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

JOÃO VICENTE FERREIRA LIMA

Controle de Granularidade com *threads* em Programas MPI Dinâmicos

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

Prof. Dr. Nicolas Bruno Maillard
Orientador

Porto Alegre, abril de 2009

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Lima, João Vicente Ferreira

Controle de Granularidade com *threads* em Programas MPI Dinâmicos / João Vicente Ferreira Lima. – Porto Alegre: PPGC da UFRGS, 2009.

65 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Orientador: Nicolas Bruno Maillard.

1. Processamento Paralelo. 2. Programação de Alto Desempenho. 3. Algoritmos Dinâmicos. 4. Granularidade. I. Maillard, Nicolas Bruno. II. Título.

AGRADECIMENTOS

Primeiramente, eu gostaria de agradecer a minha familia, minha mãe Gizela e pai Luiz, assim como minha companheira Liana. Também, quero dedicar um especial agradecimento aos meus avós que deixaram saudades e boas lembranças.

Agradeço, em especial, ao meu orientador Nicolas Maillard pelo apoio e confiança neste trabalho. Da mesma forma, agradeço aos amigos das salas 207 e 205 que me apoiaram durante este período de mestrado.

Por fim, agradeço a CAPES pelo auxílio financeiro na bolsa.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AMPI	*Adaptive MPI*
API	*Application Programming Interface*
COW	*Cluster of Workstations*
CPU	*Central Processing Unit*
DAG	*Directed Acyclic Graph*
D&C	Divisão-e-Conquista
ELF	*Executable and Linkable Format*
GPPD	Grupo de Processamento Paralelo e Distribuído
HPC	*High Performance Computing*
II	Instituto de Informática
INRIA	*Institut National de Recherche en Informatique et en Automatique*
MPI	*Message-Passing Interface*
MPP	*Massively Parallel Processor*
NOW	*Network of Workstations*
OpenMP	*Open Multi-Processing*
PAD	Processamento de Alto Desempenho
SIMD	*Single Instruction Multiple Data*
SMP	*Symmetric Multi-Processor*
SPMD	*Single Program Multiple Data*
TBB	Intel© *Threading Building Blocks*
TLS	*Thread-Local Storage*
TSP	*Travelling Salesman Problem*
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Nos últimos anos, a crescente demanda por alto desempenho tem favorecido o surgimento de arquiteturas e algoritmos cada vez mais eficientes. A popularidade das plataformas distribuídas levanta novas questões no desenvolvimento de algoritmos paralelos tais como comunicação, heterogeneidade e dinamismo de recursos. Estas questões podem resultar em aplicações com carga de trabalho conhecida somente em tempo de execução. A irregularidade do algoritmo ou da entrada de dados também pode influenciar na carga de trabalho da aplicação. Uma aplicação paralela pode solucionar estas questões por meio de algoritmos dinâmicos ao utilizar técnicas de programação que definam o trabalho de uma tarefa e possibilitem a utilização de recursos sob demanda. A granularidade, que é a razão entre processamento e comunicação, considera questões práticas de execução e é um fator importante no desempenho de algoritmos dinâmicos. A implementação de um controle de granularidade é complicada e depende do suporte dos ambientes de programação. Porém, os ambientes de programação possuem interfaces extensas e complicadas que dificultam sua utilização em PAD. Este trabalho propõe a implementação de uma biblioteca (libSpawn) que incorpora um controle de granularidade em aplicações MPI dinâmicas. A biblioteca controla a granularidade ao mapear tarefas entre processos ou *threads* de acordo com três parâmetros: *cores* da arquitetura, carga e recursos de sistema. Os tempos obtidos com processos e libSpawn demonstram ganhos significativos em *benchmarks* sintéticos utilizados por outros ambientes de programação. Não obstante, constata-se carências na implementação atual que produzem tempos anômalos, ainda que estes sejam insignificantes em relação aos tempos com processos.

Palavras-chave: Processamento Paralelo, Programação de Alto Desempenho, Algoritmos Dinâmicos, Granularidade.

Controlling Granularity of Dynamic MPI Programs with Threads

ABSTRACT

In the last years, the demand for high performance enables the emergence of more efficient computing platforms and algorithms. The increase of distributed computing platforms rises new challenges for parallel algorithm development like communication, heterogeneity, and resource management. These factors can result in applications whose work load is unknown until runtime. An irregular behavior from algorithm or data can also affect the work load. A parallel application can solve these questions through a programming technique which predicts the work load of a task and offers resource on demand. The granularity, which is the ratio of computation to communication, considers more practical issues, and is an important factor in performance of dynamic algorithms. However, this control is difficult to be designed and the support of a programming tool is needed. Yet, the programming tools have extensive and complicated interfaces which difficult your usage in HPC. This work implements a library (libSpawn) which adds a granularity control on MPI dynamic programs. The library controls the granularity by mapping tasks between processes or threads with three parameters: cores of architecture, load and resources of the operating system. The results obtained between processes and libSpawn show significant gains on synthetic benchmarks from other programming tools.

Keywords: Parallel Computing, High Performance Computing, Dynamic Algorithms, Granularity.

1 INTRODUÇÃO

A pesquisa em computação paralela tem sido aplicada em várias áreas desde aplicações científicas até comerciais. A relação custo/benefício e a necessidade de desempenho são argumentos favoráveis ao uso da computação paralela. A crescente demanda por alto desempenho implica no surgimento de arquiteturas e de algoritmos cada vez mais eficientes e escaláveis. Essa demanda favoreceu o surgimento de diferentes plataformas de execução paralelas ao longo dos anos, como discutido a seguir.

1.1 Plataformas Paralelas

Nos últimos anos, as plataformas paralelas evoluíram rapidamente na busca contínua por maior desempenho. Os supercomputadores iniciaram com a produção de máquinas vetoriais SIMD (*Single Instruction Multiple Data*). Em seguida, uma nova geração de sistemas chamados MPP (*Massively Parallel Processor*) surgiu no mercado com desempenho comparável ou melhor que o de máquinas vetoriais. Entretanto, problemas de escalabilidade direcionaram o mercado para arquiteturas de memória distribuída. A venda de estações de trabalho SMP (*Symmetric multiprocessor*) mais baratas favoreceu o surgimento de redes NOW (*Network of Workstations*), que possuem uma relação custo/benefício superior. O sucesso destas plataformas impulsionou o surgimento de arquiteturas e redes de alto desempenho cada vez mais acessíveis.

Atualmente, os agregados ou *clusters* são amplamente empregados em PAD. Esta plataforma se caracteriza por um conjunto de máquinas SMP e *multi-core* uniformes e interligadas por uma rede de alto desempenho. A popularidade dos agregados contribui para o surgimento de plataformas *cluster of clusters* que são formadas pela interligação de vários agregados. Esta nova interligação pode apresentar recursos heterogêneos entre agregados de diferentes capacidades, além de interconexões mais suscetíveis a falhas. Uma outra plataforma popular é a Grade ou *Grid*, que é uma infra-estrutura direcionada à computação altamente distribuída e sob demanda. Uma grade se caracteriza por recursos heterogêneos e dinâmicos que podem ser incluídos ou excluídos a qualquer momento.

1.2 Programação Paralela

A programação paralela também evoluiu de acordo com as diferentes plataformas ao longo dos últimos anos. Um modelo de programação é fundamental para a eficiência em determinada arquitetura e, por conseguinte, para o ganho em desempenho. O desenvolvimento em máquinas vetoriais e MPPs estava direcionado a técnicas de programação que exploram o paralelismo em memória compartilhada. Por outro lado, o surgimento

de plataformas distribuídas direcionou esforços para modelos distribuídos com a finalidade de explorar tanto a concorrência quanto a comunicação. Além disso, as plataformas largamente distribuídas implicam em novas questões a serem tratadas tais como: custo de comunicações, heterogeneidade e dinamismo de recursos. Estas características podem resultar em aplicações com carga de trabalho conhecida somente em tempo de execução.

As irregularidades do algoritmo ou da entrada de dados também podem influenciar na carga de trabalho da aplicação. Uma aplicação paralela soluciona estas questões por meio de algoritmos dinâmicos ao utilizar uma técnica de programação apropriada, como por exemplo *Mestre/Escravo* e *Divisão-e-Conquista*. Estas técnicas objetivam atribuir uma carga de trabalho conhecida às tarefas sob demanda em tempo de execução. Porém, a implementação destas técnicas deve considerar as características específicas da plataforma de execução através de uma revisão nas etapas de desenvolvimento do algoritmo. Esta revisão define a granularidade, ou seja, a razão entre os períodos de computação e comunicação.

A granularidade pode ser controlada de duas formas: por agrupamento ou por localidade. O agrupamento aumenta ou diminui o grão do problema, ou seja, o tamanho de cada subproblema na decomposição do algoritmo. Por outro lado, a localidade controla a granularidade por meio do mapeamento de tarefas para o mesmo processador. Um algoritmo dinâmico necessita controlar a granularidade a fim de obter resultados eficientes em plataformas distribuídas. Todavia, o controle de granularidade exige uma implementação complicada que envolve diversas características. Portanto, a utilização de ambientes de programação que suportam este controle facilita o desenvolvimento de aplicações dinâmicas. Porém, estes ambientes são eficientes em determinadas arquiteturas e descrevem interfaces de programação extensas.

1.3 Proposta deste Trabalho

Este trabalho propõe uma biblioteca chamada libSpawn que incorpora o controle de granularidade em aplicações MPI dinâmicas. Esta proposta define uma tarefa como um fluxo de execução que pode ser tanto um processo quanto uma *thread*. O controle de granularidade decide a representação da tarefa de acordo com três parâmetros que avaliam o contexto de execução do programa: *cores* da arquitetura, carga e recursos de sistema. Dessa forma, a biblioteca aumenta a localidade de tarefas e incorpora ao MPI as características que melhoram a eficiência de um controle de granularidade, conforme descrito no capítulo 2.

1.4 Estrutura do Texto

O capítulo 2 contextualiza o leitor ao apresentar características relacionadas às plataformas distribuídas e conceitos essenciais em algoritmos dinâmicos. Este capítulo descreve os fatores de uma plataforma que interferem na eficiência de uma aplicação paralela e podem ser resolvidos com algoritmos dinâmicos. Além disso, os conceitos essenciais e características de algoritmos dinâmicos são descritos. Não obstante, o texto apresenta duas técnicas de programação para algoritmos dinâmicos e quatro exemplos de aplicações. Por fim, esse capítulo aborda o controle de granularidade em algoritmos dinâmicos e define algumas características necessárias em um controle eficiente.

Em seguida, o capítulo 3 discute o estado da arte em ambientes de programação paralela com suporte ao dinamismo. Estes ambientes são divididos de acordo com a plata-

forma suportada: plataformas de memória compartilha ou plataformas distribuídas. Por fim, um panorama a respeito destes ambientes relaciona o suporte de cada um às características desejáveis para um controle de granularidade eficiente, descritas no capítulo 2.

Na sequência, o capítulo 4 descreve a biblioteca proposta (libSpawn) para o controle de granularidade. O texto desse capítulo apresenta a versão anterior da libSpawn que se caracteriza pelo controle estático de granularidade. Em seguida, a proposta da versão atual é apresentada em relação ao controle de granularidade e comunicação entre tarefas. Em seguida, o texto relaciona a versão atual da libSpawn com as características desejáveis de um controle de granularidade eficiente conforme o capítulo 2. Por fim, as vantagens e desvantagens dessa proposta são descritas.

O capítulo 5 demonstra os experimentos realizados com a libSpawn. As aplicações dinâmicas, descritas no capítulo 2, consistem em implementações com particionamento recursivo de dados. A avaliação compara o resultados obtidos com a criação dinâmica de processos frente aos obtidos com dinamismo de tarefas e controle de granularidade.

Por fim, o capítulo 6 apresenta as considerações finais deste trabalho.

2 GRANULARIDADE EM APLICAÇÕES DINÂMICAS

A eficiência de uma aplicação paralela depende de fatores relacionados à plataforma de execução e ao algoritmo que podem se relacionar e não devem ser desconsiderados (JÁJÁ, 1992). Alguns destes fatores podem resultar em aplicações com carga de trabalho conhecida somente em tempo de execução. A granularidade envolve questões de arquitetura e está relacionada com todas as fases de desenvolvimento de um algoritmo paralelo. Através dela, pode-se definir os períodos de computação e comunicação localmente, por recurso, ou globalmente no algoritmo.

Portanto, a granularidade possibilita à aplicação adaptar-se às características de plataformas distribuídas, descritas na seção 2.1. A fim de se explorar o dinamismo e a heterogeneidade das plataformas, o algoritmo também deverá ser dinâmico. A seção 2.2 introduz os conceitos essenciais em algoritmos dinâmicos, seguidos de duas técnicas de programação e suas características relacionadas ao controle de granularidade. Em seguida, a seção 2.3 exemplifica quatro aplicações dinâmicas utilizadas nos experimentos deste trabalho. Por fim, uma conclusão deste capítulo é apresentada na seção 2.4.

2.1 Plataformas Distribuídas para Programação Paralela

O desenvolvimento de algoritmos paralelos não pode desconsiderar aspectos relacionadas à plataforma de execução, entre eles o custo de comunicação, dinamismo e heterogeneidade de recursos. Nesta seção, as características de duas categorias de plataformas distribuídas para a programação paralela são descritas: estáticas e dinâmicas.

2.1.1 Plataformas Estáticas

Uma plataforma estática se caracteriza por um conjunto de recursos homogêneos ou heterogêneos conectados por uma rede local de alta velocidade. Esta plataforma pode ser dedicada ou não no sentido de compartilhamento de recursos com usuários ou outros programas e serviços. Esta seção apresenta duas plataformas populares classificadas como estáticas: NOW e agregados.

NOW (*Network of Workstations*), ou também conhecido como COW (*Cluster of Workstations*), é uma alternativa às arquiteturas MPP (*Massively Parallel Processor*) (WILKINSON; ALLEN, 1999) que integra vários computadores (PCs e estações de trabalho) conectados por redes de alto desempenho a fim de serem usadas em aplicações paralelas e distribuídas. A principal vantagem de uma NOW é a relação custo/benefício em comparação às arquiteturas MPP. O surgimento de PCs cada vez mais potentes e baratos contribui significadamente para a popularização de NOWs em PAD, o que também auxilia no surgimento de projetos em ambientes de programação para esta arquitetura.

Outra vantagem de plataformas NOW é a possibilidade de incluir PCs conforme a aquisição de recursos. À medida que máquinas mais potentes são adquiridas, elas podem ser facilmente incluídas na rede. Todavia, a existência de diferentes configurações torna uma NOW heterogênea em relação ao *hardware* existente. Além disso, uma NOW não dedicada pode ser compartilhada entre usuários e processos, que podem disputar o tempo de CPU. Por exemplo, PCs de propósito geral podem ser utilizados por usuários em uma NOW e gerar alguma carga de trabalho irregular e não periódica.

Um Agregado de Computadores ou *Cluster* (NAVAUX; ROSE, 2003) é uma plataforma composta de PCs ou estações de trabalho conectados por redes de alto desempenho. As ideias de um agregado são semelhantes à arquitetura NOW, porém há diferenças estruturais (DONGARRA et al., 2003). As principais características derivam do projeto Beowulf (STERLING et al., 1995) onde múltiplos computadores (PCs ou estações de trabalho) chamados *nodos* ou *nós* estão diretamente interligados com um servidor, conhecido como *front-end*. Este possui monitor e teclado para acesso às máquinas restantes, além de ser o responsável pelo armazenamento de informações sobre usuários, arquivos e demais serviços. Em sua maioria, os nós de agregados não possuem monitor e teclado, são dedicados e têm como sistema operacional uma distribuição livre GNU/Linux.

Atualmente, o surgimento de arquiteturas *multi-core* cada vez mais acessíveis contribui na popularização de agregados com um número significativo de núcleos ou *cores*. Uma configuração SMP *multi-core* em um agregado possibilita plataformas com paralelismo multi-nível em memória compartilhada (intra-nó) e distribuída (inter-nó). Além dos agregados simples, a crescente interligação de agregados (*cluster of clusters*) que podem ter diferentes configurações de *hardware* deve ser considerada.

Na programação paralela, aplicativos direcionados a estas arquiteturas devem considerar aspectos relacionados à heterogeneidade e localidade. A interligação de agregados e NOW envolve recursos heterogêneos que interferem na eficiência do algoritmo. Por sua vez, a comunicação no paralelismo multi-nível implica em diferentes custos devido às referências locais e remotas. As interligações entre agregados podem ter distâncias geográficas consideráveis tais como diferentes países ou continentes, que implicam em tempos de comunicação consideráveis.

2.1.2 Plataformas Dinâmicas

Por sua vez, as plataformas dinâmicas se caracterizam por variações na disponibilidade de recursos e maior heterogeneidade. Além disso, esta plataforma compartilha recursos e serviços de forma controlada e independente. Uma plataforma dinâmica popular é a Grade ou *Grid* (FOSTER; KESSELMAN, 1999) que denota uma infra-estrutura distribuída voltada à computação em larga escala. Ela pode ser aplicada principalmente na computação altamente distribuída e sob demanda.

O real problema na computação em grade é o compartilhamento controlado de recursos e serviços de forma dinâmica e distribuída geograficamente. Este compartilhamento não se restringe à troca de arquivos mas envolve o acesso direto a programas, computadores, dados, sensores e outros recursos. Além disso, ele é altamente controlado com servidores e consumidores que definem claramente o que é compartilhado, quem tem permissões de acesso e as condições de compartilhamento. As suas relações necessitam de uma flexibilidade que pode variar de uma estrutura cliente-servidor a ponto-a-ponto. Estas relações de compartilhamento variam em termos de recursos utilizados, tipos de acesso, participantes e disponibilidade. O dinamismo de uma grade permite a adição de recursos conforme as políticas de uso e disposição de novos recursos.

A arquitetura de uma grade envolve interconexões entre diferentes países, organizações, *clusters* e computadores pessoais. Foster e Kesselman (1999) descrevem uma perspectiva de arquitetura de grade e a infra-estrutura de aplicação necessária de acordo com os serviços básicos proporcionados por computadores convencionais. A perspectiva de arquitetura consiste em:

- **Sistemas finais** - um sistema individual com componentes altamente acoplados e homogêneos, como PC's. Tais sistemas já suportam os serviços básicos;
- ***Clusters*** - um *cluster* de computadores ou uma NOW. Os serviços nesta classe devem considerar a escalabilidade e a redução de acoplamento;
- ***Intranets*** - um largo número de recursos pertencentes a uma mesma organização. Os serviços devem abordar a heterogeneidade de recursos, a administração individual de cada sistema e a falta de conhecimento a respeito da estrutural global por parte dos recursos;
- ***Internets*** - sistemas interconectados entre múltiplas organizações geograficamente distantes. Os serviços devem estar cientes da falta de controle centralizado, da distribuição geográfica e das politicas internacionais de segurança entre diferentes países.

Dessa forma, os aspectos que devem ser considerados se acumulam conforme o número de recursos e a distância considerada. Além disso, a dinamicidade de recursos possibilita o uso de recursos sob demanda, que implica em questões de escalonamento e balanceamento de carga em tempo de execução.

2.2 Programação Paralela com Dinamismo

A seção anterior demostrou as características de plataformas, tais como dinamismo, heterogeneidade e localidade de comunicações, que interferem na carga de trabalho de aplicações paralelas. Uma forma de lidar com estes aspectos é através de algoritmos dinâmicos que possibilitem o balanceamento de carga, a utilização de recursos sob demanda e melhor localidade de comunicações.

Esta seção introduz os principais conceitos a respeito de algoritmos dinâmicos e técnicas de programação com dinamismo. Inicialmente, a seção 2.2.1 descreve dois conceitos fundamentais na programação paralela: tarefa e granularidade. Em seguida, as características de algoritmos dinâmicos são apresentadas na seção 2.2.2. Para a implementação de algoritmos dinâmicos, as seções 2.2.3 e 2.2.4 apresentam as técnicas *Mestre/Escravo* e *Divisão-e-Conquista*, respectivamente. As vantagens e desvantagens destas técnicas são listadas na seção 2.2.5 e, por fim, os aspectos em relação ao controle de granularidade são discutidos na seção 2.2.6.

2.2.1 Definição de Tarefa e Granularidade

Um algoritmo paralelo envolve mais do que a descrição de passos para solucionar um problema. Ele deve, no mínimo, especificar um conjunto de passos que podem ser executados concorrentemente, além das etapas de comunicação se necessárias. O desenvolvimento de algoritmos paralelos envolve as seguintes etapas básicas (FOSTER, 1995):

- **Decomposição** - dividir um problema inicial em unidades menores de trabalho que podem executar em paralelo;

- **Mapeamento** - atribuir partes menores do problema para execução nos processadores disponíveis;

- **Comunicações** - comunicação necessária na coordenação entre os processadores, como por exemplo o envio de entradas e recebimento de resultados.

