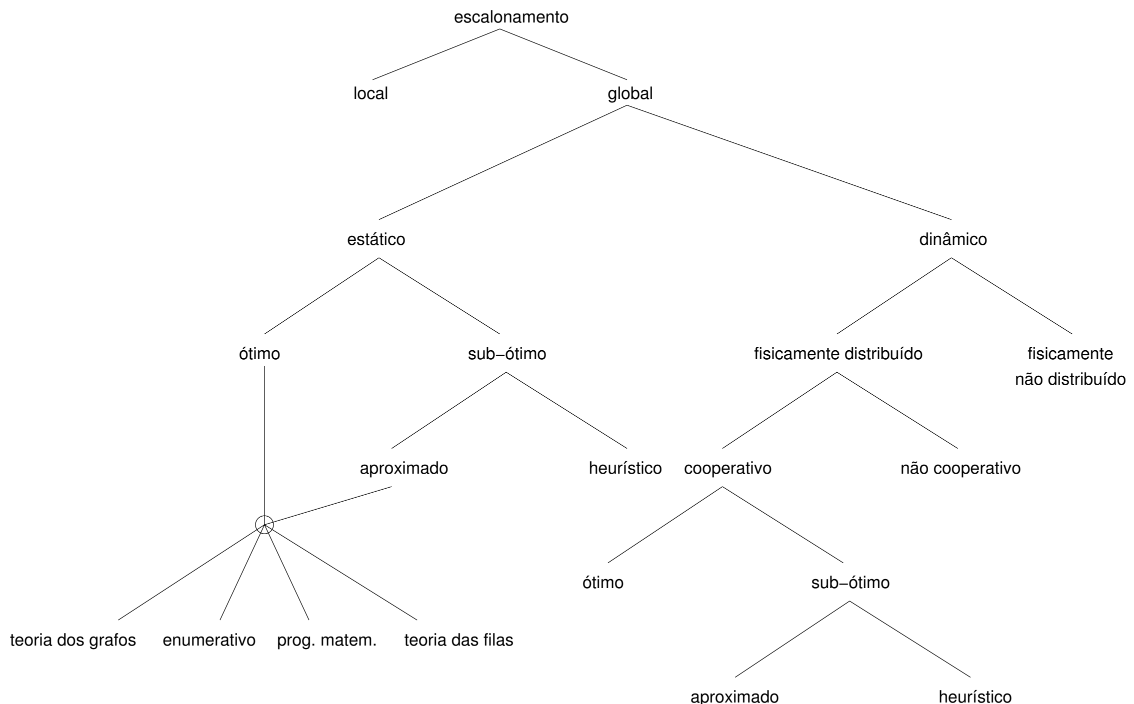


Figura 2 – Taxonomia de estratégias de escalonamento

3.2.1 Escalonamento local versus global

No nível mais alto, pode-se distinguir entre escalonamento local e global [23]. Escalonamento local refere-se à atribuição de fatias de tempo aos processos em um sistema computacional isolado. Escalonamento global (distribuído), por sua vez, é o problema de decidir **o**nde executar um processo em um sistema computacional distribuído.

Em uma máquina agregada, o trabalho de escalonamento local é realizado pelo sistema operacional de cada nó. Enquanto o escalonamento global é definido pelo ambiente de programação utilizado.

3.2.2 Escalonamento estático versus dinâmico

O próximo nível na hierarquia (abaixo de escalonamento global) divide-se entre escalonamento estático e dinâmico. Estas duas formas de escalonamento referem-se ao momento em que as decisões são tomadas.

No caso do escalonamento estático, a distribuição de processos é realizada apenas e somente no momento da execução. Já nos modelos dinâmicos, decisões de escalonamento podem ser tomadas em tempo de execução; em máquinas agregadas, isto pode ser suportado, por exemplo, através de mecanismos de migração de processos.

3.2.3 Escalonamento ótimo versus sub-ótimo

De acordo com [23], caso toda a informação sobre o estado do sistema, como comportamento da aplicação e os recursos disponíveis, sejam conhecidos de antemão, é possível realizar um escalonamento dito ótimo baseado em alguma função objetivo, que pode ser, por exemplo: Minimização do tempo total de execução, maximização da utilização dos recursos disponíveis ou maximização da vazão do sistema.

Em contrapartida, caso não seja possível conhecer estes dados antecipadamente, soluções sub-ótimas devem ser tentadas. Entre as abordagens sub-ótimas, existem duas categorias básicas: aproximadas e heurísticas, conforme veremos a seguir.

3.2.4 Escalonamento aproximado versus heurístico

Nas estratégias de escalonamento aproximadas, a idéia é utilizar modelos formais do algoritmo da aplicação, mas ao invés de procurar todo espaço de soluções possíveis em busca da solução ótima, satisfazer-se com boas soluções que possam ser encontradas de forma rápida. Entre os principais modelos utilizados para isto, destacam-se teoria dos grafos, programação matemática e teoria das filas.

As estratégias heurísticas, por sua vez, representam a categoria que baseia-se (1) na utilização dos parâmetros mais realísticos tendo por base o conhecimento *a priori* sobre a aplicação ou (2) na utilização de algoritmos que tentam de alguma forma aprender o comportamento da aplicação, para então escaloná-la da melhor maneira possível.

3.2.5 Soluções dinâmicas

A principal motivação por trás das técnicas de escalonamento dinâmicas é o fato que em muitos casos tem-se pouco conhecimento sobre as características de uma aplicação de antemão. Também pode ser desconhecido em quais ambientes a aplicação será executada. Com as técnicas estáticas, as decisões de escalonamento precisam ser feitas antes que a aplicação seja executada (ou logo no início da execução) enquanto nas técnicas dinâmicas, nenhuma decisão precisa ser tomada *a priori*, pois é responsabilidade do sistema de execução decidir onde executar um processo. No projeto de um sistema que realiza escalonamento global dinâmico, a primeira questão importante é definir onde serão tomadas as decisões, o que leva a próxima subdivisão:

3.2.6 Escalonamento global distribuído versus não-distribuído

Este aspecto diz respeito a quem é atribuída a responsabilidade pelo escalonamento dinâmico global, que pode residir fisicamente em um único processador ou pode ser fisicamente distribuída entre os processadores participantes. Resumidamente, esta característica refere-se a quem possui a autoridade para tomar decisões.

3.2.7 Escalonamento cooperativo versus não-cooperativo

No âmbito do escalonamento global dinâmico, ainda é possível distinguir entre aqueles mecanismos que envolvem cooperação entre os componentes distribuídos e aqueles em que processadores individuais tomam decisões de forma independente da ação dos outros processadores. A questão principal aqui é relacionada ao grau de autonomia que cada processador possui para determinar como seus recursos devem ser usados.

3.3 Classificação não-hierárquica

Em adição a parte hierárquica da taxonomia analisada acima, Casavant destaca uma série de características que podem ser encontradas em qualquer sistema de escalonamento, independente desta primeira classificação. A seguir são descritas algumas dessas principais características:

3.3.1 Escalonamento adaptativo versus não-adaptativo

Um algoritmo de escalonamento adaptativo é aquele que altera dinamicamente o seu comportamento (seja o algoritmo ou os parâmetros utilizados), baseado nos comportamentos atual e prévios do sistema. De modo oposto, um escalonador não-adaptativo é aquele que não modifica seus mecanismos de controle baseado no histórico de atividade do sistema.

3.3.2 Balanceamento de carga

A taxonomia proposta por Casavant enquadra o balanceamento de carga como sendo uma das várias características básicas disponíveis em um sistema de escalonamento. As estratégias que se enquadram nesta categoria tentam, de alguma forma, balancear a carga entre os processadores, de forma que todos os processos progridam no mesmo ritmo. Conforme [23], esta é

uma característica presente em um grande número de sistemas de escalonamento distribuído.

3.4 Escalonamento dinâmico versus adaptativo

Conforme [23], parece haver na literatura uma certa falta de consenso sobre o emprego dos termos "dinâmico" e "adaptativo" em relação a estratégias de escalonamento.

O termo "dinâmico" refere-se ao momento em que o escalonamento pode ser realizado, neste caso, a capacidade do escalonador de tomar decisões durante a execução de uma aplicação. Já o termo "adaptativo" representa a capacidade do escalonador de alterar políticas, parâmetros ou algoritmos em resposta a um *feedback*.

3.5 Política versus mecanismo de escalonamento

Mecanismos determinam como fazer algo, políticas decidem o que será feito. A separação de política e mecanismo é um princípio muito importante e permite a máxima flexibilidade se decisões políticas precisam ser alteradas posteriormente [25].

Uma das formas de alcançar-se isso é o desenvolvimento de um algoritmo de escalonamento parametrizado, de forma que o os parâmetros (política) possam ser alterados pelo usuário conforme necessário, sem necessidade de modificação no mecanismo em si.

3.6 Relação entre balanceamento de carga e escalonamento

Na área de computação, o termo "balanceamento de carga" significa distribuir carga de trabalho (processamento, requisições, usuários, etc) uniformemente entre muitos processos, computadores, discos ou outro tipo de recurso, de forma que nenhum elemento fique sobrecarregado.

Em [26], balanceamento de carga é definido como: "Dada uma coleção de tarefas e um conjunto de computadores nos quais estas tarefas podem ser executadas, encontrar o mapeamento de tarefas para computadores que resulta em cada computador possuir uma fatia de trabalho aproximadamente igual".

Escalonamento, por sua vez, consiste em atribuir tarefas a um conjunto de recursos. Desse modo, escalonamento pode ser visto como um meio para se alcançar balanceamento de carga.

3.7 Escalonamento em máquinas agregadas

Em uma visão ampla, é possível identificar que o escalonamento em uma máquina agregada ocorre em três níveis distintos. Pode-se considerar que a distribuição de processos aos nós (escalonamento global) ocorre em um nível intermediário. Antes disso, deve ocorrer a alocação de recursos, tarefa esta do escalonador de recursos. Por outro lado, após o escalonamento global, os vários processos que podem vir a ser disparados em um mesmo nó são escalonados através do escalonador local do sistema operacional. A figura 3 ilustra estes três níveis.

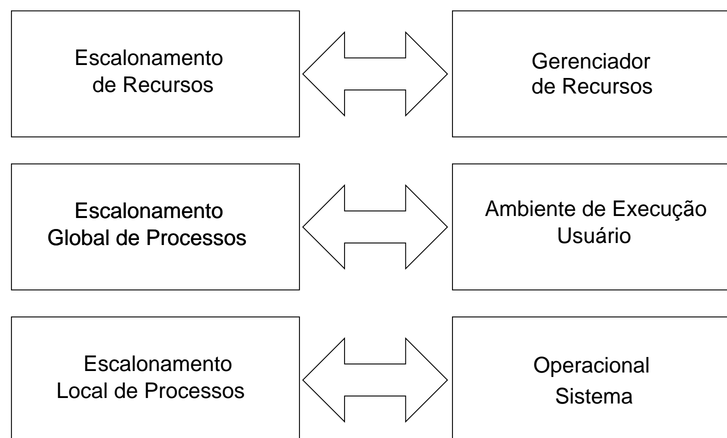


Figura 3 – Níveis de escalonamento em um agregado

- Escalonamento de Recursos: É o escalonamento (também chamado de alocação) de processadores aos usuários realizado pelo sistema de gerenciamento de recursos.
- Escalonamento Global de Processos: Escalonamento de distribuição de processos aos processadores alocados ao usuário.
- Escalonamento Local de Processos: É o escalonamento que é realizado pelo sistema operacional de cada nó.

No ambiente em que o presente trabalho está sendo desenvolvido, o escalonamento de recursos fica a cargo do gerenciador CRONO [16] e o escalonamento local é trabalho do escalonador de processos do sistema operacional Linux presente em cada nó. O escalonamento global de processos, contudo, não conta com nenhum *software* especial, sendo, até então, responsabilidade do usuário definir o mapeamento de processos para processadores através de parâmetros passados ao sistema de execução MPI, caso o usuário não esteja satisfeito com o modo padrão com o qual os processos são atribuídos aos processadores. O modo padrão consiste em, baseando-se na lista de nós que irão compor a execução e no número de processos passado pelo usuário, atribuir um processo a cada processador e cada vez que o fim da lista for atingido, retornar ao início.

4 Estado da arte

Este capítulo apresenta um estudo sobre o estado da arte no âmbito da otimização de desempenho de aplicações paralelas em sistemas computacionais heterogêneos.

Na literatura encontra-se uma série de trabalhos relacionados com a otimização do desempenho de aplicações paralelas executando em ambientes computacionais heterogêneos [9, 21, 27–33]. Em uma visão ampla, as técnicas estudadas podem ser classificadas de acordo com dois critérios:

- Momento da distribuição (estático ou dinâmico);
- Nível de atuação (aplicação ou sistema).

Enquanto os aspectos relacionados ao momento da distribuição já foram abordados no capítulo anterior, o nível de atuação refere-se ao modo como a técnica é aplicada para resolver o problema. Quando a solução é em nível de aplicação, isto implica em alterações no código fonte para que a aplicação paralela suporte o ambiente heterogêneo; já quando a solução se dá em nível de sistema, a aplicação não necessita ser modificada, pois é deixado a cargo do sistema de execução ou do sistema operacional o trabalho de otimizá-la para o sistema computacional heterogêneo subjacente.

Desse modo, diferentes combinações das técnicas (estática ou dinâmica, em nível de aplicação ou em nível de sistema) são possíveis, conforme a Figura 4.

	Aplicação	**Sistema**
Estático	Técnicas estáticas em nível de aplicação	Técnicas estáticas em nível de sistema
Dinâmico	Técnicas dinâmicas em nível de aplicação	Técnicas dinâmicas em nível de sistema

Figura 4 – Classificação das técnicas de otimização em sistemas heterogêneos

Devido a estreita relação entre balanceamento de carga e escalonamento, alguns trabalhos existentes encaram o problema como balanceamento de carga e outros como escalonamento.

Geralmente quando o problema é tratado como balanceamento de carga, são discutidos modelos de distribuição dos dados levando-se em consideração a natureza heterogênea do sistema computacional utilizado, isto é, o problema é atacado em nível de aplicação. Já quando o problema é tratado como escalonamento, significa que a otimização do desempenho da aplicação é alcançada através da distribuição de processos, isto é, em nível de sistema.

4.1 Técnicas dinâmicas

As técnicas dinâmicas dividem-se em (1) modelos em nível de aplicação para realização de escalonamento/balanceamento de carga dinâmico dos dados da aplicação e, (2) modelos que atuam em nível de sistema. Estes representam os sistemas de escalonamento dinâmico de processos, os quais realizam migração de processos. Um mecanismo deste tipo permite que processos sejam suspensos, movidos e então reiniciados em outro nó [12].

4.1.1 DRUM

O sistema DRUM (*Dynamic Resource Utilization Model*) [33] é utilizado para criar um modelo dinâmico do ambiente de execução, capturando a dinâmica e a estrutura do agregado heterogêneo, com o objetivo de ajudar na realização do balanceamento de carga.

O modelo encapsula recursos de *hardware*, suas capacidades e a sua topologia de interconexão em uma estrutura de árvore, junto a isto, provê um mecanismo para monitoração e avaliação dinâmica destes recursos. Estes monitores trabalham de forma concorrente às aplicações do usuário, coletando estatísticas de utilização de memória, tráfego de rede e utilização de processador. A informação provinda destes monitores é utilizada para o cálculo de um índice unificado de desempenho, chamado, no modelo DRUM, de “*power*”. Este índice, então, pode ser utilizado por sistemas parametrizados de balanceamento de carga, como Zoltan Toolkit [34], para realizar partições de tamanho não-uniforme.

Este modelo em forma de árvore do ambiente de execução é especificado através de um arquivo de configuração XML, o qual contém uma lista de nós e a descrição de sua topologia de interconexão. Então, inicialmente, é obtida a capacidade de computação de ponto flutuante de cada nó (MFLOPS) através do *benchmark* LINPACK [35]. Porém este índice tem por objetivo apenas dar uma idéia da capacidade dos nós.

Isto decorre do fato que este tipo de *benchmark* não é capaz de refletir de forma precisa as características de uma aplicação em particular. Desse modo, o sistema oferece ao programador funções para calibrar a aplicação para o agregado heterogêneo, usando o mesmo tipo de computação que será empregada na aplicação real. A função `startMonitoring()` lança a *thread* agente, enquanto `stopMonitoring()` finaliza a monitoração dinâmica. Estes agen-

tes de monitoração capturam informações relativas ao uso de memória, processador e tráfego de rede dos nós. As estatísticas são então combinadas com o *benchmark* inicial para obter uma estimativa dinâmica do poder de processamento.

Após o término da monitoração, a função `computePowers()` atualiza o poder de cada nó no modelo. Estes índices, que são obtidos através da função `getLocalPartSize()`, podem então serem utilizados para realizar o particionamento dos dados no procedimento de balanceamento de carga dinâmico.

