### UNIVERSIDADE FEDERAL FLUMINENSE

### JULIANA MENDES NASCENTE SILVA

## Modelo de Escalonamento e Representação de Clusters de Máquinas Multicore

NITERÓI 2013

### UNIVERSIDADE FEDERAL FLUMINENSE

### JULIANA MENDES NASCENTE SILVA

## Modelo de Escalonamento e Representação de Clusters de Máquinas Multicore

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Doutor. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Orientadora: Lúcia Maria de Assumpção Drummond

> Co-orientadora: Maria Cristina Silva Boeres

> > NITERÓI 2013

#### Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S586	<ul> <li>Silva, Juliana Mendes Nascente</li> <li>Modelo de escalonamento e representação de clusters de máquinas</li> <li>multicore / Juliana Mendes Nascente Silva. – Niterói, RJ : [s.n.],</li> <li>2013.</li> <li>97 f.</li> </ul>
	Tese (Doutorado em Computação) - Universidade Federal Fluminense, 2013. Orientadores: Lúcia Maria de Assumpção Drummond, Maria Cristina Silva Boeres.
	1. Arquitetura de redes de computadores. 2. Arquitetura multicore. 3. Algoritmo de escalonamento. 4. Cluster. I. Título.
	CDD 004.6

### Modelo de Escalonamento e Representação de Clusters de Máquinas Multicore

JULIANA MENDES NASCENTE E SILVA

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Doutor.

Aprovada por:

Prof<sup>a</sup>. D.Sc. Lúcia Maria de Assumpção Drummond / IC-UFF (Presidente)

Alla linto 1 A. rela Prof<sup>a</sup>. Ph.D. Alba Cristina M. A. de Melo / UNB Incre Prof<sup>a</sup>. D.Sc. Cristiana Barbosa Bentes / UERJ ~ Prof<sup>a</sup>. D.Sc. Simone de Lima Martins / IC-UFF J. Cebell Prof. Ph.D. Eugene Francis Vinod Rebello/ IC-UFF

Niterói, 03 de dezembro de 2013.

 $\dot{A}$  minha família, meu alicerce.

## Agradecimentos

Agradeço à Deus, meu refúgio.

Agradeço à minha família pela força, amor e paciência durante todo este tempo. Mamãe meu exemplo de vida, de caridade e humildade. Meu pai, sempre foi pra mim um exemplo de determinação. Nana e Vi, vocês que mesmo de longe torceram e acreditaram em mim. Nana, o que seria de mim sem suas orações, sem seu exemplo de fé? Vi, com você aprendi a ser alegre, a ver o lado bom das coisas e obrigada por nos ter dado a Lu, minha sobrinha linda, que fez de tantos momentos difíceis momentos mais que especiais.

Às minhas grandes orientadoras Cristina e Lúcia, pela paciência e pelos desafios lançados.

Aos meus amigos. Agradeço cada conselho, conversas, risadas, taças de vinho, idas à cantareira. Jacques, não tenho palavras para agradecer sua dedicação à nossa amizade.

A minha futura família. Marcelo, obrigada pelo amor e pela paciência durante minha ausência.

## Resumo

Normalmente, os sistemas computacionais de alto desempenho atuais estão sendo formados por máquinas *multicores*. Devido ao paralelismo inerente nesta arquitetura, aplicações que exigem um alto poder computacional podem ser executadas de forma eficiente neste ambientes. Entretanto, o grau de paralelismo utilizável em cada máquina *multicore* pode ser limitado pela contenção de acesso à hierarquia de memória. Atualmente, a arquitetura *multicore* introduziu algumas características diferentes daquelas já observadas em ambientes de memória compartilhada e distribuída. Um exemplo é que conjuntos de cores podem compartilhar diferentes subconjuntos de memória. Assim, para alcançar bom desempenho é necessário que a alocação das tarefas da aplicação nos *cores* disponíveis seja baseado em especificações e modelos que consideram não somente as características do processador, mas também a contenção de memória. Neste trabalho, é proposta uma Representação de Clusters de Multicore (RCM) e um modelo de escalonamento (MECM) que captura características relevantes de desempenho de sistema *multicore* tais como a influência da hierarquia de memória e a contenção na execução da aplicação. Para avaliar o modelo foi desenvolvida uma estratégia de balanceamento de carga de uma aplicação branch-and-bound e um escalonamento baseado em memória de uma aplicação sintética que foram baseados no MECM. Os resultados apresentados mostram que a eficiência do modelo e sua aplicabilidade em ambientes *multicore*.

**Palavras-chave**: Arquiteturas *multicore*, hierarquia de memória, *branch-and-bound*, *cluster*, escalonamento.

## Abstract

The latest trends in high-performance computing systems show an increasing demand for the use of large scale multicore systems in an efficient way, so that high compute-intensive applications can be executed reasonably well. However, the exploitation of the degree of parallelism available at each multicore component can be limited by the poor utilization of the memory hierarchy. Nowadays, the multicore architecture introduces some features that are distinct of these already observed in shared memory and distributed environments. One example is that subsets of cores can share different subsets of memory. In order to achieve high performance it is imperative that a careful allocation scheme of an application is carried out on the available cores, based on a scheduling specification that considers not only processors characteristics but also memory contention. This work proposes the *Multicore Cluster Representation* (MCR) and a scheduling model (MECM) that captures relevant performance characteristics in multicores systems such as the influence of memory hierarchy and contention. Better performance was achieved when a load balancing strategy for a branch-and-bound application applied to the Partitioning Sets Problem and a scheduling based memory for a synthetic application was based on MECM showing its efficiency and applicability to modern systems.

**Keywords**: multicore architecture, memory hierarchy, branch-and-bound applications, cluster computing, process/tasks scheduling.

# Glossário

CC	:	Duas tarefas alocadas em <i>diferentes cores</i> , mas compartilhando a
		mesma cache;
CMP	:	Duas tarefas alocadas em $cores$ que não compartilham memória
		cache, mas compartilham memória principal;
NCM	:	Duas tarefas alocadas em <i>cores</i> de máquinas diferentes, onde a
		memória global de cada máquina não é compartilhada;
RCM	:	Representação de um <i>Cluster</i> de <i>Multicore</i> ;
MECM	:	Modelo de Escalonamento para <i>Cluster</i> de <i>Multicore</i> ;
SPP	:	Set Patitioning Problem;
B&B	:	Algoritmo branch-and-bound;
$SBB_{SPP}$	:	Algoritmo $\mathit{branch}\text{-}\mathit{and}\text{-}\mathit{bound}$ sequencial aplicado ao Problema de
		Particionamento de Conjuntos;
$PBBNLB_{SPP}$	:	Algoritmo $\mathit{branch-and-bound}$ paralelo aplicado ao Problema de
		Particionamento de Conjuntos, sem balanceamento de carga;
$PBB_{SPP}$	:	Algoritmo $\mathit{branch-and-bound}$ paralelo aplicado ao Problema de
		Particionamento de Conjuntos, com balanceamento de carga ba-
		seado no MECM;
$OCLB_{SPP}$	:	Algoritmo $\mathit{branch-and-bound}$ paralelo aplicado ao Problema de
		Particionamento de Conjuntos, com balanceamento de carga ba-
		seado em Overlay Centric Tree;
GAD	:	Grafo Acíclico Direcionado;
LS	:	List Scheduling;
$LS_{BM}$	:	List Scheduling Baseado em Memória;

# Sumário

Lis	sta de	e Figuras	ix
Lis	sta de	e Tabelas	xi
1	Intro	odução	1
<b>2</b>	Rev	isão dos Modelos arquiteturais e modelo de aplicação	5
	2.1	Arquiteturas de computadores	6
	2.2	Modelos Arquiteturais	9
		2.2.1 Modelos para arquiteturas de memória compartilhada	9
		2.2.2 Modelos para arquiteturas de memória distribuída	10
		2.2.3 Modelos para arquiteturas de memória híbrida	13
	2.3	Modelo de Aplicação	16
3	Ava	aliação de desempenho de um cluster de Multicore	19
	3.1	Análise da Fase de Computação da Aplicação Sintética	22
	3.2	Análise da Fase de Comunicação da Aplicação Sintética	24
4	Clus	sters de Máquinas Multicore: representação e modelos	31
	4.1	Representação de <i>Clusters</i> de Máquinas <i>Multicore</i> (RCM)	31
	4.2	Modelo da Aplicação	32
	4.3	Modelo de Escalonamento de Tarefas para arquiteturas representadas por RCM	32

5.1	B&B Sequencial aplicado ao Problema de Particionamento de Conjuntos .	35
	5.1.1 Lower Bound	36
	5.1.2 Branching	38
	5.1.3 Experimentos computacionais do $B\&B$ sequencial	38
5.2	Branch-and-Bound paralelo aplicado ao Problema de Particionamento de Conjunto - $PBB_{SPP}$	39
	5.2.1 Framework de Balanceamento de Carga	41
	5.2.2 Algoritmos do Framework de Balanceamento de Carga do $PBB_{SPP}$	43
	5.2.3 Questões relativas à implementação do <i>Framework</i> de Balancea- mento de Carga	52
	5.2.3.1 Tamanho da lista	52
	5.2.3.2 Alocação e acesso às listas de nós	53
	5.2.3.3 Valor do parâmetro $NT$	54
Res	ultados do Balanceamento de Carga proposto para um $B\&B$ paralelo baseado	
no N	MECM	56
6.1	Comparando o $PBB_{SPP}$ com o $PBBNLB_{SPP}$ ( $B\&B$ paralelo sem Balanceamento de Carga)	57
6.2	Comparando o $PBB_{SPP}$ e o $OCLB_{SPP}$	60
	6.2.1 Escalabilidade	65
Algo	oritmo de Escalonamento baseado no modelo de escalonamento proposto	68
7.1	Algoritmo de escalonamento List Scheduling	69
7.2	Algoritmo List Scheduling Baseado em Memória, $LS_{BM}$	73
7.3	Resultados e conclusões	80
Con	clusões	88
	5.2 <b>Res</b> <b>no I</b> 6.1 6.2 <b>Alg</b> 7.1 7.2 7.3 <b>Con</b>	<ul> <li>5.1 Deep equation opticate do l'isocial de l'artechnication de Conjunto 1.</li> <li>5.1.1 Lower Bound</li></ul>

#### Referências

90

# Lista de Figuras

2.1	Exemplo de um <i>cluster</i> de <i>multicore</i>	8
2.2	Exemplo de uma aplicação representada por um GAD	17
3.1	Análise de execução de duas tarefas em dois <i>cores</i> de uma mesma máquina.	23
3.2	Análise da execução de 2, 4 e 8 tarefas usando oito $cores$ de uma máquina.	24
3.3	Execução das fases de computação e de comunicação, seguindo o modelo de comunicação <i>ping-pong.</i>	25
3.4	Tempo total da aplicação com 2 tarefas executando a fase de computação e trocando uma mensagem no modelo <i>ping-pong</i> durante a fase de comunicação.	27
3.5	Tempo total da aplicação com 2 tarefas representadas por processos execu- tando a fase de computação e trocando uma mensagem no modelo <i>ping-pong</i> durante a fase de comunicação	28
3.6	Execução das fases de computação e de comunicação, seguindo o modelo de comunicação <i>all-to-all.</i>	28
3.7	Análise do tempo total de execução de 2 tarefas executando as fases de computação e comunicação (modelo <i>all-to-all</i> )	29
5.1	Exemplo de instância do SPP.	36
5.2	Tempo de execução e tamanho dos nós para a instância I90-400-003	41
5.3	Ilustração do <i>Framework</i> de Balanceamento de Carga em um <i>cluster</i> de máquinas <i>multicore</i>	43
5.4	Exemplo de requisição de carga à <i>thread</i> vizinha.	45
5.5	Exemplo de requisição de carga às <i>threads</i> da mesma máquina	46
5.6	Exemplo de requisição de carga à <i>thread</i> gerenciadora	48
5.7	A <i>thread</i> gerenciadora compartilha a carga recebida entre as <i>threads</i> tra- balhadoras de sua máquina que estão sem carga	50

Exemplo de requisição da carga gerenciadora para as <i>threads</i> trabalhadoras.	51
A thread gerenciadora envia carga para a thread gerenciadora que requisitou carga.	51
Exemplo uma árvore lógica que representa um ambiente contendo 6 $\mathit{cores.}$ .	61
Comparação entre o $Un\_Factor$ do $PBB_{SPP}$ e do $OCLB_{SPP}$ com $Grau_{max}=3$ , 7 e 15 para as instâncias apresentadas na Tabela 6.3	63
Speedup do $PBB_{SPP}$ e $OCLB_{SPP}$ com $Grau_{max}=3, 7$ e 15 para as instâncias apresentadas na Tabela 6.3.	64
Grafo <i>G</i>	69
Escalonamento gerado pelo $LS$ para o grafo $G$ da Figura 7.1 $\ .\ .\ .\ .$	72
Existe <i>core</i> sem tarefas em $M_{pred} = \{M_0\}$	74
Todos os <i>core</i> de $M_{pred} = \{M_0\}$ possuem tarefas alocadas	75
Todos os <i>core</i> de $CM$ possuem tarefas alocadas	76
Todos os <i>core</i> de $CM$ possuem tarefas alocadas e as caches de $M_{pred}$ estão sobrecarregadas	76
Escalonamento gerado pelo $LS_{BM}$ para o grafo $G$ da Figura 7.1	80
Tipos de Grafos Acíclicos Direcionados de entrada	81
Exemplo de uma Intree com 16 tarefas	83
Exemplo de escalonamento do primeiro nível de um grafo Intree pelo (a) $LS$ e (b) $LS_{BM}$	84
	Exemplo de requisição da carga gerenciadora para as threads trabalhadoras. A thread gerenciadora envia carga para a thread gerenciadora que requisitou carga

# Lista de Tabelas

2.1	Principais modelos de computação para arquiteturas de memória compar- tilhada	11
2.2	Modelos de computação para arquiteturas de memória distribuída	14
2.3	Modelos de computação para arquiteturas de memória híbrida	17
3.1	Tempo gasto pela operação bloqueante de comunicação usando o <i>MPI_Send</i> nas alocações CC, CMP, NCM	24
3.2	Tempo gasto por duas tarefas durante a fase de comunicação: modelo <i>ping-</i> <i>pong.</i>	25
5.1	Comparação entre o $B\&B$ sequencial $(SBB_{SPP})$ e o resolvedor $Cplex$	39
5.2	Exemplo dos níveis, tamanho e tempo de execução dos nós para as instâncias I90-400-0.03, I100-500-0.03, I110-750-0.03 e I200-650-0.02-100 do $SPP$ .	40
6.1	Resultados obtidos pela execução da versão do $B\&B$ sem balanceamento de carga para o $SPP$ , $PBBNLB_{SPP}$ , utilizando duas máquinas do <i>cluster</i> (16 <i>cores</i> )	59
6.2	Resultados obtidos pelo $PBB_{SPP}$ utilizando duas máquinas do $cluster.\ .\ .$	59
6.3	Comparação entre o mecanismo de balanceamento de carga do $PBB_{SPP}$ e a versão paralela do $B\&B$ com balanceamento de carga baseado na árvore <i>Overlay-Centric</i> , $OCLB_{SPP}$	63
6.4	Informações sobre a comunicação entre as tarefas $PBB_{SPP}$ e do $OCLB_{SPP}$ , com $Grau_{max} = 7$ .	65
6.5	Tamanho das mensagens (KB) do tipo <i>Load</i> trocadas entre as tarefas no $PBB_{SPP}$	65
6.6	Execução do $PBB_{SPP}$ em quatro máquinas (32 threads) e oito máquinas (64 threads)	66

6.7	Execução do $OCLB_{SPP}$ em quatro máquinas (32 threads) e oito máquinas	
	(64 threads)	67
7.1	Resumo da definição das variáveis utilizados no $LS_{BM}$	79
7.2	Tempo de execução em segundos da execução do algoritmo $LS_{BM}$ e $LS$ para grafos Outtree. Cada tarefa tem o $\mu_v$ associado gerado aleatoriamente entre 1MB e 10MB.	82
7.3	Tempo de execução em segundos da execução do algoritmo $LS_{BM}$ e $LS$ para grafos Inttree	85
7.4	Tempo de execução em segundos da execução do algoritmo $LS_{BM}$ e $LS$ para grafos Diamante.	86
7.5	Tempo de execução em segundos da execução do algoritmo $LS_{BM}$ e $LS$ para grafos Irregulares	86
7.6	Comparação do tempo de execução teórico, $Makespan$ , com o tempo real obtido pela aplicação quando suas tarefas foram escalonamento com o $LS$ .	87

## Capítulo 1

## Introdução

O uso de dois ou mais núcleos de processamento (*cores*) por processador viabilizou o aumento do número de transistores disponíveis dentro de um único *chip*. Estas máquinas podem possuir um ou mais *sockets*, referidos neste trabalho como processadores, e cada processador pode possuir dois ou mais *cores*. Este tipo de processador, denominado *multicore*, aumentou o desempenho dos sistemas computacionais, tornando-o dominante nos sistemas encontrados atualmente. Cada *core* dentro de um processador é uma unidade de processamento independente que possui, normalmente, uma ou mais níveis de *caches* privadas. Diante disto, cada *core* pode executar simultaneamente diferentes códigos e pode ser considerado um recurso de processamento independente. Assim, arquiteturas *multicores* são apropriadas ao paralelismo.

Diante da popularização das arquiteturas *multicore*, as plataformas de alto desempenho, como os *clusters*, também aderiram a esta nova tecnologia e, hoje, são compostas de máquinas *multicore* conectadas por canais de rede dedicados ou não.

Os *clusters* de *multicore* apresentam uma estrutura hierárquica de memória onde os *cores* pertencentes a um mesmo processador podem compartilhar algum nível de cache. Por outro lado, *cores* pertencentes a diferentes processadores de uma mesma máquina, ou nó do *cluster*, podem compartilhar uma memória principal (RAM ou DRAM) e *cores* pertencentes a diferentes nós não compartilham recursos de memória [74, 86].

Considerar a hierarquia de memória na implementação de aplicações paralelas poderia melhorar o desempenho destas aplicações. Por exemplo, o tempo de comunicação das tarefas da aplicação poderia diminuir se fosse utilizada a memória compartilhada disponível nestes ambientes ao invés de utilizar trocas de mensagens via rede. Ainda, tarefas que se comunicam com mais frequência poderiam ser alocadas a *cores* que compartilham cache evitando também, comunicações na memória principal [5, 83, 86].

Entretanto, dependendo da memória necessária para comunicar e computar as tarefas da aplicação, alocar muitas tarefas em *cores* que compartilham cache poderia exceder a capacidade da mesma. Assim, muitos acessos à memória principal seriam necessários podendo causar um gargalo nos canais de acesso a esta memória. O gargalo pode piorar muito o desempenho da aplicação [59, 70, 74, 83], podendo, em alguns casos, ser pior do que utilizar a rede para trocar informações entre as tarefas.

Usar as características do ambiente para melhorar o desempenho da aplicação não é uma ideia nova. Para tanto, é necessário que exista um modelo que represente as características mais relevantes do ambiente onde a aplicação irá executar. Definir um modelo que descreva estas características não é uma tarefa fácil devido, por exemplo, a quantidade de máquinas diferentes existentes. Entretanto, os algoritmos de escalonamento ou balanceamento de carga poderiam melhorar consideravelmente o tempo de execução da aplicação se fossem propostos de acordo com um estudo prévio do ambiente de execução.

Neste trabalho, foi realizado um conjunto extenso de testes de uma aplicação sintética que identifica possíveis gargalos promovidos pelos recursos de memória compartilhados e seu impacto na comunicação e computação das tarefas. Logo, foi proposta uma *Representação de Cluster de Multicore* (RCM) que considera a hierarquia de memória dos ambientes *multicore* e um modelo de escalonamento de tarefas. O modelo considera três níveis de comunicação: i) a comunicação através de memória cache compartilhada, que acontece entre as tarefas que são alocadas aos *cores* de um mesmo processador; ii) a comunicação, também utilizando memória compartilhada, mas entre tarefas alocadas a *cores* de processadores diferentes e iii) a comunicação entre tarefas alocadas a *cores* de máquinas diferentes. Considerar os diferentes níveis de comunicação e os gargalos dos recursos compartilhados em estratégias de balanceamento de carga e escalonamento de tarefas poderia melhorar o tempo de execução da aplicação, uma vez que o ambiente de execução será aproveitado em sua totalidade.

Para avaliar e validar o RCM e o modelo de escalonamento baseado em memória proposto, foram desenvolvidos um *framework* de balanceamento de carga para um algoritmo *branch-and-bound* paralelo aplicado ao problema de Particionamento de Conjunto  $(PBB_{SPP})$  e um algoritmo de escalonamento estático para uma aplicação sintética do tipo GAD (Grafos Acíclicos Direcionados). Tanto os procedimentos do algoritmo de escalonamento, quanto o *framework* consideram as características dos *clusters* de *multicore* capturadas no modelo de escalonamento de tarefas proposto. O framework de balanceamento de carga foi utilizado na implementação de uma algoritmo branch-and-bound (B&B). O framework foi baseado no modelo proposto e o uso de seus procedimentos durante a execução do B&B evitam o gargalo de acesso à memória compartilhada, monitorando a carga de cada tarefa da aplicação. A versão paralela do B&B foi implementada baseada na versão sequencial do B&B aplicado ao problema de Particionamento de Conjunto. Esta versão sequencial utiliza uma heurística simples para obter limites inferiores para o problema e assim, agilizar a execução da árvore de enumeração do B&B.

O escalonamento estático, utilizado para mapear as tarefas da aplicação sintética no ambiente, foi baseado no algoritmo heurístico List Scheduling (LS). No LS, inicialmente é atribuída uma prioridade para cada tarefa da aplicação, logo, a tarefa de maior prioridade é atribuída a uma unidade de processamento escolhida de acordo com uma função objetivo. O algoritmo de escalonamento proposto, chamado  $LS_{BM}$ , é baseado no LS e considera que a prioridade da tarefa deve ser relativa à quantidade de memória que a mesma utiliza durante sua execução. Além disto, a unidade de processamento escolhida, ou *core*, deve ser baseada na quantidade de memória disponível para executar as tarefas alocadas a ele. O  $LS_{BM}$  foi comparado com uma versão tradicional do LS e mostra que considerar o modelo de escalonamento proposto, pode reduzir consideravelmente, na maioria dos casos, o tempo de execução da aplicação.

Dentre as principais contribuições deste trabalho, pode-se destacar:

- 1. Uma nova representação para clusters de multicore e um modelo de escalonamento de tarefas que considera não somente as características relevantes em relação ao processamento e a comunicação de tarefas via diferentes níveis de memória e rede, mas também como estas características são influenciadas pela quantidade de memória requerida por cada tarefa da aplicação. Para tanto, o impacto da quantidade de memória requerida para processar e comunicar as tarefas no custo de execução e comunicação foram medidos e modelados. O objetivo foi propor um modelo que inclui os custos de comunicação e computação associados com a contenção de diferentes níveis de memória.
- 2. Baseado no modelo, foi proposta uma estratégia de balanceamento de carga em que a hierarquia de memória é considerada durante a decisão sobre a quantidade de dados que devem ser alocados a cada tarefa e como estas tarefas se comunicam. Deste modo, a carga computacional gerada pela aplicação deve ser dividida e balanceada entre as tarefas de modo que gargalos de acesso à memória sejam evitados. Para

avaliar o modelo e o mecanismo de balanceamento de carga proposto, os procedimentos desenvolvidos foram utilizados em uma aplicação real baseada no algoritmo *branch-and-bound*. Vários trabalhos sobre algoritmos *branch-and-bound* paralelos são encontrados na literatura [15, 27, 32, 40, 35, 65, 67, 70, 72, 77, 19], entretanto poucos foram desenvolvidos considerando sistemas de computação com ambos uso de memória: distribuída e compartilhada.

3. Também baseado no modelo, foi proposta uma estratégia de escalonamento de tarefas estático que evita os gargalos de acesso à memória e maximiza o tempo de execução de uma aplicação sintética do tipo GAD.

Este trabalho está organizado como se segue. No Capítulo 2, é apresentada uma revisão detalhada da literatura a respeito dos modelos de computação e comunicação que consideram memória compartilhada, distribuída e ambas. No terceiro capítulo, é apresentada a aplicação sintética utilizada para medir e avaliar os *clusters de multicore*, bem como o conjunto de testes realizados com suas respectivas conclusões. O RCM e o modelo de escalonamento de tarefas são apresentados no Capítulo 4. No Capítulo 5, é apresentado como o modelo e a estratégia de balanceamento de carga foram aplicados ao algoritmo *branch-and-bound*, que neste caso, resolve o Problema de Particionamento de Conjuntos. A versão sequencial em que foi baseado o algoritmo paralelo também é apresentada neste capítulo. Os resultados obtidos são apresentados no Capítulo 6. No Capítulo 7, é apresentado o algoritmo  $LS_{BM}$  baseado no modelo de escalonamento proposto. Finalmente, no Capítulo 8, é descrita a conclusão deste trabalho.

## Capítulo 2

# Revisão dos Modelos arquiteturais e modelo de aplicação

Desde o desenvolvimento das primeiras máquinas paralelas, vários autores têm proposto modelos e representações que definem tais sistemas de computação. No modelo são definidos parâmetros que caracterizam os custos associados ao acesso dos recursos compartilhados, como por exemplo, a memória, a latência dos canais de comunicação e o tempo gasto para computar tarefas nos elementos computacionais disponíveis. Tais modelos são geralmente estendidos e novos parâmetros são redefinidos ou incluídos à medida que a arquitetura das máquinas que compõem os sistemas evoluem.

Apesar de ser um grande desafio definir um modelo que represente com acurácia as características mais relevantes das diversas arquiteturas encontradas atualmente, ter um modelo ou uma representação do ambiente de execução é indispensável para desenvolver aplicações paralelas eficientes. Ainda, o conhecimento *a priori* dos custos relacionados à computação e à comunicação das tarefas da aplicação em um determinado sistema poderiam ser utilizados para evitar sobrecargas e atrasos durante a computação da aplicação, e assim, maximizar o seu desempenho.

Um modelo deve representar uma boa abstração do ambiente de execução e deve ser usado como base de decisões sobre como as tarefas da aplicação devem ser executadas. Este capítulo apresenta um breve histórico dos modelos de arquitetura encontrados na literatura e também, o modelo de aplicação que é utilizado para representar as características de aplicações paralelas comumente abordadas. A combinação destes modelos é essencial para auxiliar a solução de problemas de escalonamento, como será discutido no Capítulo 4.

Entretanto, para um melhor entendimento dos modelos apresentados, a próxima seção

apresenta alguns conceitos a respeito da arquitetura de computadores e algumas características adicionais sobre *clusters* de *multicore* que são utilizados ao longo deste capítulo.

### 2.1 Arquiteturas de computadores

A tendência em arquiteturas *multicore* é que estes sejam formados por não somente por dois, quatro ou oito *cores* mas que sejam formados por dezenas ou milhares de *cores*. Além disto, os *chips* podem possuir *cores multithreading*, memórias privadas ou *cores* heterogêneos.

Diante da diversidade das arquiteturas *multicores* disponíveis, para melhor apresentação do ambiente de execução considerado neste trabalho, são introduzidos nesta seção alguns conceitos e classificações de arquiteturas de computadores.

Com o intuito de classificar as arquiteturas de computadores foi proposta uma taxonomia, denominada Taxonomia de Flynn que distingue a arquitetura de acordo com o fluxo de instruções e dados. Um processador serial que executa apenas uma instrução sobre um dado é definido como SISD (Single Instruction, Single Data). Computadores paralelos que executam apenas uma instrução mas sobre diferentes tipos de dados são chamados SIMD (Single Instruction, Multiple Data). Nesta classificação são inseridos os processadores vetoriais, e, atualmente, este tipo de processadores estão sendo utilizado nas GPUs (Graphics Processor Units). Processadores que executam múltiplas instruções sobre um único dado não são muito usuais mas são denominados MISD (Multiple Instruction, Single Data) e finalmente, os processadores *multicore* fazem parte dos processadores que são classificados como MIMD (Multiple Instruction, Multiple Data). Os processadores SIMD são muitos propícios ao paralelismo e alguns fabricante têm utilizado estes processadores associados ao uso de processadores MIMD para melhorar o desempenho da aplicação. Um exemplo é o IBM Cell que utiliza os *cores* SIMD para acelerar a execução de aplicação que possuem muito dados a serem processados [54]. Já os demais *cores* disponíveis na máquina são de propósito gerais, MIMD [30].

Em relação à memória, as máquinas que possuem memória compartilhada podem ser divididas em UMA e NUMA. Nas máquina UMA processadores possuem acesso à memória com tempo uniforme, já nas máquinas NUMA os processadores possuem acesso não uniforme à memória. Normalmente, nestas últimas máquinas o tempo de acessar a memória de outro processador é muito maior do que acessa à memória local. Máquinas de memória distribuída são formadas por máquinas interconectadas através de uma rede, dedicada ou não, que permite a comunicação entre os processos. Ambiente de alto desempenho, atualmente, são formados por máquinas híbridas, que consiste na união de uma ou mais arquiteturas para formar um ambiente de execução.