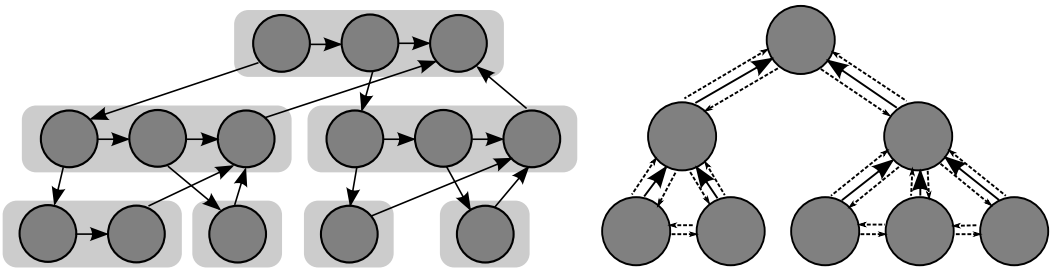
A seção 2.2.1.1 descreve o conceito de tarefa, que é fundamental na programação paralela e no entendimento deste trabalho. Em seguida, a definição de um conceito que relaciona as três etapas anteriores, a granularidade, é descrita na seção 2.2.1.2.

2.2.1.1 Tarefa

A primeira etapa no desenvolvimento de um algoritmo paralelo determina a unidade de trabalho do problema, que é a tarefa. A tarefa consiste em uma unidade sequencial abstrata que possui um bloco de instruções e, possivelmente, se comunica com outras tarefas. Uma tarefa pode depender do resultado de outras e, dessa forma, apresentar dependências. Os grafos acíclicos dirigidos ou *Directed Acyclic Graphs* (DAG) são uma abstração que expressam as dependências entre tarefas, além de apresentarem uma relativa ordem de execução.

O número de tarefas de uma aplicação pode ser determinado de duas formas: estaticamente ou dinamicamente. A estática se refere aos algoritmos onde as tarefas são determinadas antes da execução, tais como na decomposição por dados. Por outro lado, a dinâmica gera tarefas sob demanda em tempo de execução. Por exemplo, algoritmos com decomposição recursiva dividem um problema inicial em menores onde as folhas de uma árvore de subproblemas representam as tarefas. Dessa forma, o DAG de dependências na criação dinâmica é desconhecido até o momento da execução.

A representação de uma tarefa abstrata não implica necessariamente em um fluxo de execução, mas está relacionada ao ambiente de programação. A figura 2.1 ilustra exemplos de DAG referentes ao ambiente de programação Cilk e ao padrão MPI, ambos detalhados no capítulo 3. Nos exemplos, cada vértice denota uma tarefa, ao passo que as arestas direcionadas indicam as dependências entre tarefas. Porém, a implementação de tarefa difere de acordo com cada abordagem. A abordagem Cilk no DAG 2.1(a) utiliza tarefas como *Cilk threads* que são uma sequência de instruções cujo término é determinado por uma das palavras chaves do ambiente. Os retângulos em Cilk são chamados *procedures* e denotam um fluxo leve ou *thread* de execução. O DAG MPI em 2.1(b), por sua vez, denota tarefas como processos pesados que podem ou não ser gerados dinamicamente. As arestas pontilhadas do DAG MPI representam as possíveis comunicações entre os processos.



(a) DAG em Cilk (b) DAG em MPI

Figura 2.1: Exemplos de um DAG de tarefas no ambiente Cilk e no padrão MPI

2.2.1.2 Granularidade

As etapas de decomposição e comunicação determinam a concorrência de um programa paralelo, ao passo que o mapeamento consiste na revisão das etapas anteriores a fim de se obter um algoritmo eficiente. Esta etapa inicia a avaliação de características da plataforma tais como recursos de processamento, localidade e custos em comunicação e criação de tarefas (FOSTER, 1995). A partir deste ponto, os períodos de processamento e comunicação estão definidos na aplicação de forma que é possível medir a sua granularidade. Granularidade representa a razão entre processamento e comunicação de forma global ou local por tarefa.

Essa razão se divide em dois grupos de acordo com proporção dos fatores envolvidos. Uma granularidade de grão fino ou *fine-grain* apresenta períodos curtos de processamento entre as comunicações, ao passo que a de grão grosso ou *coarse-grain* possui períodos de processamento maiores entre as comunicações. A granularidade de grão grosso é apropriada para algoritmos paralelos com alto custo nas comunicações e que são regulares em relação ao trabalho das tarefas. Por outro lado, o grão fino é usado em algoritmos com carga de trabalho irregular devido a facilidade na realização de balanceamento de carga.

O controle da granularidade ocorre basicamente de duas formas tal como ilustrado na figura 2.2 (FOSTER, 1995). A primeira forma agrupa tarefas na decomposição do problema para aumentar o grão, de forma que obtenha-se um número menor de tarefas com um tamanho maior como na figura 2.2(a). Essa abordagem aumenta o trabalho de cada tarefa e pode eliminar questões relacionadas ao mapeamento se o número de tarefas igualar-se ao de processadores. A segunda forma não altera a tarefa, mas aumenta a localidade ao mapear tarefas para o mesmo processador. O exemplo da figura 2.2(b) demostra o mapeamento de ramificações de uma árvore para um mesmo processador.

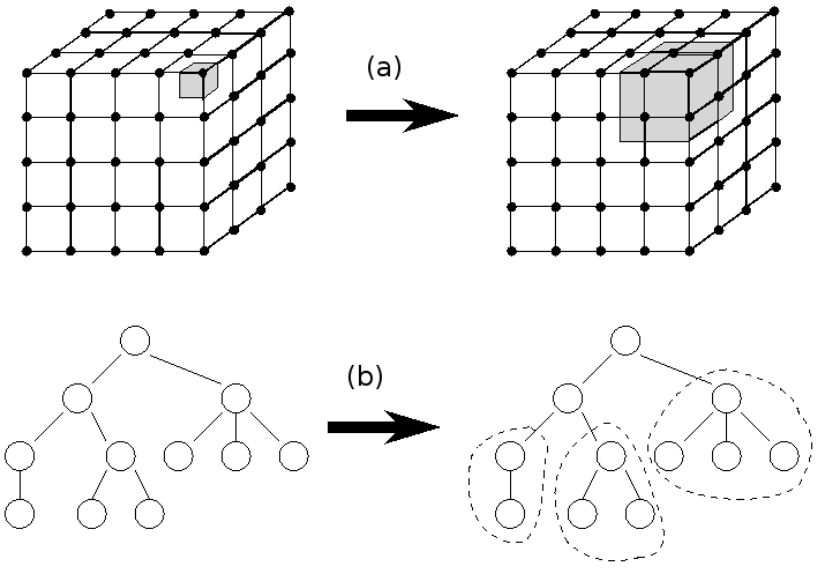


Figura 2.2: Exemplos de algoritmos que controlam a granularidade de duas formas distintas

A principal vantagem no aumento da granularidade é a redução nos custos de comunicação para ambas as abordagens de controle. Um controle por aumento do grão elimina as comunicações, ao passo que por aumento da localidade substitui comunicações remotas pelas locais de baixo custo. Os algoritmos com um número fixo de tarefas e comunicações

estruturadas alcançam resultados eficientes com um controle simples. Entretanto, problemas com variação na carga de trabalho e comunicações irregulares dificultam o controle de granularidade.

2.2.2 Algoritmos Dinâmicos

Um algoritmo dinâmico é caracterizado pela carga de trabalho e DAG de dependências serem conhecidos somente em tempo execução. A carga de trabalho se altera conforme a irregularidade do algoritmo, da entrada ou da plataforma de execução. A irregularidade pode estar associada as seguintes características (RÜNGER, 2006):

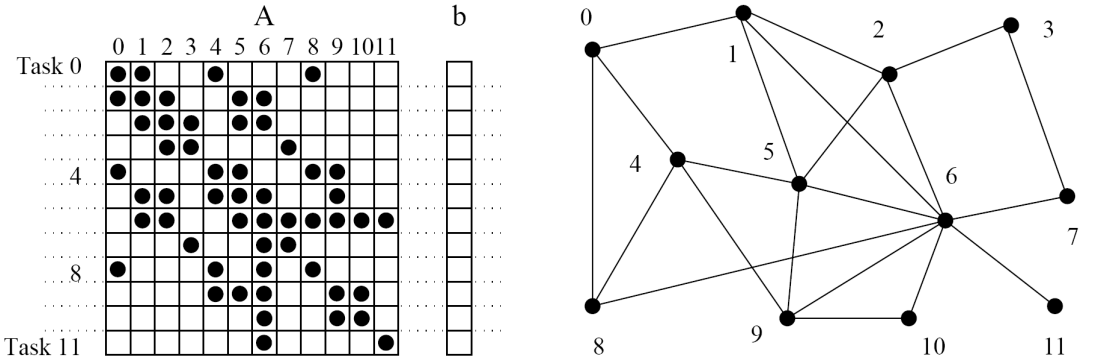
- **Entrada de dados** - problemas orientados a dados onde os dados de entrada se mostram irregulares. Uma matriz esparsa exemplifica este tipo de característica;
- **Estruturas de dados** - algoritmos com estruturas que se alteram, crescem ou diminuem, em tempo de execução;
- **Dependências de dados** - as dependências com relação a outras tarefas se alteram em tempo de execução. Elas podem ser locais, por exemplo com vizinhos, e globais, com tarefas de partes distintas ou até não envolvidas no mesmo problema;
- **Localidade de estruturas** - a distribuição de estruturas entre as tarefas se altera em tempo de execução. Uma implementação paralela pode decompor estruturas tais como vetores e grafos de forma que os processadores recebam diferentes subproblemas. Dessa forma, referências no mesmo processador causam acessos locais, ao passo que referências a processadores remotos causam acessos remotos;
- **Comunicações** - o padrão de comunicação entre tarefas se altera em tempo de execução de modo que a previsão das comunicações com relação a localidade é impossibilitada.

A figura 2.3, na página 20, ilustra como exemplo o produto $y = Ab$ de uma matriz esparsa A por um vetor b. A parte 2.3(a) mostra a matriz esparsa A e o vetor b com decomposição de uma linha por tarefa, ao passo que a parte 2.3(b) demonstra o grafo de interações necessárias entre as tarefas. As figuras mostram diversas características irregulares citadas anteriormente. Uma delas é a entrada de dados irregular da matriz esparsa A na parte 2.3(a) onde cada linha contém um número distinto de pontos não nulos. Outra característica se mostra na parte 2.3(b) devido às diferentes dependências de dados de acordo com a linha de cada tarefa.

A implementação de algoritmos dinâmicos deverá considerar as características descritas nesta seção por meio de uma técnica de programação. Em seguida, duas técnicas de decomposição são descritas. O objetivo de ambas é atribuir uma carga de trabalho conhecida às tarefas sob demanda em tempo de execução. Cada técnica tem abordagens diferentes e, como discutido, suas próprias vantagens e limitações.

2.2.3 A Técnica *Mestre/Escravo*

A técnica *Mestre/Escravo* ou também conhecida como *work pool* consiste no balanceamento de carga centralizado através de uma tarefa *mestre* que contém o conjunto de tarefas para serem executadas. Inicialmente, as tarefas *escravo* recebem uma tarefa do *mestre*. Após o término desta tarefa, os *escravos* requisitam uma nova tarefa do *mestre*. A



(a) Matriz esparsa A e vetor b

(b) Grafo de interação entre tarefas

Figura 2.3: Exemplo de multiplicação $y = Ab$ de uma matriz esparsa A por um vetor b e o grafo de interações entre tarefas

execução termina no momento que a fila do *mestre* estiver vazia, assim todos os *escravos* serão notificados do término.

Uma abordagem dinâmica é possível na medida que o *mestre* crie novas tarefas *escravo* sob demanda. Em um primeiro momento, p tarefas podem ser utilizadas na solução do problema e, na medida que novos recursos sejam necessários, mais tarefas podem ser lançadas dinamicamente. Gropp; Lusk e Thakur (1999) propõem a criação de tarefas *escravo* de acordo com os nós disponíveis à execução do programa. Uma abordagem dinâmica é tratada por Leopold e Süss (2006) onde a técnica *Mestre/Escravo* é adaptada para incluir *escravos* em tempo de execução através de métodos cliente-servidor do MPI-2 (LEOPOLD; Süß; BREITBART, 2006), discutido no capítulo 3.

Uma desvantagem significativa da técnica *Mestre/Escravo* está no *mestre* que pode enviar apenas uma tarefa por vez (WILKINSON; ALLEN, 1999). Após o envio de uma tarefa inicial para cada *escravo*, o *mestre* poderá responder as requisições por tarefas sequencialmente. Portanto, há a possibilidade de gargalo quando vários *escravos* requisitarem tarefas ao mesmo tempo. Esta técnica terá uma eficiência satisfatória com poucos *escravos* e tarefas de grão grosso.

2.2.4 A Técnica *Divisão-e-Conquista*

Divisão-e-Conquista (CORMEN et al., 2002) é uma técnica de decomposição recursiva que consiste na divisão de um problema inicial em subproblemas menores para serem resolvidos separadamente. A mesma divisão é aplicada recursivamente em cada subproblema até que a solução seja imediata. Por fim, as soluções parciais são combinadas para obter-se a solução do problema inicial. Os passos em cada recursão podem ser:

- **Dividir** o problema inicial em um determinado número de subproblemas;
- **Conquistar** os subproblemas ao resolve-los recursivamente. Se os tamanhos dos subproblemas são pequenos, basta aplicar a solução imediata;
- **Combinar** as soluções dos subproblemas, a fim de formar a solução para o problema original.

JáJá (1992) faz uma distinção entre algoritmos cujo trabalho principal é dividir o problema daqueles cujo trabalho principal é combinar os resultados, como em algoritmos

de ordenação. Ele classifica os métodos que têm o trabalho principal na combinação de resultados como D&C, ao passo que métodos cujo trabalho principal está na divisão são definidos como de *particionamento* (WILKINSON; ALLEN, 1999). Neste trabalho, não haverá este tipo de distinção.

Os passos descritos possibilitam uma definição recursiva que pode ser escrita, para uma entrada e e solução S, como:

$$S(e) = \begin{cases} direto(e) & \text{se } simples(e) \\ combina((parte_1(e)), (parte_2(e))) & \text{senão} \end{cases}$$

Uma forma de representar aplicações *D&C* é através de um DAG (KWOK; AHMAD, 1999), onde os vértices e arestas representam, respectivamente, tarefas e as dependências entre eles. A figura 2.4 ilustra um exemplo de aplicação *D&C* como um DAG onde a tarefa T_0 depende dos resultados gerados por T_1 e T_2.

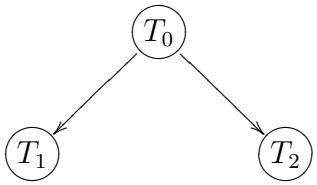


Figura 2.4: Aplicação *D&C* vista como um DAG.

2.2.4.1 Divisão-e-Conquista Paralela

A técnica de *Divisão-e-Conquista* também pode ser utilizada na obtenção de algoritmos paralelos eficientes de acordo com resultados teóricos. Vários trabalhos abordam o escalonamento de tarefas em aplicações *D&C* através de ambientes de programação, descritos no capítulo 3 (BLUMOFE; LEISERSON, 1998; NIEUWPOORT; KIELMANN; BAL, 2001; GAUTIER; BESSERON; PIGEON, 2007; PEZZI et al., 2007; CERA et al., 2006).

Uma forma de se obter uma aplicação *D&C* paralela é criar novas tarefas para resolver os subproblemas recursivamente. Cada tarefa executa em paralelo, porém deve-se considerar as dependências representadas pelas arestas dos DAGs. A figura 2.5, na página 22, mostra um exemplo de aplicação do tipo *D&C* na execução de um nível da recursão, dividido em 4 passos, com uma operação $\oplus$ efetuada em dois vetores com duas posições. Inicialmente, a tarefa T_0 recebe os dados de entrada no passo 1. No passo 2, T_0 decompõe o problema entre duas tarefas (T_1 e T_2) e envia as partes correspondentes para cada uma. No passo 3, T_1 e T_2 descobrem que a parte recebida tem solução imediata (no caso, um vetor unitário), executam a operação $\oplus$ sobre os dados e retornam os resultados parciais para T_0. Por fim, T_0 combina os resultados parciais para gerar um vetor com o resultado final (passo 4).

2.2.4.2 Comunicação em D&C

Uma implementação desta técnica em um modelo distribuído poderá trabalhar com limitações na comunicação entre tarefas. A técnica recursiva D&C permite limitar a comunicação entre tarefas criadas (tarefas filhas) e seu criador (tarefa pai) (PEZZI et al., 2006) conforme a representação de aplicações por meio de um DAG. Dessa forma, uma implementação com troca de mensagens exige duas comunicações: envio das entradas aos filhos e retorno dos resultados ao pai.

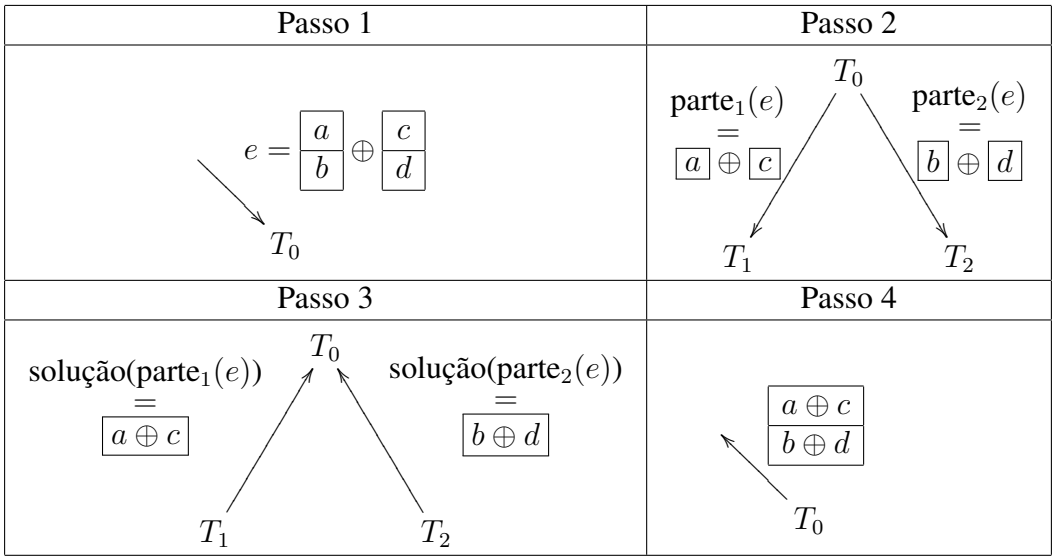


Figura 2.5: Execução de uma aplicação do tipo *D&C*.

2.2.4.3 Exemplos de problemas resolvidos com D&C

Esta seção apresenta brevemente alguns problemas onde uma solução por *D&C* é proposta com a finalidade de exemplificar os tipos de aplicações em que esta técnica pode ser aplicada. Questões relacionadas à implementação dos problemas são omitidas.

- **Ray tracing** - devido ao fato de cada linha da imagem poder ser gerada em paralelo, cada tarefa gera uma parte da imagem. Ao final, as partes são combinadas para gerar a imagem final;
- **Construção de modelos 3D por *octree*** - gera-se uma *octree* correspondente ao modelo 3D e uma lista de pontos, ou tarefas, para cada nível de profundidade. Em seguida, as listas são atribuídas ao processadores que podem roubar tarefas de outros níveis caso sua lista esteja vazia (SOARES et al., 2007);
- **Ordenação de números** - com o algoritmo *Bucket Sort*, divide-se o vetor de entrada em regiões do mesmo tamanho e atribui-se um *bucket* para cada região. Após, cada região é ordenada e os *buckets* são concatenados;
- **Caixeiro viajante (TSP)** - gera-se uma árvore com todos os possíveis caminhos e inicia-se com um valor infinito para o menor caminho. Novas tarefas surgem para as bifurcações, como na busca em árvore, e o valor do caminho é calculado. Ao completar um caminho, ou seja, ao atingir um nó folha, compara-se o valor desse caminho com o valor do menor caminho. Caso seja menor, avisa as demais tarefas que foi encontrado um caminho com menor custo;
- **Operações em sequências 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;
- **Busca de elementos em árvore** - cada bifurcação da árvore pode gerar uma nova tarefa que busca o elemento na sua sub-árvore. Assim que o elemento for encontrado, notifica-se as demais tarefas para finalizar a execução.

2.2.5 Vantagens e Desvantagens

Esta seção mostra um balanço geral do dinamismo através das vantagens e desvantagens das técnicas apresentadas para a programação paralela. As principais vantagens são:

- Incorporar recursos disponíveis dinamicamente em tempo de execução através da criação dinâmica de tarefas;
- Melhor aproveitamento de recursos em ambientes heterogêneos. A técnica *Mestre/Escravo* atribui mais tarefas para escravos em processadores com maior capacidade, ao passo que D&C possibilita dividir um problema que executa em um processador com menor capacidade entre outros processadores;
- Em *D&C*, pode-se alterar também a largura da árvore para controlar o grau de paralelismo da aplicação;
- *D&C* possibilita a implementação de aplicações paralelas com dependências entre tarefas.

As principais desvantagens, por sua vez, são:

- No *Mestre/Escravo*, o balanceamento de carga centralizado pode gerar um gargalo significativo na execução;
- A implementação das comunicações entre tarefas pode tornar-se um trabalho complicado, principalmente nos algoritmos dinâmicos em que variam os padrões de comunicação;
- Dificuldade em definir a localidade de problemas com grão fino, onde o custo das comunicações é um fator significativo que deve ser considerado.