4.1.2 MOSIX

MOSIX (*Multicomputer Operating System for UnIX*) [9] é uma extensão para o *kernel* Linux que provê balanceamento de carga adaptativo dinâmico entre máquinas Linux x86. O sistema utiliza migração de processos preemptiva para atribuir e migrar processos entre os nós de forma a maximizar a utilização dos recursos computacionais. Já foram desenvolvidas diversas versões para vários derivados do UNIX, atualmente na sétima versão, ele é baseado em Linux. A tecnologia MOSIX consiste em duas partes: um conjunto de algoritmos adaptativos de balanceamento de carga e um mecanismo preemptivo de migração de processos (PPM). Ambas as partes são implementadas em nível de núcleo, através de módulos recarregáveis.

O algoritmo de balanceamento de carga responde a variações no uso dos recursos do *cluster*, como por exemplo, desbalanceamento da distribuição de carga ou uso excessivo do disco devido a falta de memória em algum dos nós. Nestes casos, o MOSIX inicia a migração de processos de um nó para outro de forma a balancear a carga do agregado.

A granularidade da distribuição de trabalho no MOSIX é o processo. Usuários podem executar aplicações paralelas através da inicialização de múltiplos processos em um mesmo nó para então permitir que o sistema atribua estes processos aos nós com a menor carga no momento.

O recurso de migração de processos é transparente para as aplicações, isto significa que é possível executar aplicações seqüenciais e paralelas da mesma forma que se faria em um SMP. O usuário não precisa se preocupar sobre a localização em que um processo está executando. Após a criação de um novo processo, o MOSIX tenta atribuir a ele o nó com a menor carga no momento, feito isto, o sistema continua a monitorar o novo processo, bem como todos os outros, e irá migrá-los sempre que necessário para maximizar o desempenho geral do sistema. Tudo isto é implementado sem modificações à interface das chamadas de sistema já existentes no Linux, desta forma, o usuário continua vendo e controlando todos os seus processos como se eles estivessem rodando no nó em que foram criados.

Para implementar o recurso de migração, o processo migrado é divido em dois contextos [9]: o contexto de usuário, que corresponde a parte do processo que é efetivamente migrada e o contexto de sistema, o qual é dependente do nó onde foi criado e nunca é migrado. O contexto de usuário, chamado de “remoto” contém o código do programa, pilha, dados, mapas de memória

e registradores do processo. O "remoto" encapsula o processo quando ele está executando em nível de usuário. O contexto de sistema, por sua vez, é chamado de "representante", no sentido que representa o processo no seu nó original. Ele contém a descrição dos recursos que estão em uso pelo processo, e uma pilha para execução de código do *kernel* em nome do processo. O "representante", dessa forma encapsula o processo quando ele está rodando em modo *kernel*.

A interface entre o contexto de usuário e o contexto de sistema é bem definida, entretanto é possível interceptar cada interação entre estes contextos e redirecioná-la através da rede. Isto é implementado através de uma camada de ligação. Desse modo, todas as chamadas de sistema executadas pelo "remoto" são interceptadas pela camada de ligação. Se a chamada de sistema é independente de nó ela é executada localmente (no nó remoto), caso contrário, a chamada é redirecionada ao "representante" executando no nó onde o processo foi gerado.

Migração manual de processos também é possível, isto pode ser útil para implementar uma política particular ou diferentes algoritmos de escalonamento. O super-usuário tem privilégios especiais em relação à migração, ele pode tanto definir novas políticas como determinar quais nós estão disponíveis para migração.

Os algoritmos que possibilitam a migração automática são descentralizados, cada nó é ao mesmo tempo mestre para os processos criados localmente e servidor para os processos que foram criados remotamente, isto é, que foram migrados de outros nós. Isto possibilita que nós sejam adicionados e removidos do agregado a qualquer momento, com distúrbios mínimos aos processos em execução. Outro recurso interessante do MOSIX são seus algoritmos de monitoração, que detectam a velocidade de cada nó, a carga e a memória livre, bem como as taxas de E/S e IPC para cada processo. Estas informações são utilizadas para decidir o nó de destino na hora da migração de um processo.

A imagem de sistema do MOSIX é baseada no modelo "*home-node*", isto significa que é dada a visão a um usuário de que todos os processos criados em um nó estão sempre sendo executados neste mesmo nó, deste modo, o processo permanece sempre ligado ao nó em que foi criado.

Como o MOSIX provê transparência total, ele pode ser utilizado em conjunto com ambientes de programação paralela como MPI ou PVM, fornecendo a estes ambientes um mecanismo de balanceamento de carga dinâmico [36].

4.2 Técnicas estáticas

As técnicas estáticas também dividem-se em modelos em nível de aplicação, que baseiam-se, por exemplo, em índices relativos de desempenho para realizar o escalonamento/balanceamento de carga ao início da execução e em técnicas em nível de sistema que, mesmo sem o dinamismo dos mecanismos de migração de processos, tentam otimizar o desempenho ao longo das execuções da aplicação, refinando a distribuição a cada nova execução.

4.2.1 Balanceamento de carga assimétrico

O trabalho "Asymmetric Load Balancing on Heterogeneous Cluster of PCs" [29] apresenta uma forma de resolver o problema baseada na obtenção de índices de desempenho relativo dos nós e modificações nas aplicações para que utilizem estes índices como base para o balanceamento de carga. O foco do artigo é definição do índice ideal de desempenho, não sendo apresentados detalhes de como o índice escolhido é aplicado na prática.

No artigo são analisadas algumas técnicas para obter-se o desempenho relativo de cada nó, que de acordo com o estudo precisam atender os seguintes requisitos:

- ser um bom indicador geral do desempenho do sistema;
- ser escalável e portável;
- ser computacionalmente leve.

Conhecidos *benchmarks* como LINPACK, NPB, SPEC e HINT são analisados. Como vantagem, eles são tidos como bons indicadores gerais de desempenho porém são computacionalmente pesados, especialmente se forem executados antes de cada execução da aplicação. Avalia-se então a utilização do indicador de desempenho calculado pelo Linux, uma vez que este é o sistema operacional utilizado no agregado. Este indicador de desempenho, chamado de "BogoMIPS", é utilizado para calibrar funções do *kernel* relacionadas a medição de tempo e, apesar deste indicador não ser considerado um bom indicador geral de desempenho, sua vantagem é o custo computacional para obtê-lo que é inexistente. Também é considerada a utilização de uma pequena rotina para cálculo do número de operações de ponto flutuante por segundo (MFLOPS), que também não é uma boa indicadora geral de desempenho, mas além de poder ser calculada de forma rápida, não depende de um sistema operacional em particular.

Os testes realizados envolvem uma versão modificada do *benchmark* LU, parte integrante do pacote NPB. Estas modificações incluem tanto as chamadas a biblioteca que calcula o desempenho relativo de cada nó como alterações na rotina de particionamento dos dados. Os resultados demonstraram ganho de desempenho de até 92%.

A abordagem apresentada neste artigo tem como principal ponto fraco a necessidade de modificações na aplicação que, como visto, não se limitam a apenas fazer a chamada a uma biblioteca, sendo preciso alterações no código que realiza o particionamento dos dados.

4.2.2 Sputnik

O projeto Sputnik [21] fornece um modelo de particionamento de dados automático destinado à máquinas agregadas heterogêneas. O sistema provê uma biblioteca de programação

implementada em C++, sobre a infra-estrutura KeLP [37], que permite tratar um agregado heterogêneo como se fosse um sistema computacional homogêneo. Para isso, o sistema baseia-se em um processo de dois estágios para executar aplicações paralelas.

O primeiro estágio, chamado *ClusterDiscovery* obtém o índice de desempenho relativo de cada máquina. Este índice não baseia-se em parâmetros de desempenho, como capacidade de processamento de ponto flutuante, e sim no resultado da execução da aplicação original com um pequeno conjunto de dados, individualmente, em cada um dos nós. Durante este estágio também são testados diferentes ajustes que podem ser realizados em busca de um maior desempenho.

Utilizando as medições obtidas pelo primeiro estágio, o segundo estágio, chamado *ClusterOptimizer*, particiona o conjunto de dados de maneira não uniforme, de acordo com a velocidade relativa de cada nó. O sistema então executa a aplicação utilizando o particionamento ótimo, além de outras otimizações realizadas no primeiro estágio específicas para cada nó. Todas as execuções subseqüentes realizam então este mesmo particionamento.

Entretanto, este sistema não é destinado a aplicações MPI genéricas. Ele foi desenvolvido com o intuito de fornecer suporte à agregados heterogêneos ao *framework* KeLP (*Kernel Lattice Parallelism*), o qual possibilita a implementação de aplicações científicas portáveis em sistemas computacionais paralelos com memória distribuída. KeLP facilita a programação de determinados tipos de aplicações, provendo suporte em tempo de execução à decomposição de dados baseada em blocos, e ao gerenciamento da comunicação.

4.2.3 Otimizador de desempenho por força bruta desenvolvido no CPAD

O presente trabalho teve origem a partir de pesquisas realizadas no CPAD relacionadas à otimização do desempenho de aplicações paralelas em máquinas agregadas heterogêneas orientadas a aplicações genéricas, isto é, sem modificações ao código da aplicação.

Esta abordagem inicial que foi desenvolvida no CPAD baseia-se na realização de múltiplas execuções da aplicação afim de encontrar, ao longo das execuções, a melhor configuração de processos por nó, ou seja, aquela que tenha por consequência o menor tempo de execução. A idéia básica é aumentar o número total de processos, apoiando-se na premissa que uma aplicação MPI deve ser projetada para distribuir a carga igualmente entre os processos. Com isto, é possível atribuir um número maior de processos aos nós mais poderosos e um menor número às máquinas menos velozes, permitindo, dessa forma, que cada nó receba uma carga compatível com sua capacidade de processamento.

Desse modo, o sistema implementado inicialmente testa, de forma automática, diferentes combinações do número de processos para cada tipo de nó, a fim de que a melhor configuração encontrada, após um determinado número de tentativas, seja utilizada efetivamente em execuções subseqüentes. O programa segue uma regra muito simples: aumentar gradativamente o número de processos nos nós mais rápidos, para então também aumentar o número de processos nos

nós menos poderosos, sempre guiando-se pelo tempo de execução para decidir o rumo da otimização.

Esta abordagem simples demonstrou que é possível efetivamente otimizar o desempenho de um grande número de aplicações através da distribuição equilibrada de processos aos nós que participam da execução. Ela baseia-se no fato que uma aplicação é executada várias vezes, especialmente quando está em fase de desenvolvimento, possibilitando que ao longo dessas execuções seja gradativamente otimizada.

4.3 Comparação das abordagens

A Tabela 2 apresenta uma comparação resumida das técnicas estudadas. Os modelos de otimização de desempenho de aplicações paralelas em ambientes heterogêneos que tratam do problema em nível de aplicação exigem modificações no programa, o que além de não ser transparente, não é genérico, pois cada aplicação precisa ser modificada conforme necessário.

Técnica	*Estático/Dinâmico*	*Nível*	*Características*
DRUM	dinâmico	aplicação	API/poder de cada nó
MOSIX	dinâmico	sistema	migração de processos
Balanceamento assimétrico	estático	aplicação	API/poder de cada nó
Sputnik	estático	aplicação	API/poder de cada nó
Otimizador CPAD	estático	sistema	cálculo do número de processos disparados em cada nó

Tabela 2 – Comparativo das abordagens de otimização de desempenho

Os trabalhos que tratam do problema em nível de sistema, por sua vez, costumam valer-se de ambientes que suportam migração de processos para realizar escalonamento dinâmico. Tais *softwares* são bastante interessantes, contudo também possuem desvantagens, como o custo associado tanto à migração quanto aos algoritmos de tomada de decisão dinâmica, além disso, torna-se difícil para tais sistemas migrar processos quando o sistema computacional paralelo é composto por máquinas de diferentes arquiteturas.

Na literatura encontra-se apenas um sistema capaz de fazer isto, entretanto ainda é bastante experimental. O sistema "Tui" ("*The Tui System*") [38], foi desenvolvido com o objetivo de permitir que as máquinas de origem e destino em uma migração de processos possuam arquiteturas diferentes, isto é, conjuntos de instruções e formatos de dados diferentes. Isto é alcançado através de técnicas avançadas de tradução de código binário, onde toda imagem do processo na memória é traduzida durante a migração. A versão analisada se propõem a suportar quatro arquiteturas (m68000, SPARC, i486 e PowerPC), exigindo que o programa tenha sido escrito em ANSI C.

Dessa forma, com exceção do trabalho desenvolvido no próprio CPAD, percebe-se a ausência de trabalhos que ataquem o problema em nível de sistema, sem a utilização de mecanismos especiais de migração de processos e dessa forma superando a dificuldade da migração entre arquiteturas de *hardware* distintas.

Acredita-se que a abordagem implementada no CPAD possa ser melhorada orientando-se a busca não só pelo tempo de execução da aplicação paralela, mas também pela análise do comportamento de execuções prévias da mesma aplicação. Valendo-se de informações de monitoração, a busca pela melhor configuração de número de processos por nó pode ser realizada de forma mais inteligente, eliminando-se tentativas que sabidamente teriam como consequência uma degradação do desempenho.

5 Escalonamento estático auxiliado por históricos de monitoração

Com base nos estudos realizados e descritos nos capítulos 2, 3 e 4, este capítulo apresenta os principais fatores que motivaram o desenvolvimento deste trabalho, resultando na proposta do mesmo. Além das motivações e da proposta, são também apresentados os objetivos do modelo, juntamente com a sua descrição, incluindo sua arquitetura básica e aspectos relacionados.

5.1 Motivação

Como visto, o tempo de execução de um programa paralelo é definido pelo intervalo entre o instante em que o primeiro processador inicia a sua parte do processamento e o momento em que o último encerra a sua parte. A adequada distribuição de tarefas aos diversos processadores é fundamental para a minimização do tempo de execução do programa paralelo e maximização da utilização dos recursos computacionais disponíveis.

Os agregados heterogêneos introduzem uma dificuldade adicional para atingir-se esta maximização dos recursos computacionais, isto porque é necessário levar-se em consideração a existência de diferentes tipos de nós com diferentes capacidades de processamento. Desse modo, para que o tempo de execução de uma aplicação paralela executando em um sistema computacional heterogêneo seja minimizado, é necessário haver um mapeamento otimizado das tarefas ou processos da aplicação aos processadores, levando-se em consideração suas diferentes capacidades de processamento [7].

Algumas aplicações paralelas são capazes de atingir uma maximização dos recursos computacionais mesmo em tais agregados, utilizando-se para isto de mecanismos de balanceamento de carga. Tais mecanismos implicam uma maior complexidade da aplicação e observa-se a existência de um grande número de aplicações que não implementam mecanismos desta natureza. Para essas, torna-se necessária uma "ajuda externa" a fim de que os recursos computacionais disponíveis sejam mais bem aproveitados.

5.1.1 Definição do problema

Para ilustrar mais claramente o problema, temos a seguir a análise do comportamento de uma pequena aplicação para cálculo do número PI (não realiza nenhum tipo de balanceamento de carga) executada na máquina agregada Amazônia do CPAD. Foi realizada a alocação através do CRONO de 3 nós, sendo 1 do tipo A, 1 do tipo B e 1 do tipo C. Devido ao objetivo de meramente ilustrar o problema, atribuiu-se aqui letras a cada tipo de nó em vez de dar-se sua descrição; entretanto, maiores informações sobre o agregado Amazônia podem ser encontradas no capítulo 7. Na execução, foi disparado um processo para cada processador (nós biprocessados receberam dois processos).

Através da monitoração da utilização da Unidade Central de Processamento - *Central Processing Unit* (CPU) - das máquinas alocadas, após a execução da aplicação é possível constatar que houve uma subutilização dos recursos disponíveis. Conforme a Figura 5, enquanto o nó do tipo B (que se percebe ter menor capacidade de processamento em relação aos outros) teve uma utilização média de CPU de 97%, o nó do tipo A apresentou uma utilização média de CPU de 52% e o nó do tipo C teve uma utilização média de CPU de apenas 23%. Com isto, concluímos que, se fosse possível, de alguma forma, realizar uma distribuição mais eqüitativa das tarefas, levando-se em consideração as diferentes capacidades de processamento de cada nó, seria possível realizar uma maximização dos recursos disponíveis com uma consequente minimização do tempo de execução.

5.2 Proposta

A proposta deste trabalho é definir um modelo para encontrar de forma automática e transparente ao usuário um mapeamento otimizado de processos de uma dada aplicação MPI genérica aos processadores alocados pelo usuário em uma máquina agregada heterogênea. Para alcançar este objetivo, foi especificada uma estratégia de escalonamento que é definida, de acordo com a classificação hierárquica proposta por Casavant, como global, estática, sub-ótima e heurística, conforme a Figura 6. Esta estratégia também caracteriza-se por ser adaptativa, utilizando-se de históricos de monitoração de execuções anteriores da aplicação para adaptar gradativamente (ao longo das execuções) a configuração de escalonamento. A justificativa destas decisões são analisadas na seção seguinte.