Em máquinas do tipo SMP (Symetric Multiprocessor) com acesso uniforme à memória, todos os processadores acessam a memória principal através de um único barramento por onde passa apenas uma instrução. Sendo assim, oferece a todos os processadores a mesma latência de acesso à um espaço de dados. As memórias caches são utilizadas junto aos processadores para diminuir a utilização deste barramento e assim, melhorar o tempo de latência entre as instruções de acesso à memória. Nestas arquiteturas cada processador possui uma memória cache e, necessariamente, os processadores devem acessar as cópias mais recentes dessa memória. A existência de cópias do mesmo dado gera então, o problema de coerência de cache. Este problema é crítico e afeta diretamente o desempenho das aplicações paralelas.

Existem várias métodos de coerência de cache que podem ser implementados a nível de software ou no nível de hardware. Este último, apesar da necessidade de hardware especializados é mais utilizado nas máquinas atuais. Tais métodos são considerados dinâmicos pois lidam com a coerência em tempo de execução, Normalmente os protocolos de coerência de cache podem ser classificados como *snoopy* ou *directory* de acordo com a forma utilizada para informar os diversos processadores de que houve uma escrita de um dado, e que a cópia local pode estar desatualizada.

O snoopy possui os controladores de caches locais que são responsáveis por controlar as informações sobre o estado dos dados na cache. Para comunicar com as demais caches da arquitetura a ação que deve ser realizada sobre os dados compartilhados é realizado um *broadcast*. Entretanto, quando o número de processadores é muito grande este protocolo tende a ser substituído pelo protocolo *directory* que é considerado mais escalável. Neste último protocolo, é utilizado um controlador central para manter a coerência e também guardam a informação de localização das cópias de dados. É armazenado em um diretório, na memória principal, informações contendo o estado global do conteúdo das varias caches e as atualizações são realizadas analisando estes estados. A implementação destes protocolos depende da topologia de memória, quantidade de barramentos, entre outras configurações do hardware da arquitetura. Mais detalhes a respeito podem ser encontrados em [60].

O problema de coerência de cache é crítico e afeta diretamente o desempenho das aplicações principalmente quando os processadores são *multicore*. Alguns estudos têm

sido realizados de modo que sejam propostos protocolos que apresentem bom desempenho considerando os vários níveis de cache destas arquiteturas [60].

Uma característica nova dos *cluster* de *multicore* com o acesso à memória é uniforme é que estas máquinas possuem vários graus de compartilhamento de cache em diferentes níveis, diferente das máquinas SMP que, normalmente, possuem uma cache por processador. Assim, o tempo de acesso a um dado neste ambiente de execução depende de onde este dado está disponível na hierarquia de memória e se o mesmo é compartilhado ou não. Além disto, devido ao aumento da quantidade de *cores* a hierarquia de memória tem se tornado cada vez mais profunda e complexa. Muitas arquiteturas possuem *clores* com cache L1 privada, caches de nível L2 são compartilhadas por dois ou mais *cores* e caches de nível L3 são compartilhadas por quatro ou mais *cores* [74, 60].

Neste trabalho, o ambiente de execução considerado é um *cluster* de *multicore* com o acesso à memória é uniforme, onde todos os *cores* são MIMD e possuem a mesma frequência de processamento. A Figura 2.1 exemplifica um *cluster* de *multicore* composto por duas máquinas *multicores*. Neste exemplo cada máquina possui dois processadores e cada processador possui oito *cores*. Cada *core* possui uma cache L1 privada e dois *cores* de um mesmo processador compartilham uma memória cache L2. Finalmente, todos os *cores* de uma mesma máquina compartilham uma memória principal com acesso uniforme e as duas máquinas do *cluster* compartilha um *link* de rede.



Figura 2.1: Exemplo de um *cluster* de *multicore*.

### 2.2 Modelos Arquiteturais

Modelos de arquitetura são utilizados para representar as características mais relevantes de um ambiente de execução, como por exemplo: o custo de acesso à memória, heterogeneidade e disposição dos componentes de arquitetura (processadores, memórias, etc), latência dos canais de comunicação e contenção de acesso a recursos compartilhados.

Inicialmente, qualquer compartilhamento de dados em sistemas de computação paralela era realizado em uma memória global, compartilhada por todos os elementos de processamento. Com o advento das redes de computadores e a melhora dos canais de comunicação, originaram-se os sistemas de memória distribuída como *clusters* e grades, onde a troca de dados entre as tarefas é realizada através da troca de mensagens [71]. Atualmente, com o advento da tecnologia *multicore*, os *clusters* são formados por um conjunto de máquinas *multicore* conectadas via canais de rede dedicados ou não. Máquinas *multicore* possuem mais de um elemento de processamento (*core*) e compartilham um sistema hierárquico de memória [74, 86]. Nos *clusters* de *multicore*, as tarefas podem se comunicar através da memória compartilhada e/ou através da troca de mensagens.

Modelos de arquitetura foram sendo propostos mediante a evolução dos sistemas de computação paralela, considerando o modo de acesso à memória e a disposição dos elementos de processamento. Alguns modelos se destacaram e serviram de base para estudo e desenvolvimento de aplicação paralelas e de novos modelos. As próximas seções apresentam os principais modelos encontrados na literatura, que estão divididos em: modelos para arquiteturas ou sistemas de memória compartilhada, modelos para arquitetura ou sistemas de memória distribuída, e modelos para arquitetura ou sistemas de memória híbrida. Estes últimos possuem em sua formação tanto memória compartilhada quanto distribuída.

#### 2.2.1 Modelos para arquiteturas de memória compartilhada

O modelo *Parallel Random-Access Machine* (PRAM) [38] se tornou muito conhecido pela sua simplicidade e basicamente consiste de um número de processadores que computam uma mesma instrução por unidade de tempo, sobre diferentes dados. A comunicação acontece via memória compartilhada e, também, ocorre em uma unidade de tempo [51]. Devido à sua simplicidade e universalidade, o modelo PRAM teve uma grande aceitação pela comunidade teórica e um grande número de algoritmos paralelos baseados neste modelo foram propostos. Entretanto, o PRAM não é realístico, pois não incorpora parâmetros que são considerados críticos em sistemas paralelos, principalmente parâmetros relacionados à sobrecarga de comunicação e à sincronização implícita neste modelo. Vários autores estenderam o PRAM [58] com o intuito de inserir tais parâmetros, como: contenção de memória [42, 43, 61], execuções assíncronas [24, 41], latência de comunicação e largura de banda [1, 2, 3, 4, 43, 52, 64].

Dentre os modelos citados, o modelo *Hierarchical Memory Model* (HMM) proposto por Aggarwal *et al* em [1] teve como objetivo capturar o efeito da hierarquia de memória presente nas máquinas. HMM considera que o acesso randômico à localização da memória x requer o tempo de  $\lceil log x \rceil$  ao invés de ser um tempo constante como era considerado pelos modelos anteriores. Em [2], o HMM foi estendido e foi proposto o modelo *Hierarchical Memory with Block Transfer* (HMBT), onde um bloco que ocupa localizações consecutivas da memória pode ser copiado a um custo constante após o primeiro acesso ter sido contabilizado. Já em [52], HMBT é estendido e o modelo *Parallel HMBT* (P-HMBT) considera que transferências de dados entre níveis diferentes de memória podem acontecer concorrentemente.

Em [8], o *Parallel Memory Hierarchy* (PMH) modela o computador como uma árvore de módulos de memória, sendo os processadores as folhas desta árvore. Tal estrutura é utilizada para representar o custo de transferência de um bloco de dados entre os nós da árvore. O modelo *Uniform Memory Hierarchy* (UMH) [7] inclui na representação do PMH o custo de movimentar dados entre os diferentes níveis da hierarquia de memória.

A Tabela 2.1 apresenta um resumo dos principais modelos de memória compartilhada que consideram e não consideram a hierarquia de memória da arquiteturas de memória compartilhada.

#### 2.2.2 Modelos para arquiteturas de memória distribuída

Sistemas de memória distribuída se tornaram uma alternativa para a criação de sistemas escaláveis. Entretanto, o desempenho de aplicações que executam neste tipo de ambiente pode ser limitado pela largura de banda e pela latência de comunicação [64, 71, 76]. Muitos autores avaliaram o comportamento das arquiteturas de memória distribuída com o objetivo de propor um modelo paralelo de propósito geral. O Modelo de Memória Distribuída consiste de uma conjunto de processadores cada um com sua memória local, conectados por canais organizados em alguma topologia. A comunicação acontece através da troca de mensagens.

Tabela 2.1:	Principais	modelos	de	computação	para	arquiteturas	de me	emória	compa	rti-
lhada										

	Ν	/Iemória Compartilhada	
Data	Modelo		Referências
1988	PRAM	Considera um conjunto de processadores idên- ticos que operam de forma síncrona (SIMD). Existe uma memória global que é comparti- lhada entre os processadores	[64]
		Execução assíncrona	
1989	APRAM	Incorpora execuções assíncronas no modelo PRAM	[24]
1989	Asynchronous PRAM	Considera também execuções assíncronas, além de algumas restrições para execução em máquinas MIMD	[41]
	Latência, cor	ntenção de memória e largura de banda	
1984	Randomized and Deterministic Simulations of PRAMs	Considera máquinas paralelas em que a memó- ria compartilhada é organizada em módulos, e somente uma célula de cada módulo pode ser acessada por vez. Modela também algoritmos que possuem acesso irrestrito à memória	[61]
1989	BPRAM	Considera custos diferenciados para acesso a blocos contínuos de memória	[3]
1990	LPRAM	Considera que a quantidade de memória local é ilimitada e que acesso de escrita à memória global não pode ser realizado simultaneamente pelas tarefas	[4]
1994	QRQW PRAM	Utiliza uma fila para gerenciar o acesso à me- mória e considera que o custo de acesso à me- mória é função do tamanho dessa fila	[42]
1997	QSM	Extensão do modelo QRQW. Utiliza um mo- delo de abstração para simplificar a descrição de algoritmos e análises. Modela a memória locas e os limites da largura de banda de me- mórias remotas <i>Hierarquia de Memória</i>	[7, 68]
1987	HMM	Considera que memórias de níveis diferen- tes possuem diferentes tempos de acesso. O acesso à memória local $x$ depende de uma fun- ção $f(x)$	[1]
1987	HMBT	Incorpora a possibilidade de mover blocos de dados no modelo HMM	[2]
1992	UMH	Inclui o custo de movimentação de dados atra- vés da memória hierárquica. Apresenta tam- bém uma versão paralela do modelo UMH	[7]
1993-94	PHMM, PHMBT	Apresentam versões paralelas para os modelos HMM e HMBT	[52, 91]

Um dos primeiros trabalhos a incluir no modelo o custo de sincronização associado à troca de mensagens entre os elementos de processamento é apresentado em [90]. O modelo *Bulk-Synchronous Parallel* (BSP) representa um conjunto de elementos de processamento, suas velocidades e o custo de sincronização de comunicação. A aplicação executada no BSP é organizada em blocos síncronos, denominados *superpassos*. Cada processador recebe um conjunto de tarefas que executam uma computação local, depois, as mensagens referentes a esta computação são enviadas e/ou recebidas e, finalmente, é realizada a sincronização de mensagem entre os processadores. Vale ressaltar que a ausência de um modelo padrão incentivou muitos pesquisadores a estabelecerem uma ponte entre aplicações paralelas e os sistemas de computação paralelas, como é o caso do BSP, onde é definido no modelo como a aplicação deve ser executada.

Com o advento dos *clusters*, modelos com maior acurácia continuaram a ser pesquisados. Em [93] foi proposto o modelo HBSP, *Heterogeneous Bulk-Synchronous Parallel*, que estende o BSP, considerando que os elementos de processamento são heterogêneos, logo, tanto a velocidade quanto a capacidade de processamento podem ser diferentes entre eles.

Com o crescimento da utilização de redes de computadores como ambientes de alto desempenho, o modelo LogP [25] foi proposto com o objetivo de ser um modelo computacional mais realista, distinguindo o custo de transmissão das mensagens do custo de processamento das mensagens. No LogP são representados o número de processadores, a latência da transmissão, o *gap* entre as mensagens, que representa o custo de processamento da mensagem e o custo do envio e recebimento de mensagens. A ocorrência da comunicação e computação assíncrona são consideradas as características chaves deste modelo.

Outros trabalhos que estendem o modelo LogP foram propostos, entre eles, o modelo LogGP [6] que caracteriza, no parâmetro gap, o custo associado ao envio de mensagens longas, incluindo no modelo o parâmetro de largura de banda. Já no modelo LogGPS [50], é considerado o custo de sincronização que ocorre quando são enviadas mensagens longas sobre a biblioteca de comunicação de alto nível, como o MPI. Em [39] é proposto o LoPC, que utiliza os parâmetros requeridos pelo LogP para predizer o custo de contenção associado ao processamento de envio de mensagens em multiprocessadores ou em redes de computadores. O envio de mensagem é uma comunicação ponto-a-ponto, e portanto, requer movimento de dados da memória local do processo que está enviando a mensagem para o espaço local do processo que irá receber a mensagem. Os modelos  $Log_n P$  e  $Log_3P$  [22] incluem os custos de processamento relacionados ao *middleware* (aplicação responsável pela comunicação) na representação do custo da comunicação distribuída.

Vale notar que, quando comparados os modelos BSP e LogP, alguns autores classificaram o BSP como mais adequado para o desenvolvimento de aplicações paralelas, pois apresenta uma melhor abstração do ambiente de execução, enquanto o LogP oferece um gerenciamento dos recursos mais eficiente. A escolha do BSP pode ocasionar perda de desempenho, entretanto, esta perda pode ser compensada pela facilidade de desenvolvimento da aplicação [17, 69]. Em [69] é mostrado que o QSM (descrito na Tabela 2.1) pode ser adaptado para representar memória distribuída e pode ser utilizado para modelar a comunicação com uma melhor abstração do que quando utilizados os modelos BSP e o LogP. Seguindo o advento dos *clusters*, os modelos em [22, 29, 85] capturaram com mais precisão o *overhead* e a latência do envio e recebimento de mensagens. Nestes trabalhos, o custo de envio e recebimento de mensagens depende também do tamanho das mensagens transmitidas.

A Tabela 2.2 apresenta alguns modelos referenciados na literatura para memória distribuída.

Com a evolução das arquiteturas, sistemas paralelos de memória híbrida, onde computadores de memória distribuída são compostos por máquinas com memória compartilhada, como os *multicores*, tornaram-se uma realidade. Aplicações desenvolvidas para executar neste tipo de ambiente podem apresentar melhores desempenhos se tais sistemas forem estudados e suas características modeladas. A próxima seção apresenta com maiores detalhes estes sistemas.

#### 2.2.3 Modelos para arquiteturas de memória híbrida

Como já descrito, *Clusters* de *multicores* são formados por máquinas *multicore*, multiprocessadas ou não, conectadas por canais de rede sugerindo uma hierarquia de memória. No primeiro nível da hierarquia, aplicações que possuem granularidade fina podem apresentar bom desempenho, enquanto aplicações de granularidade grossa podem ser executadas no segundo nível. O modelo de paralelismo hierárquico inerente a estes sistemas é fundamental para desenvolver aplicações paralelas eficientes.

Subconjuntos de núcleos de processamento ou *cores* em uma máquina *multicore* podem compartilhar diferentes níveis de memória. Por exemplo, um pequeno subconjunto de *cores* podem compartilhar cache L2, enquanto outro subconjunto, de maior cardinalidade,

Memória Distribuída						
Data	Modelo	Considerações	Referências			
1988	Latência	Custo de comunicação entre os processadores	[58]			
		depende da latência e da largura de banda dos				
		canais de comunicação				
1990	BSP	Considera o custo de sincronização de mensa-	[17, 68, 90,			
		gens durante a comunicação	69]			
1996	LogP	Considera que a comunicação não acontece de	[17, 25, 68,			
		forma síncrona (diferente do BSP). A comuni-	69]			
		cação requer a troca de mensagens entre pro-				
		cessadores diferentes, sendo estes, interligados				
		por um canal de comunicação que possui uma				
		capacidade finita de transmissões simultâneas				
2000	HBSP	Extensão do BSP que considera que os proces-	[93]			
		sadores podem possuir capacidades de execu-				
		ção diferentes.				
1995 a 2001	LogGP, LogGPS,	Extensões do modelo LogP. Consideram sin-	[6, 39, 50, 55]			
	LoPC, LogP-HMM	cronização antes do envio de mensagem, o ta-				
		manho da mensagem enviada, contenção de				
		recursos e a execução do modelo LogP em uma				
		hierarquia de memória (Modelo LogP-HMM)				
2004	Modelo de Comu-	São consideradas três fases distintas: fase de	[29, 85]			
	nicação Realístico	envio que é o tempo de preparo da mensagem				
	para Computa-	a ser enviada acrescido ao tempo de interação				
	ção Paralela em	com a interface de rede; a fase de transferência				
	Clusters Computa-	que é o tempo gasto para que a mensagem seja				
	cionais	transferida da memória de um processo para				
		o outro; e a fase da sobrecarga de recebimento				
		referente ao tempo gasto durante o tratamento				
		de mensagens recebidas				
2007	$Log_nP, Log_3P$	Consideram na avaliação de desempenho da	[22]			
		comunicação distribuída o efeito da fragmen-				
		tação dos dados e custo do middleware utili-				
		zado durante a comunicação (tempo da cópia				
		de dados da memória local até a memória des-				
		tino)				
2008	Modelo de execução	O modelo de comunicação representa não só	[31, 76]			
	em grades compu-	o custo de comunicação, mas representa tam-				
	tacionais	bém as características do ambiente que pode				
		possuir canais de comunicação com diferentes				
		latências. O modelo de computação considera				
		que cada processador possui um índice de re-				
		tardo que define o tempo de execução da tarefa				
		naquele processador.				

pode compartilhar uma cache L3, enquanto a memória global é compartilhada por todos os *cores* da máquina [11, 23, 75, 89]. Como consequência disto, um modelo que represente o compartilhamento dos níveis de hierarquia de memória é ainda um grande desafio [74, 73], já que, além de definir os graus de compartilhamento dos diferentes níveis de cache, este grau varia de uma arquitetura *multicore* para a outra.

Máquinas *multicore* não podem ser tratadas apenas como arquiteturas de memória compartilhada (multiprocessadores), já que, tipicamente, neste caso, o compartilhamento de dados acontece por todos os processadores no mesmo nível da hierarquia de memória (ou seja, na memória principal). Já em *multicores*, os processadores e os *cores* podem possuir um grau variado de compartilhamento de cache nos diferentes níveis da hierarquia de memória. Logo, os modelos de arquitetura de memória compartilhada das máquinas multiprocessadas mas que não são *multicore* não são uma boa alternativa para representar os *clusters* de multicore.

O Unified Multicore Model (UMM) proposto em [75] assume que um conjunto de cores compartilham o nível i de cache, como L2, um subconjunto maior de cores compartilham nível i + 1, como a memória cache L3, por exemplo, e que a capacidade da cache é a mesma para todas as caches de um mesmo nível. Assim, o UMM modela vários tipos de processadores multicores com diferentes níveis de compartilhamento de cache em diferentes níveis da hierarquia de memória. Os autores mostram que a aplicação denominada S-span captura dependências em um GAD e é utilizado para obter o limite inferior dos custos do tráfego de comunicação em processadores com um único core ou multicores.

O sistema de hierarquia de memória presente nos *clusters* atuais pode ser representado por três níveis de comunicação: intra-processador, quando a comunicação acontece entre dois *cores* de um mesmo processador; inter-processador, quando a comunicação é realizada entre *cores* de processadores diferentes de uma mesma máquina; e, inter-máquina, onde a comunicação acontece entre *cores* de máquinas diferentes. Os custos de comunicação são diferentes nos três níveis [62, 82]. Mais especificadamente, em [82] é definido um modelo analítico que considera os diferentes níveis de memória e é especificado um parâmetro denominado *afinidade entre threads*, que depende da quantidade de dados que será trocada entres elas. Segundo o modelo, threads que possuem maior *afinidade* devem ser alocadas a *cores* que compartilham níveis mais baixos de memória (ou seja, cache), para evitar maiores custos de comunicação que ocorrem quando as threads estão em diferentes processadores. Segundo os autores, o custo de comunicação é menor quando esta acontece nos níveis mais baixos de memória. Entretanto, neste modelo, o tamanho da memória não é considerado e, além disto, muitas threads podem ser alocadas à mesma cache compartilhada, e consequentemente, a quantidade de falhas de cache pode ser alta, diminuindo o desempenho da aplicação [13, 23, 95].

Diante da avaliação de várias aplicações, em [23] é considerado que a representação do custo de comunicação depende da hierarquia de memória e é mostrado que otimizar a troca de mensagens intra-processadores e inter-processadores é tão importante quanto evitar troca de mensagens inter-máquinas em *clusters* de multicore. Além disto, nos trabalhos [23, 95] é mostrado que o desenvolvimento de técnicas que aproveitem a localidade de dados pode melhorar muito o desempenho da aplicação.

Valiant, em [90], estende o modelo BSP e propõe o modelo teórico Multi-BSP que é um modelo hierárquico com um número arbitrário de níveis e que define a quantidade de memória que pode ser acessada em uma quantidade de tempo fixo de um dos processadores. Os autores usam o modelo para mostrar que é viável e benéfico escrever algoritmos Multi-BSP independentes de entradas de dados previamente definidos, ou seja, livre de parâmetros, sendo assim, os passos da computação, a comunicação e custos de sincronização são fatores constantes multiplicativos de acordo com uma dada aplicação. Outros autores, [22, 89], seguem caminhos opostos e definem modelos considerados extensões do LogP incluindo parâmetros que representam a hierarquia de memória dos *multicore*, como, por exemplo, o custo de transferência (cópia de dados) entre os níveis da hierarquia de memória. Com o intuito de simplificar o modelo, em [89] são considerados apenas dois níveis de comunicação: intra-máquinas e inter-máquinas e nenhuma informação a respeito do gargalo de acesso aos recursos compartilhados são utilizados para modelar o desempenho dos *clusters* de *multicore*.

A Tabela 2.3 apresenta um resumo das principais características que devem ser consideradas em um modelo de execução para máquinas híbridas.

### 2.3 Modelo de Aplicação

Uma aplicação paralela é um conjunto de tarefas de computação que cooperam, através da troca de informação, para obter um resultado comum. O *Modelo de Aplicação* define as características de execução deste tipo de aplicação, incluindo parâmetros referentes ao tempo gasto com a computação e a dependência entre tarefas, entre outros. O modelo de aplicação GAD é amplamente utilizado na literatura ([31, 33, 51, 56, 71, 76, 82, 83, 94, 95]) e consiste em representar a aplicação através de um grafo acíclico direcionado (ou GAD),

Memória Híbrida						
Data	Considerações	Referências				
1990	Propõe um modelo Multi-BSP que considera os níveis de	[90]				
	hierarquia de memória com uma quantidade mínima de pa-					
	râmetros (máquinas SMP)					
2007-2009	extensões do LogP incluindo parâmetros que representam	[22, 89]				
	a hierarquia de memória dos <i>multicore</i>					
2009	Subconjuntos de diferentes cores compartilham subconjun-	[75]				
	tos diferentes de memória cache					
2007-2009	Três níveis de comunicação são considerados: intra-chip,	[11, 23, 75,				
	inter-chip e inter-no 89, 62					
2007-2009	Consideram que o tempo de comunicação é minimizado se	[11, 23, 75,				
	a comunicação acontecer nos níveis mais baixos de cache	89]				
2004-2007-2010	Consideram que fazer um bom uso da localidade de dados	[13, 23, 95,				
	pode melhorar muito o desempenho da aplicação	33]				

Tabela 2.3: I	Modelos de	computação	para arquiteturas	de	memória	híbrida
---------------	------------	------------	-------------------	----	---------	---------

denotado por  $G = (V, E, \epsilon, \omega)$ , onde o conjunto de vértices, V, representa as tarefas; E, a relação de precedência entre as tarefas;  $\epsilon_v$ , a quantidade de trabalho ou o peso computacional associado a cada tarefa;  $\omega(u, v) \in E$ , é a representação da quantidade de unidades de dados transmitidas entre as tarefa u para v.

O conjunto de predecessoras e sucessoras da tarefa u são, respectivamente,  $pred(v) = u|(u, v) \in E$  e  $suc(v) = z|(v, z) \in E$ . Vale ressaltar que o peso computacional da tarefa v é indivisível. A Figura 2.2 representa um GAD, contendo oito tarefas. O valor próximo ao vértice é referente ao peso da tarefa representado por aquele vértice e o valor próximo a aresta o peso de comunicação entre as tarefas.



Figura 2.2: Exemplo de uma aplicação representada por um GAD.

O modelo GAD é amplamente utilizado, inclusive para modelar aplicações que executam em arquiteturas de *multicore* [10, 20, 21, 33, 48, 74, 75]. Entretanto, tarefas com o mesmo peso computacional podem apresentar tempos de execução diferentes dependendo do subconjunto de memórias cache do *core* onde a tarefa foi alocada, mesmo considerando uma máquina homogênea. Sendo assim, existe um *gap* entre o tempo teórico estimado e real de execução da tarefa [10]. O mesmo acontece a respeito do custo de comunicação, o nível de comunicação predominante entre as tarefas da aplicação influencia diretamente no desempenho. Parâmetros que refletem com mais precisão as características da aplicação paralela podem ser utilizados para evitar os gargalos de desempenho do ambiente de execução representado no modelo arquitetural. Logo, a combinação dos modelos arquitetura e de aplicação são essenciais para guiar algoritmos de escalonamento de tarefas que têm por objetivo alocar tarefas aos elementos de processamento ou *cores* de modo que o desempenho da aplicação seja maximizado.

No próximo capítulo, são apresentadas avaliações realizadas em *clusters* de máquinas *multicore* com o intuito de identificar as contenções ocasionados pelo acesso compartilhado nos diversos níveis de hierarquia de memória.

## Capítulo 3

# Avaliação de desempenho de um cluster de Multicore

Apesar de serem propícios à execução de aplicações paralelas, os *clusters* de máquinas *multicore* possuem características que podem limitar o desempenho de mais aplicações. Se as tarefas são divididas em unidades de computação independentes e são alocadas a *cores* que compartilham uma memória cache, a competição por este recurso pode aumentar o tempo de execução da aplicação, devido, por exemplo, a falhas de acesso à cache [9, 23, 36, 37, 59, 66, 80, 88, 96]. Por outro lado, quando as tarefas necessitam compartilhar dados, o desempenho da aplicação pode ser melhorado aproveitando as informações contidas na cache. Assim, evitam-se trocas de dados nos níveis mais altos da hierarquia de memória ou via rede [23, 33, 55]. Entretanto, também neste caso, a quantidade de tarefas que acessam a estrutura de dados compartilhada pode gerar contenção de acesso a esta estrutura, diminuindo o desempenho da aplicação [11, 53, 63].

Este capítulo apresenta algumas avaliações realizadas em um *cluster* de *multicore* com o objetivo de identificar os limites de desempenho presentes neste ambiente devido ao compartilhamento de memória. Neste trabalho, devido às máquinas disponíveis, foi considerado o compartilhamento nas caches de nível L2 e os gargalos de acesso à memória principal. Estas avaliações podem ser extendidas à máquinas que possuem níveis mais altos de compartilhamento de cache, como, por exemplo, cache L3.

As avaliações foram realizadas utilizando uma aplicação sintética que foi implementada baseada na aplicação proposta em [44, 80]. Em [44] é proposto um *benchmark* para obter parâmetros do hardware, como a quantidade de níveis de cache na hierarquia e seus tamanhos e a topologia das caches compartilhadas. A aplicação sintética utiliza o modo de acesso à memória descrito em [44] e foi implementada variando a quantidade de dados utilizados durante a computação.

A aplicação consiste em duas ou mais tarefas que executam duas fases bem definidas: computação e comunicação. Durante a fase de computação, ilustrada no Algoritmo 1, cada tarefa v aloca um único vetor de inteiros de tamanho  $\mu(v)$ . Então, executa um *loop* que objetiva percorrer o vetor, durante  $\alpha$  iterações. No ambiente disponível para executar os experimentos, as máquinas são compostas por processadores do tipo Xeon E5410 - Harpertown, que utilizam linhas de cache de tamanho 512 bytes. Com o objetivo de acessar diferentes linhas de cache em cada referência ao vetor, o mesmo foi percorrido utilizando passos de 1KB, como ilustrado no Algoritmo 1 [80].

