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

GUILHERME PERETTI PEZZI

Escalonamento *Work-Stealing* de programas Divisão-e-Conquista com MPI-2

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Philippe Olivier Alexandre Navaux
Orientador

Prof. Nicolas Maillard
Co-orientador

Porto Alegre, dezembro de 2006

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pezzi, Guilherme Peretti

Escalonamento *Work-Stealing* de programas Divisão-e-Conquista com MPI-2 / Guilherme Peretti Pezzi. – Porto Alegre: PPGC da UFRGS, 2006.

69 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2006. Orientador: Philippe Olivier Alexandre Navaux; Co-orientador: Nicolas Maillard.

1. Programação paralela. 2. Ambiente de programação. 3. Divisão-e-conquista. 4. Roubo de tarefas. 5. MPI-2. I. Navaux, Philippe Olivier Alexandre. II. Maillard, Nicolas. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Prof[a]. Valquíria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

MPI	Message Passing Interface
HPC	High Performance Computing
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
SMP	Symmetric Multi-Processor
D&C	Divisão-e-Conquista
DAG	Directed Acyclic Graph
B&B	Branch-and-Bound
WAN	Wide Area Network
NOW	Network of Workstations
API	Application Programming Interface
BSP	Binary Space Partitioning
MPP	Massively Parallel Processor
TSP	Traveling Salesman Problem

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Com o objetivo de ser portável e eficiente em arquiteturas HPC atuais, a execução de um programa paralelo deve ser adaptável. Este trabalho mostra como isso pode ser atingido utilizando MPI, através de criação dinâmica de processos, integrada com programação Divisão-e-Conquista e uma estratégia *Work-Stealing* para balancear os processos MPI, em ambientes heterogêneos e/ou dinâmicos, em tempo de execução. Este trabalho explica como implementar uma aplicação segundo o modelo de Divisão-e-Conquista com MPI, bem como a implementação de uma estratégia *Work-Stealing*. São apresentados resultados experimentais baseados em uma aplicação sintética, o problema das *N-Rainhas* (*N-Queens*). Valida-se tanto a adaptabilidade e a eficiência do código. Os resultados mostram que é possível utilizar um padrão amplamente difundido como o MPI, mesmo em plataformas de HPC não tão homogêneas como um *cluster*.

Palavras-chave: Programação paralela, ambiente de programação, divisão-e-conquista, roubo de tarefas, MPI-2.

Scheduling Divide-and-Conquer programs by *Work-Stealing* with MPI-2

ABSTRACT

In order to be portable and efficient on modern HPC architectures, the execution of a parallel program must be adaptable. This work shows how to achieve this in MPI, by the dynamic creation of processes, coupled with Divide-and-Conquer programming and a *Work-Stealing* strategy to balance the MPI processes, in a heterogeneous and/or dynamic environment, at runtime. The application of Divide and Conquer with MPI is explained, as well as the implementation of a *Work-Stealing* strategy. Experimental results are provided, based on a synthetic application, the *N-Queens* computation. Both the adaptability of the code and its efficiency are validated. The results show that it is possible to use widely spread standards such as MPI, even in parallel HPC platforms that are not as homogeneous as a Cluster.

Keywords: parallel programming, programming environments, divide-and-conquer, work-stealing, MPI-2.

1 INTRODUÇÃO

A computação paralela vem sendo amplamente utilizada para atender necessidades de aplicações que exigem muito poder computacional. A cada dia surgem propostas de novas máquinas e modelos de arquiteturas paralelas, com diferentes quantidades de processadores, topologias e redes de interconexão. O universo de arquiteturas existentes é grande e elas são normalmente agrupadas como: SIMD e MIMD. Máquinas SIMD possuem um único fluxo de instrução aplicados a múltiplos dados e normalmente são específicas para um tipo de aplicação. As MIMD possuem múltiplos fluxos de instrução sendo executados em paralelo. Atualmente as máquinas MIMD são as mais utilizadas e podem ser divididas como: Multi Processadores Simétricos (SMP), Vetoriais, Massivamente paralelas (MPP), agregados de computadores (*Clusters*) e Grades computacionais (*Grids*). As máquinas SMP, Vetoriais e MPP normalmente fornecem ambientes para programação e uso eficiente, porém são arquiteturas proprietárias e com alto custo de montagem e manutenção. Já os *clusters* e os *grids* consistem basicamente de diversas unidades de processamento, que podem ser independentes, interconectadas por uma rede de comunicação. As principais vantagens nesse modelo são custos baixos e alta escalabilidade, isto é, a possibilidade de facilmente alterar, ao longo do tempo, a quantidade de unidades de processamento da arquitetura. No entanto, a computação é baseada em memória distribuída e necessita programação da troca de informações entre os processadores, o que pode ser uma tarefa muito complexa.

Para que um ambiente de programação seja adequado a essas máquinas, espera-se que ele forneça primitivas para expressão do paralelismo. Espera-se também que seja eficiente e aproveite ao máximo os recursos disponíveis, mesmo que essa disponibilidade se altere dinamicamente. Dada a heterogeneidade das arquiteturas, é interessante que o ambiente permita criar uma aplicação que seja portável entre elas. Enfim, pode-se enumerar diversas características que tornam um ambiente de programação paralela adequado para as diferentes arquiteturas encontradas atualmente e espera-se que um ambiente agregue a maior quantidade dessas características.

Para facilitar a programação e permitir o uso eficiente dessas máquinas, podem ser utilizados diferentes modelos de programação. Existem diversas técnicas de programação para esses modelos e pode-se encontrar algumas delas de forma mais detalhada em (WILKINSON; ALLEN, 1999). O modelo *Bag of Tasks* baseia-se na execução concorrente de tarefas disjuntas, ou seja, quando não há necessidade de comunicação durante a execução de cada tarefa. No modelo *Pipeline* as tarefas são divididas em partes e cada processador executa uma parte. Os dados de entrada são colocados no primeiro processador e repassados entre os processadores, após a execução de cada parte. No modelo *Fases Paralelas* (BSP) as aplicações executam

um determinado número de etapas, divididas em duas fases: processamento e sincronização. O modelo *Mestre/Escravo* centraliza a distribuição e controle da execução do trabalho em um único processador. É indicado para problemas cuja interação entre os processos não seja grande e não cause sobrecarga no processo mestre. No modelo de Divisão e Conquista (*D&C*) o problema é dividido em problemas menores, que são distribuídos entre os processadores disponíveis e o seu resultado é combinado para obter a resposta do problema original. Nesse modelo os processos podem se comunicar ao longo da execução, sem passar pelo mestre.

Aplicações trivialmente paralelizáveis e sem dependência de dados entre as tarefas normalmente podem ser paralelizadas utilizando o modelo *Bag of Tasks*. Esse modelo é adequado, por exemplo, para execução de filtros gráficos em uma grande quantidade de imagens de pequeno porte, para verificar primalidade de números (criptografia), etc. Para o modelo *Mestre/Escravo*, indica-se aplicações como geração de fractais e imagens (*Ray-Tracing*), pois sua resolução pode ser controlada por um processo mestre que envia as informações necessárias para os processos escravos e junta os resultados parciais. O modelo de *D&C* é mais abrangente, pois engloba o modelo *Mestre/Escravo*, e pode ser utilizado para implementação de inúmeras operações que envolvem matrizes e também aplicações recursivas, entre outras.

O modelo *D&C* pode ser utilizado para uma quantidade maior de aplicações, comparando-se com *Bag of Tasks* e *Mestre/Escravo*, e oferece a possibilidade de escalonamento e criação dinâmica das tarefas. Por esse motivo, torna-se interessante estudá-lo, bem como os ambientes que suportam esse modelo. Neste trabalho foi feito um levantamento dos ambientes que suportam *D&C* e, a partir disso, uma análise crítica e comparativa entre as características de cada ambiente. Verificou-se que os principais ambientes que focam aplicações *D&C* oferecem criação dinâmica de processos e o MPI (*Message Passing Interface*), que é um padrão conhecido e amplamente utilizado para programação paralela, implementou recentemente esse mecanismo.

Dado o contexto, este trabalho propõe um mecanismo de escalonamento distribuído de processos, utilizando-se como base os conceitos encontrados nos ambientes *D&C* estudados. O mecanismo proposto baseia-se no conceito de roubo de tarefas (*Work-Stealing*) e propõe-se utilizá-lo para balanceamento de carga em ambientes com heterogeneidade de processamento e interconexão de rede, bem como em *clusters* homogêneos. Para validação do desempenho, implementa-se uma aplicação sintética (*N-Queens*) e compara-se com uma implementação de um ambiente já consolidado, o Satin (NIEUWPOORT; KIELMANN; BAL, 2000). Além disso, mostra-se que o mecanismo proposto é adaptado também para o aproveitamento de recursos disponibilizados dinamicamente ao longo da execução de uma aplicação.

As principais contribuições deste trabalho são: uma modelagem para programar aplicações *D&C* utilizando MPI-2, que está detalhada neste texto e em (PEZZI et al., 2006); um mecanismo de *Work-Stealing* com controle do paralelismo da aplicação *D&C*; validação da eficiência da implementação de *Work-Stealing* com MPI-2 e comparação entre o escalonamento *Work-Stealing* proposto e escalonamento *Round-Robin*. Detalhes sobre o mecanismo *Round-Robin* utilizado podem ser encontrados em (CERA et al., 2006) e (CERA et al., 2006).

Este texto está dividido da seguinte forma: o Capítulo 2 apresenta brevemente o modelo *D&C* utilizando exemplos e contextualiza os principais ambientes que suportam o modelo. No Capítulo 3, detalha-se o modelo de aplicação paralela

tratado e o suporte disponível no MPI-2 para esse tipo de aplicações. No Capítulo 4, estudam-se as primitivas básicas de escalonamento oferecidas e apresentam-se dois modelos de escalonamento: *Work-Pushing* e *Work-Stealing*. O Capítulo 5 mostra duas implementações de roubo de tarefas: no contexto de *cluster* de computadores e de *cluster* de *clusters*. No Capítulo 6, apresentam-se resultados obtidos na validação das implementações: comparam-se diferentes implementações de *Work-Stealing* em MPI-2 e comparam-se os resultados com um ambiente já consolidado para aplicações *D&C*, o Satin. Para concluir, são feitas análises e considerações sobre o estudo realizado no Capítulo 7.

2 DIVISÃO-E-CONQUISTA COM CRIAÇÃO DINÂMICA DE PROCESSOS

Este capítulo apresenta e compara os principais ambientes que suportam a programação paralela de aplicações *D&C* com criação dinâmica de processos. Inicialmente, é feita uma introdução ao modelo de Divisão-e-Conquista (Seção 2.1). Após, a Seção 2.2 faz um levantamento dos ambientes encontrados na literatura. Por fim, é feita uma comparação entre as características de cada ambiente (Seção 2.3), seguida de algumas considerações sobre o estudo realizado (Seção 2.4).

2.1 O modelo de Divisão-e-Conquista

O modelo *D&C* (CORMEN et al., 2002) baseia-se na divisão de um problema em partes menores, que podem ser resolvidas separadamente. A divisão é feita até que o sub-problema seja simples e sua solução imediata. Suas soluções parciais são então combinadas para se obter a solução da instância original do problema. Uma definição recursiva para um dado de entrada e pode ser escrita como:

$$\text{Solução}(e): \begin{cases} \text{se simples}(e) \text{ então direto}(e) \\ \text{senão combina(} \text{ solução}(\text{parte}_1(e)), \text{solução}(\text{parte}_2(e)) \text{)} \end{cases}$$

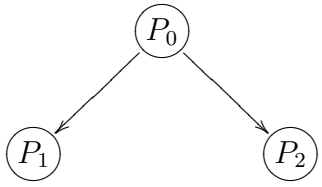

Figura 2.1: Aplicação *D&C* vista como um DAG.

Uma forma comum de representar aplicações *D&C* é através de grafos acíclicos dirigidos (KWOK; AHMAD, 1999), conhecidos como *Directed Acyclic Graphs* (ou DAG). Em um DAG, os vértices e arestas representam, respectivamente, os processos e as dependências[1] entre eles. A Figura 2.1 mostra um exemplo de aplicação *D&C* vista como um DAG: a execução de P_0 depende de dados que serão calculados por P_1 e P_2. Vale ressaltar que os vértices não representam processadores, e sim processos, que podem ou não estar no mesmo processador.

[1] A rigor, representa a redução transitiva da relação de dependência.

2.1.1 Divisão-e-Conquista paralela

Além da técnica de Divisão-e-Conquista ser classicamente usada em programação seqüencial para prover algoritmos altamente eficientes (CORMEN et al., 2002), ela pode também ser interessante para a programação paralela: resultados teóricos mostram que ela permite obter algoritmos paralelos eficientes. Em nível prático, vários ambientes de programação usaram este modelo, com excelentes resultados: Cilk (BLUMOFE; LEISERSON, 1998), Satin (NIEUWPOORT et al., 2006) e Athas-pascan (CAVALHEIRO; GALILÉE; ROCH, 1998). Além disso, foi comprovado e testado experimentalmente que se pode escalonar eficientemente os processos de um programa paralelo *D&C* (BLUMOFE; LEISERSON, 1994).

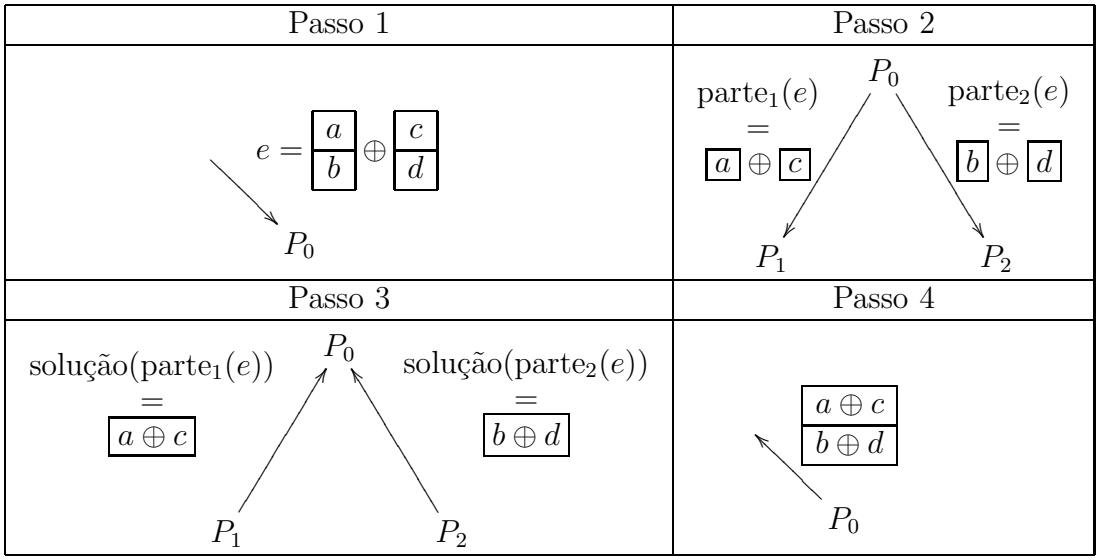

Figura 2.2: Execução de uma aplicação do tipo *D&C*.

Uma forma de ter uma aplicação *D&C* paralelizada é criar novos processos para resolver os sub-problemas. Cada processo pode ser executado em paralelo, porém deve-se considerar as dependências de dados representadas pelas arestas dos DAGs. A Figura 2.2 mostra um exemplo simples de execução de uma aplicação tipo *D&C*. É apresentada a execução de um nível de recursão, dividida em 4 passos, de uma operação $\oplus$ efetuada em dois vetores com duas posições. A recepção dos dados de entrada pelo procedimento inicial (P_0) está representada no passo 1. No passo 2, P_0 verifica que o problema pode ser decomposto e cria 2 procedimentos (P_1 e P_2), passando parte do problema para cada procedimento. No passo 3, P_1 e P_2 verificam que o problema recebido é simples (no caso, um vetor unitário), executam a operação $\oplus$ sobre os dados e retornam os resultados parciais para P_0 que, então, combina os resultados parciais para gerar um vetor com o resultado final (passo 4).

2.1.2 Exemplos de problemas resolvidos com *D&C*

Esta seção apresenta, de forma breve, alguns problemas e uma proposta de solução utilizando o *D&C*. O objetivo é contextualizar como e em que tipo de problemas a abordagem *D&C* pode ser aplicada. Maiores detalhes de implementação não são tratados.

- operações em seqüências de números: os números são armazenados em um

vetor que pode ser facilmente dividido entre os processadores disponíveis. Após a operação, a solução é obtida a partir da combinação dos resultados parciais, como ilustrado na Figura 2.2;

- teste de primalidade de Eratóstenes: cada processador inicia a marcação dos múltiplos de números diferentes e, ao final, combinam-se todas as marcações feitas por todos os processos;

- ordenação de números - utilizando o algoritmo *Bucket Sort*: Inicializa-se um vetor de *buckets* vazios com um tamanho igual ao número de elementos. Percorre-se o vetor original, atribuindo cada elemento a um dos *buckets*. Ordena-se os *buckets* não vazios e coloca-se os elemenos no vetor original.

- *ray tracing*: como cada linha da imagem pode ser gerada em paralelo, cada processo gera uma parte da imagem. No final do processamento, combinam-se as partes para gerar a imagem final;

- busca de elemento em árvore: cada bifurcação da árvore pode gerar um novo processo (em outro processador) que busca o elemento na sua sub-árvore. Assim que o elemento for encontrado, enviam-se mensagens para os demais processos para sinalizar o fim do trabalho;

- caixeiro viajante (TSP): gera-se uma árvore com todos os possíveis caminhos e inicia-se com um valor infinito para o menor caminho. Percorre-se a árvore gerando novos processos para as bifurcações, como na busca em árvore, calculando o valor do caminho. Ao completar um caminho, atingindo um nó folha, compara-se o valor desse caminho com o valor do menor caminho e, caso seja menor, avisa os demais processos que foi encontrado um caminho com menor custo.

Os exemplos acima mostram que a classe de problemas resolvidos pelo modelo *D&C* abrange aplicações bastante utilizadas, como é o caso do TSP, que exige grande poder computacional. Porém, existem outras aplicações que são adequadas para o modelo *D&C* e, a seguir, discute-se as vantagens e desvantagens do modelo.

2.1.3 Vantagens e desvantagens do modelo *D&C*

Esta seção tem como objetivo fazer um balanço geral do modelo *D&C*, que pode ser usado para se verificar quando se pode utilizá-lo. Principais vantagens:

- aproveitar recursos que se alteram dinamicamente com o tempo através da criação dinâmica de processos;

- melhor aproveitamento de recursos em ambientes heterogêneos. É possível ajustar a granularidade da tarefa executada pelos processos através da alteração da profundidade da árvore de recursão. Com isso, pode-se balancear a carga atribuindo-se uma carga diferenciada para cada recurso;

- pode-se alterar também a largura da árvore para controlar o grau de paralelismo da aplicação;

- possibilita implementação de aplicações paralelas com dependências entre as tarefas.