Os itens acima permitem algumas conclusões a respeito do desenvolvimento de algoritmos dinâmicos. Inicialmente, o balanceamento de carga centralizado dificulta a obtenção de algoritmos eficientes devido ao pior caso quando vários escravos requisitam tarefas ao mesmo tempo. Um bom desempenho é possível contanto que o algoritmo consiga reduzir os custos de comunicação e criação de tarefas. A seção a seguir descreve uma forma de solucionar tais problemas através do controle de granularidade.

2.2.6 Controle de Granularidade

Um algoritmo estático se caracteriza por apresentar uma carga de trabalho conhecida antes da execução, ou seja, com geração estática de tarefas. A granularidade destes problemas é geralmente definida por agrupamento na decomposição do problema e é controlada conforme o número de processadores disponíveis. De acordo com a decomposição de dados, uma tarefa representa um subproblema p mapeado ao processador p de tal forma que cada processador obtenha uma carga de trabalho semelhante e conhecida antes da execução (JÁJÁ, 1992). As tarefas em problemas estáticos apresentam poucos ou ausentes períodos de comunicação em relação aos períodos de processamento.

Por outro lado, um algoritmo dinâmico têm sua carga de trabalho e DAG de dependências desconhecidas antes da execução. A técnica *Mestre/Escravo* possibilita determinar a granularidade local de cada tarefa e distribuir a carga de trabalho entre os recursos ocupados por *escravos*, porém seu desempenho é limitado como discutido na seção anterior. Na

Divisão-e-Conquista, a granularidade é definida globalmente por meio do mapeamento de algumas folhas do DAG com carga de trabalho conhecida para um mesmo processador. Todavia, o número de folhas desconhecido antes da execução dificulta o controle de granularidade globalmente destas tarefas.

Um controle de granularidade depende de uma solução que envolve as seguintes características:

- **Descrição e representação de tarefa** - descrever uma tarefa como uma estrutura de dados abstrata ou representar como um processo ou *thread* em execução;
- **Localidade em comunicações** - considerar o tipo de referência, local ou remota, entre tarefas;
- **Carga de trabalho** - controlar a carga de trabalho entre os recursos utilizados;
- **Gerenciamento de recursos** - controlar os recursos disponíveis em tempo de execução.

Uma aplicação em geral não incorpora tais características devido à dificuldade de implementação ou por causa da falta de suporte das interfaces de programação. Dessa forma, a utilização de ambientes de programação que suportam o controle de granularidade simplifica a programação de aplicações dinâmicas e possibilita maior eficiência dos algoritmos desenvolvidos.

2.3 Exemplos de Aplicações Dinâmicas

A seguir, quatro aplicações dinâmicas são apresentadas. Cada seção inicia com uma descrição breve e geral a respeito da aplicação e finaliza com uma proposta de implementação paralela. As aplicações abaixo utilizam a técnica *D&C* tanto na versão sequencial quanto paralela. Os experimentos realizados neste trabalho utilizam os problemas discutidos nesta seção e seus resultados encontram-se no capítulo 5.

2.3.1 *N-Queens*

O *N-Queens* (N*-Rainhas*) consiste na colocação de N rainhas em um tabuleiro de xadrez, de tamanho $N \times N$, de forma que uma rainha não capture outra em uma jogada. O algoritmo busca configurações nas quais não existam mais de uma rainha em cada linha, coluna e diagonal. A forma mais utilizada na resolução desse problema consiste na utilização da técnica *D&C* chamada *backtrack*.

O algoritmo de *backtrack* consiste na colocação ordenada e exaustiva de rainhas linha a linha até que configurações válidas sejam encontradas. Quando um determinado posicionamento possui rainhas em posição inválida, o algoritmo retorna à última configuração válida e prossegue avaliando novas posições.

A versão paralela do *N-Queens* por *D&C* envia uma configuração parcial de tabuleiro para cada tarefa e estas geram outras tarefas para resolver as possíveis posições de rainha na próxima linha. Esse procedimento é executado recursivamente até posicionar todas as rainhas. A figura 2.6, na página 25, mostra um exemplo da divisão de trabalho em um dos níveis da recursão. Uma tarefa recebe uma configuração de tabuleiro e gera as possíveis posições de rainha na próxima linha. Em seguida, a tarefa cria novas tarefas para resolver cada uma das configurações de tabuleiro.

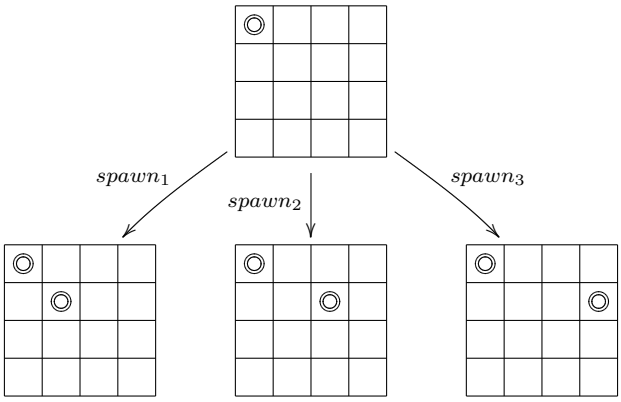


Figura 2.6: Exemplo de divisão recursiva do *N-Queens* em novas tarefas.

O algoritmo discutido é dinâmico devido ao DAG de dependências ser desconhecido antes da execução, além da imprecisão com relação a carga de trabalho em cada tarefa. Por exemplo, uma das configurações de tabuleiro na figura 2.6 não resulta em possíveis soluções porque possui duas rainhas na mesma diagonal. A tarefa que receber esse tabuleiro irá descartá-lo sem gerar novas tarefas. Por outro lado, as demais configurações poderão gerar novas tarefas e, portanto, podem ter uma carga de trabalho maior.

2.3.2 *Mergesort*

O *Mergesort* é um algoritmo de ordenação D&C que particiona os elementos recursivamente até um limite pré-determinado que corresponde ao início da ordenação sequencial. A versão desta seção é baseada no algoritmo *Cilksort* que acompanha o ambiente Cilk (FRIGO; LEISERSON; RANDALL, 1998). O algoritmo apresenta três fases de execução:

- **start** onde se efetua a leitura das entradas e envio destas para a tarefa responsável pela próximo fase;
- **sort** que divide recursivamente os elementos até um limite para aplicar a ordenação sequencial, e em seguida iniciar a união das partes;
- **merge** que combina as partes parcialmente ordenadas em um único conjunto ordenado.

O código 2.1, na página 26, apresenta a estrutura básica das chamadas recursivas no caso sequencial e omite o caso base. A entrada in de n elementos, assim como o auxiliar tmp, é divido em 4 partes de tamanho $n/4$ e estas são ordenadas recursivamente nas linhas 7 e 8. A linha 9 mostra a união das partes ordenadas *A* com *B* e *C* com *D*. Por fim, a linha 10 combina as duas partes originadas das duas uniões anteriores. A fase `merge` inicia com `median` que encontra as medianas de A e B onde o elemento médio de A (ma) é $n/2$ e de B (mb) é o maior elemento tal que seja menor ou igual a ma. As medianas formam um par (ma, mb) de forma que $ma + mb = (n + m)/2$ e todos os elementos em $A[1..ma]$ são menores que $B[mb..m]$, e todos de $B[1..mb]$ são menores que $A[ma..n]$.

Uma versão paralela pode ser obtida a partir desse exemplo ao substituir-se as chamadas recursivas por métodos de criação e sincronização de tarefas. Dessa forma, esses

```
 1: sort(in[1..n]) {
 2:   int tmp[1..n];
 3:   A = in;  B = A + n/4;
 4:   C = B + n/4; D = C + n/4;
 5:   tmpA = tmp;  tmpB = tmpA + n/4;
 6:   tmpC = tmpB + n/4; tmpD = tmpC + n/4;
 7:   sort(A, tmpA); sort(B, tmpB);
 8:   sort(C, tmpC); sort(D, tmpD);
 9:   merge(A, B, tmpA); merge(C, D, tmpC);
10:   merge(tmpA, tmpC, A);
11: }
12:
13: merge(A[1..n], B[1..m], C[1..(n+m)]) {
14:   median (A[1..n], B[1..m], ma, mb);
15:   merge(A[1..ma], B[1..mb], C[1..(n+m)/2]);
16:   merge(A[ma..n], B[mb..m], C[(n+m)/2..(n+m)]);
17: }
```

Código 2.1: Algoritmo sequencial de ordenação *Mergesort*

métodos executam concorrentemente e finalizam com uma sincronização da tarefa criada com seu criador.

2.3.3 Caixeiro Viajante

O problema do caixeiro viajante ou *Travelling Salesman Problem* (TSP) é um problema NP-completo para encontrar o menor caminho entre n cidades. Uma descrição formal do problema de decisão é:

$$\begin{aligned} TSP = \{\langle G, c, k\rangle : & \, G = (V, E) \text{ é um grafo completo,} \\ & \, c \text{ é uma função de } V \times V \rightarrow Z, \\ & \, k \in Z, \text{ e} \\ & \, G \text{ tem um ciclo Hamiltoniano com custo no máximo igual a } k\}. \end{aligned}$$

O grafo $G = (V, E)$ é um grafo não orientado completo que tem um custo inteiro não negativo $c(u, v)$ associado com cada aresta $(u, v) \in E$. O problema em questão é encontrar um ciclo Hamiltoniano de G com custo mínimo tal que seja $c(A)$ o custo total das arestas no subconjunto $A \subseteq E$ (CORMEN et al., 2002):

$$c(A) = \sum_{(u,v)\in A} c(u, v).$$

Uma implementação pode abordar este problema através de algoritmos exatos ou heurística de aproximação. Este trabalho utiliza o algoritmo exato de *branch-and-bound* (*B&B*) (NIEUWPOORT; KIELMANN; BAL, 2000) que possibilita eliminar grande parte do espaço de busca. Quando o tamanho da rota parcial atual e o tamanho mínimo do caminho que conecta as cidades restantes é maior que a melhor solução conhecida, a rota parcial é descartada. A versão paralela deste algoritmo distribui o espaço de busca entre as diferentes tarefas e implementa o caminho mínimo às demais cidades de forma redundante, ou seja, todas as tarefas conhecem os caminhos mínimos entre as cidades.

2.3.4 Números de *Fibonacci*

Os números de *Fibonacci* são uma sequência onde o primeiro número é 0, o segundo é 1 e cada subsequente é igual a soma dos 2 anteriores da sequência. O resultado de acordo com a sequência seria 0, 1, 1, 2, 3, 5, 8, etc. Em termos matemáticos, a sequência F_n é definida como:

$$F_n = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F_{n-1} + F_{n-2}, & \text{se } n > 1. \end{cases}$$

O código 2.2 ilustra a função *Fibonacci* na solução sequencial de acordo com o exemplo que acompanha o ambiente de programação Cilk (BLUMOFE et al., 1995). Se a entrada é menor que dois na linha 2, a recursão termina e a função retorna n. Caso contrário, calcula-se os números *Fibonacci* de $n-1$ e $n-2$ nas linhas 6 e 7. Em seguida, a tarefa atual espera o término das tarefas criadas e retorna a soma dos resultados nas linhas 8 e 9, respectivamente.

```
 1: int fibonacci(int n) {
 2:   if (n < 2) {
 3:     return n;
 4:   } else {
 5:     int x, y;
 6:     x = fibonacci(n - 1);
 7:     y = fibonacci(n - 2);
 8:     return (x + y);
 9:   }
10: }
```

Código 2.2: Código de uma função *Fibonacci* recursiva

A obtenção de uma versão paralela é semelhante à descrita no algoritmo de ordenação recursiva *Mergesort*. Assim, pode-se substituir as chamadas recursivas por métodos de criação e sincronização de tarefas.

2.4 Conclusão sobre o Capítulo

Este capítulo apresentou fatores relacionados à plataforma de execução e ao algoritmo paralelo que não podem ser desconsiderados e, às vezes, resultam em aplicações com carga de trabalho conhecida somente em tempo de execução. Esses fatores, tais como heterogeneidade e dinamismo, podem ser explorados através de uma técnica de programação. As técnicas *Mestre/Escravo* e *Divisão-e-Conquista* foram discutidas como uma forma de implementar algoritmos dinâmicos. Estas técnicas dependem do controle de granularidade para a obtenção de resultados eficientes. Entretanto, a implementação de um controle deste tipo é complexa devido às características relacionadas como gerenciamento de recursos, descrição de tarefa, localidade e carga de trabalho.

A implementação de algoritmos dinâmicos depende de um ambiente de programação que suporte um controle de granularidade eficiente com as características descritas. Através deste controle, uma aplicação dinâmica aumenta sua eficiência e desempenho significadamente. O próximo capítulo deste trabalho apresenta alguns ambientes de programação com suporte ao dinamismo e suas características com relação à granularidade.

3 AMBIENTES DE PROGRAMAÇÃO PARALELA COM DINAMISMO

O capítulo anterior apresentou a programação de algoritmos dinâmicos e técnicas de programação para explorar o dinamismo e heterogeneidade das plataformas distribuídas. Da mesma forma, constatou-se que a eficiência destes algoritmos depende de um controle de granularidade de tarefas, que pode ser tanto por agrupamento quanto por localidade. Todavia, a implementação de um controle eficiente é complicada devido às características necessárias que são: gerenciamento de recursos, descrição de tarefa, localidade e carga de trabalho.

Em síntese, a programação de algoritmos dinâmicos necessita de um ambiente de programação que suporte o dinamismo de tarefas e o controle de granularidade, de acordo com os aspectos relacionados. Este capítulo apresenta alguns ambientes de programação para algoritmos dinâmicos. Alguns destes ambientes não suportam a criação de tarefas dinamicamente ou o controle de granularidade, porém eles possuem características igualmente interessantes para algoritmos dinâmicos. Inicialmente, a seção 3.1 descreve os ambientes desenvolvidos para plataformas de memória compartilhada. Em seguida, a seção 3.2 apresenta os ambientes que focam a programação em plataformas distribuídas. A seção 3.3 traça um panorama dos ambientes apresentados com relação às características anteriormente citadas para um controle de granularidade eficiente e, por fim, uma conclusão deste capítulo é apresentada na seção 3.4.

3.1 Programação em Memória Compartilha

Esta seção descreve os ambientes de programação paralela para plataformas de memória compartilhada. Os ambientes apresentados são: **Cilk**, **OpenMP** e **Intel© TBB**.

Cilk (BLUMOFE et al., 1995) é um ambiente de programação paralela *multi-thread* para aplicações D&C em arquiteturas SMP. Ele estende a linguagem C através de primitivas, ou palavras-chave, que expressam o paralelismo da aplicação. Um programa Cilk sem palavras-chave é chamado de *C elision* e resulta em um programa C sintaticamente e semanticamente válido. Este ambiente implementa um escalonador de *threads* com roubo de tarefas (*Work Stealing*) em tempo de execução teoricamente eficiente. Em Cilk, as decisões de escalonamento são um conjunto de decisões definidas em tempo de compilação e execução (FRIGO; LEISERSON; RANDALL, 1998).

As primitivas de paralelismo e sincronização em Cilk são: `cilk`, `spawn`, `sync` e `return`. A primitiva `cilk` identifica uma função paralela ao ambiente definida como *Cilk procedure*. O paralelismo é iniciado na primitiva `spawn` que cria uma nova tarefa para a função especificada. A semântica desta primitiva difere de um método C porque

um `spawn` não espera pelo término da função invocada, ao contrário das chamadas em C que aguardam pelo término da execução do método. A primitiva `sync` atua como uma barreira local a fim de esperar o término das tarefas criadas pela atual. O ambiente insere um `sync` antes de todo `return` de forma implícita a fim de garantir que todas as tarefas filhas terminem antes da tarefa atual retornar.

A representação por uma DAG é a melhor maneira de visualizar a execução de um programa Cilk, como ilustrado na figura 3.1. A execução de um programa Cilk consiste em um conjunto de *procedures*, representados pelos retângulos da figura, de forma que cada *procedure* representa uma *thread* de usuário em execução. Os *procedures* são divididos em uma sequência de *Cilk threads* não bloqueantes, que são os vértices no DAG Cilk. A terminologia Cilk define uma *Cilk thread* como uma tarefa que possui uma sequência de instruções cujo término é determinado por uma das primitivas `spawn`, `sync` e `return`. A primeira *Cilk thread* que executa em um *procedure* é a *thread inicial* e as subsequentes são chamadas *successor threads*. Em tempo de execução, a relação binária do `spawn` causa a relação entre *procedures* ser estruturada como uma árvore. Além disso, as dependências entre *Cilk threads* formam uma DAG juntamente com essa árvore.

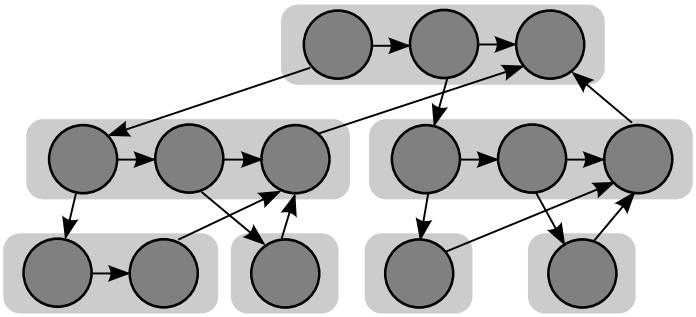


Figura 3.1: DAG de tarefas no ambiente Cilk

OpenMP (*Open Multi-Processing*) (CHAPMAN; JOST; PAS, 2007) é uma API de programação paralela para arquiteturas de memória compartilhada. Ele proporciona diretivas que possibilitam expressar paralelismo de dados, em trechos de código e laço, e paralelismo de tarefas, introduzido em sua versão 3.0. Sua API é constituída de diretivas de compilação, métodos de biblioteca e variáveis de ambiente que descrevem como o trabalho pode ser compartilhado entre diferentes *threads* que executam em diferentes processadores ou *cores*. O programador pode escolher o número de *threads* para execução através de métodos de biblioteca ou variáveis de ambientes. Além disso, o grão de tarefas no paralelismo de dados pode ser determinado pelo programador ou pelo compilador.

O paralelismo de tarefas permite a criação dinâmica de tarefas do tipo *Fork/Join* recursivamente com dependências explícitas de dados. A primitiva `task` antes de um trecho de código ou função determina uma tarefa, de forma que a *thread* que encontra essa primitiva pode executar a tarefa imediatamente ou adiar a execução. As tarefas são vinculadas à primeira *thread* por padrão, mas podem ser executadas por qualquer outra *thread* com a cláusula `notied` que garante maior eficiência em algoritmos dinâmicos. A espera por resultados de tarefas filhas é possível por meio da primitiva `taskwait`. O OpenMP 3.0 não especifica um algoritmo de escalonamento e designa esta escolha às implementações da API a fim de que elas determinem a melhor escolha em balanceamento de carga.

A biblioteca **Intel© TBB** (*Threading Building Blocks*) (REINDERS, 2007) possibilita expressar paralelismo na linguagem C++ para arquiteturas *multi-core*. Ela suporta o paralelismo de laços e tarefas, semelhante ao OpenMP, porém com diferentes aborda-

gens. TBB abstrai o conceito de tarefa para unidades de trabalho cujo grão é definido pela biblioteca no paralelismo de laços ou pelo programador no paralelismo de tarefas. Um conjunto de *threads* em modo usuário executam as tarefas disponíveis de acordo com o escalonamento por roubo de tarefas, inspirado no ambiente Cilk. Uma outra diferença significativa proporciona a utilização de programação genérica em laços paralelos de forma que estruturas de dados paralelas não se limitam aos tipos básicos da linguagem.

3.2 Programação em Memória Distribuída

Esta seção descreve os ambientes de programação paralela para plataformas de memória distribuída. Os ambientes desta seção são: **Cilk-NOW**, **Satin**, **KAAPI**, **AMPI** e **MPI**. A descrição sobre MPI é mais extensa devido a sua relação com a proposta deste trabalho, discutida no capítulo 4.

Cilk-NOW (BLUMOFE; LISIECKI, 1997) é um sistema desenvolvido como um subconjunto da linguagem Cilk para plataformas NOW. Este sistema se caracteriza principalmente pela utilização dinâmica de recursos e tolerância a falhas. Os recursos utilizados são alocados em tempo de execução conforme a sua ociosidade. Um processo em segundo plano monitora a utilização do recurso, que quando ocioso requisita a execução de tarefas. Se um usuário iniciar a utilização do recurso, o processo monitor cancela a execução de tarefas e aguarda um novo período de ociosidade.

A arquitetura Cilk-NOW se constitui basicamente de programas Cilk chamados *Cilk jobs* e *workers* que executam as tarefas de um *Cilk job*. Os *workers* executam nos recursos disponíveis e são controlados por um processo chamado *clearinghouse*. O escalonador de tarefas é uma extensão do proposto em Cilk, porém, os trabalhos relacionados não apresentam experimentos que confirmem sua eficiência teórica. Além disso, sua implementação não considera questões de escalabilidade devido ao controle centralizado de recursos na execução de um *Cilk job*.

Satin (NIEUWPOORT; KIELMANN; BAL, 2000) é um ambiente de programação inspirado em Cilk para aplicações D&C em plataformas largamente distribuídas como *cluster of clusters* e grades. Este ambiente estende a linguagem Java por meio de três primitivas semelhantes às correspondentes em Cilk: `spawn`, `sync` e `satin`. Um programa Satin sem essas primitivas é um programa sequencial Java igualmente válido. Suas tarefas são abstratas e escalonadas para processos em tempo de execução através de um algoritmo hierárquico chamado CRS (*Cluster-aware Random Stealing*) que é uma modificação do roubo de tarefas e elimina o escalonamento centralizado (NIEUWPOORT; KIELMANN; BAL, 2001). No envio e recebimento de dados, Satin permite a passagem de parâmetros por valor e a serialização de objetos sob demanda no momento que uma tarefas é roubada por outro processador. Este ambiente apresenta resultados eficientes devido ao escalonamento hierárquico em ambientes distribuídos e à serialização eficiente de parâmetros sob demanda. Porém, os experimentos não consideram os custos na utilização da máquina virtual Java.