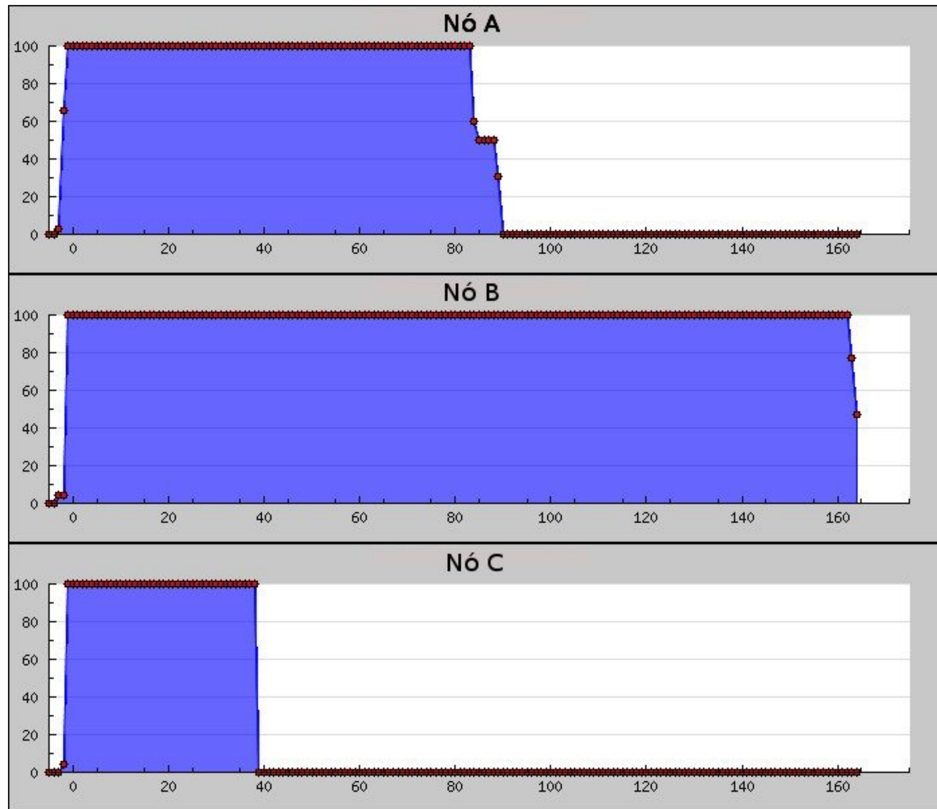


Figura 5 – Monitoração de aplicação de cálculo do PI

5.2.1 Estratégia de escalonamento utilizada

A primeira decisão, entre escalonamento local e global, é direta: O foco do trabalho é a realização de um escalonamento global dos processos, de modo a maximizar a utilização dos recursos computacionais disponíveis em um agregado heterogêneo. Em uma máquina agregada, o trabalho de escalonamento local é realizado pelo sistema operacional de cada nó. Desse modo, a estratégia selecionada é do tipo global.

A segunda opção refere-se ao momento em que é realizado o escalonamento: Antes do início da execução (estático) ou durante a execução (dinâmico). Seguindo o modelo das aplicações paralelas programadas com a biblioteca MPI-1, as quais atualmente representam a grande maioria devido a ampla aceitação do padrão e que suportam apenas gerenciamento estático de processos conforme já discutido, o escopo deste trabalho é definir um modelo estático de escalonamento de processos. Para a realização de escalonamento dinâmico seguindo os moldes da proposta deste trabalho (direcionado a aplicações genéricas), seria necessário o emprego de algum *software* capaz de realizar migração de processos.

O modelo proposto no trabalho destina-se a escalonar os processos de uma aplicação genérica; devido a isto, existe a incapacidade de saber-se de antemão todas as informações necessárias para que seja realizada um escalonamento ótimo. Dessa forma, opta-se pelas abordagens sub-ótimas.

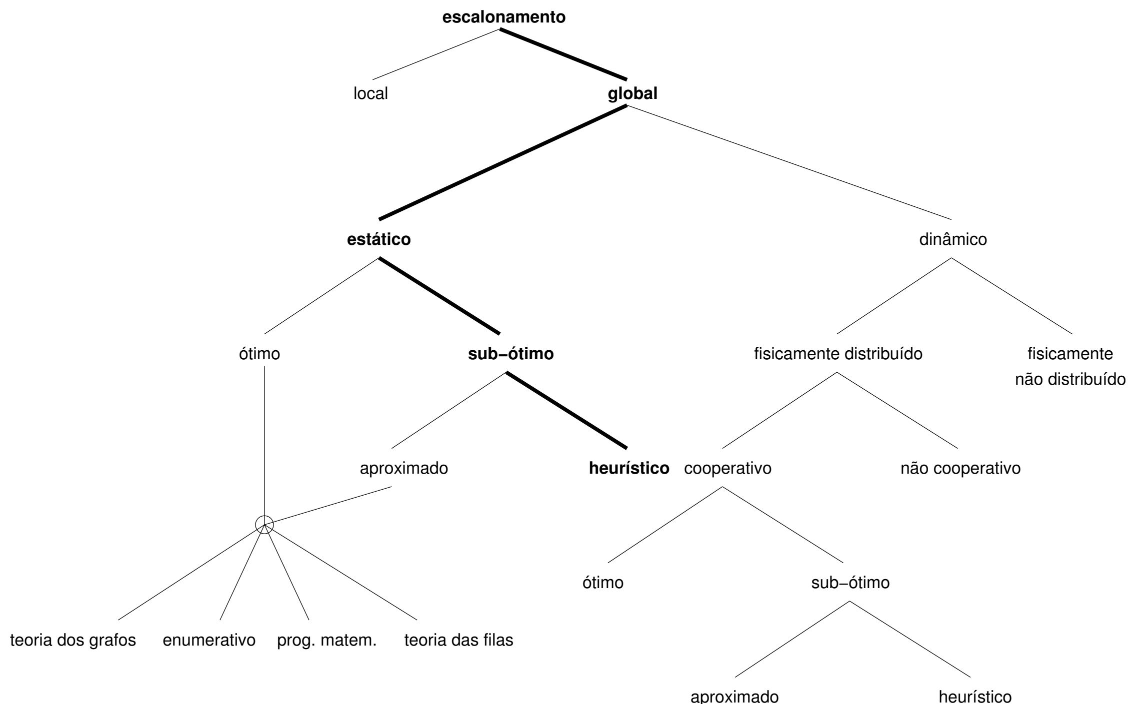


Figura 6 – Estratégia de escalonamento utilizada

As abordagens sub-ótimas dividem-se em duas categorias básicas: aproximadas e heurísticas. Conforme visto, nas estratégias de escalonamento aproximadas, a idéia é utilizar modelos formais do algoritmo da aplicação. Entre os principais modelos utilizados para isto, destacam-se teoria dos grafos, programação matemática e teoria das filas. Torna-se clara a inviabilidade da utilização de tais formalismos, uma vez que o modelo proposto é destinado a aplicações genéricas e não tem-se o conhecimento sobre a natureza da aplicação paralela. Devido a isto, recorre-se à abordagem heurística. Esta, por sua vez, baseia-se na utilização de algoritmos que tentam aprender o comportamento da aplicação, para então escaloná-la da melhor maneira possível.

Levando-se em consideração a classificação não-hierárquica proposta por Casavant, a estratégia de escalonamento adotada também caracteriza-se por ser adaptativa, o que representa a capacidade do escalonador alterar suas políticas, parâmetros ou algoritmos em resposta a um *feedback*. Neste caso, o *feedback* provém da análise dos históricos de monitoração coletados em execuções posteriores da mesma aplicação e servem de base para alteração de parâmetros de escalonamento.

A estratégia adotada representa uma política de escalonamento, isto é, ela tem por objetivo decidir o que será feito, deixando a cargo dos mecanismos do sistema de execução da biblioteca MPI executar de fato as decisões tomadas.

5.3 Objetivos do modelo

Os objetivos do modelo são:

- **otimização do desempenho de uma aplicação paralela MPI genérica executada sobre um agregado heterogêneo**. Através da especificação de uma arquitetura e um algoritmo para calcular o número de processos que serão atribuídos a cada tipo de nó a cada nova execução da aplicação, baseando-se nos históricos de monitoração das execuções prévias da aplicação;
- **implementação de uma ferramenta baseada na arquitetura e no algoritmo propostos**. Tal ferramenta será composta de sistema de monitoração, disparador de aplicações e interface gráfica;
- **avaliação** através da instalação da ferramenta implementada em um agregado heterogêneo no CPAD, e realização de testes envolvendo algumas aplicações paralelas selecionadas para otimização.

5.4 Forma de avaliação

Este modelo de otimização de aplicações paralelas executadas sobre agregados heterogêneos será avaliado através do desenvolvimento e teste de uma ferramenta que implemente a proposta. Tal ferramenta será composta por (1) sistema de monitoração, (2) disparador de aplicações e (3) interface gráfica para visualização de resultados.

Para realizações dos testes, o sistema desenvolvido será instalado em um agregado heterogêneo do CPAD e serão realizados testes com algumas aplicações paralelas selecionadas.

5.5 Arquitetura do modelo

Para dar suporte ao modelo, foi projetada uma infra-estrutura de monitoração, que opera de forma integrada ao gerenciador de recursos, capaz de armazenar as informações coletadas em banco de dados. Estas informações são utilizadas como entrada para o algoritmo que refina, no decorrer das execuções, o número de processos a serem disparados em cada classe de nó. Junta-se a isso uma interface Web que permite a análise e comparação do comportamento de aplicações paralelas, como, por exemplo, avaliar o ganho de desempenho obtido após um certo número de execuções.

A figura 7 ilustra a arquitetura para o protótipo, com seus diferentes módulos e o inter-relacionamento entre eles. A seguir temos uma descrição em alto nível do objetivo e funcionalidades de cada um destes módulos.

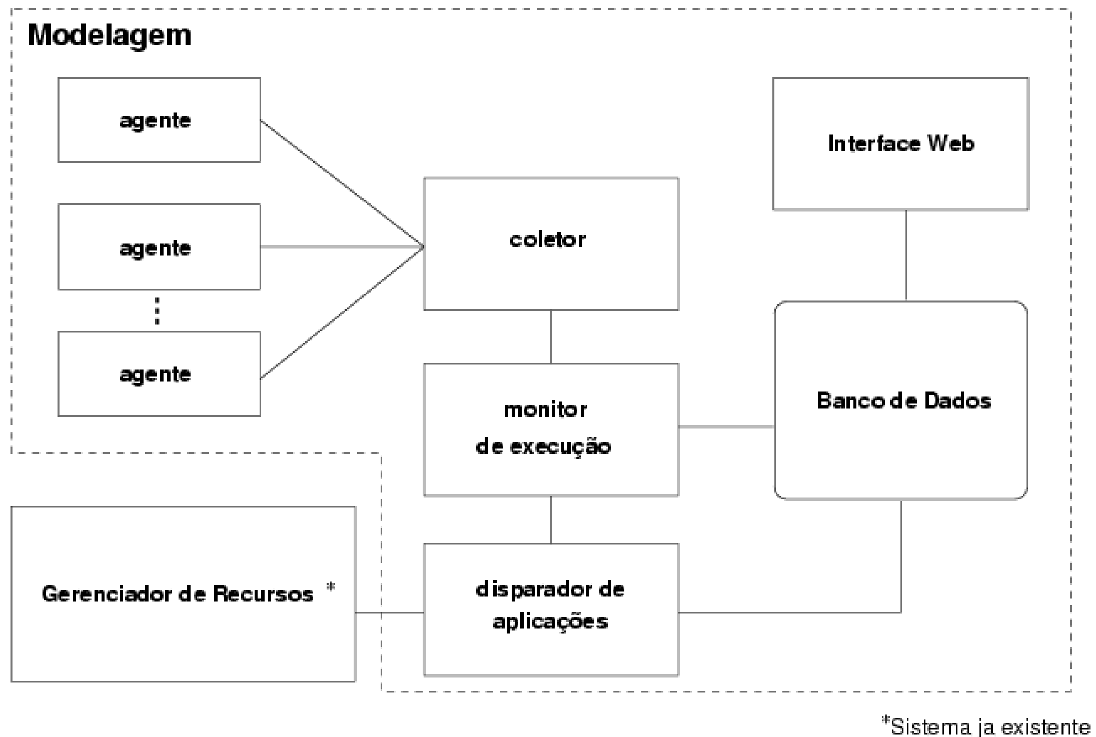


Figura 7 – Arquitetura do modelo

- `agente`: Módulo responsável por coletar as informações de monitoração em cada nó e enviá-las ao módulo coletor;
- `coletor`: Este módulo tem como objetivo recolher as informações enviadas pelos módulos agentes e torná-las disponíveis para módulos clientes, como aplicações para monitoração em tempo real, bem como para o módulo monitor de execuções;
- `monitor de execução`: O monitor de execuções interage tanto com o coletor quanto com o gerenciador de recursos. No momento de uma execução, o programa de disparo de aplicações informa este módulo sobre a nova execução, para que então seja iniciada a gravação dos dados de monitoração no banco de dados;
- `Interface Web`: Interface Web para visualização de informações, a qual permite a análise de sessões de monitoração de aplicações, identificando a utilização de processador e memória para cada nó que participou da execução de uma dada aplicação paralela, bem como diversos outros tipos de informações, como histórico das execuções, aproveitamento dos recursos alocados e análises comparativas;

- `Banco de Dados`: O banco de dados é utilizado para armazenar tanto as informações de monitoração quanto os parâmetros de balanceamento que irão ser gerados ao longo das execuções das aplicações paralelas;

- `Gerenciador de Recursos`: O sistema proposto terá por objetivo inicial trabalhar em conjunto com o gerenciador de recursos CRONO [16]. O CRONO permite a definição da quantidade de processos a serem disparados em cada máquina alocada no momento de uma execução. O sistema proposto fornecerá estes parâmetros de forma transparente ao usuário.

5.6 Algoritmo de cálculo do número de processos por nó

Nesta seção, é apresentado o algoritmo desenvolvido para o modelo que realiza o cálculo do número de processos por nó para aplicações sem restrições quanto ao número de processos. A idéia principal consiste em calcular o número de processos necessários para que os tipos de nós menos carregados alcancem uma carga equivalente à dos nós mais carregados, utilizando o processo como unidade mínima de divisão. A cada nova execução, o algoritmo é sempre aplicado tomando como ponto de partida o registro da execução com o menor tempo, isto é, aquele em que a aplicação paralela obteve o melhor desempenho. O Algoritmo 1 é apresentado a seguir.

O algoritmo não lida com o índice de utilização de processador de cada nó isolado que participou da execução da aplicação paralela, e sim com a média de carga de cada classe de máquinas. Dessa forma, se existem 8 máquinas alocadas, sendo 4 do tipo A e 4 do tipo B, é calculada a média de utilização de CPU das máquinas do tipo A e média para as do tipo B.

Na primeira execução da aplicação, é disparado um processo para cada processador, desse modo, nós biprocessados recebem dois processos, nós com quatro processadores recebem quatro processos, etc. Para determinar quantos processos devem ser disparados para cada tipo de nó, o algoritmo consulta o banco de dados, o qual contém uma descrição da configuração de cada nó que compõem a máquina agregada.

Nas execuções subseqüentes, o algoritmo utiliza três regras diferentes para o cálculo do número de processos; a seleção de qual regra será empregada se dá a partir do número de vezes que o registro da melhor execução já serviu de base para tentativas de otimização.

A primeira regra é utilizada na primeira tentativa de otimização a partir do registro da melhor execução; isto significa que esta regra sempre será aplicada na segunda vez que a aplicação é executada, pois só existe uma execução realizada até então e ela é convencionada como sendo a melhor. Neste caso, é utilizada uma regra de três simples para calcular-se o número de processos, por exemplo, se existem 8 máquinas alocadas, 4 do tipo A e 4 do tipo B, e a média de utilização das máquinas do tipo A é de 50% e a média das máquinas do tipo B é de 100%, então

Algoritmo 1 Algoritmo para aplicações sem restrições de número de processos

```
1: if primeira_execucao = VERDADEIRO then
2:     Atribuir_1_processo_por_processador()
3: else
4:     tentativas ← Obtem_numero_de_tentativas_de_otimizacao()
5:     if tentativas = 0 then
6:         Calcular_a_partir_da_melhor_execucao_numero_de_processos_por_no()
7:     else if tentativas = 1 then
8:         Adicionar_1_processo_por_no_para_nos_com_menor_carga()
9:     else if tentativas > 1 then
10:        Introduzir_valores_aleatorios_a_partir_da_melhor_execucao()
11:    end if
12:    while Configuracao_ja_testada() = VERDADEIRO do
13:        Introduzir_valores_aleatorios()
14:    end while
15: end if
16: Executar_aplicacao()
```

o número de processos nas máquinas do tipo A é dobrado. Para aplicação desta regra, leva-se sempre em consideração a porcentagem de carga do tipo de nó mais carregado, para então calcular-se quantos processos seriam necessários nos outros tipos de nós a fim de alcançarem a mesma porcentagem de carga.