Principais desvantagens:

- a modelagem e a implementação da sincronização entre as tarefas pode tornar-se um trabalho árduo;
- pode necessitar ajuste de parâmetros, como profundidade da recursão, para se adequar ao ambiente de execução e obter-se um resultado eficiente.

Após essa análise, é feita uma busca pelos ambientes de programação paralela existentes e verificada sua adequação para a programação no modelo *D&C*.

2.2 Levantamento dos principais ambientes de programação que suportam *D&C*

Nesta seção é feito um levantamento dos principais ambientes que suportam programação *D&C* paralela encontrados na literatura. A partir dessa lista, foram escolhidos os ambientes mais adaptados para uma análise mais detalhada.

Na década de 1990 foram propostos diversos ambientes de programação paralela baseados em C e C++. O ambiente **CC++** (CHANDY; KESSELMAN, 1992) oferece construções paralelas e utiliza C++ nas partes seqüenciais. Já o **Mentat** (GRIMSHAW, 1993) permite especificar objetos seqüenciais e paralelos, aumentando o controle do programador sobre o paralelismo. O ambiente **DECK** (BARRETO; NAVAUX; RIVIÈRE, 1998) permite programar aplicações em *clusters* e oferece uma linguagem orientada a objetos chamada **DPC++** (CAVALHEIRO et al., 1995). O *Athapascan* (CAVALHEIRO; GALILÉE; ROCH, 1998) é um ambiente de programação para arquiteturas com memória compartilhada e permite avaliar o custo de execução de um código sem a necessidade de um modelo de máquina.

Também foram propostos ambientes mais restritos, como o **Parallel-C++** (JO; GEORGE; TEAGUE, 1991), que oferece paralelismo através da semântica "co-begin/co-end", e o **OOMDC/C** que oferece apenas comunicação de objetos por troca de mensagens. O **ABCL/1** (YONEZAWA, 1990) oferece objetos concorrentes orientados a mensagens, que podem se comunicar através de mensagens bloqueantes, não-bloqueantes e de *futures*.

O **Charm** (KALE et al., 1995) foi uma das primeiras implementações do conceito de *Atores*, que são objetos concorrentes que se comunicam apenas por troca de mensagens. **Charm++** (KALE; KRISHNAN, 1996) é baseado no Charm e suporta diferentes modos de compartilhamento de informações. Ele reúne recursos propostos em outros ambientes, como objetos seqüenciais e paralelos, comunicação por troca de mensagens e *futures*. O **Cilk** (BLUMOFE et al., 1995) é um ambiente de programação paralelo focado em arquiteturas SMP. Ele é baseado na linguagem C e oferece construções para representar o paralelismo para aplicações *D&C*. O **Cilk** oferece também um escalonador de processos que é provado ser eficiente (VEE; HSU, 1999). Baseando-se no **Cilk**, foi desenvolvido um ambiente para execução de programas em arquiteturas com memória distribuída e aproveitamento de recursos ociosos, chamado **Cilk-NOW** (BLUMOFE; LISIECKI, 1997).

Outra linguagem que vem despertando interesse de pesquisa em ambientes de programação paralela é o Java, que é utilizado nos ambientes descritos a seguir. O **Atlas** (BALDESCHWIELER; BLUMOFE; BREWER, 1996) fornece um conjunto de classes Java e seu modelo de programação foi baseado no **Cilk**, que permite programar aplicações do tipo *D&C*. O **Javelin 3** (NEARY; CAPPELLO, 2002) também

fornece um conjunto de classes Java para programar aplicações *Mestre/Escravo* e *Branch-and-Bound* (*B&B*). O ambiente **ProActive** (CAROMEL; KLAUSER; VAYSSIÈRE, 1998) oferece uma biblioteca de programação Java que implementou recentemente uma API para desenvolvimento de aplicações *B&B*. O **Satin** (NIEUWPOORT; KIELMANN; BAL, 2000) é um ambiente baseado na plataforma **Ibis** (NIEUWPOORT et al., 2002) e foca aplicações do tipo *D&C*. Os três últimos ambientes são projetados para executar aplicações em arquiteturas com redes de interconexão heterogêneas. Para isso, o **Javelin 3** e o **Atlas** utilizam um algoritmo de escalonamento hierárquico em árvore. Já o **Satin**, além de implementar um escalonador hierárquico, implementa também outros algoritmos de escalonamento e apresenta um resultado mais eficiente para ambientes com redes heterogêneas (NIEUWPOORT; KIELMANN; BAL, 2001).

A biblioteca MPI é um padrão consolidado de programação e já foi utilizada para programar diversas aplicações que necessitam minimizar a latência da comunicação em máquinas com memória distribuída. Na implementação mais antiga e conhecida da biblioteca (norma 1.2) não são oferecidos recursos para implementar aplicações *D&C* dinâmicas. Alguns ambientes oferecem meios para facilitar a programação com MPI, como é o caso do MAP_3S (Advanced Pattern-Based Parallel Programming System) (MEHTA; AMARAL; SZAFRON, 2005), porém ainda não existe um ambiente adequado para programação com MPI para aplicações *D&C* dinâmicas.

2.3 Comparação entre os ambientes

Esta seção apresenta uma comparação entre as características mais relevantes para programação de aplicações *D&C* dinâmicas: os mecanismos para criação dinâmica de tarefas, sincronização e escalonamento de processos. Os ambientes escolhidos para serem detalhados nesta seção foram o Cilk, por ter sido um dos primeiros ambientes eficientes para *D&C*, o Satin, que baseou-se no modelo do Cilk para executar aplicações *D&C* em ambientes com memória distribuída, e o Charm++, que apresenta diversos mecanismos para expressão do paralelismo e de escalonamento.

2.3.1 Mecanismos para criação de tarefas

A seguir são apresentados os mecanismos oferecidos em cada ambiente para criação dinâmica de tarefas.

O primeiro mecanismo estudado foi o modificador `spawn`, proposto no Cilk e também utilizado no Satin. Esse modificador pode ser colocado em métodos ou funções que tenham potencial para serem executados em paralelo. Ao fazer uma chamada de uma função `spawn`, o programa segue sua execução sem ficar bloqueado à espera do resultado da função. Para utilizar os dados calculados por uma função desse tipo é necessário utilizar algum mecanismo de sincronização.

No Charm++ as tarefas são representadas pelos *chares*, que são objetos paralelos e representam unidades locais de trabalho. Cada *chare* possui dados locais, métodos para tratamento de mensagens e possibilidade de criar novos *chares*, assim como processos MPI-2. Existe ainda um tipo especial de *chare*, chamado *branch-office*, que possui uma ramificação em cada processador e um único nome global. *Branch-office chares* oferecem métodos seqüenciais que podem ser acessados por *chares* de forma transparente em qualquer processador.

2.3.2 Mecanismos de sincronização

No Para sincronizar as tarefas criadas com `spawn` deve-se utilizar o `sync`, que garante que o processo possa utilizar os dados calculados pelos filhos com coerência. Ao fazer uma chamada `sync`, o processo verifica se as tarefas criadas já terminaram sua execução e, caso já tenham terminado, ele continua sua execução normalmente. Caso contrário, o processo fica bloqueado até o término dos processos filhos.

No Charm++ a sincronização pode ser feita através de *Futures*, objetos de comunicação, replicados ou compartilhados. O *future* é uma estrutura que serve para armazenar um valor que pode ser acessado no futuro por outro *chare*. O acesso a essa estrutura é bloqueante e pode ser comparado a um acesso ao resultado de um método *spawned* precedido de uma chamada `sync`. A utilização de objetos de comunicação permite que um *chare* se comunique por troca de mensagens, tornando a comunicação semelhante à realizada com MPI. Os demais objetos introduzem conceitos para comunicação não encontrados nos outros ambientes.

Nos três ambientes estudados é possível programar aplicações *D&C* dinâmicas, porém ainda é necessário estudar e adaptar os modelos do Charm++ para se adequarem ao modelo *D&C*.

2.3.3 Mecanismos de escalonamento

O **Cilk** oferece um escalonador que utiliza o algoritmo de *Work-Stealing*. Esse algoritmo é simples e fácil de ser implementado, porém só é adequado para sistemas com conexões de rede homogêneas ou memória compartilhada.

O **Satin** implementa diversos algoritmos de escalonamento e em (NIEUWPOORT; KIELMANN; BAL, 2001) é apresentada uma comparação entre: Roubo Aleatório de Tarefas (*Random Work Stealing*), Atribuição Aleatória de Tarefas (*Random Work Pushing*), Roubo Hierárquico de Tarefas para *clusters* (*Cluster-aware Hierarchical Work Stealing*), Roubo Baseado-na-carga de Tarefas para *clusters* (*Cluster-aware Load-based Work Stealing*), Roubo Aleatório de Tarefas para *clusters* (*Cluster-aware Random Work Stealing*), Roubo Múltiplo Aleatório de Tarefas para *clusters* (*Cluster-aware Multiple Random Work Stealing*) e o Roubo Adaptativo Aleatório de Tarefas para *clusters* (*Adaptive Cluster-aware Random Work Stealing*). Destacam-se entre eles os algoritmos Roubo Aleatório de Tarefas para *clusters* e Roubo Adaptativo Aleatório de Tarefas para *clusters*, por apresentarem os melhores desempenhos em ambientes com rede heterogêneas. O Roubo Aleatório de Tarefas para *clusters*, apesar de não ter o melhor desempenho em poucos casos, é bem mais simples que o Adaptativo. Por esse motivo, o Roubo Aleatório se mostra mais adequado para escalonar processos em ambientes como *cluster* de *clusters* e grades computacionais.

No **Charm++**, o escalonamento pode ser feito de forma centralizada ou dinâmica (CHARM++ LANGUAGE MANUAL, 2005). As estratégias centralizadas são as seguintes:

- RefineLB: move objetos dos processadores mais sobrecarregados até atingir uma média;
- RefineCommLB: mesmo conceito, porém leva em consideração a comunicação entre os processos;
- RandCentLB: atribui objetos para processadores de forma aleatória;

- RecBisectBfLB: particiona recursivamente enumerando primeiro em largura;
- MetisLB: utiliza o Metis (METIS WEBSITE, 2005) para particionar o grafo de comunicação do objeto;
- GreedyLB: utiliza um algoritmo guloso, sempre pegando um objeto mais pesado para o processador com a menor carga;
- GreedyCommLB: algoritmo guloso que também leva em consideração o grafo de comunicação;
- ComboCentLB: é utilizado para combinar quaisquer das estratégias acima.

As estratégias distribuídas são as seguintes:

- NeighborLB: cada processador tenta obter uma carga média em relação aos seus vizinhos;
- WSLB: um balanceamento para *clusters* de estações de trabalho que pode detectar mudança nas cargas das máquinas e ajustar a carga sem interferir no trabalho do usuário.

Porém, não foram encontradas na bibliografia comparações entre os mecanismos de escalonamento do ambiente.

2.4 Conclusões sobre o capítulo

A seguir, é apresentado um resumo das características dos ambientes estudados, a adaptação de cada um para programar aplicações *D&C* dinâmicas e, por fim, algumas considerações sobre o estudo.

2.4.1 Comparação entre as características dos ambientes

Na Tabela 2.1, comparam-se as principais características dos ambientes estudados. A segunda e terceira coluna referem-se ao modelo de programação oferecido pelo ambiente: através de primitivas de memória compartilhada ou troca de mensagens. A quarta coluna refere-se à possibilidade de criação de tarefas dinamicamente ao longo da execução da aplicação.

Na Tabela 2.2, faz-se uma comparação entre os mecanismos de criação de tarefa, sincronização e escalonamento oferecidos pelos ambientes estudados. A segunda coluna mostra os modelos de criação de tarefa oferecidos. A terceira coluna mostra os mecanismos de sincronização oferecidos, onde "msg" significa que o ambiente oferece troca de mensagens. A quarta coluna refere-se à disponibilidade de mecanismos para escalonar as tarefas pelo ambiente.

Vale destacar que os ambientes apresentados permitem executar aplicações em arquiteturas com memória distribuída, exceto o Cilk que foi desenvolvido para máquinas SMP.

2.4.2 Facilidades para programação *D&C* dinâmica

O modelo `spawn/sync`, proposto no Cilk e utilizado no Satin, é intuitivo para programação com *D&C*. Através dele pode-se definir de forma simples o grau de

Tabela 2.1: Comparação entre as principais características dos ambientes estudados.

Ambiente	Mem. Comp.	Troca Msg	Criação Dinâmica
Cilk	•		•
Satin	•		•
Charm++	•	•	•
Cilk-NOW	•		•
Faucets	•	•	•
MAP_3S		•	
MPI-1.2	•	•	
Jade	•	•	•
Adaptive-MPI		•	•

Tabela 2.2: Comparação entre os mecanismos de criação de tarefa, sincronização e escalonamento dos ambientes estudados.

Ambiente	Criação Tarefa	Sincronização	Escalonamento
Cilk	spawn	sync	•
Satin	spawn	sync	•
Charm++	chare/future	msg/future	•
Cilk-NOW	spawn	sync	•
Faucets	chare/future	msg/future	•
MAP_3S	estática	msg/padrões	
MPI-1.2	estática	msg	
Jade	chare/future	msg/future	•
Adaptive-MPI	estática	msg	•

paralelismo (`spawn`) e os pontos de sincronização (`sync`). O modelo do Charm++ oferece os *futures*, que podem ser usados de forma semelhante ao `spawn`/`sync` e oferece também comunicação por troca de mensagens.

Quanto ao escalonamento, o Cilk oferece apenas escalonamento dentro de uma máquina com memória compartilhada. Para um melhor aproveitamento de recursos interconectados por redes WAN, o Satin compara diversos algoritmos e apresenta o Roubo Aleatório de Tarefas para *Clusters* como o mais simples e eficiente. O Cilk-NOW estende o mecanismo de roubo de tarefas do Cilk, porém é focado no escalonamento para redes de estações de trabalho (NOW). O Charm++ apresenta diferentes mecanismos de escalonamento, porém nenhum específico para máquinas interconectadas por redes WAN. Já o MPI 1.2 não permite criar processos dinamicamente.

Este capítulo apresentou os principais ambientes para programação paralela de aplicações *D&C* dinâmicas. Os ambientes encontrados na bibliografia estão bem

consolidados e apresentam resultados eficientes, como, por exemplo, o Cilk e o Satin.

Recentemente, surgiram as primeiras implementações da norma MPI-2, que prevê criação dinâmica de processos. Apesar do MPI-2 não ter sido especificamente projetado para o modelo de programação *D&C*, é possível utilizar os recursos oferecidos para programar esse tipo de aplicação. Uma vez que o MPI é um ambiente eficiente de programação paralela e amplamente utilizado, surge o interesse no estudo de como se utilizar seus novos recursos para se ter um novo ambiente de programação para aplicações *D&C* dinâmicas.

A seguir são propostas diferentes formas de se modelar, escalonar e implementar aplicações *D&C* dinâmicas utilizando os recursos do MPI-2. São apresentados, para fins de validação, os resultados obtidos e comparações com ambientes da bibliografia.

3 DIVISÃO-E-CONQUISTA COM MPI-2

O MPI suporta programas paralelos baseados na troca de mensagens. Já com MPI-2, pode-se criar processos dinamicamente, o que possibilita uma programação *D&C*, semelhante aos ambientes Cilk e Satin. No entanto, para garantir um escalonamento eficiente do programa (Capítulo 4), deve-se restringir o modelo de programa tratado.

Este capítulo apresenta o modelo de programação que é tratado neste trabalho (Seção 3.1) e os recursos do MPI que estão disponíveis para se programar seguindo o modelo proposto (Seção 3.2).

3.1 Modelo de programação paralela tratado

O modelo de aplicação tratado neste trabalho é semelhante ao definido no ambiente Cilk e utilizado no ambiente Satin. Em (PEZZI et al., 2006) encontra-se uma proposta inicial do modelo que, a seguir, é detalhado. A escolha do modelo Cilk baseia-se no fato de ser provado teoricamente e comprovado na prática que é possível atingir um escalonamento próximo do ótimo, utilizando Cilk em máquinas SMP (VEE; HSU, 1999). Além disso, o Satin obteve bons resultados adaptando um escalonador *D&C* para arquitetura com memória distribuída e rede de interconexão heterogênea (NIEUWPOORT; KIELMANN; BAL, 2001; NIEUWPOORT et al., 2006).

A Seção 3.1.1 introduz o modelo proposto pelo Cilk (BLUMOFE; LEISERSON, 1994). As seções seguintes apresentam reflexões de como adaptar o modelo para utilizá-lo com MPI-2 em ambientes com memória distribuída.

3.1.1 Divisão-e-Conquista no Cilk

Esta seção apresenta as principais características do modelo de programação *D&C* no Cilk. Uma especificação mais detalhada pode ser encontrada em (BLUMOFE; LEISERSON, 1994).

Uma computação com múltiplos fluxos de execução pode ser vista como um conjunto de fluxos, que são compostos por uma seqüência de tarefas unitárias. Essas seqüências devem ser executadas na ordem prevista e não podem ser executadas em paralelo. Para executar um fluxo é necessário alocar um espaço de memória, chamado "Quadro de Ativação" (*Activation Frame*), que será utilizado para armazenar os dados calculados. Durante a execução de um fluxo, este pode criar outros e cada criação de um novo é como uma chamada de sub-rotina, exceto que a rotina pode executar concorrentemente com a sub-rotina chamada. Fluxos criados

são chamados de filhos e podem ser criados quantos forem necessários. Dessa forma, os fluxos podem ser organizados em uma hierarquia de árvore de ativação (Figura 3.1). As arestas contínuas representam a dependência entre as tarefas. Criação e sincronização de fluxos são representadas por arestas tracejadas e onduladas, respectivamente. No modelo do Cilk, assume-se que uma tarefa possui um número máximo constante de arestas de dependência incidentes e que uma tarefa, ao atingir um ponto em que uma dependência ainda não está pronta, pára até que esta dependência seja satisfeita. É dito também que quando um fluxo termina de executar a última tarefa, ele morre.