KAAPI (GAUTIER; BESSERON; PIGEON, 2007) é uma biblioteca C++ para aplicações *multi-thread* em plataformas *multi-core*, aglomerados e grades. A sua interface é baseada na interface Athapascan com dependências explícitas entre tarefas através de primitivas. A primitiva `fork` cria uma tarefa que pode executar concorrentemente com outras, ao passo que `shared_r` e `shared_w` declaram um objeto como global para leitura e escrita, respectivamente. As comunicações em KAAPI utilizam um espaço de memória global chamado *global memory* que permite descrever as dependências entre

tarefas com acessos em objetos globais.

Uma tarefa KAAPI é descrita como uma estrutura abstrata chamada *closure* e representada como uma *thread* de usuário em execução. O escalonamento de tarefas ocorre em dois nível de forma que o ambiente escalona várias *threads* de usuário para um número fixo de processadores virtuais ou *threads* de núcleo (*kernel threads*) do mesmo processo em tempo de execução. Este escalonamento segue o algoritmo de roubo de tarefas, que é executado por uma *thread* de usuário para cada processo chamada *scheduler thread*. KAAPI possibilita a migração de *threads* de usuário caso o roubo seja entre diferentes nós. Os experimentos de trabalhos desenvolvidos com KAAPI mostram resultados eficientes para *clusters* e grades (GAUTIER; BESSERON; PIGEON, 2007; DANJEAN et al., 2007).

AMPI (*Adaptive MPI*) (HUANG; LAWLOR; KALé, 2004) é uma implementação do padrão MPI (*Message-Passing Interface*) na linguagem CHARM++ para plataformas de *cluster* e NOW. Este ambiente suporta o conceito de processadores virtuais ou *virtual processors* (VPs) e define processos MPI como *threads* de usuário em CHARM++. O ambiente possibilita a incorporação ou retirada de processadores em tempo de execução de forma a se adaptar a plataformas dinâmicas.

Através da virtualização, o número de processadores é transparente ao programador de forma que várias *threads* executam em um VP conforme o balanceamento de carga do ambiente. Este balanceamento de carga utiliza informações relacionadas à carga do sistema e ao padrão de comunicação em cada processador com a finalidade de avaliar as decisões de escalonamento. Além disso, o sistema pode distribuir a carga de trabalho por meio da migração de *threads* que é possibilitada pela alteração na alocação de memória das *threads* de usuário. O mecanismo de *Isomalloc* descreve um espaço global de memória dentro da memória virtual de cada VP de forma que este espaço é dividido em regiões mapeadas para cada *thread*. A pilha e *heap* da *thread* é alocada dentro de seu espaço mapeado de maneira que a migração para processadores diferentes não implique em conflitos de endereçamento de memória.

O ambiente AMPI contorna algumas questões relacionadas a implementação de algoritmos dinâmicos tais como balanceamento de carga e dinamismo de recursos. Todavia, ele não oferece a criação dinâmica de tarefas. Os trabalhos com experimentos em AMPI demonstram poucos ganhos com o balanceamento de carga, migração de *threads* e virtualização de processadores (HUANG; LAWLOR; KALé, 2004; HUANG et al., 2006).

MPI (FORUM, 1994, 1997) é uma interface de programação padrão amplamente empregada em aplicações com troca de mensagens, especialmente para plataformas de memória distribuída. O principal objetivo desta especificação é disponibilizar uma interface portável que possa ser implementada eficientemente em diferentes tipos de plataformas paralelas. Além disso, MPI não descreve detalhes de implementação e se concentra no comportamento esperado. Dessa forma, há várias implementações MPI atualmente com diferentes abordagens. Sua primeira versão enfatiza a descrição da interface de forma simples e eficiente que inclui: comunicações ponto-a-ponto e coletivas, criação estática de processos, etc. Uma nova versão, a MPI-2, inclui funcionalidades e melhorias ao padrão. As novas funcionalidades incluem: criação dinâmica de processos, operações em memória remota e operações de entrada e saída paralela (GROPP; LUSK; THAKUR, 1999). Além disso, as melhorias são compostas por: suporte a C++ e Fortran-90 e suporte ao uso de *threads*.

Um conceito fundamental em MPI é o de comunicadores, descrito desde a versão MPI-1. Eles são diretamente relacionados com outros dois conceitos: grupo e contexto.

Um grupo é um conjunto ordenado de processos identificados por um *rank*, enquanto um contexto divide o espaço de comunicações entre processos. Uma mensagem enviada em um contexto não pode ser recebida por um diferente contexto. Dessa forma, comunicadores são estruturas de dados compostas de contexto, grupo(s) de processos e demais atributos. Os comunicadores se dividem em dois tipos: intra-comunicador (*intra-communicator*) e inter-comunicador (*inter-communicator*). Um intra-comunicador descreve o escopo de um grupo de processos e sua topologia. Por exemplo, um programa MPI inicia a execução com um intra-comunicador pré-definido chamado `MPI_COMM_WORLD` que engloba todos os processos iniciados. Por outro lado, um inter-comunicador é uma ligação entre dois grupos de processos. A versão MPI-1 permite a criação de inter-comunicadores somente através da união entre dois intra-comunicadores, enquanto a versão MPI-2 gera um inter-comunicador na criação dinâmica de processos e na interface cliente/servidor, ambas descritas a seguir.

A versão do padrão MPI-1 especifica que o número de processos é fixo e determinado no momento da execução do programa (FORUM, 1994). Atualmente, a versão MPI-2 possibilita a criação dinâmica de processos através de dois conjuntos de funções. O primeiro conjunto engloba a função `MPI_Comm_spawn` e relacionadas que dinamicamente criam novos processos MPI. O processador de destino pode ser especificado nos atributos de um objeto do tipo `MPI_Info`, porém não há meios da aplicação controlar os recursos disponíveis em tempo de execução. O segundo conjunto consiste em funções para comunicações cliente/servidor que lembram métodos de comunicação por *sockets*. As funções de servidor incluem:

- **`MPI_Open_port`** para estabelecer um endereço de rede;
- **`MPI_Comm_accept`** que espera a conexão de um cliente;
- **`MPI_Publish_name`** que registra um endereço de rede em um servidor de nomes.

Por sua vez, as funções de cliente incluem:

- **`MPI_Comm_connect`** para estabelecer comunicação com um servidor cujo endereço é conhecido;
- **`MPI_Lookup_name`** para se obter um endereço de um servidor de nomes.

O padrão MPI não especifica o modelo de programação da aplicação, ou seja, um processo pode ser sequencial ou *multi-thread*. A partir da versão MPI-2, o MPI especifica dois requisitos para uma implementação MPI ser compatível com *threads* ou *thread-safe*: métodos MPI concorrentes devem ter resultados corretos e funções bloqueantes devem bloquear somente a *thread* que as executa. Um programa MPI que deseje usar *threads* precisa consultar o nível de concorrência suportado pela implementação MPI através da chamada `MPI_Init_thread`. Estes níveis de concorrência são:

- **`MPI_THREAD_SINGLE`** - apenas uma *thread*;
- **`MPI_THREAD_FUNNELED`** - várias *threads*, mas apenas a principal, que executa a função `MPI_Init_thread`, pode fazer chamadas MPI;
- **`MPI_THREAD_SERIALIZED`** - várias *threads* podem fazer chamadas MPI, mas uma de cada vez, ou seja, em um dado instante apenas uma das *threads* pode fazer chamadas MPI;

- **`MPI_THREAD_MULTIPLE`**- várias *threads* podem fazer chamadas MPI simultaneamente.

No padrão, não há definição de tarefa abstrata. A especificação menciona tarefas em certos trechos para se referir a processos. O programador pode utilizar processos e *threads* com o propósito de se beneficiar do paralelismo em dois níveis (LEOPOLD; Süß; BREITBART, 2006; LEOPOLD; SÜSS, 2006). Porém, o padrão não descreve a identificação de *threads* em um comunicador ou grupo. Além disso, poucas implementações MPI suportam o nível `MPI_THREAD_MULTIPLE`. Este suporte pode ser encontrado nas implementações MPI MPICH2 (GROPP, 2002) e na próxima versão estável do OpenMPI (GABRIEL et al., 2004) versão 1.3. Há trabalhos que abordam questões de implementação e desempenho em implementações MPI com suporte a *threads* (GROPP; THAKUR, 2007; THAKUR; GROPP, 2007) e otimizações na troca de mensagens (BUNTINAS; MERCIER; GROPP, 2007; BALAJI et al., 2008).

Da mesma forma, MPI não inclui questões de escalonamento entre processos dinamicamente criados. LAM-MPI (SQUYRES; LUMSDAINE, 2003) foi uma das primeiras implementações a suportar o dinamismo de processos, mas com limitações de escalonamento (CERA et al., 2006). As demais implementações não possuem suporte ou apresentam um escalonamento por *round-robin*. Outras propostas descrevem soluções para este problema tais como escalonamento centralizado (PEZZI et al., 2006; CERA et al., 2006) e hierárquico baseado em Satin (PEZZI et al., 2007) para aplicações D&C.

3.3 Visão Geral dos Ambientes de Programação

Após a apresentação dos ambientes de programação, esta seção traça um panorama destes ambientes com relação às características do controle de granularidade discutidos no capítulo anterior. Estas características, que proporcionam um controle de granularidade eficiente em aplicações dinâmicas, são: descrição e representação de tarefa, localidade de comunicações, carga de trabalho e gerenciamento de recursos. Além disso, este panorama relaciona o suporte a criação dinâmica de tarefas entre os ambientes.

- **Descrição e representação de tarefa** - o ambiente AMPI define tarefa como *thread* CHARM++ de usuário, ao passo que as implementações MPI implementam uma tarefa como um processo. Os demais ambientes descrevem uma noção abstrata de tarefa, porém representam tarefas de diversas formas. Cilk, OpenMP, Intel© TBB, Cilk-NOW e KAAPI executam tarefas como *threads* de usuário. Por outro lado, Satin executa como processos.

- **Localidade em comunicações** - neste aspecto, somente ambientes para plataformas distribuídas são relevantes. Cilk-NOW não considera questões de localidade. Satin e KAAPI aumentam a localidade ao mapear tarefas filhas em um mesmo recurso e otimizam comunicações entre recursos próximos em nível de memória compartilhada, aglomerados e grade. MPI especifica um atributo em comunicadores que permite a descrição da topologia dos processos. Dessa forma, o programador pode otimizar a localidade das comunicações. Além disso, as implementações MPI incluem algoritmos de comunicações coletivas otimizados para determinadas plataformas. AMPI aumenta a localidade ao migrar tarefas que comunicam frequentemente para um mesmo recurso.

- **Carga de trabalho** - Cilk, Intel© TBB, Cilk-NOW, Satin, KAAPI e AMPI suportam o balanceamento de carga entre recursos. OpenMP especifica que o usuário pode escolher o tipo de balanceamento, manual ou automático. O padrão MPI não descreve balanceamento de carga, que fica a cargo da implementação MPI.

- **Gerenciamento de recursos** - este aspecto, por si só, exclui ambientes de memória compartilhada. Assim, somente os ambientes de memória distribuição são relacionados. A inclusão de recursos em tempo de execução é suportado por Cilk-NOW, KAAPI, AMPI e Satin (WRZESINSKA; MAASSEN; BAL, 2007). O padrão MPI não especifica este tipo de aspecto no gerenciamento de recursos, mas isto é possível através da interface cliente/servidor entre diferentes processadores.

- **Criação dinâmica de tarefas** - nesta funcionalidade, somente o AMPI não suporta criação de tarefas em tempo de execução.

Este panorama demonstra abordagens diferentes entre os ambientes de programação para o controle da granularidade. O AMPI é um exemplo de ambiente que suporta este controle, mas não suporta a criação de tarefas sob demanda. O uso de VPs neste ambiente possibilita o uso de N VPs em M processadores físicos para qualquer número de tarefas. Se M ultrapassar o número de tarefas, as tarefas estaticamente definidas estão impossibilitadas de distribuir seu trabalho com estes novos recursos.

Em relação aos demais ambientes, as suas limitações restringem sua utilização na programação de alto desempenho (PAD). Os ambientes para programação em memória compartilhada apresentam resultados eficientes em problemas dinâmicos, mas estão focados em plataformas SMP e *multi-core*. Por outro lado, os ambientes de memória distribuída que suportam um controle de granularidade não empregam uma interface de programação padrão. O MPI, que é um padrão de fato em PAD, suporta a criação dinâmica de tarefas. Entretanto, as implementações MPI não suportam um controle de granularidade.

3.4 Conclusão sobre o Capítulo

Este capítulo listou uma série de ambientes de programação paralela para problemas dinâmicos. Estes ambientes estão divididos entre plataformas de memória compartilhada e distribuída. O panorama construído relacionou estes ambientes com as características de um controle de granularidade de acordo com os conceitos do capítulo 2. Este panorama aponta para os problemas encontrados nos ambientes de programação tais como restrição de plataforma, falta de uma interface de programação e ausência de controle de granularidade.

O próximo capítulo mostra uma proposta que incorpora um controle de granularidade com o padrão MPI. Esta proposta trabalha principalmente na criação e localidade das tarefas a fim de obter-se aplicações dinâmicas eficientes.

4 LIBSPAWN: CONTROLE DE GRANULARIDADE NO MPI

O capítulo 2 descreveu as características de plataformas estáticas e dinâmicas, além da necessidade de se implementar algoritmos dinâmicos com o propósito de explorar as características dinâmicas e heterogêneas destas plataformas. Duas técnicas de programação possibilitam o desenvolvimento de algoritmos dinâmicos, porém, constatou-se que um controle de granularidade em tempo de execução é fundamental para a eficiência da aplicação paralela.

Todavia, este controle necessita seguir características que dificultam a implementação da aplicação. Um ambiente de programação que suporte tais aspectos pode possibilitar a implementação de algoritmos dinâmicos. O capítulo 3 apresentou ambientes de programação para o desenvolvimento de aplicações dinâmicas, juntamente com o suporte dos ambientes aos aspectos relacionados à granularidade. Entretanto, as poucas arquiteturas suportadas e a falta de uma interface padrão dificultam a utilização destas ferramentas em PAD. O MPI é um padrão de fato em PAD e sua versão MPI-2 possibilita aplicações dinâmicas ao especificar a criação de tarefas como processos em tempo de execução. Porém, o MPI carece de um controle de granularidade para obter-se resultados eficientes.

Este capítulo descreve a proposta e implementação da biblioteca libSpawn que incorpora um controle de granularidade para programas MPI dinâmicos. Ela consiste em uma biblioteca que define uma tarefa e controla a granularidade em tempo de execução. Nesta proposta, o método `Spawn` do MPI-2 cria uma tarefa, ou seja, um fluxo de execução que pode ser tanto um processo quanto uma *thread*. O controle de granularidade determina o tipo de tarefa de acordo com três parâmetros propostos que avaliam o contexto de execução do programa: *cores* da arquitetura, carga e recursos de sistema. Dessa forma, esta proposta aumenta a localidade de tarefas e incorpora ao MPI os aspectos de controle de granularidade anteriormente discutidos.

A seção 4.1 mostra a versão anterior da biblioteca libSpawn, que se caracteriza pelo controle estático de granularidade. O controle de granularidade em tempo de execução proposto na versão atual é descrito na seção 4.2, juntamente com os parâmetros envolvidos. Em seguida, a seção 4.3 comenta as características sobre as comunicações entre tarefas com a libSpawn. Na sequência, a estrutura e implementação da libSpawn é apresentada na seção 4.4. A seção 4.5 constrói um panorama sobre os aspectos no controle de granularidade abordados neste trabalho de acordo com os apresentados no capítulo 2. Por fim, as limitações da implementação atual são discutidas na seção 4.6 e uma conclusão sobre este capítulo é apresentada na seção 4.7.

4.1 Primeira Versão: Granularidade Estática

A primeira versão da libSpawn (LIMA; MAILLARD, 2008a) sobrescreve dez chamadas MPI na linguagem de programação C. Os métodos MPI sobrescritos estão relacionados ao controle de granularidade e implementam a criação de tarefas em tempo de execução. Esta seção detalha as principais diferenças em relação à versão atual: controle de granularidade e comunicação entre tarefas.

O controle proposto nesta versão permite ao processo de um dado executável criar um número de *threads* menor ou igual ao limite de granularidade estabelecido. A configuração deste limite de granularidade é definida em tempo de compilação pela *macro* `MAX_THREADS` e utilizada na chamada `MPI_Comm_spawn` através de um contador global que contém o número de *threads* que estão em execução no processo. A criação de processos se inicia no momento que o número de *threads* ultrapassar este limite estabelecido.

A figura 4.1 mostra de forma simplificada os principais passos do algoritmo descrito. A chamada `MPI_Comm_spawn` concentra o controle implementado pela biblioteca em que dois testes decidem se a tarefa será um processo pesado ou leve. O primeiro em (1) compara o comando da nova tarefa com o nome do executável que deu origem ao processo atual, e se diferentes um novo processo é criado. O próximo teste em (2) valida o número de *threads* no processo em relação ao limite estabelecido. Se ultrapassado, um novo processo deve ser criado. A criação de um processo padrão MPI está em uma região crítica devido ao fato da chamada `MPI_Comm_spawn` original não permitir sua execução concorrente.

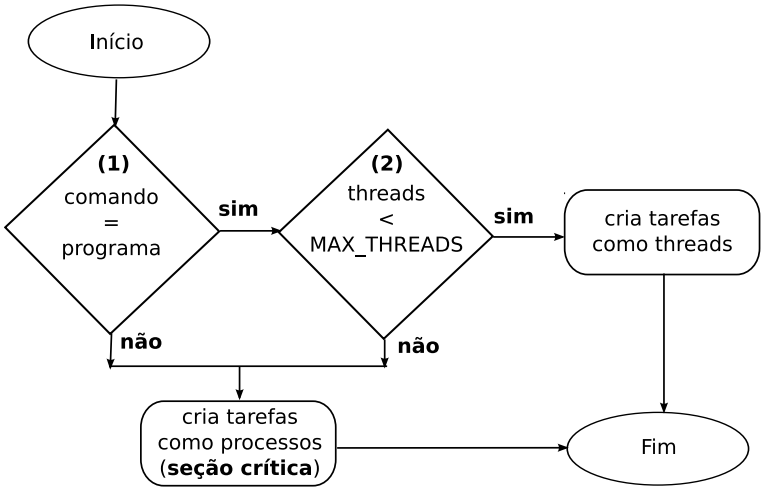


Figura 4.1: Fluxograma da função `MPI_Comm_spawn` na primeira versão da libSpawn

A libSpawn desta versão faz um controle simplificado da comunicação entre *threads* do mesmo processo nas chamadas `MPI_Send` e `MPI_Recv`. O padrão MPI define a identificação de mensagens pelo nome *message envelope* (FORUM, 1994). Esta versão implementa uma estrutura de dados que identifica as mensagens e permite a transferência de dados entre requisições correspondentes. Cada processo mantem duas listas que controlam as requisições de envio e recebimento de mensagens incompletas com exclusão mútua a fim de evitar acessos indevidos.

As tarefas envolvidas na troca de mensagens executam a inserção nas listas de espera e o bloqueio na espera. Porém, a comparação e finalização de uma mensagem é independente das tarefas. Uma *thread* criada na inicialização de cada processo realiza buscas em

ambas as listas para encontrar mensagens correspondentes através do *message envelope*. Essa *thread* transfere os dados necessários entre mensagens correspondentes e muda o estado da comunicação para finalizada. As tarefas que esperam pelas respectivas mensagens terminam a espera assim que voltarem a serem escalonadas para execução.

O código 4.1 mostra o algoritmo desenvolvido no controle de mensagens. O laço da linha 2 itera a cada nova mensagem de uma tarefa e inicia sua busca na primeira de cada lista. O laço da linha 5 percorre ambas as listas e testa na linha 6 se as mensagens de envio e recebimento são correspondentes de acordo com os identificadores descritos no *message envelope*. Se elas correspondem, os dados da mensagem são transferidos e as tarefas envolvidas voltam a ser escalonadas na linha 8 por meio de uma variável condicional.

```
 1: lock(mutex);
 2: enquanto verdadeiro faça
 3:   send = primeiro_da_lista_send;
 4:   enquanto tem send faça
 5:     receive = procura_receive_correspondente (send)
 6:     se receive existe então
 7:       completa_ambos(send, receive);
 8:       sinaliza_tarefas(send, receive);
 9:     fim se
10:     send = send.próximo;
12:   fim enquanto
13:   condição_espera(condição, mutex);
14: fim enquanto
```

Código 4.1: Algoritmo para completar comunicações na versão anterior da libSpawn

Os resultados obtidos demonstram ganhos significativos com experimentos baseados na técnica D&C (LIMA; MAILLARD, 2008a). Todavia, o problema deste controle está na definição do limite de granularidade em tempo de compilação (`MAX_THREADS`). A obtenção de uma configuração eficiente exige experimentos que não serão válidos para diferentes arquiteturas. Além disso, este controle desconsidera aspectos relacionados à carga do sistema tais como interferência de outros aplicativos e serviços. Dessa forma, um valor de granularidade alto pode degradar o desempenho de uma aplicação ao sobrecarregar os processadores disponíveis e os recursos de sistema operacional. A fim de contornar estas limitações, a seção 4.2 detalha a solução atual que controla a granularidade de tarefas em tempo de execução.