Só existe uma exceção em relação à primeira regra: caso alguma classe de nós no registro usado como base para a otimização possua uma média de carga muito baixa (convencionada como abaixo de 5%), a segunda regra é aplicada. Isto é interessante para evitar a geração de um número muito alto de processos, podendo ter como consequência uma instabilidade na aplicação.

A segunda regra é empregada quando a tentativa de otimização baseada na primeira regra não foi bem sucedida, isto significa que o registro de melhor execução ainda continua sendo o mesmo e seus parâmetros servirão novamente como base para a otimização. A aplicação desta regra consiste em adicionar um processo por nó para a classe de nós menos carregada.

Quando os dois métodos anteriores de tentativa de otimização não obtêm sucesso, recai-se na inserção de valores aleatórios a partir do registro da melhor execução para tentar achar um configuração de processos mais otimizada. Dessa forma, para cada classe, um processo por nó é adicionado ou subtraído de forma aleatória.

Independentemente de qual a regra aplicada, é testado se a configuração de processos por nó já foi utilizada antes para a aplicação. Caso positivo, também utiliza-se o mesmo método de inserção de valores aleatórios até que seja encontrada uma configuração que ainda não foi testada.

5.6.1 Aplicações com restrições quanto ao número de processos

Algumas aplicações paralelas possuem restrições quanto ao número de processos, isto é, necessitam que o número de processos obedeça a uma regra, como, por exemplo, seja um número quadrado ou potências de dois. Isto é uma consequência da forma como a aplicação foi modelada e implementada; em certos casos, por exemplo, torna-se difícil particionar os dados quando existe um número ímpar de processos.

Tais aplicações necessitam ser tratadas de forma especial pelo sistema proposto, pois não é possível especificar um número arbitrário de processos nestes casos. Após analisar-se diferentes formas de tratar o problema decidiu-se exigir que o usuário informe um número fixo de processos para serem disparados. Este número deve ser compatível com as exigências da aplicação, para que a partir de então (nas execuções subseqüentes), o sistema tente buscar um mapeamento equilibrado de processos aos diferentes tipos de nós, mas sempre mantendo o mesmo número total de processos.

Outra alternativa seria informar ao sistema quais as restrições em relação ao número de processos empregados pela aplicação (números quadrados, potências de dois) e deixar a seu cargo calcular números compatíveis com as restrições. O problema encontrado com esta abordagem é que muitas aplicações paralelas necessitam de recompilação quando se deseja alterar o número de processos que se pretende disparar. O sistema até poderia ser responsável por essa recompilação, executando o compilador quando necessário, mas esta é uma tarefa nada prática, especialmente quando se utiliza um agregado composto por máquinas de arquiteturas diferentes, e torna-se necessária a utilização de *cross*-compiladores.

Desse modo, o algoritmo descrito na seção anterior não é utilizado para aplicações que se enquadram nesta categoria. O Algoritmo 2 é descrito a seguir:

Algoritmo 2 Algoritmo para aplicações com restrições de número de processos

1: **if** $primeira_execucao = VERDADEIRO$ **then**
2: $\quad Atribuir_processos_de_forma_ciclica_aos_processadores_alocados()$
3: **else**
4: $\quad Classe_de_nos_com_maior_carga_cede_processos_a_classe_com_menor_carga()$
5: $\quad$ **while** $Configuracao_ja_testada() = VERDADEIRO$ **do**
6: $\quad\quad Realizar_trocas_aleatorias_entre_as_classes_com_menor_e_maior_carga()$
7: $\quad$ **end while**
8: **end if**
9: $Executar_aplicacao()$

- Na primeira execução da aplicação, os processos são atribuídos de forma cíclica aos nós alocados, isto é, cada nó recebe um processo, até que não existam mais processos para serem atribuídos.

- A partir de então, a cada nova execução, o registro com o melhor tempo é utilizado como

base para a otimização, da mesma forma que o algoritmo para aplicações sem restrições quanto ao número de processos.

- O cerne do algoritmo consiste em “trocar” processos entre a a classe de máquinas mais onerada e a classe de máquinas menos onerada, sempre respeitando o número total de processos estipulado pelo usuário. Desse modo, o tipo de nó que teve a maior utilização de processador cede um número de processos, calculado aleatoriamente, para o tipo de nó com a menor utilização de processador.

- Caso a configuração de processos já tenha sido testada, são realizadas trocas aleatórias, até que seja encontrada uma combinação ainda não testada.

5.6.2 Ciclo de adaptação

O algoritmo descrito é aplicado a cada nova execução da aplicação, compondo um ciclo de adaptação, visando otimizar o desempenho ao longo das execuções. Este ciclo, no entanto, tende a degradar o desempenho da aplicação ao longo de muitas execuções. Isto ocorre porque configurações não são repetidas, e uma vez que a melhor configuração tenha sido encontrada, o algoritmo não permitirá que ela seja utilizada novamente, sempre tentando configurações ainda não testadas.

Uma alternativa seria terminar o processo de adaptação em algum momento, estipulando, por exemplo, o fim do ciclo quando não fosse obtido nenhum ganho de desempenho após três tentativas consecutivas. Entretanto, considerou-se mais apropriado permitir ao algoritmo sempre tentar novas combinações, deixando a cargo do usuário decidir quando parar.

6 Ferramenta DAPSched

Neste capítulo são apresentados aspectos relativos a implementação e ao funcionamento de cada módulo que compõem o protótipo, o qual foi batizado de DAPSched - *Distributed systems Adaptive Process Scheduller*.

6.1 Sistema de monitoração

Antes de descrever a implementação do sistema de monitoração, é interessante registrar que o CPAD conta com um sistema de monitoração de propósito geral desenvolvido no próprio centro (RVISION [17]); que apesar de não ter sido utilizado devido a questões pragmáticas citadas a seguir, sua integração com a ferramenta DAPSched pode ser elencada como trabalho futuro.

- armazena informações em formato XML (*eXtensible Markup Language*), não possuindo possibilidade de utilização de banco de dados;
- é relativamente complexo;
- julgou-se mais simples e prático o desenvolvimento de um sistema de monitoração pequeno, especialmente projetado para atender aos requisitos do modelo proposto.

O sistema de monitoração projetado é divido em três programas, todos implementados em linguagem C. A linguagem C foi escolhida por ser a linguagem padrão utilizada em sistemas UNIX para programação em nível de sistema. Destaca-se também a utilização da biblioteca *PThreads* [39]. Esta biblioteca, uma implementação para Linux do padrão ANSI/IEEE POSIX 1003.1 - 1995, fornece uma Interface de Programação de Aplicações - *Application Programming Interface* (API) - para utilização de múltiplas linhas de execução. A seguir, temos uma descrição detalhada de cada módulo do sistema de monitoração.

6.1.1 Módulo agente

A função do módulo agente consiste em obter as informações de utilização de processador e memória (atualmente os índices de uso de memória, apesar de serem monitorados, não estão

sendo utilizados) dos nós do agregado alocados pelo usuário e em enviá-las periodicamente ao módulo coletor.

A aquisição destas informações é realizada através do pseudo sistema de arquivos `/proc` padrão do Linux. Este sistema de arquivos é gerado dinamicamente pelo *kernel* e possui uma grande quantidade de informações sobre o estado do sistema [40]. Desse modo, o módulo agente obtém as informações sobre utilização do processador a partir do arquivo `/proc/stat`, o qual é constantemente atualizado pelo *kernel*.

O programa, ao ser disparado, fica bloqueado à espera de uma conexão em uma porta pré-determinada. Uma vez que uma conexão seja estabelecida pelo módulo coletor, os dados obtidos passam a ser enviados a cada intervalo de tempo pré-estabelecido através do protocolo TCP.

O intervalo de tempo pré-estabelecido de envio de informações é de 1 segundo, julgando-se que este seja um valor compatível com as aplicações avaliadas, as quais não possuem um tempo de execução muito longo (dificilmente ultrapassando 10 minutos). Porém, é importante observar que aplicações paralelas com tempo de execução de horas ou mesmo dias são comuns; desse modo, seria aconselhável utilizar um intervalo de tempo maior entre o envio das informações, isto para evitar um excesso de informações armazenadas no banco de dados, tendo como possível consequência uma degradação de desempenho no acesso a estas informações.

6.1.2 Módulo coletor

O módulo coletor tem como tarefa, por um lado, conectar-se a todos os agentes e receber as informações de monitoração e, por outro, prover estas informações ao módulo monitor de execuções. A idéia da utilização de módulos separados para coleta de informações e monitoramento de execuções decorre da possibilidade de ter-se mais de um programa cliente conectado ao coletor. Isto é, além do monitor de execuções, um programa para realizar o acompanhamento em tempo real dos dados monitorados de forma gráfica (programa este implementado para fins de teste e não discutido nesta dissertação).

Ao ser executado, o programa lê, a partir de um arquivo de configuração, os *hosts* aos quais deve conectar-se. Uma vez estabelecida a conexão com cada um deles, os dados começam a ser recebidos. Paralelamente, uma *thread* aguarda bloqueada a conexão de clientes. Com a conexão de um cliente, este passa a receber os dados de monitoração no intervalo pré-determinado de tempo.

Para que um programa receba dados de múltiplas conexões ao mesmo tempo, é necessário utilizar algum método especial, uma vez que operações normais de leitura de *sockets* são bloqueantes. Uma alternativa seria a utilização de *threads*, porém, aqui, optou-se pela utilização da função *select()* para monitorar as múltiplas conexões com os módulos agentes, visando uma simplificação do código do programa. Esta função, implementada a partir de uma chamada de

sistema, permite monitorar múltiplas conexões ao mesmo tempo, indicando, quando do recebimento de informações, de qual *socket* elas provêem.

6.1.3 Módulo monitor de execuções

Este programa é o maior dos três que compõem o sistema de monitoração, sua tarefa é receber requisições do módulo de disparo de aplicações (módulo que faz a interação com o CRONO, discutido posteriormente) para então iniciar a operação de registro no banco de dados das informações monitoradas na execução de uma aplicação paralela, as quais são recebidas do módulo coletor.

O programa é composto por três *threads*:

- *thread* de comando, que tem como função ficar bloqueada aguardando requisições de início e fim de execuções, as quais são realizadas pelo módulo de disparo `cropt`. Quando uma requisição de início de execução é recebida, uma nova *thread* de registro é lançada, a qual é finalizada quando do recebimento de uma requisição de término da aplicação;
- *thread* de monitoração, cuja função consiste em conectar-se com o módulo coletor e, então, receber regularmente os dados de monitoração;
- *thread* de registro, que tem por função armazenar os dados monitorados (índices de utilização de processador) no banco de dados. Ao final da sessão de registro, são calculadas as médias de utilização de processadores (média geral, média para cada classe de nó e média para cada nó em particular), dados estes que poderão ser utilizados pelo algoritmo de cálculo do número de processos para orientar a melhor distribuição de processos em execuções posteriores.

O programa, uma vez iniciado, conecta-se com o módulo coletor através da *thread* de monitoração, para obter os dados do módulo coletor. Ao mesmo tempo, a *thread* de comando é lançada e aguarda requisições de início de execuções.

6.2 Base de dados

O sistema gerenciador de banco de dados adotado foi o MySQL [41]. Entre os fatores que contaram para sua utilização, destaca-se o fato de ser um programa de código aberto, multiplataforma e de fácil integração com várias linguagens (incluindo C e PHP). A Figura 8 apresenta o modelo entidade-relacionamento do banco de dados.

A seguir, temos uma descrição das tabelas da base de dados:

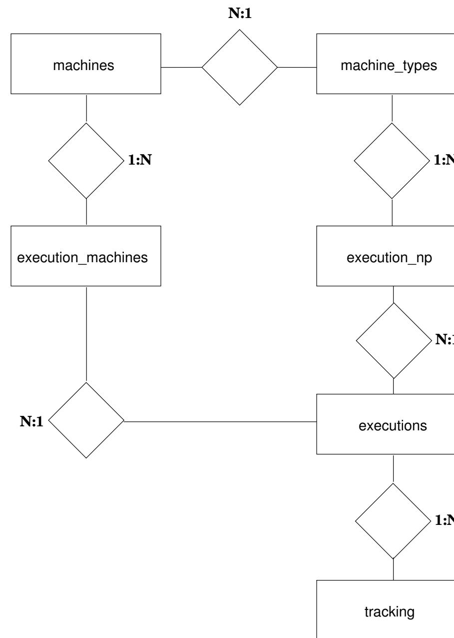


Figura 8 – Diagrama da Base de Dados

- `machines`: armazena informações relativas às máquinas que compõem o agregado, como *hostname*, tipo, etc.;
- `machine_types`: contém um registro para cada tipo de máquina que compõe o agregado, descrevendo sua arquitetura, memória, processadores, etc.;
- `app_nicks`: armazena "apelidos" dados às aplicações, isto possibilita que a mesma aplicação possa ser otimizada para diferentes conjuntos de dados;
- `executions`: contém informações sobre as execuções de aplicações, como tempo de início, tempo de fim, nome da aplicação, linha de comando, etc.;
- `execution_machines`: armazena as máquinas alocadas ao usuário em uma execução;
- `execution_np`: contém o número de processos disparados para cada tipo de máquina em uma execução;
- `tracking`: armazena os dados de monitoração da execuções.

6.3 Interface web

Para o desenvolvimento da interface web, optou-se pela linguagem PHP, devido à facilidade que ela apresenta para o desenvolvimento de sites dinâmicos. PHP tem como uma das características mais importantes o suporte nativo a um grande número de bancos de dados, incluindo

o MySQL adotado neste trabalho, isto torna a tarefa de construir um site dinâmico que faça acesso a um banco de dados uma tarefa relativamente simples.

Para geração de gráficos, utilizou-se a biblioteca JPGraph, a qual possibilita o desenvolvimento de gráficos bastante elaborados com um mínimo de código. Entre as vantagens apresentadas por esta biblioteca, pode-se destacar: (1) o fato de ter suporte a uma grande variedade de modelos de gráficos, (2) seguir uma modelagem orientada a objetos, facilitando a criação de gráficos através do mecanismo de herança e (3) ser um projeto de código aberto e de livre distribuição.

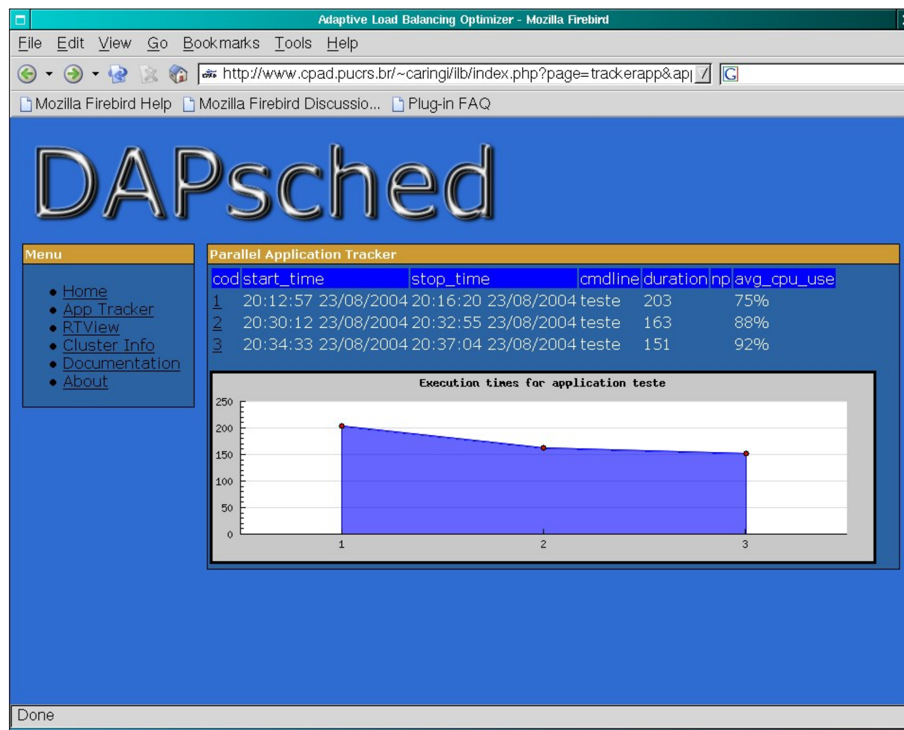


Figura 9 – Tela do tempo das execuções de uma aplicação paralela

Entre as principais funcionalidades da interface web, é possível destacar:

- listagem das execuções de uma aplicação, incluindo os seguintes dados para cada execução: *timestamp* de início e término da execução, linha de comando, duração e média geral de utilização de processador. Esta página também conta com gráfico para visualização dos tempos de execução da aplicação paralela ao longo das execuções, conforme a Figura 9;

- página com análise comparativa das diferentes execuções da mesma aplicação com gráficos de utilização média de processador para cada execução, de acordo com a Figura 11;

- análise da utilização dos processadores ao longo de uma execução paralela, como pode ser visto na Figura 10. Esta página possui uma ficha completa da execução, incluindo quais nós foram utilizados, média de utilização de processador para cada nó e a sumarização da média para cada classe de nós, etc.