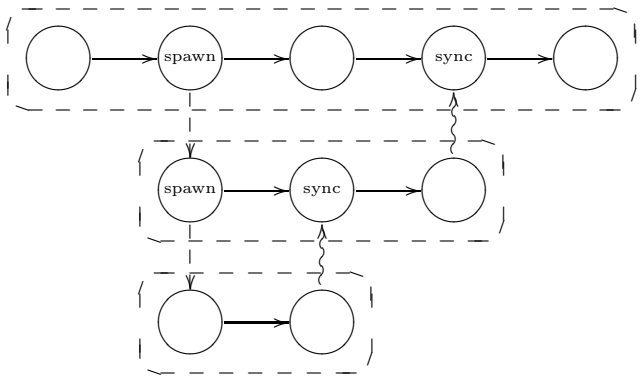

Figura 3.1: Exemplo de computação com múltiplos fluxos no Cilk.

Os fluxos de execução no ambiente Cilk são funções não-bloqueantes que podem criar fluxos filhos. Para a criação de um novo fluxo, deve ser utilizado o comando `Spawn`. Para a sincronização do fluxo com seu filho utiliza-se o comando `Sync`, que funciona como uma espécie de barreira. Um fluxo é executado enquanto tiver tarefas para executar ou até encontrar o comando `Sync`. Ao encontrar o `Sync`, ele verifica se o filho já terminou a execução (os dados estão prontos) e, neste caso, continua sua execução. Caso contrário, ele aguarda até que o filho disponibilize os dados.

Figura 3.2: Aplicação Cilk vista como um DAG.

A execução da aplicação na Figura 3.1 está representada na Figura 3.2 como um DAG. Essa execução inicia com um processo que gera um processo filho que, por sua vez, gera um terceiro. Os processos criadores continuam suas execuções em paralelo após o `Spawn` até encontrarem uma chamada `Sync`. Cada processo pai fica bloqueado na primitiva `Sync`, até que seu filho termine a execução.

A seguir, é proposto o desenvolvimento de um modelo para programação de aplicações *D&C* com MPI-2, ressaltando as semelhanças e as diferenças entre o modelo proposto e o oferecido originalmente pelo ambiente Cilk.

Tabela 3.1: Terminologia e conceitos utilizados neste trabalho.

Conceito	Cilk	MPI-2
Bloco básico de instruções	Tarefa	Tarefa
Spawn	Thread	Processo

3.1.2 Definição de tarefa em programa *D&C* MPI-2

No ambiente Cilk, uma tarefa é representada pelo processamento feito entre chamadas `Spawn/Sync`. Um `Spawn` é uma execução de função não-bloqueante, semelhante a uma *thread*, que pode ser feita em paralelo em outro processador. Já no MPI-2, um `Spawn` pode executar uma criação remota de processo, que envolve transferência de dados via rede.

Neste trabalho, uma tarefa é definida como um bloco de instruções que pode ser executado utilizando sincronização em dois pontos: no início, para receber os dados de entrada, e no final, para devolver os resultados. Como o problema pode ser resolvido por *D&C*, assume-se que um processo pode gerar recursivamente novos processos para sub-dividir com esses sua tarefa, ou resolvê-la sozinho. Na Tabela 3.1, resume-se a terminologia adotada no texto.

A seguir, são apresentadas as restrições feitas na comunicação para respeitar o modelo de comunicação proposto pelo Cilk.

3.1.3 Restrições de comunicação

Programas do tipo *D&C* que criam processos recursivamente ao longo da execução, formam uma hierarquia de comunicação em árvore. Isto é, quando um processo cria novos processos e atribui-lhes parte das tarefas, ele pode se comunicar apenas com seus processos criados (filhos) e seu criador (pai). Apesar do MPI-2 permitir que se faça comunicação direta entre quaisquer processos, isso pode modificar o grafo de dependências e pode acarretar em complicações para o escalonador.

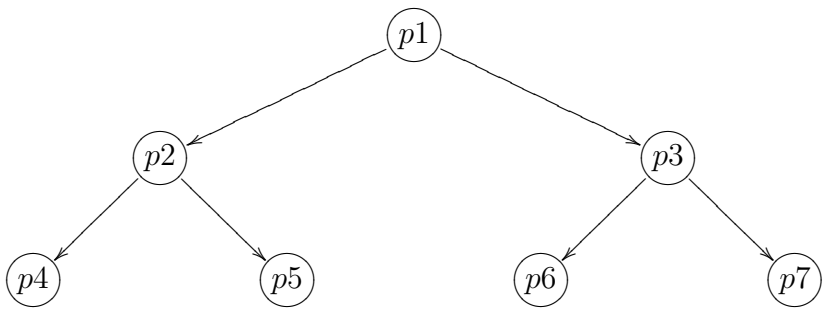

Figura 3.3: Exemplo de hierarquia de comunicação em uma aplicação *D&C*.

A Figura 3.3 mostra um exemplo de DAG de uma aplicação *D&C*, onde cada processo divide o trabalho recursivamente em duas partes. Os vértices representam os processos e as arestas representam entre quais processos pode haver comunicação.

Concluída uma descrição do modelo tratado, a próxima seção detalha os recursos oferecidos pelo MPI-2 e como podem ser utilizados para aplicações *D&C* dinâmicas.

3.2 Suporte no MPI-2 para programação de aplicações dinâmicas

Esta seção apresenta os principais recursos introduzidos no MPI-2 que podem ser utilizados para programação de aplicações *D&C* dinâmicas. Inicialmente é detalhada a primitiva de criação dinâmica de processos. Em seguida, mostra-se como pode ser feita a comunicação entre os processos e uma aplicação *D&C* de exemplo, implementada com MPI-2.

3.2.1 Criação dinâmica de processos

MPI-2 provê uma interface que permite a criação dinâmica de processos durante a execução de um programa MPI, que podem se comunicar através de troca de mensagens. Apesar da norma MPI-2 prover outras funcionalidades, a primitiva mais relevante para este trabalho é a que permite a criação dinâmica de processos. Maiores informações sobre os recursos do MPI-2 podem ser encontrados em (GROPP; LUSK; THAKUR, 1999).

A primitiva introduzida que cria processos durante a execução de um programa MPI é o `MPI_Comm_spawn`. Os principais argumentos dessa primitiva são: nome do executável, que deve ser um programa MPI padrão (com as instruções `MPI_Init` e `MPI_Finalize`); os parâmetros passados pela linha de comando; o número de processos que devem ser criados; o intra-comunicador que será enviado para os processos criados e o inter-comunicador que será criado para que o processador possa se comunicar com os processos criados. A seguir, mostra-se como pode ser feita a comunicação entre os processos criados com o `MPI_Comm_spawn`.

3.2.2 Sincronização entre os processos

No MPI a sincronização deve ser feita através de troca de mensagens. De acordo com o modelo definido na Seção 3.1, pode-se restringir entre quais processos haverá comunicação: eles podem se comunicar apenas com seus processos criados (chamados de filhos) e seu criador (chamado de pai). Dessa forma o comunicador enviado para o filho deverá conter apenas o processo pai, evitando a sobrecarga de criação e sincronização de grupos que não precisam se comunicar.

Outra limitação que se pode fazer é quanto aos momentos de troca de mensagens entre pai e filho: é feita sincronização antes do processamento de uma tarefa, para envio dos dados de entrada, e após o processamento, para envio dos resultados.

Os dados de entrada podem ser enviados no momento da criação do novo processo, utilizando o argumento de linha de comando do executável MPI. No caso de dados complexos, pode-se utilizar o recurso `MPI_Datatype` para criar um tipo de dados complexo e enviar os dados de entrada por troca de mensagens MPI. Já para o envio dos resultados ao final da execução, a única opção é utilizar troca de mensagens MPI.

3.2.3 Exemplo de aplicação dinâmica com MPI-2

Fibonacci: esta aplicação calcula o n-ésimo número de *Fibonacci*, denotado no texto por $fib(n)$, seguindo a definição recursiva. Caso o n pedido seja menor que 2, o processo devolve para o pai o próprio n através de uma mensagem de `MPI_send`. Caso contrário, o processo cria 2 novos processos para calcular $fib(n-1)$ e $fib(n-2)$, aguarda os processos devolverem o resultado através de dois `MPI_Recv` bloqueantes

e, então, soma os resultados parciais e envia para o pai o valor através de uma mensagem de `envio`.

```
cilk int fib (int n){
  if (n < 2)
    return n;
  else{
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;

   return (x+y);
   }
}
```

```
n=atoi(argv[1]);
if (n < 2 )
   MPI_Send(n, parent);
 }else{
   int x,y;
   MPI_Comm_spawn(fibo, n-1);
   MPI_Comm_spawn(fibo, n-2);
   MPI_Recv(x, children_comm[0]);
   MPI_Recv(y, children_comm[1]);
   MPI_Send(x+y, parent);
}
```

Figura 3.4: Código para cálculo do n-ésimo número de *Fibonacci*: comparação entre Cilk (esquerda) e MPI-2 (direita).

Na Figura 3.4, compara-se dois trechos de código para cálculo do n-ésimo número de *Fibonacci*: na esquerda está representado o código de uma função *spawnable* do Cilk e na direita uma possível implementação utilizando as primitivas do MPI-2. No exemplo, o método utilizado para enviar o dado de entrada é através da passagem de parâmetro para o executável e apenas o envio do resultado é feito através de troca de mensagem.

3.2.4 Limitações encontradas inicialmente

A principal limitação encontrada para executar aplicações *D&C* dinâmicas com MPI-2 é a falta de mecanismos eficientes para escalonamento dos processos. Apesar da norma MPI-2 não prever mecanismos de escalonamento, a distribuição LAM-MPI oferece uma primitiva para distribuir os processos entre os recursos disponíveis. Em (CERA et al., 2006) encontram-se testes e maiores detalhes sobre o escalonamento oferecido pelo LAM.

Tabela 3.2: Escalonamento do *Fibonacci* com *Round-Robin* oferecido pelo LAM.

Aplicação	Quantidade de processos criados				
Fibonacci(n)	Nó 1	Nó 2	Nó 3	Nó 4	Nó 5
n = 7	25	0	0	0	0
n = 10	177	0	0	0	0

A Tabela 3.2 mostra o número de processos em cada nó em duas execuções do *Fibonacci*, utilizando o mecanismo de escalonamento *Round-Robin* nativo oferecido pelo LAM. O resultado mostra que todos os processos foram executados no mesmo processador. Após alguns testes, verificou-se que o mecanismo do LAM só faz escolha

Round-Robin entre os recursos em uma mesma chamada `MPI_Comm_spawn`. Isto é, pode-se passar, como argumento da função, o número de processos que devem ser criados. Caso sejam criados mais de um processo, será feita uma distribuição de forma *Round-Robin* entre os processadores. Porém, em todas chamadas consecutivas da primitiva de criação de processos, o *Round-Robin* é iniciado sempre no mesmo nó. Como no programa *Fibonacci* cria-se apenas um processo por chamada da primitiva, o resultado é que todos os processos são criados no mesmo nó.

Para melhorar o balanceamento, pode-se alterar o nó onde é iniciado o *Round-Robin* a cada vez que um novo processo é criado. No entanto, para obter um balanceamento homogêneo, seria preciso que cada processo soubesse, no momento de criação de um novo processo, em qual processador foi criado o último processo. Como essa informação não é disponibilizada pela LAM, foi feita uma mudança na aplicação para alternar o nó de início do *Round-Robin* e melhorar a distribuição de processos. A mudança é bem simples: se um processo está sendo executado no processador n, ele criará processos no processador $n+1$. A decisão é feita localmente e, por isso, dificilmente será atingida uma boa distribuição global.

Tabela 3.3: Comparação do escalonamento nativo do LAM com escalonamento local: número de processos em cada nó.

Mecanismo de escalonamento	Quantidade de processos criados				
	Nó 1	Nó 2	Nó 3	Nó 4	Nó 5
LAM nativo	25	0	0	0	0
Escolha local	8	4	8	2	3

Na Tabela 3.3, comparam-se o escalonamento ao executar $fib(7)$ utilizando o mecanismo nativo do LAM de *Round-Robin* e um escalonamento local. Consegue-se melhorar a distribuição com o escalonamento local, mas ainda assim não se consegue fazer um bom balanceamento.

Um maneira de melhorar o balanceamento é armazenar as decisões de escalonamento, de modo que seja possível fazer uma distribuição *Round-Robin* global dos processos nos recursos disponíveis. Uma possível solução para resolver este problema é utilizar uma biblioteca de escalonamento com um escalonador centralizado (CERA et al., 2006). Essa abordagem utiliza redefinição de algumas primitivas do MPI, fazendo com que a aplicação, para efetuar o escalonamento, execute código da biblioteca nos pontos onde sejam chamadas essas primitivas.

Objetivando facilitar o escalonamento para o programador e evitar redefinição de primitivas, que podem gerar intrusão na aplicação, foi implementado um escalonador independente da aplicação. Esse escalonador é centralizado e pode ser consultado sempre que um novo processo é criado. O escalonador utiliza primitivas MPI para publicar uma porta de conexão e a aplicação pode requisitar recursos através da porta publicada. Resultados dessa solução podem ser encontrados em (PEZZI et al., 2006). No entanto, essa solução não é satisfatória pois pode-se usar uma mecanismo de *Work-Stealing*, como no Cilk e no Satin.

3.3 Conclusões sobre o capítulo

Recentemente, as distribuições MPI apresentaram implementações funcionais de alguns recursos da norma MPI-2. Entre os recursos introduzidos, encontra-se uma primitiva que oferece criação dinâmica de processos. Como essa primitiva pode servir como base para programar aplicações *D&C* recursivas, o MPI-2 torna-se um candidato potencial de ambiente para essas aplicações. No entanto, ainda é preciso escolher uma forma eficiente de programar essas aplicações utilizando os recursos oferecidos pelo MPI-2. Com o objetivo de aproveitar melhor esses recursos, faz-se uma comparação das características dos ambientes *D&C* encontrados no Capítulo 2 e do MPI-2.

Tabela 3.4: Comparação entre as principais características dos ambientes *D&C* estudados com MPI-2.

Ambiente	Mem. Comp.	Troca Msg	Criação Dinâmica
Cilk	•		•
Satin	•		•
Charm++	•	•	•
Cilk-NOW	•		•
Faucets	•	•	•
MAP_3S		•	
MPI-1.2	•	•	
Jade	•	•	•
Adaptive-MPI		•	•
MPI-2	•	•	•

Nas Tabelas 3.4 e 3.5, resumem-se algumas das características dos ambientes encontrados e compara-se com o que é oferecido pelo MPI-2. A segunda e terceira coluna da Tabela 3.4 referem-se ao modelo de programação oferecido pelo ambiente. Isto é, se eles oferecem primitivas usando modelo de memória compartilhada ou se o programador deve programar explicitamente a troca de mensagens entre os processadores. A quarta coluna refere-se à possibilidade de criação de tarefas dinamicamente ao longo da execução da aplicação.

Na Tabela 3.5, faz-se uma comparação entre os mecanismos de criação de tarefa, sincronização e escalonamento oferecidos pelos ambientes *D&C* estudados e o MPI-2. A segunda coluna mostra os modelos de criação de tarefa oferecidos. A terceira coluna mostra os mecanismos de sincronização oferecidos, onde "msg" significa que o ambiente oferece troca de mensagens. A quarta coluna refere-se à disponibilidade de mecanismos para escalonar automaticamente as tarefas.

Neste capítulo foi definido um possível exemplo para programar aplicações *D&C* utilizando MPI-2, baseando-se no modelo do Cilk. Também foi constatado que o MPI-2 oferece os principais recursos necessários para aplicações *D&C* dinâmicas. Porém, não oferece nenhum mecanismo automático de escalonamento, como é o caso de ambientes eficientes como Cilk e Satin.

Dado o contexto, o próximo capítulo trata da principal dificuldade encontrada até então: o escalonamento de processos criados dinamicamente. São apresentados algoritmos de escalonamento encontrados nos demais ambientes e é proposta a

Tabela 3.5: Comparação entre os recursos *D&C* oferecidos pelos ambientes estudados e o MPI-2.

Ambiente	Criação Tarefa	Sincronização	Escalonamento
Cilk	spawn	sync	•
Satin	spawn	sync	•
Charm++	chare/future	msg/future	•
Cilk-NOW	spawn	sync	•
Faucets	chare/future	msg/future	•
MAP_3S	estática	msg/padrões	
MPI-1.2	estática	msg	
Jade	chare/future	msg/future	•
Adaptive-MPI	estática	msg	•
MPI-2	spawn	msg	

utilização de dois mecanismos de escalonamento com MPI-2: Atribuição de tarefas (*Work Pushing*) e Roubo de Tarefas (*Work Stealing*).

4 ESCALONAMENTO DE APLICAÇÕES D&C COM MPI-2

Foi identificado, no capítulo anterior, que a principal dificuldade para programar aplicações *D&C* dinâmicas no MPI-2 é fazer o balanceamento da carga da aplicação. Dado o contexto, a Seção 4.1 apresenta o mecanismo de escalonamento oferecido pelo MPI-2. A seguir, na Seção 4.2, apresentam-se exemplos de dois escalonadores desenvolvidos utilizando o mecanismo do MPI-2. Os escalonadores não são específicos para aplicações *D&C*, porém mostra-se que é possível utilizá-los para este fim. Após, a Seção 4.3 apresenta o mecanismo de escalonamento oferecido pelo Cilk e algoritmos de balanceamento de carga utilizando *Work-Stealing* encontrados na literatura.

4.1 Escalonamento de processos criados dinamicamente

A seguir, é apresentado o mecanismo de escalonamento disponível no ambiente LAM MPI. Como esse mecanismo não oferece um controle sobre os processos semelhante ao Cilk, apresentam-se duas formas possíveis de escalonar processos criados dinamicamente no MPI-2.

Como a norma MPI-2 não trata escalonamento automático de processos, estuda-se o mecanismo manual oferecido aos programadores para distribuir os processos entre os recursos disponíveis. É apresentado como exemplo, o mecanismo oferecido pelo LAM para escalonar os processos criados dinamicamente. O LAM utiliza uma estrutura de dados `MPI_Info`, onde coloca as informações sobre o escalonamento como, por exemplo, o nó inicial e o tipo de distribuição que deve ser feita.

A Figura 4.1 mostra a configuração do LAM para escalonar processos por *Round-Robin*, escolhendo o nó seguinte na lista de nós disponíveis do LAM para iniciar a distribuição. Percebe-se que são utilizadas primitivas não especificadas na norma MPI para obter informações sobre o ambiente de execução. A chamada `getntype(0,0x02)` retorna o número total de nós disponíveis para execução e a função `getnodeid()` retorna o identificador do nó onde está sendo executado o processo. Utilizando essas informações, calcula-se o identificador do próximo nó, coloca-se esse identificador em uma estrutura de dados `MPI_Info` que, por fim, será passada como parâmetro para a primitiva de criação de processo `MPI_Comm_spawn`.