4.2 Versão Atual: Granularidade em Tempo de Execução

No contexto do MPI, uma aplicação pode controlar a granularidade de tarefas através da decisão entre criar processos ou *threads*. Este controle pode reduzir os custos de comunicação e criação de tarefas. Entretanto, a implementação do controle na aplicação é trabalhosa e seu desempenho dependerá do contexto de execução.

A biblioteca proposta implementa um controle de granularidade que decide entre criar tarefas como processos ou criar tarefas como *threads* em tempo de execução. Este controle considera parâmetros sensíveis ao contexto de execução como características de arquitetura e sistema operacional. Dessa forma, uma tarefa cria outras dinamicamente e

o controle de granularidade avalia a alternativa mais eficiente para a execução destas de acordo com os parâmetros considerados. O controle considera três parâmetros relacionados à arquitetura e ao sistema operacional:

- **Número de *cores* da arquitetura**. Este parâmetro não varia em tempo de execução e seu valor é configurado individualmente por processo em um determinado nó;
- **Carga de sistema**. Cada processo lê a carga do sistema em tempo de execução com a finalidade de avaliar se há sobrecarga do sistema;
- **Recursos de sistema**. A criação de *threads* depende do limite por processo estabelecido pelo sistema operacional, dessa forma a granularidade de processos leves se altera conforme a utilização de recursos de sistema em tempo de execução.

O primeiro parâmetro considerado proporciona adaptar a granularidade inicial de acordo com o número de *cores* disponíveis na arquitetura. A cada chamada `Spawn`, o número de *threads* em execução é comparado ao número de *cores* da arquitetura. Se o número de *threads* é menor que o de *cores*, os parâmetros seguintes não são avaliados. Dessa forma, ele simplifica o controle ao determinar se os demais parâmetros devem ser testados. O valor deste parâmetro não se altera em tempo de execução e depende do nó onde o processo está em execução.

O parâmetro seguinte é a carga de sistema em tempo de execução. Após o número de tarefas existentes se igualar ao número de *cores* da arquitetura, uma ou mais dessas tarefas podem estar bloqueadas por chamadas de sistema ou à espera de uma comunicação com outra tarefa. A carga de sistema informa o número de tarefas escalonadas para execução e através deste valor é possível detectar se os *cores* do nó estão ocupados. Em caso afirmativo, o controle decide lançar processos a fim de atribuir as nova tarefas para outros nós possivelmente ociosos. Por outro lado, se um dos *cores* estiver ocioso, o controle cria *threads* e considera o próximo parâmetro de granularidade.

Por último, o controle de granularidade considera os recursos de sistema disponíveis aos processos em execução. O sistema operacional limita a utilização de recursos por parte dos processos no sistema. Estes recursos podem ser tanto físicos, tais como memória ou armazenamento, quanto funcionais. Os limites funcionais estão relacionados à implementação do sistema operacional e dependem de configurações e políticas de administração específicas. Como exemplos, o número de *threads* e descritores de arquivos utilizados por processo são limites funcionais de implementação e configuração, respectivamente, em sistemas UNIX. Dessa forma, o controle proposto aloca recursos à medida que novas *threads* e espaços de memória são requisitados. Um processo em outro nó é necessário no momento que o sistema impossibilitar a alocação de recursos para as tarefas localmente criadas.

A figura 4.2, na página 39, ilustra a execução de um programa dinâmico MPI qualquer com tarefas distribuídas entre quatro nós. Os passos na execução estão numerados para que possam ser esclarecidos conforme o tipo de tarefa criada. A tarefa $t0$ inicia sua execução no processo $Processo\ 0$, que lê o número de *cores* disponíveis e monitora a carga do sistema. À medida que os dois primeiros parâmetros são avaliados, a tarefa $t0$ origina mais tn tarefas como *threads* (1). A partir deste ponto, o controle reconhece que: o número de tarefas excede o de *cores* e estes *cores* estão ocupados, ou o número de tarefas ultrapassa os recursos de sistema. Assim, a biblioteca lança processos pesados para outro nó como representado na operação `Spawn` em (2). O $Processo\ 1$ ocupa $nodo\ 2$ e passa a criar novas *threads* (3) como tarefas de acordo com a avaliação de parâmetros descrita.

As demais tarefas seguem o mesmo procedimento de avaliação onde as operações `Spawn` criam processos a partir de $Processo\ 1$ (4) e $Processo\ 2$ (6) em $nodo\ 4$ (5) e $nodo\ 3$ (7), respectivamente.

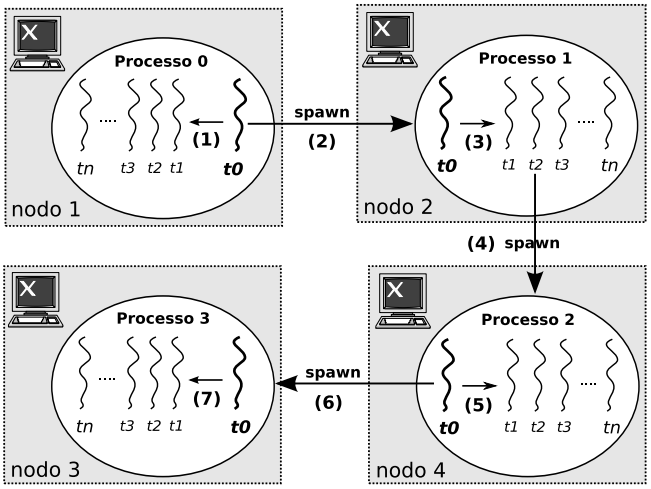


Figura 4.2: Exemplo do controle de granularidade proposto com processos e *threads*

O destino dos processos originados em chamadas `Spawn` é determinado pelo gerenciador de processos da implementação MPI sem qualquer interferência do controle proposto. Porém, um ponto que pode ser observado ocorre em casos que todos os nós estão ocupados e o controle detecta sobrecarga em um destes. Uma possível solução é centralizar o controle e determinar o nó com a menor sobrecarga de sistema. Todavia, o capítulo 2 demonstrou as limitações no controle centralizado de tarefas. A implementação MPI determina o destino de acordo com o nó com o menor número de processos em execução. Assim, pode-se pressupor que o gerenciador de processos realiza uma distribuição uniforme entre os nós.

Esta proposta controla a granularidade por aumento da localidade. O controle utiliza os três parâmetros descritos a fim de considerar a arquitetura e a carga de sistema. Entretanto, uma implementação MPI não é otimizada para a troca de mensagens entre *threads* e os comunicadores não identificam *threads* de um processo. A próxima seção descreve a solução deste trabalho para as comunicações entre tarefas.

4.3 Comunicações entre Tarefas

A programação de aplicações dinâmicas em MPI impõe restrições de comunicação entre tarefas criadas (filhas) e seu criador (pai). Essa comunicação entre tarefas de acordo com o padrão MPI se aplica através da troca de mensagens entre processos. Entretanto, as *threads* não são identificadas como tarefas nas implementações MPI. Além disso, a aplicação deve evitar comunicações incoerentes que resultem em impasses (GROPP; THAKUR, 2007). Portanto, um mecanismo de comunicação entre tarefas que considere os problemas apresentados é essencial ao funcionamento do controle de granularidade proposto. A seção anterior demonstrou as características do controle de granularidade proposto na libSpawn, ao passo que esta seção caracteriza as comunicações entre tarefas.

A libSpawn implementa os métodos de troca de mensagens com a finalidade de determinar se a comunicação será entre processos ou *threads*. Se a comunicação é entre processos, o método MPI correspondente é iniciado sem interferência da biblioteca. Por

outro lado, a biblioteca manipula inteiramente a comunicação entre *threads* através de procedimentos em memória compartilhada com exclusão mútua. Dessa forma, a biblioteca elimina impasses entre *threads* e possibilita comunicações eficientes sem o custo de chamadas MPI adicionais.

Na troca de mensagens, a operação de transferência não faz com que os endereços de memória recebidos como parâmetro sejam compartilhados pelas diferentes *threads* e, ao invés disso, os dados de envio são copiados diretamente ao endereço de destino. O padrão MPI prevê este comportamento com operações comunicantes e utilizar compartilhamento requer a definição de atributos aos endereços em questão.

A biblioteca nesta proposta não trata variáveis globais que ofereçam possíveis inconsistências nos resultados da aplicação. Outros trabalhos abordam este problema por meio de soluções em tempo de compilação que detectam declarações globais e as transformam em variáveis com valores locais a cada *thread* (DEMAINE, 1997; TANG; YANG, 2001). Os programas que utilizem a solução proposta devem estar livres de variáveis compartilhadas devido a impossibilidade de se prever a representação de uma tarefa em execução.

4.4 Estrutura e Implementação da libSpawn

A biblioteca deste trabalho implementa o controle de granularidade descrito anteriormente, assim como a troca de mensagens entre tarefas. A implementação usa o paradigma de programação orientado a objetos na linguagem de programação C++. Esta seção descreve as principais características da implementação, que são: o suporte ao MPI, estrutura de classes, descrição de tarefa, controle de granularidade, representação de uma mensagem e comunicação entre tarefas.

4.4.1 Suporte ao Padrão MPI

libSpawn implementa um subconjunto das classes MPI para troca de mensagens, além dos métodos básicos de inicialização e término de um programa. As diferenças entre as classes MPI e libSpawn incluem a redefinição da criação dinâmica de processos e comunicação de tarefas. Os métodos comunicantes englobam intra-comunicadores e inter-comunicadores, ambos com métodos bloqueantes e um subconjunto dos não-bloqueantes.

A tabela 4.1, na página 41, lista o subconjunto de métodos MPI implementados na libSpawn. A definição do *namespace* `MPI2` e a declaração de métodos semelhante ao da especificação MPI proporcionam alterações mínimas em códigos pré-implementados que estão de acordo com o padrão MPI-2. A inicialização da libSpawn limita o nível de concorrência para `MPI_THREAD_MULTIPLE` devido à utilização de tarefas como *threads*. Além disso, o suporte ao dinamismo de tarefas está restrito ao conjunto de funções relacionadas à chamada `Spawn`.

4.4.2 Estrutura de Classes

O diagrama de classes da figura 4.3, na página 41, ilustra as classes descritas no padrão MPI e as específicas da libSpawn. Quatro destas classes são do padrão MPI e uma classe, *Task*, é específica do controle de granularidade. *Task* caracteriza uma tarefa e implementa o controle de granularidade, além de estar associada com outras duas que tratam da troca de mensagens: *Request* e *Comm*.

A classe abstrata *Comm* é o comunicador de tarefas e as organiza em grupos através de um identificador ou *rank*. Essa classe atua como uma super-classe e especifica os protótipos de comunicação sobrecarregados pelas subclasses *Intracomm* e *Intercomm*.

namespace MPI2		
`Init`	`Comm.Spawn`	`Comm.Free`
`Finalize`	`Comm.Get_rank`	`Request.Free`
`Comm.Send`	`Comm.Get_size`	`Request.Test`
`Comm.Isend`	`Comm.Is_inter`	`Request.Wait`
`Comm.Recv`	`Comm.Disconnect`	`Request::Testall`
`Comm.Irecv`	`Comm::Get_parent`	`Request::Waitall`

Tabela 4.1: Métodos do padrão MPI implementados na libSpawn

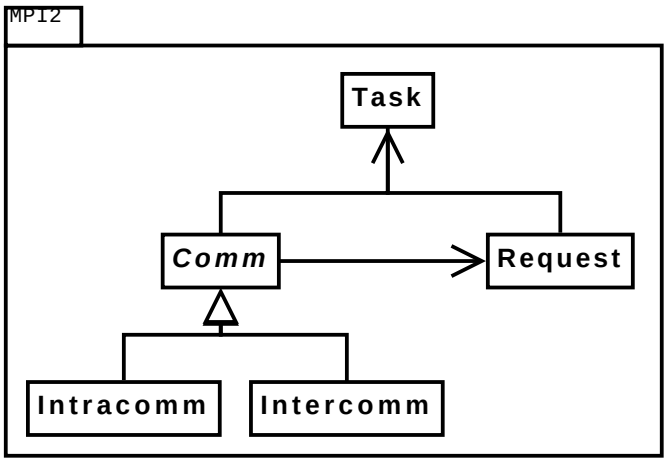


Figura 4.3: Diagrama de classes UML com a estrutura básica da libSpawn

Estas descrevem um conjunto de tarefas em um grupo e entre diferentes grupos (FORUM, 1994), respectivamente. Além disso, *Comm* controla as mensagens que aguardam para ser completadas. *Request* é uma super-classe que abstrai o tipo de comunicação na troca de mensagens. As suas classes especializadas determinam se a comunicação é entre processos (*RequestProcess*) ou *threads* (*RequestTask*). Toda troca de mensagens gera uma instância de *Request* para envio e recebimento. A classe *RequestTask* interage com a classe *Comm* a fim de inserir, remover e consultar o término de mensagens.

4.4.3 Descrição de uma Tarefa

Esta implementação descreve uma tarefa como um fluxo de execução que pode ser tanto um processo quanto uma *thread* POSIX transparente ao programador. A classe *Task* descreve uma tarefa MPI através de quatro campos que incluem comunicadores e variáveis de sincronização.

O código 4.2, contido na página 42, mostra os atributos da classe *Task*. A variável `thread_atual` na linha 2 representa o identificador da *thread* designado pela biblioteca POSIX Threads. Entre os comunicadores, `comm_world` na linha 3 declara o intra-comunicador pré-definido `MPI2::COMM_WORLD` que representa as novas tarefas originadas na chamada `Spawn`. `comm_self` na linha 4 define o intra-comunicador pré-definido `MPI2::COMM_SELF` que é individual para cada tarefa. O último comunicador na linha 5, `comm_parent`, define o inter-comunicador das tarefas criadas com seu criador. Na linha 6, `barreira` permite sincronizações de barreira entre um conjunto de tarefas, a exemplo da chamada `MPI::Comm.Barrier` onde a execução continua so-

mente se todas as tarefas executem essa função no comunicador correspondente.

```
1: class Task {
2:   pthread_t          thread_atual;
3:   Intracomm          comm_world;
4:   Intracomm          comm_self
5:   Intercomm          comm_parent;
6:   pthread_barrier_t  barreira;
7: };
8:
9: __thread Task *tarefa_atual;
```

Código 4.2: Descrição da Classe Task

Uma nova instância de *Task* é criada a cada nova tarefa e os atributos descritos são configurados. A variável global `tarefa_atual` do código 4.2 define um ponteiro à uma instância da classe *Task*, que armazena os atributos da tarefa atual, distinta para cada *thread*. O atributo `__thread` é uma extensão do compilador GNU (FOUNDATION, 2008) e proporciona valores distintos entre *threads* com o *Thread-Local Storage* (TLS) (DREPPER, 2005). O TLS especifica 2 novas seções em binários do formato ELF (*Executable and Linkable Format*) chamados `.tbss` (dados não-inicializados) e `.tdata` (dados inicializados). Dessa forma, o acesso ao valor da variável não depende de outras técnicas de sincronização e exclusão mútua.

4.4.4 Controle de Granularidade

A libSpawn implementa o controle de granularidade por meio da chamada `Spawn` na classe *Intracomm* do padrão MPI. A escolha entre criar processos ou *threads* envolve parâmetros relacionados à arquitetura de execução, carga de sistema e disponibilidade de recursos em tempo de execução. A seção 4.2 apresentou estes parâmetros e seus conceitos, ao passo que esta seção detalha cada parâmetro no contexto da implementação.

O fluxograma da figura 4.4, na página 43, demonstra os passos na decisão entre a criação de tarefas como processos ou *threads*. Os testes nessa escolha estão numerados para que possam ser esclarecidos de acordo com a ordem dos parâmetros.

O teste (1) compara o comando da nova tarefa com o nome do programa em execução a fim de permitir aplicações MPMD (*Multiple Programs Multiple Data*). Se estes nomes diferem, as tarefas serão processos do programa especificado. Caso contrário, os demais parâmetros são avaliados. O teste (2) corresponde à comparação entre o número de *cores* da arquitetura e o número de *threads* em execução no processo. Se o número de *threads* é menor que o de *cores*, o próximo teste, teste (3), não é avaliado devido à existência de *cores* ociosos para execução e as tarefas resultam em *threads*. Por fim, o teste (3) avalia se o sistema está sobrecarregado tanto com tarefas do processo atual quanto com outras tarefas de sistema. O valor da carga de sistema é uma média de leituras do sistema operacional (CERA et al., 2006).

Após os testes anteriores, a decisão do controle consiste em criar as tarefas como *threads* do processo atual. O teste (4) verifica a existência de falhas na criação destas *threads*. Conforme descrito anteriormente, este parâmetro é definido de acordo com os limites de recursos aos processos em execução tais como descritores de arquivos, memória e fluxos de execução. Portanto, testar a criação de *threads* é fundamental na verificação

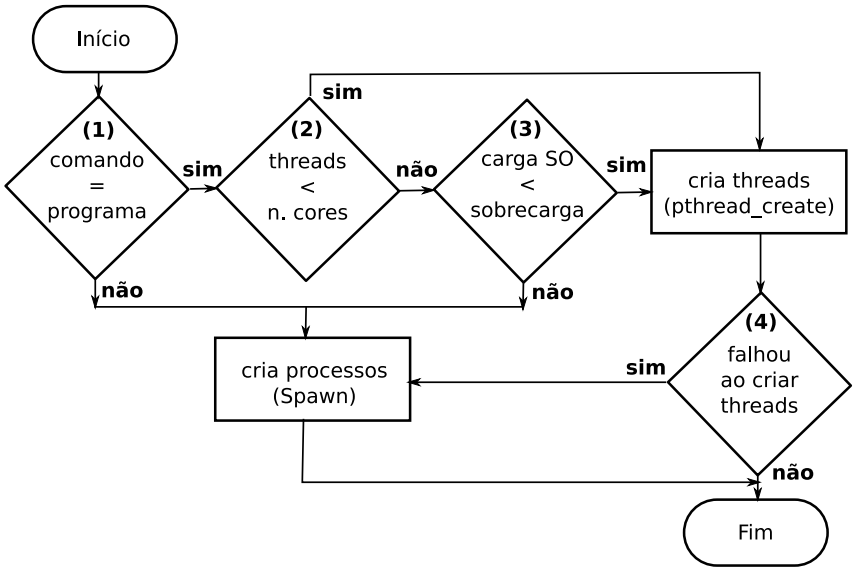


Figura 4.4: Fluxograma da função `Spawn` na libSpawn

dos recursos disponíveis. No momento que uma falha acontecer, o controle cancela as *threads* criadas antes do erro e cria estas tarefas como processos.

4.4.5 Representação de uma Mensagem

A manipulação das comunicações na libSpawn exige uma forma de representar mensagens de maneira que possibilite a comunicação em memória compartilhada e de acordo com o padrão MPI. A classe *RequestTaskContainer* descreve uma mensagem de envio e recebimento através de dois atributos: um booleano e um identificador da mensagem. O booleano armazena o estado atual da mensagem, incompleto ou completo, e sua alteração depende do encontro de uma mensagem correspondente.

Por último, o identificador de mensagens é implementado por meio da classe *MsgEnvelope*. Esta classe contém os atributos necessários ao *message envelope* do padrão MPI mais um ponteiro com o endereço de memória referente ao *buffer* das mensagens. O *message envelope* carrega informações usadas para distinguir mensagens por meio de um número fixo de campos:

- *Source* - origem da mensagem;
- *Destination* - destino da mensagem;
- *Tag* - identificador de mensagens;
- *Communicator* - comunicador utilizado.

4.4.6 Comunicação entre Tarefas

Os comunicadores em MPI são fundamentais à troca de mensagens entre tarefas. Um intra-comunicador representa um conjunto de tarefas que podem ter sido criadas por outro conjunto de tarefas através de operações em comunicadores ou da criação dinâmica de tarefas, discutido no capítulo 3. A comunicação entre dois intra-comunicadores depende do inter-comunicador gerado na criação de tarefas (`Spawn`) ou por meio de operações `Connect` e `Accept` para estabelecer comunicação entre comunicadores não relacionados.

Estes comunicadores englobam atributos essenciais para a troca de mensagens entre tarefas (FORUM, 1997) como o identificador ou *rank*. Por outro lado, um comunicador não identifica *threads* concorrentes e o programador deve evitar situações que resultem em um estado de impasse (*deadlock*) (GROPP; THAKUR, 2007). Dessa forma, o controle de granularidade proposto contorna estes impasses ao manipular a troca de mensagens entre tarefas de um mesmo processo.

A figura 4.5 mostra o diagrama de classes do mecanismo de comunicação implementado. Os comunicadores especializados de *Comm* dependem de *Request* que pode ser uma instância para comunicação entre processos ou *threads*. Há duas especializações da classe *Request*: *RequestProcess* e *RequestTask*. *RequestProcess* implementa as comunicações entre tarefas de diferentes processos e apresenta funções simples que utilizam métodos MPI sem alterações. Por outro lado, a classe *RequestTask* implementa a troca de mensagens entre tarefas de um mesmo processo através de métodos de sincronização em memória compartilhada. *RequestTaskContainer* e *MsgEnvelope* definem o formato de uma mensagem MPI entre *threads* como explicado na seção 4.4.5.