Além disto, atualmente, os compiladores otimizam agressivamente os códigos na tentativa de aproveitar a estrutura do *pipeline* do processador. Estas otimizações podem atrapalhar a avaliação de acesso à memória e para evitá-las, o valor do passo é armazenado no vetor, logo este deve ser sempre acessado (linha 7 do algoritmo). Finalmente, o vetor foi alocado com a função **posix\_memalign** [46] que garante que seu início é alocado a uma nova linha de cache. Sendo assim, esta linha não será compartilhada e não será invalidada quando requisitada por outra tarefa. O valor de  $\alpha$  foi 10000, para todos os testes. Este valor foi escolhido para aumentar o tempo de processamento da tarefa e diminuir o impacto da influência da execução dos processos do sistema operacional durante a avaliação. Após a fase de computação a tarefa executa a fase de comunicação, detalhada abaixo.

#### Algorithm 1 Fase de Computação da tarefa v.

```
1: for all i \leftarrow 0 to \mu(v) do

2: vector[i] = 1024/sizeof(int);

3: end for

4: for all i \leftarrow 0 to \alpha do

5: j = 0;

6: while j < \mu(v) do

7: j = j + vector[j];

8: end while

9: end for

10: FaseDeCommunicacao(v);
```

A fase de comunicação consiste no envio de uma mensagem de uma tarefa para a outra tarefa. O modo como esta comunicação é implementada depende de onde as tarefas estão alocadas. Se as tarefas estão alocadas na mesma máquina, a comunicação acontece via memória compartilhada e semáforos são utilizados para prevenir condições de corrida. Por outro lado, se as tarefas são alocadas em máquinas diferentes, uma mensagem é enviada via rede.

Três situações foram implementadas durante a fase de comunicação da linha 10 do Algoritmo 1. A primeira consiste no envio de uma mensagem de uma tarefa para a outra, tal que uma tarefa executa um envio bloqueante e a outra um recebimento não bloqueante. A segunda versão da *FaseDeCommunicacao(v)* consiste no envio seguindo o modelo *ping-pong* e finalmente, a terceira versão consiste na troca de mensagens entre as duas tarefas que executam, simultaneamente, um envio não bloqueante e um recebimento bloqueante. O Algoritmo 2 apresenta a execução da função *FaseDeCommunicacao(v)*. Inicialmente, a tarefa v executa a fase de computação descrita no Algoritmo 1 e, em seguida, aloca um vetor para enviá-lo à tarefa  $u_i$ . O vetor enviado da tarefa v para a  $u_i$  é denominado msg\_v[] e tem tamanho definido como  $\omega(v, u_i)$ . Logo, v executa uma operação de envio e se prepara, alocando um vetor msg\_u[], de mesmo tamanho, para receber a mensagem da tarefa  $u_i$ . Os três tipos de comunicação implementados pela função *FaseDeCommunicacao(v)* serão mais detalhadamente descritos na Seção 3.2.

 Algorithm 2 FaseDeComunicacao(v).

 1: msg\_v[]  $\leftarrow$  AlocaVetor( $\omega(v, u_i)$ );

 2: for all  $i \leftarrow 0$  to  $\omega(v, u_i)$  do

 3: msg\_v[i] = 'A';

 4: end for

 5: Envia msg\_v[] para  $u_i$ ;

 6: msg\_u[]  $\leftarrow$  AlocaVetor( $\omega(u_i, v)$ );

 7: Recebe msg\_u[] de  $u_i$ ;

Todos os experimentos descritos foram executados em pelo menos dois nós de um cluster conectado por um switch Gigabit Ethernet. Cada nó possui um processador quad-core (Intel Xeon E5410 - Harpertown). Cada core possui uma cache L1 privada (64 KB) e compartilha uma cache L2 (6MB) com outro core do mesmo processador. A quantidade total de memória cache por processador é de 12MB, resultando em 24MB por nó. Todos os cores de um mesmo nó possuem acesso uniforme à memória RAM de 16GB. O sistema operacional é Cent OS 5.3, kernel 2.6.18. A cache L1 é dividida em 32KB para instrução e 32KB para dados. Tanto a cache L1 quanto a L2 são diretamente mapeadas, set-associative, sendo a L1 de 8 e a L2 de 24. A aplicação foi implementada com Intel MPI versão 4.0.0.028 e Posix Threads foi usado para criar threads. O PAPI (Performance Application Programming Interface) [49] foi utilizado para avaliar detalhes do desempenho da aplicação. No PAPI, é definido um conjunto de rotinas capazes de acessar os contadores de eventos hardware. O eventos contabilizados pelo sistema podem ser listados através da execução da instrução papi\_avail. Na execução da aplicação sintética foram contabilizados
os eventos: i) PAPI\_L2\_TCA, que armazena o número total de acessos à cache L2; ii) PAPI\_L2\_TCH, que calcula o total de cache *hits* em relação ao total de acessos à cache L2; iii) PAPI\_L2\_TCM que contabiliza a quantidade de cache L2 *misses*. Threads foram utilizadas para representar as tarefas da aplicação.

Para avaliar a influência do compartilhamento de memória durante a execução de tarefas da aplicação, as seguintes alocações foram consideradas:

- i. duas tarefas foram alocadas em *diferentes cores*, mas compartilhando a mesma cache (CC);
- ii. duas tarefas foram alocadas em *cores* que não compartilham memória cache, mas compartilham memória principal (CMP);
- iii. duas tarefas foram alocadas em *cores* de máquinas diferentes (NCM), onde a memória global de cada máquina não é compartilhada.

Para alocar as tarefas a um *core* específico, a *system call* set\_affinity() [5, 33] foi usada. Além disto, as tarefas do sistemas operacional foram isoladas em *cores* que não estavam sendo utilizados pela aplicação sintética, utilizando a *system call* taskset().

## 3.1 Análise da Fase de Computação da Aplicação Sintética

Neste experimento, duas tarefas independentes  $v_1$  e  $v_2$ , que não se comunicam, foram executadas seguindo as alocações previamente definidas: CC e CMP. Note que neste experimento as tarefas somente percorrem o vetor *vector*[], como descrito no Algoritmo 1, e nenhum envio ou recebimento de mensagem ocorre.

Como apresentado na Figura 3.1, considerando a quantidade de dados alocados entre 3MB e 6MB por cada tarefa, a estratégia CMP apresenta desempenho superior do que a CC. O número de *cache misses* reduz o desempenho das tarefas que compartilham cache já que a soma de dados alocados pelas tarefas,  $\mu(v_1) \in \mu(v_2)$ , é maior do que a capacidade da cache. A alocação CMP pode reduzir o tempo em até 14.88% quando comparada com a CC. Para  $\mu(v) > 6MB$ , v = 1, 2, as tarefas precisam de mais memória do que a capacidade da cache, aumentando assim, a quantidade de acessos à memória global. Para dados maior que 8MB as tarefas começam a apresentar o mesmo desempenho.



Figura 3.1: Análise de execução de duas tarefas em dois cores de uma mesma máquina.

A Figura 3.2 apresenta o tempo de execução da aplicação quando as tarefas são executadas na mesma máquina mas divididas entre todos os *cores*. O número de tarefas utilizado foi n = 2, 4, 8, não mais que uma tarefa foi executada por *core*. Para n = 2, 4 as tarefas foram alocadas em *cores* que não compartilham cache. A diferença entre o tempo de 2 e 4 tarefas começa a ficar maior quando o tamanho dos dados é maior que 6MB, devido ao gargalo de acesso à memória principal. Quando mais que 2MB de dados são alocados por tarefa, para n = 8, o tempo de execução da aplicação aumenta bastante, mostrando que as tarefas alocadas a uma máquina não devem ultrapassar a capacidade da cache.



Figura 3.2: Análise da execução de 2, 4 e 8 tarefas usando oito cores de uma máquina.

## 3.2 Análise da Fase de Comunicação da Aplicação Sintética

Nas avaliações desta seção, duas tarefas executam tanto a fase de computação quanto a fase de comunicação.

No primeiro experimento, a tarefa v executa inicialmente a fase de computação, como descrito no Algoritmo 1 e então aloca um outro vetor e o envia para a outra tarefa u. Na fase de computação o tamanho do vetor varia de 1MB até 10MB, enquanto o tamanho da mensagem enviada da tarefa v para u,  $\omega(v, u)$ , é de 1MB, 2MB e 8MB, para cada teste. As duas tarefas são alocadas utilizando as estratégias CC, CMP e NCM. A Tabela 3.1 apresenta o tempo gasto durante a comunicação utilizando a função bloqueante  $MPI\_Send$  da biblioteca MPI [81]. Pode-se perceber que o tempo de comunicação quando as tarefas executam na mesma máquina é desprezível.

**Tabela 3.1:** Tempo gasto pela operação bloqueante de comunicação usando o *MPI\_Send* nas alocações CC, CMP, NCM.

$\mu(v)$ Alloc	1MB	<b>2</b> MB	3MB	4MB	5MB	6MB	<b>7</b> MB	8MB	9MB	10MB
				$\omega(u,v) =$	: 1MB mes	sage				
CC	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001
CMP	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001
NCM	0,074050	0,074154	0,074178	0,074095	0,074185	0,074185	0,074200	0,074115	0,074162	0,074194
$\omega(u,v) = 4$ MB message										
CC	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001
CMP	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000002	0,000001	0,000001
NCM	0,345687	0,345788	0,345731	0,345706	0,345729	0,345763	0,345771	0,345788	0,345780	0,345723
$\omega(u,v)=8{ m MB}~{ m message}$										
CC	0,0000008	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000002
CMP	0,0000014	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000001	0,000002
NCM	0,7022743	0,702273	0,702338	0,702368	0,702299	0,702308	0,702349	0,702374	0,702367	0,702362

Para melhor avaliar o custo associado à comunicação, duas tarefas executam a fase

de computação seguida da fase de comunicação que, neste teste, corresponde à troca de mensagens entre as duas tarefas seguindo o modelo *ping-pong*. A Figura 3.3 ilustra a execução das duas fases. A parte pontilhada superior da figura representa a fase de computação, que é a mesma descrita no Algoritmo 1. Já, a parte inferior da figura, representa a fase de comunicação, onde é alocado um vetor para armazenar a mensagem a ser trocada entre as tarefas. Assim, *u* aloca um vetor msg\_u[] e envia-o para a tarefa *v*. *v* recebe msg\_u[] e aloca o vetor msg\_v[] e envia-o para *u*. A aplicação termina quando *u* recebe a mensagem de *v*. Note que as operações de transferência de mensagens utilizadas são bloqueantes. A mensagem enviada é um conjunto de caracteres copiado para msg[]depois de sua alocação. O tempo gasto durante a fase de comunicação é apresentado na Tabela 3.2. Para as estratégias CC e CMP o tempo é muito parecido, entretanto, quando as mensagens são enviadas via rede o custo de comunicação fica bem mais alto do que para as demais estratégias.



**Figura 3.3:** Execução das fases de computação e de comunicação, seguindo o modelo de comunicação *ping-pong*.

**Tabela 3.2:** Tempo gasto por duas tarefas durante a fase de comunicação: modelo *ping-pong.* 

$\omega(u,v)$	CMP	CC	NCM
512KB	0,0000022	0,0000021	0,2537343
1MB	0,0000023	0,0000023	0,5244678
4MB	0,0000030	0,0000030	2,0644913
8MB	0,0000047	0,0000046	4,1381708
12MB	0,0000049	0,0000050	6,3099084

A Figura 3.4 apresenta o tempo total da aplicação para as estratégias CMP (3.4(a)), CC (3.4(b)) e NCM (3.4(c)). O tempo total da aplicação sintética corresponde a altura das barras. A parte mais escura da barra é referente ao tempo gasto durante a fase de comunicação, logo, a parte mais clara refere-se ao tempo associado à execução da fase de computação. O tamanho do vetor na fase de computação foi de 1MB até 10MB para os seguintes tamanhos de mensagens:  $\omega(u, v) = 512$ KB, 1MB, 4MB, 8MB e 12MB.

As estratégias CC e CMP apresentam resultados muito similares em todos os casos, mas a estratégia CC apresenta um tempo ligeiramente menor para tamanho de vetor menor que 3MB, enquanto a estratégia CMP, apresenta resultados um pouco melhores quando o vetor é maior que 3MB e menor que 6MB.

Comparando estas duas estratégias com a estratégia NCM quando o vetor de dados é menor que 6MB, CC e CMP apresentam melhor desempenho do que a NCM. Apesar do tempo gasto na fase de computação ter sido menor para NCM, o tempo gasto para enviar as mensagens, principalmente maiores que 4MB faz com que seu desempenho seja muito pior do que as estratégias CC e CMP.

Entretanto, quando o vetor de dados ultrapassa o tamanho da cache, ou seja, para  $\mu(v) \ge 6MB$ , v = 1, 2 observa-se que o tempo de execução da estratégia NCM é afetado pelo tamanho das mensagens trocadas entre as tarefas. Para tamanho de mensagens,  $\omega(u, v)$ , menores do que 4MB, a estratégia NCM apresenta melhor desempenho, apesar de o tempo associado à fase de comunicação desta estratégia ser muito mais alto do que nas CC e CMP. Neste caso, o gargalo de acesso à memória se mostra tão custoso quanto ao envio de mensagens pela rede. Contudo, para  $\omega(u, v) \ge 4MB$ , o menor custo associado à fase computação não é o suficiente para melhorar o desempenho da aplicação, e as estratégias CC e CMP, apresentam resultados muito melhores.

Vale ressaltar que, quando threads são utilizadas para representar tarefas, a comunicação entre as threads alocadas em máquinas diferentes é realizada atravésda thread principal do processo. Sendo assim, cópias internas da mensagem são necessárias, uma vez que a versão do MPI utilizada não é thread-safe, ou seja, apenas uma thread pode chamar funções do MPI por vez. A Figura 3.5 apresenta o tempo total da aplicação quando as tarefas são representadas por processos para as estratégias CMP e NCM. O tempo de comunicação entre tarefas alocadas em máquinas diferentes é menor do que nos testes anteriores, pois não são necessárias cópias internas de mensagens. Mas, considerando o tempo total de execução, os resultados são os mesmos apresentados quando as tarefas da aplicação são representadas por threads.

O próximo teste teve como objetivo avaliar o custo associado à comunicação quando todas as tarefas que executam a aplicação trocam mensagens entre si, denominado nesse trabalho, modelo *all-to-all*. Neste modelo, após  $v \in u$  executarem a fase de computação, v envia uma mensagem para  $u \in u$  envia mensagem para v. A operação de envio e



**Figura 3.4:** Tempo total da aplicação com 2 tarefas executando a fase de computação e trocando uma mensagem no modelo *ping-pong* durante a fase de comunicação.

recebimento não são bloqueantes. Este teste se assimila a algumas aplicações reais onde não são necessários pontos de sincronização durante a troca de mensagens entre as tarefas. Um exemplo de aplicação deste tipo é o algoritmo *branch-and-bound* que será detalhado no Capítulo 5. A Figura 3.6 ilustra a fase de comunicação *all-to-all*.

O tempo total da aplicação é apresentado na Figura 3.7. O tamanho do vetor de dados novamente variou de 1MB até 10MB e o tamanho da mensagem, variou de 512KB até 8MB, e seus tempos de execução são apresentados, respectivamente nos gráficos das figuras 3.7(a), 3.7(b), 3.7(c) e 3.7(d).



**Figura 3.5:** Tempo total da aplicação com 2 tarefas representadas por processos executando a fase de computação e trocando uma mensagem no modelo *ping-pong* durante a fase de comunicação.



**Figura 3.6:** Execução das fases de computação e de comunicação, seguindo o modelo de comunicação *all-to-all*.

Os resultados deste modelo foram simulares ao do modelo *ping-pong*, entretanto o tempo total da aplicação foi um pouco menor quando comparado ao modelo anterior já que as operações não são bloqueantes, isto é, foram utilizadas as funções MPL\_Isend para o envio de mensagem e MPL\_Iprobe seguida pela função MPL\_Recv para receber a mensagem. Assim, as mesmas conclusões podem ser observadas e são apresentadas de

forma mais sucinta abaixo:

• Em relação ao tamanho do vetor de computação, se  $\mu(v) < 6MB$ :

(a) Quando comparadas às estratégias CC e CMP, o desempenho destas estratégias são muito similares, entretanto, no intervalo de  $\mu(v) \ge 3MB$  e  $\mu(v) \le 6MB$ , a estratégia CMP, apresenta melhores resultados, devido ao acesso compartilhado da cache, como discutido na Seção 3.1.

(b) Independentemente do tamanho das mensagens enviadas a estratégia NCM apresenta tempo de execução maior que as demais estratégias.

• Para tamanho de vetores de computação,  $\mu(v) \ge 6MB$ :

(a) Se o tamanho da mensagem é menor do que 4MB (gráficos 3.7(a) e 3.7(b)), a estratégia NCM apresenta melhor desempenho do que CC e CMP

(b) Para mensagens maiores ou iguais a 4MB (gráficos 3.7(c) e 3.7(d)) o custo do envio de mensagem pela rede faz com que as estratégias CC e CMP apresentem tempo total de execução menor do que o NCM.



**Figura 3.7:** Análise do tempo total de execução de 2 tarefas executando as fases de computação e comunicação (modelo *all-to-all*).

Os resultados obtidos neste capítulo mostram que a hierarquia de memória compartilhada nas máquinas *multicore* pode piorar o desempenho da aplicação quando o tamanho dos dados alocados pelas tarefas que compartilham a mesma memória cache ultrapassam o tamanho desta memória. Entretanto, vale ressaltar que, na aplicação sintética, durante a fase de computação, faz-se acesso à toda memória alocada, ou seja, todas as linhas de cache associadas à estrutura alocada pela tarefa são requisitadas durante a execução da aplicação. Esta implementação foi utilizada com o objetivo de avaliar o uso intensivo da memória pelas tarefas.

# Capítulo 4

# Clusters de Máquinas Multicore: representação e modelos

Diante dos resultados obtidos no capítulo anterior é proposta uma representação para um *Clusters de Máquinas Multicore* (RCM), onde, define-se a hierarquia de memória, bem como o compartilhamento de recursos e elementos de processamento de um *cluster de multicore*. É proposto também um modelo para alocação de tarefas de uma aplicação que será executada em um *cluster de multicore*, representado por um RCM.

Neste capítulo é também apresentado uma extensão do modelo de aplicação GAD de modo que seja considerado no GAD a quantidade de memória requerida para executar as tarefas da aplicação, uma vez que esta informação pode ser utilizada para melhorar o desempenho da aplicação.

## 4.1 Representação de *Clusters* de Máquinas *Multicore* (RCM)

Um cluster de máquinas multicore é representado pelo conjunto CM, onde  $CM = \{M_0, M_1, M_2, \ldots, M_m\}$  é um conjunto de m máquinas, onde cada máquina  $M_i, 0 \le i \le m-1$  consiste de um conjunto  $P_i$  de p processadores, onde  $P_i = \{P_{(i,0)}, P_{(i,1)}, P_{(i,2)}, \ldots, P_{(i,p)}\}$ . Cada processador  $P_{(i,j)}$  consiste de um conjunto de c cores, cada um denotado por  $C_{(i,j,k)}$ .

Cores em uma mesma máquina  $M_i$  compartilham a memória global,  $gm_i$ , com capacidade  $gmc_i$ , e cores no processador  $P_{(i,j)}$  compartilham memória cache em um dado nível. Cada processador  $P_{(i,j)}$  em cada máquina  $M_i$  possui um conjunto de l memórias cache  $MC_i = \{mc_{(i,j,0)}, mc_{(i,j,1)}, \ldots, mc_{(i,j,l)}\}$ . A capacidade de cada cache  $mc_{(i,j,k)}$  é dada por  $cmc_{(i,j,k)}$ , tal que  $cmc_{(i,j,k)} < gmc_i$ , isto é, a capacidade da cache é menor que a capacidade da memória global.

A respeito do compartilhamento de cache, define-se que *cores* que compartilham a mesma memória cache são chamados de *cores vizinhos*. Além disto, todos os *cores* de uma mesma máquina compartilham uma memória global,  $gm_i$ .

Uma aresta (u, v) representa a troca de informação (ou dados) entre as tarefas  $u \in v$ , a quantidade é dados trocados é representada por  $\omega(u, v)$ . Considerando os testes referentes à fase de comunicação apresentados na Tabela 3.1 e 3.2 do capítulo anterior, na RCM, o custo de comunicação dentro de uma máquina é desprezível.

## 4.2 Modelo da Aplicação

A aplicação é modelada por um grafo  $G = (V, E, \epsilon, \omega, \mu)$ , onde o conjunto de vértices, V, representa as tarefas; E, a relação de precedência entre as tarefas;  $\epsilon_v$ , o peso de computação da tarefa v;  $\omega(u, v)$ , é o peso associado à aresta (u, v) que representa o a quantidade de dados transmitidos da tarefa u para v; e  $\mu_v$  é a quantidade de memória usada pela tarefa  $v \in V$ . O modelo de aplicação aqui proposto é o mesmo utilizado por vários autores na literatura [31, 33, 20, 21, 48, 71, 76, 74, 75, 10]. Entretanto, foi adicionado o parâmetro  $\mu_v$ . Este parâmetro deve auxiliar na decisão da alocação da tarefa da aplicação.

## 4.3 Modelo de Escalonamento de Tarefas para arquiteturas representadas por RCM

A identificação dos principais custos associados ao compartilhamento da hierarquia de memória obtida nos testes realizados com a aplicação sintética motivou a proposta de um modelo de escalonamento para *clusters* de máquinas *multicore*. Denominado *Modelo de Escalonamento de Cluster de Multicore* MECM, este modelo sugere uma alocação de tarefas baseada no tamanho dos dados alocados pelas tarefas da aplicação e o tamanho das mensagens trocadas entre elas. O MECM é baseado no RCM.

Suponha que u está alocada no core  $C_{(i0,j0,k0)}$ . A tarefa v é alocada no core  $C_{(i1,j1,k1)}$ , que é relacionado com  $C_{(i0,j0,k0)}$  dependendo das seguintes condições:

1. se  $(\mu_u + \mu_v) < cmc_{(i1,j1,k1)}$ , o tempo de computação de v, não havendo comunicação, é bem parecido quando i0 = i1 ou  $i0 \neq i1$ , isto é, se a quantidade de dados requerida pelas tarefas  $u \in v$  é menor que a capacidade da cache, o tempo de computação será menor se ambas tarefas forem alocadas em *cores* que não são vizinhos, em uma mesma máquina ou não. Entretanto, havendo comunicação, ou seja, se  $(u, v) \in E$ , o tempo total de execução de v será menor se ambas tarefas forem alocadas, necessariamente, na mesma máquina, ou seja, i0 = i1.

- 2. se  $(\mu_u + \mu_v) \ge cmc_{(i1,j1,k1)}$ , se as duas threads não se comunicam, o tempo de computação de v é menor se  $i0 \ne i1$ , isto é, se a quantidade de dados requerida por  $u \in v$  é maior que a capacidade da cache, o tempo de computação de v será menor se ambas tarefas forem executadas em *cores* pertencentes a máquinas diferentes. Entretanto, se as tarefas se comunicam, deve-se considerar  $\omega(u, v)$ , ou seja, a quantidade de dados transmitidos entre estas tarefas: para os experimentos avaliados, as tarefas poderiam ser alocadas à máquinas distintas de acordo com uma relação dada entre a latência associada com a transmissão de  $\omega(u, v)$  unidade de dados e a contenção de acesso à memória. Nos experimentos realizados neste trabalho:
  - (a) alocar as tarefas em máquinas diferentes produz tempo de execução menor do que na mesma máquina para mensagens menores do que um tamanho  $\alpha$ , ou seja,  $\omega(u, v) < \alpha$ .
  - (b) por outro lado, para mensagens maiores que α, o tempo de execução é menor quando as tarefas são alocadas na mesma máquina, pois o custo de enviar estas mensagens via rede é maior do que mantê-las localmente.

As condições 2.a e 2.b foram obtidas a partir das fases de comunicação avaliadas utilizando os modelos *ping-pong* e *all-to-all*. Para o *cluster* de *multicore* utilizado nos testes, onde a memória cache L1 é privada e a memória cache L2 é compartilhada por dois *cores* de um mesmo processador o valor de  $\alpha = 4MB$ .

# Capítulo 5

# Balanceamento de Carga para um Branch-and-Bound paralelo

De acordo com as características coletadas do ambiente de execução, foi proposta uma representação dos *clusters* de *multicore* (RCM) e um modelo de escalonamento de tarefas para estes ambientes (MECM). Com o objetivo de avaliar o modelo proposto, foi implementada uma aplicação paralela real cujos procedimentos de balanceamento de carga foram desenvolvidos de acordo com as condições do MECM. Mais precisamente, as contenções de acesso à memória foram evitadas avaliando-se dinamicamente a quantidade de memória necessária para executar a carga atribuída a cada tarefa, e consequentemente, a Condição 1. do modelo foi satisfeita. Os resultados obtidos mostram que a aplicação executou eficientemente como é apresentado no decorrer deste capítulo.

A aplicação implementada resolve o Problema de Particionamento de Conjuntos (Set Patitioning Problem - *SPP*) utilizando a técnica *branch-and-bound*, *B&B*. O *SPP* é um problema de otimização combinatória que pode ser usado para modelar importantes problemas da vida real como, por exemplo, escalonamento de tripulação em linhas aéreas e roteamento de veículos, entre outros [18].

O branch-and-bound é uma técnica exata muito utilizada para resolver problemas de otimização. O B&B efetua sucessivas partições do espaço de busca das soluções induzindo uma árvore enumerativa. Cortes na árvore podem ocorrer ao longo da enumeração de acordo com os limites superior e inferior da solução ótima calculados. Estes cortes são importantes, pois evitam o crescimento da árvore e portanto, diminuem o tempo de execução da aplicação. Uma outra característica interessante do B&B é que cada nó da árvore pode ser processado quase independentemente, e, consequentemente, o algoritmo é bem adaptável ao paralelismo. Existe uma variedade de trabalhos na literatura que propõe algoritmos branch-andbound e frameworks que são capazes de facilitar o desenvolvimento do B&B para executar em sistemas de arquitetura de memória distribuída [15, 27, 32, 40] ou compartilhada [32, 35, 65, 67, 70, 72, 77]. B&B adaptados para executar em arquiteturas de memória distribuída e compartilhada ainda são um grande desafio. Em [19] é apresentada uma biblioteca que permite a execução paralela e distribuída de algoritmos sequenciais de otimização usando o B&B em clusters de multicore. Entretanto, nenhuma informação a respeito da hierarquia de memória dos processadores multicore é utilizada durante a execução da aplicação, sendo esta aplicação baseada nos modelos Map-Reduce e Hadoop.

Neste trabalho, o B&B paralelo e distribuído foi implementado considerando em seus procedimentos a representação RCM e o modelo MECM. Este capítulo apresenta como o SPP foi resolvido através da técnica B&B e os algoritmos utilizados no *framework* de balanceamento de carga que foram aplicados em sua paralelização.

## 5.1 B&B Sequencial aplicado ao Problema de Particionamento de Conjuntos

O Problema de Particionamento de Conjuntos é um problema de grande interesse porque é modelo fundamental dentro da otimização combinatória e consiste, de em sua forma geral, em: dado um conjunto S, um conjunto de restrições que define um conjunto F de subconjuntos de S e um custo associado a cada subconjunto pertencente a este conjunto, deseja-se escolher subconjuntos de F que formem uma partição de S com custo mínimo.

O *SPP* representa uma grande parcela dos problemas de escalonamento e planejamento atuais, entre eles, o problema de escalonamento de tripulação em aviões e de frota de veículos, localização de facilidades, problemas de corte e empacotamento, escalonamento de tripulação de vôo, roteamento de veículos, problemas de Steiner em grafos [12, 18, 28].

A Figura 5.1 apresenta uma instância para o problema SPP, onde,  $S = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$  é o conjunto de n = 8 itens e o conjunto de subconjuntos de S é formado por  $S_1$ ,  $S_2$ ,  $S_3$ ,  $S_4$  e  $S_5$ . O custo  $c_i$  de cada item é apresentado no vetor de custos C. Para esta instância, os subconjuntos  $S_1$  e  $S_2$  constituem uma partição de S, e portanto, formam uma solução viável. Nenhuma outra combinação dos subconjuntos é viável, já que não existe dois ou mais subconjuntos cuja união seja igual a S [28].

Formalmente, o *SPP* pode ser modelado como segue. Sejam os itens do conjunto representados por n variáveis,  $x_1, \ldots, x_n$ , com custo correspondente a  $c_1, \ldots, c_n$  e coe-



Figura 5.1: Exemplo de instância do SPP.

ficientes 0-1  $a_{1j}, \ldots, a_{nj}$ , para  $j = 1, \ldots, m$ , onde m é o número de subconjuntos. O Problema de Particionamento de Conjuntos é o problema de alocar valores 0-1 para estas variáveis tal que  $\sum_{i=1}^{n} a_{ij}x_i = 1$ , para  $j = 1, \ldots, m$ , minimize  $\sum_{i=1}^{n} c_i x_i$ .