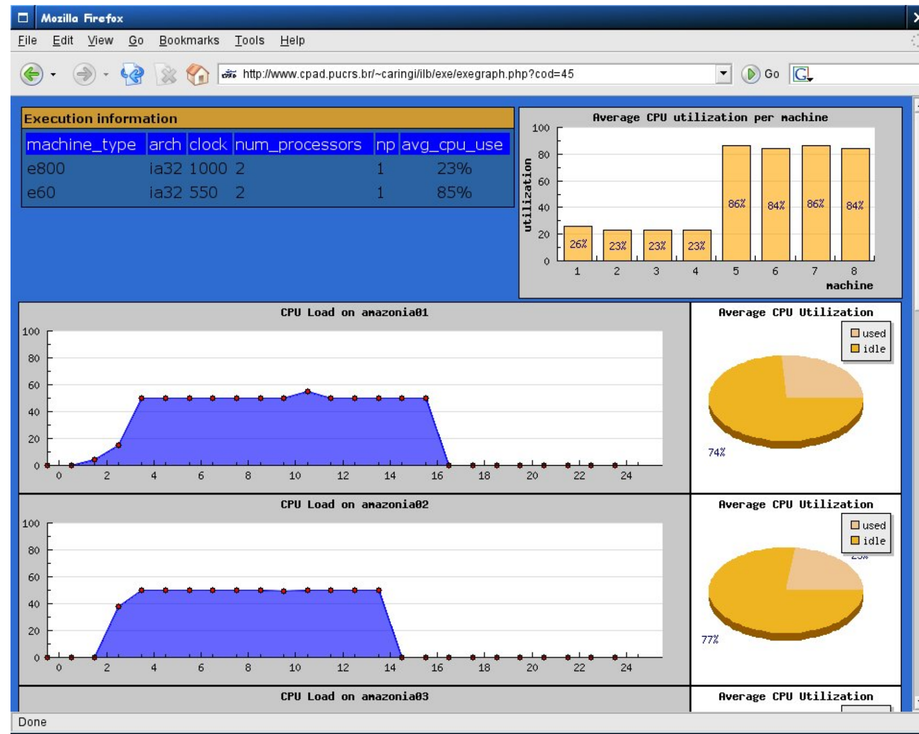


Figura 10 – Tela da análise detalhada de uma execução

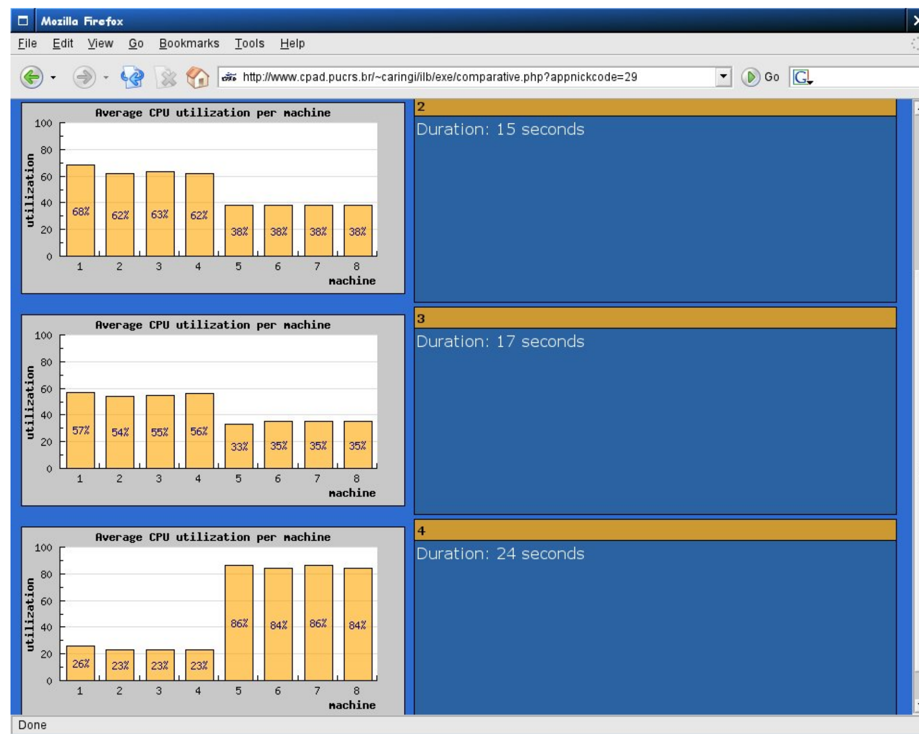


Figura 11 – Tela da comparação de execuções da mesma aplicação

6.4 Disparador de aplicações

O disparador de aplicações `cropt` (CRONO *Optimizer*) faz a integração com o gerenciador de recursos CRONO, sendo o módulo utilizado pelo usuário para executar aplicações paralelas.

Os principais parâmetros que podem ser passados a ele pela linha de comando são os seguintes:

- `-exec <appnickname> <appname>`: principal opção, utilizada para disparar aplicações paralelas. Recebe como parâmetro um apelido para a aplicação e o nome do programa executável;

- `-execfixednp <np> <appnickname> <appname>`: possibilita a otimização de aplicações que possuem restrições quanto ao número de processos, permitindo que seja especificado um número fixo de processos que será utilizado como base para a otimização;

- `-execonlymon <np> <appnickname> <appname>`: esta opção permite que seja realizada a execução de uma aplicação com o sistema de otimização desativado, tendo como objetivo permitir a análise do seu comportamento através da interface gráfica, isto é, apenas para fins de monitoração;

- `-best <appnickname <appname>`: realiza a execução da aplicação utilizando os mesmos parâmetros da melhor execução, isto é, esta opção faz com que o sistema não faça tentativas de otimização;

- `-list`: permite listar as execuções realizadas;

- `-del <appnickname>`: permite apagar um apelido e todas as suas execuções relacionadas.

A possibilidade de utilização de apelidos (*nicknames*) para as aplicações é um mecanismo projetado com o objetivo de dar flexibilidade à ferramenta. Este mecanismo torna possível que uma mesma aplicação seja otimizada em diferentes contextos, como, por exemplo, diferentes conjuntos de dados. Cada apelido representa uma "linha de otimização", de modo que as informações de execuções prévias da aplicação de um determinado apelido não são utilizadas para outro.

A opção `-exec` realiza uma série de tarefas que serão descritas a seguir e então executa a aplicação paralela passando os parâmetros corretos ao *script* `crrun` do CRONO, o qual é um *wrapper* para o *dispatcher* MPI, denominado `mpirun`.

- verifica se a aplicação já foi executada anteriormente, caso negativo utiliza o número padrão de processos para cada classe de nó. É interessante observar que, caso a aplicação já tenha sido executada sobre outro apelido, isto não é levado em consideração;

- caso a aplicação já tenha sido executada antes (dentro do mesmo apelido), o algoritmo de cálculo do número de processos, baseado no registro da melhor execução, gera um novo número de processos por nó, na tentativa de otimizar o desempenho da aplicação;

- é gravado um arquivo de máquinas que será passado ao script `crrun`, indicando o número de processos para cada máquina;

- o sistema de monitoração é informado do início de uma nova execução, e então passa a registrar no banco de dados os índices de utilização de processador nas máquinas que participam da execução da aplicação;
- a aplicação é disparada;
- ao término da execução, são calculadas as estatísticas de utilização das CPU.

6.4.1 Arquivo de Máquinas

A biblioteca MPICH utiliza um arquivo de máquinas para definir a localização e o número de processos que serão disparados em uma execução. Trata-se de em um arquivo texto contendo o nome dos nós que participarão da execução, possuindo, como exemplo, o seguinte formato:

```
amazonia01
amazonia02
amazonia03
amazonia04
amazonia05
amazonia06
amazonia07
amazonia08
```

Quando o usuário realiza uma alocação através do gerenciador de recursos CRONO, este cria automaticamente um arquivo de máquinas listando os nós alocados pelo usuário como o listado acima. Em uma execução através do *script* `crrun`, se o usuário especifica um número maior de processos, através do parâmetro `-np <num>`, do que o número de nós listados no arquivo de máquinas, os processos serão lançados ciclicamente na ordem especificada na lista, até que o número de processos desejado seja alcançado.

O sistema descrito neste trabalho apóia-se na criação de arquivos de máquinas personalizados para definir o número de processos que serão disparados em cada nó. A criação de mais de um arquivo de máquinas é necessária quando se utilizam múltiplas arquiteturas. Para especificar que seja disparado mais de um processo para cada nó, basta que o nome deste se encontre repetido na lista.

Como exemplo, em uma execução envolvendo 4 máquinas (amazonia01, amazonia02, amazonia27 e amazonia28), sendo 2 da arquitetura IA32 (amazonia01 e amazonia02) e 2 da arquitetura IA64 (amazonia27 e amazonia28), onde o algoritmo de cálculo do número de processos definiu que nas máquinas IA32 sejam disparados 2 processos por nó, e nas máquinas IA64 sejam disparados 6 processos por nó, os arquivos de máquina gerados tem o seguinte formato:

Arquivo `.machines.ia32`:

```
amazonia01
amazonia01
amazonia02
amazonia02
```

Arquivo `.machines.ia64`:

```
amazonia27
amazonia27
amazonia27
amazonia27
amazonia27
amazonia27
amazonia28
amazonia28
amazonia28
amazonia28
amazonia28
amazonia28
```

A linha de comando gerada pelo módulo `cropt` é a seguinte:

```
/usr/local/mpich-ether/bin/mpirun
    -arch ia32 -machinefile .machines.ia32 -np 4
    -arch ia64 -machinefile .machines.ia64 -np 12 appname.%a
```

6.5 Utilização da ferramenta

Com o objetivo de facilitar a execução da ferramenta, posto que a mesma é formada por mais de um programa independente, foi desenvolvido um disparador chamado `ilblaunch`. Este disparador se encarrega de executar os diferentes módulos da ferramenta na ordem correta. Para isto, primeiro são verificados quais nós estão alocados para o usuário e, então, em cada um deles, é executada uma instância do módulo agente. Concluída a primeira etapa, são executados na máquina hospedeira do agregado o módulo coletor e o módulo monitor de execuções, estando então a ferramenta pronta para utilização através do módulo `cropt`.

7 Testes e resultados atingidos

O objetivo deste capítulo é avaliar o modelo proposto a partir da análise do comportamento de algumas aplicações paralelas selecionadas quando executadas através da ferramenta implementada. Também faz-se necessário um breve estudo sobre os diferentes tipos de aplicações paralelas existentes a fim de ter-se um maior entendimento do comportamento observado nas aplicações testadas.

7.1 Classificação de aplicações paralelas

Tem sido amplamente reconhecido que as aplicações paralelas podem ser classificadas em alguns paradigmas de programação bem definidos [2]. Esta observação decorre do fato que um número reduzido de paradigmas de programação são utilizados repetidamente no desenvolvimento de muitas aplicações paralelas. Cada paradigma representa uma classe de algoritmos que possuem a mesma estrutura de controle [42].

Os dois principais aspectos que determinam a escolha do paradigma empregado são os recursos computacionais paralelos disponíveis e o tipo de paralelismo inerente ao problema. Os recursos computacionais definem o nível de granularidade que pode ser suportado de forma eficiente no sistema. O tipo de paralelismo, por sua vez, reflete a estrutura da aplicação ou dos dados e ambos os tipos podem coexistir em partes diferentes da mesma aplicação. O paralelismo que emerge da estrutura da aplicação é chamado de paralelismo funcional, enquanto o paralelismo encontrado na estrutura dos dados chama-se paralelismo de dados [2].

Na literatura, há várias propostas diferentes de classificação de aplicações paralelas [42–45]. Em [45], por exemplo, é apresentada uma classificação teórica de aplicações paralelas, a qual é dividida em três classes:

- *processor farms*, que consiste na replicação de *jobs* independentes;
- decomposição geométrica, baseada na paralelização das estrutura de dados;
- paralelismo algorítmico, que resulta do uso de fluxo de dados.

Outra classificação é proposta em [42]. O autor estudou diversas aplicações paralelas e identificou os seguintes paradigmas:

- *pipelining*;

- divisão e conquista;
- mestre/escravo;
- aplicações com autômatos celulares (baseadas no paralelismo de dados).

Em [43], é proposta uma classificação um pouco diferente, baseada na estrutura temporal dos problemas. A divisão das aplicações se dá em quatro classes:

- problemas síncronos;
- problemas fracamente síncronos;
- problemas assíncronos;
- aplicações embaraçosamente paralelas.

Em [2], são analisadas diversas classificações e o autor sumariza os modelos de aplicações paralelas em cinco tipos básicos, os quais são detalhados a seguir:

- mestre/escravo;
- SPMD;
- *pipelining*;
- divisão e conquista;
- paralelismo especulativo.

7.1.1 Aplicações mestre/escravo

As aplicações paralelas baseadas neste paradigma consistem em duas entidades: um mestre e múltiplos escravos. O mestre é responsável por decompor o problema em pequenas tarefas e distribuir estas tarefas entre o conjunto de processos escravos, bem como, ao final da execução, recolher os resultados parciais com o objetivo de produzir o resultado final da computação. Os processos escravos executam um ciclo muito simples: obtêm uma mensagem com a tarefa, processam a tarefa e enviam o resultado ao mestre [2].

7.1.2 Aplicações SPMD

O paradigma SPMD (*Single-Program Multiple-Data*) - Único-Programa Múltiplos-Dados - é o paradigma mais usualmente empregado [2]. Nele, cada processo executa o mesmo código, mas cada um em diferentes partes dos dados. Este tipo de paralelismo também é chamado de paralelismo geométrico, decomposição de domínio, ou paralelismo de dados.

Aplicações SPMD podem ser muito eficientes, se os dados estão bem distribuídos para os processos e o sistema computacional é homogêneo. Se os processos apresentam diferentes cargas de trabalho então, é aconselhável a utilização de algum esquema de balanceamento de carga [2].

Conforme [46], o padrão MPI é propício para o paradigma de programação SPMD, que, de acordo com este artigo, significa todos os nós executarem o mesmo código executável. Dessa forma, percebe-se que existe uma certa confusão na literatura quanto à utilização do termo SPMD: certas vezes, o seu significado corresponde a "aplicações onde cada processo trabalha em uma parte dos dados"; em outros casos, "aplicações onde cada processo executa o mesmo programa executável". Esta segunda definição permite enquadrar praticamente qualquer aplicação MPI como SPMD, o que não condiz muito com a realidade, pois, através de estruturas condicionais, cada processo pode executar uma tarefa diferente.

7.1.3 Aplicações *pipelining*

As aplicações desenvolvidas com este paradigma baseiam-se na decomposição funcional da aplicação, isto é, as tarefas do algoritmo capazes de operação concorrente são identificadas, e cada processador executa uma pequena parte do algoritmo total.

Os processos são organizados em um *pipeline*, cada processo correspondendo a um estágio do *pipeline* e sendo responsável por uma tarefa particular [2].

7.1.4 Aplicações de divisão e conquista

Esta técnica consiste em dividir o problema em dois ou mais sub problemas, cada sendo resolvido de forma independente e tendo seus resultados combinados para obter-se o resultado final. Processamento adicional pode ser necessário para dividir o problema original ou para combinar os resultados dos sub problemas.

Neste paradigma, a aplicação é organizada na forma de uma árvore virtual: alguns processos dividem e delegam tarefas a outros processos, sendo que o processamento é realizado pelas folhas da árvore virtual.

7.1.5 Aplicações de paralelismo especulativo

Este paradigma é empregado quando é bastante difícil obter-se paralelismo através de qualquer um dos paradigmas citados [2], sendo direcionado a alguns problemas que possuem dependências de dados complexas, o que reduz as possibilidades de exploração do paralelismo. Nestes casos, uma solução apropriada é executar pequenas partes do problema, usando um pouco de especulação para facilitar o paralelismo.

7.2 Metodologia empregada nos testes

A ferramenta implementada foi instalada e configurada no laboratório do Centro de Pesquisa em Alto Desempenho da PUCRS (CPAD-PUCRS/HP).

Os testes foram realizados na principal máquina agregada do CPAD, batizada de Amazônia. Este agregado heterogêneo é composto por 4 tipos diferentes de nós:

- 16 nós HP-E60, cada um com 2 processadores Pentium III de 550MHz e 256 MBytes de memória principal (referidos como E-60);
- 8 nós HP-E800, cada um com 2 processadores Pentium III de 1GHz e 256 MBytes de memória principal (referidos como E-800);
- 2 nós IPF, cada um com 1 processador Itanium-2 de 900MHz e 512 MBytes de memória principal (referidos como IA64MONO);
- 5 nós IPF, cada um com 2 processadores Itanium-2 de 1,5GHz e 2 GBytes de memória principal (referidos como IA64DUAL).