A seguir, são propostas duas formas de modelar o escalonamento de processos criados dinamicamente em aplicações *D&C*. A Seção 4.2 propõe utilizar atribuição de tarefas com um escalonador centralizado com duas formas de escolha do próximo recurso. Na Seção 4.3, propõe-se utilizar um escalonador distribuído para aplicações MPI-2, baseando-se no que é oferecido pelos ambientes Cilk e Satin.

```
MPI_Info local_info;
MPI_Info_create(&local_info);

maxnodeid = getntype(0,0x02);
currnodeid = getnodeid();
nextnodeid = ((currnodeid+1)%maxnodeid);

MPI_Info_set(local_info,
             "lam_spawn_sched_round_robin",nextnodeid);

MPI_Comm_spawn(command, argv , 1, local_info, myrank,
               MPI_COMM_SELF, &children_comm, errcodes);
```

Figura 4.1: Código para escalonamento de processos com LAM MPI.

4.2 Escalonamento on-line com Work-Pushing

Como os programas *D&C* tratados podem ser recursivos, novos processos podem ser criados em qualquer momento da execução. A dúvida que surge é: como escolher o recurso mais adequado para lançar o novo processo? A idéia do *Work-Pushing* é escolher sempre o recurso mais ocioso para lançar um novo processo, e a ociosidade pode ser medida de diversas formas. Por exemplo, pode-se definir como mais ocioso o processador que recebeu menos processos em um dado período, ou pode-se utilizar parâmetros como: carga, uso da rede, memória, etc.

Na Seção 4.1, mostrou-se como fazer uma escolha de recurso localmente em cada processo. Para que todos os processos possam fazer a melhor escolha, é preciso considerar as decisões dos demais processos, que podem estar em diferentes comunicadores.

A seguir são apresentadas duas possíveis formas de medir a ociosidade dos processadores e tomar as decisões de escalonamento *on-line* de processos *D&C*.

4.2.1 Escalonamento com algoritmo de Round-Robin

O algoritmo de *Round-Robin* distribui os processos igualmente entre os recursos disponíveis. O objetivo dessa implementação é melhorar a distribuição de processos do LAM, que não leva em consideração o histórico das decisões de escalonamento. A implementação desse escalonador e os resultados do escalonamento do LAM nativo encontram-se em (CERA et al., 2006).

Optou-se por utilizar um escalonador centralizado, com o objetivo de evitar uma sincronização entre todos os processos de uma execução, o que poderia ter um custo muito elevado. Nessa implementação, cada processo consulta o escalonador antes de criar um novo processo. Feita a consulta, coloca-se o identificador de nó recebido na estrutura `MPI_Info` e executa-se a primitiva `MPI_Comm_spawn`, como é mostrado na Figura 4.1.

Na Figura 4.2 estão representados resultados de escalonamento da aplicação *Fibonacci* com o escalonamento nativo do LAM e com o escalonador centralizado. São apresentadas execuções com os valores de entrada 7, 10 e 13. Não são apresentados resultados relativos ao tempo de execução pois esta é uma aplicação que tem o objetivo de apenas verificar o comportamento dos escalonadores. A execução de $fib(13)$

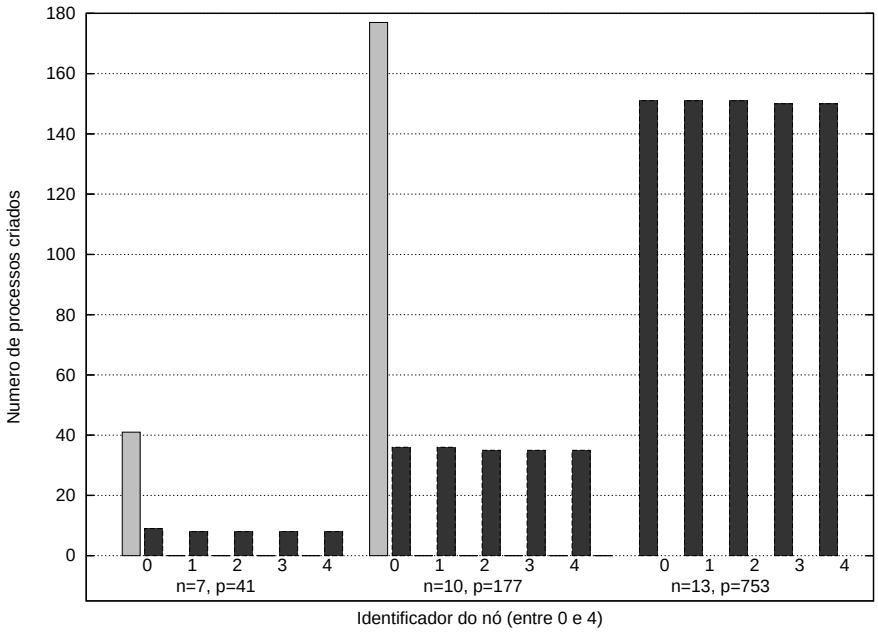

Figura 4.2: Balanceamento da carga em três execuções do *Fibonacci*, utilizando 5 nós. As barras claras representam o número de processos criados em cada nó, utilizando o escalonamento nativo do LAM. As barras escuras representam o número de processos criado em cada nó, utilizando um escalonador centralizado. Nos casos em que não aparece a barra cinza, nenhum processo foi criado naquele nó.

com o escalonador do LAM não foi concluída, pois a quantidade de processos gerados (753) é maior do que o limite suportado pelo LAM em um único nó. Pode-se ver que o escalonamento nativo do LAM é limitado e pode ser melhorado com uma solução simples, como está detalhado em (CERA et al., 2006).

Porém, existem aplicações em que não é possível dividir o trabalho igualmente entre os processos. Nesses casos, o processador que recebeu menos processos nem sempre será o mais adequado para receber uma nova carga de trabalho. Em aplicações irregulares pode ser vantajoso medir a utilização dos recursos e escolher o próximo processador em função dessas medidas. Com o objetivo de se obter um balanceamento mais adequado para esse tipo de aplicação, apresenta-se um escalonador que coleta e mantém informações sobre os recursos.

4.2.2 Escalonamento com gerenciamento de recursos

Com o objetivo de distribuir melhor o trabalho entre os recursos disponíveis, foi alterado o algoritmo de escolha do próximo recurso para considerar a carga de cada processador. Focam-se, então, aplicações onde não se consegue uma distribuição homogênea de carga entre os processos. Resumidamente, essa implementação utiliza um escalonador centralizado, como descrito na Seção 4.2.1, que coleta informações sobre a carga dos nós ao longo do tempo. A escolha do próximo nó para se lançar um processo é feita em função das informações coletadas. Este tipo de escalonamento também é chamado de *List Scheduling* e encontra-se uma descrição mais detalhada dessa implementação em (CERA et al., 2006).

Para validar seu funcionamento foi utilizada uma aplicação MPI-2 irregular que, dado um intervalo de números inteiros, verifica a primalidade de todos os números.

O intervalo é divido de forma recursiva, utilizando *D&C* e criação dinâmica de processos.

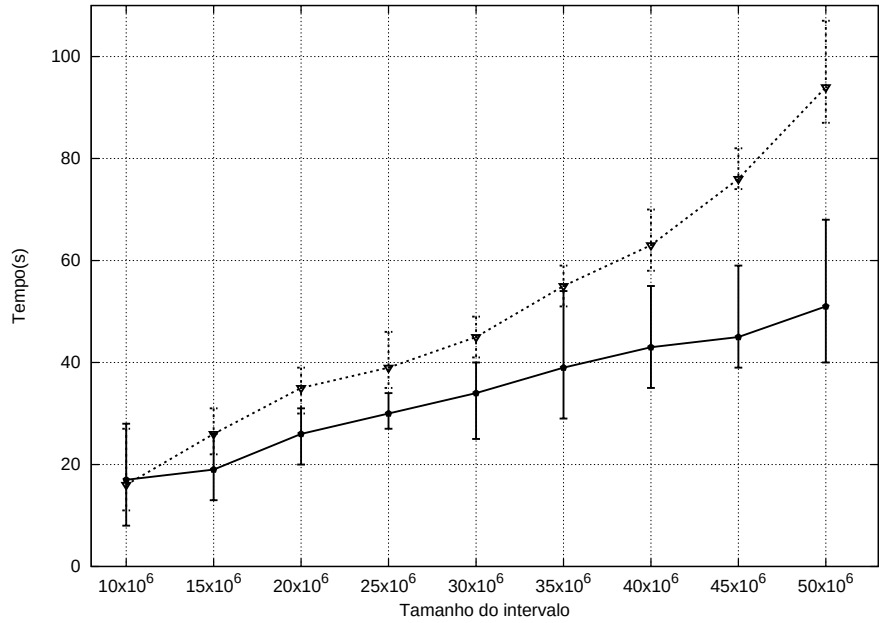

Figura 4.3: Tempos para a computação de números primos, com escalonamento *Round-Robin* (linha pontilhada) e gerenciamento de recursos (linha contínua).

Na Figura 4.3 apresentam-se os tempos obtidos em execuções com diferentes intervalos de entrada, utilizando o escalonador *Round-Robin* e o escalonador com gerenciamento de recursos. Como pode ser visto, no caso dessa computação irregular, o uso do escalonador que considera a carga dos processos para escolher o próximo recurso resulta em tempos de execução melhores.

Os resultados obtidos mostraram formas de melhorar o aproveitamento dos recursos em programas *D&C* dinâmicos com MPI-2. Além disso, os mecanismos apresentados facilitam a programação pois evitam que o programador necessite fazer manualmente a escolha dos recursos. No entanto, ainda não se tem um mecanismo distribuído semelhante ao escalonador do Cilk, que utiliza roubo de tarefas. Para melhorar o controle sobre o paralelismo das aplicações e ter-se um ambiente que permita programar as aplicações *D&C* semelhantemente ao Cilk, é proposto o uso de *Work-Stealing* com MPI-2.

4.3 Escalonamento distribuído com Work-Stealing

Esta seção apresenta a utilização de *Work-Stealing* para execução paralela de aplicações *D&C*. O conceito de *Work-Stealing* (roubo de tarefas) foi utilizado por (BLUMOFE et al., 1995) na década de 1990 e após, utilizado em outros ambientes, como por exemplo o *Satin* (NIEUWPOORT; KIELMANN; BAL, 2000).

Basicamente, *Work-Stealing* é feito da seguinte forma: o trabalho total é dividido em tarefas e distribuído entre os processadores disponíveis. Assim que um processador ficar ocioso (i.e., terminar suas tarefas) ele faz um pedido de roubo de tarefas para outro processador escolhido aleatoriamente. Caso o processador tenha

tarefas que não estejam em execução e receba um pedido de roubo, ele repassa uma parte de suas tarefas para o processador ocioso. Caso contrário, o processador ocioso continua fazendo pedidos de roubo para outros processadores, até que ele receba novas tarefas ou não haja mais tarefas livres.

Existem diferentes formas de escolher o próximo nó para fazer um pedido de roubo. A seguir, são apresentados alguns do algoritmos encontrados e em quais ambientes foram aplicados.

4.3.1 Mecanismo oferecido pelo Cilk

O objetivo do escalonamento no Cilk é determinar quais processadores de um computador executam quais tarefas em cada passo de tempo, sendo que em cada passo um processador pode executar no máximo uma tarefa. O escalonamento depende do modelo da aplicação e do número de processadores disponíveis na máquina paralela, por isso só será definido durante a execução da aplicação. Para que um escalonamento seja válido, a ordem de execução das tarefas deve obedecer às restrições dadas pelas arestas do grafo da computação. Como foi visto no Capítulo 3, assume-se que o grafo da aplicação é um DAG, de modo que existam escalonamentos possíveis.

Quando um processo executa uma chamada `Spawn`, o Cilk gera um novo fluxo de execução, que representa uma tarefa pronta para ser executada. Isto é, um `Spawn` coloca uma tarefa em uma fila de tarefas prontas, cujo início da execução pode não ser imediato, pois depende da disponibilidade de um processador.

Iniciada a execução de uma aplicação no Cilk, o ambiente divide as tarefas prontas entre os processadores existentes. Quando um processador termina de executar suas tarefas, faz-se um pedido de roubo de tarefa para outro processador até que não hajam mais tarefas para executar. Esse mecanismo é conhecido como *Work-Stealing*.

No MPI-2, não é oferecido um mecanismo de controle sobre os processos depois de uma chamada `MPI_Comm_spawn`. Ao contrário do ambiente Cilk, não existe uma fila de tarefas prontas para executar, e após o `MPI_Comm_spawn` delega-se o controle do processo para o sistema operacional. Desse modo, não é possível utilizar diretamente no MPI-2 o conceito de escalonamento visto no Cilk. Isto é, são necessárias alterações na aplicação ou no ambiente MPI para se ter o mesmo controle sobre a ordem de execução das tarefas.

4.3.2 Algoritmos de balanceamento de carga

Foram encontrados na bibliografia diferentes mecanismos de balanceamento de carga utilizando roubo de tarefas e alguns desses mecanismos podem ser adaptados para o ambiente MPI-2. Foram selecionados dois algoritmos para se detalhar nessa seção: o Roubo de Tarefas Aleatório (*Random Work-Stealing*) e o Roubo de Tarefas Aleatório para *clusters* (*Cluster-Aware Random Work-Stealing*). O critério de escolha para os algoritmos foi por ambos apresentarem bons resultados em aplicações *D&C* e terem uma implementação simples. Essas características os tornam bons pontos de partida para serem programados com MPI-2.

Random Work-Stealing: neste algoritmo, quando um processador verifica que sua fila de tarefas está vazia, ele faz uma tentativa de roubo de tarefa, escolhendo um processador de forma aleatória. Caso o pedido de roubo não seja respondido, repete-se o algoritmo até se obter uma resposta. O Cilk utiliza *Random Work-Stealing* para balancear a carga em máquinas SMP e é provado ser eficiente em

relação ao tempo, espaço e comunicação para computação no modelo *fully strict*[1]. Esse algoritmo também se mostrou eficiente em um *cluster* com o ambiente Satin (NIEUWPOORT; KIELMANN; BAL, 2001). A eficiência do mesmo algoritmo em dois ambientes diferentes é esperada, pois tanto em *cluster*, como em máquinas SMP, o custo para se fazer um pedido de roubo não varia muito em relação a qual processador foi escolhido.

Cluster-Aware Random Work-Stealing: foi proposto para adaptar o roubo de tarefas a ambientes com múltiplos *clusters* e interconexão heterogênea (WAN) (NIEUWPOORT; KIELMANN; BAL, 2001). Neste algoritmo, a idéia é a mesma do roubo aleatório de tarefas. Quando um processador faz um pedido de roubo para um *cluster* local, esse pedido é síncrono. Porém, quando um processador fora do *cluster* de origem é escolhido, o pedido de roubo é feito de forma assíncrona. Enquanto o processador espera a resposta do roubo, ele faz pedidos adicionais de roubo dentro do cluster. Este algoritmo traz as vantagens do *Random Work-Stealing*, mas mascara a latência dos pedidos distantes de roubos com pedidos locais de roubo.

No próximo capítulo, apresentam-se duas implementações de *Work-Stealing* que utilizam conceitos de ambos algoritmos de balanceamento de carga escolhidos.

4.4 Conclusões sobre o capítulo

Este capítulo apresentou o mecanismo básico de escalonamento oferecido pelo MPI-2 e diferentes formas de utilizar esse mecanismo, objetivando um melhor aproveitamento dos recursos disponíveis. Apresentaram-se dois escalonadores que foram desenvolvidos para apoiar a programação de qualquer tipo de aplicação que crie processos dinamicamente, mas que podem ser utilizados também no contexto *D&C* (Seção 4.2). Além de obter um melhor desempenho, os escalonadores automatizam o processo de escolha de recursos, tirando uma responsabilidade para o programador que o utiliza. Maiores detalhes sobre ambos escalonadores encontram-se em (CERA et al., 2006).

Foi apresentado o mecanismo de balanceamento de carga do Cilk e constatado que o MPI-2 não oferece um controle sobre os processos criados dinamicamente (Seção 4.1), como é feito no Cilk. Por esse motivo, não é possível utilizar diretamente no MPI-2 o conceito de *Work-Stealing* visto no Cilk. Dado o problema, são detalhados dois algoritmos para balanceamento de carga para verificar a possibilidade de implementá-los em MPI-2 (Seção 4.3.2).

O Capítulo 5 apresenta formas de modelar os conceitos e mecanismos apresentados na Seção 4.3 utilizando MPI-2. Em seguida, o Capítulo 6 apresenta resultados obtidos na validação dos modelos propostos, incluindo execuções de aplicações *Work-Pushing*, como apresentadas na Seção 4.2.

[1]Uma computação é chamada *fully strict* quando todas as arestas de dependência de dados apontam para o fluxo pai, como definido em (BLUMOFE; LEISERSON, 1994).

5 IMPLEMENTAÇÃO DE ROUBO DE TAREFAS PARA APLICAÇÕES D&C COM MPI-2

Este capítulo tem o objetivo de propor implementações de *Work-Stealing* para aplicações *D&C* com MPI-2 em *clusters* de computadores e em *cluster* de *clusters*. A Seção 5.2 apresenta uma implementação de *Work-Stealing*, que pode ser utilizada em um *cluster*, e a Seção 5.3 mostra como adaptar essa implementação para múltiplos *clusters*. Pode-se projetar implementações mais simples para execução em um único *cluster*, utilizando, por exemplo, o algoritmo de roubo de tarefas apresentado em (PACHECO, 1996). No entanto, como se tem o objetivo de utilizar ambientes heterogêneos com mais de um *cluster*, opta-se por uma implementação adaptável a ambos ambientes.

Antes de propor as implementações, a Seção 5.1 apresenta as escolhas de distribuição MPI e aplicação utilizada para validação da proposta.

5.1 Escolha da distribuição MPI e aplicação *D&C*

Esta seção apresenta testes feitos com algumas das principais distribuições MPI-2 encontradas atualmente, com o objetivo de verificar e comparar as funcionalidades oferecidas. Além disso, é detalhada a aplicação escolhida para ser implementada utilizando as diferentes propostas de modelos.

5.1.1 Teste das distribuições MPI-2 para D&C

O teste crucial para o emprego de uma distribuição MPI-2 para este trabalho é o funcionamento correto da primitiva `MPI_Comm_spawn`. A primeira distribuição MPI-2 testada, que apresentou resultados consistentes com o `MPI_Comm_spawn`, foi a LAM MPI, seguida do MPICH2.

Uma distribuição bastante promissora e que atualmente concentra atividade de desenvolvimento é o OpenMPI. No entanto, não se conseguiu obter resultados consistentes com o `MPI_Comm_spawn`.