Para resolver o SPP o método de busca em árvore B&B define uma enumeração implícita do espaço de busca de soluções. O método B&B é baseado no conceito de dividir para conquistar. A idéia principal é dividir o problema em subproblemas mais fáceis de serem resolvidos. Se o subproblema é ainda difícil de ser resolvido este subproblema pode ser particionado em outros subproblemas e assim, recursivamente. Quando se consegue resolver o subproblema, ele não precisa ser mais dividido e se torna um nó folha da árvore [28].

O B&B evita a enumeração completa da árvore descartando subconjunto de soluções, ou subproblemas, que não possuem a solução ótima. Este processo é chamado de poda e é realizado através da análise dos limites inferiores e superiores (ou duais) para a solução ótima. Estes limites são obtidos através das soluções viáveis encontradas e através da relaxação ou dualidade do problema. Neste trabalho, foi utilizada a heurística apresentada em [78] que é uma heurística simples, mas que, para as instâncias testadas, alcança resultados comparáveis aos apresentados em [18], que propõe heurísticas mais sofisticada para encontrar os limites. Esta heurística e a operação de *branching* são apresentadas nas próximas seções, e em seguida é apresentado os resultados obtidos. As demais seções deste capítulo apresentam a versão paralela e distribuída do B&B proposta neste trabalho.

### 5.1.1 Lower Bound

O limite inferior da solução ótima utilizado na poda da árvore para este problema pode ser calculado resolvendo a relaxação contínua:

$$Minimizar \quad \sum_{i=1}^{n} c_i x_i \tag{5.1}$$

Sujeito a 
$$\sum_{i=1}^{n} a_{ij} x_i = 1$$
  $j = 1, ..., m$  (5.2)

$$x_i \ge 0, \qquad i = 1, \dots, n \tag{5.3}$$

ou seu dual:

Maximizar 
$$\sum_{j=1}^{m} \pi_j$$
 (5.4)

Sujeito a 
$$\sum_{j=1}^{m} a_{ij} \pi_j \le c_i \quad i = 1, \dots, n.$$
 (5.5)

onde a variável  $\pi_j$  pode assumir valores positivos ou negativos.

No procedimento *branch-and-bound* apresentado neste trabalho, a heurística dual utilizada para calcular uma solução dual viável que busca a solução ótima em um tempo computacional reduzido, repete dois passos principais por um número fixo de iterações: i) o primeiro passo, chamado de *forward step*, consiste em aumentar o valor de  $\pi_j$  tanto quanto possível; ii) no segundo passo, *backward step*, algum valor de  $\pi_j$  é reduzido enquanto aumenta outros, com o intuito de melhorar o limite inferior no próximo passo, *forward step*. O *backward step* não é executado na última iteração.

O forward step também é dividido entre as iterações. Em cada iteração, o mesmo valor  $\Delta_1$  é adicionado ao  $\pi_j$  que ainda não teve sua restrição saturada, isto é,  $\Delta_1$  é adicionado ao  $\pi_j$  se, e somente se, a  $\sum_{j=1}^m a_{ij}\pi_j < c_i$  para todo *i* tal que  $a_{ij} = 1$ .  $\Delta_1$  é escolhido como valor máximo que será mantido até que todas as restrições (5.5) sejam satisfeitas. No mínimo uma nova restrição se torna saturada em cada iteração. O forward step pára quando nenhuma variável  $\pi_j$  pode ser incrementada. Este passo é bem conhecido e é uma parte de um conhecido algoritmo aproximativo para SPP [47].

No backward step, o valor de  $\pi_j$  é decrescido de  $\Delta_2(\alpha_j - 1)$ , para algum  $\Delta_2$ , onde  $\alpha_j$ é o número de restrições saturadas e  $\pi_j$  tem um coeficiente diferente de zero. Se  $\alpha_j = 0$ , então  $\pi_j$  é incrementado de  $\Delta_2$ . O valor de  $\Delta_2$  é escolhido tal que o limite inferior é multiplicado por um fator  $\theta$ . Neste trabalho,  $\theta = 0.5$  na primeira iteração do nó raiz e  $\theta = 0.3$  na primeira iteração dos demais nós. Após cada iteração,  $\theta$  é multiplicado por 0.7. Foram realizadas 10 iterações no nó raiz e 5 nos demais nós. Estes valores foram obtidos através de experimentos preliminares.

### 5.1.2 Branching

O branching ou a ramificação de um nó da árvore é realizada na restrição (5.2). Para um linha j selecionada é criada uma ramificação para cada i com  $a_{ij} = 1$  onde a variável  $x_i$  é fixada.

Um característica importante do SPP é que cada nó filho pode ser menor que seu nó pai. Sempre que uma variável  $x_i$  é fixada, cada variável  $x_k$  tal que  $a_{ij} = 1$  e  $a_{kj} = 1$  algum j pode ser fixado em zero. Então, cada restrição (5.2) onde  $x_i$  tem coeficiente diferente de zero pode ser removido. Neste método, as restrições restantes herdam  $\pi_j$  do nó pai.

O critério utilizado para escolher uma restrição j para branching é a seguinte: seja  $\delta_i$ o número de restrições  $\ell$  tal que  $a_{i\ell} = 0$ . O valor j com menor valor de  $\sum_{\substack{i \in \{1,...,n\}\\a_{ij}=1}} \delta_i$ , que representa o número total de restrições em todos os nós filhos que poderiam ser criados.

Para encontrar uma solução viável mais rapidamente, o nós filhos foram processados em ordem não crescente de  $(c_i - \sum_{j=1}^n a_{ij}\pi_j)/\delta_i$ . A árvore branch-and-bound é percorrida em profundidade. Na próxima seção são apresentados os resultados obtidos pelo B&Bsequencial aplicado ao problema de particionamento de conjuntos, o  $SBB_{SPP}$ .

### 5.1.3 Experimentos computacionais do B&B sequencial

Para realizar os testes do B&B sequencial, os valores das instâncias para o problema de particionamento de conjuntos foram geradas aleatoriamente. As instâncias foram identificadas iniciando pela letra I seguida de números referente ao modo como as mesmas foram geradas. Os dois primeiros números são, respectivamente, a quantidade de conjuntos e a quantidade de itens presentes na instância. O próximo número é a probabilidade do item aparecer nos conjuntos.

Os resultados obtidos pelo algoritmo branch-and-bound adotado neste trabalho são apresentados na Tabela 5.1 (a instância I200-650-0.02 foi gerada utilizando duas sementes de aleatoriedade diferentes, 110 e 152). Nesta tabela as colunas "*Tempo*" e "# de Nós" apresentam o tempo total de execução em segundos e o número de nós resolvidos para cada instância, quando é utilizado o  $SBB_{SPP}$  e o resolvedor Cplex, versão 12.2. A Coluna "% Redução" é a porcentagem de redução do tempo de execução do  $SBB_{SPP}$  em relação ao resolvedor Cplex. Finalmente a coluna "Sol. Ótima" apresenta a solução ótima obtida para a instância.

Os experimentos foram executados em um *core* de uma máquina Intel Xeon 5355 Clovertown *quad-core*. Cada *core* possui uma cache L1 privada (64 KB) e compartilham uma cache L2 de 8MB com outro *core* do mesmo processador. Todos os *cores* de um mesmo nó possuem acesso uniforme à memória RAM de 16GB. O sistema operacional é CentOS 5.3 com kernel versão 2.6.18.

Apesar da heurística utilizada para poda ser simples, o  $SBB_{SPP}$  reduziu os tempo de execução para a maioria das instâncias quando comparado com os tempos obtidos pelo resolvedor.

Instâncies	SBB	$SBB_{SPP}$		Cplex		Solução
Instancias	Tempo (s)	# de Nós	Tempo(s)	# de Nós		Sol. Ótima
I90-400-0.03	12,45	47.845	8,05	4.976	-35,36	26
I90-400-0.04	22,83	113.148	44,26	31.438	48,41	26
I90-400-0.05	52,27	294.629	446,87	242.488	88,30	27
I100-500-0.03	29,07	99.133	$14,\!82$	8.103	-49,04	27
I100-500-0.04	301,82	1.297.756	282,56	156.464	-6,38	25
I100-500-0.05	$423,\!66$	2.174.831	2.843,24	1.264.436	85,10	28
I110-750-0.03	11.314,61	26.806.647	$2.034,\!64$	1.048.024	-82,02	27
I110-750-0.04	30.480,57	87.862.510	20.892,74	8.887.116	-31,46	25
I110-750-0.05	23.139,53	99.519.888	28.9174,54	87.136.670	92,00	27
I200-650-0.02-100	9.088, 83	12.990.841	41.225,17	7.078.290	77,95	65
I200-650-0.02-152	15.990, 18	24.348.520	71.757,59	13.538.888	77,72	68
I120-600-0.04	$34.654,\!40$	133.972.589	132.643, 13	51.653.943	73,87	33

**Tabela 5.1:** Comparação entre o B&B sequencial  $(SBB_{SPP})$  e o resolvedor *Cplex* 

# 5.2 Branch-and-Bound paralelo aplicado ao Problema de Particionamento de Conjunto - $PBB_{SPP}$

O algoritmo branch-and-bound paralelo e distribuído proposto foi implementado baseado no algoritmo sequencial apresentado na Seção 5.1. Algumas características do  $SBB_{SPP}$ influenciam nas decisões relacionadas à divisão de tarefas entre as máquinas do *cluster*. Normalmente, a árvore gerada pelo branch-and-bound é binária, o que define duas subárvores geradas a partir de cada nó. Entretanto, a operação de branching realizada pelo  $SBB_{SPP}$  permite que várias sub-árvores sejam geradas a partir da execução de um nó. A quantidade de sub-árvores geradas não é previamente conhecida e pode variar muito.

Além disto, apesar do tempo de execução dos nós da árvore ser muito pequeno, em média 0,001 a 0,006 segundos, dependendo da instância, esses nós podem precisar de uma grande quantidade de memória para ser armazenado (para as instâncias grandes). A quantidade de memória necessária para armazenar as informações do nó da árvore é chamada, neste trabalho, de tamanho do nó.

Para analisar melhor estas características do  $SBB_{SPP}$ , a Tabela 5.2 contém informações sobre o tamanho dos nós em bytes e o seu tempo de execução correspondente. Para as quatro instâncias são apresentados os cinco menores (cinco primeiras linhas da tabela) e os cinco maiores (cinco últimas linhas da tabela) nós da árvore B&B gerada. Na tabela também é apresentado o nível associado a estes nós na árvore. O nível de um nó define a distância do nó até o nó raiz da árvore B&B.

Pela tabela pode-se observar que os tempos de execução dos nós são muito pequenos. Além disto, os nós dos menores níveis demandam mais memória do que os de níveis mais altos, uma vez que a quantidade de restrições saturadas é menor nos níveis mais baixos. Entretanto, o tempo de execução dos nós não corresponde ao tamanho dos nós nem corresponde ao seu nível. As informações a respeito do tamanho dos nós não são conhecidas *a priori*, dificultando decisões relacionadas à divisão de nós da árvore entre as máquinas do *cluster*.

Nível	Tamanho Tempo de		Nível	Tamanho	Tempo de
	do Nó Execução			do Nó	Execução
	(bytes) (segundo			(bytes)	(segundos)
	I90-400-0.03			I100-500-0.03	, <u> </u>
17	448	0,001	25	420	0,001
17	452	0,001	22	424	0,105
18	456	0,085	22	448	0,001
18	480	0,001	26	472	0,001
16	516	0,001	27	480	0,005
2	9092	0,005	1	9836	0,096
3	9208	0,005	2	10068	0,006
1	9216	0,076	1	10452	0,001
2	9344	0,005	2	10888	0,006
1	9436	0,062	1	11780	0,083
I110-750-0.03			I200-650-0.02-100		
24	508	0,001	25	2608	0,001
24	516	0,012	19	2640	0,001
29	528	0,001	23	2724	0,000
29	532	0,034	22	2728	0,001
24	532	0,026	23	2828	0,002
2	17072	0,009	1	16960	0,003
1	17476	0,006	1	17080	0,009
1	17628	0,011	2	17400	0,011
1	18492	0,003	1	18616	0,003
1	18608	0,019	1	19004	0,008

**Tabela 5.2:** Exemplo dos níveis, tamanho e tempo de execução dos nós para as instâncias I90-400-0.03, I100-500-0.03, I110-750-0.03 e I200-650-0.02-100 do SPP.

A Figura 5.2 apresenta o tempo de execução no eixo x e o tamanho do nó para a instância I90-400-0.03 no eixo y. Pela figura, pode-se perceber que a maioria dos nós

possui um tempo de computação muito pequeno com tamanho de 50 Bytes a 9.6 KBytes. Uma vez que todas as instâncias analisadas neste trabalho apresentam características similares, o gráfico é apresentado apenas para uma instância.



Figura 5.2: Tempo de execução e tamanho dos nós para a instância I90-400-003.

Na próxima seção é apresentado o *framework* de balanceamento de carga que considera o modelo de escalonamento proposto.

### 5.2.1 Framework de Balanceamento de Carga

No algoritmo  $PBB_{SPP}$  é criado um processo em cada máquina do *cluster*. Cada processo é composto por tantas *threads* quanto o número de *cores* disponíveis na máquina o qual foi alocado. Esta estratégia foi utilizada para evitar o compartilhamento de recursos de processamento que pode afetar diretamente o desempenho da aplicação. Assim, a atribuição das *threads* às unidades de processamento é estática. Entretanto, durante a execução da aplicação, procedimentos do *framework* permitem que as *threads* transfiram carga entre elas, alodas a uma mesma máquina ou não, dinamicamente quando uma tarefa se torna ociosa. Neste trabalho, o termo carga refere-se a um conjunto de nós da árvore a serem resolvidos e o termo tarefa é referente a uma *thread*.

O procedimento de criação das tarefas segue os seguintes passos: inicialmente, o processo pertencente à máquina  $M_0$  cria uma thread líder da aplicação, LT, para executar nesta máquina. LT é responsável por iniciar e terminar a aplicação. Logo, cada processo de cada máquina  $M_i$  do cluster, cria uma thread gerenciadora para esta máquina denominada  $MT_i$ .  $MT_i$  então cria as demais threads, denominadas threads trabalhadoras, que executarão em  $M_i$ . Cada uma destas threads é associada a um core  $C_{(i,j,k)}$  da máquina  $M_i$ .  $MT_i$  é responsável pela comunicação entre as threads gerenciadoras das máquinas do cluster enquanto as threads trabalhadoras são responsáveis pela execução dos nós da árvore B&B que lhe são designados pela thread gerenciadora.

Ao receber a sua carga, formada por um conjunto de nós da árvore, a *thread* trabalhadora  $T_{(i,j,k)}$  armazena-os em um lista local de nós denominada  $TL_{(i,j,k)}$ . A execução de um nó consiste em retirar um nó da lista e computá-lo utilizando os procedimentos do *branch-and-bound* descrito na Seção 5.1. Os nós (ou sub-árvore) gerados a partir da computação deste nó são também armazenados em  $TL_{(i,j,k)}$ . O tamanho da lista deve ser gerenciado de modo que a *thread* evite requisitar mais memória do que a capacidade da *cache* associada ao *core* onde está executando, neste trabalho da cache L2. A gerência do tamanho da lista garante que a Condição 1. do *Modelo de Balanceamento de Carga* apresentado na Seção 4.3 seja satisfeita, como será explicado adiante. Os nós da árvore *B&B* paralelo, assim como no *B&B* sequencial, são percorridos em profundidade.

Ao terminar a execução de seus nós uma *thread* trabalhadora,  $T_{(i,j,k)}$ , inicia um procedimento para conseguir mais nós com as demais *threads* trabalhadoras.

A thread gerenciadora  $MT_i$  é responsável por requerer carga das outras máquinas do cluster quando suas threads trabalhadoras estão sem carga, ou seja, ociosas, e também por responder a requisições de carga. Seja  $M'_i$  uma máquina cujas as listas das threads trabalhadoras não estão vazias. Ao receber uma requisição de carga de  $MT_i$ ,  $MT'_i$  remove parte dos nós da lista de todas as threads alocadas em sua máquina e envia-os a thread requisitante  $MT_i$ , que distribui a carga recebida entre suas threads. Caso contrário, se  $MT_i$  não é capaz de obter carga com nenhuma outra máquina do cluster e se todas as suas threads estão ociosas,  $MT_i$  inicia a condição de terminação e informa à líder da aplicação LT. O  $PBB_{SPP}$  termina quando LT recebe a condição de terminação local de todas as threads gerenciadoras pertencentes à aplicação.

A Figura 5.3 ilustra os procedimentos executados pelas threads do  $PBB_{SPP}$ . Nesta figura o cluster é formado por duas máquinas  $M_0 \in M_1$ . A máquina  $M_0$  possui o processador  $P_{(0,0)}$  e a máquina  $M_1$  o  $P_{(1,0)}$ . Cada processador possui dois cores que compartilham uma única cache. A thread  $LT_0$ , alocada ao core  $C_{(0,0,0)}$ , inicia a aplicação resolvendo o nó raiz da árvore. Logo, executa o procedimento de distribuição inicial. As threads gerenciadoras, alocadas aos cores  $C_{(0,0,0)}$  e  $C_{(1,0,0)}$ , são responsáveis por receber a carga e distribuí-la entre as threads trabalhadoras pertencentes à sua máquina. Finalmente, as threads trabalhadoras, após receberem suas cargas, são responsáveis pelos procedimentos relacionados a execução do B&B e pela execução do procedimento de balanceamento de carga local, enquanto as *threads*  $MT_0$  e  $MT_1$  são responsáveis pelo procedimento de balanceamento de carga entre as máquinas e identificar o término da aplicação.

A próxima seção apresenta em detalhes os algoritmos dos procedimentos executados pelo  $PBB_{SPP}$ .



**Figura 5.3:** Ilustração do *Framework* de Balanceamento de Carga em um *cluster* de máquinas *multicore*.

As próximas seções destinam-se a explicação detalhada das operações de balanceamento de carga realizadas durante a execução do  $PBB_{SPP}$ .

# 5.2.2 Algoritmos do Framework de Balanceamento de Carga do $PBB_{SPP}$

O primeiro procedimento executado pelo  $PBB_{SPP}$  após a criação das *threads* é o de distribuição inicial de carga entre as *threads* trabalhadoras. Este procedimento é iniciado pela *thread* LT, que executa o nó raiz da árvore B&B e armazena em uma lista GL os nós gerados a partir da execução deste nó, como apresentado no Algoritmo 3, line 1.

A quantidade de nós na lista GT, que é retornada no Algoritmo 3 através da execução da função numeroDeNos(GL), é igualmente dividida entre as threads trabalhadoras através das threads gerenciadoras. Vale lembrar que a quantidade de threads trabalhadoras é referente a todas as threads de todas as máquinas do cluster, logo, na linha 2, m é igual ao número de máquinas do cluster, p é igual a quantidade de processadores por máquina e cé igual ao número de cores por processador. Os nós são enviados para as threads através de uma mensagem do tipo Load (linha 7). Uma thread trabalhadora  $T_{(i,j,k)}$  pode não receber carga inicial, ou seja,  $Load = \emptyset$  se numero $DeNos(GL) < m^*p^*c$ . Neste caso,  $T_{(i,j,k)}$  inicia o procedimento de requerimento de carga executando o Algoritmo 4. Este algoritmo também é realizado quando  $T_{(i,j,k)}$ se torna ociosa, ou seja, quando  $T_{(i,j,k)}$  termina de executar todos os nós que recebeu na mensagem *Load*. A mensagem *Load* é criada retirando-se numNos da lista *GL* através da função nos(*GL*, numNos) da linha 6 do algoritmo.

Algorithm 3 Distribuição Inicial executada pela <i>thread</i> líder da aplicação, <i>LT</i> .
1: $GL = \text{Resolve}(\text{NoRoot});$
2: numNos $\leftarrow \frac{numeroDeNos(GL)}{(m \star n \star c)};$
3: for all $i \leftarrow 0$ to $m \operatorname{do}^{(maple)}$
4: for all $j \leftarrow 0$ to $p$ do
5: for all $k \leftarrow 0$ to $c$ do
6: $Load \leftarrow nos(GL, numNos);$
7: Envia Load para $T_{(i,j,k)}$ ;
8: end for
9: end for
10: end for

Quando um *thread* se torna ociosa, ou seja, sem carga, inicia o procedimento de balanceamento de carga. Um procedimento de balanceamento de carga, basicamente, consiste na transferência de carga entre as tarefas da aplicação e envolve decisões como: i) quantos e quais processadores estarão envolvidos na transferência; ii) se a transferência será provocada por uma tarefa que está sobrecarregada e envia carga para uma tarefa ociosa (*sender initiated* ou também chamada, *work-distribution*) ou se será ocasionada por uma tarefa que se torna ociosa e envia requisições de carga as tarefas vizinhas (*receiver initiated* ou *work-stealing*).; iii) a quantidade de carga que será transferida; iv) se o método será centralizado ou se transferência será distribuída, ou seja, qualquer tarefa pode provocar a transferência de carga sem a intervenção de um gerenciador [16, 34].

O procedimento de balanceamento de carga proposto neste trabalho é worker steling e a divisão de carga, como descrito anteriormente, não pode ser baseada nos parâmetros relativos ao tamanho da carga ou seu tempo de processamento, já que o tamanho ou o nível do nó na árvore do B&B não tem relação com o seu tempo de processamento. O procedimento de balanceamento é hierárquico, sendo considerado um algoritmo distribuído dentro de uma mesma máquina, já que todas as *threads* podem enviar pedidos de carga para qualquer outra *thread* trabalhadora e, somente quando estas *threads* não conseguem carga umas com as outras um pedido de carga é enviado à *thread* gerenciadora, que envia requisições de carga para outras máquinas. Sendo assim, o balanceamento de carga entre as máquinas é centralizado e realizado pela *thread* gerenciadora e será detalhado abaixo.

### 5.2 Branch-and-Bound paralelo aplicado ao Problema de Particionamento de Conjunto - $PBB_{SPP}$ 45

O procedimento de balanceamento de carga tem início quando uma thread,  $T_{(i,j,k)}$ , termina a execução dos nós de sua lista  $TL_{(i,j,k)}$ . Inicialmente  $T_{(i,j,k)}$  tenta obter carga com a thread que executa no core vizinho ao core onde está alocada. Este procedimento é realizado enviando uma mensagem do tipo LoadRequest e é ilustrado no Algoritmo 4.  $T_{(i,j,k)}$ , fica em estado de espera até receber uma mensagem da thread vizinha. Se a resposta é uma mensagem do tipo Load,  $T_{(i,j,k)}$ , copia o conjunto de nós disponibilizados por sua thread vizinha em uma lista compartilhada para sua lista local e inicia a execução dos nós recebidos. Este procedimento é ilustrado na Figura 5.4. Caso contrário, se sua vizinha está ociosa, ela responde ao pedido de carga com uma mensagem do tipo NoLoad, e a execução do Algoritmo 5 é iniciado. Vale ressaltar que, quando a requisição de carga entre threads de uma mesma máquina é realizada através de semáforos, logo, não é necessário o envio de mensagem.



Figura 5.4: Exemplo de requisição de carga à thread vizinha.

**Algorithm 4**  $T_{(i,j,k)}$  envia pedido de carga para a sua *thread* vizinha.

```
1: if TL_{(i,j,k)} = \emptyset then
```

- 2: Envia LoadRequest para  $T_{(i,j,l)}$ ;
- 3: end if

Se  $T_{(i,j,k)}$  receber uma mensagem do tipo NoLoad da sua thread vizinha,  $T_{(i,j,k)}$ , envia, sequencialmente, mensagens do tipo LoadRequest para as threads trabalhadoras que foram alocadas ao mesmo processador que ela, logo depois, para as threads que foram alocadas a processadores diferentes, como ilustrado na Figura 5.5. Os envios destas requisições são apresentados nas linhas 1-10 e linhas 11-23, respectivamente, do Algoritmo 5. As variáveis y e j definem os índices das threads as quais serão enviadas as mensagens. Se uma mensagem do tipo *Load* é recebida como resposta o procedimento de balanceamento de carga termina devido aos testes realizados nas linhas 4, 13, 15 e 25. Por outro lado, se  $T_{(i,j,k)}$  não conseguir carga com as *threads* trabalhadoras alocadas na máquina  $M_i$  onde está executando,  $T_{(i,j,k)}$  envia uma mensagem *LoadRequest* para a gerente da máquina  $MT_i$  (linha 26), Figura 5.6.



Figura 5.5: Exemplo de requisição de carga às threads da mesma máquina.

**Algorithm 5** Quando  $T_{(i,j,k)}$  recebe uma mensagem do tipo NoLoad de sua thread vizinha,  $T_{(i,j,l)}$ .

1: MSG = NoLoad2:  $y \leftarrow 0$ ; 3: /\* threads alocadas a cores não vizinhos do mesmo processador \*/ 4: while ((y < c - 1) and (MSG = NoLoad)) do if  $(y \neq k)$  then 5:Envia LoadRequest para  $T_{(i,j,y)}$ ; 6: Recebe MSG de  $T_{(i,j,y)}$ ; 7: end if 8: 9:  $y \leftarrow y + 1;$ 10: end while 11: /\* threads alocadas a cores não vizinhos de processadores diferentes \*/ 12:  $j \leftarrow j + 1;$ 13: while ((j < p-1) and (MSG = NoLoad)) do 14: $y \leftarrow 0;$ while ((y < c - 1) and (MSG = NoLoad)) do 15:if  $(y \neq k)$  then 16:Envia LoadRequest para  $T_{(i,i,y)}$ ; 17:Recebe MSG de  $T_{(i,j,y)}$ ; 18:end if 19:20:  $y \leftarrow y + 1;$ end while 21:  $j \leftarrow j + 1;$ 22:23: end while 24: /\* Avisa a  $MT_i$  que está sem carga \*/ 25: if (MSG = NoLoad) then **Envia** LoadRequest para  $MT_i$ ; 26:27: end if

Para acelerar o término do procedimento de balanceamento de carga, uma thread trabalhadora  $T_{(i,j,k)}$  deve responder imediatamente a uma mensagem do tipo LoadRequest. A mensagem com a resposta pode ser do tipo Load, onde, neste caso,  $T_{(i,j,k)}$  divide os nós de sua lista com a thread requisitante, como apresentado na linha 2 do Algoritmo 6. Entretanto, se  $TL_{(i,j,k)} = \emptyset$ ,  $T_{(i,j,k)}$  envia uma mensagem do tipo NoLoad informando que sua lista também está vazia, como apresentado na linha 6 do mesmo algoritmo.



Figura 5.6: Exemplo de requisição de carga à thread gerenciadora.

O Algoritmo 6 também é executado quando  $T_{(i,j,k)}$  recebe um pedido de carga de sua *thread* gerenciadora. Por isto, o envio das mensagens ilustradas neste algoritmo está na forma  $T_{(i,j,l)}/MT_i$ . O procedimento de balanceamento de carga executado com a participação das *threads* gerenciadores será explicado mais adiante.

<b>Algorithm 6</b> Quando $T_{(i,j,k)}$ recebe <i>LoadRequest</i> de $T_{(i,j,l)}$ ou de $MT_i$ .
1: if $(TL_{(i,j,k)} \neq \emptyset)$ then
2: numNos $\leftarrow \frac{numeroNs(TL_{(i,j,k)})}{2};$
3: $Load \leftarrow \operatorname{nos}(TL_{(i,j,k)}, \operatorname{numNos});$
4: Envia Load para $T_{(i,j,l)}/MT_i$ ;
5: else
6: Envia NoLoad para $T_{(i,j,l)}/MT_i$ ;
7: end if

Os algoritmos apresentados referem-se aos procedimentos de balanceamento de carga executados pelas threads trabalhadoras. Uma thread gerenciadora  $MT_i$ , começa a sua participação no balanceamento de carga quando recebe a primeira mensagem de LoadRequest de uma de suas threads trabalhadoras. Antes de apresentar os algoritmos, vale lembrar que, a carga executada pelas threads é criada dinamicamente, uma vez que a execução de um nó da árvore gera outros nós para serem executados. Diante disto, uma mensagem do tipo LoadRequest recebida pela gerenciadora  $MT_i$ , não indica que todas as threads alocadas à sua máquina estão sem carga, mas que  $T_{(i,j,k)}$  não conseguiu obter carga com estas threads. A thread poderia estar executando o último nó de sua lista quando recebeu a requisição de carga de  $T_{(i,j,k)}$  e por isso respondeu com uma mensagem de NoLoad. Para evitar que mensagens sejam enviadas via rede desnecessariamente, somente quando  $MT_i$  recebe pelo menos NT (explicado na Seção 5.2.3.3) requisições de carga de suas threads trabalhadoras é que  $MT_i$  envia uma requisição de carga, ou seja, envia uma mensagem do tipo GlobalLoadRequest, para uma outra máquina  $MT_{i'}$ , seguindo uma sequência de requisições, como apresentado no Algoritmo 7. Se  $MT_i$  não receber carga como resposta à sua requisição e receber uma mensagem do tipo NotLoad,  $MT_i$ inicia o procedimento de detecção de terminação, linha 12 do Algoritmo 7. O algoritmo de detecção de terminação foi considerado um procedimento dentro do framework de balanceamento de carga devido à geração de carga dinâmica do B&B.