Deste modo, totalizando 31 nós, 60 processadores e 17 GBytes de memória RAM. Quanto à tecnologia de interconexão, é utilizada uma rede Myrinet 2000 como rede primária, e uma rede FastEthernet como rede secundária.

A fim de se testar a ferramenta em condições próximas das reais, cada teste foi realizado com diferentes conjuntos de nós. Isto porque dificilmente um usuário tem permissão para alocar todos os nós do agregado. Além disso, em todos os testes, utilizou-se apenas a rede secundária, devido a um estranho comportamento identificado quando da utilização da rede Myrinet, o qual ocasionava a utilização de 100% de CPU 100% do tempo, mesmo em aplicações que sabidamente não tinham este comportamento.

7.3 Aplicações selecionadas

As aplicações paralelas selecionadas compreendem uma pequena aplicação para cálculo do número PI, uma aplicação desenvolvida no CPAD para cálculo das N-Gramas, três *benchmarks* do conhecido pacote *NAS Parallel Benchmarks*, e a aplicação de *ray-tracing* POV-Ray. Tais aplicações foram selecionadas por representarem diferentes paradigmas e por apresentarem diferentes padrões de computação / comunicação, tendo-se por objetivo analisar o comportamento da ferramenta em diferentes cenários.

7.3.1 Aplicação para cálculo do número PI

Esta pequena aplicação para cálculo do número PI (código fonte pode ser encontrado no Apêndice A), representa um modelo bastante simples de aplicação paralela, seguindo o paradigma SPMD, onde cada processo trabalha com uma fatia de igual tamanho de processamento, não havendo nenhuma comunicação entre processos durante a fase de computação. Nenhum mecanismo de balanceamento de carga é empregado, e todos os processos perfazem o mesmo número de iterações, o que faz com que todos os processos terminem o processamento simultaneamente quando a aplicação é executada em um agregado homogêneo.

Para a realização dos testes foram alocados 8 nós de 4 tipos diferentes: 2 E-800, 2 E-60, 2 IA64MONO e 2 IA64DUAL; a Tabela 3 apresenta informações sobre 10 execuções consecutivas da aplicação. A coluna **Tempo** refere-se à duração de cada execução, e as colunas referentes a cada tipo de nó empregados na execução apresentam duas informações: **NPROCS** e **CPU**, que representam, respectivamente, número de processos disparados em cada nó do tipo referido e média de utilização de processador.

	E-800		*E-60*		*IA64MONO*		*IA64DUAL*		Tempo
	NPROCS	CPU	NPROCS	CPU	NPROCS	CPU	NPROCS	CPU	
1	2	49%	2	90%	1	22%	2	13%	64s
2	3	62%	2	69%	4	69%	13	69%	25s
3	4	65%	1	29%	5	71%	14	66%	28s
4	4	71%	2	63%	4	62%	13	63%	27s
5	2	43%	1	39%	3	58%	12	70%	28s
6	2	28%	3	78%	5	63%	14	56%	34s
7	2	27%	3	76%	3	38%	14	55%	36s
8	2	29%	3	78%	5	64%	12	52%	35s
9	2	39%	1	36%	3	53%	14	75%	28s
10	3	53%	2	65%	2	30%	15	75%	28s

Tabela 3 – Número de processos por tipo de nó da aplicação de cálculo do PI

É possível constatar que a primeira execução apresenta o pior resultado (maior tempo) e,

logo na segunda execução, já é obtida a melhor marca, como também pode ser visto no gráfico da Figura 12 que apresenta os tempos das execuções. O ganho de desempenho da melhor execução em relação à primeira é de 156%.

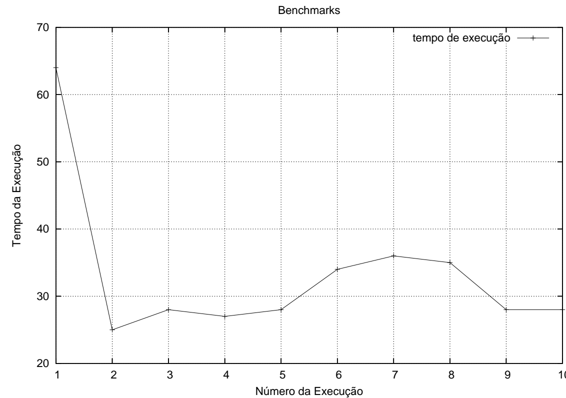


Figura 12 – Tempo das execuções da aplicação de cálculo do PI

A Figura 13 mostra a diferença de utilização de processador dos nós alocados entre a primeira (à esquerda) e a melhor execução (à direita), sendo possível visualizar de forma clara como a minimização do tempo de execução foi alcançada através de uma distribuição equilibrada de processos ao nós alocados pelo usuário. Desse modo, constata-se que este tipo de aplicação paralela caracterizada por ser SPMD, sem comunicação na fase de computação e sem balanceamento de carga, é bastante propícia ao modelo de otimização proposto neste trabalho.

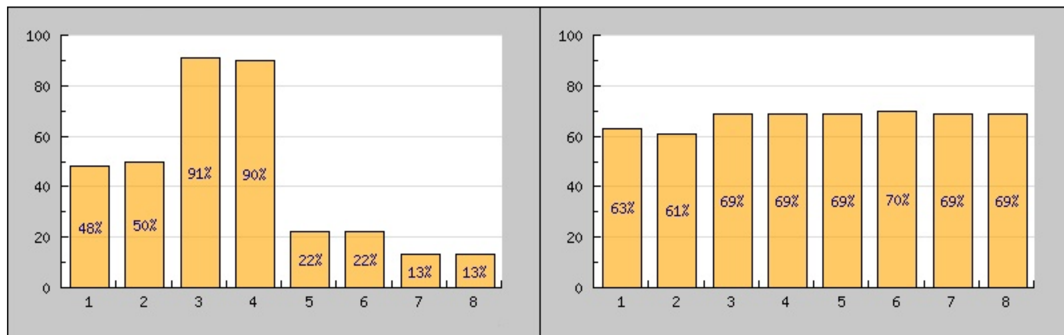


Figura 13 – Comparação entre pior e melhor execução da aplicação de cálculo do PI

7.3.2 Aplicação N-Gramas

Esta aplicação desenvolvida no CPAD realiza a paralelização do modelo N-Gramas; o objetivo deste modelo é contabilizar o número de vezes que cada seqüência de n palavras aparece em um texto significativamente extenso. O valor de n varia de 2 a 4. Para isto, é preciso armazenar cada seqüência de tamanho n e acumular o número de ocorrências. A aplicação segue o modelo mestre/escravo, possuindo dois tipos de processos: (1) Divisor, que divide o texto entre os

demais processos do tipo Contador; e (2) Contador, que calcula as ocorrências das seqüências de palavras no texto recebido. Assim como a aplicação para cálculo do PI, esta também não realiza balanceamento de carga dinâmico, porém tem como diferencial a imprevisibilidade do tempo de execução de cada processo.

Para a realização dos testes, foram alocados 8 nós de 4 tipos diferentes: 2 E-800, 2 E-60 e 2 IA64MONO e 2 IA64DUAL; a Tabela 4 apresenta informações sobre 10 execuções consecutivas da aplicação. Assim como no teste da aplicação anterior, a coluna **Tempo** refere-se à duração de cada execução, e as colunas referentes a cada tipo de nó empregados na execução apresentam duas informações: **NPROCS** e **CPU** que representam, respectivamente, número de processos disparados em cada nó do tipo referido e média de utilização de processador.

	E-800		*E-60*		*IA64MONO*		*IA64DUAL*		Tempo
	NPROCS	CPU	NPROCS	CPU	NPROCS	CPU	NPROCS	CPU	
1	2	58%	2	6%	1	0%	2	6%	141s
2	2	43%	2	45%	2	1%	2	1%	146s
3	3	44%	1	1%	3	32%	3	9%	69s
4	3	36%	2	25%	3	77%	3	2%	70s
5	4	37%	1	29%	4	33%	4	26%	34s
6	4	17%	1	1%	4	73%	5	3%	99s
7	5	39%	1	35%	5	36%	6	27%	36s
8	5	59%	1	1%	3	29%	3	3%	78s
9	5	60%	1	1%	3	28%	3	3%	77s
10	5	37%	2	14%	3	45%	3	10%	59s

Tabela 4 – Número de processos por tipo de nó da aplicação N-Gramas

Observa-se que na segunda execução, onde se deu a adição de um processo à classe de nós menos carregada, foi obtido um resultado inferior ao da primeira execução, porém nas execuções seguintes o sistema consegue otimizar o desempenho da aplicação, culminando com a 5ª execução, onde é alcançado o melhor tempo, como também pode ser visto no gráfico da Figura 14, que apresenta os tempos das execuções. O ganho de desempenho da melhor execução, em relação à primeira, é de 314%.

A Figura 15 mostra a diferença de utilização de processador dos nós alocados entre a primeira (à esquerda) e a melhor execução (à direita). Neste caso, já não é tão clara, como no caso da aplicação anterior, a forma como a minimização do tempo é alcançada, sendo apenas possível constatar que existe um menor desequilíbrio da utilização dos processadores.

Uma possível explicação para um ganho de desempenho tão grande é que, enquanto na primeira execução existem menos processos (a aplicação é menos particionada) e um processo que venha a realizar um maior número de iterações pode ser atribuído a um nó do tipo menos poderoso, na melhor execução o número de processos é maior (maior particionamento) e os processos com maior tempo de execução podem ser atribuídos aos nós mais poderosos.

Constata-se que este tipo de aplicação paralela mestre/escravo, especialmente caracterizada por não realizar balanceamento de carga e por cada processo realizar um número diferente de

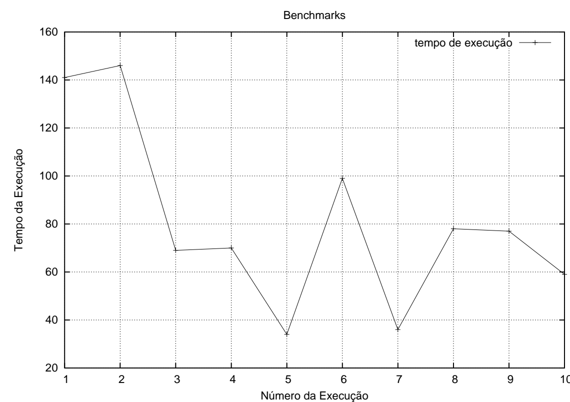


Figura 14 – Tempo das execuções da aplicação N-Gramas

iterações, é capaz de obter um ganho de desempenho relativamente grande com a ajuda do modelo de escalonamento proposto.

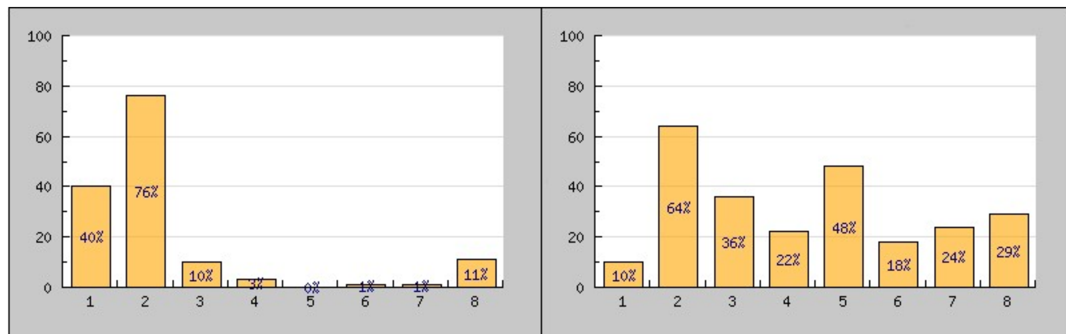


Figura 15 – Comparação entre pior e melhor execução da aplicação N-Gramas

7.3.3 *NAS Parallel Benchmarks*

O NPB (*NAS Parallel Benchmarks*) [47] é um pequeno conjunto de programas projetados para ajudar a avaliar o desempenho de máquinas paralelas. Os *benchmarks* consistem em oito problemas derivados de importantes classes de aplicações da área de aerofísica. Os problemas são divididos em cinco *kernels* e três aplicações de simulação de dinâmica de fluídos - *Computational Fluid Dynamics* (CFD) - implementados em MPI nas linguagens C e Fortran77/90. Os cinco *kernels* imitam o centro de cinco métodos numéricos usados por aplicações CFD.

Foram selecionados três *benchmarks* - *Scalar Pentadiagonal* (SP), *Embarassingly Parallel* (EP) e *Integer Sort* (IS) - por ter sido verificado através de monitoração que apresentam diferentes padrões de computação/comunicação.

Devido a um *bug* existente na versão da biblioteca MPICH empregada, envolvendo a utilização de algumas funções de comunicação coletiva em execuções com nós de arquiteturas diferentes, cada teste foi realizado apenas com máquinas da mesma arquitetura.

Scalar Pentadiagonal (SP)

Este *benchmark* simula uma aplicação que resolve múltiplos sistemas independentes de equações. A aplicação possui uma restrição quanto ao número de processos, exigindo que seja um número quadrado. Dessa forma, foi compilada uma versão da aplicação para ser executada com 25 processos.

Para realização dos testes, foram alocados 8 nós de 2 tipos diferentes: 4 E-800 e 4 E-60; a Tabela 5 apresenta informações sobre 10 execuções consecutivas da aplicação. A coluna **Tempo** refere-se à duração de cada execução, e as colunas referentes a cada tipo de nó empregados na execução apresentam duas informações: **NPROCS** e **CPU**, que representam, respectivamente, número de processos **total** disparados em cada tipo de nó e média de utilização de processador.

	E-800		*E-60*		Tempo
	NPROCS	CPU	NPROCS	CPU	
1	13	68%	12	73%	381s
2	14	64%	11	71%	225s
3	17	74%	8	47%	246s
4	18	76%	7	36%	254s
5	12	42%	13	75%	249s
6	16	74%	9	56%	232s
7	15	71%	10	63%	228s
8	11	38%	14	80%	248s
9	19	79%	6	31%	259s
10	21	80%	4	19%	290s

Tabela 5 – Número de processos por tipo de nó do *benchmark* SP

É possível constatar que a primeira execução apresenta o pior resultado (maior tempo), e a melhor marca é obtida logo na segunda execução, como pode ser visto no gráfico da Figura 16 que apresenta os tempos das execuções. O ganho de desempenho da melhor execução, em relação à primeira, é de 69%.

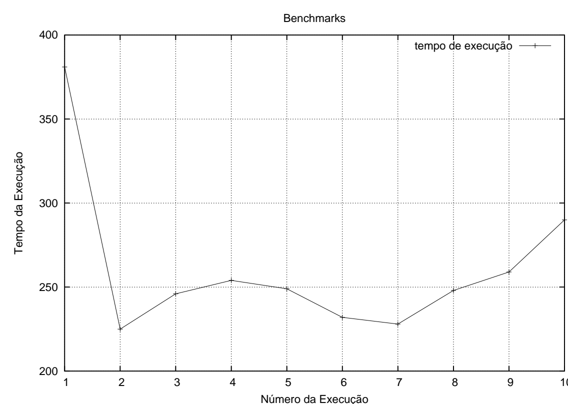


Figura 16 – Tempo das execuções do *benchmark* SP

A Figura 17 mostra a diferença de utilização de processador dos nós alocados entre a primeira (à esquerda) e a melhor execução (à direita), sendo possível verificar que a utilização de processadores na melhor execução é ligeiramente mais equilibrada. Esta aplicação mostrou-se propícia ao modelo de otimização proposto neste trabalho.

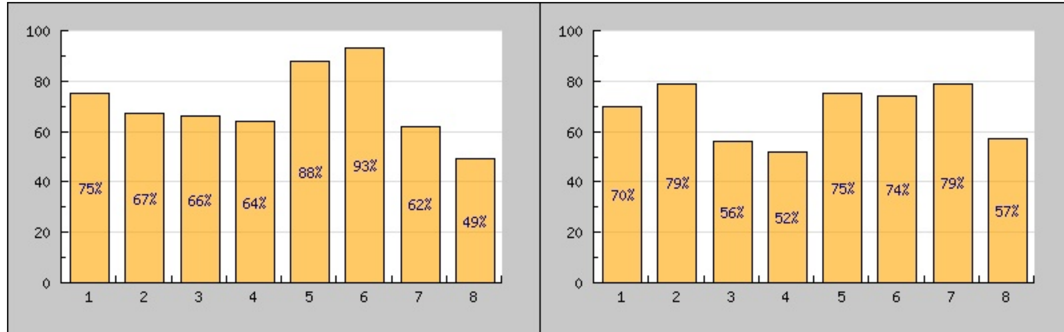


Figura 17 – Comparação entre pior e melhor execução do *benchmark* SP

Embarassingly Parallel (EP)

Este *benchmark* tem por objetivo prover uma estimativa do pico de desempenho máximo da máquina paralela em relação à matemática de ponto flutuante, isto é, o desempenho sem comunicação significativa entre nós. O *benchmark* segue o paradigma SPMD, sem comunicação durante a fase de computação, e possui uma restrição quanto ao número de processos, exigindo que sejam potências de dois. Desse modo, foi compilada uma versão da aplicação para ser executada com 32 processos.