Tabela 5.1: Suporte da primitiva `MPI_Comm_Spawn` nas principais distribuições MPI.

Distribuição MPI	Versão	Tempo $fib(8)$	Processos nó 1	Processos nó 2
LAM	7.1.2	1.392 s	66	0
MPICH2	1.0.4p1	2.341 s	33	33
OpenMPI	1.1.1	não executou	–	–

A Tabela 5.1 apresenta um dos testes realizados: a execução do programa *Fibonacci*, que foi apresentado na Seção 3.2.3. Executou-se $fib(8)$ com 2 nós de um *cluster*, sem utilizar nenhuma primitiva de escalonamento, medindo-se o tempo médio de 10 execuções e o número de processos criados em cada nó. O LAM obteve o menor tempo de execução, porém lançou todos os processos no mesmo nó. Já o MPICH2 levou mais tempo, porém dividiu igualmente os processos entre os dois nós disponibilizados. O OpenMPI compilou o *Fibonacci*, porém não terminou de executar em qualquer das tentativas. Foram testadas, sem sucesso, as versões 1.1.1 e a versão de desenvolvimento do OpenMPI, retirada do repositório SVN no dia 26/09/2006.

Outro teste realizado foi quanto à quantidade de processos MPI-2 que o LAM e o MPICH2 são capazes de executar. Para isso, foi utilizada a mesma aplicação teste *Fibonacci*. O teste consistiu em aumentar o parâmetro n, para forçar a criação de processos. O maior resultado calculado pelo MPICH2 foi $fib(8)$ com 33 processos por nó, enquanto o LAM executou $fib(10)$, mesmo lançando todos os 176 processos no mesmo nó.

Tanto o MPICH2 como o LAM apresentaram resultados consistentes e podem ser usados neste trabalho. O MPICH2 faz automaticamente um balanceamento *Round-Robin* e, para o LAM, mostrou-se no Capítulo 3 como obter o mesmo tipo de escalonamento. No entanto, nos testes realizados, o LAM suportou a execução de um número significativamente maior de processos que o MPICH2. Além disso, o LAM foi a primeira distribuição MPI a apresentar implementações estáveis da primitiva `MPI_Comm_Spawn` e, por esses motivos, o LAM foi escolhido para utilização neste trabalho.

Escolhida a distribuição MPI utilizada, é definida, a seguir, uma aplicação que pode ser resolvida recursivamente através de *D&C* e servirá de teste para implementação do roubo de tarefas.

5.1.2 Aplicação *D&C* recursiva: *N-Queens*

A aplicação escolhida para validar as implementações propostas é o problema *N-Queens* (*N-Rainhas*). Esse problema consiste na colocação de N rainhas em um tabuleiro de xadrez, de tamanho $N \times N$, de forma que não seja possível que uma rainha capture outra, em apenas uma jogada. Isto é, buscam-se configurações nas quais não existam mais de uma rainha em cada linha, coluna e diagonal. Existem muitas formas de se resolver esse problema, sendo o algoritmo de *backtrack*, desenvolvido através do modelo *D&C*, a forma mais utilizada. Neste trabalho apresentam-se duas implementações *D&C* do *N-Queens*. A primeira busca todas as possíveis soluções utilizando *backtrack*. A segunda, no Capítulo 6, apresenta uma versão que devolve apenas a primeira solução encontrada.

O algoritmo de *backtrack* utilizado consiste na colocação ordenada e exaustiva de rainhas linha a linha, até que configurações válidas sejam encontradas. Toda vez que determinado posicionamento possua rainhas em posição inválida, o algoritmo retorna à última configuração válida e prossegue avaliando novas posições. O código fonte seqüencial utilizado é o mesmo que o encontrado na implementação *N-Queens* que acompanha a distribuição do Satin e pode-se encontrar resultados de execuções em (NIEUWPOORT; KIELMANN; BAL, 2001).

Dado esse algoritmo seqüencial para resolver o problema, a paralelização do *N-Queens* utilizando a abordagem *D&C* é feita da seguinte maneira: cada processo

recebe uma configuração parcial de tabuleiro e gera processos para resolver cada possível posição de rainha na próxima linha. Esse procedimento é executado recursivamente até posicionar uma certa quantidade de rainhas e, então, as demais possibilidades são testadas de forma exaustiva e sem criação de novos processos.

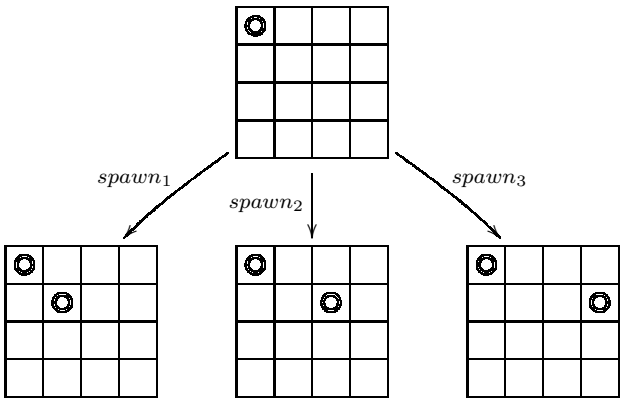

Figura 5.1: Exemplo de divisão recursiva do trabalho do *N-Queens* em novos processos.

Na Figura 5.1 representa-se um exemplo de divisão do trabalho em um dos níveis da recursão: o processo recebe uma determinada configuração de tabuleiro, gera as possíveis posições de rainha na próxima linha e cria novos processos para resolver cada uma das configurações de tabuleiro. Percebe-se que um balanceamento de carga homogêneo não é adequado para esta aplicação, pois não se pode prever a carga de cada processo gerado. Por exemplo, uma das configurações de tabuleiro na Figura 5.1 não pode resultar possíveis soluções para o problema, pois possui duas rainhas na mesma diagonal. O processo que receber esse tabuleiro irá descartá-lo, sem gerar novas tarefas. Enquanto as demais configurações poderão gerar novas tarefas e, por isso, podem ter um tempo maior de execução.

5.2 Work-Stealing hierárquico em *cluster*

A seguir, é proposta uma forma de implementar o roubo de tarefas utilizando criação dinâmica de processos do MPI-2 em um *cluster*. É apresentada uma forma de dividir as tarefas entre os processadores e o algoritmo de roubo de tarefas proposto. Além de poder ser utilizada em um *cluster*, essa mesma implementação tem o objetivo de ser adaptável para um ambiente de *cluster* de *clusters*, como é mostrado na Seção 5.3.

5.2.1 Divisão das tarefas

Para utilizar um mecanismo de roubo de tarefas, pode-se aproveitar a recursividade da aplicação e tratar cada chamada recursiva como um conjunto de tarefas. Essa divisão recursiva pode ser feita até se obter uma tarefa simples (como visto na Seção 2.1). A profundidade em que se deve entrar na recursão é uma questão que depende de cada aplicação e do grau de paralelismo oferecido pelo ambiente de execução.

A divisão recursiva das tarefas pode ser feita utilizando-se uma estrutura de uma árvore. O processo inicial (chamado de raiz) começa a execução com todo o problema e o divide, de modo que haja sub-problemas que possam ser resolvidos em paralelo. A solução trivial para dividir as tarefas, seria dividir igualmente todas as tarefas entre os processadores existentes. No entanto, ao criar novos processos, o MPI-2 permite que os processos criados comuniquem-se apenas com seus criadores. Essa limitação impede que, quando um processador encontre sua fila de tarefas vazia, faça um pedido de roubo diretamente para outro processador. Uma solução para esse problema seria unir todos os comunicadores com a primitiva `MPI_Intercomm_merge`. No entanto, essa solução foi descartada pela sobrecarga de manter um comunicador global sincronizado: cada chamada `MPI_Intercomm_merge` impõe uma barreira implícita entre todos os processos do comunicador. A solução proposta é que o processo raiz faça uma divisão igualitária de apenas uma parte das tarefas entre os processadores existentes. O restante das tarefas fica mantido no processo raiz para que, quando acabarem as tarefas de qualquer processador, possa-se fazer um pedido de roubo diretamente à raiz. Essa solução é semelhante ao modelo *Mestre/Escravo* e pode ter problemas de escalabilidade dependendo da aplicação e do ambiente de execução. Na Seção 5.3.2, propõe-se um mecanismo distribuído para dividir as tarefas em um *cluster*.

5.2.2 Etapas de execução

Pode-se resumir a execução de uma aplicação *D&C* com o modelo de *Work-Stealing* proposto em uma **fase inicial** de criação e distribuição de tarefas e uma **fase de roubo**, que faz o processamento efetivo das tarefas.

A **fase inicial** consiste em:

- criação do processo raiz (por exemplo, com `mpiexec`);
- criação de processos em cada processador (`MPI_Comm_spawn`);
- alocação de parte das tarefas para cada processador (enviadas com `MPI_Send`).

A **fase de roubo** consiste em um ciclo de:

- criação de processo folha (`MPI_Comm_spawn`), que resolve as tarefas;
- envio do resultado (`MPI_Send`).

Quando a fila de tarefas do processador estiver vazia, faz-se um pedido de roubo para o processo raiz (`MPI_Send`), que enviará novas tarefas enquanto existirem tarefas não alocadas. A idéia é semelhante ao roubo de tarefas aleatório, porém, com uma delegação de tarefas centralizada. Isto é, pode-se roubar aleatoriamente qualquer parte das tarefas, mas o pedido de roubo é feito sempre para o processo raiz.

A Figura 5.2 representa as etapas iniciais da execução. Na Etapa 1 cria-se o processo inicial, gerando as tarefas que serão executadas pelos demais processos. Na Etapa 2, o processo raiz cria um novo processo em cada processador disponível, através da primitiva *Spawn*. Na Etapa 3, o processo raiz envia parte das tarefas para cada processo criado. A Etapa 4 representa o retorno dos resultados parciais de cada processo, que ativa um pedido de roubo de tarefas para o processador de destino. Esse envio é feito assim que cada processador termina sua tarefa e, como o trabalho não é regular, os tempos de execução de cada um podem ser diferentes.

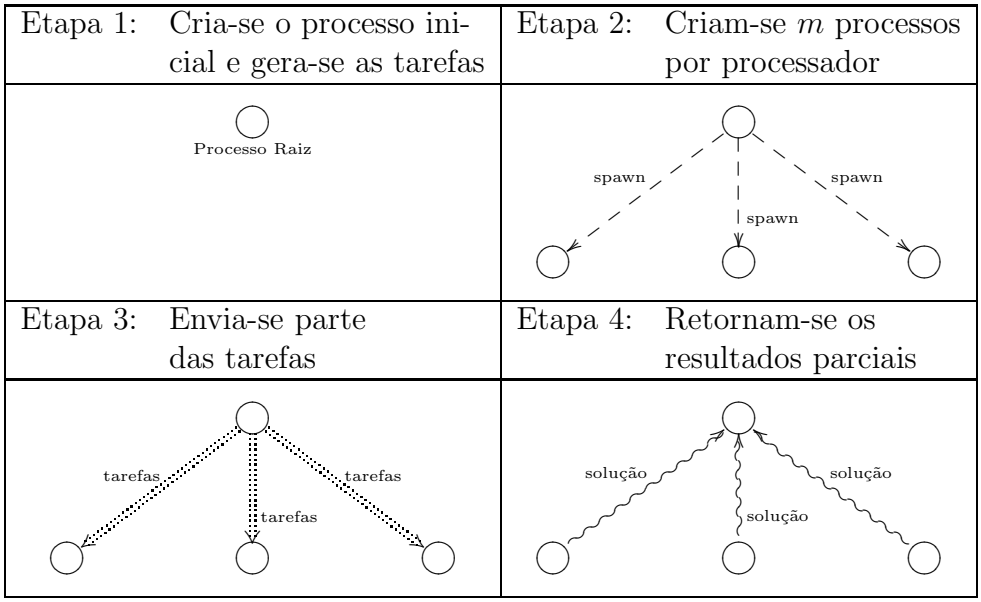

Figura 5.2: Etapas do roubo de tarefas em um *cluster*

O Algoritmo 5.2.1 representa um pseudo-código para roubo de tarefas em nível de *cluster*. No primeiro bloco de código (linhas 1 a 5) é criado o processo folha com `MPI_Comm_spawn`, enviadas as tarefas iniciais (`MPI_send`) e feito recebimento assíncrono de mensagem (`MPI_Irecv`), que serão tratadas posteriormente. Em nível de *cluster*, o laço `para` poderá ser executado para todos os nós disponíveis no ambiente de execução. O segundo bloco (linhas 6 a 26) é executado enquanto há tarefas para executar e processos folha com tarefas em execução. O código entre as linhas 16 a 23 representa o roubo de tarefas, que, nesse caso, é feito para o processo raiz. Quando o raiz responde um roubo com 0 tarefas, apenas aguarda-se o término dos processos folhas em execução.

Limitação do paralelismo: a Seção 3.2 apresentou o mecanismo de escalonamento oferecido pelo MPI-2 e, na Seção 4.3.1, verifica-se que no MPI-2 não se tem o mesmo controle do paralelismo que o Cilk oferece. Um processo criado pelo MPI-2 é controlado pelo sistema operacional e disputará o processador com todos os processos já lançados na mesma máquina. Por esse motivo, é interessante buscar meios para limitar o paralelismo, evitando que se executem muitos processos no mesmo processador e, conseqüentemente, minimizando a quantidade de trocas de contexto executadas pelo sistema operacional. Neste trabalho, propõe-se limitar em apenas 1 processo por processador disponível, para evitar que mais de um processo concorra pelo mesmo recurso.

Essa proposta para o roubo de tarefas já permite balancear a carga de aplicações *D&C* utilizando *Work-Stealing* em um *cluster*. A seguir, apresenta-se uma implementação de *Work-Stealing* que foca um ambiente com múltiplos *clusters*.

5.3 Work-Stealing hierárquico com MPI-2 em *clusters* de *clusters*

A Seção 4.3.2 apresentou um dos algoritmos de balanceamento de carga utilizado no ambiente Satin, o *Cluster-Aware Random Work-Stealing*. Ele foi proposto

Algoritmo 5.2.1 Algoritmo para roubo de tarefas em nível de *cluster*

```
1: para todos processadores disponíveis faça
2:   MPI_Comm_spawn(Procs[i])
3:   MPI_send(Procs[i], tarefas[i])
4:   MPI_Irecv(Procs[i], requisições[i], resultado_parcial[i])
5: fim para
6: enquanto folhas_vivas > 0 faça
7:   MPI_Waitany(requisições[])
8:   total+=resultado_parcial[i]
9:   folhas_vivas--
10:  tarefas[i] = aloca_tarefas(i)
11:  se tarefas[i].tamanho > 0 então
12:    MPI_Comm_spawn(Proc[i])
13:    folhas_vivas++
14:    MPI_send(Procs[i], tarefas[i].tamanho)
15:    MPI_Irecv(Procs[i], requisições[i], resultado_parcial[i])
16:    se tarefas_restantes = 0 & pai_tem_tarefas então
17:      MPI_send(pai, total) // pedido de roubo
18:      MPI_recv(pai, quantidade_tarefas)
19:      se quantidade_tarefas > 0 então
20:        MPI_recv(pai, tarefas[])
21:      senão
22:        pai_tem_tarefas=falso
23:      fim se
24:    fim se
25:  fim se
26: fim enquanto
```

para adaptar o roubo de tarefas a ambientes com múltiplos *clusters* e interconexão heterogênea (WAN). Aproveitando a idéia de considerar a localidade dos processadores no momento do roubo, propõe-se implementar um mecanismo semelhante no MPI-2. Utilizando-se a idéia de se diferenciar roubos "locais" e "remotos", pode-se aproveitar a estrutura de árvore para representar a localidade dos processos. Isto é, se um processo está próximo na árvore, assume-se que ele está numa rede próxima. Como os pedidos de roubo são feitos diretamente para o pai, que se encarrega de responder ou repassar o pedido, pode-se dizer que o roubo é feito antes localmente e caso não haja mais tarefas locais, será propagado um pedido de roubo remoto. A seguir, encontram-se maiores detalhes sobre o algoritmo de roteamento de tarefas.

Basicamente, segue-se a mesma implementação de *Work-Stealing* em *cluster*, porém acrescenta-se processos roteadores, que são responsáveis por subdividir as tarefas e rotear pedidos de roubo entre os processadores. Isto é, a divisão de tarefas, anteriormente feita pelo processo raiz, agora é feita recursivamente por um processo raiz e um processo que gerencia as tarefas em cada *cluster*.

5.3.1 Roteamento das tarefas na árvore de recursão

Ao executar uma aplicação recursiva, os processos podem: sub-dividir o trabalho e criar novos processos ou solucionar diretamente o problema. Um processo que gera

novos processos é chamado de **não folha** e um que soluciona diretamente o problema é chamado de **folha**.

Assim como na modelagem anterior, um pedido de roubo é representado por uma mensagem de envio dos resultados e é iniciado pelos processos folha. O que muda nessa modelagem é que acrescenta-se um novo nível capaz de responder diretamente ou rotear o pedido de roubo, caso necessário.

Algoritmo 5.3.1 Algoritmo de roteamento de tarefas para processos não folha na árvore de recursão.

```
1: para todos Processadores disponíveis faça
2:   MPI_Comm_spawn(Procs[i])
3:   MPI_send(Procs[i], tarefas[i])
4:   MPI_Irecv(Procs[i], requisições[i], resultado_parcial[i])
5: fim para
6: enquanto tem Procs com tarefas em execução faça
7:   MPI_Waitany(requisições[])
8:   total+=resultado_parcial[i]
9:   se mensagem.tag = fim então
10:    folhas_vivas--
11:  fim se
12:  se envio_msg_fim[i] = falso então
13:    tarefas[i]=aloca_tarefas(i)
14:    se tarefas[i].tamanho > 0 então
15:      MPI_send(Procs[i], tarefas[i].tamanho)
16:    senão
17:      envio_msg_fim[i]=verdadeiro
18:    fim se
19:    MPI_Irecv(Procs[i], requisições[i], resultado_parcial[i])
20:  fim se
21:  se tarefas_restantes = 0 & pai_tem_tarefas então
22:    MPI_send(pai, total) // pedido de roubo
23:    MPI_recv(pai, quantidade_tarefas)
24:    se quantidade_tarefas > 0 então
25:      MPI_recv(pai, tarefas[])
26:    senão
27:      pai_tem_tarefas=falso
28:    fim se
29:  fim se
30: fim enquanto
```

O Algoritmo 5.3.1 representa o pseudo-código para roteamento das tarefas em processos não folha. No primeiro bloco de código (linhas 1 a 5) são criados os filhos com `MPI_Comm_spawn` e feitos recebimentos de mensagens assíncronos (`MPI_Irecv`), que serão tratadas posteriormente. No processo raiz, o vetor `Procs` pode representar um processador de cada *cluster* e, no nível abaixo, todos os processadores do *cluster* local. O segundo bloco (linhas 6 a 30) é executado enquanto o processo tem processadores com tarefas em execução, tratando pedidos de roubo e avisando os filhos quando acabarem as tarefas.