Algorithm 7 Quando $MT_i$ recebe $LoadRequest$ de $T_{(i,j,k)}$ .
1: $MSG = LoadRequest$
2: $i' \leftarrow 0;$
3: if $(ReqLocalLoad = NT)$ then
4: while $((MSG = NoLoad) \text{ and } (i' < m - 1)) \text{ do}$
5: <b>if</b> $(i' \neq i)$ <b>then</b>
6: Envia $GlobalLoadRequest$ para $MT_{i'}$ ;
7: <b>Recebe</b> MSG de $MT_{i'}$ ;
8: end if
9: $i' \leftarrow i' + 1;$
10: end while
11: <b>if</b> $((MSG = NoLoad)$ <b>and</b> $(i' = m - 1))$ <b>then</b>
12: $DeteccaoTerminacao();$
13: end if
14: end if
15: ReqLocalLoad++;

Se  $MT_{i'}$  responder a requisição de  $MT_i$  com uma mensagem GlobalLoad,  $MT_i$  compartilha a carga recebida entre as *threads* trabalhadoras de sua máquina que estão sem carga, enviando mensagens do tipo *Load* para as *threads*, como apresentado na Figura 5.7. Este procedimento é ilustrado no Algoritmo 8. A quantidade de *threads* sem carga é armazenada na variável *ReqLocalLoad* do Algoritmo 7, linha 15.



**Figura 5.7:** A *thread* gerenciadora compartilha a carga recebida entre as *threads* trabalhadoras de sua máquina que estão sem carga.

<b>Algorithm 8</b> Quando $MT_i$ recebe GlobalLoad de $MT_{i'}$ .
1: $ML_i \leftarrow GlobalLoad;$
2: numNos $\leftarrow \frac{numeroNos(GlobalLoad)}{ReqLocalLoad};$
3: for all $(j \leftarrow 0 \text{ to } j < p)$ do
4: for all $(k \leftarrow 0 \text{ to } k < c)$ do
5: $Load \leftarrow nos(ML_i, numNodes);$
6: Envia Load para $T_{(i,j,k)}$ ;
7: end for
8: end for

Finalmente, quando uma thread gerenciadora  $MT_i$  recebe mensagem do tipo Global-Loadrequest de uma gerente  $MT_{i'}$ ,  $MT_i$  envia uma mensagem do tipo LoadRequest para todas as suas threads trabalhadoras, como ilustrado no Algoritmo 9 e na Figura 5.8.

<b>Algorithm 9</b> Quando $MT_i$ recebe <i>GlobalLoadRequest</i> from $MT_{i'}$ .				
1: numLoad $\leftarrow 0$ ;				
2: for all $(j \leftarrow 0 \text{ to } j < p)$ do				
3: for all $(k \leftarrow 0 \text{ to } k < c)$ do				
4: Envia LoadRequest para $T_{(i,j,k)}$ ;				
5: end for				
6: end for				

Quando  $MT_i$  recebe mensagem do tipo *Load* de uma *thread* trabalhadora  $T_{(i,j,k)}$ ,  $MT_i$  adiciona a carga recebida à mensagem *GlobalLoad*, que está em construção. Assim, que



Figura 5.8: Exemplo de requisição da carga gerenciadora para as threads trabalhadoras.

 $MT_i$  receber a resposta de todas as suas *threads*, linha 3 do Algoritmo 10,  $MT_i$  envia a mensagem *GlobalLoad* para a gerenciadora  $MT_{i'}$ , Figura 5.9. Assim, a quantidade de nós enviadas para que  $MT_{i'}$  distribua entre suas *threads* é igual a soma da quantidade de nós obtidos como resposta de todos as mensagens *LoadRequest* enviadas por  $MT_i$  à suas *threads* trabalhadoras, ou seja, é igual a  $\sum_{k=0}^{c} \frac{numeroNos(TL_{(i,j,k)})}{2}$ . Este procedimento é ilustrado na linha 2 do Algoritmo 10.



**Figura 5.9:** A *thread* gerenciadora envia carga para a *thread* gerenciadora que requisitou carga.

```
      Algorithm 10 Quando MT_i recebe Load from T_{(i,j,k)}.

      1: numLoad++;

      2: GlobalLoad \leftarrow GlobalLoad + Load;

      3: if (numLoad = c) then

      4: if (GlobalLoad \neq \emptyset) then

      5: Envia GlobalLoad para MT_{i'};

      6: else

      7: Envia NoLoad para MT_{i'};

      8: end if

      9: end if
```

Os algoritmos do framework apresentados foram desenvolvidos considerando a hierarquia de memória dos clusters de multicore. A próxima seção apresenta detalhes relativos à implementação do framework. É discutido também como foram escolhidos os valores de algumas constantes utilizadas, como o valor de NT e a quantidade de carga enviada diante de um recebimento de uma mensagem do tipo LoadRequest. Além disto, descreve como foi realizado o gerenciamento da memória para evitar os gargalos de compartilhamento de recursos.

### 5.2.3 Questões relativas à implementação do *Framework* de Balanceamento de Carga

O  $PBB_{SPP}$  foi implementado considerando o modelo de escalonamento proposto. Os seguintes itens foram considerados durante a implementação do  $PBB_{SPP}$ : i) tamanho e alocação das listas que armazenam os nós da árvore B&B, ii) como foram realizadas as transferências locais de carga na memória, iii) valor da constante NT, e iv) detalhes gerais de implementação.

### 5.2.3.1 Tamanho da lista

Para evitar a contenção de memória, a lista de nós  $TL_{(i,j,k)}$  é gerenciada por cada thread  $T_{(i,j,k)}$  durante a execução do  $PBB_{SPP}$ . O tamanho da lista  $TL_{(i,j,k)}$  é igual ao tamanho da cache, neste trabalho L2, disponível para a execução da thread  $T_{(i,j,k)}$ . Assim, quando  $T_{(i,j,k)}$  termina a execução de um nó da árvore B&B,  $T_{(i,j,k)}$  verifica se os nós gerados cabem na lista. Se couberem, os nós são inseridos no final da lista. Por outro lado, se a lista está cheia, ou seja, se a soma do tamanho dos nós da lista ultrapassa a quantidade de memória

disponível para  $T_{(i,j,k)}$  executar, os nós da árvore B&B são resolvidos recursivamente. Sendo assim, novos nós não são armazenados na lista evitando que a mesma ultrapasse o tamanho da memória disponível. Quando o procedimento recursivo termina,  $T_{(i,j,k)}$ seleciona o próximo nó da lista para executar, se os nós gerados não couberem na lista, o procedimento recursivo continua ativo, caso contrário, o procedimento termina e os nós voltam a ser armazenados na lista.

Durante o procedimento recursivo, a  $T_{(i,j,k)}$  pode receber mensagens do tipo LoadRequest e executar o Algoritmo 6. Note que, se este algoritmo for executado, o procedimento recursivo irá ser abortado, já que, em resposta a uma mensagem LoadRequest,  $T_{(i,j,k)}$ divide com a thread requisitante sua lista. Logo, o tamanho da lista diminui e novos nós podem ser inseridos até que o limite seja alcançado novamente e o procedimento recursivo é re-iniciado.

Vale perceber que a lista global  $ML_i$  da thread gerenciadora, pode, antes do envio da mensagem GlobalLoad ou antes da distribuição dos nós entre as threads trabalhadoras, ultrapassar a quantidade de memória livre para execução, já que possui metade dos nós de todas as listas das threads trabalhadoras. Entretanto, a transferência de carga entre máquinas ocorre em uma quantidade muito menor do que as transferências internas da máquina. Além disto, esta estratégia é utilizada porque o tempo de comunicação entre as máquinas é maior do que o tempo de execução dos nós e a transmissão de várias mensagens menores poderia aumentar a frequência da comunicação na rede e impactar negativamente no desempenho da aplicação.

#### 5.2.3.2 Alocação e acesso às listas de nós

Para utilizar dados do último nível da cache, a lista local de nós  $TL_{(i,j,k)}$  para cada thread foi implementada, aumentando as chances de processar os dados sem acessar à memória principal. Entretanto, o chamado false sharing pode ocorrer quando threads de diferentes cores atualizam posições diferentes de uma mesma linha de cache. Neste caso, uma vez que as posições de escrita de memória são diferentes, não existe problema real de coerência de cache, mas o protocolo de coerência de cache invalida a linha de cache. Quando existem requisições de acessos à outra localização desta linha, o protocolo forçará a atualização desta linha de cache na memória (mesmo isto não sendo necessário em termos lógicos). Frequentes atualizações de dados na linha de cache compartilhada podem causar sérios problemas de desempenho. Para prevenir o false sharing, a lista de nós de cada thread trabalhadora foi alocada utilizando a função posix\_memalign [46]. Esta função aloca a primeira posição da lista na primeira posição de uma linha de cache, e consequentemente, as *threads* da aplicação não compartilharão a mesma linha.

Como previamente descrito, durante o procedimento de balanceamento de carga, uma lista global,  $ML_i$ , de nós é criada por  $MT_i$ . A thread gerenciadora utiliza esta lista para armazenar os nós que serão transferidos para outra máquina e para disponibilizar os nós que são recebidos em uma mensagem do tipo GlobalLoad. As threads trabalhadoras copiam de  $ML_i$  para sua lista os nós que lhe foram designados pela thread gerenciadora. Logo,  $ML_i$  é compartilhada por todas as threads trabalhadoras da máquina  $M_i$  e a remoção dos nós desta lista é implementada utilizando as regras clássicas do problema produtor-consumidor.

A transferência de carga entre as *threads* trabalhadoras alocadas na mesma máquina é realizada através de uma lista temporária de nós. A *thread*  $T_{(i,j,k)}$  que está doando a carga, cria uma lista temporária contendo a metade de seus nós, como apresentado anteriormente no Algoritmo 6. Depois disto,  $T_{(i,j,k)}$  está livre para continuar sua execução e a *thread* requisitante também está livre para transferir a carga da lista temporária para sua lista. A *thread* requisitante é responsável por apagar esta lista após a operação de cópia.

#### 5.2.3.3 Valor do parâmetro NT

O valor de NT é utilizado pela thread  $MT_i$  para gerenciar quantas threads trabalhadoras lhe enviaram mensagem do tipo LoadRequest. Atingindo a quantidade NT de requisições,  $MT_i$  envia mensagem de requisição de carga pela rede. O overhead deste procedimento é alto pois, além de implicar em troca de mensagens pela rede, exige a comunicação da thread gerenciadora com todas as suas thread trabalhadoras. Note que, além disto, para executar os seus procedimentos a thread gerenciadora deve trocar de contexto com uma thread trabalhadora pois elas compartilham o mesmo recurso de processamento. Logo, a execução deste procedimento deve ser evitada tanto quanto possível para não piorar o desempenho da aplicação. Vale ressaltar que alguns testes foram realizados para avaliar se criar menos uma thread por máquina para evitar este compartilhamento levaria a melhores resultados. Foi observado que deixar de utilizar um core durante a execução de toda a aplicação prejudica muito o desempenho da mesma, visto que este core não é utilizado para resolver os nós da árvore e permanece ocioso durante a maior parte da execução.

Uma thread  $T_{(i,j,k)}$  que está com a sua lista vazia pode gerar carga durante a execução. Esta thread só está de fato ociosa se a sua lista está vazia e ela não está executando. Sendo assim, após  $T_{(i,j,k)}$  enviar mensagem LoadRequest para  $MT_i$  ao invés de entrar em estado de espera, a thread continua executando o Algoritmo 5 até que receba carga de  $MT_i$  ou receba carga de uma das threads trabalhadoras. Considerando o último caso,  $T_{(i,j,k)}$  avisa a  $MT_i$  a sua desistência de participar do balanceamento de carga entre máquinas e o valor de NT é decrementado.

O limite de requisições necessárias para a execução do Algoritmo 7 foi avaliado durante a realização dos testes. Como exemplo, considere os valores extremos para NT como segue. Para NT = 1 o procedimento será realizado quando apenas uma *thread* trabalhadora não conseguir carga com uma das *threads* trabalhadoras de sua máquina. A quantidade de carga que será recebida é igual a soma da metade das listas das trabalhadoras da máquina que está doando a carga. Logo, a tendência é que a máquina requisitante fique mais sobrecarregada do que aquela que doou carga, já que apenas uma *thread* irá receber toda esta carga. Assim, se apenas uma *thread* está com sua lista vazia, ela poderia tentar conseguir os nós gerados durante a computação das demais *threads*. Por outro lado, se NTfor igual à quantidade de *threads* trabalhadoras, a transferência de carga ocorrerá somente quando todas as *threads* estiverem sem carga, logo, a máquina poderá ficar ociosa.

Neste trabalho, o valor atribuído a NT foi igual a metade do número de *threads* trabalhadoras da máquina. Os resultados mostram que este valor ajudou a melhorar o desempenho, pois foram evitadas transferências desnecessárias de carga via rede.

Vale ressaltar que durante a transferência de carga entre as *threads* trabalhadoras da mesma máquina foram utilizadas memória compartilhada e semáforos para garantir a consistência de dados [84]. Para a transferência de carga entre as *threads* foram utilizados MPI\_Iprobe e MPI\_Irecv para testar e receber requisições de carga ou carga sem qualquer bloqueio. As mensagens enviadas foram implementadas utilizando MPI\_Isend para acelerar a execução do  $PBB_{SPP}$  [81].

# Capítulo 6

# Resultados do Balanceamento de Carga proposto para um *B*&*B* paralelo baseado no MECM

Este capítulo apresenta os resultados obtidos pela execução do  $PBB_{SPP}$ . Os experimentos foram realizados como o intuito de avaliar se o framework baseado no RCM e no MECM obtém resultados relevantes quanto ao acesso à memória e a estratégia de balanceamento de carga proposta. Para verificar a eficiência do framework, o  $PBB_{SPP}$  foi comparado com uma versão do algoritmo B&B aplicado ao SPP, mas que não utiliza qualquer procedimento de balanceamento de carga. Esta versão é denominado  $PBBNLB_{SPP}$ . Entretanto, como apresentado anteriormente, o B&B gera a árvore dinamicamente e um algoritmo paralelo sem balanceamento de carga não apresenta bom desempenho. Diante disto, foi implementado uma versão do algoritmo B&A que utiliza em seus procedimentos de balanceamento de carga uma estratégia de comunicação p2p baseada em uma árvore, denominada Overley-Centric proposta em [92]. Esta versão é referenciada no texto como  $OCLB_{SPP}$ .

Os experimentos apresentados neste capítulo foram executados no *Cluster Oscar* que é composto por 32 máquinas. Cada umas destas máquinas possui dois processadores *quadcore* (Intel Xeon 5355 Clovertown). Cada *core* possui uma cache L1 privada (64 KB) e compartilham uma cache L2 de 8MB com outro *core* do mesmo processador. Todos os *cores* de um mesmo nó possuem acesso uniforme à memória RAM de 16GB. O sistema operacional é CentOS 5.3 com kernel versão 2.6.18. Vale lembrar que, a quantidade de *threads* trabalhadoras é referente a quantidade de *cores* da máquina, ou seja, igual a oito. Além disto, uma vez que uma *thread* compartilha cache com outra *thread* o tamanho de sua lista não excede a metade do tamanho da cache L2 disponível (4MB). Experimentos e avaliações a respeito da quantidade de mensagens trocadas entre as threads e da escalabilidade do algoritmo também foram realizados. As próximas seções apresentam os resultados de cada experimento contendo detalhes da implementação dos algoritmos utilizados nas comparações.

# 6.1 Comparando o $PBB_{SPP}$ com o $PBBNLB_{SPP}$ (B&B paralelo sem Balanceamento de Carga)

O  $PBBNLB_{SPP}$  executa o mesmo procedimento de Distribuição de Carga do  $PBB_{SPP}$ , apresentado no Algoritmo 3. Quando uma thread  $T_{(i,j,k)}$  termina a execução da carga que lhe foi atribuída pela thread gerenciadora  $T_{(i,j,k)}$  se torna ociosa e fica em estado de espera até que a aplicação termine. Os algoritmos de balanceamento de carga somente são executados nesta versão do B&B se  $T_{(i,j,k)}$  não recebe carga inicial. Assim,  $T_{(i,j,k)}$  inicia o procedimento de busca de carga seguindo a hierarquia de requisições estabelecidas no Algoritmo 5 do Capítulo 5. Este procedimento foi utilizado para que todas as threads trabalhadoras de todas as máquinas participem da execução da aplicação.

Vale ressaltar que o procedimento de detecção de terminação da aplicação é mais simples e consome menos tempo de execução no  $PBBNLB_{SPP}$  do que no  $PBB_{SPP}$ . Uma vez que nenhum balanceamento de carga é realizado, quando  $T_{(i,j,k)}$  termina a execução da carga que lhe foi atribuída avisa à *thread* gerenciadora o seu término. Quando a *thread* gerenciadora recebe a mensagem *Idle* de todas suas trabalhadoras, esta envia mensagem para a *thread* líder indicando o término de execução em sua máquina. No algoritmo  $PBB_{SPP}$  o procedimento de detecção de terminação é mais complexo já que uma *thread* que termina a execução da carga pode voltar a executar outra carga recebida. Este algoritmo foi baseado nos algoritmos clássicos de detecção de terminação descritos em [14].

Para avaliar a qualidade da distribuição de carga entre as *threads* durante a execução da aplicação um fator de grau de "desbalanceamento" de carga foi utilizado. Este fator, denominado  $Un\_Factor$ , foi proposto em [57] e é obtido de acordo com a equação 6.1, onde  $T_{med}$  é o tempo médio de execução total de todas as *threads* e  $T_{max}$  é o maior tempo de execução entre todas as *threads*.

$$Un\_Factor = 1 - \frac{T_{med}}{T_{max}}$$
(6.1)
As Tabelas 6.1 e 6.2 apresentam os resultados obtidos pelos algoritmos sem balanceamento de carga e com balanceamento de carga, respectivamente. Nas duas tabelas são apresentados: o tempo de execução (*wall clock time*) dos algoritmos em "*Tempo (s)*", o número de nós da árvore B&B processados durante a execução em "# de Nós", o  $Un\_Factor$ , o Speedup e em "E" a porcentagem de melhora do algoritmo relação a versão sequencial, ou seja, quanto o tempo do  $SBB_{SPP}$  foi reduzido considerando a versão paralela implementada. Na Tabela 6.2, além dos valores previamente descritos, foi adicionada a coluna "% Redução" que representa a porcentagem de redução em relação ao tempo de execução (tempo de resposta ou tempo de parede) do  $PBB_{SPP}$  quando comparado com a versão sem balanceamento de carga. Devido a aleatoriedade da aplicação, todas as versões do algoritmo foram executadas dez vezes e os resultados (tempo e nós resolvidos) apresentados é igual a média destes dez resultados. Estes testes foram executados com 16 *threads* trabalhadoras, divididas igualmente entre duas máquinas do *cluster*.

Comparando as Tabelas 6.1 e 6.2 pode-se perceber que o tempo de execução do algoritmo paralelo para ambos os casos é muito menor do que o tempo obtido pela versão sequencial. Além disto, o tempo de redução do  $PBB_{SPP}$  em relação a versão sem balanceamento de carga chega atingir 90% para a instância I200-650-0.02-152. Note que o valor de  $Un_factor$  é quase zero para todas as instâncias, indicando que o  $PBB_{SPP}$  mantém as threads balanceadas durante a execução. Entretanto, isto não indica que as threads executaram a mesma quantidade de nós do B&B, mas que elas se mantiveram com carga ou trabalho durante o tempo total de execução.

Uma vez que as threads ficaram ociosas por muito tempo na versão  $PBBNLB_{SPP}$ , tanto o valor de  $Un\_factor$  quanto o Speedup, se mostram longe do esperado para uma aplicação paralela. Entretanto, quando analisado os valores do Speedup obtidos, a versão  $PBB_{SPP}$  alcançou valores super-lineares para algumas instâncias. Speedups superlineares, além de indicar uma boa paralelização da técnica, também é da natureza do B&B. Se uma boa solução é descoberta nas sub-árvores da árvore do B&B que são exploradas primeiro, as próximas sub-árvores podem ser podadas mais rapidamente e o tempo total de execução é então reduzido, uma vez que a quantidade de nós da árvore diminui. Por outro lado, se o algoritmo não tem sorte, o tempo pode ser pior devido a busca em sub-árvore profundas que não possuem boas soluções. O algoritmo sequencial é mais afetado por este fenômeno do que o algoritmo paralelo, uma vez que na versão paralela, várias partes da árvore estão sendo exploradas simultaneamente.

**Tabela 6.1:** Resultados obtidos pela execução da versão do B&B sem balanceamento de carga para o SPP,  $PBBNLB_{SPP}$ , utilizando duas máquinas do *cluster* (16 *cores*).

$PBBNLB_{SPP}$								
Intâncias	Tempo(s)	# de Nós	Un_Factor	Speedup	$\mathbf{E}$			
I90-400-0.03	4,91	44.929,50	0,92	2,54	$60,\!57$			
I90-400-0.04	$10,\!61$	$117.290,\!60$	0,99	$2,\!15$	$53,\!51$			
I90-400-0.05	21,57	285.740,00	$0,\!98$	2,42	58,74			
I100-500-0.03	13,70	132.010,50	$0,\!95$	$2,\!12$	$52,\!88$			
I100-500-0.04	67,08	792.794,11	$0,\!99$	4,50	77,77			
I100-500-0.05	217,00	2.107.049,40	$1,\!00$	$1,\!95$	48,78			
I110-750-0.03	3.319,32	24.055.515,10	$0,\!99$	$3,\!41$	$70,\!66$			
I110-750-0.04	$14.559,\!37$	$193.599.713,\!80$	$1,\!00$	2,09	$52,\!23$			
I110-750-0.05	6.099,06	$112.916.399,\!67$	$1,\!00$	3,79	$73,\!64$			
I200-650-0.02-100	$5.225,\!55$	$13.021.809{,}50$	$1,\!00$	1,74	$42,\!51$			
I200-650-0.02-152	$10.897,\!56$	24.322.846,30	0,91	$1,\!47$	$31,\!85$			
I200-600-0.04	$9.691,\!90$	$141.767.609,\!90$	$1,\!00$	$3,\!58$	$72,\!03$			

Tabela 6.2: Resultados obtidos pelo  $PBB_{SPP}$  utilizando duas máquinas do *cluster*.

$PBB_{SPP}$								
Intâncias	% Redução	Tempo(s)	# de Nós	Un_Factor	Speedup	$\mathbf{E}$		
I90-400-0.03	86,68	$0,\!65$	20.587,00	0,03	$19,\!05$	94,75		
I90-400-0.04	$80,\!63$	2,06	$120.645,\!40$	0,01	$11,\!10$	$90,\!99$		
I90-400-0.05	80,62	4,18	$290.608,\!60$	0,02	$12,\!51$	$92,\!00$		
I100-500-0.03	82,76	2,36	97.671,00	0,01	$12,\!31$	$91,\!87$		
I100-500-0.04	$82,\!53$	11,72	$597.673,\!80$	0,01	25,76	$96,\!12$		
I100-500-0.05	$86,\!46$	$29,\!38$	2.154.848,00	0,00	$14,\!42$	$93,\!06$		
I110-750-0.03	80,71	$640,\!39$	22.322.896,00	0,00	$17,\!67$	$94,\!34$		
I110-750-0.04	$77,\!54$	3.269,41	$155.945.803,\!00$	0,00	9,32	$89,\!27$		
I110-750-0.05	$74,\!49$	$1.555,\!93$	$89.970.708,\!25$	0,00	$14,\!87$	$93,\!28$		
I200-650-0.02-100	88,60	$595,\!97$	12.956.182,00	0,00	$15,\!25$	$93,\!44$		
I200-650-0.02-152	$90,\!42$	1.044,06	$24.328.951,\!83$	$0,\!00$	$15,\!32$	$93,\!47$		
I200-600-0.04	$76,\!55$	$2.272,\!50$	$132.762.999,\!14$	0,00	$15,\!25$	$93,\!44$		

## 6.2 Comparando o $PBB_{SPP}$ e o $OCLB_{SPP}$ .

Os resultados apresentados na seção anterior mostram a eficiência do  $PBB_{SPP}$ . Entretanto, a estratégia de balanceamento de carga utilizada foi comparada apenas com uma versão que não utiliza qualquer função de balanceamento. Para melhor avaliar os procedimentos do *framework* proposto, foi implementada uma versão do B&B aplicado ao SPPque utiliza os procedimentos de balanceamento de carga propostos em [92].

Poucos trabalhos na literatura apresentam implementação do B&B que considera a arquitetura de memória compartilhada e distribuída presentes nos *clusters* de *multicore*. Dentre estes trabalhos dois deles, propostos em [19] e [92], foram analisados para serem implementados. Tanto em [19] quanto em [92] é utilizado o paralelismo a nível de *cores* e a execução distribuída a nível de máquina.

Em [19] é proposta uma biblioteca para automatizar a execução distribuída do algoritmo B&B. Entretanto, é utilizado um modelo de computação limitado que impõe algumas restrições importantes na implementação do algoritmo. Os procedimentos de balanceamento de carga, devido às restrições, executam em *rounds*. Uma máquina que se tornou ociosa durante a execução do *round* deverá esperar até o final do *round* para obter carga, que é quando acontece a re-distribuição de carga entre as máquinas. Assim, todas as máquinas iniciam um novo *round* com, aproximadamente, a mesma quantidade de sub-problemas abertos. Entende-se como sub-problemas abertos as sub-árvores ainda não exploradas do B&B. Durante a execução da aplicação acontecem vários pontos de sincronização referentes aos períodos de re-distribuição de carga. Apesar destes pontos de sincronização para realizar o balanceamento de carga, os autores mostram que um algoritmo B&B aplicado ao problema de Steiner em Grafos é eficientemente executado neste modelo.

Já em [92] é proposta uma organização dos recursos computacionais (cores) em uma árvore lógica seguindo a topologia peer-to-peer, onde cada peer representa um core. Esta árvore define como os cores estão logicamente conectados. Durante o procedimento de balanceamento de carga, quando uma tarefa se torna ociosa, pode enviar requisições de carga somente para as tarefas alocada aos cores que estão à uma aresta de distância do core em que está executando. Assim, a tarefa  $v_i$  poderá enviar mensagem para a tarefa alocada ao core que é seu pai na árvore lógica e às tarefas alocadas aos cores que são seus filhos. A Figura 6.1 ilustra uma árvore lógica de um ambiente multicore que possui seis cores. Neste caso, por exemplo, a tarefa alocada ao core 2 só poderá enviar requisições de carga às tarefas alocadas aos *cores* 0,  $3 \in 5$  (arestas pontilhadas da figura). Vale notar que, pela árvore lógica não se sabe se os *cores* pertence ou não à mesma máquina.



Figura 6.1: Exemplo uma árvore lógica que representa um ambiente contendo 6 cores.

Inicialmente, as mensagens de requisição de carga de uma tarefa são enviadas, sequencialmente, às suas tarefas filhas, ou seja, às tarefas alocadas aos *cores* filhos do *core* onde a tarefa ociosa está alocada na árvore. A requisição é enviada à uma tarefa filha selecionada aleatoriamente a cada requisição. Se nenhuma das tarefas alocadas aos processos filhos possuem carga, a tarefa requisita carga à tarefa pai. Entretanto, para agilizar a obtenção de carga, é possível criar arestas lógicas na árvore que permitem que a tarefa envie pedido de carga a qualquer outra tarefa alocada a um *core* que não faz parte da sua sub-árvore. Além disto, quando a tarefa alocada ao *core* pai é notificada que uma tarefa alocada a um *core* filho está sem carga, essa tarefa compartilha carga com a tarefa ociosa.

Os procedimentos de balanceamento de carga proposto em [92] seguem os seguintes princípios:

- Grau da Árvore/Diâmetro: a árvore lógica é obtida atribuindo aleatoriamente cores a seus nós. O grau da árvore é uma parâmetro de criação da mesma. No exemplo da Figura 6.1, a árvore é de grau igual a 2. A respeito da escolha do grau da árvore, vale notar que, quando o grau usado como parâmetro é muito alto, as tarefas poderão obter carga mais rapidamente do que quando um grau pequeno é usado para definir a árvore. Entretanto, neste caso, gargalos podem causar problemas de escalabilidade.
- Quantidade de carga: A carga transferida é calculada considerando os tamanhos das sub-árvores e o tipo de dependência da tarefa requisitante e da tarefa que recebeu a requisição de carga na árvore lógica. Se uma tarefa u envia uma requisição a uma tarefa v que está alocada ao *core* que é o pai lógico do *core* onde u está alocada, a quantidade de carga a ser transferida é igual a  $T_u/T_v$ , onde  $T_u$  é o tamanho da sub-árvore de u. Por outro lado, quando v enviar requisição para u, a quantidade de carga transferida é igual a  $(T_v T_u)/T_v$ .