Para realização dos testes, foram alocados 8 nós de 2 tipos diferentes: 4 E-800 e 4 E-60; a Tabela 6 apresenta informações sobre 10 execuções consecutivas da aplicação. A coluna **Tempo** refere-se à duração de cada execução, e as colunas referentes a cada tipo de nó empregados na execução apresentam duas informações: **NPROCS** e **CPU**, que representam, respectivamente, número de processos **total** disparados em cada tipo de nó e média de utilização de processador.

Constata-se que a segunda execução obtém um resultado pior do que a primeira, fruto de uma configuração que gera um desequilíbrio maior do que o existente na primeira execução, porém a partir da terceira execução obteve-se um ganho de desempenho, sendo alcançado o melhor tempo na 5ª execução, como também pode ser visto no gráfico da Figura 18, que apresenta os tempos das execuções. O ganho de desempenho da melhor execução em relação à primeira é de 33%.

A Figura 19 mostra a diferença de utilização de processador dos nós alocados entre a primeira (à esquerda) e a melhor execução (à direita), sendo possível visualizar, da mesma forma que no caso da aplicação para cálculo do PI, como a minimização do tempo de execução foi alcançada através de uma distribuição equilibrada de processos ao nós alocados pelo usuário. Verifica-se que este *benchmark* segue o mesmo paradigma da aplicação para cálculo do PI,

	E-800		*E-60*		Tempo
	NPROCS	CPU	NPROCS	CPU	
1	16	55%	16	97%	384s
2	18	50%	14	70%	468s
3	21	85%	11	79%	323s
4	19	65%	13	79%	381s
5	20	91%	12	97%	288s
6	22	85%	10	67%	342s
7	24	97%	8	57%	324s
8	25	87%	7	43%	377s
9	23	94%	9	64%	323s
10	15	41%	17	83%	476s

Tabela 6 – Número de processos por tipo de nó do *benchmark* EP

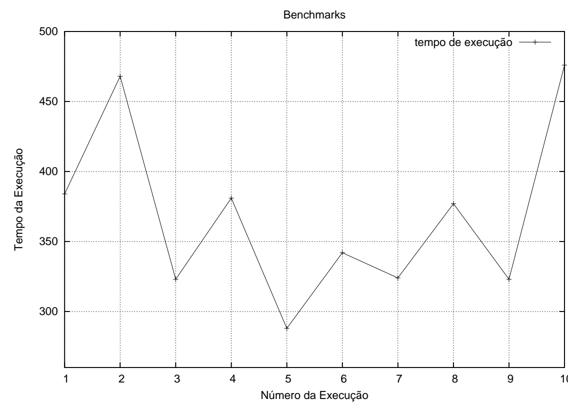


Figura 18 – Tempo das execuções do *benchmark* EP

porém com restrições quanto ao número de processos, mas também é capaz de obter um ganho desempenho considerável com a ajuda do modelo de otimização proposto neste trabalho.

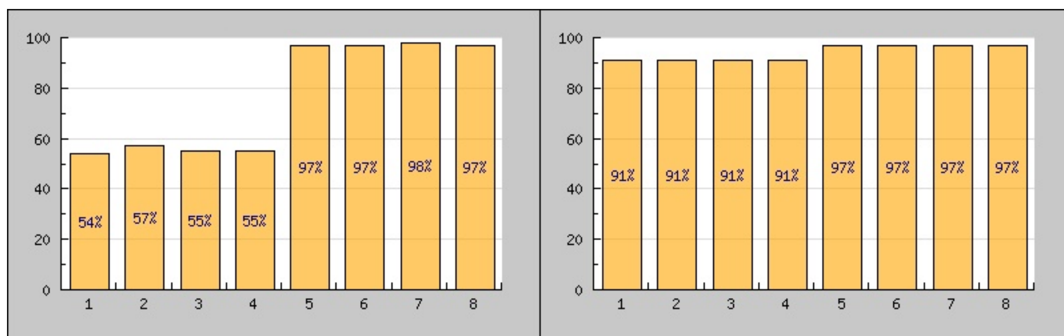


Figura 19 – Comparação entre pior e melhor execução do *benchmark* EP

Integer Sort (IS)

Este *benchmark* avalia o desempenho na tarefa de ordenação de números inteiros. Seu objetivo é testar tanto a capacidade de computação da matemática de números inteiros, quanto o

desempenho de comunicação. A aplicação possui uma restrição quanto ao número de processos, exigindo que sejam potências de dois. Desse modo, foi compilada uma versão da aplicação para ser executada com 32 processos.

Para realização dos testes, foram alocados 8 nós de 2 tipos diferentes: 4 E-800 e 4 E-60; a Tabela 7 apresenta informações sobre 10 execuções consecutivas da aplicação. A coluna **Tempo** refere-se à duração de cada execução, e as colunas referentes a cada tipo de nó empregados na execução apresentam duas informações: **NPROCS** e **CPU**, que representam, respectivamente, número de processos **total** disparados em cada tipo de nó e média de utilização de processador.

	E-800		*E-60*		Tempo
	NPROCS	CPU	NPROCS	CPU	
1	16	24%	16	66%	129s
2	18	28%	14	71%	125s
3	19	31%	13	73%	133s
4	15	23%	17	70%	132s
5	22	41%	10	66%	125s
6	24	46%	8	56%	123s
7	25	47%	7	48%	126s
8	23	44%	9	60%	120s
9	21	36%	11	66%	123s
10	28	53%	4	30%	125s

Tabela 7 – Número de processos por tipo de nó do *benchmark* IS

Das aplicações paralelas testadas, esta foi a que obteve o menor ganho de desempenho. O melhor tempo foi obtido na oitava execução, como também pode ser visto no gráfico da Figura 20 que apresenta os tempos das execuções. O ganho de desempenho da melhor execução em relação à primeira é de apenas 7%.

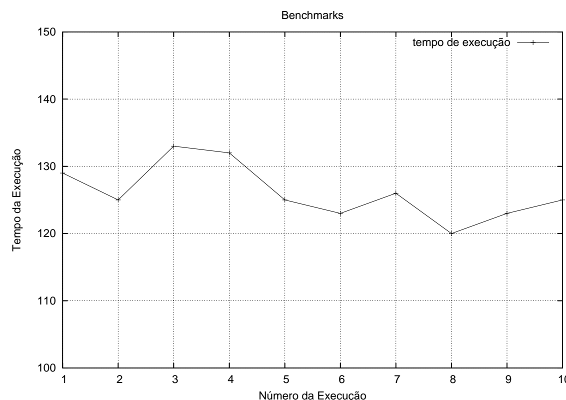


Figura 20 – Tempo das execuções do *benchmark* IS

A Figura 21 mostra a diferença de utilização de processador dos nós alocados entre a primeira (à esquerda) e a melhor execução (à direita), sendo possível visualizar que foi obtido uma

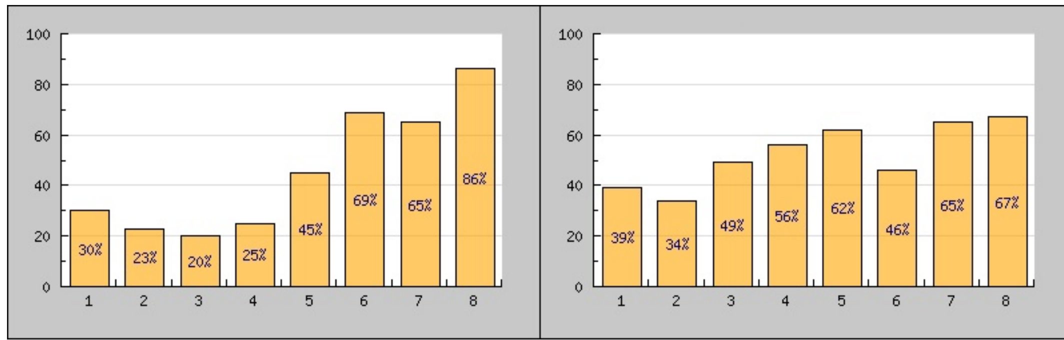


Figura 21 – Comparação entre pior e melhor execução do *benchmark* IS

atenuação no desequílibrio através de uma distribuição otimizada de processos ao nós alocados pelo usuário.

7.3.4 POV-Ray

A aplicação POV-Ray [48] é utilizada para criar imagens tridimensionais usando uma técnica de renderização chamada *ray-tracing*. A aplicação lê um arquivo contendo informação descrevendo os objetos e iluminação de uma cena e gera uma imagem da cena partir do ponto de visualização da câmera, também descrita no arquivo. *Ray-tracing* não é um processo rápido, mas produz imagens de alta qualidade com reflexões realísticas, sombras, perspectivas e outros efeitos.

A aplicação POV-Ray segue o paradigma mestre/escravo e realiza um particionamento estático da imagem a ser renderizada no início da aplicação. O mestre tem como responsabilidade dividir a imagem em pequenos blocos, que são então atribuídos aos escravos. Cada escravo, quando termina de renderizar o seu bloco, o envia de volta ao mestre, o qual vai combinando as partes recebidas a fim de gerar a imagem final.

Para realização dos testes foram alocados 8 nós de 4 tipos diferentes: 2 E-800, 2 E-60 e 2 IA64MONO e 2 IA64DUAL; a Tabela 8 apresenta informações sobre 10 execuções consecutivas da aplicação.

É possível constatar que a primeira execução apresenta o pior resultado (maior tempo). A segunda execução obtém uma diminuição significativa no tempo de execução, e a melhor marca é obtida na terceira execução, conforme pode ser visualizado no gráfico da Figura 22, que apresenta os tempos das execuções. A maior diferença entre tempos ocorre entre a primeira e a segunda execuções, e o ganho de desempenho da segunda execução em relação à primeira é de 67%, e da terceira (melhor) em relação a primeira é de 78%.

A Figura 23 expõe a diferença de utilização de processador dos nós alocados entre a primeira (à esquerda) e a melhor execução (à direita), sendo novamente possível visualizar como a minimização do tempo de execução foi alcançada através de uma distribuição equilibrada de

	E-800		*E-60*		*IA64MONO*		*IA64DUAL*		Tempo
	NPROCS	CPU	NPROCS	CPU	NPROCS	CPU	NPROCS	CPU	
1	2	32%	2	89%	1	43%	2	30%	476s
2	5	82%	2	68%	2	78%	5	59%	286s
3	5	79%	2	65%	2	75%	6	73%	268s
4	6	81%	1	27%	3	92%	7	67%	293s
5	5	72%	3	93%	2	72%	6	65%	277s
6	4	51%	1	27%	3	92%	5	47%	383s
7	6	84%	3	88%	1	42%	7	68%	279s
8	4	52%	3	87%	1	33%	5	48%	375s
9	4	80%	1	43%	1	48%	5	77%	285s
10	6	83%	1	28%	3	97%	5	52%	314s

Tabela 8 – Número de processos por tipo de nó da aplicação POV-Ray

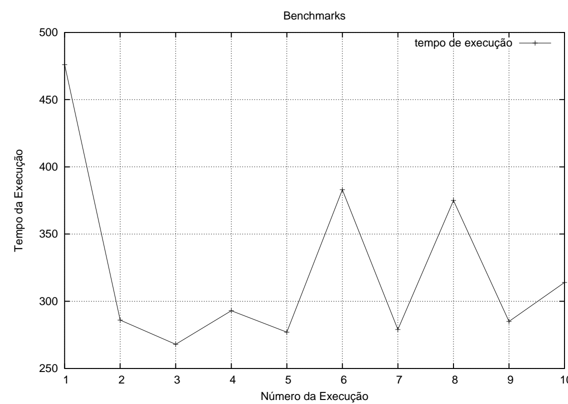


Figura 22 – Tempo das execuções da aplicação POV-Ray

processos ao nós alocados pelo usuário.

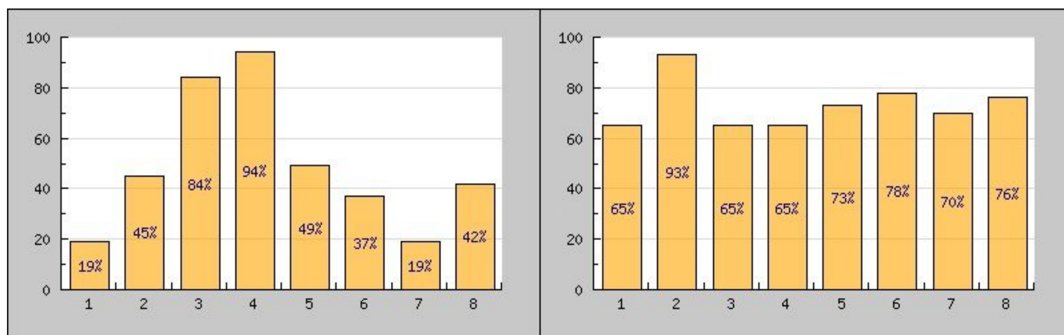


Figura 23 – Comparação entre pior e melhor execução da aplicação POV-Ray

7.4 Variabilidade dos resultados

Esta seção tem por objetivo avaliar a variabilidade dos resultados devido a fatores como memória cache, cache de disco, etc. Na tentativa de identificar se os ganhos de desempenho alcançados são efetivamente fruto do escalonamento otimizado dos processos da aplicação. Dessa forma, conforme a Tabela 9, para cada aplicação paralela utilizada nos testes, foram realizadas dez execuções consecutivas com os mesmos parâmetros.

Número	*PI*	*N-Gramas*	*NAS SP*	*NAS EP*	*NAS IS*	*POV-Ray*
1	1m18.885s	2m59.451s	4m36.532s	9m57.204s	3m14.358s	1m27.566s
2	1m16.814s	2m55.475s	4m39.374s	10m4.355s	3m12.164s	1m24.378s
3	1m16.258s	2m57.667s	4m37.942s	9m54.457s	3m11.495s	1m27.219s
4	1m21.932s	2m58.668s	4m35.335s	9m50.916s	3m15.712s	1m27.073s
5	1m14.959s	2m58.912s	4m41.094s	10m12.113s	3m10.320s	1m25.978s
6	1m16.366s	2m56.154s	4m38.277s	9m49.620s	3m11.509s	1m27.912s
7	1m16.051s	2m58.738s	4m36.951s	10m6.699s	3m11.331s	1m27.224s
8	1m15.863s	2m56.031s	4m41.224s	9m48.172s	3m12.894s	1m26.490s
9	1m19.004s	2m53.319s	4m40.638s	9m48.359s	3m11.744s	1m26.576s
10	1m18.790s	2m56.012s	4m37.119s	10m1.429s	3m13.421s	1m26.713s

Tabela 9 – Variação de tempo das aplicações em execuções com mesmos parâmetros

Os resultados obtidos demonstram que existe uma pequena variabilidade do tempo de execução das aplicações quando as mesmas são executadas sucessivas vezes com parâmetros idênticos. A causa disso, além dos fatores citados acima, também pode ser atribuída a programas que compõem o *middleware* do agregado e estão em permanente execução nos nós, consumindo recursos de forma não muito previsível.

7.5 Considerações sobre os testes

A Tabela 10 contém um resumo das aplicações testadas, incluindo suas características e ganho de desempenho obtido na melhor execução em relação à primeira.

Aplicação	*Paradigma* [1]	*Restrições num. processos*	*Ganho obtido*
PI	SPMD	não	156%
N-Gramas	mestre/escravo	não	314%
NAS SP *benchmark*	SPMD	sim	69%
NAS EP *benchmark*	SPMD	sim	33%
NAS IS *benchmark*	SPMD	sim	7%
POV-Ray	mestre/escravo	não	78%

Tabela 10 – Comparativo das aplicações paralelas testadas

O ganho de desempenho médio geral é de 109,5%, o ganho de desempenho médio entre as aplicações sem restrições quanto ao número de processos é de 182,6% e, entre as aplicações com restrições desse tipo é de 36,3%.

Deve-se considerar que provavelmente o principal fator pelo qual as aplicações sem restrições em relação ao número de processos obtiveram um maior ganho de desempenho é que, nestes casos, utilizou-se 4 tipos diferentes de nós, gerando, por conseqüência, um maior desequilíbrio de carga em relação aos testes do NAS, que envolveram apenas 2 tipos de nós.

Através da análise da variabilidade dos resultados é possível constatar que ela é relativamente baixa se comparada à variação dos resultados decorrente da distribuição otimizada dos processos da aplicação. Com exceção do *benchmark* NAS EP, onde o ganho obtido pode ser considerado dentro da margem de variabilidade, as outras aplicações apresentam ganhos de desempenho significativos.