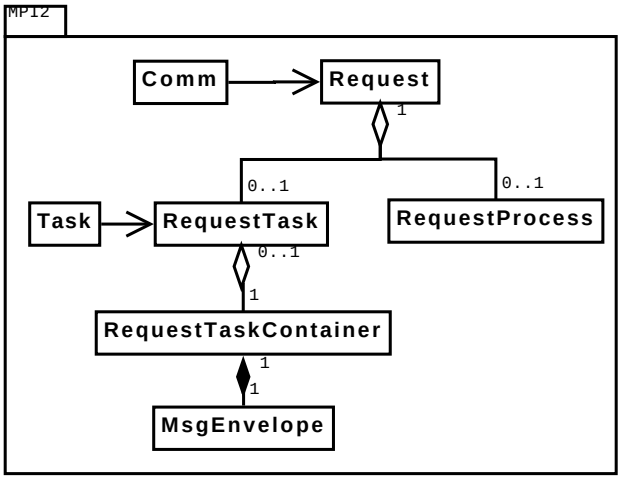


Figura 4.5: Diagrama de Classes na comunicação do controle de granularidade

O processo de envio e recebimento de mensagens se inicia na classe *Comm* e subclasses *Intracomm* e *Intercomm*. O código 4.3, na página 45, ilustra a sequência de passos executados por uma chamada `Send`. Inicialmente, a linha 3 identifica se a mensagem será entre tarefas em diferentes processos ou não. Em seguida, um *Request* é criado como uma instância de *RequestTask* ou *RequestProcess* de acordo com o tipo de comunicação nas linhas 4 e 6, respectivamente. Por fim, o método *Wait* aguarda a finalização da comunicação e mantém a tarefa bloqueada. Este método na classe *RequestTask* bloqueia a tarefa atual através de uma variável condicional contida no comunicador.

A implementação de *RequestTask* apresenta características mais complexas devido ao controle da comunicação entre as *threads*. Ela executa métodos contidos na classe *Comm* para a manipulação das mensagens não concluídas. A classe *Comm* mantém duas listas duplamente encadeadas que correspondem às mensagens `Send` e `Recv` não concluídas. Uma mensagem é inserida em sua respectiva lista a cada instância de *RequestTask* onde um objeto da classe *RequestTaskContainer* recebe os atributos de identificação, discutidos anteriormente. As tarefas envolvidas na troca de mensagens executam a inserção nas listas de espera e, se possível, a transferência de dados. Há duas possibilidades das mensagens serem completadas em *RequestTask*: na instância da classe ou no método `Wait`.

```
1: void Send(...) {
2:   Request *request;
3:   if (isCommThreads())
4:     request = new RequestTask(...);
5:   else
6:     request = new RequestProcess(...);
7:   request.Wait();
8: }
```

Código 4.3: Código simplificado de uma chamada `Send`

O código 4.4 apresenta os passos na instância da classe *RequestTask* para uma mensagem `Send` ou `Recv`. A instância de um objeto *RequestTask* cria um *RequestTaskContainer* na linha 1. Em seguida, a função entra em uma seção crítica no comunicador, linha 2, para procurar uma mensagem correspondente nas listas de mensagens, linha 3. Se a mensagem é encontrada, ambas são finalizadas na linha 5 sem a inserção da nova mensagem. Caso contrário, a nova mensagem é inserida em uma das listas na linha 7. Por fim, a tarefa deixa a seção crítica do comunicador na linha 9.

```
1: msg = new RequestTaskContainer(...);
2: comunicador.lock();
3: msg_alvo = comunicador.procura_correspondente(msg);
4: se msg_alvo não é nula faça
5:   comunicador.completa_mensagens(msg, msg_alvo);
6: senão faça
7:   comunicador.insere(msg);
8: fim se
9: comunicador.unlock();
```

Código 4.4: Algoritmo de uma instância da classe *RequestTask* da libSpawn para as funções `Send` ou `Recv`

O código 4.5, na página 46, apresenta o término de mensagens através da chamada `Wait`, que bloqueia a tarefa atual até que a comunicação seja concluída. A linha 1 inicia a seção crítica no comunicador da mensagem e, em seguida, a busca pela mensagem correspondente é iniciada. Se ela não é encontrada, a tarefa aguarda o comunicador avisar quando novas mensagens surgirem. Caso contrário, a tarefa completa a comunicação na linha 8 e avisa ao comunicador sobre este evento na linha 9. Por fim, a tarefa deixa a seção crítica na linha 11. Conforme descrito, a tarefa opta por duas opções: espera outra tarefa completar a comunicação ou ela mesma completa. A escolha de uma das opções depende do encontro da mensagem correspondente a fim de ser finalizada.

Este método de comunicação entre tarefas oferece melhorias significativas em relação a versão anterior da libSpawn, descrita na seção 4.1. A antiga versão mantem duas listas de mensagens por processo com uma *thread* para gerenciar o término das mensagens. Porém, essa solução é ineficiente para um número alto de tarefas como *threads* porque a espera por mensagens limita a concorrência das *threads*. A implementação desta seção transfere as listas de mensagens para o comunicador a fim de restringir o número de tarefas que esperam por uma mensagem. Dessa forma, a concorrência entre *threads* aumenta significadamente.

```
 1: comunicador.lock();
 2: msg_alvo = comunicador.procura_correspondente(msg);
 3: se msg_alvo é nula faça
 4:   enquanto msg não está completa faça
 5:     comunicador.espera_nova_mensagem();
 6:   fim enquanto
 7: senão faça
 8:   comunicador.completa_mensagens(msg, msg_alvo);
 9:   comunicador.notifica_tarefas();
10: fim se
11: comunicador.unlock();
```

Código 4.5: Algoritmo de uma função `Wait` para o término de mensagens na classe *RequestTask* da libSpawn

4.5 Aspectos de Granularidade Abordados

O capítulo 2 discutiu os aspectos que melhoram a eficiência de um controle de granularidade em algoritmos dinâmicos. Esta seção demonstra de forma sucinta a maneira que esta proposta trabalha com estes aspectos, juntamente com o dinamismo de tarefas. Assim, os aspectos e abordagens da libSpawn são:

- **Descrição e representação de tarefa** - esta proposta não define uma tarefa como uma entidade abstrata. Por outro lado, sua representação é transparente ao programador e consiste em dois tipos: processo ou *thread*. Um processo é criado pela chamada `Spawn` quando o controle de granularidade decide não criar mais tarefas no nó em questão, ao passo que uma *thread* POSIX (`pthread_create`) surge quando se decide aumentar a localidade;

- **Localidade em comunicações** - este controle de granularidade aumenta a localidade no momento que as tarefas são criadas como *threads* localmente no mesmo nó. Além disso, a libSpawn implementa comunicadores que se especializam entre referências remotas (entre processos) e locais (entre *threads*) com a finalidade de otimizar a troca de mensagens entre tarefas;

- **Carga de trabalho** - não há um balanceamento de carga centralizado nesta proposta. O controle de granularidade coleta a carga de sistema em tempo de execução a fim de determinar se os processadores ou *cores* estão ocupados. Esta coleta é feita por cada processo em execução;

- **Gerenciamento de recursos** - libSpawn não gerencia os recursos disponíveis para execução. Por outro lado, o gerenciador de processos da implementação MPI controla o destino dos processos dinamicamente criados. Dessa forma, esta proposta pressupõe que o gerenciador da implementação MPI distribui os processos igualmente entre os nós disponíveis. Porém, a inclusão de recursos sob demanda não é suportada pela implementação MPI utilizada, o MPICH2 (MPICH, 2009);

- **Criação dinâmica de tarefas** - esta proposta possibilita a criação dinâmica de tarefas tanto como processos ou *threads*. Dessa forma, o desempenho na criação sob demanda melhora significadamente.

4.6 Limitações da Biblioteca Proposta

A biblioteca libSpawn oferece um controle de granularidade e primitivas de criação de tarefas e comunicação. Todavia, a implementação apresenta limitações em alguns aspectos, entre eles sincronização e inconsistências em variáveis compartilhadas. A seguir, uma lista de pontos que não preenchem as exigências do padrão MPI é demonstrada:

- Um `Spawn` por intermédio deste controle possibilita criar tarefas somente como um conjunto de processos ou um conjunto de *threads*. A criação híbrida de processos e *threads* não é suportada devido a necessidade de alterações no mecanismo de comunicação descrito;
- Na versão atual, não há suporte à execução coletiva do método `Spawn` em um comunicador, ou seja, a implementação atual suporta somente uma tarefa pai para várias tarefas criadas;
- A questão das variáveis globais foi discutida anteriormente no contexto das sincronizações entre tarefas. A utilização de variáveis globais poderá ocasionar inconsistências nos resultados da aplicação;
- Não há suporte às comunicações coletivas entre tarefas de um intra-comunicador e inter-comunicador;
- A proposta não inclui um gerenciamento de recursos em tempo de execução;
- As dependências entre tarefas não são consideradas no controle de granularidade;
- Não há garantias que a utilização da biblioteca em conjunto com outras *threads* da aplicação resultem em programas corretos.

4.7 Conclusão sobre o Capítulo

Este capítulo apresentou um controle de granularidade com tarefas para programas dinâmicos na interface padrão MPI. A biblioteca libSpawn implementa um controle de granularidade que considera os aspectos discutidos para a eficiência de algoritmos dinâmicos. Em síntese, esta implementação define uma tarefa como um fluxo de execução transparente e manipula a troca de mensagens entre tarefas. Uma primeira versão em C define uma granularidade estática por processo com um valor fixo em tempo de compilação. Mas a dependência de experimentos para um valor de granularidade ideal exigiu uma segunda proposta que considera parâmetros relacionados ao contexto de execução da aplicação. Além disso, a versão atual melhora a comunicação a fim de aumentar a concorrência das tarefas.

O próximo capítulo apresenta os experimentos realizados na avaliação deste trabalho. As aplicações desenvolvidas são baseadas em implementações anteriormente desenvolvidas para outros ambientes de programação para algoritmos dinâmicos. Além disso, a plataforma de execução consiste em um agregado de nós *multi-core* com a finalidade de avaliar o desempenho da biblioteca com paralelismo de dois níveis.

5 EXPERIMENTOS E RESULTADOS OBTIDOS

O capítulo anterior descreveu a proposta de uma biblioteca para o controle de granularidade em programas MPI dinâmicos chamada libSpawn. A biblioteca aumenta a localidade de tarefas e reduz os custos em comunicação e criação de tarefas. A fim de validar essa proposta, este capítulo apresenta os experimentos usados na avaliação da libSpawn. Esta avaliação objetiva determinar as vantagens de um controle de granularidade em relação ao uso de processos em aplicações MPI dinâmicas. A plataforma dos experimentos consiste em agregados de arquitetura *multi-core* que proporcionam o paralelismo de dois níveis: processos e *threads*.

As aplicações dinâmicas da avaliação consistem em *benchmarks* sintéticos anteriormente implementados em outros ambientes de programação. Seus algoritmos foram introduzidos na seção 2.3 e se referem aos seguintes aplicativos: *N-Queens*, Caixeiro Viajante, *Fibonacci* e *Mergesort*. A seguir, a seção 5.1 descreve a plataforma Grid'5000 onde os experimentos foram realizados. A seção 5.2 introduz o formato dos gráficos e tabelas com os tempos obtidos. Em seguida, a seção 5.3 apresenta os experimentos e resultados deste trabalho. Por fim, a seção 5.4 conclui este capítulo com as considerações finais a respeito dos resultados obtidos.

5.1 Plataforma de Execução dos Experimentos

Os experimentos foram realizados na plataforma francesa Grid'5000 (BOLZE et al., 2006) por intermédio do convênio entre o Grupo de Processamento Paralelo e Distribuído (GPPD) do Instituto de Informática (II) da Universidade Federal do Rio Grande do Sul (UFRGS) e o *Institut National de Recherche en Informatique et en Automatique* (INRIA) como equipe associada. Três diferentes *sites* da plataforma foram utilizados, um para cada aplicação. O número de nós usados em cada *site* é 30, que correspondente a 120 *cores* no total. Os *sites* e *clusters* usados são:

- **Toulouse** (*Fibonacci* e Caixeiro Viajante) - *cluster* Pastel, CPU AMD© Opteron 2218 2.6Hz, 2 CPUs por nó, 2 *cores* por CPU, total de 4 *cores* por nó, 8GB de memória, rede Gigabit Ethernet;
- **Sophia** (*N-Queens*) - *cluster* Sol, CPU AMD© Opteron 2218 2.6GHz, 2 CPUs por nó, 2 *cores* por CPU, total de 4 *cores* por nó, 4 GB de memória, rede Gigabit Ethernet;
- **Rennes** (*Mergesort*) - *cluster* Paraquad e Paramount, CPU Intel© Xeon 5148 LV 2.33Ghz, 2 CPUs por nó, 2 *cores* por CPU, total de 4 *cores* por nó, 8 GB de memória, rede Gigabit Ethernet.

5.2 Gráficos e Símbolos Utilizados

Os experimentos deste capítulo apresentam os resultados obtidos por meio de dois gráficos e uma tabela. Em cada experimento, o primeiro gráfico ilustra a média dos tempos obtidos com a biblioteca proposta e a distribuição de resultados na amostra de acordo com o número de *cores*. O segundo gráfico compara as médias obtidas com processos e com libSpawn nos experimentos. Por fim, a tabela complementa a compreensão dos resultados com o desvio padrão, erro e ganho da biblioteca em relação a execução com processos.

O primeiro gráfico mostra as médias das amostras com uma linha e a simetria dos valores obtidos com um *boxplot*, que sumariza quatro valores de uma amostra: o menor tempo, *quartile* inferior (Q1), mediana (Q2), *quartile* superior (Q3) e o maior tempo. Q1 representa 25% dos valores menores que Q2, enquanto Q3 consiste em 25% dos valores maiores que Q2. O intervalo entre Q1 e Q3, chamado *interquartile range* (IQR), é representado por uma caixa. Q2 aparece como uma linha horizontal dentro da caixa de IQR. As linhas verticais que partem das partes inferior e superior das caixas mostram os resultados maiores que $Q1 - 1,5 * IQR$ e menores que $Q3 + 1,5 * IQR$, respectivamente. Resultados das amostras menores ou maiores que estes valores são chamados *outliers*, que são resultados anômalos representados por um ponto $\circ$.

O segundo gráfico de cada experimento apresenta as médias obtidas do experimento de acordo com o número de *cores* utilizados. Ele objetiva demonstrar visualmente a diferença entre os tempos de execução com processos e com libSpawn. Da mesma forma, este gráfico mostra o desvio padrão da amostra através de linhas verticais.

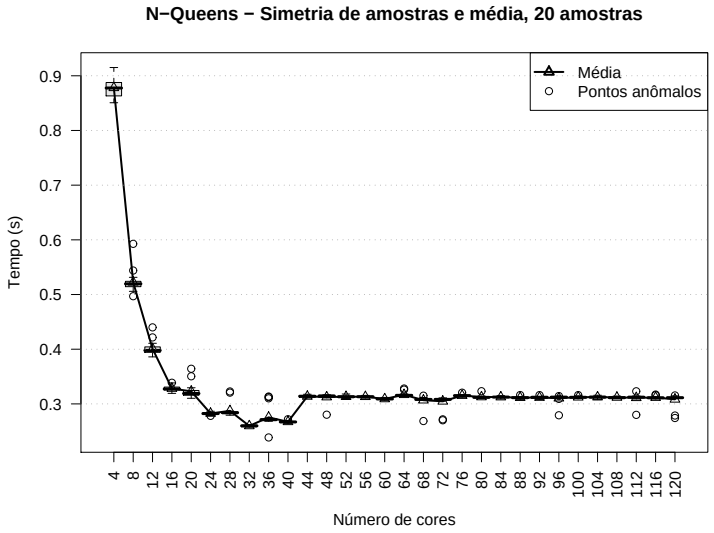
A tabela de resultados ilustra maiores detalhes a respeito dos resultados obtidos para algumas amostras. Ela compara numericamente os resultados com processos e com libSpawn através das médias e da razão entre elas. Os símbolos utilizados estão descritos na tabela 5.1. O erro das amostras segue a distribuição t de Student com $n - 1$ graus de liberdade a fim de se estimar o intervalo de confiança com um número pequeno de medições. Os experimentos usam vinte medições em cada amostra com um intervalo de confiança de 95%.

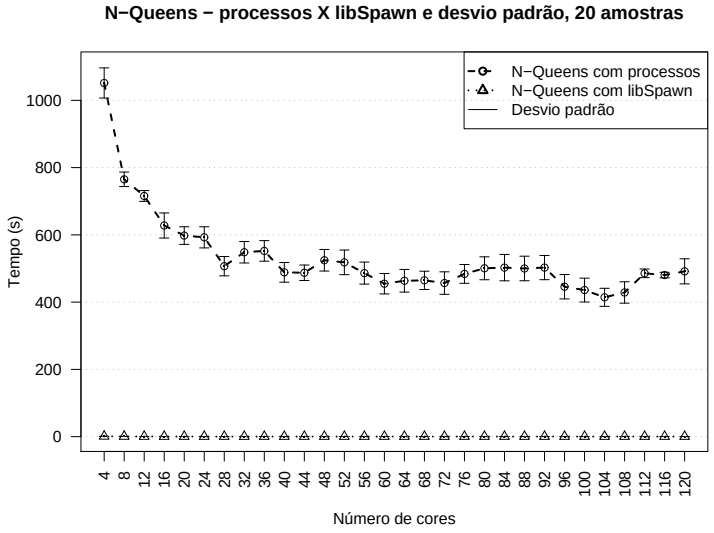
Símbolo	Descrição
$\overline{X}$	média da amostra
S	desvio padrão da amostra
α	nível de significância para determinar o nível de confiança da amostra
E	erro da amostra com tolerância $(1 - \alpha)$ conforme distribuição t de Student
Ganho	razão das médias $\overline{X}_{processos}/\overline{X}_{libSpawn}$

Tabela 5.1: Símbolos empregados na apresentação dos resultados obtidos

5.3 Descrição de Experimentos e Resultados

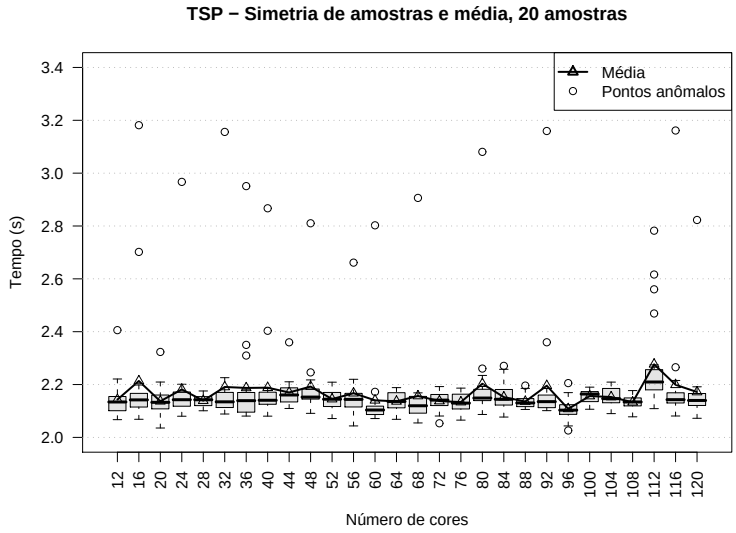
A seção anterior descreveu o significado dos gráficos e tabelas empregadas na apresentação dos resultados obtidos. Esta seção mostra os experimentos realizados na avaliação da biblioteca proposta através de quatro *benchmarks* sintéticos que avaliam a criação dinâmica de tarefas e comunicação. Os algoritmos destes aplicativos são descritos na seção 2.3 do capítulo 2. Os gráficos e tabelas de tempos obtidos estão entre as páginas 50 e 56.