• Arestas Pontes: Novas arestas podem ser criadas entre os *cores* da árvore para acelerar a transferência de cargas. Se uma tarefa v não consegue obter carga com as tarefas alocadas aos *cores* filhos ou pai do *core* onde executa, uma aresta ponte é criada para conectar aleatoriamente v a uma outra tarefa u. A quantidade de carga transferida entre elas é igual a  $T_v/(T_u + T_v)$ .

Considerando que a estratégia de balanceamento de carga proposta em [19] é limitada pela execução de *rounds* sincronizados e a um ambiente de execução com *softwares* específicos, a versão do balanceamento de carga proposta em [92] foi implementada para o SPPe comparada com o  $PBB_{SPP}$ . Como descrito acima, esta versão é chamada  $OCLB_{SPP}$  e, assim como no  $PBB_{SPP}$ , uma *thread* foi criada por *core*.

A comparação entre o  $PBB_{SPP}$  e o  $OCLB_{SPP}$  foi realizada, inicialmente, considerando 16 cores do Cluster Oscar. Baseado nos resultados apresentados em [92] e no ambiente de teste disponível, três diferentes árvores foram geradas com graus máximos iguais a  $Grau_{max} = 3$ , 7, e 15. Note que, este último valor de  $Grau_{max}$  corresponde a usual técnica mestre-trabalhador. A Tabela 6.3 apresenta o tempo de execução dos algoritmos  $PBB_{SPP}$  e  $OCLB_{SPP}$ . Independente do grau da árvore escolhida para o  $OCLB_{SPP}$ , o tempo de execução do  $PBB_{SPP}$  é muito melhor em todos os casos. Observe que nenhuma informação a respeito do ambiente de execução é utilizada no  $OCLB_{SPP}$ . Gargalos de acesso a memória não são evitados e, além disto, e como a árvore é gerada aleatoriamente, cores pertencente a máquinas diferentes podem ser pai e filho, o que aumenta a comunicação via rede. Além disto, as tarefas ficam mais tempo ociosas no  $OCLB_{SPP}$ , como pode ser visto no gráfico da Figura 6.2, que mostram a comparação do  $Un_Factor$  para as duas versões. O valor do  $Un_Factor$  foi muito maior para todas as instâncias quando utilizado o  $OCLB_{SPP}$ .

O speedup entre as versões também foi comparado e é apresento no gráfico da Figura 6.3. O valor do speedup obtido pelo  $OCLB_{SPP}$  é baixo para a maioria das instâncias, independente do grau da árvore. Somente a instância I90-400-0.03, a que possui menor tempo de execução, que apresentou um speedup super-linear.

Para avaliar a sobrecarga da paralelização utilizada nos algoritmos  $PBB_{SPP}$  e no  $OCLB_{SPP}$  com  $Grau_{max} = 7$  as diferentes fases das estratégias de balanceamento de carga foram avaliadas considerando o tempo de execução de cada fase e o número de requisição de carga enviadas pelas tarefas. O grau igual a 7 foi escolhido porque apresentou o melhor resultados médio entre os graus testados.

**Tabela 6.3:** Comparação entre o mecanismo de balanceamento de carga do  $PBB_{SPP}$  e a versão paralela do B&B com balanceamento de carga baseado na árvore *Overlay-Centric*,  $OCLB_{SPP}$ .

Intâncias	$PBB_{SPP}$	$OCLB_{SPP}$	$OCLB_{SPP}$	$OCLB_{SPP}$
		$Grau_{max} = 3$	$Grau_{max} = 7$	$Grau_{max} = 15$
I90-400-0.03	$0,\!65$	1,56	0,73	1,25
I90-400-0.04	2,06	$4,\!15$	3,36	3,97
I90-400-0.05	4,18	10,99	$8,\!93$	10,70
I100-500-0.03	$2,\!36$	6,40	$5,\!36$	$6,\!90$
I100-500-0.04	11,72	32,23	$32,\!30$	46,53
I100-500-0.05	29,38	$116,\!57$	$64,\!84$	128,87
I110-750-0.03	$640,\!39$	$2.630,\!47$	$2.241,\!30$	$1.862,\!62$
I110-750-0.04	3.269,41	11.331,73	$8.359{,}58$	$10.351,\!12$
I110-750-0.05	$1.555,\!93$	7.659,91	$4.423,\!00$	$4.587,\!23$
I200-650-0.01-100	$595,\!97$	$3.101,\!51$	$2.708,\!97$	$2.929,\!91$
I200-650-0.02-152	1.044,06	4.005,26	$3.887,\!32$	$4.946,\!57$
I200-600-0.04	$2.272,\!50$	$5.554,\!66$	$6.926,\!67$	$6.871,\!57$



**Figura 6.2:** Comparação entre o  $Un\_Factor$  do  $PBB_{SPP}$  e do  $OCLB_{SPP}$  com  $Grau_{max}=3, 7$  e 15 para as instâncias apresentadas na Tabela 6.3.

Na Tabela 6.4, para o algoritmo proposto  $PBB_{SPP}$  é apresentada a porcentagem de tempo médio em que a thread permanece esperando por carga em relação ao tempo total de execução, quando as requisições acontecem entre as tarefas alocadas a cores de uma mesma máquina e entre as threads gerenciadoras. Estes valores são apresentados nas colunas "% de Tempo\_Local" e "% de Tempo\_Global", respectivamente. Como o  $OCLB_{SPP}$ não distingue a localidade dos cores, a porcentagem de tempo apresentada na coluna "% de Tempo" é referente ao tempo total de ociosidade de todas as threads da aplicação. Pode-se perceber que o tempo em que as threads permaneceram ociosas na estratégia  $OCLB_{SPP}$  é, em geral, muito maior do que a soma dos tempos do  $PBB_{SPP}$ .



**Figura 6.3:** Speedup do  $PBB_{SPP} \in OCLB_{SPP} \mod Grau_{max}=3$ , 7 e 15 para as instâncias apresentadas na Tabela 6.3.

Nesta tabela também é apresentado o número de mensagens do tipo LoadRequest enviadas entre as tarefas de uma mesma máquina na coluna "# de Req\_Local", e em máquina diferentes, coluna "# de Req\_Global". É visto que o número de requisições locais no  $PBB_{SPP}$  é maior do que o número de requisições globais, mas a troca de mensagens locais impactam menos no tempo de execução da aplicação do que as enviadas via rede. No  $OCLB_{SPP}$ , apesar do número de requisições enviadas ser menor do que no  $PBB_{SPP}$ , tanto na mesma máquina quanto em máquinas diferentes, o tempo médio de espera por carga é muito maior. Isto ocorre porque uma tarefa v, no pior caso, requer carga para todas as tarefas filhas. Se v não receber carga, uma requisição é enviada à tarefa pai. Se não obtiver carga, mensagens são enviadas aleatoriamente às demais tarefas, através da criação de arestas pontes. Se a tarefa v não conseguir obter carga, durante este procedimento, v fica em estado de espera até receber carga do pai. No  $PBB_{SPP}$ , mesmo tendo enviado pedido de carga para à thread gerenciadora, a tarefa continua tentando obter carga com as tarefas de sua máquina e transferências de carga podem ocorrer entre estas threads antes que a thread gerenciadora consiga mais carga.

Os tamanhos das mensagens trocadas via rede no  $PBB_{SPP}$  são apresentados na Tabela 6.5. Este teste foi realizado com o intuito de verificar o impacto do tamanho das mensagens trocadas no desempenho total da aplicação, uma vez que no Capítulo 4, o MECM indica que longas mensagens enviadas pela rede pode piorar o desempenho da aplicação. Como apresentado na Tabela 6.5, as mensagens trocadas não foram maiores que 4 Mbytes, onde "Maior", indica o tamanho da maior mensagem enviada para cada instância executada,

		PBB	SPP		OCLB	SPP Gro	$au_{max} = 7$
Instâncias	% de	# de	% de	# de	% de	# de	# de
	Tempo	Req	Tempo	Req	Tempo	Req	Req
	Local	Local	Global	Global		Local	Global
I90-400-0.03	$1,\!27$	24,02	40,64	$3,\!45$	36,01	9,89	3,69
I90-400-0.04	$0,\!62$	$54,\!21$	23,71	$5,\!34$	$45,\!03$	13,75	$3,\!87$
I90-400-0.05	$0,\!45$	$95,\!08$	$17,\!32$	6,71	$52,\!85$	$23,\!09$	4,05
I100-500-0.03	0,99	$53,\!95$	$19,\!42$	$5,\!39$	$51,\!05$	21,09	4,85
I100-500-0.04	$0,\!49$	$157,\!30$	$12,\!82$	$12,\!30$	$43,\!68$	$41,\!59$	$5,\!19$
I100-500-0.05	$0,\!34$	$262,\!53$	5,85	$16,\!60$	$45,\!81$	46,77	$^{5,15}$
I110-750-0.03	0,26	705,96	2,56	24,77	58,78	$78,\!28$	6,92
I110-750-0.04	$0,\!44$	946, 16	0,93	25,72	29,77	106, 13	6,30
I110-750-0.05	$0,\!45$	930, 31	0,86	$23,\!28$	$53,\!44$	66,22	$7,\!53$
I200-650-0.02-100	0,16	695, 31	$2,\!45$	$35,\!91$	$71,\!19$	$52,\!23$	$6,\!18$
I200-650-0.02-152	$0,\!17$	$793,\!89$	1,72	38,11	66,30	48,89	$5,\!65$
I120-600-0.04	$0,\!50$	$1.110,\!62$	$1,\!13$	$28,\!29$	$55,\!86$	$65,\!11$	$^{5,20}$

**Tabela 6.4:** Informações sobre a comunicação entre as tarefas  $PBB_{SPP}$  e do  $OCLB_{SPP}$ , com  $Grau_{max} = 7$ .

"*Menor*" o tamanho da menor mensagem e "*Média*", a média entre todos os tamanhos das mensagens do tipo *Load* enviadas.

**Tabela 6.5:** Tamanho das mensagens (KB) do tipo *Load* trocadas entre as tarefas no  $PBB_{SPP}$ .

Instâncias	Maior	Menor	Média
I90-400-0.03	591,73	$^{8,50}$	$336,\!28$
I90-400-0.04	480,80	29,10	310,38
I90-400-0.05	$174,\!81$	$^{3,32}$	$75,\!38$
I100-500-0.03	770,92	$16,\!48$	$403,\!37$
I100-500-0.04	982,40	$^{8,28}$	$524,\!22$
I100-500-0.05	810,58	$4,\!43$	$351,\!16$
I110-750-0.03	3.468,72	11,08	$1.903,\!83$
I110-750-0.04	3.076, 16	20,02	736, 14
I110-750-0.05	$215,\!04$	14,76	$839,\!86$
I200-650-0.02-100	1.730,42	86,31	894,08
I200-650-0.02-152	$1.279,\!84$	30,08	$748,\!10$
I200-600-0.04	1.395,79	4,92	$736,\!14$

#### 6.2.1 Escalabilidade

Para avaliar a escalabilidade do  $PBB_{SPP}$  e do  $OCLB_{SPP}$  com  $Grau_{max} = 7$  o número de máquinas disponíveis para executá-lo foi incrementado. À medida que mais máquinas são adicionadas no ambiente, aumenta-se também o número de *threads* executando. Em todos os casos foi alocada uma *thread* em cada *core*.

As Tabelas 6.6 e 6.7 mostram o tempo de execução total do  $PBB_{SPP}$  e  $OCLB_{SPP}$  com  $Grau_{max} = 7$  quando executados em quatro máquinas, sendo 32 threads trabalhadoras, e oito máquinas, com 64 threads trabalhadoras. Como pode ser observado o tempo de

execução do  $PBB_{SPP}$  melhorou à medida que mais máquinas, e consequentemente, mais threads trabalhadoras foram utilizadas. Uma vez que a quantidade de tarefas executando a aplicação é maior, a quantidade de mensagens trocadas entre elas também aumenta. Para as instâncias que possuem menor tempo de execução, aumentar a quantidade de máquinas não é interessante já que a quantidade de trabalho não é suficiente para manter todas as threads trabalhadoras com carga por um período de tempo significante.

Note que aumentar o número de máquinas piora o tempo de execução para a versão  $OCLB_{SPP}$ , uma vez que na estratégia de balanceamento de carga utilizada as threads ficam parte da execução ociosa. O valor de  $Un\_Factor$  apresentado é alto. Além disto, outra desvantagem do algoritmo  $OCLB_{SPP}$  é que para alcançar bons resultados é necessário avaliar vários valores de graus para a árvore, com o intuito de definir de forma mais eficientes quantos e em quais canais de comunicação as threads devem se comunicar de modo que obtenham carga o mais rapidamente. Finalmente, se a árvore fosse gerada baseada no compartilhamento de memória dos cores os resultados poderiam ser melhores. Neste caso, os cores filhos e o core pai de um determinado core da árvore deveriam ser cores pertencentes à sua máquina, tanto quanto possível.

**Tabela 6.6:** Execução do  $PBB_{SPP}$  em quatro máquinas (32 threads) e oito máquinas (64 threads).

	$PBB_{SPP} \text{ com } 32 \text{ threads}$				$PBB_{SPP} \text{ com } 64 \text{ threads}$			
Instâncias	Tempo	# de Nós	$Un_{-}$	Speedup	Tempo	# de Nós	$Un_{-}$	Speedup
			Factor				Factor	
I90-400-0.03	0,88	31.667,90	0,10	14,22	1,04	35.220,00	0,15	12,00
I90-400-0.04	1,97	119.814,00	0,03	$11,\!60$	1,87	116.481,38	0,03	12,19
I90-400-0.05	3,30	291.509,43	0,04	$15,\!84$	4,25	291.772,00	0,05	12,30
I100-500-0.03	2,47	107.840,40	0,05	11,79	2,16	99.261,00	0,11	13,48
I100-500-0.04	10,20	711.568,70	0,01	29,60	13,14	765.128,90	0,01	22,97
I100-500-0.05	19,75	2.137.913,70	0,01	21,45	19,91	2.160.679,57	0,01	21,28
I110-750-0.03	405,09	21.613.244,71	0,00	27,93	210,02	21.805.859,71	0,00	$53,\!87$
I110-750-0.04	2.644,35	191.386.674,39	0,00	11,52	1.602,74	198.347.752,50	0,00	19,01
I110-750-0.05	1.019,06	107.704.982,14	0,00	22,71	517,52	107.665.128, 14	0,00	44,71
I200-650-0.02-100	429,17	12.961.413,10	0,00	21,18	229,03	12.960.323,56	0,00	$39,\!68$
I200-650-0.02-152	751,19	24.326.748,30	0,00	21,29	381,44	24.325.998,00	0,00	41,92
I120-600-0.04	1.479,59	$137.254.088,\!60$	0,00	23,42	747,05	136.636.500, 89	0,00	46,39

Os resultados apresentados neste capítulo mostram a eficiência da aplicação quando utilizados os procedimentos de balanceamento de carga do *framework* proposto. Estes procedimentos, como descrito anteriormente, foram desenvolvidos de acordo com o modelo de escalonamento, mostrando que o MECM e a representação do ambiente RCM podem auxiliar na implementação eficientes de mecanismos de escalonamento de tarefas. Entretanto, devido às características desta aplicação, não foi possível avaliar as demais condições do modelo (Condições 2.a e 2.b). Para tanto, foi proposta uma aplicação sintética do tipo DAG e um escalonamento de tarefas estático baseado no List Scheduling com o intuito de verificar qual o comportamento da aplicação quando as caches das máquinas

**Tabela 6.7:** Execução do  $OCLB_{SPP}$  em quatro máquinas (32 threads) e oito máquinas (64 threads).

	$OCLB_{SPP}$ com 32 threads		$OCLB_{SPP}$ com 64 threads					
Instâncias	Tempo	# de Nós	$Un_{-}$	Speedup	Tempo	# de Nós	$Un_{-}$	Speedup
			Factor				Factor	
I90-400-0.03	1,20	23.819,80	0,73	10,38	1,24	24.803,50	0,84	10,05
I90-400-0.04	5,37	120.116,30	0,81	4,25	4,27	118.520,50	0,87	$^{5,35}$
I90-400-0.05	$13,\!66$	283.757,00	0,80	3,83	10,95	287.278,00	0,89	4,77
I100-500-0.03	$5,\!80$	$114.113,\!60$	0,74	$^{5,02}$	7,05	115.912, 11	0,89	$^{4,12}$
I100-500-0.04	35,69	819.663,75	0,75	8,46	30,88	763.865,50	0,83	9,77
I100-500-0.05	$57,\!80$	2.129.071,75	$0,\!69$	7,33	89,35	2.020.138,80	0,90	4,74
I110-750-0.03	$2.258,\!81$	27.548.793,20	0,76	$^{5,01}$	2.560,37	29.185.224,83	0,90	$^{4,42}$
I110-750-0.04	8.550,28	172.032.773,00	0,73	3,56	8.817, 15	185.406.405,71	0,85	$^{3,46}$
I110-750-0.05	$3.553,\!84$	107.407.665, 29	0,71	6,51	3.665,68	108.684.298,20	0,85	6,31
I200-650-0.02-100	2.533,48	13.136.969,80	0,85	3,59	2.474,28	11.878.829,40	0,93	$^{3,67}$
I200-650-0.02-152	2.869,05	24.314.417,00	0,80	5,57	3.709, 11	24.327.694,00	0,91	$^{4,31}$
I120-600-0.04	5.571,96	133.864.723,00	0,74	6,22	8.388,51	136.160.274,50	0,91	$4,\!13$

do *cluster* são sobrecarregadas. Detalhes da aplicação e do algoritmo de escalonamento proposto são apresentados no próximo capítulo.

## Capítulo 7

# Algoritmo de Escalonamento baseado no modelo de escalonamento proposto

O problema de escalonamento de aplicações paralelas representadas por grafos acíclicos direcionados (GADs) em um conjunto de processadores é em sua forma geral, NP-completo [45, 79]. Algoritmos List Scheduling (LS) pertencem a uma classe de algoritmos heurísticos muito utilizados para escalonar aplicações modeladas por GADs devido à sua baixa complexidade e à boa qualidade das soluções produzidas [56, 87], principalmente quando os processadores possuem poder computacional distintos. No LS, as tarefas são ordenadas de acordo com uma prioridade que é atribuída a cada tarefa do GAD. A cada iteração a tarefa de maior prioridade é alocada a um recurso computacional que é escolhido segundo algum critério previamente definido, como por exemplo, o processador que produz o menor tempo de fim de execução para a tarefa que está sendo escalonada.

Para avaliar o modelo de escalonamento proposto foi desenvolvido um List Scheduling baseado em Memória,  $LS_{BM}$ , que atribui prioridades às tarefas considerando a quantidade de memória que as mesmas necessitam para executar. O  $LS_{BM}$  foi baseado no mecanismo de prioridades do LS e utiliza as Condições 2.a e 2.b do modelo de escalonamento proposto no Capítulo 4 para guiar a escolha do processador ao qual a tarefa será atribuída.

As aplicações sintéticas utilizadas nos testes deste capítulo são representadas por um GAD. Cada tarefa  $v \in V$  executa o Algoritmo 1, apresentado no Capítulo 3. Logo, após a fase de computação, a tarefa v envia uma mensagem de peso  $\omega(v, u)$  para a sua sucessora u. O peso de  $\mu_v$  é gerado aleatoriamente, entre os valores de 1MB a 10MB. O valor de  $\epsilon_v$  é referente ao tempo de execução da tarefa quando a mesma requer  $\mu_v$  (tamanho de dados) para executar. Este valor foi obtido executando a tarefa v em um *core* de uma máquina dedicada. O custo associado à comunicação é igual para todas as arestas do grafo. A Figura 7.1 ilustra um grafo G contendo oito vértices,  $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$  e oito arestas de comunicação,  $E = \{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_0, v_5), (v_1, v_6), (v_5, v_7), (v_6, v_7)\}$ . A dupla de valores  $(\mu_v, \epsilon_v)$  é associado a cada vértice  $v \in V$ . O valor próximo as arestas se refere ao peso da comunicação da aresta.



Figura 7.1: Grafo G

Para avaliar o algoritmo proposto, uma versão tradicional do algoritmo LS foi implementada e comparada com o  $LS_{BM}$ .

### 7.1 Algoritmo de escalonamento List Scheduling

O algoritmo de escalonamento implementado neste trabalho se baseia no algoritmo List Scheduling tradicional, *LS* [79]. O escalonamento utilizado neste trabalho é estático. Após a execução do algoritmo, um mapeamento das tarefas da aplicação no ambiente de execução é gerado. Este mapeamento é passado como parâmetro de execução e não é alterado durante a execução da aplicação.

O LS é dividido em duas fases: na primeira fase é atribuída uma prioridade a cada tarefa de acordo com um critério ou prioridade específica, representada na linha 1 do Algoritmo 11. Na segunda fase, inicialmente, é criada uma lista  $V_{livre}$  com todas as tarefas prontas para serem escalonadas ou livres. Uma tarefa livre, aqui denominada  $v_{livre}$ , é aquela que não possui predecessora no GAD ou que todas as suas predecessoras já foram escalonadas. No próximo passo do algoritmo, a tarefa  $v_{livre}$  de maior prioridade é atribuída a um elemento de processamento p de acordo com uma função de custo. A segunda fase do LS, ilustrada nas linhas 3-7 do Algoritmo 11, executa iterativamente até que todas as tarefas da aplicação sejam escalonadas. A cada iteração a lista  $V_{livre}$  é atualizada com as tarefas que se tornaram livres após a atribuição de  $v_{livre}$  a p, ou seja, após o escalonamento de  $v_{livre}$ . Esta operação é representada na linha 6 do Algoritmo 11. Vale lembrar que pred(v) e suc(v) é igual ao conjunto de predecessoras e de sucessoras de v, respectivamente.

Algorithm 11 List Scheduling - $LS(G(V, E))$
1: Calcule a prioridade $b_{level}(v_{livre})$ de cada $v \in V$ ;
2: $V_{livre} \leftarrow \{(v \in V) \mid pred(v) = \emptyset\};$
3: while $(V_{livre} \neq \emptyset)$ do
4: Selecione a tarefa $v_{livre} \in V_{livre}$ com maior prioridade e retire-a de $V_{livre}$ ;
5: Selectione $p \in P$ para $v_{livre}$ ;
6: $V_{livre} \leftarrow V_{livre} \cup \{ (v_i \in suc(v_{livre})) \mid v_i \in V \};$
7: end while

A prioridade atribuída a uma tarefa v no algoritmo LS é estática e é baseada no nível desta tarefa no grafo G. A prioridade de uma tarefa v no LS corresponde ao tamanho do maior caminho de v até o nó de saída de G, denominado  $b_{level}(v)$ . O  $b_{level}(v)$  é a maior soma de comunicação e peso de computação de uma tarefa v até o nó folha de G. O uso do  $b_{level}(v)$  como prioridade, estabelece a ordem de precedência das tarefas do grafo, já que um nó sempre possui  $b_{level}(v)$  menor do que o de sua predecessora. Assim, no LSimplementado, as tarefas de  $V_{livre}$  são ordenadas em ordem decrescente de  $b_{level}(v)$  [79].

Para que uma tarefa v inicie sua execução é necessário que a tarefa tenha recebido todas as mensagens de suas predecessoras. Logo, se uma tarefa u pertence ao conjunto de predecessores de  $v, u \in pred(v)$ , o tempo de início de v deve ser maior do que o tempo de fim de u mais o tempo gasto na transmissão da mensagem de u para v. Além disto, considerando que o termo disp(p) é definido como o tempo mais cedo em que o elemento de processamento p está livre para execução de uma nova tarefa, o tempo de início de uma tarefa v em p, ST(v, p), pode ser representado pela expressão:

$$ST(v,p) = \max\{disp(p), \max\{((FT(u,p) + \omega(u,v)) \mid u \in pred(v)\}\}$$

$$(7.1)$$

Considerando que  $e(\epsilon_v, p)$  é o tempo gasto para executar v em p, FT(v, p) define o tempo de fim de uma tarefa v em p e é dado por:

$$FT(v, p) = ST(v, p) + e(\epsilon_v, p)$$
(7.2)

Considerando que P é o conjunto processadores disponível no ambiente, o tempo total de execução da aplicação paralela é dado por:

$$makespan = \max_{v \in V, \ p \in P} \{FT(v, p)\}$$
(7.3)

A função objetivo do LS é encontrar um escalonamento para a aplicação que minimize o makespan. O processador p escolhido para escalonar  $v_{livre}$  é aquele onde a execução de  $v_{livre}$  termina "mais cedo" quando comparada com os outros processadores que compõem o ambiente de execução.

Neste trabalho, o elemento de processamento p é referente a um *core* do *cluster*. Logo, a partir daqui, a variável que identifica um *core* no *cluster*,  $C_{(i,j,k)}$ , será usada no lugar de p, como definido na representação do *cluster* de *multicore*, RCM, apresentado no Capítulo 4.

A Figura 7.2 ilustra o escalonamento gerado pelo algoritmo *LS* para o grafo da Figura 7.1 em um *cluster* composto por duas máquinas. Cada máquina possui dois processadores e cada um deles possui dois *cores*. Os *cores* de um mesmo processador compartilham uma memória cache L2 de capacidade igual a 2MB. A memória principal é dividida por todos os *cores* de uma mesma máquina.

Baseado no LS, a tarefa  $v_0$ , é inicialmente a única tarefa livre e, como nenhuma tarefa foi ainda atribuída aos cores,  $FT(v_0, C_{(i,j,k)})$  possui o mesmo valor para todos os cores do cluster. Logo,  $v_0$  será alocada no primeiro core da primeira máquina,  $C_{(0,0,0)}$ . Após a atribuição de  $v_0$  as tarefas  $v_1$ ,  $v_2$ ,  $v_3$ ,  $v_4$  e  $v_5$ , tornam-se livres. Como o LS classifica as tarefas livres em ordem decrescente do blevel, a lista de tarefas livres ordenadas será  $V_{livre} = (v_1, v_5, v_2, v_3, v_4)$ .

Ao escalonar a tarefa  $v_1$ , uma vez que a comunicação na mesma máquina é considerada igual a zero, o  $FT(v_1, C_{0,j,k}) \forall k \in P_{(0,j)} \in \forall j \in M_0$  será menor do que  $FT(v_1, C_{1,j,k}) \forall k \in$  $P_{(1,j)}, \forall j \in M_1$  já que  $\omega(v_0, v_1) = 1$ , se  $v_1$  for alocada em  $M_1$ . Assim,  $v_1$  será alocada ao core  $C_{(0,0,0)}$ . Quando  $v_1$  é escalonada  $v_6$  se torna livre e é inserida na lista de tarefas livres. A lista é reordenada e a próxima tarefa a ser escalonada é  $v_5$  seguida de  $v_6, v_2, v_3$ e  $v_4$ . A tarefa  $v_5$  é atribuída ao  $C_{(0,0,1)}$ , pois, é um core que oferece o menor tempo final para  $v_5$ . Já o menor tempo de fim para a tarefa  $v_6$  é igual a 3 se  $v_6$  for alocada a um dos cores de  $M_0$ , assim,  $v_6$  é atribuída ao core  $C_{(0,0,0)}$ . As tarefas  $v_2$  e  $v_3$  possuem menor tempo de fim iguais a 1, e são atribuídas, respectivamente, aos cores  $C_{(0,1,0)}$  e  $C_{(0,1,1)}$ .

Ao alocar  $v_4$ , seu tempo final será o mesmo se for atribuída ao core  $C_{(0,0,1)}$  ou a

qualquer outro *core* pertencente a  $M_1$ , uma vez que, em  $C_{(0,0,1)}$  e na máquina  $M_1$ ,  $v_4$ só poderá começar a executar na unidade de tempo igual a 2. O *LS* atribui então,  $v_4$ a  $C_{(0,0,1)}$ . A última tarefa escalonada é a tarefa  $v_7$ , que se torna livre depois de suas predecessoras  $v_6$  e  $v_5$  terem sido escalonadas. Esta última tarefa terá o menor tempo de fim se for alocada a qualquer um dos *cores* de  $M_0$ .