Algoritmo 5.3.2 Algoritmo do mecanismo de *Work-Stealing*.

```
1: se sou folha então
2:   MPI_recv(pai, tarefasPorFolha)
3:   processa // código específico da aplicação
4:   MPI_send(pai, resultado)
5: senão
6:   se sou raiz então
7:     gera tarefas // código específico da aplicação
8:     gera lista de processadores (getntype)
9:   senão
10:    MPI_recv(pai, quantidade_tarefas)
11:    MPI_recv(pai, tarefas[])
12:    MPI_recv(pai, Procs[])
13:  fim se
14:  se np ≥ minProc × ramos então
15:    executa Algoritmo 5.2.1
16:  senão
17:    executa Algoritmo 5.3.1
18:  fim se
19:  se sou raiz então
20:    calcula resultado final
21:  senão
22:    MPI_send(pai, resultado, tag=fim)
23:  fim se
24: fim se
```

No Algoritmo 5.3.2, representa-se o pseudo código do mecanismo de *Work-Stealing* implementado. Entre as linhas 1 e 4, representa-se o caso mais simples de execução: o processo folha recebe tarefas, as processa e envia o resultado. Entre as linhas 5 e 23, representa-se o caso do processo não folha. Das linhas 6 a 13, faz-se a inicialização: se é raiz gera tarefas e a lista de processadores, caso contrário os recebe. Nas linhas de 14 a 18, escolhe-se se o processo subdivide suas tarefas e processadores ou não. Essa escolha é feita dependendo do número de processadores disponíveis para o processo e dos parâmetros de entrada (linha 14), executa-se o Algoritmo 5.2.1 ou 5.3.1.

5.3.2 Escalonamento distribuído de tarefas dentro de um *cluster*

Com o intuito de reduzir a sobrecarga do processo raiz causada pela geração de tarefas e pelo tratamento de pedidos de roubo, propõe-se uma forma de escalonamento distribuído de tarefas. Inicialmente, faz-se uma divisão recursiva das tarefas para aproximá-las dos processos que irão resolvê-las (processos folha). Essa divisão é feita em forma de árvore e tem o objetivo de manter tarefas no processo raiz, para que possam ser roubadas por qualquer processador. Com esta estratégia garante-se que os roubos acontecerão, em um primeiro momento, localmente e, caso faça-se necessário, remotamente (lembrando que o processo raiz coordena processos não folha, que por sua vez escalonam tarefas dentro de clusters).

A Figura 5.3 mostra o código do procedimento que faz o cálculo da quantidade

```
int qtdTarefasRamo(int np){
    if(np < minProc * ramos){
      return np * tarefasPorFolha * fatorRoubo;
    }else{
      int acumulador = 0;
      for(i=0; i< ramos;i++)
        acumulador += qtdTarefasRamo(np/ramos + (i<np%ramos?1:0));
      return acumulador * fatorRoubo;
   }
}
```

Figura 5.3: Código para cálculo do número de tarefas que serão enviadas inicialmente para cada ramo na recursão.

de tarefas que devem ser distribuídas inicialmente. Essa função é executada para calcular o número de tarefas que serão enviadas na linha 3 do Algoritmo 5.3.1. A árvore gerada por essa função recursiva é isomorfa à árvore de divisão de tarefas (Seção 5.2.1), pois utiliza os mesmo critérios. Os parâmetros e sua nomenclatura no código são os seguintes:

- número de processadores disponíveis no nível atual - chamado de `np`;
- número de ramos para divisão em cada nível da recursão - chamado de `ramos`;
- mínimo de processadores para o último nível - chamado de `minProc`;
- um fator multiplicativo para o cálculo do número de tarefas a serem destinadas a um processo não folha, o qual representa quantos pedidos de roubo por filho serão suportados localmente por ele - chamado de `fatorRoubo`;
- tarefas calculadas por processo folha - chamado de `tarefasPorFolha`.

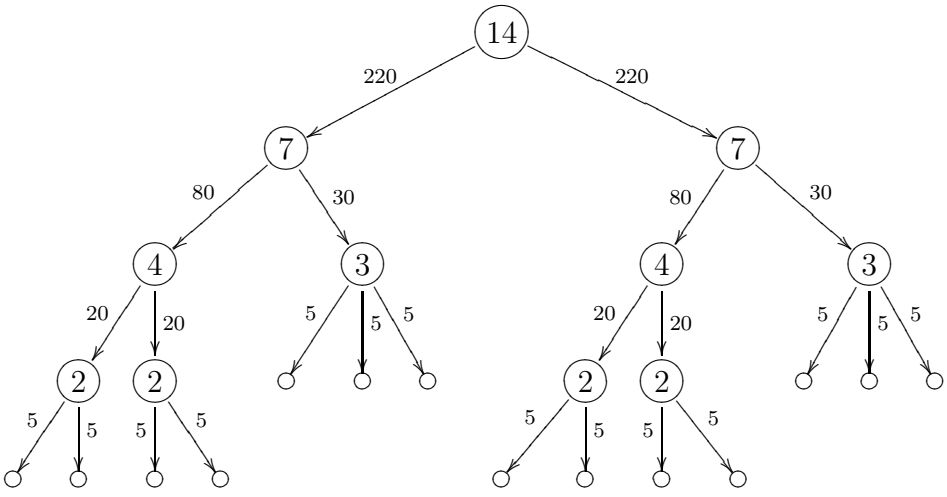

Figura 5.4: Exemplo de escalonamento distribuído de tarefas: nós numerados representam processos não folha e, os demais, folha. Parâmetros da execução: `np=14`, `ramos=2`, `minProc=2`, `tarefasPorFolha=5`, `fatorRoubo=2`.

Na Figura 5.4, representa-se um exemplo de distribuição recursiva de tarefas utilizando o algoritmo da Figura 5.3. Os valores nas arestas representam a quantidade de tarefas enviadas inicialmente e os valores dentro dos vértices representam a quantidade de processadores disponíveis (`np`). Nos processos do último nível, o valor `np` representa quantos processos folha serão criados pelo não folha.

A raíz é um processo não folha e recebe a totalidade de processadores disponíveis. Cada processo não folha recebe uma quantidade de processadores e faz a seguinte avaliação. Se essa quantidade de processadores puder ser dividida entre novos processos não folha, sendo um processo em cada ramo, de forma que nenhum desses processos receba menos processadores que `minProc`, então essa divisão é feita (de forma recursiva). Isto é, cria-se um processo não folha em cada ramo e os processadores são divididos entre esses novos processos. Nota-se que os primeiros `np mod ramos` processos criados recebem 1 processador a mais que os restantes, onde `mod` representa o resto da divisão inteira. Se a divisão de `np` por `ramos` for inteira, todos recebem a mesma quantidade de processadores. Caso a quantidade de processadores recebidos pelo processo não folha em questão for inferior a `minProc` $\times$ `ramos` (ou seja, se fosse dividida pela quantidade de ramos, algum processo não folha receberia menos processadores que `minProc`), então a divisão não é feita e este o processo não folha passa a criar processos folhas, um para cada processador que ele possui.

No exemplo da Figura 5.4, o processo raiz tem 14 processadores disponíveis. Como 14 dividido por 2 ramos é maior que 2 (`minProc`), dois novos processos não folha são criados, um em cada ramo, e cada um deles recebe 7 processadores. Para cada um desses dois, 7 dividido por 2 é maior que 2, então, novamente é feita a divisão entre dois novos não folha, um recebendo 4 e o outro, 3 processos. O não folha que recebeu 4 processadores ainda pode dividí-los e o faz para dois novos não folhas, cada um recebendo 2 processadores. Mas o não folha que recebeu 3 processadores, não pode dividí-los por 2 ramos, pois um deles teria apenas 1 processador, valor inferior a `minProc`. Com isso, cada processo que recebe 2 processadores cria 2 processos folha e cada processo que recebe 3 processadores cria 3 processos folha. Como `minProc=2`, não existe processo que crie apenas 1 folha[1].

Para as tarefas, a idéia do cálculo começa de baixo. Cada processo folha recebe `tarefasPorFolha` tarefas. Assim, um processo (não folha) que cria folhas, deve dispor de, pelo menos, `tarefasPorFolha` $\times$ `np` tarefas pois ele criará `np` folhas. Mas se um não folha tiver exatamente essa quantidade de tarefas, assim que criar todas as folhas, ficará sem tarefas na sua fila e, no primeiro pedido de roubo terá que propagar o pedido de tarefas para seu pai (que pode representar um pedido remoto). Para evitar isso, existe o `fatorRoubo`, que deve ser maior que 1. O processo não folha que criará processos folha, receberá `tarefasPorFolha` $\times$ `np` $\times$ `fatourRoubo` tarefas. Dessa forma, ao criar todas as folhas, ainda terá tarefas para atender alguns pedidos de roubo. O raciocínio é parecido para os outros processos não folha (aqueles que criam outros processos não folha). Um processo não folha enviará uma quantidade de tarefas para cada processo não folha que criar em cada ramo. As quantidades em cada ramo podem não ser iguais pois os ramos podem receber quantidades difirentes de processadores. A quantidade de tarefas que o processo não folha deve ter recebido para que não fique vazio após criar seus filhos, deve ser maior que o somatório das

[1] Aqui, observa-se que a variável `ramos` define a quantidade de ramos apenas na parte da árvore que lida exclusivamente com processos não folha. A quantidade de folhas criadas por um nó não folha é sempre maior ou igual a `minProc` e menor que `minProc` $\times$ `ramos`.

quantidades enviadas para cada ramo. Essa quantidade é `fatorRoubo` multiplicado pelo somatório.

No exemplo da Figura 5.4, cada folha recebe `tarefasPorFolha=5` tarefas. Todos os nós não folha que têm 2 processadores, ao criarem as folhas, irão enviar 10 tarefas, sendo 5 para cada folha. Multiplicando-se 10 por `fatorRoubo`, o resultado é 20 e essa é a quantidade que esse não folha recebe de seu pai. Da mesma forma, os não folha que têm 3 processadores, de início, enviam 15 tarefas, portanto recebem 30. Os não folha que têm 4 processadores, de início enviam 40 tarefas, portanto recebem 80. Por fim, os que enviam $80 + 30 = 110$, recebem 220.

5.4 Estudo de caso: implementação do *N-Queens* com Work-Stealing

Seguindo a modelagem *D&C* proposta, o processo raiz inicia com o tabuleiro vazio e gera todas as configurações possíveis de tabuleiro para uma quantidade x de rainhas. Após, o processo raiz cria np processos (folhas) e envia uma quantidade z_i de tabuleiros (tarefas) para cada processo i criado. Sendo que x é um parâmetro a ser escolhido, np representa a quantidade de processadores disponíveis e cada z_i é obtido pela função `qtdTarefasRamo` (Figura 5.3).

Os processos folha recebem uma quantidade de tabuleiros com x rainhas já posicionadas, calculam as possíveis soluções, enviam o resultado e terminam. O envio do resultado é interpretado pelo processo raiz como um pedido de roubo de tarefas. O processo raiz cria, então, um novo processo e envia novas configurações de tabuleiros. Repete-se o processo até que todo sub-espaço do problema tenha sido percorrido e o resultado final é calculado pelo processo raiz através da soma das soluções parciais.

No *N-Queens* é possível gerar tarefas suficientes para ocupar todos os processadores em apenas um nível da recursão. Por isso, optou-se por gerar as tarefas no primeiro nível (processo raiz) e fazer uma distribuição das tarefas entre os ramos da recursão. Resultados de execuções dessa implementação e comparações com outras versões do *N-Queens* serão apresentados no Capítulo 6.

5.5 Conclusões sobre o capítulo

Neste trabalho, propõe-se utilizar MPI-2 para implementar um mecanismo de *Work-Stealing*, similarmente ao que é feito nos ambientes Cilk e Satin. Inicialmente, propõe-se um mecanismo de *Work-Stealing* na Seção 5.2, que permite fazer roubo de tarefas dentro de um *cluster*. Em seguida, na Seção 5.3, propõe-se um mecanismo de *Work-Stealing* que contempla ambientes com múltiplos *clusters*. O mecanismo proposto é hierárquico e utiliza o conceito de roubo local e remoto, como é prosposto pelo Satin em (NIEUWPOORT; KIELMANN; BAL, 2001). O roubo de tarefa é local quando o processo do nível acima (pai) responde diretamente o pedido de roubo. Caso contrário, o pai faz roteamento do roubo, caracterizando um roubo remoto. Utilizando esse mesmo mecanismo hierárquico, propõe-se descentralizar a distribuição das tarefas dentro de cada *cluster* (Seção 5.3.2).

Empregando-se as técnicas de roubo de tarefas descrita neste capítulo, desenvolveu-se uma versão de aplicação *N-Queens*. O próximo capítulo detalha os resultados obtidos na validação dessa implementação. Para complementar a análise, serão feitas

comparações entre diferentes versões da aplicação com MPI-2 e também com uma versão implementada no ambiente Satin.

6 APRESENTAÇÃO E AVALIAÇÃO DOS RESULTADOS

Neste capítulo, apresenta-se e analisa-se o desempenho obtido com a implementação de *Work-Stealing* para validação da modelagem proposta. Inicialmente, faz-se algumas medições sobre o custo de criação de processos por `Spawn` no MPI-2, incluindo comparações com custo da mesma primitiva no Cilk (Seção 6.1). Após, compara-se o balanceamento de carga obtido em uma aplicação utilizando escalonamento *Work-Pushing* e *Work-Stealing* (Seção 6.2). Em seguida, valida-se o desempenho da implementação MPI-2, através de uma comparação com um ambiente consolidado para programação *D&C* em ambientes com memória distribuída (Seção 6.3). Por fim, verifica-se o funcionamento de algumas das características oferecidas pelo modelo de *Work-Stealing* proposto (Seção 6.4).

Os tempos médios nas execuções apresentadas neste capítulo foram obtidos com no mínimo 4 execuções para execuções que levam mais do que $1200s$ e 10 execuções para os demais casos. As variações no tempo de execução foram menores do que 3%, salvo raras exceções. O *cluster* utilizado foi o LabTec, que possui 20 nós biprocessados Pentium III $1.1GHz$ com 1 Gb de memória RAM e rede de interconexão *Giga Ethernet*. O sistema operacional utilizado foi o GNU/Linux (distribuição Debian Sarge) e a distribuição MPI foi a LAM 7.1.2. Em todos os experimentos foi utilizada uma quantidade diferente de nós do *cluster* LabTec, exceto na Seção 6.4 onde utiliza-se também nós de outro *cluster*.

6.1 Custo de criação de processos por `Spawn`

No MPI-2, um `Spawn` executa uma criação remota de processo, que envolve transferência de dados via rede. Essa transferência, em conjunto com a execução de um processo pesado, acrescenta uma sobrecarga em relação ao `Spawn` do Cilk, que é executado através de uma *thread*. As seções seguintes apresentam medições e comparações que têm o objetivo de verificar o custo e o impacto da utilização do `Spawn` em aplicações MPI-2.

6.1.1 Comparando Cilk x MPI-2

Para comparar os dois ambientes, foi utilizado o exemplo do problema *N-Queens* fornecido pelo ambiente Cilk (versão 5.4.2.3). Essa implementação retorna apenas a primeira solução possível encontrada e foi programada de forma recursiva. A função recebe como argumento um tabuleiro com as rainhas posicionadas até o momento e, para cada nova posição válida de rainha, executa recursivamente a função com a

nova configuração de tabuleiro. Quando N rainhas estiverem posicionadas, a função retorna a solução e termina os demais fluxos de execução. No Cilk, cada fluxo gerado por um `spawn` é implementado por uma *thread* com memória compartilhada. No MPI-2, cada fluxo é implementado por um processo que se comunica por troca de mensagens, no início e no fim da execução, conforme descrito na Seção 3.1.

Tabela 6.1: Execuções do *N-Queens* MPI-2 baseado no exemplo do ambiente Cilk ($N = 7$).

	Núm. processos criados	tempo(ms)	Tempo/criação(ms)
	64	1100	17,19
	74	1120	15,14
	84	1100	13,10
	83	1100	13,25
	89	1120	12,58
	92	1200	13,04
	111	1260	11,35
Média	85,29	1142,86	13,66

A execução do *N-Queens* com $N=7$ no ambiente Cilk tem tempo médio de $3ms$ em uma máquina bi-processada. Já com MPI-2, a execução tem um tempo médio de $1142ms$ utilizando 5 máquinas bi-processadas. A Tabela 6.1 mostra a quantidade de processos MPI-2 criados, o tempo total de execução e uma estimativa do tempo para a criação de cada processo MPI-2. Essa estimativa foi feita desconsiderando-se o trabalho executado em cada processo, que é muito pequeno comparado ao custo de criação remota de processos - pois envolve transferência do executável e troca de mensagens pela rede.

Com os resultados obtidos foi possível adaptar-se uma aplicação restrita a ambientes com memória compartilhada para ambientes com memória distribuída, que possuem maior escalabilidade. Porém, os tempos apresentados evidenciam um fator crucial na paralelização de aplicações: é necessário fazer um balanceamento adequado entre o tempo de processamento e o tempo de criação de processos. Não se obtém nenhum ganho criando processos pesados para granularidade muito pequena de trabalho.

6.1.2 Comparando *Work-Stealing* com `Spawn` e por troca de mensagens

Esse experimento tem o objetivo de verificar, de forma prática, o custo da criação de processos para fazer o roubo de tarefas. Alterou-se o modelo de *Work-Stealing* para que cada roubo seja feito apenas por troca de mensagens. Isto é, quando um processo folha termina suas tarefas, ao invés de se fazer `Spawn` de um novo processo no processador ocioso, apenas se envia uma mensagem com novas tarefas para o processo existente. Essa comparação é interessante para refletir sobre a necessidade de alterar a quantidade de vezes que uma aplicação executa um `Spawn`. É preciso definir uma granularidade de tarefa adequada para aplicações *D&C* no MPI-2, para se ter um bom balanceamento entre processamento útil da aplicação e processamento das primitivas de paralelismo.