[1] Segundo a classificação proposta por Buyya em [2]

8 Conclusão e trabalhos futuros

Este trabalho apresentou um modelo para otimização do desempenho de aplicações paralelas MPI genéricas executadas sobre máquinas agregadas heterogêneas. Tal modelo baseia-se no escalonamento estático dos processos que compõem a aplicação, visando um mapeamento equilibrado de processos aos nós, apoiando-se, para isto, em dados obtidos na monitoração de execuções prévias da aplicação. Isto é realizado de forma transparente ao usuário, refinando-se de forma gradativa e automática o balanceamento de carga ao longo das execuções da aplicação.

Com o objetivo de avaliar este modelo, foi desenvolvida uma ferramenta que implementa o método proposto, a qual foi instalada e configurada no laboratório do Centro de Pesquisa em Alto Desempenho da PUCRS. A ferramenta é composta por:

- sistema de monitoração para obtenção das informações de monitoração das aplicações e gravação em base de dados;
- disparador de aplicações para aplicar o algoritmo de cálculo do número de processos por tipo de nó, baseando-se nos dados de monitoração de execuções prévias da aplicação;
- interface web para análise gráfica do ganho de desempenho das aplicações e comparações entre execuções.

Esta ferramenta mostrou-se bastante útil para analisar-se o comportamento de aplicações paralelas executadas sobre máquinas agregadas heterogêneas, ajudando a encontrar, de forma automática, uma distribuição equilibrada de processos entre os nós, visando uma otimização do desempenho. O ganho de desempenho médio geral das aplicações testadas foi de 109,5%, o ganho de desempenho médio entre as aplicações sem restrições quanto ao número de processos foi de 182,6% e, entre as aplicações com restrições desse tipo foi de 36,3%. Entre as vantagens da utilização da ferramenta, destacam-se:

- libera o usuário da tarefa de ter que decidir o número de processos a ser disparado em cada tipo de nó;
- através do algoritmo de cálculo do número de processos por tipo de nó, realiza uma otimização automática e transparente da aplicação ao longo das execuções;
- mesmo que a ferramenta não consiga encontrar o melhor ajuste para otimizar o desempenho, a interface gráfica fornece ao usuário subsídios para tentar realizar ajustes manuais;

- segue a tendência de integração do sistema de gerência ao sistema de monitoração, como já proposto na ferramenta Clane [49], desenvolvida no CPAD, possibilitando a análise de históricos de execuções;

- pode ser utilizada de forma didática, possibilitando que programadores iniciantes tenham idéia das dificuldades da programação em máquinas agregadas heterogêneas, e ajudando-os a encontrar soluções.

Algumas das desvantagens observadas em relação ao método proposto são:

- otimização da aplicação pode ser alcançada somente após a mesma já ter sido executada pelo menos uma vez;

- dificuldade em lidar com aplicações que possuem restrições quanto ao número de processos;

- impossibilidade de otimizar determinadas aplicações, fazendo com que certas execuções em busca de uma melhor otimização tenham um desempenho inferior à execução inicial.

Com o objetivo de dar continuidade ao trabalho, abaixo relacionam-se algumas propostas para trabalhos futuros:

- refinamento do algoritmo de cálculo do número de processos por nó;

- definição de uma melhor estratégia para lidar com aplicações que possuem restrições quanto ao número de processos;

- melhorias na interface gráfica, afim de prover uma gama maior de informações ao usuário;

- melhoramento do sistema de monitoração, de forma que o sistema não precise ser disparado explicitamente pelo usuário. De forma que seja automaticamente ativado quando o usuário executar alguma aplicação através do disparador `cropt`.

Anexo A Código da aplicação de cálculo do PI

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc, char *argv[])
{
    int myid, numprocs, i, j;
    unsigned long long n;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    while (1)
    {
        if (myid == 0)
        {
            printf("Starting... ");
            n = 2000000000;
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        h   = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs)
        {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
                   0, MPI_COMM_WORLD);
        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n", pi,
        fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

Anexo B *Script* SQL de especificação do banco de dados

```
use ilb;

drop table machines;
drop table machine_types;
drop table executions;
drop table tracking;
drop table execution_machines;
drop table execution_np;
drop table app_nicks;

create table machines
(
    cod      int primary key auto_increment,
    name     varchar(20),
    cod_type int
);

create table machine_types
(
    cod       int primary key auto_increment,
    type_name varchar(20),
    arch      varchar(20),
    num_procs int,
    mem_size  int,
    clock     int
);

create table app_nicks
(
    cod      int primary key auto_increment,
    nickname varchar(50)
);

create table executions
(
    cod         int primary key auto_increment,
    start_time  datetime,
    stop_time   datetime,
    cmdline     varchar(250),
    avg_cpu_use int,
    app_code    int,
    counter     int,
    opt1        int,
    opt2        int
```

```
);

create table execution_machines
(
    cod_execution int,
    cod_machine   int,
    avg_cpu_use   int,
    primary key(cod_execution, cod_machine)
);

create table execution_np
(
    cod_execution    int,
    cod_type         int,
    np               int,
    avg_cpu_type_use int,
    primary key(cod_execution, cod_type)
);

create table tracking
(
    cod_execution int,
    time          datetime,
    machine01     int(4),
    machine02     int(4),
    machine03     int(4),
    machine04     int(4),
    machine05     int(4),
    machine06     int(4),
    machine07     int(4),
    machine08     int(4),
    machine09     int(4),
    machine10     int(4),
    machine11     int(4),
    machine12     int(4),
    machine13     int(4),
    machine14     int(4),
    machine15     int(4),
    machine16     int(4),
    machine17     int(4),
    machine18     int(4),
    machine19     int(4),
    machine20     int(4),
    machine21     int(4),
    machine22     int(4),
    machine23     int(4),
    machine24     int(4),
    machine25     int(4),
    machine26     int(4),
    machine27     int(4),
    machine28     int(4),
```

```
    machine29     int(4),
    machine30     int(4),
    machine31     int(4),
    primary key(cod_execution, time)
);

insert into machine_types values (0, "e800",     "ia32",
                                  2,  256, 1000);
insert into machine_types values (0, "e60",      "ia32",
                                  2,  256,  550);
insert into machine_types values (0, "ia64mono", "ia64",
                                  1,  512,  800);
insert into machine_types values (0, "ia64dual", "ia64",
                                  2, 2048, 1500);

insert into machines values (0, "amazonia01", 1);
insert into machines values (0, "amazonia02", 1);
insert into machines values (0, "amazonia03", 1);
insert into machines values (0, "amazonia04", 1);
insert into machines values (0, "amazonia05", 1);
insert into machines values (0, "amazonia06", 1);
insert into machines values (0, "amazonia07", 1);
insert into machines values (0, "amazonia08", 1);
insert into machines values (0, "amazonia09", 2);
insert into machines values (0, "amazonia10", 2);
insert into machines values (0, "amazonia11", 2);
insert into machines values (0, "amazonia12", 2);
insert into machines values (0, "amazonia13", 2);
insert into machines values (0, "amazonia14", 2);
insert into machines values (0, "amazonia15", 2);
insert into machines values (0, "amazonia16", 2);
insert into machines values (0, "amazonia17", 2);
insert into machines values (0, "amazonia18", 2);
insert into machines values (0, "amazonia19", 2);
insert into machines values (0, "amazonia20", 2);
insert into machines values (0, "amazonia21", 2);
insert into machines values (0, "amazonia22", 2);
insert into machines values (0, "amazonia23", 2);
insert into machines values (0, "amazonia24", 2);
insert into machines values (0, "amazonia25", 3);
insert into machines values (0, "amazonia26", 3);
insert into machines values (0, "amazonia27", 4);
insert into machines values (0, "amazonia28", 4);
insert into machines values (0, "amazonia29", 4);
insert into machines values (0, "amazonia30", 4);
insert into machines values (0, "amazonia31", 4);
```

Referências

[1] Grand Challenge Applications. *http://www-fp.mcs.anl.gov/grand-challenges/*. Último acesso em 14 de Março de 2006.

[2] SILVA, L. Moura e; BUYYA, R. Parallel Programming Models and Paradigms. Em: BUYYA, R. (Ed.). *High Performance Cluster Computing: Programming and Applications, Volume 2*. Upper Saddle River, New Jersey 07458: Prentice Hall PTR, 1999. p. 4–27.

[3] BODEN, N. J. et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, v. 15, n. 1, p. 29–36, 1995.

[4] IBEL, M. et al. High-performance cluster computing using scalable coherent interface. Em: *Proceedings of 7th International Workshop on SCI-based High-Performance Low-Cost Computing*. [S.l.: s.n.], 1997. p. 45–54.

[5] Message Passing Interface Forum MPI. *MPI: A Message-Passing Interface Standard*. [S.l.], 1994. Disponível em: <citeseer.nj.nec.com/article/forum94mpi.html>.

[6] MEUER, H. et al. *Top 500 Supercomputer Sites*. Http://www.top500.org. Último acesso em 14 de Março de 2006.

[7] EL-REWINI, H. Partitioning and Scheduling. Em: ZOMAYA, A. (Ed.). *Handbook of Parallel and Distributed Computing*. New York (USA): McGraw-Hill, 1996. cap. 9, p. 239–273.

[8] BHANDARKAR, M. et al. Adaptive Load Balancing for MPI Programs. *International Conference on Computational Science*, v. 2074, p. 108–117, 2001.

[9] BARAK, A.; LA'ADAN, O. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, v. 13, n. 4–5, p. 361–372, 1998.

[10] ROSE, C. A. F. D.; NAVAUX, P. *Arquiteturas Paralelas*. Porto Alegre: Editora Sagra Luzzatto, 2003.

[11] PLASTINO, A. et al. Load Balancing in SPMD Applications: Concepts and Experiments. Em: YANG, L.; PAN, Y. (Ed.). *High Performance Scientific and Engineering Computing - Hardware/Software Support*. [S.l.]: Kluwer Academic Publishers, 2004. p. 95–107.

[12] BAKER, M.; BUYYA, R. Cluster computing: the commodity supercomputer. *Software Practice and Experience*, v. 29, n. 6, p. 551–576, 1999.

[13] GEIST, A. et al. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press, 1994.

[14] WALKER, D. W. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, v. 20, n. 4, p. 657–673, 1994.

[15] STERLING, T. An Introduction to PC Clusters for High Performance Computing. Em: BAKER, M. (Ed.). *Cluster Computing White Paper*. [S.l.: s.n.], 2000. cap. 1, p. 1–10.

[16] NETTO, M. A. S.; ROSE, C. A. F. D. CRONO: A Configurable and Easy to Maintain Resource Manager Optimized for Small and Mid-Size GNU/Linux Clusters. *Proceedings of the 32nd International Conference on Parallel Processing (ICPP'03)*, p. 555–562, 2003.

[17] FERRETO, T. C.; ROSE, C. A. F. D.; ROSE, L. D. Rvision: An open and high configurable tool for cluster monitoring. Em: *CCGRID*. [S.l.]: IEEE Computer Society, 2002. p. 75–. ISBN 0-7695-1582-7.

[18] FOSTER, I. T. *Designing and Building Parallel Programs*. Boston, MA: Addison-Wesley Publishing Company, 1995. 379 p.

[19] MPIF, M. P. I. F. *MPI-2: Extensions to the Message-Passing Interface*. 1996. Technical Report, University of Tennessee, Knoxville.

[20] GROPP, W. et al. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, v. 22, n. 6, p. 789–828, 1996.

[21] PEISERT, S. P.; BADEN, S. B. *A Programming Model for Automated Decomposition on Heterogeneous Clusters of Multiprocessors*. [S.l.], 2001.

[22] EL-REWINI, H.; LEWIS, T. G.; ALI, H. H. *Task scheduling in parallel and distributed systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN 0-13-099235-6.

[23] CASAVANT, T.; KUHL, J. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, v. 14:2, p. 141–154, 1988.

[24] MULLENDER, S. J. *Distributed Systems*. Wokingham: ACM Press, 1989. 601 p.

[25] TANENBAUM, A. S. *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN 0130313580.

[26] HARCHOL-BALTER, M.; DOWNEY, A. B. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, v. 15, n. 3, p. 253–285, 1997.

[27] MICHAILIDIS, P. D.; MARGARITIS, K. G. Parallel Text Searching Application on a Heterogeneous Cluster of Workstations. Em: *30th International Workshops on Parallel Processing (ICPP 2001 Workshops)*. Valencia, Spain: IEEE Computer Society, 2001. p. 169–175.

[28] BEAUMONT, O. et al. Matrix-Matrix Multiplication on Heterogeneous Platforms. Em: *International Conference on Parallel Processing*. ENS Lyon: [s.n.], 2000. p. 289–298.

[29] BOHN, C. Asymmetric load balancing on a heterogeneous cluster of PCs. *Asymmetric load balancing on a heterogeneous cluster of PCs. MSCE Thesis, AFIT/GE/ENG/99M-02, Graduate School of Engineering, Air Force Institute of Technology (AETC), WrightPatterson AFB OH.*, 1999.

[30] NGUYEN, T. D.; VASWANI, R.; ZAHORJAN, J. Maximizing speedup through self-tuning of processor allocation. *Proceedings of the International Parallel Processing Symposium*, p. 463–468, 1996.

[31] CHRONOPOULOS, A. T. et al. A class of loop self-scheduling for heterogeneous clusters. Em: *CLUSTER*. [S.l.]: IEEE Computer Society, 2001. p. 282–292. ISBN 0-7695-1116-3.

[32] WOLFFE, G. S.; HOSSEINI, S. H.; VAIRAVAN, K. An Experimental Study of Workload Indices for Non-dedicated, Heterogeneous Systems. Em: ARABNIA, H. R. (Ed.). *PDPTA*. [S.l.]: CSREA Press, 1997. p. 470–478. ISBN 0-9648666-8-4.

[33] TERESCO, J. D.; FAIK, J.; FLAHERTY, J. E. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science & Engineering*, v. 7, n. 2, p. 40–50, 2005.

[34] DEVINE, K. et al. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, v. 4, n. 2, p. 90–97, 2002.

[35] STEWART, J. J. D. e C. B. Moler e J. R. Bunch e G. W. *Linpack Users' Guide*. [S.l.]: SIAM, 1979. (Society for Industrial and Applied Mathematics).

[36] BARAK, A. et al. Performance of PVM with the MOSIX Preemptive Process Migration. Em: *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering*. Herzliya: [s.n.], 1996. p. 38–45.

[37] FINK, S. J.; BADEN, S. B.; KOHN, S. R. Efficient run-time support for irregular block-structured applications. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 50, n. 1-2, p. 61–82, 1998. ISSN 0743-7315.

[38] SMITH, P.; HUTCHINSON, N. C. Heterogeneous process migration: The Tui system. *Software Practice and Experience*, v. 28, n. 6, p. 611–639, 1998.

[39] MUELLER, F. *Pthreads Library Interface. Technical report, Florida State University.* 1993.

[40] BOVET, D.; CESATI, M. *Understanding the Linux Kernel, Second Edition*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[41] Mysql AB. *MySQL Reference Manual*. 2004. Http://www.mysql.com/doc/en/. Último acesso em 14 de Março de 2006.

[42] HANSEN, P. B. Model programs for computational science: A programming methodology for multicomputers. *Concurrency*, v. 5, n. 5, p. 407–423, 1993.

[43] FOX, G. Parallel computing comes of age: Supercomputer level parallel computations at caltech. *Concurrency - Practice and Experience*, v. 1, n. 1, p. 63–103, 1989.

[44] WILSON, G. *Parallel Programming for Scientists and Engineers*. Cambridge, MA: MIT Press, 1995.

[45] PRITCHARD, D. Mathematical Models of Distributed Computation. Em: *Proceedings of OUG-7, Parallel Programming on Transputer Based Machines*. [S.l.]: IOS Press, 1988. p. 25–36.

[46] VINTER, B.; BJORNDALEN, J. M. A Comparison of Three MPI Implementations. Em: EAST, D. I. R. et al. (Ed.). *Communicating Process Architectures 2004*. [S.l.: s.n.], 2004. p. 127–136. ISBN 1 58603 458 8.

[47] BAILEY, D. H. et al. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, v. 5, n. 3, p. 63–73, Fall 1991.

[48] VAUGHT, A. Creating animations with pov-ray. *Linux J.*, Specialized Systems Consultants, Inc., Seattle, WA, USA, v. 1997, n. 36es, p. 4, 1997. ISSN 1075-3583.

[49] FERRETO, T. C.; ROSE, C. A. F. D. Clane: Um ambiente para análise comportamental de máquinas agregadas. Em: *Anais da IV Escola Regional de Alto Desempenho (ERAD'2004)*. [S.l.]: IV Escola Regional de Alto Desempenho (ERAD'2004), Pelotas, RS, 2004. v. 1, p. 175–176.