Figura 7.2: Escalonamento gerado pelo LS para o grafo G da Figura 7.1

Considerando o escalonamento apresentado na Figura 7.2 as tarefas  $v_1$ ,  $v_2$ ,  $v_3$ ,  $v_4$ e  $v_5$  irão executar concorrente. Considerando que a cache L2 tem capacidade igual a 2MB, durante a computação estas tarefas necessitarão utilizar mais memória do que a capacidade da cache associada aos *cores* onde foram alocadas. Diante disto, baseado no modelo de escalonamento proposto neste trabalho, o peso de computação dessas tarefas será acrescido de unidades de tempo relativas ao custo associado aos gargalos de acesso às memórias compartilhadas, neste caso tanto do acesso à memória cache quanto da memória principal. Deste modo, o tempo de fim da tarefa utilizado pelo algoritmo *LS* não representa o tempo real de execução das tarefas em ambientes que a contenção de memória é uma realidade. O desempenho desta aplicação poderia ser melhorado se as tarefas fossem melhor divididas entre as duas máquinas, já que em  $M_1$  nenhum *core* foi utilizado.

Na próxima seção é apresentado o algoritmo  $LS_{BM}$  que considera durante o escalonamento os gargalos de acesso à memória de acordo com os testes realizados no Capítulo 3 e descritos no modelo de escalonamento MEMC proposto no Capítulo 4. No  $LS_{BM}$  as tarefas são alocadas aos *cores* das máquina considerando a quantidade de memória  $\mu_v$  que a tarefa v utilizará em sua execução. Vale lembrar que, neste trabalhado,  $\mu_v$  também é referido como o tamanho da tarefa v.

# 7.2 Algoritmo List Scheduling Baseado em Memória, $LS_{BM}$

Baseado no modelo de escalonamento apresentado no Capítulo 4, foi proposto um algoritmo List Scheduling Baseado em Memória,  $LS_{BM}$ , que assim como o LS, atribui a cada tarefa v da aplicação uma prioridade. Entretanto, a escolha de um elemento de processamento a ser alocado a uma tarefa não é realizada de acordo com o tempo de fim, descrito na expressão 7.2, e sim, nas condições definidas no modelo, MECM. No  $LS_{BM}$ , as tarefas que requerem maior quantidade de memória para executar sua computação são escalonadas primeiro. Os cores são avaliados em função da quantidade de memória que será utilizada pelas tarefas alocadas aos cores que compartilham a cache a qual este core está associado.

O pseudocódigo do  $LS_{BM}$  é apresentado no Algoritmo 12 e as principais variáveis na Tabela 7.1, que tem como entrada um grafo do tipo GAD e tem como saída um mapeamento que define os *cores* onde as tarefas do grafos deverão ser executadas. O algoritmo  $LS_{BM}$  é detalhado ao longo do texto.

Como no LS, as tarefas livres são inseridas no conjunto  $V_{livre}$  cujos elementos são classificados em ordem decrescente de prioridade, linha 1 e 2 do Algoritmo 12. No entanto, no  $LS_{BM}$ , a prioridade atribuída a cada tarefa é igual a quantidade de memória requerida para executar a tarefa v, ou seja,  $\mu_v$ , como visto na Linha 4 do Algoritmo 12.

Quando uma tarefa  $v_{livre}$  é escalonada em um *core*  $C_{(i,j,k)}$ ,  $v_{livre}$  é inserida no conjunto de tarefas deste *core*, denominado  $CT_{(i,j,k)}$ . A escolha do *core* para alocar  $v_{livre}$  implica em analisar duas situações. Uma delas é referente à localização das tarefas predecessoras, para evitar o envio de mensagens via rede. Para tanto, para cada tarefa pertencente ao conjunto  $V_{livre}$ , é criado um conjunto de máquinas  $M_{pred}$  contendo o índice das máquinas onde as predecessoras de  $v_{livre}$  foram alocadas (linha 5 do Algoritmo 12). Este conjunto de máquinas é ordenado em ordem decrescente da quantidade de tarefas predecessores de  $v_{livre}$  em cada máquina.

A outra situação analisada envolve a escolha do *core* de acordo com as condições do modelo de escalonamento proposto. Assim, a escolha do *core* para alocar  $v_{livre}$  depende, também, da quantidade de memória disponível nas *caches* da máquinas do *cluster* no momento do escalonamento da tarefa. No Algoritmo  $LS_{BM}$  as linhas de 5 a 28 são referentes a escolha da unidade de processamento, que equivale a linha 5 do Algoritmo 11.

Para armazenar informações sobre a quantidade de memória requerida pelas tarefas alocadas a seus respectivos *cores*, foi definido que cada memória cache  $mc_{(i,j,l)}$  possui um peso  $PesoCache_{(i,j,l)}$  que, considerando que os *cores*  $C_{(i,j,k)}$  e  $C_{(i,j,k')}$  são vizinhos, é obtido pela expressão 7.4.

$$PesoCache_{(i,j,l)} = \sum_{t \in CT_{(i,j,k)}} \mu_t + \sum_{t' \in CT_{(i,j,k')}} \mu_{t'} + \mu_{v_{livre}}$$
(7.4)

Assim, uma tarefa  $v_{livre}$  é inserida no conjunto de tarefas  $CT_{(i,j,k)}$  de um *core*  $C_{(i,j,k)}$ , ou seja, é atribuída a  $C_{(i,j,k)}$  dependendo das seguintes condições:

Caso 1 - Existe core sem tarefas alocadas, ou vazio, em uma das máquinas de M<sub>pred</sub>: este caso é ilustrado na Figura 7.3, considerando que M<sub>pred</sub> = {M<sub>0</sub>}. Em M<sub>0</sub> existem dois cores livres, ou vazios, e um deles pode ser escolhido para alocar v<sub>livre</sub>.



Figura 7.3: Existe *core* sem tarefas em  $M_{pred} = \{M_0\}$ .

Este primeiro caso considera a Condição 1. do modelo, onde define que a soma da quantidade de memória das tarefas alocadas a dois *cores* vizinhos não deve ultrapassar a capacidade da cache associada a eles. Se esta condição for satisfeita, tarefas dependentes devem ser alocadas na mesma máquina para evitar o envio de mensagem pela rede. Neste caso, as contenções de acesso à memória são minimizadas.

Assim, é selecionado um core  $C_{(i,j,k)}$  que possua  $CT_{(i,j,k)} = \emptyset$ . O core escolhido deve ser aquele que não possui nenhuma tarefa atribuída para que seja evitada a concorrência pelo mesmo recurso de processamento. Além disto, o core selecionado deve ser aquele que está associado à cache com menor peso. No Algoritmo 12 é criado um conjunto de cores  $C_{pred}$  contendo todos os cores pertencentes às máquinas de  $M_{pred}$  com  $CT_{(i,j,k)} = \emptyset$  (linha 7). Os cores deste conjunto são analisados em ordem decrescente do número de precedessoras alocadas à máquina que ele pertence, ou seja, máquinas com maior número de tarefas predecessoras de  $v_{livre}$  alocadas são analisadas primeiro. Para cada core vazio, o peso da cache associada a ele é calculado e inserido em um conjunto *PesoCaches* na linha 24 do algoritmo. Logo, na linha 25, é identificada a cache que possui menor peso. Finalmente, nas linhas 27-29 a tarefa livre é atribuída ao core que possui menor quantidade de tarefas, que, neste caso, é igual a 0 já que foi selecionado um core que não possui tarefa atribuída.

• Caso 2 - Não existe core vazio em uma das máquinas de  $M_{pred}$ : considerando novamente que  $M_{pred} = \{M_0\}$ , este caso é ilustrado na Figura 7.4. Em  $M_0$  os quatro cores já possuem tarefas alocadas, mas  $M_1$  possui core sem tarefas alocadas, ou seja, vazio.



Figura 7.4: Todos os *core* de  $M_{pred} = \{M_0\}$  possuem tarefas alocadas.

Neste caso, apesar da Condição 1. sugerir que a tarefa seja alocada na mesma máquina que seu predecessor para evitar o envio de mensagens pela rede, existem *cores* livres em outra máquina do *cluster* que não pertence a  $M_{pred}$ . Com o objetivo de evitar o compartilhamento de recurso de processamento, as máquinas do *cluster* que não pertencem ao conjunto de máquinas predecessoras de  $v_{livre}$  são então analisadas. É criado o conjunto de *cores*  $C_{\{CM-M_{pred}\}}$  contendo os *cores* pertencentes a estas máquinas (linha 10 do Algoritmo 12) com  $CT_{(i,j,k)} = \emptyset$ . Assim como no caso anterior, o peso das caches pertencentes às máquinas do conjunto  $C_{\{CM-M_{pred}\}}$  são calculados e o *core* associado com a cache de menor peso é escolhido para alocar a tarefa livre. As linhas correspondentes a estas operações são as linhas 27-29, como no caso anterior.

• Caso 3 - Não existe core vazio nas máquinas do cluster: a Figura 7.5 ilustra este



caso onde todos os *cores* possuem pelo menos uma tarefa escalonada.

Figura 7.5: Todos os *core* de *CM* possuem tarefas alocadas.

Este caso ocorre quando os conjuntos  $C_{pred}$  e  $C_{\{CM-M_{pred}\}}$  são vazios, ou seja, não foi encontrado pelo menos um *core*  $C_{(i,j,k)}$  com  $CT_{(i,j,k)} = \emptyset$ . Uma vez que todos os *cores* possuem tarefas alocadas, analisa-se então somente os *cores* pertencentes às máquinas onde as tarefas predecessoras de  $v_{livre}$  estão alocadas, como no Caso 1.

Assim, como anteriormente, escolhe-se o *core* associado a cache com menor peso entre os *cores* pertencentes a uma das máquinas onde foram alocadas as predecessoras de  $v_{livre}$ ,  $M_{pred}$ . Para tanto, é calculado o peso de todas as *caches* pertencentes às máquinas do conjunto  $M_{pred}$  (linha 15) e a cache com menor peso é selecionada (linha 16). Vale notar que, ainda neste caso, a Condição 1. está sendo satisfeita.

 Caso 4 - Todas as caches pertencentes às máquinas de M<sub>pred</sub> estão com uso de memória acima de sua capacidade: a Figura 7.6 ilustra este caso, onde M<sub>pred</sub> = M<sub>0</sub>, as caches representadas com cores mais escuras possuem um peso maior que sua capacidade.



**Figura 7.6:** Todos os *core* de CM possuem tarefas alocadas e as caches de  $M_{pred}$  estão sobrecarregadas.

Neste caso, a Condição 1. do modelo não pode ser mais satisfeita quando consideradas as máquinas pertencente a  $M_{pred}$ , pois, para todas as *caches* pertencentes a estas máquinas,  $PesoCache_{(i,j,l)} > cmc_{(i,j,l)}$ , como verificado na linha 17 do algoritmo. Logo, são analisadas as demais máquinas do *cluster* e é escolhida a cache com menor peso. Esta decisão é baseada na Condição 2.a do modelo proposto, que define que quando as caches estão "sobrecarregadas" o tempo da aplicação é menor se for evitado o gargalo de memória. Logo, é mais interessante enviar a mensagem via rede do que alocar a tarefa na mesma máquina, excedendo ainda mais a capacidade das caches disponíveis. Vale ressaltar que a latência do envio de mensagens via rede possui custo menor do que o gargalo de acesso à memória local para uma determinada faixa de tamanho de mensagens. Nos ambientes onde os testes foram realizados, o tamanho da mensagem identificada foi menor do que 4MB. Nos experimentos realizados, para comunicações com mensagens maiores que esta faixa, o caso 4 é substituído pelo caso 3, de acordo com a Condição 2.b do modelo.

Vale notar que, quando as todas as máquinas atingem esta mesma condição, ou seja, todas as *caches* de todas as máquinas possuem peso maior que a sua capacidade a opção é escolher um *core* associado à cache com menor peso de uma máquina de  $M_{pred}$ .

As definições utilizadas pelo Algoritmo  $LS_{BM}$  são sumarizadas na Tabela 7.1.

A Figura 7.7 ilustra a execução do algoritmo  $LS_{BM}$  utilizando a mesma quantidade de máquinas e o mesmo grafo G da Figura 7.2. A tarefa  $v_0$  é, como no exemplo anterior, alocada no *core*  $C_{(0,0,0)}$ , entretanto, não pelo tempo final da tarefa  $v_0$  neste *core*, mas porque não possui nenhuma tarefa alocada. O mesmo é válido para os demais *cores* do *cluster*, assim, o *core* escolhido é o de menor índice.

Após a alocação de  $v_0$ , as tarefas que se tornaram livres são ordenadas de acordo com o tamanho de dados, sendo  $V_{livre} = (v_2, v_3, v_1, v_4, v_5)$ . A próxima tarefa a ser alocada,  $v_2$ , será atribuída ao core  $C_{(0,0,1)}$ . A próxima tarefa  $v_3$  será alocada ao core  $C_{(0,1,0)}$  e  $v_1$ alocada ao core  $C_{(0,1,1)}$ , pois são os cores vazios da máquina onde  $v_0$  foi alocada. Após a alocação de  $v_1$  a lista de tarefas livres é atualizada e reordenada, assim,  $V_{livre} = (v_6, v_4, v_5)$ . Com o objetivo de evitar a concorrência pelo mesmo recursos computacional, a tarefa  $v_6$ é alocada ao core  $C_{(1,0,0)}$ , pois, apesar de sua predecessora ter sido atribuída a um core da máquina  $M_0$ , esta máquina não possui mais cores vazios. Logo, as tarefas  $v_4$  e  $v_5$  são alocadas aos cores  $C_{(1,0,1)}$  e  $C_{(1,1,0)}$ , respectivamente. Quando  $v_5$  é alocada, a última tarefa  $v_7$  se torna livre e é alocada então core  $C_{(1,1,1)}$  da máquina  $M_1$ , onde suas predecessoras foram alocadas, cujo core ainda não havia sido alocado a nenhuma tarefa.

Comparando o escalonamento gerado pelo  $LS_{BM}$  apresentado na Figura 7.7 e pelo LS

Algorithm 12 List Scheduling Baseado em Memória -  $LS_{BM}(G(V, E))$ 

- 1: Calcule a prioridade  $\mu(v_{livre})$  de cada  $v \in V$ ;
- 2:  $V_{livre} \leftarrow \{(v \in V) \mid pred(v) = \emptyset\};$
- 3: while  $(V_{livre} \neq \emptyset)$  do
- 4: Selecione a tarefa  $v_{livre} \in V_{livre}$  com maior prioridade;
- 5:  $M_{pred} \leftarrow \{i \mid ((u \in CT_{(i,j,k)}) \in (u \in pred(v_{livre}))) \};$
- 6: /\* Cria conjunto C contendo todos os cores pertencentes às máquinas de  $M_{pred}$  \*/
- 7:  $C_{pred} \leftarrow \{C_{(i,j,k)} \mid CT_{(i,j,k)} = \emptyset, \forall k \in P_{(i,j)}, \forall j \in P_i, \forall i \in M_{pred}\};$
- 8: **if**  $(C_{pred} = \emptyset)$  **then**
- 9: /\* Cria conjunto C contendo todos os *cores* das máquinas do *cluster* que não pertencem a  $M_{pred}$  \*/
- 10:  $C_{\{CM-M_{pred}\}} \leftarrow \{C_{(i,j,k)} \mid CT_{(i,j,k)} = \emptyset, \forall k \in P_{(i,j)}, \forall j \in P_i, \forall i \in (CM M_{pred}) \};$ 11: end if
- 12: /\* Se todos os *cores* possuem tarefas alocadas \*/
- 13: if  $(C_{pred} = \emptyset)$  and  $(C_{\{CM-M_{pred}\}} = \emptyset)$  then
- 14: /\* Calcula o peso das caches L2 associada aos cores das máquinas de  $M_{pred}$  \*/
- 15:  $PesoCaches \leftarrow CalculaPesoCaches(C_{pred});$
- 16: Selectione  $mc_{(i,j,l)} \in PesoCaches$  com menor  $PesoCache_{(i,j,l)}$ ;
- 17: **if**  $(PesoCache_{(i,j,l)} > cmc_{(i,j,l)})$  **then**
- 18: /\* Calcula o peso das caches L2 pertencentes às máquinas do *cluster* que não pertence a  $M_{pred}^*/$
- 19:  $PesoCaches \leftarrow CalculaPesoCaches(C_{\{CM-M_{pred}\}});$
- 20: Selectione  $mc_{(i,j,l)} \in PesoCaches$  com menor  $PesoCache_{(i,j,l)}$ ;
- 21: end if
- 22: else
- 23: /\* Calcula o peso das caches L2 dos *cores* que ainda não possuem tarefas \*/
- 24:  $PesoCaches \leftarrow CalculaPesoCaches(C_{pred} \cup C_{\{CM-M_{pred}\}});$
- 25: Selectione  $mc_{(i,j,l)} \in PesoCaches$  com menor  $PesoCache_{(i,j,l)}$ ;
- 26: end if
- 27: Aloque  $v_{livre}$  a  $C_{(i,j,k)}$  associado a  $mc_{(i,j,l)}$  com menor quantidade de tarefas em  $CT_{(i,j,k)}$ ;
- 28:  $CT_{(i,j,k)} \leftarrow CT_{(i,j,k)} \cup v_{livre};$
- 29: Remova  $v_{livre}$  de  $V_{livre}$ ;
- 30:  $V_{livre} \leftarrow V_{livre} \cup \{ (v_i \in suc(v_{livre})) \mid v_i \in V \};$
- 31: end while

Símbolo	Definição
pred(v) =	Conjunto de tarefas que precedem a tarefa $v$ .
$\{u (u,v)\in E\}$	
suc(v) =	Conjunto de tarefas que sucedem a tarefa $v$ .
$\{w (v,w)\in E\}$	
$v_{livre}$	Tarefa cujas predecessores já foram escalonados.
$V_{livre}$	Conjunto da tarefas livres.
$\mu_{v_{livre}}$	Quantidade de memória utilizada pela tarefa livre du-
	rante a execução.
$M_{pred}$	Conjunto de máquinas onde estão alocadas as tarefas
	predecessoras da tarefa livre $v_{livre}$ .
$CT_{(i,j,k)}$	Conjunto de tarefas alocadas ao core $C_{(i,j,k)}$ .
$C_{pred}$	Conjunto de <i>cores</i> pertencentes às máquinas do conjunto
	$M_{pred}.$
$C_{\{CM-M_{pred}\}}$	Conjunto de <i>cores</i> pertencentes às máquinas do <i>cluster</i>
	que não pertençam às máquinas do conjunto $M_{pred}$ .
$cmc_{(n,j,l)}$	Capacidade da cache $mc_{(n,j,l)}$ compartilhada pelos cores
	$C_{(i,j,l)} \in C_{(i,j,k)}.$
PesoCaches	Conjunto de pesos de todas as <i>caches</i> associadas a um
	conjunto de <i>cores</i> .
$PesoCache_{(i,j,l)}$	É igual a soma da quantidade de memória de todas as
	tarefas alocadas aos <i>cores</i> associados à cache $mc_{(n,j,l)}$ .

**Tabela 7.1:** Resumo da definição das variáveis utilizados no  $LS_{BM}$ .

na Figura 7.2, pode-se perceber que os recursos de memória foram melhor distribuídos entre as tarefas pelo algoritmo  $LS_{BM}$ . Entretanto, este algoritmo envia, via rede, três mensagens, que são evitadas no exemplo de execução do LS. Segundo o modelo, como todas as caches da máquina exemplo estão sendo utilizadas em sua totalidade, o custo associado ao envio destas mensagens, desde que estas sejam pequenas é compensado pela menor disputa de acesso aos recursos de memória pelas tarefas.



**Figura 7.7:** Escalonamento gerado pelo  $LS_{BM}$  para o grafo G da Figura 7.1

### 7.3 Resultados e conclusões

Os algoritmos LS e  $LS_{BM}$  foram implementados e executados para uma aplicação sintética. Quatro tipos de grafos foram considerados: Outtree, Intree, Diamante e Irregular. A Figura 7.8 mostra instâncias para a visualização topológica destes. O grafo Diamante pode ser utilizado para modelar o problema de multiplicação de matrizes. Este tipo de grafo possui um único nó de entrada e saída. O grafo Intree possui vários nós de entrada e um nó de saída e pode ser utilizado para representar aplicações onde o número de operações é reduzido no decorrer da execução, como por exemplo, a operação de soma paralela. Já o grafo Outtree possui um nó de entrada e vários nós de saída e podem representar operações de difusão. O grafo Irregular não possui uma forma única e foi utilizado como exemplo de aplicações que não possuem uma característica bem definida de execução [26].

Cada tipo de grafo foi executado variando o número de tarefas e o tamanho da memória requerida por elas. O grafo Intree e Outtree foram executadas com 15, 31, 127, 255 e 511



Figura 7.8: Tipos de Grafos Acíclicos Direcionados de entrada

tarefas. Já o grafo Diamante para 36, 64, 225, 400 e 1024 tarefas, e para o Irregular 152, 185, 256 e 357 tarefas. Os tamanhos das mensagens utilizados no experimento foram de 1MB, 4MB e 8MB.

As Tabelas 7.2, 7.3, 7.4 e 7.5 apresentam, respectivamente, os resultados obtidos para a execução dos grafos Intree, Outtree, Diamante e Irregular. As colunas LS e  $LS_{BM}$ apresentam o tempo total de execução, em segundos, da aplicação quando utilizado o algoritmo LS e quando utilizado o  $LS_{BM}$ , respectivamente. A coluna %Redução apresenta a redução do tempo de execução da aplicação do  $LS_{BM}$  em relação ao LS. Os testes foram realizados em duas máquinas do ambiente definido no Capítulo 3.

Para as instâncias 15 e 31 do grafo Outtree (Tabela 7.2), o tempo de execução produzido pelo algoritmo  $LS_{BM}$  não reduziu o tempo de execução quando comparado ao LS. Devido à quantidade de tarefas destes grafos, que é pequena, e ao  $\mu_v$  associado a elas, o acesso aos recursos de memória não se tornou um gargalo. Por exemplo, para o Outtree com 31 tarefas, somente no último nível da árvore é que existe a possibilidade de todos os *cores* executarem as tarefas concorrentemente. Apesar de priorizar a alocação das tarefas nas máquinas de suas predecessoras, no  $LS_{BM}$  as tarefas tenderam a ser alocadas, inicialmente, aos *cores* os quais não foram atribuídas tarefas, fazendo com que as tarefas fossem alocadas em máquinas diferentes de suas predecessoras. No entanto, nestas instâncias, o custo de acesso às memórias compartilhadas foi muito pequeno e não compensou o custo associado aos envios de mensagens via rede. Vale perceber que o tempo de execução da aplicação escalonada segundo o  $LS_{BM}$  é ainda maior em relação ao tempo do LS para o grafo com 15 tarefas do que para o grafo de 31 tarefas, uma vez que neste grafo não acontece a concorrência de acesso à memória compartilhada. Além disto, o tempo do  $LS_{BM}$  piora à medida que o tamanho da mensagem enviada pelas tarefas aumenta.

Entretanto, para os grafos do tipo Outtree que possuem uma maior quantidade de tarefas, a redução de tempo obtida chegou a aproximadamente 54%, mostrando que evitar os gargalos de acesso à memória podem melhorar muito o desempenho da aplicação.

**Tabela 7.2:** Tempo de execução em segundos da execução do algoritmo  $LS_{BM}$  e LS para grafos Outtree. Cada tarefa tem o  $\mu_v$  associado gerado aleatoriamente entre 1MB e 10MB.

$\omega(u,v)$	LS	$LS_{BM}$	%Redução				
Outtree 15							
1MB	0,04619	0,06724	-31,31				
4MB	0,09871	0,20189	-51,11				
8MB	0,16983	$0,\!38392$	-55,76				
	Out	tree 31					
1MB	0,10022	0,11437	-12,37				
4MB	$0,\!28769$	$0,\!33801$	-14,89				
8MB	0,54233	$0,\!65303$	-16,95				
	Outtree 127						
1MB	0,50640	$0,\!35571$	29,76				
4MB	1,59246	0,97237	$38,\!94$				
8MB	3,04701	1,81199	$40,\!53$				
	Outt	ree 255					
1MB	1,08236	0,82200	$24,\!05$				
4MB	3,29810	2,16086	$34,\!48$				
8MB	$6,\!32676$	$3,\!98323$	$37,\!04$				
Outtree 511							
1MB	2,45443	1,50073	38,86				
4MB	$7,\!84678$	3,87497	$50,\!62$				
8MB	$15,\!20861$	6,97614	$54,\!13$				

Analogamente, nos grafos do tipo Intree, o acesso aos recursos de memória compar-

tilhada também pode se tornar um gargalo já que existe maior possibilidade das tarefas executarem concorrentemente. Neste caso, o  $LS_{BM}$  só não reduziu o tempo de execução do grafo com 15 tarefas. Para o Intree com 31 tarefas, inicialmente, 16 tarefas iniciam a execução de sua computação ao mesmo tempo, uma em cada *core*, compartilhando necessariamente recursos de memória principal, assim como no LS. O comportamento de execução neste caso é mais "síncrono" do que no caso do Outtree, pois todas as tarefas origens (independentes entre si) começarão ao mesmo tempo. Como no LS,  $\mu_v$  não é considerado, este algoritmo tende a escalonar em *cores* vizinhos duas tarefas  $v_i e v_j$  cuja soma de  $\mu_{v_i}$  com  $\mu_{v_j}$  é maior do que a capacidade da cache associada a este *core*.  $LS_{BM}$  tenta evitar esta situação, pois considera a capacidade da cache. Para as outras quantidades de tarefas o  $LS_{BM}$  reduziu o tempo de execução da aplicação até 82%.

A Figura 7.10 apresenta um exemplo do escalonamento realizado pelo LS e pelo  $LS_{BM}$ das 8 tarefas do primeiro nível de uma grafo Intree contendo 16 tarefas em um *cluster* (Figura 7.9), cujas caches L2 tem capacidade igual a 2MB. No LS, como as tarefas não possuem precedência, são escalonadas considerando seu índice, fazendo com que as caches da máquina  $M_0$  se tornem muito mais sobrecarregadas do que as da máquina  $M_1$ . Devido à contenção da cache e da memória principal, estas tarefas terão seu tempo de execução acrescido devido aos custos relacionados à contenção de memória. No  $LS_{BM}$  as tarefas que solicitam uma maior quantidade de memória para executar são alocadas a *cores* de diferentes caches, diminuindo o gargalo de acesso tanto à memória cache quanto à memória principal.



Figura 7.9: Exemplo de uma Intree com 16 tarefas.

Para o grafo Diamante, o  $LS_{BM}$  também reduziu o tempo de execução em relação ao LS. A redução foi menor para este grafo já que a concorrência pelo acesso de memória é menor, pois, devido à sua estrutura e a variação do tamanho dos dados nas instâncias geradas. Seja  $n_d$  o número de tarefas na diagonal principal do grafo diamante (coloridas



**Figura 7.10:** Exemplo de escalonamento do primeiro nível de um grafo Intree pelo (a) LS e (b)  $LS_{BM}$ .

no Figura 7.8). Devido ao  $\mu_v$  associado às tarefas predecessoras, parte das  $n_d$  tarefas, digamos  $n_c < n_d$ , podem ser executadas mais cedo e assim, uma maior quantidade de cache ficará disponível para as  $n_d - n_c$  tarefas restantes.  $LS_{BM}$  não considerando o tempo de execução e sim,  $\mu_v$ , tende a espalhar todas as  $n_d$  tarefas em *cores* de máquinas distintas, enquanto LS tende a colocar junto com as predecessoras.

Para o grafo Irregular, Tabela 7.5, o escalonamento gerado pelo  $LS_{BM}$  se mostrou melhor do que o LS para todas as instâncias.

A Tabela 7.6 apresenta o *makespan* que corresponde ao tempo teórico final de execução da aplicação obtido pelo escalonamento gerado pelo algoritmo *LS*, ou seja, é obtido pela Equação 7.6. Como discutido anteriormente, o tempo de execução real da tarefa é difícil de ser estimado *a priori*, já que este tempo depende da quantidade de tarefas que disputam pelos recursos de memória compartilhada.

O custo exato associado aos gargalos de acesso à memória são difíceis de prever, pois

$\omega(u,v)$	LS	$LS_{BM}$	%Redução			
1MB	0,06284	0,08101	-22,43			
4MB	$0,\!10712$	0,19214	-44,25			
8MB	$0,\!18015$	$0,\!34080$	$-47,\!14$			
	Inti	ree 31				
1MB	0,12042	0,07719	$35,\!90$			
4MB	$0,\!31706$	$0,\!17339$	$45,\!31$			
8MB	0,58302	$0,\!27746$	$52,\!41$			
Intree 127						
1MB	0,61532	0,25895	$57,\!92$			
4MB	2,13136	0,52257	$75,\!48$			
8MB	4,21778	0,93209	$77,\!90$			
	Intr	ee 255				
1MB	1,32013	0,51754	$60,\!80$			
4MB	4,48674	$0,\!85571$	$80,\!93$			
8MB	8,83435	1,52213	$82,\!77$			
Intree 511						
$1\overline{\mathrm{MB}}$	2,74436	1,02755	62,56			
4MB	10,01332	2,06021	$79,\!43$			
8 MB	19.84093	3.72902	81.21			

**Tabela 7.3:** Tempo de execução em segundos da execução do algoritmo  $LS_{BM}$  e LS para grafos Inttree.

dependem de vários fatores, como por exemplo, da criação das tarefas. Se houver atraso na criação de tarefas que executam concorrentemente pode acontecer que menos tarefas executam, de fato, concorrentemente e o tempo pode não ser tão alto quanto previsto. Ou o contrário, o atraso na criação de algumas tarefas pode acarretar na execução concorrente desta tarefa com outra tarefa que não foi previsto pelo algoritmo. Várias tentativas de definir o custo da influência dos gargalos de acesso à memória foram realizados, mas sempre ficaram muito longe do tempo real da aplicação. Comparando o *makespan* do escalonador gerado por LS com o tempo de execução deste escalonamento (7.6) pode-se perceber que o valor teórico não corresponde ao valor de execução real da aplicação. Os resultados da Tabela 7.6 são referentes ao tamanho de mensagem igual a 1MB.