De acordo com a alteração no modelo, um mesmo processo pode executar um ou mais roubo de tarefas, como pode ser visto na comparações na Figura 6.1. Neste

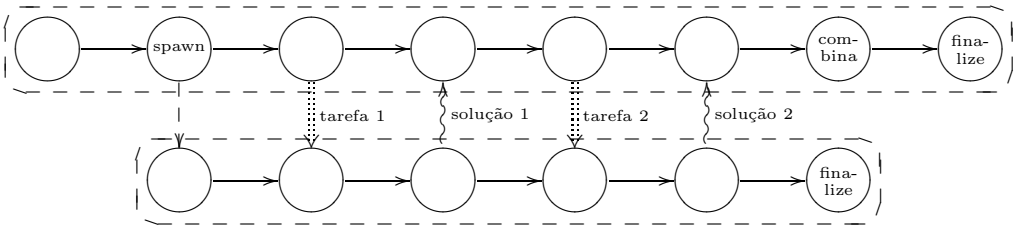

Figura 6.1: Processo *D&C* resolvendo mais de uma tarefa.

experimento utiliza-se **troca de mensagem a cada roubo**, que funciona de forma similar ao roubo *Work-Stealing* apresentado no Capítulo 5: um processo raiz cria um processo por processador utilizando `Spawn` com parte das tarefas; após terminar seu trabalho e retornar o resultado parcial, o processo aguarda novas tarefas. O processo raiz, ao receber um resultado, detecta que o processador está ocioso e envia de volta novas tarefas. Quando acabarem as tarefas no processo raiz, ele envia, ao invés de novas tarefas, uma mensagem para avisar não existe mais trabalho na fila e aguarda o retorno dos demais processos em execução.

Tabela 6.2: Comparando execuções com roubo de tarefas por `spawn` e por troca de mensagens.

Tarefas por roubo	Tempo	
	Troca de mensagens	Spawn
20	267,31s	268,52s
10	259,49s	264,28s
5	253,03s	263,38s

Na Tabela 6.2, apresentam-se os tempos de execução do *N-Queens* com $N = 19$ em 8 nós do *cluster*. Percebe-se que a diferença nos tempos é pequena. No entanto, a diferença aumenta quando se reduz muito a granularidade do roubo.

Com os resultados apresentados, conclui-se que a utilização do roubo de tarefas com criação de processos é viável, porém, é preciso ajustar a granularidade do roubo de tarefas.

6.2 *D&C* com MPI-2: comparando escalonamento *Work-Pushing* e *Work-Stealing*

A utilização de *Work-Pushing* em programas *D&C* com MPI-2 não exige restrições na modelagem da aplicação, pois consiste apenas de um mecanismo de escolha dos recursos para lançamento dos processos. A seguir, é detalhada a implementação paralela utilizada para executar *N-Queens* com *Work-Pushing*.

O processo inicial gera todas as configurações possíveis de tabuleiro posicionando uma quantidade x de rainhas. Criam-se, então, novos processos e envia-se uma quantidade y de tabuleiros até que todas as configurações possíveis tenham sido percorridas. Os processos recém criados recebem os tabuleiros, processam e enviam os resultados parciais. Após, o processo inicial aguarda até que todos os resultados parciais sejam enviados, soma e retorna o resultado final. Essa modelagem é bastante simples e não há garantias de uma boa distribuição de carga: o número de processos

criados é obtido em função dos parâmetros de entrada e não considera-se o número de processadores disponíveis.

O mecanismo de escalonamento escolhido é centralizado e utiliza um algoritmo de *Round-Robin* para distribuir os recursos, como descrito na Seção 4.2.1. A integração do *N-Queens* com um escalonador do tipo *Work-Pushing* é direta: ao encontrar um `MPI_Comm_spawn`, consulta-se o escalonador para escolher onde lançar o novo processo.

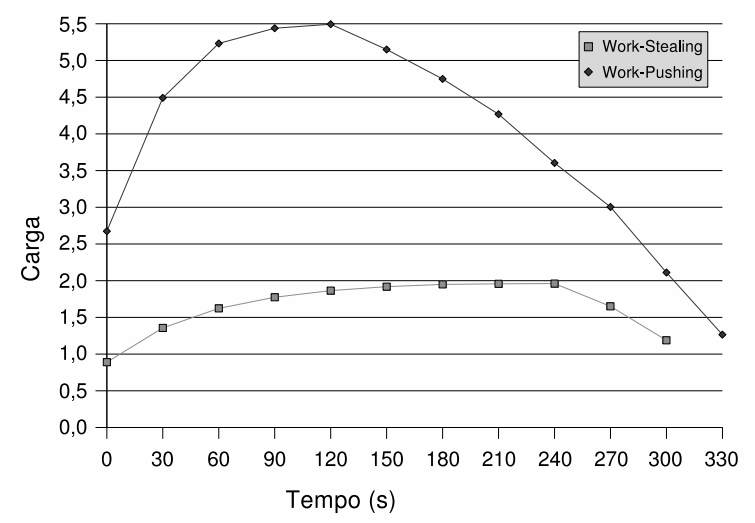

Figura 6.2: Gráfico da carga média de todos nós durante toda a execução do *N-Queens* com *Work-Pushing* (Round-Robin) e *Work-Stealing*. Cada valor representa a média das cargas dos 7 nós durante o último minuto.

Comparou-se essa implementação do *N-Queens* com *Work-Pushing* com uma implementação do *N-Queens* com *Work-Stealing* em *cluster* (proposta na Seção 5.2). Foram feitas execuções com $N = 19$, utilizando 7 nós do *cluster*. Utilizou-se os mesmos parâmetros x e a mesma granularidade para criação de processos de cálculo: no *Work-Pushing* utilizou-se $y = 30$ e no *Work-Stealing* `tarefasPorFolha`$= 30$. Desse modo, no *Work-Pushing* criam-se todos os processos de cálculo (com 30 tabuleiros) no inicio da execução. Já no *Work-Stealing*, cria-se um processo de cálculo (também com 30 tabuleiros) por processador e, quando um processo retorna seu resultado, cria-se outro processo no mesmo processador. Os tempos médios de execução com essas configurações não foram muito diferentes: $320s$ com *Work-Pushing* e $316s$ com *Work-Stealing*. Durante a execução, foi medida a carga média no último minuto, de cada processador utilizado. A carga média no último minuto é apenas uma aproximação que representa quantos processos estão competindo pelo processador. Esse valor foi obtido a cada intervalo de $30s$, acessando-se o `/proc/loadvg`. Na Figura 6.2, encontra-se um gráfico gerado utilizando-se a média das cargas de cada processador ao longo da execução do *N-Queens*. Nota-se que há uma sobrecarga dos processadores no início da execução com *Work-Pushing*, enquanto no *Work-Stealing* a carga tende a ficar próxima de 2, que é a quantidade de processadores disponível em cada nó. O desvio padrão máximo no *Work-Pushing* foi de $1,48$ e no *Work-Stealing* foi de $0,38$, o que mostra um balanceamento mais homogêneo no *Work-Stealing*.

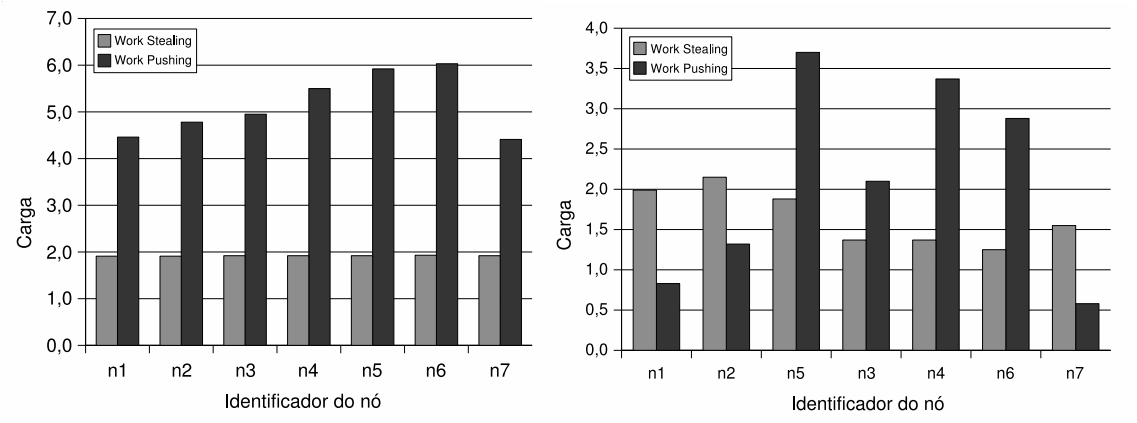

Figura 6.3: Gráfico da carga média dos nós em dois instantes da execução do *N-Queens* com *Work-Pushing* (Round-Robin) e *Work-Stealing*. O gráfico à esquerda foi obtido aos 150*s* da execução e o à direita, aos 300*s*. Os valores representam a carga média individual de cada nó no último minuto.

Com o objetivo de analisar mais detalhadamente a carga de cada nó, a Figura 6.3 mostra as cargas médias individuais obtidas em duas situações: à esquerda, aos 150*s* e, à direita, aos 300*s* da execução. Percebe-se que, aos 150*s*, o balancamento de carga no *Work-Stealing* é homogêneo entre todos os nós, enquanto no *Work-Pushing* a carga varia entre $4, 5$ e 6. Na direita, representam-se valores obtidos um pouco antes do término da execução. Percebe-se que no *Work-Pushing* existem nós com mais processos do que processadores enquanto outros nós estão subutilizados. Já no *Work-Stealing*, também verificam-se nós subutilizados, porém a diferença da carga entre os nós é menor. Uma alternativa para reduzir essa diferença de carga no final da execução é diminuir o grão da tarefa em cada processo. Essa alteração pode ser feita tanto no *Work-Stealing* como no *Work-Pushing*, porém no *Work-Pushing* acarretaria no aumento do número de processos competindo pelo processador ao mesmo tempo e, conseqüentemente, o aumento de trocas de contexto pelo sistema operacional.

6.3 Comparando as implementações MPI-2 e Satin

Com o objetivo de validar o mecanismo de *Work-Stealing* proposto, buscou-se uma implementação do *N-Queens* em um ambiente já consolidado para comparação: o Satin. A implementação Satin do *N-Queens* baseia-se no mesmo código seqüencial utilizado na versão MPI-2[1] e pode ser encontrada no pacote de aplicações Ibis[2], no diretório `nqueens_contest`. A versão utilizada do Ibis, que acompanha o Satin, foi a 1.4.

6.3.1 Ajustando os parâmetros de execução do *N-Queens* no Satin

A aplicação *N-Queens* do Satin precisa apenas ajustar um parâmetro: até qual nível de profundidade deve ser feito `Spawn`. Apenas alterando esse parâmetro e

[1]http://www.ic-net.or.jp/home/takaken/e/queen/nqueens.c
[2]http://www.cs.vu.nl/ibis/downloads.html

analisando as estatísticas geradas pelo ambiente, é possível saber se a aplicação está tendo um bom aproveitamento da máquina paralela.

Tabela 6.3: Execuções do *N-Queens* no Satin com diferentes profundidades de `Spawn`. O tamanho de tabuleiro utilizado foi 20 e o número de nós do cluster foi 20.

Nível Spawn	Número de Spawns	Pedidos Roubo	Roubos Efetivos		Tempo útil	Sobre-carga	Tempo total
1	163	35.213.228	122	0,00%	1291,76	458,92	1750,68
1	163	25.529.894	111	0,00%	1279,65	324,91	1604.55
2	2.175	3.009.875	552	0,01%	1278,65	43,75	1234,90
3	24.889	120.388	1.693	1,40%	1230,92	5,40	1236,31
4	246.482	11.868	2.056	17,32%	1229,93	3,00	1232,92
5	2.126.480	4.176	1.992	47,70%	1230,34	2,84	1233,18
6	15.883.558	5.195	2.747	52,87%	1232,96	2,85	1235,81

Na Tabela 6.3, apresentam-se os resultados de execuções com diferentes valores para o nível de `Spawn` do *N-Queens* no Satin. Comparam-se os números de `Spawns` executados, tentativas de roubo, roubos efetuados com sucesso, o tempo útil para aplicação (médio por máquina), o tempo de sobrecarga (médio por máquina) e o tempo total de execução. Aparecem duas execuções com profundidade 1 para ilustrar que houve uma variação grande no tempo de execução com o mesmo argumento, o que não aconteceu com profundidades maiores. A partir dos resultados apresentados, percebe-se que a partir da profundidade 3, os tempos de execução ficam bem próximos e o Satin consegue balancear eficientemente a carga da aplicação. Nos demais casos a variação

6.3.2 Ajustando os parâmetros de execução do *N-Queens* no MPI-2

No MPI-2, ainda não se tem implementados os mesmos recursos para análise das execuções de uma aplicação *D&C* que o Satin. O parâmetro mais significativo para a implementação MPI-2 é o número de tarefas que cada processo folha deve processar numa chamada `Spawn`. O ajuste de parâmetros foi feito em função dos tempos de execução obtidos.

Tabela 6.4: Execuções do *N-Queens* no MPI-2 com diferentes quantidade de tarefas por processo folha. Melhores parâmetros para tamanhos de tabuleiro 18 a 20.

Tamanho do tabuleiro	Total de tarefas	Tarefas por folha
18	1354	40
19	1581	20
20	2012	10

A Tabela 6.4 apresenta os melhores valores encontrados para os tamanhos de tabuleiro de 17 a 20, executando-se com diferentes quantidades de nós do *cluster*. O

total de tarefas representa o número de tabuleiros pré-marcados que foram gerados em cada execução.

6.3.3 Comparação entre os tempos de execução

Após ajustar os parâmetros de execução até obter os melhores tempo de execução, fez-se uma comparação entre execuções do mesmo problema utilizando o mesmo ambiente.

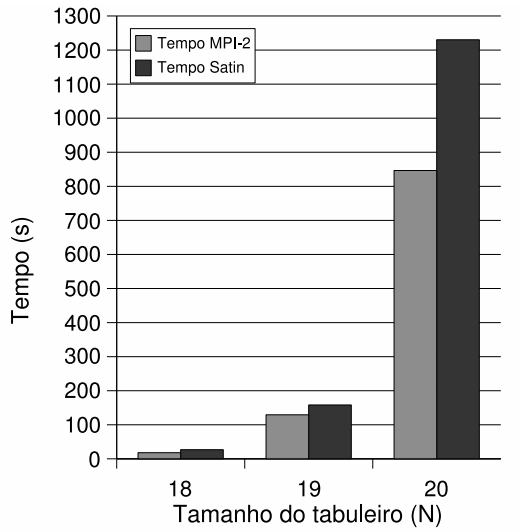

Figura 6.4: Comparação do *N-Queens* com MPI-2 e Satin: execução com tabuleiro de tamanhos 18, 19 e 20 com 20 nós do *cluster*.

Na Figura 6.4, comparam-se os tempos de execução do *N-Queens* utilizando tamanhos de tabuleiro 18, 19 e 20 com 20 nós do *cluster*. Apesar do ganho com MPI-2 ser pequeno nos problemas menores, com o tabuleiro de tamanho 20 obtém-se um ganho de 30% com MPI-2 em relação ao tempo de execução no Satin.

Na Figura 6.5, comparam-se os tempos de execução do *N-Queens* com $N = 18$, porém com diferentes quantidades de nós do *cluster*. Novamente, o tempo de execução com MPI-2 é menor em todas as configurações de ambiente utilizadas.

6.4 Verificação de características oferecidas pelo modelo de *Work-Stealing* proposto

Esta seção apresenta execuções do *N-Queens* com MPI-2 para verificar o funcionamento de algumas das características esperadas nessa implementação de *Work-Stealing*: escalonamento distribuído de tarefas, utilização em ambiente heterogêneo e o aproveitamento de recursos disponibilizados dinamicamente.

6.4.1 *Work-Stealing* com escalonamento distribuído de tarefas

Na Seção 5.3.2, propõe-se uma forma de escalonar as tarefas de forma distribuída dentro de um cluster, com o objetivo de reduzir a sobrecarga do processo raiz. Pode-se distribuir a geração de tarefas e o tratamento de pedido de roubos. Porém, a geração de tarefas na aplicação de teste tem um custo muito baixo em comparação

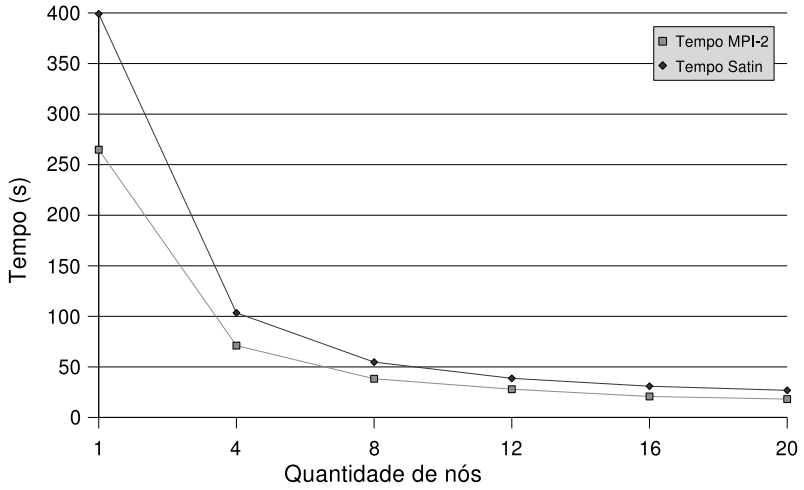

Figura 6.5: Comparação do *N-Queens* com MPI-2 e Satin: execução com tabuleiro de tamanho 18 com diferentes quantidades de nós do *cluster*.

ao trabalho total, então opta-se por manter a geração no processo raiz e distribuir apenas os pedidos de roubo.

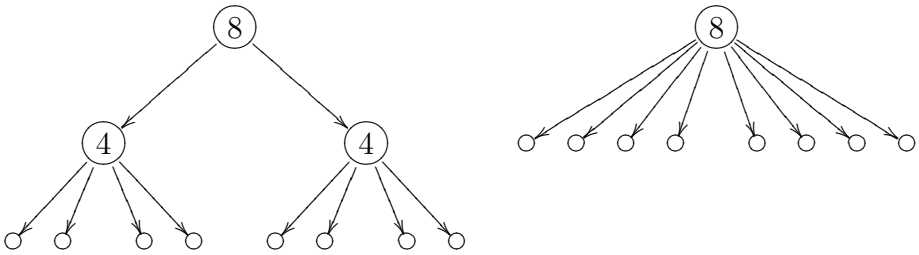

Figura 6.6: Hierarquias para controle de roubos de tarefas distribuído (à esquerda) e centralizado (à direita) utilizadas para execuções dentro de um *cluster*. Os valores nos vértices representam a quantidade de processadores disponíveis para o processo. Os vértices sem valor representam os processos folha.