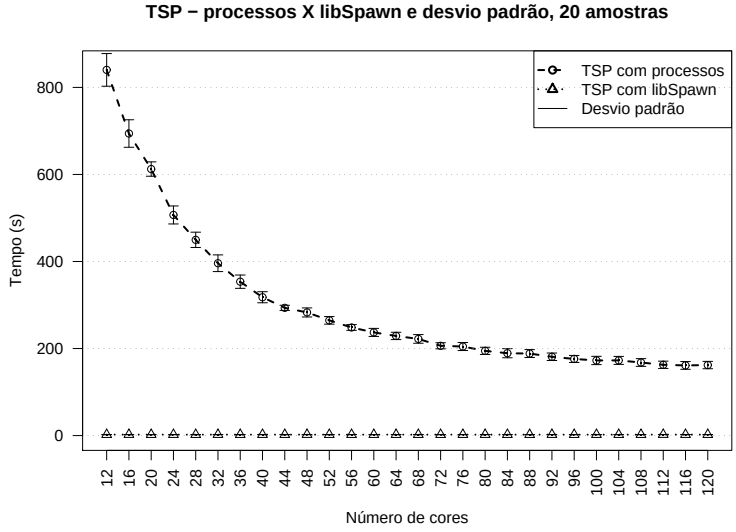




N-Queens **n=10, 32.979 tarefas, 20 amostras,** $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
4	$1.051,647$	$44,867$	$\pm 17,347$	$0,878$	$0,018$	$\pm 0,007$	$1.197,74$
16	$627,905$	$37,251$	$\pm 14,403$	$0,328$	$0,005$	$\pm 0,002$	$1.913,66$
32	$548,354$	$31,882$	$\pm 12,327$	$0,260$	$0,001$	$\pm 0,001$	$2.111,28$
48	$524,526$	$32,057$	$\pm 12,394$	$0,313$	$0,008$	$\pm 0,003$	$1.678,45$
64	$463,517$	$33,718$	$\pm 13,037$	$0,317$	$0,004$	$\pm 0,001$	$1.463,43$
80	$500,666$	$34,082$	$\pm 13,177$	$0,313$	$0,003$	$\pm 0,001$	$1.601,41$
96	$445,861$	$36,271$	$\pm 14,024$	$0,310$	$0,007$	$\pm 0,003$	$1.436,30$
112	$485,712$	$12,823$	$\pm 4,958$	$0,311$	$0,008$	$\pm 0,003$	$1.560,87$
120	$491,582$	$37,306$	$\pm 14,424$	$0,308$	$0,011$	$\pm 0,004$	$1.594,31$

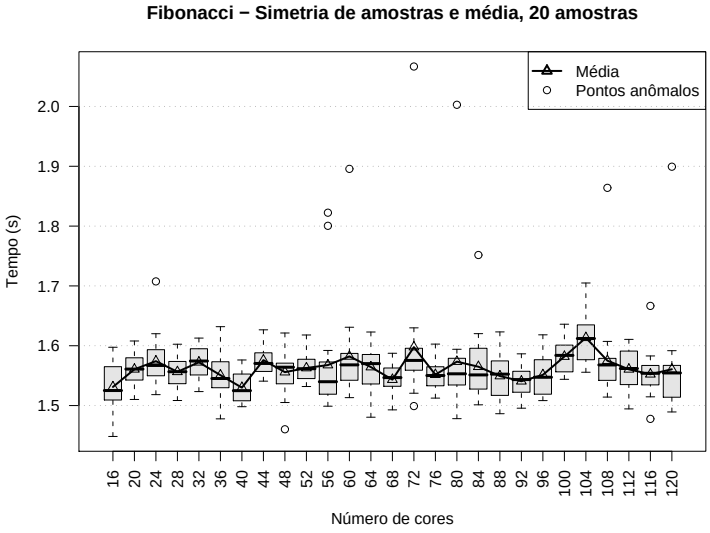
Figura 5.1: Tempos obtidos do *N-Queens* em execuções com processos e libSpawn que geram 32.979 tarefas

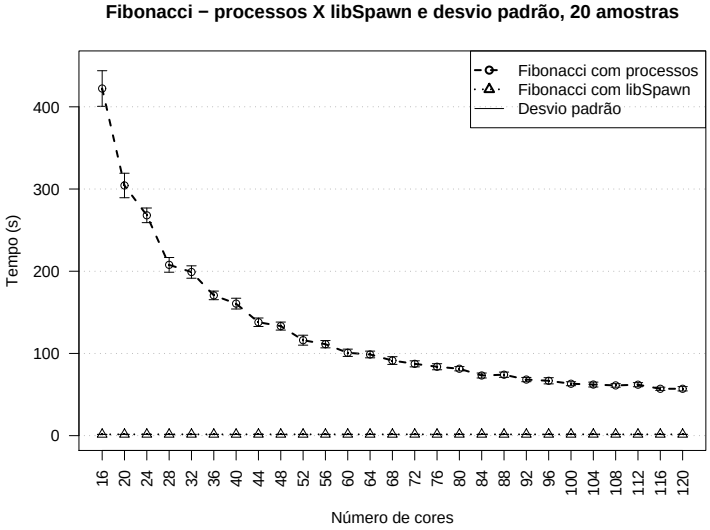




TSP n=10, 17.860 tarefas, 20 amostras, $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
12	$840,259$	$37,507$	$\pm 14,502$	$2,143$	$0,072$	$\pm 0,028$	$392,09$
16	$694,033$	$31,598$	$\pm 12,217$	$2,214$	$0,263$	$\pm 0,102$	$313,54$
32	$396,000$	$19,176$	$\pm 7,414$	$2,191$	$0,230$	$\pm 0,089$	$180,76$
48	$282,936$	$10,401$	$\pm 4,021$	$2,192$	$0,149$	$\pm 0,058$	$129,05$
64	$228,977$	$8,413$	$\pm 3,253$	$2,136$	$0,036$	$\pm 0,014$	$107,18$
80	$194,669$	$8,315$	$\pm 3,215$	$2,203$	$0,211$	$\pm 0,081$	$88,36$
96	$176,103$	$8,061$	$\pm 3,117$	$2,108$	$0,040$	$\pm 0,016$	$83,55$
112	$162,664$	$8,374$	$\pm 3,238$	$2,277$	$0,181$	$\pm 0,070$	$71,45$
120	$162,036$	$8,462$	$\pm 3,272$	$2,172$	$0,157$	$\pm 0,061$	$74,61$

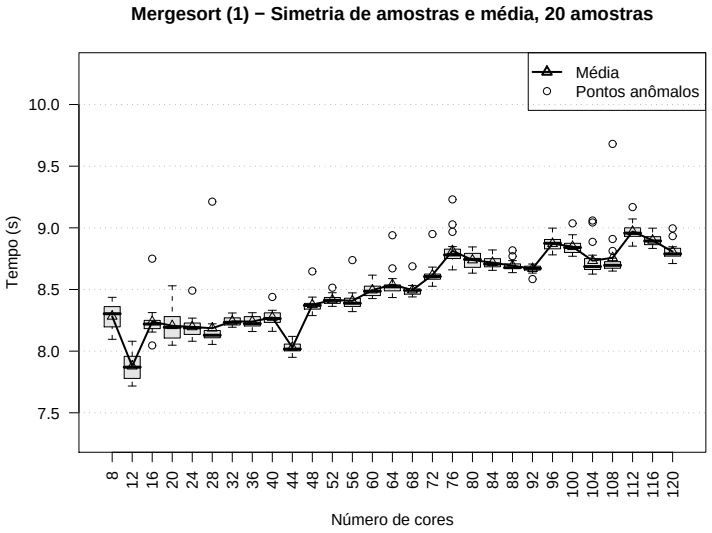
Figura 5.2: Tempos obtidos do TSP em execuções com processos e libSpawn que geram 17.860 tarefas

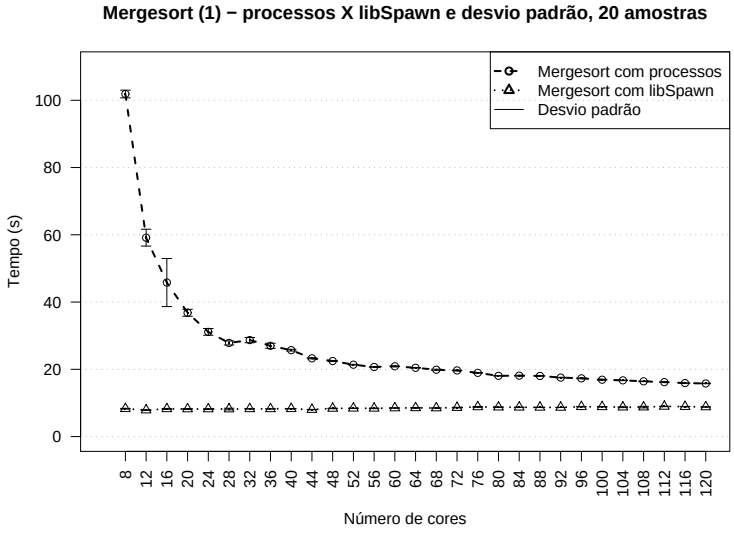




***Fibonacci* n=20, 13.530 tarefas, 20 amostras,** $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
16	$422,261$	$21,652$	$\pm 8,371$	$1,531$	$0,036$	$\pm 0,014$	$275,79$
32	$199,043$	$7,535$	$\pm 2,913$	$1,572$	$0,027$	$\pm 0,010$	$126,59$
48	$133,287$	$4,874$	$\pm 1,884$	$1,556$	$0,035$	$\pm 0,014$	$85,67$
64	$98,753$	$4,093$	$\pm 1,583$	$1,564$	$0,037$	$\pm 0,014$	$63,13$
80	$81,398$	$2,588$	$\pm 1,001$	$1,573$	$0,106$	$\pm 0,041$	$51,73$
96	$66,744$	$3,763$	$\pm 1,455$	$1,551$	$0,033$	$\pm 0,013$	$43,02$
112	$61,985$	$2,477$	$\pm 0,958$	$1,561$	$0,034$	$\pm 0,013$	$39,72$
120	$57,024$	$2,613$	$\pm 1,010$	$1,561$	$0,085$	$\pm 0,033$	$36,54$

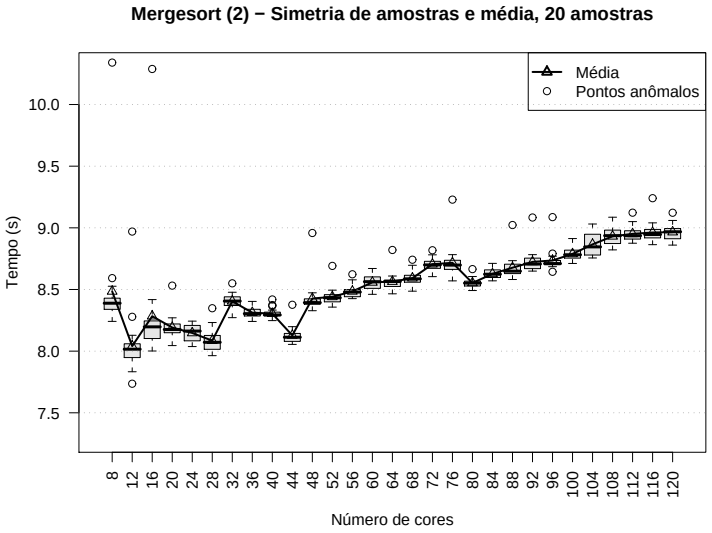
Figura 5.3: Tempos obtidos do *Fibonacci* em execuções com processos e libSpawn que geram 13.530 tarefas

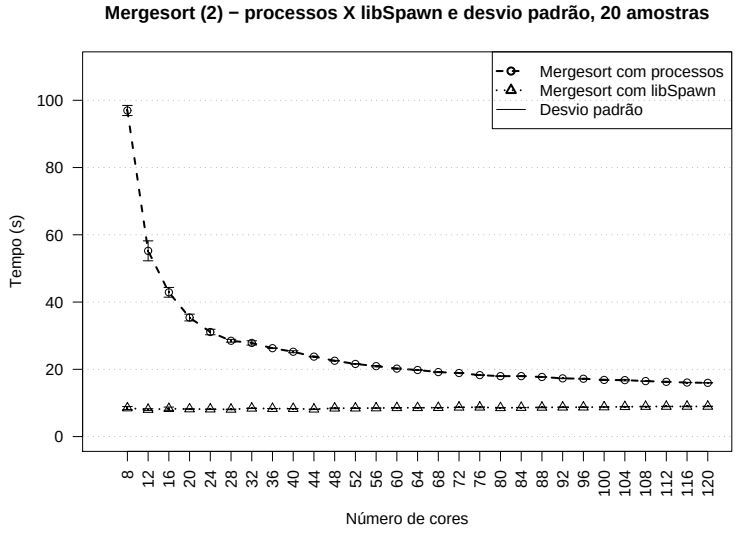




***Mergesort* n=2.000.000, 10.813 tarefas, 20 amostras,** $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
8	$101,861$	$1,145$	$\pm 0,443$	$8,282$	$0,102$	$\pm 0,039$	$12,30$
16	$45,809$	$7,149$	$\pm 2,764$	$8,239$	$0,133$	$\pm 0,051$	$5,56$
32	$28,692$	$0,781$	$\pm 0,302$	$8,242$	$0,037$	$\pm 0,014$	$3,48$
48	$22,470$	$0,158$	$\pm 0,061$	$8,376$	$0,073$	$\pm 0,028$	$2,68$
64	$20,447$	$0,102$	$\pm 0,039$	$8,539$	$0,109$	$\pm 0,042$	$2,39$
80	$18,063$	$0,080$	$\pm 0,031$	$8,739$	$0,069$	$\pm 0,027$	$2,07$
96	$17,319$	$0,130$	$\pm 0,050$	$8,872$	$0,055$	$\pm 0,021$	$1,95$
112	$16,189$	$0,089$	$\pm 0,034$	$8,970$	$0,069$	$\pm 0,027$	$1,80$
120	$15,795$	$0,080$	$\pm 0,031$	$8,807$	$0,064$	$\pm 0,025$	$1,79$

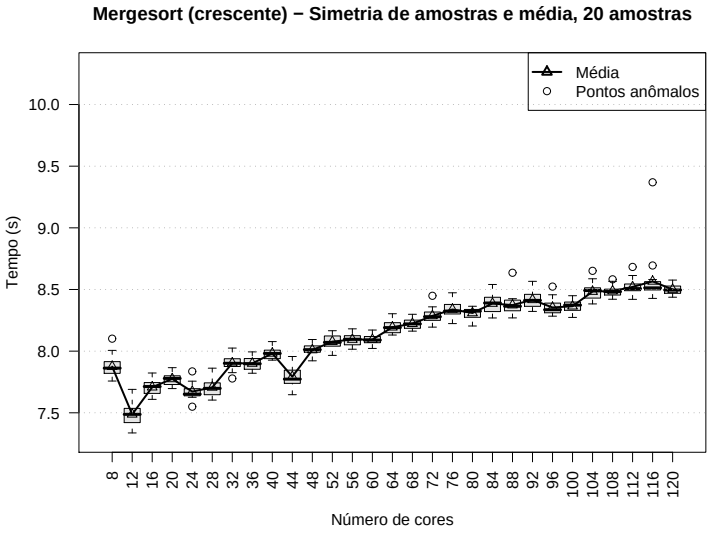
Figura 5.4: Tempos obtidos no *Mergesort* com números aleatórios (1) com processos e libSpawn que geram 10.813 tarefas

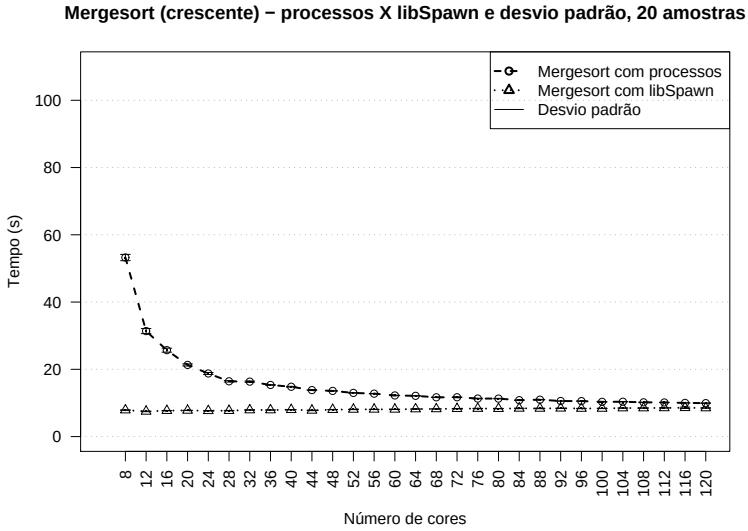




Mergesort **n=2.000.000, 10.813 tarefas, 20 amostras,** $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
8	$96,963$	$1,507$	$\pm 0,583$	$8,485$	$0,444$	$\pm 0,172$	$11,73$
16	$42,899$	$1,444$	$\pm 0,558$	$8,284$	$0,483$	$\pm 0,187$	$5,18$
32	$27,848$	$0,638$	$\pm 0,247$	$8,405$	$0,064$	$\pm 0,025$	$3,31$
48	$22,551$	$0,106$	$\pm 0,041$	$8,427$	$0,131$	$\pm 0,051$	$2,68$
64	$19,799$	$0,140$	$\pm 0,054$	$8,564$	$0,071$	$\pm 0,027$	$2,31$
80	$17,974$	$0,123$	$\pm 0,047$	$8,553$	$0,040$	$\pm 0,015$	$2,10$
96	$17,201$	$0,135$	$\pm 0,052$	$8,735$	$0,089$	$\pm 0,034$	$1,97$
112	$16,286$	$0,120$	$\pm 0,046$	$8,951$	$0,064$	$\pm 0,025$	$1,82$
120	$15,983$	$0,030$	$\pm 0,011$	$8,965$	$0,068$	$\pm 0,026$	$1,78$

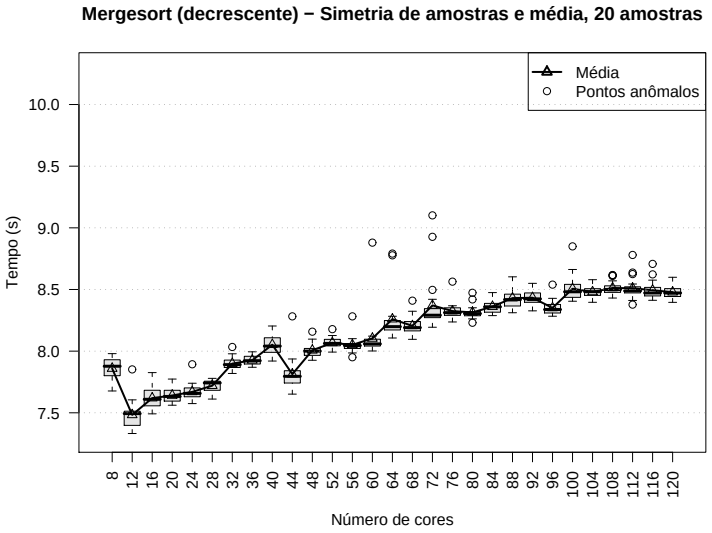
Figura 5.5: Tempos obtidos no *Mergesort* com números aleatórios (2) em execuções com processos e libSpawn que geram 10.813 tarefas

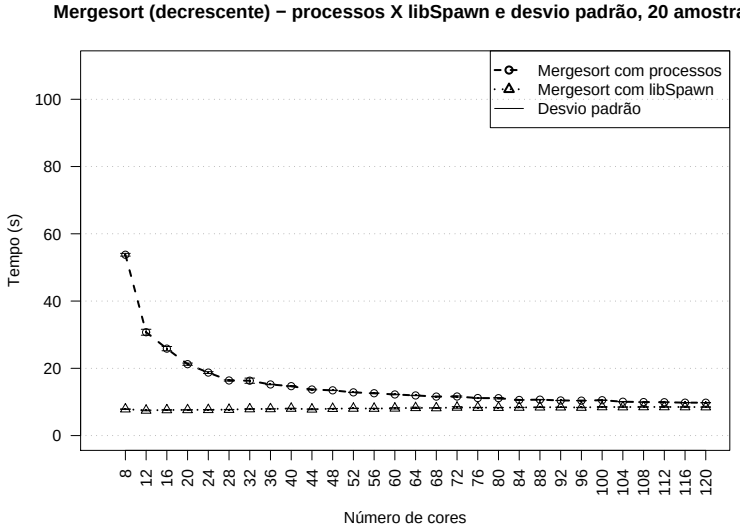




Mergesort **n=2.000.000, 5.419 tarefas, 20 amostras,** $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
8	$53,239$	$0,941$	$\pm 0,364$	$7,874$	$0,087$	$\pm 0,034$	$6,76$
16	$25,697$	$0,594$	$\pm 0,230$	$7,708$	$0,055$	$\pm 0,021$	$3,33$
32	$16,339$	$0,165$	$\pm 0,064$	$7,903$	$0,059$	$\pm 0,023$	$2,07$
48	$13,583$	$0,064$	$\pm 0,025$	$8,009$	$0,043$	$\pm 0,017$	$1,70$
64	$12,115$	$0,059$	$\pm 0,023$	$8,194$	$0,049$	$\pm 0,019$	$1,48$
80	$11,294$	$0,073$	$\pm 0,028$	$8,310$	$0,044$	$\pm 0,017$	$1,36$
96	$10,551$	$0,073$	$\pm 0,028$	$8,354$	$0,059$	$\pm 0,023$	$1,26$
112	$10,144$	$0,061$	$\pm 0,024$	$8,521$	$0,061$	$\pm 0,024$	$1,19$
120	$9,944$	$0,058$	$\pm 0,022$	$8,500$	$0,042$	$\pm 0,016$	$1,17$

Figura 5.6: Tempos obtidos no *Mergesort* com números em ordem crescente em execuções com processos e libSpawn que geram 5.419 tarefas





Mergesort **n=2.000.000, 5.419 tarefas, 20 amostras,** $\alpha = 0,95$							
	Processos			**libSpawn**			
cores	$\overline{X}$	S	E	$\overline{X}$	S	E	**Ganho**
8	$53,774$	$0,407$	$\pm 0,157$	$7,859$	$0,089$	$\pm 0,034$	$6,84$
16	$25,857$	$0,633$	$\pm 0,245$	$7,619$	$0,083$	$\pm 0,032$	$3,39$
32	$16,322$	$0,768$	$\pm 0,297$	$7,899$	$0,052$	$\pm 0,020$	$2,07$
48	$13,451$	$0,070$	$\pm 0,027$	$8,007$	$0,059$	$\pm 0,023$	$1,68$
64	$11,935$	$0,063$	$\pm 0,024$	$8,259$	$0,185$	$\pm 0,071$	$1,45$
80	$11,121$	$0,210$	$\pm 0,081$	$8,314$	$0,054$	$\pm 0,021$	$1,34$
96	$10,396$	$0,046$	$\pm 0,018$	$8,351$	$0,062$	$\pm 0,024$	$1,24$
112	$9,919$	$0,120$	$\pm 0,046$	$8,510$	$0,087$	$\pm 0,034$	$1,17$
120	$9,780$	$0,055$	$\pm 0,021$	$8,478$	$0,052$	$\pm 0,020$	$1,15$

Figura 5.7: Tempos obtidos no *Mergesort* com números em ordem decrescente entre execuções com processos e libSpawn que geram 5.419 tarefas

5.3.1 *N-Queens*

A implementação do *N-Queens* é baseada no algoritmo descrito na seção 2.3.1 e se caracteriza por uma granularidade de grão fino e poucas comunicações. Por tarefa, as operações comunicantes consistem no envio de um tabuleiro n com as rainhas posicionadas até o momento e no retorno do número de jogadas possíveis. Os experimentos usam um tabuleiro 10×10 que resulta em 32.979 tarefas. Os gráficos e tabela de resultados são mostrados na figura 5.1 da página 50.