Diante dos resultados obtidos, pode-se perceber que o algoritmo  $LS_{BM}$ , seguindo o MECM, apresenta resultados melhores do que a versão do LS implementada. Como no  $LS_{BM}$  os custos associados à contenção de memória são evitados, quanto maior a quantidade de tarefas disputando concorrentemente por este recurso, melhores são os resultados apresentados pelo algoritmo proposto. Vale notar que, este algoritmo ainda pode ser melhorado se considerasse melhor a relação entre quantidade de mensagens enviadas via rede

$\omega(u,v)$	LS	$LS_{BM}$	%Redução			
	Dian	nante 36				
1MB	0,17070	0,16288	$4,\!58$			
4MB	$0,\!37339$	0,38883	-3,97			
8MB	$0,\!67446$	$0,\!69884$	-3,49			
	Dian	nante 64				
1MB	0,29133	0,29803	-2,25			
4MB	$0,\!68108$	$0,\!87929$	-22,54			
8MB	$1,\!24879$	$1,\!63032$	-23,40			
Diamante 225						
1MB	1,00776	0,91806	8,90			
4MB	$3,\!08625$	$2,\!45197$	$20,\!55$			
8MB	$5,\!96417$	$4,\!49644$	$24,\!61$			
	Diam	ante 400				
1MB	1,46594	$1,\!24577$	$15,\!02$			
4MB	$4,\!35362$	$3,\!35850$	$22,\!86$			
8 MB	8,28104	6,22425	$24,\!84$			
Diamante 1024						
1MB	6,45586	3,60441	44,17			
4MB	$83,\!36784$	$38,\!53758$	$53,\!77$			
8MB	$217,\!55772$	155,84147	$28,\!37$			

**Tabela 7.4:** Tempo de execução em segundos da execução do algoritmo  $LS_{BM}$  e LS para grafos Diamante.

**Tabela 7.5:** Tempo de execução em segundos da execução do algoritmo  $LS_{BM}$  e LS para grafos Irregulares.

$\omega(u,v)$	LS	$LS_{BM}$	%Redução	
Irregular 152				
1MB	1,91770	1,86896	$2,\!54$	
4MB	$7,\!29618$	$7,\!16254$	$1,\!83$	
8MB	$14,\!43687$	$14,\!22712$	$1,\!45$	
Irregular 186				
1MB	$1,\!90005$	1,06212	$44,\!10$	
4MB	$5,\!98803$	$3,\!58123$	$40,\!19$	
8MB	$10,\!34557$	6,94390	$32,\!88$	
Irregular 256				
1MB	2,44994	1,51112	38,32	
4MB	7,70797	4,99256	$35,\!23$	
8MB	$13,\!45382$	9,79308	$27,\!21$	
Irregular 357				
1MB	11,50505	$5,\!63039$	$51,\!06$	
4MB	$67,\!48253$	$44,\!25213$	$34,\!42$	
8MB	$195,\!91485$	172,64068	$11,\!88$	

Instância	$Escalonamento \ LS$		
	Makespan	Tempo Real	
Intree 15	0,0249	0,04619	
Intree 31	$0,\!1019$	0,10022	
Intree 127	$0,\!2220$	0,50640	
Intree 255	$0,\!3161$	1,08236	
Intree 511	$0,\!4484$	$2,\!45443$	
Outtree 15	0,0296	0,06284	
Outtree 31	$0,\!0456$	0,12042	
Outtree 127	0,1164	$0,\!61532$	
Outtree 255	0,2190	1,32013	
Outtree 511	$0,\!3087$	2,74436	
Diamante 36	0,0924	$0,\!17070$	
Diamante 64	$0,\!1285$	0,29133	
Diamante 225	$0,\!2891$	1,00776	
Diamante 400	$0,\!4737$	1,46594	
Diamante 1024	1,0782	$6,\!45586$	
Irregular 152	0,3338	1,91770	
Irregular 186	$0,\!2073$	1,90005	
Irregular 256	0,2692	2,44994	
Irregular 357	0,3028	11,50505	

**Tabela 7.6:** Comparação do tempo de execução teórico, *Makespan*, com o tempo real obtido pela aplicação quando suas tarefas foram escalonamento com o LS

e a contenção de memória, explicado na Condição 2 do modelo.

## Capítulo 8

## Conclusões

Este trabalho propõe uma representação para *Cluster de Multicore* (RCM) e um modelo de escalonamento de tarefas que considera as características mais relevantes deste ambiente. O modelo proposto foi baseado em testes de uma aplicação sintética que foi desenvolvida com o objetivo de capturar os custos relativos aos gargalos de acesso aos recursos compartilhados de memória e os custos de comunicação inerentes ao uso de memória distribuída. Três níveis de comunicação foram identificadas nestes sistemas: i) a comunicação através de memória cache L2 compartilhada, que acontece entre as tarefas que são alocadas aos *cores* de um mesmo processador; ii) a comunicação, também utilizando memória compartilhada, mas entre tarefas alocadas a *cores* de processadores diferentes e iii) a comunicação entre tarefas alocadas a *cores* de máquinas diferentes.

Para avaliar o modelo foram propostos um algoritmo de escalonamento de tarefas  $(LS_{BM})$  baseado no *List Scheduling* e um *framework* de balanceamento de carga para um B&B paralelo aplicado ao Problema de Particionamento de Conjuntos,  $PBB_{SPP}$ . Tanto o escalonamento quanto os procedimentos de balanceamento de carga foram desenvolvidos considerando as condições definidas no modelo de escalonamento de tarefas, MEBM, e na representação dos *clusters* de *multicore*.

Os experimentos relativos ao algoritmo  $LS_{BM}$  foram realizados com uma aplicação sintética do tipo GAD. Os resultados obtidos quando as tarefas da aplicação são mapeadas nos *cores* utilizando o  $LS_{BM}$  foram comparados com um mapeamento gerado a partir da execução de um algoritmo LS tradicional. Estes resultados mostram que quando o escalonamento evita os gargalos de acesso à memória atribuindo tarefas de acordo com a capacidade da memória cache (L2) disponível para executá-las o tempo total de execução da aplicação diminuiu muito, principalmente para aplicações que possuem um maior número de tarefas executando concorrentemente, como no caso das instâncias Outtree.

A aplicação utilizada para testar o  $LS_{BM}$  é sintética e para avaliar o modelo de escalonamento proposto sob uma aplicação real uma versão paralela de um B&B foi implementada. O framework de balanceamento de carga deste B&B foi desenvolvido de acordo com as definições do modelo e seus procedimentos tentam garantir que os gargalos de memória sejam evitados através do gerenciamento da quantidade de memória utilizada por cada tarefa da aplicação. Além disto, os procedimentos do framework garantem uma ordem hierárquica de requisições de carga de modo que são priorizadas as comunicações entre tarefas alocadas a *cores* vizinhos, e, logo, entre as tarefas pertencente a uma mesma máquina para evitar a sobrecarga do envio de mensagens grandes via rede. O  $PBB_{SPP}$  foi comparado com uma versão paralela do B&B sem qualquer balanceamento de carga e com uma versão de balanceamento de carga para um B&B da literatura adaptada para resolver o mesmo problema. Os resultados mostram que o  $PBB_{SPP}$  apresenta tempo de execução muito menor quando comparado com os tempos de execução das demais versões. Além disto, o *PBB<sub>SPP</sub>* se mostrou escalável em relação ao aumento do número de máquinas no *cluster*. Vale lembrar que, a implementação dos procedimentos foi realizada utilizando funções de baixo nível que aceleram o acesso aos dados.

Apesar dos bons resultados obtidos pelas aplicações testadas, alguns parâmetros associados à sobrecarga de acesso aos recursos compartilhados poderiam ser associados ao modelo de escalonamento para que os algoritmos de escalonamento para aplicações que executam nestes ambiente apresentem resultados ainda melhores. Uma exemplo disto é que foi observado que existe um *tradeoff* entre o gargalo de acesso aos recursos de memória e a latência de envio de mensagem via rede, entretanto, os testes realizados não foram suficientes para definir este parâmetros, que poderiam ser utilizados para calcular com maior acurácia o *makespan* da aplicação. Este problema é conhecido na literatura, que considera a definição destes parâmetros, apesar de ser o ideal, muito difícil, visto que os ambientes de computação distribuídos e paralelos reais sofrem com a interferência de parâmetros externos imprevisíveis, como por exemplo, a ordem de execução das tarefas mesmo em um ambiente dedicado.

Além disto, os *clusters* de *multicore* já estão sendo formados por arquiteturas *multicore* heterogêneas. Neste caso, os *cores* possuem frequências de execuções diferentes. Além disto, cada *core* pode possuir uma especificidade de execução. Logo, o modelo deve ser adaptado para descrever esta nova característica, que influencia diretamente o custo de execução das tarefas e o custo associado ao acesso à hierarquia de memória.

## Referências

- AGGARWAL, A.; ALPERN, B.; CHANDRA, A.; SNIR, M. A model for hierarchical memory. In Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (New York, NY, USA, 1987), STOC '87, ACM, pp. 305–314.
- [2] AGGARWAL, A.; CHANDRA, A. K.; SNIR, M. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science* (1987), SFCS '87, Los Angeles, USA, pp. 204–216.
- [3] AGGARWAL, A.; CHANDRA, A. K.; SNIR, M. On communication latency in PRAM computations. In Proceedings of the First annual ACM Symposium on Parallel Algorithms and Architectures (New York, NY, USA, 1989), SPAA '89, ACM, pp. 11–21.
- [4] AGGARWAL, A.; CHANDRA, A. K.; SNIR, M. Communication complexity of PRAMs. Theor. Comput. Sci. 71, 1 (1990), 3–28.
- [5] ALAM, S. R.; BARRETT, R. F.; KUEHN, J. A.; ROTH, P. C.; VETTER, J. S. Characterization of scientific workloads on systems with multi-core processors. In Proceedings of the 2006 IEEE International Symposium on Workload Characterization, IISWC (San Jose, California, USA, 2006), IEEE, pp. 225–236.
- [6] ALEXANDROV, A.; IONESCU, M. F.; SCHAUSER, K. E.; SCHEIMAN, C. Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation. Tech. rep., Santa Barbara, CA, USA, 1995.
- [7] ALPERN, B.; CARTER, L.; FEIG, E.; SELKER, T. The uniform memory hierarchy model of computation. *Algorithmica 12* (1992), 12–2.
- [8] ALPERN, B.; CARTER, L.; FERRANTE, J. Modeling parallel computers as memory hierarchies. In *In Proc. Programming Models for Massively Parallel Computers* (1993), IEEE Computer Society Press, pp. 116–123.
- [9] ANDERSON, J. H.; CAL, J. M.; DEVI, U. C. Real-time scheduling on multicore platforms. In Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp (2006), Chapman Hall/CRC, Boca, pp. 179–190.
- [10] AUGONNET, C.; THIBAULT, S.; NAMYST, R.; WACRENIER, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.
- [11] BADIA, R. M.; PEREZ, J. M.; AYGUADE, E.; LABARTA, J. Impact of the memory hierarchy on shared memory architectures in multicore programming models. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 437–445.

- [12] BALAS, E.; PADBERG, M. W. Set partitioning: A survey. SIAM Review 18, 4 (1976).
- [13] BARBOSA, A. C.; CATABRIGA, L.; DE SOUZA, A. F.; P.VALLI, A. M. Análise do processamento paralelo em clusters multi-core na simulação de escoamento miscível implementado pelo método dos elementos finitos. In Sistemas Computacionais (WSCAD-SSC) (2009).
- [14] BARBOSA, V. C. An introduction to distributed algorithms. MIT Press, 1996.
- [15] BARRETO, L.; BAUER, M. Parallel branch and bound algorithm a comparison between serial, openmp and mpi implementations. *Journal of Physics: Conference Series 256*, 1 (2010), 012018.
- [16] BATOUKOV, R.; SØREVIK, T. A generic parallel branch and bound environment on a network of workstations. In In: Proceedings of HiPer'99 (1999), pp. 474–483.
- [17] BILARDI, G.; HERLEY, K. T.; PIETRACAPRINA, A.; PUCCI, G.; SPIRAKIS, P. G. BSP vs LogP. In Symposium on Parallel Algorithms and Architectures (1996), SPAA '96, pp. 25–32.
- [18] BOSCHETTI, M. A.; MINGOZZI, A.; RICCIARDELLI, S. A dual ascent procedure for the set partitioning problem. *Discret. Optim.* 5, 4 (Nov. 2008), 735–747.
- [19] BUDIU, M.; DELLING, D.; WERNECK, R. DryadOpt: Branch-and-bound on distributed data-parallel execution engines. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Anchorage, AK, May 16-20 2011).
- [20] BUTTARI, A.; DONGARRA, J.; KURZAK, J.; LANGOU, J.; TOMOV, S. The impact of multicore on math software. In Workshp on State-of-the-art in Scientific and Parallel Computing (Umea, Sweden, 2006), PARA '06.
- [21] BUTTARI, A.; LANGOU, J.; KURZAK, J.; DONGARRA, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (Jan. 2009), 38–53.
- [22] CAMERON, K. W.; GE, R.; SUN, X.-H. log<sub>n</sub> p and log<sub>3</sub> p: Accurate analytical models of point-to-point communication in distributed systems. *IEEE Transactions on Computers 56* (2007), 314–327.
- [23] CHAI, L.; GAO, Q.; PANDA, D. K. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Proceedings* of the Seventh IEEE International Symposium on Cluster Computing and the Grid (Washington, DC, USA, 2007), CCGRID '07, IEEE Computer Society, pp. 471–478.
- [24] COLE, R.; ZAJICEK, O. The APRAM: incorporating asynchrony into the PRAM model. In Proceedings of the first annual ACM symposium on Parallel algorithms and architectures (New York, NY, USA, 1989), SPAA '89, ACM, pp. 169–178.
- [25] CULLER, D. E.; KARP, R. M.; PATTERSON, D.; SAHAY, A.; SANTOS, E. E.; SCHAUSER, K. E.; SUBRAMONIAN, R.; VON EICKEN, T. LogP: a practical model of parallel computation. *Commun. ACM 39* (November 1996), 78–85.

- [26] DA SILVA, J. A. STATS: Uma ferramenta para escalonamento estático de tarefas em programas MPI. Dissertação de Mestrado, Universidade Federal Fluminense, 2002.
- [27] DE A. DRUMMOND, L. M.; UCHOA, E.; GONÇALVES, A. D.; SILVA, J. M. N.; SANTOS, M. C. P.; DE CASTRO, M. C. S. A grid-enabled distributed branchand-bound algorithm with application on the steiner problem in graphs. *Parallel Computing* 32, 9 (2006), 629–642.
- [28] DE ALMEIDA SAMPAIO BRAGA, A. Relaxações lagrangianas e planos de corte faciais na resolução de problemas de particionamento de conjuntos. Dissertação de Mestrado, Universidade Estadual de Campinas. Instituto de Computação, 2011.
- [29] DE AMORIM MENDES, H. HLogP : Um modelo de escalonamento para execução de aplicações MPI em grades computacionais. Dissertação de Mestrado, IC, Universidade Federal Fluminense, 2004.
- [30] DE COMPUTADORES: UMA ABORDAGEM QUANTITATIVA, A. John L. Hennessy and David A. Patterson, 1st ed. Editora Campus, São Paulo, SP, Brasil, 2003.
- [31] DE PAULA NASCIMENTO, A. Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais Processamento Paralelo e Distribuído. Tese de Doutorado, Universidade Federal Fluminense, 2008.
- [32] DJERRAH, A.; CUN, B. L.; CUNG, V.-D.; ROUCAIROL, C. Bob++: Framework for solving optimization problems with branch-and-bound methods. In *HPDC* (2006), pp. 369–370.
- [33] DONGARRA, J.; MOORE, S.; MUCCI, P.; SEYMOUR, K.; YOU, H. Accurate cache and TLB characterization using hardware counters. In *International Conference on Computational Science* (Krakow, Poland, 2004).
- [34] E SILVA, J. M. N. Estratégias de balanceamento de carga para um algoritmo branchandbound paralelo para executar em grids computacionais. Dissertação de Mestrado, Universidade Federal Fluminense, 2006.
- [35] ECKSTEIN, J.; PHILLIPS, C. A.; HART, W. E. Pico: An object-oriented framework for parallel branch and bound. *Studies in Computational Mathematics* 8 (2001), 219– 265.
- [36] FEDOROVA, A.; SELTZER, M.; SMALL, C.; NUSSBAUM, D. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 26–26.
- [37] FEDOROVA, D. V. A.; DOUCETTE, D. Operating system scheduling on heterogeneous core systems. In First Workshop on Operating System Support for Heterogeneous Multicore Architectures (2007).
- [38] FORTUNE, S.; WYLLIE, J. Parallelism in random access machine. In 10th ACM Symposium on Theory of Computation (STOC) (New York, USA, May 1978), pp. 114– 118.

- [39] FRANK, M. I.; AGARWAL, A.; VERNON, M. K. LoPC: Modeling contention in parallel algorithms. In Proceedings of the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, June 18–21, 1997 (1997), ACM.
- [40] GALEA, F.; CUN, B. L. A parallel exact solver for the three-index quadratic assignment problem. In *IPDPS Workshops* (2011), pp. 1940–1949.
- [41] GIBBONS, P. B. A more practical PRAM model. In Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures (New York, NY, USA, 1989), SPAA '89, ACM, pp. 158–168.
- [42] GIBBONS, P. B.; MATIAS, Y.; RAMACHANDRAN, V. The QRQW PRAM: accounting for contention in parallel algorithms. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1994), SODA '94, Society for Industrial and Applied Mathematics, pp. 638–648.
- [43] GIBBONS, P. B.; MATIAS, Y.; RAMACHANDRAN, V. Can shared-memory model serve as a bridging model for parallel computation? In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 1997), SPAA '97, ACM, pp. 72–83.
- [44] GONZÁLEZ-DOMÍNGUEZ, J.; TABOADA, G. L.; FRAGUELA, B. B.; MARTÍN, M. J.; TOURIÑO, J. Servet: A benchmark suite for autotuning on multicore clusters. In *IPDPS'10* (2010), pp. 1–9.
- [45] GRAHAM, R. L.; LAWLER, E. L.; LENSTRA, J. K.; KAN, R. Optimization and approximation in deterministic sequencing and scheduling a survey. *Discrete Mathematics* (1978).
- [46] GROUP, T. O. The open group base specifications issue 6 @ONLINE, June 2004.
- [47] HOCHBAUM, D. S. Approximation algorithms for NP-hard problems. PWS Publishing Co., Boston, MA, USA, 1997.
- [48] HOGG, J. D.; REID, J. K.; SCOTT, J. A. Design of a multicore sparse cholesky factorization using DAGs. SIAM J. Sci. Comput. 32, 6 (Dec. 2010), 3627–3649.
- [49] INNOVATING COMPUTING LABORATORY, UNIVERSITY OF TENNESSEE. Performance application programming interface, 2004. http://icl.cs.utk.edu/papi/.
- [50] INO, F.; FUJIMOTO, N.; HAGIHARA, K. LogGPS: a parallel computational model for synchronization analysis. SIGPLAN Not. 36 (June 2001), 133–142.
- [51] JAJÁ, J. An introduction to parallel algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.
- [52] JUURLINK, B.; JUURLINK, B. H. H.; WIJSHOFF, H. A. G. The parallel hierarchical memory model. In In Proc. Scandinavian Workshop on Algorithms Theory, LNCS 824 (1994), Springer-Verlag, pp. 240–251.
- [53] KAYI, A.; EL-GHAZAWI, T. A.; NEWBY, G. B. Performance issues in emerging homogeneous multi-core architectures. *Simulation Modelling Practice and Theory* 17, 9 (2009), 1485–1499.
- [54] KIRK, D. B.; HWU, W.-M. W. Programming Massively Parallel Processors: A Hands-on Approach, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.
- [55] LI, Z.; MILLS, P. H.; REIF, J. H. Models and resource metrics for parallel and distributed computation. In *Proceedings of the 28th Hawaii International Conference* on System Sciences (Washington, DC, USA, 1995), IEEE Computer Society, pp. 51–.
- [56] LIU, G. Q.; POH, K. L.; XIE, M. Iterative list scheduling for heterogeneous computing. J. Parallel Distrib. Comput. 65 (May 2005), 654–665.
- [57] MA, K.-L. Parallel volume ray-casting for unstructured-grid data on distributedmemory architectures. In *Proceedings of the IEEE Symposium on Parallel Rendering* (New York, NY, USA, 1995), PRS '95, ACM, pp. 23–30.
- [58] MAGGS, B. M.; MATHESON, L. R.; TARJAN, R. E. Models of parallel computation: A survey and synthesis, 1995.
- [59] MARS, J.; TANG, L.; SOFFA, M. L. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference* on High Performance and Embedded Architectures and Compilers (Vienna, Austria, 2011), HiPEAC '11, ACM, pp. 167–176.
- [60] MARTY, M. R. Cache Coherence Techniques for multicore processors. Tese de Doutorado, Madison, WI, USA, 2008.
- [61] MEHLHORN, K.; VISHKIN, U. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. Acta Inf. 21 (1984), 339–374.
- [62] MERCIER, G.; CLET-ORTEGA, J. Towards an efficient process placement policy for MPI applications in multicore environments. In *Lecture Notes in Computer Science* (Espoo, Finland, Sept. 2009), vol. 5759 of *EuroPVM/MPI*, Springer, pp. 104–115.
- [63] MOORE, R. W.; CHILDERS, B. R. Using utility prediction models to dynamically choose program thread counts. In *ISPASS* (2012), IEEE, pp. 135–144.
- [64] PAPADIMITRIOU, C.; YANNAKAKIS, M. Towards an architecture-independent analysis of parallel algorithms. In STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing (New York, NY, USA, 1988), ACM, pp. 510–513.
- [65] PARK, S.; KIM, T.; PARK, J.; KIM, J.; IM, H. Parallel skyline computation on multicore architectures. In *Proceedings of the 2009 IEEE International Conference* on Data Engineering (Washington, DC, USA, 2009), ICDE '09, IEEE Computer Society, pp. 760–771.
- [66] PETRUCCI, V. T. Optimization of power and performance for heterogeneous server systems Redes de Computadores e Sistemas Distribuídos e Paralelos. Tese de Doutorado, Universidade Federal Fluminense, 2012.
- [67] RALPHS, T. K.; GÜZELSOY, M.; MAHAJAN, A. SYMPHONY version 5.3 user's manual. Tech. rep., COR@L Laboratory, Lehigh University, 2011.

- [68] RAMACHANDRAN, V. QSM: A general purpose shared-memory model for parallel computation. In Foundations of Software Technology and Theoretical Computer Science (1997), pp. 1–5.
- [69] RAMACHANDRAN, V.; GRAYSON, B.; DAHLIN, M. Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design. J. Parallel Distrib. Comput. 63, 12 (2003), 1175–1192.
- [70] RASHID, H.; NOVOA, C.; QASEM, A. An evaluation of parallel knapsack algorithms on multicore architectures. In *International Conference on Scientific Computing* (2010), CSC'10, pp. 230–235.
- [71] RIOS, E. F. S. Heurísticas híbridas para escalonamento estático de tarefas em sistemas com processadores heterogêneos. Dissertação de Mestrado, Universidade Federal Fluminense, 2004.
- [72] SANJUAN-ESTRADA, J. F.; CASADO, L. G.; GARCÍA, I. Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures. *The Journal of Supercomputing* (2011), 376–384.
- [73] SAVAGE, J. E. Models of Computation: Exploring the Power of Computing, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [74] SAVAGE, J. E.; ZUBAIR, M. A unified model for multicore architectures. In Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies (New York, NY, USA, 2008), IFMT '08, ACM, pp. 9:1–9:12.
- [75] SAVAGE, J. E.; ZUBAIR, M. Evaluating multicore algorithms on the unified memory model. Sci. Program. 17 (December 2009), 295–308.
- [76] SENA, A. Um modelo alternativo para execução eficiente de aplicações paralelas MPI nas Grades Computacionais. Tese de Doutorado, Universidade Federal Fluminense, 2008.
- [77] SHINANO, Y.; HIGAKI, M.; HIRABAYASHI, R. A generalized utility for parallel branch and bound algorithms. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing* (Washington, DC, USA, 1995), SPDP '95, IEEE Computer Society, pp. 392–.
- [78] SILVA, J. M. N.; BOERES, C.; DRUMMOND, L. M. A.; PESSOA, A. A. Memory aware load balance strategy on a parallel branch-and-bound application. Tech. rep., COR@L Laboratory, Lehigh University, 2013.
- [79] SINNEN, O. Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing). Wiley-Interscience, 2007.
- [80] SMITH, A. J.; SAAVEDRA, R. H. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.* 44, 10 (Oct. 1995), 1223–1235.
- [81] SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. MPI -The Complete Reference, Volume 1: The MPI Core. MIT Press, Cambridge, MA, USA, 1998.

- [82] SONG, F.; MOORE, S.; DONGARRA, J. Analytical modeling for affinity-based thread scheduling on multicore plataforms. In *Symposim on Principles and Practice of Parallel Programming* (2009).
- [83] SONG, F.; YARKHAN, A.; DONGARRA, J. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *International Conference for High Performance Computing, Networking Storage and Analysis,* (2009).
- [84] STEVENS, R. W.; RAGO, S. A. Advanced Programming in the UNIX(R) Environment (2nd Edition). Addison-Wesley Professional, 2005.
- [85] TAM, A. T.; WANG, C.-L. Realistic communication model for parallel computing on cluster. In 1st IEEE International Workshop on Cluster Computing (1999).
- [86] TANG, L.; MARS, J.; SOFFA, M. L. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (New York, NY, USA, 2011), EXADAPT '11, ACM, pp. 12–21.
- [87] TANG, X.; LI, K.; LIAO, G.; LI, R. List scheduling with duplication for heterogeneous computing systems. *Journal of Parallel and Distributed Computing* 70 (2010), 323–329.
- [88] TIAN, K.; JIANG, Y.; SHEN, X. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of the 6th ACM conference on Computing frontiers* (New York, NY, USA, 2009), CF '09, ACM, pp. 41–50.
- [89] TU, B.; FAN, J.; ZHAN, J.; ZHAO, X. Accurate analytical models for message passing on multi-core clusters. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 133–139.
- [90] VALIANT, L. G. A bridging model for parallel computation. Commun. ACM 33, 8 (1990), 103–111.
- [91] VITTER, S. J.; M SHRIVER., A. E. Algorithms for parallel memory II: Hierarchical multilevel memories. Tech. rep., Durham, NC, USA, 1993.
- [92] VU, T.-T.; DERBEL, B.; ASIM, A.; BENDJOUDI, A.; MELAB, N. Overlay-centric load balancing: Applications to uts and b&b. In *CLUSTER* (2012), pp. 382–390.
- [93] WILLIAMS, T. L.; PARSONS, R. J. The heterogeneous bulk synchronous parallel model. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, UK, 2000), IPDPS '00, Springer-Verlag, pp. 102–108.
- [94] XIA, Y.; PRASANNA, V. K. Collaborative scheduling of DAG structured computations on multicore processors. In *Proceedings of the 7th ACM international conference* on Computing frontiers (New York, NY, USA, 2010), CF '10, ACM, pp. 63–72.
- [95] XIA, Y.; PRASANNA, V. K.; LI, J. Hierarchical scheduling of DAG structured computations on manycore processors with dynamic thread grouping. In *Proceedings* of the 15th international conference on Job scheduling strategies for parallel processing (Berlin, Heidelberg, 2010), JSSPP'10, Springer-Verlag, pp. 154–174.

[96] ZENG, X.; SODAN, A. C. Job scheduling strategies for parallel processing. Springer-Verlag, Berlin, Heidelberg, 2009, ch. Job Scheduling with Lookahead Group Matchmaking for Time/Space Sharing on Multi-core Parallel Machines, pp. 232–258.