Executou-se o *N-Queens* com $N = 19$ distribuindo-se o controle do roubo de tarefas em 4 nós do *cluster* e comparou-se com a execução utilizando controle centralizado. Na Figura 6.6, representam-se a hierarquia distribuída (à esquerda) e a centralizada (à direita). Em 5 execuções com controle distribuído, a média do tempo foi de $71,10s$ e, de forma centralizada foi de $71,03s$. O objetivo dessa comparação é verificar o impacto de se acrescentar um nível intermediário de processo para fazer o controle e roteamento de tarefas.

Verificou-se que a diferença nos tempos de execução é pequena, o que demonstra ser viável utilizar esse mecanismo em aplicações que necessitem distribuir a geração de tarefas ou em ambientes de maior escala. No entanto, são necessários mais testes para se comprovar que é possível melhorar o desempenho da aplicação utilizando esse mecanismo.

6.4.2 *Work-Stealing* em ambiente heterogêneo

Com o objetivo de mostrar que o mecanismo de *Work-Stealing* proposto implementado com MPI-2 consegue tirar proveito de recursos heterogêneos, executou-se o *N-Queens* com $N = 20$ em 20 nós do *cluster* LabTec e 14 nós do *cluster* Corisco. A rede de interconexão que liga os nós LabTec é *Giga Ethernet*, enquanto a ligação entre os nós Corisco e entre os dois *clusters* é *Fast Ethernet*. Cada nó LabTec possui dois processadores Pentium III $1266MHz$ e os nós Corisco têm dois processadores Pentium III $1100MHz$.

Tabela 6.5: Execuções do *N-Queens* no MPI-2 com $N = 20$ em 20 nós LabTec e juntando 34 nós de dois *clusters* diferentes.

Tempo 20 nós (LabTec)	Tempo 34 nós (LabTec+Corisco)
846,40s	514,53s

A Tabela 6.5 mostra os tempos de execução nos ambientes com 20 e 34 nós. Obteve-se um ganho de aproximadamente 40% no tempo de execução em relação ao ambiente homogêneo, mostrando que pode-se obter bom desempenho em um ambiente heterogêneo.

6.4.3 *Work-Stealing* com aproveitamento de recursos disponibilizados dinamicamente

Uma das características dessa implementação de *Work-Stealing* é que pode ser facilmente adaptada para utilizar recursos disponibilizados dinamicamente. O LAM permite acrescentar novos recursos ao ambiente de execução MPI através do comando `lamgrow`. Para utilizar os novos recursos, deve-se verificar o total de processadores disponíveis ao longo da execução. Essa verificação pode ser feita através da primitiva LAM `getntype(0,0x02)`, que retorna o número total de nós no ambiente de execução.

Objetivando exemplificar esse aproveitamento de recursos foi executado o *N-Queens* com tabuleiro de tamanho 19, iniciando-se com 1 nó do *cluster*. A cada minuto foram acrescentados dois novos nós, até os 5 minutos da execução. Na Figura 6.7, tem-se uma representação gráfica do número de `Spawns` executados em cada nó. O número de tarefas executadas é proporcional ao número de `Spawns` executados em cada nó, exceto no caso do $n0$, que recebe processos extras. Esses processos são necessários devido a uma limitação encontrada ao executar o `lamgrow`: como apenas processos criados após a adição de novos nós conseguem disparar processos em nós adicionados, a cada vez que identifica-se uma inserção de recurso, é necessária a criação de um processo que será capaz de criar outros processos nos nós adicionados. No caso da Figura 6.7, esses processos extras foram criados em $n0$.

Na Tabela 6.6, apresenta-se o número total de `Spawns` executados em cada nó logo antes de cada acréscimo de novos recursos. O tempo total dessa execução foi de $410s$, enquanto o tempo médio da execução do mesmo problema em apenas um nó é de $1998s$. Com estes resultados, conclui-se que é possível empregar *Work-Stealing* em ambientes dinâmicos e obter-se um desempenho satisfatório.

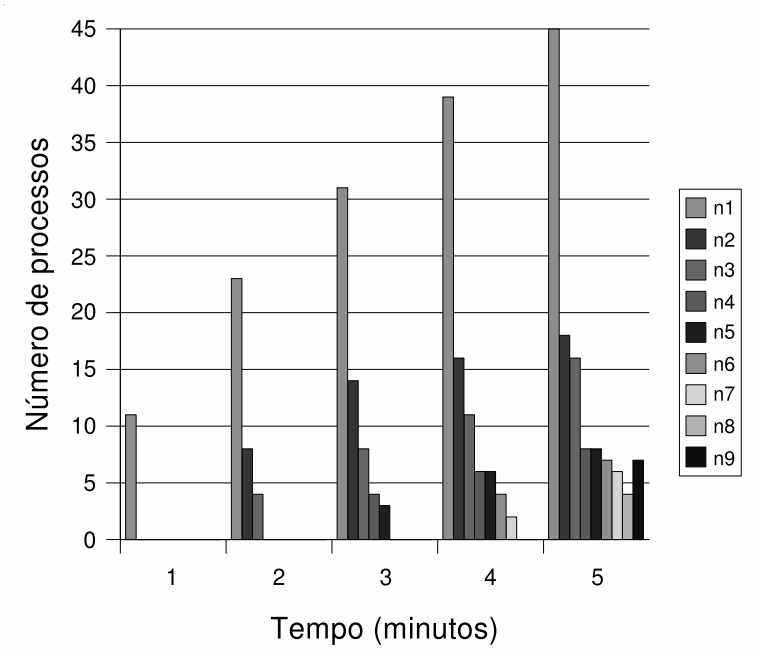

Figura 6.7: Gráfico da quantidade de processo criados com a primitiva `Spawn` em cada nó ao longo da execução.

Tabela 6.6: Número de processos lançados em cada nó em execuções do *N-Queens* no MPI-2 com aumento do número de nós disponíveis ao longo da execução.

Tempo	Nó 1	Nó 2	Nó 3	Nó 4	Nó 5	Nó 6	Nó 7	Nó 8	Nó 9
1 min	11	0	0	0	0	0	0	0	0
2 min	23	8	4	0	0	0	0	0	0
3 min	31	14	8	4	3	0	0	0	0
4 min	39	16	11	6	6	4	2	0	0
5 min	45	18	16	8	8	7	6	4	7

6.5 Conclusões sobre o capítulo

Com os resultados apresentados neste capítulo, mostrou-se que a utilização de *Work-Stealing* com os recursos oferecidos pelo MPI-2 pode trazer bons resultados. Mostrou-se que é possível obter um melhor balanceamento de carga em comparação com um mecanismo mais simples de escalonamento, o *Work-Pushing*. Comparou-se o desempenho da implementação proposta com a mesma aplicação desenvolvida no ambiente Satin. Além disso, mostrou-se que a utilização de um mecanismo de escalonamento distribuído não impacta negativamente no desempenho da aplicação exemplo. A distribuição do escalonamento pode ser interessante para outras aplicações e para ambientes de maior escala do que o ambiente de teste. Verificou-se que a implementação é adaptada para ambientes heterogêneos e para aproveitamento de recursos disponibilizados dinamicamente.

7 CONSIDERAÇÕES FINAIS

A seguir, resumem-se as principais contribuições obtidas a partir deste trabalho e os passos seguidos para atingir os objetivos propostos.

7.1 Contribuições

Este trabalho iniciou com um levantamento dos principais ambientes de programação paralela que suportam *D&C* e identificou-se as características mais importantes de cada um. Com as recentes implementações da norma MPI-2, que prevê criação dinâmica de processos, verificou-se que, apesar do MPI-2 não ter sido especificamente projetado para o modelo de programação *D&C*, é possível utilizar os recursos oferecidos para programar esse tipo de aplicação. Uma vez que o MPI é um ambiente eficiente de programação paralela e amplamente utilizado, surge o interesse no estudo de como se utilizar seus novos recursos para se ter um novo ambiente de programação para aplicações *D&C* dinâmicas.

Propõe-se, então, um exemplo de modelagem para programar as aplicações *D&C* utilizando MPI-2. No entanto, o MPI-2 não oferece nenhum mecanismo automático de escalonamento, como é o caso de ambientes eficientes como Cilk e Satin. Dado o problema, estuda-se os mecanismos de escalonamento existentes nesses ambientes (vide Tabela 7.1) e propõe-se uma implementação MPI-2 de *Work-Stealing* para aplicações *D&C*. Essa proposta tem como principais objetivos: obter um bom balanceamento de carga e permitir utilizar eficientemente as aplicações em ambientes heterogêneos e dinâmicos.

Para validar o modelo proposto, uma aplicação sintética que pode ser resolvida utilizando *D&C* foi implementada. Comparou-se resultados de escalonamento dessa aplicação com *Work-Stealing* e com um mecanismo mais simples, o *Work-Pushing* com escolha *Round-Robin* de recursos. Compara-se também, com a mesma aplicação implementada em um ambiente eficiente e focado para o modelo *D&C*, o Satin. Mostram-se também execuções da aplicação em ambientes dinâmicos e heterogêneos.

7.2 Trabalhos futuros

Neste trabalho, mostraram-se resultados focando uma única aplicação, o *N-Queens*. Este é o primeiro passo para validar o conceito, porém para se obter um ambiente que pode ser facilmente utilizado por outras aplicações, ainda é preciso que o mecanismo proposto seja facilmente adaptado para outras aplicações. Para

Tabela 7.1: Tabela comparativa que resume alguns dos tipos de escalonamento estudados.

Escalonamento	Vantagens	Tipo de aplicações
Work-Pushing com *Round-Robin*	Simplicidade	Carga regular
Work-Pushing com Ger. Recursos	Adaptabilidade	Carga regular
Work-Stealing em cluster	Aproveitamento de recursos	*D&C* dinâmicas
Work-Stealing hierárquico	Esconde latência da rede	*D&C* dinâmicas

utilizar o mecanismo proposto atual, é necessário que o programador defina um `MPI_Datatype` que represente uma tarefa na sua aplicação, o que pode se tornar uma dificuldade técnica.

Uma questão fundamental para o bom desempenho é o ajuste dos parâmetros de granularidade do roubo de tarefas. O ambiente Satin provê estatísticas detalhadas, permitindo que o usuário constate facilmente se os parâmetros utilizados resultaram em uma execução eficiente. Seria interessente que a implementação de *Work-Stealing* proposta fornecesse estatísticas da execução, como por exemplo: tentativas de roubo, roubos bem sucedidos, número de `spawns` e sobrecarga da parelização.

Mostrou-se como uma aplicação pode aproveitar recursos disponibilizados dinamicamente. No entanto, em alguns ambientes dinâmicos pode acontecer também a saída de recursos. O suporte na distribuição LAM já existe para isso, através do comando `lamshrink`. Porém, é preciso adaptar o mecanismo de *Work-Stealing* para não criar mais processos no recurso retirado do ambiente. Como trabalho futuro, também pode-se avaliar a capacidade de tolerância a falhas e testar se o ambiente de execução continua consistente após a queda de um recurso.

Apresentou-se um algoritmo de escalonamento distribuído que pode ser utilizado em *clusters* com interconexão de rede WAN. No entanto, apenas foram feitos testes desse mecanismo em um único *cluster*. Seria interessante refazer esses testes utilizando mais de um *cluster* interconectado por rede WAN, para verificar a eficiência de se utilizar a hierarquia de roteamento para mapear a localidade dos recursos.

Com os resultados obtidos, considera-se que é possível utilizar MPI para programar de forma eficiente aplicações *D&C* e utilizar recursos heterogêneos e dinâmicos. Obteve-se com MPI-2 um ganho de até 30% no tempo de execução em relação ao Satin, no caso de execuções mais longas. A utilização de um mecanismo de *Work-Stealing*, em conjunto com o ajuste da granularidade de trabalho de cada processo, mostraram-se suficientes para atingir um bom desempenho de uma aplicação *D&C* com MPI-2 em ambientes com diferentes quantidades de processadores.

REFERÊNCIAS

BALDESCHWIELER, J. E.; BLUMOFE, R. D.; BREWER, E. A. ATLAS: an infrastructure for global computing. In: ACM SIGOPS EUROPEAN WORKSHOP, 7., 1996, New York, NY, USA. **Proceedings...** New York: ACM Press, 1996. p.165–172.

BARRETO, M. E.; NAVAUX, P. O. A.; RIVIÈRE, M. P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 1998. **Proceedings...** [S.l.: s.n.], 1998. v.2, p.623–637.

BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. In: SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 5., 1995. **Proceedings...** [S.l.: s.n.], 1995.

BLUMOFE, R. D.; LEISERSON, C. Scheduling multithreaded computations by work stealing. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 35., 1994, Santa Fe, New Mexico. **Proceedings...** [S.l.: s.n.], 1994. p.356–368.

BLUMOFE, R. D.; LEISERSON, C. E. Space-efficient scheduling of multithreaded computations. **SIAM Journal on Computing**, [S.l.], v.27, n.1, p.202–229, 1998.

BLUMOFE, R. D.; LISIECKI, P. A. Adaptive and Reliable Parallel Computing on Networks of Workstations. In: ANNUAL TECHNICAL CONFERENCE ON UNIX AND ADVANCED COMPUTING SYSTEMS, USENIX, 1997. **Proceedings...** [S.l.: s.n.], 1997. p.133–147.

CAROMEL, D.; KLAUSER, W.; VAYSSIÈRE, J. Towards seamless computing and metacomputing in Java. **Concurrency: Practice and Experience**, [S.l.], v.10, n.11–13, p.1043–1061, 1998.

CAVALHEIRO, G. G. H.; GALILÉE, F.; ROCH, J.-L. Athapascan-1: parallel programming with asynchronous tasks. In: YALE MULTITHREADED PROGRAMMING WORKSHOP, 1998, Yale, USA. **Proceedings...** [S.l.: s.n.], 1998.

CAVALHEIRO, G. G. H.; KRUG, R. C.; RIGO, S. J.; NAVAUX, P. O. A. DPC++: an object-oriented distributed language. In: SCCC, 1995. **Proceedings...** [S.l.: s.n.], 1995. p.92–103.

CERA, M. C.; PEZZI, G. P.; MATHIAS, E. N.; MAILLARD, N.; NAVAUX, P. O. A. Improving the Dynamic Creation of Processes in MPI-2. In: EUROPEAN PVMMPI USERS GROUP MEETING, 13., 2006, Bonn, Germany. **Proceedings...** Berlin: Springer, 2006. p.247–255. (Lecture Notes in Computer Science, v.4192).

CERA, M.; PEZZI, G.; PILLA, M.; MAILLARD, N.; NAVAUX, P. Scheduling Dynamically Spawned Processes in MPI-2. In: WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 2006, Saint-Malo, France. **Proceedings...** [S.l.: s.n.], 2006.

CHANDY, K. M.; KESSELMAN, C. **Compositional C++**: compositional parallel programming. Pasadena, CA, USA: [s.n.], 1992.

CHARM++ Language Manual. Disponível em: <http://charm.cs.uiuc.edu/manuals/>. Acesso em: nov. 2006.

CORMEN, T. H. et al. **Algoritmos**: teoria e prática. [S.l.]: Campus, 2002.

GRIMSHAW, A. S. Easy-to-Use Object-Oriented Parallel Processing with Mentat. **Computer**, Los Alamitos, CA, USA, v.26, n.5, p.39–51, 1993.

GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2 Advanced Features of the Message-Passing Interface**. Cambridge, Massachusetts, USA: The MIT Press, 1999.

JO, C. H.; GEORGE, K. M.; TEAGUE, K. A. Parallelizing translator for an object-oriented parallel programming language. In: ANNUAL PHOENIX CONFERENCE ON COMPUTERS AND COMMUNICATIONS, 10., 1991. **Proceedings...** [S.l.]: IEEE Computer Society Press, 1991.

KALE, L. V.; KRISHNAN, S. Charm++: parallel programming with message-driven objects. In: WILSON, G. V.; LU, P. (Ed.). **Parallel Programming using C++**. [S.l.]: MIT Press, 1996. p.175–213.

KALE, L.; RAMKUMAR, B.; SINHA, A.; GURSOY, A. **The Charm Parallel Programming Language and System**: part i – description of language features. Urbana-Champaign: Department of Computer Science, University of Illinois, 1995 (Technical Report #95-02).

KWOK, Y.-K.; AHMAD, I. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. **ACM Computing Surveys**, [S.l.], v.31, n.4, p.406–471, 1999.

MEHTA, P.; AMARAL, J. N.; SZAFRON, D. Is MPI Suitable for a Generative Design-Pattern System? In: WORKSHOP ON PATTERNS IN HIGH PERFORMANCE COMPUTING, 2005. **Proceedings...** [S.l.: s.n.], 2005.

METIS Website. Disponível em: <http://www-users.cs.umn.edu/ karypis/metis/>. Acesso em: nov. 2006.

NEARY, M. O.; CAPPELLO, P. Advanced eager scheduling for Java-based adaptively parallel computing. In: ACM-ISCOPE CONFERENCE ON JAVA GRANDE, 2002, New York, NY, USA. **Proceedings...** New York: ACM Press, 2002. p.56–65.

NIEUWPOORT, R. V. van et al. Ibis: an efficient java-based grid programming environment. In: ACM-ISCOPE CONFERENCE ON JAVA GRANDE, 2002, New York, NY, USA. **Proceedings...** New York: ACM Press, 2002. p.18–27.

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Satin: efficient parallel divide-and-conquer in java. In: EURO-PAR, 2000, Munich, Germany. **Proceedings...** [S.l.]: Springer, 2000. p.690–699.

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Efficient load balancing for wide-area divide-and-conquer applications. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICES OF PARALLEL PROGRAMMING, 2001, New York, NY, USA. **Proceedings...** [S.l.]: ACM Press, 2001. p.34–43.

NIEUWPOORT, R. V. van et al. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. **Journal of Supercomputing**, [S.l.], 2006.

PACHECO, P. **Parallel Programming With MPI**. [S.l.]: Morgan Kaufmann Publishers, 1996.

PEZZI, G. P.; CERA, M. C.; MATHIAS, E. N.; MAILLARD, N.; NAVAUX, P. O. A. Escalonamento Dinâmico de programas MPI-2 utilizando Divisão e Conquista. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2006, Ouro Preto - MG. **Anais...** [S.l.: s.n.], 2006.

VEE, V.-Y.; HSU, W.-J. Applying Cilk in Provably Efficient Task Scheduling. **The Computer Journal**, [S.l.], v.42, n.8, p.699–712, 1999.

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

YONEZAWA, A. (Ed.). **ABCL**: an object-oriented concurrent system. Cambridge, MA, USA: MIT Press, 1990.