No primeiro gráfico, os tempos com libSpawn mostram um ganho considerável entre 4 e 16 *cores*. Porém, os tempos entre 16 e 44 *cores* apresentam irregularidades. O segundo gráfico não demonstra as tendências do gráfico anterior devido a sua escala. Por outro lado, os tempos com libSpawn em relação aos tempos com processos são significadamente menores. Além disso, os tempos com processos são irregulares a partir de 24 *cores*. A tabela de resultados demostra claramente as irregularidades tanto com processos quanto com libSpawn nos valores apresentados do ganho obtido.

5.3.2 Caixeiro Viajante

O caixeiro viajante ou TSP tem sua implementação baseada no algoritmo descrito na seção 2.3.3 e caracteriza-se pelos períodos de comunicação maiores. As comunicações se resumem no envio dos caminhos mínimos entre cidades, da posição e do caminho mínimo encontrado no espaço de busca atual. Estes experimentos usam 10 cidades que resultam em 17.860 tarefas. O limite de recursividade, que determina o início da busca sequencial, inicia a partir de 6 cidades. Os gráficos e tabela de resultados estão na figura 5.2 da página 51.

O primeiro gráfico com os resultados da libSpawn mostra médias relativamente regulares que se mantém entre 2 e $2,3$ segundos. Entretanto, algumas amostras apresentam vários pontos anômalos como em 36 e 112 *cores*. Em seguida, o segundo gráfico demonstra que resultados com processos e libSpawn diferem significadamente. Este gráfico e a tabela de resultados apresentam tempos regulares com processos e confirma os ganhos obtidos com a libSpawn.

5.3.3 *Fibonacci*

A sequência de *Fibonacci* tem sua implementação baseada no exemplo demostrado na seção 2.3.4 e se caracteriza pelo grão fino de tarefas. Sua comunicação está relacionada ao envio das entradas e retorno de resultados. A entrada dos experimentos deste trabalho consiste no cálculo da sequência *Fibonacci* para o número 20 que resulta em 13.530 tarefas. Os gráficos e tabela de resultados se encontram na figura 5.3 da página 52.

O primeiro gráfico com a libSpawn demostra que as médias variam entre $1,5$ e $1,6$ segundos, mas com pontos anômalos nas amostras. O segundo gráfico apresenta uma diferença significativa entre o uso de processos e libSpawn que diminui à medida que o número de *cores* aumenta. A tabela de resultados confirma esta tendência e mostra que os tempos com libSpawn mantêm-se em aproximadamente $1,5$ segundos.

5.3.4 *Mergesort*

A implementação da ordenação por *Mergesort* é baseada no algoritmo descrito na seção 2.3.2 e caracteriza-se por uma granularidade de grão grosso, apesar das longas comunicações nos primeiros níveis da recursão. A comunicação entre tarefas consiste no envio e recebimento de $n/4$ e $n/2$ elementos nas fases de *sort* e *merge*, respectivamente.

Os experimentos com *Mergesort* usam quatro entradas com 2 milhões de elementos em diferentes ordens: dois em ordem aleatória, um em ordem crescente e um em ordem decrescente. As entradas aleatórias resultam em 10.813 tarefas, ao passo que as entradas em ordem crescente e decrescente resultam em 5.419 tarefas. Os gráficos e tabelas de resultados são mostrados nas figuras: 5.4 da página 53 para a primeira entrada aleatória, 5.5 da página 54 para a segunda entrada aleatória, 5.6 da página 55 para a entrada em ordem crescente e 5.7 da página 56 para a entrada em ordem decrescente.

Os gráficos de tempos com a libSpawn apresentam uma tendência semelhante nas quatro entradas: o aumento no número de *cores* eleva o tempo de execução em pouco menos de 1 segundo em um intervalo de 8 a 120 *cores*. Estes gráficos também mostram tempos anômalos nas amostras, principalmente nas entradas de ordem aleatória e decrescente. Em seguida, os gráficos de tempos com processos e libSpawn demostram que o aumento no número de *cores* diminui o tempo de execução com processos. Dessa forma, o ganho obtido diminui e se aproxima do valor 1.

5.4 Conclusão sobre o Capítulo

Os *benchmarks* sintéticos apresentados avaliam aspectos essenciais de um ambiente de programação tais como: criação de tarefas, localidade e comunicação entre tarefas. Os resultados obtidos demonstram vantagens significativas da biblioteca libSpawn, porém as amostras indicam carências na implementação proposta.

As quatro aplicações demostram ganhos significativos com o controle de granularidade proposto. As execuções com poucos *cores* resultam nos maiores ganhos e o aumento do número de *cores* mantém ganhos significativos. Em um primeiro momento, a redução nos tempo de execução é atribuída à redução do custo em criação de tarefas. A redução em criação de tarefas localmente e como *threads* melhora o desempenho da aplicação. Por conseguinte, estas tarefas criadas localmente realizam comunicações em memória compartilhada com custos menores.

Porém, a ordenação por *Mergesort* demonstra ganhos menores que os demais aplicativos e próximos a 1 para um número maior de *cores*. Este ganho reduzido se deve ao fato do número de fases na ordenação estar separado em três programas. De acordo com o controle de granularidade da libSpawn, um programa MPI-2 que executa uma chamada `Spawn` para um executável diferente do programa atual cria uma tarefa como processo. Dessa forma, o *Mergesort* cria um número maior de processos e reduz os ganhos no aumento da localidade.

Além disso, os gráficos com tempos da libSpawn mostram resultados anômalos em todos os experimentos realizados. O motivo dessas anomalias se deve à falta de escalonamento em tarefas criadas localmente, tanto processos quanto *threads*. Alguns dos ambientes de programação mostrados no capítulo 2, por exemplo Cilk em arquiteturas compartilhadas e KAAPI em distribuídas, proporcionam dinamismo e incorporam um escalonador de tarefas teoricamente eficiente e com resultados previsíveis. O escalonamento de tarefas não é tratado neste trabalho devido ao foco no controle de granularidade.

Mesmo com estes resultados anômalos, o pior tempo de execução em cada amostra da libSpawn não supera o melhor tempo das amostras com processos. As amostras das aplicações *N-Queens*, *Fibonacci* e TSP mostram uma diferença significativa entre o pior tempo da libSpawn e o melhor tempo com processos. Portanto, conclui-se que o controle de granularidade deste trabalho propicia ganhos significativos.

6 CONCLUSÃO

Este trabalho tratou de controlar a granularidade de programas MPI dinâmicos ao mapear as tarefas em *threads* ou processos de acordo com três parâmetros em tempo de execução: *cores* da arquitetura, carga e recursos de sistema. Esta proposta se concretiza na biblioteca libSpawn que controla a granularidade e a comunicação entre tarefas locais. O controle implementado aumenta a localidade de tarefas criadas em tempo de execução ao avaliar os parâmetros de arquitetura de execução, carga de sistema e utilização de recursos do sistema. Dessa forma, a libSpawn possibilita reduzir custos em criação e comunicação de tarefas. Os resultados obtidos com *benchmarks* sintéticos permitiram constatar que o controle de granularidade pode oferecer uma redução significativa em criação e comunicação de tarefas. Este trabalho também constatou tempos anômalos devido à falta de um escalonamento de tarefas sensível aos níveis de paralelismo entre processos e *threads*.

6.1 Contribuições

A principal contribuição foi confirmar a redução significativa de custos em criação e comunicação de tarefas por meio de um controle de granularidade no padrão MPI. As outras contribuições a partir desta dissertação foram:

- Apresentar características relacionadas à programação de algoritmos dinâmicos e exemplos;
- Constatar que implementações MPI dificilmente suportam dinamismo e programação *multi-thread*;
- Incorporar os conceitos de tarefa e granularidade a programas MPI dinâmicos;
- Desenvolver comunicações MPI eficientes e livres de conflitos entre *threads*;
- Avaliar a biblioteca implementada frente a quatro *benchmarks* sintéticos comumente empregados na avaliação de ambientes de programação com suporte a dinamismo;
- Disponibilizar ao grupo de pesquisa uma nova abordagem a respeito de programas MPI dinâmicos através de tarefas.

Um artigo e três resumos foram escritos para eventos nacionais e regionais durante o desenvolvimento deste trabalho. Foi submetido e aprovado um artigo para o evento

nacional da área WSCAD 2008 (LIMA; MAILLARD, 2008a). Além desse evento, resumos foram publicados nos eventos regionais ERAD 2008 (LIMA; MAILLARD, 2008b), WSPPD 2008 (LIMA; MAILLARD, 2008c) e ERAD 2009 (LIMA; MAILLARD, 2009).

6.2 Trabalhos Futuros

Os trabalhos futuros preveem a continuidade do trabalho desenvolvido a partir de novas funcionalidades e redução de suas limitações. No grupo GPPD, a biblioteca será o ponto de partida para a inclusão de um escalonamento em dois níveis teoricamente eficiente entre nós e *cores* de um agregado. Dessa forma, o mapeamento será de um processo por nó e uma *thread* por *core*. Este trabalho possibilitou determinar a granularidade de tarefas e um escalonamento em dois níveis permitirá um balanceamento de carga eficiente e previsível.

Não obstante, pretende-se ampliar as avaliações de desempenho. O desenvolvimento de *benchmarks* de algoritmos dinâmicos está previsto dentro de alguns trabalhos do grupo GPPD. Da mesma forma, espera-se que novos experimentos sejam realizados em outras plataformas tais como *cluster of clusters* e grades.

REFERÊNCIAS

BALAJI, P.; BUNTINAS, D.; GOODWELL, D.; GROPP, W.; THAKUR, R. Toward Efficient Support for Multithreaded MPI Communication. In: THE 15TH EUROPEAN PVM/MPI USERS' GROUP CONFERENCE, EUROPVM/MPI 2008, 2008, Dublin, IRL. **Proceedings...** Springer-Verlag, 2008. p.120–129.

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. In: THE FIFTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPOPP'95, 1995, Santa Barbara, USA. **Proceedings...** ACM Press, 1995. p.207–216.

BLUMOFE, R. D.; LEISERSON, C. E. Space-Efficient Scheduling of Multithreaded Computations. **SIAM Journal on Computing**, Philadelphia, USA, v.27, 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, 1997, Anaheim, USA. **Proceedings...** USENIX Association, 1997. p.133–147.

BOLZE, R. et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. **International Journal of High Performance Computing Applications**, Thousand Oaks, USA, v.20, n.4, p.481–494, 2006.

BUNTINAS, D.; MERCIER, G.; GROPP, W. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. **Parallel Computing**, Amsterdam, NLD, v.33, n.9, p.634–644, September 2007.

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: THE 13H EUROPEAN PVM/MPI USERS GROUP MEETING, EUROPVM/MPI 2006, 2006, Bonn, DEU. **Proceedings...** Springer Berlin / Heidelberg, 2006. p.247–255. (Lecture Notes in Computer Science, v.4192/2006).

CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using OpenMP**: portable shared memory parallel programming. Cambridge, USA: The MIT Press, 2007.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos**: teoria e prática. [S.l.]: Editora Campus, 2002.

DANJEAN, V. et al. Adaptive Loops with Kaapi on Multicore and Grid: Applications in Symmetric Cryptography. In: THE 2007 INTERNATIONAL WORKSHOP ON PARALLEL SYMBOLIC COMPUTATION, PASCO'07, 2007, New York, USA. **Proceedings...** ACM, 2007. p.33–42.

DEMAINE, E. D. A Threads-Only MPI Implementation for the Development of Parallel Programs. In: THE 11TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTING SYSTEMS, HPCS'97, 1997, Winnipeg, CAN. **Proceedings...** ACM Press, 1997. p.153–163.

DONGARRA, J.; FOSTER, I.; FOX, G.; GROPP, W.; KENNEDY, K.; TORCZON, L.; WHITE, A. **Sourcebook of Parallel Computing**. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2003.

DREPPER, U. **ELF Handling For Thread-Local Storage**. Disponível em: <http://people.redhat.com/drepper/tls.pdf>. Acesso em: fev. 2009.

FORUM, M. P. I. **MPI: a message-passing interface standard**. Knoxville, USA: University of Tennessee, 1994. (CS-94-230).

FORUM, M. P. I. **MPI-2**: Extensions to the Message-Passing Interface. Knoxville, USA: University of Tennessee, 1997. (CDA-9115428).

FOSTER, I. **Designing and Building Paralllel Programs**. [S.l.]: Addison-Wesley, 1995. Disponível em: <http://www.mcs.anl.gov/dbpp>. Acesso em: fev. 2009.

FOSTER, I.; KESSELMAN, C. Computational Grids. **The Grid: Blueprint for a New Computing Infrastructure**, San Francisco, USA, p.15–51, 1999.

FOUNDATION, F. S. **Using the GNU Compiler Collection (GCC)**. [S.l.]: Free Software Foundation, 2008. Disponível em: <http://gcc.gnu.org/onlinedocs>. Acesso em: fev. 2009.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. **ACM SIGPLAN Notices**, New York, USA, v.33, n.5, p.212–223, May 1998.

GABRIEL, E. et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: THE 13H EUROPEAN PVM/MPI USERS GROUP MEETING, EUROPVM/MPI 2004, 2004, Bonn, DEU. **Proceedings...** Springer Berlin / Heidelberg, 2004. p.97–104. (Lecture Notes in Computer Science, v.3241/2004).

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: THE 2007 INTERNATIONAL WORKSHOP ON PARALLEL SYMBOLIC COMPUTATION, PASCO'07, 2007, London, CAN. **Proceedings...** ACM, 2007.

GROPP, W. MPICH2: A New Start for MPI Implementations. In: THE 9TH EUROPEAN PVM/MPI USERS GROUP MEETING, EUROPVM/MPI 2002, 2002, Linz, AUT. **Proceedings...** Springer Berlin / Heidelberg, 2002. p.37–42. (Lecture Notes in Computer Science, v.2474/2002).

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

GROPP, W.; THAKUR, R. Thread-safety in an MPI implementation: Requirements and analysis. **Parallel Computing**, Amsterdam, NLD, v.33, n.9, p.595–604, Sept. 2007.

HUANG, C. et al. Performance Evaluation of Adaptive MPI. In: THE ELEVENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPOPP'06, 2006, New York, USA. **Proceedings...** ACM, 2006. p.12–21.

HUANG, C.; LAWLOR, O.; KALé, L. V. Adaptive MPI. In: THE 16TH INTERNATION WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING, LCPC 2003, 2004, College Station, USA. **Proceedings...** Springer Berlin / Heidelberg, 2004. p.306–322. (Lecture Notes in Computer Science, v.2958/2004).

JÁJÁ, J. **An Introduction to Parallel Algorithms**. Redwood City, USA: Addison Wesley Longman Publishing Co., 1992.

KWOK, Y.-K.; AHMAD, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. In: ACM COMPUTING SURVEYS, 1999, New York, USA. **Proceedings...** ACM, 1999. v.31, n.4, p.406–471.

LEOPOLD, C.; SÜSS, M. Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments. In: THE 13TH EUROPEAN PVM/MPI USER'S GROUP MEETING, EUROPVM/MPI 2006, 2006, Bonn, DEU. **Proceedings...** Springer, 2006. p.285–292. (Lecture Notes in Computer Science, v.4192).

LEOPOLD, C.; Süß, M.; BREITBART, J. Programming for Malleability with Hybrid MPI-2 and OpenMP - Experiences with a Simulation Program for Global Water Prognosis. In: THE 20TH EUROPEAN CONFERENCE ON MODELLING AND SIMULATION, ECMS 2006, 2006, Bonn, DEU. **Proceedings...** [S.l.: s.n.], 2006. p.665–670.

LIMA, J. V. F.; MAILLARD, N. Controle de Granularidade com threads em Programas MPI Dinâmicos. In: IX SIMPóSIO EM SISTEMAS COMPUTACIONAIS, WSCAD-SSC 2008, 2008, Campo Grande, BRA. **Anais...** SBC, 2008. p.117–124.

LIMA, J. V. F.; MAILLARD, N. Aplicações Dinâmicas MPI-2 com threads. In: VIII ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD 2008, 2008, Santa Cruz do Sul, BRA. **Anais...** SBC, 2008. p.131–132.

LIMA, J. V. F.; MAILLARD, N. Controlling Task Granularity of Dynamic MPI Programs with threads. In: VI WORKSHOP DE PROCESSAMENTO PARALELO E DISTRIBUíDO, WSPPD 2008, 2008, Porto Alegre, BRA. **Anais...** [S.l.: s.n.], 2008. Disponível em: <http://gppd.inf.ufrgs.br/wsppd/2008/papers/Lima.pdf>.

LIMA, J. V. F.; MAILLARD, N. Controle de Granularidade com threads em Programas MPI Dinâmicos. In: IX ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD 2009, 2009, Caxias do Sul, BRA. **Anais...** SBC, 2009. p.111–112.

MPICH. **MPICH2**: high-performance and widely portable mpi. Disponível em: <http://www.mcs.anl.gov/mpi/mpich2>. Acesso em: fev. 2009.

NAVAUX, P. O. A.; ROSE, C. A. F. D. **Arquiteturas Paralelas**. 1.ed. [S.l.]: Editora Sagra Luzzato, 2003. n.15. (Série Livros Didáticos).

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Satin: efficient parallel divide-and-conquer in java. In: THE 6TH INTERNATIONAL EURO-PAR CONFERENCE, 2000, Munich, DEU. **Proceedings...** Springer, 2000. p.690–699. (Lecture Notes in Computer Science, v.1900).

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Efficient load balancing for wide-area divide-and-conquer applications. In: THE EIGHTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICES OF PARALLEL PROGRAMMING, PPOPP'01, 2001, New York, USA. **Proceedings...** ACM, 2001. p.34–43.

PEZZI, G. P.; CERA, M. C.; MATHIAS, E.; MAILLARD, N.; NAVAUX, P. O. A. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. In: THE 19TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD 2007, 2007, Gramado, BRA. **Proceedings...** SBC, 2007. p.247–254.

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: VII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD'06, 2006, Ouro Preto, BRA. **Anais...** SBC, 2006. v.7, p.71–78.

REINDERS, J. **Intel Threading Building Blocks**: outfitting c++ for multi-core processor parallelism. Sebastopol, USA: O'Reilly & Associates, Inc., 2007.

RÜNGER, G. Parallel Programming Models for Irregular Algorithms. In: HOFFMAN, K. H.; MEYER, A. (Ed.). **Parallel Algorithms and Cluster Computing - Implementations, Algorithms and Applications**. [S.l.]: Springer, 2006. p.3–23. (Lecture Notes in Computational Science and Engineering, v.52).

SOARES, L. et al. Parallel Adaptive Octree Carving for Real-time 3D Modeling. In: IEEE VIRTUAL REALITY CONFERENCE, 2007, Charlotte, USA. **Proceedings...** IEEE Computer Society Press, 2007.

SQUYRES, J. M.; LUMSDAINE, A. A Component Architecture for LAM/MPI. In: THE 10TH EUROPEAN PVM/MPI USERS' GROUP MEETING, 2003, Venice, ITA. **Proceedings...** Springer-Verlag, 2003. n.2840, p.379–387. (Lecture Notes in Computer Science).

STERLING, T. et al. Beowulf: a parallel workstation for scientific computation. In: THE 24TH INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP 1995, 1995, Urbana-Champain, USA. **Proceedings...** [S.l.: s.n.], 1995. v.1, p.11–14.

TANG, H.; YANG, T. Optimizing Threaded MPI Execution on SMP Clusters. In: THE 15TH INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ICS'01, 2001, Sorrento, ITA. **Proceedings...** ACM Press, 2001. p.381–392.

THAKUR, R.; GROPP, W. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In: THE 14H EUROPEAN PVM/MPI USERS GROUP MEETING, EUROPVM/MPI 2007, 2007, Paris, FRA. **Proceedings...**

Springer Berlin / Heidelberg, 2007. p.46–55. (Lecture Notes in Computer Science, v.4757).

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

WRZESINSKA, G.; MAASSEN, J.; BAL, H. E. Self-adaptive Applications on the Grid. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPOPP'07, 2007, New York, USA. **Proceedings...** ACM, 2007. p.121–129.